DD2424 Deep Learning in Data Science
Assignment 3

Lingxi Xiong
lingxi@kth.se

27th May, 2019

1 Gradient check

The first debug check is if MX matrix and MF matrix is correct. The result of MX * ConvNet.F1(:) and MF * x_input is equal. Therefore the second check is to compare the analytical gradient result with numerical gradient.

When first applying 5 filters of width 5 in both layers, the differences between analytical gradients of F1, F2, W and numerical gradients when batch size=1 are: 3.3398e-11, 3.8313e-11, and 1.2494e-10, respectively.

The same setting but batch size=2: 6.6120e-11, 5.0664e-11, 8.2653e-10; batch size=3: 6.1553e-11, 4.7647e-11, 1.2034e-09.

Change the parameter to 20 filters of width 5 in first layer, 20 filters of width 3 in second layer, and batch size=3. The difference is still very small: 1.0353e-10, 9.2555e-11, 3.3362e-09.

Hence, we can draw the conclusion that the gradient computing is correct.

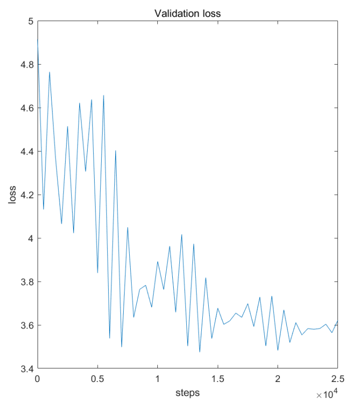
2 Imbalanced dataset

To deal with imbalanced dataset, I random sampled the same amount of samples from each class as training data. And every epoch the training data are resampled. The implementation can be found in function *RandomSample* , where MX matrix is calculated accordingly.

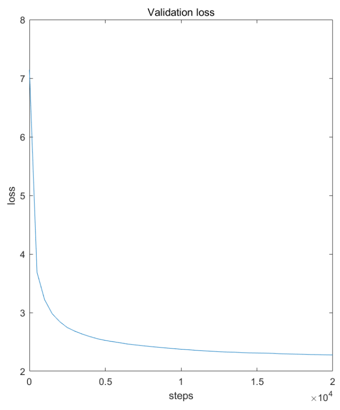
3 Longish training

The validation loss for 20000 update steps for a network with k1 = 5, k2 = 3 and n1 = 20, n2 = 20 without compensating for the unbalanced dataset is shown in Figure 1(a), and Figure1(b) is after compensating. Figure 2 shows the final confusion matrix on the validation set in both cases.

From Figure2 we can see that when training with unbalanced data, the network tends to overfit on the classes that has most data(class 15 has more

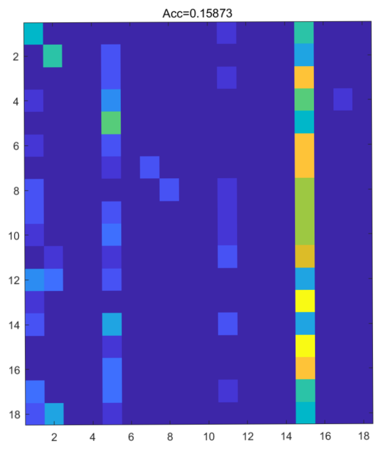


(a) Unbalanced training

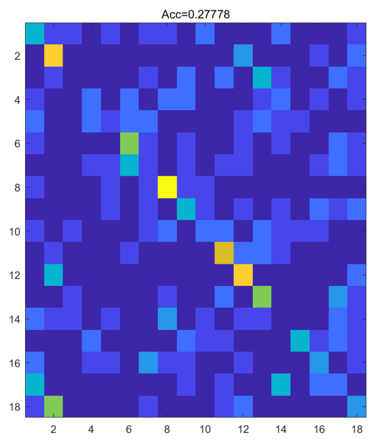


(b) Balanced training

Figure 1: Validation loss

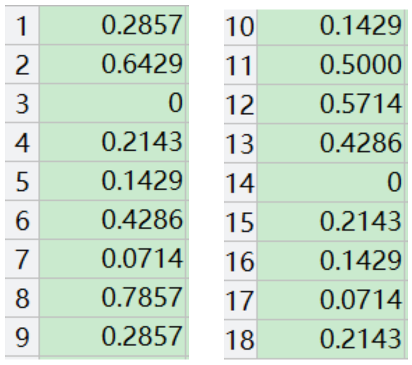


(a) Unbalanced training



(b) Balanced training

Figure 2: Final confusion matrix on the validation set



1	0.2857	10	0.1429
2	0.6429	11	0.5000
3	0	12	0.5714
4	0.2143	13	0.4286
5	0.1429	14	0
6	0.4286	15	0.2143
7	0.0714	16	0.1429
8	0.7857	17	0.0714
9	0.2857	18	0.2143

