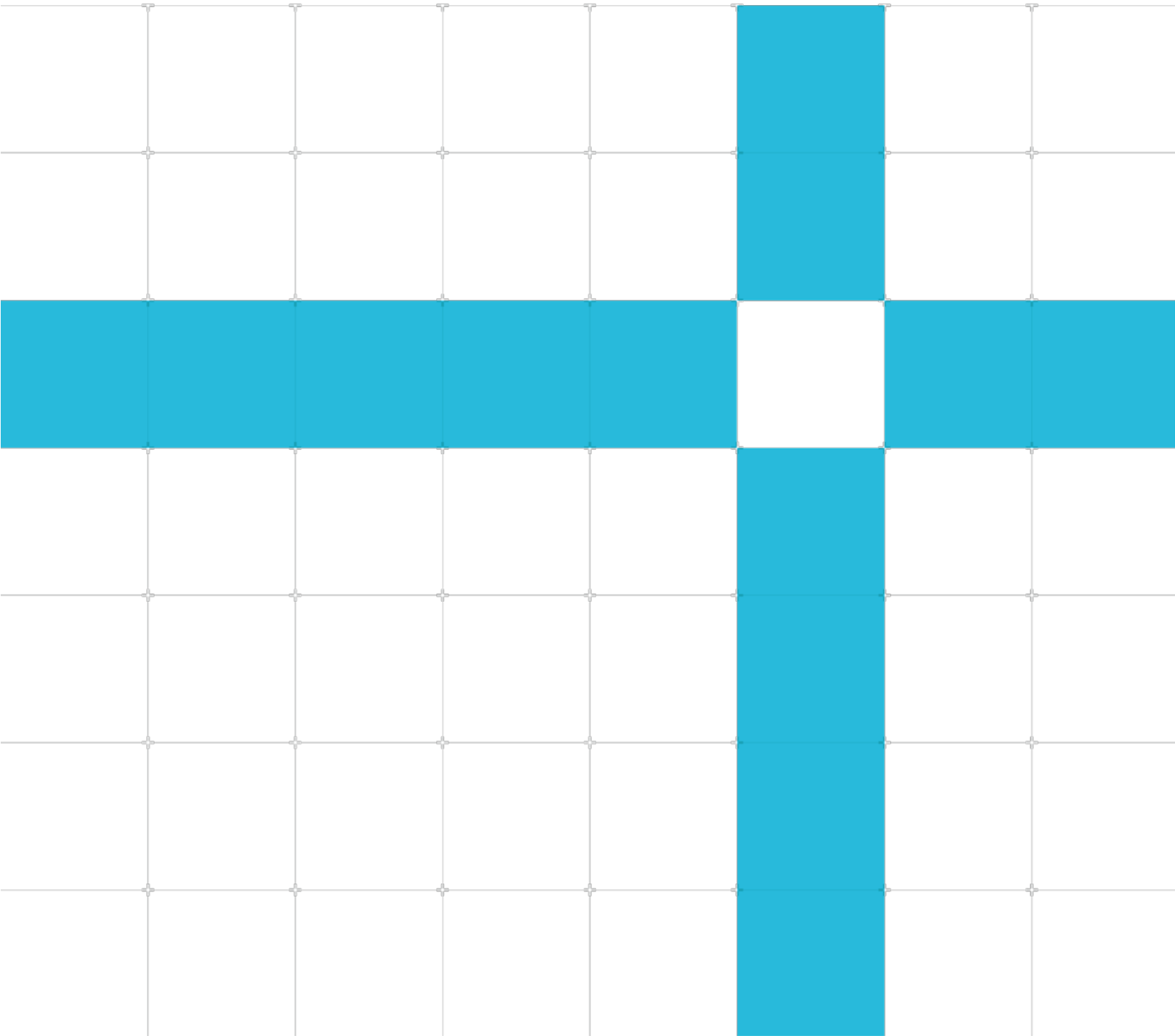
Add a new operator to Arm NN

Non-Confidential

Issue 1.1
ARM062-948681440-3526



Add a new operator to Arm NN

Release information

Document history

Issue	Date	Confidentiality	Change
01	16 October 2020	Non-Confidential	First release
02	21 October 2020	Non-Confidential	Updates Before you begin section details on how to clone Arm NN from Linaro ML Platform

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

Confidentiality Status

Product Status

The information in this document is Final, that is for a developed product.

Web Address

www.arm.com

Contents

1 Overview

Adding a new operator to Arm NN involves several steps. In this guide, we show you all the steps to add an example Softmax operator, `SoftmaxLayer`, to the reference backend. We use the example of adding a `SoftmaxLayer` operator to the reference backend because Arm NN controls the reference backend. If you want to enable an operator in a different backend, implement the backend-specific workloads when you add the reference workload support.

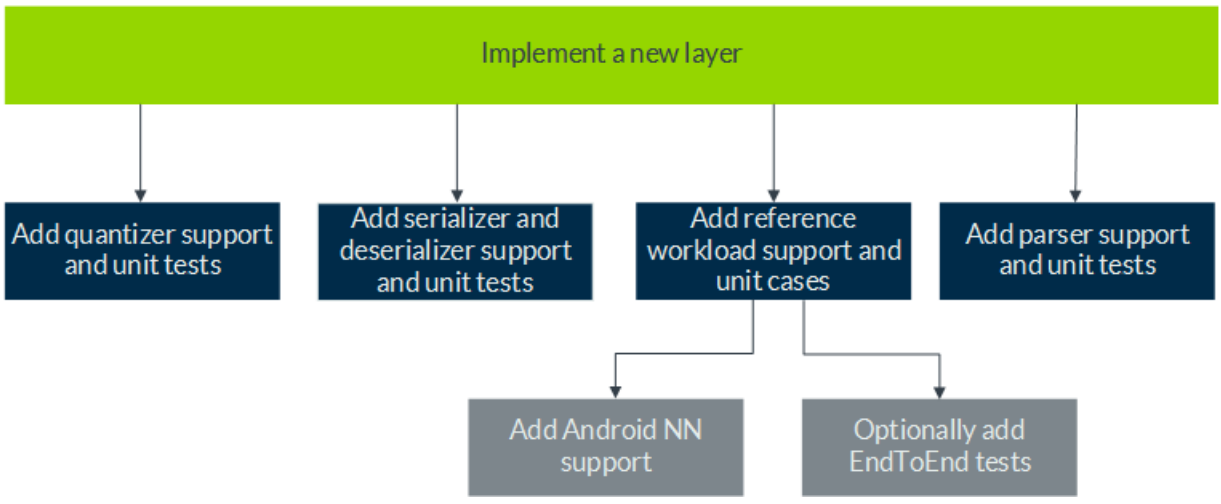
When a new operator has been added to Arm NN, you must add the new operator to the following programs and libraries. These program and libraries are extra features of Arm NN:

- The `armnnSerializer` is a library for serializing an Arm NN network to a stream.
- The `armnnDeserializer` is a library for loading neural networks defined by Arm NN FlatBuffers files into the Arm NN runtime.
- The `ArmnnQuantizer` is a program that loads a 32-bit float network into Arm NN and converts it into a quantized asymmetric 8-bit network, or a quantized symmetric 16-bit network.

This guide also includes details on how to support the Android NN Driver and support for any required parsers. Both the Android NN driver and parsers are optional.

