CHAPTER 16

Deep learning

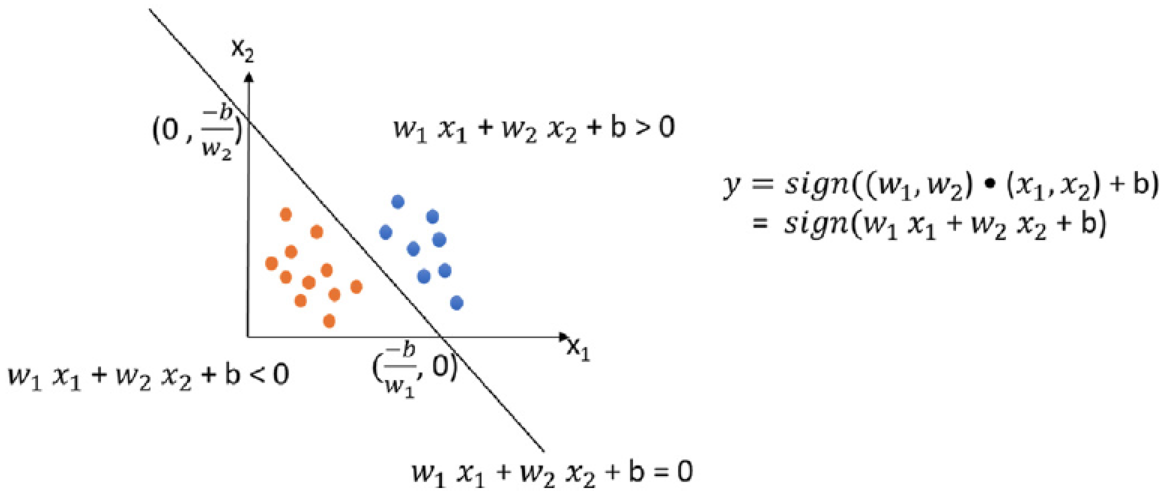

FIGURE 16.1

A perceptron linear classifier example in which the input is a two-dimensional vector.

