Feature Detection with Neural Networks in Image Processing

Deep Hathi, Jasper Mishra

CSE 415, Autumn 2012

12/12/2012

The goal of this project is to demonstrate a multi-layer perceptron network to differentiate and recognize chess pieces in a provided image. The network's input is a feature vector containing the heights of the objects in the image and the Hough Transform local maxima for each object. To get these parameters, the images were first converted to grayscale and scaled to default dimensions. A Canny edge detector utilizing the Sobel operator was then applied to find the outside boundaries of each individual object in the image. These boundaries were then used to segment the image into smaller images with individual objects in each image. The pre-processing workflow is summarized below in Figure 1.

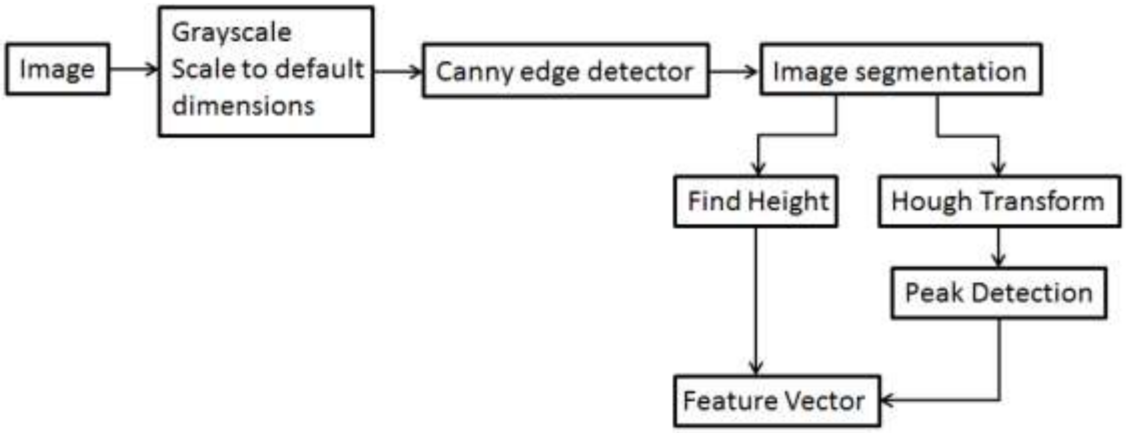

Figure 1: Summary of Image pre-processing

A multi-layer perceptron network was trained on a subset of training images using backpropagation. In this algorithm, the hidden layers apply a sigmoid threshold function. The output from each hidden node is propagated forward until reaching an output state. The output state is then compared to the expected output for both positive and negative classifications. The error is then calculated and propagated back through the perceptron layers using the delta rule (Equation 1). The delta rule modifies the weights at each perceptron to rectify the error based on the training rate η, which starts large and grows smaller with each successive iteration. The basic algorithm was adapted from the lecture slides and online resources [1].

$$\nabla w_{i,j} = \eta \delta_j F_j \tag{1}$$

Where $\nabla w_{i,j}$ is the change in weight for a given weight $w_{i,j}$, δ_j is the error between the computed output and the expected output, and F_j is the computed output given a training set X_i.

The error calculation is different for weights connecting to output nodes (Equation 2.1) or to hidden nodes (Equation 2.2).

$$\delta_j = (t_j - F_j)g'(h) \tag{2.1}$$
$$\delta_j = g_j'(h_j)\Sigma_k \delta_k w_{k,j} \tag{2.2}$$

Where t_j is the expected output.

In theory, training of the neural network is completed upon convergence of the expected and computed outputs. However, we were unsuccessful in getting the outputs to converge, resulting in termination of training upon reaching maximum iterations. The individual weights for the heights in the feature vector for each chess piece are given in Table 1. The weights were independent of the piece, thus showing convergence is not possible. This is worse for the Hough Transform local maxima.

Table 1: Weights for the heights in the feature vector

Piece	Height (normalized)	Weight
King	122	1
Queen	112	1
Bishop	101	1
Knight	87	1
Rook	77	1
Pawn	66	1

