February 20-24: Advanced machine learning and data analysis for the physical sciences

Morten Hjorth-Jensen[1,2]

[1]Department of Physics and Center for Computing in Science Education, University of Oslo, Norway
[2]Department of Physics and Astronomy and Facility for Rare Isotope Beams, Michigan State University, East Lansing, Michigan, USA

February 20-24, 2023

Plans for February 20-24

- Basics and mathematics of Convolutional Neural Networks.
- Discussion of first project with examples
- Video of lecture
- Reading recommendations:
    1. For neural networks we recommend Goodfellow et al chapters 6 and 7. For CNNs, see Goodfellow et al chapter 9. See also chapter 11 and 12 on practicalities and applications
    2. Reading suggestions for implementation of CNNs: Aurelien Geron's chapter 13.

Excellent lectures on CNNs and Neural Networks.

- Video on Deep Learning
- Video on Convolutional Neural Networks from MIT
- Video on CNNs from Stanford

m

And Lecture material on CNNs.

- Lectures from IN5400 spring 2019
- Lectures from IN5400 spring 2021
- See also Michael Nielsen's Lectures

Mathematics of CNNs

The mathematics of CNNs is based on the mathematical operation of **convolution**. In mathematics (in particular in functional analysis), convolution is represented by mathematical operations (integration, summation etc) on two functions in order to produce a third function that expresses how the shape of one gets modified by the other. Convolution has a plethora of applications in a variety of disciplines, spanning from statistics to signal processing, computer vision, solutions of differential equations,linear algebra, engineering, and yes, machine learning.

Mathematically, convolution is defined as follows (one-dimensional example): Let us define a continuous function $y(t)$ given by

$$y(t) = \int x(a)w(t-a)da,$$

where $x(a)$ represents a so-called input and $w(t-a)$ is normally called the weight function or kernel.

The above integral is written in a more compact form as

$$y(t) = (x * w)(t).$$

The discretized version reads

$$y(t) = \sum_{a=-\infty}^{a=\infty} x(a)w(t-a).$$

Computing the inverse of the above convolution operations is known as deconvolution and the process is commutative.

How can we use this? And what does it mean? Let us study some familiar examples first.

Convolution Examples: Polynomial multiplication

Our first example is that of a multiplication between two polynomials, which we will rewrite in terms of the mathematics of convolution. In the final stage, since the problem here is a discrete one, we will recats the final expression in terms of a matrix-vector multiplication, where the matrix is a so-called Toeplitz matrix.

Let us look a the following polynomials to second and third order, respectively:

$$p(t) = \alpha_0 + \alpha_1 t + \alpha_2 t^2,$$

and

$$s(t) = \beta_0 + \beta_1 t + \beta_2 t^2 + \beta_3 t^3.$$

The polynomial multiplication gives us a new polynomial of degree 5

$$z(t) = \delta_0 + \delta_1 t + \delta_2 t^2 + \delta_3 t^3 + \delta_4 t^4 + \delta_5 t^5.$$

Efficient Polynomial Multiplication

Computing polynomial products can be implemented efficiently if we rewrite the more brute force multiplications using convolution. We note first that the new coefficients are given as

We note that $\alpha_i = 0$ except for $i \in \{0, 1, 2\}$ and $\beta_i = 0$ except for $i \in \{0, 1, 2, 3\}$.

We can then rewrite the coefficients δ_j using a discrete convolution as

$$\delta_j = \sum_{i=-\infty}^{i=\infty} \alpha_i \beta_{j-i} = (\alpha * \beta)_j,$$

or as a double sum with restriction $l = i + j$

$$\delta_l = \sum_{ij} \alpha_i \beta_j.$$

A more efficient way of coding the above Convolution

Since we only have a finite number of α and β values which are non-zero, we can rewrite the above convolution expressions as a matrix-vector multiplication

$$\boldsymbol{\delta} = \begin{bmatrix} \alpha_0 & 0 & 0 & 0 \\ \alpha_1 & \alpha_0 & 0 & 0 \\ \alpha_2 & \alpha_1 & \alpha_0 & 0 \\ 0 & \alpha_2 & \alpha_1 & \alpha_0 \\ 0 & 0 & \alpha_2 & \alpha_1 \\ 0 & 0 & 0 & \alpha_2 \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix}.$$

