Chapter-16
Application case study— machine learning

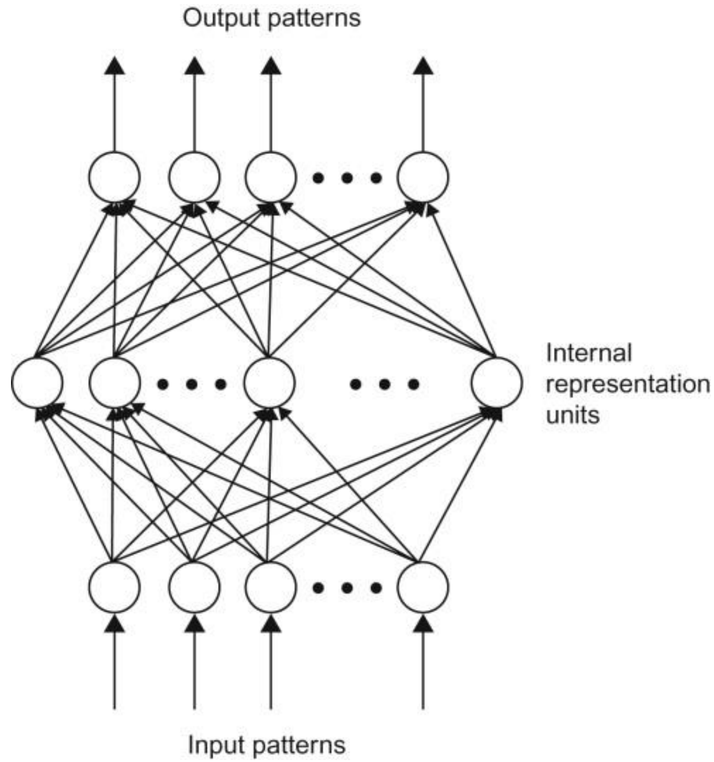

FIGURE 16.1: A multilayer feedforward network.

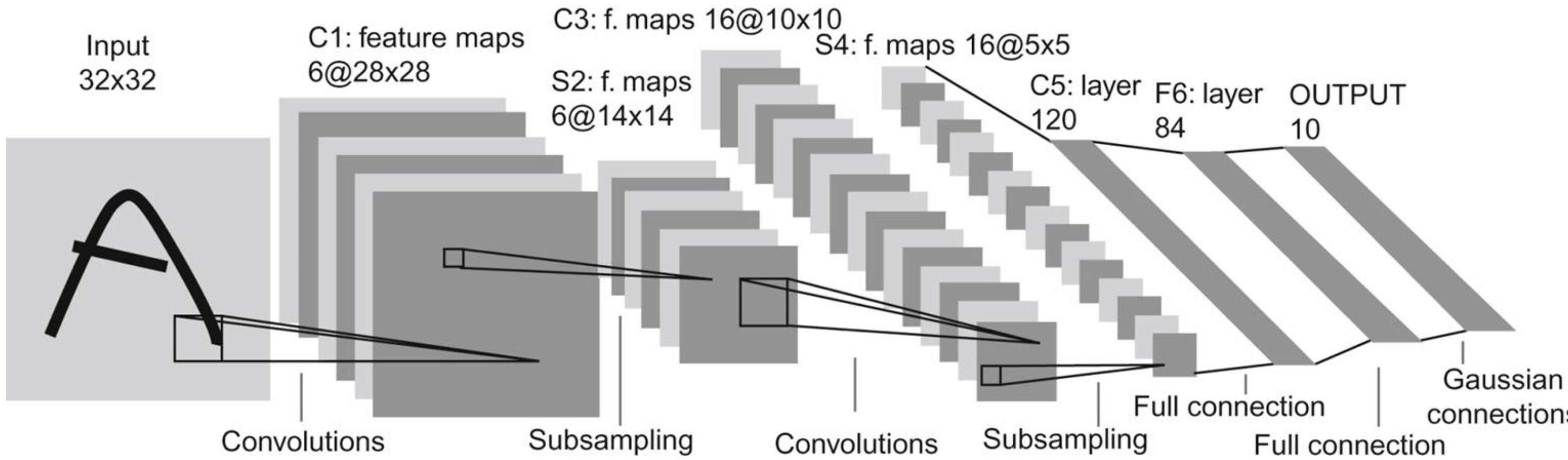

FIGURE 16.2: LeNet-5, a convolutional neural network for handwritten digit recognition.

