CS 179: LECTURE 16

MODEL COMPLEXITY, REGULARIZATION, AND CONVOLUTIONAL NETS

LAST TIME

- Intro to cuDNN
- Deep neural nets using cuBLAS and cuDNN

TODAY

- Building a better model for image classification
- Overfitting and regularization
- Convolutional neural nets

MODEL COMPLEXITY

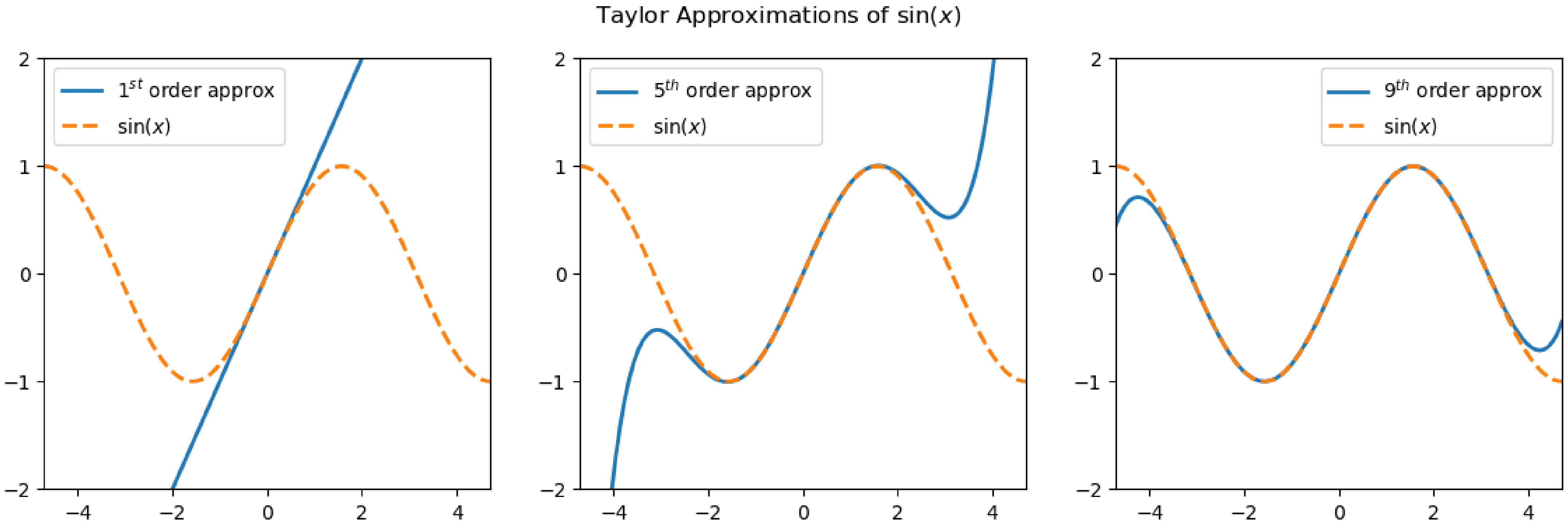
- Consider a class of models $f(x; w)$
  - A function f of an input x with parameters w
  - For now, let's just consider $x \in \mathbb{R}$ (1D input) as a toy example
- Polynomial regression fits a polynomial of degree d to our input, i.e. $f(x; w) = w_0 + w_1 x + w_2 x^2 + \cdots + w_d x^d$
- Intuitively, a higher degree polynomial is a more complex model function than a lower degree polynomial

INTUITION: TAYLOR SERIES

- More formally, one model class is more complex than another if it contains more functions
- If we already know the function g that we want to approximate, we can use Taylor polynomials
  - For many functions g, we have $g(x) = \sum_{k=0}^{\infty} w_k x^k$
  - One way to approximate is as $g(x) \approx \sum_{k=0}^{d} w_k x^k$
- Higher degree polynomial gives a better approximation?