The process is commutative and we can easily see that we can rewrite the multiplication in terms of a matrix holding β and a vector holding α. In this case we have

$$\boldsymbol{\delta} = \begin{bmatrix} \beta_0 & 0 & 0 \\ \beta_1 & \beta_0 & 0 \\ \beta_2 & \beta_1 & \beta_0 \\ \beta_3 & \beta_2 & \beta_1 \\ 0 & \beta_3 & \beta_2 \\ 0 & 0 & \beta_3 \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \end{bmatrix}.$$

Note that the use of these matrices is for mathematical purposes only and not implementation purposes. When implementing the above equation we do

not encode (and allocate memory) the matrices explicitely. We rather code the convolutions in the minimal memory footprint that they require.

The above matrices are examples of so-called Toeplitz matrices. A Toeplitz matrix is a matrix in which each descending diagonal from left to right is constant. For instance the last matrix, which we rewrite as

$$\boldsymbol{A} = \begin{bmatrix} a_0 & 0 & 0 \\ a_1 & a_0 & 0 \\ a_2 & a_1 & a_0 \\ a_3 & a_2 & a_1 \\ 0 & a_3 & a_2 \\ 0 & 0 & a_3 \end{bmatrix},$$

with elements $a_{ii} = a_{i+1,j+1} = a_{i-j}$ is an example of a Toeplitz matrix. Such a matrix does not need to be a square matrix. Toeplitz matrices are also closely connected with Fourier series discussed below, because the multiplication operator by a trigonometric polynomial, compressed to a finite-dimensional space, can be represented by such a matrix. The example above shows that we can represent linear convolution as multiplication of a Toeplitz matrix by a vector.

Convolution Examples: Principle of Superposition and Periodic Forces (Fourier Transforms)

For problems with so-called harmonic oscillations, given by for example the following differential equation

$$m\frac{d^2x}{dt^2} + \eta\frac{dx}{dt} + x(t) = F(t),$$

where $F(t)$ is an applied external force acting on the system (often called a driving force), one can use the theory of Fourier transformations to find the solutions of this type of equations.

If one has several driving forces, $F(t) = \sum_n F_n(t)$, one can find the particular solution $x_{pn}(t)$ to the above differential equation for each F_n. The particular solution for the entire driving force is then given by a series like

$$x_p(t) = \sum_n x_{pn}(t). \tag{1}$$

This is known as the principle of superposition. It only applies when the homogenous equation is linear. Superposition is especially useful when $F(t)$ can be written as a sum of sinusoidal terms, because the solutions for each sinusoidal (sine or cosine) term is analytic.

Driving forces are often periodic, even when they are not sinusoidal. Periodicity implies that for some time t our function repeats itself periodically after a period τ, that is

$$F(t+\tau) = F(t). \tag{2}$$

One example of a non-sinusoidal periodic force is a square wave. Many components in electric circuits are non-linear, for example diodes. This makes many wave forms non-sinusoidal even when the circuits are being driven by purely sinusoidal sources.

Simple Code Example

The code here shows a typical example of such a square wave generated using the functionality included in the **scipy** Python package. We have used a period of $\tau = 0.2$.

```
import numpy as np
import math
from scipy import signal
import matplotlib.pyplot as plt

# number of points
n = 500
# start and final times
t0 = 0.0
tn = 1.0
# Period
t = np.linspace(t0, tn, n, endpoint=False)
SqrSignal = np.zeros(n)
SqrSignal = 1.0+signal.square(2*np.pi*5*t)
plt.plot(t, SqrSignal)
plt.ylim(-0.5, 2.5)
plt.show()
```

For the sinusoidal example the period is $\tau = 2\pi/\omega$. However, higher harmonics can also satisfy the periodicity requirement. In general, any force that satisfies the periodicity requirement can be expressed as a sum over harmonics,

$$F(t) = \frac{f_0}{2} + \sum_{n>0} f_n \cos(2n\pi t/\tau) + g_n \sin(2n\pi t/\tau). \tag{3}$$

Wrapping up Fourier transforms

We can write down the answer for $x_{pn}(t)$, by substituting f_n/m or g_n/m for F_0/m. By writing each factor $2n\pi t/\tau$ as $n\omega t$, with $\omega \equiv 2\pi/\tau$,

$$F(t) = \frac{f_0}{2} + \sum_{n>0} f_n \cos(n\omega t) + g_n \sin(n\omega t). \tag{4}$$

