Lecture 3: CNNs I - Architecture & Training

Lan Xu
SIST, ShanghaiTech
Fall, 2023

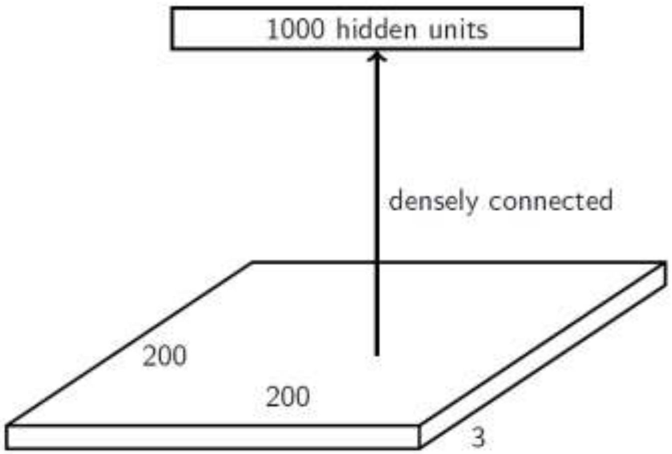
Outline

- Why Convolutional Neural Network (CNN)?
  - Motivation and overview
- What is the CNN?
  - Convolution layers & model complexity
  - Closer look at activation functions
  - Pooling layers & model complexity
  - Math properties
- Examples of CNNs

Acknowledgement: Roger Grosse@UofT & Feifei Li's cs231n notes

Motivation

- Visual recognition
  - Suppose we aim to train a network that takes a 200x200 RGB image as input



  - What is the problem with have full connections in the first layer?
    - Too many parameters! 200x200x3x1000 = 120 million
    - What happens if the object in the image shifts a little?

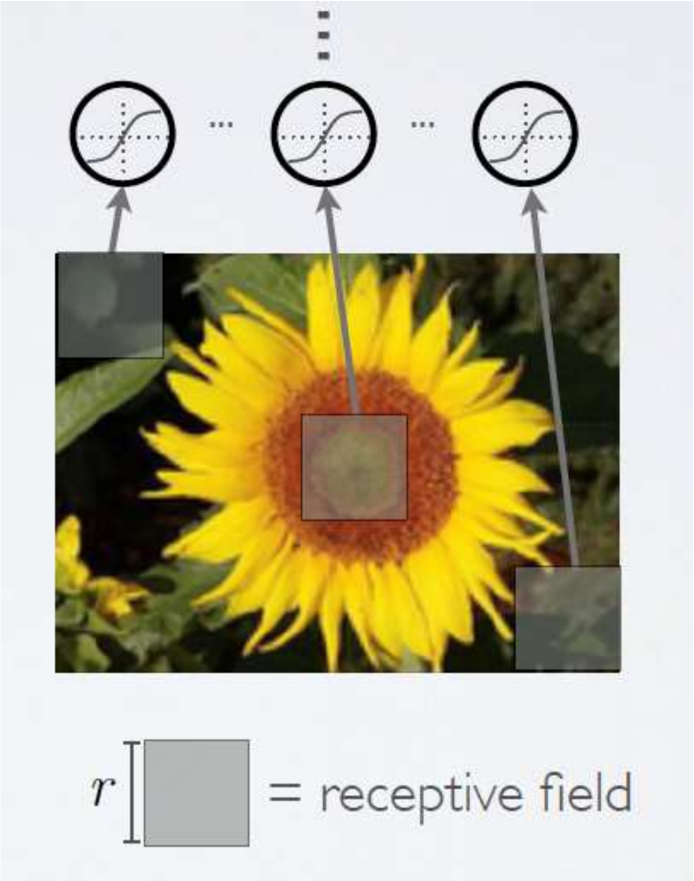
Our goal

- Visual Recognition: Design a neural network that
  - Much deal with very high-dimensional inputs
  - Can exploit the 2D topology of pixels in images
  - Can build in invariance/equivariance to certain variations we can expect
    - Translation, small deformations, illumination, etc.

- Convolution networks leverage these ideas
  - Local connectivity
  - Parameter sharing
  - Pooling/subsampling hidden units

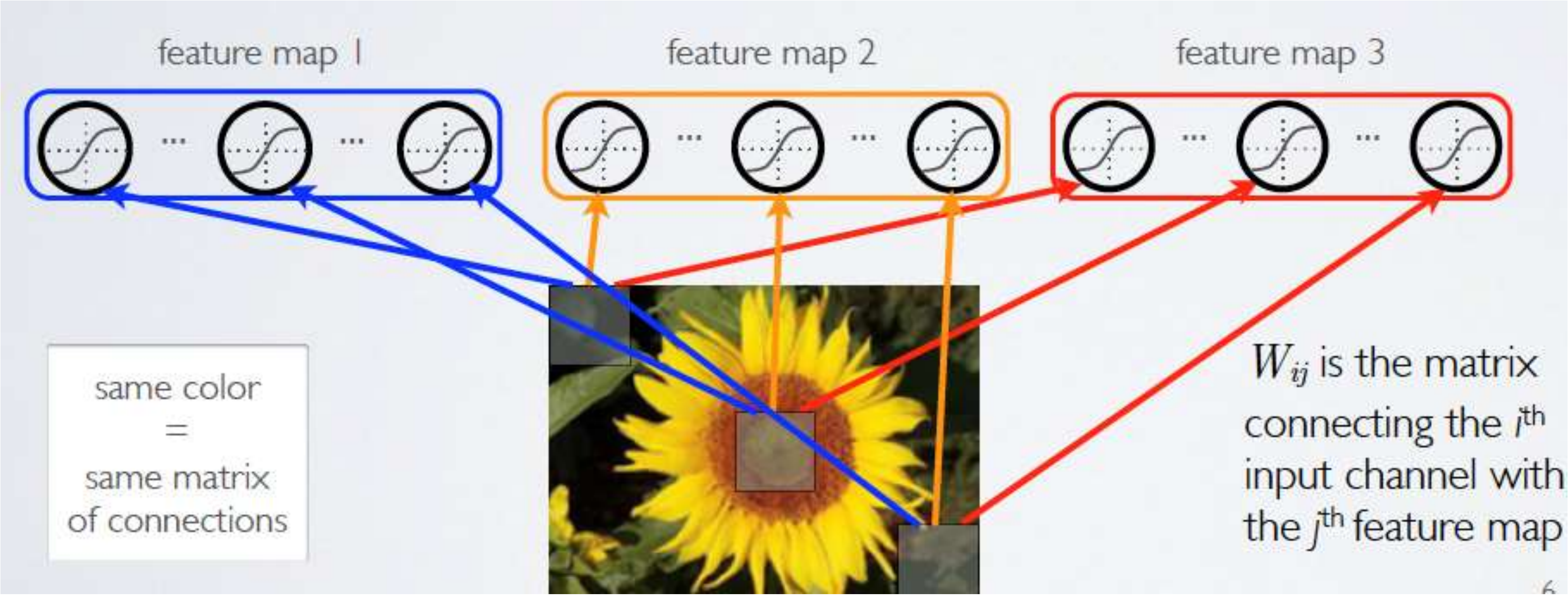
Overview of CNNs

- First idea: Use a local connectivity of hidden units
  - Each hidden unit is connected only to a subregion (patch) of the input image
  - Usually it is connected to all channels
  - Each neuron has a local receptive field



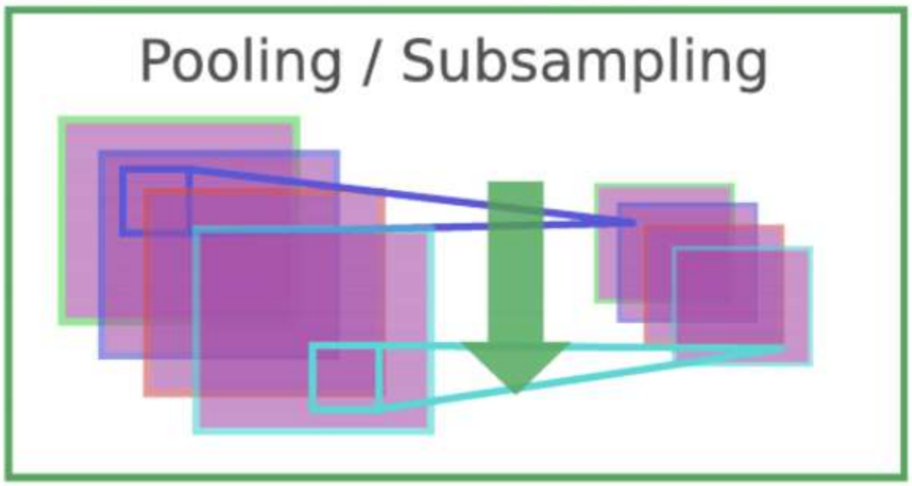
Overview of CNNs

- Second idea: share weights across certain units
  - Units organized into the same “feature map” share weight parameters
  - Hidden units within a feature map cover different positions in the image



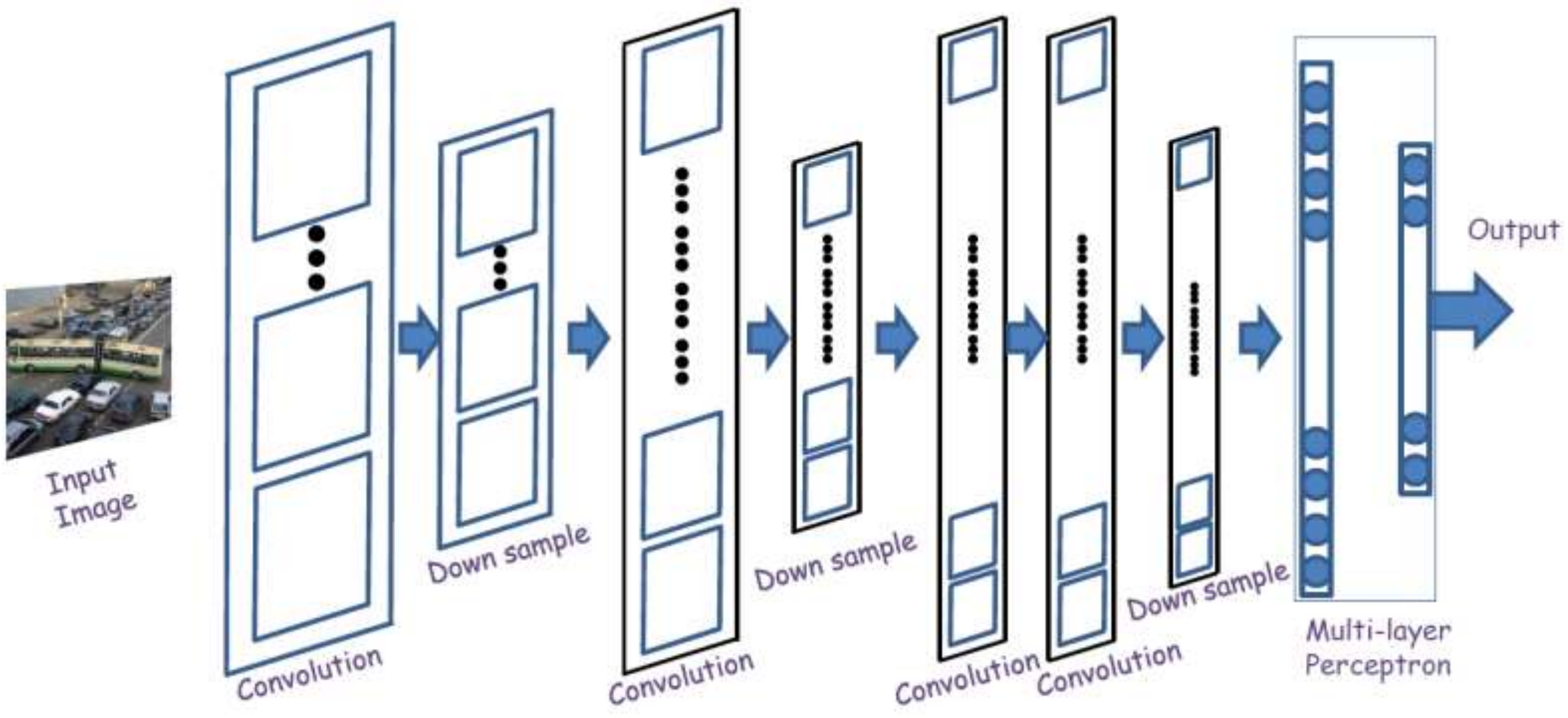
Overview of CNNs

- Third idea: pool hidden units in the same neighborhood
  - Averaging or Discarding location information in a small region
  - Robust toward small deformations in object shapes by ignoring details.



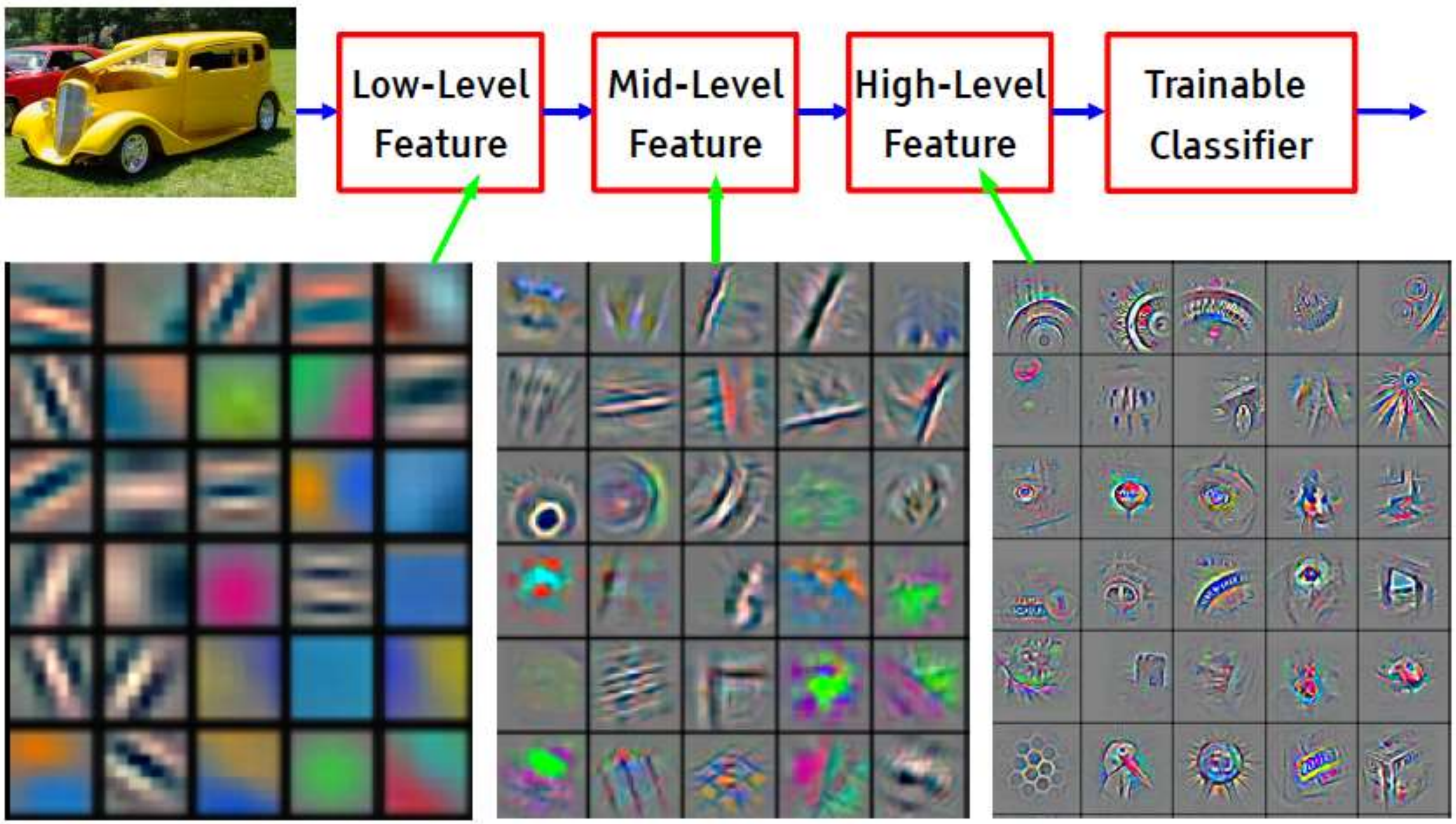
Overview of CNNs

- Fourth idea: Interleaving feature extraction and pooling operations
  - Extracting abstract, compositional features for representing semantic object classes



Overview of CNNs

- Artificial visual pathway: from images to semantic concepts (Representation learning)



Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

Outline

- Why Convolutional Neural Network (CNN)?
  - Motivation and overview
- What is the CNN?
  - Convolution layers & model complexity
  - Closer look at activation functions
  - Pooling layers & model complexity
  - Math properties
- Examples of CNNs

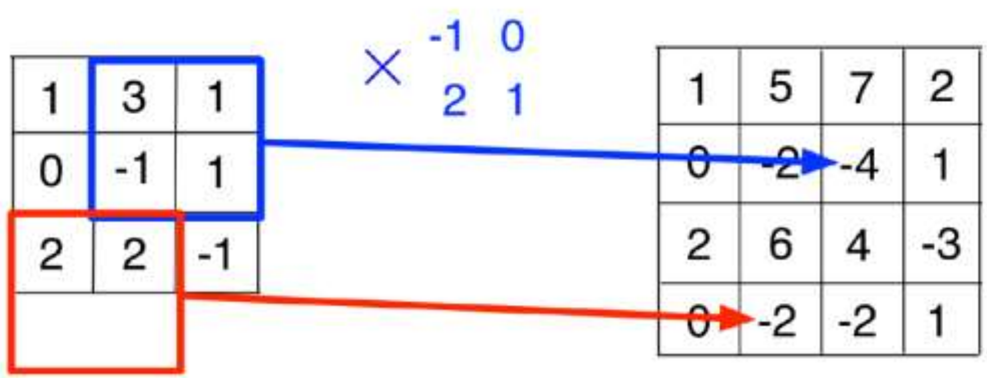
Acknowledgement: Roger Grosse@UofT & Feifei Li's cs231n notes

2D Convolution

If A and B are two 2-D arrays, then:

$$(A * B)_{ij} = \sum_s \sum_t A_{st} B_{i-s,j-t}.$$

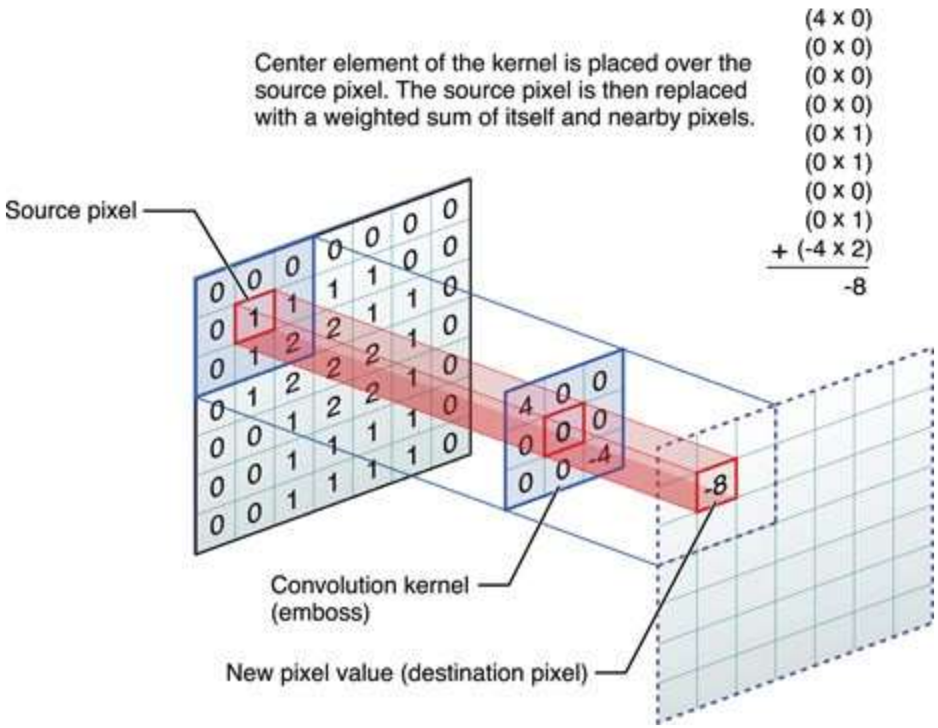
$$\begin{array}{|c|c|c|}\hline 1 & 3 & 1 \\ \hline 0 & -1 & 1 \\ \hline 2 & 2 & -1 \\ \hline \end{array} * \begin{array}{|c|c|}\hline 1 & 2 \\ \hline 0 & -1 \\ \hline \end{array}$$

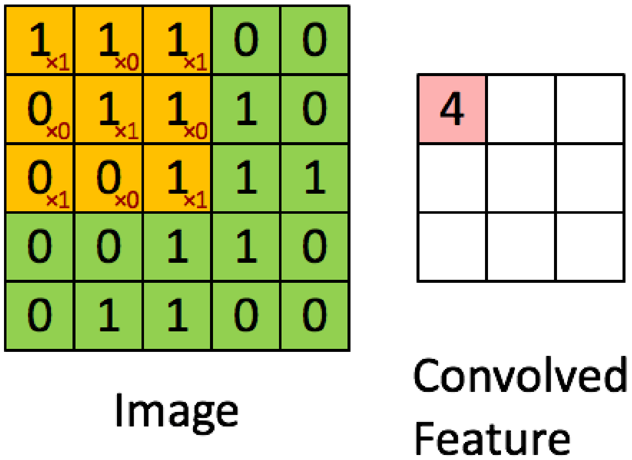


2D Convolution

If A and B are two 2-D arrays, then:

$$(A * B)_{ij} = \sum_s \sum_t A_{st} B_{i-s,j-t}.$$

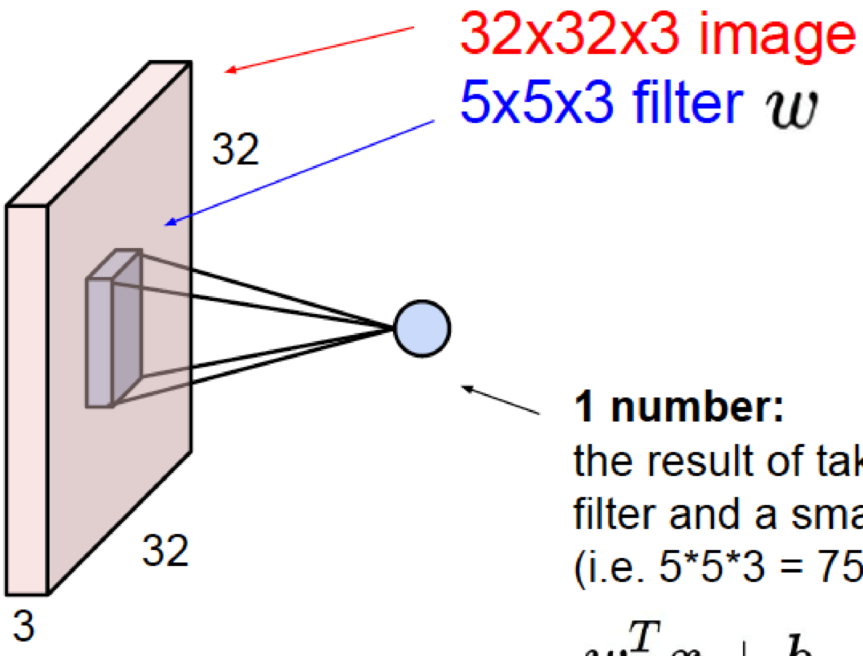




Picture Courtesy: developer.apple.com

Convolution Layers

- Formal definition

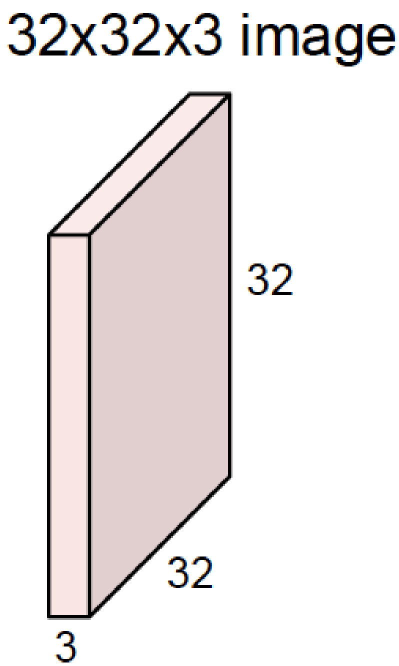


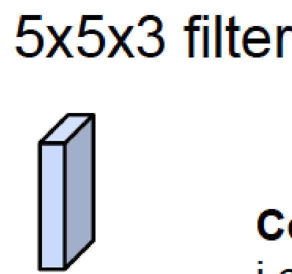
1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)

