DD2424 Deep Learning in Data Science

Assignment 1 - Lab report, Valdemar Gezelius (vgez@kth.se)

April 6, 2020

About the lab

The lab focused on to what extent we can train a one layered network to be able to classify pictures. The data-set came in batches of 10000 data-points with 3072 features each, divided into 10 unique classes. A *softmax* classification using a mini-batch gradient descent model meant that for each data-point, the model makes a prediction stored in a *probability vector* and *argmax* of said vector is predicted to be the class of the data-point. For each mini-batch, the weights and biases is updated.

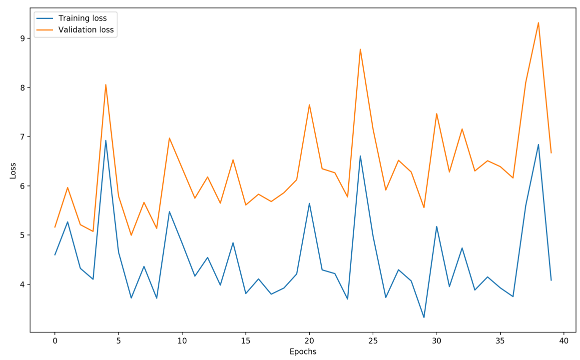
Gradient examinations

I have provided a pdf containing the code where I've (after some trail and error) successfully implemented the functions needed to compute the gradient analytically. I then used *Pythons* built-in *Unittest* to compare my results against the numerically computed gradient. First comparing the two gradients directly using the built-in function *assertAlmostEqual*(g_a, g_n), where we measure that the analytically computed gradient and the numerical does not differ down to a threshold $<10^{-6}$. Secondly by implementing *Eq.1*, where the relative error between the numerical gradient g_n and the analytical gradient g_a, is computed (here *eps* a very small positive number)

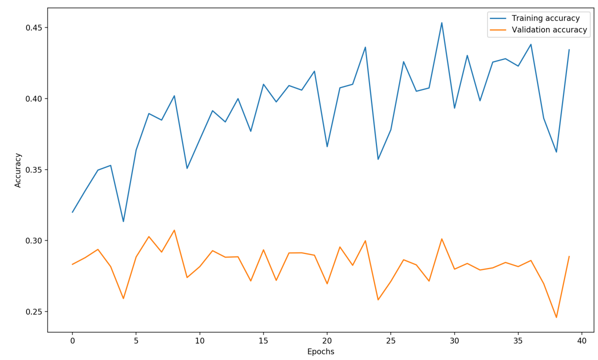
$$\frac{|g_a - g_n|}{max(eps, |g_a| + |g_n|)} \tag{1}$$

The function I then used was the built-in *assertLessEqual(g, num)*, that return *True* iff g is less than or equal to *num* (where *num* is a small number, for testing I set this to 10^{-6}).

A reflection I made during the testing was that though I got a result that I wanted, the testing was a bit unstable and worked better when presented with the full dimension of the input, and even better when calculated over several data-points. Since that was the way it would be used in the lab, and my results (shown later in the report) seemed to be accurate, I came to the conclusion that (to the extend of my knowledge) the gradient calculations worked the way they where supposed to.



