# 1 Outline

In this assignment, you are asked to design a convolutional layer and maxpool layer.

# 2 Specification

**In this assignment, there are two tasks.**

1. **You are asked to write a Python code, writing methods for classes implementing convolutional and maxpool layer for neural networks.**
2. **You are asked to label random images from MNIST dataset.**

The implementation details are provided in the following sections.

# 3 Implementation notes: convolutional and maxpool layer

There are classes defined for each layer:

1. `nn_convolutional_layer`
2. `nn_maxpool_layer`

## 3.1 class `nn_convolutional_layer`

The class have two methods to be implemented: `forward` and `backprop`.

### 3.1.1 `forward` method

Forward method has the following format:
`nn_convolutional_layer.forward(x)`: Performs a forward pass of convolutional layer. Input parameters are as follows:
**Parameters:** $x$*:nd-array*.
4-dimensional `numpy.array` input map to the convolutional layer. Each dimension of $x$ has meaning

```
x.shape=(batch_size, input_channel_size, in_width, in_height)
```

For example, if `x.shape` returns $(16, 3, 32, 32)$, it means that `x` is an image/activation map of size $32 \times 32$, with three input channels (like RGB), and there are 16 of them treated as one batch (as in mini-batch SGD).

**Returns:** `out`:*nd-array*.
4-dimensional `numpy.array` output feature map as a result of convolution. Each dimension of `out` has meaning

```
out.shape=(batch_size, num_filter, out_width, out_height)
```

For example, if `out.shape` will return $(16, 8, 28, 28)$, for the input `x` with shape $(16, 3, 32, 32)$: if the convolutional layer has each filter size $5 \times 5 \times 3$, and there are a total of 8 filters. Note that the input width and height changes from 32 to 28. Batch size remains the same.

The forward pass of convolution will be done without zero-padding and stride of 1. So, if the input map has width $N$, and the filter size is given by $F$, the width of output map will be $N - F + 1$.

### 3.1.2 Tips for convolution operations

Implementing convolutional layer can be tricky. It may be implemented in many ways, but if it is not done properly, the operation can be slow. This is because our data is usually high-dimensional. Some of widely used libraries for deep learning, such as Pytorch or Tensorflow, uses specialized code for accelerating the convolutional forward/backprop operations. The goal in this course is, however, to build reasonably good layer for small models, but not fancy ones like public libraries.

Here we will use the following useful function to implement multi-dimensional convolution.

```
from skimage.util.shape import view_as_windows
```

The `view_as_windows` function provides a *rearranged* view of the input map, which makes it easier to perform convolution with filters. Our approach for convolution is

- Rearrange the input map using `view_as_windows`.
- perform **inner product** operation (that is, matrix multiplication) between input map and the filter.

The documents for the function can be found in the link
`https://scikit-image.org/docs/dev/api/skimage.util.html#skimage.util.view_as_windows`
**Example:** For simplicity, let's consider 2-D case. Consider an input map $x$ given by

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Suppose we would like to do convolution of $x$ and 2-by-2 filter

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

The convolution output will be 2-by-2. If you manually perform convolution by sliding the filter over $x$, you can check that the result is

$$\begin{bmatrix} 12 & 16 \\ 24 & 28 \end{bmatrix}$$

This is done by, first use `view_as_windows` to create a view into $x$ through 2-by-2 window in a sliding manner. This is done by calling

```
y=view_as_windows(x,(2,2))
```

This will create $y$, which is a 2-by-2 (output map size) where each output element is a 2-by-2 (filter size) sliding view of input $x$. Thus the shape of $y$ is (2,2,2,2). Then $y$ is reshaped to (2,2,4) by "flattening" the windowed view – this is then inner-product with filter which is reshaped to (4,1).

