Training Quantized Neural Networks
Nick Fraser, Giulio Gambardella, Michaela Blott, Thomas Preusser
Xilinx Research, Ireland
Xilinx Research - Ireland

- Part of the CTO organization
  - 9 (out of 35 worldwide) researchers
- With a very active internship program
  - 6-10 students & visiting scholars
- Visiting professors on sabbatical
- Postdoc on Marie-Curie Fellowship

IDA Ireland
Demonstrated to work well for numerous use cases
Neural Networks

- NNs are the predominant AI algorithm
  - Can outperform humans and traditional CV algorithms for image recognition

- NNs have the theoretical property of being a “universal approximation function”
  - Empirically outperforming other approximator functions

Increasing adoption: replacing other solutions and for previously unsolved problems
Neural Networks: Training vs Inference

**Training**
Process for a machine to learn by optimizing models (weights) from data.

- Requires little expertise/specialization in the actual target domain.

**Inference**
Using trained models to predict or estimate outcomes from new observations.
Challenges: Wide & Increasing Range of Applications

- Translation Service
- AlphaGo
- ADAS
- 3D Reconstruction from Drone Images
- Medical Diagnoses
- Real-Time, Sensor-Based Control
- IBM Watson Health Assistance
- Recommender Systems
- Hearing Aids
Challenges: Different Figures of Merits

- **Accuracy requirements vary with applications**: Recommender systems, data analytics vs ADAS.

- **Reduced latency**: Results in a better user experience in cloud-based systems (Google defines 7ms) and vital for robotics.

- **Real-time systems**: Have clearly defined throughput and latency constraints.

- **Embedded Systems**: Heavily power constrained

  Data Centers: $\text{OPEX} = f(\text{energy})$
The predominant CNN computation is linear algebra

- Demands lots of (simple) computation and lots of parameters (memory)
  - AlexNet: 244 MB & 1.5 GOPS, VGG16: 552 MB & 30.8 GOPS; GoogleNet: 41.9 MB & 3.0 GOPS for ImageNet

Challenges:

Highly Compute and Memory Intensive

Challenge 2:

billions of multiply-accumulate ops & tens of megabytes of parameter data
Challenges: Neural Networks Will Continue to Change

- Number and types of layers are changing
- Data representations and quantization methods are changing
- Graph Connectivity is changing
- Continuous stream of new algorithms

- AlexNet (2012)
- GoogleNet (2014)
- DenseNet (2016)
Challenge: Multidimensional Design Space

- Each combination yields a different point in the design space: error, cost, throughput, latency, power

ML task (growing list)
- Vision: image classification, recognition, segmentation, SLAM
- Audio: voice control, noise filtering
- Gaming strategy
- Recommender systems

Neural Network topology
- AlexNet, GoogleNet, ResNet-X, Enet, Yolo

Training data sets
- ImageNet, COCO, VOC, GTSRB, MNIST, CIFAR, GSVHN, City KITTY

Training framework
- Mxnet, Caffe, Tensorflow, Theano, Darknet

Compute & data types
- Numerical representation & precision, quantization functions, activation functions, non-linearity

Hyperparameters
- Learning rates, regularization scheme, cost function, optimization scheme, pre/post processing

Architecture
- Dataflow, systolic array, compression engines, sparse representations

End Device

Trained Neural Network
Opportunity: Customized Neural Networks

Design and training of FPGA-friendly neural networks that provide end-solutions that are high-performance and more power-efficient than any other hardware

- Hardware cost, power, performance, latency
Opportunity: Customized ML Processor Datapath

Latency vs generality

Generality vs Performance
Latency vs Resources

Weights, Thresholds
IFM buffers

generic template architecture
Memory subsystem

Weights, Thresholds
**Focus**: Reduced Precision - Quantization

- Cost per operation is greatly reduced
- Memory cost is greatly reduced
  - Large networks can fit entirely into on-chip memory (OCM) (UltraRAM, BRAM)

- Today’s FPGAs have a much higher peak performance for reduced precision operations

