GRAPHCORE

PopART Python API Reference

Version latest

Graphcore Ltd

Oct 21, 2021

CONTENTS

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.

This document describes the PopART Python API. Many classes are wrappers around the equivalent C++ class, for example `popart.builder.Builder` wraps the C++ `Builder` class (renamed `BuilderCore` in Python). There are more detailed descriptions of some functions in the PopART C++ API.

For more information about PopART, please refer to the PopART User Guide.

CHAPTER

TWO

POPART PYTHON API

2.1 Sessions

2.1.1 Training session

class popart.**TrainingSession**(*fnModel*, *dataFlow*, *loss*, *optimizer*, *deviceInfo*, *inputShapeInfo=<popart_core.InputShapeInfo object>*, *patterns=None*, *userOptions=<popart_core.SessionOptions object>*, *name='training'*)

Create a runtime class for executing an ONNX graph on a set of IPU hardware for training

A wrapper around the `Session` C++ class, renamed `SessionCore` in pybind, to enable more Pythonic use. See `session.hpp` for parameter descriptions.

Parameters

- **fnModel** – ONNX model proto. Usually a loaded ONNX model, or from `builder.getModelProto()`.
- **dataFlow** – Configuration for the data feeds and fetches.
- **loss** – A TensorId of the final scalar loss to use when training.
- **optimizer** – The type of optimizer to use when training and it's properties.
- **deviceInfo** – DeviceInfo object specifying device type (IPU, IPUModel, CPU) and count.
- **inputShapeInfo** – Information about the shapes of input and output tensors. Default: `popart.InputShapeInfo()`.
- **patterns** – Optimization patterns to apply. Default: `None`.
- **userOptions** – Session options to apply. Default: `popart.SessionOptions()`.
- **name** – Session name used in debug to identify this session Default: `training`

property accumulationFactor

compileAndExport(*filename*)

Compiles the graph and exports it to the specified file.

This will form the snap::Graph and compile the polar::Executable before exporting the executable and metadata.

Parameters

- **filename** – Where to save the executable and metadata. If
- **does not exist** (`it`) –
- **will be created.** (`it`) –

Raises

- **popart.OutOfMemoryException** – If an out of memory event occurs

- **OSError** – Thrown in the event of any file system related errors during the export

Return type None

property dataFlow

initAnchorArrays()
Create the anchor arrays to feed data back into Python with.

Returns Dict of anchor names and their relevant np arrays.

Return type Dict[str, numpy.array]

prepareDevice(*loadEngine=True*)
Prepare the network for execution.

This will create the `snap::Graph` and `poplar::Engine`, and set up `poplar::Streams`.

Parameters **loadEngine** (`bool`) – Load the engine and connect the streams once the device is ready.

Raises **popart.OutOfMemoryException** – If an out of memory event occurs

Return type None

property replicationFactor

2.1.2 Inference session

class popart.**InferenceSession**(*fnModel*, *dataFlow*, *deviceInfo*, *inputShapeInfo=<popart_core.InputShapeInfo object>*, *patterns=None*, *userOptions=<popart_core.SessionOptions object>*, *name='inference'*)
Create a runtime class for executing an ONNX graph on a set of IPU hardware for inference.

A wrapper around the `Session` C++ class, renamed `SessionCore` in pybind, to enable more Pythonic use. See `session.hpp` for parameter descriptions.

Parameters

- **fnModel** – ONNX model proto. Usually a loaded ONNX model, or from `builder.getModelProto()`.
- **dataFlow** – Configuration for the data feeds and fetches.
- **deviceInfo** – `DeviceInfo` object specifying device type. (one of `IPU`, `IPUModel` or `CPU`) and count.
- **inputShapeInfo** – Information about the shapes of input and output tensors. Default: `popart.InputShapeInfo()`.
- **patterns** – Patterns to be run for optimization etc. Note: default for patterns must not be `popart.Patterns()`. When `import popart` is run, the default arguments are created. If the user then loads a custom pattern using `ctypes.cdll.LoadLibrary(custom_pattern_lib.so)` then the already constructed `popart.Patterns` will not include the custom pattern. Default `None`.
- **userOptions** – Session options to apply. Default: `popart.SessionOptions()`.
- **name** – Session name used in debug to identify this session Default: `inference`

property accumulationFactor

compileAndExport(*filename*)
Compiles the graph and exports it to the specified file.

This will form the snap::Graph and compile the polar::Executable before exporting the executable and metadata.

Parameters

- **filename** – Where to save the executable and metadata. If
- **does not exist** (it) –
- **will be created.** (it) –

Raises

- **popart.OutOfMemoryException** – If an out of memory event occurs
- **OSError** – Thrown in the event of any file system related errors during the export

Return type None

property dataFlow

classmethod fromIr(*ir*, *deviceInfo*, *name='fromIr'*)

Parameters

- **deviceInfo** (popart_core.DeviceInfo) –
- **name** (str) –

Return type popart.session.InferenceSession

initAnchorArrays()

Create the anchor arrays to feed data back into Python with.

Returns Dict of anchor names and their relevant np arrays.

Return type Dict[str, numpy.array]

prepareDevice(*loadEngine=True*)

Prepare the network for execution.

This will create the `snap::Graph` and `poplar::Engine`, and set up `poplar::Streams`.

Parameters **loadEngine** (bool) – Load the engine and connect the streams once the device is ready.

Raises **popart.OutOfMemoryException** – If an out of memory event occurs

Return type None

property replicationFactor

2.1.3 Data input and output

Note: The base class for IO in PopART is `IStepIO`. The way in which this class is used is detailed in our C++ API documentation.

class popart.**PyStepIO**

This class is an implementation of the IStepIO interface backed by user-provided dictionaries for both input and output. These dictionaries map TensorId values to numpy arrays for PopART to read from and write to, respectively.

__init__(*self: popart_core.PyStepIO*, *inputs: Dict[str, numpy.ndarray]*, *outputs: Dict[str, numpy.ndarray]*) → None

Construct a new PyStepIO instance.

Args:

inputs: A dictionary with an entry for every input tensor, comprising a TensorId for the `key` and a numpy array for a `value` for PopART to read from. The numpy arrays are assumed to be size-compatible with a tensor of shape [`replicationFactor`, `accumulationFactor`, `batchesPerStep`, `<tensor shape>`].

outputs: A dictionary with an entry for every output tensor, comprising a TensorId for the `key` and a numpy array `value` to which PopART will write the associated data. The expected shape of this numpy array is explained in the C++ API documentation for popart::AnchorReturnTypeId. The convenience method Session.initAnchorArrays() is typically used to create a dictionary with suitable arrays.

enableRuntimeAsserts(*self: popart_core.PyStepIO*, *arg0: bool*) → None

Enable (or disable) run-time checks that check the sizes of the provided numpy arrays.

Args:

arg0: Flag to enable/disable checks

class popart.**PyStepIOCallback**

This class is an implementation of the IStepIO interface backed by user-provided callback functions. This class inherits from IStepIO and implements those member functions by delegating the logic to the callback functions passed in the constructor. This gives the user full control as to how data buffers are provisioned."

__init__(*self: popart_core.PyStepIOCallback*, *input_callback: Callable[[str, bool], numpy.ndarray]*, *input_complete_callback: Callable[[str], None]*, *output_callback: Callable[[str], numpy.ndarray]*, *output_complete_callback: Callable[[str], None]*) → None

Construct a new PyStepIOCallback instance.

Args:

input_callback: Callable object that the PyStepIOCallback instance will use when `IStepIO::in()` is called. See IStepIO for details on how to implement this method.

input_complete_callback: Callable object that the PyStepIOCallback instance will use when `IStepIO::inComplete()` is called. See IStepIO for details on how to implement this method.

output_callback: Callable object that the PyStepIOCallback instance will use when `IStepIO::out()` is called. See IStepIO for details on how to implement this method.

output_complete_callback: Callable object that the PyStepIOCallback instance will use when `IStepIO::outComplete()` is called. See IStepIO for details on how to implement this method.

2.2 Builder

class popart.builder.**AiGraphcore**(*builder*, *version*)

Bases: popart.builder.Opset

Return the builder interface for the given ai.graphcore version.

Raises **ValueError** – Thrown if an invalid ai.graphcore opset version provided.

call(*args*, *num_outputs*, *callee*, *debugName=''*)

Add a call operation to the model

This is a poplar extension, to expose manual code re-use to the builder

Parameters

- **args** (List[int]) – List of tensor ids to feed as arguments.
- **num_outputs** (int) – Number of output tensors from the called graph.
- **callee** (popart.builder.Builder) – SubgraphBuilder for the graph to be called.
- **debugName** (str) –

Keyword Arguments **debugName** – A string to prepend to the name of the tensor. Default: "".

Returns Output tensor ids.

Return type List[str]

packedDataBlock(*args*, *maxSequenceLengths*, *resultSize*, *callbackBatchSize*, *callback*, *debugName=''*)

Parameters

- **args** (List[str]) –
- **maxSequenceLengths** (List[int]) –
- **resultSize** (int) –
- **callbackBatchSize** (int) –
- **callback** (popart.builder.Builder) –
- **debugName** (str) –

Return type str

class popart.builder.**AiGraphcoreOpset1**(*builder*, *version*)

Bases: popart.builder.AiGraphcore

Sub-class for backwards compatibility. Will forward all calls to AiGraphcore class.

class popart.builder.**AiOnnx**(*builder*, *version*)

Bases: popart.builder.Opset

Base class for the various AiOnnx builder interfaces. The most recent version of ONNX operators that require special treatment such as Loop, Scan, Logical_If etc. go here. While, older versions where the function signature differs are implemented on a corresponding subclass.

Parameters

- **builder** – Parent class for access.
- **version** – ai.Onnx opset version to use; 6 <= version <= 10. Default: 10.

logical_if(*args*, *num_outputs*, *else_branch*, *then_branch*, *name=''*)

If conditional operation.

Parameters

- **args** (List[str]) – List of tensor ids to feed as arguments.
- **num_outputs** (int) – Number of output tensors from the if operator.
- **else_branch** (popart.builder.Builder) – SubgraphBuilder for the graph to run if condition is false. Has num_outputs outputs: values you wish to live-out to the subgraph created by the if operation, other tensors will not be accessible to the wider graph. The number of outputs must match the number of outputs in the then_branch.
- **then_branch** (popart.builder.Builder) – SubgraphBuilder for the graph to run if condition is true. Has num_outputs outputs: values you wish to be live-out to the enclosing scope. The number of outputs must match the number of outputs in the else_branch.
- **name** (str) –

Keyword Arguments **name** – A string to prepend to the name of the tensor. Default: “”.

Returns Output tensor ids.

Return type List[str]

loop(*args*, *num_outputs*, *body*, *debugContext=''*)

Generic Looping construct op.

Parameters

- **args** (List[str]) – List of tensor ids to feed as arguments.
- **num_outputs** (int) – Number of output tensors from the loop operator.

- **body** (popart.builder.Builder) – SubgraphBuilder for the graph to run in the loop.
- **debugContext** (str) –

Keyword Arguments **debugContext** – A string to prepend to the name of the tensor. Default: "".

Returns Output tensor ids.

Return type List[str]

class popart.builder.**AiOnnx10**(*builder*, *version*)
Bases: popart.builder.AiOnnx9

Minimal builder interface for ai.onnx version 10. Once ai.onnx version 11 becomes the standard opset, this class must be updated to inherit from AiOnnx11, as described in T12084

class popart.builder.**AiOnnx11**(*builder*, *version*)
Bases: popart.builder.AiOnnx10

Minimal builder interface for ai.onnx version 11.

class popart.builder.**AiOnnx6**(*builder*, *version*)
Bases: popart.builder.AiOnnx

Minimal builder interface for ai.onnx version 6.

class popart.builder.**AiOnnx7**(*builder*, *version*)
Bases: popart.builder.AiOnnx6

Minimal builder interface for ai.onnx version 7.

class popart.builder.**AiOnnx8**(*builder*, *version*)
Bases: popart.builder.AiOnnx7

Minimal builder interface for ai.onnx version 8.

scan(*args*, *num_outputs*, *body*, *num_scan_inputs*, *directions=[]*, *debugContext=''*)
Scan-8 specific construct op.

Parameters

- **args** (List[str]) – List of tensor ids to feed as arguments.
- **num_outputs** (int) – Number of output tensors from the scan operator.
- **body** (popart.builder.Builder) – SubgraphBuilder for the graph to run in the scan.
- **num_scan_inputs** (int) – The number of scan_inputs
- **directions** (List[int]) – A list of int which specifies the direction
- **the scan_input. 0 indicates forward direction and 1** (of) –
- **reverse direction. If not omitted** (indicates) –
- **tensors** (scan_input) –
- **be scanned in the forward direction.** (will) –
- **debugContext** (str) –

Keyword Arguments **debugContext** – A string to prepend to the name of the tensor. Default: "".

Returns Output tensor ids.

Return type List[str]

class popart.builder.**AiOnnx9**(*builder*, *version*)
Bases: popart.builder.AiOnnx8

Minimal builder interface for ai.onnx version 9.

scan(*args*, *num_outputs*, *body*, *num_scan_inputs*, *scan_input_axes=[]*, *scan_input_directions=[]*, *scan_output_axes=[]*, *scan_output_directions=[]*, *debugContext=''*)
Generic Scan construct op.

Parameters

- **args** (List[str]) – List of tensor ids to feed as arguments.
- **num_outputs** (int) – Number of output tensors from the scan operator.
- **body** (popart.builder.Builder) – SubgraphBuilder for the graph to run in the scan.
- **num_scan_inputs** (int) – The number of scan_inputs
- **scan_input_axes** (List[int]) – A list that specifies the axis to be scanned for the scan_input. If omitted, 0 will be used as the scan axis for every scan_input.
- **scan_input_directions** (List[int]) – A list that specifies the direction to be scanned for the scan_input tensor. 0 indicates forward direction and 1 indicates reverse direction. If omitted, all scan_input tensors will be scanned in the forward direction.
- **scan_output_axes** (List[int]) – A list that specifies the axis for the scan_output. The scan outputs are accumulated along the specified axis. If omitted, 0 will be used as the scan axis for every scan_output.
- **scan_output_directions** (List[int]) – A list specifies whether the scan_output should be constructed by appending or prepending a new value in each iteration: 0 indicates appending and 1 indicates prepending. If omitted, all scan_output tensors will be produced by appending a value in each iteration.
- **debugContext** (str) –

Keyword Arguments **debugContext** – A string to prepend to the name of the tensor. Default: "".

