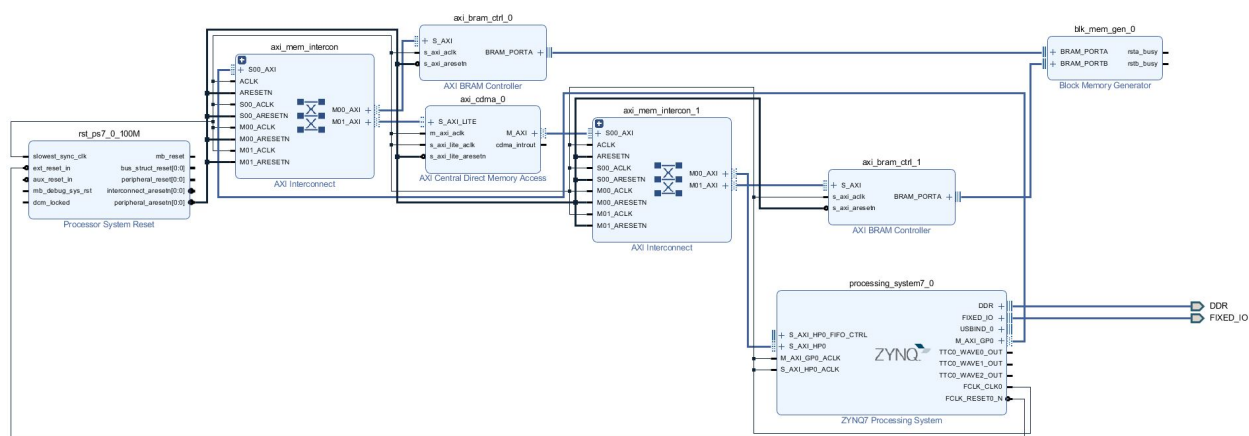


- Implement convolution lowering in C++.
- Integrate convolution lowering into the pretrained model (CNN).
  - On MNIST dataset as Figure ??.



## 1 Implementation

MLP 모델의 경우 28 x 28 크기의 숫자 이미지이며 이를 벡터화 하여 여러 Layer를 거쳐 마지막에 output layer에 10개의 숫자 중 나타날 확률을 학습하게 된다. 행렬과 벡터 곱셈이 매우 크므로 공간이나 시간적 복잡도를 최적화하기 위해서는 Tiling method를 사용하였으며 이 방법을 CNN 모델의 행렬 곱셈에서도 사용하게 된다. 행렬과 벡터 또는 행렬과 행렬을 일정한 크기로 나누어 가속기를 통해 쓰레드를 나누어 빠른 연산 속도를 가능하게 한다.

### 1. Matrix-Vector Multiplication

## 2. Matrix-Matrix Multiplication

### 3. Convolution Lowering

Image와 Convolution Filter 데이터를 재정렬하여 Matrix Multiplication로 바꾸는 행렬화 작업 [?, ?]

**Practice #11. DMA (Direct Memory Access)**  
**Jiwon Lee, Sangjun Son**

---

## 1.1 Matrix-Vector Multiplication

행렬 벡터 연산을 수행할 때는 작은 크기의 행렬 벡터 연산 (Tiling)으로 축소 대응시켜 계산을 하게 되는데 이 때 자주 사용되는 함수가 block operation이다. 함수를 사용하기 위해서 중간 중간에 data\_라는 데이터 중간 저장소를 통해 곱셈에 사용될 행렬과 벡터를 fetching 하고 연산된 output vector를 덮어쓰는 과정을 반복하게 된다. 코드에 대한 자세한 설명은 Lab 2를 참고하면 된다 [?].

### FPGA::largeMV

```
1 void FPGA::largeMV(const float* large_mat, const float* input, float* output, int num_input, int num_output)
2 {
3     float* vec = this->vector();
4     float* mat = this->matrix();
5
6     // 0) Initialize output vector
7     for(int i = 0; i < num_output; ++i)
8         output[i] = 0;
9
10    for(int i = 0; i < num_output; i += m_size_)
11    {
12        for(int j = 0; j < num_input; j += v_size_)
13        {
14            // 0) Initialize input vector
15            int block_row = min(m_size_, num_output-i);
16            int block_col = min(v_size_, num_input-j);
17
18            // 1) Assign a vector
19            for (int col = 0; col < block_col; col++)
20                data_[col] = input[j + col];
21            for (int col = block_col; col < v_size_; col++)
22                data_[col] = 0;
23
24            // 2) Assign a matrix
25            for (int row = 0; row < block_row; row++)
26                for (int col = 0; col < block_col; col++)
27                    data_[(row+1)*v_size_ + col] = large_mat[(i+row)*num_input + (j+col)];
28
29            // 3) Call a function 'blockMV()' to execute MV multiplication
30            const float* ret = this->blockMV();
31
32            // 4) Accumulate intermediate results
33            for(int row = 0; row < block_row; ++row)
34                output[i + row] += ret[row];
35        }
36    }
37 }
38 }
```

