TypeCNN: CNN Development Framework With Flexible Data Types

Petr Rek and Lukas Sekanina
Brno University of Technology, Faculty of Information Technology, IT4Innovations Centre of Excellence
Brno, Czech Republic
Email: rekp@email.cz; sekanina@fit.vutbr.cz

Abstract—**The rapid progress in artificial intelligence technologies based on deep and convolutional neural networks (CNN) has led to an enormous interest in efficient implementations of neural networks in embedded devices and hardware. We present a new software framework for the development of (approximate) convolutional neural networks in which the user can define and use various data types for forward (inference) procedure, backward (training) procedure and weights. Moreover, non-standard arithmetic operations such as approximate multipliers can easily be integrated into the CNN under design. This flexibility enables to analyze the impact of chosen data types and non-standard arithmetic operations on CNN training and inference efficiency. The framework was implemented in C++ and evaluated using several case studies.**

I. Introduction

The rapid progress in machine learning technology based on *deep neural networks* (DNNs) is tightly connected with the availability of open source libraries implementing DNNs and widely accessible high-performance computing systems that are needed for training complex DNNs. A significant attention is currently devoted to hardware and embedded implementations of DNNs as only highly optimized implementations of DNNs can be executed on low power embedded systems, IoT nodes and in consumer electronics [1]. However, finding a good tradeoff between the quality of service, performance and energy efficiency is a challenging task in the context of DNNs. In this paper, we propose a new software framework developed to implement *convolutional neural networks* (CNNs) and analyze the impact of simplified data representations and arithmetic operations on CNN properties.

Artificial neural networks (NNs) consist of layers that contain many computing primitives – *artificial neurons* – operating in parallel. A typical neuron sums weighted input signals and produces an output signal according to a non-linear activation function applied on the sum. CNNs are deep NNs primarily introduced for image classification. In addition to common fully-connected layers of neurons, they employ special layers, in particular, *convolutional* layers capable of extracting useful features from the raw data.

In the *training phase*, which is typically implemented with a backpropagation algorithm, the objective is to determine suitable values of the weight connections and other items (such as convolutional kernels and biases) in order to minimize a given error metric.

Various approaches have been developed to simplify CNNs for embedded systems. A common approach is to reduce bit width for the weights and all intermediate results. Arithmetic operations are then performed on reduced number of bits either exactly or approximately. Approximate multipliers applied in CNNs have already led to significant energy savings [2].

When an input (image) and a fully trained CNN are provided, CNN output value (image class) is computed in the so-called *inference* procedure which involves just forward computations. However, the training procedure requires to perform forward computations (to determine the error) as well as backward computations to update the weights. As training can be very sensitive to gradients of the activation functions used in a given CNN, it is almost exclusively performed in a common floating point (FP) number representation in order to ensure reasonable convergence and classification accuracy. However, many papers showed that the inference process can be simplified without any significant impact on the classification accuracy. Instead of the common 32-bit FP representation, a suitable fixed point representation is used and the number of bits for weights and/or activations is reduced (see a survey of techniques in Table 3 of [1]).

In this paper, we present a new open-source software framework TypeCNN[1] created to simplify the development of (approximate) convolutional neural networks for embedded systems and specialized hardware. While existing frameworks are well-performing, they do not allow the designer to introduce all relevant modifications to CNNs that are needed for their evaluation on very specific hardware platforms. For example, one could be interested in a detailed analysis of the impact of a given simplified data type on inference and training under the assumption that only limited resources are available for all accumulators (for example, note that no overflows are assumed in the accumulation operations in Ristretto [3]). One could also be interested in the impact of (approximate) fault-tolerance mechanisms (that are introduced to a CNN) on the accuracy of classification in various environments. Hence, the main focus of TypeCNN is on providing more flexible modifications of data types, arithmetic operations and inference phase than commonly available frameworks support. In particular, TypeCNN provides three independent data type aliases – for the weights, the forward propagation (inference)

[1] https://github.com/rekpet/TypeCNN

and the backward propagation – that have to be defined in the compilation time. Different layers can use different data types. The user can create a CNN with different data types and different (approximate) arithmetic operations in all layers. This flexibility enables to analyze the impact of chosen data types and non-standard arithmetic operations on CNN training and inference efficiency.

