ECE408/CS483/CSE408 Spring 2019

Convolutional Neural Networks

Contributed by Carl Pearson
pearson@illinois.edu

Objective

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

LeNet-5: CNN for hand-written digit recognition

Anatomy of a Convolution Layer

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

- Convolution Layer
  - \(A \times B\) convolution kernels \((K_1 \times K_2)\)

- Output Features/channels
  - \(B\) outputs
    - \((N_4 - K_4/2) \times (N_4 - K_4/2)\)
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 (classification) 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

Sequential Code: Forward Convolutional Layer

A Small Convolution Layer Example
Parallelism in a Convolution Layer

- All 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 will need atomic operation or tree reduction
Design of a Basic Kernel

- Each block computes a tile of output pixels
  - TILE_WIDTH pixels in each dimension
- The x dimension in the grid maps to the M output feature maps
- The y dimension in the grid maps to the tiles in the output feature maps (linearization)

A Small Example

- Assume 4 output feature maps (M = 4)
  - Each output feature map is 8x8 image (W_out = H_out = 8)
  - We have 4 blocks in the x dimension
- If we use tiles of 4 pixels on each side (TILE_WIDTH = 4)
  - We have 4 blocks in the x dimension

  • Top two blocks in each column calculates the top row of tiles in the corresponding output feature map
  • Bottom two blocks in each column calculates the bottom row of tiles in the corresponding output feature map

Host Code for a Basic Kernel: CUDA Grid

W_out and H_out are the output feature map width and height

```
#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);
dim3 gridDim(M, Y, 1);
ConvLayerForward_Kernel<<< gridDim, blockDim>>>(…);
```

A Basic Conv. Layer Forward Kernel (Code is Incomplete!)

```
__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 + threadIdx.y;
  int w = blockIdx.y % W_grid + threadIdx.x;
  float acc = 0.;
  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;
}
```
Some Observations

- The amount of parallelism is quite high as long as the total number of pixels across all output feature maps is large
  - This matches the CNN architecture well
- Each input tile is loaded multiple times, once for each block that calculates the output tile that requires the input tile
  - Not very efficient in global memory bandwidth

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 the batch
        for (int m = 0; m < M; ++m) // for each output feature map
            for (int h = 0; h < H/K; ++h) // for each output element
                for (int w = 0; w < W/K; ++y) {
                    S[m, h, w] = 0.;
                    for (int p = 0; p < K; ++p)
                        for (int q = 0; q < K; ++q)
                            S[m, h, w] += Y[b, m, K*h + p, K*w + q] / (K*K);
                }
    S[b, m, h, w] = sigmoid(S[b, m, h, w] + b[m]) // bias, non-linearity
}
```

Kernel Implementation of Subsampling Layers

- Straight forward mapping from grid to subsampled output feature map pixels
  - Need to manipulate index mapping for accessing the output feature map pixels of the previous convolution layer
- It is often merged into the previous convolution layer to save memory bandiwidth

Optimizing Convolution Layers
Implementing a Convolution Layer with Matrix Multiplication

Simple Matrix Multiplication

Each product matrix element is an output feature map pixel.

This inner product generates element 0 of output feature map 0.

Tiled Matrix Multiplication

2x2 Example

Each block calculates one output tile – 2 elements from each output map
Each input element is reused 2 times in the shared memory

Tiled Matrix Multiplication

2x4 Example

Each block calculates one output tile – 4 elements from each output map
Each input element is reused 2 times in the shared memory
Efficiency Analysis: Total Input Replication

- Replicated input features are shared among output maps
  - There are \( H_{out} \times W_{out} \) output feature map elements
  - Each requires \( K \times K \) elements from the input feature maps
  - So, the total number of input element after replication is \( H_{out} \times W_{out} \times K \times K \) times for each input feature map
  - The total number of elements in each original input feature map is \( (H_{out}+K-1) \times (W_{out}+K-1) \)

Analysis of a Small Example

\[ H_{out} = 2 \]
\[ W_{out} = 2 \]
\[ K = 2 \]
There are 3 input maps (channels)
The total number of input elements in the replicated ("unrolled") input matrix is \( 3 \times 2 \times 2 \times 2 \).
The replicating factor is \( \frac{(3 \times 2 \times 2 \times 2)}{(3 \times 3 \times 3)} = 1.78 \)

Memory Access Efficiency of Original Convolution Algorithm

- Assume that we use tiled 2D convolution
- For input elements
  - Each output tile has \( \text{TILE\_WIDTH} \) elements
  - Each input tile has \( (\text{TILE\_WIDTH}+K-1)^2 \)
  - The total number of input feature map element accesses was \( \text{TILE\_WIDTH} \times K^2 \)
  - The reduction factor of the tiled algorithm is \( K^2 \times \text{TILE\_WIDTH} / (\text{TILE\_WIDTH}+K)^2 \)
- The convolution filter weight elements are reused within each output tile

Properties of the Unrolled Matrix

- Each unrolled column correspond to an output feature map element
- For an output feature element \((h,w)\), the index for the unrolled column is \( h \times W_{out} + w \) (linearized index of the output feature map element)
Properties of the Unrolled Matrix (cont.)

- Each section of the unrolled column correspond to an input feature map
- Each section of the unrolled column has $k^2$ elements (convolution mask size)
- For an input feature map $c$, the vertical index of its section in the unrolled column is $c^t k^t$ (linearized index of the output feature map element)

To Find the Input Elements

- For output element $(h, w)$, the base index for the upper left corner of the input feature map $c$ is $(c, h, w)$
- The input element index for multiplication with the convolution mask element $(p, q)$ is $(c, h+p, w+q)$

Input to Unrolled Matrix Mapping

- Output element $(h, w)$
- Mask element $(p, q)$
- Input feature map $c$

To Find the Input Elements

- For output element $(h, w)$, the base index for the upper left corner of the input feature map $c$ is $(c, h, w)$
- The input element index for multiplication with the convolution mask element $(p, q)$ is $(c, h+p, w+q)$

Function to generate “unrolled” X

```c
void unroll(int B, int C, int H, int W, int K, float *X, float *X_unroll) {
    // calculate the horizontal matrix index
    int w_base = c * (K*K);
    // find the beginning of the unrolled horizontal matrix index
    for (int h = 0; h < H_out; ++h) // for each output feature map element
        for (int w = 0; w < W_out; ++w) { // for each output feature map element
            X_unroll[b, h_unroll, w_unroll] = X[b, c, h + p, w + q];
        }
}
```
Implementation Strategies for a Convolution Layer

- **Baseline**
  - Tiled 2D convolution implementation, use constant memory for convolution masks

- **Matrix-Multiplication Baseline**
  - Input feature map unrolling kernel, constant memory for convolution masks as an optimization
  - Tiled matrix multiplication kernel

- **Matrix-Multiplication with built-in unrolling**
  - Perform unrolling only when loading a tile for matrix multiplication
  - The unrolled matrix is only conceptual
  - When loading a tile element of the conceptual unrolled matrix into the shared memory, use the properties in the lecture to load from the input feature map

- **More advanced Matrix-Multiplication**
  - Use joint register-shared memory tiling

Enjoy the project!