CNN Architecture

ML Instruction Team, Fall 2022

CE Department
Sharif University of Technology

Ali Sharifi-Zarchi
Behrooz Azarkhalili
Arian Amani
Hamidreza Yaghoubi

Brief History of Computer Vision

What is Computer Vision?

Figure: Source

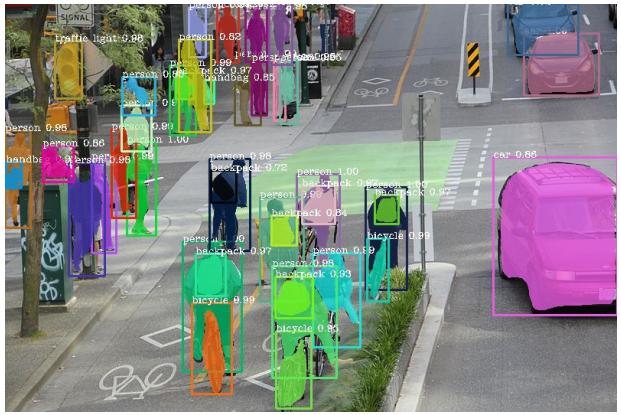


Figure: Source

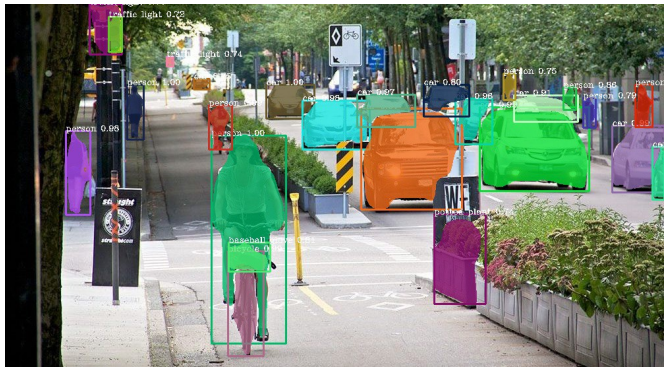


Figure: Source

Brief History of Computer Vision

Before Deep Learning: Hand-Crafted Features

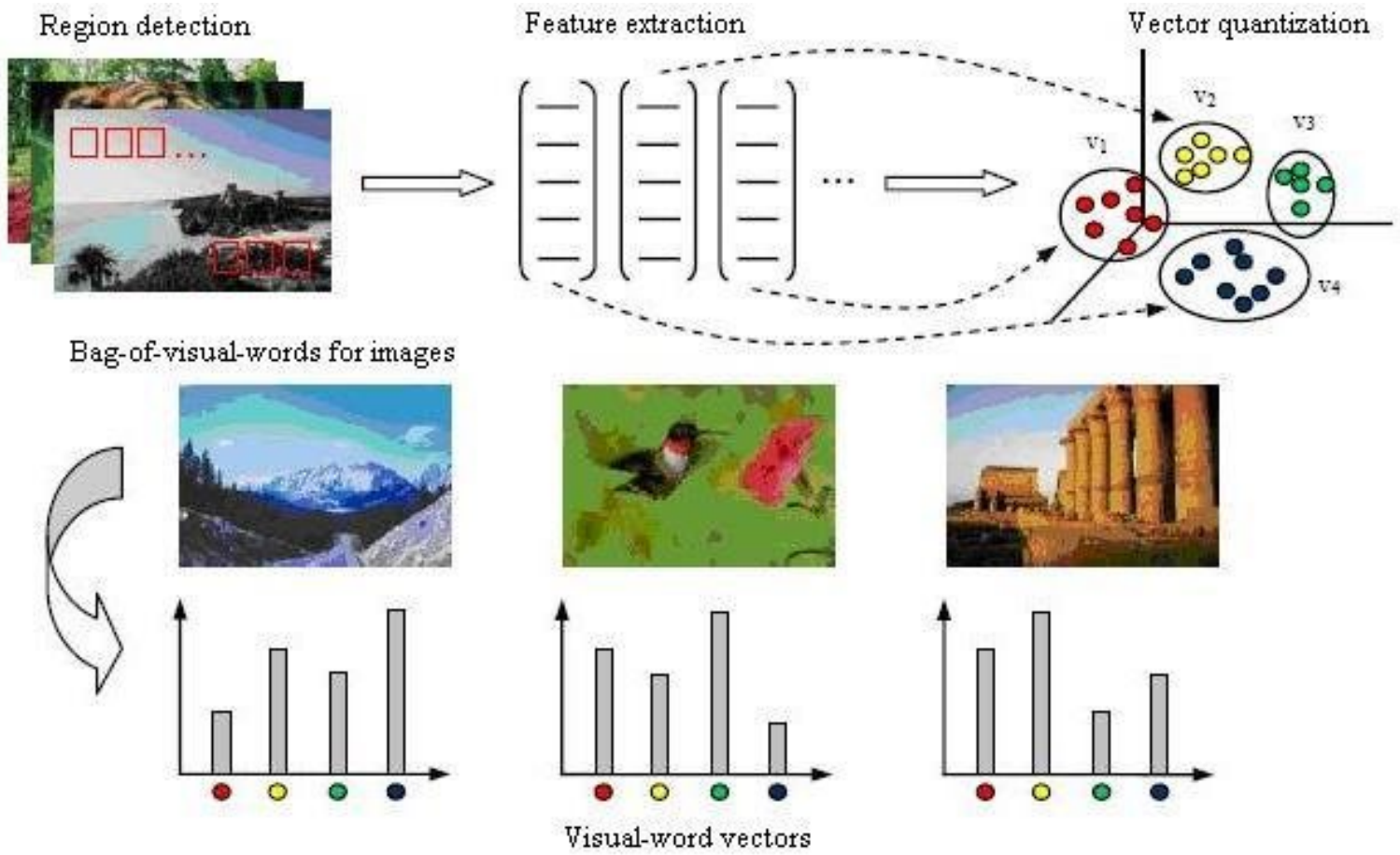


Figure: [1]

Brief History of Computer Vision

Before Deep Learning: Hand-Crafted Features

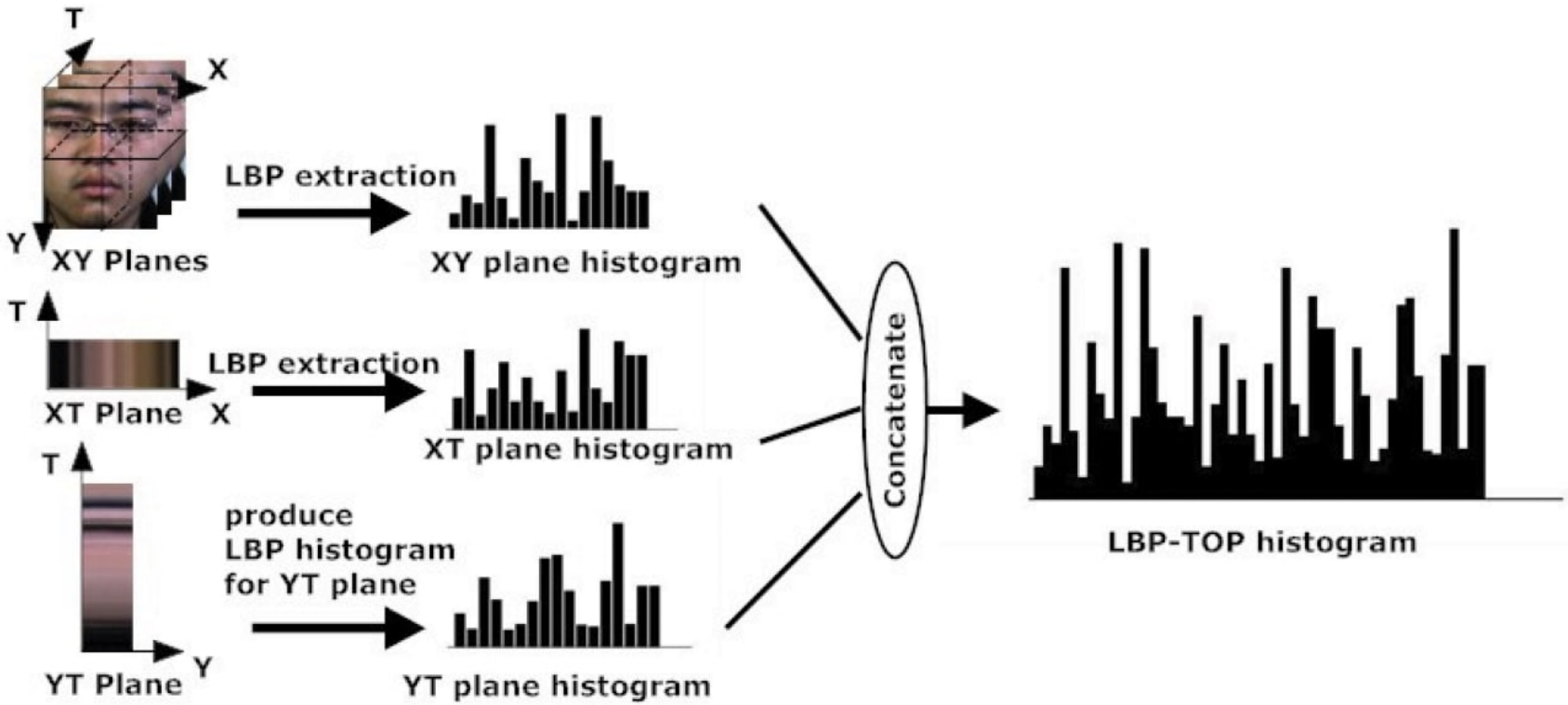


Figure: [2]

Brief History of Computer Vision

Effect of Deep Learning: Comparison between Deep Learning and Traditional Models

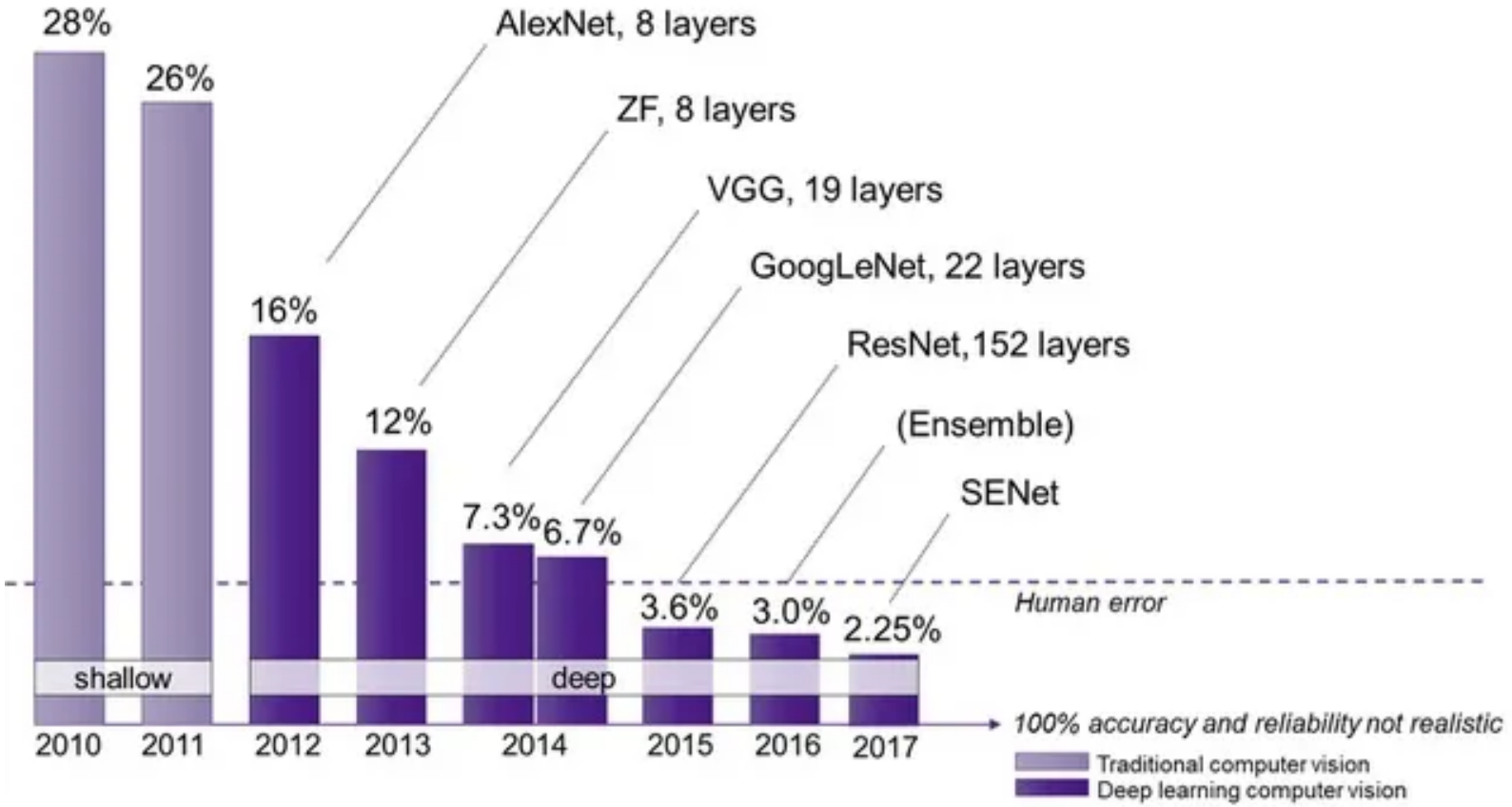
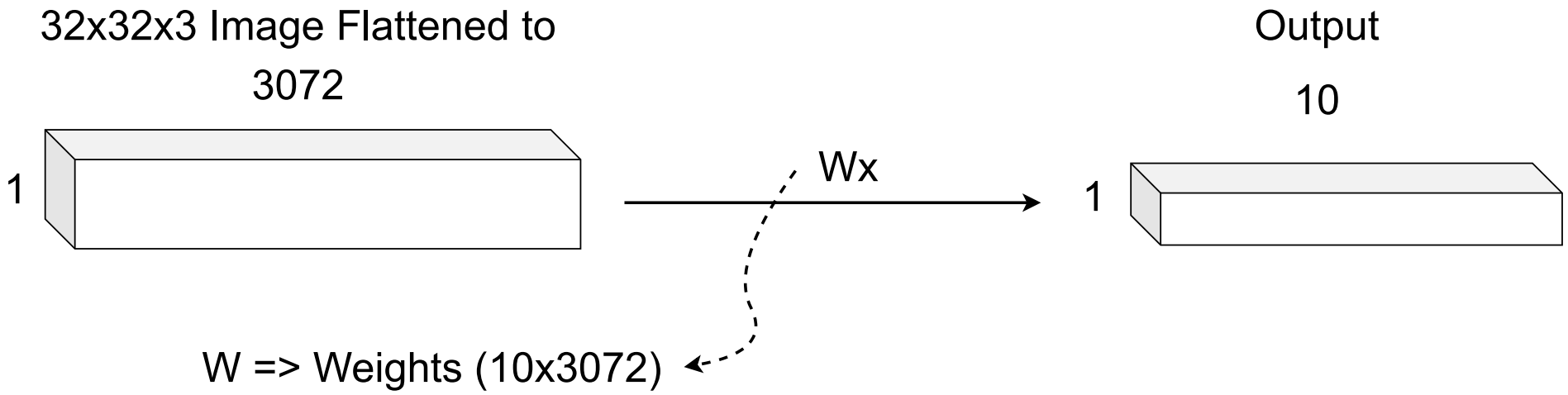


Figure: ImageNet Competetion Results Over Time, Source

CNNs

What we've been using:

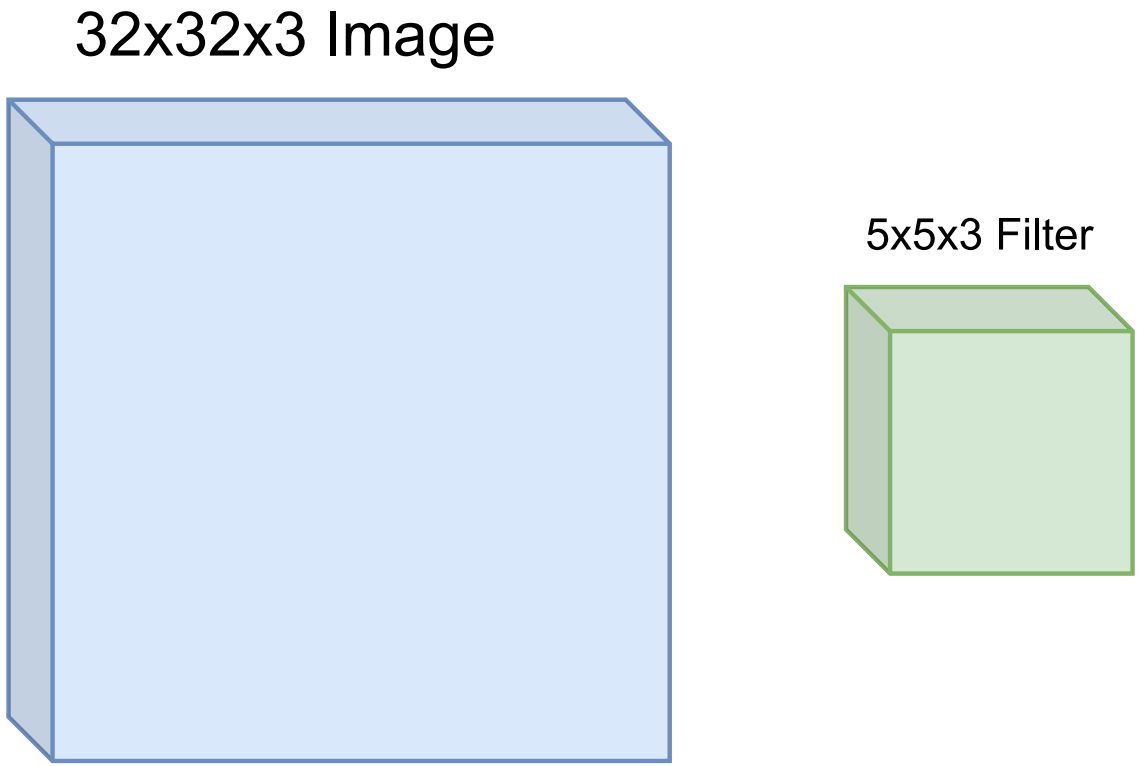
- Fully Connected Layers



CNNs

What we're going to learn:

- Convolutional Layer



CNNs vs. FCNs

First of all...

CNNs vs. FCNs

Why use ConvNets instead of Fully Connected Networks with images?

CNNs vs. FCNs

Let's do an experiment.

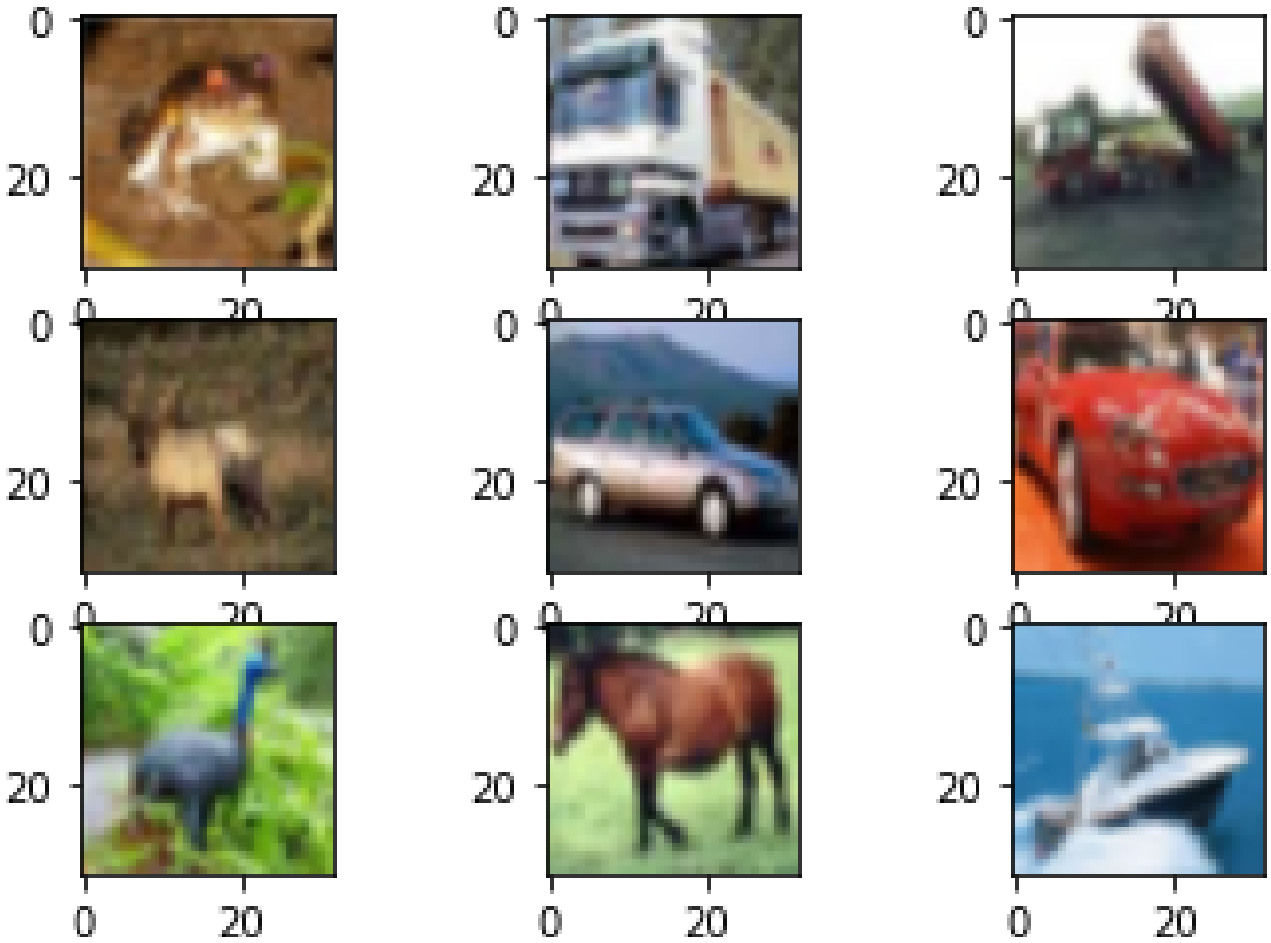
CNNs vs. FCNs

We're going to use the CIFAR10 dataset

```
# example of loading the cifar10 dataset
from matplotlib import pyplot
from keras.datasets import cifar10
# load dataset
(trainX, trainy), (testX, testy) = cifar10.load_data()
# summarize loaded dataset
print('Train: X=%s, y=%s' % (trainX.shape, trainy.shape))
print('Test: X=%s, y=%s' % (testX.shape, testy.shape))
# plot first few images
for i in range(9):
  # define subplot
  pyplot.subplot(330 + 1 + i)
  # plot raw pixel data
  pyplot.imshow(trainX[i])
# show the figure
pyplot.show()
```

CNNs vs. FCNs

We're going to use the CIFAR10 dataset



CNNs vs. FCNs

First, we train on a Fully Connected Network

```
model = Sequential()
model.add(Dense(2048, input_dim=32*32*3, activation='relu'))
model.add(Dense(1024, activation='relu'))
model.add(Dense(512, activation='relu'))
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])


model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_15 (Dense)            (None, 2048)              6293504

 dense_16 (Dense)            (None, 1024)              2098176

 dense_17 (Dense)            (None, 512)               524800

 dense_18 (Dense)            (None, 128)               65664

 dense_19 (Dense)            (None, 10)                1290

=================================================================
Total params: 8,983,434
Trainable params: 8,983,434
Non-trainable params: 0
_________________________________________________________________
```

