Lecture 11:

Optimizing Inference via Approximation

Visual Computing Systems
CMU 15-769, Fall 2016

Take-home midterm

- To be released Sunday morning 10/23. Hand-in noon Tuesday 10/25.

- Example forms of questions:
  - Short answer questions
    - Design exercise: choose between processor A and processor B for a particular task and state reason
  - Read a paper (similar to ones we've read in class) and offer an analysis of it (or answer a few questions about it)

Assignment 2 coming out next week

- We have finished our "basics of deep learning" prep, now time to get hands on

- Assignment 2: implement a mini-deep learning framework in any programming language you wish
  - Released Tuesday 10/18, due Friday 11/4
  - We will give you basic stater code in C and in Halide
  - You will implement basic layers needed to train a mini-network
  - We will have a "fun" class race on fastest forward evaluation and backward evaluation

Course projects

- **Project proposal is due 10/27**
  - Short proposal document (2-3 pages max)
  - I will likely accept any project that "embodies the themes of the course" and passes the "will the project answer a question that a student in the class not know the answer to" test
  - For Ph.D. students, this means the idea is ambitious enough to be the seed for a research project
- **Project presentations are due during our final exam slot: 12/16**
- **May work in teams of 2 (in rare circumstances I'll okay a team of 3)**
- **I'll post a list of rough project ideas on the web over the weekend**

