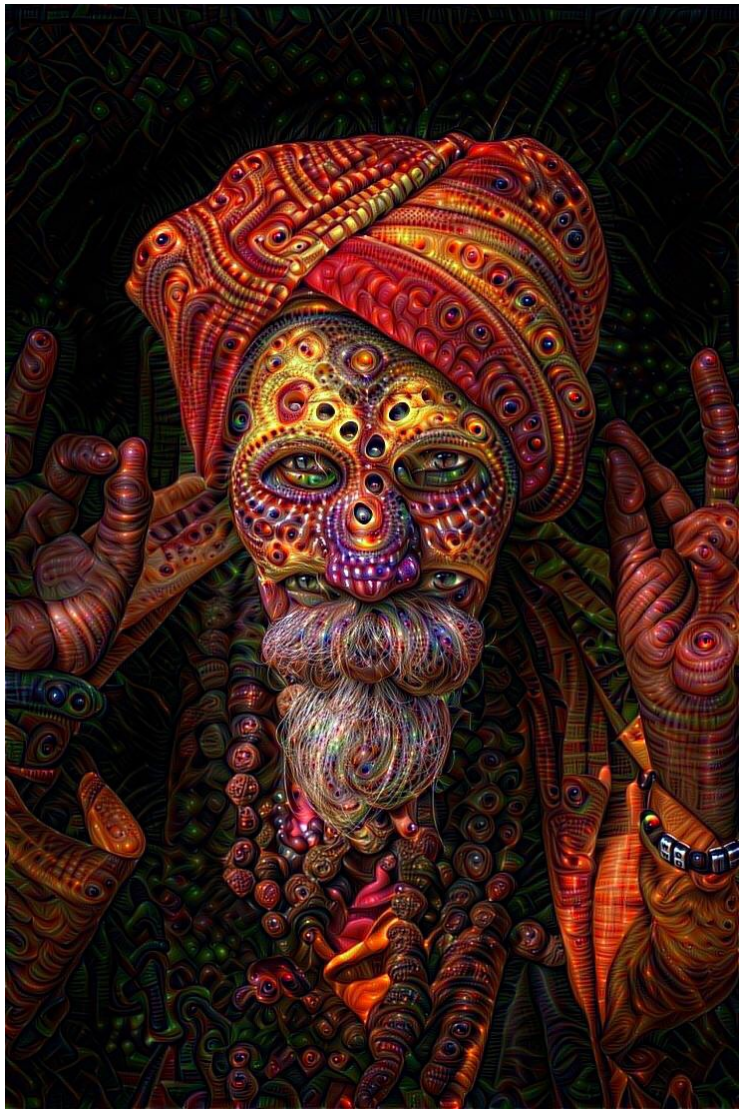
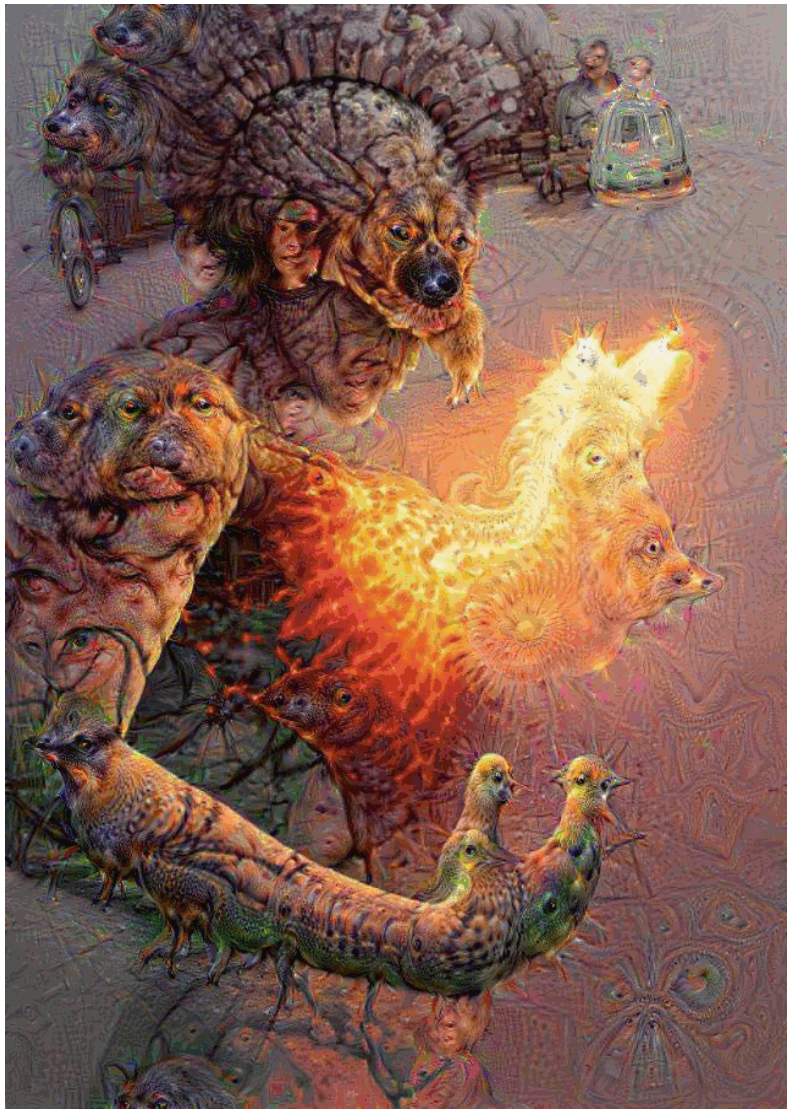
Deep Learning (for Computer Vision)

Arjun Jain | 8 April 2017

Sources

A lot of the material has been shamelessly and gratefully collected from:

- http://cs231n.stanford.edu/

DeepDream https://github.com/google/deepdream

```lua
require 'nn'
require 'cunn'
require 'cudnn'
require 'image'


local cuda = true
torch.setdefaulttensortype('torch.FloatTensor')
net = torch.load('./GoogLeNet_v2.t7'):cuda()

local Normalization = {mean = 118.380948/255, std = 61.896913/255}

function reduceNet(full_net,end_layer)
    local net = nn.Sequential()
    for l=1,end_layer do
        net:add(full_net:get(l))
    end
    print(net)
    return net
end

function make_step(net, img, clip,step_size, jitter)..

function deepdream(net, base_img, iter_n, octave_n, octave_scale, end_layer, clip, visualize)..

img = image.load('./sky1024px.jpg')
x = deepdream(net,img,20)
image.display(x)
```

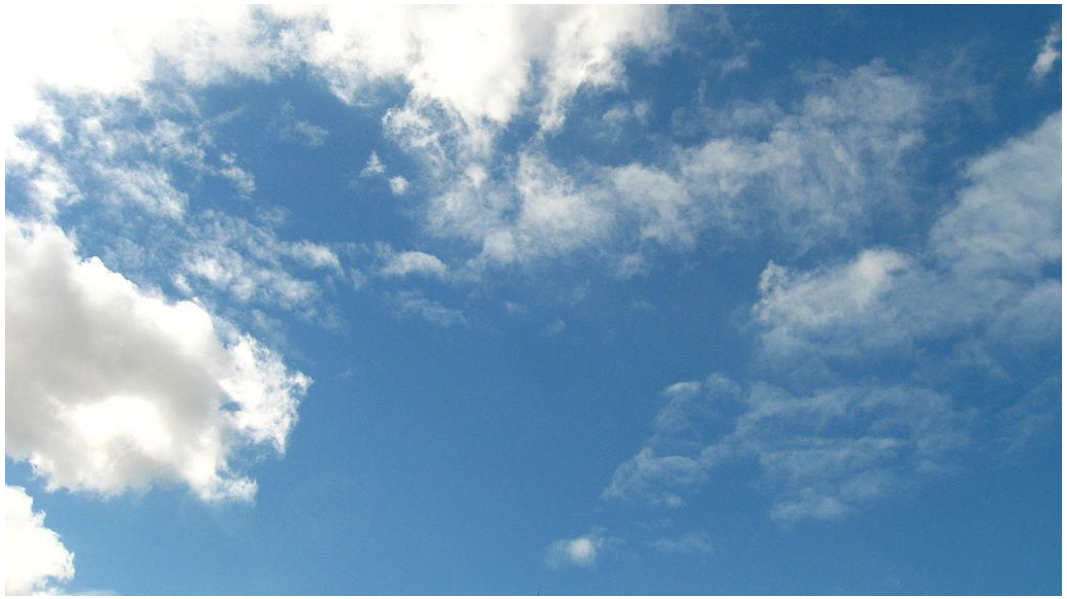
https://github.com/eladhoffer/DeepDream.torch

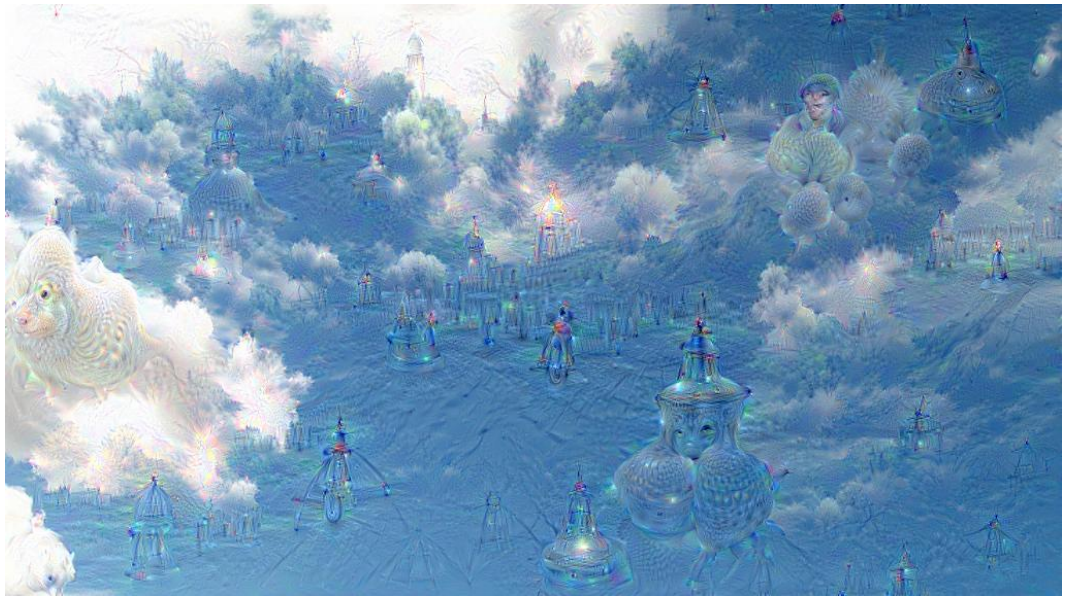
```lua
function make_step(net, img, clip,step_size, jitter)
    local step_size = step_size or 0.01
    local jitter = jitter or 32
    local clip = clip
    if clip == nil then clip = true end

    local ox = 0--2*jitter - math.random(jitter)
    local oy = 0--2*jitter - math.random(jitter)
    img = image.translate(img,ox,oy) -- apply jitter shift
    local dst, g
    dst = net:forward(img)
    g = net:updateGradInput(img,dst)
    -- apply normalized ascent step to the input image
    img:add(g:mul(step_size/torch.abs(g):mean()))

    img = image.translate(img,-ox,-oy) -- apply jitter shift
    if clip then
        bias = Normalization.mean/Normalization.std
        img:clamp(-bias,1/Normalization.std-bias)
    end
    return img
end
```