<table>
<thead>
<tr>
<th>Precision</th>
<th>Cost per Op LUT</th>
<th>Cost per Op DSP</th>
<th>MB needed (AlexNet)</th>
<th>TOps/s (KU115)*</th>
<th>TOps/s (VU9P)**</th>
<th>TOps/s (ZU19EG)*</th>
</tr>
</thead>
<tbody>
<tr>
<td>1b</td>
<td>2.5</td>
<td>0</td>
<td>7.6</td>
<td>~46</td>
<td>~100</td>
<td>~66</td>
</tr>
<tr>
<td>4b</td>
<td>16</td>
<td>0</td>
<td>30.5</td>
<td>~11</td>
<td>~15</td>
<td>~16</td>
</tr>
<tr>
<td>8b</td>
<td>45</td>
<td>0</td>
<td>61</td>
<td>~3</td>
<td>~6</td>
<td>~4</td>
</tr>
<tr>
<td>16b</td>
<td>15</td>
<td>0.5</td>
<td>122</td>
<td>~1</td>
<td>~4</td>
<td>~1</td>
</tr>
<tr>
<td>32b</td>
<td>178</td>
<td>2</td>
<td>244</td>
<td>~0.5</td>
<td>~1</td>
<td>~0.3</td>
</tr>
</tbody>
</table>

*Assumptions: Application can fill device to 70% (fully parallelizable) 250MHZ
**Assumptions: Application can fill device to 70% (fully parallelizable) 300MHZ
Quantizing and Fixed Point saves Power

<table>
<thead>
<tr>
<th>Operation</th>
<th>Energy (pJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8b Add</td>
<td>0.03</td>
</tr>
<tr>
<td>16b Add</td>
<td>0.05</td>
</tr>
<tr>
<td>32b Add</td>
<td>0.1</td>
</tr>
<tr>
<td>16b FP Add</td>
<td>0.4</td>
</tr>
<tr>
<td>32b FP Add</td>
<td>0.9</td>
</tr>
<tr>
<td>8b Mult</td>
<td>0.2</td>
</tr>
<tr>
<td>32b Mult</td>
<td>3.1</td>
</tr>
<tr>
<td>16b FP Mult</td>
<td>1.1</td>
</tr>
<tr>
<td>32b FP Mult</td>
<td>3.7</td>
</tr>
<tr>
<td>32b SRAM Read (8KB)</td>
<td>5</td>
</tr>
<tr>
<td>32b DRAM Read</td>
<td>640</td>
</tr>
</tbody>
</table>

Source: Bill Dally (Stanford), Cadence Embedded Neural Network Summit, February 1, 2017
Do we loose Accuracy?
Compensating Quantization with Network Complexity

- Just reducing precision, reduce hardware cost & increases error
- Recuperate accuracy by retraining & increasing network size
- 1b, 2b and 4b provide pareto optimal solutions

Intel: Wide Reduced Precision Networks
Accuracy of Quantized Neural Networks (QNNs) Improving Published Results for FP CNNs, QNNs and binarized NNs (BNNs)

- Accuracy results are improving rapidly through for example new training techniques, topological changes and other methods
Quantized Neural Networks provide the opportunity to create hardware implementations that are faster, smaller, or more power efficient.
Agenda

- **Introduction to Neural Networks:**
  - Neural network layers
  - The backpropagation algorithm

- **Quantized Neural Networks**
  - Data representations
  - Binarized Neural Networks
  - Quantization-aware backpropagation

- **Training Binary Neural Networks in Lasagne**
Neural Networks: A Quick Introduction
Neural networks are computational graphs constructed from one or more layers.

Layers: Usually linear operations followed by a non-linear activation function

- Dot product = fully connected layer
- 2D convolution = convolutional layer

Other common layers:
- Pooling layers (Max / Average)
- Batch normalization
Neural Networks – Fully Connected Layer

- Also known as: inner product layer or dense layer.
- Each neuron is connected to every neuron of the previous layer.
- A weight is associated with each “synapse”.
- Can be written as a matrix-vector product with an element-wise non-linearity applied afterwards.

\[ W_i \times a_i = a_{i+1} \]

Act_func(•)

\[ W_i \cdot a_i \]
Each neuron applies a convolution to all images in the previous layer.

