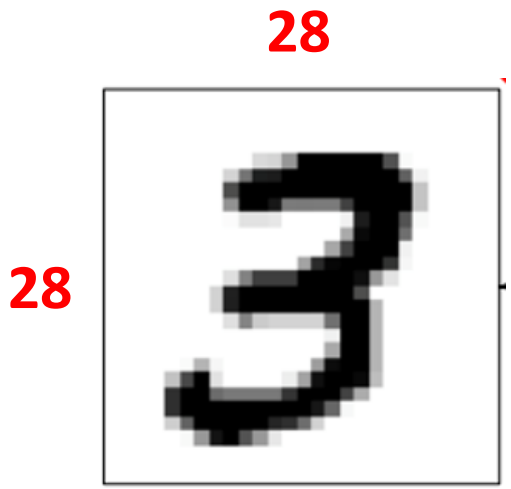
Practice 2

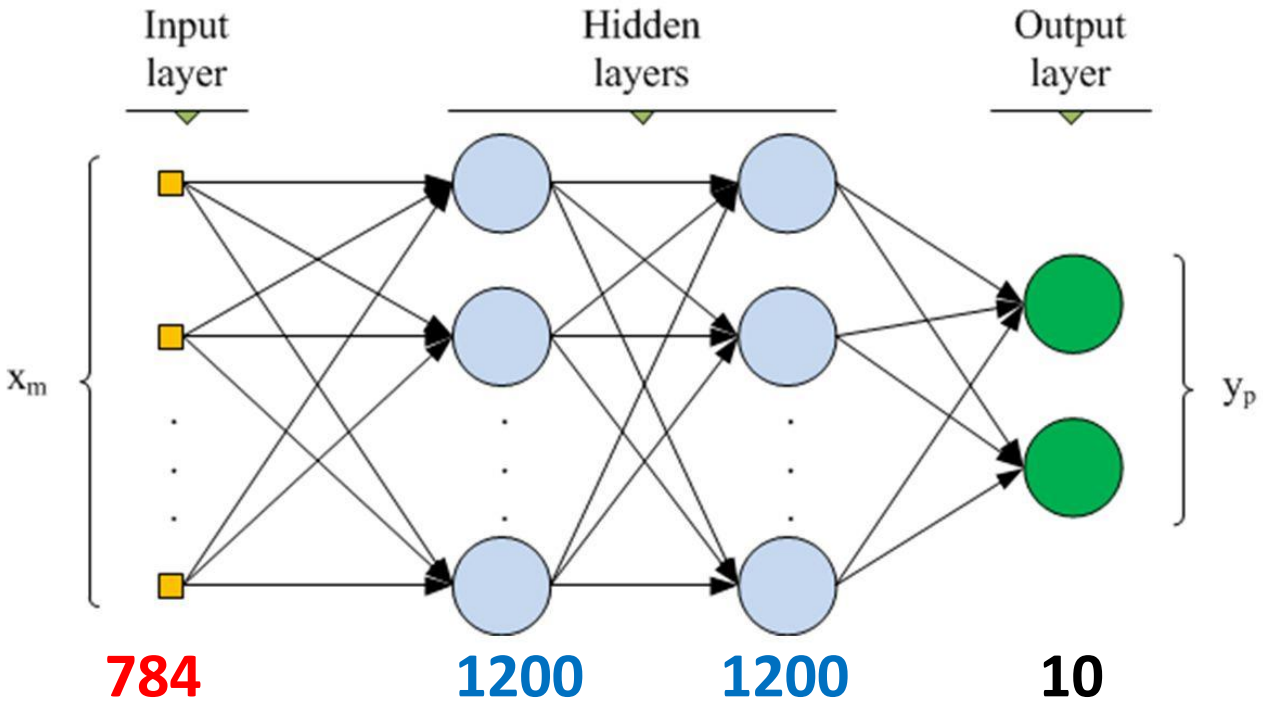
- SW app on MNIST

Computing Memory Architecture Lab.

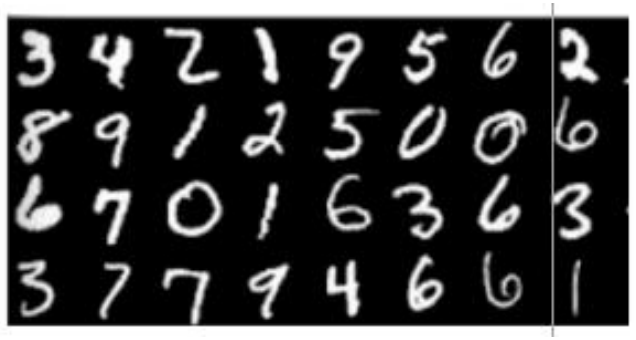
Lab 2: Overview

- Goal
  - Implement matrix-vector(MV) multiplication in C++
  - Integrate MV multiplication into the pretrained model(MLP)
    - On MNIST





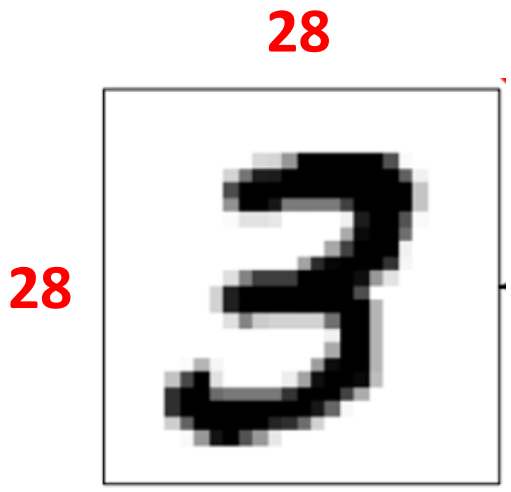
MNIST, CIFAR 10

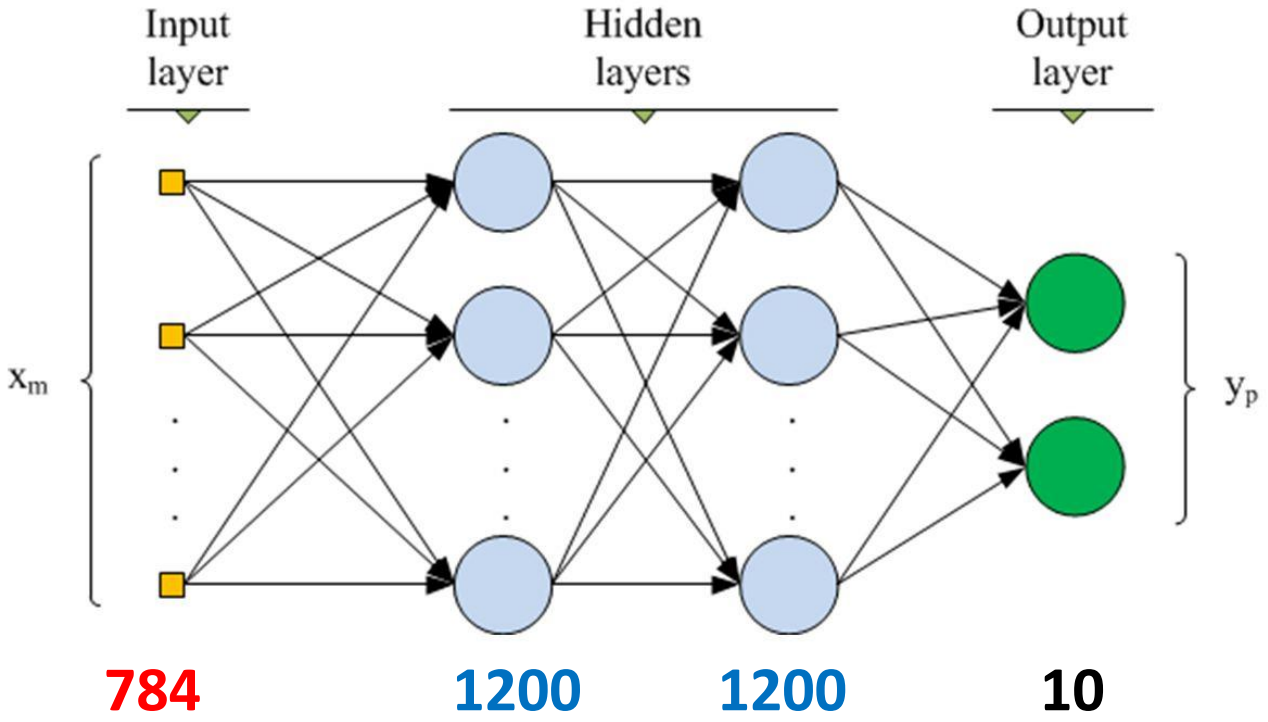


Dataset	MNIST	CIFAR 10
Category	digits(0-9)	airplane, automobile, …, truck
Image size	28x28	32x32
Color	Gray scale	RGB
# images (training/test)	60000/10000	60000/10000

