Quantisation

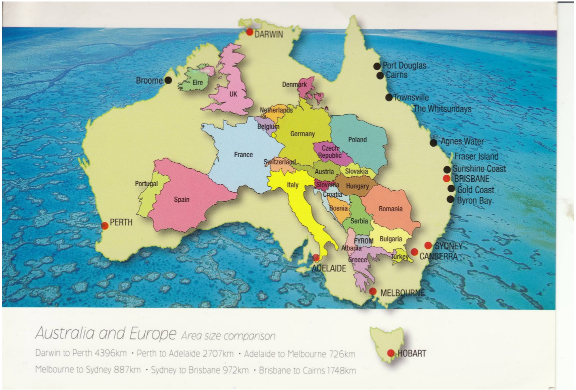
Efficient implementation of convolutional neural networks

Philip Leong and Julian Faraone

Computer Engineering Lab
The University of Sydney

July 2019 / NOVA Workshop

Australia



- Australia 25M, Guangdong 100M

Outline

THE UNIVERSITY OF SYDNEY

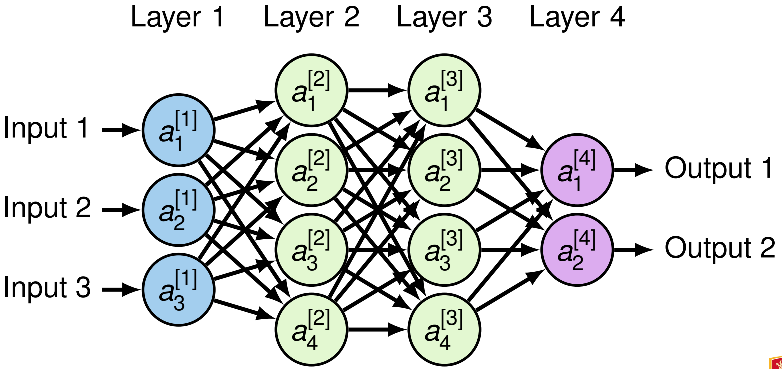
Outline

Scope

- There are several degrees of freedom to explore when optimising DNNs
    - NN architecture (SqueezeNet, MobileNet, EfficientNet [TL19])
    - Compression (SVD, Deep Compression, Circulant [Din+17])
    - Quantization (FP16, TF-Lite, FINN, DoReFa-Net)
- This talk: quantisation

Backpropagation Algorithm [HH18]

- Forward Pass $z^{[i]} = W^{[i]}a^{[i-1]}, a^{[i]} = \sigma(z^{[i]})$
- dActivations $\delta^{[i]} = diag(\sigma'(z^{[i]}))(W^{[i+1]})^T\delta^{[i+1]}$
- dWeights $\Delta W^{[i]} = \delta^{[i]}(a^{[i-1]}))^T$

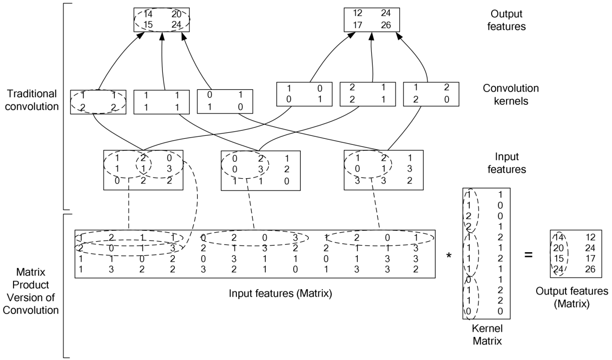


Backpropagation Algorithm [HH18]

function BACKPROP(x,y)
    $a^{[1]} = x$
    for $i = 2, \ldots, L$ **do** ▷ forward pass
        $z^{[i]} = W^{[i]} a^{[i-1]}$
        $a^{[i]} = \sigma(z^{[i]})$
    end for
    $\delta^{[L]} = diag(\sigma'(z^{[L]}))(a^{[L]} - y)$
    for $i = L-1, \ldots, 2$ **do** ▷ backward pass δ
        $\delta^{[i]} = diag(\sigma'(z^{[i]}))(W^{[i+1]})^T \delta^{[i+1]}$
    end for
    for $i = L, \ldots, 2$ **do** ▷ backward pass ΔW
        $\Delta W^{[i]} = \delta^{[i]} (a^{[i-1]})^T$
    end for
    return ΔW
end function

Convolution Layer as MM

- Convolution layers converted to GEMM [CPS06]
- Efficient BLAS libraries can be exploited



DNN Computation

Computational problem in DNNs is to compute a number of dot products

$$a = \sigma(\boldsymbol{w}^T \boldsymbol{x}) \tag{1}$$

where

- σ is an element-wise nonlinear activation function
- $\boldsymbol{x} \in \mathbb{R}^{i.w.h}$ is the input vector
- $\boldsymbol{w} \in \mathbb{R}^{i.w.h}$ is the weight vector

Arithmetic Intensity

- Computation of a DNN layer is MV multiplication
- For MV multiply is $O(1)$, for MM is $O(b)$ where b is block size
- Efficient CPU/GPU implementations use batch size $\gg 1$ (process a number of inputs together)
- For latency-critical applications (e.g. object detection for self-driving car), we want a batch size of 1
- Make sure comparisons are at the same batch size!

