Homework 1 Bonus
An Introduction to Autograd

11-785: Introduction to Deep Learning (Fall 2023)

OUT: **September 6th**
DUE: **November 23rd 11:59pm EST**
VERSION: **1.0**

Start Here

- **Collaboration policy:**
  - You are expected to comply with the University Policy on Academic Integrity and Plagiarism.
  - You are allowed to talk with / work with other students on homework assignments
  - You can share ideas but not code, you must submit your own code. All submitted code will be compared against all code submitted this semester and in previous semesters using MOSS.
- **Overview:**
  - **MyTorch:** An explanation of the library structure, the local autograder, and how to submit to Autolab. You can get the starter code from Autolab or the course website.
  - **Autograd:** An introduction to Automatic Differentiation and our implementation of Autograd, the various classes, their attributes and methods.
  - **Building Autograd:** The assignment, building the autograd engine, adding backward operations, and using this toolkit to build an MLP!
  - **Appendix:** This contains information on some theory that will be helpful in understanding the homework.
- **Directions:**
  - You are required to do this assignment in the Python (version 3) programming language. Do not use any auto-differentiation toolboxes (PyTorch, TensorFlow, Keras, etc) - you are only permitted and recommended to vectorize your computation using the Numpy library.
  - We recommend that you look through all of the problems before attempting the first problem. However we do recommend you complete the problems in order, as the difficulty increases, and questions often rely on the completion of previous questions.
  - If you haven't done so, use pdb to debug your code effectively (One much simple way, just use print()! Print the shape or anything else that can help you find the bug!) and please PLEASE Google your error messages before posting on Piazza.

1 Objective

If you complete this bonus homework successfully, you would ideally have learned:

- How the Automatic Differentiation framework works
- How to build an autograd engine for reverse mode autodiff (Autograd)
  - How to save operations consisting of inputs, outputs, gradients, backward functions
  - How to implement backward functions for each operation
  - How to backpropagate through the operations
  - How to update the gradients for operation inputs
- How to build activations, losses using Autograd
  - How to decompose modules to the granularity of operations
- How to build and run a MLP with your Autograd implementation

2 MyTorch Structure

In HW1P1, your implementation of MyTorch worked at the granularity of a single layer - thus, stacking several Linear Layers followed by activations (and, optionally, BatchNorm) allowed you to build your very own MLP. In this bonus assignment, we will build an alternative implementation of MyTorch based on a popular Automatic Differentiation framework called Autograd that works at the granularity of a single operation. As you will discover, this alternate implementation more closely resembles the internal working of popular Deep Learning frameworks such as PyTorch and TensorFlow (version 2.0 onwards), and offers more flexibility in building arbitrary network architectures. For Homework 1 Bonus, MyTorch will have the following structure:

handout

- `mytorch/`
  - `autograd_engine.py`
  - `utils.py`
  - `sandbox.py`
  - `nn/`
    - `functional_hw1.py`
    - `modules/`
      - `activation.py`
      - `linear.py`
      - `loss.py`
- `hw1_bonus/`
  - `data`
  - `mlp.py`
  - `mlp_runner.py`
- `autograder/`
  - `runner_hw1.py`
  - `helpers.py`
  - `test_activation.py`
  - `test_autograd.py`
  - `test_basic_functional.py`
  - `test_linear.py`
  - `test_loss.py`
- `create_tarball.sh`

- **Install** Python3, NumPy and PyTorch in order to run the local autograder on your machine:

```
pip install numpy
pip install torch
```

- **Hand-in** your code by running the following command from the top level directory (**inside** `handout/` **and outside** `mytorch/`), then **SUBMIT** the created *handin.tar* file to autolab:

```
sh create_tarball.sh
```

- **Autograde** your code by running the following command **inside** `autograder/`:

```
python3 runner_hw1.py
```

- **DO:**
  - We strongly recommend that you understand working of the Autograd Engine before you start coding.
- **DO NOT:**
  - Import any other external libraries other than numpy, as extra packages that do not exist in autolab will cause submission failures. Also do not add, move, or remove any files or change any file names.

3 Autograd

3.1 Background : Automatic Differentiation

Automatic Differentiation [1], or "Autodiff", is a framework that allows us to calculate the derivatives of any arbitrarily complex mathematical function. It does so by repeatedly applying the chain rule of differentiation since all computer functions can be rewritten in the form of nested differentiable operations. Autodiff, which is different from Symbolic Differentiation, and Numerical Differentiation, has several desirable properties: two that we care most about are computational efficiency, and numerical accuracy. In practice, there are several different ways to implement autodiff, which can be broadly categorised into two types - forward accumulation, or forward mode (which computes the derivatives of the chain rule from inside to outside) and reverse accumulation, or reverse mode (which computes the derivatives of the chain rule from outside to inside). "Autograd" is just one such implementation of reverse mode automatic differentiation, which is most widely used in the context of machine learning applications.