Figure: FCN with around 9M parameters

CNNs vs. FCNs

First, we train on a Fully Connected Network

```
with tf.device('/device:GPU:0'):
  h = model.fit(dense_train, trainY, batch_size=128, epochs=20, verbose=1, validation_split=0.3)
```

```
Epoch 1/20
274/274 [==============================] - 3s 9ms/step - loss: 2.0624 - accuracy: 0.2470 - val_loss: 1.8395 - val_accuracy: 0.3358
Epoch 2/20
274/274 [==============================] - 2s 7ms/step - loss: 1.7670 - accuracy: 0.3629 - val_loss: 1.6996 - val_accuracy: 0.3894
Epoch 3/20
274/274 [==============================] - 2s 7ms/step - loss: 1.6788 - accuracy: 0.3949 - val_loss: 1.7055 - val_accuracy: 0.3825
Epoch 4/20
274/274 [==============================] - 2s 7ms/step - loss: 1.6096 - accuracy: 0.4198 - val_loss: 1.5953 - val_accuracy: 0.4262
Epoch 5/20
274/274 [==============================] - 2s 7ms/step - loss: 1.5433 - accuracy: 0.4465 - val_loss: 1.6123 - val_accuracy: 0.4380
Epoch 6/20
274/274 [==============================] - 2s 7ms/step - loss: 1.5039 - accuracy: 0.4607 - val_loss: 1.5651 - val_accuracy: 0.4429
Epoch 7/20
274/274 [==============================] - 2s 7ms/step - loss: 1.4629 - accuracy: 0.4749 - val_loss: 1.5050 - val_accuracy: 0.4704
Epoch 8/20
274/274 [==============================] - 2s 8ms/step - loss: 1.4190 - accuracy: 0.4884 - val_loss: 1.5047 - val_accuracy: 0.4656
Epoch 9/20
274/274 [==============================] - 2s 8ms/step - loss: 1.3906 - accuracy: 0.4997 - val_loss: 1.4724 - val_accuracy: 0.4795
Epoch 10/20
274/274 [==============================] - 2s 7ms/step - loss: 1.3650 - accuracy: 0.5088 - val_loss: 1.5368 - val_accuracy: 0.4519
Epoch 11/20
274/274 [==============================] - 2s 7ms/step - loss: 1.3220 - accuracy: 0.5265 - val_loss: 1.4542 - val_accuracy: 0.4876
Epoch 12/20
274/274 [==============================] - 2s 8ms/step - loss: 1.2900 - accuracy: 0.5363 - val_loss: 1.4557 - val_accuracy: 0.4850
Epoch 13/20
274/274 [==============================] - 2s 8ms/step - loss: 1.2618 - accuracy: 0.5434 - val_loss: 1.4326 - val_accuracy: 0.4989
Epoch 14/20
274/274 [==============================] - 2s 7ms/step - loss: 1.2302 - accuracy: 0.5561 - val_loss: 1.4411 - val_accuracy: 0.4935
Epoch 15/20
274/274 [==============================] - 2s 8ms/step - loss: 1.1912 - accuracy: 0.5664 - val_loss: 1.4684 - val_accuracy: 0.4919
Epoch 16/20
274/274 [==============================] - 2s 8ms/step - loss: 1.1538 - accuracy: 0.5873 - val_loss: 1.4573 - val_accuracy: 0.5014
Epoch 17/20
274/274 [==============================] - 2s 7ms/step - loss: 1.1219 - accuracy: 0.5960 - val_loss: 1.4639 - val_accuracy: 0.4989
Epoch 18/20
274/274 [==============================] - 2s 7ms/step - loss: 1.0836 - accuracy: 0.6053 - val_loss: 1.5006 - val_accuracy: 0.4937
Epoch 19/20
274/274 [==============================] - 2s 7ms/step - loss: 1.0367 - accuracy: 0.6239 - val_loss: 1.4961 - val_accuracy: 0.4996
Epoch 20/20
274/274 [==============================] - 2s 7ms/step - loss: 1.0142 - accuracy: 0.6314 - val_loss: 1.5319 - val_accuracy: 0.4929
```

Figure: Training for 20 epochs

CNNs vs. FCNs

First, we train on a Fully Connected Network

```
[ ]  test_accuracy = model.evaluate(dense_test, testY)[1]
     print("Test Accuracy",np.round((test_accuracy)*100,2))
```

```
313/313 [==============================] - 1s 3ms/step - loss: 1.5011 - accuracy: 0.4960
Test Accuracy 49.6
```

Figure: Test results after the training is done; Test Accuracy = 49.6%

CNNs vs. FCNs

Now, let's train on a Convolutional Neural Network

```
# CNN architecture
model2 = Sequential()
model2.add(Conv2D(32, (3, 3), input_shape=(32, 32, 3),activation='relu'))
model2.add(MaxPool2D(pool_size=(2, 2)))
model2.add(Conv2D(64, (3, 3),activation='relu'))
model2.add(Conv2D(128, (3, 3),activation='relu'))
model2.add(Flatten())
model2.add(Dense(128, activation='relu'))
model2.add(Dense(10, activation='softmax'))

model2.compile(loss='categorical_crossentropy',
               optimizer='adam',
               metrics=['accuracy'])


model2.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 30, 30, 32)        896

 max_pooling2d (MaxPooling2D  (None, 15, 15, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 13, 13, 64)        18496

 conv2d_2 (Conv2D)           (None, 11, 11, 128)       73856

 flatten (Flatten)           (None, 15488)             0

 dense_20 (Dense)            (None, 128)               1982592

 dense_21 (Dense)            (None, 10)                1290

=================================================================
Total params: 2,077,130
Trainable params: 2,077,130
Non-trainable params: 0
_________________________________________________________________
```

Figure: CNN with around 2M parameters (A lot less compared to our 9M parameter FCN)

CNNs vs. FCNs

Now, let's train on a Convolutional Neural Network

```
with tf.device('/device:GPU:0'):
  h2 = model2.fit(x_train, trainY, batch_size=128, epochs=20, verbose=1, validation_split=0.3)
```

```
Epoch 1/20
274/274 [==============================] - 12s 13ms/step - loss: 1.5989 - accuracy: 0.4210 - val_loss: 1.3550 - val_accuracy: 0.5141
Epoch 2/20
274/274 [==============================] - 3s 11ms/step - loss: 1.2401 - accuracy: 0.5602 - val_loss: 1.1999 - val_accuracy: 0.5781
Epoch 3/20
274/274 [==============================] - 3s 11ms/step - loss: 1.0605 - accuracy: 0.6252 - val_loss: 1.1123 - val_accuracy: 0.6082
Epoch 4/20
274/274 [==============================] - 3s 11ms/step - loss: 0.9441 - accuracy: 0.6711 - val_loss: 0.9710 - val_accuracy: 0.6662
Epoch 5/20
274/274 [==============================] - 3s 11ms/step - loss: 0.8403 - accuracy: 0.7043 - val_loss: 0.9403 - val_accuracy: 0.6789
Epoch 6/20
274/274 [==============================] - 3s 11ms/step - loss: 0.7642 - accuracy: 0.7325 - val_loss: 0.9674 - val_accuracy: 0.6662
Epoch 7/20
274/274 [==============================] - 3s 10ms/step - loss: 0.6824 - accuracy: 0.7634 - val_loss: 0.9586 - val_accuracy: 0.6791
Epoch 8/20
274/274 [==============================] - 3s 11ms/step - loss: 0.5991 - accuracy: 0.7905 - val_loss: 1.0435 - val_accuracy: 0.6629
Epoch 9/20
274/274 [==============================] - 4s 14ms/step - loss: 0.5114 - accuracy: 0.8228 - val_loss: 1.0063 - val_accuracy: 0.6769
Epoch 10/20
274/274 [==============================] - 3s 11ms/step - loss: 0.4273 - accuracy: 0.8526 - val_loss: 1.0087 - val_accuracy: 0.6933
Epoch 11/20
274/274 [==============================] - 3s 11ms/step - loss: 0.3506 - accuracy: 0.8793 - val_loss: 1.1035 - val_accuracy: 0.6885
Epoch 12/20
274/274 [==============================] - 3s 11ms/step - loss: 0.2777 - accuracy: 0.9041 - val_loss: 1.1748 - val_accuracy: 0.6844
Epoch 13/20
274/274 [==============================] - 3s 11ms/step - loss: 0.2197 - accuracy: 0.9261 - val_loss: 1.2806 - val_accuracy: 0.6861
Epoch 14/20
274/274 [==============================] - 3s 11ms/step - loss: 0.1615 - accuracy: 0.9469 - val_loss: 1.5005 - val_accuracy: 0.6733
Epoch 15/20
274/274 [==============================] - 3s 11ms/step - loss: 0.1364 - accuracy: 0.9544 - val_loss: 1.4768 - val_accuracy: 0.6916
Epoch 16/20
274/274 [==============================] - 3s 11ms/step - loss: 0.0983 - accuracy: 0.9676 - val_loss: 1.7132 - val_accuracy: 0.6821
Epoch 17/20
274/274 [==============================] - 3s 11ms/step - loss: 0.0906 - accuracy: 0.9700 - val_loss: 1.8566 - val_accuracy: 0.6713
Epoch 18/20
274/274 [==============================] - 3s 12ms/step - loss: 0.0856 - accuracy: 0.9707 - val_loss: 1.8752 - val_accuracy: 0.6745
Epoch 19/20
274/274 [==============================] - 3s 11ms/step - loss: 0.0807 - accuracy: 0.9731 - val_loss: 1.9362 - val_accuracy: 0.6767
Epoch 20/20
274/274 [==============================] - 3s 11ms/step - loss: 0.0606 - accuracy: 0.9802 - val_loss: 2.1431 - val_accuracy: 0.6763
```

Figure: Training for 20 epochs

CNNs vs. FCNs

Now, let's train on a Convolutional Neural Network

```
test2_accuracy = model2.evaluate(x_test, testY)[1]
print("Test Accuracy",np.round((test2_accuracy)*100,2))
```

```
313/313 [==============================] - 1s 4ms/step - loss: 2.1878 - accuracy: 0.6647
Test Accuracy 66.47
```

Figure: Test results after the training is done; Test Accuracy = 66.5%

CNNs vs. FCNs

- As you can see, CNNs have a great effect on both results and computation efficiency while working with visual imagery.

Model Architecture	# Params	Test Accuracy
CNN	**2M**	**66.5%**
FCN	9M	49.6%

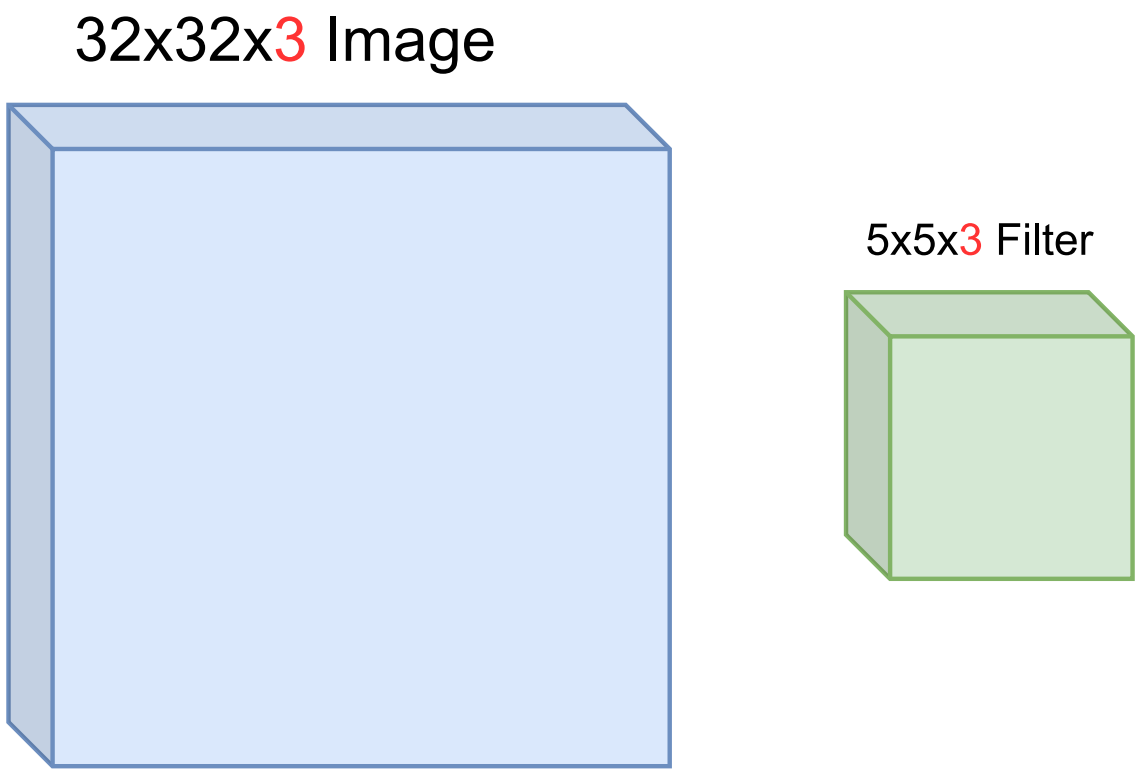
CNNs vs. FCNs

Now that we understand the importance of Deep Learning and Convolutional Neural Networks, let's learn the details.

CNNs

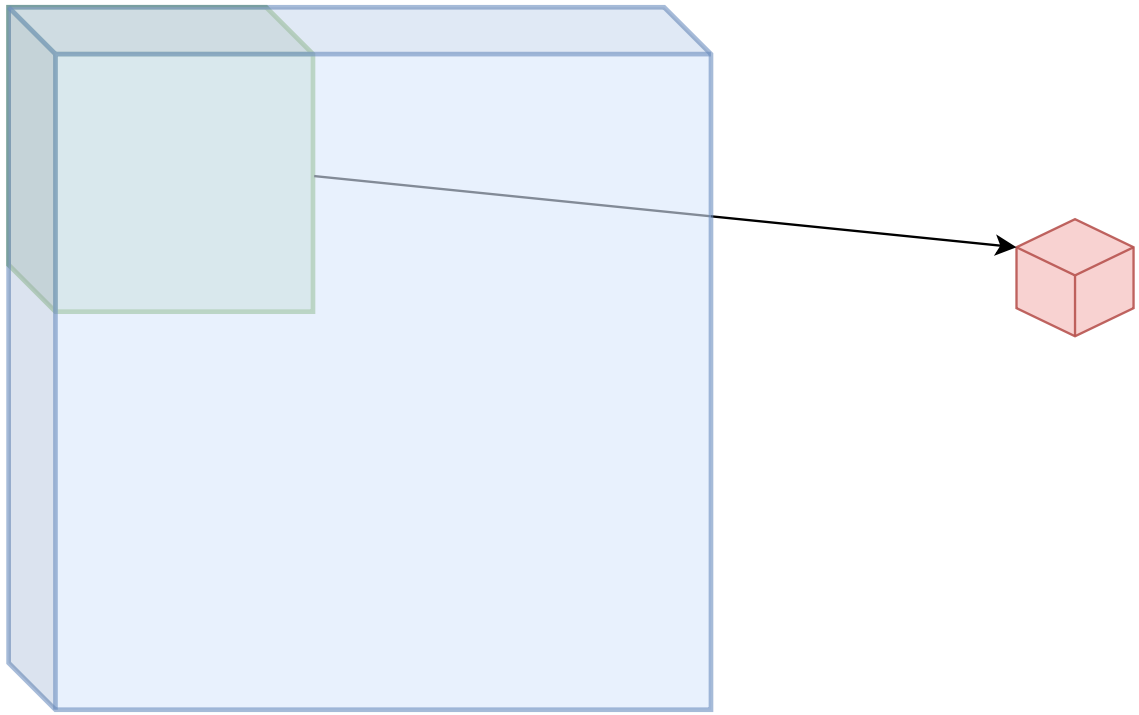
Convolutional Layer

- Filters always extend the full depth of the input. (#Input channels == #Filter Channels)



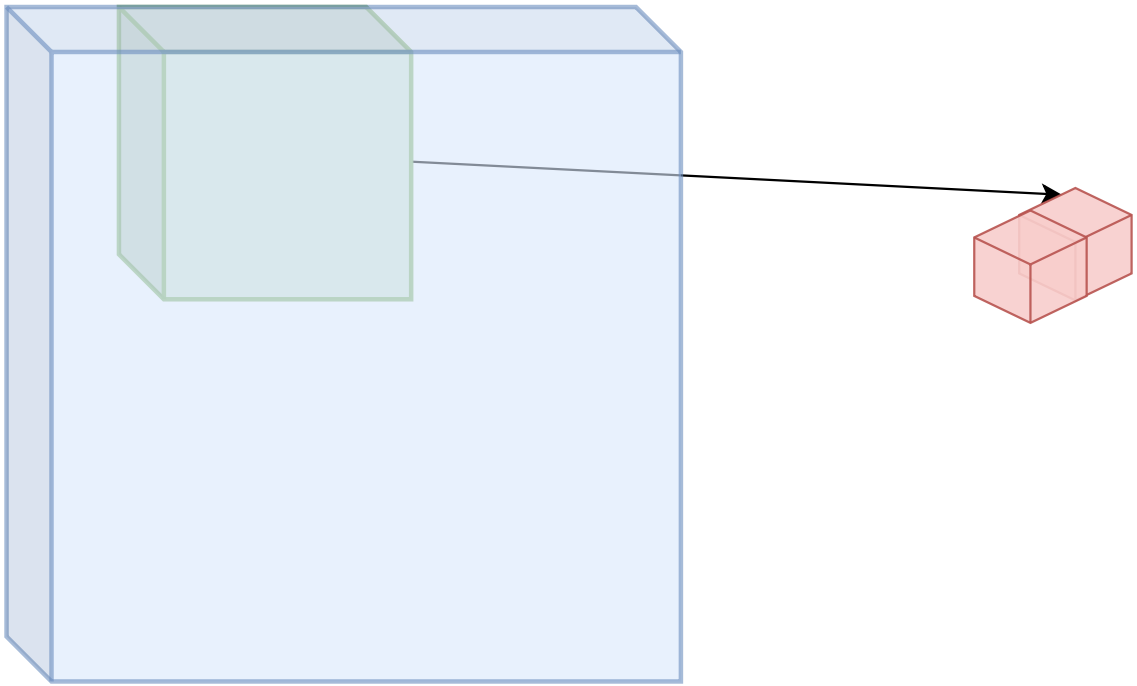
What is Convolution and how does it work?

- We apply the same filter on different regions of the input



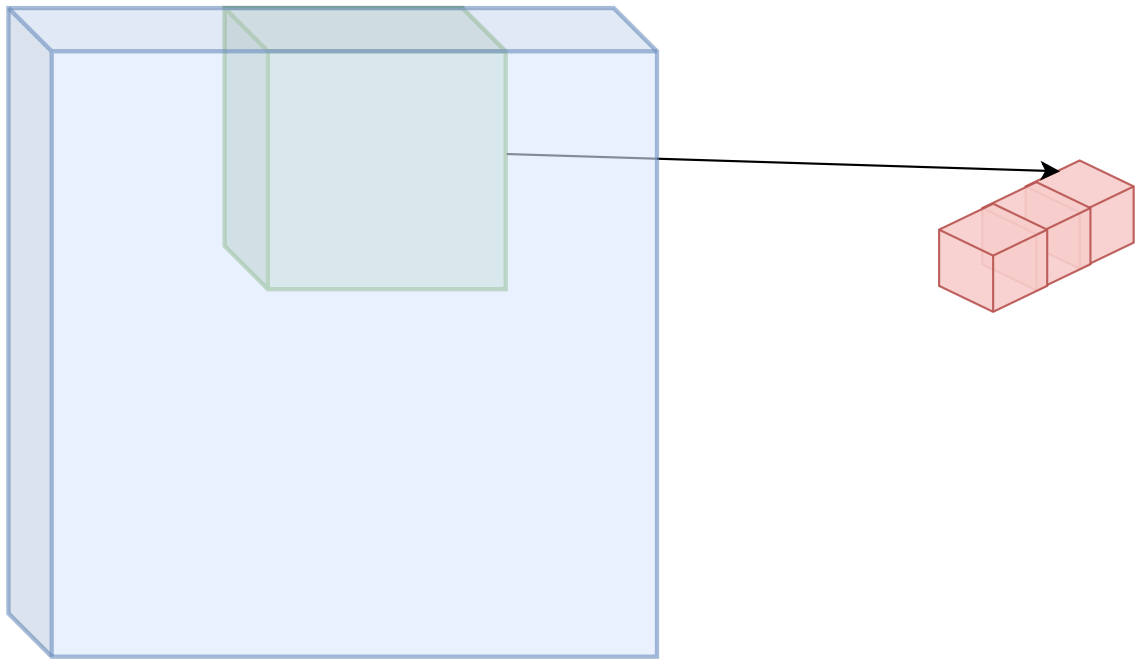
What is Convolution and how does it work?

- We apply the same filter on different regions of the input
- Convolutional filters learn to make decisions based on local spatial input, which is an important attribute when working with images.



What is Convolution and how does it work?

- We apply the same filter on different regions of the input
- Convolutional filters learn to make decisions based on local spatial input, which is an important attribute when working with images.
- Uses a lot fewer parameters compared to Fully Connected Layers.



What is Convolution and how does it work?

Consider this, the filter we are going to use:

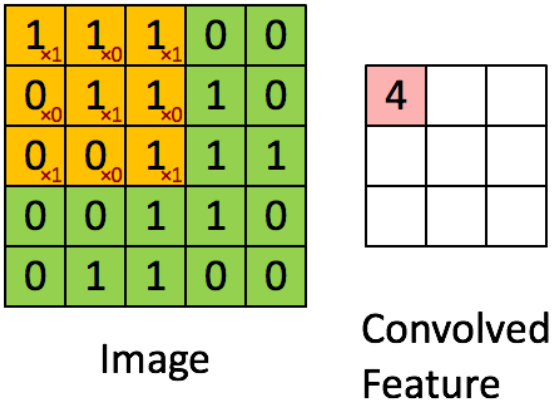
1	0	1
0	1	0
1	0	1

What is Convolution and how does it work?

This is how we calculate the convolutional layer's output:

$$ConvolvedFeature(i,j) = (I * K)(i,j) = \sum_{a=0}^{k_h-1} \sum_{b=0}^{k_w-1} I(i+a, j+b)K(a,b)$$

I: Input Image
K: Our Kernel
k_h and k_w: The height and width of the Kernel



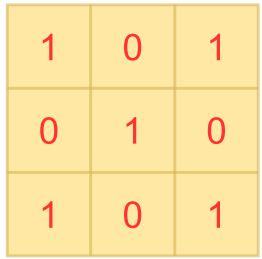


Figure: Convolving Kernel

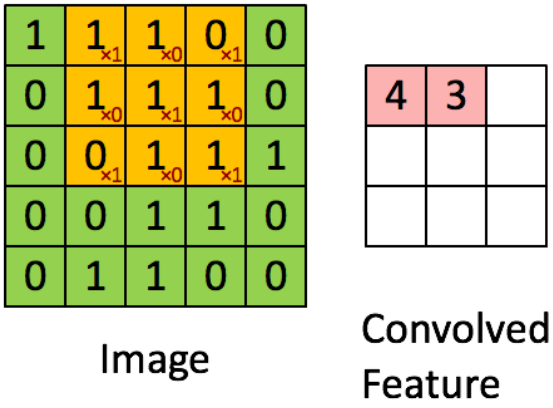
Figure: Convoluting a 5x5x1 image with a 3x3x1 kernel to get a 3x3x1 convolved feature, Source

What is Convolution and how does it work?

This is how we calculate the convolutional layer's output:

$$ConvolvedFeature(i,j) = (I * K)(i,j) = \sum_{a=0}^{k_h-1} \sum_{b=0}^{k_w-1} I(i+a, j+b)K(a,b)$$

I: Input Image
K: Our Kernel
k_h and k_w: The height and width of the Kernel



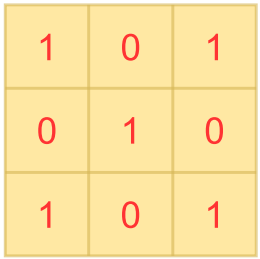


Figure: Convolving Kernel

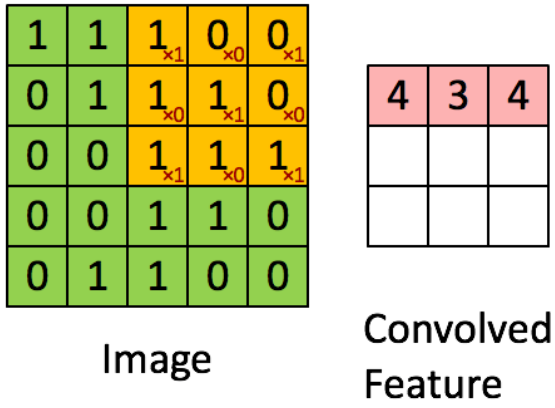
Figure: Convoluting a 5x5x1 image with a 3x3x1 kernel to get a 3x3x1 convolved feature, Source

What is Convolution and how does it work?

This is how we calculate the convolutional layer's output:

$$ConvolvedFeature(i,j) = (I * K)(i,j) = \sum_{a=0}^{k_h-1} \sum_{b=0}^{k_w-1} I(i+a, j+b)K(a,b)$$

I: Input Image
K: Our Kernel
k_h and k_w: The height and width of the Kernel



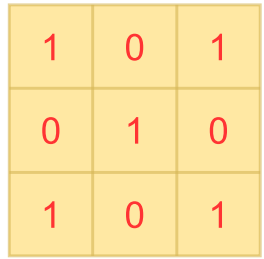


Figure: Convolving Kernel

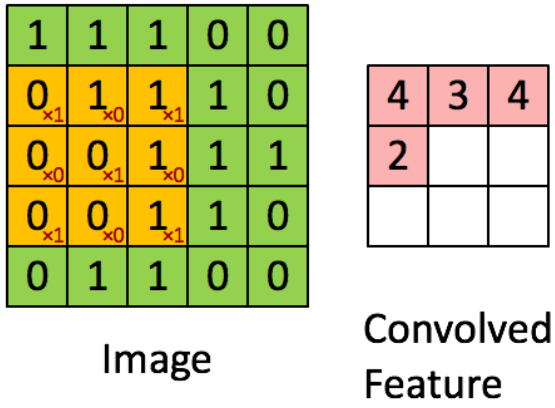
Figure: Convoluting a 5x5x1 image with a 3x3x1 kernel to get a 3x3x1 convolved feature, Source

What is Convolution and how does it work?

This is how we calculate the convolutional layer's output:

$$ConvolvedFeature(i,j) = (I * K)(i,j) = \sum_{a=0}^{k_h-1} \sum_{b=0}^{k_w-1} I(i+a, j+b)K(a,b)$$

I: Input Image
K: Our Kernel
k_h and k_w: The height and width of the Kernel



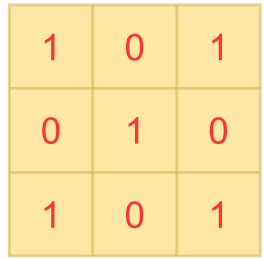


Figure: Convolving Kernel

Figure: Convoluting a 5x5x1 image with a 3x3x1 kernel to get a 3x3x1 convolved feature, Source

What is Convolution and how does it work?

