Quantizing Neural Network Models in MXNet for Strict Consistency on Blockchain

Introduction

There are emerging interests in deploying deep learning models on various platforms and devices. Deep neural networks are not just being used in supercomputers or GPU clouds but are also experiencing increasingly widespread adoption in edge devices that have low computing power with memory and energy consumption constraints. In the emerging field of blockchain, where it demands every step of computation to be strictly deterministic and have minimal resource consumption, the situation is even harder for a DNN model to deploy.

As the technology of distributed ledger develop, needs for more complex computations emerge. The nondeterministic nature of floating-point arithmetic and parallel computation, e.g., summation over a series of floating-point number, should not be a problem for DNN models in a cloud computing environment, since the purpose of DNN model inference is often for better precision, not for future auditing or bit-level comparison. However, the nondeterministic result is undesirable in the occasion of blockchain, where different nodes need to verify the transactions to reach consensus before finalizing on the blockchain. Each node has its own hardware specification running different versions of operating systems, making it harder to standarize the computation. In other words, each single run of a single model, with the same input, in heterogeneous computing systems must yield bit-level identical results.

In this post, we propose a framework to accelerate DNN models' inference and eliminate nondeterministic behavior in model inference for blockchain systems. Before we dive into implementation details, let's go through a few key observations and intuitions that leads to the discovery of our framework first.

Researchers have proposed several approaches to run DNN models in limited-resource environments:

1. **Fake Quantization**: quantizing floating-point numbers into 8-bit integers and transfering data to the accelerator, which takes linear time. The most costly part of the calculation, e.g., conv, only happens in the accelerator which is dedicated in the 8-bit arithmetic. Afterwards, results are transformed back to floating-point numbers.
2. **Integer-Only Inference**: quantization scheme that allows inference to be carried out using integer-only arithmetic, which can be implemented more efficiently than floating-point inference on commonly available integer-only hardware. Fine-tune procedure is usually utilized to preserve model accuracy.

We apply integer-only inference in our approach. The current implementation in MXNet's Contrib library follows the fake quantization routine and redirects the computation to MKLDNN math library in runtime. However, in blockchain's deterministic scenario, floating-point numbers will introduce undesired behaviors. Integer-only inference, on the other hand, fits blockchain's

heterogeneous environments perfectly. Moreover, we put a strict numerical bound to avoid integer overflow by rewriting computation graph.

Implementation

Cortex is a ethereum-based blockchain platform where we practice our approach. The framework includes two major components: MRT for quantization and CVM for inference. First, We implement a converter using MXNet's NNVM module **Model Representation Tool** (MRT) to convert MXNet Model Zoo to quantized models. Second, we run the quantized models in the Cortex blockchain's virtual machine **Cortex Virtual Machine** (CVM), the runtime environment for smart contracts equipped with machine learning models.

Fusion and Operator Rewriting

We illustrate our approach by following examples.

MAC Decomposition

Suppose we are calculating the inner dot of two vector $x \in Z^n_{\text{int8}}$ and $y \in Z^n_{\text{int8}}$, which may results in a 32-bit integer, sepecifically, $s =< x, y >= \sum_i^n x_i y_i \in Z^n_{\text{int32}}$. However, this condition of numerical bound holds only when n is less than 2^{16}. In other words, we cannot assume that there is no overflow when n is large, which may introduce nondeterministic behavior during parallel computing. To resolve this problem, we decomposite the computation into smaller sub-problems in graph level and aggregate the results. Mathematically,
$s =< x^{(1)}, y^{(1)} > + < x^{(2)}, y^{(2)} > + \ldots + < x^{(K)}, y^{(K)} >$, $x^{(k)}$ is the k-th part of vector x with each partition's length smaller than 2^{16}.

Matrix multiplication operator `matmul` can also be rewritten in the same fashion, generating a series of `elemwise_add` operators that sum over several intermediate matrices. The semantics is unchanged although this step introduces additional operators in the computation graph.

Fusing BatchNorm

$$
\begin{aligned}
y_{\cdot i \cdot \cdot} &= \text{BatchNorm}(x_{\cdot i \cdot \cdot}) \\
&= \frac{x_{\cdot i \cdot \cdot} - \mu_i^X}{\sigma_i^X} * \gamma_i + \lambda_i \\
&= x_{\cdot i \cdot \cdot} * \alpha_i + \beta_i
\end{aligned}
$$

where α_i is $\frac{\gamma_i}{\sigma_i^X}$ and β_i is $\lambda_i - \mu_i * \gamma_i / \sigma_i^X$.