## 1.2 Matrix-Matrix Multiplication

Section ??과 마찬가지로 Tiling Method으로 축소 대응시켜 계산을 하면 된다. 함수를 사용하기 위해서 중간 중간에 data\_M라는 데이터 중간 저장소를 통해 곱셈에 사용될 2개의 행렬을 fetching 하고 연산된 output vector를 덮어쓰는 과정을 반복하게 된다.

Figure ??에서 볼 수 있듯이 기본적으로 v\_size 간격으로 작은 Block operation을 수행하지만 행렬의 가로, 세로 크기가 항상 v\_size의 배수가 아니므로 경계부분에서의 예외처리를 위해 Block 사이즈를 나타내는 변수 block\_row, block\_col\_1, block\_col\_2를 도입한다.

### FPGA::largeMM

```
1 void FPGA::largeMM(const float* weight_mat, const float* input_mat, float* output,
2                     int num_input, int num_output, int num_matrix2)
3 {
```

**Practice #11. DMA (Direct Memory Access)**  
**Jiwon Lee, Sangjun Son**

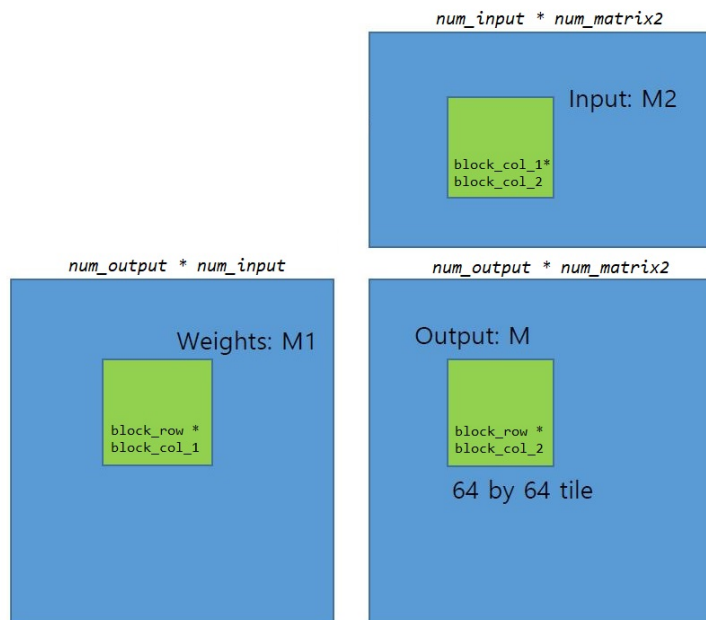


Figure 2: Block operation에 사용되는 행렬의 곱셈 연산 부분, 파란색 행렬을 곱셈하기 위해 Tiling Method으로 쪼개었을 때 연두색 행렬끼리의 곱셈으로 Output Matrix를 채워나갈 수 있다. 위 예시는 쪼개는 단위인 v\_size가 64이다.

```

4  float* m1 = this->matrix_M1();
5  float* m2 = this->matrix_M2();
6
7  // 0) Initialize output vector
8  for(int i = 0; i < num_output*num_matrix2; ++i)
9      output[i] = 0;
10
11 for(int i = 0; i < num_output; i += v_size_)
12 {
13     for(int j = 0; j < num_input; j += v_size_)
14     {
15         for(int k = 0; k < num_matrix2; k += v_size_)
16         {
17             // 0) Initialize input vector
18             int block_row = min(v_size_, num_output-i);
19             int block_col_1 = min(v_size_, num_input-j);
20             int block_col_2 = min(v_size_, num_matrix2-k);
21
22             // 1) Assign a m1
23             for (int row = 0; row < block_row; row++) {
24                 for (int col = 0; col < block_col_1; col++)
25                     data_M[row*v_size_ + col] = weight_mat[(i+row)*num_input + (j+col)];
26                 for (int col = block_col_1; col < v_size_; col++)
27                     data_M[row*v_size_ + col] = 0;
28             }
29             for (int l = block_row*v_size_; l < m1_size_; l++)
30                 data_M[l] = 0;
31
32             // 2) Assign a m2
33             for (int row = 0; row < block_col_1; row++) {
34                 for (int col = 0; col < block_col_2; col++)
35                     data_M[m1_size_ + row*v_size_ + col] = input_mat[(j+row)*num_matrix2 + (k+col)];