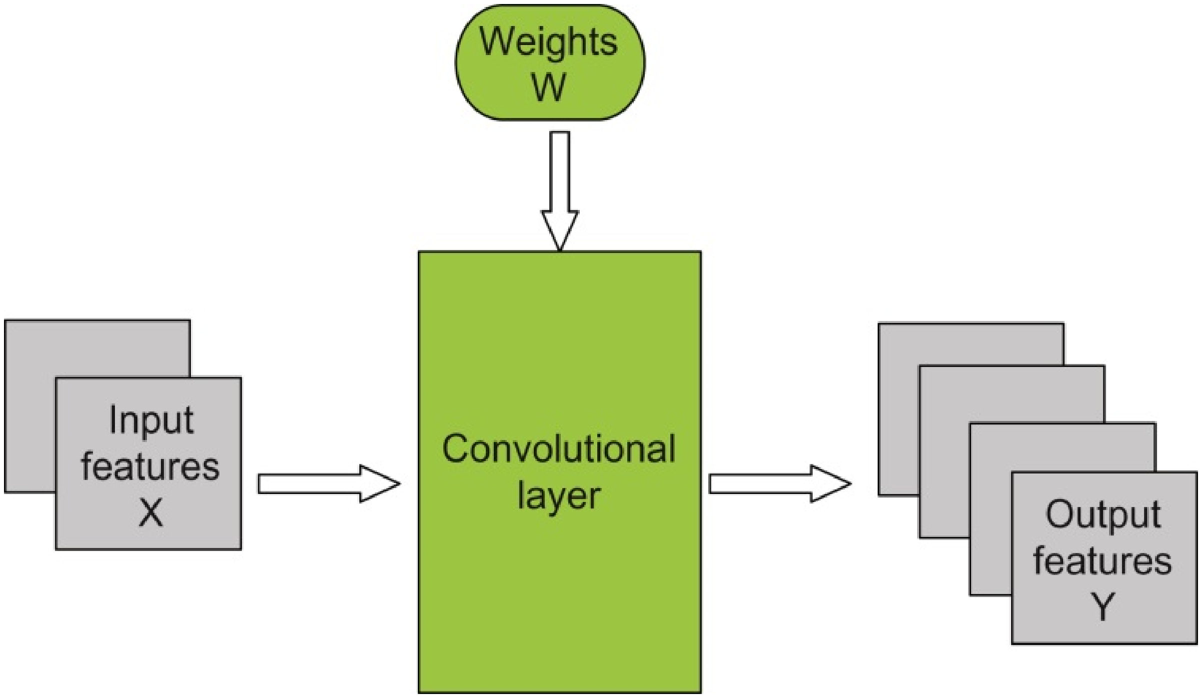

FIGURE 16.3: Overview of the forward propagation path of a convolution layer.

```
void convLayer_forward(int M, int C, int H, int W, int K, float* X, float* W, float* Y)
{
   int m, c, h, w, p, q;
   int H_out = H – K + 1;
   int W_out = W – K + 1;

   for(m = 0;  m < M;  m++)            // for each output feature maps
     for(h = 0; h < H_out; h++)        // for each output element
       for(w = 0; w < W_out; w++) {
         Y[m, h, w] = 0;
         for(c = 0;  c < C; c++)        // sum over all input feature maps
           for(p = 0; p < K; p++)       // KxK  filter
             for(q = 0; q < K; q++)
               Y[m, h, w] +=  X[c, h + p, w + q] * W[m, c, p, q];
       }
}
```

FIGURE 16.4: A sequential implementation of the forward propagation path of a convolution layer.

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

FIGURE 16.5: A sequential C implementation of the forward propagation path of a subsampling layer.

