Concrete-ML API Documentation

Contents

Module `src.concrete.ml`

ML module.

Sub-modules

- src.concrete.ml.common
- src.concrete.ml.deployment
- src.concrete.ml.onnx
- src.concrete.ml.quantization
- src.concrete.ml.sklearn
- src.concrete.ml.torch
- src.concrete.ml.version

Module `src.concrete.ml.common`

Module for shared data structures and code.

Sub-modules

- src.concrete.ml.common.check_inputs
- src.concrete.ml.common.debugging
- src.concrete.ml.common.utils

Module `src.concrete.ml.common.check_inputs`

Check and conversion tools.

Utils that are used to check (including convert) some data types which are compatible with scikit-learn to numpy types.

Functions

Function `check_X_y_and_assert`

```
def check_X_y_and_assert(
    X,
    y,
    *args,
    **kwargs
)
```

sklearn.utils.check_X_y with an assert.

Equivalent of sklearn.utils.check_X_y, with a final assert that the type is one which is supported by Concrete-ML.

Args —–= **X** : ndarray, list, sparse matrix : Input data

y : ndarray, list, sparse matrix Labels
***args** The arguments to pass to check_X_y
****kwargs** The keyword arguments to pass to check_X_y

Returns —–= The converted and validated arrays

Function `check_array_and_assert`

```
def check_array_and_assert(
    X
)
```

sklearn.utils.check_array with an assert.

Equivalent of sklearn.utils.check_array, with a final assert that the type is one which is supported by Concrete-ML.

Args —–= **X** : object : Input object to check / convert

Returns —–= The converted and validated array

Module `src.concrete.ml.common.debugging`

Module for debugging.

Sub-modules

- src.concrete.ml.common.debugging.custom_assert

Module `src.concrete.ml.common.debugging.custom_assert`

Provide some variants of assert.

Functions

Function `assert_false`

```
def assert_false(
    condition: bool,
    on_error_msg: str = '',
    error_type: Type[Exception] = builtins.AssertionError
)
```

Provide a custom assert to check that the condition is False.

Args ——= condition(bool): the condition. If True, raise AssertionError on_error_msg(str): optional message for precising the error, in case of error **`error_type`** : Type[Exception] : the type of error to raise, if condition is not fullfilled. Default to AssertionError

Function `assert_not_reached`

```
def assert_not_reached(
    on_error_msg: str,
    error_type: Type[Exception] = builtins.AssertionError
)
```

Provide a custom assert to check that a piece of code is never reached.

Args ——= on_error_msg(str): message for precising the error **`error_type`** : Type[Exception] : the type of error to raise, if condition is not fullfilled. Default to AssertionError

Function `assert_true`

```
def assert_true(
    condition: bool,
    on_error_msg: str = '',
    error_type: Type[Exception] = builtins.AssertionError
)
```

Provide a custom assert to check that the condition is True.

Args ——= condition(bool): the condition. If False, raise AssertionError on_error_msg(str): optional message for precising the error, in case of error **`error_type`** : Type[Exception] : the type of error to raise, if condition is not fullfilled. Default to AssertionError

Module `src.concrete.ml.common.utils`

Utils that can be re-used by other pieces of code in the module.

Functions

Function `generate_proxy_function`

```
def generate_proxy_function(
    function_to_proxy: Callable,
    desired_functions_arg_names: Iterable[str]
) -> Tuple[Callable, Dict[str, str]]
```

Generate a proxy function for a function accepting only *args type arguments.

This returns a runtime compiled function with the sanitized argument names passed in desired_functions_arg_names as the arguments to the function.

Args ——= **`function_to_proxy`** : Callable : the function defined like def f(*args) for which to return a function like f_proxy(arg_1, arg_2) for any number of arguments.

desired_functions_arg_names : Iterable[str] the argument names to use, these names are sanitized and the mapping between the original argument name to the sanitized one is returned in a dictionary. Only the sanitized names will work for a call to the proxy function.

Returns ——= Tuple[Callable, Dict[str, str]] : the proxy function and the mapping of the original arg name to the new and sanitized arg names.

Function `get_onnx_opset_version`

```
def get_onnx_opset_version(
    onnx_model: onnx.onnx_ml_pb2.ModelProto
) -> int
```

Return the ONNX opset_version.

Args ——= **`onnx_model`** : onnx.ModelProto : the model.

Returns ——= int : the version of the model

Function `replace_invalid_arg_name_chars`

```
def replace_invalid_arg_name_chars(
    arg_name: str
) -> str
```

Sanitize arg_name, replacing invalid chars by _.

This does not check that the starting character of arg_name is valid.

Args ——= **`arg_name`** : str : the arg name to sanitize.

Returns ——= str : the sanitized arg name, with only chars in _VALID_ARG_CHARS.

Module `src.concrete.ml.deployment`

Module for deployment of the FHE model.

Sub-modules

- src.concrete.ml.deployment.fhe_client_server

Module `src.concrete.ml.deployment.fhe_client_server`

APIs for FHE deployment.

Classes

Class `FHEModelClient`

```
class FHEModelClient(
    path_dir: str,
    key_dir: str = None
)
```

Client API to encrypt and decrypt FHE data.

Initialize the FHE API.

Args ——= **`path_dir`** : str : the path to the directory where the circuit is saved

key_dir : str the path to the directory where the keys are stored

Class variables

Variable client Type: `concrete.numpy.compilation.client.Client`

Methods

Method `deserialize_decrypt_dequantize`

```
def deserialize_decrypt_dequantize(
    self,
    serialized_encrypted_quantized_result: concrete.compiler.public_arguments.PublicArguments
) -> numpy.ndarray
```

Deserialize, decrypt and dequantize the values.

Args —–= **serialized_encrypted_quantized_result** : cnp.PublicArguments : the serialized, encrypted and quantized result

Returns —–= numpy.ndarray : the decrypted, dequantized values

Method `generate_private_and_evaluation_keys`

```
def generate_private_and_evaluation_keys(
    self,
    force=False
)
```

Generate the private and evaluation keys.

Args —–= **force** : bool : if True, regenerate the keys even if they already exist

Method `get_serialized_evaluation_keys`

```
def get_serialized_evaluation_keys(
    self
) -> concrete.compiler.evaluation_keys.EvaluationKeys
```

Get the serialized evaluation keys.

Returns —–= cnp.EvaluationKeys : the evaluation keys

Method `load`

```
def load(
    self
)
```

Load the quantizers along with the FHE specs.

Method `quantize_encrypt_serialize`

```
def quantize_encrypt_serialize(
    self,
    x: numpy.ndarray
) -> concrete.compiler.public_arguments.PublicArguments
```

Quantize, encrypt and serialize the values.

Args —–= **x** : numpy.ndarray : the values to quantize, encrypt and serialize

Returns —–= cnp.PublicArguments : the quantized, encrypted and serialized values

Class FHEModelDev

```
class FHEModelDev(
    path_dir: str,
    model: Any = None
)
```

Dev API to save the model and then load and run the FHE circuit.

Initialize the FHE API.

Args ——= **path_dir** : str : the path to the directory where the circuit is saved

model : Any the model to use for the FHE API

Class variables

Variable `model` Type: `Any`

Methods

Method `save`

```
def save(
    self
)
```

Export all needed artifacts for the client and server.

Raises ——= Exception : path_dir is not empty

Class FHEModelServer

```
class FHEModelServer(
    path_dir: str
)
```

Server API to load and run the FHE circuit.

Initialize the FHE API.

Args ——= **path_dir** : str : the path to the directory where the circuit is saved

Class variables

Variable `server` Type: `concrete.numpy.compilation.server.Server`

Methods

Method `load`

```
def load(
    self
)
```

Load the circuit.

Method `run`

```
def run(
    self,
    serialized_encrypted_quantized_data: concrete.compiler.public_arguments.PublicArguments,
    serialized_evaluation_keys: concrete.compiler.evaluation_keys.EvaluationKeys
) -> concrete.compiler.public_result.PublicResult
```

Run the model on the server over encrypted data.

Args ——= **`serialized_encrypted_quantized_data`** : cnp.PublicArguments : the encrypted, quantized and serialized data

`serialized_evaluation_keys` : cnp.EvaluationKeys the serialized evaluation keys

Returns ——= cnp.PublicResult : the result of the model

Module `src.concrete.ml.onnx`

ONNX module.

Sub-modules

- src.concrete.ml.onnx.convert
- src.concrete.ml.onnx.onnx_model_manipulations
- src.concrete.ml.onnx.onnx_utils
- src.concrete.ml.onnx.ops_impl

Module `src.concrete.ml.onnx.convert`

ONNX conversion related code.

Functions

Function `get_equivalent_numpy_forward`

```
def get_equivalent_numpy_forward(
    onnx_model: onnx.onnx_ml_pb2.ModelProto,
    check_model: bool = True
) -> Callable[..., Tuple[numpy.ndarray, ...]]
```

Get the numpy equivalent forward of the provided ONNX model.

Args ——= **`onnx_model`** : onnx.ModelProto : the ONNX model for which to get the equivalent numpy forward.

`check_model` : bool set to True to run the onnx checker on the model. Defaults to True.

Raises ——= ValueError : Raised if there is an unsupported ONNX operator required to convert the torch model to numpy.

Returns ——= Callable[..., Tuple[numpy.ndarray, ...]] : The function that will execute the equivalent numpy function.

Function `get_equivalent_numpy_forward_and_onnx_model`

```
def get_equivalent_numpy_forward_and_onnx_model(
    torch_module: torch.nn.modules.module.Module,
    dummy_input: Union[torch.Tensor, Tuple[torch.Tensor, ...]],
    output_onnx_file: Union[pathlib.Path, str, None] = None
) -> Tuple[Callable[..., Tuple[numpy.ndarray, ...]], onnx.onnx_ml_pb2.GraphProto]
```

Get the numpy equivalent forward of the provided torch Module.

Args ——= **`torch_module`** : torch.nn.Module : the torch Module for which to get the equivalent numpy forward.

`dummy_input` : Union[torch.Tensor, Tuple[torch.Tensor, ...]] dummy inputs for ONNX export.
`output_onnx_file` : Optional[Union[Path, str]], optional Path to save the ONNX file to. Will use a temp file if not provided. Defaults to None.

Returns ——= Tuple[Callable[..., Tuple[numpy.ndarray, ...]], onnx.GraphProto] : The function that will execute the equivalent numpy code to the passed torch_module and the generated ONNX model.

Module `src.concrete.ml.onnx.onnx_model_manipulations`

Some code to manipulate models.

Functions

Function `clean_graph_after_sigmoid`

```
def clean_graph_after_sigmoid(
    onnx_model: onnx.onnx_ml_pb2.ModelProto
)
```

Clean the graph of the onnx model, by removing nodes after the sigmoid.

Args ——= **`onnx_model`** : onnx.ModelProto : the onnx model

Returns ——= onnx.ModelProto : the cleaned onnx model

Function `cut_onnx_graph_after_node_name`

```
def cut_onnx_graph_after_node_name(
    onnx_model: onnx.onnx_ml_pb2.ModelProto,
    node_name: str
) -> str
```

Cut the graph after the node with the given name.

Args ——= **`onnx_model`** : onnx.ModelProto : the ONNX model to modify.

`node_name` : `str` the name of the node after which the graph will be cut. (node_name is included in the new graph)

Returns ——= str : the name of the output to keep

Function `keep_following_outputs_discard_others`

```
def keep_following_outputs_discard_others(
    onnx_model: onnx.onnx_ml_pb2.ModelProto,
    outputs_to_keep: Iterable[str]
)
```

Keep the outputs given in outputs_to_keep and remove the others from the model.

Args ——= **`onnx_model`** : onnx.ModelProto : the ONNX model to modify.

`outputs_to_keep` : Iterable[str] the outputs to keep by name.

Function `remove_identity_nodes`

```
def remove_identity_nodes(
    onnx_model: onnx.onnx_ml_pb2.ModelProto
)
```

Remove identity nodes from a model.

Args ——= **`onnx_model`** : onnx.ModelProto : the model for which we want to remove Identity nodes.

Function `remove_unused_constant_nodes`

```
def remove_unused_constant_nodes(
    onnx_model: onnx.onnx_ml_pb2.ModelProto
)
```

Remove unused Constant nodes in the provided onnx model.

Args —– = **onnx_model** : onnx.ModelProto : the model for which we want to remove unused Constant nodes.

Function replace_uncessary_nodes_by_identity

```
def replace_uncessary_nodes_by_identity(
    onnx_model: onnx.onnx_ml_pb2.ModelProto,
    op_type_to_replace: list
)
```

Replace unecessary nodes by Identity nodes.

Args —– = **onnx_model** : onnx.ModelProto : the ONNX model to modify.

op_type_to_replace : list the op_type of the nodes to be replaced by Identity nodes.

Raises —– = ValueError : Wrong replacement by an Identity node.

Function simplify_onnx_model

```
def simplify_onnx_model(
    onnx_model: onnx.onnx_ml_pb2.ModelProto
)
```

Simplify an ONNX model, removes unused Constant nodes and Identity nodes.

Args —– = **onnx_model** : onnx.ModelProto : the model to simplify.

Module src.concrete.ml.onnx.onnx_utils

Utils to interpret an ONNX model with numpy.

Functions

Function execute_onnx_with_numpy

```
def execute_onnx_with_numpy(
    graph: onnx.onnx_ml_pb2.GraphProto,
    *inputs: numpy.ndarray
) -> Tuple[numpy.ndarray, ...]
```

Execute the provided ONNX graph on the given inputs.

Args —– = **graph** : onnx.GraphProto : The ONNX graph to execute.

***inputs** The inputs of the graph.

Returns —– = Tuple[numpy.ndarray] : The result of the graph's execution.

Function get_attribute

```
def get_attribute(
    attribute: onnx.onnx_ml_pb2.AttributeProto
) -> Any
```

Get the attribute from an ONNX AttributeProto.

Args —– = **attribute** : onnx.AttributeProto : The attribute to retrieve the value from.

Returns —– = Any : The stored attribute value.

Module src.concrete.ml.onnx.ops_impl

ONNX ops implementation in python + numpy.

Functions

Function cast_to_float

```
def cast_to_float(
    inputs
)
```

Cast values to floating points.

Args ——= **inputs** : Tuple[numpy.ndarray] : The values to consider.

Returns ——= Tuple[numpy.ndarray] : The float values.

Function numpy_abs

```
def numpy_abs(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute abs in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Abs-13

Args ——= **x** : numpy.ndarray : Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function numpy_acos

```
def numpy_acos(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute acos in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Acos-7

Args ——= **x** : numpy.ndarray : Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function numpy_acosh

```
def numpy_acosh(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute acosh in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Acosh-9

Args ——= **x** : numpy.ndarray : Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function numpy_add

```
def numpy_add(
    a: numpy.ndarray,
    b: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute add in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Add-13

Args ——= **a** : numpy.ndarray : First operand.

b : numpy.ndarray Second operand.

Returns ——= Tuple[numpy.ndarray] : Result, has same element type as two inputs

Function numpy_asin

```
def numpy_asin(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute asin in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Asin-7

Args ——= **x** : numpy.ndarray : Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function numpy_asinh

```
def numpy_asinh(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute sinh in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Asinh-9

Args ——= **x** : numpy.ndarray : Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function numpy_atan

```
def numpy_atan(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute atan in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Atan-7

Args ——= **x** : numpy.ndarray : Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function numpy_atanh

```
def numpy_atanh(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute atanh in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Atanh-9

Args ——= **x** : numpy.ndarray : Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function `numpy_batchnorm`

```
def numpy_batchnorm(
    x: numpy.ndarray,
    scale: numpy.ndarray,
    bias: numpy.ndarray,
    input_mean: numpy.ndarray,
    input_var: numpy.ndarray,
    /,
    *,
    epsilon=1e-05,
    momentum=0.9,
    training_mode=0
) -> Tuple[numpy.ndarray]
```

Compute the batch normalization of the input tensor.

This can be expressed as:

Y = (X - input_mean) / sqrt(input_var + epsilon) * scale + B

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#BatchNormalization-14

Args ——= **`x`** : numpy.ndarray : tensor to normalize, dimensions are in the form of (N,C,D1,D2,...,Dn), where N is the batch size, C is the number of channels.

`scale` : numpy.ndarray scale tensor of shape (C,)
`bias` : numpy.ndarray bias tensor of shape (C,)
`input_mean` : numpy.ndarray mean values to use for each input channel, shape (C,)
`input_var` : numpy.ndarray variance values to use for each input channel, shape (C,)
`epsilon` : float avoids division by zero
`momentum` : float momentum used during training of the mean/variance, not used in inference
`training_mode` : int if the model was exported in training mode this is set to 1, else 0

Returns ——= numpy.ndarray : Normalized tensor

Function `numpy_cast`

```
def numpy_cast(
    data: numpy.ndarray,
    /,
    *,
    to: int
) -> Tuple[numpy.ndarray]
```

Execute ONNX cast in Numpy.

Supports only booleans for now, which are converted to integers.

See: https://github.com/onnx/onnx/blob/main/docs/Operators.md#Cast

Args ——= **`data`** : numpy.ndarray : Input encrypted tensor

`to` : int integer value of the onnx.TensorProto DataType enum

Returns ——= result (numpy.ndarray): a tensor with the required data type

Function `numpy_celu`

```
def numpy_celu(
    x: numpy.ndarray,
    /,
    *,
    alpha: float = 1
) -> Tuple[numpy.ndarray]
```

Compute celu in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Celu-12

Args ——= **x** : numpy.ndarray : Input tensor

alpha : float Coefficient

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function numpy_clip

```
def numpy_clip(
    a: numpy.ndarray,
    /,
    min=None,
    max=None
) -> Tuple[numpy.ndarray]
```

Compute clip in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Clip-13

Args ——= **a** : numpy.ndarray : Input tensor whose elements to be clipped.

min : [type], optional Minimum value, under which element is replaced by min. It must be a scalar(tensor of empty shape). Defaults to None.

max : [type], optional Maximum value, above which element is replaced by max. It must be a scalar(tensor of empty shape). Defaults to None.

Returns ——= Tuple[numpy.ndarray] : Output tensor with clipped input elements.

Function numpy_constant

```
def numpy_constant(
    **kwargs
)
```

Return the constant passed as kwarg.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Constant-13

Args ——= ****kwargs** : keyword arguments

Returns ——= Any : The stored constant.

Function numpy_cos

```
def numpy_cos(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute cos in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Cos-7

Args ——= **x** : numpy.ndarray : Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function numpy_cosh

```
def numpy_cosh(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute cosh in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Cosh-9

Args ——= **x** : numpy.ndarray : Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function `numpy_div`

```
def numpy_div(
    a: numpy.ndarray,
    b: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute div in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Div-14

Args ——= **a** : numpy.ndarray : Input tensor

b : numpy.ndarray Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function `numpy_elu`

```
def numpy_elu(
    x: numpy.ndarray,
    /,
    *,
    alpha: float = 1
) -> Tuple[numpy.ndarray]
```

Compute elu in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Elu-6

Args ——= **x** : numpy.ndarray : Input tensor

alpha : float Coefficient

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function `numpy_equal`

```
def numpy_equal(
    x: numpy.ndarray,
    y: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute equal in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Equal-11

Args ——= **x** : numpy.ndarray : Input tensor

y : numpy.ndarray Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function `numpy_erf`

```
def numpy_erf(
    x: numpy.ndarray,
    /
```

```
) -> Tuple[numpy.ndarray]
```

Compute erf in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Erf-13

Args ——= **x** : numpy.ndarray : Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function `numpy_exp`

```
def numpy_exp(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute exponential in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Exp-13

Args ——= **x** : numpy.ndarray : Input tensor

Returns ——= Tuple[numpy.ndarray] : The exponential of the input tensor computed element-wise

Function `numpy_flatten`

```
def numpy_flatten(
    x: numpy.ndarray,
    /,
    *,
    axis: int = 1
) -> Tuple[numpy.ndarray]
```

Flatten a tensor into a 2d array.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Flatten-13.

Args ——= **x** : numpy.ndarray : tensor to flatten

axis : int axis after which all dimensions will be flattened (axis=0 gives a 1D output)

Returns ——= result : flattened tensor

Function `numpy_gemm`

```
def numpy_gemm(
    a: numpy.ndarray,
    b: numpy.ndarray,
    /,
    c: Optional[numpy.ndarray] = None,
    *,
    alpha: float = 1,
    beta: float = 1,
    transA: int = 0,
    transB: int = 0
) -> Tuple[numpy.ndarray]
```

Compute Gemm in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Gemm-13

Args ——= **a** : numpy.ndarray : Input tensor A. The shape of A should be (M, K) if transA is 0, or (K, M) if transA is non-zero.

b : numpy.ndarray Input tensor B. The shape of B should be (K, N) if transB is 0, or (N, K) if transB is non-zero.

c : Optional[numpy.ndarray] Optional input tensor C. If not specified, the computation is done as if C is a scalar 0. The shape of C should be unidirectional broadcastable to (M, N). Defaults to None.
alpha : float Scalar multiplier for the product of input tensors A * B. Defaults to 1.
beta : float Scalar multiplier for input tensor C. Defaults to 1.
transA : int Whether A should be transposed. The type is kept as int as it's the type used by ONNX and it can easily be interpreted by python as a boolean. Defaults to 0.
transB : int Whether B should be transposed. The type is kept as int as it's the type used by ONNX and it can easily be interpreted by python as a boolean. Defaults to 0.

Returns —–= Tuple[numpy.ndarray] : The tuple containing the result tensor

Function `numpy_greater`

```
def numpy_greater(
    x: numpy.ndarray,
    y: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute greater in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Greater-13

Args —–= **x** : numpy.ndarray : Input tensor

y : numpy.ndarray Input tensor

Returns —–= Tuple[numpy.ndarray] : Output tensor

Function `numpy_greater_float`

```
def numpy_greater_float(
    x: numpy.ndarray,
    y: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute greater in numpy according to ONNX spec and cast outputs to floats.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Greater-13

Args —–= **x** : numpy.ndarray : Input tensor

y : numpy.ndarray Input tensor

Returns —–= Tuple[numpy.ndarray] : Output tensor

Function `numpy_greater_or_equal`

```
def numpy_greater_or_equal(
    x: numpy.ndarray,
    y: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute greater or equal in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#GreaterOrEqual-12

Args —–= **x** : numpy.ndarray : Input tensor

y : numpy.ndarray Input tensor

Returns —–= Tuple[numpy.ndarray] : Output tensor

Function `numpy_greater_or_equal_float`

```
def numpy_greater_or_equal_float(
    x: numpy.ndarray,
    y: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute greater or equal in numpy according to ONNX specs and cast outputs to floats.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#GreaterOrEqual-12

Args —–= **x** : numpy.ndarray : Input tensor

y : numpy.ndarray Input tensor

Returns —–= Tuple[numpy.ndarray] : Output tensor

Function `numpy_hardsigmoid`

```
def numpy_hardsigmoid(
    x: numpy.ndarray,
    /,
    *,
    alpha: float = 0.2,
    beta: float = 0.5
) -> Tuple[numpy.ndarray]
```

Compute hardsigmoid in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#HardSigmoid-6

Args —–= **x** : numpy.ndarray : Input tensor

alpha : float Coefficient
beta : float Coefficient

Returns —–= Tuple[numpy.ndarray] : Output tensor

Function `numpy_hardswish`

```
def numpy_hardswish(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute hardswitch in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#hardswish-14

Args —–= **x** : numpy.ndarray : Input tensor

Returns —–= Tuple[numpy.ndarray] : Output tensor

Function `numpy_identity`

```
def numpy_identity(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute identity in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Identity-14

Args —–= **x** : numpy.ndarray : Input tensor

Returns —–= Tuple[numpy.ndarray] : Output tensor

Function `numpy_leakyrelu`

```
def numpy_leakyrelu(
    x: numpy.ndarray,
    /,
    *,
    alpha: float = 0.01
) -> Tuple[numpy.ndarray]
```

Compute leakyrelu in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#LeakyRelu-6

Args ——= **x** : numpy.ndarray : Input tensor

alpha : float Coefficient

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function `numpy_less`

```
def numpy_less(
    x: numpy.ndarray,
    y: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute less in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Less-13

Args ——= **x** : numpy.ndarray : Input tensor

y : numpy.ndarray Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function `numpy_less_float`

```
def numpy_less_float(
    x: numpy.ndarray,
    y: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute less in numpy according to ONNX spec and cast outputs to floats.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Less-13

Args ——= **x** : numpy.ndarray : Input tensor

y : numpy.ndarray Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function `numpy_less_or_equal`

```
def numpy_less_or_equal(
    x: numpy.ndarray,
    y: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute less or equal in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#LessOrEqual-12

Args ——= **x** : numpy.ndarray : Input tensor

y : numpy.ndarray Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function `numpy_less_or_equal_float`

```
def numpy_less_or_equal_float(
    x: numpy.ndarray,
    y: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute less or equal in numpy according to ONNX spec and cast outputs to floats.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#LessOrEqual-12

Args ——= `x` : numpy.ndarray : Input tensor

y : numpy.ndarray Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function `numpy_log`

```
def numpy_log(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute log in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Log-13

Args ——= `x` : numpy.ndarray : Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function `numpy_matmul`

```
def numpy_matmul(
    a: numpy.ndarray,
    b: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute matmul in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#MatMul-13

Args ——= `a` : numpy.ndarray : N-dimensional matrix A

b : numpy.ndarray N-dimensional matrix B

Returns ——= Tuple[numpy.ndarray] : Matrix multiply results from A * B

Function `numpy_mul`

```
def numpy_mul(
    a: numpy.ndarray,
    b: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute mul in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Mul-14

Args ——= `a` : numpy.ndarray : Input tensor

b : numpy.ndarray Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function numpy_not

```
def numpy_not(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute not in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Not-1

Args ——= **x** : numpy.ndarray : Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function numpy_not_float

```
def numpy_not_float(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute not in numpy according to ONNX spec and cast outputs to floats.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Not-1

Args ——= **x** : numpy.ndarray : Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function numpy_or

```
def numpy_or(
    a: numpy.ndarray,
    b: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute or in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Or-7

Args ——= **a** : numpy.ndarray : Input tensor

b : numpy.ndarray Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function numpy_or_float

```
def numpy_or_float(
    a: numpy.ndarray,
    b: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute or in numpy according to ONNX spec and cast outputs to floats.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Or-7

Args ——= **a** : numpy.ndarray : Input tensor

b : numpy.ndarray Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function numpy_pad

```
def numpy_pad(
    data: numpy.ndarray,
    pads: numpy.ndarray,
    constant_value: Optional[numpy.ndarray] = None,
    /,
    *,
    mode: str
) -> Tuple[numpy.ndarray]
```

Apply padding in numpy according to ONNX spec.