DeepDream: set dx = x :)

https://github.com/eladhoffer/DeepDream.torch

inception_4c/output

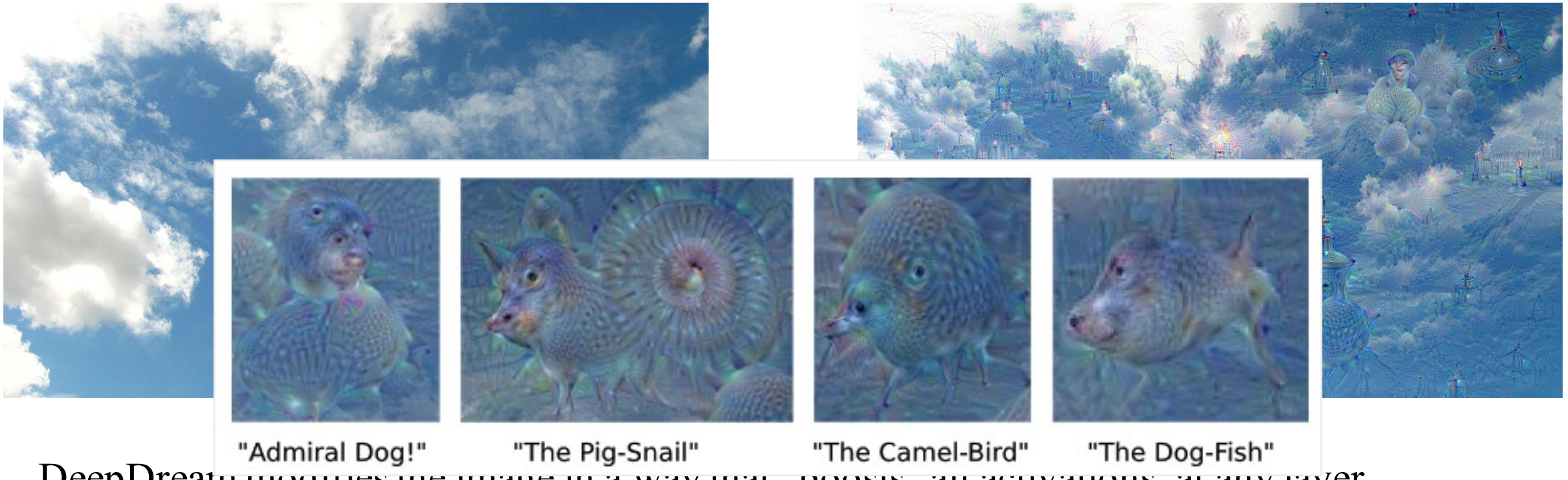




DeepDream modifies the image in a way that “boosts” all activations, at any layer

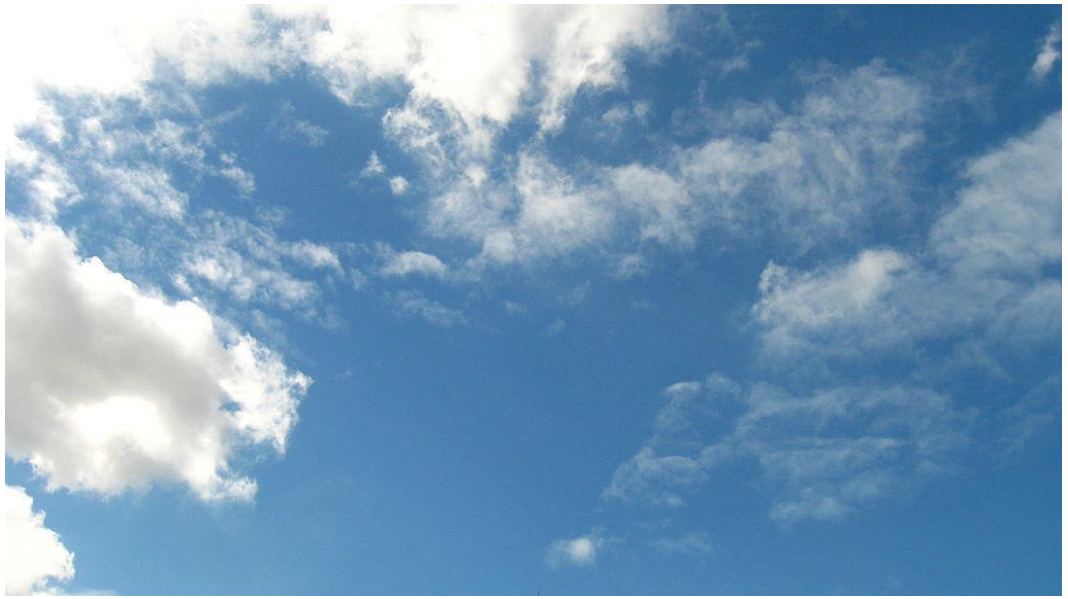
this creates a feedback loop: e.g. any slightly detected dog face will be made more and more dog like over time

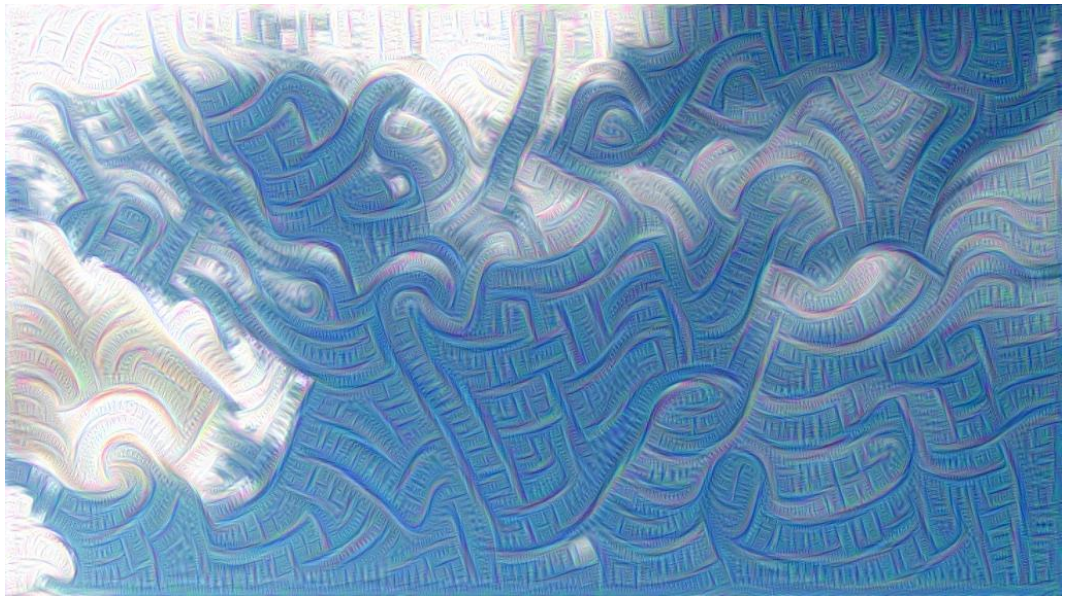
inception_4c/output



DeepDream modifies the image in a way that "boosts" all activations, at any layer

inception_3b/5x5_reduce





DeepDream modifies the image in a way that “boosts” all activations, at any layer

Bonus videos

Deep Dream (Grocery **Trip**)

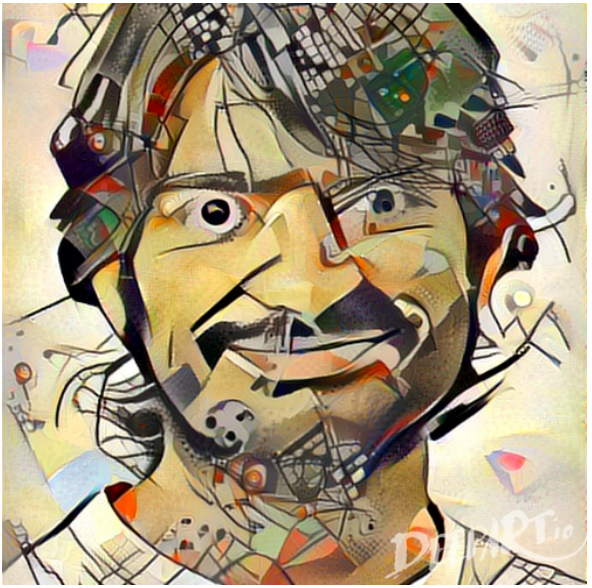
https://www.youtube.com/watch?v=DgPaCWJL7XI

NeuralStyle

[A Neural Algorithm of Artistic Style by Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge, 2015]

good implementation in Torch:

https://github.com/jcjohnson/neural-style








make your own easily on deepart.io

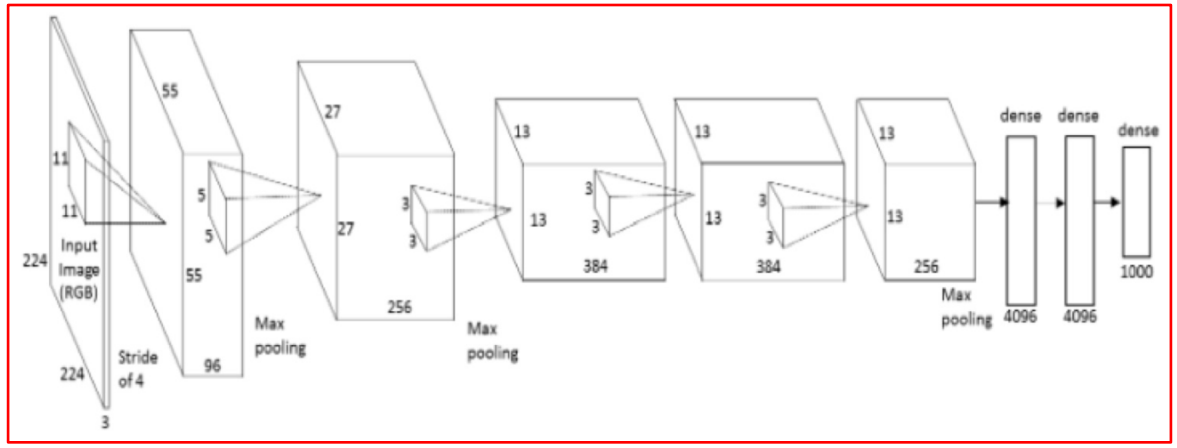
https://github.com/junyanz/CycleGAN

Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks

Jun-Yan Zhu*, Taesung Park*, Phillip Isola, Alexei A. Efros
Berkeley AI Research Lab, UC Berkeley

In arxiv, 2017. (* equal contributions)

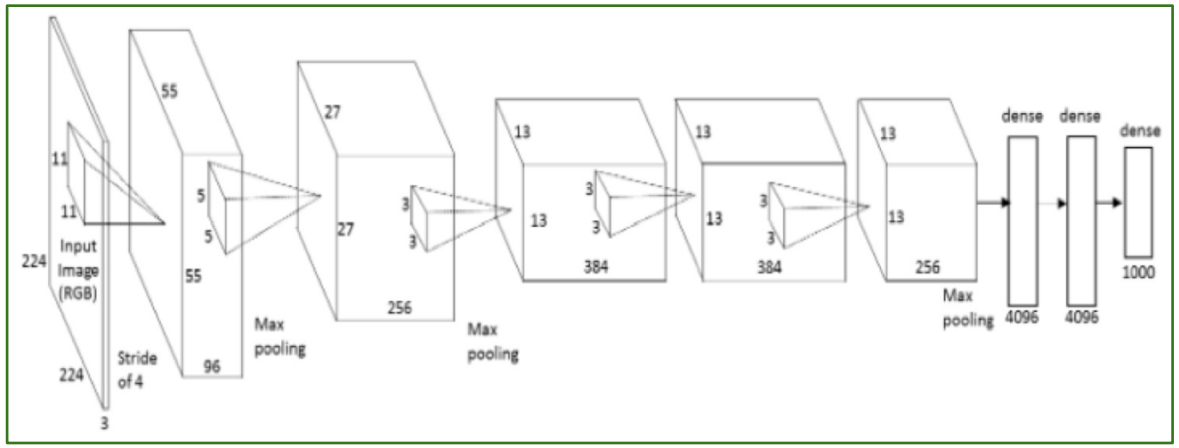
Step 1: Extract **content targets** (ConvNet activations of all layers for the given content image)



content activations

e.g.
at CONV5 layer we would have a [14x14x512] array of target activations

Step 2: Extract **style targets** (Gram matrices of ConvNet activations of all layers for the given style image)



style gram matrices

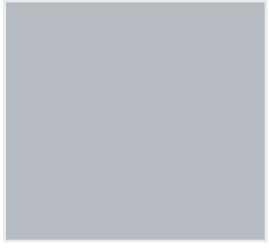
$$G = V^{\mathrm{T}} V$$

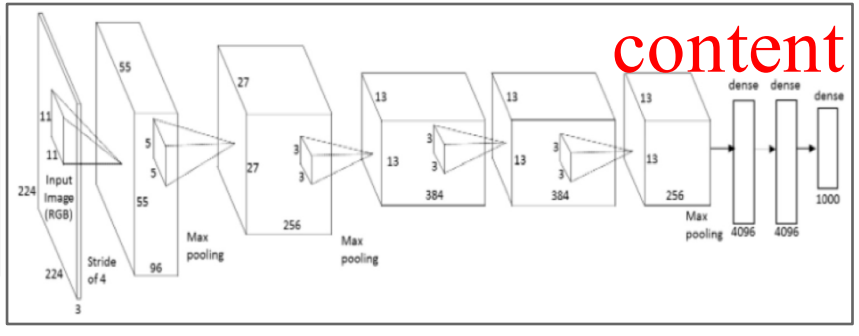
e.g.
at CONV1 layer (with [224x224x64] activations) would give a [64x64] Gram matrix of all pairwise activation covariances (summed across spatial locations)

Step 3: Optimize over image to have:

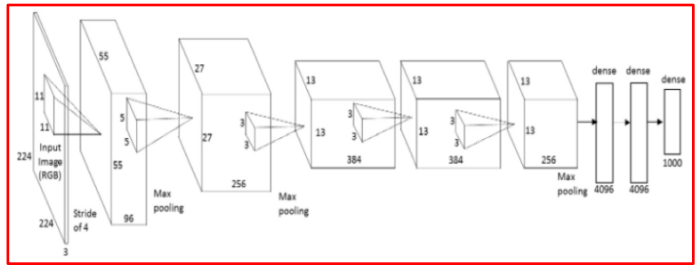
- The **content** of the content image (activations match content)
- The **style** of the style image (Gram matrices of activations match style)

$$\mathcal{L}_{total}(\vec{p}, \vec{a}, \vec{x}) = \alpha \mathcal{L}_{content}(\vec{p}, \vec{x}) + \beta \mathcal{L}_{style}(\vec{a}, \vec{x})$$

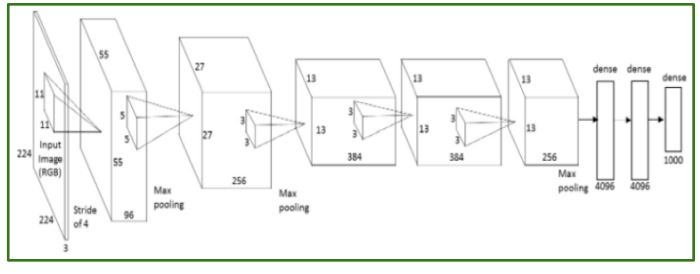




match content



match style

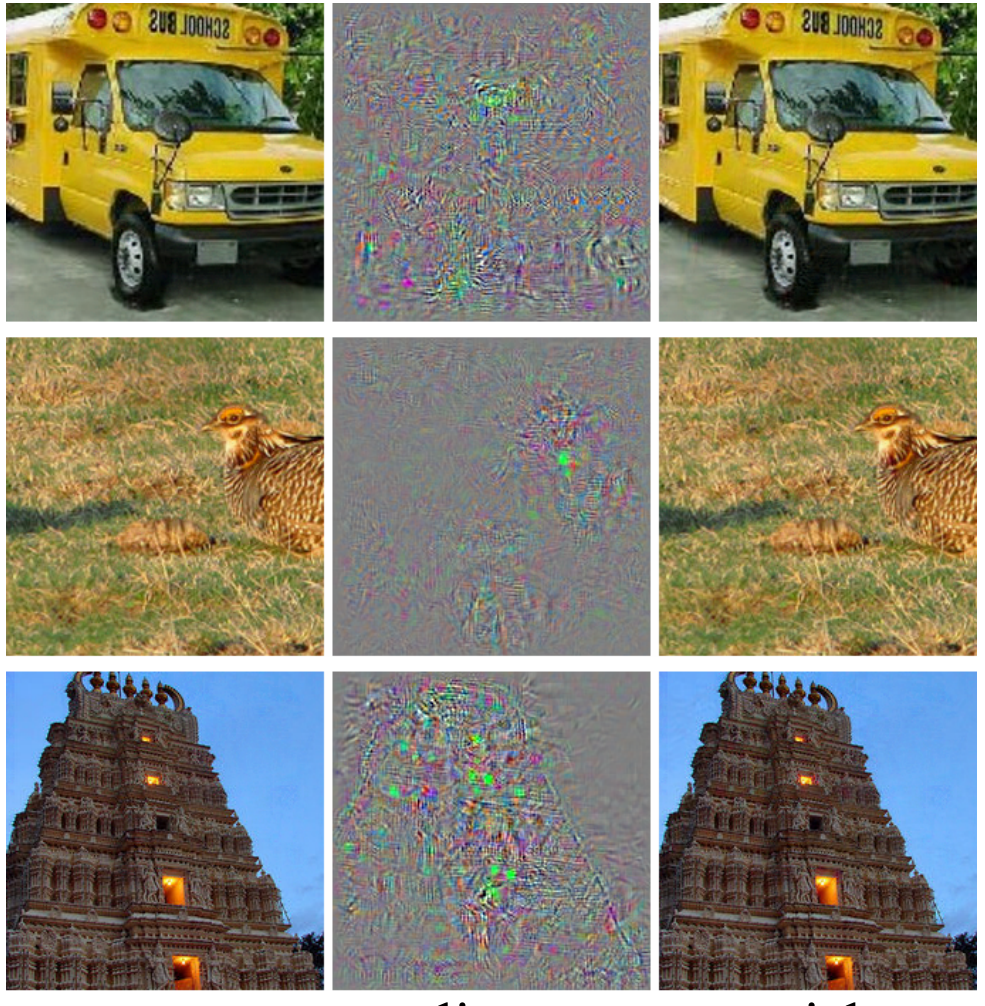


We can pose an optimization over the input image to maximize any class score.
That seems useful.

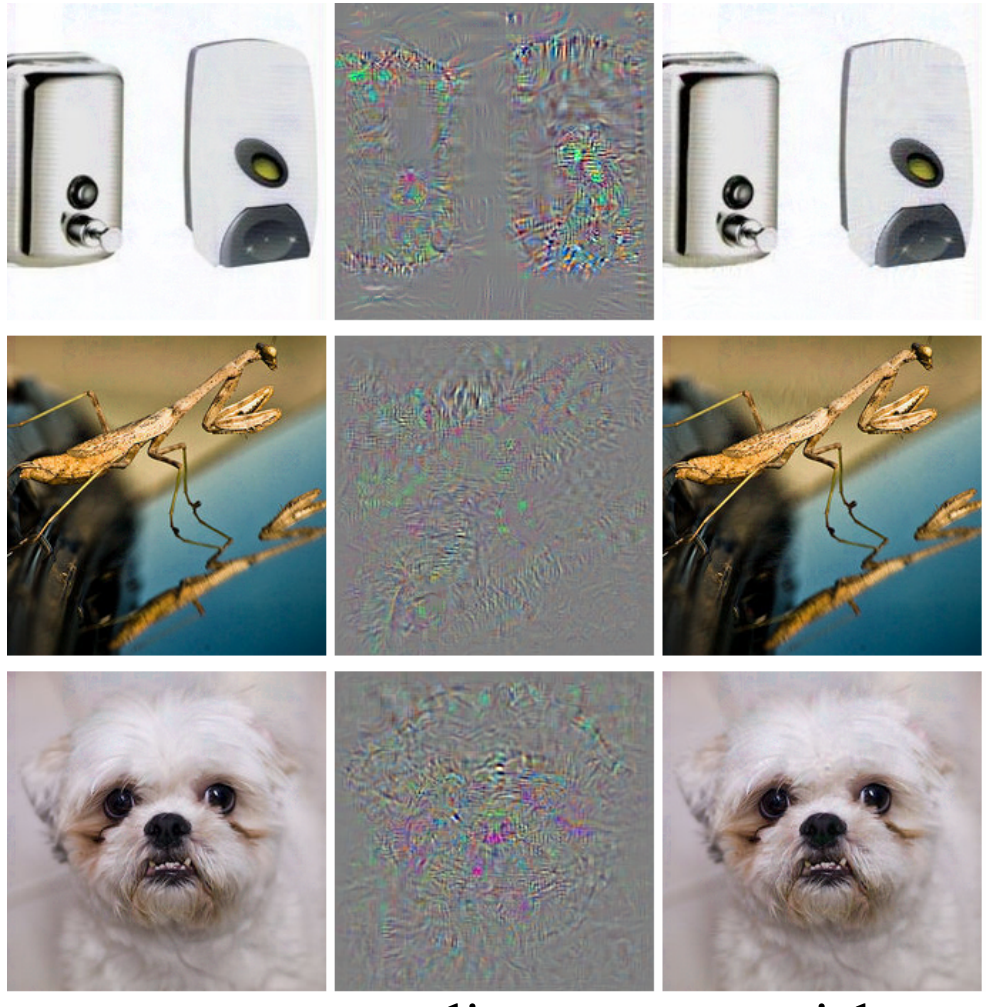
Question: Can we use this to “fool” ConvNets?

spoiler alert: yeah

[Intriguing properties of neural networks, Szegedy et al., 2013]