Figure 3: Validation accuracy for each class

than 9000 training data, class 5 has more than 3000 training data, whereas class 14 and 18 only have less than 100 training data). Yet when training with balanced data, the confusion matrix shows more reasonable result. What's more, the change of validation loss of balanced training is much smoother than unbalanced one.

4 Best performing CovNet

By applying the loss function:

$$L_{unsampled}(\mathcal{B}, \boldsymbol{F_1}, \boldsymbol{F_2}, W) = \frac{1}{\mathcal{B}} \sum_{(X,y)\in\mathcal{B}} p_y l(X, y, \boldsymbol{F_1}, \boldsymbol{F_2}, W) \tag{1}$$

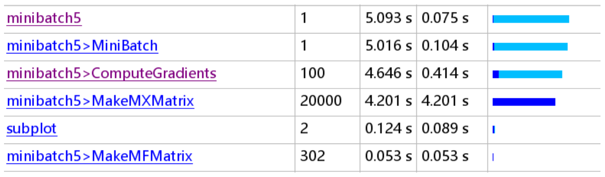
where $p_y = \frac{1}{K}\frac{1}{n_y}$ is a weight applied to the loss for each individual training example in the batch, n_y is the total number of training examples in the whole training set with label y, and K is the number of classes.

Keep the setting of 20 filters for each layer with 5 and 3 filter width respectively, random sample same number(56) of samples from every class as training data(shuffle afterwards) for every epoch, after training 24000 steps with 100 as batch size, the final validation accuracy is 28.57%. Figure s shows the accuracy for each class. Notice that class 8(Greek) 2(Chinese) 12(Korean) has higher accuracy maybe because of their more unique use and order of alphabets.

5 Efficiency gains

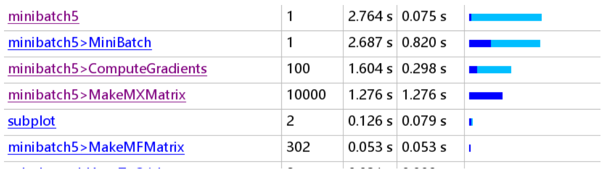
From profile reviewer we can see the bottleneck of the programe is MakeMX-Matrix function. After pre-computing MX matrix for the first layer and make them sparse, the total running time goes from 5.093s to 2.764s (with only 5 filters each layer and 100 update steps with mini-batches of size 100).

Creating $M^{input}_{x_j,k_2}$ instead of $M^{input}_{x_j,k_2,n_2}$ (and $\boldsymbol{g_j}$ matrix accordingly) and put it inside the loop, the total running time becomes 2.245s, resulting in 56% of upgrade in speed.



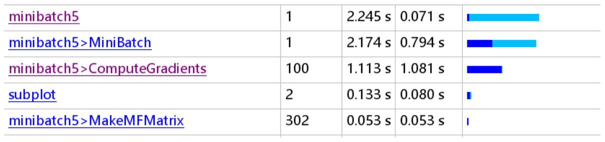
minibatch5	1	5.093 s	0.075 s	
minibatch5>MiniBatch	1	5.016 s	0.104 s	
minibatch5>ComputeGradients	100	4.646 s	0.414 s	
minibatch5>MakeMXMatrix	20000	4.201 s	4.201 s	
subplot	2	0.124 s	0.089 s	
minibatch5>MakeMFMatrix	302	0.053 s	0.053 s	

(a) Before



minibatch5	1	2.764 s	0.075 s	
minibatch5>MiniBatch	1	2.687 s	0.820 s	
minibatch5>ComputeGradients	100	1.604 s	0.298 s	
minibatch5>MakeMXMatrix	10000	1.276 s	1.276 s	
subplot	2	0.126 s	0.079 s	
minibatch5>MakeMFMatrix	302	0.053 s	0.053 s	

