A CNN source-code dissection, for non-expert readers

sergio.r.f.oliveira@ieee.org – 28-April-2020

This article is a very detailed explanation, for non-expert readers, of a source code that implements a Convolutional Neural Network. I myself am not an expert in this area, but I "did my homework" to understand the subject and I took the time to implement the source code and write this article, since I've found no other detailed, satisfying text written for non-specialist people like me. The better approximation of such endeavor is this article by Victor Zhou, who presents an algorithm that is used in my program, and has great graphical resources for explaining the general principles of a CNN. I do not intend to repeat his explanations, which are very well done, so I recommend the reader to refer to his article before reading mine.

Note: if, for some reason, the above URL becomes unavailable, you may try his GitHub page.

I decided to implement Victor Zhou's algorithm in MATLAB, not in Python, for two reasons: 1) in my experience, MATLAB code runs faster than Python code for crunching numbers; 2) I did not want my program to be a simple reincarnation of his code. If you don't have a MATLAB license, no worries: the code was tested successfully in its freeware incarnation OCTAVE. However, be aware that it runs much slower in OCTAVE than in MATLAB! Additionally, my program was conceived to process CIFAR-10 datasets, whereas Victor Zhou's code was written to process the MNIST datasets.

An important disclaimer: Since it's a program written for pleasure and education, it was not optimized for speed or efficiency, but for understanding and clarity. It is a tribute to the beauty and elegance of CNNs.

The program is composed of 16 files, as seen in Figure 1. Each file implements one and only one function. In Figure 1, the order of execution of those files is from top-down, left-right. Let's explain each one of them.

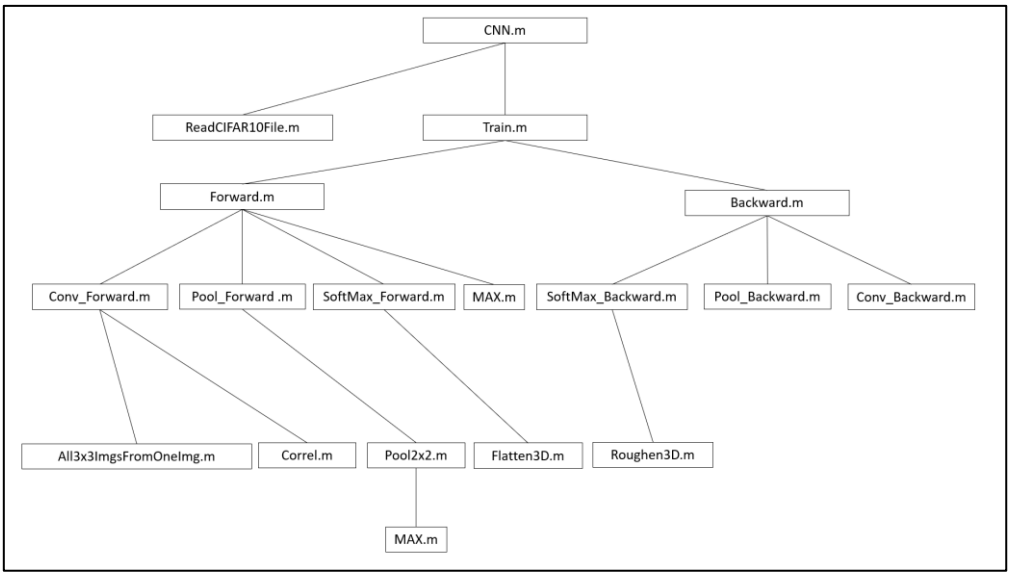

Figure 1

CNN.m, the entry point, is a MATLAB script that calls the other files. It contains 3 sections: one for initialization of variables, one for training the CNN, and one for testing the CNN. The most important variable is a structure called LastData. The first 3 fields of that structure – Weights, Filters and Biases – are filled in the initialization section. Other fields will be created and filled by the functions Conv_Forward, Pool_Forward and SoftMax_Forward. The variable LastData plays the hole of an encapsulation structure, since I'm not using Object-Oriented Programming. Each "forward" function fills data in it that will be latter consumed by the "backward" functions.

ReadCIFAR10File.m, as the name suggests, reads one CIFAR10 data file and returns ten thousand 32x32 grayscale images and their respective labels. CIFAR-10 datasets contain, overall, sixty thousand 32x32 RGB images of ten categories (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck), as well as their sixty thousand respective labels (0…9). The images are returned in a 3D array (10000x32x32) and their labels are returned in a 10000x1 matrix. I opted for converting the original RGB images to grayscale for the sake of simplicity (color has not such an importance in this demonstration). Since MATLAB does uses zero-based indexes, it was necessary to add 1 to each label to obtain the range 1…10. (A side note: this is one of my favorite characteristics of MATLAB compared to other languages: there's no zero indexes – all things begin from 1)

There are five training files in the dataset, each one containing ten thousand images. The first for loop iterates on each file, after being shuffled by the "randperm" instruction. The objective of shuffling the files is to increase the randomness of the training process.

