# Applied Deep Learning - Part 4: Convolutional Neural Networks

Arden Dertat
Nov 8, 2017 · 23 min read

## Overview

Welcome to Part 4 of Applied Deep Learning series. Part 1 was a hands-on introduction to Artificial Neural Networks, covering both the theory and application with a lot of code examples and visualization. In Part 2 we applied deep learning to real-world datasets, covering the 3 most commonly encountered problems as case studies: binary classification, multiclass classification and regression. Part 3 explored a specific deep learning architecture: Autoencoders.

Now we will cover the most popular deep learning model: Convolutional Neural Networks.

1. Introduction
2. Architecture
3. Intuition
4. Implementation
5. VGG Model
6. Visualization
7. Conclusion
8. References

The code for this article is available here as a Jupyter notebook, feel free to download and try it out yourself.

# 1. Introduction

Convolutional Neural Networks (CNN) are everywhere. It is arguably the most popular deep learning architecture. The recent surge of interest in deep learning is due to the immense popularity and effectiveness of convnets. The interest in CNN started with AlexNet in 2012 and it has grown exponentially ever since. In just three years, researchers progressed from 8 layer AlexNet to 152 layer ResNet.

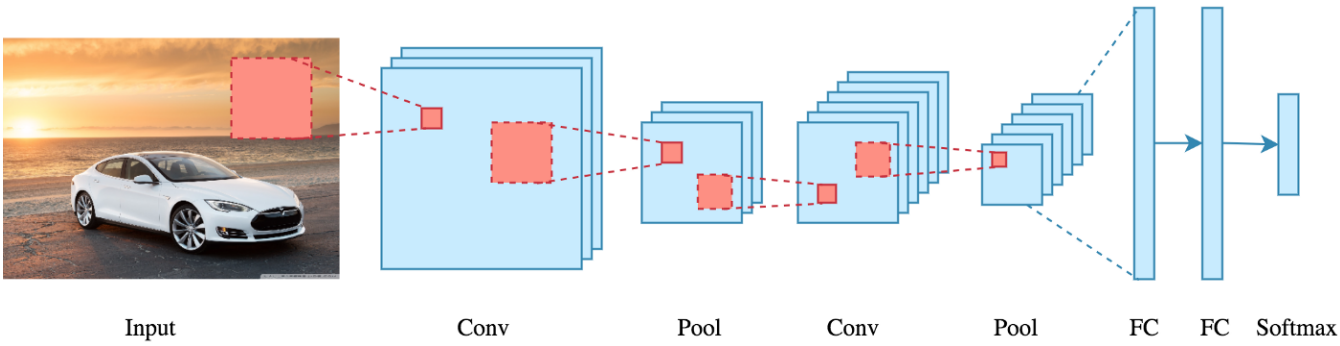
CNN is now the go-to model on every image related problem. In terms of accuracy they blow competition out of the water. It is also successfully applied to recommender systems, natural language processing and more. The main advantage of CNN compared to its predecessors is that it automatically detects the important features without any human supervision. For example, given many pictures of cats and dogs it learns distinctive features for each class by itself.

CNN is also computationally efficient. It uses special convolution and pooling operations and performs parameter sharing. This enables CNN models to run on any device, making them universally attractive.

All in all this sounds like pure magic. We are dealing with a very powerful and efficient model which performs automatic feature extraction to achieve superhuman accuracy (yes CNN models now do image classification better than humans). Hopefully this article will help us uncover the secrets of this remarkable technique.

# 2. Architecture

All CNN models follow a similar architecture, as shown in the figure below.




There is an input image that we're working with. We perform a series convolution + pooling operations, followed by a number of fully connected layers. If we are performing multiclass classification the output is softmax. We will now dive into each component.

## 2.1) Convolution

The main building block of CNN is the convolutional layer. Convolution is a mathematical operation to merge two sets of information. In our case the convolution is applied on the input data using a *convolution filter* to produce a *feature map*. There are a lot of terms being used so let's visualize them one by one.

| | | | | |
|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |

Input

| | | |
|---|---|---|
| 1 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |

Filter / Kernel

On the left side is the input to the convolution layer, for example the input image. On the right is the convolution *filter*, also called the *kernel*, we will use these terms interchangeably. This is called a *3x3 convolution* due to the shape of the filter.

We perform the convolution operation by sliding this filter over the input. At every location, we do element-wise matrix multiplication and sum the result. This sum goes into the feature map. The green area where the convolution operation takes place is called the *receptive field*. Due to the size of the filter the receptive field is also 3x3.

| | | | | |
|---|---|---|---|---|
| 1x1 | 1x0 | 1x1 | 0 | 0 |
| 0x0 | 1x1 | 1x0 | 1 | 0 |
| 0x1 | 0x0 | 1x1 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |

| | | |
|---|---|---|
| 4 | | |
| | | |
| | | |

Input x Filter Feature Map

Here the filter is at the top left, the output of the convolution operation “4” is shown in the resulting feature map. We then slide the filter to the right and perform the same operation, adding that result to the feature map as well.

| | | | | |
|---|---|---|---|---|
| 1 | 1x1 | 1x0 | 0x1 | 0 |
| 0 | 1x0 | 1x1 | 1x0 | 0 |
| 0 | 0x1 | 1x0 | 1x1 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |

| | | |
|---|---|---|
| 4 | 3 | |
| | | |
| | | |

Input x Filter Feature Map

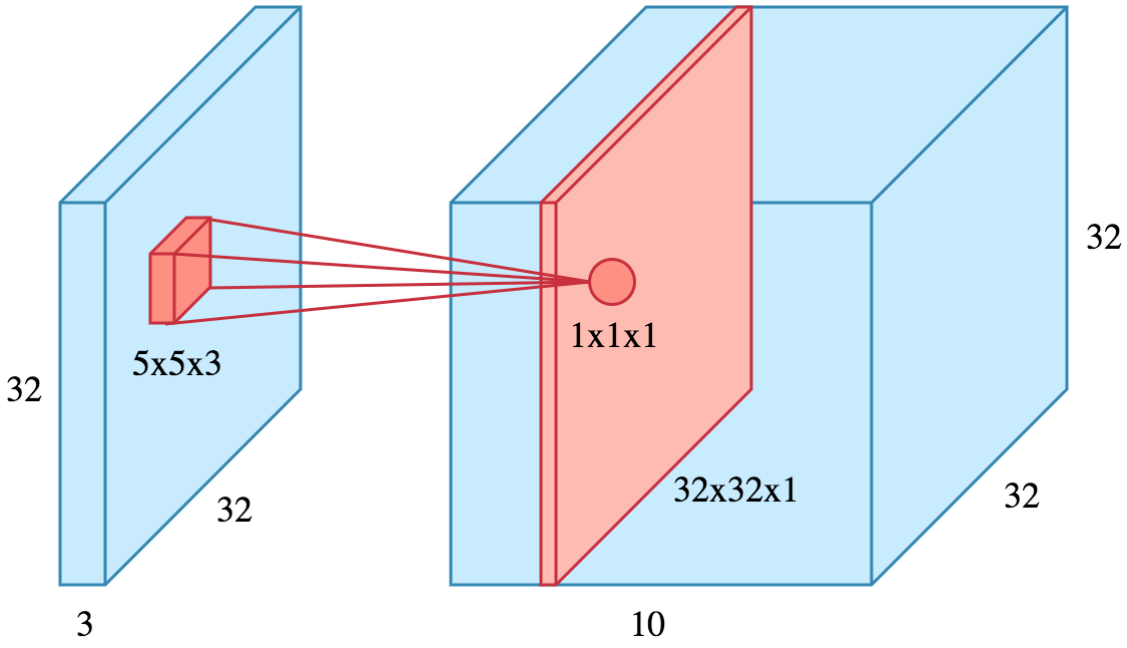
We continue like this and aggregate the convolution results in the feature map. Here’s an animation that shows the entire convolution operation.

| | | | | |
|---|---|---|---|---|
| 1x1 | 1x0 | 1x1 | 0 | 0 |
| 0x0 | 1x1 | 1x0 | 1 | 0 |
| 0x1 | 0x0 | 1x1 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |

| | | |
|---|---|---|
| 4 | | |
| | | |
| | | |

This was an example convolution operation shown in 2D using a 3x3 filter. But in reality these convolutions are performed in 3D. In reality an image is represented as a 3D matrix with dimensions of height, width and depth, where depth corresponds to color channels (RGB). A convolution filter has a specific height and width, like 3x3 or 5x5, and by design it covers the entire depth of its input so it needs to be 3D as well.

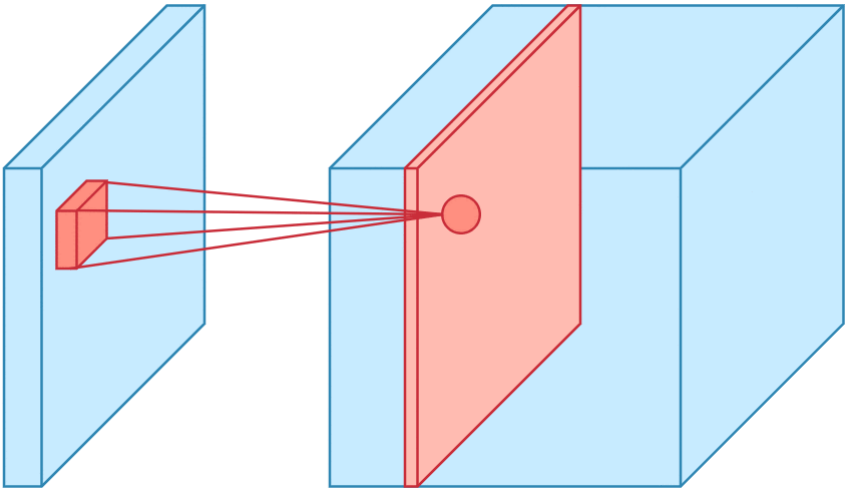
One more important point before we visualize the actual convolution operation. We perform multiple convolutions on an input, each using a different filter and resulting in a distinct feature map. We then stack all these feature maps together and that becomes the final output of the convolution layer. But first let's start simple and visualize a convolution using a single filter.



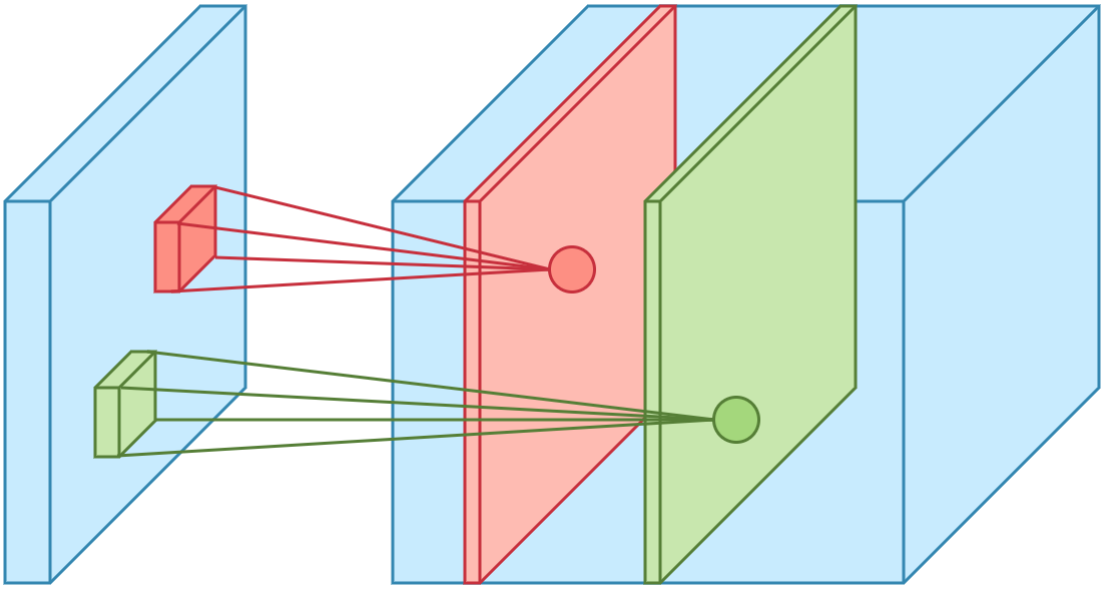

Let's say we have a 32x32x3 image and we use a filter of size 5x5x3 (note that the depth of the convolution filter matches the depth of the image, both being 3). When the filter is at a particular location it covers a small volume of the input, and we perform the convolution operation described above. The only difference is this time we do the sum of matrix multiply in 3D instead of 2D, but the result is still a scalar. We slide the filter over the input like above and perform the convolution at every location aggregating the result in a feature map. This feature map is of size 32x32x1, shown as the red slice on the right.