Below is the example Python code.

```
import numpy as np
from skimage.util.shape import view_as_windows

x=np.arange(9).reshape(3,3)+1
print('x.shape',x.shape)
print('x')
print(x)

# filter
filt=np.ones((2,2))

y=view_as_windows(x,(2,2))
print('y.shape',y.shape)
print('y')
print(y)

y=y.reshape((2,2,-1))
print('reshaped y')
print(y)

# perform convolution by dot (inner) product!
print('convolution result:')
result=y.dot(filt.reshape((-1,1)))
result=np.squeeze(result,axis=2)
print(result)
```

This will produce result output:

```
x.shape (3, 3)
x
[[1 2 3]
```

```
 [4 5 6]
 [7 8 9]]
y.shape (2, 2, 2, 2)
y
[[[[1 2]
   [4 5]]

  [[2 3]
   [5 6]]]


 [[[4 5]
   [7 8]]

  [[5 6]
   [8 9]]]]
reshaped y
[[[1 2 4 5]
  [2 3 5 6]]

 [[4 5 7 8]
  [5 6 8 9]]]
convolution result:
[[12. 16.]
 [24. 28.]]
```

Also see Figure 1 for this example.

Of course, if possible you can use other functions to do convolution – as long as they are not pre-built public libraries from Pytorch/Tensorflow/Keras, etc. Also, you can define other functions as many as you want, if needed.

#### 3.1.3 `backprop` method

Backprop method has the following format:

`nn_convolutional_layer.backprop(x,dLdy)`: Performs a backpropagation of convolutional layer. Input parameters are as follows:

**Parameters:**

- $x$ :*nd-array*.

  4-dimensional `numpy.array` input map to the convolutional layer. The shape of $x$ is given by

  ```
  x.shape=(batch_size, input_channel_size, in_width, in_height)
  ```

  `batch_size` is the batch size, `input_channel_size` is the number of input channels, `in_width` and `in_height` are the width and height of input map.

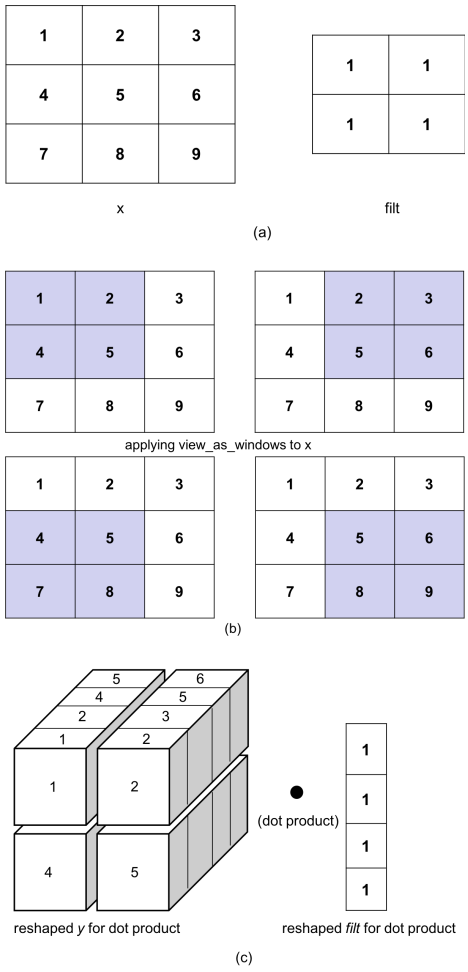


Figure 1: (a) Input map $x$ and filter to perform convolution. (b)apply `view_as_window` to $x$ to create a sliding-window view $y$ into $x$. (c) reshape $y$ and filter, and perform a dot product for the convolution.

- `dLdy` :*nd-array*.

  The upstream gradient passed to the convolutional layer. It will have shape

  ```
  dLdy.shape=(batch_size, num_filter, out_width, out_height)
  ```

  `batch_size` is the batch size, `num_filter` is the number of filters, `out_width` and `out_height` are the width and height of the output activation map.