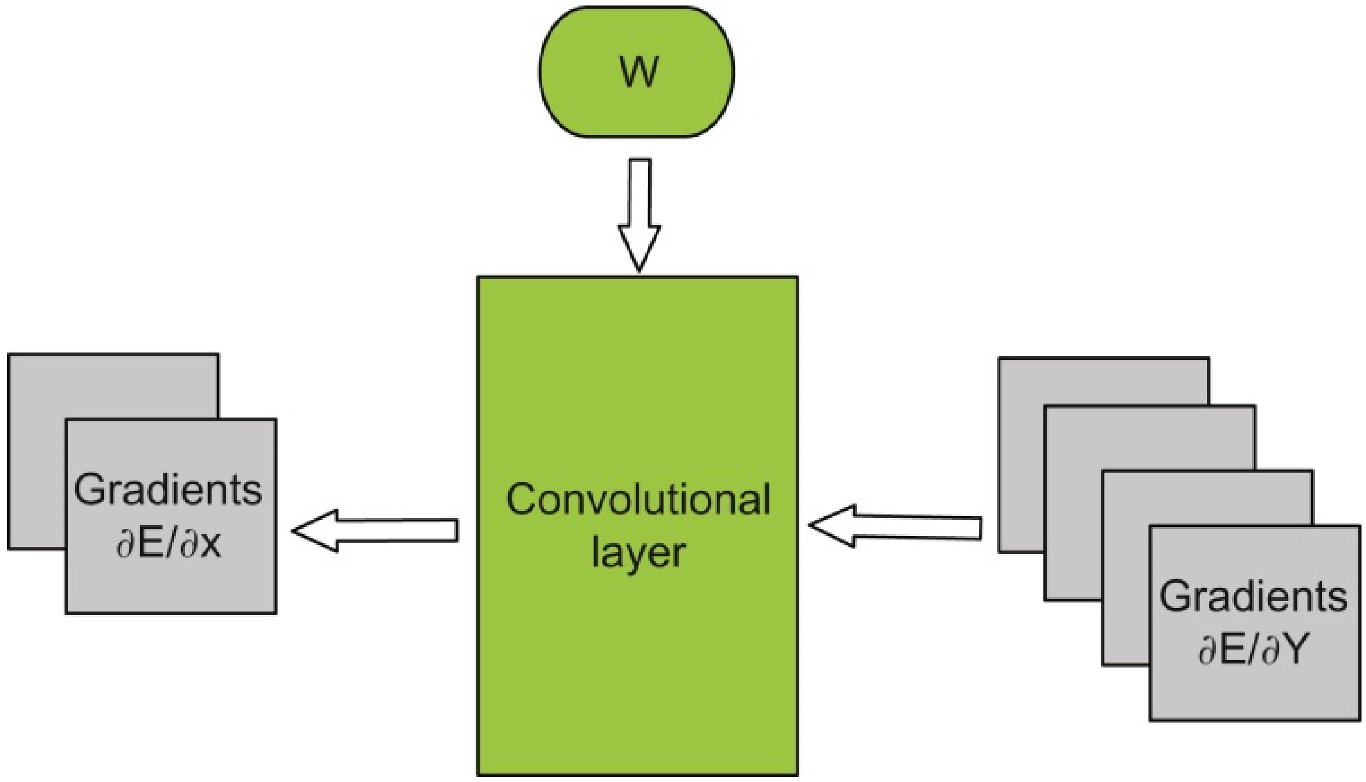

FIGURE 16.6: Convolutional layer: Backpropagation of ∂E/∂X.

