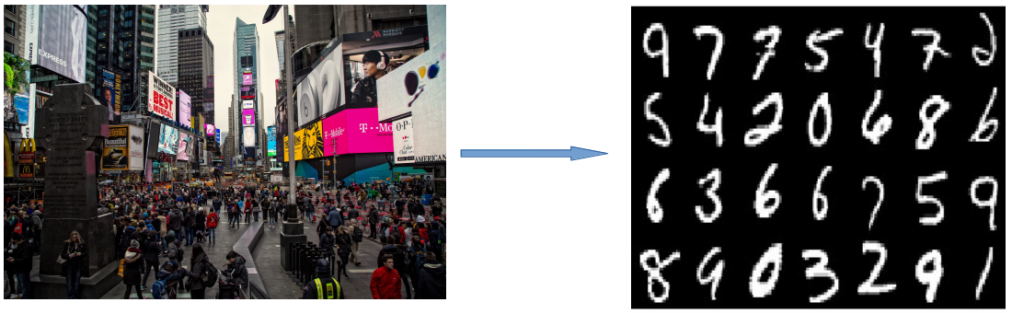
Unit 8 – Convolutional Networks

One of the earliest and most successful specialized architectures in Deep Learning is what we know as a ***convolutional network***. In order to understand how they are built, how they work, and how their architecture is designed we first have to think of the particular problems they were designed to solve.

We have already successfully trained neural networks to carry out fairly sophisticated pattern recognition – we built fully connected networks capable of recognizing hand-written digits, discovered that a fully connected network with a single hidden layer can perform this task quite well, but we also discovered that on more complex problems (classifying images in the CIFAR-10 dataset) the network struggles to learn a model that generalizes well to previously unseen images.

There are several limitations imposed by the fully-connected architecture that limit the usefulness of such networks for tasks that have to be performed on images: pattern recognition (such as OCR, object detection, tracking of objects on video), image classification, image segmentation, automatic annotation, or image restoration tasks (noise removal, de-blurring, contrast or colour enhancements, etc.).

Firstly – a fully-connected network requires a ***fixed size input*** image – by design, the network requires an input of a fixed size. For the digit recognition task, all our inputs had to be scaled to the correct size, and contain the digits we are interested in classifying at a reasonably similar size and at the correct location and orientation in each image. This is not particularly realistic if what we want is a network that detects digits (or other characters) on images of any type – document scans, photographs, graphic art. Such images will have any size, the characters in them will appear at ***anwhere,*** and will show a variety of distortions in ***size***, ***orientation***, ***shape***, ***colour***, and more.



Text and other characters on regular images can be at any location, have any size/orientation, be drawn on any font and colour, and have an arbitrary background – how would we transform these into perfectly scaled, centered, clean (no background) images to feed to our fully connected digit classification network? (Images: right side by George Hodan, Public Domain. Left side, samples from the MNIST digit recognition dataset)

We could imagine training a fully connected network to carry out the digit recognition task on much larger images – for instance, we could fix the image resolution at 1920x1080 pixels (a standard HD image frame) and see how far we get.

Here's how we get in trouble with a fully-connected network:

* Each neuron in the hidden layer will require just over 2 million weights to connect to pixels in

the input image

* Suppose we train a network with 625 units (as we did on the MNIST dataset), that will mean the first layer of neurons will have over 1.2 billion weights connecting units to the pixels in the input image

This is a lot of weights, simply for the first layer of a network to have access to the input – most of which will be ***irrelevant*** to the task most of the time (ask yourself what is the likelihood that on a particular image, a particular pixel is part of a pattern that sets it out as a digit we need to classify). The problem itself is way too difficult to solve as a general task – we would be requiring the network to learn every possibly location, orientation, size, colour, font, and background that could identify a particular digit or character we are interested in recognizing.

To handle such complexity we would expect to require a network with a very large number of units distributed across some number of hidden layers – the number of network weights would quickly grow to amounts that are not practical. Beyond requiring huge amounts of data to train (each digit or character must appear multiple times at each possible location, orientation, size, colour, and background), it is not clear that it would be even possible to train such a large network to perform well on this task.

The point of the above is to say that ***a fully connected network is not the right kind of architecture*** for approaching the problem.

Problems in image processing have common properties that motivate the use of a different architecture that ***provides invariance to scale, orientation, contrast, and background***. Fortunately, we can draw on decades of research in image processing and computer vision to guide us in designing a network architecture that can achieve all of these.

Two important lessons from earlier computer vision research

1) Meaningful feature extraction is essential

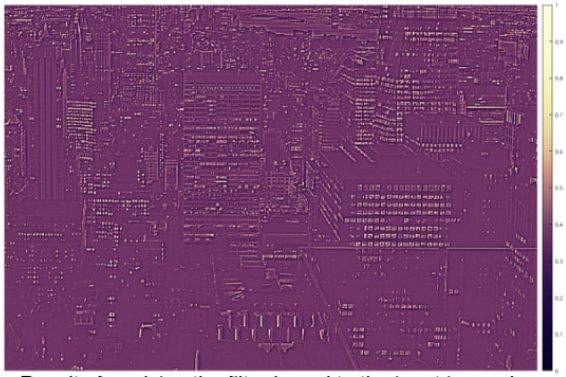
Images have too many pixels, and pixels are not very informative on their own – the value of a particular pixel *(i,j)* tells us very little *in general* about anything at all. So, handling any reasonably interesting image-related problem requires that we move away from pixel values, and instead think in terms of ***meaningful features***.

In this context, a feature should be thought of as ***a pattern of interest*** that is present within a ***spatially contiguous region*** of an image. As an example, consider the image below – and imagine we are trying to solve the following task: *Identify and count the illuminated windows in all the buildings in the scene*.

Input image – New York Skyline
(image by Sarah Meyer, Wikimedia Commons, CC-SA-3.0)

9x9 pixels ***feature*** representing a single window

This is used as a filter ***kernel***



Result of applying the filter kernel to the input image by ***convolution*** – bright areas show strong response to this particular filter – so they resemble the ***feature*** it represents

Since we are looking for illuminated windows, we want to look for bright regions that are surrounded by dark borders. Individual bright pixels or individual dark pixels aren't very useful because each window contains many such – however, we can design a small (in this case 9 pixel by 9 pixel) ***feature*** of interest that basically looks like a bright blob with the general shape of what we might expect a window to look like.

The feature is simply a smaller image. We will now ***slide the feature image over the large input photograph*** and measure how well the input photo resembles our feature of interest at each possible location, creating a ***feature map*** that shows how strongly the input image resembles our blob at every location we may place it.

The result of this process is shown above – the feature image has been scaled up (otherwise it would be too tiny to see!). The resulting ***feature map*** is shown on the right hand side – bright pixels in the feature map show locations where the input image is very similar to our feature. Notice that many of the windows have been highlighted!

A couple of important things are taking place here:

- We are taking an input image and ***extracting a feature map*** showing where an ***interesting feature may be present***. Such features of interest are not individual pixels, but rather, small image regions that capture a pattern that is somehow useful to the task at hand.

