Convolutional Neural Network Image Classification

Ashton Hunger

CS410

Abstract

Image classification is a well known challenge in visual computing. In recent modern history a new solution has emerged using neural networks, contrasting previous situational algorithms. Convolutional Neural Networks (CNN) are a specialized type of neural network for interpreting and generalizing spatial data. The following paper implements a CNN inspired by popular architectures AlexNet, GoogleNet, and various details from other sources. Training and testing the against the Cifar-10 dataset, the architecture outlined in the paper evaluated with a Top 1 accuracy of 83.6%, a Top 3 accuracy of 96.8% and Top 5 accuracy of 99.1%. Based on training loss graphs, it is implied with more training time the scores would have improved, but the proven results are as expected with the architecture used.

Introduction

Image classification is a difficult challenge in visual computing. In the past, solutions had been built using explicit algorithms which were designed for particular image classes, but never produced fruitful results. In recent modern history a new solution has emerged using neural networks. Rather than designing an algorithm for a particular class, neural networks, built on the concept of convolution filters, implicitly learn the features which compose the each class in a set. The topic of this project 2 report is implementing a convolutional neural network(CNN) to classify the cifar-10 dataset.

Setup

- Install Python 3.53x64 (https://www.python.org/downloads/release/python-353/)
- Install TensorFlow with the command (pip install tensorflow)
    - Make sure it is installed to the 3.53x64 Python Interpreter
- Download the source code (https://github.com/ahunger603/CNNImageClassifier)
- Download the Cifar-10 Binary Dataset (https://www.cs.toronto.edu/~kriz/cifar.html)
    - Place cifar-10-binary.tar.gz into folder (<directory>/data/cifar10_data)

Use Instructions

The following section explains each Python file, and how they're used. To start, "CNN_train.py" is a support file and is used to extract the image information from the binary into useful numpy array format. Next, "CNN_model.py" defines the model computational graph used by TensorFlow. In the software, there are *three* entry points, train, evaluation and test. "CNN_train.py" must be run first (unless you use pre-made training data) to train the model, and create checkpoints to be used by evaluation and test. **Important!** If you want to use pre-made training data, copy all the checkpoint information from the folder "<directory>\data\trained_checkpoints" to the folder "<directory>\data\cifar10_train". "CNN_eval.py" is used to evaluate the model on images excluded from training, and can be run in parallel to training to keep track of the true accuracy. Finally, “CNN_test.py” can be used after training as a visualization of the evaluation, allowing the user to view inferences of individual images.

The Layers

The architecture used in the CNN has 4 fundamental layer types: Convolutional, Max-Pool, Average Pool, and Fully Connected. Convolutional layers are the core of CNN functionality. Each convolutional layer is composed of filters which each have an output activation layer. Convolutional layer filters are applied to each patch of the input layer, storing the result in the associated place in the activation layer. At construction the filters are initialized with 5e-2 truncated standard deviation random noise, and are given a multiplicative weight decay of 0.003. The initialized noise is to prevent filters from converging on similar features, and to kick start activation results. Padding for each layer is dynamically calculated to reduce dimensionality by the Stride, e.g. Stride 2 would reduce spatial data by 2, where Stride 1 would leave spatial data the same. For activation function, the convolutional layers use exponential linear unit functions. The multiplicative weight decay is to help alleviate overfitting. Convolutional layers also have a stride and padding which define how granular filters are calculated and the size of the activation layer. Max-Pooling layers are a method subsampling. Each Max-Pooling layer samples patches of a fixed size, only sampling the strongest activation in the set, outputting a layer *n* times smaller than the input, where *n* is the chosen dimension of the Max-pool patch size. Average pooling layers are functionally similar to Max-Pooling layers. Rather than sampling the strongest activation from each patch, Average pooling layers average the activation values in each patch, still reducing the output layer by the chosen dimension. Finally, fully-connected layers, which are conventional neural networks. As the name implies each neuron in the fully connected layer is connected fully to every activation in the input layer. The fully connected layers are each unique, so the associated variables will be described in the architecture.

The Architecture

Below is a detailed technical explanation of the architecture used in the project. For all layers the activation function is exponential linear unit and truncated standard deviation noise was used for bias initialization. All convolutional layers have a weight decay of 0.003. More specific layer details are displayed below.

Input -> Conv 1 -> Conv 2 -> Max Pool 1 -> Conv 3 -> Conv 4 -> Max Pool 2-> Avg. Pool -> FC1 -> FC2-> FC3

Fig. 1 Architecture layers and information flow

Layer	Input Dimensions	Filter Size	Stride	Features	Output Dimensions
Conv. 1	128x32x32x3	5x5	2	64	128x16x16x64
Conv. 2	128x16x16x64	5x5	1	64	128x16x16x64
Conv. 3	128x8x8x64	3x3	1	96	128x8x8x96
Conv. 4	128x8x8x96	3x3	1	128	128x8x8x128

Fig. 2 Convolution Layer Details

Layer	Input Dimensions	Pool Size	Stride	Output Dimensions
Max Pool 1	128x16x16x64	2x2	1	128x8x8x64
Max Pool 2	128x8x8x128	2x2	1	128x4x4x128
Average Pool	128x4x4x128	4x4	1	128x1x1x128

Fig. 3 Pooling Layer Details

Layer	Input Dimensions	Dropout	Weight Decay	Output Dimensions
FC1	128x128	40%	0.002	128x384
FC2	128x384	40%	0.002	128x192
FC3	128x192	0%	0.004	128x10

Fig. 4 Fully Connected Layer Details

Training

Training the model is done in over epochs in batches. Each batch is taken from the 50,000 image cifar-10 training set. Batches are 128 images each. Within each batch the images brightness and contrast are independently distorted, and randomly flipped horizontally to help prevent overfitting and synthetically increase the training set. Sets of batches are segmented into epochs, of 4000 examples. During training, checkpoints are created which store the network state to be used for evaluation. To learn the model uses basic gradient descent with an exponentially decaying learning rate. A learning rate of 0.15 is used which decays exponentially each 25 epochs. Loss for optimization is calculated using cross entropy. The model was trained for about 70,000 steps over 10 hours. By the end of the training, the model was obtaining an average of ~84% image inference accuracy per batch.

