Convolutional Neural Networks with Python Quick Start Guide

Deep convolutional neural networks and computer vision applications with Keras

Viktor Ivanov

Contents

1. Introduction to Convolutions

In this chapter you will get introduced to the basic theory behind convolutional neural networks or CNNs as we will usually call them. Understanding the basic theory is of course not mandatory, but it will definitely provide you with deeper understanding and better intuition for why do things actually work. In the field of practical machine learning it's quite common to not completely understand every piece of technology or functionality you use, but still having deeper knowledge of some of the basic concepts is very useful.

1.1 What is Convolution?

In mathematics convolution is operation on two functions that produces a third function. In particular

$$f * g = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$$

The resulting integral is function of t, so if we want to know the value of $(f * g)(5)$ we have to compute $\int_{-\infty}^{\infty} f(\tau)g(5-\tau)d\tau$

Of course there is simple explanation of that otherwise complex definition. If we imagine shifting the function g over the function f then the convolution of f and g simply expresses the amount of overlap of g over f.

Don't worry if you don't understand that definition. Though there is, in my opinion, relation between mathematical convolution and convolutional neural networks, you still can become an expert in CNNs even if you don't completely understand the above definition.

Example:

Lets f and g be normal distributions respectively with mean $\mu = 0$ and standard deviation $\sigma = 0.3$ and g with mean $\mu = 1$ and standard deviation $\sigma = 0.3$

The result of the convolution $f * g$ would be

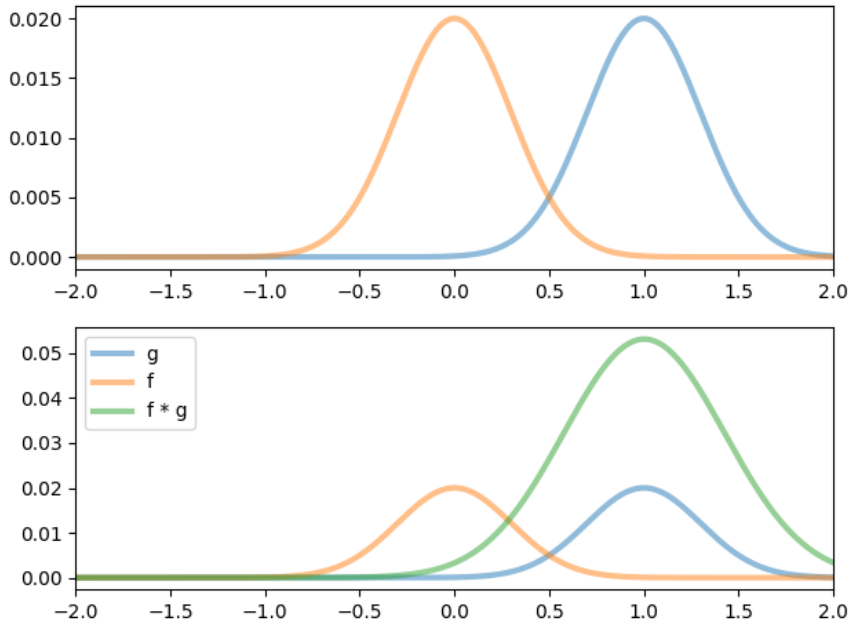


Figure 1.1: Convolution of two functions

You will later see the analogy between the mathematical definition and the machine learning term convolution.

1.2 Images as functions

I know that when you see an image of a puppy you actually think of a puppy. But from another point of view you can think of it as a function. Images can be interpreted as 2D functions of light. The image essentially represents the value of light at each point on the camera's matrix. That's why the sun on a image having brighter pixel values than a shadow is the same as saying that the function of light at the point where the sun is on the image has higher value than the point where the shadow is.

Let's consider that image of this cute puppy

Figure 1.2: Cute dog

for the purpose of the example I chose a simple cartoon instead of real image, because this cartoon has much more flat areas than a picture and the function graphic would be much more observable.

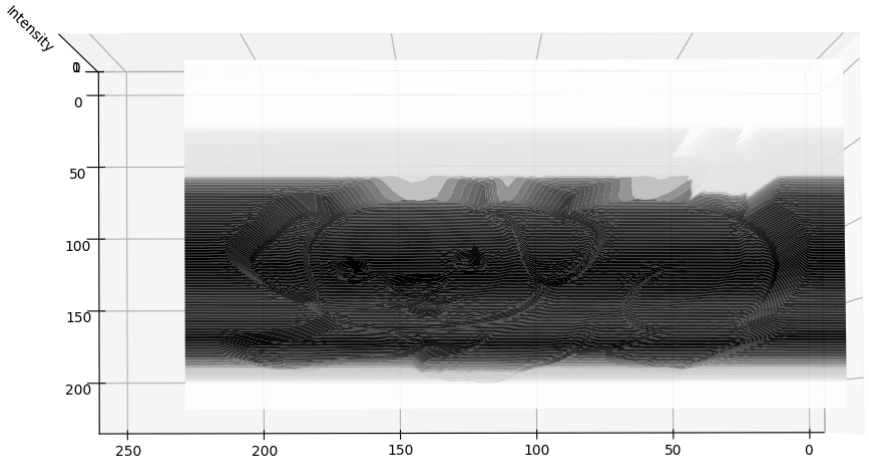


Figure 1.3: The dog image as function in the 3D space

On the X and the Y axis we have the 2D pixel grid and on the Z axis we have the functional value which in the case is the pixel value, or the brightness of the pixel. In the case of real photo we can consider this as the amount of light the device has captured and accumulated into one pixel.

We can then express the image as the following 2 dimensional function

$$I_2(x, y) = z$$

Which is simply a mathematical 2D function not any different from the function $f(x, y) = x^2 + y^2$

So far we know what a convolution is from a mathematical perspective, how to convolve two functions and that we can think of images as functions. Good enough to get into practice.

1.3 Example: Object detection

Let's consider the following image:

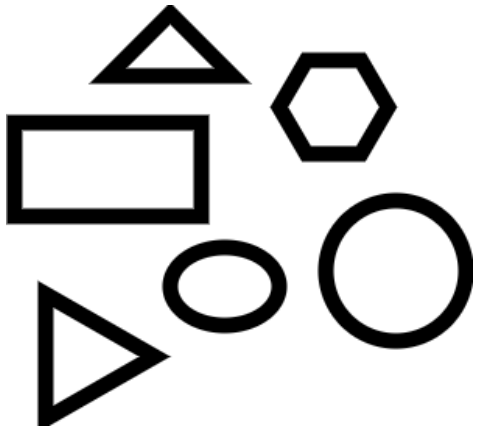


Figure 1.4: Image with different shapes

Given a pattern like:

Figure 1.5: Pattern to detect

How would we detect if there is an object on the upper image matching the pattern?

Now we will see the machine learning perspective of convolution.

Let's start with **dot product**. What is dot product? Dot product is simply multiplication and addition. By definition given two vectors $x = [x_1, x_2, ..., x_n]$ and $y = [y_1, y_2, ..., y_n]$ their dot product is defined as

$$x \cdot y = \sum_{i=1}^{n} x_i y_i$$

Which after multiplying all the pairs and summing them up is simply a number. So far so good, but what does dot product have to do with images and matrices? From mathematical point of view vectors can have any non-negative number of dimensions (including zero - numbers are actually

0 dimensional vectors). So it would be totally fine to actually extend the definition of dot product to 2D arrays, which helps us apply it for images. Dot product of two 2D arrays X and Y, which are usually called matrices would be:

$$X = \begin{pmatrix} x_{1,1} & \dots & x_{1,m} \\ & \cdot & \\ & \cdot & \\ & \cdot & \\ x_{n,1} & \dots & x_{n,m} \end{pmatrix}, Y = \begin{pmatrix} y_{1,1} & \dots & y_{1,m} \\ & \cdot & \\ & \cdot & \\ & \cdot & \\ y_{n,1} & \dots & y_{n,m} \end{pmatrix}$$

Note that the dimensions of the two matrices X and Y are both $[n, m]$

$$X \cdot Y = \sum_{i=1}^{n} \sum_{j=1}^{m} x_{i,j} y_{i,j}$$

So to compute the dot product of the two matrices we just have to multiply them element-wise and sum the result.

Here's a simple python function computing the dot product between X and Y

Listing 1.1: Function computing dot product of X and Y

```
def dot_product(X, Y):
    result = 0

    for i in range(n):
        for j in range(m):
            result += X[i][j] * Y[i][j]

    return result
```

Now let's see what does convolution mean in terms of machine learning. Well it's simply dot product over matrices. Let's see how to apply this to our initial problem of detecting the pattern of Figure 1.5 in Figure 1.4. Let's convolve Figure 1.5 over Figure 1.4. Just as we convolved two mathematical functions by sliding one over the another we will slide Figure 1.5 over Figure 1.4 and compute dot product for the overlapping area. Before trying to picture that, let's try to imagine the output of the process.

1. The output will still be image.
2. It will be smaller or equal to the initial image being convolved. *(In which case the result will have the same shape as the input ?)*

3. Considering that we multiply and add images with non-negative pixel values, the resulting image will also be with non-negative pixels.

Example:
Let's have smaller in size grayscale image I and pattern p to detect in it:

$$I = \begin{pmatrix} 255 & 185 & 133 & 121 \\ 155 & 144 & 188 & 201 \\ 111 & 88 & 171 & 211 \\ 110 & 194 & 143 & 101 \end{pmatrix}, p = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Of course this would not be a scale of a real image, but for the purpose of the explanation let's imagine this is 4×4 image and a chess pattern p of size 2×2.

The first iteration the pattern "covers" the upper left corner.

$$\begin{pmatrix} 255_{*1} & 185_{*0} & 133 & 121 \\ 155_{*0} & 144_{*1} & 188 & 201 \\ 111 & 88 & 171 & 211 \\ 110 & 194 & 143 & 101 \end{pmatrix}$$

The first element of the result will be:

$$r_{0,0} = (255 * 1) + (185 * 0) + (155 * 0) + (144 * 1)$$

to compute the second element of the first row $r_{0,1}$ we move the pattern one step to the right.

$$\begin{pmatrix} 255 & 185_{*1} & 133_{*0} & 121 \\ 155 & 144_{*0} & 188_{*1} & 201 \\ 111 & 88 & 171 & 211 \\ 110 & 194 & 143 & 101 \end{pmatrix}$$

So $r_{0,1}$ will be:

$$r_{0,1} = (185 * 1) + (133 * 0) + (144 * 0) + (188 * 1)$$

and so on:

$$\begin{pmatrix} 255 & 185 & 133_{*1} & 121_{*0} \\ 155 & 144 & 188_{*0} & 201_{*1} \\ 111 & 88 & 171 & 211 \\ 110 & 194 & 143 & 101 \end{pmatrix}$$

$$\begin{pmatrix} 255 & 185 & 133 & 121 \\ 155_{*1} & 144_{*0} & 188 & 201 \\ 111_{*0} & 88_{*1} & 171 & 211 \\ 110 & 194 & 143 & 101 \end{pmatrix}$$

...

$$\begin{pmatrix} 255 & 185 & 133 & 121 \\ 155 & 144 & 188 & 201 \\ 111 & 88 & 171_{*1} & 211_{*0} \\ 110 & 194 & 143_{*0} & 101_{*1} \end{pmatrix}$$

and the final result will be:

$$R = \begin{pmatrix} 399 & 373 & 334 \\ 243 & 315 & 399 \\ 305 & 231 & 272 \end{pmatrix}$$

Let's again go back to Figure 1.4 and Figure 1.5. Let's convolve Figure 1.5 over Figure 1.4. The result is going to be the following image:

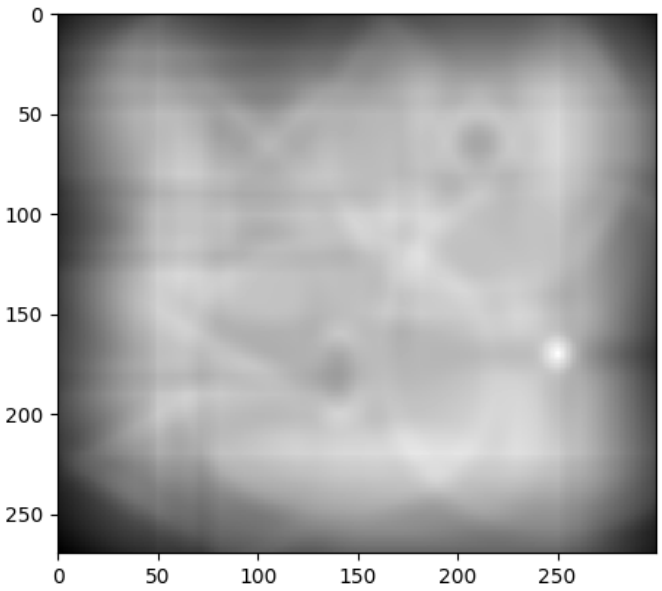


Figure 1.6: Result of convolving Figures 1.4 and 1.5

Notice the bright light dot.
This is actually the center of the pattern we are looking for!

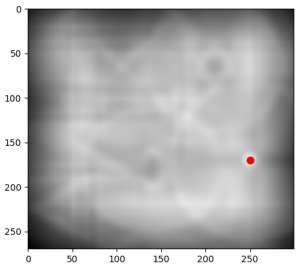


Figure 1.7: Global optimum

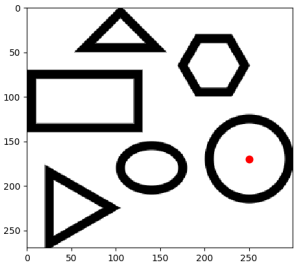


Figure 1.8: Center of the pattern we were looking for

Okay, but how does it work?

Let's consider our circle - the pattern made of 0s and 1s. For simplicity we consider grayscale image, but the same approach applies for RGB as well.

During the convolution the pattern covers different parts of the image as we are sliding it over it as shown below.

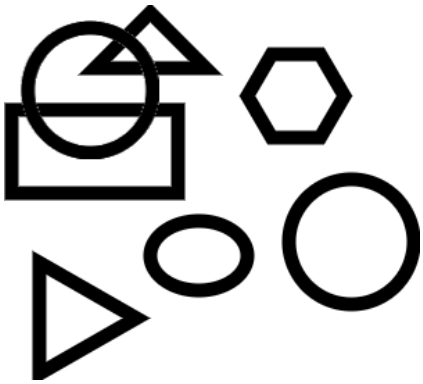


Figure 1.9: Single step of the convolution between Figures 4 and 5

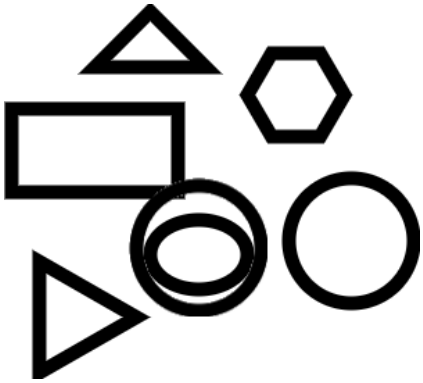


Figure 1.10: Another step of the convolution between Figures 4 and 5

The result of sliding the circle pattern over the image with the figures and computing the dot product with the overlapping area is a number which expresses "the amount of similarity" between the pattern and the overlapped area. You see that if we move the pattern with a pixel to any direction the overlapping area is still going to be almost the same with slight increase, or decrease. That's why the bright dot is not exactly of size 1 pixel, but instead it gets darker at the corners, because the pattern gets further from the circle itself and the perfect match which occurs when the two circles overlap exactly.

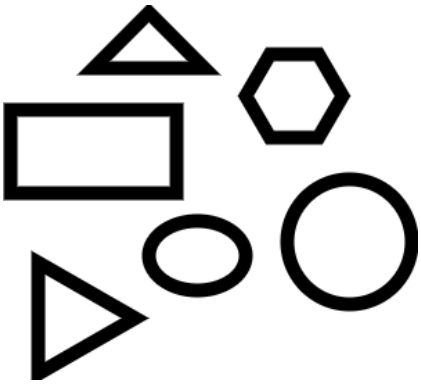


Figure 1.11: Perfect match. The pattern circle exactly overlaps with the circle on the image

When the pattern matches the circle from the image with the figures we get the highest score. That's how we can detect the circle, or any other figure. Again the same approach applies to objects like cars, humans, eyes, bananas, letters and so on ...

But still why? Why does this approach work? It's not that obvious (at least it wasn't to me) why this actually works. So again let's take the approach which always helped me to understand math, no matter how complex or abstract it was. Take an example and go through the calculations until they *"become a part of your DNA"* as a friend of mine says.

Let's consider the following small image with two diagonal lines:

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 255 & 0 & 255 & 0 & 0 \\ 0 & 255 & 0 & 0 & 0 & 255 & 0 \\ 255 & 0 & 0 & 0 & 0 & 0 & 255 \end{matrix}$$

Let's search for the following pattern:

$$\begin{matrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{matrix}$$

Let's calculate the dot product for a few of the overlapping areas, for example:

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 255 \\ 0 & 255 & 0 \end{pmatrix} * \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} = 0$$

The pattern does not match the subimage at all. That's why we got 0.

$$\begin{pmatrix} 255 & 0 & 0 \\ 0 & 255 & 0 \\ 0 & 0 & 255 \end{pmatrix} * \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} = 255$$

On this subimage we found better match of our pattern. Let's continue to see if we can find better match.

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 255 & 0 \\ 255 & 0 & 0 \end{pmatrix} * \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} = 510$$

We found even better match. If we compute the result

$$R = \begin{pmatrix} 0 & 510 & 0 & 255 & 0 \\ 765 & 0 & 255 & 0 & 255 \end{pmatrix}$$

We will see that the best result is the following:

$$\begin{pmatrix} 0 & 0 & 255 \\ 0 & 255 & 0 \\ 255 & 0 & 0 \end{pmatrix} * \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} = 765$$

Indeed we found match of the pattern. We see that the there could be some local optima where the pattern is partially matched. Also there might not always be a perfect match, so in this case we would take the closest to perfect match.

Hopefully so far you have pretty good intuition on how does convolution work in both mathematical and machine learning perspective.

What are the limitations of this "sliding pattern" approach? Usually, when we are doing object detection we do not have fixed size of the objects. We also expect our model to be capable of classifying dogs in all sizes, shapes and rotations, so no matter if we take a picture of our dog while it's running far away in the park, or it is sleeping on the couch and grandma takes one of her really close shots. But what will happen if we have a picture-pattern of our dog and try to match it on images later in future?

Let's consider this in terms of the previous example with the figures. Let's all the figures get bigger and again search for the circle (without changing it's size).

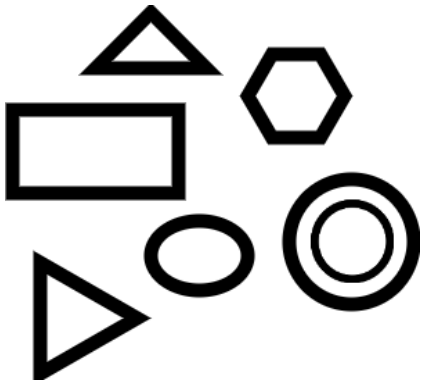


Figure 1.12: The pattern not matching the circle

There is no chance for the pattern to match the circle better then the other figures on the image. For example the hexagon now becomes the best match in the image.

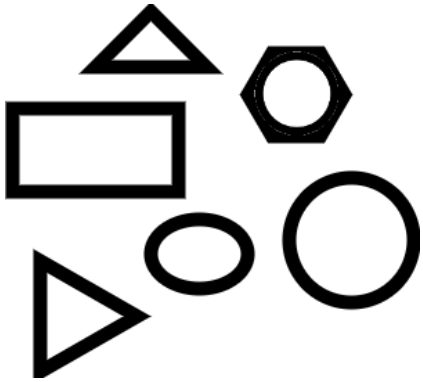


Figure 1.13: The pattern matching better the hexagon

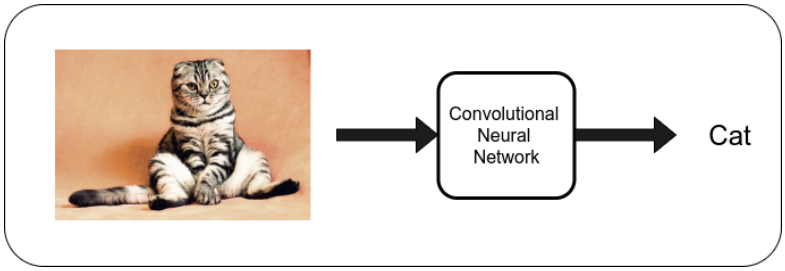
This approach is really sensitive to almost any changes of the input, but still the concept of these type of patterns is somehow elegant and powerful. Though the limitations, it provides a way to search of arbitrary object by just encoding it into matrix!

What if modify the approach a bit so it benefits more from the concept of machine learning? What if we don't encode the pattern, but instead let a model learn it? That's exactly what a convolutional neural network would do!