$$w^T x + b$$

Convolution Layers

- Define a neuron corresponding to a 5x5 filter

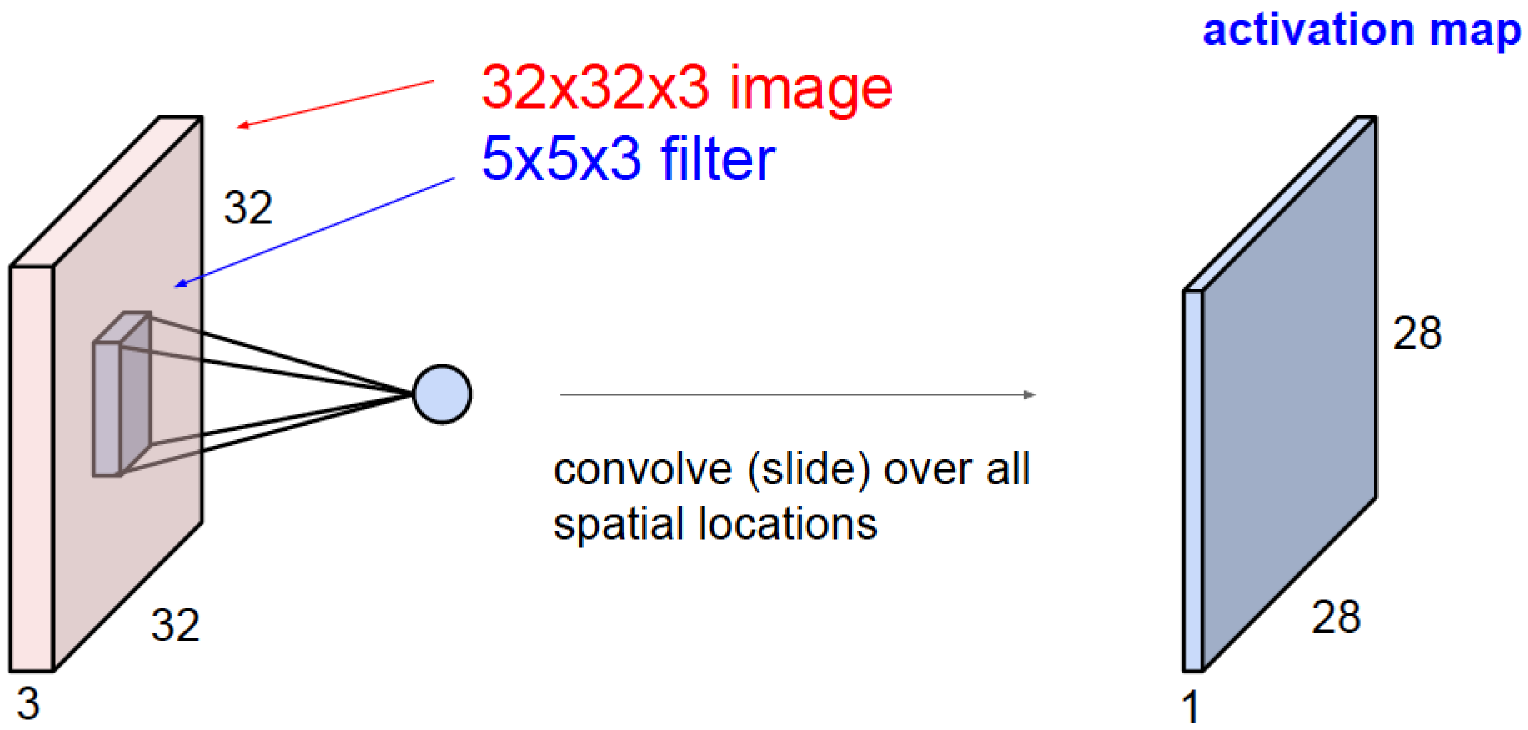




Convolve the filter with the image
i.e. “slide over the image spatially,
computing dot products”

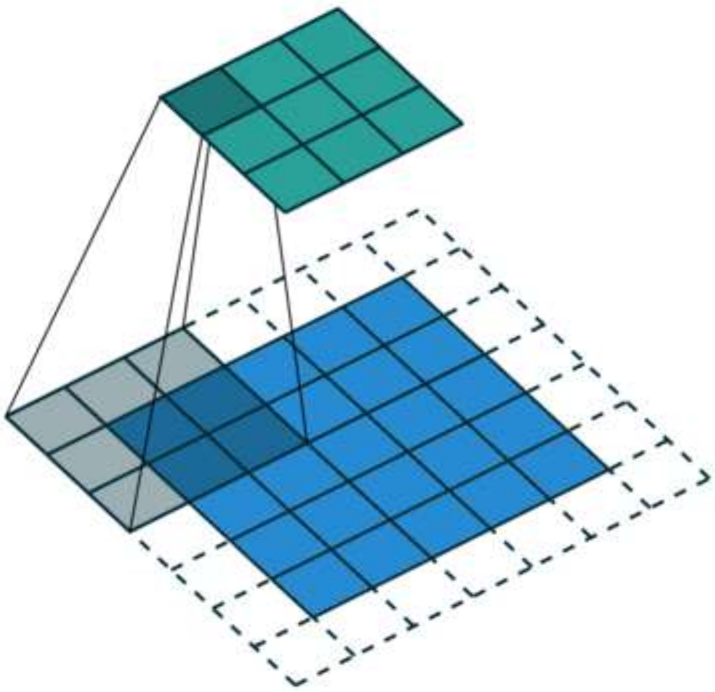
Convolution Layers

- Convolution operation
  - Parameter sharing
  - Spatial information



Convolution Layers

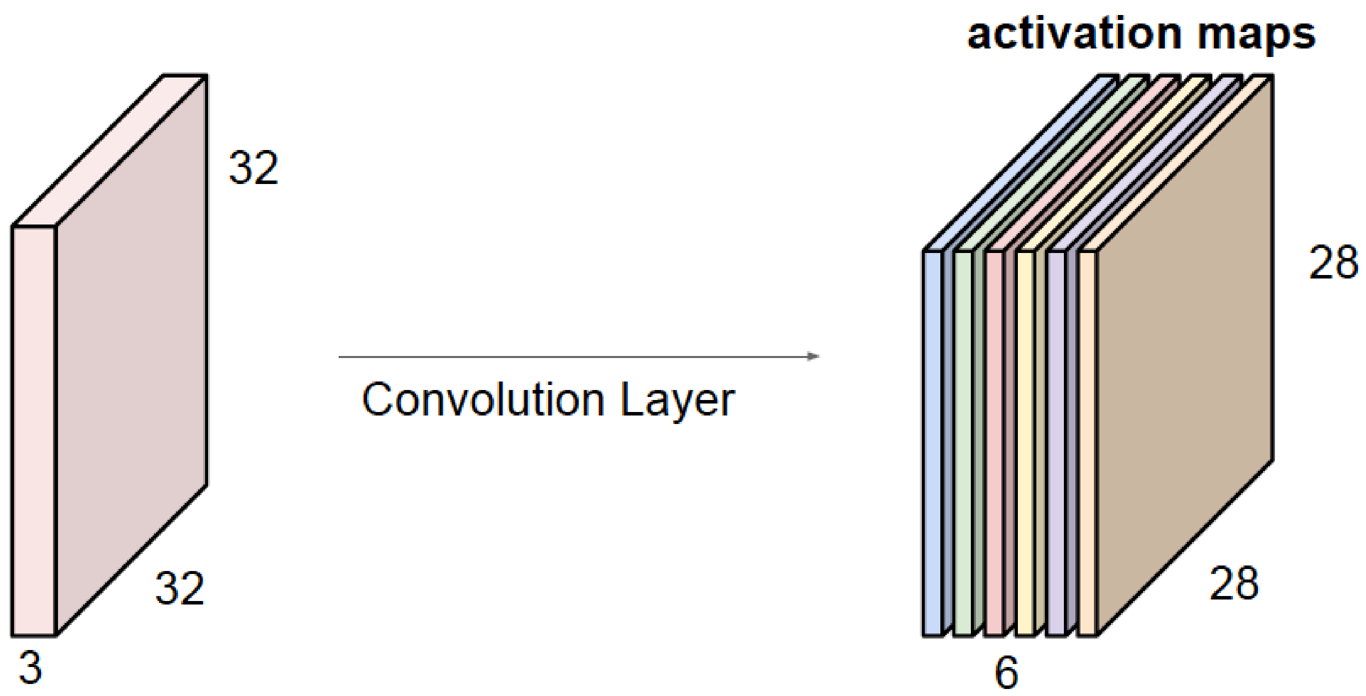
- Convolution operation
  - Parameter sharing
  - Spatial information



Convolution Layers

- Multiple kernels/filters

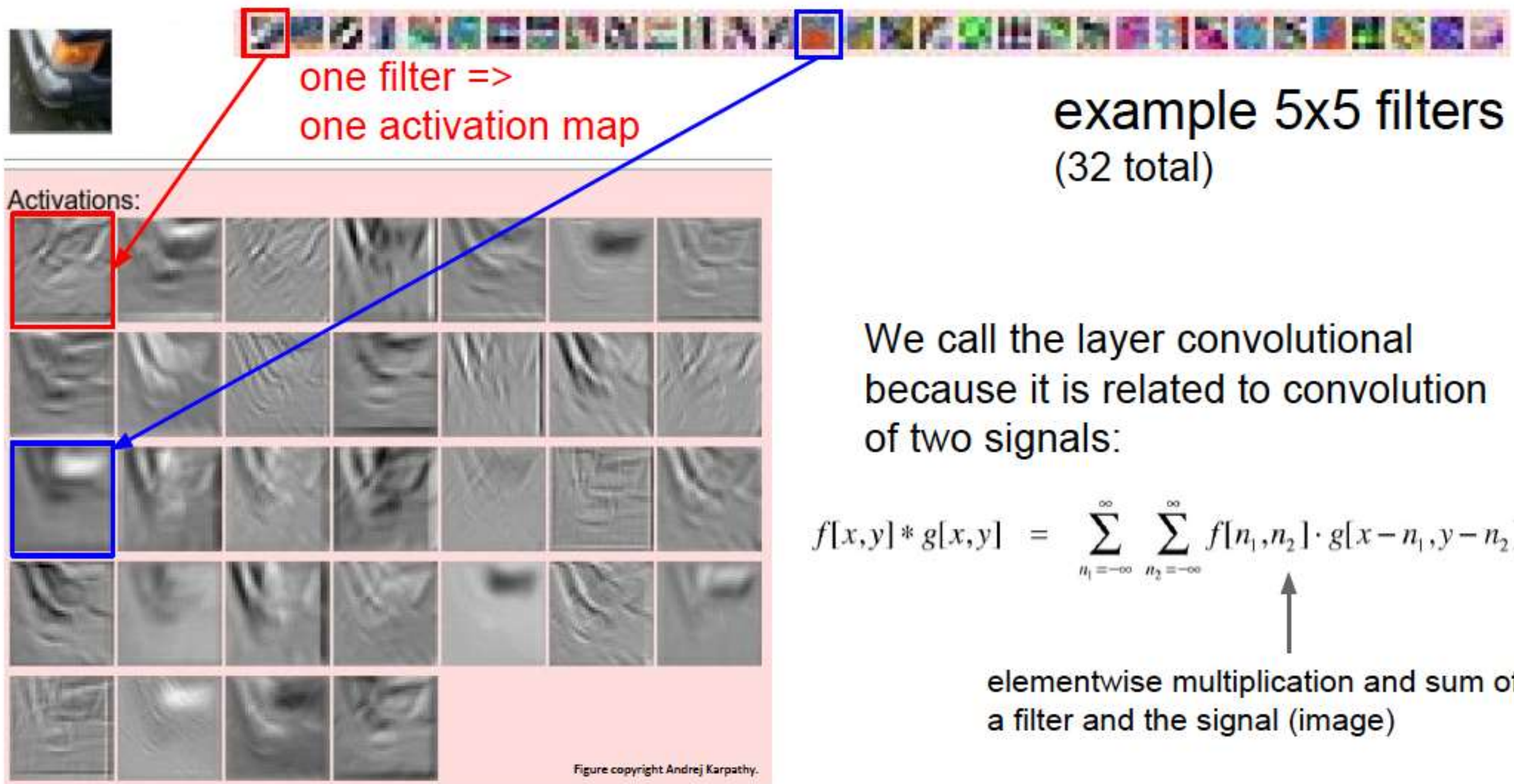
For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



We stack these up to get a "new image" of size 28x28x6!

Convolution Layers

- Visualizing the filters and their outputs



example 5x5 filters
(32 total)

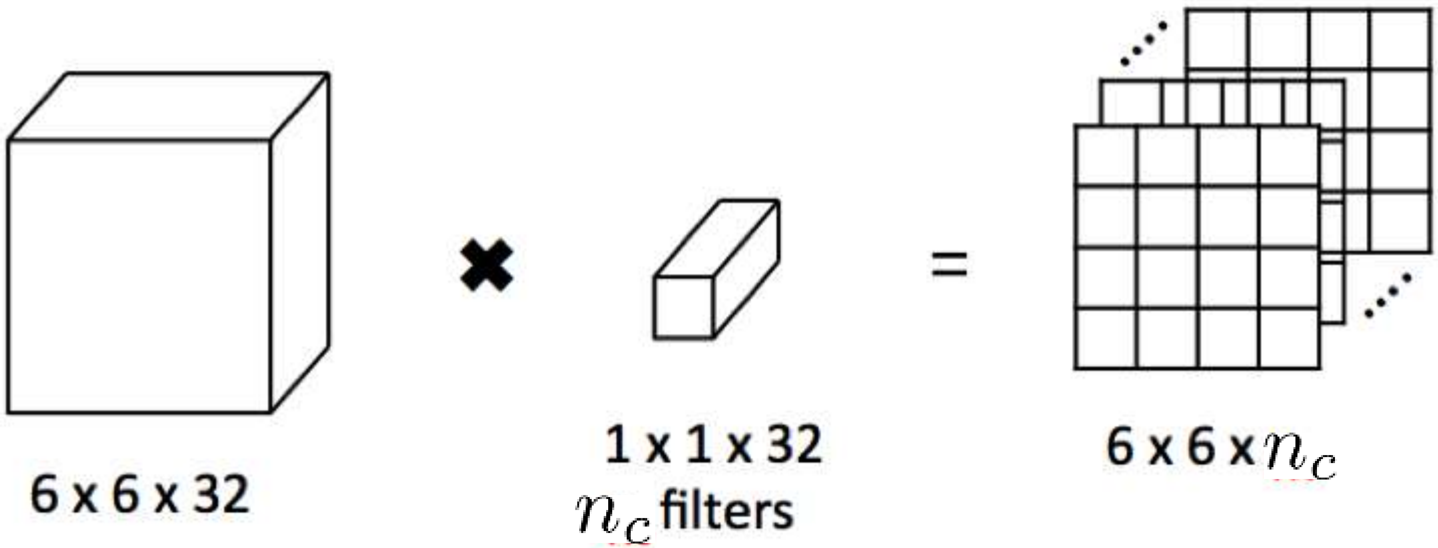
We call the layer convolutional because it is related to convolution of two signals:

$$f[x,y]*g[x,y] = \sum_{n_1=-\infty}^{\infty}\sum_{n_2=-\infty}^{\infty} f[n_1,n_2]\cdot g[x-n_1,y-n_2]$$

elementwise multiplication and sum of a filter and the signal (image)

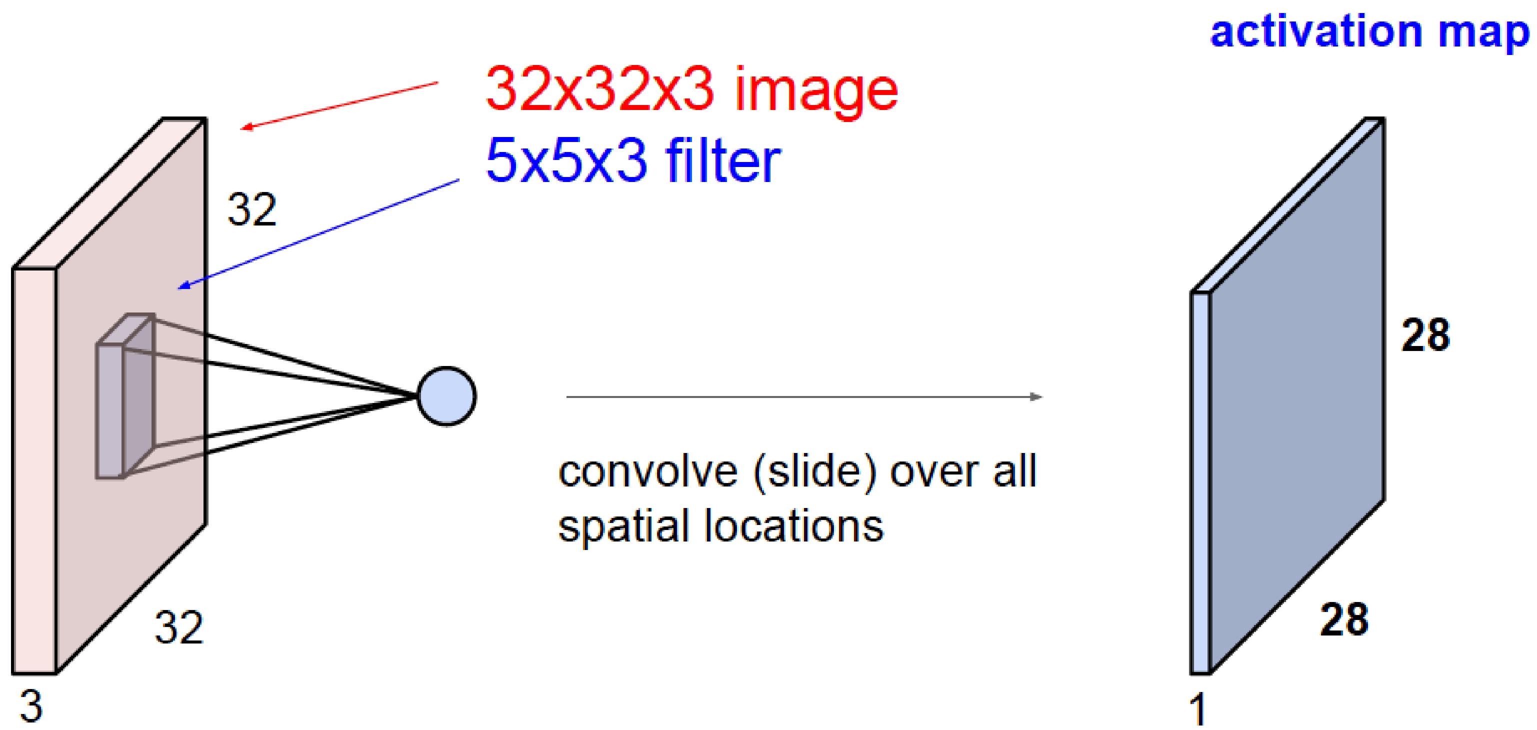
Special Convolutions

- 1x1 convolutions
  - Used in Network-in-network, GoogleNet
  - Reduce or increase dimensionality
  - Can be considered as ‘feature pooling”



