Convolutional Neural Networks

A. M. Sadeghzadeh, Ph.D.

Sharif University of Technology
Computer Engineering Department (CE)
Data and Network Security Lab (DNSL)

March 12, 2023

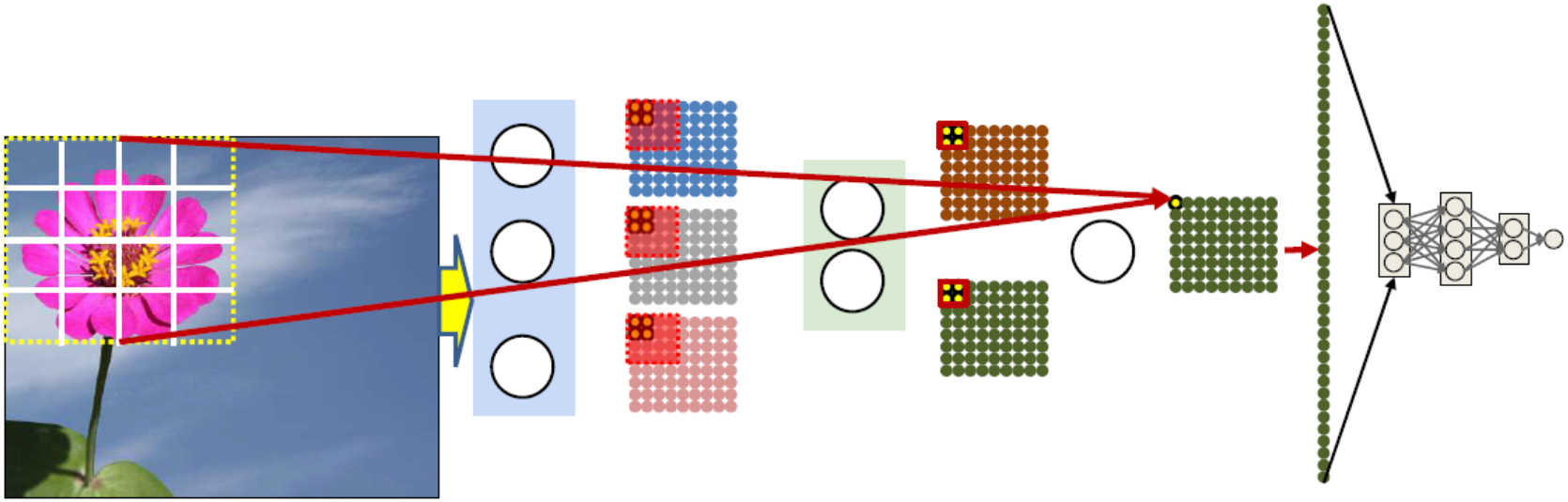
Most slides have been adapted from Justin Johnson, EECS 498-007, University of Michigan 2020, and Fei Fei Li, cs231n, Stanford 2017

Today's agenda

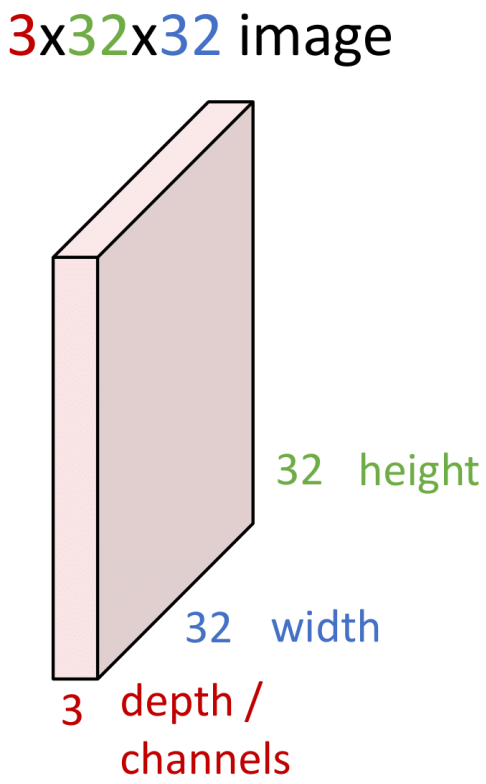
Recap

Convolutional Neural Networks (CNNs)

- Advantage of CNNs
  - Shift invariant
  - Parameter sharing
  - Sparse weights
  - Preserve spatial structures

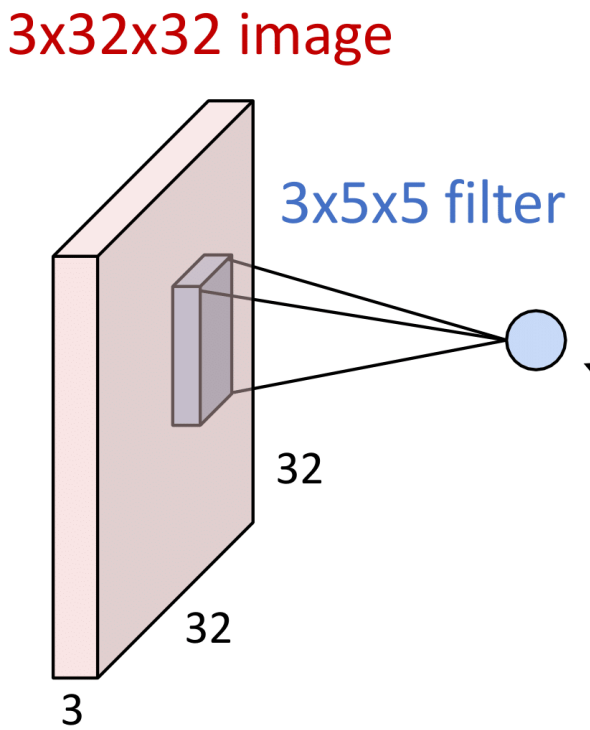


Convolution Layer



Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”

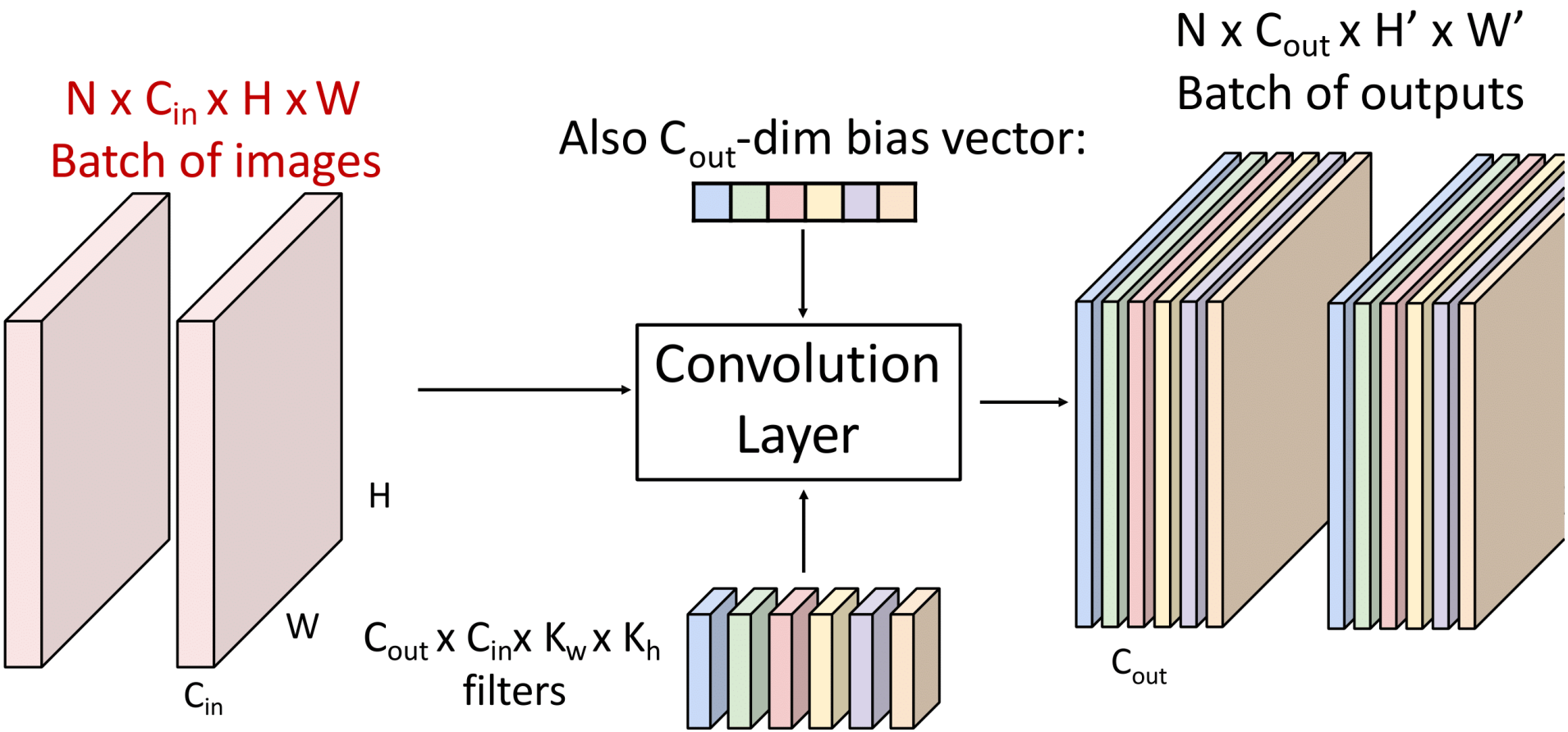
Convolution Layer



1 number:
the result of taking a dot product between the filter and a small 3x5x5 chunk of the image
(i.e. 3*5*5 = 75-dimensional dot product + bias)