3.2 Autograd and Backprop

Recall from Lecture 2, "The neural network as a universal approximator", that neural networks are just large, large functions. Also recall from Lecture 3, that in order to train a neural network, we need to calculate the derivatives (or gradients) of this large function (with respect to its inputs) - which is the backpropagation algorithm - and use these gradients in an optimisation algorithm such as gradient descent to update the parameters of the network. Finally, recall from earlier in this writeup that autodiff provides an efficient way to compute exactly these required gradients by repeatedly applying the chain rule. The Autograd framework keeps track of the sequence of operations that are performed on the input data leading up to the final loss calculation. It then performs backpropagation and calculates all the necessary gradients.

3.3 Implementation Details

While several popular implementations of Autograd deal with complex data structures (such as computational graphs), our implementation will be far simpler and resemble a single "linear" sequence of operations going forward and backward. This is based on the key observation that regardless of the actual network architecture that one constructs (a graph, or otherwise) there is a sequential order in which all operations can be performed in order to achieve the correct result. (Readers who have a CS background may draw an analogy with the concept of serialized transactions/operations in distributed/parallel computing). We break down our implementation into two main classes - the `Operation`, and the `Autograd` classes - and a single helper class (`GradientBuffer`).

3.3.1 Operation Class

The objects of this class represent every operation that is performed in the network. Thus, for every operation that you perform on the data (say, multiplication, or addition), you will need to initialize a new `Operation` object that specifies the type of operation being performed. Note that to calculate the derivative of any operation in the network, we need to know the inputs that were passed to this node, and the outputs that were generated. Storing the type of operation, the inputs, and the outputs are the primary responsibilities of the `Operation` class.

Class attributes:

- `inputs`: The inputs to the operation.
- `outputs`: The output(s) generated by applying the operation to the inputs.
- `gradients_to_update`: These are the gradients corresponding to the operation inputs that must be updated on the backward pass.
- `backward_function`: A backward function implemented for a specific operation (ex: `add_backward` for operation add - see section 4.1.2 for more details). This function is called during backward pass to calculate and update the gradients for operation inputs.

3.3.2 Autograd Class

This is the main class for autograd engine that is responsible for keeping track of the sequence of operations being performed, and kicking off the backprop algorithm once the forward pass is complete.

Class attributes:

- `gradient_buffer`: An instance of the `GradientBuffer` class, used to store a mapping between input data and their gradients.
- `operation_list`: A Python list that is used to store sequence of operations that are performed on the input data. Concretely, this stores `Operation` objects.

Class methods:

- `add_operation(inputs, output, gradients_to_update, backward_operation)`: Initialises a new instance of `Operation` with given arguments, and adds it to `operation_list`.
- `backward(divergence)`: Kicks off backpropagation. Traverses the `operation_list` in reverse and calculates the gradients at every node.

For this assignment, you will need to implement `add_operation`and `backward` methods.

3.3.3 GradientBuffer Class

This is a simple wrapper class around a Python dictionary with a few useful methods that allow for storing and updating the gradients. While it's not necessary to modify this class for your assignment, we strongly recommend familiarizing yourself with the class attributes and methods to gain a better understanding of its functionality.

Class attributes:

- `memory`: A Python dictionary that holds the NumPy array corresponding to its gradient. For a given NumPy array `np_array`, the key is the memory location of `np_array` and the value is the gradient array associated with `np_array`. Note: Using the memory location as a key is a simple trick that eliminates the need to perform extra bookkeeping of maintaining unique keys for all gradients.

Class methods:

- `get_memory_loc(np_array)`: Returns the memory location of `np_array`, used in other functions to get keys.
- `is_in_memory(np_array)`: Checks if a gradient array corresponding to `np_array` is already in memory.
- `add_spot(np_array)`: Allocates a zero gradient array corresponding to `np_array` in `memory`.
- `update_param(np_array, gradient)`: Increments the gradient array corresponding to `np_array` by the amount of `gradient`.
- `get_param(np_array)`: Returns the gradient array corresponding to `np_array`.
- `clear()`: Clears the `memory` dictionary.

3.4 Example Walkthrough

Suppose that we are building a single layer MLP. Therefore, we perform the following operations on input data x:

$$h = x * W \tag{1}$$

$$y = h + b \tag{2}$$

The following steps need to be executed for this simple operation:

- First, we need to create an instance of autograd engine using:

  ```
  autograd = autograd_engine.Autograd()
  ```

- For equation (1), we need to add a node to the computation graph performing multiplication, which would be done in the following way:

  ```
  autograd_engine.add_operation(
      inputs = [x, W], output = h,
      gradients_to_update = [None, dW],
      backward_operation = matmul_backward
  )
  ```

- Similarly for equation (2),

  ```
  autograd_engine.add_operation(
      inputs = [h, b], output = y,
      gradients_to_update = [None, db],
      backward_operation = add_backward
  )
  ```

- Invoke backpropagation by:

  ```
  autograd_engine.backward(divergence)
  ```

  `dW` and `db` should be updated after this.

