Formatting instructions for NeurIPS 2018

Stephan Grzelkowski
10342931

1 MLP backprop and NumPy implementation

1.1 Analytical derivation of gradients

(a)
(i)
Derivative of the loss in respect to the last layer activations $x^{(N)}$

$$\frac{\partial L}{\partial x^{(N)}}$$

Get scalar valued

$$(\frac{\partial L}{\partial x^{(N)}})_i = (\frac{\partial L}{\partial x_i^{(N)}})$$

$$(\frac{\partial L}{\partial x_i^{(N)}})$$

$$(\frac{\partial - \sum_j t_j log(x_j^{(N)})}{\partial x_i^{(N)}})$$

Since derivative only depends on x_j if j=i:

$$= -(\frac{t}{x})_i$$

In vector form since t is transposed:

$$-(\frac{t}{x})^T$$

Where division is applied elementwise

(ii)
Derivative of the softmax

$$(\frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}})_{ij}$$

I'll omit the layer indication for ease

$$= (\frac{\partial \frac{e_i^x}{\sum_j e^{x_j}}}{\partial \tilde{x}^{(N)}})_{ij}$$

We have to deal with two cases, $i = j$ and $i \neq j$. For the case $i = j$ the denominator becomes $e^{\tilde{x}_i}$. Through quotient rule we then get for the case $i = j$

$$= \tilde{x}_i(1 - \tilde{x}_i)$$

The case $i \neq j$:

$$= -\tilde{x}_i\tilde{x}_j$$

32nd Conference on Neural Information Processing Systems (NeurIPS 2018), Montréal, Canada.

Putting the two back together in vector form:

$$\tilde{x}I - (\tilde{x}\tilde{x}^T)$$

Where I is the identity matrix
(iii)
Derivative of the leaky-relu. Since the relu is applied element wise:

$$(\frac{\partial x^{(l)}}{\partial \tilde{x}^{(l)}})_i$$

$$(\frac{\partial f(\tilde{x}_i^{(l)})}{\partial \tilde{x_i}^{(l)}})$$

Where f(x) is the leaky relu:

$$f(x) = \begin{cases} x & \text{if } x >= 0, \\ \alpha x & \text{else} \end{cases}$$

The derivative of this is

$$(\frac{\partial f(\tilde{x}_i^{(l)})}{\partial \tilde{x_i}^{(l)}}) = \begin{cases} 1 & \text{if } x >= 0, \\ \alpha & \text{else} \end{cases}$$

With this we can build a vector dx that has values alpha and 1 according to above equations.

(iv)

The derivative of the units activation w.r.t. the previous' layer output:

$$\frac{\partial \tilde{x}^{(l)}}{=} \partial x^{(l-1)} (\frac{\partial \sum_j W_{ji}^{(l)} x_j^{(l-1)} + b_i^{(l)}}{\partial x_i^{(l-1)}})_{ji}$$

Since this is simply the derivative of a linear equation

$$= W^{(l)}$$

(v)

The derivative of the units activation w.r.t. to the biases:

$$(\frac{\partial \sum_j W_{ji}^{(l)} x_j^{(l-1)} + b_i^{(l)}}{\partial b_i})_{ji}$$

Since the first term of the function does not depend on b and there is further no constant the derivative is simply an array of ones:

$$= (1)_i$$

In vector form this becomes the Identity matrix

(vi)

$$(\frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}})_{ijk}$$

I got stuck on this one. I asked for one of the TA's help but I need to check again cause I couldn't really replicate the solution:
4D matrix with each entry in the 4th dim:

$$[\hat{e}_i x^{(l-1)T}]^{d_{(l)}}$$

(b)
I need to finish the implementation first. I haven't had the time yet to type in latex.
(c)
The loss becomes an average over the batch. And all the gradients have an added extra dimension that is the batch-size.

1.2 numpy implementation

I didn't have the time to finish it. Sorry, I hope there will still be partial points for some implementations.

