Lecture 38 - Introduction to Deep Learning; Convolutional Neural Networks (CNNs)

The most impactful training strategies include:

- Speedup learning by using the ADAM optimization algorithm, it includes a momentum term and adaptively changes the learning rate
- Bagging and boosting
- Network pruning by means of regularization terms or dropout
- Mini-batch learning
- Choosing appropriate activation functions
- Choosing appropriate objective function
    - Cross-Entropy is preferred for classification
    - Mean Squared Error is one of the best choices for regression

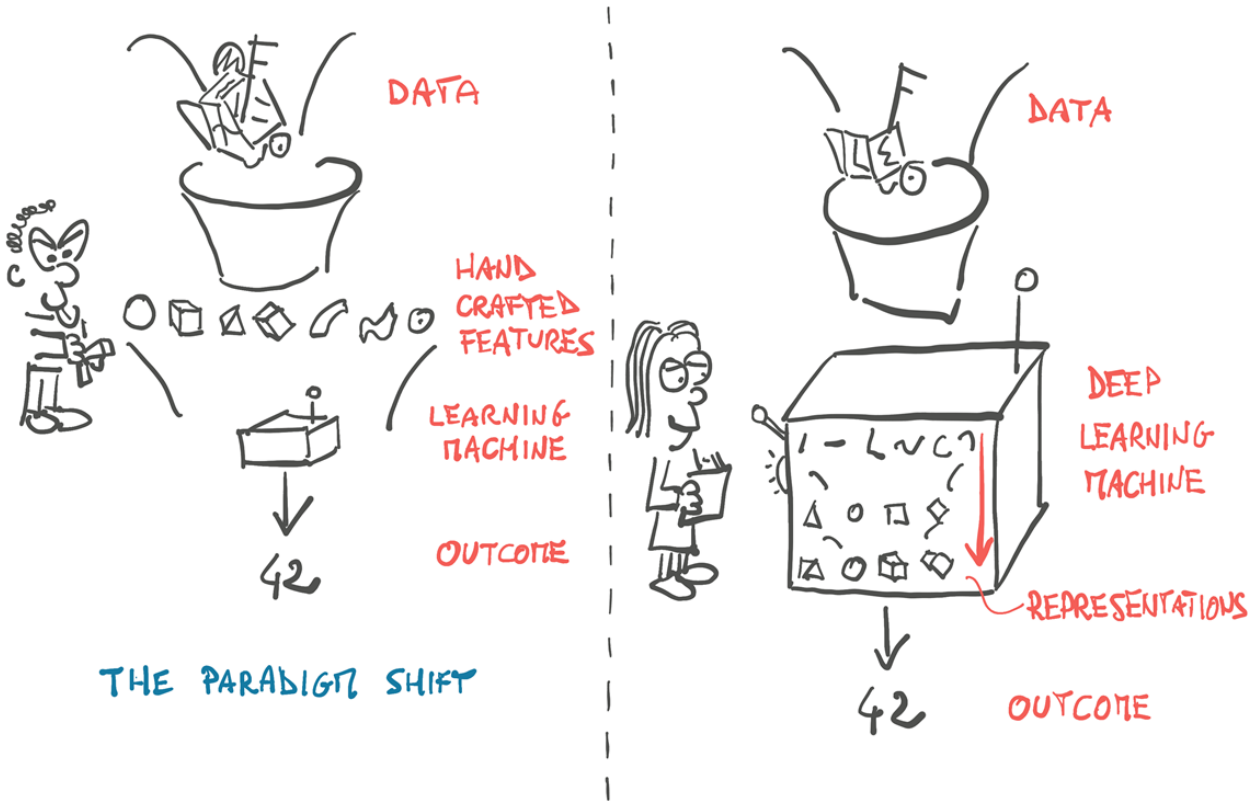
Introduction to Deep Learning

Until the late 2000's, the broader class of systems that fell under the label **machine learning** heavily relied on **feature engineering**. Features are transformations on input data that result in numerical features that facilitate a downstream algorithm, like a classifier, to produce correct outcomes on new data. Feature engineering is aimed at taking the original data and coming up with representations of the same data that can then be fed to an algorithm to solve a problem.

