Ryan Farr

Richard Hill

November 25th, 2018

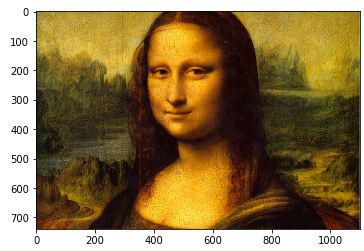
CS 5785 – Applied Machine Learning

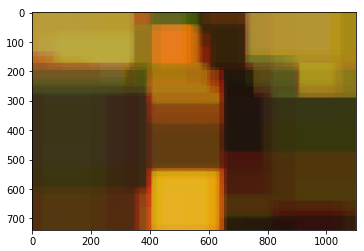
**Homework 4 Programming Exercises**

1. **Approximating images with neural networks**

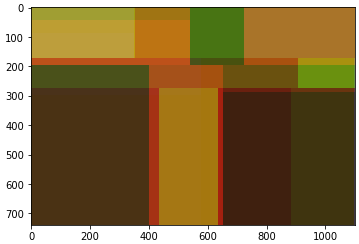
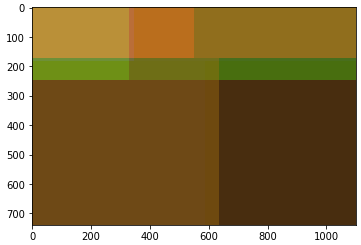
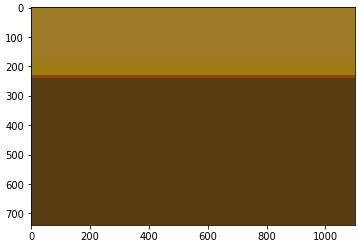
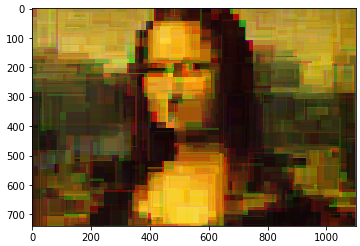
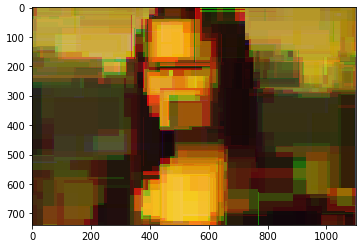
1. The default network has:
   * One input layer that takes 2 dimensional input points (x, y position on a grid)
   * 7 fully connected layers with 20 neurons with a ReLu activation.
   * One loss layer that uses L2 regression and 3 neurons that output R/G/B
2. The default network uses an L2 regression layer to calculate loss. It calculates the L2 distance between the output and the target values (referenced [convnet docs](mailto:https://cs.stanford.edu/people/karpathy/convnetjs/docs.html)).
3. The model seems to converge to a loss around 0.004
4. By halving the learning rate every 1,000 iterations, we were able to make the network converge to a lower loss function, around 0.0035
5. The performance of 4 and 5 layers is pretty similar to the performance of 7 layers. We can comment out about 2 layers before we see a noticeable quality drop.
   * With 7 layers, our loss converges to around 0.004
   * With 5 layers, our loss converges around 0.0043.
   * With 4 layers, our loss converges around 0.0045.
   * With 3 layers, our loss converges around 0.005.
   * With 2 layers, our loss converges around 0.01.
   * With 1 layer, our loss converges around 0.025.
6. We added 5 more layers (total of 12 layers). Performance was much slower but we did not see any improvement in terms of loss function over 7 layers. Loss still converged around 0.004.

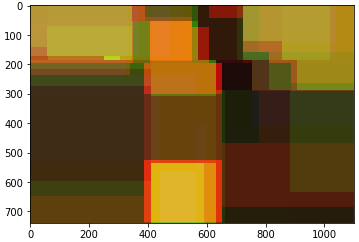
2. **Random forests for image approximation**

* + We’ve chosen to work with the following Mona Lisa image.
  + 5,000 points were randomly sampled from the original image. No preprocessing was deemed necessary; any subtraction or normalization only changes the value of the splitting points but not their relative position, which is to say that preprocessing these points shouldn’t change the output.
  + To process the output, we’ve chosen to learn three separate functions, one for each color. We opted not to do grayscale because We wanted the output to look as similar to the input as possible. The pixel intensities already fell between 0 and 1 and no other processing was deemed necessary for similar reasons to part b: it shouldn’t affect the output.
  + For random forest regression we’ve chosen to use sklearn’s RandomForestRegressor (documentation for which is cited in *main.ipynb*). Below is the output with a maximum depth of five for each of ten trees.

