CS 179: LECTURE 17

CONVOLUTIONAL NETS IN CUDNN

LAST TIME

- Motivation for convolutional neural nets
- Forward and backwards propagation algorithms for convolutional neural nets (at a high level)
- Down-sampling data using pooling operations
- Foreshadowing to how we will use cuDNN to do it

TODAY

- Understanding cuDNN's internal representations for convolutions and pooling objects
- Implementing convolutional nets using cuDNN

MORE SPECIFICALLY …

- Today, we will discuss how to use cuDNN SW to
  - Perform convolutions (can use FFT or straight computation)
  - Backpropagate gradients with respect to convolutions
  - Perform "pooling" operations for dimensionality reduction and backpropagate their gradients
- Much of this material can help HW6 -- should be a good additional resource in addition to the NVIDIA docs.

SW, REPRESENTING CONVOLUTIONS

- For tensors and their descriptors, we now also have `cudnnFilterDescriptor_t` (to describe a conv kernel/filter) and `cudnnConvolutionDescriptor_t` (to describe an actual convolution)
- We also have a `cudnnPoolingDescriptor_t` to represent a "pooling" operation (max pool, mean pool, etc.)
  - See https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/
- These have their own constructors, accessors, mutators, and destructors

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 ($c = 1$)

0	0	0	0	0	0
0	105	102	100	97	96
0	103	99	103	101	102
0	101	98	104	102	100
0	99	101	106	104	99
0	104	104	104	100	98

Image Matrix

Kernel Matrix

0	-1	0
-1	5	-1
0	-1	0

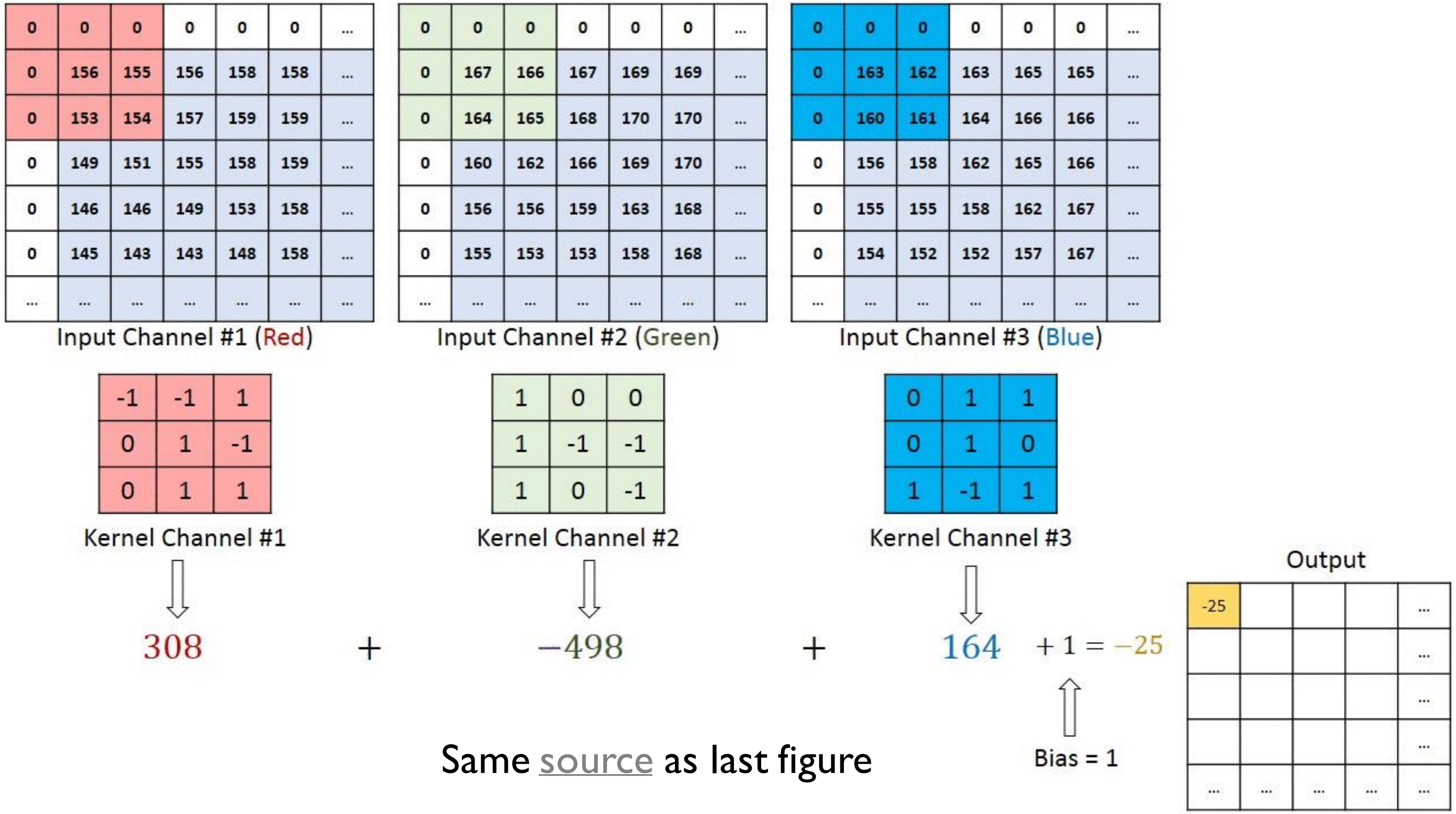
$$0 * 0 + 0 * -1 + 0 * 0 \\ +0 * -1 + 105 * 5 + 102 * -1 \\ +0 * 0 + 103 * -1 + 99 * 0 \quad = 320$$