INTUITION: TAYLOR SERIES

- Taylor expansions of $\sin(x)$ about 0 for $d = 1,5,9$



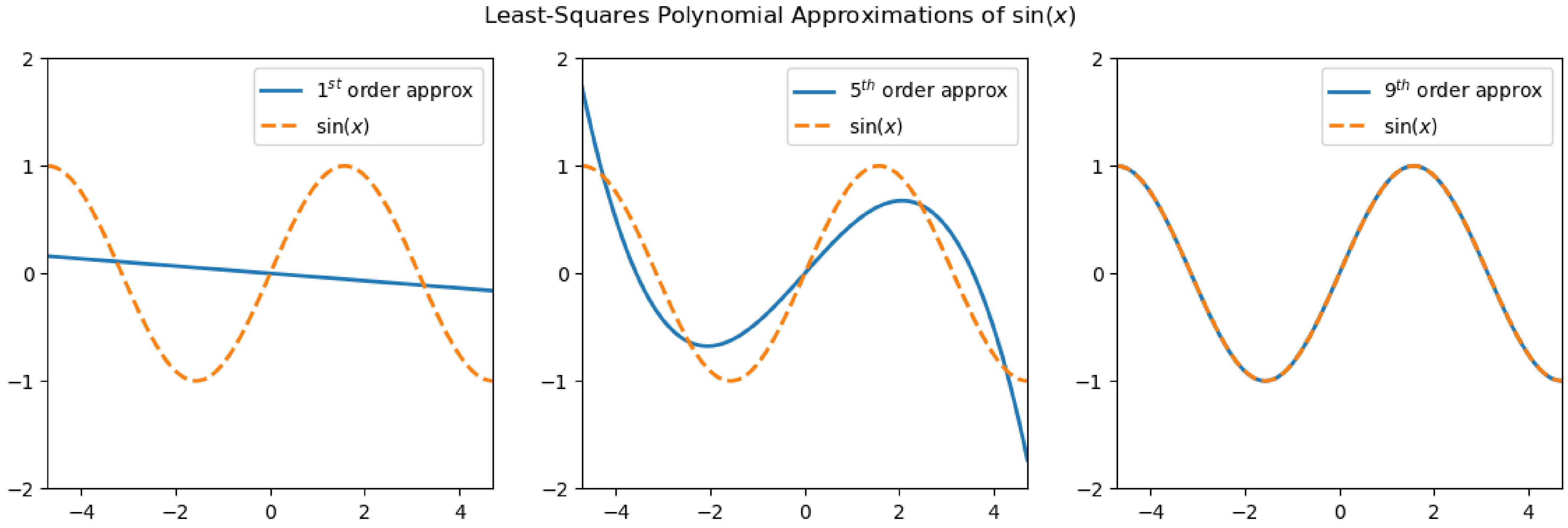
LEAST SQUARES FITTING

- Generally, we don't know the true function a priori
- Instead, we approximate it with a model function $f(x; w)$
- Rather than Taylor coefficients, we *really* want parameters $w^\star$ that minimize some loss function $J(w)$ on a dataset $\left\{\left(x^{(i)}, y^{(i)}\right)\right\}_{i=1}^{N}$, e.g. mean squared error:

$$w^\star = \operatorname*{argmin}_w J(w) = \operatorname*{argmin}_w \frac{1}{N} \sum_{i=1}^{N} \left(y^{(i)} - f\left(x^{(i)}; w\right)\right)^2$$

LEAST SQUARES FITTING

- Least squares polynomial fits of $\sin(x)$ for $d = 1,5,9$

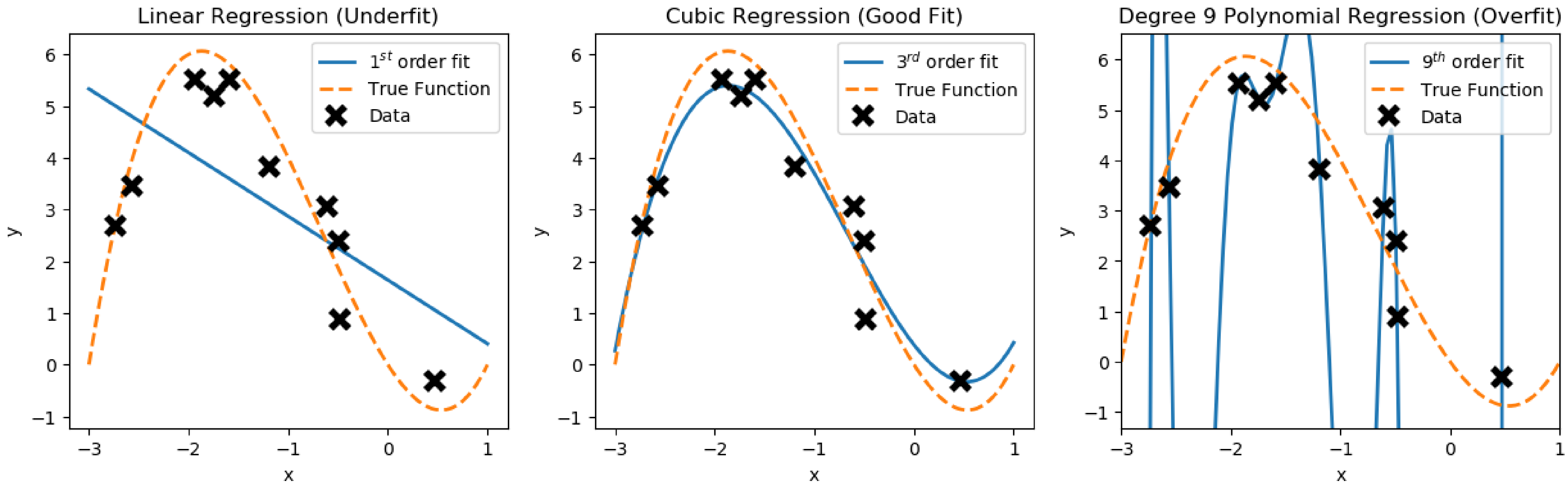


WHY SHOULD YOU CARE?

- So far, it seems like you should always prefer the more complex model, right?
- That's because these toy examples assume
  - We have a LOT of data
  - Our data is noiseless
  - Our model function behaves well between our data points
- In the real world, these assumptions are almost always false!

