This directory contains quantized Caffe2 operators optimized for Intel and AMD x86 processors, using FBGEMM backend for matrix multiplications and convolutions. Caffe2's quantized resnet50 (https://github.com/caffe2/models/tree/master/resnet50_quantized) uses the operators implemented here. We call these DNNLOWP (deep neural network low-precision, an homage to Google's gemmlowp library that our basic quantization method is based on) operators. We need to explicitly set the engine to DNNLOWP to use these operators. Otherwise, Int8* operators will use the implementation in pytorch/caffe2/operators/quantized that use QNNPACK backend whose primary optimization target is ARM mobile processors. Please use Int8* op types because op types without Int8 prefix and DNNLOWP engine is deprecated.

Quantization method

The basic quantization method is similar to that used by gemmlowp (https://github.com/google/gemmlowp/) and TensorFlow Lite. That is we use a linear quantization method where quantization and dequantization is a simple affine transformation (plus rounding and saturation in quantization). Therefore, quantization bins are uniform. Similar to gemmlowp, our quantized operators use asymmetric quantization by default but there's an option to use symmetric quantization (this can be controlled globally using caffe2_dnnlowp_preserve_activation_sparsity and caffe2_dnnlowp_preserve_weight_sparsity gflags options or per-operator basis using preserve_activation_sparsity and preserve_weight_sparsity arguments). Unsigned 8-bit integers are used for activations and signed 8-bit integers are used for weights (this design choice is mostly because int8 SIMD instructions in x86 has one input operand unsigned and the other input operand signed). We also support per-output-channel quantization similarly to gemmlowp (Int8FC with DNNLOWP_ROWWISE engine). Note that only the weights can have multiple pairs of scale and zero_offset (per output channel) and activations can still have only one pair of scale and zero_offset. This is because if an activation has per-channel quantization, inner-products in a GEMM that multiplies the activation would require summing up numbers with different scales, which is significant overhead. We also support group-wise quantization (quantize_groupwise argument of Int8Conv operator).

To compute the quantization parameters of activation tensors, we need to know their value distributions (except when we use dynamic quantization which is explained below). We have histogram_observer that can be attached to Caffe2 nets and collect the distribution. By default, the quantization parameters are selected based on the min and max, but we highly recommend to use the quantization method that minimizes the L2 norm of quantization errors or its much faster approximated version (see norm_minimization.cc). The L2 minimizing quantization can be selected globally by setting gflags caffe2_dnnlowp_activation_quantization_kind and caffe2_dnnlowp_weight_quantization_kind to L2 or L2_APPROX, or per-

operator basis using activation_quantization_kind and weight_quantization_kind arguments).

Differences from gemmlowp and TensorFlow Lite

- Floating-point requantization

Unlike gemmlowp using fixed-point operations that emulates floating point operations of requantization, fbgemm just uses single-precison floating-point operations. This is because in x86 just using single-precision floating-point operations is faster. Probably, gemmlowp used pure fixed-point operations for low-end mobile processors. QNNPACK also has similar constraints as gemmlowp and provides multiple options of requantization implementations. The users could modify the code to use a different requantization implementation to be bit-wise idential to the HW they want to emulate for example. If there're enough requests, we could consider implementing a few popular fixed-point requantization as QNNPACK did.

- 16-bit accumulation with outlier-aware quantization

In current Intel processors, int8 multiplication with int32 accumulation doesn't provide very high speedup: 3 instructions vpmaddubsw + vpmaddwd + vpadd are needed. With 16-bit accumulation, we can use 2 instructions instead with up to 2x instruction throughput per cycle. However, 16-bit accumulation can lead to frequent saturation and hence a big accuracy drop. We minimize the saturation by splitting the weight matrix into two parts, W = W_main + W_outlier, where W_main contains values with small magnitude and W_outlier contains the residual. The matrix multiplication, X x W^T is calculated in two stages, where X x W_main^T uses 16-bit accumulation, and X x W_outlier^T uses 32-bit accumulation. W_outlier is typically sparse hence X x W_outlier^T accounts for a small fraction of the total time. This implementation can be used by setting the Caffe2 operator engine to DNNLOWP_ACC16. Conv, ConvRelu, and FC support DNNLOWP_ACC16. The threshold for outlier can be controlled by nbits_in_non_outlier argument of the operator. For example, when nbits_in_non_outlier=7, a value is outlier if it needs more than 7-bit (e.g. the value is <= -65 or >= 64).

