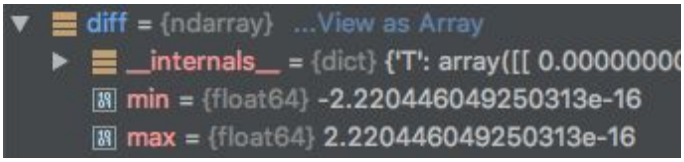
**Assignment 3 Option 2**
**Peiyang Shi**
**900420-8238**
**pyshi@kth.se**

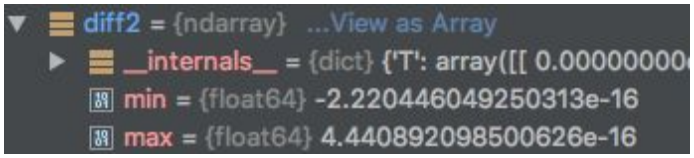
**Gradient Comutation**

There were two main components to the gradient computation: MX and MF matrix generation and the back propagation. MX and MF matrix generation were verified with the provided 'DebugInfo.mat'.

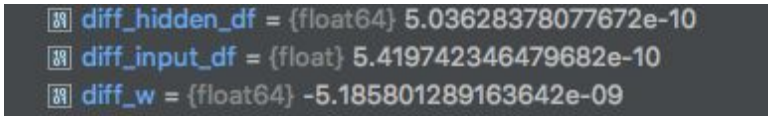
MF generator function was verified by passing in 'F' filter from DebugInfo, acquiring mf matrix, then take the dot product with x_input. The output is then compared with the expected output, vecS. The overall difference between was between -2.22e^-16 to 2.22e^-16. Such small differences were within acceptable range.




Similarly, MX was verified by passing 'x_input' into my MX generator function, acquiring `mx` matrix and dot with vectorized filter F, and the output was compared with the expected output vecS. Again, the difference was in very small.




Finally, gradients were computed with a numerical gradient method. Since this assignment involves very sparse matrices and small gradients, by simply comparing the mean diff is insufficient to tell if the gradient is correct. One way to check the gradient is by normalizing the gradient difference: (anal_grad - num_grad)/grad_order where the differences are normalized with the overall gradient. A second method is to take the absolute sum difference of two gradients, $\sum\{|anal\_grad|\} - \sum\{|num\_grad|\}$. Though this method will be less helpful in debugging but easier to implement, so I implemented the second method as a preliminary check. To my surprised, the gradient difference was small enough in the first run, thus I conclude that the gradient computation was correct.



The figure above showed the absolute sum of difference, and the error was sufficiently small that no further verifications were needed and can be assumed to be correct.

**Data balance**

In this assignments, the dataset provided was unbalanced, in the sense that certain classes have more data than others. Thus it will cause the system to overfit on data with more labels.

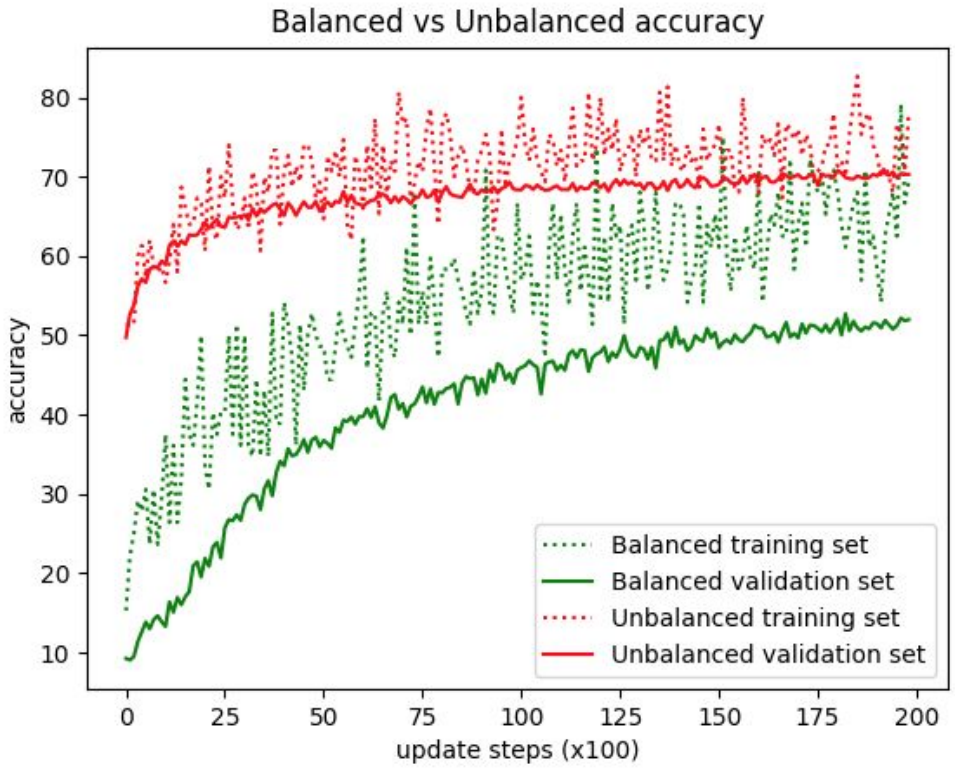
I chose to use a simple sampling technique to balance data, where each class is sampled evenly. In my case, taking the validation set into account, the training set was sampled evening across each class, with a total sample size of 180 for 18 classes, each class will get 10 samples.