UNDERFITTING & OVERFITTING

- Fitting polynomials to noisy data from the orange function



UNDERFITTING & OVERFITTING

- Goal: learn a model that generalizes well to **unseen** test data
- **Underfitting**: model is too simple to learn any meaningful patterns in the data – high training error and high test error
- **Overfitting**: model is so complex that it doesn't generalize well to unseen data because it pays too much attention to the training data – low training error but high test error

UNDERFITTING & OVERFITTING

- Underfitting is easy to deal with – try using a more complex model class because it is more **expressive**
  - **Complexity** is roughly the "size" of the function space encoded by a model class (the set of all functions the class can represent)
  - **Expressiveness** is how well that model class can approximate the functions we are interested in
- If a more complex model class overfits, can we reduce its complexity while retaining its expressiveness?

REGULARIZATION

- If we make certain structural assumptions about the model we want to learn, we can do just this!
- These assumptions are called **regularizers**
- Most commonly, we minimize an **augmented loss function**

$$\tilde{J}(w) = J(w) + \lambda R(w)$$

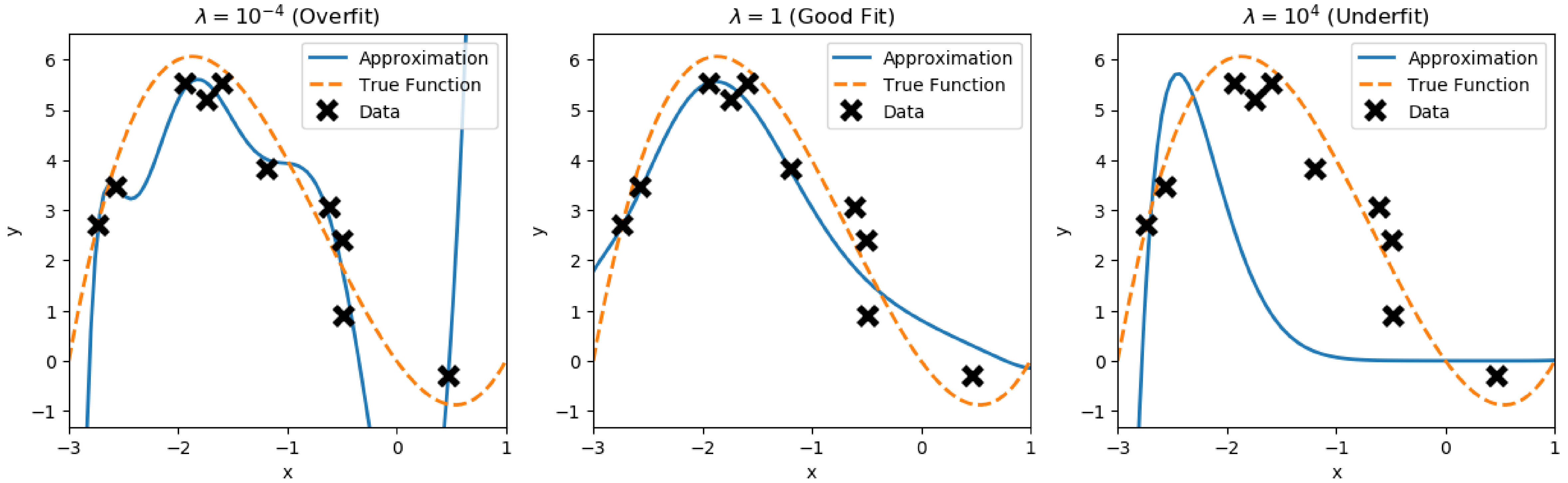
- $J(w)$ is the original loss function, λ is the regularization strength, and $R(w)$ is a regularization term

L_2 WEIGHT DECAY

- In L_2 weight decay regularization, $R(w) = w^T w = \sum_{k=1}^{d} w_k^2$
- Minimizing $\tilde{J}(w) = J(w) + \lambda w^T w$
  - Balances the goals of minimizing the loss $J(w)$ and finding a set of weights w that are small in magnitude
  - High λ means we care more about small weights, while low λ means we care more about a low (un-augmented) loss
- Intuitively, small weights w → smoother function (no huge oscillations like the 9th degree polynomial we overfit)

L_2 WEIGHT DECAY

- Regularizing a degree 9 polynomial fit with L_2 weight decay



RETURNING TO NEURAL NETS

- All of the intuition we've built for polynomials is also valid for neural nets!
- The complexity of a deep neural net is related (roughly) to the number of learned parameters and the number of layers
- More complex neural nets, i.e. **deeper** (more layers) and/or **wider** (more hidden units) are much more likely to overfit to the training data.

