8-bit Integer Quantization

2020-20284 장원영

1. float min, float max로부터 quantization parameter (scale, zero point)가 어떻게 계산되는가?

GetAsymmetricQuantizationParams 함수는 비대칭적인 값을 가지는 activation을 위한 quantization parameter를 계산하며, GetSymmetricQuantizationParams 함수는 대칭적인 값을 가지는 weight를 위한 quantization parameter를 계산한다.

먼저 GetAsymmetricQuantizationParams의 경우에, scale은 float인 max와 min의 차를 quant_max_float과 quant_min_float의 차로 나눈 값이다. 이때 quant_max_float과 quant_min_float은 초기에 integer로 할당한 quant_max와 quant_min 값을 stactic_cast 함수를 이용해 float 타입으로 변환한 값이다.

```
void GetAsymmetricQuantizationParams(
    float min, float max, const int quant_min, const int quant_max,
    QuantizationParametersT* quantization_params) {
  const float quant_min_float = static_cast<float>(quant_min);
  const float quant_max_float = static_cast<float>(quant_max);
  // Adjust the boundaries to guarantee 0 is included.
  min = std::min(static_cast<float>(min), 0.0f);
  max = std::max(static_cast<float>(max), 0.0f);
  const float scale = (max - min) / (quant_max_float - quant_min_float);
  // Scale can be zero if min and max are exactly 0.0f.
```

activation의 zero point는 integer 범위 내 어느 곳에서나 zero point를 가질 수 있으며, 경우에 따라 다르게 계산된다. 해당 값은 zero_point_from_min에 따라 달라지는데, 이 값은 앞서 계산한 scale에 따라 달라질 수 있다. scale이 0이라면 quant_min_float 값을 갖고, 그렇지 않다면 quant_min_float - min / scale로 계산된다.

zero_point_from_min이 quant_min_float보다 작다면 zero point는 quant_min 값이 된다. 반면 quant_max_float보다도 크다면 zero point는 quant_max 값이 된다. 두 조건을 모두 만족하지 않을 경우에는 zero_point_from_min을 반올림한 integer 값을 갖는다.

```
  float zero_point_from_min = quant_min_float;
  if (scale != 0) {
    zero_point_from_min = quant_min_float - min / scale;
  }
  int64_t zero_point;
  if (zero_point_from_min < quant_min_float) {
    zero_point = static_cast<int64_t>(quant_min);
  } else if (zero_point_from_min > quant_max_float) {
    zero_point = static_cast<int64_t>(quant_max);
  } else {
    zero_point = static_cast<int64_t>(std::round(zero_point_from_min));
  }
```

GetSymmetricQuantizationParams의 scale은 float 값인 max와 min의 절댓값 중 큰 값을 half_quant_range로 나눈 값이다. zero point는 0으로 강제해서 zero point에 activation 값을 곱하는 비용을 없애게 된다.

```
void GetSymmetricQuantizationParams(
    float min, float max, const int half_quant_range,
    QuantizationParametersT* quantization_params) {
  // Adjust the boundaries to guarantee 0 is included.
  min = std::min(min, 0.0f);
  max = std::max(max, 0.0f);
  const float scale = std::max(std::abs(max), std::abs(min)) / half_quant_range;
  quantization_params->min = std::vector<float>(1, min);
  quantization_params->max = std::vector<float>(1, max);
  quantization_params->scale = std::vector<float>(1, scale);
  quantization_params->zero_point = std::vector<int64_t>(1, 0);
}
```

2. 3가지 layer에 대해 quantization parameter를 이용해서 어떤 식으로 integer-only arithmetic을 수행하는가?

- ReLU, logistic

```
template <typename T>
void QuantizedReluX(float act_min, float act_max, const TfLiteTensor* input,
                    TfLiteTensor* output, const ReluOpData* data) {
  ReluParams params;
  params.quantized_activation_min =
      std::max(static_cast<int32_t>(std::numeric_limits<T>::min()),
               output->params.zero_point +
                   static_cast<int32>(roundf(act_min / output->params.scale)));
  params.quantized_activation_max =
      act_max == std::numeric_limits<float>::infinity()
          ? static_cast<int32_t>(std::numeric_limits<T>::max())
          : std::min(
                static_cast<int32_t>(std::numeric_limits<T>::max()),
                output->params.zero_point +
                    static_cast<int32>(roundf(act_max / output->params.scale)));
  params.input_offset = input->params.zero_point;
  params.output_offset = output->params.zero_point;
  params.output_multiplier = data->output_multiplier;
  params.output_shift = data->output_shift;
  optimized_ops::ReluX(params, GetTensorShape(input), GetTensorData<T>(input),
                       GetTensorShape(output), GetTensorData<T>(output));
}
```

