10-714: Homework 1

This homework will get you started with your implementation of the **needle** (**ne**cessary **e**lements of **d**eep **le**arning) library that you will develop throughout this course. In particular, the goal of this assignment is to build a basic **automatic differentiation** framework, then use this to re-implement the simple two-layer neural network you used for the MNIST digit classification problem in HW0.

First, as you did for HW0, make a copy of this notebook file by selecting "Save a copy in Drive" from the "File" menu, and then run the code block below. Then run the code below to set up and install the necessary packages.

```
# Code to set up the assignment
from google.colab import drive
drive.mount('/content/drive')
%cd /content/drive/MyDrive/
!mkdir -p 10714
%cd /content/drive/MyDrive/10714
!git clone https://github.com/dlsys10714/hw1.git
%cd /content/drive/MyDrive/10714/hw1

!pip3 install --upgrade --no-deps git+https://github.com/dlsys10714/mugrade.git
!pip3 install numdifftools


import sys
sys.path.append('./python')
sys.path.append('./apps')
from simple_ml import *
```

Introduction to `needle`

For an introduction to the `needle` framework, refer to Lecture 5 in class and this Jupyter notebook from the lecture. For this homework, you will be implementing the basics of automatic differentiation using a `numpy` CPU backend (in later assignments, you will move to your own linear algebra library including GPU code). All code for this assignment will be written in Python.

For the purposes of this assignment, there are two important files in the `needle` library, the `python/needle/autograd.py` file (which defines the basics of the computational graph framework, and also will form the basis of the automatic differentiation framework), and the `python/needle/ops.py` .file (which contains implementations of various operators that you will use implement throughout the assignment and the course).

Although the basic framework for automatic differentiation is already set up in the `autograd.py` file, you should familiarize yourself with the basic concepts of the library as it relates to a few different defined classes. Note that we would **not** recommend attempting to read through the entire code base before starting your implementations (some of the functionality will likely make more sense after you have implemented something), but you should have a basic understanding of the basic structure and organization of the classes that needle defines. Specifically, you should get familiar with the basic concepts behind the following classes:

- `Value`: A value computed in a compute graph, i.e., either the output of some operations applied to other `Value` objects, or a constant (leaf) `Value` objects We use a generic class here (which we then specialize to e.g. Tensors), in order to allow for other data structures in later version of needle, but for now you will interact with this class mostly through its subclass `Tensor` (see below).
- `Op`: An operator in a compute graph. Operators need to define their "forward" pass in the `compute()` method (i.e., how to compute the operator on the underlying data of the `Value` objects), as well as their "backward" pass via the `gradient()` method, which defines how to multiply by incoming output gradients. The details of writing such operators will be given below.
- `Tensor`: This is a subclass of `Value` that corresponds to an actual tensor output, i.e., a multi-dimensional array within a computation graph. All of your code for this assignment (and most of the following) will use this subclass of `Value` rather than the generic class above. We have provided several convenience functions (e.g., operator overloading) that let you operate on tensor using normal Python conventions, but these will not work properly until you implement the corresponding operations.
- `TensorOp`: This is a subclass for `Op` for operators that return Tensors. All the operations you implement for this assignment will be of this type.

Question 1: Implementing forward computation [10 pts]

First, you will implement the forward computation for new operators. To see how this works, consider the `EWiseAdd` operator in the `ops.py` file:

```
class EWiseAdd(TensorOp):
    def compute(self, a: NDArray, b: NDArray):
        return a + b

    def gradient(self, out_grad: Tensor, node: Tensor):
```

```
        return out_grad, out_grad


def add(a, b):
    return EWiseAdd()(a, b)
```

The conventions for implementations of this class are the following. The `compute()` function computes the "forward" pass, i.e., just computes the operation itself. However, it is important to emphasize the inputs to compute are both `NDArray` objects (i.e., in this initial implementation, they are just `numpy.ndarray` objects, though in a later assignment you will implement your own NDArray). That is, `compute()` computes the forward pass on the *raw data objects* themselves, not on Tensor objects within the automatic differentiation.

