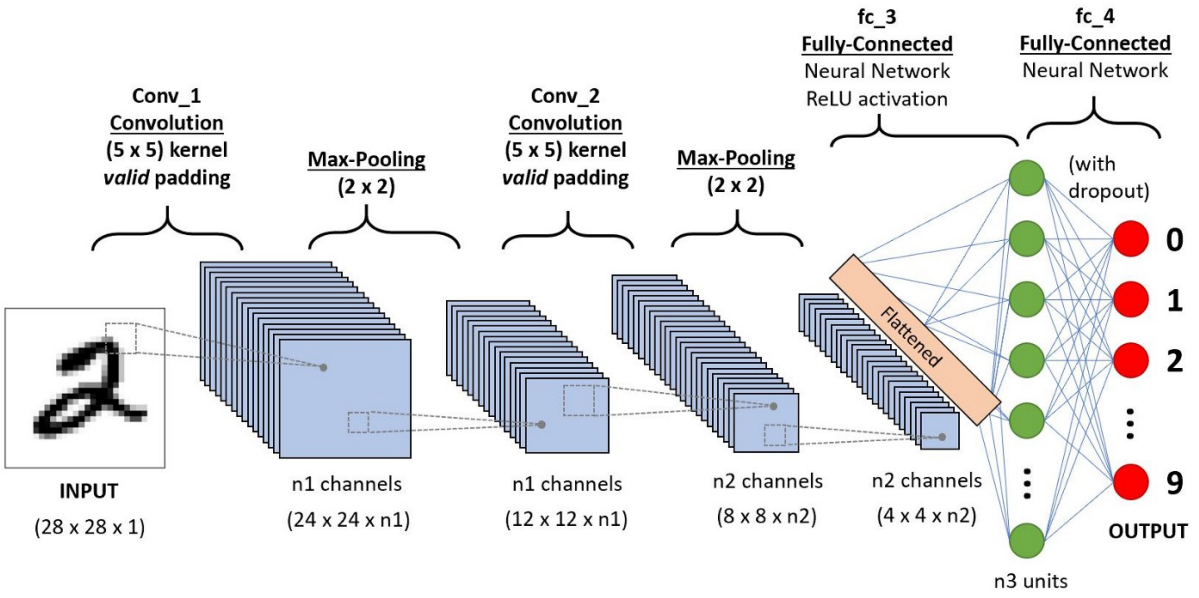
CNN and YOLO

CNN

Neural Networks in general are composed of a collection of neurons that are organized in layers, each with their own learnable weights and biases.

The convolutional neural network, or CNN for short, is a specialized type of neural network model designed for working with two-dimensional image data, although they can be used with one-dimensional and three-dimensional data.



Convolutional layers are the major building blocks used in convolutional neural networks. A convolution is the simple application of a filter to an input that results in an activation. Repeated application of the same filter to an input results in a map of activations called a feature map, indicating the locations and strength of a detected feature in an input, such as an image.

Let’s break down CNN into its basic building blocks.

1. A **tensor** can be thought of as an n-dimensional matrix. In the CNN above, tensors will be 3-dimensional with the exception of the output layer.
2. A **neuron** can be thought of as a function that takes in multiple inputs and yields a single output.
3. A **layer** is simply a collection of neurons with the same operation, including the same hyperparameters.
4. **Kernel weights and biases**, while unique to each neuron, are tuned during the training phase, and allow the classifier to adapt to the problem and dataset provided.
5. A CNN conveys a **differentiable score function**, which is represented as **class scores**.

What does each layer of the network do?

Let’s walk through each layer in the network.

Input Layer

The input layer (leftmost layer) represents the input image into the CNN. Because we use RGB images as input, the input layer has three channels, corresponding to the red, green, and blue channels, respectively.

Convolutional Layers

The convolutional layers are the foundation of CNN, as they contain the learned kernels (weights), which extract features that distinguish different images from one another.

The convolutional neuron performs an elementwise dot product with a unique kernel and the output of the previous layer's corresponding neuron. This will yield as many intermediate results as there are unique kernels. The convolutional neuron is the result of all of the intermediate results summed together with the learned bias.