Deep learning, on the other hand, deals with finding such representations automatically, from raw data, in order to successfully perform a task. This is not to say that **feature engineering** has no place with deep learning; we often need to inject some form of prior knowledge in a learning system. However, the ability of a neural network to ingest data and extract useful representations on the basis of examples is what makes deep learning so powerful. The focus of deep learning practitioners is not so much on hand-crafting those representations, but on operating on a mathematical entity so that it discovers representations from the training data autonomously. Often, these automatically-created features are better than those that are hand-crafted! As with many disruptive technologies, this fact has led to a change in perspective.

In [1]:
```
from IPython.display import Image
Image('figures/deeplearning.png', width=700)

#"Deep Learning with PyTorch" by Eli Stevens and Luca Antiga, Manning Publications, 202
```

Out[1]:



At the core of **deep learning** are **neural networks** with *many* layers (hence the *deep* architecture), mathematical entities capable of representing complicated functions through a composition of simpler functions.

Deep learning focuses on designing and optimizing different neural network architecture.

Popular architectures include:

- Convolutional Neural Networks (CNNs)
- Recurrent Neural Networks (RNNs)
- Auto-Encoders (AEs)
- Variational Auto-Encoders (VAEs)
- Generative Adversarial Networks (GANs)
- and many more...

Brief History of Convolutional Neural Networks (CNNs or ConvNets)

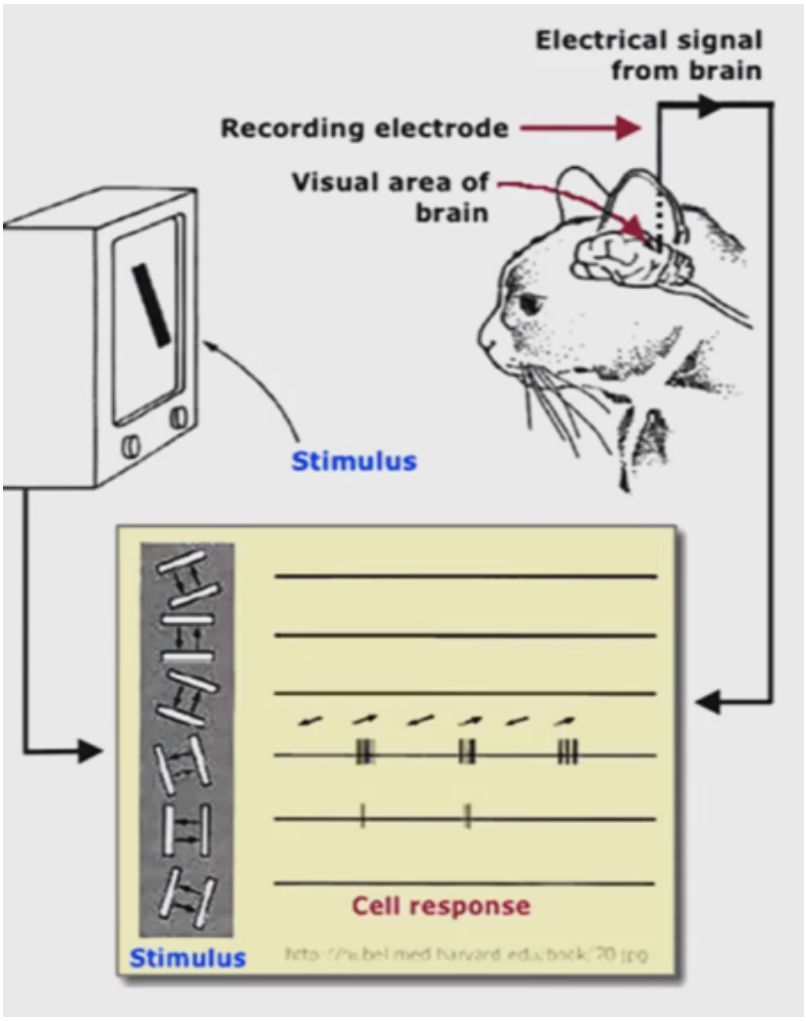
1950-1962 Hubel & Wiesel - Neural Basics of Visual Perception

- Introduced the Neural Basics of Visual Perception, https://www.youtube.com/watch?v=IOHayh06LJ4&t=3s

- Won the 1981 Nobel Prize for Physiology or Medicine
- Hubel, D. H., & Wiesel, T. N.: Receptive fields of single neurones in the cat's striate cortex. The Journal of physiology, 148(3), 574-591 (1959).