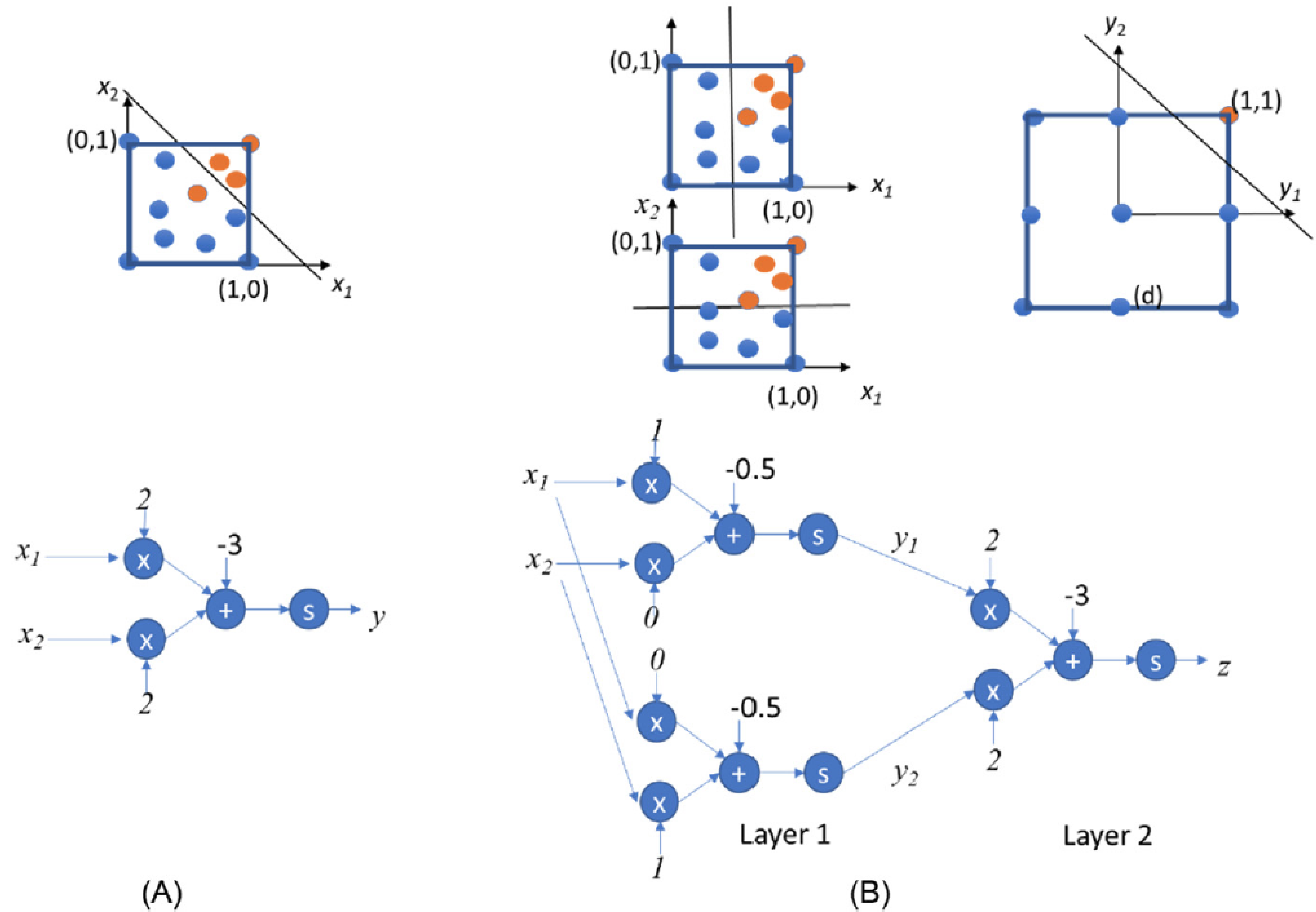

FIGURE 16.2

A multilayer perceptron example.

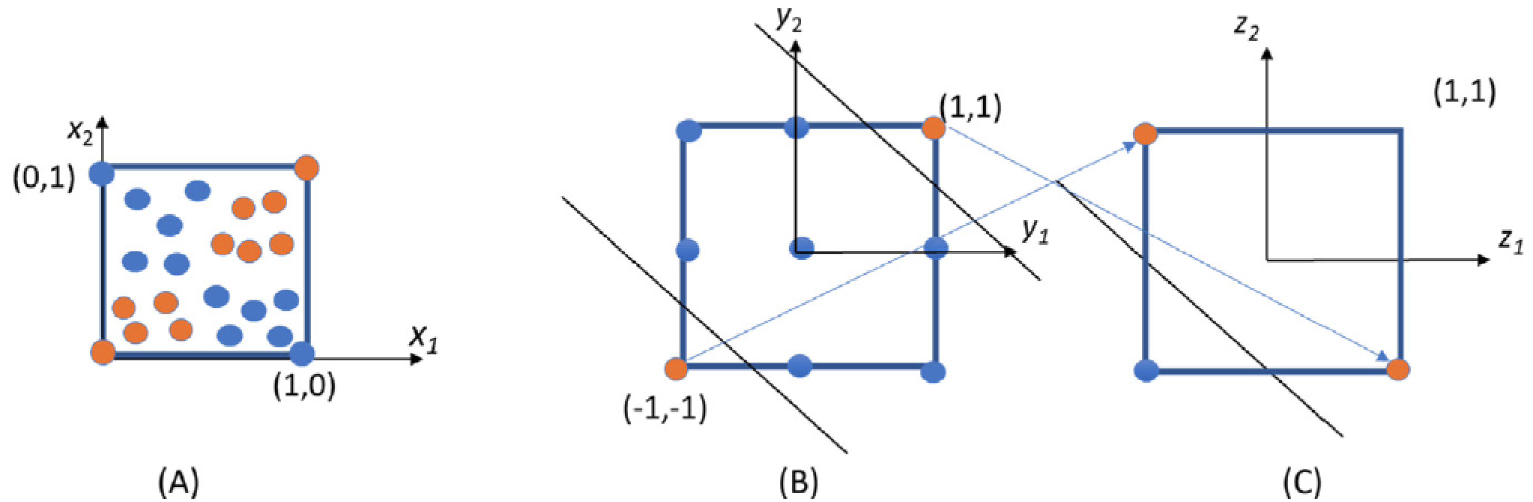

FIGURE 16.3

Need for perceptrons with more than two layers.