The solutions for $x(t)$ then come from replacing ω with $n\omega$ for each term in the particular solution,

$$
\begin{aligned}
x_p(t) &= \frac{f_0}{2k} + \sum_{n>0} \alpha_n \cos(n\omega t - \delta_n) + \beta_n \sin(n\omega t - \delta_n), \qquad (5)\\
\alpha_n &= \frac{f_n/m}{\sqrt{((n\omega)^2 - \omega_0^2) + 4\beta^2 n^2 \omega^2}},\\
\beta_n &= \frac{g_n/m}{\sqrt{((n\omega)^2 - \omega_0^2) + 4\beta^2 n^2 \omega^2}},\\
\delta_n &= \tan^{-1}\left(\frac{2\beta n\omega}{\omega_0^2 - n^2\omega^2}\right).
\end{aligned}
$$

Finding the Coefficients

Because the forces have been applied for a long time, any non-zero damping eliminates the homogenous parts of the solution. We need then only consider the particular solution for each n.

The problem is considered solved if one can find expressions for the coefficients f_n and g_n, even though the solutions are expressed as an infinite sum. The coefficients can be extracted from the function $F(t)$ by

$$
\begin{aligned}
f_n &= \frac{2}{\tau}\int_{-\tau/2}^{\tau/2} dt\ F(t)\cos(2n\pi t/\tau), \qquad (6)\\
g_n &= \frac{2}{\tau}\int_{-\tau/2}^{\tau/2} dt\ F(t)\sin(2n\pi t/\tau).
\end{aligned}
$$

To check the consistency of these expressions and to verify Eq. (6), one can insert the expansion of $F(t)$ in Eq. (4) into the expression for the coefficients in Eq. (6) and see whether

$$
f_n = \frac{2}{\tau}\int_{-\tau/2}^{\tau/2} dt\ \left\{\frac{f_0}{2} + \sum_{m>0} f_m\cos(m\omega t) + g_m\sin(m\omega t)\right\}\cos(n\omega t).
$$

Immediately, one can throw away all the terms with g_m because they convolute an even and an odd function. The term with $f_0/2$ disappears because $\cos(n\omega t)$ is equally positive and negative over the interval and will integrate to zero. For all the terms $f_m\cos(m\omega t)$ appearing in the sum, one can use angle addition formulas to see that $\cos(m\omega t)\cos(n\omega t) = (1/2)(\cos[(m+n)\omega t] + \cos[(m-n)\omega t]$. This will integrate to zero unless $m = n$. In that case the $m = n$ term gives

$$
\int_{-\tau/2}^{\tau/2} dt\ \cos^2(m\omega t) = \frac{\tau}{2}, \qquad (7)
$$

and

$$f_n = \frac{2}{\tau}\int_{-\tau/2}^{\tau/2} dt\ f_n/2 = f_n.$$

The same method can be used to check for the consistency of g_n.

Final words on Fourier Transforms

The code here uses the Fourier series applied to a square wave signal. The code here visualizes the various approximations given by Fourier series compared with a square wave with period $T = 0.2$ (dimensionless time), width 0.1 and max value of the force $F = 2$. We see that when we increase the number of components in the Fourier series, the Fourier series approximation gets closer and closer to the square wave signal.

```
import numpy as np
import math
from scipy import signal
import matplotlib.pyplot as plt

# number of points
n = 500
# start and final times
t0 = 0.0
tn = 1.0
# Period
T =0.2
# Max value of square signal
Fmax= 2.0
# Width of signal
Width = 0.1
t = np.linspace(t0, tn, n, endpoint=False)
SqrSignal = np.zeros(n)
FourierSeriesSignal = np.zeros(n)
SqrSignal = 1.0+signal.square(2*np.pi*5*t+np.pi*Width/T)
a0 = Fmax*Width/T
FourierSeriesSignal = a0
Factor = 2.0*Fmax/np.pi
for i in range(1,500):
    FourierSeriesSignal += Factor/(i)*np.sin(np.pi*i*Width/T)*np.cos(i*t*2*np.pi/T)
plt.plot(t, SqrSignal)
plt.plot(t, FourierSeriesSignal)
plt.ylim(-0.5, 2.5)
plt.show()
```

