# DLE Homework 2 Backpropagation

Vojtěch Michal, michavo3

***Abstract*—The assigned[1] homework focuses on familiarization with PyTorch tensors and the algorithm backpropagation used for neural net training. Equality of numerically and analytically calculated gradient was verified. The test error of trained predictor is investigated.**

## I. TENSOR BASICS

See 1 for code relevant for this task. To ensure all tensors are created with *dtype = torch.float32*, it was required to explicitly specify it for tensors *w* and *b* that would default to int. When performing calculations on tensors with some arguments with *requires_grad = True*, results define *grad_fn* attribute as a function handle. Calculating derivative of *l* w.r.t. *y* yields $2(y-t) = 6$, as expected. When calling *l.backward()*, all required gradients are filled; for example *w.grad* changed from None to tensor with appropriate value (here 38). When constructing a loop dependency *w = w - 0.1 * w.grad*, the library is raising warnings about access to non-leaf tensors, whereas executing *w.data = w.data - 0.1 * w.grad* yields expected results.

```
import torch
import numpy as np

w = torch.tensor(1, dtype=torch.float, requires_grad
    =True)
x = torch.tensor(2.0)
t = torch.tensor(np.float32(3))
b = torch.tensor(4, dtype = torch.float32)

a = x + b
y = max(a*w, 0)
l = (y - t)**2 + w**2

l.backward()
w.data = w.data - 0.1 * w.grad

w = torch.tensor(1.0, requires_grad=True)
def loss(w):
    x = torch.tensor(2.0)
    b = torch.tensor(3.0)
    a = x + b
    y = torch.exp(w)
    l = (y-a)**2
    # y/=2
    del y,a,x,b,w
    return l
loss(w).backward()
```

Listing 1: Task 1 source code

In the last part of the code (around the definition of *loss* function, the gradient is computed correctly since no modification (or deletion) of *w* that occurs in the function's scope has influence on the object referenced by *w* in the outer scope. The *del* statement is redundant since all local variables are deleted regardless when they go out of scope. Calling in-place operations such as *y /= 2* results in another complaint from the library mentioning keyword ”versions” – another ”version” of the variable is present in the computation graph (required to evaluate the gradient) than is available among local variables.

## II. GRADIENT AND NETWORK TRAINING

Pytorch tensors and functions were used to implement *FFModel.score()* and *FFModel.mean_loss()* for a neural network using tanh activation function in the hidden layer. All parameters $W_1$, $w$, $b_1$ and $b$ were randomly initialized using uniform distribution on the interval $[-1, 1]$. Afterwards, numeric and analytic gradient calculation was compared to observe that the difference is of the order $O(\varepsilon^2)$, as illustrated by tables I and II for the case *dtype = torch.float64*, size of the hidden layer 500 and perturbation magnitudes $\varepsilon = 10^{-3}$ or $\varepsilon = 10^{-5}$, respectivelly.

| parameter | $W_1$ | $w$ | $b_1$ | $b$ |
|---|---|---|---|---|
| gradient error | 2.943e-10 | 2.469e-09 | 1.515e-11 | 3.714e-09 |

TABLE I: Difference between numeric and analytic gradient for $\varepsilon = 10^{-3}$.

| parameter | $W_1$ | $w$ | $b_1$ | $b$ |
|---|---|---|---|---|
| gradient error | 1.136e-11 | 3.375e-12 | 1.655e-11 | 1.345e-11 |

TABLE II: Difference between numeric and analytic gradient for $\varepsilon = 10^{-5}$.

Afterwards, the network was trained for various sizes of the hidden layer. From the prediction boundary before training – shown in Fig. 1 – predictors got to boundaries shown in Fig. 2, 3, 4 and 5 for hidden layer sizes 5, 10, 100 and 500, respectively. Note that all prediction boundaries are smooth in spite of the presence of redundant parameters.

Using the Hoeffding inequality for the case $\varepsilon = 0.01$, $(\Delta l)^2 = 1$ and $P(|\, R(h) - R_{T^m}(h)\, | > \varepsilon) < 0.01$, one derives that the minimal test sample size is $m = 26492$. Using this large test set, I obtained results in table III.