(a) Loss curve, test1



(b) Accuracy curve, test1

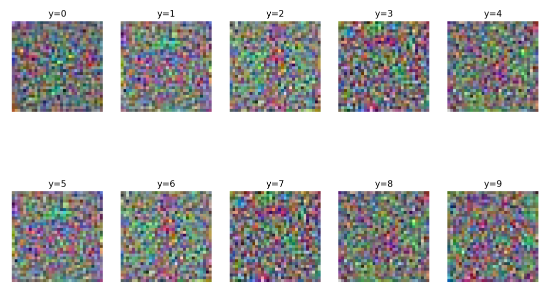
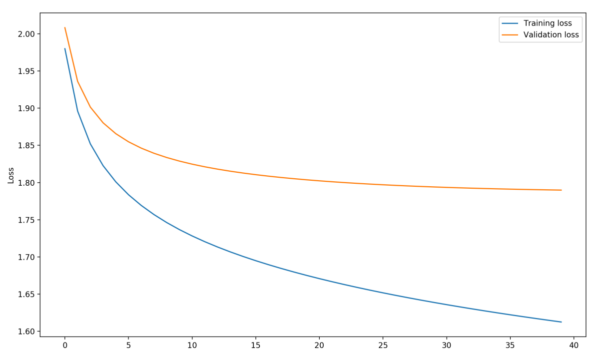
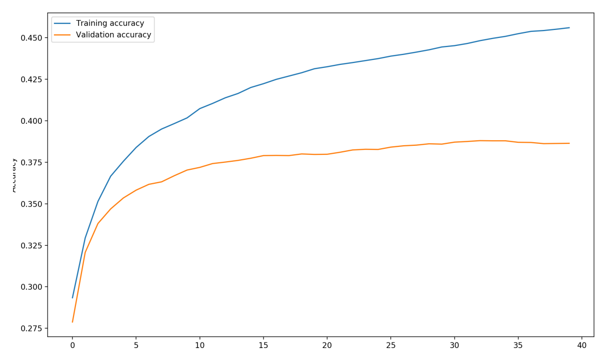


Figure 1: Test1 - the *loss*- and *accuracy* plots together with the learnt *W-matrix* visualized as class template images. The network was trained with the following parameter settings: *nbatch*=100, *eta*=.1, *nepochs*=40 and *lambda*=0



(a) Loss curve, test2



(b) Accuracy curve, test2

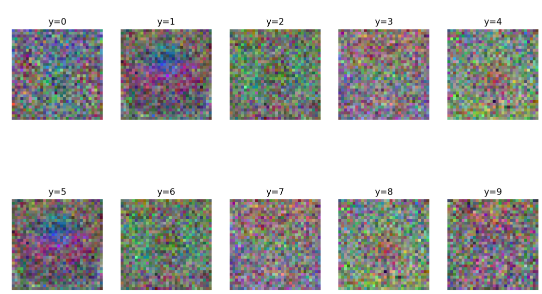
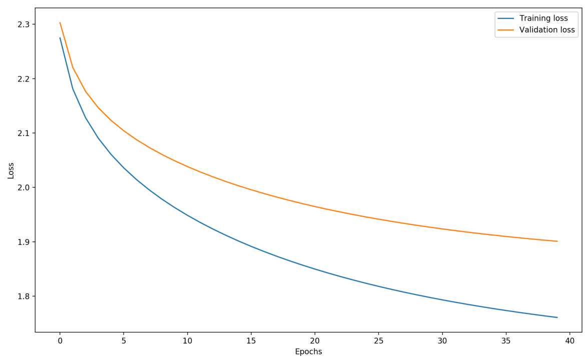
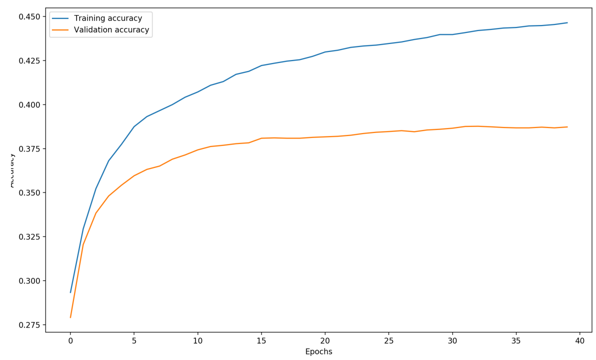


Figure 2: Test2 - the *loss*- and *accuracy* plots together with the learnt *W-matrix* visualized as class template images. The network was trained with the following parameter settings: *nbatch*=100, *eta*=.001, *nepochs*=40 and *lambda*=0

