Solution 1

(a) To find the cluster centers for the SLIC algorithm, we start by creating a grid of center points evenly distributed a distance S apart, where

$$S = \sqrt{\frac{N}{K}} \tag{1}$$

and N is the number of pixels in an image, and K is the desired number of clusters. The method works best for a square image and the number of clusters also being a square number. Then, to avoid a cluster center being on an edge, we take consider the gradient of the image in a neighborhood around the cluster center. If we assume edges are thin, we can use a 3×3 neighborhood. We then set the center to be the point in the neighborhood with the smallest gradient using an argmin over the neighborhood. The resulting center points are shown as a grid that is not quite exact Figure 1.

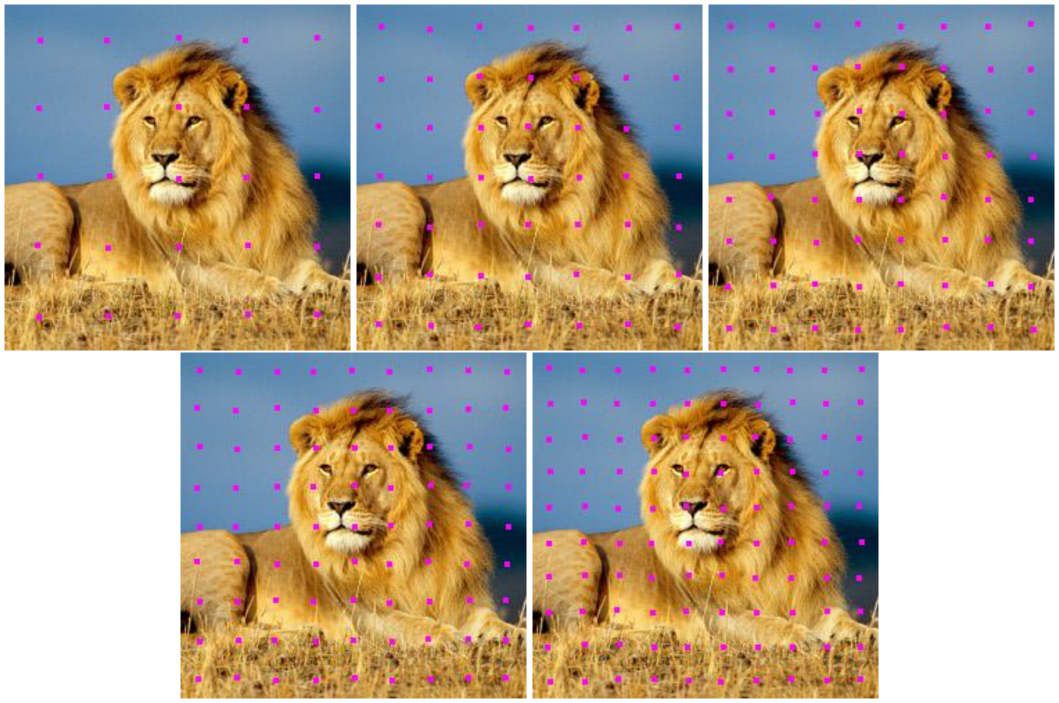
Figure 1: Center Locations for SLIC Clusters

(b) In this part we implement the SLIC Algorithm to find superpixel clusters. We start initializing the centers found in part (a) as the centers for each cluster, μ_k, and running one instance of the algorithm to assign each pixel to a cluster. Then it runs T iterations of calculating the mean value of a cluster and updating pixels to be assigned to new clusters. To improve efficiency we only look at pixels within a $2S \times 2S$ spatial neighborhood from the cluster centers.

Next, to improve clusters, we adjust the spatial weight assigned to the augmented vector. A small spatial weight means the algorithm prioritizes pixels that have similar intensities over pixels that are close together, this leads to clusters not actually clustering and just becoming separated groups of pixels with similar intensities. A large spatial weight means the algorithm prioritizes pixels that are close together, even if they don't share any features, leading to blocky superpixels with little information about the image. The optimal spatial weight is different for different numbers of clusters. A small number of clusters will need a smaller spatial weight because the distance between clusters is larger, so that close to the edges the cost from spatial distance is larger; and a large number of clusters will require a larger spatial weight to compensate for the fact that the maximum spatial distance is much smaller. I settled for a square root scaling of spatial weight to number of clusters,

$$weight = \sqrt{K}/2 \tag{2}$$

where K is the number of clusters. I chose square root scaling, because the distance between clusters scales with $S \propto \frac{1}{\sqrt{K}}$, so to compensate for that I selected a weight that would approximately cancel that out. The downside to larger scaling for more clusters is that some of the smaller details like the eyes get lost, but I think this is preferable to the noisiness from a smaller weight. The results are shown in Figure 2.