Understanding Hyperparameters

1. **Padding** is often necessary when the kernel extends beyond the activation map. Padding conserves data at the borders of activation maps, which leads to better performance, and it can help preserve the input's spatial size, which allows an architecture designer to build depper, higher performing networks. There exist many padding techniques, but the most commonly used approach is zero-padding because of its performance, simplicity, and computational efficiency. The technique involves adding zeros symmetrically around the edges of an input.
2. **Kernel size**, often also referred to as filter size, refers to the dimensions of the sliding window over the input. Choosing this hyperparameter has a massive impact on the image classification task. For example, small kernel sizes are able to extract a much larger amount of information containing highly local features from the input. a smaller kernel size also leads to a smaller reduction in

layer dimensions, which allows for a deeper architecture. A large kernel size extracts less information, which leads to a faster reduction in layer dimensions, often leading to worse performance. Large kernels are better suited to extract features that are larger.

Activation Functions

ReLU

ReLU stands for **Rectified linear Unit**. It is the most widely used activation function implemented in hidden layers. ReLu is less computationally expensive than tanh and sigmoid because it involves simpler math operations. At a time only a few neurons are activated making the network sparse making it efficient and costly for computation. ReLU learns much faster than sigmoid and tanh function.

Softmax

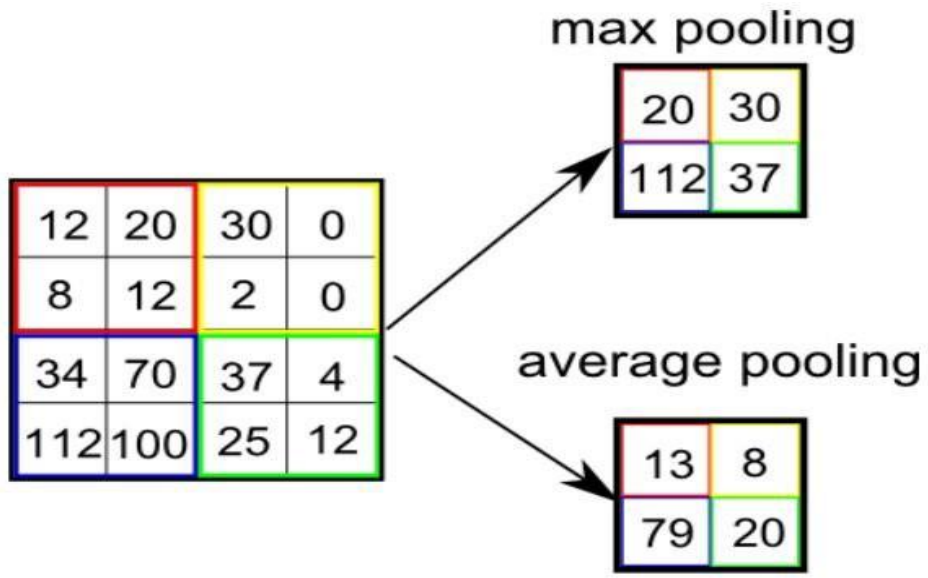
This is also a type of sigmoid function but is handy when we are trying to handle classification problems. It is used when trying to handle multiple classes. The softmax function would squeeze the outputs for each class between 0 and 1 and would also divide by the sum of the outputs.

Pooling Layers

Similar to the Convolutional Layer, the Pooling layer is responsible for reducing the spatial size of the Convolved Feature. This is to decrease the computational power required to process the data through dimensionality

reduction. Furthermore, it is useful for extracting dominant features which are rotational and positional invariant, thus maintaining the process of effectively training the model.

There are two types of Pooling: Max Pooling and Average Pooling. **Max Pooling** returns the **maximum value** from the portion of the image covered by the Kernel. On the other hand, **Average Pooling** returns the **average of all the values** from the portion of the image covered by the Kernel. Max Pooling also performs as a Noise Suppressant. It discards the noisy activations altogether and also performs de-noising along with dimensionality reduction. On the other hand, Average Pooling simply performs dimensionality reduction as a noise suppressing mechanism. Hence, we can say that **Max Pooling performs a lot better than Average Pooling**.



The Convolutional Layer and the Pooling Layer, together form the i-th layer of a Convolutional Neural Network. Depending on the complexities in the images, the number of such layers may be increased for capturing low-level details even further, but at the cost of more computational power.

Flatten Layer

This layer converts a three-dimensional layer in the network into a one-dimensional vector to fit the input of a fully-connected layer for classification. For example, a 5x5x2 tensor would be converted into a vector of size 50. The previous convolutional layers of the network extracted the features from the input image, but now it is time to classify the features. We use the softmax function to classify these features, which requires a 1-dimensional input. This is why the flatten layer is necessary. This layer can be viewed by clicking any output class.