$$w^T x + b$$

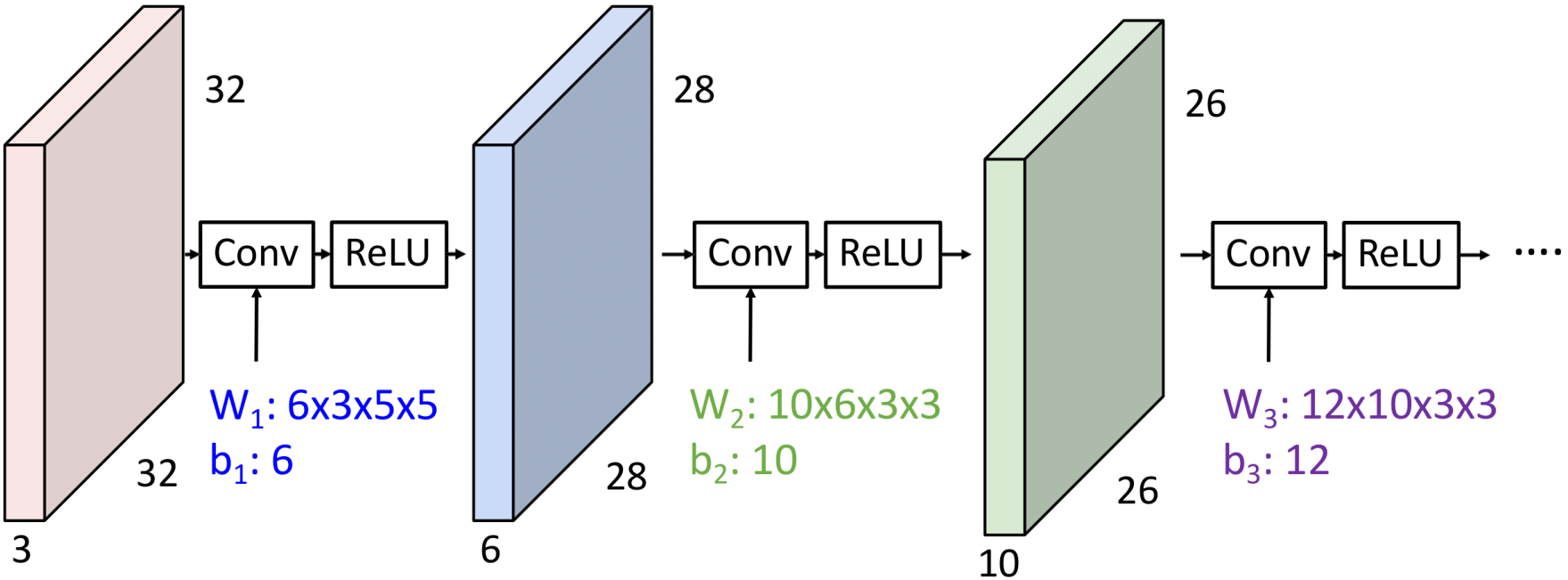
Convolution Layer



Stacking convolutions

Q: What happens if we stack two convolution layers? (Recall $y=W_2W_1x$ is a linear classifier)

A: We get another convolution!

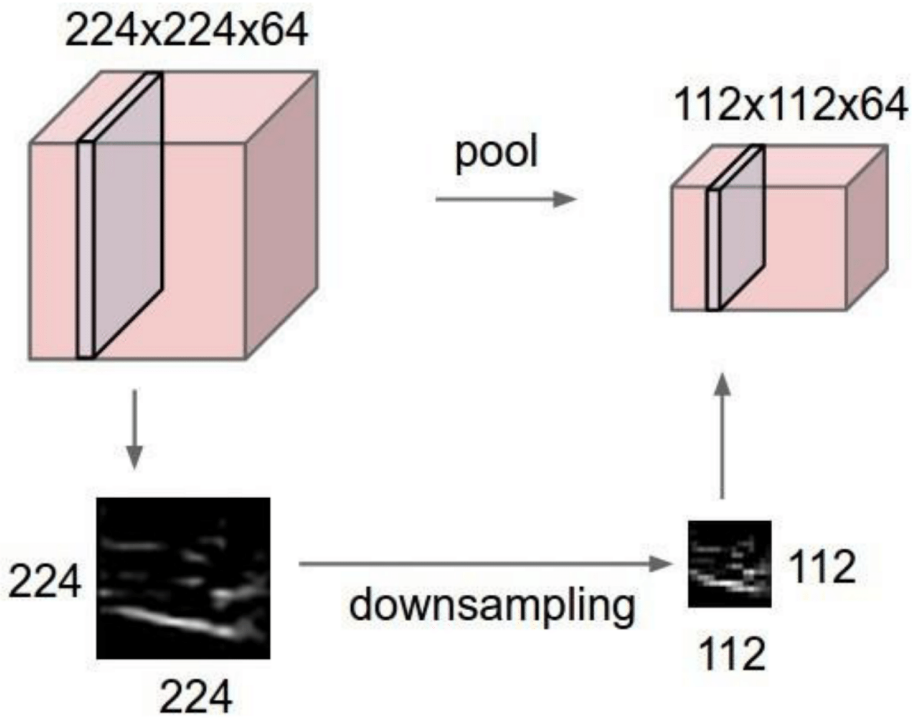


Input:
N x 3 x 32 x 32

First hidden layer:
N x 6 x 28 x 28

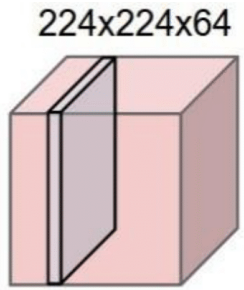
Second hidden layer:
N x 10 x 26 x 26

Pooling Layers: Another way to downsample

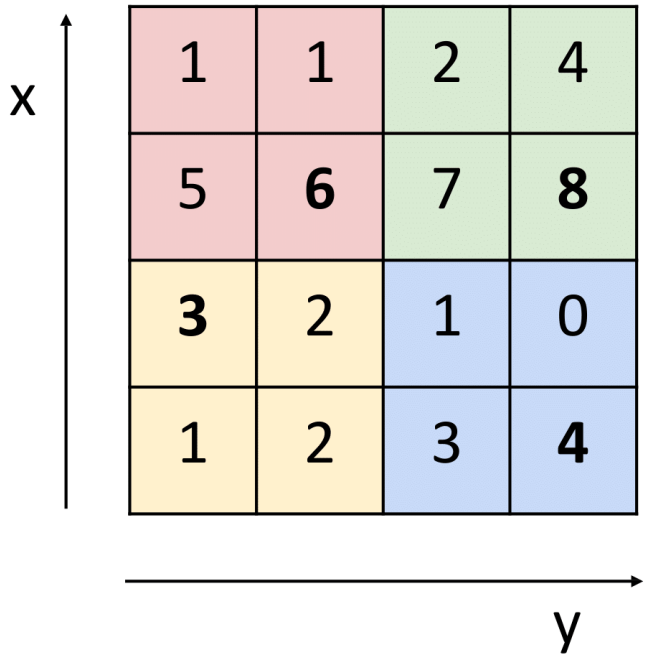


Hyperparameters:
Kernel Size
Stride
Pooling function

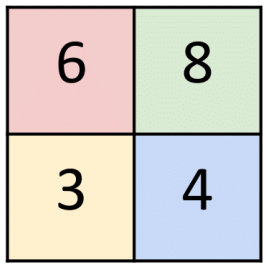
Max Pooling



Single depth slice



Max pooling with 2x2 kernel size and stride 2



Introduces **invariance** to small spatial shifts

No learnable parameters!

Pooling Summary

Input: C x H x W

Hyperparameters:

- Kernel size: K
- Stride: S
- Pooling function (max, avg)

Output: C x H' x W' where

- H' = (H – K) / S + 1
- W' = (W – K) / S + 1

Learnable parameters: None!

Common settings:

max, K = 2, S = 2

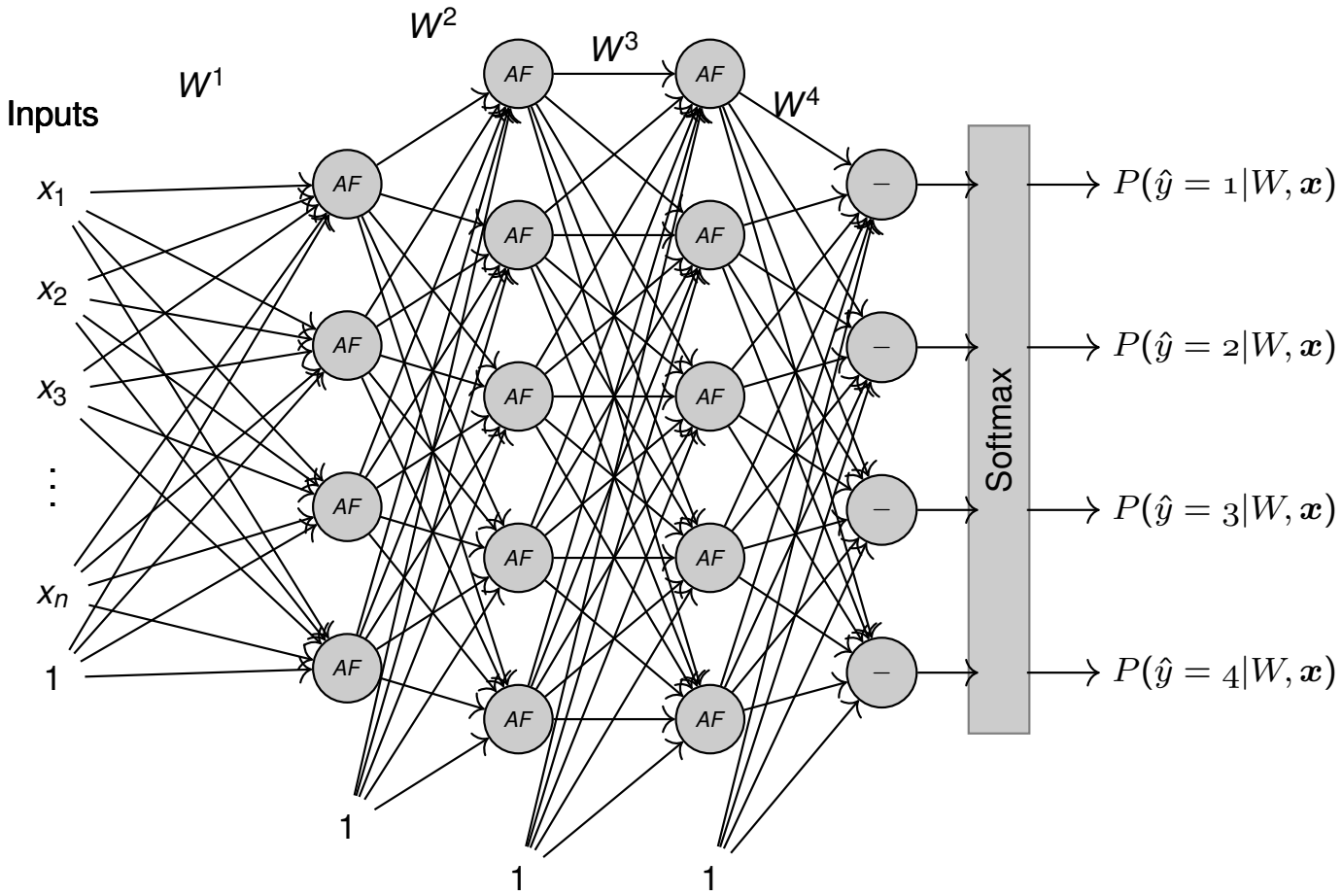
max, K = 3, S = 2 (AlexNet)

