
E0294: SYSTEMS FOR MACHINE LEARNING

Assignment #1

Author

Anmol Mahendra Kumar Gupta
CSA, IISc
23-jan-2026

Contents

1 Problem Solution	3
1.1 Parameters	3
1.2 Part (a): 7-Loop Naive Implementation	3
1.3 Part (b): Flattened Matrix Implementation	4
1.4 Part (c): Equality Verification	6
1.5 Part (d): Performance Measurement	7
1.5.1 Naive Implementation Results:	7
1.5.2 Flattened Implementation Results:	7
1.5.3 Performance Comparison Table:	7
1.6 Part (e): Analysis	8
2 Problem Solution	8
2.1 part (a)	8
2.2 part (b)	8
2.2.1 Input:	8
2.2.2 Kernels:	8
2.2.3 X' construction :	9
2.2.4 W' construction:	9
2.2.5 Matrix multiplication:	10
2.2.6 Reshape to Y :	10

1 Problem Solution

1.1 Parameters

- $N = 8$ (batch size)
- $C = 4$ (input channels)
- $H = 28, W = 28$ (input dimensions)
- $R = 5, S = 5$ (kernel dimensions)
- $M = 16$ (output channels)
- Stride = 2, no padding
- Average pooling: 2×2 window, stride 2

1.2 Part (a): 7-Loop Naive Implementation

Question: Implement the 7-loop naive implementation of convolution.

Listing 1: 7-loop naive implementation

```
1 def problem_a():
2     random.seed(42)
3     N, C, H, W = 8, 4, 28, 28
4     R, S, M = 5, 5, 16
5     stride = 2
6     E = (H - R) // stride + 1 # 12
7     F = (W - S) // stride + 1 # 12
8
9     # Initialize tensors with random signed values
10    I = [[[random_signed() for _ in range(W)]
11          for _ in range(H)] for _ in range(C)]
12          for _ in range(N)]
13    Wgt = [[[[random_signed() for _ in range(S)]
14              for _ in range(R)] for _ in range(C)]
15              for _ in range(M)]
16
17    # 7-loop convolution
18    O_conv = [[[0.0 for _ in range(F)]
19                  for _ in range(E)] for _ in range(M)]
20                  for _ in range(N)]
21
22    for n in range(N):
23        for m in range(M):
24            for x in range(E):
25                for y in range(F):
26                    val = 0.0
27                    for i in range(R):
28                        for j in range(S):
29                            for k in range(C):
```

```

30             input_x = x*stride + i
31             input_y = y*stride + j
32             val += I[n][k][input_x][input_y] *
33                 Wgt[m][k][i][j]
34             O_conv[n][m][x][y] = val
35
36     # Average pooling (2x2, stride 2)
37     pool_size = 2
38     pool_stride = 2
39     E_pool = (E - pool_size) // pool_stride + 1 # 6
40     F_pool = (F - pool_size) // pool_stride + 1 # 6
41
42     O_final = [[[0.0 for _ in range(F_pool)]
43                  for _ in range(E_pool)] for _ in range(M)]
44                  for _ in range(N)]
45
46     for n in range(N):
47         for m in range(M):
48             for x in range(E_pool):
49                 for y in range(F_pool):
50                     total = 0.0
51                     for i in range(pool_size):
52                         for j in range(pool_size):
53                             conv_x = x*pool_stride + i
54                             conv_y = y*pool_stride + j
55                             total += O_conv[n][m][conv_x][conv_y]
56             O_final[n][m][x][y] = total/(pool_size*pool_size)
57
58     return O_final

```

1.3 Part (b): Flattened Matrix Implementation

Question: Implement the convolution by flattening.

Listing 2: Flattened matrix implementation

```

1 def problem_b():
2     random.seed(42)
3     N, C, H, W = 8, 4, 28, 28
4     R, S, M = 5, 5, 16
5     stride = 2
6     pool_size, pool_stride = 2, 2
7
8     E = (H - R) // stride + 1 # 12
9     F = (W - S) // stride + 1 # 12
10    E_pool = (E - pool_size) // pool_stride + 1 # 6
11    F_pool = (F - pool_size) // pool_stride + 1 # 6
12
13    # Initialize tensors
14    I = [[[random_signed() for _ in range(W)]
```

```

15     for _ in range(H)] for _ in range(C)]
16     for _ in range(N)]
17 Wgt = [[[random_signed() for _ in range(S)]
18         for _ in range(R)] for _ in range(C)]
19         for _ in range(M)]
20
21 patch_size = R * S * C # 100
22 num_patches = E * F # 144
23
24 # Flatten weights: M * (R*S*C)
25 W_flat = [[0.0 for _ in range(patch_size)] for _ in range(M)]
26 for m in range(M):
27     idx = 0
28     for k in range(C):
29         for i in range(R):
30             for j in range(S):
31                 W_flat[m][idx] = Wgt[m][k][i][j]
32                 idx += 1
33
34 O_final = [[[0.0 for _ in range(F_pool)]
35             for _ in range(E_pool)] for _ in range(M)]
36             for _ in range(N)]
37
38 for n in range(N):
39     # Extract patches: (R*S*C) * (E*F)
40     X_flat = [[0.0 for _ in range(num_patches)]
41                 for _ in range(patch_size)]
42     for x in range(E):
43         for y in range(F):
44             col_idx = x * F + y
45             idx = 0
46             for k in range(C):
47                 for i in range(R):
48                     for j in range(S):
49                         input_x = x*stride + i
50                         input_y = y*stride + j
51                         X_flat[idx][col_idx] = I[n][k][input_x][
52                             input_y]
53                         idx += 1
54
55     # Matrix multiplication: M * (E*F)
56     O_flat = [[0.0 for _ in range(num_patches)] for _ in range(M)]
57     for m in range(M):
58         for col in range(num_patches):
59             val = 0.0
60             for k in range(patch_size):
61                 val += W_flat[m][k] * X_flat[k][col]
62             O_flat[m][col] = val