Block Floating Point

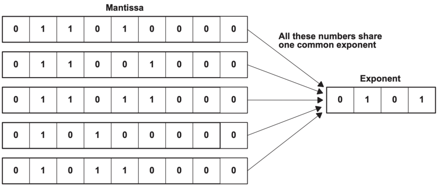
Block floating point

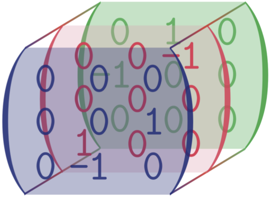
Operations mostly integer

dim=-1 The whole tensor shares a single

dim=n all elements of a dimension share a

Use fx(k) notation





Stochastic Rounding

- Suppose we want to round 10.4 to an integer
- Stochastic rounding

$$r(x) = \begin{cases} \lfloor x \rfloor & \text{with probability } 1 - (x - \lfloor x \rfloor) \\ \lfloor x \rfloor + 1 & \text{with probability } x - \lfloor x \rfloor \end{cases}$$

- Implement by adding a random number $\in [0, 1)$ and truncating

THE UNIVERSITY OF SYDNEY

Python Stochastic Rounding Implementation

```
import numpy as np
from timeit import timeit

# from https://medium.com/@minghz42/what-is-stochastic-rounding-b78670d0c4a
# Generates s randomly rounded values of x. The mean should be x.
def rstoc(x, s):
    r = []
    for i in range(s):
        decimal = abs(x - np.trunc(x))
        random_selector = np.random.random_sample()
        if (random_selector < decimal):
            adjustor = 1
        else:
            adjustor = 0
        if(x < 0):
            adjustor = -1 * adjustor
        r.append(np.trunc(x) + adjustor)
    return r
```

Testing Stochastic Rounding

```python
# computes elementwise relative error of elements of lists x1 and x2
def relative_error(x1, x2):
    ret = []
    for (i, r) in zip(x1, x2):
        ret.append(abs(i - r) / r)
    return ret

# For each element x of the list v, compute the mean of rstoc(x, s). Return as another list
def E_seq(v, s):
    return map(lambda x : np.mean(rstoc(x, s)), v)

spower = 7      # samples to try (execution time grows exponentiall)
v = [-1.234, 0.5, 0.6789]

# this function runs
def run_rstoc():
    print("Values")
    for i in range(spower):
        print(10 ** i, list(E_seq(v, 10 ** i)))
    print("Relative Errors")
    for i in range(spower):
        print(10 ** i, list(relative_error(E_seq(v, 10 ** i), v)))
t_rstoc = timeit(run_rstoc, number=1)
print("Execution time rstoc()", t_rstoc, '\n')
```

Output Stochastic Rounding

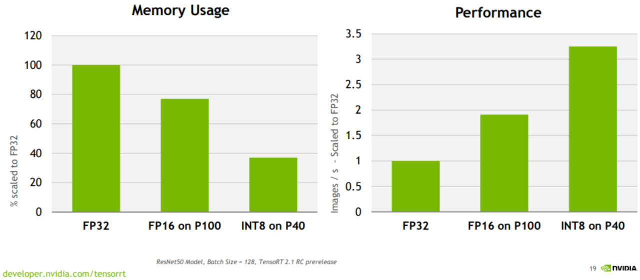
```
Values
1 [-1.0, 0.0, 0.0]
10 [-1.2, 0.6, 0.5]
100 [-1.26, 0.44, 0.7]
1000 [-1.225, 0.511, 0.685]
10000 [-1.2389, 0.5005, 0.6842]
100000 [-1.23358, 0.50052, 0.67682]
1000000 [-1.233241, 0.500033, 0.678435]
Relative Errors
1 [-0.18962722852512154, 1.0, 0.47297098247164543]
10 [-0.10858995137763362, 0.3999999999999999, 0.11621741051701277]
100 [-0.061588330632090814, 0.020000000000000018, 0.016349977905435263]
1000 [-0.0032414910858995167, 0.040000000000000036, 0.004271615849167628]
10000 [-0.00826580226904375, 0.005200000000000934, 0.006922963617616625]
100000 [-0.00024311183144243677, 0.0013999999999999568, 0.0017970245986154057]
1000000 [-0.00020826580226900463, 0.0009320000000000439, 0.0015510384445425994]
Execution time rstoc() 25.999158156999783
```

Outline

Outline

Role of Wordlength on Performance

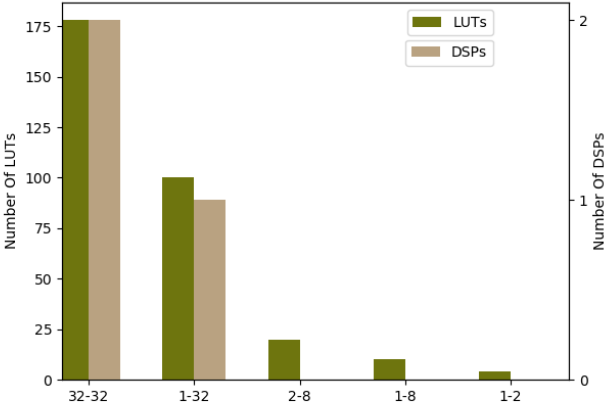
- CPU/GPU
  - Floating point performance comparable to fixed
  - Integer data types usually vectorisable hence faster
  - Nvidia offers FP64, FP32 and FP16 ($>$ Tegra X1 and Pascal)
- FPGA
  - Datapath is flexible
  - No floating point unit so fixed point normally preferred



THE UNIVERSITY OF SYDNEY

Role of Wordlength on Resources

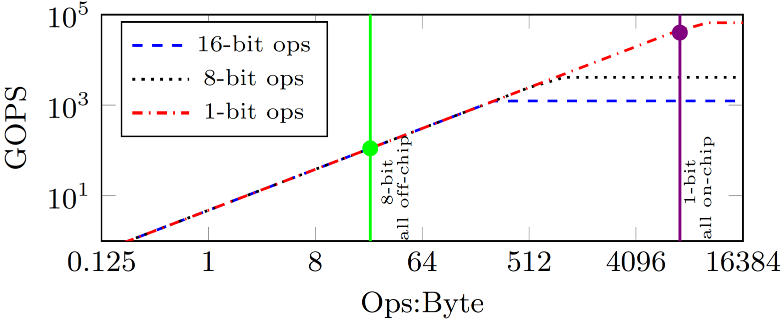
- X axis is bitwidth (weight-activation) and Y axis Number of LUTs/DSPs for MAC
- For k-bits, area is $\mathcal{O}(k^2)$



