Exploiting the spatial structure of unstructured data:

Structured vs Unstructured data:

Structured data is data that has been predefined and formatted to a set structure before being placed in data storage, which is often referred to as schema-on-write. The best example of structured data is the relational database: the data has been formatted into precisely defined fields, such as credit card numbers or address, in order to be easily queried with SQL. Unstructured data on other hand, is data stored in its native format and not processed, which is known as schema-on-read. It comes in a myriad of file formats, audio (speech, environmental, music), images (digital cameras, medical imaging, satellite imagery), Industrial and IoT sensor data, email, social media posts and chats.

Now, the definition we use when we talk about data for Machine learning, is relatively similar, except that we usually make the distinction between the two classes of data, by saying structured data is one that is stored in tabular like format (rows and columns), and reordering it based on different feature doesn't affect its meaning. Unstructured data (images, audio) on the other hand, is one that loses its meaning when it gets reordered.

Now, why making this distinction?

Historically it has been much harder for computers to make sense of unstructured data compared to structured data. In fact, we as human being are very good at understanding unstructured data, I am talking about acoustic and visual ques (audio and images), and so one of the most exciting things about the rise of neural networks is that thanks to neural networks computers are now much better at interpreting unstructured data as well. And this creates opportunities for many new exciting applications that use speech recognition image recognition natural language processing, much more than was possible even just ten years ago. And as a personal statement I think we, human have a natural empathy to understanding unstructured data, you might hear about NNs successes on unstructured data more in the media because it's just cool when something we build (computer) learn to interpret and recognize the content of these types of unstructured data that we as human naturally good at. When a computer recognizes a cat, we all like that, because we all know what that means.

Up to this point we have only been discussing fully connected feed-forward neural networks, and the model we have developed in the previous chapter did very well on the task of handwritten image classification, since we've obtained a classification accuracy up to 98 %, with a relatively simple model, and that's of course after employing some optimization (better cost function and better weight initialization approach), and regularization techniques. And we can summarize its working machinery in few lines: A vector is received as input and is multiplied with a matrix to produce an output, to which a bias vector is usually added - this first part is known as affine transformation - before passing the result through a non-linearity. This is applicable to any type of

input, not just to image data, a sound clip or an unordered collection of features: whatever their dimensionality, their representation can always be FLATTENED into a vector before the transformation [30]

Images, sound clips and many other similar kinds of data have an intrinsic structure, despite that they belong to unstructured data class. More formally, they share these important properties:

- They are stored as multi-dimensional arrays.
- They feature one or more axes for which ordering matters (e.g., width and height axes for an image, time axis for a sound clip).
- One axis, called the channel axis, is used to access different views of the data (e.g., the red, green and blue channels of a color image, or the left and right channels of a stereo audio track).

Convolutional Neural Networks (CNNs):

It turns out that fully connected feed-forward networks are not the best choice for these types of unstructured data that has an implicit structure, since they do not take into account the implicit spatial structure present in the input data. In fact, all the axes are treated in the same way and the topological information is not exploited. For instance, they treat input pixels which are far apart and close together on exactly the same footing. Such concepts of spatial structure must instead be inferred from the training data. What if, instead of starting with a NN architecture that is tabula rasa (agnostic to the spatial structure), we start with a NN architecture that take into account the spatial structure. Convolutional Neural Networks, use a special architecture which is particularly well-adapted for tasks like pattern recognition and classification. You can think of CNN, as being a NN that has some type of specialization for being able to pick up or detect pattern and make sense of them [1-3, 30].

Convolutional Neural Networks or CNNs, are a specialized kind of neural network for processing data that has a known, grid-like topology. Examples include time-series data, which can be thought of as a 1D grid taking samples at regular time intervals, and image data, which can be thought of as a 2D grid of pixels [2]. Although CNNs have been used as early as the nineties to solve character recognition tasks (Le Cun et al., 1997), their current widespread application is due to much more recent work, when a deep CNN was used to beat state-of-the-art in the ImageNet image classification challenge (Krizhevsky et al., 2012) [30].

The Convolution Operation:

As the name “convolutional neural network” implies CNN employs the convolution mathematical operation - In fact with a slight modification as we will see shortly. Convolution is a specialized kind of linear operation. Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers [2]

A convolution is an operation on two functions of a real-valued argument, it is defined as the integral of the product of the two functions after one is reversed and shifted. The integral is evaluated for all values of shift. It is often notated with an asterisk ($*$) operator, where:

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

Usually, when we work with data on a computer, data will be discretized. So, the discrete version of the convolution operation:

$$s(n) = (f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m] = \sum_{m=-\infty}^{\infty} f[n-m]g[m]$$

Algebraically is the sum of Hadamard product (aka the element-wise product) of the kernel parameters with the corresponding receptive field in the input.

