Classifying Images Using Neural Nets

CS291K : Advanced Data Mining, MP1

Neeraj Kumar (9408485)

Introduction

One of the most well studied machine learning problem is *classification*. Given a set of objects belonging to different classes, we want to label an object with its appropriate classe. There are so many approaches to this problem, for example k-nearest neighbor search, neural networks to name a few. In this project, we implement a Neural Network based model which learns the features using a *training* dataset and uses those features to classify the images in the *test* dataset. We use the CIFAR-10 dataset [1] to evaluate our model. The next few sections go into details of our implementation and discuss the observed results.

Implementation Details

Our neural network comprises of three layers. The input layer size is the number of pixels times three (one for each color). The hidden layer size H is a hyperparameter, we will fine tune to get to the correct size. Following is a rough overview.

1. The top level file is called `driver.py`. It basically preprocesses data, initializes the network and calls it on training and test sets.

2. The file `network.py` implements the neural net. The forward propagation is straightforward. For the backpropagation, observe that the gradient at the softmax layer $\delta_3 = \hat{y} - y$, where $\hat{y}$ is the predicted output and y is the actual. The other gradient values are updated as in the lecture notes.

3. For updating the parameters, we use simple vanilla update using the learning rate η. We also try a bunch of other update propagation techniques such as `RMSProp` and *momentum* updates.

4. The file `utils.py` contains a bunch of utility functions such as activations and their differentials.

Model Building

In this section, we will discuss the hyperparameters for our model and how exactly we arrived at those values. We fix the mini-batch size to be 1000, and the number of iterations to 1200. These values are based on the tradeoff bewteen accuracy and training time.

1. **Learning Rate** This is used to determine the rate at which we update the weights and biases. We tried a few random samples as $\eta = 10^{\texttt{uniform(-6, -2)}}$. Learning rates higher than this resulted in low accuracy. A plot is shown in Figure 2. We finally settled for the value $\eta = 5e-3$.

2. **Learning Rate Decay** In training neural networks, adjusting the learning rate over iterations can really come in handy. We try a few values from the set $[0.95, 0.99, 0.998]$. We found that the best validation accuracy is achieved with higher that decay rate, and adjusting the learning rate by that factor on every iteration. Based on the observations in figure 1, we settle for the value 0.998.

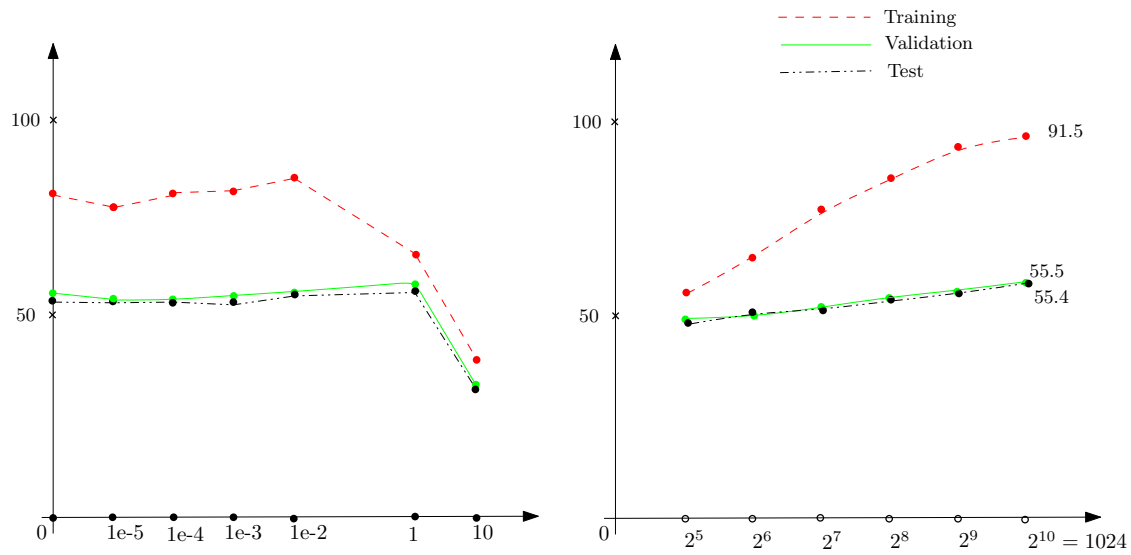

Figure 1: Effect of (a) Regularization and (b) Hidden Layer Size on the accuracies.

3. **Hidden Layer Size** After fixing the learning rate and decay, it makes sense to fix the hidden layer size H of our model. As one may expect, the accuracy improves with increasing value of the hidden layer, and the number of iterations. Quite naturally, the time taken to train the model also increases. We try the sizes $[16, 32, 64, 128, 256, 512, 1024]$ and see an increasing trend (Figure 1). Tha maximum accuracy of 55.46 on the test data was achieved with a hidden layer of 1024 neurons, trained for 1200 iterations with a total running time of about 44 minutes.