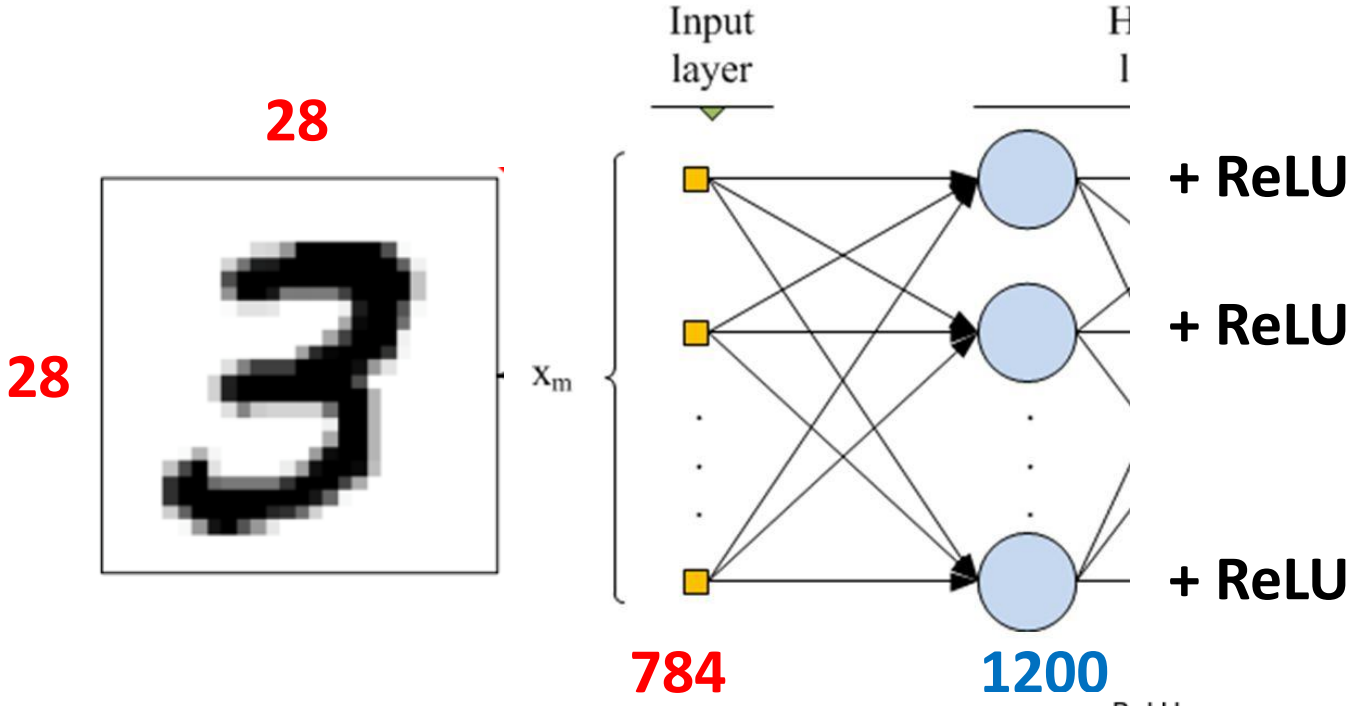
(pretrained) Multi-Layer Perceptron(MLP)

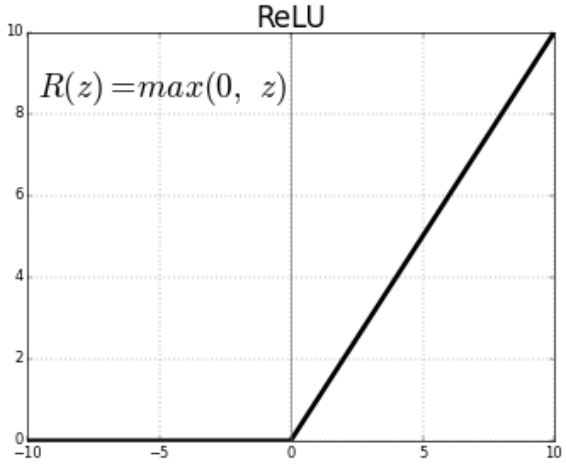
- Input: **28x28** pixels → **1200** values → **1200** values → **10** values
- 1st layer: **784** inputs → e.g., **1200** outputs ➔ Large M*V multiplication
  - M (784x1200) x V (784)





Implementation Detail: 1st layer





1) Reshape

28x28 → 784

2) (MV) Multiplication

Output[0] = Σ_j Input[j]*W[0,j]

Output[1] = Σ_j Input[j]*W[1,j]

…

Output[1199] = Σ_j Input[j]*W[1199,j]

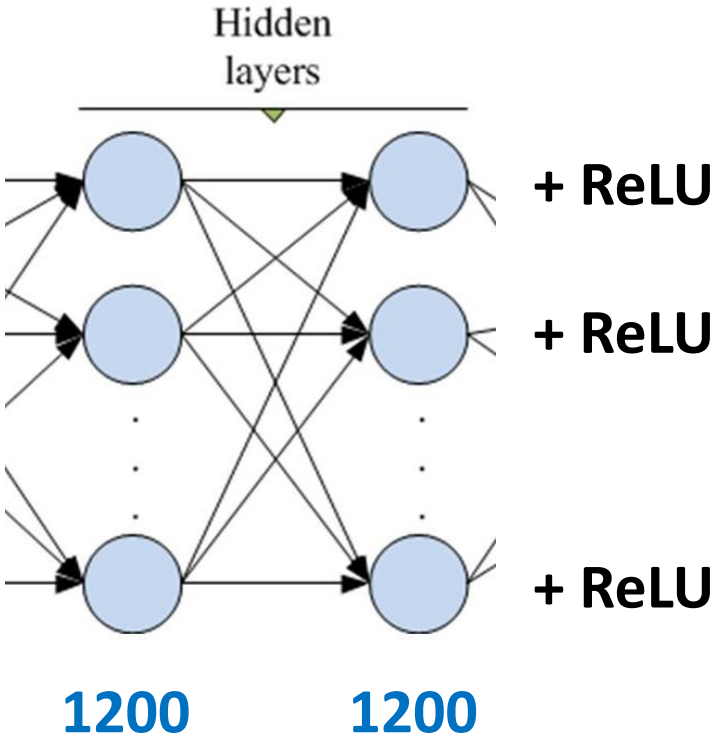
3) Activation(ReLU)

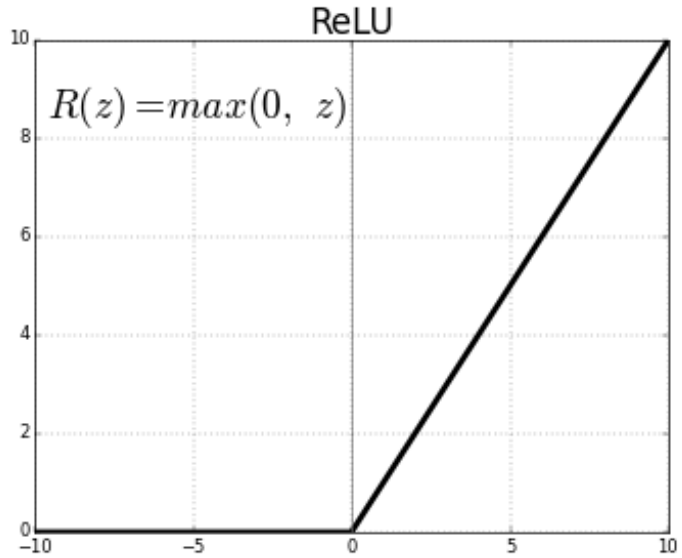
Output[0] = max(0, Output[0])