Vanishing/Exploding Gradient

The vanishing gradient problem for deep networks

- A particular problem with training deep networks
  - The gradient of the error with respect to weights is unstable

Fully connected neural networks



$$\hat{y} = f^N(W^N f^{N-1}(W^{N-1} f^{N-2}(...f^2(W^2(f^1(W^1\boldsymbol{x}))))))$$

The problem with training deep networks

- A multilayer perceptron is a nested function

$$\hat{y} = f^N(W^N f^{N-1}(W^{N-1} f^{N-2}(...f^2(W^2(f^1(W^1\boldsymbol{x})))))$$

  - W^k is the weights matrix at the k^{th} layer
  - f^k is the activation function of the k^{th} layer
- The error for X can be written as

$$L = CE(f^N(W^N f^{N-1}(W^{N-1} f^{N-2}(...f^2(W^2(f^1(W^1\boldsymbol{x})))))), y)$$

  - CE is cross entropy

Training deep networks

- Let $\boldsymbol{z}^{\ell} = W^{\ell}\boldsymbol{o}^{\ell-1}$ and $\boldsymbol{o}^{\ell} = f^{\ell}(\boldsymbol{z}^{\ell})$

$$\frac{\partial L}{\partial \boldsymbol{o}^{\ell-1}} = ---$$

Training deep networks

- Let $\boldsymbol{z}^{\ell} = W^{\ell}\boldsymbol{o}^{\ell-1}$ and $\boldsymbol{o}^{\ell} = f^{\ell}(\boldsymbol{z}^{\ell})$

$$\frac{\partial L}{\partial \boldsymbol{o}^{\ell-1}} = --\frac{\partial L}{\partial \boldsymbol{o}^{\ell}}$$

Training deep networks

- Let $\boldsymbol{z}^{\ell} = W^{\ell}\boldsymbol{o}^{\ell-1}$ and $\boldsymbol{o}^{\ell} = f^{\ell}(\boldsymbol{z}^{\ell})$

$$\frac{\partial L}{\partial \boldsymbol{o}^{\ell-1}} = -\frac{\partial \boldsymbol{o}^{\ell}}{\partial \boldsymbol{z}^{\ell}}\frac{\partial L}{\partial \boldsymbol{o}^{\ell}}$$

Training deep networks

- Let $\boldsymbol{z}^\ell = W^\ell \boldsymbol{o}^{\ell-1}$ and $\boldsymbol{o}^\ell = f^\ell(\boldsymbol{z}^\ell)$

$$\frac{\partial L}{\partial \boldsymbol{o}^{\ell-1}} = \frac{\partial \boldsymbol{z}^\ell}{\partial \boldsymbol{o}^{\ell-1}} \frac{\partial \boldsymbol{o}^\ell}{\partial \boldsymbol{z}^\ell} \frac{\partial L}{\partial \boldsymbol{o}^\ell}$$

Training deep networks

- Let $\boldsymbol{z}^{\ell} = W^{\ell}\boldsymbol{o}^{\ell-1}$ and $\boldsymbol{o}^{\ell} = f^{\ell}(\boldsymbol{z}^{\ell})$

$$\frac{\partial L}{\partial \boldsymbol{o}^{\ell-1}} = \frac{\partial \boldsymbol{z}^{\ell}}{\partial \boldsymbol{o}^{\ell-1}} \frac{\partial \boldsymbol{o}^{\ell}}{\partial \boldsymbol{z}^{\ell}} \frac{\partial L}{\partial \boldsymbol{o}^{\ell}}$$

$$\nabla_{\boldsymbol{o}^{\ell-1}} L = W^{\ell} . \nabla_{\boldsymbol{z}^{\ell}} f^{\ell} . \nabla_{\boldsymbol{o}^{\ell}} L$$

- Where
  - $\nabla_{\boldsymbol{x}} f$ is the jacobian matrix of $f(\boldsymbol{x})$ w.r.t $\boldsymbol{x}$

Training deep networks

- For

$$L = CE(f^N(W^N f^{N-1}(W^{N-1} f^{N-2}(...f^1(W^1 \boldsymbol{x})))), y)$$

Training deep networks

- For

$$L = CE(f^N(W^N f^{N-1}(W^{N-1} f^{N-2}(...f^1(W^1 \boldsymbol{x})))), y)$$

- We get

$$\nabla_{W^k} L = \boldsymbol{o}^{k-1}.\nabla_{\boldsymbol{z}^k} f^k.W^{k+1}.\nabla_{\boldsymbol{z}^{k+1}} f^{k+1}.....W^{N-1}.\nabla_{\boldsymbol{z}^{N-1}} f^{N-1}.W^N.\nabla_{\boldsymbol{z}^N} f^N.\nabla_{\hat{\boldsymbol{y}}} CE$$

$$\nabla_{W^k} L = \boldsymbol{o}^{k-1}.\nabla_{\boldsymbol{z}^k} f^k.\left(\prod_{i=k+1}^{N} W^i.\nabla_{\boldsymbol{z}^i} f^i\right).\nabla_{\hat{\boldsymbol{y}}} CE$$

- Where
  - W^N is the gradient of the affine transforation $\boldsymbol{z}^n$ w.r.t the output of the $n-1^{th}$ layer ($\boldsymbol{o}^{n-1}$) of the network
  - $\nabla_{\boldsymbol{z}^n} f^n$ is gradient of activation function $f^n(.)$ w.r.t to affine transforation $\boldsymbol{z}^n$ in the n^{th} layer of the network
  - $\boldsymbol{o}^{n-1}$ is the gradient of the affine transforation $\boldsymbol{z}^n$ w.r.t the weights of the n^{th} layer (W^n) of the network
  - $\nabla_{\hat{\boldsymbol{y}}} CE$ is the gradient of cross entropy w.r.t the output of the network ($\hat{\boldsymbol{y}}$)

Training deep networks

- For

$$L = CE(f^N(W^N f^{N-1}(W^{N-1} f^{N-2}(...f^1(W^1 \boldsymbol{x})))), y)$$

- We get

$$\nabla_{W^k} L = \boldsymbol{o}^{k-1}.\nabla_{\boldsymbol{z}^k} f^k.W^{k+1}.\nabla_{\boldsymbol{z}^{k+1}} f^{k+1}.....W^{N-1}.\nabla_{\boldsymbol{z}^{N-1}} f^{N-1}.W^N.\nabla_{\boldsymbol{z}^N} f^N.\nabla_{\hat{\boldsymbol{y}}} CE$$

$$\nabla_{W^k} L = \boldsymbol{o}^{k-1}.\nabla_{\boldsymbol{z}^k} f^k.\left(\prod_{i=k+1}^{N} W^i.\nabla_{\boldsymbol{z}^i} f^i\right).\nabla_{\hat{\boldsymbol{y}}} CE$$

- Where
  - W^N is the gradient of the affine transforation $\boldsymbol{z}^n$ w.r.t the output of the $n-1^{th}$ layer ($\boldsymbol{o}^{n-1}$) of the network
  - $\nabla_{\boldsymbol{z}^n} f^n$ is gradient of activation function $f^n(.)$ w.r.t to affine transforation $\boldsymbol{z}^n$ in the n^{th} layer of the network
  - $\boldsymbol{o}^{n-1}$ is the gradient of the affine transforation $\boldsymbol{z}^n$ w.r.t the weights of the n^{th} layer (W^n) of the network
  - $\nabla_{\hat{\boldsymbol{y}}} CE$ is the gradient of cross entropy w.r.t the output of the network ($\hat{\boldsymbol{y}}$)

Training deep networks

- For

$$L = CE(f^N(W^N f^{N-1}(W^{N-1} f^{N-2}(...f^1(W^1 \boldsymbol{x})))), y)$$

- We get

$$\nabla_{W^k} L = \boldsymbol{o}^{k-1}.\nabla_{\boldsymbol{z}^k} f^k.W^{k+1}.\nabla_{\boldsymbol{z}^{k+1}} f^{k+1}.....W^{N-1}.\nabla_{\boldsymbol{z}^{N-1}} f^{N-1}.W^N.\nabla_{\boldsymbol{z}^N} f^N.\nabla_{\hat{\boldsymbol{y}}} CE$$

$$\nabla_{W^k} L = \boldsymbol{o}^{k-1}.\nabla_{\boldsymbol{z}^k} f^k.\left(\prod_{i=k+1}^{N} W^i.\nabla_{\boldsymbol{z}^i} f^i\right).\nabla_{\hat{\boldsymbol{y}}} CE$$

- Where
  - W^N is the gradient of the affine transforation $\boldsymbol{z}^n$ w.r.t the output of the $n-1^{th}$ layer ($\boldsymbol{o}^{n-1}$) of the network
  - $\nabla_{\boldsymbol{z}^n} f^n$ is gradient of activation function $f^n(.)$ w.r.t to affine transforation $\boldsymbol{z}^n$ in the n^{th} layer of the network
  - $\boldsymbol{o}^{n-1}$ is the gradient of the affine transforation $\boldsymbol{z}^n$ w.r.t the weights of the n^{th} layer (W^n) of the network
  - $\nabla_{\hat{\boldsymbol{y}}} CE$ is the gradient of cross entropy w.r.t the output of the network ($\hat{\boldsymbol{y}}$)

Training deep networks

- For

$$L = CE(f^N(W^N f^{N-1}(W^{N-1} f^{N-2}(...f^1(W^1 \boldsymbol{x})))), y)$$

- We get

$$\nabla_{W^k} L = \boldsymbol{o}^{k-1} . \nabla_{\boldsymbol{z}^k} f^k . W^{k+1} . \nabla_{\boldsymbol{z}^{k+1}} f^{k+1} ..... W^{N-1} . \nabla_{\boldsymbol{z}^{N-1}} f^{N-1} . W^N . \nabla_{\boldsymbol{z}^N} f^N . \nabla_{\hat{\boldsymbol{y}}} CE$$

$$\nabla_{W^k} L = \boldsymbol{o}^{k-1} . \nabla_{\boldsymbol{z}^k} f^k . \left(\prod_{i=k+1}^{N} W^i . \nabla_{\boldsymbol{z}^i} f^i \right) . \nabla_{\hat{\boldsymbol{y}}} CE$$

