Implementation

Arthur J. Redfern
axr180074@utdallas.edu
Oct 24, 2018

Disclaimer

- Previous math lectures
  - A broad presentation of material (linear algebra, calculus, probability and algorithms)
  - Perhaps a little biasing from me in terms of presentation, importance and intuition
  - But in general the material stands on it's own and there's not ambiguity at our level of review

- Previous xNN lectures
  - A broad presentation of material (design and training)
  - A little more biasing from me in terms of presentation, importance and intuition
  - But pointers to many many references were provided for you to go deeper on any topic

- This series of implementation lectures
  - We'll still cover a lot of topics, but the presentation of material will be more narrow
  - You'll get more of my opinion in terms of the right way to do things
  - But pointers will still be given to additional references and you're free to draw your own differing conclusions

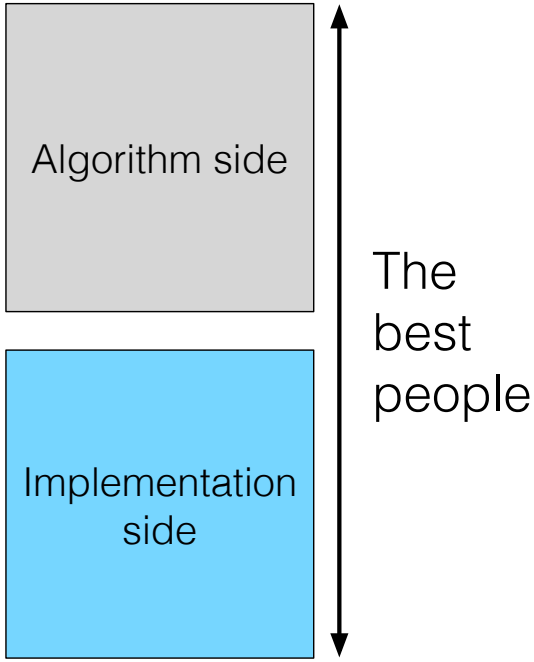
Outline

- Motivation
- Networks
- Hardware
- Software

Motivation

Why Discuss Implementation

- Progress in xNNs is directly linked to progress in improved implementations
- At large companies with big ML related business
  - About 1/2 the people are on the algorithm side
  - About 1/2 the people are on the implementation side
- The best people understand both
  - I want you to understand both



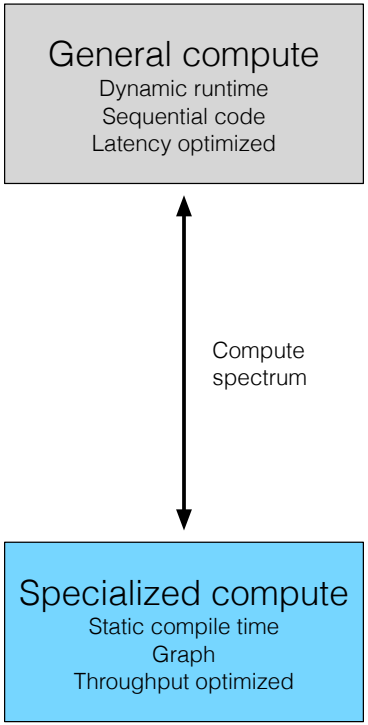
The Future Of Hardware And Software

Yes, that's a slightly grandiose slide title / slight exaggeration; no, it's not that far from the truth

- Is a bifurcation where only 2 points matter
  - Big code, small general compute
  - Small code, big specialized compute

- Big code, small general compute → map to host (x86, ARM, RISC-V, ...)
  - Hardware agnostic software
  - Runtime intelligent hardware
    - Cache, branch prediction, out of order processing, speculative execution, ...
    - This has been beaten to death, gains are small and incremental; you're picking up crumbs
  - Examples: high level operating systems, control code, ...

- Small code, big specialized compute → map to ~ DSA
  - Compile time intelligent software
  - Runtime deterministic hardware
    - This is where the action and ability to differentiate in hardware is
  - Examples: CNNs, almost all other technologies you're going to be interested in, ...

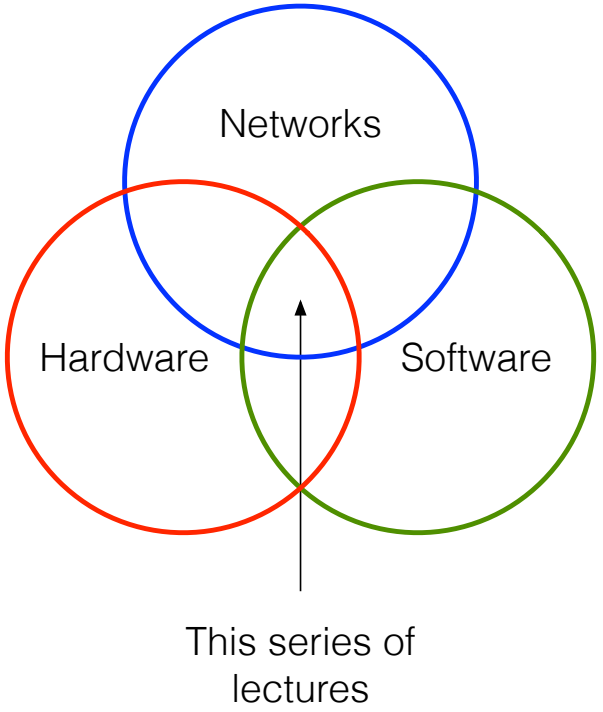


Co Design For Optimality

- Design networks for optimal performance on hardware
  - Layer sizes and operations designed to efficiently map to hardware
  - Quantization and simplification to reduce memory, reduce data movement and improve compute
- Design hardware to be an optimal target for networks
  - Memory sized such that most feature maps remain on device
  - DMA allowing parallel background data movement with transformation for remaining off device transfers
  - Computational primitive approach to big compute for ASIC efficiency with mathematical generality
  - Small general host approach for future proofing
  - Deterministic control via sequencing through a low level compile time optimized graph
- Design software to optimally map networks to hardware
  - High level hardware agnostic graph as a starting point
  - Graph compiler creating low level graphs with edges mapped to memory and nodes mapped to hardware accelerators
  - Runtime bootstrap, initialization and execution graphs with full control of hardware