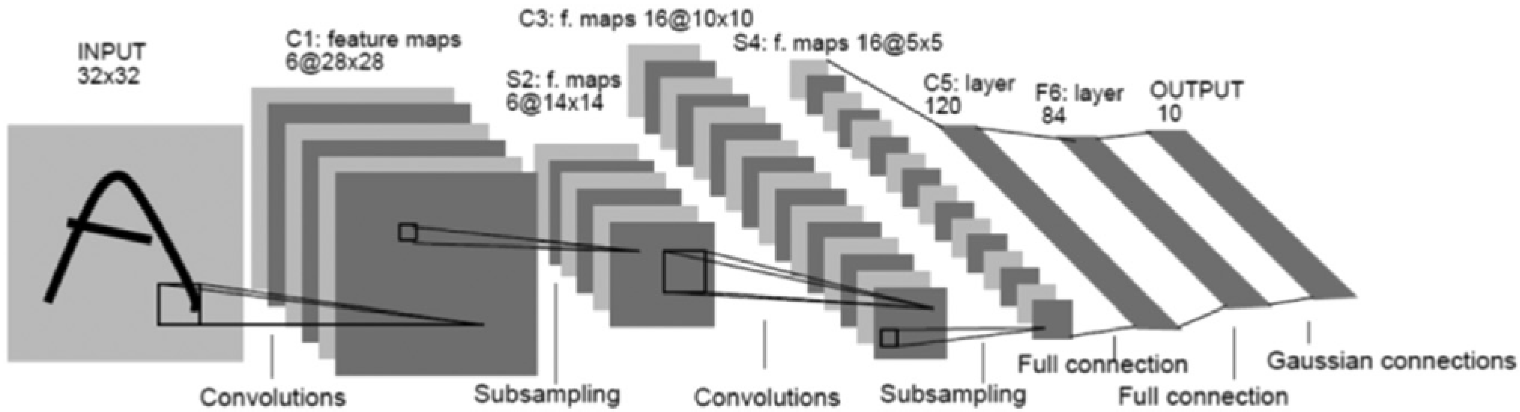

FIGURE 16.4

LeNet-5, a convolutional neural network for handwritten digit recognition. The letter A in the input should be classified as none of the ten classes (digits).

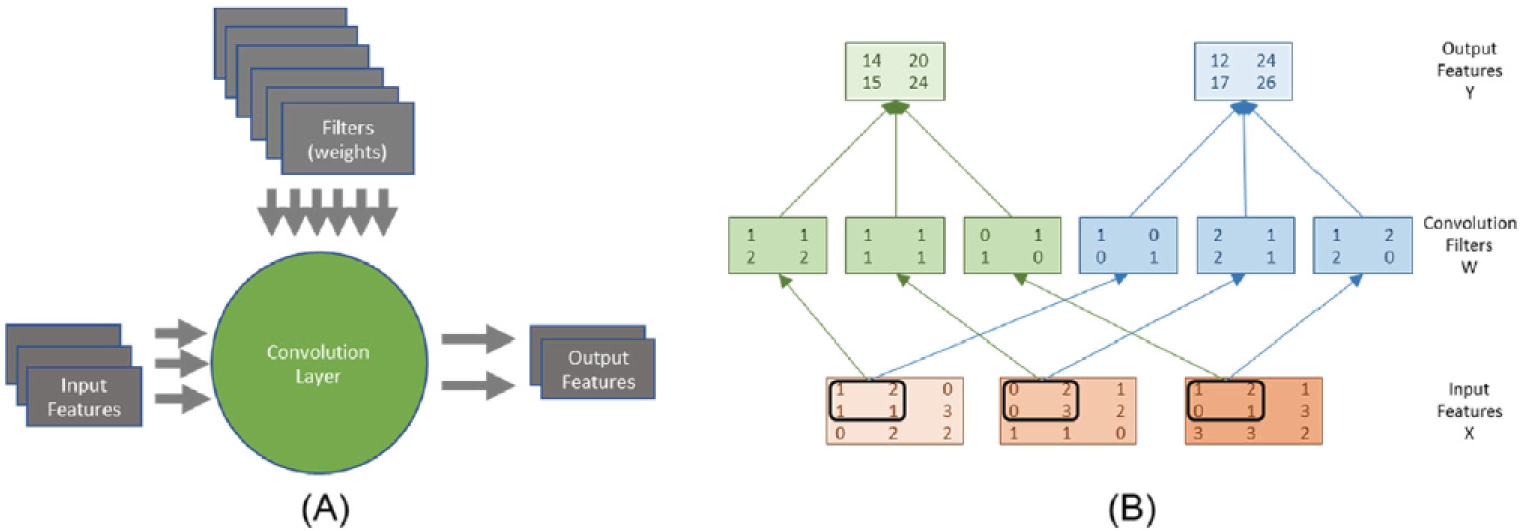

FIGURE 16.5

Forward propagation path of a convolutional layer.

```
void convLayer_forward(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 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[c, h + p, w + q] * W[m, c, p, q];
          }
 }
```