- The ***feature map*** is ***sparse*** – large regions in it are quite dark, meaning they are likely of no interest in solving our specific problem. At the same time, we get responses to the filter anywhere over the image. There is no expectation the pattern may appear at a particular location, and the input image itself can have any size.

Discrete convolutions in 2-D

The process informally discussed above is carried out in practice through the mathematical process of convolution. In the context of image processing, a discrete convolution in 2-D takes as input a pair of 2-D arrays, one of which corresponds to input data, and the second of which represents a ***filter kernel*** – this latter array is typically much smaller than the input image, and defines an operation we

want to perform over image regions of the same size as the ***kernel***. A ***2-D discrete convolution*** is defined as:

$$g(x,y) = \sum_{i=-w/2}^{w/2} \sum_{j=-w/2}^{w/2} k(-i,-j) \cdot f(x+i, y+j),$$

here, the kernel ***k(i,j)*** has size ***w*** (it is assumed to be square, and the size is ***odd*** so it has a unique entry that can be called the kernel's center). The input image is ***f(x,y)*** and the result of the convolution is ***g(x,y)***.

- To compute ***g(x,y),*** the kernel is placed over the input image ***f(x,y)*** so that the kernel's center is exactly at ***(x,y).*** This means the kernel will overlap a square region of size ***w*** centered at ***(x,y).***

- The product of the ***flipped kernel*** with the input image region it overlaps is then computed by multiplying together the corresponding pixel and kernel entries.

- The sum of these products over the entire region is the result of the convolution.

A small but important point is that the kernel is ***flipped*** in both directions before the convolution is computed (notice the indexing on ***f(x,y)*** above, it effectively flips the order of kernel entries as they are applied to corresponding image pixels). This is because by convention the convolution operation must preserve the property that on an input that corresponds to a ***delta function*** (a single 1, everywhere else 0), the result of the convolution should be the filter kernel itself.

This is an expected property of ***linear, shift-invariant filters***.

In summary – image processing tasks usually require ***detecting meaningful features*** on images, the detection process relies on using a ***filter kernel*** and applying it to the input image by ***convolution***. The resulting ***feature map*** encodes the responses to the filter kernel. Because we perform feature extraction over all possible image locations, ***this process achieves (discrete) translation invariance***. We no longer care where the pattern of interest appears, the feature map will record its presence.

A few notes:

- Filter ***kernels*** can be designed to perform a wide variety of tasks, not only ***feature detection***. For example, a ***Gaussian kernel*** can be used for ***smoothing*** the input image, a ***Laplacian kernel*** can be used for ***edge detection***, and so on.

- To be effective for ***feature detection***, a kernel must be ***zero mean***, and ***pattern matching*** (the kernel's spatial arrangement mirrors the feature we intend to detect).

- Kernels can have more than 2 dimensions. As we will see shortly in the context of convolutional networks, a ***3-D kernel*** with multiple ***layers*** can be convolved with a ***3-D array*** using a corresponding ***3-D convolution***. Generalizing this to ***N-D*** inputs and kernels is straightforward.

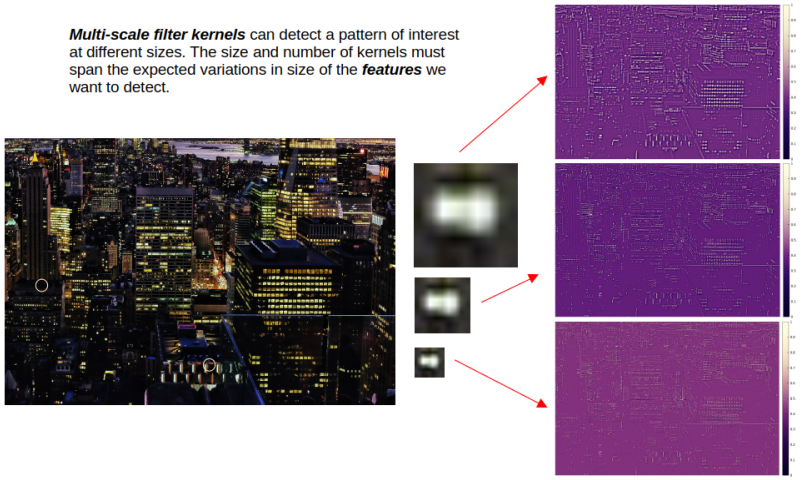
- Care must be taken at the boundaries where the kernel may have regions that are outside of the image. Typically, these regions are ***padded with zeroes*** for the purpose of computing the convolution, but other options are possible.

2) Achieving scale invariance requires processing at multiple scales

One of the stronger sources of variation in images is ***scale***. Consider once again the problem of counting illuminated windows, in the context of the image below:

Consider the different in size of the illuminated window in the two circled regions – a single ***kernel*** won't be able to detect both
(Image: Lenny K, FLICKR, CC BY-2.0)

The ***filter kernel*** we used to highlight possible windows has a specific size and shape. As noted above, feature detecting ***kernels*** are ***pattern matching filters,*** therefore it can only respond strongly to windows that have a fairly similar size and shape. If we want to detect the same ***feature of interest*** at whatever size it may appear in an image, we need to ***use kernels of different sizes*** (with different ***spatial scales***).



We will come back to this idea in short order, as it is built into the architecture of convolutional networks.

Let's recap: complex image-related tasks rely on ***feature detection*** which is carried out via ***convolution*** with ***carefully designed filter kernels*** at ***multiple scales***. The feature detection process extracts meaningful features and focuses further processing on image regions that appear useful for the task at hand. ***Multi-scale processing*** provides (discrete) ***invariance to scale*** and allows feature extraction to detect meaningful features whatever their size.

The two principles discussed above provide a powerful framework for image understanding and image processing, and they form the foundation for the design and computation performed by ***convolutional networks***.

Why ConvNets?

The framework described above can be used to perform fairly complex image processing tasks – but each new task requires designing a set of ***kernels*** that are adapted to the particular problem being solved, building an ad-hoc processing pipeline that extracts ***features,*** and feeding those features to a ***machine learning*** algorithm trained on the task that needs solving.

We know neural networks are very powerful at ***learning interesting patterns*** in input data, so ***why not use a neural net to learn the set of features that is helpful for solving a task?*** At the same time, we can leverage the ability of multi-layer networks to ***build more complexity and expressive power*** layer by layer, enabling them to successfully solve very complex tasks.

Convolutional Networks (***ConvNets***) are specialized deep learning architectures whose purpose is precisely that: The network will ***learn from training data the set of kernels*** that can extract useful features for whatever task the network is being trained to solve. It will apply these ***kernels*** by ***convolution*** to input data to ***extract feature maps,*** and through repeated application of this process across multiple layers of ***kernels*** and ***convolutions*** it will build ***more meaningful, powerful, and expressive*** features. Because of the architecture of the network (as will be seen shortly), high-level features will be ***scale and translation invariant,*** while ***the use of a suitably large set of learned kernels*** will provide invariance to ***rotation, contrast, background*** and ***other sources of variability*** in the input patterns the network is designed to detect.

