Objective

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

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:
  - $62,500 \times (250^2)$ 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 \rightarrow \text{Convolution kernels}
  - Can we use them in neural networks?
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

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

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.

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
  $$(N_1 - K_1 + 1) \times (N_2 - K_2 + 1)$$

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

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
### 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

### Convolution

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

### Multi-Layer Percep.

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

### Forward Propagation

**Weights $W$**
- M feature maps
- C channels per map
- K x K pixels per channel

**Input Features $X$**
- B images
- C channels per image
- H x W pixels per channel

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

**Convolve $W$ with $X$ and sum over channels**

**Convolution Output $Y$**
- B images
- M features per image
- H$_{out}$ x W$_{out}$ values per feature

### Outputs Must Use Full Mask/Kernel

- Compute only this part of $Y$. 

**Weights $W$**
- M feature maps
- C channels per map
- K x K pixels per channel

**Input $X$**
- B images
- C channels per image
- H x W pixels per channel

**Output $Y$**
- B images
- M features per image
- H$_{out}$ x W$_{out}$ values per feature
Example of the Forward Path of a Convolution Layer

Output Size
\[ H_{out} = H - K + 1 \]
\[ W_{out} = W - K + 1 \]

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

Weights W
- M=2 feature maps
- C=3 channels per map
- \( K \times K = 2 \times 2 \) pixels per channel

Input X
- B=1 image
- C=3 channels
- \( H \times W = 3 \times 3 \) pixels per channel

Sequential Code: Forward Convolutional Layer
```c
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

<table>
<thead>
<tr>
<th>X[b, 0, _, _]</th>
<th>X[b, 1, _, _]</th>
<th>X[b, 2, _, _]</th>
<th>W[0, 0, _, _]</th>
<th>Y[b, 0, _, _]</th>
<th>Y[b, 1, _, _]</th>
<th>Y[b, 2, _, _]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 0 1</td>
<td>0 2 1 0</td>
<td>1 2 1 0</td>
<td>1 1 1 1</td>
<td>1 0 1 0</td>
<td>0 1 1 1</td>
<td>1 2 1 0</td>
</tr>
<tr>
<td>1 1 3 2</td>
<td>1 1 0 2</td>
<td>1 3 2 0</td>
<td>2 2 2 0</td>
<td>2 1 0 3</td>
<td>1 3 2 0</td>
<td>2 1 0 3</td>
</tr>
<tr>
<td>0 2 2 0</td>
<td>0 3 2 1</td>
<td>0 1 3 2</td>
<td>2 1 0 3</td>
<td>2 1 0 3</td>
<td>0 1 3 2</td>
<td>2 1 0 3</td>
</tr>
<tr>
<td>2 1 0 3</td>
<td>1 1 0 2</td>
<td>3 3 2 0</td>
<td>2 1 0 3</td>
<td>2 1 0 3</td>
<td>3 3 2 0</td>
<td>2 1 0 3</td>
</tr>
</tbody>
</table>
### A Small Convolution Layer Example

**c = 1**

$$
\begin{array}{cccc}
0 & 2 & 1 & 0 \\
1 & 1 & 3 & 2 \\
1 & 2 & 0 & 1 \\
0 & 2 & 2 & 0 \\
2 & 1 & 0 & 3
\end{array}
\quad
\begin{array}{cccc}
1 & 1 & 1 & 2 \\
2 & 2 & 3 & 1 \\
2 & 1 & 0 & 3 \\
1 & 1 & 0 & 2 \\
0 & 2 & 1 & 0
\end{array}
\quad
\begin{array}{cccc}
W[0,0, \_] \\
W[0,1, \_] \\
W[0,2, \_] \\
W[0,3, \_] \\
W[0,4, \_]
\end{array}
\quad
\begin{array}{c}
Y[b,0, \_]
\end{array}
$$

### A Small Convolution Layer Example

**c = 2**

$$
\begin{array}{cccc}
0 & 2 & 1 & 0 \\
1 & 1 & 3 & 2 \\
1 & 2 & 0 & 1 \\
0 & 2 & 2 & 0 \\
2 & 1 & 0 & 3
\end{array}
\quad
\begin{array}{cccc}
1 & 1 & 1 & 2 \\
2 & 2 & 3 & 1 \\
2 & 1 & 0 & 3 \\
1 & 1 & 0 & 2 \\
0 & 2 & 1 & 0
\end{array}
\quad
\begin{array}{cccc}
W[0,0, \_] \\
W[0,1, \_] \\
W[0,2, \_] \\
W[0,3, \_] \\
W[0,4, \_]
\end{array}
\quad
\begin{array}{c}
Y[b,0, \_]
\end{array}
$$

---

### 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.**

### 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.)
Assume

- \( M = 4 \) (4 output feature maps),
- thus 4 blocks in the X dimension, and
- \( W_{\text{out}} = H_{\text{out}} = 8 \) (8x8 output features).

If \( \text{TILE\_WIDTH} = 4 \), 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.

Consider an output feature map:

- width is \( W_{\text{out}} \), and
- height is \( H_{\text{out}} \).

Assume these are multiples of \( \text{TILE\_WIDTH} \).

Let \( X_{\text{grid}} \) be the number of blocks needed in X dim (5 above).
Let \( Y_{\text{grid}} \) be the number of blocks needed in Y dim (4 above).

Host Code for a Basic Kernel: CUDA Grid

(Assuming \( W_{\text{out}} \) and \( H_{\text{out}} \) are multiples of \( \text{TILE\_WIDTH} \).)

```c
#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;

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

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

Partial Kernel Code for a Convolution Layer

```c
__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++)
            for (int q = 0; q < K; q++)
                acc += X[c, h + p, w + q] * W[m, c, p, q];
    }
    Y[m, h, w] = acc;
}
```
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**

<table>
<thead>
<tr>
<th>Convolution Output Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>B images</td>
</tr>
<tr>
<td>M features per image</td>
</tr>
<tr>
<td>H_{out} x W_{out} values per feature</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subsampling/Pooling Output S</th>
</tr>
</thead>
<tbody>
<tr>
<td>B images</td>
</tr>
<tr>
<td>M features per image</td>
</tr>
<tr>
<td>H_{S(N)} x W_{S(N)} values per feature</td>
</tr>
</tbody>
</table>

Average over N x N blocks, then calculate sigmoid

Output Size

\[
H_{S(N)} = \text{floor}\left(\frac{H_{out}}{N}\right) \\
W_{S(N)} = \text{floor}\left(\frac{W_{out}}{N}\right)
\]

---

**Sequential Code: Forward Pooling Layer**

```c
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 N x N 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.

\[
Y = W \cdot X
\]

\[
\frac{dE}{dX} = \frac{dE}{dY} \cdot \frac{dY}{dX} = W \cdot \frac{dE}{dY}
\]

Calculating $dE/dX$ from $dE/dY$

```c
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)
                            dE_dX[b, c, h + p, w + q] += dE_dY[b, m, h, w] * W[m, c, p, q];
    }
}
```

Calculating $dE/dW$

```c
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)
                            dE_dW[b, m, c, p, q] += X[b, c, h + p, w + q] * dE_dY[b, m, h, w];
    }
}
```