320				

Output Matrix

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

RECAP: CONVOLUTIONS ($c = 3$)



Same source as last figure

RELATION OF CONVOLUTION TO FOURIER TRANSFORM

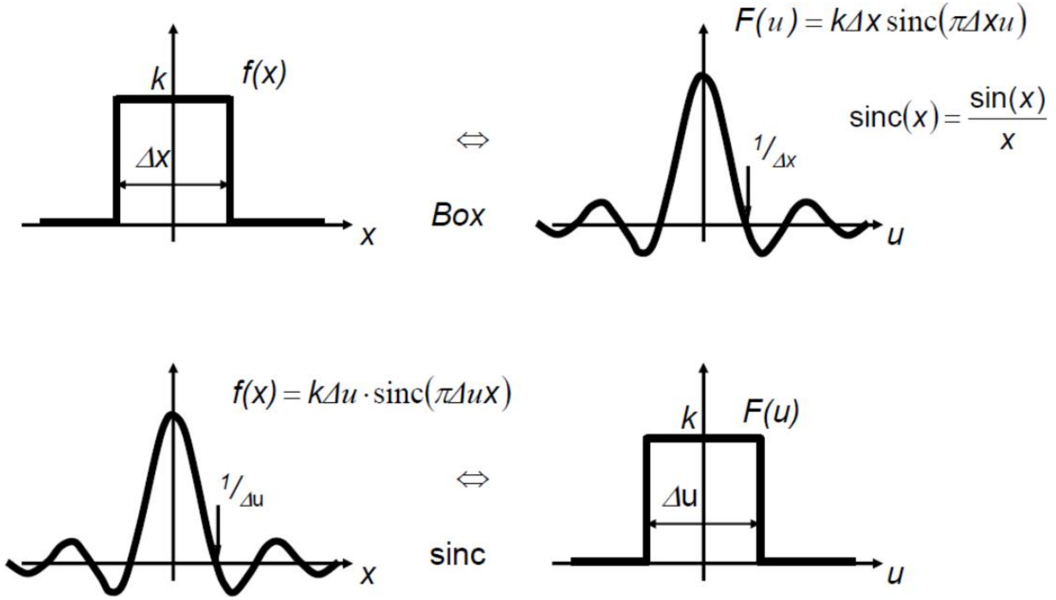
- Convolution of g and h in continuous domain defined by:
  - $$r(x) = \{g * h\}(x) \triangleq \int_{-\infty}^{\infty} g(\tau)h(x-\tau)\,d\tau = \int_{-\infty}^{\infty} g(x-\tau)h(\tau)\,d\tau$$
- Convolution Theorem
  - Fourier transform of convolution of two functions (or signals) is also the pointwise product of their Fourier transforms.
  - Convolution in one domain (e.g., time domain) equals point-wise multiplication in the other domain (e.g., frequency domain). On GPU we can compute convolutions using the inverse Fourier Transform! (Or not. It will be a choice!) Other choices too.

$$r(x) = \{g * h\}(x) = \mathcal{F}^{-1}\{G \cdot H\}$$

- https://en.wikipedia.org/wiki/Convolution_theorem

FFT, CHOICE FOR COMPUTING CONVOLUTIONS ON GPUS?