See: https://github.com/onnx/onnx/blob/main/docs/Operators.md#Pad

Args —–= **data** : numpy.ndarray : Input variable/tensor to pad

pads : numpy.ndarray List of pads (size 8) to apply, two per N,C,H,W dimension
constant_value : float Constant value to use for padding
mode : str padding mode: constant/edge/reflect

Returns —–= res (numpy.ndarray): Padded tensor

Function numpy_pow

```
def numpy_pow(
    a: numpy.ndarray,
    b: numpy.ndarray
) -> Tuple[numpy.ndarray]
```

Compute pow in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Pow-13

Args —–= **a** : numpy.ndarray : Input tensor whose elements to be raised.

b : numpy.ndarray The power to which we want to raise.

Returns —–= Tuple[numpy.ndarray] : Output tensor.

Function numpy_prelu

```
def numpy_prelu(
    x: numpy.ndarray,
    slope: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute prelu in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#prelu-16

Args —–= **x** : numpy.ndarray : Input tensor

slope : numpy.ndarray Slope of PRelu

Returns —–= Tuple[numpy.ndarray] : Output tensor

Function numpy_reduce_sum

```
def numpy_reduce_sum(
    a: numpy.ndarray,
    /,
    axes: Optional[numpy.ndarray] = None,
    *,
    keepdims: int = 1,
    noop_with_empty_axes: int = 0
```

```
) -> Tuple[numpy.ndarray]
```

Compute ReduceSum in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Operators.md#ReduceSum

Args ——= **a** : numpy.ndarray : Input tensor whose elements to sum.

axes : Optional[numpy.ndarray] Array of integers along which to reduce. The default is to reduce over all the dimensions of the input tensor if 'noop_with_empty_axes' is false, else act as an Identity op when 'noop_with_empty_axes' is true. Accepted range is [-r, r-1] where r = rank(data). Default to None.

keepdims : int Keep the reduced dimension or not, 1 means keeping the input dimension, 0 will reduce it along the given axis. Default to 1.

noop_with_empty_axes : int Defines behaviour if 'axes' is empty or set to None. Default behaviour with 0 is to reduce all axes. When axes is empty and this attribute is set to true 1, input tensor will not be reduced, and the output tensor would be equivalent to input tensor. Default to 0.

Returns ——= numpy.ndarray : Output reduced tensor.

Function `numpy_relu`

```
def numpy_relu(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute relu in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Relu-14

Args ——= **x** : numpy.ndarray : Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function `numpy_reshape`

```
def numpy_reshape(
    x: numpy.ndarray,
    newshape: numpy.ndarray,
    /,
    *,
    allowzero=0
) -> Tuple[numpy.ndarray]
```

Compute reshape in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Reshape-13

Args ——= **x** : numpy.ndarray : Input tensor

newshape : numpy.ndarray New shape
allowzero : int ONNX legacy parameter, by default 0 -> behave like numpy reshape

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function `numpy_round`

```
def numpy_round(
    a: numpy.ndarray
) -> Tuple[numpy.ndarray]
```

Compute round in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Round-11 Remark that ONNX Round operator is actually a rint, since the number of decimals is forced to be 0

Args ——= **a** : numpy.ndarray : Input tensor whose elements to be rounded.

Returns ——= Tuple[numpy.ndarray] : Output tensor with rounded input elements.

Function `numpy_selu`

```
def numpy_selu(
    x: numpy.ndarray,
    /,
    *,
    alpha: float = 1.6732632423543772,
    gamma: float = 1.0507009873554805
) -> Tuple[numpy.ndarray]
```

Compute selu in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Selu-6

Args ——= **x** : numpy.ndarray : Input tensor

alpha : float Coefficient
gamma : float Coefficient

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function `numpy_sigmoid`

```
def numpy_sigmoid(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute sigmoid in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Sigmoid-13

Args ——= **x** : numpy.ndarray : Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function `numpy_sin`

```
def numpy_sin(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute sin in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Sin-7

Args ——= **x** : numpy.ndarray : Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function `numpy_sinh`

```
def numpy_sinh(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute sinh in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Sinh-9

Args ——= **x** : numpy.ndarray : Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function numpy_softplus

```
def numpy_softplus(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute softplus in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Softplus-1

Args ——= **x** : numpy.ndarray : Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function numpy_sub

```
def numpy_sub(
    a: numpy.ndarray,
    b: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute sub in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Sub-14

Args ——= **a** : numpy.ndarray : Input tensor

b : numpy.ndarray Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function numpy_tan

```
def numpy_tan(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute tan in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Tan-7

Args ——= **x** : numpy.ndarray : Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function numpy_tanh

```
def numpy_tanh(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute tanh in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Tanh-13

Args ——= **x** : numpy.ndarray : Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function numpy_thresholdedrelu

```
def numpy_thresholdedrelu(
    x: numpy.ndarray,
    /,
```

```
        *,
        alpha: float = 1
    ) -> Tuple[numpy.ndarray]
```

Compute thresholdedrelu in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#ThresholdedRelu-10

Args ——= **x** : numpy.ndarray : Input tensor

alpha : float Coefficient

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function numpy_transpose

```
    def numpy_transpose(
        x: numpy.ndarray,
        /,
        *,
        perm=None
    ) -> Tuple[numpy.ndarray]
```

Transpose in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Transpose-13

Args ——= **x** : numpy.ndarray : Input tensor

perm : numpy.ndarray Permutation of the axes

Returns ——= Tuple[numpy.ndarray] : Output tensor

Function numpy_where

```
    def numpy_where(
        c: numpy.ndarray,
        t: numpy.ndarray,
        f: numpy.ndarray,
        /
    ) -> Tuple[numpy.ndarray]
```

Compute the equivalent of numpy.where.

Args ——= **c** : numpy.ndarray : Condition operand.

t : numpy.ndarray True operand.
f : numpy.ndarray False operand.

Returns ——= numpy.ndarray : numpy.where(c, t, f)

Function numpy_where_body

```
    def numpy_where_body(
        c: numpy.ndarray,
        t: numpy.ndarray,
        f: Union[numpy.ndarray, int],
        /
    ) -> numpy.ndarray
```

Compute the equivalent of numpy.where.

This function is not mapped to any ONNX operator (as opposed to numpy_where). It is usable by functions which are mapped to ONNX operators, eg numpy_div or numpy_where.

Args ——= **c** : numpy.ndarray : Condition operand.

t : numpy.ndarray True operand.

f : numpy.ndarray False operand.

Returns ——= numpy.ndarray : numpy.where(c, t, f)

FIXME: can it be improved with a native numpy.where in Concrete Numpy?

https://github.com/zama-ai/concrete-numpy-internal/issues/1429

Function torch_avgpool

```
def torch_avgpool(
    x: numpy.ndarray,
    /,
    *,
    ceil_mode: int,
    kernel_shape: Tuple[int, ...],
    pads: Tuple[int, ...],
    strides: Tuple[int, ...]
) -> Tuple[numpy.ndarray]
```

Compute Average Pooling using Torch.

Currently supports 2d average pooling with torch semantics. This function is ONNX compatible.

See: https://github.com/onnx/onnx/blob/main/docs/Operators.md#AveragePool

Args ——= **x** : numpy.ndarray : input data (many dtypes are supported). Shape is N x C x H x W for 2d

ceil_mode : int ONNX rounding parameter, expected 0 (torch style dimension computation)
kernel_shape : Tuple[**int**] shape of the kernel. Should have 2 elements for 2d conv
pads : Tuple[**int**] padding in ONNX format (begin, end) on each axis
strides : Tuple[**int**] stride of the convolution on each axis

Returns ——= res (numpy.ndarray): a tensor of size (N x InChannels x OutHeight x OutWidth). See https://pytorch.org/docs/stable/generated/torch.nn.AvgPool2d.html

Raises ——= AssertionError : if the pooling arguments are wrong

Function torch_conv

```
def torch_conv(
    x: numpy.ndarray,
    w: numpy.ndarray,
    b: numpy.ndarray,
    /,
    *,
    dilations: Tuple[int, ...],
    group: int = 1,
    kernel_shape: Tuple[int, ...],
    pads: Tuple[int, ...],
    strides: Tuple[int, ...]
) -> Tuple[numpy.ndarray]
```

Compute N-D convolution using Torch.

Currently supports 2d convolution with torch semantics. This function is also ONNX compatible.

See: https://github.com/onnx/onnx/blob/main/docs/Operators.md#Conv

Args ——= **x** : numpy.ndarray : input data (many dtypes are supported). Shape is N x C x H x W for 2d

w : numpy.ndarray weights tensor. Shape is (O x I x Kh x Kw) for 2d
b : numpy.ndarray, Optional bias tensor, Shape is (O,)
dilations : Tuple[int] dilation of the kernel, default 1 on all dimensions.
group : int number of convolution groups, default 1
kernel_shape : Tuple[int] shape of the kernel. Should have 2 elements for 2d conv
pads : Tuple[int] padding in ONNX format (begin, end) on each axis
strides : Tuple[int] stride of the convolution on each axis

Returns ——= res (numpy.ndarray): a tensor of size (N x OutChannels x OutHeight x OutWidth). See https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html

Raises ——= AssertionError : if the convolution arguments are wrong

Module `src.concrete.ml.quantization`

Modules for quantization.

Sub-modules

- src.concrete.ml.quantization.base_quantized_op
- src.concrete.ml.quantization.post_training
- src.concrete.ml.quantization.quantized_array
- src.concrete.ml.quantization.quantized_module
- src.concrete.ml.quantization.quantized_ops

Module `src.concrete.ml.quantization.base_quantized_op`

Base Quantized Op class that implements quantization for a float numpy op.

Classes

Class `QuantizedOp`

```
class QuantizedOp(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Base class for quantized ONNX ops implemented in numpy.

Args ——= **n_bits_output** : int : The number of bits to use for the quantization of the output

int_input_names : Set[str] The set of names of integer tensors that are inputs to this op
constant_inputs : Optional[Union[Dict[str, Any], Dict[int, Any]]] The constant tensors that are inputs to this op
input_quant_opts : QuantizationOptions Input quantizer options, determine the quantization that is applied to input tensors (that are not constants)

Descendants

- src.concrete.ml.quantization.quantized_ops.QuantizedAbs
- src.concrete.ml.quantization.quantized_ops.QuantizedAdd
- src.concrete.ml.quantization.quantized_ops.QuantizedAvgPool
- src.concrete.ml.quantization.quantized_ops.QuantizedBatchNormalization
- src.concrete.ml.quantization.quantized_ops.QuantizedCast
- src.concrete.ml.quantization.quantized_ops.QuantizedCelu
- src.concrete.ml.quantization.quantized_ops.QuantizedClip
- src.concrete.ml.quantization.quantized_ops.QuantizedConv

- src.concrete.ml.quantization.quantized_ops.QuantizedDiv
- src.concrete.ml.quantization.quantized_ops.QuantizedElu
- src.concrete.ml.quantization.quantized_ops.QuantizedErf
- src.concrete.ml.quantization.quantized_ops.QuantizedExp
- src.concrete.ml.quantization.quantized_ops.QuantizedFlatten
- src.concrete.ml.quantization.quantized_ops.QuantizedGemm
- src.concrete.ml.quantization.quantized_ops.QuantizedGreater
- src.concrete.ml.quantization.quantized_ops.QuantizedGreaterOrEqual
- src.concrete.ml.quantization.quantized_ops.QuantizedHardSigmoid
- src.concrete.ml.quantization.quantized_ops.QuantizedHardSwish
- src.concrete.ml.quantization.quantized_ops.QuantizedIdentity
- src.concrete.ml.quantization.quantized_ops.QuantizedLeakyRelu
- src.concrete.ml.quantization.quantized_ops.QuantizedLess
- src.concrete.ml.quantization.quantized_ops.QuantizedLessOrEqual
- src.concrete.ml.quantization.quantized_ops.QuantizedLog
- src.concrete.ml.quantization.quantized_ops.QuantizedMul
- src.concrete.ml.quantization.quantized_ops.QuantizedNot
- src.concrete.ml.quantization.quantized_ops.QuantizedOr
- src.concrete.ml.quantization.quantized_ops.QuantizedPRelu
- src.concrete.ml.quantization.quantized_ops.QuantizedPad
- src.concrete.ml.quantization.quantized_ops.QuantizedPow
- src.concrete.ml.quantization.quantized_ops.QuantizedReduceSum
- src.concrete.ml.quantization.quantized_ops.QuantizedRelu
- src.concrete.ml.quantization.quantized_ops.QuantizedReshape
- src.concrete.ml.quantization.quantized_ops.QuantizedRound
- src.concrete.ml.quantization.quantized_ops.QuantizedSelu
- src.concrete.ml.quantization.quantized_ops.QuantizedSigmoid
- src.concrete.ml.quantization.quantized_ops.QuantizedSoftplus
- src.concrete.ml.quantization.quantized_ops.QuantizedTanh
- src.concrete.ml.quantization.quantized_ops.QuantizedWhere

Class variables

Variable `POSITIONAL_ARGUMENTS_KINDS`

Variable `attrs` Type: `Dict[str, Any]`

Variable `constant_inputs` Type: `Dict[int, Any]`

Variable `impl` Type: `Optional[Callable[..., Tuple[numpy.ndarray, ...]]]`

Variable `input_quant_opts` Type: `src.concrete.ml.quantization.quantized_array.QuantizationOptions`

Variable `n_bits` Type: `int`

Variable `output_quant_params` Type: `Optional[src.concrete.ml.quantization.quantized_array.UniformQuan`

Variable `output_quant_stats` Type: `Optional[src.concrete.ml.quantization.quantized_array.MinMaxQuanti`

Methods

Method `calibrate`

```
def calibrate(
    self,
    *inputs: numpy.ndarray
) -> numpy.ndarray
```

Create corresponding QuantizedArray for the output of the activation function.

Args ——= **`*inputs`** : numpy.ndarray : Calibration sample inputs.

Returns ——= numpy.ndarray : the output values for the provided calibration samples.

Method `call_impl`

```
def call_impl(
    self,
    *inputs: numpy.ndarray,
    **attrs
) -> numpy.ndarray
```

Call self.impl to centralize mypy bug workaround.

Args ——= **`*inputs`** : numpy.ndarray : real valued inputs.

`**attrs` the QuantizedOp attributes.

Returns ——= numpy.ndarray : return value of self.impl

Method `can_fuse`

```
def can_fuse(
    self
) -> bool
```

Determine if the operator impedes graph fusion.

This function shall be overloaded by inheriting classes to test self._int_input_names, to determine whether the operation can be fused to a TLU or not. For example an operation that takes inputs produced by a unique integer tensor can be fused to a TLU. Example: f(x) = x * (x + 1) can be fused. A function that does f(x) = x * (x @ w + 1) can't be fused.

Returns ——= bool : whether this instance of the QuantizedOp produces Concrete Numpy code that can be fused to TLUs

Method `prepare_output`

```
def prepare_output(
    self,
    qoutput_activation: numpy.ndarray
) -> src.concrete.ml.quantization.quantized_array.QuantizedArray
```

Quantize the output of the activation function.

The calibrate method needs to be called with sample data before using this function.

Args ——= **`qoutput_activation`** : numpy.ndarray : Output of the activation function.

Returns ——= QuantizedArray : Quantized output.

Method `q_impl`

```
def q_impl(
    self,
    *q_inputs: src.concrete.ml.quantization.quantized_array.QuantizedArray,
    **attrs
) -> src.concrete.ml.quantization.quantized_array.QuantizedArray
```

Execute the quantized forward.

Args ——= **`*q_inputs`** : QuantizedArray : Quantized inputs.

`**attrs` the QuantizedOp attributes.

Returns ——= QuantizedArray : The returned quantized value.

Module `src.concrete.ml.quantization.post_training`

Post Training Quantization methods.

Classes

Class `ONNXConverter`

```
class ONNXConverter(
    n_bits: Union[int, Dict[~KT, ~VT]],
    numpy_model: src.concrete.ml.torch.numpy_module.NumpyModule,
    is_signed: bool = False
)
```

Base ONNX to Concrete ML computation graph conversion class.

This class provides a method to parse an ONNX graph and apply several transformations. First, it creates QuantizedOps for each ONNX graph op. These quantized ops have calibrated quantizers that are useful when the operators work on integer data or when the output of the ops is the output of the encrypted program. For operators that compute in float and will be merged to TLUs, these quantizers are not used. Second, this converter creates quantized tensors for initializer and weights stored in the graph.

This class should be sub-classed to provide specific calibration and quantization options depending on the usage (Post-training quantization vs Quantization Aware training).

Arguments —–= n_bits (int, Dict[str, int]): number of bits for quantization, can be a single value or a dictionary with "net_inputs", "op_inputs", "op_weights", "net_outputs" keys, with a bitwidth for each of these elements. When using a single value for n_bits, it is assigned to "op_inputs" and "op_weights" bits and a default value is assigned to the number of output bits. This default is a compromise between model accuracy and runtime performance in FHE. Output bits give the precision of the final network output, while "net_input" bits give the precision of quantization of network inputs. "op_inputs" and "op_weights" control the quantization for the inputs and weights of all layers. numpy_model (NumpyModule): Model in numpy. is_signed (bool): Whether the weights of the layers can be signed. Currently, only the weights can be signed.

Descendants

- src.concrete.ml.quantization.post_training.PostTrainingAffineQuantization
- src.concrete.ml.quantization.post_training.PostTrainingQATImporter

Class variables

Variable `is_signed` Type: `bool`

Variable `n_bits` Type: `Union[int, Dict[~KT, ~VT]]`

Variable `numpy_model` Type: `src.concrete.ml.torch.numpy_module.NumpyModule`

Variable `quant_ops_dict` Type: `Dict[str, Tuple[Tuple[str, ...], src.concrete.ml.quantization.base_qua`

Variable `quant_params` Type: `Dict[str, src.concrete.ml.quantization.quantized_array.QuantizedArray]`

Instance variables

Variable `n_bits_net_inputs` Get the number of bits to use for the quantization of the last layer's output.

Returns —–= n_bits (int): number of bits for output quantization

Variable `n_bits_net_outputs` Get the number of bits to use for the quantization of the last layer's output.

Returns ——= n_bits (int): number of bits for output quantization

Variable `n_bits_op_input_quant` Get the number of bits to use for the quantization of any constant (usually weights).

Returns ——= n_bits (int): number of bits for constants quantization

Variable `n_bits_weights` Get the number of bits to use for the quantization of any constant (usually weights).

Returns ——= n_bits (int): number of bits for constants quantization

Methods

Method `quantize_module`

```
def quantize_module(
    self,
    *calibration_data: numpy.ndarray
) -> src.concrete.ml.quantization.quantized_module.QuantizedModule
```

Quantize numpy module.

Following https://arxiv.org/abs/1712.05877 guidelines.

Args ——= **`*calibration_data`** : numpy.ndarray : Data that will be used to compute the bounds, scales and zero point values for every quantized object.

Returns ——= QuantizedModule : Quantized numpy module

Class `PostTrainingAffineQuantization`

```
class PostTrainingAffineQuantization(
    n_bits: Union[int, Dict[~KT, ~VT]],
    numpy_model: src.concrete.ml.torch.numpy_module.NumpyModule,
    is_signed: bool = False
)
```

Post-training Affine Quantization.

Create the quantized version of the passed numpy module.

Args ——= **`n_bits`** : int, Dict : Number of bits to quantize the model. If an int is passed for n_bits, the value will be used for activation, inputs and weights. If a dict is passed, then it should contain "net_inputs", "op_inputs", "op_weights" and "net_outputs" keys with corresponding number of quantization bits for: - net_inputs : number of bits for model input - op_inputs : number of bits to quantize layer input values - op_weights: learned parameters or constants in the network - net_outputs: final model output quantization bits

`numpy_model` : NumpyModule

```
Model in numpy.
```

`is_signed`

```
Whether the weights of the layers can be signed.
        Currently, only the weights can be signed.
```

Returns ——= QuantizedModule : A quantized version of the numpy model.

Ancestors (in MRO)

- src.concrete.ml.quantization.post_training.ONNXConverter

Class variables

Variable `is_signed` Type: `bool`

Variable `n_bits` Type: `Union[int, Dict[~KT, ~VT]]`

Variable `numpy_model` Type: `src.concrete.ml.torch.numpy_module.NumpyModule`

Variable `quant_ops_dict` Type: `Dict[str, Tuple[Tuple[str, ...], src.concrete.ml.quantization.base_qua`

Variable `quant_params` Type: `Dict[str, src.concrete.ml.quantization.quantized_array.QuantizedArray]`

Class `PostTrainingQATImporter`

```
class PostTrainingQATImporter(
    n_bits: Union[int, Dict[~KT, ~VT]],
    numpy_model: src.concrete.ml.torch.numpy_module.NumpyModule,
    is_signed: bool = False
)
```

Converter of Quantization Aware Training networks.

This class provides specific configuration for QAT networks during ONNX network conversion to Concrete ML computation graphs.

