Abstract

The algorithms that I decided to implement, experiment with, and compare was a Convolutional Neural Network and a multi-layered Neural Network. In the end, the Convolutional Neural Network was able to obtain the lowest error rate of 2.45%, using 3x3 filters along with pooling and a 2-layer Neural Network. The minimum error rate achieved by a 2-layer Neural Network alone was a 9% and the minimum error rate achieved by a 3-layer Neural Network alone was a 6%. Within the Convolutional Neural Network, it was determined that including pooling layers was equally to more effective than not including a pooling layer, and that 3x3 kernels were slightly more effective at classification than 5x5 filters. Within the Neural Network, it was determined that classification accuracy generally went up as more layers were added, and up to 6 layers were tested. It was also concluded that increasing the number of layers of the neural network at the end does not have a significant impact on the end results. Results aimed at comparing algorithm parameters are based on short term (not long enough for over fitting to become a problem), and are potentially not representative of the highest accuracy attainable if given more time to train.

Theory

The multi-layer neural network is based on the neurons of the brain. The initial layer takes in information and assigns weights to each of the inputs according to how important each combination of features is at classifying its targets. Each layer of weights outputs to the next layer of weights to make sense of what which combination of combination of features is important, and so on, until the final layer is reached. The weights are initialized randomly at first, but learn their optimal values overtime. The activation function used for each neuron in this case sigmoid, though this can be changed depending on the situation. This network is one of the more fundamental and versatile types of networks out there.

The Convolutional Neural Network attempts to retain the orientational relationships of the pixels in an image. By assigning outputs to become the result of multiple inputs within proximity of each other through filters, this algorithm is able to factor in an input's orientation to one another as well. Each filter initially is set up with random values, though with each training iteration, the filters will learn image features that result in the least error when classifying images. The results of these filters at each location is outputted as the intensity values of another image to be filtered. Each filter outputs their own image, and all the images stacked together all become an input for the next layer of filters, and so on. With each layer of kernels, more complicated features, based on the orientation of simpler features will be found before it. To give an oversimplified example

for the sake of intuition, if one of the main distinguishing features of a target image is that it contains squares, in the first layer, one of the filters may learn to find horizontal lines, and another vertical lines as shown.

	1	1	
1			1
1			1
	1	1	

Horizontal Line							
1	1	1			Hline	Hline	
0	0	0					
-1	-1	-1					
					Hline	Hline	
Vertical Line							
-1	0	1		VLine			VLine
-1	0	1		VLine			VLine
-1	0	1					

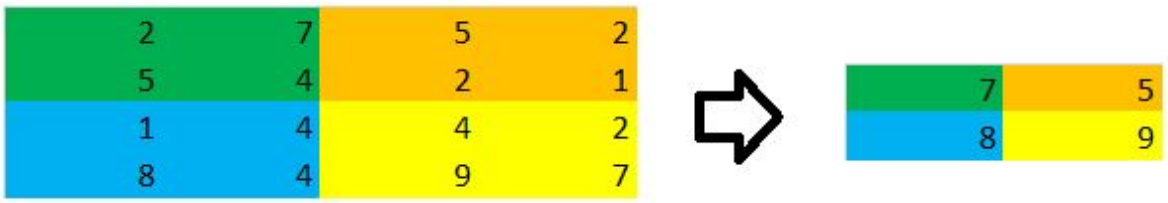
Then a filter on the second layer whose job was learned to find corners would find locations where horizontal and vertical edge outputs are high and are in proximity to each other.

Upper Left corner		
	Hline	Hline
Vline		
Vline		

A filter on the third layer, whose learned job is to find squares would then find where a four corners and four lines are orientated at, and so on. (ULC = upper left corner, LRC = lower right corner, etc)

Square		
ULC	Hline	URC
Vline		Vline
LLC	Hline	LRC

Though spatial retention is beneficial, it may at times work against classification at times due features being indicated at strict pixels locations. In most cases, distinguishing features won't always appear at the exact same pixel locations, and may vary slightly in location with each sample image. To help remedy this, one popular feature featured in many convolutional neural networks are max pooling layers. These layers will section the image into different sections, and within them, preserve only the largest values.



Through this, a significant feature will provide a more predictable and consistent feedback as long as it appears in generally the same location, removing emphasis away exact pixel location. Pooling layers also have the beneficial effect of reducing the memory space necessary to classify an image. When applied to an image, usually a default 2x2 pooling layer will cut the image size down to ¼ its original size, while keeping the most significant features relevant.