…

Output[1199] = max(0, Output[1199])

Implementation Detail: 2nd layer





1) (MV) Multiplication

Output[0] = Σ_j Input[j]*W[0,j]

Output[1] = Σ_j Input[j]*W[1,j]

…

Output[1199] = Σ_j Input[j]*W[1199,j]

2) Activation(ReLU)

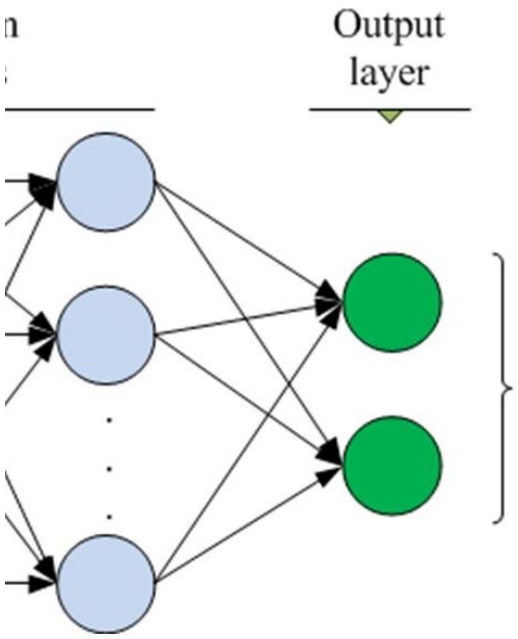
Output[0] = max(0, Output[0])

Output[1] = max(0, Output[1])

…

Output[1199] = max(0, Output[1199])

Implementation Detail: 3rd layer



1200 10

+ Softmax

$$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_i}}$$

1) (MV) Multiplication

Output[0] = Σ_j Input[j]*W[0,j]
Output[1] = Σ_j Input[j]*W[1,j]
…
Output[9] = Σ_j Input[j]*W[9,j]

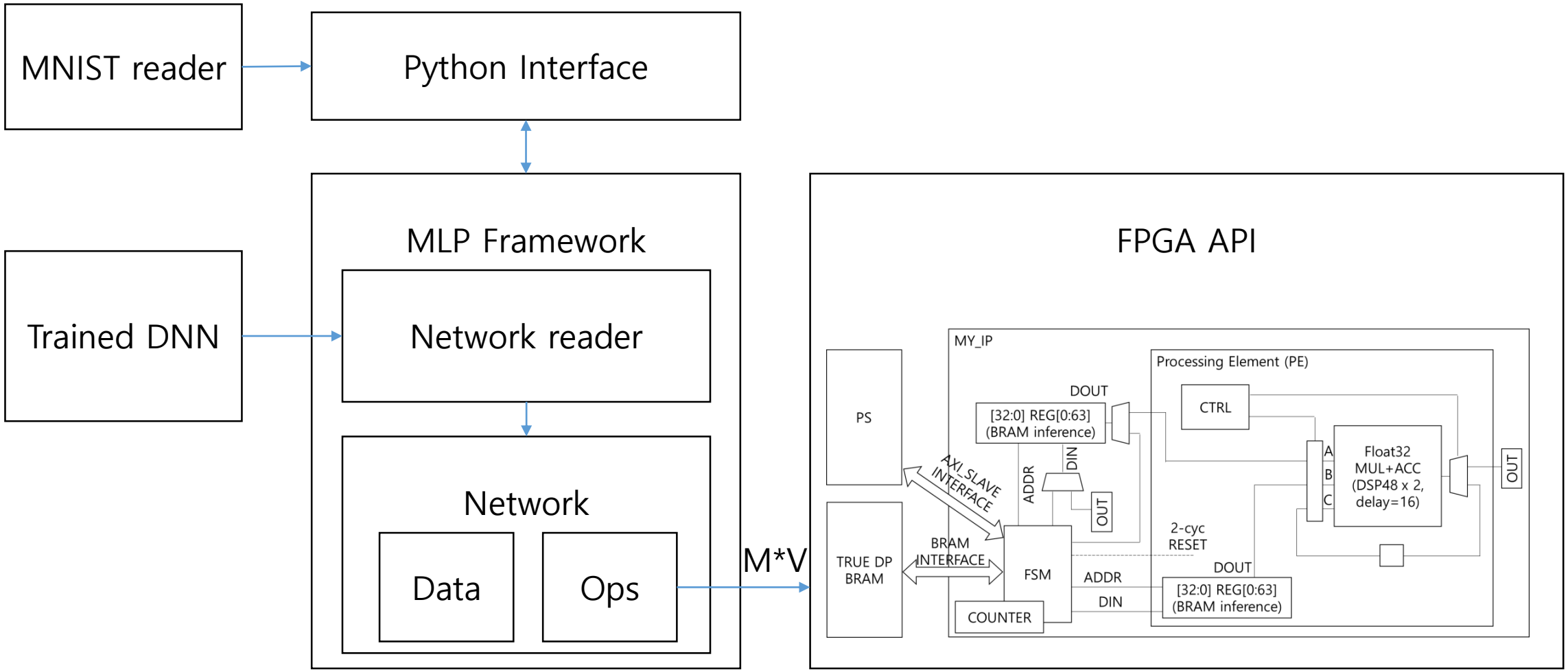
2) Softmax

Output[0] = softmax(Output[0])
…
Output[9] = softmax(Output[9])