II. Related work

The most popular frameworks for DNNs are Caffe and TensorFlow[2]. Higher-level libraries such as Keras[3], developed in Python for TensorFlow and other frameworks, provide more comfortable user interface and faster development. There are many smaller frameworks, for example, TinyDNN or even one-man projects such as SimpleCNN[4].

As these major frameworks are highly optimized for performance, it is quite difficult to introduce very specific modifications (such as the support for various data types in different NN layers or approximate arithmetic operators) to their source code and, at the same time, to keep high performance.

By means of Ristretto [3] – a CNN approximation framework developed over Caffe – it is possible to investigate the tradeoffs between various number representations that can be employed in CNNs and the classification accuracy. Based on a detailed analysis, Ristretto can find suitable bit widths for weights, activations, and intermediate results of convolutional and fully connected layers. Ristretto supports various simplified representations such as dynamic fixed point and minifloats. All these computations are internally simulated using standard floating point representation which enables to use highly optimized matrix multiplication routines for both the forward and backward propagation and thus reach high performance.

Specialized hardware accelerators utilizing simplified data representation have been introduced to accelerate processing of already trained DNNs. For example, in Tensor Processing Unit (TPU), only 8-bit operations are implemented in MAC units. TPU exploits a systolic array composed of 64 thousands of 8-bit MAC units [4]. The approximation techniques developed for circuit implementations of NNs were surveyed in [5]. As millions of multiplications have to be performed in the inference phase of complex CNNs, various approximate multipliers specialized for CNNs have been introduced to reduce the overhead of multiplication [2]. Since fine-tuning conducted after inserting approximate multipliers into CNNs can require executing a non-standard learning algorithm for some approximate multipliers [2], the use of popular CNN libraries is not directly possible.

III. TypeCNN: A new CNN software framework

The TypeCNN implements all key operations over CNNs: design, training, and validation.

```
auto trainingData = IdxParser::parseLabelledImages
("tr-imgs.idx3","tr-labels.idx1",10);
auto validationData = IdxParser::parseLabelledImages
("test-imgs.idx3","test-labels.idx1",10);

auto inputDimensions =
  trainingData[0].first.getDimensions();

auto layer1 = std::make_shared<Convolution>
          (inputDimensions, 1, 8, 5, 0, true);
auto layer2 = std::make_shared<ReLU>
          (layer1->getOutputSize());
auto layer3 = std::make_shared<MaxPooling>
          (layer2->getOutputSize(), 2, 2);
auto layer4 = std::make_shared<FullyConnected>
          (layer3->getOutputSize(),
          Dimensions{ 10, 1, 1 }, true);
auto layer5 = std::make_shared<Sigmoid>
          (layer4->getOutputSize());

auto cnn = ConvolutionalNeuralNetwork
          (TaskType::Classification);
cnn.addLayer(layer1);
cnn.addLayer(layer2);
cnn.addLayer(layer3);
cnn.addLayer(layer4);
cnn.addLayer(layer5);

TrainingSettings settings;
settings.epochs = 5;
settings.errorOutputRate = 10000;
settings.shuffle = true;

cnn.enableOutput();

auto optimizer = std::make_shared
            <SgdWithMomentum>();
optimizer->momentum = 0.6f;

cnn.train(settings, trainingData,
   LossFunctionType::MeanSquaredError, optimizer);

cnn.validate(validationData);
```