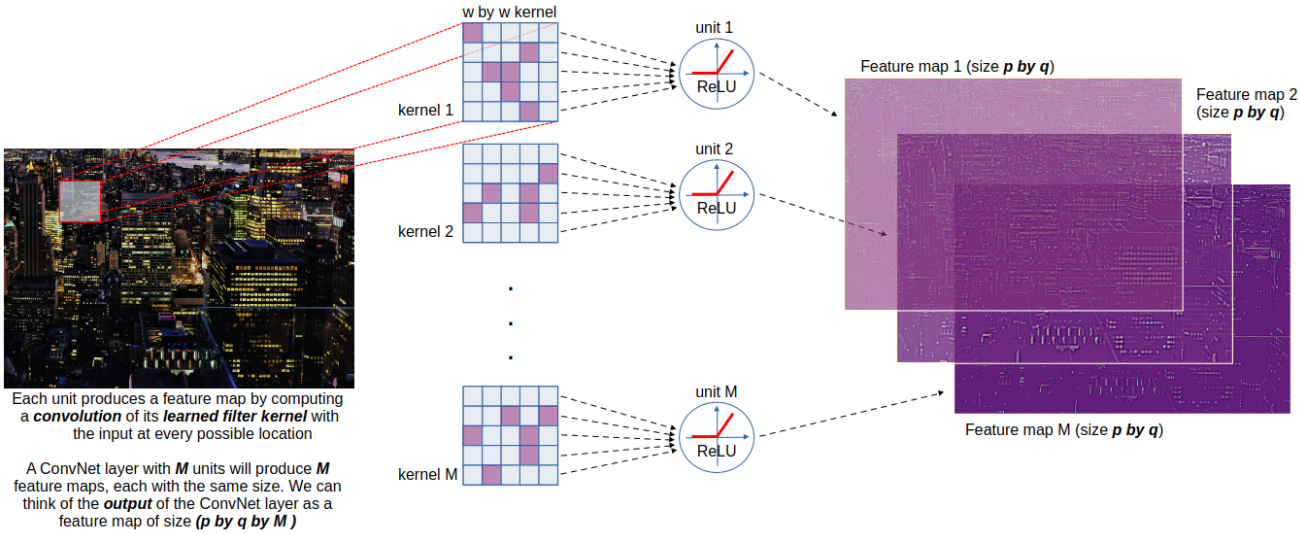
A final ***fully-connected*** layer can then take the ***final feature maps*** produced by the convolutional layers in the network and carry out ***classification***, ***regression, annotation*** or any other related task using as input a set of features that are optimized for the task.

Let's see how this is actually implemented in terms of the architecture of a ConvNet. And for that we should begin by looking at a single ***convolutional layer***.

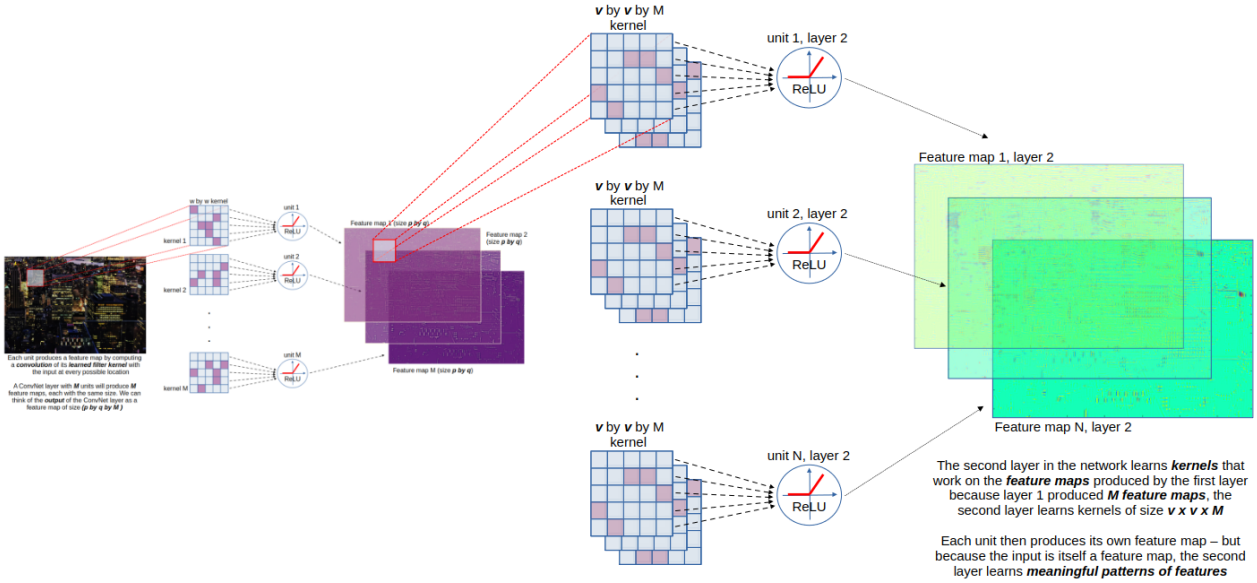
The architecture of a convolutional layer

Unlike regular, fully-connected networks, each unit in a convolutional network's layer is connected to a ***set of weights that form a square kernel with a pre-defined size***. The kernel size is set as a

parameter and is fixed for ***all the units within the layer*** (though note that different layers can have different kernel sizes). The organization of the computation units in a ConvNet layer is shown below (the kernel images show the ***flipped*** kernels, as per the convolution definition).



Units in a ***convolutional network layer*** work as specialized ***feature detectors***. Each of them will ***learn a kernel*** of a pre-defined size. The kernel can be 2-D (for instance if the input is a grayscale image), or it can be N-D: For example, if the input is an RGB image the kernel would be 3-D, and further layers inside the network can have many layers corresponding to the number of feature maps produced by the preceding layer. The ***non-linearity*** (often a ReLU or similar function) introduced by the ***activation function*** in each unit is ***essential***. The ***feature maps*** are ***non-linear activations*** resulting from responses to the specific ***kernel*** learned by each unit in the layer. The output of a ConvNet layer is a ***feature map with M layers*** – such a feature map provides ***abstraction*** and allows succeeding layers to ***learn kernels that detect even more meaningful/useful features***. To make this point clear, consider the image below:



Notice that the second ConvNet layer is using as input ***feature maps*** produced by the first layer – its input is already ***more meaningful/useful*** for the task the network is being trained to solve, and now a second convolutional layer can extract ***meaningful features in the patterns of interest identified by the first layer*** – the second layer is learning to find ***combinations of features*** from the first layer that are even ***more interesting/useful*** for the task being learned.

Stacking additional layers results in progressively more ***abstract***, and more ***informative*** feature maps. In the same way that the ability of a fully connected network to fit extremely complicated functions is compounded with the addition of each hidden layer, the ability of a convolutional network to extract meaningful information from images is compounded with each additional layer of convolution kernels and their non-linear units.

