Programming Assignment 0: Single-Variable Scalar Autograd

Due: Thursday 09/11/2025 @11:59pm EST

Disclaimer

Written assignments must be typeset in LaTeX. I have released the `.tex` files that were used to generate this pdf which you are welcome to use as the foundation of your solutions. These questions are **implementation** question in the Python programming language. In this assignment you will be implementing a symbolic differentiation system that supports single-variable scalar expressions. You will have to flex your skills with polymorphism and tree data structures and will get you in the right mindset for how neural network packages are implemented.

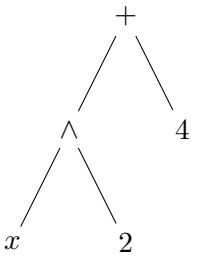
Note that you are **NOT** allowed to use any help from LLMs, online solutions, old solutions, etc. when solving these problems. Your solutions are your own. You **are** allowed to chat with your classmates, but not with detail granular enough to copy each others work.

Task 1: Single-Variable Scalar Computation Trees

Suppose we want a compute to be able to reason about mathematics in a similar way to Mathematica or Wolfram Alpha. The very first step in designing such a system would be to give the machine a representation of symbolic expressions. This representation would need to be manipulatable programmatically as well as be expressive enough to capture the set of expressions we wish the system to reason with. A clever solution to this problem has already been developed by programming languages (and also lambda calculus): encode expressions as tree data structures. Leaf vertices in this tree are constants and variables, while internal vertices are operations that relate their children together using mathematical semantics. For example, the single-variable scalar expression

$$x^2 + 4$$

is equivalently expressed using the following tree:



This data structure is called a *computation* tree. Tree traversal algorithms give us mechanisms for manipulating computation trees. For instance, a DFS traversal could be used to evaluate the above expression for a specific value of x. One of the main ways of manipulating symbolic expressions relevant to our class is differentiation. If we wanted to differentiate this expression, we would have to express differentiation as a tree traversal algorithm. So, how can we convert differentiation into tree traversals?

The first step is to remember our differentiation rules. In our expression, there are two kinds of operations occuring: addition and exponentiation (by a constant). Here are their individual derivative rules:

1. $\frac{d}{dx}(f(x)+g(x)) = \frac{d}{dx}f(x) + \frac{d}{dx}g(x)$. Abstracted further, we get $\frac{d}{dx}\sum_{i=1}^{n} f_i(x) = \sum_{i=1}^{n}\frac{d}{dx}f_i(x)$ or as I like to say "the derivative of a sum is the sum of its derivatives." Remember, differentiation is a *linear* operator, so it slides through sums (and other linear operations).
2. $\frac{d}{dx}(f(x))^c = c(f(x))^{c-1}\frac{d}{dx}f(x)$. Don't forget chain rule!

Since the derivative rule for addition of multiple components is the sum of the component derivatives, if we were to do a tree traversal and encounter an addition vertex, we know what to do! Our "derivative tree" should put an addition vertex in the same place, and for children have vertices for the derivatives of the original children! In fact, since all operators have their own individual derivative rules, we can comfortably convert differentiation into a DFS traversal! The derivative operation in code would take a computation tree as input and produce the "derivative" computation tree as output! Note that if we wanted to do multi-variable scalar computation trees and their derivatives, we wouldn't need to change much (our derivatives would change to partial derivatives thats all).

Task 2: Neural Networks and Computation Graphs

Part of what you learn in this class (or what you may know already) are than Neural Networks are just large symbolic expressions. However, instead of being a tree structure, Neural Networks form a computation *graph*. The reason for this is that a Neural Network may have multiple distinct outputs as well as multiple distinct inputs (for instance a Neural Network model could take as input text from a news article as well as an image and produce three predictions: the probability the image belongs to the article, a distribution over the topic the article contains, and a summary of the article in natural language).

Neural Networks are trained primarily with the gradient descent algorithm. If you don't know what that is yet, thats ok. Just know that one part of the gradient descent algorithm in this context is to calculate the derivative of our Neural Network (which remember is just a giant symbolic expression). Therefore, we will need to differentiate the computation graph of the Neural Network to get its "derivative" computation graph, which works much like differentiating our computation trees above. One major difference between Neural Networks and our computation trees above is its complexity: Neural Networks are not expressed as scalar symbolic expressions, they are expressed as *linear algebra* symbolic expressions. This results in the differentiation rules being more complicated than the scalar derivative rules you all know and love. But we'll get there. The first step is understanding how scalar computation trees work first.

