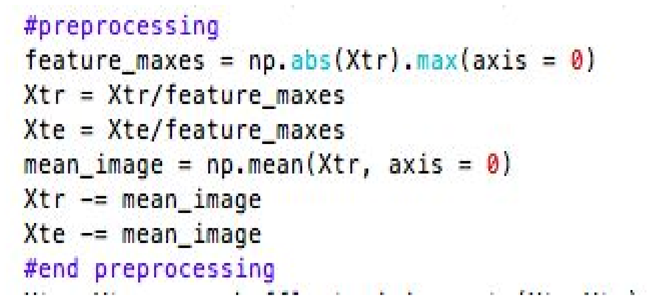
Metehan Ozten
Professor Yan

CS291K Report

Implementation:

Preprocessing:
Before the training, test and validation data is even passed into the Neural Network, several computations are performed on the the images to allow them to perform better within the neural network. The preprocessing code is rather simply and is shown to the left in Figure 1.

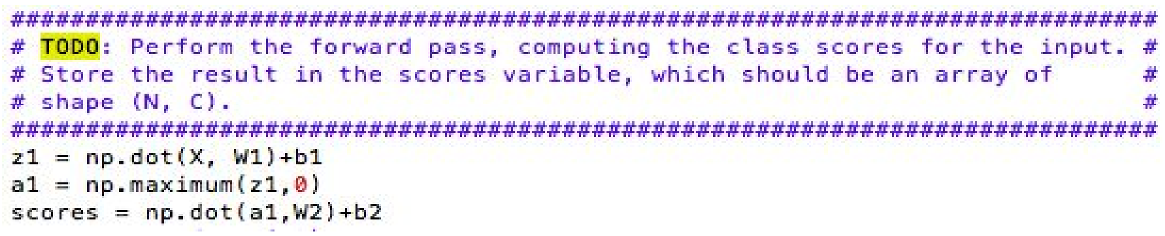

```
#preprocessing
feature_maxes = np.abs(Xtr).max(axis = 0)
Xtr = Xtr/feature_maxes
Xte = Xte/feature_maxes
mean_image = np.mean(Xtr, axis = 0)
Xtr -= mean_image
Xte -= mean_image
#end preprocessing
```

Initially we begin by calculating max values for each color for each pixel of the training image set and then we divide the training, test, and validation sets by this vector. This is a form of min-max normalization and greatly improves the accuracy of the Neural Network as well as prevented me from running to mathematical errors such as overflow and divide-by-0 errors. Next, we subtract the mean-image (of the training set) from the test,training and validation sets in order to improve accuracy as well as improve the functionality of the neural network due to the fact that if the mean image isn't subtracted the neural network doesn't operate as well, values start exploding leading to overflows and other mathematical errors.

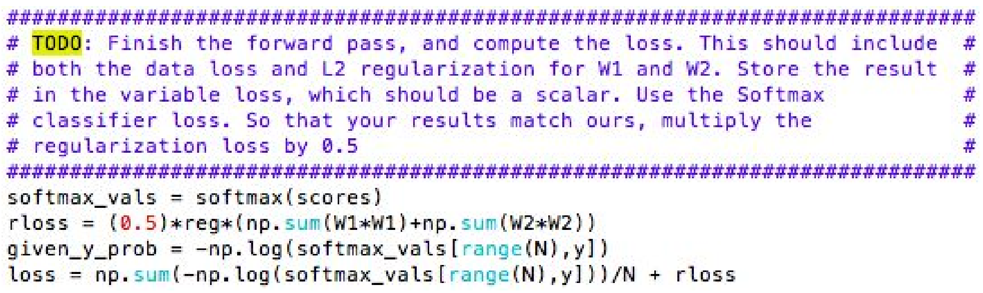
The Neural Network Implementation:

Forward Propagation:

Our implementation of forward propagation was relatively straight-forward since the forward propagation of neural networks is essentially the simple portion of the algorithm. The image (below) contains the code snippet that will allow us to perform forward propagation (without the final softmax activation function). The np.maximum call is used for our relu hidden-layer activation function.


```
#############################################################################
# TODO: Perform the forward pass, computing the class scores for the input. #
# Store the result in the scores variable, which should be an array of      #
# shape (N, C).                                                             #
#############################################################################
z1 = np.dot(X, W1)+b1
a1 = np.maximum(z1,0)
scores = np.dot(a1,W2)+b2
```

Softmax Function and Loss Calculation



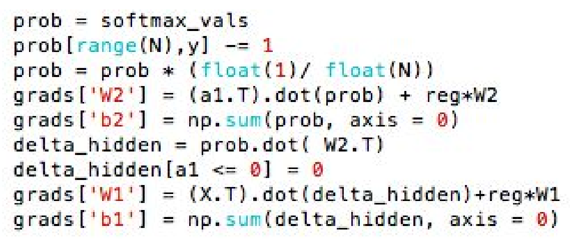
```
#############################################################################
# TODO: Finish the forward pass, and compute the loss. This should include  #
# both the data loss and L2 regularization for W1 and W2. Store the result  #
# in the variable loss, which should be a scalar. Use the Softmax           #
# classifier loss. So that your results match ours, multiply the            #
# regularization loss by 0.5                                                #
#############################################################################
softmax_vals = softmax(scores)
rloss = (0.5)*reg*(np.sum(W1*W1)+np.sum(W2*W2))
given_y_prob = -np.log(softmax_vals[range(N),y])
loss = np.sum(-np.log(softmax_vals[range(N),y]))/N + rloss
```

Above is the code that calculates the softmax of the scores which was calculated in the last section. The softmax is used to assign a probability to each member of the output vectors contained in scores. The softmax is therefore also useful because the vector column with the highest probability is the class that image is classified under. In this code we also perform L2 regularization in the second line, as well as calculate the softmax loss. The variable given_y_prob is essentially an array of probability values of the dimension $numtrainingsamples * 1$, which is the probability that the classifier assigned to the image belonging to the correctly labeled class. If you look at the equation shown below, this array essentially contains the -log p(Y=Y_i| X = x_i), which are later summed.

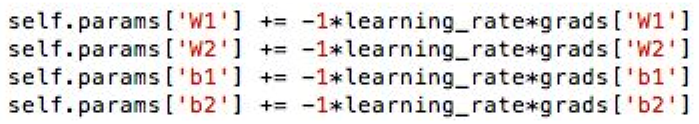
$$L = -\frac{1}{N}\sum_{i=1}^{N}\log p(Y = y_i \mid X = x_i) + \frac{1}{2}\lambda \| W \|_2^2 ,$$

Backpropagation and Weight Update

The implementation for backpropagation and weight updating code is contained in the two images below respectively . The code to the right shows how the backpropagation was performed such that we could obtain the gradients for the initial layer and the hidden layer (as well as the biases). The code that updates the gradients is shown below and notice we are adding the negative of the gradients (subtracting them from W1 and W2) because we wish to minimize the loss.



```
prob = softmax_vals
prob[range(N),y] -= 1
prob = prob * (float(1)/ float(N))
grads['W2'] = (a1.T).dot(prob) + reg*W2
grads['b2'] = np.sum(prob, axis = 0)
delta_hidden = prob.dot( W2.T)
delta_hidden[a1 <= 0] = 0
grads['W1'] = (X.T).dot(delta_hidden)+reg*W1
grads['b1'] = np.sum(delta_hidden, axis = 0)
```



```
self.params['W1'] += -1*learning_rate*grads['W1']
self.params['W2'] += -1*learning_rate*grads['W2']
self.params['b1'] += -1*learning_rate*grads['b1']
self.params['b2'] += -1*learning_rate*grads['b2']
```

Model Building

Once the implementation of the model was complete and after I played with the parameter values by hand for a little bit, I then began developing a hyper-parameterization script (contained in hyper.py) which essentially trains many neural networks with varying parameters and dumps the results including training accuracy, validation accuracy, loss, model parameters and time to train to a CSV file.

