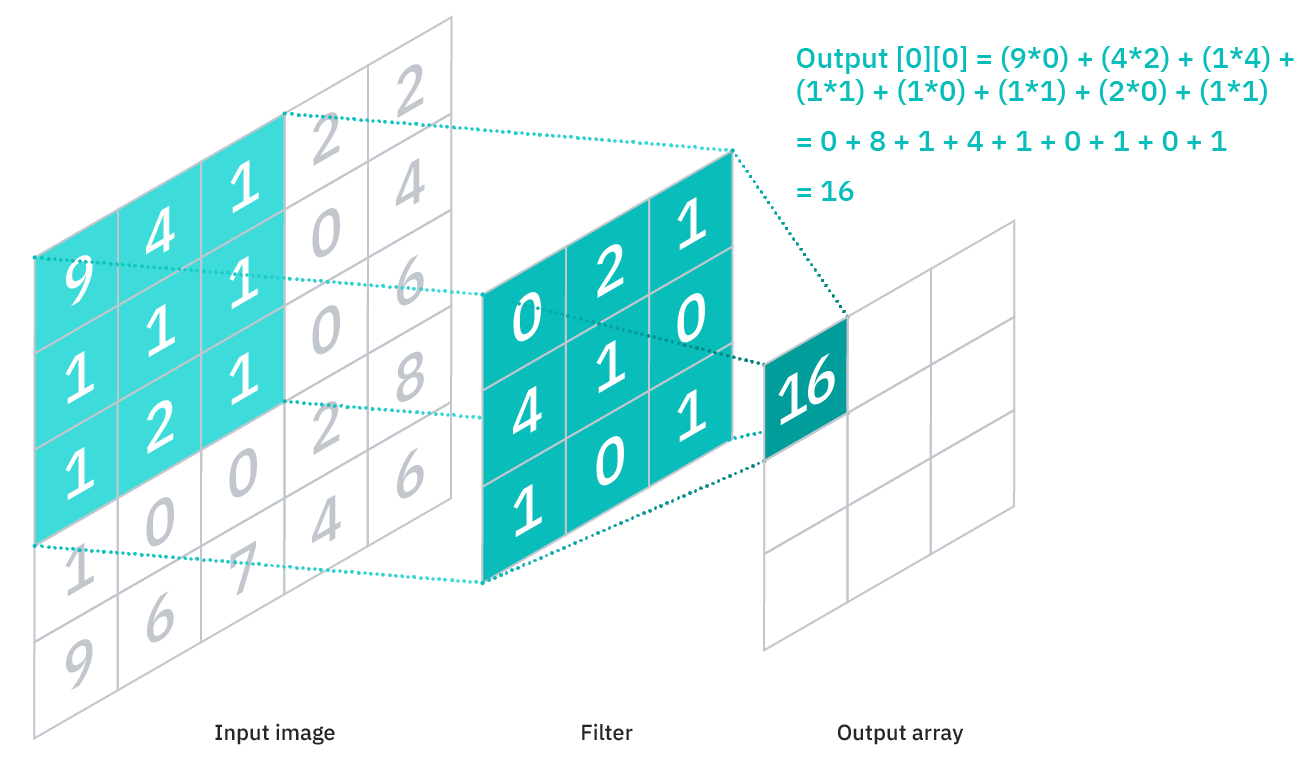
Convolution Layer

“Sliding Window Method”:



Used model for storage:

Block2D:

Matrix[depth] vals;

Matrix:

double[rows][cols] mat;

where, a Block2D object can be formulated by: .

A kernel, on the other hand, is a collection of Block2D objects in an array, which can be formulated by

The **bias**, however, can be a simple vector on the depth, and every kernel just add the same bias no matter where it is.

Formulation:

Where, ***s*** is the **stride** and ***p*** is the **pad**.

The value of where are out of the ranges specified would yield 0, or equivalently, is zero-padded automatically.

Size of output:

As for the basic, elementary process in Convolution, we can see that:

can be reduced into the form:

.

.

With this definition of , we can now write:

Derivatives:

Derivative with respect to kernel :

Note that the position of the stride *s* is multiplied to the kernel counters *i* and *j*, instead of the position of the cell. This is what I call an *instride* (stride to the input).

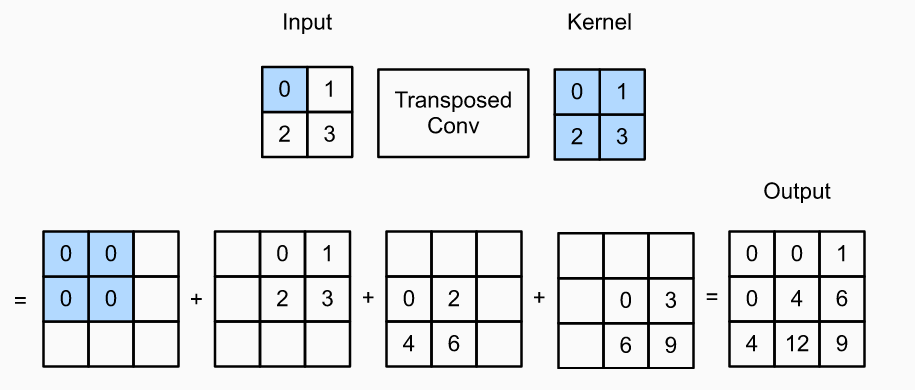
In my code, this is denoted by a **negative** **stride** (but if to be mathematically rigorous, this should be denoted by a **reciprocal stride**.)

