### Quantizing NN models for deployment on blockchain

Towards A Novel Deterministic Inference Infrastructure on Blockchain

# Introduction

There are emerging interests in deploying deep learning models on various platforms and devices. Deep networks are not just being used on supercomputers or GPU cloud computing that have high performance but are also seeing increasingly used in the edge devices that have lower computing capabilities with constraints in memory and power consumption. The situation is even more critical to deploy DNN models on the blockchain that has even more crucial environment. Not to mention that being deterministic in the blockchain environment is another issue that needs to be solved, e.g., each running of a single model on the different devices must produce a bit-level identical result.

In this post, we propose a methodology to accelerate DNN models' inference and eliminate nondeterministic behavior in model inference for blockchain adoption. Before we go into the detail of implementation, we first go through the observation and intuition behind this methodology.

Unlike GPU, the float-point number arithmetic causes nondeterministic results, e.g. summation over a series of float-point number, which is undesirable on the edge devices and blockchain. In a public blockchain, transactions need to be verified and reached consensus by the nodes before being written on the blockchain. Each node has its own hardware specification running a different version of operating systems. As a result, being deterministic is crucial on a constrained environment.

Thus, researchers have proposed several approaches to tackle this problem:

1. **Fake Quantization**: quantizing float-point number into 8-bit integer and transfer data to the accelerator, which takes linear time to apply this operation. The most costly part of the calculation, e.g., conv, only happens in the accelerator that dedicated in 8-bit arithmetic. Afterward, results are transformed back to float-point.
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 post-quantization

The current implementation in MXNet's Contrib library follows the fake quantization routine and redirect the computation to MKLDNN math library in runtime. However, in blockchain's deterministic-sensitive scenario, the float-point number is unacceptable. Integer-only inference, on the other hand, suits blockchain's heterogeneous environments. Besides, the numerical bound is checked to avoid integer overflow by utilizing graph level rewriting. Therefore, we propose to adopt integer-only inference as our methodology.

# Implementation

We implement a converter using MXNet's nnvm module called **MRT** (Model Representation Tool) that transforms a plain MXNet model that can be inferred on the **Cortex Virtual Machine** (CVM), the runtime environment for smart contracts with machine learning models on the blockchain.

## Fusion and Operator Rewriting

### Fuse Constant

After the fusion processes listed below, we do constant-fuse process to reduce graph complexity for better quantization performance.

#### 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 is only held when $n$ is less than $2^{16}$. In other words, we cannot assume abense of overflow when $n$ is large, which may introduce nondeterministic behavior during parallel computing. To resolve this problem, we decomposite the computation into small peices in graph level and aggreate the results. Mathmatically, $s =< x^{(1)}, y^{(1)} > + < x^{(2)}, y^{(2)} > + \ldots + < x^{(K)}, y^{(K)} >$, $x^{(k)}$ is the $k$-th part of vector $x$ with each part of vector has length smaller than $2^{16}$.

Matrix multiplication operator `matmul` can also be rewritten in the same fashion, which results in a series of `elemwise_add` operator that sum over several intermediate matrices. Although this rewriting introduces additional operators in the computation graph, semantic remains unchanged.

#### Fuse BatchNorm

*gamma, beta, data_mean, data_var*: attributes.

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

, where $\alpha$ is $\frac{\gamma}{\sigma}$ and $\beta$ is $\lambda - \mu * \gamma / \sigma$.

when $y = \text{Convolution}(x)$, we can get equation as belows:

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

## Simulated quantization

Before we can make the whole computational graph integer-only, we should first rewrite float-point number into simulated quantized representation. In the current implementation, we adopt a symmetric quantization approach to quantize float-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 on how to transform float-point operator to an integer operator.

let's define float-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 weighted scalar $s_w$, we can further determine requantization scalar $s_q$ by definition. Thus, we can rewrite the original graph to an annotated graph as the figure shown below:

## Calibrating Requantization Parameter

Suppose our purpose is to quantize weight and activation into $[-127, 127]$ that can be placed into a signed 8-bit integer. We need to determinate a 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 value may need to be cliped in order to obtain better quantization precision.