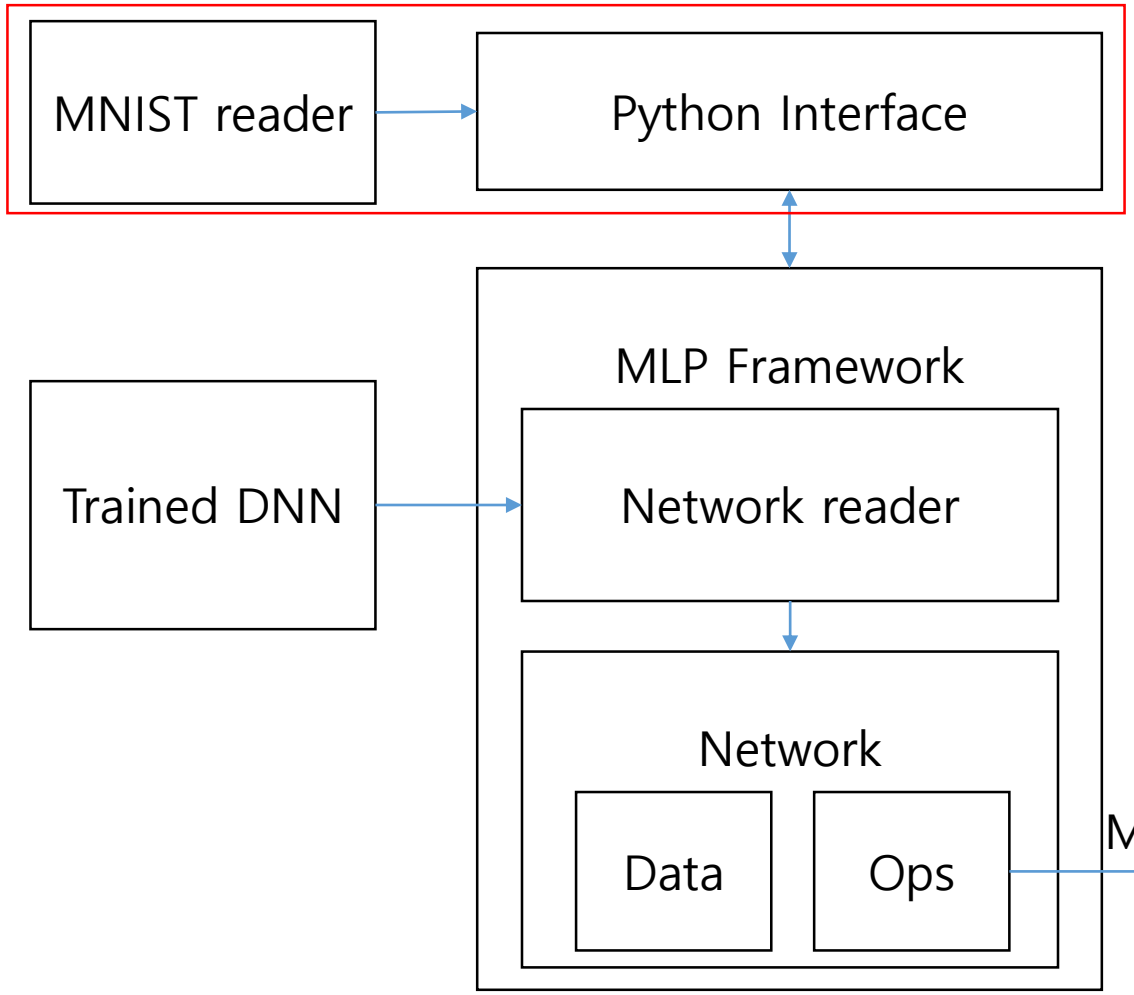
3) Argmax

Prediction = argmax_i (Output[i: 0~9])

MLP Framework on FPGA



MLP Framework on FPGA(1)

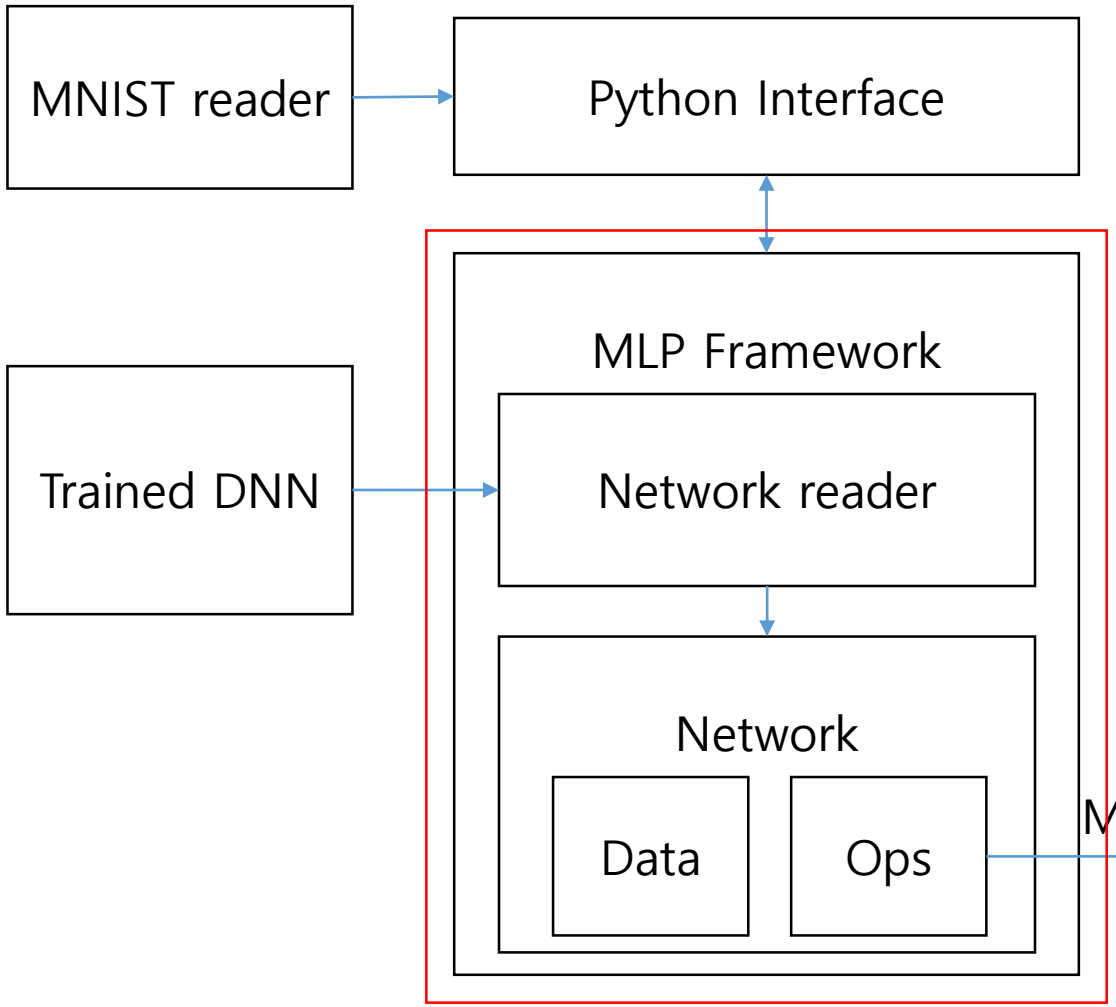


eval.py

py_lib.cpp

```
#-*- coding: utf-8 -*-
import numpy as np
import ctypes

import sys
sys.path.append('./mnist')
from load_mnist import load_mnist

images, labels = load_mnist("testing", path="./mnist")
images = images.astype(np.float32)/255.

class Network(object):
    def __init__(self, net_path):
        self.lib = ctypes.cdll.LoadLibrary("./build/libpylib.so")
        self.net = self.lib.getNet(ctypes.c_char_p(net_path.
        encode('utf-8')))
        self.out_buf = np.zeros((10), dtype = np.float32)

    def __del__(self):
        self.lib.delNet(self.net)

    def inference(self, input):
        self.lib.inference(self.net, input.ctypes.data, self.
        out_buf.ctypes.data)
        return self.out_buf

net = Network("./mnist_model/mnist_iter_10000.caffemodel")

for idx in xrange(5):
    print("image %d"%(idx))
    out = net.inference(images[idx,:,:].copy())
    print("real number: %d"%(labels[idx, :]))
    print("prediction result: %d"%(np.argmax(out)))
```

MLP Framework on FPGA(2)