Note: In CNN terminology, the first argument (in this example, the function f) to the convolution is often referred to as the input and the second argument (in this example, the function g) as the filter (kernel). The output is sometimes referred to as the feature map.

From practical perspective, convolution can be seen as the measure of overlap between two signal as one slides over the other as demonstrated in (fig below). The output can be thought as the blend of the two signals.

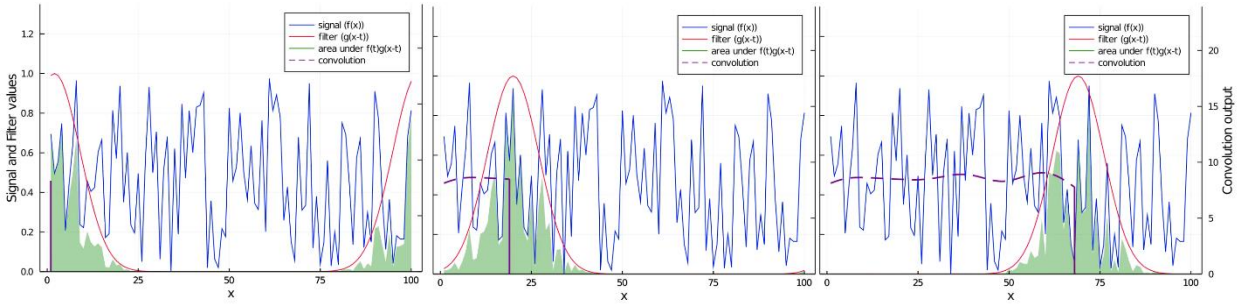

Figure 1 One-dimensional convolution of a Gaussian filter with a random signal

In the above figure, the output of the convolution operation doesn't resemble the original signal. In fact, it doesn't look anything like the original signal, that's because we applied a relatively wide Gaussian filter, so that the input signal got heavily filtered. Because, the signal above is a one-dimensional time series data, filtering in the situation means filtering out the high frequencies, but if the input was two-dimensions like image data, the effect would be a blurring effect, if the filter was two-dimensional Gaussian filter.

Cross-correlation

Similar operation to the convolution is Cross-correlation (denoted by a star $\star$), where the only difference is, one of the functions is first flipped about the y-axis before performing the standard convolution, so that the cross-correlation of f and g is f(x)*g(-x). Note, when using cross-correlation we lose the commutativity property, that's:

$$s(t) = (f \star g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t+\tau)d\tau \neq \int_{-\infty}^{\infty} f(t+\tau)g(\tau)d\tau$$

Note: the convolution operation that I've just demonstrated above is heavily used in many fields such: digital signal processing, image processing, engineering, physics, computer vision and differential equations just to name a few. The main difference between the applications of the convolution in these fields and in machine learning is that, the filters (or kernels) parameters are learned by the network during the learning phase.

In machine learning applications, the input is usually a multidimensional array of data and the kernel is usually a multidimensional array of parameters that are adapted by the learning algorithm. And we often use convolutions over more than one axis at a time. For example, if we use a two-dimensional image I as our input, we probably also want to use a two-dimensional kernel

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_m I(m,n)I(i-m, j-n)$$

Convolution in any number of dimensions is commutative just like in one-dimension, meaning we can equivalently write: $S(i,j) = (I * K)(i,j) = \sum_m \sum_m I(i-m, j-n)K(m,n)$

While the commutative Property is mathematically handy, it is usually not an important property for a neural network implementation. Instead, most NN libraries implement the cross-correlation and call it convolution, which is the same as convolution but without flipping the kernel.

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_m I(i+m, j+n)K(m,n)$$

The CNN architecture is somehow reminiscent to the organization of the Visual Cortex. Individual neurons respond to stimuli, only in a restricted region of the visual field known as the Receptive Field. A collection of such fields overlap to cover the entire visual area.

- Local Receptive Fields: a small region of the input image (a little window on the input), covered by the kernel. The local receptive field (kernel, or filter) moves to the right with a certain Stride Value till it parses the complete width. Moving on, it hops down to the beginning (left) of the image with the same Stride Value and repeats the process until the entire image is traversed (Fig.).

- Shared Weights, Shared Biases: all the neurons in the feature map (output of the convolutional layer), will use the same collection of weights and biases (same filter). This means that all the neurons in the first hidden layer detect exactly the same feature*, just at different locations in the input image.