The following figure shows the steps that you must follow to add a new operator:

Figure 1-1 The process to add a new operator



2 Before you begin

This guide assumes that you have a working Arm NN installation. If you do not have a working installation, Arm NN is available to clone from both **Github** and **Linaro ML Platform**.

However, if you wish to submit your new operator to be included in the Arm NN source code you must clone Arm NN from Linaro ML Platform.

3 Add boilerplate frontend support

Before you add any specific backend implementation for a new operator, you must first add basic boilerplate front-end support.

3.1 Add a descriptor

A descriptor enables operators to use unique configurable parameters. Arm NN uses a descriptor class to describe the parameters. For example, to describe the parameters that are commonly found in the convolution operator, you create a descriptor called `Convolution2dDescriptor` and initialize it with padding, stride, and dilation parameters.

It is not always necessary to implement a descriptor. For example, the `Abs` operator takes one input and outputs the output as an absolute value. Therefore, the `Abs` operator does not need any extra parameters and the corresponding `AbsLayer` does not require a descriptor.

To add a descriptor:

1. Define a new descriptor at `armnn/include/armnn/Descriptors.hpp`. The new descriptor must have unique parameters.
2. Include the new descriptor in the list in `DescriptorsFwd.hpp`. This list is used for forward declaration to reduce build times.

3.2 Add default layer support

A layer is a representation of an operator in the Arm NN network graph. To represent an operator as a layer in an Arm NN network graph, first add default layer support.

To add default layer support:

1. Define `Is<LayerName>Supported()` as a virtual function in the `ILayerSupport` interface at `armnn/include/armnn/ILayerSupport.hpp`. In the `Is<LayerName>Supported()` function, include a layer-specific descriptor, to check parameters to verify that a layer is supported. The following example shows the `Is<LayerName>Supported()` function for a `SoftmaxLayer` operator:

```
virtual bool IsSoftmaxSupported(const TensorInfo& input,
                                const TensorInfo& output,
                                const SoftmaxDescriptor& descriptor,
                                Optional<std::string&> reasonIfUnsupported =
EmptyOptional()) const = 0;
```

2. Add `Is<LayerName>Supported()` to `backends/backendsCommon/LayerSupportBase.hpp` to update the class `LayerSupportBase`. This addition extends `IlayerSupport`.
3. Enable basic front-end support by using the `DefaultLayerSupport()` function to implement `LayerSupportBase.cpp`. The `DefaultLayerSupport()` returns a `false` value. The following code shows an example implementation of `LayerSupportBase.cpp` for a `SoftmaxLayer` operator.

```
bool LayerSupportBase::IsSoftmaxSupported(const TensorInfo&, // input
                                          const TensorInfo&, // output
                                          const SoftmaxDescriptor&, // descriptor
                                          Optional<std::string&> reasonIfUnsupported) const
{
    return DefaultLayerSupport(__func__, __FILE__, __LINE__, reasonIfUnsupported);
}
```

3.3 Add a layer class implementation

Arm NN represents a new operator by using a `<LayerName>Layer` class.

To add your layer class implementation:

1. Add a layer class to `armnn/src/armnn/layers/<LayerName>Layer.hpp` and an implementation to `armnn/src/armnn/layers/<LayerName>Layer.cpp`. Depending on the type of layer, the layer can extend any of the following layer classes:
   - `Layer`. This means that operator layers that do not have configurable parameters extend the `Layer` class.
   - `LayerWithParameters<<LayerType>Descriptor>`. This means that operator layers that have configurable parameters like `Convolution2dLayer`, `DepthwiseConvolution2dLayer`, or `FullyConnectedLayer`, extend the `LayerWithParameters<<LayerType>Descriptor>` layer class. For example, `LayerWithParameters<DepthwiseConvolution2dDescriptor> extends the LayerWithParameters<<LayerType>Descriptor>` layer class. These layers require a descriptor.
   - `ElementwiseBaseLayer`. This means that operator layers that encapsulate element by element operations extend this `ElementwiseBaseLayer` class. For example, the `Maximum`, `Minimum`, `Multiplication`, `Division`, `Subtraction`, or `Addition operator layers extend the ElementwiseBaseLayer` class.
2. Add the layer to the list of layer types in `armnn/src/armnn/InternalTypes.hpp`. The following code shows the `InternalTypes.hpp`:

```
/// This list uses X macro technique.
/// See https://en.wikipedia.org/wiki/X_Macro for more info
#define LIST_OF_LAYER_TYPE \
    X(Activation) \
    X(Addition) \
    X(ArgMinMax) \
    X(BatchNormalization) \
    ...
```

Include `<LayerName>Layer.hpp` and declare the layer in `armnn/src/armnn/LayersFwd.hpp f`or forward declaration and to reduce build times. The following code includes `<LayerName>Layer.hpp`:

```
#include "layers/SoftmaxLayer.hpp"
```

```
...
...
DECLARE_LAYER(Softmax)
```

3. If you are adding Android support, add `<LayerName>Layer.cpp` to the list of layers in the `armnn/Android.mk` file.
4. Add `<LayerName>Layer.hpp` and `<LayerName>Layer.cpp` to the list of layers in the `armnn/CMakeLists.txt` file.
5. Add the virtual function `Visit<LayerName>Layer()` to `armnn/include/armnn/ILayerVisitor.hpp`. The following code shows the inclusion of an example `SoftmaxLayer` to the `ILayerVisitor.hpp`:

```
Public:
    /// Function that a softmax layer should callback to when its Accept(ILayerVisitor&)
function is invoked.
    /// @param layer - pointer to the layer which is calling back to this visit function.
    /// @param softmaxDescriptor - SoftmaxDescriptor to configure the softmax.
    /// @param name - Optional name for the layer.
    virtual void VisitSoftmaxLayer(const IConnectableLayer* layer,
                                   const SoftmaxDescriptor& softmaxDescriptor,
                                   const char* name = nullptr) = 0;
```

6. Add `Visit<LayerName>Layer()` to the `armnn/include/armnnLayerVisitorBase.hpp` implementation. The following code shows this addition for an example `SoftmaxLayer`:

```
Public:
    void VisitSoftmaxLayer(const IConnectableLayer*,
                           const SoftmaxDescriptor&,
                           const char*) override { DefaultPolicy::Apply(__func__); }
```

7. Add `Visit<LayerName>Layer()` to `src/armnnSerializer/Serializer.hpp`. Implement the visit function of the new operator or layer so that it throws an unimplemented exception in `Serializer.cpp`. The following code shows this addition for an example `SoftmaxLayer`:

```
void VisitSoftmaxLayer(const armnn::IConnectableLayer* layer,
                           const armnn::SoftmaxDescriptor& softmaxDescriptor,
                           const char* name = nullptr) override;
```

8. Add the virtual function `Add<LayerName>Layer()` to `armnn/include/armnn/INetwork.hpp`.
9. Add the implementation of `Add<LayerName>Layer()` to `armnn/src/armnn/Network.hpp` and `Network.cpp`. The following code shows this addition for an example `SoftmaxLayer`:

```
IConnectableLayer* AddSoftmaxLayer(const SoftmaxDescriptor& softmaxDescriptor,
        const char* name = nullptr) override;
```

3.4 Add no-op factory implementations for all backends

During optimization, Arm NN assigns a backend to each layer, depending on the information that the application running the network gives. The layer is executed on the assigned backend. The following are the backends:

- `CpuRef` for the reference backend
- `CpuAcc` for the Neon backend
- `GpuAcc` for the compute library backend

Each layer is executed as a workload. Workloads are bits of work that are put in an execution queue. The runtime consumes these workloads from the queue. To create the workload for a layer and backend pair, each layer uses a backend-specific factory. The backend-specific factory is an implementation of `IWorkloadFactory`.