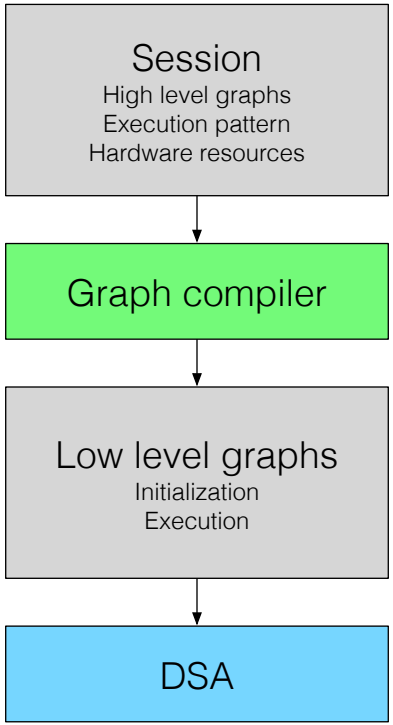
A Lecturer's Apology

- Presentation of material in this lecture is sequential
  - Networks
  - Hardware
  - Software
- But all of these topics are actually optimized together at the same time
  - As such, there are dependencies in the material
- Presentation / review strategy to address this interdependence
  - Start with a bit of a high level view of everything
  - Then sequential presentation of topics in more detail from me
  - Then go back and re read having seen everything once before



Software Preview

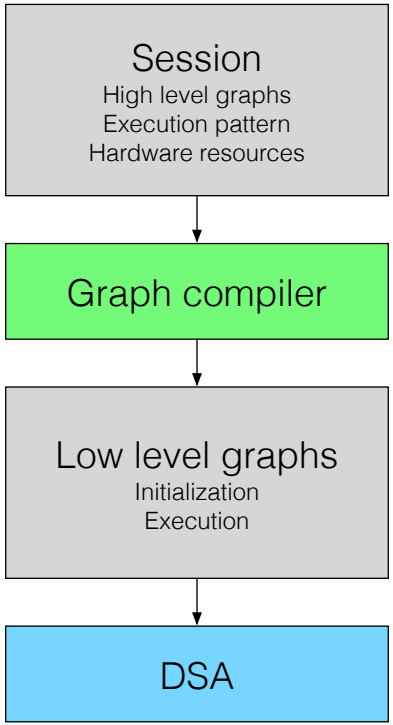
- High level graphs
  - Network specification
- Session
  - The graph compiler input
  - High level graphs + execution pattern
  - Hardware resources
- Graph compiler
  - Compile high level graphs to low level graphs
  - Edges are memory and nodes are operators
  - Separate initialization and execution graphs
  - Exploits compile time information
  - Everything is on the graph to simplify hardware
  - Fully deterministic



Software Preview

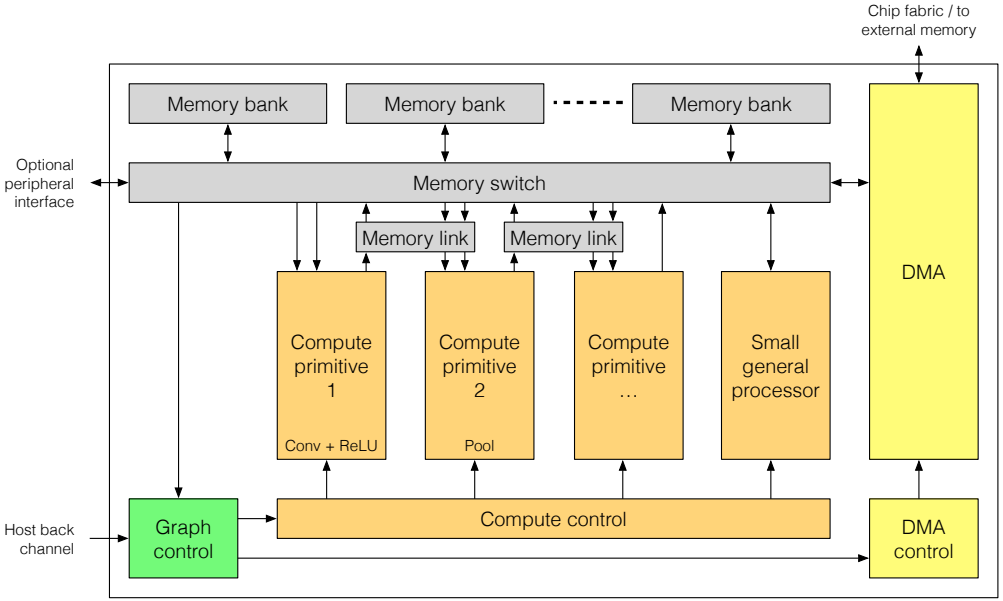
- Low level graphs
  - Map 1 to 1 to hardware
  - Node descriptors encode structure
  - Node instructions control operations
  - Operations include instruction movement, data movement and compute

- Software flow
  - Initialization
    - Called 1x
    - Link static data
    - Run initialization graph
  - Execution
    - Called for each new input
    - Link dynamic data
    - Run execution graph



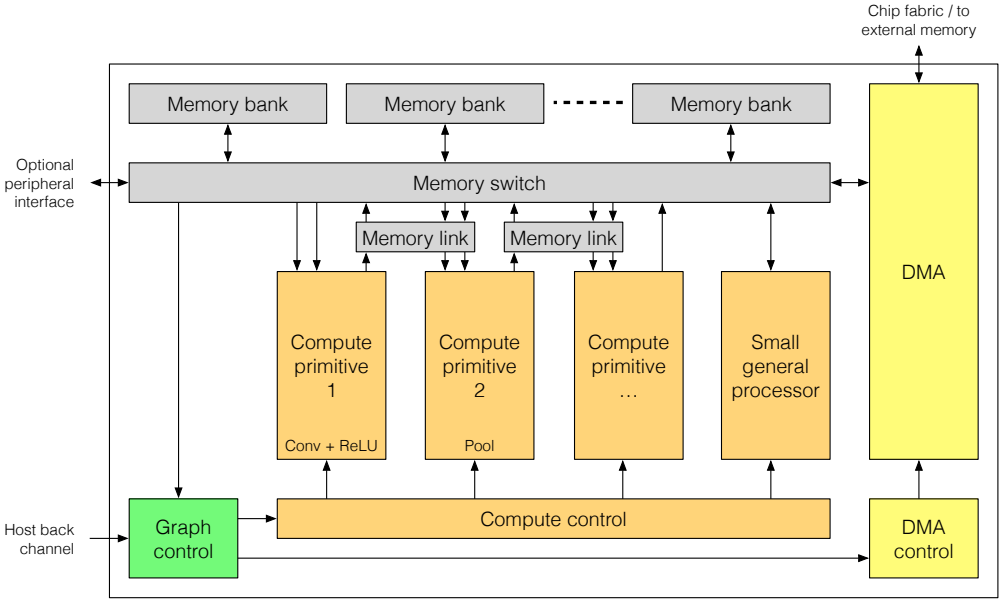
Hardware Preview

- Graph control
  - Uses node descriptors to sequence node instructions that control compute and DMA operations
- Compute and DMA control
  - Parallel compute and DMA control allows parallel data movement and compute
  - Used in a ping pong fashion for efficiency
- Memory
  - Sized to allow most feature maps to remain on device