Note: think of the feature detected by a neuron as the kind of input pattern that will cause the neuron to activate: it might be an edge in the image, for instance, or some other type of shape.

For this reason, we sometimes call the map from the input layer to the hidden layer a feature map. We call the weights defining the feature map the shared weights. And we call the bias defining the feature map in this way the shared bias. The shared weights and bias are often said to define a kernel or filter.

- Filter (aka Kernel): is the matrix of the weights + bias, that will be applied to the local receptive field, to produce the output of the convolutional layer.

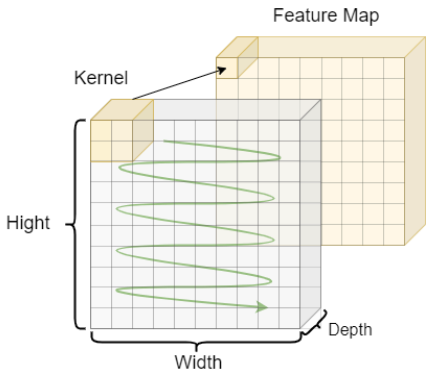

Figure 2 Standard convolution

Pooling Layers: usually used immediately after convolutional layers to simplify the information in the output from the convolutional layer. Normally, there is pooling layer for each feature map that prepares a condensed feature map, for instance each unit in the pooling layer, may summarize a region of say(2*2) in the previous layer [1].

There are two types of Pooling: Max Pooling and Average Pooling (aka L2 pooling). Max Pooling returns the maximum value from the portion of the image covered by the Kernel. On the other hand, Average Pooling returns the average of all the values (square root of the sum, to be prices) from the portion of the image covered by the Kernel.

Fig Max and Average pooling

We can think of max-pooling as a way for the network to ask whether a given feature is found anywhere in a region of the image. It then throws away the exact positional information. The intuition is that once a feature has been found, its exact location isn't as important as its rough location relative to other features [1].

Max Pooling can also be thought as performing Noise Suppressing along with dimensionality reduction. Where, it discards the noise (activations) of nearby neurons and just returns the most dominant feature in the kernel. On the other hand, Average Pooling simply performs dimensionality reduction. Thus, Max Pooling usually is the default option for pooling.

The Convolutional and the Pooling layer are usually counted as a single layer in CNN. Depending on the size and the complexities of the input, the number of such layers may need to increase.

Putting it all together

Yes, convolutional layers enable the model to extract features present in the input, but to make the final prediction, the model needs to put the pieces of the puzzles together, and for those one or more fully connected layers are usually used. Where the output of the final convolutional layer is flattened before being fed to those fully connected layers.

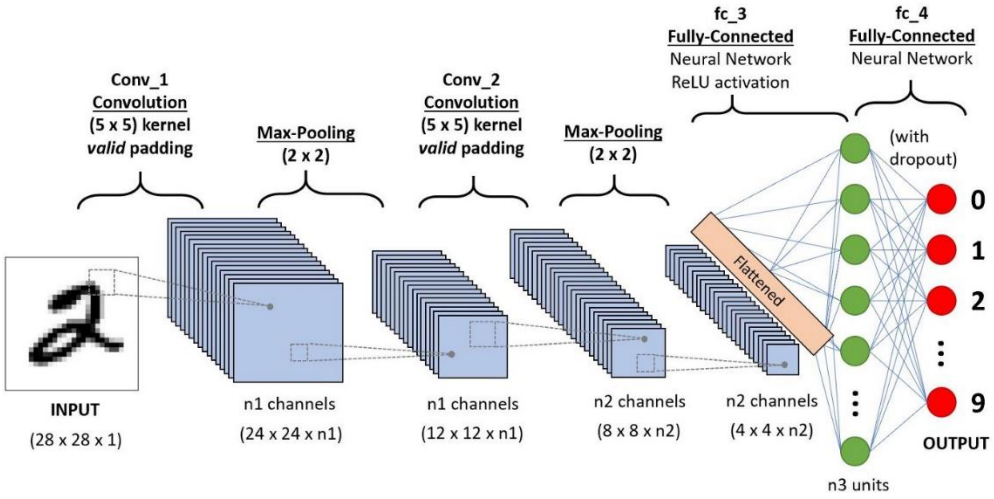

Figure 3 The full CNN architecture for classifying Handwritten digits

The back-propagation in convolutional neural network:

This is a bit of an overkill to derive the full formulas for the back-propagation algorithm in CCN, but since the aim of this research to demystify the theoretical details of the Neural networks, I am going to include it anyway.

