Deep Convolutional Neural Network Using Simulated Data

Austin Feydt

29 March 2018

Overview

Deep neural networks can be extremely powerful learners, as they are able to discover obscure features in data that may not be at all intuitive to humans. However, in order to be useful and discover these specific details, they often require a large data set to train on. In image recognition, a deep network will not be able to converge without thousands of unique images to learn from. This can become a large barrier for machine learning research, because if there is not already a database somewhere online, then one must gather their own data to train on. Data collection can be painstaking, especially image collection. Thus, we wish to explore automating the data collection process. To do this, we will gather all of our images in simulation, rather than in the physical world. We hope that the images gathered in simulation will be ”good enough” to develop a competent neural network.

Background

The bulk of this project involves the data collection/preprocessing of the data, which is pretty much undocumented. However, I wanted to at least recall the function of convolutional neural networks, and explore a famous network using convolutions. Convolutional neural networks are similar to a traditional neural network, as they also involve neurons with corresponding weights and biases, which are trained via backpropagation. However, convolutional neural networks are specifically used with the intended input of images, and utilize the structure of images. As opposed to traditional network inputs, images are not just matrices of pixel values. The ordering of the pixels and how they change in value across the image are extremely important to the image. Convolutions take advantage of this structure by using immutable filters on images. These filters (also known as kernels), which are usually 3x3 pixels to 5x5 pixels large, are ”slid” across the image, and the weighted sum of the pixels under the filter are calculated, creating a new image. By choosing different kernels, we can discover different features in our input image. As we

can see in Figure 1, kernels can be used to detect edges in images [1].

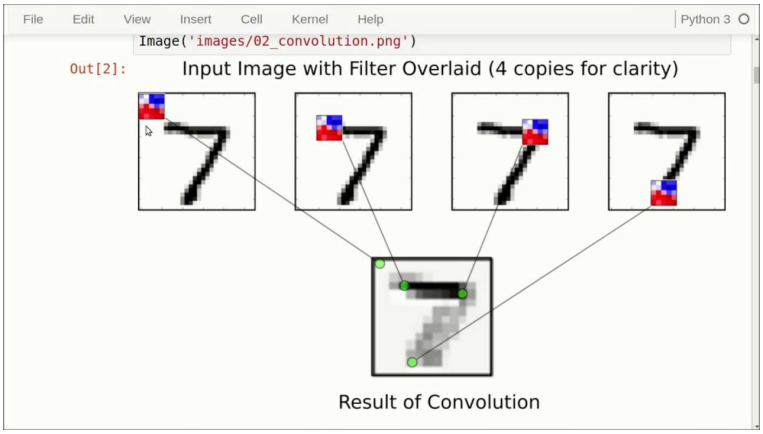

Figure 1: Example of a horizontal edge detector

One of the first successful convolutional neural networks (CNNs) was the LeNet, developed by machine learning researcher Yann LeCun in the 1990's. LeNet was a ground-breaking network, able to recognize hand-written digits and letters with incredible precision. The structure of it can be seen in Figure 2:

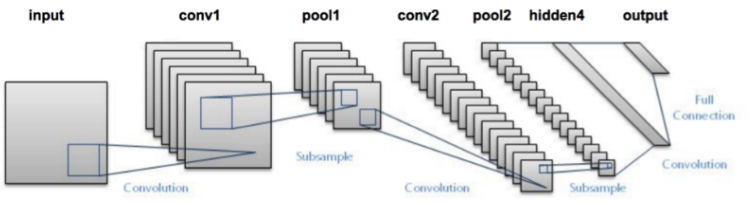

Figure 2: The famous LeNet

To dissect the structure of LeNet, we begin with the first layer. "conv1" is a layer of convolutions, each by a different kernel. This generates n new images, all of the same dimension. Next, in order to reduce dimensionality and avoid possible overfitting, the "pool1" layer is introduced. The most common form of pooling is 2x2 max pooling, in which a 2x2 grid is slid along the image. Within this grid, the pixel with the largest value is kept, and the other 3 pixels are thrown away. This generates a new image half the width and height of the orignal image. Now, when the next convolution "conv2" is applied, it is able to detect more basic features, as now all of our images only include the strongest features of the previous convolution. "pool2" performs yet another 2x2 max pool, followed by more convolutions, and then a traditional fully-connected layer leading to the output.[1]. This network has inspired many more complex CNNs that are used in industry today. Our network will be a variation on the original LeNet,

using more convolutions, and introducing a flattening operation on the images before the fully-connected portion.

Data Collection Process

As mentioned earlier, the goal of this project is to collect all data in a computer simulation before feeding it into a Deep CNN. The process for collecting data is as follows:

1. .stl files for various objects were obtained via www.grabcad.com These files represent everday objects, like basketballs, staplers, etc. Each .stl file was then loaded into MeshLab. Once loaded into the program, they were scaled to appropriate, ”real life” dimensions. Then, their mesh files were extracted to be used in simulation.