FIGURE 16.6

A C implementation of the forward propagation path of a convolutional layer.

```
void subsamplingLayer_forward(int M, int H, int W, int K, float* Y, float* S){
    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; w++) {      // this code assumes that H and W
             S[m, x, y] = 0.;                 // are multiples of K
             for(int p = 0; p < K; p++) {     // loop over KxK input samples
                for(int q = 0; q < K; q++)
                    S[m, h, w] += Y[m, K*h + p, K*w+ q] /(K*K);
             }
                          // add bias and apply non-linear activation
             S[m, h, w] = sigmoid(S[m, h, w] + b[m]);
          }
}
```

FIGURE 16.7

A sequential C implementation of the forward propagation path of a subsampling layer. The layer also includes an activation function, which is included in a convolutional layer if there is no subsampling layer after the convolutional layer.

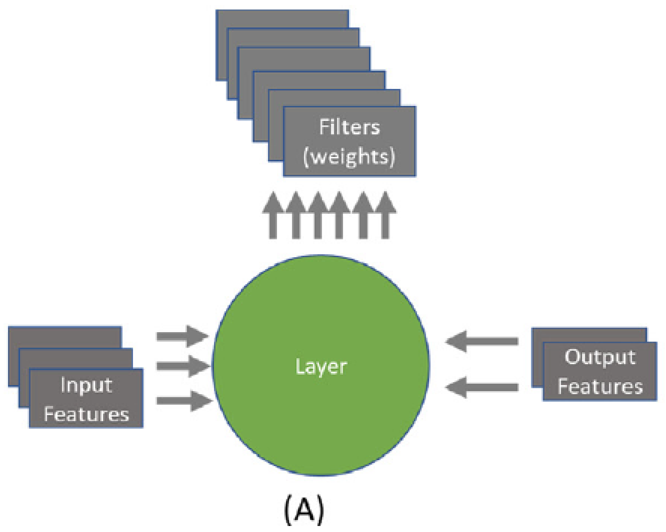

FIGURE 16.8

Backpropagation of (*A*) $\frac{\partial \mathbf{E}}{\partial \mathbf{w}}$ and (*B*) $\frac{\partial \mathbf{E}}{\partial \mathbf{x}}$ for a layer in CNN.