- Box Filter in one domain (frequency) is sinc function in other domain (time/space domain)
- Can cut off high frequencies by multiplying by box-like filter in frequency domain, which is same as convolving with sinc-like filter in space domain (for antialiasing on CPUs).
- FFT on GPUs gives other efficient choices for computing convolutions, for use in NNs
- See https://jeheonpark93.medium.com/image-processing-convolution-kernel-aliasing-3dc18ff5e373



$$\{g * h\} = \mathcal{F}^{-1}\{\mathcal{F}\{g\} \cdot \mathcal{F}\{h\}\}$$

SW, CONVOLUTIONAL FILTERS

- `cudnnFilterDescriptor_t`
  - Allocate by calling `cudnnCreateFilterDescriptor(cudnnFilterDescriptor_t *filterDesc)`
  - Free by calling `cudnnDestroyFilterDescriptor(cudnnFilterDescriptor_t filterDesc)`
  - The filter itself is just an array of numbers on the device
    - We will be using 4D arrays to store filters
    - However, this is still just a long linear array, like everything else

CONVOLUTIONAL FILTERS

- `cudnnFilterDescriptor_t`
  - Set by calling `cudnnSetFilter4dDescriptor( cudnnFilterDescriptor_t filterDesc, cudnnDataType_t datatype, cudnnTensorFormat_t format, int k, int c, int h, int w)`
  - Use `TENSOR_FORMAT_NCHW` for `format` parameter
  - `k` = # of output channels, `c` = # of input channels

CONVOLUTIONAL FILTERS

- `cudnnFilterDescriptor_t`
  - **Get contents by calling** `cudnnGetFilter4dDescriptor(cudnnFilterDescriptor_t filterDesc, cudnnDataType_t *datatype, cudnnTensorFormat_t *format, int *k, int *c, int *h, int *w)`
  - As usual, this function returns by setting pointers to output parameters

DESCRIBING CONVOLUTIONS

- `cudnnConvolutionDescriptor_t`
  - Allocate with `cudnnCreateConvolutionDescriptor(cudnnConvolutionDescriptor_t *convDesc)`
  - Free with `cudnnDestroyConvolutionDescriptor(cudnnConvolutionDescriptor_t convDesc)`
  - For HW6 we will be considering 2D convolutions only

DESCRIBING CONVOLUTIONS

- `cudnnConvolutionDescriptor_t`
  - **Set with** `cudnnSetConvolution2dDescriptor(`

```
cudnnConvolutionDescriptor_t convDesc,
int pad_h,        int pad_w,
int u,            int v,
int dilation_h, int dilation_w,
cudnnConvolutionMode_t mode,
cudnnDataType_t computeType)
```

DESCRIBING CONVOLUTIONS

- `cudnnConvolutionDescriptor_t`
  - `pad_h` and `pad_w` are respectively the number of rows and columns of zeros to pad the input with – use 0 for both
  - `u` and `v` are respectively the vertical and horizontal stride of the convolution (to downsample w/o pooling) – use 1 for both
  - Use 1 for both `dilation_h` and `dilation_w` (roughly a stretch factor for filters, but beyond the scope of this class)

DESCRIBING CONVOLUTIONS

- `cudnnConvolutionDescriptor_t`
  - `cudnnConvolutionMode_t` is an enum saying whether to do a convolution or cross-correlation. For this set, use `CUDNN_CONVOLUTION` for the `mode` argument.
  - `cudnnDataType_t` is an enum indicating the kind of data being used (float, double, int, long int, etc.). For this set, use `CUDNN_DATA_FLOAT` for the `computeType` argument.

DESCRIBING CONVOLUTIONS

- `cudnnConvolutionDescriptor_t`
  - **Get with** `cudnnGetConvolution2dDescriptor(`

```
cudnnConvolutionDescriptor_t convDesc,
int *pad_h,        int *pad_w,
int *u,            int *v,
int *dilation_h, int *dilation_w,
cudnnConvolutionMode_t *mode,
cudnnDataType_t *computeType)
```

DESCRIBING CONVOLUTIONS

- Given descriptors for an input and the filter we want to convolve it with, we can get the shape of the output via

```
cudnnGetConvolution2dForwardOutputDim(
cudnnConvolutionDescriptor_t convDesc,
cudnnTensorDescriptor_t inputTensorDesc,
cudnnFilterDescriptor_t filterDesc,
int *n, int *c, int *h, int *w)
```