2. For each mesh, an .sdf file was created for them. These .sdf files are what are actually spawned into the simulation. They define the physical properties of the objects (dimension, texture, color, mass, inertial properties, etc.), and are linked to the meshes. Also, a .launch file was created for each .sdf file. This is what is used in the terminal to spawn the corresponding object into the simulation. All of the meshes, .sdf files, and launch files can be found in the simulation folder.

3. ROS was installed on the machine and a ROS workspace was created under the simulation folder. Once correctly configured, an empty ROS world was launched in gazebo using the following command:

   roslaunch gazebo_ros empty_world.launch

4. In another terminal, a table and kinect camera were installed using the following commands:

   roslaunch image_recognition add_table.launch

   roslaunch image_recognition add_kinect.launch

5. Now, we can begin the automated data collection. A shell script was created, entitled ”collect_data.sh” (in the image_work directory). This file is called in the terminal. Once executed, it spawns one of the .sdf files into the gazebo simulation using a .launch file, the kinect takes a pointcloud image of the scene, it is saved to the machine, and then the object is deleted from the simulation. This can be run overnight to produce as many pictures as you want (although, be wary that pointcloud files are large, so your machine can run out of room quickly if you’re only working on a small Ubuntu partition like I am).

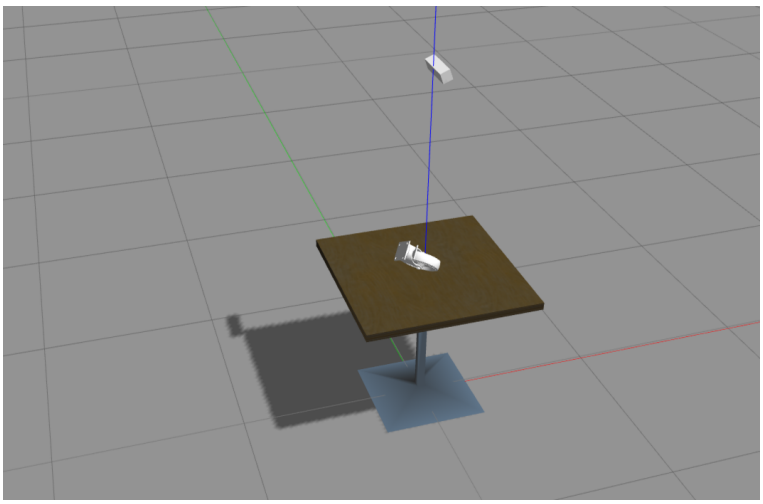

Figure 3: The simulation

6. Once all of the pointclouds have been created, they can be converted to .png files using an external library called ”pcl_pcd2png”, and then further cropped using ”crop.py” (both also found in the image_work directory). All of the images in the labeled folders in image _work are the cropped pngs.

