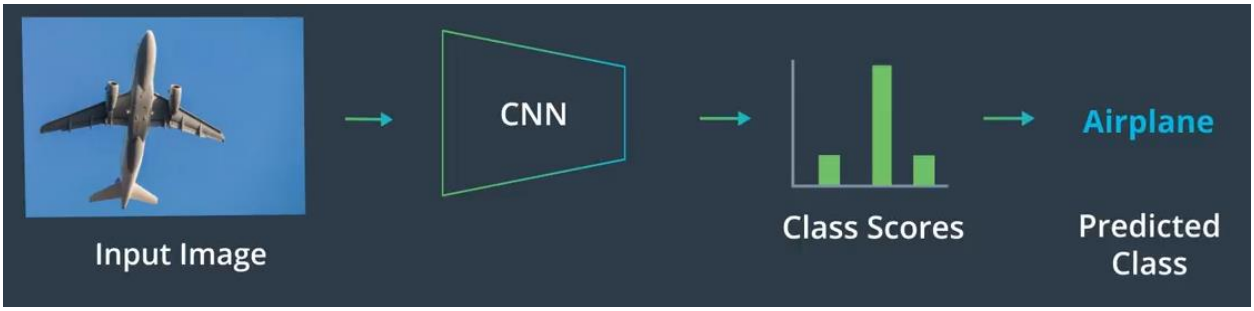
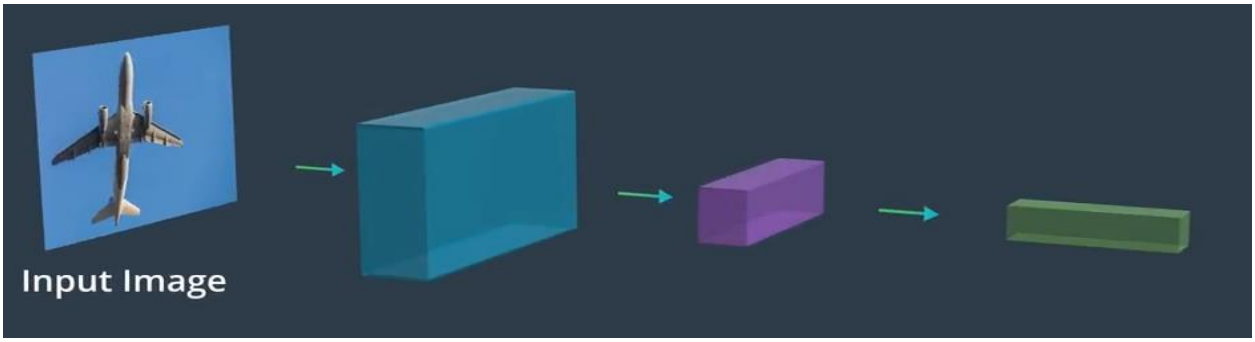
CNN_FEATURE VISUALIZATION

How to extract shape and color features from image in the example you `re gone through, it was up to you to decide what features and filters were the most useful for grouping pixel data into similar cluster or classes This is similar how Convolutional Neural Networks or CNN`s learned to recognize patterns in images

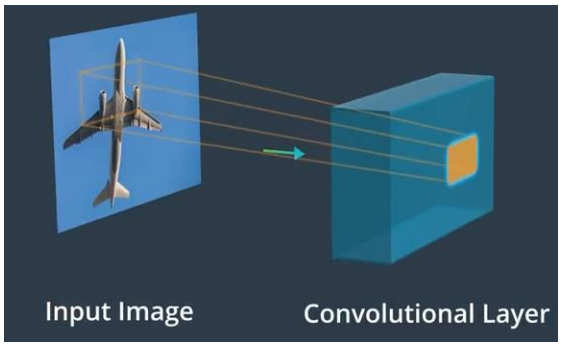
For example , say we `re creating a CNN for image Classifications , this CNN should take in image as input and output a distribution of class scores from which we can get the predicated class for that image.



The CNN made a series of layers to extract relevant features out of any image

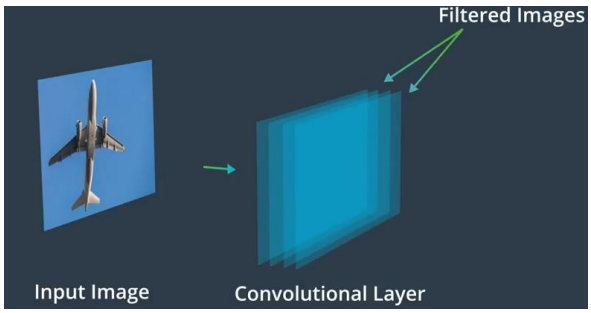


The backbone of a CNN is the convolutional layer.



CNN_FEATURE VISUALIZATION

The Convolutional layer applies a series of a different image filter also known as convolutional kernel to an input



The resulting filtered images have different appearances

The filters may have extracted feature like the edge of objects in that image or the colors that distinguish classes of image.

As the CNN trains , it updates the weights that define the image filters in the convolutional layer using Backpropagation.

The end result is a classifier with conventional Layers that have learned to filter images to extract distinguishing features

Data and lesson Outline Render

We will talk about the layers that make up an image classification CNN.

You will learn how to define these layers , and what role each plays in extracting information from an input image After learning about these layers , you will see how to define a CNN that aims to classify image from FashionMNIST data set .

This is a clothing data set that is made of thousands of image of clothing types like T-shirt , coats , sandals , and sneakers .

The images in this data set are small in size and preprocessed foe ease of training .

We will work with this data set so that you can really focus on defining a CNN as opposed to data preprocessing, and so that you can test your neural network without needing use a GPU as you might with larger images .

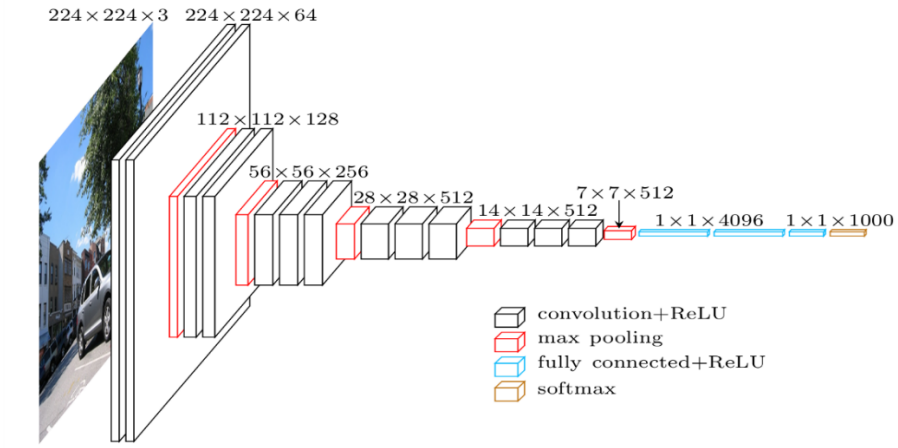
Finally after defining and training a CNN for classifying items of clothing you will learn and implement some feature visualizing techniques that allow to you see what kind of features your model learn to look for , and what it see as an image moves through the layers of a CNN Let`s begin by learning about all the layers that make up a CNN.

CNN Architecture, VGG-16

Convolutional Neural Networks (CNN's)

The type of deep neural network that is most powerful in image processing tasks, such as sorting images into groups, is called a Convolutional Neural Network (CNN). CNN's consist of layers that process visual information. A CNN first takes in an input image and then passes it through these layers. There are a few different types of layers, and we'll start by learning about the most commonly used layers: convolutional, pooling, and fully-connected layers.

First, let's take a look at a complete CNN architecture; below is a network called VGG-16, which has been trained to recognize a variety of image classes. It takes in an image as input, and outputs a predicted class for that image. The various layers are labeled and we'll go over each type of layer in this network in the next series of videos.



VGG-16 architecture

Convolutional Layer

The first layer in this network, that processes the input image directly, is a convolutional layer.

- A convolutional layer takes in an image as input.
- A convolutional layer, as its name suggests, is made of a set of convolutional filters (which you've already seen and programmed).
- Each filter extracts a specific kind of feature, ex. a **high-pass filter** is often used to detect the edge of an object.
- The output of a given convolutional layer is a set of **feature maps** (also called activation maps), which are filtered versions of an original input image.

Activation Function

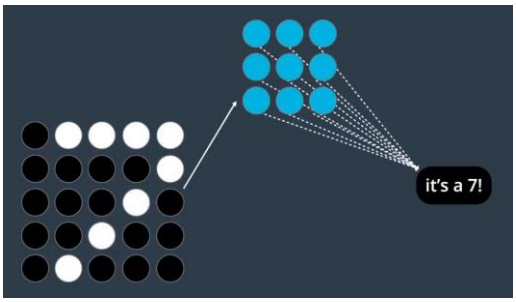
You may also note that the diagram reads "convolution + ReLu," and the **ReLu** stands for Rectified Linear Unit (ReLU) activation function. This activation function is zero when the input x <= 0 and then linear with a slope = 1 when x > 0. ReLu's, and other activation functions, are typically placed after a convolutional layer to slightly transform the output so that it's more efficient to perform backpropagation and effectively train the network.

Convolutional neural network

A single region in the image may have many different patterns that want to detect . Consider this region for instance. This region has teeth, some whiskers , and a tongue in that case , to understand this image.

We need filters for detecting all three of these characteristics, one for each teeth , whiskers and tongue

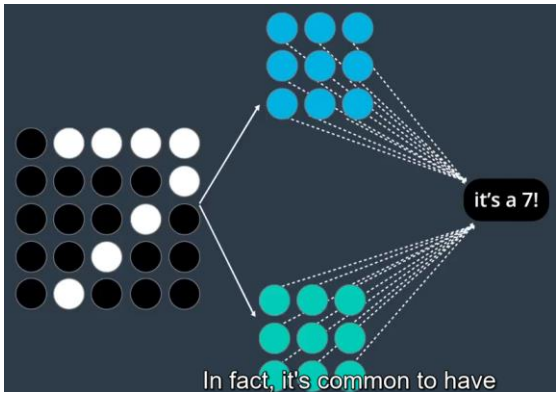
Recall the case of a single convolutional filter. Adding another filter is probably exactly what you`d expect,



Where we just populate an additional collection of nodes in the convolutional layer.

This collection has its own shared set of weights that differ from the blue nodes above them

CNN_FEATURE VISUALIZATION

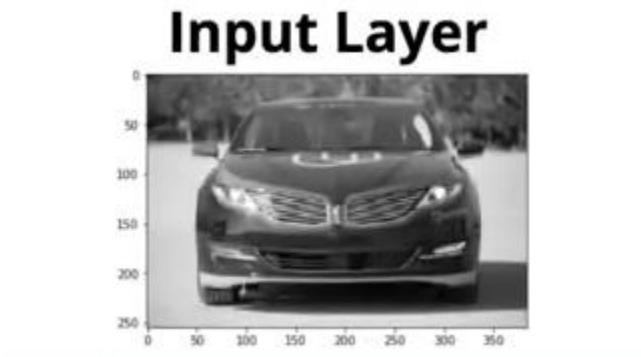


In fact it`s common have tens to hundreds of these collection in convolutional layer , each corresponding to their own filter.

Let`s know execute some code to see what these collections look like After all each is formatted in the same way an in image namely as matrix of values

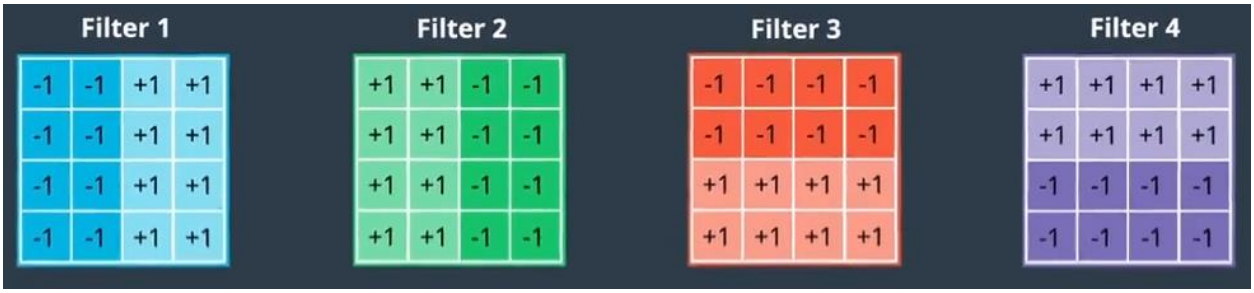
We `ll be visualizing the output of a Jupyter Notebook if you like , you can follow along with the link bellow

So, say we`re working with an image of Udacity`s self-driving car as input



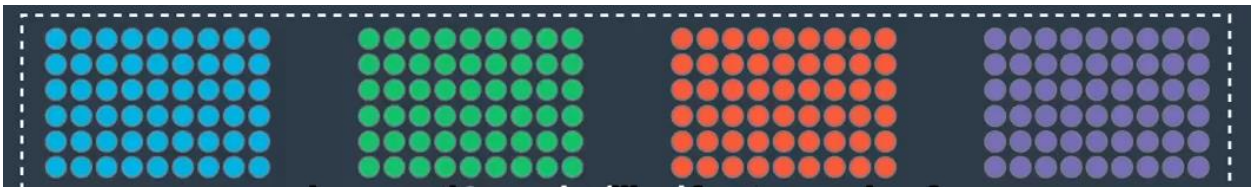
CNN_FEATURE VISUALIZATION

Let`s use four filters , each four pixel high and four pixel wide .



Recall , each filter will be converted across the height and width of the image to produce an entire collection of nodes in the convolutional layer .