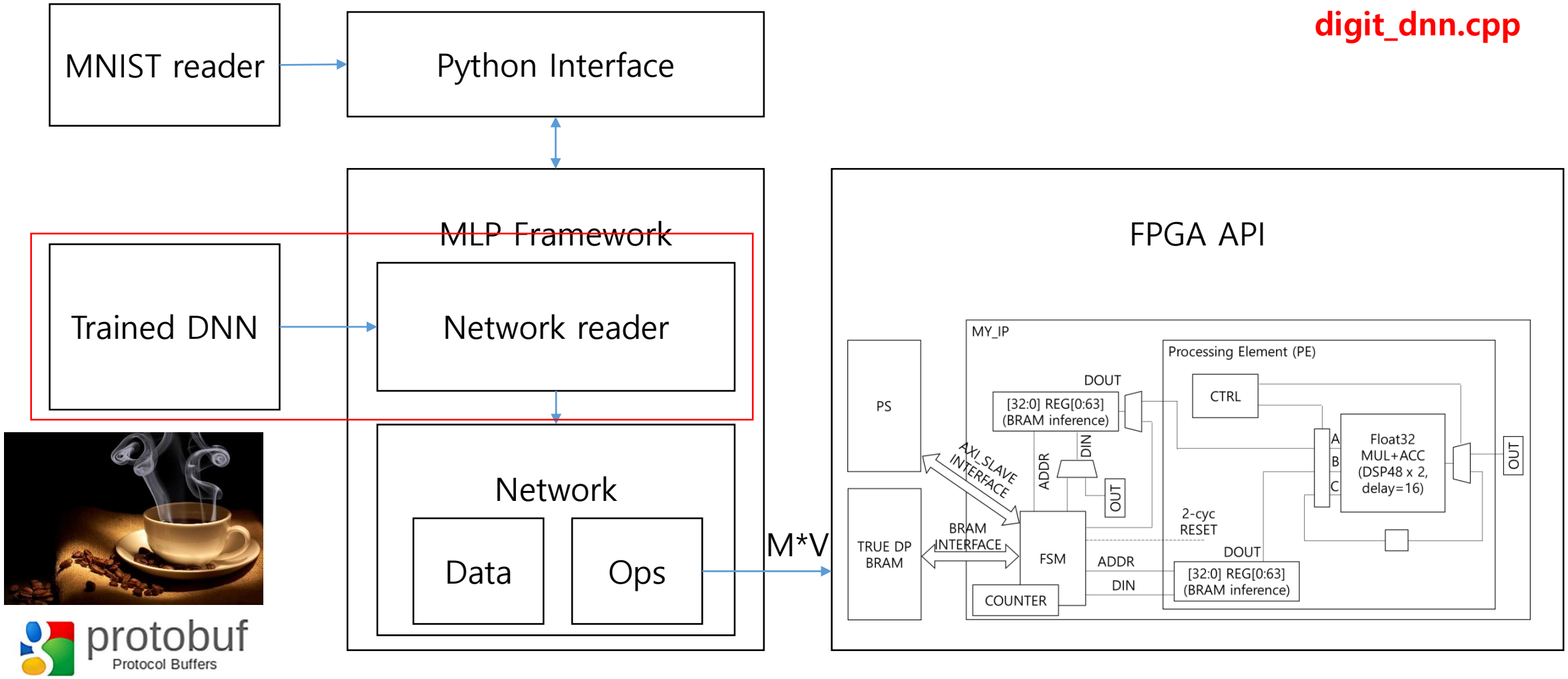


digit_dnn.cpp

```
class DigitDNN
{
private:
  caffe::NetParameter net_;
  caffe::NetParameter net_trained_;
  std::vector<std::unique_ptr<Op> > ops_;
  std::vector<std::pair<int, int> > op_pair_;

  std::vector<std::vector<float> > data_;
  std::unordered_map<std::string, int> data_name2idx_;

  FPGA* dev_;
  int input_idx_;
  int output_idx_;

  int regDataIdx(std::string name, int size);
  int getDataIdx(std::string name);

public:
  DigitDNN(std::string model, FPGA* dev);
  const float* run(const float* input, bool copy);
  FPGA* getDev();

};
```

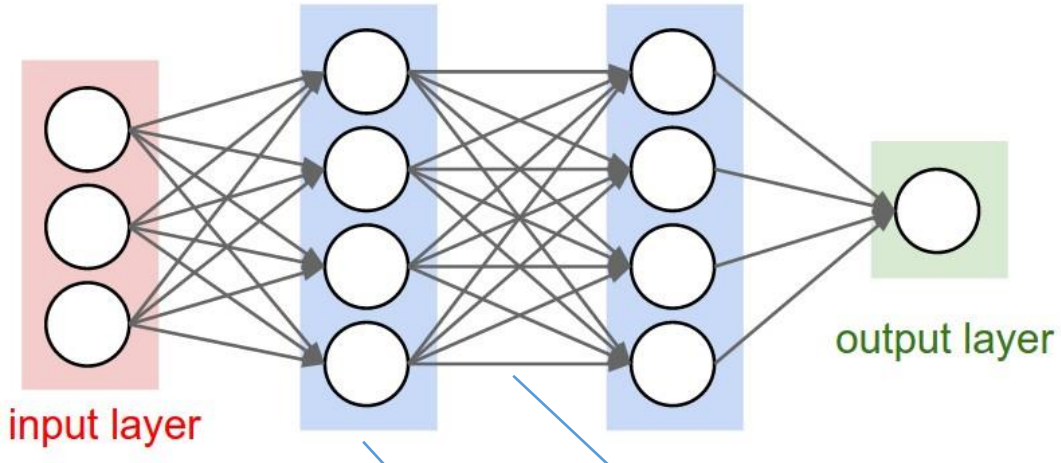
MLP Framework on FPGA(3)

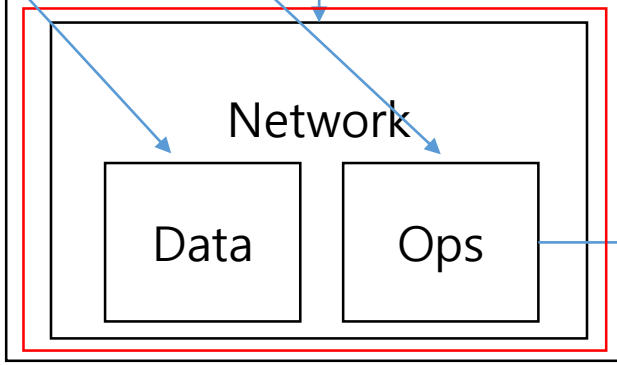
digit_dnn.cpp



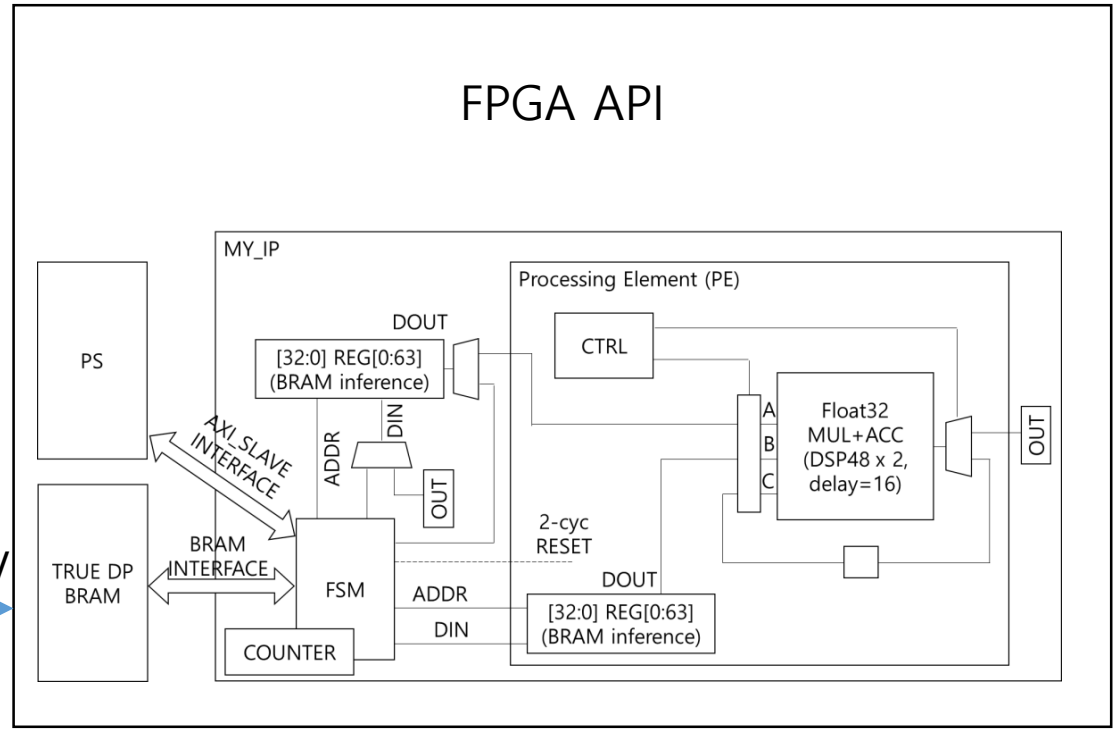
MLP Framework on FPGA(4)