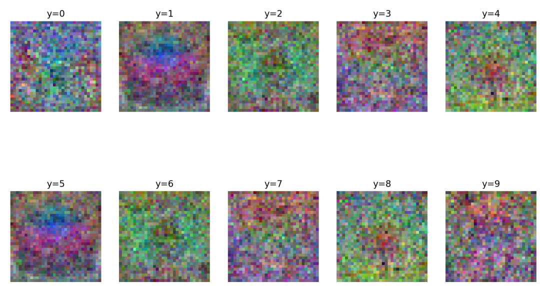


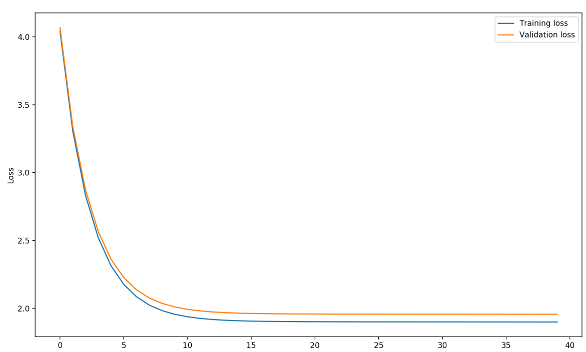
(a) Loss curve, test3



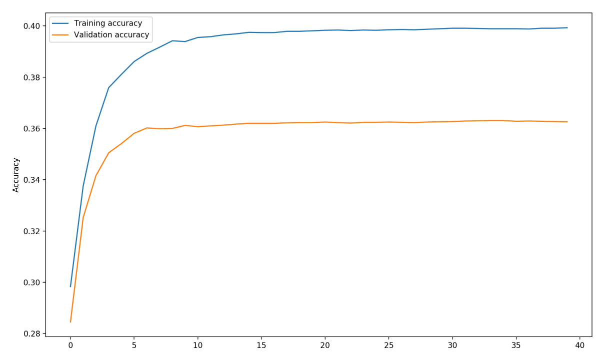
(b) Accuracy curve, test3

Figure 3: Test3 - the *loss*- and *accuracy* plots together with the learnt *W-matrix* visualized as class template images. The network was trained with the following parameter settings: *nbatch*=100, *eta*=.001, *nepochs*=40 and *lambda*=.1



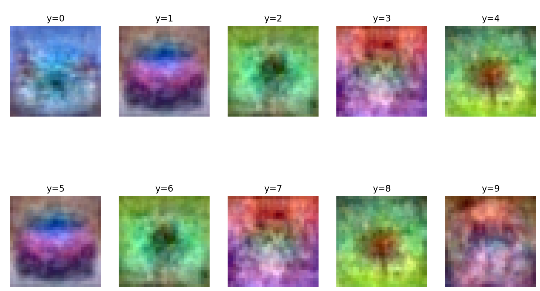


(a) Loss curve, test4



(b) Accuracy curve, test4

Figure 4: Test4 - the *loss*- and *accuracy* plots together with the learnt *W-matrix* visualized as class template images. The network was trained with the following parameter settings: *nbatch*=100, *eta*=.001, *nepochs*=40 and *lambda*=1



Test	lambda	eta	accuracy	loss
1	0	0.1	0.3029	6.4767
2	0	0.001	0.3975	1.7548
3	0.1	0.001	0.3969	1.8668
4	1	0.001	0.3791	1.9292

Table 1: Table compiling the accuracy and loss of the network for the test set. *nbatch*=100 and *nepochs*=40 for all four tests.

Results

I conducted four test cases of my code with different parameter value combinations. The resulting plots and visualizations can be seen in *Figure 1-4* above. The results from the each tests test-data are compiled in *Table1*.

Regarding the learning rate parameter *eta* one can see that when *eta* is "big" (as in *test1*) the resulting graphs for loss and accuracy is really chaotic. That could be explained by the fact that the updates to the weight matrix W and the bias vector b are not as small as they need to be and there is a chance that local minimums are missed and therefor, for the next epoch, this is compensated for, resulting in a jagged graph. When *eta* decreases (for *test2-4*) the corresponding graphs become more smooth.

Regarding the regularization parameter *lambda* one can see that when *lambda* is "big" (as in *test4*), the resulting loss curve has a steeper slope, both the training- and validation loss takes fewer epochs to decrease and after about 15 epochs the network has found its minimum. The visualization of W becomes smoother when *lambda* increases as the updates start including a regularization term.

To conclude, from the tests I ran I got the best results on the test data for *test2* when there was no regularization at all. I find this a bit strange since regularization is suppose to counteract that the network overfits to the training data. But this is of course only a small test in a network with one hidden layer and the accuracy is still low so I would imagine regularization has a bigger impact in networks with many hidden layers.