4. **L2 Regularization** The initial loss value on CIFAR-10 dataset is 2.30. We start with zero regularization which gave a minimum value of loss 1.14. However, there is quite a difference between training and validation accuracy. This suggests trying non-zero regularization values (see Figure 1). As expected, the loss function goes up. Based on the accuracy achieved, we settle for a value of $1e-3$.

Results

The following table gives a summary of our results. For all these results, we set regularization to $5e-3$ and mini-batch size to 1000, decay rate to 0.998.

Hidden Layers	Iterations	Training Acc.	Validation Acc.	Testing Acc.	Training Time
32	1200	59.2	49.44	49.01	88s
128	1200	67.3	50.84	50.9	3m13s
256	1200	82.5	53.88	53.57	7m11s
512	1200	90.1	54.94	54.82	17m8s
1024	1200	91.5	55.55	55.46	44m9s

Extra Credits

In order to make the model more effective, we also added a few extra features.

- *Momentum* is an efficient update propagation technique. The idea is to have an additional hyper-parameter μ, which will help converge to the minima faster. It behaves similar to the coefficient of

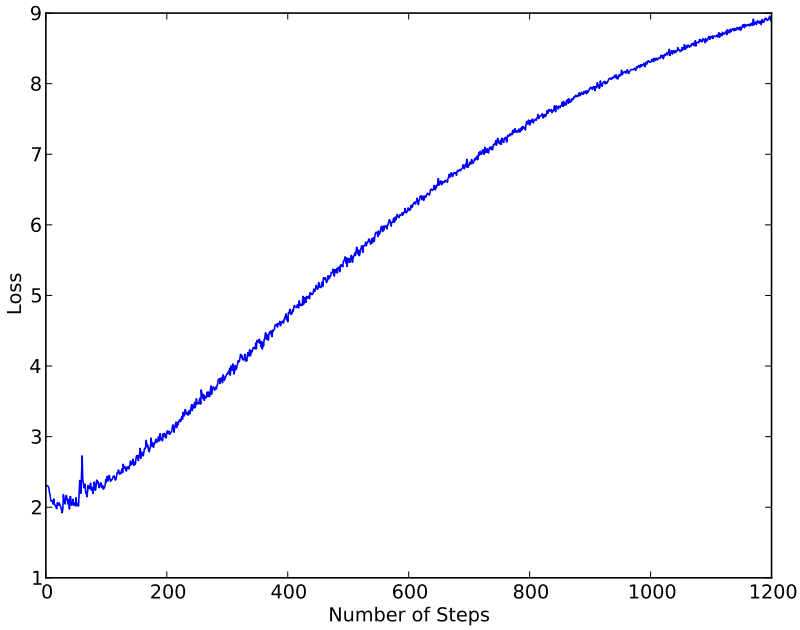

Figure 2: Loss function plotted over 1200 iterations.

friction and reduces brings the object to minima quickly. The equation is straightforward.

$$v = \mu \cdot v - \eta \cdot \partial W \qquad \eta \text{ is learning rate}$$
$$W += v$$

We pick a value $\mu = 0.5$. The effects of momentum are shown in following table.

μ	Num Iterations to convergence
disabled	950
0.5	650
0.7	800

- *Dropout* is a technique to reduce overfitting by randomly sampling neurons based on a threshold probability p and disabling them. The effects are shown in following table.

Probability	Training	Validation	Test
1	59.2	49.44	49.01
0.7	44.4	36.9	35.3
0.5	31.8	28.6	28.5

It appears that dropout is not very effective in our case.

- *Activation Functions* We also tried our network with different activation functions, namely `ReLU`, `sigmoid`, `tanh`, `pReLU`. The effects are shown in following table. These values are with hidden layer size 32 and hence accuracies are small.

Activation	Training	Validation	Test
ReLU	59.2	49.44	49.01
Sigmoid	48.3	45.81	45.87
tanh	57.3	48.1	48.39
pReLU, $\alpha = 0.75$	60.5	49.32	48.8

- *Initialization* It is common to initialize the neural nets with small random numbers picked from a gaussian distribution with mean zero and variance 1. We use a multiplication factor of `std = 0.01`. Another initialization technique is to have an additional factor of $\sqrt{\frac{2}{n}}$.

Initialization	Training	Validation	Test
Gaussian * $\sqrt{\frac{2}{n}}$	59.2	49.44	49.01
Gaussian	63.3	49.5	49.8
Zeros	19.1	20.1	19.2

  Clearly, random initialization is better.

Challenges and Improvements

One of the challenges faced was to deal with large exponential values for the softmax function. If the input image values are fed as it is, the exponential becomes too big in course of time. Therefore, we normalize the input data so that all the values have absolute values within 1. Additionally, we also add a term ϵ to the softmax denominator to avoid divide by zero issues.

One possible improvement could be to do a PCA (Principal Component Analysis) of the input image and then work on it. This is known to give better accuracy. However, it turns out that naive PCA implementation using `numpy` linear algebra library takes a long time on these large matrices. One could look into ways to workaround this problem.

References

[1] CIFAR-10 dataset. `https://www.cs.toronto.edu/~kriz/cifar.html`.