Multi-scale Feature Detection

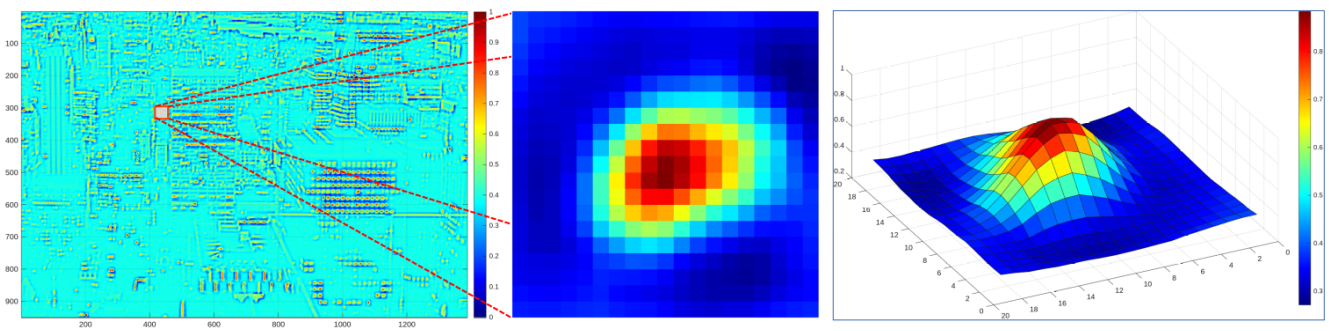
The above explains the the ***computational and feature extraction*** functions of a convolutional layer, but there is one more component that is ***often, though not always*** attached to convolutional layers like the ones discussed above.

Recall that earlier on we raised the issue of using ***kernels at multiple scales*** to detect ***features at multiple scales***. A standard convolutional layer has fixed-size kernels (all the units in the same layer work on kernels with the same dimension). In order to achieve (discrete) ***scale invariance***, convolutional networks ***often*** take advantage of the following observation:

- ***Kernel responses have strong spatial correlations.*** This is the result of applying a ***kernel*** representing a ***feature bigger than individual pixels*** to every location in the image. Given a specific position ***(x,y)*** that contains the feature of interest, we can expect a strong response to the ***kernel*** at ***(x,y)***, but also at nearby locations, such as ***(x-1,y), (x,y-1), (x-1, y-2),*** and so on – within a small radius whose width depends on the size of the kernel as well as the shape of the ***feature of interest*** that it represents.

- This means that information in the ***extracted feature maps*** is redundant within small image regions. As a result of this, we can ***reduce the resolution*** of the ***feature maps*** output by one convolutional layer without missing any of the locations where a particular feature has been detected.

To illustrate this, consider the image below showing a magnified region within the feature map extracted with the original ***kernel*** we used earlier to detect windows in buildings:



[Left] Original, full-resolution feature map. [Middle] magnified patch showing the responses to the filter and their strong spatial correlations. [Right] 3D profile of the same region. The ***kernel*** reponse ***peaks*** at the location that best matches the ***kernel*** and decays smoothly away from that location. We only need to keep the ***peak*** value and its location – surrounding responses provide no additional information

As seen above, filter responses are not localized with very high resolution – the general principle is this:

- Responses to a ***filter kernel with wide spatial support*** are ***spatially correlated*** and show a pattern similar to what is shown above: the response has a ***peak*** at whatever location best agrees with the ***kernel's pattern***, and decays smoothly around it

- As a result of the above, we ***do not need*** and in fact ***do not want*** to keep all the ***kernel responses***. We only need the ***peak*** response for a particular location

Convolutional networks take advantage of the above in order to achieve ***scale invariance*** and ***reduce the computational expense*** of feature extraction. This is done with a combination of:

1) Computing convolutions ***not at every (x,y) location,*** but instead, evaluate the ***kernel*** at locations spaced a fixed distance away from each other. This is called a ***stride*** and is a design parameter for the convolutional layer. If, for instance, the ***stride*** for the layer is ***3***, the feature map will be computed by evaluating the ***kernel*** at a ***grid of locations 3 pixels away from each other*** along both the x and y directions. The resulting ***feature map*** will have ***1/3 the resolution of the input in either direction***.

2) An additional ***pooling layer*** which keeps either ***the maximum, or the average kernel response*** within small, non-overlapping regions of the ***feature map***. For instance, a ***3x3 maxpool layer*** will preserve only the maximum value within a 3x3 region of the ***feature map***.

Therefore, a typical ***convolutional layer*** consists of:

- A layer of units that compute ***convolution with filter kernels*** followed by a ***non-linear activation*** (typically ***ReLU***). Convolutions are computed at locations spaced by the appropriate ***stride*** to produce ***feature maps*** with a smaller size than the input.

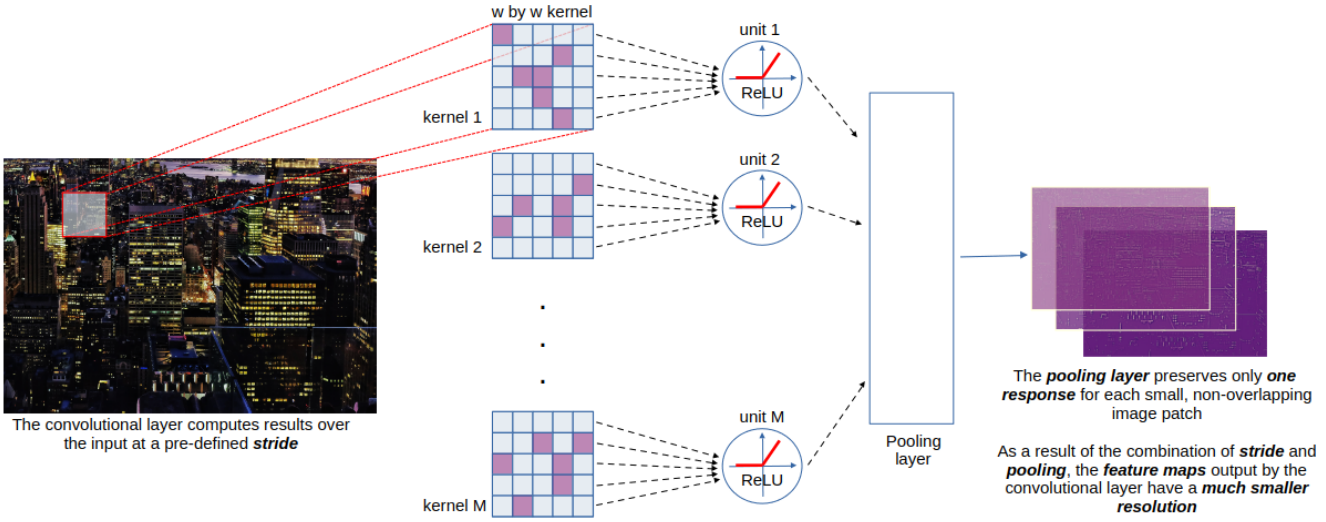
- An optional (thought very often added) ***pooling layer*** that finds and reports either the ***max*** or the ***average*** response within small, non-overlapping image regions. The ***pooling layer*** has no weights that need to be learned. If a ***a pooling layer*** exists, then the final feature map will contain ***only one response for each small neighbourhood*** processed by the ***pooling layer***.

