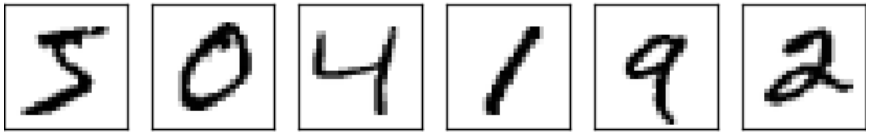
Lecture: Deep Networks and CNNs

- Topics: wrap-up of deep networks, accelerator basics,
  Diannao accelerator

- Resources: http://www.cs.utah.edu/~rajeev/cs7960/notes/
- Canvas/registration

Deep Networks for Image Classification

- MNIST: 784-pixel images of hand-written digits; 50K training images; 10K testing images



We will primarily focus on image classification because it is one of the success stories for deep learning. Let's start by examining the workloads used to evaluate deep learning. The most commonly used benchmark is the MNIST dataset. It has been over-used to the point that it is starting to lose relevance as a research platform. Each image in this dataset is a 28x28 (784 pixels) image of a hand-written digit. 50K images are used for training and 10K images are used for testing.

Deep Networks for Image Classification

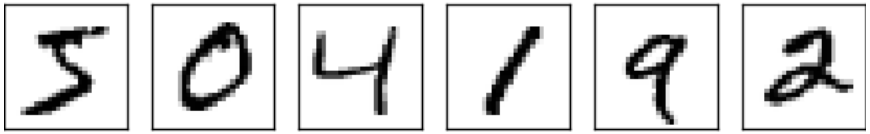
- ILSVRC: e.g., 1000 categories, 1.2 million training images, 150K test images, top-5 criterion




A second commonly used benchmark set is ImageNet, which consists of over 16 million images (examples above) with objects in 20K different categories. A subset of these are used for the annual ILSVRC competition, e.g., in some years, they have used 1.2 million images for training and 150K images for testing. These images only have objects in 1000 categories. Some of these images may have multiple objects, e.g., a dog chasing a ball, but the image label says "dog". To deal with such situations, the competition defines a "top-5 criterion" that allows the network to make 5 best guesses – if one of these matches the label, the prediction is deemed accurate. Clearly, the "top-1 criterion" is also a relevant metric. Recently, even this simplified subset has been "conquered", with deep networks achieving near-human accuracy (top-5 rate of 95%, top-1 rate of close to 80%). Of course, these are relatively easy problems, so much work remains in designing better deep networks.

Deep Networks for Image Classification

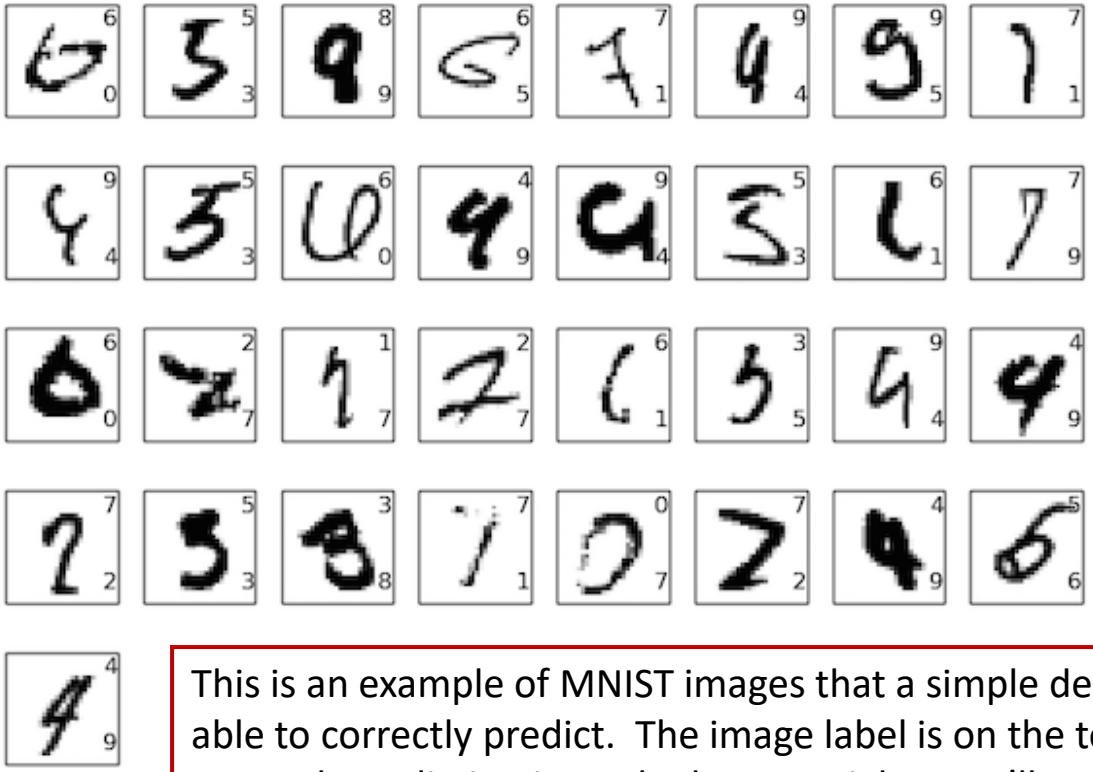
- MNIST: 784-pixel images of hand-written digits; 50K training images; 10K testing images



- ILSVRC: e.g., 1000 categories, 1.2 million training images, 150K test images, top-5 criterion

Deep Learning Success

- Modern deep learning is better than humans on both MNIST (>99%) and ILSVRC top-5 (>95%)



This is an example of MNIST images that a simple deep network was not able to correctly predict. The image label is on the top-right and the network prediction is on the bottom-right. You'll see that for the most part, these are examples of poor hand-writing, not poor network behavior.

Accuracies

This slide shows how we can begin with a shallow network and keep adding layers to get higher accuracies on MNIST. The final network has two convolutional/pooling layers, two fully-connected layers, and a bunch of tricks to prevent over-fitting and manage the weights.

- Baseline: single hidden layer with 100 neurons: 97.8%
- CNN: added 20 5x5 filters and a 2x2 max-pool: 98.78%
- CNN: added 40 20x5x5 filters and 2x2 max-pool: 99.06%
- CNN: substitute sigmoid with ReLU: 99.23%
- CNN: expand the training data: 99.37%
- CNN: adding two fully-connected layers: 99.43%
- CNN: 1000 neurons in fully-connected layers: 99.47%
- CNN: adding dropout: 99.60%
- CNN: voting among an ensemble of 5 nets: 99.67%

(ILSVRC winners have 152 layers)

Many ways to avoid over-fitting in fully-connected networks (conv layers don't need these because the weights are shared): L2 regularization, dropout, expanded inputs.

Accuracies

These tricks include using a better activation function (ReLU), converting each training image into 5 training images (by shifting each image to the top/left/right/bottom by 1 pixel), dropout (randomly eliminating some activations during training), L2 regularization (including the weights in the cost function so the weights don't grow too large), etc.

- Baseline: single hidden layer with 100 neurons: 97.8%
- CNN: added 20 5x5 filters and a 2x2 max-pool: 98.78%
- CNN: added 40 20x5x5 filters and 2x2 max-pool: 99.06%
- CNN: substitute sigmoid with ReLU: 99.23%
- CNN: expand the training data: 99.37%
- CNN: adding two fully-connected layers: 99.43%
- CNN: 1000 neurons in fully-connected layers: 99.47%
- CNN: adding dropout: 99.60%
- CNN: voting among an ensemble of 5 nets: 99.67%

(ILSVRC winners have 152 layers)

Many ways to avoid over-fitting in fully-connected networks (conv layers don't need these because the weights are shared): L2 regularization, dropout, expanded inputs.