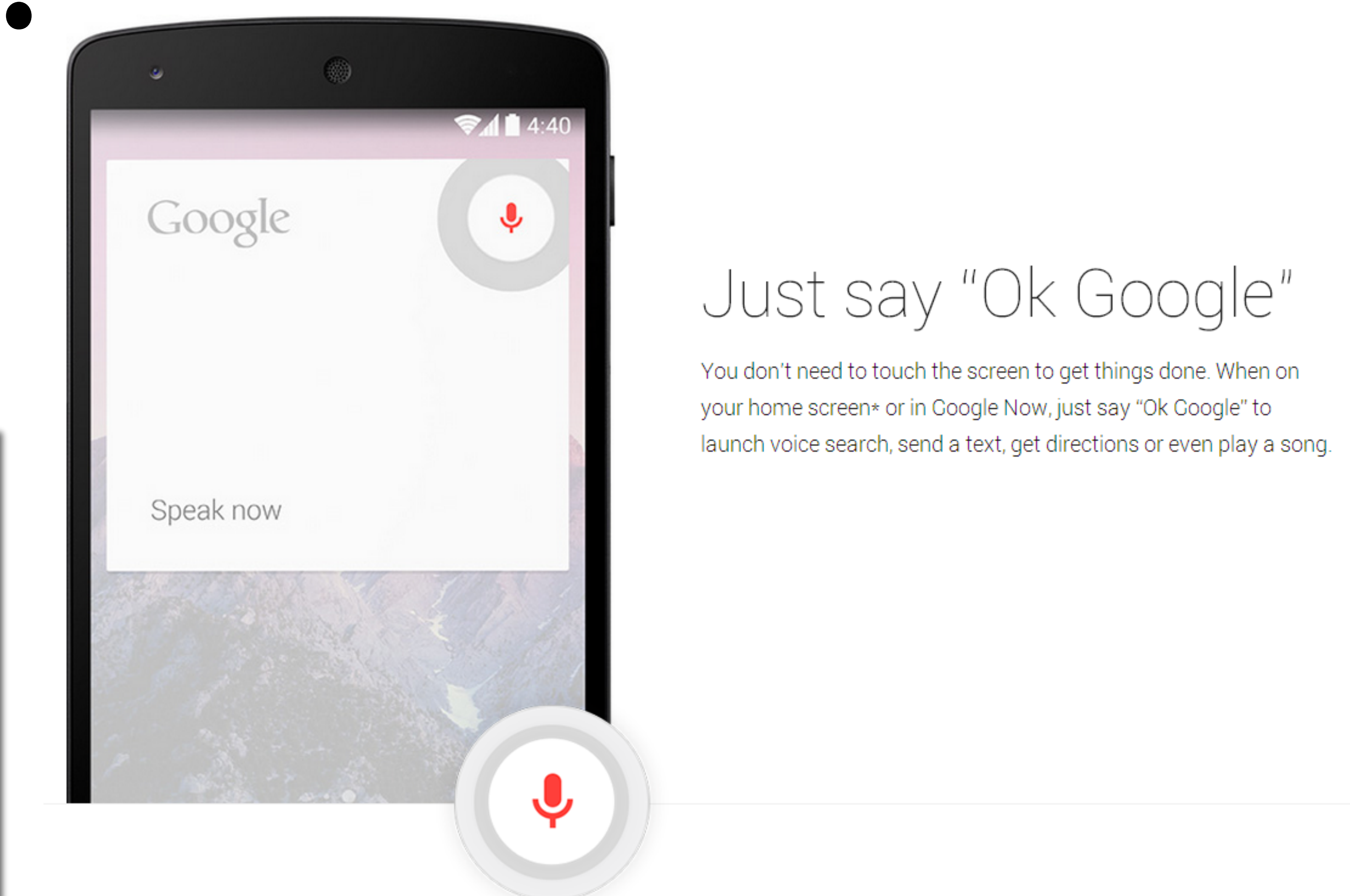
Three ideas for accelerating inference

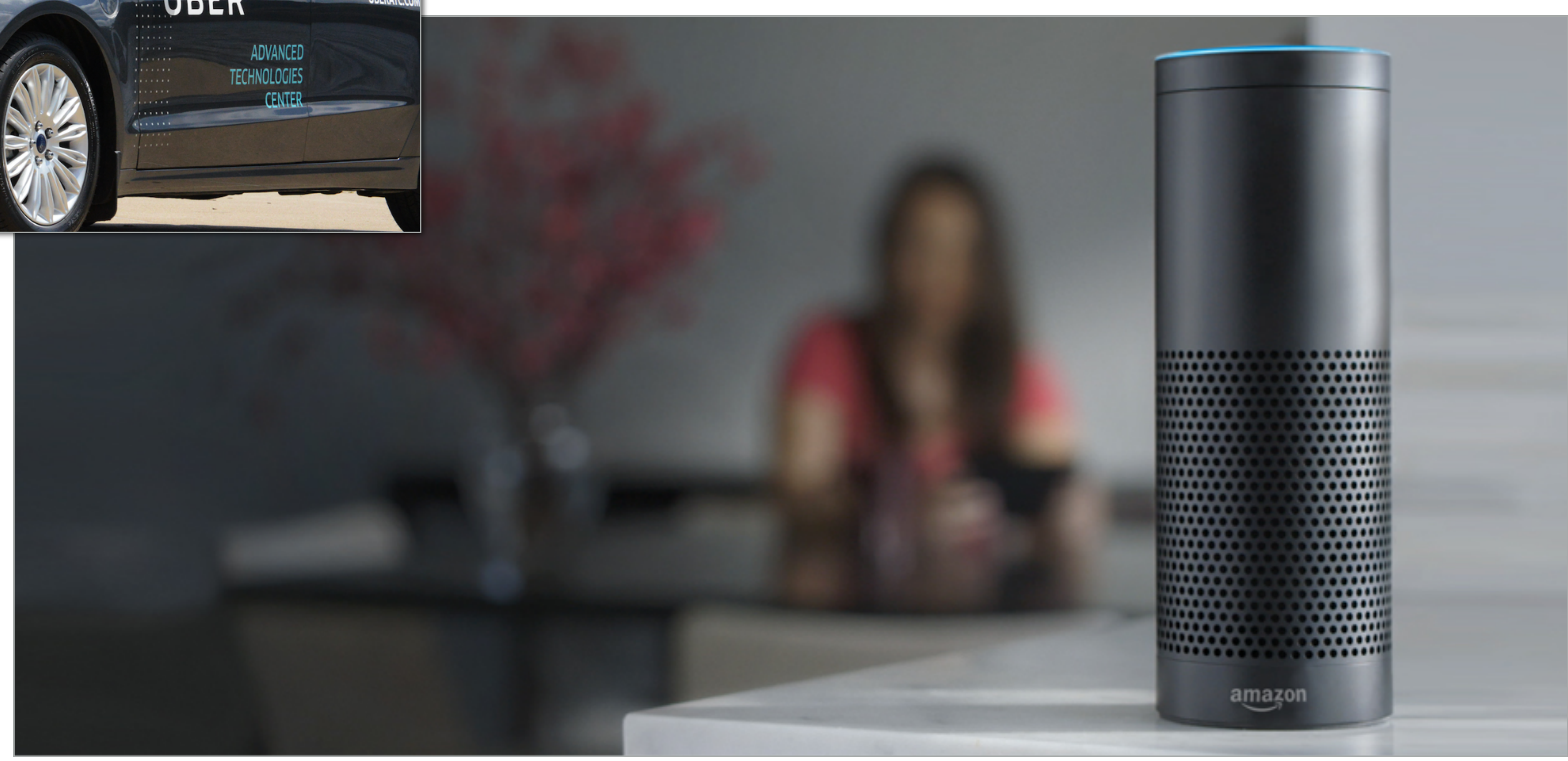
- **Sparsification**
  - Remove unnecessary parts of DNN

- **Quantization**
  - Represent values that remain more coarsely

- **Exploiting temporal redundency (for video)**
  - Skip evaluating parts of a DNN that do not change significantly with time

Why efficient inference?





Last time (and last class' reading)

- Expert knowledge used to manually design more compact and more efficient (to train and to evaluate) network topologies

Finding a good network topology

- Common practice in modern network design: brute-force parameter sweep
- Train multiple versions of a topology in parallel
  - Vary number of layers, width fully-connected layers
  - Vary number of filters in a cone layer
- Pick the smallest (most efficient) topology that yields adequate performance (accuracy) on task at hand

Reminder: energy cost of data access

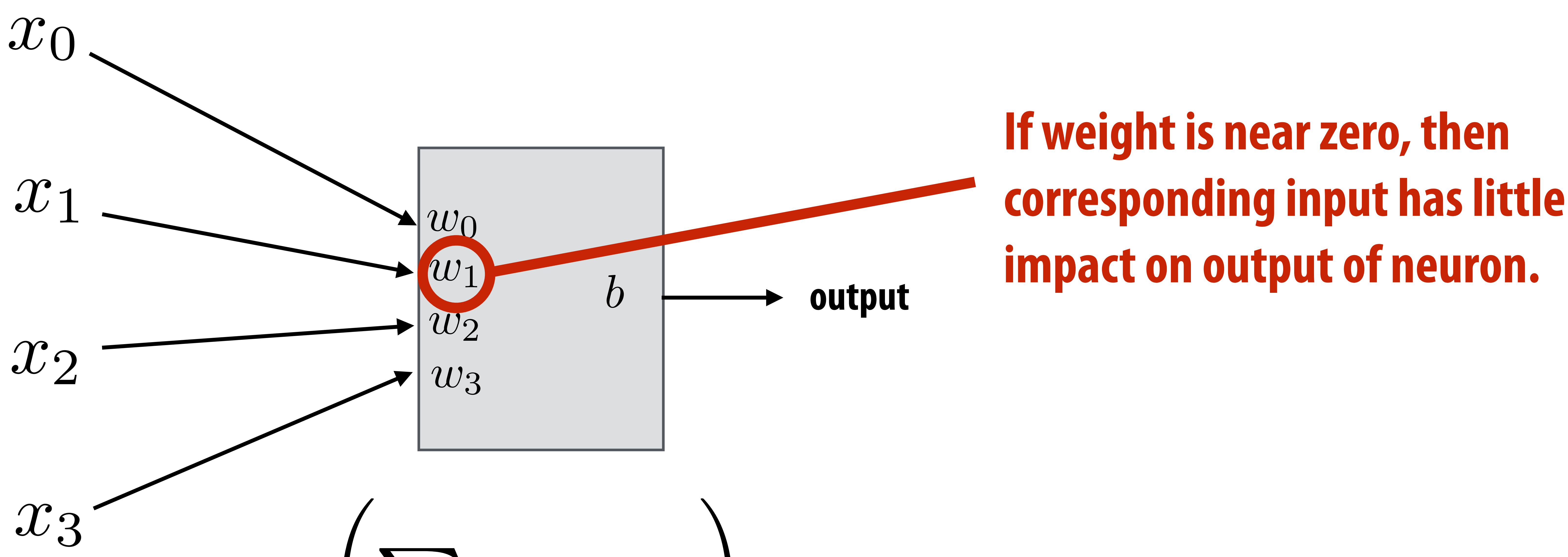
Significant fraction of energy expended moving data to processor ALUs

Operation	Energy [pJ]	Relative Cost
32 bit int ADD	0.1	1
32 bit float ADD	0.9	9
32 bit Register File	1	10
32 bit int MULT	3.1	31
32 bit float MULT	3.7	37
32 bit SRAM Cache	5	50
32 bit DRAM Memory	**640**	**6400**

Estimates for 45nm process
[Source: Mark Horowitz]

Recall: AlexNet has over 68m weights (>260MB if 4 bytes/weight)
Executing at 30fps, that's 1.3 Watts just to read the weights