Fourier transforms and convolution. We can use Fourier transforms in our studies of convolution as well. To see this, assume we have two functions f and g and their corresponding Fourier transforms $\hat{f}$ and $\hat{g}$. We remind the reader that the Fourier transform reads (say for the function f)

$$\hat{f}(y) = \boldsymbol{F}[f(y)] = \frac{1}{2\pi}\int_{-\infty}^{\infty} d\omega \exp -i\omega y f(\omega),$$

and similarly we have

$$\hat{g}(y) = \boldsymbol{F}[g(y)] = \frac{1}{2\pi}\int_{-\infty}^{\infty} d\omega \exp -i\omega y g(\omega).$$

The inverse Fourier transform is given by

$$\boldsymbol{F}^{-1}[g(y)] = \frac{1}{2\pi}\int_{-\infty}^{\infty} d\omega \exp i\omega y g(\omega).$$

The inverse Fourier transform of the product of the two functions $\hat{f}\hat{g}$ can be written as

$$\boldsymbol{F}^{-1}[(\hat{f}\hat{g})(x)] = \frac{1}{2\pi}\int_{-\infty}^{\infty} d\omega \exp i\omega x \hat{f}(\omega)\hat{g}(\omega).$$

We can rewrite the latter as

$$\boldsymbol{F}^{-1}[(\hat{f}\hat{g})(x)] = \int_{-\infty}^{\infty} d\omega \exp i\omega x \hat{f}(\omega)\left[\frac{1}{2\pi}\int_{-\infty}^{\infty} g(y)dy \exp -i\omega y\right] = \frac{1}{2\pi}\int_{-\infty}^{\infty} dy g(y)\int_{-\infty}^{\infty} d\omega \hat{f}(\omega)\exp i\omega(x-y),$$

which is simply

$$\boldsymbol{F}^{-1}[(\hat{f}\hat{g})(x)] = \int_{-\infty}^{\infty} dy g(y) f(x-y) = (f*g)(x),$$

the convolution of the functions f and g.

Two-dimensional Objects

We are now ready to start studying the discrete convolutions relevant for convolutional neural networks. We often use convolutions over more than one dimension at a time. If we have a two-dimensional image I as input, we can have a **filter** defined by a two-dimensional **kernel** K. This leads to an output S

$$S_(i,j) = (I*K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n).$$

Convolution is a commutatitave process, which means we can rewrite this equation as

$$S_(i,j) = (I*K)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n).$$

Normally the latter is more straightforward to implement in a machine larning library since there is less variation in the range of values of m and n.

Many deep learning libraries implement cross-correlation instead of convolution (although it is referred to s convolution)

$$S_(i,j) = (I*K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n).$$

More on Dimensionalities

In fields like signal processing (and imaging as well), one designs so-called filters. These filters are defined by the convolutions and are often hand-crafted. One may specify filters for smoothing, edge detection, frequency reshaping, and similar operations. However with neural networks the idea is to automatically learn the filters and use many of them in conjunction with non-linear operations (activation functions).

As an example consider a neural network operating on sound sequence data. Assume that we an input vector $\boldsymbol{x}$ of length $d = 10^6$. We construct then a neural network with onle hidden layer only with 10^4 nodes. This means that we will have a weight matrix with $10^4 \times 10^6 = 10^{10}$ weights to be determined, together with 10^4 biases.

Assume furthermore that we have an output layer which is meant to train whether the sound sequence represents a human voice (true) or something else (false). It means that we have only one output node. But since this output node connects to 10^4 nodes in the hidden layer, there are in total 10^4 weights to be determined for the output layer, plus one bias. In total we have

$$\text{NumberParameters} = 10^{10} + 10^4 + 10^4 + 1 \approx 10^{10},$$

that is ten billion parameters to determine.

Further Dimensionality Remarks

In today's architecture one can train such neural networks, however this is a huge number of parameters for the task at hand. In general, it is a very wasteful and inefficient use of dense matrices as parameters. Just as importantly, such trained network parameters are very specific for the type of input data on which they were trained and the network is not likely to generalize easily to variations in the input.

The main principles that justify convolutions is locality of information and repetion of patterns within the signal. Sound samples of the input in adjacent spots are much more likely to affect each other than those that are very far away. Similarly, sounds are repeated in multiple times in the signal. While slightly simplistic, reasoning about such a sound example demonstrates this. The same principles then apply to images and other similar data.

