TVM Quantization Experiments

With ResNet50 Machine Learning Model

Pavle Sarenac, March 2024

- Question 1: *Get an instance of a ResNet50 model implemented in PyTorch. It's available in the torchvision package.*
- Answer 1:

```
##########################################################################################################################
# Getting an instance of a pretrained PyTorch ResNet50 model in inference mode
# ------------------------------------------------------------------------------------------------------------------------
model_name = "resnet50"
model = getattr(torchvision.models, model_name)(pretrained=True)
model = model.eval()
##########################################################################################################################
##########################################################################################################################
# Getting a TorchScripted representation of the ResNet50 model so that it can later be converted to a Relay graph
# ------------------------------------------------------------------------------------------------------------------------
# Obviously, we will be getting this TorchScripted model also in inference mode
input_shape = [1, 3, 224, 224]  # Input for our ResNet50 model will be a 4D tensor with NCHW data layout
input_data = torch.randn(input_shape)  # Randomized dummy input for the purpose of getting a TorchScripted model
torch_scripted_model = torch.jit.trace(model, input_data).eval()
##########################################################################################################################
```

- Question 2: *It's a good idea to try TVM on an un-quantized DNN first. Give TVM the network and a sample input to the network, and compile the network into a function object that can be called from Python side to produce DNN outputs. The TVM how-to guides has complete tutorials on how to do this step. Pay attention to the compilation target: which hardware (CPU? GPU?) the model is being compiled for, and understand how to specify it. Compile for GPU, if you have one, or CPU otherwise.*
- Answer 2: *tvm_runtime_module.run() can be called from Python side to produce DNN outputs. Hardware device (CPU) is specified when creating tvm_runtime_module based on tvm_compiled_module.*

```
######################################################################################################################
# Executing our model on a desired hardware device using TVM runtime
# ---------------------------------------------------------------------------------------------------------------------
hardware_device = tvm.cpu(0)
input_data_type = "float32"
tvm_runtime_module = graph_executor.GraphModule(tvm_compiled_module["default"](hardware_device))

tvm_runtime_module.set_input(input_name, tvm.nd.array(cat_image.astype(input_data_type)))
tvm_runtime_module.run()

tvm_inference_result = tvm_runtime_module.get_output(0)
######################################################################################################################
```

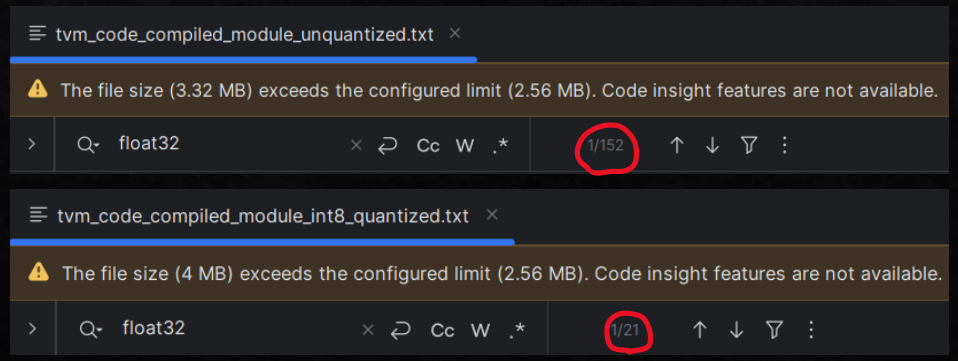
- Question 3: *Now, quantize the model down to int8 precision. TVM itself has utilities to quantize a DNN before compilation; you can find how-tos in the guides and forum. Again, you should get a function object that can be called from Python side. Hint: there is a namespace tvm.relay.quantize and everything you need is somewhere in there.*
- Answer 3: *quantized_relay_ir is later used for the TVM compilation of the model.*

```
###############################################################################################################
# Quantizing our model down to int8 precision
# --------------------------------------------------------------------------------------------------------------
# For (nbit_activation, dtype_activation) = (8, "int8"), a significant model precision drop is noticed by the
# different results that TVM and PyTorch models return
# For (nbit_activation, dtype_activation) = (16, "int16") TVM and PyTorch models return equal result
# For (nbit_activation, dtype_activation) = (32, "int32") TVM and PyTorch models return equal result
with relay.quantize.qconfig(
    nbit_input=8,
    nbit_weight=8,
    nbit_activation=16,
    dtype_input="int8",
    dtype_weight="int8",
    dtype_activation="int16"
):
    quantized_relay_ir = relay.quantize.quantize(relay_ir, inference_parameters)
###############################################################################################################
```

