API

API Overview

ONNX Runtime loads and runs inference on a model in ONNX graph format, or ORT format (for memory and disk constrained environments).

The data consumed and produced by the model can be specified and accessed in the way that best matches your scenario.

Load and run a model

InferenceSession is the main class of ONNX Runtime. It is used to load and run an ONNX model, as well as specify environment and application configuration options.

```
session = onnxruntime.InferenceSession('model.onnx')

outputs = session.run([output names], inputs)
```

ONNX and ORT format models consist of a graph of computations, modeled as operators, and implemented as optimized operator kernels for different hardware targets. ONNX Runtime orchestrates the execution of operator kernels via *execution providers*. An execution provider contains the set of kernels for a specific execution target (CPU, GPU, IoT etc). Execution provides are configured using the *providers* parameter. Kernels from different execution providers are chosen in the priority order given in the list of providers. In the example below if there is a kernel in the CUDA execution provider ONNX Runtime executes that on GPU. If not the kernel is executed on CPU.

```
session = onnxruntime.InferenceSession(
        model, providers=['CUDAExecutionProvider', 'CPUExecutionProvider']
)
```

The list of available execution providers can be found here: Execution Providers.

Since ONNX Runtime 1.10, you must explicitly specify the execution provider for your target. Running on CPU is the only time the API allows no explicit setting of the *provider* parameter. In the examples that follow, the *CUDAExecutionProvider* and *CPUExecutionProvider* are used, assuming the application is running on NVIDIA GPUs. Replace these with the execution provider specific to your environment.

You can supply other session configurations via the *session options* parameter. For example, to enable profiling on the session:

```python
options = onnxruntime.SessionOptions()
options.enable_profiling=True
session = onnxruntime.InferenceSession(
        'model.onnx',
        sess_options=options,
        providers=['CUDAExecutionProvider', 'CPUExecutionProvider'])
)
```

Data inputs and outputs

The ONNX Runtime Inference Session consumes and produces data using its OrtValue class.

Data on CPU

On CPU (the default), OrtValues can be mapped to and from native Python data structures: numpy arrays, dictionaries and lists of numpy arrays.

```python
# X is numpy array on cpu
ortvalue = onnxruntime.OrtValue.ortvalue_from_numpy(X)
ortvalue.device_name()  # 'cpu'
ortvalue.shape()        # shape of the numpy array X
ortvalue.data_type()    # 'tensor(float)'
ortvalue.is_tensor()    # 'True'
np.array_equal(ortvalue.numpy(), X)  # 'True'

# ortvalue can be provided as part of the input feed to a model
session = onnxruntime.InferenceSession(
        'model.onnx',
        providers=['CUDAExecutionProvider', 'CPUExecutionProvider'])
)
results = session.run(["Y"], {"X": ortvalue})
```

By default, *ONNX Runtime* always places input(s) and output(s) on CPU. Having the data on CPU may not optimal if the input or output is consumed and produced on a device other than CPU because it introduces data copy between CPU and the device.

Data on device

ONNX Runtime supports a custom data structure that supports all ONNX data formats that allows users to place the data backing these on a device, for example, on a CUDA supported device. In ONNX Runtime, this called *IOBinding*.

To use the *IOBinding* feature, replace *InferenceSession.run()* with *InferenceSession.run_with_iobinding()*.

A graph is executed on a device other than CPU, for instance CUDA. Users can use IOBinding to copy the data onto the GPU.

```
# X is numpy array on cpu
session = onnxruntime.InferenceSession(
        'model.onnx',
        providers=['CUDAExecutionProvider', 'CPUExecutionProvider'])
)
io_binding = session.io_binding()
# OnnxRuntime will copy the data over to the CUDA device if 'input' is consumed by nod
io_binding.bind_cpu_input('input', X)
io_binding.bind_output('output')
session.run_with_iobinding(io_binding)
Y = io_binding.copy_outputs_to_cpu()[0]
```

The input data is on a device, users directly use the input. The output data is on CPU.

```
# X is numpy array on cpu
X_ortvalue = onnxruntime.OrtValue.ortvalue_from_numpy(X, 'cuda', 0)
session = onnxruntime.InferenceSession(
        'model.onnx',
        providers=['CUDAExecutionProvider', 'CPUExecutionProvider'])
)
io_binding = session.io_binding()
io_binding.bind_input(name='input', device_type=X_ortvalue.device_name(), device_id=0,
io_binding.bind_output('output')
session.run_with_iobinding(io_binding)
Y = io_binding.copy_outputs_to_cpu()[0]
```