At the end of the convolution layers, a 2 layer neural network is set up to decide which target values contain which combination of complex. The activation used in the convolution layers is the leaky relu, which will multiply any negative output by .001 in order to weaken the influence that they have on the result.

The way in which the weights learn how to distinguish significant features is through backwards propagation, a method which tracks the output error caused down to each individual weight and updates them accordingly in an attempt to reduce it output error by as much as possible. By encouraging weights that help it get closer to the target result, and discouraging weights that get it further, the weights eventually adjust to pick up features with significant impact. The change of a given weight by back propagation is the chain of weight multiplications and activation functions that it takes for the output of a weight to reach the final output of the system.

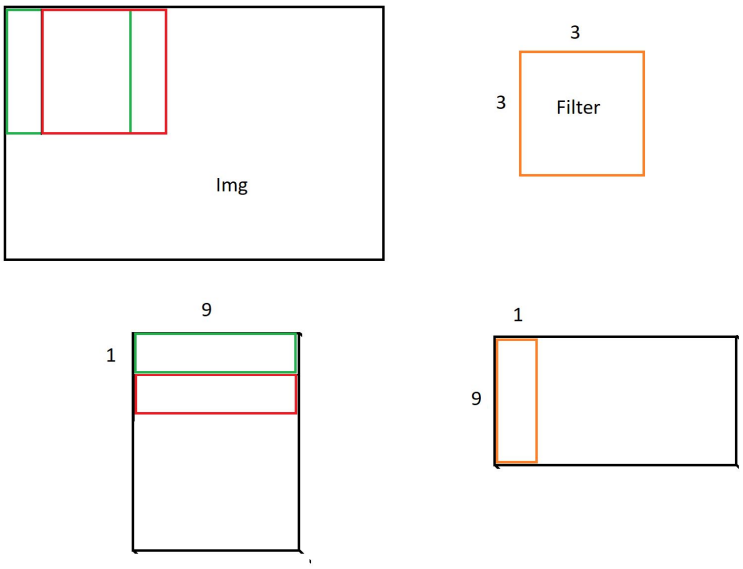
All algorithms used also use the Adam algorithm to adjust the learning rates of all their weights.

Implementation

Forward Propagation

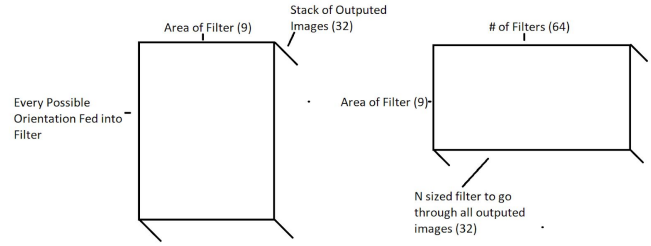
Convolutional Layers

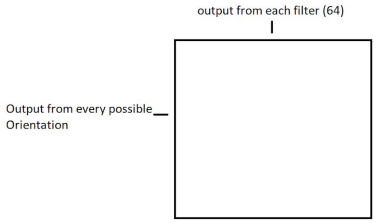
The kernel filters are first flattened to keep dimensionality as low as possible. The input is also prearranged to cover every possible kernel orientation.

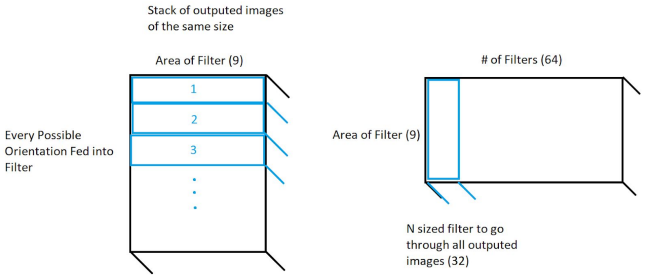


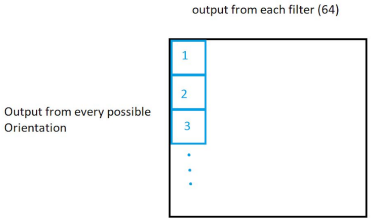
Though taxing in memory, as this multiplies the original image space by around the size of the kernel, this format also proves useful when keeping track of back propagation and enables for easier to manage vectorized operations. Each filter is then convoluted with the entire image by doing an element wise multiplication and summation at each location, each producing their own flattened output image. These flattened outputs are then fed to the next layer after the leaky relu function is applied, and the process is repeated with each subsequent convolutional layer.