We'll call:

- $a^0_{j,k}, 0 \leq j,k \leq 27$, the input activations;
- $w^1_{l,m}, 0 \leq l,m \leq 4$, the shared weights for the convolutional layer;
- b^1, the shared bias for the convolutional layer;
- $z^1_{j,k} 0 \leq j,k \leq 23$, the weighted input to neuron (j,k) (line j, column k) in the conv layer:

$$z^1_{j,k} = b^1 + \sum_{l=0}^{4}\sum_{m=0}^{4} w^1_{l,m} a^0_{j+l,k+m}$$

$a^1_{j,k}, 0 \le j,k \le 23$, the activation of neuron (j,k) in the convolutional layer:

$$a^1_{j,k} = \sigma\left(z^1_{j,k}\right)$$

$a^2_{j,k}, 0 \le j,k \le 11$, the activation of neuron (j,k) in the max-pooling layer:

$$a^2_{j,k} = max\left(a^1_{2j,2k}, a^1_{2j,2k+1}, a^1_{2j+1,2k}, a^1_{2j+1,2k+1}\right)$$

So, neuron $(\boldsymbol{j},\boldsymbol{k})$ in the convolutional layer will contribute to the computation of the max for neuron $\left(\left\lfloor\frac{\boldsymbol{j}}{\boldsymbol{2}}\right\rfloor, \left\lfloor\frac{\boldsymbol{k}}{\boldsymbol{2}}\right\rfloor\right)$.

Note: the symbol $\left\lfloor\frac{\mathbf{j}}{\mathbf{2}}\right\rfloor$, indicate the floor function of $\frac{\mathbf{j}}{\mathbf{2}}$.

- Note that the max-pooling layer doesn't have any weights, biases, or weighted inputs!

- $\boldsymbol{w^3_{l;j,k}0 \le j,k \le 11, 0 \le l \le}$, the weight of the connection between neuron $(\boldsymbol{j},\boldsymbol{k})$ in the max-pooling layer and neuron $\boldsymbol{l}$ in the output layer;

- $\boldsymbol{b^3_l 0 \le l \le}$, the bias of neuron $\boldsymbol{l}$ in the output layer;

- $\boldsymbol{z^3_l 0 \le l \le}$, the weighted input of neuron $\boldsymbol{l}$ in the output layer:

$$z^3_l = b^3_l + \sum_{0 \le j,k \le 11} w^3_{l;j,k} a^2_{j,k}$$

  - $a^3_l 0 \le l \le$, the output activation of neuron l in the output layer:

$$a^3_l = \sigma(z^3_l)$$

Now for comparison, here are equations BP1 - BP4 for regular fully connected networks:

- **BP1**: $\delta^L_j = \frac{\partial C}{\partial a^L_j}\sigma'\left(z^L_j\right)$
- **BP2**: $\delta^l_j = \sum_k w^{l+1}_{kj}\,\delta^{l+1}_k \sigma'\left(z^l_j\right)$
- **BP3**: $\frac{\partial C}{\partial b^l_j} = \delta^l_j$

- **BP4**: $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$

And their shortened derivations (only writing $\frac{\partial x}{\partial y}$ when y has an influence on x):

BP1: $$\delta_j^L = \frac{\partial C}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\sigma'\left(z_j^L\right) \quad (1)$$

BP2: $$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial a_j^l}\frac{\partial a_j^l}{\partial z_j^l} = \sum_k \delta_k^{l+1} w_{kj}^{l+1}\sigma'\left(z_j^l\right) \quad (2)$$

BP3: $$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l \times 1 \quad (3)$$

BP4: $$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1} \quad (4)$$

Let's look at each equation in turn, with our new network architecture.

- **BP1:** The last layer following the previous network architecture, we see that the derivation of BP1 remains correct. Therefore, BP1 doesn't change.
- **BP2:** since the max-pooling layer doesn't have any weighted inputs, we'll just have to compute $\delta_{j,k}^1$.

$$\delta_{j,k}^1 = \frac{\partial C}{\partial z_{j,k}^1}$$

$$= \sum_{l=0}^{9} \frac{\partial C}{\partial z_l^3}\frac{\partial z_l^3}{\partial z_{j,k}^1}$$

$$= \sum_{l=0}^{9} \delta_l^3 \frac{\partial z_l^3}{\partial a_{j',k'}^2}\frac{\partial a_{j',k'}^2}{\partial z_{j,k}^1}$$

with $j' = \left\lfloor \frac{j}{2} \right\rfloor, \mathrm{k}' = \left\lfloor \frac{k}{2} \right\rfloor$

$a_{j',k'}^2$ being the only activation in the max-pooling layer affected by $z_{j,k}^1$.