- Repeated multiplication by the weights matrix and jacobian matrices of activation functions will result in exploding or vanishing gradients
  - Depends on the spectral norm (largest singular value) of jacobian matrices
- The gradients in the lower/earlier layers can explode or vanish
  - Resulting in insignificant or unstable gradient descent updates
  - Problem gets worse as network depth increases

Batch Normalization

Batch Normalization

Idea: “Normalize” the outputs of a layer so they have zero mean and unit variance

Why? Helps reduce “internal covariate shift”, improves optimization

We can normalize a batch of activations like this:

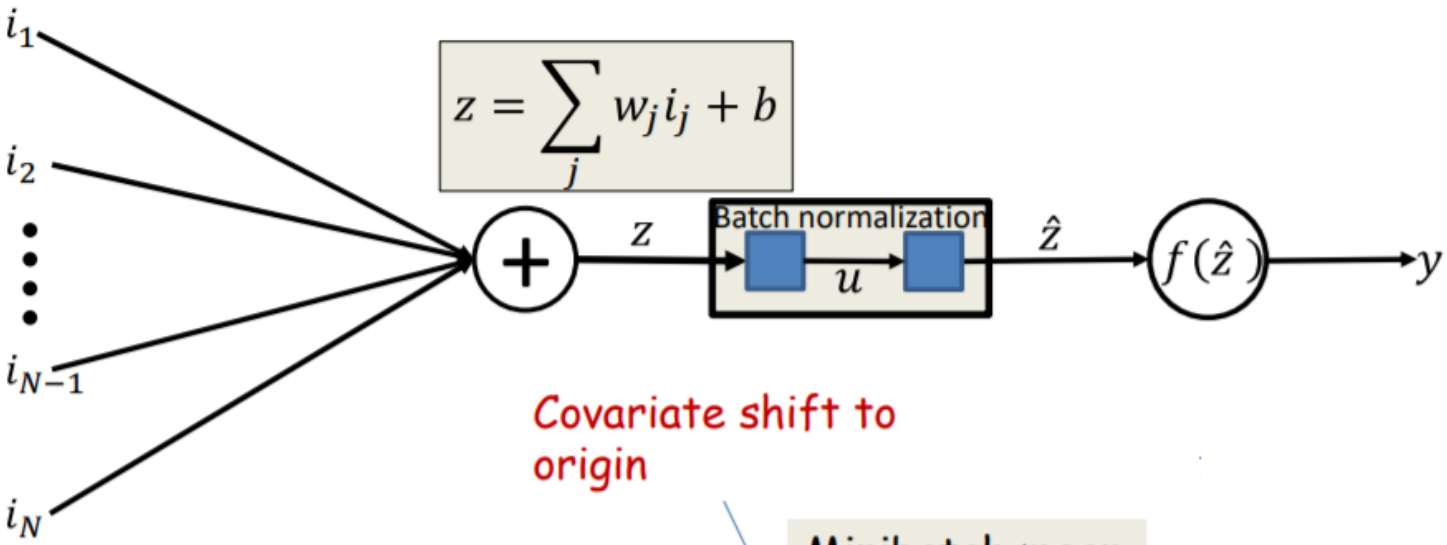
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

This is a **differentiable function**, so we can use it as an operator in our networks and backprop through it!

Ioffe and Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift”, ICML 2015

- "We define Internal Covariate Shift as the change in the distribution of network activations due to the change in network parameters during training."