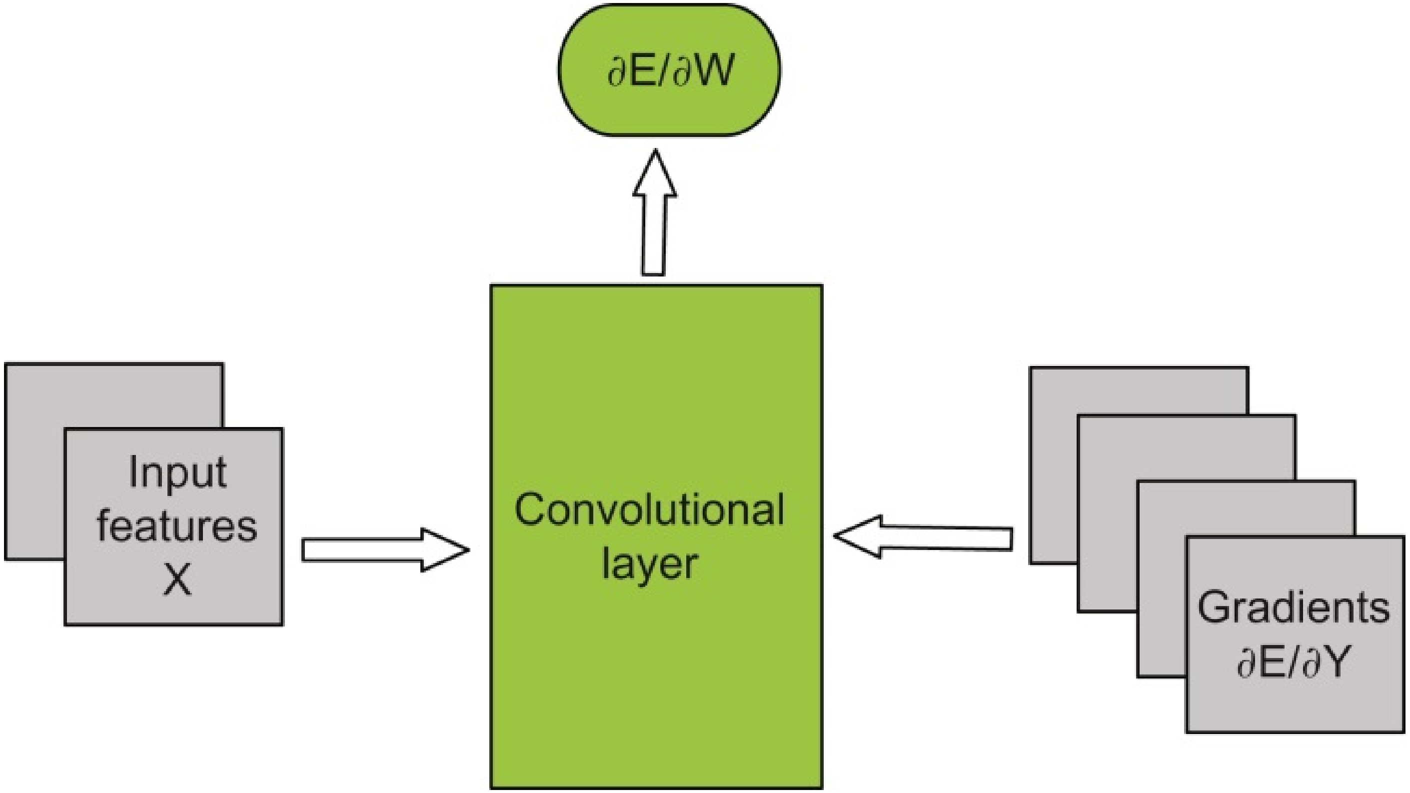

FIGURE 16.7: Convolutional layer: Backpropagation of ∂E/∂w.

```
void convLayer_backward_xgrad(int M, int C, int H_in, int W_in, int K,
        float* dE_dY, float* W, float* dE_dX)
{
  int m, c, h, w, p, q;
  int H_out = H_in – K + 1;
  int W_out = W_in – K + 1;
  for(c = 0;  c < C; c++)
   for(h = 0; h < H_in; h++)
    for(w = 0; w < W_in; w++)
     dE_dX[c, h, w] = 0.;

  for(m = 0;  m < M;  m++)
   for(h = 0; h < H_out; h++)
    for(w = 0; w < W_out; w++)
     for(c = 0;  c < C; c++)
      for(p = 0; p < K; p++)
       for(q = 0; q < K; q++)
        dE_dX[c, h + p, w + q] += dE_dY[m, h, w] * W[m, c, p, q];
}
```

FIGURE 16.8: dE/dX calculation of the backward path of a convolution layer.

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

  for(m = 0;  m < M;  m++)
   for(h = 0; h < H_out; h++)
    for(w = 0; w < W_out; w++)
     for(c = 0;  c < C; c++)
      for(p = 0; p < K; p++)
       for(q = 0; q < K; q++)
        dE_dW[m, c, p, q] += X[c, h + p, w + q] * dE_dY[m, c, h, w];
}
```

FIGURE 16.9: dE/dW calculation of the backward path of a convolutional layer.

```
void convLayer_forward(int N, int M, int C, int H, int W, int K, float* X, float* W, float* Y)
{
   int n, m, c, h, w, p, q;
   int H_out = H– K + 1;
   int W_out = W – K + 1;
   for(n = 0;  n < N;  n++)                // for each sample in the mini-batch
    for(m = 0;  m < M;  m++)               // for each output feature maps
     for(h = 0; h < H_out; h++)            // for each output element
      for(w = 0; w < W_out; w++) {
       Y[n, m, h, w] = 0;
       for (c = 0;  c < C; c++)            // sum over all input feature maps
        for (p = 0; p < K; p++)            // KxK  filter
         for (q = 0; q < K; q++)
           Y[n, m, h, w]  += X[n, c, h + p, w + q] * W[m, c, p, q];
      }
}
```

FIGURE 16.10: Forward path of a convolutional layer with mini-batch training.