We will discuss the `gradient()` call in the next section, but it is important to emphasize here that this call is different from forward in that it takes `Tensor` arguments. This means that any call you make within this function *should* be done via `TensorOp` operations themselves (so that you can take gradients of gradients).

Finally, note that we also define a helper `add()` function, to avoid the need to call `EWiseAdd()(a,b)` (which is a bit cumbersome) to add two `Tensor` objects. These functions are all written for you, and should be self-explanatory.

For this question, you will need to implement the `compute` call for each of the following classes. These calls are very straightforward, and should be essentially one line that calls to the relevant numpy function. Note that because in later homeworks you will use a backend other than numpy, we have imported numpy as `import numpy as array_api`, so that you'll need to call `array_api.add()` etc, if you want to use the typical `np.X()` calls.

- `PowerScalar`: raise input to an integer (scalar) power
- `EWiseDiv`: true division of the inputs, element-wise (2 inputs)
- `DivScalar`: true division of the input by a scalar, element-wise (1 input, `scalar` - number)
- `MatMul`: matrix multiplication of the inputs (2 inputs)
- `Summation`: sum of array elements over given axes (1 input, `axes` - tuple)
- `BroadcastTo`: broadcast an array to a new shape (1 input, `shape` - tuple)
- `Reshape`: gives a new shape to an array without changing its data (1 input, `shape` - tuple)
- `Negate`: numerical negative, element-wise (1 input)
- `Transpose`: reverses the order of two axes (axis1, axis2), defaults to the last two axes (1 input, `axes` - tuple)

```
!python3 -m pytest -v -k "forward"
```

```
============================= test session starts ==============================
platform darwin -- Python 3.9.7, pytest-6.2.4, py-1.10.0, pluggy-0.13.1 -- /Users/zkolter/opt/anaconda3/bin/python3
cachedir: .pytest_cache
rootdir: /Users/zkolter/Dropbox/class/10-714/homework/hw1
plugins: pytest_check-1.0.4, anyio-2.2.0
collected 21 items / 13 deselected / 8 selected

tests/test_autograd_hw.py::test_divide_forward PASSED                    [ 12%]
tests/test_autograd_hw.py::test_divide_scalar_forward PASSED             [ 25%]
tests/test_autograd_hw.py::test_matmul_forward PASSED                    [ 37%]
tests/test_autograd_hw.py::test_summation_forward PASSED                 [ 50%]
tests/test_autograd_hw.py::test_broadcast_to_forward PASSED              [ 62%]
tests/test_autograd_hw.py::test_reshape_forward PASSED                   [ 75%]
tests/test_autograd_hw.py::test_negate_forward PASSED                    [ 87%]
tests/test_autograd_hw.py::test_transpose_forward PASSED                 [100%]

======================= 8 passed, 13 deselected in 0.48s =======================
```

```
!python3 -m mugrade submit 'YOUR_GRADER_KEY_HERE' -k "forward"
```

Question 2: Implementing backward computation [25 pts]

Now that you have implemented the functions within our computation graph, in order to implement automatic differentiation using our computational graph, we need to be able to compute the backward pass, i.e., multiply the relevant derivatives of the function with the incoming backward gradients.

The easiest way to perform these computations is, again, via taking "fake" partial derivatives (assuming everything is a scalar), and then matching sizes: here the tests we provide will automatically check against numerical derivatives to ensure that your solution is correct.

The general goal of reverse mode auto-differentiation is to compute the gradient of some downstream function ℓ of $f(x, y)$ with respect to x (or y). Written formally, we could write this as trying to compute

gradient() 需要返回的目标 → $\frac{\partial \ell}{\partial x} = \frac{\partial \ell}{\partial f(x,y)} \frac{\partial f(x,y)}{\partial x}$.

The "incoming backward gradient" is precisely the term $\frac{\partial \ell}{\partial f(x,y)}$, so we want our `gradient()` function to ultimately compute the *product* between this backward gradient the function's own derivative $\frac{\partial f(x,y)}{\partial x}$.