Task 3: Implementation

To develop a single-variable scalar expression package, we will heavily rely on polymorphism. This matters less because we're programming in Python, but would matter much more in a strongly-typed language so we'll use it here anyways. The first step in our polymorphism is to create an abstract class `Expression`. This class will serve as a placeholder for a generic expression and also represents a generic vertex in our computation tree. This class is abstract so that we can define all the methods that must be implemented by concrete descendents when we start implementing specific vertex types that can appear in our computation trees. I have already provided the `Expression` class in `autograd/expr.py`, but here it is again:

```
# SYSTEM IMPORTS
from typing import Type             # so we can do type annotation
from abc import ABC, abstractmethod # abstract class functionality in python


# PYTHON PROJECT IMPORTS


# TYPES DECLARED IN THIS MODULE
ExpressionType = Type["Expression"] # kinda like typedef in c/c++


# CONSTANTS


class Expression(ABC):
    @abstractmethod
    def differentiate(self: ExpressionType) -> ExpressionType:
        """
            A method for differentiating this expression. Produces as Expression.
        """
        ...

    @abstractmethod
    def eval(self: ExpressionType,
             x: float) -> float:
        """
            A method for evaluating this expression with argument "x". Produces a float.
        """
        ...

    @abstractmethod
    def deepcopy(self: ExpressionType) -> ExpressionType:
        """
            A method for getting a deepcopy of this expression. Produces an Expresssion.
        """
        ...
```

Part A: Constant Expressions (5 points)

Create a class in file `autograd/const.py` called `Constant` that extends class `Expression`. The `Constant` class represents any constant value (e.g. a `float`) that will be used as part of a larger expression. The value of this constant should be provided as an argument called `val: float` in the constructor and saved in a field with the same name and datatype as the argument. Remember that you now have to override all the methods decorated with `abstractmethod` from `Expression`. Also remember the derivative rule for a constant:

$$\frac{d}{dx}c = 0$$

You should also probably implement two special methods in Python called `__str__` and `__repr__`. These methods are useful if you want to print your `Constant` and are rough equivalents to the `toString()` method from Java.

Also remember to implement the `eval` and `deepcopy` methods!. The `eval` method should return the value of this expression when supplied with the value for variable "x". Since this is a constant expression, the value of this expression is just the constant. The `deepcopy` method should just return a new instance of the `Constant` class with the same fields.

Part B: Variable Expressions (5 points)

Create a class in file `autograd/var.py` called `Variable` that extends class `Expression`. The `Variable` class represents variable "x" and is our only variable in this assignment. Remember the derivative rule for a variable:

$$\frac{d}{dx}x = 1$$

Also don't forget to implement `__str__` and `__repr__`, `eval` and `deepcopy`!

Part C: BinaryOp and Power Expressions (20 points)

These two operators depend on each other so they need to be implemented together. Lets focus on the first: create class `BinaryOp` in `autograd/binop.py`. The `BinaryOp` class represents a binary vertex in our tree, and can be configured to represent addition, subtraction, multiplication, and division. You will need to create an enum to represent this configuration (please put it above class `BinaryOp`):

```
from enum import Enum
from typing import Type

OpType = Type["Op"]

class Op(Enum):
    ADD = 1
    SUB = 2
    MUL = 3
    DIV = 4

    def __str__(self: OpType) -> str:
        op_str: str = None
        if self == Op.ADD:
            op_str = "+"
        elif self == Op.SUB:
            op_str = "-"
        elif self == Op.MUL:
            op_str = "*"
        elif self == Op.DIV:
            op_str = "/"
        else:
            raise ValueError("ERROR: unknown op [{0}]".format(self))
        return op_str

    def __repr__(self: OpType) -> str:
        return self.__str__()
```

A `BinaryOp` needs to take three arguments in its constructor (listed in order): `lhs: Expression`, `op: Op`, `rhs: Expression`. Please save these in fields with the same names and datatypes as the arguments. Also remember the derivative rules of a `BinaryOp` (conditioned on the type of configuration):

1. $\frac{d}{dx}\Big(f(x)+g(x)\Big) = \frac{d}{dx}f(x) + \frac{d}{dx}g(x)$
2. $\frac{d}{dx}\Big(f(x)-g(x)\Big) = \frac{d}{dx}f(x) - \frac{d}{dx}g(x)$
3. $\frac{d}{dx}\Big(f(x)g(x)\Big) = \Bigg(\Big(\frac{d}{dx}f(x)\Big)g(x)\Bigg) + \Bigg(f(x)\Big(\frac{d}{dx}g(x)\Big)\Bigg)$
4. $\frac{d}{dx}\Big(\frac{f(x)}{g(x)}\Big) = \frac{\Bigg[\Bigg(g(x)\Big(\frac{d}{dx}f(x)\Big)\Bigg) - \Bigg(f(x)\Big(\frac{d}{dx}g(x)\Big)\Bigg)\Bigg]}{(g(x))^2}$