Hardware Preview

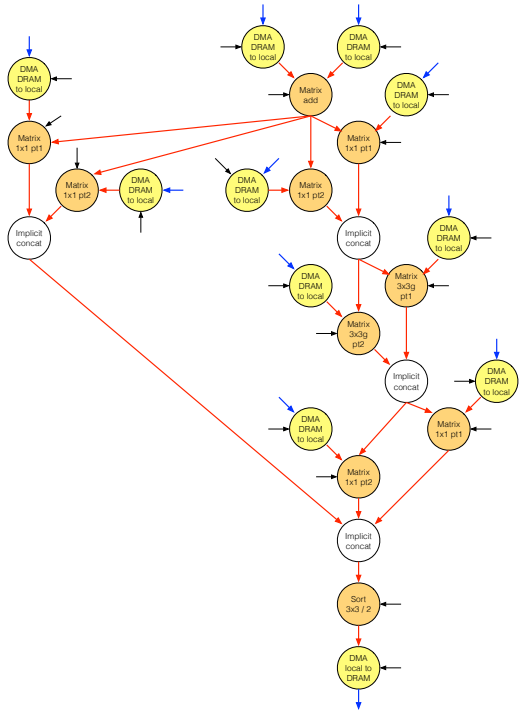
- Compute primitives
  - Use node instructions to setup and execute compute primitives
  - Compute primitives implement low level graph compute operations
  - Compute primitives include matrix, sort and a small general processor
- DMA primitives
  - Use node instructions to setup and execute DMA primitives
  - DMA primitives implement low level graph internal / external data movement operations
  - DMA primitives include multidimensional transformations and compression



Networks

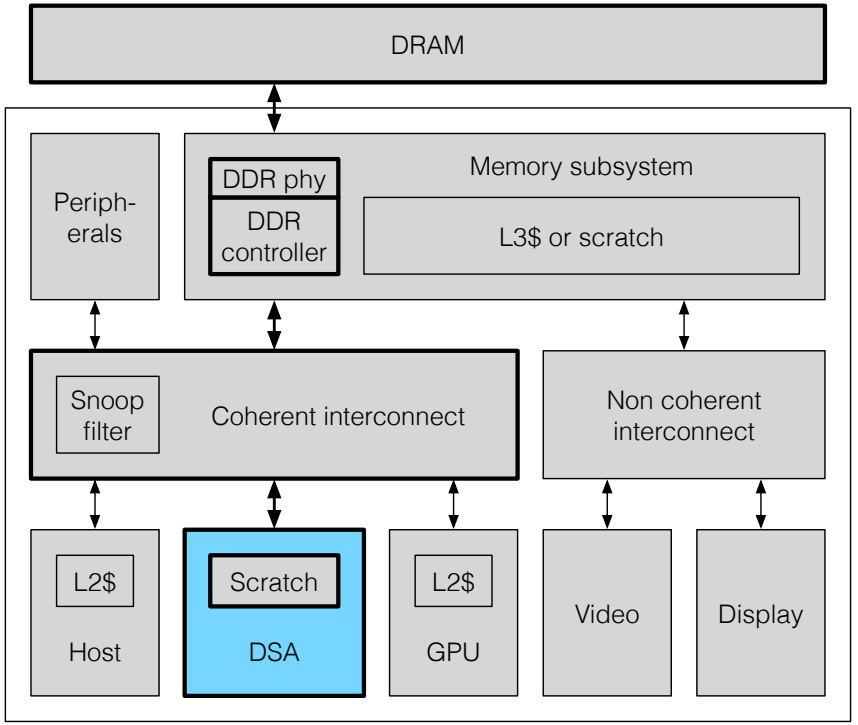
Bookkeeping

- Complexity
  - Typically learn about complexity in terms of order of operations
  - Here we're going to count operations exactly
  - All scale factors, constants, precision, memory, … matter
- Starting point
  - High level graph, edges are memory and nodes are operators
  - Generic model of off device memory, off device to on device bandwidth, on device memory and compute next to on device memory
- Track
  - Memory per edge
  - Operations per node (also type)



Bookkeeping

- Complexity
  - Typically learn about complexity in terms of order of operations
  - Here we're going to count operations exactly
  - All scale factors, constants, precision, memory, … matter
- Starting point
  - High level graph, edges are memory and nodes are operators
  - Generic model of off device memory, off device to on device bandwidth, on device memory and compute next to on device memory
- Track
  - Memory per edge
  - Operations per node (also type)



Theoretical Complexity

- Model complexity is to a 1st order approximation proportional to input pixels
  - True for CNN style 2D convolution
  - True for pooling

- Compute and filter parameter memory of a CNN style 2D convolutional layer is proportional to the square of the number of feature maps
  - 2x input / output feature maps → 4x compute and 4x filter parameter memory
  - Note that the above calculation is without grouping
  - With grouping that maintains the same number of input and output feature maps per group and just increases the number of groups the complexity increase is back to proportional

CNN style 2D convolution

MACs (assuming F – 1 pad) =
$N_i N_o F_r F_c L_r L_c$

Filter memory =
$N_i N_o F_r F_c$

Feature map memory =
$(N_i + N_o) L_r L_c$

Precision Affects Practical Complexity

- Hardware feature map and filter parameter memory complexity of a convolutional layer is as a 1st order approximation proportional to the number of bits per element
  - Why compression is important
- Hardware compute complexity of a convolutional layer is as a 1st order approximation proportional to the square of the number of bits per element
  - Key operations include additions and multiplications
  - Multiplies dominate the practical hardware complexity calculation
  - Hardware adder complexity is to a 1st order approximation proportional to the number of bits
  - Hardware multiplier complexity is to a 1st order approximation proportional to the square of the number of bits
  - Why quantization is important

