上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

CS4302-01
Parallel and Distributed Computing

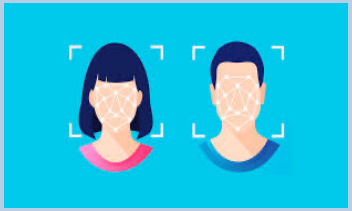
Lecture 9 CUDA for DNN

Zhuoran Song

2023/10/24

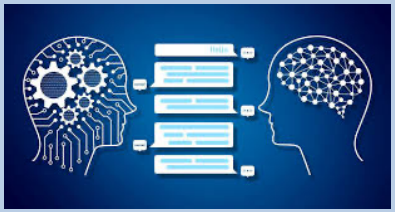
Applications

app

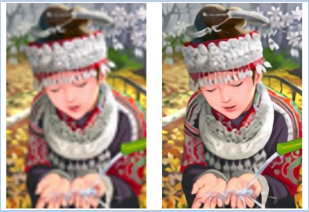


Face recognition

Automatic driving

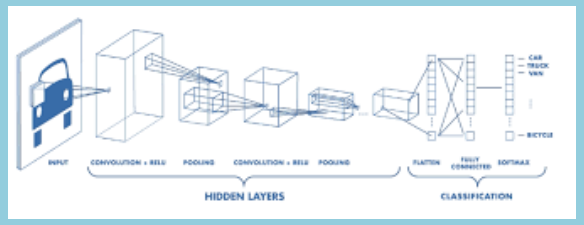


Text generation

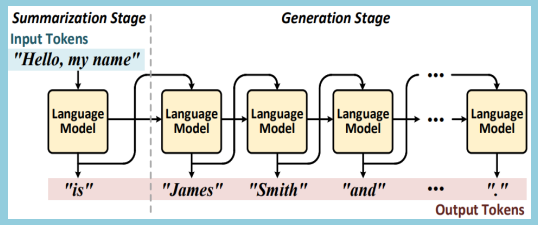


Super-resolution

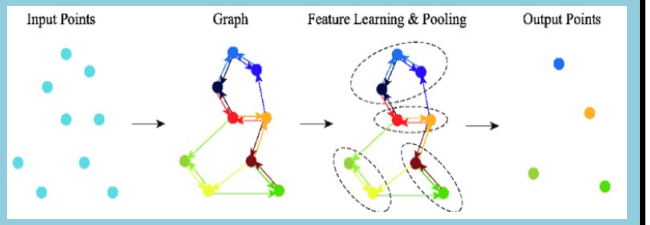
algo



CNN



Transformer

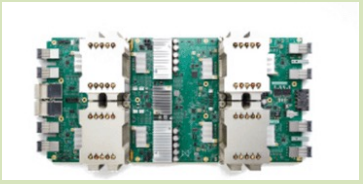


Point cloud neural network

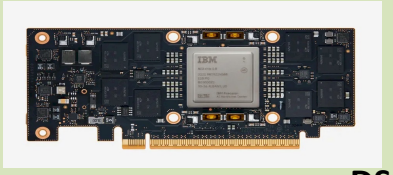
arch

CPU

GPU

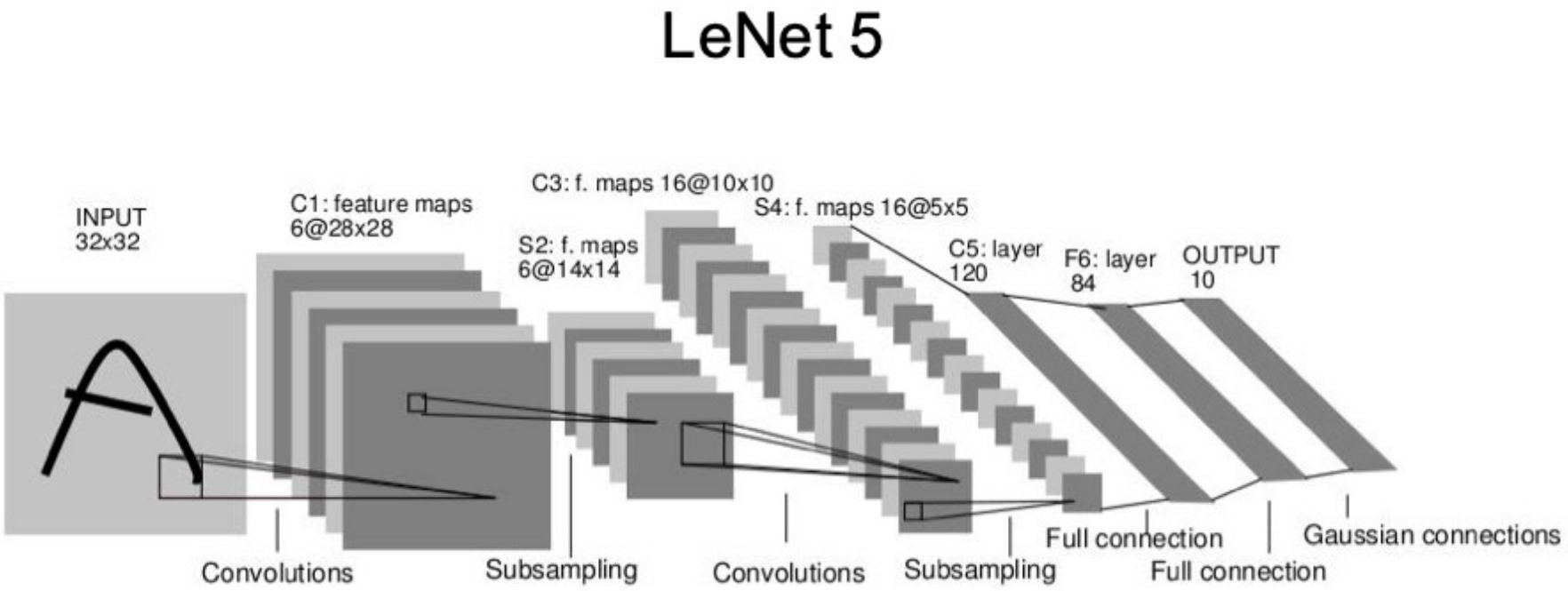


TPU

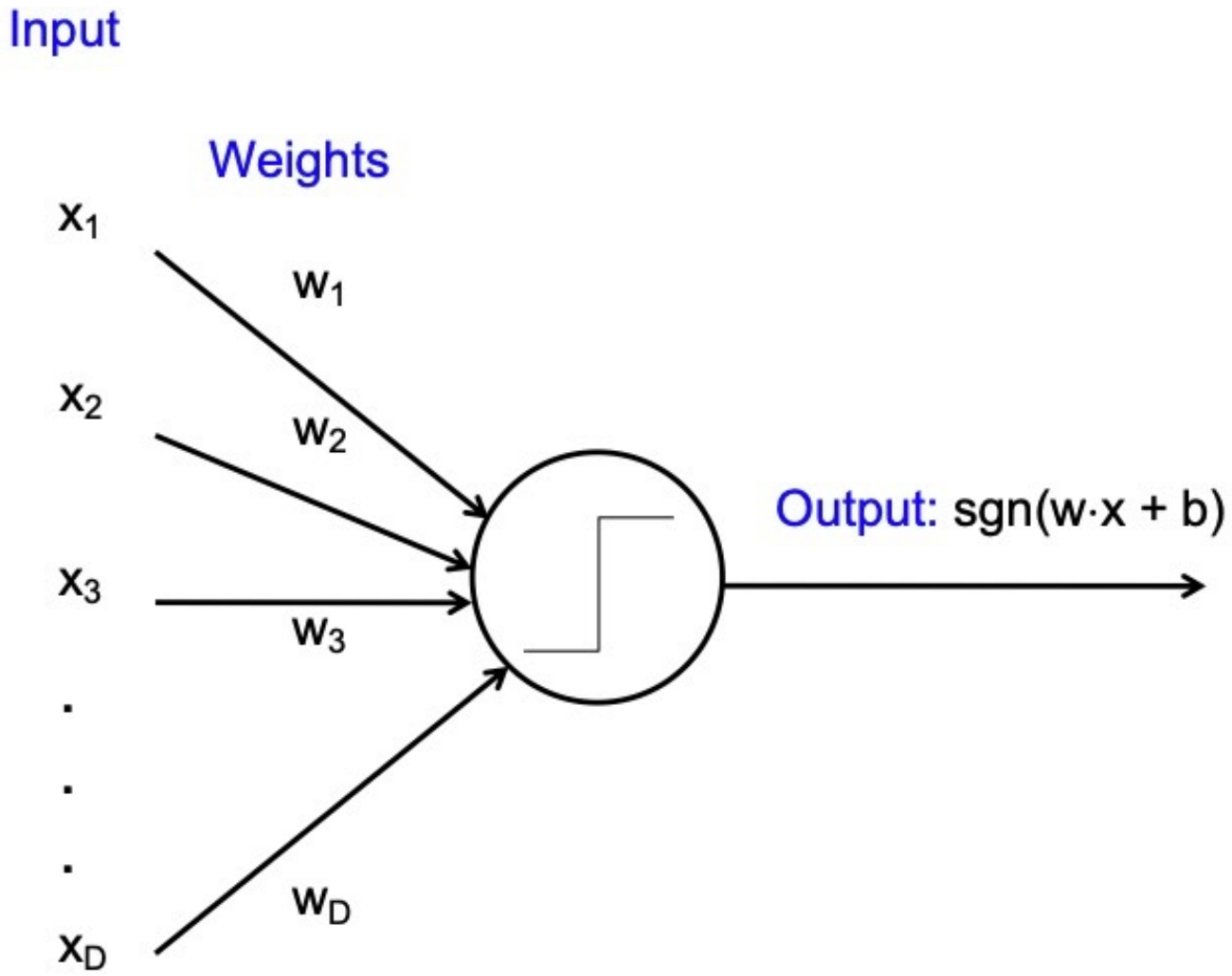


DSA

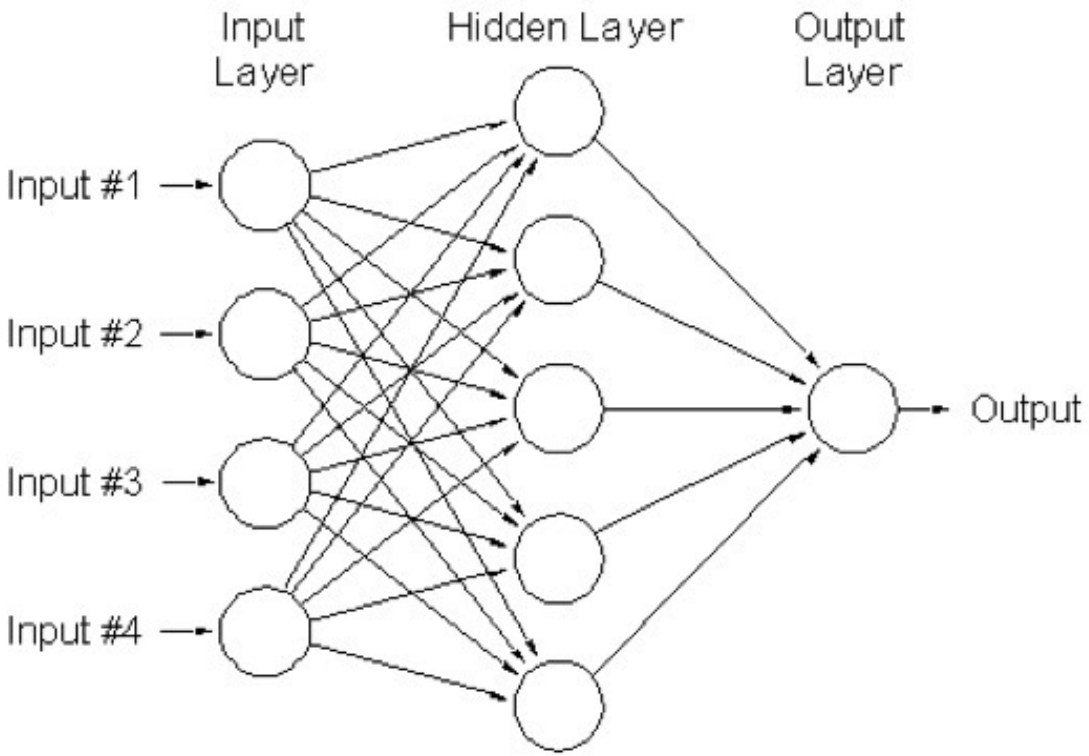
Deep Neural Network



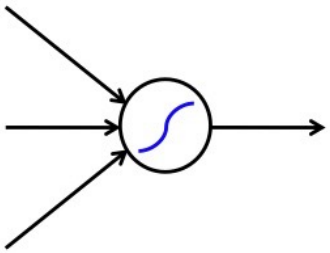