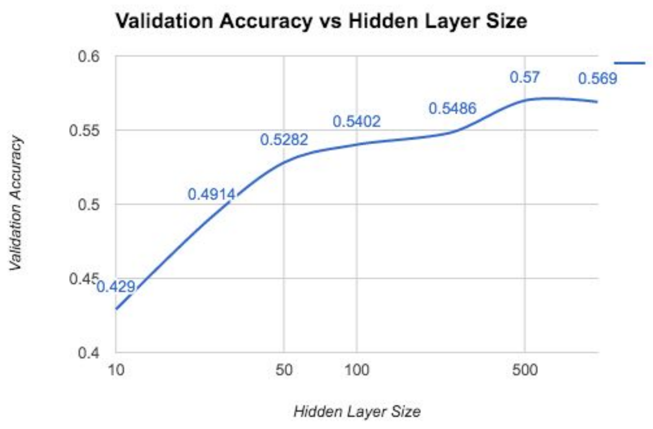
As you can see above, this instantiation of the hyper-parameterization script requires the training of many models and would take too long to complete (considering I wrote this script 1 day before the project was due). So what I did in order to lower the hyper-parameterization time was to have multiple hyper-parameterization batches where I only varied a subset of the parameters. The first parameters that I varied were regularization strength and hidden_layer_size, which I read online were related. I varied the regularization strength from .1 to .0001 and the hidden layer size from 10 to 1000. After running this hyper-parameterization, I came to the conclusion that regularization strength of 1e-3 and hidden layer size of 500 was optimal. Next, I ran another batch of hyper-parameterizations where I varied learning-rate and learning-decay and came to the conclusion that a learning rate of 1 was optimal and a decay of .9 was also optimal. My decay rate was probably the most accurate at .9 because I always used 5000 iterations in my model.

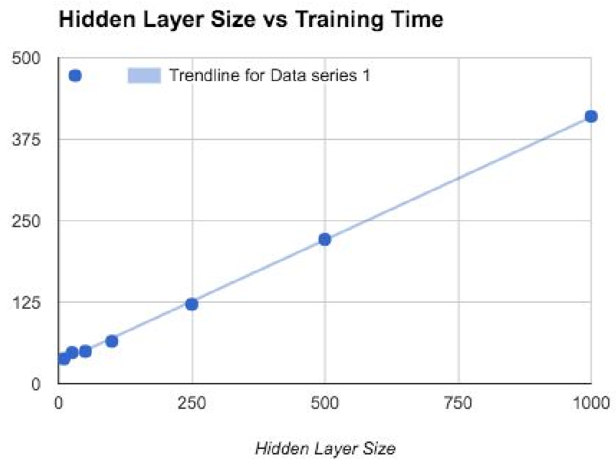
Results

The optimal performance of my model occurred when I used the following parameters. Although I tried many different parameters and received varying results, they are not all in this report due 4 page limit.

Hidden Layer Size	Learning Rate	Learning Rate Decay	Reg	Batch Size	Num-iterations
500	1	.9	.001	250	5000

With these parameters I obtained a testing accuracy of 57% with a training time of 221 seconds.





Potential Improvements

There were a couple of improvements that I could have implemented if I had more time. The first being image-whitening which would improve the algorithm by removing redundancies in the data. However, I believe this would mainly improve your training-time, requiring fewer training iterations. The second improvement that I would implement would be drop out, which I heard greatly improves accuracy by preventing overfitting.

Challenges

The implementation of the neural network was by far my biggest challenge, as I didn't know that normalization and mean removal were important, and as such my loss grew as I ran more iterations and ran into many arithmetic issues such as overflow and divide by 0 errors. I probably spent around 12-13 working hours trying to fix the problems in my program until I realized how crucial removing the mean image was.