2. Introduction to Convolutional Neural Networks

2.1 What is a CNN?

In Deep Learning CNNs are powerful class of neural networks commonly used for processing visual data. A lot effort has been put in the last years to develop some state of the art architectures which prove that CNNs are the most powerful tool for processing visual data. CNNs are applied to wide range of problems including Object classification, detection, recognition, face detection and recognition, image segmentation and many more, some of which we will conver in this book as examples.



The building block of Convolutional neural network is the filter or also called kernel. As said at the end of the previous chapter filters are just vectors of numbers, which can also be interpreted as images. A CNNs is made of layers and each layer is made of several filters. When building a CNN we don't hardcode the values of the filters, but instead we let the network learn the best filters which help the model the solve the problem it is expected to. It is actually really similar to Multilayer perceptron, but instead of neurons the building block is convolutional filter or just filter. Similar to Multilayer perceptron we have an input which in the case of CNNs is usually an image which we propagate through the parameters of the model to the end where we get the output of the model. In the case of classification we will expect a

label of class.

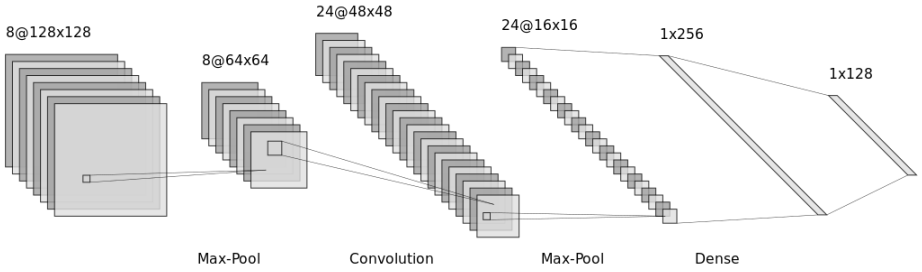


Figure 2.1: Diagram of CNNs architecture

2.2 Build your first convolutional neural network

Now let's get introduced to Keras. Keras is high level neural networks API running on top of CNTK, Theano or Google's TensorFlow. Though it's higher level API you are not going to hit any limitations when using Keras instead of TensorFlow.

We will start with the "hello world" dataset in the field of CNNs - the MNIST dataset. The data consists of $70,000$ images of handwritten digits of size 28×28.

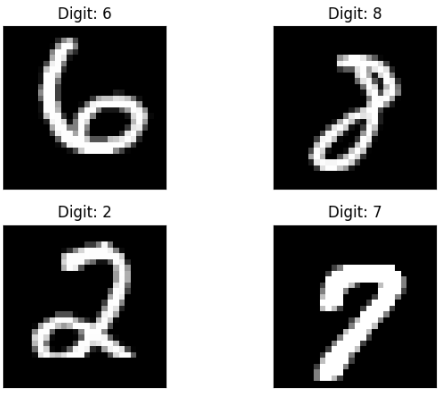


Figure 2.2: MNIST dataset sample

Let's first see how we create a CNN model in Keras.

Listing 2.1: CNN in Keras

```
model = Sequential()
model.add(Conv2D(filters=32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=(28, 28)))
model.add(Conv2D(filters=64,
                 kernel_size=(3, 3),
                 activation='relu'))
model.add(Flatten())
model.add(Dense(units=128, activation='relu'))
model.add(Dense(units=10, activation='softmax'))

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

Let's now dissect this piece of code line by line.

2.3 Sequential

Basically Keras provides two model APIs: **Sequential and Functional.** We will start with the simpler, but more limited one: The Sequential.

The Sequential API allows you to create models layer by layer and requires to provide just input shape to the model. The model then infers the input size for the next layers. That's why we pass the parameter *input_shape=(28, 28)* to the first layer.

The sequential API is considered to be simpler, but it comes with several limitations. Sequential models cannot share layers and cannot have multiple inputs or outputs. Though this limitation is hard to be hit you will sometimes implement more exotic architectures which require multiple inputs, or outputs.

The instance of the Sequential has *.add()* method which takes arbitrary Keras Layer instance.

2.3.1 Conv2D layer

2D convolution layer. This layer performs convolution on the input to produce multidimensional output vector. If provided it applies activation func-

tion on the output. The activation functions in the above example are the *relu* and *softmax*.

The most common parameters you will use with this layer are:

- filters

  Number of convolutional filters per layer. These are the trainable parameters which we hardcoded in chapter 1. For example the circle pattern we searched in the image of figures will be one of this filters, but instead of circle it may converge to any pattern which helps the network to solve the problem it is trained to. The topic of filters interpretation and visualization is becoming more and more popular as scientists are trying to understand deep models better. Later in the book you will see examples of filters from different layers plotted and analysed.

- kernel_size

  This is the size of each of the filters the layer has. The above example of CNN has 32 filters of size 3×3 for layer 1. Usually when one trains CNN they do not train one large in size filter as the one we used in chapter 1, but instead trains lots of small filters. One way to decide the size of your filters is the size of the details in your input images, that you think would help the network to perform better, but usually you will find that most architectures go with filters of size 3×3, 5×5, 7×7. I know that probably you think none of these sizes can encode a whole eye of a human in picture of size 480×640 for example, but still networks manage to find a way to encode information in such a way that down the layers they capture even more than just a single eye. Again you will see examples of filters visualized later in the book.

- activation

  The activation is exactly what activation means in terms of machine learning. After we convolve the input image with each of the filters we pass the result through the activation function to get the final output of the layer. The purpose of this is to achieve nonlinearity just as in Multilayer Perceptron. Do not get confused that instead of number the layer outputs a multidimensional image. The way to pass it through the activation is to actually pass each pixel of the image one by one. Keras provides wide range of activations most common of which:

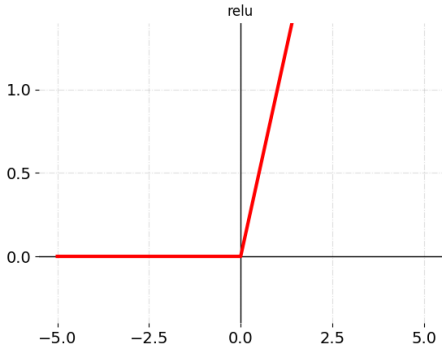


Figure 2.3: relu

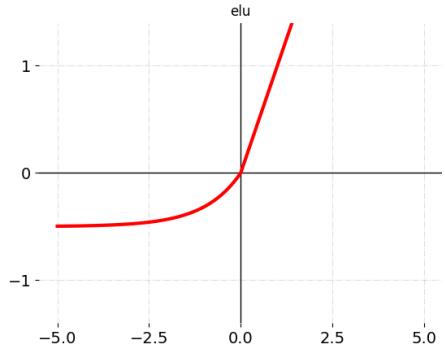


Figure 2.4: elu

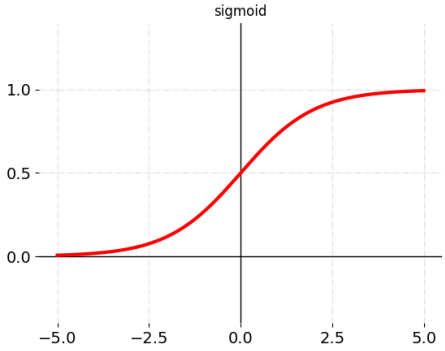


Figure 2.5: sigmoid

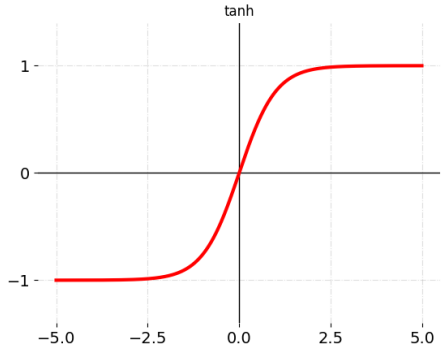


Figure 2.6: tanh

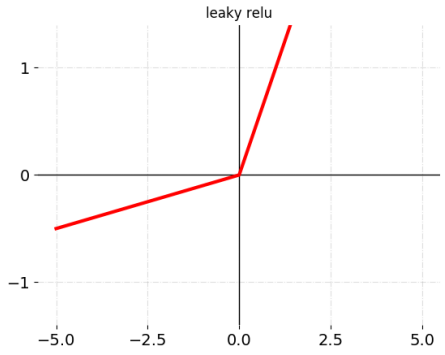


Figure 2.7: leaky relu

Remember that if you need you can always provide a custom activation function.

- input_shape

  As mentioned above, when talking about the Sequential API, this parameter must be passed to the first layer of the network. Keep in mind that, because we usually train on batches Keras requires the first dimension to be the number of images, so though your training set will be of shape $(1000, 128, 128, 3)$ where each image from the dataset is of size $(128, 128, 3)$ and the dataset itself is made of 1000 images you will provide as input shape only $(128, 128, 3)$ and Keras will infer that the actual input will be of size $(16, 128, 128, 3)$ if you train with batch size of 16.

- stride

You already know how to apply convolution over an image with given filter. You simply slide the filter over the image and compute the dot product of the overlapping area. Well, this is actually a convolution with stride 1×1. Stride is a parameter that provides you with the opportunity to set the set the size of the step across axis. When you slide the filter over the image you can take arbitrary big stride step and stride steps can be different across horizontal and vertical axis if this makes sense for your architecture. Sometimes, when you have really detailed input image which you cannot downsize, because you want to preserve the details as much as possible, but also do not want to end up with a model having 500M parameters you can take more aggressive approach and use bigger filters with bigger stride. This will result in a network that downsamples it's input faster with each layer. Here are the formulas which help you easily calculate the output shape of your layer.

$$O_w = \frac{W - F_w + 2P}{S_w} + 1$$

$$O_h = \frac{H - F_h + 2P}{S_h} + 1$$

where W, H is the input width and height, F_w, F_h is the filter width and height, S_w, S_h is the horizontal and vertical stride and P is the padding, which is described below.

You can easily see that if you increase the filter width or height with constant this reduces the output size with the same constant, while if you increase the stide with 2 this reduces the output by factor of 2.

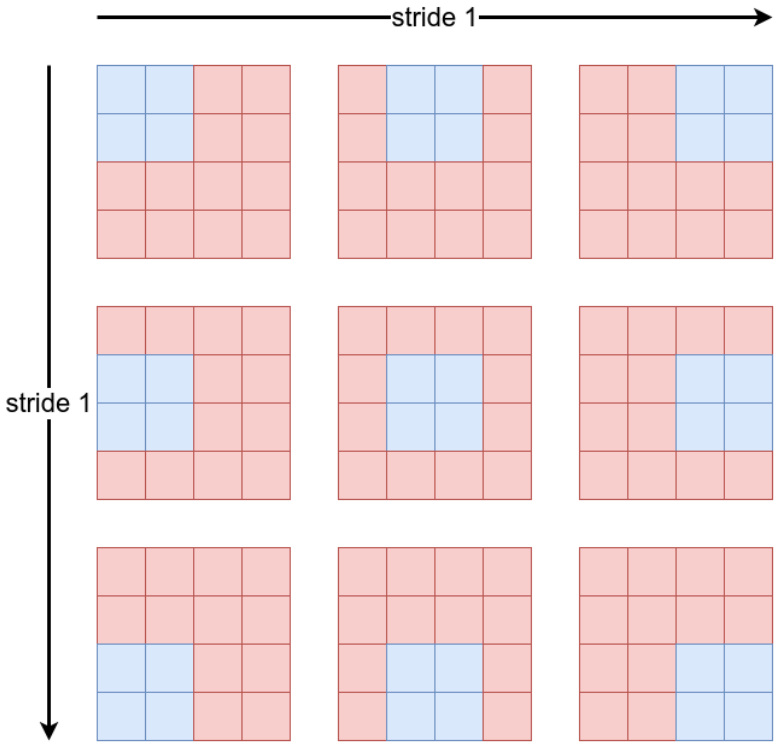


Figure 2.8: Applying convolution with stride 1x1

Here's the same example but with horizontal and vertical stride of 2, instead 1.

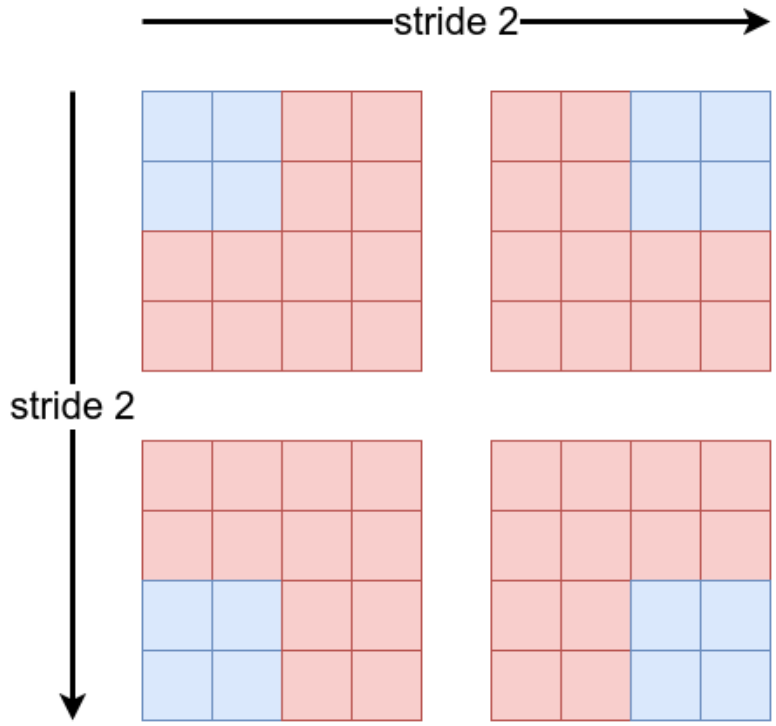


Figure 2.9: Applying convolution with stride 2x2

As you see both fig 2.8 and fig 2.9 we have input of size 4×4 and filter of size 2×2, but the number of dot products 9 and 4 respectively. Stride also increases performance of your model while still extracting information.

- padding

Padding is another quite simple operation that in essence is just adding extra pixels at the corners of the input image. The values of these pixels can vary, but usually there are several common approaches like repeating nearest value, mirroring values, padding with zeros, or another common case in CNNs is to pad with $-\infty$ in case of MaxPooling layer following.

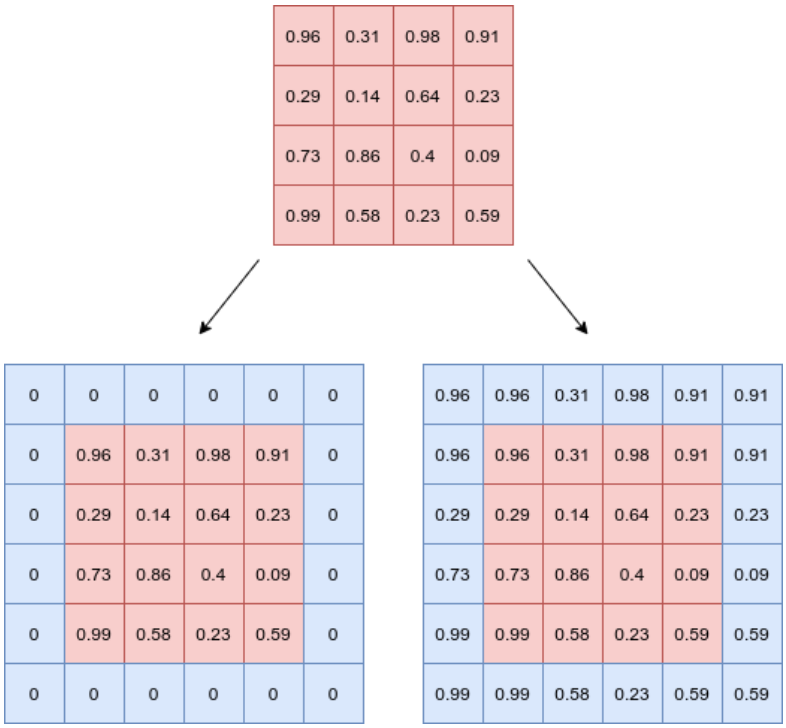


Figure 2.10: Example of zero and repetitive padding.

Though when we add padding we usually talk about it in terms of number of pixels by each side, Keras' way is not to provide the option to set padding in terms of pixels, but instead choose one of the two options:

padding="valid"

This is the default value. In essence this means do not add padding. The result is as you would expect it after applying the convolution. The output image is with $\frac{kernel\_size}{2}$ smaller on each side of the input.

The other option is:

padding="same"

What this does is simply pad the input with $\frac{kernel\_size}{2}$ pixels on each side, so the output has the *same* size as the input.

2.3.2 Flatten

This layer just flattens the input and returns it as 1D vector.

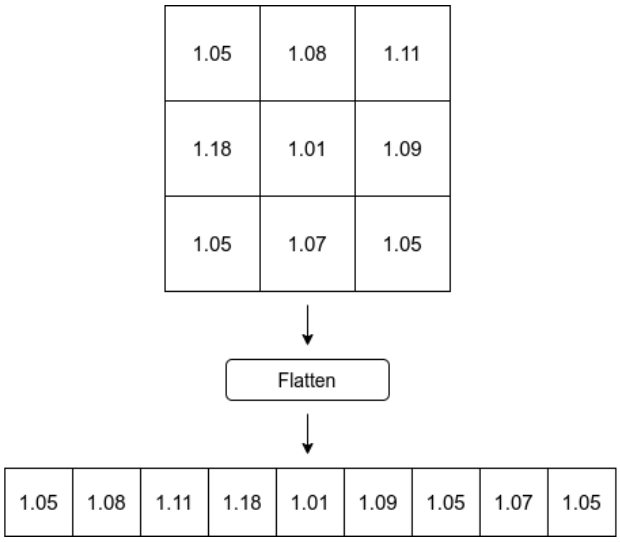


Figure 2.11: Flattening 3x3 filter(kernel)

As you already know CNNs are mostly used to process visual data. Let's consider the case of the MNIST data. We have 28×28 pixels image which we want to feed to our model and produce output a class value in the range $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$. Usually in machine learning when we solve such classification problems we represent the output as N dimensional vector where N is the number of classes. In the MNIST case $N = 10$. But we know that the output of convolution over image is still image. How will we end up with 10 dimensional vector? That's why we use flattening. Though it's really simple and non-trainable layer it's really effective and widely used in combination with CNNs.

The *.compile* method basically initializes the model and configures it for training. It also takes as parameters the loss function for the neural network and the optimizing algorithm. Keras provides wide range of both loss functions and optimizers, so you rarely have to worry about implementing them on your own. Probably most often you will implement your own loss function which describes better what is actually important for you and what you want you model to emphasize on. For now we will keep it simple and use the losses that Keras provides, but keep in mind you can also provide your own loss functions!

Compiling the model gives us the following model summary:

Listing 2.2: Summary of the compiled example model

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 24, 24, 64)        18496
_________________________________________________________________
flatten_1 (Flatten)          (None, 36864)             0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               4718720
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1290
=================================================================
Total params: 4,738,826
Trainable params: 4,738,826
Non-trainable params: 0
_________________________________________________________________
```

The **None**s in Output Shape just mean that it can accept arbitrary number of arrays of shape $26 \times 26 \times 32$. This allows the network the consume any number of images in a batch.

How does an image pass through a CNN?

Let's break into steps the process of image passing through a CNN.

If you have some experience with Multilayer Perceptrons you probably know how vectors of features pass through a network, but I consider it important to have the intuition so I will go through an example anyway.

Let's consider as an example the MNIST dataset and the above architecture.

We have the input image of size 28×28. The first layer of the CNN is of size $3 \times 3 \times 32$. To compute the result of the first layer we simply apply the well known to us convolution between each of the 32 filters and the input image. We take the first filter of size 3×3 and slide it over the input, computing the dot product and save the result in output image of size 26×26. Then we take the second filter and apply the same operation. We do this 32 times for all the kernels. The result is 32 images of size 26×26. But instead of considering this result as 32 separate images we will stack them

into one $3D$ array of size $26 \times 26 \times 32$. You can think of this as one 26×26 image with 32 channels, or just $3D$ array - whichever is easier for you. Then we apply the activation function. You probably wonder how do we apply a function to something that big as this array? We simply pass each of the numbers in the array through the function and save the value. The result is the output of the first layer of the network.

Not that hard, isn't it?

We apply absolutely the same procedure to compute the output of the second layer. The result will be of size $24 \times 24 \times 64$, because again the filters are of size 3×3 and the number of filters is 64.

