CPSC 340 Assignment 7 (Optional, “due” 2021-06-30)

This assignment is marked on completeness, not correctness. It is still a worthwhile effort to learn the materials present in this assignment. We will reward full marks to each subsection of this assignment if it satisfies *ALL* of the following conditions:

1. The answer clearly includes the derivations or the images requested.
2. The answer is filled out in a sensible manner (i.e. not random garbles), addressing the question as best as possible.
3. Your work is clearly shown and demonstrates that you have put effort into getting the correct answer.

1 Jacobians

Recall that Jacobians are generalizations of multi-variate derivatives. A Jacobian can be between any two variables involved in a computational graph. For example, for a least squares error:

$$f(w) = \frac{1}{2}\sum_{i=1}^{n}\left(w^T x_i - y_i\right)^2,$$

we can have Jacobians such as $\frac{\partial f}{\partial w}$, $\frac{\partial f}{\partial \hat{y}}$ $\frac{\partial \hat{y}}{\partial w}$, etc.

The shape of a Jacobian is an important thing to note. A Jacobian can be a vector, a matrix, or a *tensor* of arbitrary ranks, depending on what the numerator and the denominator of the Jacobian expression is. For example, if f is a scalar and w is a $d \times 1$ vector, the Jacobian $\frac{\partial f}{\partial w}$ is a $d \times 1$ vector. If $\hat{y}$ is a $n \times 1$ vector and w is a $d \times 1$ vector, the Jacobian $\frac{\partial \hat{y}}{\partial w}$ is a $d \times n$ matrix.

As you can see, the shape of the Jacobian is generally determined as (shape of denominator)$\times$(shape of numerator). However, this gets funky when matrices are involved in the numerator and/or the denominator. For example, if we computed $\hat{X} = ZW$ as in matrix factorization, the Jacobian $\frac{\partial \hat{X}}{\partial Z}$ is a tensor of shape $(n \times k) \times (n \times d)$.

In this portion of the assignment, we will get ourselves familiar with Jacobians, which are useful for understanding backpropagation and implementing automatic differentiation by ourselves.

1.1 A Jacobian Workout

Suppose we have:

- X, an $n \times d$ matrix
- y, an $n \times 1$ vector
- Y, a $n \times k$ matrix
- w, a $d \times 1$ vector
- W, a $k \times d$ matrix
- Z, an $n \times k$ matrix

For the following items, compute (1) the shape of each Jacobian, and (2) an expression for each Jacobian:

1. $f(w) = c$ (constant)

2. $f(w) = ||w||^2$ (squared L2-norm)
3. $f(w) = \sum_{j=1}^{d} w_j$ (sum)
4. $f(w) = ||w||^2 + \sum_{j=1}^{d} w_j$ (sum of functions of vector)
5. $f(w) = w^T x_i$ (vector dot product)
6. $f(w) = Xw$ (matrix-vector product)
7. $f(w) = w$ (vector identity function)
8. $f(w, x_i) = w + x_i$ (vector-vector sum)
9. $f(w) = w^2$ (element-wise power)
10. $f(W) = W$ (matrix identity function)

1.2 Jacobian of Matrix-Matrix Product

A matrix-matrix product has a pretty funky Jacobian, which is hard to express mathematically. The usual chain rule for products holds, i.e.

$$\frac{\partial}{\partial Z} ZW = \frac{\partial}{\partial Z} Z \bigodot W + Z \bigodot \frac{\partial}{\partial Z} W$$

where $\bigodot$ is a "broadcast matrix multiplication" operator between a 4D tensor and a matrix. For example, if A is an $(a \times b) \times (n \times d)$ tensor and B is a $d \times k$ matrix, then $C = A \bigodot B$ is an $(a \times b) \times (n \times k)$ matrix, where $C[i, j]$, $i \in 1, 2, \cdots, a$ and $j \in 1, 2, \cdots, b$ is an $n \times k$ matrix resulting from the matrix product $A[i, j]B$.

