Deep Learning — Homework 1

Anirudhan J Rajagopalan, Michele Ceru
ajr619, mc3784

February 11, 2016

1 Backpropogation

1. Using the chain rule:
$$\frac{\partial E}{\partial x_{\text{in}}} = \frac{\partial E}{\partial x_{\text{out}}} \frac{\partial x_{\text{out}}}{\partial x_{\text{in}}}$$
Using the expression of x_{out} given in the question we have:
$$\frac{\partial x_{\text{out}}}{\partial x_{\text{in}}} = \frac{\partial}{\partial x_{\text{in}}}\left(\frac{1}{1+e^{-x_{\text{in}}}}\right) = \frac{e^{-x_{\text{in}}}}{(1+e^{-x_{\text{in}}})^2}$$
Summing and subtracting 1 in the numerator:
$$\frac{\partial x_{\text{out}}}{\partial x_{\text{in}}} = \frac{1+e^{-x_{\text{in}}}-1}{(1+e^{-x_{\text{in}}})^2} = \frac{1}{1-e^{-x_{\text{in}}}}\Big[1 - \frac{1}{1-e^{-x_{\text{in}}}}\Big]$$
And using the expression of x_{out} given in the question:
$$\frac{\partial E}{\partial x_{\text{in}}} = \frac{\partial E}{\partial x_{\text{out}}} x_{\text{out}}[1 - x_{\text{out}}]$$

2. Using the expression of x_{out} given in the question:
$$\frac{\partial (x_{\text{out}})_i}{\partial (x_{\text{in}})_j} = \frac{\partial}{\partial (x_{\text{in}})_j}\Big(\frac{e^{-\beta (x_{\text{in}})_i}}{\sum_k e^{-\beta (x_{\text{in}})_k}}\Big)$$
In the case $i = j$:
$$\frac{\partial (x_{\text{out}})_i}{\partial (x_{\text{in}})_i} = \frac{-\beta e^{-\beta (x_{\text{in}})_i}(\sum_k e^{-\beta (x_{\text{in}})_k}) - e^{-\beta (x_{\text{in}})_i}(-\beta e^{-\beta (x_{\text{in}})_i})}{\sum_k e^{-\beta (x_{\text{in}})_k}} =$$
$$= -\frac{\beta e^{-\beta (x_{\text{in}})_i}}{\sum_k e^{-\beta (x_{\text{in}})_k}}\Big[1 - \frac{\beta e^{-\beta (x_{\text{in}})_i}}{\sum_k e^{-\beta (x_{\text{in}})_k}}\Big]$$
and substituting the expression of x_{out}:
$$\frac{\partial (x_{\text{out}})_i}{\partial (x_{\text{in}})_i} = \beta (x_{\text{out}})_i [1 - (x_{\text{out}})_i]$$
In the case $i \neq j$:
$$\frac{\partial (x_{\text{out}})_i}{\partial (x_{\text{in}})_j} = -\beta e^{-\beta (x_{\text{in}})_j}(\sum_k e^{-\beta (x_{\text{in}})_k})^{-2} e^{-\beta (x_{\text{in}})_i} =$$
$$= \beta \frac{e^{-\beta (x_{\text{in}})_j}}{\sum_k e^{-\beta (x_{\text{in}})_k}} \frac{e^{-\beta (x_{\text{in}})_i}}{\sum_k e^{-\beta (x_{\text{in}})_k}} = \beta (x_{\text{out}})_j (x_{\text{out}})_i$$
Writing the solution all together:
$$\frac{\partial (x_{\text{out}})_i}{\partial (x_{\text{in}})_j} = \begin{cases} \beta (x_{\text{out}})_i [1 - (x_{\text{out}})_i] & \text{if } i = j \\ \beta (x_{\text{out}})_j (x_{\text{out}})_i & \text{if } i \neq j \end{cases}$$

2 Torch (MNIST handwriting recognition)

2.0.1 Model file

The model file can be found at http://cs.nyu.edu/ ajr619/model.net

2.0.2 Github repo

The repository for the changes and result.lua is at
https://github.com/rajegannathan/Deep-Learning/tree/master/hw1

2.0.3 Best performing model

We got a best test set performance of 99.57% for a slightly modified convolutional net wherein we rotate the image by angles produced by normal distribution with 0 mean and 0.2 deviation. The randomly generated values are taken as the radians by which the image should be rotated.

2.0.4 Experiments

We ran a number of experiments starting with tweakign the options being passed to the test, train and model lua scripts.

SGD Batch size: Changing the batch size of SGD decreases the performance drastically. For batch size of 32, 64 and 128 the model's accuracy was in the range of 78%, 60% and 45% respectively.

Momentum and Learning rate: Theoretically, using a value of momentum greater than zero (and lesser than one) should help us jump over local minimum. When combined with proper value of learning rate we should be able to find a non local minimum. But from our experiments, we were unable to find a good combination of learning rate and momentum that gives a better performance than the default momentum of zero and learning rate of 10^{-3}. The performance drop was considerable and we were able to achieve accuracy in the range of around 30% only.

Optimization algorithm Another theoritical point which we considered was to use SGD in the starting few epoches and then switch the optimization algorithm to something better such as BFGS. Theoretically, SGD is supposed to help us skip saddle points and once we are near a minima, the other optimization algorithm could help us converge faster. So we tried experimenting by changing the optimization algorithm after epochs 5, 7, 15 and 30. We were able to get performance on par with the default SGD implementation (99.2% accuracy) and we didn't observe any improvements in performance.

We also tried with other optimization routines such as ASGD and CG. But the script was giving a number of exceptions and we didn't explore it further due to time constraints.

2.0.5 Figures

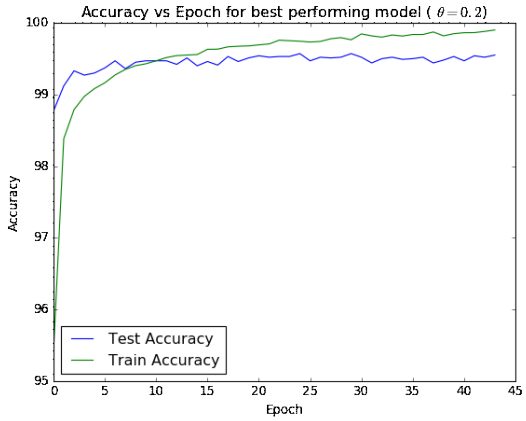

Figure 1: Best performance figures. We added a layer that rotates the images by random angles drawn from a normal distribution with zero mean and 0.2 standard deviation.

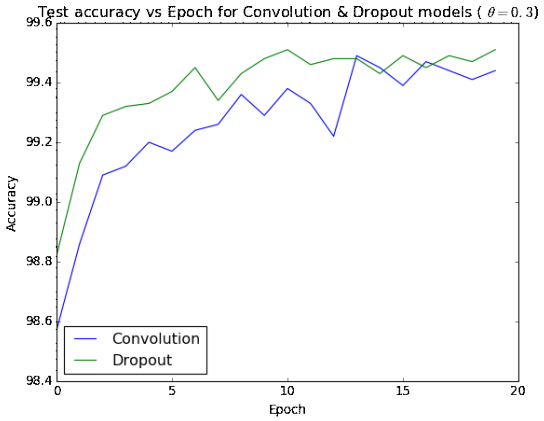

Figure 2: Adding convolutional layers and dropout model for the best performing model.