## III. TEST ERROR

Assuming the 0/1 loss, calculation of the empirical test error was implemented. Results from $10^4$ iterations of evaluation on a randomly generated test set of size $m = 1000$ are shown

[1]The homework assignment is available on https://cw.fel.cvut.cz/wiki/courses/bev033dle/labs/lab1_backprop/start

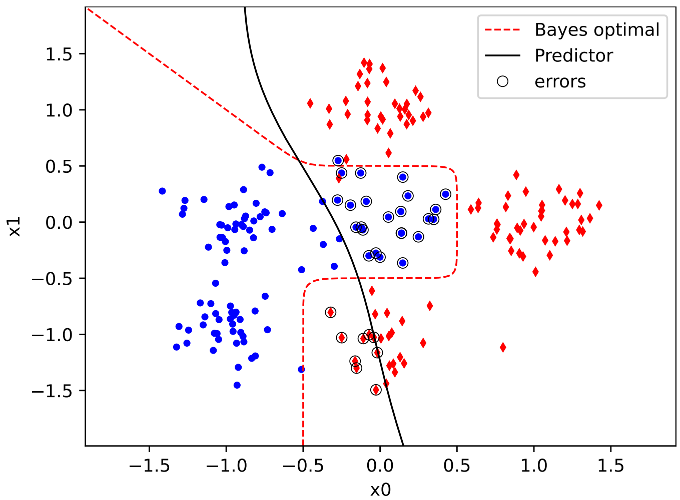


Fig. 1: Prediction boundary before any training for hidden layer of size 500

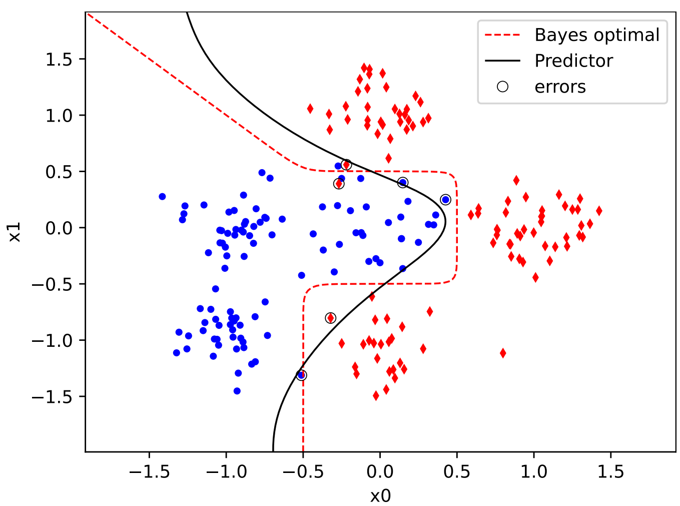


Fig. 2: Prediction boundary for hidden layer of size 5

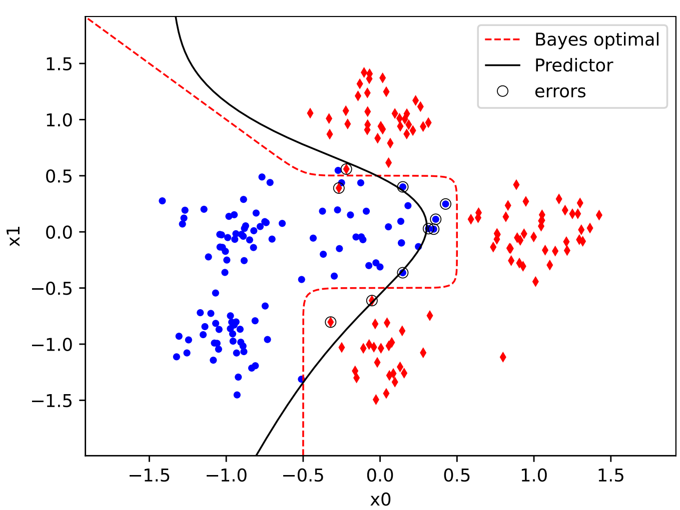


Fig. 3: Prediction boundary for hidden layer of size 10

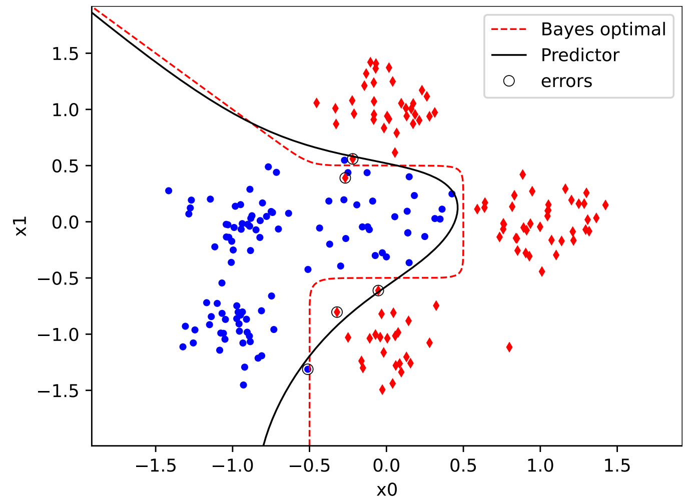


Fig. 4: Prediction boundary for hidden layer of size 100

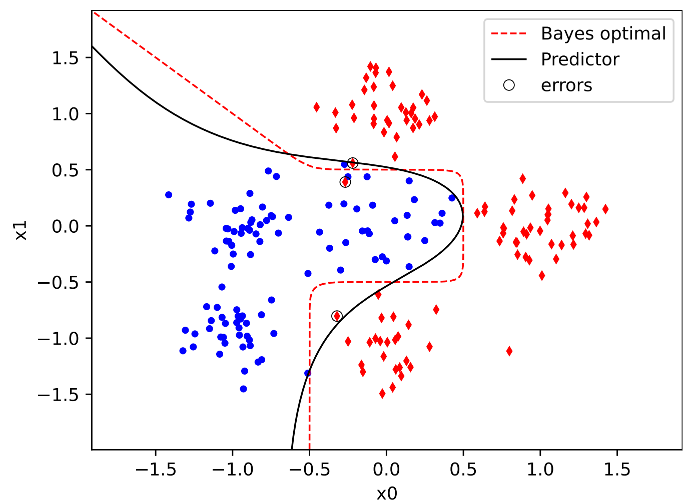


