Paul Galloway

CS291K
Machine Problem 1
Report

Implementation:

The starting point of my implementation was the neural_net.py file provided. The key points of this file are the functions contained in the TwoLayerNet class. The first function in this class is the loss function.

The loss function serves the dual purpose of providing the forward pass to produce a prediction on unseen input. It does this via a set of linear algebraic transformations and activation functions. The second part of this function implements the backward pass when the function is provided with a set of expected output. This part of the function initially transforms the output (y) vector into a one-hot encoding and computes the loss between this one-hot encoding and what the existing forward pass transformations computed. It utilizes a softmax loss function and L2 regularization on the existing weights to compute an overall loss value. This is followed by the backward pass in which it uses the difference between the y-label matrix and the output from the forward pass to backpropagate the error. The outcome of this is the newly computed gradients. It then returns the latest loss and gradients in order to provide for the next step in the stochastic gradient descent algorithm. After this the loss function is called again with the new weights provided from the latest learning step.

The rest of the implementation was more-or-less provided to us and is described in the following section.

Model Building:

The neural network is trained via a stochastic gradient descent (SGD) approach. From a larger training set we sample a random mini-batch. We compute the loss on this mini-batch and also generate a set of gradients to update the existing weights. This new set of weights are then used on another random mini-batch that we generate. This process continues for the number of iterations we specify. We break up these iterations into epochs and sample how we are doing along the way.

In terms of choosing the hyperparameters I took a more manual approach to this rather than re-implementing a GridSearch-esque approach. After varying the number of iterations and batch sizes I noticed significantly diminishing returns and so I chose to not pursue these values beyond 10000 total iterations and a batch size of 200.

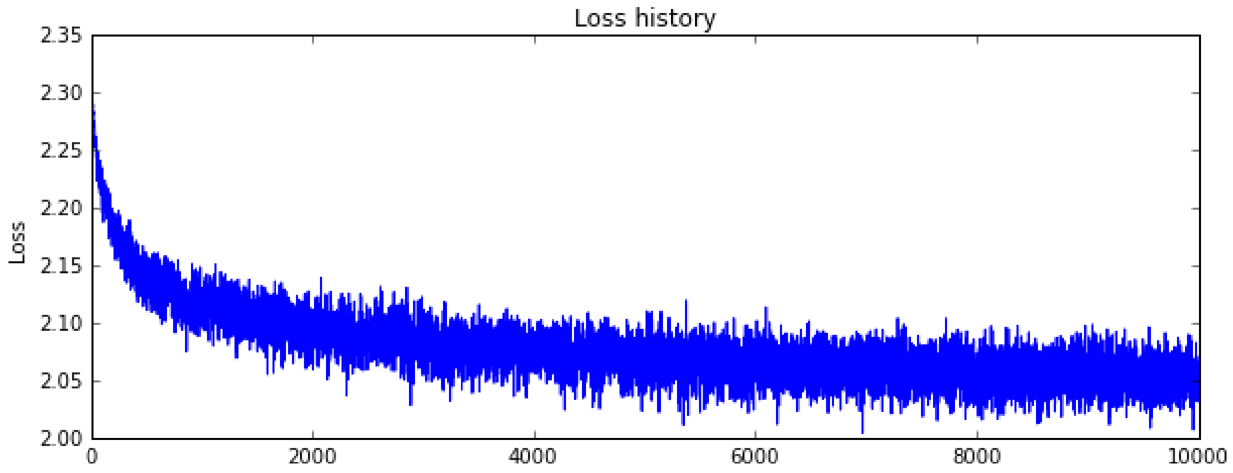
Results:

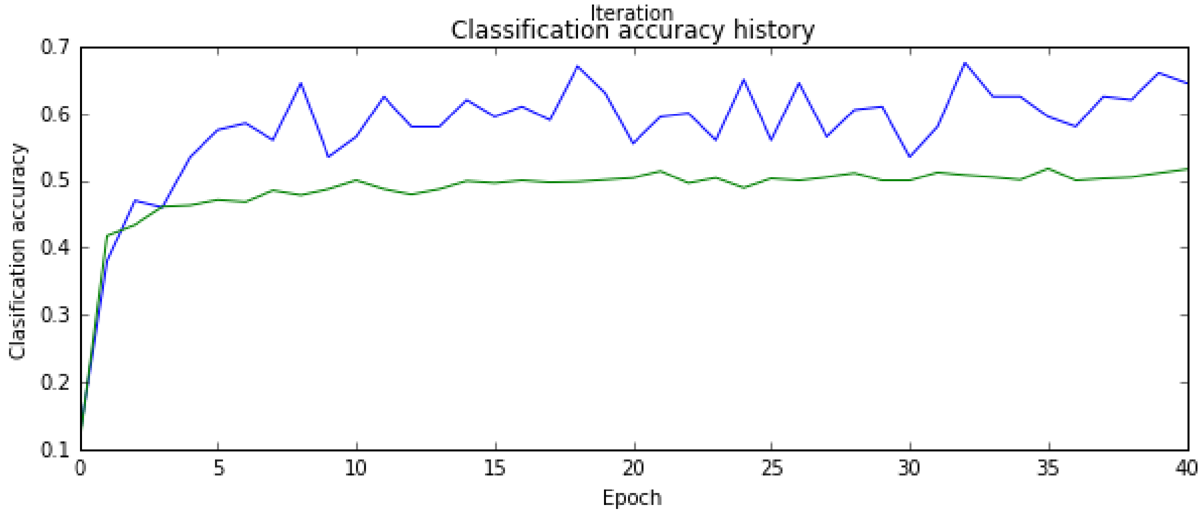
Training accuracy:	**0.592367346939**
Validation accuracy:	**0.502**
Test accuracy:	**0.495**
Training time (in mins):	**2.20339854956**