If we used 10 different filters we would have 10 feature maps of size 32x32x1 and stacking them along the depth dimension would give us the final output of the convolution layer: a volume of size 32x32x10, shown as the large blue box on the right. Note that the height and width of the feature map are unchanged and still 32, it's due to padding and we will elaborate on that shortly.

To help with visualization, we slide the filter over the input as follows. At each location we get a scalar and we collect them in the feature map. The animation shows the sliding operation at 4 locations, but in reality it's performed over the entire input.



Below we can see how two feature maps are stacked along the depth dimension. The convolution operation for each filter is performed independently and the resulting feature maps are disjoint.

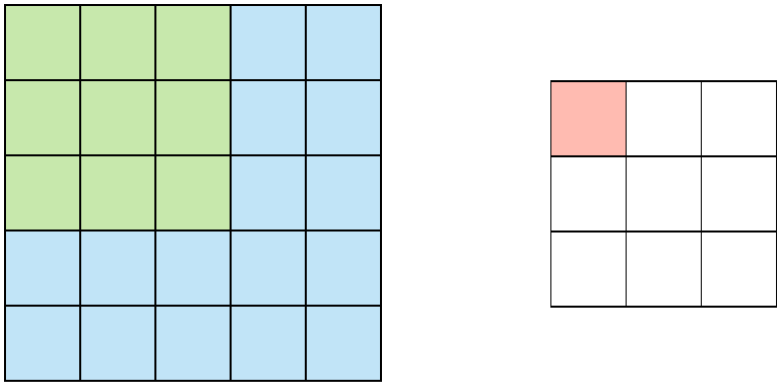


## 2.2) Non-linearity

For any kind of neural network to be powerful, it needs to contain non-linearity. Both the ANN and autoencoder we saw before achieved this by passing the weighted sum of its inputs through an activation function, and CNN is no different. We again pass the result of the convolution operation through *relu* activation function. So the values in the final feature maps are not actually the sums, but the relu function applied to them. We have omitted this in the figures above for simplicity. But keep in mind that any type of convolution involves a relu operation, without that the network won't achieve its true potential.

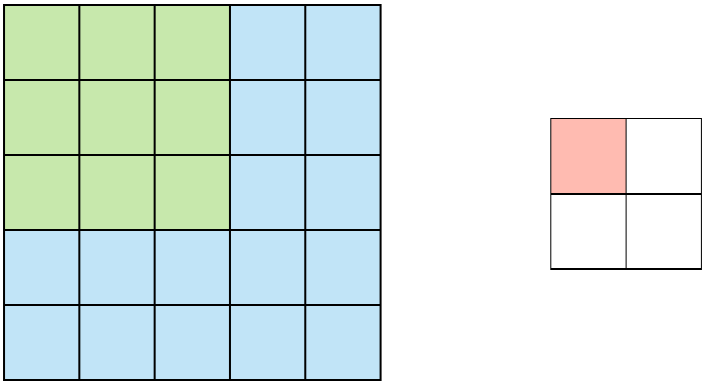
## 2.3) Stride and Padding

*Stride* specifies how much we move the convolution filter at each step. By default the value is 1, as you can see in the figure below.



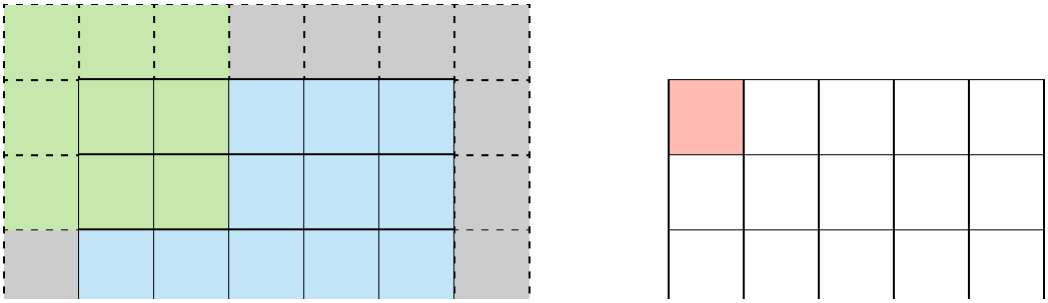
Stride 1 Feature Map

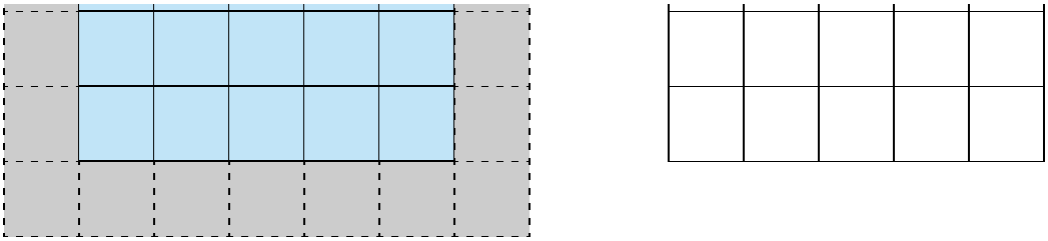
We can have bigger strides if we want less overlap between the receptive fields. This also makes the resulting feature map smaller since we are skipping over potential locations. The following figure demonstrates a stride of 2. Note that the feature map got smaller.



Stride 2 Feature Map

We see that the size of the feature map is smaller than the input, because the convolution filter needs to be contained in the input. If we want to maintain the same dimensionality, we can use *padding* to surround the input with zeros. Check the animation below.





