Lesson 06 - Convolutional Neural Networks

Notes by: Jvinniec

Introduction

Some quick terms

- MLP (multi-layer perceptron)
- CNN (convolutional neural network)

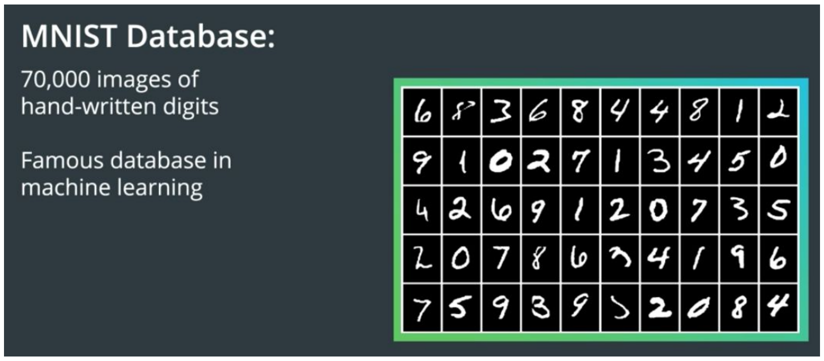
Applications of Convolutional Neural Networks

CNN's have a variety of use cases Including:

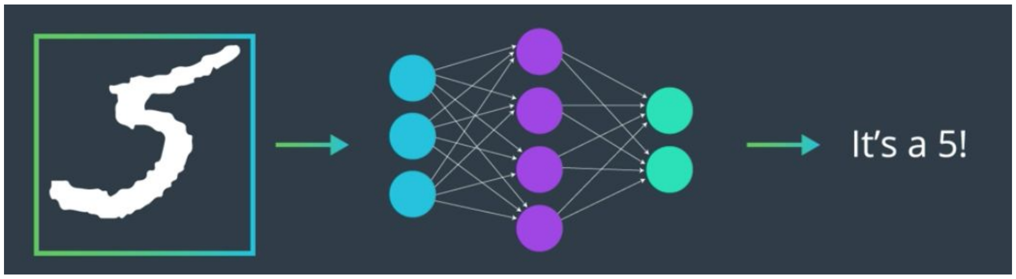
- Text to speech (WaveNet)
- Text classification
- Language translation
- Playing video games (also pictionary and Go too)
- Computer vision (such as drones and self-driving cars)

MNIST Data

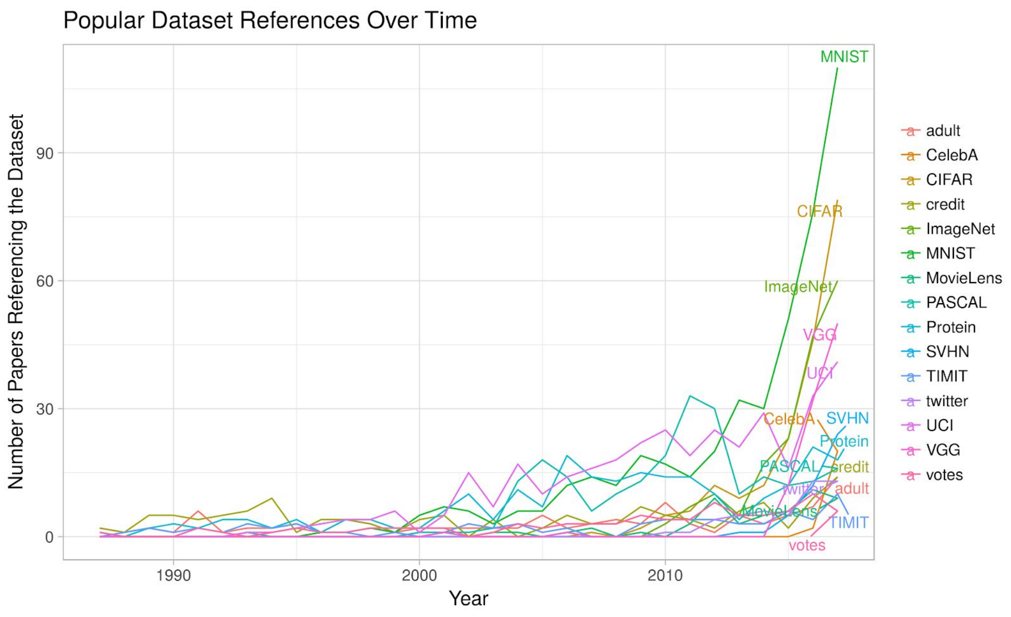
A collection of images representing hand drawn numbers.


MNIST Database:
70,000 images of hand-written digits
Famous database in machine learning

It can be used as a base test for new neural network and image classification techniques.



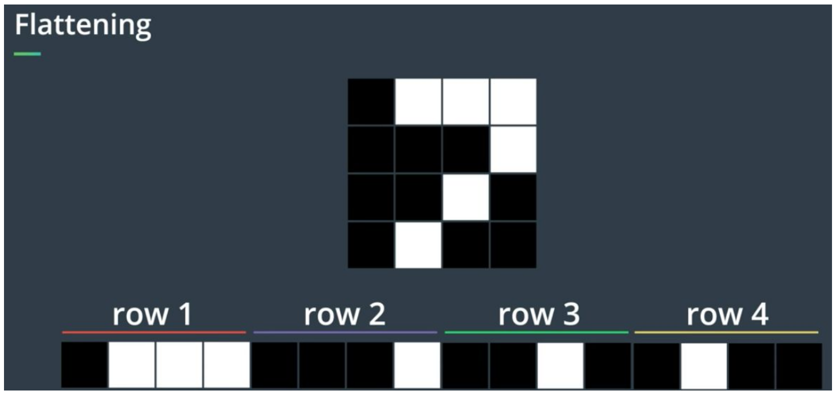
It is arguably one of the most referenced datasets involving deep learning.