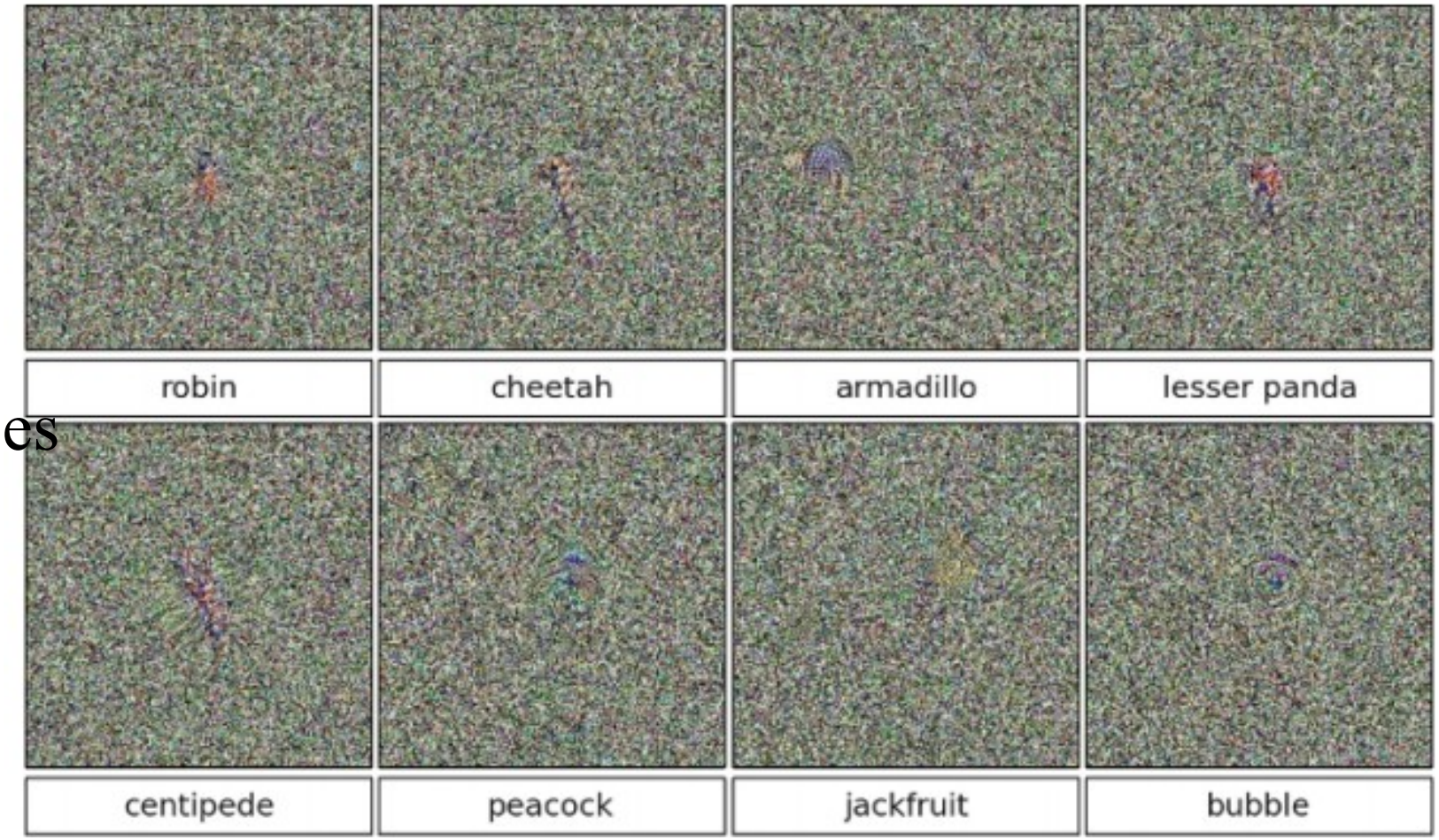
correct +distort ostrich



correct +distort ostrich

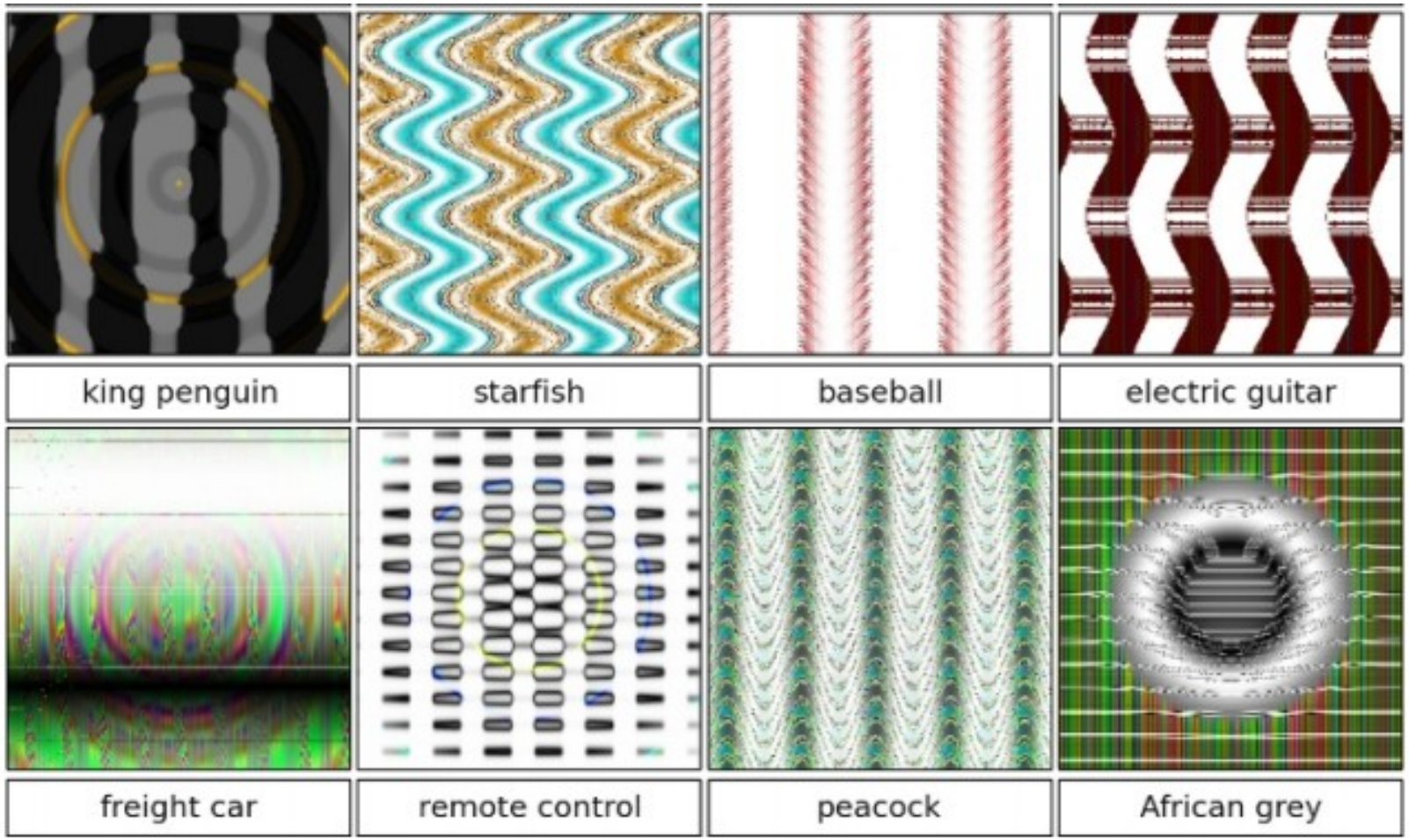
[Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images Nguyen, Yosinski, Clune, 2014]

>99.6% confidences



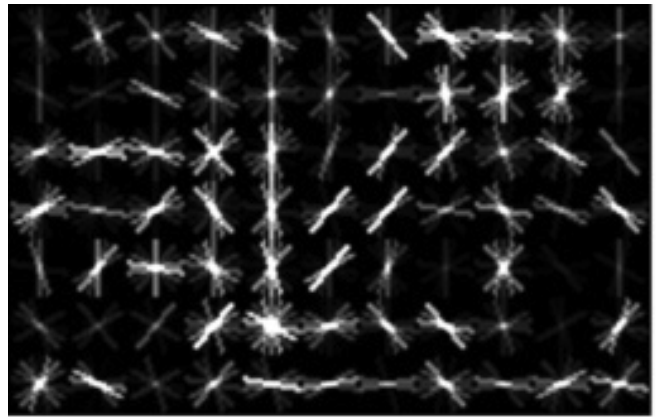
[Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images Nguyen, Yosinski, Clune, 2014]

>99.6% confidences



These kinds of results were around even before ConvNets…

[Exploring the Representation Capabilities of the HOG Descriptor, Tatu et al., 2011]



Identical HOG represention

Backpropping to the image is powerful

It can be used for:

- **Understanding** (e.g. visualize optimal stimuli for arbitrary neurons)
- **Segmenting** objects in the image (kind of)
- **Inverting** codes and introducing privacy concerns
- **Fun** (NeuralStyle/DeepDream)
- **Confusion and chaos** (Adversarial examples)

Deep Learning is Expensive

- Deep neural networks (DNNs) are expensive: Memory, Compute and Power
  - E.g.: AlexNet has 61M parameters (249MB of memory) and performs 1.5B single precision operations to classify one image
- Prohibitive for smaller devices like cell phones and embedded electronics
- Currently the only option is to run them on the cloud
  - Network delay
  - Power budget
  - User privacy
- Overhead when pushing new models to clients

Model Compression

- Smaller size – compress AlexNet by **461 x**
- With no loss or EVEN improved accuracy
- Speedup – faster inference with lesser number
- Lower power usage - energy consumption is dominated by memory access

Operation	Energy [pJ]
32 bit int ADD	.1
32 bit float ADD	.9
32 bit int MULT	3.1
32 bit float MULT	3.7
32 bit SRAM Cache READ	5
32 bit DRAM Memory READ	640

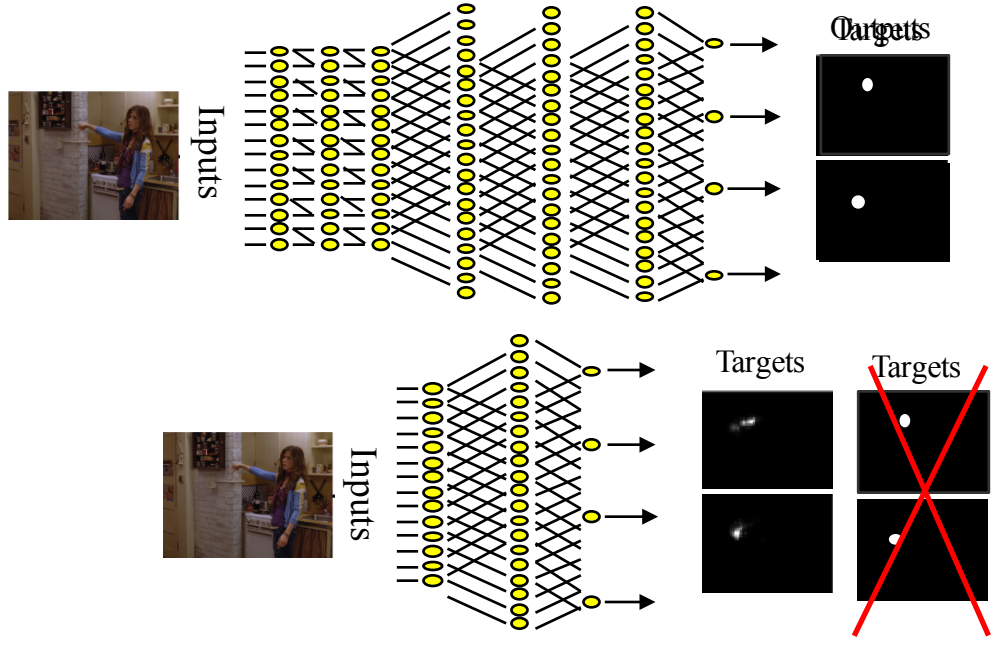
45nm CMOS technology for an FPGA

Hardware Acceleration

- Most embedded hardware platforms such as FPGAs do not contain FPU
- Thus, convert to Fixed Point (integer) Arithmetic
- For high performance, must have optimal usage of bandwidth and computing resources.
- However, most often limited by either memory or compute speed

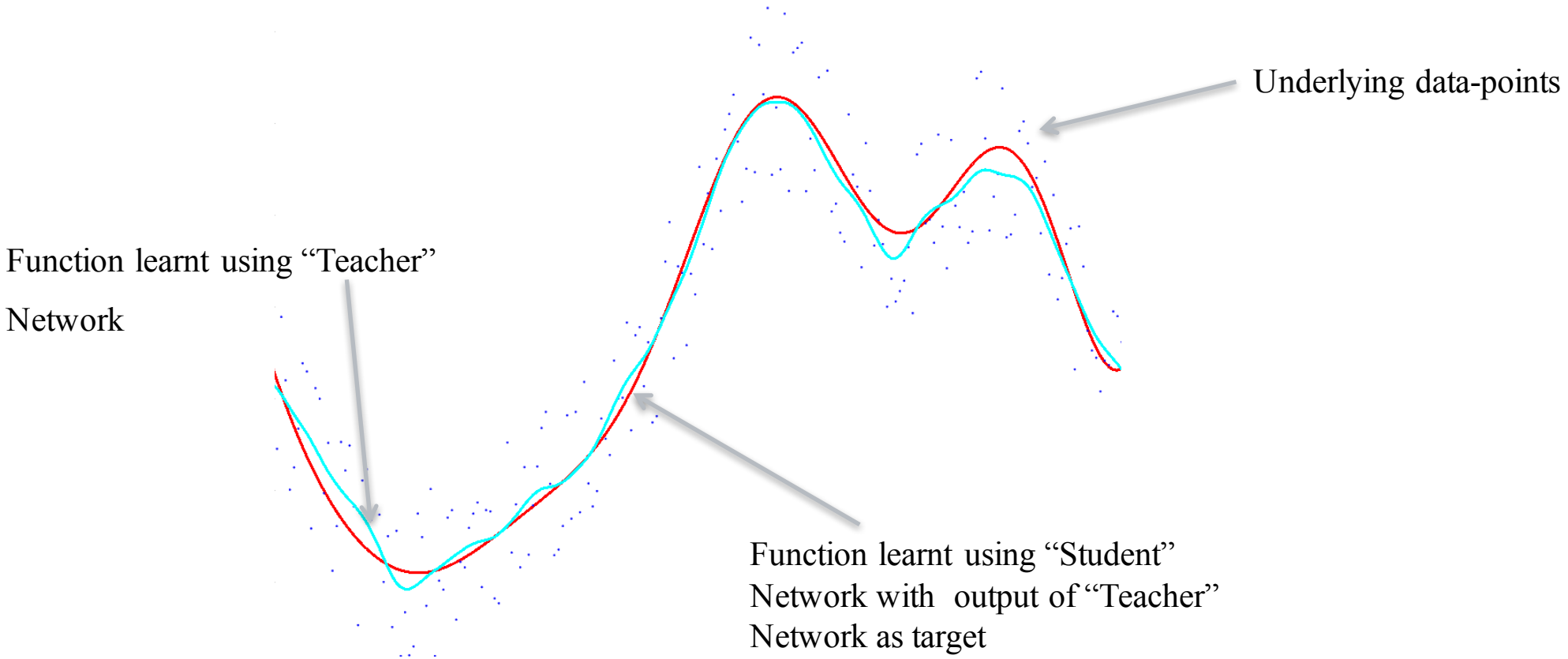
Student-Teacher Networks

- *Do Deep Nets Really Need to be Deep?* Jimmy Ba and Rich Caruana, **ICLR 2013 Workshop, NIPS 2014**
- *Distilling the Knowledge in a Neural Network.* Hinton et al., **NIPS DL Workshop 2014**
- *FitNets: Hints for thin Deep Nets.* Romero et al., **ICLR 2015**
- Train a shallower (*student*) network with outputs of the larger network (*teacher*) as target



Student-Teacher Networks

- Train a shallower "Student" network with outputs of the larger "Teacher" network as target



Student-Teacher Networks

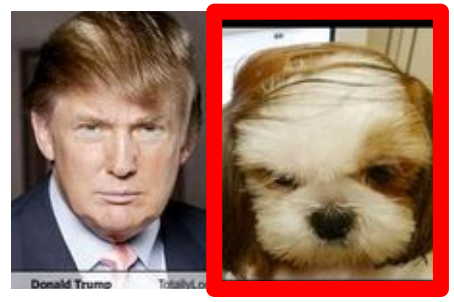
- If network uses MSE Loss (regression), simply use output of teacher network as target for student network
- If the network uses cross-entropy (classification) loss with SoftMax as the last layer output:
  - Use MSE as cost for the *student* network with the output of the *teacher* network before the soft max as target (*Do Deep Nets Really Need to be Deep?* Jimmy Ba and Rich Caruana, **NIPS 2014**)
  - Use soft max with a high *temperature* as target $q_i = \frac{exp(z_i/T)}{\sum_j exp(z_j/T)}$ (*Distilling the Knowledge in a Neural Network*. Hinton et al., **NIPS DL Workshop 2014**) and as cost minimize cross-entropy between:

$$\mathrm{P}_{\mathrm{T}}^{\tau} = \mathrm{softmax}\left(\frac{\mathbf{a}_T}{\tau}\right), \quad \mathrm{P}_{\mathrm{S}}^{\tau} = \mathrm{softmax}\left(\frac{\mathbf{a}_S}{\tau}\right)$$

$$\mathcal{L}_{KD}(\mathbf{W_S}) = \mathcal{H}(\mathbf{y_{true}}, \mathrm{P_S}) + \lambda \mathcal{H}(\mathrm{P}_{\mathrm{T}}^{\tau}, \mathrm{P}_{\mathrm{S}}^{\tau})$$

