Image Classification using CNN

Emma Cai, Aaron Leslie

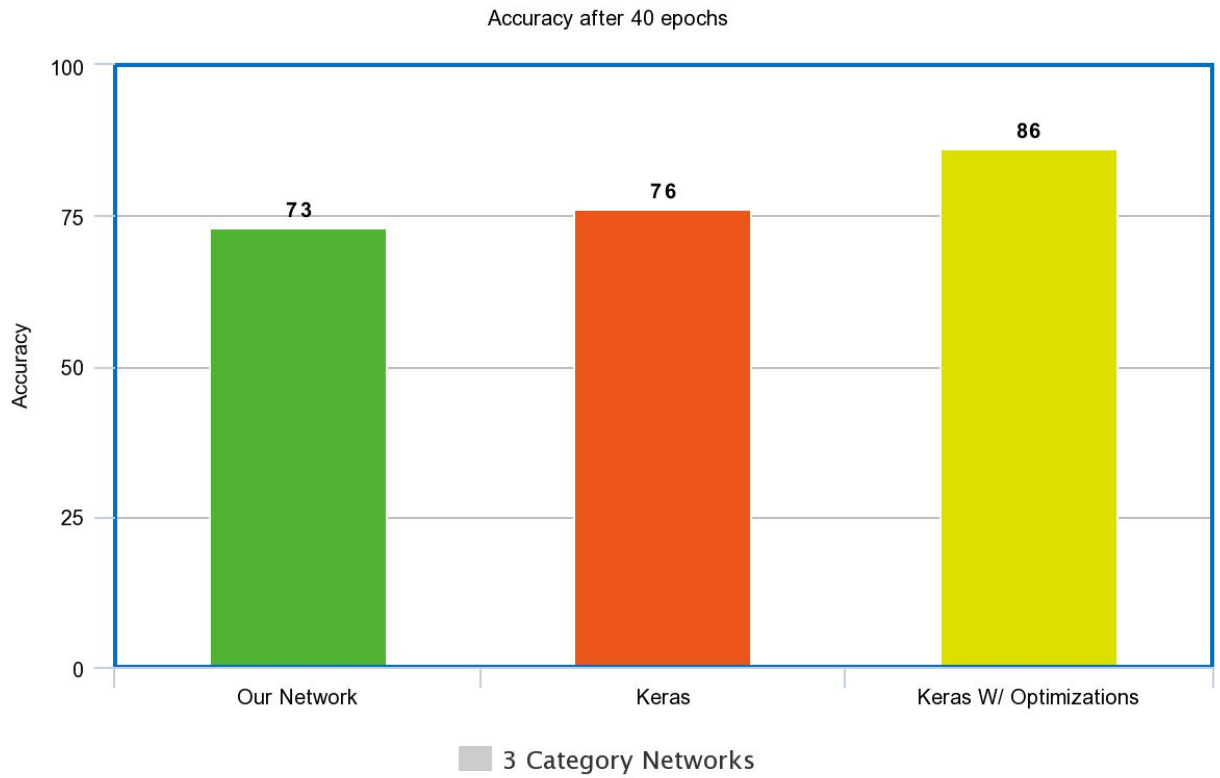
In our project, we designed and implemented a convolutional neural network to classify 32x32 RGB images as one of “Airplane”, “Bird”, or “Ship”. We used a subset of the existing labeled and organized dataset CIFAR-10, which contains 60,000 32x32 RGB images labeled each as one of ten categories. In total, we had 5,000 training images for each of our classifications, and 1,000 testing images. Our goal was to achieve a meaningful accuracy over the provided test data.

A convolutional neural network models human thought processes by seeking to process images firstly in a way that is thought to be similar to how humans do, using filters. Filters can be as simple as isolating edges in an image, to reducing the existence of a wing or head down to one or two pixels worth of data. The meaningfully summarized version of the image, after passing through several convolutional layers, is then fed into a traditional artificial neural network, which is able to categorize its inputs based on discovered patterns. For example, if our filters found two well defined triangles, our artificial neural network may be likely to classify our image as an airplane or a bird.

The major challenges for this project were likely to be both the complexity of implementing and tuning the hyper parameters for a convolutional neural network, and the amount of time required to train a convolutional neural network that is not well optimized for available hardware. We set out with no intention of facilitating any form of GPU acceleration nor multi-threading. This meant that time taken to train would massively impact our ability to tune the hyper parameters, as feedback would be slow.

Ultimately, we settled on a network with the following structure: 32 filter 3x3 stride 1 convolutional layer, leaky relu activation layer, 64 filter 5x5 stride 2 convolutional layer, leaky relu activation layer, 2x2 max pooling layer, 128 neuron densely connected layer, leaky relu activation layer, 3 neuron densely connected output layer, softmax activation layer. This network was acknowledged to be more limited than ideal, but computation and training time was a critical factor in our decision.

We used simple mean squared error for calculating loss for back propagation, and stochastic gradient descent with momentum for the back propagation itself. We used a momentum of 0.7, and a learning rate of 4 x 10^-6, with a 4% decay every epoch, and batch sizes of 64. Our network trained for a total of 40 epochs over the training data, before validating with an accuracy of 73% (2181/3000) on the unseen test data.



An implementation of a network with identical hyper parameters using Keras achieved 75.8% (2274/3000) accuracy, proving the core logic of our program to be accurate. Simple stochastic gradient descent, versus more modern optimization algorithms like RMSProp and adam, performs rather poorly, but does eventually reach meaningful results. Furthermore, mean squared error, versus categorical cross-entropy is also below ideal. For training a larger network on a harder task, more modern algorithms and optimizations such as the aforementioned would be necessary. For example, a network with the same layers, but using adam and categorical cross-entropy in Keras achieved 86% (2574/3000) accuracy after 40 epochs.

Even further optimizations could be made in the way of image pre-processing. Such as normalizing the average luminance of the images, training on rotations, reflections, and small translations of the original training set. And of course, optimizations in the way of computing time could be made with multi-threading or GPU acceleration, as our implementation is bound to a single CPU thread, and our math is done with numpy array manipulation.

Overall, we are satisfied with the results our network achieved in this problem space. Our final results are comparable to those of other off the shelf open source solutions and achieved a meaningful level of accuracy. While not competitive in terms of processing time (a single epoch would take a few hours on an intel i7-6700k on our network, versus ~20 seconds in Keras), we demonstrated an accurate and fully functional implementation of the intended technologies.

Given more time, the most effective optimizations would not necessarily be algorithm changes, but rather processing efficiency changes, such as manipulating arrays in more efficient ways, and ensuring support for GPU acceleration or at least multi-threaded batching.

Furthermore, more precise tuning of the number of layers and filters to better serve our specific problem space would be ideal.

We entered into this project with relatively little understanding of machine learning beyond simple neural networks, and implementing a convolutional neural network from the ground up not only gave us insight into the underlying logic and math behind them, but also some insight into where headway has been made in more modern advancements in convolutional neural networks.

Notes:

Code for loading CIFAR-10 data set sourced from University of Toronto's website. This is just to load data of their specific format https://www.cs.toronto.edu/~kriz/cifar.html

Code for converting CIFAR-10 data into 3 channel arrays sourced from Magnus Erik Hvass Pedersen. This just converts data into a more useful format for input into a CNN

https://github.com/Hvass-Labs/TensorFlow-Tutorials/blob/master/cifar10.py

Matrix math all implemented using numpy