8 and 16 bit fixed point multiplication

Let x_8^{lo}, x_8^{hi}, y_8^{lo} and y_8^{hi} be 8 bit integers and let x_{16} and y_{16} be 16 bit integers such that

$$x_{16} = 2^8 x_8^{hi} + x_8^{lo}$$
$$y_{16} = 2^8 y_8^{hi} + y_8^{lo}$$

Then 16 bit integer multiplication can be implemented via 4x 8 bit multiplication, 3 shifts and 3 adds

$$x_{16} y_{16} = (2^8 x_8^{hi} + x_8^{lo}) (2^8 y_8^{hi} + y_8^{lo}) = 2^{16} x_8^{hi} y_8^{hi} + 2^8 x_8^{hi} y_8^{lo} + 2^8 x_8^{lo} y_8^{hi} + x_8^{lo} y_8^{lo}$$

2x the number of bits → ~ 4x the integer multiplier complexity

Hardware Size Vs Model Size

- A models run on hardware of a given size

- Hardware size vs model size leads to 3 possibilities with respect complexity from an efficiency perspective
  - Too small to fully exercise compute resources
  - Optimally sized to fully exercise compute resources and feature maps fit fully on device
  - Too big for feature maps to fit on device and off device data movement increases nonlinearly

Training Vs Testing

Training

- Can batch inputs (allowing the amortizing of weight movement across multiple inputs)
- Need batch norm operations
- Need error calculation
- Need to maintain memory space for reverse mode automatic differentiation so there's less reuse of memory
- Typically need higher precision floating point

Testing

- Sometimes can batch inputs; sometimes process 1 input at a time for latency reasons
- Absorb batch norm operations
- Need network output
- Don't need to maintain memory space for reverse mode automatic differentiation so there's more reuse of memory possible
- Frequently ok with lower precision fixed point

Quantization

Goal Is To Reduce Bits Per Memory Element

- The purpose of quantization
  - Reduce the number of bits per memory element to reduce memory and bandwidth requirements and improve (practical implementations of) computation while minimizing accuracy loss

- Rules of thumb
  - Memory is proportional to the number of bits
  - Bandwidth is proportional to the number of bits
  - Adder complexity is proportional to the number of bits
  - Multiplier complexity is proportional to the square of the number of bits
  - Comparator complexity is proportional to the number of bits

Data Formats

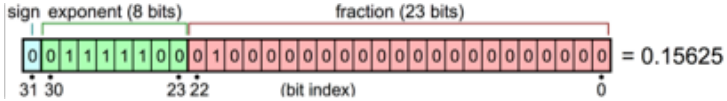
- Common data formats
  - 64 bit float 53.11 (IEEE 754 double)
  - 32 bit float 24.8 (IEEE 754 single)
  - 16 bit float 11.5 (IEEE 754 half; more precision less range than bfloat16)
  - 16 bit float 8.8 (bfloat16; more range less precision than IEEE 754 half)
  - 16 bit fixed (signed and unsigned)
  - 8 bit fixed (signed and unsigned)
  - ---------- (mainstream above line, research-y below line; becoming a candidate for analog)
  - 4 bit fixed
  - 2 bit fixed
  - 1.5 bit fixed {-1, 0, 1}
  - 1 bit fixed Nice for compactness but no 0

- Additional data formats that have positive hardware qualities
  - Power of 2 filter coefficients
    - Simplifies multiplication to shift and add
    - Results in non uniform step sizes which are not always good
    - But still use arbitrary precision for bias coefficients

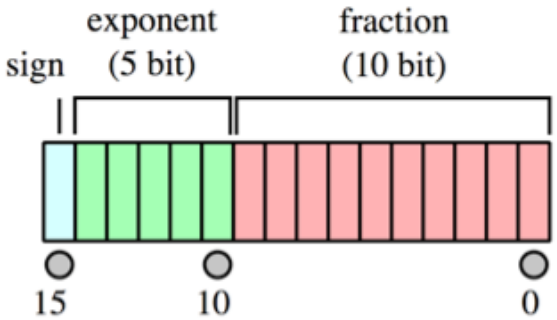
Data Formats For Network Training

Training typically needs more bits than testing; IEEE 754 32 bit float is most common now, bfloat16 will likely gain in popularity going forward; int8 with block scaling is also becoming more common to either integrate with training from the start or after an initial floating point warm up period

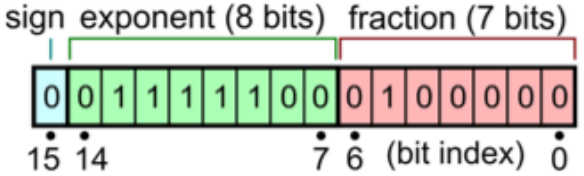
- IEEE 754 float 32: 1 bit sign, 8 bits exponent, 23 bits significand
  - Ignoring special values signaled by exponent values of 0 and 255
  - Value = sign x range x precision
    = $(-1)^{sign}$ x $2^{exponent - 127}$ x $1.significand_2$
    = {–1, 1} x $2^{\{-126, \ldots, 127\}}$ x $[1, 2)_{23 \text{ bits precision}}$



- IEEE 754 float 16: 1 bit sign, 5 bits exponent, 10 bits significand
  - Ignoring special values signaled by exponent values of 0 and 31
  - Value = $(-1)^{sign}$ x $2^{exponent - 15}$ x $1.significand_2$
    = {–1, 1} x $2^{\{-14, \ldots, 15\}}$ x $[1, 2)_{10 \text{ bits precision}}$



- bfloat 16: 1 bit sign, 8 bits exponent, 7 bits significand
  - Ignoring special values signaled by exponent values of 0 and 255
  - Value = $(-1)^{sign}$ x $2^{exponent - 127}$ x $1.significand_2$
    = {–1, 1} x $2^{\{-126, \ldots, 127\}}$ x $[1, 2)_{7 \text{ bits precision}}$



Figures from https://en.wikipedia.org/wiki/IEEE_754 and https://en.wikipedia.org/wiki/Bfloat16_floating-point_format