$$= \sum_{l=0}^{9} \delta_l^3 w_{l;j',k'}^3 \frac{\partial a_{j',k'}^2}{\partial z_{j,k}^1}$$

$$= \sum_{l=0}^{9} \delta_l^3 w_{l;j',k'}^3 \frac{\partial a_{j',k'}^2}{\partial a_{j,k}^1} \frac{\partial a_{j,k}^1}{\partial z_{j,k}^1}$$

$$= \sum_{l=0}^{9} \delta_l^3 w_{l;j',k'}^3 \frac{\partial a_{j',k'}^2}{\partial a_{j,k}^1} \sigma'\left(z_{j,k}^1\right)$$

Now since $a_{j',k'}^2 = \max\left(a_{2j',2k'}^1, a_{2j',2k'+1}^1, a_{2j'+1,2k'}^1, a_{2j'+1,2k'+1}^1\right)$ and we're talking about infinitesimal changes, we have:

$$\frac{\partial a_{j',k'}^2}{\partial a_{j,k}^1} = \begin{cases} 0, & if\ \ a_{\{j,k\}}^1 \neq\ max\ \left(\ a_{\{2j',2k'\}}^1, a_{\{2j',2k'+1\}}^1, a_{\{2j'+1,2k'\}}^1, a_{\{2j'+1,2k'+1\}}^1\right) \\ 1, & if\ \ a_{\{j,k\}}^1 =\ max\ \left(\ a_{\{2j',2k'\}}^1, a_{\{2j',2k'+1\}}^1, a_{\{2j'+1,2k'\}}^1, a_{\{2j'+1,2k'+1\}}^1\right) \end{cases}$$

This is because $a_{j,k}^1$ only affects $a_{j',k'}^2$ if $a_{j,k}^1$ is the maximum activation in its local pooling field. In this case, we have $a_{j',k'}^2 = a_{j,k}^1$, so $\frac{\partial a_{j',k'}^2}{\partial a_{j,k}^1}$ =1.

And so to conclude the derivation of our new BP2:

$$\delta_{j,k}^1 = \begin{cases} 0, & if\ \ a_{\{j,k\}}^1 \neq\ max\ \left(\ a_{\{2j',2k'\}}^1, a_{\{2j',2k'+1\}}^1, a_{\{2j'+1,2k'\}}^1, a_{\{2j'+1,2k'+1\}}^1\right) \\ \sum_{l=0}^{9} \delta_l^3 w_{l;j',k'}^3 \sigma'\left(z_{j,k}^1\right), & if\ \ a_{\{j,k\}}^1 =\ max\ \left(\ a_{\{2j',2k'\}}^1, a_{\{2j',2k'+1\}}^1, a_{\{2j'+1,2k'\}}^1, a_{\{2j'+1,2k'+1\}}^1\right) \end{cases}$$

- **BP3**: we consider two cases:
  - $\frac{\partial C}{\partial b_l^3} = \delta_l^3$ as the third layer respects the previous architecture (the derivation still works);
  - $\frac{\partial C}{\partial b^1}$. This one is different, since the bias b^1 is shared for all neurons in the convolutional layer.

We have:

$$\frac{\partial C}{\partial b^1} = \sum_{0 \leq j,k \leq 23} \frac{\partial C}{\partial z_{j,k}^1} \frac{\partial z_{j,k}^1}{\partial b^1}$$

$$= \sum_{0 \leq j,k \leq 23} \delta_{j,k}^1 \frac{\partial z_{j,k}^1}{\partial b^1}$$

$$= \sum_{0 \le j,k \le 23} \delta^1_{j,k} \qquad \text{as } z^1_{j,k} = b^1 + \sum_{l=0}^{4} \sum_{m=0}^{4} w^1_{l,m} a^0_{j+l,k+m}$$

- **BP4**:
  - $\frac{\partial C}{\partial w^3_{l;j,k}} = a^2_{j,k} \delta^3_l$ since, again, the derivation still works for the third layer;
  - $\frac{\partial C}{\partial w^1_{l,m}}, 0 \le l, m \le 4$. These 25 weights are shared, and each of them is used in the computation of the weighted input of each neuron in the convolutional layer:

$$\frac{\partial C}{\partial w^1_{l,m}} = \sum_{0 \le j,k \le 23} \frac{\partial C}{\partial z^1_{j,k}} \frac{\partial z^1_{j,k}}{\partial w^1_{l,m}}$$

$$= \sum_{0 \le j,k \le 23} \delta^1_{j,k} \frac{\partial z^1_{j,k}}{\partial w^1_{l,m}}$$

$$= \sum_{0 \le j,k \le 23} \delta^1_{j,k} a^0_{j+l,k+m} \qquad \text{as } z^1_{j,k} = b^1 + \sum_{l=0}^{4} \sum_{m=0}^{4} w^1_{l,m} a^0_{j+l,k+m}$$