In [2]:
```
from IPython.display import Image
Image('figures/Hubel_and_Wiesel.png', width=400)
```

Out[2]:



1980 Fukushima - Neocognitron

- Kunihiko Fukushima introduces **Neocognitron**, inspired by the work of Hubel & Wiesel.
- Neocognitron introduced the basic layers of convolutional networks: convolutional layers and downsampling layers.
- Fukushima, K. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. Biol. Cybernetics 36, 193–202 (1980).

1987 Waibel - Time Delay Neural Network (TDNN)

- First convolutional network was the **Time Delay Neural Network (TDNN)** introduced by Alex Waibel, it achieved shift invariance and it was trained with Backpropagation, using the same

hierarchical structure of the Neocognitron.

- TDNNs are convolutional networks that share weights along the temporal dimension.
- Waibel, Alex: Phoneme Recognition Using Time-Delay Neural Networks. Meeting of the Institute of Electrical, Information and Communication Engineers (IEICE). Tokyo, Japan (1987).

1989 Hampshire & Waibel - Variant of TDNN with two-dimensional convolution

- As TDNNs operated on spectrograms, this two-dimensional convolution variant allows for the phoneme recognition system to both shift in time and in frequency.
- This inspired translation invariance in image processing with convolutional networks.
- Hampshire, J.B and Waibel, A.: Connectionist Architectures for Multi-Speaker Phoneme Recognition. Advances in Neural Information Processing Systems, Morgan Kaufmann (1990).

1990 Yamaguchi - Max-Pooling

- Fixed filtering operation that calculates and propagates the maximum value of a given region.
- Yamaguchi, K.; Sakamoto, K.; Akabane, T.; Fujimoto, Y.: A Neural Network for Speaker-Independent Isolated Word Recognition. First International Conference on Spoken Language Processing, 1077-1980 (1990).

1989 LeCun - Convolution Neural Networks (CNNs)

- Introduced a CNN-based system to recognize hand-written ZIP Code numbers.
- The first system involved convolutions in which the kernel coefficients had been laboriously hand designed.
- Later, LeCun designed a system that used backpropagation to learn the convolution kernel coefficients directly from images, completely automatic.
- Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, L. D. Jackel: Backpropagation Applied to Handwritten Zip Code Recognition, AT&T Bell Laboratories (1989).

2004 GPU Breakthrough

- In 2004, it was shown by K. S. Oh and K. Jung that standard neural networks can be greatly accelerated on GPUs. Their implementation was 20 times faster than an equivalent implementation on CPU.
- Many other successes...

2012 Krizhevsky - ImageNet Competition

- A GPU-based CNN won the "ImageNet Large Scale Visual Recognition Challenge 2012".
- A very deep CNN with over 100 layers by Microsoft won the ImageNet 2015 contest.
- Krizhevsky, Alex; Sutskever, Ilya; Hinton, Geoffrey E.: ImageNet classification with deep convolutional neural networks. Communications of the ACM. 60 (6): 84–90 (2017).

⋮

Interesting Reads

- Rosenfeld, A., Zemel, R., & Tsotsos, J. K. (2018). The elephant in the room. arXiv preprint:1808.03305
- Quanta Magazine: Machine Learning Confronts the Elephant in the Room
- New York Times: Artificial Intelligence Hits the Barrier of Meaning
- Ahsan, U., Madhok, R., & Essa, I. Video jigsaw: Unsupervised learning of spatiotemporal context for video action recognition. In 2019 IEEE Winter Conference on Applications of Computer Vision (WACV), 179-189 (2018).
- Import AI 109: Why solving jigsaw puzzles can lead to better video recognition, learning to spy on people in simulation and transferring to reality, why robots are more insecure than you might think
- Dataset: Totally-Looks-Like

Convolutional Neural Networks (CNNs or ConvNets)

In the last few years, CNNs have become popular in the areas of image recognition, object detection, segmentation, and many other tasks in the field of computer vision. They are also becoming popular in the field of **natural language processing (NLP)**.