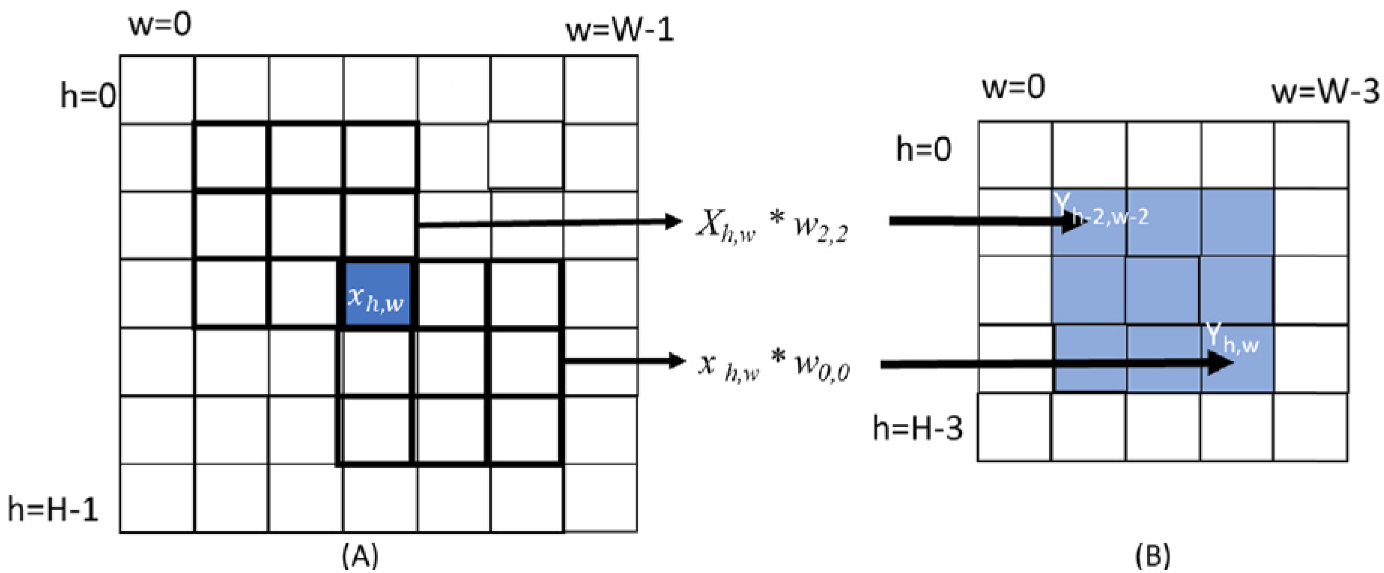

FIGURE 16.9

Convolutional layer. Backpropagation of (*A*) $\partial E/\partial w$ and (*B*) $\partial E/\partial x$.

```
void convLayer_backward_x_grad(int M, int C, int H_in, int W_in, int K,
                               float* dE_dY, float* W, float* dE_dX) {
   int H_out = H_in - K + 1;
   int W_out = W_in - K + 1;
   for(int c = 0;  c < C; c++)
      for(int h = 0; h < H_in; h++)
         for(int w = 0; w < W_in; w++)
              dE_dX[c, h, w] = 0;

   for(int m = 0;  m < M;  m++)
     for(int h = 0; h < H-1; h++)
       for(int w = 0; w < W-1; w++)
         for(int c = 0;  c < C; c++)
           for(int p = 0; p < K; p++)
             for(int q = 0; q < K; q++)
               if(h-p >= 0 && w-p >=0 && h-p < H_out and w-p < W_OUT)
                  dE_dX[c, h, w] += dE_dY[m, h-p, w-p] * W[m, c, k-p, k-q];
}
```

FIGURE 16.10

$\frac{\partial E}{\partial x}$ calculation of the backward path of a convolutional layer.

```
void convLayer_backward_w_grad(int M, int C, int H, int W, int K, float*
                               dE_dY, float* X, float* dE_dW) {
   int H_out = H - K + 1;
   int W_out = W - K + 1;
   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[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[m, c, p, q] += X[c, h+p, w+q] * dE_dY[m, c, h, w];
  }
```

FIGURE 16.11

$\frac{\partial E}{\partial w}$ calculation of the backward path of a convolutional layer.

```
void convLayer_batched(int N, 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 n = 0;  n < N;  n++)      // for each sample in the mini-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[n, 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[n,m,h,w] = Y[n,m,h,w] + X[n, c, h+p, w+q]*W[m,c,p,q];
            }
      }
```

FIGURE 16.12

Forward path of a convolutional layer with minibatch training.

```
# define TILE_WIDTH 16
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
T = H_grid * W_grid;
dim3 blockDim(TILE_WIDTH, TILE_WIDTH, 1);
dim3 gridDim(M, T, N);
ConvLayerForward_Kernel<<< gridDim, blockDim>>>(…);
```

FIGURE 16.13

Host code for launching a convolutional layer kernel.