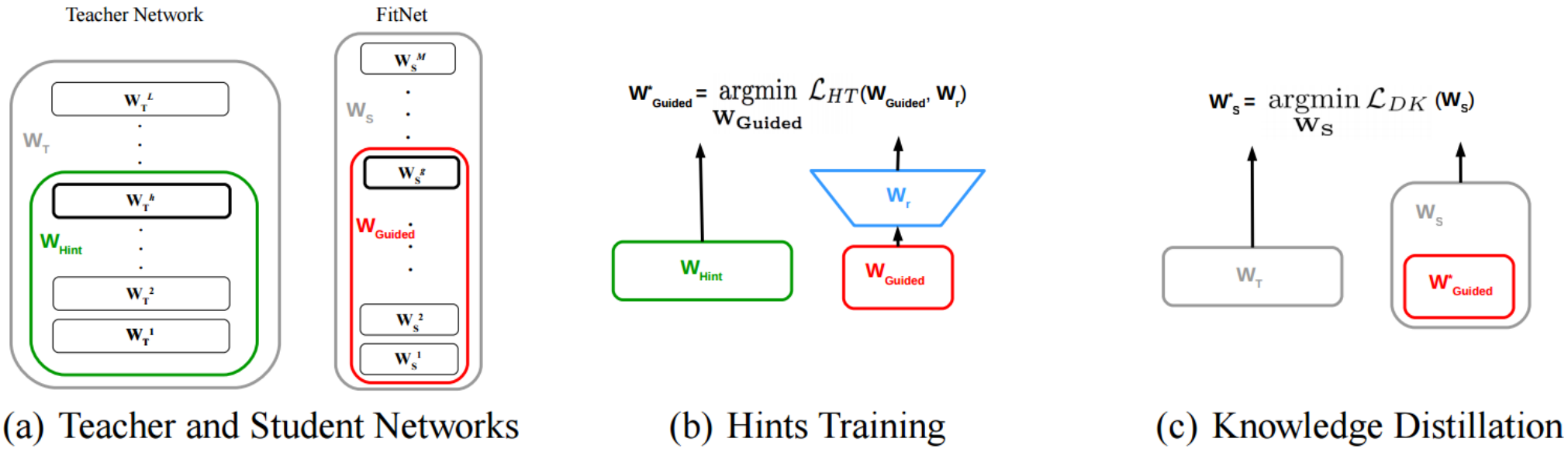
  - Softened outputs reveals the ***dark knowledge*** of the network

Cow	Dog	Trump	Bush	
0	1	0	0	Original Hard Targets
0.05	0.9	0.8	0.005	Softened Output



Student-Teacher Networks

- In FitNets (*FitNets: Hints for thin Deep Nets*. Romero et al., **ICLR 2015)**, they also add intermediate supervision



(a) Teacher and Student Networks (b) Hints Training (c) Knowledge Distillation

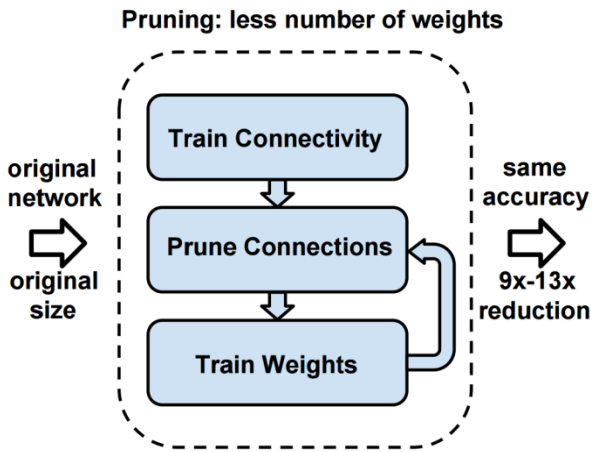
Deep Compression: Network Pruning

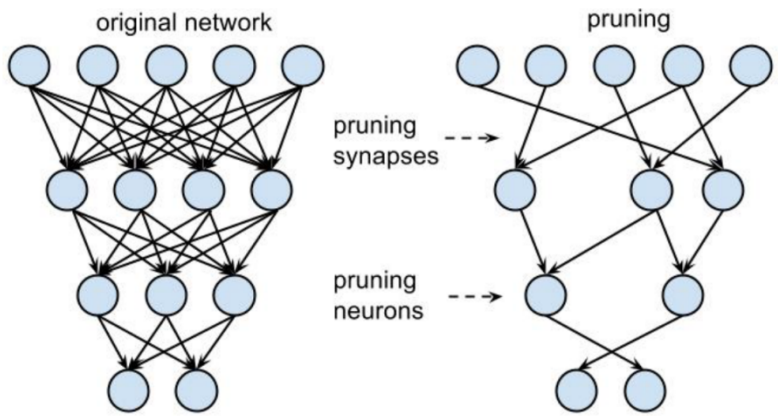
- Learn only the important connections, remove redundant ones
- Idea around forever - *Optimal Brain Damage*, LeCun et al., **NIPS1990**
- Motivation: The synapses pruning mechanism of the human brain

Age	Number of Connections	Stage
At birth	50 Trillion	Newly formed
1 year old	1000 Trillion	Peak
10 year old	500 Trillion	Pruned and stabilized

Deep Compression: Network Pruning

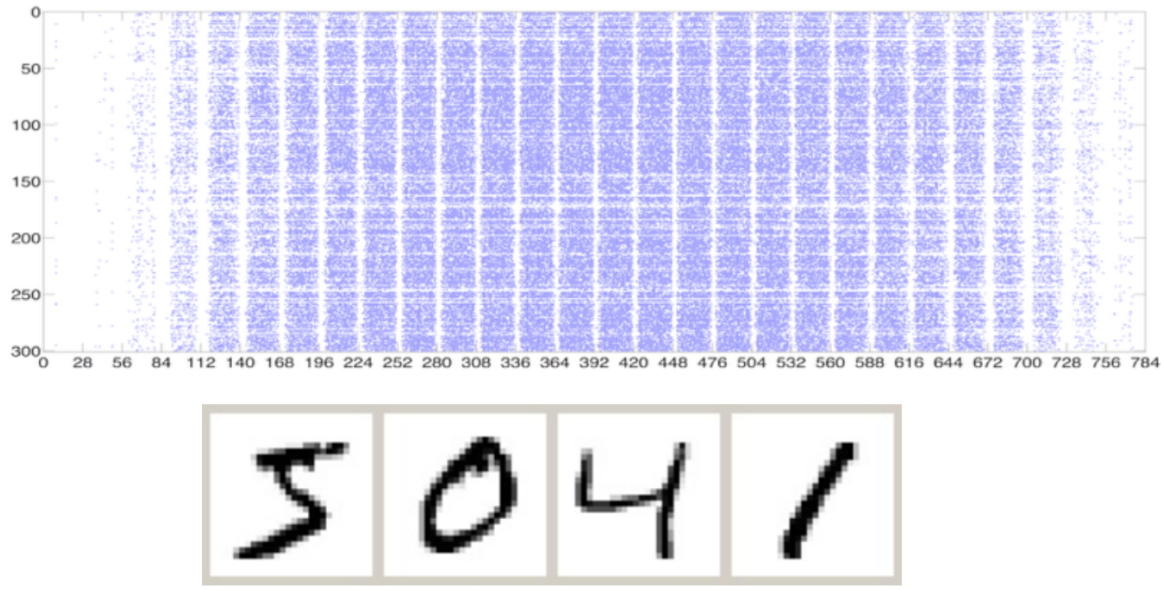
- Start by learning the connectivity via normal network training
- Prune the small-weight connections: all connections with weights below a threshold are removed
- Retrain the network to learn the final weights for the remaining sparse connections
- Reduction: 9X for AlexNet and 13x VGG-16





Deep Compression: Network Pruning

- Sparsity is natural in deep learning
- It has a banded structure repeated 28 times, which correspond to the un-pruned parameters in the center of the images, since the digits are written in the center.



Deep Compression: Network Pruning

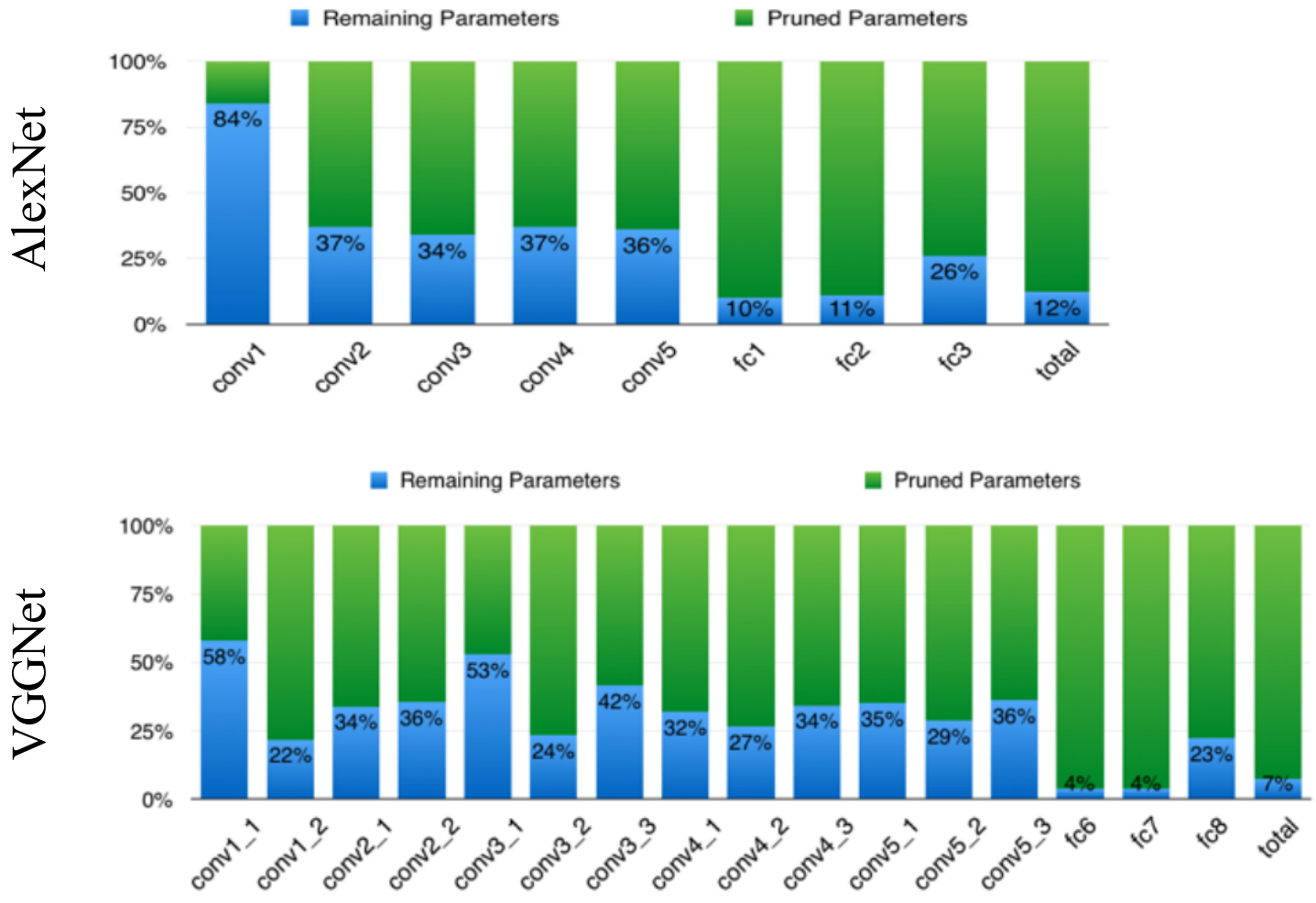
- Parameter compression:

Network	Top-1 Error	Top-5 Error	Parameters	Compression Rate
AlexNet Ref	42.78%	19.73%	61M	
AlexNet Pruned	42.77%	19.67%	**6.7M**	**9**×
VGG16 Ref	31.50%	11.32%	138M	
VGG16 Pruned	31.34%	10.88%	**10.3M**	**13**×