Depth-wise Separable Convolution (DSC) - A FASTER CONVOLUTION! –

The standard convolution operation is relatively slow to perform, however we can speed this up with an alternative method Depth wise Separable Convolution

The Standard Convolution.

let's first very quickly go over the basics of convolution on an input volume. Consider an input volume F, of shape DF×DF×M where DF is the width and height of the input volume and M is the number of input channels, if a color image was an input, then M would be equal to 3 for the RGB channels. We apply convolution on a kernel K of shape DK×DK×M this will give us an output of shape DG×DG×1, if we apply n such kernels on the input then we get an output volume G of shape DG×DG×N (Fig below).

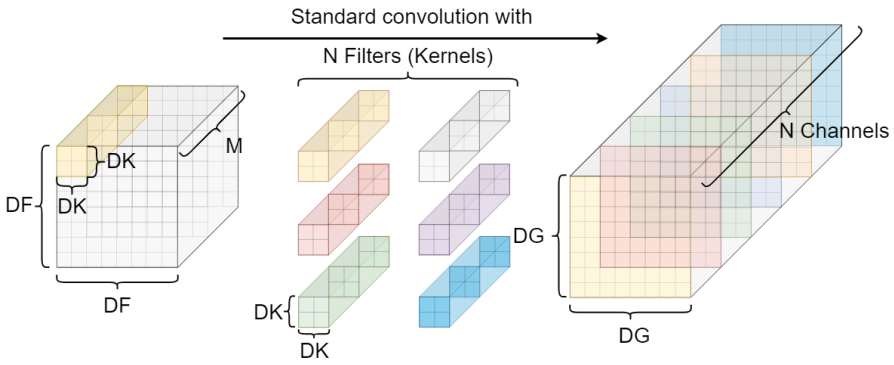

Figure 4 Standard convolution with N filters

The concern

I'm more concerned now with the cost of this convolution operation

So, let's take a look at that, we can measure the computation required for convolution by taking a look at the number of multiplications required.

But, why is that? it's because multiplication is an expensive operation relative to addition, so let's determine the number of multiplications for one convolution operation. The number of multiplications is the number of elements in that kernel so that would be $DK \times DK \times M$ multiplications, but we slide this kernel over the input we perform DG (size of the feature map) convolutions along the width, and DG convolutions along the height and hence $DG \times DG$ convolutions over all, so the number of multiplications in the convolution of one kernel over the entire input F is $DG^2 \times DK^2 \times M$ now this is for just one kernel, but if we have N such kernels which makes the absolute total number of multiplications become $DG^2 \times DK^2 \times M \times N$ multiplications.

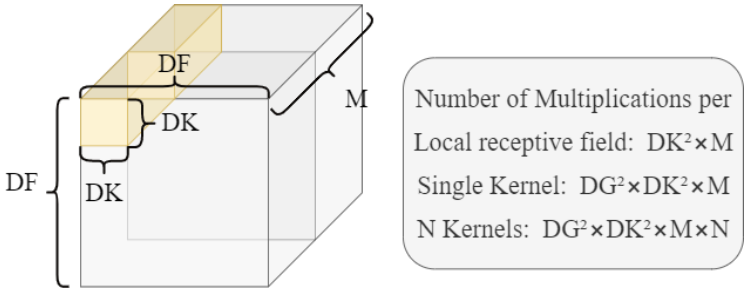

Figure 5 Number of multiplications in the standard convolution

Depth wise separable convolution operation.

let's now take a look at Depth wise Separable Convolutions (DSC), in standard convolution the application of filters across all input channels and the combination of these values are done in a single step. DSC on the other hand breaks us down into two parts, the first is depth wise convolution that is it performs the filtering stage, and then point wise convolution which performs the combining stage.

let's get into some details here, depth wise convolution applies convolution to a single input channel at a time, this is different from the standard convolution that applies convolution to all channels, let us take the same input volume F to understand this process F has a shape DF×DF×M, where DF is the width and height of the input volume and M is the number of input channels, like I mentioned before.

For depth wise convolution we use filters or kernels K of shape DK×DK×1, here DK is the width and height of the square kernel and it has a depth of 1 because this convolution is only applied to a channel unlike standard convolution which is applied throughout the entire depth, and since we apply one kernel to a single input channel, we require M such DK×DK×1 kernels over the entire input volume F.