CNNs in more detail

Let assume we have an input matrix I of dimensionality 3×3 and a 2×2 filter W given by the following matrices

$$\boldsymbol{I} = \begin{bmatrix} i_{00} & i_{01} & i_{02} \\ i_{10} & i_{11} & i_{12} \\ i_{20} & i_{21} & i_{22} \end{bmatrix},$$

and

$$\boldsymbol{W} = \begin{bmatrix} w_{00} & w_{01} \\ w_{10} & w_{11} \end{bmatrix}.$$

We introduce now the hyperparameter S **stride**. Stride represents how the filter W moves the convolution process on the matrix I. We strongly recommend the repository on Arithmetic of deep learning by Dumoulin and Visin

Here we set the stride equal to $S = 1$, which means that, starting with the element i_{00}, the filter will act on 2×2 submatrices each time, starting with the upper corner and moving according to the stride value column by column.

Here we perform the operation

$$S_(i,j) = (I * W)(i,j) = \sum_m \sum_n I(i-m, j-n)W(m,n),$$

and obtain

$$\boldsymbol{S} = \begin{bmatrix} i_{00}w_{00} + i_{01}w_{01} + i_{10}w_{10} + i_{11}w_{11} & i_{01}w_{00} + i_{02}w_{01} + i_{11}w_{10} + i_{12}w_{11} \\ i_{10}w_{00} + i_{11}w_{01} + i_{20}w_{10} + i_{21}w_{11} & i_{11}w_{00} + i_{12}w_{01} + i_{21}w_{10} + i_{22}w_{11} \end{bmatrix}.$$

We can rewrite this operation in terms of a matrix-vector multiplication by defining a new vector where we flatten out the inputs as a vector $\boldsymbol{I}'$ of length 9 and a matrix $\boldsymbol{W}'$ with dimension 4×9 as

$$\boldsymbol{I}' = \begin{bmatrix} i_{00} \\ i_{01} \\ i_{02} \\ i_{10} \\ i_{11} \\ i_{12} \\ i_{20} \\ i_{21} \\ i_{22} \end{bmatrix},$$

and the new matrix

$$\boldsymbol{W}' = \begin{bmatrix} w_{00} & w_{01} & 0 & w_{10} & w_{11} & 0 & 0 & 0 & 0 \\ 0 & w_{00} & w_{01} & 0 & w_{10} & w_{11} & 0 & 0 & 0 \\ 0 & 0 & 0 & w_{00} & w_{01} & 0 & w_{10} & w_{11} & 0 \\ 0 & 0 & 0 & 0 & w_{00} & w_{01} & 0 & w_{10} & w_{11} \end{bmatrix}.$$

We see easily that performing the matrix-vector multiplication $\boldsymbol{W}'\boldsymbol{I}'$ is the same as the above convolution with stride $S = 1$, that is

$$S = (\boldsymbol{W} * \boldsymbol{I}),$$

is now given by $\boldsymbol{W}'\boldsymbol{I}'$ which is a vector of length 4 instead of the originally resulting 2×2 output matrix.

The collection of kernels/filters W defining a discrete convolution has a shape corresponding to some permutation of $(n, m, k_1, \ldots, k_N)$, where

$$
\begin{aligned}
n &\equiv \text{number of output feature maps},\\
m &\equiv \text{number of input feature maps},\\
k_j &\equiv \text{kernel size along axis } j.
\end{aligned}
$$

The following properties affect the output size o_j of a convolutional layer along axis j:

1. i_j: input size along axis j,
2. k_j: kernel/filter size along axis j,
3. stride (distance between two consecutive positions of the kernel/filter) along axis j,
4. zero padding (number of zeros concatenated at the beginning and at the end of an axis) along axis j.

For instance, the above examples shows a 2×2 kernel/filter $\boldsymbol{W}$ applied to a 3×3 input padded with a 0×0 border of zeros using 1×1 strides.

Note that strides constitute a form of **subsampling**. As an alternative to being interpreted as a measure of how much the kernel/filter is translated, strides can also be viewed as how much of the output is retained. For instance, moving the kernel by hops of two is equivalent to moving the kernel by hops of one but retaining only odd output elements.