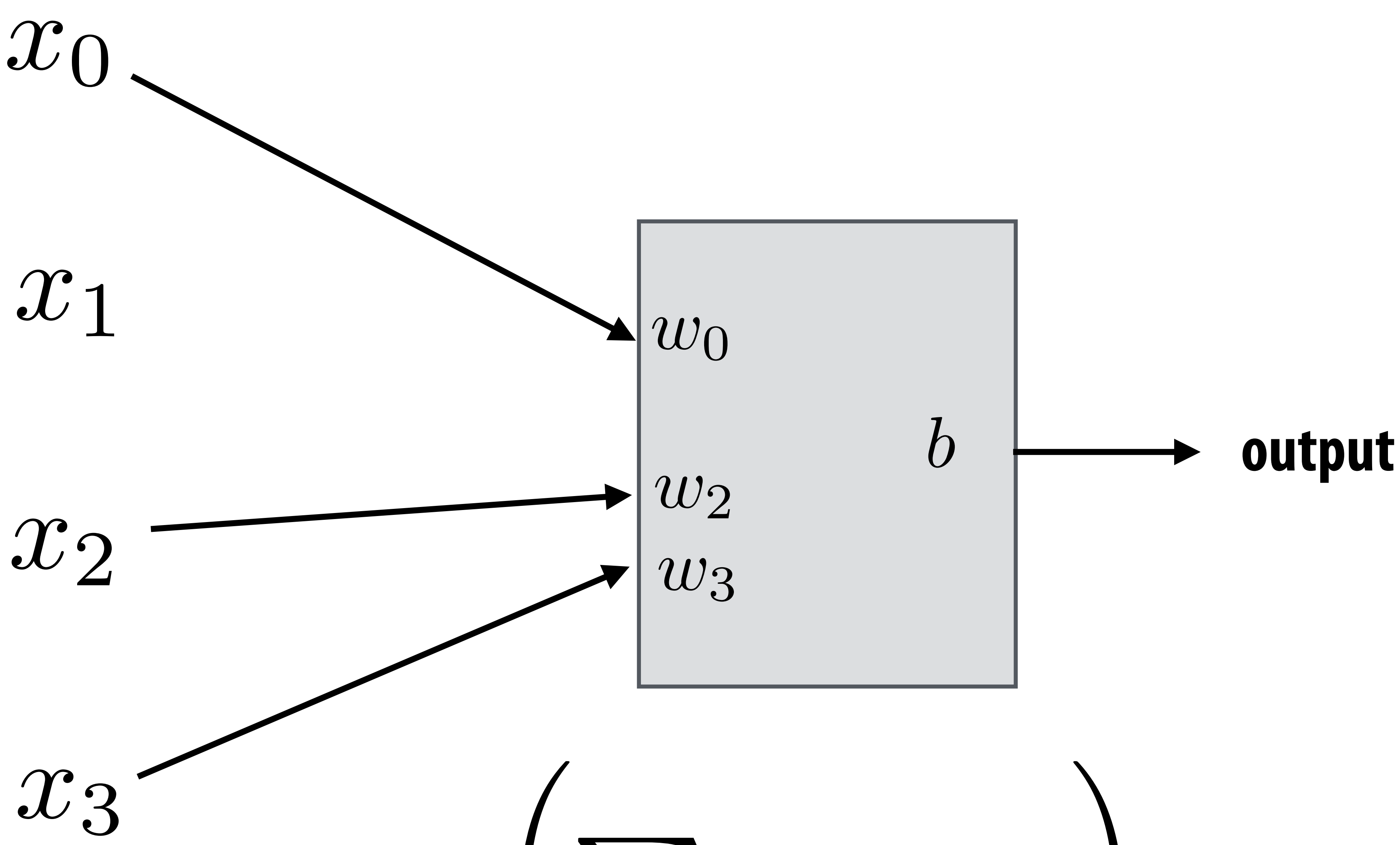
“Pruning” (sparsifying) a network



$$f\left(\sum_i x_i w_i + b\right)$$

$$f(x) = max(0, x)$$

"Pruning" (sparsifying) a network



$$f\left(\sum_i x_i w_i + b\right)$$

$$f(x) = max(0, x)$$

Idea: prune connections with near zero weight

Remove entire units if all connections are pruned.

Why not look for large dL/dw_i?

More principled to look at second derivatives d^2L/dw_idw_j, but costly…

Iterative pruning

[Han 2015]

- Step 1: train network (as normal)
- Step 2: remove connections with weight less than threshold (threshold may be chosen based on std-dev of weights)
  - The network is now sparse
- Step 3: retrain network using surviving connections
  - Retraining (fine-tuning) is initialized with previously learned weights
- Repeat steps 2 and 3 as necessary
  - Since a connection is gone forever once pruned, best to prune conservatively each step, use multiple iterations to remove connections
  - L2 regularization during training

Iterative process learns both topology (what connections are needed) and the weights on those connections.

Incur increased training time to accelerate inference.
(recall design of networks like Inception sought to accelerate training as well)

Results of pruning

AlexNet:

Layer	Weights	FLOP	Act%	Weights%	FLOP%
conv1	35K	211M	88%	84%	84%
conv2	307K	448M	52%	38%	33%
conv3	885K	299M	37%	35%	18%
conv4	663K	224M	40%	37%	14%
conv5	442K	150M	34%	37%	14%
fc1	38M	75M	36%	9%	3%
fc2	17M	34M	40%	9%	3%
fc3	4M	8M	100%	25%	10%
Total	61M	1.5B	54%	**11%**	**30%**

VGG-16:

Layer	Weights	FLOP	Act%	Weights%	FLOP%
conv1_1	2K	0.2B	53%	58%	58%
conv1_2	37K	3.7B	89%	22%	12%
conv2_1	74K	1.8B	80%	34%	30%
conv2_2	148K	3.7B	81%	36%	29%
conv3_1	295K	1.8B	68%	53%	43%
conv3_2	590K	3.7B	70%	24%	16%
conv3_3	590K	3.7B	64%	42%	29%
conv4_1	1M	1.8B	51%	32%	21%
conv4_2	2M	3.7B	45%	27%	14%
conv4_3	2M	3.7B	34%	34%	15%
conv5_1	2M	925M	32%	35%	12%
conv5_2	2M	925M	29%	29%	9%
conv5_3	2M	925M	19%	36%	11%
fc6	103M	206M	38%	4%	1%
fc7	17M	34M	42%	4%	2%
fc8	4M	8M	100%	23%	9%
total	138M	30.9B	64%	**7.5%**	**21%**

Recall similar numbers from SqueezeNet paper: 33% weights survive (recall SqueezeNet is fully convolutional… see convlayers above)

Efficiently storing the surviving connections

Store surviving weights compactly in compressed sparse row (CSR) format

```
Indicies    1    4   9  ...
Value     1.8  0.5  2.1
```

0	1.8	0	0	0.5	0	0	0	0	1.1	...

