Quick links: Schedule [https://intranet.fel.cvut.cz/cz/education/rozvrhy-ng.B232/public/html/predmety/61/70/p6170206.html] | Forum [https://cw.felk.cvut.cz/forum/forum-1874.html] | BRUTE [https://cw.felk.cvut.cz/brute/teacher/course/1595] | Lectures [https://cw.fel.cvut.cz/b232/courses/bev033dle/lectures] | Labs [https://cw.fel.cvut.cz/b232/courses/bev033dle/labs/start]

Lab 2: Backpropagation, Computational Graph

In this lab we will get familiar with PyTorch basics:

- operations with tensors
- computation graph and backpropagation

We will also implement and train a simple neural network using PyTorch tensors. In subsequent labs we will use higher level PyTorch classes and methods, which essentially encapsulate these tensor operations. For simplicity, we continue using the Gaussian mixture model from Lab 1.

Use the provided template [/wiki/_media/courses/bev033dle/labs/lab1_backprop/template.zip].

Template last updated: 7.3.2024 at 16:02.
plot_boundary – legend was incorrect, the curve was sometimes incorrect.
renamed files for consistency with lab1 (same toy_model.py)

Part 1. Tensor basics (2p)

Tensors in Pytorch are like multidimensional NumPy arrays with all standard operations available and following a very similar syntax. They also support many additional functions needed in NNs. Most importantly, whenever an operation on tensors is performed the resulting tensor also remembers from which operation it was created and what the operands were – this allows to dynamically track the computation graph and perform backpropagation. Also, the data and operations can be carried in CPU or GPU depending on the `device` attribute of the tensor.

We propose the following simple exercise to get acquainted with tensors. To start with,

- Install PyTorch locally on your computer pytorch.org/get-started [https://pytorch.org/get-started/locally/]
- In this lab we will need "only" the Tensor class documentation Tensor [https://pytorch.org/docs/stable/tensors.html]

Now a small task:

1. Let's start from the following code snippet defining several scalar tensors

```
import torch
import numpy as np

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

Other ways of constructing tensors are detailed in creation ops [https://pytorch.org/docs/stable/torch.html#tensor-creation-ops].

Check which data type your tensors have by inspecting their `dtype` attribute. Modify the code so that all tensors would be of the type `torch.float32` . As a rule, for backpropagation and parameter optimization you would want this data type.

1. Set `w.requires_grad = True` (works only for a floating point tensor)

2. Compute $a = x + b, y = \max(aw, 0)$ and $l = (y - t)^2 + w^2$ using operations on tensors. Result of every operation tracks the history – computation graph but only as long as some tensors which require a gradient computation are involved. Inspect 'grad_fn' attribute of y, l and a. Draw the DAG of this computation on a paper. Read more about autograd mechanics [/wiki/lib/exe/fetch.php?tok=713bdf&media=https%3A%2F%2Fpytorch.org%2Fdocs%2Fstable%2Fnotes%2Fautograd.html#autograd-mechanics].
3. Compute and print derivative of l w.r.t. y by using torch.autograd.grad [/wiki/lib/exe/fetch.php?tok=6ca76c&media=https%3A%2F%2Fpytorch.org%2Fdocs%2Fstable%2Fgenerated%2Ftorch.autograd.grad.html#torch.autograd.grad].
4. Compute derivative w.r.t all leaf variables using `l.backward()` and inspect `w.grad`. Note that the default behaviour is to accumulate gradients rather than populate them with "fresh" ones. They therefore need to be reset manually by e.g. `w.grad=None` when needed.
5. With the gradient computed as above, make a gradient descent step: $w = w - 0.1\nabla_w l$ using the `.data` attribute of the weight tensor w to perform the assignment or using tensor w in the no_grad() context (see torch.no_grad [/wiki/lib/exe/fetch.php?tok=b9c664&media=https%3A%2F%2Fpytorch.org%2Fdocs%2Fstable%2Fgenerated%2Ftorch.no_grad.html]). What would happen if the step was implemented like `w = w - 0.1*w.grad` and w would be subsequently used in further computations and differentiated?
6. In the following code