```

**Practice #11. DMA (Direct Memory Access)**  
**Jiwon Lee, Sangjun Son**

---

```
36     for (int col = block_col_2; col < v_size_; col++)
37     {
38         data_M[m1_size_ + row*v_size_ + col] = 0;
39     }
40     for (int l = block_col_1*v_size_; l < m2_size_; l++)
41     {
42         data_M[m1_size_ + l] = 0;
43     }
44     // 3) Call a function `blockMM()` to execute Matrix matrix multiplication
45     const float* ret = this->blockMM();
46     // 4) Accumulate intermediate results
47     for(int n = 0; n<block_row; ++n)
48     {
49         for(int m = 0; m<block_col_2; ++m)
50         {
51             output[(i + n) + (k + m)*num_output] += ret[n*v_size_ + m];
52         }
53     }
54 }
55 }
56 }
```

- Block OP를 수행할 부분인 Weight Matrix를 data\_M에 먼저 넣는다 (24 라인).
- 이 행렬의 크기는 block\_row \* block\_col\_1 이므로 첫 번째 행렬 부분에 해당하는 m1\_size\_ 중 사용하지 않는 부분은 모두 0으로 초기화한다 (26-29 라인).
  - 그렇지 않다면 전 step에서 사용되었던 벡터 값이 잘못된 연산 결과를 초래할 수 있기 때문이다 [?].
- 다음으로 Input Matrix를 data\_M에 넣는다 (34 라인).
- 이 행렬의 크기는 block\_col\_2 \* block\_col\_1 이므로 두 번째 행렬 부분에 해당하는 m2\_size\_ 중 사용하지 않는 부분은 모두 0으로 초기화한다 (36-39 라인).

