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 layers? Bigger image?
  - Too many weights, computational and memory expensive

- Traditional feature detection in image processing
  - Filters → 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 limb</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</td>
<td>Convolve time axis: invariant to shifts in time</td>
</tr>
<tr>
<td>3-D Volumetric data, e.g. medical imaging</td>
<td>Color video: 2-D data across 1-D time for R,G,B channels</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
- \(A \times B\) outputs
  \((N_1 - K_1/2) \times (N_2 - K_2/2)\)

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
- \( Y = w \cdot x + b \)
- Maximum receptive field
- More weights

---

### Forward Propagation

**Example of the Forward Path of a Convolution 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;
    int W_out = W - K + 1;
    for (int b = 0; b < B; ++b) // for each image in batch
        for(int m = 0; m < M; m++) // for each output feature map
            for(int h = 0; h < H_out; h++) // for each output element
                for(int w = 0; w < W_out; w++) {
                    Y[m, h, w] = 0;
                    for(int c = 0; c < C; c++) // sum over all input feature maps
                        for(int p = 0; p < K; p++) // KxK filter
                            for(int q = 0; q < K; q++)
                                Y[m, h, w] += X[b, c, h + p, w + q] * W[m, c, p, q];
                }
}
```

**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;
    int W_out = W - K + 1;
    for (int b = 0; b < B; ++b) // for each image in batch
        for(int m = 0; m < M; m++) // for each output feature map
            for(int h = 0; h < H_out; h++) // for each output element
                for(int w = 0; w < W_out; w++) {
                    Y[m, h, w] = 0;
                    for(int c = 0; c < C; c++) // sum over all input feature maps
                        for(int p = 0; p < K; p++) // KxK filter
                            for(int q = 0; q < K; q++)
                                Y[m, h, w] += X[b, c, h + p, w + q] * W[m, c, p, q];
                }
}
```
Sequential Code: Forward Pooling Layer

```c
void poolingLayer_forward(int B, int M, int H, int W, int K, float* Y, float* S) {
    for (int b = 0; b < B; ++b) // for each image in batch
        for (int m = 0; m < M; ++m) // for each output feature maps
            for (int h = 0; h < H/K; ++h) // for each output element
                for (int w = 0; w < W/K; ++w) {
                    S[m, x, y] = 0.;
                    for (int p = 0; p < K; ++p) // loop over KxK input samples
                        for (int q = 0; q < K; ++q)
                            S[m, x, y] += Y[b, m, K*x + p, K*y + q] / (K*K);
                }
        S[b, m, h, w] = sigmoid(S[b, m, h, w] + b[m]) // bias, non-linearity
    }
```

Backpropagation

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;
    int W_out = W – K + 1;
    for (int b = 0; b < B; ++b)
        for (int c = 0; c < C; ++c) {
            int w_base = c * (K*K);
            for (int p = 0; p < K; ++p)
                for (int q = 0; q < K; ++q)
                    for (int h = 0; h <  H_out; ++h)
                        for (int w = 0; w < W_out; ++w) {
                            int w_unroll = w_base + p * K + q;
                            int h_unroll = h * W_out + w;
                            X_unroll[b, h_unroll, w_unroll] = X[b, c, h + p, w + q];
                        }
        }
}
```

Calculating dE/dX

```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;
    int W_out = W – K + 1;
    for (int b = 0; b < B; ++b) {
        for (int c = 0; c < C; ++c)
            for (int h = 0; h < H; ++h)
                for (int w = 0; w < W; ++w)
                    dE_dX[b, c, h, w] = 0;
        for (int m = 0; m < M; ++m)
            for (int h = 0; h < H_out; ++h)
                for (int w = 0; w < W_out; ++w)
                    for (int c = 0; c < C; ++c)
                        for (int p = 0; p < K; ++p)
                            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];
    }
```
void convLayer_backward_wgrad(int B, int K, int C, int H, int W, int K, float *dE_dY, float *X, float *dE_dW) {
    const int H_out = H - K + 1;
    const int W_out = W - K + 1;
    for (int b = 0; b < B; ++b) {
        for(int m = 0; m < M; ++m)
            for(int c = 0; c < C; ++c)
                for(int p = 0; p < K; ++p)
                    for(int q = 0; q < K; ++q)
                        dE_dW[b, m, c, p, q] = 0;
        for(int m = 0;  m < M;  ++m)
            for(int h = 0; h < H_out; ++h)
                for(int w = 0; w < W_out; ++w)
                    for(int c = 0;  c < C; ++c)
                        for(int p = 0; p < K; ++p)
                            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, c, h, w];
    }
}