Floating Point To Fixed Point Conversion

Signed fixed point is an integer in $\{-2^{bits-1}, \ldots, 2^{bits-1} - 1\}$ and unsigned fixed point is an integer in $\{0, \ldots, 2^{bits} - 1\}$

- Signed example for $\{\mathbf{X}_q, s_X\}$ = quantize($\mathbf{X}$, bits)
  - Target range of $\{-2^{bits-1} - 1, \ldots, 2^{bits-1} - 1\}$
    - Keeping symmetric about 0 by ignoring the value -2^{bits-1}
    - For 8 bits this is {–127, …, 127}
    - Assume scale s_X is chosen such that there's no clipping
  - maxAbsX = max(|$\mathbf{X}$|)
  - s_X = maxAbsX / $(2^{bits-1} - 1)$
  - $\mathbf{X}_q$ = round($\mathbf{X}$ / s_X)

- Unsigned example for $\{\mathbf{X}_q, s_X\}$ = quantize($\mathbf{X}$, bits)
  - Target range of $\{0, \ldots, 2^{bits} - 1\}$
    - For 8 bits this is {0, …, 255}
    - Assume input $\mathbf{X}$ is non negative
    - Assume scale s_X is chosen such that there's no clipping
  - maxX = max($\mathbf{X}$)
  - s_X = maxX / $(2^{bits} - 1)$
  - $\mathbf{X}_q$ = round($\mathbf{X}$ / s_X)

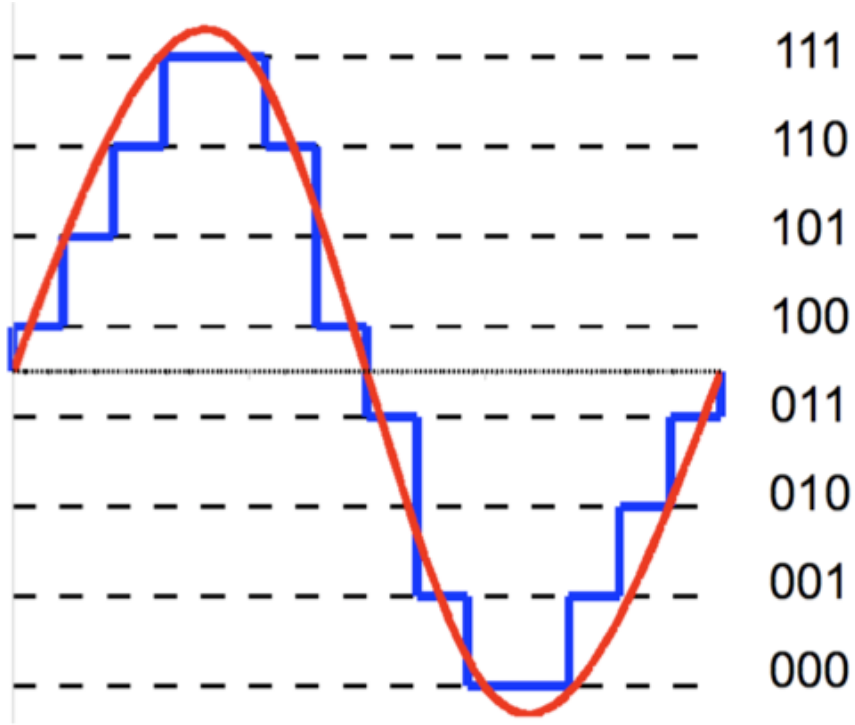


Figure from https://en.wikipedia.org/wiki/Quantization_(signal_processing)

Items That Can Be Quantized

- All memory elements
  - Feature maps
  - Filter coefficients and other parameters
  - Gradients and associated training terms
    - Possible useful for training
    - But also make training more difficult and it's already a hassle
    - So skip for now and focus on quantizing memory used in testing
    - Note that will still cover quantized training
    - Quantized values will be in the forward path though
- Feature maps and filter coefficients can use different (but compatible) quantization choices
  - Ex: 4 bit signed fixed point filter coefficients and 8 bit unsigned fixed point feature maps
- Need to understand the implications of the reduction in different values that a memory element can take
  - Appropriately balance reduction in range and precision
  - Goal is to maximize efficiency gain and minimize accuracy loss

Fx Pt Quantized CNN Style 2D Convolution

- Start from what can be calculated with fixed point hardware
  - $\mathbf{Y}_q = \text{clip}(\text{round}(s_c\,(\mathbf{H}_q\,\mathbf{X}_q + \mathbf{V}_q)))$
  - $\mathbf{Y}_q$ is the fixed point quantized 2D matrix created via re arranging the 3D tensor of output feature maps
  - $\mathbf{H}_q$ is the fixed point quantized 2D matrix created via re arranging the 4D tensor of filter coefficients
  - $\mathbf{X}_q$ is the fixed point quantized 2D matrix created via a Toeplitz style arrangement of the 3D tensor of input feature maps
  - $\mathbf{V}_q$ is the fixed point quantized 2D bias matrix created via an outer product of a bias vector and 1s row vector

- Compute scale s_c
  - s_c is a scale selected to constrain the output fixed point range to the target number of bits
  - Can select a static s_c based on training data
  - Can select a dynamic s_c based on monitoring accumulator values to maximize dynamic range individually for each input (though this requires some downstream adjustments where additions occur)

- Note
  - $\mathbf{H}_q = \text{clip}(\text{round}(\mathbf{H} / s_H)) \approx \mathbf{H} / s_H$ static, typically selected to maximize range
  - $\mathbf{X}_q = \text{clip}(\text{round}(\mathbf{X} / s_X)) \approx \mathbf{X} / s_X$ static or dynamic, from the previous layer
  - $\mathbf{V}_q = \text{clip}(\text{round}(\mathbf{V} / s_V)) \approx \mathbf{V} / (s_H\, s_X)$ static or dynamic, dependent on the filter and input scale; $s_V = s_H\, s_X$ to align bias (additive) scale with combined filter and input (multiplicative) scale

Fx Pt Quantized CNN Style 2D Convolution

- Substituting in and ignoring clipping and rounding
  - $\mathbf{Y}_q \approx s_c ((\mathbf{H} / s_H)(\mathbf{X} / s_X) + (\mathbf{V} / (s_H s_X)))$
    $\approx (s_c / (s_H s_X)) (\mathbf{H X} + \mathbf{V})$