```
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()
```

   will the gradient in w be computed correctly? Can you explain what is stored in the computation graph and why the 'del' statement is actually redundant? Can you explain why the commented out in-place operation 'y/=2' that modifies y (in contrast to deleting it), leads to an error if uncommented?

Part 2. Gradient and Network Training (5p)

Using PyTorch tensors only (not Modules yet) implement a neural network with `input_size` inputs, one hidden layer with `hidden_size` units and tanh/ReLU activations and, finally, the logistic regression model in the last layer, i.e. a linear transform and the logistic sigmoid function S. Formally, the network is specified as

$$\phi(x) = \tanh(W_1 x + b_1)$$
$$s(x) = w^{\mathrm{T}}\phi(x) + b$$
$$p(y = 1 \,|\, x; \theta) = \sigma(s(x)),$$

where σ is the logistic sigmoid function and θ denotes all parameters, i.e. $\theta = (W_1, b_1, w, b)$. Use only the Tensor class and functions of tensors (i.e. not NLLLoss class). As in the previous lab, x represents a matrix of all data points and has size $[N \times d]$, where N is the number of data points and d is the network input size. Therefore hidden layer output should be of size $[N \times \text{hidden\_size}]$.

Given the training set $(x_i, y_i)_{i=1}^N$, where $y_i \in \{-1, 1\}$, the training loss is the negative log-likelihood (NLL) of the data:

$$\mathrm{L}(\theta) = -\frac{1}{N}\sum_{i=1}^{N} \log p(y_i \,|\, x_i; \theta) = -\frac{1}{N}\sum_{i=1}^{N} \log \sigma(y_i s(x_i)).$$

Note, because exponents can quickly overflow, in real applications a numerically stable implementation of logarithm of sigmoid function is needed. There are `logsigmoid`, `log_softmax`, `logsumexp`, `nll_loss` functions available in PyTorch. In this lab it should not be an issues, but we encourage you to use them.

Step 1

Initialize W_1 and b_1, w, b randomly, e.g. uniformly in $[-1, 1]$. Run the forward pass evaluating the loss for the whole training set and run the backward pass to accumulate gradients.

Step 2

We will now check that gradients indeed well approximate the function behaviour when the weights are changed slightly from their original values. Consider varying a parameter vector $w \in \mathrm{R}^n$ (here abstractly, any tensor amongst our network parameters). The model has several parameter vectors, considering one at a time will help to isolate errors. The following method is explained in detail in *solved Assignment 1 (Gradient checking)* in examples.pdf [/wiki/_media/courses/bev033dle/labs/examples.pdf]). We will compare the analytically computed derivative and a numerically computed derivative. For this purpose we create a randomized test, checking that directional derivatives in some random direction match. Let $u \in \mathrm{R}^n$, $\|u\| = 1$ be a random direction. It can be generated by generating a vector from $\mathrm{Uniform}[-1, 1]^n$ and normalizing it. Then the directional derivative along u can be computed numerically using the symmetric finite difference:

$$g(\varepsilon) = \frac{\mathrm{L}(w + \varepsilon u) - \mathrm{L}(w - \varepsilon u)}{2\varepsilon},$$

where $\varepsilon \ll 1$. The true directional derivative is the limit

$$g = \lim_{\varepsilon \to 0+} g(\varepsilon).$$

If we have correctly computed the gradient $\nabla_w \mathrm{L}$, there should hold $g = \langle \nabla_w \mathrm{L}, u \rangle$. We therefore expect

$$| g(\varepsilon) - \langle \nabla_w \mathrm{L}, u \rangle | = O(\varepsilon^2).$$

The $O(\varepsilon^2)$ notation means that as ε approaches zero, the absolute difference on the left hand side is asymptotically $c\varepsilon^2$ for some constant c. The step size ε should be chosen much smaller than 1 (order of coordinates of w at initialization) but large enough so that the numerical precision in the finite difference is still sufficient. Try to verify that this limit holds numerically by testing with $\varepsilon = 10^{-2}, 10^{-3}, 10^{-4}$.

Report the results of your gradient verification (1) for all model parameters. Example for $\varepsilon = 10^{-4}$, data type 'torch.float32' and number of 'hidden_size = 500':

```
Grad error in w1: 0.0007544
Grad error in b1: 0.000235
Grad error in w2: 0.0004883
Grad error in b2: 9.517e-05
```

Same with $\varepsilon = 10^{-5}$ and 'torch.float64' data type we get:

```
Grad error in w1: 6.669e-11
Grad error in b1: 5.313e-11
Grad error in w2: 4.548e-11
Grad error in b2: 2.979e-11
```

confirming $O(\varepsilon^2)$ approximation. When implementing this gradient verification make sure that after you perturb the weights for computing $\mathrm{L}(w + \varepsilon u)$, you then restore them back to the original value.

Step 3. Network training

Implement gradient descent step with constant step length ε in all network parameters θ:

$$\theta^{t+1} = \theta^t - \varepsilon \nabla_\theta \mathrm{L}(\theta^t).$$

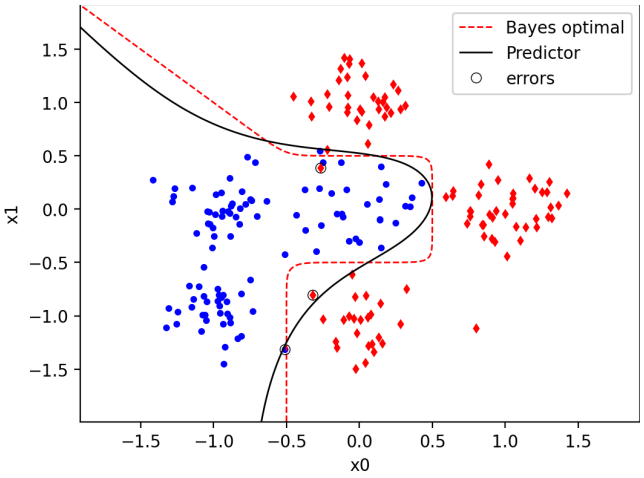
Train the network, by performing several gradient descent steps, following the template. Verify that the training loss improves during the training.

1. Train a network with `hidden_size` = 5 on a training set of 200 points with learning rate 0.1 for 100 epochs (set in the template).
2. Plot the resulting classifier decision boundary with the help of `G2Model.plot_predictor` the "interface" Predictor class in the template.
3. Repeat the experiment increasing the number of hidden units to 10, 100, 500. What we want to observe is that the classifier decision boundary fits the data better but stays smooth despite the abundant amount of parameters.

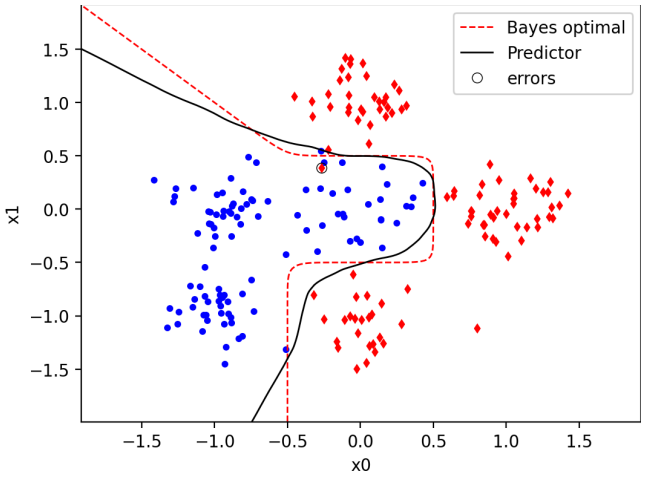
Draw a test set from the ground truth model. Compute and report the empirical estimate of the test error and the generalization gap (difference between training and test error). Use the Hoeffding inequality (see lecture 2: Example 1) to select the test set size so that the probability of being off in the estimate of the test accuracy by more than 1% is less than 0.01.

Report: test set size, test error rate and classification boundary plots for `hidden_size` = 5, 100, 500. Example classification boundary for 500 hidden units:

500 tanh units



500 ReLU units



Part 3. Test Error (3p)

After we trained our probabilistic predictive model $p(y \mid x)$, we want to use it for decision making. Let $l(y, y') = [[y \neq y']]$ be the 0-1 loss, penalizing the prediction y' if the true class was y. Recall that the optimal classification decision strategy according to this loss function and the predictive model $p(y \mid x)$ is

$$h(x) = \arg\max_y p(y \mid x) = \arg\max_y \sigma(y s(x)) = \mathrm{sign}(s(x)).$$

The *risk* is then the error rate of the model. Assume we have a test set $T = (x_i, y_i)_{i=1}^m$ of a fixed size $m = 1000$. The empirical risk (empirical test error) is

$$R_T = \frac{1}{m} \sum_i [[y_i \neq h(x_i)]].$$

As a function of a randomly drawn test set, R_T is itself random. It is important to keep that in mind, when reporting the test performance of a model or deciding which of several proposed models is better based on their test performance.

In this part we will inspect in more detail the randomness of the test error and use bootstrap method and concentration inequalities to construct confidence intervals.

True error distribution

The result of evaluation on a randomly drawn test set of this size is random. Repeat generating the test set and evaluate accuracy of your model 10000 times and record all evaluation results. Plot the histogram of the empirical test error from these 10000 trials. It gives us a clear picture, how the measured test performance may fluctuate due to the randomness of the test set selection. The mean of this distribution is the true test error rate and our empirical test error rate from one test set can be off.

Bootstrap

In real applications we don't have the possibility to draw independent test sets multiple times. Now suppose we have only a single fixed test set T. Let $e_i = 1$ if the network makes an error in classifying test point x_i and zero otherwise, i.e.

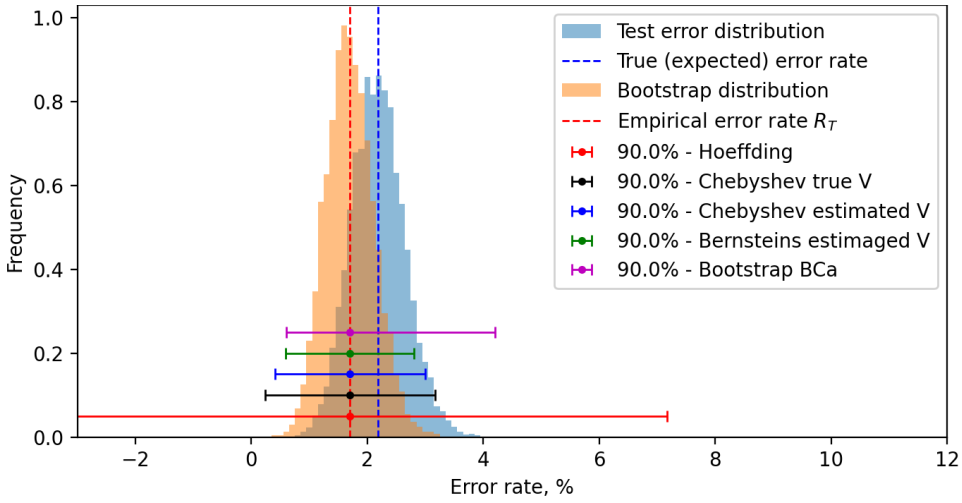
$$e_i = l(y_i, h(x_i)) = [[y_i \neq h(x_i)]]$$

The empirical test error rate is then the mean statistics of e_i: $R_T = \frac{1}{m}\sum_{i=1}^{m} e_i$. Bootstrap method draws the data from our empirical population multiple times with replacement. This mimics drawing test sets multiple times. Evaluating the mean statistics on such multiple ``bootstrap samples'', we can estimate the distribution of the statistic.

Use the bootstrap [https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.bootstrap.html] function from SciPy (version > 1.10.0) on the array of errors e:

