

## Homework 4: Decision Trees, Boosting, and Neural Networks

**Due:** Tuesday, Dec 3rd, 2024 at 11:59AM EST

**Instructions:** Your answers to the questions below, including plots and mathematical work, should be submitted as a single PDF file. It's preferred that you write your answers using software that typesets mathematics (e.g. LaTeX, LyX, or MathJax via iPython), though if you need to you may scan handwritten work. You may find the minted package convenient for including source code in your LaTeX document. If you are using LyX, then the listings package tends to work better. **The optional problems should not take you too much time and help you navigate the material, consider taking a shot at them.**

---

### 1 Decision Tree Implementation

In this problem we'll implement decision trees for both classification and regression. The strategy will be to implement a generic class, called `Decision_Tree`, which we'll supply with the loss function we want to use to make node splitting decisions, as well as the estimator we'll use to come up with the prediction associated with each leaf node. For classification, this prediction could be a vector of probabilities, but for simplicity we'll just consider hard classifications here. We'll work with the classification and regression data sets from previous assignments.

1. Complete either the `compute_entropy` or `compute_gini` functions.
2. Complete the class `Decision_Tree`, given in the skeleton code. The intended implementation is as follows: Each object of type `Decision_Tree` represents a single node of the tree. The depth of that node is represented by the variable `self.depth`, with the root node having depth 0. The main job of the fit function is to decide, given the data provided, how to split the node or whether it should remain a leaf node. If the node will split, then the splitting feature and splitting value are recorded, and the left and right subtrees are fit on the relevant portions of the data. Thus tree-building is a recursive procedure. We should have as many `Decision_Tree` objects as there are nodes in the tree. We will not implement pruning here. Some additional details are given in the skeleton code.
3. Run the code provided that builds trees for the two-dimensional classification data. Include the results. For debugging, you may want to compare results with sklearn's decision tree (code provided in the skeleton code). For visualization, you'll need to install `graphviz`.
4. Complete the function `mean_absolute_deviation_around_median` (MAE). Use the code provided to fit the `Regression_Tree` to the krr dataset using both the MAE loss and median predictions. Include the plots for the 6 fits with max depth varying from 1 to 6.

### 2 Ensembling

Recall the general gradient boosting algorithm, for a given loss function  $\ell$  and a hypothesis space  $\mathcal{F}$  of regression functions (i.e. functions mapping from the input space to  $\mathbb{R}$ ):

- 0: Initialize  $f_0(x) = 0$ .

1: For  $m = 1$  to  $M$ :

(a) Compute:

$$\mathbf{g}_m = \left( \frac{\partial}{\partial f_{m-1}(x_j)} \sum_{i=1}^n \ell(y_i, f_{m-1}(x_i)) \right)_{j=1}^n$$

(b) Fit regression model to  $-\mathbf{g}_m$ :

$$h_m = \arg \min_{h \in \mathcal{F}} \sum_{i=1}^n ((-\mathbf{g}_m)_i - h(x_i))^2.$$

(c) Choose fixed step size  $\nu_m = \nu \in (0, 1]$ , or take

$$\nu_m = \arg \min_{\nu > 0} \sum_{i=1}^n \ell(y_i, f_{m-1}(x_i) + \nu h_m(x_i)).$$

(d) Take the step:

$$f_m(x) = f_{m-1}(x) + \nu_m h_m(x)$$

3: Return  $f_M$ .

This method goes by many names, including gradient boosting machines (GBM), generalized boosting models (GBM), AnyBoost, and gradient boosted regression trees (GBRT), among others. One of the nice aspects of gradient boosting is that it can be applied to any problem with a subdifferentiable loss function.

## Gradient Boosting Regression Implementation

First we'll keep things simple and consider the standard regression setting with square loss. In this case, we have  $\mathcal{Y} = \mathbb{R}$ , our loss function is given by  $\ell(\hat{y}, y) = 1/2 (\hat{y} - y)^2$ , and at the  $m$ 'th round of gradient boosting, we have

$$h_m = \arg \min_{h \in \mathcal{F}} \sum_{i=1}^n [(y_i - f_{m-1}(x_i)) - h(x_i)]^2.$$

You can derive the above equation using (a) and (b).