To add a no-op implementation:

1. Define `<LayerName>QueueDescriptor` in `armnn/src/backends/backendsCommon/WorkloadData.hpp`. If a layer requires unique parameters, you must use the `QueueDescriptor` to extend the `QueueDescriptorWithParameters<<LayerName>Descriptor>` class. If a layer does not require unique parameters, you must use the `<LayerName>QueueDescriptor` class to extend the `QueueDescriptor`. Both types of `QueueDescriptor` must provide a `Validate()` function.

   The following code shows this definition for an example `SoftmaxLayer` layer:

   ```
   // Softmax layer workload data.
   struct SoftmaxQueueDescriptor : QueueDescriptorWithParameters<SoftmaxDescriptor>
   {
       void Validate(const WorkloadInfo& workloadInfo) const;
   };
   ```

2. Implement the `<LayerName>QueueDescriptor::Validate()` function at `armnn/src/backends/backendsCommon/WorkloadData.cpp`. The specific validation checks that you must implement depend on the layer. You can find guidance on what you must check in the technical documentation of the different frameworks that provide a description of the operator. Some example checks are:
   - Validate the number of inputs.
   - Validate the number of outputs.
   - Validate that the number of elements in the input tensor and output tensor match.
   - Validate that the data type is supported.
   - Validate the number of dimensions in the input and output tensor.
   - Validate the input width and height.
3. Add a virtual `Create<LayerName>()` function to the `IWorkloadFactory` in `armnn/src/backends/backendsCommon/WorkloadFactory.hpp`. The following code shows this addition for an example `SoftmaxLayer`:

   ```
   virtual std::unique_ptr<IWorkload> CreateSoftmax(const SoftmaxQueueDescriptor& descriptor,
                                                    const WorkloadInfo&           info) const;
   ```

4. Add a `LayerType` switch case for the new layer in the `IsLayerSupprted()` function, in `armnn/src/backends/backendsCommon/WorkloadFactory.cpp`. The following code shows this addition for an example `SoftmaxLayer`:

```
switch(layer.GetType())
{
...
        case LayerType::Softmax:
        {
            auto cLayer = PolymorphicDowncast<const SoftmaxLayer*>(&layer);
            const TensorInfo& input = layer.GetInputSlot(0).GetConnection()->GetTensorInfo();
            const TensorInfo& output = layer.GetOutputSlot(0).GetTensorInfo();
            result = layerSupportObject->IsSoftmaxSupported(OverrideDataType(input, dataType),
                                                            OverrideDataType(output, dataType),
                                                            cLayer->GetParameters(),
                                                            reason);
            break;
        }
...
```

5. Add a default implementation of the `Create<LayerName>()` function in `armnn/src/backends/backendsCommon/WorkloadFactory.cpp`. The following code shows this addition for an example `SoftmaxLayer`:

```
std::unique_ptr<IWorkload> IWorkloadFactory::CreateSoftmax(const SoftmaxQueueDescriptor&
/*descriptor*/,
                                                           const WorkloadInfo& /*info*/) const
{
    return std::unique_ptr<IWorkload>();
}
```

3.5 Add layer visitor unit tests

You must add layer visitor unit tests to `IsLayerSupportedTestImpl.hpp`. These unit tests must check that the Arm NN network graph supports the layer or operator. To do this check, use random values to create the relevant workload for the individual layer or operator.

To add unit tests:

1. Add your layer to `armnn/src/backends/backendsCommon/test/IsLayerSupportedTestImpl.hpp`. The following code shows an example for a layer with constructors that take one parameter, `name`:

```
// Every entry in the armnn::LayerType enum must be accounted for below.
DECLARE_LAYER_POLICY_1_PARAM(Abs)
```

The following code shows an example for a layer with constructors that take two parameters, `descriptor` and `name`:

```
// Every entry in the armnn::LayerType enum must be accounted for below.
DECLARE_LAYER_POLICY_2_PARAM(Softmax)
```

2. Add your layer to `src/armnn/test/TestNameOnlyLayerVisitor.hpp` or `src/armnn/test/TestNameAndDescriptorLayerVisitor.hpp`, depending on your layer constructor parameters.
   - If your layer has no descriptor, use the following form to add it to `TestNameOnlyLayerVisitor.hpp`:

     ```
     DECLARE_TEST_NAME_ONLY_LAYER_VISITOR_CLASS(Addition)
     DECLARE_TEST_NAME_ONLY_LAYER_VISITOR_CLASS(Division)
     ```

   - If your layer has any descriptors, use the following form to add it to `src/armnn/test/TestNameAndDescriptorLayerVisitor.hpp`:

     ```
     DECLARE_TEST_NAME_AND_DESCRIPTOR_LAYER_VISITOR_CLASS(Softmax)
     ```

Arm recommends using the `*_1_PARAM` macros for layers that do not have a descriptor and the `*_2_PARAM` macros for layers that do have a descriptor.

4 Add reference workload support

A backend is an abstraction that maps the layers of a network graph to the hardware that executes the supported layers. Backends create specific workloads for the layers that they support. A workload includes the implementation of the layer and is used to queue a layer for computation. Each layer is executed using a workload.

The `CpuRef` reference backend represents the non-accelerated CPU of the device Arm NN runs on. The `CpuRef` backend is mainly used for testing purposes. Reference backend workloads are stored under `backends/reference/workloads`.

1. Add a workload header and source file for your layer to `src/backends/<BackendName>/workloads/<BackendName><LayerName>Workload.hpp` and `src/backends/<BackendName>/workloads/<BackendName><LayerName>Workload.cpp`.

For example, to implement `src/backends/reference/workloads/RefSoftmaxWorkload.hpp` and `src/backends/reference/workloads/RefSoftmaxWorkload.cpp` for an example `SoftmaxLayer`, make the following changes in `/<BackendName>/<LayerName>Workload.hpp:`

```
#pragma once

#include <backendsCommon/Workload.hpp>
#include <backendsCommon/WorkloadData.hpp>

namespace armnn
{

class RefSoftmaxWorkload : public BaseWorkload<SoftmaxQueueDescriptor>
{
public:
    using BaseWorkload<SoftmaxQueueDescriptor>::BaseWorkload;
    virtual void Execute() const override;
};

} //namespace armnn
```

The following code shows the example changes to implement `src/backends/reference/workloads/RefSoftmaxWorkload.hpp` and `src/backends/reference/workloads/ RefSoftmaxWorkload.cpp` in `/<BackendName><LayerName>Workload.cpp.`

```
#include "RefSoftmaxWorkload.hpp"

...

namespace armnn
{
```

```
void RefSoftmaxWorkload::Execute() const
{
    // Execute Implementation
}
} //namespace armnn
```