```
res = scipy.stats.bootstrap((err.numpy(),), np.mean, confidence_level=alpha, method='BCa')
```

It will return the bootstrap distribution and a confidence interval. Plot the histogram of this bootstrap distribution and the confidence interval for confidence level $\alpha = 0.9$.



Issue discovered: the plot shows histograms as densities while the y-axis is named frequency. Use any in your solution.

Because the error rate R_T takes on a discrete set of values $\frac{1}{m}, \frac{2}{m}, \ldots 1$, you want each bin to contain the same number of discrete levels. Therefore for the histogram of error rate in percents, $R_T 100\%$, appropriate bin size is (100/m)*k for a natural k, e.g. k=1. Unlike the true distribution, it is centered on the empirical test error rate, not on the true test error rate. So from run to run, it may be shifted to the left or to the right, but its shape should be more or less stable, which allows one to estimate the uncertainty, i.e. the confidence interval that bootstrap function outputs.

Concentration Inequalities

As above, we have only a single fixed test set T of size m. We want to estimate the confidence intervals on the empirical test error R_T. In the Hoeffding inequality, set the confidence level $\alpha = 0.9$, i.e. the desired probability that the empirical risk is within ε from the true risk, $\mathrm{P}(|R(h) - R_T(h)| < \varepsilon) = \alpha$.

Solve for ε given α and m. Include the formula into the report. Print the test error in the format $R_T \pm \varepsilon$. Graphically display the symmetric confidence interval $[R_T - \varepsilon, R_T + \varepsilon]$ on top of the above histogram (centered on empirical error rate).

Derive the confidence interval (find ε) from Chebyshev inequality similarly to the above. Include the formula into the report. Estimate the variance v of test errors as the sample variance of the vector e. Print (and optionally display graphically) the confidence interval.

Bonus: Bernstein's inequality

Notice that Hoeffding inequality holds when the random variables e_i are bounded, i.e. we know $\Delta l = 1$. Chebyshev inequality does not need this assumption and so is applicable to unbounded errors, i.e. in regression, but the variance of the errors must be known. The Bernstein's inequality below makes use of both assumptions: bounded range of variables and known variance and so allows to obtain a tighter confidence interval.

Let $Z_1, \ldots Z_n$ be n independent random variables such that $|Z_i| \leq c$. Then, for $t \geq 0$,

$$\mathrm{P}\left(\left|\frac{1}{n}\sum_{i=1}^{n} Z_i - \frac{1}{n}\sum_{i=1}^{n} \mathrm{E}\left[Z_i\right]\right| \geqslant t\right) \leqslant 2\exp\left(-\frac{nt^2}{2\sigma^2 + 2ct/3}\right),$$

where $\sigma^2 = \frac{1}{n}\sum_i \mathrm{V}[Z_i]$. Apply this inequality to obtain confidence interval on the empirical error rate.

Hint: Z_i are our e_i above, σ^2 can be estimated as sample variance of e. To find confidence interval, you will need to solve a quadratic equation in t.