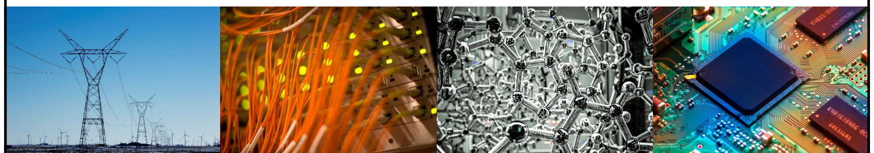
ECE408/CS483/CSE408 Spring 2020
Convolutional Neural Networks



Carl Pearson
pearson@illinois.edu

1

Objective

- To learn to implement the different types of layers in a Convolutional Neural Network (CNN)

2

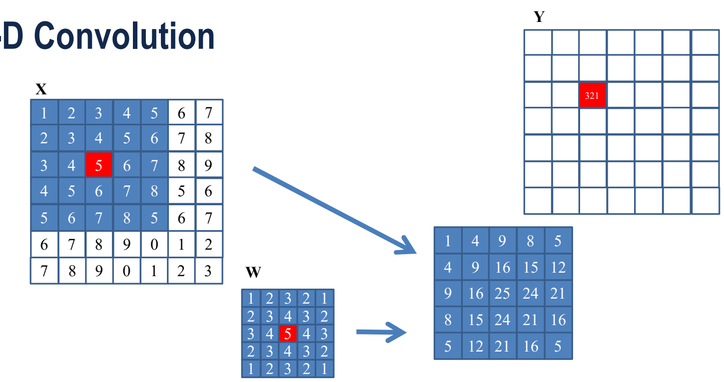
MLP (Multi-Layer Perceptron) for an Image

Consider a 250 x 250 image…
- input: 2D image treated as 1D vector
- Fully connected layer is huge:
  - 62,500 (250^2) weights per node!
  - Comparable number of nodes gives ~4B weights total!
- Need >1 hidden layer? Bigger images?
- Too much computation, and too much memory.

Traditional feature detection in image processing uses
- Filters → Convolution kernels
- Can we use them in neural networks?

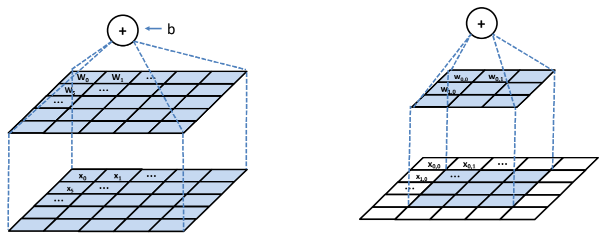
3

2-D Convolution



4

Convolution vs Fully-Connected (Weight Sharing)



Convolution Naturally Supports Varying Input Sizes

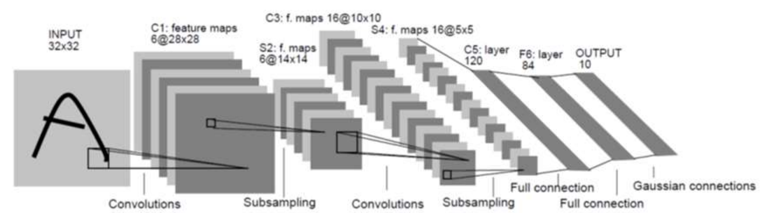
- As discussed so far,
  - perceptron layers have fixed structure, so
  - number of inputs / outputs is fixed.
- Convolution enables variably-sized inputs (observations of the same kind of thing)
  - Audio recording of different lengths
  - Image with more/fewer pixels