Returns Output tensor ids.

Return type List[str]

class popart.builder.**AiOnnxMl**(*builder*, *version*)
Bases: popart.builder.Opset

Return the builder interface for the given ai.onnx.ml version.

Raises **ValueError** – Thrown if an invalid ai.onnx.ml opset version provided.

class popart.builder.**Builder**(*modelProtoOrFilename=None*, *opsets=None*, *builderCore=None*)
Bases: object

A wrapper around the Builder C++ class, renamed BuilderCore in pybind, to enable more Pythonic use. See builder.hpp for the class definition.

Parameters

- **modelProtoOrFilename** – Model protobuf string or file path of saved ONNX model proto. Default: None.
- **opsets** – Dict of opset versions. Default: None.
- **builderCore** – _BuilderCore object if you want to create a subgraph builder using an existing buildercore object. Default: None.

aiOnnxOpsetVersion(*version*)

Parameters **version** (int) –

Return type None

createSubgraphBuilder()
Create a child builder to add ops to a subgraph using a call operation.

Returns The child builder.

Return type *popart.builder.Builder*

reshape_const(*aiOnnx*, *args*, *shape*, *debugContext=''*)
Const version of the reshape op.

Parameters

- **aiOnnx** (popart.builder.Opset) – Versioned aiOnnx opset, for example: aiOnnxOpset11.
- **args** (List[str]) – List of tensor ids to feed as arguments.
- **shape** (Iterable[int]) – Shape to reshape to, for example [3, 2, 4].
- **debugContext** (str) –

Keyword Arguments **debugContext** – String to use as a debug Context. Default: “”.

Returns Output tensor ids.

Return type List[int]

class popart.builder.**Opset**(*builder*, *version*)
Bases: object

Minimal base class for the opsets

Parameters

- **builder** – An interface for a Builder, used for creating ONNX graphs.
- **version** – Opset version to use for the given opset sub-class.

2.3 Tensor information

class popart.tensorinfo.**TensorInfo**(**args*)
Bases: popart_internal_ir.TensorInfo

Python wrapper to TensorInfo to handle numpy types in constructor.

For example: TensorInfo(dtype, shape) TensorInfo(numpy.ndarray)

Raises **TypeError** – Raised if incorrect type is used to create a tensorinfo.

2.4 Writer

Framework independent functionality for driving PopART

class popart.writer.**NetWriter**(*inNames*, *outNames*, *optimizer*, *dataFlow*, *inputShapeInfo*)
Bases: object

Base class, to be inherited once per framework

Parameters

- **inNames** – A list (in order) of all the inputs to the ONNX Model.
- **outNames** – names of the outputs of the ONNX Model.
- **optimizer** – An optimizer (ConstSGD, SGD, etc) or None if in inference mode.
- **anchors** – Only relevant if in training mode: the names of tensors which must be computed and returned. If not in training mode, then outputs of forward are the (only) tensors to return.
- **dataFlow** – Configuration for the data feeds and fetches.

- **inputShapeInfo** – For every loss stream input and standard input: the shape, ONNX DataType and how to get data.

infer(*inputsMap*)
Perform `batchesPerStep` inference steps. This function only needs to be implemented by frameworks which will be used to verify PopART. See `torchwriter.py` for an example implementation.

saveModel(*filename*)
To be implemented once per framework: framework specific details of generating the ONNX model and writing it to file

train(*inputsMap*)
Perform `batchesPerStep` training steps. This function only needs to be implemented by frameworks which will be used to verify PopART. See `torchwriter.py` for an example implementation.

2.5 Builder

class popart_core.**_BuilderCore**

addInitializedInputTensor(**args*, ***kwargs*)
Overloaded function.

1. addInitializedInputTensor(self: popart_core._BuilderCore, initVal: numpy.ndarray, debugPrefix: popart_internal_ir.DebugContext = '') -> str
2. addInitializedInputTensor(self: popart_core._BuilderCore, initVal: numpy.ndarray, debugContext: popart_internal_ir.DebugContext = '') -> str

addInputTensor(**args*, ***kwargs*)
Overloaded function.

1. addInputTensor(self: popart_core._BuilderCore, tensorInfo: popart_internal_ir.TensorInfo, debugPrefix: popart_internal_ir.DebugContext = '') -> str
2. addInputTensor(self: popart_core._BuilderCore, tensorInfo: popart_internal_ir.TensorInfo, debugContext: popart_internal_ir.DebugContext = '') -> str
3. addInputTensor(self: popart_core._BuilderCore, dataType: str, shape: List[int], debugPrefix: popart_internal_ir.DebugContext = '') -> str
4. addInputTensor(self: popart_core._BuilderCore, dataType: str, shape: List[int], debugContext: popart_internal_ir.DebugContext = '') -> str
5. addInputTensor(self: popart_core._BuilderCore, tensorInfo: popart_internal_ir.TensorInfo, debugPrefix: popart_internal_ir.DebugContext = '') -> str
6. addInputTensor(self: popart_core._BuilderCore, tensorInfo: popart_internal_ir.TensorInfo, settings: popart_core.InputSettings = <popart_core.InputSettings object at 0x7f0f99665880>, debugContext: popart_internal_ir.DebugContext = '') -> str
7. addInputTensor(self: popart_core._BuilderCore, dataType: str, shape: List[int], settings: popart_core.InputSettings = <popart_core.InputSettings object at 0x7f0f996658b8>, debugPrefix: popart_internal_ir.DebugContext = '') -> str
8. addInputTensor(self: popart_core._BuilderCore, dataType: str, shape: List[int], settings: popart_core.InputSettings = <popart_core.InputSettings object at 0x7f0f996658f0>, debugContext: popart_internal_ir.DebugContext = '') -> str

addInputTensorFromParentGraph(*self:* popart_core._BuilderCore, *tensorId: str*) → None
Add a new named input tensor to the model.

Parameter tensorId: The identifier string of the input tensor. This identifier must already exist in the parent GraphProto's name scope and must appear topologically before this sub-graph.

addNodeAttribute(**args*, ***kwargs*)
Overloaded function.

1. addNodeAttribute(self: popart_core._BuilderCore, attributeName: str, attributeValue: int, nodeOutputNames: Set[str]) -> None
2. addNodeAttribute(self: popart_core._BuilderCore, attributeName: str, attributeValue: List[int], nodeOutputNames: Set[str]) -> None
3. addNodeAttribute(self: popart_core._BuilderCore, attributeName: str, attributeValue: float, nodeOutputNames: Set[str]) -> None
4. addNodeAttribute(self: popart_core._BuilderCore, attributeName: str, attributeValue: List[float], nodeOutputNames: Set[str]) -> None
5. addNodeAttribute(self: popart_core._BuilderCore, attributeName: str, attributeValue: str, nodeOutputNames: Set[str]) -> None
6. addNodeAttribute(self: popart_core._BuilderCore, attributeName: str, attributeValue: List[str], nodeOutputNames: Set[str]) -> None

addOutputTensor(*self:* popart_core._BuilderCore, *outputName: str*) → None

addUntypedInputTensor(**args*, ***kwargs*)
Overloaded function.

1. addUntypedInputTensor(self: popart_core._BuilderCore, debugPrefix: popart_internal_ir.DebugContext = '') -> str

Add a new input tensor without a type or shape to the model.

Parameter debugContext: Optional debug information.

Returns The unique name of the input tensor.

2. addUntypedInputTensor(self: popart_core._BuilderCore, debugContext: popart_internal_ir.DebugContext = '') -> str

Add a new input tensor without a type or shape to the model.

Parameter debugContext: Optional debug information.

Returns The unique name of the input tensor.

checkpointOutput(*self:* popart_core._BuilderCore, *nodeOutputNames: List[str]*) → List[str]

commGroup(*self:* popart_core._BuilderCore, *type: int = 0*, *groupSize: int = 0*) → AttributeContextManager

customOp(*self:* popart_core._BuilderCore, *opName: str*, *opVersion: int*, *domain: str*, *inputs: list*, *attributes: dict*, *numOutputs: int = 1*, *name: str = ''*) → List[str]

excludePatterns(*self:* popart_core._BuilderCore, *nodeOutputName: str*, *patternNames: List[str]*) → None

executionPhase(**args*, ***kwargs*)
Overloaded function.

1. executionPhase(self: popart_core._BuilderCore, nodeOutputNames: str, value: int = 0) -> None
2. executionPhase(self: popart_core._BuilderCore, nodeOutputNames: Set[str], value: int = 0) -> None
3. executionPhase(self: popart_core._BuilderCore, value: int = 0) -> AttributeContextManager

getAllNodeAttributeNames(*self:* popart_core._BuilderCore, *nodeOutputNames: Set[str]*) → List[str]
Get all the attribute names from the ONNX node. This functions will throw an exception if it can't find the unique node.

Parameter nodeOutputNames: Names of the output tensors of the ONNX node used to find the node in the ONNX model.

getExecutionPhase(*self:* popart_core._BuilderCore) → int

getFloatNodeAttribute(*self:* popart_core._BuilderCore, *attributeName:* *str*, *nodeOutputNames:* *Set[str]*) → float

Get the `float` value of the attribute for the ONNX node. This functions will throw an exception if it can't find the unique node or the attribute does not exist or it has not been set to the `float` type.

Parameter attributeName: The name of the attribute to find.

Parameter nodeOutputNames: Names of the output tensors of the ONNX node used to find the node in the ONNX model.

Returns Value of the attribute.

getFloatVectorNodeAttribute(*self:* popart_core._BuilderCore, *attributeName:* *str*, *nodeOutputNames:* *Set[str]*) → List[float]

Get the ``std::vector``<float> value of the attribute for the ONNX node. This functions will throw an exception if it can't find the unique node or the attribute does not exist.

Parameter attributeName: The name of the attribute to find.

Parameter nodeOutputNames: Names of the output tensors of the ONNX node used to find the node in the ONNX model.

Returns Value of the attribute.

getInputTensorIds(*self:* popart_core._BuilderCore) → List[str]

getInt64NodeAttribute(*self:* popart_core._BuilderCore, *attributeName:* *str*, *nodeOutputNames:* *Set[str]*) → int

Get the `int64_t` value of the attribute for the ONNX node. This functions will throw an exception if it can't find the unique node or the attribute does not exist or it has not been set to the `int64_t` type.

Parameter attributeName: The name of the attribute to find.

Parameter nodeOutputNames: Names of the output tensors of the ONNX node used to find the node in the ONNX model.

Returns Value of the attribute.

getInt64VectorNodeAttribute(*self:* popart_core._BuilderCore, *attributeName:* *str*, *nodeOutputNames:* *Set[str]*) → List[int]

Get the ``std::vector``<int64_t> value of the attribute for the ONNX node. This functions will throw an exception if it can't find the unique node or the attribute does not exist or it has not been set to the ``std::vector``<int64_t> type.

Parameter attributeName: The name of the attribute to find.

Parameter nodeOutputNames: Names of the output tensors of the ONNX node used to find the node in the ONNX model.

Returns Value of the attribute.

getModelProto(*self:* popart_core._BuilderCore) → bytes

getNameScope(*self:* popart_core._BuilderCore, *name:* *str* = '') → str

getOutputTensorIds(*self:* popart_core._BuilderCore) → List[str]

getPartialsType(*self:* popart_core._BuilderCore, *nodeOutputName:* *str*) → str

Get the partials type for the given node.

Parameter nodeOutputName: Name of the output tensor of the ONNX node.

getPipelineStage(*self:* popart_core._BuilderCore) → int

getRecomputeOutputInBackwardPass(**args*, ***kwargs*)
Overloaded function.

1. getRecomputeOutputInBackwardPass(self: popart_core._BuilderCore, nodeOutputName: str) -> bool

2. getRecomputeOutputInBackwardPass(self: popart_core._BuilderCore, nodeOutputNames: Set[str]) -> bool

getStringNodeAttribute(*self:* popart_core._BuilderCore, *attributeName:* *str*, *nodeOutputNames:* *Set[str]*) → str
Get the `std::string` value of the attribute for the ONNX node. This functions will throw an exception if it can't find the unique node or the attribute does not exist or it has not been set to the `std::string` type.

Parameter attributeName: The name of the attribute to find.

Parameter nodeOutputNames: Names of the output tensors of the ONNX node used to find the node in the ONNX model.

Returns Value of the attribute.

getStringVectorNodeAttribute(*self:* popart_core._BuilderCore, *attributeName:* *str*, *nodeOutputNames: Set[str]*) → List[str]
Get the ``std::vector``<std::string> value of the attribute for the ONNX node. This functions will throw an exception if it can't find the unique node or the attribute does not exist.

Parameter attributeName: The name of the attribute to find.

Parameter nodeOutputNames: Names of the output tensors of the ONNX node used to find the node in the ONNX model.

Returns Value of the attribute.

getTensorDtypeString(*self:* popart_core._BuilderCore, *id:* *str*) → str

getTensorShape(*self:* popart_core._BuilderCore, *id:* *str*) → List[int]
Return an ONNX graph tensor shape, from either the input, output, or value_info lists in the GraphProto.

Parameter id: Tensor id.

Returns A vector of tensor dimensions.

getTrainableTensorIds(*self:* popart_core._BuilderCore) → List[str]

getValueTensorIds(*self:* popart_core._BuilderCore) → List[str]

getVirtualGraph(**args*, ***kwargs*)
Overloaded function.