digit_dnn.cpp

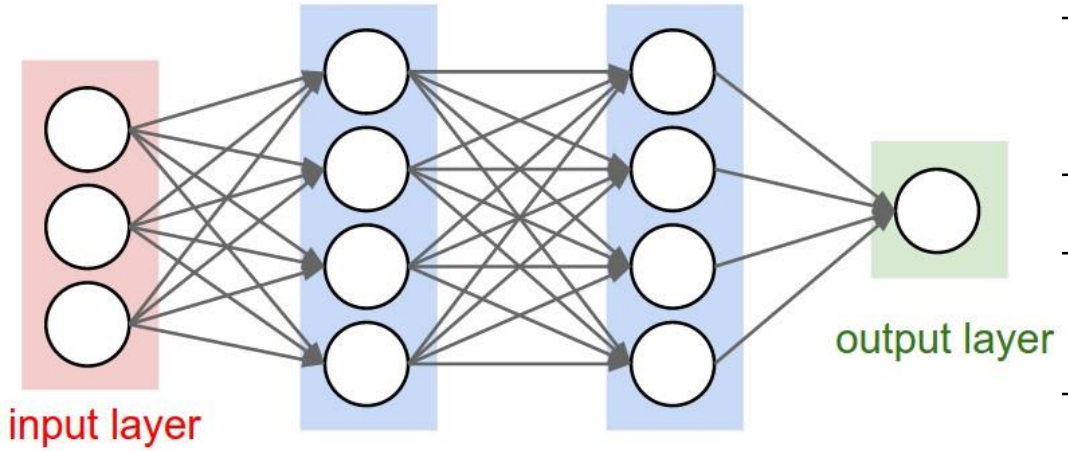


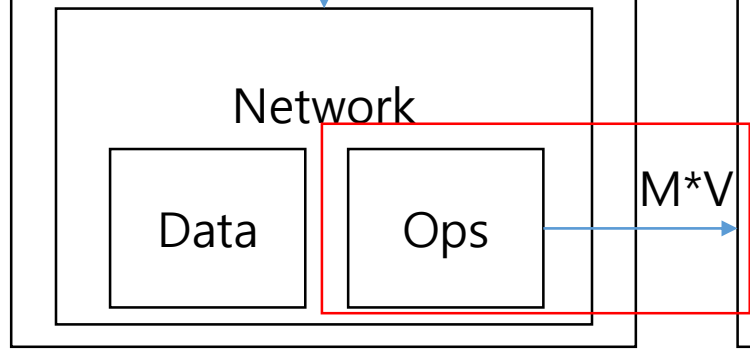


M*V



MLP Framework on FPGA(5)

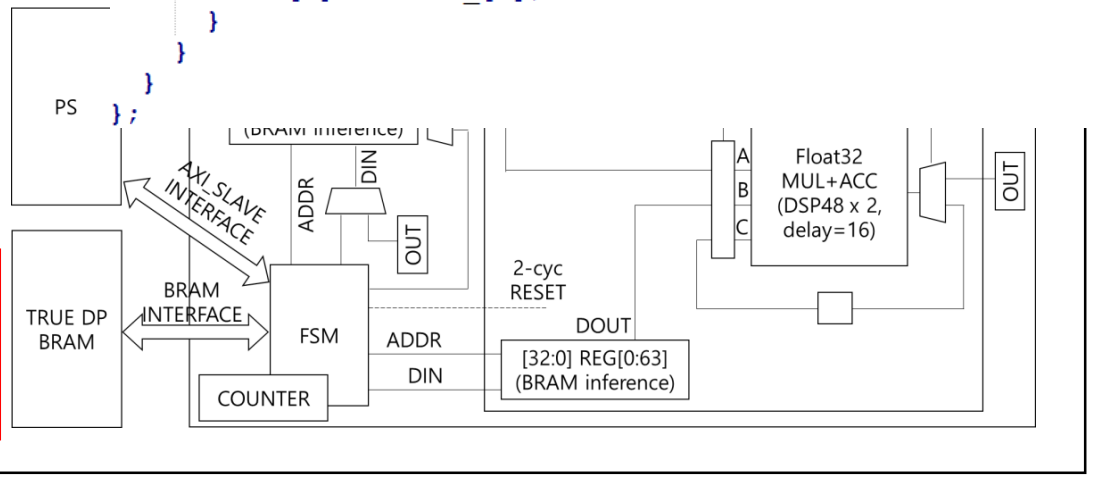




M*V

fpga_api.cpp

```
struct MatVecOp:Op
{
  FPGA* dev_;
  const float* weight_;
  const float* bias_;
  int input_size_;
  int output_size_;

  MatVecOp(FPGA* dev, const float* weight,
    const float* bias, int input_size, int output_size)
    :dev_(dev), weight_(weight), bias_(bias),
    input_size_(input_size), output_size_(output_size){}

  void run(const float* src, float* dst)
  {
    dev_->largeMV(weight_, src, dst, input_size_, output_size_);

    if(bias_ != nullptr)
    {
      for(int i = 0 ; i < output_size_; ++i)
      {
        dst[i] += bias_[i];
      }
    }
  }
};
```

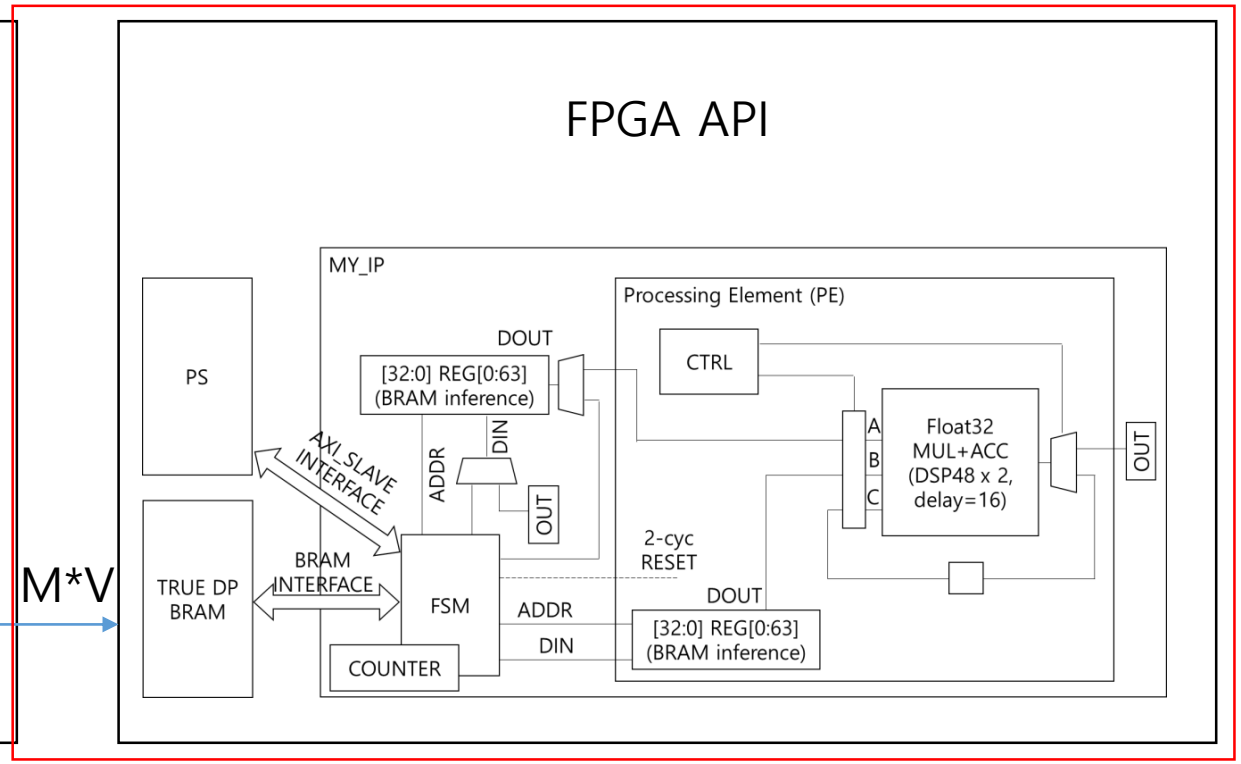


MLP Framework on FPGA(6)