Size of output: .

Derivative with respect to input :

When we define , then:

This kind of “inverse of convolution” is sometimes called **transpose convolution**, or **deconvolution**. Padding is used to **remove the outermost boundary from the output**, instead of adding an outermost boundary to the input.



Derivative with respect to bias :

If we define , then .

Time Complexity: if accessed sequentially.

Space Complexity:

Parameters:

Gradients: the same amount as parameters for each gradient type.

Storing input, output for the intermediate layers:

Storing input, output for the last layer:

Comparing to a **fully connected layer** with the same input and output size:

Time Complexity: if accessed sequentially.

Space Complexity:

Parameters:

Gradients: the same amount as parameters.

Storing input, output for each layer: the same as ConvNet.

Considering the Space and Time Complexity of convolution networks, they could be better optimized than dense layers to reduce array access while looping (as the parameters are being reused frequently). However, it is impossible to reduce the time complexity of the dense layers further, as the high space complexity of storing the parameters guarantees a minimum amount of array access calls to the order of in each pass.

For a normal network, where and , the time and space complexities of CNN is generally smaller, while having better stability over spatial translation, rotation and resizing of the features.

**Speed comparison over different devices:**

Speed of convolution layers in my code implementation in *NeuralNetwork3 v202206051700 [CNNImgTest\_MNIST]*:

Layer orders:

Keys:

Or in short, :

Layers =

Dimension of tensors between layers:

Estimated number of operations:

***14,000 per epoch, 5 epochs***

Time taken for 5 epochs of a total of 70,000 data = 14,000 data / epoch \* 5:

= 470,924.2277 ms

Time taken per datum per epoch: 6.72748897 ms / datum

Estimated time taken per operation: 21.130641 ns / operation

Estimated rate of operation: 47.324641 MHz

Clock rate of processor used: 1.19 GHz

Instruction cycle ≈ 25.1454628 x clock cycle

***60,000 per epoch, 5 epochs***

Time taken for 5 epochs of a total of 300,000 data = 60,000 data / epoch \* 5:

= 2,025,139.863 ms

Time taken per datum per epoch: 6.75046621 ms / datum

Estimated time taken per operation: 21.2028112 ns / operation

Estimated rate of operation: 47.1635573 MHz

Clock rate of processor used: 1.19 GHz

Instruction cycle ≈ 25.2313453 x clock cycle

***14,000 per epoch, 5 epochs (PyTorch)***

Time taken for 5 epochs of a total of 70,000 data = 14,000 data / epoch \* 5:

= 333,893.5028 ms

Time taken per datum per epoch: 4.76990718 ms / datum

Estimated time taken per operation: 14.9819936 ns / operation

Estimated rate of operation: 66.7467913 MHz

Clock rate of processor used: 1.19 GHz

Instruction cycle ≈ 17.8285724 x clock cycle

***14,000 per epoch, 5 epochs (PyTorch, on Google Colab CPU)***

Time taken for 5 epochs of a total of 70,000 data = 14,000 data / epoch \* 5:

= 235,976.7513 ms

Time taken per datum per epoch: 3.37109645 ms / datum

Estimated time taken per operation: 10.5884126 ns / operation

Estimated rate of operation: 94.4428629 MHz

Clock rate of processor used: 2.30GHz

Instruction cycle ≈ 24.353349 x clock cycle

**Accuracy comparison over my algorithm vs other implementations:**

Our implementation on *NeuralNetwork3 v202206051700 [CNNImgTest\_MNIST]*

Number of epochs: 5

Datum per epoch: 14,000

Total number of datum: 70,000

Test Precision: 90.08016032064128%

Confusion Matrix:

A screenshot of a computer

Description automatically generated with medium confidence

Our implementation on *NeuralNetwork3 v202206051700 [CNNImgTest\_MNIST]*

MSE Loss used.

Number of epochs: 5

Datum per epoch: 60,000

Total number of datum: 300,000

Test Precision: 97.69539078156313%

Training Graph:

Graphical user interface, text

Description automatically generated

Confusion Matrix:

A screenshot of a computer

Description automatically generated

Our implementation on *NeuralNetwork3 v202206051700 [CNNImgTest\_MNIST]*

Cross Entropy Loss used.

Number of epochs: 5

Datum per epoch: 14,000

Total number of datum: 70,000

Test Precision: 94.18837675350701%

Training Graph:

Graphical user interface, application, Word

Description automatically generated

Our implementation on *NeuralNetwork3 v202206051700 [CNNImgTest\_MNIST]*

Cross Entropy Loss used. Conv layer 2: depth changes from 16 to 6.

Number of epochs: 5

Datum per epoch: 14,000

Total number of datum: 70,000

Test Precision: 96.19238476953907%

Training Graph:

Graphical user interface

Description automatically generated

PyTorch implementation: *mnist.py* from AIST2602 course, CUHK 2021 ©.

The accuracy actually rises pretty rapidly in the first 4,000 data in **first epoch**.

Cross Entropy Loss used. Same configuration as the immediately previous one.

Number of epochs: 5

Datum per epoch: 14,000

Total number of datum: 70,000

Test Accuracy: 98 - 99% (PyTorch accuracy)

Training Graph:

Graphical user interface, application, Word

Description automatically generated

**Conclusion**

PyTorch implementation is more efficient and accurate.