Perceptron



Two-layer neural network

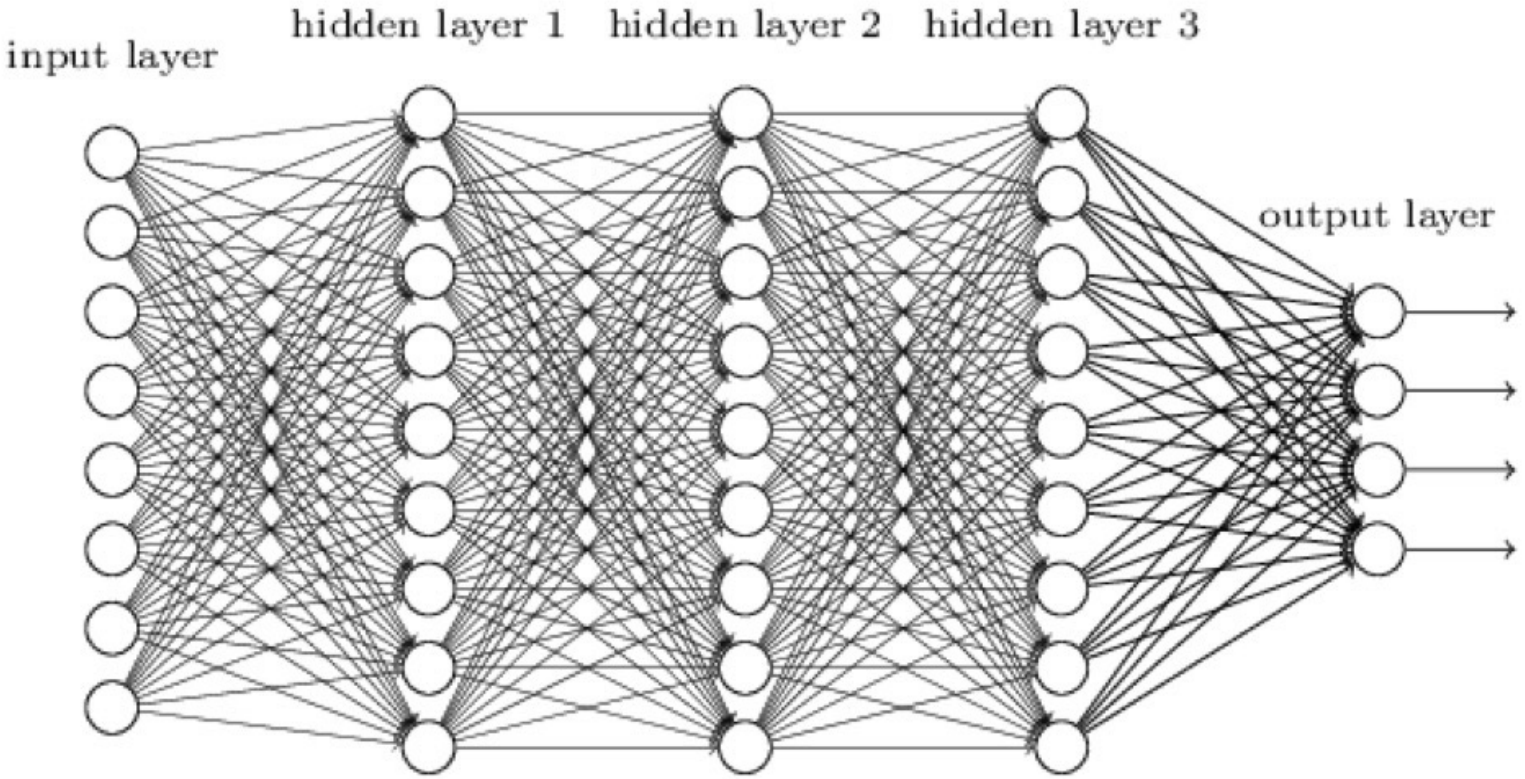


- **Can learn nonlinear functions provided each perceptron has a differentiable nonlinearity**



Sigmoid: $g(t) = \frac{1}{1+e^{-t}}$

Multi-layer neural network

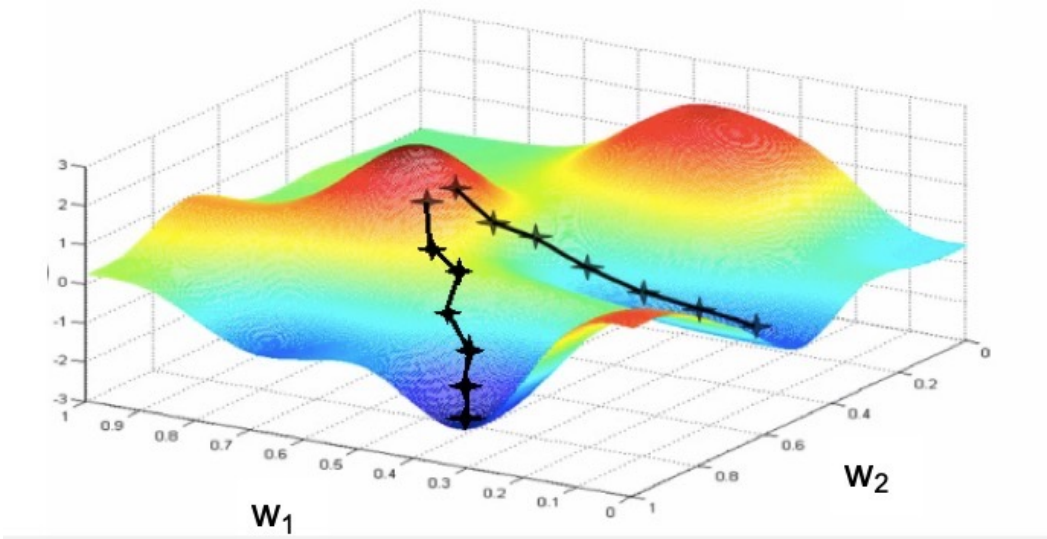


Training of multi-layer network

- **Find network weights to minimize the *training error* between true and estimated labels of training examples, e.g.:**

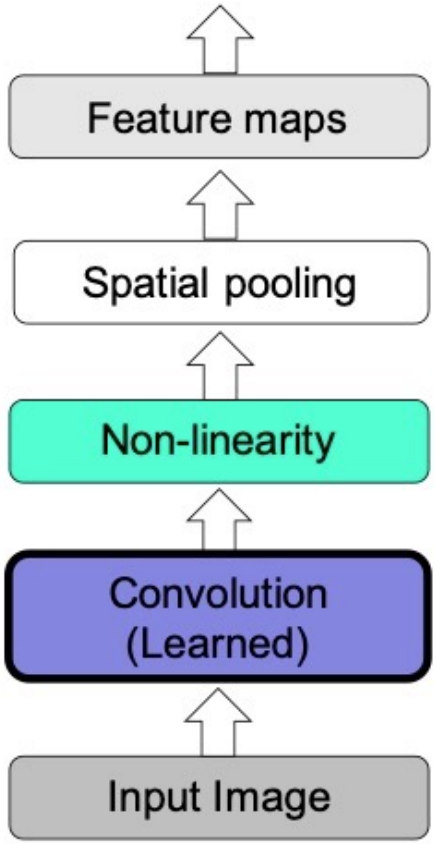
$$E(\mathbf{w}) = \sum_{i=1}^{N} \left(y_i - f_{\mathbf{w}}(\mathbf{x}_i) \right)^2$$

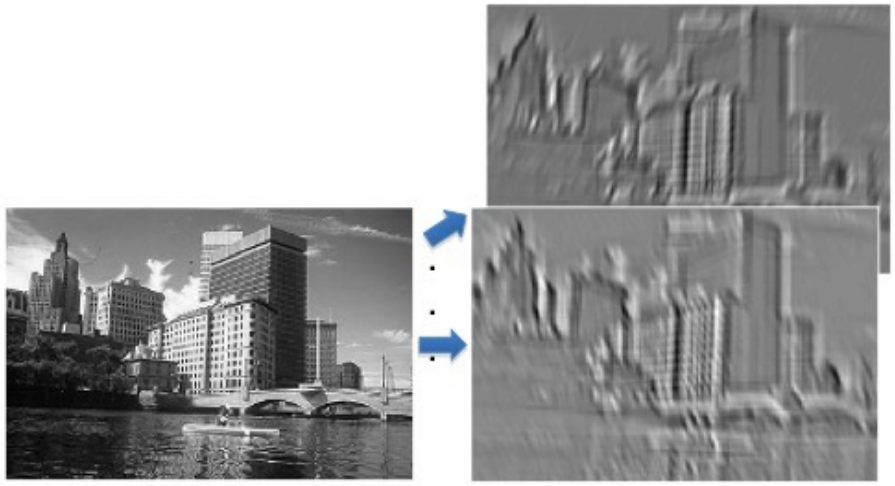
- **Update weights by gradient descent:** $\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial E}{\partial \mathbf{w}}$



Key operations

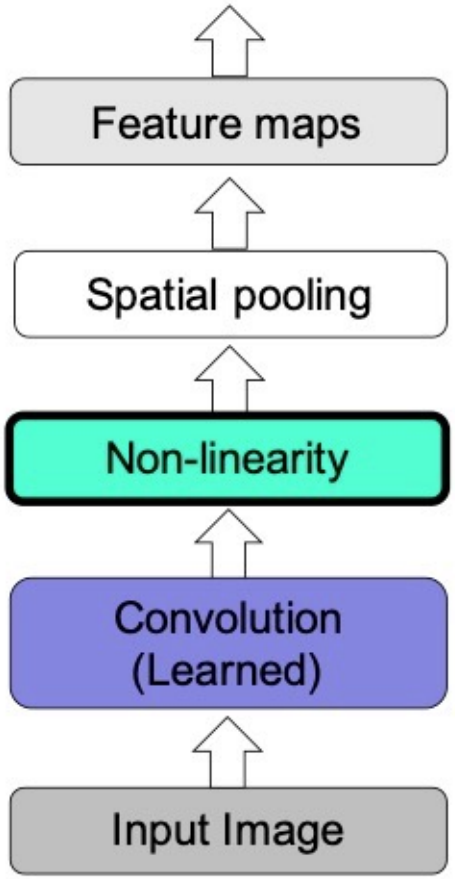
- **Convolution**
- **Non-linearity, Activation**
- **Pooling**