We can see that BatchNorm can be fused into Convolution's weight. As a result, we have:

$$
\begin{aligned}
z &= \text{BatchNorm}(\text{Convolution}(x)) \\
&= (x \circledast W + b) * \alpha + \beta = x \circledast (W * \alpha) + (b * \alpha + \beta) \\
&= \text{Convolution}(x, \text{weight} = W * \alpha, \text{bias} = b * \alpha + \beta)
\end{aligned}
$$

Rewrite GlobalAvgPooling

$$\text{GlobalAvgPooling}(x) = \frac{1}{K * K} \sum_{k_i} \sum_{k_j} x_{..k_i k_j}$$
$$= \text{broadcast\_mul}(\text{sum}(\text{data}, \text{axis}=(2, 3)), \text{scale})$$

where scale equals $1/(K * K)$. Following this routine, we are able to rewrite complex operators into simpler ones, which can be further safely quantized.

Fusing Constant

After the fusion processes described above, we run a constant-fusing procedure to reduce graph complexity for better quantization performance.

Simulated quantization

Before we can make the whole computation graph integer-only, we should first rewrite floating-point numbers into simulated quantized representation. In the current implementation, we adopt a symmetric quantization approach to quantize floating-point vector x to signed 8-bit type x^Q, specifically, $x = sx^Q$ where $x \in \mathbf{R}^n, s \in \mathbf{R}, x^Q \in Z^n_{\text{int8}}$

After applying quantization, we reorder the operators in the graph for further processing.

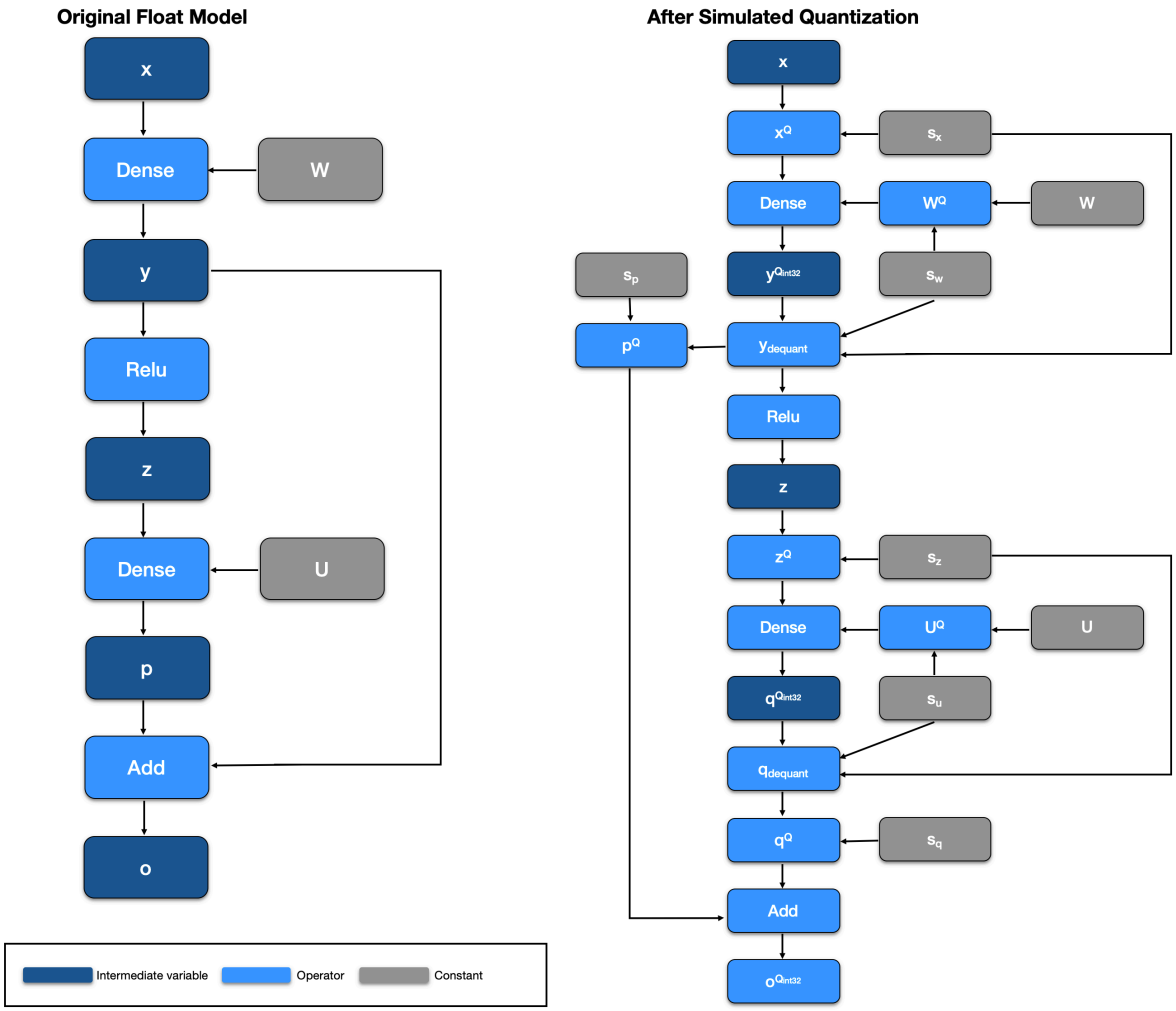
As `matmul` is the core of NN's workflows, we use it as an example to illustrate how to transform a floating-point operator to an integer operator.