2. Add a `Create<LayerName>()` function to the workload factory. The `<LayerName>LayerCreateWorkload()` function calls the `Create<LayerName>()` function as part of the workload creation process. For example, the following code adds the `CreateSoftmax()` function to `RefWorkloadFactory.hpp`:

```
std::unique_ptr<IWorkload> CreateSoftmax(const SoftmaxQueueDescriptor& descriptor,
                                         const WorkloadInfo& info) const override;
```

   The following code adds the example `CreateSoftmax()` function to `RefWorkloadFactor.cpp`:

```
std::unique_ptr<IWorkload> RefWorkloadFactory::CreateSoftmax(const SoftmaxQueueDescriptor&
descriptor,
                                                             const WorkloadInfo& info) const
{
    return std::make_unique<RefSoftmaxWorkload>(descriptor, info);
}
```

3. Add your workload to `backends/reference/workloads/CMakeLists.txt`. The following code shows how to add an example `SoftmaxLayer` workload:

```
RefSliceWorkload.cpp
    RefSliceWorkload.hpp
    RefSoftmaxWorkload.cpp
    RefSoftmaxWorkload.hpp
    RefSpaceToBatchNdWorkload.cpp
    RefSpaceToBatchNdWorkload.hpp
```

Add your workload to `backends/reference/backend.mk` for the Android NN HAL driver. You must add your workload to `backends/reference/backend.mk`. The Android NN HAL driver requires this workload even if you do not plan full Android NN support for the operator. See the Add Android NN support section for more information. The following code shows the addition of a workload for an example `SoftmaxLayer`:

```
workloads/RefSliceWorkload.cpp \
        workloads/RefSoftmaxWorkload.cpp \
        workloads/RefSpaceToBatchNdWorkload.cpp \
```

4. Add your workload to a list in `backends/reference/workloads/RefWorkloads.hpp`. The following code shows the addition of the workload for an example `SoftmaxLayer`:

```
#include "Resize.hpp"
#include "Softmax.hpp"
#include "Splitter.hpp"
```

5. Add the `Is<LayerName>Supported()` function to `backends/reference/RefLayerSupport.hpp` and `backends/reference/RefLayerSupport.cpp`. The calling of this function does not depend on whether the related backend supports the operator `<LayerName>`. Ensure that you add the supported conditions inside the function. The following code shows this addition to the `RefLayerSupport.hpp` file:

```
bool IsSoftmaxSupported(const TensorInfo& input,
                        const TensorInfo& output,
                        const SoftmaxDescriptor& descriptor,
                        Optional<std::string&> reasonIfUnsupported = EmptyOptional()) const
override;
```

   The following code shows this addition to the `RefLayerSupport.cpp` file:

```
bool RefLayerSupport::IsSoftmaxSupported(const TensorInfo& input,
                                         const TensorInfo& output,
                                         const SoftmaxDescriptor& descriptor,
                                         Optional<std::string&> reasonIfUnsupported) const
{
    IgnoreUnused(descriptor);
    bool supported = true;
    std::array<DataType,7> supportedTypes =
    {
        DataType::BFloat16,
        DataType::Float32,
        DataType::Float16,
        DataType::QSymmS8,
        DataType::QAsymmS8,
        DataType::QAsymmU8,
        DataType::QSymmS16
    };

    supported &= CheckSupportRule(TypeAnyOf(input, supportedTypes), reasonIfUnsupported,
                                  "Reference Softmax: output type not supported");

    supported &= CheckSupportRule(TypeAnyOf(output, supportedTypes), reasonIfUnsupported,
                                  "Reference Softmax: input type not supported");

    supported &= CheckSupportRule(TypesAreEqual(input, output), reasonIfUnsupported,
                                  "Reference Softmax: input type not supported");

    return supported;
}
```

4.1 Add layer unit tests

Layer unit tests use the most basic graph possible to check the functionality of each new layer or new operator with different data types and configurations. To do this check, a layer unit test uses a basic graph like InputLayer→NewLayer→OutputLayer. You can write variations of the unit test to use different-sized input tensors and different quantization parameters, configuration parameters, and data types with the basic graph.

To verify the functionality of the new layer or operator, create a unit test suite.

To create a unit test suite:

1. Add the header file, `<LayerName>TestImpl.hpp`, and source file, `<LayerName>TestImpl.cpp`, to `backends/backendsCommon/test/LayerTests`. `<LayerName>` is the name of the layer you are implementing.
2. Declare one or more tests for your new layer in the header file. The following code shows an example unit test for the `SoftmaxLayer`:

```
#pragma once

#include "LayerTestResult.hpp"

#include <Half.hpp>

#include <armnn/backends/IBackendInternal.hpp>
#include <backendsCommon/WorkloadFactory.hpp>

LayerTestResult<float, 2> SimpleSoftmaxTest(
    armnn::IWorkloadFactory& workloadFactory,
    const armnn::IBackendInternal::IMemoryManagerSharedPtr& memoryManager,
    const armnn::ITensorHandleFactory& tensorHandleFactory,
    float beta);
```

3. Implement the unit tests and add template specializations for each data type your layer supports in the source file. The following code shows an example implementation of a unit test for the `SoftmaxLayer` using a `Float32` data type. If supported by the new layer, you must add tests for other data types like `Float16`, `Signed32`, `QAsymmS8`, `QAsymm16`, and `QAsymmU8`:

```
template<armnn::DataType ArmnnType, typename T = armnn::ResolveType<ArmnnType>>
LayerTestResult<T, 2> SimpleSoftmaxTestImpl(
    armnn::IWorkloadFactory& workloadFactory,
    const armnn::IBackendInternal::IMemoryManagerSharedPtr& memoryManager,
    const armnn::ITensorHandleFactory& tensorHandleFactory,
    float beta)
{
    using std::exp;
    const armnn::TensorShape inputShape{ 2, 4 };
```

```
    float x0[4] = { exp((0.f - 1.0f) * beta), exp((1.0f - 1.0f) * beta),
                    exp((0.0f - 1.0f) * beta), exp((0.0f - 1.0f) * beta) };
    float sum0 = x0[0] + x0[1] + x0[2] + x0[3];
    float x1[4] = { exp((0.5f - 0.5f) * beta), exp((0.0f - 0.5f) * beta),
                    exp((0.0f - 0.5f) * beta), exp((0.0f - 0.5f) * beta) };
    float sum1 = x1[0] + x1[1] + x1[2] + x1[3];


    const std::vector<float> outputData = { x0[0] / sum0, x0[1] / sum0, x0[2] / sum0, x0[3] /
sum0,
                                            x1[0] / sum1, x1[1] / sum1, x1[2] / sum1, x1[3] /
sum1 };

    const std::vector<float> inputData =
            {
                0.f, 1.f, 0.f, 0.f,
                .5f, 0.f, 0.f, 0.f,
            };

    return SimpleSoftmaxBaseTestImpl<ArmnnType, 2>(workloadFactory, memoryManager,
tensorHandleFactory, beta,
                                                   inputShape, outputData, inputData);
}


...


LayerTestResult<float,2> SimpleSoftmaxTest(
    armnn::IWorkloadFactory& workloadFactory,
    const armnn::IBackendInternal::IMemoryManagerSharedPtr& memoryManager,
    const armnn::ITensorHandleFactory& tensorHandleFactory,
    float beta)
{
    return SimpleSoftmaxTestImpl<armnn::DataType::Float32>(workloadFactory, memoryManager,
tensorHandleFactory, beta);
}
```

4. Include the test suite in `backends/backendsCommon/test/LayerTests.hpp`. The following code shows this inclusion for the example `SoftmaxLayer`:

```
#include <backendsCommon/test/layerTests/SliceTestImpl.hpp>
+#include <backendsCommon/test/layerTests/SoftmaxTestImpl.hpp>
#include <backendsCommon/test/layerTests/SpaceToBatchNdTestImpl.hpp>
```

5. Add the unit tests to `backends/<BackendName>/test/<BackendName>LayerTests.cpp`. `<BackendName>` is the backend that you are targeting for your layer.