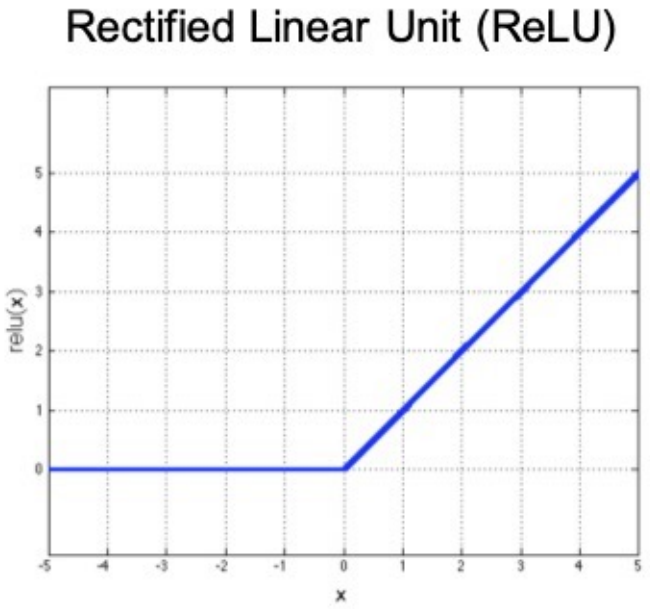




Key operations

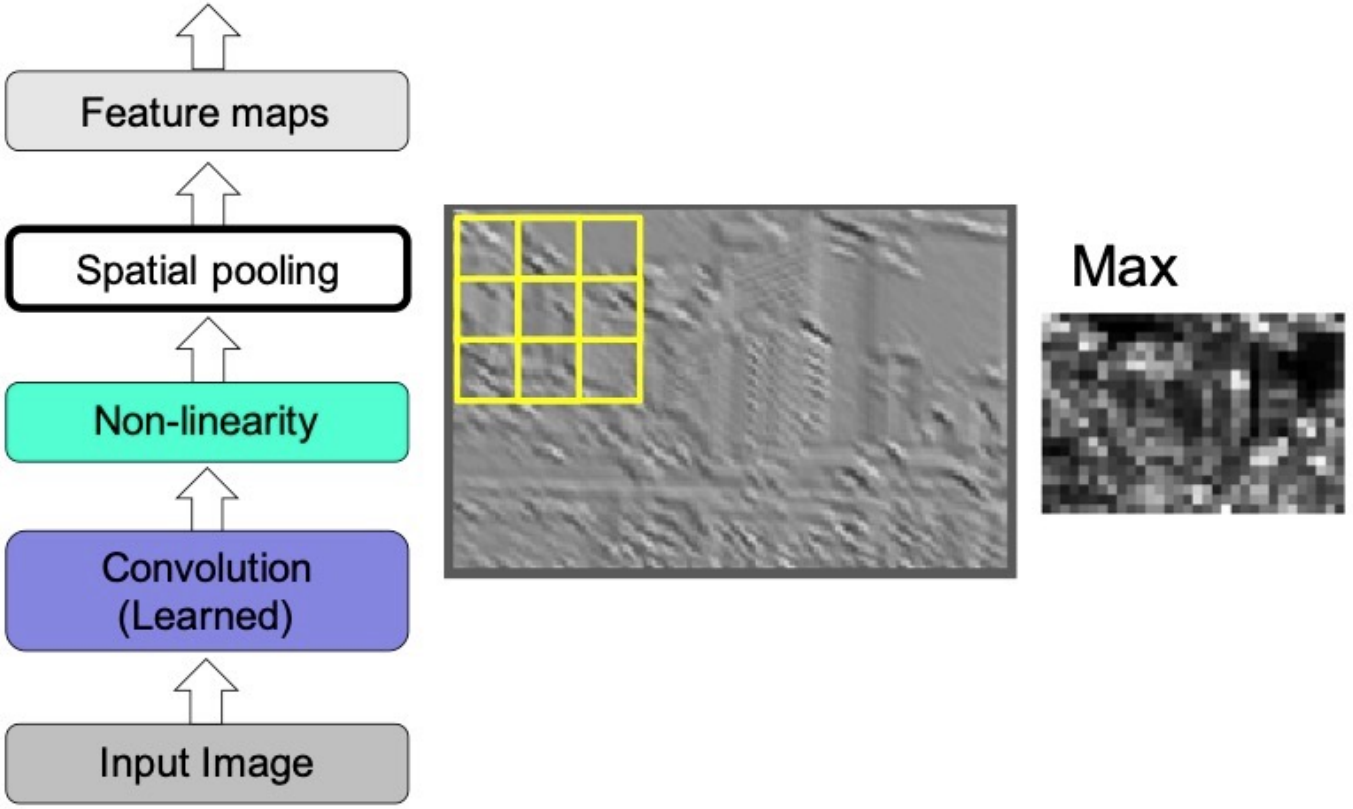
- **Convolution**
- **Non-linearity, Activation**
- **Pooling**





Key operations

- **Convolution**
- **Non-linearity, Activation**
- **Pooling**

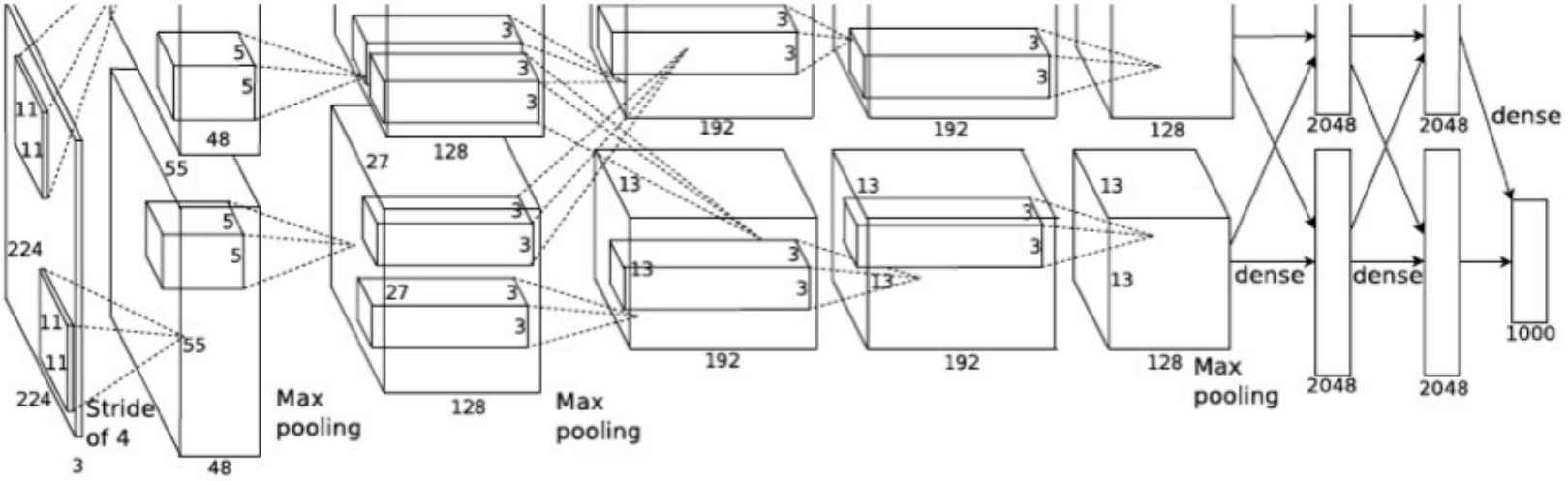


Big Data

- **~14 million labeled images, 20k classes**
- **Images gathered from Internet**
- **Human labels via Amazon MTurk**
- **ImageNet Large-Scale Visual Recognition Challenge (ILSVRC): 1.2 million training images, 1000 classes**

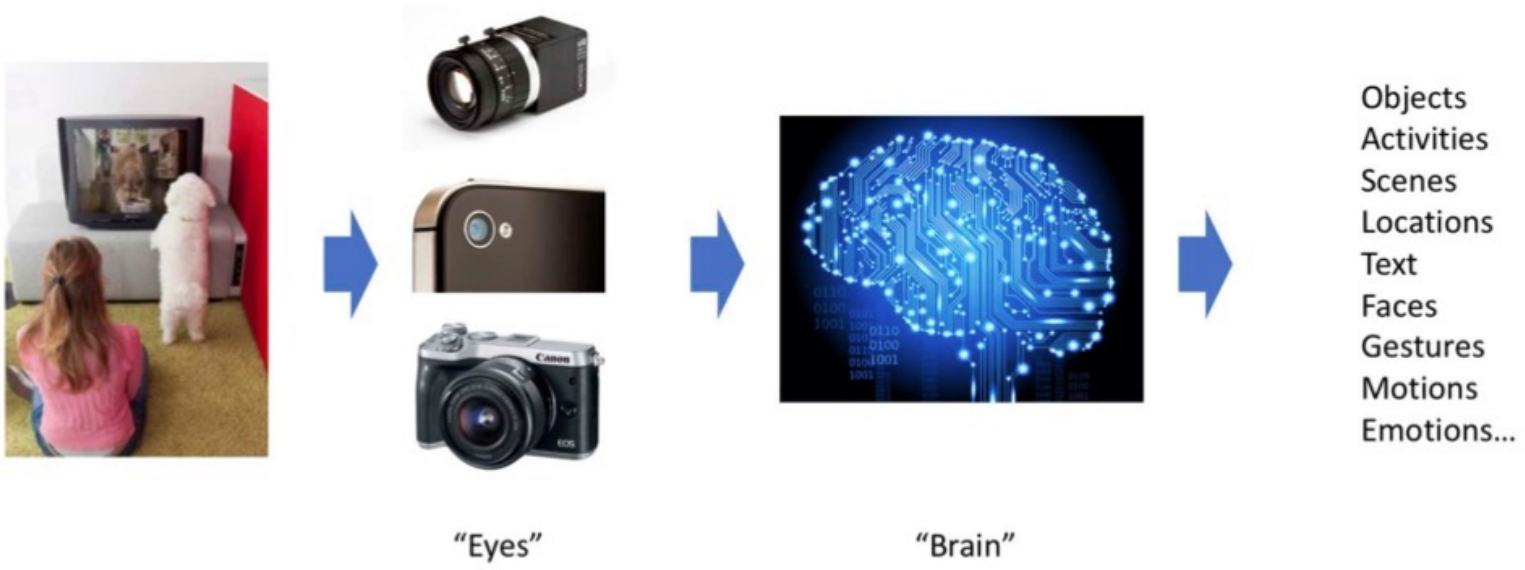
AlexNet

- **Similar framework to LeNet but:**
  - **Max pooling, ReLU nonlinearity**
  - **More data and bigger model (7 hidden layers, 650K units, 60M params)**
  - **Dropout regularization**