The input data and output data are both on a device, users directly use the input and also place output on the device.

```
#X is numpy array on cpu
X_ortvalue = onnxruntime.OrtValue.ortvalue_from_numpy(X, 'cuda', 0)
Y_ortvalue = onnxruntime.OrtValue.ortvalue_from_shape_and_type([3, 2], np.float32, 'cu
session = onnxruntime.InferenceSession(
        'model.onnx',
        providers=['CUDAExecutionProvider', 'CPUExecutionProvider'])
)
io_binding = session.io_binding()
io_binding.bind_input(
        name='input',
        device_type=X_ortvalue.device_name(),
        device_id=0,
        element_type=np.float32,
        shape=X_ortvalue.shape(),
        buffer_ptr=X_ortvalue.data_ptr()
)
io_binding.bind_output(
        name='output',
        device_type=Y_ortvalue.device_name(),
        device_id=0,
        element_type=np.float32,
        shape=Y_ortvalue.shape(),
        buffer_ptr=Y_ortvalue.data_ptr()
)
session.run_with_iobinding(io_binding)
```

Users can request *ONNX Runtime* to allocate an output on a device. This is particularly useful for dynamic shaped outputs. Users can use the *get_outputs()* API to get access to the *OrtValue* (s) corresponding to the allocated output(s). Users can thus consume the *ONNX Runtime* allocated memory for the output as an *OrtValue*.

```
#X is numpy array on cpu
X_ortvalue = onnxruntime.OrtValue.ortvalue_from_numpy(X, 'cuda', 0)
session = onnxruntime.InferenceSession(
        'model.onnx',
        providers=['CUDAExecutionProvider', 'CPUExecutionProvider'])
)
io_binding = session.io_binding()
io_binding.bind_input(
        name='input',
        device_type=X_ortvalue.device_name(),
        device_id=0,
        element_type=np.float32,
        shape=X_ortvalue.shape(),
        buffer_ptr=X_ortvalue.data_ptr()
)
#Request ONNX Runtime to bind and allocate memory on CUDA for 'output'
io_binding.bind_output('output', 'cuda')
session.run_with_iobinding(io_binding)
# The following call returns an OrtValue which has data allocated by ONNX Runtime on C
ort_output = io_binding.get_outputs()[0]
```

In addition, *ONNX Runtime* supports directly working with *OrtValue* (s) while inferencing a model if provided as part of the input feed.

Users can bind *OrtValue* (s) directly.

```
#X is numpy array on cpu
#X is numpy array on cpu
X_ortvalue = onnxruntime.OrtValue.ortvalue_from_numpy(X, 'cuda', 0)
Y_ortvalue = onnxruntime.OrtValue.ortvalue_from_shape_and_type([3, 2], np.float32, 'cu
session = onnxruntime.InferenceSession(
        'model.onnx',
        providers=['CUDAExecutionProvider', 'CPUExecutionProvider'])
)
io_binding = session.io_binding()
io_binding.bind_ortvalue_input('input', X_ortvalue)
io_binding.bind_ortvalue_output('output', Y_ortvalue)
session.run_with_iobinding(io_binding)
```

You can also bind inputs and outputs directly to a PyTorch tensor.

```
# X is a PyTorch tensor on device
session = onnxruntime.InferenceSession('model.onnx', providers=['CUDAExecutionProvider
binding = session.io_binding()

X_tensor = X.contiguous()

binding.bind_input(
    name='X',
    device_type='cuda',
    device_id=0,
    element_type=np.float32,
    shape=tuple(x_tensor.shape),
    buffer_ptr=x_tensor.data_ptr(),
    )

## Allocate the PyTorch tensor for the model output
Y_shape = ... # You need to specify the output PyTorch tensor shape
Y_tensor = torch.empty(Y_shape, dtype=torch.float32, device='cuda:0').contiguous()
binding.bind_output(
    name='Y',
    device_type='cuda',
    device_id=0,
    element_type=np.float32,
    shape=tuple(Y_tensor.shape),
    buffer_ptr=Y_tensor.data_ptr(),
)

session.run_with_iobinding(binding)
```

You can also see code examples of this API in in the ONNX Runtime inferences examples.

API Details

InferenceSession

```
class onnxruntime.InferenceSession(path_or_bytes: str | bytes | os.PathLike,
    sess_options: Sequence[onnxruntime.SessionOptions] | None = None, providers:
    Sequence[str | tuple[str, dict[Any, Any]]] | None = None, provider_options:
    Sequence[dict[Any, Any]] | None = None, **kwargs)                 [source]
```

This is the main class used to run a model.