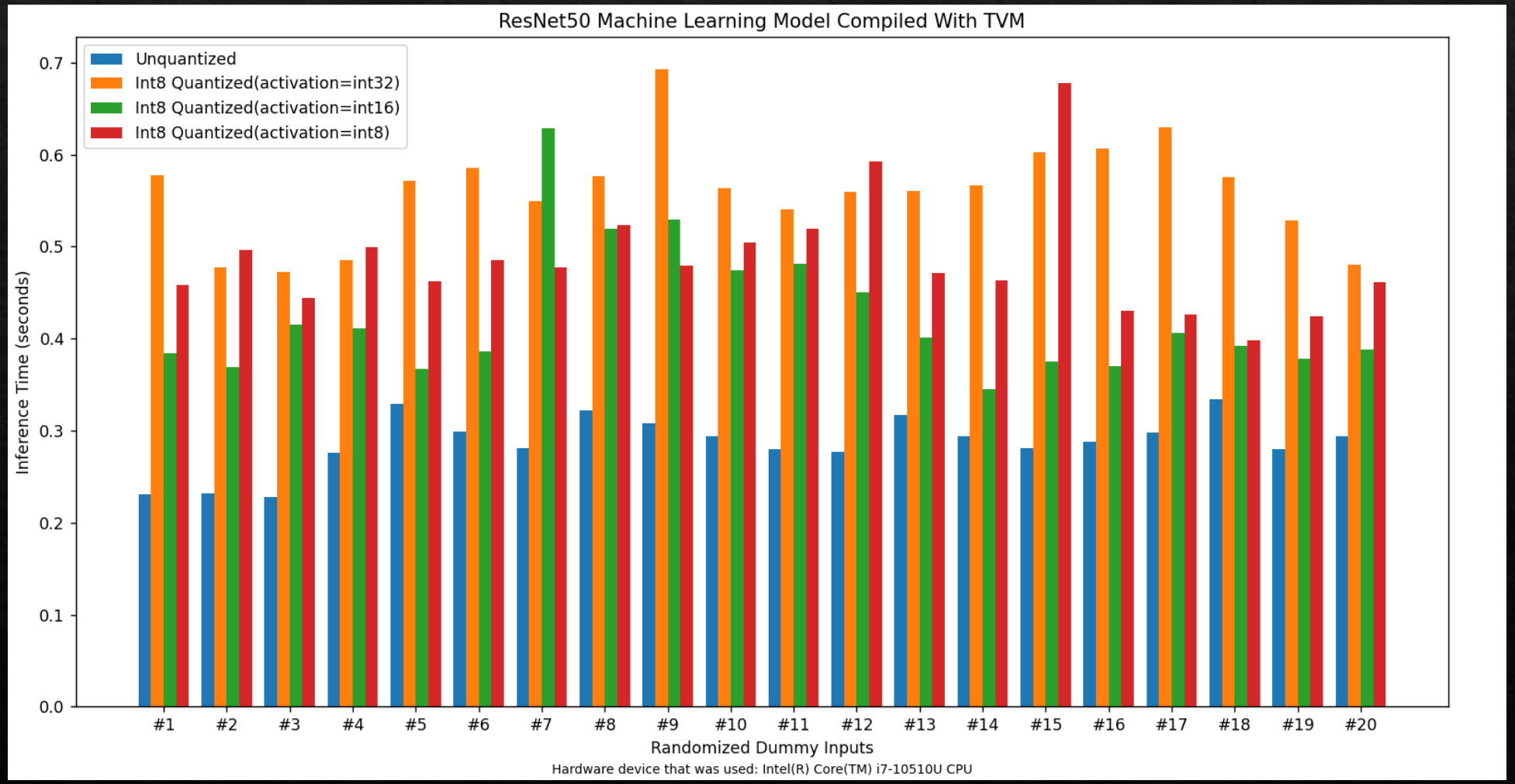
- Question 4: *Just for your own check - how can you see the TVM code in the compiled module? Did the quantization actually happen, for example, did the data types in the code change?*
- Answer 4: *By comparing LLVM IR codes generated when compiling unquantized and quantized models, it is obvious that the number of "float32" data types has drastically decreased after quantization, so that is the proof that the model was quantized successfully.*

```
################################################################################################################
# Saving the compiled tvm module code to a file for comparison with quantized models to see if data types in the
# code actually changed after quantization
# ----------------------------------------------------------------------------------------------------------------
with open("tvm_code_compiled_module_unquantized.txt", "w") as file:
    file.write(tvm_compiled_module.get_lib().get_source())
################################################################################################################
```

```
################################################################################################################
# Saving the compiled tvm module code to a file for comparison with the unquantized model to see if data types in
# the code actually changed after quantization
# ----------------------------------------------------------------------------------------------------------------
with open("tvm_code_compiled_module_int8_quantized.txt", "w") as file:
    file.write(tvm_compiled_module.get_lib().get_source())
################################################################################################################
```