```
void convLayer_forward(int N, int M, int C, int H, int W, int K, float* X, float* W, float* Y)
{
   int n, m, c, h, w, p, q;
   int H_out = H - K + 1;
   int W_out = W - K + 1;
   parallel_for(n = 0;  n < N;  n++)
    parallel_for (m = 0;  m < M;  m++)
      parallel_for(h = 0; h < H_out; h++)
       parallel_for(w = 0; w < W_out; w++) {
         Y[n, m, h, w] = 0;
         for (c = 0;  c < C; c++)
          for (p = 0; p < K; p++)
            for (q = 0; q < K; q++)
             Y[n, m, h, w] += X[n, c, h + p, w + q] * W[m, c, p, q];
       }
}
```

FIGURE 16.11: Parallelization of the forward path of a convolutional layer with mini-batch training.

```
__global__ void
ConvLayerForward_Kernel(int C, int W_grid, intK, float* X, float* W, float* Y)
{
   int n, m, h, w, c, p, q;
   n = blockId.x;
   m = blockId.y;
   h = blockId.z / W_grid + threadId.y;
   w = blockId.z % W_grid + threadId.x;
   float acc = 0.;
   for (c = 0;  c < C; c++) {            // sum over all input channels
    for (p = 0; p < K; p++)              // loop over KxK  filter
     for (q = 0; q < K; q++)
      acc = acc + X[n, c, h + p, w + q] * W[m, c, p, q];
   }
   Y[n, m, h, w] = acc;
}
```

FIGURE 16.12: Kernel for the forward path of a convolution layer.

```
__global__ void
ConvLayerForward_Kernel(int C, int W_grid, int K, float* X, float* W, float* Y)
{
   int n, m, h0, w0, h_base, w_base, h, w;
   int X_tile_width = TILE_WIDTH + K-1;
   extern __shared__ float shmem[];
   float* X_shared = &shmem[0];
   float* W_shared = &shmem[X_tile_width * X_tile_width];
   n = blockIdx.x;
   m = blockIdx.y;
   h0 = threadIdx.x;  // h0 and w0 used as shorthand for threadIdx.x and threadIdx.y
   w0 = threadIdx.y;
   h_base = (blockIdx.z / W_grid) * TILE_SIZE; // vertical base out data index for the block
   w_base = (blockIdx.z % W_grid) * TILE_SIZE; // horizontal base out data index for the block
   h = h_base+ h0;
   w = w_base+ w0;

   float acc = 0.;
   int c, i, j, p, q;
   for (c = 0; c < C; c++) {                              // sum over all input channels

      if (( h0 < K) && ( w0 < K))
       W_shared[h0, w0]= W [m, c, h0, w0];  // load weights for W [m, c,..],
      __syncthreads()                        // h0 and w0 used as shorthand for threadIdx.x
                                             // and threadIdx.y