Roofline Model

Roofline model for Xilinx ZU19EG

- X axis is computational intensity (ops to perform / byte fetch), Y axis is performance
- Diagonal parts show memory-bandwith limited space
- Horizontal parts show computation limited space
- Actually this is a better metric to optimise than say GOPs/s
- Low precision extremely advantageous for performance



Integer Quantization [Jac+18]

A way to map numbers $r \in \mathbb{R}$ to unsigned integers $q \in \mathbb{U}+$ is via an affine transformation

$$r = S(q - Z) \tag{2}$$

- $\mathbb{U}+$ is the set of unsigned W-bit integers
- S, Z are the quantisation parameters
  - $S \in \mathbb{R}+$ represents a scaling constant
  - $Z \in \mathbb{U}+$ represents a zero-point

Integer MM [Jac+18]

- $N \times N$ MM defined as

$$r_3^{(i,k)} = \sum_{j=1}^{N} r_1^{(i,j)} r_2^{(j,k)}, \tag{3}$$

substituting $r = S(q - Z)$ (2) and rewriting[1] we get

$$q_3^{(i,k)} = Z_3 + M\left(NZ_1Z_2 - Z_1a_2^{(k)} - Z_2\overline{a}_1^{(i)} + \sum_{j=1}^{N} q_1^{(i,j)} q_2^{(j,k)}\right) \tag{4}$$

- Multiplication with $M = \frac{S_1 S_2}{S_3}$ is implemented in (high-precision) two's complement fixed point
- $a_2^{(k)}$ and $\overline{a}_1^{(i)}$ together only take $2N^2$ additions
- Sum in (4) takes $2N^3$ and is a standard integer MAC
- CPU implementation uses uint8, accumulated as int32

[1] A good exercise is to derive Equation (4) from (3) and (2)

Quantisation Range [Jac+18]

For each layer, quantisation parameterised by (a,b,n):

$$\begin{aligned} clamp(r; a, b) &= min(max(x, a), b) \\ s(a, b, n) &= \frac{b - a}{n - 1} \end{aligned}$$

$$q(r; a, b, n) = rnd(\frac{clamp(r; a, b) - a}{s(a, b, n)})s(a, b, n) + a \qquad (5)$$

where $r \in \mathbb{R}$ is number to be quantised, [a,b] is quantisation range, n is number of quantisation levels and $rnd()$ rounds to nearest integer

Figure from [Jac+18] (with permission)

Training Algorithm [Jac+18]

1. Create training graph of the floating-point model
2. Insert quantisation operations for integer computation in inference path using (5)
3. Train with quantised inference but floating-point backpropagation until convergence
4. Use weights thus obtained for low-precision inference

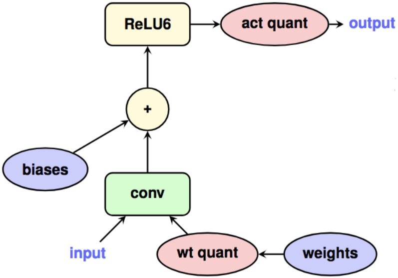


Figure from [Jac+18] (with permission)

Accuracy vs Precision [Jac+18]

ResNet50 on ImageNet, comparison with other approaches

Scheme	BWN	TWN	INQ	FGQ	Ours
Weight bits	1	2	5	2	8
Activation bits	float32	float32	float32	8	8
Accuracy	68.7%	72.5%	74.8%	70.8%	74.9%

Table 4.2: ResNet on ImageNet: Accuracy under various quantization schemes, including binary weight networks (BWN [21, 15]), ternary weight networks (TWN [21, 22]), incremental network quantization (INQ [33]) and fine-grained quantization (FGQ [26])

Figure from [Jac+18] (with permission)

THE UNIVERSITY OF SYDNEY

Accuracy vs Latency [Jac+18]

ImageNet classifier on Google Pixel 2 (Qualcomm Snapdragon 835 big cores)

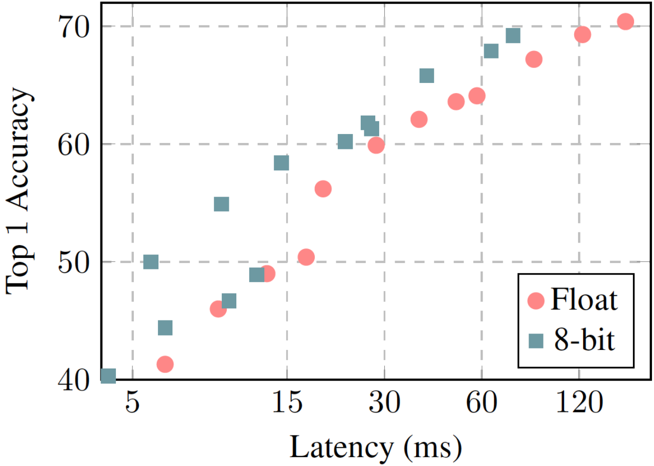


Figure from [Jac+18] (with permission)

Outline

THE UNIVERSITY OF SYDNEY

PyTorch

- PyTorch is a DNN training/inference environment
- QPyTorch is an excellent quantization library for PyTorch
  - Supports floating point, fixed point and block floating point

Headers

```python
# import useful modules
import argparse
import os
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
from qtorch.quant import Quantizer
from qtorch.optim import OptimLP
from torch.optim import SGD
from qtorch import BlockFloatingPoint, FloatingPoint, FixedPoint
from tqdm import tqdm
```