1. getVirtualGraph(self: popart_core._BuilderCore) -> int

2. getVirtualGraph(self: popart_core._BuilderCore, nodeOutputNames: str) -> int

3. getVirtualGraph(self: popart_core._BuilderCore, nodeOutputNames: Set[str]) -> int

4. getVirtualGraph(self: popart_core._BuilderCore, nodeOutputNames: List[str]) -> int

hasExecutionPhase(*self:* popart_core._BuilderCore) → bool

hasPipelineStage(*self:* popart_core._BuilderCore) → bool

hasVirtualGraph(*self:* popart_core._BuilderCore) → bool

isInitializer(*self:* popart_core._BuilderCore, *id: str*) → bool
Returns true if the ONNX tensor is in the initializer list of the GraphProto.

Parameter id: Tensor id.

Returns A boolean.

nameScope(*self:* popart_core._BuilderCore, *name: str*) → NameContextManager

nodeHasAttribute(*self:* popart_core._BuilderCore, *attributeName: str*, *nodeOutputNames: Set[str]*) → bool
Check whether the ONNX node has an attribute set. This functions will throw an exception if it can't find the unique node.

Parameter attributeName: The name of the attribute to find.

Parameter nodeOutputNames: Names of the output tensors of the ONNX node used to find the node in the ONNX model.

outlineAttributes(*self:* popart_core._BuilderCore, *arg0: dict*) → KeyValueContextManager

outputTensorLocation(**args*, ***kwargs*)
Overloaded function.

1. outputTensorLocation(self: popart_core._BuilderCore, nodeOutputNames: str, value: popart_core.TensorLocation = <popart_core.TensorLocation object at 0x7f0f996659d0>) -> None

2. outputTensorLocation(self: popart_core._BuilderCore, value: popart_core.TensorLocation = <popart_core.TensorLocation object at 0x7f0f99665a08>) -> AttributeContextManager

pipelineStage(**args*, ***kwargs*)
Overloaded function.

1. pipelineStage(self: popart_core._BuilderCore, nodeOutputNames: str, value: int = 0) -> None

2. pipelineStage(self: popart_core._BuilderCore, value: int) -> AttributeContextManager

recomputeOutput(**args*, ***kwargs*)
Overloaded function.

1. recomputeOutput(self: popart_core._BuilderCore, nodeOutputNames: str, value: popart_core.RecomputeType = <RecomputeType.Undefined: 0>) -> None

2. recomputeOutput(self: popart_core._BuilderCore, value: popart_core.RecomputeType = <RecomputeType.Undefined: 0>) -> AttributeContextManager

recomputeOutputInBackwardPass(**args*, ***kwargs*)
Overloaded function.

1. recomputeOutputInBackwardPass(self: popart_core._BuilderCore, nodeOutputName: str, value: popart_core.RecomputeType = <RecomputeType.Recompute: 2>) -> None

2. recomputeOutputInBackwardPass(self: popart_core._BuilderCore, nodeOutputNames: Set[str], value: popart_core.RecomputeType = <RecomputeType.Recompute: 2>) -> None

removeNodeAttribute(*self:* popart_core._BuilderCore, *attributeName: str*, *nodeOutputNames: Set[str]*) → None
Remove an attribute from the ONNX node. This functions will throw an exception if it can't find the unique node or the attribute does not exist.

Parameter attributeName: The name of the attribute to find.

Parameter nodeOutputNames: Names of the output tensors of the ONNX node used to find the node in the ONNX model.

saveInitializersExternally(*self:* popart_core._BuilderCore, *ids: List[str]*, *filename: str*) → None
Save tensor data externally.

The model data cannot exceed 2GB - the maximum size of a Protobuf message. To avoid this, for large models ONNX tensor data can be saved separately.

Parameter ids: The names of tensors whose data is to be saved externally.

Parameter fn: The name of a file containing the binary tensor data. This can be an absolute or relative path. If a relative path, when the ONNX model is saved, external tensor data will be written to a path relative to your current working directory.

saveModelProto(*self:* popart_core._BuilderCore, *filename: str*) → None
Save the builder's ONNX ModelProto into the builder and validate it.

Parameter fn: The name of a file containing an ONNX model protobuf.

schedulePriority(*self:* popart_core._BuilderCore, *value: float*) → AttributeContextManager

setAvailableMemoryProportion(*self:* popart_core._BuilderCore, *nodeOutputName: str*, *availableMemoryProportion: float*) → None
Set the available memory for the given node. Used on the convolution op.

Parameter nodeOutputName: Name of the output tensor of the ONNX node.

Parameter availableMemoryProportion: The available memory proportion 0 < x <= 1.

setEnableConvDithering(*self:* popart_core._BuilderCore, *nodeOutputName: str*, *enableConvDithering: int*) → None
Enable convolution dithering.

Parameter nodeOutputName: Name of the output tensor of the ONNX node.

Parameter value: 1, if convolution dithering should be enabled. 0, otherwise.

setGraphName(**args*, ***kwargs*)
Overloaded function.

1. setGraphName(self: popart_core._BuilderCore, name: str) -> None

Specifies a graph name.

Parameter name: String to name the graph.

2. setGraphName(self: popart_core._BuilderCore, name: str) -> None

setInplacePreferences(*self:* popart_core._BuilderCore, *nodeOutputName: str*, *prefs: Dict[str, float]*) → None

setPartialsType(*self:* popart_core._BuilderCore, *nodeOutputName: str*, *partialsType: str*) → None
Set the partials type for the given node. Used on the convolution op.

Parameter nodeOutputName: Name of the output tensor of the ONNX node.

Parameter partialsType: The type for the partials. Can be either FLOAT or HALF.

setSerializeMatMul(*self:* popart_core._BuilderCore, *nodeOutputName: Set[str]*, *mode: str*, *factor: int = 0*, *keep_precision: bool = False*) → None

virtualGraph(**args*, ***kwargs*)
Overloaded function.

1. virtualGraph(self: popart_core._BuilderCore, nodeOutputNames: str, value: int = 0) -> None
2. virtualGraph(self: popart_core._BuilderCore, nodeOutputNames: Set[str], value: int = 0) -> None
3. virtualGraph(self: popart_core._BuilderCore, nodeOutputNames: List[str], value: int = 0) -> None
4. virtualGraph(self: popart_core._BuilderCore, value: int) -> AttributeContextManager

2.5.1 AiGraphcoreOpset1

class popart_core.**AiGraphcoreOpset1**

abort(**args*, ***kwargs*)
Overloaded function.

1. abort(self: popart_core.AiGraphcoreOpset1, args: List[str] = [], debugPrefix: popart_internal_ir.DebugContext = '') -> None

Add abort operation to the model.

The operation can be conditional or unconditional.

Parameter args: Optional input tensor to test condition

2. abort(self: popart_core.AiGraphcoreOpset1, args: List[str] = [], debugContext: popart_internal_ir.DebugContext = '') -> None

Add abort operation to the model.

The operation can be conditional or unconditional.

Parameter args: Optional input tensor to test condition

atan2(**args*, ***kwargs*)
Overloaded function.

1. atan2(self: popart_core.AiGraphcoreOpset1, args: List[str], debugPrefix: popart_internal_ir.DebugContext = '') -> str

Add an `atan2` operation to the model.

Returns the element-wise angle theta as a tensor, -pi < theta <= pi, such that for two input tensors x and y and given r != 0, x = r cos theta, and y = r sin theta, element-wise.

In the case of x > 0, theta = arctan(y/x).

Parameter args: Vector of input tensor ids: [y, x].

Parameter name: Optional identifier for operation.

Returns The name of the result tensor containing element wise theta values.

2. atan2(self: popart_core.AiGraphcoreOpset1, args: List[str], debugContext: popart_internal_ir.DebugContext = '') -> str

Add an `atan2` operation to the model.

Returns the element-wise angle theta as a tensor, -pi < theta <= pi, such that for two input tensors x and y and given r != 0, x = r cos theta, and y = r sin theta, element-wise.

In the case of x > 0, theta = arctan(y/x).

Parameter args: Vector of input tensor ids: [y, x].

Parameter name: Optional identifier for operation.

Returns The name of the result tensor containing element wise theta values.

bitwiseand(*self:* popart_core.AiGraphcoreOpset1, *args: List[str] = []*, *debugContext: popart_internal_ir.DebugContext = ''*) → str
Add a bitwise AND operation to the model.

The operation computes the bitwise AND of given two integer tensors.

Parameter args: Two broadcastable input tensors of type integer.

Returns The name of the result tensor.

bitwisenot(*self:* popart_core.AiGraphcoreOpset1, *args:* *List[str]* = [], *debugContext:* *popart_internal_ir.DebugContext* = '') → str

Add a bitwise NOT operation to the model.

The operation computes the bitwise NOT of a given integer tensor.

Parameter args: Input tensor of type integer.

Returns The name of the result tensor.

bitwiseor(*self:* popart_core.AiGraphcoreOpset1, *args:* *List[str]* = [], *debugContext:* *popart_internal_ir.DebugContext* = '') → str

Add a bitwise OR operation to the model.

The operation computes the bitwise OR of given two integer tensors.

Parameter args: Two broadcastable input tensors of type integer.

Returns The name of the result tensor.

bitwisexnor(*self:* popart_core.AiGraphcoreOpset1, *args:* *List[str]* = [], *debugContext:* *popart_internal_ir.DebugContext* = '') → str

Add a bitwise XNOR operation to the model.

The operation computes the bitwise XNOR of given two integer tensors.

Parameter args: Two broadcastable input tensors of type integer.

Returns The name of the result tensor.

bitwisexor(*self:* popart_core.AiGraphcoreOpset1, *args:* *List[str]* = [], *debugContext:* *popart_internal_ir.DebugContext* = '') → str

Add a bitwise XOR operation to the model.

The operation computes the bitwise XOR of given two integer tensors.

Parameter args: Two broadcastable input tensors of type integer.

Returns The name of the result tensor.

call(**args*, ***kwargs*)

Overloaded function.

1. call(self: popart_core.AiGraphcoreOpset1, args: List[str], num_outputs: int, callee: popart::Builder, debugPrefix: popart_internal_ir.DebugContext = '') -> List[str]

Add a call operation to the model

This is a Poplar extension, to expose manual code re-use to the builder.

Parameter args: Vector of input tensor ids.

Parameter callee: The subgraph to call into.

Parameter debugContext: Optional debug context.

Returns A vector of tensors; the subgraph outputs.

2. call(self: popart_core.AiGraphcoreOpset1, args: List[str], num_outputs: int, callee: popart::Builder, debugContext: popart_internal_ir.DebugContext = '') -> List[str]

Add a call operation to the model

This is a Poplar extension, to expose manual code re-use to the builder.

Parameter `args`: Vector of input tensor ids.

Parameter `callee`: The subgraph to call into.

Parameter `debugContext`: Optional debug context.

Returns A vector of tensors; the subgraph outputs.

`copyvarupdate`(**args*, ***kwargs*)

Overloaded function.

1. copyvarupdate(self: popart_core.AiGraphcoreOpset1, args: List[str], debugPrefix: popart_internal_ir.DebugContext = '') -> str

Copies a tensor to an initalised tensor (variable)

This is used to update an initalised tensor (a variable created using addInitializedInputTensor) which retains its value between iterations, by setting the value to the value of another tensor (the updater). The purpose is to manually update the tensor in use cases for variables other than trained parameters (weights) or tensors used by other ops.

Parameter `args`: A vector of the input tensors [tensor to update, updater]

Parameter `debugContext`: Optional debug context.

Returns to ensure correct ordering of the updated variable, you should use this variable for any op which should operate on the updated variable.

Return type An alias to the updated variable

2. copyvarupdate(self: popart_core.AiGraphcoreOpset1, args: List[str], debugContext: popart_internal_ir.DebugContext = '') -> str

Copies a tensor to an initalised tensor (variable)

This is used to update an initalised tensor (a variable created using addInitializedInputTensor) which retains its value between iterations, by setting the value to the value of another tensor (the updater). The purpose is to manually update the tensor in use cases for variables other than trained parameters (weights) or tensors used by other ops.

Parameter `args`: A vector of the input tensors [tensor to update, updater]

Parameter `debugContext`: Optional debug context.

Returns to ensure correct ordering of the updated variable, you should use this variable for any op which should operate on the updated variable.

Return type An alias to the updated variable

`ctcbeamsearchdecoder`(*self: popart_core.AiGraphcoreOpset1*, *args: List[str]*, *blank: int = 0*, *beam_width: int = 100*, *top_paths: int = 1*, *debug_context: popart_internal_ir.DebugContext = ''*) → List[str]

Add a connectionist temporal classification (CTC) beam search decoder operation to the model.

Calculate the most likely p topPaths labels and their probabilities given the input p logProbs with lengths p dataLengths.

Parameter `args`: Vector of input tensor ids. These are [logProbs, dataLengths], where logProbs is of shape [maxTime, batchSize, numClasses], and dataLengths is of shape [batchSize].

Parameter `blank`: The integer representing the blank class.

Parameter `beamWidth`: The number of beams to use when decoding.

Parameter `topPaths`: The number of most likely decoded paths to return, must be less than or equal to p beamWidth.

Parameter debugContext: Optional debug context.

Returns The names of the result tensors. These are [labelProbs, labelLengths, decodedLabels], where labelProbs is of shape [batchSize, topPaths], labelLengths is of shape [batchSize, topPaths], and decodedLabels is of shape [batchSize, topPaths, maxTime].

ctcloss(*self: popart_core.AiGraphcoreOpset1*, *args: List[str]*, *reduction: popart_core.ReductionType = <ReductionType.Mean: 1>*, *blank: int = 0*, *outDataType: str = 'UNDEFINED'*, *debugContext: popart_internal_ir.DebugContext = ''*) → str

Add a connectionist temporal classification (CTC) loss operation to the model.

With T being maximum input length, N being batch size, C being number of classes, S being a maximum target length, this op calculates the CTC loss for a logarithmised probabilities tensor with shape [T, N, C], a class target tensor with shape [N, S], an input lengths tensor [N] and a target lengths tensor [N].