The hyper-parameters that this final model used were:

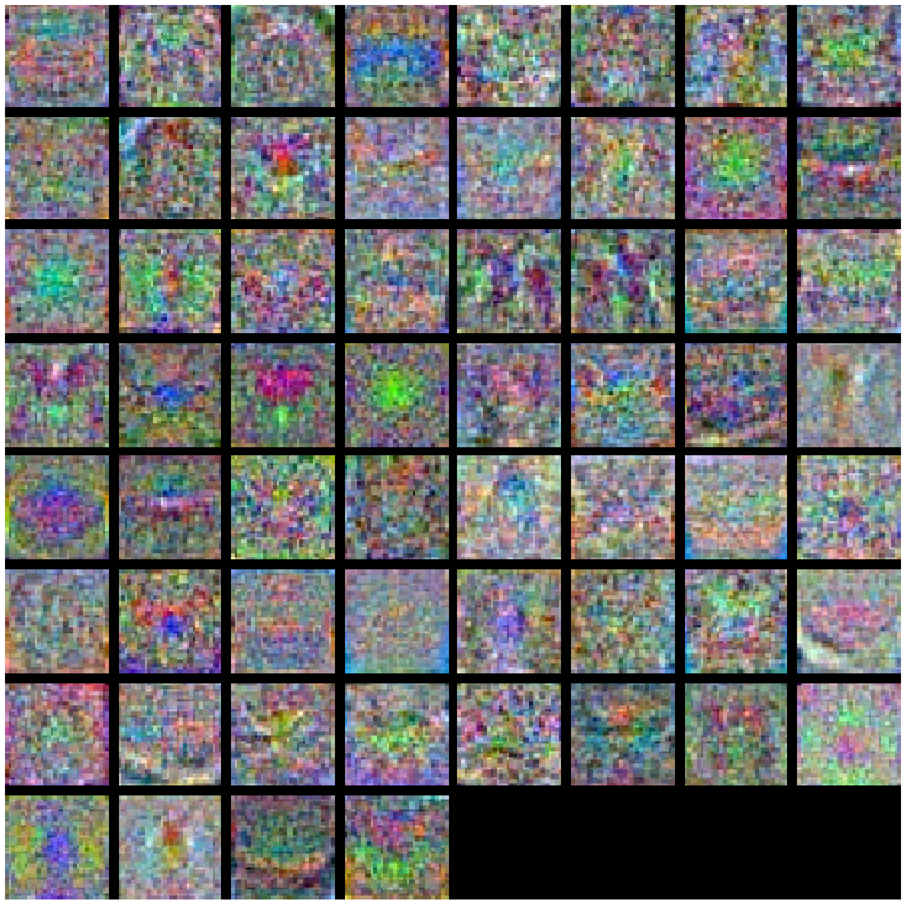
Hidden Size: **60**
Number of iterations: **10000**
Batch Size: **200**
Learning Rate: **0.001**
Learning Rate Decay: **0.95**
Regularization term: **0.1**

Featured below are graphs showing how Loss and Classification accuracy varied with the number of iterations. These graphs were generated using the Stanford Assignment ipython notebook. (Green = validation, Blue = training)





Additionally, a visualization of the final net weights is shown here:



Challenges:

There was only one key problem I faced and it was getting my computed gradients to have a low relative error to the actual numerical gradient descent during gradient checking. This ultimately appeared to not matter that much for my accuracy in the end. But I might've achieved a better loss if I had computed these gradients more accurately. Additionally, the loss history curve above might've not been as volatile had the gradient values been more accurate.

Possible Improvements:

As was stated above, one key thing that I could probably do is figure out why the gradients I computed had relative error higher than expected when doing gradient checking. Beyond this, I think I probably could have found at least a marginally more accurate model had I used a rigorous GridSearch-style scan of the hyper-parameter space. But based on the prior knowledge that this type of network is not specialized for vision tasks and the diminishing returns I had already manually observed, I did not pursue this this path.