- Dynamic quantization

DNNLOWP operators support dynamic quantization that selects quantization parameter per mini-batch. This can be useful for example in neural machine translation models that spends most of its time in FC (while it's challenging to do end-to-end quantization due to the diverse set of operator types) and its batch size is small so dynamic quantization does not add much additional overhead. The advantage of dynamic quantization is two folds: no need for collecting the value distribution of activation tensors and possibly higher accuracy. This option can be used by setting dequantize_output operator argument.

Quantization operators

The following quantized operators are currently implemented * Int8BatchMatMul * Int8BatchPermutation * Int8ChannelShuffle * Int8Concat * Int8Conv and Int8ConvRelu : additionally available engine DNNLOWP_ACC16 (16-bit accumulation) * Int8Dequantize * Int8Add * Int8ElementwiseLinear * Int8Mul * Int8Sum and Int8SumRelu * Int8FC : additionally available engine DNNLOWP_ACC16 (16-bit accumulation) and DNNLOWP_ROWWISE * Int8GroupNorm * Int8LSTMUnit * Int8AveragePool * Int8MaxPool * Int8Relu * Int8ResizeNearest * Int8SpatialBN * Int8Tanh : uses lookup table * Int8Sigmoid : reuses the same lookup table for Tanh * Int8Gather

Differences from mobile quantization operators

The aim is Int8* operators in caffe2/operators/quantized (primarily optimized for mobile processors) compatible with the operators in this directory, but there're a few minor differences we will soon to fix

- The implementation of Int8AveragePool in this directory can have different quantization parameters for its input and output.
- Int8Sum in caffe2/operators/quantized assumes 2 input tensors while the implementation in this directory can work with arbitrary number of input tensors.

Extra functionality

- Propagation of quantization parameters

The quantized operators in this directory can have followed_by argument like "Relu", "Sigmoid", and "Tanh". Setting this can improve the quantization accuracy for example by narrowing down the quantization parameter of an output tensor for example saturating negative values to 0 if it's followed by Relu. In fact, it's currently mandatory to set followed_by to Sigmoid and Tanh . This limitation will be fixed.

- Measure quantization error

To facilitate numerical debugging, setting measure_quantization_error argument will run a shadow copy of single-precision floating-point operator and reports the L2 error compared to quantized outputs. This can help identifying which operator introduces the biggest error to narrow down the numerical issues.

- Different precision

Our operators also have some functionality to emulate different fixed-point HW implementations. For example, setting activation_quantization_precision and weight_quantization_precision operator attributes to 4 can emulate 4-bit HW. We can also emulate more than 8-bit. Using that requires a different engine

DNNLOWP_16 (sorry about a poor choice of name because it's confusing with DNNLOWP_ACC16).

- Pre-packing weights

The FBGEMM needs the weight tensor pre-packed in its custom layout for high performance (note that this is not the case for activation and can stay in the standard NHWC layout because having custom activation layout will be complicated due to inter-operability with producer/consumer operators). We need other pre-processings like computing column offsets when input activations use asymmetric quantization, or separating out outliers as a sparse matrix for 16-bit accumulation. Having the results of these kind of pre-processing as a part of Conv/FC operator will duplicate the results when a weight tensor is shared among multiple operators (can happen if we run multiple copies of the same net sharing weights). fbgemm_pack_matrix_cache.h provides a quick workaround of caching the pre-processing results but we strongly recommend using pre-packing operators as shown in the following example.

init_net:

```
op {
  input: "Wq"
  input: "bq"
  output: "Wq_packed"
  name: "Pack_Wq"
  type: "Int8ConvPackWeight"
  engine: "DNNLOWP"
}
...
```

predict_net:

```
...
op {
  input: "Xq"
  input: "Wq_packed"
  output: "Yq"
  name: "Conv_example"
  type: "Int8Conv"
  arg {
    name: "kernel"
    i: 2
  }
  arg {
    name: "order"
    s: "NHWC"
  }
  arg {
    name: "Y_scale"
```

```
    f: 1.0
  }
  arg {
    name: "Y_zero_point"
    i: 0
  }
  engine: "DNNLOWP"
}
```