To see how this works a bit more concretely, consider the element-wise addition function we presented above

$$f(x, y) = x + y.$$

Let's suppose that in this setting $x, y \in \mathbb{R}^n$, so that $f(x, y) \in \mathbb{R}^n$ as well. Then via simple differentiation

$$\frac{\partial f(x, y)}{\partial x} = 1$$

so that

$$\frac{\partial \ell}{\partial x} = \frac{\partial \ell}{\partial f(x, y)} \frac{\partial f(x, y)}{\partial x} = \frac{\partial \ell}{\partial f(x, y)}$$

i.e., the product of the function's derivative with respect to its first argument x is just exactly the same as the backward incoming gradient. The same is true of the gradient with respect to the second argument y. This is precisely what is captured by the following method of the `EWiseAdd` operator.

```
def gradient(self, out_grad: Tensor, node: Tensor):
    return out_grad, out_grad
```

i.e., the function just results the incoming backward gradient (which actually *is* here the product between the backward incoming gradient and the derivative with respect to each argument of the function). And because the size of $f(x, y)$ is the same as the size of both x and y, we don't even need to worry about dimensions here.

Now consider another example, the (element-wise) multiplication function

$$f(x, y) = x \circ y$$

where $\circ$ denotes element-wise multiplication between x and y. The partial derivative of this function is given by

$$\frac{\partial f(x, y)}{\partial x} = y$$

and similarly

$$\frac{\partial f(x, y)}{\partial y} = x$$

Thus to compute the product of the incoming gradient

$$\frac{\partial \ell}{\partial x} = \frac{\partial \ell}{\partial f(x, y)} \frac{\partial f(x, y)}{\partial x} = \frac{\partial \ell}{\partial f(x, y)} \cdot y$$

If $x, y \in \mathbb{R}^n$ like in the previous example, then $f(x, y) \in \mathbb{R}^n$ as well so the first element returned back the gradient function would just be the element-wise multiplication

$$\frac{\partial \ell}{\partial f(x, y)} \circ y$$

This is captures in the `gradient()` call of the `EWiseMul` class.

```
class EWiseMul(TensorOp):
    def compute(self, a: NDArray, b: NDArray):
        return a * b

    def gradient(self, out_grad: Tensor, node: Tensor):
        lhs, rhs = node.inputs
        return out_grad * rhs, out_grad * lhs
```

Implementing backward passes

Note that, unlike the forward pass functions, the arguments to the `gradient` function are `needle` objects. It is important to implement the backward passes using only `needle` operations (i.e. those defined in `python/needle/ops.py`), rather than using `numpy` operations on the underlying `numpy` data, so that we can construct the gradients themselves via a computation graph (one exception is for the `ReLU` operation defined below, where you could directly access data within the Tensor without risk because the gradient itself is non-differentiable, but this is a special case).

To complete this question, fill in the `gradient` function of the following classes:

- `EWiseDiv`
- `DivScalar`
- `MatMul`
- `Summation`
- `BroadcastTo`
- `Reshape`
- `Negate`
- `Transpose`

gradient() 中对 Tensor 进行操作的过程中，也逐步构建了导数的计算图

All of the `gradient` functions can be computed using just the operations defined in `python/needle/ops.py`, so there is no need to define any additional forward functions.

Hint: while gradients of multiplication, division, etc, may be relatively intuitive to compute it can seem a bit less intuitive to compute backward passes of items like `Broadcast` or `Summation`. To get a handle on these, you can check gradients numerically and print out their actual values, if you don't know where to start (see the `tests/test_autograd_hw.py`, specifically the `check_gradients()` function within that file to get a sense about how to do this). And remember that the size of `out_grad` will always be the size of the *output* of the operation, whereas the sizes of the `Tensor` objects *returned* by `gradient()` have to always be the same as the original *inputs* to the operator.