Deep Compression: Network Pruning

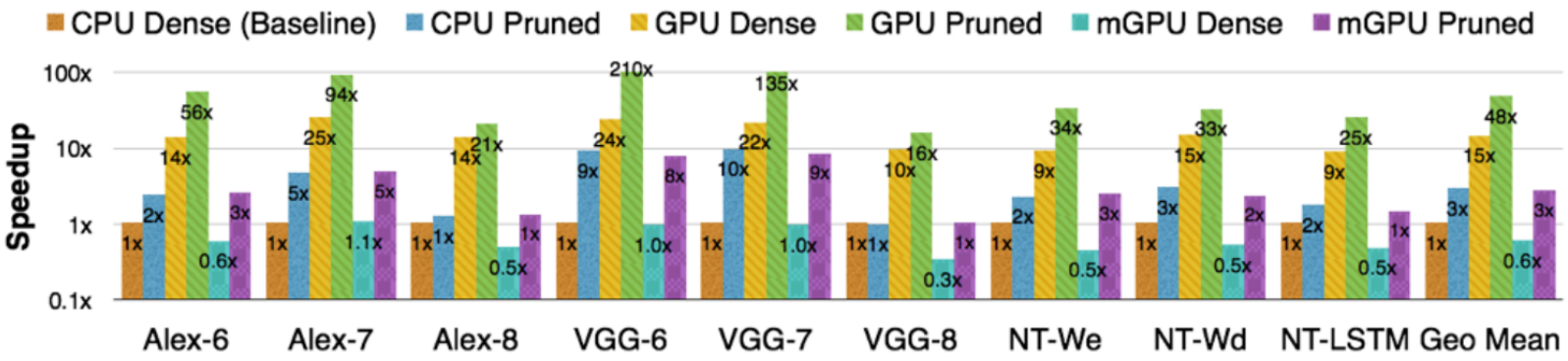
- Parameter compression

  (Layer wise)



Deep Compression: Network Pruning

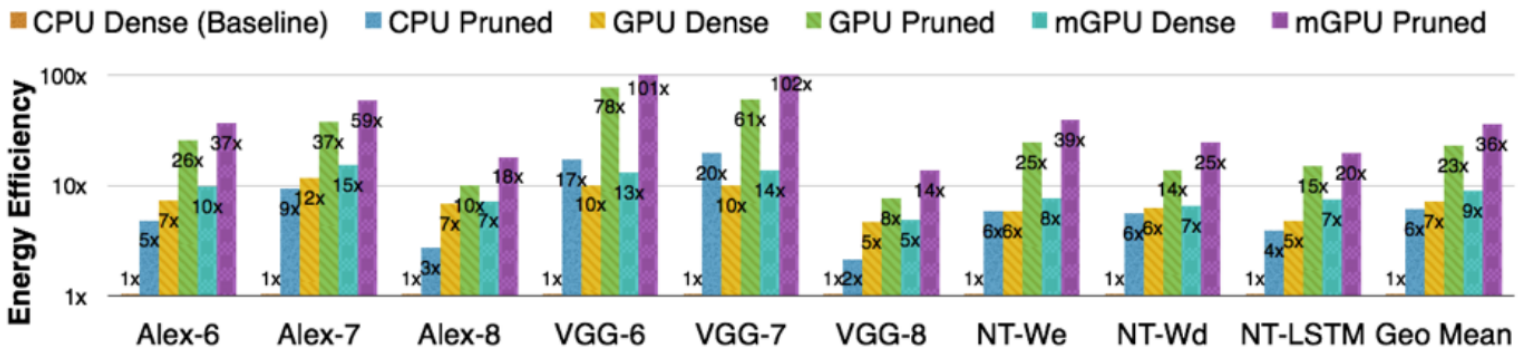
- Speedup (FC Layer)



- Intel Core i7 5930K: MKL CBLAS GEMV, MKL SPBLAS CSRMV
- NVIDIA GeForce GTX Titan X: cuBLAS GEMV, cuSPARSE CSRMV
- NVIDIA Tegra K1: cuBLAS GEMV, cuSPARSE CSRMV

Deep Compression: Network Pruning

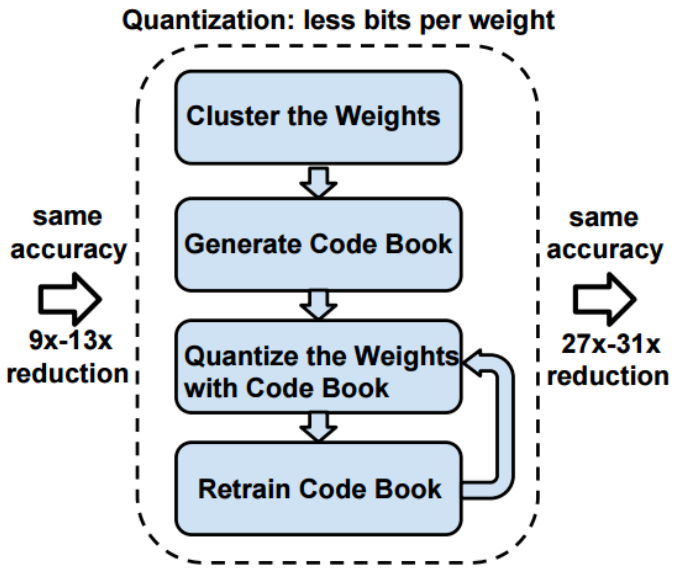
- Energy Efficiency (FC Layer)

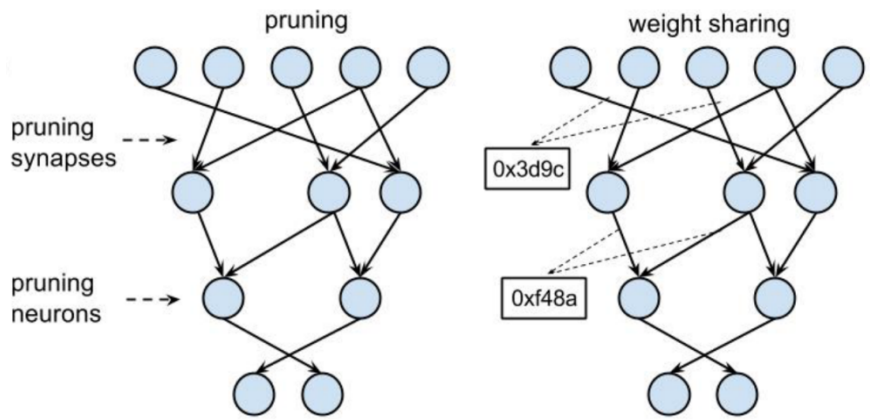


- Intel Core i7 5930K: CPU socket and DRAM power are reported by pcm-power utility
- NVIDIA GeForce GTX Titan X: reported by nvidia-smi utility
- NVIDIA Tegra K1: measured the total power consumption with a power-meter, 15% AC to DC conversion loss, 85% regulator efficiency and 15% power consumed by peripheral components => 60% AP+DRAM power

Deep Compression: Trained Quantization

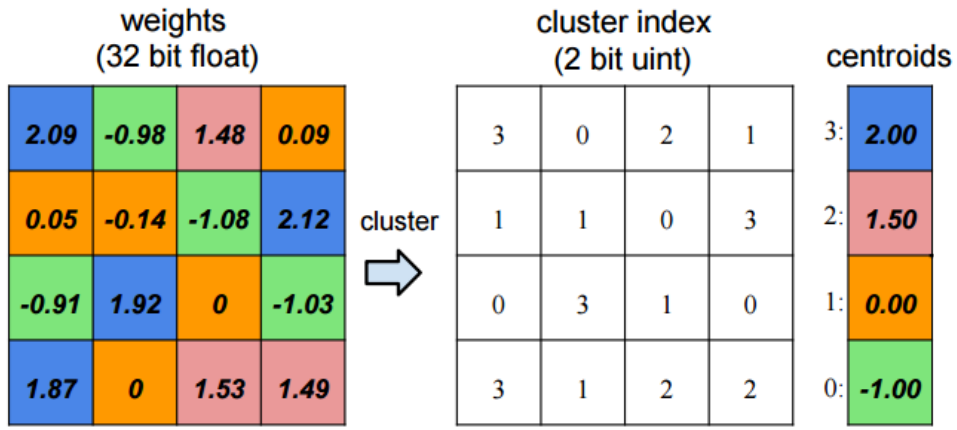
- Further compresses by reducing the number of bits required to represent each weight
- Limit the number of effective weights we need to store by having multiple connections share the same weight
- Fine-tune those shared weights





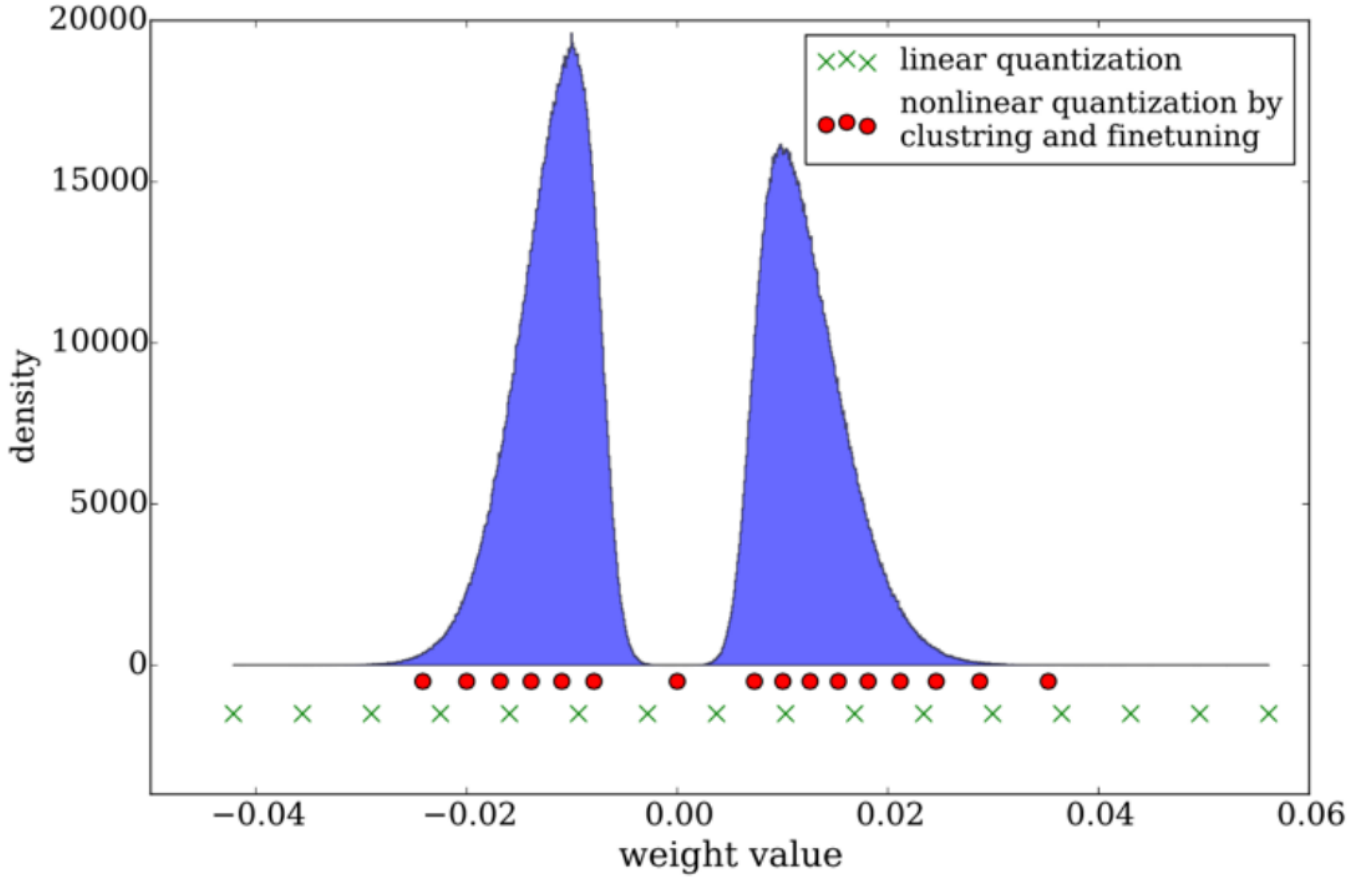
Deep Compression: Trained Quantization

- Further compresses by reducing the number of bits required to represent each weight



Deep Compression: Trained Quantization

- Distribution of weights (blue) and distribution of codebook before (green cross) and after fine-tuning (red dot)