The fundamental difference between fully connected layers and convolution layers is the way the weights are connected to each other in the intermediate layers.

One of the biggest challenges of using a linear layer or fully connected layers for computer vision is that they lose all spatial information, and the complexity in terms of the number of weights used by fully connected layers is too big. For example, when we represent a 224×224 color image as a flat array, we would end up with $150,528$-dimensional feature vector ($224 \times 224 \times 3$). When the image is flattened, we lose all the spatial information.

There is a better way! It consists in replacing the dense, fully-connected affine transformation in our neural network unit with a different linear operation: **convolution**.

Why would you use a convolution?

Convolutions are very common operations. Here are some image processing examples:

- **Edge Detection:** can detect edges by convolving with edge masks (e.g., the Sobel edge detectors)

$$S_v = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$$S_h = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

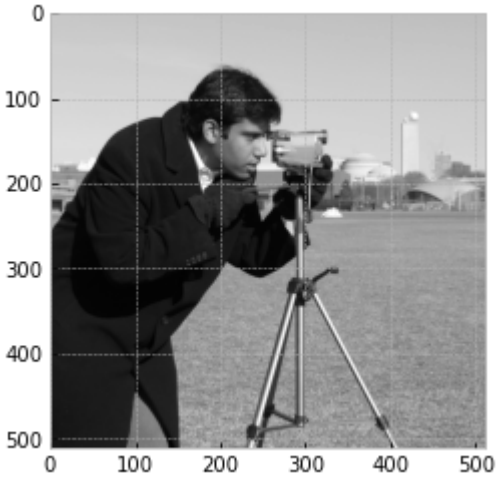
The vertical and horizontal Sobel edge masks.

- **Image Smoothing:** can smooth/blur images using a mean filter.
- **Unsharp Masking:** can sharpen imagery by subtracting a mean filtered image from the original.
- and many more...

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('bmh')
```

In [4]:
```
from skimage import data

# image = data.checkerboard()
image = data.camera()
# image = data.clock()
# image = data.coins()
# image = data.moon()
# image = data.text()
plt.imshow(image, cmap='gray');

image.shape
```

Out[4]:
```
(512, 512)
```

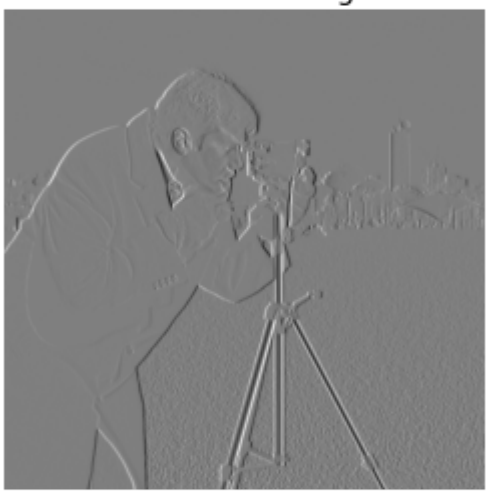


In [3]:

```
import scipy.signal as signal

vMask = np.array([[-1,0,1],[-2,0,2],[-1,0,1]]) # sobel vertical mask/kernel
hMask = np.array([[-1,-2,-1],[0,0,0],[1,2,1]]) # sobel horizontal kernel

gradV = signal.convolve2d(image, vMask, boundary='symm', mode='same')
gradH = signal.convolve2d(image, hMask, boundary='symm', mode='same')

fig, (ax_orig, ax_mag, ax_mag2) = plt.subplots(3,1,figsize=(6,15))
ax_orig.imshow(image, cmap='gray')
ax_orig.set_title("Original")
ax_orig.set_axis_off()
ax_mag.imshow(gradV, cmap='gray')
ax_mag.set_title("Vertical Gradient Magnitude")
ax_mag.set_axis_off()
ax_mag2.imshow(gradH, cmap='gray')
ax_mag2.set_title("Horizontal Gradient Magnitude")
ax_mag2.set_axis_off()
```

Original

Vertical Gradient Magnitude



Horizontal Gradient Magnitude

In [5]:

```
# Mean Filter Mask
mMask = (1/100)*np.ones((10,10))

# Blurring
blurI = signal.convolve2d(image, mMask, boundary='symm', mode='same')
```

```
# Unsharp Masking
sharpI = image + (image - blurI)

fig, (ax_orig, ax_mag, ax_mag2) = plt.subplots(3,1,figsize=(6,15))
ax_orig.imshow(image, cmap='gray')
ax_orig.set_title("Original")
ax_orig.set_axis_off()
ax_mag.imshow(blurI, cmap='gray')
ax_mag.set_title("Blurred Image")
ax_mag.set_axis_off()
ax_mag2.imshow(sharpI, cmap='gray')
ax_mag2.set_title("Sharpened Image")
ax_mag2.set_axis_off()
```

Original

Blurred Image

Sharpened Image

- So, various convolutions have the ability to enhance and extract features of interest.
- The idea behind a convolutional neural network is to learn the features needed to perform classification (or regression) during the learning process for the neural network. This is in

contrast with approaches in which you first identify features of importance, extract them in advance, and then train a classifier (e.g., a neural network) on the extracted features.

Example: if we ought to recognize patterns corresponding to objects, like an airplane in the sky, we will likely need to look at how nearby pixels are arranged, and we would be less interested at how pixels that are far from each other appear in combination. Essentially, it doesn't matter if our image of a Spitfire has a tree or cloud or kite in the corner or not.

In order to translate this intuition in mathematical form, we could compute the weighted sum of a pixel with its immediate neighbors, rather than with all other pixels in the image. This would be equivalent to building weight matrices, one per output feature and output pixel location, in which all weights beyond a certain distance from a center pixel are zero. This will still be a weighted sum, i.e. a linear operation.

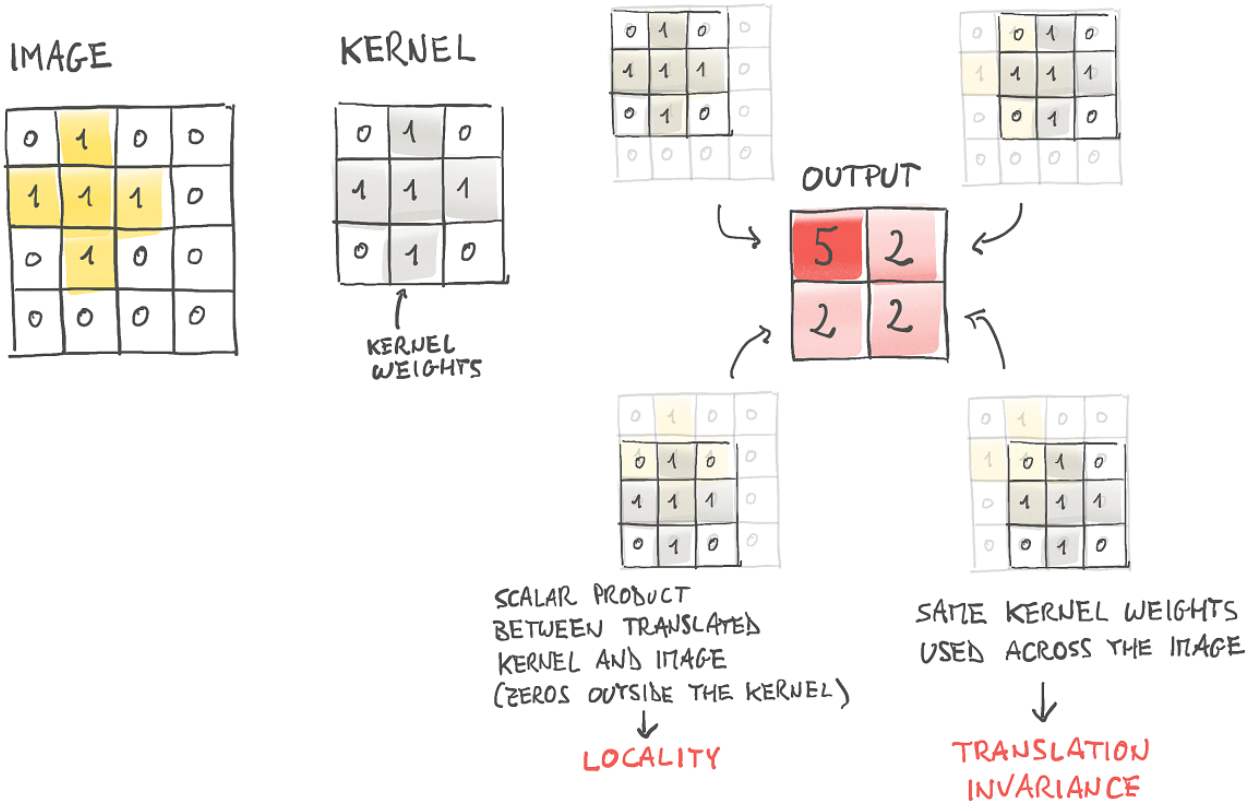