- As usual, `n`, `c`, `h`, and `w` are set by reference as outputs

USING THESE IN A CONV NET

- All of cuDNN’s functions for forward and backward passes in conv nets will extensively use these descriptor types
- This is why we are establishing them now
- One more aside before discussing the actual functions for doing the forward and backward passes…

CONVOLUTION ALGORITHMS

- Many ways to perform convolutions on GPUs!
  - Do it explicitly
  - Turn it into a matrix multiplication
  - Can use FFT to transform into frequency domain, multiply pointwise, and inverse FFT back
- cuDNN lets you choose the algorithm you want to use for all operations in the forward and backward passes

CONVOLUTION ALGORITHMS

- Different algorithms are better suited for different situations!
  - Most important factor: amount of global memory available for intermediate computations (workspace)
- Tradeoff b/w time and space complexity – faster algorithms tend to need more space for intermediate computations
- cuDNN lets you specify preferences, and it gives you an algorithm that best matches your preferences

CONVOLUTION ALGORITHMS

- The choice of algorithm is represented via the enums cudnnConvolution<type>Preference_t and cudnnConvolution<type>Algo_t, and cudnnConvolution<type>AlgoPerf_t, where <type> is one of Fwd, BwdFilter, and BwdData
- Feel free to look at NVIDIA docs for these types and related functions, but we will be handling them for you in HW6

FORWARD PASS: CONVOLUTION

- The forward pass for a conv layer with input $\mathbf{X}^{(\ell-1)}$, filter $\mathbf{K}^{(\ell)}$, and bias $b^{(\ell)}$ is $\mathbf{Z}^{(\ell)} = \mathbf{K}^{(\ell)} \otimes \mathbf{X}^{(\ell-1)} + b^{(\ell)}$
- In HW6, we will give you code that deals with the bias term
- Your job will be to perform the convolution $\mathbf{K}^{(\ell)} \otimes \mathbf{X}^{(\ell-1)}$ using `cudnnConvolutionForward()` – see next slide for a description of how to call this function

FORWARD PASS: CONVOLUTION

- ```
  cudnnConvolutionForward(
  cudnnHandle_t handle,
  void *alpha,
  cudnnTensorDescriptor_t xDesc, void *x,
  cudnnFilterDescriptor_t kDesc, void *k,
  cudnnConvolutionDescriptor_t convDesc,
  cudnnConvolutionFwdAlgo_t algo,
  void *workSpace, size_t workSpaceBytes,
  void *beta,
  cudnnTensorDescriptor_t zDesc, void *z)
  ```
  ```

FORWARD PASS: CONVOLUTION

- This function sets the contents of the output tensor `z` to `alpha[0] * conv(k, x) + beta[0] * z`
- The convolution algorithm, workspace, and size of the workspace will be supplied to you in HW6 (unnecessary complication for you to consider for this set)
- With `alpha[0] = 1` and `beta[0] = 0`, this is exactly what you need to call!

BACKWARD PASS: CONVOLUTION

- With the neural net architecture given, we will have:
  - The output of the convolution $\mathbf{Z}^{(\ell)} = \mathbf{K}^{(\ell)} \otimes \mathbf{X}^{(\ell-1)} + b^{(\ell)}$
  - The gradient $\nabla_{\mathbf{Z}^{(\ell)}}[J]$ with respect to the output of the convolution (propagated backwards from the next layer)
- We want to find the gradients with respect to:
  - The filter $\mathbf{K}^{(\ell)}$ and the bias $b^{(\ell)}$ to do gradient descent
  - The input data $\mathbf{X}^{(\ell-1)}$ to propagate backwards

BACKWARD PASS: CONVOLUTION

- Key to argument names
  - `x` is the input data $\mathbf{X}^{(\ell-1)}$
  - `k` is the filter $\mathbf{K}^{(\ell-1)}$
  - `dz` is the gradient $\nabla_{\mathbf{Z}^{(\ell)}}[J]$ with respect to the output $\mathbf{Z}^{(\ell)}$
  - `dx` is the gradient $\nabla_{\mathbf{X}^{(\ell-1)}}[J]$ with respect to input data $\mathbf{X}^{(\ell-1)}$
  - `dk` is the gradient $\nabla_{\mathbf{K}^{(\ell)}}[J]$ with respect to the filter $\mathbf{K}^{(\ell)}$
  - `db` is the gradient $\nabla_{b^{(\ell)}}[J]$ with respect to the bias $b^{(\ell)}$

BACKWARD PASS: CONVOLUTION

- Key to argument names
  - As always, the `alpha` and `beta` arguments are pointers to mixing parameters
  - If we are using a buffer `out` to accumulate the results of performing an operation `op` on an input buffer `in`, we have `out = alpha[0] * op(in) + beta[0] * out`

GRADIENT WRT BIAS