Batch Normalization



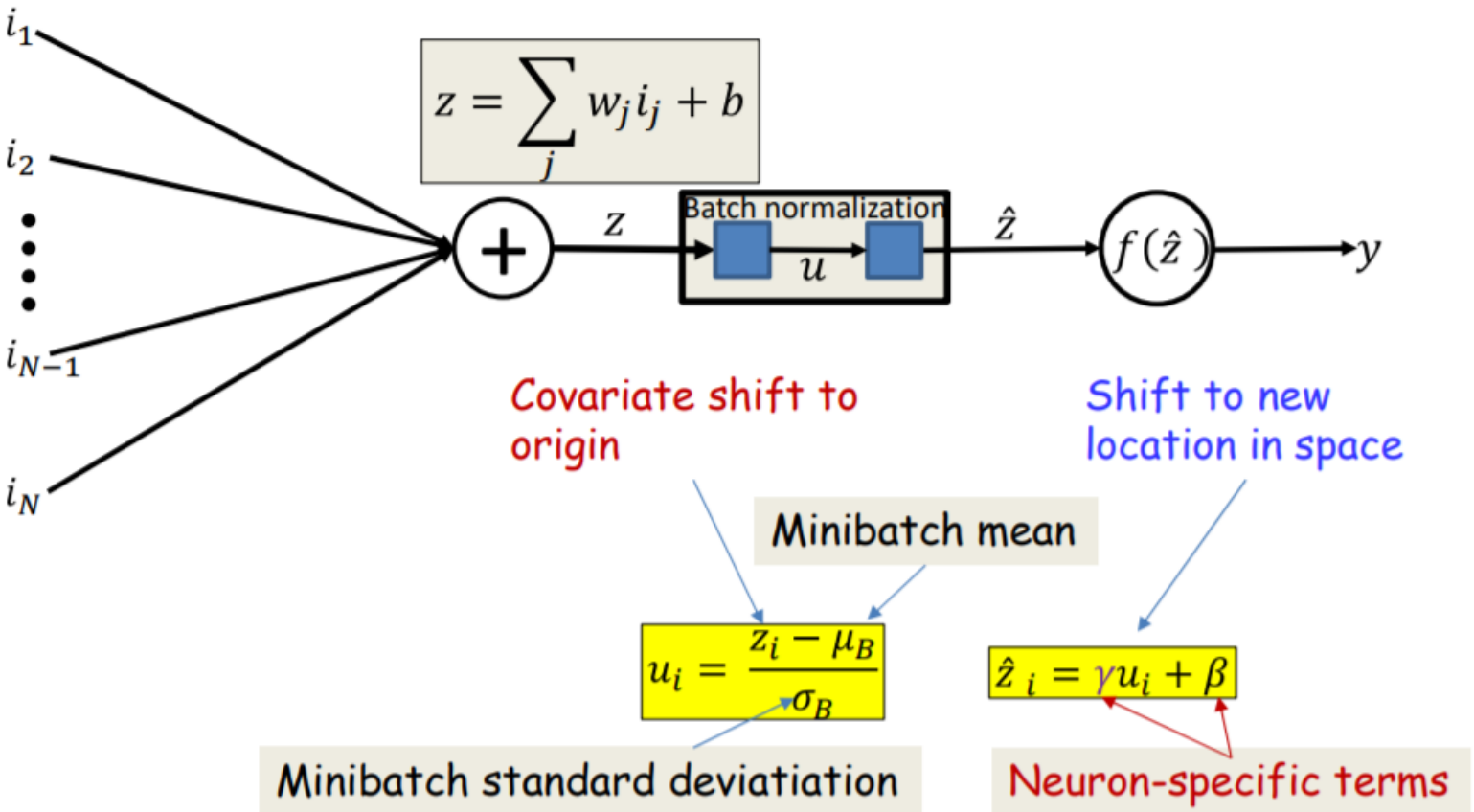
Covariate shift to origin

Minibatch mean

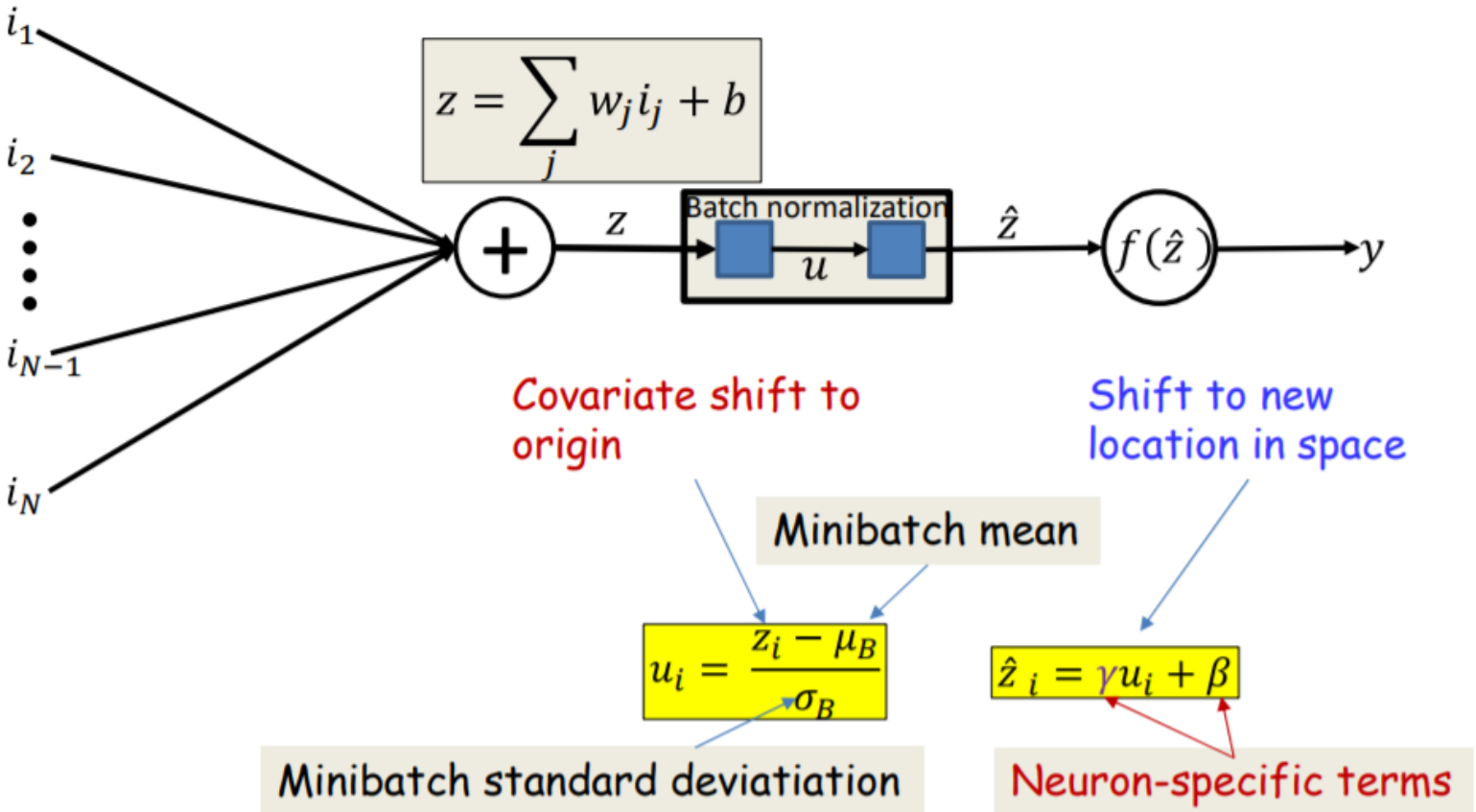
$$u_i = \frac{z_i - \mu_B}{\sigma_B}$$

Minibatch standard deviatiation

Batch Normalization



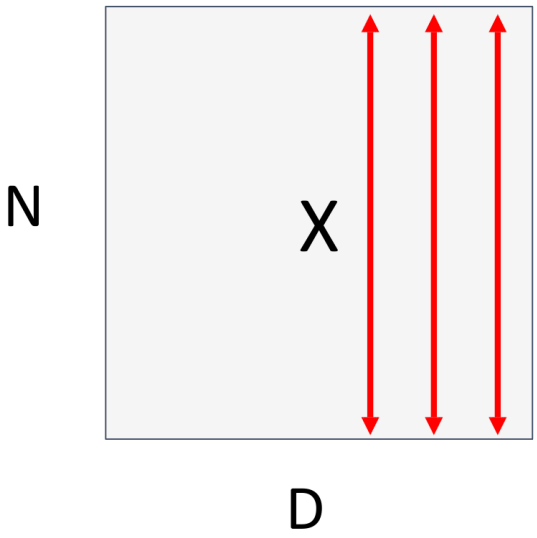
Batch Normalization



- Note that simply normalizing each input of a layer may change what the layer can represent. For instance, normalizingthe inputs of a sigmoid would constrain them to the linear regime of the nonlinearity. To address this, we make sure that the transformation inserted in the network can represent the identity transform.

Batch Normalization

Input: $x : N \times D$



$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$

Per-channel std, shape is D

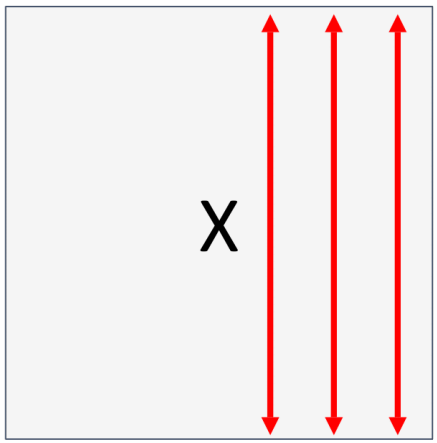
$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

Ioffe and Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", ICML 2015

Batch Normalization

Input: $x : N \times D$



$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$

Per-channel std, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

Problem: What if zero-mean, unit variance is too hard of a constraint?

Ioffe and Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", ICML 2015

Batch Normalization

Input: $x : N \times D$

Learnable scale and shift parameters:

$$\gamma, \beta : D$$

Learning $\gamma = \sigma$, $\beta = \mu$ will recover the identity function!

$$\mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2$$

Per-channel std, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

Batch Normalization: Test-Time

Problem: Estimates depend on minibatch; can't do this at test-time!

Input: $x : N \times D$

Learnable scale and shift parameters:

$$\gamma, \beta : D$$

Learning $\gamma = \sigma$, $\beta = \mu$ will recover the identity function!

$$\mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2$$

Per-channel std, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

Batch Normalization: Test-Time

Input: $x : N \times D$

Learnable scale and shift parameters:

$$\gamma, \beta : D$$

Learning $\gamma = \sigma$, $\beta = \mu$ will recover the identity function!

$$\mu_j = \text{(Running) average of values seen during training}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \text{(Running) average of values seen during training}$$

Per-channel std, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

Batch Normalization: Test-Time

Input: $x : N \times D$

Learnable scale and shift parameters:

$$\gamma, \beta : D$$

During testing batchnorm becomes a linear operator!
Can be fused with the previous fully-connected or conv layer

$$\mu_j = \text{(Running) average of values seen during training}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \text{(Running) average of values seen during training}$$

Per-channel std, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

Batch Normalization for ConvNets

Batch Normalization for **fully-connected** networks

$x: N \times D$

Normalize

$\mu, \sigma: 1 \times D$

$\gamma, \beta: 1 \times D$

$y = \gamma(x-\mu)/\sigma+\beta$

Batch Normalization for **convolutional** networks (Spatial Batchnorm, BatchNorm2D)

$x: N \times C \times H \times W$

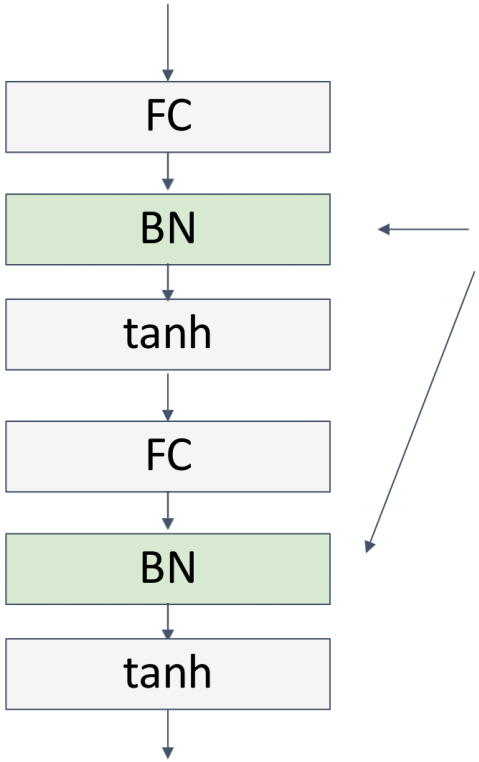
Normalize

$\mu, \sigma: 1 \times C \times 1 \times 1$

$\gamma, \beta: 1 \times C \times 1 \times 1$

$y = \gamma(x-\mu)/\sigma+\beta$

Batch Normalization

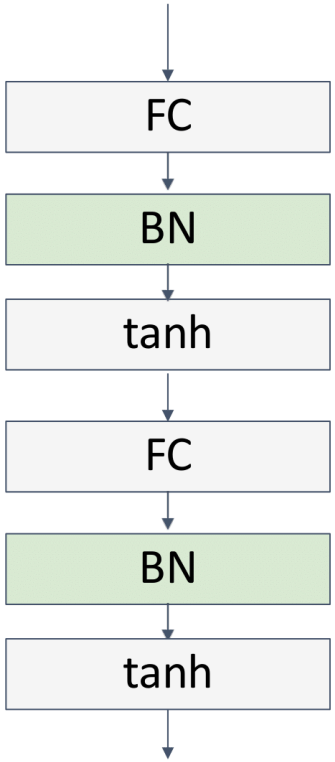


Ioffe and Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", ICML 2015

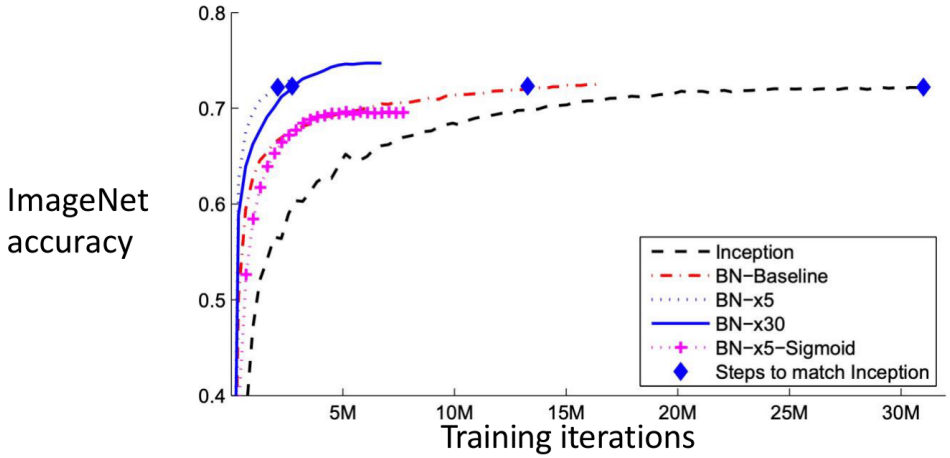
Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Batch Normalization

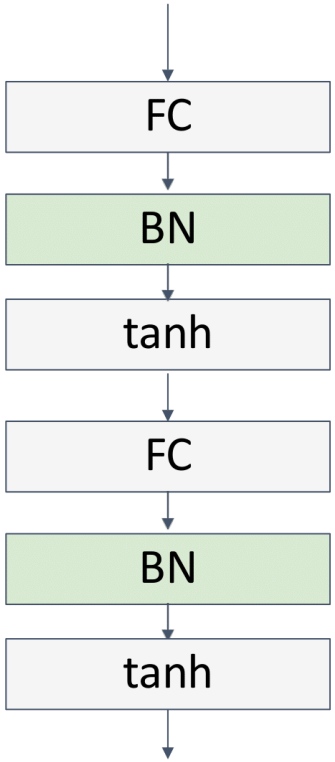


- Makes deep networks **much** easier to train!
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!



