Neural networks and back-propagation algorithm

//basics of machine learning, linear algebra, neural network architecture, cost functions, optimization methods, training/test sets, activation functions/what they do, etc.

**Problem**

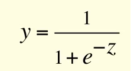
**Computer Vision**

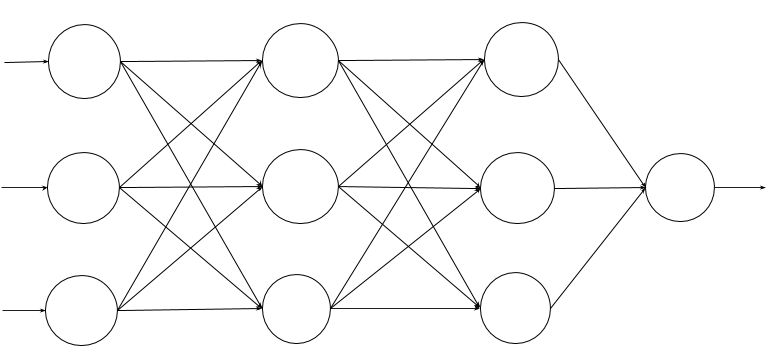
Computer vision — which is really an interdisciplinary field — is about computers being able to gain high-level understandings and make conclusions, predictions, and/or statements from some kind of visual input.

This field does not refer to a one concrete tasks but to a group of tasks that machine learning has attempted to solve. One of them is that of object classification

Neural Networks

Neural Networks are essentially mathematical models to solve an optimization problem. They are made of neurons, the basic computation unit of neural networks. A neuron takes an input(say x), do some computation on it(say: multiply it with a variable w and adds another variable b ) to produce a value (say; z= wx+b). This value is passed to a non-linear function called **activation function(f)** to produce the final output(activation) of a neuron. There are many kinds of activation functions. One of the popular activation function is **Sigmoid,** which is:



Fundamentally, neural networks are nothing more than really good function approximations  which must be trained. To train our network to estimate an unknown function, we give it a collection of data points — which we denote the “training set” — that the network will learn from and generalise on to make future inferences.

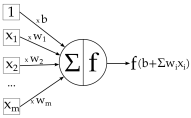
*This is what a neural network looks like.*

*Each circle is a* ***neuron****, and the arrows are connections between neurons in consecutive* ***layers****.*

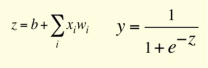
Neural networks are structured as a series of **layers**, each composed of one or more **neurons** (as depicted above). Each neuron produces an output, or **activation**, based on the outputs of the previous layer and a set of weights.

When using a neural network to approximate a function, the data is forwarded through the network layer-by-layer until it reaches the final layer. The final layer’s activations are the predictions that the network actually makes.

The neuron which uses sigmoid function as an activation function will be called **Sigmoid neuron**. Depending on the activation functions, neurons are named and there are many kinds of them like **RELU, TanH** etc(remember this). One neuron can be connected to multiple neurons, like this:



This is the complete picture of a sigmoid neuron which produces output y:



Operations on individual pixels aren’t enough to let us figure out if these pixels represent a spoon. Instead, we need to figure out the pixel groupings that make these edges/lines/corners and see how groupings of those edges/lines/corners then go on to form the characteristics. Furthermore, flattening images into single vectors — though retaining information of the pixels — loses information such as the **structure** of the image. Our neural network would not be able to exploit this structure, which is certainly important information when it comes to recognising objects.

**Moravec’s Paradox**

Marvin Minsky (R.I.P.) said it best, though:

*In general, we’re least aware of what our minds do best… we’re more aware of simple processes that don’t work well than of complex ones that work flawlessly.*

**Architecture of a Convolutional Neural Network**

So the problem with using a normal neural network is that it’s basically impossible to extract any meaningful features from a bunch of pixels directly. To solve this, a CNN uses a hierarchical approach to *learn feature detectors that are increasingly abstract*.In a convolutional neural network, we have a very similar principle — a convolutional **kernel** (or **filter**) describes an individual pattern, which is then applied to every part of our image.

With CNNs, we talk about **volumes** instead of normal vectors. Instead of a 1-D vector of numbers that we pass into our network, it’s conceptually easier to envision our image as a 100 x 100 x 3 volume of numbers (100 pixels wide, 100 pixels tall, and 3 channels [R, G, B] deep for the colors).