Time Execution

```
import time
start_time = time.time()
```

Create loaders for MNIST

```python
# loading data
ds = torchvision.datasets.MNIST
path = os.path.join("./data", "MNIST")
transform_train = transforms.Compose([
                            transforms.ToTensor(),
                            transforms.Normalize((0.1307,), (0.3081,))])
transform_test = transforms.Compose([
                            transforms.ToTensor(),
                            transforms.Normalize((0.1307,), (0.3081,))
                        ])
train_set = ds(path, train=True, download=True, transform=transform_train)
test_set = ds(path, train=False, download=True, transform=transform_test)
loaders = {
        'train': torch.utils.data.DataLoader(
            train_set,
            batch_size=64,
            shuffle=True,
            num_workers=1,
            pin_memory=True
        ),
        'test': torch.utils.data.DataLoader(
            test_set,
            batch_size=64,
            num_workers=1,
            pin_memory=True
        )
}
```

Define a neural network

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, 5, 1)
        self.conv2 = nn.Conv2d(20, 50, 5, 1)
        self.fc1 = nn.Linear(4*4*50, 500)
        self.fc2 = nn.Linear(500, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2, 2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2, 2)
        x = x.view(-1, 4*4*50)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
```

Create an Optimiser

```
use_cuda = False
device = torch.device("cuda" if use_cuda else "cpu")
model = Net().to(device)
optimizer = SGD(model.parameters(), lr=0.05, momentum=0.9, weight_decay=1e-4)
mxepochs = 5
```

Train for an epoch

```
def run_epoch(loader, model, criterion, optimizer=None, phase="train"):
    assert phase in ["train", "eval"], "invalid␣running␣phase"
    loss_sum = 0.0
    correct = 0.0

    if phase=="train": model.train()
    elif phase=="eval": model.eval()

    ttl = 0
    with torch.autograd.set_grad_enabled(phase=="train"):
        for i, (input, target) in tqdm(enumerate(loader), total=len(loader)):
            input = input.to(device=device)
            target = target.to(device=device)
            output = model(input)
            loss = criterion(output, target)
            loss_sum += loss.cpu().item() * input.size(0)
            pred = output.data.max(1, keepdim=True)[1]
            correct += pred.eq(target.data.view_as(pred)).sum()
            ttl += input.size()[0]

            if phase=="train":
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

    correct = correct.cpu().item()
    return {
        'loss': loss_sum / float(ttl),
        'accuracy': correct / float(ttl) * 100.0,
    }
```

Run Training

```
for epoch in range(mxepochs):
    fp_train_res = run_epoch(loaders['train'], model, F.cross_entropy,
                             optimizer=optimizer, phase="train")
    fp_test_res = run_epoch(loaders['test'], model, F.cross_entropy,
                             optimizer=optimizer, phase="eval")
    print('epoch', epoch)
    print(fp_train_res)
    print(fp_test_res)
```

Quantized Network

- Define the network. In the definition, we insert quantization module after every convolution layer. Note that the quantization of weight, gradient, momentum, and gradient accumulator are not handled here.

```
# let's define the model we are using
class lp_Net(nn.Module):
    def __init__(self, quant=None):
        super(lp_Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, 5, 1)
        self.conv2 = nn.Conv2d(20, 50, 5, 1)
        self.fc1 = nn.Linear(4*4*50, 500)
        self.fc2 = nn.Linear(500, 10)
        self.quant = quant()

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.quant(x)
        x = F.max_pool2d(x, 2, 2)
        x = self.quant(x)
        x = F.relu(self.conv2(x))
        x = self.quant(x)
        x = F.max_pool2d(x, 2, 2)
        x = self.quant(x)
        x = x.view(-1, 4*4*50)
        x = F.relu(self.fc1(x))
        x = self.quant(x)
        x = self.fc2(x)
        x = self.quant(x)
        return F.log_softmax(x, dim=1)
```

Quantization Types

- First we define a low and high precision format and give each part of the forward and backward passes a precision

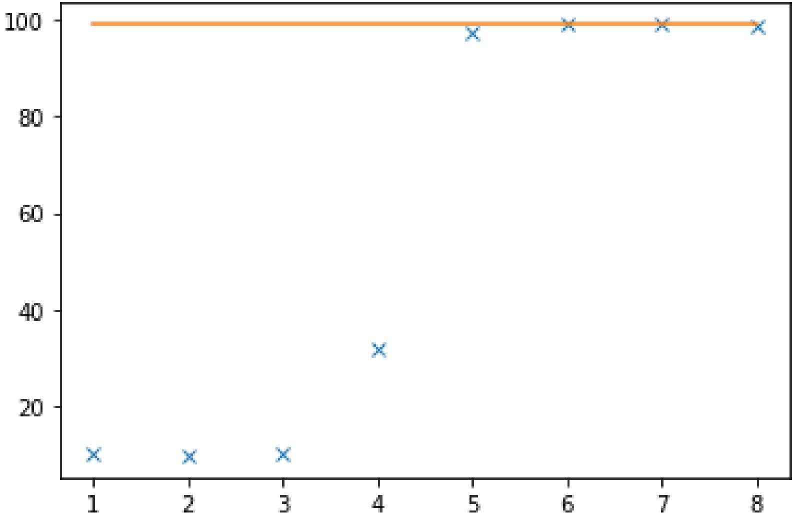
```
# define two floating point formats
lowp = FixedPoint(wl=8, fl=7)
highp = FloatingPoint(exp=8, man=7)  # this is bfloat16

# define quantization functions
weight_quant = Quantizer(forward_number=lowp, backward_number=None,
                        forward_rounding="nearest", backward_rounding="nearest")
grad_quant = Quantizer(forward_number=lowp, backward_number=None,
                        forward_rounding="nearest", backward_rounding="stochastic")
momentum_quant = Quantizer(forward_number=highp, backward_number=None,
                        forward_rounding="nearest", backward_rounding="stochastic")
acc_quant = Quantizer(forward_number=highp, backward_number=None,
                        forward_rounding="nearest", backward_rounding="nearest")