Weights represent the filters used for convolutions.

Can be *lowered* to a matrix-matrix multiply.

Non-linear activation applied to each output pixel.

Source: http://cs231n.github.io/assets/conv-demo/index.html
Neural Networks – Activation Functions

➢ Most popular: the rectified linear unit (ReLU)

➢ Other common ones include: tanh, leaky ReLU.

➢ For binarized neural networks, the step function is often used.

Neural Networks – Pooling Layer

- Crude downsamplers of images.
- Reduces compute in subsequent layers.
- Max pooling takes the maximum value from a window of pixels.
- Average pooling is another common type.

Source: http://cs231n.github.io/convolutional-networks/
Batch Normalization Layer

- Normalizes the statistics of activation values of particular neurons.
- Adds post-scaling to allow some neurons to be “more important” than others.
- Significantly reduces the training time of networks.
- Can improve the accuracy.
Training Neural Networks - Backpropagation

- **Purpose:** calculate the gradients associated with each weight within a network.

- **Forward path is the same as inference.**

- **Gradients calculated from a semi-differentiable loss function.**

- **Gradients passed back and transformed layer-by-layer.**

- **Weights updated from the provided gradients, input activations and an optimization algorithm.**
Backpropagation: Forward Path

➡️ Same as Inference:

\[ W_i \times a_i = W_i \cdot a_i \xrightarrow{\text{Act_func(•)}} a_{i+1} \]
Backpropagation: Backward Path

- Pass gradients back through network:

\[ W_i^T \times g_{i+1} = W_i^T \cdot g_{i+1} \]
Backpropagation: Weight Update

Typically with an optimized weight update:

– *Stochastic* gradient descent.
– Adam.

\[ W_i^+ := W_i + a_i \times g_{i+1}^T \]
Quantized Neural Networks
Data Representations & Reduced Precision

- **Floating Point**
  - Usually 32-bits
  - Large range, high precision

- **Fixed Point**
  - Fixed range
  - Simpler hardware

- **Binarized**
  - Multiply-accumulate becomes XNOR-popcount
  - 32x memory reduction
  - Extreme performance possible on FPGAs
Key Training Challenges When Reducing Precision

➢ Training must be aware of quantization
  • Direct quantization from FP -> RP tends to ruin accuracy when going below 8 bits.

➢ How to pass gradients through quantized activation functions?

Quantization-Aware Forward Path

- On-the-fly quantization of weights
- Quantizing activation function

\[ W_i \]

\[ Q(W_i) \times Q(a_i) = Q(W_i) \cdot Q(a_i) \]

\[ a_{i+1} \]

Quantization-Aware Forward Path

- On-the-fly quantization of weights
- Quantizing activation function
Quantization-Aware Backpropagation

- Non-quantized gradients
- Backpropagation based on quantized weights

\[ W_i^T \]

\[ Q(W_i^T) \]

\[ \times \]

\[ g_{i+1} \]

\[ = \]

\[ Q(W_i^T) \cdot g_{i+1} \]

\[ \text{Act}_\text{func}'(\cdot) \]

\[ g_i \]
Quantization-Aware Weight Update

» Update real weights

\[ W_i^+ := W_i + Q(a_i) \times g_{i+1}^T \]
Differentiating the sign function:

- Choose an activation function, \( a \), which tends towards \( \pm 1 \) as \( x \) tends towards \( \pm \infty \).
  (The hard hyperbolic tangent function is a common, nice choice)

- Create a quantized activation function as the composition \( a \circ Q : x \mapsto Q(a(x)) \).

- For the purpose of differentiation, pretend that the quantization function \( Q \) had a gradient of 1 everywhere.

Clip gradients outside of range (optional, but recommended).
Quantizing ReLU

- Clip ReLU at the maximum value you want to support.
- Create a quantized activation function as the composition \( a^{\circ}Q : x \mapsto Q(a(x)) \).
  - Equal distance quantization over the specified range is a good choice and ensures a local average gradient of 1.
- For the purpose of differentiation, pretend that the quantization function \( Q \) had a gradient of 1 everywhere.