Practice #11. DMA (Direct Memory Access)  
Jiwon Lee, Sangjun Son

### 1.3 Convolution Lowering

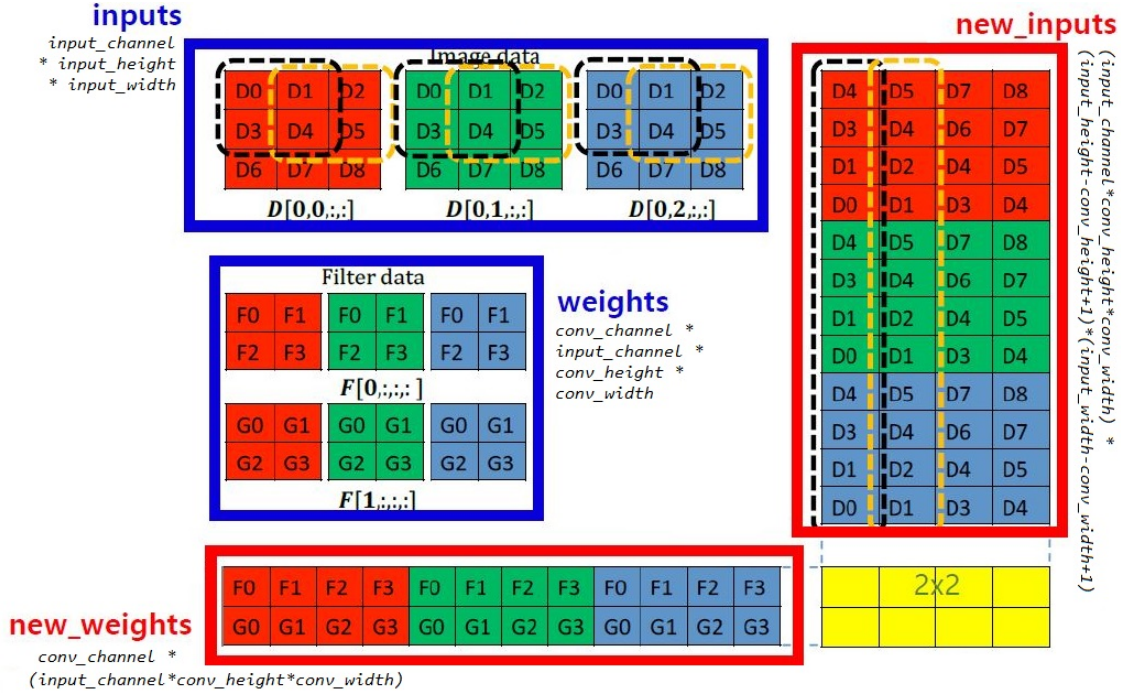


Figure 3: Convolution Filter와 Image data에 해당하는 3차원 텐서를 행렬화 시키기 위한 과정을 도식화 한 것 [?, ?]. 여기서 Convolution Lowering은 *inputs*와 *weight*를 각각 *new\_inputs*와 *new\_weight*로 매핑하는 과정이다.

Figure ??를 참조하면 *new\_weight*의 행들은 *weight*의 channel로 나뉘지고 각 행은 *input\_channel* 순서로 *weight* 값이 순서대로 나온다. 아래의 코드의 30-31 라인처럼 *conv\_channel*, *input\_channel*, *conv\_height*, *conv\_width* 순서대로 for문을 돌면서 *new\_weight* 원소 값을 채워준다.

*new\_weight*와 마찬가지로 *new\_inputs*를 구성할 수 있다. 한 Filter가 Input에 방문하는 횟수는 행으로는  $\text{input\_height} - \text{conv\_height} + 1$ 이고 열의 방향으로는  $\text{input\_width} - \text{conv\_width} + 1$ 이 될 것이다. 순서대로 for문을 돌면서 *new\_inputs*의 행열 순서로 채워간다 (38-39 라인).

#### FPGA: :convLowering

```
1 void FPGA::convLowering(const std::vector<std::vector<std::vector<std::vector<float>>>>& cnn_weights,
2   std::vector<std::vector<float>>& new_weights,
3   const std::vector<std::vector<std::vector<float>>>& inputs,
4   std::vector<std::vector<float>>& new_inputs) {
5   /*
6    * Arguments:
7    *
8    * conv_weights: [conv_channel, input_channel, conv_height, conv_width]
9    * new_weights: [conv_channel, input_channel*conv_height*conv_width]
10   * inputs: [input_channel, input_height, input_width]
11   * new_inputs: [input_channel*conv_height*conv_width, (input_height-conv_height+1)*(input_width-conv_width+1)]
12   *
13   */
```