Stride 1 with Padding  Feature Map

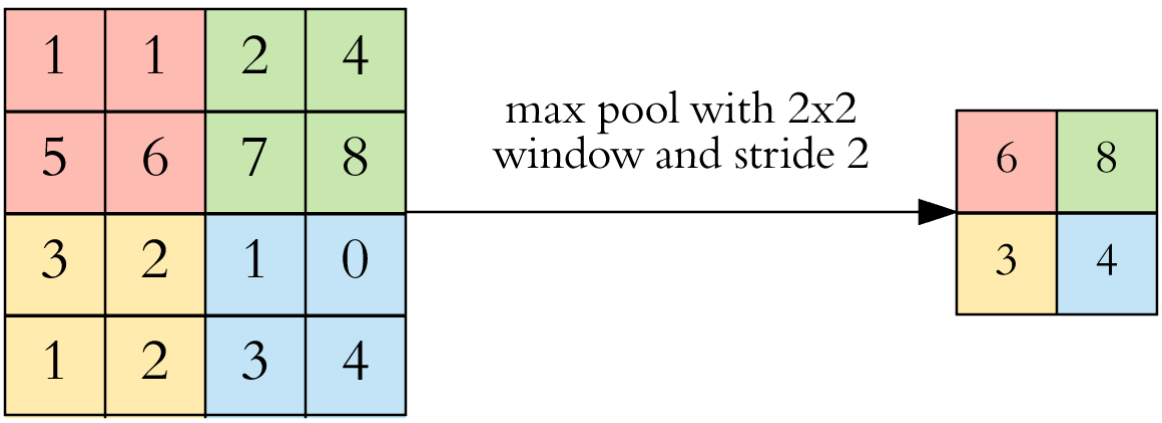
The gray area around the input is the padding. We either pad with zeros or the values on the edge. Now the dimensionality of the feature map matches the input. Padding is commonly used in CNN to preserve the size of the feature maps, otherwise they would shrink at each layer, which is not desirable. The 3D convolution figures we saw above used padding, that's why the height and width of the feature map was the same as the input (both 32x32), and only the depth changed.

## 2.4) Pooling

After a convolution operation we usually perform *pooling* to reduce the dimensionality. This enables us to reduce the number of parameters, which both shortens the training time and combats overfitting. Pooling layers downsample each feature map independently, reducing the height and width, keeping the depth intact.

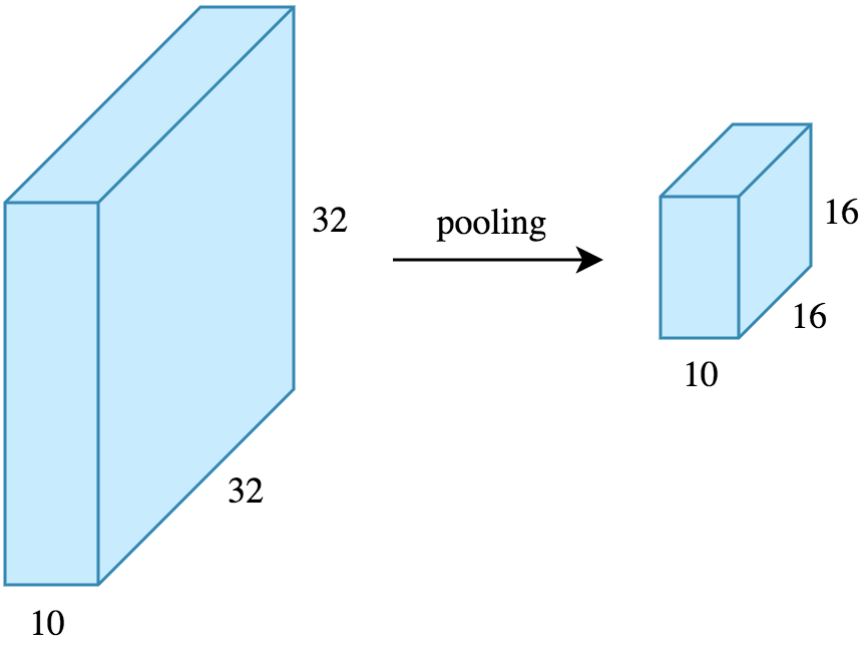
The most common type of pooling is *max pooling* which just takes the max value in the pooling window. Contrary to the convolution operation, pooling has no parameters. It slides a window over its input, and simply takes the max value in the window. Similar to a convolution, we specify the window size and stride.

Here is the result of max pooling using a 2x2 window and stride 2. Each color denotes a different window. Since both the window size and stride are 2, the windows are not overlapping.



Note that this window and stride configuration halves the size of the feature map. This is the main use case of pooling, downsampling the feature map while keeping the important information.

Now let's work out the feature map dimensions before and after pooling. If the input to the pooling layer has the dimensionality 32x32x10, using the same pooling parameters described above, the result will be a 16x16x10 feature map. Both the height and width of the feature map are halved, but the depth doesn't change because pooling works independently on each depth slice the input.




By halving the height and the width, we reduced the number of weights to 1/4 of the input. Considering that we typically deal with millions of weights in CNN architectures, this reduction is a pretty big deal.

In CNN architectures, pooling is typically performed with 2x2 windows, stride 2 and no padding. While convolution is done with 3x3 windows, stride 1 and with padding.

## 2.5) Hyperparameters

Let's now only consider a convolution layer ignoring pooling, and go over the hyperparameter choices we need to make. We have 4 important hyperparameters to decide on:

- Filter size: we typically use 3x3 filters, but 5x5 or 7x7 are also used depending on the application. There are also 1x1 filters which we will explore in another article, at first sight it might look strange but they have interesting applications. Remember

that these filters are 3D and have a depth dimension as well, but since the depth of a filter at a given layer is equal to the depth of its input, we omit that.

- Filter count: this is the most variable parameter, it's a power of two anywhere between 32 and 1024. Using more filters results in a more powerful model, but we risk overfitting due to increased parameter count. Usually we start with a small number of filters at the initial layers, and progressively increase the count as we go deeper into the network.
- Stride: we keep it at the default value 1.
- Padding: we usually use padding.