Example Convolution Inputs

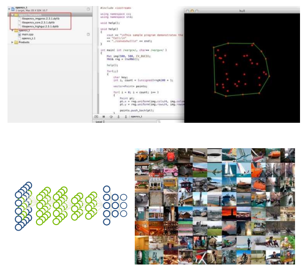
	Single-channel	Multi-channel
1D	audio waveform	Skeleton animation data: 1-D joint angles for each joint
2D	Fourier-transformed audio data Convolve over frequency axis: invariant to frequency shifts Convolve over time axis: invariant to shifts in time	Color image data: 2D data for R,G,B channels
3D	Volumetric data (example: medical imaging)	Color video: 2D data across 1D time for R,G,B channels

Deeplearningbook.org, ch 9, p 355

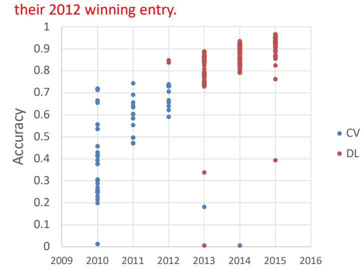
LeNet-5:CNN for hand-written digit recognition



Deep Learning Impact in Computer Vision



The Toronto team used GPUs and trained on 1.2M images in their 2012 winning entry.



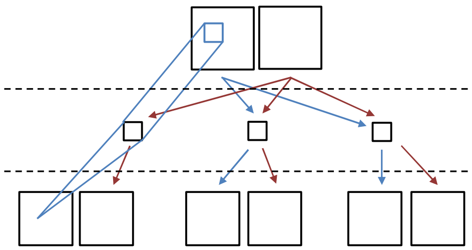
9

Anatomy of a Convolution Layer

Input features/channels
- A inputs ($N_1 \times N_2$)

Convolution Layer (or per channel)
- B convolution kernels ($K_1 \times K_2$)

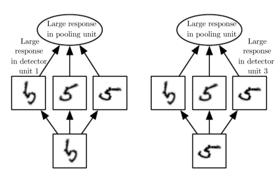
Output Features/channels
(or summed over channels)
- A × B outputs
  $(N_1 - K_1 + 1) \times (N_2 - K_2 + 1)$



10

2-D Pooling (Subsampling)

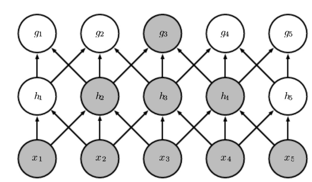
- A subsampling layer
  - Sometimes with bias and non-linearity built in
- Common types: max, average, L^2 norm, weighted average
- Helps make representation invariant to size scaling and small translations in the input



11

Why Convolution (1)

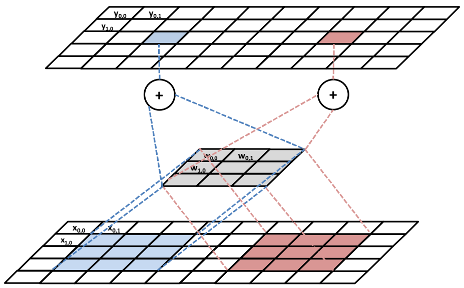
- Sparse interactions
  - Meaningful features in small spatial regions
  - Need fewer parameters (less storage, better statistical characteristics, faster training)
  - Need multiple layers for wide receptive field



12

Why Convolution (2)

- Parameter sharing
  - Kernel is reused when computing layer output
- Equivariant Representations
  - If input is translated, output is translated the same way
  - Map of where features appear in input



13

Convolution

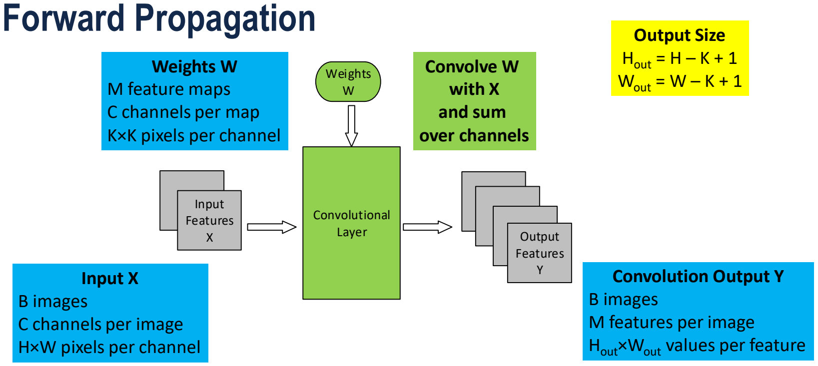
- 2-D Matrix
- $Y = W \otimes X$
- Kernel smaller than input: smaller receptive field
- Fewer Weights

Multi-Layer Percep.