PARAMETERS:

- **path_or_bytes** – Filename or serialized ONNX or ORT format model in a byte string.
- **sess_options** – Session options.
- **providers** – Optional sequence of providers in order of decreasing precedence. Values can either be provider names or tuples of (provider name, options dict). If not provided, then all available providers are used with the default precedence.
- **provider_options** – Optional sequence of options dicts corresponding to the providers listed in 'providers'.

The model type will be inferred unless explicitly set in the SessionOptions. To explicitly set:

```
so = onnxruntime.SessionOptions()
# so.add_session_config_entry('session.load_model_format', 'ONNX') or
so.add_session_config_entry('session.load_model_format', 'ORT')
```

A file extension of '.ort' will be inferred as an ORT format model. All other filenames are assumed to be ONNX format models.

'providers' can contain either names or names and options. When any options are given in 'providers', 'provider_options' should not be used.

The list of providers is ordered by precedence. For example *['CUDAExecutionProvider', 'CPUExecutionProvider']* means execute a node using *CUDAExecutionProvider* if capable, otherwise execute using *CPUExecutionProvider*.

`disable_fallback()`

Disable session.run() fallback mechanism.

`enable_fallback()`

Enable session.Run() fallback mechanism. If session.Run() fails due to an internal Execution Provider failure, reset the Execution Providers enabled for this session. If GPU is enabled, fall back to CUDAExecutionProvider. otherwise fall back to CPUExecutionProvider.

`end_profiling()`

End profiling and return results in a file.

The results are stored in a filename if the option `onnxruntime.SessionOptions.enable_profiling()`.

`get_inputs()`

Return the inputs metadata as a list of `onnxruntime.NodeArg`.

get_modelmeta()

Return the metadata. See `onnxruntime.ModelMetadata`.

get_outputs()

Return the outputs metadata as a list of `onnxruntime.NodeArg`.

get_overridable_initializers()

Return the inputs (including initializers) metadata as a list of `onnxruntime.NodeArg`.

get_profiling_start_time_ns()

Return the nanoseconds of profiling's start time Comparable to time.monotonic_ns() after Python 3.3 On some platforms, this timer may not be as precise as nanoseconds For instance, on Windows and MacOS, the precision will be ~100ns

get_provider_options()

Return registered execution providers' configurations.

get_providers()

Return list of registered execution providers.

get_session_options()

Return the session options. See `onnxruntime.SessionOptions`.

io_binding()

Return an onnxruntime.IOBinding object`.

run(output_names, input_feed, run_options=None)

Compute the predictions.

PARAMETERS:

- **output_names** – name of the outputs
- **input_feed** – dictionary `{ input_name: input_value }`
- **run_options** – See `onnxruntime.RunOptions`.

RETURNS:

list of results, every result is either a numpy array, a sparse tensor, a list or a dictionary.

```
sess.run([output_name], {input_name: x})
```

run_async(output_names, input_feed, callback, user_data, run_options=None)

Compute the predictions asynchronously in a separate cxx thread from ort intra-op threadpool.

PARAMETERS:

- **output_names** – name of the outputs
- **input_feed** – dictionary `{ input_name: input_value }`
- **callback** – python function that accept array of results, and a status string on error. The callback will be invoked by a cxx thread from ort intra-op threadpool.
- **run_options** – See `onnxruntime.RunOptions`.

::

```
class MyData:
    def __init__(self):
        # ...

    def save_results(self, results):
        # ...

def callback(results: np.ndarray, user_data: MyData, err: str) -> None:
    if err:
        print (err)

    else:
        # save results to user_data

sess.run_async([output_name], {input_name: x}, callback)
```

`run_with_iobinding``(iobinding, run_options=None)`

Compute the predictions.

PARAMETERS:

- **iobinding** – the iobinding object that has graph inputs/outputs bind.
- **run_options** – See `onnxruntime.RunOptions`.

`run_with_ort_values``(output_names, input_dict_ort_values, run_options=None)`

Compute the predictions.

PARAMETERS:

- **output_names** – name of the outputs
- **input_dict_ort_values** – dictionary `{ input_name: input_ort_value }` See `OrtValue` class how to create *OrtValue* from numpy array or *SparseTensor*
- **run_options** – See `onnxruntime.RunOptions`.

RETURNS:

an array of *OrtValue*

```
sess.run([output_name], {input_name: x})
```

`run_with_ortvaluevector(run_options, feed_names, feeds, fetch_names, fetches, fetch_devices)`

Compute the predictions similar to other run_*() methods but with minimal C++/Python conversion overhead.