A ***complete convolutional network*** consists of:

- A ***series of convolutional layers***, each with their own ***kernel size***, ***stride***, ***number of units***, and optional ***pooling layer*** (the typical configuration is shown above).

- A final ***fully-connected network*** that takes as input ***the feature maps output by the last convolutional layer*** and uses those to ***solve the specified task***. Typical tasks that convolutional networks excel at include, among others, ***object detection, classification, annotation,*** and ***image restoration***.

- Depending on the problem being solved, the ***fully-connected*** layer may be replaced by a different type of network architecture (e.g. for image restoration tasks, more convolutional layers are employed, as will be discussed below).



For a network with the architecture described above, the ***training process*** has the goal of allowing the different layers to layers to learn the ***kernels*** that allow the network to perform the task it was designed for. The network ***learns a large filter bank*** that is able to detect ***relevant features*** at any ***orientation and position***. By applying ***non-linear activations*** and ***multi-scale processing*** the detected features become ***abstracted and more representative*** of meaningful image content while allowing the network to handle the large variability present in typical images.

Let us summarize what we have learned thus far:

- Convolutional networks rely on the ***detection of meaningful features*** using ***specialized kernels*** learned from ***training data*** and optimized for the task the network is being trained to solve

- They achieve invariance to ***translation and other transformations*** of the relevant patterns involved in solving a task by learning ***a large bank of filter kernels*** that detects ***relevant features at multiple orientations, and at any location in the input***

- They achieve ***scale invariance*** in a ***computationally efficient way*** by using a combination of ***stride*** steps greater than 1, and a ***pooling layer*** that preserves only one value for each small, non-overlapping region in the feature maps output by the non-linear, convolution units.

- They often have a final ***fully-connected network*** whose job is to use the final, small-size ***feature maps*** produced by the last layer of convolutions to make a ***decision*** that helps solve a difficult image processing problem. Because the final layer has ***small feature maps***, implementing the ***fully-connected network*** is feasible and doesn't require huge numbers of units.

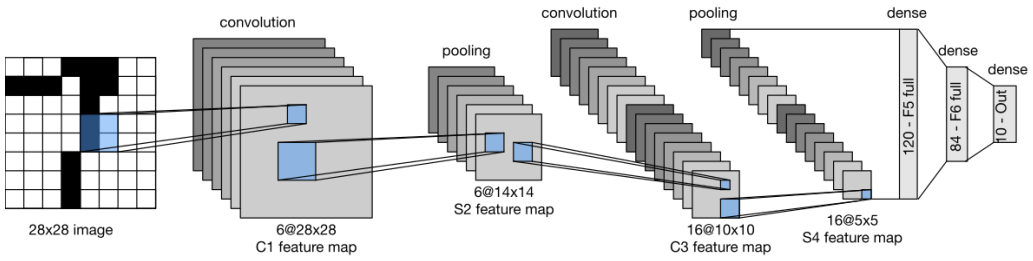
The design of the network includes ***the number and configuration of each convolutional layer*** (the number of units, the type of activation function, the stride, the size of the kernels, and whether or

not to use a pooling layer), the ***configuration*** of the ***final layer*** (which often is a fully-connected network). A lot of the work required in solving a particular problem goes toward figuring out a configuration that works well.

It is useful to spend a bit of time learning about ***well known designs*** that have shown to perform well for ***particular problems.*** These designs became the starting point for further refinements and advances in specific fields of application.

Case studies in applications of convolutional networks

1) LeNet – Proposed in the early 1990's by LeCun et al., it consisted of a convolutional network for the purpose of hand-written digit recognition, and was demonstrated on the MNIST dataset.



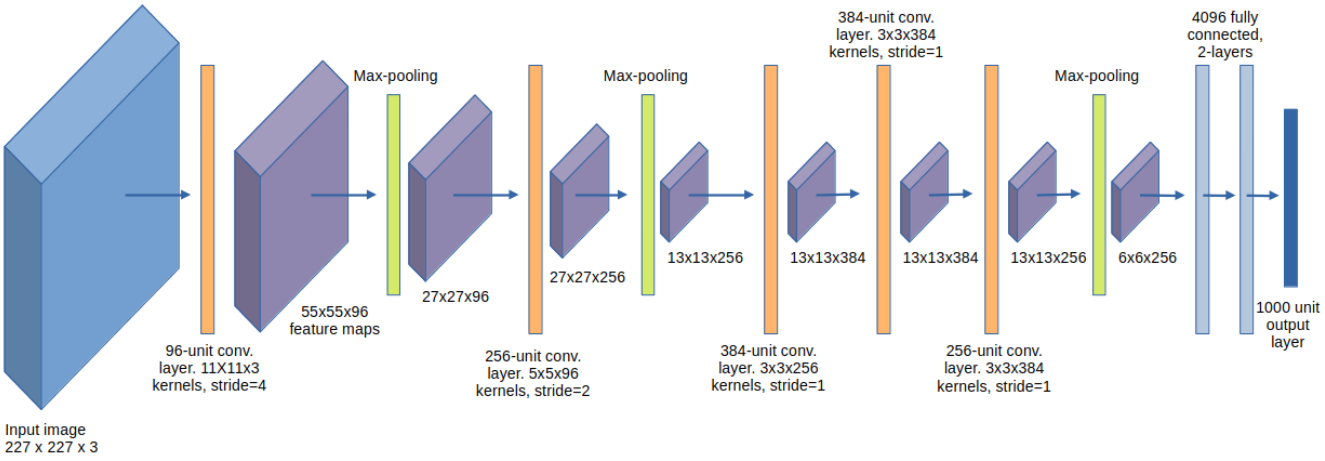
LeNet-5, 1998. (Image: Zhang et al., Wikimedia Commons, CC-SA-4.0)

The first layer of convolutions uses 6 units to create 6 feature maps, same size as the input. A ***pooling layer*** reduces these maps to a size of 14x14 (which means the pooling layer looks at 2x2 patches). The next layer of convolutions contains 16 units and computes 16 feature maps, which are the passed to another ***pooling layer***. The final feature maps have a size of 5x5, and there are 16 of them. This yields a total of ***400 values*** that are input to a ***fully connected network with 3 layers***. The last layer with ***10 units*** is used to detect each of the hand-written digits that may be present in the input (the outputs are passed through a softmax function which turns them into probabilities for each class).

This network is interesting because we can compare it with our ***fully-connected***, ***2-layer*** network trained on the MNIST digit data. LeNet achieves a higher correct classification rate on ***test data*** (the reported accuracy is close to 99%). More importantly, it performs significantly better on the CIFAR-10 dataset (with accuracy close to 75%, compared to 40% from the largest, 2-layer fully connected network we have trained ourselves).

This should not be too surprising – LeNet-5 is a ***deeper network*** (on top of two convolutional layers, it also has one more hidden layer), and takes advantage of the abstraction and feature detection power of convolutional networks. What should be remarkable is how a relatively small convolutional network can quickly outperform fairly large fully-connected networks on image-related tasks.