Glossary

- Sigmoid activation function: $f(x) = 1/(1+e^{-x})$ – provides a smooth step function; tanh is similar.
- ReLU activation function: f(x) = max (0,x) – it allows the output to grow larger than 1, and has a larger σ' of 0 or 1.
- Softmax activation function: $f(z_j) = e^{zj}/\Sigma_k e^{zk}$ -- it's normalizing the neuron outputs in a layer so they sum to 1 and the largest neuron sticks out
- Feature map and filter: a filter or kernel is the grid of weights; a feature map is the resulting set of values when a filter is applied to a set of inputs
- Max pooling: extracts the largest value in a 2D input grid
- L2 pooling: computes the square root of the sum of squares of the values in a 2D input grid
- L2 regularization: includes the weights in the cost function during training so we're trying to not only reduce the error, but also the values of the weights
- Expanded inputs: to avoid over-fitting, the (say) 50K training images are expanded to 250K images. Each image is shifted slightly to the left/right/top/bottom.
- Dropout: some activation functions are randomly dropped during training (to avoid overfitting).
- DNN/CNN: CNNs use shared kernels for all neurons in a feature map, while DNNs use private kernels for each neuron in a feature map.
- SVM: a mathematical approach to incrementally define hyperplanes that separate clusters – an alternative way to classify inputs into different categories.

LSTMs

- The neurons in MLPs and CNNs do not have state – every input image results in brand new computations with no memory of previous images.
- An LSTM is better suited for speech/text processing where interpreting a new syllable or pronoun may depend on past inputs.
- Recurrent neural networks (RNNs, where a layer's output feeds back as input for the next computation) have evolved into LSTMs

LSTMs

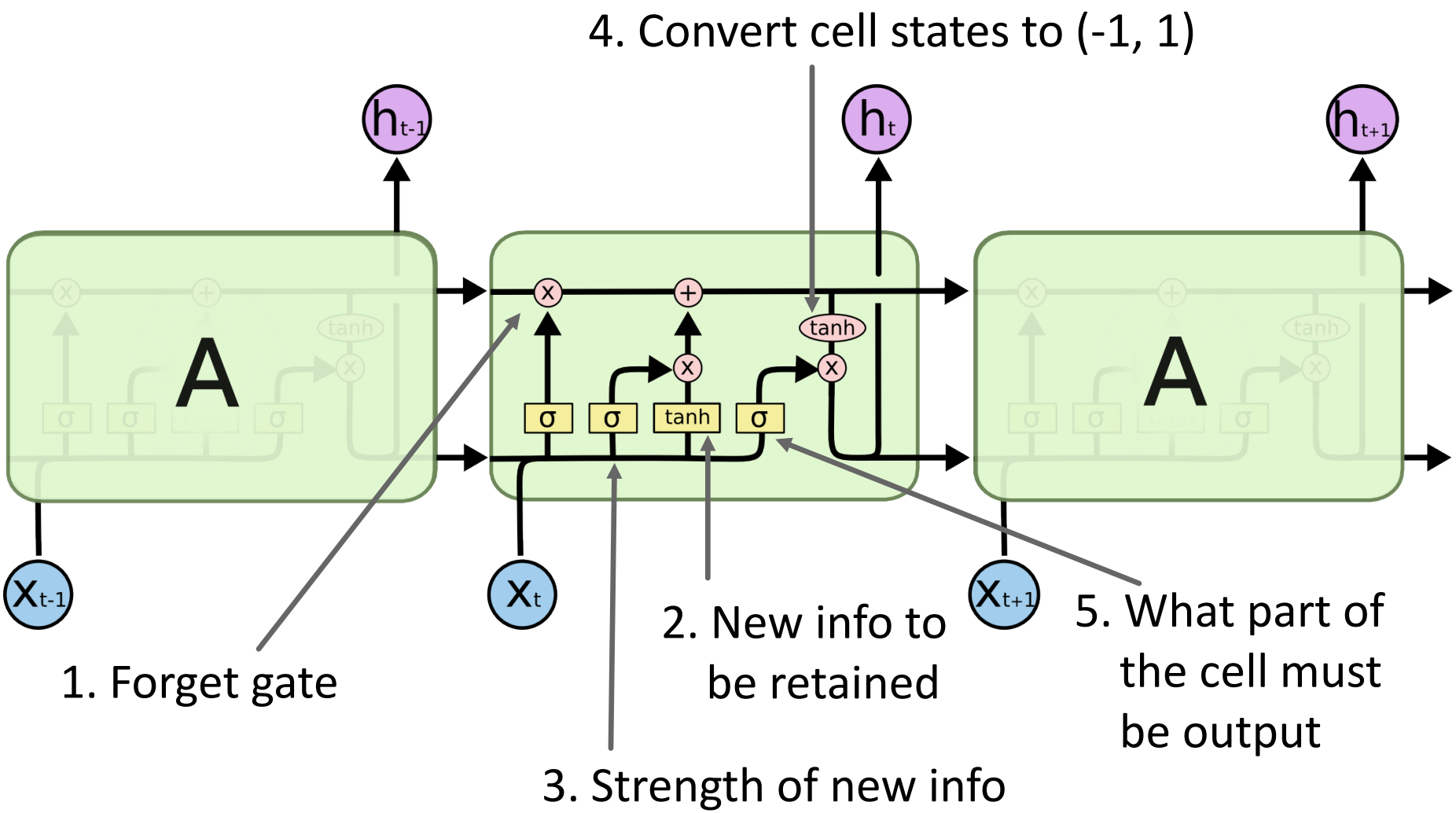


Image credit: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Accelerator Basics

- Software optimizations: loop ordering, tiling
- Hardware techniques: prefetching, SIMD units, near-data processing, …

The last lecture discussed recent progress in deep learning algorithms. These algorithms (inference and training) consume large amounts of time and energy on modern systems. Custom accelerator hardware will clearly help. We'll start by first examining a few software optimizations, followed by accelerators that rely on digital units.