Ancestors (in MRO)

- src.concrete.ml.quantization.post_training.ONNXConverter

Class variables

Variable `is_signed` Type: `bool`

Variable `n_bits` Type: `Union[int, Dict[~KT, ~VT]]`

Variable `numpy_model` Type: `src.concrete.ml.torch.numpy_module.NumpyModule`

Variable `quant_ops_dict` Type: `Dict[str, Tuple[Tuple[str, ...], src.concrete.ml.quantization.base_qua`

Variable `quant_params` Type: `Dict[str, src.concrete.ml.quantization.quantized_array.QuantizedArray]`

Module `src.concrete.ml.quantization.quantized_array`

Quantization utilities for a numpy array/tensor.

Functions

Function `fill_from_kwargs`

```
def fill_from_kwargs(
    obj,
    klass,
    **kwargs
)
```

Fill a parameter set structure from kwargs parameters.

Args —–= `obj` : an object of type klass, if None the object is created if any of the type's members appear in the kwargs

klass the type of object to fill
kwargs parameter names and values to fill into an instance of the klass type

Returns ——= obj : an object of type klass

kwargs remaining parameter names and values that were not filled into obj

Raises ——= TypeError : if the types of the parameters in kwargs could not be converted to the corresponding types of members of klass

Classes

Class `MinMaxQuantizationStats`

```
class MinMaxQuantizationStats
```

Calibration set statistics.

This class stores the statistics for the calibration set or for a calibration data batch. Currently we only store min/max to determine the quantization range. The min/max are computed from the calibration set.

Descendants

- src.concrete.ml.quantization.quantized_array.UniformQuantizer

Class variables

Variable `rmax` Type: `Optional[float]`

Variable `rmin` Type: `Optional[float]`

Variable `uvalues` Type: `Optional[numpy.ndarray]`

Instance variables

Variable `quant_stats` Get a copy of the calibration set statistics.

Returns ——= MinMaxQuantizationStats : a copy of the current quantization stats

Methods

Method `compute_quantization_stats`

```
def compute_quantization_stats(
    self,
    values: numpy.ndarray
) -> None
```

Compute the calibration set quantization statistics.

Args ——= **`values`** : numpy.ndarray : Calibration set on which to compute statistics.

Method `copy_stats`

```
def copy_stats(
    self,
    stats
) -> None
```

Copy the statistics from a different structure.

Args ——= **`stats`** : MinMaxQuantizationStats : structure to copy statistics from.

Class QuantizationOptions

```
class QuantizationOptions(
    n_bits,
    is_signed: bool = False,
    is_symmetric: bool = False,
    is_qat: bool = False
)
```

Options for quantization.

Determines the number of bits for quantization and the method of quantization of the values. Signed quantization allows negative quantized values. Symmetric quantization assumes the float values are distributed symmetrically around x=0 and assigns signed values around 0 to the float values. QAT (quantization aware training) quantization assumes the values are already quantized, taking a discrete set of values, and assigns these values to integers, computing only the scale.

Descendants

- src.concrete.ml.quantization.quantized_array.UniformQuantizer

Class variables

Variable `is_qat` Type: `bool`

Variable `is_signed` Type: `bool`

Variable `is_symmetric` Type: `bool`

Variable `n_bits` Type: `int`

Instance variables

Variable `quant_options` Get a copy of the quantization parameters.

Returns ——= UniformQuantizationParameters : a copy of the current quantization parameters

Methods

Method `copy_opts`

```
def copy_opts(
    self,
    opts
)
```

Copy the options from a different structure.

Args ——= **`opts`** : QuantizationOptions : structure to copy parameters from.

Class QuantizedArray

```
class QuantizedArray(
    n_bits,
    values: Optional[numpy.ndarray],
    value_is_float: bool = True,
    options: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    stats: Optional[src.concrete.ml.quantization.quantized_array.MinMaxQuantizationStats] = Non
    params: Optional[src.concrete.ml.quantization.quantized_array.UniformQuantizationParameters]
    **kwargs
)
```

Abstraction of quantized array.

Contains float values and their quantized integer counter-parts. Quantization is performed by the quantizer member object. Float and int values are kept in sync. Having both types of values is useful since quantized operators in Concrete ML graphs might need one or the other depending on how the operator works (in float or in int). Moreover, when the encrypted function needs to return a value, it must return integer values.

See https://arxiv.org/abs/1712.05877.

Args —–= **`values`** : numpy.ndarray : Values to be quantized.

`n_bits` : int The number of bits to use for quantization.
`value_is_float` : bool, optional Whether the passed values are real (float) values or not. If False, the values will be quantized according to the passed scale and zero_point. Defaults to True.
`options` : QuantizationOptions Quantization options set
`stats` : Optional[MinMaxQuantizationStats] Quantization batch statistics set
`params` : Optional[UniformQuantizationParameters] Quantization parameters set (scale, zero-point)
`kwargs` Any member of the options, stats, params sets as a key-value pair. The parameter sets need to be completely parametrized if their members appear in kwargs.

Class variables

Variable `STABILITY_CONST`

Variable `quantizer` Type: `src.concrete.ml.quantization.quantized_array.UniformQuantizer`

Variable `qvalues` Type: `numpy.ndarray`

Variable `values` Type: `numpy.ndarray`

Methods

Method `dequant`

```
def dequant(
    self
) -> numpy.ndarray
```

Dequantize self.qvalues.

Returns —–= numpy.ndarray : Dequantized values.

Method `quant`

```
def quant(
    self
) -> Optional[numpy.ndarray]
```

Quantize self.values.

Returns —–= numpy.ndarray : Quantized values.

Method `update_quantized_values`

```
def update_quantized_values(
    self,
    qvalues: numpy.ndarray
) -> numpy.ndarray
```

Update qvalues to get their corresponding values using the related quantized parameters.

Args ——= **`qvalues`** : numpy.ndarray : Values to replace self.qvalues

Returns ——= values (numpy.ndarray): Corresponding values

Method `update_values`

```
def update_values(
    self,
    values: numpy.ndarray
) -> numpy.ndarray
```

Update values to get their corresponding qvalues using the related quantized parameters.

Args ——= **`values`** : numpy.ndarray : Values to replace self.values

Returns ——= qvalues (numpy.ndarray): Corresponding qvalues

Class `UniformQuantizationParameters`

```
class UniformQuantizationParameters
```

Quantization parameters for uniform quantization.

This class stores the parameters used for quantizing real values to discrete integer values. The parmeters are computed from quantization options and quantization statistics.

Descendants

- src.concrete.ml.quantization.quantized_array.UniformQuantizer

Class variables

Variable `offset` Type: `Optional[int]`

Variable `scale` Type: `Optional[float]`

Variable `zero_point` Type: `Optional[int]`

Instance variables

Variable `quant_params` Get a copy of the quantization parameters.

Returns ——= UniformQuantizationParameters : a copy of the current quantization parameters

Methods

Method `compute_quantization_parameters`

```
def compute_quantization_parameters(
    self,
    options: src.concrete.ml.quantization.quantized_array.QuantizationOptions,
    stats: src.concrete.ml.quantization.quantized_array.MinMaxQuantizationStats
) -> None
```

Compute the quantization parameters.

Args ——= **`options`** : QuantizationOptions : quantization options set

`stats` : MinMaxQuantizationStats calibrated statistics for quantization

Method `copy_params`

```
def copy_params(
    self,
    params
) -> None
```

Copy the parameters from a different structure.

Args ——= **`params`** : UniformQuantizationParameters : parameter structure to copy

Class `UniformQuantizer`

```
class UniformQuantizer(
    options: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    stats: Optional[src.concrete.ml.quantization.quantized_array.MinMaxQuantizationStats] = None
    params: Optional[src.concrete.ml.quantization.quantized_array.UniformQuantizationParameters]
    **kwargs
)
```

Uniform quantizer.

Contains all information necessary for uniform quantization and provides quantization/dequantization functionality on numpy arrays.

Args ——= **`options`** : QuantizationOptions : Quantization options set

`stats` : Optional[MinMaxQuantizationStats] Quantization batch statistics set
`params` : Optional[UniformQuantizationParameters] Quantization parameters set (scale, zero-point)

Ancestors (in MRO)

- src.concrete.ml.quantization.quantized_array.UniformQuantizationParameters
- src.concrete.ml.quantization.quantized_array.QuantizationOptions
- src.concrete.ml.quantization.quantized_array.MinMaxQuantizationStats

Class variables

Variable `offset` Type: `Optional[int]`

Variable `scale` Type: `Optional[float]`

Variable `zero_point` Type: `Optional[int]`

Methods

Method `dequant`

```
def dequant(
    self,
    qvalues: numpy.ndarray
) -> numpy.ndarray
```

Dequantize values.

Args ——= **`qvalues`** : numpy.ndarray : integer values to de-quantize

Returns ——= numpy.ndarray : Dequantized float values.

Method `quant`

```
def quant(
    self,
    values: numpy.ndarray
) -> numpy.ndarray
```

Quantize values.

Args ——= **`values`** : numpy.ndarray : float values to quantize

Returns ——= numpy.ndarray : Integer quantized values.

Module `src.concrete.ml.quantization.quantized_module`

QuantizedModule API.

Classes

Class `QuantizedModule`

```
class QuantizedModule(
    ordered_module_input_names: Iterable[str] = None,
    ordered_module_output_names: Iterable[str] = None,
    quant_layers_dict: Dict[str, Tuple[Tuple[str, ...], src.concrete.ml.quantization.base_quant
)
```

Inference for a quantized model.

Class variables

Variable `forward_fhe` Type: `Optional[None]`

Variable `input_quantizers` Type: `List[src.concrete.ml.quantization.quantized_array.UniformQuantizer]`

Variable `ordered_module_input_names` Type: `Tuple[str, ...]`

Variable `ordered_module_output_names` Type: `Tuple[str, ...]`

Variable `output_quantizers` Type: `List[src.concrete.ml.quantization.quantized_array.UniformQuantizer]`

Variable `quant_layers_dict` Type: `Dict[str, Tuple[Tuple[str, ...], src.concrete.ml.quantization.base_`

Instance variables

Variable `fhe_circuit` Type: `concrete.numpy.compilation.circuit.Circuit`

Get the FHE circuit.

Returns ——= Circuit : the FHE circuit

Variable `is_compiled` Type: `bool`

Return the compiled status of the module.

Returns ——= bool : the compiled status of the module.

Variable `onnx_model` Get the ONNX model.

.. # noqa: DAR201

Returns ——= _onnx_model (onnx.ModelProto): the ONNX model

Variable post_processing_params Type: Dict[str, Any]

Get the post-processing parameters.

Returns ——= Dict[str, Any] : the post-processing parameters

Methods

Method compile

```
def compile(
    self,
    q_inputs: Union[Tuple[numpy.ndarray, ...], numpy.ndarray],
    configuration: Optional[concrete.numpy.compilation.configuration.Configuration] = None,
    compilation_artifacts: Optional[concrete.numpy.compilation.artifacts.DebugArtifacts] = None
    show_mlir: bool = False,
    use_virtual_lib: bool = False,
    p_error: Optional[float] = 6.3342483999973e-05
) -> concrete.numpy.compilation.circuit.Circuit
```

Compile the forward function of the module.

Args ——= **q_inputs** : Union[Tuple[numpy.ndarray, ...], numpy.ndarray] : Needed for tracing and building the boundaries.

configuration : Optional[Configuration] Configuration object to use during compilation
compilation_artifacts : Optional[DebugArtifacts] Artifacts object to fill during
show_mlir : bool if set, the MLIR produced by the converter and which is going to be sent to the compiler backend is shown on the screen, e.g., for debugging or demo. Defaults to False.
use_virtual_lib : bool set to use the so called virtual lib simulating FHE computation. Defaults to False.
p_error : Optional[float] probability of error of a PBS.

Returns ——= Circuit : the compiled Circuit.

Method dequantize_output

```
def dequantize_output(
    self,
    qvalues: numpy.ndarray
) -> numpy.ndarray
```

Take the last layer q_out and use its dequant function.

Args ——= **qvalues** : numpy.ndarray : Quantized values of the last layer.

Returns ——= numpy.ndarray : Dequantized values of the last layer.

Method forward

```
def forward(
    self,
    *qvalues: numpy.ndarray
) -> numpy.ndarray
```

Forward pass with numpy function only.

Args ——= ***qvalues** : numpy.ndarray : numpy.array containing the quantized values.

Returns ——= (numpy.ndarray): Predictions of the quantized model

Method forward_and_dequant

```
def forward_and_dequant(
    self,
    *q_x: numpy.ndarray
) -> numpy.ndarray
```

Forward pass with numpy function only plus dequantization.

Args —–= **`*q_x`** : numpy.ndarray : numpy.ndarray containing the quantized input values. Requires the input dtype to be uint8.

Returns —–= (numpy.ndarray): Predictions of the quantized model

Method post_processing

```
def post_processing(
    self,
    qvalues: numpy.ndarray
) -> numpy.ndarray
```

Post-processing of the quantized output.

Args —–= **`qvalues`** : numpy.ndarray : numpy.ndarray containing the quantized input values.

Returns —–= (numpy.ndarray): Predictions of the quantized model

Method quantize_input

```
def quantize_input(
    self,
    *values: numpy.ndarray
) -> Union[Tuple[numpy.ndarray, ...], numpy.ndarray]
```

Take the inputs in fp32 and quantize it using the learned quantization parameters.

Args —–= **`*values`** : numpy.ndarray : Floating point values.

Returns —–= Union[numpy.ndarray, Tuple[numpy.ndarray, …]] : Quantized (numpy.uint32) values.

Method set_inputs_quantization_parameters

```
def set_inputs_quantization_parameters(
    self,
    *input_q_params: src.concrete.ml.quantization.quantized_array.UniformQuantizer
)
```

Set the quantization parameters for the module's inputs.

Args —–= **`*input_q_params`** : UniformQuantizer : The quantizer(s) for the module.

Module src.concrete.ml.quantization.quantized_ops

Quantized versions of the ONNX operators for post training quantization.

Classes

Class QuantizedAbs

```
class QuantizedAbs(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
```

```
)
```

Quantized Abs op.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method `impl`

```
def impl(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute abs in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Abs-13

Args —–= **x** : numpy.ndarray : Input tensor

Returns —–= Tuple[numpy.ndarray] : Output tensor

Class `QuantizedAdd`

```
class QuantizedAdd(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Quantized Addition operator.

Can add either two variables (both encrypted) or a variable and a constant

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Descendants

- src.concrete.ml.quantization.quantized_ops.QuantizedSub

Class variables

Variable `b_sign` Type: `int`

Methods

Method `can_fuse`

```
def can_fuse(
    self
) -> bool
```

Determine if this op can be fused.

Add operation can be computed in float and fused if it operates over inputs produced by a single integer tensor. For example the expression x + x * 1.75, where x is an encrypted tensor, can be computed with a single TLU.

Returns ——= bool : Whether the number of integer input tensors allows computing this op as a TLU

Method `impl`

```
def impl(
    a: numpy.ndarray,
    b: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute add in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Add-13

Args ——= **a** : numpy.ndarray : First operand.

b : numpy.ndarray Second operand.

Returns ——= Tuple[numpy.ndarray] : Result, has same element type as two inputs

Class `QuantizedAvgPool`

```
class QuantizedAvgPool(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Quantized Average Pooling op.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method `can_fuse`

```
def can_fuse(
    self
) -> bool
```

Determine if this op can be fused.

Avg Pooling operation can not be fused since it must be performed over integer tensors and it combines different elements of the input tensors.

Returns ——= bool : False, this operation can not be fused as it adds different encrypted integers

Method `impl`

```
def impl(
    x: numpy.ndarray,
    /,
    *,
    ceil_mode: int,
    kernel_shape: Tuple[int, ...],
    pads: Tuple[int, ...],
    strides: Tuple[int, ...]
) -> Tuple[numpy.ndarray]
```

Compute Average Pooling using Torch.

Currently supports 2d average pooling with torch semantics. This function is ONNX compatible.

See: https://github.com/onnx/onnx/blob/main/docs/Operators.md#AveragePool

Args ——= **x** : numpy.ndarray : input data (many dtypes are supported). Shape is N x C x H x W for 2d

ceil_mode : int ONNX rounding parameter, expected 0 (torch style dimension computation)
kernel_shape : Tuple[int] shape of the kernel. Should have 2 elements for 2d conv
pads : Tuple[int] padding in ONNX format (begin, end) on each axis
strides : Tuple[int] stride of the convolution on each axis

Returns ——= res (numpy.ndarray): a tensor of size (N x InChannels x OutHeight x OutWidth). See https://pytorch.org/docs/stable/generated/torch.nn.AvgPool2d.html

Raises ——= AssertionError : if the pooling arguments are wrong

Class `QuantizedBatchNormalization`

```
class QuantizedBatchNormalization(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Quantized Batch normalization with encrypted input and in-the-clear normalization params.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method `impl`

```
def impl(
    x: numpy.ndarray,
    scale: numpy.ndarray,
    bias: numpy.ndarray,
    input_mean: numpy.ndarray,
    input_var: numpy.ndarray,
    /,
    *,
    epsilon=1e-05,
    momentum=0.9,
    training_mode=0
) -> Tuple[numpy.ndarray]
```

Compute the batch normalization of the input tensor.

This can be expressed as:

Y = (X - input_mean) / sqrt(input_var + epsilon) * scale + B

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#BatchNormalization-14

Args ——= **x** : numpy.ndarray : tensor to normalize, dimensions are in the form of (N,C,D1,D2,...,Dn), where N is the batch size, C is the number of channels.

scale : numpy.ndarray scale tensor of shape (C,)
bias : numpy.ndarray bias tensor of shape (C,)

input_mean : numpy.ndarray mean values to use for each input channel, shape (C,)
input_var : numpy.ndarray variance values to use for each input channel, shape (C,)
epsilon : float avoids division by zero
momentum : float momentum used during training of the mean/variance, not used in inference
training_mode : int if the model was exported in training mode this is set to 1, else 0

Returns ——= numpy.ndarray : Normalized tensor

Class QuantizedCast

```
class QuantizedCast(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Cast the input to the required data type.

In FHE we only support a limited number of output types. Booleans are cast to integers.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method impl

```
def impl(
    data: numpy.ndarray,
    /,
    *,
    to: int
) -> Tuple[numpy.ndarray]
```

Execute ONNX cast in Numpy.

Supports only booleans for now, which are converted to integers.

See: https://github.com/onnx/onnx/blob/main/docs/Operators.md#Cast

Args ——= **data** : numpy.ndarray : Input encrypted tensor

to : int integer value of the onnx.TensorProto DataType enum

Returns ——= result (numpy.ndarray): a tensor with the required data type

Class QuantizedCelu

```
class QuantizedCelu(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Quantized Celu op.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method `impl`

```
def impl(
    x: numpy.ndarray,
    /,
    *,
    alpha: float = 1
) -> Tuple[numpy.ndarray]
```

Compute celu in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Celu-12

Args —–= **x** : numpy.ndarray : Input tensor

alpha : float Coefficient

Returns —–= Tuple[numpy.ndarray] : Output tensor

Class `QuantizedClip`

```
class QuantizedClip(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Quantized clip op.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method `impl`

```
def impl(
    a: numpy.ndarray,
    /,
    min=None,
    max=None
) -> Tuple[numpy.ndarray]
```

Compute clip in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Clip-13

Args —–= **a** : numpy.ndarray : Input tensor whose elements to be clipped.

min : [type], optional Minimum value, under which element is replaced by min. It must be a scalar(tensor of empty shape). Defaults to None.

max : [type], optional Maximum value, above which element is replaced by max. It must be a scalar(tensor of empty shape). Defaults to None.

Returns —–= Tuple[numpy.ndarray] : Output tensor with clipped input elements.

Class `QuantizedConv`

```
class QuantizedConv(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Quantized Conv op.

Construct the quantized convolution operator and retrieve parameters.

Args —–= **`n_bits_output`** : number of bits for the quantization of the outputs of this operator

`int_input_names` names of integer tensors that are taken as input for this operation
`constant_inputs` the weights and activations
`input_quant_opts` options for the input quantizer
`attrs` convolution options dilations (Tuple[int]): dilation of the kernel, default 1 on all dimensions. group (int): number of convolution groups, default 1 kernel_shape (Tuple[int]): shape of the kernel. Should have 2 elements for 2d conv pads (Tuple[int]): padding in ONNX format (begin, end) on each axis strides (Tuple[int]): stride of the convolution on each axis

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method `can_fuse`

```
def can_fuse(
    self
) -> bool
```

Determine if this op can be fused.

Conv operation can not be fused since it must be performed over integer tensors and it combines different elements of the input tensors.

Returns —–= bool : False, this operation can not be fused as it adds different encrypted integers

Method `impl`

```
def impl(
    x: numpy.ndarray,
    w: numpy.ndarray,
    b: numpy.ndarray,
    /,
    *,
    dilations: Tuple[int, ...],
    group: int = 1,
    kernel_shape: Tuple[int, ...],
    pads: Tuple[int, ...],
    strides: Tuple[int, ...]
) -> Tuple[numpy.ndarray]
```

Compute N-D convolution using Torch.

Currently supports 2d convolution with torch semantics. This function is also ONNX compatible.

See: https://github.com/onnx/onnx/blob/main/docs/Operators.md#Conv

Args ——= **x** : numpy.ndarray : input data (many dtypes are supported). Shape is N x C x H x W for 2d

w : numpy.ndarray weights tensor. Shape is (O x I x Kh x Kw) for 2d
b : numpy.ndarray, Optional bias tensor, Shape is (O,)
dilations : Tuple[int] dilation of the kernel, default 1 on all dimensions.
group : int number of convolution groups, default 1
kernel_shape : Tuple[int] shape of the kernel. Should have 2 elements for 2d conv
pads : Tuple[int] padding in ONNX format (begin, end) on each axis
strides : Tuple[int] stride of the convolution on each axis

Returns ——= res (numpy.ndarray): a tensor of size (N x OutChannels x OutHeight x OutWidth). See https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html

Raises ——= AssertionError : if the convolution arguments are wrong

Method `q_impl`

```
def q_impl(
    self,
    *q_inputs: src.concrete.ml.quantization.quantized_array.QuantizedArray,
    **attrs
) -> src.concrete.ml.quantization.quantized_array.QuantizedArray
```

Compute the quantized convolution between two quantized tensors.

Allows an optional quantized bias.

Args ——= **q_inputs** : input tuple, contains x (numpy.ndarray): input data. Shape is N x C x H x W for 2d w (numpy.ndarray): weights tensor. Shape is (O x I x Kh x Kw) for 2d b (numpy.ndarray, Optional): bias tensor, Shape is (O,)

attrs convolution options handled in constructor

Returns ——= res (QuantizedArray): result of the quantized integer convolution

Class `QuantizedDiv`

```
class QuantizedDiv(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Div operator /.

This operation is not really working as a quantized operation. It just works when things got fused, as in eg Act(x) = 1000 / (x + 42))

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method `can_fuse`

```
def can_fuse(
    self
) -> bool
```

Determine if this op can be fused.

Div can be fused and computed in float when a single integer tensor generates both the operands. For example in the formula: f(x) = x / (x + 1) where x is an integer tensor.