Pooling

In addition to discrete convolutions themselves, *pooling* operations make up another important building block in CNNs. Pooling operations reduce the size of feature maps by using some function to summarize subregions, such as taking the average or the maximum value.

Pooling works by sliding a window across the input and feeding the content of the window to a *pooling function*. In some sense, pooling works very much like a discrete convolution, but replaces the linear combination described by the kernel with some other function. Poolin provides an example for average pooling, and does the same for max pooling.

The following properties affect the output size o_j of a pooling layer along axis j:

1. i_j: input size along axis j,
2. k_j: pooling window size along axis j,
3. s_j: stride (distance between two consecutive positions of the pooling window) along axis j.

The analysis of the relationship between convolutional layer properties is eased by the fact that they don't interact across axes, i.e., the choice of kernel size, stride and zero padding along axis j only affects the output size of axis j. Because of that, we will focus on the following simplified setting:

1. 2-D discrete convolutions ($N = 2$),
2. square inputs ($i_1 = i_2 = i$),
3. square kernel size ($k_1 = k_2 = k$),
4. same strides along both axes ($s_1 = s_2 = s$),
5. same zero padding along both axes ($p_1 = p_2 = p$).

This facilitates the analysis and the visualization, but keep in mind that the results outlined here also generalize to the N-D and non-square cases.

No zero padding, unit strides

The simplest case to analyze is when the kernel just slides across every position of the input (i.e., $s = 1$ and $p = 0$).

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

$$o = (i - k) + 1.$$

Zero padding, unit strides

To factor in zero padding (i.e., only restricting to $s = 1$), let's consider its effect on the effective input size: padding with p zeros changes the effective input size from i to $i + 2p$. In the general case, we can infer the following relationship

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

$$o = (i - k) + 2p + 1.$$

Half (same) padding

Having the output size be the same as the input size (i.e., $o = i$) can be a desirable property:

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

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

Full padding

While convolving a kernel generally decreases the output size with respect to the input size, sometimes the opposite is required. This can be achieved with proper zero padding:

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

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

This is sometimes referred to as full padding, because in this setting every possible partial or complete superimposition of the kernel on the input feature map is taken into account.

Pooling arithmetic

In a neural network, pooling layers provide invariance to small translations of the input. The most common kind of pooling is **max pooling**, which consists in splitting the input in (usually non-overlapping) patches and outputting the maximum value of each patch. Other kinds of pooling exist, e.g., mean or average pooling, which all share the same idea of aggregating the input locally by applying a non-linearity to the content of some patches.

Since pooling does not involve zero padding, the relationship describing the general case is as follows:

For any i, k and s,

$$o = \left\lfloor \frac{i-k}{s} \right\rfloor + 1.$$

Building convolutional neural networks in Tensorflow and Keras

As discussed above, CNNs are neural networks built from the assumption that the inputs to the network are 2D images. This is important because the number of features or pixels in images grows very fast with the image size, and an enormous number of weights and biases are needed in order to build an accurate network.

As before, we still have our input, a hidden layer and an output. What's novel about convolutional networks are the **convolutional** and **pooling** layers stacked in pairs between the input and the hidden layer. In addition, the data is no longer represented as a 2D feature matrix, instead each input is a number of 2D matrices, typically 1 for each color dimension (Red, Green, Blue).

Setting it up

It means that to represent the entire dataset of images, we require a 4D matrix or **tensor**. This tensor has the dimensions:

$$(n_{inputs}, n_{pixels,width}, n_{pixels,height}, depth).$$

The MNIST dataset again

The MNIST dataset consists of grayscale images with a pixel size of 28×28, meaning we require $28 \times 28 = 724$ weights to each neuron in the first hidden layer.

If we were to analyze images of size 128×128 we would require $128 \times 128 = 16384$ weights to each neuron. Even worse if we were dealing with color images, as most images are, we have an image matrix of size 128×128 for each color dimension (Red, Green, Blue), meaning 3 times the number of weights = 49152 are required for every single neuron in the first hidden layer.

Strong correlations

Images typically have strong local correlations, meaning that a small part of the image varies little from its neighboring regions. If for example we have an image of a blue car, we can roughly assume that a small blue part of the image is surrounded by other blue regions.