- Vector
- $Y = w\,x + b$
- Maximum receptive field
- More weights

14

Forward Propagation



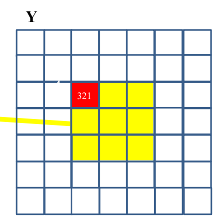
15

Outputs Must Use Full Mask/Kernel

X

1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	5	6
5	6	7	8	5	6	7
6	7	8	9	0	1	2
7	8	9	0	1	2	3

Compute only this part of Y.



W

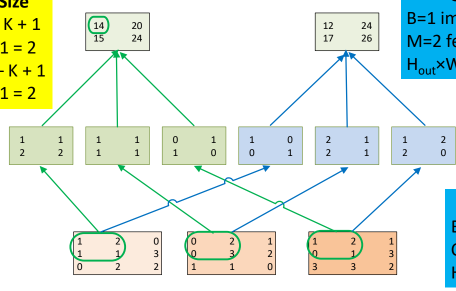
1	2	3	2	1
2	3	4	3	2
3	4	5	4	3
2	3	4	3	2
1	2	3	2	1

1	4	9	8	5
4	9	16	15	12
9	16	25	24	21
8	15	24	21	16
5	12	21	16	5

16

Example of the Forward Path of a Convolution Layer

Output Size
$H_{out} = H - K + 1$
$= 3 - 2 + 1 = 2$
$W_{out} = W - K + 1$
$= 3 - 2 + 1 = 2$



Convolution Output Y
B=1 image
M=2 features per image
$H_{out} \times W_{out}$=2×2 values per feature

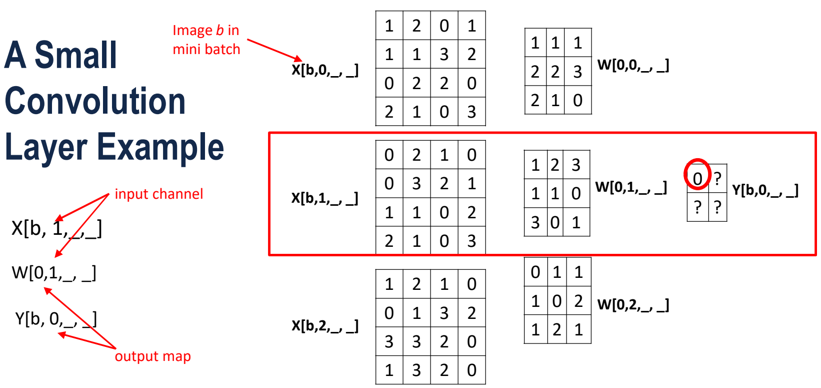
Weights W
M=2 feature maps
C=3 channels per map
K×K=2×2 pixels per channel