This is how we calculate the convolutional layer's output:

$$ConvolvedFeature(i,j) = (I * K)(i,j) = \sum_{a=0}^{k_h-1} \sum_{b=0}^{k_w-1} I(i+a, j+b)K(a,b)$$

I: Input Image
K: Our Kernel
k_h and k_w: The height and width of the Kernel

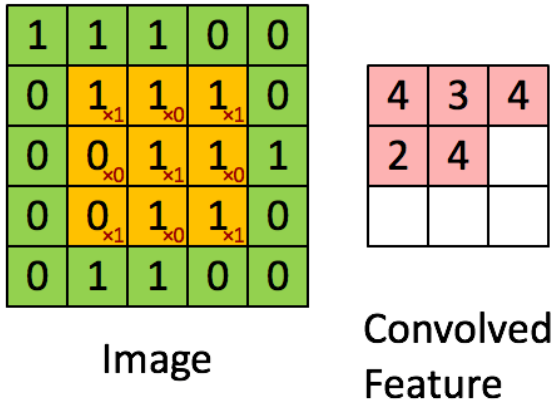


Figure: Convoluting a 5x5x1 image with a 3x3x1 kernel to get a 3x3x1 convolved feature, Source

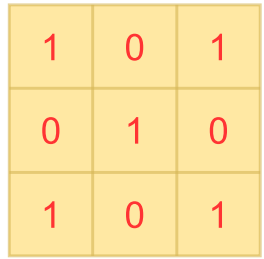


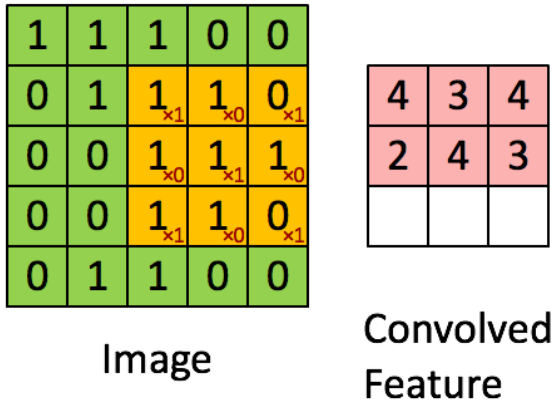
Figure: Convolving Kernel

What is Convolution and how does it work?

This is how we calculate the convolutional layer's output:

$$ConvolvedFeature(i,j) = (I * K)(i,j) = \sum_{a=0}^{k_h-1} \sum_{b=0}^{k_w-1} I(i+a, j+b)K(a,b)$$

I: Input Image
K: Our Kernel
k_h and k_w: The height and width of the Kernel



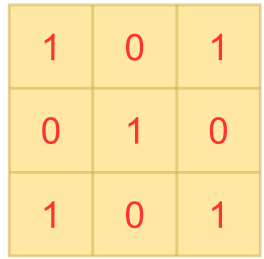


Figure: Convolving Kernel

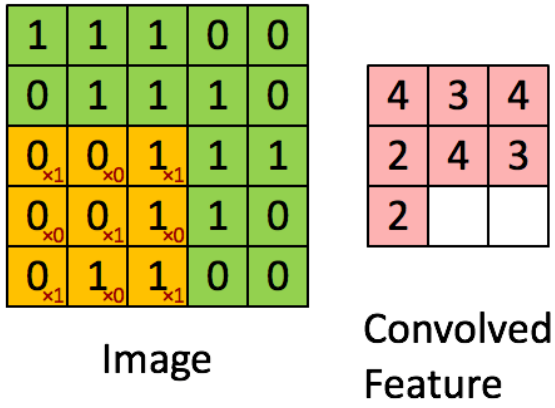
Figure: Convoluting a 5x5x1 image with a 3x3x1 kernel to get a 3x3x1 convolved feature, Source

What is Convolution and how does it work?

This is how we calculate the convolutional layer's output:

$$ConvolvedFeature(i,j) = (I * K)(i,j) = \sum_{a=0}^{k_h-1} \sum_{b=0}^{k_w-1} I(i+a, j+b)K(a,b)$$

I: Input Image
K: Our Kernel
k_h and k_w: The height and width of the Kernel



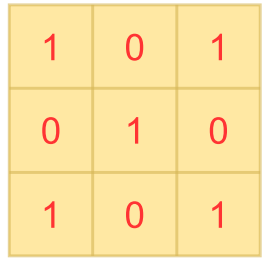


Figure: Convolving Kernel

Figure: Convoluting a 5x5x1 image with a 3x3x1 kernel to get a 3x3x1 convolved feature, Source

What is Convolution and how does it work?

This is how we calculate the convolutional layer's output:

$$ConvolvedFeature(i,j) = (I * K)(i,j) = \sum_{a=0}^{k_h-1} \sum_{b=0}^{k_w-1} I(i+a, j+b)K(a,b)$$

I: Input Image
K: Our Kernel
k_h and k_w: The height and width of the Kernel

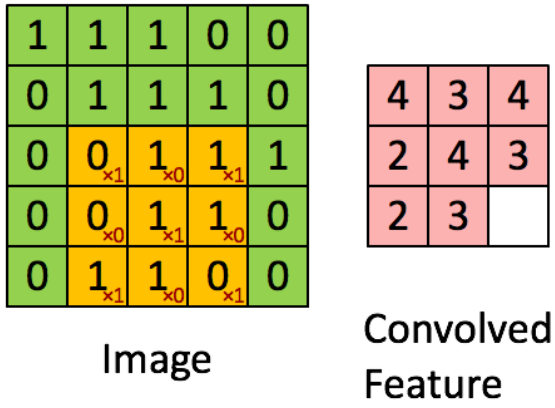


Figure: Convoluting a 5x5x1 image with a 3x3x1 kernel to get a 3x3x1 convolved feature, Source

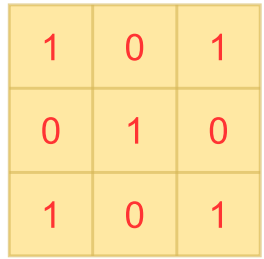


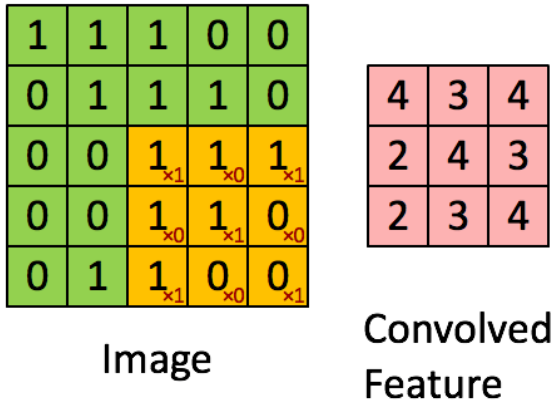
Figure: Convolving Kernel

What is Convolution and how does it work?

This is how we calculate the convolutional layer's output:

$$ConvolvedFeature(i,j) = (I * K)(i,j) = \sum_{a=0}^{k_h-1} \sum_{b=0}^{k_w-1} I(i+a, j+b)K(a,b)$$

I: Input Image
K: Our Kernel
k_h and k_w: The height and width of the Kernel



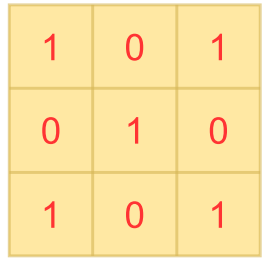


Figure: Convolving Kernel

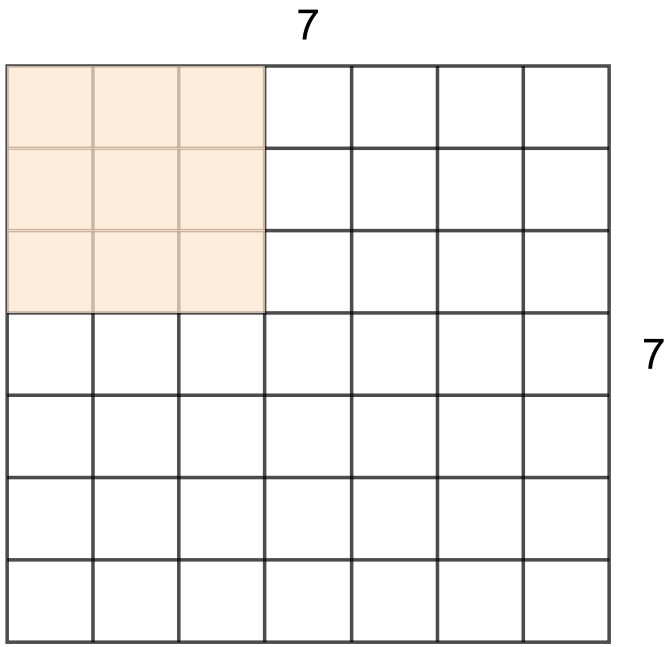
Figure: Convoluting a 5x5x1 image with a 3x3x1 kernel to get a 3x3x1 convolved feature, Source

What is a Stride

The amount of movement between applications of the filter to the input image is referred to as the stride, and it is almost always symmetrical in height and width dimensions.

Closer look

- 7x7 input with 3x3 filter

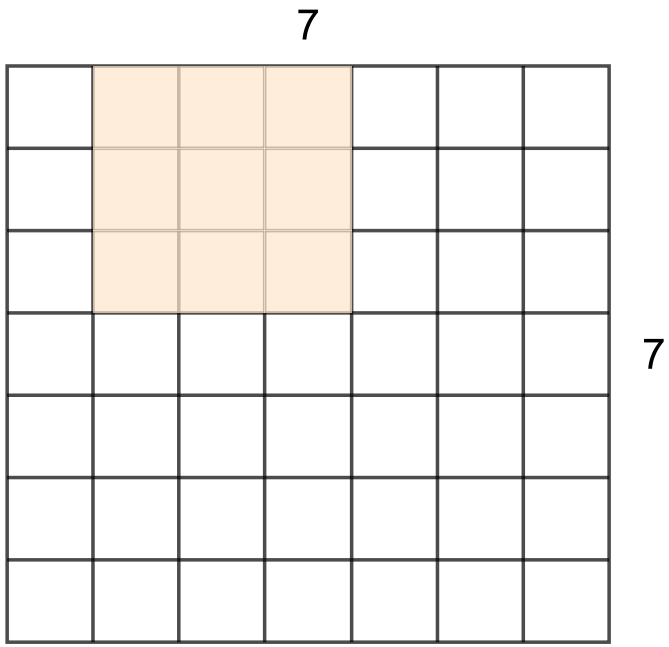


What is a Stride

The amount of movement between applications of the filter to the input image is referred to as the stride, and it is almost always symmetrical in height and width dimensions.

Closer look

- 7x7 input with 3x3 filter

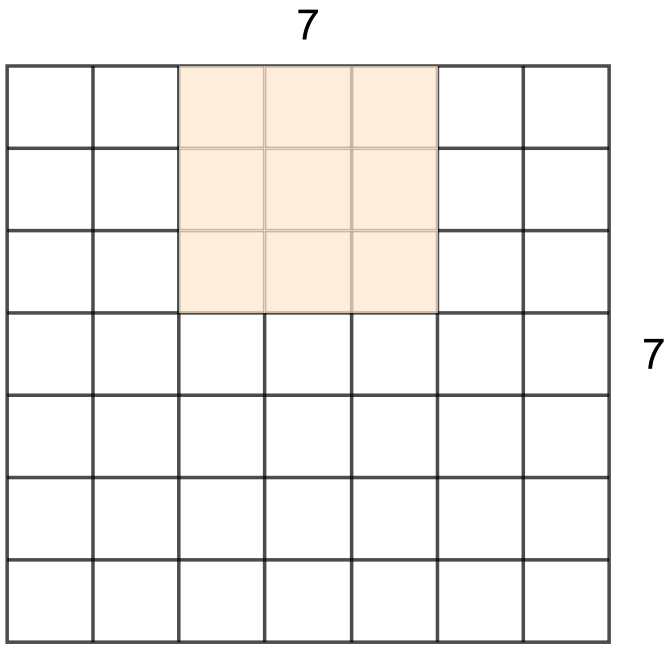


What is a Stride

The amount of movement between applications of the filter to the input image is referred to as the stride, and it is almost always symmetrical in height and width dimensions.

Closer look

- 7x7 input with 3x3 filter

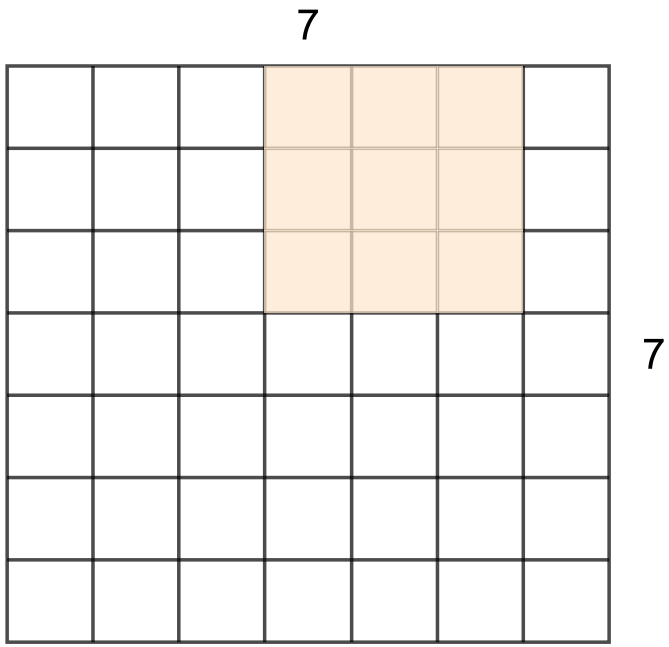


What is a Stride

The amount of movement between applications of the filter to the input image is referred to as the stride, and it is almost always symmetrical in height and width dimensions.

Closer look

- 7x7 input with 3x3 filter

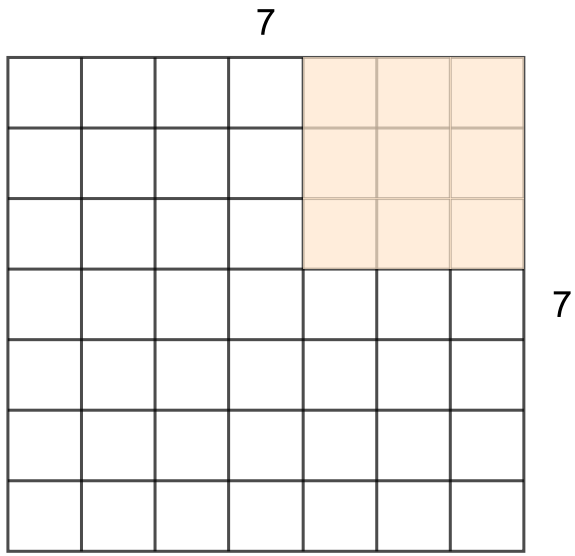


What is a Stride

The amount of movement between applications of the filter to the input image is referred to as the stride, and it is almost always symmetrical in height and width dimensions.

Closer look

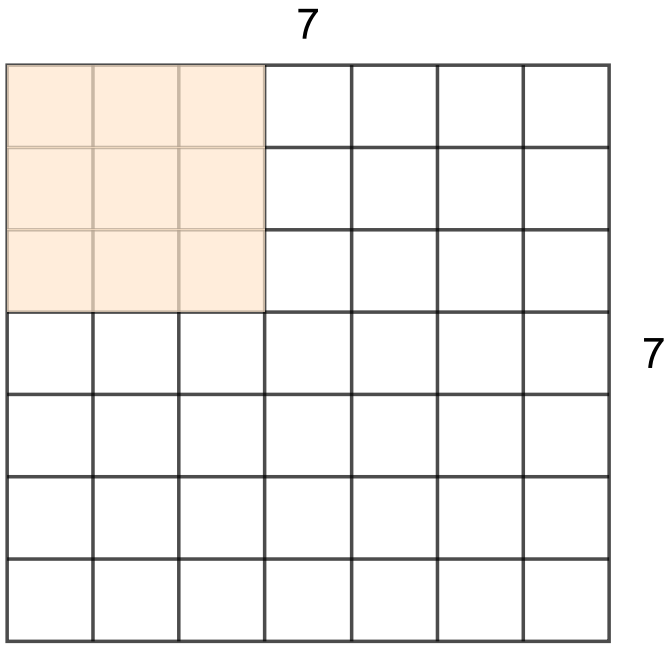
- 7x7 input with 3x3 filter
- This was a Stride 1 filter
- => Outputs 5x5



Strides

Now let's use Stride 2

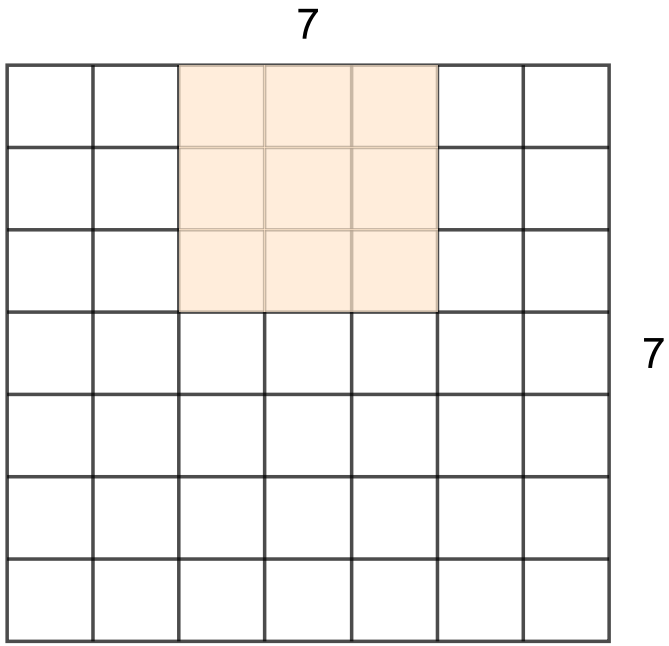
- 7x7 input with 3x3 filter



Strides

Now let's use Stride 2

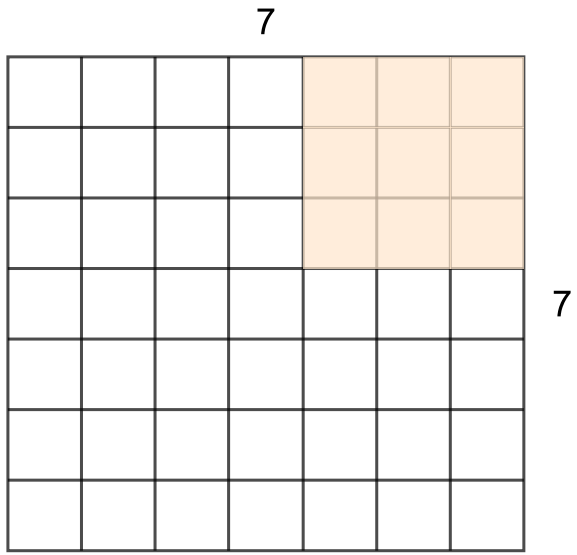
- 7x7 input with 3x3 filter



Strides

Now let's use Stride 2

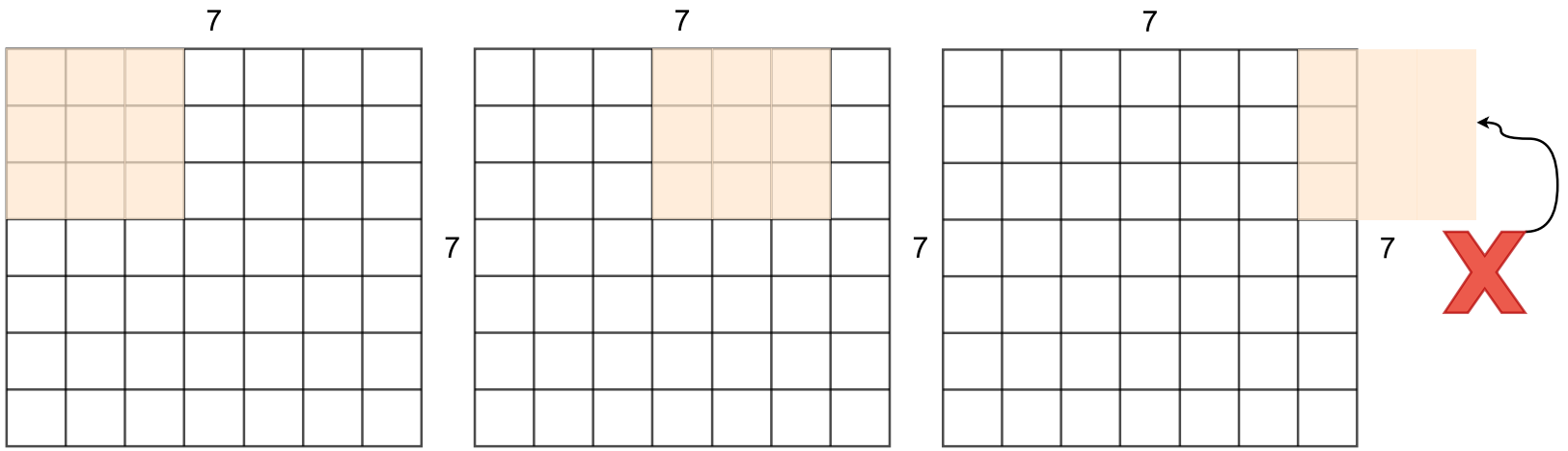
- 7x7 input with 3x3 filter
- This was a Stride 2 filter
- => Outputs 3x3



Strides

Stride 3?

- 7x7 input with 3x3 filter

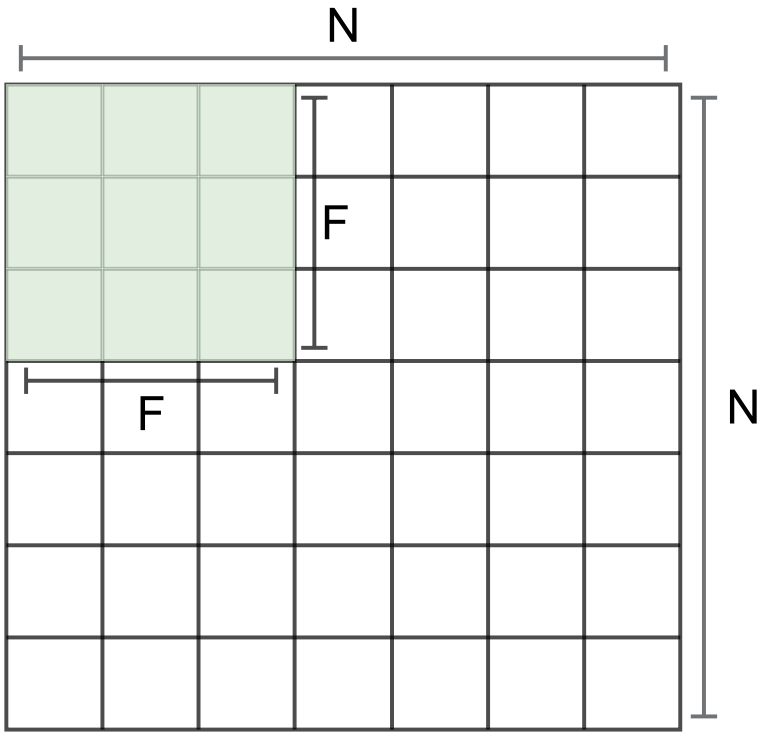


Strides

So 7x7 input with 3x3 filter and stride 3 doens't work!

Let't do the calculations:

$$OutputSize = (N - F)/Stride + 1$$

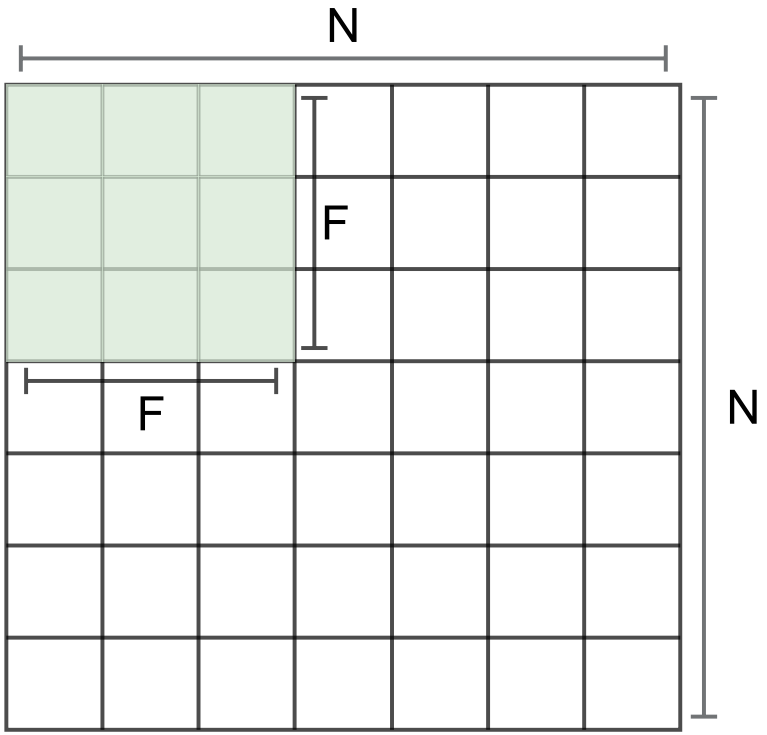


Strides

$$OutputSize = (N - F)/Stride + 1$$

N = 7, F = 3 =>

- Stride 1 ==> $(7 - 3)/1 + 1 = 5$
- Stride 2 ==> $(7 - 3)/2 + 1 = 3$
- Stride 3 ==> $(7 - 3)/3 + 1 = 2.33$:))



Strides

Do you see any problems?

Strides

- 1. Borders don't get enough attention.