A benchmark network was setup:
5000 update steps, 5x20 input conv layer, 3x20 hidden conv layer and a dense output layer.

With a balanced dataset, sample size of 68, the network achieved 55% accuracy on validation set.

Without balancing, we still used random sampling but randomly selected across all classes (to prevent overfitting on the first class). The end results was around 68% for the same hyperparameters. Although counter intuitive, we believe this is because the validation itself is unbalanced. By forcing the a balance in the dataset, we actually see a drop in the validation accuracy.

**Large unbalanced dataset**


Balanced vs Unbalanced accuracy
accuracy
update steps (x100)
Balanced training set
Balanced validation set
Unbalanced training set
Unbalanced validation set
10
20
30
40
50
60
70
80
0
25
50
75
100
125
150
175
200

The convergence of unbalanced data appears to be converging much faster than balanced data with the validation set. This is primarily believed that the nature of the data is heavily unbalanced, therefore forcing the system to recognize unbalanced data takes further fine tuning. In unbalanced data, more identifications tend to converge on Arabic, English and Russian, causing overfit on those samples. In the balanced case, though more validation data are biased towards russian, English and Arabic, the system was training in an unbiased manner and thus having higher error when predicting those three main languages.

**Efficiency gains**

There were two major improvements on the existing architecture.

First improvement is to preprocess x and reduce the MX1 dot operation. Because the input X is so sparse, we compressed it down to indices and created a compressed MX matrix from the compressed input X. Also, because X and MX are a binary matrices, we can avoid multiplication operations to improve the speed further. The end product is simply a set of indices to pick out of $G_{batch}$ matrix, and summing it up to replace the dot product. This precompression technique allows the time to drop from 220 seconds down to 5 seconds. For a batch of 100 samples, the update step dropped from 140 seconds down to 13 seconds.

Second improvement uses a dense representation of MX2 dotting a reshaped G matrix. This allows the update time to drop further from 13 seconds down to 4 seconds per update step.

Both are expected improvements, first improvement allowed us to represent a sparse matrix (99.8% elements are zeros) into a compact representation, reducing unnecessary multiplications with 0. Second improvement allowed us to represent a MX2 in a denser form, reducing ~82% sparsity.

**Friend's names:**

Five names were tried (and their correct labelling)
**Last name: Shi (Chinese)**
Arabic 8.75%
Chinese 42.51%
Czech 1.63%
Dutch 3.63%
English 3.58%
French 0.34%
German 1.55%
Greek 2.59%
Irish 1.27%
Italian 1.24%
Japanese 4.08%
Korean 9.37%
Polish 1.92%
Portuguese 0.17%
Russian 9.54%
Scottish 0.18%
Spanish 1.94%
Vietnamese 5.69%

**Last name: Herman (English)**
Arabic 0.01%
Chinese 0.03%
Czech 1.63%
Dutch 0.18%
English 54.18%
French 2.89%
German 2.33%
Greek 0.15%
Irish 9.09%
Italian 0.44%

Japanese 0.10%
Korean 0.04%
Polish 0.06%
Portuguese 0.18%
Russian 24.99%
Scottish 2.68%
Spanish 0.99%
Vietnamese 0.03%

**Last name: Leusink (Dutch)**
Arabic 0.18%
Chinese 0.00%
Czech 13.15%
Dutch 3.90%
English 16.16%
French 0.18%
German 6.40%
Greek 0.00%
Irish 0.47%
Italian 0.00%
Japanese 0.38%
Korean 0.00%
Polish 1.52%
Portuguese 0.00%
Russian 57.53%
Scottish 0.09%
Spanish 0.05%
Vietnamese 0.00%

**Last name: Silverstri (Italian)**
Arabic 0.00%
Chinese 0.00%
Czech 0.12%
Dutch 0.01%
English 1.03%
French 0.01%
German 0.05%
Greek 0.02%
Irish 0.01%
Italian 78.26%
Japanese 0.70%
Korean 0.00%
Polish 0.10%

Portuguese 0.01%
Russian 18.74%
Scottish 0.02%
Spanish 0.94%
Vietnamese 0.00%

**Last Name: Satoshi (Japanese)**
Arabic 8.79%
Chinese 0.00%
Czech 1.13%
Dutch 0.69%
English 0.88%
French 0.02%
German 0.00%
Greek 1.70%
Irish 0.11%
Italian 4.11%
Japanese 76.20%
Korean 0.00%
Polish 0.04%
Portuguese 0.00%
Russian 4.71%
Scottish 0.05%
Spanish 1.56%
Vietnamese 0.00%

It seems like that most of the names were classified correctly except for Leusink, which is Dutch but was recognized as Russian. This is believed that the data was heavily biased towards Russian, English and Arabic names. There was not enough training samples in dutch to classify non-classical dutch names.