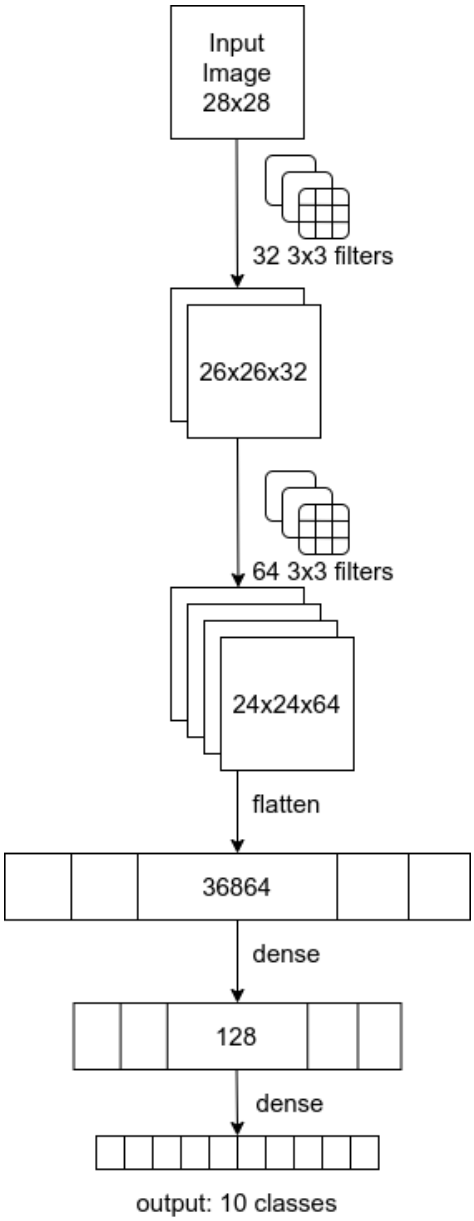


Figure 2.12: Layers of the CNN example

You maybe notice that what I call layer is actually the intermediate operation between an input and output? Common misconception is that a layer is the input or the output, but actually it makes more sense to consider the operation that network applies as a layer, instead of the input or the output of that operation.

Then we flatten the $24 \times 24 \times 64$ $3D$ array to get $1D$ array of size 36864. You already know how flatten works.

2.3.3 Dense layers

The next layer is **Dense** layer of size 128. This means that we have dense connection between the flatten layer output and this layer. Dense connection is a connection that each unit is connected to each unit from the other layer.

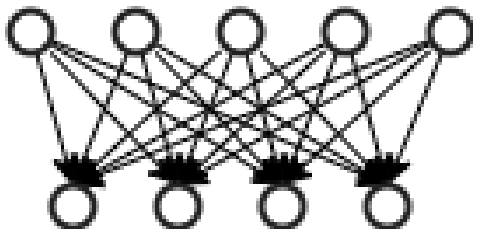


Figure 2.13: Densely connected layers

The last layer is the output of the network. It is vector of size 10. Exactly as much as as our output classes.

If you look at the architecture you will notice that the CNN actually consists of two *stacked* networks. One made of convolutional layers and one of dense layers. They are glued with the **flatten** layer mentioned above. That gives pretty good intuition on why in terms of optimization we can consider CNNs as non-linear approximators just as Multilayer Perceptrons. The only difference is that they are designed to work on input images, but from mathematical perspective they do not have more computational power than Multilayer Perceptron. Recurrent Neural networks, on the other side, are Turing complete[1], so they have much more computational power than CNNs and MLPs which can approximate arbitrary function, but cannot simulate whole program, which is the definition of Turing Completeness.

That's pretty much everything you need to know to build that simple neural network. Before explaining the actual process of training the neural network let's discuss a few more details which are **very** important.

2.3.4 ReLU

Note that the activation function of the last layer is not *relu*, but *softmax*. Why is that? Let's start with what is Rectified linear unit[2] or just ReLU. The last couple of years this is the most common activation function, because

of it's simplicity and effectiveness. It has proved to be very effective dealing with exploding / vanishing gradients.

ReLU is believed to simulate the activation function in our brain which neuroscientists claim to filter the negative values and to propagate only positive values without magnifying them. That's what exactly the ReLU does.

$$ReLU(x) = max(0, x)$$

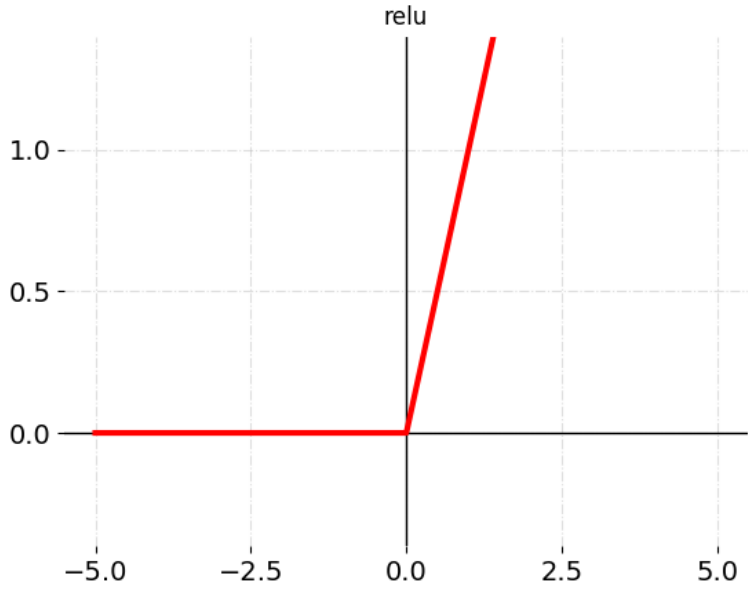


Figure 2.14: Rectified linear unit (ReLU).

2.3.5 Softmax

Why we use softmax on the end instead of ReLU? We can use ReLU! The result will be output like this one for, example:

$$[-0.113, 1.24, 12.1, 8.401, -0.999, -3.1144, 0.101, 4.0, 2.909, 5.8]$$

We can then represent our labels in such a way that we have $+\infty$ on the correct position and $-\infty$ on all other positions. So for example the label for the digit 2 will be:

$$[-\infty, -\infty, +\infty, -\infty, -\infty, -\infty, -\infty, -\infty, -\infty, -\infty]$$

In that case considering the above output we see that 12.1 is the highest value so we can interpret this output as the networks has predicted the digit 2.

Though this approach is theoretically acceptable it's not that convenient from practical perspective. In deep learning, especially in really deep architectures, we want to avoid huge numbers which can lead to exploding gradients.

Softmax is really convenient function that takes a vector of size n and maps it to a n dimensional probability distribution proportional to the input space. Explained in simple words it would map the input 10 dimensional vector into new 10 dimensional vector that is proportional to the initial one, but sums to 1 and consists of non-negative elements which we can interpret as probabilities.

$$\sigma(v) = \frac{e^{v_i}}{\sum_{j=1}^{n} e^{v_j}}, i = 1, ..., n$$

For example the above 10 dimensional vector would become the following array:

$$[0, 0, 0.9737, 0.0241, 0, 0, 0, 0.0003, 0.0001, 0.0018]$$

So we can say that our model has 0.9737% confidence that the digit on the input is the digit 2.

2.4 Regularization

When a machine learning model performs way better on the training set compared to the test set we say that the models overfits the training data. We do not want our models to overfit, instead we want them to generalize better, so when we feed them with data we have never seen before they still perform as good as we have evaluated them offline.

One of the techniques to fight overfitting is **regularization**. Overfitting restricts the model in such a way that prevents it to overfit the training data. For example in the case of linear regression, common regularization technique is to add the weights of the model to the loss function. Smaller in absolute value weights somehow correspond to simpler model and on the opposite bigger in absolute value weights correspond to more complex model, so restricting the weights keeps our model simpler.

Let's consider the common loss function

$$L(X, y) = \sum_{i=1}^{n} (y_i - f(X_i))^2$$

Where

$$f(X) = a_1 X_1 + a_2 X_2 + ... + a_n X_n + b$$

is our linear model with weights $f[a_1, a_2, ..., a_n, b]$.

Regularizing the model with adding the weights to the loss function would result in the following loss function:

$$L(X, y) = \sum_{i=1}^{n} (y_i - f(X_i))^2 + \sum_{i=1}^{n} a_i + b$$

So the bigger the weights get the complex the model gets, but also the bigger the value of the loss function becomes. Having an optimization algorithm to minimize the value of the loss function results in optimal values for the weights so they are big enough in absolute value to generalize, but yet small enough to keep the loss function value smaller.

Probably you wonder how do we apply this to CNNs? You probably saw that the model we built has more than $4M$ parameters so it would be really tedious to sum them on each input propagation. It would also be hard to keep the result in 32 or even 64 bits. Keep in mind that this is relatively small model. Some of the state of the art architectures have more than several hundred million parameters.

There are other more commonly used regularization techniques for CNNs.

2.4.1 Dropout

Dropout is probably the simplest method for regularization. What it essentially does is to set to 0 an unit (drop it) with probability rate $r \in [0, 1]$. Dropout is only applied during the training period. After the model is trained and we deploy it to production we do not apply it anymore. This is simply achieved by setting the drop rate to 0.

In Keras it is represented as regularization **layer**.

```
keras.layers.Dropout(rate)
```

When training the network we have to account every single operation we apply on the forward step when we propagate back, otherwise we would be computing the gradients for some other function, instead of the one our model is computing. On the forward propagation some of the weights are dropped with probability rate and the ones that are not dropped are scaled by $\frac{1}{1-rate}$*. On*

the backward step the zeroed units' gradients are also zeroed out and similar to the forward step the other units' gradients are scaled up with $\frac{1}{1-rate}$.

The intuition behind dropout is the following: Consider our big model as one powerful approximator that relies on it's weights and input. One way to cope with overfitting would be to reduce it's size and duplicate it several times to create ensemble and then avarage the result of each classifier in the ensemble. This will work, but this is pretty computationally ineffective. Instead of reducing it's size and computational power we will train several models simultaneously. Dropping some of the weights and the inputs we don't allow the model to rely that much on single feature or weight. We instead force it to "distribute knowledge" through all the weights so even if we drop some of them it can still come with reasonable prediction. On the other hand dropping some of the weights results in model with different number of weights and features, which we can consider as weaker estimator. After the training process we have trained many estimators, each of which was a result of the initial model after dropping some of the features and weights.

Dropout can be applied to the input and the middle layers, but not the output layer.

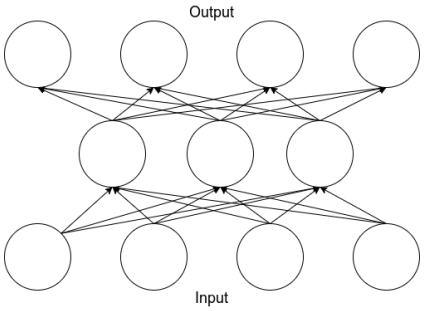


Figure 2.15: Before dropout

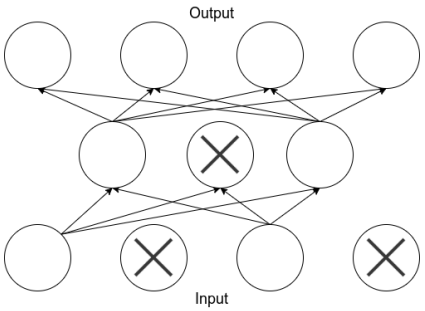


Figure 2.16: After dropout

Though it is usually not recommended in the case of CNNs[3].

2.4.2 Max-Pooling

Max-pooling is another regularization technique which is very commonly used in CNNs.

Again in Keras it is represented as regularization layer.

```
keras.layers.MaxPooling2D()
```

Max-pooling is simple operation which "pools" the maximum of several numbers. Let's have for example input image of size 100×100 and a convolutional layer of size $3 \times 3 \times 1$, which mean 1 filter of size 3×3. After passing the input image through the convolutional layer the output would be of size 98×98. Let's now have **MaxPooling2D** layer with pool_size=2×2. You can think of the pooling layer as a filter, just like the convolutional one, but instead of the dot product we will perform *max* operation over the *covered* pixels.

The output of the max-pooling layer will be tensor of size 49×49.

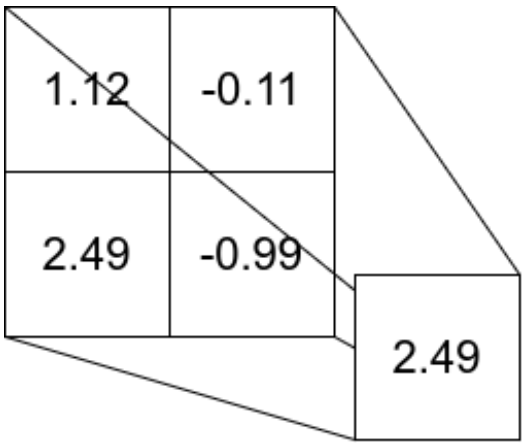


Figure 2.17: the Max-Pooling operation.

Potential interpretation of this operation is the following: Just as the ReLU function, which we said propagates only the positive numbers, which we interpret as information, the Max-Pooling propagates only the strongest signal from each layer.

It is not that obvious how do we train a CNNs with this layers in-between. When we apply the max-pooling while propagating the input forward we can think of this operation as zeroing out all the numbers except the biggest one. Then it becomes more obvious how we implement the backpropagation for this layer. When we propagate the gradients back we just replace the gradients of all the elements except the max one with zeros.

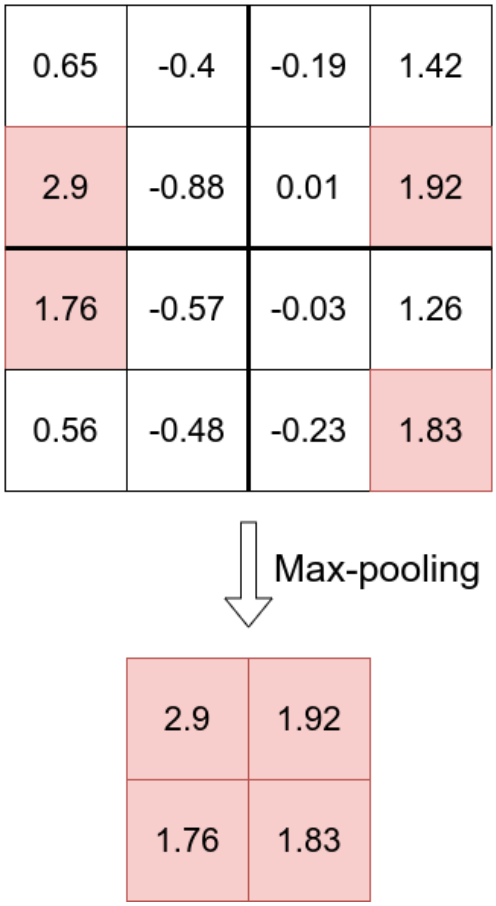


Figure 2.18: Max-pooling example.

As you can see max-pooling reduces the size of the tensors much more than convolution. In addition to that max-pooling is really strong and effective regularizer. It also helps to reduce the size of our neural network. If you have higher resolution input images which you cannot downsize, because you want to preserve the details you can apply max-poolings to reduce the size of the network faster, so it does not explode in too many parameters, which will make the training process harder. Instead this may help the network to encode the input and propagate the information effectively.

2.5 Training the model.

Keras provides wide range of layers and utilities out of the box. You will see more examples through the book and you will get used to them and will gain confidence to build your own custom models. When you finally come up

with a model you would like to train it on the data you have. Here's the next great power of Keras. It provides powerful tools out of the box to effectively train your model, monitor the training and make the whole process easier both for you and for the model.

Before you are able to train your model you have to compile it. This is Keras' way to actually set up the model, connect the dots, instantiate model components and initialize the weights.

To compile a model Keras provides the function **.compile()**.

Listing 2.3: Compiling Keras model.

```
model = Sequential()

...

model.compile(loss=keras.losses.binary_crossentropy,
              optimizer=keras.optimizers.Adam(),
              metrics=['accuracy'])
```

When you compile the model the most important thing that you should remember is that you provide the loss function, the optimizer and optionally the metrics you want to monitor when you train your model. Later you will see lots of examples for metrics you can monitor.

Keras optimizes over the loss function, but you usually aim for high accuracy of your model. Usually it's not convenient to optimize on accuracy, because of non stable gradients and not mathematically convenient functional definiton of accuracy, so you use your loss function as proxy of accuracy, or other metric. Basically this means that you assume that if your model has low loss on the dataset it will have higher accuracy. Most of the times this assumption is quite reasonable and the results are as expected.

So far you have loaded and prepared your data, built the architecture and you are ready to train the compiled model. If you are familiar with other ML frameworks you probably have some expectations of the interface to have something like **.train()**. Every Keras model instance provides a function **.fit()** that takes as input your training data and many parameters that enable you to build powerful model with just 1 parameter value changed. Some the most common parameters that you will usually use are:

- X, y

  This is your training set.

- batch_size

Keras provides a way to set you batch size when training through that parameter. You pass your training data X and y and then Keras samples batches of size *batch_size*, passes them through the model and updates the weights with respect of the gradients.

- epochs

  Epochs are the number of times Keras will sample a batch from the training set and update the weights based on the performance of the model over the batch. Keras will sample $\frac{N}{batch\_size}$ times from the dataset per epoch, unless other is specified with the param *steps_per_epoch*

- callbacks

  Keras introduces the concept of callbacks which essentially is just a function applied during different stages of the training of your model. The *callbacks* argument takes a list of callbacks.

  Example for Keras callbacks:

  – ModelCheckpoint

  Listing 2.4: Model checkpoint.

```
from keras.callbacks import ModelCheckpoint

# building and compiling model here

filepath = "models/" \
           "model-{epoch:02d}-{loss:.4f}.h5"

checkpoint = ModelCheckpoint(
    filepath,
    monitor='loss',
    save_best_only=False,
    save_weights_only=False,
    verbose=1
)
```

  ModelCheckpoint provides a way to checkpoint your model at the end of an epoch. Sometimes training a model requires weeks even on big clusters and epoch is one of the hyperparameters of the model that you have to tune. ModelCheckpoints enables you to train your model for many epochs and save all the models, so even if your models start overfitting from given point in time during the

training you can stop it, or just use a checkpoint before that. It also enables you to continue training from given checkpoint. You can train a model for 50 epochs, but if you are not happy with the results you can load the model from epoch 15, change some params and continue training from that point. Last but not least this can save your model if something happens with your machine and the training has not been finished, so the model has not been saved.

You can modify the checkpoint in such a way that it can save the best models only and set the params *mode* to *min* in case of choosing best based on loss or *max* if choosing best based on accuracy. You can also save the model architecture and weights, so then you can load it even if you don't have the architecture implemented again, or just save the weights and load them into the model. Usually it's better to save the model and the architecture, considering that computers have enough memory to handle these couple of MBs.

Finally you can provide a string that the ModelCheckpoint will modify depending on the format you have required. In the example above I have set the filepath to be the folder *models* and the name of each model to be **model-** and then the current epoch and the loss of the model on that epoch. The format that Keras saves models is h5. If you do not set the loss, or the epoch in the name of the model and set a name like "models/model.h5" Keras will overwrite the model on each epoch.

– ReduceLROnPlateau

Optimization algorithms like Adam and RMSProp are pretty smart to update learning rate during training based on the loss function, but still the models sometimes get stuck, or get close to minima, but do not end up in the exact minima. To deal with these problems Keras provides this really useful callback that reduces learning rate when a metric has stopped improving.

Listing 2.5: Reduce learning rate on plateau.

```
from keras.callbacks import ReduceLROnPlateau

# building and compiling model here

reduce_lr = ReduceLROnPlateau(
    monitor='val_loss',
    mode='min',
    factor=0.2,
    patience=2,
    min_lr=1e-6,
    verbose=1
)
```

You provide the quantity to be monitored and the mode for the monitoring. In case of mode *min* learning rate will be reduced when the quantity has stopped decreasing, otherwise it will reduced when the quantity stops increasing. Factor sets the factor by which the learning rate will be reduced. Patience sets the number of epochs that the monitored quantity has to not improve in order to reduce the learning rate. Finally you can set the minimum value for the learning rate, so you can guarantee yourself that it will not become too small.

- LambdaCallback

  Lambda callback allows you to create any custom callback that you need. Usually you will use this if you want to predict with your model on each epoch and save the output so you can monitor the improvement on given samples of your dataset.

- TensorBoard

  TensorBoard is a model monitoring tool provided by tensorflow. You can use this callback to log data to monitor in tensorboard live during the training. Tensorboard is not covered in this book.

- CSVLogger

  CSVLogger enables you to stream the results of each epoch to csv file. You can later plot each of the metrics and explore the training process in details in terms of loss value on training and validation data, metrics on training and validation data and choose the best epoch to use as final model.

- validation_split / validation_data

Instead of manually splitting your data Keras provides convenient way to just pass X and y to the fit function and set the validation_split param to a number in the range $(0, 1)$ and it will split it to train and validation data with respect of that ratio. In case you have augmented your training set and you want to train with augmented training set and evaluate on non-augmented validation set you can use the *validation_data* param and pass a tuple like (X_val, y_val) and Keras will use that to compute validation loss, accuracy and other metrics you have required.

After the model is compiled and the checkpoints are instantiated you can finally train the model.