MNIST Preprocessing

Each image in the MNIST dataset consists of 28x28 pixels. Each image has been processed so that:

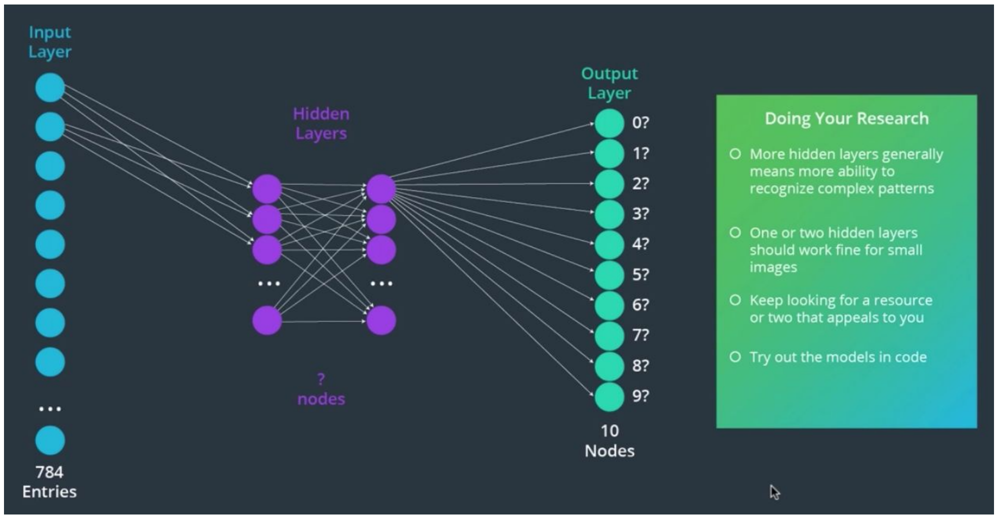
- Pixel values are normalized to the range [0,1]. This is important for standardizing the network inputs and helps our network train faster. This is done by subtracting the mean pixel value, then dividing by the standard deviation of pixel values. More information on linear transforms is available here.
- Each image has been "Flattened" into a single 1D array of 784 pixels.



QUESTION: How many hidden layers to use for the MNIST image classification problem?

ANSWER: “1” or “2” should be sufficient based on this source:

- https://www.learnopencv.com/image-classification-using-feedforward-neural-network-in-keras/



Doing Your Research

- More hidden layers generally means more ability to recognize complex patterns
- One or two hidden layers should work fine for small images
- Keep looking for a resource or two that appeals to you
- Try out the models in code

Review: Topics previously covered

Loss & Optimization

For a reminder on Loss and Optimization, review notes from lesson 3.

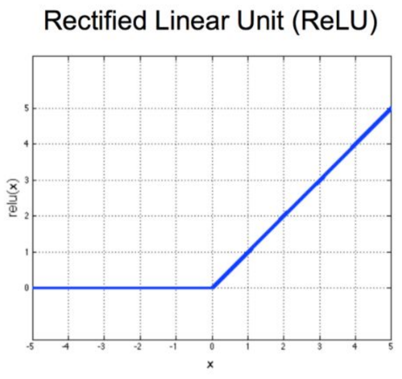
Defining a Network in PyTorch

For a review on how to construct models in PyTorch, review notes from lesson 5. Some brief reminders:

- We can define a model in a number of ways including as a class or using torch.nn.Sequential.

- The ReLU activation function scales the outputs of a layer so that they are a consistent, small value. This ensures our model trains efficiently. It is one of the most common activation functions in image classifications because it automatically sets negative image values to '0'.

$$\text{relu}(x) \begin{cases} 0, \text{ if } x < 0 \\ x, \text{ if } x >= 0 \end{cases}$$

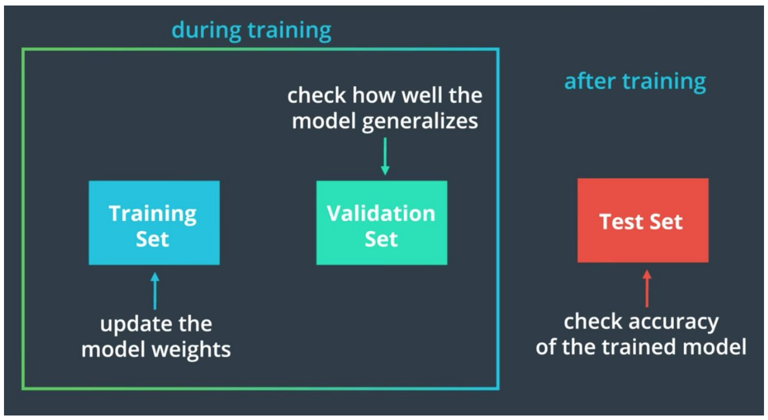


- The PyTorch implementation of cross-entropy loss involves the following steps.
    - Apply a softmax function to any output it sees
    - Apply NLLLoss (negative log likelihood loss)
    - Return a score for each class
- nn.CrossEntropy() can be replaced by calling 'nn.SoftMax(x, dim=1)' and using 'nn.NLLLoss()' as the loss function. This is useful if we want the model to return probabilities for each class.
- For loading the data we can use 'torch.utils.data.DataLoader' which has the following parameters:
    - batch_size: The number of data samples to load each time
    - num_workers: Number of processes to use when loading the data (0=use only the primary process).

Model Validation

Model validation is an important process when building a model. Typically we break the full training data into 3 sets:

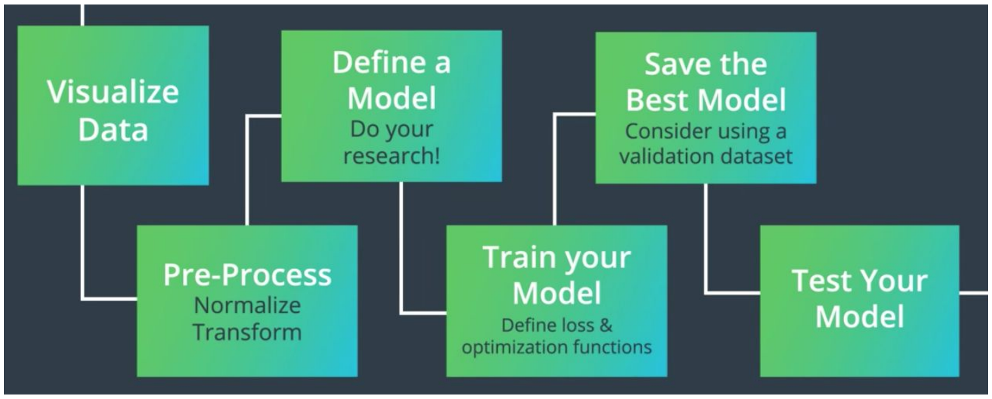
1. Training set: Portion of the data to use for training.
2. Validation set: Portion of the data used for validation DURING training. Generally, the best model during training has the smallest loss on the validation set.
3. Test set: Portion of data used to check the final accuracy of our data



Note that because we are using the validation set to decide things like “when to stop training” we are still using it as part of our training process, so we typically should not use it as our final testing set.

Image Classification Steps

The general steps to build an image classification model are as follows:

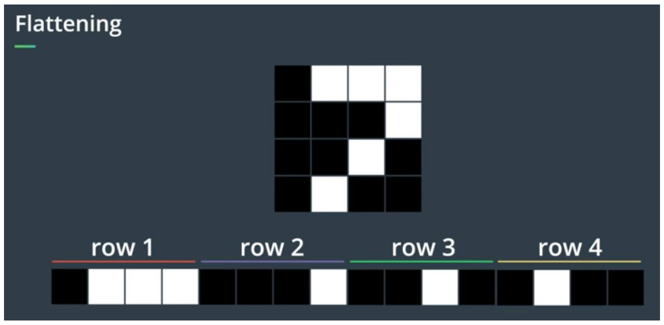


MLPs vs. CNNs

MNIST dataset is a unique case among image classification problems:

- It is heavily cleaned
- Numbers are approximately the same relative size and position.