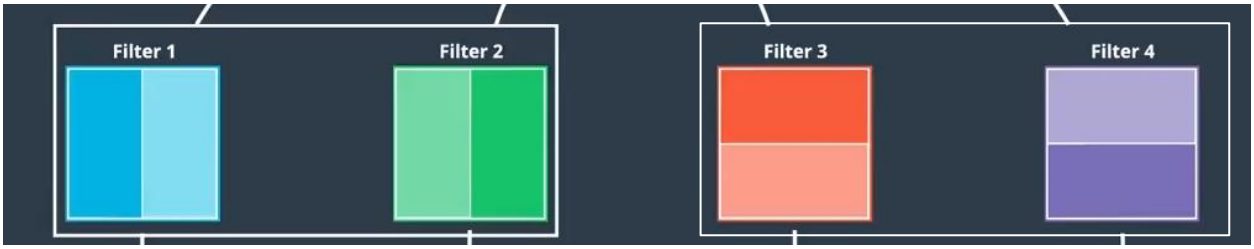
In this case , since we have four filter , we will have four collections of nodes



In practice we will refer to each of these four collections feature map or activation maps when we visualize these feature maps we see that they look like filtered images

That is , we `re taken all of the complicated , dense information in the original image and each four these cases outputted a much simpler image with less information by peeking at the structure of the filters

That you can see that first two filters discover vertical edges and where last two detect horizontal edge in the image



CNN_FEATURE VISUALIZATION

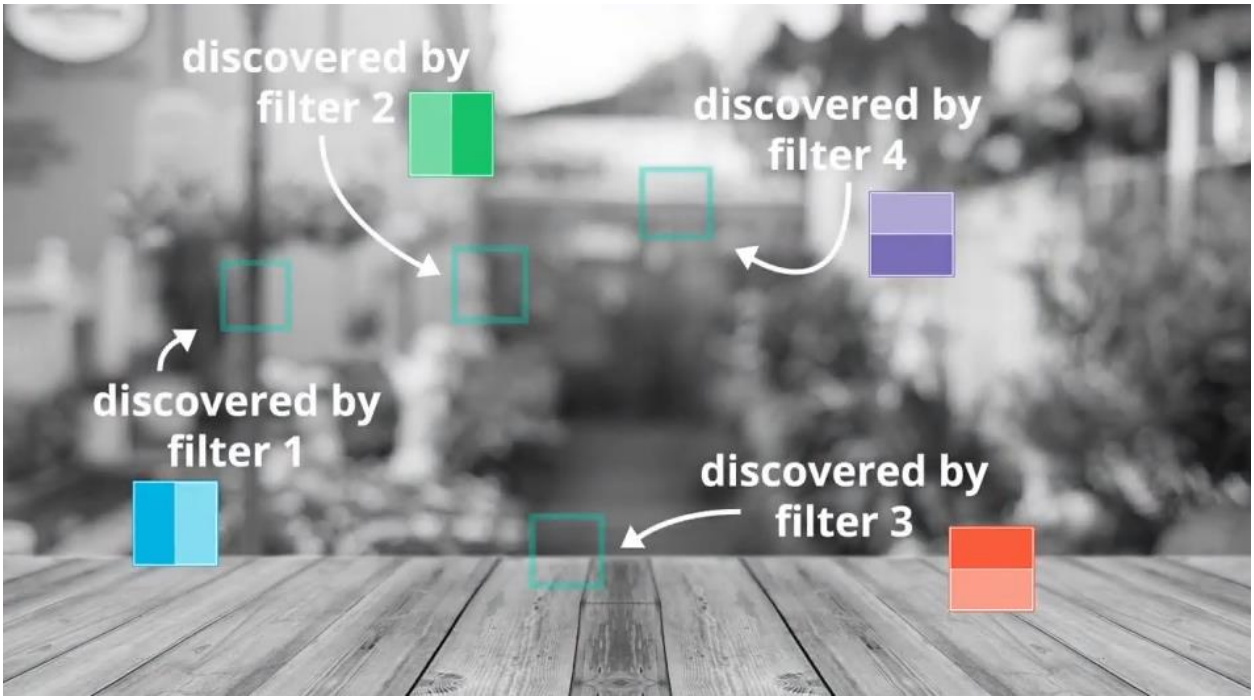
Remember that lighter values in the feature map mean the pattern was detect in the image.

So, you can match the lighter region in each feature map with their corresponding in the original image ?

In this activation map for instance

We can see a clear white line defining the right edge of the car

This is because all of the corresponding regions in the car image closely resemble تشابة the filter where we have a vertical line of dark pixel to the left of vertical line of lighter pixels .



CNN_FEATURE VISUALIZATION

If you think about it you will notice that edges and images appear as a line of lighter pixels next to a line of darker pixel This image for instance contain many region that would be discovered or detected by one of the four filters we defined before.

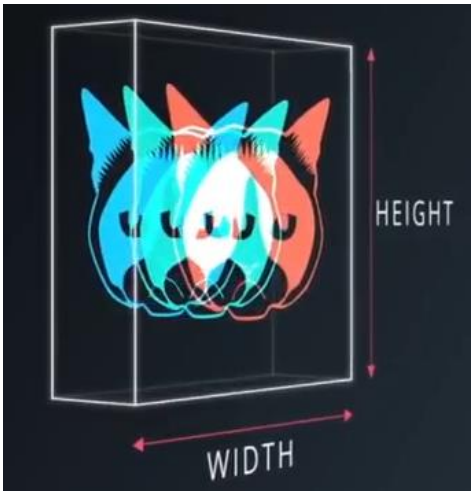
Filters that functions as edge detectors are very important in CNNs , and we will revisit them later.

So , now we know how to understand convolutional layer that have a grayscale image as input but what about a color image !!!!

Well , we have seen that grayscale images are interpreted by the computer as 2D array.

Width , height and width

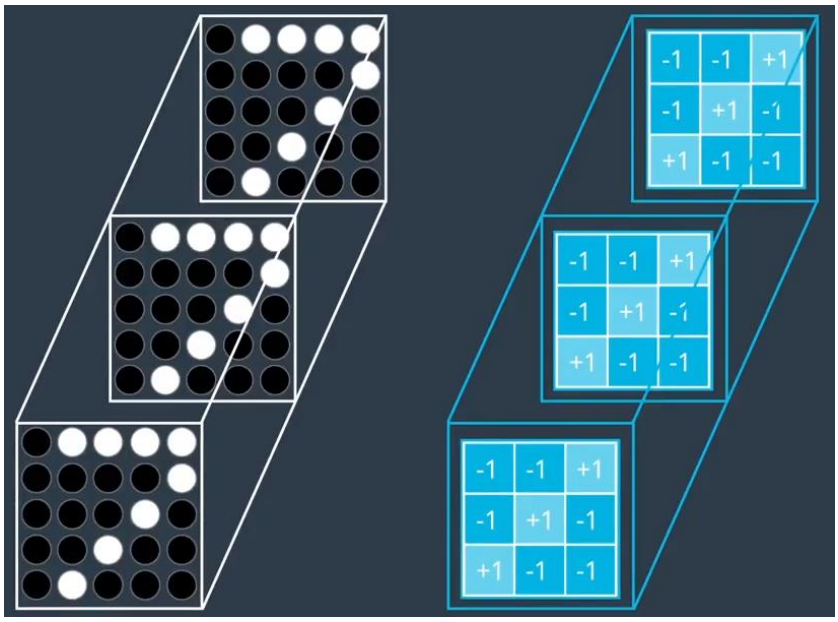
In the color image are interpreted by the computer as a 3D array with height width and depth. In the case of RGB image the depth is three.



This 3D array is best conceptualized as a stack of three two-dimensional matrices. Where we have matrices corresponding to the red , green and blue channel of the images. So how do we perform a convolution on a color image.

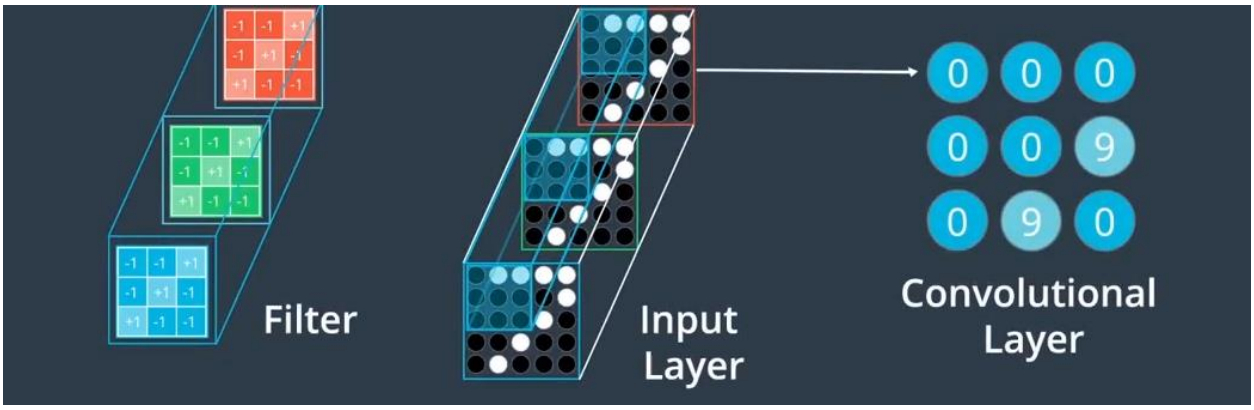
CNN_FEATURE VISUALIZATION

As well as the case with grayscale images we still move filter horizontally and vertically across the image Only now the filter is itself three – dimensional to have a value for each color channel at each horizontal location in the image array.

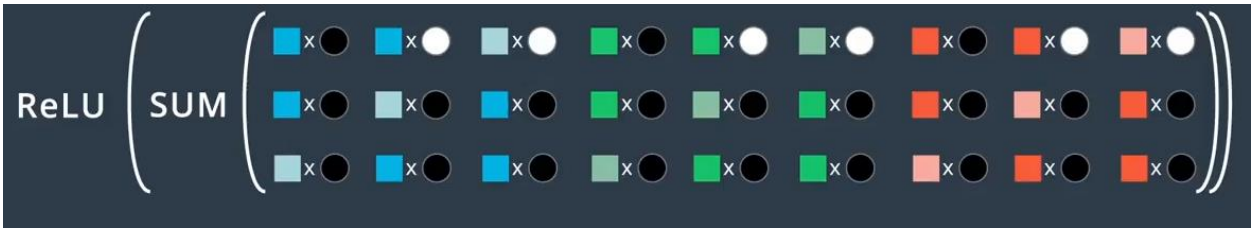


Just as we think of the color image as a stack of three two - dimensional matrices , you can also think of the filter as stack of three , two – dimensional matrices both the color image and the filter have red ,green and blue channels .

Now to obtain the values of the nodes in the feature map corresponding to this filter , we do pretty much the same thing we did before.



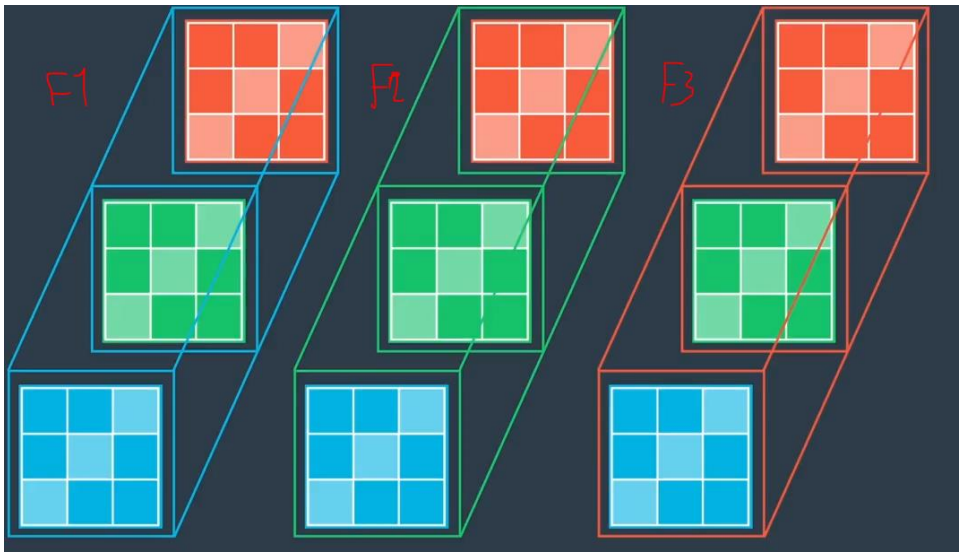
Only now are some as over three times as many terms



We emphasize that here we `ve depicted the calculation of the value of a single node in a convolutional layer , for one filter on a color image

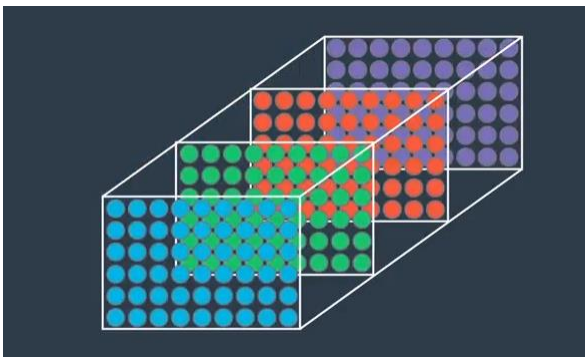
If wanted to picture the case of a color image with multiple filters instead of having a single 3d array with multiple filters instead of having a single 3d array which corresponds to one filter

We would define multiple 3D arrays each defining a filter . Here we have depicted three filters

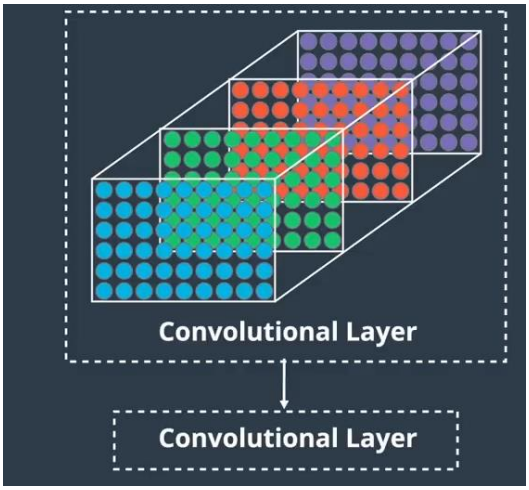


Each is a 3D array that you can think of as a stack of three , 2D arrays .