- Complete the `gradient_boosting` class. As the base regression algorithm to compute the argmin, you should use sklearn's regression tree. You should use the square loss for the tree splitting rule (`criterion` keyword argument) and use the default sklearn leaf prediction rule from the `predict` method<sup>1</sup>. We will also use a constant step size  $\nu$ .
- Use the `Decision_Tree` code provided to build gradient boosting models on the regression data sets `krr-train.txt`, and include the plots generated. For debugging you can use the sklearn implementation of `GradientBoostingRegressor`<sup>2</sup>.

<sup>1</sup>Examples of usage are given in the skeleton code to debug previous problems, and you can check the docs <https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html>

<sup>2</sup><https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html>

### 3 Neural Network Introduction

There is no doubt that neural networks are a very important class of machine learning models. Given the sheer number of people who are achieving impressive results with neural networks, one might think that it's relatively easy to get them working. This is a partly an illusion. One reason so many people have success is that, thanks to GitHub, they can copy the exact settings that others have used to achieve success. In fact, in most cases they can start with “pre-trained” models that already work for a similar problem, and “fine-tune” them for their own purposes. It's far easier to tweak and improve a working system than to get one working from scratch. If you create a new model, you're kind of on your own to figure out how to get it working: there's not much theory to guide you and the rules of thumb do not always work. Understanding even the most basic questions, such as the preferred variant of SGD to use for optimization, is still a very active area of research.

One thing is clear, however: If you do need to start from scratch, or debug a neural network model that doesn't seem to be learning, it can be immensely helpful to understand the low-level details of how your neural network works – specifically, back-propagation. With this assignment, you'll have the opportunity to linger on these low-level implementation details. Every major neural network type (RNNs, CNNs, Resnets, etc.) can be implemented using the basic framework we'll develop in this assignment.

To help things along, Philipp Meerkamp, Pierre Garapon, and David Rosenberg have designed a minimalist framework for computation graphs and put together some support code. The intent is for you to read, or at least skim, every line of code provided, so that you'll know you understand all the crucial components and could, in theory, create your own from scratch. In fact, creating your own computation graph framework from scratch is highly encouraged – you'll learn a lot.

### 4 Computation Graph Framework

To get started, please read the tutorial on the computation graph framework we'll be working with. (Note that it renders better if you view it locally.) The use of computation graphs is not specific to machine learning or neural networks. Computation graphs are just a way to represent a function that facilitates efficient computation of the function's values and its gradients with respect to inputs. The tutorial takes this perspective, and there is very little in it about machine learning, per se.

To see how the framework can be used for machine learning tasks, we've provided a full implementation of linear regression. You should start by working your way through the `__init__` of the `LinearRegression` class in `linear_regression.py`. From there, you'll want to review the node class definitions in `nodes.py`, and finally the class `ComputationGraphFunction` in `graph.py`. `ComputationGraphFunction` is where we repackage a raw computation graph into something that's more friendly to work with for machine learning. The rest of `linear_regression.py` is fairly routine, but it illustrates how to interact with the `ComputationGraphFunction`.

As we've noted earlier in the course, getting gradient calculations correct can be difficult. To help things along, we've provided two functions that can be used to test the backward method of a node and the overall gradient calculation of a `ComputationGraphFunction`. The functions are in `test_utils.py`, and it's recommended that you review the tests provided for the linear regression implementation in `linear_regression.t.py`. (You can run these tests from the command line with `python3 linear_regression.t.py`.) The functions actually doing the testing, `test_node_backward` and `test_ComputationGraphFunction`, may seem a bit intricate, but

they're implementing the exact same `gradient_checker` logic we saw in the second homework assignment.