**Practice #11. DMA (Direct Memory Access)**  
**Jiwon Lee, Sangjun Son**

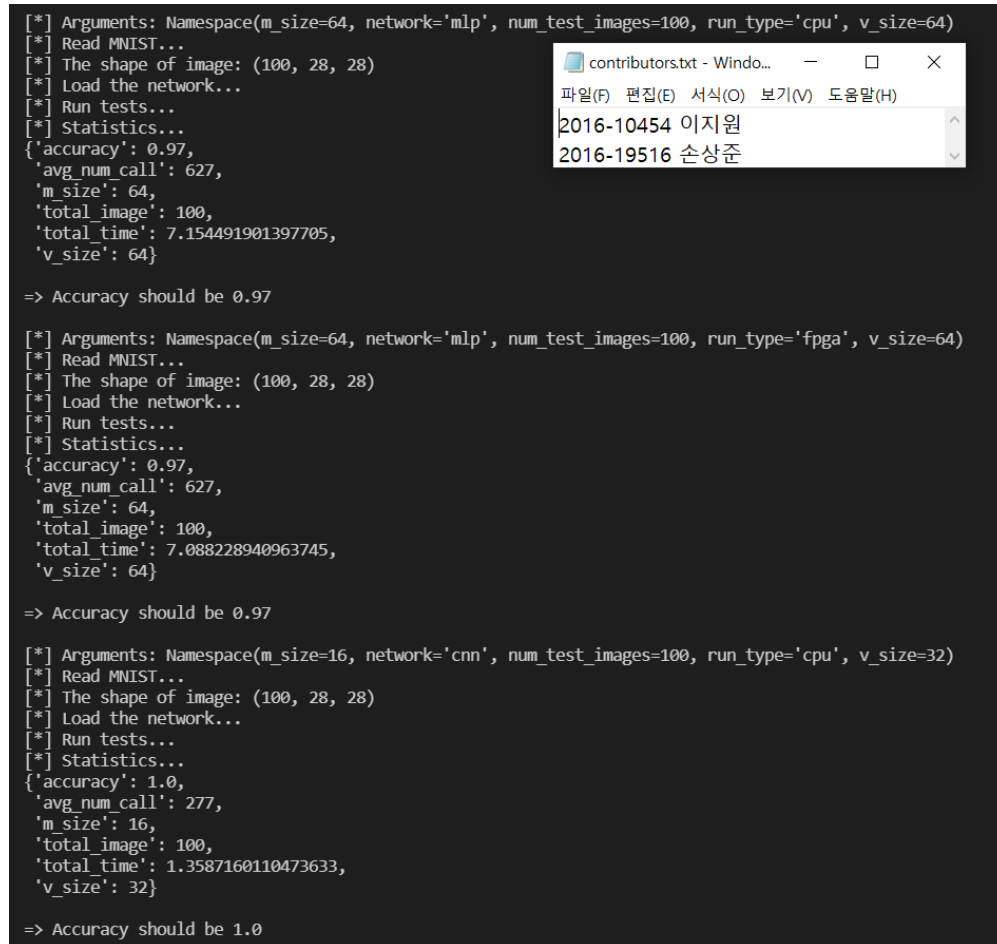
---

```
14
15 int conv_channel = cnn_weights.size();
16 int input_channel = cnn_weights[0].size();
17 int conv_height = cnn_weights[0][0].size();
18 int conv_width = cnn_weights[0][0][0].size();
19 //int input_channel = cnn_weights.size();
20 int input_height = inputs[0].size();
21 int input_width = inputs[0][0].size();
22
23 // For example,
24 // new_weights[0][0] = cnn_weights[0][0][0][0];
25 // new_inputs[0][0] = inputs[0][0][0];
26 for (int i = 0; i < conv_channel; i++)
27     for (int j = 0; j < input_channel; j++)
28         for (int k = 0; k < conv_height; k++)
29             for (int l = 0; l < conv_width; l++)
30                 new_weights[i][j*conv_height*conv_width + k*conv_width + l]
31                     = cnn_weights[i][j][k][l];
32
33 for (int i = 0; i < input_channel; i++)
34     for (int j = 0; j < conv_height; j++)
35         for (int k = 0; k < conv_width; k++)
36             for (int l = 0; l < input_height-conv_height+1; l++)
37                 for (int m = 0; m < input_width-conv_width+1; m++)
38                     new_inputs[i*conv_height*conv_width + j*conv_width + k][l*(input_width-conv_width+1) + m]
39                         = inputs[i][j+1][k+m];
40
41 }
```

Practice #11. DMA (Direct Memory Access)  
Jiwon Lee, Sangjun Son

## 2 Result

구현한 코드를 CPU 상에서 Pre-trained MLP와 CNN network를 사용하여 정확도를 측정해보았다. MLP는 0.97의 정확도를 보였으며 CNN은 MLP보다 높은 0.98에서 1.0의 정확도를 보였다. 아래의 Figure ??는 Lab 9에서 주어진 `benchmark.sh`를 수행하였을 때의 결과이다.



```
[*] Arguments: Namespace(m_size=64, network='mlp', num_test_images=100, run_type='cpu', v_size=64)
[*] Read MNIST...
[*] The shape of image: (100, 28, 28)
[*] Load the network...
[*] Run tests...
[*] Statistics...
{'accuracy': 0.97,
 'avg_num_call': 627,
 'm_size': 64,
 'total_image': 100,
 'total_time': 7.154491901397705,
 'v_size': 64}

=> Accuracy should be 0.97

[*] Arguments: Namespace(m_size=64, network='mlp', num_test_images=100, run_type='fpga', v_size=64)
[*] Read MNIST...
[*] The shape of image: (100, 28, 28)
[*] Load the network...
[*] Run tests...
[*] Statistics...
{'accuracy': 0.97,
 'avg_num_call': 627,
 'm_size': 64,
 'total_image': 100,
 'total_time': 7.088228940963745,
 'v_size': 64}

=> Accuracy should be 0.97

[*] Arguments: Namespace(m_size=16, network='cnn', num_test_images=100, run_type='cpu', v_size=32)
[*] Read MNIST...
[*] The shape of image: (100, 28, 28)
[*] Load the network...
[*] Run tests...
[*] Statistics...
{'accuracy': 1.0,
 'avg_num_call': 277,
 'm_size': 16,
 'total_image': 100,
 'total_time': 1.3587160110473633,
 'v_size': 32}

=> Accuracy should be 1.0
```

Figure 4: Zedboard CPU/FPGA상에서 Pre-trained MLP와 CNN를 사용해 MNIST 데이터셋에 대하여 test 한 결과. `benchmark.sh`에 포함되지 않은 FPGA CNN은 포함하지 않았다.