RETURNING TO NEURAL NETS

- L_2 weight decay helps us learn smoother neural nets by encouraging learned weights to be smaller.
- To incorporate L_2 weight decay, just do stochastic gradient descent on the augmented loss function

$$\tilde{J}\left(\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(L)}\right) = J\left(\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(L)}\right) + \lambda \sum_{i,j,\ell} \mathbf{W}_{ij}^{(\ell)^2}$$

$$\nabla_{\mathbf{W}^{(\ell)}}[\tilde{J}] = \nabla_{\mathbf{W}^{(\ell)}}[J] + 2\lambda \mathbf{W}^{(\ell)}$$

NEURAL NETS AND IMAGE DATA

- Let's now consider the special case of doing machine learning on image data with neural nets
- As we've studied them so far, neural nets model relationships between every single pair of pixels
- However, in any image, the color and intensity of neighboring pixels are much more strongly correlated than those of faraway pixels, i.e. images have **local structure**

NEURAL NETS AND IMAGE DATA

- Images are also **translation invariant**
  - A face is still a face, regardless of whether it's in the top left of an image or the bottom right
- Can we encode these assumptions of local structure into a neural network as a regularizer?
- If we could, we would get models that learned something about our data set **as a collection of images**.

RECAP: CONVOLUTIONS

- Consider a c-by-h-by-w convolutional **kernel** or **filter** array $\mathbf{K}$ and a C-by-H-by-W array representing an **image** $\mathbf{X}$
- The convolution (technically cross-correlation) $\mathbf{Z} = \mathbf{K} \otimes \mathbf{X}$ is

$$\mathbf{Z}[i,j,k] = \sum_{\ell=0}^{c-1} \sum_{m=0}^{h-1} \sum_{n=0}^{w-1} \mathbf{K}[\ell,m,n]\, \mathbf{X}[i+\ell, j+m, k+n]$$

- There are multiple ways to deal with boundary conditions; for now, ignore any indices that are out of bounds

RECAP: CONVOLUTIONS ($c = 1$)