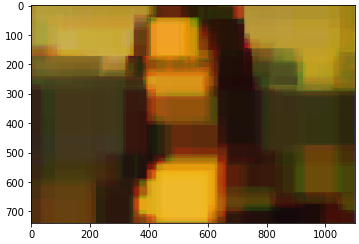


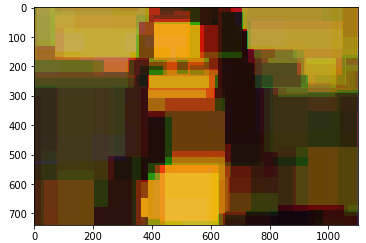
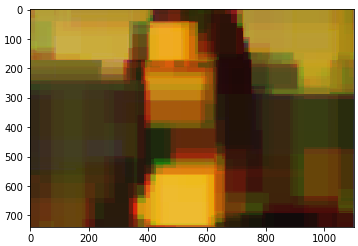
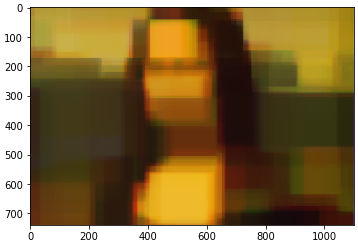
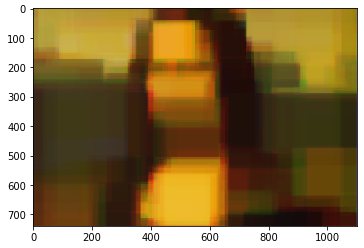
* + Experimentation
    1. Below are the results using a single decision tree and values of 1, 2, 3, 5, 10, and 15 for depth.



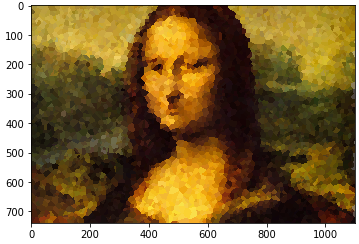
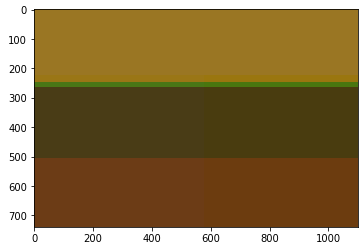
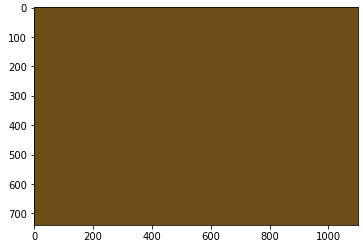


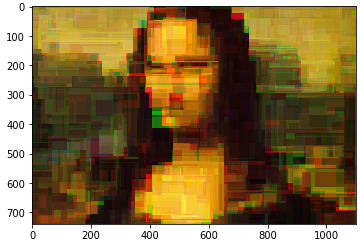
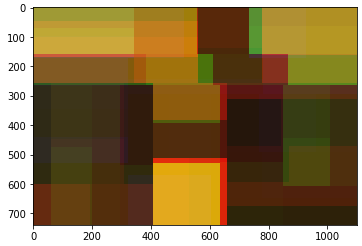
As can be seen, the depth heavily influences the result. With additional depth we see a larger number of colors used and a reasonable approximation of the image at a depth of 15. By increasing the depth we’re exponentially increasing the splits and number of colors. As can be seen, even with a depth of 15 it looks quite blocky and some of the colors appear more extreme than they should be – the blocky nature is because the tree is simply splitting the space and the extreme colors are because we only have one tree voting. With a true forest we would expect those colors to become smoother, though this shouldn’t affect how “blocky” the output is.

* + 1. Below are the results using a random forest of depth seven with 1, 3, 5, 10, and 100 trees.

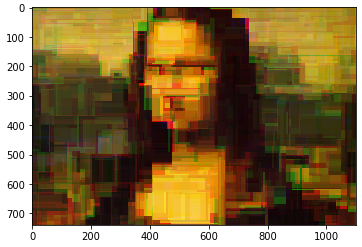
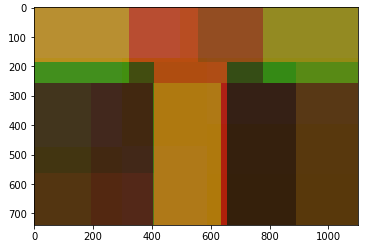
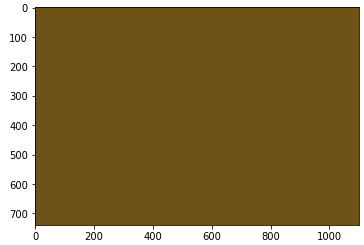