Listing 2.6: Example of starting to train a model.

```
# building and compiling model here
filepath = "models/" \
           "model-{epoch:02d}-{loss:.4f}.h5"

checkpoint = ModelCheckpoint(
    filepath,
    monitor='loss',
    save_best_only=False,
    save_weights_only=False,
    verbose=1
)

reduce_lr = ReduceLROnPlateau(
    monitor='val_loss',
    mode='min',
    min_lr=1e-6,
    verbose=1
)
callbacks_list = [checkpoint, reduce_lr]

history = model.fit(
    X_train, y_train,
    epochs=15,
    validation_data=(X_test, y_test),
    batch_size=16,
    callbacks=callbacks_list
)
```

The *.fit* function returns history of the training process which contains all the information of the training process like values of the loss function on each epoch, accuracy and so on. You can use it to produce some graphics like the ones below.

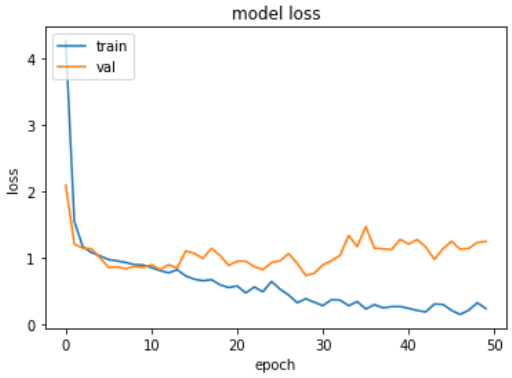


Figure 2.19: Loss function values during training.

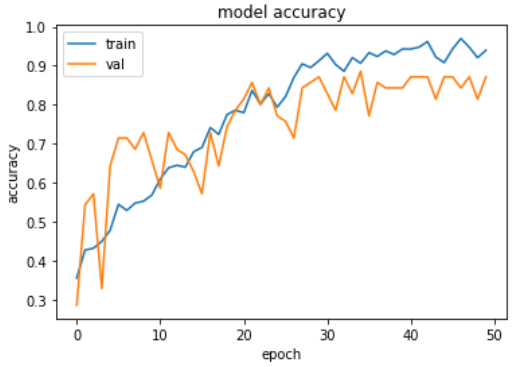


Figure 2.20: Accuracy values during training

This is the code used to generate the graphics.

Listing 2.7: Plot model training history.

```
import matplotlib.pyplot as plt

plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'val'])
plt.show()

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'val'])
plt.show()
```

3. CNN input preprocessing

Just as any other ML models we can treat images, or fixed length sequence of images as input to our model and as any input our model usually can benefit from applying some preprocessing.

3.1 Normalization

Normalization of the input has became a default step when training a model. The main reason for most of the models to benefit normalization is actually the optimization algorithm itself. The goal of normalization is to map the numeric features in the dataset to a common scale, without distorting the relative difference of the values. The simple intuition behind this is that when the optimization algorithm performs a step it has to scale that steps with respect to the different features. For some reasons models are not always that good at this, though simple for us task. Mapping the feature space in the range $[0, 1]$, or $[-1, 1]$ usually let's the optimization algorithms to gain extra performance.

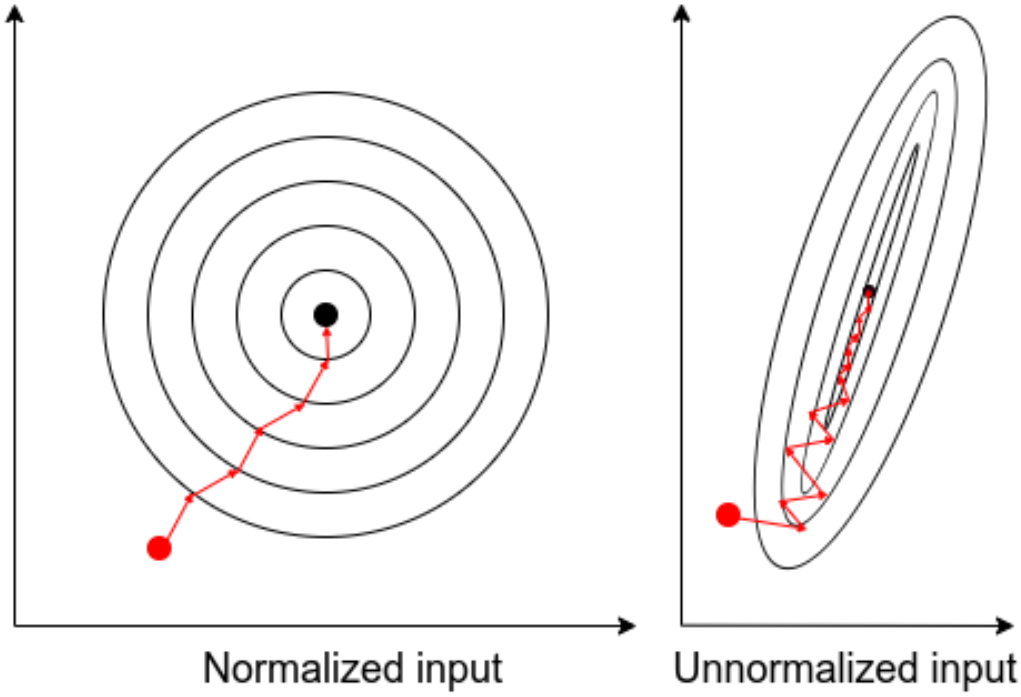


Figure 3.1: Example of normalization effect on the loss function.

What you see above is called level lines or contour lines. It is the result of *slicing* a 3D function with several planes and plot the intersections in 2D plane. This kind of graphics are quite often used as examples in machine learning and they usually look quite hard to bare with, but they are actually the same contour lines as the ones on a geographical map where a mountain is displayed as lines with the height on each of the lines. All this is is the intersection of the mountain with a plate on height 1200, 1300 meters and so on. Same applies for a function.

As you see on the example when the input is normalized the coefficients are also in the same range, so the loss function has the same scale on the different axis. The result is that when the algorithm performs optimization step on each batch it does not have to worry about scales at all and it takes relatively smooth steps towards the minimum. In case of unnormalized input, on the other hand, one of the inputs is, for example in the range $[-5000, 25000]$ and another one is in the range $[0, 1]$. The result is that the network comes up with really small in absolute value weight for the feature in the range $[-5000, 250000]$ and relatively much bigger weight for the feature in the range $[0, 1]$ compared to the other weight. That's how the loss functions appears to be squashed like the one on the right on fig 3.1. That's how at the

end the gradient for the one axis is relatively much smaller to the gradient corresponding to the other axis.

After all, images are just another type of input to your model, so usually you will improve your model if you divide the image by 255.0 in the case of RGB image, or the max value, in case of more than 8 bits per channel image. Later you will see that actually Keras provides really easy and convenient way to normalize your input while you load it and feed it to your model, so you won't even need to do it manually.

3.2 Augmentation

Usually in practice we have lots of data, but there are some cases when the data is not enough, or even the more common case having a dataset with non-uniform distribution of the classes. For example let's say you want to classify if an image has a cat or a panda. It would not be odd to assume that you will have orders of more cat pictures than pandas, just because there are millions of cats and a few thousand pandas in the world. So how would that affect your model? There are a few cases:

1. You will have deep CNN with several millions or even tens of millions parameters and 1M cat pictures vs 10k panda pictures. The result will be Biased dataset and very likely biased model after the training.

2. Randomly sample 10k from the cat images and train your network on 20k total images. Very likely the network is not going to learn anything from such low amount of data.

3. Another, probably most reasonable, approach is to augment your less represented class to generate new dataset.

3.2.1 What is augmentation

Data augmentation is just the process of applying operations on the data to generate new data. In terms of images this usually comes to be mirroring, flipping, rotating, cropping and many more. You can just think of it as an operation that changes the image in such a way that it's totally new for the computer, but still contains the information required to predict the label. For example mirroring an image with panda will be still image with panda. The key part is that from pixels perspective this is going to be totally different matrix, but still a human eye can easily detect it's just a mirrored image. Of

course you should always apply these transformations carefully. In the case of self driving cars you probably do not want your model to consider the left lane and the right lane the same way, so the perspective there will be critically important and mirroring the image would make the coming towards you cars appear in different perspective, which may result in catastrophic model.

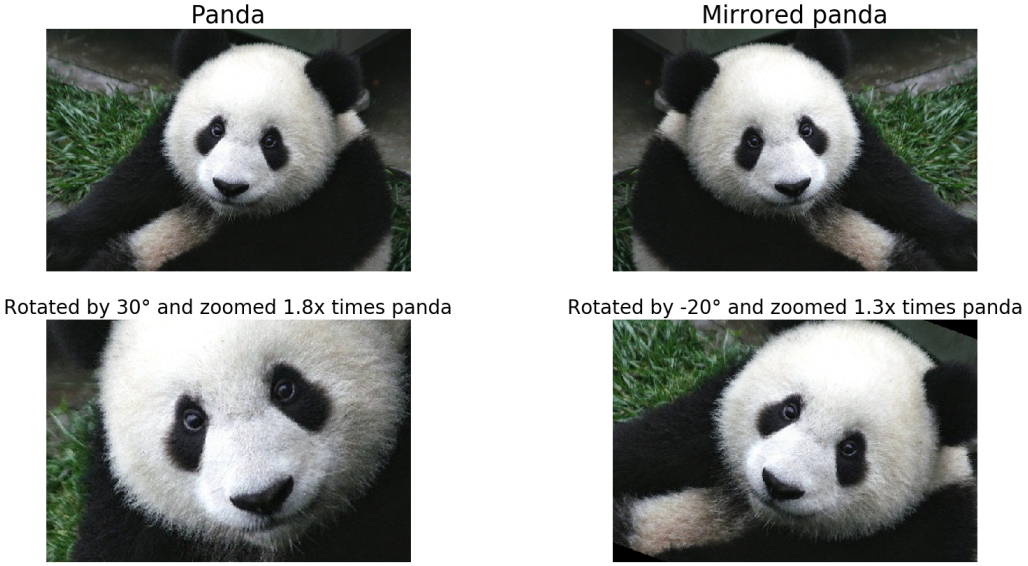


Figure 3.2: Panda augmentation example.

3.3 Applying filters

Computer vision is a field much older than convolutional neural networks. Scientists have tried to solve complex problems related to visionary data. As a result there is wide range of algorithms developed like haar cascades for face detection, Kalman filters for object tracking and so on. Though modern CNN architectures outperform these algorithms and achieve higher accuracy and are easily scalable and generalizable, these algorithms are the foundation of computer vision and most of the concepts have inspired CNNs to be what they actually are.

You will see how you can use certain types of filters as input preprocessing in order to extract features and achieve high accuracy with smaller models which is sometimes crucial. Definitely if you have deep enough model with hundreds of millions parameters this will usually outperform any preprocessing and will achieve high accuracy, but it comes with the challenges to train such a giant model. It requires tons of data, it's slow for training, hard for

optimizing and convergence and finally it's slow and usually impossible to run in production service where you have requirement of 100 ms per request for example. In this case you will go for smaller model and these techniques may help you squeeze another percent of accuracy.

3.3.1 Image gradient

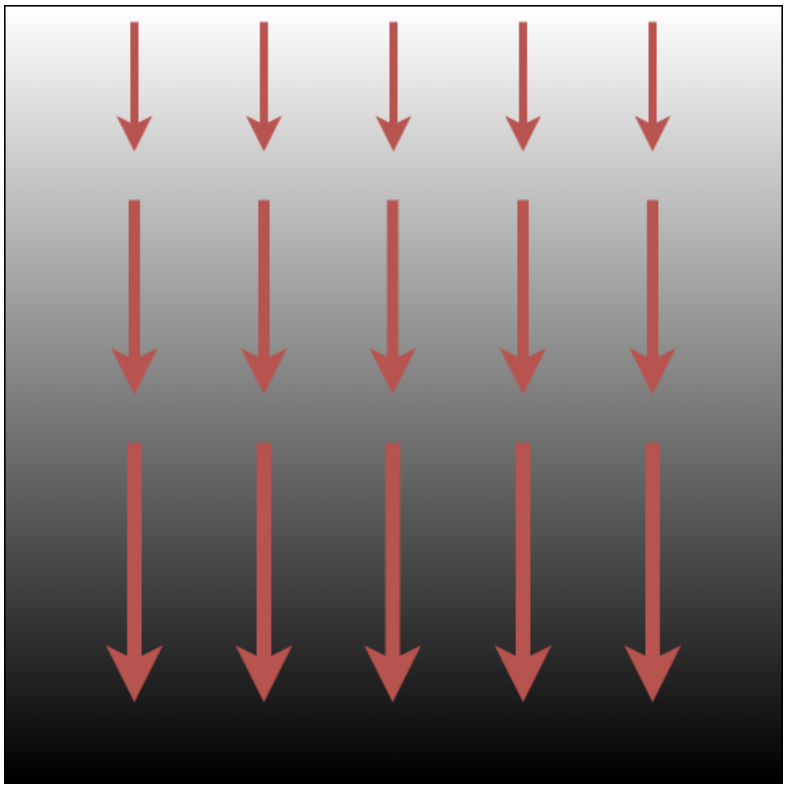


Figure 3.3: Direction of gradients on image.

This is one of the fundamentals in image processing. Though it may seem that image gradients are something that does not make sense, considering that gradients are something associated with a derivative, but you already know how to interpret images as functions, so hopefully this does not sound that odd now. If you think of image as 2D function then image gradient is exactly the steepest direction in small local region. In terms of images this means the direction of largest possible intensity decrease, because each pixel value is actually the intensity of light at that point.

Derivative is defined with limits, but there is also a method to approximate the partial derivatives with finite differences across the two dimensions

of the image x and y. This means that the derivative of an image I can be calculated as follows:

$$\nabla I = \begin{bmatrix} \frac{\partial I}{\partial x} \\ \frac{\partial I}{\partial y} \end{bmatrix}$$

You can calculate the two partial derivatives the following way:

$$\frac{\partial I}{\partial x} = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} * I$$

Where the $*$ sign means convolution in the meaning you already know it. Same applies for the second dimension:

$$\frac{\partial I}{\partial y} = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} * I$$

The result of applying these filters and computing the gradients over the following sudoku picture will be:

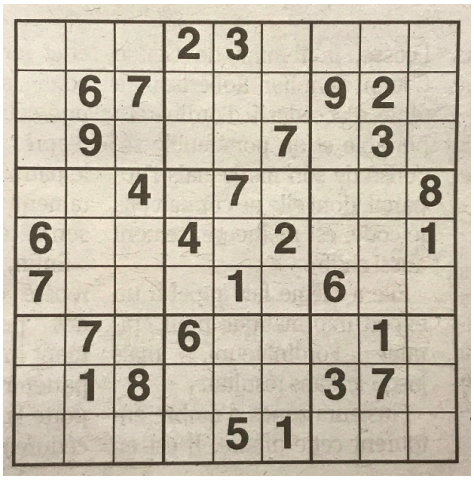


Figure 3.4: Sudoku picture

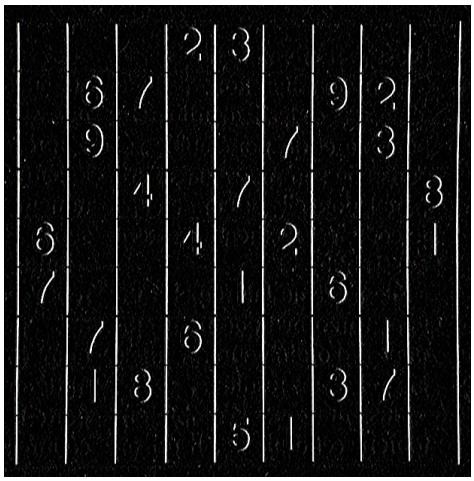


Figure 3.5: Sudoku gradients over the x axis

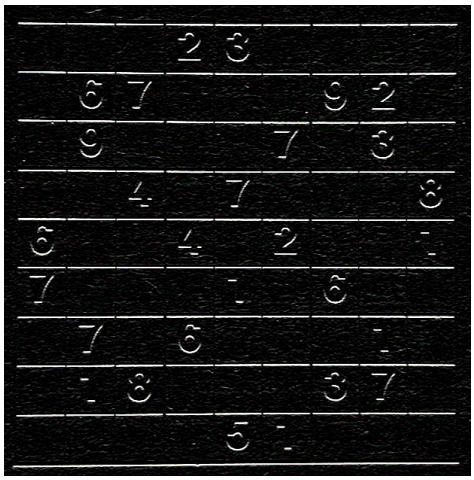


Figure 3.6: Sudoku gradients over the y axis

As you can see the x derivative highlights the vertical edges and the y derivative highlights the horizontal edges. These filters have been used in computer vision long time ago for edge detection. This is useful property you can benefit and a reason to use these filters as preprocessing for your input.

Keep in mind that when you apply such preprocessing of your input you do not completely replace the raw input image. You still keep it and actually end up with several images for input which you stack together ending up in a tensor which you can interpret as multichannel image. For example if you have rgb image of size 128×128 *as input, then you compute the* x *and* y

derivatives and end up with 2 more images as the ones above you can feed your network a tensor of size $128 \times 128 \times 5$*: 3 for the red, green and blue channels and 2 for each of the* x *and* y *derivatives.*

3.3.2 The Sobel operator

Once you are familiar with image gradients all other types of filters are easy, because the concept is exactly the same. Sobel filters are slightly modified gradient filters that have higher weight for the middle pixels. Sobel filters are used in image processing for edge detection. the Sobel filter produces image with highly emphasized edges.

In order to apply Sobel filter over image you use convolution with the following filters.

$$I_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & 1 \end{bmatrix} * I$$

$$I_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * I$$

The result will be the following.

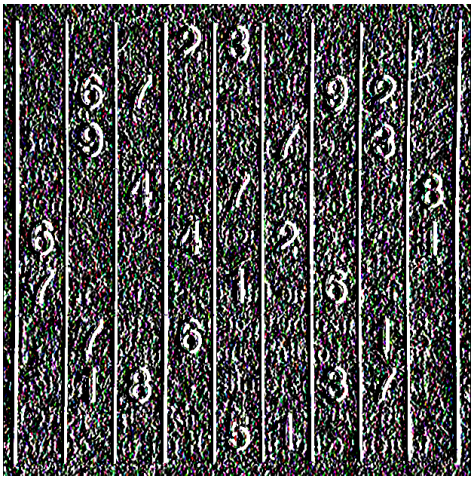


Figure 3.7: Sudoku sobel over the x axis

Figure 3.8: Sudoku sobel over the y axis

3.3.3 The Laplacian operator

Another a bit more advanced operator is the Laplacian operator. The major difference between Laplacian and Sobel operator or simple derivative is that they are first order derivative operators, while Laplacian is a second order derivative operator.

The Laplacian filter looks as follows:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Notice that it adds up to zero. This actually applies to all the filters that we saw so far. This is important property that prevents the filter from distorting the image intensity. If you apply a filter that sum to more or less than zero this will make the result brighter or darker than it initially was.

Another difference between Laplacian and other operators is that unlike other operators Laplacian does not take out edges in any particular direction but it takes out inward and outward edges.

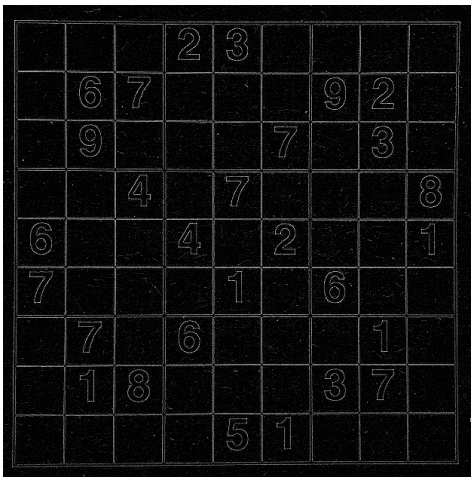


Figure 3.9: Sudoku Laplacian operator applied

There are many more filters, but remember that you can also modify these ones, add weights to some of the pixels if this makes sense for your application, or come up with totally different filters to apply. Filters are not finite group of already discovered objects that you can only apply. Think of them as a tool that you can use, just as all the other tools you will learn.

4. Image classification. Imagenet

Image classification is the task of predicting a class based on image. For example if you have an image of dog you want your model to output the label for dog (or any encoding of it). We already have solved a classification problem with the MNIST dataset consisting of images of handwritten digits. Now we will see another example of image classification and introduce several really famous CNN architectures.

Imagenet is a project providing the most famous dataset for image classification consisting of more than 20000 classes and 14M images. There is also a competition helt once a year. Usually every year there is at least one state of art model that introduces an innovative concept. The first architecture which we are going to start with is the first deep model introduced to the competition. It also reduces the error on the performance from about 25% *to about* 15%.

4.1 AlexNet

AlexNet was introduced back in 2012 when there was no TensorFlow, Keras, Pytorch or many of todays deep learning frameworks. It contains 8 layers: five convolutional followed by max-pooling layers and 3 dense. Though from today's point of view this is nothing great and is usually an architecture you will write in 10-15 minutes and train for proof of concept, but back then it has been truly state of the art model.