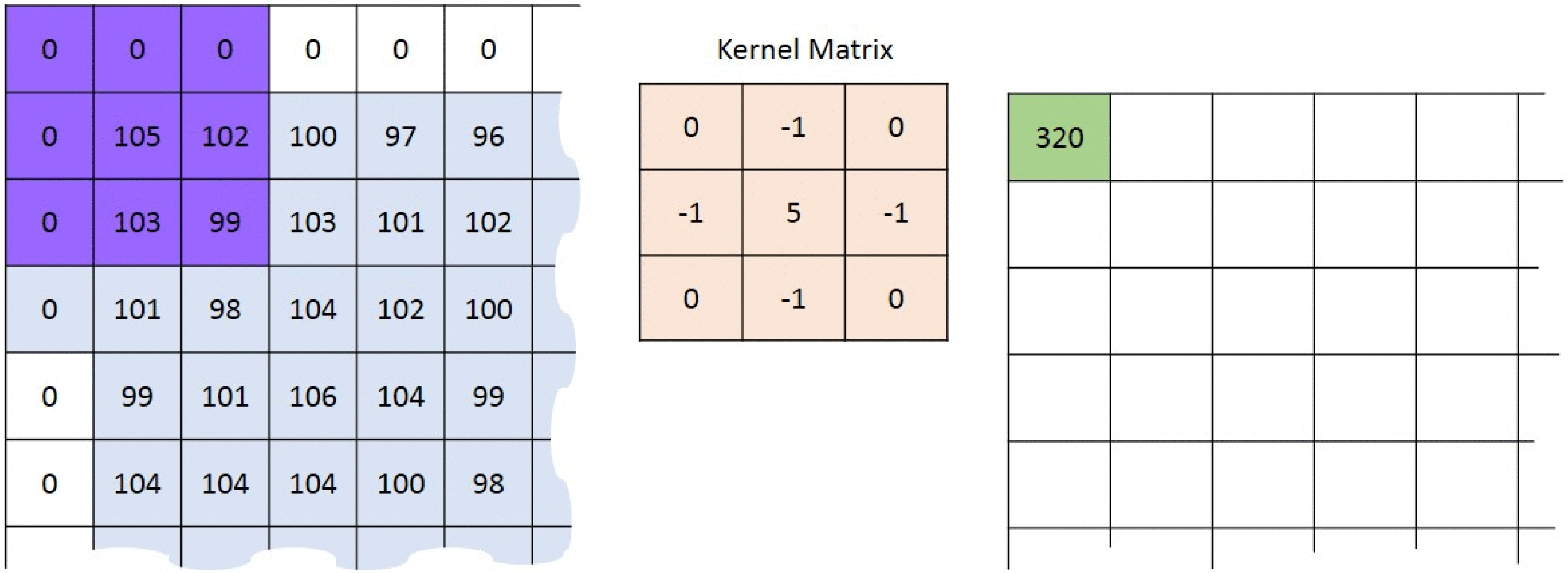


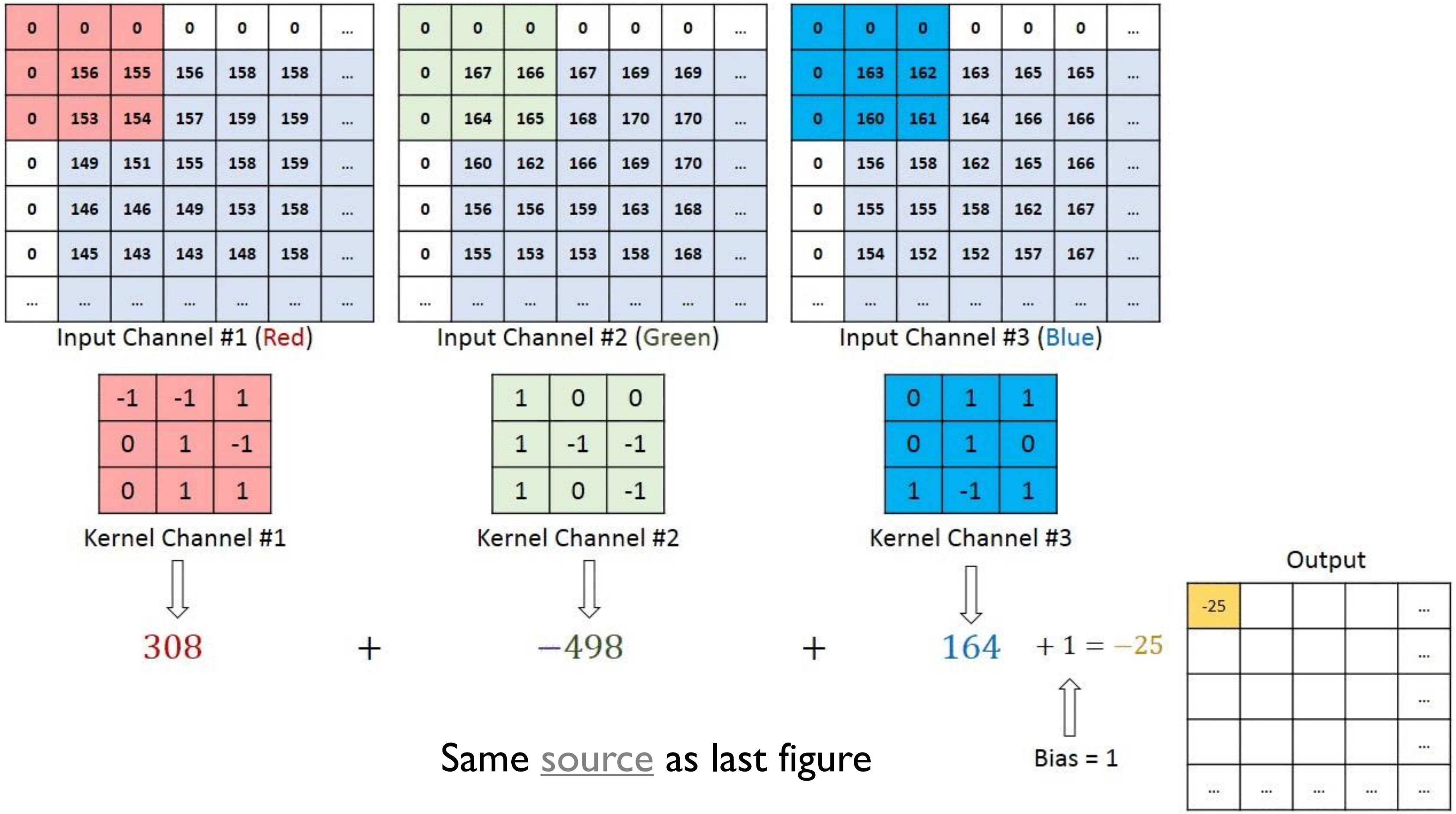
Image Matrix

$$0 * 0 + 0 * -1 + 0 * 0$$
$$+0 * -1 + 105 * 5 + 102 * -1$$
$$+0 * 0 + 103 * -1 + 99 * 0 \quad = 320$$

Output Matrix

http://machinelearninguru.com/computer_vision/basics/convolution/convolution_layer.html

RECAP: CONVOLUTIONS ($c = 3$)