Consider this function:

$$f(Z, W) = ZW$$

Write down the shapes of the Jacobians $\frac{\partial f}{\partial Z}$ and $\frac{\partial f}{\partial W}$. Then, derive the form of the Jacobians $\frac{\partial f}{\partial Z}$ and $\frac{\partial f}{\partial W}$. That is, express the Jacobians in the forms that don't have partial derivative operations e.g. $\frac{\partial f}{\partial Z}$.

Hint: it's pretty nasty to write it down mathematically, so a trick like the "Einstein summation" might be necessary. Drawing a picture of the resulting Jacobian is perfectly fine as long as it can be implemented in code.

2 Multi-Variate Chain Rule and the Computational Graph

Suppose we have:

- X, an $n \times d$ matrix
- y, an $n \times 1$ vector
- Y, a $n \times k$ matrix
- w, a $d \times 1$ vector
- W, a $k \times d$ matrix.

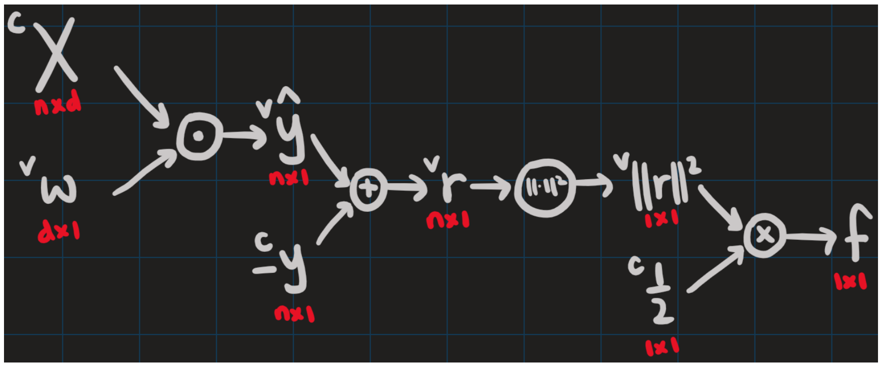
2.1 Drawing the Computational Graph

A computational graph is a directed graph where the nodes correspond to one of these three: (a) a variable, (b) a constant, or (c) an operation. Drawing these computation graphs will help us visualize the process of multi-variate chain rule, a crucial component of automatic differentiation.

Draw the computation graph for the following functions. Use $\hat{y}$ and $\hat{Y}$ to explicitly mark the output of the models. Make sure to clearly mark (1) constants and variables and (2) the shape of each symbol. See the below example for hints.

- $f(w) = \frac{1}{2}||Xw - y||^2$ (ordinary least squares)

  Answer:



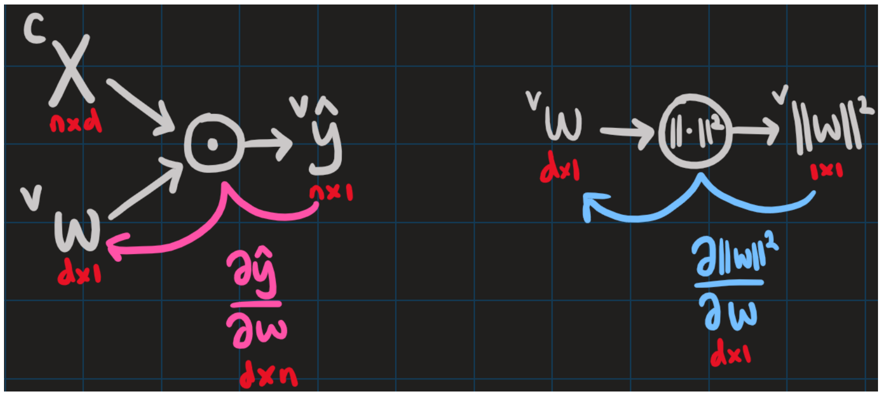
As you can see, I'm marking variables as "v" and constants as "c" on the upper-left corner of each symbol. Note that any operation involving a variable results in a variable.