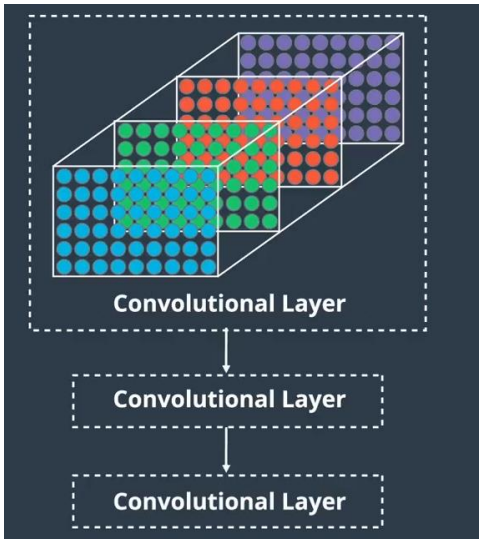
Here `s where it start to get really cool you can think about of each the feature map in convolutional layer along the same line as image channel and stuck them to get a 3D array



We can use this 3D array as input to still another convolutional layer to discover pattern within the patterns that we discovered in the first convolutional Layer.



We can do this again to discover pattern within patterns within patterns.



Remember that in some sense convolutional layers are not to different from the dense layers that you saw in the previous section

Dense layer are fully connected. Meaning that the nodes are connected to every node in the previous layer.

Convolutional layers are locally connected. Where their nodes are connected to only a small subset of the previously S nodes.

Convolutional layers also had this added parameter sharing, but in both cases with convolutional and dense layers, inference works the same way.

Both have weights and biases that are initially randomly generated. So ,

in the case of CNNs Where the weights take the form of convolutional filters , those filter are randomly generated, and so are the patterns that they `re initially designed to detect.

As with ML piece when we construct a CNN we will always specify a loss function in the case of multiclass the will be categorial cross entropy loss.

Then as we train the model through back propagations, the filters are updated at each epic to take on values that minimize the loss function.

In others words, the CNN determines what kind of patterns it need to detect based on the loss functions.

We will visualize these pattern later and see that for instance, if our data set contains dogs the CNN is able to on its own learn filters that look like dogs. So with CNN`s to emphasize we won`t specify the values of the filters or tell the CNN what kind of patterns it need to detect . These will be learned from data.

Defining Layers in PyTorch

Define a Network Architecture

The **various layers** that make up any neural network are documented, here. For a convolutional neural network, we'll use a simple series of layers:

- Convolutional layers
- Max_pooling layers

Fully-connected (linear) layers

To define a neural network in PyTorch, you'll create and name a new neural network class, define the layers of the network in a function `____` and define the feedforward behavior of the network that employs those initialized layers in the function `forward`, which takes in an input image tensor, `x`. The structure of such a class, called `Net` is shown below.

Note: During training, PyTorch will be able to perform backpropagation by keeping track of the network's feedforward behavior and using autograd to calculate the update to the weights in the network.

```
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):

    def __init__(self, n_classes):
        super(Net, self).__init__()

        # 1 input image channel (grayscale), 32 output channels/feature maps
        # 5x5 square convolution kernel
        self.conv1 = nn.Conv2d(1, 32, 5)

        # maxpool layer
        # pool with kernel_size=2, stride=2
        self.pool = nn.MaxPool2d(2, 2)

        # fully-connected layer
        # 32*4 input size to account for the downsampled image size after pooling
        # num_classes outputs (for n_classes of image data)
        self.fc1 = nn.Linear(32*4, n_classes)

    # define the feedforward behavior
    def forward(self, x):
        # one conv/relu + pool layers
        x = self.pool(F.relu(self.conv1(x)))

        # prep for linear layer by flattening the feature maps into feature vectors
        x = x.view(x.size(0), -1)
        # linear layer
        x = F.relu(self.fc1(x))

        # final output
        return x

# instantiate and print your Net
n_classes = 20 # example number of classes
net = Net(n_classes)
print(net)
```

Let's go over the details of what is happening in this code.

Define the Layers in `__init__`

Convolutional and max_pooling layers are defined in `__init__`:

```
# 1 input image channel (for grayscale images), 32 output channels/feature maps, 3x3 square convolution kernel
self.conv1 = nn.Conv2d(1, 32, 3)

# maxpool that uses a square window of kernel_size=2, stride=2
self.pool = nn.MaxPool2d(2, 2)
```

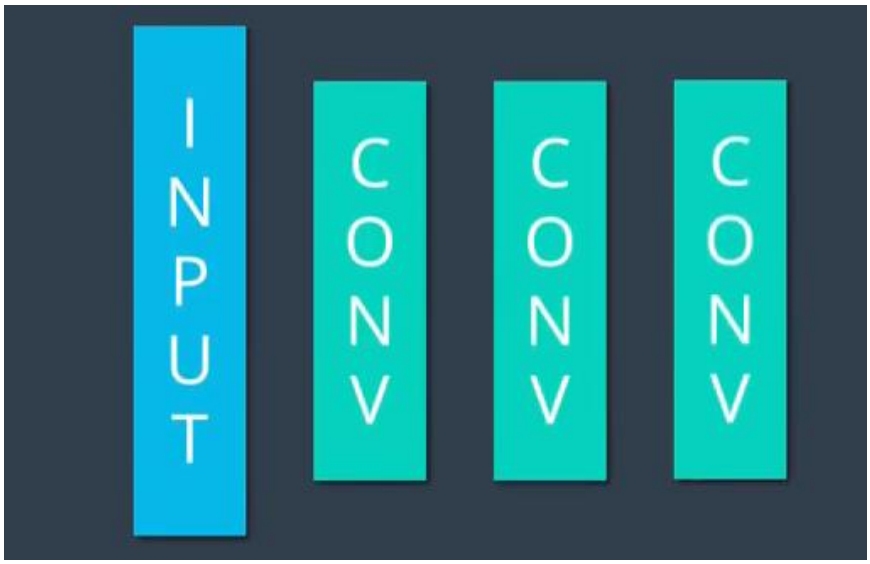
Refer to Layers in `forward`

Then these layers are referred to in the `forward` function like this, in which the conv1 layer has a ReLu activation applied to it before max_pooling is applied:

```
x = self.pool(F.relu(self.conv1(x)))
```

Best practice is to place any layers whose weights will change during the training process in `__init__` and refer to them in the `forward` function; any layers or functions that always behave in the same way, such as a pre-defined activation function, may appear in the `__init__` or in the `forward` function; it is mostly a matter of style and readability.

Pooling Layers



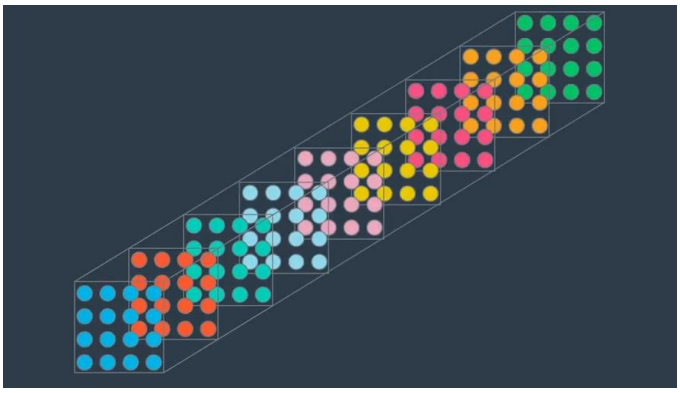
Final type of layer : that we will need to introduce before building our own convolutional neural network.

These so-called pooling layer (sampled_layer) often take convolutional layer as input .

CNN_FEATURE VISUALIZATION

Recall that a convolutional layer is a stack of feature-map when we have one feature map for each filter.

A Complicated dataset with many different object categories will require a large number of filters,



Each responsible for finding a pattern in image. More filters are means a bigger stack , which means that the dimensionally of our convolutional layers can get quite large.

Higher dimensionally means we will need to use more parameters , which can lead to overfitting .

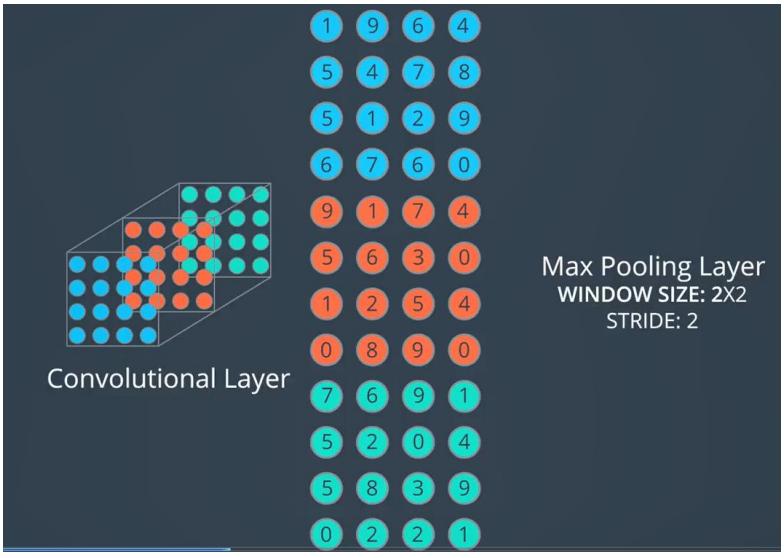
Thus we need a method to reducing the dimensionally.

This is rule of pooling layers within a convolutional neural network.

We will focus on two different types of pooling layers .

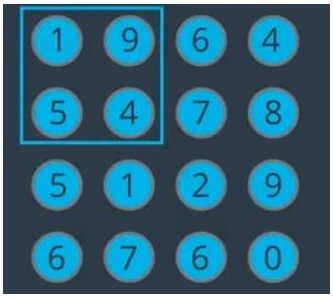
1- The first type is the Max pooling Layer (will take a stack of feature maps as input.) Here we have enlarged and visualizing all three of feature map . As with convolutional layers , we will define a windows size and stride **in this case**
   a. Window size is 2
   b. Stride is 2

   To Construct the max pooling layer, we will work with each feature mapped separately

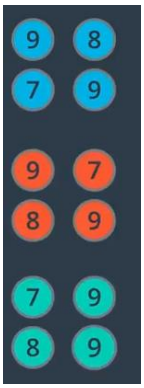


CNN_FEATURE VISUALIZATION

Let`s begin with the first feature map. We start with our window in the top left corner of the image.



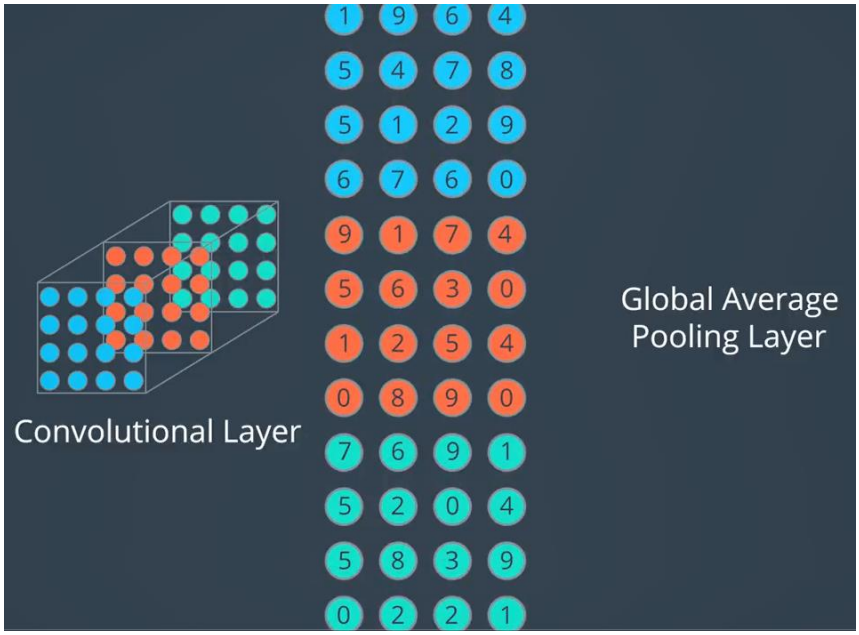
The value of corresponding node in the max pooling layer is calculated by just taking the maximum of the pixels contained in the window in this case 1 , 9 , 5 , 4 in our window so 9 was the maximum if we continue this process and do it for all our feature maps.



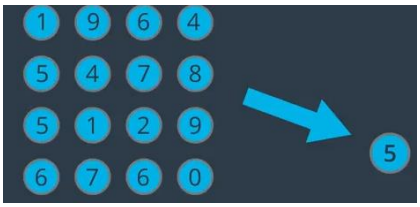
The output is a stack with the same number of feature maps but each feature map has been reduced in width and height .

In this case the width and height are half size of the previous convolutional layer.

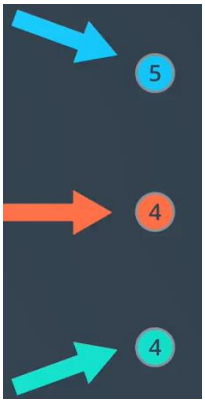
Global average pooling is a bit different. For a layer of this type we specify neither window size nor stride. This type of pooling is a more extreme type of dimensionally reduction. It takes a stack of feature map and computes he average value of nodes for each map in the stack As before , we will work with each feature map.



As before, we will work with each feature map separately beginning with the first feature map to get the average value of the nodes we first sum up all node which yield 80 then we divide by the total number of nodes, which is 16.



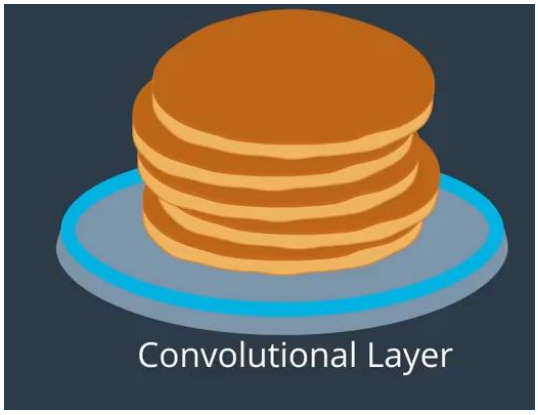
Repeating this process for the remaining two feature maps , we get final output is a stack of feature maps where each feature map has been reduced to a single value.