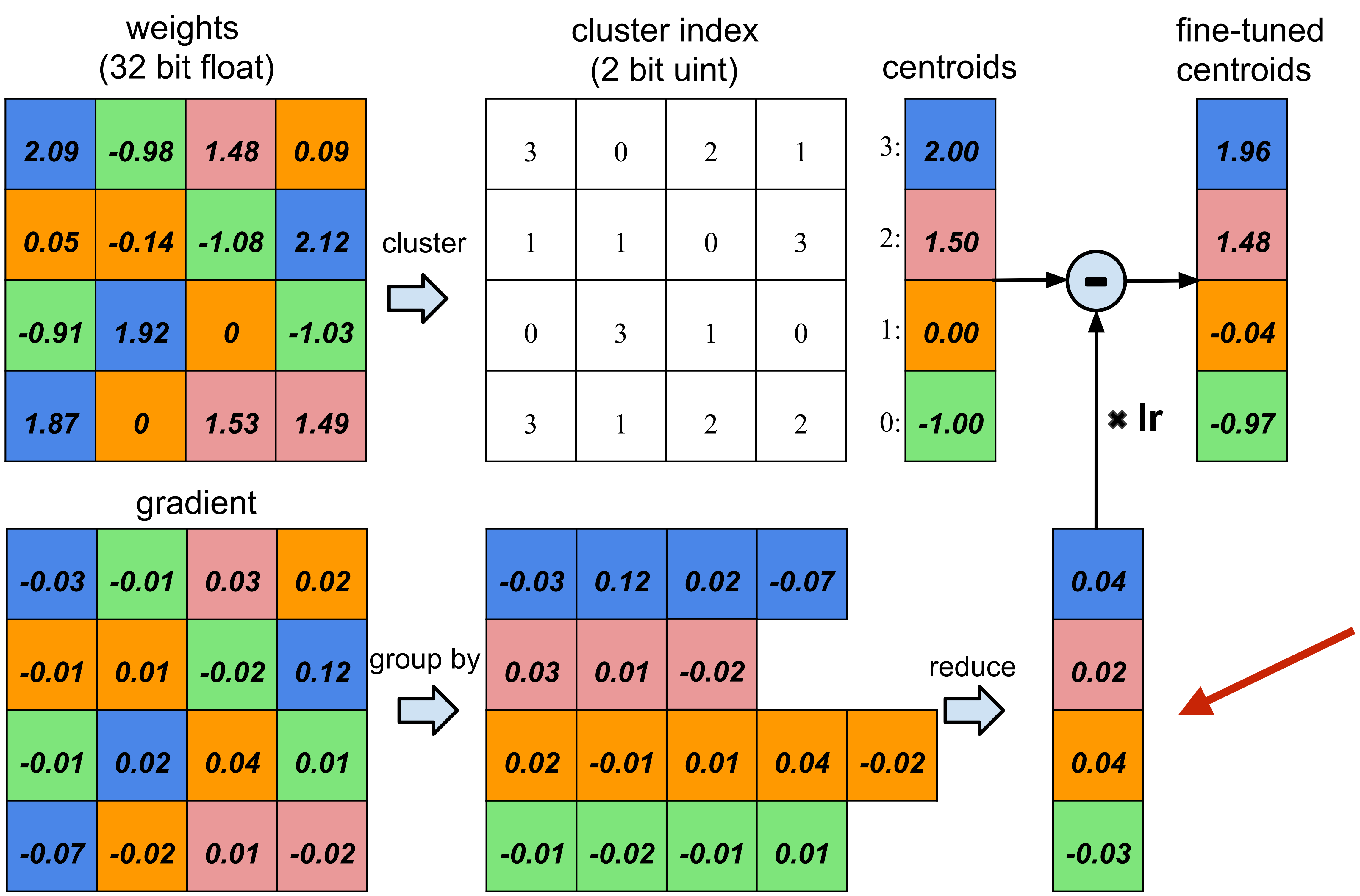
Reduce storage over head of indices by delta encoding them to fit in 8 bits

```
Indicies    1    3   6  ...
Value     1.8  0.5  2.1
```

Efficiently storing the surviving connections

Weight sharing: make surviving connections share a small set of weights

- Cluster weights via k-means clustering
- Compress weights by only storing index of assigned cluster (lg(k) bits)
- Retraining only modifies the cluster centroids (so that back-prop is constrained to not induce more unique weight values)



Sum weight gradients of all weights in a cluster: use result to update cluster center to avoid divergence of weight values

Pruning, quantization, huffman-encoding of VGG-16

Layer	#Weights	Weights% (P)	Weigh bits (P+Q)	Weight bits (P+Q+H)	Index bits (P+Q)	Index bits (P+Q+H)	Compress rate (P+Q)	Compress rate (P+Q+H)
conv1_1	2K	58%	8	6.8	5	1.7	40.0%	29.97%
conv1_2	37K	22%	8	6.5	5	2.6	9.8%	6.99%
conv2_1	74K	34%	8	5.6	5	2.4	14.3%	8.91%
conv2_2	148K	36%	8	5.9	5	2.3	14.7%	9.31%
conv3_1	295K	53%	8	4.8	5	1.8	21.7%	11.15%
conv3_2	590K	24%	8	4.6	5	2.9	9.7%	5.67%
conv3_3	590K	42%	8	4.6	5	2.2	17.0%	8.96%
conv4_1	1M	32%	8	4.6	5	2.6	13.1%	7.29%
conv4_2	2M	27%	8	4.2	5	2.9	10.9%	5.93%
conv4_3	2M	34%	8	4.4	5	2.5	14.0%	7.47%
conv5_1	2M	35%	8	4.7	5	2.5	14.3%	8.00%
conv5_2	2M	29%	8	4.6	5	2.7	11.7%	6.52%
conv5_3	2M	36%	8	4.6	5	2.3	14.8%	7.79%
fc6	103M	4%	5	3.6	5	3.5	1.6%	1.10%
fc7	17M	4%	5	4	5	4.3	1.5%	1.25%
fc8	4M	23%	5	4	5	3.4	7.1%	5.24%
Total	138M	7.5%(13×)	6.4	4.1	5	3.1	3.2% (**31**×)	2.05% (**49**×)

Problem

- Techniques discussed above significantly reduce network size, but turn a "HW-friendly" dense workload into a sparse one
  - Many heavily-optimized CPU/GPU libraries for dense conv/fc layers
  - Sparse matrix-vector product significantly less efficient on modern processors
- Also incur new costs during inference time for decompression
  - Huffman decode
  - Process relative index offsets
  - Lookup weight from weight id

Sparse, weight-sharing fully-connected layer

$$b_i = ReLU\left(\sum_{j=0}^{n-1} W_{ij}a_j\right)$$

Fully-connected layer:
Matrix-vector multiplication of activation vector a against weight matrix W

$$b_i = ReLU\left(\sum_{j \in X_i \cap Y} S[I_{ij}]a_j\right)$$

Sparse, weight-sharing representation:
I_{ij} = index for weight W_{ij}
S[] = table of shared weight values
X_i = list of non-zero indices in row i
Y = list of non-zero indices in a

Note: activations are sparse due to ReLU

Efficient inference engine (EIE) ASIC

Custom hardware for decode and evaluate sparse, compressed DNNs

Hardware represents weight matrix in compressed sparse column (CSC) format to exploit sparsity in activations:

```
for each nonzero a_j in a:
    for each nonzero M_ij in column M_j:
      b_i += M_ij * a_j
```