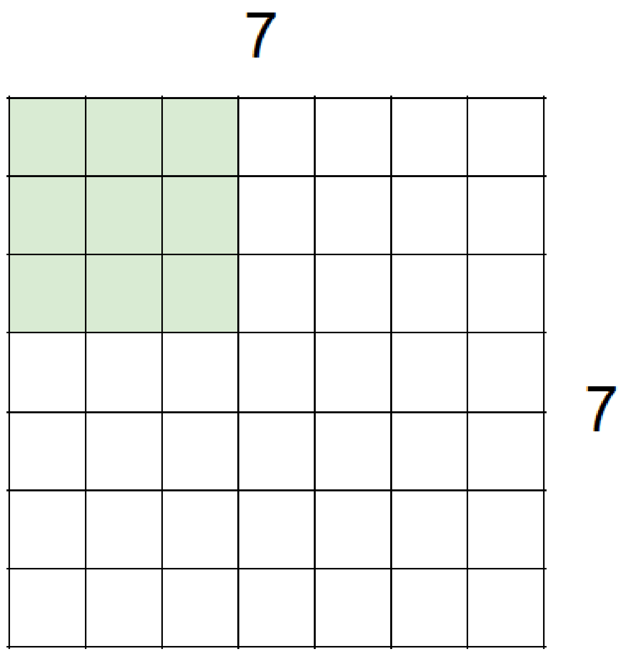
Complexity of Convolution Layers

- Sizes of activation maps and number of parameters



Complexity of Convolution Layers

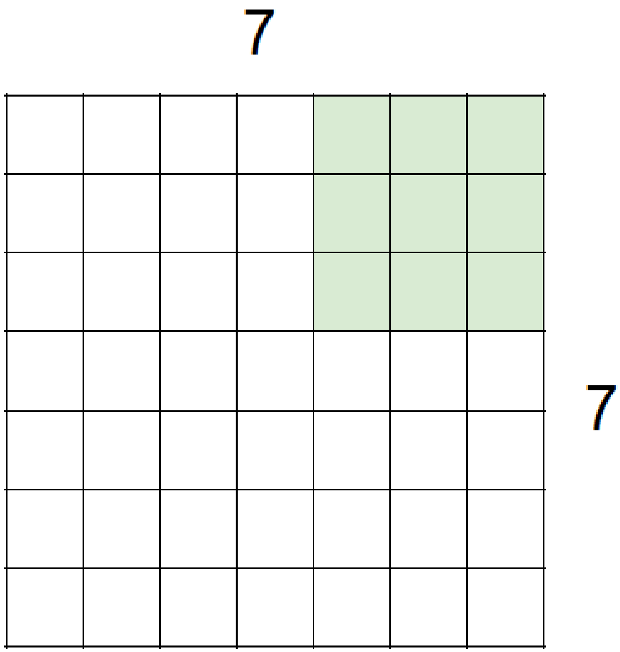
- Size of activation maps



7x7 input (spatially)
assume 3x3 filter

Complexity of Convolution Layers

- Size of activation maps

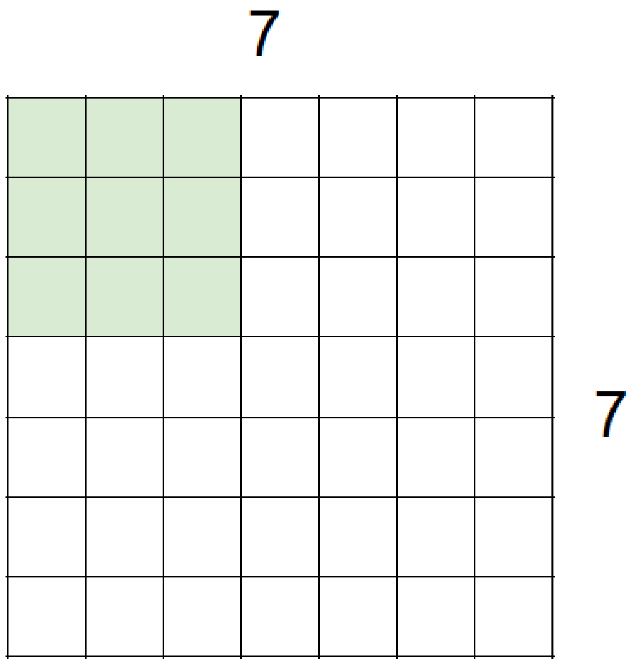


7x7 input (spatially)
assume 3x3 filter

=> 5x5 output

Complexity of Convolution Layers

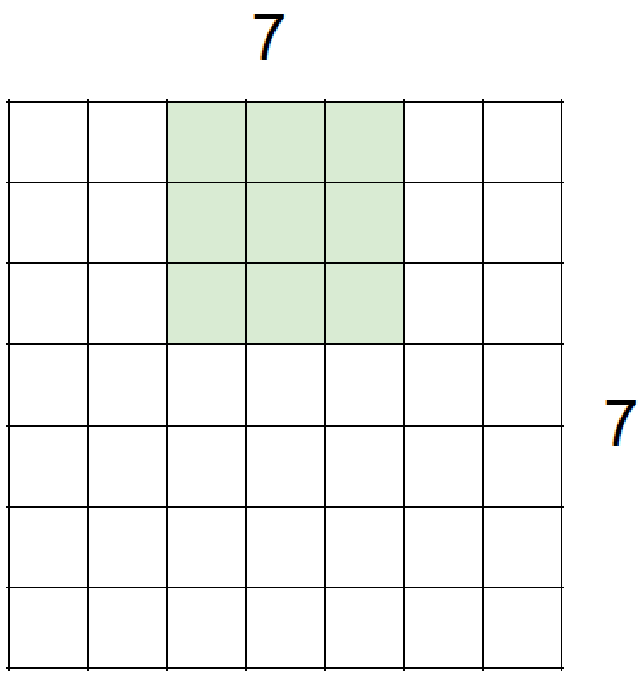
- Case: Stride > 1



7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

Complexity of Convolution Layers

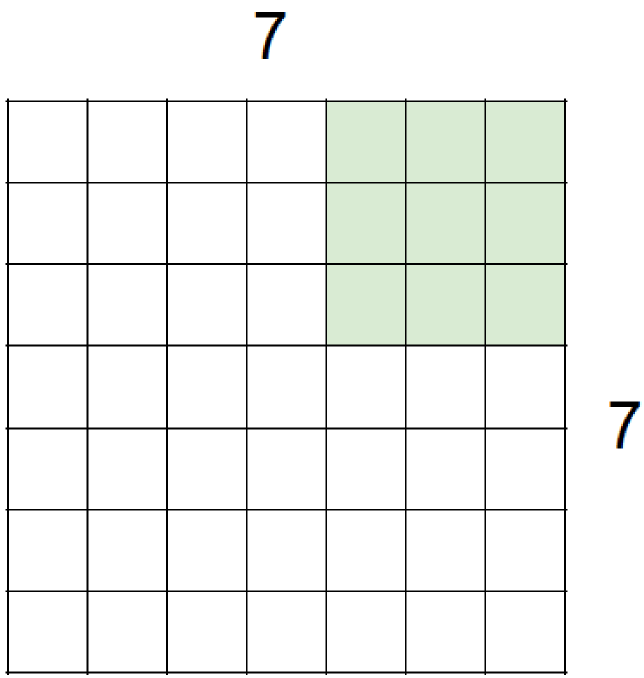
- Case: Stride > 1



7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

Complexity of Convolution Layers

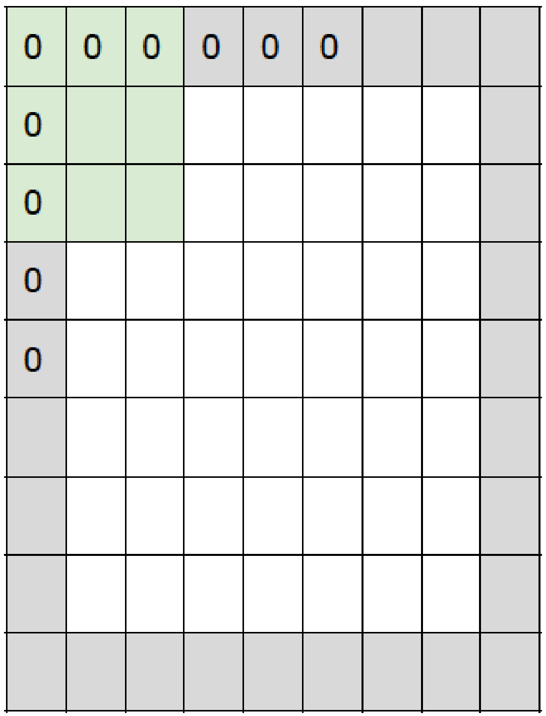
- Case: Stride > 1



7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**
=> 3x3 output!

Complexity of Convolution Layers

- Zero padding to handle non-integer cases or control the output sizes



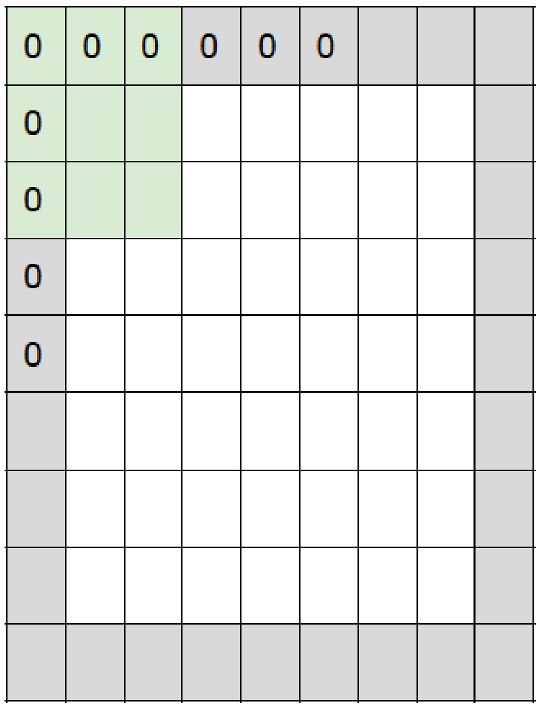
e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7x7 output!

(recall:)
(N - F) / stride + 1

Complexity of Convolution Layers

- Zero padding to handle non-integer cases or control the output sizes

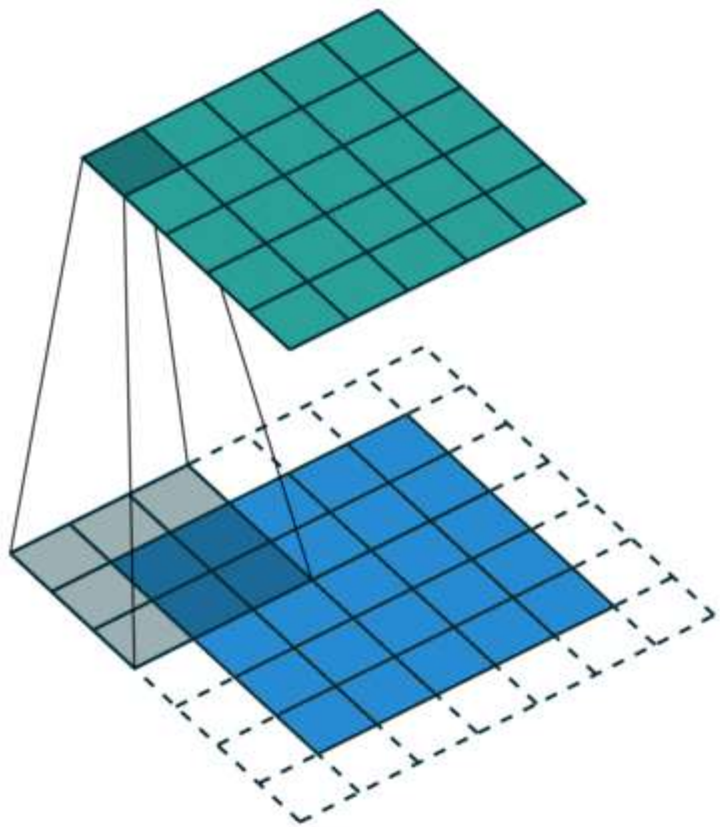


e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7x7 output!
in general, common to see CONV layers with stride 1, filters of size FxF, and zero-padding with (F-1)/2. (will preserve size spatially)
e.g. F = 3 => zero pad with 1
F = 5 => zero pad with 2
F = 7 => zero pad with 3

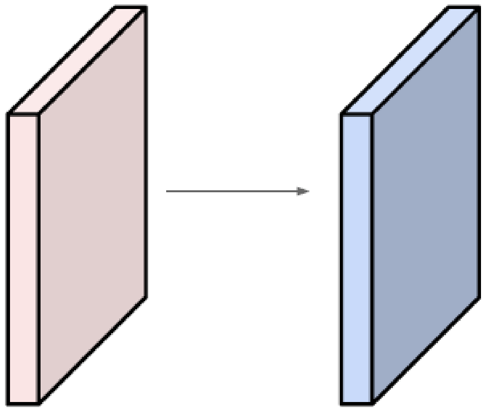
Complexity of Convolution Layers

- Zero padding to handle non-integer cases or control the output sizes



Complexity of Convolution Layers

Examples time:

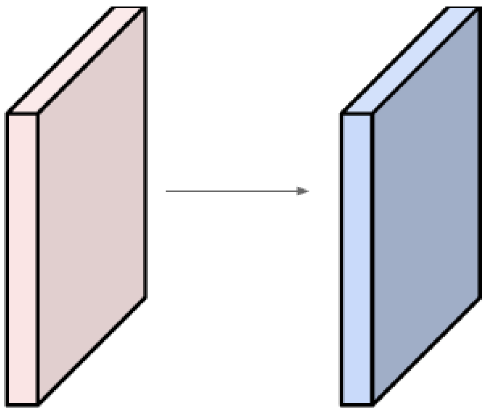


Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Output volume size:
(32+2*2-5)/1+1 = 32 spatially, so
32x32x10

Complexity of Convolution Layers

Examples time:



Input volume: **32x32x3**

10 5x5 filters with stride 1, pad 2

Number of parameters in this layer?

each filter has 5*5*3 + 1 = 76 params (+1 for bias)

=> 76*10 = **760**

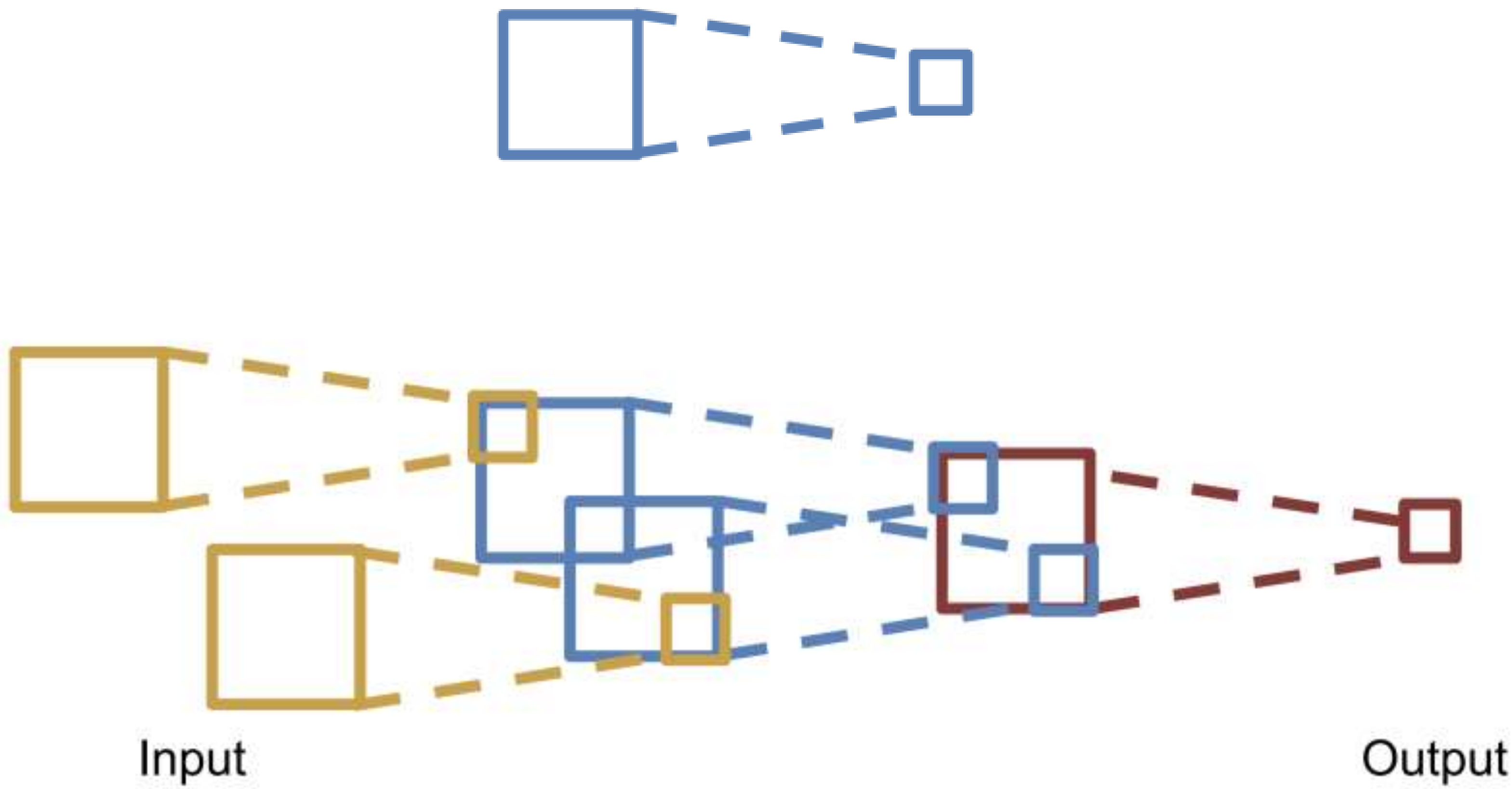
Complexity of Convolution Layers

- Summary

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.

Receptive Fields

- For convolution with kernel size K, each element in the output depends on a K x K receptive field in the input



Outline

- Why Convolutional Neural Network (CNN)?
  - Motivation and overview
- What is the CNN?
  - Convolution layers & model complexity
  - Closer look at activation functions
  - Pooling layers & model complexity
  - Math properties
- Examples of CNNs

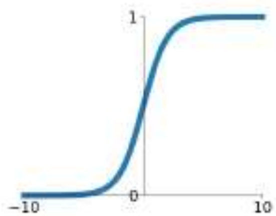
Acknowledgement: Roger Grosse@UofT & Feifei Li's cs231n notes

Review: Activation Function

- Zoo of Activation functions

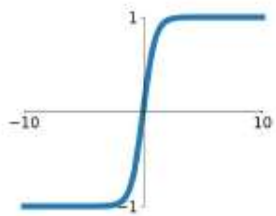
Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$



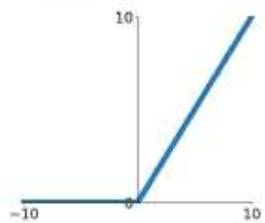
tanh

$\tanh(x)$



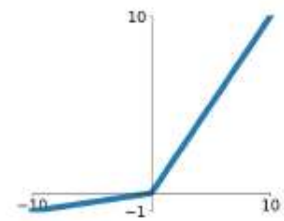
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

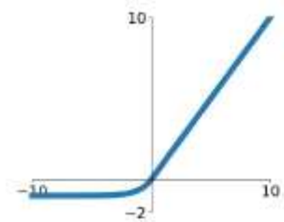


Maxout

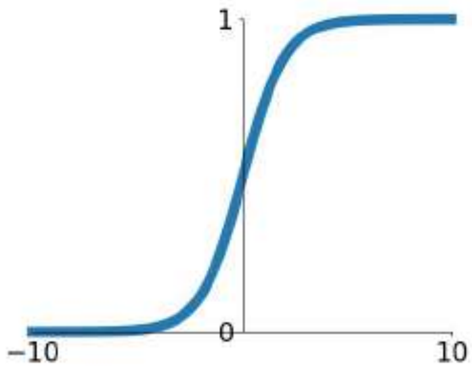
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Sigmoid function



Sigmoid

$$\sigma(x) = 1/(1 + e^{-x})$$

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

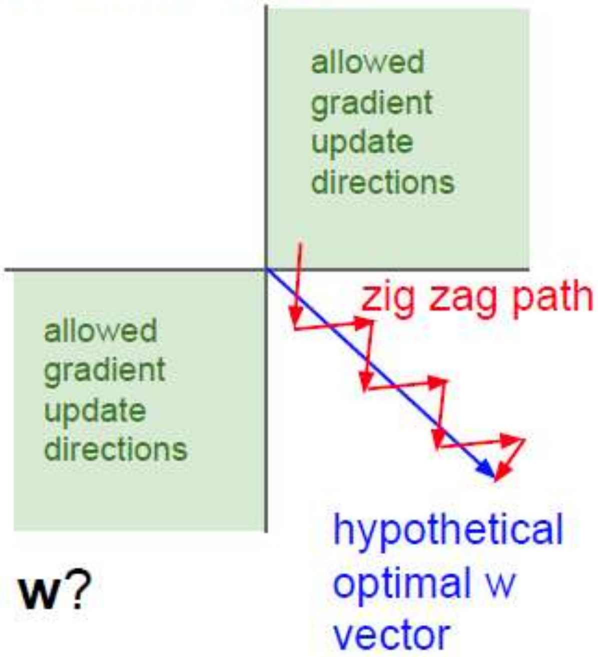
3 problems:

1. Saturated neurons "kill" the gradients
2. Sigmoid outputs are not zero-centered
3. exp() is a bit compute expensive

Sigmoid function

Consider what happens when the input to a neuron is always positive...

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

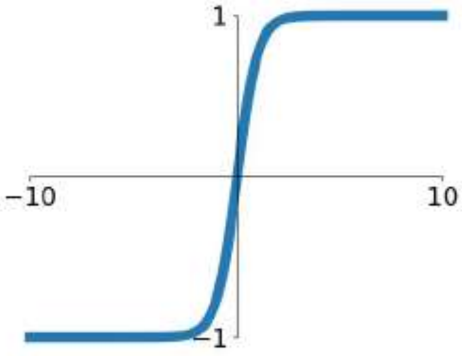


What can we say about the gradients on **w**?

Always all positive or all negative :(

(this is also why you want zero-mean data!)

Tanh function



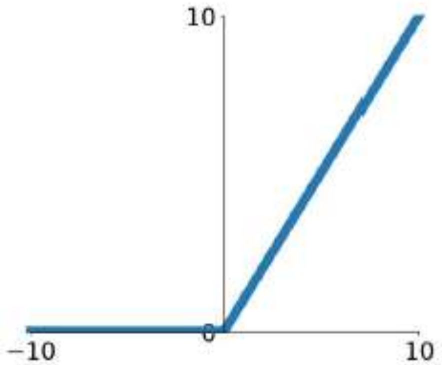
tanh(x)

- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

[LeCun et al., 1991]

- Recurrent neural networks: LSTM, GRU

Rectified Linear Unit



ReLU
(Rectified Linear Unit)

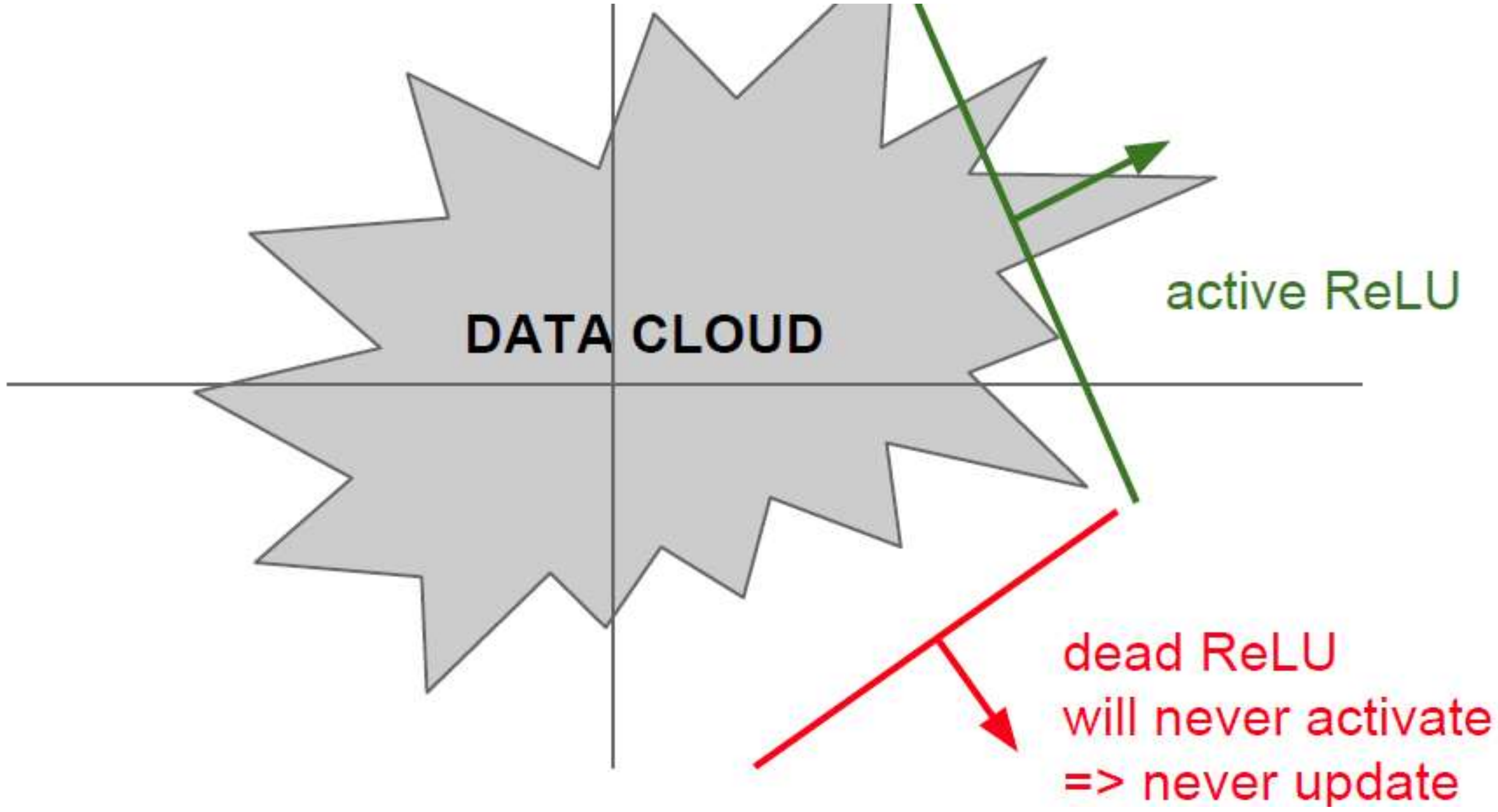
- Computes **f(x) = max(0,x)**

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Actually more biologically plausible than sigmoid

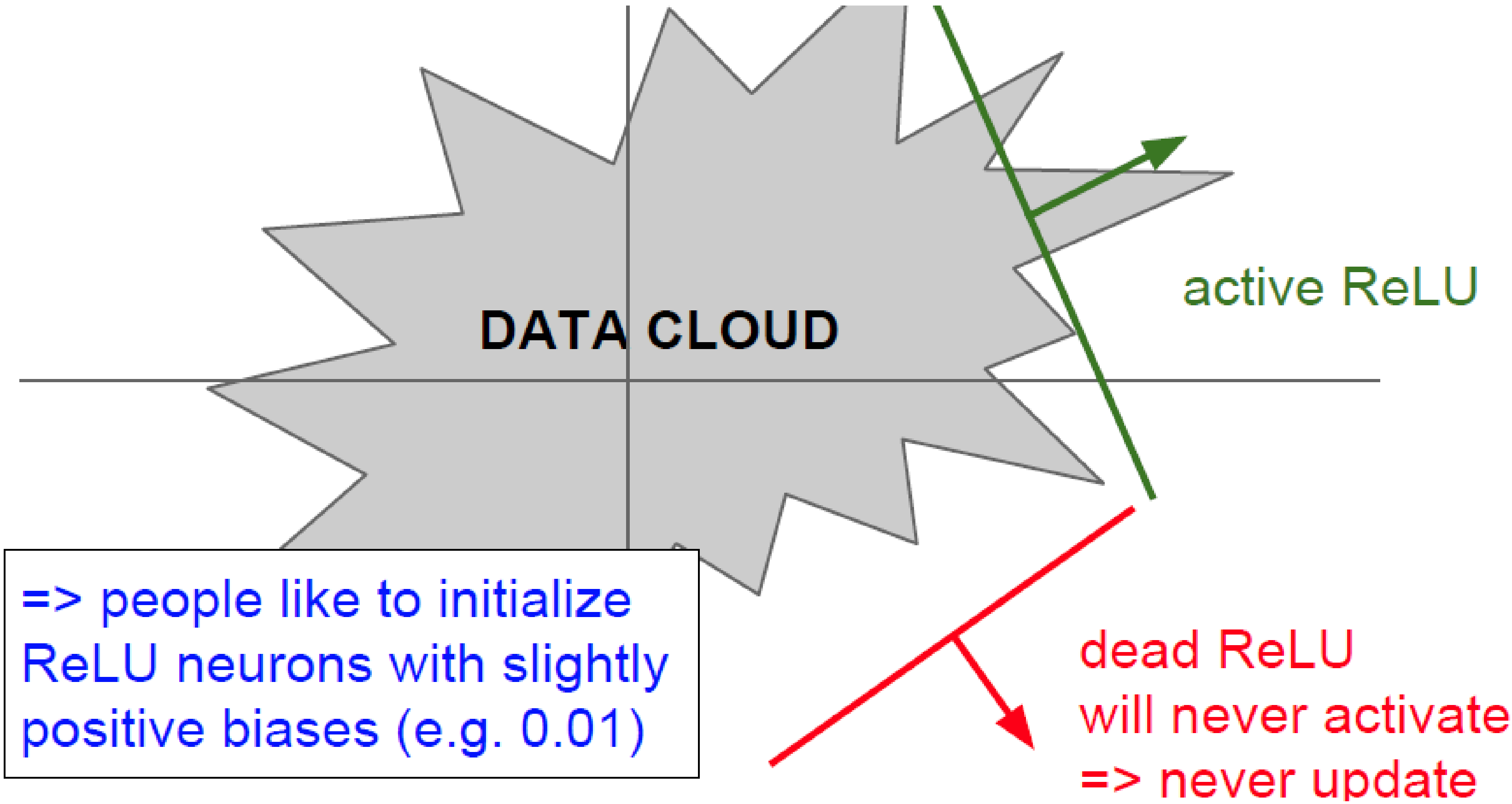
- Not zero-centered output
- An annoyance:

hint: what is the gradient when x < 0?