We would like these localized patterns to have an effect on the output no matter their location in the image, i.e. to be **translation-invariant**.

To do that, we would need to force the weights in each per-output-pixel family of patterns to have same values, regardless of pixel location. To achieve this goal, we would need to initialize all weight matrices in a family with the same values, and, during back-propagation, average the gradients for all pixel locations and apply that average as the update to all weights in the family.

For this reason, CNNs are often called **shared weight neural networks**. This is because several connections in the network are tied together to have the same value.

In [3]:
```
#"Deep Learning with PyTorch" by Eli Stevens and Luca Antiga, Manning Publications, 202
Image('figures/convolution.png', width=700)
```

Out[3]:



When we switch standard fully connected networks (MLPs) with CNNs, we get:

- local operations on neighborhoods
- translation-invariance
- models with a lot fewer parameters

CNNs: shared-weight Neural Networks

Consider a 2-D convolution (used, for example, in image processing):

$$g(x,y) * f(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} g(s,t) f(x-s, y-t)$$

where g is the filter and f is the image to be convolved. Essentially, we flip both horizontally and vertically and, then, slide g accross f where at each location we perform a pointwise multiplication and then a sum.

To understand better how are we exactly creating a neural network that extracts features using convolution operations, we need to first consider that a convolutional can be written as a *linear operation* with a **doubly block circulant matrix**.

$$H(x,y) = F(x,y) * g(x,y)$$

is the same as

$$h = Gf$$

where f and h are the vectorized forms of F and G is a doubly block circulant matrix.

Consider the following small image:

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