수행 성능에 영향을 미치는 환경변수는 `m_size`, `v_size`, `num_test_images`가 있으며 네트워크의 종류 또한 변화시키면서 실험을 진행하였다. 기본 값으로는 `m_size = v_size = 16`과 `num_test_images = 100`으로 설정하였다. 관측 변수로는 정확도 `accuracy`, 걸리는 시간 `total_time`, `avg_num_call`을 측정하였다.

**Practice #11. DMA (Direct Memory Access)**  
**Jiwon Lee, Sangjun Son**

Network	Control Variable	<i>total_time</i>	<i>avg_num_call</i>	<i>accuracy</i>
cnn	num_test_images(1)	0.007s	553	1
cnn	num_test_images(10)	0.067s	553	1
cnn	num_test_images(100)	0.661s	553	1
cnn	num_test_images(1000)	6.635s	553	0.98
cnn	num_test_images(10000)	66.534s	553	0.98
cnn	v_size(1)	1.589s	44646	1
cnn	v_size(2)	0.799s	9141	1
cnn	v_size(4)	0.619s	3050	1
cnn	v_size(8)	0.56s	1188	1
cnn	v_size(16)	0.662s	553	1
cnn	v_size(32)	1.34s	277	1
cnn	v_size(64)	4.042s	140	1
cnn	v_size(128)	37.86s	71	1
mlp	num_test_images(1)	0.068s	9375	1
mlp	num_test_images(10)	0.675s	9375	0.9
mlp	num_test_images(100)	6.738s	9375	0.97
mlp	num_test_images(1000)	67.441s	9375	0.92
mlp	num_test_images(10000)	675.352s	9375	0.9159
mlp	v_size(1)	13.591s	150000	0.97
mlp	v_size(2)	9.524s	75000	0.97
mlp	v_size(4)	7.34s	37500	0.97
mlp	v_size(8)	6.323s	18750	0.97
mlp	v_size(16)	6.743s	9375	0.97
mlp	v_size(32)	6.537s	4763	0.97
mlp	v_size(64)	6.398s	2419	0.97
mlp	v_size(128)	6.562s	1285	0.97
mlp	m_size(1)	8.767s	149550	0.97
mlp	m_size(2)	7.663s	74775	0.97
mlp	m_size(4)	7.104s	37425	0.97
mlp	m_size(8)	6.83s	18750	0.97
mlp	m_size(16)	6.745s	9375	0.97
mlp	m_size(32)	6.789s	4787	0.97
mlp	m_size(64)	6.957s	2431	0.97
mlp	m_size(128)	7.502s	1315	0.97

Table 1:  $m\_size = v\_size = 16$ 과  $num\_test\_images = 100$ 로 설정하고 매개 변수를 하나씩 바꿔가면서 측정한 연산 성능 *accuracy*, *total\_time*, *avg\_num\_call* 비교



**Practice #11. DMA (Direct Memory Access)**  
**Jiwon Lee, Sangjun Son**

---

### 3 Conclusion

이번 실습에서는 Convolution Lowering을 사용해 CNN 연산을 Matrix Multiplication 연산으로 바꿔보았다. 또한 Lab 2에서 구현한 Matrix Vector Multiplication도 불러와 함께 연산에 사용하였다. MNIST 데이터에 대해서 각각 MLP, CNN으로 inference 한 후 비교해 본 결과 CNN 결과가 조금 더 높은 정확도를 보였다.

Convolution Filter의 Weights과 Inputs을 2차원 Matrix를 바꾸는 과정에서 같은 값이 여러 번 사용되어 행렬을 구성하는 것을 확인할 수 있었다. 다시 말하면 더 많은 Memory Allocation을 필요로 하였고 이 부분에서 더 최적화 가능할 것이라고 판단하였다. 또한 실제로 구현되는 CNN을 보면 매개변수로 Stride나 Padding을 넘겨줄 수 있다. 이번 Lab 9에는 이 부분이 빠져 있지만 추가된다면 더욱 더 일반적인 네트워크를 위한 Convolution Lowering을 구현할 수 있을 것이다.