Note that C includes a blank class (default=0). The probabilities tensor is padded as required. Target sequences are also padded and are populated with values less than equal to C, not including the blank class, up to their respective target lengths. Note that target lengths cannot exceed input lengths.

Parameter args: [log_probs,targets,input_lengths,target_lengths]

Parameter reduction: Type of reduction to perform on the individual losses

Parameter blank: The integer representing the blank class.

Parameter debugContext: Optional debug context

Returns The name of the result tensor

depthtospace(**args*, ***kwargs*)

Overloaded function.

1. depthtospace(self: popart_core.AiGraphcoreOpset1, args: List[str], blocksize: int, mode: str, debugPrefix: popart_internal_ir.DebugContext = '') -> str

Add the `DepthToSpace` to the model. (This allows DepthToSpace_11 to be targeted from earlier opsets.)

The purpose of Depth to Space, also known as pixel shuffling, is to rearrange data from the depth (channels) dimension into the spacial (width and height) dimensions. It is an efficient means of learning upsampling alongside mixing convolution with bilinear interpolation and using transpose convolution.

https://github.com/onnx/onnx/blob/master/docs/Operators.md#DepthToSpac e

Parameter args: Vector containing single tensor input id.

Parameter blocksize: Indicates the scale factor: if the input is [N, C, H, W] and the blocksize is B, the output will be [N, C/(B*B), H*B, W*B].

Parameter mode: Specifies how the data is rearranged: * “DCR”: depth-column-row order * “CRD”: column-row-depth order

Parameter debugContext: Optional debug context.

Returns A tensor which is a rearrangement of the input tensor.

2. depthtospace(self: popart_core.AiGraphcoreOpset1, args: List[str], blocksize: int, mode: str, debugContext: popart_internal_ir.DebugContext = '') -> str

Add the `DepthToSpace` to the model. (This allows DepthToSpace_11 to be targeted from earlier opsets.)

The purpose of Depth to Space, also known as pixel shuffling, is to rearrange data from the depth (channels) dimension into the spacial (width and height) dimensions. It is an efficient means of learning upsampling alongside mixing convolution with bilinear interpolation and using transpose convolution.

https://github.com/onnx/onnx/blob/master/docs/Operators.md#DepthToSpac e

Parameter `args`: Vector containing single tensor input id.

Parameter `blocksize`: Indicates the scale factor: if the input is [N, C, H, W] and the blocksize is B, the output will be [N, C/(B*B), H*B, W*B].

Parameter `mode`: Specifies how the data is rearranged: * “DCR”: depth-column-row order * “CRD”: column-row-depth order

Parameter `debugContext`: Optional debug context.

Returns A tensor which is a rearrangement of the input tensor.

`detach`(**args*, ***kwargs*)

Overloaded function.

1. detach(self: popart_core.AiGraphcoreOpset1, args: List[str], debugPrefix: popart_internal_ir.DebugContext = '') -> str

Add a detach operation to the model.

Parameter `args`: Vector of input tensor ids.

Parameter `debugContext`: Optional debug context.

Returns The name of the result tensor.

2. detach(self: popart_core.AiGraphcoreOpset1, args: List[str], debugContext: popart_internal_ir.DebugContext = '') -> str

Add a detach operation to the model.

Parameter `args`: Vector of input tensor ids.

Parameter `debugContext`: Optional debug context.

Returns The name of the result tensor.

`dynamicadd`(**args*, ***kwargs*)

Overloaded function.

1. dynamicadd(self: popart_core.AiGraphcoreOpset1, args: List[str], axes: List[int], sizes: List[int], debugPrefix: popart_internal_ir.DebugContext = '') -> str

Add a dynamic add operation to the model.

Creates a copy of `tensor` with `slice` added at `offset`. For example:

out = tensor, out[offset] += slice

Parameter `args`: Vector of input tensor ids: [tensor, offset, slice].

Parameter `axes`: Axes along which to add.

Parameter `sizes`: Size of the slice in each axis.

Parameter `debugContext`: Optional debug context.

Returns The name of the result tensor.

2. dynamicadd(self: popart_core.AiGraphcoreOpset1, args: List[str], axes: List[int], sizes: List[int], debugContext: popart_internal_ir.DebugContext = '') -> str

Add a dynamic add operation to the model.

Creates a copy of `tensor` with `slice` added at `offset`. For example:

out = tensor, out[offset] += slice

Parameter args: Vector of input tensor ids: [tensor, offset, slice].

Parameter axes: Axes along which to add.

Parameter sizes: Size of the slice in each axis.

Parameter debugContext: Optional debug context.

Returns The name of the result tensor.

dynamicslice(**args*, ***kwargs*)

Overloaded function.

1. dynamicslice(self: popart_core.AiGraphcoreOpset1, args: List[str], axes: List[int], sizes: List[int], noOverlap: int = 0, debugPrefix: popart_internal_ir.DebugContext = '') -> str

Add a dynamic slice operation to the model.

Creates a new slice tensor. For example:

slice = tensor[offset]

Parameter args: Vector of input tensor ids: [tensor, offset].

Parameter axes: Axes along which to slice.

Parameter sizes: Size of the slice in each axis.

Parameter debugContext: Optional debug context.

Returns The name of the result tensor.

2. dynamicslice(self: popart_core.AiGraphcoreOpset1, args: List[str], axes: List[int], sizes: List[int], noOverlap: int = 0, debugContext: popart_internal_ir.DebugContext = '') -> str

Add a dynamic slice operation to the model.

Creates a new slice tensor. For example:

slice = tensor[offset]

Parameter args: Vector of input tensor ids: [tensor, offset].

Parameter axes: Axes along which to slice.

Parameter sizes: Size of the slice in each axis.

Parameter debugContext: Optional debug context.

Returns The name of the result tensor.

dynamicupdate(**args*, ***kwargs*)

Overloaded function.

1. dynamicupdate(self: popart_core.AiGraphcoreOpset1, args: List[str], axes: List[int], sizes: List[int], noOverlap: int = 0, debugPrefix: popart_internal_ir.DebugContext = '') -> str

Add a dynamic update operation to the model.

Creates a copy of a `tensor` with a `slice` inserted at `offset`. For example:

out = tensor, out[offset] = slice

Parameter args: Vector of input tensor ids: [tensor, offset, slice].

Parameter axes: Axes along which to update.

Parameter sizes: Size of the slice in each axis.

Parameter debugContext: Optional debug context.

Returns The name of the result tensor.

2. dynamicupdate(self: popart_core.AiGraphcoreOpset1, args: List[str], axes: List[int], sizes: List[int], noOverlap: int = 0, debugContext: popart_internal_ir.DebugContext = '') -> str

Add a dynamic update operation to the model.

Creates a copy of a `tensor` with a `slice` inserted at `offset`. For example:

out = tensor, out[offset] = slice

Parameter `args`: Vector of input tensor ids: [tensor, offset, slice].

Parameter `axes`: Axes along which to update.

Parameter `sizes`: Size of the slice in each axis.

Parameter `debugContext`: Optional debug context.

Returns The name of the result tensor.

`dynamiczero`(**args*, ***kwargs*)

Overloaded function.

1. dynamiczero(self: popart_core.AiGraphcoreOpset1, args: List[str], axes: List[int], sizes: List[int], debugPrefix: popart_internal_ir.DebugContext = '') -> str

Add a dynamic zero operation to the model.

Creates a copy of `tensor` with a slice at `offset` set to zero. For example:

out = tensor, out[offset] = 0.0

Parameter `args`: Vector of input tensor ids [tensor, offset].

Parameter `axes`: Axes along which to erase.

Parameter `sizes`: Size of the slice in each axis.

Parameter `debugContext`: Optional debug context.

Returns The name of the result tensor.

2. dynamiczero(self: popart_core.AiGraphcoreOpset1, args: List[str], axes: List[int], sizes: List[int], debugContext: popart_internal_ir.DebugContext = '') -> str

Add a dynamic zero operation to the model.

Creates a copy of `tensor` with a slice at `offset` set to zero. For example:

out = tensor, out[offset] = 0.0

Parameter `args`: Vector of input tensor ids [tensor, offset].

Parameter `axes`: Axes along which to erase.

Parameter `sizes`: Size of the slice in each axis.

Parameter `debugContext`: Optional debug context.

Returns The name of the result tensor.

`expm1`(**args*, ***kwargs*)

Overloaded function.

1. expm1(self: popart_core.AiGraphcoreOpset1, args: List[str], debugPrefix: popart_internal_ir.DebugContext = '') -> str

Add `expm1` operation to the model. It computes exp(x) - 1. Calculates the element-wise exponential of the input tensor and subtracts one.

Parameter `args`: Vector of input tensor ids.

Parameter `name`: Optional identifier for operation.

Returns The name of the result tensor.

2. expm1(self: popart_core.AiGraphcoreOpset1, args: List[str], debugContext: popart_internal_ir.DebugContext = '') -> str

Add `expm1` operation to the model. It computes exp(x) - 1. Calculates the element-wise exponential of the input tensor and subtracts one.

Parameter `args`: Vector of input tensor ids.

Parameter `name`: Optional identifier for operation.

Returns The name of the result tensor.

`fmod`(**args*, ***kwargs*)

Overloaded function.

1. fmod(self: popart_core.AiGraphcoreOpset1, args: List[str], debugContext: popart_internal_ir.DebugContext = '') -> str

Add fmod operation to the model.

This is equivalent to C's fmod function. The result has the same sign as the dividend.

Parameter `args`: Input tensors.

Returns Computes the element-wise remainder of division. The remainder has the same sign as the dividend.

2. fmod(self: popart_core.AiGraphcoreOpset1, args: List[str], debugPrefix: popart_internal_ir.DebugContext = '') -> str

Add fmod operation to the model.

This is equivalent to C's fmod function. The result has the same sign as the dividend.

Parameter `args`: Input tensors.

Returns Computes the element-wise remainder of division. The remainder has the same sign as the dividend.

`gelu`(**args*, ***kwargs*)

Overloaded function.

1. gelu(self: popart_core.AiGraphcoreOpset1, args: List[str], debugPrefix: popart_internal_ir.DebugContext = '') -> str

Add a GELU operation to the model.

This is a Poplar extension.

Parameter `args`: Vector of input tensor ids.

Parameter `debugContext`: Optional debug context.

Returns The name of the result tensor.

2. gelu(self: popart_core.AiGraphcoreOpset1, args: List[str], debugContext: popart_internal_ir.DebugContext = '') -> str

Add a GELU operation to the model.

This is a Poplar extension.

Parameter `args`: Vector of input tensor ids.

Parameter `debugContext`: Optional debug context.

Returns The name of the result tensor.

`groupnormalization`(**args*, ***kwargs*)

Overloaded function.

1. groupnormalization(self: popart_core.AiGraphcoreOpset1, args: List[str], num_groups: int, epsilon: float = 9.999999747378752e-06, debugPrefix: popart_internal_ir.DebugContext = '') -> List[str]

Add a group normalization operation to the model.

This is a Poplar extension.

The group will be created from a strided input.

Parameter `args`: A vector of input tensors: [x, scale, bias].

Parameter `num_groups`: The number of groups to separate the channels into.

Parameter `epsilon`: The epsilon value to use to avoid division by zero.

Parameter `debugContext`: Optional debug context.

Returns [y, mean, var].

Return type A vector of tensors

2. groupnormalization(self: popart_core.AiGraphcoreOpset1, args: List[str], num_groups: int, epsilon: float = 9.999999747378752e-06, debugContext: popart_internal_ir.DebugContext = '') -> List[str]

Add a group normalization operation to the model.

This is a Poplar extension.

The group will be created from a strided input.

Parameter `args`: A vector of input tensors: [x, scale, bias].

Parameter `num_groups`: The number of groups to separate the channels into.

Parameter `epsilon`: The epsilon value to use to avoid division by zero.

Parameter `debugContext`: Optional debug context.

Returns [y, mean, var].

Return type A vector of tensors

`identityloss`(**args*, ***kwargs*)

Overloaded function.

1. identityloss(self: popart_core.AiGraphcoreOpset1, args: List[str], reduction: popart_core.ReductionType = <ReductionType.Mean: 1>, debugPrefix: popart_internal_ir.DebugContext = '') -> str

Add an identity loss operation to the model.

Calculates the loss using the identity operator.

Parameter `args`: Vector of input tensor ids.

Parameter `reduction`: Type of reduction to perform on the individual losses.

Parameter `debugContext`: Optional debug context.

Returns The name of the result tensor

2. identityloss(self: popart_core.AiGraphcoreOpset1, args: List[str], reduction: popart_core.ReductionType = <ReductionType.Mean: 1>, debugContext: popart_internal_ir.DebugContext = '') -> str

Add an identity loss operation to the model.

Calculates the loss using the identity operator.

Parameter `args`: Vector of input tensor ids.

Parameter `reduction`: Type of reduction to perform on the individual losses.

Parameter `debugContext`: Optional debug context.

Returns The name of the result tensor

`init`(**args*, ***kwargs*)

Overloaded function.

1. init(self: popart_core.AiGraphcoreOpset1, shape: List[int], data_type: int, init_type: int, batch_axis: int, debugPrefix: popart_internal_ir.DebugContext = '') -> str

Add an init operation to the model.

Parameter `shape`: Shape of the tensor to initialise.

Parameter `data_type`: Data type to initialise tensor with.

Parameter `init_type`: Mode of tensor initialisations.

Parameter `batch_axis`: Axis relative to batch size.

Parameter `debugContext`: Optional debug context.

Returns The name of the result tensor.

2. init(self: popart_core.AiGraphcoreOpset1, shape: List[int], data_type: int, init_type: int, batch_axis: int, debugContext: popart_internal_ir.DebugContext = '') -> str

Add an init operation to the model.

Parameter `shape`: Shape of the tensor to initialise.

Parameter `data_type`: Data type to initialise tensor with.

Parameter `init_type`: Mode of tensor initialisations.

Parameter `batch_axis`: Axis relative to batch size.

Parameter `debugContext`: Optional debug context.

Returns The name of the result tensor.