- Let $s_Y = (s_H s_X) / s_c$ and $\mathbf{Y} = s_Y \mathbf{Y}_q$ then
  - $s_Y \mathbf{Y}_q \approx \mathbf{H X} + \mathbf{V}$
  - $\mathbf{Y} \approx \mathbf{H X} + \mathbf{V}$

- This implies we can approximate floating point CNN style 2D convolution $\mathbf{Y} = \mathbf{H X} + \mathbf{V}$ with fixed point CNN style 2D convolution $\mathbf{Y}_q = \text{clip}(\text{round}(s_c (\mathbf{H}_q \mathbf{X}_q + \mathbf{V}_q)))$ and the following constraints
  - Bias scale $s_V = s_H s_X$ is dependent on the filter and input scale; as such, the bias typically uses 2x – 4x the number of bits vs multiplicative parameters
  - Output feature map scale $s_Y = (s_H s_X) / s_c$ is a function of the filter, input and compute scales; for convenience of implementation the compute scale s_c may have some constraints (e.g., only powers of 2)
  - Note the coupling of scales from 1 layer to the next
  - Note that typically do accumulation at 4x input scale before compute scale

Fx Pt Quantized Pooling

- Not a big deal
  - For max pooling: the set of possible output values come from the set of input values
  - For avg pooling: the set of possible output values is bound by the range of input values

31	21	33	34	5	2	15
10	29	32	6	27	16	13
7	4	28	20	24	30	26
25	18	14	35	22	1	3
17	23	12	8	19	9	11

Max pool ↓ 3x3 / 2

33	34	30
28	35	30

32	36	68	56	72	48
8	40	64	84	80	12
28	96	92	76	16	4
52	88	44	20	60	24

Avg pool ↓ 2x2 / 2

29	68	53
66	58	26

Fx Pt Quantized N Input 1 Output Operations

With additive (definitely) or concatenative (probably) operations

- A bit of a hassle
  - Need to align scales of inputs
  - A simple way to do this is to us the largest scale as a common scale to bring all other scales to, then perform the operation

- Examples
  - Element wise addition (as in ResNet)
  - Concatenation (as in GoogLeNet / Inception, DenseNet, ...)
  - After grouped convolutions (if different scales are used for different groups)

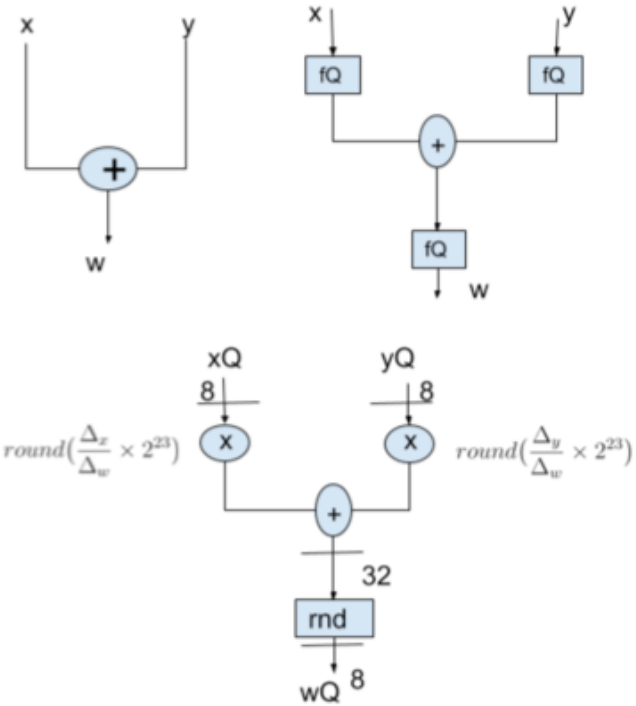


Figure from https://arxiv.org/abs/1806.08342

Fx Pt Quantized Batch Norm

- A bit of a hassle
  - Because it's data dependent
  - But on the upside it's needed during training
  - Typically do via 2 passes through convolution

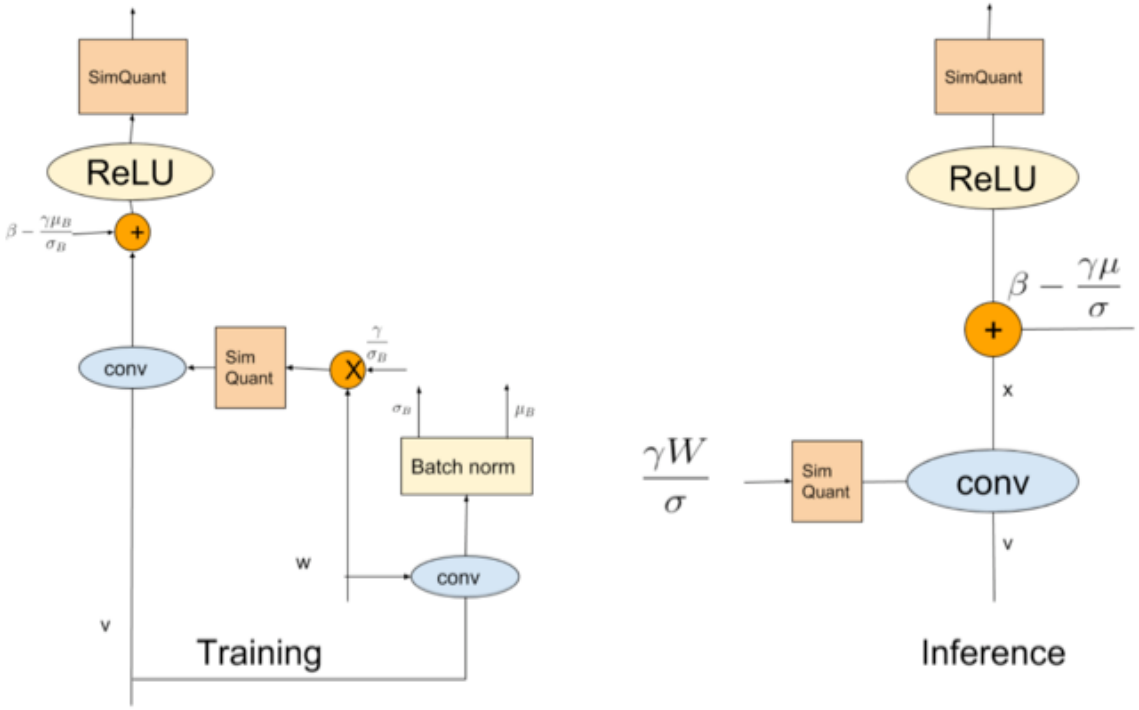


Figure from https://arxiv.org/abs/1806.08342

Range Options

- Previous examples selected scales such that we don't clip but that's not the only option
  - Clipping becomes more useful for static compute scale selection