Rectified Linear Unit



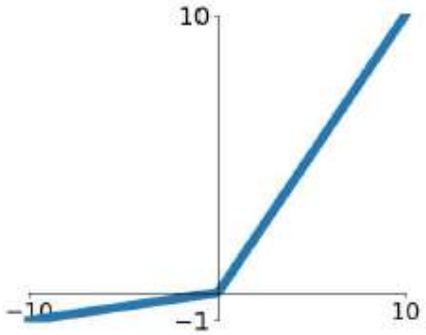
Rectified Linear Unit



=> people like to initialize ReLU neurons with slightly positive biases (e.g. 0.01)

Leaky ReLU

[Mass et al., 2013]
[He et al., 2015]



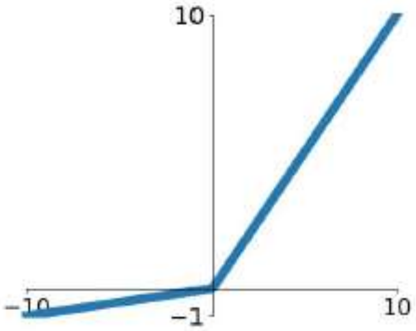
- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not “die”.**

Leaky ReLU

$$f(x) = \max(0.01x, x)$$

Leaky ReLU

[Mass et al., 2013]
[He et al., 2015]



- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not “die”.**

Leaky ReLU

$$f(x) = \max(0.01x, x)$$

Parametric Rectifier (PReLU)

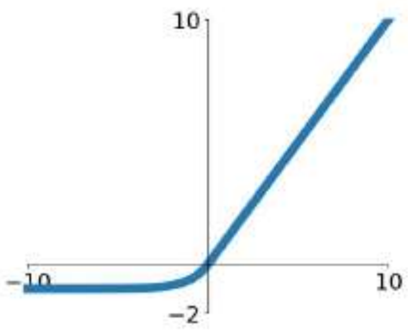
$$f(x) = \max(\alpha x, x)$$

backprop into \alpha
(parameter)

Exponential Linear Units (ELU)

[Clevert et al., 2015]

Exponential Linear Units (ELU)



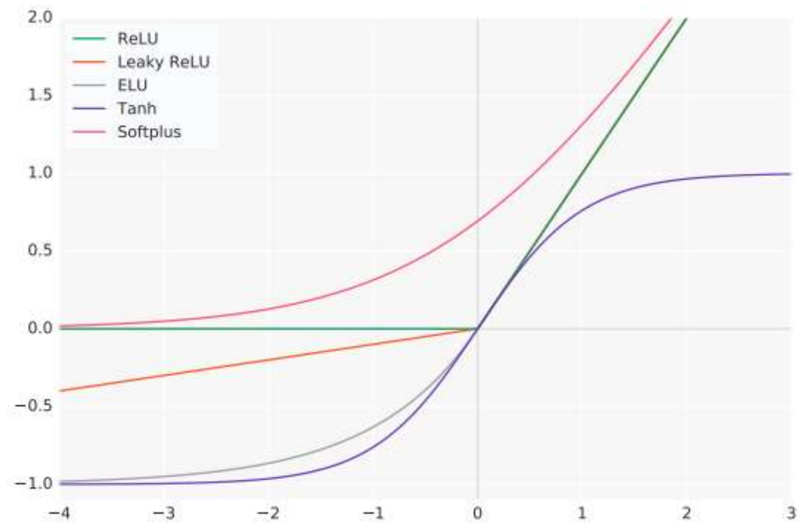
- All benefits of ReLU
- Closer to zero mean outputs
- Negative saturation regime compared with Leaky ReLU adds some robustness to noise

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \le 0 \end{cases}$$

- Computation requires exp()

Summary: Activation function

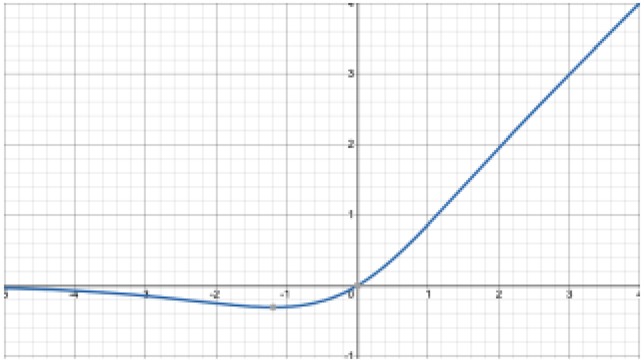
- For internal layers in CNNs
  - Use ReLU. Be careful with your learning rates
  - Try out Leaky ReLU / Maxout / ELU
  - Try out tanh but don't expect much
  - Don't use sigmoid



- For output layers
  - Task dependent
  - Related to your loss function

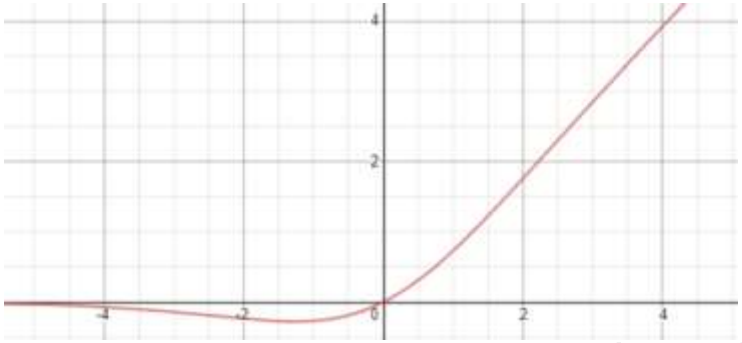
Summary: Activation function

- Recent progresses
  - Mish $f(x) = x \cdot \tanh(\varsigma(x))$, $\varsigma(x) = \ln(1+e\hat{\ }x)$,



  - Swish $f(x) = x * (1 + \exp(-x))^{-1}$

https://arxiv.org/abs/1908.08681



https://arxiv.org/abs/1710.05941

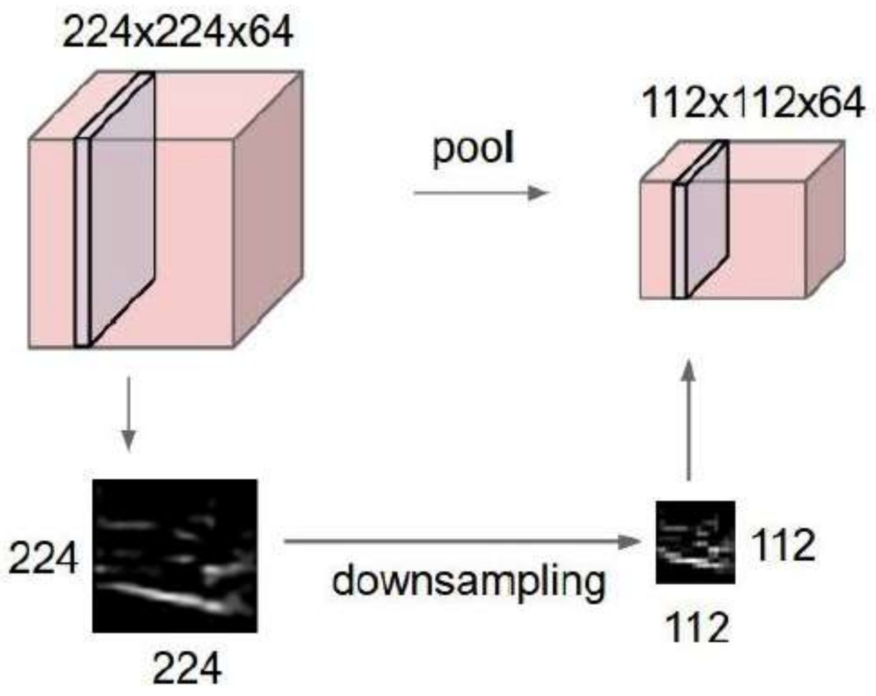
Outline

- Why Convolutional Neural Network (CNN)?
  - Motivation and overview
- What is the CNN?
  - Convolution layers & model complexity
  - Closer look at activation functions
  - Pooling layers & model complexity
  - Math properties
- Examples of CNNs

Acknowledgement: Roger Grosse@UofT & Feifei Li's cs231n notes

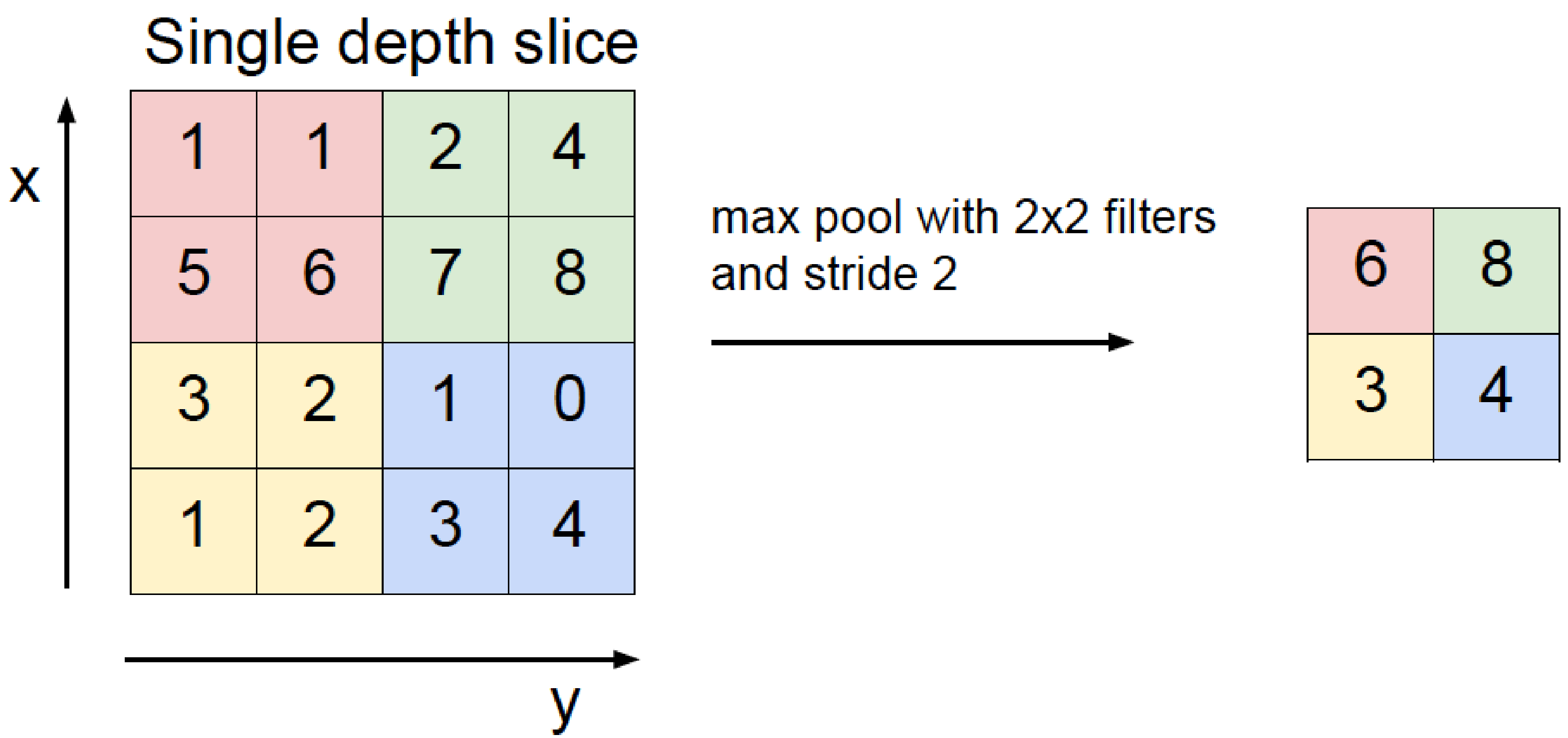
Pooling Layers

- Reducing the spatial size of the feature maps
  - Smaller representations
  - On each activation map independently
  - Low resolution means fewer details



Pooling Layers

- Example: max pooling
- Spatial invariance; no learnable parameters!



Complexity of Pooling Layers

- Summary

  - Accepts a volume of size $W_1 \times H_1 \times D_1$
  - Requires three hyperparameters:
    - their spatial extent F,
    - the stride S,
  - Produces a volume of size $W_2 \times H_2 \times D_2$ where:
    - $W_2 = (W_1 - F)/S + 1$
    - $H_2 = (H_1 - F)/S + 1$
    - $D_2 = D_1$
  - Introduces zero parameters since it computes a fixed function of the input
  - Note that it is not common to use zero-padding for Pooling layers

Outline

- Why Convolutional Neural Network (CNN)?
  - Motivation and overview
- What is the CNN?
  - Convolution layers & model complexity
  - Closer look at activation functions
  - Pooling layers & model complexity
  - Math properties
- Examples of CNNs

Acknowledgement: Roger Grosse@UofT & Feifei Li's cs231n notes

Math Properties of CNNs

- What representations a CNN can capture in general?
- Consider a representation ϕ as an abstract function

$$\phi : \mathbf{x} \rightarrow \phi(\mathbf{x}) \in \mathbb{R}^d$$

- We want to look at how the representation changes upon transformations of input image.
  - Transformations represent the potential variations in the natural images
  - Translation, scale change, rotation, local deformation etc.

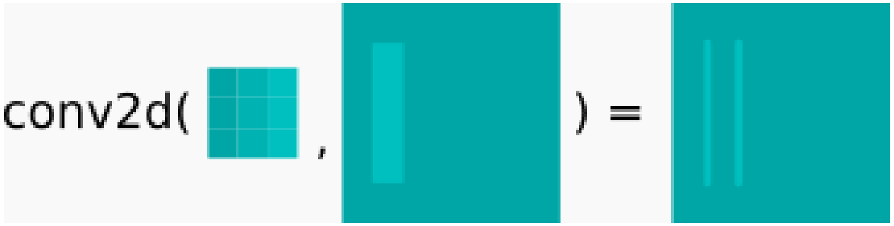
Math Properties of CNNs

- Two key properties of representations
  - Equivariance

A representation ϕ is equivariant with a transformation g if the transformation can be transferred to the representation output.

$$\exists \text{ a map } M_g : \mathbb{R}^d \to \mathbb{R}^d \text{ such that:}$$
$$\forall \mathbf{x} \in \mathcal{X} : \phi(g\mathbf{x}) \approx M_g \phi(\mathbf{x})$$

  - Example: convolution w.r.t. translation



Math Properties of CNNs

- Two key properties of representations
  - Invariance

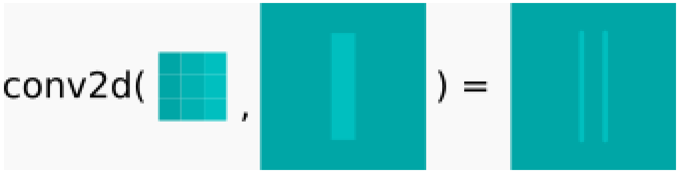
    A representation ϕ is invariant with a transformation g if the transformation has no effect on the representation output.

$$\forall \mathbf{x} \in \mathcal{X} : \phi(g\mathbf{x}) \approx \phi(\mathbf{x})$$

  - Example: convolution+pooling+FC w.r.t. translation

Math Properties of CNNs

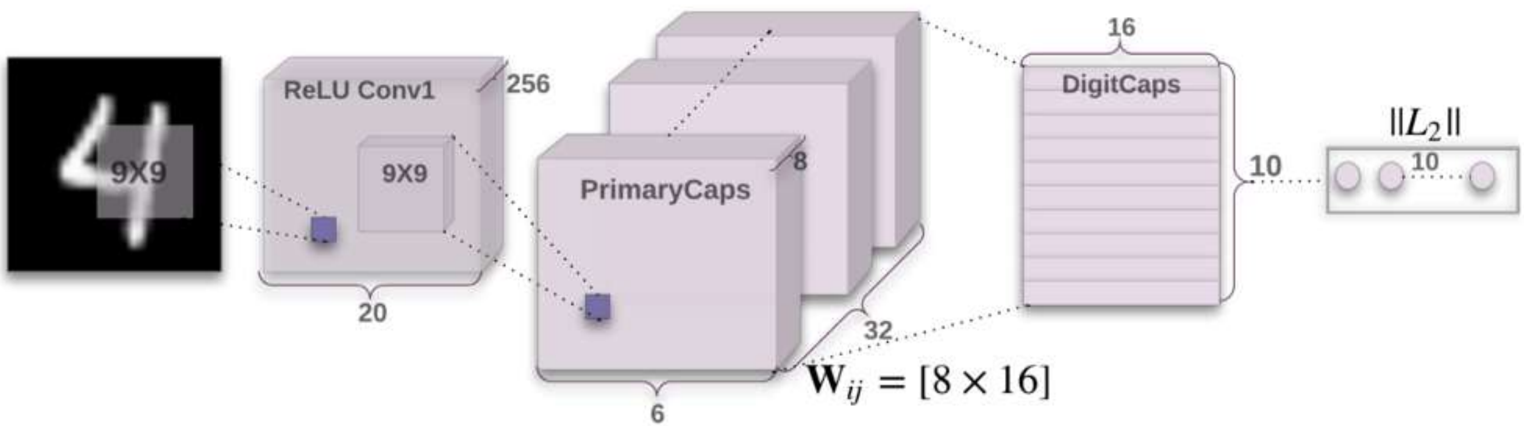
- Recent results on convolution layers
  - Convolutions are equivariant to translation
  - Convolutions are not equivariant to other isometries of the sampling lattice, e.g., rotation



  - What if a CNN learns rotated copies of the same filter?
    - The stack of feature maps is equivariant to rotation.

Math Properties of CNNs

- Recent results on convolution layers
  - Ordinary CNNs can be generalized to Group Equivariant Networks (Cohen and Welling ICML'16, Kondor and Trivedi ICML'18)
    - Redefining the convolution and pooling operations
    - Equivariant to more general transformation from some group *G*
  - Replacing pooling by other network designs
    - Capsule network (Sabour et al, 2017) https://arxiv.org/abs/1710.09829



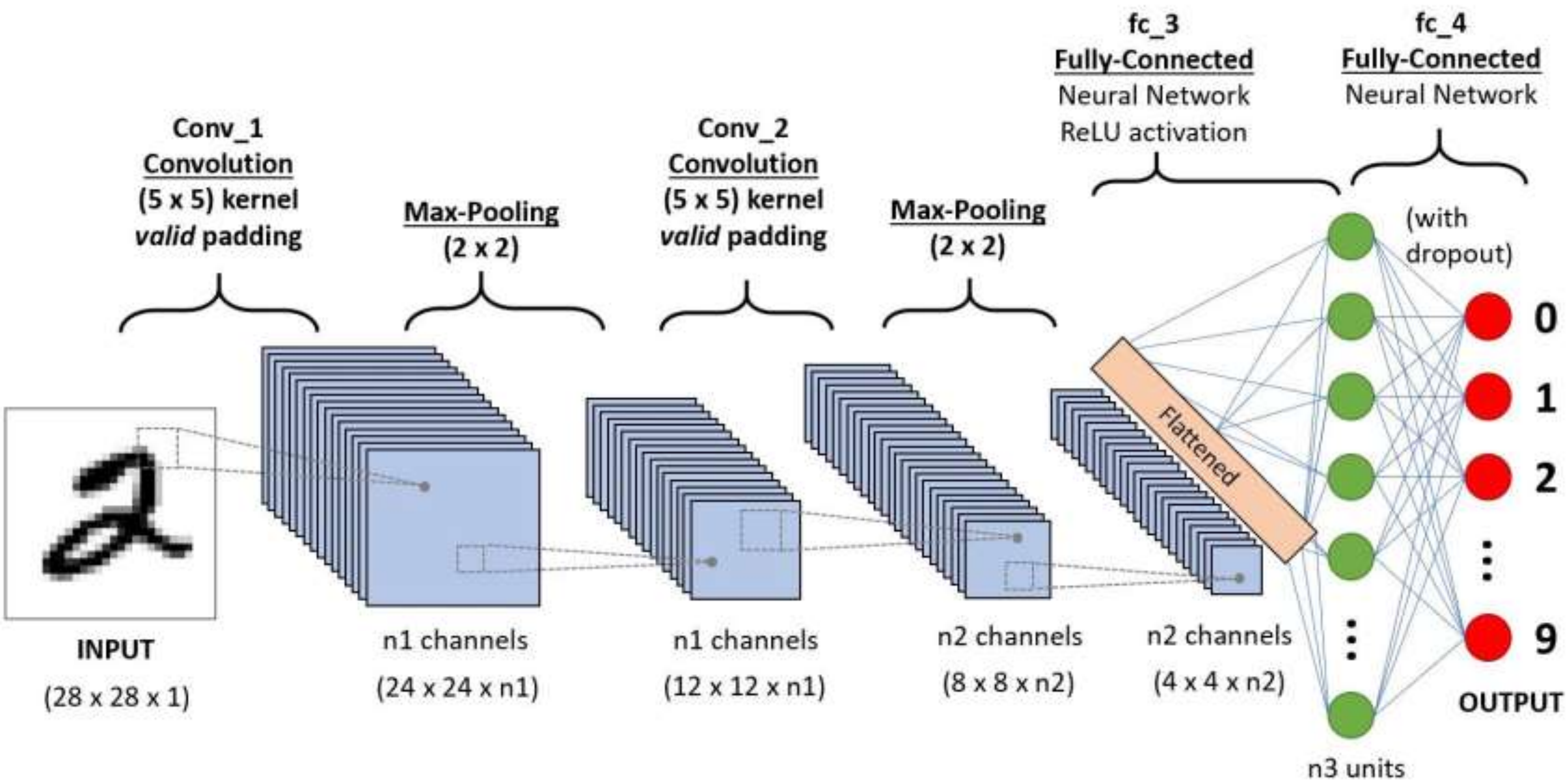
Outline

- Why Convolutional Neural Network (CNN)?
  - Motivation and overview
- What is the CNN?
  - Convolution layers & model complexity
  - Closer look at activation functions
  - Pooling layers & model complexity
  - Math properties
- Examples of CNNs

Acknowledgement: Roger Grosse@UofT & Feifei Li's cs231n notes

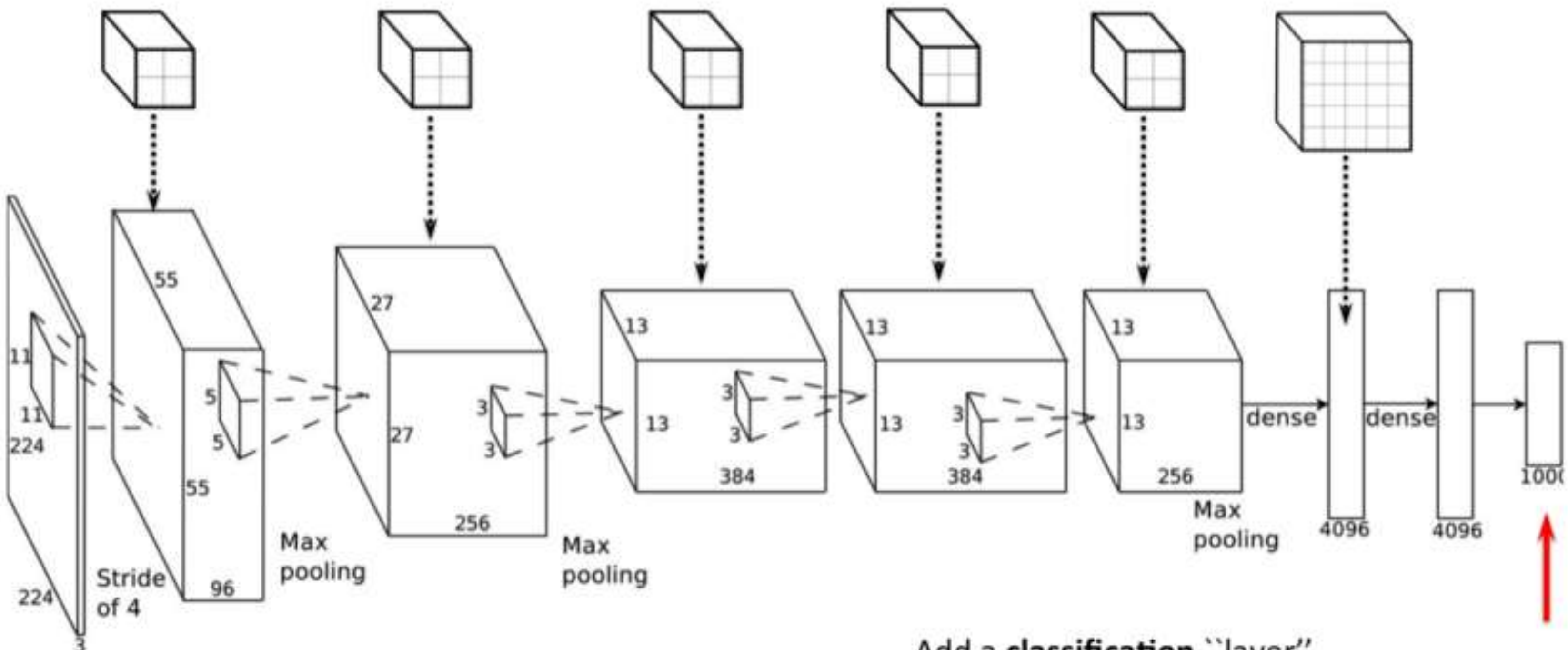
LeNet-5

- Handwritten digit recognition



AlexNet

- Deeper network structure



Add a **classification** ``layer''.