Once you've understood how linear regression works in our framework, you're ready to start implementing your own algorithms. To help you get started, please make sure you are able to execute the following commands:

- `cd /path/to/hw4`
- `python3 linear_regression.py`
- `python3 linear_regression.t.py`

## 5 Ridge Regression

When moving to a new system, it's always good to start with something familiar. But that's not the only reason we're doing ridge regression in this homework. In ridge regression the parameter vector is "shared", in the sense that it's used twice in the objective function. In the computation graph, this can be seen in the fact that the node for the parameter vector has two outgoing edges. This sharing is common many popular neural networks (RNNs and CNNs), where it is often referred to as *parameter tying*.

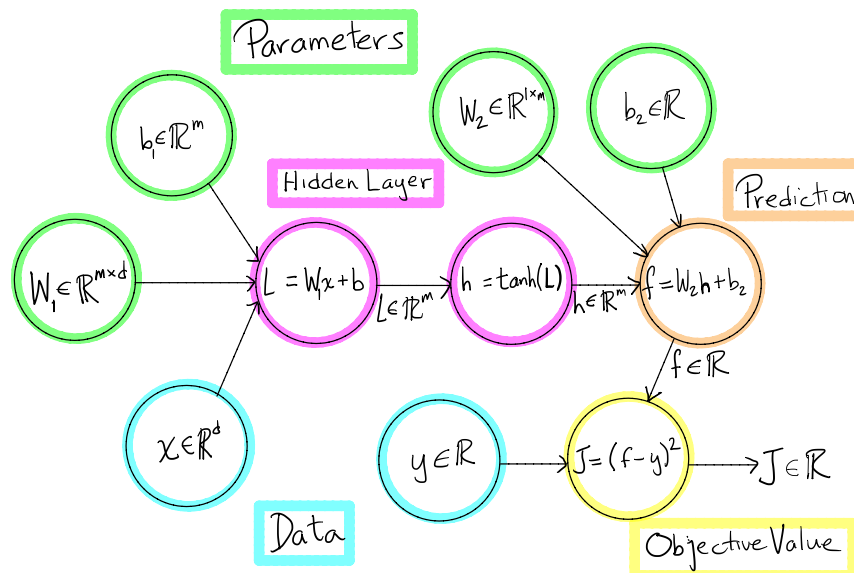
`ridge_regression.py` provides a skeleton code and `ridge_regression.t.py` is a test code, which you should eventually be able to pass.

7. Complete the class `L2NormPenaltyNode` in `nodes.py`. If your code is correct, you should be able to pass `test_L2NormPenaltyNode` in `ridge_regression.t.py`. Please attach a screenshot that shows the test results for this question.
8. Complete the class `SumNode` in `nodes.py`. If your code is correct, you should be able to pass `test_SumNode` in `ridge_regression.t.py`. Please attach a screenshot that shows the test results for this question.
9. Implement ridge regression with  $w$  regularized and  $b$  unregularized. Do this by completing the `__init__` method in `ridge_regression.py`, using the classes created above. When complete, you should be able to pass the tests in `ridge_regression.t.py`. Report the average square error on the **training** set for the parameter settings given in the `main()` function.

## 6 Multilayer Perceptron

Let's now turn to a multilayer perceptron (MLP) with a single hidden layer and a square loss. We'll implement the computation graph illustrated below:

## Multilayer Perceptron, 1 hidden layer, square loss



The crucial new piece here is the nonlinear **hidden layer**, which is what makes the multilayer perceptron a significantly larger hypothesis space than linear prediction functions.

### 6.1 The standard non-linear layer

The multilayer perceptron consists of a sequence of “layers” implementing the following non-linear function

$$h(x) = \sigma(Wx + b),$$

where  $x \in \mathbb{R}^d$ ,  $W \in \mathbb{R}^{m \times d}$ , and  $b \in \mathbb{R}^m$ , and where  $m$  is often referred to as the number of **hidden units** or **hidden nodes**.  $\sigma$  is some non-linear function, typically  $\tanh$  or  $\text{ReLU}$ , applied element-wise to the argument of  $\sigma$ . Referring to the computation graph illustration above, we will implement this nonlinear layer with two nodes, one implementing the affine transform  $L = W_1x + b_1$ , and the other implementing the nonlinear function  $h = \tanh(L)$ . In this problem, we’ll work out how to implement the backward method for each of these nodes.