- No clipping
  - Signed
    - -MaxAbs to MaxAbs (parameters and feature maps in general)
    - Min to Max (implies extra operations, get at most 1 extra bit)
  - Unsigned
    - 0 to Max (feature maps after ReLU)
    - Min to Max (implies extra operations)

- Clipping
  - For traditional signal processing set clip value to maximize SNR but it's different for CNNs
  - Data space is usually smooth but feature space is spikey with many small values
  - SNR as an evaluation metric is the wrong criteria by itself
  - Really care about information and final accuracy

Simulating Fx Pt In Floating Pt Hardware

- A common trick is to introduce quantize – un quantize blocks into the graph
  - Quantize $\mathbf{X}_q$ = round($\mathbf{X}$ / s_X) a tensor to introduce loss of information
  - Un quantize the tensor $\mathbf{X} \approx s_X \mathbf{X}_q$ to bring back to original range
  - Perform operation
  - Take care to properly handle N input 1 output and batch norm operations

Quantizing A Trained Network

- Example: quantizing a trained network with 32 bit floating point filter coefficients to 8 bit fixed point multiplicative filter coefficients and 32 bit additive filter coefficients for deployment

- Permutations
  - With or without dynamically selecting compute scale s_c
  - Note that dynamic scale selection requires extra min / max tracking at the accumulator precision

- Comments
  - Can typically tolerate some clipping in the beginning but not the end (when trying to find the max)
  - Highly grouped networks are a challenge
  - Very low precisions are a challenge; sometimes alter network structure to implicitly increase precision

Example strategy for 32 bit float to 8 bit fixed

- Run a large number of inputs through the network and compute the average of the min and max value for each feature map at every point in the network
- Consider multiplying these scales by a ramp that starts at 1.0 for the first layer and ends at 2.0 for the last layer as the net is more tolerant of clipping in the beginning than end
- Select the multiplicative filter coefficient scales for all layers using the maxAbsX approach for a symmetric range about 0
- Select the scale to quantize the input using maxX or maxAbsX
- Starting at the first layer select the scale of additive params based on the input and multiplicative parameters then select the scale of the compute to match the target output range
- Iteratively walk forward from the tail of the network to the head and repeat the selection of the scale of the additive parameters and the compute (note that this captures the linking of scales from 1 layer to the next)

Quantized Network Training / Retraining

- Permutations
  - From scratch
  - From a pre trained floating point model

- Notes
  - Likely accumulate gradients at a higher precision
  - May start training at a higher number of bits (float or fixed) then reduce the number of bits as ranges stabilize

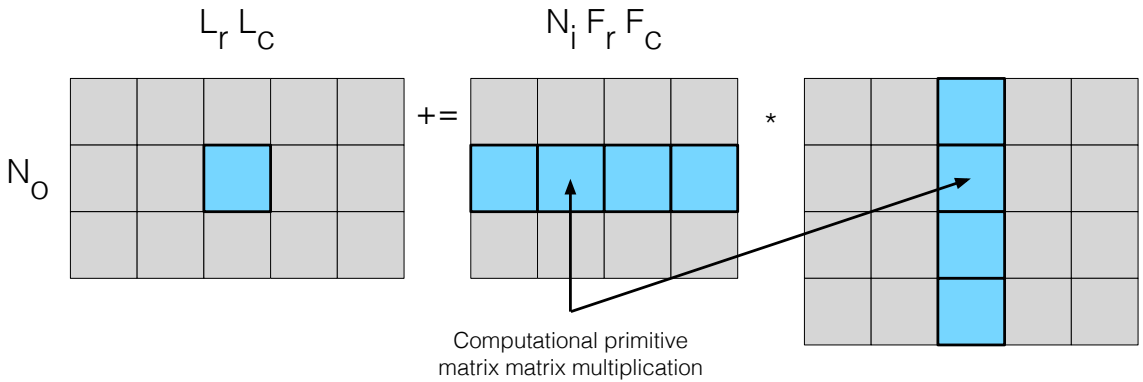
For more details see:

Quantizing deep convolutional networks for efficient inference: A whitepaper
https://arxiv.org/abs/1806.08342

Efficient Sizing

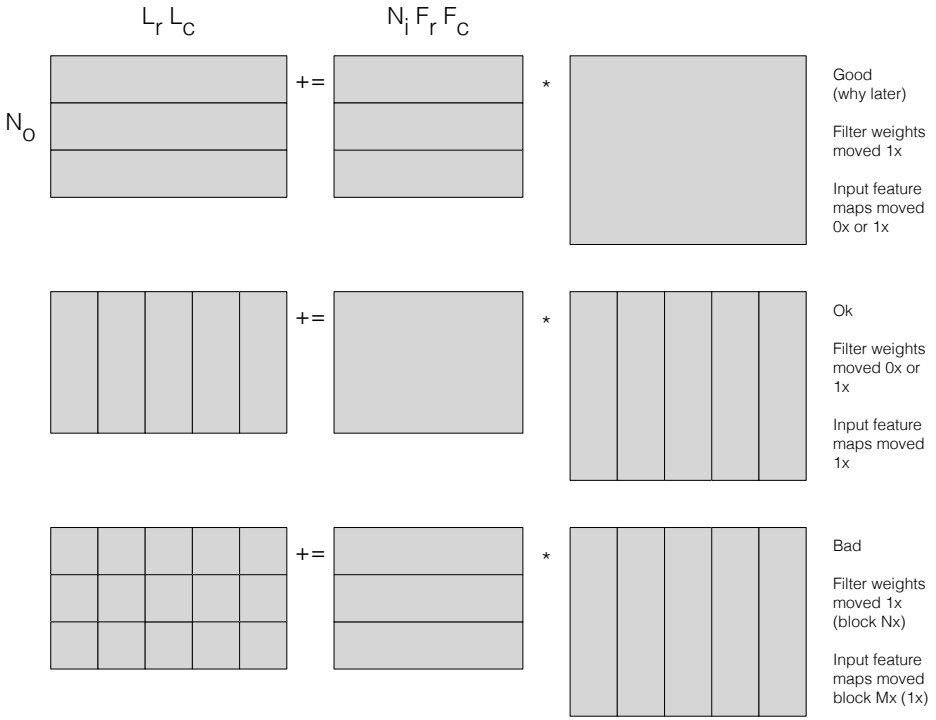
CNN Style 2D Convolution Computation

- A preview
  - We're going to specify hardware with an optimized matrix matrix multiplication primitive
  - Large matrix matrix multiplication is going to be implemented via tiled smaller matrix matrix multiplication
  - Data movement limits compute
- Tiling efficiency
  - In this figure N_o, L_r L_c and N_i F_r F_c are all integer multiples of the tile size
  - Excess compute (inefficiency) occurs when they're not
  - Extreme case: fully grouped $N_i = N_o = 1$