In this way we see that a global average pooling layer takes a 3d array and turn it into vector.

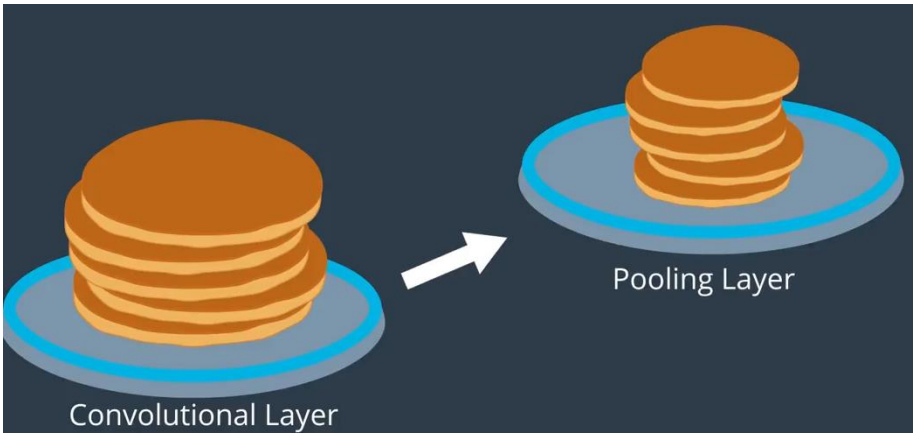
Here we have a vector with three entries.

Let`s summarize what we have learned with the food analogy we will think of a convolutional layer as a stack of pancakes with one pancakes for each feature map.

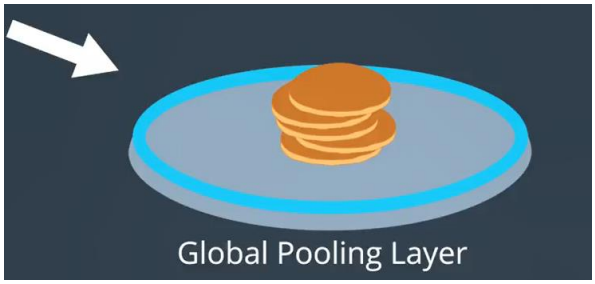


Pooling layer take that stack and give us back a stack with the same number of feature map (pancakes) except the output pancakes are smaller in width and height.

Non-global pooling layers represent a moderate reduction in pancakes size. Where each pancakes a generally about a half as tall and half as wide as its corresponding input pancake.

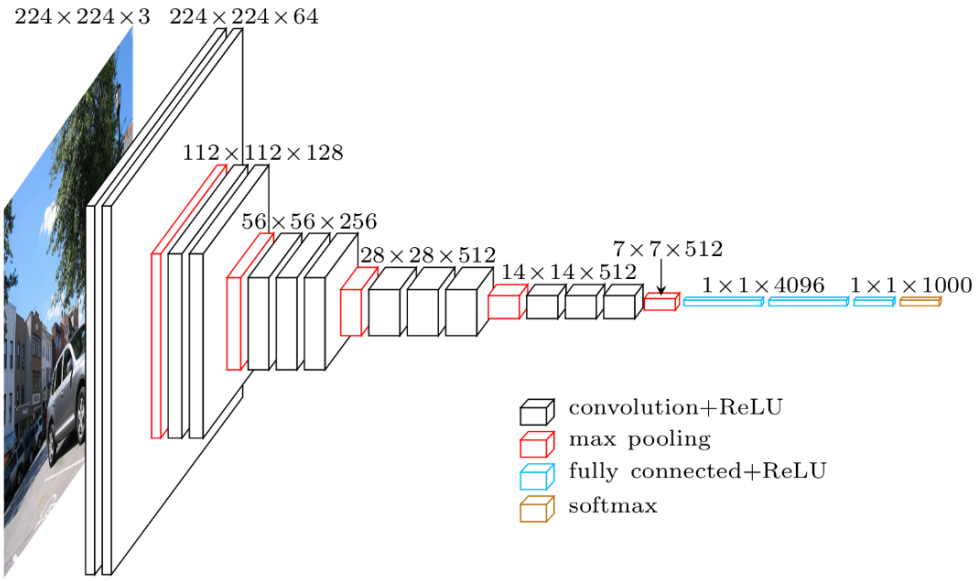


The global pooling layers reduce each input pancakes to essentially an output crumb. But, we still have one crumb for each input pancake.



Free documentation: https://pytorch.org/docs/stable/nn.html#pooling-layers

Fully-Connected Layers, VGG-16



Fully-Connected Layer

A fully-connected layer's job is to connect the **input** it sees to a desired form of output. Typically, this means converting a matrix of image features into a feature vector whose dimensions are 1xC, where C is the number of classes. As an example, say we are sorting images into ten classes, you could give a fully-connected layer a set of [**pooled**, **activated**] feature maps as input and tell it to use a combination of these features (multiplying them, adding them, combining them, etc.) to output a 10-item long feature vector. This vector compresses the information from the feature maps into a single feature vector.

Softmax

The *very* last layer you see in this network is a **softmax function**. The softmax function, can take any vector of values as input and returns a **vector of the same length whose values are all in the range (0, 1)** and, together, these values will add up to 1. This function is often seen in classification models that have to turn a feature vector into a **probability distribution**.

Consider the same example again; a network that groups images into one of 10 classes. The fully-connected layer can turn feature maps into a single feature vector that has dimensions 1x10. Then the softmax function turns that vector into a 10-item long probability distribution in which each number in the resulting vector represents the probability that a given input image falls in **class 1**, **class 2**, **class 3**, … class 10. This output is sometimes called the **class scores** and from these scores, you can extract the most likely class for the given image!

Overfitting

Convolutional, **pooling**, and **fully-connected** layers are all you need to construct a complete CNN, but there are additional layers that you can add to avoid overfitting, too. One of the most common layers to add to prevent overfitting is a dropout layer.

Dropout layers essentially turn off certain nodes in a layer with some probability, **p**. This ensures that **all nodes get an equal chance to try and classify different images during training,** and it reduces the likelihood that only a few, heavily-weighted nodes **will dominate the process**. تسيطر على prco

Now, you're familiar with all the major components of a **complete convolutional neural network**, and given some examples of **PyTorch** code, you should be well equipped to build and train your own CNN's! Next, it'll be up to you to define and train a CNN for **clothing recognition!**

Training in PyTorch

Once you've loaded a **training dataset**, next your job will be to define a CNN and train it to classify that set of images.

Loss and Optimizer

To train a model, you'll need to define *how* it trains by -- **selecting a loss function** and optimizer. These functions decide how the model updates its parameters as it trains and can affect how quickly the model converges, as well.

Learn more about loss functions and optimizers in the online documentation.

For a classification problem like this, one typically uses cross entropy loss, which can be defined in code like: `criterion = nn.CrossEntropyLoss()`.PyTorch also includes some standard stochastic optimizers like stochastic gradient descent and Adam. You're encouraged to try different optimizers and see how your model responds to these choices as it trains.

Classification vs. Regression

The loss function you should choose depends on the kind of CNN you are trying to create; cross entropy is generally good for classification tasks, but you might choose a different loss function for, say, a regression problem that tried to predict (x,y) locations for the center or edges of clothing items instead of class scores.

Training the Network

Typically, we train any network for a number of epochs or cycles through the training dataset

Here are the steps that a training function performs as it iterates over the training dataset:

1. Prepares all input images and label data for training
2. Passes the input through the network (forward pass)
3. Computes the loss (how far is the predicted classes are from the correct labels)
4. Propagates gradients back into the network's parameters (backward pass)
5. Updates the weights (parameter update)

It repeats this process until the average loss has sufficiently decreased.

And in the next notebook, you'll see how to train and test a CNN for clothing classification, in detail.

Please also checkout the linked, exercise repo for multiple solutions to the following training challenge!

https://github.com/udacity/CVND_Exercises/tree/master/1_5_CNN_Layers

Dropout

Dropout and Momentum

The next solution will show a different (improved) model for clothing classification. It has two main differences when compared to the first solution:

1. It has an additional dropout layer
2. It includes a momentum term in the optimizer: stochastic gradient descent

So, why are these improvements?

Dropout

Dropout randomly turns off perceptrons (nodes) that make up the layers of our network, with some specified probability. It may seem counterintuitive to throw away a connection in our network, but as a network trains, some nodes can dominate others or end up making large mistakes, and dropout gives us a way to balance our network so that every node works equally towards the same goal, and if one makes a mistake, it won't dominate the behavior of our model. You can think of dropout as a technique that makes a network resilient; it makes all the nodes work well as a team by making sure no node is too weak or too strong. In fact it makes me think of the Chaos Monkey tool that is used to test for system/website failures.

I encourage you to look at the PyTorch dropout documentation, here, to see how to add these layers to a network.

Momentum

When you train a network, you specify an optimizer that aims to reduce the errors that your network makes during training. The errors that it makes should generally reduce over time but there may be some bumps along the way. Gradient descent optimization relies on finding a local minimum for an error, but it has trouble finding the *global minimum* which is the lowest an error can get. So, we add a momentum term to help us find and then move on from local minimums and find the global minimum!

Dropout

Let`s say this is you. And one day you decide to practice sports.

1- Monday you play tennis
2- Tuesday play lift weights
3- Wednesday you play American Football
4- Thursday you play baseball
5- Friday you play basketball
6- Saturday you play ping pong

Noticed that you've done most of them with your dominant hand اليد المسيطرة

So , you 're developing a large muscle on that arm but not on the other arm. This is disappointing

S, what can you do ? Well let`s spice it up on the next week. What we will do is on Monday we will tie our tight hand behind our back and try to play tennis with left hand.

Our Tuesday we will , tie our left hand. Behind you back and try to left weights with right hand.

Then on Wednesday again , we will tie our right hand and play American football with the left one.

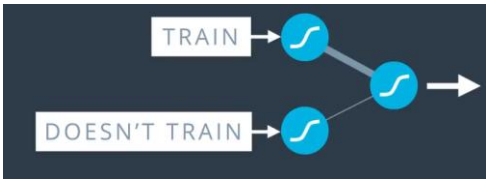
On Thursday we will take it easy and play baseball with both hands, that fine.

Then , on Friday we will tie both hands behind our back and try to play basketball. **That won`t work out too well. But it is ok . it`s training process.**

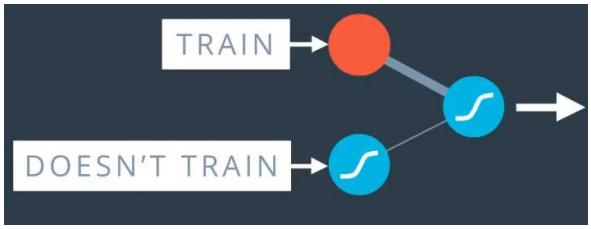
And Saturday again, we tie our left hand behind our back and play ping pong with the right.

After week ,we see that we have developed both our biceps. Pretty good job.

This is something that happen a lot when we train neural network. Sometimes one part of the network has very large weights and it ends up dominating all the training, while another part of the network doesn`t really play much of a role so it does not get trained. So, what we will do to solve this is something during training,



We will turn this part off and let the rest of the network train.



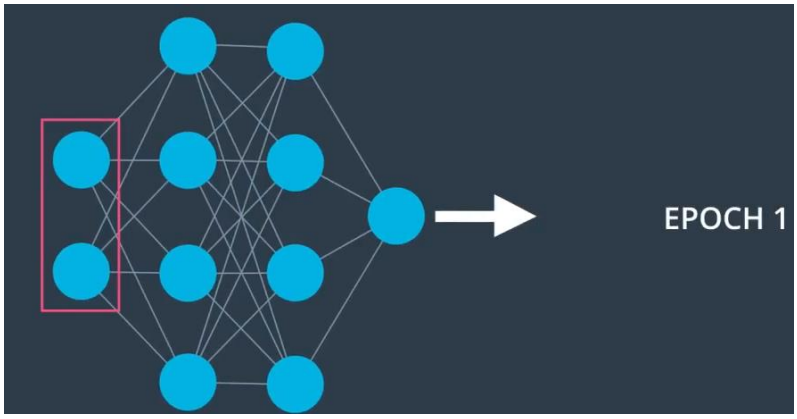
More thoroughly بعناية, what we do is as we go through the epochs فترة .

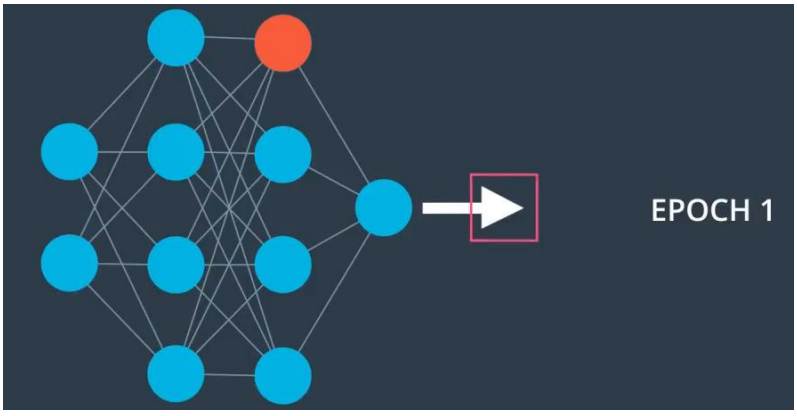
We randomly turn off some of the nodes and say, you shall not pass through here.

In that case , the other nodes have to pick up the slack and take more part in the training.