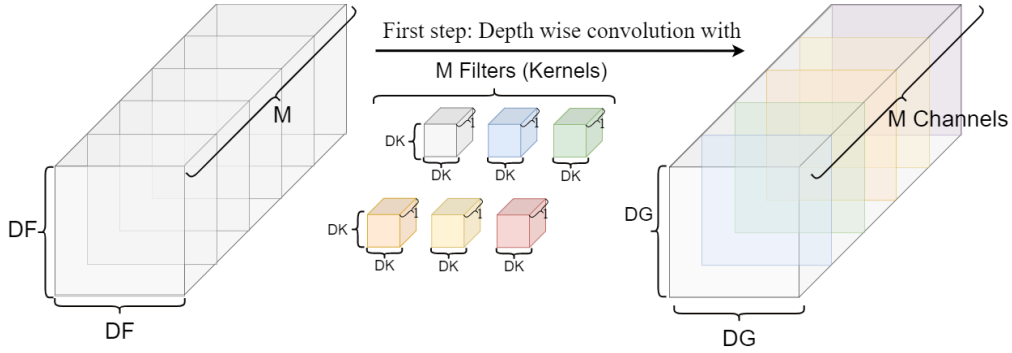

Figure 6 First step of Depth wise Separable Convolution

For each of these M convolutions we end up with an output DG×DG×1 in shape, now stacking these outputs together we have an output volume of G which is of shape DG×DG×M, this is the end of the first phase that is the end of depth wise convolution.

Now this is succeeded by point wise convolution point wise convolution involves performing the linear combination of each of these layers, here the input is the volume of shape DG×DG×M, the filter KPC has a shape 1×1×M, this is basically a 1×1 convolution operation over all M layers. The output will thus have the same input width and height as the input DG×DG for each filter assuming that we want to use N such filters the output volume becomes DG×DG×N.

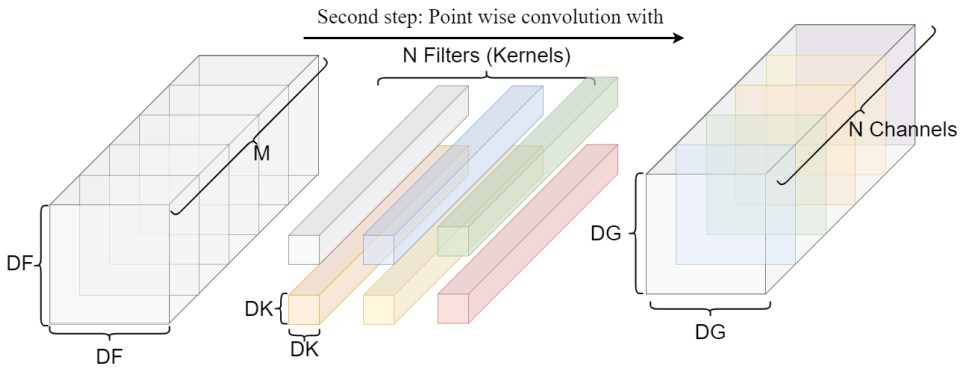

Figure 7 Second step of Depth wise Separable Convolution

Depth wise separable convolution complexity.

Now let's take a look at the complexity of this convolution, we can split this into two parts as we have two phases, first we compute the number of multiplications in depth wise convolution, so here the kernels have a shape DK×DK×1, so the number of multiplications on one convolution operation is all DK×DK, or DK^2, when applied over the entire input channel this convolution is performed DG×DG number of times, so the number of multiplications for the kernel over the input channel becomes $DG^2 \times DK^2$. Now such multiplications are applied over all M input channels, for each channel we have a different kernel and hence the total number of multiplications in the first phase, that is depth wise convolution is $M \times DG^2 \times DK^2$.

Next, we compute the number of multiplications in the second phase that is point wise convolution, here the kernels have a shape 1×1×M where m is the depth of the input volume and hence the number of multiplications for one instance of convolution is M, this is applied to the entire output of the first phase which has a width and height of DG, so the total number of multiplications for this kernel is DG×DG×M, so for some N kernels will have DG×DG×M×N, such multiplications. Thus, the total number of multiplications is the sum of multiplications in the depth wise convolution stage, plus the number of multiplications in the point-wise convolution stage.

$$\text{Total Number of Multiplications} = M \times DG^2 \times DK^2 + DG^2 \times M \times N = M \times DG^2 \, (DK^2 + N)$$

Now we compare the standard convolution with depth wise convolution we get the ratio as:

$$\frac{N.Muts\ in\ DepthwiseSeparableConvolution}{N.Muts\ in\ StandardConvolution} = \frac{M \times DG^2(DK^2 + N)}{M \times DG^2 \times DK^2 \times N} = \frac{DK^2 + N}{DK^2 \times N} = \frac{1}{N} + \frac{1}{DK^2}$$

To put this into perspective of how effective depth wise convolution is let us take an example so consider the output feature volume n of 1024 and a kernel of size 3 that's DK=3. Plugging these values into the relation we get 0.112, in other words standard convolution has 9 times more the number of multiplications as that of depth wise separable convolution, this is a lot of computing power.