Returns ——= bool : Can fuse

Method `impl`

```
def impl(
    a: numpy.ndarray,
    b: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute div in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Div-14

Args ——= **`a` : numpy.ndarray** : Input tensor

`b` : numpy.ndarray Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Class `QuantizedElu`

```
class QuantizedElu(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Quantized Elu op.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method `impl`

```
def impl(
    x: numpy.ndarray,
    /,
    *,
    alpha: float = 1
) -> Tuple[numpy.ndarray]
```

Compute elu in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Elu-6

Args ——= **`x` : numpy.ndarray** : Input tensor

`alpha` : float Coefficient

Returns ——= Tuple[numpy.ndarray] : Output tensor

Class QuantizedErf

```
class QuantizedErf(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Quantized erf op.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method impl

```
def impl(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute erf in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Erf-13

Args —–= **x** : numpy.ndarray : Input tensor

Returns —–= Tuple[numpy.ndarray] : Output tensor

Class QuantizedExp

```
class QuantizedExp(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Quantized Exp op.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method impl

```
def impl(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute exponential in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Exp-13

Args —–= **x** : numpy.ndarray : Input tensor

Returns —–= Tuple[numpy.ndarray] : The exponential of the input tensor computed element-wise

Class `QuantizedFlatten`

```
class QuantizedFlatten(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Quantized flatten for encrypted inputs.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method `can_fuse`

```
def can_fuse(
    self
) -> bool
```

Determine if this op can be fused.

Flatten operation can not be fused since it must be performed over integer tensors.

Returns ——= bool : False, this operation can not be fused as it is manipulates integer tensors.

Method `impl`

```
def impl(
    x: numpy.ndarray,
    /,
    *,
    axis: int = 1
) -> Tuple[numpy.ndarray]
```

Flatten a tensor into a 2d array.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Flatten-13.

Args ——= **`x`** : numpy.ndarray : tensor to flatten

`axis : int` axis after which all dimensions will be flattened (axis=0 gives a 1D output)

Returns ——= result : flattened tensor

Method `q_impl`

```
def q_impl(
    self,
    *q_inputs: src.concrete.ml.quantization.quantized_array.QuantizedArray,
    **attrs
) -> src.concrete.ml.quantization.quantized_array.QuantizedArray
```

Flatten the input integer encrypted tensor.

Args ——= **`q_inputs`** : an encrypted integer tensor at index 0

`attrs` contains axis attribute

Returns ——= result (QuantizedArray): reshaped encrypted integer tensor

Class QuantizedGemm

```
class QuantizedGemm(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Quantized Gemm op.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Descendants

- src.concrete.ml.quantization.quantized_ops.QuantizedMatMul

Methods

Method can_fuse

```
def can_fuse(
    self
)
```

Determine if this op can be fused.

Gemm operation can not be fused since it must be performed over integer tensors and it combines different values of the input tensors.

Returns ——= bool : False, this operation can not be fused as it adds different encrypted integers

Method impl

```
def impl(
    a: numpy.ndarray,
    b: numpy.ndarray,
    /,
    c: Optional[numpy.ndarray] = None,
    *,
    alpha: float = 1,
    beta: float = 1,
    transA: int = 0,
    transB: int = 0
) -> Tuple[numpy.ndarray]
```

Compute Gemm in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Gemm-13

Args ——= **a** : numpy.ndarray : Input tensor A. The shape of A should be (M, K) if transA is 0, or (K, M) if transA is non-zero.

b : numpy.ndarray Input tensor B. The shape of B should be (K, N) if transB is 0, or (N, K) if transB is non-zero.

c : Optional[numpy.ndarray] Optional input tensor C. If not specified, the computation is done as if C is a scalar 0. The shape of C should be unidirectional broadcastable to (M, N). Defaults to None.

alpha : float Scalar multiplier for the product of input tensors A * B. Defaults to 1.

beta : float Scalar multiplier for input tensor C. Defaults to 1.

transA : int Whether A should be transposed. The type is kept as int as it's the type used by ONNX and it can easily be interpreted by python as a boolean. Defaults to 0.

transB : int Whether B should be transposed. The type is kept as int as it's the type used by ONNX and it can easily be interpreted by python as a boolean. Defaults to 0.

Returns ——= Tuple[numpy.ndarray] : The tuple containing the result tensor

Class QuantizedGreater

```
class QuantizedGreater(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Comparison operator >.

Only supports comparison with a constant.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method impl

```
def impl(
    x: numpy.ndarray,
    y: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute greater in numpy according to ONNX spec and cast outputs to floats.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Greater-13

Args ——= **x** : numpy.ndarray : Input tensor

y : numpy.ndarray Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Class QuantizedGreaterOrEqual

```
class QuantizedGreaterOrEqual(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Comparison operator >=.

Only supports comparison with a constant.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method impl

```
def impl(
    x: numpy.ndarray,
    y: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute greater or equal in numpy according to ONNX specs and cast outputs to floats.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#GreaterOrEqual-12

Args —–= **x** : numpy.ndarray : Input tensor

y : numpy.ndarray Input tensor

Returns —–= Tuple[numpy.ndarray] : Output tensor

Class QuantizedHardSigmoid

```
class QuantizedHardSigmoid(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Quantized HardSigmoid op.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method impl

```
def impl(
    x: numpy.ndarray,
    /,
    *,
    alpha: float = 0.2,
    beta: float = 0.5
) -> Tuple[numpy.ndarray]
```

Compute hardsigmoid in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#HardSigmoid-6

Args —–= **x** : numpy.ndarray : Input tensor

alpha : float Coefficient
beta : float Coefficient

Returns —–= Tuple[numpy.ndarray] : Output tensor

Class QuantizedHardSwish

```
class QuantizedHardSwish(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
```

```
)
```

Quantized Hardswish op.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method `impl`

```
def impl(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute hardswitch in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#hardswish-14

Args —–= **x** : numpy.ndarray : Input tensor

Returns —–= Tuple[numpy.ndarray] : Output tensor

Class `QuantizedIdentity`

```
class QuantizedIdentity(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Quantized Identity op.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method `impl`

```
def impl(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute identity in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Identity-14

Args —–= **x** : numpy.ndarray : Input tensor

Returns —–= Tuple[numpy.ndarray] : Output tensor

Class `QuantizedLeakyRelu`

```
class QuantizedLeakyRelu(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
```

```
        **attrs
    )
```

Quantized LeakyRelu op.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method impl

```
    def impl(
        x: numpy.ndarray,
        /,
        *,
        alpha: float = 0.01
    ) -> Tuple[numpy.ndarray]
```

Compute leakyrelu in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#LeakyRelu-6

Args —–= **x** : numpy.ndarray : Input tensor

alpha : float Coefficient

Returns —–= Tuple[numpy.ndarray] : Output tensor

Class QuantizedLess

```
    class QuantizedLess(
        n_bits_output: int,
        int_input_names: Set[str] = None,
        constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
        input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
        **attrs
    )
```

Comparison operator <.

Only supports comparison with a constant.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method impl

```
    def impl(
        x: numpy.ndarray,
        y: numpy.ndarray,
        /
    ) -> Tuple[numpy.ndarray]
```

Compute less in numpy according to ONNX spec and cast outputs to floats.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Less-13

Args —–= **x** : numpy.ndarray : Input tensor

y : numpy.ndarray Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Class `QuantizedLessOrEqual`

```
class QuantizedLessOrEqual(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Comparison operator <=.

Only supports comparison with a constant.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method `impl`

```
def impl(
    x: numpy.ndarray,
    y: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute less or equal in numpy according to ONNX spec and cast outputs to floats.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#LessOrEqual-12

Args ——= **x** : numpy.ndarray : Input tensor

y : numpy.ndarray Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Class `QuantizedLog`

```
class QuantizedLog(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Quantized Log op.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method `impl`

```
def impl(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute log in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Log-13

Args ——= **x** : numpy.ndarray : Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Class `QuantizedMatMul`

```
class QuantizedMatMul(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Quantized MatMul op.

Ancestors (in MRO)

- src.concrete.ml.quantization.quantized_ops.QuantizedGemm
- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method `impl`

```
def impl(
    a: numpy.ndarray,
    b: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute matmul in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#MatMul-13

Args ——= **a** : numpy.ndarray : N-dimensional matrix A

b : numpy.ndarray N-dimensional matrix B

Returns ——= Tuple[numpy.ndarray] : Matrix multiply results from A * B

Class `QuantizedMul`

```
class QuantizedMul(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Multiplication operator.

Only multiplies an encrypted tensor with a float constant for now. This operation will be fused to a (potentially larger) TLU.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method `can_fuse`

```
def can_fuse(
    self
) -> bool
```

Determine if this op can be fused.

Multiplication can be fused and computed in float when a single integer tensor generates both the operands. For example in the formula: f(x) = x * (x + 1) where x is an integer tensor.

Returns ——= bool : Can fuse

Method `impl`

```
def impl(
    a: numpy.ndarray,
    b: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute mul in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Mul-14

Args ——= **a** : numpy.ndarray : Input tensor

b : numpy.ndarray Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Class `QuantizedNot`

```
class QuantizedNot(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Quantized Not op.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method `impl`

```
def impl(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute not in numpy according to ONNX spec and cast outputs to floats.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Not-1

Args ——= **x** : numpy.ndarray : Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Class QuantizedOr

```
class QuantizedOr(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Or operator ||.

This operation is not really working as a quantized operation. It just works when things got fused, as in eg Act(x) = x || (x + 42))

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method can_fuse

```
def can_fuse(
    self
) -> bool
```

Determine if this op can be fused.

Or can be fused and computed in float when a single integer tensor generates both the operands. For example in the formula: f(x) = x || (x + 1) where x is an integer tensor.

Returns ——= bool : Can fuse

Method impl

```
def impl(
    a: numpy.ndarray,
    b: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute or in numpy according to ONNX spec and cast outputs to floats.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Or-7

Args ——= **a** : numpy.ndarray : Input tensor

b : numpy.ndarray Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Class QuantizedPRelu

```
class QuantizedPRelu(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Quantized PRelu op.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method `impl`

```
def impl(
    x: numpy.ndarray,
    slope: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute prelu in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#prelu-16

Args —–= **x** : numpy.ndarray : Input tensor

slope : numpy.ndarray Slope of PRelu

Returns —–= Tuple[numpy.ndarray] : Output tensor

Class `QuantizedPad`

```
class QuantizedPad(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Quantized Padding op.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method `can_fuse`

```
def can_fuse(
    self
) -> bool
```

Determine if this op can be fused.

Pad operation can not be fused since it must be performed over integer tensors.

Returns —–= bool : False, this operation can not be fused as it is manipulates integer tensors

Method `impl`

```
def impl(
    data: numpy.ndarray,
    pads: numpy.ndarray,
    constant_value: Optional[numpy.ndarray] = None,
    /,
    *,
    mode: str
) -> Tuple[numpy.ndarray]
```

Apply padding in numpy according to ONNX spec.

See: https://github.com/onnx/onnx/blob/main/docs/Operators.md#Pad

Args ——= **data** : numpy.ndarray : Input variable/tensor to pad

pads : numpy.ndarray List of pads (size 8) to apply, two per N,C,H,W dimension
constant_value : float Constant value to use for padding
mode : str padding mode: constant/edge/reflect

Returns ——= res (numpy.ndarray): Padded tensor

Class QuantizedPow

```
class QuantizedPow(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Quantized pow op.

Only works for a float constant power. This operation will be fused to a (potentially larger) TLU.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method can_fuse

```
def can_fuse(
    self
) -> bool
```

Determine if this op can be fused.

Power raising can be fused and computed in float when a single integer tensor generates both the operands. For example in the formula: f(x) = x ** (x + 1) where x is an integer tensor.

Returns ——= bool : Can fuse

Method impl

```
def impl(
    a: numpy.ndarray,
    b: numpy.ndarray
) -> Tuple[numpy.ndarray]
```

Compute pow in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Pow-13

Args ——= **a** : numpy.ndarray : Input tensor whose elements to be raised.

b : numpy.ndarray The power to which we want to raise.

Returns ——= Tuple[numpy.ndarray] : Output tensor.

Class QuantizedReduceSum

```
class QuantizedReduceSum(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: Optional[src.concrete.ml.quantization.quantized_array.QuantizationOptions]
    **attrs
)
```

ReduceSum with encrypted input.

Construct the quantized ReduceSum operator and retrieve parameters.

Args ——= **n_bits_output** : int : Number of bits for the operator's quantization of outputs.

int_input_names : Optional[Set[str]] Names of input integer tensors. Default to None.

constant_inputs : Optional[Dict] Input constant tensor. axes (Optional[numpy.ndarray]): Array of integers along which to reduce. The default is to reduce over all the dimensions of the input tensor if 'noop_with_empty_axes' is false, else act as an Identity op when 'noop_with_empty_axes' is true. Accepted range is [-r, r-1] where r = rank(data). Default to None.

input_quant_opts : Optional[QuantizationOptions] Options for the input quantizer. Default to None.

attrs : dict RecuseSum options. keepdims (int): Keep the reduced dimension or not, 1 means keeping the input dimension, 0 will reduce it along the given axis. Default to 1. noop_with_empty_axes (int): Defines behaviour if 'axes' is empty or set to None. Default behaviour with 0 is to reduce all axes. When axes is empty and this attribute is set to true 1, input tensor will not be reduced, and the output tensor would be equivalent to input tensor. Default to 0.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method impl

```
def impl(
    a: numpy.ndarray,
    /,
    axes: Optional[numpy.ndarray] = None,
    *,
    keepdims: int = 1,
    noop_with_empty_axes: int = 0
) -> Tuple[numpy.ndarray]
```

Compute ReduceSum in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Operators.md#ReduceSum

Args ——= **a** : numpy.ndarray : Input tensor whose elements to sum.

axes : Optional[numpy.ndarray] Array of integers along which to reduce. The default is to reduce over all the dimensions of the input tensor if 'noop_with_empty_axes' is false, else act as an Identity op when 'noop_with_empty_axes' is true. Accepted range is [-r, r-1] where r = rank(data). Default to None.

keepdims : int Keep the reduced dimension or not, 1 means keeping the input dimension, 0 will reduce it along the given axis. Default to 1.

noop_with_empty_axes : int Defines behaviour if 'axes' is empty or set to None. Default behaviour with 0 is to reduce all axes. When axes is empty and this attribute is set to true 1, input tensor will not be reduced, and the output tensor would be equivalent to input tensor. Default to 0.

Returns ——= numpy.ndarray : Output reduced tensor.

Method q_impl

```
def q_impl(
    self,
    *q_inputs: src.concrete.ml.quantization.quantized_array.QuantizedArray,
    **attrs
) -> src.concrete.ml.quantization.quantized_array.QuantizedArray
```

Sum the encrypted tensor's values over axis 1.

Args —–= **q_inputs** : QuantizedArray : An encrypted integer tensor at index 0.

attrs : Dict Contains axis attribute.

Returns —–= (QuantizedArray): The sum of all values along axis 1 as an encrypted integer tensor.

Class QuantizedRelu

```
class QuantizedRelu(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Quantized Relu op.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method impl

```
def impl(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute relu in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Relu-14

Args —–= **x** : numpy.ndarray : Input tensor

Returns —–= Tuple[numpy.ndarray] : Output tensor

Class QuantizedReshape

```
class QuantizedReshape(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Quantized Reshape op.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method `impl`

```
def impl(
    x: numpy.ndarray,
    newshape: numpy.ndarray,
    /,
    *,
    allowzero=0
) -> Tuple[numpy.ndarray]
```

Compute reshape in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Reshape-13

Args ——= **x** : numpy.ndarray : Input tensor

newshape : numpy.ndarray New shape
allowzero : int ONNX legacy parameter, by default 0 -> behave like numpy reshape

Returns ——= Tuple[numpy.ndarray] : Output tensor

Method `q_impl`

```
def q_impl(
    self,
    *q_inputs: src.concrete.ml.quantization.quantized_array.QuantizedArray,
    **attrs
) -> src.concrete.ml.quantization.quantized_array.QuantizedArray
```

Reshape the input integer encrypted tensor.

Args ——= **q_inputs** : an encrypted integer tensor at index 0 and one constant shape at index 1

attrs additional optional reshape options

Returns ——= result (QuantizedArray): reshaped encrypted integer tensor

Class `QuantizedRound`

```
class QuantizedRound(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Quantized round op.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method `impl`

```
def impl(
    a: numpy.ndarray
) -> Tuple[numpy.ndarray]
```

Compute round in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Round-11 Remark that ONNX Round operator is actually a rint, since the number of decimals is forced to be 0

Args ——= **a** : numpy.ndarray : Input tensor whose elements to be rounded.

Returns ——= Tuple[numpy.ndarray] : Output tensor with rounded input elements.

Class QuantizedSelu

```
class QuantizedSelu(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Quantized Selu op.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method impl

```
def impl(
    x: numpy.ndarray,
    /,
    *,
    alpha: float = 1.6732632423543772,
    gamma: float = 1.0507009873554805
) -> Tuple[numpy.ndarray]
```

Compute selu in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Selu-6

Args ——= **x** : numpy.ndarray : Input tensor

alpha : float Coefficient
gamma : float Coefficient

Returns ——= Tuple[numpy.ndarray] : Output tensor

Class QuantizedSigmoid

```
class QuantizedSigmoid(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Quantized sigmoid op.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method `impl`

```
def impl(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute sigmoid in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Sigmoid-13

Args ——= **x** : numpy.ndarray : Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Class `QuantizedSoftplus`

```
class QuantizedSoftplus(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Quantized Softplus op.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method `impl`

```
def impl(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute softplus in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Softplus-1

Args ——= **x** : numpy.ndarray : Input tensor

Returns ——= Tuple[numpy.ndarray] : Output tensor

Class `QuantizedSub`

```
class QuantizedSub(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Subtraction operator.

This works the same as addition on both encrypted - encrypted and on encrypted - constant.

Ancestors (in MRO)

- src.concrete.ml.quantization.quantized_ops.QuantizedAdd
- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Class variables

Variable `b_sign` Type: `int`

Methods

Method `impl`

```
def impl(
    a: numpy.ndarray,
    b: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute sub in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Sub-14

Args —–= **a** : numpy.ndarray : Input tensor

b : numpy.ndarray Input tensor

Returns —–= Tuple[numpy.ndarray] : Output tensor

Class `QuantizedTanh`

```
class QuantizedTanh(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Quantized Tanh op.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method `impl`

```
def impl(
    x: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute tanh in numpy according to ONNX spec.

See https://github.com/onnx/onnx/blob/main/docs/Changelog.md#Tanh-13

Args —–= **x** : numpy.ndarray : Input tensor

Returns —–= Tuple[numpy.ndarray] : Output tensor

Class `QuantizedWhere`

```
class QuantizedWhere(
    n_bits_output: int,
    int_input_names: Set[str] = None,
    constant_inputs: Union[Dict[str, Any], Dict[int, Any], None] = None,
    input_quant_opts: src.concrete.ml.quantization.quantized_array.QuantizationOptions = None,
    **attrs
)
```

Where operator on quantized arrays.

Supports only constants for the results produced on the True/False branches.

Ancestors (in MRO)

- src.concrete.ml.quantization.base_quantized_op.QuantizedOp

Methods

Method `impl`

```
def impl(
    c: numpy.ndarray,
    t: numpy.ndarray,
    f: numpy.ndarray,
    /
) -> Tuple[numpy.ndarray]
```

Compute the equivalent of numpy.where.

Args —–= `c` : numpy.ndarray : Condition operand.

`t` : numpy.ndarray True operand.
`f` : numpy.ndarray False operand.

Returns —–= numpy.ndarray : numpy.where(c, t, f)

Module `src.concrete.ml.sklearn`

Import sklearn models.

Sub-modules

- src.concrete.ml.sklearn.base
- src.concrete.ml.sklearn.glm
- src.concrete.ml.sklearn.linear_model
- src.concrete.ml.sklearn.qnn
- src.concrete.ml.sklearn.rf
- src.concrete.ml.sklearn.svm
- src.concrete.ml.sklearn.torch_module
- src.concrete.ml.sklearn.tree
- src.concrete.ml.sklearn.tree_to_numpy
- src.concrete.ml.sklearn.xgb

Module `src.concrete.ml.sklearn.base`

Module that contains base classes for our libraries estimators.

Classes

Class `BaseTreeEstimatorMixin`

```
class BaseTreeEstimatorMixin(
    n_bits: int
)
```

Mixin class for tree-based estimators.

This class is used to add functionality to tree-based estimators, such as the tree-based classifier.

Initialize the TreeBasedEstimatorMixin.

Args —–= **`n_bits`** : int : number of bits used for quantization

Ancestors (in MRO)

- sklearn.base.BaseEstimator

Descendants

- src.concrete.ml.sklearn.rf.RandomForestClassifier
- src.concrete.ml.sklearn.tree.DecisionTreeClassifier
- src.concrete.ml.sklearn.xgb.XGBClassifier

Class variables

Variable `class_mapping_` Type: `Optional[dict]`

Variable `classes_` Type: `str`

Variable `framework` Type: `str`

Variable `init_args` Type: `Dict[str, Any]`

Variable `input_quantizers` Type: `List[concrete.ml.quantization.quantized_array.UniformQuantizer]`

Variable `n_bits` Type: `int`

Variable `n_classes_` Type: `int`

Variable `output_quantizers` Type: `List[concrete.ml.quantization.quantized_array.UniformQuantizer]`

Variable `random_state` Type: `Union[numpy.random.mtrand.RandomState, int, None]`

Variable `sklearn_alg` Type: `Any`

Variable `sklearn_model` Type: `Any`

Instance variables

Variable `onnx_model` Type: `onnx.onnx_ml_pb2.ModelProto`

Get the ONNX model.

.. # noqa: DAR201

Returns —–= onnx.ModelProto : the ONNX model

Methods

Method `compile`

```
def compile(
    self,
    X: numpy.ndarray,
    configuration: Optional[concrete.numpy.compilation.configuration.Configuration] = None,
    compilation_artifacts: Optional[concrete.numpy.compilation.artifacts.DebugArtifacts] = None
    show_mlir: bool = False,
    use_virtual_lib: bool = False,
    p_error: Optional[float] = 6.3342483999973e-05
) -> concrete.numpy.compilation.circuit.Circuit
```

Compile the model.

Args -----= **`X`** : numpy.ndarray : the unquantized dataset

`configuration` : Optional[Configuration] the options for compilation
`compilation_artifacts` : Optional[DebugArtifacts] artifacts object to fill during compilation
`show_mlir` : bool whether or not to show MLIR during the compilation
`use_virtual_lib` : bool set to True to use the so called virtual lib simulating FHE computation. Defaults to False
`p_error` : Optional[float] probability of error of a PBS

Returns -----= Circuit : the compiled Circuit.

Method `fit`

```
def fit(
    self,
    X,
    y: numpy.ndarray,
    **kwargs
) -> Any
```

Fit the tree-based estimator.

Args -----= X : training data By default, you should be able to pass: * numpy arrays * torch tensors * pandas DataFrame or Series **`y`** : numpy.ndarray : The target data.

`**kwargs` args for super().fit

Returns -----= Any : The fitted model.

Method `fit_benchmark`

```
def fit_benchmark(
    self,
    X: numpy.ndarray,
    y: numpy.ndarray,
    *args,
    random_state: Optional[int] = None,
    **kwargs
) -> Tuple[Any, Any]
```

Fit the sklearn tree-based model and the FHE tree-based model.

Args -----= **`X`** : numpy.ndarray : The input data.

`y` : numpy.ndarray The target data.

random_state (Optional[Union[int, numpy.random.RandomState, None]]): The random state. Defaults to None. **`*args`** : args for super().fit

`**kwargs` kwargs for super().fit

Returns -----= Tuple[ConcreteEstimators, SklearnEstimators]: The FHE and sklearn tree-based models.

Method `post_processing`

```
def post_processing(
    self,
    y_preds: numpy.ndarray
) -> numpy.ndarray
```

Apply post-processing to the predictions.

Args —–= **`y_preds`** : numpy.ndarray : The predictions.

Returns —–= numpy.ndarray : The post-processed predictions.

Method `predict`

```
def predict(
    self,
    X: numpy.ndarray,
    execute_in_fhe: bool = False
) -> numpy.ndarray
```

Predict the target values.

Args —–= **`X`** : numpy.ndarray : The input data.

`execute_in_fhe` : bool Whether to execute in FHE. Defaults to False.

Returns —–= numpy.ndarray : The predicted target values.

Method `predict_proba`

```
def predict_proba(
    self,
    X: numpy.ndarray,
    execute_in_fhe: bool = False
) -> numpy.ndarray
```

Predict the probabilities.

Args —–= **`X`** : numpy.ndarray : The input data.

`execute_in_fhe` : bool Whether to execute in FHE. Defaults to False.

Returns —–= numpy.ndarray : The predicted probabilities.

Method `quantize_input`