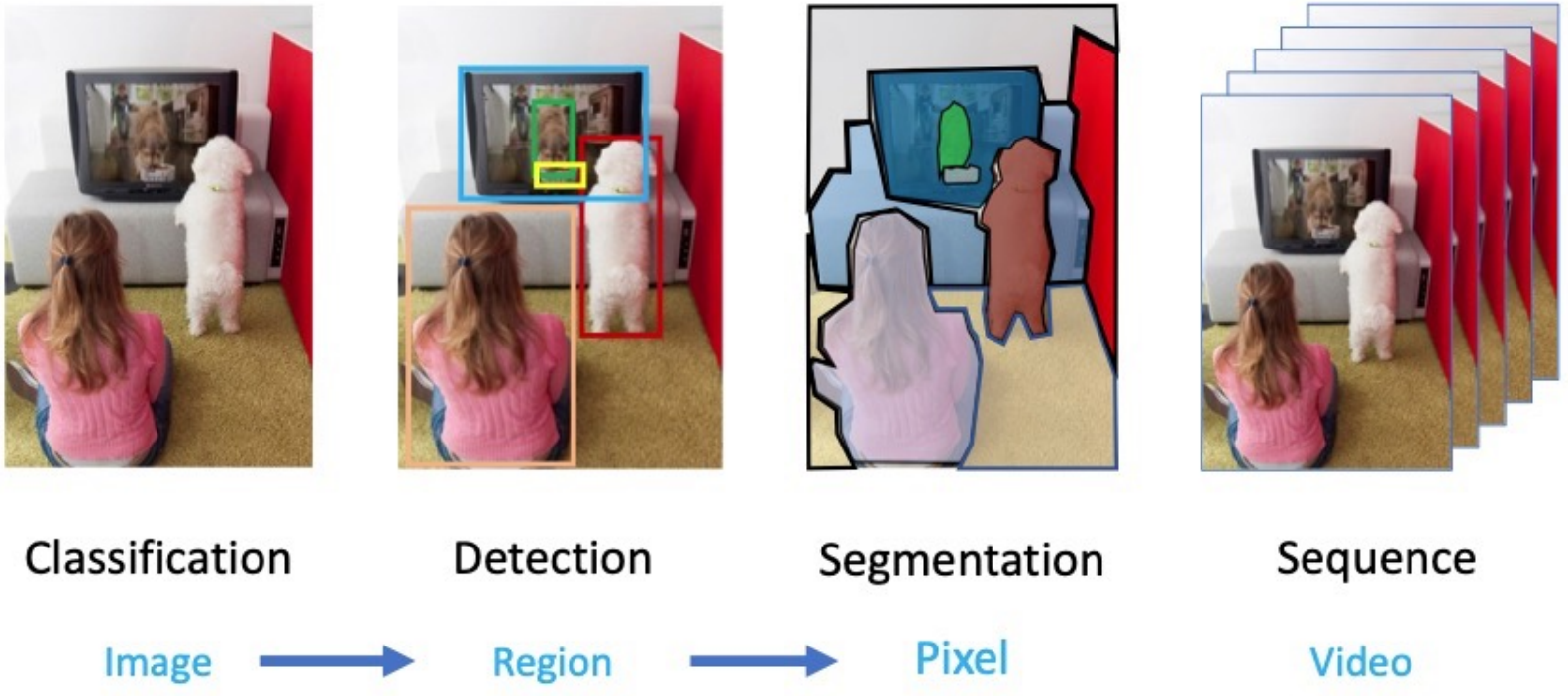
Computer Vision

- Humans use their eyes and their brains to visually sense the world.
- Computers user their cameras and computation to visually sense the world



More Tasks

- Image
- Detection
- Segmentation
- Video Tasks

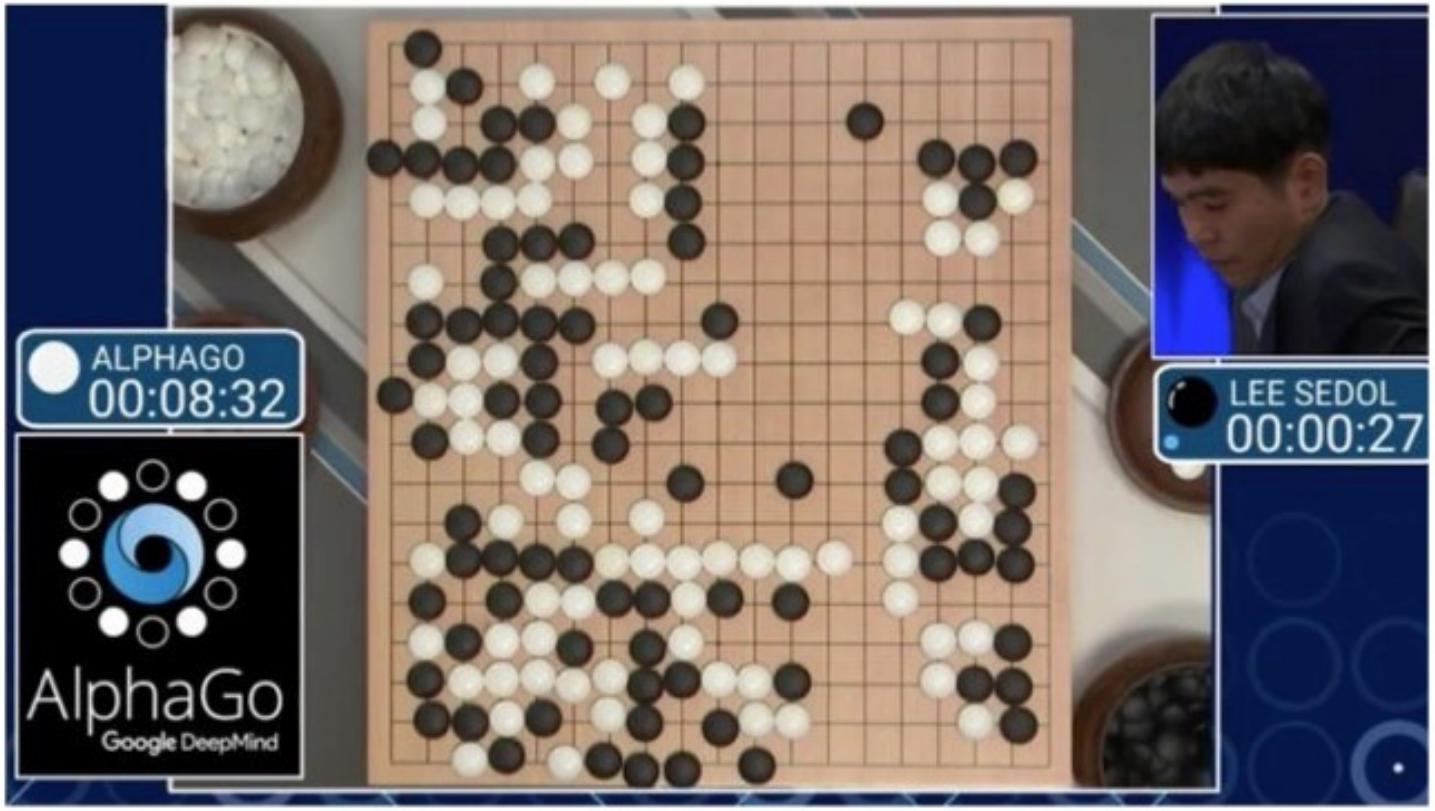


Winter of Neural Networks (mid 90' – 2006)

- **No theory to play**
- **Lack of training data**
- **Benchmark is insensitive**
- **Difficulties in optimization**
- **Hard to reproduce results**

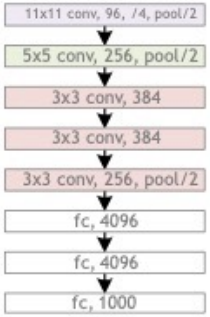
Renaissance of Deep Learning (2006 –)

- A fast learning algorithm for deep belief nets. [Hinton et.al 1996]
- Data + Computing + Industry Competition
- NVidia's GPU, Google Brain (16,000 CPUs)
- Speech: Microsoft [2010], Google [2011], IBM
- Image: AlexNet, 8 layers [Krizhevsky et.al 2012] (26.2% -> 15.3%)

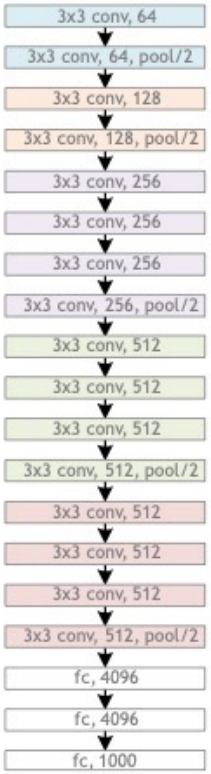


Revolution of Depth

AlexNet, 8 layers (ILSVRC 2012)



VGG, 19 layers (ILSVRC 2014)



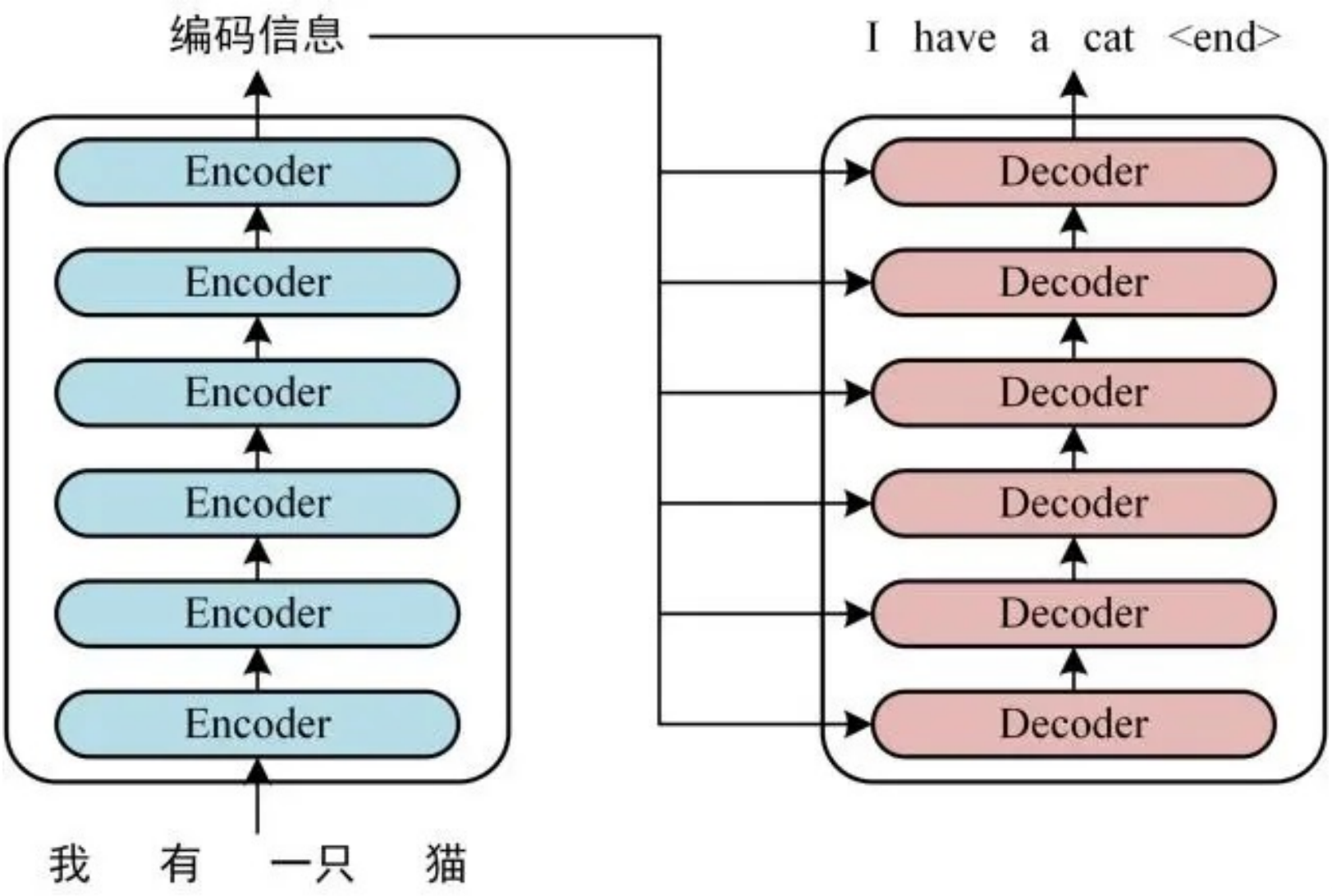
GoogleNet, 22 layers (ILSVRC 2014)