Listing 1. Definition of a simple CNN in TypeCNN

The target NN is modeled using an object-oriented paradigm in C++. The framework implements common CNN layers with C++ classes: ConvolutionalLayer, PoolingLayer, ActivationLayer, DropoutLayer, and FullyConnectedLayer, where forwardPropagation() and backwardPropagation() are their methods performing the inference and back propagation operation. The ActivationLayer class (supporting sigmoid, ReLU, Leaky ReLU, Tanh and Softmax) is implemented independently of the other layers in order to provide more flexibility. In addition to common CNN frameworks, ConversionLayer was proposed in order to enable flexible changes in data types between the neighboring layers of the NN in the inference procedure. Example of a CNN definition in TypeCNN is given in Listing 1. In order to accelerate processing of convolutional and pooling layers, all relevant values (such as connection patterns of neurons or kernel addressing schemes) are pre-calculated in the initialization phase and the inference and training procedures use them.

[2]See caffe.berkeleyvision.org and tensorflow.org
[3]keras.io
[4]See github.com/tiny-dnn and github.com/can1357/simple_cnn

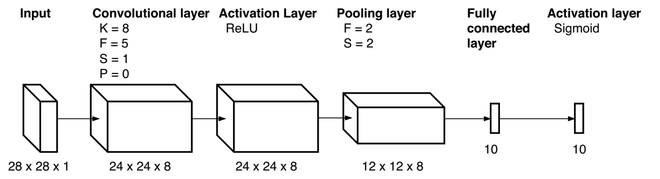

Fig. 2. CNN architecture used to compare TypeCNN with other frameworks (K – number of filters, F – window size, S – stride, P – padding).

TABLE I
BASIC COMPARISON OF CNN FROM FIG. 2 ON MNIST WHEN TRAINED USING DIFFERENT FRAMEWORKS FOR 10 EPOCHS

Framework	Training time [s]	Accuracy [%]
SimpleCNN	746	97.20
TinyDNN	151	98.23
Keras (TensorFlow)	744	98.34
TypeCNN	538	98.31

The inputs and outputs of layers, weights and other parameters are modeled using the Image class in which a common array known from C is used to implement vectors and 2- and 3-dimensional arrays. The training algorithm is based on the backpropagation algorithm in which five different optimizers can be employed: Stochastic Gradient Descent (SGD), Momentum, Nesterov momentum, Adagrad, and Adam [6]. The default settings of the optimizers is as recommended in the literature. All the optimizers support the weight decays principles. The error function is either the mean squared error (MSE) or cross entropy. In addition to the random weight initialization, a smart initialization procedure is supported to set the weights to small but non-zero values because unconstrained weight seeding can quickly lead to overflows for the FX number representation.

The framework also supports reading and storing network architectures (an xml file) and weights (a separate text file). The input data are accepted in idx, binary and png formats.

The key feature of TypeCNN, enabling thus modelling of very heterogenous data representations and manipulations in CNNs, is the existence of three type aliases:

- *WeightType* is a data type used for weights during inference and when they are saved to a disk.
- *ForwardType* is a data type used for all the computations performed during inference.
- *BackwardType* is a data type used while updating learnable parameters (gradients, precise weights etc.) in the training phase.

The data type used for the weights and inference can be defined separately for each layer. While the inference procedure is performed in the data type defined for a particular layer (including all intermediate results), training is always performed using the same data type in all layers. The default data type is the float.

The user can introduce application-specific data types. For example, TypeCNN provides a fixed point data representation FX$< f, p >$, where f is the number of integer bits and p is the number of fractional bits. FX is based on the common 32 bit integer (or 64 bits in FX64). The data size and operations are implemented using bit masks. In order to accelerate data operations, lookup tables can be employed. The data type utilizes the concept of saturation which means that overflow (resp. underflow) immediately leads to returning the maximum (resp. minimum) value on a given number representation. As this data type can also be used in the accumulation phase, we can analyze the impact of the inexact summing of products and activations. Other relevant data types such as dynamic fixed point, minifloat or various quantization approaches can easily be integrated to TypeCNN. Contrasted to other frameworks in which the quantization is supported for some layers, but the data processing is internally implemented on floating point representation (e.g. Ristretto), the training can be performed using a user-specified data type such as FX.