Input X
B=1 image
C=3 channels
H×W=3×3 pixels per channel

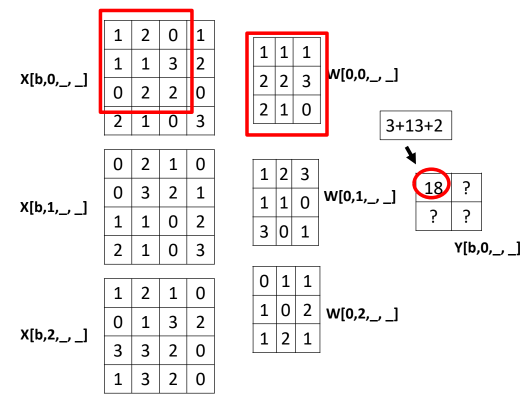
Sequential Code: Forward Convolutional Layer

```
void convLayer_forward(int B, int M, int C, int H, int W, int K, float* X, float* W, float* Y) {
  int H_out = H – K + 1;                     // calculate H_out, W_out
  int W_out = W – K + 1;

  for (int b = 0; b < B; ++b)                // for each image
    for(int m = 0;  m < M;  m++)             // for each output feature map
      for(int h = 0; h < H_out; h++)         // for each output value (two loops)
        for(int w = 0; w < W_out; w++) {
          Y[b, m, h, w] = 0.0f;              // initialize sum to 0
          for(int c = 0; c < C; c++)         // sum over all input channels
            for(int p = 0; p < K; p++)       // KxK filter
              for(int q = 0; q < K; q++)
                Y[b, m, h, w] += X[b, c, h + p, w + q] * W[m, c, p, q];
       }
}
```

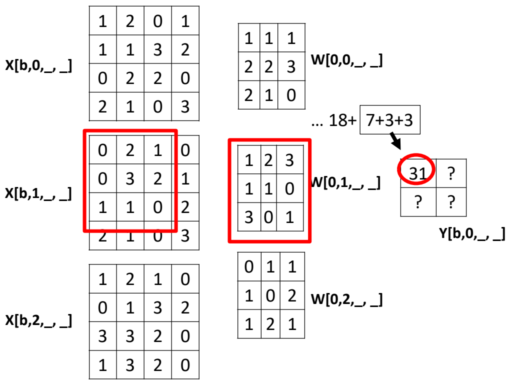

A Small Convolution Layer Example



A Small Convolution Layer Example c = 0

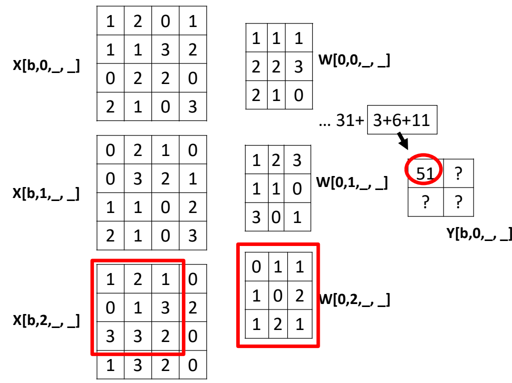


A Small Convolution Layer Example c = 1



21

A Small Convolution Layer Example c = 2



22

Parallelism in a Convolution Layer

Output feature maps can be calculated in parallel
- Usually a small number, not sufficient to fully utilize a GPU

All **output** feature map **pixels** can be calculated in parallel
- All rows can be done in parallel
- All pixels in each row can be done in parallel
- Large number but diminishes as we go into deeper layers

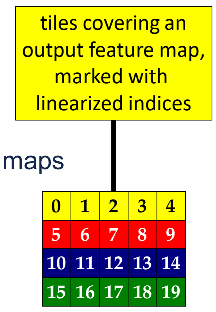
All **input feature maps** can be processed in parallel,
but need atomic operation or tree reduction (we'll learn later)

Different layers may demand different strategies.

23

Design of a Basic Kernel

- Each block computes
  - a tile of output pixels for one feature
  - TILE_WIDTH pixels in each dimension
- Grid's X dimension maps to M output feature maps
- Grid's Y dimension maps to the tiles in the output feature maps (linearized order).
- (Grid's Z dimension is used for images in batch, which we omit from slides.)



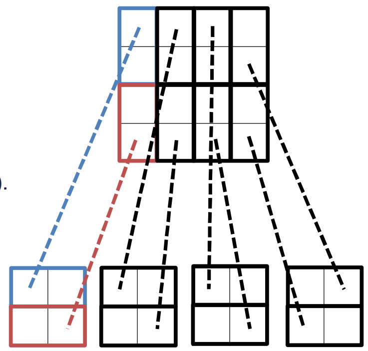
24

A Small Example

Assume
- M = 4 (4 output feature maps),
- thus 4 blocks in the X dimension, and
- W_out = H_out = 8 (8x8 output features).