Checking backward passes

To reiterate the above, remember that we can check that these backward passes are correct by doing numerical gradient checking as covered in lecture:

$$\delta^T \nabla_\theta f(\theta) = \frac{f(\theta + \epsilon\delta) - f(\theta - \epsilon\delta)}{2\epsilon} + o(\epsilon^2)$$

We provide the function `gradient_check` for doing this numerical checking in `tests/test_autograd.py`.

```
!python3 -m pytest -l -v -k "backward"
```

```
============================= test session starts ==============================
platform darwin -- Python 3.9.7, pytest-6.2.4, py-1.10.0, pluggy-0.13.1 -- /Users/zkolter/opt/anaconda3/bin/python3
cachedir: .pytest_cache
rootdir: /Users/zkolter/Dropbox/class/10-714/homework/hw1
plugins: pytest_check-1.0.4, anyio-2.2.0
collected 21 items / 12 deselected / 9 selected

tests/test_autograd_hw.py::test_divide_backward PASSED                   [ 11%]
tests/test_autograd_hw.py::test_divide_scalar_backward PASSED            [ 22%]
tests/test_autograd_hw.py::test_matmul_simple_backward PASSED            [ 33%]
tests/test_autograd_hw.py::test_matmul_batched_backward PASSED           [ 44%]
tests/test_autograd_hw.py::test_reshape_backward PASSED                  [ 55%]
tests/test_autograd_hw.py::test_negate_backward PASSED                   [ 66%]
tests/test_autograd_hw.py::test_transpose_backward PASSED                [ 77%]
tests/test_autograd_hw.py::test_broadcast_to_backward PASSED             [ 88%]
tests/test_autograd_hw.py::test_summation_backward PASSED                [100%]

====================== 9 passed, 12 deselected in 0.53s ======================
```

```
!python3 -m mugrade submit 'YOUR_GRADER_KEY_HERE' -k "backward"
```

Question 3: Topological sort [20 pts]

Now your system is capable of performing operations on tensors which builds up a computation graph. Next you will write one of the main utilities needed for automatic differentiation - the topological sort. This will allow us to traverse through (forward or backward) the computation graph, computing gradients along the way. Furthermore, the previously built components will allow for the operations we perform during this reverse topological traversal to further add to our computation graph (as discussed in lecture), and will therefore give us higher-order differentiation "for free."

Fill out the `find_topo_sort` method and the `topo_sort_dfs` helper method (in `python/needle/autograd.py`) to perform this topological sorting.

Hints:

- Ensure that you do a post-order depth-first search, otherwise the test cases will fail.
- The `topo_sort_dfs` method is not required, but we find it useful to use this as a recursive helper function.
- The "Reverse mode AD by extending computational graph" section of the Lecture 4 slides walks through an example of the proper node ordering.
- We will be traversing this sorting backwards in later parts of this homework, but the `find_topo_sort` should return the node ordering in the forward direction.