PARAMETERS:

- **run_options** – See `onnxruntime.RunOptions`.
- **feed_names** – list of input names.
- **feeds** – list of input OrtValue.
- **fetch_names** – list of output names.
- **fetches** – list of output OrtValue.
- **fetch_devices** – list of output devices.

`set_providers(providers=None, provider_options=None)`

Register the input list of execution providers. The underlying session is re-created.

PARAMETERS:

- **providers** – Optional sequence of providers in order of decreasing precedence. Values can either be provider names or tuples of (provider name, options dict). If not provided, then all available providers are used with the default precedence.
- **provider_options** – Optional sequence of options dicts corresponding to the providers listed in 'providers'.

'providers' can contain either names or names and options. When any options are given in 'providers', 'provider_options' should not be used.

The list of providers is ordered by precedence. For example *['CUDAExecutionProvider', 'CPUExecutionProvider']* means execute a node using CUDAExecutionProvider if capable, otherwise execute using CPUExecutionProvider.

Options

RunOptions

`class onnxruntime.RunOptions(self: onnxruntime.capi.onnxruntime_pybind11_state.RunOptions)`

Configuration information for a single Run.

`add_run_config_entry(self: onnxruntime.capi.onnxruntime_pybind11_state.RunOptions, arg0: str, arg1: str) → None`

Set a single run configuration entry as a pair of strings.

`get_run_config_entry(self: onnxruntime.capi.onnxruntime_pybind11_state.RunOptions, arg0: str) → str`

Get a single run configuration value using the given configuration key.

`property log_severity_level`

Info, 2:Warning. 3:Error, 4:Fatal. Default is 2.

TYPE:

Log severity level for a particular Run() invocation. 0

TYPE:

Verbose, 1

`property log_verbosity_level`

VLOG level if DEBUG build and run_log_severity_level is 0. Applies to a particular Run() invocation. Default is 0.

`property logid`

To identify logs generated by a particular Run() invocation.

`property only_execute_path_to_fetches`

Only execute the nodes needed by fetch list

`property terminate`

Set to True to terminate any currently executing calls that are using this RunOptions instance. The individual calls will exit gracefully and return an error status.

`property training_mode`

Choose to run in training or inferencing mode

SessionOptions

```
class onnxruntime.SessionOptions(self:
    onnxruntime.capi.onnxruntime_pybind11_state.SessionOptions)
```

Configuration information for a session.

`add_external_initializers(self: onnxruntime.capi.onnxruntime_pybind11_state.SessionOptions, arg0: list, arg1: list) → None`

`add_free_dimension_override_by_denotation(self: onnxruntime.capi.onnxruntime_pybind11_state.SessionOptions, arg0: str, arg1: int) → None`

Specify the dimension size for each denotation associated with an input's free dimension.

`add_free_dimension_override_by_name(self: onnxruntime.capi.onnxruntime_pybind11_state.SessionOptions, arg0: str, arg1: int) → None`

Specify values of named dimensions within model inputs.

`add_initializer(self: onnxruntime.capi.onnxruntime_pybind11_state.SessionOptions, arg0: str, arg1: object) → None`

`add_session_config_entry(self: onnxruntime.capi.onnxruntime_pybind11_state.SessionOptions, arg0: str, arg1: str) → None`

Set a single session configuration entry as a pair of strings.

property `enable_cpu_mem_arena`

Enables the memory arena on CPU. Arena may pre-allocate memory for future usage. Set this option to false if you don't want it. Default is True.

property `enable_mem_pattern`

Enable the memory pattern optimization. Default is true.

property `enable_mem_reuse`

Enable the memory reuse optimization. Default is true.

property `enable_profiling`

Enable profiling for this session. Default is false.

property `execution_mode`

Sets the execution mode. Default is sequential.

property `execution_order`

Sets the execution order. Default is basic topological order.

get_session_config_entry(self: onnxruntime.capi.onnxruntime_pybind11_state.SessionOptions, arg0: str) → str

Get a single session configuration value using the given configuration key.

property graph_optimization_level

Graph optimization level for this session.

property inter_op_num_threads

Sets the number of threads used to parallelize the execution of the graph (across nodes). Default is 0 to let onnxruntime choose.

property intra_op_num_threads

Sets the number of threads used to parallelize the execution within nodes. Default is 0 to let onnxruntime choose.

property log_severity_level

Log severity level. Applies to session load, initialization, etc. 0:Verbose, 1:Info, 2:Warning. 3:Error, 4:Fatal. Default is 2.

property log_verbosity_level

VLOG level if DEBUG build and session_log_severity_level is 0. Applies to session load, initialization, etc. Default is 0.