Train.m is then called for each image by the second for loop (total 50000 calls, after being shuffled by the second "randperm" instruction – as before, the objective of shuffling the images is to increase the randomness of the training process). By its turn, Train.m calls Forward.m, calculates the gradient of probabilities, then calls Backward.m. Note that the only gradient assigned a value is the one corresponding to the correct label: all the other gradients keep zero values. Those probabilities, as we will see momentarily, are produced by the SoftMax_Forward function.

Note: The word "gradient" will be very used here, because the algorithm described takes advantage of the Gradient Descent method.

Forward.m calls three functions: Conv_Foward (to execute the correlations of images and filters), Pool_Forward (to take samples of the correlations), and SoftMax_Forward (to calculate probabilities of the image being one of the ten categories). After that, the Forward function decides the accuracy: one, if the maximum calculated probability (which came from the SoftMax_Forward function) coincides with the correct label, zero otherwise. To obtain the maximum value from the probability array, it's used the MAX function, described next.

Observe that when calling Conv_Foward, each pixel value of the image is divided by 255 and subtracted of 0.5. The effect of that operation is converting a 0…254 range to a

–0.5…+0.5 range. This is necessary because the correlation operations will be done against filters having values concentrated in this range.

MAX.m is a simple function that takes a matrix as input, returns its maximum value and the first coordinate (row, column) where the maximum value was found.

Conv_Forward.m saves the input image inside the LastData structure (it will be used later by the Conv_Backward function), then calls two functions: All3x3ImgsFromOneImg and Correl. The first one generates all possible 3x3 images derived from the image it received in the parameters. (In fact, the sub-images are generated by a 3x3 window, sliding from left to right, top to down, one pixel at a time). Those sub-images are saved inside the LastData structure for future use by the backward functions, and are also used by the next function – Correl – to compute correlations between each filter and each sub-image.

All3x3ImgsFromOneImg.m, as said above, receives one 32x32 image as input parameter, and generates a 30x30x3x3 output array containing all the 3x3 sub-images taken from the input image. Why 30x30 and not 32x32? Well, if you do the math, a slide window of size 3x3, moving from left to right, top to down, one pixel at a time, can only generate (N-2)x(N-2) sub-images from a NxN image.

Correl.m computes the correlations between each one of the filters and each one of the sub-images received in the parameters. The correlation computation is basically the piecewise multiplication of a 3x3 filter by a 3x3 sub-image and the summation of the results (one 3x3 filter correlated against one 3x3 sub-image generates just one scalar value, which is proportional to the "similarity" between the filter and the sub-image). Since we have F filters with dimension Fx3x3, and 30x30 sub-images from one image, the result of the correlation operations is a 30x30xF array. This operation is the core of the CNN algorithm, and in my opinion the name "Convolutional Neural Network" should be "Correlational Neural Network" (convolution is a mathematical operation much more complex then correlation). This kind of filtering has been used for decades in the signal processing arena. For more information, see https://en.wikipedia.org/wiki/Matched_filter.

Pool_Foward.m is the next function to be executed. Again, the name "pooling" sounds inappropriate: in my opinion, "sampling" would be a better designation. Anyway, that's the name that prevailed in the field. What this function does is to extract samples of the array generated by the Conv_Forward function to reduce its size and, consequently, to reduce the number of following computations. The sampling is done by the Pool2x2 function, to be described next.

Pool2x2.m is the function that executes the sampling of the previous correlation array. The sampling algorithm used is this: have a 2x2 window sliding through the correlation array, in row-major order; pick the maximum value inside the window (using the MAX function, already described); store that value and the position (row, column) where it was found. Since the input array has dimension 30x30xF (where F is the number of filters) and the sliding windows has dimension 2x2, the result is an array with dimension 15x15xF (for the samples) and another array 15x15xFx2 (for the positions).

Before exiting, the Pool_Forwad function stores inside the LastData structure the samplings and its respective sampling coordinates for future use by the Pool_Backward function.