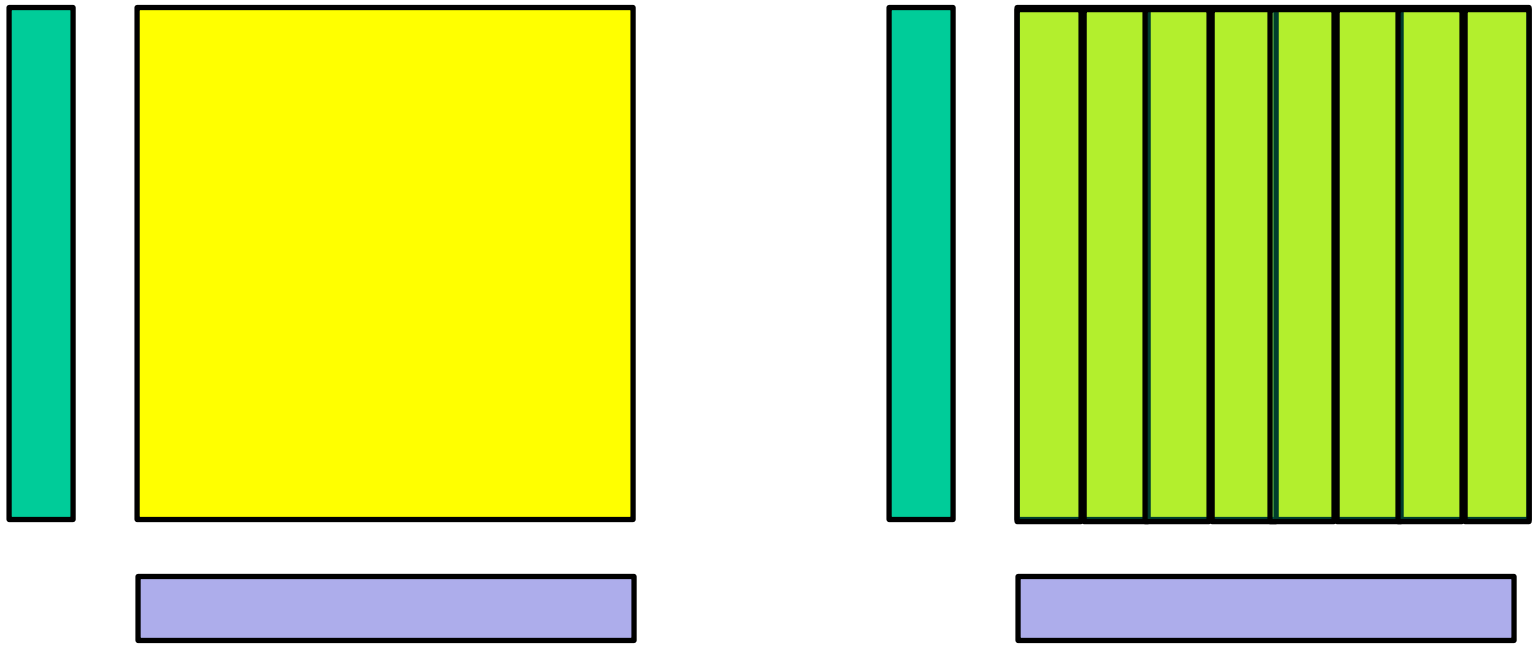
Loop Nesting

```
for n = 1 to N          Do all images
  for k = 1 to K          Do all output feature maps
    for c = 1 to C          Do all input feature maps
      for w = 1 to W          Do all horizontal pixels in output fmap
        for h = 1 to H          Do all vertical pixels in output fmap
          for r = 1 to R          Do all horizontal pixels in kernel
            for s = 1 to S          Do all vertical pixels in kernel
              out[n][k][w][h] +=
                in[n][c][w+r-1][h+s-1] *
                filter[k][c][r][s];
```

Figure 3: 7-dimensional CNN loop nest.

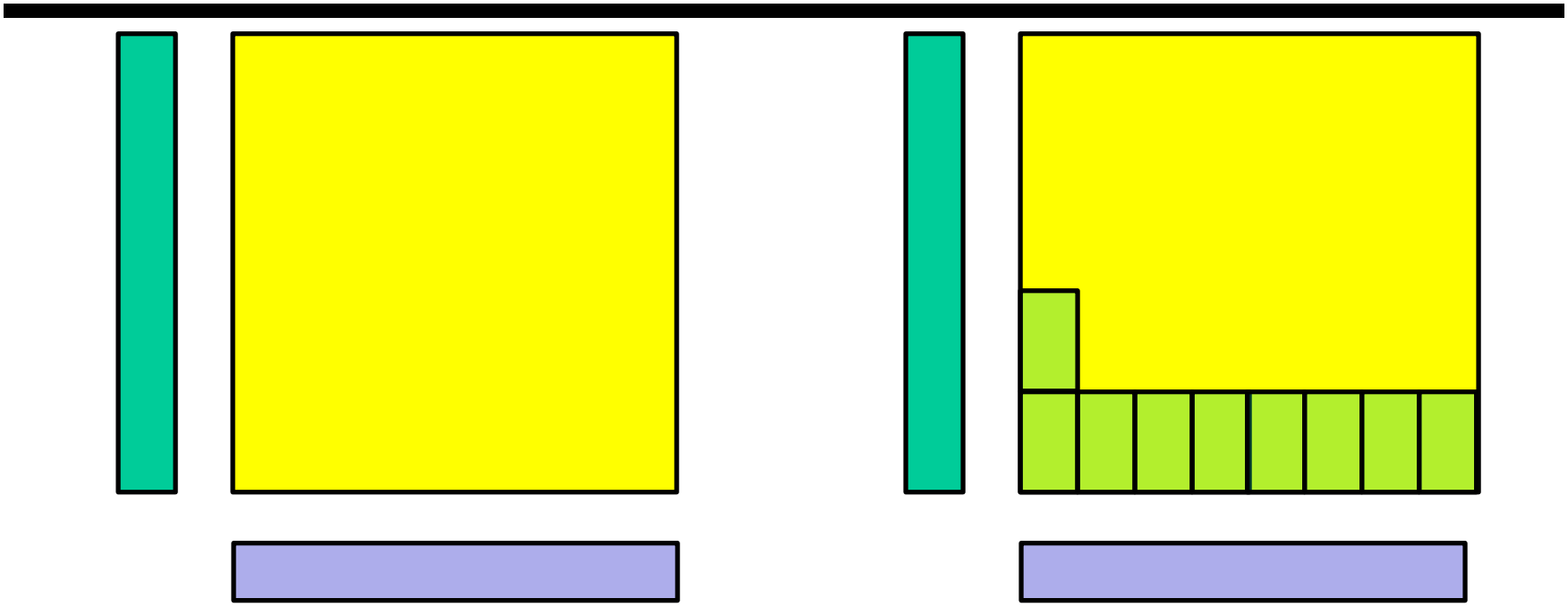
Source: "SCNN: An Accelerator for Compressed-Sparse Convolutional Neural Networks," A. Parashar et al., ISCA 2017

Data Access Pattern – Classifier Layer



Let's first understand tiling in general terms. Consider the vector matrix multiplication that is required in the fully-connected classifier layer. The inputs come from the left (green) and each column computes a dot-product for its neuron (violet). As shown on the right, the columns of the synaptic weight matrix are accessed one column at a time. If we assume that the matrices are large and that even a single row or column is too large to fit in cache, the input vector keeps entering the cache and keeps getting evicted out of the cache. So, even though the input vector has reuse, it is not exploiting the cache. The synaptic weights have no reuse; they have to be fetched entirely from memory, used once, and then thrown away. Similarly, the neurons have no reuse. They are written exactly once.