2) AlexNet – Proposed in 2012 by Krizhevsky et al. It demonstrated what at the time was unprecedented performance in a large-scale, image recognition task (the ImageNet Large Scale Image Recognition Challenge, the task is to recognize objects belonging to one of 1000 different categories). The structure of AlexNet is shown below – the convolutional layers use ReLU activations:



AlexNet architecture, convolution layers are light orange, pooling layers are green, fully-connected layers are blue

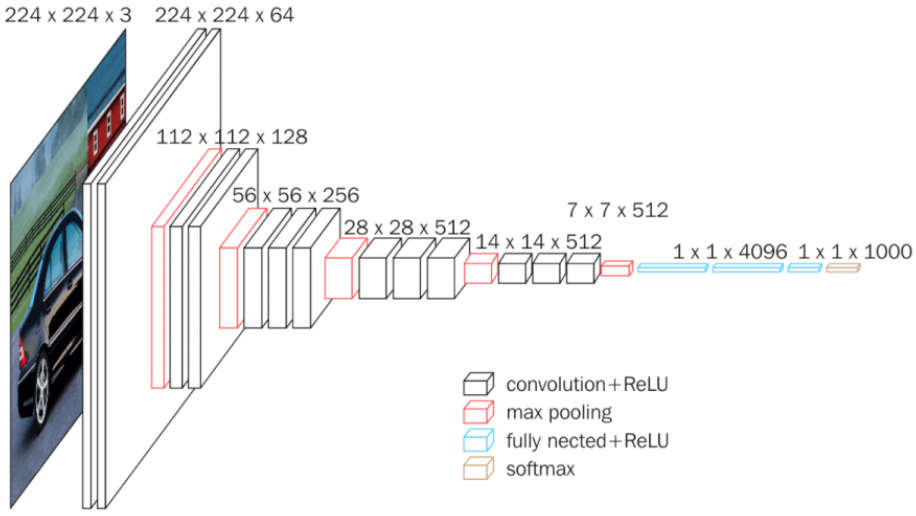
AlexNet is important in that it resulted in a major shift toward research in convolutional networks and deep learning. Further work has yielded even stronger models for image classification, object recognition, and image processing.

3) VGG-16 – Proposed by Simonyan and Zisserman in 2014, it consists of a significantly ***deeper network*** of convolutional layers, and achieved a significant improvement in classification accuracy on the same recognition challenge. The structure of VGG-16 is shown below, and consists of ***16 computation layers*** (that is, not counting the max-pooling layers or the final softmax classification layer).

It is important to note that ***the architectures of VGG and AlexNet*** are pretty much identical. What changes are the specific parameters of the network: The number of layers, the number of units, kernel sizes, strides, and organization of the pooling layers. But the processing pipeline is clear to see:

Use a sequence of ***convolutional layers*** *to extract* ***progressively more meaningful features*** *while* ***reducing the size*** *of the feature maps to a point where it is practical to use a* ***fully-connected network*** *to perform the classification task.*

The input to the ***fully-connected network*** is a ***highly abstracted, expressive, informative feature map*** that provides it with the power required to accurately identify an image as containing an object from 1 of the 1000 possible classes.



Architecture of the VGG-16 network
(Image: Paras Varshney, Apache 2.0 license)

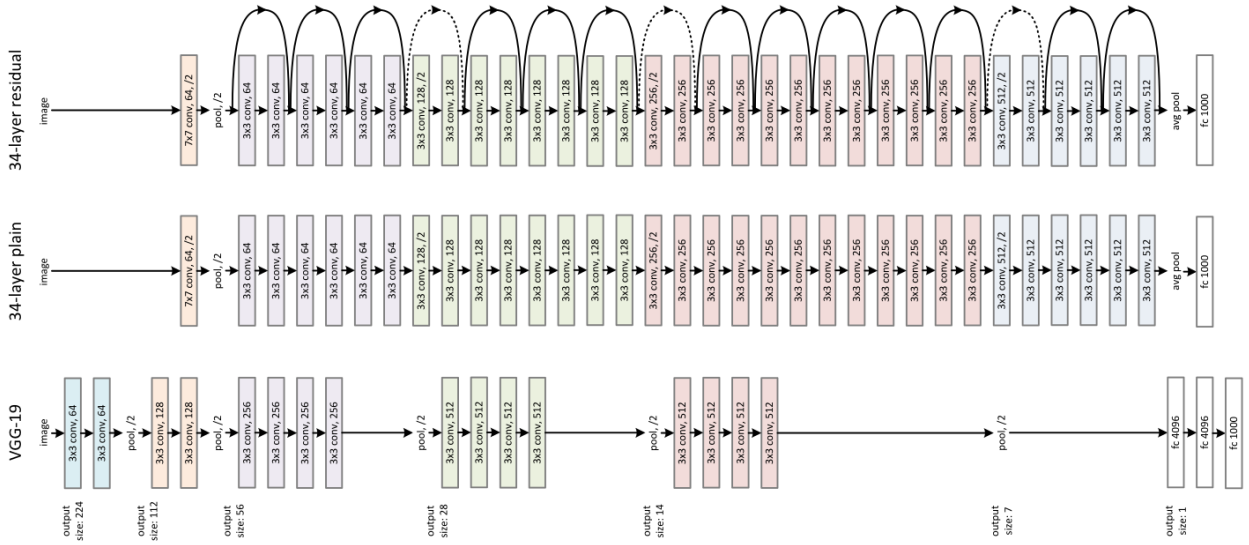
It should be noted that without a major difference in architecture, significant improvements in classification accuracy were achieved by increasing the number of layers in the network. Variations of VGG with additional layers exist, and specialized versions trained to perform other image classification tasks have been developed.

4) ResNet – Proposed in 2015 by He et al. As we have discussed in the context of deep, fully connected networks, adding layers to the network makes the training challenging due to problems with gradient propagation (vanishing gradient, exploding gradient).

As we saw above, improvements in performance on tasks such as image categorization were achieved by increasing the number of layers in the model – the gradient propagation issues become much more challenging as we add layers, limiting the ***depth*** of networks we can effectively train.

We have mentioned before that adding ***skip*** (also called ***residual***) connections to a deep network can help reduce the problems caused by vanishing or exploding gradients, so it makes sense we should take advantage of such connections in ConvNets.

ResNet demonstrated that it is possible to build ***very deep ConvNets*** (with over 150 layers!) by organizing them into ***residual blocks*** – these blocks consist of a sequence of ***convolutional layers*** with a skip connection going across them. The basic architecture is shown below (keep in mind that many variations are possible, the ***residual blocks*** often contain ***pooling layers*** and ***batch normalization*** layers as well as convolutions). In the example below, solid skip connections are just an element-wise sum, while dotted-line skip connections include a pooling layer (notice the factor of 1/2 in the size of the input at these points in the network).