```

```

63     # Reshape to M * E * F
64     O_conv = [[[0.0 for _ in range(F)]
65                 for _ in range(E)] for _ in range(M)]
66     for m in range(M):
67         for x in range(E):
68             for y in range(F):
69                 col_idx = x * F + y
70                 O_conv[m][x][y] = O_flat[m][col_idx]
71
72     # Average pooling
73     for m in range(M):
74         for x in range(E_pool):
75             for y in range(F_pool):
76                 total = 0.0
77                 for i in range(pool_size):
78                     for j in range(pool_size):
79                         conv_x = x*pool_stride + i
80                         conv_y = y*pool_stride + j
81                         total += O_conv[m][conv_x][conv_y]
82                 O_final[n][m][x][y] = total/(pool_size*pool_size)
83
84     return O_final

```

1.4 Part (c): Equality Verification

Question: Show that both cases produce equal output.

Listing 3: Equality verification

```

1  # verify.py
2  from problem_a import problem_a
3  from problem_b import problem_b
4
5  print("-----Part (c): Equality Checking-----")
6  O_naive = problem_a()
7  O_flat = problem_b()
8
9  N, M = 8, 16
10 E_pool, F_pool = 6, 6
11
12 max_diff = 0.0
13 for n in range(N):
14     for m in range(M):
15         for x in range(E_pool):
16             for y in range(F_pool):
17                 diff = abs(O_naive[n][m][x][y] - O_flat[n][m][x][y])
18                 if diff > max_diff:
19                     max_diff = diff
20
21 print(f"Max absolute difference: {max_diff:.10f}")

```

```
22 | print(f"Outputs {'MATCH' if max_diff < 1e-7 else 'DO NOT MATCH'}")|
```

Output:

```
-----Part (c): Equality Checking-----
Max absolute difference: 0.0000000000
Outputs MATCH
```

1.5 Part (d): Performance Measurement

Question: Obtain and report number of instructions and execution time using perf command.

```
# Command for naive implementation:
perf stat -e instructions,task-clock python3 problem_a.py

# Command for flattened implementation:
perf stat -e instructions,task-clock python3 problem_b.py
```

Performance Results:

1.5.1 Naive Implementation Results:

```
Problem a checksum: -73.898123
```

```
Performance counter stats for 'python3 problem_a.py':
      5,090,119,936      instructions
                  273.04 msec task-clock
                  0.274051658 seconds time elapsed
```

1.5.2 Flattened Implementation Results:

```
Problem b checksum: -73.898123
```

```
Performance counter stats for 'python3 problem_b.py':
      2,468,146,759      instructions
                  133.44 msec task-clock
                  0.134470922 seconds time elapsed
```

1.5.3 Performance Comparison Table:

Metric	Naive	Flattened	Improvement
Instructions	5.09×10^9	2.47×10^9	51.5% reduction
Execution Time	273.04 ms	133.44 ms	51.1% faster
CPU Utilization	99.6%	99.2%	Similar

1.6 Part (e): Analysis

Question: Analyze and comment on the above results.