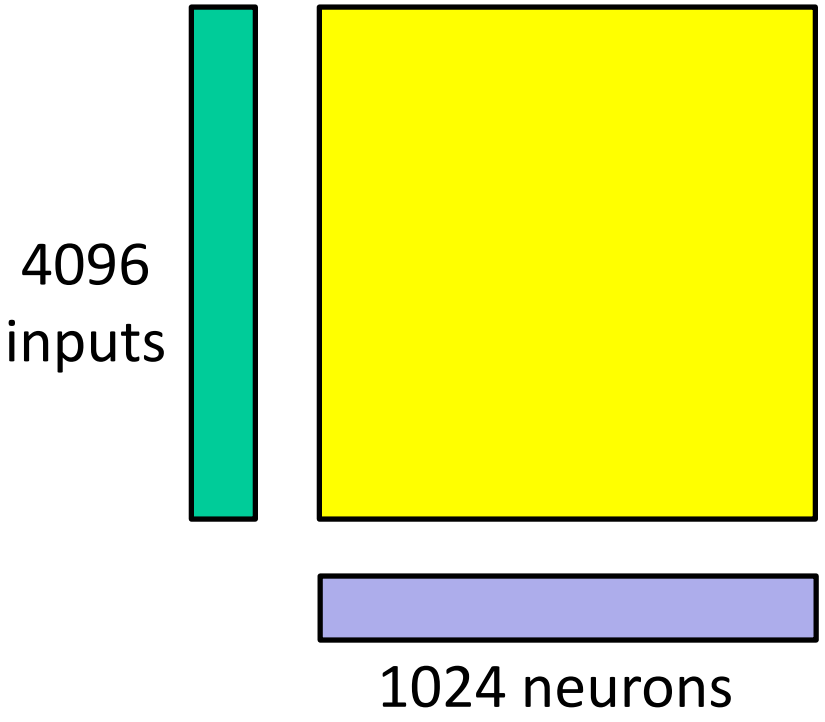
Tiling to Reduce Data Fetches



So all we can do is re-organize the access pattern so that the input vector reuse can exploit the cache. We break the input vector into a small tile that fits in cache (light green, shown on the right) and multiply it with a subset of the first column. This produces a partial result for the first neuron. Next, we apply that small tile to a subset of the second column. And we repeat this for all the columns. Then, we carve out the next small tile of the input vector and apply it to subsets of all the columns. Each small tile creates an increment for the neuron. So the neuron is now written multiple times (instead of once as before). But the input vector is only fetched once from memory. Depending on the sizes of the row, column, tile, and cache, this may be a worth-while trade-off. As before, the synaptic weights are fetched exactly once.

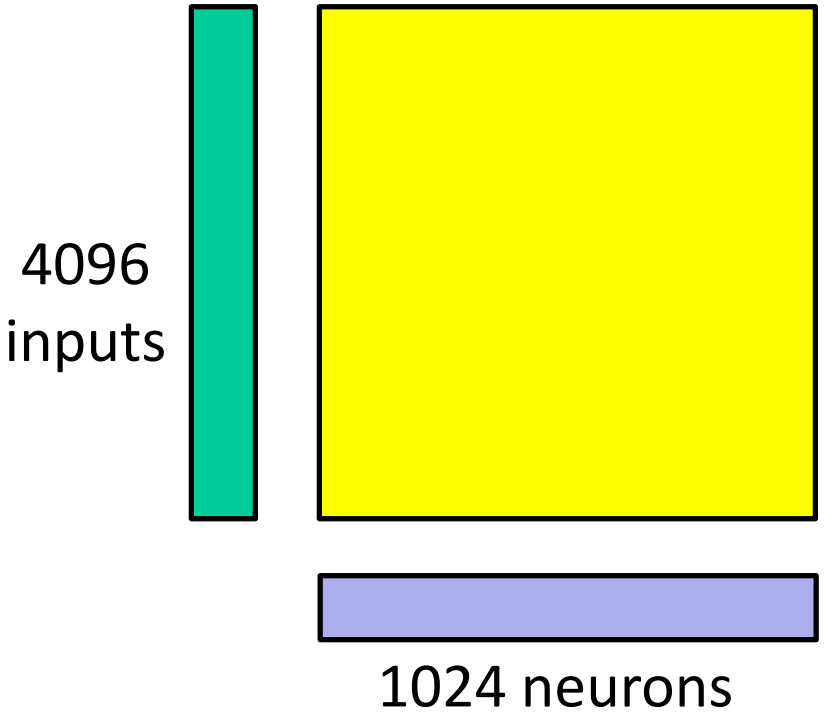
Tiling Example

Cache size of 256



Tiling Example

Cache size of 256



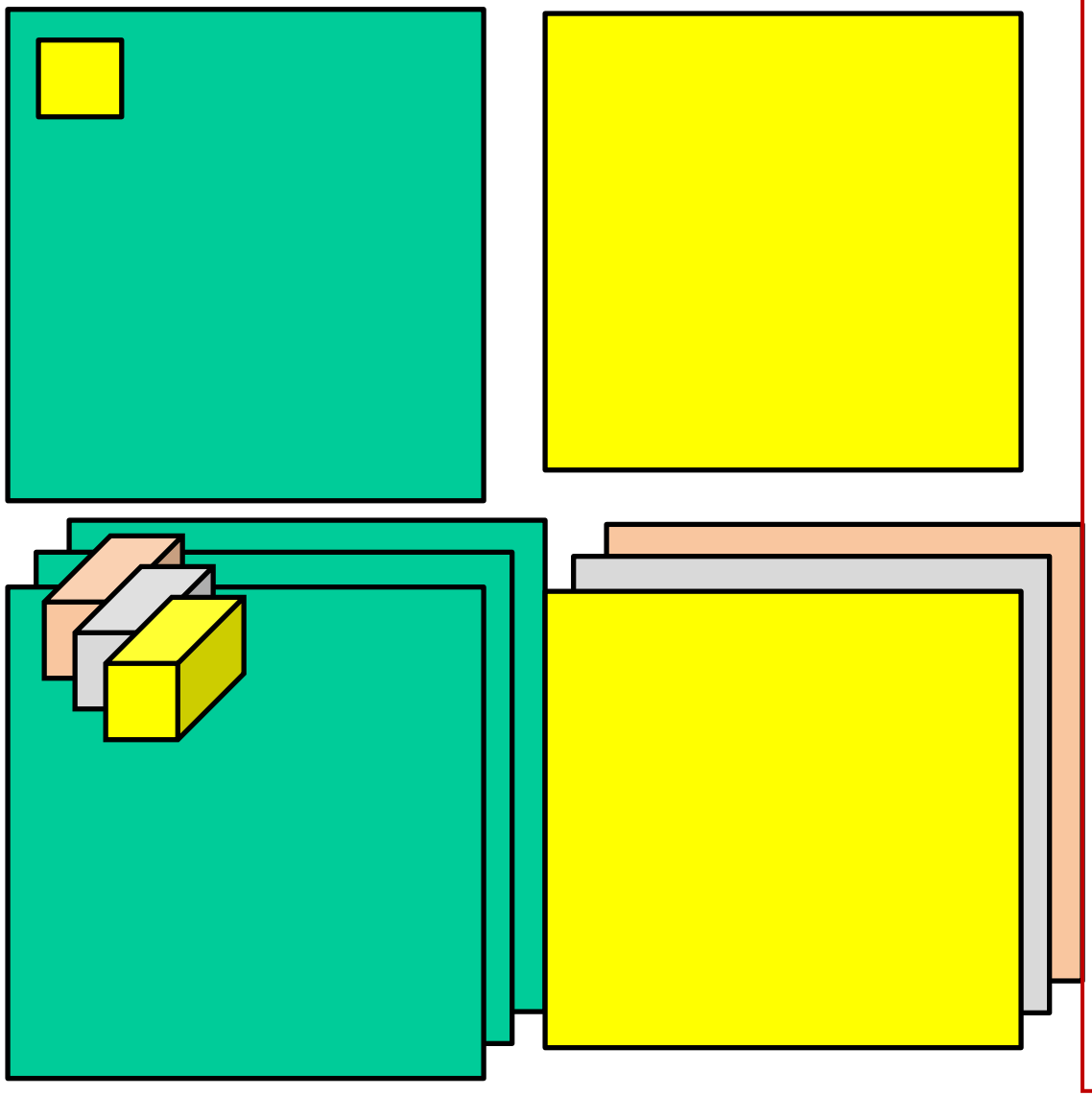
Baseline:
Total fetch = 4M (i) + 1K (n) + 4M (s)

Tile 256 inputs:
Must fetch 1024 x 4096/256 neurons
Total fetch = 4K (i) + 16K (n) + 4M (s)

Tile 128 inputs and 128 neurons:
Total fetch = 32K (i) + 1K (n) + 4M (s)

As we can see from this example, tiling can reduce the bandwidth requirement of the classifier layer by ~2X. You could try more tiling tricks (e.g., see the last row above), but at this point, the weight matrix is the bottleneck and this cannot be fixed. The only way to improve reuse of the weight matrix is to fetch multiple images (batching) and process them all in parallel, essentially converting our vector-matrix multiplication into a matrix-matrix multiplication. That is, fetch a tile of weights and use it N times to process N different input images.