We can vectorize it and obtain:

$$I^T = [1, 2, 3, \ldots, 14, 15, 16]$$

Let's consider the following kernel:

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

Let G be:

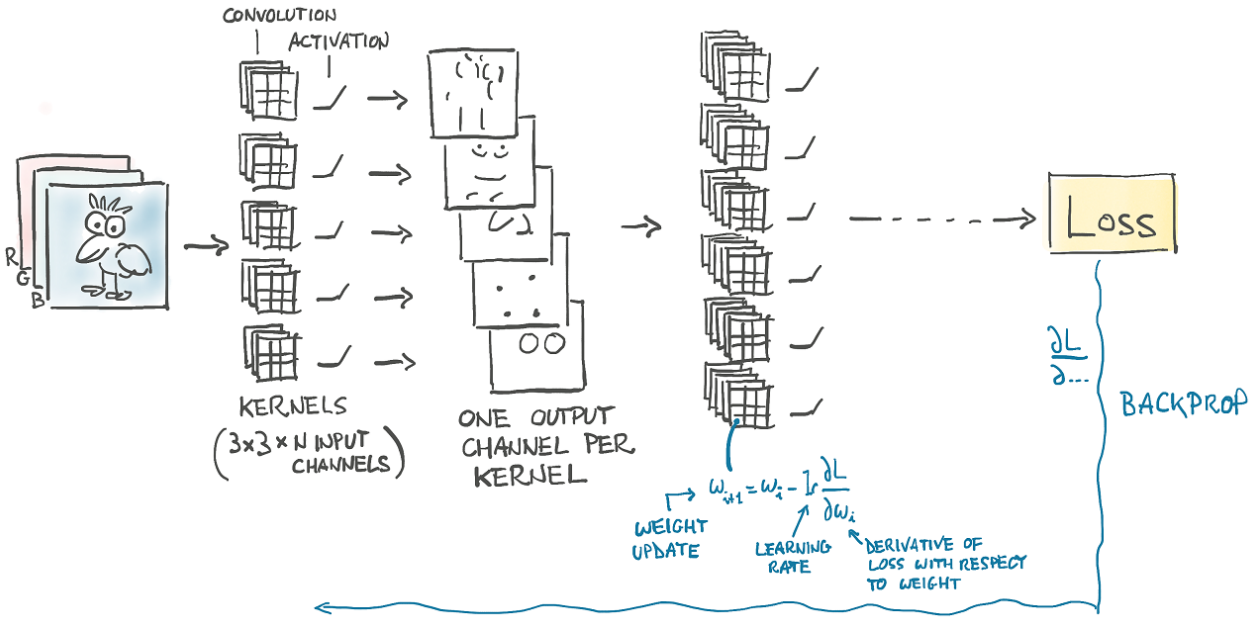
$$G = \begin{bmatrix} -1 & -2 & -3 & 0 & -4 & -5 & -6 & 0 & -7 & -8 & -9 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & -2 & -3 & 0 & -4 & -5 & -6 & 0 & -7 & -8 & -9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & -2 & -3 & 0 & -4 & -5 & -6 & 0 & -7 & -8 & -9 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & -2 & -3 & 0 & -4 & -5 & -6 & 0 & -7 & -8 & -9 \end{bmatrix}$$

So, we can write the convolution as the matrix multiplication:

$$GI$$

In [4]:
```
#"Deep Learning with PyTorch" by Eli Stevens and Luca Antiga, Manning Publications, 202
Image('figures/filters.png', width=700)
```

Out[4]:

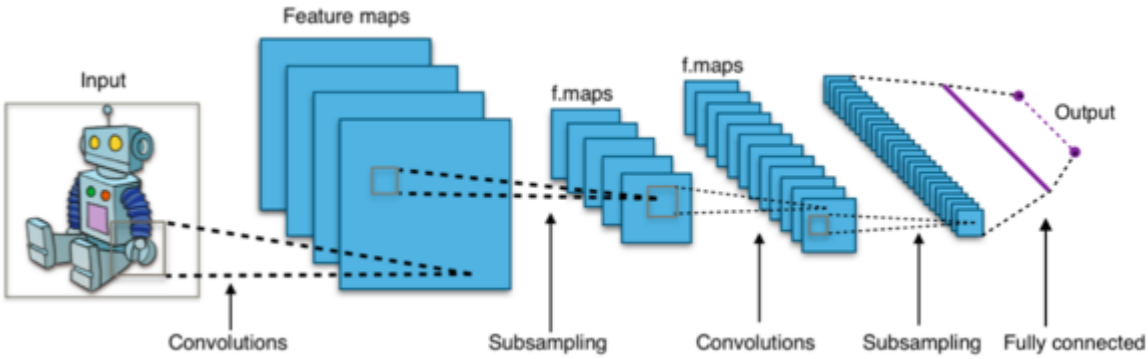


Kernel/Filter Sizes

Small kernels, like 3×3 or 5×5 provide very local information. The problem is how do we know that big picture all structures in our images are 3 pixels or 5 pixels wide?

In order to identify larger objects in an image we will need large convolution kernels. Well, sure, at the limit we could get a 32×32 kernel for a 32×32 image, but we would converge to the old fully connected, affine transformation and lose all the nice properties of convolution.

Another option, which is what is used in convolutional neural networks, is stacking one convolution after the other, and at the same time downsampling the image in-between successive convolutions.



So, on one hand, the first set of kernels operates on small neighborhoods on first-order, low-level features, while the second set of kernels effectively operates on wider neighborhoods, producing features that are compositions of the previous features. This is a very powerful mechanism that provides convolutional neural networks with the ability to see into very complex scenes