  This is the upstream gradient $\frac{\partial L}{\partial y}$, where $y$ denotes the output of this layer, passed from the next layer. Thus `backprop` need to return the following gradient,

which will be passed as the upstream gradient to the previous layer:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x}$$

Also there are two trainable parameters $W$ and $b$. In order to perform gradient descent, the following should be evaluated: for $W$

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial W}$$

and for $b$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial b}$$

Note that $W$ and $b$ has the following shape.

```
W.shape=(num_filter,in_channel_size,filt_width,filt_height)
```

and

```
b.shape=(1,num_filter,1,1)
```

`num_filter` is the number of filters, `in_channel_size` is the number of the input channels, `filt_width` and `filt_height` are the width and height of the filter. Always have in mind that, `dLdW` has the same shape as `W`, and `dLdb` as `b`.

**Returns:**

- `dLdx`:*nd-array*.

  4-dimensional `numpy.array` output of gradient $\frac{\partial L}{\partial x}$, which will be passed onto the previous layer. The shape is given by

  ```
  dLdx.shape=(batch_size, input_channel_size, in_width, in_height)
  ```

- `dLdW` :*nd-array*.

  4-dimensional `numpy.array` output of gradient $\frac{\partial L}{\partial W}$, which will be passed onto the previous layer. The shape is given by

  ```
  dLdW.shape=(num_filter, in_channel_size, filter_width, filter_height)
  ```

- `dLdb` :*nd-array*.

  4-dimensional `numpy.array` output of gradient $\frac{\partial L}{\partial b}$, which will be passed onto the previous layer. The shape is given by

  ```
  dLdb.shape=(1,num_filter, 1,1)
  ```

#### 3.1.4 Tips for backprop operations

- Think carefully what the backprop should be. Start with a small example, and try to calculate $\frac{\partial L}{\partial x}$, $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial b}$. **You will recognize that, the backprop will also use the convolution function.**
- So, I recommend that you separately define a function which performs a convolution of two 4-dimensional arrays. You can use it when you implement `forward` method, and also for implementing `backward` method.
- Be careful, especially when calculating backprops, **you may need to use some zero padding.** Also, **the filter `W` may have to be inversely rearranged before convolution.** Again, work on small examples to see what these mean.

### 3.2 class `nn_maxpool_layer`

The objects of class `nn_maxpool_layer` are created as follows:

```
maxpool_obj=nn_maxpool_layer(stride=2,pool_size=2)
```

Here we pass two parameters, `stride` is the stride of `pool_size` is the size of window over which we take the max pooling. In this example, we slide 2-by-2 window, with stride 2, and take the maximum value from the 2-by-2 windowed view of input.

The following two methods, `forward` and `backprop` need to be implemented.

#### 3.2.1 `forward` method

Forward method has the following format:

`nn_maxpool_layer.forward(x)`: Performs a forward pass of convolutional layer. Input parameters are as follows:

**Parameters:** $x$:*nd-array*.

4-dimensional `numpy.array` input map to the convolutional layer. Each dimension of $x$ has meaning

```
x.shape=(batch_size, input_channel_size, in_width, in_height)
```

For example, if `x.shape` returns $(16, 3, 32, 32)$, it means that `x` is an image/activation map of size $32 \times 32$, with three input channels (like RGB), and there are 16 of them treated as one batch (as in mini-batch SGD).

**Returns:** `out`:*nd-array*.

4-dimensional `numpy.array` output feature map as a result of convolution. Each dimension of `out` has meaning

```
out.shape=(batch_size, input_channel_size, out_width, out_height)
```

For example, if `out.shape` will return $(16, 3, 16, 16)$, for the input `x` with shape $(16, 3, 32, 32)$, assuming that the `stride`=2 and `pool_size`=2.

### 3.2.2 Tips for maxpool operations

Similar to convolutional layer, it is convenient to use `view_as_windows` function. The function takes the parameter `step`, which is equivalent to the stride. So you can create, for example 2-by-2 windowed view into the input, with the stride 2.