5 Add serializer support

`armnnSerializer` is a library for serializing Arm NN neural networks, so that they require less memory.

To add a layer to the `armnnSerializer` library:

1. Add the layer or operator to the schema file at `armnn/src/armnnSerializer/ArmnnSchema.fbs`. The following code shows this addition for an example `SoftmaxLayer`:

```
//Softmax
table SoftmaxLayer {
    base:LayerBase;
    descriptor:SoftmaxDescriptor;
}
.
.
enum LayerType : uint {
    Addition = 0,
    Input = 1,
    ...
    Softmax = 6,
}
.
.
union Layer {
    ActivationLayer,
    AdditionLayer,
    ...
    SoftmaxLayer
}
```

2. Declare and implement a `Visit<LayerName>()` function in `armnn/src/armnnSerializer/Serializer.hpp` and `armnn/src/armnnSerializer/Serializer.cpp`. This function serializes the layer and any other information that describes the layer, like the layers descriptor. The Arm NN Serializer uses the `Visit<LayerName>()` function to serialize the layer when it traverses the network graph. The following code shows this declaration and implementation for an example `SoftmaxLayer` in the `Serializer.hpp` file:

```
void VisitSoftmaxLayer(const armnn::IConnectableLayer* layer,
                       const armnn::SoftmaxDescriptor& softmaxDescriptor,
                       const char* name = nullptr) override;
```

   The following code shows this declaration and implementation for an example `SoftmaxLayer` in the `Serializer.cpp` file:

```
// Build FlatBuffer for Softmax Layer
```

```
void SerializerVisitor::VisitSoftmaxLayer(const armnn::IConnectableLayer* layer,
                                          const armnn::SoftmaxDescriptor& softmaxDescriptor,
                                          const char* name)
{
    IgnoreUnused(name);

    // Create FlatBuffer BaseLayer
    auto flatBufferSoftmaxBaseLayer = CreateLayerBase(layer,
serializer::LayerType::LayerType_Softmax);

    // Create the FlatBuffer SoftmaxDescriptor
    auto flatBufferSoftmaxDesc =
        serializer::CreateSoftmaxDescriptor(m_flatBufferBuilder, softmaxDescriptor.m_Beta);

    // Create the FlatBuffer SoftmaxLayer
    auto flatBufferSoftmaxLayer =
        serializer::CreateSoftmaxLayer(m_flatBufferBuilder,
                                       flatBufferSoftmaxBaseLayer,
                                       flatBufferSoftmaxDesc);

    CreateAnyLayer(flatBufferSoftmaxLayer.o, serializer::Layer::Layer_SoftmaxLayer);
}
```

3. Add the newly supported layer to the serializer documentation in `src/armnnSerializer/SerializerSupport.md`. The following code shows this documentation for an example `SoftmaxLayer`:

```
The Arm NN SDK Serializer currently supports the following layers:
*Abs
...
*Softmax
.
```

6 Add deserializer support

`armnnDeserializer` is a library for loading neural networks, that Arm NN FlatBuffers files define, into the Arm NN runtime.

To add a layer to the `armnnDeserializer`:

1. Declare and implement a `Parse<LayerName>()` function in `src/armnnDeserializer/Deserializer.hpp` and `src/armnnDeserializer/Deserializer.cpp`. This function creates the layer and retrieves any necessary information about the layer, for example, descriptors. The following code shows the `Deserializer.hpp` for an example `SoftmaxLayer`:

```
// Softmax
    void ParseSoftmax(GraphPtr graph, unsigned int layerIndex);
```

   The following code shows the `Deserializer.cpp` for an example `SoftmaxLayer`:

```
// Softmax

void Deserializer::ParseSoftmax(GraphPtr graph, unsigned int layerIndex)
{
    CHECK_LAYERS(graph, 0, layerIndex);

    Deserializer::TensorRawPtrVector inputs = GetInputs(graph, layerIndex);
    CHECK_VALID_SIZE(inputs.size(), 1);

    Deserializer::TensorRawPtrVector outputs = GetOutputs(graph, layerIndex);
    CHECK_VALID_SIZE(outputs.size(), 1);

    armnn::SoftmaxDescriptor descriptor;
    descriptor.m_Beta = graph->layers()->Get(layerIndex)->layer_as_SoftmaxLayer()->descriptor()-
>beta();
    auto layerName = GetLayerName(graph, layerIndex);

    IConnectableLayer* layer = m_Network->AddSoftmaxLayer(descriptor, layerName.c_str());

    armnn::TensorInfo outputTensorInfo = ToTensorInfo(outputs[0]);
    layer->GetOutputSlot(0).SetTensorInfo(outputTensorInfo);

    RegisterInputSlots(graph, layerIndex, layer);
    RegisterOutputSlots(graph, layerIndex, layer);
}
```

2. Register the `Parse<LayerName>()` function in `m_Parser_Functions` to map the layer to the `enum` that is declared in `ArmnnSchema.fbs`. The following code shows this registration for an example `SoftmaxLayer`:

```
//Softmax
m_ParserFunctions(Layer_MAX+1, &Deserializer::ParseUnsupportedLayer)
{
    // register supported layers
    m_ParserFunctions[Layer_SoftmaxLayer]                = &Deserializer::ParseSoftmax;    .
    .
}
```

3. Add the layer to the switch case in `Deserializer::GetBaseLayer`. This switch case retrieves the layer using a layer index from the network graph. The following code shows the addition of an example `SoftmaxLayer` to a switch case:

```
//Softmax
Deserializer::LayerBaseRawPtr Deserializer::GetBaseLayer(const GraphPtr& graphPtr, unsigned int
layerIndex)
{
    auto layerType = graphPtr->layers()->Get(layerIndex)->layer_type();
    switch(layerType)
    {
       case Layer::Layer_SoftmaxLayer:
            return graphPtr->layers()->Get(layerIndex)->layer_as_SoftmaxLayer()->base();
       .
       .
    }
}
```