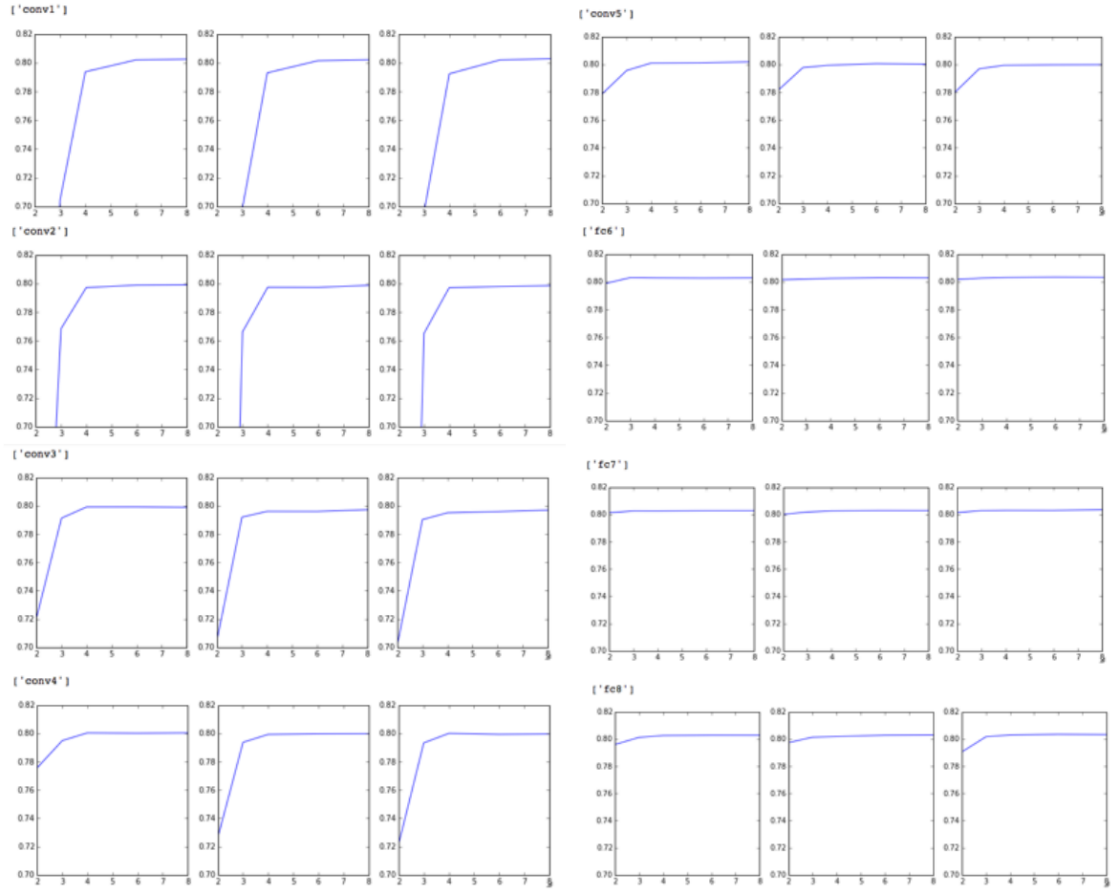


Deep Compression: Trained Quantization

- 128 (input channels) * 128 (output channels) * 5 * 5 = **409600 params => 409600 * 16 bits = 6553600 bits**
- **After training quantization => 409600 * 8 bits + 256 * 16 bits (2^8=256) = 3280896 bits = 2x reduction**
- 8/5 bit quantization results in no accuracy loss
- 8/4 bit quantization results in no top-5 accuracy loss, 0.1% top-1 accuracy loss
- 4/2 bit quantization results in -1.99% top-1 accuracy loss, and -2.60% top-5 accuracy loss, not that bad!

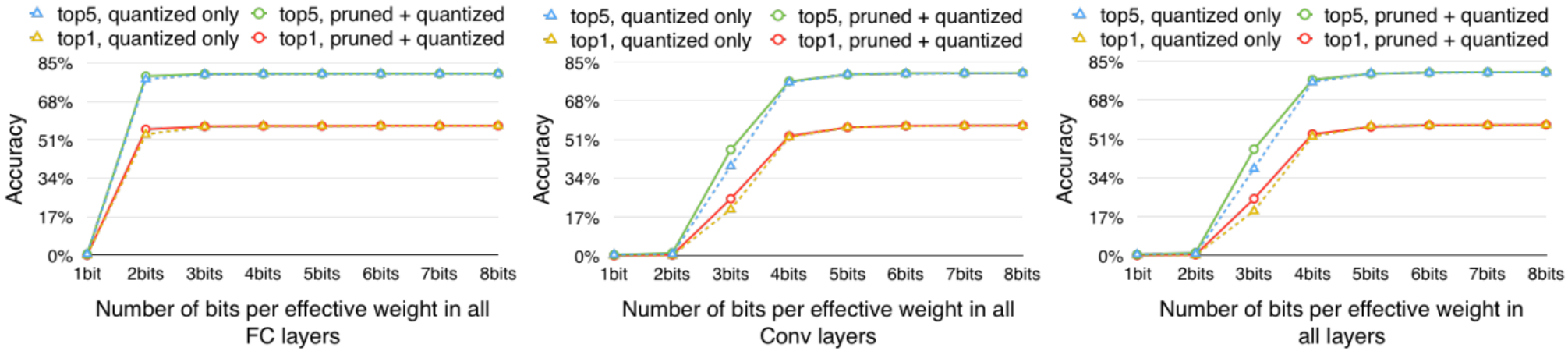
Deep Compression: Trained Quantization

- Accuracy - #Bits on 5 Conv Layers and 3 FC layers on AlexNet



Deep Compression: Trained Quantization

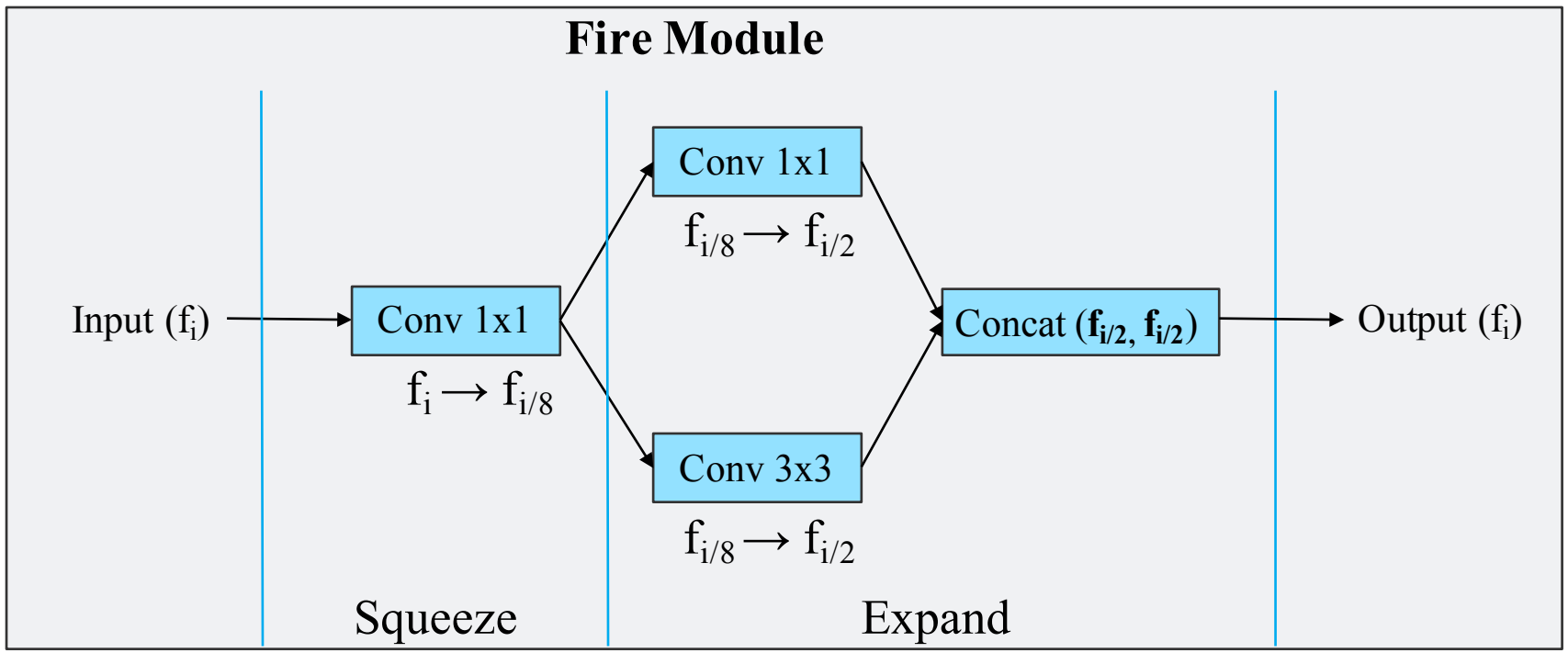
- Pruning and Quantization work well together



- Dashed: quantization on unpruned network, Solid: quantization on pruned network
- Accuracy begins to drop at the same number of quantization bits whether or not the network has been pruned
- Although pruning made the number of parameters less, quantization still works well, or even better (3 bits case on the middle figure) as in the unpruned network

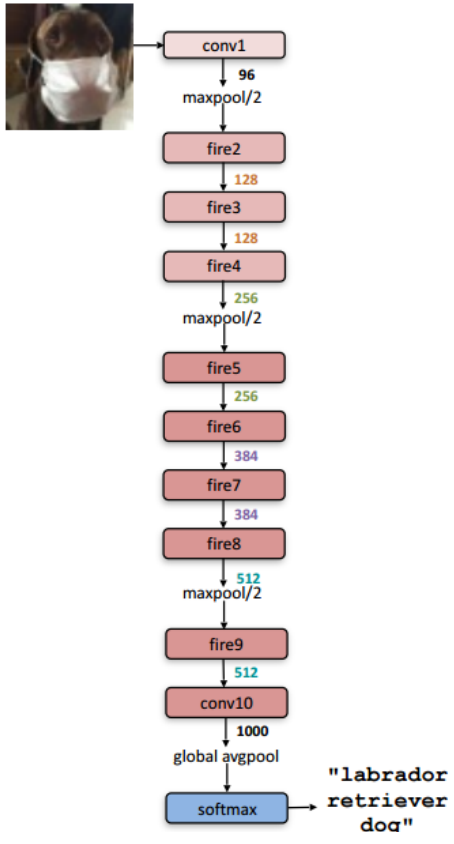
SqueezeNet: Fire Module

- Strategy 1. Replace 3x3 filters with 1x1 filters
- Strategy 2. Decrease the number of input channels to 3x3 filters
- Strategy 3. Downsample late in the network so that convolution layers have large activation maps

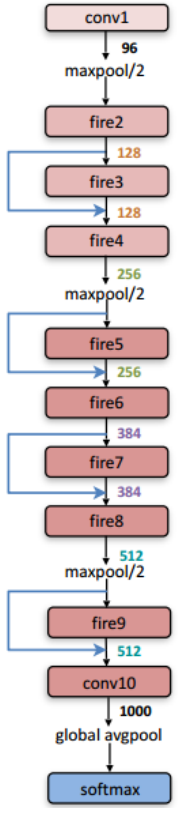


SqueezeNet: Fire Module

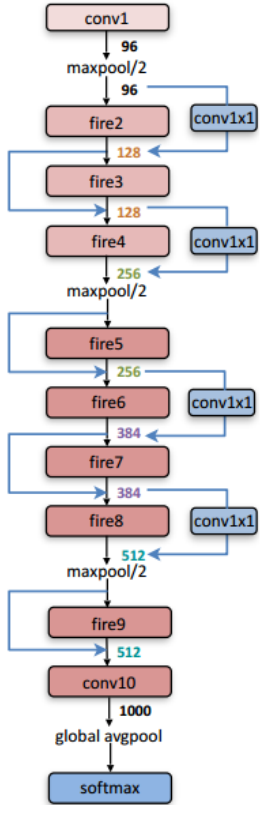
SqueezeNet



SqueezeNet with simple bypass



SqueezeNet with complex bypass



SqueezeNet: Fire Module

Architecture	Top-1 Accuracy	Top-5 Accuracy	Model Size
Vanilla SqueezeNet	57.5%	80.3%	4.8MB
SqueezeNet + Simple Bypass	**60.4%**	**82.5%**	4.8MB
SqueezeNet + Complex Bypass	58.8%	82.0%	7.7MB

SqueezeNet: Fire Module

Compression Approach	DNN Architecture	Original Model Size	Compressed Model Size	Reduction in Model Size vs. AlexNet	Top-1 ImageNet Accuracy	Top-5 ImageNet Accuracy
None (baseline)	AlexNet	240MB	240MB	1x	57.2%	80.3%
SVD	AlexNet	240MB	48MB	5x	56.0%	79.4%
Network Pruning	AlexNet	240MB	27MB	9x	57.2%	80.3%
Deep Compression	AlexNet	240MB	6.9MB	35x	57.2%	80.3%
None	**SqueezeNet**	**4.8MB**	**4.8MB**	**50x**	57.5%	80.3%
Deep Compression	**SqueezeNet**	**4.8MB**	**0.52MB**	**461x**	57.5%	80.3%

