Reinforcement Learning WS 2018/19
Instructor: Sebastian Blaes `<sblaes@tuebingen.mpg.de>`
Due-date: 02.11.2018 (hand in at beginning of recitation session)
Exercise Sheet 2
October 26, 2018

The goal of this homework is: 1) to get more confortable with computing update gradients, 2) to become more fluent in programming in Python, and 3) to get hands-on experience with supervised learning and training neural networks. Next week we will make use of it for something more meaningful.

1 Backpropagation for Neural Networks

(a) Calculate the update equation for gradient descent for one neuron with squared loss:

$$\hat{y}_i = \tanh(w^\top x_i + b) \tag{1}$$

$$L(w, b) = \frac{1}{2}\sum_i (\hat{y}_i - y_i)^2, \tag{2}$$

where $\hat{y}_i$ is the output of the neuron. Compute $\frac{\partial L(w,b)}{\partial w}$ and $\frac{\partial L(w,b)}{\partial b}$ and write down the update-equation for w and b, i.e. the equation of the form $w \leftarrow w + \epsilon \ldots$ where ϵ is the learning rate or stepsize.

Hint: $\tanh'(z) = 1 - \tanh(z)^2$, *remember that we perform a gradient descent.*

(b) Now we turn to a neural network described by the following equation:

$$l_i = \tanh(W_1^\top x_i + b_1) \tag{3}$$

$$\hat{y}_i = W_2^\top l_i + b_2, \tag{4}$$

where l_i is the output of the hidden layer and $\hat{y}_i$ is the output of the network for input x_i.

(a) Make a sketch of the network architecture.

(b) Calculate the update-equation for the neural network with for the squared-loss:

$$L(W_1, W_2, b_1, b_2) = \frac{1}{2}\sum_i (\hat{y}_i - y_i)^2 \tag{5}$$

That means: compute $\frac{\partial L}{\partial W_1}$, $\frac{\partial L}{\partial W_2}$, $\frac{\partial L}{\partial b_1}$, and $\frac{\partial L}{\partial b_2}$ and write down the update-equations.

2 Implement a neural network

Your task is to complete the implementation of a neural network and train on a given dataset. You get a jupyter notebook with some basic code to get started: `Exercise-2.ipyth` and a data file: `airfoil_self_noise.dat` . Fill in the missing bits to implement a fully functional network with trainable weights.

(a) write the `train_with_square_loss` function. For that you need the update-equations computed above. *Hint: In math we prefer to compute with matrices in a right to left fashion: Wx where x is a column vector transforms x. In* `numpy` *it is often more convenient to use xW where x is a row vector and W is transposed, so we compute from left to right. Thus, you need to transfer your math into that format for the implementation.*

If you can't find the derivate after trying hard, you will find a solution for one term at the end to get you going.

(b) First train using the entire dataset in one update step (batch gradient descent). Run for 10000 steps with learning rate 0.00005. Check the learning progress and validation error. What happens if you train longer? (at least to 50000)

(c) Try changing the learning rate and continue training. What are the performances you get?

(d) Implement a stochastic gradient descent (using a mini-batch size of 16 points). Use a learning rate of 0.0002 and train for the same time (number of examples feed to the network) What do you observe at the training curve and the final performance

(e) * Add a momentum term to the gradient

(f) * Add a weight regularization $\lambda\|W\|^2$ to the training and try a bigger network with different regularization constants.

Hint for 2a: $\frac{\partial L}{\partial W_1} = \delta_1 x^\top$, *where* $\delta_1 = \tanh'(z) W_2^\top(\hat{y} - y)$ *with* $z = W_1 x + b_1$