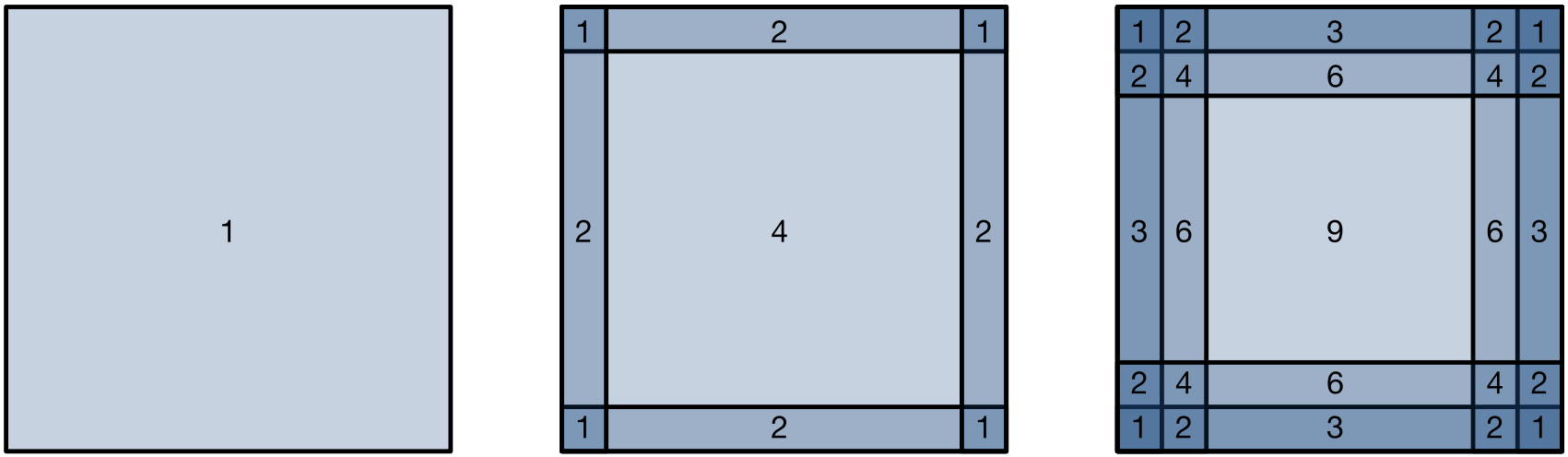


Figure: [3], Pixel utilization for convolutions of size 1×1, 2×2, and 3×3 respectively.

Strides

- 1. Borders don't get enough attention.
- 2. Outputs shrink!

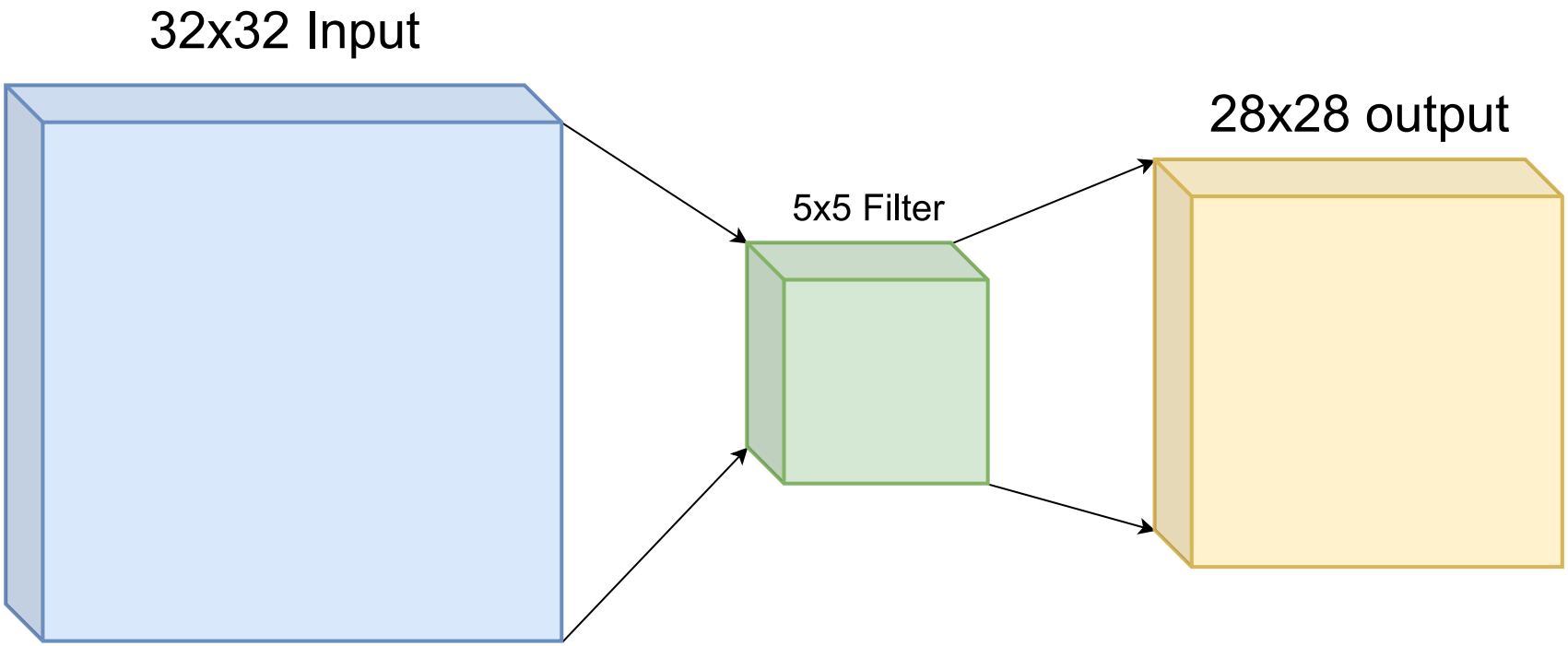


Figure: 32x32 input shrinks to 28x28 output. (information loss)

Strides

What is the solution?

Padding

Enters Padding

Padding

- We can use Padding to preserve the dimensionality
- It ensures that all pixels are used equally frequently

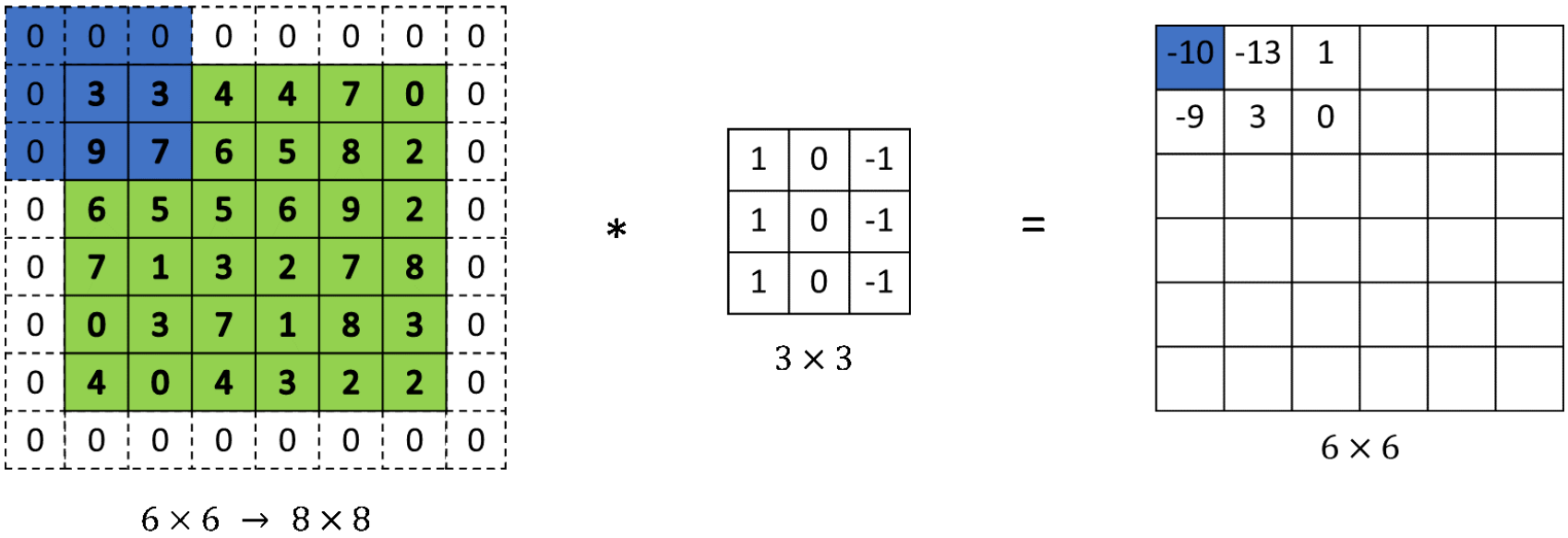


Figure: DataHacker.rs: CNN Padding - Applying padding of 1 before convolving with 3×3 filter

Padding

- It is common to use $P = (F - 1)/2$ with stride 1 to preserve the input size.

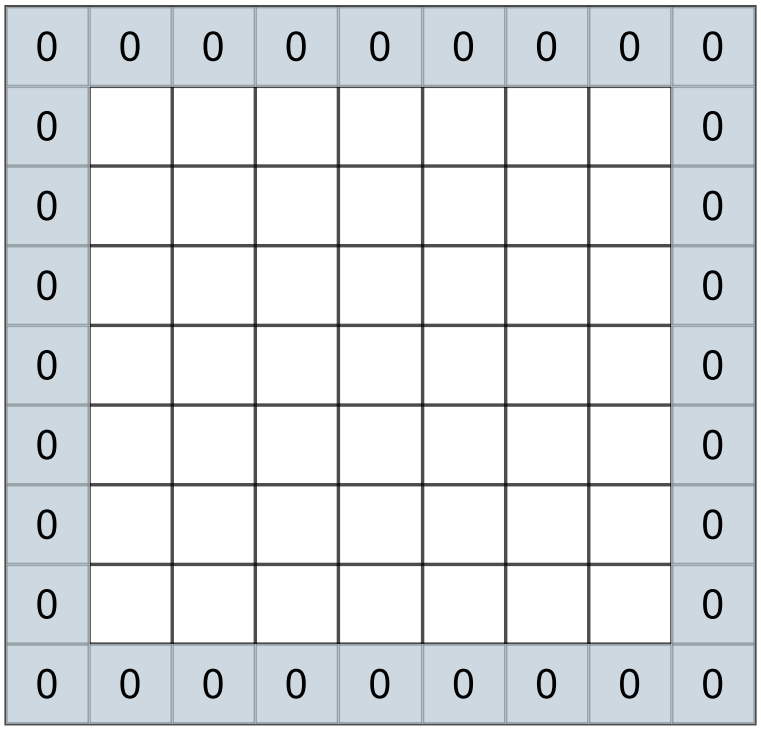


Figure: Applying zero-padding to a 7x7 input with padding 1

Padding

- It is common to use $P = (F - 1)/2$ with stride 1 to preserve the input size.
- $OutputSize = (N + 2P - F)/Stride + 1$

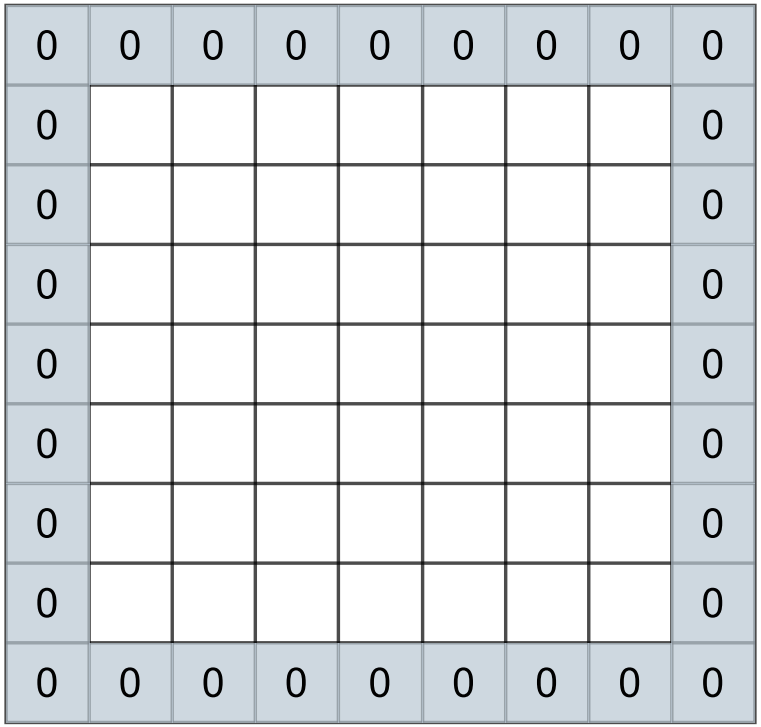


Figure: Applying zero-padding to a 7x7 input with padding 1

Padding

- It is common to use $P = (F-1)/2$ with stride 1 to preserve the input size.
- $OutputSize = (N + 2P - F)/Stride + 1$

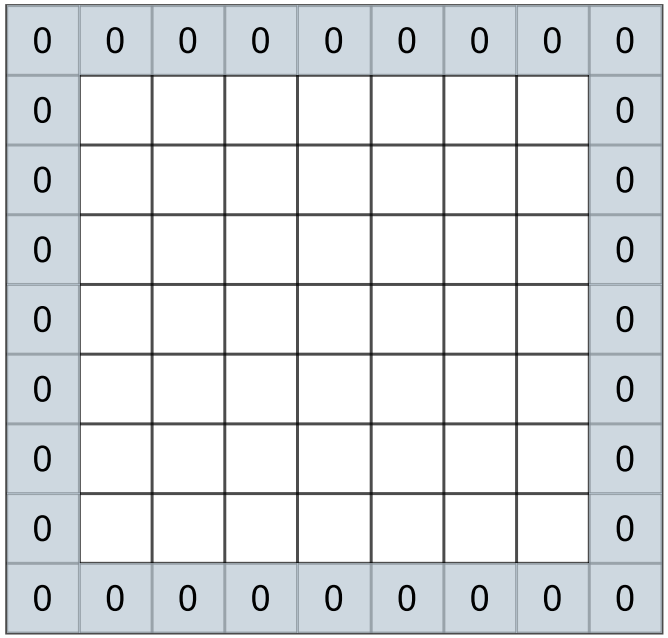


Figure: Applying zero-padding to a 7x7 input with padding 1

7x7 input, stride 1, P to preserve the dimentions?

- $F = 3 - - > P = 1$
- $F = 5 - - > P = 2$
- $F = 7 - - > P = 3$

Pooling

- Pooling is performed in neural networks to reduce variance and computation complexity

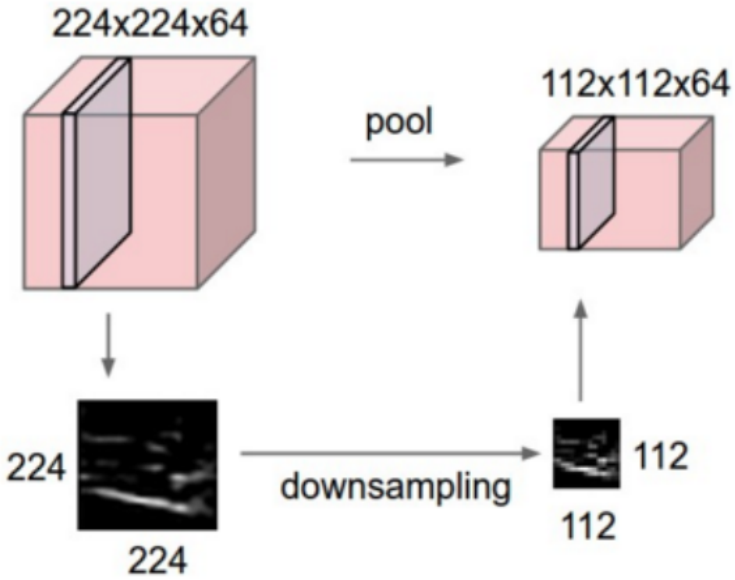


Figure: Pooling Layer

Pooling: How It Works

- Pooling layers aggregate the inputs using an aggregation function such as the max or mean.

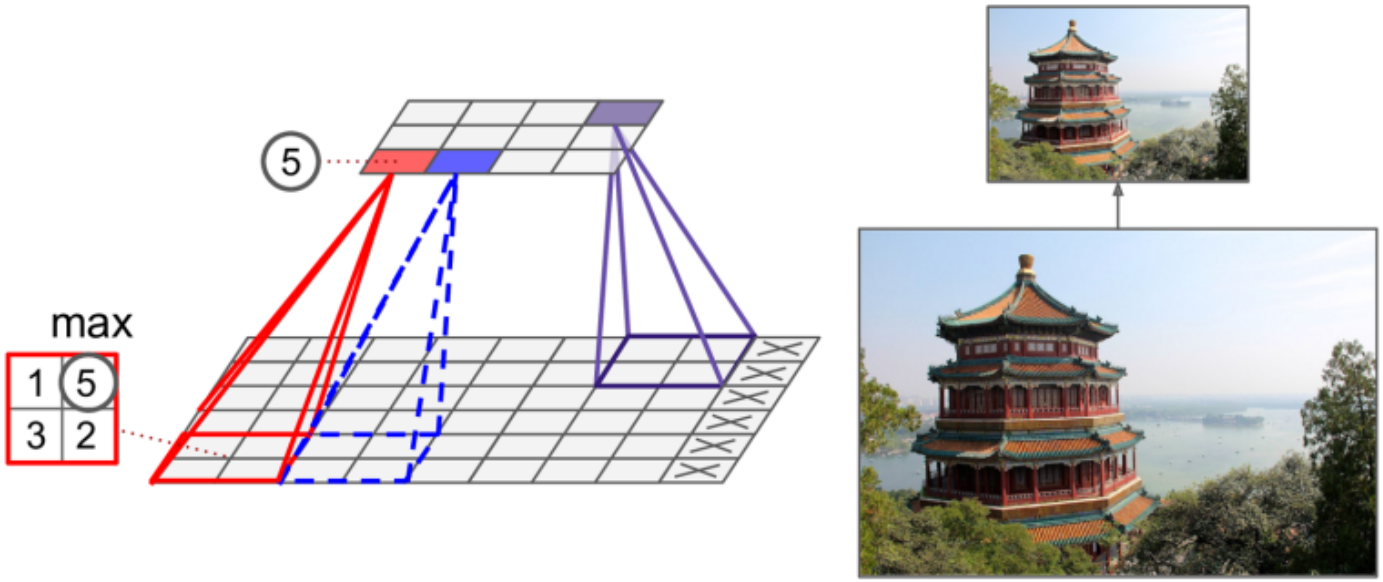


Figure: Max pooling layer (2 × 2 pooling kernel, stride 2, no padding)

Pooling: Benefits

- A Pooling layer's goal is to sub-sample the input in order to reduce:
  - The computational load
  - The memory usage
  - The number of parameters
  - Limiting the risk of overfitting

Pooling: Types

- Max Pooling: gives better results when the background is white and the object is dark
- Min Pooling: gives better results when the background is dark and the object is white
- Average Pooling: smoothes the picture

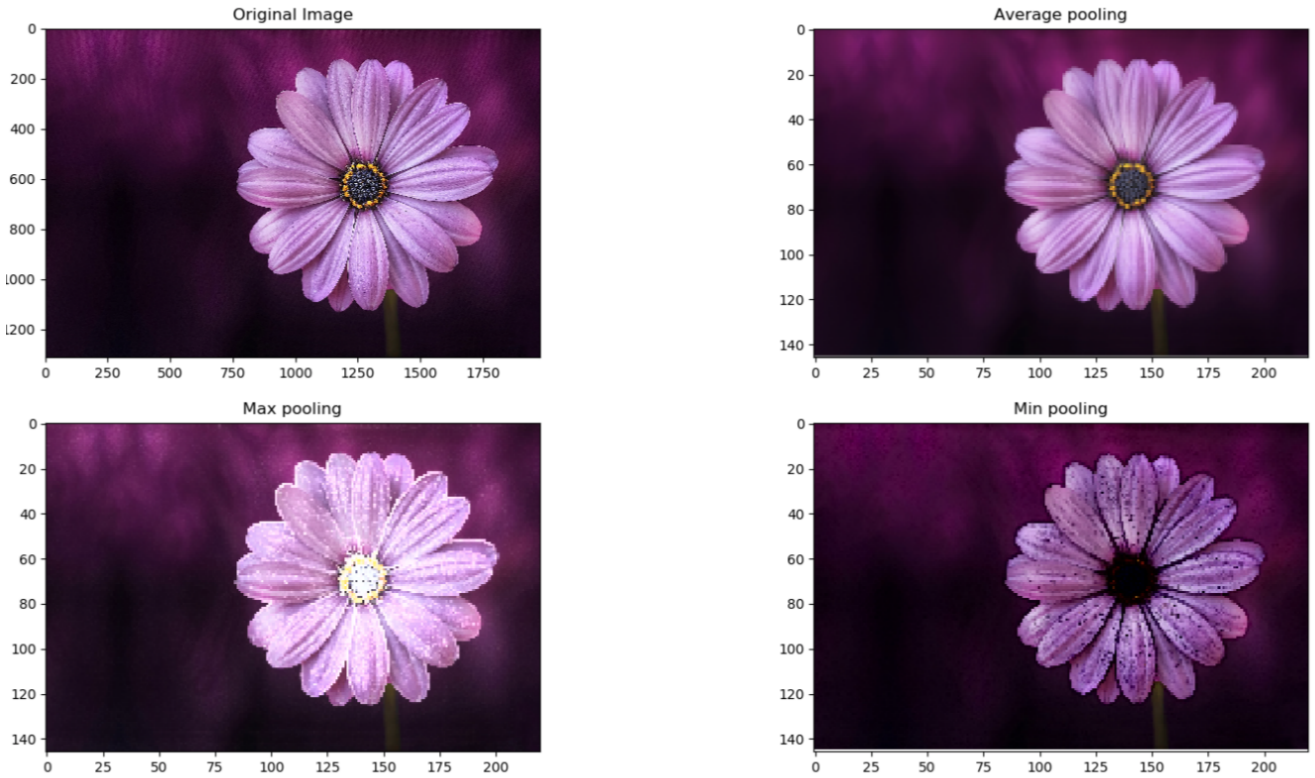


Figure: Effects of different pooling layers

Pooling: Max Pooling

- Most common type of pooling layer
- Invariance to small translations

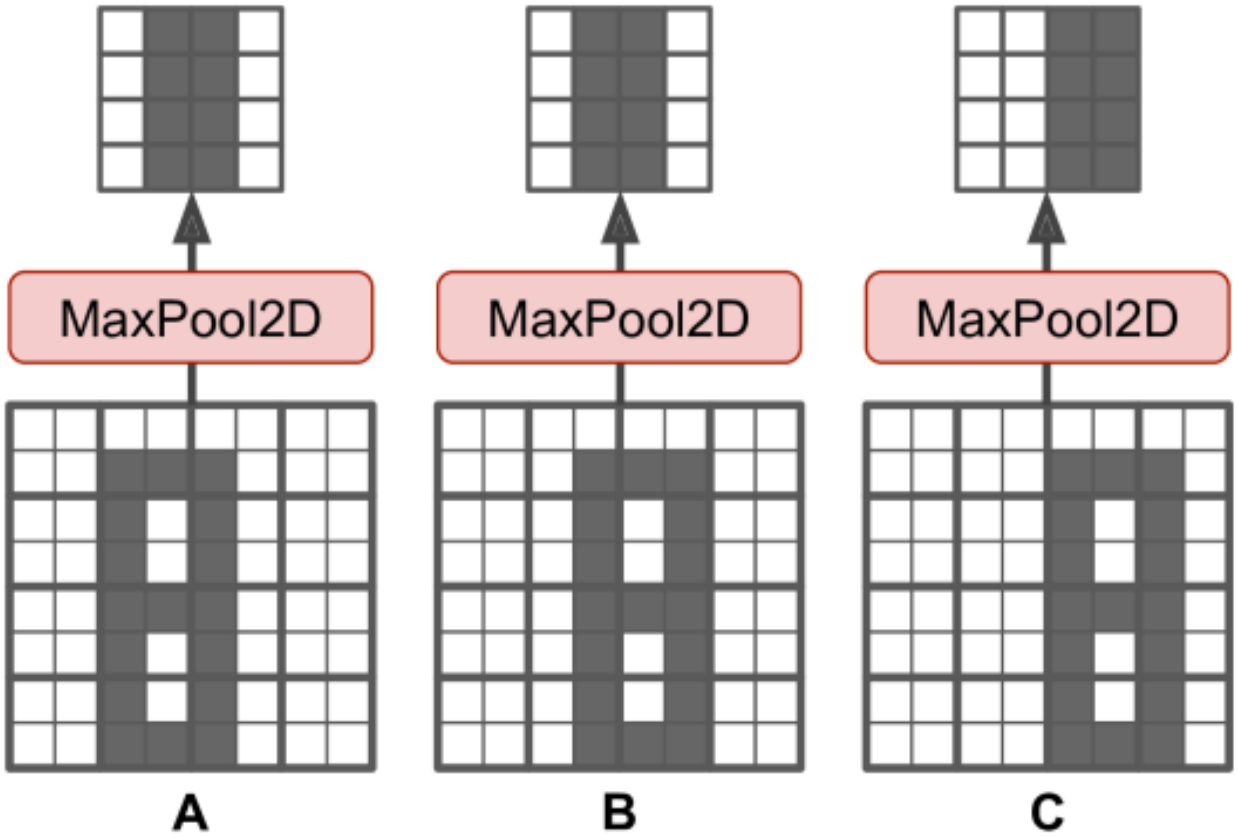


Figure: Invariance to small translations

Dilation

- It expands the kernel (input) by inserting spaces between its consecutive elements