The concept above could be leveraged in building more complex computation steps (with few lines of code).

Gradients for Input Data vs. Gradients for Network Parameters

In our implementation, there are two types of operation inputs: input data and network parameters, and we track their gradients by different ways. In the above example, `x` and `h` are input data, `W` and `b` are network parameters. For input data, there is no explicit gradient passed (note how we set `gradients_to_update = None` for those), so we store them internally with gradient buffer. In the chain of operations, the output for a previous operation will always be the input for a later operation at the same time. Therefore, when we retrieve the gradient for the output of current operation, it would be updated already as the input to the backward function of later operations (consider `h` in the above example). For network parameters, their gradients are tracked externally with passed arguments (ex: `dW` and `db`). Because the network stores these variables, the autograd engine can access the value from the network without storing them internally.

4 Building Autograd

Now that we've covered the working details of our implementation of Autograd, it is time to look at what the actual assignment is. The main components that you need to complete for this bonus assignment are:

1. Complete the Autograd Engine [10]
2. Add Operation Backward Functions [6]
3. Build Modules [25]
4. Build a Network [9]

4.1 Completing the Autograd Engine [10]

4.1.1 Forward [5]

Complete the `add_operation` function in `autograd_engine.py`. This method should do the followings:

1. Initialise a spot in the gradient buffer for each of the inputs whose gradients needs to be internally tracked, i.e input data. Hint: We can check how gradients are tracked by checking `gradients_to_update`.
2. Create an `Operation` object with the correct arguments (note, these are already being passed to `add_operation()`), and append this object to the `operation_list` of the `Autograd` class.

In the single layer MLP example in 3.4, we will allocate a spot to store gradients for `x`, and append an add `Operation` for Equation (1). Similarly, we allocate a spot for `h`, and append a matmul `Operation` for Equation (2).

4.1.2 Backward [5]

Complete the `backward(divergence)` function in `autograd_engine.py`:

1. Iterate over the `operation_list` in reverse order.
2. Retrieve the "grad of output": this is the gradient of the output that was initialized in the forward pass and has now been calculated during this backward pass. Recall that we added a spot for this in the `GradientBuffer` and only need to retrieve it using the input. Note: the output for the final operation is the divergence and is just passed to the backward function explicitly.
3. Call the operation's backward operation on this "grad of output".
4. Finally, iterate over the inputs, gradients of the inputs, and the gradients to update to update the appropriate gradients for that operation. Note: For gradients that are tracked externally, we should update the value in `gradients_to_update` (ex: `dW` and `db` in 3.4); for gradients that are tracked internally, we should update the gradients stored in `GradientBuffer` (ex: `x` and `h` in 3.4).

We further explain using example in 3.4. In reverse order, we first backprop the operation for Equation (2). Since this is the first operation in backpropagation stage, "grad of output" is simply the argument `divergence`. We call `add_backward` to get the gradients w.r.t. `h` and `b`. We update the gradient stored in gradient buffer for `h` since it is tracked internally, while we update the variable `db` from `gradients_to_update` directly. Next, we backprop the operation for Equation (1). Here, the output is `h`, the gradient of which is already calculated and stored in gradient buffer as mentioned before. Therefore, we retrieve "grad of ouput" by accessing the gradient buffer for output `h`. Similarly, we call `matmul_backward` to get the gradients w.r.t. `x` and `W`, update gradient for `x` in gradient buffer and update variable `dW`.

Important: Note that there is a one-to-one correspondence between the inputs list passed to the Operation object and the gradients to update list. Make sure that when you call `add_operation` in the remaining parts of the homework, you pass the inputs and corresponding gradients **in the same order**.

4.2 Adding Operation Backward Functions [6]

Recall that Autograd is defined to work at the operation level and not the layer level. This means that we must define a backward function for every operation that we will use. Note that **you have already calculated and implemented these backward functions** in HW1P1 - you just need to define them individually and explicitly now.

In `nn/functional.py` complete the following

- `sub_backward` [1]
- `matmul_backward` [1]
- `mul_backward` [1]
- `div_backward` [1]
- `log_backward` [1]
- `exp_backward` [1]

These functions should return gradients to the inputs of a single operation in the same order. That is, for a operation `f(x,y,z)`, `f_backward` should calculate and return `dx,dy,dz`. This is not a complete list of possible operations so you may need to add more as you see fit. As a reference, we already provide `add_backward()`.