Ioffe and Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", ICML 2015

Batch Normalization

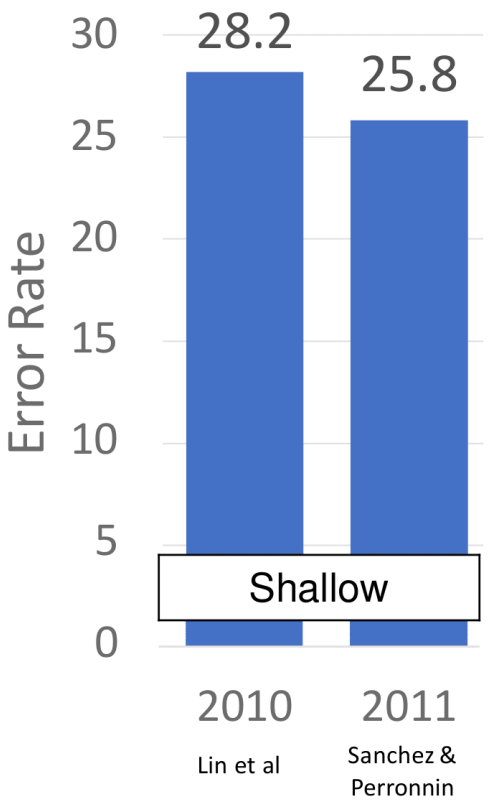


Ioffe and Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", ICML 2015

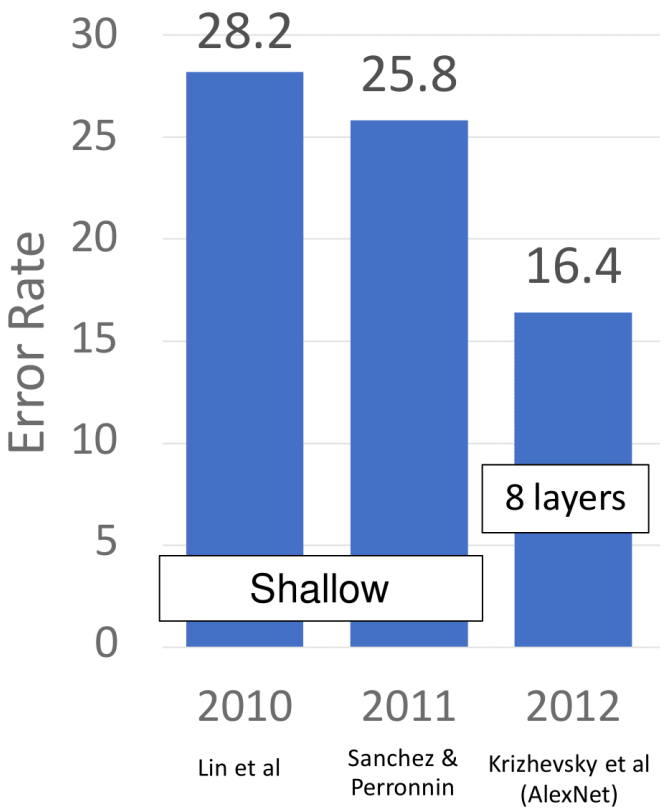
- Makes deep networks **much** easier to train!
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!
- Not well-understood theoretically (yet)
- Behaves differently during training and testing: this is a very common source of bugs!

CNN Architectures

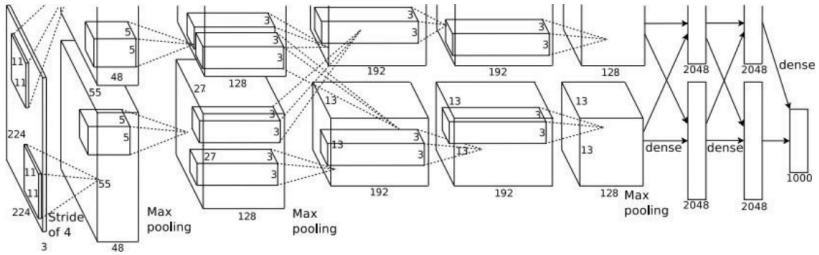
ImageNet classification challenge



ImageNet classification challenge

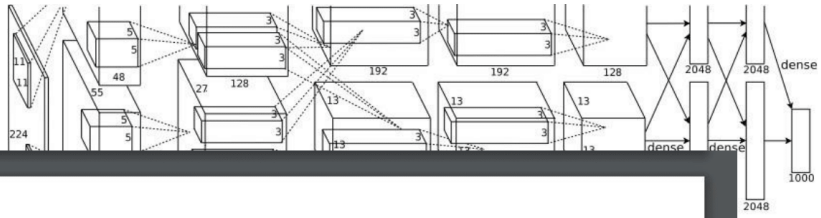


AlexNet



227 x 227 inputs
5 Convolutional layers
Max pooling
3 fully-connected layers
ReLU nonlinearities

AlexNet



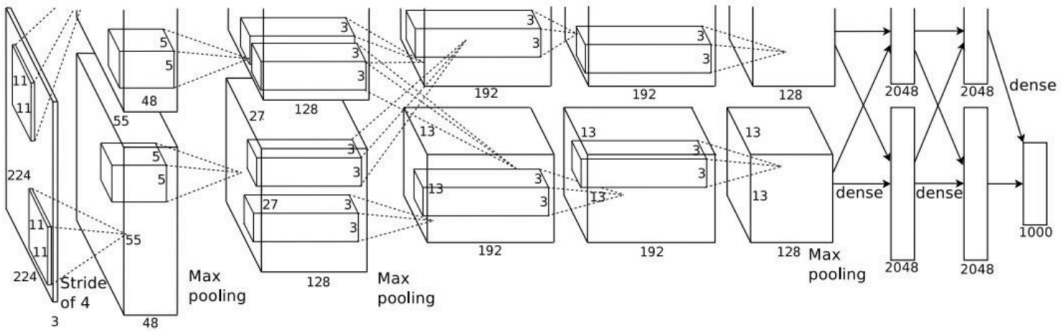
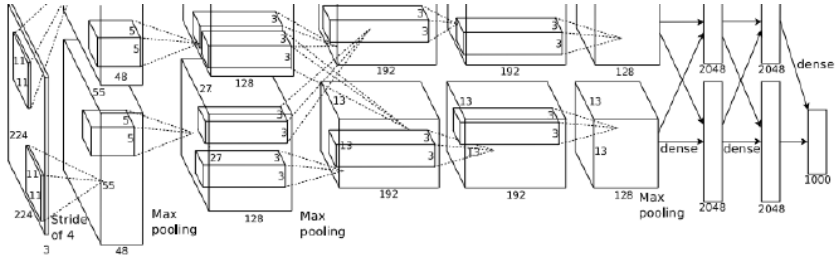


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

AlexNet

AlexNet

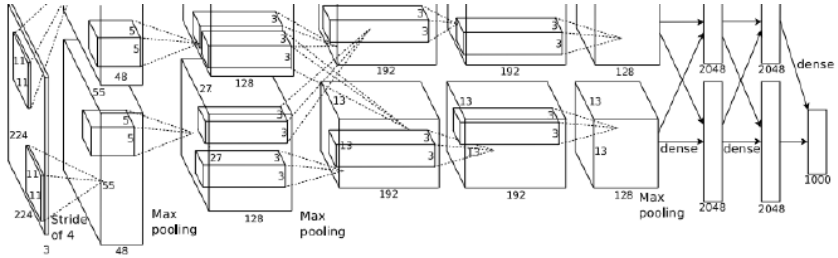


	Input size		Layer				Output size	
Layer	C	H / W	filters	kernel	stride	pad	C	H / W
conv1	3	227	64	11	4	2	?	

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

AlexNet

AlexNet



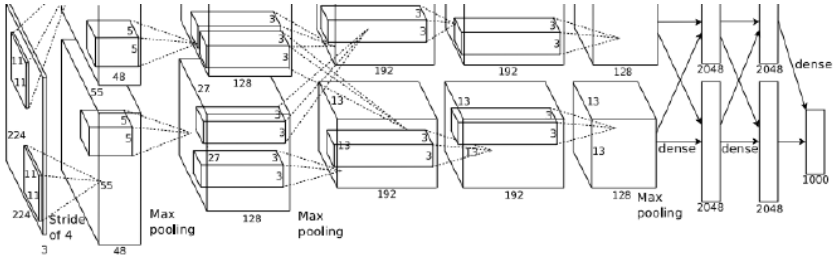
	Input size		Layer				Output size	
Layer	C	H / W	filters	kernel	stride	pad	C	H / W
conv1	3	227	64	11	4	2	64	?

Recall: Output channels = number of filters

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

AlexNet

AlexNet



	Input size		Layer				Output size	
Layer	C	H / W	filters	kernel	stride	pad	C	H / W
conv1	3	227	64	11	4	2	64	56

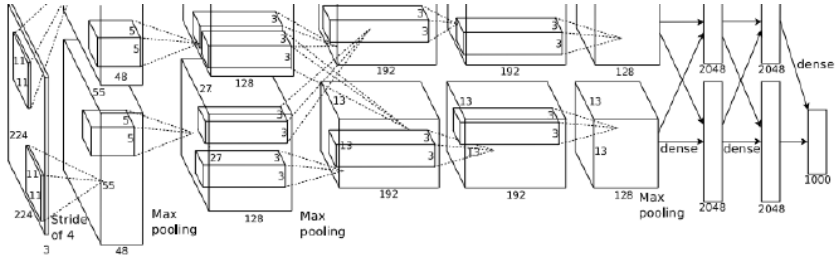
Recall: $W' = (W - K + 2P) / S + 1$

$= 227 - 11 + 2*2) / 4 + 1$

$= 220/4 + 1 = 56$

AlexNet

AlexNet

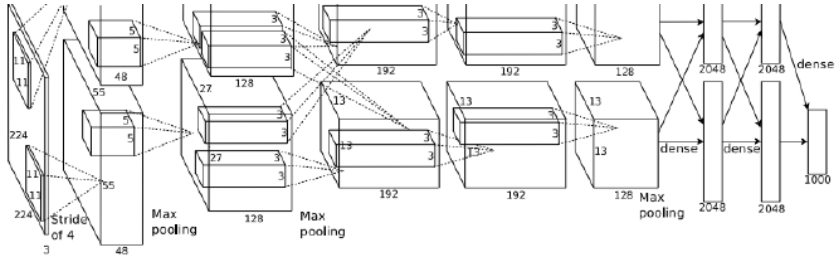


	Input size		Layer				Output size		
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)
conv1	3	227	64	11	4	2	64	56	?