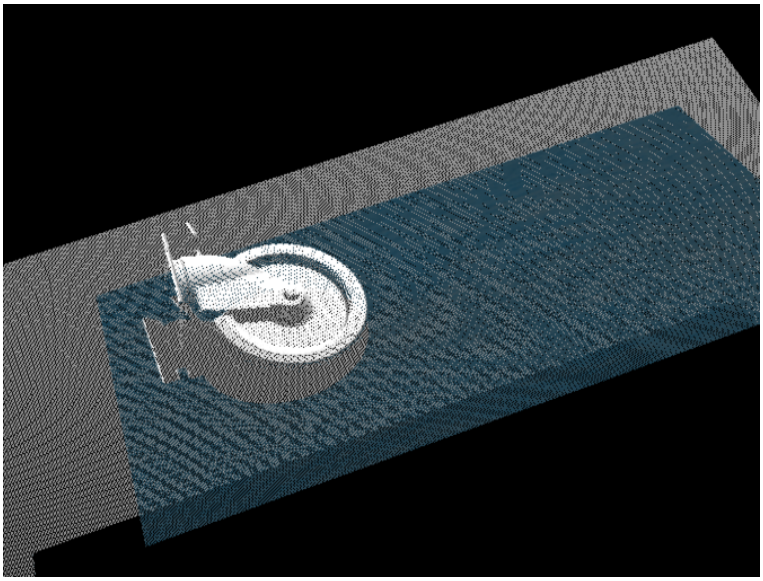

Figure 4: The pointcloud image

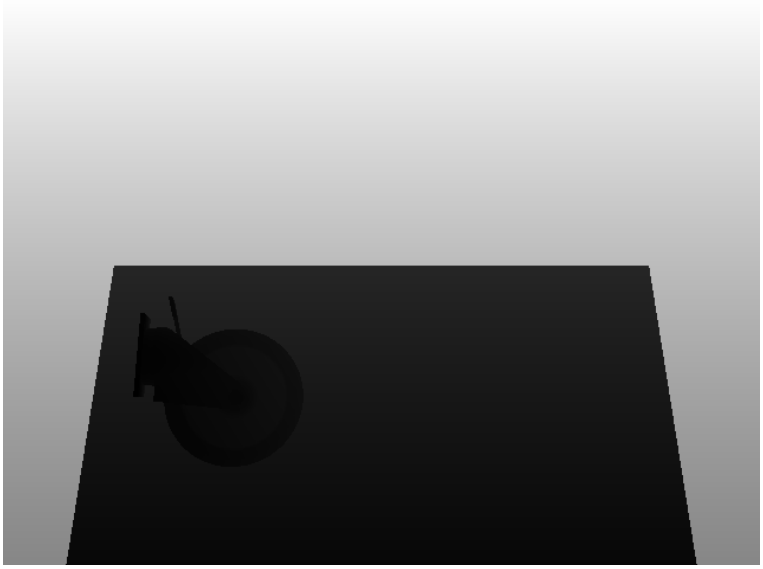

Figure 5: The cropped png

7. Now, we can further process the data by converting all images to jpegs and serializing them into .tfrecord files. This will reduce how much memory this program takes up, and also is convenient for reading into a tensorflow program, as it retains the labels of all the images within the .tfrecord. First, I created a label.txt file that lists all images to be included in this .tfrecord. I also found a python file build_image_data.py to automatically create separate training and validation records. These files can also be found in the image_work directory. Finally, we have our data neatly organized and ready to train!

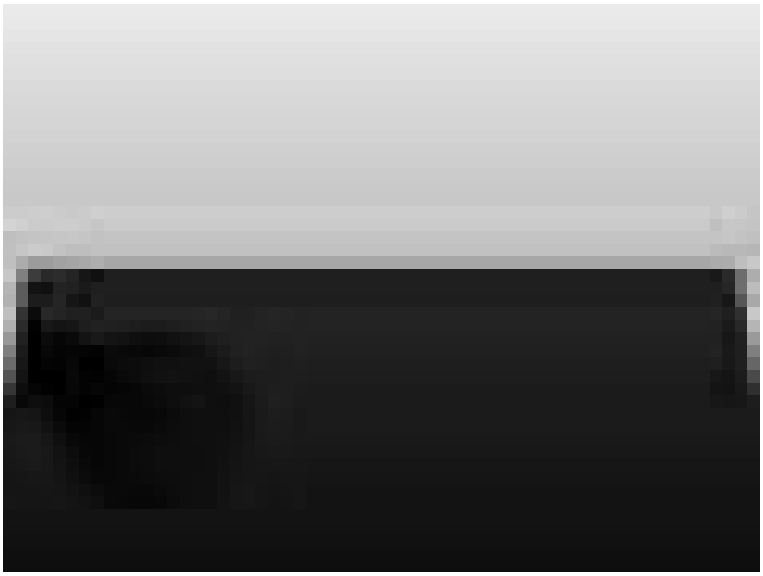

Figure 6: Low-res jpeg

The network structure I decided to go with is similar to LeNet, but with a few changes. I have 3 convolutional layers, fanning the image out from 32 to 64 to 96 convolutions, with a 2x2 maxpool between each layer. Then, the images are flattened, and fed through 2 fully connected layers, rather than LeNet's single layer. The output of the network are logits for each of the labels (the confidence that the image is classified as each label). The maximum of all the outputs is considered the predicted label of the object.

Results

My CNNs can be found in the learner subdirectory. The first learner I created was for 6 objects, and I trained for about 15 minutes. If you run "6obj_train.py", it will skip training and just evaluate the current network. If you wish to start the network from scratch, you must delete the outputs folder, which contains the checkpoints and hyperparameters of the network. You should see that the network is only about 66% accurate. If you look out the outputs of the evaluation, it is clear that some of the objects (golf ball, basketball, and coffee mug are always classified correctly. However, the other three objects don't seem to have been actually learned by the network. If I had more time on this assignment, I would've explored creating deeper convolutions in the earlier layers of the CNN, because I think the network was not able to find the correct features in those objects. The other network, "3obj_train.py" only trains on the basketball, golf ball, and coffee mug. This was much quicker to train, and yields almost perfect accuracy.

Conclusions

While brief, I believe that this report shows the value in investing more time in the data collection process. While it is not clear how well a network trained on simulated data would fair in the real world, it is definitely a useful strategy when tackling problems that may have never been explored before.

References

[1] Andrej Karpathy. Convolutional neural networks for visual recognition. *http://cs231n.github.io/convolutional-networks/*, 2017.