# define a lambda function so that the Quantizer module can be duplicated easily
act_error_quant = lambda : Quantizer(forward_number=lowp, backward_number=lowp,
                        forward_rounding="nearest", backward_rounding="nearest")
```

Quantized Training

- A low precision optimiser that uses the previously defined types

```
use_cuda = False
device = torch.device("cuda" if use_cuda else "cpu")
model = lp_Net(act_error_quant).to(device)
optimizer = SGD(model.parameters(), lr=0.05, momentum=0.9, weight_decay=1e-4)
lp_optimizer = OptimLP(optimizer,
                    weight_quant=weight_quant,
                    grad_quant=grad_quant,
                    momentum_quant=momentum_quant,
                    acc_quant=acc_quant
)
for epoch in range(mxepochs):
    train_res = run_epoch(loaders['train'], model, F.cross_entropy,
                          optimizer=lp_optimizer, phase="train")
    test_res = run_epoch(loaders['test'], model, F.cross_entropy,
                          optimizer=lp_optimizer, phase="eval")
    print('epoch', epoch)
    print(train_res)
    print(test_res)
```

Result



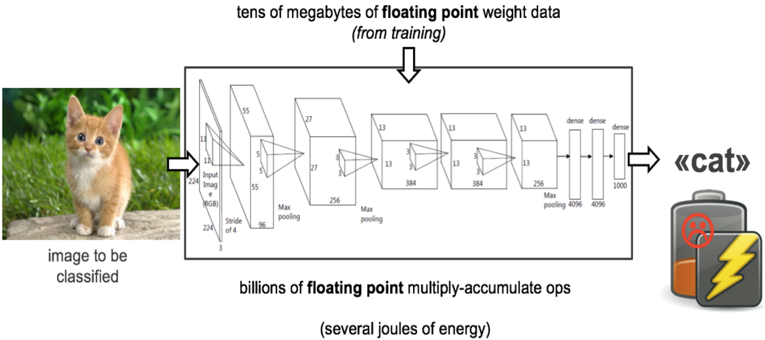
Outline

Outline

1. Introduction
2. Neural Networks
   - Integer Quantisation
   - QPyTorch
3. Implementation
   - FINN: A Binarised Neural Network
   - SYQ: Low Precision DNN Training
4. Tutorial

Inference with Convolutional Neural Networks

Slides from Yaman Umuroglu et. al., “FINN: A framework for fast, scalable binarized neural network inference,” FPGA’17

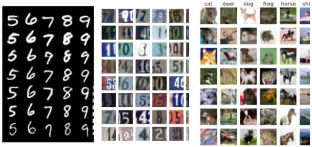


Slide © Copyright 2016 Xilinx

Binarized Neural Networks

› The extreme case of quantization
  - Permit only two values: +1 and -1
  - Binary weights, binary activations
  - Trained from scratch, not truncated FP

› Courbariaux and Hubara et al. (NIPS 2016)
  - Competitive results on three smaller benchmarks
  - Open source training flow
  - Standard “deep learning” layers
    - Convolutions, max pooling, batch norm, fully connected…



	MNIST	SVHN	CIFAR-10
Binary weights & activations	0.96%	2.53%	10.15%
FP weights & activations	0.94%	1.69%	7.62%
BNN accuracy loss	-0.2%	-0.84%	-2.53%

% classification error (lower is better)

Slide © Copyright 2016 Xilinx

Advantages of BNNs

Vivado HLS estimates on Xilinx UltraScale+ MPSoC ZU19EG

› *Much* smaller datapaths
  - Multiply becomes XNOR, addition becomes popcount
  - No DSPs needed, everything in LUTs
  - Lower cost per op = more ops every cycle

› *Much* smaller weights
  - Large networks can fit entirely into on-chip memory (OCM)
  - More bandwidth, less energy compared to off-chip

Precision	Peak TOPS	On-chip weights
1b	~66	~70 M
8b	~4	~10 M
16b	~1	~5 M
32b	~0.3	~2 M

200x 30x

› **fast** inference with **large BNNs**

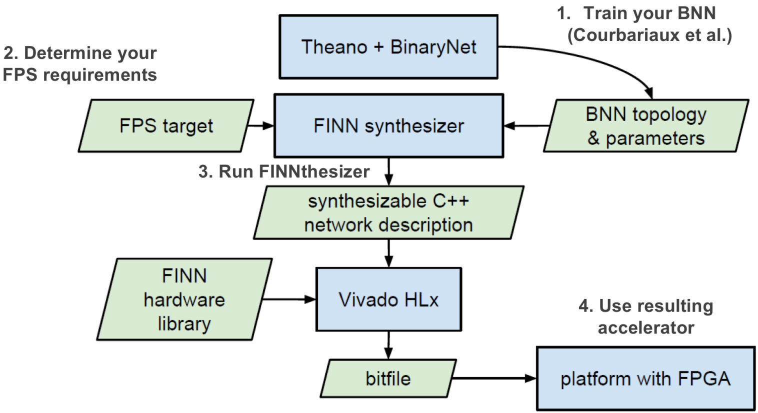
Slide © Copyright 2016 Xilinx

Design Flow

- One size does not fit all - Generate tailored hardware for network and use-case
- Stay on-chip - Higher energy efficiency and bandwidth
- Support portability and rapid exploration - Vivado HLS (High-Level Synthesis)
- Simplify with BNN-specific optimizations - Exploit compile time optimizations to simplify hardware, e.g. batchnorm and activation => thresholding

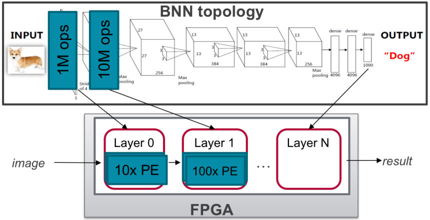
Slide © Copyright 2016 Xilinx

Design Flow



Slide © Copyright 2016 Xilinx

Heterogeneous Streaming Architecture



1x FPS

10x FPS

- **One hardware layer per BNN layer, parameters built into bitstream**
  – Both inter- and intra-layer parallelism
- **Heterogeneous: Avoid "one-size-fits-all" penalties**
  – Allocate compute resources according to FPS and network requirements
- **Streaming: Maximize throughput, minimize latency**
  – Overlapping computation and communication, batch size = 1

Slide © Copyright 2016 Xilinx

Matrix-Vector Threshold Unit (MVTU)

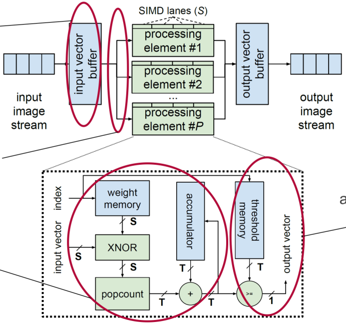