CNN Style 2D Convolution Data Movement

- A preview
  - We're going to specify hardware with a fixed amount of on device memory
  - Compute is out of on device memory
  - Off device to on device data movement is slow and consumes a lot of power
- Memory efficiency
  - A good situation is that input feature maps for a given layer fit on device and filter coefficients are block row moved for that layer
  - An ok situation is that filter coefficients can fit fully on device for a given layer and input feature maps are block row moved
  - A bad situation is that neither input feature maps or filter coefficients fit fully on device for a given layer and multiple moves are needed



Simplification

Simplifying Networks Improves Performance

- Given a trained network network make structural modifications to reduce complexity
  - Modifications can be un structured perturbations
    - Random sparsity in filter coefficients
  - Modifications can be structured perturbations
    - Removal of whole feature maps
    - Remove of whole filters or more likely groups of filters
  - Modifications can change graph structure
    - Transforming 3 layers to 1

An incomplete laundry list of methods

- Optimal brain damage
  - http://yann.lecun.com/exdb/publis/pdf/lecun-90b.pdf
- Optimal brain surgeon and general network pruning
  - https://authors.library.caltech.edu/54981/1/Optimal%20Brain%20Surgeon%20and%20general%20network%20pruning.pdf
- Efficient and accurate approximations of nonlinear convolutional networks
  - https://arxiv.org/abs/1411.4229
- Accelerating very deep convolutional networks for classification and detection
  - https://arxiv.org/abs/1505.06798
- Learning efficient convolutional networks through network slimming
  - https://arxiv.org/abs/1708.06519
- To prune or not to prune: exploring the efficacy of pruning for model compression
  - https://openreview.net/pdf?id=Sy1iIDkPM

Simplify Or Design A Simpler Network?

- Taking an untrained network, reducing it's complexity via structural modifications and training it is not referred to here as network simplification
  - That’s really just simple network design and training
- Starting from a simple design is likely a better strategy than starting from a more complex design and simplifying
  - However, some places have found it easier to train a larger model first then simplify
  - But that could be because the structure of the simpler network was sub optimal in the first place
- However, just designing a simple network isn't practical if the simplifications of interest aren’t predictable at network design time
  - An example of this would be random sparsity in filters

Rules Of Thumb

- Memory
  - Early in the network feature maps tend to dominate memory
  - Later in the network filter coefficients tend to dominate memory
- Compute
  - Early in the network tends to dominate compute
  - Compute is typically proportional to data volume
  - Data volume shrinks by ~ 2 after every pooling stage (1/4 space, 2x channel)

CNN style 2D convolution

MACs (assuming F – 1 pad) =
$N_i N_o F_r F_c L_r L_c$

Filter memory =
$N_i N_o F_r F_c$

Feature map memory =
$(N_i + N_o) L_r L_c$

Simplifying Convolutional Layers

- Typically means reducing compute
- Examples
  - Random sparsity in filter coefficients
    - Can be encouraged during training
    - Ex: using a threshold and a deflationary approach
  - Structured sparsity in filter coefficients
    - Can be forced in training
    - Ex: grouping
  - Input or output feature map channel reductions
    - Induces fewer filter coefficients
- Also means matching the shape of the convolutional layer optimally to the hardware

Simplifying Pooling Layers

- Typically means removing overlap when striding to allow for better memory locality and no buffering of overlap

- Examples
  - Use 2x2/2 vs 3x3/2

Simplifying Fully Connected Layers

- Typically means reducing memory

- Examples
  - Decompositions of the filter matrix that exploit structure
  - SVD / outer product based decompositions of the matrix that reduce the number of required parameters

Hardware

Software

References

Quantization

- Estimating or propagating gradients through stochastic neurons for conditional computation
  - https://arxiv.org/abs/1308.3432
- Training deep neural networks with low precision multiplications
  - https://arxiv.org/abs/1412.7024
- Hardware-oriented approximation of convolutional neural networks
  - https://arxiv.org/abs/1604.03168
- Quantized neural networks: training neural networks with low precision weights and activations
  - https://arxiv.org/abs/1609.07061
- Mixed precision training
  - https://arxiv.org/abs/1710.03740
- Quantization and training of neural networks for efficient integer-arithmetic-only inference
  - https://arxiv.org/abs/1712.05877
- Mixed precision training of convolutional neural networks using integer operations
  - https://arxiv.org/abs/1802.00930
- Quantizing deep convolutional networks for efficient inference: A whitepaper
  - https://arxiv.org/abs/1806.08342

Quantization

- Ristretto: a framework for empirical study of resource-efficient inference in convolutional neural networks
  - http://lepsucd.com/?page_id=621
  - http://lepsucd.com/?page_id=637
  - https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8318896

- IEEE 754
  - https://en.wikipedia.org/wiki/IEEE_754
- bfloat16 floating-point format
  - https://en.wikipedia.org/wiki/Bfloat16_floating-point_format

Simplification

- Optimal brain damage
  - http://yann.lecun.com/exdb/publis/pdf/lecun-90b.pdf
- Optimal brain surgeon and general network pruning
  - https://authors.library.caltech.edu/54981/1/Optimal%20Brain%20Surgeon%20and%20general%20network%20pruning.pdf
- Efficient and accurate approximations of nonlinear convolutional networks
  - https://arxiv.org/abs/1411.4229
- Accelerating very deep convolutional networks for classification and detection
  - https://arxiv.org/abs/1505.06798
- Learning efficient convolutional networks through network slimming
  - https://arxiv.org/abs/1708.06519
- To prune or not to prune: exploring the efficacy of pruning for model compression
  - https://openreview.net/pdf?id=Sy1iIDkPM