1. $f(w) = \frac{1}{2}||Xw - y||^2 + \frac{\lambda}{2}||w||^2$ (L2-regularized least squares)
2. $f(w) = \sum_{i=1}^{n} \max\{0, 1 - y_i w^T x_i\} + \frac{\lambda}{2}||w||^2$ (L2-regularized hinge loss)
3. $f(Z, W) = \frac{1}{2}||ZW - X||_F^2$ (matrix factorization)

2.2 Visualizing Multi-Variate Chain Rule

Our gradient computation in this course have been implicitly using multi-variate chain rule. The reason we did not cover multi-variate chain rule in depth is because the formalism is difficult to convey, and it can be mostly bypassed by "expanding" the loss function first before taking gradients term-by-term.

For this exercise, we will compute the gradients of the well-known objective functions covered in this course, this time using multi-variate chain rule, involving Jacobians. Instead of overly formal math, we will use the "stepping stone" intuition, as portrayed in this image:



Remarks:

- The *operation* determines what the Jacobians look like.
- We only need backward edges going into variables, so we can ignore constants.
- Each *operation* has a Jacobian, which connects its output variable to its input variables.
- The shape of the Jacobian is `input_shape` $\times$ `output_shape`, which is easy to see from the picture.

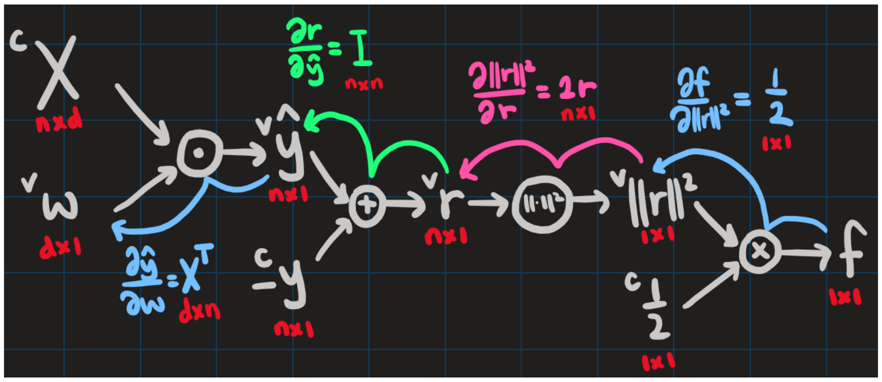
For each computation graph you created in the previous section, do the following:

- Create the "backward edges" that connect a parent variable to its child variable (or children variables) via a Jacobian stepping stone.
- Compute the gradient ($\frac{\partial f}{\partial w}$, $\frac{\partial f}{\partial W}$ or $\frac{\partial f}{\partial Z}$) using the backward edges created.
- Mark the shape of each Jacobian clearly.

See the below example for hints.

- $f(w) = \frac{1}{2}||Xw - y||^2$ (ordinary least squares)

  Answer:



$$\underset{d \times 1}{\frac{\partial f}{\partial w}} = \underset{d \times n}{\frac{\partial \hat{y}}{\partial w}} \underset{n \times n}{\frac{\partial r}{\partial \hat{y}}} \underset{n \times 1}{\frac{\partial \|r\|^2}{\partial r}} \underset{1 \times 1}{\frac{\partial f}{\partial \|r\|^2}} = X^T\left(I \cdot \left(2r \cdot \tfrac{1}{2}\right)\right)$$
$$= X^T r = X^T(Xw - y)$$

1. $f(w) = \frac{1}{2}||Xw - y||^2 + \frac{\lambda}{2}||w||^2$ (L2-regularized least squares)