ResNet architecture, compared to VGG-19 Notice the ***residual connections*** – the 34-layer network with ***residual connections*** is much easier to train (shown in the paper in terms of convergence), and with skip connections it is possible to train very deep networks
(ResNet with 152 layers achieves impressive performance on the Image Classification Challenge)
(Image: He et al., tech report on ArXiv)

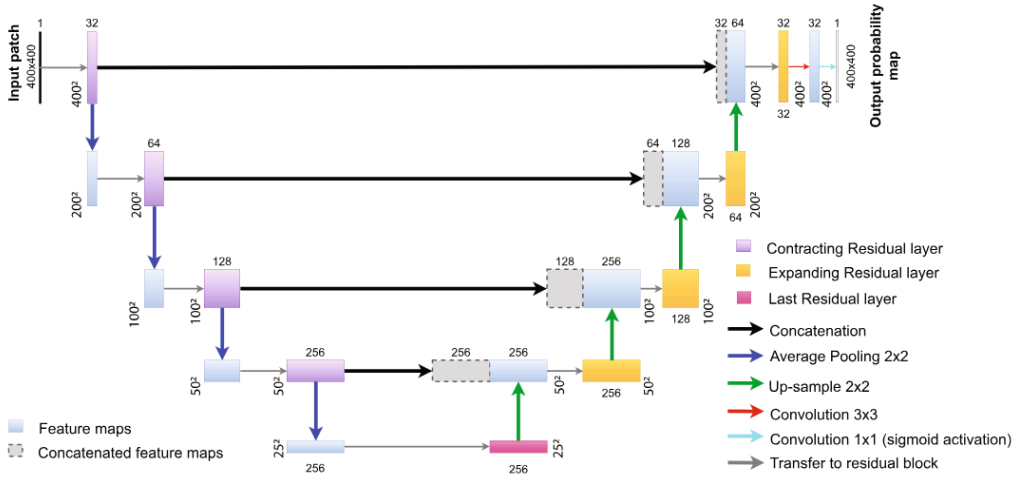
Modern deep learning models almost always include some form of ***residual connections*** as an integral component of their building blocks.

5) U-Net – Proposed in 2015 by Ronneberger et al., U-Net is important because it provided an architecture capable of working to solve pixel-level tasks. The architectures discussed above all assume that the final layer of neurons is carrying out a task that is to be solved at the ***image level*** – a single answer (such as for example, the class the image belongs to) is obtained after considering all the information contained in the input image.

However, many image-related problems demand a solution at the ***pixel level***. For instance, ***semantic image segmentation*** which is the process of ***dividing an image into regions*** each of which corresponds to an ***object from a known class***, and labelling each region with the appropriate class. The output in this case must be an image of the same size as the input, but instead of pixels we expect object labels. Similarly, image restoration tasks such as ***noise removal, de-blurring, contrast adjustment,*** and ***white balancing*** require as output an image of the same size as the input.

Network architectures we have seen so far will not work for these tasks – they succeed specifically by abstracting and condensing meaning from large inputs, and reducing the size of the feature maps progressively. They are not appropriate to solve pixel-level problems.

U-Net provided an architecture capable of addressing processing pipelines whose final output is a pixel map. Originally designed for ***semantic segmentation***, it (or variations of it) have been applied to all sorts of image processing tasks. U-Net consists of ***two symmetric networks attached to one another*** as shown below:



U-Net architecture. The first half of the network is a standard ConvNet in the style of AlexNet or VGG. The second half is a ***mirror image*** with ***residual connections*** whose job (on top of further processing) is to ***up-scale*** the feature maps produced by the first half of the network so as to obtain a full-size result. (Image: Gomez-de-Mariscal et al., CC-BY-4.0)

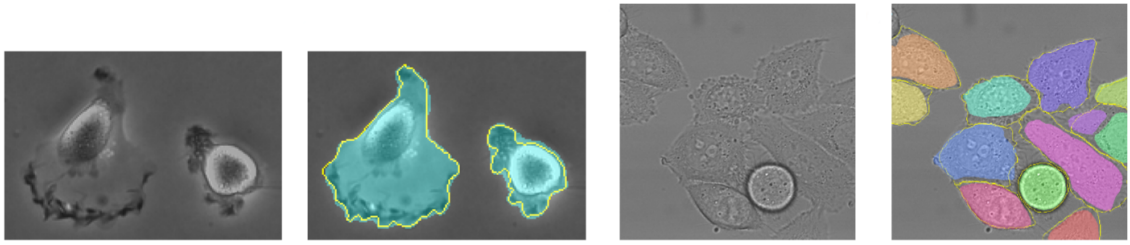
This type of architecture has become very common in applications of convolutional networks. The first half of the network is an ***encoder*** – it is a standard ConvNet similar to ***AlexNet***, and it can have any number and configuration of ***convolutional layers, pooling layers,*** and ***batch normalization***. Just like ***AlexNet*** or ***VGG*** the job of the ***encoder*** is to extract ***meaningful features*** that help the network achieve its goal.

The second half of the network is called the ***decoder*** – its job is to take the ***feature map*** produced by the ***encoder*** and ***produce larger feature maps*** that ***are weighted combinations*** of the lower resolution ***features***. The weights are ***learned by training*** to provide the interpolation from features at lower resolution to features at higher resolution that best helps the network with the task at hand.

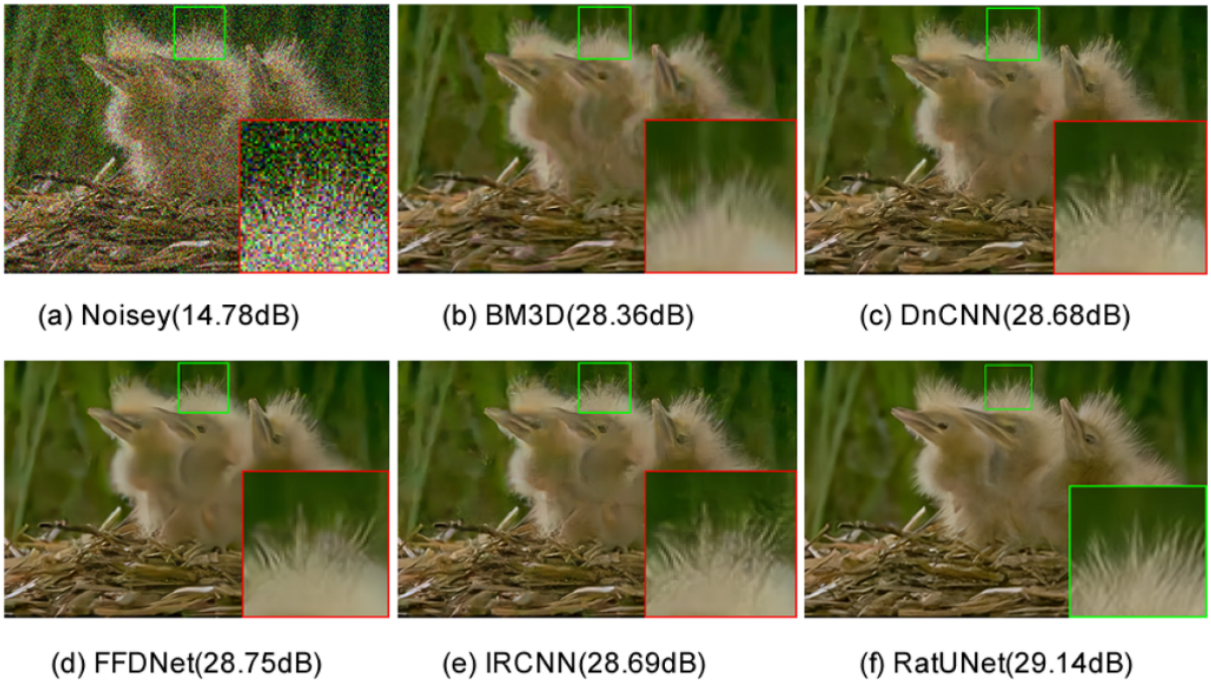
Importantly, each layer in the ***decoder*** works with two sources of information: The ***feature map*** produced by the previous layer, and the ***feature map*** produced by the ***corresponding layer in the encoder*** (this is a ***residual*** connection). These two maps are concatenated, and then processed through a ***1x1xN*** convolution ***kernel*** to produce a ***combined feature map*** with the correct dimensions. Finally, the layer ***up-samples*** the map using its own bank of ***interpolation kernels*** to produce a result twice the size (along the width and height).