### 2.6) Fully Connected

After the convolution + pooling layers we add a couple of fully connected layers to wrap up the CNN architecture. This is the same fully connected ANN architecture we talked about in Part 1.

Remember that the output of both convolution and pooling layers are 3D volumes, but a fully connected layer expects a 1D vector of numbers. So we *flatten* the output of the final pooling layer to a vector and that becomes the input to the fully connected layer. Flattening is simply arranging the 3D volume of numbers into a 1D vector, nothing fancy happens here.

### 2.7) Training

CNN is trained the same way like ANN, backpropagation with gradient descent. Due to the convolution operation it's more mathematically involved, and it's out of the scope for this article. If you're interested in the details refer here.

## 3. Intuition

A CNN model can be thought as a combination of two components: feature extraction part and the classification part. The convolution + pooling layers perform feature extraction. For example given an image, the convolution layer detects features such as two eyes, long ears, four legs, a short tail and so on. The fully connected layers then act as a classifier on top of these features, and assign a probability for the input image being a dog.

The convolution layers are the main powerhouse of a CNN model. Automatically detecting meaningful features given only an image and a label is not an easy task. The

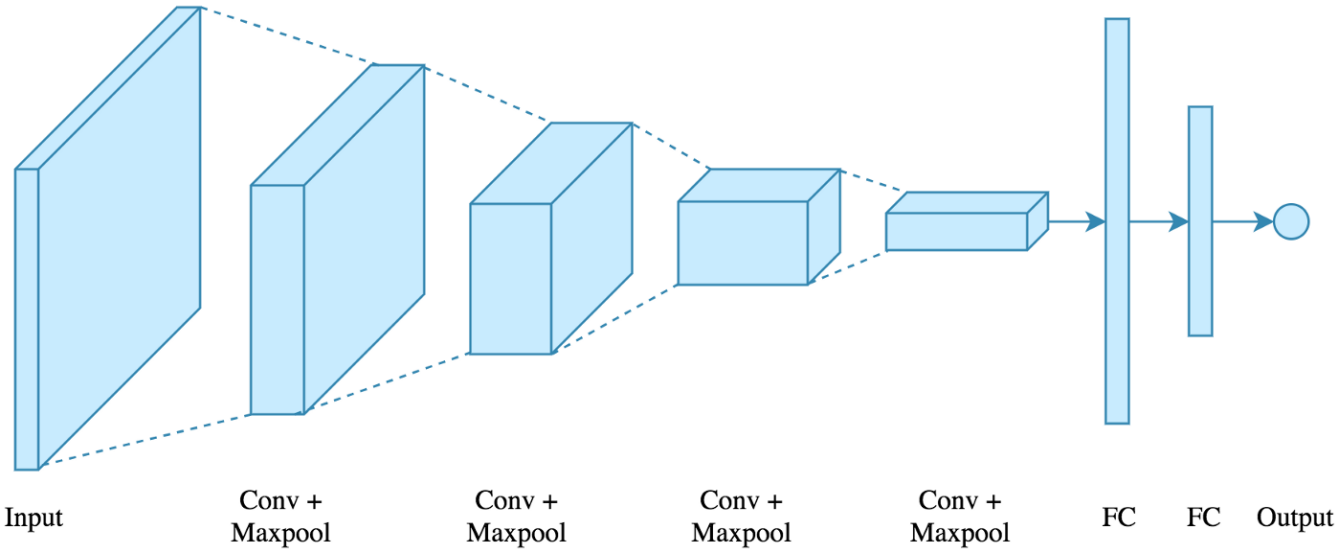
convolution layers learn such complex features by building on top of each other. The first layers detect edges, the next layers combine them to detect shapes, to following layers merge this information to infer that this is a nose. To be clear, the CNN doesn't know what a nose is. By seeing a lot of them in images, it learns to detect that as a feature. The fully connected layers learn how to use these features produced by convolutions in order to correctly classify the images.

All this might sound vague right now, but hopefully the visualization section will make everything more clear.

## 4. Implementation

After this lengthy explanation let's code up our CNN. We will use the Dogs vs Cats dataset from Kaggle to distinguish dog photos from cats.

We will use the following architecture: 4 convolution + pooling layers, followed by 2 fully connected layers. The input is an image of a cat or dog and the output is binary.




Here's the code for the CNN:

Structurally the code looks similar to the ANN we have been working on. There are 4 new methods we haven't seen before:

- *Conv2D*: this method creates a convolutional layer. The first parameter is the filter count, and the second one is the filter size. For example in the first convolution layer we create 32 filters of size 3x3. We use *relu* non-linearity as activation. We also enable padding. In Keras there are two options for padding: *same* or *valid*.

Same means we pad with the number on the edge and valid means no padding. Stride is 1 for convolution layers by default so we don't change that. This layer can be customized further with additional parameters, you can check the documentation here.

- *MaxPooling2D*: creates a maxpooling layer, the only argument is the window size. We use a 2x2 window as it's the most common. By default stride length is equal to the window size, which is 2 in our case, so we don't change that.
- *Flatten*: After the convolution + pooling layers we flatten their output to feed into the fully connected layers as we discussed above.
- *Dropout*: we will explain this in the next section.

### 4.1) Dropout

Dropout is by far the most popular regularization technique for deep neural networks. Even the state-of-the-art models which have 95% accuracy get a 2% accuracy boost just by adding dropout, which is a fairly substantial gain at that level.

Dropout is used to prevent overfitting and the idea is very simple. During training time, at each iteration, a neuron is temporarily "dropped" or disabled with probability $p$. This means all the inputs and outputs to this neuron will be disabled at the current iteration. The dropped-out neurons are resampled with probability p at every training step, so a dropped out neuron at one step can be active at the next one. The hyperparameter $p$ is called the *dropout-rate* and it's typically a number around 0.5, corresponding to 50% of the neurons being dropped out.

It's surprising that dropout works at all. We are disabling neurons on purpose and the network actually performs better. The reason is that dropout prevents the network to be too dependent on a small number of neurons, and forces every neuron to be able to operate independently. This might sound familiar from constraining the code size of the autoencoder in Part 3, in order to learn more intelligent representations.