More detailed version:

```
int16* a_values;
PTR*   M_j_start;   // column j
int4*  M_j_values;
int4*  M_j_indices;
int16* lookup; // lookup table for
               // cluster values
```

```
for j=0 to length(a):
    if (a[j] == 0) continue;  // scan to nonzero
    col_values = M_j_values[M_j_start[j]];
    col_indices = M_j_indices[M_j_start[j]];
    col_nonzeros = M_j_start[j+1]-M_j_start[j];
    for i=0, i_count=0 to col_nonzeros:
       i += col_indices[i_count]
       b[i] += lookup[M_j_values[i]] *
                       a_values[j_count]
```

*** Keep in mind there's a unique lookup table for each chunk of matrix values**

Parallelization of sparse-matrix-vector product

Stride rows of matrix across processing elements

Output activations strided across processing elements

$$\vec{a} \begin{pmatrix} 0 & 0 & \boldsymbol{a_2} & 0 & a_4 & a_5 & 0 & a_7 \end{pmatrix}$$

$$\times$$

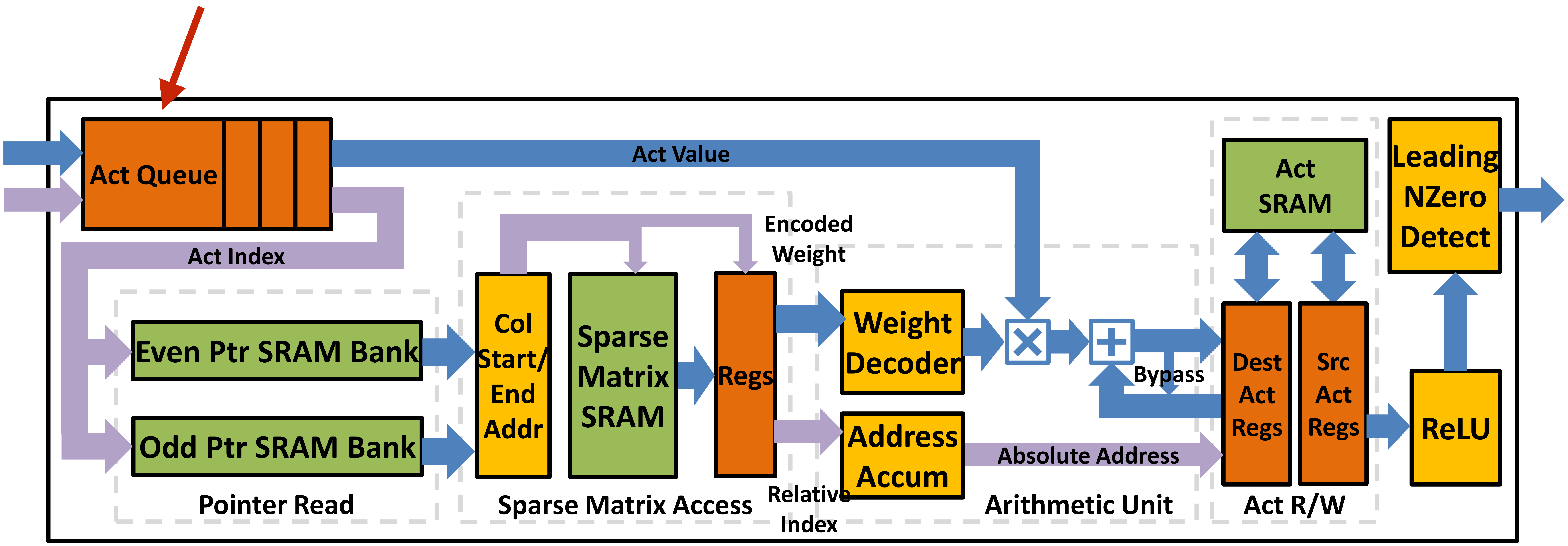
$$\begin{array}{c} PE0 \\ PE1 \\ PE2 \\ PE3 \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \end{array}
\begin{pmatrix}
w_{0,0} & 0 & \boldsymbol{w_{0,2}} & 0 & w_{0,4} & w_{0,5} & w_{0,6} & 0 \\
0 & w_{1,1} & \boldsymbol{0} & w_{1,3} & 0 & 0 & w_{1,6} & 0 \\
0 & 0 & \boldsymbol{w_{2,2}} & 0 & w_{2,4} & 0 & 0 & w_{2,7} \\
0 & w_{3,1} & \boldsymbol{0} & 0 & 0 & w_{0,5} & 0 & 0 \\
0 & w_{4,1} & \boldsymbol{0} & 0 & w_{4,4} & 0 & 0 & 0 \\
0 & 0 & \boldsymbol{0} & w_{5,4} & 0 & 0 & 0 & w_{5,7} \\
0 & 0 & \boldsymbol{0} & 0 & w_{6,4} & 0 & w_{6,6} & 0 \\
w_{7,0} & 0 & \boldsymbol{0} & w_{7,4} & 0 & 0 & w_{7,7} & 0 \\
w_{8,0} & 0 & \boldsymbol{0} & 0 & 0 & 0 & 0 & w_{8,7} \\
w_{9,0} & 0 & \boldsymbol{0} & 0 & 0 & 0 & w_{9,6} & w_{9,7} \\
0 & 0 & \boldsymbol{0} & 0 & w_{10,4} & 0 & 0 & 0 \\
0 & 0 & \boldsymbol{w_{11,2}} & 0 & 0 & 0 & 0 & w_{11,7} \\
w_{12,0} & 0 & \boldsymbol{w_{12,2}} & 0 & 0 & w_{12,5} & 0 & w_{12,7} \\
w_{13,0} & w_{13,2} & \boldsymbol{0} & 0 & 0 & 0 & w_{13,6} & 0 \\
0 & 0 & \boldsymbol{w_{14,2}} & w_{14,3} & w_{14,4} & w_{14,5} & 0 & 0 \\
0 & 0 & \boldsymbol{w_{15,2}} & w_{15,3} & 0 & w_{15,5} & 0 & 0
\end{pmatrix}
=
\overset{\vec{b}}{\begin{pmatrix} b_0 \\ b_1 \\ -b_2 \\ b_3 \\ -b_4 \\ b_5 \\ b_6 \\ -b_7 \\ -b_8 \\ -b_9 \\ b_{10} \\ -b_{11} \\ -b_{12} \\ b_{13} \\ b_{14} \\ -b_{15} \end{pmatrix}}
\overset{ReLU}{\Rightarrow}
\begin{pmatrix} b_0 \\ b_1 \\ 0 \\ b_3 \\ 0 \\ b_5 \\ b_6 \\ 0 \\ 0 \\ 0 \\ b_{10} \\ 0 \\ 0 \\ b_{13} \\ b_{14} \\ 0 \end{pmatrix}$$