- Question 5.1: *Use TVM's utility functions to benchmark the inference time of the quantized model vs. the un-quantized model.*
- Answer 5.1:



- Question 5.2: *In this task we will not try to maximize the performance of the quantized DNN, but if there is no speedup, you should try to understand it and formulate a guess. Hint: TVM may print the following when you compile the DNN: "One or more operators have not been tuned. Please tune your model for better performance. Use DEBUG logging level to see more details". What does it mean?*
- Answer 5.2:

- Question 6.1: *In your quantization setup, how did TVM know that you wanted to quantize to int8? Look into that, and vary the number of bits of quantization (the n in int-n). Searching in forum and peeking the source code of the quantizer class will both help.*
- Answer 6.1: *TVM knew that I wanted to quantize to int8 because I explicitly set that using tvm.relay.quantize.qconfig as I have shown previously in my answer to question number 3. However, even if I hadn't explicitly set that, the TVM would have done the quantization to int8 because that is the default quantization configuration as can be seen in the QConfig class found in the tvm/python/tvm/relay/quantize/quantize.py file.*

```
class QConfig(Object):
    """Configure the quantization behavior by setting config variables.

    Note
    ----
    This object is backed by node system in C++, with arguments that can be
    exchanged between python and C++.

    Do not construct directly, use qconfig instead.

    The fields that are backed by the C++ node are immutable once an instance
    is constructed. See _node_defaults for the fields.
    """

    _node_defaults = {
        "nbit_input": 8,
        "nbit_weight": 8,
        "nbit_activation": 32,
        "dtype_input": "int8",
        "dtype_weight": "int8",
        "dtype_activation": "int32",
```

- Question 6.2: *Try out int8 -> int4 -> int2 -> int1; note which precisions work. When it doesn't work, note exactly which part is failing.*
- Answer 6.2: *int8 quantization works fine, but the program crashes if int4, int2 or int1 quantization are attempted. TVM prints an error that says that those data types are unknown. I believe that is the case because TVM has no direct support for those data types. As proof for that, I have found a C++ macro in the tvm/src/relay/transforms/pattern_utils.h file that obviously contains all the data types that TVM supports, and int4, int2, and int1 data types are not among them.*

```
/*!
 * \brief Dispatch DataType to the C++ data type
 *  during runtime.
 */
#define TVM_DTYPE_DISPATCH(type, DType, ...)                                \
  if (type == DataType::Float(64)) {                                        \
    typedef double DType;                                                   \
    { __VA_ARGS__ }                                                         \
  } else if (type == DataType::Float(32)) {                                 \
    typedef float DType;                                                    \
    { __VA_ARGS__ }                                                         \
  } else if (type == DataType::Float(16)) {                                 \
    typedef uint16_t DType;                                                 \
    { __VA_ARGS__ }                                                         \
  } else if (type == DataType::BFloat(16)) {                                \
    typedef uint16_t DType;                                                 \
    { __VA_ARGS__ }                                                         \
  } else if (type == DataType::Int(64)) {                                   \
    typedef int64_t DType;                                                  \
    { __VA_ARGS__ }                                                         \
  } else if (type == DataType::Int(32)) {                                   \
    typedef int32_t DType;                                                  \
    { __VA_ARGS__ }                                                         \
  } else if (type == DataType::Int(16)) {                                   \
    typedef int16_t DType;                                                  \
    { __VA_ARGS__ }                                                         \
  } else if (type == DataType::Int(8)) {                                    \
    typedef int8_t DType;                                                   \
    { __VA_ARGS__ }                                                         \
  } else if (type == DataType::UInt(64)) {                                  \
```