ReLU는 activations.cc 문서 line 763의 TfLiteStatus ReluEval 함수에서 확인할 수 있다. 8-bit integer로 quantization 하는 경우, 같은 문서의 line 176에서 정의한 QuantizedReluX 함수를 이용한다. quantized_activation_min은 1) numeric type T로 표현할 수 있는 가장 작은 finite 값과 2) zero point에 act_min / scale을 더한 값 중 더 큰 값으로 결정된다.

quantized_activation_max 역시 비슷하게 결정되는데, act_max가 positive infinity일 경우 numeric type T로 표현할 수 있는 가장 큰 finite 값으로 결정된다. 반면 act_max가 positive infinity가 아닐

경우 1) numeric type T로 표현할 수 있는 가장 큰 finite 값과 2) zero_point에 act_max / scale을 더한 값 중 더 작은 값으로 결정된다.

위와 같이 계산된 값들은 reference_ops.h 문서에 정의된 ReluX 함수에서 사용된다. 최종적으로는 TfLiteStatus ReluEval 함수에서 uint8이나 int8로 변환되어 integer-only arithmetic을 수행한다.

```
// Sigmoid is also know as "Logistic".
template <KernelType kernel_type>
TfLiteStatus SigmoidEval(TfLiteContext* context, TfLiteNode* node) {
  OpData* data = reinterpret_cast<OpData*>(node->user_data);

  const TfLiteTensor* input;
  TF_LITE_ENSURE_OK(context, GetInputSafe(context, node, 0, &input));
  TfLiteTensor* output;
  TF_LITE_ENSURE_OK(context, GetOutputSafe(context, node, 0, &output));
  switch (input->type) {
```

logistic은 sigmoid로도 알려져 있다. activations.cc 문서 line 995의 TfLiteStatus SigmoidEval 함수에서 확인할 수 있다. 8-bit integer로 quantization 하는 경우, kernel_type이 kFixedPointOptimized와 같다면 optimized_ops.h 문서에 정의된 Logistic16bitPrecision 함수를 이용한다. 해당 함수에서는 input uint8이나 int8 값을 읽고, 여기에 input_zero_point를 빼는 등의 integer-only arithmetic을 수행한다.

```
case kTfLiteUInt8: {
  if (kernel_type == kFixedPointOptimized) {
    LogisticParams params;
    params.input_zero_point = input->params.zero_point;
    params.input_range_radius = data->input_range_radius;
    params.input_multiplier = data->input_multiplier;
    params.input_left_shift = data->input_left_shift;
    optimized_ops::Logistic16bitPrecision(
        params, GetTensorShape(input), GetTensorData<uint8_t>(input),
        GetTensorShape(output), GetTensorData<uint8_t>(output));
  } else {
    EvalUsingLookupTable(data, input, output);
  }
  break;
}
```

- Fully connected

Fully connected는 fully_connected.cc 문서 line 962의 TfLiteStatus Eval 함수에서 확인할 수 있다. 주로 같은 문서의 line 738에서 정의한 EvalQuantized 함수를 이용하며, 8-bit integer로 quantization 하는 경우, fully_connected.h 문서에 정의된 FullyConnected 함수를 이용한다.

FullyConnected 함수에서는 input data와 filter data를 곱해준 다음 bias data까지 더해주는 연산을 진행하며, 최종적으로 output_activation_min과 output_activation_max 값들과 비교를 통해 output data로 내보내는 integer-only arithmetic을 수행한다.

```
const int accum_depth = filter_shape.Dims(filter_dim_count - 1);
for (int b = 0; b < batches; ++b) {
  for (int out_c = 0; out_c < output_depth; ++out_c) {
    int32_t acc = 0;
    for (int d = 0; d < accum_depth; ++d) {
      int32_t input_val = input_data[b * accum_depth + d];
      int32_t filter_val = filter_data[out_c * accum_depth + d];
      acc += (filter_val + filter_offset) * (input_val + input_offset);
    }
    if (bias_data) {
      acc += bias_data[out_c];
    }
    acc = MultiplyByQuantizedMultiplier(acc, output_multiplier, output_shift);
    acc += output_offset;
    acc = std::max(acc, output_activation_min);
    acc = std::min(acc, output_activation_max);
    output_data[out_c + output_depth * b] = static_cast<int8_t>(acc);
  }
}
```