(b) After optimizing first layer MX



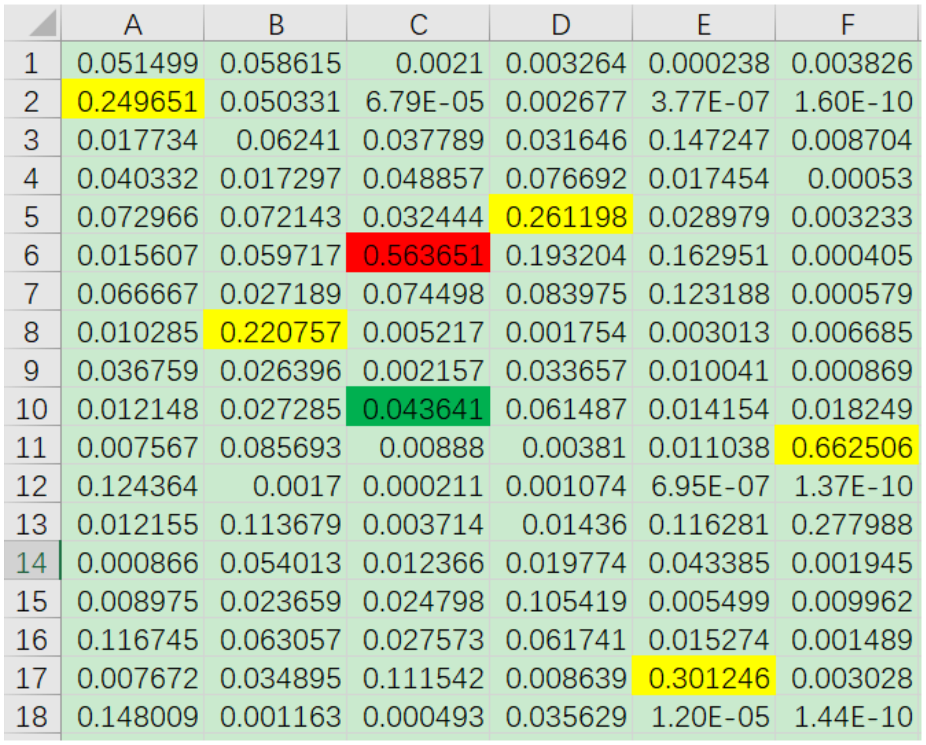
minibatch5	1	2.245 s	0.071 s	
minibatch5>MiniBatch	1	2.174 s	0.794 s	
minibatch5>ComputeGradients	100	1.113 s	1.081 s	
subplot	2	0.133 s	0.080 s	
minibatch5>MakeMFMatrix	302	0.053 s	0.053 s	

(c) After optimizing second layer MX

Figure 4: Speed of before and after optimization

6 Result

The probability vectors output by the best network applied to my surname(Xiong) and 5 of my friends' surnames, Gagos(Greek), Corbetta(Italy), Vincent(US/English), Alquezar(Spain), and Miyazaki(Japan) are shown in Figure 5. Except for the Italian name was said as French, the others are correct.



	A	B	C	D	E	F
1	0.051499	0.058615	0.0021	0.003264	0.000238	0.003826
2	0.249651	0.050331	6.79E-05	0.002677	3.77E-07	1.60E-10
3	0.017734	0.06241	0.037789	0.031646	0.147247	0.008704
4	0.040332	0.017297	0.048857	0.076692	0.017454	0.00053
5	0.072966	0.072143	0.032444	0.261198	0.028979	0.003233
6	0.015607	0.059717	0.563651	0.193204	0.162951	0.000405
7	0.066667	0.027189	0.074498	0.083975	0.123188	0.000579
8	0.010285	0.220757	0.005217	0.001754	0.003013	0.006685
9	0.036759	0.026396	0.002157	0.033657	0.010041	0.000869
10	0.012148	0.027285	0.043641	0.061487	0.014154	0.018249
11	0.007567	0.085693	0.00888	0.00381	0.011038	0.662506
12	0.124364	0.0017	0.000211	0.001074	6.95E-07	1.37E-10
13	0.012155	0.113679	0.003714	0.01436	0.116281	0.277988
14	0.000866	0.054013	0.012366	0.019774	0.043385	0.001945
15	0.008975	0.023659	0.024798	0.105419	0.005499	0.009962
16	0.116745	0.063057	0.027573	0.061741	0.015274	0.001489
17	0.007672	0.034895	0.111542	0.008639	0.301246	0.003028
18	0.148009	0.001163	0.000493	0.035629	1.20E-05	1.44E-10

Figure 5: Probability vector output of different names