4. Add a unit test for the serializer in `src/armnnSerializer/test/SerializerTests.cpp`. A serializer unit test creates a simple InputLayer→NewLayer→OutputLayer network. The unit test serializes that network to a string, deserializes the network, and passes the network an implementation of the `LayerVerifierBase`, using the visitor pattern. To add the serializer unit test:

   a. Create an implementation of the test class `LayerVerifierBase` or `LayerVerifierBaseWithDescriptor`. Use the helper macro function `DECLARE_LAYER_VERIFIER_CLASS()` or `DECLARE_LAYER_VERIFIER_CLASS_WITH_DESCRIPTOR`. The following code shows this implementation for an example `SoftmaxLayer`:

```
BOOST_AUTO_TEST_CASE(SerializeSoftmax)
{
    DECLARE_LAYER_VERIFIER_CLASS_WITH_DESCRIPTOR(Softmax)


...
```

   b. Create a simple network that uses the layer being tested. The following code shows an example network that uses an example `SoftmaxLayer`:

```
const std::string layerName("softmax");
    const armnn::TensorInfo info({1, 10}, armnn::DataType::Float32);

    armnn::SoftmaxDescriptor descriptor;
    descriptor.m_Beta = 1.0f;

    armnn::INetworkPtr network = armnn::INetwork::Create();
    armnn::IConnectableLayer* const inputLayer   = network->AddInputLayer(0);
    armnn::IConnectableLayer* const softmaxLayer = network->AddSoftmaxLayer(descriptor,
layerName.c_str());
    armnn::IConnectableLayer* const outputLayer  = network->AddOutputLayer(0);

    inputLayer->GetOutputSlot(0).Connect(softmaxLayer->GetInputSlot(0));
    softmaxLayer->GetOutputSlot(0).Connect(outputLayer->GetInputSlot(0));

    inputLayer->GetOutputSlot(0).SetTensorInfo(info);
    softmaxLayer->GetOutputSlot(0).SetTensorInfo(info);
```

c. Serialize, then deserialize, the network. Then send the network the visitor test class, which requires deserialization support. The following example code serializes, then deserializes, the network and sends the network the visitor test class:

```
armnn::INetworkPtr deserializedNetwork = DeserializeNetwork(SerializeNetwork(*network));
    BOOST_CHECK(deserializedNetwork);

    SoftmaxLayerVerifier verifier(layerName, {info}, {info}, descriptor);
    deserializedNetwork->Accept(verifier);
```

5. Add a unit test for the new layer. To add the test, create a header file in `armnn/src/armnnDeserializer/test/Deserialize<Layer_Name>.hpp` and a source file in `armnn/src/armnnDeserializer/test/Deserialize<Layer_Name>.cpp`. In the unit test, write a JSON string that describes a small network that includes your layer. The `ParserFlatbuffersSerializeFixture` class deserializes and tests this JSON string. Ensure that the header of this class is included in the deserializer unit test. You can generate the JSON string using the serializer test. See Step 6 for more information on generating the JSON string using the serializer test. You must create a JSON unit test as this type of test is reliable.
6. Write the JSON test. Writing this JSON test can be complicated and time consuming. With some changes, you can use Arm NN to create this test using the serialization test. To do this:
   a. Change the FlatBuffers in `CMakeLists.txt` to build as a shared library and rebuild the FlatBuffers. The following code shows how to make this change:

```
option(FLATBUFFERS_BUILD_SHAREDLIB
       "Enable the build of the flatbuffers shared library"
       ON)
```

b. Change `libflatbuffers.a` to `libflatbuffers.so` in `armnn/GlobalConfig.cmake`. The following code shows the result of this change:

```
# Flatbuffers support for TF Lite and Armnn Serializer
if(BUILD_TF_LITE_PARSER OR BUILD_ARMNN_SERIALIZER)
    # verify we have a valid flatbuffers include path
    find_path(FLATBUFFERS_INCLUDE_PATH flatbuffers/flatbuffers.h
              HINTS ${FLATBUFFERS_ROOT}/include /usr/local/include /usr/include)

    message(STATUS "Flatbuffers headers are located at: ${FLATBUFFERS_INCLUDE_PATH}")

    find_library(FLATBUFFERS_LIBRARY
                 NAMES libflatbuffers.so flatbuffers
                 HINTS ${FLATBUFFERS_ROOT}/lib /usr/local/lib /usr/lib)

    message(STATUS "Flatbuffers library located at: ${FLATBUFFERS_LIBRARY}")
endif()
```

c. Add `idl.h` to the serializer include and add the following code to the `Serialize()` function:

```
#include <flatbuffers/idl.h>
...

void Serializer::Serialize(const INetwork& inNetwork)
{
    // Iterate through to network
    inNetwork.Accept(m_SerializerVisitor);
    flatbuffers::FlatBufferBuilder& fbBuilder = m_SerializerVisitor.GetFlatBufferBuilder();

    // Create FlatBuffer SerializedGraph
    auto serializedGraph = serializer::CreateSerializedGraph(
        fbBuilder,
        fbBuilder.CreateVector(m_SerializerVisitor.GetSerializedLayers()),
        fbBuilder.CreateVector(m_SerializerVisitor.GetInputIds()),
        fbBuilder.CreateVector(m_SerializerVisitor.GetOutputIds()),
        m_SerializerVisitor.GetVersionTable());

    // Serialize the graph
    fbBuilder.Finish(serializedGraph);

    // Code to be added, delete after use
    std::string schemafile;
    flatbuffers::LoadFile(("path/to/ArmnnSchema.fbs"),
```

```
                          false, &schemafile);
    std::string jsongen;
    flatbuffers::Parser parser;
    parser.Parse(schemafile.c_str());
    GenerateText(parser, fbBuilder.GetBufferPointer(), &jsongen);
    std::cout << jsongen;
}
```

d. Run the serialization test for the layer that you are creating. A working JSON output is printed. You can use this JSON file to make your deserializer test.

7. Add the unit test file to `ml/armnn/CMakeLists.txt`:

```
//Softmax
   if(BUILD_ARMNN_SERIALIZER AND ARMNNREF)
       enable_language(ASM)
       list(APPEND unittest_sources
            src/armnnSerializer/test/ActivationSerializationTests.cpp
            src/armnnSerializer/test/SerializerTests.cpp
            src/armnnDeserializer/test/Deserialize<Layer_Name>.cpp
    .
    .
```

8. Add the new supported layer to the deserializer documentation in `src/armnnDeserializer/`:

```
//Softmax
The Arm NN SDK Deserialize parser currently supports the following layers:
* Abs
* Activation
...
* Softmax
.
```

7 Add quantizer support

The `ArmnnQuantizer` program loads a 32-bit floating-point network into Arm NN and converts the network into a quantized asymmetric 8-bit network or a quantized symmetric 16-bit network. A quantized network can reduce the computational and memory cost of executing neural networks.