Slide Credit: He et al. (MSRA)

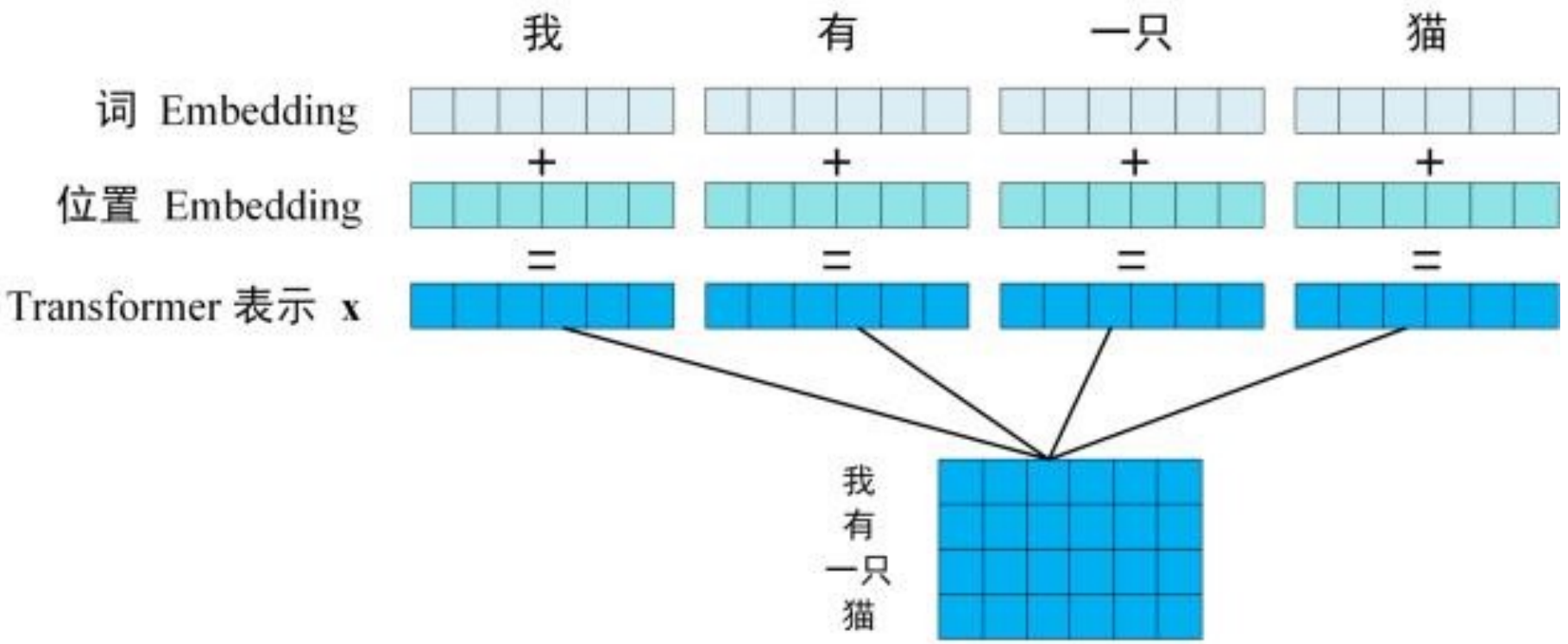
Recent Classification Architectures

- AlexNet (Krizhevsky, Sutskever, and E. Hinton 2012) 233MB
- Network in Network (Lin, Chen, and Yan 2013) 29MB
- VGG (Simonyan and Zisserman 2015) 549MB
- GoogleNet (Szegedy, Liu, et al. 2015) 51MB
- ResNet (He et al. 2016) 215MB
- Inception-ResNet (Szegedy, Vanhoucke, et al. 2016) 23MB
- DenseNet (Huang et al. 2017) 80MB
- Xception (Chollet 2017) 22MB
- MobileNetV2 (Sandler et al. 2018) 14MB
- ShuffleNet (Zhang et al. 2018) 22MB