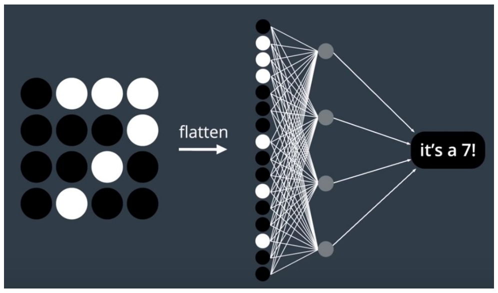
These points make it easier for MLPs to achieve relatively high accuracy on the MNIST data set because each pixel, though treated as an independent input, has a high probability of having a consistent value for all images from a given class.



On the other hand, CNNs are capable of taking spatial correlations between the different pixels into account in order to achieve a higher accuracy.

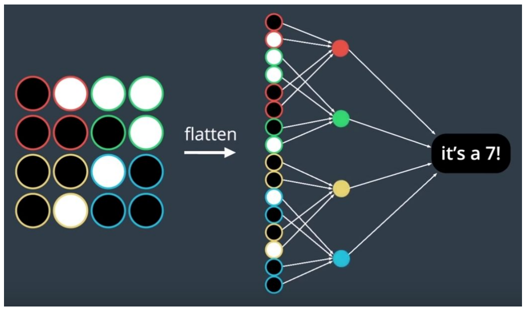
Issues with MLPs in Image Classification

- Only use fully connected layers
- Only accept vectors as inputs



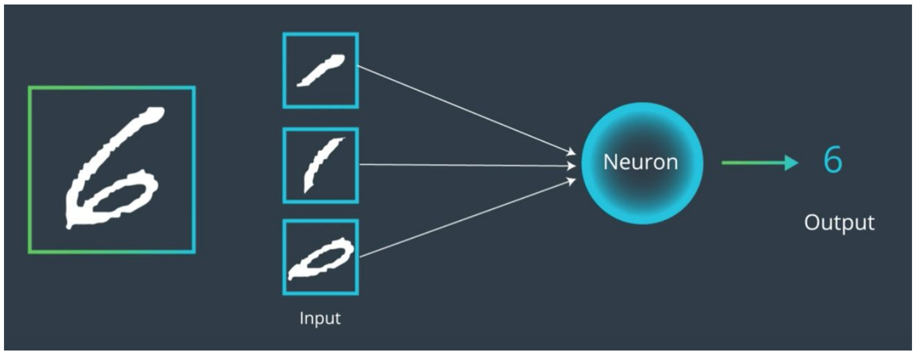
Benefits of CNNs in Image Classification

- Use sparsely connected layers
- Also accept matrices as inputs



Introduction to CNNs

CNNs learn the spatial patterns in an image that represent certain features. The weight of all of these features in our image tells us the most probable object in that image.



In the above image, we have a '6'. To the right of that we see 3 image features that our CNN model has learned. These are then combined in the final neuron to give our model's prediction.

Filters & Edges

To detect changes in intensity in an image, we create and use specific image filters. These filters look at groups of pixels and react to alternating patterns of dark/light pixels. Filters produce an output that shows edges of objects and differing textures. Filters are used to either:

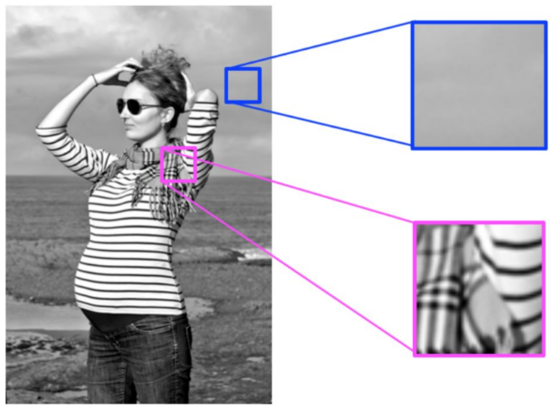
1. Filter out unwanted information

2. Amplify features of interest

Image Frequency relates to the rate of change of intensity of pixels in an image:

- Low frequency: Relatively uniform in brightness
- High frequency: Changes rapidly between bright and dim pixels.

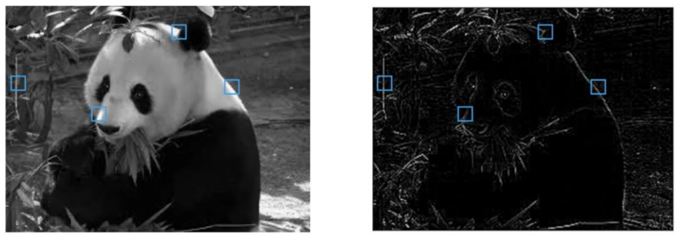
In the following image, the blue square highlights a region of low image frequency, while the pink square highlights a region of high image frequency.



High frequency components in an image correspond to the edges of objects in images, which can be used to help us classify them.

High-Pass Filter

A **high-pass filter** sharpens an image and enhances high-frequency parts of an image.



Edge Handling

Kernel convolution relies on centering a pixel and looking at its neighbors, but when we are looking at pixels on the border that don't have nearest neighbors we can use the following:

- **Extend**: Nearest border pixels are extended. Corner pixels are extended in 90 deg wedges.
- **Padding**: Image is padded with border of 0's.
- **Crop**: Any pixel in the output image that would require values from beyond the image is skipped. Method can cause output image to be slightly smaller.

Question: Given the following kernels choose which is the best for finding **horizontal** edges and lines?

a)

-1	-1	-1
-1	8	-1
-1	-1	-1

b)

-1	0	1
-2	0	2
-1	0	1

c)

0	-1	0
-2	6	-2
0	-1	0

d)

-1	-2	-1
0	0	0
1	2	1

Answer: We want to pick the kernel that represents a large difference in the top compared to the bottom. Such a kernel will pick out horizontal features. This would be kernel **(d)**.

OpenCV

OpenCV is a computer vision and machine learning software library which includes image analysis algorithms and tools to help with things such as:

- Image processing (e.g. area-of-interest selection)
- Python image processing (simply `import cv2`)

Creating a Filter/Edge Detection

Notes here derived from the 'Finding Edges' notebook.
Reading jpg image is very simple:

```
import matplotlib.image as mpimg    # matplotlib image loader
import cv2                          # OpenCV
# Load an image
image = mpimg.imread('images/curved_lane.jpg')
# Convert to grayscale for filtering
gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
```

The '**`cv2.cvtColor`**' method is converting the 3-channel RGB image into a 1-channel grayscale image.