let's define floating-point `matmul` as $y = Wx$, where $y \in \mathbf{R}^m, x \in \mathbf{R}^n, W \in \mathbf{R}^{m \times n}$. First we rewrite x, y and W into quantized representation $s_y * y^Q = (s_w W^Q)(s_x X^Q)$, and rewrite it into

$$y^Q = (\frac{s_w s_x}{s_y}) W^Q X^Q = s_q W^Q X^Q$$

where $s_q = \frac{s_w s_x}{s_y}$ is the requantization scalar.

In our approach, scalar s_y is determined in advance by calibration. With calibrated scalar s_y, for output y of each operator and weights scalar s_w, we can further determine requantization scalar s_q by definition. Thus, we can rewrite the original graph into an annotated graph as the figure shown below:



Calibrating Requantization Parameter

Assume our purpose is to quantize weight and activation into $[-127, 127]$ that can be placed into a signed 8-bit integer. We need to determine a bounding range $[-h, h]$, so that we can map data into $[-127, 127]$. Formally, we have $s_x = h/127, x^Q = \text{round}(\text{clip}(x; -h, h)/s_x)$, note that large values may need to be clipped for better quantization precision.

To simplify things, we do only layer-wise quantization. For quantizing weight w, we set $h = \max(\{|a||a \in x\})$. In terms of activation, we need to feed some realworld data to collect the intermediate result y. Then we can use a heuristic approach to calibrate a threshold h to get y^Q that best approximates y. If using MXNet's quantization package, we can utilize the entropy-based calibration method to find the best fit.

We take an approach of using shift bit instead of a floating point for requantization that reduces work in symbol realization. For a positive floating-point scale s, we rewrite it as $s \sim s_0 2^{-b}$, where s_0 and b are positive integer.

Realizing Integer-only Inference

After rewriting the graph, the float operation only occurs on `broadcast` operator, e.g., `broadcast_multiply.` Taking it as an example, this operator is mainly introduced by the requantization procedure. We rewrite it as $y = s_0 2^{-b} y^Q_{\text{int32}} = ((s_0 >> p)(y^Q >> q)) >> (r + b)$, where p, q and r can be calibrated for the best performance. The first two `shift` operators are