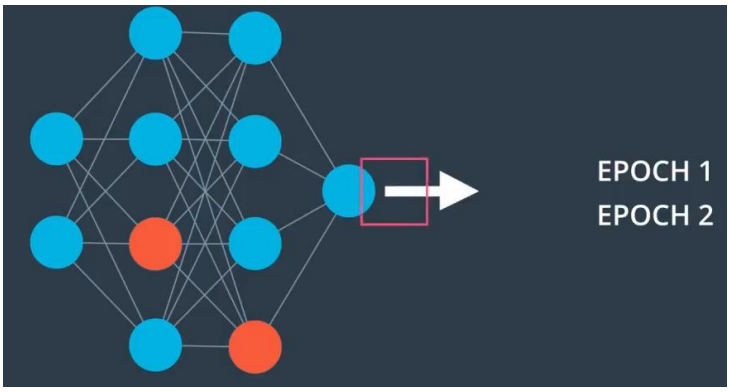
So for example,

1-in the first epoch we `re not allowed to use this node. So we do feat forward and our back propagation passes without using it

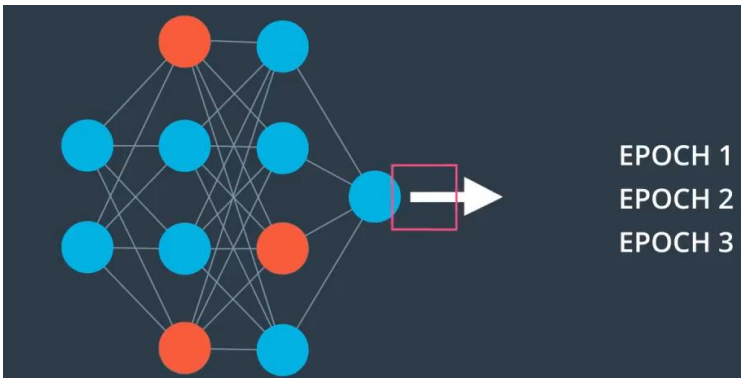




2- In the second epoch 2 we can`t use these two nodes again , we do our feet forward and back prop.



3- And the third epoch we can`t use these nodes over here. So again , we do forward and back prop.



4- And Finally, in last epoch , we can`t use these nodes over here. So we continue like that.

What we will do to drop the nodes is we will give the algorithm a parameter Default p = 0.5

This parameter is probability that each nodes gets dropped at a particular epoch.

For example , if we give it at 0.2 it means each epochs, each node gets turned off with a probability of 20 percent.

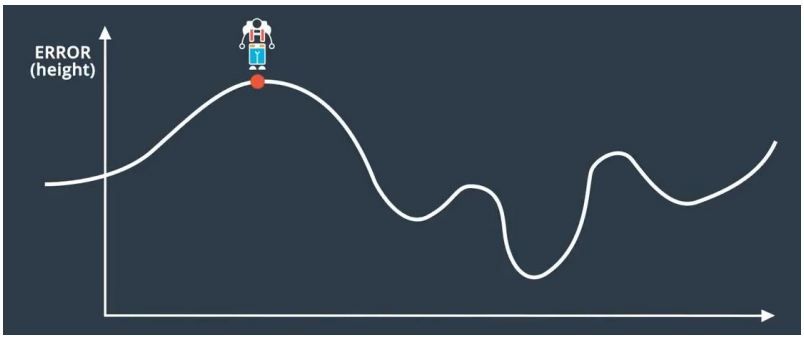
Notice that some nodes may get turned off **more than** others and some others may never get turned off. And this is ok since we are doing it over and over and over on average each node will get the same treatment. This method is called dropout and it`s really common and useful to train neural network.

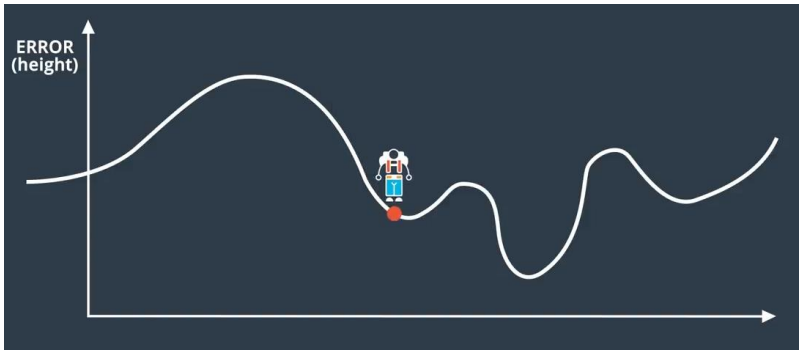
Gradient Descent

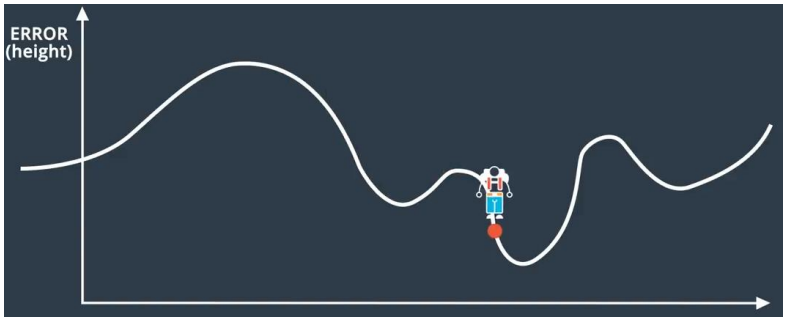
Solve local minimum problem

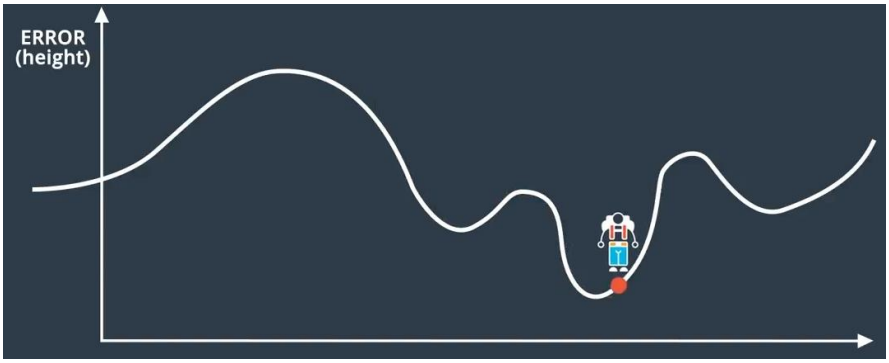
The idea is to walk a bit fast with momentum and determination in a way that if you get stuck in a local minimum

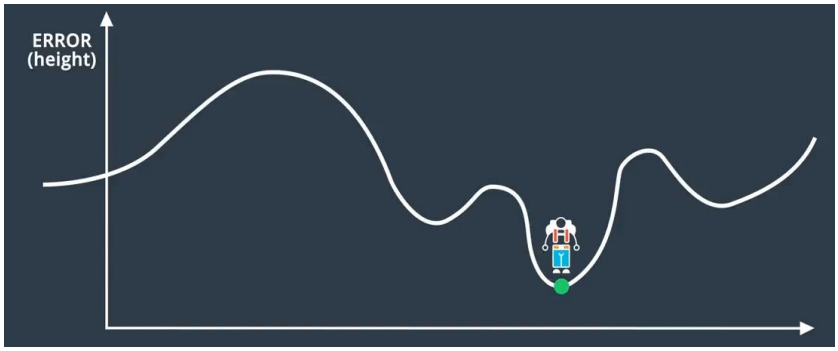
Continue









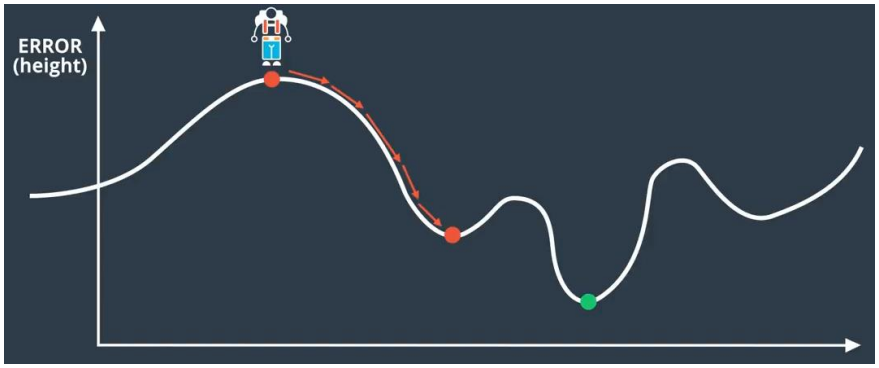


Local minimum

You can sort of power through and get over the hump to look for a lower minimum.

So let`s look at what normal gradient descent does.

It gets us all the way here No problem.

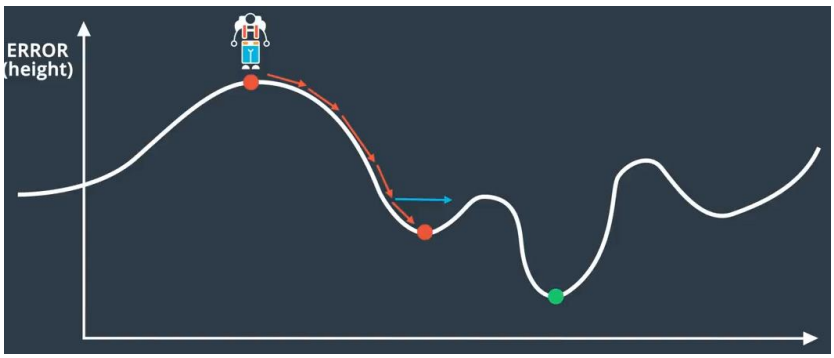


Now, we want to go over the hump but by now the gradient is zero or too small, So it won`t give us a good step

What if we look at the previous one?

What about say the average of the last few steps.

If we take the average, the will takes us in directions and push us a bit towards the hump.



Now the average seems a bit drastic جزرية بعض الشئ since the step we made 10 steps ago is much less relevant than the step we last made. So , we can say, for example,

1- The average of the last **three** or **four** steps Even better , we can weight each step so that the previous step matters a lot and the steps before that matter less and less.

IDEA: MOMENTUM

STEP ⟶ AVERAGE OF PREVIOUS STEPS

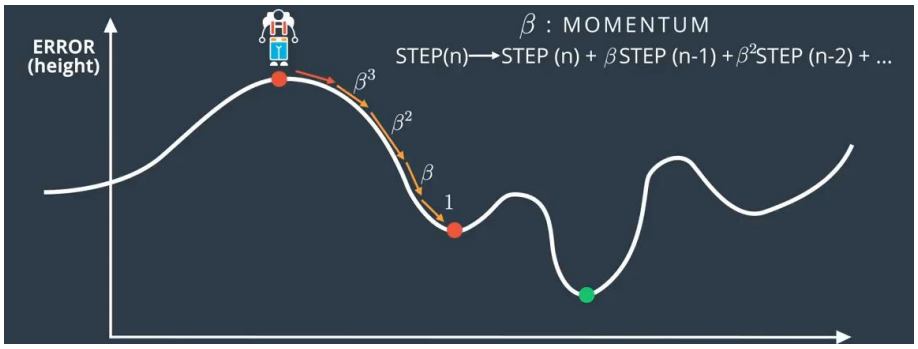
Here is where we introduce momentum.

β : MOMENTUM

Momentum it a constant beta between 0 and 1 that attaches to the steps as follows: the previous step gets multiplied by 1

$$STEP(n) \longrightarrow STEP(n) + \beta STEP(n-1) + \beta^2 STEP(n-2) + \dots$$

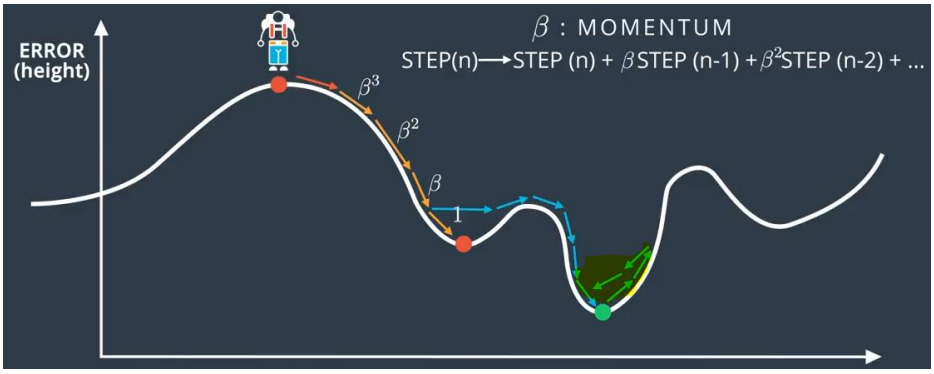
The one before , by data , the one before ,by beta squared, the one before ,by beta cubed by etc.



In this way, the steps that happened a lone time ago will matter less than the ones that happened recently.

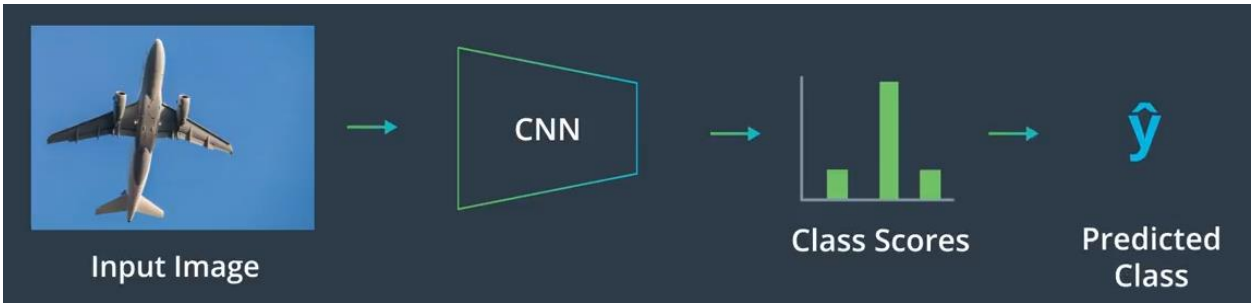
We can see that that gets us over the hump. But now ,once we get to the global minimum, it will still be pushing us away a bit but not as much. This may seen vague مشكلة

But the algorithms that use momentum seem to work really well in practice.