2. $f(w) = \sum_{i=1}^{n} \max\{0, 1 - y_i w^T x_i\} + \frac{\lambda}{2}||w||^2$ (L2-regularized hinge loss)

2.3 Multi-Variable Chain Rule with Matrix Multiplication

We will put all of these ideas together to derive the Jacobian of the objective function associated with a matrix factorization model:

$$f(Z, W) = \frac{1}{2}||ZW - X||_F^2 = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{d}(\langle w^j, z_i\rangle - x_{ij})^2$$

Use the computational graph with the Jacobian of matrix multiplication to derive the forms for $\frac{\partial f}{\partial Z}$ and $\frac{\partial f}{\partial W}$.

Hint: it may look complicated at a glance, but you can apply the exact same "stepping stone" steps to compute this gradient recursively. You will run into a scenario where a 4D tensor is multiplied to a matrix. You can use these two facts about Jacobians:

1. A "generalized identity" tensor of shape $(n \times d) \times (n \times d)$ will preserve the shape and the content of another tensor being multiplied with it
2. Multiplying a $(n \times k) \times (n \times d)$ tensor A with an $n \times d$ matrix B is analogous to matrix-vector multiplication, i.e. element-wise multiply the matrix entries of the tensor A with B, and add the products together. Hence the output is an $n \times k$ matrix.

3 Automatic Differentiation

My go-to procrastination for this term was to build an in-house automatic differentiation library, so I can do some fun things with it. In `nampy.py`, you will find function object class definitions that are *composable* to enable foward and backward evaluations of a function.

If you run `python main.py -q 3`, the program does the following:

1. Initialize a random $n \times d$ matrix X, a random $d \times 1$ vector w, and a random $n \times 1$ vector y.
2. Evaluate the least squares objective function $f(w) = \frac{1}{2}||Xw - y||^2$
3. Evaluate the gradient of f with respect to w, i.e. $\nabla f(w) = \frac{\partial f}{\partial w}$.
4. Check that the evaluated gradient matches $\nabla f(w) = X^T X w - X^T y$.

If you read the code in `main.py`, you will notice that the gradient is automatically generated, not via a hard-coded value like we've been doing. This is because the operations involved in computing f are differentiable, and there are closed-form solutions for computing Jacobians. These Jacobians are computed during the `backward()` method call.

3.1 Implementing Logistic Loss

Implementing functions that are tedious to differentiate, e.g. logistic loss, becomes significantly easier once we start using differentiable programming. Just to refresh your memory, the logistic loss is defined as:

$$f(w) = \sum_{i=1}^{n} \log\left(1 + \exp(-y_i w^T x_i)\right)$$

where $y_i \in \{-1, +1\}$.

Finish the implementation of logistic loss (`f_var`) using the composition of differentiable operations and ensure that the test passes. Submit your code snippet from `main.py`.

You can run `python main.py -q 3.1` to check the correctness of your implementation.

Hint: you will want to think about using vectorized operations for computing the loss. I provided `nampy.sum()` which is the differentiable version of `np.sum()` which you can use in conjunction with the vectorized compositions of differentiable operations.

3.2 Implementing Forward and Backward Calls

Inside `nampy.py`, you will see a bunch of classes that implement `forward()` and `backward()` methods, as well as "shortcut" methods like `sum()`, `exp()`, and `log()`. Some of them are left for you to finish implementing. Finish implementing the following items:

- Element-wise sigmoid activation: complete the shortcut method named `sigmoid()`. Use the already-implemented functions and compose their constructors.
- Element-wise ReLU activation: the shortcut `relu()` is already written, but you must complete the class named `Maximum`. Implement the `forward()` method.
- Matrix multiplication: complete the class named `MatMul`.

You can run `python main.py -q 3.2` to check the correctness of your implementation. Ensure that these tests pass.

Hint: you will find `np.einsum()` especially useful for implementing matrix multiplication.