Before implementing quotient rule, please create class `Power` in `autograd/pow.py`. The `Power` class represents a vertex in our tree when expression $f(x)$ is raised to a constant power c. The constructor of `Power` should take two arguments listed in order: `base: Expression` and `exp: float`. Please store these in fields with the same name and datatypes as the arguments. Now that this class exists, you can return to your `BinaryOp` to use it for quotient rule. To avoid a circular import (your `Power` class is going to have to import `BinaryOp`), only import `Power` within the `differentiate()` method of `BinaryOp`.

Now return to class `Power`. Remember the derivative rule for expressions raised to a constant power:

$$\frac{d}{dx}(f(x))^c = \Big(c(f(x))^{c-1}\Big)\frac{d}{dx}f(x)$$

Also remember to implement `__str__` and `__repr__`, `eval`, and `deepcopy`! I would really recommend adding lots of parenthases to show the nesting! I have added the nesting I want you to implement with parenthases in the derivative rules! Also note that when doing `deepcopy`, every time you want to refer to the any expression field of the deepcopy, those should be deepcopies of themselves!

Part D: Sin and Cos Expressions (10 points)

These two operators also depend on each other so they need to be implemented together. Lets focus on the first: create class `Sin` in `autograd/sin.py`. The `Sin` class represents the sinusoid trigonometric function and will be a unary vertex in our tree. A `Sin` needs to take one argument in its constructor: `arg: Expression`. Please save this in a field with the same name and datatype as the argument. Remember the derivative rule for $sin(f(x))$:

$$\frac{d}{dx}sin(f(x)) = \Big(\frac{d}{dx}f(x)\Big)cos(f(x))$$

Before you can implement this derivative rule, you will need to create class `Cos` in `autograd/cos.py`. The `Cos` class represents the cosine trigonometric function and is also a unary vertex in our tree. A `Cos` needs to take one argument in its constructor: `arg: Expression`. Please save this in a field with the same name and datatype as the argument. When implementing the derivative for `Cos`, you will want to import the `Sin` class within the `differentiate` method, and vice versa for the `Sin` class (to avoid a circular import). Remember the derivative rule for $cos(f(x))$:

$$\frac{d}{dx}cos(f(x)) = \Bigg(-1\Big(\frac{d}{dx}f(x)\Big)\Bigg)sin(f(x))$$

Also remember to implement `__str__` and `__repr__`, `eval` and `deepcopy`! I would really recommend adding lots of parenthases to show the nesting! I have added the nesting I want you to implement with parenthases in the derivative rules!

Part E: Log and Exponential Expressions (10 points)

These last two operators don't depend on each other but are topically related. In this assignment we will refer to all logarithms as natural logarithms, and all exponential operators as base e. Create class `Log` in file `autograd/log.py`. The `Log` class will refer to the natural logarithm operator, and will be a unary vertex in our tree. A `Log` needs to take one argument in its constructor: `arg: Expression`. Please save this in a field with the same name and datatype as the argument. Remember the derivative rule for $\log(f(x))$:

$$\frac{d}{dx}log(f(x)) = \frac{\frac{d}{dx}f(x)}{f(x)}$$

Now create class `Exp` in file `autograd/exp.py`. The `Exp` class will refer to $e^{f(x)}$ and will also be a unary vertex in our tree. A `Exp` needs to take one argument in its constructor: `arg: Expression`. Please save this in a field with the same name and datatype as the argument. Remember the derivative rule for $e^{f(x)}$:

$$\frac{d}{dx}e^{f(x)} = \Big(\frac{d}{dx}f(x)\Big)e^{f(x)}$$

Also remember to implement `__str__` and `__repr__`, `eval` and `deepcopy`!

Future Assignments

Building a fully-fledged scalar derivative package isn't much harder than this. All it takes to make our package more expressive is to add more operators, and maybe redefine some of the existing ones to be more general (for instance we could merge `Power` and `Exp` into a single operator that represents $f(x)^{g(x)}$, abstracting the base for `Log`, etc.). Making this package multi-variate would require changing the `Variable` operator to represent different variables (for instance adding the name of the variable in the constructor), and adding the variable to differentiate with respect to in the `differentiate` method.

Extending this to matrix expressions on the other hand is much more difficult. Most noteably, scalar derivatives are full of chain rule (e.g. lots of products) and the ordering of the terms in these products doesn't matter. However, the order of matrix multiplication matters, so we would have to be much more precise with the term ordering if we wished to extend this to linear algebra.