SoftMax_Forward.m is the next function to be executed. It receives as input the samples of correlated values. It saves the shape of the input for future use by the backward functions (in this case, the shape is 15x15xF). It does that because it will flatten the array, that is, it will transform the 3D array into a single row of data: a matrix of dimension 1x(15x15xF). The function that performs the flattening is called Flatten3D (to be described next). It also saves the flattened array inside the LastData structure for future use by the backward functions. After that, it performs a matrix multiplication of the flattened array by the weights. The flattened array is a matrix with dimension 1x(15*15*F), and the weights are a matrix with dimension (15*15*F)x10, so the matrix multiplication has dimension 1x10. Next, it transposes the result, obtaining a 10x1 matrix, and adds the biases to it (if you don't remember how to multiply and transpose matrices, see https://en.wikipedia.org/wiki/Matrix_(mathematics)). Next, it applies the exponential function to the results, and divides by the sum of the results for obtaining values in the range 0…1 with sum equal to 1 (formally, this is called a normalization operation). In other words, those values are equivalent to probabilities. Remember when I described the Train function? I've mentioned theses probabilities there. They are proportional to the probabilities of the image being processed to be one of the 10 possible categories. Obviously, these probabilities must be "calibrated". In fact, the weights, filters and biases must be "calibrated". The calibration process will be done by the SoftMax_Backward, Pool_Backward and Conv_Backward functions, to be described momentarily. Finally, the SoftMax_Forward function saves two intermediate results inside the LastData structure for future use.

The SoftMax algorithm is one of many algorithms that can be used to translate a set of real numbers into a set of probabilities. I don't know why, but it seems to be the preferred algorithm used in the Convolutional Neural Network arena. Personally, I find the Sigmoid algorithm more elegant, but that's the way it is.

Note: My code implements a trick to prevent numerical instabilities in the SoftMax algorithm (look for "exp-normalize trick" at Google).

Flatten3D.m, a function called by SoftMax_Forward, requires no deep explanation. It takes as input a 3D array with dimension (NxMxK) and transform it into a 1x(N*M*K) matrix, following the row-major order.

The SoftMax_Forward is the last function called by the Forward function described above. Next, the execution is returned to the Train function. Remember: after calling the Forward function, the Train function calculates the gradient of probabilities, but the only gradient assigned a value is the one corresponding to the correct label. This trick seems to be fundamental in the "calibration" of filters, weights and biases, which is done by the backward functions! It is presented in the Victor Zhou article without deep explanation, but its origin seems to be explained by this article, at the sentence: […] *if*

any of the neurons in set L were not connected to neuron j, they would be independent of w_{ij} and the corresponding partial derivative under the summation would vanish to 0".

The computed gradient is then passed to the Backward function, which is described next.

Backward.m calls three functions, in sequence: SoftMax_Backward, Pool_Backward and Conv_Backward. To the first one it passes the gradient of probabilities computed by the Train function, as well as the correct label and a parameter called Learning Rate.

SoftMax_Backward.m is the most complicated function, since it involves a lot of Calculus. It begins recovering from the LastData structure some intermediate values previously computed and saved by the SoftMax_Forwad function (not repeating those computations saves some CPU cycles). Next, it applies two different formulas to compute the gradients of the probabilities: one formula is applied to the probabilities of the wrong categories, and the other formula is applied to the probability of the correct category. The derivation of those formulas may be seen in the Victor Zhou's article mentioned above. Next, it applies the chain rule to compute the updated gradients of probabilities, weights and biases. Then, it calls the Roughen3D function to recover the original shape of the data that had been flattened by the SoftMax_Forward function and saved inside the LastData structure. Finally, it computes and saves inside the LastData structure the new, "calibrated" values of weights and biases.

Roughen3D.m does the inverse of what had been done by the Flatten3D function described above.

When the function SoftMax_Backward exits, the control returns to the function Backward, which now call the function Pool_Backward, passing to it the updated probability gradients.

Pool_Backward.m begins recovering from the LastData structure the original sampled 15x15xF array and its respective sampling coordinates (row, column) that were computed and saved by the Pool_Foward function. From them, it reconstructs a new 30x30xF array using the following trick: picking the coordinates of each sample, it stores the respective probability gradient in that position, leaving zeros in the rest. This trick is very, very interesting! The reconstructed 30x30xF array will be fed into the Conv_Backward function.

Conv_Backward.m recovers the 3x3 sub-images that were saved by the Conv_Forward function. Then it updates the sub-images using the gradients received from the tricky Pool_Backward function. After that, it multiplies the result by the Learning Rate parameter. To compute the updated, "calibrated" filters, it subtracts these values from the old filters recovered from the LastData structure and saves the result again inside the LastData structure.

After exiting the Conv_Backward function, the control is returned to the Backward function, then to the Train function, up to the CNN.m script, which exhibits the new percentual accuracy. Its inner loop resumes, and after its outer loop resumes. THE END.