- ```
  cudnnConvolutionBackwardBias(
  cudnnHandle_t handle,
  void *alpha,
  cudnnTensorDescriptor_t dzDesc, void *dz,
  cudnnConvolutionDescriptor_t convDesc,
  void *beta,
  cudnnTensorDescriptor_t dbDesc, void *db)
  ```
- We will handle this for you in HW6
  ```

GRADIENT WRT FILTER

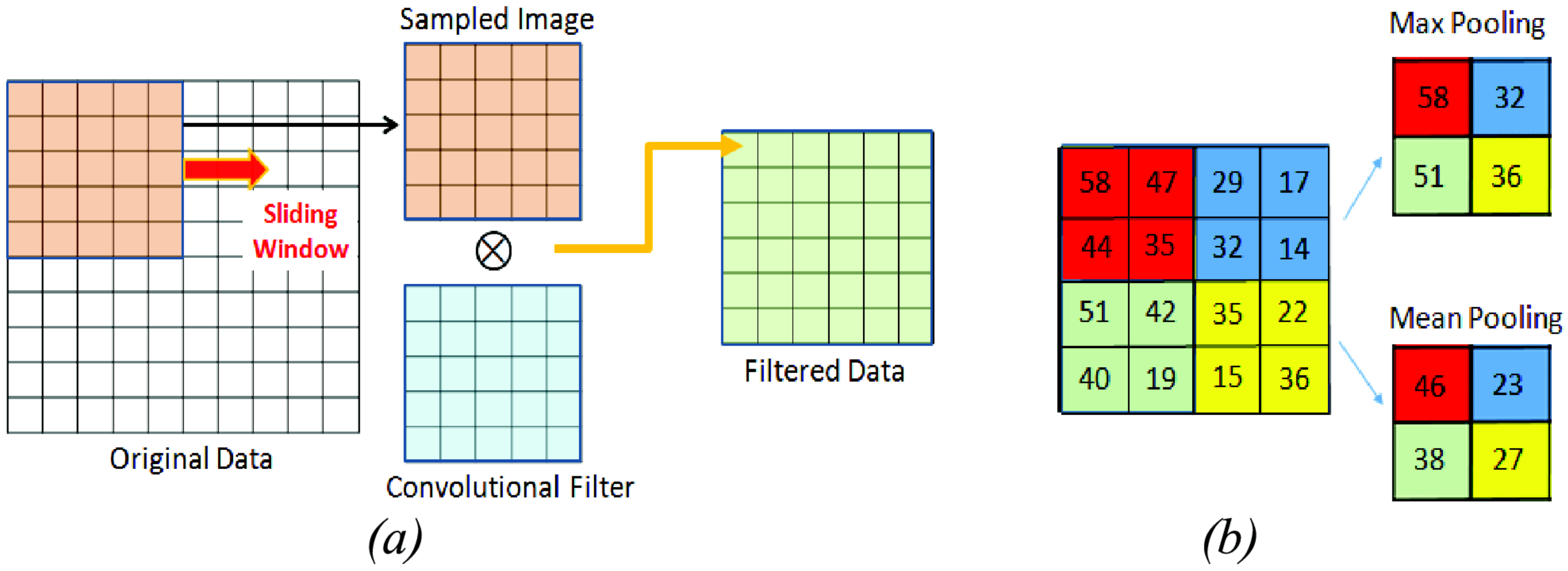
- ```
  cudnnConvolutionBackwardFilter(
  cudnnHandle_t handle,
  void *alpha,
  cudnnTensorDescriptor_t xDesc,  void *x,
  cudnnTensorDescriptor_t dzDesc, void *dz,
  cudnnConvolutionDescriptor_t convDesc,
  cudnnConvolutionBwdFilterAlgo_t algo,
  void *workSpace, size_t workSpaceBytes,
  void *beta,
  cudnnFilterDescriptor_t dkDesc, void *dk)
  ```
  ```

GRADIENT WRT INPUT DATA