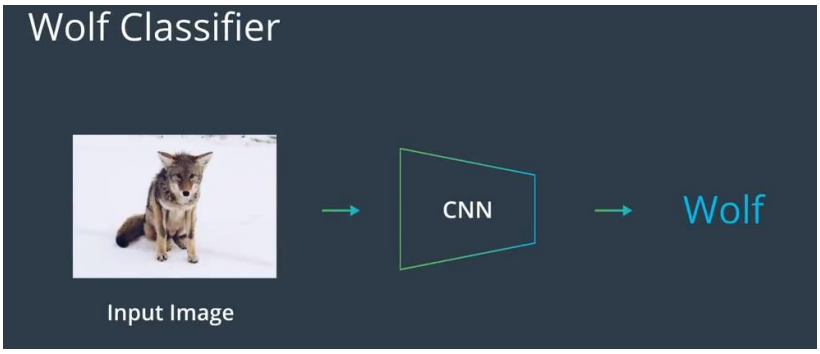
Feature visualization

Let`s look at the architecture a classification CNN that takes in an image and output a predicated class for the image class for the image.



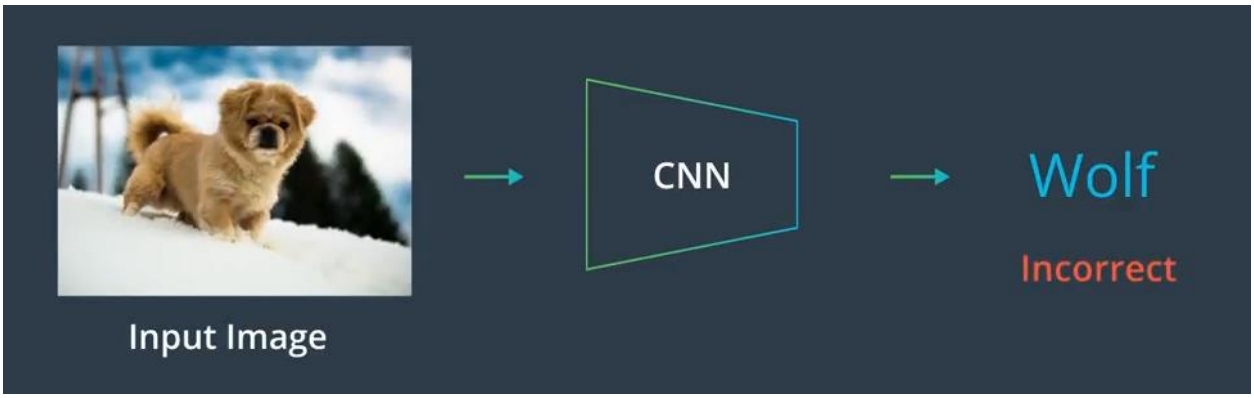
You have seen to train CNN`s like this on sets of labeled image data, and you have learned about the layers that make up a CNN.

Thing like Convolutional Layers that learned to filter images and pooling layers that reduced the size of your input. We know in theory what`s happening in these layer but after a network has been trained and especially in networks with many layers , how can we know exactly what each layer has learned and why is this important ?

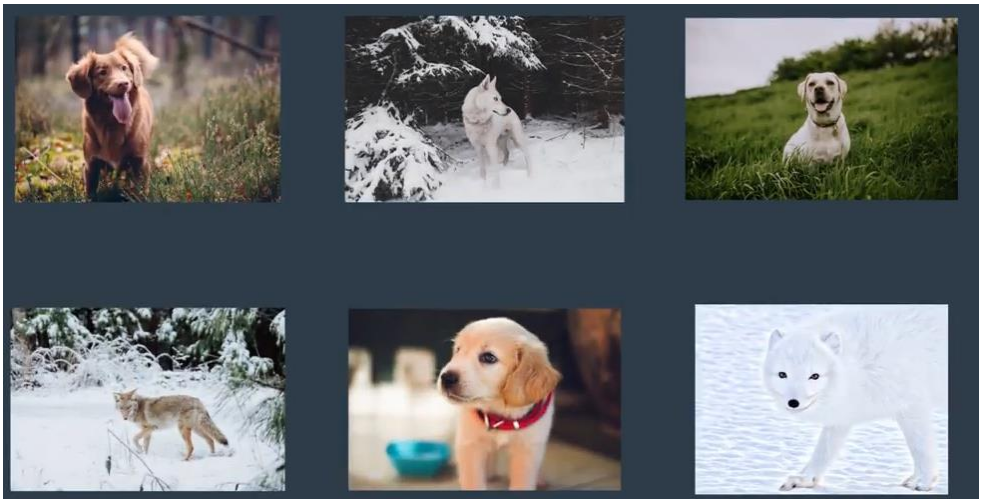


Take this classifier as an example. Say it`s only been training for a little while and it`s learning to distinguish images of doges from images of wolves but it is making some mistake.

Mistake



Here are a few image from the training set of comparison



Well can you see that distinguishes these images ? But what does network see ?

It turns out that since this model was training on a set of images, in which wolves were primarily pictured on a snowy landscape.

it actually learned that snow was an important feature to extract.

If there was snow, then an image was most likely classified as a wolf.

Even if it was really an image of dog.

So because of the limited training data, the **model learning** to identify snow, rather than really identifying features of a dog and a wolf and if we had some insight into what features as model was extracting we could actually visualize this shortcoming.

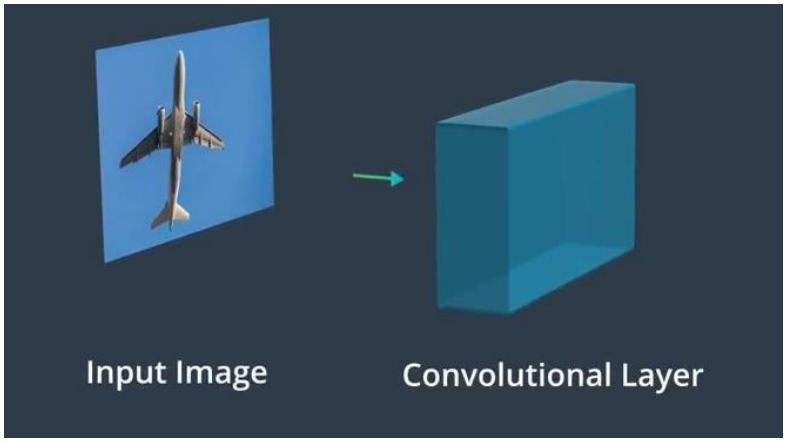
So how can we see kind of **features** and **network** has learned to recognize ?

This area of study is called feature visualization and it`s all about techniques that allow you at each layer of the model.

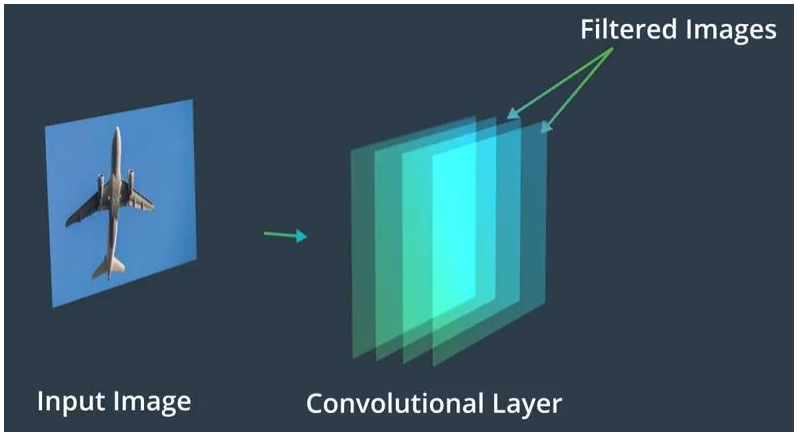
What kind of **images features** and **network** has learned to extract ?

Feature map

We have talked about how the first convolutional layer in a CNN consists of a number of filtered images



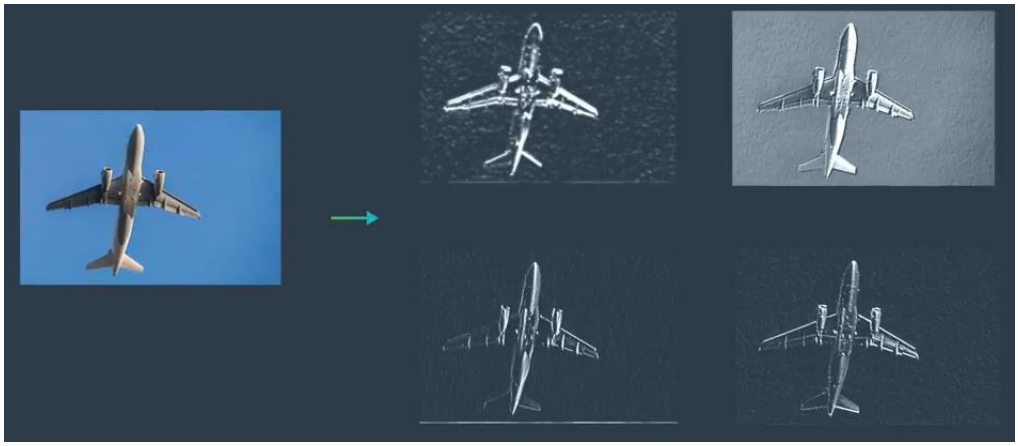
That are produced as the input image is convolved with a set of image filters .



CNN_FEATURE VISUALIZATION

This filter are just grids of weights.

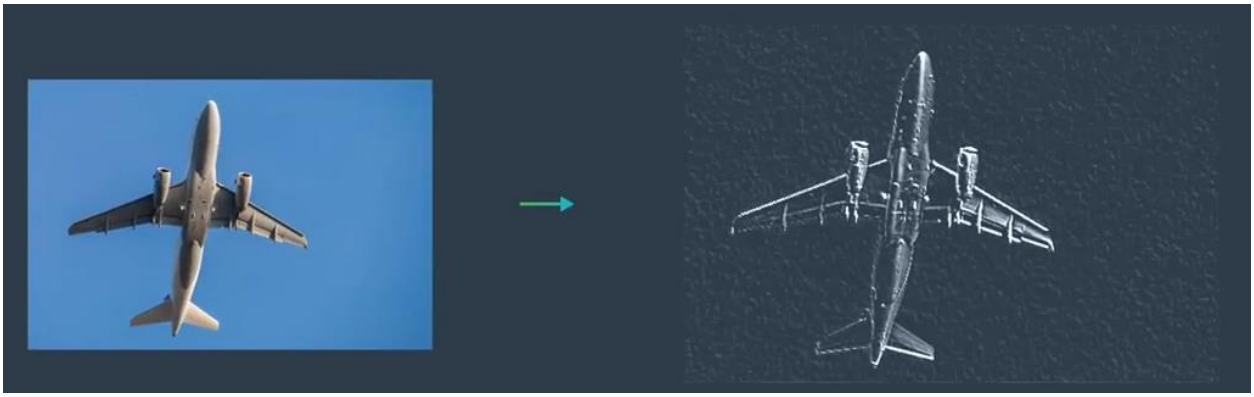
For convolutional layer with four filters, four filtered outputs will be produced.



Each of these filtered output images is also called feature map or activation map because each filtered image should extract certain features from the original image and ignore other information.

For a given image, each of these maps will activate in some way, displaying activated bright pixel or not in each map.

A good example of a filter is a high-pass filter,



Which when applied to an input image creates an activation map the most when it sees high frequency features.

You have programmed your own **high-pass filter**. One interesting thing about most CNN`s is that as the network trains , and the weights that make up each convolutional kernel are updated.

1- The **first convolutional layer** often learns to create high pass filter just like these.

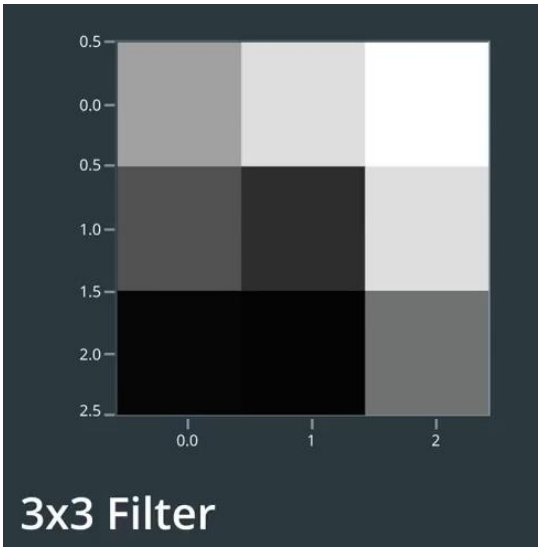
First Convolutional Layer

The first convolutional layer in CNN applied a set of image filters to an input and output a stack of feature maps**. After such a network is trained we can get a sense of what kinds of features this first layer has learned to extract by visualizing the learned weights for each of these filters.**

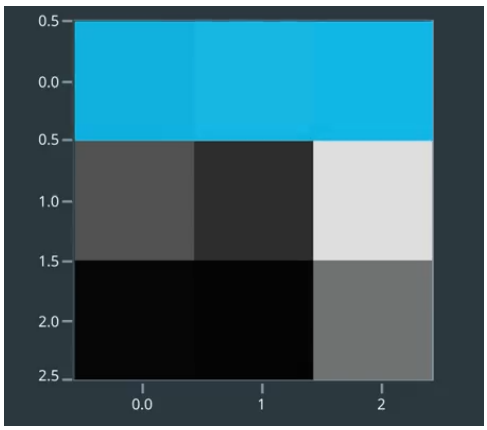
we know that filters are girds of **weights values** and so we can visualize them by treating them a small image.

By 3x3 filter for a grayscale image can be pictured a three by three grayscale pixel image, where each pixel contains a learned weights value.