For an input image, the value in a particular dimension of this vector tells you the probability of the corresponding object class.

AlexNet

- Deeper network structure

Outline

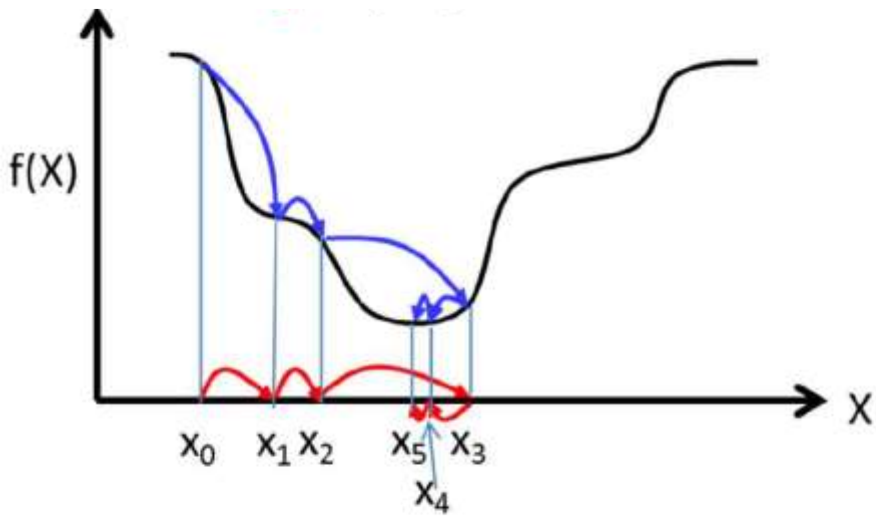
- Why Convolutional Neural Network (CNN)?
- What is the CNN?
- Examples of CNN
- CNN training as optimization
  - Data preprocessing
  - Weight initialization
  - Parameter update
  - Batch normalization (maybe next time)

Acknowledgement: Roger Grosse@UofT & Feifei Li's cs231n notes

Training overview

- Supervised learning paradigm
- Mini-batch SGD

  Loop:
  - Sample a (mini-)batch of data
  - Forward propagation it through the network, compute loss
  - Backpropagation to calculate the gradients
  - Update the parameters using the gradient



Training overview

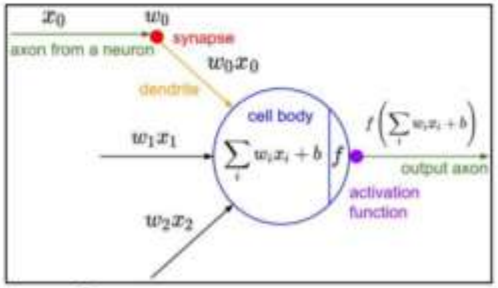
- Two aspects of training networks
  - Optimization
    - How do we minimize the loss function effectively?
  - Generalization
    - How do we avoid overfitting?
- CNN training pipeline
  - Data processing
  - Weight initialization
  - Parameter updates
  - Batch normalization
- Avoid overfitting
  - Next time ……

Data Preprocessing

- Motivation

Consider what happens when the input to a neuron is always positive...

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



What can we say about the gradients on **w**?

We know that local gradient of sigmoid is always positive

We are assuming x is always positive

So!! Sign of gradient **for all w$_i$** is the same as the sign of upstream scalar gradient!

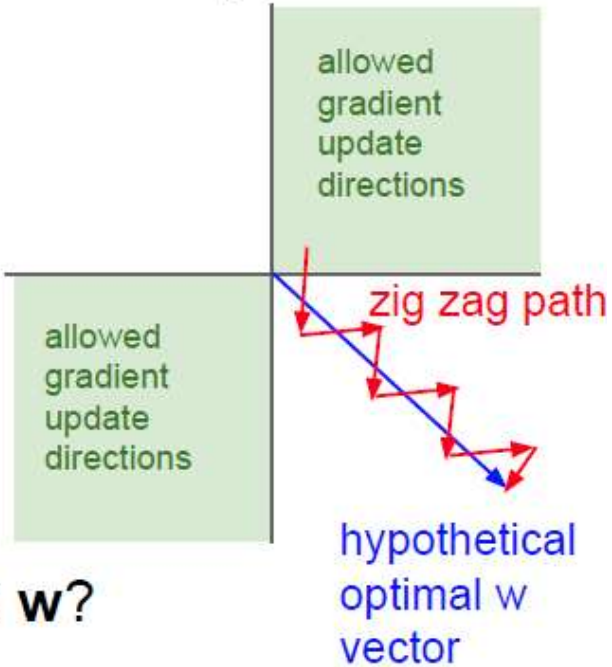
$$\boxed{\frac{\partial L}{\partial w}} = \boxed{\sigma(\textstyle\sum_i w_i x_i + b)(1-\sigma(\sum_i w_i x_i + b))}\boxed{x} \times \boxed{upstream\_gradient}$$

Data Preprocessing

- Motivation

Remember: Consider what happens when the input to a neuron is always positive...

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



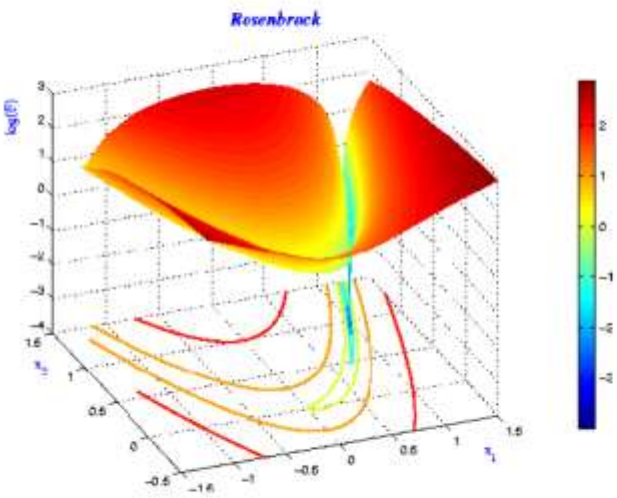
What can we say about the gradients on **w**?

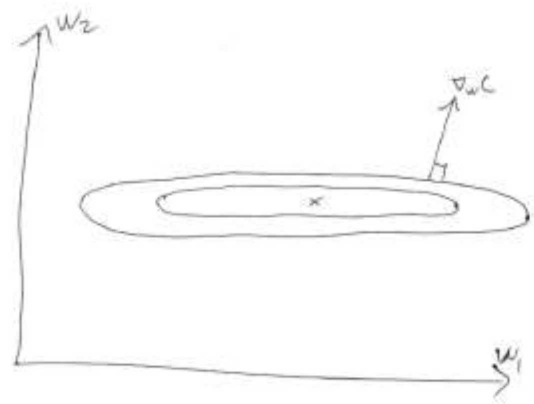
Always all positive or all negative :(

(this is also why you want zero-mean data!)

Data Preprocessing

- Motivation
  - Error surfaces with long, narrow ravines



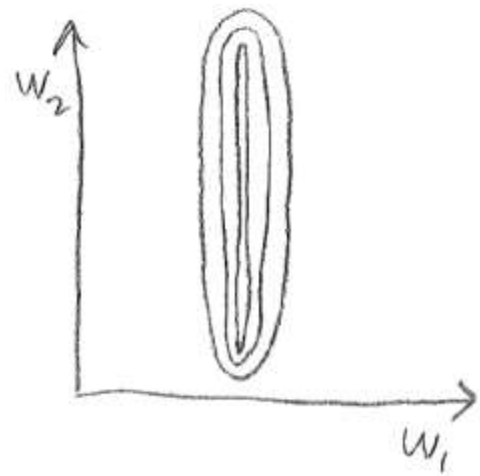


Data Preprocessing

- Motivation
  - Example of linear regression

x_1	x_2	t
114.8	0.00323	5.1
338.1	0.00183	3.2
98.8	0.00279	4.1
⋮	⋮	⋮

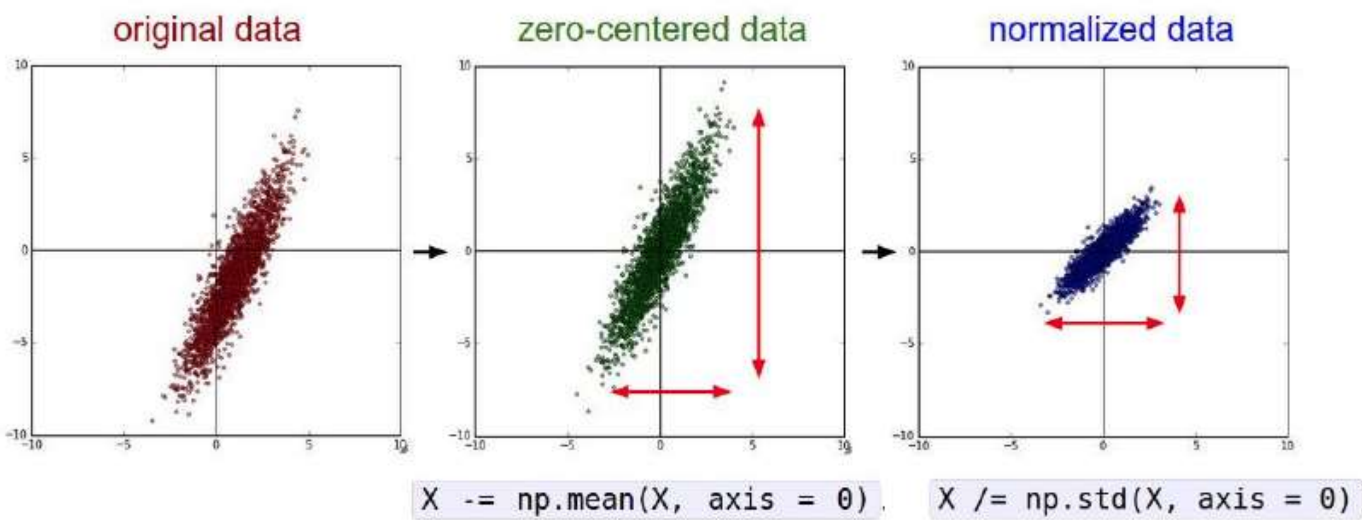
$$\overline{w_i} = \overline{y}\, x_i$$



  - Which direction of weights has a larger gradient updates?
  - Which one do you want to receive a larger update?

Data Preprocessing

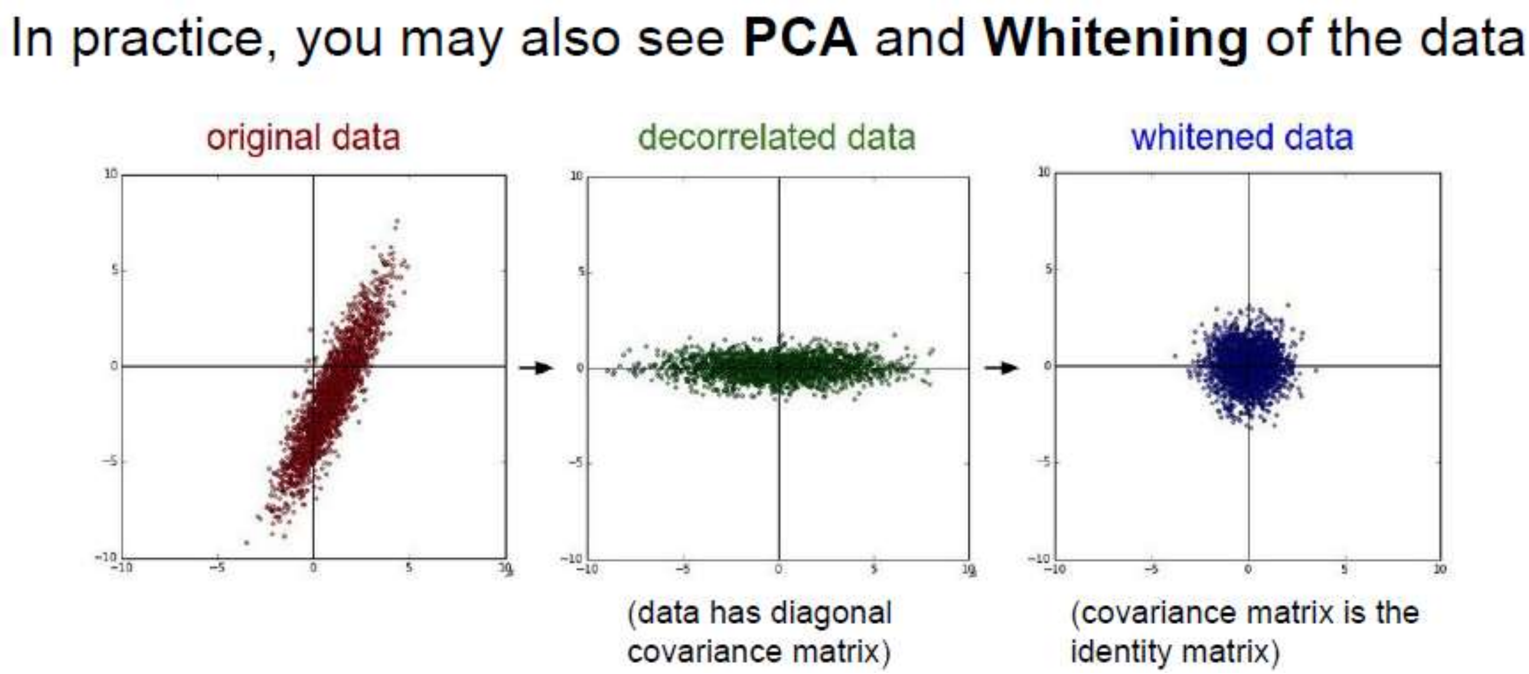
- Data normalization
  - To avoid these problems, center your inputs to zero mean and unit variance



(Assume X [NxD] is data matrix,
each example in a row)

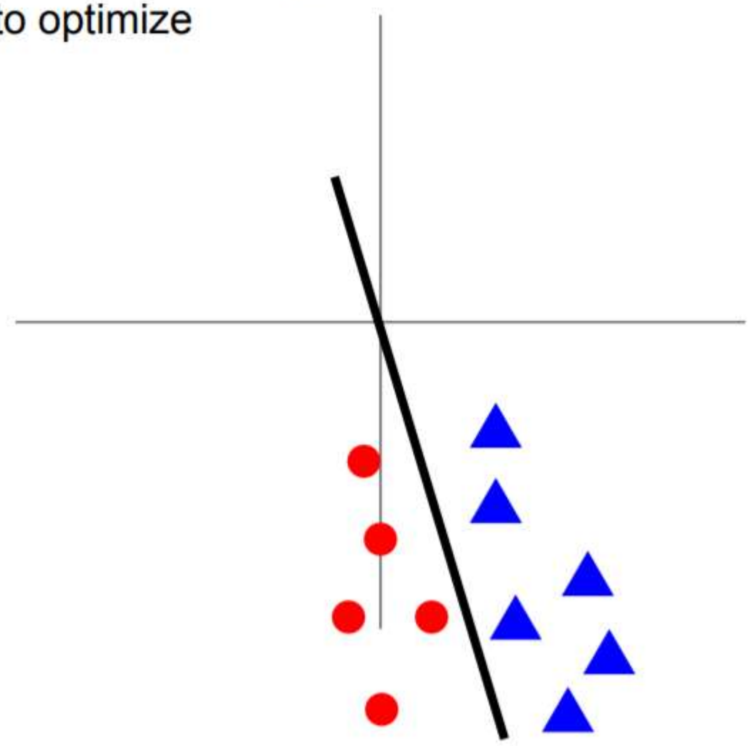
Data Preprocessing

- More advanced methods

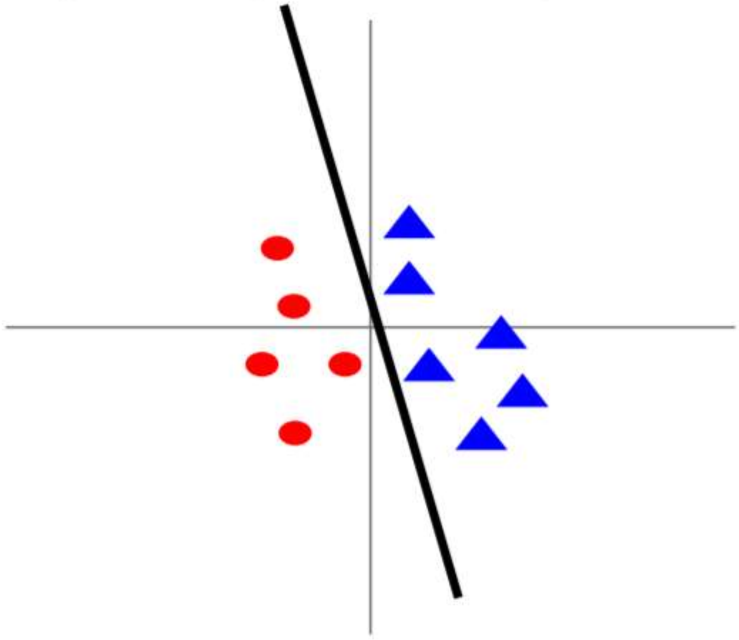


Data Preprocessing

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



After normalization: less sensitive to small changes in weights; easier to optimize



Data Preprocessing

- For visual recognition tasks
  - In practice for images: centering only
  - Not common to do PCA or whitening
- For example, CIFAR-10
  - Subtract the mean image (e.g. AlexNet)

    (mean image = [32,32,3] array)
  - Subtract per-channel mean (e.g. VGGNet)

    (mean along each channel = 3 numbers)
  - Subtract per-channel mean and Divide by per-channel std (e.g. ResNet)

    (mean along each channel = 3 numbers)

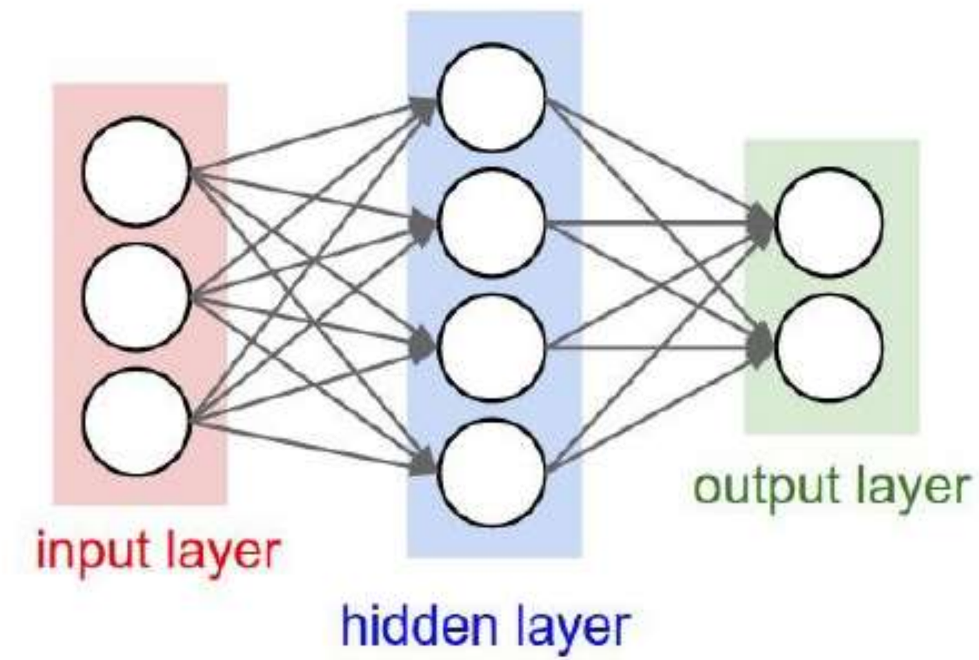
Outline

- Overview of CNN training
- CNN training as optimization
  - Data preprocessing
  - Weight initialization
  - Parameter update
  - Batch normalization

Acknowledgement: UofT, CMU & Feifei Li's cs231n notes

Weight Initialization

- Non-convex objective functions
  - Neural nets have a weight symmetry: permute all the hidden units in a given layer and obtain an equivalent solution.
  - Q: What happens when W=0 initialization is used?



A: All output are 0, all gradients are the same! No “symmetry breaking”

Weight Initialization

- First idea: Small random numbers
  - Gaussian with zero mean and 1e-2 std

```
W = 0.01* np.random.randn(D,H)
```

  - Simpler models to start
  - Outputs are close to uniform for classification

Works ~okay for small networks, but problems with deeper networks.