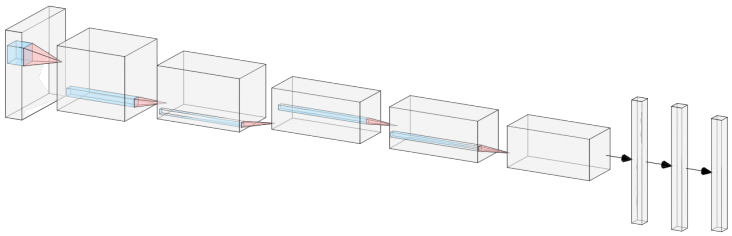


Figure 4.1: AlexNet architecture

The network achieved a top-5 error of 15.3%, more than 10.8 % lower than the previous top performing model.

Here is exact implementation of AlexNet, but with 4x times reduced size (filters or units divided by 4) to make training easier with less data.

Listing 4.1: AlexNet implementation in Keras

```
model = Sequential()

model.add(Conv2D(filters=24,
                 input_shape=(128, 128, 1),
                 kernel_size=(11, 11),
                 strides=(2, 2),
                 padding="valid",
                 activation="relu"))
model.add(MaxPool2D(pool_size=(3, 3),
                    strides=(2, 2),
                    padding="valid"))
model.add(Conv2D(filters=64,
                 kernel_size=(5, 5),
                 strides=(1, 1),
                 padding="same",
                 activation="relu"))
model.add(MaxPool2D(pool_size=(3, 3),
                    strides=(2, 2),
                    padding="valid"))
model.add(Conv2D(filters=96,
                 kernel_size=(3, 3),
                 strides=(1, 1),
                 padding="same",
                 activation="relu"))
model.add(Conv2D(filters=96,
```

```
                    kernel_size=(3, 3),
                    strides=(1, 1),
                    padding="same",
                    activation="relu"))
model.add(Conv2D(filters=64,
                    kernel_size=(3, 3),
                    strides=(1, 1),
                    padding="same",
                    activation="relu"))
model.add(MaxPool2D(pool_size=(3, 3),
                        strides=(2, 2),
                        padding="valid"))
model.add(Flatten())
model.add(Dense(units=2304, activation="relu"))
model.add(Dense(units=1024, activation="relu"))
model.add(Dense(1024, activation="relu"))
model.add(Dense(8, activation="softmax"))

optimizer = Adam(lr=0.001)
model.compile(optimizer=optimizer,
                loss='categorical_crossentropy',
                metrics=['accuracy'])
```

After AlexNet another architecture called VGG was introduced. It was made of the same building blocks, but was even deeper. I'm not going to cover it in this book, but you can compare it's performace to the model shown above, using Keras' pretrained VGG models.

4.2 ResNet

After the first breakthrough in convolutional neural networks and AlexNet's great performance there was another Eureka moment when ResNet was introduced on the Imagenet contest. ResNet was 152 layers deep neural network achieving 3.57% error rate on the Imagenet dataset.

There is a theorem stating that a neural network with a single layer is sufficient to approximate any function. However this layer might be enormous and the network might result being ineffective and prone to overfitting. That's why scientists have focused in developing deeper and deeper neural networks in order to achieve higher accuracy and produce models that have capacity closer to human's mind.

For reference the measured human performance on the Imagenet dataset is around 5.1% error rate, which is above what ResNet has achieved.

The challenge in coming with a really deep model is not actually in coming up with the architecture itself. You can always stack Conv layers tens of times and get deeper and deeper network, but unfortunately sooner or later (probably sooner) you will hit a point where you won't be able to train your network. The reason for that is called Vanishing gradients or Exploding gradients.

The concept of vanishing / exploding gradients is pretty simple. You can think of a network with input single number and each layer single number weight. Let's also the network to have 100 layers. Then the actual process of passing forward input through the network is going to be the below expression

$$x * w_1 * w_2 * ... * w_{99} * w_{100}$$

Let's assume three different scenarios.

Let's first most of the weights w_i be bigger than 1. For the sake of the explanation assume each weight w_i is about 1.5. The output of the network is going to be $4.065611775 \times 10^{17}$

Let's now most of the weights w_i be smaller than 1. For the sake of the explanation assume each weight w_i is about 0.8. The output of the network is going to be 0, or at least my computer rounds it to zero.

These are examples for exploding and vanishing gradients respectively. If you remember backpropagation - the algorithm that enables us to train deep networks it's actually based on multiplication.

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial z}\frac{\partial z}{\partial y}...\frac{\partial y}{\partial x}$$

That's why when the networks gets deeper the more multiplications result in really big or really close to zero numbers, which stops the network from actually computing the gradients and updating the weights, which essentially means not being able to get trained.

The third scenario is the case where the weights w_i are about 1, or distributed in such a way that multiplying them will still be a result that is not extremely big or small enough to be considered zero, but unfortunately usually this is not the case.

So in order to deal with this problem scientists have seen that compressing the paths through the network can improve the gradients transfer, because if the result of one layer is passed to another layer skipping several layers between them, then the gradients would also be computed with respect to this transfer, so the actual result would be also compressed paths for the

gradients to be propagated through. In essence having a 150 layers network with skips over 3 layers would have the same capacity to propagate gradients as a network with only 50 layers depth.

ResNet is the first very deep network that aims to achieve this. The very smart approach the authors of the network take is based on the following building block of the network.

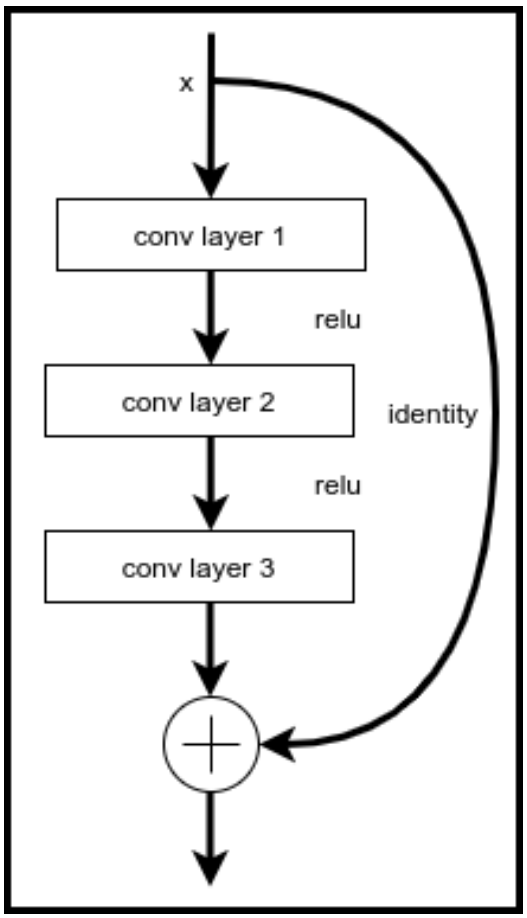


Figure 4.2: Residual block - the building block of the Residual Network (ResNet)

Residual blocks are units that you can consider as single layer with input and output. Though they are not as simple as Conv layers, but instead are made of several smaller units like Conv layers, or sometimes pooling layers, or others. The original ResNet paper uses Conv layers only with identity connection between the input and the output of the residual block unit. This identity funciton basically just copies the input and propagates it to

the end of the block and then combines it to the result of the conv layers summing the two results to produce the final result of the residual block.

In terms of gradients this means that whenever we propagate the gradients we are going to once compute the gradients corresponding to the conv layers and once just propagate the previous layers without touching them, because the derivative of the identity function is the constant 1, so this means that the input gradients of the model are going to be multiplied by 1.

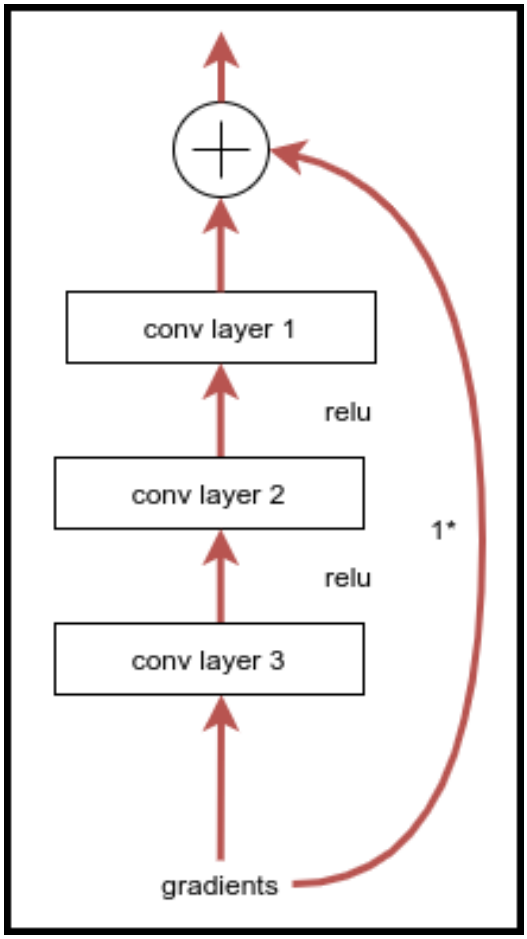


Figure 4.3: The flow of the gradients through the residual block

These skip paths actually reduce the size in terms of gradients paths which enables training really deep networks like ResNet.

Keras provides imlpementations of different ResNet architectures and pre-trained weights with the imagenet dataset that you can use for your applications, or if you want to do a proof of concept fast and relatively simple.

When you want to build really deep model quickly you usually do not build and train models like ResNet from scratch. Instead you use the weights of a pretrained model, replace the last 1-2-3 layers and put a layer corresponding to the output shape your dataset has. Then you freeze all the other layers except the ones you added and train the network. The result is that only the newly added layers are trained. This is achieved by simply zeroeing the gradients for the freezed layers. This means that the weights are not being changed.

After that you can unfreeze all the layers and train for several additional epochs to fine tune all the weights to your specific problem and dataset.

This is called transfer learning and these techniques are usually used in research teams trying to squeeze another bit of performance of their model.

Listing 4.2: Pretrained ResNet50 Model Keras

```
pretrained_model = ResNet50(
    include_top=False,
    weights='imagenet'
)

for layer in pretrained_model.layers[0:-5]:
    layer.trainable = False

model = Sequential()
model.add(ZeroPadding2D((96, 96),
          input_shape=(32, 32, 3)))
model.add(pretrained_model)
model.add(Flatten())
model.add(Dense(num_classes, activation='softmax'))
model.compile(
    loss='categorical_crossentropy',
    optimizer='adam',
    metrics=['accuracy']
)

model.summary()

model.fit(
    X_train, y_train,
    epochs=5
)

for layer in pretrained_model.layers:
    layer.trainable = True

model.fit(
    X_train, y_train,
    epochs=3
)
```

The code above shows exactly how this technique can be applied in Keras. You instantiate the ResNet50 model and specify to use the pretrained weights on the imagenet dataset. Just because you use the pretrained weights this means the input layer expects fixed size which in terms of ResNet is 224×224, so the simplest thing we can do is to add zero padding of the 32×32 cifar

10 image in order to input image with the expected shape.

Figure 4.4: Sample image of class frog from the CIFAR10 dataset

Figure 4.5: Zero padded image to size 224×224

Though this the simplest approach it may affect the performance in negative way, but for the purpose of explaining the concept it works fine.

So once loading the pretrained ResNet model we use it to build new sequential model by adding a zero padding layer for the input and then pass that to the pretrained ResNet model, flattening the ouput of the model and connecting it to a 10 units dense layer for each of the classes of CIFAR10.

After freezing all the layers except the last five we end up with the following model.

Listing 4.3: Keras transfer learning example using ResNet50 backbone

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
zero_padding2d_1 (None, 224, 224, 3)       0
_________________________________________________________________
resnet50 (Model)             multiple                  23587712
_________________________________________________________________
flatten_1                    (None, 2048)              0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                20490
=================================================================
Total params: 23,608,202
Trainable params: 1,075,210
Non-trainable params: 22,532,992
```

4.3 DenseNet

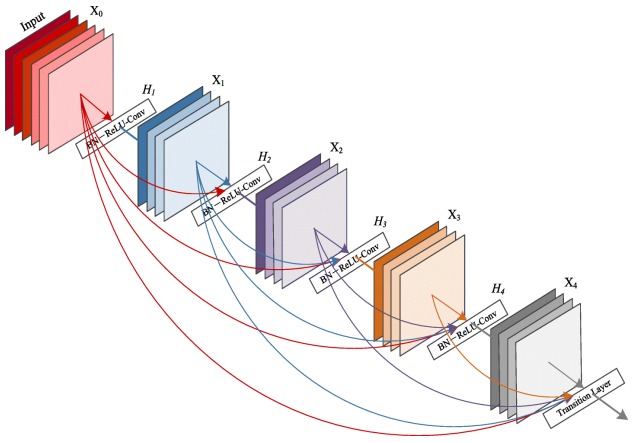


Figure 4.6: DenseNet

In order to solve the problem with gradients transfer through the network more and more architectures appear. One of them made another big step achieving great performance in general and made it's place among the other famous architectures. DenseNet introduces the idea for dense connections between layers. This means that there is connection between every two layers in the network. This definitely enables gradients flow. Of course this comes

with a problem. You either have to keep the size of each layer tha same, or you have to set function to combine the output of two layers into one, which is going to be kinda tedious. The other problem is that having a network of n layers results in $\frac{n(n-1)}{2}$ connections which makes it really easy to create enormous network that is slow to train and work with. But still the concept has lots of value in it, so you can actually come with idea similar to the one of ResNet and use dense blocks.

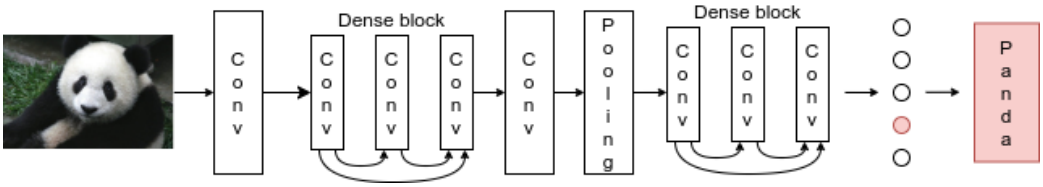


Figure 4.7: Dense blocks

That's just another example how you can take any concept and modify it anyway you want in order to solve the problem you want to solve and build your application. There are tons of CNNs and many more are to come, but after all once you feel comfortable with Keras you will start building your own architectures using concepts and other architectures as building blocks for your own model.

4.4 Other architectures

There is a bunch of other famous architectures like VGG, MobileNet, InceptionNet, EfficientNet and many more. We will not cover them in the book, but you may give them a look and become familiar what approach they take and what problems are they usually used for, so once you have to solve that problem you have them in your toolbox.

4.5 ImageDataGenerator

When you train CNN you usually train on images. Imagenet for example is about 150GB, which means that this may be difficult to train on a single machine. Another problem that Imagenet has is that some of the classes do not have enough samples, which makes the training process even harder. These are the real problems you usually hit when doing Deep learning: you either have too little or too much data.

To solve these problems Keras provides really convenient solution. **ImageDataGenerator** provides a way to solve both of the problems.

ImageDataGenerator for data augmentation:

Listing 4.4: Data augmentation with Keras ImageDataGenerator

```
datagenerator = ImageDataGenerator(
        samplewise_center=True,
        samplewise_std_normalization=True,
        rotation_range=30,
        width_shift_range=0.25,
        height_shift_range=0.25,
        shear_range=0.2,
        zoom_range=0.3,
        fill_mode='nearest',
        horizontal_flip=True)

    datagenerator.fit(X_train)

    model.fit_generator(
        datagenerator.flow(x_train, y_train,
                           batch_size=batch_size),
        epochs=epochs,
        validation_data=(x_test, y_test),
        workers=4
    )
```

In case you don't have enough data and your model does not actually train you may try increasing the size of your dataset as explained in the **Data Augmentation** section. Of course you can always go with custom augmentation functions, but for your convenience Keras' **ImageDataGenerator** provides most of the common augmentation functions like rotation, shearing, flipping, shifting, resclaing and so on. It also provides easy way to normalize your input data images and directly feed them into your model which turns the whole process of preparing the data and training a model into just reading the images and passing them to the generator that's going to deal with generating batches, randomly applying augmentation on each image of a batch and feeding it to the model during the training. Basically you don't have to deal with anything except building the model and the generator, reading the images and tuning parameters of the model letting you skip the tedious part of preparing images, trying to fit them in memory, augmenting them manually and verifying it and so on.

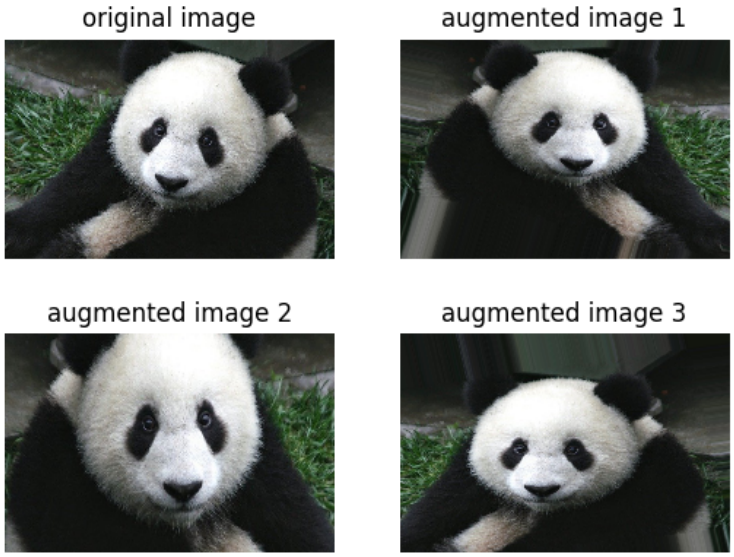


Figure 4.8: Sample result of ImageDataGenerator

4.5.1 ImageDataGenerator for dealing with huge datasets:

In case you want to train a neural network on millions of images you probably won't be able to store them in the operational memory of your machine. But still the concept of batch training actually provides a way to train on all of the images and yet pass them batch by batch to the network. That's why you can actually store them on the hard disk of your machine and just load a batch whenever you need it, and then free the memory once the batch is processed. You can, for example, store the names of the files on your filesystem and randomly select batches from the pool of names and then load the actual images.

ImageDataGenerator provides an option for this.

Listing 4.5: ImageDataGenerator generating from directory