IV. EXPERIMENTAL EVALUATION

As TypeCNN is not targeting the largest existing CNNs, but rather smaller CNNs, potentially running on very specialized hardware, we devoted this experimental evaluation to analysing on how the classification accuracy of basic CNNs is changed when FX data type is used for all operations in these CNNs including accumulation and activation. However, before conducting these investigations, we will report a basic comparison with several other CNN frameworks. All experiments were performed on Intel Core i7-4720HQ (2.60 GHz$\times$8) CPU with 8 GB RAM and Ubuntu 14.04 LTS (64-bit) OS.

A. Basic functionality

In order to evaluate basic functionality and performance of TypeCNN, we implemented a simple CNN (taken from SimpleCNN, see Fig. 2) and compared the time and quality of learning of TypeCNN with three frameworks (SimpleCNN, TinyDNN and Keras). Table I reports the average learning time and the average classification accuracy on MNIST from 5 independent runs on each framework. TypeCNN used floats for training as well as inference. All frameworks were tested with the following settings: stochastic gradient descent learning (with momentum), 10 epochs, batch size = 1, loss function = MSE, learning rate = 0.01, momentum = 0.9, weight decay = 0.0. These elementary evaluations proved that TypeCNN is capable of providing competitive results for mid-size networks on a personal computer.

B. Fixed point number representation in a common MLP

As TypeCNN enables to construct common multi-layer perceptrons (MLP), we also considered this type of NN in our study. In this case, the input layer contains 784 neurons (28 $\times$ 28 pixels), 256 and 64 neurons are included in the hidden layers (with the tanh activation) and the output layer has 10 neurons (with the Leaky ReLU activation). In the first phase, the MLP was trained on MNIST in 10 epochs in which all operations were performed on floats. After changing the data type (for inference and weights), 5 epochs were executed in

TABLE II
ACCURACY OF MLP ON MNIST AFTER: (A) 10 EPOCHS IN FLOATS AND CONVERSION TO FX; (B) 10 EPOCHS IN FLOATS AND FINE-TUNING FOR 5 EPOCHS IN FX; (C) 10 EPOCHS WITH FX REPRESENTATION USED FOR INFERENCE AND WEIGHTS AND THE BACKWARD OPERATION IN FLOATS.

Data type	Validation accuracy [%]		
	(A)	(B)	(C)
float	97.37	97.92	97.35
FX$< 16, 16 >$	97.36	97.92	97.38
FX$< 12, 4 >$	11.39	96.00	96.20
FX$< 8, 8 >$	96.77	97.91	97.36
FX$< 4, 12 >$	97.20	97.90	97.27
FX$< 6, 2 >$	14.69	85.84	86.34
FX$< 4, 4 >$	9.01	95.94	95.99
FX$< 2, 6 >$	23.75	78.20	86.65

TABLE III
ACCURACY OF LENET-5 ON MNIST AFTER: (A) 10 EPOCHS IN FLOATS AND CONVERSION OF THE WEIGHT AND INFERENCE DATA TYPE TO FX; (B) ADDITIONAL FINE-TUNING FOR 5 EPOCHS; (C) 10 EPOCHS WITH FX REPRESENTATION USED FOR INFERENCE AND WEIGHTS AND THE BACKWARD OPERATION IN FLOATS.

Data type	Validation accuracy [%]		
	(A)	(B)	(C)
float	98.60	99.17	98.62
FX$< 16, 16 >$	86.37	99.15	98.79
FX$< 8, 8 >$	86.91	99.13	98.67
FX$< 4, 4 >$	10.58	79.59	64.38

the fine-tuning phase. Table II shows the accuracy obtained at the end of the first phase and at the end of fine-tuning. All results reported are average values from 5 independent training runs of a given NN. As the weights in NN are small, the accuracy before fine-tuning is primarily influenced by the number of fractional bits.