Weight Initialization

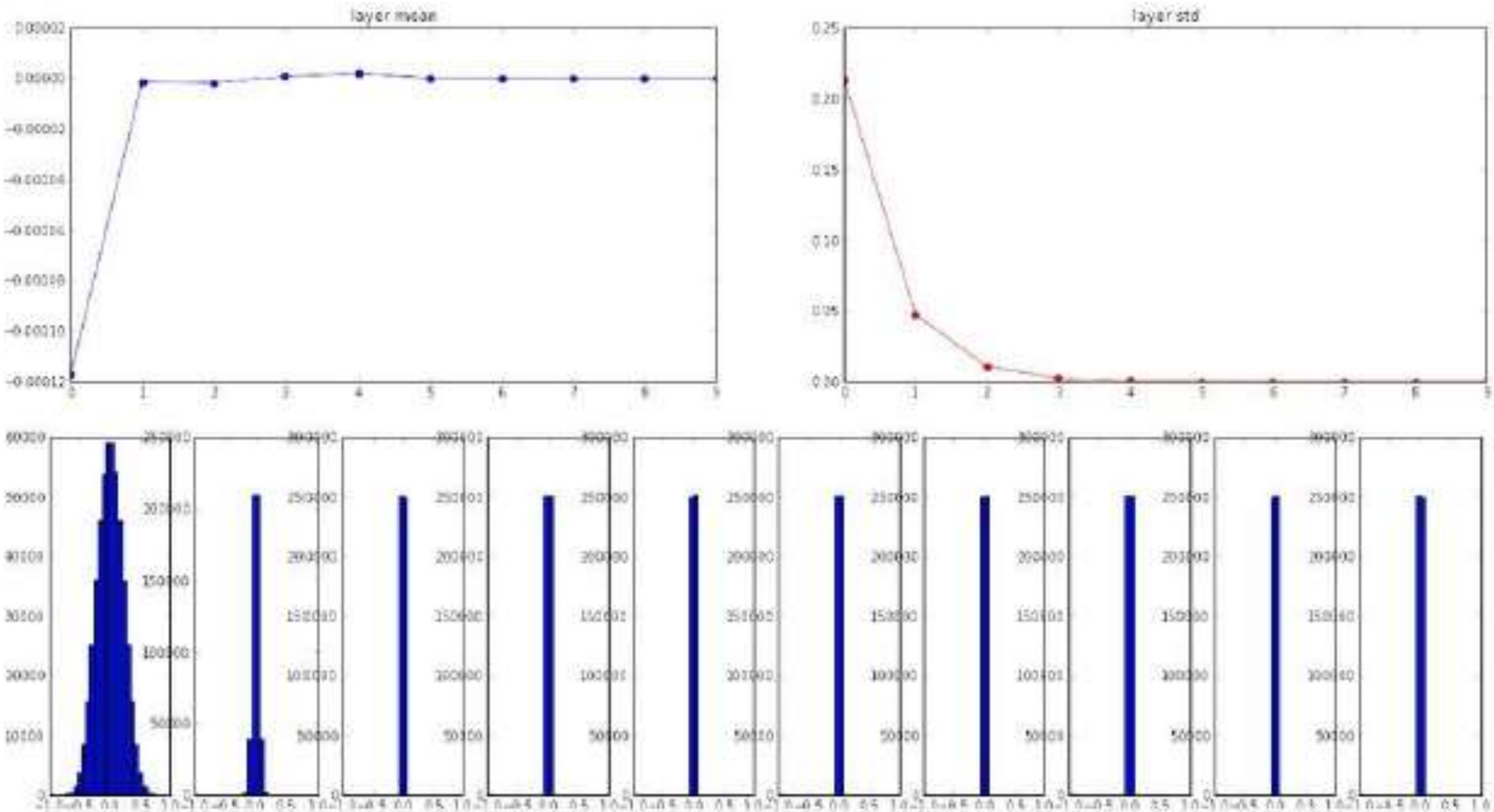
- Motivating example
  - Look at some activation statistics
  - E.g., 10-layer net with 500 neurons on each layer using tanh non-linearities.

```
# assume some unit gaussian 10-D input data
D = np.random.randn(1000, 500)
hidden_layer_sizes = [500]*10
nonlinearities = ['tanh']*len(hidden_layer_sizes)
```

```
act = {'relu':lambda x:np.maximum(0,x), 'tanh':lambda x:np.tanh(x)}
Hs = {}
for i in xrange(len(hidden_layer_sizes)):
    X = D if i == 0 else Hs[i-1] # input at this layer
    fan_in = X.shape[1]
    fan_out = hidden_layer_sizes[i]
    W = np.random.randn(fan_in, fan_out) * 0.01 # layer initialization

    H = np.dot(X, W) # matrix multiply
    H = act[nonlinearities[i]](H) # nonlinearity
    Hs[i] = H # cache result on this layer
```

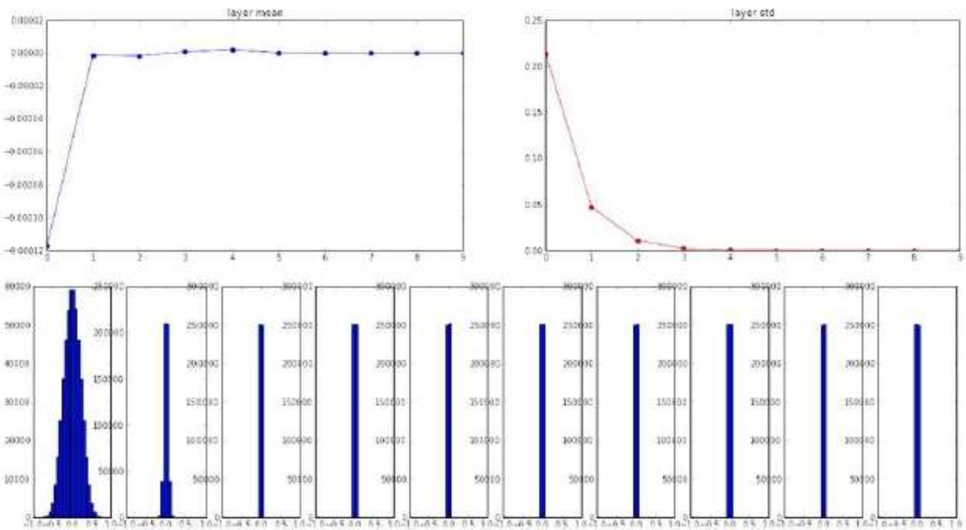
Weight Initialization

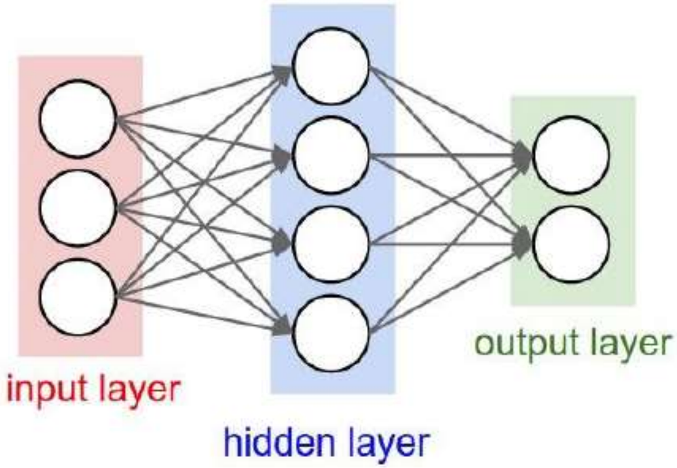
- Motivating example



Weight Initialization

- Motivating example
  - All activations tend to zero for deeper network layers
  - Q: What do the gradients dL/dW look like?





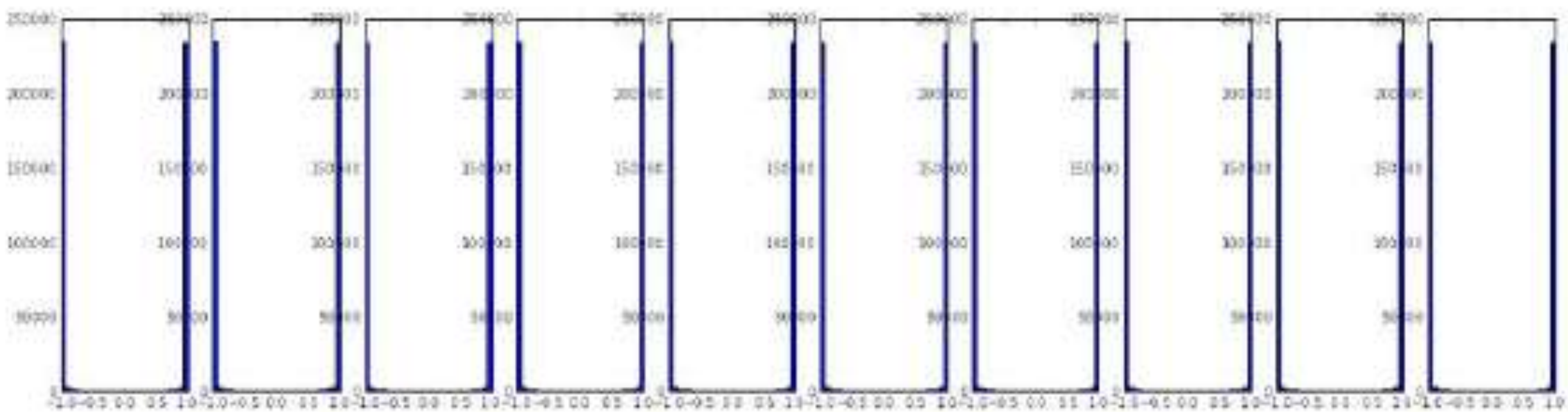
Weight Initialization

- Motivating example

```
W = np.random.randn(fan_in, fan_out) * 1.0 # layer initialization
```

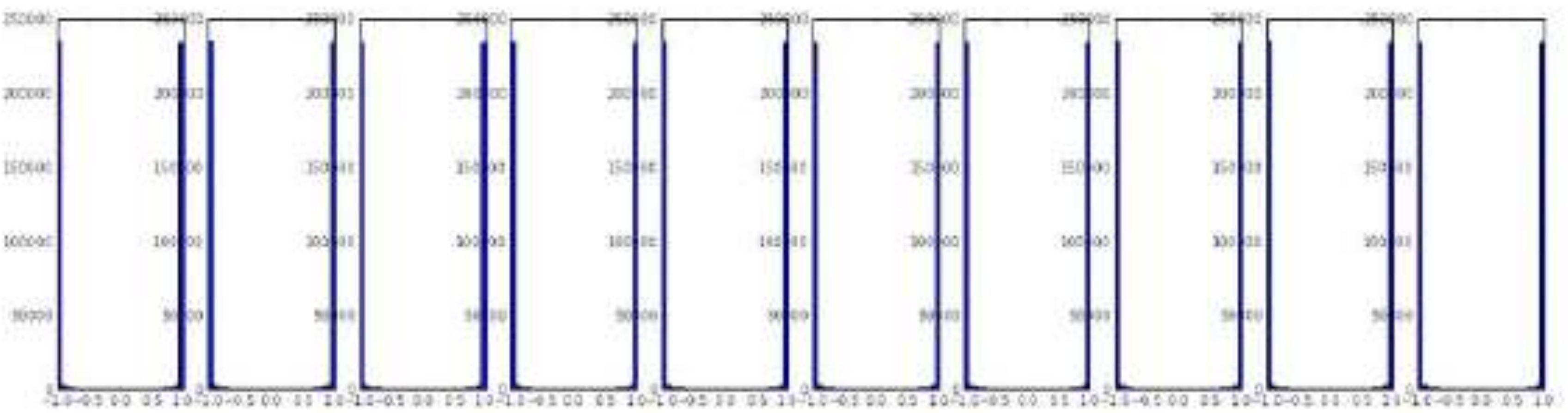
```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean -0.000430 and std 0.981879
hidden layer 2 had mean -0.000849 and std 0.981649
hidden layer 3 had mean 0.000566 and std 0.981601
hidden layer 4 had mean 0.000483 and std 0.981755
hidden layer 5 had mean -0.000682 and std 0.981614
hidden layer 6 had mean -0.000401 and std 0.981560
hidden layer 7 had mean -0.000237 and std 0.981520
hidden layer 8 had mean -0.000448 and std 0.981913
hidden layer 9 had mean -0.000899 and std 0.981728
hidden layer 10 had mean 0.000584 and std 0.981736
```

*1.0 instead of *0.01



Weight Initialization

- Motivating example
  - All activations saturate
  - Q: What do the gradients look like?
  - A: Local gradients all zero

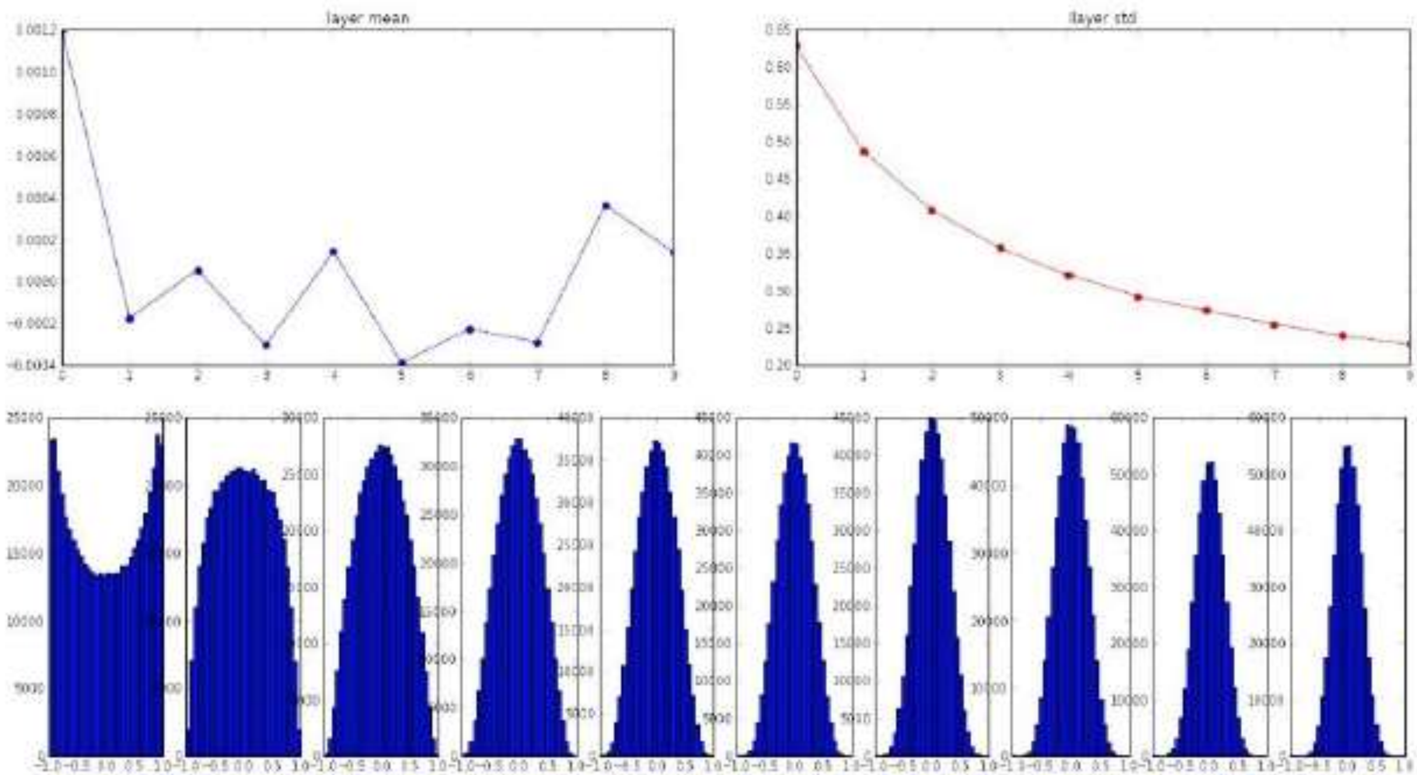


Weight Initialization

- Xavier initialization [Glorot and Bengio, AISTAT 2010]

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

  - std = 1/sqrt(fan_in): activations are nicely scaled for all layers

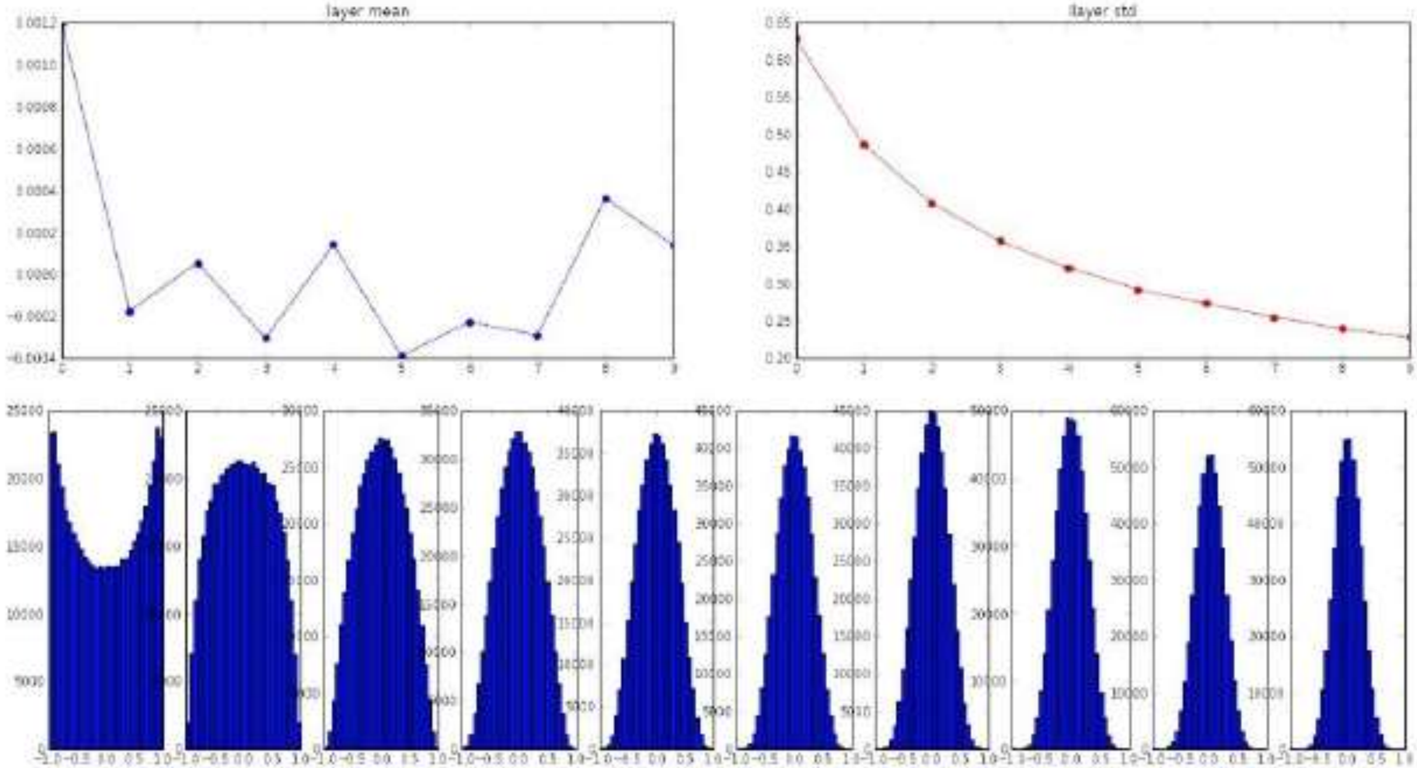


Weight Initialization

- Xavier initialization [Glorot and Bengio, AISTAT 2010]

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

  - For conv layers, fan_in is filter_size^2 * input_channels



Weight Initialization

- Theoretic analysis

Suppose we have an input X with n components and a fully connected layer (also denoted linear or dense) with random weights W that outputs a number Y such that

$$Y = W_1X_1 + W_2X_2 + \ldots + W_nX_n$$

To make sure that the weights remain in a reasonable range, we expect that $Var(Y) = Var(X_i)_{i \in [1,n]}$

We also know how to compute the variance of the product of two random variables. Therefore

$$Var(W_iX_i) = E[X_i]^2 Var(W_i) + E[W_i]^2 Var(X_i) + Var(W_i)Var(X_i)$$

Both our inputs and weights have a mean 0. It simplifies to

$$Var(W_iX_i) = Var(W_i)Var(X_i)$$

Now we make a further assumption that the X_i and W_i are all independent and identically distributed (iid).

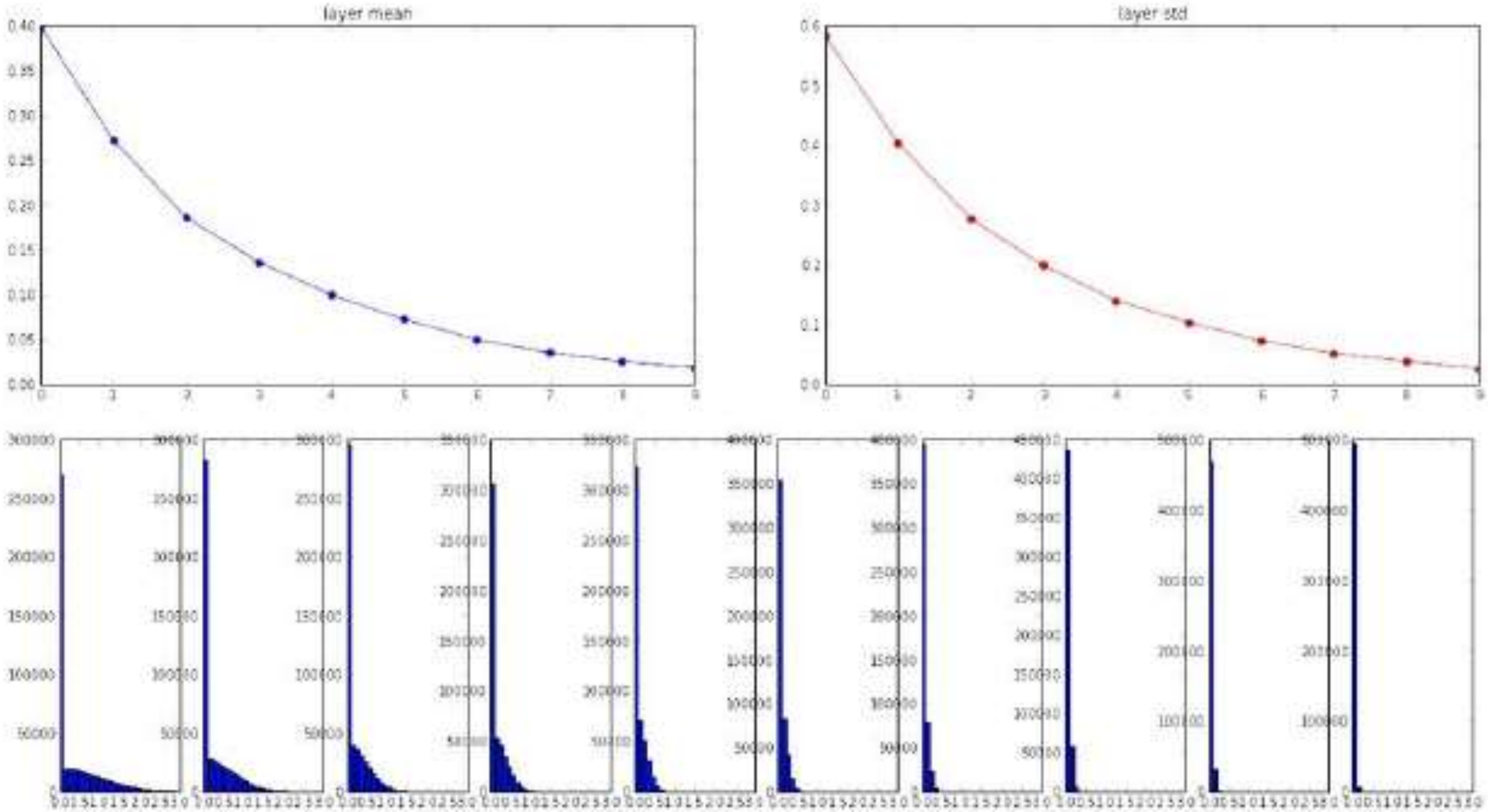
$$Var(Y) = Var(W_1X_1 + W_2X_2 + \ldots + W_nX_n) = nVar(W_i)Var(X_i)$$

It turns that, if we want to have $Var(Y) = Var(X_i)$, we must enforce the condition $nVar(W_i) = 1$.