The Arm NN Quantizer quantizes Arm NN networks to the following data types:

- `QAsymmS8`
- `QsymmS16`
- `QAsymmU8`

To add a network layer to the quantizer:

1. Declare and implement the `Visit<LayerName>function()` in the `src/armnn/QuantizerVisitor.hpp` and `src/armnn/QuantizerVisitor.cpp` descriptors. The quantizer uses the `Visit<LayerName>function()` to visit a layer while traversing the network graph during the quantization process. The following code shows the `QuantizerVisitor.hpp` declaration for an example `SoftmaxLayer`:

```
// Softmax

    void VisitSoftmaxLayer(const IConnectableLayer* layer,
                           const SoftmaxDescriptor& softmaxDescriptor,
                           const char* name = nullptr) override;
```

   The following code shows the `QuantizerVisitor.cpp` for an example `SoftmaxLayer`:

```
//Softmax
void QuantizerVisitor::VisitSoftmaxLayer(const IConnectableLayer* layer, const char* name)
{
    IConnectableLayer* newLayer = m_QuantizedNetwork->AddSoftmaxLayer(name);
    RecordLayer(layer, newLayer);
    SetQuantizedInputConnections(layer, newLayer);
}
```

2. Add unit tests in `/src/armnn/test/QuantizerTest.cpp`. Each unit test creates a simple network with the new layer. The unit test quantizes that network and uses the visitor pattern to pass the network a test class to run.
   a. Create an implementation of the test class `TestSoftmaxQuantization`. If the default values are not correct, you must specify the `TestQuantizationParams.` The following example code creates an implementation of the test class `TestSoftmaxQuantization`:

```
// Softmax

BOOST_AUTO_TEST_CASE(QuantizeSoftmax)
{
```

```
    class TestSoftmaxQuantization : public TestQuantization
    {
    public:
        TestSoftmaxQuantization(const TensorShape& inputShape, const TensorShape&
outputShape)
        : TestQuantization(inputShape, outputShape) {}

        TestSoftmaxQuantization(const QuantizerOptions& options,
                                const TensorShape& inputShape,
                                const TensorShape& outputShape)
        : TestQuantization(options, inputShape, outputShape) {}

        void VisitSoftmaxLayer(const IConnectableLayer* layer,
                               const SoftmaxDescriptor& descriptor,
                               const char* name = nullptr) override
        {
            IgnoreUnused(descriptor, name);
            TensorInfo info = layer->GetOutputSlot(0).GetTensorInfo();

            // Based off default static range [0.0f, 1.0f]
            TestQuantizationParams(info, {1.0f / g_AsymmU8QuantizationBase, 0},
                                         {1.0f / g_AsymmS8QuantizationBase, -128},
                                         {1.0f / g_SymmS8QuantizationBase,  0},
                                         {1.0f / g_SymmS16QuantizationBase, 0});
        }
    };

...
```

b. The test network is `input →newLayer→output`. The following code shows how you create this network for the `SoftmaxLayer` using the `CreateNetworkWithSoftmaxLayer()` helper function:

```
// Softmax

    SoftmaxDescriptor descriptor;
    descriptor.m_Beta = 1.0f;

    const TensorShape shape{1U};
    INetworkPtr network = CreateNetworkWithSoftmaxLayer(descriptor, shape);

    INetworkPtr quantizedNetworkQAsymmU8 = INetworkQuantizer::Create(network.get())-
>ExportNetwork();
```

```
    TestSoftmaxQuantization validatorQAsymmU8(shape, shape);
    VisitLayersTopologically(quantizedNetworkQAsymmU8.get(), validatorQAsymmU8);
```

c. Quantize and test the network for `QSymmS8`, `QAsymmS8`, `QAsymmU8`, and `QAsymmS16`. The following code shows how to quantize and test the network:

```
// Softmax


    const QuantizerOptions qAsymmS8Options(DataType::QAsymmS8);
    INetworkPtr quantizedNetworkQAsymmS8 = INetworkQuantizer::Create(network.get(),
qAsymmS8Options)->ExportNetwork();
    TestSoftmaxQuantization validatorQAsymmS8(qAsymmS8Options, shape, shape);
    VisitLayersTopologically(quantizedNetworkQAsymmS8.get(), validatorQAsymmS8);


    // test QSymmS8 quantization
    const QuantizerOptions qSymmS8Options(DataType::QSymmS8);
    INetworkPtr quantizedNetworkQSymmS8 = INetworkQuantizer::Create(network.get(),
qSymmS8Options)->ExportNetwork();
    TestSoftmaxQuantization validatorQSymmS8(qSymmS8Options, shape, shape);
    VisitLayersTopologically(quantizedNetworkQSymmS8.get(), validatorQSymmS8);


    const QuantizerOptions qSymmS16options(DataType::QSymmS16);
    INetworkPtr quantizedNetworkQSymmS16 = INetworkQuantizer::Create(network.get(),
qSymmS16options)->ExportNetwork();
    TestSoftmaxQuantization validatorQSymmS16(qSymmS16options, shape, shape);
    VisitLayersTopologically(quantizedNetworkQSymmS16.get(), validatorQSymmS16);
```

8 Add EndToEnd tests

The `EndToEnd` tests are generic integration tests to check that the integration of the new operator into Arm NN has been executed correctly. Adding an `EndToEnd` test is optional, but we recommend that you add it. For more information on how to implement an `EndToEnd` test, refer to `armnn/src/backends/backendsCommon/test/` where these tests are located.

An `EndToEnd` test generally uses a simple network like `InputLayer`→<`NewLayer`>→`OutputLayer` to test that the new layer works correctly in a network graph.

9 Add parser support

You must add parser functionality and unit tests to enable support for your required parsers.

9.1 Add parser functionality

Add parser functionality to the corresponding parser. For example, add parser functionality for the TensorFlow, TensorFlow Lite, Caffe, or ONNX parsers.

Arm NN includes the following parser libraries in `armnn/src`:

- The `armnnCaffeParser` is a library for loading neural networks that are defined in Caffe protobuf files, into the Arm NN runtime.
- The `armnnTfParser` is a library for loading neural networks that TensorFlow protobuf files define, into the Arm NN runtime.
- The `armnnTfLiteParser` is a library for loading neural networks that TensorFlow Lite FlatBuffers, define, into the Arm NN runtime.
- The `armnnOnnxParser` is a library for loading neural networks that are defined in ONNX protobuf files, into the Arm NN runtime.

To add new layer support for those parsers:

1. Declare the `Parse<LayerName>()` function in `<X_Parser>.hpp`.
2. Implement the body of the `Parse<LayerName>()` in `<X_Parser>.cpp`.

9.2 Add parser unit tests

You can add parser unit tests to the new operator to test and verify that the parsing functionality of the new operator works. The parser unit test must contain a JSON representation of a simple network that contains an input layer and the expected output values. For example, `inputLayer→<newLayer>→outputLayer`.

The parser unit test infrastructure parses the network graph into an Arm NN format, then executes and compares the output values with the expected output values.

The parser unit tests are in `armnn/src/armm<X_Parser>/test`.

10 Add Android NN support

The Android NN driver is an implementation of the Android NN driver HAL which uses the Arm NN library to execute neural network interfaces.

Support for new operators is introduced in different Android versions. To indicate this, each Android version has a corresponding HAL policy:

- NeuralNetworks HAL 1.0 - The first HAL, present from Android OMR1
- NeuralNetworks HAL 1.1 - Present from Android P
- NeauralNetwork HAL 1.2 - Present from Android Q

The HalPolicy for each HAL version is in `/android-nn-driver/1.X/HalPolicy.hpp` where `X` represents the version.

Refer to the Android NeuralNetwork documentation to see which HAL policy supports a given operator. Related information includes a link to the Android NeuralNetwork documentation.