Table II (C column) also shows the accuracy obtained after 10 epochs when the fixed point representation is used for weights and inference from the beginning of training (the backward procedure is always implemented in floats). The results are very close to the experiment reported in column B.

C. Fixed point number representation in CNN

The experiments reported in previous section were repeated for CNN LeNet-5 containing three convolutional layers (with 6, 16 and 120 filters) and two fully connected layers,

Table III (the A column) shows the classification accuracy obtained at the end of the first phase of training (10 epochs; all data types are floats) and after replacing the floats with another data type (weights and inference). The B column gives the classification accuracy after additional fine-tuning for 5 epochs. The C column shows the accuracy obtained after 10 epochs when the fixed point representation is used for weights and inference from the beginning of training.

We also tested the impact of the data type used for weights on the accuracy when the data type for inference remains

TABLE IV
THE IMPACT OF THE DATA TYPE USED FOR THE WEIGHTS WHEN THE NETWORK IS TRAINED USING FLOATS (10 EPOCHS) AND FINE-TUNED USING FX$< 8, 8 >$ IN THE INFERENCE PHASE FOR 5 EPOCHS.

Weights (data type)	Validation accuracy [%]	
	Before re-training	After re-training
FX$< 8, 8 >$	86.91	99.13
FX$< 4, 4 >$	81.97	98.97
FX$< 1, 3 >$	31.85	98.61
FX$< 2, 2 >$	10.05	97.67
FX$< 1, 1 >$	9.80	93.77

TABLE V
THE BEST-OBTAINED CLASSIFICATION ACCURACY AFTER ADDITIONAL FINE-TUNING WITH FLOATS OR FX$< 8, 8 >$ USED FOR THE WEIGHTS AND THE INFERENCE PROCEDURE.

Data set	Validation accuracy [%]	
	Float	FX$< 8, 8 >$
MNIST	99.37	99.37
CIFAR-10	73.59	73.54

unchanged (i.e. FP$< 8, 8 >$). Table IV shows that even 2-bit weights can lead to a reasonable accuracy after fine-tuning.

Finally, Table V shows the best obtained classification accuracy on MNIST and CIFAR-10 with additional fine-tuning on floats (see column Float) and additional fine-tuning with FX$< 8, 8 >$ (the backward procedure was again performed with floats).

V. CONCLUSIONS

We presented a new open-source software framework TypeCNN and analyzed the impact of employing fixed point data types across all the CNN layers on the classification accuracy. Our future work will be devoted to extending this first version of TypeCNN to support more specialized data types and improving performance.

This work was supported by Czech Science Foundation project 19-10137S.

REFERENCES

[1] V. Sze, Y. Chen, T. Yang, and J. S. Emer, “Efficient processing of deep neural networks: A tutorial and survey,” *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[2] S. S. Sarwar, S. Venkataramani, A. Ankit, A. Raghunathan, and K. Roy, “Energy-efficient neural computing with approximate multipliers,” *J. Emerg. Technol. Comput. Syst.*, vol. 14, no. 2, pp. 16:1–16:23, 2018.

[3] P. Gysel, J. Pimentel, M. Motamedi, and S. Ghiasi, “Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 29, no. 11, pp. 5784–5789, 2018.

[4] N. P. Jouppi, C. Young, N. Patil *et al.*, “In-datacenter performance analysis of a tensor processing unit,” in *Proc. of the 44th Annual Int. Symposium on Computer Architecture*. ACM, 2017, pp. 1–12.

[5] P. Panda, A. Sengupta, S. S. Sarwar, G. Srinivasan, S. Venkataramani, A. Raghunathan, and K. Roy, “Invited – cross-layer approximations for neuromorphic computing: From devices to circuits and systems,” in *53nd Design Automation Conference*. IEEE, 2016, pp. 1–6.

[6] S. Ruder, “An overview of gradient descent optimization algorithms,” *CoRR*, vol. abs/1609.04747, 2016.