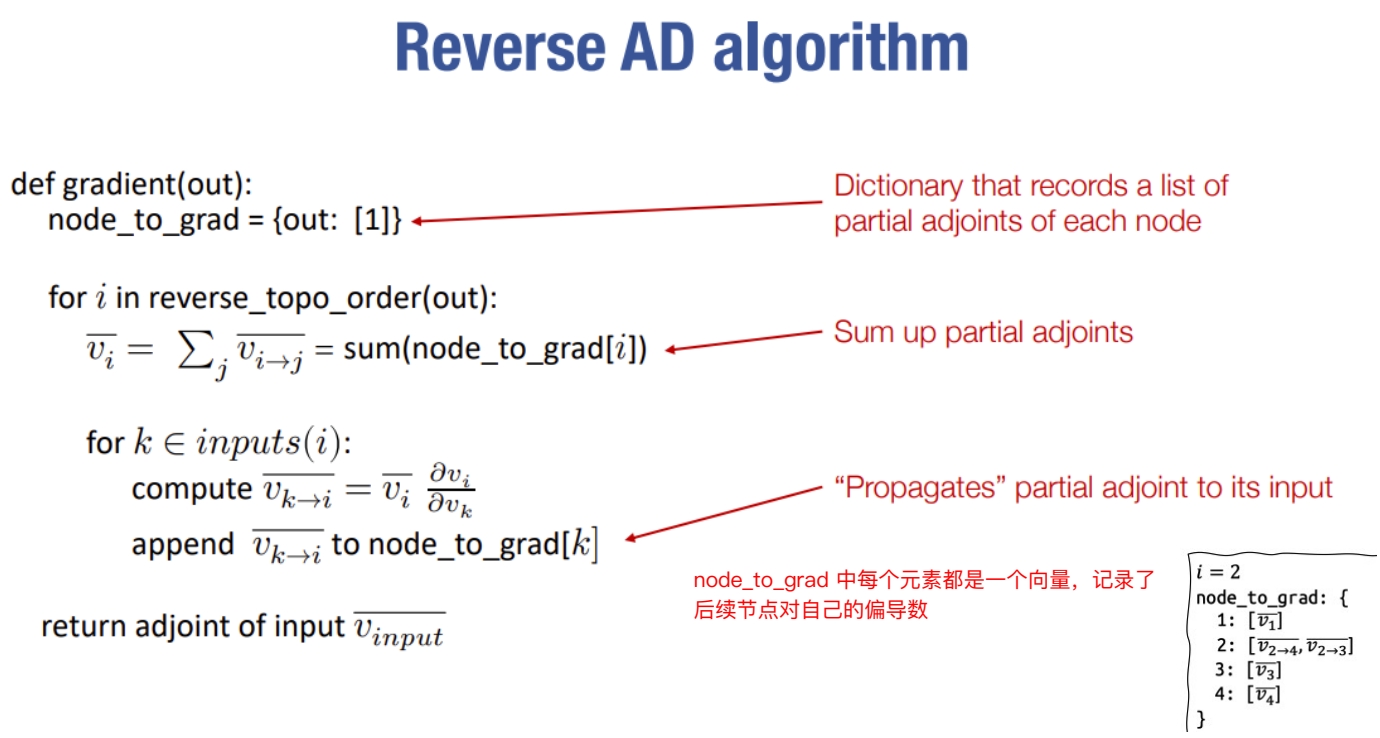
```
!python3 -m pytest -k "topo_sort"
```

```
============================= test session starts ==============================
platform darwin -- Python 3.9.7, pytest-6.2.4, py-1.10.0, pluggy-0.13.1
rootdir: /Users/zkolter/Dropbox/class/10-714/homework/hw1
plugins: pytest_check-1.0.4, anyio-2.2.0
collected 21 items / 20 deselected / 1 selected

tests/test_autograd_hw.py .                                              [100%]

====================== 1 passed, 20 deselected in 0.43s ======================
```

```
!python3 -m mugrade submit 'YOUR_GRADER_KEY_HERE' -k "topo_sort"
```

Question 4: Implementing reverse mode differentiation [25 pts]

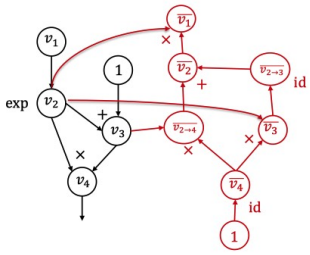
Once you have correctly implemented the topological sort, you will next leverage it to implement reverse mode automatic differentiation. As a recap from last lecture, we will need to traverse the computational graph in reverse topological order, and construct the new adjoint nodes. For this question, implement the Reverse AD algorithm in the `compute_gradient_of_variables` function in `python/needle/autograd.py`. This will enable use of the `backward` function that computes the gradient and stores the gradient in the `grad` field of each input `Tensor`. With this completed, our reverse mode auto-differentiation engine is functional. We can check the correctness of our implementation in much the same way that we numerically checked the individual backward gradients, by comparing the numerical gradient to the computed one, using the function `gradient_check` in `tests/test_autograd.py`.