2 Pytorch MLP

1. With the standard paramters we reach about 43 pecent accuracy on the test set. Although clearly above chance, there is still a lot of room for improvement

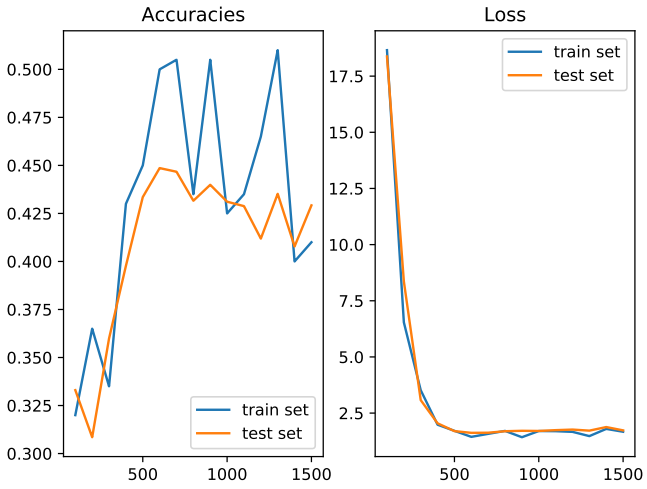

Figure 1: MLP results with standard parameters.

2. As a simple first step to try to increase performance i increased the number of layers, by adding a second linear layer with 100 Units. I only saw a couple percentage increase so I experimented further.

3. Next, I increased the layer sizes to both 512. Again, I saw a little improvements. However, since we now have much more parameters maybe we need more training epochs
4. I increased the number of training epochs. This mostly resulted in a small increase in the training accuracy. Not what we were looking. The training and testing error diverging is a sign of overfitting. Tired of changing one parameter by one I changed a whole bunch at once.
5. Why try leaky-relu if the standard relu is already so effective. I transformed the leaky-relu into a relu by setting the negative slope to 0. I also added a third layer 256 (aren't these power 2 numbers nice?!). After one run with these settings the training seemed a bit noisy (accuracy curves jumped around a bit). So I reduced the training rate. This yielded nice results with stable training accuracy around 53% There is a high discrepancy between training and testing accuracies. This overfitting might be caused by the increased "deepness" of the network. Might be solved through adding some form of regularization. Notably, the test loss even goes up: better stop earlier next time.
I tried a few more settings but there wasn't much improvement to be found.
All curves are a bit noisy since I didn't have the time to implement different testing on the entire set.

3 Batch Normalization

Convnet reached high accuracies very quickly. Towards the end of the training train and test sets seemed to diverge a bit.

4 Conv Net

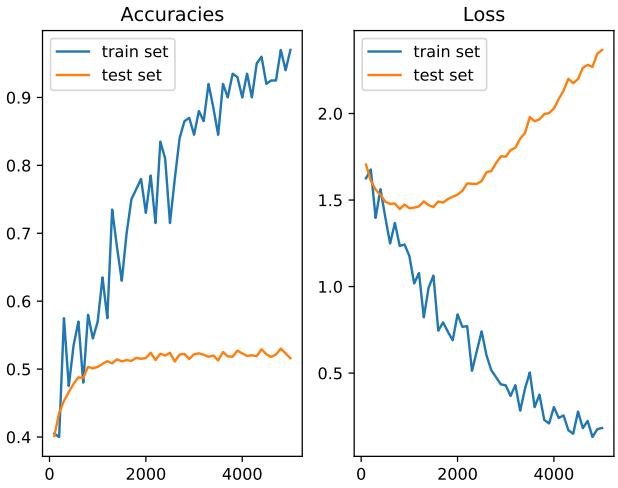

Figure 2: MLP results with [512,512,256] hidden units, 0.0001 learning rate, 0.0 negative slope leaky-relu, 5000 training epochs.

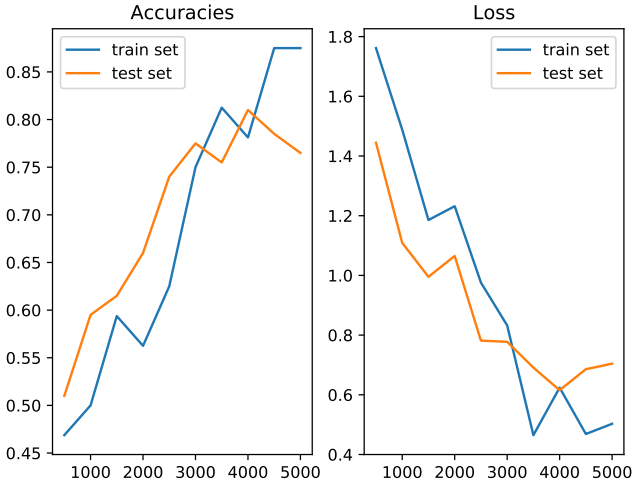

Figure 3: Convnetresults with standard parameters.