property logid

Logger id to use for session output.

property optimized_model_filepath

File path to serialize optimized model to. Optimized model is not serialized unless optimized_model_filepath is set. Serialized model format will default to ONNX unless: - add_session_config_entry is used to set 'session.save_model_format' to 'ORT', or - there is no 'session.save_model_format' config entry and optimized_model_filepath ends in '.ort' (case insensitive)

property profile_file_prefix

The prefix of the profile file. The current time will be appended to the file name.

register_custom_ops_library(self: onnxruntime.capi.onnxruntime_pybind11_state.SessionOptions, arg0: str) → None

Specify the path to the shared library containing the custom op kernels required to run a model.

property use_deterministic_compute

Whether to use deterministic compute. Default is false.

class onnxruntime.**ExecutionMode**(self: onnxruntime.capi.onnxruntime_pybind11_state.ExecutionMode, value: int)

Members:

ORT_SEQUENTIAL

ORT_PARALLEL

property name

class onnxruntime.**ExecutionOrder**(self: onnxruntime.capi.onnxruntime_pybind11_state.ExecutionOrder, value: int)

Members:

DEFAULT

PRIORITY_BASED

property name

class onnxruntime.**GraphOptimizationLevel**(self: onnxruntime.capi.onnxruntime_pybind11_state.GraphOptimizationLevel, value: int)

Members:

ORT_DISABLE_ALL

ORT_ENABLE_BASIC

ORT_ENABLE_EXTENDED

ORT_ENABLE_ALL

property name

class onnxruntime.**OrtAllocatorType**(self: onnxruntime.capi.onnxruntime_pybind11_state.OrtAllocatorType, value: int)

Members:

INVALID

ORT_DEVICE_ALLOCATOR

ORT_ARENA_ALLOCATOR

property name

class onnxruntime.**OrtArenaCfg**(*args, **kwargs)

Overloaded function.

1. __init__(self: onnxruntime.capi.onnxruntime_pybind11_state.OrtArenaCfg, arg0: int, arg1: int, arg2: int, arg3: int) -> None
2. __init__(self: onnxruntime.capi.onnxruntime_pybind11_state.OrtArenaCfg, arg0: dict) -> None

`class onnxruntime.OrtMemoryInfo(self: onnxruntime.capi.onnxruntime_pybind11_state.OrtMemoryInfo, arg0: str, arg1: onnxruntime.capi.onnxruntime_pybind11_state.OrtAllocatorType, arg2: int, arg3: onnxruntime.capi.onnxruntime_pybind11_state.OrtMemType)`

`class onnxruntime.OrtMemType(self: onnxruntime.capi.onnxruntime_pybind11_state.OrtMemType, value: int)`

Members:

CPU_INPUT

CPU_OUTPUT

CPU

DEFAULT

`property name`

Functions

Allocators

`onnxruntime.create_and_register_allocator(arg0: OrtMemoryInfo, arg1: OrtArenaCfg) → None`

`onnxruntime.create_and_register_allocator_v2(arg0: str, arg1: OrtMemoryInfo, arg2: Dict[str, str], arg3: OrtArenaCfg) → None`

Telemetry events

`onnxruntime.disable_telemetry_events() → None`

Disables platform-specific telemetry collection.

`onnxruntime.enable_telemetry_events() → None`

Enables platform-specific telemetry collection where applicable.

Providers

`onnxruntime.get_all_providers() → List[str]`

Return list of Execution Providers that this version of Onnxruntime can support. The order of elements represents the default priority order of Execution Providers from highest to lowest.

`onnxruntime.get_available_providers() → List[str]`

Return list of available Execution Providers in this installed version of Onnxruntime. The order of elements represents the default priority order of Execution Providers from highest to lowest.

Build, Version

`onnxruntime.get_build_info() → str`

`onnxruntime.get_version_string() → str`

Device

`onnxruntime.get_device() → str`

Return the device used to compute the prediction (CPU, MKL, …)

Logging

`onnxruntime.set_default_logger_severity(arg0: int) → None`

Sets the default logging severity. 0:Verbose, 1:Info, 2:Warning, 3:Error, 4:Fatal

`onnxruntime.set_default_logger_verbosity(arg0: int) → None`

Sets the default logging verbosity level. To activate the verbose log, you need to set the default logging severity to 0:Verbose level.

Random

`onnxruntime.set_seed(arg0: int) → None`

Sets the seed used for random number generation in Onnxruntime.