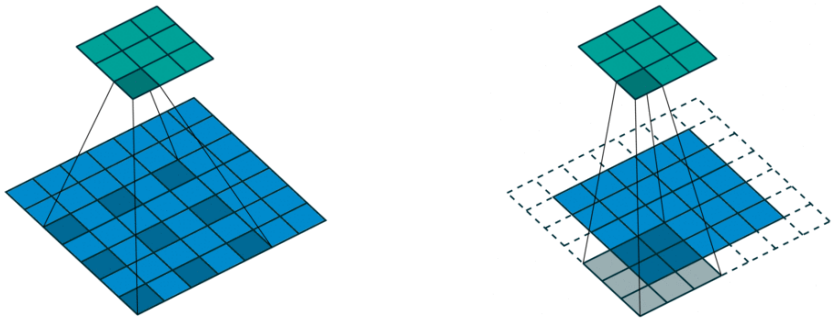


Figure: Dilated Convolution ($l = 2$) vs. Standard Convolution ($l = 1$)

Dilation: Benefits

- Since dilated convolutions support exponential expansion in the context of the receptive field, there is no loss of resolution.
- Dilated convolutions use 'l' as the parameter for the dilation rate. As we increase the value of 'l' it allows one to have a larger receptive field which is really helpful as we are able to view more data points thus saving computation and memory costs

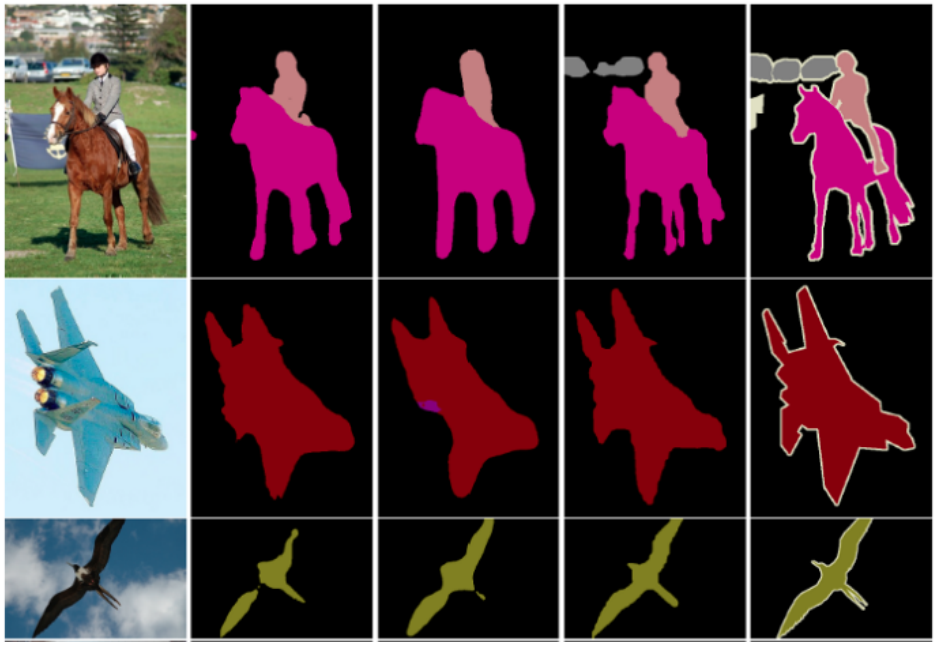


Figure: Image - FCN-8s - DeepLab - DilatedNet - Ground Truth

Channels

So far, we have talked about 2D inputs, but although images are 2D, we need to represent them as 3D matrices to show their colors:

- Pixel values range from 0 to 255
- We cannot represent all the colors in a picture with only one channel of numbers from 0 to 255

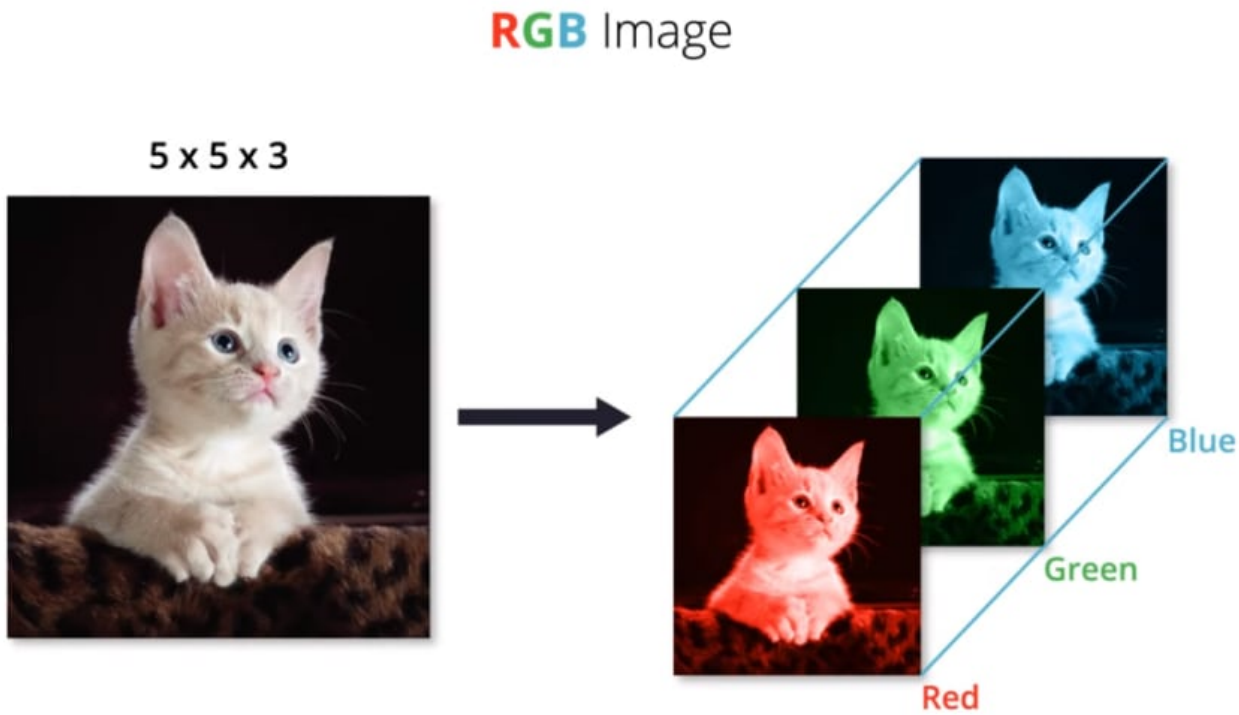


Figure: RGB Image, Source

Channels

So far, we have talked about 2D inputs, but although images are 2D, we need to represent them as 3D matrices to show their colors:

- So we represent them in 3 channels
- They consist of 3 channels: Red, Green and Blue (RGB images)
- Shape: (height x width x 3)

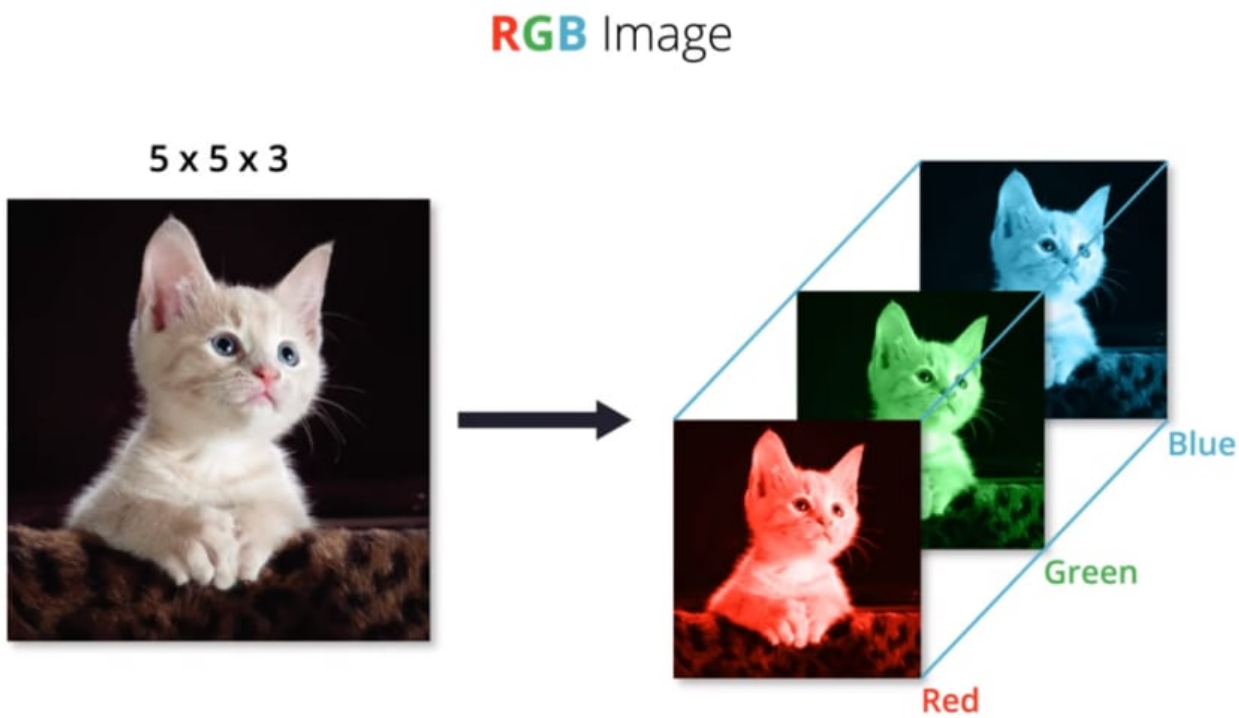


Figure: RGB Image, Source

Channels

So far, we have talked about 2D inputs, but although images are 2D, we need to represent them as 3D matrices to show their colors:

- They consist of 3 channels: Red, Green and Blue (RGB images)
- Shape: (height x width x 3)

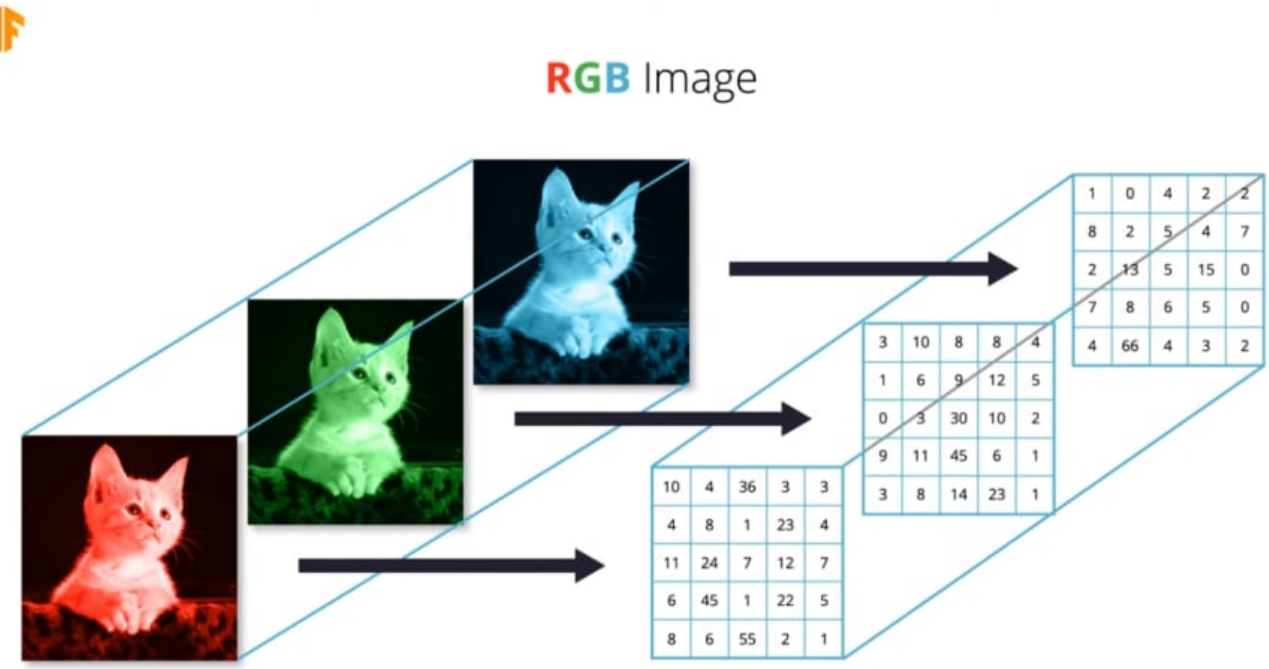


Figure: RGB Image, Source

Channels

So far, we have talked about 2D inputs, but although images are 2D, we need to represent them as 3D matrices to show their colors:

- They consist of 3 channels: Red, Green and Blue (RGB images)
- Shape: (height x width x 3)

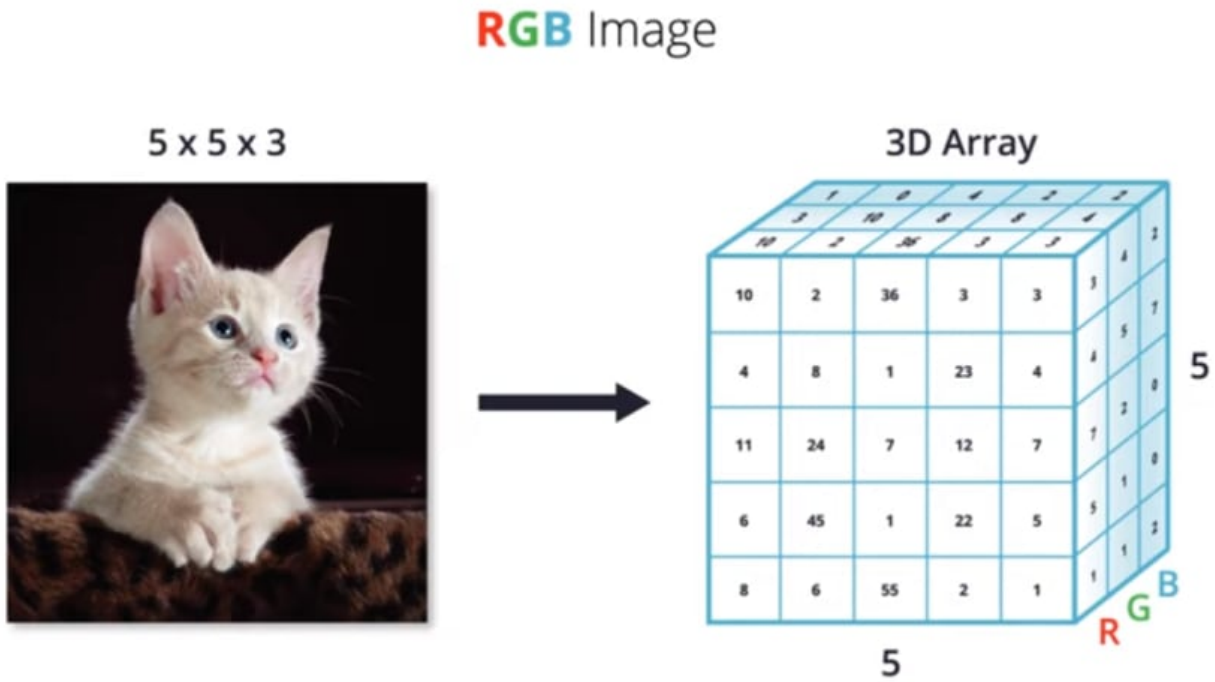


Figure: RGB Image, Source

Channels

- As said before: ”Filters always extend the full depth of the input. (#Input channels == #Filter Channels)”

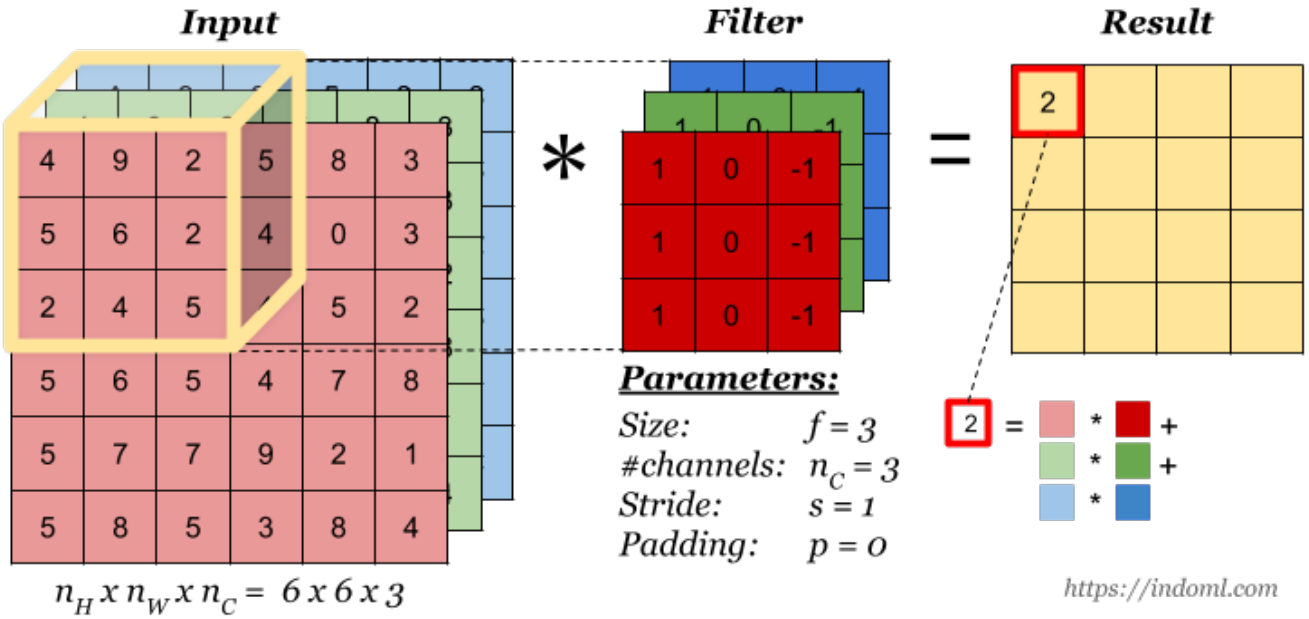


Figure: The total number of multiplications to calculate the result is (4 x 4) x (3 x 3 x 3) = 432, Source

Channels

- As said before: ”Filters always extend the full depth of the input volume. (#Input channels == #Filter Channels)”

- Each filter results in one output channel. We can apply multiple different filters to obtain multiple output channels. Each channel can learn something distinct.

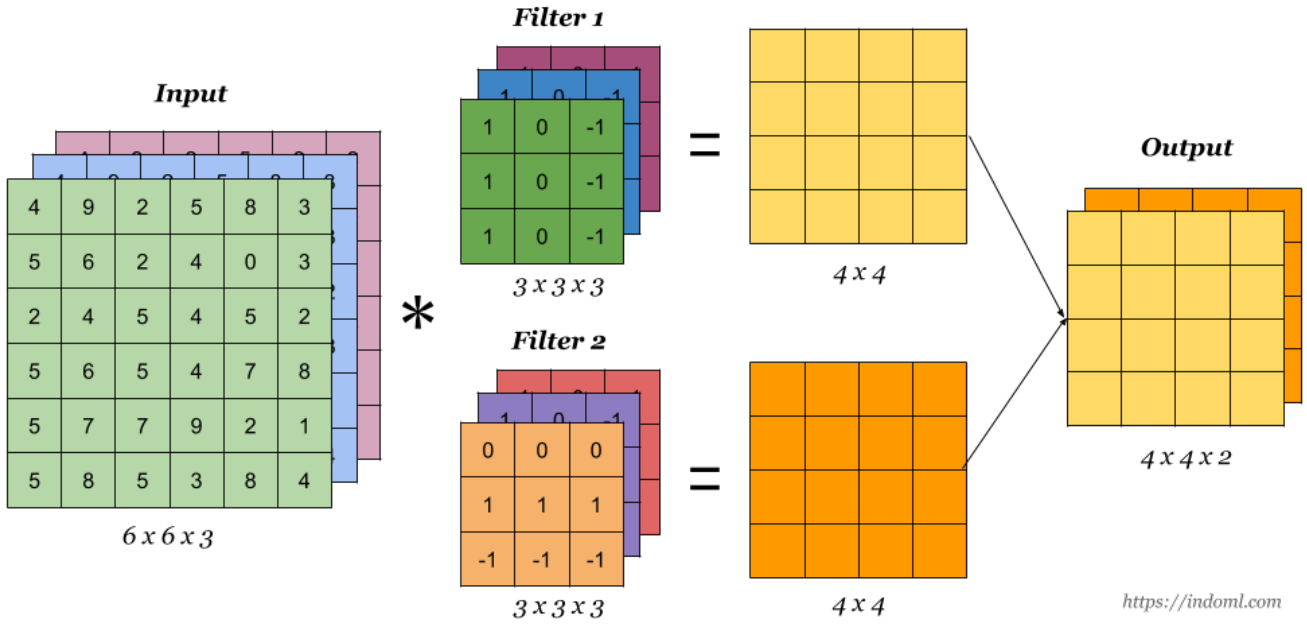


Figure: The total number of multiplications to calculate the result is (4 x 4 x 2) x (3 x 3 x 3) = 864, Source

Channels

Let's have a closer look to how the calculations with more than one filter work:

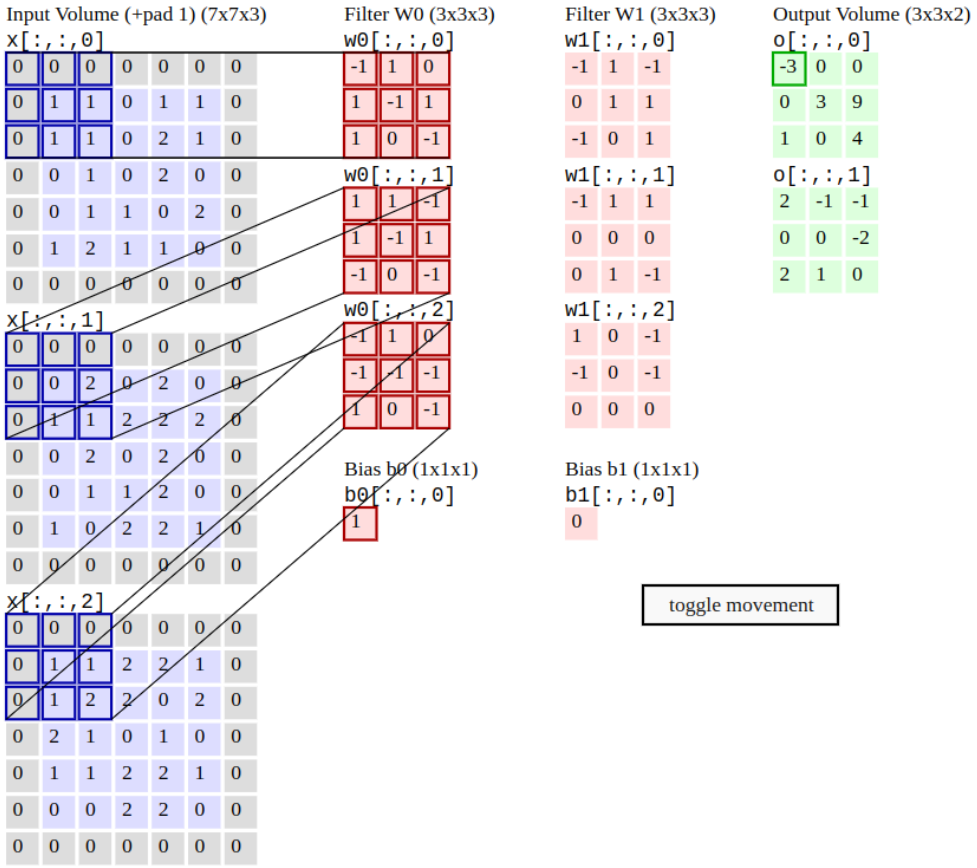


Figure: Great Visualization by CS231n, Source

Channels

Let's have a closer look to how the calculations with more than one filter work:

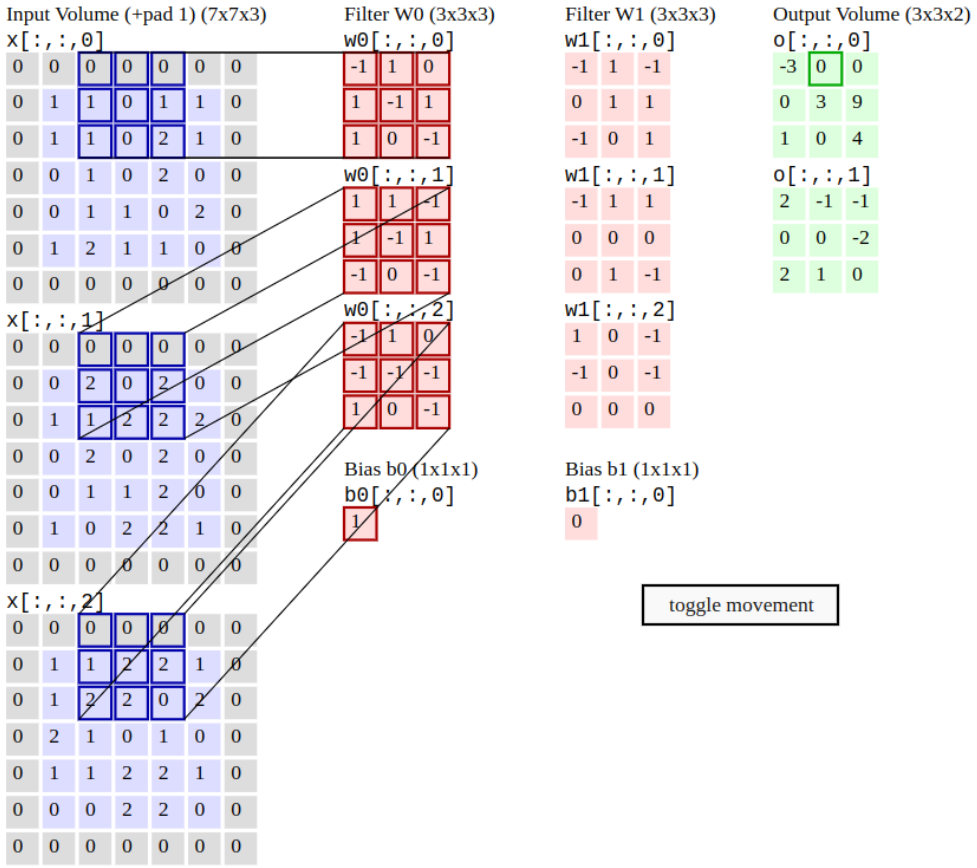


Figure: Great Visualization by CS231n, Source

Channels

Let's have a closer look to how the calculations with more than one filter work:

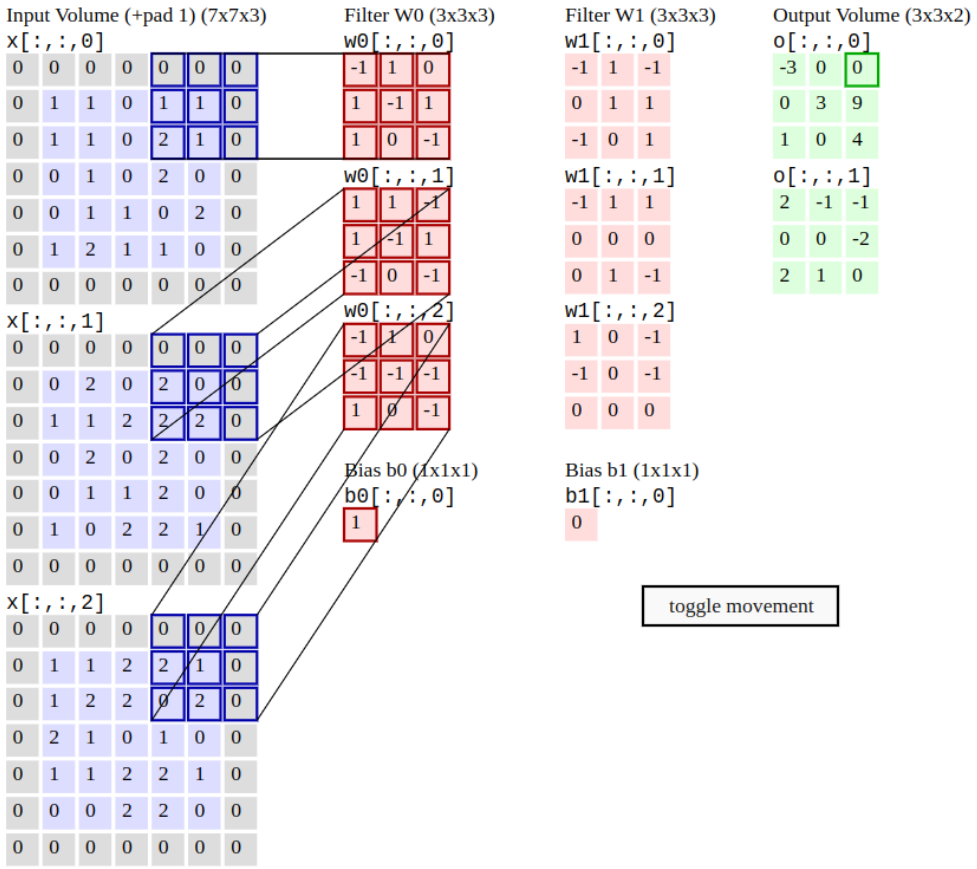


Figure: Great Visualization by CS231n, Source

Channels

Let's have a closer look to how the calculations with more than one filter work:

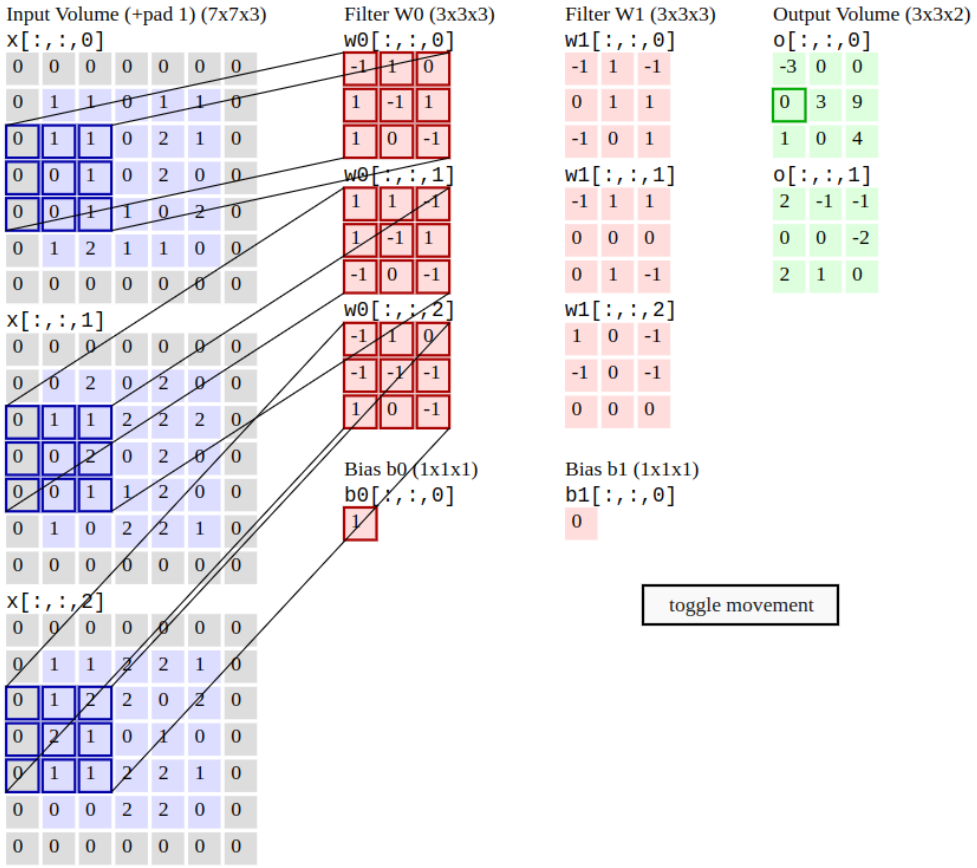


Figure: Great Visualization by CS231n, Source

Channels

Let's have a closer look to how the calculations with more than one filter work:

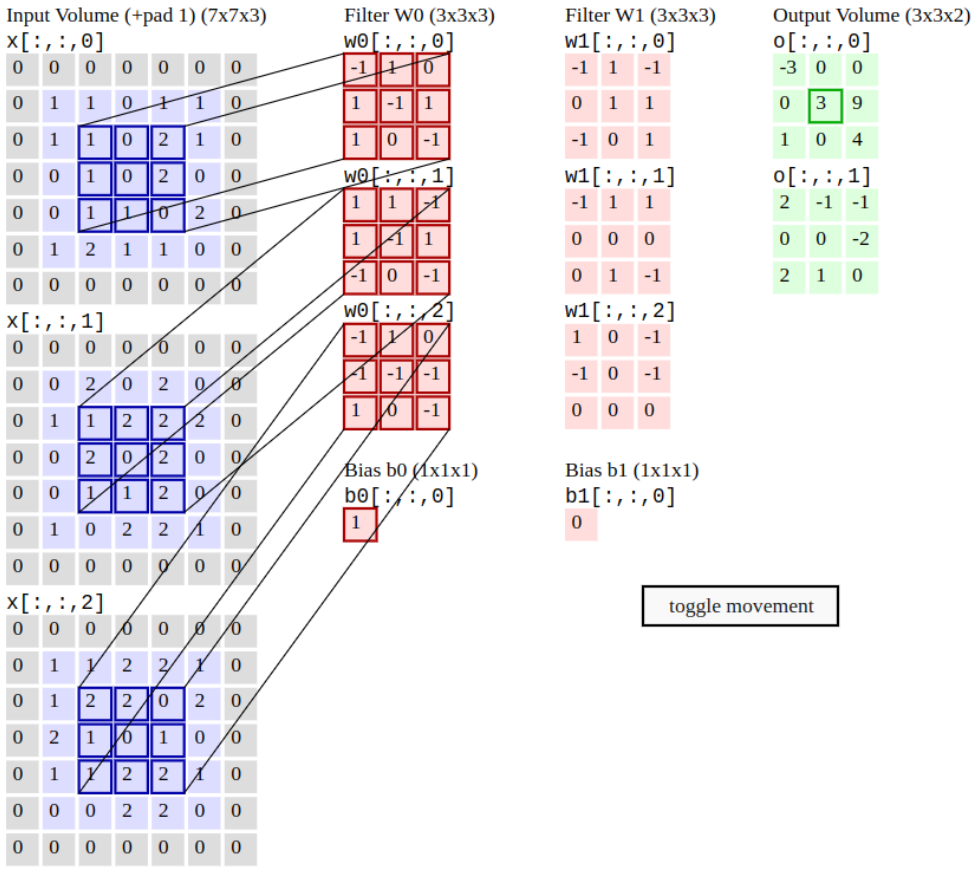


Figure: Great Visualization by CS231n, Source

Channels

Let's have a closer look to how the calculations with more than one filter work:

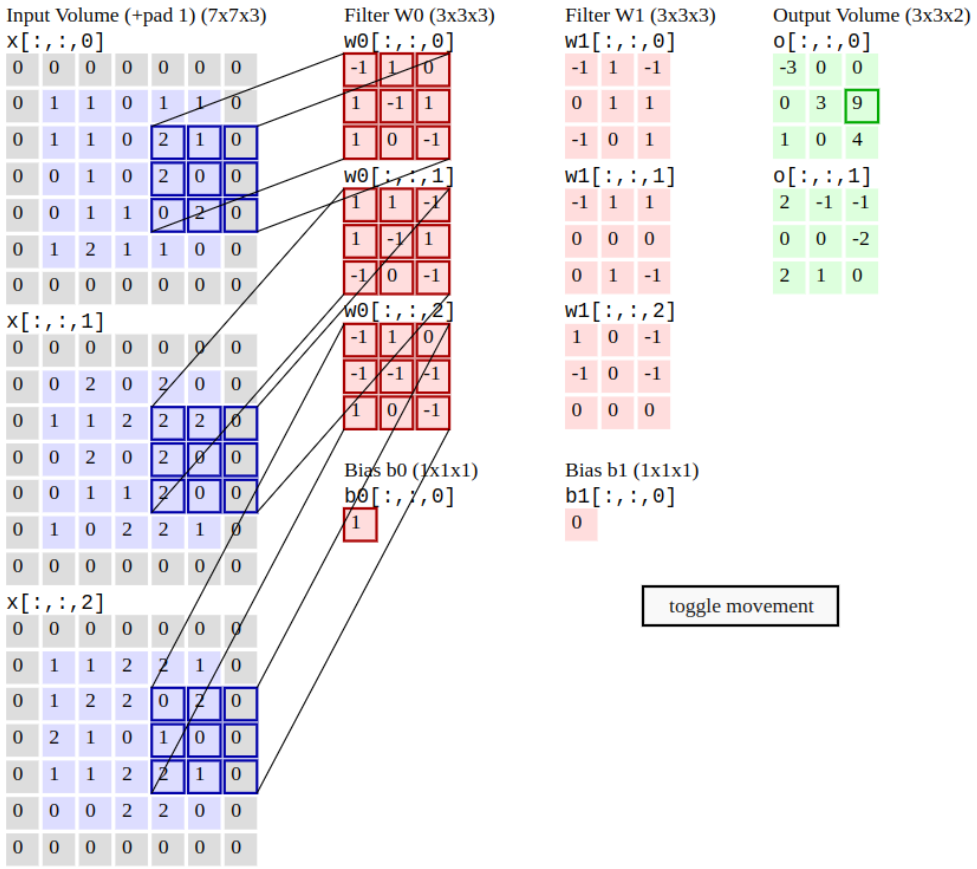


Figure: Great Visualization by CS231n, Source

Channels

Let's have a closer look to how the calculations with more than one filter work:

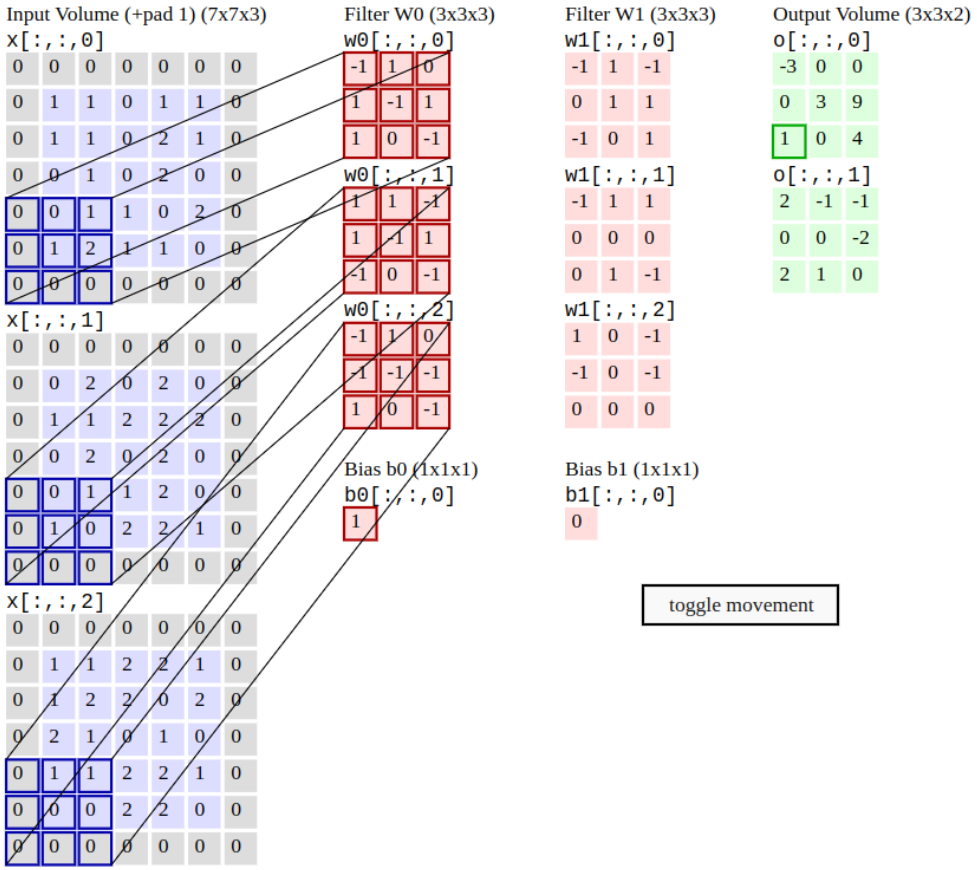


Figure: Great Visualization by CS231n, Source

Channels

Let's have a closer look to how the calculations with more than one filter work:

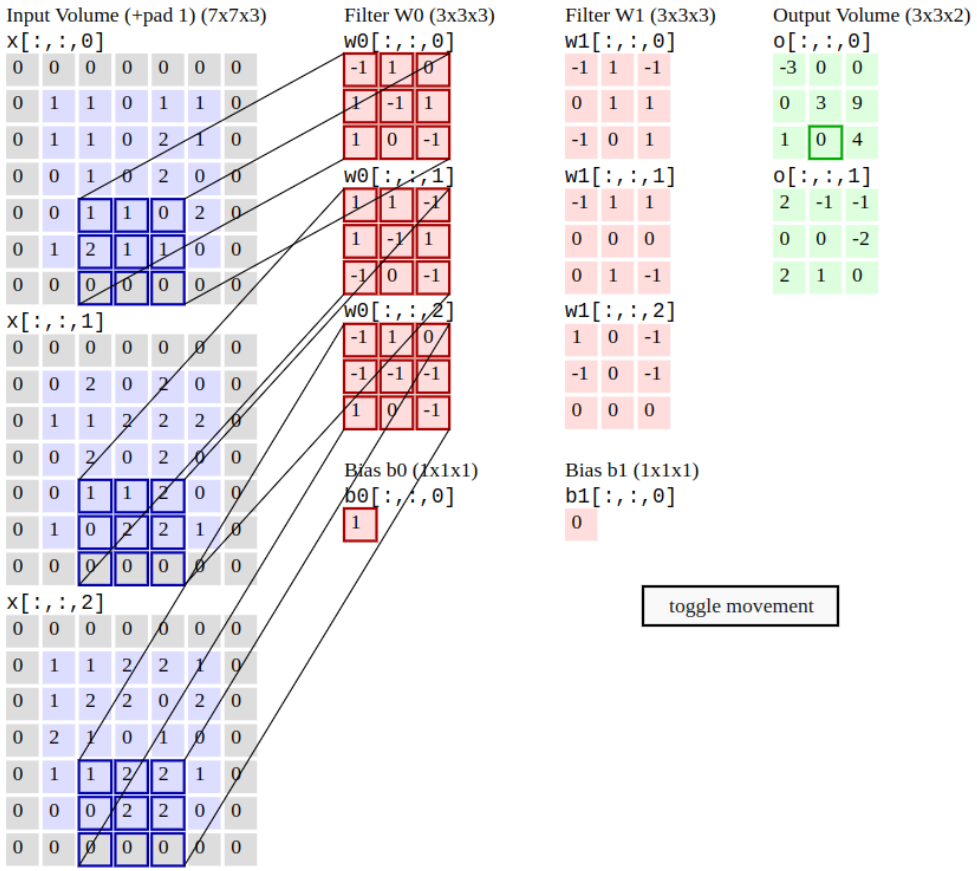


Figure: Great Visualization by CS231n, Source

Channels

Let's have a closer look to how the calculations with more than one filter work:

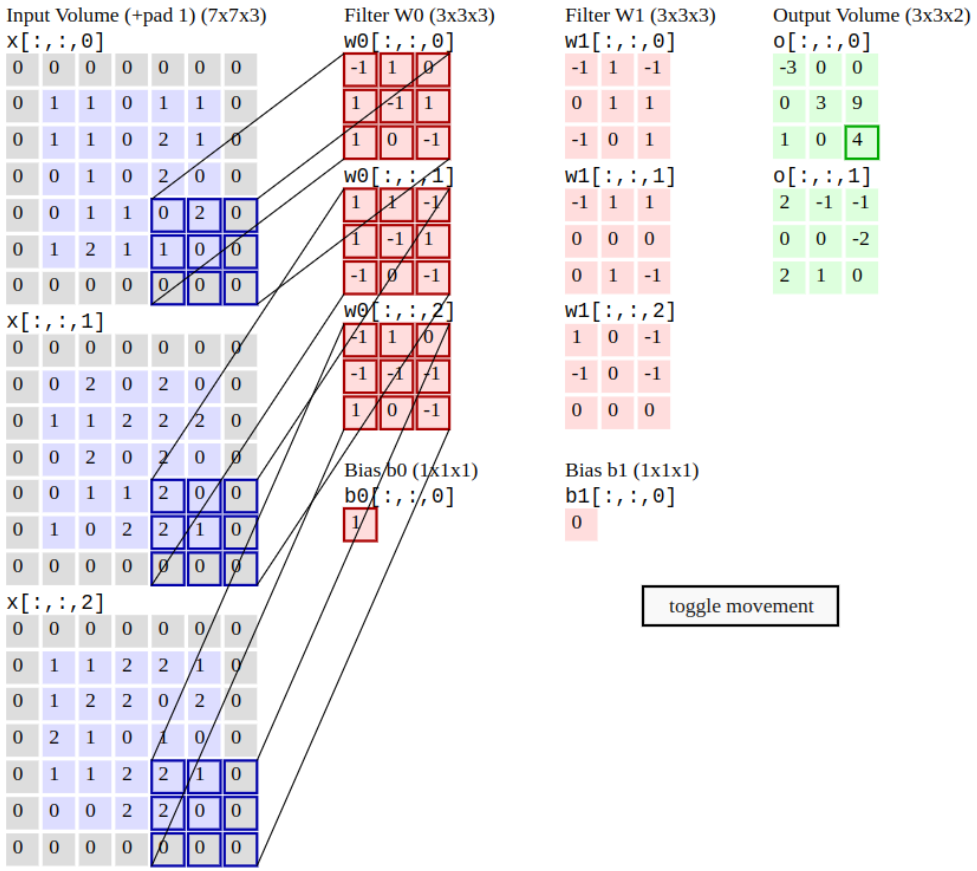


Figure: Great Visualization by CS231n, Source

Channels

Let's have a closer look to how the calculations with more than one filter work:

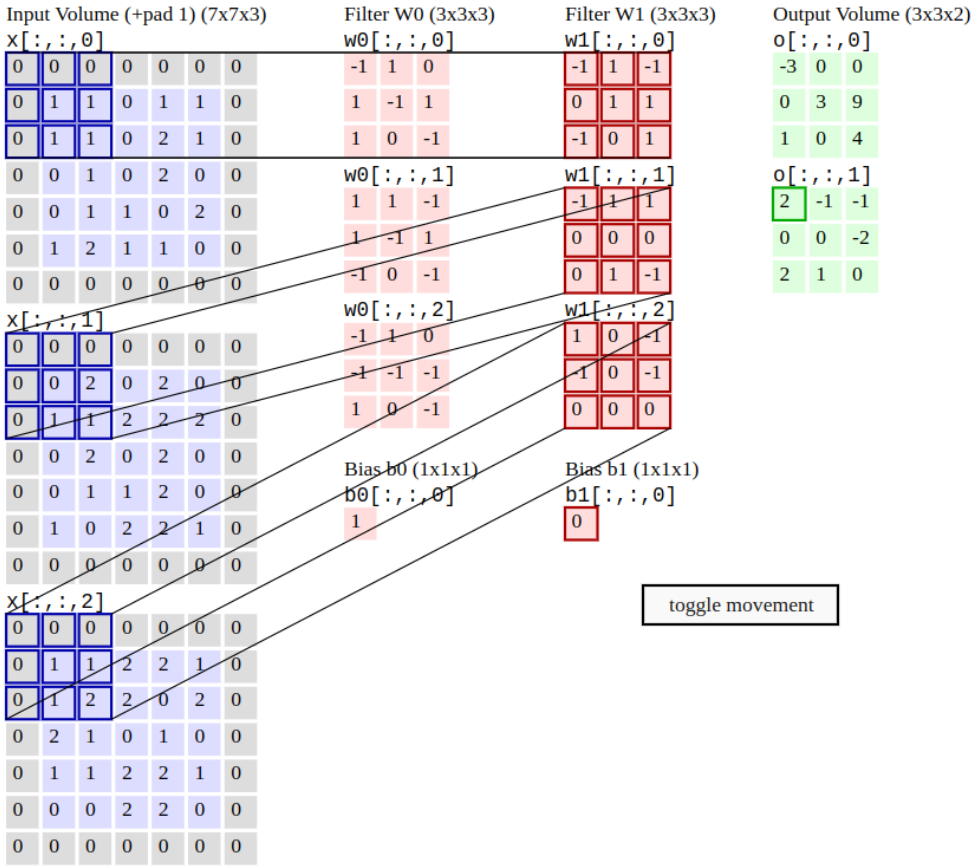


Figure: Great Visualization by CS231n, Source

Channels

Let's have a closer look to how the calculations with more than one filter work:

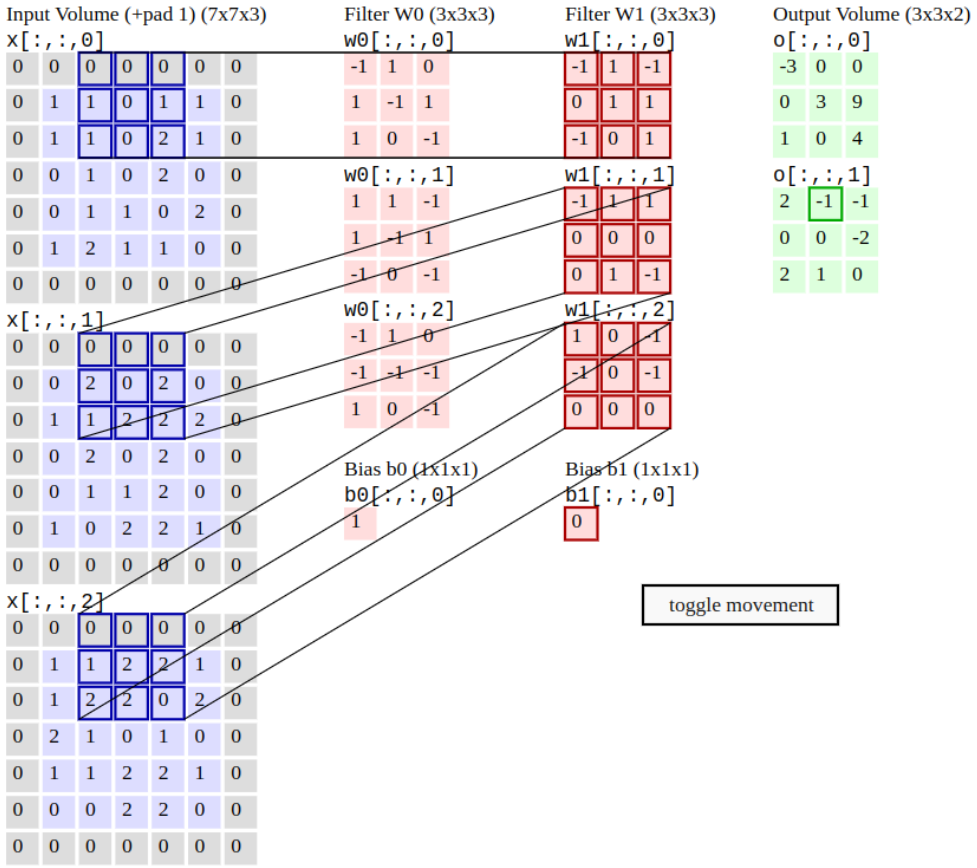


Figure: Great Visualization by CS231n, Source

Channels

Let's have a closer look to how the calculations with more than one filter work:

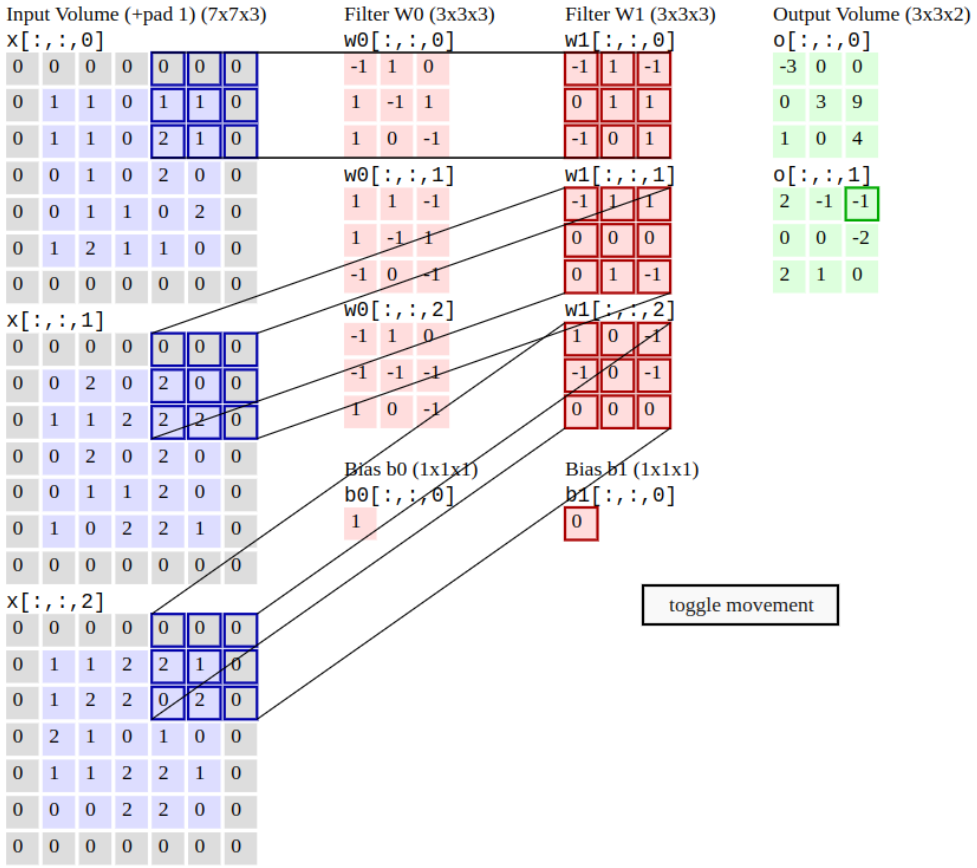


Figure: Great Visualization by CS231n, Source

Channels

Let's have a closer look to how the calculations with more than one filter work:

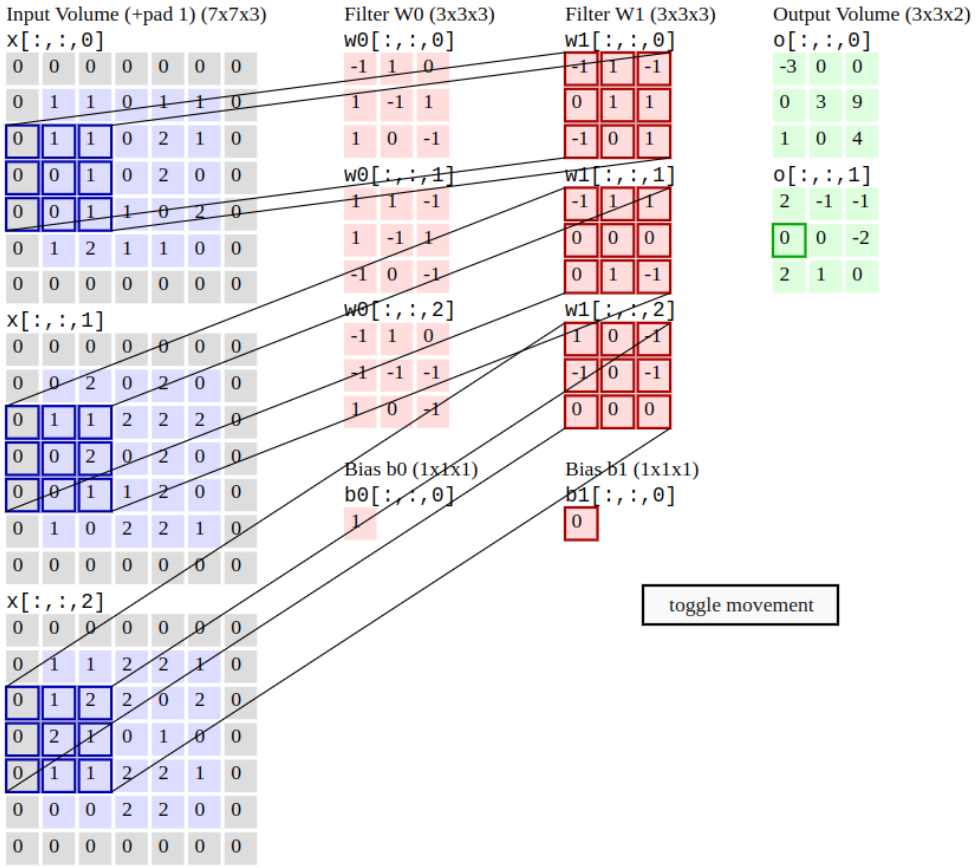


Figure: Great Visualization by CS231n, Source

Channels

Let's have a closer look to how the calculations with more than one filter work:

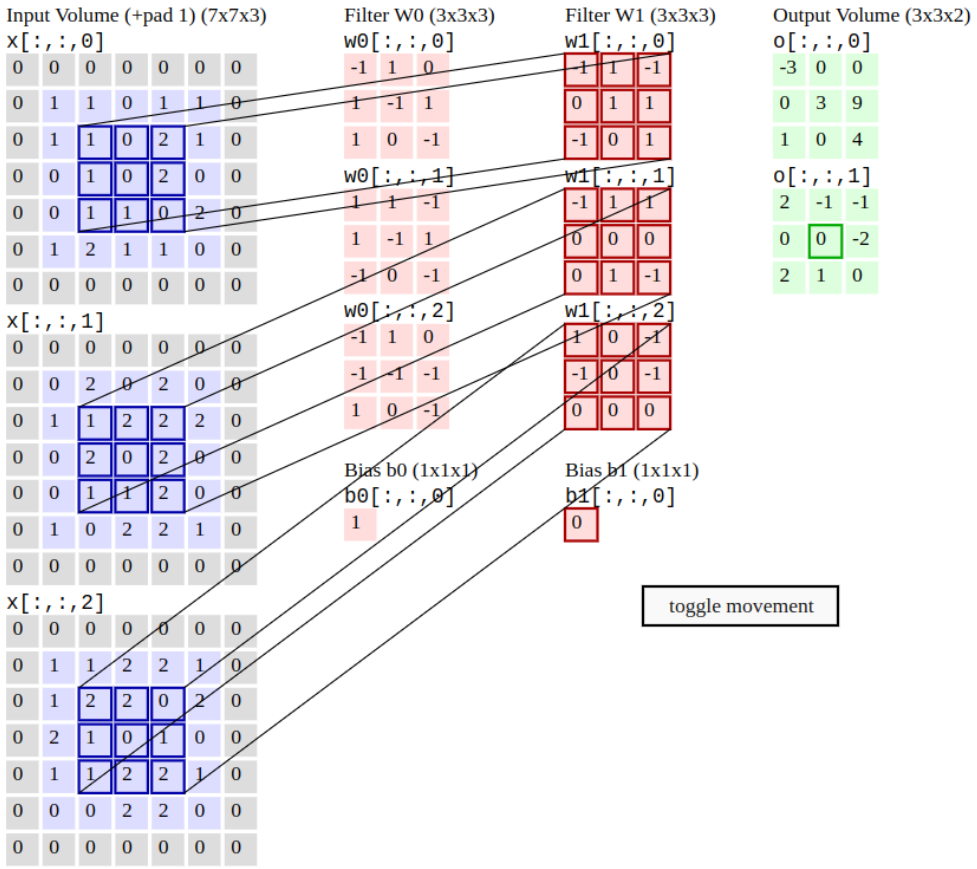


Figure: Great Visualization by CS231n, Source

Channels

Let's have a closer look to how the calculations with more than one filter work:

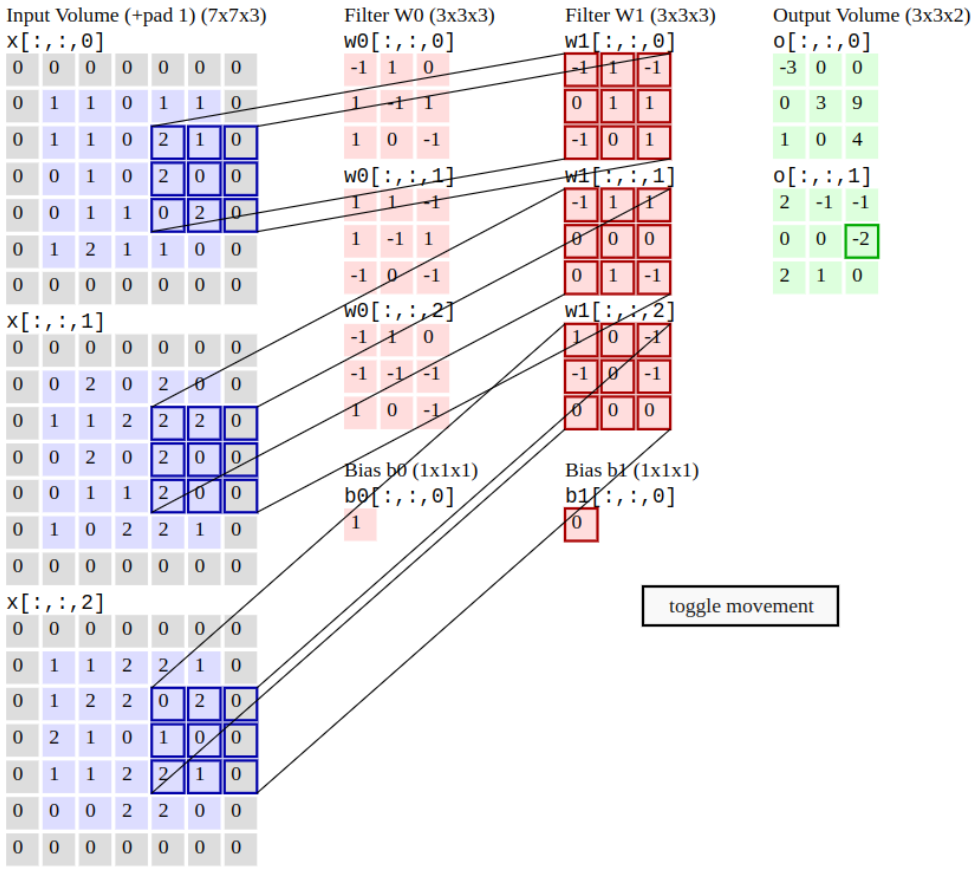


Figure: Great Visualization by CS231n, Source

Channels

Let's have a closer look to how the calculations with more than one filter work:

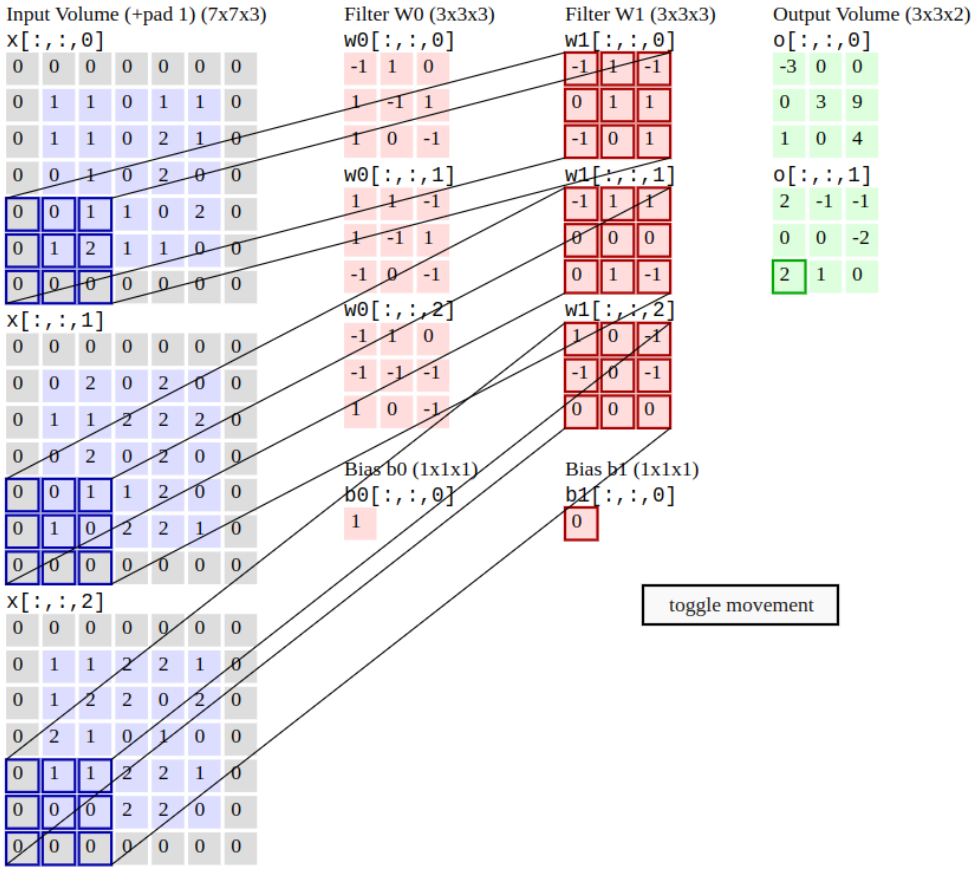


Figure: Great Visualization by CS231n, Source

Channels

Let's have a closer look to how the calculations with more than one filter work:

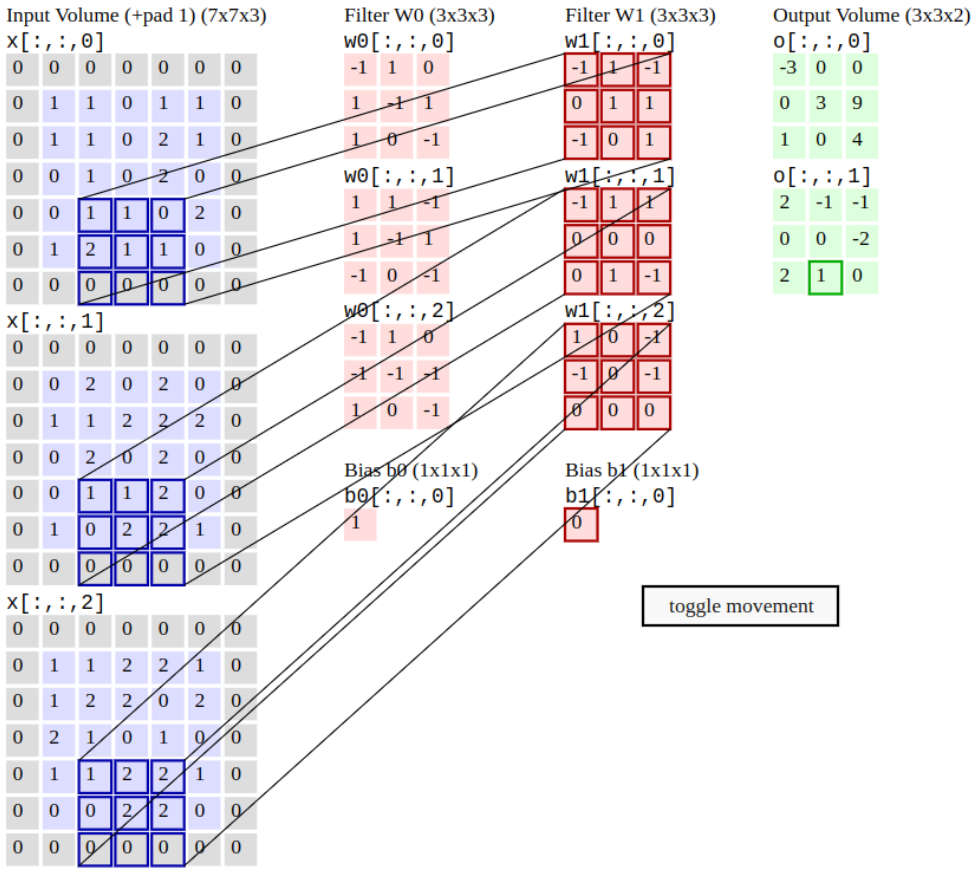


Figure: Great Visualization by CS231n, Source

Channels

Let's have a closer look to how the calculations with more than one filter work:

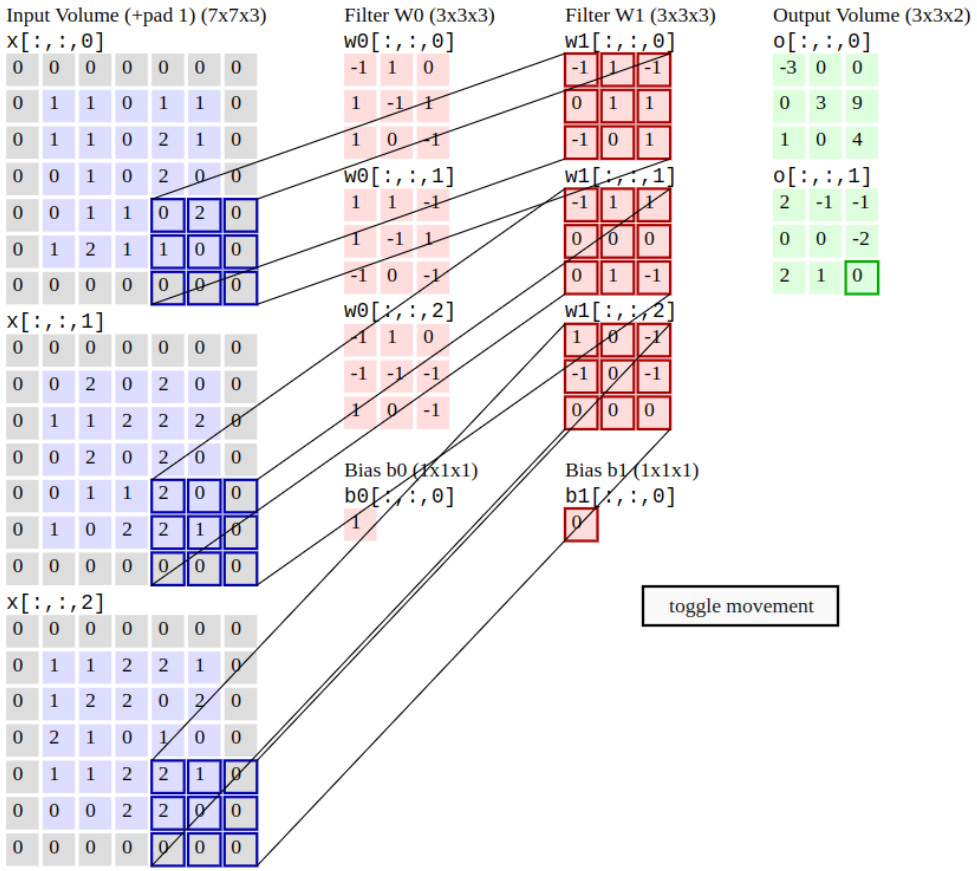


Figure: Great Visualization by CS231n, Source

Channels

- We apply several filters and stack the output together to get a multilayer output, with each layer representing something it learned.

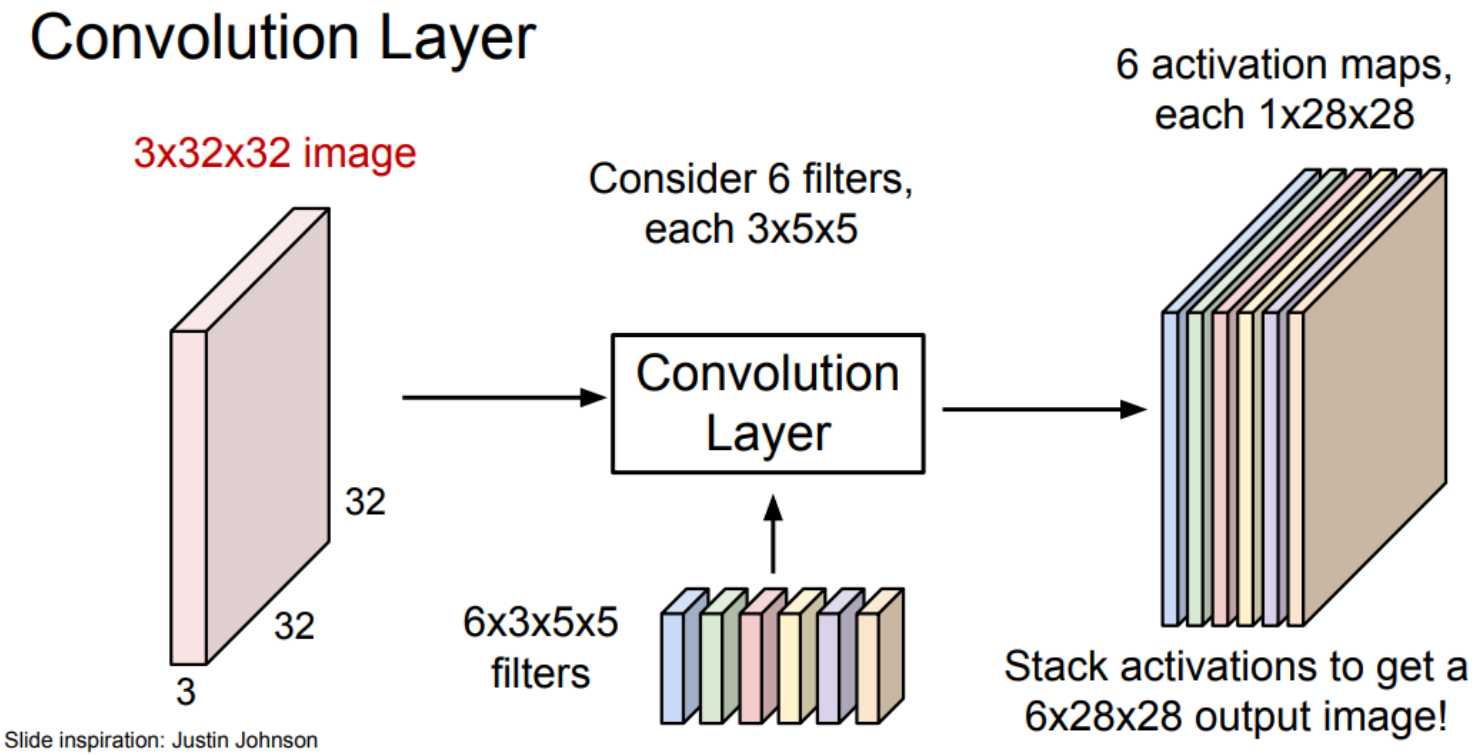


Figure: Slide from CS231n

Channels

- We apply several filters and stack the output together to get a multilayer output, with each layer representing something it learned.
- Some kernels learn to recognize vertical lines, some circles, and some, specific objects:

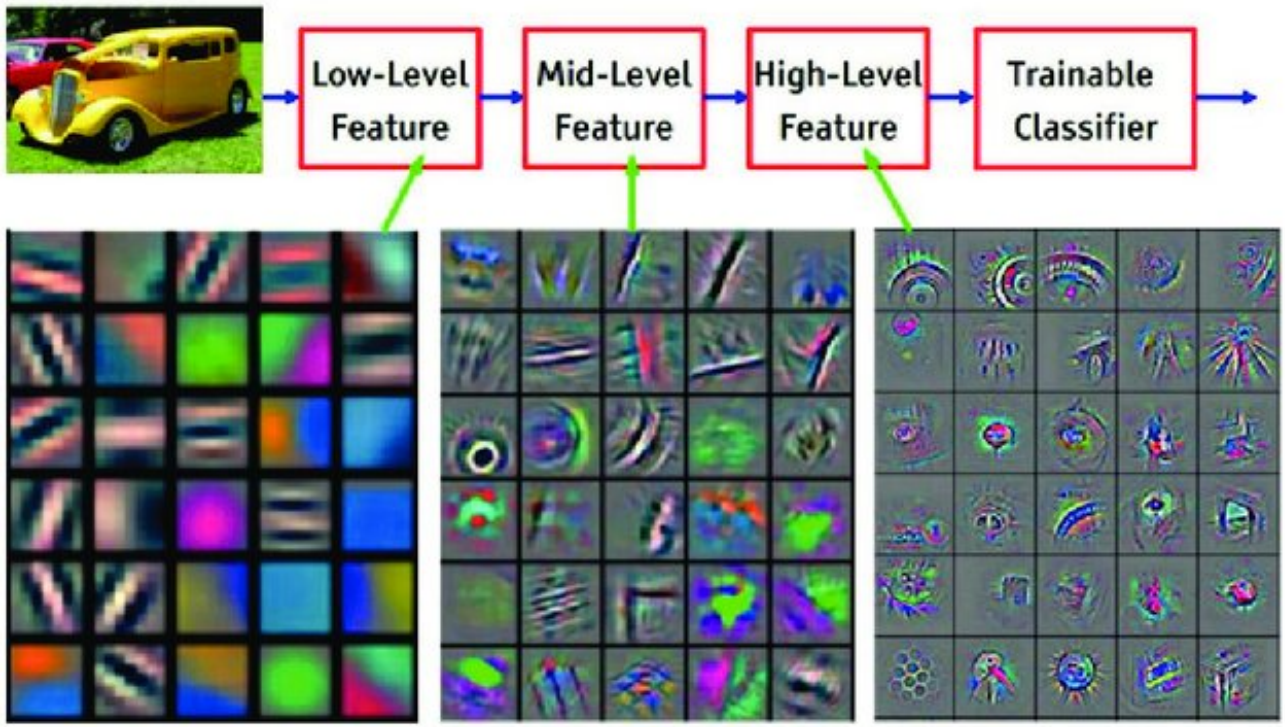


Figure: Each kernel learning something different, Source

Arithmetic of CNNs

We will focus on the following simplified setting:

- 2-D discrete convolutions ($N = 2$)
- Square inputs ($i_1 = i_2 = i$)
- Square kernel size ($k_1 = k_2 = k$)
- Same strides along both axes ($s_1 = s_2 = s$)
- Same zero padding along both axes ($p_1 = p_2 = p$)

Arithmetic of CNNs: No zero padding, unit strides

- For any i and k, and for $s = 1$ and $p = 0$:

$$o = (i - k) + 1 \tag{1}$$

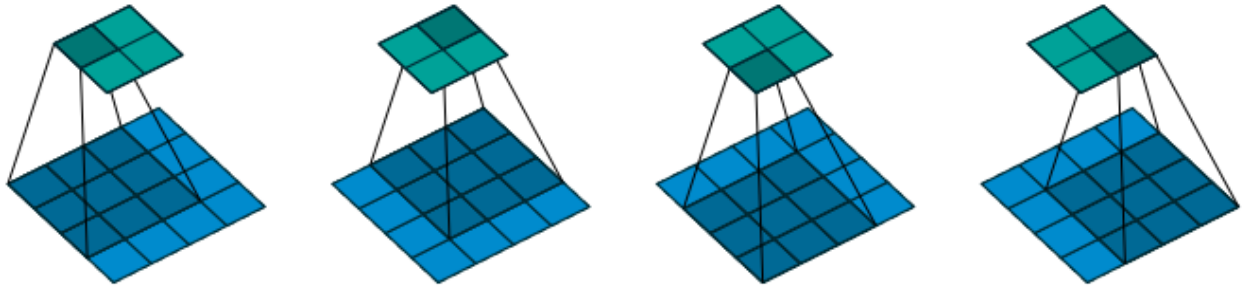


Figure: Convolving a 3 × 3 kernel over a 4 × 4 input using unit strides (i.e., i = 4, k = 3, s = 1 and p = 0)

Arithmetic of CNNs: Zero padding, unit strides

- For any i, k and p, and for $s = 1$:

$$o = (i - k) + 2p + 1 \tag{2}$$

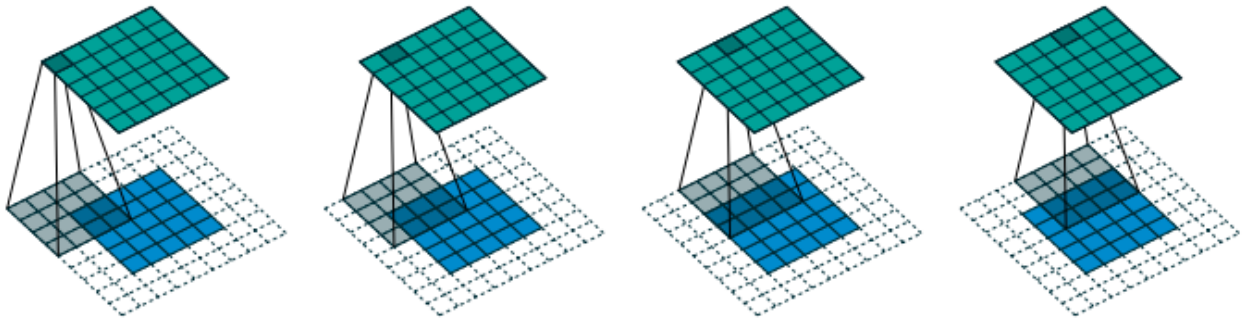


Figure: Convolving a 4 × 4 kernel over a 5 × 5 input padded with a 2 × 2 border of zeros using unit strides (i.e., i = 5, k = 4, s = 1 and p = 2)

Arithmetic of CNNs: Half (same) padding

- For any i and for$k$ odd ($k = 2n + 1, n \in \mathbb{N}$), $s = 1$ and $p = \lfloor k/2 \rfloor = n$:

$$\begin{aligned} o &= i + 2\lfloor k/2 \rfloor - (k-1) \\ &= i + 2n - 2n = i \end{aligned} \tag{3}$$

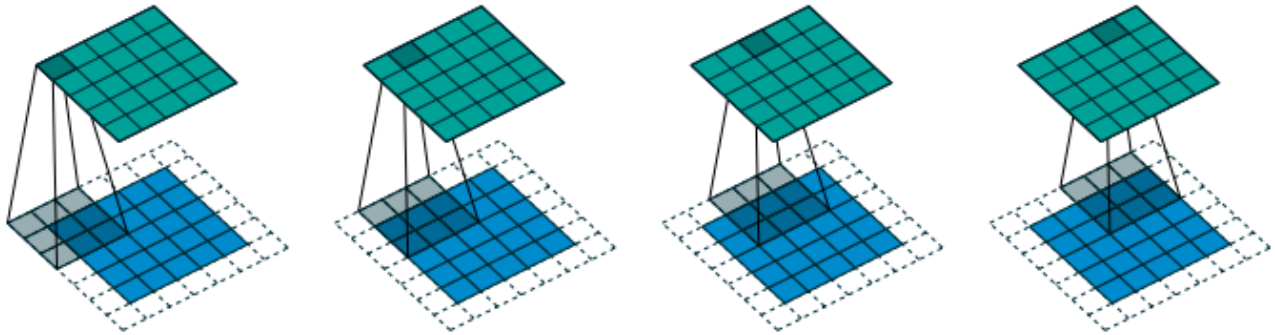


Figure: Convolving a 3 × 3 kernel over a 5 × 5 input using half padding and unit strides (i.e., i = 5, k = 3, s = 1 and p = 1)

Arithmetic of CNNs: Full padding

- For any i and k, and for $p = k-1$ and $s = 1$:

$$\begin{aligned} o &= i + 2(k-1) - (k-1) \\ &= i + 2(k-1 \end{aligned} \tag{4}$$

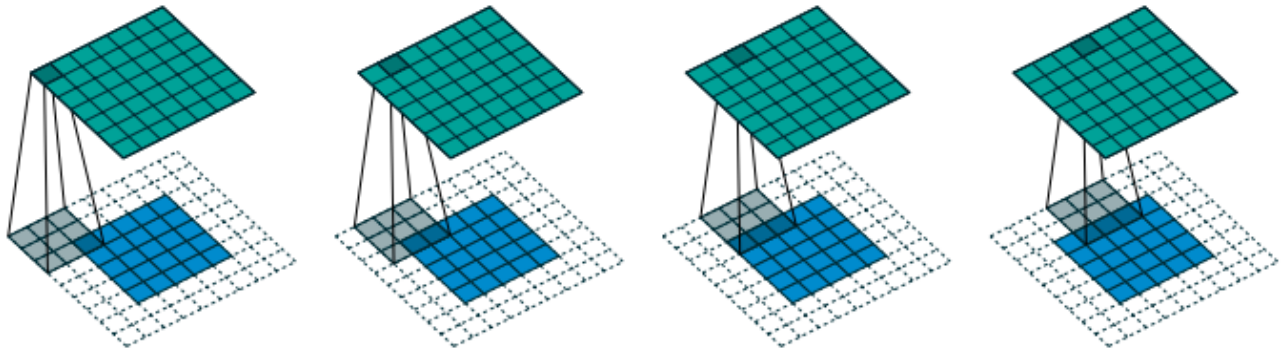


Figure: Convolving a 3 × 3 kernel over a 5 × 5 input using full padding and unit strides (i.e., i = 5, k = 3, s = 1 and p = 2)

Arithmetic of CNNs: No zero padding, non-unit strides

- For any i, k and s, and for $p = 0$:

$$o = \frac{i - k}{s} + 1 \tag{5}$$

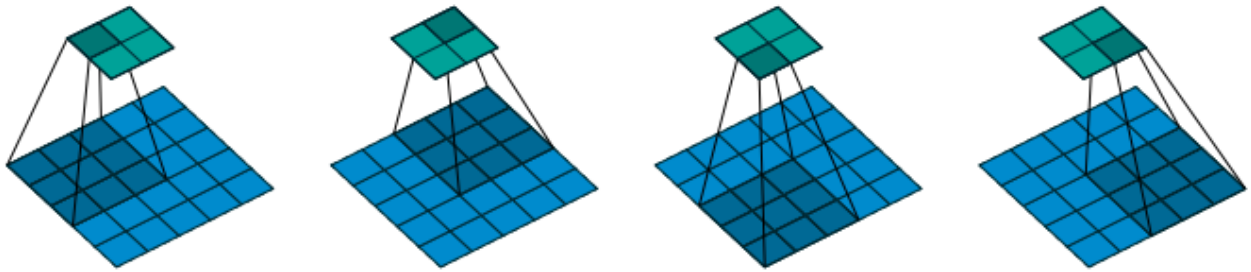


Figure: Convolving a 3 × 3 kernel over a 5 × 5 input using 2 × 2 strides (i.e., i = 5, k = 3, s = 2 and p = 0)

Arithmetic of CNNs: Zero padding, non-unit strides

- For any i, k and s, and for $p = 0$:

$$o = \frac{i + 2p - k}{s} + 1 \tag{6}$$

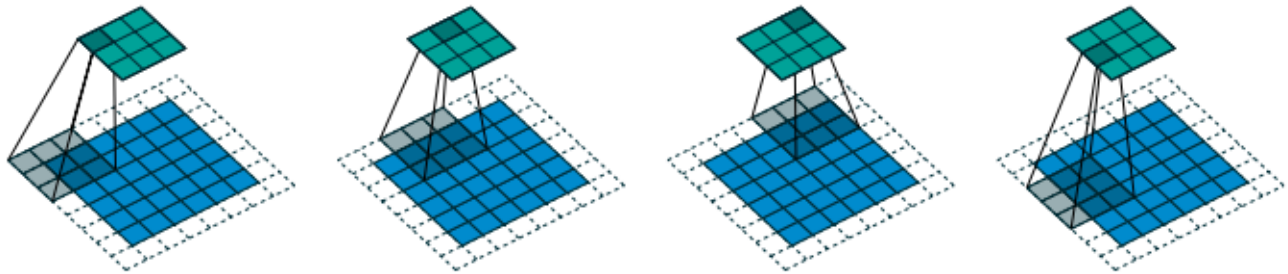


Figure: Convolving a 3 × 3 kernel over a 6 × 6 input padded with a 1 × 1 border of zeros using 2 × 2 strides (i.e., i = 6, k = 3, s = 2 and p = 1). In this case, the bottom row and right column of the zero-padded input are not covered by the kernel

Upsample CNN

- Resizing feature maps is a common operation in many neural networks, especially those that perform some kind of image segmentation task
- This kind of architecture is famously known as the Encoder-Decoder network

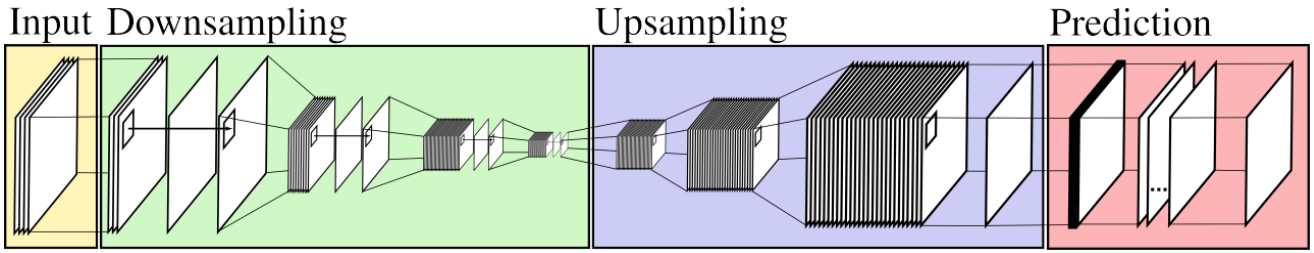


Figure: Schematic of the Downsampling and Upsampling

Upsample CNN: Techniques

- 1- Nearest Neighbors: In Nearest Neighbors, as the name suggests, we take an input pixel value and copy it to the K-Nearest Neighbors, where K depends on the expected output.

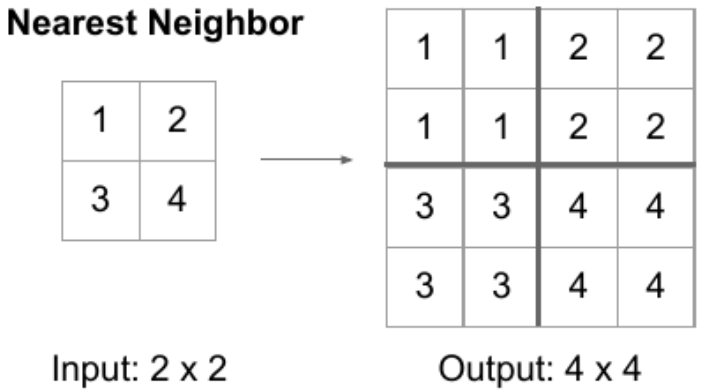


Figure: Nearest Neighbors Upsampling

Upsample CNN: Techniques

- 2- Bi-Linear Interpolation: In Bi-Linear Interpolation, we take the 4 nearest pixel value of the input pixel and perform a weighted average based on the distance of the four nearest cells smoothing the output

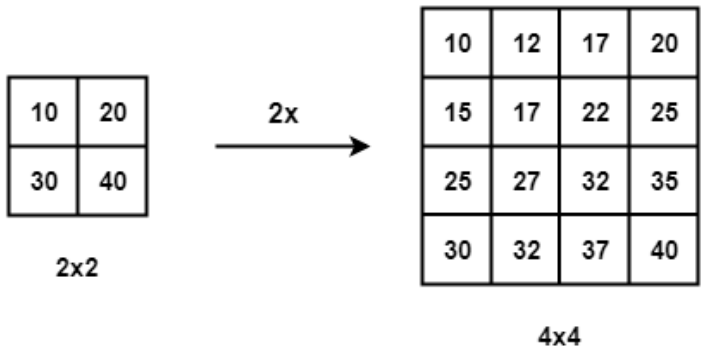


Figure: Bi-Linear Interpolation

Upsample CNN: Techniques

- 3- Bed Of Nails: In Bed of Nails, we copy the value of the input pixel at the corresponding position in the output image and filling zeros in the remaining positions.

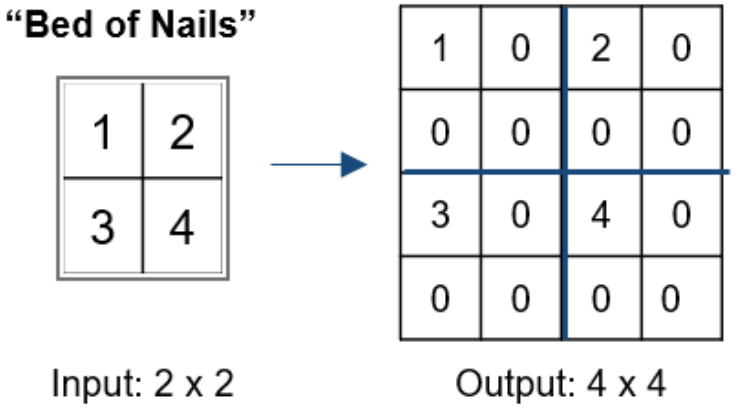


Figure: Bed of Nails Upsampling

Upsample CNN: Techniques

- 4- Max-Unpooling: To perform max-unpooling, first, the index of the maximum value is saved for every max-pooling layer during the encoding step. The saved index is then used during the Decoding step where the input pixel is mapped to the saved index, filling zeros everywhere else

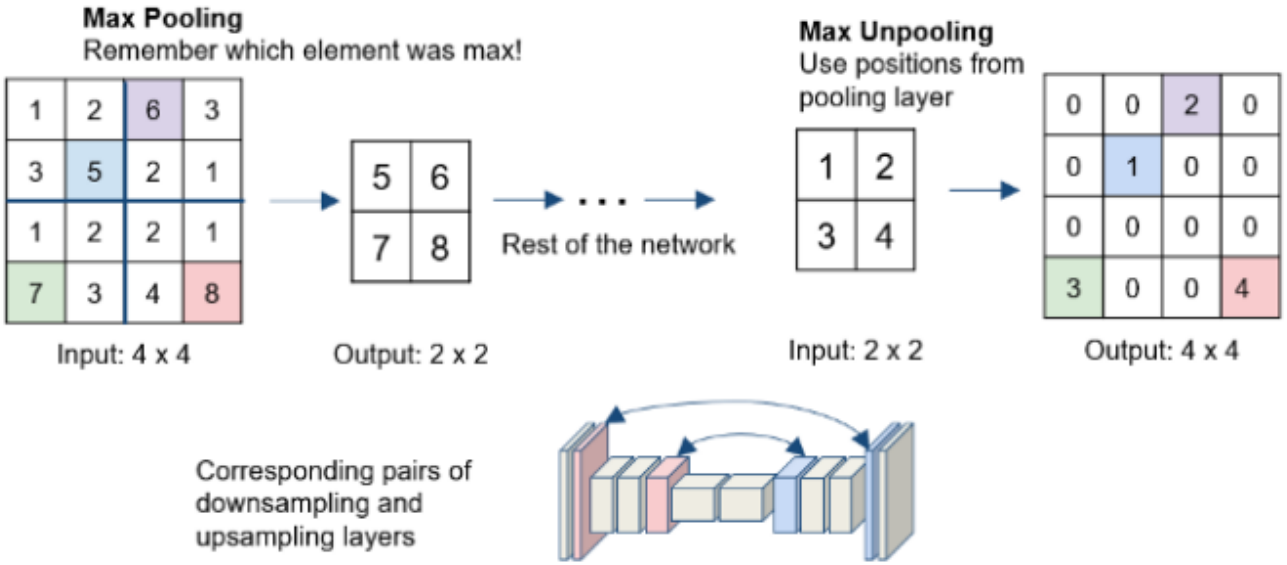


Figure: Max-Unpooling Upsampling

Transposed CNN

- Transposed Convolutions are used to upsample the input feature map to a desired output feature map using some learnable parameters
- They are the backbone of the modern segmentation and super-resolution algorithms
- They provide the best and most generalized upsampling of abstract representations

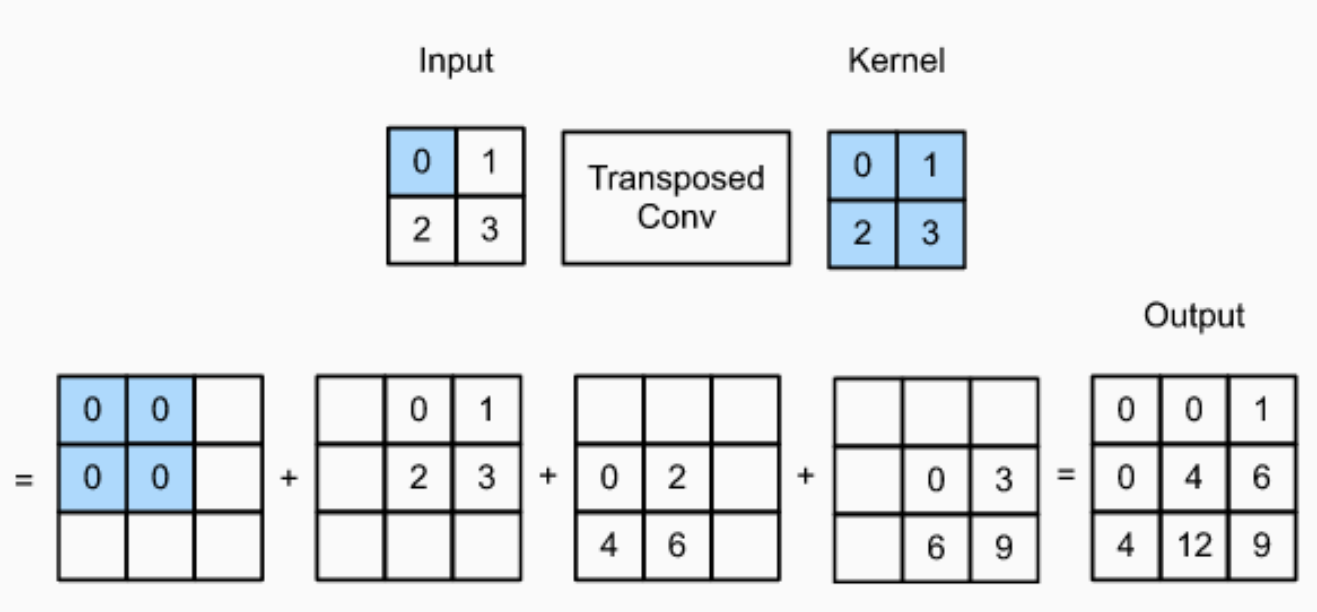


Figure: Transposed convolution with a 2×2 kernel. The shaded portions are a portion of an intermediate tensor as well as the input and kernel tensor elements used for the computation.

Transposed CNN: Problem

- Transposed convolutions suffer from chequered board effects as shown below. The main cause of this is uneven overlap at some parts of the image causing artifacts. This can be fixed or reduced by using kernel-size divisible by the stride, for e.g taking a kernel size of 2×2 or 4×4 when having a stride of 2.

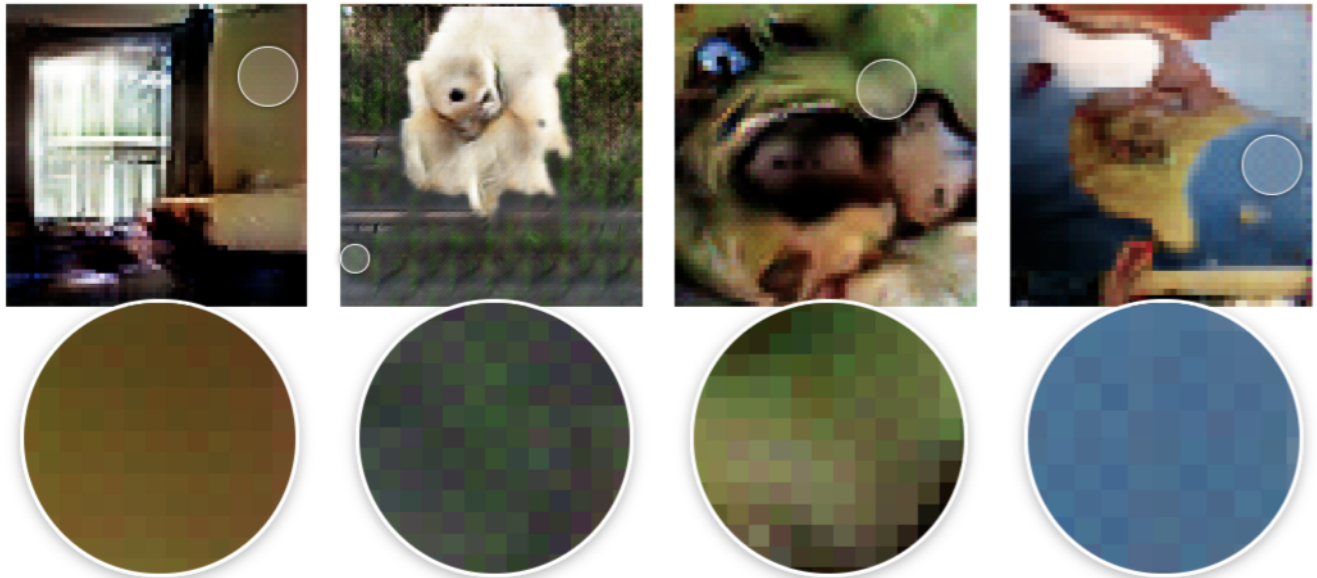


Figure: Checkerboard artifacts effect

Final Notes

Thank You!

Any Question?

Refrences

- Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow - Aurélien Géron - 2019 - Link
- Deep Learning for Vision Systems - Mohamed Elgendy - 2020 - Link
- Dense semantic labeling of sub-decimeter resolution images with convolutional neural networks - Michele Volpi, Devis Tuia - 2016 - Link
- Fully Convolutional Networks for Semantic Segmentation - Jonathan Longm, Evan Shelhamer, Trevor Darrell - 2015 - Link
- Deconvolution and Checkerboard Artifacts - Augustus Odena, Vincent Dumoulin, Chris Olah - 2016 - Link

Refrences


Dongping Tian.
A review on image feature extraction and representation techniques.
International Journal of Multimedia and Ubiquitous Engineering, 8:385–395, 01 2013.


Yandan Wang, John See, Raphael C.-W. Phan, and Yee-Hui Oh.
Efficient spatio-temporal local binary patterns for spontaneous facial micro-expression recognition.
PLOS ONE, 05 2015.


Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola.
Dive into deep learning.
arXiv preprint arXiv:2106.11342, 2021.