ECE408/CS483/CSE408 Spring 2019
Convolutional Neural Networks

Carl Pearson
pearson@illinois.edu

MLP (Multi-Layer Perceptron) for an Image

- Consider a 250 x 250 image
  - Vectorize the 2D image to a 1D vector as input feature
  - Each hidden node would require 250x250 weights ~ 62,500 and the number of
    nodes is the number of output pixels of the hidden layer
  - How about multiple hidden layer? Bigger image?
    - Too many weights, computational and memory expensive
- Traditional feature detection in image processing
  - Filters $\rightarrow$ Convolution kernels
  - Can we use it in neural networks?

2-D Convolution

Convolution vs Fully-Connected (Weight Sharing)
Applicability of Convolution

- Fixed-size inputs
- Variably-sized inputs
  - Varying 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>1-D Audio waveform</td>
<td>Skeleton animation data: 1-D joint angles for each joint</td>
</tr>
<tr>
<td>2-D Fourier-transformed audio data</td>
<td>Color image data: 2-D data for R,G,B channels</td>
</tr>
<tr>
<td>Convolve frequency axis: invariant to frequency shifts Convolve time axis: invariant to shifts in time</td>
<td>Color video: 2-D data across 1-D time for R,G,B channels</td>
</tr>
<tr>
<td>3-D Volumetric data, e.g. medical imaging</td>
<td></td>
</tr>
</tbody>
</table>

LeNet-5: CNN for hand-written digit recognition

Deep Learning 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**
  - B convolution kernels \((K_1 \times K_2)\)

- **Output Features/channels**
  - A × B outputs
    \((N_1 - K_1 + 1) \times (N_2 - K_2 + 1)\)

Aside: 2-D Pooling

- 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 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

**MLP**
- Vector
- \( Y = w \times x + b \)
- Maximum receptive field
- More weights

---

**Forward Propagation**

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

**Convolve \( W \) with \( X \) and sum over channels**

**Output Size**
- \( H_{out} = H - K + 1 \)
- \( W_{out} = W - K + 1 \)

---

**Example of the Forward Path of a Convolution Layer**

**Input X**
- \( B \) images
- \( C \) channels per image
- \( H \times W \) pixels per channel

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

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

---

**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];
                }
}
```
**Subsampling (Pooling) by Scale N**

- **Convolution Output Y**
  - B images
  - M features per image
  - \( H_{\text{out}} \times W_{\text{out}} \) values per feature

- **Subsampling/Pooling Output S**
  - B images
  - M features per image
  - \( H_{\text{S(N)}} \times W_{\text{S(N)}} \) values per feature

**Average over N×N blocks, then calculate sigmoid**

- Output Size
  - \( H_{\text{S(N)}} = \text{floor}(H_{\text{out}} / N) \)
  - \( W_{\text{S(N)}} = \text{floor}(W_{\text{out}} / N) \)

---

**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) {
                    S[b, m, x, y] = 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)
                            S[b, m, x, y] += Y[b, m, N*x + p, N*y + q];
                    S[b, m, x, y] /= N * N; // calculate average over block
                    S[b, m, x, y] = sigmoid(S[b, m, x, y] + bias[m]) // bias, non-linearity
                }
}
```

---

**Reorganize Input X for Convolution**

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

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

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

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

**Consider one image b. One thread computes one value of Y for all features.**

---

**Reorganize Input X for Convolution**

- **Input X**
  - B images
  - C channels per image
  - \( H \times W \) pixels per channel

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

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

- **Unrolled Input X**
  - B images
  - \( H_{\text{out}} \times W_{\text{out}} \) pixels per image
  - \( C \times K \times K \) values per pixel

**Each thread needs a K×K block of X for each channel.**

**Note:** Likely to be larger than X, since values are replicated.
Function to generate “unrolled” X

```c
void unroll(int B, int C, int H, int W, int K, float* X, float* X_unroll) {
    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
            int w_base = c * (K*K);                  // per-channel offset for smallest X_unroll index
            for (int p = 0; p < K; ++p)              // for each element of KxK filter (two loops)
                for (int q = 0; q < K; ++q) {
                    for (int h = 0; h <  H_out; ++h)     // for each thread (each output value, two loops)
                        for (int w = 0; w < W_out; ++w) {
                            int w_unroll = w_base + p * K + q;  // smallest index--data needed by one thread
                            int h_unroll = h * W_out + w;       // next index--across threads (output values)
                            X_unroll[b, h_unroll, w_unroll] = X[b, c, h + p, w + q]; // copy input pixels
                        }
        }
}
```

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)
                            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

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

\[
Y = W \cdot X
\]

\[
\frac{dE}{dW} = \frac{dE}{dY} \frac{dY}{dW} = \frac{dE}{dY} W
\]

Remember that Y is a linear sum of X values over channels (for each output feature). Derivatives are W values.
Calculating dE/dW (Why per image? Not sure.)

```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)
                            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];

    }
}
```