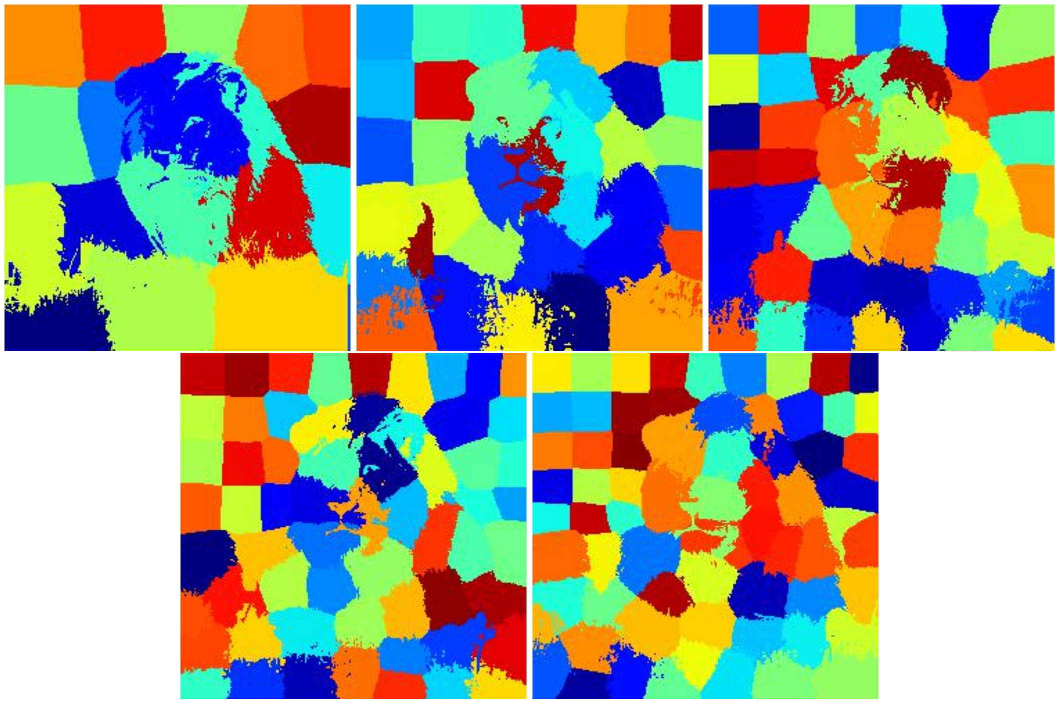
Figure 2: Super Pixel Clusters from SLIC Algorithm

Solution 2

(a) In this problem we looked at an implementation of the autograd system to train a neural network with a single hidden layer. I tested varying three parameters: the batch size for batch gradient descent; the number of units in the hidden layer; and the learning rate of the algorithm. I varied each individually before combining the best of each individual parameter to get the best overall result.