Therefore, instead of connecting every single pixel to a neuron in the first hidden layer, as we have previously done with deep neural networks, we can instead connect each neuron to a small part of the image (in all 3 RGB depth dimensions). The size of each small area is fixed, and known as a receptive.

Layers of a CNN

The layers of a convolutional neural network arrange neurons in 3D: width, height and depth. The input image is typically a square matrix of depth 3.

A **convolution** is performed on the image which outputs a 3D volume of neurons. The weights to the input are arranged in a number of 2D matrices, known as **filters**.

Each filter slides along the input image, taking the dot product between each small part of the image and the filter, in all depth dimensions. This is then passed through a non-linear function, typically the **Rectified Linear (ReLu)** function, which serves as the activation of the neurons in the first convolutional layer. This is further passed through a **pooling layer**, which reduces the size of the convolutional layer, e.g. by taking the maximum or average across some small regions, and this serves as input to the next convolutional layer.

Systematic reduction

By systematically reducing the size of the input volume, through convolution and pooling, the network should create representations of small parts of the input, and then from them assemble representations of larger areas. The final pooling layer is flattened to serve as input to a hidden layer, such that each neuron in the final pooling layer is connected to every single neuron in the hidden layer. This then serves as input to the output layer, e.g. a softmax output for classification.

Prerequisites: Collect and pre-process data

```
# import necessary packages
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets


# ensure the same random numbers appear every time
np.random.seed(0)

# display images in notebook
%matplotlib inline
plt.rcParams['figure.figsize'] = (12,12)


# download MNIST dataset
digits = datasets.load_digits()

# define inputs and labels
inputs = digits.images
labels = digits.target

# RGB images have a depth of 3
# our images are grayscale so they should have a depth of 1
inputs = inputs[:,:,:,np.newaxis]

print("inputs = (n_inputs, pixel_width, pixel_height, depth) = " + str(inputs.shape))
print("labels = (n_inputs) = " + str(labels.shape))


# choose some random images to display
n_inputs = len(inputs)
indices = np.arange(n_inputs)
random_indices = np.random.choice(indices, size=5)

for i, image in enumerate(digits.images[random_indices]):
    plt.subplot(1, 5, i+1)
    plt.axis('off')
    plt.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
    plt.title("Label: %d" % digits.target[random_indices[i]])
plt.show()
```

Importing Keras and Tensorflow

```
from tensorflow.keras import datasets, layers, models
from tensorflow.keras.layers import Input
from tensorflow.keras.models import Sequential      #This allows appending layers to existing mod
from tensorflow.keras.layers import Dense           #This allows defining the characteristics of
from tensorflow.keras import optimizers             #This allows using whichever optimiser we wan
from tensorflow.keras import regularizers           #This allows using whichever regularizer we w
from tensorflow.keras.utils import to_categorical   #This allows using categorical cross entropy
#from tensorflow.keras import Conv2D
#from tensorflow.keras import MaxPooling2D
#from tensorflow.keras import Flatten

from sklearn.model_selection import train_test_split

# representation of labels
```

```
labels = to_categorical(labels)

# split into train and test data
# one-liner from scikit-learn library
train_size = 0.8
test_size = 1 - train_size
X_train, X_test, Y_train, Y_test = train_test_split(inputs, labels, train_size=train_size,
                                                    test_size=test_size)
```

Running with Keras

```
def create_convolutional_neural_network_keras(input_shape, receptive_field,
                                              n_filters, n_neurons_connected, n_categories,
                                              eta, lmbd):
    model = Sequential()
    model.add(layers.Conv2D(n_filters, (receptive_field, receptive_field), input_shape=input_shape
              activation='relu', kernel_regularizer=regularizers.l2(lmbd)))
    model.add(layers.MaxPooling2D(pool_size=(2, 2)))
    model.add(layers.Flatten())
    model.add(layers.Dense(n_neurons_connected, activation='relu', kernel_regularizer=regularizers
    model.add(layers.Dense(n_categories, activation='softmax', kernel_regularizer=regularizers.l2(

    sgd = optimizers.SGD(lr=eta)
    model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['accuracy'])

    return model

epochs = 100
batch_size = 100
input_shape = X_train.shape[1:4]
receptive_field = 3
n_filters = 10
n_neurons_connected = 50
n_categories = 10

eta_vals = np.logspace(-5, 1, 7)
lmbd_vals = np.logspace(-5, 1, 7)
```