```
train_datagen = ImageDataGenerator(
    rescale=1./255
)

train_generator = train_datagen.flow_from_directory(
    'imagenet/imagenet_images',
    target_size=(150, 150),
    batch_size=32,
    class_mode='categorical'
)

model.fit_generator(
    train_generator,
    steps_per_epoch=2000,
    epochs=5
)
```

Here's and example how you can use the ImageDataGenerator class to load images from the directory *imagenet/imagenet_images*. You can also add paramaters for augmentation as the generator above, but for simplicity this generator just normalizes the image pixel values to be in the range $[0, 1]$ instead of $[0, 255]$. The generator assumes that the directory will have N subdirectories each for one of the classes your model will predict. It also handles the resizing of the read image so it can fit to the input size your network will expect. Another important parameter here is the *class_mode*. What happens when you select **class_mode='categorical'** is that the folder names are being label encoded and then one hot encoded so that when you load the image you also load the label as one hot encoded vector that you can directly feed to your model and train with categorical cross entropy for example.

class_mode='binary' will load the data only label encoded and

class_mode=None will not load labels at all. The last one is usually used when you want to predict on a trained model using a generator and the predict_generator() method.

Here's example of the difference between class_mode *categorical* vs *binary*.

Let's say that you have 3 subdirectories in the training folder named **cat, dog, panda**.

If you have 3 images in each directories and you use the

class_mode='categorical' the generator will load 9 images in total for X and the labels y will look like the following:

$$y = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

3 vectors for the cat label, 3 for the dog and 3 for the panda label.

The **binary** option on the other hand will generate the following label vector:

$$y = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 2 & 2 & 2 \end{pmatrix}$$

Where the class cat has label 0, dog has label 1 and pandas has the class 2.

Usually you will use the binary option only when you solve binary classification problem and the categorical option when you have more than two classes.

As you can see Keras generator is a really powerful tool enabling you to work with image data and helping you deal with problems like small amount of data, or huge amount of data and all this out of the box, so you don't actually have to implement it, but instead focus on your model. It's really easy to wrap this concept and train production quality models on large scale.

5. Face Detection and Recognition. Siamese CNNs. Triplet loss.

5.1 Face Detection

Face detection is the task of identifying if a given image contains human face. It has nothing to do with identifying the person. All it aims to do is to answer if there is a face or no.

There are pretty good CNNs solving this problem with accuracy of almost 100%. Before exploring them we will go through an older method that has proved itself through the years, has high accuracy and is foundation of face detection and recognition and CNNs at all. In the next section we will build CNN for face recognition, instead of detection, but once you understand it you will easily be able to modify it in order to solve the easier problem for face detection.

5.1.1 Haar-like features

This is probably the closest to CNNs approach that is based on non-trained filters. Haar features are set of filters based on assumtions like that in the human face usually the region of the eyes is darker than the region of the cheeks. They are similar to convolution filters, but they are manually hard-coded, instead of learnt via optimization algorithm.

Figure 5.1: Example Haar filters

Paul Viola and Michael Jones have used these features to build the famous Viola–Jones object detection framework, that you can use to detect faces with pretty high accuracy and speed. It uses the features with a bunch of weak classifiers on top of them, but we will not cover the system here, though it might be useful to read more about it later, if you are interested in the topic and a more classic approach.

Hopefully this helps you to get more intuition on how CNN filters actually work and why they work at all. If there is still something unclear about it the idea will get polished in the filters visualization chapter later.

5.2 Face Recognition

Whenever CNNs are mentioned face recognition is a problem that usually comes to mind, let's actually try to solve it. To make it clear we will explicitly say what face recognition means to us.

> Face recognition is the task of identifying a person from a digital image.

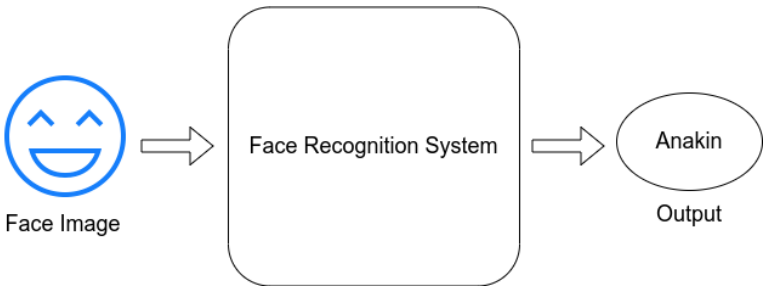


Figure 5.2: Face recognition system

Let's consider a company which wants to develop security system based on deep learning. What would they expect the system to be like? They

would basically expect it to separate people on two groups - people who have access to the company and people who do not have access.

Stated the problem in this way we would think of it as a binary classification. So how would be implement the underlying algorithm? Well, as we mentioned in the definition above the system takes a digital image as input, so this is going to be the input of our algorithm as well. On the other end we expect binary answer **yes/no** if the person on the image should be granted access or no.

We already solved this type of problem in a previous chapter. We will simply build a CNN and will train it on images of the company's employees.

Here we start facing the problems of this approach.

First of all we saw that to actually train a CNN you need lots of data. An by lots we mean **a lot**. Even if you augment your data you will need to take lots of pictures of your employees. Even if we assume that your find enough smart ways to augment your data so you end up with enough data to actually train your model you will hit another problem.

Once the model is trained it maps fixed set of employees to the set **yes/no**. What happens when the company hires a new employee, or someone leaves? Well, after all you will have to train the model whenever someone joins or leaves. This is neither scalable nor sustainable.

We will examine two approaches to solve this problem both very effective and build on top of state of the art deep learning models.

5.3 Siamese CNNs

Let's try to state the problem again, but in a slightly different way. Instead of feeding an image to the model and producing a label representing employee let's feed two images to the model and ask it to tell us if the images are photos of the same person or no. It's still a binary classification problem, but instead of one image we feed to the model two images as ask it to compare them.

Though this looks as a minor change you will see that it removes the requirement to pretrain the model on every joining / leaving employee.

Siamese neural networks are class of deep neural networks that consumes a pair of input vectors and shares weights while working on the different input vectors to compute the output.

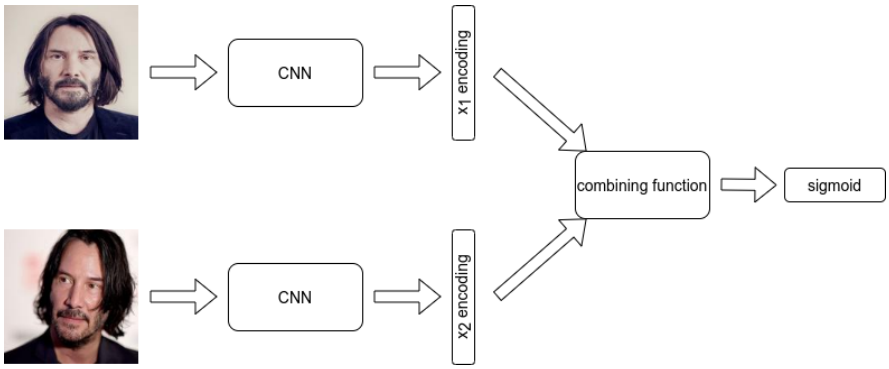


Figure 5.3: Siamese convolutional neural network architecture

Traditionally neural networks are trained and predict over multiple classes. As in the example above this causes a problem when we need to add or remove a class to the data. Also traditional NNs require a lot of data for each class, while siamese networks aim to learn **similarity function** which allows us to train on a few examples which resolves the problem of both low amount of data for some classes and class imbalance.

One of the philosophical interpretation of siamese neural networks is that they are able to generalize knowledge similar to humans. For example an avarage 20 years old person has a lot of experience, but still has the chance to come to an object he has never seen before. Let's say a man sees an eyelash curler for the first time. Though he may not be sure what it is exactly, he will be almost definitely sure this object is not something you can ride, or fly on. What siamese neural networks aim for initially is gain knowledge like this and transfer it to new never seen before objects. This is not exact and scientific explanation, but more like intuition about the idea of them.

There are several important details about the siamese neural networks.

As mentioned above they consume a pair as input. In our case this is pairs of images. Then the network *shares* weights across the forward pass path of the network. What this fancy term actually means is that this is just one CNN. Though you will almost always see architecture diagrams having two blocks for CNNs **it is actually just one CNN** which takes one image as input and what happens behind the scenes is that one of the images is passed forward until the *encoding* part, then the second one until the same point. So far the network has produced two outputs which we call *encodings*.

What encoding actually means is just another representation of the image. Definitely in the general case this is not a human understandable representation, but it's a representation that the model can use to describe the input with less words. As you remember we talked about images as $N \times M$ dimen-

sional points in the space. Well, what this encoding is is just less dimensional data point of the same image. It works as dimensionality reduction, which keeps as much information as possible.

So the CNNs has produced two, let's say 128 dimensional, vectors representing the two input images. Now the next step is to combine them in some way. You have infinite options for combining function, but for the sake of the example let's go with the L_2 euclidean distance.

$$L_2(x_1, x_2) = \sqrt{\sum_{i=1}^{128}(x_1^i - x_2^i)^2}$$

This will produce a number which we will pass to a sigmoid function and finally produce a number in the range $[0, 1]$ which we will interpret as similarity score.

Second important thing is the actual generation of training pairs. Consider having a dataset of 1M images consisting 100K people with 10 images for each person. How will we train siamese CNN? What we will do is simply randomly pick two images of the dataset and pair them. If they are of the same class we will assign label 1 to the pair, otherwise we will assign 0 to the pair.

This is not exactly true, because picking randomly from such a huge set would almost always result in different images and the training process is actually going to be really slow, since the examples are going to be easy for the network. There are strategies which prioritize pairing hard examples over easy examples, but we will not cover them in this book.

Finally let's explain how we will solve the face recognition problem for the company we develop the system for.

Using this architecture we are now not relying on images of exactly the employees of the company. Now we can use any face recognition dataset online and train a model like the one described above that produces similarity score. When we have such a model all we need is one picture per employee and then whenever an employee tries to access the company we can pair their image with all the images we had previously and try to match it to the employees the company has. Now adding or removing an employee is as simple as deleting an image from the file system.

5.4 FaceNet

To actually implement Siamese NN you have to use the functional API of Keras, because as already mentioned the Sequential API that was covered

so far introduces the limitation of having single input or output and makes weights sharing impossible.

5.5 Functional API

Keras' functional API is a way to build more complex models that have multiple inputs or outputs, share layers or have direct acyclic graph-like structure. Once you are familiar Keras and it's sequential API you will see that the functional API is not much harder to learn, though it's extremely powerful. That's why Keras is such an amazing framework and it's so widely adopted, because it provides great power with exactly as much complexity as it is required and no more. The functional API provides the same layers, parameters and syntax with the only minor differences for the input and that the tensors are manually passed by you to the layers as function calls. This actually provides you with the ability to have a network that branches in the middle, has several inputs, or outputs and shares layers. When you have access to the tensor on each layer you can do whatever you want with it. The functional API really works just like function calls. Once you get used to it you will probably not see difference when you read the source of a Keras model.

Let's look at an example of the same model we built several chapters ago, but this time using the Sequential API and the functional API.

Here's simple CNN with 2 Conv2D layers followed by MaxPool2D layers then flattened and densely connected with 5 output units, which means that it outputs one of five classes, if we assume this is classification problem.

Consider that though we study examples of classification problems and use CNNs to solve them there is no reason to use CNNs to solve regression problems. It is usually recommended to convert regression problems into classification, because deep models are usually much better at classification than regression, but this is the general case and you should remember that there is no theoretical reason to not be able to train CNN for regression problem. The difference will be that instead of softmax as activation function on the output you will use linear function, for example.

Listing 5.1: Simple Sequential API CNN

```
sequential_model = Sequential()
sequential_model.add(Conv2D(filters=16,
                            input_shape=(32, 32, 3),
                            kernel_size=(3, 3),
                            activation="relu"))
sequential_model.add(MaxPool2D(pool_size=(3, 3),
                               padding="valid"))
sequential_model.add(Conv2D(filters=32,
                            kernel_size=(3, 3),
                            activation="relu"))
sequential_model.add(MaxPool2D(pool_size=(3, 3),
                               padding="valid"))
sequential_model.add(Flatten())
sequential_model.add(Dense(units=5,
                           activation="softmax"))

opt = keras.optimizers.RMSprop(lr=0.0001, decay=1e-6)

sequential_model.compile(
    loss='categorical_crossentropy',
    optimizer=opt,
    metrics=['accuracy']
)
```

Here's the same model, but built using the functional API. Notice that they are syntactically almost the same, but the layers are being *passed to* each other after each operation.

Listing 5.2: Simple Functional API CNN

```
input_layer = Input(shape=(32, 32, 3))
x = Conv2D(filters=16,
           kernel_size=(3, 3),
           activation="relu")(input_layer)
x = MaxPool2D(pool_size=(3, 3),
              padding="valid")(x)
x = Conv2D(filters=32,
           kernel_size=(3, 3),
           activation="relu")(x)
x = MaxPool2D(pool_size=(3, 3),
              padding="valid")(x)
```

```
x = Flatten()(x)
output = Dense(units=5,
                activation="softmax")(x)

opt = keras.optimizers.RMSprop(lr=0.0001, decay=1e-6)

functional_model = Model(input_layer, output)

functional_model.compile(
    loss='categorical_crossentropy',
    optimizer=opt,
    metrics=['accuracy']
)
```

If you print the summary of the models you will get this as output.

Listing 5.3: Sequential model summary

```
_________________________________________________
Layer (type)                 Output Shape              Param #
=================================================
conv2d_1 (Conv2D)            (None, 30, 30, 16)        448
_________________________________________________
max_pooling2d_1              (None, 10, 10, 16)        0
_________________________________________________
conv2d_2 (Conv2D)            (None, 8, 8, 32)          4640
_________________________________________________
max_pooling2d_2              (None, 2, 2, 32)          0
_________________________________________________
flatten_1 (Flatten)          (None, 128)               0
_________________________________________________
dense_1 (Dense)              (None, 5)                 645
=================================================
Total params: 5,733
Trainable params: 5,733
Non-trainable params: 0
_________________________________________________
```

Listing 5.4: Functional model summary

```
_________________________________________________
Layer (type)                 Output Shape              Param #
=================================================
```

```
input_1              (None, 32, 32, 3)  0
_________________________________________________
conv2d_3 (Conv2D)    (None, 30, 30, 16) 448
_________________________________________________
max_pooling2d_3      (None, 10, 10, 16) 0
_________________________________________________
conv2d_4 (Conv2D)    (None, 8, 8, 32)   4640
_________________________________________________
max_pooling2d_4      (None, 2, 2, 32)   0
_________________________________________________
flatten_2 (Flatten)  (None, 128)        0
_________________________________________________
dense_2 (Dense)      (None, 5)          645
=================================================
Total params: 5,733
Trainable params: 5,733
Non-trainable params: 0
_________________________________________________
```

5.6 Triplet Loss

As mentioned in Face Recognition section there is another way to recognize employees without implicitly classifying them with CNN producing a class per employee. This method is very similar to the one described above, where compare two images and produce binary output saying if they are pictures of the same person, or no. We can extend this concept a bit more and instead of perform on pairs of images make it take triplets of images (A, P, N) which we will call **anchor**, **positive** and **negative**.

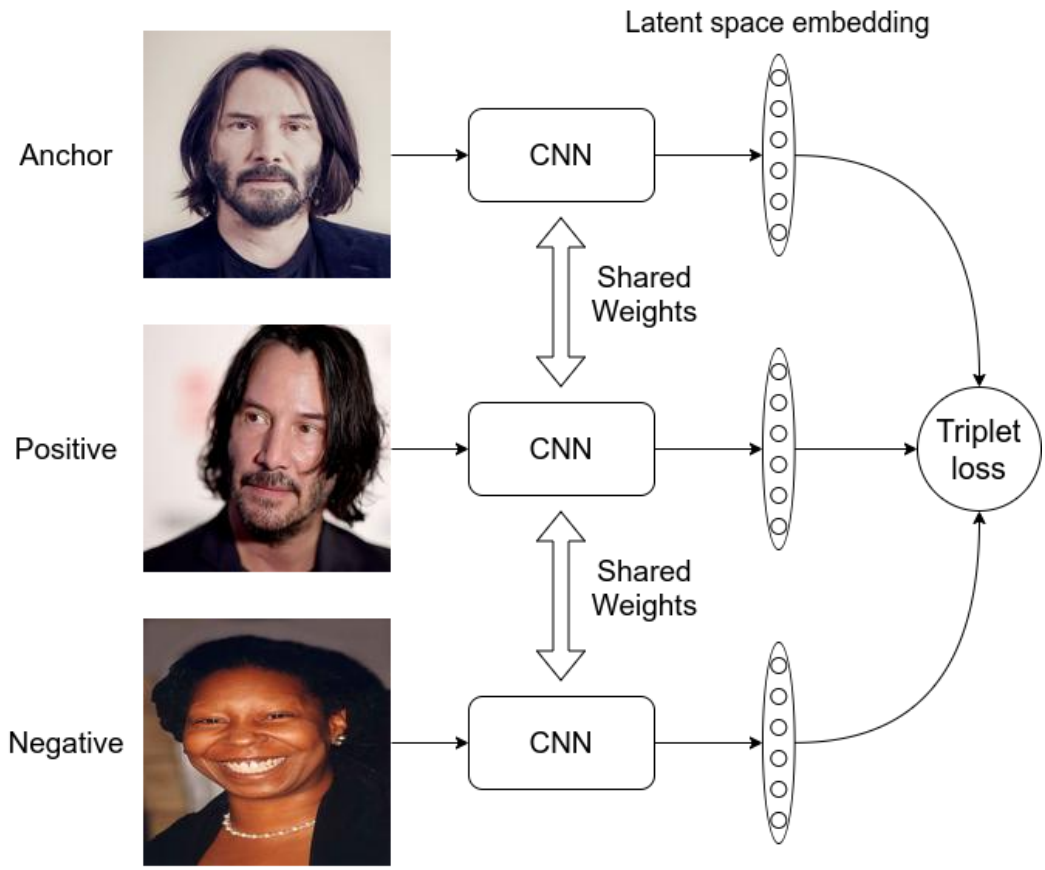


Figure 5.4: Triplet loss diagram.

What is the intuition behind loss functions and their job in ML? They have clear requirements from the models and they reward it whenever the requirements are satisfied and punish the model whenever the requirements are not satisfied.

What triplet loss is asking the model for is to produce such a *Latent space embedding* that similar objects are closer to each other. Basically it makes the model to push apart images with different meaning and pull together images with similar context. In terms of face recognition it will make the model to produce such embedding that different pictures of the same person are going to be close and pictures of different people are going to be further from each other.

5.6.1 Latent space embedding

Latent space is just fancy term of mathematical space. When we train a model, let's say CNN with 128×128 input shape, and we want it to map it's input to 64 dimensional space, for example, in such a way that we preserve as much information as possible then we call this 64 dimensional space latent. The vector in this latent space we call latent space embedding or just embedding. The purpose of this embedding is to represent the original input, but with less parameters. For example the 128×128 image has 16384 parameters, but when we map it to the latent space it is encoded in 64 parameters. If you want to have a model that produces embeddings of cats and dogs the simplest way you can get such a model you can simply train a model to classify dogs and cats and then just drop the last output layer and keep the one before the last layer which you can set initially to be Dense layer with 64 units. Now the output of your model is 64 dimensional vector which you can call latent space embedding.

Back to triplet loss.

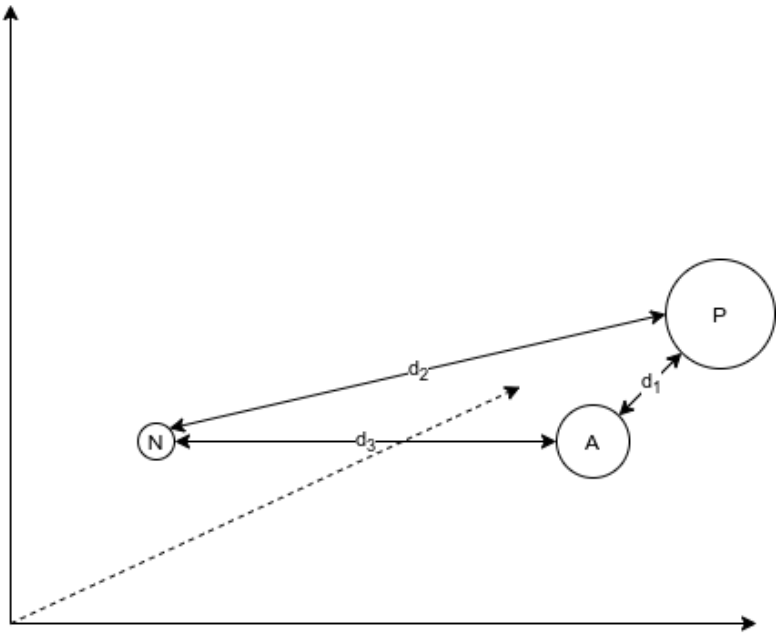


Figure 5.5: Triplet loss illustration.

The ultimate goal for the model to make the loss function happy is to make d_1 as smaller as possible and d_2 and d_3 as large as possible. That's because A and P are points representing the pictures of the same person and

N is someone else who our model has to distinguish between the person on the A and P pictures.

If you are familiar with word embeddings like Word2Vec and GloVe they are trained by exactly the same principle. Words with similar meaning are closer to each other than words with different meaning.

Let's look into the details of triplet loss function. Here's an example of a triplet loss comparing points using Euclidean distance.

$$L(A, P, N) = max(\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha, 0)$$

Let's try to give an explanation for this expression piece by piece.

$$\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2$$

This is the, though the most complex in terms of notation, the simplest part of the function. It just calculates the L^2 norm of the pairs $(A, P), (A, N)$ and subtracts them. L^2 norm is just the mathematical way to calculate the distance between two points. The two most common ways to calculate distance between vectors in ML are the L^1 and L^2 norms. L^1 is also called Manhattan distance and L^2 is simply Euclidean distance. On the diagram below you can see why L^1 is called Manhattan distance (It's because in order to get from p_1 to p_2 you walk either horizontally or vertically just like walking in Manhattan).

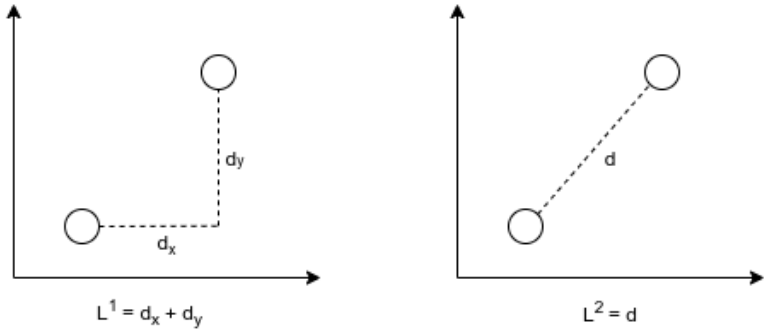


Figure 5.6: L^1 and L^2 distances

Simply said the signs between $\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2$ tell us that $\|f(A) - f(P)\|^2$ has to be as small as possible, because it's positive and it's going to contribute to the loss function value.

$\|f(A) - f(N)\|^2$ has to be as big as possible, because it's negative and it's going to be subtracted from the loss function value.

If we again recall what the two expressions mean we see that it's exactly the same as making the anchor and the positive example close to each other and the anchor and negative apart from each other.

The next piece in the loss function expression is the

$$+\alpha$$

Let's consider what will happen if there is no $+\alpha$. Assume that our model is smart enough (this is not ridiculous assumption) to come with the simplest strategy to just output the same encoding for all the images so that $f(A) = f(P) = f(N)$, so then the Loss $L(A, P, N) = 0$ which is the global minimum of the loss function. This is definitely what we are aiming for. If you are familiar with Support Vector Machines (SVM) they enforce exactly the same restriction for the two classes to have at least α margin between them, so that when they are separated by a linear model the model itself is no closer than $\frac{\alpha}{2}$ to each element of the classes. The same is applied here.

Here's geometric explanation:

On fig 5.5 d_1 is the distance between the anchor and the positive example. d_3 is the distance between the anchor and the negative example.

The loss function written in this terms is:

$$L(A, P, N) = max(d_1 - d_3 + \alpha, 0)$$

d_1 and d_3 considered as vectors can be added to each other and the result will be the vector d_2. Recall how we add vectors with the figure below. *the red vector is shifted only for visibility.*

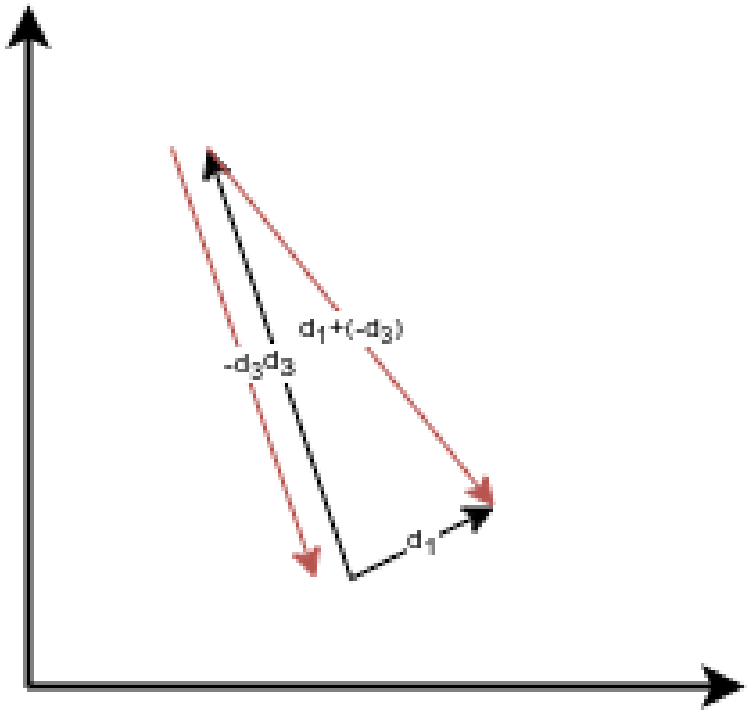


Figure 5.7: Adding vectors diagram.

So you see that when you add the vectors $-d_3$ and d_1 the result is actually the vector d_2 on figure 5.5. That's why when we want the difference of the length of the vectors d_1 and d_3 to be at least α this results in that the length of d_2 which is the length of the vector connecting the positive and the negative example to be also no less than α.

You probably now wonder why do we need the max function? We can achieve more than only α difference between the positive and the negative example if we remove the max function and instead let the optimizer to pull apart the two embeddings infinitely. The answer is that we can do it. But this would result in really unstable optimization problem and probably scales on each of the N axis of the N-dimensional encoding that are hard to be compared and optimized. That's why we aim for just α difference and restrict the loss function to have global minimum with value 0. Actually this does not restrict our model, or make it less powerful. We are totally okay with distance between the samples $\alpha = 1$. Computers are pretty good with relatively small numbers and if the embeddings are well clustered by samples and the distance between the clusters is α this is still perfect for us.

This is the reason we need the max function and the 0 in $max(..., 0)$.

Let's now get back to FaceNet.

FaceNet is a deep model trained on millions of face images and achieving incredible performance on famous face image datasets like *Labeled Faces in the Wild (LFW)* and *On YouTube Faces DB*. FaceNet is trained using triplet loss and produces 128 dimensional vector called face embedding.

We can download one of these datasets, build the architecture and train a FaceNet model hoping to achieve the same accuracy Google has achieved when training this model with tons of data on clusters. You will also cross your fingers to have implemented everything correctly, because if you have to start the training again it will probably take you another week.

You will almost always find yourself using pretrained models as part of your ML systems. You can either use the pretrained model to generate emebeddings which you can use for your own model, or use them as submodel of your model etc. In the examples for this book you will find face recognition system developed using FaceNet and pretrained model trained by google, which we just load using Keras and predict with it on any face image we want.

Keras provides really convinient way to serialize machine learning models, architectures and weights, so if you want to share models, version control them, or store them somewhere Keras provides a way to do this in one function call. You can choose to save either the weights only or the weights + the architecture, so if someone wants to load your model later they don't have to implement the exactly same architecture as you to load the weights on it. They can just do

Listing 5.5: Keras model loading.