Weights stored local to PEs. Must broadcast non-zero a_j's to all PEs

Accumulation of each output b_i is local to PE

EIE unit for quantized sparse/matrix vector product

Tuple representing non-zero activation (a_j, j) arrives and is enqueued



EIE Efficiency

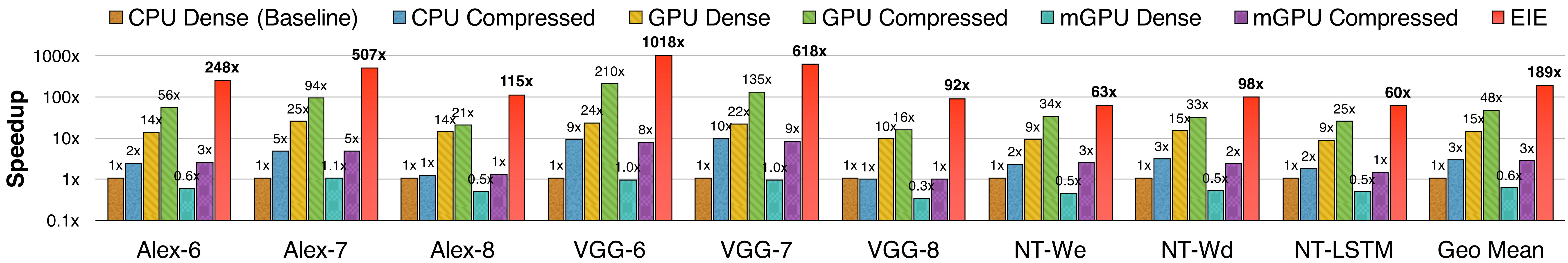
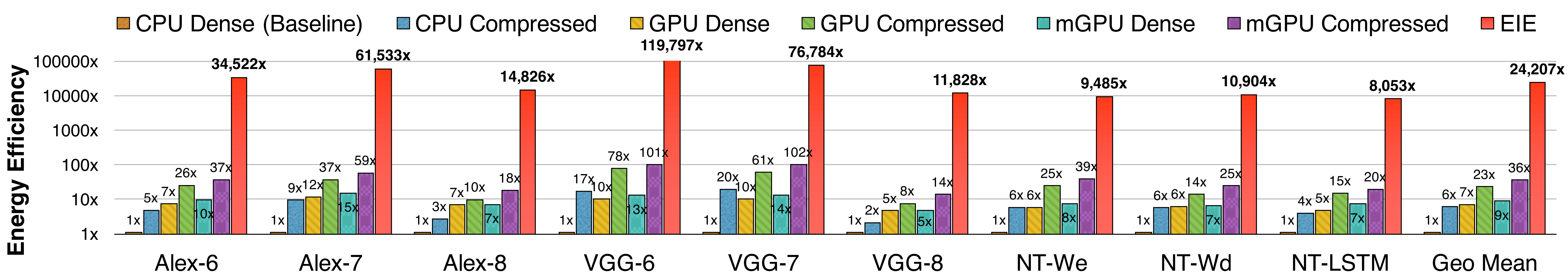


Figure 6. Speedups of GPU, mobile GPU and EIE compared with CPU running uncompressed DNN model. There is no batching in all cases.



CPU: Core i7 5930k (6 cores)
GPU: GTX Titan X
mGPU: Tegra K1

Warning: these are not end-to-end: just fully connected layers! And recall most of the compute is in the convo layers!

Sources of energy savings:

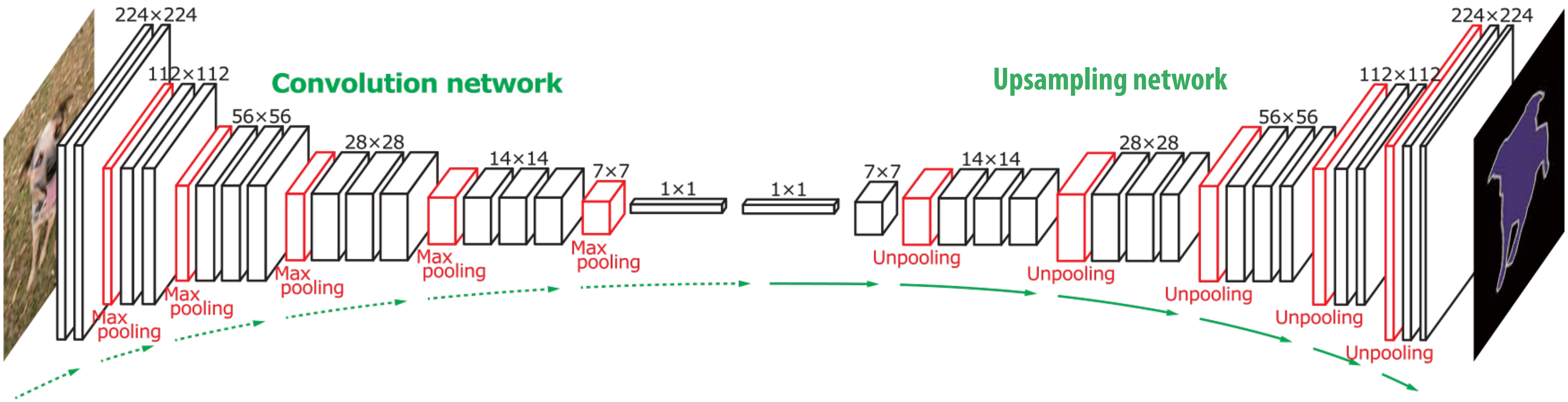
- Compression allows all weights to be stored in SRAM (few DRAM loads)
- Low-precision 16-bit fixed-point math (5x more efficient than 32-bit fixed math)
- Skip math on inputs activations that are zero (65% less math)