3. init(self: popart_core.AiGraphcoreOpset1, shape: List[int], data_type: int, init_type: int, debugPrefix: popart_internal_ir.DebugContext = '') -> str

Add an init operation to the model.

Parameter `shape`: Shape of the tensor to initialise.

Parameter `data_type`: Data type to initialise tensor with.

Parameter `init_type`: Mode of tensor initialisations.

Parameter `debugContext`: Optional debug context.

Returns The name of the result tensor.

4. init(self: popart_core.AiGraphcoreOpset1, shape: List[int], data_type: int, init_type: int, debugContext: popart_internal_ir.DebugContext = '') -> str

Add an init operation to the model.

Parameter `shape`: Shape of the tensor to initialise.

Parameter `data_type`: Data type to initialise tensor with.

Parameter `init_type`: Mode of tensor initialisations.

Parameter `debugContext`: Optional debug context.

Returns The name of the result tensor.

`l1loss`(**args*, ***kwargs*)

Overloaded function.

1. l1loss(self: popart_core.AiGraphcoreOpset1, args: List[str], lambda: float, reduction: popart_core.ReductionType = <ReductionType.Mean: 1>, debugPrefix: popart_internal_ir.DebugContext = '') -> str

Add an `l1` loss operation to the model.

Calculates the mean absolute error between each element in the input with a zero target.

Parameter `args`: Vector of input tensor ids.

Parameter `lambda`: Scale factor of L1 loss.

Parameter `reduction`: Type of reduction to perform on the individual losses.

Parameter `debugContext`: Optional debug context.

Returns The name of the result tensor.

2. l1loss(self: popart_core.AiGraphcoreOpset1, args: List[str], lambda: float, reduction: popart_core.ReductionType = <ReductionType.Mean: 1>, debugContext: popart_internal_ir.DebugContext = '') -> str

Add an `l1` loss operation to the model.

Calculates the mean absolute error between each element in the input with a zero target.

Parameter `args`: Vector of input tensor ids.

Parameter `lambda`: Scale factor of L1 loss.

Parameter `reduction`: Type of reduction to perform on the individual losses.

Parameter `debugContext`: Optional debug context.

Returns The name of the result tensor.

`log1p`(**args*, ***kwargs*)

Overloaded function.

1. log1p(self: popart_core.AiGraphcoreOpset1, args: List[str], debugPrefix: popart_internal_ir.DebugContext = '') -> str

Add `log1p` operation to the model. It computes log(x + 1). This calculates the element-wise logarithm of the input tensor plus one.

Parameter `args`: Vector of input tensor ids.

Parameter `name`: Optional identifier for operation.

Returns The name of the result tensor.

2. log1p(self: popart_core.AiGraphcoreOpset1, args: List[str], debugContext: popart_internal_ir.DebugContext = '') -> str

Add `log1p` operation to the model. It computes log(x + 1). This calculates the element-wise logarithm of the input tensor plus one.

Parameter `args`: Vector of input tensor ids.

Parameter `name`: Optional identifier for operation.

Returns The name of the result tensor.

`lstm`(**args*, ***kwargs*)

Overloaded function.

1. lstm(self: popart_core.AiGraphcoreOpset1, args: List[str], outputFullSequence: int = 1, debugPrefix: popart_internal_ir.DebugContext = '') -> List[str]
2. lstm(self: popart_core.AiGraphcoreOpset1, args: List[str], outputFullSequence: int = 1, debugContext: popart_internal_ir.DebugContext = '') -> List[str]

`multiconv`(**args*, ***kwargs*)

Overloaded function.

1. multiconv(self: popart_core.AiGraphcoreOpset1, args: List[List[str]], dilations: List[List[int]] = [], inDilations: List[List[int]] = [], pads: List[List[int]] = [], outPads: List[List[int]] = [], strides: List[List[int]] = [], availableMemoryProportions: List[float] = [], partialsTypes: List[str] = [], planType: Optional[str] = None, perConvReservedTiles: Optional[int] = None, cycleBackOff: Optional[float] = None, enableConvDithering: List[int] = [], debugPrefix: popart_internal_ir.DebugContext = '') -> List[str]

Add a multi-convolution to the model.

Using this multi-convolution API ensures that the convolutions are executed in parallel on the device.

Functionally, a multi-convolution is equivalent to a series of single convolutions. Using this multi-convolution API is always equivalent to calling the single-convolution API (conv) once for each argument.

For example, calling:

A0 = conv({X0, W0, B0}) A1 = conv({X1, W1})

Is functionally equivalent to calling:

{A0, A1} = multiconv({{X0, W0, B0}, {X1, Q1}).

It is possible that any two convolutions cannot be executed in parallel due to topological constraints. For example, the following:

B = conv({A, W0}); C = B + A D = conv({C, W1});

Cannot be converted to:

{B, D} = multiconv({{A, W0}, {C, W1}}).

Note that it is not possible to create such a cycle by adding a multi- convolution with this API.

Calls to multiconv() are mapped to poplar::poplin::multiconv::convolution().

Parameter `tensors`: List of [DataId, WeightId, BiasId (optional)] for each convolution.

Parameter `dilations`: The dilations attributes for each convolution.

Parameter `inDilations`: The input dilations attributes for each convolution.

Parameter `pads`: The pads for each convolution.

Parameter `outPads`: The output padding for each convolution.

Parameter `strides`: The strides for each convolution.

Parameter `availableMemoryProportions`: The available memory proportions per conv, each [0, 1).

Parameter `partialsTypes`: The partials type per convolution.

Parameter `planType`: Run convolutions in parallel or series.

Parameter `perConvReservedTiles`: Tiles to reserve per convolution when planning.

Parameter `cycleBackOff`: Cycle back-off proportion, [0, 1).

Parameter `enableConvDithering`: Enable convolution dithering per convolution. If true, then convolutions with different parameters will be laid out from different tiles in an effort to improve tile balance in models.

Parameter `debugContext`: Optional debug context.

All input vectors must be either empty, or equal in length to the number of convolutions. Note that groups for each convolution are automatically inferred from the shapes of the data and weight inputs.

Returns The TensorId of the output tensor from each convolution.

2. multiconv(self: popart_core.AiGraphcoreOpset1, args: List[List[str]], dilations: List[List[int]] = [], inDilations: List[List[int]] = [], pads: List[List[int]] = [], outPads: List[List[int]] = [], strides: List[List[int]] = [], availableMemoryProportions: List[float] = [], partialsTypes: List[str] = [], planType: Optional[str] = None, perConvReservedTiles: Optional[int] = None, cycleBackOff: Optional[float] = None, enableConvDithering: List[int] = [], debugContext: popart_internal_ir.DebugContext = '') -> List[str]

Add a multi-convolution to the model.

Using this multi-convolution API ensures that the convolutions are executed in parallel on the device.

Functionally, a multi-convolution is equivalent to a series of single convolutions. Using this multi-convolution API is always equivalent to calling the single-convolution API (conv) once for each argument.

For example, calling:

A0 = conv({X0, W0, B0}) A1 = conv({X1, W1})

Is functionally equivalent to calling:

{A0, A1} = multiconv({{X0, W0, B0}, {X1, Q1}).

It is possible that any two convolutions cannot be executed in parallel due to topological constraints. For example, the following:

B = conv({A, W0}); C = B + A D = conv({C, W1});

Cannot be converted to:

{B, D} = multiconv({{A, W0}, {C, W1}}).

Note that it is not possible to create such a cycle by adding a multi- convolution with this API.

Calls to multiconv() are mapped to poplar::poplin::multiconv::convolution().

Parameter `tensors`: List of [DataId, WeightId, BiasId (optional)] for each convolution.

Parameter `dilations`: The dilations attributes for each convolution.

Parameter `inDilations`: The input dilations attributes for each convolution.

Parameter pads: The pads for each convolution.

Parameter outPads: The output padding for each convolution.

Parameter strides: The strides for each convolution.

Parameter availableMemoryProportions: The available memory proportions per conv, each [0, 1).

Parameter partialsTypes: The partials type per convolution.

Parameter planType: Run convolutions in parallel or series.

Parameter perConvReservedTiles: Tiles to reserve per convolution when planning.

Parameter cycleBackOff: Cycle back-off proportion, [0, 1).

Parameter enableConvDithering: Enable convolution dithering per convolution. If true, then convolutions with different parameters will be laid out from different tiles in an effort to improve tile balance in models.

Parameter debugContext: Optional debug context.

All input vectors must be either empty, or equal in length to the number of convolutions. Note that groups for each convolution are automatically inferred from the shapes of the data and weight inputs.

Returns The TensorId of the output tensor from each convolution.

nllloss(**args*, ***kwargs*)
Overloaded function.

1. nllloss(self: popart_core.AiGraphcoreOpset1, args: List[str], reduction: popart_core.ReductionType = <ReductionType.Mean: 1>, ignoreIndex: Optional[int] = None, inputIsLogProbability: bool = False, debugPrefix: popart_internal_ir.DebugContext = '') -> str

Add a negative log-likelihood loss operation to the model.

Calculates the nll loss given a probability tensor over classes, and a target tensor containing class labels.

Parameter args: Vector of input tensor ids: probability and tensor.

Parameter reduction: Type of reduction to perform on the individual losses.

Parameter ignoreIndex: Optional class index to ignore in loss calculation.

Parameter inputIsLogProbability: Specifies if the input tensor contains log-probabilities or raw probabilities (false, default).

Parameter debugContext: Optional debug context.

Returns The name of the result tensor.

2. nllloss(self: popart_core.AiGraphcoreOpset1, args: List[str], reduction: popart_core.ReductionType = <ReductionType.Mean: 1>, ignoreIndex: Optional[int] = None, inputIsLogProbability: bool = False, debugContext: popart_internal_ir.DebugContext = '') -> str

Add a negative log-likelihood loss operation to the model.

Calculates the nll loss given a probability tensor over classes, and a target tensor containing class labels.

Parameter args: Vector of input tensor ids: probability and tensor.

Parameter reduction: Type of reduction to perform on the individual losses.

Parameter ignoreIndex: Optional class index to ignore in loss calculation.

Parameter inputIsLogProbability: Specifies if the input tensor contains log-probabilities or raw probabilities (false, default).

Parameter debugContext: Optional debug context.

Returns The name of the result tensor.

nop(**args*, ***kwargs*)
Overloaded function.

1. nop(self: popart_core.AiGraphcoreOpset1, args: List[str], debugPrefix: popart_internal_ir.DebugContext = '') -> str

Add a no-op operation to the model.

Parameter args: Vector of input tensor ids.

Parameter debugContext: Optional debug context.

Returns The name of the result tensor.

2. nop(self: popart_core.AiGraphcoreOpset1, args: List[str], debugContext: popart_internal_ir.DebugContext = '') -> str

Add a no-op operation to the model.

Parameter args: Vector of input tensor ids.

Parameter debugContext: Optional debug context.

Returns The name of the result tensor.

packedDataBlock(*self: popart_core.AiGraphcoreOpset1*, *args: List[str]*, *maxSequenceLengths: List[int]*, *resultSize: int*, *callbackBatchSize: int*, *callback: popart::Builder*, *debugPrefix: popart_internal_ir.DebugContext = ''*) → str
Add a call operation to the model

This is a Poplar extension, to expose manual code re-use to the builder.

Parameter args: Vector of input tensor ids.

Parameter callee: The subgraph to call into.

Parameter debugContext: Optional debug context.

Returns A vector of tensors; the subgraph outputs.

printtensor(**args*, ***kwargs*)
Overloaded function.

1. printtensor(self: popart_core.AiGraphcoreOpset1, args: List[str], print_gradient: int = 1, debugPrefix: popart_internal_ir.DebugContext = '', title: str = '') -> str

Add a print tensor operation to the model.

This is a Poplar extension.

Parameter args: Vector of tensor ids to print.

Parameter print_gradient: $Parameter `debugContext`:

Optional debug context.

Parameter title: $Returns:

The name of the result tensor.

2. printtensor(self: popart_core.AiGraphcoreOpset1, args: List[str], print_gradient: int = 1, debugContext: popart_internal_ir.DebugContext = '', title: str = '') -> str

Add a print tensor operation to the model.

This is a Poplar extension.

Parameter args: Vector of tensor ids to print.

Parameter `print_gradient`: $Parameter `debugContext`:

Optional debug context.

Parameter `title`: $Returns:

The name of the result tensor.

`reducemedian`(*self*: popart_core.AiGraphcoreOpset1, *args*: *List[str]*, *axes*: *Optional[List[int]]* = *None*, *keepdims*: *int* = *1*, *debugContext*: *popart_internal_ir.DebugContext* = '') → List[str]

`remainder`(**args*, ***kwargs*)

Overloaded function.

1. remainder(self: popart_core.AiGraphcoreOpset1, args: List[str], debugPrefix: popart_internal_ir.DebugContext = '') -> str

Add remainder operation to the model.

This is equivalent to Python's modulo operator %. The result has the same sign as the divisor.

Parameter `args`: Input tensors.

Returns Computes the element-wise remainder of division. The remainder has the same sign as the divisor.

2. remainder(self: popart_core.AiGraphcoreOpset1, args: List[str], debugContext: popart_internal_ir.DebugContext = '') -> str

Add remainder operation to the model.

This is equivalent to Python's modulo operator %. The result has the same sign as the divisor.

Parameter `args`: Input tensors.

Returns Computes the element-wise remainder of division. The remainder has the same sign as the divisor.

`replicatedallreduce`(**args*, ***kwargs*)

Overloaded function.

1. replicatedallreduce(self: popart_core.AiGraphcoreOpset1, args: List[str], commGroup: Optional[List[int]] = None, debugPrefix: popart_internal_ir.DebugContext = '') -> str

Add a replicated all-reduce operation to the model.

This is a Poplar extension, to expose manual code re-use to the builder.

Parameter `args`: Vector of input tensor ids to reduce across.

Parameter `commGroup`: GCL CommGroup parameter.

Parameter `debugContext`: Optional debug context.

Returns The name of the result tensor.