fpag_api.cpp

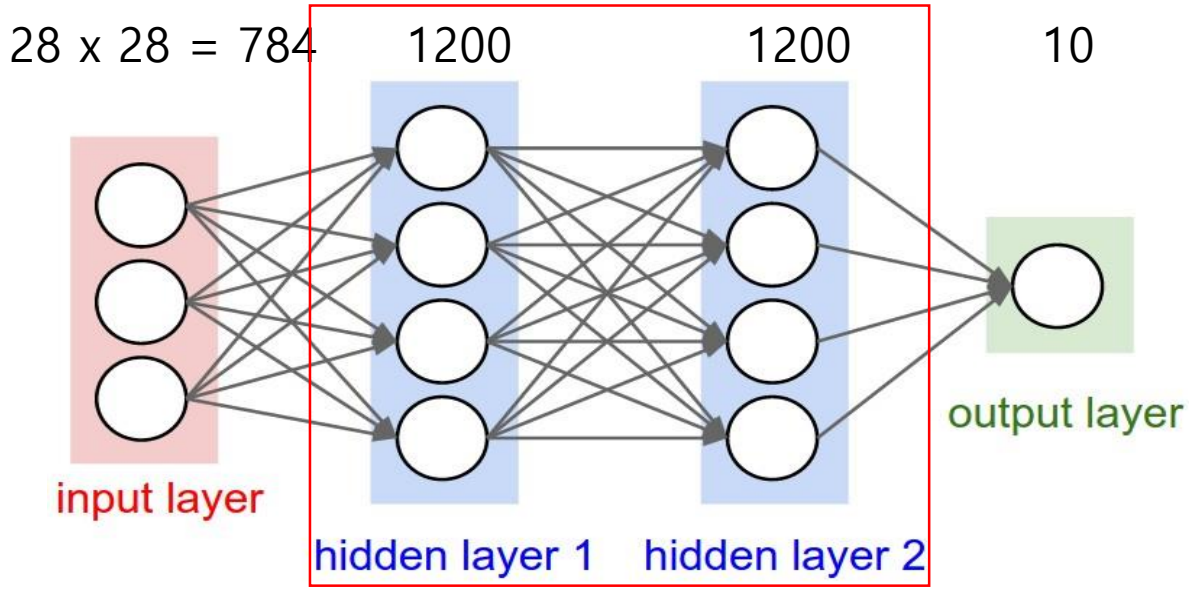
```
class FPGA
{
private:
    int fd_;
    float* data_;
    unsigned int* api_;

public:
    FPGA(off_t data_addr, off_t api_addr);
    ~FPGA();

    // return internal pointer for the data
    float* matrix(void);
    float* vector(void);

    // perform matrix multiplication and return output array pointer
    const float* run();

    // input vector size: M
    // matrix size: N by M
    // output vector size: N
    // O = M * I
    void largeMV(const float* mat, const float* input,
        float* output, int M, int N);
};
```

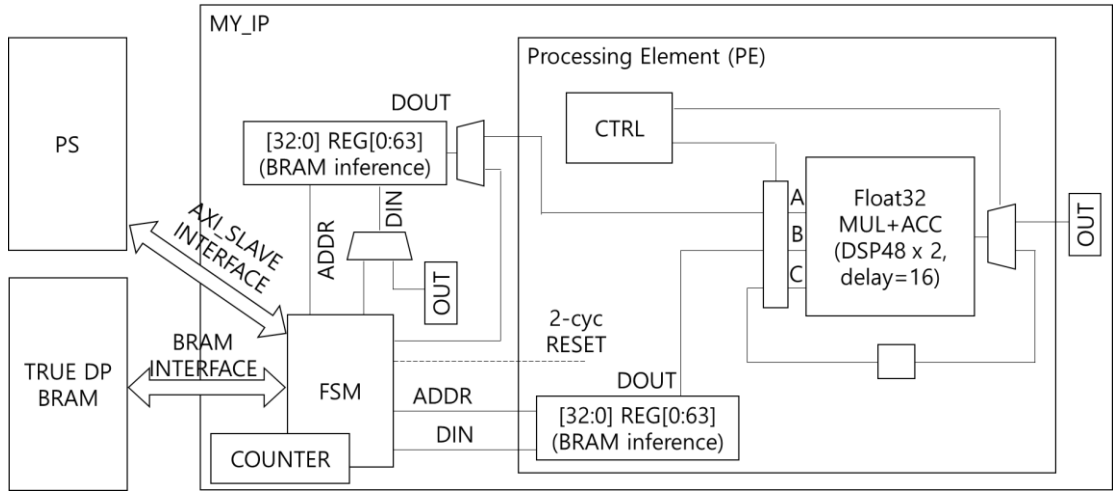
M*V



Problem?

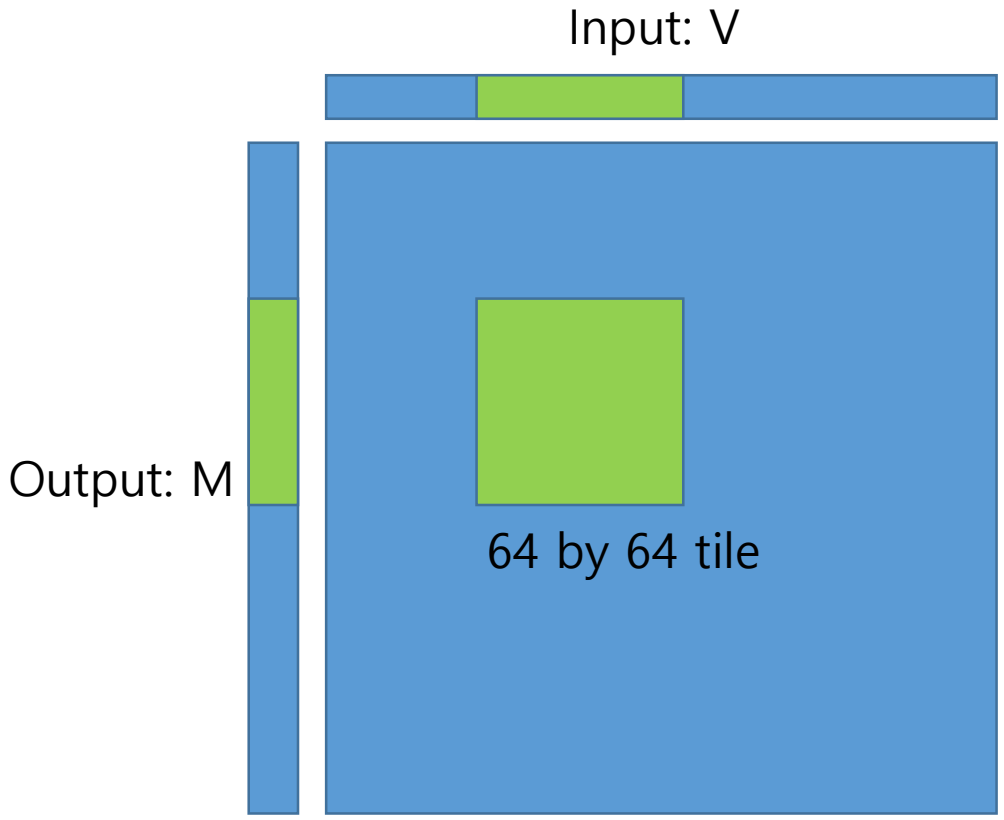
- Our board only supports 64 by 64 matrix-vector multiplication
  - We will use the FPGA board in the following sessions
- MLP requires 1200 by 1200 matrix-vector multiplication





Solution

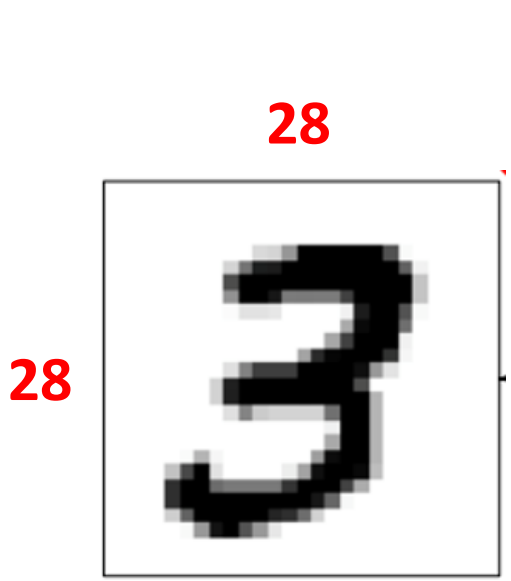
- Tiling: calculate the matrix-vector multiplication by splitting the matrix into small tiles which are supported by the accelerator

