Homework 02

IANNwTF

November 2, 2022

!! Reminder: Each member of your group is expected to hand in your solutions via the homework submission form!!

This week's deadline is *Saturday November 12, 23:59*.

Submit your homework via https://forms.gle/ApAZ5ubY8ewgNmJA9

As homework 1 did not feature any required submissions, no homework review is necessary this week.

Contents

1 Recap

If you followed along with our instructions to install conda last week, *remember to activate your virtual environment*! Make it a habit to check which environment you are working in whenever you are about to execute your code or to install a new package.

The following questions are only there to provide you with some food for thought as you work on your assignment, and to help you recap what you learned so far. You do not have to hand in your answers, but feel free to share and discuss them with other groups.

- What is the purpose of an activation function?
- How is the sigmoid function defined? What does it look like when plotted as a graph? What about ReLu?
- What is the derivative of the sigmoid function? What is the derivative of ReLu?
- Why would one choose sigmoid or ReLU over the step function when deciding on which activation function to use?

2 Assignment: Multi-Layer Perceptron

The universal approximation theorem states that a feed-forward artificial neural network can learn to approximate any continuous function with just one hidden layer, given enough units. This week you are going to implement a multi-layer perceptron ('MLP') and see if you can teach it some math.

2.1 Task 01 - Building your data set

You are going to need some data to train your network on. Use NumPy[1] to:

1. Randomly generate 100 numbers between 0 and 1 and save them to an array `'x'`. These are your *input values*.
2. Create an array `'t'`. For each entry `x[i]` in `x`, calculate `x[i]**3-x[i]**2` and save the results to `t[i]`. These are your *targets*.

Optional: Plot your data points along with the underlying function which generated them.

[1] https://numpy.org/doc/stable

2.2 Task 02 - Perceptrons

Next, implement a simple layer using perceptrons. The layer should be fully-connected, i.e. each unit in the layer should be connected to every unit in the preceding layer[2]. You are going to need several of these for your MLP, so you should write a `Layer` class to be able to easily instantiate as many of them as you need. Your class should have:

1. A constructor
   - The constructor should accept an integer argument `n_units`, indicating the number of units in the layer.
   - The constructor should accept an integer argument `'input_units'`, indicating the number of units in the preceding layer.
   - The constructor should instantiate a bias vector and a weight matrix of shape (n inputs, n units). Use random values for the weights and zeros for the biases.
   - instantiate empty attributes for layer-input, layer preactivation and layer activation
2. A method called `'forward_step'`, which returns each unit's activation (i.e. output) using ReLu as the activation function.
3. A method called `backward_step`, which updates each unit's parameters (i.e. weights and bias).
   - Compute the gradients using

     $$\frac{\partial \mathcal{L}}{\partial W_l} = {input_l}^T(\sigma'(preactivation) \circ \frac{\partial \mathcal{L}}{\partial activation}) \tag{1}$$

     and for the layer's bias vector

     $$\frac{\partial \mathcal{L}}{\partial b_l} = \sigma'(preactivation) \circ \frac{\partial \mathcal{L}}{\partial activation} \tag{2}$$

     where the pre-activation is the output of the layer's matrix multiplication. The activation is the output of the layer's activation function (and thus the next layer's input). $\frac{\partial \mathcal{L}}{\partial activation}$ must be obtained from layer l+1 (or directly from the loss function derivative if l is the output layer).

     $\circ$ here denotes an element-wise or broadcasted element-wise product. AB on the other hand denotes a matrix multiplication of A and B. ${input_l}^T$ is the transpose of the layer's input. It makes sense to store layer activations, pre-activations and layer input in attributes when doing the forward computation of a layer.

[2] You can think of a network's input a an 'input layer'

On top of the gradients w.r.t. the layer's parameters, we need the layer to compute the gradients w.r.t. its input (the activation of layer l-1).

$$\frac{\partial \mathcal{L}}{\partial input_l} = (\sigma'(preactivation) * \frac{\partial \mathcal{L}}{\partial activation})W_l{}^T) \tag{3}$$

- After having computed the gradients of the loss w.r.t. the layer's input and its weights and bias, update the layer's parameters using

$$\theta_{new} = \theta_{old} - \eta \nabla_\theta \mathcal{L} \tag{4}$$

where η is the learning rate (should generally be smaller than 0.05)

2.3 Task 03 - Multi-Layer Perceptron

Create a `MLP` class which combines instances of your `Layer` class into into a MLP. Implement two methods:

- A `forward_step` method which passes an input through the entire network
- A `backpropagation` method which updates all the weights and biases in the network given a loss value.

2.4 Task 04 - Training

To compute the network's loss, you should use the mean squared error:

$$\mathcal{L}_{MSE} = \frac{1}{2}(y - t)^2,$$

where y is the output of your network and t is the intended target.

1. Create a MLP with **1 hidden layer** consisting of **10 units** which all receive a single input, and an **output layer with just 1 unit**.
2. Train your MLP on your data set for 1000 epochs:
   - Once per epoch, show every data point in your data set to the MLP one at a time.
   - For each data point, have the MLP perform a forward step using the input value and then propagate the error backwards through the network.
   - After each data point, record the loss for later.

2.5 Task 05 - Visualization

Visualize the training progress using Matplotlib. Plot the epochs on the x-axis and the average loss on the y-axis.

If training was successful, the average loss should approach 0, though it is fine if it doesn't! There are many things that can go wrong during training, and there is much room for improvement when it comes to your neural network structure and optimization... But more on that soon!

3 Bonus

!! This task is entirely voluntary; there will be no reward for doing it!!

Try out different mathematical functions to generate your target values. Does your network perform equally well for all of them? Does it help if you increase the size of your data set? What if you use equally spaced input values instead of random values? What about adding more units to the hidden layer? Try a chaotic function, e.g. $f(x) = sin(\frac{1}{x})$