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

AlexNet

AlexNet



	Input size		Layer				Output size		
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)
conv1	3	227	64	11	4	2	64	56	784

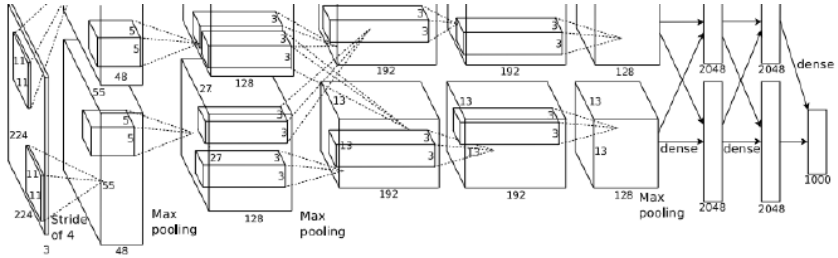
Number of output elements = C * H' * W'
= 64*56*56 = 200,704

Bytes per element = 4 (for 32-bit floating point)

KB = (number of elements) * (bytes per elem) / 1024
= 200704 * 4 / 1024
= **784**

AlexNet

AlexNet

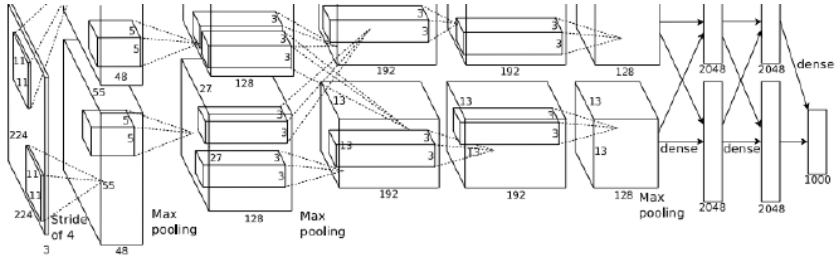


	Input size		Layer				Output size			
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)
conv1	3	227	64	11	4	2	64	56	784	?

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

AlexNet

AlexNet



	Input size		Layer				Output size			
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)
conv1	3	227	64	11	4	2	64	56	784	23

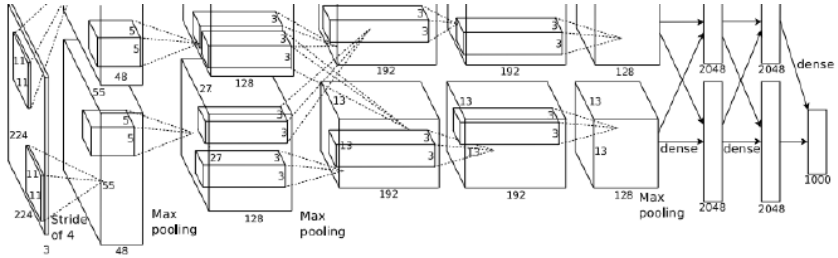
Weight shape = C_{out} x C_{in} x K x K
= 64 x 3 x 11 x 11

Bias shape = C_{out} = 64

Number of weights = 64*3*11*11 + 64
= **23,296**

AlexNet

AlexNet

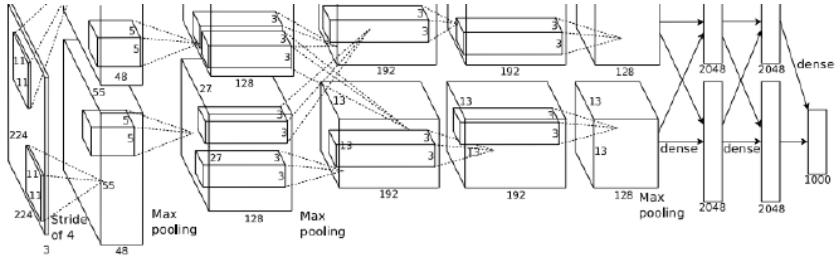


	Input size		Layer				Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)
conv1	3	227	64	11	4	2	64	56	784	23	?

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

AlexNet

AlexNet

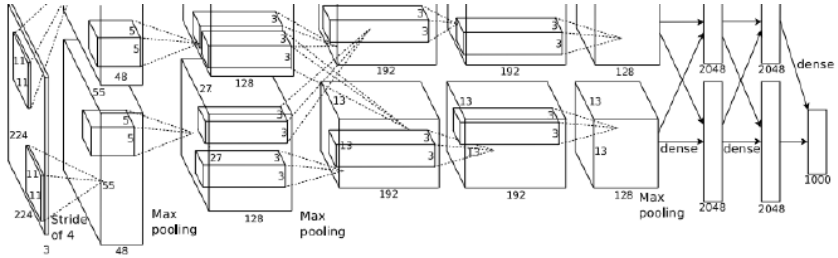


	Input size		Layer				Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)
conv1	3	227	64	11	4	2	64	56	784	23	73

Number of floating point operations (multiply+add)
= (number of output elements) * (ops per output elem)
= (C_{out} x H' x W') * (C_{in} x K x K)
= (64 * 56 * 56) * (3 * 11 * 11)
= 200,704 * 363
= **72,855,552**

AlexNet

AlexNet

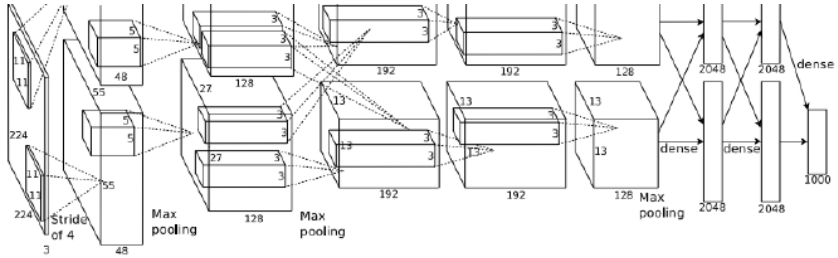


	Input size		Layer				Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)
conv1	3	227	64	11	4	2	64	56	784	23	73
pool1	64	56		3	2	0		?			

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

AlexNet

AlexNet



	Input size		Layer				Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)
conv1	3	227	64	11	4	2	64	56	784	23	73
pool1	64	56		3	2	0	64	27			

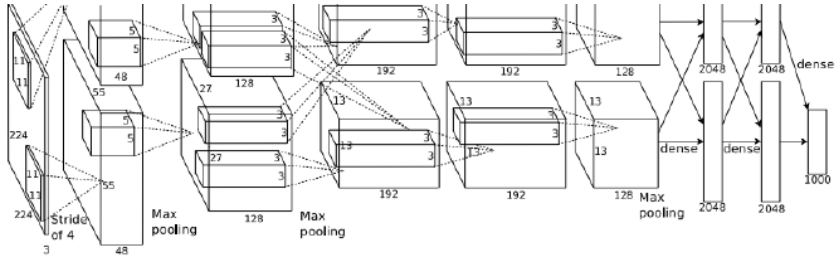
For pooling layer:

#output channels = #input channels = 64

W' = floor((W – K) / S + 1)
= floor(53 / 2 + 1) = floor(27.5) = **27**

AlexNet

AlexNet

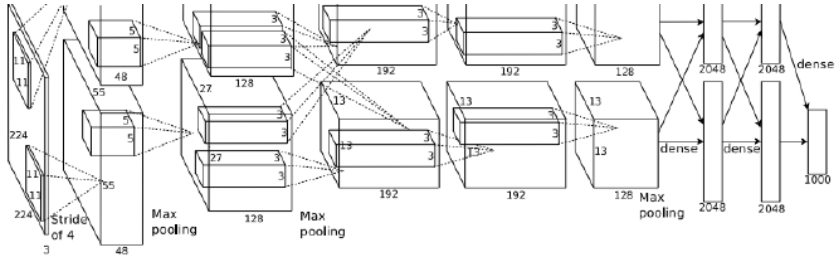


	Input size		Layer				Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)
conv1	3	227	64	11	4	2	64	56	784	23	73
pool1	64	56		3	2	0	64	27	182	?	

#output elems = C_{out} x H' x W'
Bytes per elem = 4
KB = C_{out} * H' * W' * 4 / 1024
= 64 * 27 * 27 * 4 / 1024
= **182.25**

AlexNet

AlexNet



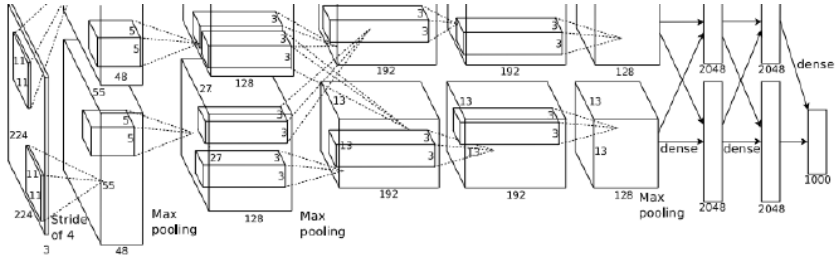
	Input size		Layer				Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)
conv1	3	227	64	11	4	2	64	56	784	23	73
pool1	64	56		3	2	0	64	27	182	0	?

Pooling layers have no learnable parameters!

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

AlexNet

AlexNet

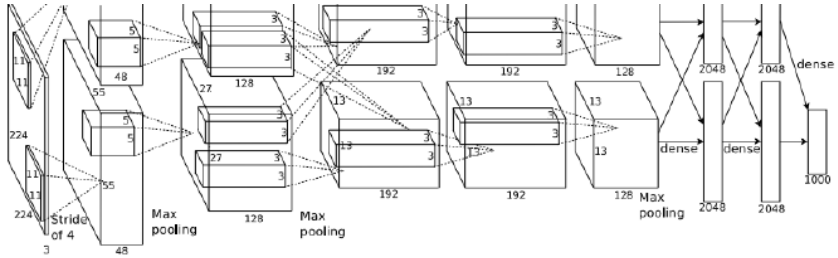


	Input size		Layer				Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)
conv1	3	227	64	11	4	2	64	56	784	23	73
pool1	64	56		3	2	0	64	27	182	0	0

Floating-point ops for pooling layer
= (number of output positions) * (flops per output position)
= (C_{out} * H' * W') * (K * K)
= (64 * 27 * 27) * (3 * 3)
= 419,904
= **0.4 MFLOP**

AlexNet

AlexNet



	Input size		Layer				Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)
conv1	3	227	64	11	4	2	64	56	784	23	73
pool1	64	56		3	2	0	64	27	182	0	0
conv2	64	27	192	5	1	2	192	27	547	307	224
pool2	192	27		3	2	0	192	13	127	0	0
conv3	192	13	384	3	1	1	384	13	254	664	112
conv4	384	13	256	3	1	1	256	13	169	885	145
conv5	256	13	256	3	1	1	256	13	169	590	100
pool5	256	13		3	2	0	256	6	36	0	0
flatten	256	6					9216		36	0	0