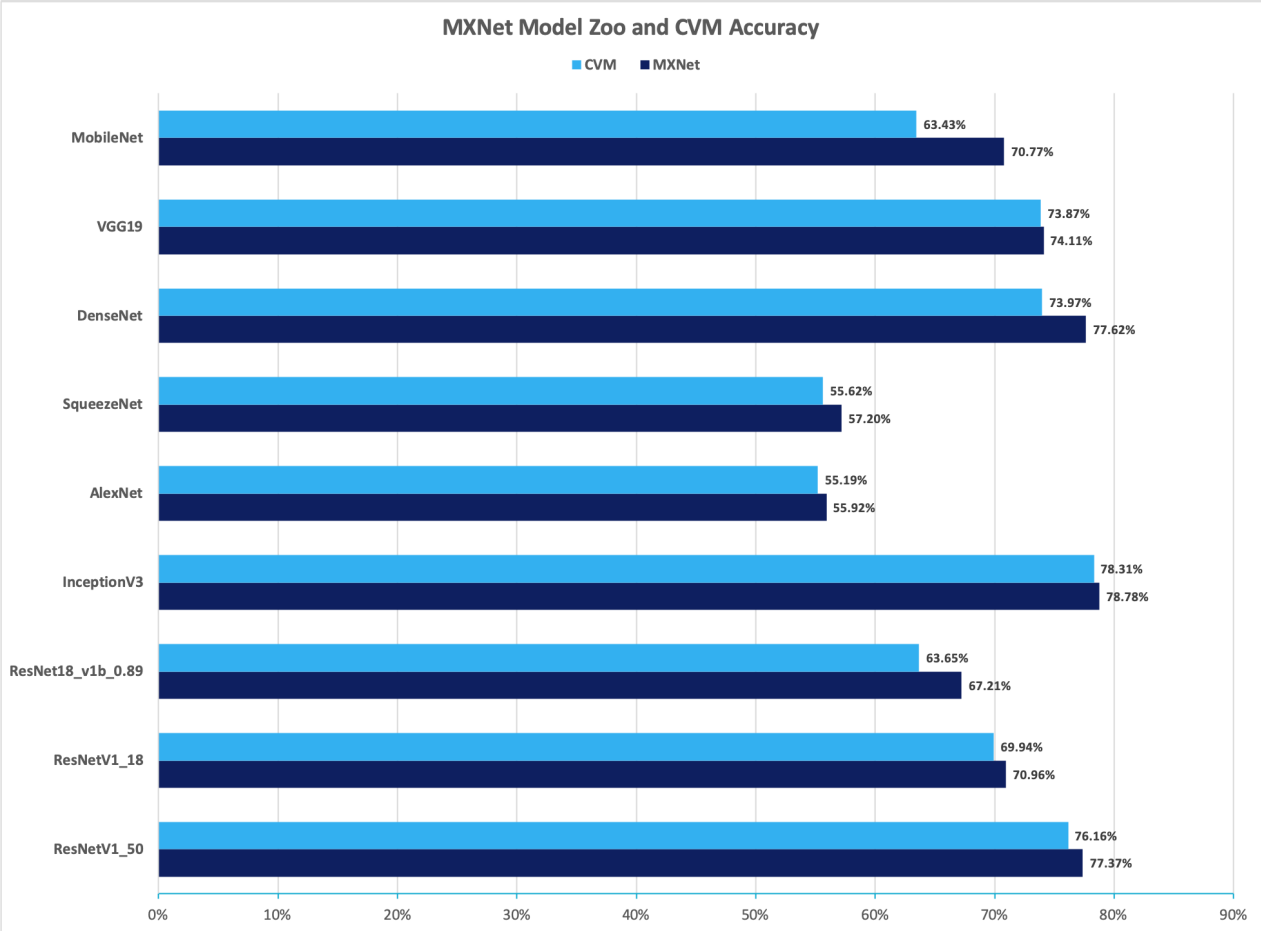
used to avoid overflow during computation, and the last `shift` is used for requantization. Note that both y and y^Q are tensors.

Experiment

Converting the original floating-point model to our CVM representation results in approximately 4x model size reduction while the model accuracy does not decrease significantly. Besides, we only introduce minor additional computation overheads, e.g. requantization, leading to the operations (OPs) to remain in the same order of magnitudes. All operators in the model can be further optimized using vectorization techniques, which will reduce the computation time dramatically, e.g., avx512-vnni instruction set.

We apply the proposed converter on pre-trained models with ImageNet dataset from MXNet Model Zoo. The result is shown as below:

Imagenet MODEL	**MXNet**	**CVM**
ResNetV1_50	77.37%	76.16%
ResNetV1_18	70.96%	69.94%
ResNet18_v1b_0.89	67.21%	63.65%
InceptionV3	78.78%	78.31%
AlexNet	55.92%	55.19%
SqueezeNet	57.20%	55.62%
DenseNet	77.62%	73.97%
VGG19	74.11%	73.87%
MobileNet	70.77%	63.43%



We can see that the accuracies for ResNetV1 and InceptionV3 on ImageNet dataset are retained after our quantization scheme.

We also apply the proposed converter on pre-trained models with MNIST dataset. The result is shown as below:

MNIST MODEL	MXNet	CVM
DigitalClashNet	99.18%	99.18%
LeNet	99.18%	99.16%
MLP	97.68%	97.69%

We can also see that our quantization scheme does not lose accuracy. DigitalClashNet, which has been deployed in Cortex Testnet and used in a DApp, retains the accuracy.

Conclusion

Based on MXNet, we have built a deterministic quantization framework, where model inference can be enabled on the limited-resource and strictly deterministic environment of blockchain, enabling a new domain of smart contracts with machine learning models. This framework can be found useful in a variety of applications: Decentralized Finance, Entertainment, Information service, Blockchain as a Service, etc.

Future work

Quantization is a relatively new field that has more to study, like federated training, assymetric quantization, channel/group-wise quantization, privacy, etc. We will continue to explore possible applications and implementations of quantization in MXNet.