Computer Vision Anomaly Detection

Ted Jung

Jul 1, 2024

1 Introduction to the Problem

The given problem is that there are images that need classifying in two ways. First, there is the class, which talks about the object that is in the image, with 15 different classes. Then, there is also the state, which talks about the type of anomaly that could be in the image, such as bent-wire, hole, etc. There are 49 different states, and in total, there are 88 different labels that include both pieces of information for each image. An example of a label would be "tile-good" or "cable-bent_wire." The first part of the prompt is to distinguish between the classes, then the second part is to distinguish between "good" and "bad" images.

1.1 Dataset

The training dataset contains 4,277 images, and the testing dataset contains 2,154 images. All these images are in folders called train and test respectively, and my idea was to create a new folder that would contain in it folders corresponding to each label, to pre-distinguish the images from the training set. Looking back, this part may not be necessary, as we could call the CSV file every time to just read what the label would be, but the function ImageFolder makes it easy to make a dataset, so I left it. In my code, this would be the train_folder, with folders for every label in it.
One caveat that must be noted is that for this model to run on Google Colab, the folder of testing and training images must be locally available. The two ways I found was to either upload the folder as a zip file onto Google Colab directly or download the folder on Google Drive and reference a pathway to it as code. The first option I estimated to take somewhere between 3-5 days, and the second took about three hours, so that's what I did. If there is a better or more time-efficient way to reference the data, please feel free to let me know.

2 Model

To perform this task, I used a Convolutional Neural Network (CNN). This is because CNNs are good at extracting data from bigger matrices of set sizes, which is what images are. It can take certain features from images and match them to known values, which allows for a high accuracy when compared to other neural network models.

2.1 CNNs

CNNs have the same basic format of a neural network, with an input layer, an output layer, and some hidden layers that extract features of the inputs and work some magic to come up with a probability in the output. But before this can happen, CNNs have additional layers before that extract features from the input by using filters to slide over the input–in this case an image–and take note of certain features that are prevalent. Like shown in Figure 1, this part is the feature extraction, which includes other layers to introduce non-linearity and reduce overfitting as much as possible. The final "output" from this phase is flattened, then introduced into

defined fully connected layers (FC layers), which then acts like a typical neural network to provide the most likely classification.
Convolutional layers work by mostly maintaining spatial features, albeit downsized. There are a number of filters, each of which produces a feature map that it was specifically looking for. Depending on the filter size, stride size, and padding, the feature map for each kernel might look different. The filter size affects how finite the details in the input image could be detected, the stride size affects how fast the filter goes through the image (resulting in different qualities of feature maps), and padding (additional columns and rows of zeros on the borders) help the filters to reach even the edges of the image with the same sort of attention as the rest of the input.

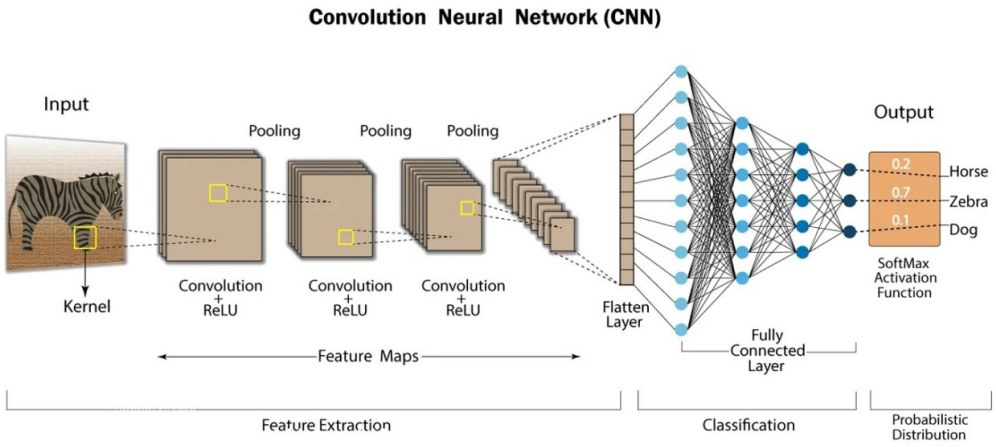

Figure 1: CNN Overview

In the other layers in the feature extraction, there are activation functions dispersed throughout. As shown in Figure 1, a basic format would be a Convolutional layer, followed by some sort of non-linearity (ReLU in this case), and then a pooling layer to downsample. This pattern will happen a couple of times, then flattened and given to a number of FC layers.

2.2 Implementation

The CNN code that I used was very simple, and in the future I would add much more layers and try different types of activations as well. I have a convolutional layer, a ReLU function, then a pooling layer as one "set," with two "sets." Then, I flattened the code into a length by one matrix, and used two Linear layers with a ReLU function in between.
For the first convolutional layer, I had three input channels (RGB values), and used eight filters in this layer, with a 5x5 filter size. In the second convolutional layer, I took in eight input channels (from the previous layer), and outputted 16 with the same sized filters (5x5). There was no real reason behind most of these numbers, and I just chose numbers that weren't too big or too small. Similarly, I chose a 2x2 size for the pooling layer, with a stride of 2. This is the most common, so I kept it.
For the FC layers, I chose simple Linear models, with the first layer going from the total number of individual values (16 * 253 * 253) to 150, then from 150 to the number of classifications. Again, the intermediate numbers were mostly randomly chosen, and would have to be tweaked for maximum optimization.

3 Code Overview

The overall form of models that utilize CNNs are pretty similar: loading the datasets, transforming the data so that it's suitable for training, building the model, training the model, then using it on the testing dataset.
My code included some additional parts. First, I used the ImageFolder function in PyTorch, which allowed me to create a dataset from sub-folders directly. As mentioned previously, it could have also been possible to do this directly from the train folder, but ImageFolder also works the same.
The next part that was also added was a way to normalize the data. To do this, I took all the images in the training dataset and found the mean and standard deviation, which I then used to normalize later when I created the training and testing datasets. For a loss function, I used CrossEntropyLoss, and for the optimizer function, I used Adam, an advanced version of gradient descent. Again, these are choices and I would have to try different optimization and loss functions to get the optimal result.