$$Var(W_i) = \frac{1}{n} = \frac{1}{n_{in}}$$

Weight Initialization

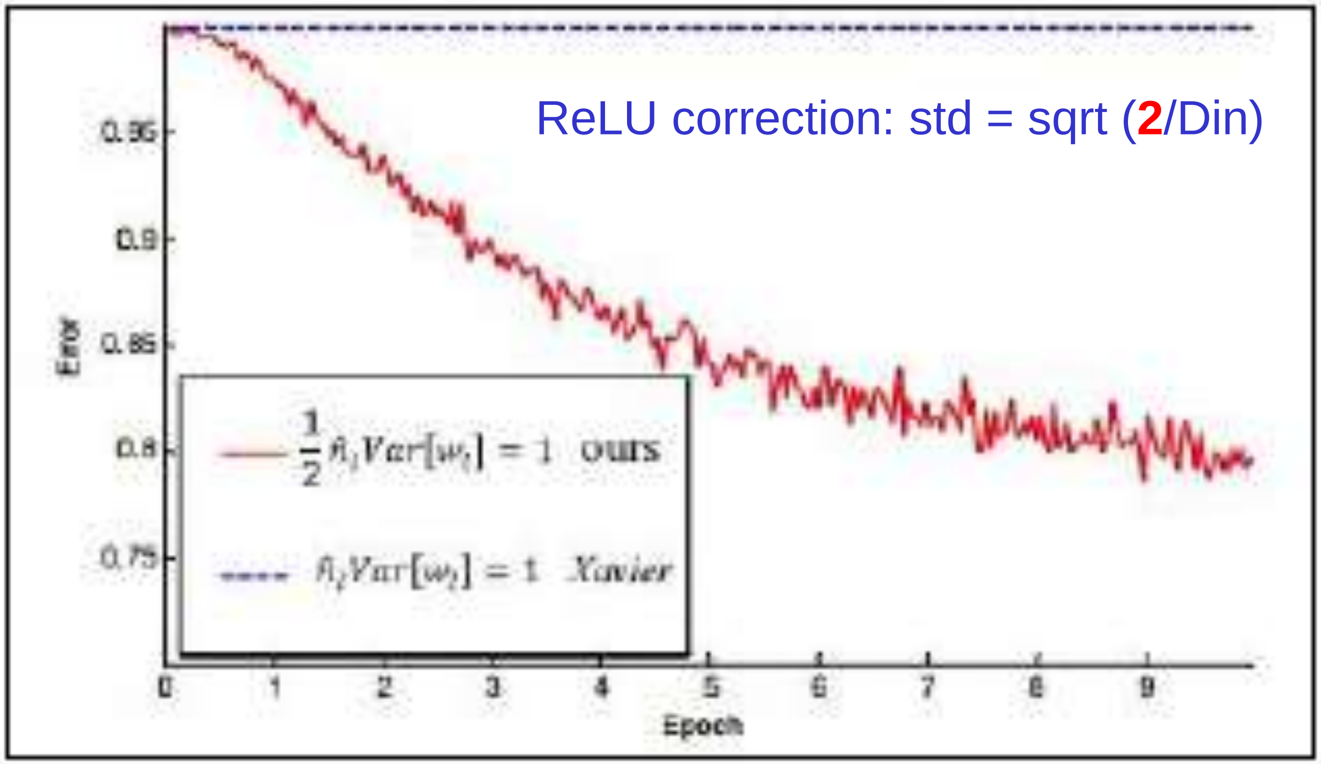
- Problems with ReLU activation
  - Xavier initialization assumes zero centered activation function, and hence breaks under ReLU



Weight Initialization

- Initialization for CNNs with ReLU [He et al., 2015]

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in/2) # layer initialization
```



He et al, “Delving Deep into Rectifiers: Surpassing Human-level Performance on ImageNet Classification”, ICCV 2015

Weight Initialization

- Weight initialization is an active are of research…
  - Understanding the difficulty of training deep feedforward neural networks *by Glorot and Bengio, 2010*
  - Exact solutions to the nonlinear dynamics of learning in deep linear neural networks *by Saxe et al, 2013*
  - Random walk initialization for training very deep feedforward networks *by Sussillo and Abbott, 2014*
  - Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification *by He et al., 2015*
  - Data-dependent Initializations of Convolutional Neural Networks *by Krähenbühl et al., 2015*
  - All you need is a good init, *Mishkin and Matas, 2015*
  - Fixup Initialization: Residual Learning Without Normalization, *Zhang et al, 2019*
  - The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks, *Frankle and Carbin, 2019*

Outline

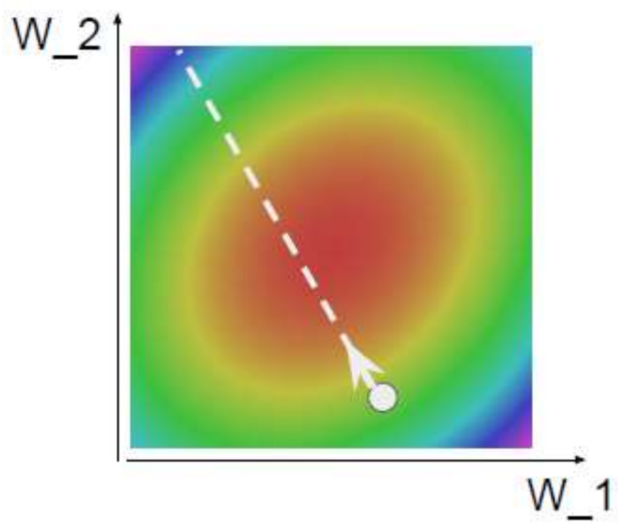
- Overview of CNN training
- CNN training as optimization
  - Data preprocessing
  - Weight initialization
  - Parameter update
  - Batch normalization

Acknowledgement: UofT, CMU & Feifei Li's cs231n notes

Optimization

- Stochastic Gradient Descent

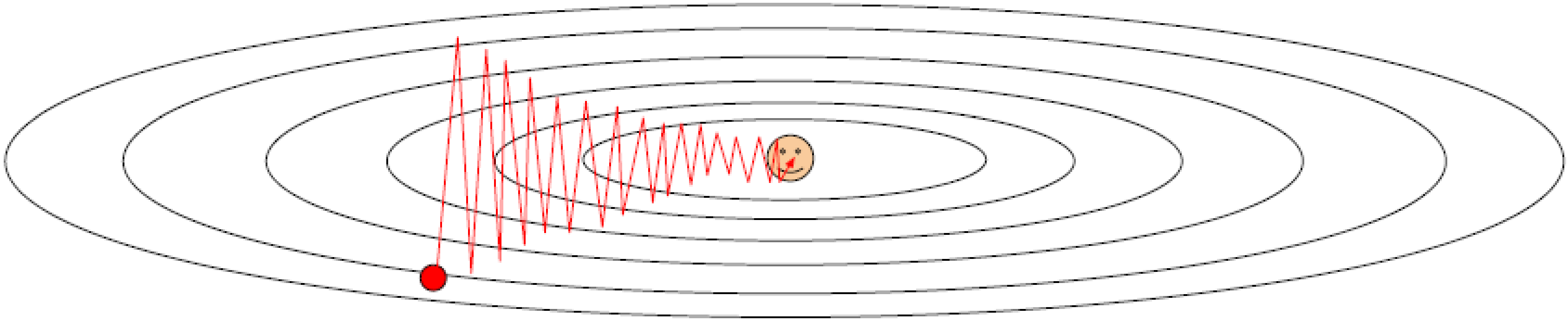
```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```



Optimization

- Problems with SGD

What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?
Very slow progress along shallow dimension, jitter along steep direction



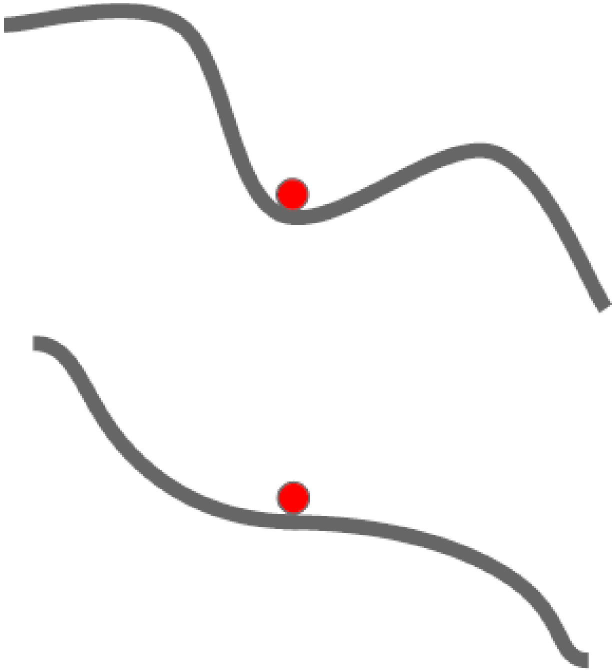
Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

Optimization

- Problems with SGD

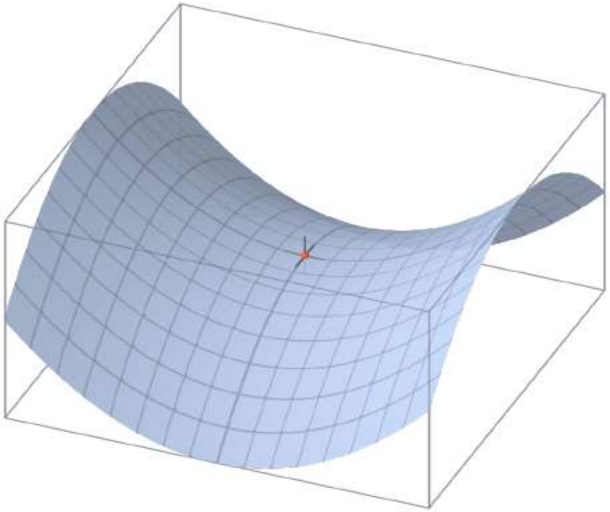
What if the loss function has a **local minima** or **saddle point**?

Zero gradient, gradient descent gets stuck



Optimization

- Problems with SGD
  - Saddle points are more common in high-dim space



At a saddle point $\frac{\partial \mathcal{E}}{\partial \boldsymbol{\theta}} = 0$, even though we are not at a minimum. Some directions curve upwards, and others curve downwards.

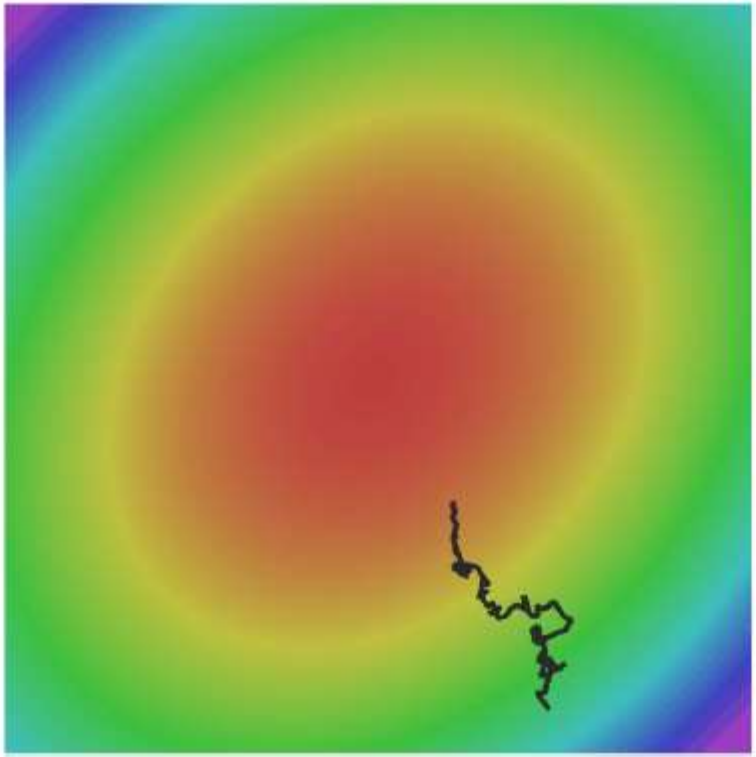
Optimization

- Problems with SGD

Our gradients come from minibatches so they can be noisy!

$$L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(x_i, y_i, W)$$

$$\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W)$$



Optimization

- SGD + Momentum

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x += learning_rate * dx
```

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

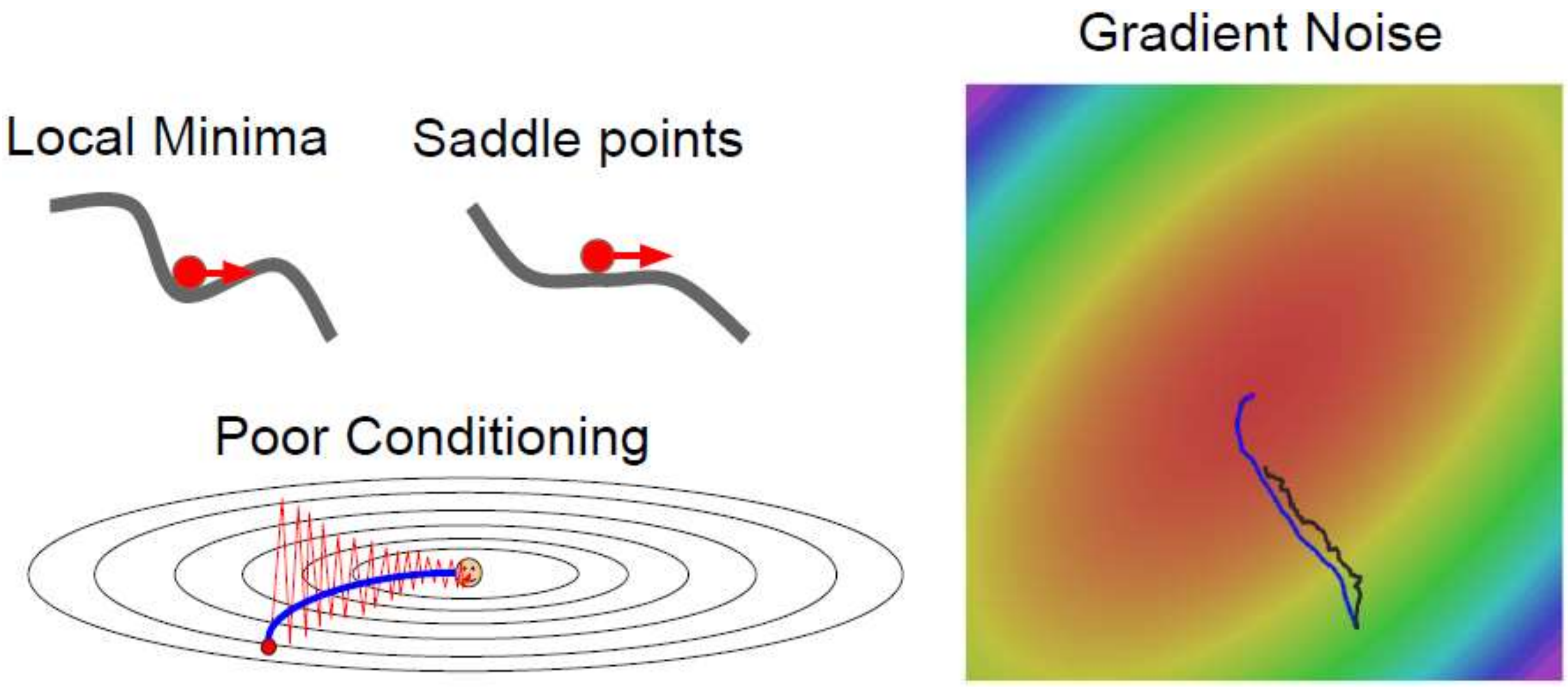
```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x += learning_rate * vx
```

- Build up "velocity" as a running mean of gradients
- Rho gives "friction"; typically rho=0.9 or 0.99

Sutskever et al, "On the importance of initialization and momentum in deep learning", ICML 2013

Optimization

- SGD + Momentum
  - Momentum sometimes helps a lot, and almost never hurts



Optimization

- SGD + Momentum
  - You may see SGD+Momentum formulated different ways, but they are equivalent - give same sequence of x

SGD+Momentum

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t)$$
$$x_{t+1} = x_t + v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx - learning_rate * dx
  x += vx
```

SGD+Momentum

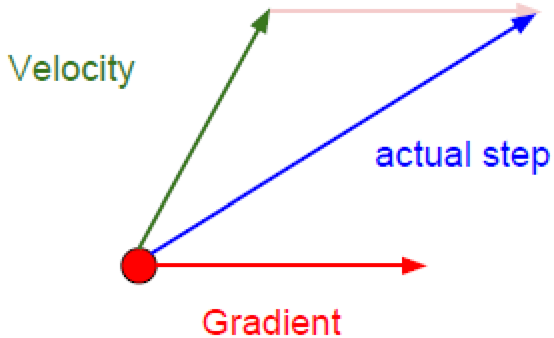
$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

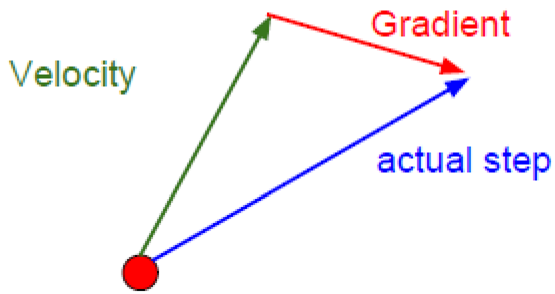
Optimization

- Nesterov Momentum
  - "Look ahead" to the point where updating using velocity would take us;
  - Compute gradient there and mix it with velocity to get actual update direction

Momentum update:



Nesterov Momentum



$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$
$$x_{t+1} = x_t + v_{t+1}$$

Nesterov, "A method of solving a convex programming problem with convergence rate O(1/k^2)", 1983
Nesterov, "Introductory lectures on convex optimization: a basic course", 2004
Sutskever et al, "On the importance of initialization and momentum in deel learning", ICML 2013

Optimization

- Nesterov Momentum

$$v_{t+1} = \rho v_t - \alpha \nabla f(\boxed{x_t + \rho v_t})$$
$$x_{t+1} = x_t + v_{t+1}$$

Annoying, usually we want update in terms of $x_t, \nabla f(x_t)$

Optimization

- Nesterov Momentum

$$v_{t+1} = \rho v_t - \alpha \nabla f(\boxed{x_t + \rho v_t})$$
$$x_{t+1} = x_t + v_{t+1}$$

Annoying, usually we want update in terms of $x_t, \nabla f(x_t)$

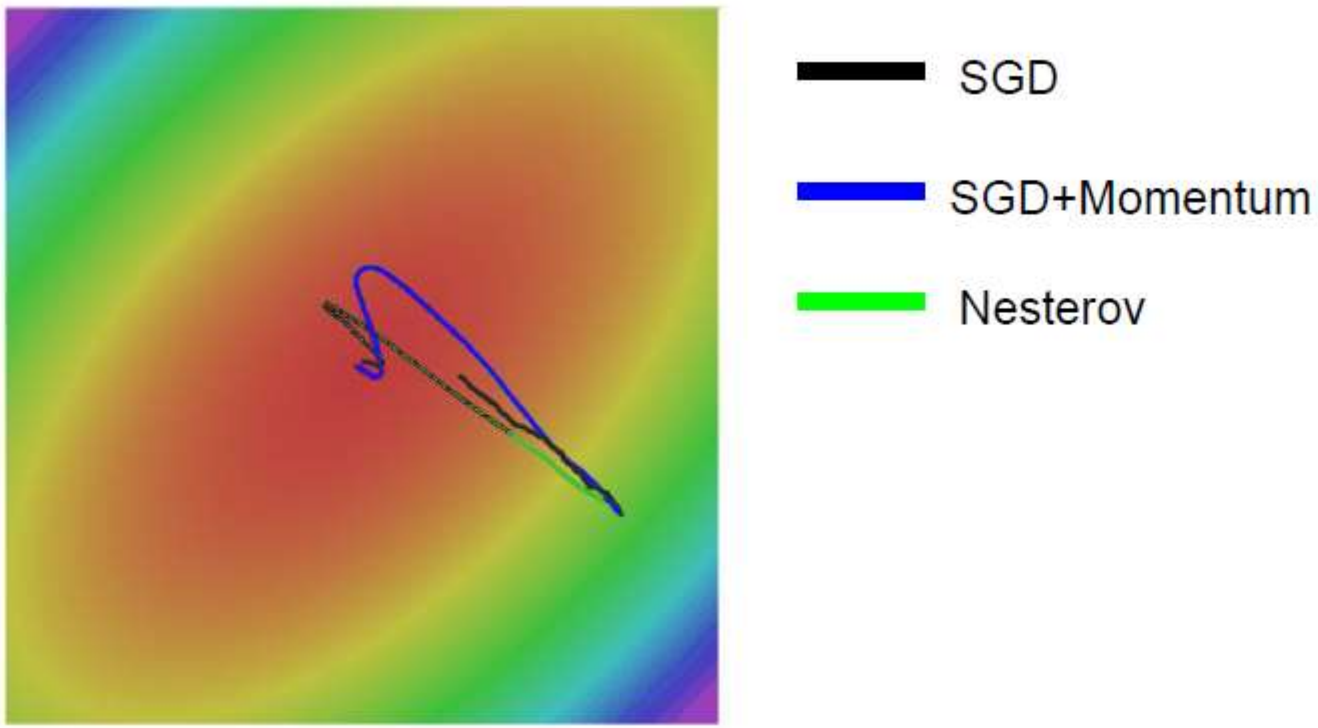
Change of variables $\tilde{x}_t = x_t + \rho v_t$ and rearrange:

$$\begin{aligned} v_{t+1} &= \rho v_t - \alpha \nabla f(\tilde{x}_t) \\ \tilde{x}_{t+1} &= \tilde{x}_t - \rho v_t + (1+\rho) v_{t+1} \\ &= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t) \end{aligned}$$

```
dx = compute_gradient(x)
old_v = v
v = rho * v - learning_rate * dx
x += -rho * old_v + (1 + rho) * v
```

Optimization

- Nesterov Momentum



Optimization

- AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

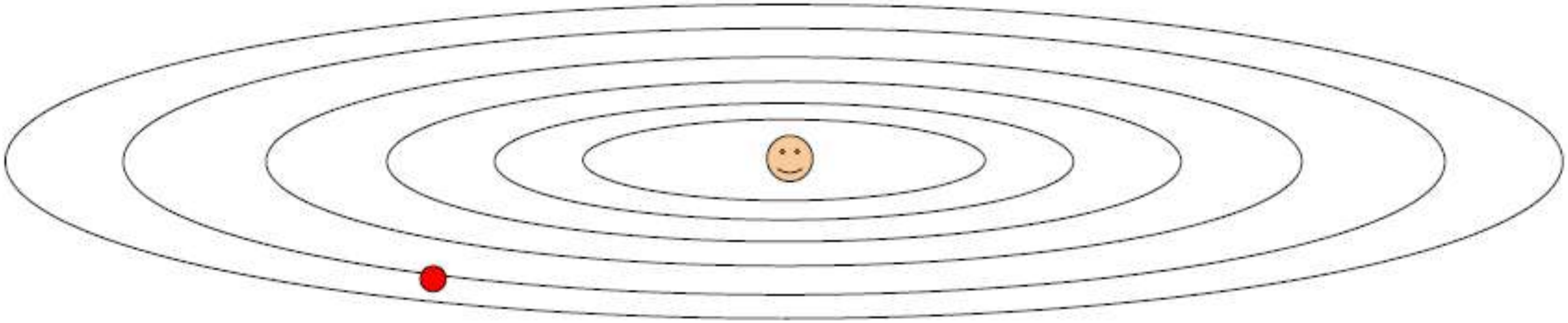
“Per-parameter learning rates” or “adaptive learning rates”

Duchi et al, “Adaptive subgradient methods for online learning and stochastic optimization”, JMLR 2011

Optimization

- AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



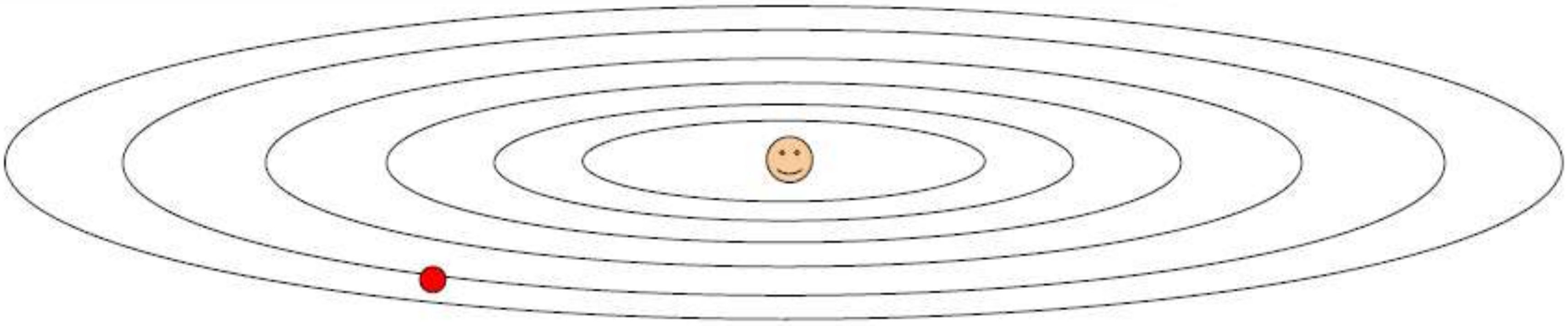
Q: What happens with AdaGrad?