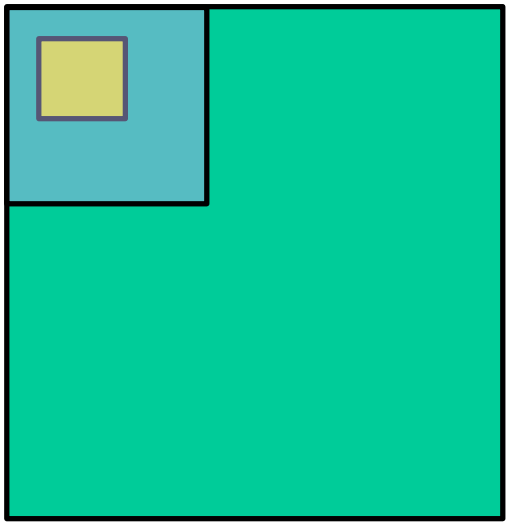
Access Pattern – Convolution Layer

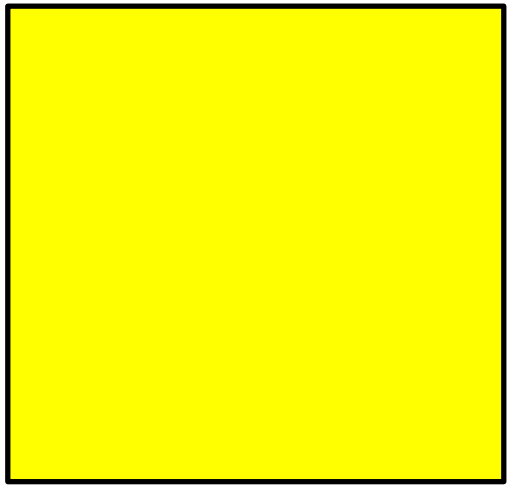


Now let's consider the convolution layer. Note that the 3D kernel below turns out to just be an over-sized version of the 2D kernel example. For simplicity, let's just look at the 2D kernel example. The kernel is relatively small and let's say it fits in cache (if it does not, then tiling the kernel will help too). The output image (the large yellow block on the right) is written exactly once. The green input image pixels do exhibit reuse. If we assume a 5x5 kernel that keeps shifting by one unit, an input pixel is involved in 25 different convolutions. 20 of those will be cache hits because once fetched, a pixel is involved in 5 consecutive convolutions. But every input pixel is fetched 5 times from memory, which is expensive.

Tiling – Convolution Layer

2x2 kernel





4096 inputs

3969 neurons

Cache size = 64

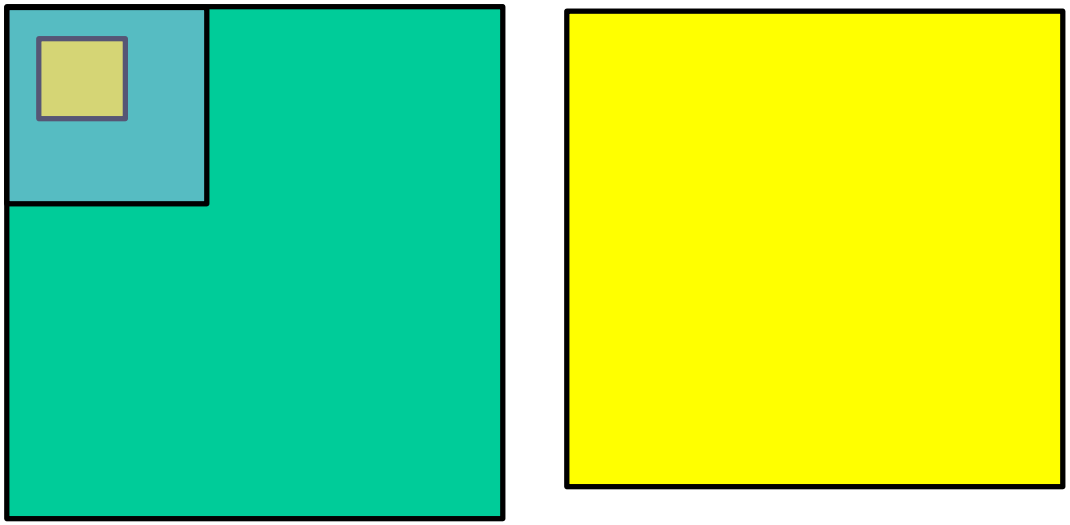
What is the tile size?

Data fetched?

To reduce this overhead, we'll tile the input image. The blue tile on the left is large enough to fit in cache. Once that tile has been brought in, we'll apply convolutions to maximize reuse of that tile. In the example here, we bring in an 8x8 input image tile so it fits in the 64 entry cache. We then apply the 2x2 kernel 49 times within this tile before moving on to the next tile. So we fetch 64 input pixels to process 49 convolutions. We are fetching 3969 x 64/49 input pixels and writing 3969 neuron values. In the baseline, each of the 4K inputs is fetched twice. So tiling does help reduce convolution bandwidth requirements.

Tiling – Convolution Layer

2x2 kernel



4096 inputs

3969 neurons

Cache size = 64

What is the tile size? 8x8

Data fetched? Baseline = 8K (i) + 4K (n)

Tiling = 4K x 64/49 (i) + 4K (n)

DianNao Intro

It's all about memory management! DianNao uses tiling to reduce memory bandwidth needs. Beyond this, DianNao does little to alleviate the memory bottleneck. The follow-up architecture, DaDianNao does a much better job addressing the memory bottleneck.

- Deep learning: many layers, large amounts of data moving between layers, many synaptic weights
- Most prior work did little to address the high cost of bringing data to/from memory – DianNao uses tiling, buffering, prefetching to reduce these costs – DaDianNao does even better
- Focus only on inference for now
- Programmable, so it can adapt to algorithm tweaks

Spatially Unfolded Design

- Implement the full neural network with dedicated latches, multipliers, adders, and sigmoid units
- Reasonable for small networks, e.g., 90-10-10 network can be implemented with 974x lower energy than a general-purpose core

One way to implement a neural network is to have dedicated logic for every neuron. In this figure, we see a register for every weight. The weight and the input are fed to a multiplier. An adder sums up the outputs of the multipliers. That result then goes to a look-up table that implements the activation function (more on this later). The activation result then feeds the multipliers of the next neuron layer.