2. replicatedallreduce(self: popart_core.AiGraphcoreOpset1, args: List[str], commGroup: Optional[List[int]] = None, debugContext: popart_internal_ir.DebugContext = '') -> str

Add a replicated all-reduce operation to the model.

This is a Poplar extension, to expose manual code re-use to the builder.

Parameter `args`: Vector of input tensor ids to reduce across.

Parameter `commGroup`: GCL CommGroup parameter.

Parameter `debugContext`: Optional debug context.

Returns The name of the result tensor.

reshape(**args*, ***kwargs*)
Overloaded function.

1. reshape(self: popart_core.AiGraphcoreOpset1, args: str, shape: List[int], debugPrefix: popart_internal_ir.DebugContext = '') -> str

Add reshape operation to the model. Reshape the input tensor. This reshape takes the shape to reshape into as an attribute instead of a tensor input as the ONNX reshape op.

Parameter arg: Vector with single input tensor id.

Parameter shape: The shape of the output Tensor. The output Tensor must contain the same number of elements as the input Tensor.

Parameter name: Optional identifier for operation.

Returns The name of the result tensor.

2. reshape(self: popart_core.AiGraphcoreOpset1, args: str, shape: List[int], debugContext: popart_internal_ir.DebugContext = '') -> str

Add reshape operation to the model. Reshape the input tensor. This reshape takes the shape to reshape into as an attribute instead of a tensor input as the ONNX reshape op.

Parameter arg: Vector with single input tensor id.

Parameter shape: The shape of the output Tensor. The output Tensor must contain the same number of elements as the input Tensor.

Parameter name: Optional identifier for operation.

Returns The name of the result tensor.

reverse(*self:* popart_core.AiGraphcoreOpset1, *args: List[str]*, *dimensions: List[int]*, *debugContext: popart_internal_ir.DebugContext = ''*) → str
Add a reverse operator to the model.

Reverse, or 'flip', the tensor along the specified dimensions

Parameter args: Input tensors.

Parameter dimensions: Dimensions along which to reverse the tensor. If this is empty then this is equivalent to the identity operator

Returns The name of the result tensor.

round(**args*, ***kwargs*)
Overloaded function.

1. round(self: popart_core.AiGraphcoreOpset1, args: List[str], debugPrefix: popart_internal_ir.DebugContext = '') -> str

Add a Round operation to the model. (This allows Round_11 to be targeted from earlier opsets.)

https://github.com/onnx/onnx/blob/master/docs/Operators.md#Round

Parameter args: Vector of input tensor ids.

Parameter debugContext: Optional debug context.

Returns The normalized output tensor ids.

2. round(self: popart_core.AiGraphcoreOpset1, args: List[str], debugContext: popart_internal_ir.DebugContext = '') -> str

Add a `Round` operation to the model. (This allows `Round_11` to be targeted from earlier opsets.)

https://github.com/onnx/onnx/blob/master/docs/Operators.md#Round

Parameter `args`: Vector of input tensor ids.

Parameter `debugContext`: Optional debug context.

Returns The normalized output tensor ids.

`scale`(**args*, ***kwargs*)

Overloaded function.

1. scale(self: popart_core.AiGraphcoreOpset1, args: List[str], scale: float, debugPrefix: popart_internal_ir.DebugContext = '') -> str

Add a scale operation to the model.

This is a Poplar extension.

Parameter `args`: Vector of input tensor ids.

Parameter `scale`: The scale to apply.

Parameter `debugContext`: Optional debug context.

Returns The name of the result tensor.

2. scale(self: popart_core.AiGraphcoreOpset1, args: List[str], scale: float, debugContext: popart_internal_ir.DebugContext = '') -> str

Add a scale operation to the model.

This is a Poplar extension.

Parameter `args`: Vector of input tensor ids.

Parameter `scale`: The scale to apply.

Parameter `debugContext`: Optional debug context.

Returns The name of the result tensor.

`scaledadd`(**args*, ***kwargs*)

Overloaded function.

1. scaledadd(self: popart_core.AiGraphcoreOpset1, args: List[str], scale0: float = 1.0, scale1: float = 1.0, debugPrefix: popart_internal_ir.DebugContext = '') -> str

Add a scaled add operation to the model.

X = scale0 * T0 + scale1 * T1

Parameter `args`: Vector of input tensor ids: [T0, T1, scale0, scale1].

Parameter `scale0`: The scale to apply (if no `scale0` tensor is supplied).

Parameter `scale1`: The scale to apply (if no `scale1` tensor is supplied).

Parameter `debugContext`: Optional debug context.

Returns The name of the result tensor.

2. scaledadd(self: popart_core.AiGraphcoreOpset1, args: List[str], scale0: float = 1.0, scale1: float = 1.0, debugContext: popart_internal_ir.DebugContext = '') -> str

Add a scaled add operation to the model.

X = scale0 * T0 + scale1 * T1

Parameter `args`: Vector of input tensor ids: [T0, T1, scale0, scale1].

Parameter `scale0`: The scale to apply (if no `scale0` tensor is supplied).

Parameter `scale1`: The scale to apply (if no `scale1` tensor is supplied).

Parameter `debugContext`: Optional debug context.

Returns The name of the result tensor.

`scatterreduce`(*self: popart_core.AiGraphcoreOpset1*, *args: List[str] = []*, *axis_size: int*, *axis: int = -1*, *reduction: popart_core.ScatterReduction = <ScatterReduction.Sum: 0>*, *debugContext: popart_internal_ir.DebugContext = ''*) → str

`sequenceslice`(*self:* popart_core.AiGraphcoreOpset1, *args: List[str]*, *zeroUnused: int = 0*, *debugContext: popart_internal_ir.DebugContext = ''*) → str

Slice a 2D tensor based on offsets specified by a tensor.

The outermost dimension is sliced; tOut[tOutOffset:tOutOffset+tN][...] = tIn[tInOffset:tInOffset+tN][...] for each entry in tN/tInOffset/tOutOffset; entries after the first tN==0 may be ignored. Unreferenced elements of tOut are zeroed if zeroUnused is set. The same output element should not be written by multiple inputs.

tIn and tOut must have rank greater than or equal to 2. The outer dimension is sliced; the product of the inner dimensions must match. tInOffset, tOutOffset and tN must be 1d and the same size. param [source, destination, N, sourceOffset, destinationOffset]

Parameter `zeroUnused`: Whether to zero unreferenced tOut elements.

Parameter `debugContext`: Optional debug context.

`shapeddropout`(**args*, ***kwargs*)

Overloaded function.

1. shapeddropout(self: popart_core.AiGraphcoreOpset1, args: List[str], shape: List[int], ratio: float = 0.5, debugPrefix: popart_internal_ir.DebugContext = '') -> str

   Add a shaped dropout operation to the model.

   Applies a shaped dropout to the input tensor. This operator requires a shape parameter that is used to define the shape of the dropout mask so that strongly correlated features in the input tensor can be preserved. The provided shape must be broadcastable to the input tensor. Note that this operation targets the poprand library function of the same name.

   Parameter `args`: Vector of input tensor ids.

   Parameter `shape`: Shape of dropout mask. Must be broadcastable to the input.

   Parameter `ratio`: Probability of dropping an input feature (default = 0.5).

   Parameter `name`: Optional identifier for operation.

   Returns The name of the result tensor.

2. shapeddropout(self: popart_core.AiGraphcoreOpset1, args: List[str], shape: List[int], ratio: float = 0.5, debugContext: popart_internal_ir.DebugContext = '') -> str

   Add a shaped dropout operation to the model.

   Applies a shaped dropout to the input tensor. This operator requires a shape parameter that is used to define the shape of the dropout mask so that strongly correlated features in the input tensor can be preserved. The provided shape must be broadcastable to the input tensor. Note that this operation targets the poprand library function of the same name.

Parameter `args`: Vector of input tensor ids.

Parameter `shape`: Shape of dropout mask. Must be broadcastable to the input.

Parameter `ratio`: Probability of dropping an input feature (default = 0.5).

Parameter `name`: Optional identifier for operation.

Returns The name of the result tensor.

`subsample`(**args*, ***kwargs*)
Overloaded function.

1. subsample(self: popart_core.AiGraphcoreOpset1, args: List[str], strides: List[int], debugPrefix: popart_internal_ir.DebugContext = '') -> str

Add a sub-sample operation to the model.

This is a Poplar extension.

If multiple tensors are provided that strides will applied to them all.

Parameter `args`: Vector of tensor ids to sub-sample.

Parameter `strides`: The strides to use.

Parameter `debugContext`: Optional debug context.

Returns The name of the result tensor.

2. subsample(self: popart_core.AiGraphcoreOpset1, args: List[str], strides: List[int], debugContext: popart_internal_ir.DebugContext = '') -> str

Add a sub-sample operation to the model.

This is a Poplar extension.

If multiple tensors are provided that strides will applied to them all.

Parameter `args`: Vector of tensor ids to sub-sample.

Parameter `strides`: The strides to use.

Parameter `debugContext`: Optional debug context.

Returns The name of the result tensor.

`swish`(*self:* popart_core.AiGraphcoreOpset1, *args: List[str] = []*, *debugContext: popart_internal_ir.DebugContext = ''*) → str
Add a swish operation to the model.

The operation computes the swish activation function, also known as the SiLU activation.

Parameter `args`: Vector with single input tensor id.

Returns The name of the result tensor.

2.6 Patterns

class popart.**Patterns**

enablePattern(*self: popart_core.Patterns*, *arg0: str*, *arg1: bool*) → popart_core.Patterns

enableRuntimeAsserts(*self: popart_core.Patterns*, *arg0: bool*) → popart_core.Patterns

isPatternEnabled(*self: popart_core.Patterns*, *arg0: str*) → bool

2.7 Session Options

class popart.**AccumulateOuterFragmentSettings**

property excludedVirtualGraphs
A setting to explicitly tell PopART to avoid to try and parallelise the given virtual graph ids. This setting is experimental and may change.

property schedule
Tell PopART how you would like to schedule the accumulate outer fragment. This setting is experimental and may change.

class popart.**AccumulateOuterFragmentSchedule**
Enum type that determines how the operations in the accumulate outer fragment will be scheduled accross virtual graphs (only relevant to pipelined modes).

Members:

Scheduler : Don't add additional constraints and let the scheduler work it out.

Serial : Add constraints that ensure ops are executed in virtual graph ID

order.

OverlapCycleOptimized : Try and parallelise ops with different virtual graph IDs as much as

possible.

OverlapMemoryOptimized : Try and parallelise ops with different virtual graph IDs but avoid

certain steps that are costly in terms of memory usage.

class popart.**AutodiffSettings**

class popart.**AutodiffStitchStrategy**
Members:

RecomputeMinimal

RecomputeAllNonInputs

AddFwdOutputs

SafeAddFwdOutputs

class popart.**BatchSerializationSettings**
A structure containing batch serialization settings.

property batchSchedule
Experimental value that changes how operations are scheduled.

property concatOnExecutionPhaseChange
Break batch serialization chains when the execution phase changes (by concatenating the compute batches to the local batch).

property concatOnPipelineStageChange
Break batch serialization chains when the pipeline stage changes (by concatenating the compute batches to the local batch).

property concatOnVirtualGraphChange
Break batch serialization chains when the virtual graph changes (by concatenating the compute batches to the local batch).

property factor
The number of compute batches to split operations into.

property method
Experimental value to control how batch serialization is applied.

property transformContext
Experimental value to control when batch serialization is applied.

class popart.**BatchSerializationBatchSchedule**
Enum type that describes how to change the batch serialisation subgraph schedule before outlining. **NOTE:** This setting is experimental and may change.

Members:

Scheduler : Don't encourage any particular scheduling for ops within batch

subgraphs (leave it to the scheduler) but tell the scheduler to schedule subgraphs in sequence.

Isomorphic : Encourage all ops within batch subgraphs to be scheduled identically

and for each subgraph to be scheduled in sequence (good for outlineability).

OverlapOnIo : Attempt to put the RemoteLoad for batch N+1 right after the compute

phase of batch N.

OverlapOnCompute : Attempt to put the RemoteLoad for batch N+1 right before the compute

phase of batch N.

class popart.**DotCheck**
Enum type used to identify at which stages of IR construction to export `.dot` files.

Members:

Fwd0 : Generate graph after construction of the forward pass.

Fwd1 : Generate graph after running pre-aliasing patterns.

Bwd0 : Generate graph after backwards construction.

PreAlias : Generate graph after all transformations, patterns, except the

aliasing.

Final : Generate graph after running aliasing patterns (the final IR).

property name

class popart.**ExecutionPhaseSchedule**
Enum type to specify the order of processing optimizer operations for different weights of the same execution phase.

The steps for phased execution consists of: - Copy to IO tiles if necessary (1) - Run collective operations if necessary (2) - Load optimizer state (3) - Update optimizer state (4) - Apply optimizer (5) - Store updated tensor if necessary (6)

Members:

Interleaving : Process above steps for one weight at a time (for example: 123456,

123456, 123456). The scheduler may interleave these steps.

Batch : Process above steps for all weights together, in a way that maximises

overlap potential between compute and exchange (for example: 333, 111, 222, 444, 555, 666).

BatchClusteredIO : Process above steps for all weights together, in a way that maximises

overlap potential between compute and exchange, and maximise stream copy merges by keeping RemoteLoad/RemoteStore operations clustered (for example: 333, 111, 222, 444, 555, 666).

class popart.**TensorLocationSettings**

property location
The default tensor location for this tensor type.

property minElementsForOffChip
A minimum number of elements below which offloading won't be considered.

property minElementsForReplicatedTensorSharding
A minimum number of elements below which replicated tensor sharding (RTS) won't be considered.

class popart.**ReplicatedTensorSharding**
Enum type to specify whether to shard tensors over replicas.

Members:

Off : Don't shard tensors over replicas.

On : Do shard tensors over replicas.

class popart.**TensorLocation**

class popart.**TensorStorage**
Enum type that determines where a tensor is stored.

Members:

OnChip : Store the tensor in on-chip memory.