Sobel Filter

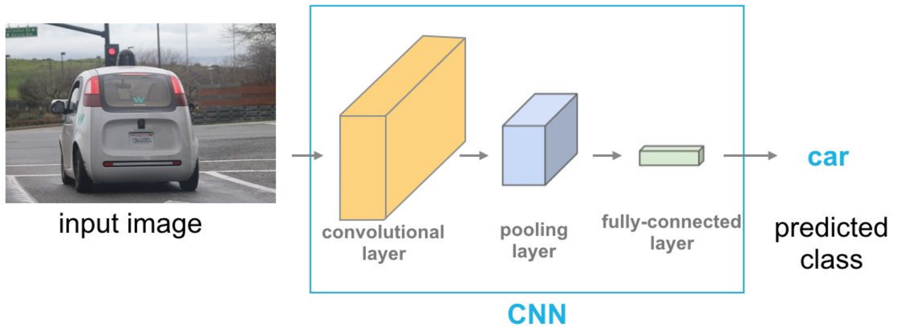
The Sobel filter is a common edge detection filter used to detect horizontal (S_y) and vertical (S_x) edges in an image:

$$S_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \qquad S_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

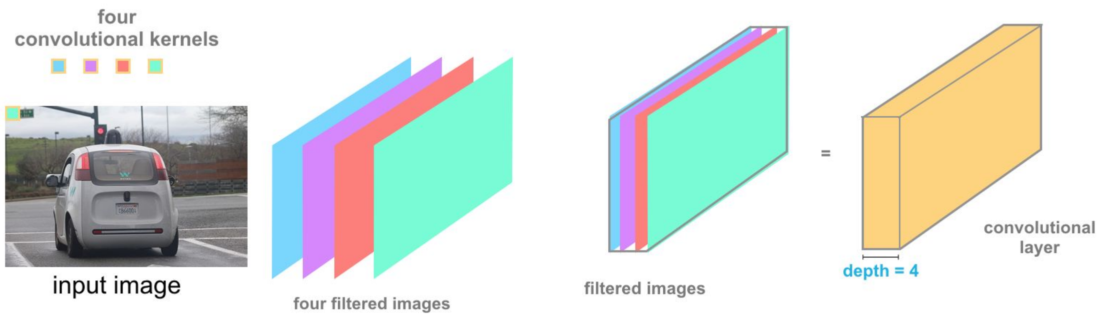
In practice, many neural networks learn to detect the edges of images, as the edges of objects contain valuable information about the object's shape.

Convolutional Layer

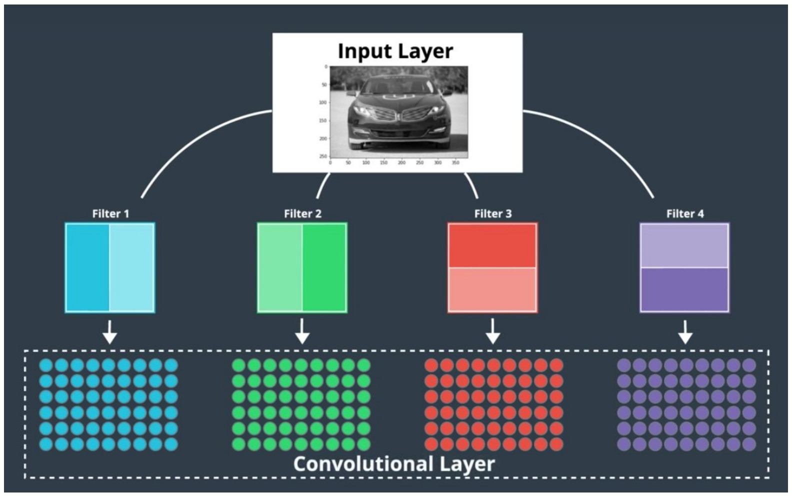
Convolutional neural networks (CNNs) are capable of keeping track of spatial information and learn to extract features like edges of objects. To do this, CNNs use a '**convolutional layer**':



Convolutional layers are produced by applying a series of many different image filters (i.e. the kernels like the ones we saw above). Each applied filter generates a different filtered output image that when stacked form a complete convolutional layer:

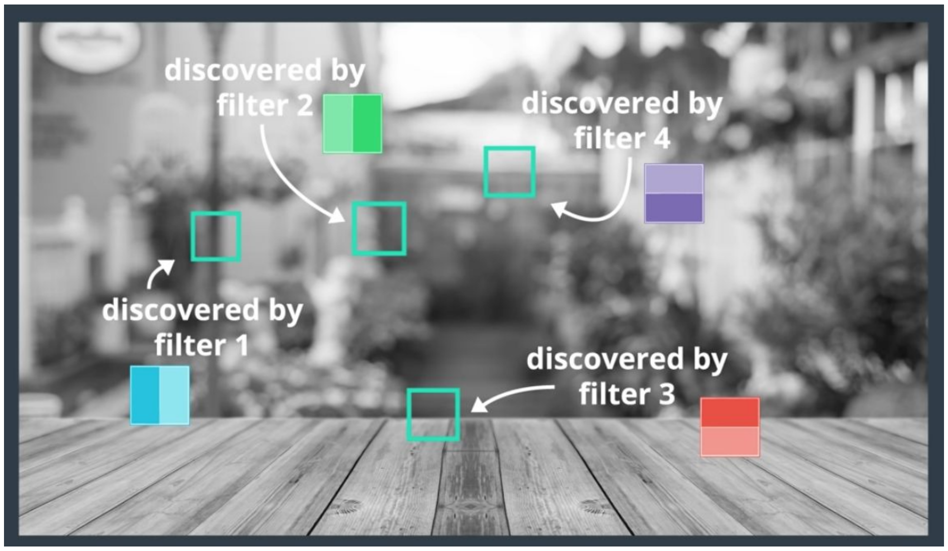


In CNNs the weights are applied to the filters and fitting modifies the values of these weights. Here's another way of visualizing the convolutional layer:



Convolutional Layer

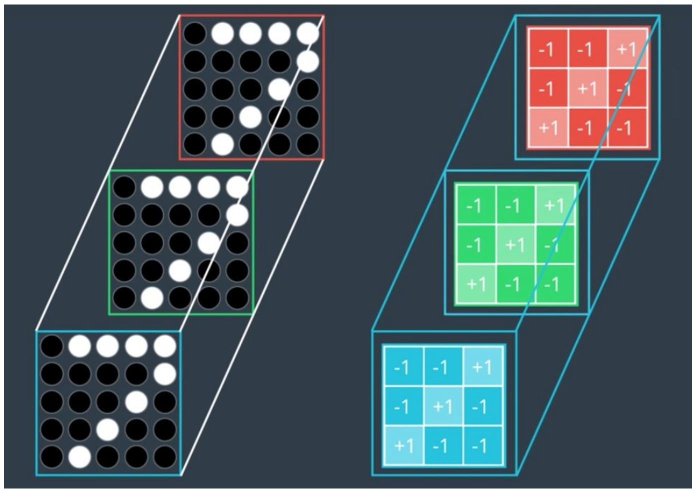
The filters in the above example extract different features by amplifying regions where the gradient across the image is large in a particular direction. For example:



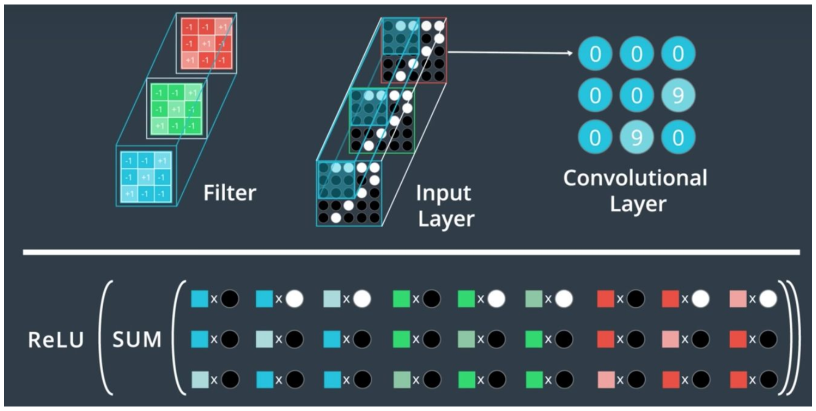
Color Images

Black and white images are simple 2D images with a height and width. Color images, on the other hand, also have depth. Specifically, color images have 3 color channels for RGB. Similarly, we can think of our convolution kernels as aslso having 3 color channels:

In the image below, our image is represented on the left while our 3D kernel is shown on the right.



We can represent the actual convolution as something like the following; each layer of our image (the ‘input layer’) is convolved by the convolved by the filter.

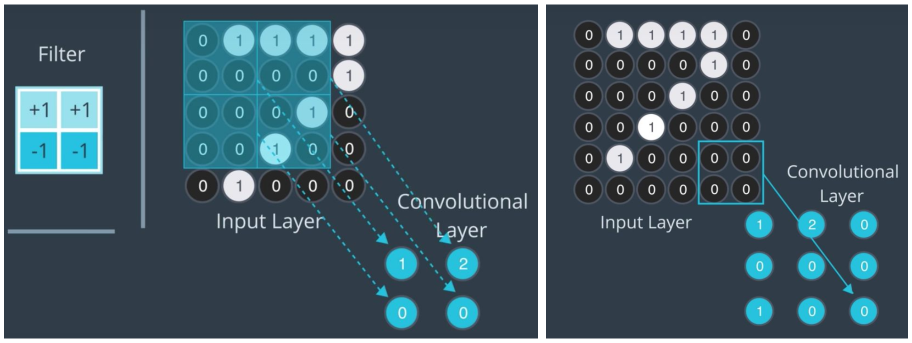


Stride and Padding

Stride is a tuning parameter in CNNs. It measures how many pixels we move each time we apply the filter to our image (in the examples above the stride was always 1).

- The stride effects the size of the generated convolutional layer, where a stride of N will result in a convolutional layer with width and height each a factor of ~1/N times smaller. So a stride of 2 will result in a convolutional layer that is half the width and height of the input.

How we treat border pixels also matters when it comes to our stride. If we ignore regions where the filter does not fully overlap the image, we can end up with regions of our image that our CNN has no knowledge of (see following image which uses stride=2). This is not ideal as we are losing information!



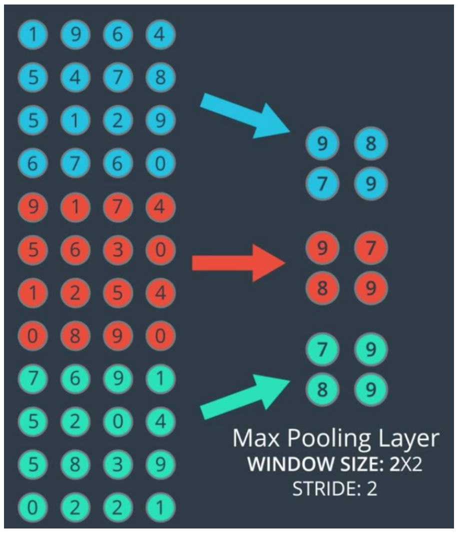
Padding is what we do to get around this. By knowing the size of our filter and stride we can pad the border of the image with 0’s (right image) to preserve the information near the edge.

Pooling Layer

Convolutional layers are stacks of feature maps. But there can be many feature maps making the number of parameters very large which increases the complexity of our model. Pooling is a way to reduce the size of our convolutional layer.

A few types of pooling:

- **Max Pooling**
  Max Pooling takes our convolutional layer and applies a window to it. It keeps only the largest pixel value within that window and ignores the rest (see image).
- **Average Pooling**
  Average pooling is the same as Max Pooling, but it takes the average of the pixels in the window instead of the maximum.



Max pooling is typically preferred over average pooling because it is better at noticing the most important details about edges and other features.

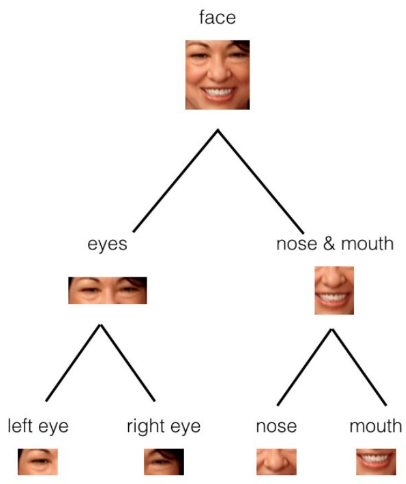
Pooling Alternatives

Because pooling ends up throwing away information (such as the orientation between features), other methods exist to better encapsulate

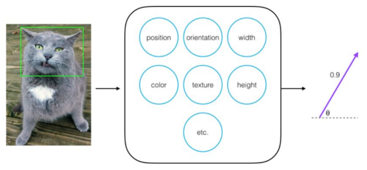
Capsule Networks

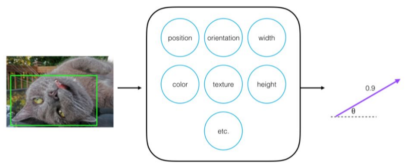
Capsule networks provide a way to detect parts of objects in an image and represent spatial relationships between those parts.