Final part

```
CNN_keras = np.zeros((len(eta_vals), len(lmbd_vals)), dtype=object)

for i, eta in enumerate(eta_vals):
    for j, lmbd in enumerate(lmbd_vals):
        CNN = create_convolutional_neural_network_keras(input_shape, receptive_field,
                                              n_filters, n_neurons_connected, n_categories,
                                              eta, lmbd)
        CNN.fit(X_train, Y_train, epochs=epochs, batch_size=batch_size, verbose=0)
        scores = CNN.evaluate(X_test, Y_test)

        CNN_keras[i][j] = CNN

        print("Learning rate = ", eta)
        print("Lambda = ", lmbd)
        print("Test accuracy: %.3f" % scores[1])
        print()
```

Final visualization

```
# visual representation of grid search
# uses seaborn heatmap, could probably do this in matplotlib
import seaborn as sns

sns.set()

train_accuracy = np.zeros((len(eta_vals), len(lmbd_vals)))
test_accuracy = np.zeros((len(eta_vals), len(lmbd_vals)))

for i in range(len(eta_vals)):
    for j in range(len(lmbd_vals)):
        CNN = CNN_keras[i][j]

        train_accuracy[i][j] = CNN.evaluate(X_train, Y_train)[1]
        test_accuracy[i][j] = CNN.evaluate(X_test, Y_test)[1]


fig, ax = plt.subplots(figsize = (10, 10))
sns.heatmap(train_accuracy, annot=True, ax=ax, cmap="viridis")
ax.set_title("Training Accuracy")
ax.set_ylabel("$\eta$")
ax.set_xlabel("$\lambda$")
plt.show()

fig, ax = plt.subplots(figsize = (10, 10))
sns.heatmap(test_accuracy, annot=True, ax=ax, cmap="viridis")
ax.set_title("Test Accuracy")
ax.set_ylabel("$\eta$")
ax.set_xlabel("$\lambda$")
plt.show()
```

The CIFAR01 data set

The CIFAR10 dataset contains 60,000 color images in 10 classes, with 6,000 images in each class. The dataset is divided into 50,000 training images and 10,000 testing images. The classes are mutually exclusive and there is no overlap between them.

```
import tensorflow as tf

from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt

# We import the data set
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()

# Normalize pixel values to be between 0 and 1 by dividing by 255.
train_images, test_images = train_images / 255.0, test_images / 255.0
```

Verifying the data set

To verify that the dataset looks correct, let's plot the first 25 images from the training set and display the class name below each image.

```
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']

plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    # The CIFAR labels happen to be arrays,
    # which is why you need the extra index
    plt.xlabel(class_names[train_labels[i][0]])
plt.show()
```

Set up the model

The 6 lines of code below define the convolutional base using a common pattern: a stack of Conv2D and MaxPooling2D layers.

As input, a CNN takes tensors of shape (image$_h eight, image_w idth, color_c hannels), ignoringthebatchsize.If y$

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))

# Let's display the architecture of our model so far.

model.summary()
```

You can see that the output of every Conv2D and MaxPooling2D layer is a 3D tensor of shape (height, width, channels). The width and height dimensions tend to shrink as you go deeper in the network. The number of output channels for each Conv2D layer is controlled by the first argument (e.g., 32 or 64). Typically, as the width and height shrink, you can afford (computationally) to add more output channels in each Conv2D layer.

Add Dense layers on top

To complete our model, you will feed the last output tensor from the convolutional base (of shape (4, 4, 64)) into one or more Dense layers to perform classification. Dense layers take vectors as input (which are 1D), while the current output is a 3D tensor. First, you will flatten (or unroll) the 3D output to 1D, then add one or more Dense layers on top. CIFAR has 10 output classes, so you use a final Dense layer with 10 outputs and a softmax activation.

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10))
Here's the complete architecture of our model.
```

```
model.summary()
```

As you can see, our (4, 4, 64) outputs were flattened into vectors of shape (1024) before going through two Dense layers.

Compile and train the model

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

history = model.fit(train_images, train_labels, epochs=10,
                    validation_data=(test_images, test_labels))
```

Finally, evaluate the model

```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.5, 1])
plt.legend(loc='lower right')

test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)

print(test_acc)
```