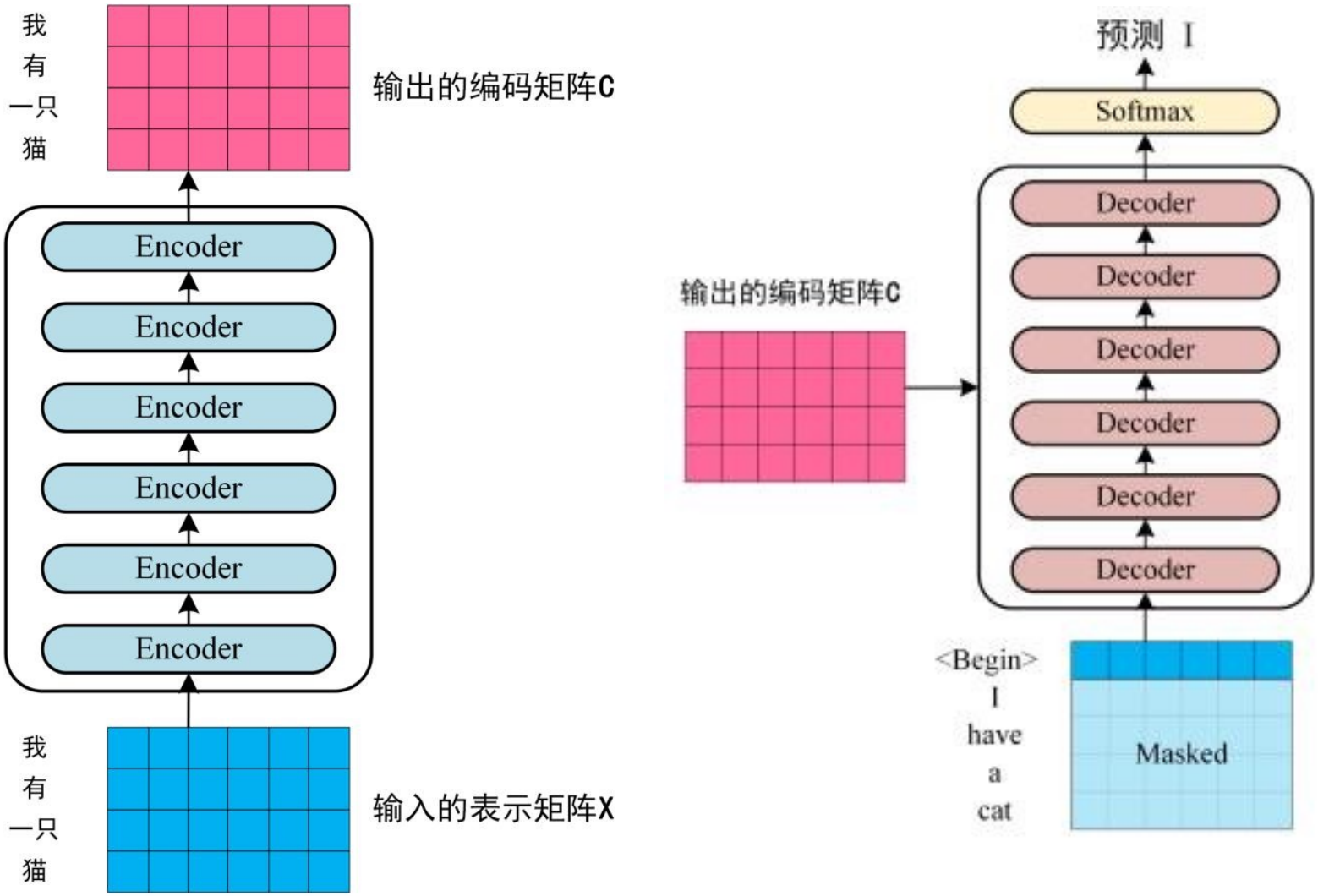
Transformer



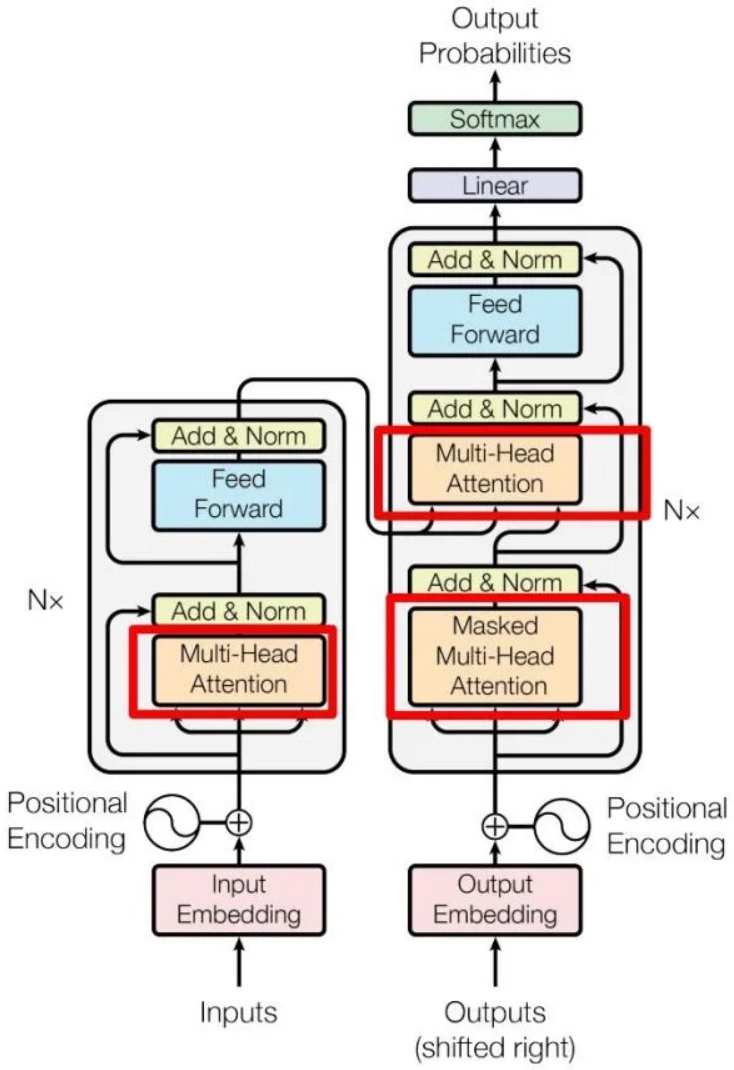
Transformer



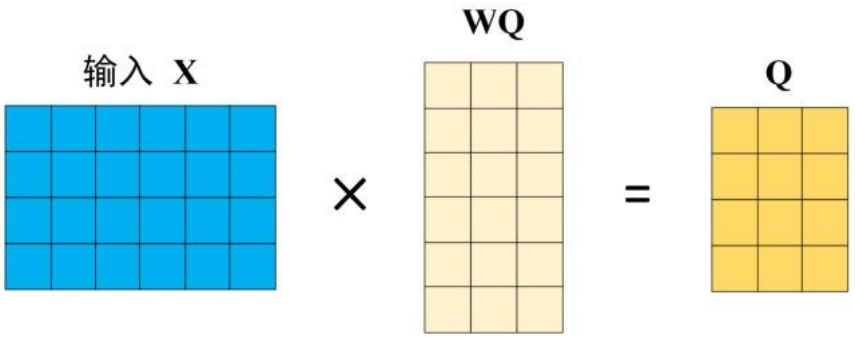
Transformer

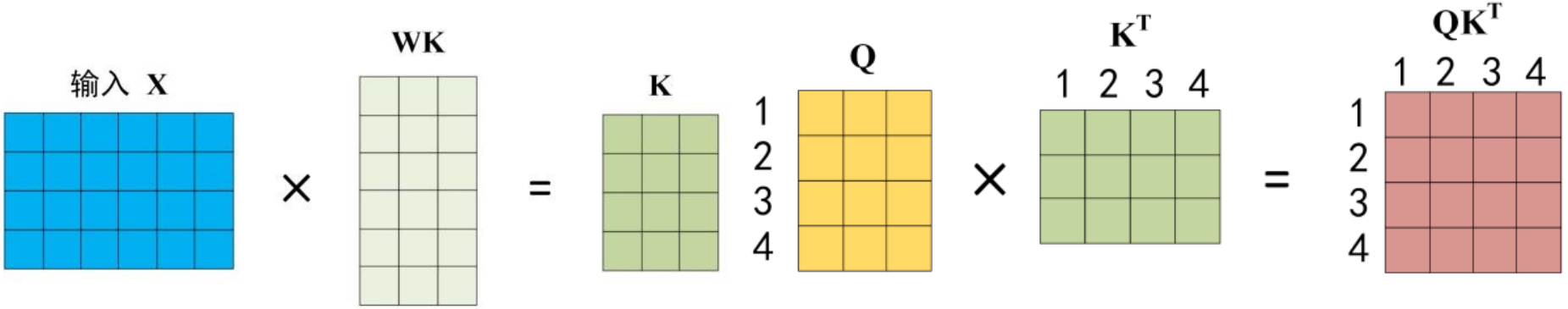


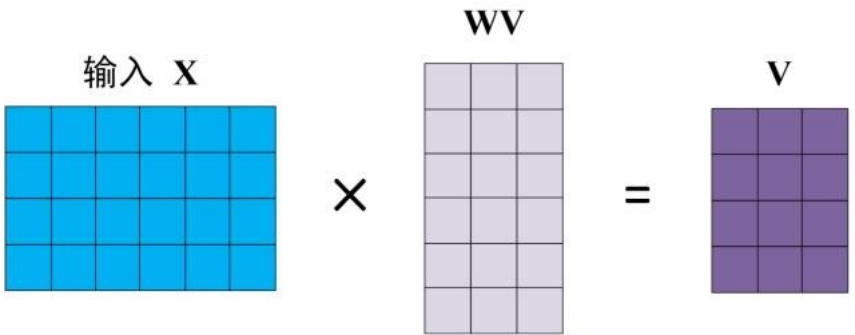
Transformer



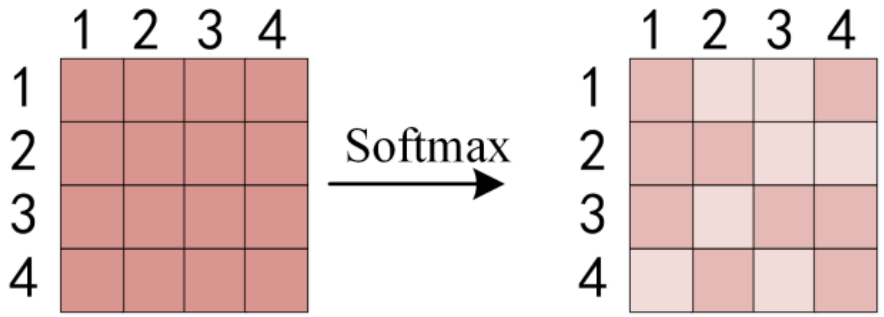
Transformer

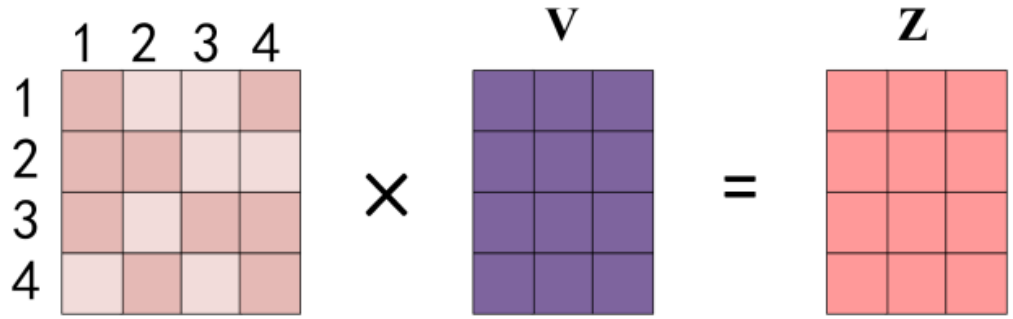


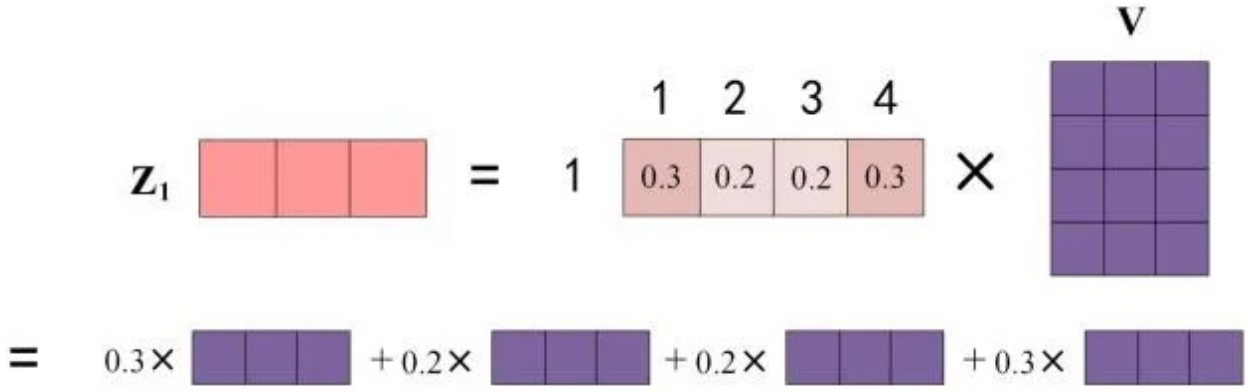




Transformer

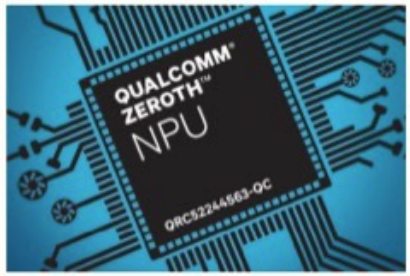






When Machine Learning Meets Hardware

- **Convolution layer is one of the most expensive layers**
  - **Computation pattern**
  - **Emerging challenges**
- **More and more end-point devices with limited memory**
  - **Cameras**
  - **Smartphone**
  - **Autonomous driving**

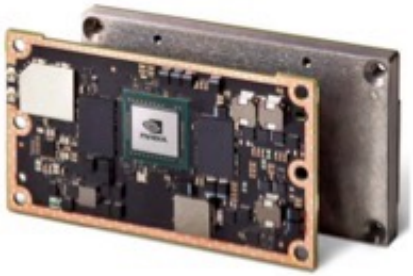


XILINX

MOBILEYE
An Intel Company

Flexibility vs. Efficiency

CPU
(Raspberry Pi3)



GPU
(Jetson TX2)

FPGA
(UltraZed)

ASIC
(Movidius)

Flexibility

Power/Performance
Efficiency

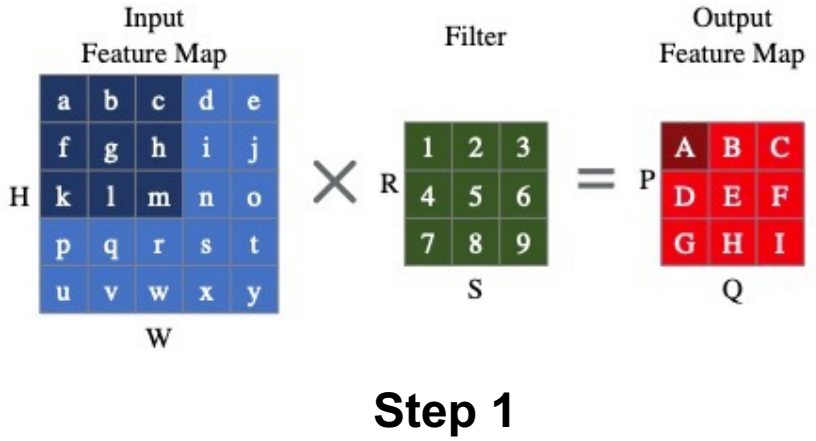
Comparisons: FPGA, ASIC, GPU



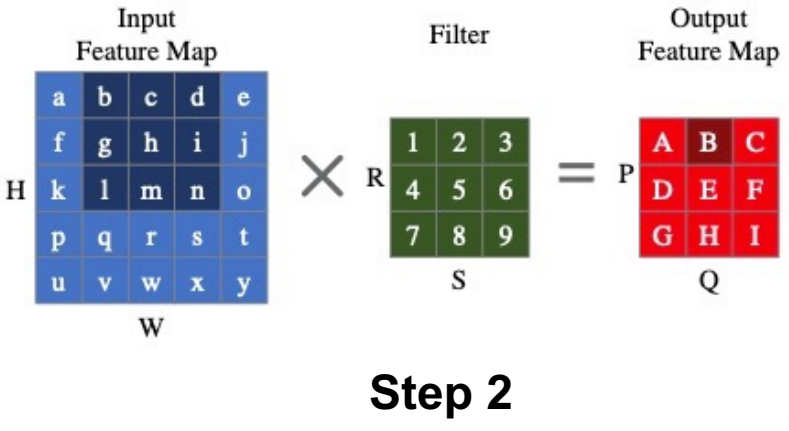


	Xilinx ZCU102	Xilinx ZCU104	Huawei Atlas 200	nVIDIA Jetson TX2	Cambricon MLU 270
price	3K RMB	2K RMB	4K RMB	2.8K RMB	12K RMB
MobileNet-V1	1.14 ms	1.37 ms	1.8 ms	12.44 ms	1.85 ms
ResNet50	5.23 ms	6.81 ms	3.6 ms	24.70 ms	2.54 ms
Inception_v2	2.68 ms	3.35 ms	6.0 ms	10.81 ms	5.12 ms
Inception_v3	6.44 ms	8.53 ms	5.7 ms	32.53 ms	4.71 ms
Inception_v4	11.87 ms	17.06 ms	9.3 ms	44.37 ms	11.33 ms