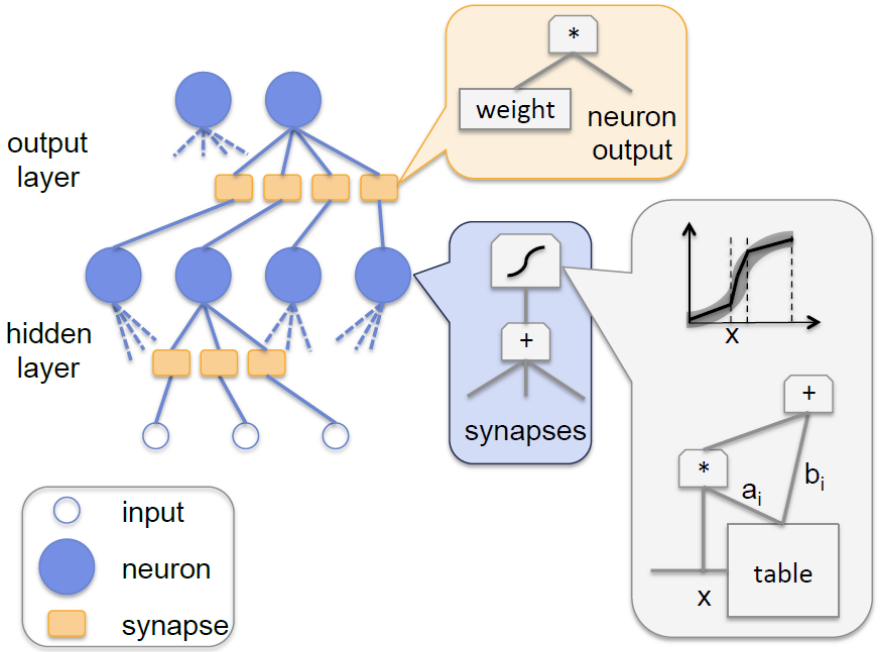


Figure 9. *Full hardware implementation of neural networks.*

Spatially Unfolded Design

- Implement the full neural network with dedicated latches, multipliers, adders, and sigmoid units
- Reasonable for small networks, e.g., 90-10-10 network can be implemented with 974x lower energy than a general-purpose core

This is a very expensive way to implement a neural network. The graph on the next slide shows how the implementation overheads increase more than linearly with the number of neurons. Even implementing a small 90-10-10 network will take up most of the chip area. But they do consume significantly lower energy than a general-purpose core.

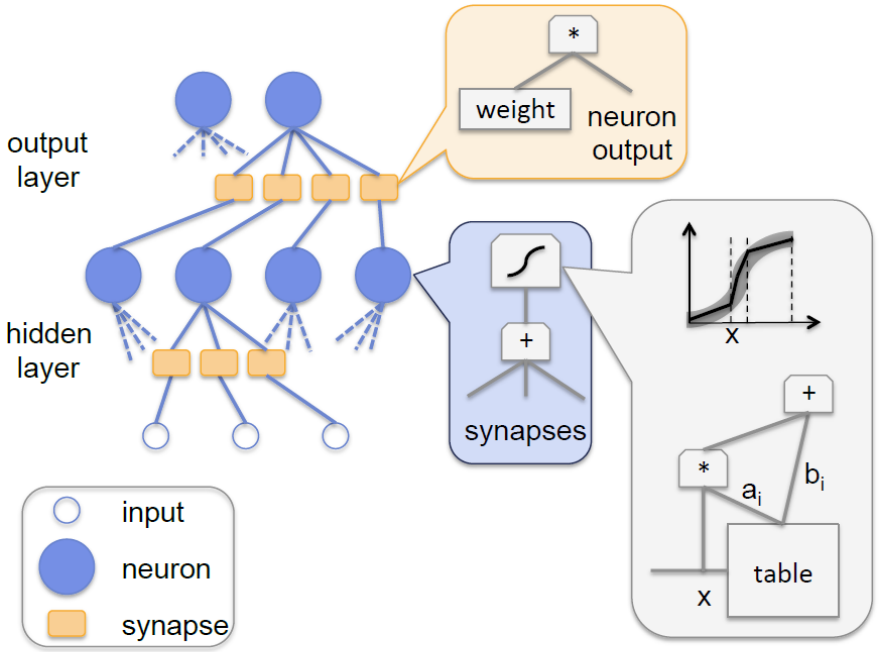


Figure 9. *Full hardware implementation of neural networks.*

Scalability

Spatially unfolded approach does not scale well

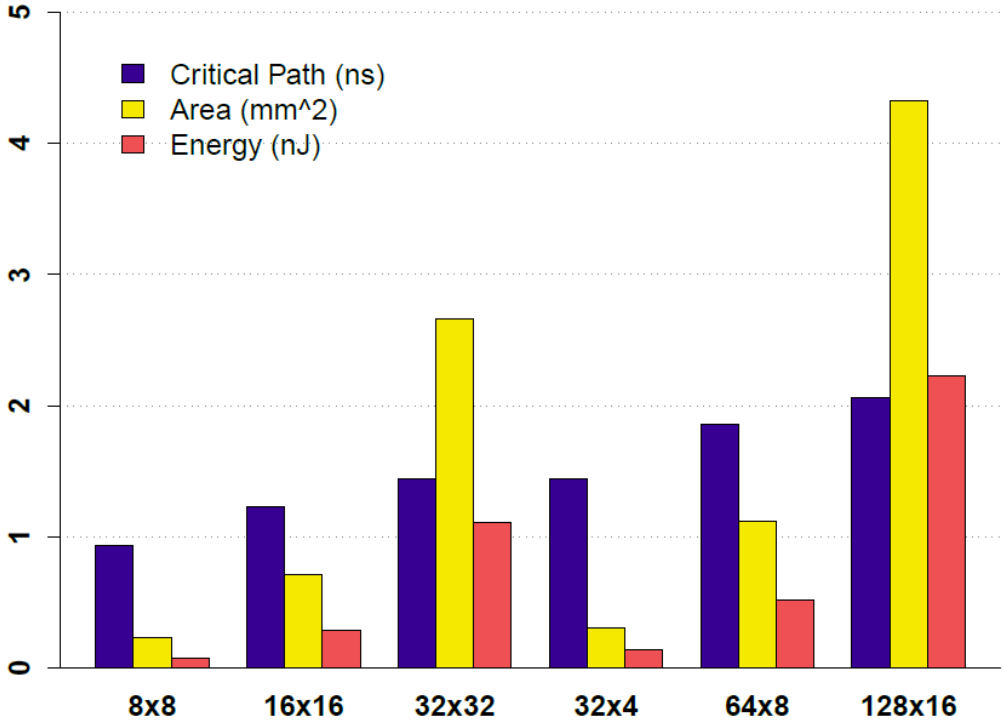


Figure 10. *Energy, critical path and area of full-hardware layers.*

DianNao Approach

- Place data in memory and prefetch “tiles” into buffers
- Each data type has a different requirement, so implement multiple buffers
- The ALUs must be time-multiplexed across several neurons

DianNao uses a “folded” approach, where a few hardware units are used by several neurons in a time-multiplexed manner. Since the bottleneck is the access of large sets of inputs/weights/outputs, DianNao uses a set of buffers to bring in the required elements. Tiling is used to maximize reuse of the data that is brought into these buffers. There are separate buffers for inputs, weights, and outputs since each has a different access pattern.