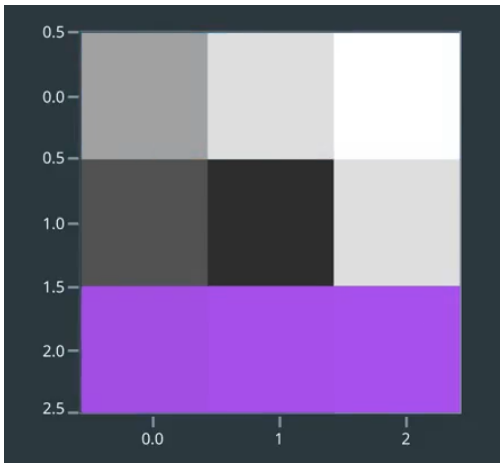
1- **The bright pixel can be thought** of as positive or high values for weights.
2- **The darker pixels correspond** to small or negative weights.



So, the kernel that looks like this with **bright values** on the top,



and darker values on the bottom.

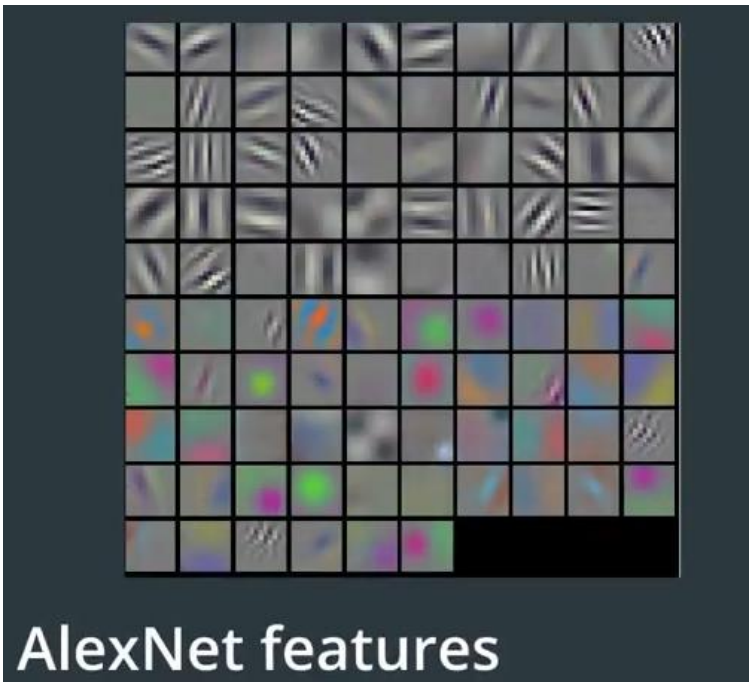


We might think that it subtracts bottom pixels from top pixels in a patch and so , it can detect horizontal edges in an image.

For CNN that analyzes color image, the filters just have another dimension for color, and can be displayed as small color images.

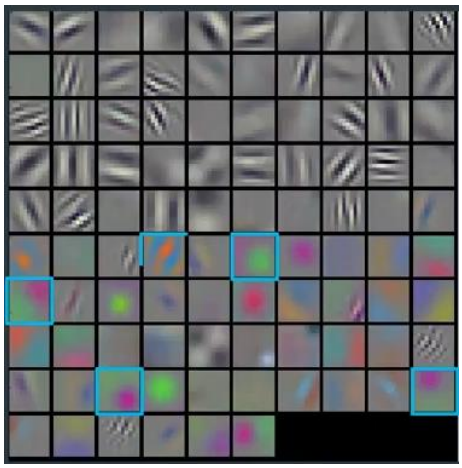
CNN_FEATURE VISUALIZATION

Here are some examples of filters from the first convolutional layer of a CNN with 11 by 11 color filters.



You can see that a lot of these filters are looking for specific kinds of geometric patterns, vertical or horizontal edges , which are represented by alternating bars of light and dark at different angles.

You can also see the corner detector and filters that appears to detect area of opposing colors with magenta and green bunches or blue and orange lines.



Now you might be thinking great, this is really simple I can just look at the filter weights of trans CNN to see what my what network is learning.

But this becomes tricky as soon as you move further into the network and try to look at the second or even third convolutional layer.

This is because these later layers are no longer directly connected to the input image.

They are connected to some output that has likely been pooled and processed by an activation function

So, visualizing the filters in these deeper layers doesn`t clearly tell us what these layers are actually seen in the original input image.

So will need to develop a different techniques to really see what`s going on in these intermediate layers.

Visualizing activations

for intermediate layer like the second convolutional layer in CNN, visualizing the learned weights in each filter doesn`t give us easy to read information.

So, how can we visualize what these deeper layers are seeing ?

Well, what can give us useful information is to look at the feature maps of these layers as the network looks at specific images.

This is called layer activation, and it mean looking at how a certain layer of feature maps activates when it see a specific input image such as image of a face.

The filter is deeper convolutional layers will often show high level or bright spots of activations in localized areas.

The important thing is to see that these maps are not just producing blobby , noisy outputs

They should produce a noticeably different response for different classes of images.

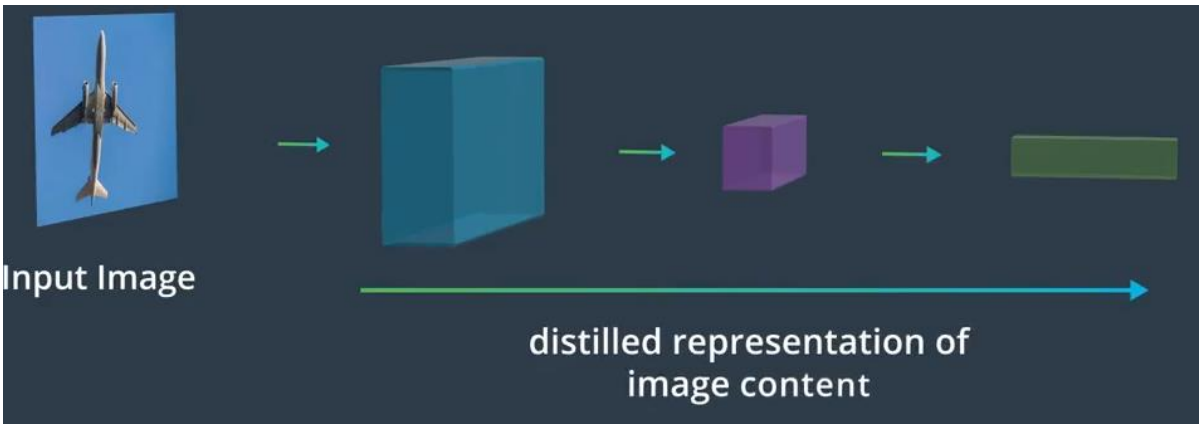
You have already seen a few examples of activations like these but next let`s a look at how we can extract activations maps from a trained CNN,

Summary of feature

Several approaches for understand and visualizing the convolutional networks have been developed in the literature. Partly, as a response to the common criticism that the learned features in a a neural network are not interpretable ليست قابلة للتفسير

The most commonly used feature visualization techniques from looking at filter weights to looking at layer activations.

In addition to giving you insight into what features a CNN has learned to extract from an image these techniques should give you a greater understanding of how CNNs work.



For classifications CNN, you can actually see as you look at deeper layer in a model that the CNN transforms am input image into a smaller more distilled representation of the content of that image.

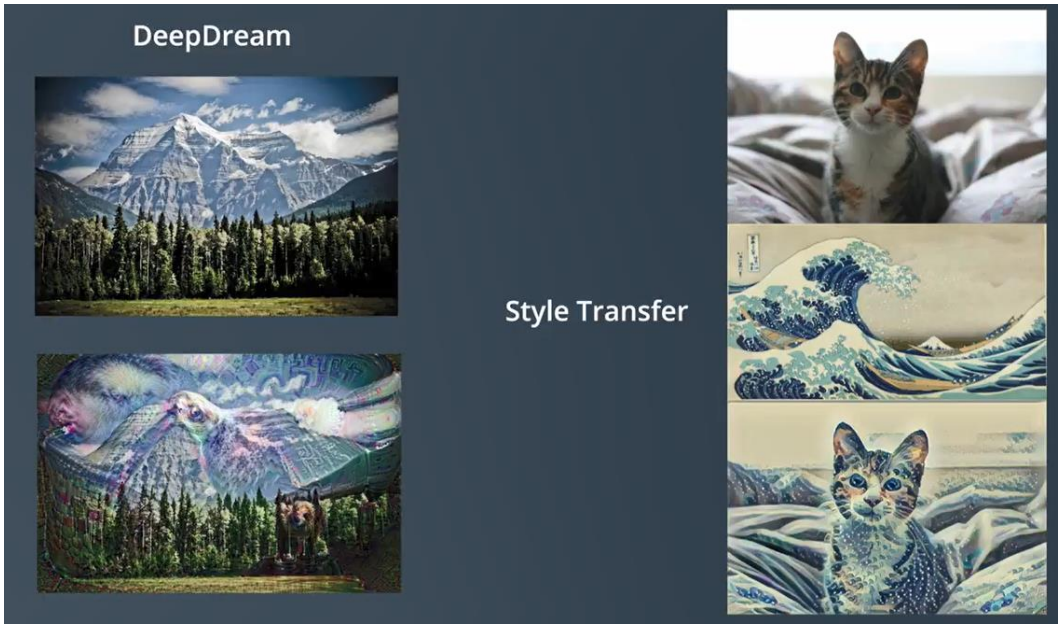
By the last layer of a CNN, it has learned to extract high level features that still contain enough information about an image to classify it .

These technique are actually the basis for applications like

1- **style Transfer**
2- **DeepDream**

That compose images based on layer activations and extracted features

Perhaps most importantly feature visualization gives you a way to show and communications to another to other people what your networks have learned.

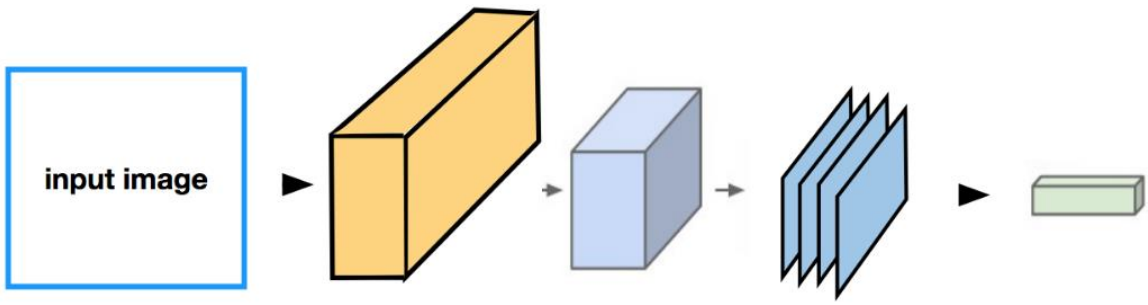


Last Feature Vector and t-SNE

Last Layer

In addition to looking at the first layer(s) of a CNN, we can take the opposite approach, and look at the last linear layer in a model.

We know that the output of a classification CNN, is a fully-connected class score layer, and one layer before that is a **feature vector that represents the content of the input image in some way**. This feature vector is produced after an input image has gone through all the layers in the CNN, and it contains enough distinguishing information to classify the image.



CNN_FEATURE VISUALIZATION

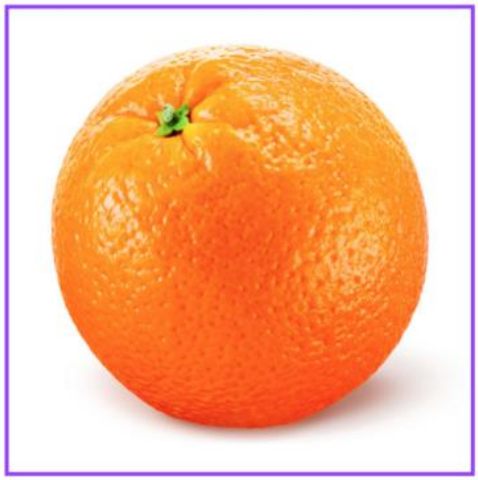
An input image going through some conv/pool layers and reaching a fully-connected layer. In between the feature maps and this fully-connected layer is a flattening step that creates a feature vector from the feature maps.

** Final Feature Vector**

So, how can we understand what's going on in this final feature vector? What kind of information has it distilled from an image?

To visualize what a vector represents about an image, we can compare it to other feature vectors, produced by the same CNN as it sees different input images. We can run a bunch of different images through a CNN and record the last feature vector for each image. This creates a feature space, where we can compare how similar these vectors are to one another.

We can measure vector-closeness by looking at the **nearest neighbors** in feature space. Nearest neighbors for an image is just an image that is near to it; that matches its pixels values as closely as possible. So, an image of an orange basketball will closely match other orange basketballs or even other orange, round shapes like an orange fruit, as seen below.



A basketball (left) and an orange (right) that are nearest neighbors in pixel space; these images have very similar colors and round shapes in the same x-y area.

Nearest neighbors in feature space

In feature space, the nearest neighbors for a given feature vector are the vectors that most closely match that one; we typically compare these with a metric like MSE or L1 distance. And *these* images may or may not have similar pixels, which the nearest-neighbor pixel images do; instead they have very similar content, which the feature vector has distilled.

In short, to visualize the last layer in a CNN, we ask: which feature vectors are closest to one another and which images do those correspond to?

And you can see an example of nearest neighbors in feature space, below; an image of a basketball that matches with other images of basketballs despite being a different color.

Dimensionality reduction

Another method for visualizing this last layer in a CNN is to reduce the dimensionality of the final feature vector so that we can display it in 2D or 3D space.

For example, say we have a CNN that produces a 256-dimension vector (a list of 256 values). In this case, our task would be to reduce this 256-dimension vector into 2 dimensions that can then be plotted on an x-y axis. There are a few techniques that have been developed for compressing data like this.

Principal Component Analysis

One is PCA, principal component analysis, which takes a high dimensional vector and compresses it down to two dimensions. It does this by looking at the feature space and creating two variables (x, y) that are functions of these features; these two variables want to be as different as possible, which means that the produced x and y end up separating the original feature data distribution by as large a margin as possible.