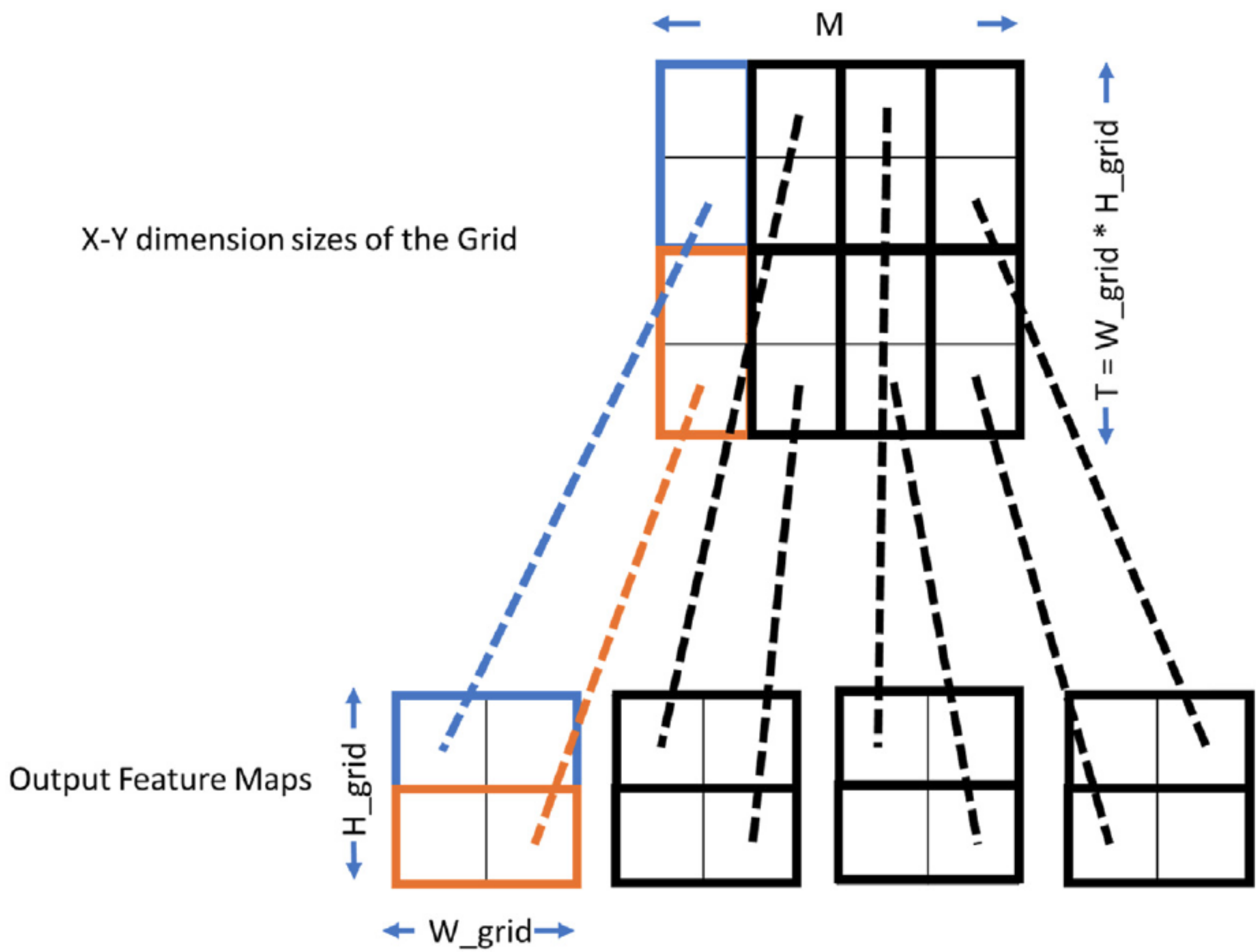

FIGURE 16.14

Mapping output feature map tiles to blocks in the *X-Y* dimension of the grid.

```
__global__ void
ConvLayerForward_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;
    int n = blockIdx.z;
    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[n, c, h + p, w + q] * W[m, c, p, q];
    }
    Y[n, m, h, w] = acc;
}
```

FIGURE 16.15

Kernel for the forward path of a convolutional layer.