- Allows capsule networks to recognize the same object in a variety of different poses even if it has not seen that pose in training.
- Networks are made up of parent and child nodes that build up a complete picture of an object



- Capsules are a collection of nodes each containing information about a specific part (width, orientation, color, position, etc…)
- Each capsule **outputs a vector** with a magnitude (probability that a part exists) and orientation (the state of the part properties, 'theta')

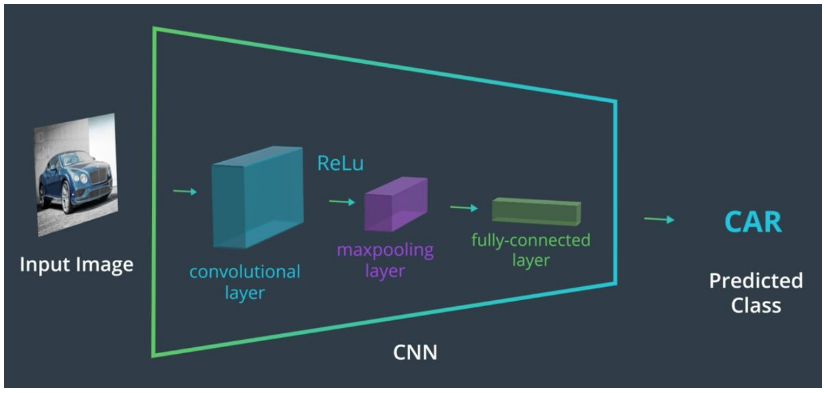




- More details here: https://cezannec.github.io/Capsule_Networks/

Putting it All Together

CNN's work by determining appropriate filters that are capable of extracting features from an image given its classification. Applying a ReLU activation function causes all negative negative pixels in the resulting convolutional layer to be set to 0. 'Pooling' enables the network to be compact while still having enough information to describe complex features.



Question: How might you define a Maxpooling layer such that it down-samples an input by a factor of 4?

> **Answer**: By using a pooling function with a stride of 4 ('nn.MaxPool2d(2,4)' or 'nn.MaxPool2d(4,4)')

Question: If you want to define a convolutional layer that is the same x-y size as an input array, what padding should you have for a `kernel_size` of 7?

> **Answer**: padding=3. Noting that the center pixel in the kernel (i.e. the 4th pixel for a 7-pixel kernel) is fixed at each pixel during convolution, we will need to pad the image to account for the first 3 pixels.

CNN's in PyTorch

Pytorch uses the following functions when defining CNNs:

- `nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0)`
    - **In_channels**: depth of the output (grayscale=1, color=3)
    - **out_channels**: desired depth of the output (or number of filtered images)
    - **Kernel_size**: size of convolutional kernel (most commonly 3 for 3x3 kernel)
    - stride & padding: have defaults, but should be set depending on desired size of output.
- `nn.MaxPool2d(kernel_size, stride=None, padding=0, dilation=1, return_indices=False, ceil_mode=False)`
    - **kernel_size**: size of window to take max over
    - stride: stride of the window (default = kernel_size)
    - padding: implicit zero padding to be added on both sides
    - dilation: controls the stride of elements in the window
    - return_indices: if True, returns the max indices along with outptus
    - ceil_mode: if True, use ceil instead of floor to compute the output shape

Convolutional Layers

Construction of convolutional layers is basically applying 3 steps in succession:

1. **Apply the convolution with 'nn.Conv2d'**
   The inputs to this layer are indicated above
2. **Apply the forward function such as 'nn.ReLU'**
   Inputs here are the outputs from 'nn.Conv2d'
3. **Apply the pooling layer with 'nn.MaxPool2d'**
   Inputs are as defined above.

This should construct the convolutional layers and the max pooling layer.

Example 1:
We want to construct a CNN with the following parameters:

1. Input layer accepts 200x200 pixels grayscale images
2. Feed these images into a convolutional layer with 16 2x2 pixel filters that take steps of 2 pixels.
3. The filter will not extend beyond the image boundaries (i.e. no padding)

To construct this network we would use the following line:

```
self.conv1 = nn.Conv2d(1, 16, 2, stride=2)
```

Example 2:
We want to construct a convolutional layer as that:

1. takes the output from the first layer
2. puts it into another convolutional layer with 32 3x3 pixel filters and stride of 1.

3. Should have the same width and height as the input layer (i.e. we will need padding).

This layer looks like this:

```
self.conv2 = nn.Conv2d(16, 32, 3, stride=1)
```

Sequential Models

We can simplify model construction by using a `nn.Sequential` model that contains all of the convolutional, pooling, and activation layers.

```
def __init__(self):
   super(ModelName, self).__init__()
   self.features = nn.Sequential(
           # First Convolutional layer
           nn.Conv2d(1, 16, 2, stride=2),
           nn.MaxPool2d(2, 2),
           nn.ReLU(True),

           # Second Convolutional layer
           nn.Conv2d(16, 32, 3, padding=1),
           nn.MaxPool2d(2, 2),
           nn.ReLU(True)
       )
```

Note that we've defined both layers in a neat little package.

Number of parameters in a Convolutional Layer

The number of parameters in a convolutional layer depends on:

- K (out_channels): number of filters in the cnn layer
- F (kernel_size): height and width of the convolutional filters
- D_in (input_shape[-1]): depth of previous layer

Where:

- K*F*F*D_in = number of weights
- K = number of biases

So the total number of parameters is:

$$N_{pars} = K * F * F * D_{in} + K$$

Shape of a Convolutional Layer

The spatial shape of a given convolutional layer can be represented as:

$$L_{shape} = 1 + (W_{in} - F + 2P)/S$$

where:

- W_{in} : width/height (square) of the previous layer
- F (kernel_size) : height and width of the convolutional filters
- S (stride) : stride of the convolution
- P (padding) : the padding
- K (out_channels): the number of filters in the convolutional layer

Question:
Suppose we have a 130x130 pixel color image that goes through the following layers in order:

```
nn.Conv2d(3, 10, 3)
nn.MaxPool2d(4, 4)
nn.Conv2d(10, 20, 5, padding=2)
nn.MaxPool2d(2, 2)
```

What is the resulting depth of the final output?

Answer: The final depth is always determined by the number of output channels in the final convolutional layer. In the above, that is '20', so the **final depth is 20**.

Question:
What is the x-y size of the output of the final maxpooling layer in the above network?