If TILE_WIDTH = 4,
we also need 4 blocks in the Y dimension:
- for each output feature,
- top two blocks in each column calculates the top row of tiles, and
- bottom two calculate the bottom row.

CUDA Grid and Threadblocks



Output Feature Maps and Tiles

25

Host Code for a Basic Kernel: CUDA Grid

Consider an output feature map:
- width is W_out, and
- height is H_out.
- Assume these are multiples of TILE_WIDTH.

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19

Let X_grid be the number of blocks needed in X dim (5 above).
Let Y_grid be the number of blocks needed in Y dim (4 above).

26

Host Code for a Basic Kernel: CUDA Grid

(Assuming W_out and H_out are multiples of TILE_WIDTH.)

```
#define TILE_WIDTH 16         // We will use 4 for small examples.
W_grid = W_out/TILE_WIDTH;    // number of horizontal tiles per output map
H_grid = H_out/TILE_WIDTH;    // number of vertical tiles per output map
Y = H_grid * W_grid;

dim3 blockDim(TILE_WIDTH, TILE_WIDTH, 1); // output tile for untiled code
dim3 gridDim(M, Y, 1);

ConvLayerForward_Kernel<<< gridDim, blockDim>>>(…);
```


27

Partial Kernel Code for a Convolution Layer

```
__global__ void ConvLayerForward_Basic_Kernel
    (int C, int W_grid, int K, float* X, float* W, float* Y)
{
    int m = blockIdx.x;
    int h = (blockIdx.y / W_grid) * TILE_WIDTH + threadIdx.y;
    int w = (blockIdx.y % W_grid) * TILE_WIDTH + threadIdx.x;
    float acc = 0.0f;
    for (int c = 0;  c < C; c++) {       // sum over all input channels
       for (int p = 0; p < K; p++)       // loop over KxK filter
          for (int q = 0; q < K; q++)
             acc += X[c, h + p, w + q] * W[m, c, p, q];
    }
    Y[m, h, w] = acc;
}
```


28

Some Observations

Enough parallelism

- if the total number of pixels
- across all output feature maps is large
- (often the case for CNN layers)

Each input tile

- loaded M times (number of output features), so
- **not efficient in global memory bandwidth,**
- but block scheduling in X dimension should give cache benefits.

Subsampling (Pooling) by Scale N

Convolution Output Y
B images
M features per image
H_{out}×W_{out} values per feature

Average over N×N blocks, then calculate sigmoid

Output Size
$H_{S(N)}$ = floor (H_{out} / N)
$W_{S(N)}$ = floor (W_{out} / N)

Subsampling/Pooling Output S
B images
M features per image
$H_{S(N)}$×$W_{S(N)}$ values per feature

Sequential Code: Forward Pooling Layer

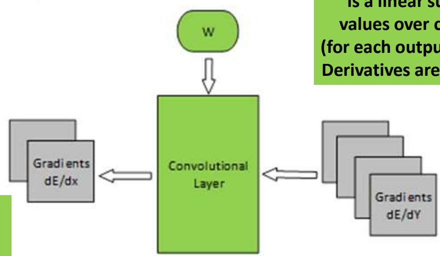
```
void poolingLayer_forward(int B, int M, int H_out, int W_out, int N, float* Y, float* S)
{
  for (int b = 0; b < B; ++b)                    // for each image
    for (int m = 0;  m < M; ++m)                 // for each output feature map
      for (int x = 0; x < H_out/N; ++x)          // for each output value (two loops)
        for (int y = 0; y < W_out/N; ++y) {
          float acc = 0.0f;                      // initialize sum to 0
          for (int p = 0; p < N; ++p)            // loop over NxN block of Y (two loops)
            for (int q = 0; q < N; ++q)
              acc += Y[b, m, N*x + p, N*y + q];
          acc /= N * N;              // calculate average over block
          S[b, m, x, y] = sigmoid(acc + bias[m])  // bias, non-linearity
        }
}
```