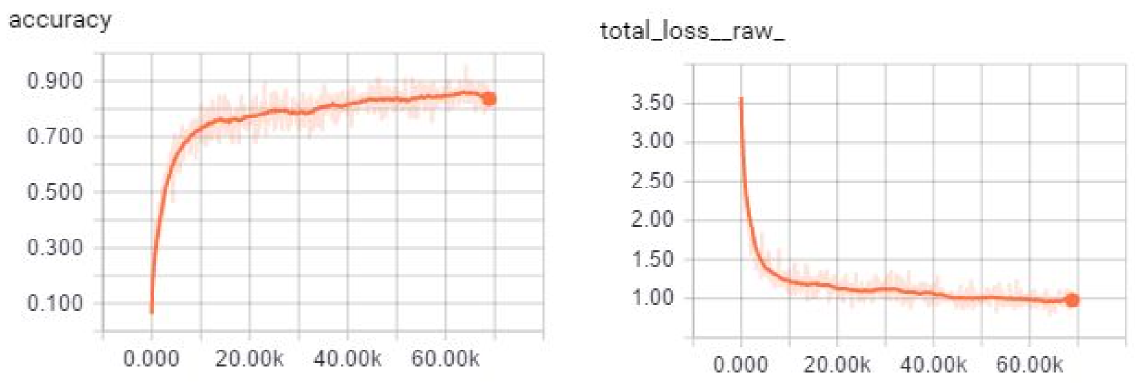

Fig. 5 Accuracy and loss during training

Evaluation

Evaluation is done with a completely separate set of 10,000 images, that has not been used for training. The reasoning behind keeping the evaluation set separate is to prove the model hasn’t overfit the training set, by learning features specific to the training set. While evaluating, dropout is removed. The results from evaluating the model were 83.6% image inference accuracy, 96.3% Top-3 accuracy, 99.1% Top-5 accuracy. A significant observation is the evaluation and training accuracy only have a difference of 0.4%, which shows how successful the regularization was. Below is examples of the display mode, which shows individual inference results.

Conclusion

Overall, the model performed as expected based on the source material used. Given more time, there is massive room for improvement, including basic changes like more training time. For the purposes of the project, 99.1% Top-5 accuracy is completely sufficient, and the practice using TensorFlow and understanding about neural networks was invaluable.

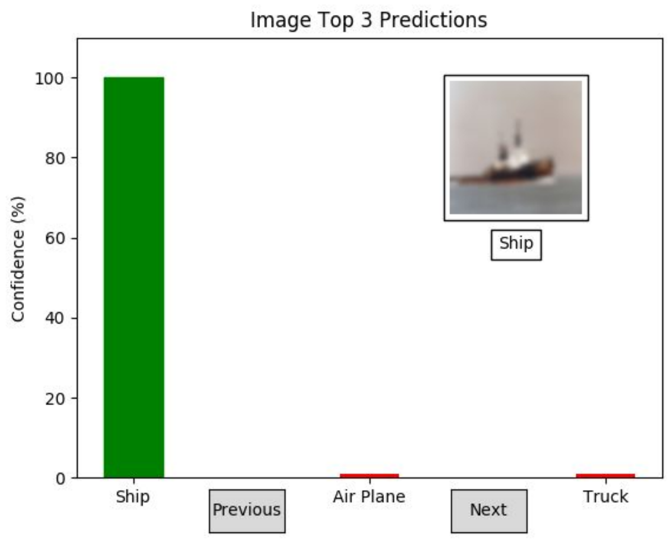

Fig. 6 Display Example

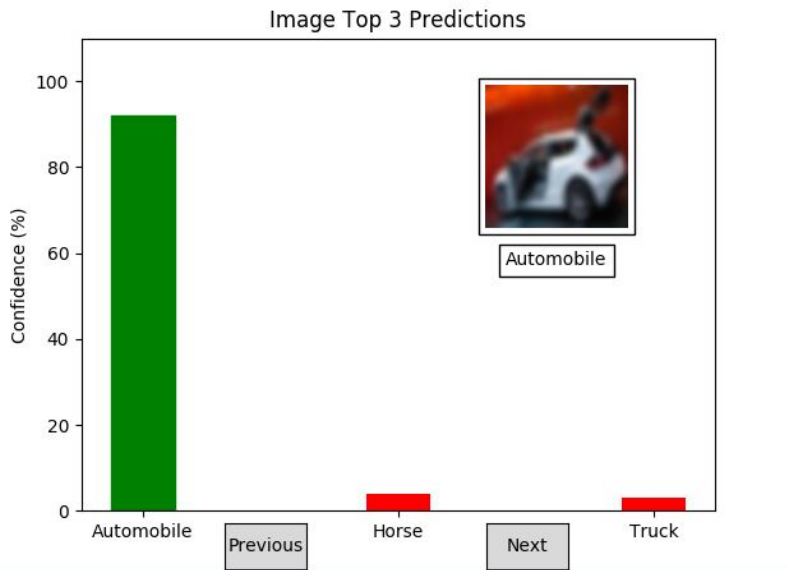

Fig. 7 Display Example 2