```
def quantize_input(
    self,
    X: numpy.ndarray
)
```

Quantize the input.

Args —–= **`X`** : numpy.ndarray : the input

Returns —–= the quantized input

Method `update_post_processing_params`

```
def update_post_processing_params(
    self
)
```

Update the post processing parameters.

Class `QuantizedTorchEstimatorMixin`

```
class QuantizedTorchEstimatorMixin(
    *args,
    **kwargs
)
```

Mixin that provides quantization for a torch module and follows the Estimator API.

This class should be mixed in with another that provides the full Estimator API. This class only provides modifiers for .fit() (with quantization) and .predict() (optionally in FHE)

Descendants

- src.concrete.ml.sklearn.qnn.QuantizedSkorchEstimatorMixin

Class variables

Variable `post_processing_params` Type: `Dict[str, Any]`

Instance variables

Variable `base_estimator_type` Get the sklearn estimator that should be trained by the child class.

Variable `base_module_to_compile` Get the Torch module that should be compiled to FHE.

Variable `fhe_circuit` Type: `concrete.numpy.compilation.circuit.Circuit`

Get the FHE circuit.

Returns ——= Circuit : the FHE circuit

Variable `input_quantizers` Type: `List[src.concrete.ml.quantization.quantized_array.QuantizedArray]`

Get the input quantizers.

Returns ——= List[QuantizedArray] : the input quantizers

Variable `n_bits_quant` Get the number of quantization bits.

Variable `onnx_model` Get the ONNX model.

.. # noqa: DAR201

Returns ——= *onnx_model* (onnx.ModelProto): the ONNX model

Variable `output_quantizers` Type: `List[src.concrete.ml.quantization.quantized_array.QuantizedArray]`

Get the input quantizers.

Returns ——= List[QuantizedArray] : the input quantizers

Variable `quantize_input` Type: `Callable`

Get the input quantization function.

Returns ——= Callable : function that quantizes the input

Methods

Method `compile`

```
def compile(
    self,
    X: numpy.ndarray,
    configuration: Optional[concrete.numpy.compilation.configuration.Configuration] = None,
    compilation_artifacts: Optional[concrete.numpy.compilation.artifacts.DebugArtifacts] = None
    show_mlir: bool = False,
    use_virtual_lib: bool = False,
    p_error: Optional[float] = 6.3342483999973e-05
) -> concrete.numpy.compilation.circuit.Circuit
```

Compile the model.

Args —–= **`X`** : numpy.ndarray : the unquantized dataset

`configuration` : Optional[Configuration] the options for compilation
`compilation_artifacts` : Optional[DebugArtifacts] artifacts object to fill during compilation
`show_mlir` : bool whether or not to show MLIR during the compilation
`use_virtual_lib` : bool whether to compile using the virtual library that allows higher bitwidths
`p_error` : Optional[float] probability of error of a PBS

Returns —–= Circuit : the compiled Circuit.

Raises —–= ValueError : if called before the model is trained

Method `fit`

```
def fit(
    self,
    X,
    y,
    **fit_params
)
```

Initialize and fit the module.

If the module was already initialized, by calling fit, the module will be re-initialized (unless warm_start is True). In addition to the torch training step, this method performs quantization of the trained torch model.

Args —–= X : training data By default, you should be able to pass: * numpy arrays * torch tensors * pandas DataFrame or Series **`y`** : numpy.ndarray : labels associated with training data

`**fit_params` additional parameters that can be used during training, these are passed to the torch training interface

Returns —–= self : the trained quantized estimator

Method `fit_benchmark`

```
def fit_benchmark(
    self,
    X,
    y,
    *args,
    **kwargs
)
```

Fit the quantized estimator and return reference estimator.

This function returns both the quantized estimator (itself), but also a wrapper around the non-quantized trained NN. This is useful in order to compare performance between the quantized and fp32 versions of the classifier

Args ——= X : training data By default, you should be able to pass: * numpy arrays * torch tensors * pandas DataFrame or Series **`y`** : numpy.ndarray : labels associated with training data

`*args` The arguments to pass to the sklearn linear model.
`**kwargs` The keyword arguments to pass to the sklearn linear model.

Returns ——= self : the trained quantized estimator

fp32_model trained raw (fp32) wrapped NN estimator

Method `get_params_for_benchmark`

```
def get_params_for_benchmark(
    self
)
```

Get the parameters to instantiate the sklearn estimator trained by the child class.

Returns ——= params (dict): dictionary with parameters that will initialize a new Estimator

Method `post_processing`

```
def post_processing(
    self,
    y_preds: numpy.ndarray
) -> numpy.ndarray
```

Post-processing the output.

Args ——= **`y_preds`** : numpy.ndarray : the output to post-process

Raises ——= ValueError : if unknown post-processing function

Returns ——= numpy.ndarray : the post-processed output

Method `predict`

```
def predict(
    self,
    X,
    execute_in_fhe=False
)
```

Predict on user provided data.

Predicts using the quantized clear or FHE classifier

Args ——= X : input data, a numpy array of raw values (non quantized) execute_in_fhe : whether to execute the inference in FHE or in the clear

Returns ——= y_pred : numpy ndarray with predictions

Method `predict_proba`

```
def predict_proba(
    self,
    X,
    execute_in_fhe=False
)
```

Predict on user provided data, returning probabilities.

Predicts using the quantized clear or FHE classifier

Args ——= X : input data, a numpy array of raw values (non quantized) execute_in_fhe : whether to execute the inference in FHE or in the clear

Returns ——= y_pred : numpy ndarray with probabilities (if applicable)

Raises ——= ValueError : if the estimator was not yet trained or compiled

Method `update_post_processing_params`

```
def update_post_processing_params(
    self
)
```

Update the post-processing parameters.

Class `SklearnLinearModelMixin`

```
class SklearnLinearModelMixin(
    *args,
    n_bits: Union[int, Dict[~KT, ~VT]] = 2,
    **kwargs
)
```

A Mixin class for sklearn linear models with FHE.

Initialize the FHE linear model.

Args ——= **`n_bits`** : int, Dict : Number of bits to quantize the model. If an int is passed for n_bits, the value will be used for activation, inputs and weights. If a dict is passed, then it should contain "net_inputs", "op_inputs", "op_weights" and "net_outputs" keys with corresponding number of quantization bits for: - net_inputs : number of bits for model input - op_inputs : number of bits to quantize layer input values - op_weights: learned parameters or constants in the network - net_outputs: final model output quantization bits Default to 2.

`*args` The arguments to pass to the sklearn linear model.
`**kwargs` The keyword arguments to pass to the sklearn linear model.

Ancestors (in MRO)

- sklearn.base.BaseEstimator

Descendants

- src.concrete.ml.sklearn.glm._GeneralizedLinearRegressor
- src.concrete.ml.sklearn.linear_model.LinearRegression
- src.concrete.ml.sklearn.linear_model.LogisticRegression
- src.concrete.ml.sklearn.svm.LinearSVC
- src.concrete.ml.sklearn.svm.LinearSVR

Class variables

Variable `random_state` Type: `Union[numpy.random.mtrand.RandomState, int, None]`

Variable `sklearn_alg` Type: `Callable`

Instance variables

Variable `fhe_circuit` Type: `concrete.numpy.compilation.circuit.Circuit`

Get the FHE circuit.

Returns ——= Circuit : the FHE circuit

Variable `input_quantizers` Type: `List[src.concrete.ml.quantization.quantized_array.QuantizedArray]`

Get the input quantizers.

Returns ——= List[QuantizedArray] : the input quantizers

Variable `onnx_model` Type: `onnx.onnx_ml_pb2.ModelProto`

Get the ONNX model.

.. # noqa: DAR201

Returns —–= onnx.ModelProto : the ONNX model

Variable `output_quantizers` Type: `List[src.concrete.ml.quantization.quantized_array.QuantizedArray]`

Get the input quantizers.

Returns —–= List[QuantizedArray] : the input quantizers

Variable `quantize_input` Type: `Callable`

Get the input quantization function.

Returns —–= Callable : function that quantizes the input

Methods

Method `clean_graph`

```
def clean_graph(
    self,
    onnx_model: onnx.onnx_ml_pb2.ModelProto
)
```

Clean the graph of the onnx model.

This will remove the Cast node in the onnx.graph since they have no use in the quantized/FHE model.

Args —–= **`onnx_model`** : onnx.ModelProto : the onnx model

Returns —–= onnx.ModelProto : the cleaned onnx model

Method `compile`

```
def compile(
    self,
    X: numpy.ndarray,
    configuration: Optional[concrete.numpy.compilation.configuration.Configuration] = None,
    compilation_artifacts: Optional[concrete.numpy.compilation.artifacts.DebugArtifacts] = None
    show_mlir: bool = False,
    use_virtual_lib: bool = False,
    p_error: Optional[float] = 6.3342483999973e-05
) -> concrete.numpy.compilation.circuit.Circuit
```

Compile the FHE linear model.

Args —–= **`X`** : numpy.ndarray : The input data.

`configuration` : Optional[Configuration] Configuration object to use during compilation
`compilation_artifacts` : Optional[DebugArtifacts] Artifacts object to fill during compilation
`show_mlir` : bool if set, the MLIR produced by the converter and which is going to be sent to the compiler backend is shown on the screen, e.g., for debugging or demo. Defaults to False.
`use_virtual_lib` : bool whether to compile using the virtual library that allows higher bitwidths with simulated FHE computation. Defaults to False
`p_error` : Optional[float] probability of error of a PBS

Returns —–= Circuit : the compiled Circuit.

Method fit

```
def fit(
    self,
    X,
    y: numpy.ndarray,
    *args,
    **kwargs
) -> None
```

Fit the FHE linear model.

Args —–= X : training data By default, you should be able to pass: * numpy arrays * torch tensors * pandas DataFrame or Series **y** : numpy.ndarray : The target data.

***args** The arguments to pass to the sklearn linear model.
****kwargs** The keyword arguments to pass to the sklearn linear model.

Method fit_benchmark

```
def fit_benchmark(
    self,
    X: numpy.ndarray,
    y: numpy.ndarray,
    *args,
    random_state: Optional[int] = None,
    **kwargs
) -> Tuple[Any, Any]
```

Fit the sklearn linear model and the FHE linear model.

Args —–= **X** : numpy.ndarray : The input data.

y : numpy.ndarray The target data.

random_state (Optional[Union[int, numpy.random.RandomState, None]]): The random state. Defaults to None. ***args** : args for super().fit

****kwargs** kwargs for super().fit

Returns —–= Tuple[SklearnLinearModelMixin, sklearn.linear_model.LinearRegression]: The FHE and sklearn LinearRegression.

Method post_processing

```
def post_processing(
    self,
    y_preds: numpy.ndarray
) -> numpy.ndarray
```

Post-processing the output.

Args —–= **y_preds** : numpy.ndarray : the output to post-process

Returns —–= numpy.ndarray : the post-processed output

Method predict

```
def predict(
    self,
    X: numpy.ndarray,
    execute_in_fhe: bool = False
) -> numpy.ndarray
```

Predict on user data.

Predict on user data using either the quantized clear model, implemented with tensors, or, if execute_in_fhe is set, using the compiled FHE circuit

Args ——= **X** : numpy.ndarray : the input data

execute_in_fhe : bool whether to execute the inference in FHE

Returns ——= numpy.ndarray : the prediction as ordinals

Module `src.concrete.ml.sklearn.glm`

Implement sklearn's Generalized Linear Models (GLM).

Classes

Class `GammaRegressor`

```
class GammaRegressor(
    *,
    n_bits: Union[int, dict] = 2,
    alpha: float = 1.0,
    fit_intercept: bool = True,
    max_iter: int = 100,
    tol: float = 0.0001,
    warm_start: bool = False,
    verbose: int = 0
)
```

A Gamma regression model with FHE.

Initialize the FHE linear model.

Args ——= **n_bits** : int, Dict : Number of bits to quantize the model. If an int is passed for n_bits, the value will be used for activation, inputs and weights. If a dict is passed, then it should contain "net_inputs", "op_inputs", "op_weights" and "net_outputs" keys with corresponding number of quantization bits for: - net_inputs : number of bits for model input - op_inputs : number of bits to quantize layer input values - op_weights: learned parameters or constants in the network - net_outputs: final model output quantization bits Default to 2.

***args** The arguments to pass to the sklearn linear model.
****kwargs** The keyword arguments to pass to the sklearn linear model.

Ancestors (in MRO)

- src.concrete.ml.sklearn.glm._GeneralizedLinearRegressor
- src.concrete.ml.sklearn.base.SklearnLinearModelMixin
- sklearn.base.BaseEstimator
- sklearn.base.RegressorMixin

Class variables

Variable `random_state` Type: `Union[numpy.random.mtrand.RandomState, int, None]`

Variable `sklearn_alg` Type: `Callable`

Generalized Linear Model with a Gamma distribution.

This regressor uses the 'log' link function.

Read more in the :ref:`User Guide <Generalized_linear_regression>`.

Added in version: 0.23:

Parameters

alpha : float, default=1 Constant that multiplies the penalty term and thus determines the regularization strength. `alpha = 0` is equivalent to unpenalized GLMs. In this case, the design matrix X must have full column rank (no collinearities). Values must be in the range [0.0, inf).

fit_intercept : bool, default=True Specifies if a constant (a.k.a. bias or intercept) should be added to the linear predictor (X @ coef + intercept).

max_iter : int, default=100 The maximal number of iterations for the solver. Values must be in the range [1, inf).

tol : float, default=1e-4 Stopping criterion. For the lbfgs solver, the iteration will stop when `max{|g_j|, j = 1, ..., d} <= tol` where g_j is the j-th component of the gradient (derivative) of the objective function. Values must be in the range (0.0, inf).

warm_start : bool, default=False If set to True, reuse the solution of the previous call to fit as initialization for coef_ and intercept_ .

verbose : int, default=0 For the lbfgs solver set verbose to any positive number for verbosity. Values must be in the range [0, inf).

Attributes

coef_ : array of shape (n_features,) Estimated coefficients for the linear predictor (`X * coef_ + intercept_`) in the GLM.

intercept_ : float Intercept (a.k.a. bias) added to linear predictor.

n_features_in_ : int Number of features seen during :term:fit.

Added in version: 0.24:

n_iter_ : int Actual number of iterations used in the solver.

feature_names_in_ : ndarray of shape (n_features_in_,) Names of features seen during :term:fit. Defined only when X has feature names that are all strings.

Added in version: 1.0:

See Also

PoissonRegressor Generalized Linear Model with a Poisson distribution.
TweedieRegressor Generalized Linear Model with a Tweedie distribution.

Examples

```
>>> from sklearn import linear_model
>>> clf = linear_model.GammaRegressor()
>>> X = [[1, 2], [2, 3], [3, 4], [4, 3]]
>>> y = [19, 26, 33, 30]
>>> clf.fit(X, y)
GammaRegressor()
>>> clf.score(X, y)
0.773...
>>> clf.coef_
array([0.072..., 0.066...])
>>> clf.intercept_
2.896...
>>> clf.predict([[1, 0], [2, 8]])
array([19.483..., 35.795...])
```

Class PoissonRegressor

```
class PoissonRegressor(
    *,
    n_bits: Union[int, dict] = 2,
    alpha: float = 1.0,
    fit_intercept: bool = True,
    max_iter: int = 100,
```

```
        tol: float = 0.0001,
        warm_start: bool = False,
        verbose: int = 0
    )
```

A Poisson regression model with FHE.

Initialize the FHE linear model.

Args ——= **n_bits** : int, Dict : Number of bits to quantize the model. If an int is passed for n_bits, the value will be used for activation, inputs and weights. If a dict is passed, then it should contain "net_inputs", "op_inputs", "op_weights" and "net_outputs" keys with corresponding number of quantization bits for: - net_inputs : number of bits for model input - op_inputs : number of bits to quantize layer input values - op_weights: learned parameters or constants in the network - net_outputs: final model output quantization bits Default to 2.

`*args` The arguments to pass to the sklearn linear model.
`**kwargs` The keyword arguments to pass to the sklearn linear model.

Ancestors (in MRO)

- src.concrete.ml.sklearn.glm._GeneralizedLinearRegressor
- src.concrete.ml.sklearn.base.SklearnLinearModelMixin
- sklearn.base.BaseEstimator
- sklearn.base.RegressorMixin

Class variables

Variable `random_state` Type: `Union[numpy.random.mtrand.RandomState, int, None]`

Variable `sklearn_alg` Type: `Callable`

Generalized Linear Model with a Poisson distribution.

This regressor uses the 'log' link function.

Read more in the :ref:User Guide <Generalized_linear_regression>.

Added in version: 0.23:

Parameters

`alpha` : float, default=1 Constant that multiplies the penalty term and thus determines the regularization strength. `alpha = 0` is equivalent to unpenalized GLMs. In this case, the design matrix X must have full column rank (no collinearities). Values must be in the range [0.0, inf).
`fit_intercept` : bool, default=True Specifies if a constant (a.k.a. bias or intercept) should be added to the linear predictor (X @ coef + intercept).
`max_iter` : int, default=100 The maximal number of iterations for the solver. Values must be in the range [1, inf).
`tol` : float, default=`1e-4` Stopping criterion. For the lbfgs solver, the iteration will stop when `max{|g_j|, j = 1, ..., d} <= tol` where g_j is the j-th component of the gradient (derivative) of the objective function. Values must be in the range (0.0, inf).
`warm_start` : bool, default=False If set to True, reuse the solution of the previous call to fit as initialization for coef_ and intercept_ .
`verbose` : int, default=0 For the lbfgs solver set verbose to any positive number for verbosity. Values must be in the range [0, inf).

Attributes

`coef_` : array of shape (n_features,) Estimated coefficients for the linear predictor (`X @ coef_ + intercept_`) in the GLM.

`intercept_` : float Intercept (a.k.a. bias) added to linear predictor.

`n_features_in_` : int Number of features seen during :term:fit.

Added in version: 0.24:

`feature_names_in_` : ndarray of shape (n_features_in_,) Names of features seen during :term:fit. Defined only when X has feature names that are all strings.

Added in version: 1.0:

`n_iter_` : int Actual number of iterations used in the solver.

See Also

TweedieRegressor Generalized Linear Model with a Tweedie distribution.

Examples

```
>>> from sklearn import linear_model
>>> clf = linear_model.PoissonRegressor()
>>> X = [[1, 2], [2, 3], [3, 4], [4, 3]]
>>> y = [12, 17, 22, 21]
>>> clf.fit(X, y)
PoissonRegressor()
>>> clf.score(X, y)
0.990...
>>> clf.coef_
array([0.121..., 0.158...])
>>> clf.intercept_
2.088...
>>> clf.predict([[1, 1], [3, 4]])
array([10.676..., 21.875...])
```

Class `TweedieRegressor`

```
class TweedieRegressor(
    *,
    n_bits: Union[int, dict] = 2,
    power: float = 0.0,
    alpha: float = 1.0,
    fit_intercept: bool = True,
    link: str = 'auto',
    max_iter: int = 100,
    tol: float = 0.0001,
    warm_start: bool = False,
    verbose: int = 0
)
```

A Tweedie regression model with FHE.

Initialize the FHE linear model.

Args ——= **`n_bits`** : int, Dict : Number of bits to quantize the model. If an int is passed for n_bits, the value will be used for activation, inputs and weights. If a dict is passed, then it should contain "net_inputs", "op_inputs", "op_weights" and "net_outputs" keys with corresponding number of quantization bits for: - net_inputs : number of bits for model input - op_inputs : number of bits to quantize layer input values - op_weights: learned parameters or constants in the network - net_outputs: final model output quantization bits Default to 2.

`*args` The arguments to pass to the sklearn linear model.
`**kwargs` The keyword arguments to pass to the sklearn linear model.

Ancestors (in MRO)

- src.concrete.ml.sklearn.glm._GeneralizedLinearRegressor
- src.concrete.ml.sklearn.base.SklearnLinearModelMixin

- sklearn.base.BaseEstimator
- sklearn.base.RegressorMixin

Class variables

Variable `random_state` Type: `Union[numpy.random.mtrand.RandomState, int, None]`

Variable `sklearn_alg` Type: `Callable`

Generalized Linear Model with a Tweedie distribution.

This estimator can be used to model different GLMs depending on the power parameter, which determines the underlying distribution.

Read more in the :ref:`User Guide <Generalized_linear_regression>`.

Added in version: 0.23:

Parameters

`power` : float, default=0 The power determines the underlying target distribution according to the following table:

```
+-------+------------------------+
| Power | Distribution           |
+=======+========================+
| 0     | Normal                 |
+-------+------------------------+
| 1     | Poisson                |
+-------+------------------------+
| (1,2) | Compound Poisson Gamma |
+-------+------------------------+
| 2     | Gamma                  |
+-------+------------------------+
| 3     | Inverse Gaussian       |
+-------+------------------------+

For ``0 < power < 1``, no distribution exists.
```

`alpha` : float, default=1 Constant that multiplies the penalty term and thus determines the regularization strength. `alpha = 0` is equivalent to unpenalized GLMs. In this case, the design matrix X must have full column rank (no collinearities). Values must be in the range [0.0, inf).

`fit_intercept` : bool, default=True Specifies if a constant (a.k.a. bias or intercept) should be added to the linear predictor (X @ coef + intercept).

`link` : {'auto', 'identity', 'log'}, default='auto' The link function of the GLM, i.e. mapping from linear predictor `X @ coeff + intercept` to prediction y_pred. Option 'auto' sets the link depending on the chosen power parameter as follows:

- 'identity' for `power <= 0`, e.g. for the Normal distribution
- 'log' for `power > 0`, e.g. for Poisson, Gamma and Inverse Gaussian distributions

`max_iter` : int, default=100 The maximal number of iterations for the solver. Values must be in the range [1, inf).

`tol` : float, default=1e-4 Stopping criterion. For the lbfgs solver, the iteration will stop when `max{|g_j|, j = 1, ..., d} <= tol` where g_j is the j-th component of the gradient (derivative) of the objective function. Values must be in the range (0.0, inf).

`warm_start` : bool, default=False If set to True, reuse the solution of the previous call to fit as initialization for coef_ and intercept_ .

`verbose` : int, default=0 For the lbfgs solver set verbose to any positive number for verbosity. Values must be in the range [0, inf).

Attributes

coef_ : array of shape (n_features,) Estimated coefficients for the linear predictor (`X @ coef_ + intercept_`) in the GLM.

intercept_ : float Intercept (a.k.a. bias) added to linear predictor.

n_iter_ : int Actual number of iterations used in the solver.

n_features_in_ : int Number of features seen during :term:fit.

Added in version: 0.24:

feature_names_in_ : ndarray of shape (n_features_in_,) Names of features seen during :term:fit. Defined only when X has feature names that are all strings.

Added in version: 1.0:

See Also

PoissonRegressor Generalized Linear Model with a Poisson distribution.
GammaRegressor Generalized Linear Model with a Gamma distribution.

Examples

```
>>> from sklearn import linear_model
>>> clf = linear_model.TweedieRegressor()
>>> X = [[1, 2], [2, 3], [3, 4], [4, 3]]
>>> y = [2, 3.5, 5, 5.5]
>>> clf.fit(X, y)
TweedieRegressor()
>>> clf.score(X, y)
0.839...
>>> clf.coef_
array([0.599..., 0.299...])
>>> clf.intercept_
1.600...
>>> clf.predict([[1, 1], [3, 4]])
array([2.500..., 4.599...])
```

Module `src.concrete.ml.sklearn.linear_model`

Implement sklearn linear model.

Classes

Class `LinearRegression`