Assuming you did so, you would arrive at a package that is really close to how tensorflow-v1 was implemented. This autograd design is very expressive, however it isn't as flexible as other solutions like Pytorch or Jax. The real improvement those packages made was to avoid a "static" computation graph. In our package, we cannot really express piecewise functionality: to let data flow through one branch of the tree sometimes, and flow along other branches other times. This design is much more successful, and Google even admitted it by redesigning tensorflow to mostly copy the Pytorch design. Pytorch composes the computation graph as data flows through the model, not beforehand as our package does. Once the "forward" pass is complete, the computation graph is traversed in reverse topological order, and each piece of the computation graph is auto-differentiated "in place" (again not beforehand like our package). For future assignments, we will mimic this style of autograd.

Task 4: Lasso and Ridge Regression

Lets get some practice with matrix derivatives. As you likely know, one of the first regression models (and one of the most widely used) is called *Linear Regression*, which is vectorized to make predictions for an entire dataset using the following equations:

$$\hat{\vec{y}} = f_{\vec{w}}(\mathbf{X}) = \mathbf{X}\vec{w}$$

where $\mathbf{X} \in \mathbb{R}^{n \times d}$ is your dataset consisting of n points and d features, $\vec{w}$ is the parameters of the model, and $\hat{\vec{y}} \in \mathbb{R}^n$ is a matrix consisting of n linear predictions. Linear Regression is often trained using the "least-squares" loss function:

$$e = \mathcal{L}(\hat{\vec{y}}, \vec{y}_{gt}) = \sum_{i=1}^{n} (\hat{y}^{(i)} - y_{gt}^{(i)})^2$$

where $\vec{y}_{gt} \in \mathbb{R}^{n \times 1}$ is ground truth data, $\hat{y}^{(i)}$ is the prediction for sample i, and $y_{gt}^{(i)}$ is the ground truth for sample i.This model (called "least-squares" regression) is offically formalized as the following optimization problem:

$$\vec{w}^* = \arg\min_{\vec{w} \in \mathbb{R}^n} \mathcal{L}(f_{\vec{w}}(\mathbf{X}), \vec{y}_{gt})$$

This model is often augmented with prior assumptions about what solutions $\vec{w}$ are preferred (called a *prior* distribution). In the prescence of a (non-trivial) prior distribution, our optimization objective changes from performing maximum-likelihood estimation (MLE) to maximum a-posteriori estimation (MAP). You will show this in wa0, but suffice to say that our new objective becomes:

$$\vec{w}^* = \arg\min_{\vec{w} \in \mathbb{R}^n} \mathcal{L}(f_{\vec{w}}(\mathbf{X}), \mathbf{Y}_{gt}) + \log Pr[\vec{w}|\mathbf{X}, \vec{y}_{gt}]$$

where $Pr[\vec{w}|\mathbf{X}, \vec{y}_{gt}]$ is the prior distribution. Depending on our choice of (non-trivial) prior distribution, we can get different behavior out of our optimization (i.e. the solution will exhibit different properties). In this task, you will verify that under two common choices of prior distributions: $\vec{w} \sim \mathcal{N}(\vec{0}, \sigma\mathbf{I})$ and $\vec{w} \sim Laplace(0, b)$, solutions will become smaller in magnitude, and sparser respectively. Note that the two choices of priors create the following additions (called regularizers) to the loss function:

1. Ridge regression: $\vec{w} \sim \mathcal{N}(\vec{0}, \sigma\mathbf{I}) \rightarrow \vec{w}^* = \arg\min_{\vec{w} \in \mathbb{R}^n} \mathcal{L}(f_{\vec{w}}(\mathbf{X}), \mathbf{Y}_{gt}) + \lambda||\vec{w}||_2^2$
2. Lasso regression: $\vec{w} \sim Laplace(0, b) \rightarrow \vec{w}^* = \arg\min_{\vec{w} \in \mathbb{R}^n} \mathcal{L}(f_{\vec{w}}(\mathbf{X}), \mathbf{Y}_{gt}) + \lambda||\vec{w}||_1$

Part A: Ridge Regression (25 points)

In the file `regressors/models/ridge.py` you will find a partially completed class called `RidgeRegressor`. You will have to complete three methods (ordered by difficulty):