```
from keras.models import load_model

model = load_model('facenet_keras.h5')
embedding = model.predict(employee_image)
```

In 3 lines you can load and predict using state of the art model. That's one of the great powers of Keras!

Before we end up this chapter let's examine complete siamese CNN with distance based pair loss function. We will train a model on the MNIST dataset and that model will produce 128 dimensional embedding as output.

Listing 5.6: Siamese CNN with contrasive loss.

```
def contrastive_loss(y_true, y_pred):
    margin = 1
    square_pred = K.square(y_pred)
    margin_square = K.square(K.maximum(margin - y_pred,
                                       0))
    return K.mean(y_true * square_pred +
                  (1 - y_true) * margin_square)

def create_base_network(input_shape):
    input = Input(shape=input_shape)
    x = Conv2D(filters=8, kernel_size=(3, 3),
               activation='relu')(input)
    x = MaxPool2D(pool_size=(3, 3), padding="valid")(x)
    x = Conv2D(filters=16, kernel_size=(3, 3),
               activation='relu')(x)
    x = MaxPool2D(pool_size=(3, 3), padding="valid")(x)
    x = Flatten()(x)
    x = Dense(128, activation='relu')(x)
    x = Dense(128, activation='relu')(x)
    return Model(input, x)

input_shape = (28, 28, 1)

processed_a = base_network(input_a)
processed_b = base_network(input_b)

distance = Lambda(euclidean_distance)([processed_a,
                                       processed_b])

model = Model([input_a, input_b], distance)

rms = RMSprop()
model.compile(loss=contrastive_loss,
              optimizer=rms,
              metrics=[accuracy])
```

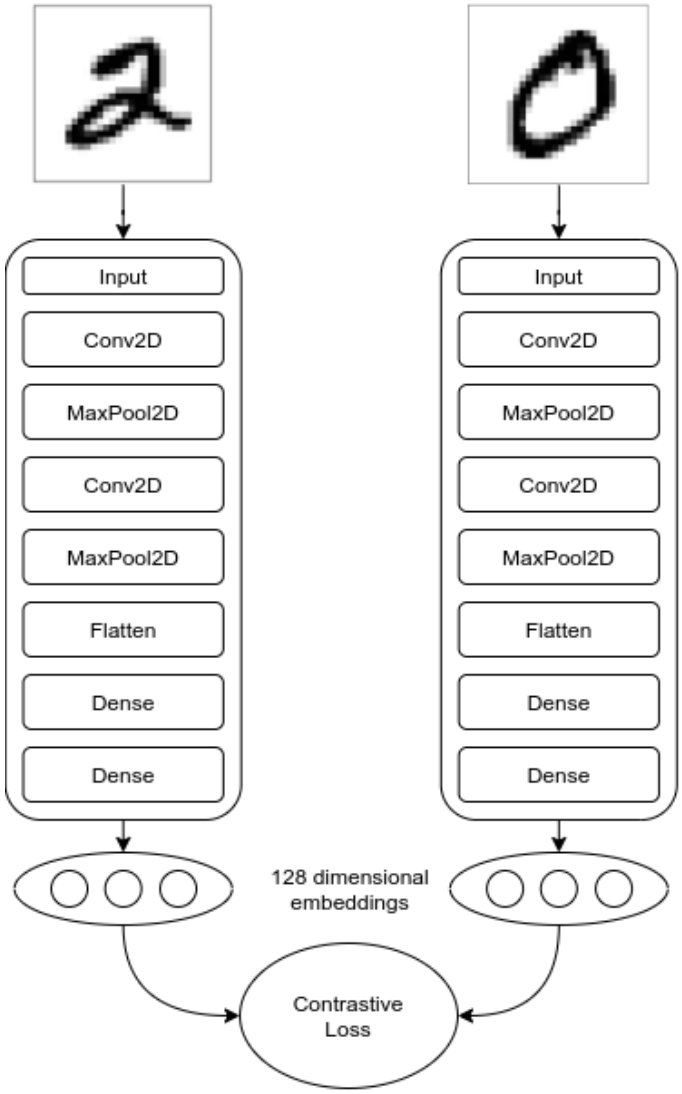


Figure 5.8: Diagram of the architecture of the Siamese CNN example.

It's important to notice that this model does not share weights. There are two models defined with the **base_function** which are later combined with the **Lambda** layer.

Also, though we have put the embeddings as another box this is actually the last Dense layer which consists of 128 units which we consider the embeddings that the model produces.

After 12 epochs on NVIDIA GeForce GTX 1060 this architecture achieved the following results:

Accuracy on training set: 98.85

Accuracy on test set: 98.19

6. Image segmentation. Transposed convolutions. U-Net.

So far you know what image classification is. Image segmentation is another fundamental task in the field of computer vision. It takes complexity on a next level, where we train a model not to predict a class given input image, but to predict a class for each pixel of the input image.

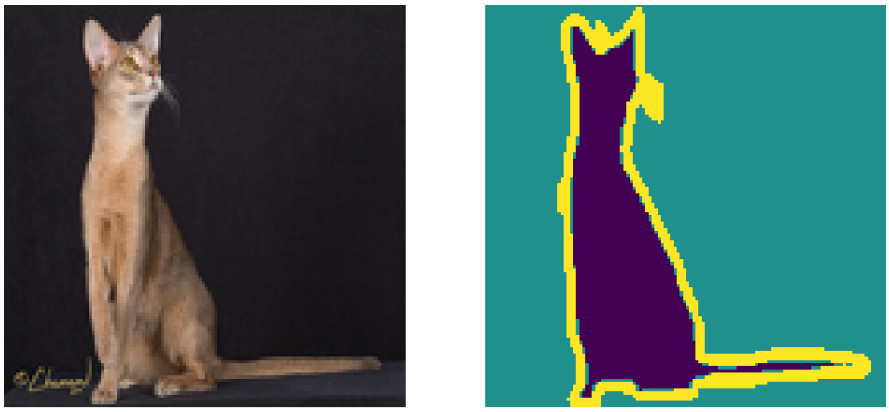


Figure 6.1: Sample segmented image from the The Oxford-IIIT Pet Dataset.

Image segmentation is important for many reasons, but generally a model that is capable of segmenting image pixel by pixel provides more information. You not only know that there is a cat on the picture. You know which pixels of the image the cat has covered. Image segmentation has many applications, but one of the most common and noticeable ones you will come across are health care where given image of a human to segment the organs on v-ray image, or given image of an organ to segment it, localize tumors, measure tissues volume and so on and self-driving cars where given image of outdoor scene you segment it in order to distinguish between objects like the road, humans, trees and other cars.

In order to solve this problem we will cover several new loss functions,

metrics and architectures which enable achieving great performance on this task.

6.1 U-Net

The U-Net was developed by Olaf Ronneberger et al. for Bio Medical Image Segmentation. The architecture consists of two main components called encoder and decoder. You are already familiar with encoder like parts and this encoder is just like the encoders we talked about, but instead it be the output of the network it is in the middle. As the encoders you are familiar with, this one is used to capture the context and encode it in 1024 dimensional latent space.

The second component is the decoder. It's main purpose is to take the value of the latent space and return it to the initial space where the input image is sampled from, or simply said, to upsize the latent space and turn it back to image.

So far we have mentioned only convolutions, maxpooling and other operations which all result in downsizing, so how would we came up with a model that upsamples it's input?

Upsampling image is quite well known problem and has many solutions for example *nearest neighbor interpolation, Bi-linear interpolation* and more advanced ways like transposed convolutions which we will focus on.

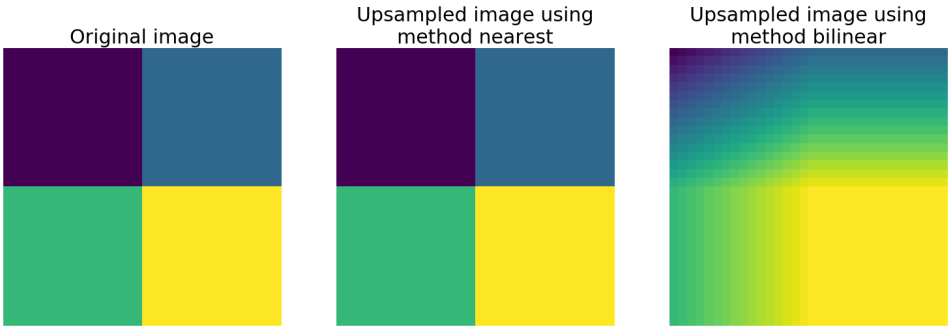


Figure 6.2: Upsampling example.

Keras provides 2 common ways to implement upsampling of an image.

6.2 UpSampling2D

The first way is to use the **UpSampling2D** layer with interpolation method *nearest*.

```
keras.layers.UpSampling2D(size=(2, 2),
                          data_format=None,
                          interpolation='nearest')
```

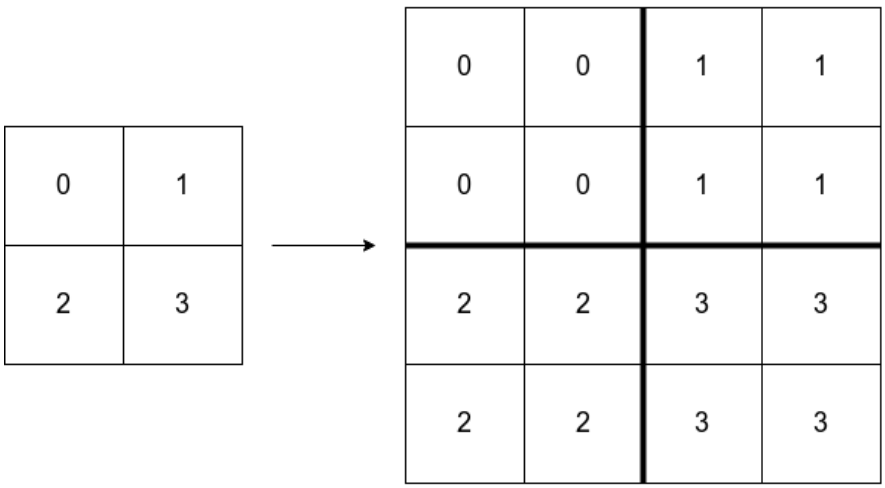


Figure 6.3: Upsampling example using nearest neighbor method.

Or you can use the **UpSampling2D** layer with interpolation method *bilinear*.

```
keras.layers.UpSampling2D(size=(2, 2),
                          data_format=None,
                          interpolation='bilinear')
```

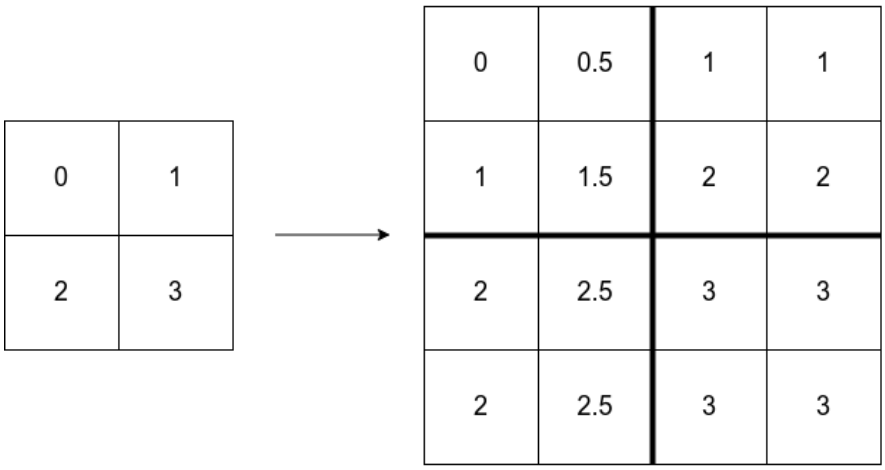


Figure 6.4: Upsampling example using bilinear method.

On fig 6.2 you can see what do both the methods produce as a result on a 2×2 input image upsized to 32×32.

6.3 Transposed convolution.

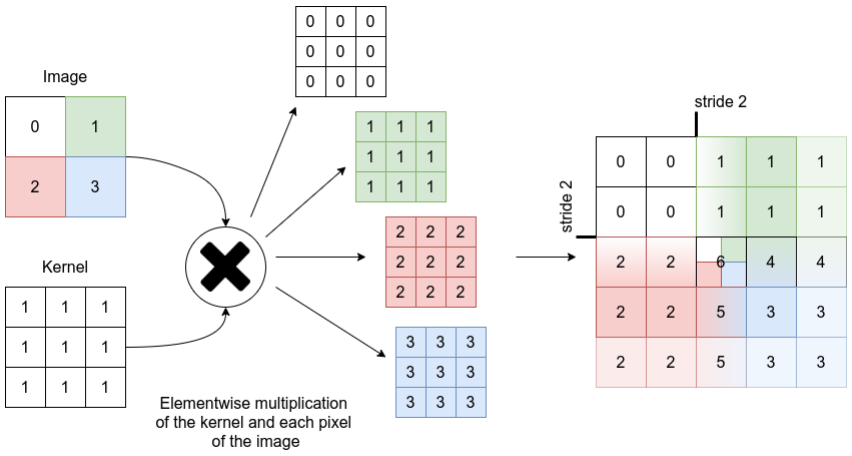


Figure 6.5: Diagram of the transposed convolution operation

Transposed convolutions, rather than having a deterministic way to upsample given image like the methods mentioned above, benefits from machine learning's fundamental concept of learning things instead of assuming them.

The operation itself is not that complex, but is very powerful and when used in deep models results in much better performance.

Just like convolutions, transposed convolutions are based on kernels that are initially randomly initialized and then are being optimized in order to find the best weights to perform the required task.

In several steps transposed convolution is performed as follows:

We have input image and matrix of predefined size for example $(2, 2)$. We perform elementwise multiplication of each pixel of the image and the kernel. The result is as many matrices of size the size of the kernel as pixels the image itself has. Then we will use these matrices to form the upsampled image in the following way: we will combine them, for example summing the overlapping regions starting from the upper left corner and placing each matrix and then moving *stride* steps to the right until we end the line, then go *stride* steps down and repeat the operation. If the result is bigger than the target size we will crop it by dropping the extra rows and columns as in the example above (the transparent row and column). Finally the result is passed through an activation function to produce the output of the transpose convolutional layer.

That's it! Though it sounds complex when it's written, if you give a second look to the diagram above you will see that it's actually not that complex.

Now when you are familiar with all the building blocks of U-Net we can look at it in detail.

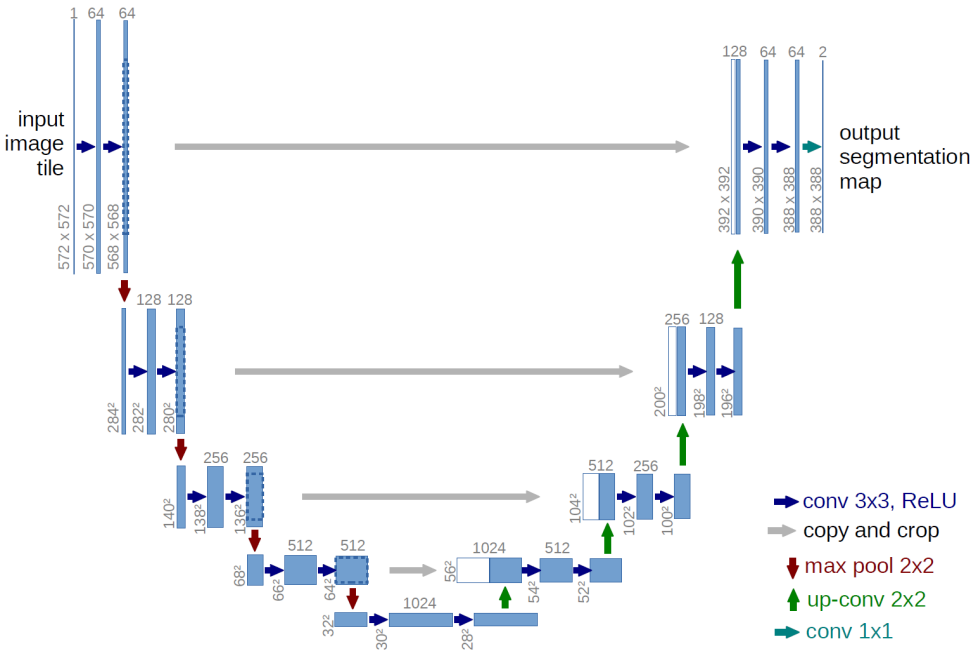


Figure 6.6: U-Net's architecture as it was described initially in it's paper.

One of the important parts of U-Net is the connections between the symmetrical parts of the network. The first layer is connected with the last one, the second with the one before the last one and so on until we reach the layer before the middle layer that is connected to the layer after the middle layer.

Keep in mind that people usually call an architecture U-Net though it is not exactly the one you see above. U-Net is called so because of the encoder-decoder architecture that when plotted on diagram looks like U, so people usually refer to all U-like architectures as U-Net. You will see below similar architecture that we will train to segment images and will call U-Net, though it is a smaller version of the one above.

6.4 Training U-Net on The Oxford-IIIT Pet Dataset.

Here's smaller version of U-Net that we will use to train on The Oxford-IIIT Pet Dataset in order to segment images. The Oxford-IIIT pet dataset consists of 37 categories of pet images with roughly 200 images for each class.

All images have an associated ground truth annotation like the one of fig 6.1. The ground truth trimap images consist of 3 classes:

Class 1 : Pixel belonging to the pet.
Class 2 : Pixel bordering the pet.
Class 3 : Surrounding pixel.

Below is complete implementation of the U-Net architecture.

Listing 6.1: U-Net model in Keras