```
class LinearRegression(
    n_bits=2,
    fit_intercept=True,
    normalize='deprecated',
    copy_X=True,
    n_jobs=None,
    positive=False
)
```

A linear regression model with FHE.

Initialize the FHE linear model.

Args —–= **n_bits** : int, Dict : Number of bits to quantize the model. If an int is passed for n_bits, the value will be used for activation, inputs and weights. If a dict is passed, then it should contain "net_inputs", "op_inputs", "op_weights" and "net_outputs" keys with corresponding number of quantization bits for: - net_inputs : number of bits for model input - op_inputs : number of bits to quantize layer input values - op_weights: learned parameters or constants in the network - net_outputs: final model output quantization bits Default to 2.

`*args` The arguments to pass to the sklearn linear model.
`**kwargs` The keyword arguments to pass to the sklearn linear model.

Ancestors (in MRO)

- src.concrete.ml.sklearn.base.SklearnLinearModelMixin
- sklearn.base.BaseEstimator
- sklearn.base.RegressorMixin

Class variables

Variable `random_state` Type: `Union[numpy.random.mtrand.RandomState, int, None]`

Variable `sklearn_alg` Type: `Callable`

Ordinary least squares Linear Regression.

LinearRegression fits a linear model with coefficients w = (w1, …, wp) to minimize the residual sum of squares between the observed targets in the dataset, and the targets predicted by the linear approximation.

Parameters

`fit_intercept` : bool, default=True Whether to calculate the intercept for this model. If set to False, no intercept will be used in calculations (i.e. data is expected to be centered).

`normalize` : bool, default=False This parameter is ignored when fit_intercept is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use :class:`~sklearn.preprocessing.StandardScaler` before calling fit on an estimator with `normalize=False`.

Deprecated since version: 1.0: normalize was deprecated in version 1.0 and will be removed in 1.2.

`copy_X` : bool, default=True If True, X will be copied; else, it may be overwritten.

`n_jobs` : int, default=None The number of jobs to use for the computation. This will only provide speedup in case of sufficiently large problems, that is if firstly `n_targets > 1` and secondly X is sparse or if positive is set to True. None means 1 unless in a :obj:joblib.parallel_backend context. `-1` means using all processors. See :term:`Glossary <n_jobs>` for more details.

`positive` : bool, default=False When set to True, forces the coefficients to be positive. This option is only supported for dense arrays.

Added in version: 0.24:

Attributes

`coef_` : array of shape (n_features,) or (n_targets, n_features) Estimated coefficients for the linear regression problem. If multiple targets are passed during the fit (y 2D), this is a 2D array of shape (n_targets, n_features), while if only one target is passed, this is a 1D array of length n_features.

`rank_` : int Rank of matrix X. Only available when X is dense.

`singular_` : array of shape (min(X, y),) Singular values of X. Only available when X is dense.

`intercept_` : float or array of shape (n_targets,) Independent term in the linear model. Set to 0.0 if `fit_intercept = False`.

`n_features_in_` : int Number of features seen during :term:fit.

Added in version: 0.24:

`feature_names_in_` : ndarray of shape (n_features_in_,) Names of features seen during :term:fit. Defined only when X has feature names that are all strings.

Added in version: 1.0:

See Also

Ridge Ridge regression addresses some of the problems of Ordinary Least Squares by imposing a penalty on the size of the coefficients with l2 regularization.
Lasso The Lasso is a linear model that estimates sparse coefficients with l1 regularization.
ElasticNet Elastic-Net is a linear regression model trained with both l1 and l2 -norm regularization of the coefficients.

Notes

From the implementation point of view, this is just plain Ordinary Least Squares (scipy.linalg.lstsq) or Non Negative Least Squares (scipy.optimize.nnls) wrapped as a predictor object.

Examples

```
>>> import numpy as np
>>> from sklearn.linear_model import LinearRegression
>>> X = np.array([[1, 1], [1, 2], [2, 2], [2, 3]])
>>> # y = 1 * x_0 + 2 * x_1 + 3
>>> y = np.dot(X, np.array([1, 2])) + 3
>>> reg = LinearRegression().fit(X, y)
>>> reg.score(X, y)
1.0
>>> reg.coef_
array([1., 2.])
>>> reg.intercept_
3.0...
>>> reg.predict(np.array([[3, 5]]))
array([16.])
```

Class `LogisticRegression`

```
class LogisticRegression(
    n_bits=2,
    penalty='l2',
    dual=False,
    tol=0.0001,
    C=1.0,
    fit_intercept=True,
    intercept_scaling=1,
    class_weight=None,
    random_state=None,
    solver='lbfgs',
    max_iter=100,
    multi_class='auto',
    verbose=0,
    warm_start=False,
    n_jobs=None,
    l1_ratio=None
)
```

A logistic regression model with FHE.

Initialize the FHE linear model.

Args —–= **`n_bits`** : int, Dict : Number of bits to quantize the model. If an int is passed for n_bits, the value will be used for activation, inputs and weights. If a dict is passed, then it should contain "net_inputs", "op_inputs", "op_weights" and "net_outputs" keys with corresponding number of quantization bits for: - net_inputs : number of bits for model input - op_inputs : number of bits to quantize layer input values - op_weights: learned parameters or constants in the network - net_outputs: final model output quantization bits Default to 2.

`*args` The arguments to pass to the sklearn linear model.
`**kwargs` The keyword arguments to pass to the sklearn linear model.

Ancestors (in MRO)

- src.concrete.ml.sklearn.base.SklearnLinearModelMixin
- sklearn.base.BaseEstimator
- sklearn.base.ClassifierMixin

Class variables

Variable `random_state` Type: `Union[numpy.random.mtrand.RandomState, int, None]`

Variable `sklearn_alg` Type: `Callable`

Logistic Regression (aka logit, MaxEnt) classifier.

In the multiclass case, the training algorithm uses the one-vs-rest (OvR) scheme if the 'multi_class' option is set to 'ovr', and uses the cross-entropy loss if the 'multi_class' option is set to 'multinomial'. (Currently the 'multinomial' option is supported only by the 'lbfgs', 'sag', 'saga' and 'newton-cg' solvers.)

This class implements regularized logistic regression using the 'liblinear' library, 'newton-cg', 'sag', 'saga' and 'lbfgs' solvers. **Note that regularization is applied by default**. It can handle both dense and sparse input. Use C-ordered arrays or CSR matrices containing 64-bit floats for optimal performance; any other input format will be converted (and copied).

The 'newton-cg', 'sag', and 'lbfgs' solvers support only L2 regularization with primal formulation, or no regularization. The 'liblinear' solver supports both L1 and L2 regularization, with a dual formulation only for the L2 penalty. The Elastic-Net regularization is only supported by the 'saga' solver.

Read more in the :ref:User Guide <logistic_regression>.

Parameters

`penalty` : `{'l1', 'l2', 'elasticnet', 'none'}`, default=`'l2'` Specify the norm of the penalty:

- `'none'`: no penalty is added;
- `'l2'`: add a L2 penalty term and it is the default choice;
- `'l1'`: add a L1 penalty term;
- `'elasticnet'`: both L1 and L2 penalty terms are added.

Warning: Some penalties may not work with some solvers. See the parameter solver below, to know the compatibility between the penalty and solver.

Added in version: 0.19: l1 penalty with SAGA solver (allowing 'multinomial' + L1)

`dual` : bool, default=False Dual or primal formulation. Dual formulation is only implemented for l2 penalty with liblinear solver. Prefer dual=False when n_samples > n_features.

`tol` : float, default=`1e-4` Tolerance for stopping criteria.

`C` : float, default=1.0 Inverse of regularization strength; must be a positive float. Like in support vector machines, smaller values specify stronger regularization.

`fit_intercept` : bool, default=True Specifies if a constant (a.k.a. bias or intercept) should be added to the decision function.

`intercept_scaling` : float, default=1 Useful only when the solver 'liblinear' is used and self.fit_intercept is set to True. In this case, x becomes [x, self.intercept_scaling], i.e. a "synthetic" feature with constant value equal to intercept_scaling is appended to the instance vector. The intercept becomes `intercept_scaling * synthetic_feature_weight`.

Note! the synthetic feature weight is subject to l1/l2 regularization as all other features. To lessen the effect of regularization on synthetic feature weight (and therefore on the intercept) intercept_scaling has to be increased.

`class_weight` : dict or `'balanced'`, default=None Weights associated with classes in the form `{class_label: weight}`. If not given, all classes are supposed to have weight one.

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`.

Note that these weights will be multiplied with sample_weight (passed through the fit method) if sample_weight is specified.

Added in version: 0.17: *class_weight='balanced'*

`random_state` : int, RandomState instance, default=None Used when solver == 'sag', 'saga' or 'liblinear' to shuffle the data. See :term:`Glossary <random_state>` for details.

`solver` : `{'newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'}`, default=`'lbfgs'` Algorithm to use in the optimization problem. Default is 'lbfgs'. To choose a solver, you might want to consider the following aspects:

```
- For small datasets, 'liblinear' is a good choice, whereas 'sag'
  and 'saga' are faster for large ones;
- For multiclass problems, only 'newton-cg', 'sag', 'saga' and
  'lbfgs' handle multinomial loss;
- 'liblinear' is limited to one-versus-rest schemes.
```

Warning: The choice of the algorithm depends on the penalty chosen: Supported penalties by solver:

- 'newton-cg' - ['l2', 'none']
- 'lbfgs' - ['l2', 'none']
- 'liblinear' - ['l1', 'l2']
- 'sag' - ['l2', 'none']
- 'saga' - ['elasticnet', 'l1', 'l2', 'none']

Note: 'sag' and 'saga' fast convergence is only guaranteed on features with approximately the same scale. You can preprocess the data with a scaler from :mod:sklearn.preprocessing.

Seealso: Refer to the User Guide for more information regarding :class:LogisticRegression and more specifically the :ref:`Table <Logistic_regression>` summarizing solver/penalty supports.

Added in version: 0.17: Stochastic Average Gradient descent solver.

Added in version: 0.19: SAGA solver.

Changed in version: 0.22: The default solver changed from 'liblinear' to 'lbfgs' in 0.22.

`max_iter` : int, default=100 Maximum number of iterations taken for the solvers to converge.

`multi_class` : `{'auto', 'ovr', 'multinomial'}`, default=`'auto'` If the option chosen is 'ovr', then a binary problem is fit for each label. For 'multinomial' the loss minimised is the multinomial loss fit across the entire probability distribution, *even when the data is binary*. 'multinomial' is unavailable when solver='liblinear'. 'auto' selects 'ovr' if the data is binary, or if solver='liblinear', and otherwise selects 'multinomial'.

Added in version: 0.18: Stochastic Average Gradient descent solver for 'multinomial' case.

Changed in version: 0.22: Default changed from 'ovr' to 'auto' in 0.22.

`verbose` : int, default=0 For the liblinear and lbfgs solvers set verbose to any positive number for verbosity.

`warm_start` : bool, default=False When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. Useless for liblinear solver. See :term:`the Glossary <warm_start>`.

Added in version: 0.17: *warm_start* to support *lbfgs*, *newton-cg*, *sag*, *saga* solvers.

n_jobs : int, default=None Number of CPU cores used when parallelizing over classes if multi_class='ovr'". This parameter is ignored when the solver is set to 'liblinear' regardless of whether 'multi_class' is specified or not. None means 1 unless in a :obj:joblib.parallel_backend context. -1 means using all processors. See :term:`Glossary <n_jobs>` for more details.

l1_ratio : float, default=None The Elastic-Net mixing parameter, with `0 <= l1_ratio <= 1`. Only used if `penalty='elasticnet'`. Setting `l1_ratio=0` is equivalent to using `penalty='l2'`, while setting `l1_ratio=1` is equivalent to using `penalty='l1'`. For `0 < l1_ratio <1`, the penalty is a combination of L1 and L2.

Attributes

classes_ : ndarray of shape (n_classes,) A list of class labels known to the classifier.

coef_ : ndarray of shape (1, n_features) or (n_classes, n_features) Coefficient of the features in the decision function.

coef_ is of shape (1, n_features) when the given problem is binary. In particular, when `multi_class='multinomial'`, coef_ corresponds to outcome 1 (True) and `-coef_` corresponds to outcome 0 (False).

intercept_ : ndarray of shape (1,) or (n_classes,) Intercept (a.k.a. bias) added to the decision function.

If fit_intercept is set to False, the intercept is set to zero. intercept_ is of shape (1,) when the given problem is binary. In particular, when `multi_class='multinomial'`, intercept_ corresponds to outcome 1 (True) and `-intercept_` corresponds to outcome 0 (False).

n_features_in_ : int Number of features seen during :term:fit.

Added in version: 0.24:

feature_names_in_ : ndarray of shape (n_features_in_,) Names of features seen during :term:fit. Defined only when X has feature names that are all strings.

Added in version: 1.0:

n_iter_ : ndarray of shape (n_classes,) or (1,) Actual number of iterations for all classes. If binary or multinomial, it returns only 1 element. For liblinear solver, only the maximum number of iteration across all classes is given.

Changed in version: 0.20: In SciPy <= 1.0.0 the number of lbfgs iterations may exceed max_iter. n_iter_ will now report at most max_iter.

See Also

SGDClassifier Incrementally trained logistic regression (when given the parameter `loss="log"`).
LogisticRegressionCV Logistic regression with built-in cross validation.

Notes

The underlying C implementation uses a random number generator to select features when fitting the model. It is thus not uncommon, to have slightly different results for the same input data. If that happens, try with a smaller tol parameter.

Predict output may not match that of standalone liblinear in certain cases. See :ref:`differences from liblinear <liblinear_differences>` in the narrative documentation.

References

L-BFGS-B – Software for Large-scale Bound-constrained Optimization Ciyou Zhu, Richard Byrd, Jorge Nocedal and Jose Luis Morales. http://users.iems.northwestern.edu/~nocedal/lbfgsb.html

LIBLINEAR – A Library for Large Linear Classification https://www.csie.ntu.edu.tw/~cjlin/liblinear/

SAG – Mark Schmidt, Nicolas Le Roux, and Francis Bach Minimizing Finite Sums with the Stochastic Average Gradient https://hal.inria.fr/hal-00860051/document

SAGA – Defazio, A., Bach F. & Lacoste-Julien S. (2014). :arxiv:`"SAGA: A Fast Incremental Gradient Method With Support for Non-Strongly Convex Composite Objectives" <1407.0202>`

Hsiang-Fu Yu, Fang-Lan Huang, Chih-Jen Lin (2011). Dual coordinate descent methods for logistic regression and maximum entropy models. Machine Learning 85(1-2):41-75. https://www.csie.ntu.edu.tw/~cjlin/papers/maxent_dual.pdf

Examples

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.linear_model import LogisticRegression
>>> X, y = load_iris(return_X_y=True)
>>> clf = LogisticRegression(random_state=0).fit(X, y)
>>> clf.predict(X[:2, :])
array([0, 0])
>>> clf.predict_proba(X[:2, :])
array([[9.8...e-01, 1.8...e-02, 1.4...e-08],
       [9.7...e-01, 2.8...e-02, ...e-08]])
>>> clf.score(X, y)
0.97...
```

Methods

Method `clean_graph`

```
def clean_graph(
    self,
    onnx_model: onnx.onnx_ml_pb2.ModelProto
)
```

Clean the graph of the onnx model.

Args ——= **`onnx_model`** : onnx.ModelProto : the onnx model

Returns ——= onnx.ModelProto : the cleaned onnx model

Method `decision_function`

```
def decision_function(
    self,
    X: numpy.ndarray,
    execute_in_fhe: bool = False
) -> numpy.ndarray
```

Predict confidence scores for samples.

Args ——= **`X`** : samples to predict

`execute_in_fhe` if True, the model will be executed in FHE mode

Returns ——= numpy.ndarray : confidence scores for samples

Method `predict_proba`

```
def predict_proba(
    self,
    X: numpy.ndarray,
    execute_in_fhe: bool = False
) -> numpy.ndarray
```

Predict class probabilities for samples.

Args ——= **`X`** : samples to predict

`execute_in_fhe` if True, the model will be executed in FHE mode

Returns ——= numpy.ndarray : class probabilities for samples

Module `src.concrete.ml.sklearn.qnn`

Scikit-learn interface for concrete quantized neural networks.

Classes

Class `FixedTypeSkorchNeuralNet`

```
class FixedTypeSkorchNeuralNet
```

A mixin with a helpful modification to a skorch estimator that fixes the module type.

Descendants

- src.concrete.ml.sklearn.qnn.NeuralNetClassifier
- src.concrete.ml.sklearn.qnn.NeuralNetRegressor

Methods

Method `get_params`

```
def get_params(
    self,
    deep=True,
    **kwargs
)
```

Get parameters for this estimator.

Args —–= **`deep`** : bool : If True, will return the parameters for this estimator and contained subobjects that are estimators.

`**kwargs` any additional parameters to pass to the sklearn BaseEstimator class

Returns —–= params : dict, Parameter names mapped to their values.

Class `NeuralNetClassifier`

```
class NeuralNetClassifier(
    *args,
    criterion=torch.nn.modules.loss.CrossEntropyLoss,
    classes=None,
    optimizer=torch.optim.adam.Adam,
    **kwargs
)
```

Scikit-learn interface for quantized FHE compatible neural networks.

This class wraps a quantized NN implemented using our Torch tools as a scikit-learn Estimator. It uses the skorch package to handle training and scikit-learn compatibility, and adds quantization and compilation functionality.

The datatypes that are allowed for prediction by this wrapper are more restricted than standard scikit-learn estimators as this class needs to predict in FHE and network inference executor is the NumpyModule.

Ancestors (in MRO)

- src.concrete.ml.sklearn.qnn.FixedTypeSkorchNeuralNet
- src.concrete.ml.sklearn.qnn.QuantizedSkorchEstimatorMixin
- src.concrete.ml.sklearn.base.QuantizedTorchEstimatorMixin
- skorch.classifier.NeuralNetClassifier
- skorch.net.NeuralNet
- sklearn.base.ClassifierMixin

Class variables

Variable `post_processing_params` Type: `Dict[str, Any]`

Class `NeuralNetRegressor`

```
class NeuralNetRegressor(
    *args,
    optimizer=torch.optim.adam.Adam,
    **kwargs
)
```

Scikit-learn interface for quantized FHE compatible neural networks.

This class wraps a quantized NN implemented using our Torch tools as a scikit-learn Estimator. It uses the skorch package to handle training and scikit-learn compatibility, and adds quantization and compilation functionality.

The datatypes that are allowed for prediction by this wrapper are more restricted than standard scikit-learn estimators as this class needs to predict in FHE and network inference executor is the NumpyModule.

Ancestors (in MRO)

- src.concrete.ml.sklearn.qnn.FixedTypeSkorchNeuralNet
- src.concrete.ml.sklearn.qnn.QuantizedSkorchEstimatorMixin
- src.concrete.ml.sklearn.base.QuantizedTorchEstimatorMixin
- skorch.regressor.NeuralNetRegressor
- skorch.net.NeuralNet
- sklearn.base.RegressorMixin

Class variables

Variable `post_processing_params` Type: `Dict[str, Any]`

Class `QuantizedSkorchEstimatorMixin`

```
class QuantizedSkorchEstimatorMixin(
    *args,
    **kwargs
)
```

Mixin class that adds quantization features to Skorch NN estimators.

Ancestors (in MRO)

- src.concrete.ml.sklearn.base.QuantizedTorchEstimatorMixin

Descendants

- src.concrete.ml.sklearn.qnn.NeuralNetClassifier
- src.concrete.ml.sklearn.qnn.NeuralNetRegressor

Class variables

Variable `post_processing_params` Type: `Dict[str, Any]`

Instance variables

Variable `base_module_to_compile` Get the module that should be compiled to FHE. In our case this is a torch nn.Module.

Returns —= module (nn.Module): the instantiated torch module

Variable `n_bits_quant` Return the number of quantization bits.

This is stored by the torch.nn.module instance and thus cannot be retrieved until this instance is created.

Returns —= n_bits (int): the number of bits to quantize the network

Raises —= ValueError : with skorch estimators, the module_ is not instantiated until .fit() is called. Thus this estimator needs to be .fit() before we get the quantization number of bits. If it is not trained we raise an exception

Methods

Method `get_params_for_benchmark`

```
def get_params_for_benchmark(
    self
)
```

Get parameters for benchmark when cloning a skorch wrapped NN.

We must remove all parameters related to the module. Skorch takes either a class or a class instance for the module parameter. We want to pass our trained model, a class instance. But for this to work, we need to remove all module related constructor params. If not, skorch will instantiate a new class instance of the same type as the passed module see skorch net.py NeuralNet::initialize_instance

Returns —= params (dict): parameters to create an equivalent fp32 sklearn estimator for benchmark

Method `infer`

```
def infer(
    self,
    x,
    **fit_params
)
```

Perform a single inference step on a batch of data.

This method is specific to Skorch estimators.

Args —= **`x`** : torch.Tensor : A batch of the input data, produced by a Dataset

**fit_params (dict) : Additional parameters passed to the forward method of the module and to the self.train_split call.

Returns —= A torch tensor with the inference results for each item in the input

Method `on_train_end`

```
def on_train_end(
    self,
    net,
    X=None,
    y=None,
    **kwargs
)
```

Call back when training is finished by the skorch wrapper.

Check if the underlying neural net has a callback for this event and, if so, call it.

Args —= **`net`** : estimator for which training has ended (equal to self)

X data
y targets
kwargs other arguments

Class SparseQuantNeuralNetImpl

```
class SparseQuantNeuralNetImpl(
    input_dim,
    n_layers,
    n_outputs,
    n_hidden_neurons_multiplier=4,
    n_w_bits=3,
    n_a_bits=3,
    n_accum_bits=8,
    activation_function=torch.nn.modules.activation.ReLU
)
```

Sparse Quantized Neural Network classifier.

This class implements an MLP that is compatible with FHE constraints. The weights and activations are quantized to low bitwidth and pruning is used to ensure accumulators do not surpass an user-provided accumulator bit-width. The number of classes and number of layers are specified by the user, as well as the breadth of the network

Sparse Quantized Neural Network constructor.

Args ——= **input_dim** : Number of dimensions of the input data

n_layers Number of linear layers for this network
n_outputs Number of output classes or regression targets
n_w_bits Number of weight bits
n_a_bits Number of activation and input bits
n_accum_bits Maximal allowed bitwidth of intermediate accumulators
n_hidden_neurons_multiplier A factor that is multiplied by the maximal number of active (non-zero weight) neurons for every layer. The maximal number of neurons in the worst case scenario is: 2^n_max-1 max_active_neurons(n_max, n_w, n_a) = floor(————————) (2^n_w-1)*(2^n_a-1)) The worst case scenario for the bitwidth of the accumulator is when all weights and activations are maximum simultaneously. We set, for each layer, the total number of neurons to be: n_hidden_neurons_multiplier* max_active_neurons(n_accum_bits, n_w_bits, n_a_bits) Through experiments, for typical distributions of weights and activations, the default value for n_hidden_neurons_multiplier, 4, is safe to avoid overflow.
activation_function a torch class that is used to construct activation functions in the network (e.g. torch.ReLU, torch.SELU, torch.Sigmoid, etc)

Raises ——= ValueError : if the parameters have invalid values or the computed accumulator bitwidth is zero

Ancestors (in MRO)

- torch.nn.modules.module.Module

Class variables

Variable dump_patches Type: bool

Variable training Type: bool

Methods

Method enable_pruning

```
def enable_pruning(
    self
)
```

Enable pruning in the network. Pruning must be made permanent to recover pruned weights.

Raises ——= ValueError : if the quantization parameters are invalid

Method forward

```
def forward(
    self,
    x
) -> Callable[..., Any]
```

Forward pass.

Args ——= **x** : torch.Tensor : network input

Returns ——= x (torch.Tensor): network prediction

Method make_pruning_permanent

```
def make_pruning_permanent(
    self
)
```

Make the learned pruning permanent in the network.

Method max_active_neurons

```
def max_active_neurons(
    self
)
```

Compute the maximum number of active (non-zero weight) neurons.