We can also quickly compare the number of parameters in both convolutions in standard convolution each kernel has DK×DK×M learn about parameters since there are N such kernels there are N×M×DK² parameters in depth wise separable convolutions will split this once again into two parts, in the depth wise convolution phase we use M kernels of shape DK×DK in point wise convolution we use n kernels of shape 1×1×M so the total is M×DK² + M×N, or we can just take M common, taking the ratio we get the same ratio as we did for computational power required.

$$\frac{N.ParametersDepthwiseSeparableConvolution}{N.ParametersStandardConvolution} = \frac{M \times (DK^2 + N)}{M \times DK^2 \times N} = \frac{DK^2 + N}{DK^2 \times N} = \frac{1}{N} + \frac{1}{DK^2}$$

Where depth wise has been used?

Now, we understood exactly what depth wise convolution is, and also its computation power with respect to the traditional standard convolution, but where exactly has this been used? well there are some very interesting papers here, the first is on multi model neural networks these are networks designed to solve multiple problems using a single network a multi model network has four parts, the first is modality Nets to convert different input types to a universal internal representation then we have an encoder to process inputs we have a mixer to encode inputs with previous outputs and we have a decoder to generate outputs. A fundamental component of each of these parts is depth wise separable convolution it works effectively in such large networks.

Next up we have Xception (Franc¸ois Chollet from Google) a CNN architecture based entirely on depth wise separable convolutional layers, it has shown the state-of-the-art performance on large datasets like Google's JFT image dataset, which is a repository of 350 million images with 17,000 class labels, to put this into perspective the popular image net took 3 days to Train however to Train even a subset of this JFT data set it took a month and it didn't even converge. In fact, it would have approximately taken about three months to converge. This paper has pushed convolution neural networks to use DSC as a default convolution.

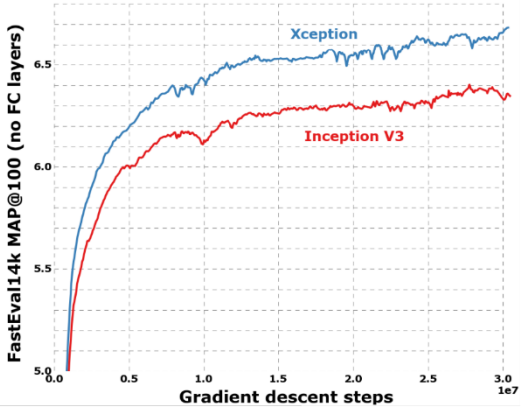

Figure 8 Training profile on JFT, without fully-connected layers

Up third we have mobile Nets a neural network architecture that strives to minimize latency of smaller scale networks so that computer vision applications run well on mobile devices. Mobile Nets used DSC in its 28-layer architecture. This paper compares the performance of mobile nets with fully connected layers versus DSC layers. It turns out the accuracy on image net only drops a 1% while using significantly less number of parameters, from twenty nine point three million the number of parameters it's down to just 4.2 million. We can see the Mult-Adds which is a direct measure of computation has also significantly decreased for DSC mobile Nets.

Model	ImageNet Accuracy	Million Mult-Adds	Million Parameters
Conv MobileNet	71.7%	4866	29.3
MobileNet	70.6%	569	4.2

Things to remember about Depth-wise Separable Convolution:

So here are some things to remember, depth wise separable convolution decreases the computation and number of parameters when compared to standard convolution, second is that DSC is a combination of depth wise convolution followed by a point wise convolution, depth wise convolution is the filtering step and point wise convolution can be thought of as the combination step. Finally, they have been successfully implemented in neural network architectures like multi model networks exception and mobile nets.

Employing Depth-wise Separable Convolution: (this should be above takeaways)

We already compared CNNs that use the standard convolution to fully connected networks. So, no need to go back to the part, we know that CNNs are superior. Now I want to compare CNNs that use the standard convolution to CNNs that use the Depth-wise Separable Convolution.