#### The Affine Transformation

In a general neural network, there may be quite a lot of computation between any given affine transformation  $Wx + b$  and the final objective function value  $J$ . We will capture all of that in a function  $f: \mathbb{R}^m \rightarrow \mathbb{R}$ , for which  $J = f(Wx + b)$ . Our goal is to find the partial derivative of  $J$  with respect to each element of  $W$ , namely  $\partial J / \partial W_{ij}$ , as well as the partials  $\partial J / \partial b_i$ , for each element of  $b$ . For convenience, let  $y = Wx + b$ , so we can write  $J = f(y)$ . Suppose we have already computed the partial derivatives of  $J$  with respect to the entries of  $y = (y_1, \dots, y_m)^T$ , namely  $\frac{\partial J}{\partial y_i}$  for  $i = 1, \dots, m$ . Then by the chain rule, we have

$$\frac{\partial J}{\partial W_{ij}} = \sum_{r=1}^m \frac{\partial J}{\partial y_r} \frac{\partial y_r}{\partial W_{ij}}.$$

10. Show that  $\frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial y_i} x_j$ , where  $x = (x_1, \dots, x_d)^T$ . [Hint: Although not necessary, you might find it helpful to use the notation  $\delta_{ij} = \begin{cases} 1 & i = j \\ 0 & \text{else} \end{cases}$ . So, for examples,  $\partial_{x_j} (\sum_{i=1}^n x_i^2) = 2x_j \delta_{ij} = 2x_j$ .]
11. Now let's vectorize this. Let's write  $\frac{\partial J}{\partial y} \in \mathbb{R}^{m \times 1}$  for the column vector whose  $i$ th entry is  $\frac{\partial J}{\partial y_i}$ . Let's also define the matrix  $\frac{\partial J}{\partial W} \in \mathbb{R}^{m \times d}$ , whose  $ij$ 'th entry is  $\frac{\partial J}{\partial W_{ij}}$ . Generally speaking, we'll always take  $\frac{\partial J}{\partial A}$  to be an array of the same size ("shape" in numpy) as  $A$ . Give a vectorized expression for  $\frac{\partial J}{\partial W}$  in terms of the column vectors  $\frac{\partial J}{\partial y}$  and  $x$ . [Hint: Outer product.]
12. In the usual way, define  $\frac{\partial J}{\partial x} \in \mathbb{R}^d$ , whose  $i$ 'th entry is  $\frac{\partial J}{\partial x_i}$ . Show that

$$\frac{\partial J}{\partial x} = W^T \left( \frac{\partial J}{\partial y} \right)$$

[Note, if  $x$  is just data, technically we won't need this derivative. However, in a multilayer perceptron,  $x$  may actually be the output of a previous hidden layer, in which case we will need to propagate the derivative through  $x$  as well.]

13. Show that  $\frac{\partial J}{\partial b} = \frac{\partial J}{\partial y}$ , where  $\frac{\partial J}{\partial b}$  is defined in the usual way.

## Element-wise Transformers

Our nonlinear activation function nodes take an array (e.g. a vector, matrix, higher-order tensor, etc), and apply the same nonlinear transformation  $\sigma : \mathbb{R} \rightarrow \mathbb{R}$  to every element of the array. Let's abuse notation a bit, as is usually done in this context, and write  $\sigma(A)$  for the array that results from applying  $\sigma(\cdot)$  to each element of  $A$ . If  $\sigma$  is differentiable at  $x \in \mathbb{R}$ , then we'll write  $\sigma'(x)$  for the derivative of  $\sigma$  at  $x$ , with  $\sigma'(A)$  defined analogously to  $\sigma(A)$ .