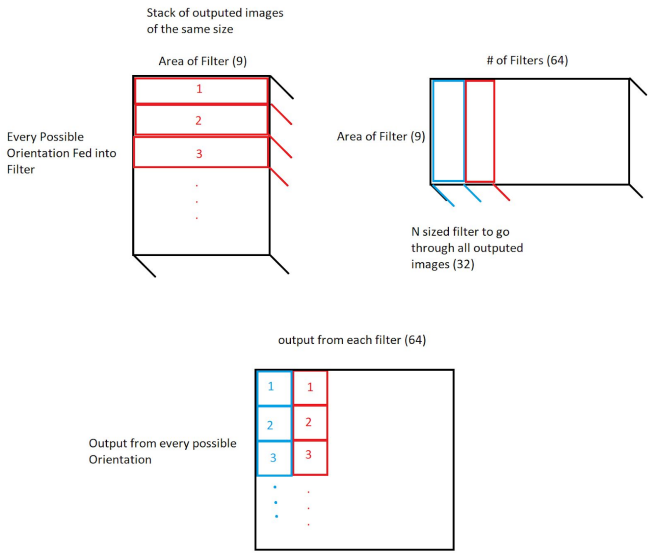
Example of calculating the convolution output of 64 3x3x32 filters, after receiving from 32 3x3 filters.


Area of Filter (9)
Stack of Outputed Images (32)
Every Possible Orientation Fed into Filter
# of Filters (64)
Area of Filter (9)
N sized filter to go through all outputed images (32)


output from each filter (64)
Output from every possible Orientation


Stack of outputed images of the same size
Area of Filter (9)
1
2
3
Every Possible Orientation Fed into Filter
# of Filters (64)
Area of Filter (9)
N sized filter to go through all outputed images (32)


output from each filter (64)
1
2
3
Output from every possible Orientation



Padding

In the case of padding, the input is divided into numerous segments without overlap and each segment is searched for its maximum. After all maximums are found, a new input for the next layer is created, and all the non maximums are set to 0 in order to prevent non contributors from being involved during backward propagation.

Neural Network

Each neuron receives output equal to the sigmoid of each input, and it's associated weight to the neuron. These outputs then becomes input for the next layer until the final layer is reached.

Backwards propagation

In order to calculate the update to a weight, 3 variables are needed. The derivative of the error with respect to the output, the derivative of the output with respect to the input, and the derivative of the input with respect to the weight. Of these 3 variables, only 2 can be calculated instantly at a given weight, while the third needs to be passed down from the layers after it.

Dir(): derivative of ()

Err : error at output, Out: output, In: input

wrt : with respect to

dir(Err wrt Out) = Err*(chain of dir(output to next layer input) and dir(next layer's activation function) until output is reached)

dir(output to next layer input) = weights leading to next layer

dir(Out wrt In)= dir(current activation function)

dir(In wrt Weight) = output from last layer

This leaves us with the more simplified formula

Change in weight = dir(Err wrt Out) * dir(current activation function) * output from last layer,

Where dir(Err wrt Out) = Err*(chain of next layer weights and dir(next layer's activation function) until output is reached)

Neural network

Change in weight = dir(Err wrt Out) * dir(sigmoid) * output from last layer

Initial dir(Err wrt Out) = error

Successive dir(Err wrt Out) = dir(Err wrt Out) * next layer weight * dir(next layer sigmoid)

Note: though this should be correct, when experimenting with the last layer's dir(Err wrt Out), excluding the dir(next layer sigmoid) from the calculation resulted in a much more rapid initial learning rate than when it was included. Though perhaps in the long term, the derived formula could have the potential to provide a more stable and consistent descent, and could potentially surpass my current results with enough time, the dir(next layer sigmoid) was forgoed on the second to last layer for the sake of a more quicker training session. It is also interesting to note that in my experimenting, it is necessary for the first neural network layer that connects the convolutional neural network to pass down the dir(next layer sigmoid) for the dir(Err wrt Out), otherwise it gets stuck at around an accuracy rate of about 10%. Also, including or excluding the dir(next layer sigmoid) part from all the layers in between the first and second to last layer of the neural network seemed to cause no difference regarding end results or learning rate. Overall from my findings, the neural network requires the last layer to ignore the the dir(next layer sigmoid), while also requiring that first layer doesn't in order for it to learn as effectively as possible.