Important: The provided tests check **completeness instead of correctness**. You need to write your own test cases for your implementation in this part.

4.3 Building Modules [25]

4.3.1 Activations [20]

In `nn/modules/activations.py`, complete the classes for the activations - Identity [5], Sigmoid [5], ReLU [5], and Tanh [5]. Feel free to add more. Each activation function is a chain of simple operations that you implemented in 4.2 above. You need to identify the operations and call `add_operation` appropriately in `forward()`.

Note: Please note that you will have to complete `Linear` from `nn/modules/linear.py` before you work on Activation functions.

4.3.2 Loss Functions [5]

In `nn/modules/loss.py`, complete the class `SoftmaxCrossEntropy`. One option is that you can decompose the loss function into simple operations similar to what you've done for activations (you may need to add additional backward functions in `nn/functional.py`). Alternatively, you can take advantage of the simple form of the derivative of the SoftmaxCrossEntropy Loss (remember HW1P1). SoftmaxCrossEntropy can be viewed as **one** special operation and you can implement a special backward function for this operation in `nn/functional.py`. You may want to include additional inputs to this operation based on your implementation. Hint: you may set the arguments to the backward function so that the gradients can be easily calculated from the arguments.

Optional: Complete the `MSELoss` class.

4.4 Building a Network [9]

Finally, you can put to use the new Autograd module that you have so painstakingly crafted to actually build a MLP Network.

4.4.1 A simple Linear Layer [9]

Why return to the Layer Level abstraction if Autograd lives at the Operation Level Abstraction? Defining Layer classes is an effort to regress to good coding practices. Concretely, we achieve the following by doing this:

- We hide the autograd engine object and do not explicitly expose its internals to the user.
- We make our code more modular - defining a Linear Layer class for example, avoids repeatedly making calls to the add operation function of the autograd engine.
- This is how most deep learning frameworks such as PyTorch define layers and we believe this homework will help solidify that syntax in your mind.

In the file `nn/modules/linear.py` complete the `Linear` class. This is a simple wrapper class around the matmul and addition operations to implement the affine transformation of the Linear Layer. Your task is to complete the `forward()` function of this class:

1. First, actually compute the outputs for the forward computation from the given input and the layer parameters.
2. Note the parameters that are being passed to the forward method.
3. Remember the syntax for using the `add_operation()` function.
4. Also remember to ensure that the input parameters are in the same order as the gradients to update parameters.

4.4.2 Building an MLP [Ungraded]

In `hw1_bonus/mlp.py`, stack Linear layers and activations together to build an the MLP class by implementing the following network architecture: Input → Linear Layer → ReLU → Linear Layer → Sigmoid → Output Note that this section is not graded and is simply provided as proof-of-concept for the Autograd library that you have built.

5 Limitations of MyTorch Autograd

By now you should realize that our Autograd is very different from the production-level automatic differentiation system, e.g. PyTorch that you are using in Homework P2s, where you don't need to manually construct the computational graph via `add_operation` in `forward` at all. This is because PyTorch encapsulates all Autograd utilities in the abstraction of `Tensor`. PyTorch overloads the operators for `Tensor` such as matrix multiplication, elementwise multiplication, etc., and automatically constructs the computational graph under the hood for you. Each `Tensor` instance would store the gradient so that you would not need to store them in an explicit way as we do. You would be able to easily access the gradient value via `.grad` if the tensor is specified to require gradients. Also, PyTorch constructs computational graphs in the form of a DAG instead of a linked list in our case. This allows more performance optimization, more flexible computation, and more efficient memory usage. For more details, please refer to [4].

6 Beyond Deep Learning

(Optional Reading for the Interested Student) The concept of Autodiff is not new - in fact, it significantly predates Deep Learning. Justin Domke's notes on Autodiff [2] provide an interesting perspective on the state of ML research, as it were roughly 10 years ago, and its surprising ignorance of autodiff. As [2] explains, much of the effort in incumbent ML research lay around calculating derivatives for complex objective functions by hand - this could all have been abstracted away using some simple implementation of autodiff. As time shows, this is exactly what has happened over the last 5 years - several ML/DL frameworks have sprung up, all of which implement autodiff/autograd and abstract away the need to calculate complex derivatives by hand.

The Autograd that you have built in this assignment is a very powerful tool which can be applied to problems far outside the realm of deep learning - from solving partial differential equations, to computational finance. The simple, core idea is that it is possible to calculate the derivative of any computable function, simply by applying the chain rule over and over again.

7 References

[1] Automatic Differentiation

[2] Justin Domke, "Automatic Differentiation: The most criminally underused tool in the potential Machine Learning toolbox?"

[3] William Cohen, "Automatic Reverse-Mode Differentiation"

[4] "PyTorch: An Imperative Style, High-Performance Deep Learning Library"