- ```
  cudnnConvolutionBackwardData(
  cudnnHandle_t handle,
  void *alpha,
  cudnnFilterDescriptor_t kDesc,  void *k,
  cudnnTensorDescriptor_t dzDesc, void *dz,
  cudnnConvolutionDescriptor_t convDesc,
  cudnnConvolutionBwdDataAlgo_t algo,
  void *workSpace, size_t workSpaceBytes,
  void *beta,
  cudnnTensorDescriptor_t dxDesc, void *dx)
  ```
  ```

POOLING OPERATIONS

- Reminder: pooling lets us down-sample or focus on parts of images to reduce dimensionality while preserving information



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

POOLING OPERATIONS

- `cudnnPoolingDescriptor_t`
  - Allocate with `cudnnCreatePoolingDescriptor( cudnnPoolingDescriptor_t *poolingDesc)`
  - Free with `cudnnDestroyPoolingDescriptor( cudnnPoolingDescriptor_t poolingDesc)`
  - We will only be using 2D pooling operations in HW6

POOLING OPERATIONS

- `cudnnPoolingDescriptor_t`
  - Set with `cudnnSetPooling2dDescriptor(`

```
cudnnPoolingDescriptor_t poolingDesc,
cudnnPoolingMode_t poolingMode,
cudnnNanPropagation_t nanProp,
int windowHeight,    int windowWidth,
int verticalPad,     int horizontalPad,
int verticalStride, int horizontalStride)
```

POOLING OPERATIONS

- `cudnnPoolingDescriptor_t`
  - `cudnnPoolingMode_t` is an enum specifying the kind of pooling to do, i.e. max (`CUDNN_POOLING_MAX`) or average (`CUDNN_POOLING_AVERAGE_COUNT_INCLUDE_PADDING` or `CUDNN_POOLING_AVERAGE_COUNT_EXCLUDE_PADDING`)
  - For `nanProp`, use `CUDNN_PROPAGATE_NAN`
  - Use 0 for horizontal and vertical padding
  - Make the strides equal to the window dimensions

POOLING OPERATIONS

- `cudnnPoolingDescriptor_t`
  - **Get with** `cudnnGetPooling2dDescriptor(`

```
cudnnPoolingDescriptor_t *poolingDesc,
cudnnPoolingMode_t *poolingMode,
cudnnNanPropagation_t *nanProp,
int *windowHeight,    int *windowWidth,
int *verticalPad,     int *horizontalPad,
int *verticalStride, int *horizontalStride)
```

POOLING OPERATIONS

- We can get the output shape of a pooling operation on some input using the function
  - ```
    cudnnGetPooling2dForwardOutputDim(
    cudnnPoolingDescriptor_t poolingDesc,
    cudnnTensorDescriptor_t  inputDesc,
    int *n, int *c, int *h, int *w)
    ```
  - `n`, `c`, `h`, and `w` are output parameters to be set by reference
    ```

POOLING OPERATIONS

- To perform a pooling operation in the forward direction, use
  - ```
    cudnnPoolingForward(
    cudnnHandle_t handle,
    cudnnPoolingDescriptor_t poolingDesc,
    void *alpha,
    cudnnTensorDescriptor_t xDesc,  void *x,
    void *beta,
    cudnnTensorDescriptor_t zDesc, void *z)
    ```
    ```

POOLING OPERATIONS

- To differentiate with respect to a pooling operation, use
  - ```
    cudnnPoolingBackward(
    cudnnHandle_t handle,
    cudnnPoolingDescriptor_t poolingDesc,
    void *alpha,
    cudnnTensorDescriptor_t zDesc,  void *z,
    cudnnTensorDescriptor_t dzDesc, void *dz,
    void *beta,
    cudnnTensorDescriptor_t dxDesc, void *dx)
    ```
    ```

POOLING OPERATIONS

- In the previous slides, `x` is the input to the pooling operation, `dx` is its gradient, `z` is the output of the pooling operation, and `dz` is its gradient
- `alpha` and `beta` are pointers to mixing parameters as usual
- In all cases, the last buffer given as an argument is the output array

SUMMARY

- Today, we discussed how to use cuDNN to
  - Perform convolutions
  - Backpropagate gradients with respect to convolutions
  - Perform pooling operations and backpropagate their gradients
- For HW6, these slides should be a good alternative reference to the NVIDIA docs.