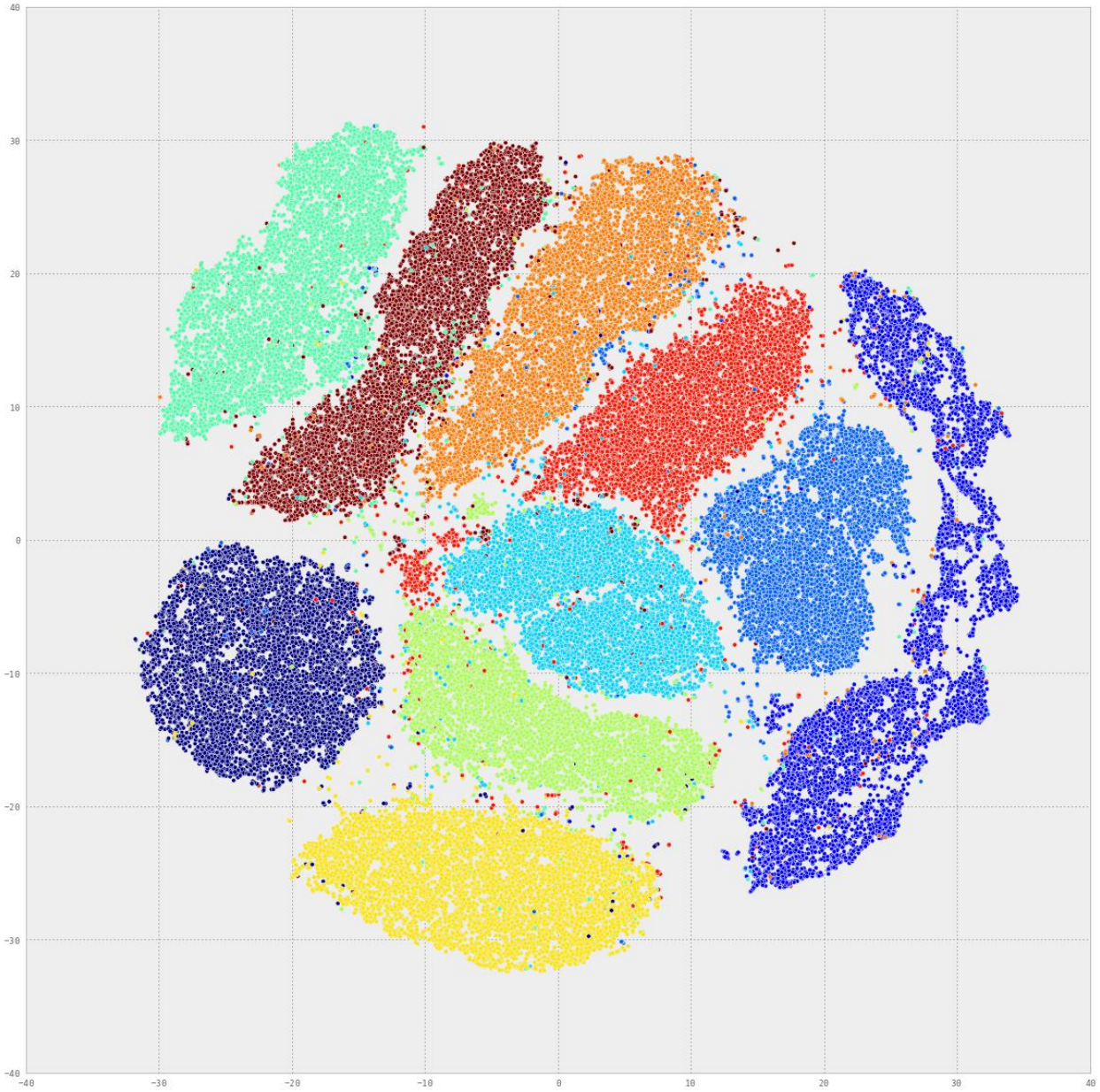
t-SNE

Another really powerful method for visualization is called t-SNE (pronounced, tea-SNEE), which stands for t-distributed stochastic neighbor embeddings. It's a non-linear dimensionality reduction that, again, aims to separate data in a way that clusters similar data close together and separates differing data.

As an example, below is a t-SNE reduction done on the MNIST dataset, which is a dataset of thousands of 28x28 images, similar to FashionMNIST, where each image is one of 10 hand-written digits 0-9.

The 28x28 pixel space of each digit is compressed to 2 dimensions by t-SNE and you can see that this produces ten clusters, one for each type of digits in the dataset!

CNN_FEATURE VISUALIZATION



t-SNE run on MNIST handwritten digit dataset. 10 clusters for 10 digits. You can see the generation code on Github.

t-SNE and practice with neural networks

If you are interested in learning more about neural networks, take a look at the **Elective Section: Text Sentiment Analysis**. Though this section is about text classification and not images or visual data, the instructor, Andrew Trask, goes through the creation of a neural network step-by-step, including setting training parameters and changing his model when he sees unexpected loss results.

He also provides an example of t-SNE visualization for the sentiment of different words, so you can actually see whether certain words are typically negative or positive, which is really interesting!

This elective section will be especially good practice for the upcoming section Advanced Computer Vision and Deep Learning, which covers RNN's for analyzing sequences of data (like sequences of text). So, if you don't want to visit this section now, you're encouraged to look at it later on.

Other Feature Visualization Techniques

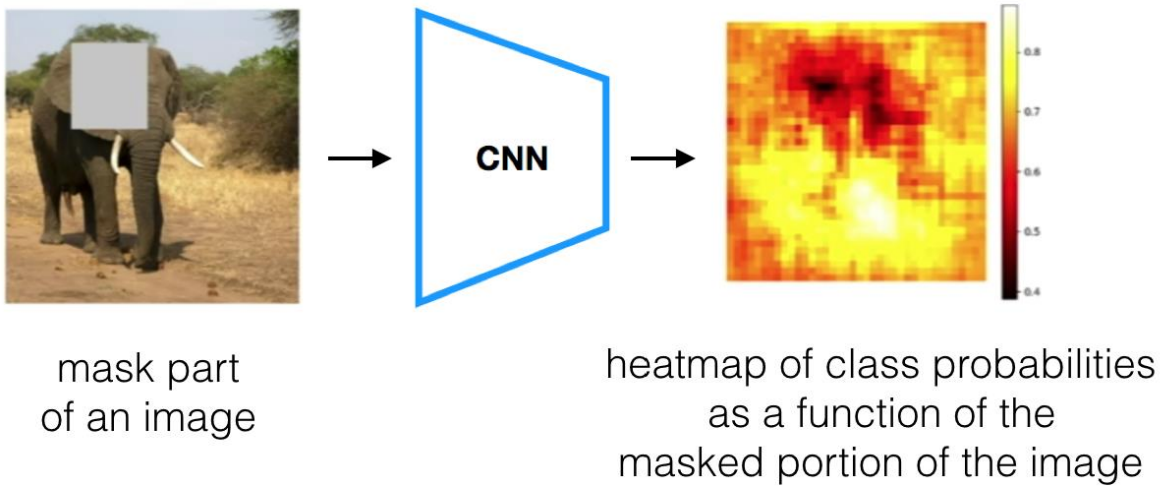
Feature visualization is an active area of research and before we move on, I'd like like to give you an overview of some of the techniques that you might see in research or try to implement on your own!

Occlusion Experiments

Occlusion means to block out or mask part of an image or object. For example, if you are looking at a person but their face is behind a book; this person's face is hidden (occluded). Occlusion can be used in feature visualization by blocking out selective parts of an image and seeing how a network responds.

The process for an occlusion experiment is as follows:

1. Mask part of an image before feeding it into a trained CNN,
2. Draw a heatmap of class scores for each masked image,
3. Slide the masked area to a different spot and repeat steps 1 and 2.
   The result should be a heatmap that shows the predicted class of an image as a function of which part of an image was occluded. The reasoning is that **if the class score for a partially occluded image is different than the true class, then the occluded area was likely very important**!



Occlusion experiment with an image of an elephant.

Saliency Maps

Salience can be thought of as the importance of something, and for a given image, a saliency map asks: Which pixels are most important in classifying this image?

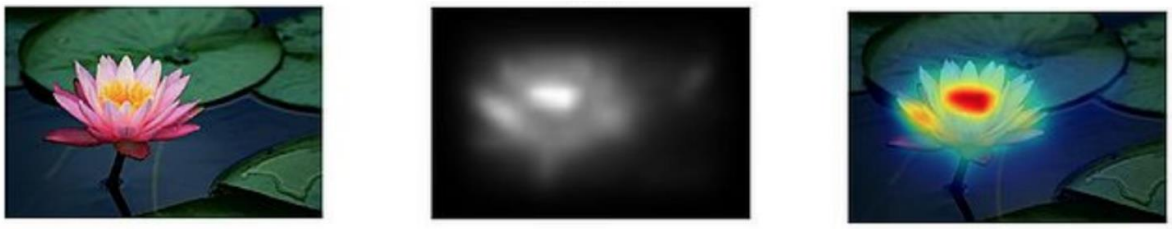
Not all pixels in an image are needed or relevant for classification. In the image of the elephant above, you don't need all the information in the image about the background and you may not even need all the detail about an elephant's skin texture; only the pixels that distinguish the elephant from any other animal are important.

Saliency maps aim to show these important pictures by computing the gradient of the class score with respect to the image pixels. A gradient is a measure of change, and so, the gradient of the class score with respect to the image pixels is a measure of how much a class score for an image changes if a pixel changes a little bit.

Measuring change

A saliency map tells us, for each pixel in an input image, if we change it's value slightly (by *dp*), how the class output will change. If the class scores change a lot, then the pixel that experienced a change, dp, is important in the classification task.

Looking at the saliency map below, you can see that it identifies the most important pixels in classifying an image of a flower. These kinds of maps have even been used to perform image segmentation (imagine the map overlay acting as an image mask)!



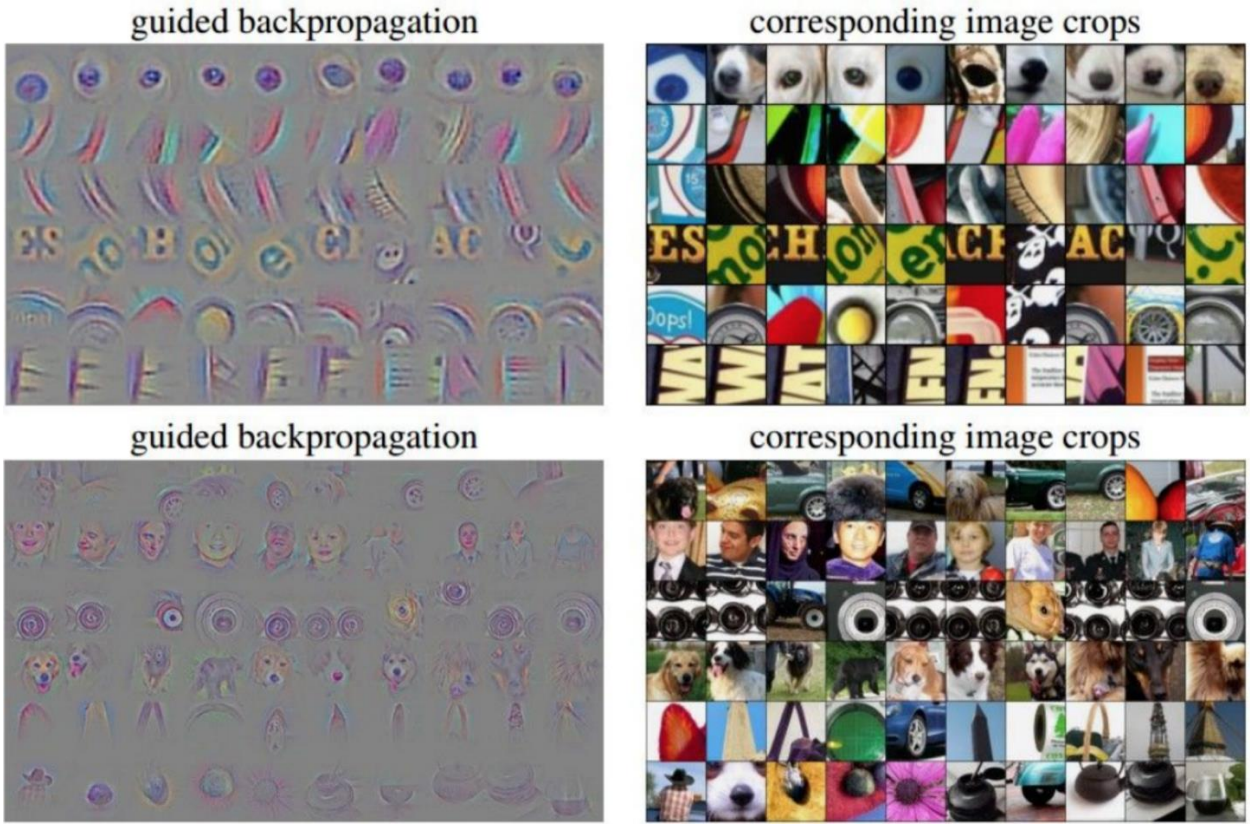
original image | saliency map | map overlay

Graph-based saliency map for a flower; the most salient (important) pixels have been identified as the flower-center and petals.

Guided Backpropagation

Similar to the process for constructing a saliency map, you can compute the gradients for mid level neurons in a network with respect to the input pixels. Guided backpropagation looks at each pixel in an input image, and asks: if we change it's pixel value slightly, how will the output of a particular neuron or layer in the network change. If the expected output change a lot, then the pixel that experienced a change, is important to that particular layer.

This is very similar to the backpropagation steps for measuring the error between an input and output and propagating it back through a network. Guided backpropagation tells us exactly which parts of the image patches, that we've looked at, activate a specific neuron/layer.



Examples of guided backpropagation, from this paper.

Supporting Materials

Visualizing Conv Nets

Guided Backprop Network Simplicity