Let's visualize dropout, it will be much easier to understand.

Dropout can be applied to input or hidden layer nodes but not the output nodes. The edges in and out of the dropped out nodes are disabled. Remember that the nodes which were dropped out change at each training step. Also we don't apply dropout during test time after the network is trained, we do so only in training.

A real-life analogy to dropout would be as follows: let's say you were the only person at your company who knows about finance. If you were guaranteed to be at work every day, your coworkers wouldn't have an incentive to pick up finance skills. But if every morning you tossed a coin to decide whether you will go to work or not, then your coworkers will need to adapt. Some days you might not be at work but finance tasks still need to get done, so they can't rely only at you. Your coworkers will need to learn about finance and this expertise needs to be spread out between various people. The workers need to cooperate with several other employees, not with a fixed set of people. This makes the company much more resilient overall, increasing the quality and skillset of the employees.

Almost all state-of-the-art deep networks now incorporate dropout. There is another very popular regularization technique called *batch normalization* and we will cover it in another article.

**4.2) Model Performance**

Let's now analyze the performance of our model. We will take a look at loss and accuracy curves, comparing training set performance against the validation set.

Training loss keeps going down but the validation loss starts increasing after around epoch 10. This is the textbook definition of overfitting. The model is memorizing the training data, but it's failing to generalize to new instances, and that's why the validation performance goes worse.

We are overfitting despite the fact that we are using dropout. The reason is we are training on very few examples, 1000 images per category. Usually we need at least 100K training examples to start thinking about deep learning. No matter which regularization technique we use, we will overfit on such a small dataset. But fortunately there is a solution to this problem which enables us to train deep models on small datasets, and it's called *data augmentation*.

## 4.3) Data Augmentation

Overfitting happens because of having too few examples to train on, resulting in a model that has poor generalization performance. If we had infinite training data, we wouldn't overfit because we would see every possible instance.

The common case in most machine learning applications, especially in image classification tasks is that obtaining new training data is not easy. Therefore we need to make do with the training set at hand. Data augmentation is a way to generate more training data from our current set. It enriches or "augments" the training data by generating new examples via random transformation of existing ones. This way we

artificially boost the size of the training set, reducing overfitting. So data augmentation can also be considered as a regularization technique.

Data augmentation is done dynamically during training time. We need to generate realistic images, and the transformations should be learnable, simply adding noise won't help. Common transformations are: rotation, shifting, resizing, exposure adjustment, contrast change etc. This way we can generate a lot of new samples from a single training example. Also, data augmentation is only performed on the training data, we don't touch the validation or test set.

Visualization will help understanding the concept. Let's say this is our original image.

Using data augmentation we generate these artificial training instances. These are new training instances, applying transformations on the original image doesn't change the fact that this is still a cat image. We can infer it as a human, so the model should be able to learn that as well.

Data augmentation can boost the size of the training set by even 50x. It's a very powerful technique that is used in every single image-based deep learning model, no exceptions.

There are some data cleaning tricks that we typically use on images, mainly *whitening* and *mean normalization*. More information about them is available here.

## 4.4) Updated Model

Now let's use data augmentation in our CNN model. The code for the model definition will not change at all, since we're not changing the architecture of our model. The only change is how we feed in the data, you can check the jupyter notebook here.

It's pretty easy to do data augmentation with Keras, it provides a class which does all the work for us, we only need to specify some parameters. The documentation is available here.

The loss and accuracy curves look as follows using data augmentation.

This time there is no apparent overfitting. And validation accuracy jumped from 73% with no data augmentation to 81% with data augmentation, 11% improvement. This is a pretty big deal. There are two main reasons that the accuracy improved. First, we are training on more images with variety. Second, we made the model transformation invariant, meaning the model saw a lot of shifted/rotated/scaled images so it's able to recognize them better.

## 5. VGG Model

Let's now take a look at an example state-of-the art CNN model from 2014. VGG is a convolutional neural network from researchers at Oxford's Visual Geometry Group, hence the name VGG. It was the runner up of the ImageNet classification challenge with 7.3% error rate. ImageNet is the most comprehensive hand-annotated visual dataset, and they hold competitions every year where researchers from all around the world compete. All the famous CNN architectures make their debut at that competition.

Among the best performing CNN models, VGG is remarkable for its simplicity. Let's take a look at its architecture.

VGG is a 16 layer neural net, not counting the maxpool layers and the softmax at the end. It's also referred to as VGG16. The architecture is the one we worked with above. Stacked convolution + pooling layers followed by fully connected ANN. A few observations about the architecture:

- It only uses 3x3 convolutions throughout the network. Note that two back to back 3x3 convolutions have the effective receptive field of a single 5x5 convolution. And three stacked 3x3 convolutions have the receptive field of a single 7x7 one. Here's the visualization of two stacked 3x3 convolutions resulting in 5x5.

- Another advantage of stacking two convolutions instead of one is that we use two relu operations, and more non-linearity gives more power to the model.
- The number of filters increase as we go deeper into the network. The spatial size of the feature maps decrease since we do pooling, but the depth of the volumes increase as we use more filters.
- Trained on 4 GPUs for 3 weeks.

VGG is a very fundamental CNN model. It's the first one that comes to mind if you need to use an off-the-shelf model for a particular task. The paper is also very well written, available here. There are much more complicated models which perform better, for example Microsoft's ResNet model was the winner of 2015 ImageNet challenge with 3.6% error rate, but the model has 152 layers! Details available in the paper here. We will cover all these CNN architectures in depth in another article, but if you want to jump ahead here is a great post.

## 6. Visualization

Now comes the most fun and interesting part, visualization of convolutional neural networks. Deep learning models are known to be very hard to interpret, that's why they are usually treated as black boxes. But CNN models are actually the opposite, and we can visualize various components. This will give us an in depth look into their internal workings and help us understand them better.

We will visualize the 3 most crucial components of the VGG model:

- Feature maps
- Convnet filters
- Class output

### 6.1) Visualizing Feature Maps

Let's quickly recap the convolution architecture as a reminder. A convnet filter operates on the input performing the convolution operation and as a result we get a feature map. We use multiple filters and stack the resulting feature maps together to obtain an output volume. First we will visualize the feature maps, and in the next section we will explore the convnet filters.