1. Declare the `Convert<LayerName>` function to each `HalPolicy.hpp` that supports the operator. The following code shows an example declaration for a `SoftmaxLayer` function to `android-nn-driver/1.2/HalPolicy.hpp`:

```
private:

static bool ConvertSoftmax(const Operation& operation, const Model& model, ConversionData&
data);
```

2. Define a `Convert<LayerName>()` function in the `HalPolicy.cpp` file for each HAL policy that supports the layer. Ensure that this function returns a `Convert<LayerName>()` function that is shared with all HAL Policies that support this operator. The following code shows this addition and function creation for an example `SoftmaxLayer` operator:

```
bool HalPolicy::ConvertOperation(const Operation& operation, const Model& model, ConversionData&
data)

{

    switch (operation.type)

    {

        ...

        case V1_2::OperationType::SOFTMAX:

            return ConvertSoftmax(operation, model, data);...

...

bool HalPolicy::ConvertSoftmax(const Operation& operation, const Model& model, ConversionData&
data)

{

    ALOGV("hal_1_2::HalPolicy::ConvertSoftmax()");

    return ::ConvertSoftmax<hal_1_2::HalPolicy>(operation, model, data);

}
```

3. Implement the `Convert<LayerName>()` function in the `ConversionUtils.hpp` file that corresponds to the earliest HAL policy that supports the operator. The available `ConversionUtils.hpp` files are:
   - `android-nn-driver/ConversionUtils.hpp` for HAL policy 1.0 and 1.1.
   - `android-nn-driver/ConversionUtils.hpp_1_2` for HAL policy 1.2,
   - `android-nn-driver/ConversionUtils_1_3.hpp` for HAL policy 1.3.

   The following code shows this implementation for an example `SoftmaxLayer` operator:

```
template<typename HalPolicy,
         typename HalOperation = typename HalPolicy::Operation,
         typename HalModel     = typename HalPolicy::Model>
bool ConvertSoftmax(const HalOperation& operation, const HalModel& model, ConversionData& data)
{
    using HalOperand     = typename HalPolicy::Operand;
    using HalOperandType = typename HalPolicy::OperandType;

    ALOGV("HalPolicy::ConvertSoftmax()");

    LayerInputHandle input = ConvertToLayerInputHandle<HalPolicy>(operation, 0, model, data);
    if (!input.IsValid())
    {
        return Fail("%s: Operation has invalid inputs", __func__);
    }

    const HalOperand* outputOperand = GetOutputOperand<HalPolicy>(operation, 0, model);
    if (!outputOperand)
    {
        return Fail("%s: Operation has no outputs", __func__);
    }

    const TensorInfo& outputInfo = GetTensorInfoForOperand(*outputOperand);

    SoftmaxDescriptor desc;
    HalOperandType outputType = outputOperand->type;

    // Read beta value
    if (outputType == HalOperandType::TENSOR_FLOAT16)
    {
        Half value;
```

```
        if (!GetInputScalar<HalPolicy>(operation, 1, HalOperandType::FLOAT16, value, model,
data))
        {
            return Fail("%s: Operation has invalid inputs %d", __func__, outputType);
        }

        desc.m_Beta = static_cast<float>(value);
    }
    else
    {
        if (!GetInputFloat32<HalPolicy>(operation, 1, desc.m_Beta, model, data))
        {
            return Fail("%s: Operation has invalid inputs %d", __func__, outputType);
        }
    }

    if (operation.inputs.size() > 2 && !GetInputScalar<HalPolicy>(operation,
                                                                  2,
                                                                  HalOperandType::INT32,
                                                                  desc.m_Axis,
                                                                  model,
                                                                  data))
    {
        return Fail("%s: Operation has invalid inputs", __func__);
    }

    bool isSupported = false;
    auto validateFunc = [&](const armnn::TensorInfo& outputInfo, bool& isSupported)
    {
        FORWARD_LAYER_SUPPORT_FUNC(__func__,
                                   IsSoftmaxSupported,
                                   data.m_Backends,
                                   isSupported,
                                   input.GetTensorInfo(),
                                   outputInfo,
                                   desc);
        };

    if(IsDynamicTensor(outputInfo))
    {
```

```
        isSupported = AreDynamicTensorsSupported();
    }
    else
    {
        validateFunc(outputInfo, isSupported);
    }

    if (!isSupported)
    {
        return false;
    }

    IConnectableLayer* layer = data.m_Network->AddSoftmaxLayer(desc);
    assert(layer != nullptr);
    input.Connect(layer->GetInputSlot(0));

    return SetupAndTrackLayerOutputSlot<HalPolicy>(operation, 0, *layer, model, data, nullptr,
validateFunc);
}
```

4. Run the Android tests for your operator. These tests are in `VTS/NeuralNetworks`. For example, run a VTS for the `SoftmaxLayer` operator. The following example code shows starting the driver, running a test and the results that the test prints to the console:

```
/vendor/bin/hw/android.hardware.neuralnetworks@1.2-service-armnn -v -c CpuRef &

/data/nativetest64/VtsHalNeuralnetworksV1_2TargetTest/VtsHalNeuralnetworksV1_2TargetTest --
hal_service_instance=android.hardware.neuralnetworks@1.2::IDevice/armnn --
gtest_filter="NeuralnetworksHidlTest.softmax_v1_2"

Note: Google Test filter = NeuralnetworksHidlTest.softmax_v1_2

[==========] Running 1 test from 1 test suite.

[----------] Global test environment set-up.

[----------] 1 test from NeuralnetworksHidlTest

[ RUN      ] NeuralnetworksHidlTest.softmax_v1_2

[       OK ] NeuralnetworksHidlTest.softmax_v1_2 (59 ms)

[----------] 1 test from NeuralnetworksHidlTest (61 ms total)

[----------] Global test environment tear-down

[==========] 1 test from 1 test suite ran. (61 ms total)

[  PASSED  ] 1 test.
```

11 Add an operator to the Arm NN source code

If you would like to share your Arm NN operator with the rest of the Arm NN community, you must add the operator to the Arm NN source code.

If you would like to have your new operator added to the Arm NN source code, you must submit your changes for code review. All code reviews are performed on the Linaro ML Platform. For this platform, you require GitHub account credentials to create an account. You must then configure your username and user email to be able to have your patch signed off when you submit it.

Enter the following commands to configure your name and email:

```
cd armnn // Top level of the Arm NN repository
git config user.name "FIRST_NAME SECOND_NAME"
git config user.email your@email.address
```

Enter the following commands to commit using sign off and to push your patch for code review:

```
git commit -s
git push origin HEAD:refs/for/master
```

Your patch is now on the ML Platform Gerrit. Your patch requires a `+1` verified from CI testing and verification, and a `+2` from a reviewer, before it can be merged. Core contributors from the Arm NN team can give `+2` or `-2` reviews and submit code. All other contributors can give `+1`, `0`, `-1` code reviews.

After your patch is reviewed, verified, and submitted, the patch gets merged to the Arm NN Master branch and it is available for anyone to use.

12 Related information

Here are some resources related to material in this guide:

- **Android NeuralNetworks documentation**
- **Arm Community**
- **Arm NN**
- **Build Arm NN custom backend plugins**: guide for information on building custom backend plugins for Arm NN
- **Gerrit Review UI documentation**: includes information on using Gerrit.
- **Linaro ML Platform**
- **Linaro ML Plaform Contributor Guide**: contains details of copyright notice and developer certificate of origin sign off

13 Next steps

The guide has covered the steps to add an example Softmax operator to Arm NN. Now that you have completed adding a Softmax operator to Arm NN, you are ready to add any operator to Arm NN and optionally add your operator to the Arm NN source code.

You can read the **Build Arm NN custom backend plugins** how-to guide to learn more about writing a custom backend for Arm NN.