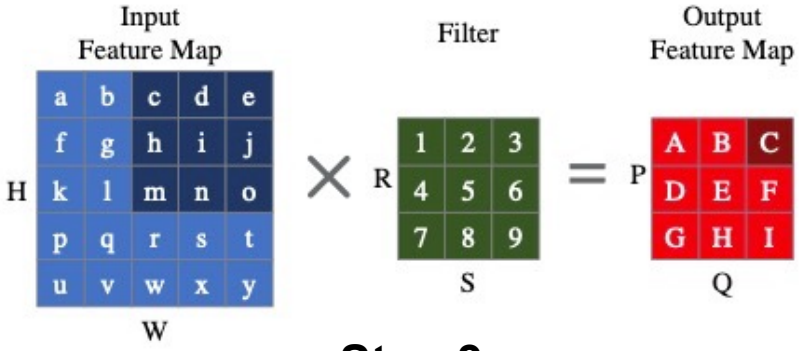
2D-Convolution



Step 1



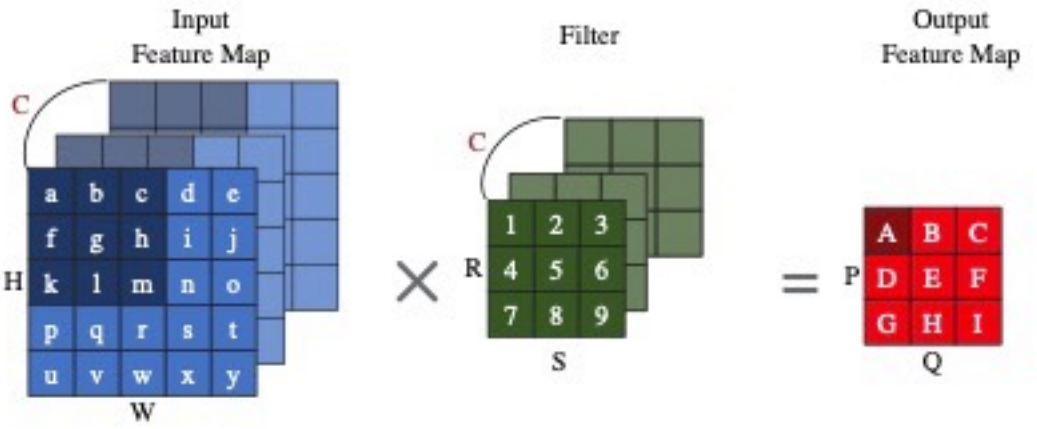
Step 2



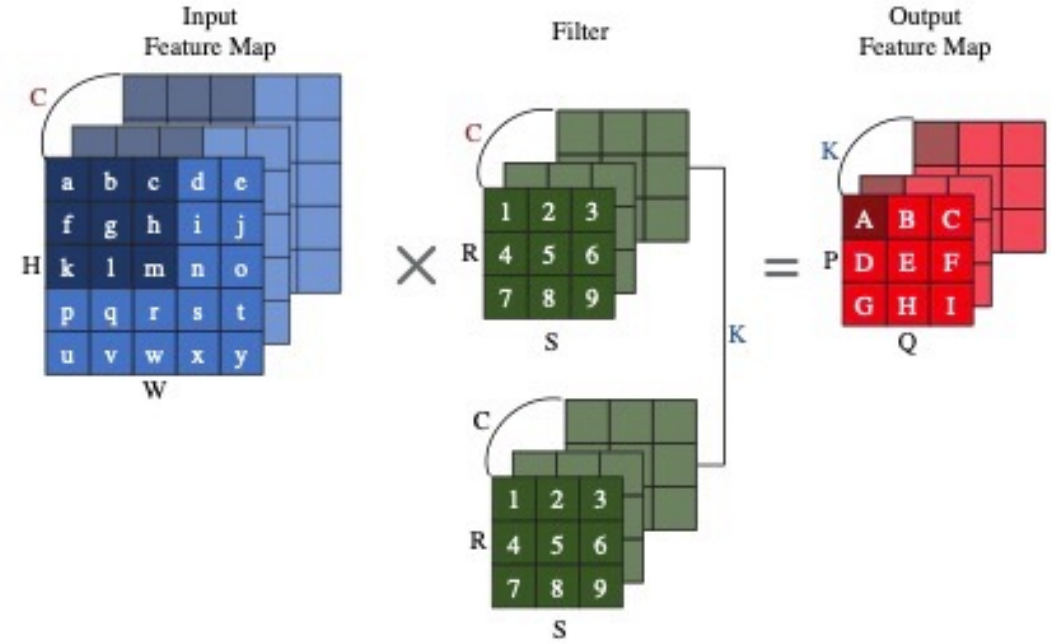
Step 3

3D-Convolution

Step 1

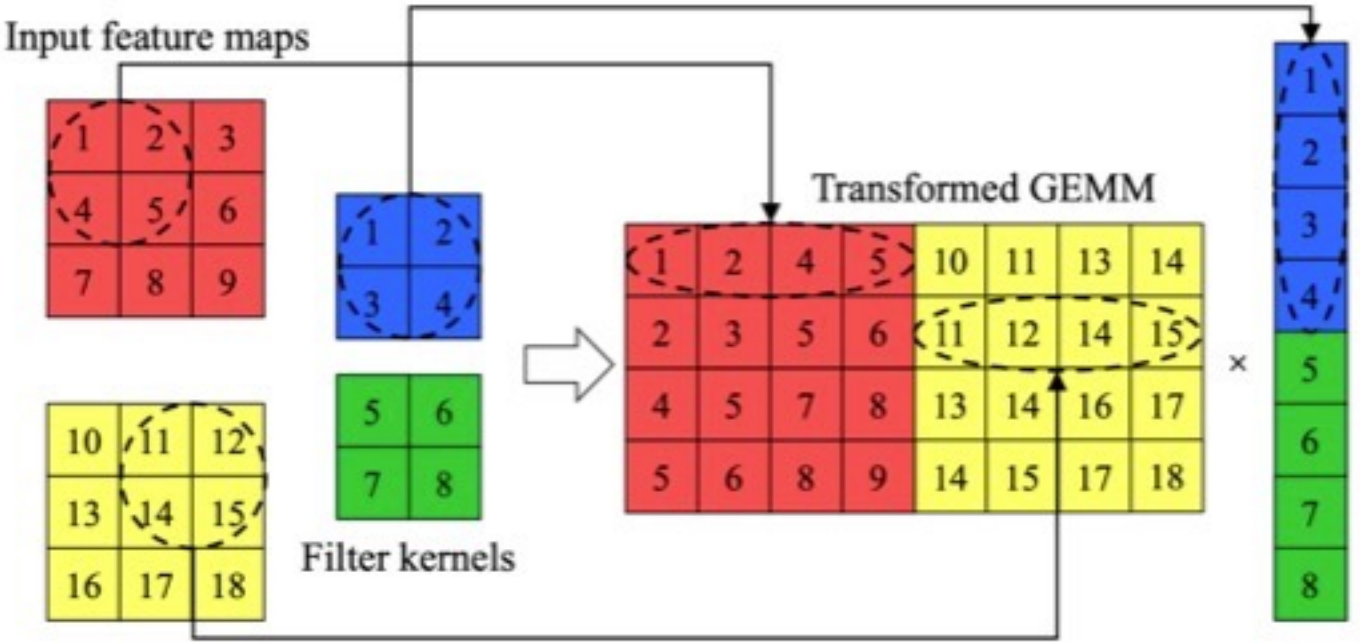


Step 2



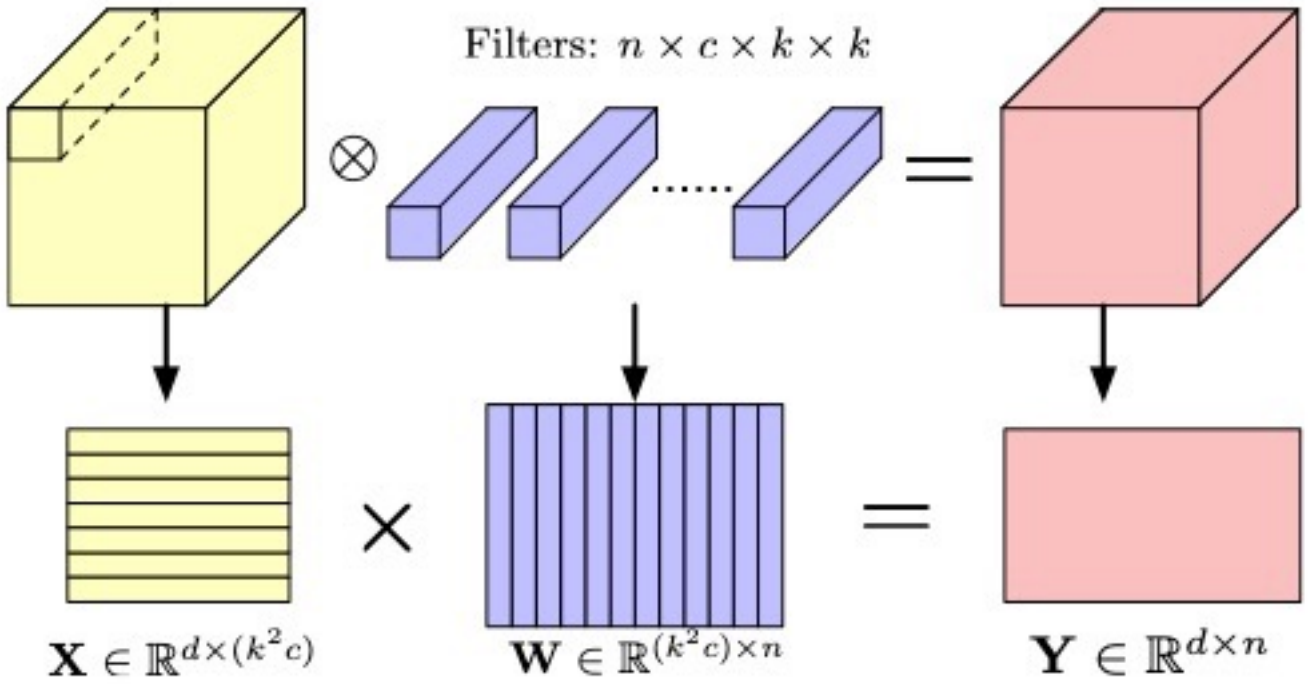
im2col

- Large extra memory overhead
- Good performance
- BLAS-friendly memory layout to enjoy SIMD/locality/parallelism
- Applicable for any convolution configuration on any platform



im2col Convolution

- **Transform the convolution to matrix multiplication**
- **Unified convolution and fully connected layer**