Below is an example. For simplicity, let's consider 2-D case. Consider an input map $x$ given by

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

Suppose we would like to perform max pooling over 2-by-2 sliding window with stride of 2. Below is the example Python code.

```
import numpy as np
from skimage.util.shape import view_as_windows

x=np.arange(16).reshape(4,4)+1
print('x.shape',x.shape)
print('x')
print(x)

# filter
filt=np.ones((2,2))

y=view_as_windows(x,(2,2),step=2)
print('y.shape',y.shape)
print('y')
print(y)
```

This will produce result output:

```
x.shape (4, 4)
x
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]
y.shape (2, 2, 2, 2)
y
[[[[ 1  2]
   [ 5  6]]

  [[ 3  4]
   [ 7  8]]]
```

```
[[[ 9 10]
  [13 14]]

 [[11 12]
  [15 16]]]]
```

#### 3.2.3 `backprop` method

Backprop method has the following format:
`nn_maxpool_layer.backprop(x,dLdy)`: Performs a backpropagation of convolutional layer. Input parameters are as follows:
**Parameters:**

- $x$ :*nd-array*.

  4-dimensional `numpy.array` input map to the convolutional layer. The shape of $x$ is given by

  ```
  x.shape=(batch_size, input_channel_size, in_width, in_height)
  ```

  `batch_size` is the batch size, `input_channel_size` is the number of input channels, `in_width` and `in_height` are the width and height of input map.

- `dLdy` :*nd-array*.

  The upstream gradient passed to the convolutional layer. It will have shape

  ```
  dLdy.shape=(batch_size, input_channel_size, out_width, out_height)
  ```

  `batch_size` is the batch size, `input_channel_size` is the number of input channels, `out_width` and `out_height` are the width and height of the output activation map.

**Returns:**

- `dLdx`:*nd-array*.

  4-dimensional `numpy.array` output of gradient $\frac{\partial L}{\partial x}$, which will be passed onto the previous layer. The shape is given by

  ```
  dLdx.shape=(batch_size, input_channel_size, in_width, in_height)
  ```

In summary, **you are asked to complete** `forward` **and** `backprop` **methods for** `nn_convolutional_layer` **and** `nn_maxpool_layer` **classes.**

# 4 Implementation notes: studying MNIST dataset

## 4.1 Introduction

MNIST dataset is a database of images of handwritten digits. There are handwritten numbers of 0 to 9. The dataset contains 60,000 training images and 10,000 test images. The goal of a classifier is to classify the digits correctly to numbers 0 to 9. So this is a classification problem of 10 classes (0 to 9).

The size of single image is 28-by-28 pixels. The image is grayscale, so there is only one color channel (unlike RGB three-channels color images from CIFAR-10).

In the `hw4_mnist.py` file, there is a module that downloads and extracts the training and test dataset. We create four variables, `X_train, y_train, X_test, y_test`.

1. `X_train` : contains the handwritten images, and has shape (60000,1,28,28). The meanings are: 60000 is the number of images, 1 is the number of channels (grayscale means only 1 channel), 28 is width and 28 is height of the image.

2. `y_train` : has shape (60000,). The meanings are: 60000 is the number of images. This data contains the **ground truth labels** of the corresponding images, and contains the number between 0 and 9. For example, if `X_train[0]` contains a handwritten image for number 5, then `y_train[0]` is 5.

3. `X_test` : has shape (10000,1,28,28). This contains the handwritten images for testing.

4. `y_test` : has shape (10000,). This contains the ground truth labels for testing images.

The script `hw4_mnist.py` will print out three randomly selected images from MNIST dataset. **Your task is to put proper titles to these three plots.** An example is shown in Figure 2. As shown in the figure, you should modify `Q5` part in `hw4_mnist.py`, so that the randomly shown images have proper titles, as the numbers in the red circle of Figure 2.