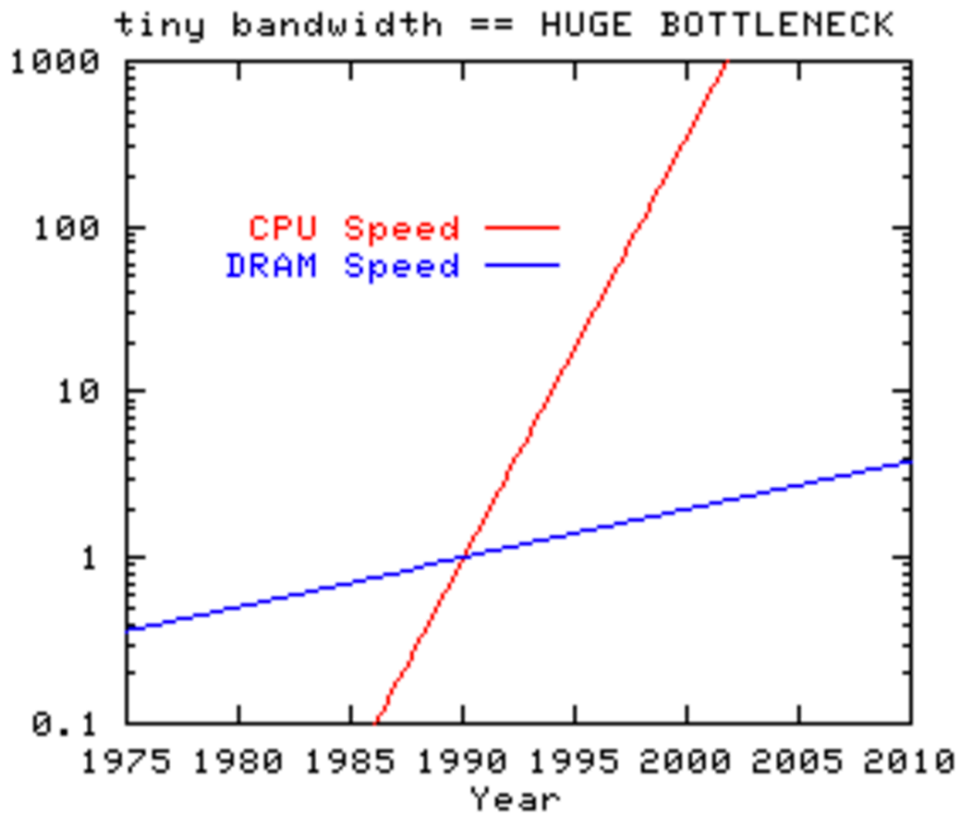
SqueezeNet: Fire Module

- Enormous DNN models training speedup

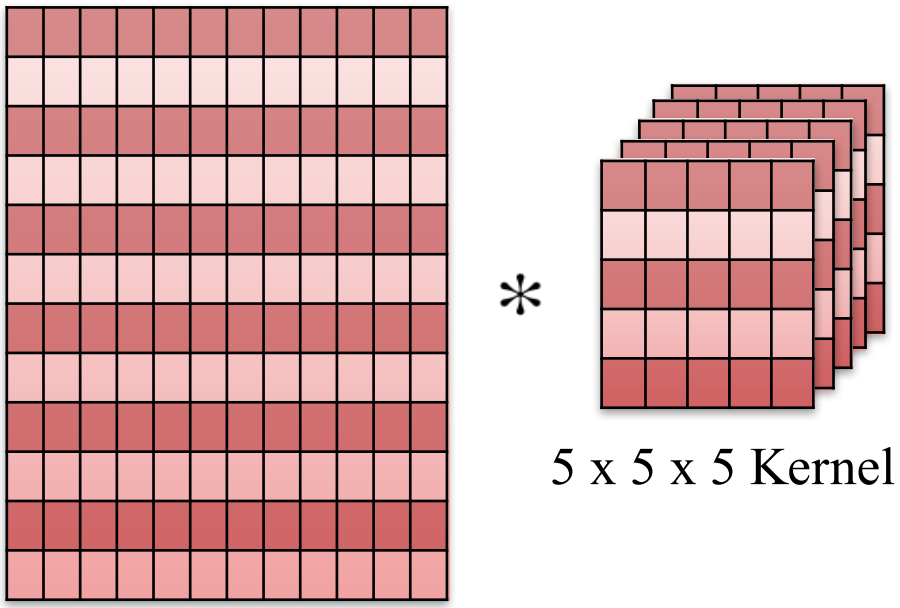
	Hardware	Net	Batch Size	Initial Learning Rate	Epochs	Train time	Speedup	Top-1 ImageNet Accuracy	Top-5 ImageNet Accuracy
Caffe	single K20 GPU	GoogLe-Net	32	0.01	64	**20.5 days**	1x	68.3%	88.7%
FireCaffe	32 K20 GPUs (Titan cluster)	GoogLe-Net	1024	0.08	72	**23.4 hours**	20x	68.3%	88.7%
FireCaffe	128 K20 GPUs (Titan cluster)	GoogLe-Net	1024	0.08	72	**10.5 hours**	47x	68.3%	88.7%

Bandwidth vs Compute Limitations

- Deep learning based applications either limited by bandwidth or compute speed (usually bandwidth!)
- Gains in processer performance via increased clock speeds and outpaced memory performance
- Known as "*Memory Gap*" or "*Memory Wall*"
- Now standard for small amounts of SRAM to be integrated onto the processor chip as various levels of cache memory



Bandwidth vs Compute Limitations

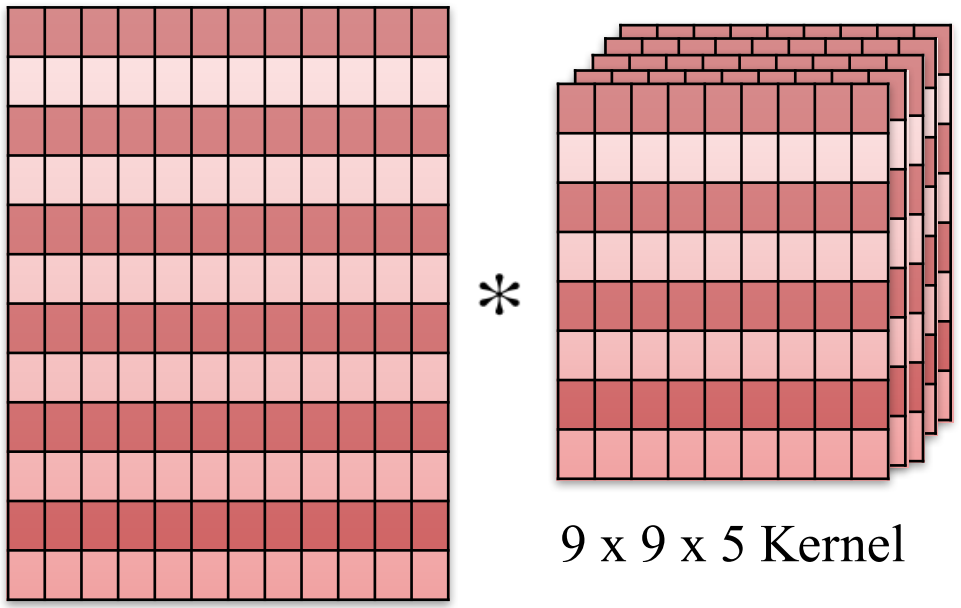


13 x 13 Image

5 x 5 x 5 Kernel

- Assume that we have the image and the weights in internal cache
- We have to perform convolution and write the feature map back to DRAM / main memory
- Say we have 25 cores / DSP multipliers, and every clock we can do 25 multiplications in parallel
- We get 1 output per 25 multiplications. Also assume that we are able to write one number back to DRAM per clock cycle (So totally we need 9 x 9 x 5 = 405 clock cycles)
- Thus, optimal usage of compute resources and memory bandwidth

Bandwidth vs Compute Limitations



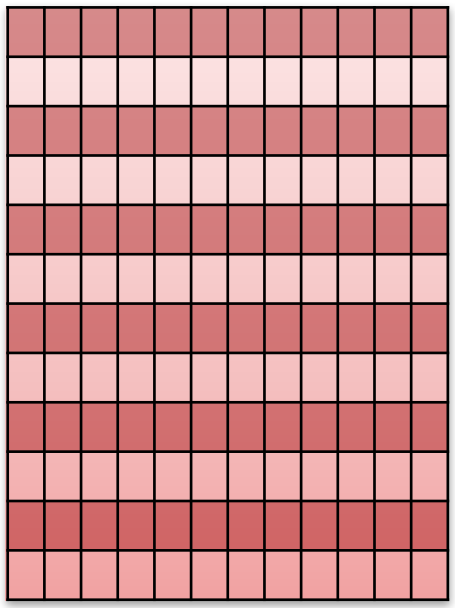
13 x 13 Image

*

9 x 9 x 5 Kernel

- Assume that we have the image and the weights in internal cache
- We have to perform convolution and write the feature map back to DRAM / main memory
- Say we have 25 cores / DSP multipliers, and every clock we can do 25 multiplications in parallel
- We get 1 output per 81 multiplications. So, we need 4 clock cycles for 1 output and totally 4 x 5 x 5 x 5 = 500
- Thus, we are compute limited

Bandwidth vs Compute Limitations



13 x 13 Image

1x 1 x 5 Kernel

- Assume that we have the image and the weights in internal cache
- We have to perform convolution and write the feature map back to DRAM / main memory
- Say we have 25 cores / DSP multipliers, and every clock we can do 25 multiplications in parallel
- We get 1 output per 1 multiplications. So can we get up to 25 outputs per clock, but we are able to write only 1
- Thus, we are bandwidth limited

Fixed Point vs Floating Point Arithmetic

- Fixed Point
  - 11010.1_2

    $= 1 * + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0* 2^0 + 1 * 2^{-1}$

    $= 16 + 8 + 2 + 0.5$

    $= 26.5$
  - Negative Numbers: 2's compliment
  - Fixed <NOB, NOIB>, where NOB = total number of bits, NOIB = total number of integer bits
  - Fixed <6, 1> of 110101 = 26.5

Fixed Point vs Floating Point Arithmetic

- IEEE 754 Standard Floating Point Numbers - 32 bits of which
  - 1 bit = sign bit(s)
  - 8 = Biased exponent bits (e)
  - 23 = mantissa (m)

$$\text{Decimal Equivalent Value} = (-1)^{s} \times 1.m \times 2^{(e-bias)}$$

$$bias = 2^{(e-1)} - 1$$

0	1000 0001	0101 0000 0000 0000 0000 000
Sign bit (1)	Exponent (8) bits	Mantissa (23) bits

$$bias = 2^{(8-1)} - 1$$
$$= 127$$

$$\text{Decimal value} = (-1)^{0} \times 1.0101\ 0000\ 0000\ 0000\ 0000\ 000 \times 2^{(129-127)}$$
$$= 1.0101\ 0000\ 0000\ 0000\ 0000\ 000 \times 2^{(2)}$$
$$= 101.01\ 0000\ 0000\ 0000\ 0000\ 000 \text{ (Binary fraction)}$$

↓ Convert binary fraction to decimal

$$= (1x2^{2})+(0x2^{1})+(1x2^{0}) . +(0x2^{-1}) + (1x2^{-2}) + (0x2^{-3}) \ldots\ldots (0x2^{-21})$$
$$= (4+0+1+0+0.25)$$
$$= 5.25$$

Fixed Point vs Floating Point Arithmetic

- Trick: Force the weights output and weights to be fixed point, keep a reference floating point weight copy

  Training deep neural networks with low precision multiplications, Courbariaux et al., **ICLR 2015**

```
# NOB = Number Of Bits = bit-width
# NOIB = Number Of Integer Bits = position of the radix point = range
def fixed_point(X,NOB, NOIB):

  power = 2.**(NOB - NOIB)
  max = (2.**NOB)-1
  value = X*power
  value = round(value) # rounding
  value = numpy.clip(value, -max, max) # saturation arithmetic
  value = value/power
  return value
```

Fixed Point vs Floating Point Arithmetic

- Trick: Force the weights output and weights to be fixed point, keep a reference floating point weight copy

  Training deep neural networks with low precision multiplications, Courbariaux et al., **ICLR 2015**

```
In [1]: %run test.py

In [2]: fixed_point(1.25, 3, 1)
Out[2]: 1.25

In [3]: fixed_point(1.25, 3, 0)
Out[3]: 0.875

In [4]: fixed_point(1.25, 3, 2)
Out[4]: 1.5
```

$= 1 * 2^{-1} + 1 * 2^{-2} + 1 * 2^{-3}$

$= 1 * 2^{-1} + 1 * 2^{-2} + 1 * 2^{-3}$

$= 1 * 2^{0} + 1 * 2^{-1}$

Thank you!