- **Batch Size:** I found that the smaller batch sizes converged faster and had smaller error after convergence, however it took longer to compute an epoch for smaller batches. After one epoch, a batch size of 25 had 78% validation accuracy and 1.42 validation loss, while a batch size of 200 had 36.5% accuracy and 2.28 loss in validation. After 50 epochs, the 25 batch size had accuracy and validation loss of 93% and 0.255, and the 200 batch size had 89% and 0.492. Additionally the small batch had worse loss and accuracy in its training set than in the validation set, while the larger batch size had the opposite. For batch size, it seems that smaller batches are better if you are willing to wait longer for training. A compromise might be trying to max out your CPUs parallel processing capabilities with the batch size to get as fast as possible while still getting the benefits of a reasonably small batch.
- **Learning Rate:** For learning rate, I found that increasing it increased the accuracy, and how fast it reached that point. For a learning rate of 10 times the initial parameter at $\eta = 0.01$, the trained network was performing comparably to the original parameters final values after only 10 epochs, and continued to improve ending with validation accuracy and loss of 96% and 0.139 after 50 epochs. This implies that (a), the initial parameters likely hadn't finished converging after 50 epochs, and (b), the learning rate is smaller than optimal and can likely be increased further, until performance begins to suffer. To test for that, I tried a learning rate of $\eta = 0.05$; with this value the in sample accuracy and loss were much smaller, however the validation loss and accuracy did not improve much, implying that the optimal learning rate is somewhere in the range of 0.01 and 0.05. Increasing the learning rate doesn't cost any more computational resources, so it definitely should be increased from the initial parameter.
- **Number of Hidden Units:** I found that the changes for increasing and decreasing this resulted in only small changes to the algorithm. Doubling the number of hidden units resulted in slightly better results, and halving the units resulted in slightly worse results, but not significantly. This is likely because the neural network has more hidden units than is required to classify numbers, so changing it doesn't result in a significant change. I tested how far I had to decrease the number of hidden units before performance significantly dropped, and was able to make 16 units work with a final validation accuracy and loss of 91.4% and 0.340, only slightly worse than the initial parameter of 1024 units. The in sample accuracy and loss were also comparable. However, decreasing down to 8 hidden units significantly decreased the accuracy and loss of the network for both the in sample and validation sets. This means that the minimum hypothesis class to reasonably classify numbers from a 28×28 pixel image is between 8 and 16 hidden units for a one hidden-layer neural network.

For a final test to see how well my observations worked, I tried seeing if I could quickly train a neural network to perform well, using 64 hidden units, a learning rate of $\eta = 0.02$ and a batch size of 25. The network trained 50 epochs in less than a minute and had validation accuracy and loss of 96.8% and 0.107, and comparable values for in sample loss and accuracy. Compared to the original parameters that took multiple minutes to train 50 epochs and ended with 92% and 0.301 validation accuracy and loss, and worse values for in sample accuracy and validation, this is a large improvement.

The reason the weights are initialized using the Xavier function so that each activation layer has the same magnitude. To do this, we make sure the variance of the outputs is equal to the variance of the inputs, which we set to be 1. The outputs are

$$y = W^T x \tag{3}$$

and if we assume the approximation that the variance of a single term of the output is $var(wx) \approx var(w)var(x)$, then we can solve for the variance required for any given w. Since W is $N \times M$ dimensional and x is N dimensional, an element of the output is the sum of N terms from a column of W with x, each with variance σ^2. Since we know the variance of a sum is N times the variance of a term, we get

$$var(y_i) = N\sigma^2 \tag{4}$$

and we can solve for σ^2 with $var(y_i) = 1$ to get

$$\sigma^2 = 1/N \tag{5}$$

Therefore, the Xavier initialization for the weights is just a uniform distribution with mean 0 and variance $1/N$. To get 0 mean, we set the distribution to be over $[-a, a]$, and then we need to find a that gives us a variance of $1/N$. The variance of a uniform distribution is

$$\sigma^2 = a^2/3 \tag{6}$$

we solve for a with $\sigma^2 = 1/N$ to get

$$a = \sqrt{\frac{3}{N}} \tag{7}$$

(b) In this problem we implement gradient descent with momentum into the autograd algorithm. There is only two changes we have to make: first we have to initialize a history variable for each parameter, then we change the SGD function slightly to make use of the history. Because we can broadcast for the first iteration of momentum in Python, we can simple initialize the history variables to be the scalar 0. Then for each iteration of gradient descent, we update the history to be

$$g_i \leftarrow \nabla_{w_i} + \gamma g_i \tag{8}$$

this which also updates the history value to be the correct size through broadcasting for the first iteration, matching the gradient. Then we just use this history value in place of the gradient to update our weights.