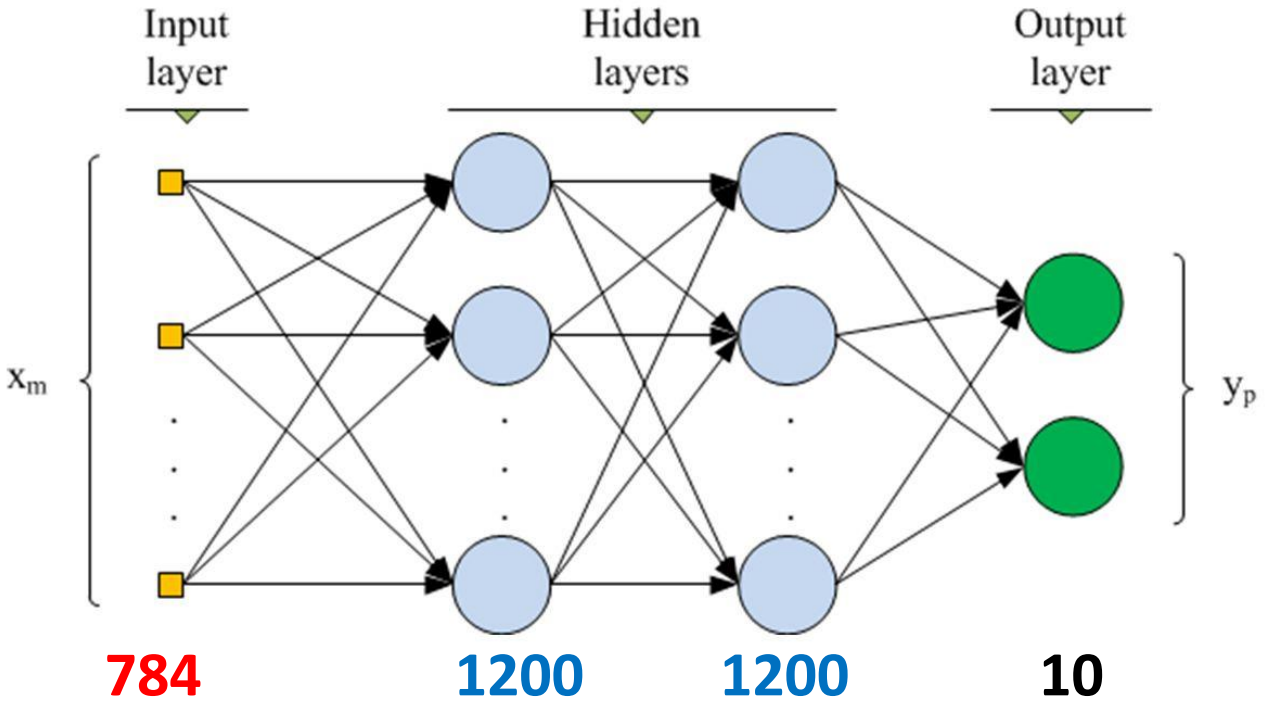


You should try different block size as follows:
block size(M, V): (64, 64), (16, 16), (8, 16), (16, 8)

Lab 2: Overview

- Goal
  - Implement **matrix-vector(MV) multiplication** in C++
    - By splitting the matrix into tiles**(=block operation)**
  - Integrate MV multiplication into the pretrained model(MLP)
    - On MNIST





Skeleton Code

```
.
|- Makefile
|- src
|    |- fpga_api.cpp      # IMPLEMENT THIS!(only MV multiplication)
|    |- digit_dnn.cpp     # don't need to edit
|    `- py_lib.cpp        # don't need to edit
|
|- eval.py                # evaluate the pretrained model
|                         # using MV multiplication that you implemented
|
|- pretraiend_weights     # MLP(784-1200-1200-10)
|- include
`- dataset
```

Environment

- Server
  - Ubuntu 16.04, Python 2.7
  - **No GPU**
- How to connect a server
  - ssh root@147.46.15.78 -p YOUR_PORT
  - Password: YOUR_PASSWORD
- For example:
  - YOUR_PORT: 7000

Implement MV multiplication

- Edit `src/fpga_api.cpp`

```
// src/fpga_api.cpp
void FPGA::largeMV(const float* large_mat, const float* input,
                   float* output, int M, int N)
{
    // IMPLEMENT HERE
    ...
}
```

- [Note] Don’t need to implement reshape(28x28->784) and activation functions such as ReLU, Softmax

Download dataset and pretrained weights

```
root@833b6a966b05:/base# sh download.sh
--2019-03-10 04:41:47--  https://dl.dropbox.com/s/mdwy0kzf57nfl5f/t10k-images.idx3-ub
yte
Resolving dl.dropbox.com (dl.dropbox.com)... 162.125.80.6, 2620:100:6030:6::a27d:5006
Connecting to dl.dropbox.com (dl.dropbox.com)|162.125.80.6|:443... connected.
HTTP request sent, awaiting response... 302 FOUND
Location: https://dl.dropboxusercontent.com/s/mdwy0kzf57nfl5f/t10k-images.idx3-ubyte
[following]
--2019-03-10 04:41:48--  https://dl.dropboxusercontent.com/s/mdwy0kzf57nfl5f/t10k-ima
ges.idx3-ubyte
Resolving dl.dropboxusercontent.com (dl.dropboxusercontent.com)... 162.125.80.6, 2620
:100:6030:6::a27d:5006
Connecting to dl.dropboxusercontent.com (dl.dropboxusercontent.com)|162.125.80.6|:443
... connected.
HTTP request sent, awaiting response... 200 OK
```

Build your MV multiplication

```
root@833b6a966b05:/base# make
protoc -I=./proto --cpp_out=./proto proto/caffe.proto
g++ -fPIC -std=c++11 -O3 -I ./include -I./proto -o build/py_lib.o -c src/py_lib.cpp
g++ -fPIC -std=c++11 -O3 -I ./include -I./proto -o build/caffe.pb.o -c proto/caffe.pb
.cc
g++ -fPIC -std=c++11 -O3 -I ./include -I./proto -o build/digit_dnn.o -c src/digit_dnn
.cpp
g++ -fPIC -std=c++11 -O3 -I ./include -I./proto -o build/fpga_api.o -c src/fpga_api.c
pp
g++ -shared -o build/libpylib.so  build/py_lib.o build/caffe.pb.o build/digit_dnn.o b
uild/fpga_api.o -lprotobuf
```

Test MV multiplication on MNIST

```
root@833b6a966b05:/base# python eval.py
read dataset...
('images', (10000, 28, 28))
create network...
run test...
{'accuracy': 0.098,
 'avg_num_call': 627,
 'm_size': 64,
 'total_image': 10000,
 'total_time': 26.326011896133423,
 'v_size': 64}
```

Homework

- Requirements
  - Result
    - Attach your codes(e.g., fpga_api.cpp)
    - Attach your results on MNIST with [student_number, name]
      - Block size(M, V): (64, 64), (16, 16), (8, 16), (16, 8)
      - How many times blockMV() is called?
      - Accuracy
      - Total time
  - Report
    - Explain MV multiplication that you implemented
    - In your own words
  - **Result + Report to a .zip**
- Upload the report on ETL individually
  - Due: 3/30(MON) 23:59
    - **No Late Submission**
  - Either in Korean or in English
  - # of pages does not matter
  - **PDF only!!**

```
// src/fpga_api.cpp
void FPGA::largeMV(const float* l
                   float* output
{
    // IMPLEMENT HERE
    ...
}
```

```
root@833b6a966b05:/base# python eval.py
read dataset...
('images', (10000, 28, 28))
create network...
run test...
{'accuracy': 0.098,
 'avg_num_call': 627,
 'm_size': 64,
 'total_image': 10000,
 'total_time': 26.326011896133423,
 'v_size': 64}
```