1. `predict`: This method is where you will implement $f_{\vec{w}}$ so that when provided with a data matrix, your model can use its parameter vector $\vec{w}$ to make predictions.
2. `loss`: This method is where you will implement $\mathcal{L}(\hat{\vec{y}}, \vec{y}_{gt})$ so that when provided with predictions and ground truth, your model can estimate how far off its predictions are.
3. `grad`: This method is where you will implement $\frac{\partial}{\partial \vec{w}} \mathcal{L}(\hat{\vec{y}}, \vec{y}_{gt})$ so that your model will know how to adjust $\vec{w}$ in order to minimize $\mathcal{L}$. Since this is your first assignment and you might be rusty, I will give you this derivative equation. However, in the future you will be responsible for differentiating these matrix equations yourself:

$$\frac{\partial}{\partial \vec{w}} \mathcal{L}(\hat{\vec{y}}, \vec{y}_{gt}) = 2\mathbf{X}^T(\hat{\vec{y}} - \vec{y}_{gt}) + 2\lambda\vec{w}$$

To test your code, I would recommend make a testing script inspired by the file `regressors/train.py`). You can be more confident about your gradients if you can see the average loss decreasing smoothly. Additionally, you are welcome to implement numerical gradient checking using the following code:

```
def grad_check(X: np.ndarray, Y_gt: np.ndarray, m: object, # m is the regressor object
               epsilon: float = 1e-7, delta: float = 1e-6) -> None:
    num_grad: np.ndarray = np.zeros_like(m.w)   # we will fill this in with numerical approx
    sym_grad: np.ndarray = m.grad(X, Y_gt)      # your symbolic gradient

    # gotta be the same shape
    assert(sym_grad.shape == num_grad.shape)

    # go over each parameter one at a time and measure (f(x+eps) - f(x-eps)) / (2*eps)
    # where f is our loss function, x is a specific parameter
    # and eps (epsilon) is the amount we're nudging
    for index, v in np.ndenumerate(m.w):
        m.w[index] += epsilon
        num_grad[index] += m.loss(m.predict(X), Y_gt)

        m.w[index] -= 2*epsilon
        num_grad[index] -= m.loss(m.predict(X), Y_gt)

        # set param back to normal
        m.w[index] = v
        num_grad[index] /= (2*epsilon)

    # measure ratio of norms to see how close the symbolic grad are to the numeric grad
    ratio: float = np.linalg.norm(sym_grad-num_grad) / np.linalg.norm(sym_grad+num_grad)
    if ratio > delta:
        raise RuntimeError(f"ERROR: failed grad check. delta: {delta}, ratio: {ratio}")
```

Part B: Lasso Regression (25 points)

In the file `regressors/models/lasso.py` you will find a partially completed class called `LassoRegressor`. You will have to complete three methods (ordered by difficulty):

1. `predict`: This method is where you will implement $f_{\vec{w}}$ so that when provided with a data matrix, your model can use its parameter vector $\vec{w}$ to make predictions.
2. `loss`: This method is where you will implement $\mathcal{L}(\hat{\vec{y}}, \vec{y}_{gt})$ so that when provided with predictions and ground truth, your model can estimate how far off its predictions are.
3. `grad`: This method is where you will implement $\frac{\partial}{\partial \vec{w}}\mathcal{L}(\hat{\vec{y}}, \vec{y}_{gt})$ so that your model will know how to adjust $\vec{w}$ in order to minimize $\mathcal{L}$. Since this is your first assignment and you might be rusty, I will give you this derivative equation. However, in the future you will be responsible for differentiating these matrix equations yourself:

$$\frac{\partial}{\partial \vec{w}}\mathcal{L}(\hat{\vec{y}}, \vec{y}_{gt}) = 2\mathbf{X}^T(\hat{\vec{y}} - \vec{y}_{gt}) + \lambda sign(\vec{w})$$

Note that $sign(\vec{w})$ is the *sign* function which is defined as: $sign(x) = \begin{cases} +1 & x > 0 \\ -1 & x < 0 \\ 0 & \text{otherwise} \end{cases}$ The sign function is element-wise independent, meaning that if the argument to $sign(\cdot)$ is a vector, it produces a vector where the scalar version of $sign(\cdot)$ has been applied to each element independently (i.e. "f of a vector is a vector full of fs").

Once you are satisfied that your code is working, run `regressors/train.py`. It may take a few minutes. The program should produce two plots, one plot contains the elements of $\vec{w}^*$ from ridge regression, the other plot contains the elements of $\vec{w}^*$ from lasso regression. What kind of differences do you see between the two solutions? Note that one solution is much sparser than the other.

Task 5: Submitting Your Assignment

Please turn in `[var|const|cos|binop|pow|sin|cos|log|exp|ridge|lasso].py` on gradescope. You should be able to just drag and drop them in!