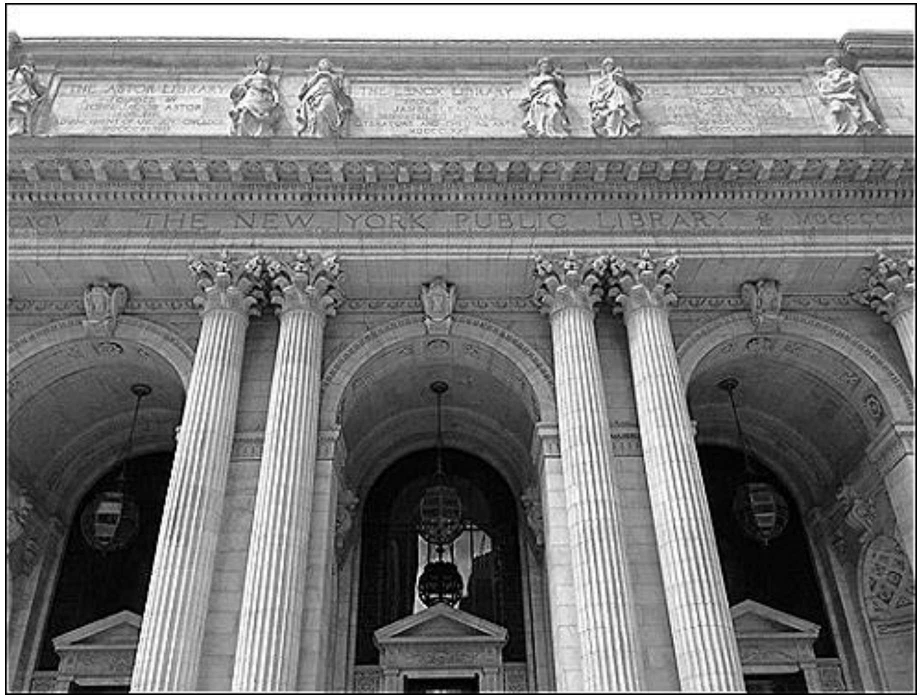
Same source as last figure

EXAMPLE CONVOLUTIONS WITH RELU

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

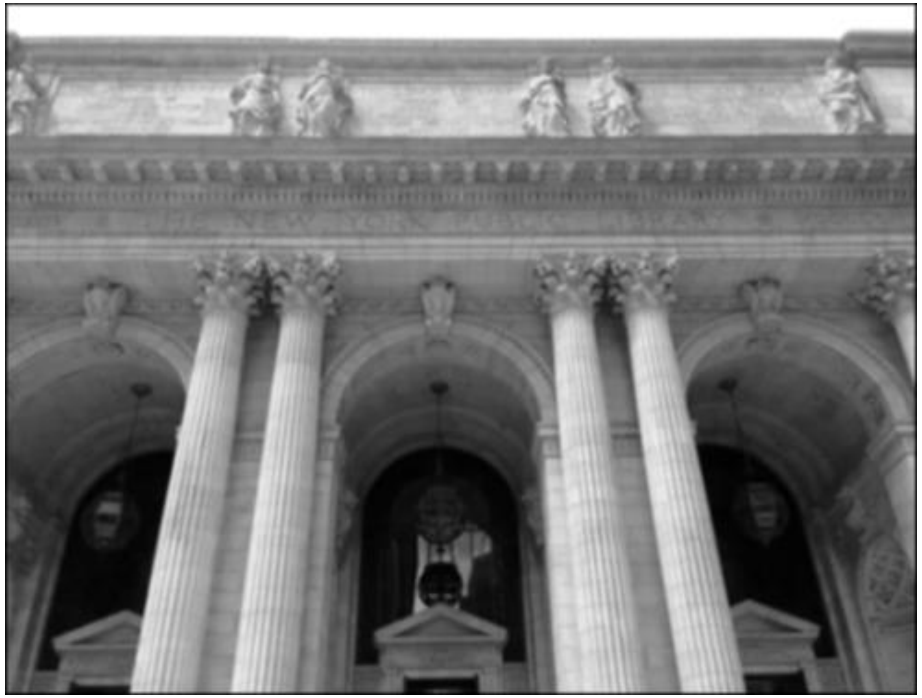
EXAMPLE CONVOLUTIONS WITH RELU

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$



EXAMPLE CONVOLUTIONS WITH RELU

$$\frac{1}{16}\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix}$$



EXAMPLE CONVOLUTIONS WITH RELU

$$\frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

EXAMPLE CONVOLUTIONS WITH RELU

$$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$$

EXAMPLE CONVOLUTIONS WITH RELU

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

ADVANTAGES OF CONVOLUTION

- By sliding the kernel along the image, we can extract the image's **local structure**!
  - Large objects (by blurring)
  - Sharp edges and outlines
- Since each output pixel of the convolution is highly local, the whole process is also **translation invariant**!
- Convolution is a **linear operation**, like matrix multiplication

CONVOLUTIONAL NEURAL NETS

- So far, the main downside of convolutions is that the coefficients of the kernels seem like magic numbers ☹
- But if we fit a 1D quadratic regression and get the model $f(x) = 0.382x^2 - 15.4x + 7$, then aren't the coefficients $0.382, -15.4,$ and 7 just magic numbers too?
- Idea: **learn convolutional kernels instead of matrices** to extract something meaningful from our image data, and then feed that into a dense neural network (with matrices)

CONVOLUTIONAL NEURAL NETS

- We can do this by creating a new kind of layer, and adding it to the front (closer to the input) of our neural network
- In the forward pass, we convolve our input $\mathbf{X}^{(\ell-1)}$ with a learned kernel $\mathbf{K}^{(\ell)}$, add a scalar bias $b^{(\ell)}$ to every element of $\mathbf{Z}^{(\ell)}$, and apply a nonlinearity θ to obtain our output $\mathbf{X}^{(\ell)}$

$$\mathbf{Z}^{(\ell)} = \mathbf{K}^{(\ell)} \otimes \mathbf{X}^{(\ell-1)} + b^{(\ell)}$$
$$\mathbf{X}^{(\ell)} = \theta\left(\mathbf{Z}^{(\ell)}\right)$$