Suppose the objective function value  $J$  is written as  $J = f(\sigma(A))$ , for some function  $f : S \mapsto \mathbb{R}$ , where  $S$  is an array of the same dimensions as  $\sigma(A)$  and  $A$ . As before, we want to find the array  $\frac{\partial J}{\partial A}$  for any  $A$ . Suppose for some  $A$  we have already computed the array  $\frac{\partial J}{\partial S} = \frac{\partial f(S)}{\partial S}$  for  $S = \sigma(A)$ . At this point, we'll want to use the chain rule to figure out  $\frac{\partial J}{\partial A}$ . However, because we're dealing with arrays of arbitrary shapes, it can be tricky to write down the chain rule. Appropriately, we'll use a tricky convention: We'll assume all entries of an array  $A$  are indexed by a single variable. So, for example, to sum over all entries of an array  $A$ , we'll just write  $\sum_i A_i$ .

14. Show that  $\frac{\partial J}{\partial A} = \frac{\partial J}{\partial S} \odot \sigma'(A)$ , where we're using  $\odot$  to represent the **Hadamard product**. If  $A$  and  $B$  are arrays of the same shape, then their Hadamard product  $A \odot B$  is an array with the same shape as  $A$  and  $B$ , and for which  $(A \odot B)_i = A_i B_i$ . That is, it's just the array formed by multiplying corresponding elements of  $A$  and  $B$ . Conveniently, in `numpy` if  $A$  and  $B$  are arrays of the same shape, then `A*B` is their Hadamard product.

## 6.2 MLP Implementation

15. Complete the class `AffineNode` in `nodes.py`. Be sure to propagate the gradient with respect to  $x$  as well, since when we stack these layers,  $x$  will itself be the output of another

node that depends on our optimization parameters. If your code is correct, you should be able to pass `test_AffineNode` in `mlp_regression.t.py`. Please attach a screenshot that shows the test results for this question.

16. Complete the class `TanhNode` in `nodes.py`. As you'll recall,  $\frac{d}{dx} \tanh(x) = 1 - \tanh^2 x$ . Note that in the forward pass, we'll already have computed `tanh` of the input and stored it in `self.out`. So make sure to use `self.out` and not recalculate it in the backward pass. If your code is correct, you should be able to pass `test_TanhNode` in `mlp_regression.t.py`. Please attach a screenshot that shows the test results for this question.
17. Implement an MLP by completing the skeleton code in `mlp_regression.py` and making use of the nodes above. Your code should pass the tests provided in `mlp_regression.t.py`. Note that to break the symmetry of the problem, we initialize our weights to small random values, rather than all zeros, as we often do for convex optimization problems. Run the MLP for the two settings given in the `main()` function and report the average **training** error. Note that with an MLP, we can take the original scalar as input, in the hopes that it will learn nonlinear features on its own, using the hidden layers. In practice, it is quite challenging to get such a neural network to fit as well as one where we provide features.

### 6.3 Multiclass classification with an MLP [Optional]

We consider a generic classification problem with  $K$  classes over inputs  $x$  of dimension  $d$ . Using a MLP we will compute a  $K$ -dimensional vector  $z$  representing scores,

$$z = W_2 \tanh(W_1 x + b_1) + b_2,$$

with  $W_1 \in \mathbb{R}^{m \times d}$ ,  $b_1 \in \mathbb{R}^m$ ,  $W_2 \in \mathbb{R}^{K \times m}$  and  $b_2 \in \mathbb{R}^K$ . Our model assumes that  $x$  belongs to class  $k$  with probability

$$e^{z_k} / \sum_{k=1}^K e^{z_k},$$

which corresponds to applying a Softmax to the scores. Given this probabilistic model we can train the model by minimizing the negative log-likelihood.

18. Implement a Softmax node. We provided skeleton code for class `SoftmaxNode` in `nodes.py`. If your code is correct, you should be able to pass `test_SoftmaxNode` in `multiclass.t.py`. Please attach a screenshot that shows the test results for this question.
19. Implement a negative log-likelihood loss node for multiclass classification. We provided skeleton code for class `NLLNode` in `nodes.py`. The test code for this question is combined with the test code for the next question.
20. Implement a MLP for multiclass classification by completing the skeleton code in `multiclass.py`. Your code should pass the tests in `test_multiclass` provided in `multiclass.t.py`. Please attach a screenshot that shows the test results for this question.