Convolutional Neural network

Change in weight = dir(Err wrt Out) * dir(relu) * output from last layer

Since the convolutional neural network isn't directly connected to the output, it doesn't need to set the the initial value of dir(Err wrt Out)

Successive dir(Err wrt Out) = dir(Err wrt Out) * next layer weight * dir(next layer relu)

However,

dir(relu) = 1

This simplifies change in weight formula down to

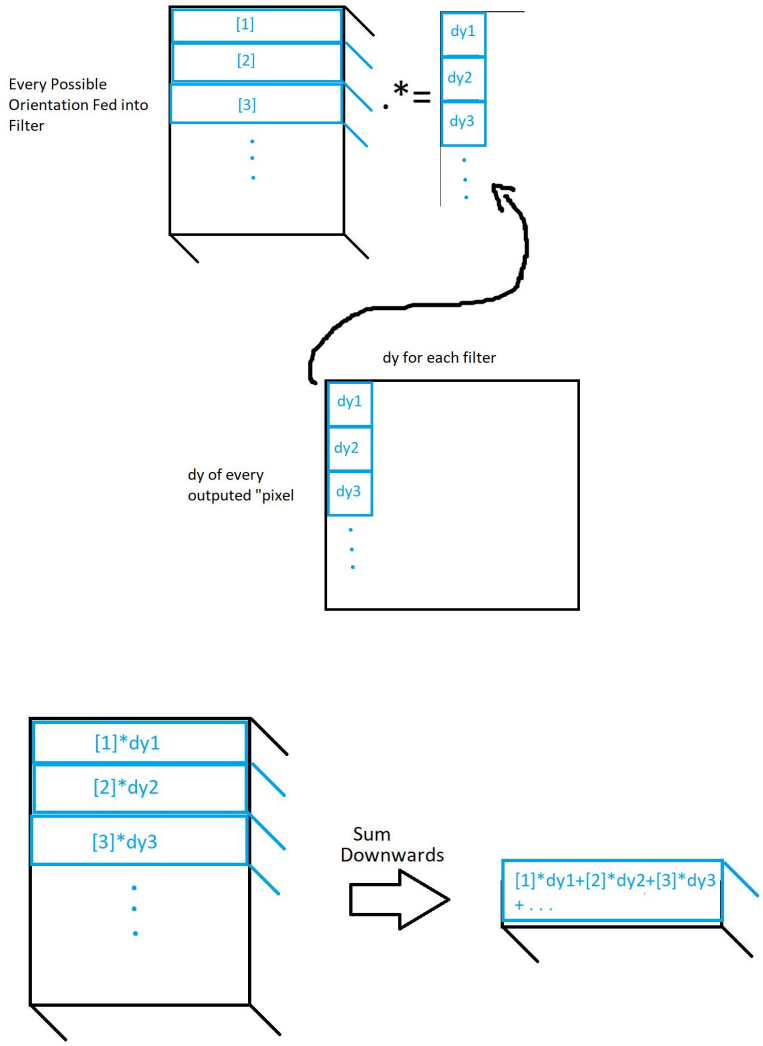
Change in weight = dir(Err wrt Out) * output from last layer

Successive dir(Err wrt Out) = dir(Err wrt Out) * next layer weight

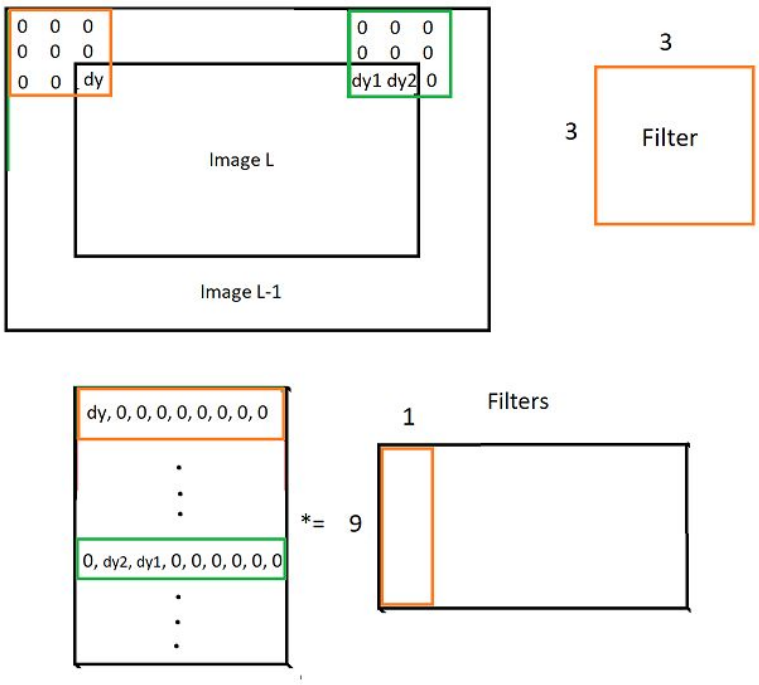
Unlike the normal neural network because the convolutional neural network has outputs with multiple inputs, it is important to note how to map these outputs and respective inputs together.