CONVOLUTIONAL NEURAL NETS

- Note that we will actually be attempting to learn **multiple** (specifically c_ℓ) kernels of shape $c_{\ell-1} \times h_\ell \times w_\ell$ per layer ℓ!
  - $c_{\ell-1}$ is the number of channels in input $\mathbf{X}^{(\ell-1)}$, so convolving any individual kernel with $\mathbf{X}^{(\ell-1)}$ will yield 1 output channel
  - The output $\mathbf{X}^{(\ell)}$ is the result of *all* c_ℓ of these convolutions stacked on top of each other (1 output channel per kernel)
- If input $\mathbf{X}^{(\ell-1)}$ has shape $c_{\ell-1} \times H_\ell \times W_\ell$, then output $\mathbf{X}^{(\ell)}$ will have shape $c_\ell \times (H_\ell - h_\ell + 1) \times (W_\ell - w_\ell + 1)$

CONVOLUTIONAL NEURAL NETS

- We then feed the output $\mathbf{X}^{(\ell)}$ into the next layer as its input
  - If the next layer is a dense layer, we will re-shape $\mathbf{X}^{(\ell)}$ into a vector (instead of a multi-dimensional array)
  - If the next layer is also convolutional, we can pass $\mathbf{X}^{(\ell)}$ as is
- To actually learn good kernels that stage well with the layers we feed them into, we can just use the backpropagation algorithm to do stochastic gradient descent!

CONVOLUTIONAL BACKPROP

- Assume that we have $\Delta^{(\ell)} = \nabla_{\mathbf{X}^{(\ell)}}[J]$ (the gradient with respect to the input of the next layer, which is also the output of this layer)
- By the chain rule, for each kernel $\mathbf{K}^{(\ell)}$ at this layer ℓ,

$$\frac{\partial J}{\partial \mathbf{K}_{ijk}^{(\ell)}} = \sum_{a=1}^{c_\ell} \sum_{b=1}^{w_\ell} \sum_{c=1}^{h_\ell} \frac{\partial J}{\partial \mathbf{Z}_{abc}^{(\ell)}} \frac{\partial \mathbf{Z}_{abc}^{(\ell)}}{\partial \mathbf{K}_{ijk}^{(\ell)}}$$

CONVOLUTIONAL BACKPROP

- By the chain rule (again)

$$\frac{\partial J}{\partial \mathbf{Z}_{abc}^{(\ell)}} = \frac{\partial J}{\partial \mathbf{X}_{abc}^{(\ell)}} \frac{\partial \mathbf{X}_{abc}^{(\ell)}}{\partial \mathbf{Z}_{abc}^{(\ell)}} = \Delta_{abc}^{(\ell)} \theta' \left(\mathbf{Z}_{abc}^{(\ell)} \right)$$

- This gives us $\nabla_{\mathbf{Z}^{(\ell)}}[J]$, the gradient with respect to the output of the convolution
- We can find this with `cudnnActivationBackward()` (see Lecture 15) ☺

CONVOLUTIONAL BACKPROP

- If you give cuDNN the
  - Gradient with respect to the convolved output $\nabla_{\mathbf{Z}^{(\ell)}}[J]$
  - Input to the convolution $\mathbf{X}^{(\ell-1)}$
- cuDNN can compute each $\nabla_{\mathbf{K}^{(\ell)}}[J]$, the gradient of the loss with respect to each kernel $\mathbf{K}^{(\ell)}$ (Lecture 17) ☺
- With the $\nabla_{\mathbf{K}^{(\ell)}}[J]$'s computed, we can do gradient descent!