The computation is done using the quantization parameters passed to the constructor. Warning: With the current quantization algorithm (asymmetric) the value returned by this function is not guaranteed to ensure FHE compatibility. For some weight distributions, weights that are 0 (which are pruned weights) will not be quantized to 0. Therefore the total number of active quantized neurons will not be equal to max_active_neurons.

Returns ——= n (int): maximum number of active neurons

Method on_train_end

```
def on_train_end(
    self
)
```

Call back when training is finished, can be useful to remove training hooks.

Module src.concrete.ml.sklearn.rf

Implements RandomForest models.

Classes

Class RandomForestClassifier

```
class RandomForestClassifier(
    n_bits: int = 6,
```

```
        n_estimators=100,
        criterion='gini',
        max_depth=None,
        min_samples_split=2,
        min_samples_leaf=1,
        min_weight_fraction_leaf=0.0,
        max_features='sqrt',
        max_leaf_nodes=None,
        min_impurity_decrease=0.0,
        bootstrap=True,
        oob_score=False,
        n_jobs=None,
        random_state=None,
        verbose=0,
        warm_start=False,
        class_weight=None,
        ccp_alpha=0.0,
        max_samples=None
    )
```

Implements the RandomForest classifier.

Initialize the RandomForestClassifier.

noqa: DAR101

Ancestors (in MRO)

- src.concrete.ml.sklearn.base.BaseTreeEstimatorMixin
- sklearn.base.BaseEstimator
- sklearn.base.ClassifierMixin

Class variables

Variable `framework` Type: `str`

Variable `n_bits` Type: `int`

Variable `q_x_byfeatures` Type: `List[src.concrete.ml.quantization.quantized_array.QuantizedArray]`

Variable `q_y` Type: `src.concrete.ml.quantization.quantized_array.QuantizedArray`

Variable `sklearn_alg` Type: `Any`

A random forest classifier.

A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is controlled with the max_samples parameter if `bootstrap=True` (default), otherwise the whole dataset is used to build each tree.

Read more in the :ref:`User Guide <forest>`.

Parameters

`n_estimators` : int, default=100 The number of trees in the forest.

Changed in version: 0.22: The default value of n_estimators changed from 10 to 100 in 0.22.

`criterion` : {"gini", "entropy", "log_loss"}, default="gini" The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "log_loss" and "entropy" both for the Shannon information gain, see :ref:tree_mathematical_formulation. Note: This parameter is tree-specific.

`max_depth` : int, default=None The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

`min_samples_split` : int or float, default=2 The minimum number of samples required to split an internal node:

- If int, then consider min_samples_split as the minimum number.
- If float, then min_samples_split is a fraction and `ceil(min_samples_split * n_samples)` are the minimum number of samples for each split.

Changed in version: 0.18: Added float values for fractions.

`min_samples_leaf` : int or float, default=1 The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least min_samples_leaf training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

- If int, then consider min_samples_leaf as the minimum number.
- If float, then min_samples_leaf is a fraction and `ceil(min_samples_leaf * n_samples)` are the minimum number of samples for each node.

Changed in version: 0.18: Added float values for fractions.

`min_weight_fraction_leaf` : float, default=0.0 The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

`max_features` : `{"sqrt", "log2", None}`, int or float, default=`"sqrt"` The number of features to consider when looking for the best split:

- If int, then consider max_features features at each split.
- If float, then max_features is a fraction and `max(1, int(max_features * n_features_in_))` features are considered at each split.
- If "auto", then `max_features=sqrt(n_features)`.
- If "sqrt", then `max_features=sqrt(n_features)`.
- If "log2", then `max_features=log2(n_features)`.
- If None, then `max_features=n_features`.

Changed in version: 1.1: The default of max_features changed from `"auto"` to `"sqrt"`.

Deprecated since version: 1.1: The `"auto"` option was deprecated in 1.1 and will be removed in 1.3.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features.

`max_leaf_nodes` : int, default=None Grow trees with max_leaf_nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

`min_impurity_decrease` : float, default=0.0 A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following::

```
N_t / N * (impurity - N_t_R / N_t * right_impurity
                    - N_t_L / N_t * left_impurity)
```

where N is the total number of samples, N_t is the number of samples at the current node, N_t_L is the number of samples in the left child, and N_t_R is the number of samples in the right child.

N, N_t, N_t_R and N_t_L all refer to the weighted sum, if sample_weight is passed.

Added in version: 0.19:

`bootstrap` : bool, default=True Whether bootstrap samples are used when building trees. If False, the whole dataset is used to build each tree.

oob_score : bool, default=False Whether to use out-of-bag samples to estimate the generalization score. Only available if bootstrap=True.

n_jobs : int, default=None The number of jobs to run in parallel. :meth:fit, :meth:predict, :meth:decision_path and :meth:apply are all parallelized over the trees. None means 1 unless in a :obj:joblib.parallel_backend context. `-1` means using all processors. See :term:`Glossary <n_jobs>` for more details.

random_state : int, RandomState instance or None, default=None Controls both the randomness of the bootstrapping of the samples used when building trees (if `bootstrap=True`) and the sampling of the features to consider when looking for the best split at each node (if `max_features < n_features`). See :term:`Glossary <random_state>` for details.

verbose : int, default=0 Controls the verbosity when fitting and predicting.

warm_start : bool, default=False When set to True, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new forest. See :term:`the Glossary <warm_start>`.

class_weight : {"balanced", "balanced_subsample"}, dict or list of dicts, default=None Weights associated with classes in the form `{class_label: weight}`. If not given, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of y.

Note that for multioutput (including multilabel) weights should be defined for each class of every column in its own dict. For example, for four-class multilabel classification weights should be [{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}] instead of [{1:1}, {2:5}, {3:1}, {4:1}].

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`

The "balanced_subsample" mode is the same as "balanced" except that weights are computed based on the bootstrap sample for every tree grown.

For multi-output, the weights of each column of y will be multiplied.

Note that these weights will be multiplied with sample_weight (passed through the fit method) if sample_weight is specified.

ccp_alpha : non-negative float, default=0.0 Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than ccp_alpha will be chosen. By default, no pruning is performed. See :ref:minimal_cost_complexity_pruning for details.

Added in version: 0.22:

max_samples : int or float, default=None If bootstrap is True, the number of samples to draw from X to train each base estimator.

- If None (default), then draw X.shape[0] samples.
- If int, then draw max_samples samples.
- If float, then draw `max_samples * X.shape[0]` samples. Thus, max_samples should be in the interval (0.0, 1.0].

Added in version: 0.22:

Attributes

base_estimator_ : DecisionTreeClassifier The child estimator template used to create the collection of fitted sub-estimators.

estimators_ : list of DecisionTreeClassifier The collection of fitted sub-estimators.

classes_ : ndarray of shape (n_classes,) or a list of such arrays The classes labels (single output problem), or a list of arrays of class labels (multi-output problem).

n_classes_ : int or list The number of classes (single output problem), or a list containing the number of classes for each output (multi-output problem).

n_features_ : int The number of features when fit is performed.

Deprecated since version: 1.0: Attribute n_features_ was deprecated in version 1.0 and will be removed in 1.2. Use n_features_in_ instead.

n_features_in_ : int Number of features seen during :term:fit.

Added in version: 0.24:

feature_names_in_ : ndarray of shape (n_features_in_,) Names of features seen during :term:fit. Defined only when X has feature names that are all strings.

Added in version: 1.0:

n_outputs_ : int The number of outputs when fit is performed.

feature_importances_ : ndarray of shape (n_features,) The impurity-based feature importances. The higher, the more important the feature. The importance of a feature is computed as the (normalized) total reduction of the criterion brought by that feature. It is also known as the Gini importance.

Warning: impurity-based feature importances can be misleading for high cardinality features (many unique values). See :func:sklearn.inspection.permutation_importance as an alternative.

oob_score_ : float Score of the training dataset obtained using an out-of-bag estimate. This attribute exists only when oob_score is True.

oob_decision_function_ : ndarray of shape (n_samples, n_classes) or (n_samples, n_classes, n_outputs Decision function computed with out-of-bag estimate on the training set. If n_estimators is small it might be possible that a data point was never left out during the bootstrap. In this case, oob_decision_function_ might contain NaN. This attribute exists only when oob_score is True.

See Also

sklearn.tree.DecisionTreeClassifier A decision tree classifier.
sklearn.ensemble.ExtraTreesClassifier Ensemble of extremely randomized tree classifiers.

Notes

The default values for the parameters controlling the size of the trees (e.g. max_depth, min_samples_leaf, etc.) lead to fully grown and unpruned trees which can potentially be very large on some data sets. To reduce memory consumption, the complexity and size of the trees should be controlled by setting those parameter values.

The features are always randomly permuted at each split. Therefore, the best found split may vary, even with the same training data, `max_features=n_features` and `bootstrap=False`, if the improvement of the criterion is identical for several splits enumerated during the search of the best split. To obtain a deterministic behaviour during fitting, random_state has to be fixed.

References

.. [1] L. Breiman, “Random Forests”, Machine Learning, 45(1), 5-32, 2001.

Examples

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.datasets import make_classification
>>> X, y = make_classification(n_samples=1000, n_features=4,
...                            n_informative=2, n_redundant=0,
...                            random_state=0, shuffle=False)
>>> clf = RandomForestClassifier(max_depth=2, random_state=0)
>>> clf.fit(X, y)
RandomForestClassifier(...)
>>> print(clf.predict([[0, 0, 0, 0]]))
[1]
```

Variable `sklearn_model` Type: `Any`

Module `src.concrete.ml.sklearn.svm`

Implement Support Vector Machine.

Classes

Class `LinearSVC`

```
class LinearSVC(
    n_bits=2,
    penalty='l2',
    loss='squared_hinge',
    *,
    dual=True,
    tol=0.0001,
    C=1.0,
    multi_class='ovr',
    fit_intercept=True,
    intercept_scaling=1,
    class_weight=None,
    verbose=0,
    random_state=None,
    max_iter=1000
)
```

A Classification Support Vector Machine (SVM).

Initialize the FHE linear model.

Args ——= **`n_bits`** : int, Dict : Number of bits to quantize the model. If an int is passed for n_bits, the value will be used for activation, inputs and weights. If a dict is passed, then it should contain "net_inputs", "op_inputs", "op_weights" and "net_outputs" keys with corresponding number of quantization bits for: - net_inputs : number of bits for model input - op_inputs : number of bits to quantize layer input values - op_weights: learned parameters or constants in the network - net_outputs: final model output quantization bits Default to 2.

`*args` The arguments to pass to the sklearn linear model.
`**kwargs` The keyword arguments to pass to the sklearn linear model.

Ancestors (in MRO)

- src.concrete.ml.sklearn.base.SklearnLinearModelMixin
- sklearn.base.BaseEstimator
- sklearn.base.ClassifierMixin

Class variables

Variable `random_state` Type: `Union[numpy.random.mtrand.RandomState, int, None]`

Variable `sklearn_alg` Type: `Callable`

Linear Support Vector Classification.

Similar to SVC with parameter kernel='linear', but implemented in terms of liblinear rather than libsvm, so it has more flexibility in the choice of penalties and loss functions and should scale better to large numbers of samples.

This class supports both dense and sparse input and the multiclass support is handled according to a one-vs-the-rest scheme.

Read more in the :ref:User Guide <svm_classification>.

Parameters

penalty : {'l1', 'l2'}, default='l2' Specifies the norm used in the penalization. The 'l2' penalty is the standard used in SVC. The 'l1' leads to coef_ vectors that are sparse.

loss : {'hinge', 'squared_hinge'}, default='squared_hinge' Specifies the loss function. 'hinge' is the standard SVM loss (used e.g. by the SVC class) while 'squared_hinge' is the square of the hinge loss. The combination of `penalty='l1'` and `loss='hinge'` is not supported.

dual : bool, default=True Select the algorithm to either solve the dual or primal optimization problem. Prefer dual=False when n_samples > n_features.

tol : float, default=1e-4 Tolerance for stopping criteria.

C : float, default=1.0 Regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive.

multi_class : {'ovr', 'crammer_singer'}, default='ovr' Determines the multi-class strategy if y contains more than two classes. `"ovr"` trains n_classes one-vs-rest classifiers, while `"crammer_singer"` optimizes a joint objective over all classes. While crammer_singer is interesting from a theoretical perspective as it is consistent, it is seldom used in practice as it rarely leads to better accuracy and is more expensive to compute. If `"crammer_singer"` is chosen, the options loss, penalty and dual will be ignored.

fit_intercept : bool, default=True Whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (i.e. data is expected to be already centered).

intercept_scaling : float, default=1 When self.fit_intercept is True, instance vector x becomes [x, self.intercept_scaling], i.e. a "synthetic" feature with constant value equals to intercept_scaling is appended to the instance vector. The intercept becomes intercept_scaling * synthetic feature weight Note! the synthetic feature weight is subject to l1/l2 regularization as all other features. To lessen the effect of regularization on synthetic feature weight (and therefore on the intercept) intercept_scaling has to be increased.

class_weight : dict or 'balanced', default=None Set the parameter C of class i to `class_weight[i]*C` for SVC. If not given, all classes are supposed to have weight one. The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`.

verbose : int, default=0 Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in liblinear that, if enabled, may not work properly in a multithreaded context.

random_state : int, RandomState instance or None, default=None Controls the pseudo random number generation for shuffling the data for the dual coordinate descent (if `dual=True`). When `dual=False` the underlying implementation of :class:LinearSVC is not random and random_state has no effect on the results. Pass an int for reproducible output across multiple function calls. See :term:`Glossary <random_state>`.

max_iter : int, default=1000 The maximum number of iterations to be run.

Attributes

coef_ : ndarray of shape (1, n_features) if n_classes == 2 else (n_classes, n_features) Weights assigned to the features (coefficients in the primal problem).

coef_ is a readonly property derived from raw_coef_ that follows the internal memory layout of liblinear.

intercept_ : ndarray of shape (1,) if n_classes == 2 else (n_classes,) Constants in decision function.

classes_ : ndarray of shape (n_classes,) The unique classes labels.

n_features_in_ : int Number of features seen during :term:fit.

Added in version: 0.24:

feature_names_in_ : ndarray of shape (n_features_in_,) Names of features seen during :term:fit. Defined only when X has feature names that are all strings.

Added in version: 1.0:

n_iter_ : int Maximum number of iterations run across all classes.

See Also

SVC Implementation of Support Vector Machine classifier using libsvm: the kernel can be non-linear but its SMO algorithm does not scale to large number of samples as LinearSVC does. Furthermore SVC multi-class mode is implemented using one vs one scheme while LinearSVC uses one vs the rest. It is possible to implement one vs the rest with SVC by using the :class:`~sklearn.multiclass.OneVsRestClassifier` wrapper. Finally SVC can fit dense data without memory copy if the input is C-contiguous. Sparse data will still incur memory copy though.

`sklearn.linear_model.SGDClassifier : SGDClassifier can optimize the same cost function as LinearSVC by adjusting the penalty and loss parameters. In addition it requires less memory`, allows incremental (online) learning, and implements various loss functions and regularization regimes.

Notes

The underlying C implementation uses a random number generator to select features when fitting the model. It is thus not uncommon to have slightly different results for the same input data. If that happens, try with a smaller tol parameter.

The underlying implementation, liblinear, uses a sparse internal representation for the data that will incur a memory copy.

Predict output may not match that of standalone liblinear in certain cases. See :ref:`differences from liblinear <liblinear_differences>` in the narrative documentation.

References

`LIBLINEAR: A Library for Large Linear Classification <https://www.csie.ntu.edu.tw/~cjlin/liblinear/>`

Examples

```
>>> from sklearn.svm import LinearSVC
>>> from sklearn.pipeline import make_pipeline
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.datasets import make_classification
>>> X, y = make_classification(n_features=4, random_state=0)
>>> clf = make_pipeline(StandardScaler(),
...                     LinearSVC(random_state=0, tol=1e-5))
>>> clf.fit(X, y)
Pipeline(steps=[('standardscaler', StandardScaler()),
                ('linearsvc', LinearSVC(random_state=0, tol=1e-05))])
```

```
>>> print(clf.named_steps['linearsvc'].coef_)
[[0.141...   0.526... 0.679... 0.493...]]
```

```
>>> print(clf.named_steps['linearsvc'].intercept_)
[0.1693...]
>>> print(clf.predict([[0, 0, 0, 0]]))
[1]
```

Methods

Method `clean_graph`

```
def clean_graph(
    self,
    onnx_model: onnx.onnx_ml_pb2.ModelProto
)
```

Clean the graph of the onnx model.

Args —–= **`onnx_model`** : onnx.ModelProto : the onnx model

Returns —–= onnx.ModelProto : the cleaned onnx model

Method `decision_function`

```
def decision_function(
    self,
    X: numpy.ndarray,
    execute_in_fhe: bool = False
) -> numpy.ndarray
```

Predict confidence scores for samples.

Args —–= **X** : samples to predict

execute_in_fhe if True, the model will be executed in FHE mode

Returns —–= numpy.ndarray : confidence scores for samples

Method `predict_proba`

```
def predict_proba(
    self,
    X: numpy.ndarray,
    execute_in_fhe: bool = False
) -> numpy.ndarray
```

Predict class probabilities for samples.

Args —–= **X** : samples to predict

execute_in_fhe if True, the model will be executed in FHE mode

Returns —–= numpy.ndarray : class probabilities for samples

Class `LinearSVR`

```
class LinearSVR(
    n_bits=2,
    epsilon=0.0,
    tol=0.0001,
    C=1.0,
    loss='epsilon_insensitive',
    fit_intercept=True,
    intercept_scaling=1.0,
    dual=True,
    verbose=0,
    random_state=None,
    max_iter=1000
)
```

A Regression Support Vector Machine (SVM).

Initialize the FHE linear model.

Args —–= **n_bits** : int, Dict : Number of bits to quantize the model. If an int is passed for n_bits, the value will be used for activation, inputs and weights. If a dict is passed, then it should contain "net_inputs", "op_inputs", "op_weights" and "net_outputs" keys with corresponding number of quantization bits for: - net_inputs : number of bits for model input - op_inputs : number of bits to quantize layer input values - op_weights: learned parameters or constants in the network - net_outputs: final model output quantization bits Default to 2.

***args** The arguments to pass to the sklearn linear model.
****kwargs** The keyword arguments to pass to the sklearn linear model.

Ancestors (in MRO)

- src.concrete.ml.sklearn.base.SklearnLinearModelMixin
- sklearn.base.BaseEstimator

- sklearn.base.RegressorMixin

Class variables

Variable `random_state` Type: `Union[numpy.random.mtrand.RandomState, int, None]`

Variable `sklearn_alg` Type: `Callable`

Linear Support Vector Regression.

Similar to SVR with parameter kernel='linear', but implemented in terms of liblinear rather than libsvm, so it has more flexibility in the choice of penalties and loss functions and should scale better to large numbers of samples.

This class supports both dense and sparse input.

Read more in the :ref:`User Guide <svm_regression>`.

Added in version: 0.16:

Parameters

`epsilon` : float, default=0.0 Epsilon parameter in the epsilon-insensitive loss function. Note that the value of this parameter depends on the scale of the target variable y. If unsure, set `epsilon=0`.

`tol` : float, default=1e-4 Tolerance for stopping criteria.

`C` : float, default=1.0 Regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive.

`loss` : {'epsilon_insensitive', 'squared_epsilon_insensitive'}, default='epsilon_insensitive' Specifies the loss function. The epsilon-insensitive loss (standard SVR) is the L1 loss, while the squared epsilon-insensitive loss ('squared_epsilon_insensitive') is the L2 loss.

`fit_intercept` : bool, default=True Whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (i.e. data is expected to be already centered).

`intercept_scaling` : float, default=1.0 When self.fit_intercept is True, instance vector x becomes [x, self.intercept_scaling], i.e. a "synthetic" feature with constant value equals to intercept_scaling is appended to the instance vector. The intercept becomes intercept_scaling * synthetic feature weight Note! the synthetic feature weight is subject to l1/l2 regularization as all other features. To lessen the effect of regularization on synthetic feature weight (and therefore on the intercept) intercept_scaling has to be increased.

`dual` : bool, default=True Select the algorithm to either solve the dual or primal optimization problem. Prefer dual=False when n_samples > n_features.

`verbose` : int, default=0 Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in liblinear that, if enabled, may not work properly in a multithreaded context.

`random_state` : int, RandomState instance or None, default=None Controls the pseudo random number generation for shuffling the data. Pass an int for reproducible output across multiple function calls. See :term:`Glossary <random_state>`.

`max_iter` : int, default=1000 The maximum number of iterations to be run.

Attributes

`coef_` : ndarray of shape `(n_features) if n_classes == 2 else (n_classes, n_features)` Weights assigned to the features (coefficients in the primal problem).

coef_ is a readonly property derived from raw_coef_ that follows the internal memory layout of liblinear.

`intercept_` : ndarray of shape `(1) if n_classes == 2 else (n_classes)` Constants in decision function.

`n_features_in_` : int Number of features seen during :term:fit.

Added in version: 0.24:

feature_names_in_ : ndarray of shape (n_features_in_,) Names of features seen during :term:fit. Defined only when X has feature names that are all strings.

Added in version: 1.0:

n_iter_ : int Maximum number of iterations run across all classes.

See Also

LinearSVC Implementation of Support Vector Machine classifier using the same library as this class (liblinear).

`SVR : Implementation of Support Vector Machine regression using libsvm: the kernel can be non-linear but its SMO algorithm does not scale to large number of samples as LinearSVC does.`

`sklearn.linear_model.SGDRegressor : SGDRegressor can optimize the same cost function as LinearSVR by adjusting the penalty and loss parameters. In addition it requires less memory,` allows incremental (online) learning, and implements various loss functions and regularization regimes.

Examples

```
>>> from sklearn.svm import LinearSVR
>>> from sklearn.pipeline import make_pipeline
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.datasets import make_regression
>>> X, y = make_regression(n_features=4, random_state=0)
>>> regr = make_pipeline(StandardScaler(),
...                      LinearSVR(random_state=0, tol=1e-5))
>>> regr.fit(X, y)
Pipeline(steps=[('standardscaler', StandardScaler()),
                ('linearsvr', LinearSVR(random_state=0, tol=1e-05))])
```

```
>>> print(regr.named_steps['linearsvr'].coef_)
[18.582... 27.023... 44.357... 64.522...]
>>> print(regr.named_steps['linearsvr'].intercept_)
[-4...]
>>> print(regr.predict([[0, 0, 0, 0]]))
[-2.384...]
```

Module `src.concrete.ml.sklearn.torch_module`

Implement torch module.

Module `src.concrete.ml.sklearn.tree`

Implement the sklearn tree models.

Classes

Class `DecisionTreeClassifier`

```
class DecisionTreeClassifier(
    criterion='gini',
    splitter='best',
    max_depth=None,
    min_samples_split=2,
    min_samples_leaf=1,
    min_weight_fraction_leaf=0.0,
    max_features=None,
    random_state=None,
```

```
        max_leaf_nodes=None,
        min_impurity_decrease=0.0,
        class_weight=None,
        ccp_alpha: float = 0.0,
        n_bits: int = 6
    )
```

Implements the sklearn DecisionTreeClassifier.

Initialize the DecisionTreeClassifier.

noqa: DAR101

Ancestors (in MRO)

- src.concrete.ml.sklearn.base.BaseTreeEstimatorMixin
- sklearn.base.BaseEstimator
- sklearn.base.ClassifierMixin

Class variables

Variable `class_mapping_` Type: `Optional[dict]`

Variable `fhe_tree` Type: `concrete.numpy.compilation.circuit.Circuit`

Variable `framework` Type: `str`

Variable `n_classes_` Type: `int`

Variable `q_x_byfeatures` Type: `list`

Variable `q_y` Type: `src.concrete.ml.quantization.quantized_array.QuantizedArray`

Variable `sklearn_alg` Type: `Any`

A decision tree classifier.

Read more in the :ref:`User Guide <tree>`.

Parameters

`criterion` : {`"gini"`, `"entropy"`, `"log_loss"`}, default=`"gini"` The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "log_loss" and "entropy" both for the Shannon information gain, see :ref:tree_mathematical_formulation.

`splitter` : {`"best"`, `"random"`}, default=`"best"` The strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split.

`max_depth` : int, default=None The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

`min_samples_split` : int or float, default=2 The minimum number of samples required to split an internal node:

- If int, then consider min_samples_split as the minimum number.
- If float, then min_samples_split is a fraction and `ceil(min_samples_split * n_samples)` are the minimum number of samples for each split.

Changed in version: 0.18: Added float values for fractions.

min_samples_leaf : int or float, default=1 The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least min_samples_leaf training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

- If int, then consider min_samples_leaf as the minimum number.
- If float, then min_samples_leaf is a fraction and `ceil(min_samples_leaf * n_samples)` are the minimum number of samples for each node.

Changed in version: 0.18: Added float values for fractions.

min_weight_fraction_leaf : float, default=0.0 The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

max_features : int, float or {"auto", "sqrt", "log2"}, default=None The number of features to consider when looking for the best split:

```
- If int, then consider <code>max\_features</code> features at each split.
- If float, then <code>max\_features</code> is a fraction and
  `max(1, int(max_features * n_features_in_))` features are considered at
  each split.
- If "auto", then `max_features=sqrt(n_features)`.
- If "sqrt", then `max_features=sqrt(n_features)`.
- If "log2", then `max_features=log2(n_features)`.
- If None, then `max_features=n_features`.