Answer: Since everything we're dealing with is square (i.e. x,y are the same size in all filters and the image input) we only need to look at one dimension. For this we apply the formula above from "Shape of Convolutional Layers":

First "Conv2d":
in_spatial(W_{in})=130, kernel_size(F)=3, padding(P)=0, ~~out_channels=10~~, stride(S)=1.
So we have a total number of output channels of:
$out = 1 + (130 - 3 + 2 \cdot 0) / 1 = 128$

First "MaxPool2d":
~~kernel_size=4~~, stride=4 We know that the maxpooling will downsample the image by a factor equal to its stride, so
$out = 128 / 4 = 32$

Second "Conv2d":
in_spatial(W_{in})=32, kernel_size(F)=5, padding(P)=2, stride(S)=1
$out = 1 + (32 - 5 + 2 \cdot 2) / 1 = 32$

Second "MaxPool2d":
stride=2
$out = 32 / 2 = 16$

So the final number of spatial pixels will be **16x16** pixels

Question:
How many parameters will there be after passing through the entire network?

Answer: The final number of parameters will be equal to the number of spatial pixels times the depth, or 16x16x20 = 5120

Flattening

Flattening is an important part of building a CNN so that all parameters can be seen as a vector by a linear classification layer.

Putting it all together

In the end, a full CNN has the following *general* form:

A series of convolution/maxpooling layers that each do the following:

1. '`nn.Conv2d`' : Convolve the input image by a set of filters
2. '`nn.ReLU`' : Apply some activation, typically ReLU
3. '`nn.MaxPool2d`' : Pool the resulting images to down-sample them

A flattening operation to format the output of the above convolution layers:

- Typically of the form '`x = x.view(-1, <spatial bins>)`'

A series of fully connected layers to generate the output. These follow the same format as our MLP layers:

1. '`nn.Dropout`' : Pass the flattened array through a dropout layer
2. '`nn.Linear`' : Fully connected layer
3. '`nn.ReLU`' : Activation function

A final fully connected layer to generate the output classifications

CIFAR10

CIFAR 10 is a dataset of tiny (32x32 pixels) color images that contain objects from a series of 10 categories. It is available in the "`torchvision.datasets`" database. View a sample network generated to classify these images here and here.

Image Augmentation

We want the classification algorithm to learn an **invariant representation** of our objects. Meaning, we want our model to correctly identify objects regardless of

- position(**translation invariance**),
- orientation (**rotation invariance**), or
- size (**scale invariance**)

of the object in the image.

We can achieve this through “**data augmentation"** which involves creating copies of our images that are flipped, rotated, or scaled.

Image Augmentation in PyTorch

PyTorch has a builtin library for image augmentation: “torchvision.transforms”. Some useful transforms include:

- ‘`transforms.ColorJitter`’ : Randomly changes brightness, contrast, and saturation of an image.
- ‘`torchvision.transforms.Grayscale`’ : Convert an image to grayscale
- ‘`torchvision.transforms.RandomCrop`’ : Crop an image at a random location
- ‘`torchvision.transforms.RandomHorizontalFlip`’ : Flip an image horizontally with a given probability
- ‘`torchvision.transforms.RandomVerticalFlip`’ : Flip an image vertically with a given probability
- ‘`torchvision.transforms.RandomRotation`’ : Rotate image by a random angle within a specified range
- ‘`torchvision.transforms.Resize`’ : Resize image to a given size

These transformations are applied when we load the training and testing data and are added to the ‘`transforms.Compose`’ before applying ‘`transforms.ToTensor`’.

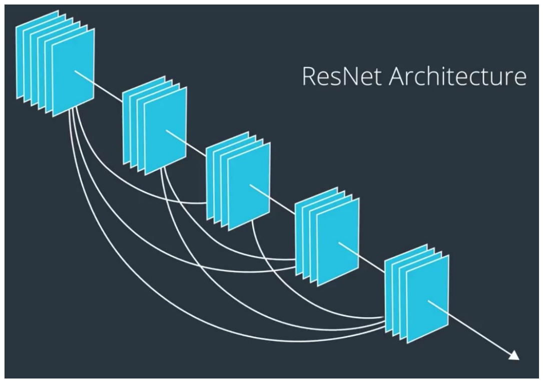
Groundbreaking CNN Architectures

ImageNet is a massive database of images. Benchmarks for some of the different CNN architectures on this dataset are available here. Here are descriptions of some of the best performing architectures over the years:

AlexNet (2012) was a breakthrough CNN architecture that pioneered the use of the **ReLU** activation function and **dropout** to avoid overfitting.

VGGNet (2014) is an architecture of repeating 3x3 convolution layers broken up by 2x2 pooling layers and finished with 3 fully connected layers. It pioneered the use of small 3x3 convolution windows (AlexNet used 11x11 windows).

ResNet (2015) used a massive series of convolution layers connected in a way that allows the gradient computation to skip layers. This allows the network to bypass the vanishing gradients problem that often plagues networks with many layers.

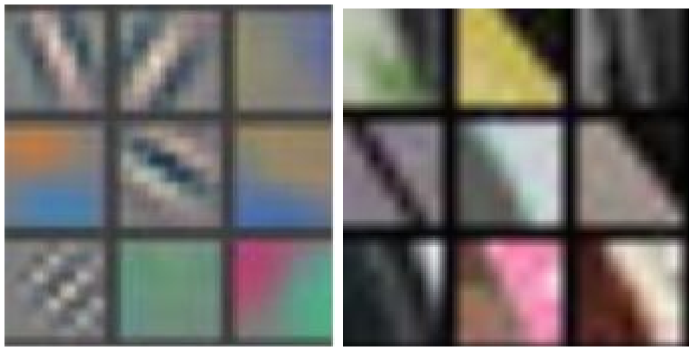


Visualizing CNNs

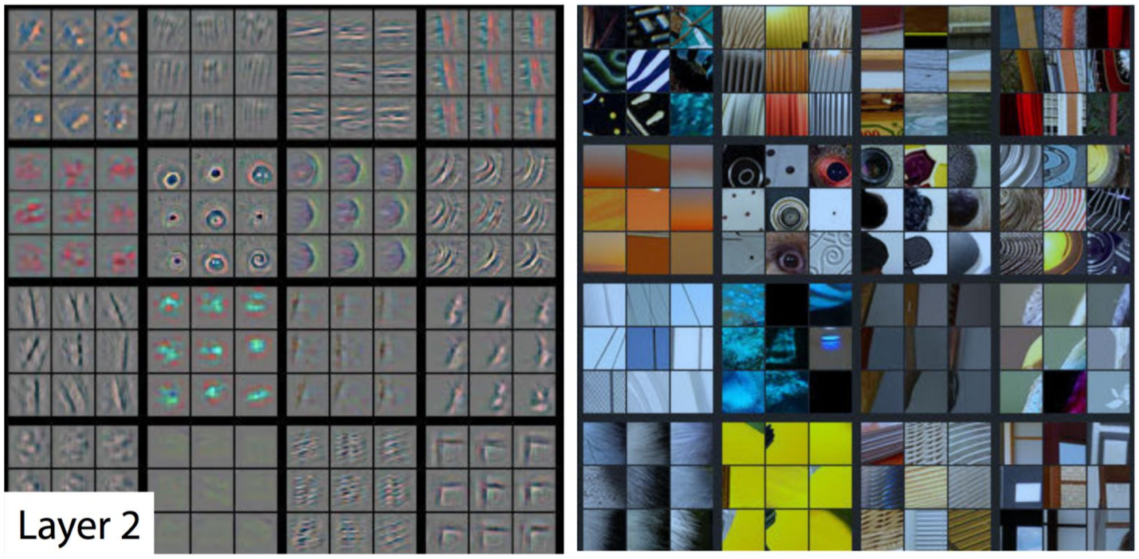
One of the non-trivial things in CNNs is to understand what features a CNN has learned and is using to classify an image. One way to do this is by taking an image that scores very high for a given category, then visualize the independent output convolutions from each layer. These visuals will show which regions of the image are most associated with that category, and thus what the CNNs are picking up on.