Progress along “steep” directions is damped;
progress along “flat” directions is accelerated

Optimization

- AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



Q2: What happens to the step size over long time?

Decays to zero

Optimization

- RMSProp: smoothed version

AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

↓

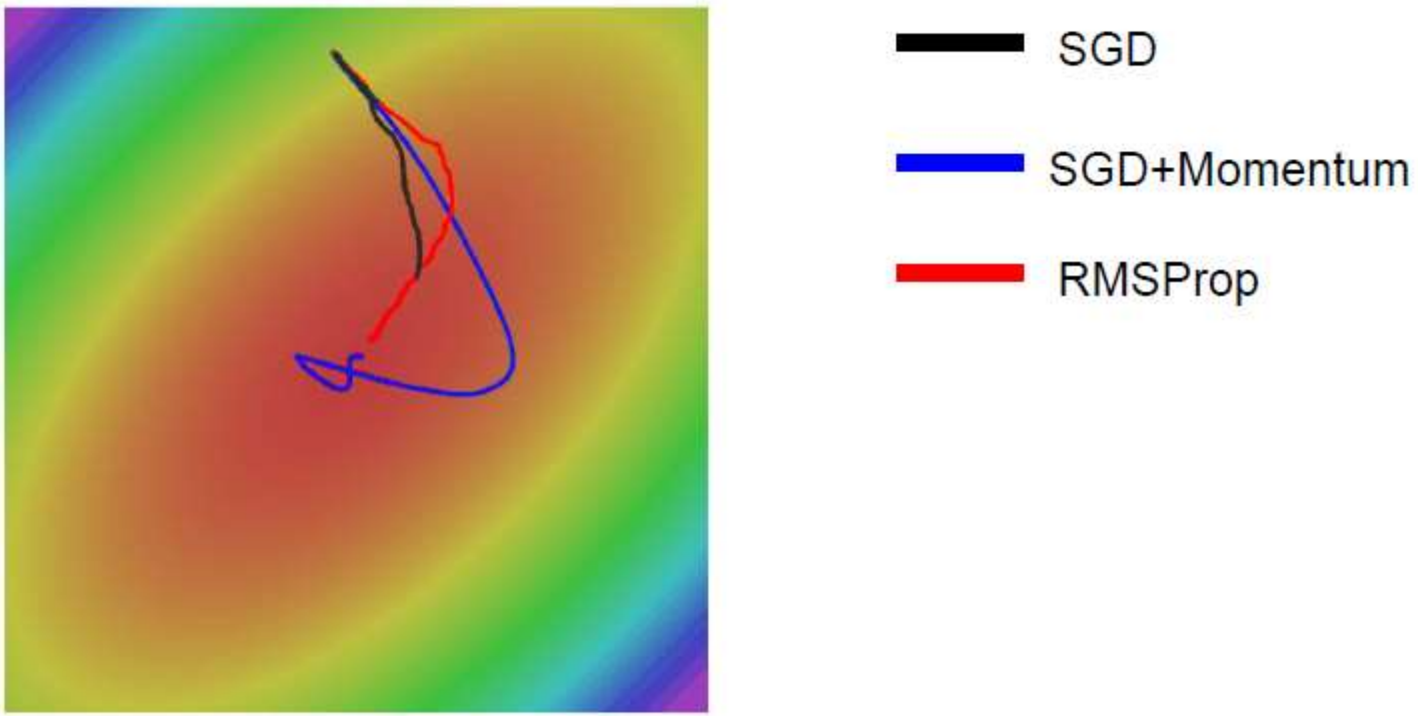
RMSProp

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Tieleman and Hinton, 2012

Optimization

- RMSProp



Optimization

- Adam (almost)

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

Optimization

- Adam (almost)

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Momentum

AdaGrad / RMSProp

Sort of like RMSProp with momentum

Q: What happens at first timestep?

Optimization

- Adam (full form)

```
first_moment = 0
second_moment = 0
for t in range(num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

AdaGrad / RMSProp

Bias correction for the fact that first and second moment estimates start at zero

Optimization

- Adam (full form)

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

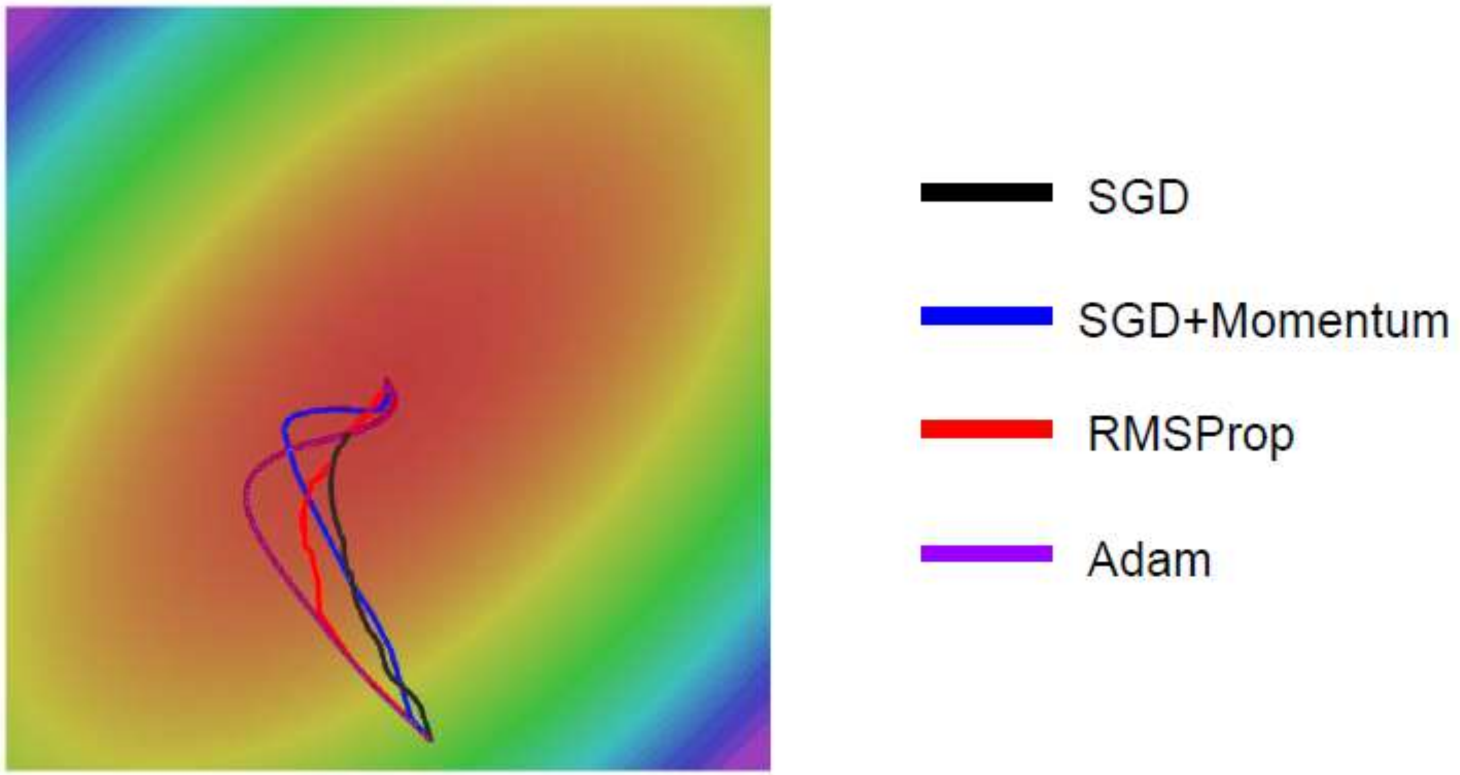
AdaGrad / RMSProp

Bias correction for the fact that first and second moment estimates start at zero

Adam with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-3 or 5e-4 is a great starting point for many models!

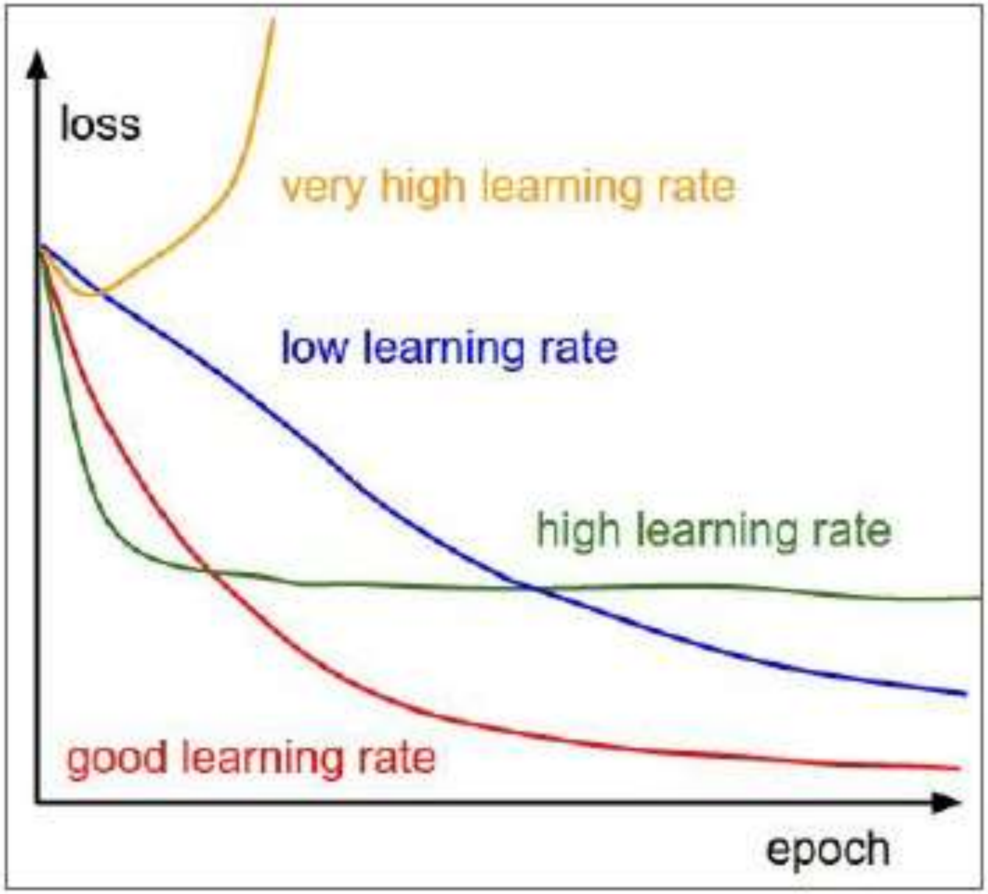
Optimization

- Adam (full form)



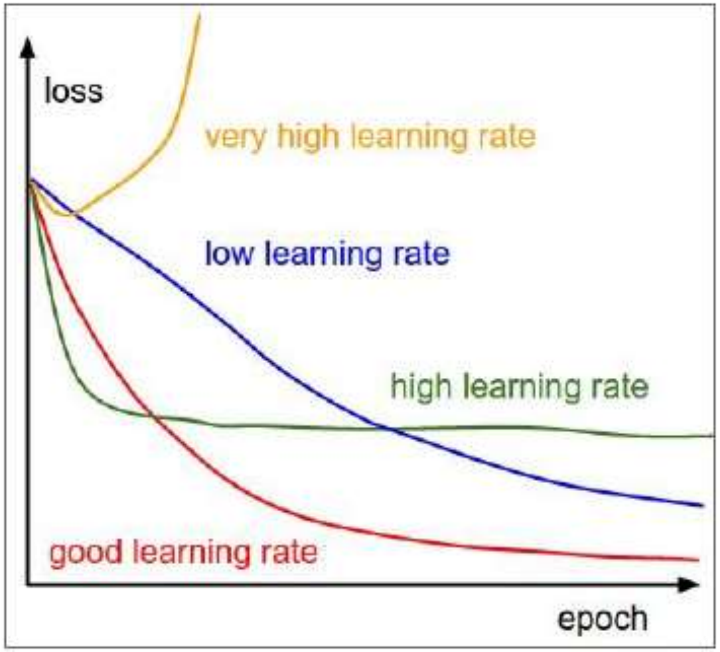
Learning rate

- SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have learning rate as a hyperparameter



Learning rate

- SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have learning rate as a hyperparameter



=> Learning rate decay over time!

step decay:
e.g. decay learning rate by half every few epochs.

exponential decay:

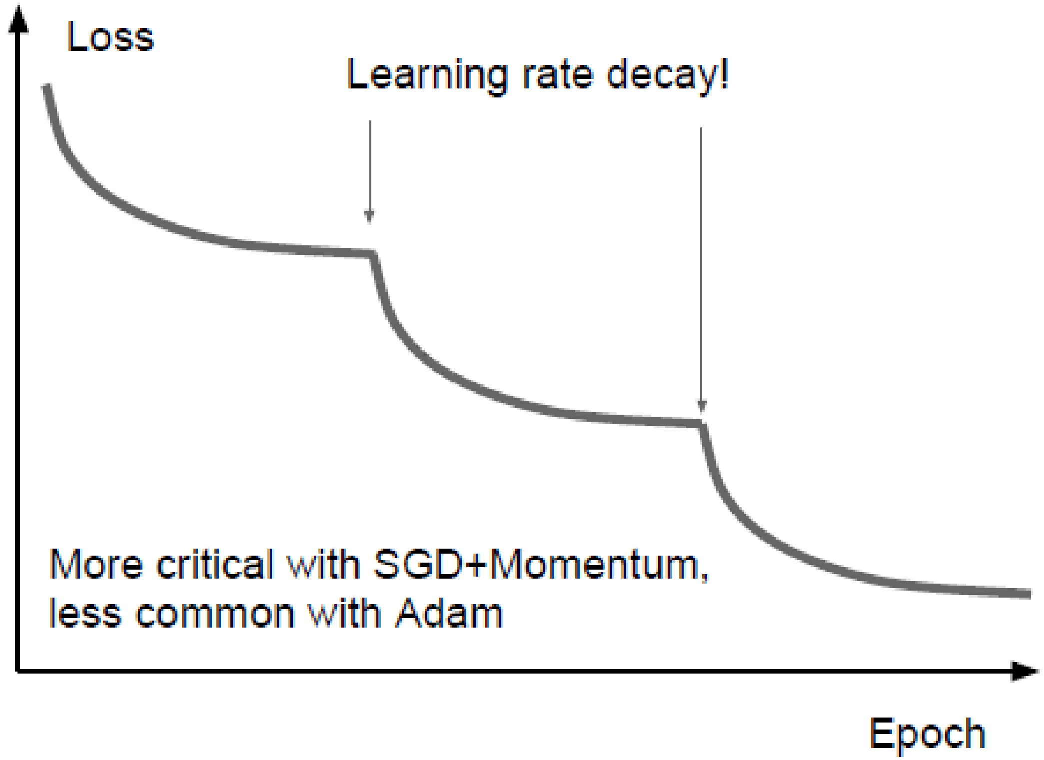
$$\alpha = \alpha_0 e^{-kt}$$

1/t decay:

$$\alpha = \alpha_0 / (1 + kt)$$

Learning rate decay

- Step: reduce learning rate at a few fixed points.
  - E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.



Learning rate decay

- Cosine

$$\alpha_t = \frac{1}{2}\alpha_0\left(1 + \cos(t\pi/T)\right)$$

α_0 : Initial learning rate
α_t : Learning rate at epoch t
T : Total number of epochs

- Linear

$$\alpha_t = \alpha_0(1 - t/T)$$

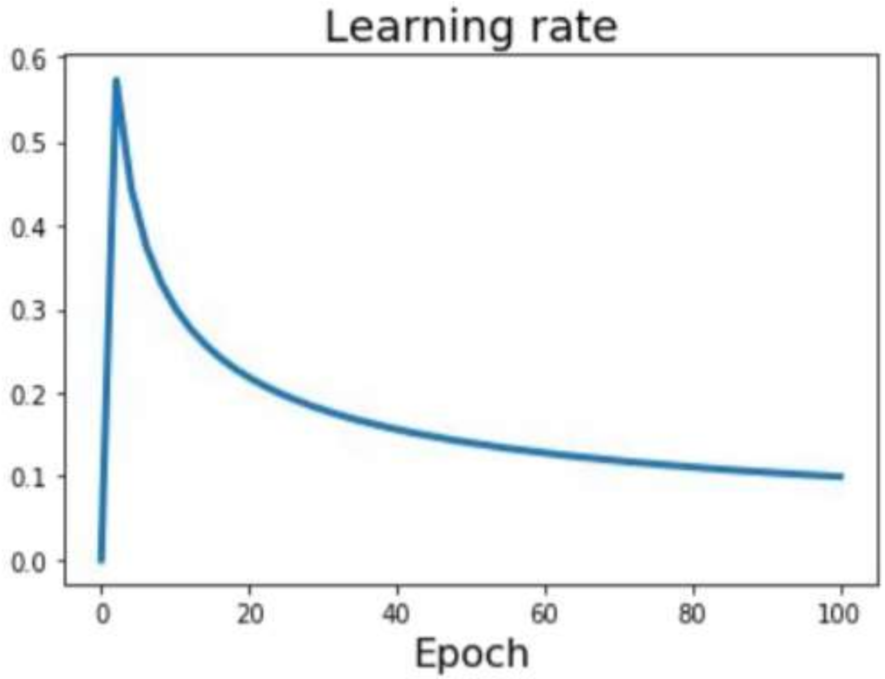
- Inverse sqrt

$$\alpha_t = \alpha_0/\sqrt{t}$$

Loshchilov and Hutter, "SGDR: Stochastic Gradient Descent with Warm Restarts", ICLR 2017
Radford et al, "Improving Language Understanding by Generative Pre-Training", 2018
Feichtenhofer et al, "SlowFast Networks for Video Recognition", arXiv 2018
Child at al, "Generating Long Sequences with Sparse Transformers", arXiv 2019
Devlin et al, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding", 2018
Vaswani et al, "Attention is all you need", NIPS 2017

Learning rate decay

- Linear warmup



High initial learning rates can make loss explode; linearly increasing learning rate from 0 over the first ~5000 iterations can prevent this

Empirical rule of thumb: If you increase the batch size by N, also scale the initial learning rate by N

Goyal et al, "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour", arXiv 2017

What can we find

- Popular hypothesis
  - In large networks, saddle points are far more common than local minima
  - Gradient descent algorithms often get “stuck” in saddle points
  - Most local minima are equivalent and close to global minimum

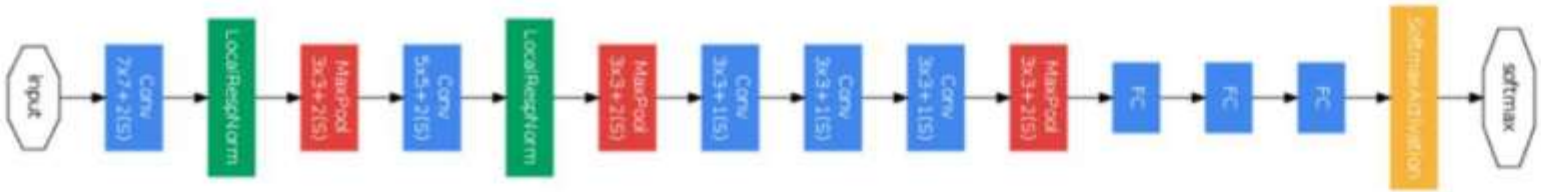
Outline

- Overview of CNN training
- CNN training as optimization
  - Data preprocessing
  - Weight initialization
  - Parameter update
  - Batch normalization

Acknowledgement: UofT, CMU & Feifei Li's cs231n notes

Batch Normalization

- Problem in deep network learning



$$\ell = F_2(F_1(\mathrm{u}, \Theta_1), \Theta_2)$$

  - Change of distribution in activation across layers

Batch Normalization

- Normalize the inputs to a layer:

"you want unit gaussian activations? just make them so."

consider a batch of activations at some layer.
To make each dimension unit gaussian, apply:

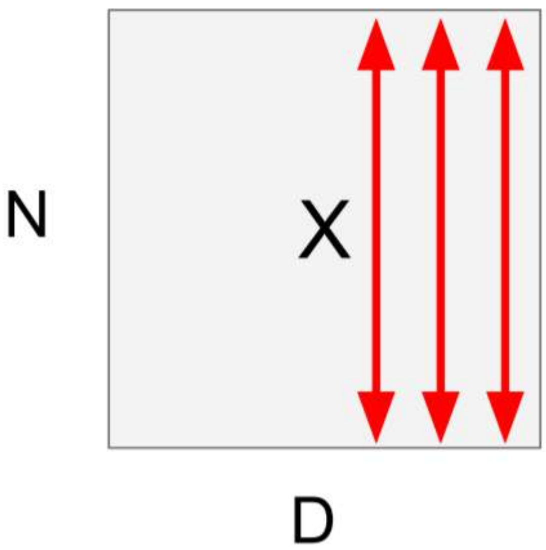
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

this is a vanilla differentiable function...

Batch Normalization

- Layer details

Input: $x : N \times D$



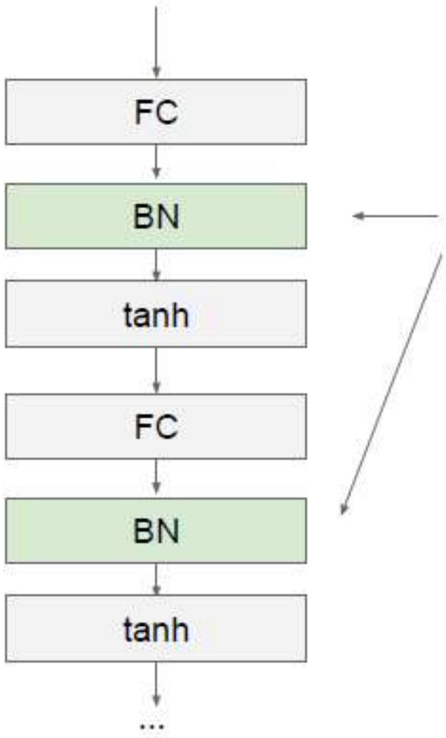
$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$ Per-channel mean, shape is D

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

Batch Normalization

- Layer details



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

- Extra capacity:

Input: $x : N \times D$

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$ Per-channel mean, shape is D

Learnable scale and shift parameters:

$$\gamma, \beta : D$$

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

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

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$ Output, Shape is N x D

Batch Normalization

- Algorithm

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

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

$\mu_j =$ (Running) average of values seen during training — Per-channel mean, shape is D

$\sigma_j^2 =$ (Running) average of values seen during training — Per-channel var, shape is D

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

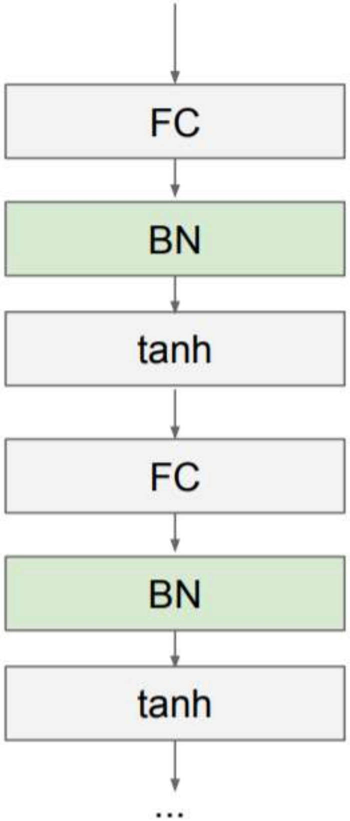
Normalized x, Shape is N x D

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

Output, Shape is N x D

Batch Normalization

- Benefits



- Makes deep networks **much** easier to train!
- Improves gradient flow
- 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!
- Behaves differently during training and testing: this is a very common source of bugs!

Summary of CNNs

- CNN properties [Bronstein et al., 2018]
  - Convolutional (Translation invariance)
  - Scale Separation (Compositionality)
  - Filters localized in space (Deformation Stability)
  - O(1) parameters per filter (independent of input image size n)
  - O(n) complexity per layer (filtering done in the spatial domain)
  - O(log n) layers in classification tasks
- CNN training as optimization task
  - Non-convex and local minimal
  - Overcoming ravines in loss surfaces
  - Data pre-processing + weight initialization + first-order update
  - Batch normalization
- Next time …
  - Structure design of Modern CNNs
- Reference
  - CS231n course notes http://cs231n.github.io/convolutional-networks/
  - D2L Chapter 6 + DLBook Chapter 9