We will visualize the feature maps to see how the input is transformed passing through the convolution layers. The feature maps are also called *intermediate activations* since the output of a layer is called the activation.

Remember that the output of a convolution layer is a 3D volume. As we discussed above the height and width correspond to the dimensions of the feature map, and each depth channel is a distinct feature map encoding independent features. So we will visualize individual feature maps by plotting each channel as a 2D image.

How to visualize the feature maps is actually pretty simple. We pass an input image through the CNN and record the intermediate activations. We then randomly select some of the feature maps and plot them.

VGG convolutional layers are named as follows: blockX_convY. For example the second filter in the third convolution block is called block3_conv2. In the architecture diagram above it corresponds to the second purple filter.

For example one of the feature maps from the output of the very first layer (block1_conv1) looks as follows.

Bright areas are the “activated” regions, meaning the filter detected the pattern it was looking for. This filter seems to encode an eye and nose detector.

Instead of looking at a single feature map, it would be more interesting to visualize multiple feature maps from a convolution layer. So let's visualize the feature maps corresponding to the first convolution of each block, the red arrows in the figure below.



The following figure displays 8 feature maps per layer. Block1_conv1 actually contains 64 feature maps, since we have 64 filters in that layer. But we are only visualizing the first 8 per layer in this figure.

There are some interesting observations about the feature maps as we progress through the layers. Let's take a look at one feature map per layer to make it more obvious.

- The first layer feature maps (block1_conv1) retain most of the information present in the image. In CNN architectures the first layers usually act as edge detectors.
- As we go deeper into the network, the feature maps look less like the original image and more like an abstract representation of it. As you can see in block3_conv1 the cat is somewhat visible, but after that it becomes unrecognizable. The reason is that deeper feature maps encode high level concepts like "cat nose" or "dog ear" while lower level feature maps detect simple edges and shapes. That's why deeper feature maps contain less information about the image and more about the class of the image. They still encode useful features, but they are less visually interpretable by us.
- The feature maps become sparser as we go deeper, meaning the filters detect less features. It makes sense because the filters in the first layers detect simple shapes,

and every image contains those. But as we go deeper we start looking for more complex stuff like “dog tail” and they don’t appear in every image. That’s why in the first figure with 8 filters per layer, we see more of the feature maps as blank as we go deeper (block4_conv1 and block5_conv1).

### 6.2) Visualizing Convnet Filters

Now we will visualize the main building block of a CNN, the filters. There is one catch though, we won’t actually visualize the filters themselves, but instead we will display the patterns each filter maximally responds to. Remember that the filters are of size 3x3 meaning they have the height and width of 3 pixels, pretty small. So as a proxy to visualizing a filter, we will generate an input image where this filter activates the most.

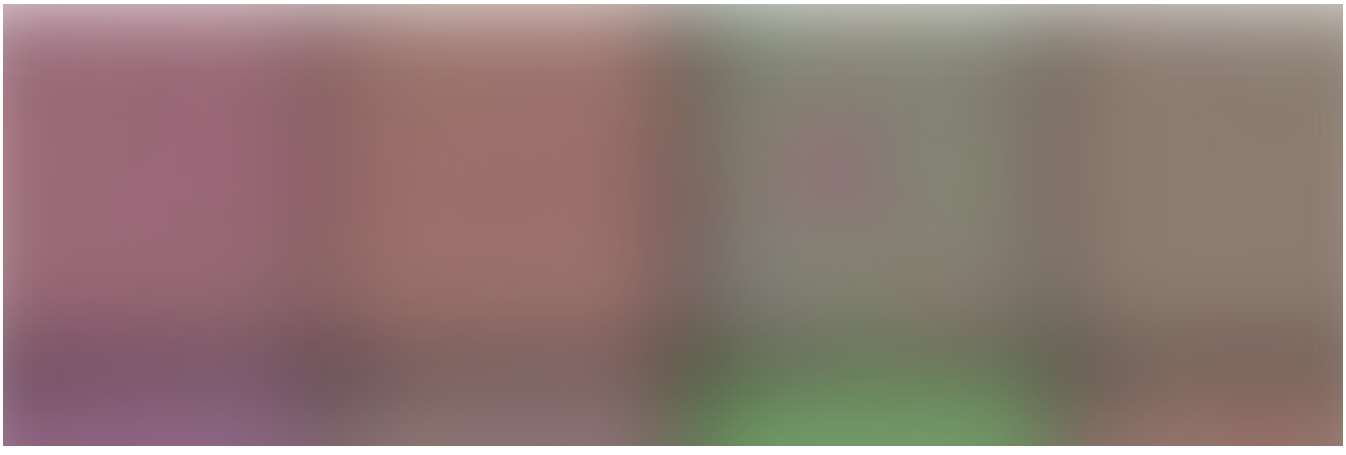
The full details of how to do this is somewhat technical, and you can check the actual code in the jupyter notebook. But as a quick summary it works as follows:

- Choose a loss function that maximizes the value of a convnet filter.
- Start from a blank input image.
- Do *gradient ascent* in input space. Meaning modify the input values such that the filter activates even more.
- Repeat this in a loop.

The result of this process is an input image where the filter is very active. Remember that each filter acts as a detector for a particular feature. The input image we generate will contain a lot of these features.

We will visualize filters at the last layer of each convolution block. To clear any confusion, in the previous section we visualized the feature maps, the output of the convolution operation. Now we are visualizing the filters, the main structure used in the convolution operation.

We will visualize 8 filters per layer.



They look pretty surreal! Especially the ones in the last layers. Here are some observations about the filters:

- The first layer filters (block1_conv2 and block2_conv2) mostly detect colors, edges and simple shapes.
- As we go deeper into the network, the filters build on top of each other, and learn to encode more complex patterns. For example filter 41 in block5_conv3 seems to be a bird detector. You can see multiple heads in different orientations, because the particular location of the bird in the image is not important, as long as it appears somewhere the filter will activate. That's why the filter tries to detect the bird head in several positions by encoding it in multiple locations in the filter.