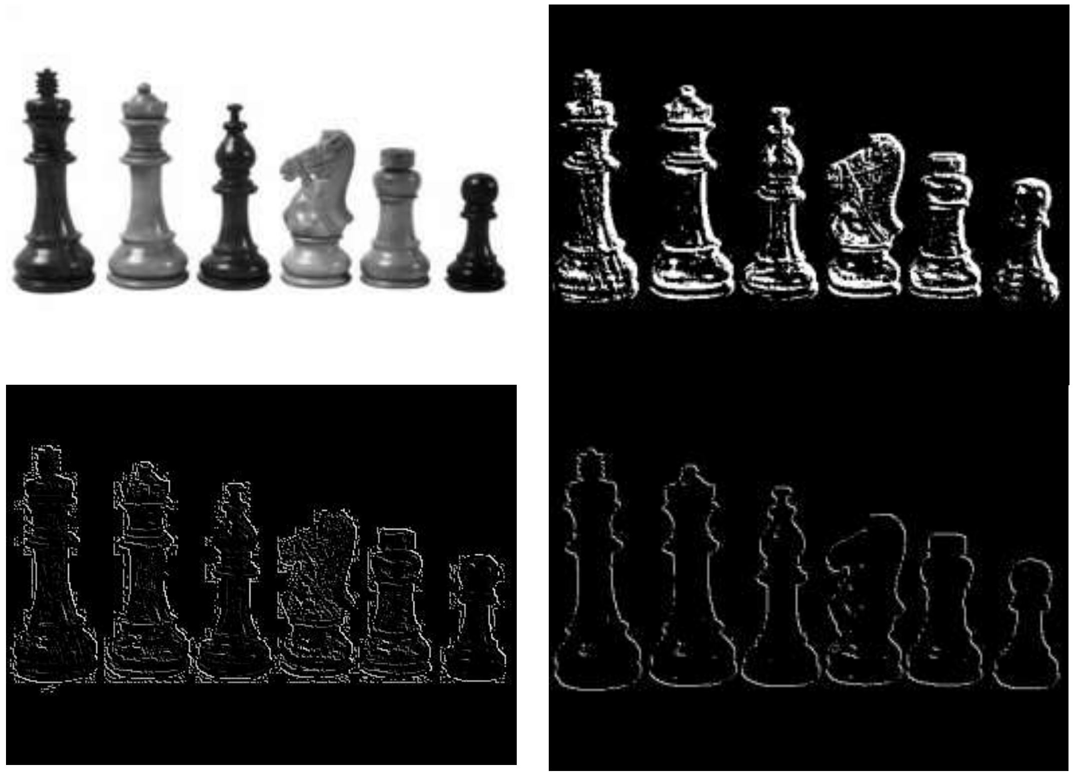

Figure 2: (top left) Original, grayscale image; (top right) Edge map with Sobel edge detector; (bottom left) Edges after non-maximal suppression; (bottom right) Outside boundaries shown after hysteresis

To run the program, trainImageSet.py is loaded into PixelMath and evaluated. The output shows error after every 100 iterations and finally, the weight for each height in the training set. The image file is specified in this python file.

An interesting part of the code was in the pre-processing. To segment the image, the image was passed through a Canny edge detector, which combines a Gaussian filter to smooth out speckle and random noise, a Sobel operator with non-maximal suppression, and hysteresis to remove extra edge effects [2]. We found the hysteresis operation quite interesting, since it applies a high and low threshold to remove extraneous edges. Grayscale values of a pixel that are above the high threshold are automatically accepted, while those that fall below the low threshold are removed. For the rest of the pixels, the code recurses through neighboring pixels to find a pixel that falls above the high threshold. If no such pixel is found, the original edge pixel is removed. This method has the dual advantage of reducing noise in the final edge set, while tracing paths in the edge image.

```
def hysteresis(edgeSet, highT, lowT,mag_ang):
  # traverses points in the edge set and checks to see if they are valid
  # Valid if: magnitude of gradient >= high threshold
  #         low threshold <= magnitude < high threshold; recurse through
  #         neighboring edges to see if they are valid
  # Invalid if: magnitude < low threshold
  edgeSet = sorted(set(edgeSet))
  output = []
  visited = []
  for (x,y) in edgeSet:
    if (x,y) not in visited:
      visited.append((x,y))
      mag, ang = mag_ang[(x,y)]
      if mag >= lowT:
        # valid points for consideration
        if mag >= highT:
          # point is a valid edge
          output.append((x,y))
        else:
          # check pixels nearby to see if edge is valid
          output = follow_hysteresis((x,y),visited,output,highT,lowT,\
                                mag_ang)
  return output

def follow_hysteresis(point, visited, output, highT, lowT, mag_ang):
  # recursive helper function for hysteresis
  magP, ang = mag_ang[point]
  x,y = point
  magSet = {}
  for mx in range(-1,2):
    for my in range(-1,2):
      if (x+mx,y+mx) not in visited:
        mag, ang = mag_ang[(x+mx,y+mx)]
        magSet[(x+mx,y+mx)] = mag

  for point in magSet:
    visited.append(point)
    if magSet[point] >= highT:
      # there is a valid edge next to (x,y) so (x,y) likely to be an edge
      # as well; point compared is also appended to the output set
      output.append((x,y))
```

```
        output.append(point)
        return output
      elif magSet[point] >= lowT:
        # recurse again
        return follow_hysteresis(point, visited, output,highT,lowT,mag_ang)
  return output
```

If more time were available, we would improve the project in both pre-processing and perceptron network algorithms. For the pre-processing, a generalized Hough Transform could be written, since this algorithm would provide a more robust classification method compared to the normalized heights and local maxima in the Hough space. In addition, a foreground/background differentiating algorithm could be used [3] to easily detect interesting features in a complex image containing other objects. For the perceptron training, finding a way to get convergence would be prioritized. Once this is accomplished, we would also provide a user interface to input images for testing.

References

[1] M. Bernacki and P. Włodarczyk. (2004, December 1, 2012). *Principles of training multi-layer neural network using backpropagation*. Available: http://galaxy.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html

[2] M. Sonka*, et al.*, "Image Processing, Analysis and Machine Vision. 1999. Brooks," ed: Cole Publishing Company.

[3] K. Kim*, et al.*, "Real-time foreground–background segmentation using codebook model," *Real-Time Imaging,* vol. 11, pp. 172-185, 2005.

Code provided in separate files