      for (i = h; i < h_base+ X_tile_width; i += TILE_WIDTH) {
       for (j = w; j < w_base + X_tile_width; j += TILE_WIDTH)
         X_shared[i -h_base, j -w_base] = X[n, c, h, w]
      }
                                             // load tile from X[n, c,…]into shared memory
      __syncthreads();
      for (p = 0; p < K; p++) {
       for (q = 0; q < K; q++)
         acc = acc + X_shared[h + p, w + q] * W_shared[p, q];
      }
      __syncthreads();
   }
   Y[n, m, h, w] = acc;
}
```

FIGURE 16.13: A kernel that uses shared memory tiling to reduce the global memory traffic of the forward path of the convolutional layer.

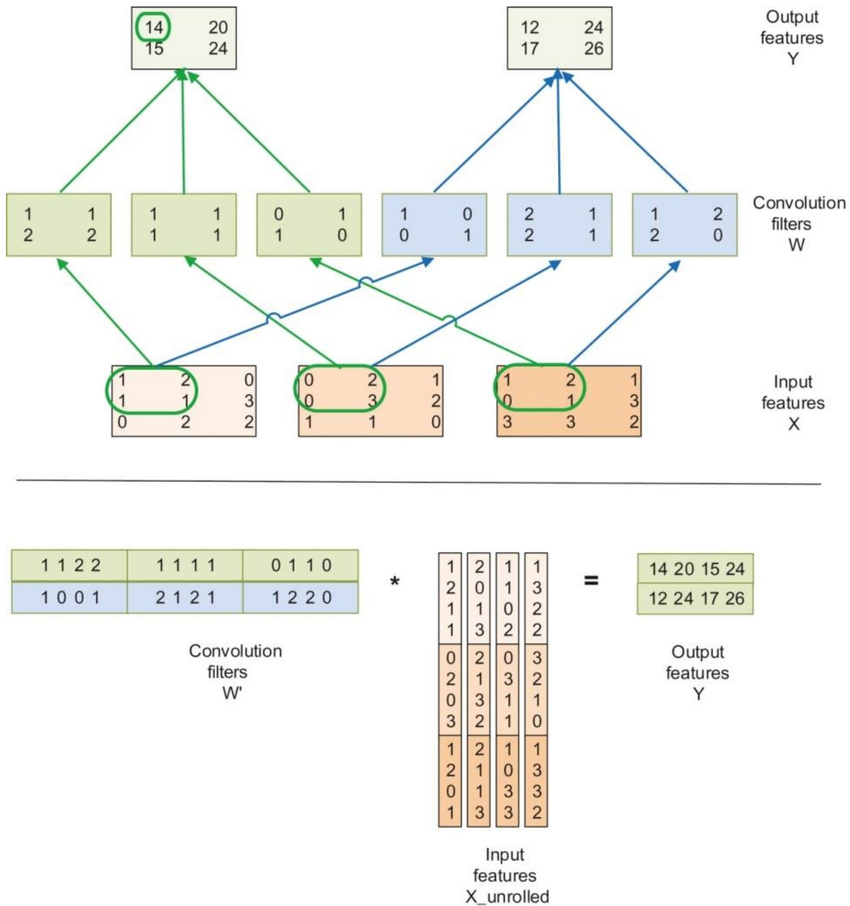

FIGURE 16.14: Reduction of a convolutional layer to GEMM.

```
void convLayer_forward(int N, int M, int C, int H, int W, int K, float* X, float* W_unroll, float* Y)
{
  int W_out = W– K + 1;
  int H_out = H– K + 1;
  int W_unroll = C * K * K;
  int H_unroll = H_out * W_out;
  float* X_unrolled = malloc(W_unroll * H_unroll * sizeof(float));
  for (int n=0; n < N; n++) {
    unroll(C, H, W, K, n,X, X_unrolled);
    gemm(H_unroll, M, W_unroll, X_unrolled, W, Y[n]);
  }
}
```

FIGURE 16.15: Implementing the forward path of a convolutional layer with matrix multiplication.

```
void unroll(int C, int H, int W, int K, float* X, float* X_unroll)
{
   int c, h, w, p, q, w_base, w_unroll, h_unroll;
   int H_out = H– K + 1;
   int W_out = W– K + 1;
   for(c = 0; c < C; c++) {
    w_base = c * (K*K);
    for(p = 0; p < K; p++)
     for(q = 0; q < K; q++) {
      for(h = 0; h <  H_out; h++)
       for(w = 0; w < W_out; w++){
         w_unroll = w_base + p * K + q;
         h_unroll = h * W_out + w;
         X_unroll(h_unroll, w_unroll) = X(c, h + p, w + q);
       }
    }
}
```

FIGURE 16.16: The function that generates the unrolled X matrix.

```
void unroll_gpu(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;
   int num_threads = C * H_out * W_out;
   int num_blocks = ceil((C * H_out * W_out) / CUDA MAX_NUM_THREADS);
   unroll_Kernel<<<num_blocks, CUDA MAX_NUM_THREADS>>>();
}
```

FIGURE 16.17: Host code for invoking the unroll kernel.

```
__global__ void unroll_Kernel(int C, int H, int W, int K, float* X, float* X_unroll)
{
  int c, s, h_out, w-out, h_unroll, w_base, p, q;
  int t = blockId.x * CUDA MAX_NUM_THREADS + threadId.x;
  int H_out = H– K + 1;
  int W_out = W– K + 1;
  int W_unroll = H_out * W_out;

  if (t < C * W_unroll) {
   c = t / W_unroll;
   s = t % W_unroll;
   h_out = s / W_out;
   w_out = s % W_out;
   h_unroll = h_out * W_out + w_out;
   w_base = c * K * K;
   for(p = 0; p < K; p++)
    for(q = 0; q < K; q++) {
     w_unroll = w_base + p * K + q;
     X_unroll(h_unroll, w_unroll) = X(c, h_out + p, w_out + q);
    }
  }
}
```

FIGURE 16.18: High-performance implementation of the unroll kernel.