YOLO

The YOLO framework (You Only Look Once) deals with object detection in a different way. It takes the entire image in a single instance and predicts the bounding box coordinates and class probabilities for these boxes. The biggest advantage of using YOLO is its superb speed – it's incredibly fast and can process 45 frames per second. YOLO also understands generalized object representation.

This is one of the best algorithms for object detection and has shown a comparatively similar performance to the R-CNN algorithms.

How does the YOLO Framework Function?

- YOLO first takes an input image:

- The framework then divides the input image into grids (say a 3 X 3 grid):

- Image classification and localization are applied on each grid. YOLO then predicts the bounding boxes and their corresponding class probabilities for objects (if any are found, of course).

We need to pass the labelled data to the model in order to train it. Suppose we have divided the image into a grid of size 3 X 3 and there are a total of 3 classes which we want the objects to be classified into. Let's say the classes are Pedestrian, Car, and Motorcycle respectively. So, for each grid cell, the label y will be an eight dimensional vector:

y =	pc
	bx
	by
	bh
	bw
	c1
	c2
	c3

Here,

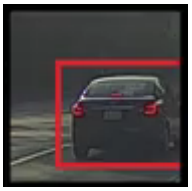
- pc defines whether an object is present in the grid or not (it is the probability)
- bx, by, bh, bw specify the bounding box if there is an object
- c1, c2, c3 represent the classes. So, if the object is a car, c2 will be 1 and c1 & c3 will be 0, and so on

Let's say we select the first grid from the above example:

Since there is no object in this grid, pc will be zero and the y label for this grid will be:

y =	0
	?
	?
	?
	?
	?
	?
	?

Here, '?' means that it doesn't matter what bx, by, bh, bw, c1, c2, and c3 contain as there is no object in the grid. Let's take another grid in which we have a car (c2 = 1):



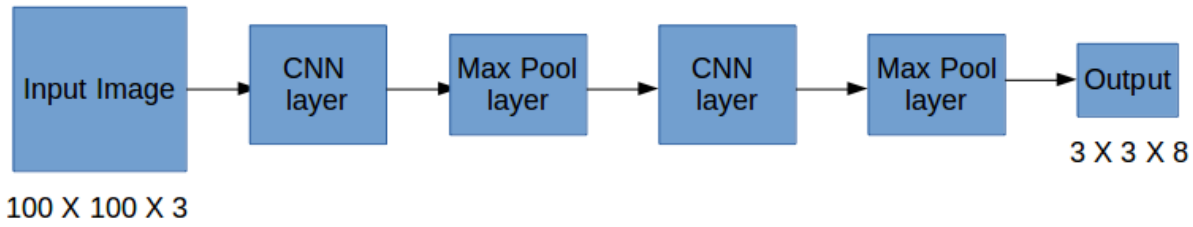
Before we write the y label for this grid, it's important to first understand how YOLO decides whether there actually is an object in the grid. In the above image, there are two objects (two cars), so YOLO will take the mid-point of these two objects and these objects will be assigned to the

grid which contains the mid-point of these objects. The y label for the centre left grid with the car will be:

y =	1
	bx
	by
	bh
	bw
	0
	1
	0

Since there is an object in this grid, pc will be equal to 1. bx, by, bh, bw will be calculated relative to the particular grid cell we are dealing with. Since a car is the second class, c2 = 1 and c1 and c3 = 0. So, for each of the 9 grids, we will have an eight dimensional output vector. This output will have a shape of 3 X 3 X 8.

So now we have an input image and its corresponding target vector. Using the above example (input image – 100 X 100 X 3, output – 3 X 3 X 8), our model will be trained as follows:



We will run both forward and backward propagation to train our model. During the testing phase, we pass an image to the model and run forward propagation until we get an output y. In order to keep things simple, I have

explained this using a 3 X 3 grid here, but generally in real-world scenarios we take larger grids (perhaps 19 X 19).

Even if an object spans out to more than one grid, it will only be assigned to a single grid in which its mid-point is located. We can reduce the chances of multiple objects appearing in the same grid cell by increasing the number of grids (19 X 19, for example).

Comparison

- YOLO and Faster RCNN both share some similarities. They both use an anchor box based network structure, both use bounding regression.
- But Something that differs YOLO from Faster RCNN is that it makes classification and bounding box regression at the same time.
- YOLO however does have it's drawback in object detection. YOLO has difficulty detecting objects that are small and close to each other due to only two anchor boxes in a grid predicting only one class of object. It doesn't generalize well when objects in the image show rare aspects of ratio.
- Faster RCNN on the other hand, does detect small objects well since it has nine anchors in a single grid, however it fails to do real-time detection with its two step architecture.