OffChip : Store the tensor in streaming memory.

class popart.**TileSet**
Enum type to specify a set of tiles.

Members:

Compute : The set of tiles designated for compute operations.

IO : The set of tiles designated for IO operations.

class popart.**Instrumentation**
Members:

Outer : Outer loop instrumentation, graph over all IPUs.

Inner : Inner loop instrumentation, graph per IPU.

class popart.**MergeVarUpdateType**
Enum type used to specify which `VarUpdateOp` ops to merge.

Members:

Off : Do not merge VarUpdateOp ops.

All : Merge all VarUpdateOp ops into as few groups as possible. This is a

good choice when memory is not a constraint.

AutoTight : Merge into groups, so that VarUpdateOp ops process tensors of exactly

`mergeVarUpdateMemThreshold` in size.

AutoLoose : Merge into groups while attempting not to increase maximum variable

liveness, and also not slice tensor variables so they they will need to be processed by different VarUpdateOp ops.

class popart.**SyntheticDataMode**
Members:

Off : Use real data.

Zeros : Input tensors are initialised to all zeros.

RandomNormal : Input tensors are initialised with distribution ~N(0,1).

class popart.**RecomputationType**
Enum type to specify which ops to recompute in the backwards pass when doing auto-recomputation.

Members:

NoRecompute : No ops are recomputed.

Standard : Algorithm to pick checkpoints to try and minimise max liveness.

NormOnly : Only Norm ops (+ non-linearities, if following) are recomputed.

RecomputeAll : Every ops are recomputed.

Pipeline : Recompute all forward pipeline stages.

class popart.**VirtualGraphMode**
Members:

Off : Virtual graphs are not enabled.

Manual : User must set the `virtualGraph` attribute on all ops.

Auto : Use `autoVirtualGraph` transform.

ExecutionPhases : Virtual graphs are tied to execution phases.

class popart.**SubgraphCopyingStrategy**
Members:

OnEnterAndExit : Copy all inputs before the start of the subgraph, copy all outputs

after all ops in the subgraph. With this strategy subgraphs will always map to a single Poplar function.

JustInTime : Copy inputs just before they are consumed and copy outputs as soon as

they are produced. With this strategy subgraphs may be lowered into multiple Poplar functions.

class popart.**ExecutionPhaseSettings**

property activationIOSchedule
The execution phase IO schedule for activation and gradient tensors.

property phases
Number of ExecutionPhases for the whole model

property stages
Parallel streaming memory, default for 1 IPU / replica 2: PingPong between 2 IPUs, default for >= 2 IPUs / replica

Type Number of overlapping stages 1

property weightIOSchedule
The execution phase IO schedule for weight tensors.

class popart_core.**SessionOptions**

property accumulateOuterFragmentSettings
Configuration setting for operations in the accumulate outer fragment.

property accumulationAndReplicationReductionType
Specify how gradients are reduced when using gradient accumulation and graph replication.

property accumulationFactor
Specify the number of micro-batches to accumulate before applying the varUpdate.

property accumulatorTensorLocationSettings
Tensor location for gradient accumulator tensors.

property activationTensorLocationSettings
Tensor location settings for activation/gradient tensors.

property aliasZeroCopy
Enable zero-copy for subgraphs.

property autoRecomputation
Enable recomputation of operations in the graph in the backwards pass to reduce model size at the cost of computation cycles.

property batchSerializationSettings
Configuration setting for batch serialization.

property cachePath
Folder to save the `poplar::Executable` to.

property compileEngine
If false, the backend will build the Poplar graph but not compile it into an Engine. In this case, no execution can be performed, and nothing can be transferred to the device. API calls which retrieve information from the graph building stage, such as tile mapping introspection, can still be used.

property constantWeights
An optimization for an inference session to have constant weights, true by default. Set this option to false if you are going to want to change the weights with a call to Session::resetHostWeights after the session has been prepared. This option has no effect on a training session

property customCodeletCompileFlags
Compile flags for the custom codelets. For example `-g` to generate debug info.

property customCodelets
List of codelets (with filetype) to be added to the Poplar graph. See the Poplar documentation for more information.

property decomposeGradSum
Replaces single sums of partial gradients with a tree of additions. This can reduce max liveness at the cost of extra cycles. A typical use case for this would be if a large weight tensor is used as an input to many operations.

property delayVarUpdates
Options to delay variable updates as much as possible.

property disableGradAccumulationTensorStreams
If true, the weight gradient tensors are not saved off the device when `devicex`.weightsFromHost() is called. Note: this option is overridden if #syntheticDataMode is not #SyntheticDataMode::Off.

property dotChecks
When to write `.dot` files during Ir construction.

property dotOpNames
Include the Op name in the `.dot` file (the Op type is always exported).

property enableDistributedReplicatedGraphs
Enable training with Poplar replicated graphs across multiple PopART instances.

property enableEngineCaching
Enable Poplar executable caching.

property enableExplicitMainLoops
Enables explicit main loop transformation, and disables implicit training loops. This will become deprecated and enabled by default.

property enableFloatingPointChecks
Throw an exception when floating point errors occur.

property enableFullyConnectedPass
Enable the global #fullyConnectedPass option for matmuls.

property enableGradientAccumulation
Enable gradient accumulation.

property enableLoadAndOffloadRNGState
Allows to load/offload device RNG state from host.

property enableMergeExchange
Enables merging remote and host IO operations to facilitate IO overlap

property enableNonStableSoftmax
By default, we use the stable softmax Poplar function. The input tensor to softmax, _x_, is preprocessed by subtracting max(_x_) from each element before computing the exponentials, ensuring numerical stability. If you are sure the inputs to your softmax operations are small enough to not cause overflow when computing the exponential, you can enable the non-stable version instead, to increase the speed.

property enableOutlining
Identify and extract repeated parts of computational graph into subgraphs.

property enableOutliningCopyCostPruning
When `true` the cost of copying of cached sections should be included in the outlining cost model.

property enablePipelining
Enable pipelining of virtual graphs

property enableReplicatedGraphs
Enable replication of graphs.

property enableStableNorm
If true, computes the mean first and subtracts the activations from it before computing the variance. The implementation with this flag set to true is slower than when set to false. The stable version requires the first order moment to be estimated and applied to the sample set before the second order central moment is calculated.

property enableStochasticRounding
Enable stochastic rounding.

property enableSupportedDataTypeCasting
If enabled, casts any tensor of unsupported data types to supported data types when lowering to Poplar Currently, this implies casting: INT64 -> INT32 UINT64 -> UINT32 The cast will error for incompatible data types and over/underflows, and inform on narrowing casts

property executionPhaseSettings
Configuration settings for execution phases.

property explicitRecomputation
Enable explicit recomputation.

property exportPoplarComputationGraph
Export Poplar computation graph.

property exportPoplarVertexGraph
Export Poplar vertex graph.

property finalDotOp
See #firstDotOp.

property firstDotOp
The ops to write to the `.dot` file will be a continuous interval of the schedule, controlled by firstDotOp and finalDotOp. In particular, it will be [min(0, firstDotOp), max(N ops in Ir, finalDotOp)).

property globalReplicaOffset
The first replica index that this PopART instance is running.

property globalReplicationFactor
The total number of replicas in a multi instance replicated graph training session (this should be left as the default value (1) if distributed replicated graphs are disabled). This value includes local replication.

property groupHostSync
Allows to group the streams from host at the beginning and the streams to host at the end, this trades off sum-liveness efficiency for cycle efficiency.

property instrumentWithHardwareCycleCounter
Add instrumentation to your program to count the number of device cycles (of a single tile, on a single IPU) that your main program takes to execute. Expect this to have a small detrimental impact on performance.

property kahnTieBreaker
The initial scheduling is done with Kahn's algorithm. When several Ops are free to be scheduled, this controls which method is used.

property logDir
A directory for log traces to be written into.

property meanAccumulationAndReplicationReductionStrategy
Specify when to divide by a mean reduction factor when accumulationAndReplicationReductionType is set to ReductionType::Mean.

property mergeVarUpdate
Enable merging of VarUpdates into groups of VarUpdates, by flattening and concatenating variable tensors and updating tensors.

property mergeVarUpdateMemThreshold
The #MergeVarUpdateType::AutoLoose and #MergeVarUpdateType::AutoTight VarUpdateOp merging algorithms have a threshold on the total memory of variable tensors to merge for updating. Defined as total memory in bytes.

property optimizerStateTensorLocationSettings
Tensor location for optimizer state tensors.

property opxAliasChecking
Run Opx checks to verify IR tensor aliasing information corresponds to lowered Poplar tensor aliasing.

property opxModifyChecking
Run Opx checks to verify IR tensor modification information corresponds to lowered Poplar tensor modifications.

property outlineSequenceBreakCost
The penalty applied to outlining potential sub-graphs if the sub-graph to be created breaks up a sequence of operations that are more efficient (for example for overlapping compute and exchange) when outlined together. Default value is set to ~10 * Op::getHighSubgraphValue().

property outlineThreshold
The incremental value that a sub-graph requires, relative to its nested sub-graphs (if any), to be eligible for outlining. A high threshold results in fewer sub-graphs being outlined, a negative value results in all being outlined. The gross value of a sub-graph is the sum of its constituent Ops' Op::getSubgraphValue() values. To disable outlining, it is better to set enableOutlining to false than to set this value to infinity. The default value of 1.0f results in all high value operations such as convolution being cached, but standalone low Value operations such as Relu will not be.

property partialsTypeMatMuls
Set the partials type globally for matmuls. Can be overridden individually with Builder.setPartialsType().

Valid values are `"float"` and `"half"`. By default, this is not set, so no global partials type is imposed.

property rearrangeAnchorsOnHost
Before anchor tensors are streamed from device to host, they are not necessarily arranged in memory as required when they are to be copied from host stream to host. This can be done on the device or on the host. Done on host by default to save memory, but often at the expense of cycles, especially for larger anchor tensors.

property replicatedGraphCount
If enableReplicatedGraphs is true, `replicatedGraphCount` will set the number of model replications. For example, if your model uses 1 IPU, a `replicatedGraphCount` of 2 will use 2 IPUs. If your model is pipelined across 4 IPUs, a `replicatedGraphCount` of 4 will use 16 IPUs total. Therefore, the number of IPUs you request must be a multiple of `replicatedGraphCount`. If the training is done across multiple instances then the `replicatedGraphCount` is the number of replicas for this instance.

property scheduleNonWeightUpdateGradientConsumersEarly
When #shouldDelayVarUpdates is true, the other ops in the proximity of the delayed var updates may inherit the -inf schedule priority used to delay the var updates. This is undesirable for some ops that consume gradients, as we would like to consume (and thus be able to recycle the memory of) those gradients as soon as possible. Two examples are HistogramOps when doing automatic loss scaling, and the AccumulateOps that accumulate the gradients when doing gradient accumulation.

If true, if #shouldDelayVarUpdates is true, this option will cause the schedule priority of the above described ops to be re-overriden to +inf.

property separateCallOpPdfs
When generating PDFs of IR graphs, create separate PDFs for each subgraph.

property serializedPoprithmsAnnealGraphsDir
PopART uses Poprithms for scheduling PopART graphs. The Poprithms graphs created for scheduling can be optionally serialised (written to file). The string below specified the directory to serialize Poprithms graphs to. If it is empty, then the graphs will not be serialised. The names of serialization files will be `poprithms_shift_graph_i.json` for the lowest non-existing values of `i`. The directory must already exist, PopART will not create it.

property serializedPoprithmsShiftGraphsDir
PopART uses Poprithms for scheduling PopART graphs. The Poprithms graphs created for scheduling can be optionally serialised (written to file). The string below specified the directory to serialize Poprithms graphs to. If it is empty, then the graphs will not be serialised. The names of serialization files will be `poprithms_shift_graph_i.json` for the lowest non-existing values of `i`. The directory must already exist, PopART will not create it.

property strictOpVersions
Strict op version checks will throw an error if the exact version of an op required for the models opset is not supported. Turning this check off will cause PopART to fall back to the latest implementation of the op that is supported. Warning, turning off these checks may cause undefined behaviour.

property subgraphCopyingStrategy
This setting determines how copies for inputs and outputs for subgraphs are lowered. By setting this value to JustInTime you may save memory at the cost of fragmenting subgraphs into multiple Poplar functions. This may be particularly useful when a number of weight updates are outlined in one subgraph, as it may prevent multiple weight tensors from being live at the same time inside the subgraph.

property swapLimitScheduler
The maximum number of improving steps allowed by the scheduling algorithm before a solution must be returned.

property syntheticDataMode
disable data transfer to/from the host. Set to #SyntheticDataMode::Off to use real data.

Type Use synthetic data

property timeLimitScheduler
The maximum allowed time that can be spent searching for a good graph schedule before a solution

must be returned.

property weightTensorLocationSettings
Tensor location for weight tensors.

2.8 Optimizers

class popart_core.**Optimizer**

getLossScalingVal(*self:* popart_core.Optimizer) → float

2.8.1 SGD

class popart_core.**SGD**
Stochastic Gradient Descent (%SGD) optimizer.

Akin to any optimizer implementation, this class is responsible for updating each weight tensor (w) in the model using the gradient (g) of the loss function with respect to the weight as calculated during the backwards pass.

The %SGD optimizer has the following **state** for each weight:

- *velocity* (v)

The %SGD optimizer has the following **hyper parameters**:

- *learning rate* (lr) * *momentum* (mm) * *weight

decay* (wd) * *dampening* (dm) * *velocity scaling* (vs) * *loss scaling* (ls) * *clip norm settings*

The values of these parameters can be shared between all weights but some can be overridden with weight-specific values (see SGD::insertSpecific). Hyper parameters are captured using OptimizerValue objects and therefore can be either a constant value or a non-constant value that can be adjusted by the user.

In the following we will describe how this optimizer updates a weight using a gradient. In the context of this description the gradient is is the value of the gradient *after* any gradient accumulation has been performed and *after* the application of a loss scaling factor to the gradient has been corrected for.