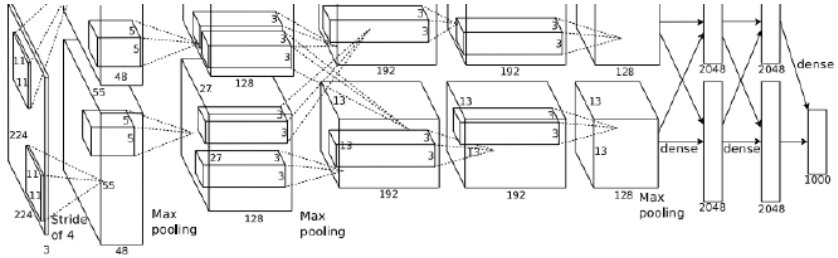
Flatten output size = C_{in} x H x W

= 256 * 6 * 6

= **9216**

AlexNet

AlexNet



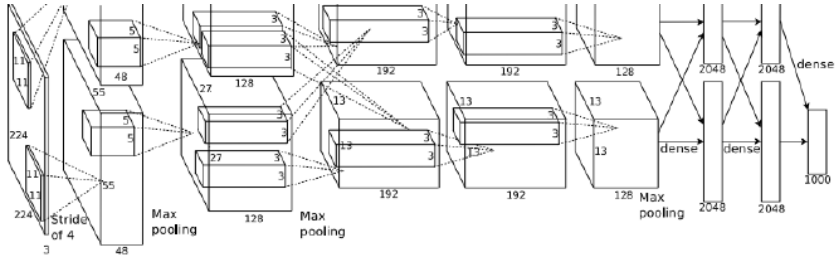
	Input size		Layer				Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)
conv1	3	227	64	11	4	2	64	56	784	23	73
pool1	64	56		3	2	0	64	27	182	0	0
conv2	64	27	192	5	1	2	192	27	547	307	224
pool2	192	27		3	2	0	192	13	127	0	0
conv3	192	13	384	3	1	1	384	13	254	664	112
conv4	384	13	256	3	1	1	256	13	169	885	145
conv5	256	13	256	3	1	1	256	13	169	590	100
pool5	256	13		3	2	0	256	6	36	0	0
flatten	256	6					9216		36	0	0
fc6	9216		4096				4096		16	37,749	38

FC params = $C_{in} * C_{out} + C_{out}$
= 9216 * 4096 + 4096
= 37,725,832

FC flops = $C_{in} * C_{out}$
= 9216 * 4096
= 37,748,736

AlexNet

AlexNet

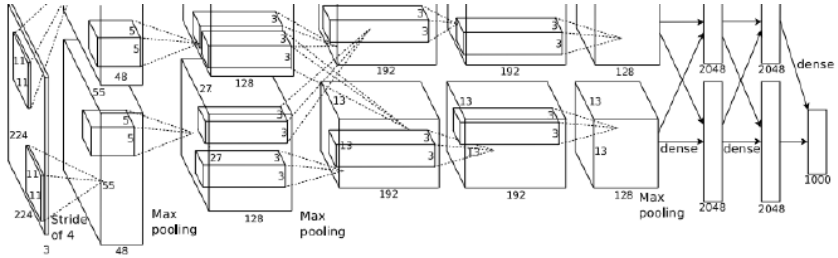


	Input size		Layer				Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)
conv1	3	227	64	11	4	2	64	56	784	23	73
pool1	64	56		3	2	0	64	27	182	0	0
conv2	64	27	192	5	1	2	192	27	547	307	224
pool2	192	27		3	2	0	192	13	127	0	0
conv3	192	13	384	3	1	1	384	13	254	664	112
conv4	384	13	256	3	1	1	256	13	169	885	145
conv5	256	13	256	3	1	1	256	13	169	590	100
pool5	256	13		3	2	0	256	6	36	0	0
flatten	256	6					9216		36	0	0
fc6	9216		4096				4096		16	37,749	38
fc7	4096		4096				4096		16	16,777	17
fc8	4096		1000				1000		4	4,096	4

AlexNet

AlexNet

How to choose this?
Trial and error =(

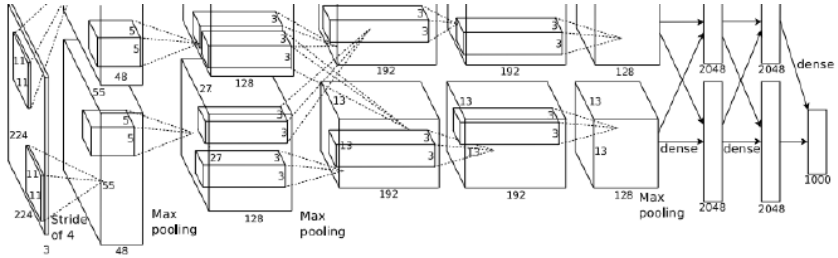


	Input size		Layer				Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)
conv1	3	227	64	11	4	2	64	56	784	23	73
pool1	64	56		3	2	0	64	27	182	0	0
conv2	64	27	192	5	1	2	192	27	547	307	224
pool2	192	27		3	2	0	192	13	127	0	0
conv3	192	13	384	3	1	1	384	13	254	664	112
conv4	384	13	256	3	1	1	256	13	169	885	145
conv5	256	13	256	3	1	1	256	13	169	590	100
pool5	256	13		3	2	0	256	6	36	0	0
flatten	256	6					9216		36	0	0
fc6	9216		4096				4096		16	37,749	38
fc7	4096		4096				4096		16	16,777	17
fc8	4096		1000				1000		4	4,096	4

AlexNet

AlexNet

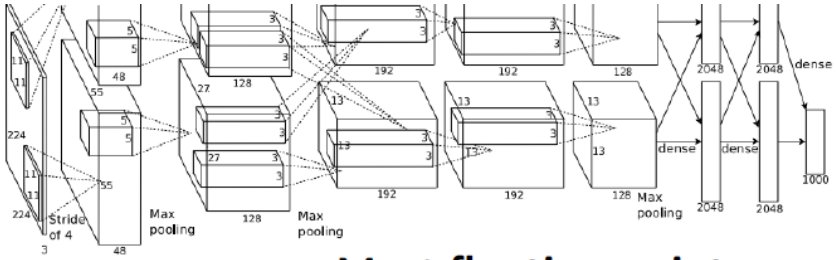
Interesting trends here!



	Input size		Layer				Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)
conv1	3	227	64	11	4	2	64	56	784	23	73
pool1	64	56		3	2	0	64	27	182	0	0
conv2	64	27	192	5	1	2	192	27	547	307	224
pool2	192	27		3	2	0	192	13	127	0	0
conv3	192	13	384	3	1	1	384	13	254	664	112
conv4	384	13	256	3	1	1	256	13	169	885	145
conv5	256	13	256	3	1	1	256	13	169	590	100
pool5	256	13		3	2	0	256	6	36	0	0
flatten	256	6					9216		36	0	0
fc6	9216		4096				4096		16	37,749	38
fc7	4096		4096				4096		16	16,777	17
fc8	4096		1000				1000		4	4,096	4

AlexNet

AlexNet

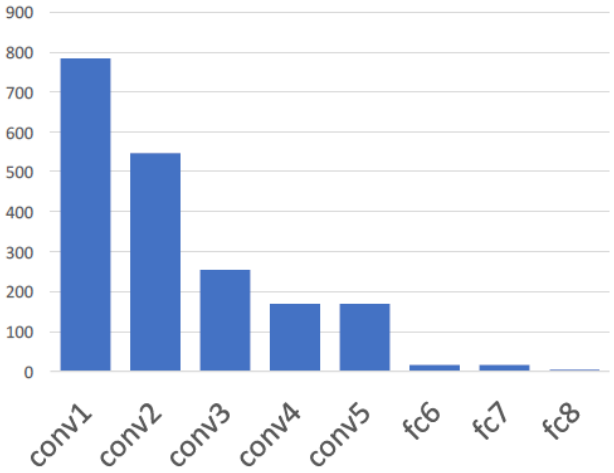


Most of the **memory usage** is in the early convolution layers

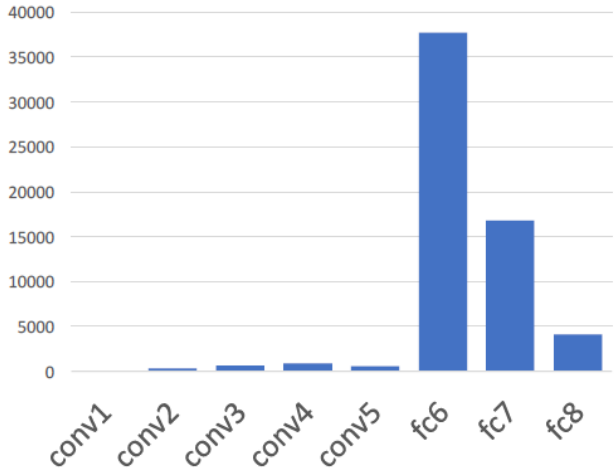
Nearly all **parameters** are in the fully-connected layers

Most **floating-point ops** occur in the convolution layers

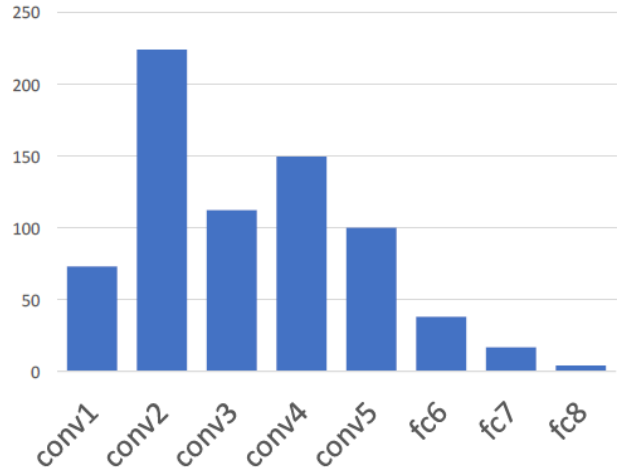
Memory (KB)



Params (K)



MFLOP



References

- Deep Learning, Ian Goodfellow, MIT Press, Ch. 9
- Aston Zhang, Dive into Deep Learning, Ch 7 & Ch. 8