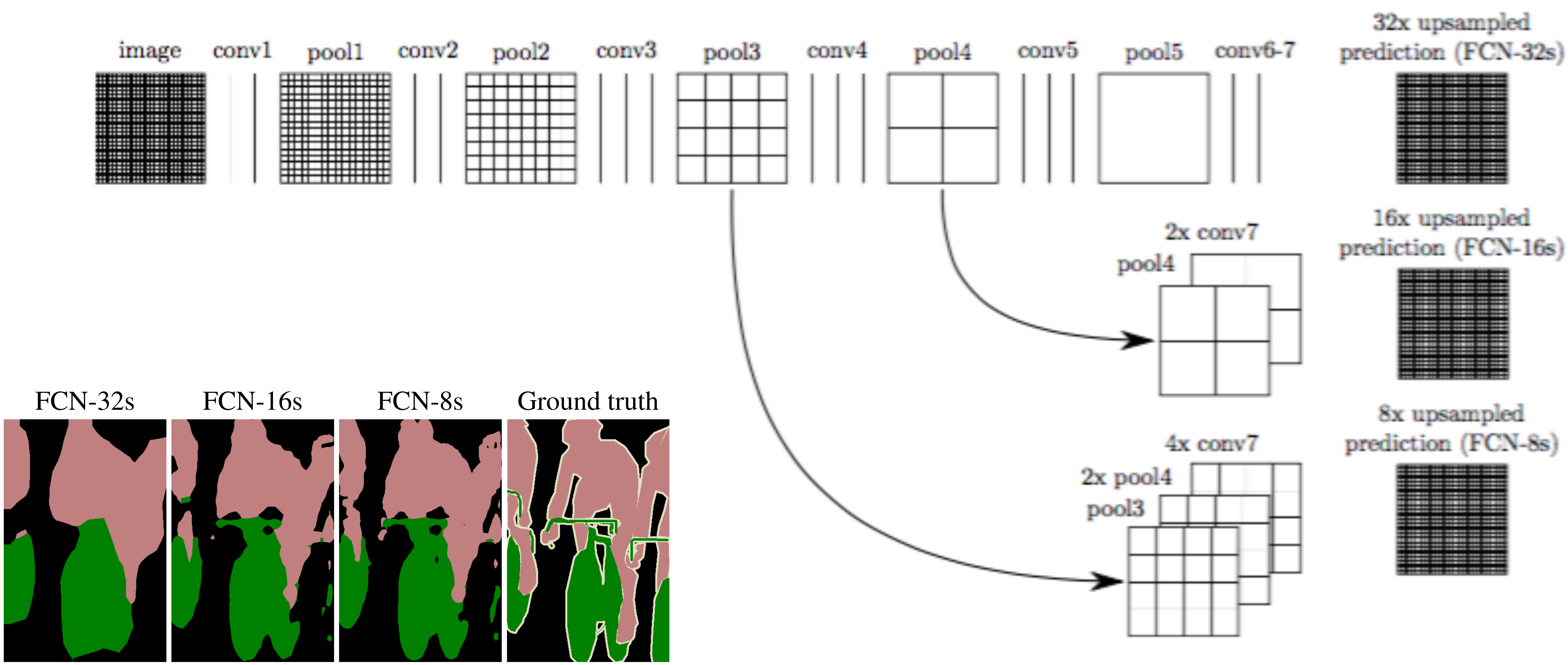
Thoughts

- EIE paper highlights performance on fully connected layers (see graph above)
  - Final layers of networks like AlexNet, VGG…
  - Common in recurrent network topologies like LSTMs

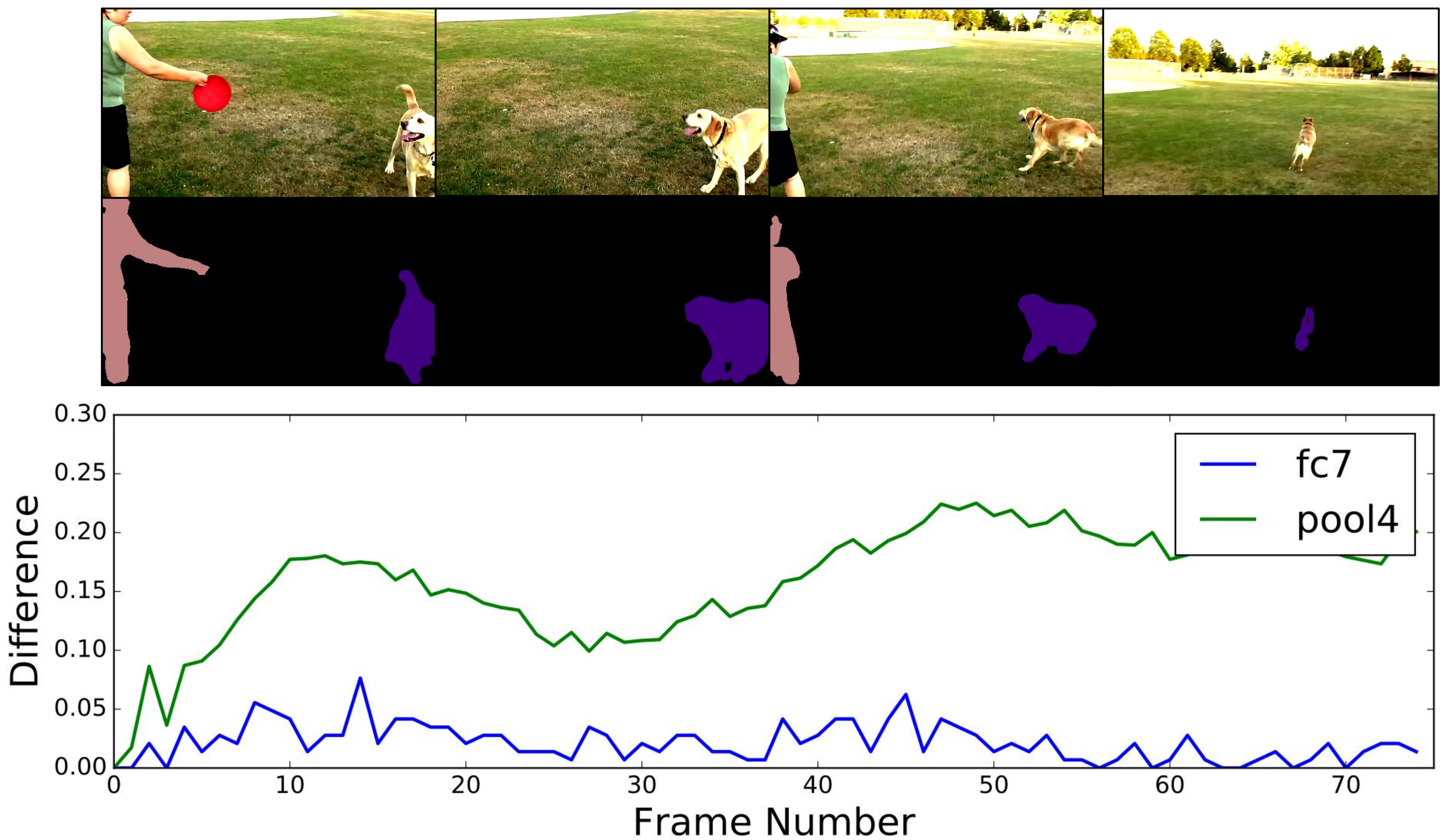
- But many state-of-the-art image processing networks have moved to fully convolutional solutions
  - Recall Inception, SqueezeNet, etc..