When the optimizer needs to update a weight, w, using a gradient, g, it first updates the optimizer state as follows:

f[v' := v * \text{mm} + (1 - \text{dm}) * (g + \text{wd} * w) \text{ . } f]

Following the update of the optimizer state the optimizer uses said state to update the weight:

f[w' := w - \text{lr} * v' \text{ . } f]

In addition to the above, the *velocity scaling* hyper parameter is a scaling factor that can provide improved numerical stability by ensuring the values stored in the optimizer state, v, are scaled by this value. When using this parameter PopART will automatically deal with the artificially scaled velocity value during the weight update and other hyper parameters do not need to be adjusted).

In addition, the *loss scaling* hyper parameter is similar in nature to the velocity scaling parameter. It is a scaling value that is applied to the loss gradient at the start of the the backwards pass and, at the end of the backwards pass, this scaling is reversed by multiplying the gradients for each weight with the inverse of the loss scaling value prior to updating the optimizer state. Using loss scaling can also improve numerical stability in some cases.

Finally, it is possible to add clip norm settings for this optimizer. These clip norms compute the L2 norm for a group of weights and adds a scalar term to the weight update that effectively divides it by the norm (or a constant value that is provided as part of the clip norm, which ever is greater).

See the SGD notes in optimizer.hpp for a more detailed and comprehensive derivation of the SGD optimizer step in PopART.

dampenings(*self*: popart_core.SGD) → popart_core.OptimizerValueMap

insertSpecific(*self*: popart_core.SGD, *arg0*: *str*, *arg1*: *dict*) → None

learningRates(*self*: popart_core.SGD) → popart_core.OptimizerValueMap

momentums(*self*: popart_core.SGD) → popart_core.OptimizerValueMap

velocityScalings(*self*: popart_core.SGD) → popart_core.OptimizerValueMap

weightDecays(*self*: popart_core.SGD) → popart_core.OptimizerValueMap

2.8.2 ConstSGD

class popart_core.**ConstSGD**

Stochastic Gradient Descent (SGD) optimizer with constant learning rate, weight decay, loss scaling and clip norm settings (and default values for momentum, dampening or velocity scaling).

NOTE: See SGD for detailed meaning for these parameters.

NOTE: This class exists for backwards compatibility with the Python API and may be removed at some point in the future.

2.8.3 Adam

class popart_core.**Adam**

AdamW, Lamb and AdaMax optimizer implementation.

Akin to any optimizer implementation, this class is responsible for updating each weight tensor (w) in the model using the gradient (g) of the loss function with respect to the weight as calculated during the backwards pass.

The optimizer has the following **state** for each weight:

- *first-order momentum* (m) * *second-order momentum* (v) * *time

step* (t)

The optimizer has the following **hyper parameters**:

- *learning rate* (lr) * *weight decay* (wd) *

beta1 (β_1) * *beta2* (β_2) * *epsilon* (ϵ) * *loss scaling* (ls) * *maximum weight norm* (mwn)

The values of these parameters can be shared between all weights but some can be overridden with weight-specific values (see Adam::insertSpecific). Hyper parameters are captured using OptimizerValue objects and therefore can be either a constant value or a non-constant value that can be adjusted by the user.

The values of #AdamMode and #WeightDecayMode passed to the constructor determines how weights are updated (see below).

In the following we will describe how this optimizer updates a weight using a gradient. In the context of this description the gradient is is the value of the gradient *after* any gradient accumulation has been performed and *after* the application of a loss scaling factor to the gradient has been corrected for.

When the optimizer needs to update a weight, w, using a gradient, g, it first computes a term g_tmp, which is effectively is g with L2 regularization applied if the #WeightDecayMode is set to WeightDecayMode::L2Regularization this, as follows:

f[g_\text{tmp} := \left\{\begin{aligned} g & \text{ ; (Decay) } \\ (g + \text{wd} * w) & \text{ ; (L2Regularization) ; . } \\ \end{aligned}\right.\ f]

Secondly, the optimizer updates the optimizer state as follows:

f[m' &:= beta_1 * m + (1 - beta_1) * g_text{tmp} \ v' &:= left{begin{aligned} beta_2 * v + (1 - beta_2) * g_text{tmp}^2 & text{ ; (Adam/AdamNoBias) } \ beta_2 * v + (1 - beta_2) * g_text{tmp}^2 & text{ ; (Lamb/LambNoBias) } \ text{max}(beta_2 * v, |g_\text{tmp}|) & text{ ; (AdaMax) } \ end{aligned}right.\ t' &:= t + 1 \ f]

Next, it computes the following terms:

f[m_text{tmp} &:= left{begin{aligned} m' & text{ ; (AdamNoBias/LambNoBias) } \ frac{m'}{(1 - beta_1^{t'})} & text{ ; (Adam/Lamb/AdaMax) } \ end{aligned}right.\ v_text{tmp} &:= left{begin{aligned} v' & text{ ; (AdamNoBias/LambNoBias) } \ frac{v'}{(1 - beta_2^{t'})} & text{ ; (Adam/Lamb/AdaMax) } \ end{aligned}right.\ u_text{tmp} &:= left{begin{aligned} frac{m_text{tmp}}{(sqrt{v_text{tmp}} + epsilon)} + text{wd} * w &text{ ; (Decay) } \ frac{m_text{tmp}}{(sqrt{v_text{tmp}} + epsilon)} &text{ ; (L2Regularization) } \ end{aligned}right. f]

Finally, the optimizer updates the weight as follows:

f[w' := left{begin{aligned} w - text{lr} * u_text{tmp} &text{ ; (Adam/AdamNoBias/AdaMax) } \ w - biggl(frac{text{min}(lVert{w}rVert, text{mwn})}{lVert{u_text{tmp}}rVert}biggr) * text{lr} * u_text{tmp} &text{ ; (Lamb/LambNoBias) } \ end{aligned}right. f]

In addition to the above, the *loss scaling* hyper parameter is similar in nature to the velocity scaling parameter. It is a scaling value that is applied to the loss gradient at the start of the the backwards pass and, at the end of the backwards pass, this scaling is reversed by multiplying the gradients for each weight with the inverse of the loss scaling value prior to updating the optimizer state. Using loss scaling can also improve numerical stability of the gradient calculations. If scaledOptimizerState is enabled then the the lossScaling will not be removed before updating the optimizer state. This can improve the numerical stability when accl1_type is set to FLOAT16.

NOTE: The maximum weight norm is referred to as phi in [You et al., 2020](https://arxiv.org/abs/1904.00962).

beta1s(*self*: popart_core.Adam) → popart_core.OptimizerValueMap

beta2s(*self*: popart_core.Adam) → popart_core.OptimizerValueMap

epss(*self*: popart_core.Adam) → popart_core.OptimizerValueMap

insertSpecific(*self*: popart_core.Adam, *arg0*: *str*, *arg1*: *dict*) → None

learningRates(*self*: popart_core.Adam) → popart_core.OptimizerValueMap

maxWeightNorms(*self*: popart_core.Adam) → popart_core.OptimizerValueMap

weightDecays(*self*: popart_core.Adam) → popart_core.OptimizerValueMap

2.9 `popart.ir` (experimental)

Warning: This Python module is currently experimental and may be subject to change in future releases in ways that are backwards incompatible without deprecation warnings.

The `popart.ir` module is an experimental PopART python module through which it is possible to create (and to a limited degree manipulate) PopART IRs directly.

class popart.ir.**Constant**

Wraps a Tensor in the PopART IR that has TensorType.Constant

copy_to_ipu(*dst*, *src*)

Returns ops.ipu_copy(self, dst, src). Must provide a src value.

Parameters

- **dst** (int) –
- **src** (int) –

Return type popart.ir.tensor.Tensor

class popart.ir.**DeviceToHostStream**(*tensor*)
A device-to-host stream in the Ir.

Can be created in the main graph of the Ir only.

You can pass a DeviceToHostStream and a Tensor to ops.host_store in any subgraph(s) any number of times, and in all cases PopART will stream the value of the provided tensor down the provided stream.

class popart.ir.**HostToDeviceStream**(*tensor*)
A host-to-device stream in the Ir.

Can be created in the main graph of the Ir only.

You can pass a HostToDeviceStream and a Tensor to ops.host_load in any subgraph(s) any number of times, and in all cases PopART will stream the next value into the provided tensor.

class popart.ir.**Ir**
Class that represents the PopART IR.

This class contains a main graph. Furthermore, it defines methods and decorators for creating additional graphs from Python functions.

get_graph(*fn*, **args*, ***kwargs*)
Create a graph from a Python function.

Parameters

- **fn** (Callable[.., Any]) – The Python function that defines the graph.
- ***args** (Any) – Arguments passed to the Python function that defines the graph.
- ****kwargs** (Any) – Keyword arguments passed to the Python function that defines the graph.
- **args** (Any) –
- **kwargs** (Any) –

Returns A graph that corresponds to the input Python function.

Return type Graph

main_graph()
Every IR is initialised with a main graph. This method returns this graph.

Returns The main graph of the IR.

Return type Graph

class popart.ir.**Mapping**(**args*, ***kwds*)

class popart.ir.**Module**
Callable class from which user-defined layers can inherit.

The #build method should be overriden and should build the subgraph.

The benefit of inheriting from this class rather than passing a function is that you can save input tensors as fields on self, then later when you call the subgraph, you can pass a mapping from the input tensor ids to the corresponding parent tensor you wish to pass.

class popart.ir.**Variable**
Wraps a Tensor in the PopART IR that has TensorType.Variable

copy_to_ipu(*dst*, *src*)
Returns ops.ipu_copy(self, dst, src). Must provide a src value.

Parameters

- **dst** (int) –

- **src** (int) –

Return type popart.ir.tensor.Tensor

popart.ir.**constant**(*data*, *dtype=popart.ir.dtypes.float32*, *name=None*)

A constant tensor that is initialised with data during graph creation.

This tensor cannot change during the runtime of a model. The inteded use of this class is when doing operations between `popart.ir.Tensor` instances and other types, such as `numpy.ndarray` objects, numbers, or list or tuples of numbers.

Example

```
>>> import popart.ir as pir
>>> with pir.Ir().main_graph():
>>>     a = pir.variable(0)
>>>     # The `1` will be implicitly converted to a `Constant`.
>>>     b = a + 1
```

Parameters

- **data** (`np.array, or a value numpy can use to construct an np.ndarray`) – The data used to initialise the tensor.
- **dtype** (`dtype`) – The data type of the tensor. Defaults to `pir.float32`.
- **name** (`Optional[str]`) – The name of the tensor. Defaults to `None`.

Return type popart.ir.tensor.Constant

popart.ir.**gcg**()

Get the graph that is at the top of the global graph stack.

Raises `RuntimeError` – If the stack is empty.

Returns The graph at the top of the global graph stack.

Return type Graph

popart.ir.**get_current_graph**()

Get the graph that is at the top of the global graph stack.

Raises `RuntimeError` – If the stack is empty.

Returns The graph at the top of the global graph stack.

Return type Graph

popart.ir.**in_sequence**(*enabled=True*)

Force Ops created in this context to executing in the order that they are created.

Parameters **enabled** (`bool`) –

popart.ir.**subgraph_input**(*dtype*, *shape*, *unscoped_name*)

Create a new input tensor to the current graph.

You can use this function when defining a subgraph to create a new input tensor. When you call that subgraph, you will have to pass a tensor to the subgraph for this input.

Example

```
>>> import popart.ir as pir
>>>
>>> def add_w(x):
>>>     w = pir.subgraph_input(x.dtype, x.shape, "w")
>>>     return w + x
>>>
>>> ir = pir.Ir()
>>> with ir.main_graph():
>>>     w = pir.variable(1)
>>>     x = pir.variable(3)
>>>     add_w_graph = ir.get_graph(add_w, x, w)
>>>     y = ops.call(add_w_graph, x, w)
```

Parameters

- **data** (`np.array, or a value numpy can use to construct an np.ndarray`) – The data used to initialise the tensor.
- **dtype** (`dtype`) – The data type of the tensor. Defaults to `pir.float32`.
- **unscoped_name** (`str`) – The name of the tensor. Note this should not include the Graph scope.
- **shape** (`Tuple[int]`) –

Return type popart.ir.tensor.Tensor

popart.ir.**subgraph_output**(*t*)

Mark a tensor as an output in the current graph.

You can use this function when defining a subgraph to mark an existing tensor in the subgraph as an output. When you call that subgraph, it will return that tensor in the parent graph.

Example

```
>>> import popart.ir as pir
>>>
>>> def add_w(x):
>>>     w = pir.subgraph_input(x.dtype, x.shape, "w")
>>>     y = w + x
>>>     pir.subgraph_output(y)
>>>
>>> ir = pir.Ir()
>>> with ir.main_graph():
>>>     w = pir.variable(1)
>>>     x = pir.variable(3)
>>>     add_w_graph = ir.get_graph(add_w, x, w)
>>>     y = ops.call(add_w_graph, x, w)
```

Parameters **t** (`Tensor`) – The subgraph tensor to mark as an output in the current graph.

Return type None

Throws:

ValueError: If #t is not in the current graph.

popart.ir.**variable**(*data*, *dtype=popart.ir.dtypes.float32*, *name=None*)

A variable tensor that is initialised with data during graph creation.

This tensor can be used to represent a model weight or any other parameter that can change while running a model.

Must be created in the main graph scope. Example:

```
>>> import popart.ir as pir
>>> with pir.Ir().main_graph():
>>>     a = pir.variable(0)
```

Parameters

- **data** (np.ndarray, or a value numpy can use to construct an np.ndarray) – The data used to initialise the tensor.
- **dtype** (dtype) – The data type of the tensor. Defaults to pir.float32.
- **name** (Optional[str]) – The name of the tensor. Defaults to None.

Return type popart.ir.tensor.Variable

popart.ir.**virtual_graph**(*vgid*)

Set the virtual graph id on Ops created in this context.

Parameters **vgid** (int) –

CHAPTER

THREE

INDEX

CHAPTER

FOUR

TRADEMARKS & COPYRIGHT

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

V

W