3.3 Implementing Multi-Output Linear Regression

Recall that the multi-output least squares uses an $n \times q$ matrix Y as labels and uses the following objective function:

$$f(W) = \frac{1}{2}||XW^T + b - Y||_F^2 = \sum_{c=1}^{q} \frac{1}{2} \left(Xw_c + \beta_c - y_c\right)^2$$

where W is the $q \times d$ matrix (equivalent of storing q columns of linear model weights), and b is a $q \times 1$ vector, whose entry β_c corresponds to the bias for label column c. Note that I'm abusing the notation to write $XW^T + b$, which is meant to add b to every row of XW^T (aka broadcasting).

In `generic_models.py`, you will find a class named `GenericSupervisedModel`. We have reached the point where the behaviour of a model is truly separated into the three things: (1) the objective function, (2) the optimization strategy, and (3) how the predictions are generated. Now all that a "model" does is to orchestrate the interaction among these components and the provide a simple interface via its two main methods: `fit()` and `predict()`.

In `mappers.py`, you will find a class named `MapperLinear`. The `reset()` method of this class instantiates the weight matrix W and the bias vector b as `Variable` instances. You will want to pay attention to how `compose()` and `map()` methods work: differentiable compositions with the variables W and b will be used, as opposed to native NumPy arrays. Now that you've implemented the differentiable matrix multiplication, you should be able to find the f value and the gradient $\nabla f(W)$ using `forward()` and `backward()`.

In `fun_obj.py` you will find a class `FunObjLeastSquaresMultiOutput`. Finish the implementation of the multi-output least squares loss using the composition of differentiable operations. Submit your `evaluate()` method.

You can run `python main.py -q 3.3` to check the correctness of your implementation. Ensure that these tests pass.

3.4 Implementing Neural Networks Regression

We now have all the ingredients needed to implement a neural network with differentiable programming. Recall that an objective function for a single-layer neural network may look like this:

$$f(v, W) = \frac{1}{2}\sum_{i=1}^{n}\left(v^T h(Wx_i + b_W) + b_v - y\right)^2$$

where W is the encoder's $k \times d$ matrix and v is the predictor's $k \times 1$ matrix. b_W and b_v are the biases for the encoder and the predictor respectively. Now, consider a multi-output single-layer neural network:

$$f(V, W) = \frac{1}{2}\sum_{i=1}^{n}\left(h(Wx_i + b_W)V^T + b_V - Y\right)^2$$

where Y is the multi-output labels captured into an $n \times q$ matrix (i.e. labels are q-dimensional), hence the predictor's weights are now a $q \times k$ matrix. b_W and b_V are the biases for the encoder and the predictor respectively

This objective function can be re-written as a Frobenius norm:

$$f(V, W) = \frac{1}{2}||h(XW^T + b_W)V^T + b_V - Y||_F^2 = \frac{1}{2}||\hat{Y} - Y||_F^2$$

where $\hat{Y}$ is the $n \times q$ prediction generated by performing a *forward pass* routine of the neural network, i.e. performing all matrix multiplications and non-linear activations.

Inside `mappers.py`, you will find a class named `MapperNeuralNet`, whose constructor takes a mapper for the encoder and a mapper for the predictor as input arguments. This class simply composes the encoder and the predictor together while maintaining differentiability, and our `FunObjLeastSquaresMultiOutput` and `GenericSupervisedModel` classes can work right out of the box.

The only thing left for you to do is to implement a class for a non-linear encoder as we saw in Assignment 6, where we compose multiple matrix multiplication-based encoders together with non-linear activations. In `mappers.py`, complete the `map()` method inside the `MapperMultiLayer` class so that it uses the ReLU activation. In `main.py`, assemble a neural network as a `GenericSupervisedModel` to fit the provided synthetic dataset. Submit your `map()` and the `main.py` code instantiating a neural network.

You can run `python main.py -q 3.4` to check the correctness of your implementation. Ensure that these tests pass.