Data

OrtValue

```
class onnxruntime.OrtValue(ortvalue, numpy_obj=None)
```
[source]

A data structure that supports all ONNX data formats (tensors and non-tensors) that allows users to place the data backing these on a device, for example, on a CUDA supported device. This class provides APIs to construct and deal with OrtValues.

`as_sparse_tensor()` [source]

The function will return SparseTensor contained in this OrtValue

`data_ptr()` [source]

Returns the address of the first element in the OrtValue's data buffer

`data_type()` [source]

Returns the data type of the data in the OrtValue

`device_name()` [source]

Returns the name of the device where the OrtValue's data buffer resides e.g. cpu, cuda, cann

`element_type()` [source]

Returns the proto type of the data in the OrtValue if the OrtValue is a tensor.

`has_value()` [source]

Returns True if the OrtValue corresponding to an optional type contains data, else returns False

`is_sparse_tensor()` [source]

Returns True if the OrtValue contains a SparseTensor, else returns False

`is_tensor()` [source]

Returns True if the OrtValue contains a Tensor, else returns False

`is_tensor_sequence()` [source]

Returns True if the OrtValue contains a Tensor Sequence, else returns False

`numpy()` [source]

Returns a Numpy object from the OrtValue. Valid only for OrtValues holding Tensors. Throws for OrtValues holding non-Tensors. Use accessors to gain a reference to non-Tensor objects such as SparseTensor

`static ort_value_from_sparse_tensor(sparse_tensor)` [source]

The function will construct an OrtValue instance from a valid SparseTensor The new instance of OrtValue will assume the ownership of sparse_tensor

`static ortvalue_from_numpy(numpy_obj, device_type='cpu', device_id=0)` [source]

Factory method to construct an OrtValue (which holds a Tensor) from a given Numpy object A copy of the data in the Numpy object is held by the OrtValue only if the device is NOT cpu

PARAMETERS:

- **numpy_obj** – The Numpy object to construct the OrtValue from
- **device_type** – e.g. cpu, cuda, cann, cpu by default
- **device_id** – device id, e.g. 0

`static ortvalue_from_shape_and_type(shape=None, element_type=None, device_type='cpu', device_id=0)` [source]

Factory method to construct an OrtValue (which holds a Tensor) from given shape and element_type

PARAMETERS:

- **shape** – List of integers indicating the shape of the OrtValue
- **element_type** – The data type of the elements in the OrtValue (numpy type)
- **device_type** – e.g. cpu, cuda, cann, cpu by default
- **device_id** – device id, e.g. 0

`shape()` [source]

Returns the shape of the data in the OrtValue

`update_inplace(np_arr)` [source]

Update the OrtValue in place with a new Numpy array. The numpy contents are copied over to the device memory backing the OrtValue. It can be used to update the input valuess for an InferenceSession with CUDA graph enabled or other scenarios where the OrtValue needs to be updated while the memory address can not be changed.

SparseTensor

```
class onnxruntime.SparseTensor(sparse_tensor)
```
[source]

A data structure that project the C++ SparseTensor object The class provides API to work with the object. Depending on the format, the class will hold more than one buffer depending on the format

Internal constructor

as_blocksparse_view() [source]

The method will return coo representation of the sparse tensor which will enable querying BlockSparse indices. If the instance did not contain BlockSparse format, it would throw. You can query coo indices as:

```
block_sparse_indices = sparse_tensor.as_blocksparse_view().indices()
```

which will return a numpy array that is backed by the native memory

as_coo_view() [source]

The method will return coo representation of the sparse tensor which will enable querying COO indices. If the instance did not contain COO format, it would throw. You can query coo indices as:

```
coo_indices = sparse_tensor.as_coo_view().indices()
```

which will return a numpy array that is backed by the native memory.

as_csrc_view() [source]

The method will return CSR(C) representation of the sparse tensor which will enable querying CRS(C) indices. If the instance dit not contain CSR(C) format, it would throw. You can query indices as:

```
inner_ndices = sparse_tensor.as_csrc_view().inner()
outer_ndices = sparse_tensor.as_csrc_view().outer()
```

returning numpy arrays backed by the native memory.

data_type() [source]

Returns a string data type of the data in the OrtValue

dense_shape() [source]

Returns a numpy array(int64) containing a dense shape of a sparse tensor

device_name() [source]

Returns the name of the device where the SparseTensor data buffers reside e.g. cpu, cuda `format()` [source]

Returns a OrtSparseFormat enumeration

`static sparse_coo_from_numpy(dense_shape, values, coo_indices, ort_device)` [source]

Factory method to construct a SparseTensor in COO format from given arguments

PARAMETERS:

- **dense_shape** – 1-D numpy array(int64) or a python list that contains a dense_shape of the sparse tensor must be on cpu memory
- **values** – a homogeneous, contiguous 1-D numpy array that contains non-zero elements of the tensor of a type.
- **coo_indices** – contiguous numpy array(int64) that contains COO indices for the tensor. coo_indices may have a 1-D shape when it contains a linear index of non-zero values and its length must be equal to that of the values. It can also be of 2-D shape, in which has it contains pairs of coordinates for each of the nnz values and its length must be exactly twice of the values length.
- **ort_device** –
    - describes the backing memory owned by the supplied nummpy arrays. Only CPU memory is

  suppored for non-numeric data types.