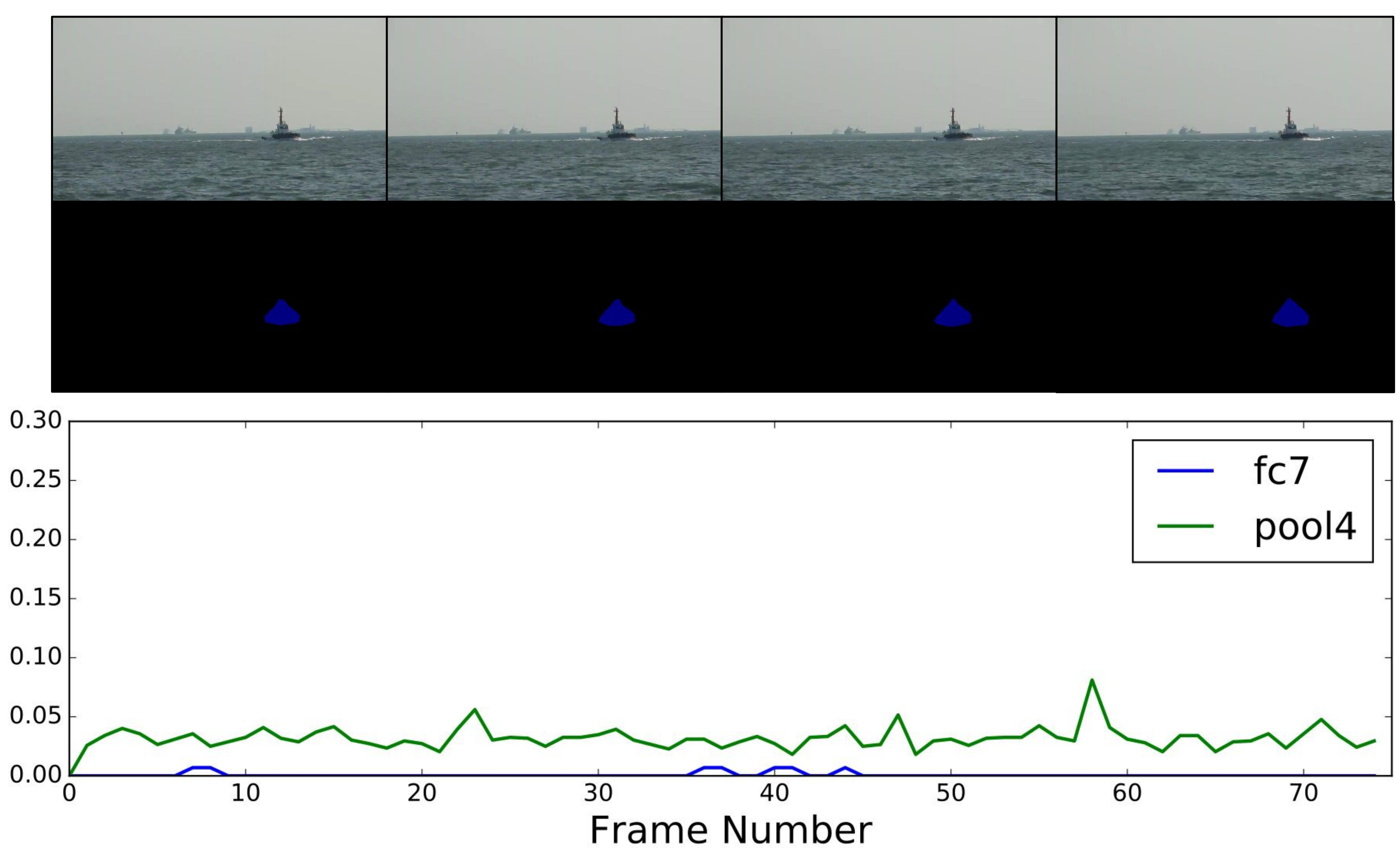
A fully convolutional network for image segmentation





Temporal stability of deep features

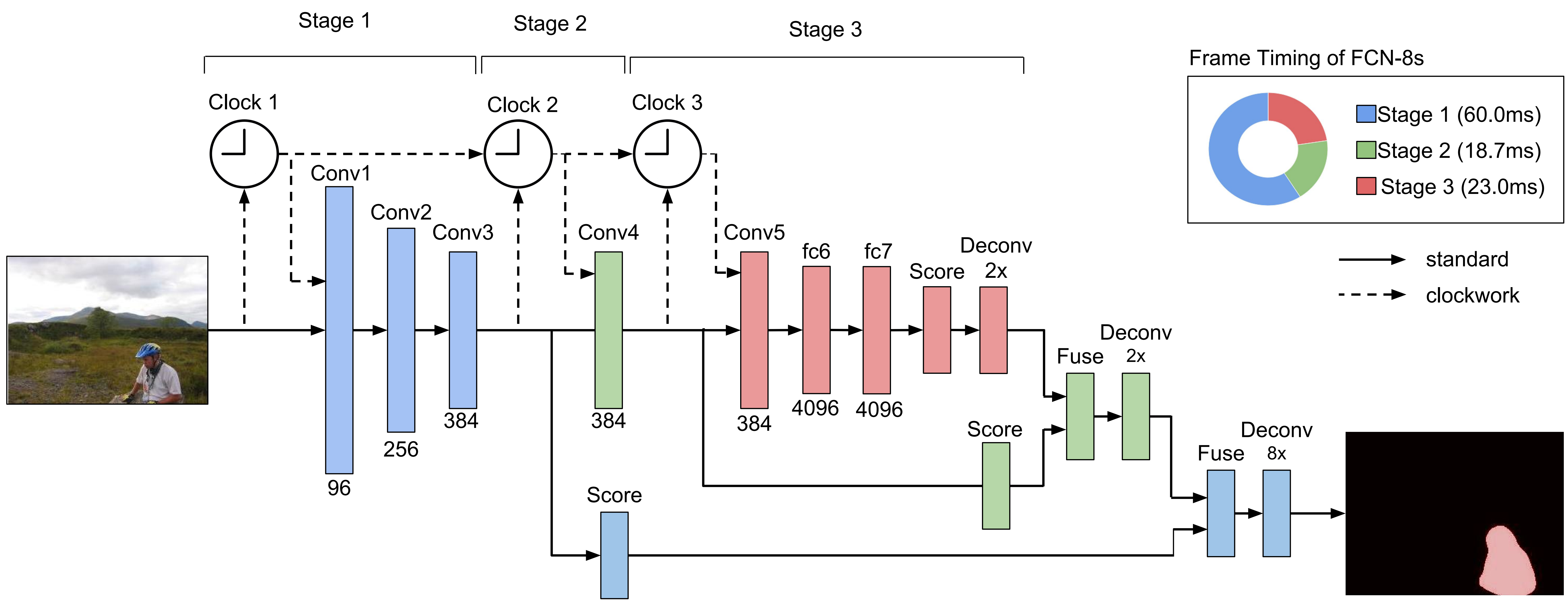




Observation:
Deeper features feature more temporal stability

(more semantic information changes less rapidly in a scene)

Clockwork network: reuse deeper layer outputs in subsequent frames



Evaluate lower (early) layers each frame

Optionally combine (fresh) output of lower layers with output of higher layers from previous frames.

Today: three types of optimizations

- Static, manual: human construction of new, more efficient topologies (e.g., Inception, SqueezeNet)

- Static, automatic analysis driven: (e.g., deep compression) analyze contents of network to determine how to prune topology or quantize weights

- Dynamic: analyze network activations during inference to determine when subsequent work can be elided (e.g., clockwork network)

  Note: EIE hardware also exploited dynamic sparsity in activations (e.g., due to ReLUs), but this was not an approximation technique like the ones above

- Custom specialized hardware to handle irregularity introduced by these optimizations