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

We adopt a simple method in our implementation, which uses shift bit instead of a floating scale for requantization that will reduce work in symbol realization. For a positive float-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

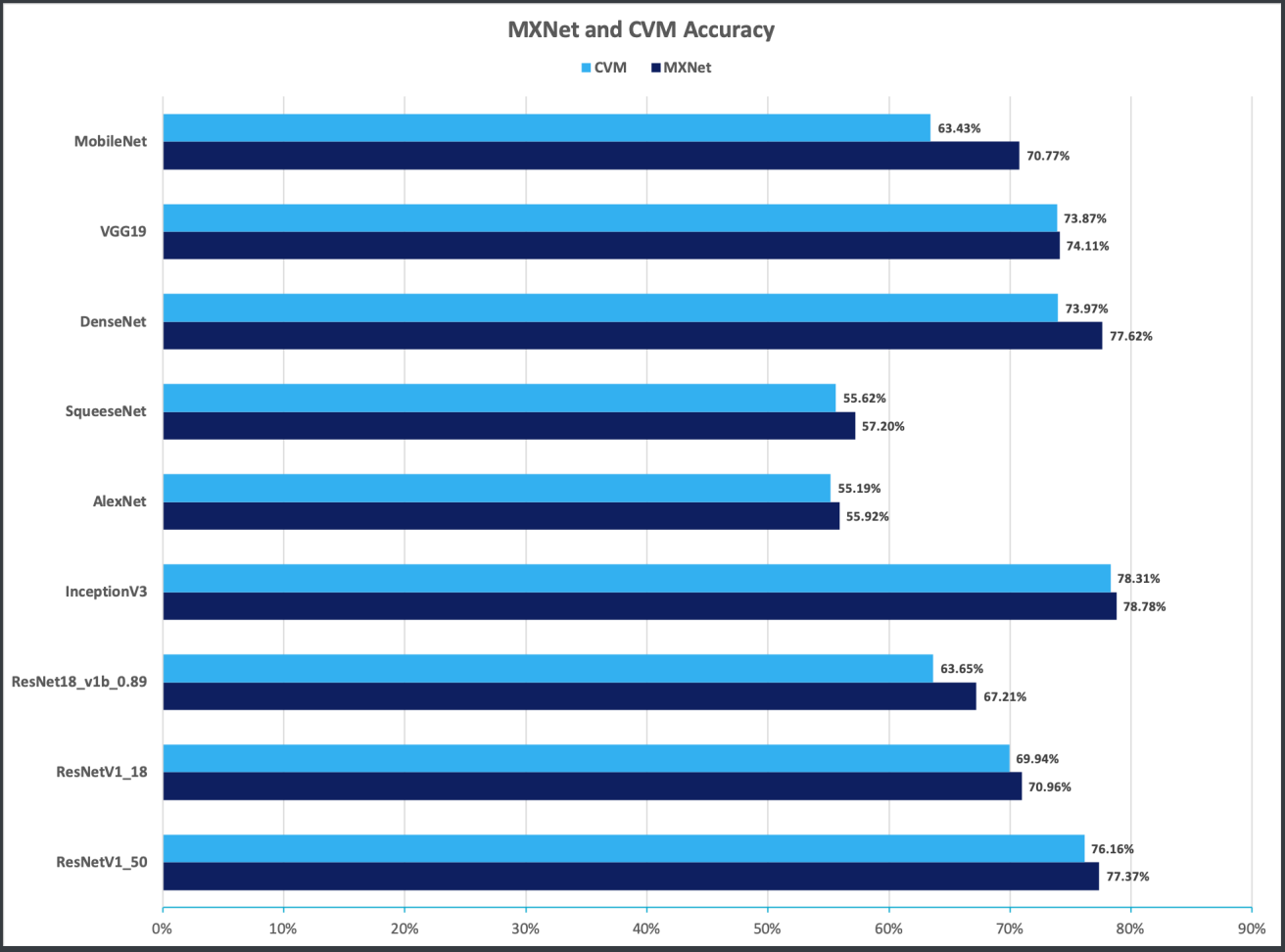
After rewriting the graph, the float operation only occurs on `broadcast` operator, e.g., `broadcast_multiply.` Taking it as an example, this operator is mainly introduced by the requantization procedure. We rewrite it as $y = s_0 2^{-b} y^Q_{\text{int32}} = ((s_0 >> p)(y^Q >> q)) >> (r + b)$, where $p, q$ 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 float-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 operators (OPs) to be in the same order of magnitudes. All operators in the model can be optimized using vectorization techniques, which will reduce the time of computation intensively, 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% |
| SqueeseNet | 57.20% | 55.62% |
| DenseNet | 77.62% | 73.97% |
| VGG19 | 74.11% | 73.87% |
| MobileNet | 70.77% | 63.43% |




We can observe that the accuracy for ResNetV1 and InceptionV3 on ImageNet dataset retains 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 observe 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

Using MXNet's quantization technology, model inference can be enabled on the limited-resource and strict environment of blockchain, unlocking a novel domain of smart contracts with machine learning models. The use case could be DeFi, Entertainment, Information service, BaaS, etc.

## Future work

Enhancing privacy, accuracy, and efficiency. Mobile/edge computing realization is also one of our goals.