Kernel Implementation of Subsampling Layers

- straightforward mapping from grid to subsampled output feature map pixels
- in GPU kernel,
  - need to manipulate index mapping
  - for accessing the output feature map pixels
  - of the previous convolution layer.
- often merged into the previous convolution layer to save memory bandwidth

Backpropagation

Remember that Y is a linear sum of X values over channels (for each output feature). Derivatives are W values.



Y = W · X

$$\frac{dE}{dX} = \frac{dE}{dY}\frac{dY}{dX} = W \cdot \frac{dE}{dY}$$

Calculating dE/dX from dE/dY

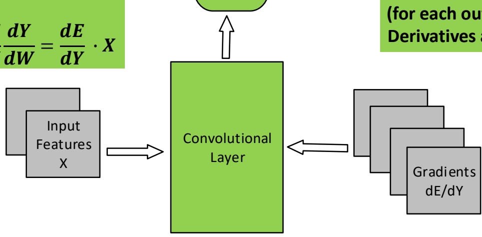
```
void convLayer_backward_dgrad(int B, int M, int C, int H, int W, int K, float *dE_dY, float *W, float *dE_dX)  {
  int H_out = H - K + 1;             // calculate H_out, W_out
  int W_out = W - K + 1;
  for (int b = 0; b < B; ++b) {      // for each image
    for (int c = 0;  c < C; ++c)     // for each input channel
      for (int h = 0; h < H; ++h)    // for each input pixel (two loops)
       for (int w = 0; w < W; ++w)
          dE_dX[b, c, h, w] = 0.0f;  // initialize to 0

    for (int m = 0;  m < M;  ++m)          // for each output feature map
      for (int h = 0; h < H_out; ++h)      // for each output value (two loops)
        for (int w = 0; w < W_out; ++w)
          for (int c = 0;  c < C; ++c)     // for each input channel
            for (int p = 0; p < K; p)      // for each element of KxK filter (two loops)
              for (int q = 0; q < K; ++q)
                dE_dX[b, c, h + p, w + q] += dE_dY[b, m, h, w] * W[m, c, p, q];
  }
}
```


Backpropagation

Y = W · X

$$\frac{dE}{dW} = \frac{dE}{dY}\frac{dY}{dW} = \frac{dE}{dY} \cdot X$$

Remember that Y is a linear sum of X values over channels (for each output feature). Derivatives are W values.



Calculating dE/dW

```
void convLayer_backward_wgrad(int B, int M, int C, int H, int W, int K, float *dE_dY, float *X, float *dE_dW) {
  const int H_out = H - K + 1;           // calculate H_out, W_out
  const int W_out = W - K + 1;
  for (int b = 0; b < B; ++b) {          // for each image
    for(int m = 0; m < M; ++m)           // for each output feature map
     for(int c = 0; c < C; ++c)          // for each channel
        for(int p = 0; p < K; ++p)       // for each element of KxK filter (two loops)
          for(int q = 0; q < K; ++q)
            dE_dW[b, m, c, p, q] = 0.0f;  // initialize to 0

    for(int m = 0;  m < M;  ++m)         // for each output feature map
      for(int h = 0; h < H_out; ++h)     // for each output value (two loops)
        for(int w = 0; w < W_out; ++w)
          for(int c = 0;  c < C; ++c)    // for each channel
            for(int p = 0; p < K; ++p)   // for each element of KxK filter (two loops)
              for(int q = 0; q < K; ++q)
                dE_dW[b, m, c, p, q] += X[b, c, h + p, w + q] * dE_dY[b, m, h, w];
  }
}
```