A final pair of layers perform additional ***convolutions*** to produce the network's result, which is an image with the same with and height as the input. In the case of the original ***U-Net***, the value for each pixel is a ***label*** (an integer number) identifying a particular object in the input.

Variations of U-Net have been developed for a wide range of applications in ***segmentation***, ***object detection***, and ***image restoration***.



Segmentation results from the original U-Net paper by Ronneberger et al., the segmentation from U-Net is shown as a solid colour, the yellow outline is the ground truth boundary for each cell, as marked by an expert human observer



Results comparing a variant of U-Net (RatUNet) with several competing methods for ***image denoising*** (including alternate deep-network architectures). RatUNet achieves the best results in terms of RMS error (Image: Zhang et al., CC-BY-4.0)

Transfer Learning

A particularly useful property of deep neural networks is that often entire blocks of them can be re-purposed for tasks other than the original one they were trained with. The process of re-training a model to perform a related task so that only a limited amount of computation and training is required to achieve good results is called ***transfer learning***.

Suppose we wanted to train a convolutional network to recognize images of different products offered at a small convenience store (for instance, so that the automatic cashier can recognize produce without the user having to input a product code or scan a barcode).

We could, of course, design and train our own convolutional network in the style of ***AlexNet*** or ***VGG,*** but why not make use of a model that has already been trained to solve a similar task (image category recognition) though on a different dataset?

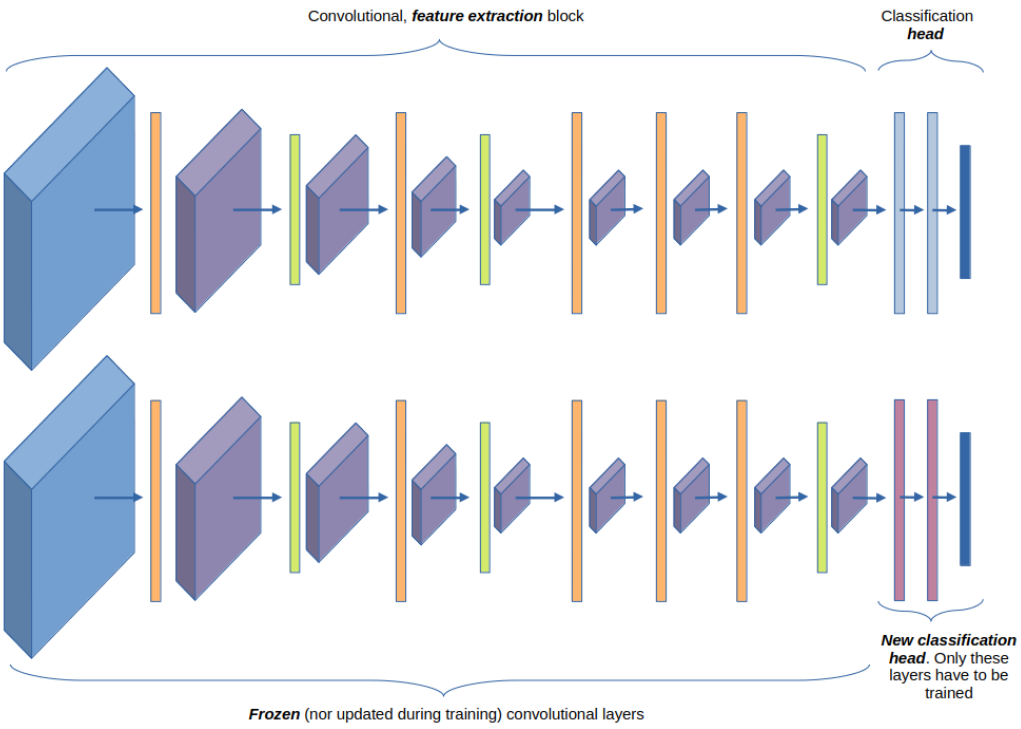
In our example above, we can rely on the following observation:

- The convolutional parts of a model like ***AlexNet, VGG,*** or ***ResNet*** have been trained to be very effective at extracting ***meaningful features*** on ***regular images*** – even though their training comes from a specific dataset, the dataset itself is very large and rich in terms of the diversity of visual information contained in images the network has already seen during training.

- Therefore, we can think of the convolutional layers of such a model as a fairly good ***general purpose feature extraction*** model for image categorization tasks.

- Therefore, we can take the ***pre-trained convolutional layers*** of one of these models, and ***attach a different classification head*** – this means, replacing the final, ***fully-connected network*** with a new one ***that we can train for our specific image categorization task*** (recognizing produce).

- The training process ***only needs to learn the weights of the new classification head*** and does not change any of the weights in the convolutional layers. This means we require ***much less training data***, and the computational expense of the training process is significantly reduced.



The process is shown above with the original ***AlexNet*** – we can ***download a pre-trained model,*** replace the ***classification head***, and re-train only the final, fully-connected part of the network on the new training data. Because we are training only a couple of layers, and because these are the layers closest to the output, training is much easier, faster, and requires less training data to achieve good results.

After the new ***classification head*** has been trained, we an ***optionally*** do some fine-tuning of the ***convolutional layers*** by performing a small amount of training on the full network. This has to be done

only after the new ***classification head*** has become as good as possible with the pre-trained ***feature detection*** block.

Transfer learning has many variants, and the specific sequence of layers that needs to be re-trained depends on the type of model that is being used. However, it has proven to be incredibly useful, and libraries of pre-trained models for a variety of purposes are nowadays readily available for every conceivable platform, language, and task. The ability to quickly re-purpose an existing model for a new (related) task is one of the reasons deep learning has been so successful and gained such wide adoption.

*Acknowledgements: A big **thank you** to Allan Jepson for carefully reading through these notes and providing me with detailed and very precise feedback on how to make them better!*