Homework 1 - Bonus

An Introduction to Autograd

11-785: Introduction to Deep Learning (Spring 2021)

OUT: **February 9, 2021**
DUE: **March 7, 2021, 11:59 PM EST**

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:**
  - **NewGrad**: 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 that you may (read: will) find useful.
- **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.

1 MyTorch

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:

- new_grad
  - mytorch/
    - autograd_engine.py
    - utils.py
    - nn/
      - functional.py
      - modules
      - linear.py
      - loss.py
  - autograder/
    - runner.py
  - create_tarball.sh

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

```
pip3 install numpy
pip3 install torch
```

- **Hand-in** your code by running the following command from the top level directory, then **SUBMIT** the created *handin.tar* file to autolab:

```
sh create_tarball.sh
```

- **Autograde** your code by running the following command from the top level directory:

```
python3 autograder/runner.py
```

- **DO:**
  - We strongly recommend that you understand working of the AutogradEngine 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.

2 Introduction

2.1 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.

2.2 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.

2.3 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 our implementation into two main classes - the Operation, and the AutogradEngine classes - and a single helper class (MemoryBuffer).

2.3.1 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, for our implementation, we refer to the input data, and network parameters as 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 inputs that must be updated on the backward pass.

- backward_function : A backward function implemented for a specific operation (ex : add_backward - see section 3.1 for more details). This is used to specify what type of operation was performed and to compute derivatives by calling the appropriate backward function.

2.3.2 AutogradEngine Class

This is the main Autograd class 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:

- memory_buffer : An instance of the MemoryBuffer class, used to store the 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_node : Initialises and adds an instance of the Operation class to the operations_list.
- backward : Kicks off backprop. Traverses the operations_list in reverse and calculates the gradients at every Node.

2.3.3 MemoryBuffer Class

This is a simple wrapper class around a Python dictionary with a few useful methods that allow for updating the gradients

Class attributes:

- memory: A Python dictionary that holds the NumPy arrays corresponding to the gradients of the network. The key is the memory location of the NumPy array and the value is the actual 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: Returns the corresponding parameter array (could be a gradient or something else depending on what the MemoryBuffer is used for) for the input array.
- is_in_memory: Checks if an array is already provided for in memory.
- add_spot: Initialises a location in memory for a new array.
- update_param: Updates the parameter in memory by adding a new update to it.
- get_param: Returns the appropriate param from memory dict.
- clear: Resets the memory dictionary to effectively zero out all gradients.

2.4 Example Walkthrough

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

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

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

Following steps needs to be executed for this simple operation:

- 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_node(inputs = [x, W], output = h,`
  `gradients_to_update = [None, dW], backward_operation = matmul_backward)`
- Similarly for equation (2),
  `autograd_engine.add_node(inputs = [h, b], output = y,`
  `gradients_to_update = [None, db], backward_operation = add_backward)`
- After these operations have been added to the operation list, we can iterate through the operation_list to perform back propagation.

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

3 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,
2. Add Backward Functions,
3. Build a Network,

3.1 Completing the Autograd Engine [10]

First, complete the add_operation [**5**] function in autograd_engine.py. This method should do the following:

1. initialise a spot in the memory buffer for each of the inputs,
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

Next, complete the backward() function in autograd_engine.py [**5**]:

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 MemoryBuffer 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_input,
4. finally, iterate over the inputs, gradients of the inputs, and the gradients to update to update the appropriate gradients for that operation

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.

3.2 Adding Operation Backward Functions

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.

3.2.1 In nn/functional.py complete the following [6] :

- sub_backward [**1**],
- matmul_backward [**1**],
- mul_backward [**1**],
- divide_backward [**1**],
- log_backward [**1**],
- exp_backward [**1**],

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().

3.2.2 Activations in nn/modules/activations.py [20]

Similarly, complete the classes for the activations - Identity [**5**], Sigmoid [**5**], ReLU [**5**], and Tanh [**5**] which are simply wrapper classes around the operations that you implemented in 3.2.1 above. Feel free to add more.

3.2.3 Loss Function in nn/modules/loss.py [5]

Complete the class SoftmaxCrossEntropy in nn/modules/loss.py. For this you may either choose to implement additional backward functions in the functional.py file. Alternatively, you can take advantage of the simple form of the derivative of the SoftmaxCrossEntropy Loss (remember HW1P1) and implement a special backward function for this loss directly in functional.py.
There is also an MSE Loss class which you can optionally complete. Feel free to add more.

3.3 Building a Network

Finally, in hw1_bonus/mlp.py you can put to use the new Autograd module that you have so painstakingly crafted to actually build a Neural Network.

3.3.1 A simple Linear Layer [9]

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.

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.

3.3.2 Building an MLP [Ungraded]

Finally in hw1_bonus.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.

4 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.

5 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" : http://www.cs.cmu.edu/ wcohen/10-605/assignments/2016-fall/drafts/autodiff.pdf