Clip gradients outside of range (optional, but recommended).
Batch Normalization

- Improves convergence time, and accuracy of RPNNs.

- **Fixed** post-scaling gives full control over output distribution parameters, e.g.:
  \[
  \gamma = 1, \quad \beta = 0 \quad \text{for} \quad \mu = 0, \quad \sigma_B^2 = 1
  \]

- For extreme reduced precision, BN is free at inference time.

- For higher precisions, shift-based BN can be used.

---

**Input:** Values of \(x\) over a mini-batch: \(B = \{x_1, \ldots, x_m\}\);
Parameters to be learned: \(\gamma, \beta\)

**Output:** \(\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}\)

\[
\begin{align*}
  \mu_B & \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i \quad \text{// mini-batch mean} \\
  \sigma_B^2 & \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2 \quad \text{// mini-batch variance} \\
  \hat{x}_i & \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \quad \text{// normalize} \\
  y_i & \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \quad \text{// scale and shift}
\end{align*}
\]

QNNs In Lasagne
Frameworks with Reduced Precision Training Support

- **Lasagne (Theano)**
  - Supports binarized weights / activations
  - Extended to support fixed-point data types

- **Tensorpack (TensorFlow)**
  - Supports reduced-precision weights / activations

- **Caffe**
  - C++ framework
  - Supports binarized weights / activations
  - Supports uniform and non-uniform quantization

- **Darknet**
  - C-based NN library
  - Supports binarized weights / activations

- **Torch**
  - Lua based
  - Supports binarized weights / activations
  - Supports shift-based Adam / batch normalization

- **MXNet**
  - Supports binarized weights / activations

Popularity of reduced precision neural networks growing – support in other frameworks will probably arrive soon!
Features of Lasagne

➢ **Python interface**
  – Easy integration with Numpy.

➢ **Automatic Differentiation**
  – Less code = fewer bugs!

➢ **CPU / GPU support**
  – Switch between CPU / GPU by simply setting an environment variable.

➢ **Extreme Flexibility**
  – Can implement any dataflow graph as a neural network.
Full Installation Instructions Available on Github

Custom reduced precision layers and trainers

Topology definitions

Training scripts

Weight packing

Source: https://github.com/Xilinx/BNN-PYNQ
Test Networks

**LFC**
- Input images: 28x28 pixels, binarized images
- Number of layers: 3 FC layers, 1024 neurons each
- Compute requirement: 5.8 MOps/Frame

**CNV (VGG-16 derivative)**
- Input images: 32x32 pixels, RGB image
- Number of layers: 2 (3x3) Conv + Max Pool + 2 (3x3) Conv + Max Pool + 2 Convolutional + Max Pool + 3 FC
- Compute requirement: 1.23 GOPs/Frame
BinaryNet in Lasagne – Training Script (mnist.py)

- ~150 lines of code
- Python library imports
- Setting hyperparameters
- Importing dataset
- Constructing the topology
  → Changes require bitstream update.
- Setting the loss function / network output
- Training the network
BinaryNet in Lasagne – Importing the Dataset

➤ Import sets and separate into training, validation and test sets – these are simply numpy arrays!
  – Rule of thumb: 60% training, 20% validation, 20% test.
  – Beware of duplicates and data order.

➤ Binarize input values (only required for LFC)