Direct Conv

- **Each thread for one output**

```
__global__ void convolution_kernel(float *input, float *output, float *kernel, int input_channels, int
input_height, int input_width, int output_channels, int output_height, int output_width, int
kernel_height, int kernel_width, int pad, int stride) {
    int oIdx = blockIdx.x * blockDim.x + threadIdx.x;


    if (oIdx < output_channels * output_height * output_width) {
        int oChannel = oIdx / (output_height * output_width);
        int oRow = (oIdx % (output_height * output_width)) / output_width;
        int oCol = (oIdx % (output_height * output_width)) % output_width;

        float sum = 0.0f;

        for (int iChannel = 0; iChannel < input_channels; ++iChannel) {
            for (int kRow = 0; kRow < kernel_height; ++kRow) {
                for (int kCol = 0; kCol < kernel_width; ++kCol) {
                    int iRow = oRow * stride + kRow - pad;
                    int iCol = oCol * stride + kCol - pad;

                    if (iRow >= 0 && iRow < input_height && iCol >= 0 && iCol < input_width) {
                        float inputVal = input[iChannel * input_height * input_width + iRow * input_width
                        + iCol];
                        float kernelVal = kernel[oChannel * input_channels * kernel_height * kernel_width
                        + iChannel * kernel_height * kernel_width + kRow * kernel_width + kCol];
                        sum += inputVal * kernelVal;
                    }
                }
            }
        }

        output[oIdx] = sum;
    }
}
```

Direct Conv

- **Using shared memory**

```
__global__ void convolution_shared_memory(float *input, float *output, float *kernel, int input_channels,
int input_height, int input_width, int output_channels, int output_height, int output_width, int
kernel_height, int kernel_width, int pad, int stride) {
    __shared__ float input_tile[TILE_SIZE ][TILE_SIZE];

    int output_x = blockIdx.x * blockDim.x + threadIdx.x;
    int output_y = blockIdx.y * blockDim.y + threadIdx.y;
    int output_channel = blockIdx.z * blockDim.z + threadIdx.z;

    int input_x_start = output_x * stride - pad;
    int input_y_start = output_y * stride - pad;

    float result = 0.0f;
    for (int ic = 0; ic < input_channels; ++ic) {
        for (int ky = 0; ky < kernel_height; ++ky) {
            for (int kx = 0; kx < kernel_width; ++kx) {
                int input_x = input_x_start + kx;
                int input_y = input_y_start + ky;

                if (input_x >= 0 && input_x < input_width && input_y >= 0 && input_y < input_height) {
                    input_tile[threadIdx.y + ky][threadIdx.x + kx] = input[(ic * input_height + input_y) *
                    input_width + input_x];
                } else {
                    input_tile[threadIdx.y + ky][threadIdx.x + kx] = 0.0f;
                }
            }
        }
        __syncthreads(); // 等待所有线程加载共享内存
        for (int ky = 0; ky < kernel_height; ++ky) {
            for (int kx = 0; kx < kernel_width; ++kx) {
                result += input_tile[threadIdx.y + ky][threadIdx.x + kx] * kernel[((ic * output_channels +
                output_channel) * kernel_height + ky) * kernel_width + kx];
            }
        }
        __syncthreads(); // 等待所有线程完成计算
    }
```

Direct Conv

- **Using shared memory, without additional loading**

```
float value = 0;
for (int ic = 0; ic < input_channels; ++ic) {
    if (thread_row == TILE_SIZE - 1 && thread_col == TILE_SIZE - 1) { //右下角
        for (int i = 0; i < kernel_height; i++)
        {
            for (int j = 0; j < kernel_width; j++)
            {
                input_tile[thread_row + i][thread_col + j] =
                    input[begin_pos + OFFSET(thread_row + i, thread_col + j,
                    input_width)];
            }
        }
    }
    else if (thread_row == TILE_SIZE - 1) {  //下边界
    for (int i = 0; i < kernel_height; i++)
        {
            input_tile[thread_row + i][thread_col] =
                input[begin_pos + OFFSET(thread_row + i, thread_col, input_width)];
        }

    }
    else if (thread_col == TILE_SIZE - 1) // 右边界向外延伸
    {
        for (int i = 0; i < kernel_width; i++)
        {
            input_tile[thread_row][thread_col + i] =
                input[begin_pos + OFFSET(thread_row, thread_col + i, input_width)];
        }
```

Direct Conv

- **Using shared memory, without additional loading**

```
else if (thread_col == TILE_SIZE - 1) // 右边界向外延伸
{
    for (int i = 0; i < kernel_width; i++)
    {
        input_tile[thread_row][thread_col + i] =
            input[begin_pos + OFFSET(thread_row, thread_col + i, input_width)];
    }
}
else {  //边界内数据
        input_tile[thread_row][thread_col] =
        input[begin_pos + OFFSET(thread_row, thread_col, input_width)];
    // 0号线程同时转移kernel
    if (thread_row == 0 && thread_col == 0)
    {
        for (int i = 0; i < kernel_height; i++)
        {
            for (int j = 0; j < kernel_width; j++)
            {
                s_kernel[i][j] = kernel[OFFSET(i, j, kernel_width)];
            }
        }
    }
}
__syncthreads(); // 等待所有线程加载共享内存
```

Conv+ im2col

```
__global__ void kernel_im2col_gpu2(float* __restrict__ output, float* __restrict__ input, int N, int P,
int Q, int C, int H, int W, int KH, int KW, int SH, int SW, int left, int top, size_t tcount)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    //每个线程处理一个数据到输出列矩阵中
    if (tid >= tcount) return;

    int q_idx = tid % Q; //输出矩阵中的列索引
    int idx = tid / Q;
    int p_idx = idx % P;  //输出矩阵中的行索引
    idx /= P;
    int b_idx = idx % N;  //输入数据的批次索引
    idx /= N;
    int kw_idx = idx % KW;
    idx /= KW;
    int kh_idx = idx % KH;
    idx /= KH;
    int k_idx = kw_idx + kh_idx * KW;
    int c_idx = idx % C; //输入数据的通道索引
    int w_idx = q_idx * SW - left + kw_idx;
    int h_idx = p_idx * SH - top + kh_idx;

    int n_index2 = c_idx * P * Q * N * KH * KW
        + k_idx * P * Q * N
        + b_idx * P * Q
        + p_idx * Q
        + q_idx;

    if (w_idx < 0 || w_idx >= W || h_idx < 0 || h_idx >= H) {
        output[tid] = 0.f;
    }
    else {
        int s_idx = b_idx * C * H * W + c_idx * H * W + h_idx * W + w_idx;
        output[n_index2] = input[s_idx];
    }
}
```

Conv+ im2col

- **Using shared memory, without additional loading**

```
__global__ void kernel_im2col_gpu2_shared(float* __restrict__ output, float* __restrict__
input, int N, int P, int Q, int C, int H, int W, int KH, int KW, int SH, int SW, int left,
int top, size_t tcount)
{


    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    //一个block负责输入矩阵的一小块
    __shared__ float input_block[TILE_SIZE+KERNEL_HEIGHT-1][TILE_SIZE+KERNEL_WIDTH-1];
    if (col < W && row < H) {

        //边缘线程多加载一些数据
        if(threadIdx.x == TILE_SIZE -1 ) {
                for(int remain = 0 ;remain < KERNEL_WIDTH;remain ++) {
                    input_block[threadIdx.y][threadIdx.x+remain] = input[row * W + col
                    +remain];
                }
        } else if (threadIdx.y == TILE_SIZE -1){
            for (int remain = 0;remain <KERNEL_HEIGHT;remain ++ ) {
                input_block[threadIdx.y + remain][threadIdx.x] = input[(row+remain) * W +
                col];
            }
        } else {
            input_block[threadIdx.y][threadIdx.x] = input[row * W + col];
        }
        __syncthreads();
    }
```

Conv+ im2col

```
cublasStatus_t Sgemm(
    cublasHandle_t Blas,
    cublasOperation_t AOp, cublasOperation_t BOp,
    const float* dev_A, int WidthA, int HeightA,
    const float* dev_B, int WidthB, int HeightB,
    float *dev_C,
    float Alpha = 1.0f, float Beta = 0.0f)
{
    int lda = WidthA;
    int ldb = WidthB;

    if (AOp != CUBLAS_OP_N) {
        int tmp = WidthA;
        WidthA = HeightA;
        HeightA = tmp;
    }
    if (BOp != CUBLAS_OP_N) {
        int tmp = WidthB;
        WidthB = HeightB;
        HeightB = tmp;
    }
    int m = WidthB;
    int n = HeightA;
    int k = WidthA;

    return cublasSgemm(Blas, BOp, AOp, m, n, k, &Alpha, dev_B, ldb, dev_A, lda, &Beta, dev_C, m);
}
```

Conv+ im2col

```
__global__ void kernel_col2im_gpu(float* __restrict__ output, float* __restrict__ input, int N, int K, int
P, int Q, int tcount)
{

    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid >= tcount) return;

    int q_idx = tid % Q;// Q
    int idx = tid / Q;
    int p_idx = idx % P;// P
    idx /= P;
    int k_idx = idx % K;// K
    int b_idx = idx / K;// N

    int s_idx = b_idx * K * P * Q + k_idx * P * Q + p_idx * Q + q_idx;
    int n_idx = k_idx * N * P * Q + b_idx * P * Q + p_idx * Q + q_idx;

    output[s_idx] = input[n_idx];
}
void conv2d_gemm(float* f_output, float* output, float * sgemmout, const float* weight, float* input, int
N, int K, int P, int Q, int C, int H, int W, int KH, int KW, int SH, int SW, int left, int top,
cudaStream_t stream,
    cublasHandle_t Blas, cublasOperation_t AOp, cublasOperation_t BOp)
{
    conv2d_im2col_gpu2(output, input, N, K, P, Q, C, H, W, KH, KW, SH, SW, left, top, stream);

    Sgemm(Blas, AOp, BOp, weight, KW * KH * C, K, output, P * Q * N, KH * KW * C, sgemmout);

    conv2d_col2im_gpu(f_output, sgemmout, N, K, P, Q, stream);
}
```

Average Pooling

```
__global__ void avg_pool_kernel(float *input, float *output, int input_height, int input_width, int
num_channels,
                                int kernel_size, int output_height, int output_width, int padding, int
                                stride) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid < num_channels * output_height * output_width) {
        int c = tid / (output_height * output_width);
        int spatial_tid = tid % (output_height * output_width);
        int row = spatial_tid / output_width;
        int col = spatial_tid % output_width;

        int input_row_start = row * stride - padding;
        int input_row_end = input_row_start + kernel_size;
        int input_col_start = col * stride - padding;
        int input_col_end = input_col_start + kernel_size;

        float sum = 0.0f;
        int count = 0;

        for (int i = input_row_start; i < input_row_end; i++) {
            for (int j = input_col_start; j < input_col_end; j++) {
                if (i >= 0 && i < input_height && j >= 0 && j < input_width) {
                    int input_idx = c * input_height * input_width + i * input_width + j;
                    sum += input[input_idx];
                    count++;
                }
            }
        }

        int output_idx = c * output_height * output_width + row * output_width + col;
        output[output_idx] = sum / count;
    }
}
```

Activation

```
__global__ void relu_kernel(float *input, float *output, int size) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid < size) {
        output[tid] = max(0.0f, input[tid]);
    }
}

__global__ void sigmoid_kernel(float *input, float *output, int size) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid < size) {
        output[tid] = 1.0 / (1.0 + exp(-input[tid]));
    }
}
```