- **Core computational element of FINN, tiled matrix-vector multiply**
- **Computes a (P) row x (S) column chunk of matrix every cycle, per-layer configurable tile size**

buffer input for reuse (neuron folding)

broadcast S bits of input every cycle

multiply-accumulate, but with XNOR and popcount instead

apply thresholding and send to next layer



Slide © Copyright 2016 Xilinx

Convolutional Layers

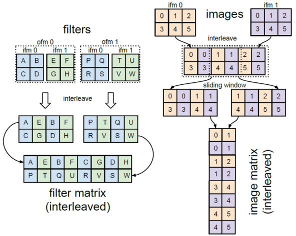
- **Lower convolutions to matrix-matrix multiplication, $W \cdot I$**
  - W : filter matrix (generated offline)
  - I: image matrix (generated on-the-fly)

- **Two components:**

convolution layer

slide window over image (generate I matrix)

matrix-matrix multiply, one vector at a time



Slide © Copyright 2016 Xilinx

Performance

		Accuracy	FPS	Power (chip)	Power (wall)	kFPS / Watt (chip)	kFPS / Watt (wall)	Precision
FINN	MNIST, SFC-max	95.8%	12.3 M	7.3 W	21.2 W	1693	583	1
FINN	MNIST, LFC-max	98.4%	1.5 M	8.8 W	22.6 W	177	269	1
FINN	CIFAR-10, CNV-max	80.1%	21.9 k	3.6 W	11.7 W	6	2	1
FINN	SVHN, CNV-max	94.9%	21.9 k	3.6 W	11.7 W	6	2	1
Prior Work	MNIST, Alemdar et al.	97.8%	255.1 k	0.3 W	-	806	-	2
Prior Work	CIFAR-10, TrueNorth	83.4%	1.2 k	0.2 W	-	6	-	1
Prior Work	SVHN, TrueNorth	96.7%	2.5 k	0.3 W	-	10	-	1

Max accuracy loss: ~3%

10 – 100x better performance

CIFAR-10/SVHN energy efficiency comparable to TrueNorth ASIC

Slide © Copyright 2016 Xilinx

THE UNIVERSITY OF SYDNEY

Outline

THE UNIVERSITY OF SYDNEY

Symmetric Quantisation (SYQ) [Far+18]

- To compute quantised weights from FP weights

$$\boldsymbol{Q}_l = sign(\boldsymbol{W}_l) \odot \boldsymbol{M}_l \tag{6}$$

with,

$$M_{l_{i,j}} = \begin{cases} 1 & \text{if} \quad \left| W_{l_{i,j}} \right| \geq \eta_l \\ 0 & \text{if} \quad -\eta_l < W_{l_{i,j}} < \eta_l \end{cases} \tag{7}$$

$$sign(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ \text{-1} & \text{otherwise} \end{cases} \tag{8}$$

where $\boldsymbol{M}$ represents a masking matrix, η is the quantization threshold hyperparameter (0 for binarised)

Scaling Factors

- To recover the dynamic range of each layer

$$\boldsymbol{Q}_l = \alpha_l sign(\boldsymbol{W}_l) \odot \boldsymbol{M}_l \tag{9}$$

- Further accuracy by using a vector $\boldsymbol{\alpha_l}$ for fine-grained scaling
- However, to ensure hardware amenability we need ordered scaling

Scaling Factors ctd..

$$w = \begin{bmatrix} 0.8 & 0.2 & 0.1 & -0.4 \\ -0.4 & -0.2 & 0.1 & 0.9 \\ 0.3 & -0.4 & 0.2 & 0.4 \end{bmatrix}$$

$$=> sign(w) = \begin{bmatrix} 1.0 & 1.0 & 1.0 & -1.0 \\ -1.0 & -1.0 & 1.0 & 1.0 \\ 1.0 & -1.0 & 1.0 & 1.0 \end{bmatrix}$$

- total quantization error = 7.6

THE UNIVERSITY OF SYDNEY

Scaling Factors ctd..

$$w = \begin{bmatrix} 0.8 & 0.2 & 0.1 & -0.4 \\ -0.4 & -0.2 & 0.1 & 0.9 \\ 0.3 & -0.4 & 0.2 & 0.4 \end{bmatrix}$$

$$\alpha = \|w\| = 0.37$$

$$=> \alpha * sign(w) = \begin{bmatrix} 0.37 & 0.37 & 0.37 & -0.37 \\ -0.37 & -0.37 & 0.37 & 0.37 \\ 0.37 & -0.37 & 0.37 & 0.37 \end{bmatrix}$$

- total quantization error = 2.2

Scaling Factors ctd..

$$w = \begin{bmatrix} 0.8 & 0.2 & 0.1 & -0.4 \\ -0.4 & -0.2 & 0.1 & 0.9 \\ 0.3 & -0.4 & 0.2 & 0.4 \end{bmatrix}$$

$$\boldsymbol{\alpha} = \begin{bmatrix} \|w_1\| \\ \|w_2\| \\ \|w_3\| \end{bmatrix} = \begin{bmatrix} 0.375 \\ 0.4 \\ 0.325 \end{bmatrix}$$

$$=> \boldsymbol{\alpha} * sign(w) = \begin{bmatrix} 0.375 & 0.375 & 0.375 & -0.375 \\ -0.4 & -0.4 & 0.4 & 0.4 \\ 0.325 & -0.325 & 0.325 & 0.325 \end{bmatrix}$$

- total quantization error = 1.925

Symmetric Quantisation (SYQ) [Far+18]

- Make approximation $W_l \approx \alpha_l Q_l$, $Q_l \in \mathcal{C}$
- $\mathcal{C}$ is the codebook, $\mathcal{C} \in \{C_1, C_2, \ldots\}$ e.g. $\mathcal{C} = \{-1, +1\}$ for binary, $\mathcal{C} = \{-1, 0, +1\}$ for ternary
- A diagonal matrix $\boldsymbol{\alpha_l}$ is defined by the vector $\boldsymbol{\alpha_l} = \left[\alpha_l^1, ..., \alpha_l^m\right]$:

$$\boldsymbol{\alpha} = diag(\boldsymbol{\alpha}) := \begin{bmatrix} \alpha^1 & 0 & .. & 0 & 0 \\ 0 & \alpha^2 & .. & : & 0 \\ : & : & .. & \alpha^{m-1} & : \\ 0 & 0 & .. & 0 & \alpha^m \end{bmatrix}$$

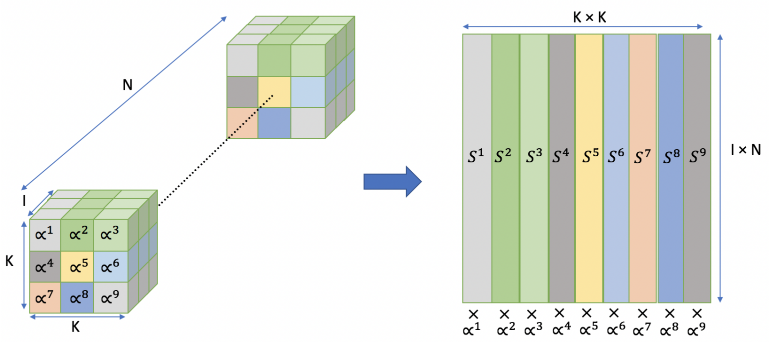
- Train by solving

$$\alpha_l^* = argmin_\alpha E(\alpha, \boldsymbol{Q}) \quad s.t. \quad \alpha \geq 0,\ Q_{l_{i,j}} \in \mathcal{C}$$

THE UNIVERSITY OF SYDNEY

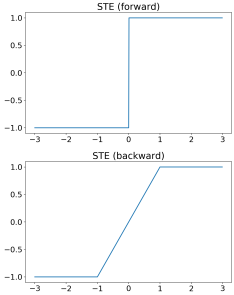
Subgroups

- Finer-grained quantisation improves weight approximation
- Pixel-wise shown, layer-wise has similar accuracy



Dealing with Non-differentiable Functions

- Recall (6) $\boldsymbol{Q}_l = sign(\boldsymbol{W}_l) \odot \boldsymbol{M}_l$
- This step function has a derivative which is zero everywhere: *vanishing gradients* problem
- Address via a straight through estimator (STE)
- Consider $q = sign(r)$ and $g_r \approx \frac{\partial C}{\partial q}$ then $\frac{\partial C}{\partial r} \approx g_q 1_{|r| \leq 1}$



SYQ Forward

- Quantise the inference step

Initialize: Set subgrouping granularity for S_l^i and set $\alpha_{l_0}^i$.
Inputs: Minibatch of inputs & targets (I, Y), Error function $E(Y, \hat{Y})$, current weights $\boldsymbol{W_t}$ and learning rate, γ_t
Outputs: Updated $\boldsymbol{W_{t+1}}$, $\boldsymbol{\alpha_{t+1}}$ and γ_{t+1}

SYQ Forward:
for l=2 to L-1 **do**
    $\boldsymbol{Q}_l = sign(\boldsymbol{W}_l) \odot \boldsymbol{M}_l$ **with** η

    for ith subgroup in lth layer **do**
        Apply α_l^i to S_l^i
    end for
end for
$\hat{Y}$ = **SYQForward** $(I, Y, \boldsymbol{Q}_l, \boldsymbol{\alpha}_l)$

SYQ Backward

- Do the backprop step with FP32 weights
- Compute $\boldsymbol{W}$ and $\boldsymbol{\alpha}$

SYQ Backward:

$\frac{\partial \hat{E}}{\partial \boldsymbol{Q}_l} = \textbf{WeightBackward}(\boldsymbol{Q}_l, \boldsymbol{\alpha}_l, \frac{\partial \hat{E}}{\partial \hat{Y}})$

$\frac{\partial \hat{E}}{\partial \boldsymbol{\alpha}_l} = \textbf{ScalarBackward}(\frac{\partial \hat{E}}{\partial \boldsymbol{Q}_l}, \boldsymbol{\alpha}_l, \frac{\partial \hat{E}}{\partial \hat{Y}})$

$\boldsymbol{W_{t+1}} = \textbf{UpdateWeights}(\boldsymbol{W_t}, \frac{\partial \hat{E}}{\partial \boldsymbol{Q}_l}, \gamma)$

$\boldsymbol{\alpha_{t+1}} = \textbf{UpdateScalars}(\boldsymbol{\alpha_t}, \frac{\partial \hat{E}}{\partial \boldsymbol{\alpha}_l}, \gamma)$

$\gamma_{t+1} = \textbf{UpdateLearningRate}(\gamma_t, t)$

Results for 8-bit activations

Model		Bin	Tern	FP32	Reference
AlexNet	Top-1	**56.6**	**58.1**	56.6	57.1
	Top-5	**79.4**	**80.8**	80.2	80.2
VGG	Top-1	**66.2**	**68.7**	69.4	-
	Top-5	**87.0**	**88.5**	89.1	-
ResNet-18	Top-1	**62.9**	**67.7**	69.1	69.6
	Top-5	**84.6**	**87.8**	89.0	89.2
ResNet-34	Top-1	**67.0**	**70.8**	71.3	73.3
	Top-5	**87.6**	**89.8**	89.1	91.3
ResNet-50	Top-1	**70.6**	**72.3**	76.0	76.0
	Top-5	**89.6**	**90.9**	93.0	93.0

- Our ResNet and AlexNet reference results are obtained from https://github.com/facebook/fb.resnet.torch and https://github.com/BVLC/caffe

Alexnet Comparison

Model	Weights	Act.	Top-1	Top-5
DoReFa-Net [Zho+16]	1	2	49.8	-
QNN [Hub+16]	1	2	51.0	73.7
HWGQ [Cai+17]	1	2	52.7	76.3
SYQ	**1**	**2**	**55.2**	**78.4**
DoReFa-Net [Zho+16]	1	4	53.0	-
SYQ	**1**	**4**	**56.2**	**79.4**
BWN [Ras+16]	1	32	56.8	79.4
SYQ	**1**	**8**	**56.6**	**79.4**
SYQ	**2**	**2**	**55.7**	**79.1**
FGQ [Mel+17]	2	8	49.04	-
TTQ [Zhu+16]	2	32	57.5	79.7
SYQ	**2**	**8**	**58.1**	**80.8**

Summary

- Reducing precision
  - Significantly reduce computational costs in DNNs
  - Data may now fit entirely on chip, avoiding external memory accesses
  - Computations greatly simplified
  - Key dimension for optimisation in CPU/GPU/FPGA implementations
- Convolutional layer can be computed as a MM
- Still an active research topic

Outline

Installation

- To load:

```
docker load -i nova-quant-docker.tar
```

- To run:

```
docker run -it -p8888:8888 \
pytorch-jupyter-docker_jupytertorch:latest bash
```

- Then:

```
git clone https://github.com/phwl/nova-quant
cd nova-quant
jupyter notebook --allow-root --no-browser --ip='*'
```

- On your browser:

```
http://127.0.0.1:8888/?token=1b14e...
```

Questions

1. Follow the instructions in the following jupyter notebooks to complete the tutorial.
2. `sround.ipynb` an implementation of stochastic rounding (slide 12).
3. Read the QPyTorch Functionality Overview at `https://github.com/Tiiiger/QPyTorch/blob/master/examples/tutorial/Functionality_Overview.ipynb`
4. `mnist.ipynb` training of a low-precision MNIST inference network.

References I

[TL19] Mingxing Tan et al. "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks". In: *arXiv e-prints*, arXiv:1905.11946 (May 2019), arXiv:1905.11946. arXiv: `1905.11946 [cs.LG]`.

[Din+17] Caiwen Ding et al. "CirCNN: Accelerating and Compressing Deep Neural Networks Using Block-circulant Weight Matrices". In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-50 '17. Cambridge, Massachusetts: ACM, 2017, pp. 395–408. ISBN: 978-1-4503-4952-9. DOI: `10.1145/3123939.3124552`. URL: `http://doi.acm.org/10.1145/3123939.3124552`.

References II

[HH18] Catherine F. Higham et al. "Deep Learning: An Introduction for Applied Mathematicians". In: *arXiv e-prints*, arXiv:1801.05894 (Jan. 2018), arXiv:1801.05894. arXiv: `1801.05894 [math.HO]`.

[CPS06] Kumar Chellapilla et al. "High Performance Convolutional Neural Networks for Document Processing". In: *Tenth International Workshop on Frontiers in Handwriting Recognition*. Ed. by Guy Lorette. http://www.suvisoft.com. Université de Rennes 1. La Baule (France): Suvisoft, Oct. 2006. URL: `https://hal.inria.fr/inria-00112631`.

THE UNIVERSITY OF SYDNEY

References III

[Jac+18] Benoit Jacob et al. "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2018.

[Far+18] Julian Faraone et al. "SYQ: Learning Symmetric Quantization For Efficient Deep Neural Networks". In: *Proc. Computer Vision and Pattern Recognition (CVPR)*. Utah, US, June 2018.

[Zho+16] Shuchang Zhou et al. "DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients". In: *CoRR* abs/1606.06160 (2016). URL: http://arxiv.org/abs/1606.06160.

THE UNIVERSITY OF SYDNEY

References IV

[Hub+16] Itay Hubara et al. "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations". In: *CoRR* abs/1609.07061 (2016). arXiv: `1609.07061`. URL: `http://arxiv.org/abs/1609.07061`.

[Cai+17] Zhaowei Cai et al. "Deep Learning with Low Precision by Half-wave Gaussian Quantization". In: *CoRR* abs/1702.00953 (2017). arXiv: `1702.00953`. URL: `http://arxiv.org/abs/1702.00953`.

[Ras+16] Mohammad Rastegari et al. "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks". In: *CoRR* abs/1603.05279 (2016). URL: `http://arxiv.org/abs/1603.05279`.

References V

[Mel+17] Naveen Mellempudi et al. "Ternary Neural Networks with Fine-Grained Quantization". In: *CoRR* abs/1705.01462 (2017). URL: http://arxiv.org/abs/1705.01462.

[Zhu+16] Chenzhuo Zhu et al. "Trained Ternary Quantization". In: *CoRR* abs/1612.01064 (2016). URL: http://arxiv.org/abs/1612.01064.

THE UNIVERSITY OF SYDNEY