We make these transformations using **convolutional layers**. In a normal fully-connected layer, we would have one weight per input for each neuron; in a convolutional layer, we instead learn the weights of a **filter** that we apply to every part of our input volume. If our initial input volume is 100 x 100 x 3, we might learn a 5 x 5 x 3 filter that we apply to each individual part/sub-section of our image.

The most important parameter in a convolution neutron is filter size.

Convolution is a mathematical operation that’s used in single processing to filter signals, find patterns in signals etc.

**Convolutional Layer:**

In a convolutional layer, all neurons apply convolution operation to the inputs, hence they are called convolutional neurons.

Typically, we use more than 1 filter in one convolution layer.Instead of convolving just one filter and creating just one filter map, we learn many filters (often as many as hundreds per layer) and apply each one to the input volume. This gives us as many feature maps as we have many filters, which we can stick together to create a larger volume.Each filter map will give us a different unique perspective on the image (unique patterns, lines, borders, etc.), which when combined together give us a new representation of the image which is far more meaningful than the initial image or any individual feature map.

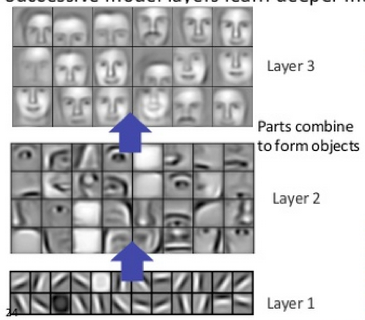
**Pooling Layer:**

Pooling layers are a way to reduce the size of our volume as it flows through our network. We’ll start by looking at one particular kind of pooling layer, the max-pooling layer, but the same principle applies to related layers, like average-pooling.

Notice that pooling layers have *no parameters* (only hyperparameters) — they simply apply a function to regions of its input. Intuitively, you can think about pooling layers as reducing the “resolution” of our input volume; if we go back to our feature map of lines, and apply a pooling operation, we will end up with basically the same picture just with fewer “pixels”.

Convolutional and pooling layers make up the bulk of most CNNs, but we still need to somehow come up with a probability distribution over classes. We can do this with our normal fully-connected layers we are so used to seeing in regular neural networks. Our last convolutional or pooling layer gives us one last output volume, with very high-level and abstract features (ears, eyes, leaves, etc.).

So now, how do we actually learn or define these filters and their specifically weights/biases? Our objective is to find filters that minimize error/cost and in doing so maximize the percentage of correctly classified images. We of course use the backpropagation algorithm to compute our derivatives and then apply a first-order or second-order optimization algorithm like stochastic gradient descent or L-BFGS respectively. The cost function we use can be any typical one, for example cross-entropy (or “logistic”) loss.

You need a lot of data to train one of these networks successfully, and depending on the problem you’re solving, there isn’t always all that much data to go around. Data augmentation can help, but nothing beats a ginormous dataset with hours of GPU training time. But because of the nature of pictures, a lot of the features that CNNs are learn are fairly generic; the lines, patterns, textures, and so on that a network learns to look for can often be found in most images, even in categories it may not have been explicitly trained on. For this reason, it’s common to start training with a popular architecture and an already-trained model; if you freeze the weights of the convolutional layers and just train some fully-connected layers on top of them, you can learn to recognize an entirely different distribution of objects with far fewer images.

<https://deeplearning4j.org/>neuralnet-overview

**The Back-propagation Algorithm**

For any given supervised machine learning problem, we (aim to) select weights that provide the optimal estimation of a function that models our training data. In other words, we want to find a set of weights **W** that minimises on the output of **J(W)**.

For other machine learning algorithms like logistic regression or linear regression, computing the derivatives is an elementary application of differentiation. This is because the outputs of these models are just the inputs multiplied by some chosen weights, and at most fed through a single activation function (the sigmoid function in logistic regression). The same, however, cannot be said for neural networks.

**Implementing Back-propagation**

The implementation is divided into 3 different steps:

* **Feed-forward**. In this step, we take the inputs and forward them through the network, layer by layer, to generate the output activations (as well as all of the activations in the hidden layers). When we are actually using our network (rather than training it), this is the only step we’ll need to perform.
* **Backpropagation**. Here, we’ll take our error function and compute the weight gradients for each layer. We’ll use the algorithm just described to compute the derivative of the cost function w.r.t. each of the weights in our network, which will in turn allow us to complete step 3.
* **Weight update**. Finally, we’ll use the gradients computed in step 2 to update our weights. We can use any of the update rules discussed previously during this step (gradient descent, momentum, and so on).