Fig. 5: Prediction boundary for hidden layer of size 500

together with results of the bootstrap method in Fig. 6. As expected, the result of bootstrap slightly underestimates the error. Widths of individual histogram bins were chosen as 0.2 % (according to the formula $2 \cdot 100m^{-}1$). The figure also contains the confidence interval obtained by solving the Hoeffding inequality for $\alpha = 0.9$ and the given $m$.

| hidden layer size | test error [%] | generalization gap [%] |
|---|---|---|
| 5 | 3.7672 | 0.7672 |
| 10 | 5.2922 | 0.2922 |
| 100 | 3.2651 | 0.7651 |
| 500 | 2.8763 | 0.8763 |

TABLE III: Test error for various predictors using proper $m$ to obtain confidence more than 99 %.

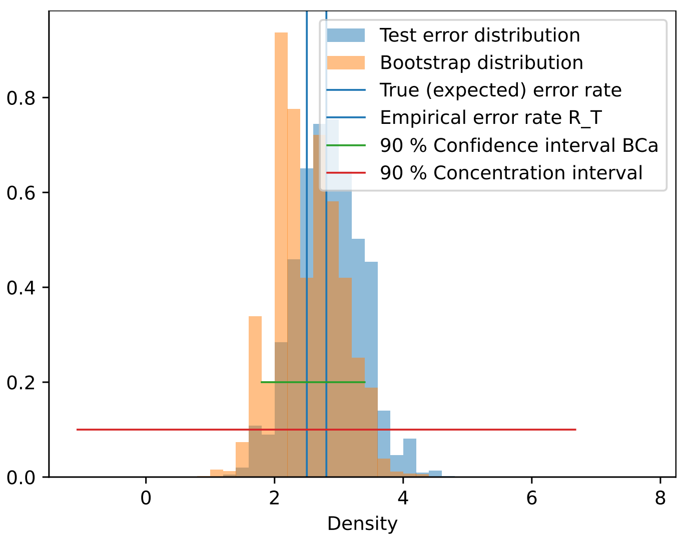


Fig. 6: Statistical visualization of the empirical test error and results of the bootstrap method