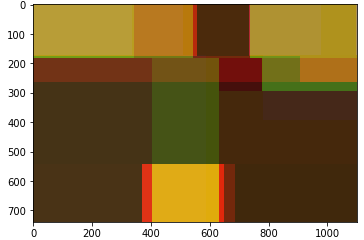
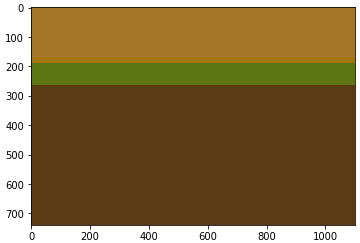
As can be seen, adding trees to our forest does influence the result, though not as drastically as the depth of the trees. We see that the number of colors across the images appears constant and all images have similar segmentation, but with more trees the image appears much less “blocky” (not because the segmentation is better, but because the colors are less extreme) and ends up with colors much closer to the true values. This is because we have the same number of leaf nodes per tree (same number of colors possible) but more trees voting on the output color of each pixel, which helps get the colors used closer to the true values.

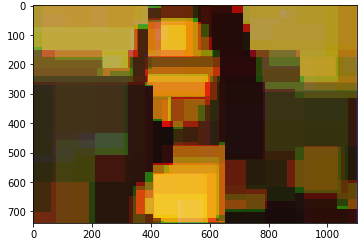
* + 1. Below is the output of a 1-NN regressor on the same Mona Lisa image. The output appears almost painterly with a great assortment of colors (up to 5,000 unique colors). It appears this way because each pixel is taking the color of the closest neighbor in the randomly selected points, which has the effect of making blotches of color in the output.
    2. Below is a pruning experiment that sets the minimum number of samples per leaf to 1, 10, 1000, and 1000000 for a forest composed of one tree of depth 15. As can be seen, increasing this value decreases the fidelity of the resulting image and the number of color patches because we’re reducing the number of leaf nodes.

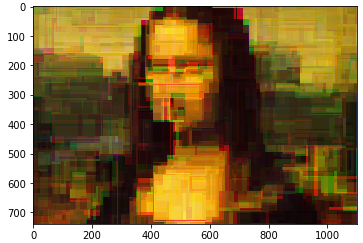
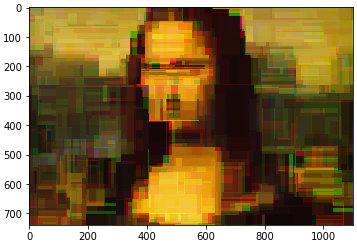


Below is a pruning experiment that sets *min\_weight\_fraction\_leaf* (“minimum weighted fraction of the sum total of weights required to be at a leaf node” from the documentation cited above) to 0.0, 0.1, and 0.5. Much like increasing the minimum number of samples per leaf, this has the effect of reducing the fidelity of the output image because we’re reducing the size of our tree.



Below is the result of varying the maximum number of leaf nodes, setting values of 2, 10, 100, 1000, and 1000000. As can be seen, increasing the maximum number of leaf nodes helps reduce the pixilation in the resulting image up to a point but when we go from 1,000 to 1,000,000, we don’t see an improvement. This is because at these values the number of leaf nodes is not limited by the parameter we’re setting, but instead by the maximum allowable depth of the tree.





* + Analysis
    1. Each split point in a tree is a simple if-statement. In the case of our image regressor the split point is an if-statement asking if the pixel’s *x* or *y* value is beyond a given threshold. For a forest made up of one tree of depth two the regressor produced a tree with a root splitting rule where is the second pixel coordinate.
    2. The resulting image looks “blocky” – distinct rectangles of color can be seen which makes the image look somewhat pixelated. It looks this way because our trees are simply producing splits along each axis. Of course, the split positions aren’t selected randomly, they’re positioned to minimize error, which is why we see the rectangles approximate the shapes in the original image (for the Mona Lisa we see splits separating the portions of the background, the face, hair, chest, etc.)
    3. There can be as many patches of color as there are leaves in the tree. If a tree is dense then this will be where is the depth of the tree.
    4. The number of patches of color in the resulting image shouldn’t be affected by , the number of trees, because each tree is only offering a vote for the final color. The number of patches of color depends on the number of leaf nodes in the largest tree within the forest (assuming consistent tie-breaking rules). Consider a forest of 100 trees, each tree with one split and two unique color votes: given a point you’ll have 100 votes for different colors and must then choose one, and you’ll only be able to produce two color patches in the output. If you have 99 trees in your forest of depth one and one tree of depth two then you can have up to four color patches in the output if the tie-breaking always favors the tree of depth two.