DianNao implements a single Neural Functional Unit (NFU) shown on the next slides. The NFU has the three buffers mentioned above (purple), the computational unit (light blue), and control units (yellow). The NFU is set up to handle 16 neurons in parallel. Each of these neurons can handle 16 inputs at a time. If a neuron has more than 16 inputs, its result is computed over many cycles and partial sums are aggregated.

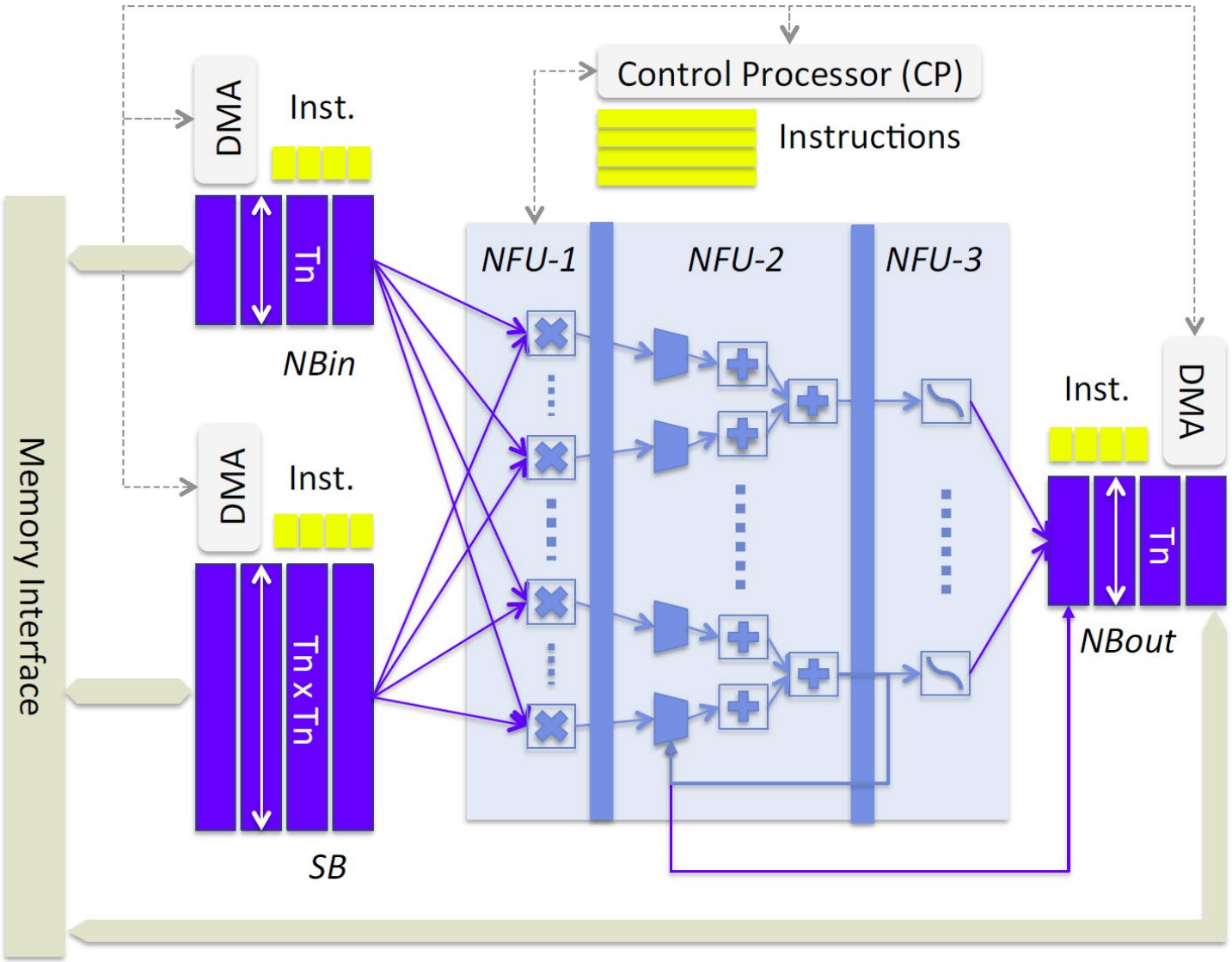
Reasoning about Performance, Power

The computational unit has 256 parallel multipliers (to handle 16 inputs for 16 neurons). This is followed by 16 adder-trees, one per neuron. Each tree has 15 adders to aggregate the results of 16 multipliers. The sum-of-products is re-used in the next cycle if we have more inputs to aggregate. If we're done, the sum-of-products is sent to the activation function. The activation function is implemented with piecewise linear interpolation, i.e., the non-linear function is split into 16 linear segments. A look-up table tracks the end-points of each segment and allows us to quickly apply any user-defined activation function. These units are implemented as an 8-stage pipeline.
The NBin buffer supplies the 16 neuron inputs. The SB buffer supplies the 256 synaptic weights. The NBout buffer receives the 16 neuron outputs. This also specifies the width/wiring for each buffer.
The Control Unit spits out instructions to advance the computations in every cycle. The authors use tiling and prefetching to hide the memory bottleneck, but I don't believe that's actually happening. It may help in the convolutional layers, where a tile is brought into NBin and then reused several times. Although, note that for every reuse, the inputs to the multipliers have to be shuffled around – the authors address that in a later paper. But tiling and prefetching provide no help for the final classifier layers where the weights are never re-used and 256 new weights have to be provided every cycle. Fetching these 256 weights every cycle is a critical bottleneck and will overwhelm the memory system. This requires 512 B/cycle, roughly 512 GB/s. A modern DDR3/DDR4 memory channel provides only ~16GB/s. So we would need 32 memory channels to feed the SB (today's best processors only have a handful of memory channels)!

DianNao

What this means is that the memory system is the clear bottleneck for the classifier layer and designing a fancy accelerator has zero impact. Even the convolution layer needs 2-3 memory channels worth of bandwidth. This is why I said that DianNao is not a significant contribution (although, they do deserve great credit for identifying a very important problem and putting forth the first solution) – DaDianNao does a much better job addressing the real bottleneck.
Similarly, the impact on power is also low because most of the power is in memory accesses and we're doing little to address the power for weight accesses (Amdahl's Law).

DianNao


Control Processor (CP)
Instructions
DMA
Inst.
Tn
NBin
DMA
Inst.
Tn x Tn
SB
NFU-1
NFU-2
NFU-3
Inst.
DMA
Tn
NBout
Memory Interface

Design Summary

- Different bit widths for each buffer (Tn = 16, 64 entries)
- Separate “scratchpads” avoid tags/unpredictability/conflicts, and improve latency, parallelism
- 256 parallel multiplies (16 neurons, each with 16 inputs)
- 16 parallel accumulates (each is a tree with 15 adders)
- Partial sums if necessary
- Sigmoid (or any function) is implemented with piecewise linear interpolation – 16 hard-wired segments and coefficients can be programmed into a small table
- Control processor has a number of instructions that specify how data is loaded/accessed in buffers
- Memory bandwidth/energy bottleneck and Amdahl’s Law

Fixed Point Arithmetic

- DianNao uses 16b fixed-point arithmetic; much more efficient than 32b floating-point arithmetic, and little impact on accuracy

Type	Area (μm^2)	Power (μW)
16-bit truncated fixed-point multiplier	1309.32	576.90
32-bit floating-point multiplier	7997.76	4229.60

Table 2. *Characteristics of multipliers.*

Many studies have shown that neural networks can tolerate low precision in the weights and inputs. This study shows that you can reduce area/power by ~7x if you went from 32b floating-point math to 16b fixed-point math. The graph on the next slide shows that the impact on accuracy is also small (at least for small-scale workloads).
As a reminder, fixed-point represents each number as an integer (or perhaps with a binary point in some fixed position, but that has no impact on the arithmetic) and there is an implied scaling factor.