**Deprecated since version: 1.1:**
The `"auto"` option was deprecated in 1.1 and will be removed
in 1.3.
```

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features.

random_state : int, RandomState instance or None, default=None Controls the randomness of the estimator. The features are always randomly permuted at each split, even if splitter is set to `"best"`. When `max_features < n_features`, the algorithm will select max_features at random at each split before finding the best split among them. But the best found split may vary across different runs, even if `max_features=n_features`. That is the case, if the improvement of the criterion is identical for several splits and one split has to be selected at random. To obtain a deterministic behaviour during fitting, random_state has to be fixed to an integer. See :term:Glossary <random_state> for details.

max_leaf_nodes : int, default=None Grow a tree with max_leaf_nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

min_impurity_decrease : float, default=0.0 A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following::

```
N_t / N * (impurity - N_t_R / N_t * right_impurity
                    - N_t_L / N_t * left_impurity)
```

where N is the total number of samples, N_t is the number of samples at the current node, N_t_L is the number of samples in the left child, and N_t_R is the number of samples in the right child.

N, N_t, N_t_R and N_t_L all refer to the weighted sum, if sample_weight is passed.

Added in version: 0.19:

class_weight : dict, list of dict or "balanced", default=None Weights associated with classes in the form `{class_label: weight}`. If None, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of y.

Note that for multioutput (including multilabel) weights should be defined for each class of every column in its own dict. For example, for four-class multilabel classification weights should be [{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}] instead of [{1:1}, {2:5}, {3:1}, {4:1}].

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`

For multi-output, the weights of each column of y will be multiplied.

Note that these weights will be multiplied with sample_weight (passed through the fit method) if sample_weight is specified.

`ccp_alpha` : `non-negative float`, default=0.0 Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than ccp_alpha will be chosen. By default, no pruning is performed. See :ref:minimal_cost_complexity_pruning for details.

Added in version: 0.22:

Attributes

`classes_` : ndarray of shape (n_classes,) or list of ndarray The classes labels (single output problem), or a list of arrays of class labels (multi-output problem).

`feature_importances_` : ndarray of shape (n_features,) The impurity-based feature importances. The higher, the more important the feature. The importance of a feature is computed as the (normalized) total reduction of the criterion brought by that feature. It is also known as the Gini importance [4]_.

Warning: impurity-based feature importances can be misleading for high cardinality features (many unique values). See :func:sklearn.inspection.permutation_importance as an alternative.

`max_features_` : int The inferred value of max_features.

`n_classes_` : int or list of int The number of classes (for single output problems), or a list containing the number of classes for each output (for multi-output problems).

`n_features_` : int The number of features when fit is performed.

Deprecated since version: 1.0: n_features_ is deprecated in 1.0 and will be removed in 1.2. Use n_features_in_ instead.

`n_features_in_` : int Number of features seen during :term:fit.

Added in version: 0.24:

`feature_names_in_` : ndarray of shape (n_features_in_,) Names of features seen during :term:fit. Defined only when X has feature names that are all strings.

Added in version: 1.0:

`n_outputs_` : int The number of outputs when fit is performed.

`tree_` : Tree instance The underlying Tree object. Please refer to help(sklearn.tree._tree.Tree) for attributes of Tree object and :ref:sphx_glr_auto_examples_tree_plot_unveil_tree_structure.py for basic usage of these attributes.

See Also

DecisionTreeRegressor A decision tree regressor.

Notes

The default values for the parameters controlling the size of the trees (e.g. max_depth, min_samples_leaf, etc.) lead to fully grown and unpruned trees which can potentially be very large on some data sets. To reduce memory consumption, the complexity and size of the trees should be controlled by setting those parameter values.

The :meth:predict method operates using the :func:numpy.argmax function on the outputs of :meth:predict_proba. This means that in case the highest predicted probabilities are tied, the classifier will predict the tied class with the lowest index in :term:classes_.

References

.. [1] https://en.wikipedia.org/wiki/Decision_tree_learning

.. [2] L. Breiman, J. Friedman, R. Olshen, and C. Stone, "Classification and Regression Trees", Wadsworth, Belmont, CA, 1984.

.. [3] T. Hastie, R. Tibshirani and J. Friedman. "Elements of Statistical Learning", Springer, 2009.

.. [4] L. Breiman, and A. Cutler, "Random Forests", https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm

Examples

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import cross_val_score
>>> from sklearn.tree import DecisionTreeClassifier
>>> clf = DecisionTreeClassifier(random_state=0)
>>> iris = load_iris()
>>> cross_val_score(clf, iris.data, iris.target, cv=10)
...                             # doctest: +SKIP
...
array([ 1.     ,  0.93...,  0.86...,  0.93...,  0.93...,
        0.93...,  0.93...,  1.     ,  0.93...,  1.      ])
```

Module `src.concrete.ml.sklearn.tree_to_numpy`

Implements the conversion of a tree model to a numpy function.

Functions

Function `tree_to_numpy`

```
def tree_to_numpy(
    model: onnx.onnx_ml_pb2.ModelProto,
    x: numpy.ndarray,
    framework: str,
    output_n_bits: Optional[int] = 8
) -> Tuple[Callable, List[src.concrete.ml.quantization.quantized_array.UniformQuantizer], onnx.
```

Convert the tree inference to a numpy functions using Hummingbird.

Args —–= **`model`** : onnx.ModelProto : The model to convert.

`x` : numpy.ndarray The input data.
`framework` : str The framework from which the onnx_model is generated. (options: 'xgboost', 'sklearn')
`output_n_bits` : int The number of bits of the output.

Returns —–= Tuple[Callable, List[QuantizedArray], onnx.ModelProto] : A tuple with a function that takes a numpy array and returns a numpy array, QuantizedArray object to quantize and dequantize the output of the tree, and the ONNX model.

Module `src.concrete.ml.sklearn.xgb`

Implements XGBoost models.

Classes

Class XGBClassifier

```
class XGBClassifier(
    n_bits: int = 6,
    max_depth: Optional[int] = 3,
    learning_rate: Optional[float] = 0.1,
    n_estimators: Optional[int] = 20,
    objective: Optional[str] = 'binary:logistic',
    booster: Optional[str] = None,
    tree_method: Optional[str] = None,
    n_jobs: Optional[int] = None,
    gamma: Optional[float] = None,
    min_child_weight: Optional[float] = None,
    max_delta_step: Optional[float] = None,
    subsample: Optional[float] = None,
    colsample_bytree: Optional[float] = None,
    colsample_bylevel: Optional[float] = None,
    colsample_bynode: Optional[float] = None,
    reg_alpha: Optional[float] = None,
    reg_lambda: Optional[float] = None,
    scale_pos_weight: Optional[float] = None,
    base_score: Optional[float] = None,
    missing: float = nan,
    num_parallel_tree: Optional[int] = None,
    monotone_constraints: Union[Dict[str, int], str, None] = None,
    interaction_constraints: Union[str, List[Tuple[str]], None] = None,
    importance_type: Optional[str] = None,
    gpu_id: Optional[int] = None,
    validate_parameters: Optional[bool] = None,
    predictor: Optional[str] = None,
    enable_categorical: bool = False,
    use_label_encoder: bool = False,
    random_state: Union[numpy.random.mtrand.RandomState, int, None] = None,
    verbosity: Optional[int] = None
)
```

Implements the XGBoost classifier.

Initialize the TreeBasedEstimatorMixin.

Args ——= **n_bits** : int : number of bits used for quantization

Ancestors (in MRO)

- src.concrete.ml.sklearn.base.BaseTreeEstimatorMixin
- sklearn.base.BaseEstimator
- sklearn.base.ClassifierMixin

Class variables

Variable framework Type: `str`

Variable n_bits Type: `int`

Variable n_classes_ Type: `int`

Variable output_quantizers Type: `List[concrete.ml.quantization.quantized_array.UniformQuantizer]`

Variable `q_x_byfeatures` Type: `List[src.concrete.ml.quantization.quantized_array.QuantizedArray]`

Variable `sklearn_alg` Type: `Any`

Implementation of the scikit-learn API for XGBoost classification.

Parameters

```
n_estimators : int
    Number of boosting rounds.

max_depth :  Optional[int]
    Maximum tree depth for base learners.
max_leaves :
    Maximum number of leaves; 0 indicates no limit.
max_bin :
    If using histogram-based algorithm, maximum number of bins per feature
grow_policy :
    Tree growing policy. 0: favor splitting at nodes closest to the node, i.e. grow
    depth-wise. 1: favor splitting at nodes with highest loss change.
learning_rate : Optional[float]
    Boosting learning rate (xgb's "eta")
verbosity : Optional[int]
    The degree of verbosity. Valid values are 0 (silent) - 3 (debug).
objective : typing.Union[str, typing.Callable[[numpy.ndarray, numpy.ndarray], typing.Tuple[numpy.nda
    Specify the learning task and the corresponding learning objective or
    a custom objective function to be used (see note below).
booster: Optional[str]
    Specify which booster to use: gbtree, gblinear or dart.
tree_method: Optional[str]
    Specify which tree method to use.  Default to auto.  If this parameter is set to
    default, XGBoost will choose the most conservative option available.  It's
    recommended to study this option from the parameters document :doc:`tree method
    </treemethod>`
n_jobs : Optional[int]
    Number of parallel threads used to run xgboost.  When used with other Scikit-Learn
    algorithms like grid search, you may choose which algorithm to parallelize and
    balance the threads.  Creating thread contention will significantly slow down both
    algorithms.
gamma : Optional[float]
    (min_split_loss) Minimum loss reduction required to make a further partition on a
    leaf node of the tree.
min_child_weight : Optional[float]
    Minimum sum of instance weight(hessian) needed in a child.
max_delta_step : Optional[float]
    Maximum delta step we allow each tree's weight estimation to be.
subsample : Optional[float]
    Subsample ratio of the training instance.
sampling_method :
    Sampling method. Used only by <code>gpu\_hist</code> tree method.
      - <code>uniform</code>: select random training instances uniformly.
      - <code>gradient\_based</code> select random training instances with higher probability when
        the gradient and hessian are larger. (cf. CatBoost)
colsample_bytree : Optional[float]
    Subsample ratio of columns when constructing each tree.
colsample_bylevel : Optional[float]
    Subsample ratio of columns for each level.
colsample_bynode : Optional[float]
    Subsample ratio of columns for each split.
```

```
reg_alpha : Optional[float]
    L1 regularization term on weights (xgb's alpha).
reg_lambda : Optional[float]
    L2 regularization term on weights (xgb's lambda).
scale_pos_weight : Optional[float]
    Balancing of positive and negative weights.
base_score : Optional[float]
    The initial prediction score of all instances, global bias.
random_state : Optional[Union[numpy.random.RandomState, int]]
    Random number seed.

    **Note:**
    Using gblinear booster with shotgun updater is nondeterministic as
    it uses Hogwild algorithm.

missing : float, default np.nan
    Value in the data which needs to be present as a missing value.
num_parallel_tree: Optional[int]
    Used for boosting random forest.
monotone_constraints : Optional[Union[Dict[str, int], str]]
    Constraint of variable monotonicity.  See :doc:`tutorial </tutorials/monotonic>`
    for more information.
interaction_constraints : Optional[Union[str, List[Tuple[str]]]]
    Constraints for interaction representing permitted interactions.  The
    constraints must be specified in the form of a nested list, e.g. ``[[0, 1], [2,
    3, 4]]``, where each inner list is a group of indices of features that are
    allowed to interact with each other.  See :doc:`tutorial
    </tutorials/feature_interaction_constraint>` for more information
importance_type: Optional[str]
    The feature importance type for the feature_importances\_ property:

    * For tree model, it's either "gain", "weight", "cover", "total_gain" or
      "total_cover".
    * For linear model, only "weight" is defined and it's the normalized coefficients
      without bias.

gpu_id : Optional[int]
    Device ordinal.
validate_parameters : Optional[bool]
    Give warnings for unknown parameter.
predictor : Optional[str]
    Force XGBoost to use specific predictor, available choices are [cpu_predictor,
    gpu_predictor].
enable_categorical : bool

    **Added in version: 1.5.0:**


    **Note: This parameter is experimental:**


    Experimental support for categorical data.  When enabled, cudf/pandas.DataFrame
    should be used to specify categorical data type.  Also, JSON/UBJSON
    serialization format is required.

max_cat_to_onehot : Optional[int]

    **Added in version: 1.6.0:**
```

```
    **Note: This parameter is experimental:**


    A threshold for deciding whether XGBoost should use one-hot encoding based split
    for categorical data.  When number of categories is lesser than the threshold
    then one-hot encoding is chosen, otherwise the categories will be partitioned
    into children nodes.  Only relevant for regression and binary classification.
    See :doc:`Categorical Data </tutorials/categorical>` for details.

eval_metric : Optional[Union[str, List[str], Callable]]

    **Added in version: 1.6.0:**


    Metric used for monitoring the training result and early stopping.  It can be a
    string or list of strings as names of predefined metric in XGBoost (See
    doc/parameter.rst), one of the metrics in :py:mod:<code>sklearn.metrics</code>, or any other
    user defined metric that looks like <code>sklearn.metrics</code>.

    If custom objective is also provided, then custom metric should implement the
    corresponding reverse link function.

    Unlike the <code>scoring</code> parameter commonly used in scikit-learn, when a callable
    object is provided, it's assumed to be a cost function and by default XGBoost will
    minimize the result during early stopping.

    For advanced usage on Early stopping like directly choosing to maximize instead of
    minimize, see :py:obj:<code>xgboost.callback.EarlyStopping</code>.

    See :doc:`Custom Objective and Evaluation Metric </tutorials/custom_metric_obj>`
    for more.

    **Note:**
    This parameter replaces <code>eval\_metric</code> in :py:meth:<code>fit</code> method.  The old
    receives un-transformed prediction regardless of whether custom objective is
    being used.

    .. code-block:: python

        from sklearn.datasets import load_diabetes
        from sklearn.metrics import mean_absolute_error
        X, y = load_diabetes(return_X_y=True)
        reg = xgb.XGBRegressor(
            tree_method="hist",
            eval_metric=mean_absolute_error,
        )
        reg.fit(X, y, eval_set=[(X, y)])

early_stopping_rounds : Optional[int]

    **Added in version: 1.6.0:**


    Activates early stopping. Validation metric needs to improve at least once in
    every **early_stopping_rounds** round(s) to continue training.  Requires at least
    one item in **eval_set** in :py:meth:<code>fit</code>.
```

```
    The method returns the model from the last iteration (not the best one).  If
    there's more than one item in **eval_set**, the last entry will be used for early
    stopping.  If there's more than one metric in **eval_metric**, the last metric
    will be used for early stopping.

    If early stopping occurs, the model will have three additional fields:
    :py:attr:<code>best\_score</code>, :py:attr:<code>best\_iteration</code> and
    :py:attr:<code>best\_ntree\_limit</code>.

    **Note:**
    This parameter replaces <code>early\_stopping\_rounds</code> in :py:meth:<code>fit</code> method

callbacks : Optional[List[TrainingCallback]]
    List of callback functions that are applied at end of each iteration.
    It is possible to use predefined callbacks by using
    :ref:`Callback API <callback_api>`.

    **Note:**
    States in callback are not preserved during training, which means callback
    objects can not be reused for multiple training sessions without
    reinitialization or deepcopy.

    .. code-block:: python

        for params in parameters_grid:
            # be sure to (re)initialize the callbacks before each run
            callbacks = [xgb.callback.LearningRateScheduler(custom_rates)]
            xgboost.train(params, Xy, callbacks=callbacks)

kwargs : dict, optional
    Keyword arguments for XGBoost Booster object.  Full documentation of parameters
    can be found :doc:`here </parameter>`.
    Attempting to set a parameter via the constructor args and \*\*kwargs
    dict simultaneously will result in a TypeError.

    **Note: \*\*kwargs unsupported by scikit-learn:**
    \*\*kwargs is unsupported by scikit-learn.  We do not guarantee
    that parameters passed via this argument will interact properly
    with scikit-learn.

    **Note: Custom objective function:**
    A custom objective function can be provided for the <code>objective</code>
    parameter. In this case, it should have the signature
    ``objective(y_true, y_pred) -> grad, hess``:

    y_true: array_like of shape [n_samples]
        The target values
    y_pred: array_like of shape [n_samples]
        The predicted values

    grad: array_like of shape [n_samples]
        The value of the gradient for each sample point.
    hess: array_like of shape [n_samples]
        The value of the second derivative for each sample point
```

Variable `sklearn_model` Type: `Any`

Methods

Method `update_post_processing_params`

```
def update_post_processing_params(
    self
)
```

Update the post processing params.

Module `src.concrete.ml.torch`

Modules for torch to numpy conversion.

Sub-modules

- src.concrete.ml.torch.compile
- src.concrete.ml.torch.numpy_module

Module `src.concrete.ml.torch.compile`

torch compilation function.

Functions

Function `compile_onnx_model`

```
def compile_onnx_model(
    onnx_model: onnx.onnx_ml_pb2.ModelProto,
    torch_inputset: Union[torch.Tensor, numpy.ndarray, Tuple[Union[torch.Tensor, numpy.ndarray]
    import_qat: bool = False,
    configuration: Optional[concrete.numpy.compilation.configuration.Configuration] = None,
    compilation_artifacts: Optional[concrete.numpy.compilation.artifacts.DebugArtifacts] = None
    show_mlir: bool = False,
    n_bits=8,
    use_virtual_lib: bool = False,
    p_error: Optional[float] = 6.3342483999973e-05
) -> src.concrete.ml.quantization.quantized_module.QuantizedModule
```

Compile a torch module into an FHE equivalent.

Take a model in torch, turn it to numpy, quantize its inputs / weights / outputs and finally compile it with Concrete-Numpy

Args —–= **`onnx_model`** : onnx.ModelProto : the model to quantize

`torch_inputset` : Dataset the inputset, can contain either torch tensors or numpy.ndarray, only datasets with a single input are supported for now.
`import_qat` : bool Flag to signal that the network being imported contains quantizers in in its computation graph and that Concrete ML should not re-quantize it.
`configuration` : Configuration Configuration object to use during compilation
`compilation_artifacts` : DebugArtifacts Artifacts object to fill during compilation
`show_mlir` : bool if set, the MLIR produced by the converter and which is going to be sent to the compiler backend is shown on the screen, e.g., for debugging or demo
`n_bits` the number of bits for the quantization
`use_virtual_lib` : bool set to use the so called virtual lib simulating FHE computation. Defaults to False.
`p_error` : Optional[float] probability of error of a PBS

Returns —–= QuantizedModule : The resulting compiled QuantizedModule.

Function `compile_torch_model`

```
def compile_torch_model(
    torch_model: torch.nn.modules.module.Module,
    torch_inputset: Union[torch.Tensor, numpy.ndarray, Tuple[Union[torch.Tensor, numpy.ndarray]
    import_qat: bool = False,
    configuration: Optional[concrete.numpy.compilation.configuration.Configuration] = None,
    compilation_artifacts: Optional[concrete.numpy.compilation.artifacts.DebugArtifacts] = None
    show_mlir: bool = False,
    n_bits=8,
    use_virtual_lib: bool = False,
    p_error: Optional[float] = 6.3342483999973e-05
) -> src.concrete.ml.quantization.quantized_module.QuantizedModule
```

Compile a torch module into an FHE equivalent.

Take a model in torch, turn it to numpy, quantize its inputs / weights / outputs and finally compile it with Concrete-Numpy

Args —–= **`torch_model`** : torch.nn.Module : the model to quantize

`torch_inputset` : Dataset the inputset, can contain either torch tensors or numpy.ndarray, only datasets with a single input are supported for now.
`import_qat` : bool Set to True to import a network that contains quantizers and was trained using quantization aware training
`configuration` : Configuration Configuration object to use during compilation
`compilation_artifacts` : DebugArtifacts Artifacts object to fill during compilation
`show_mlir` : bool if set, the MLIR produced by the converter and which is going to be sent to the compiler backend is shown on the screen, e.g., for debugging or demo
`n_bits` the number of bits for the quantization
`use_virtual_lib` : bool set to use the so called virtual lib simulating FHE computation. Defaults to False
`p_error` : Optional[float] probability of error of a PBS

Returns —–= QuantizedModule : The resulting compiled QuantizedModule.

Function `convert_torch_tensor_or_numpy_array_to_numpy_array`

```
def convert_torch_tensor_or_numpy_array_to_numpy_array(
    torch_tensor_or_numpy_array: Union[torch.Tensor, numpy.ndarray]
) -> numpy.ndarray
```

Convert a torch tensor or a numpy array to a numpy array.

Args —–= **`torch_tensor_or_numpy_array`** : Tensor : the value that is either a torch tensor or a numpy array.

Returns —–= numpy.ndarray : the value converted to a numpy array.

Module `src.concrete.ml.torch.numpy_module`

A torch to numpy module.

Classes

Class `NumpyModule`

```
class NumpyModule(
    model: Union[torch.nn.modules.module.Module, onnx.onnx_ml_pb2.ModelProto],
    dummy_input: Union[torch.Tensor, Tuple[torch.Tensor, ...], None] = None,
    debug_onnx_output_file_path: Union[pathlib.Path, str, None] = None
)
```

General interface to transform a torch.nn.Module to numpy module.

Args —–= **`torch_model`** : Union[nn.Module, onnx.ModelProto] : A fully trained, torch model along with its parameters or the onnx graph of the model.

`dummy_input` : Union[torch.Tensor, Tuple[torch.Tensor, ...]] Sample tensors for all the module inputs, used in the ONNX export to get a simple to manipulate nn representation.

`debug_onnx_output_file_path` (Optional[Union[Path, str]], optional): An optional path to indicate where to save the ONNX file exported by torch for debug. Defaults to None.

Instance variables

Variable `onnx_model` Get the ONNX model.

.. # noqa: DAR201

Returns —–= _onnx_model (onnx.ModelProto): the ONNX model

Methods

Method `forward`

```
def forward(
    self,
    *args: numpy.ndarray
) -> Union[Tuple[numpy.ndarray, ...], numpy.ndarray]
```

Apply a forward pass on args with the equivalent numpy function only.

Args —–= **`*args`** : the inputs of the forward function

Returns —–= Union[numpy.ndarray, Tuple[numpy.ndarray, ...]] : result of the forward on the given inputs

Module `src.concrete.ml.version`

File to manage the version of the package.

Generated by *pdoc* 0.10.0 (https://pdoc3.github.io).