For primitive types, the method will map values and coo_indices arrays into native memory and will use them as backing storage. It will increment the reference count for numpy arrays and it will decrement it on GC. The buffers may reside in any storage either CPU or GPU. For strings and objects, it will create a copy of the arrays in CPU memory as ORT does not support those on other devices and their memory can not be mapped.

`static sparse_csr_from_numpy(dense_shape, values, inner_indices, outer_indices, ort_device)` [source]

Factory method to construct a SparseTensor in CSR format from given arguments

PARAMETERS:

- **dense_shape** – 1-D numpy array(int64) or a python list that contains a dense_shape of the sparse tensor (rows, cols) must be on cpu memory
- **values** – a contiguous, homogeneous 1-D numpy array that contains non-zero elements of the tensor of a type.
- **inner_indices** – contiguous 1-D numpy array(int64) that contains CSR inner indices for the tensor. Its length must be equal to that of the values.
- **outer_indices** – contiguous 1-D numpy array(int64) that contains CSR outer indices for the tensor. Its length must be equal to the number of rows + 1.
- **ort_device** –
  - describes the backing memory owned by the supplied nummpy arrays. Only CPU memory is

  suppored for non-numeric data types.

For primitive types, the method will map values and indices arrays into native memory and will use them as backing storage. It will increment the reference count and it will decrement then count when it is GCed. The buffers may reside in any storage either CPU or GPU. For strings and objects, it will create a copy of the arrays in CPU memory as ORT does not support those on other devices and their memory can not be mapped.

`to_cuda(ort_device)` [source]

Returns a copy of this instance on the specified cuda device

PARAMETERS:

ort_device – with name 'cuda' and valid gpu device id

The method will throw if:

- this instance contains strings
- this instance is already on GPU. Cross GPU copy is not supported
- CUDA is not present in this build
- if the specified device is not valid

`values()` [source]

The method returns a numpy array that is backed by the native memory if the data type is numeric. Otherwise, the returned numpy array that contains copies of the strings.

Devices

IOBinding

```
class onnxruntime.IOBinding(session: Session)
```
[source]

This class provides API to bind input/output to a specified device, e.g. GPU.

`bind_cpu_input(name, arr_on_cpu)` [source]

bind an input to array on CPU :param name: input name :param arr_on_cpu: input values as a python array on CPU

`bind_input(name, device_type, device_id, element_type, shape, buffer_ptr)` [source]

PARAMETERS:

- **name** – input name
- **device_type** – e.g. cpu, cuda, cann
- **device_id** – device id, e.g. 0
- **element_type** – input element type
- **shape** – input shape
- **buffer_ptr** – memory pointer to input data

`bind_ortvalue_input(name, ortvalue)` [source]

PARAMETERS:

- **name** – input name
- **ortvalue** – OrtValue instance to bind

`bind_ortvalue_output(name, ortvalue)` [source]

PARAMETERS:

- **name** – output name
- **ortvalue** – OrtValue instance to bind

`bind_output(name, device_type='cpu', device_id=0, element_type=None, shape=None, buffer_ptr=None)` [source]

PARAMETERS:

- **name** – output name
- **device_type** – e.g. cpu, cuda, cann, cpu by default
- **device_id** – device id, e.g. 0
- **element_type** – output element type
- **shape** – output shape
- **buffer_ptr** – memory pointer to output data

`copy_outputs_to_cpu()` [source]

Copy output contents to CPU (if on another device). No-op if already on the CPU.

`get_outputs()` [source]

Returns the output OrtValues from the Run() that preceded the call. The data buffer of the obtained OrtValues may not reside on CPU memory

```
class onnxruntime.SessionIOBinding(self:
    onnxruntime.capi.onnxruntime_pybind11_state.SessionIOBinding, arg0:
```

`onnxruntime.capi.onnxruntime_pybind11_state.InferenceSession)`

`bind_input``(*args, **kwargs)`

Overloaded function.