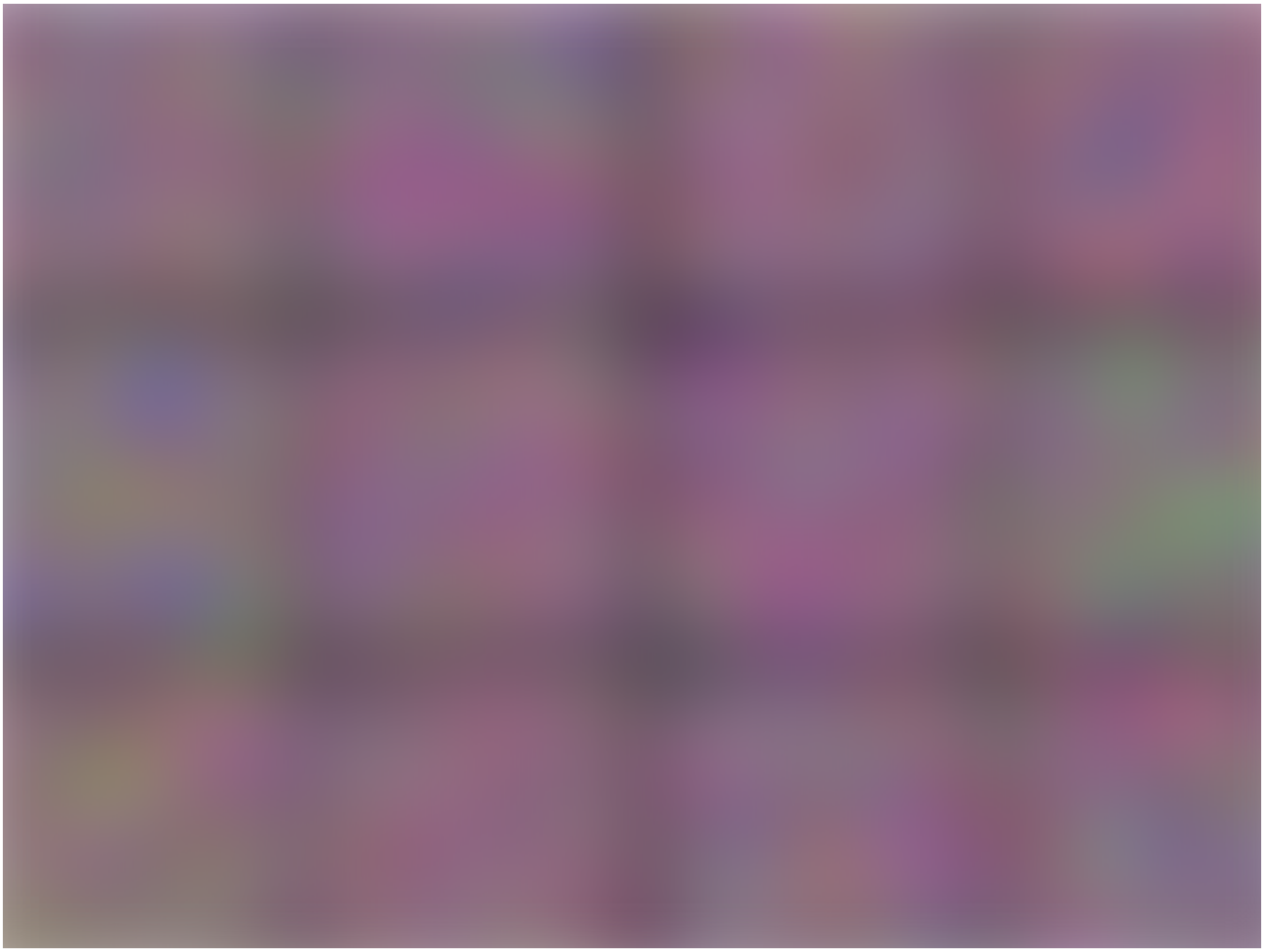
These observations are similar to what we discussed in the feature map section. Lower layers encode/detect simple structures, as we go deeper the layers build on top of each other and learn to encode more complex patterns. This is how we humans start discovering the world as babies too. First we learn simple structures and with practice we excel at understanding more complicated things, building on top of our existing knowledge.

## 6.3) Visualizing Class Outputs

Let's do one final visualization, it will be similar to the convnet filter. Now we will visualize at the final softmax layer. Given a particular category, like hammer or lamp, we will ask the CNN to generate an image that maximally represents the category. Basically the CNN will draw us an image of what it thinks a hammer looks like.

The process is similar to the convnet filter steps. We start from a blank image and do modifications such that the probability assigned to a particular category increases. The code is again available in the notebook.

Let’s visualize some categories.



They look pretty convincing. An object appears several times in the image, because the class probability becomes higher that way. Multiple tennis balls in an image is better than a single tennis ball.

All these visualizations were performed using the library keras-vis.

## 7. Conclusion

CNN is a very fundamental deep learning technique. We covered a wide range of topics and the visualization section in my opinion is the most interesting. There are very few resources on the web which do a thorough visual exploration of convolution filters and feature maps. I hope it was helpful.

The entire code for this article is available here if you want to hack on it yourself. If you have any feedback feel free to reach out to me on twitter.

## 8. References

There is an abundance of CNN tutorials on the web, but the most comprehensive one is the Stanford CS231N course by Andrej Karpathy. The reading material is available here, and the video lectures are here. Excellent source of information, highly recommended.

If you want to dig more into visualization Deepvis is a great resource available here. There is an interactive tool, open source code, paper and a detailed article.

A great image classification tutorial from the author of Keras is available here. This article was highly influenced that tutorial. It covers a lot of the material we talked about in detail.

A very thorough online free book about deep learning can be found here, with the CNN section available here.

If you're interested in applying CNN to natural language processing, this is a great article. Another very detailed one is available here.

A 3-part article series covering state-of-the are CNN papers can be found here.

All articles of Chris Olah are packed with great information and visualizations. CNN related posts are available here and here.

Another popular CNN introductory post is here.

A very accessible 3-part series about CNN is available here.