So what are CNNs seeing? Consider a multi-layer CNN

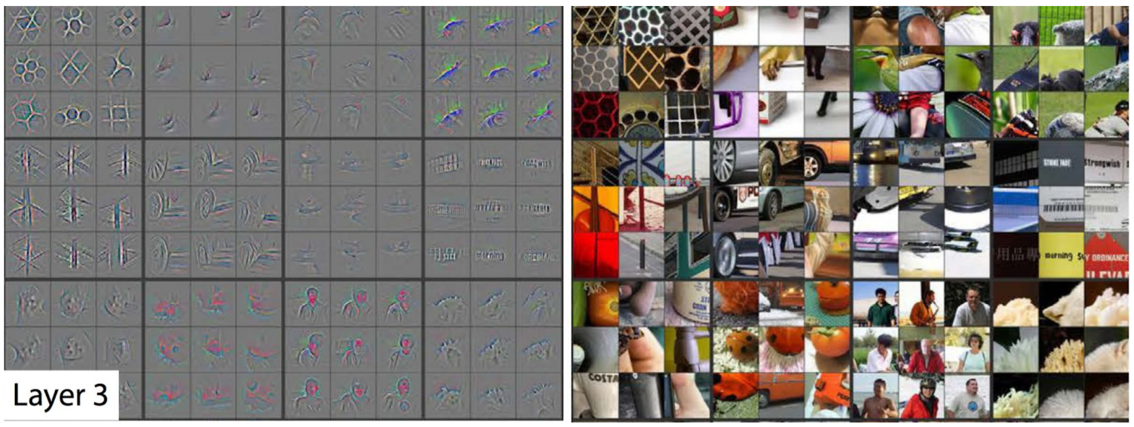
1. The first layer is typically simple features such as lines or gradients across an image.



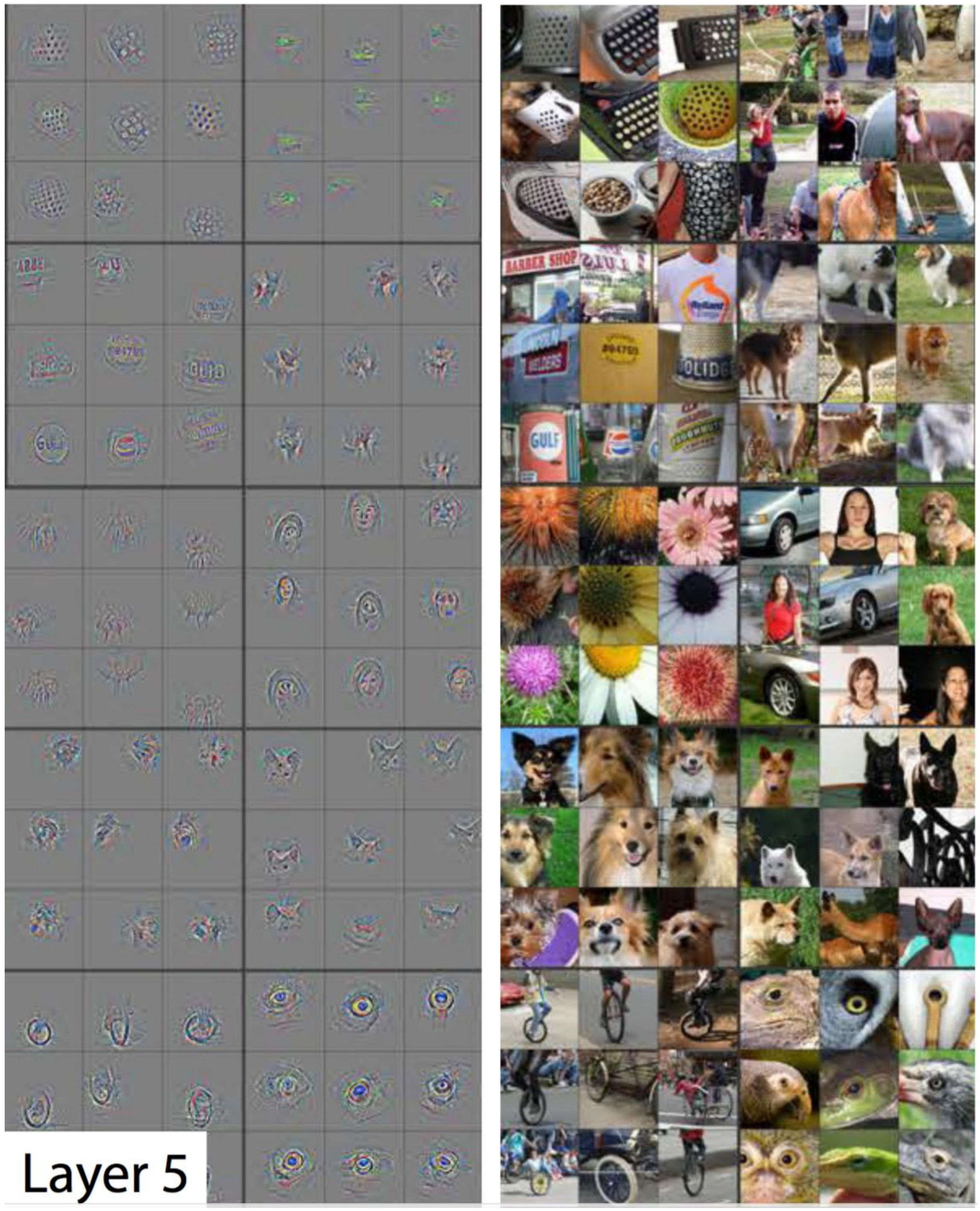
2. The second layer picks out slightly more complex features, such as circles, stripes, or corners



3. The third layer starts to pick up on complex combinations of features from the second layer, such as grids, honeycombs, wheels, even faces



4. Further layers continue this progression until very complex features have been learned, such as distinct types of faces or facial features.



Here is a further resource on visualizing what convolutional networks learn.