From the stored input of the forward propagation, every output from the last layer encountered per location is already organized by associated weight. The resulting dir(Err wrt Out) from the layers afterwards line up as well.

From here, doing an element wise multiplication and then taking the sum of each column will result in the desired change in weight.



To calculate the dir(Err wrt Out) for the next layer to use, the weights associated with each output can be taken by doing a sort of reverse convolution with the current dir(Err wrt Out), or dy. By padding the output to be of the same size of the input, you can observe the orientation of each weight at the time each dy1 was calculated from the weights perspective, which would be the opposite side of where the weight actually is. Thus, the kernel order is reversed in order to reflect this.



Applying this reverse convolution algorithm, and then matrix multiplying with the filter will produce the dir(Err wrt Out) to be passed on to the next layer. This is repeated with every filter, and it's associated dir(Err wrt Out) as well.

Pooling

Because all the unpooled values have been zeroed, their lack of contribution to the error is accounted for. All that is left is to reverse the forward propagation algorithm the make fit it's original format.

Reproducibility

Reproducing the results on this report is rather simple with the program. The main function, FinalProj(layersK, layersN), takes two arrays as it's input. The first array, layersK , is a 2D array whose size is based on how many convolution layers and pooling layers there are. The format for adding a convolutional is as follows: [(filter height) (filter length) (filter number)]. For example, to add a layer that involves an input using 32, 3x3 filters, you would enter [3 3 32]. Note that the program only works with square filters, and it is important that the filter height and filter length are the same size to avoid error. The format for adding a max pooling layer is as follows: [0 (pooling height) (pooling width)]. For example, to add a pooling layer, you would enter [0 2 2]. If the pooling chosen doesn't fit the image to be pooled (even sized pooling on odd sized image) then the program will automatically decrement the pooling height or width that doesn't fit until it does. To combine multiple convolutional and pooling layers together, you would stack them

from top to bottom in the order that appear in the form of a 2D array. For example, creating a convolutional neural network with an initial layer of 32 5x5 filters, a 2x2 max pooling layer afterwards, and a final convolutional layer of 64 5x5 filters, [5 5 32; 0 2 2; 5 5 64] would be inputed.

The second array, layersN, is a 1x2 sized array that determines the properties that the neural network at the end of the convolutional network will have. The format for specifying ending neural network goes as follows: [(number of nodes per layer) (number of layers)]. For example, if you wanted the network to contain a neural network with 6 layers, each consisting of 4096 nodes, you would input [4096, 6]. Note that while you can enter an empty array as the layersK if you only want the network to contain a neural network, the network will always default to a 2 layer neural network if a lower number is specified in order to have enough layers to accommodate convolutional layer.

Results

For my results, with regards the input format, [3 3 32; 3 3 64; 0 2 2] performed the best with an error rate of .0245, in a time of 58378.446598 seconds and using (51.01 G allocations: 83.752 TiB, 10.96 gc time) From a 2 layer neural network, I got an error rate of .0793, within 809.634 seconds and using (101.99 M allocationsL 607.272 GiB, 47.09% gc time). From a 3 layer neural network, I got an error rate of .0672, within 19469 seconds and 720 training iteration.

Training was interrupted if the average error of the past 8 validations surpassed the average error of the 8 to 16 past validations.

The best convolutional neural network format was based upon training using the same criteria above, except the up to the past 5 validations. With this criteria, [3 3 32; 3 3 64; 0 2 2] scored an error rate .0323, [3 3 32; 3 3 64] scored an error rate of .0396, and [5 5 32; 0 2 2; 5 5 64] an error rate of .0433

With 6 layers, a normal neural network reached an error rate of .0501, the lowest of that algorithm given the limited criteria