- **Instruction Count:** Flattened approach executes 2.62 billion fewer instructions (51.5% reduction)
- **Execution Time:** Flattened approach is 51.1% faster
- **Equality:** Both implementations produce identical results
- **Memory Access:** Flattened approach has better cache locality and contiguous access
- **Complexity:** Naive: $O(N \times M \times E \times F \times R \times S \times C)$, Flattened: same complexity but better constant factors

2 Problem Solution

2.1 part (a)

$$H_{\text{in}} = 3, W_{\text{in}} = 3, C_{\text{in}} = 2$$

$$K_h = 2, K_w = 2, C_{\text{out}} = 3$$

$$P = 0, S = 1$$

$$\begin{aligned} H_{\text{out}} &= \left\lfloor \frac{H_{\text{in}} - K_h}{S} \right\rfloor + 1 = \left\lfloor \frac{3 - 2}{1} \right\rfloor + 1 = 1 + 1 = 2 \\ W_{\text{out}} &= \left\lfloor \frac{W_{\text{in}} - K_w}{S} \right\rfloor + 1 = \left\lfloor \frac{3 - 2}{1} \right\rfloor + 1 = 1 + 1 = 2 \\ C_{\text{out}} &= 3 \end{aligned}$$

$$\text{Dim}(Y) = [2, 2, 3]$$

$$\text{Elements} = 2 \times 2 \times 3 = 12$$

2.2 part (b)

2.2.1 Input:

$$X^{(1)} = \begin{bmatrix} -8 & 4 & 8 \\ -2 & 6 & -7 \\ 7 & -5 & 7 \end{bmatrix}, \quad X^{(2)} = \begin{bmatrix} -7 & 6 & 7 \\ -9 & -3 & -8 \\ 2 & -3 & 0 \end{bmatrix}$$

2.2.2 Kernels:

$$\begin{aligned} W_{1,1} &= \begin{bmatrix} -8 & 1 \\ -2 & 3 \end{bmatrix}, \quad W_{2,1} = \begin{bmatrix} -6 & -6 \\ 2 & 4 \end{bmatrix} \\ W_{1,2} &= \begin{bmatrix} -6 & 8 \\ -7 & 5 \end{bmatrix}, \quad W_{2,2} = \begin{bmatrix} -4 & 7 \\ -1 & 0 \end{bmatrix} \end{aligned}$$

$$W_{1,3} = \begin{bmatrix} -2 & -3 \\ -2 & 0 \end{bmatrix}, \quad W_{2,3} = \begin{bmatrix} -3 & -9 \\ 1 & 0 \end{bmatrix}$$

2.2.3 X' construction :

Patches at $(0,0), (0,1), (1,0), (1,1)$:

$$\text{Col}_1 : \begin{bmatrix} -8 \\ 4 \\ -2 \\ 6 \\ -7 \\ 6 \\ -9 \\ -3 \end{bmatrix}, \quad \text{Col}_2 : \begin{bmatrix} 4 \\ 8 \\ 6 \\ -7 \\ 6 \\ 7 \\ -3 \\ -8 \end{bmatrix}$$

$$\text{Col}_3 : \begin{bmatrix} -2 \\ 6 \\ 7 \\ -5 \\ -9 \\ -3 \\ 2 \\ -3 \end{bmatrix}, \quad \text{Col}_4 : \begin{bmatrix} 6 \\ -7 \\ -5 \\ 7 \\ -3 \\ -8 \\ -3 \\ 0 \end{bmatrix}$$

$$X' = \begin{bmatrix} -8 & 4 & -2 & 6 \\ 4 & 8 & 6 & -7 \\ -2 & 6 & 7 & -5 \\ 6 & -7 & -5 & 7 \\ -7 & 6 & -9 & -3 \\ 6 & 7 & -3 & -8 \\ -9 & -3 & 2 & -3 \\ -3 & -8 & -3 & 0 \end{bmatrix}$$

$$\dim(X') = 8 \times 4$$

2.2.4 W' construction:

$$\text{Row}_1 = [-8, 1, -2, 3, -6, -6, 2, 4]$$

$$\text{Row}_2 = [-6, 8, -7, 5, -4, 7, -1, 0]$$

$$\text{Row}_3 = [-2, -3, -2, 0, -3, -9, 1, 0]$$

$$W' = \begin{bmatrix} -8 & 1 & -2 & 3 & -6 & -6 & 2 & 4 \\ -6 & 8 & -7 & 5 & -4 & 7 & -1 & 0 \\ -2 & -3 & -2 & 0 & -3 & -9 & 1 & 0 \end{bmatrix}$$

$$\dim(W') = 3 \times 8$$

2.2.5 Matrix multiplication:

$$Y' = W' \times X'$$

$$Y' = \begin{bmatrix} 66 & -173 & 57 & 36 \\ 203 & -9 & -1 & -63 \\ -34 & -128 & 28 & 97 \end{bmatrix}$$

$$\dim(Y') = 3 \times 4$$

2.2.6 Reshape to Y :

$$Y^{(1)} = \begin{bmatrix} 66 & -173 \\ 57 & 36 \end{bmatrix}$$

$$Y^{(2)} = \begin{bmatrix} 203 & -9 \\ -1 & -63 \end{bmatrix}$$

$$Y^{(3)} = \begin{bmatrix} -34 & -128 \\ 28 & 97 \end{bmatrix}$$

$$Y \in \mathbb{R}^{2 \times 2 \times 3}$$

$$Y' = W' \times X'$$

$$\dim(Y') = 3 \times 4$$