```
inputs = Input((128, 128, 3))

c1 = Conv2D(64, (3, 3), activation='relu',
            padding='same')(inputs)
c1 = Dropout(0.2)(c1)
c1 = Conv2D(64, (3, 3), activation='relu',
            padding='same')(c1)
p1 = MaxPooling2D((2, 2))(c1)

c2 = Conv2D(128, (3, 3), activation='relu',
            padding='same')(p1)
c2 = Dropout(0.2)(c2)
c2 = Conv2D(128, (3, 3), activation='relu',
            padding='same')(c2)
p2 = MaxPooling2D(pool_size=(2, 2))(c2)

c3 = Conv2D(256, (3, 3), activation='relu',
            padding='same')(p2)
c3 = Dropout(0.3)(c3)
c3 = Conv2D(256, (3, 3), activation='relu',
            padding='same')(c3)

u1 = Conv2DTranspose(128, (2, 2), strides=(2, 2),
                     padding='same')(c3)
u1 = concatenate([u1, c2])
c4 = Conv2D(128, (3, 3), activation='relu',
            padding='same')(u1)
c4 = Dropout(0.2)(c4)
c4 = Conv2D(128, (3, 3), activation='relu',
            padding='same')(c4)

u2 = Conv2DTranspose(64, (2, 2), strides=(2, 2),
                     padding='same')(c4)
```

```
u2 = concatenate([u2, c1])
c5 = Conv2D(64, (3, 3), activation='relu',
            padding='same')(u2)
c5 = Dropout(0.2)(c5)
c5 = Conv2D(64, (3, 3), activation='relu',
            padding='same')(c5)

outputs = Conv2D(3, (1, 1),
                 activation='softmax')(c5)

model = Model(inputs=[inputs], outputs=[outputs])
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['acc', 'mse',
                       jaccard, dice_coef])
```

The resulting model has 1,862,979 params in total.

After training it for 15 epochs it achieves near 90% accuracy, 0.88 Jaccard coefficient, 0.85 dice coefficient on the training data and 88% accuracy, 0.88 Jaccard coefficient, 0.85 dice coefficient on the validation set.

6.4.1 Jaccard similarity coefficient

The Jaccard coefficient is used to measure similarity between two sets. It is defined as

$$J(A,B) = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

Which is basically the area of overlap over the area of union. Visually it's much easier to understand.

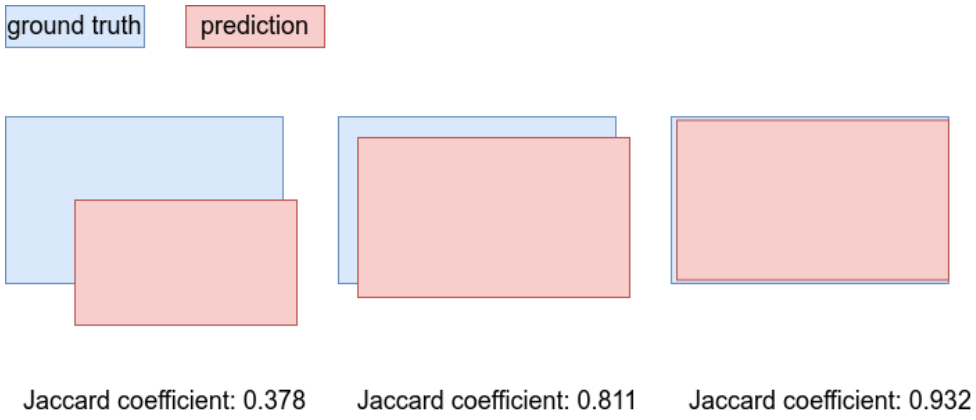


Figure 6.7: Jaccard coefficient for different configurations of the sets A and B.

6.4.2 Dice coefficient

Another very similar metric that is usually used in image segmentation is the dice coefficient. It is defined as

$$D(A, B) = \frac{|A \cap B|}{|A| + |B|}$$

The dice coefficient is not much different from Jaccard. Actually if you have one of them you can compute the other one using the following formula.

$$J = \frac{D}{2 - D}$$

$$D = \frac{2J}{1 + J}$$

7. Understanding CNNs. Filters visualization.

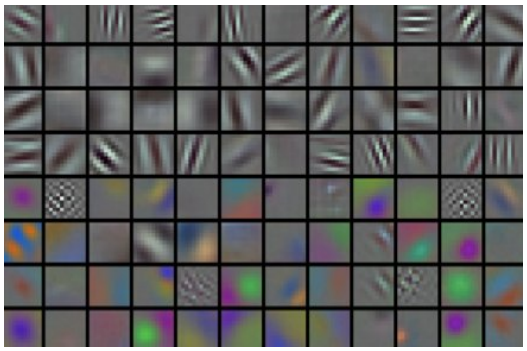


Figure 7.1: CNN filters

So far you have the understanding of how and why filters look, but still for various reasons it is useful to actually touch a raw filter. The more you decompose and explore an object the more you understand it.

What does actually a filter look for in a image, once it is trained?

7.1 Activation maps

Our goal here is to somehow understand what our model is looking for in order to distinguish between different inputs.

Let's again consider the MNIST dataset as example for the sake of simplicity. We will use the model that we trained previously.

Our input will be the following two images

Figure 7.2: Sample of the digit 5 from the MNIST dataset

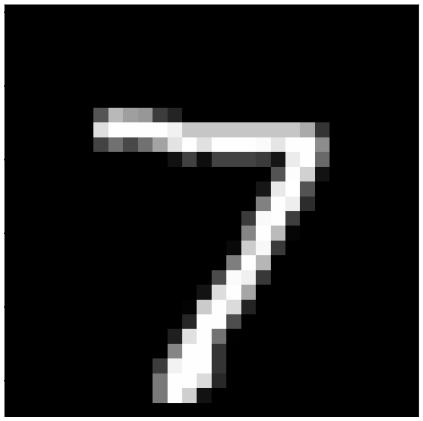


Figure 7.3: Sample of the digit 7 from the MNIST dataset

Let's pass them through the network and take the feature map of each layer and each feature. What this means is simply take the result after applying each of the 32 filters of layer 1 and plot them. The result images are called feature maps. The assumption is that higher value pixels contribute to the outcome more than pixels with lower value and that's why we say the network pays attention to them.

Here's a heatmap of the activations of layer 1 of the digit 5 image.

Figure 7.4: Activations as a heatmap overlaid on the image

Blue color means lower value and red means high value. You can see that the network focuses on certain edges and curves that look like typical for the digit 5. There is one feature map that has activated on the flat area around the digit separating the digit pixels from the background. It has lots of high value pixels, so it must contribute a lot for the final result.

This analysis is not only important and useful for understanding the network and it's decision, but for performance analysis. If you notice lots of similar activations you can assume that the network would not perform

worse even with less filters.

Below you can see the same layer feature maps for the digit 7.

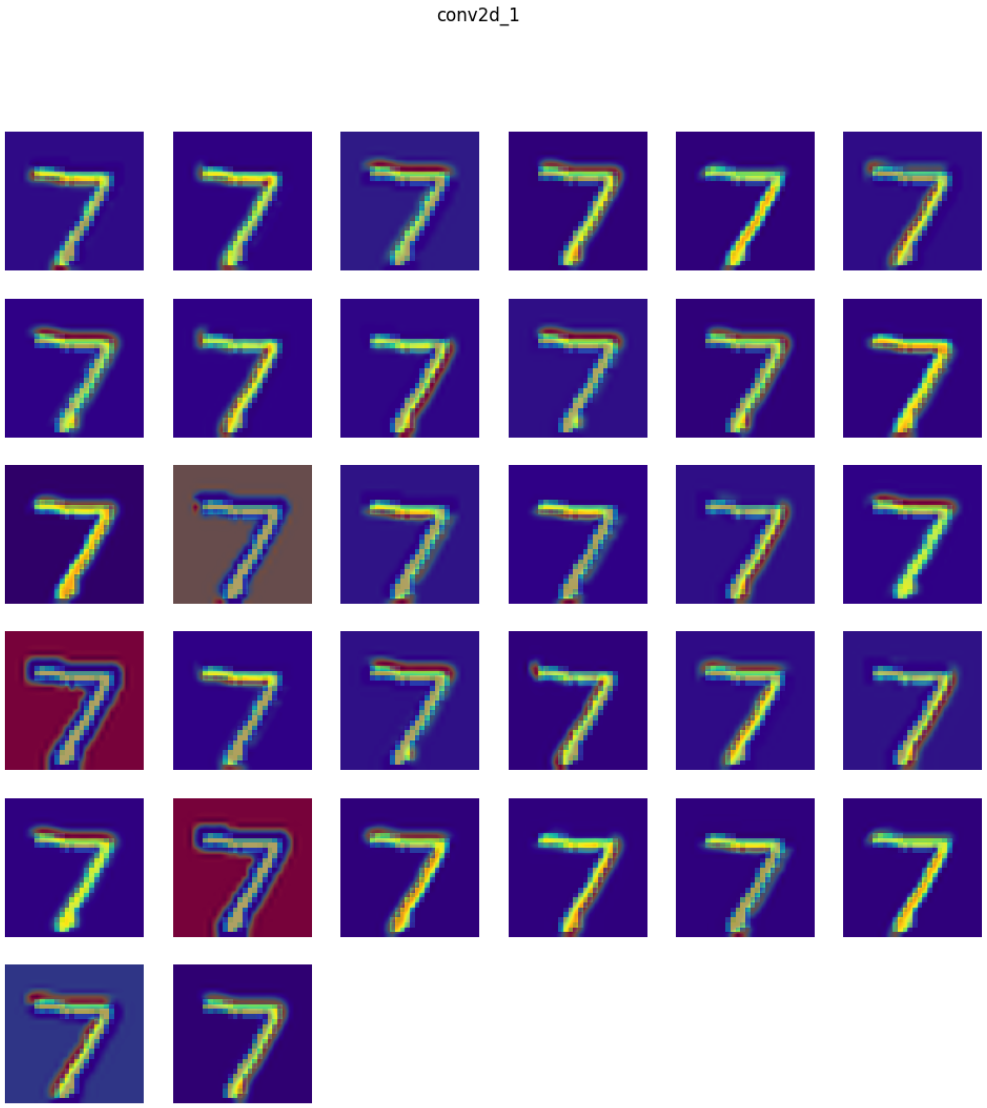


Figure 7.5: Activations as a heatmap overlaid on the image

It's clear that the network focuses on two straight lines to classify the digit 7. The horizontal line on the top of the digit and the diagonal line of the digit.

Below are the activation heatmaps of the second layer for both of the digits.

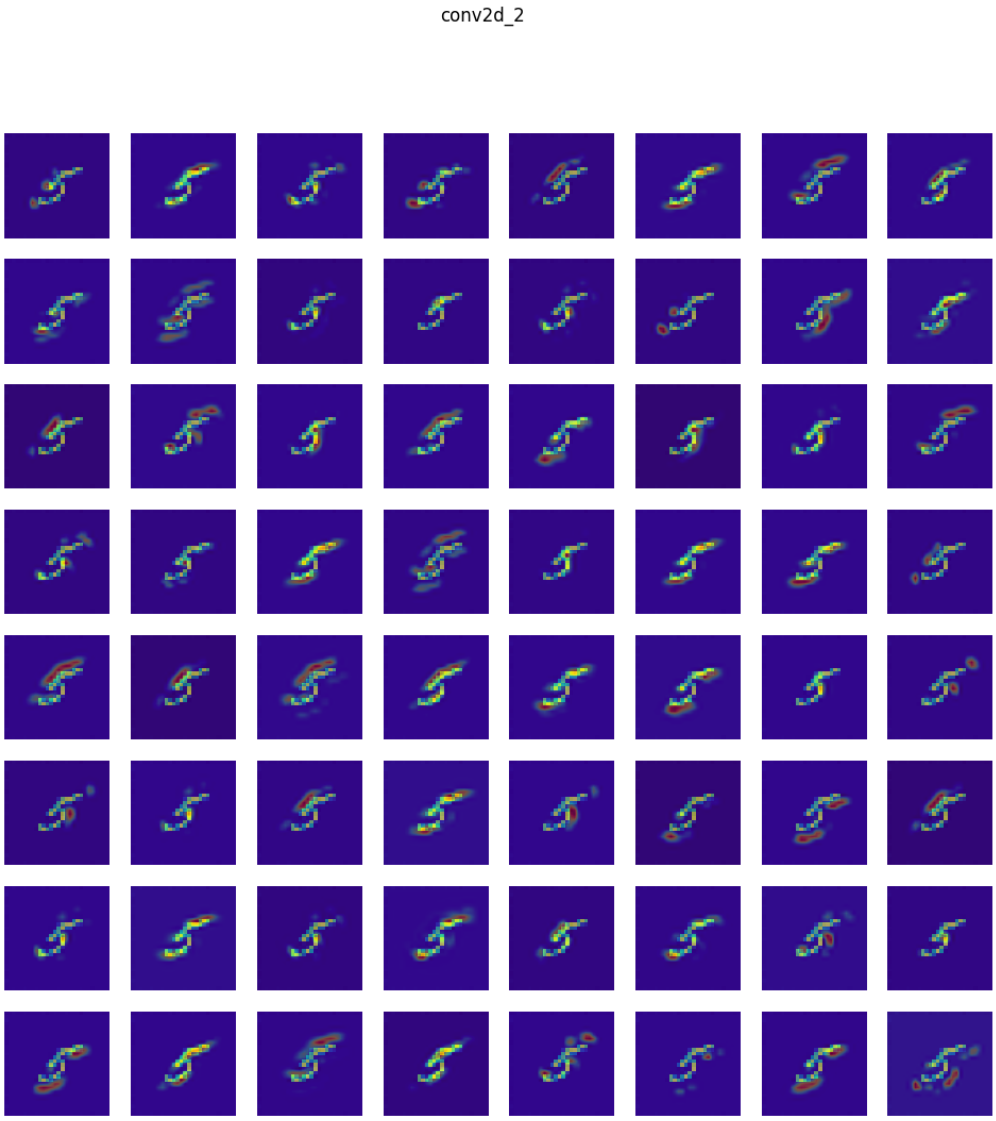


Figure 7.6: Activations as a heatmap overlaid on the image

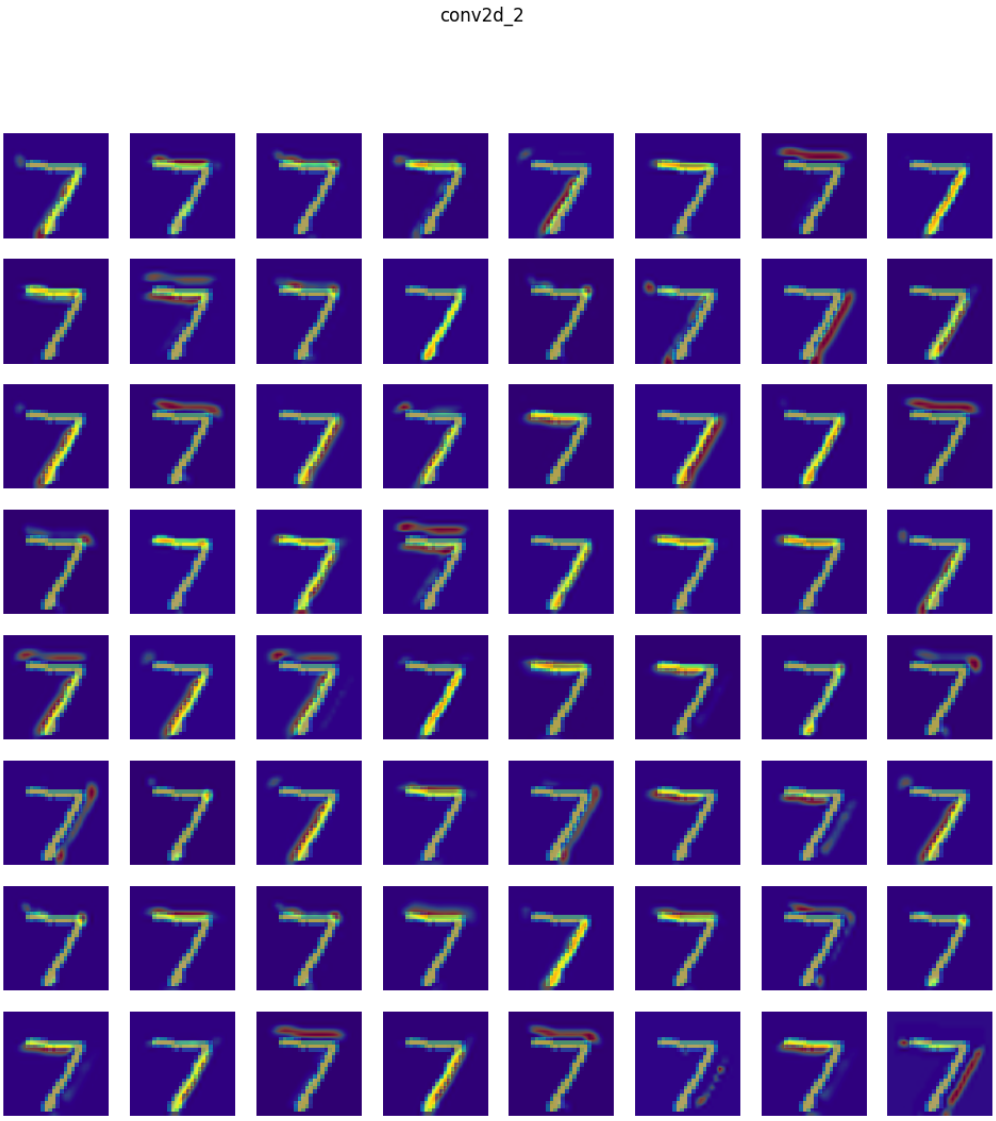


Figure 7.7: Activations as a heatmap overlaid on the image

It's arguable if there is any valuable information in the second layer. Noticing this you may come with a good experiment and evaluate the performance of the network without the second layer, if this makes sense for your application. If performance is critical to you such an optimization might be very valuable.

The End.

8. References:

1. Siegelmann, Sontag (1992), On The Computational Power Of Neural Nets.

2. V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," Haifa, 2010, pp. 807–814.

3. Yarin Gal and Zoubin Ghahramani, "Bayesian Convolutional Neural Networks with Bernoulli Approximate Variational Inference" 2015 arXiv:1506.02158

4. LeCun, Y. & Cortes, C. (2010). MNIST handwritten digit database.

5. Alex Krizhevsky and Vinod Nair and Geoffrey Hinton, "CIFAR-10 (Canadian Institute for Advanced Research"

6. Deng, J. et al., 2009. Imagenet: A large-scale hierarchical image database. In 2009 IEEE conference on computer vision and pattern recognition. pp. 248–255.

7. Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton, "ImageNet classification with deep convolutional neural networks" 2012 NIPS'12: Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1December 2012 Pages 1097–1105

8. Schroff, F., Kalenichenko, D. & Philbin, J., 2015. Facenet: A unified embedding for face recognition and clustering. In Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 815–823.

9. Olaf Ronneberger, Philipp Fischer, Thomas Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation", 2015

10. Omkar M Parkhi and Andrea Vedaldi and Andrew Zisserman and C. V. Jawahar, "The Oxford-IIIT Pet Dataset"