➤ Convert labels into a 1D array of class indices

➤ 1-hot encode the class labels

➤ Modify result to match loss function

```python
print('Loading MNIST dataset...')

train_set = MNIST(which_set='train', start=0, stop=50000, center=False)
valid_set = MNIST(which_set='train', start=50000, stop=60000, center=False)
test_set = MNIST(which_set='test', center=False)

# bc01 format
# Inputs in the range [-1,+1]
# print("Inputs in the range [-1,+1]")
train_set.X = 2*train_set.X.reshape(-1, 1, 28, 28) - 1.
valid_set.X = 2*valid_set.X.reshape(-1, 1, 28, 28) - 1.
test_set.X = 2*test_set.X.reshape(-1, 1, 28, 28) - 1.

# Binarise the inputs.
train_set.X = np.where(train_set.X < 0, -1, 1).astype(theano.config.floatX)
valid_set.X = np.where(valid_set.X < 0, -1, 1).astype(theano.config.floatX)
test_set.X = np.where(test_set.X < 0, -1, 1).astype(theano.config.floatX)

# flatten targets
train_set.y = np.hstack(train_set.y)
valid_set.y = np.hstack(valid_set.y)
test_set.y = np.hstack(test_set.y)

# Onehot the targets
train_set.y = np.float32(np.eye(10)[train_set.y])
valid_set.y = np.float32(np.eye(10)[valid_set.y])
test_set.y = np.float32(np.eye(10)[test_set.y])

# for hinge loss
train_set.y = 2*train_set.y - 1.
valid_set.y = 2*valid_set.y - 1.
test_set.y = 2*test_set.y - 1.
```
~60 lines of code

Configure global parameters

Construct the topology

Modifying the code here will mean the weights may not work with the overlay!!
BinaryNet in Lasagne – Defining Layers

▷ Basic layer pattern: Dense (or Conv2D) -> BatchNorm -> Activation -> Dropout (optional)

▷ Instantiate a layer with binary weights

▷ Binarize activations

▷ Modifying the code here will mean the weights may not work with the overlay!
# Accuracy of Binary and Almost Binary Networks

## Published Results

<table>
<thead>
<tr>
<th>Dataset</th>
<th>FP32</th>
<th>BNN</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST</td>
<td>99%</td>
<td>99%</td>
<td>[1]</td>
</tr>
<tr>
<td>SVHN</td>
<td>98%</td>
<td>97%</td>
<td>[1]</td>
</tr>
<tr>
<td>CIFAR-10</td>
<td>92%</td>
<td>90%</td>
<td>[1]</td>
</tr>
<tr>
<td>ImageNet (AlexNet arch)</td>
<td>80% top-5</td>
<td>69% top-5</td>
<td>[2]</td>
</tr>
<tr>
<td>ImageNet (ResNet-18 arch)</td>
<td>89% top-5</td>
<td>73% top-5</td>
<td>[2]</td>
</tr>
<tr>
<td>ImageNet (GoogleNet arch)</td>
<td>90% top-5</td>
<td>86% top-5</td>
<td>[2]</td>
</tr>
<tr>
<td>ImageNet (DoReFaNet)</td>
<td>56% top-1</td>
<td>50% top-1</td>
<td>[4] 2b activations</td>
</tr>
</tbody>
</table>

- Similar accuracy on small networks and promising results for larger networks

Quantizing networks from floating point to binary will introduce a drop in accuracy.

Sometimes conversion of an existing network will “just work”.

Often, hyperparameters or even the network topology will have to change to get good accuracy results.

Common methods to improve accuracy:
- Add batch normalization before activations.
- Reduce learning rate.
- Increase number of epochs.
- Increase the size of the network:
  - Larger layers,
  - Deeper network (more layers).
Combining quantized neural networks & FPGAs allows opportunities to create extreme high-throughput, low-power neural networks.

There is some drop in accuracy compared to floating point accuracy. This is typically compensated by re-training and increasing the size of the network.

Pynq + Lasagne – great platforms to get started training and implementing your own high-performance neural networks.
Hands-On Opportunities

» GPU support for training helps a lot, AWS EC2 might help out.

» Checkout open-source QNN examples with trained models and Jupyter notebooks for Pynq-Z1 at [http://www.pynq.io/community.html](http://www.pynq.io/community.html):
  
  – Xilinx/BNN-PYNQ
    - LFC, CNV: CIFAR10, MNIST, Road Signs, …
  – Xilinx/QNN-MO-PYNQ
    - TinierYolo, DorefaNet: Object Detection, ImageNet Classification
  – tukl-msd/LSTM-PYNQ
    - LSTM: OCR for Fraktur text

» Expect the QNN story to unfold for more platforms:
  
  – Support for more boards.
  – AWS F1 solution.

» See the XILINX booth!
Thank You.