Fixed Point Error Rates

Type	Error Rate
32-bit floating-point	0.0311
16-bit fixed-point	0.0337

Table 1. *32-bit floating-point vs. 16-bit fixed-point accuracy for MNIST (metric: error rate).*

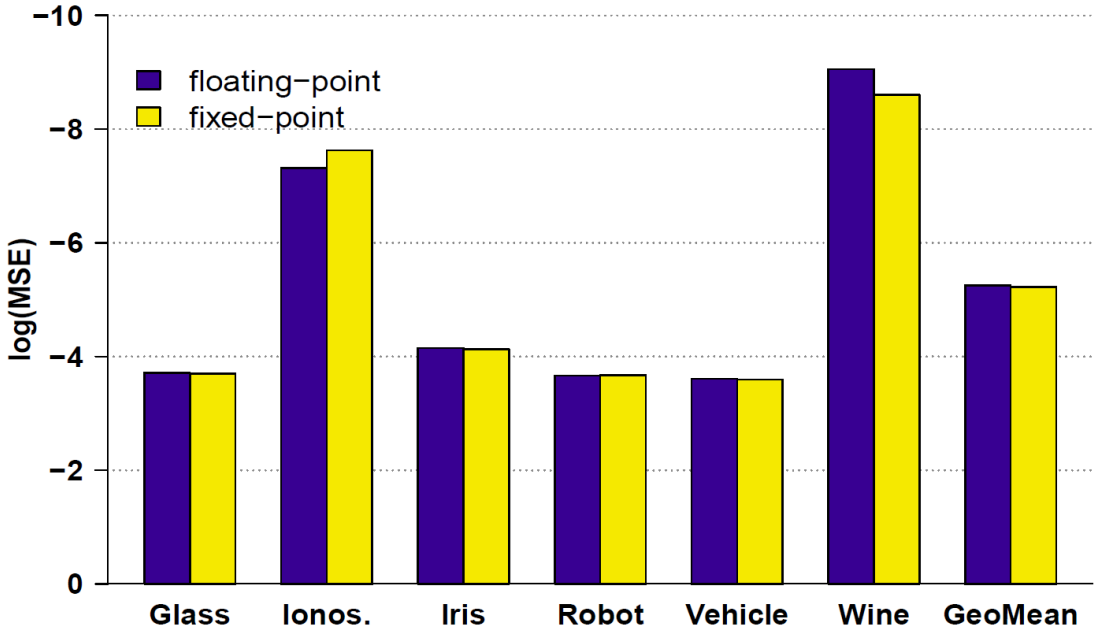
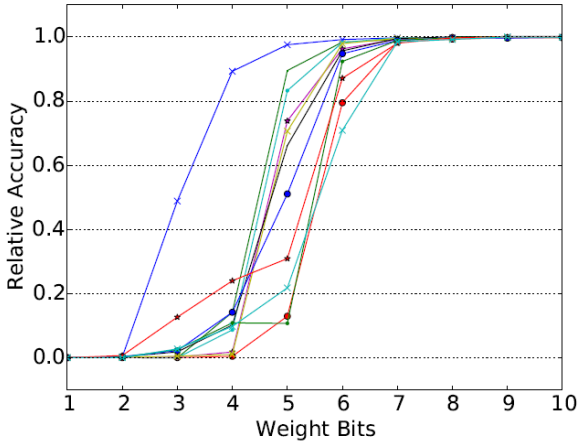


Figure 12. *32-bit floating-point vs. 16-bit fixed-point accuracy for UCI data sets (metric: log(*Mean Squared Error*)).*

Precision Analysis for DNNs

Network	Data (Per Layer)	Weights (Uniform)
LeNet[12]	2,4,3,3	7
Convnet[13]	8,7,7,5,5	9
AlexNet[14]	10,8,8,8,8,8,6,4	10
NiN[15]	10,10,9,12,12,11,11,11,10,10,10,9	10
GoogLeNet[4]	14,10,12,12,12,12,11,11,11,10,9	9

TABLE I: Minimum precision, in bits, for data and weights for a set of neural networks.



(m) GoogLeNet: Weights

Min bits required per layer for accuracy within 1%
Source: Proteus, Judd et al., WAPCO'16

Implementation Details/Results

- Cycle time of 1.02ns, area of 3mm2, 485mW power
- The NFU is composed of 8 pipeline stages
- Peak activity is nearly 500 GOP/s
- 44KB of RAM capacity
- Buffers are about 60% of area/power, while NFU is ~30%
- Energy is 21x better than a SIMD baseline; this is limited because of the high cost of memory accesses
- Big performance boosts as well: higher computational density, tiling, prefetching

Benchmarks

Layer	N_x	N_y	K_x	K_y	N_i	N_o	Description
CONV1	500	375	9	9	32	48	Street scene parsing
POOL1	492	367	2	2	12	-	(CNN) [13], (e.g.,
CLASS1	-	-	-	-	960	20	identifying "building", "vehicle", etc)
CONV2*	200	200	18	18	8	8	Detection of faces in YouTube videos (DNN) [26], largest NN to date (Google)
CONV3	32	32	4	4	108	200	Traffic sign
POOL3	32	32	4	4	100	-	identification for car
CLASS3	-	-	-	-	200	100	navigation (CNN) [36]
CONV4	32	32	7	7	16	512	Google Street View house numbers (CNN) [35]
CONV5*	256	256	11	11	256	384	Multi-Object
POOL5	256	256	2	2	256	-	recognition in natural images (DNN) [16], winner 2012 ImageNet competition

Table 5. *Benchmark layers (CONV=convolutional, POOL=pooling, CLASS=classifier; CONVx* indicates private kernels).*

DianNao Conclusions

- Tiling to reduce memory traffic
- Efficient NFU and buffers to reduce energy/op and prefetch
- Even with these innovations, memory is the bottleneck, especially in a small accelerator with few pins
- For example, each classifier layer step needs 256 new synaptic weights (512 bytes), while 4 memory channels can only bring in 64 bytes per cycle
- Energy consumed by the DianNao pipeline per cycle = 500pJ
- Energy per cycle for fetching 64 bytes from memory = 35nJ

  (70 pJ/b for DDR3 at 100% utilization, Malladi et al., ISCA'12)

It's all about the memory bandwidth/energy !!!

The GPU Option

- Modest (but adequate) memory capacities (a few giga-bytes)
- High memory bandwidth, e.g., 208 GB/s (NVIDIA K20M)
- But, high compute-to-cache ratio on the chip
- Therefore, average GPU power of ~75 W, plus expensive memory accesses ➔ GPU card TDP of ~225W
- A GPU out-performs DianNao by ~2X

GPUs are the current platforms for machine learning. They are great in terms of memory bandwidth and in their SIMD-ness. This also means high power (GPUs are still "general-purpose", so more customization will help). DianNao is 2x slower, but the DianNao paper is not clear about memory bandwidth assumptions (we know that is key to this comparison).

References

- “DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine Learning”, T. Chen et al., Proceedings of ASPLOS, 2014