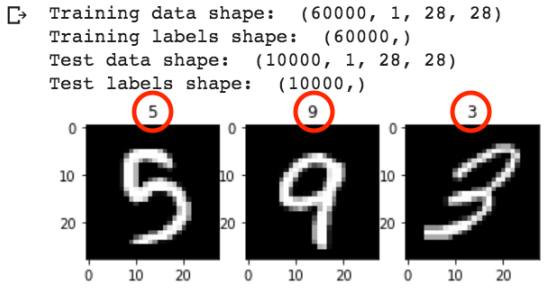


Figure 2: Example of plotting MNIST images.

# 5 What to submit

You will be given two files `hw4_nn.py` and `hw4_mnist.py`. **You are asked to complete Q1–Q4 parts in `hw4_nn.py`, and Q5 in `hw4_mnist.py`.**

- Submit **TWO files,**, a modified Python file `hw4_nn.py` and `hw4_mnist.py`.
- Upload your files at Blackboard before deadline. (Please submit the file in time, no late submission will be accepted).

# 6 How your module will be graded

For the MNIST part, the grading is clear: whether your titling of images are correct.

For the convolutional/maxpool layer part, the following criteria are used.

Suppose you have corrected implemented the forward and backprop methods. Let the input to convolutional layer by $x$, and the output $y$; then the convolutional layer can be considered as some function $f$ such that

$$y = f(x)$$

Now suppose we create a small vector $\delta$. Then the following first-order approximation will hold:

$$f(x + \delta) \approx f(x) + \frac{\partial y}{\partial x} \cdot \delta$$

Next we will create some arbitrary $\frac{\partial L}{\partial y}$ which is the upstream gradient. We move $f(x)$ to the left hand side, then multiply $\frac{\partial L}{\partial y}$ on both sides, which leads to

$$\frac{\partial L}{\partial y}[f(x + \delta) - f(x)] \approx \frac{\partial L}{\partial y}\frac{\partial y}{\partial x} \cdot \delta = \frac{\partial L}{\partial x} \cdot \delta$$

This means that, if you have implemented $f()$ (forward) and $\frac{\partial L}{\partial x}$ (backprop) correctly, the above equation will hold for small enough $\delta$!

The later part of file `hw4_nn.py` contains the above method to measure accuracy of your implementation. As in the last lines of `hw4_nn.py`, your accuracy test must yield numbers close to 100%. **The target accuracy is 90%.** If your implementation is correct, the number will be very close to 100%. An example test run with correctly implemented modules is shown in Fig. 3.

As a general advice, first test your module with very simple inputs and filters – so that you can easily calculate the forward and backprop by hand. Start with setting batch size and input channel size, filter numbers to 1, and see if you can get correct results.

There are 4 backprops to be tested: $\frac{\partial L}{\partial x}$, $\frac{\partial L}{\partial W}$ $\frac{\partial L}{\partial b}$ for convolutional, and $\frac{\partial L}{\partial x}$ for max pooling.

```
dLdx test
exact change 20.311882259667694
apprx change 20.311882259694034
dLdW test
exact change 2217.1044512922053
apprx change 2217.1044512921753
dLdb test
exact change -373.0871411058653
apprx change -373.0871411058695
dLdx test for pooling
exact change -479.1293234747733
apprx change -479.12932325856315
accuracy results
conv layer dLdx 99.99999999941095 %
conv layer dLdW 99.9999999999897 %
conv layer dLdb 99.99999999996118 %
maxpool layer dLdx 99.99999987115955 %
```

Figure 3: Console output of the accuracy test.

# 7 Grading

- 16 points for Q1–Q4 done correctly. Specifically, if each of 4 accuracy tests gets more than 90%, you get 4 points for each. So the total is 16 points.
- 2 points for each of 4 accuracy test results between 50–90%.
- 1 points for each of 4 accuracy test results below 50%.
- 4 points for Q5 done correctly.
- 1 points for Q5 getting wrong answers.
- 0 point if you do not submit the file by deadline.

In the blackboard, you can upload your files as many times as you like, before the deadline. The last uploaded file will be used for grading. After deadline, the submission menu will be closed and you will not be able to make submission.