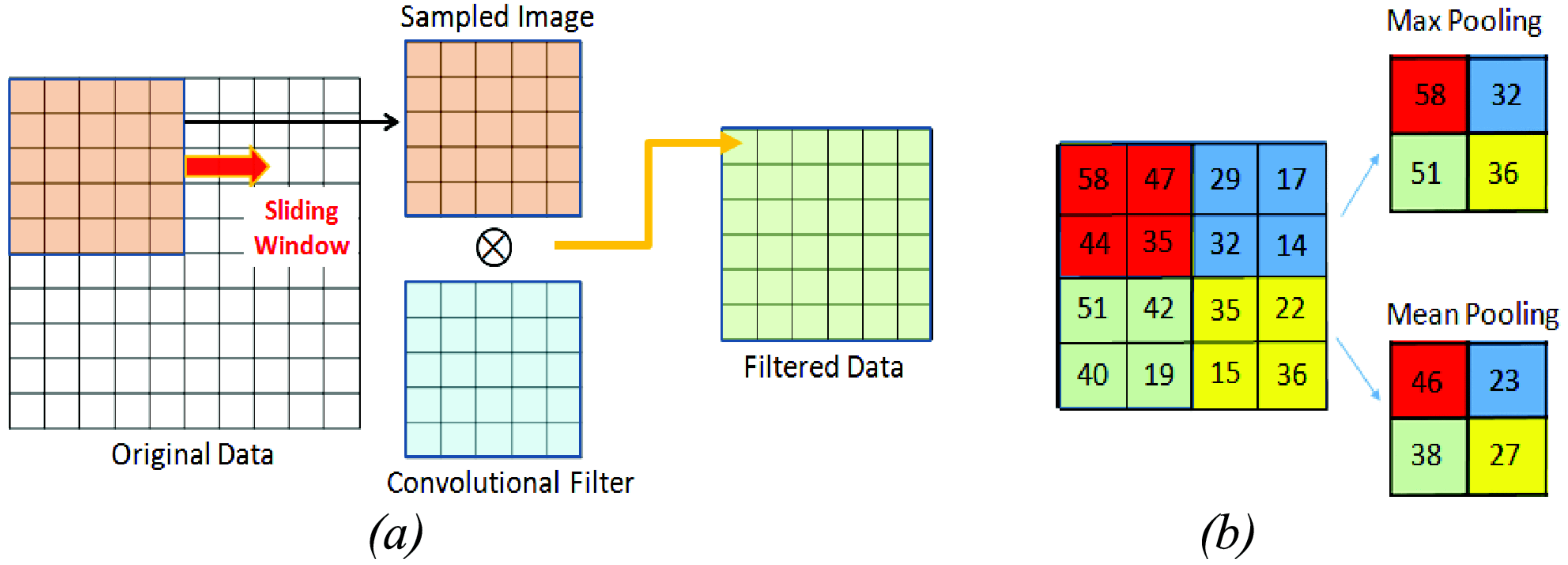
CONVOLUTIONAL BACKPROP

- All that remains is for us to find the gradient with respect to the input to this layer $\Delta^{(\ell-1)} = \nabla_{\mathbf{X}^{(\ell-1)}}[J]$
  - This is also the gradient with respect to the output of the next layer, and will be used to *continue* doing backpropagation.
- Again, cuDNN has a function for it (Lecture 17)
  - You need to provide it the kernels $\mathbf{K}^{(\ell)}$ and the gradient with respect to the output $\Delta^{(\ell)} = \nabla_{\mathbf{X}^{(\ell)}}[J]$ (like a dense neural net)

POOLING LAYERS

- After each convolutional layer, it is common to add a **pooling layer** to down-sample the input
- Most commonly, one would take every non-overlapping $n \times n$ window of a convolved output, and replace each window with a single pixel whose intensity is either
  - The maximum intensity found in that $n \times n$ window
  - The mean intensity of the pixels in that $n \times n$ window

EXAMPLE OF 2 × 2 POOLING



http://ieeexplore.ieee.org/document/7590035/all-figures

POOLING LAYERS

- Motivation: convolution compresses the amount of information in the image spatially
  - Blur → nearby pixels are more similar
  - Edge → “important” pixels are brighter than their surroundings
- Why not use that compression to reduce dimensionality?
- Forward and backwards propagation for pooling layers are fairly straightforward, and cuDNN can do both (Lecture 17)

WHY BOTHER?

- Consider the MNIST dataset of handwritten digits
  - Each image is 28×28 pixels → 784 input dimensions, and it can be one of 10 output classes
  - If we want to train even a linear classifier (not even a neural net), we would need $(784 + 1) \times 10 = 7850$ parameters
  - We're also modeling relationships between *every* pair of pixels; most of the relationships we learn probably aren't meaningful

CONV NETS ARE BETTER

- Let's instead consider the following convolutional net:
  - Layer 1: Twenty $(1 \times 5 \times 5)$ kernels
  - Layer 2: 2×2 pooling
  - Layer 3: Five $(20 \times 3 \times 3)$ kernels
  - Layer 4: 2×2 pooling
  - Layer 5: Dense layer with 50 hidden units
  - Layer 6: Dense layer with 10 output units

CONV NETS ARE BETTER

- Input shape $(1 \times 28 \times 28)$ (MNIST image)
- Twenty $(1 \times 5 \times 5)$ kernels
  - $20 \times \big((1 \times 5 \times 5) + 1\big) = 520$ parameters
  - Output shape $(20 \times 24 \times 24)$
- 2×2 pooling
  - Output shape $(20 \times 12 \times 12)$

CONV NETS ARE BETTER

- Input shape $(20 \times 12 \times 12)$ (conv 1 + pool 1)
- Five $(20 \times 3 \times 3)$ kernels
  - $5 \times \big((20 \times 3 \times 3) + 1\big) = 905$ parameters
  - Output shape $(5 \times 10 \times 10)$
- 2×2 pooling
  - Output shape $(5 \times 5 \times 5)$

CONV NETS ARE BETTER

- Input shape $(5 \times 5 \times 5)$ (conv 2 + pool 2)
- Flatten into a 125-dimensional vector
- Dense layer with 50 hidden units
  - $50 \times (125 + 1) = 6300$ parameters
  - Output is a 50-dimensional vector
- Dense layer with 10 output units
  - $10 \times (50 + 1) = 510$ parameters

CONV NETS ARE BETTER

- This gives us a total of $520 + 905 + 6300 + 510 = 8235$ parameters, similar to the vanilla linear classifier's 7850
- However, with the same number of parameters, this model
  - Learns something more meaningful about image structure
  - Achieves a significantly better accuracy on unseen data
- We've effectively regularized the neural net to perform well on image data! HW6: implement it and see for yourself.