$$w_i \leftarrow w_i - \lambda g_i \tag{9}$$

This method removes some of the stochasticity from SGD, while retaining the quicker computation time. With a large value for the momentum (0.9 in the assignment), the updates can be much larger if several iterations have a gradient with the same magnitude and direction. For example, if we assume a N iterations have a gradient with in the same direction, g_i would be written as the geometric sum

$$g_i = \nabla_{w_i} \sum_{n=0}^{N-1} (0.9)^n \tag{10}$$

for a maximum value of 10 times the original gradient, Additionally, it is easier to overshoot your goal with momentum. This means our learning rate be smaller than for SGD. Testing this theory, I used the same parametres as I used for the last part of (a). The algorithm converged quickly, within 5 epochs, but did not have the same final accuracy as it did without momentum, and it bounced around in accuracy and loss for the remaining 45 epochs. Reducing the training rate by a factor of 10 to $\eta = 0.002$, The algorithm trained slightly slower, but converged to a better maximum. Larger batches had worse performance with momentum, which makes sense, since the direction and magnitudes of the gradients will be similar to smaller batches, however will be fewer iterations per epoch, so after 50 epochs less training will have happened for the larger batch.

Solution 3

In this problem I implemented a (suboptimal) convolution layer for the autograd algorithm that was provided. The forward section I coded naively by looping through all the pixels and performing a convolution, with the exception that I used a vector kernel to apply the input-channel layer rather than an extra for loop.

The back propagation was more difficult, especially trying to deal with changing the number of channels. I started by considering the known information: the partial derivative definition and, the shape of the gradients. The definition,

$$\frac{\partial L}{\partial x[l,m,n]} = \sum_i \sum_j \sum_k \frac{\partial y[i,j,k]}{\partial x[l,m,n]} \frac{\partial L}{\partial y[i,j,k]} \tag{11}$$

where $\partial L/\partial y[i,j,k]$ is known from the previous iteration of the back-propagation. We also know the size of the gradients has to match their original shape, so $\nabla_x L$ is an array of image shaped objects equal to the batch size, and $\nabla_k L$ is going to be shaped like the 4-D kernel.

Next, we consider how any given pixel affects the output y. Since we are considering a convolution with square kernel of size $K \times K$ (ignoring the batch and channel dimensions for now), only $x[i,j]$ to $x[i+K, j+K]$ affect $y[i,j]$, and the partial derivative is just the value of the kernel at that point.

$$\frac{\partial y[i,j]}{\partial x[i+l, j+m]} = K[l,m] \tag{12}$$

Including the last two dimensions now, we get

$$\frac{\partial y[i,j,c_2]}{\partial x[l,m,c_1]} = K[l-i, m-j, c_1, c_2] \tag{13}$$

Similarly, we get the result for $\partial y/\partial K$:

$$\frac{\partial y[i,j,c_2]}{\partial k[l,m,c_1,c_2]} = x[i+l, j+m, c_1] \tag{14}$$

To get the gradients, we sum and multiply by the gradient of the previous step, like in equation 11.

$$\frac{\partial L}{\partial x[l,m,c_1]} = \sum_i \sum_j \sum_{c_2} \frac{\partial L}{\partial y[i,j,c_2]} \frac{\partial y[i,j,c_2]}{\partial x[l,m,c_1]} \tag{15}$$

$$\frac{\partial L}{\partial x[l,m,c_1]} = \sum_i \sum_j \sum_{c_2} \frac{\partial L}{\partial y[i,j,c_2]} K[l-i, m-j, c_1, c_2] \tag{16}$$

and

$$\frac{\partial L}{\partial k[l,m,c_1,c_2]} = \sum_i \sum_j \frac{\partial L}{\partial y[i,j,c_2]} \frac{\partial y[i,j,c_2]}{\partial k[l,m,c_1,c_2]} \tag{17}$$

$$\frac{\partial L}{\partial k[l,m,c_1,c_2]} = \sum_i \sum_j \frac{\partial L}{\partial y[i,j,c_2]} x[i+l, j+m, c_1] \tag{18}$$

I coded equations 16 and 18 to loop over i, j processing element of the gradient in parallel.

Information

This problem set took approximately 25 hours of effort.

I discussed this problem set with no one.

I also got hints from the following sources:

- Wikipedia article on uniform distributions for variance: `https://en.wikipedia.org/wiki/Continuous_uniform_distribution`
- Numpy documentation for argmin in Spyder (for np.unravel_index in prob1)