1. bind_input(self: onnxruntime.capi.onnxruntime_pybind11_state.SessionIOBinding, arg0: str, arg1: object) -> None
2. bind_input(self: onnxruntime.capi.onnxruntime_pybind11_state.SessionIOBinding, arg0: str, arg1: onnxruntime.capi.onnxruntime_pybind11_state.OrtDevice, arg2: object, arg3: List[int], arg4: int) -> None

`bind_ortvalue_input``(self: onnxruntime.capi.onnxruntime_pybind11_state.SessionIOBinding, arg0: str, arg1: onnxruntime.capi.onnxruntime_pybind11_state.OrtValue) → None`

`bind_ortvalue_output``(self: onnxruntime.capi.onnxruntime_pybind11_state.SessionIOBinding, arg0: str, arg1: onnxruntime.capi.onnxruntime_pybind11_state.OrtValue) → None`

`bind_output``(*args, **kwargs)`

Overloaded function.

1. bind_output(self: onnxruntime.capi.onnxruntime_pybind11_state.SessionIOBinding, arg0: str, arg1: onnxruntime.capi.onnxruntime_pybind11_state.OrtDevice, arg2: object, arg3: List[int], arg4: int) -> None
2. bind_output(self: onnxruntime.capi.onnxruntime_pybind11_state.SessionIOBinding, arg0: str, arg1: onnxruntime.capi.onnxruntime_pybind11_state.OrtDevice) -> None

`clear_binding_inputs``(self: onnxruntime.capi.onnxruntime_pybind11_state.SessionIOBinding) → None`

`clear_binding_outputs``(self: onnxruntime.capi.onnxruntime_pybind11_state.SessionIOBinding) → None`

`copy_outputs_to_cpu``(self: onnxruntime.capi.onnxruntime_pybind11_state.SessionIOBinding) → List[object]`

`get_outputs``(self: onnxruntime.capi.onnxruntime_pybind11_state.SessionIOBinding) → onnxruntime.capi.onnxruntime_pybind11_state.OrtValueVector`

`synchronize_inputs``(self: onnxruntime.capi.onnxruntime_pybind11_state.SessionIOBinding) → None`

`synchronize_outputs``(self: onnxruntime.capi.onnxruntime_pybind11_state.SessionIOBinding) → None`

OrtDevice

`class onnxruntime.OrtDevice(c_ort_device)` [source]

A data structure that exposes the underlying C++ OrtDevice

Internal constructor

Internal classes

These classes cannot be instantiated by users but they are returned by methods or functions of this library.

ModelMetadata

`class onnxruntime.ModelMetadata`

Pre-defined and custom metadata about the model. It is usually used to identify the model used to run the prediction and facilitate the comparison.

`property custom_metadata_map`

additional metadata

`property description`

description of the model

`property domain`

ONNX domain

`property graph_description`

description of the graph hosted in the model

`property graph_name`

graph name

`property producer_name`

producer name

`property version`

version of the model

NodeArg

class `onnxruntime.`**`NodeArg`**

Node argument definition, for both input and output, including arg name, arg type (contains both type and shape).

property **`name`**

node name

property **`shape`**

node shape (assuming the node holds a tensor)

property **`type`**

node type

Backend

In addition to the regular API which is optimized for performance and usability, *ONNX Runtime* also implements the ONNX backend API for verification of *ONNX* specification conformance. The following functions are supported:

`onnxruntime.backend.is_compatible(model, device=None, **kwargs)`

Return whether the model is compatible with the backend.

PARAMETERS:

- **model** – unused
- **device** – None to use the default device or a string (ex: *'CPU'*)

RETURNS:

boolean

`onnxruntime.backend.prepare(model, device=None, **kwargs)`

Load the model and creates a `onnxruntime.InferenceSession` ready to be used as a backend.

PARAMETERS:

- **model** – ModelProto (returned by *onnx.load*), string for a filename or bytes for a serialized model
- **device** – requested device for the computation, None means the default one which depends on the compilation settings
- **kwargs** – see `onnxruntime.SessionOptions`

RETURNS:

`onnxruntime.InferenceSession`

`onnxruntime.backend.run(model, inputs, device=None, **kwargs)`

Compute the prediction.

PARAMETERS:

- **model** – `onnxruntime.InferenceSession` returned by function *prepare*
- **inputs** – inputs
- **device** – requested device for the computation, None means the default one which depends on the compilation settings
- **kwargs** – see `onnxruntime.RunOptions`

RETURNS:

predictions

`onnxruntime.backend.supports_device(device)`

Check whether the backend is compiled with particular device support. In particular it's used in the testing suite.