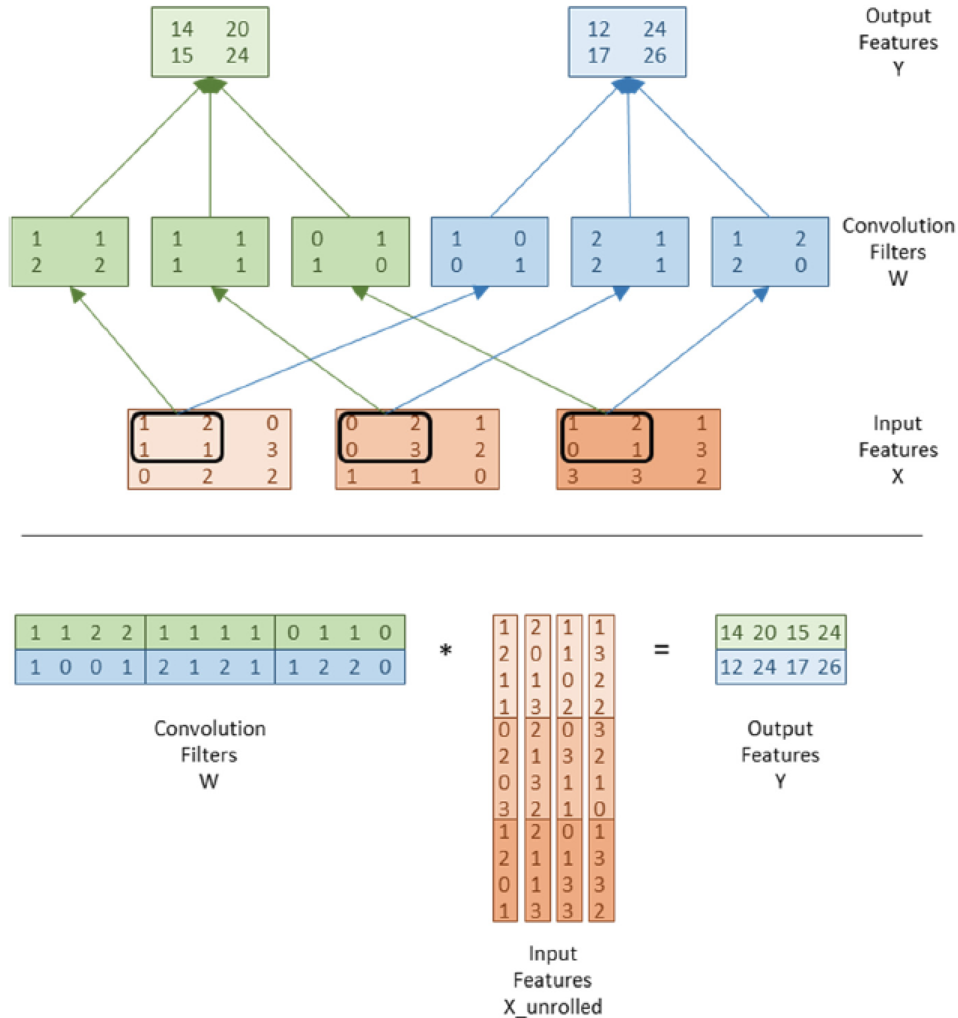

FIGURE 16.16

Formulation of convolutional layer as GEMM.

```
void unroll(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 c = 0; c < C; c++) {
        // Beginning row index of the section for channel C input feature
        // map in the unrolled matrix
        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++) {
                    int h_unroll = w_base + p*K + q;
                    for(int w = 0; w < W_out; w ++) {
                        int w_unroll = h * W_out + w;
                        X_unroll[h_unroll, w_unroll) = X(c, h + p, w + q);
                    }
                }
            }
        }
    }
}
```

FIGURE 16.17

A C function that generates the unrolled X matrix. The array accesses are in multidimensional indexing form for clarity and need to be linearized for the code to be compilable.

```
__global__ void
unroll_Kernel(int C, int H, int W, int K, float* X, float* X_unroll) {
    int t = blockIdx.x * blockDim.x + threadIdx.x;
    int H_out = H – K + 1;
    int W_out = W – K + 1;
    // Width of the unrolled input feature matrix
    int W_unroll = H_out * W_out;
    if (t < C * W_unroll) {
        // Channel of the input feature map being collected by the thread
        int c = t / W_unroll;
        // Column index of the unrolled matrix to write a strip of
        // input elements into (also, the linearized index of the output
        // element for which the thread is collecting input elements)
        int w_unroll = t % W_unroll;
        // Horizontal and vertical indices of the output element
        int h_out = w_unroll / W_out;
        int w_out = w_unroll % W_out;
        // Starting row index for the unrolled matrix section for channel c
        int w_base = c * K * K;
        for(int p = 0; p < K; p++)
            for(int q = 0; q < K; q++) {
                // Row index of the unrolled matrix for the thread to write
                // the input element into for the current iteration
                int h_unroll = w_base + p*K + q;
                X_unroll[h_unroll, w_unroll] = X[c, h_out + p, w_out + q];
        }
    }
}
```

FIGURE 16.18

A CUDA kernel implementation for unrolling input feature maps. The array accesses are in multidimensional indexing form for clarity and need to be linearized for the code to be compilable.

Table 16.1 Convolution parameters for CUDNN. Note that the CUDNN naming convention is slightly different from what we used in previous sections.

Parameter	Meaning
N	Number of images in minibatch
C	Number of input feature maps
H	Height of input image
W	Width of input image
K	Number of output feature maps
R	Height of filter
S	Width of filter
u	Vertical stride
v	Horizontal stride
pad_h	Height of zero padding
pad_w	Width of zero padding