As discussed in lecture the result of reverse mode AD is still a computational graph. We can extend that graph further by composing more operations and run reverse mode AD again on the gradient (the last two tests of this problem).

```
!python3 -m pytest -k "compute_gradient"
```

```
============================= test session starts ==============================
platform darwin -- Python 3.9.7, pytest-6.2.4, py-1.10.0, pluggy-0.13.1
rootdir: /Users/zkolter/Dropbox/class/10-714/homework/hw1
plugins: pytest_check-1.0.4, anyio-2.2.0
collected 21 items / 20 deselected / 1 selected

tests/test_autograd_hw.py .                                              [100%]

====================== 1 passed, 20 deselected in 0.47s =======================
```




```
!python3 -m mugrade submit 'YOUR_GRADER_KEY_HERE' -k "compute_gradient"
```

Question 5: Softmax loss [10 pts]

The following questions will be tested using the MNIST dataset, so we will use the `parse_mnist` function we wrote in the Homework 0.

1. First, copy and paste your solution to Question 2 of Homework 0 to the `parse_mnist` function in the `apps/simple_ml.py` file.

In this question, you will implement the softmax loss as defined in the `softmax_loss()` function in `apps/simple_ml.py`, which we defined in Question 3 of Homework 0, except this time, the softmax loss takes as input a `Tensor` of logits and a `Tensor` of one hot encodings of the true labels. As a reminder, for a multi-class output that can take on values $y \in \{1, \dots, k\}$, the softmax loss takes as input a vector of logits $z \in \mathbb{R}^k$, the true class $y \in \{1, \dots, k\}$ (which is encoded for this function as a one-hot-vector), and returns a loss defined by

$$\ell_{\text{softmax}}(z, y) = \log \sum_{i=1}^{k} \exp z_i - z_y.$$

You will first need to implement the forward and backward passes of one additional operator: `log`.

2. Fill out the `compute()` function in the `Log` operator in `python/needle/ops.py`.

3. Fill out the `gradient()` function in the `Log` operator in `python/needle/ops.py`.

Once those operators have been implemented,

4. Implement the function `softmax_loss` in `apps/simple_ml.py`.

You can start with your solution from Homework 0, and then modify it to be compatible with `needle` objects and operations. As with the previous homework, the function you implement should compute the *average* softmax loss over a batch of size m, i.e. logits `Z` will be an $m \times k$ `Tensor` where each row represents one example, and `y_one_hot` will be an $m \times k$ `Tensor` that contains all zeros except for a 1 in the element corresponding to the true label for each row. Finally, note that the average softmax loss returned should also be a `Tensor`.

```
!python3 -m pytest -k "softmax_loss_ndl"
```

```
============================= test session starts ==============================
platform darwin -- Python 3.9.7, pytest-6.2.4, py-1.10.0, pluggy-0.13.1
rootdir: /Users/zkolter/Dropbox/class/10-714/homework/hw1
plugins: pytest_check-1.0.4, anyio-2.2.0
collected 21 items / 20 deselected / 1 selected

tests/test_autograd_hw.py .                                              [100%]

======================= 1 passed, 20 deselected in 0.69s =======================
```

```
!python3 -m mugrade submit 'YOUR_GRADER_KEY_HERE' -k "softmax_loss_ndl"
```

Question 6: SGD for a two-layer neural network [10 pts]

As you did in Homework 0, you will now implement stochastic gradient descent (SGD) for a simple two-layer neural network as defined in Question 5 of Homework 0.

Specifically, for input $x \in \mathbb{R}^n$, we'll consider a two-layer neural network (without bias terms) of the form

$$z = W_2^T \text{ReLU}(W_1^T x)$$

where $W_1 \in \mathbb{R}^{n \times d}$ and $W_2 \in \mathbb{R}^{d \times k}$ represent the weights of the network (which has a d-dimensional hidden unit), and where $z \in \mathbb{R}^k$ represents the logits output by the network. We again use the softmax / cross-entropy loss, meaning that we want to solve the optimization problem, overloading the notation to describe the batch form with matrix $X \in \mathbb{R}^{m \times n}$: 即求出使得 loss 最小的 W1，W2

$$\min_{W_1, W_2} \quad \ell_{\text{softmax}}(\text{ReLU}(XW_1)W_2, y).$$

First, you will need to implement the forward and backward passes of the `relu` operator.

1. Begin by filling out the function `ReLU` operator in `python/needle/ops.py`.
2. Then fill out the `gradient` function of the class `ReLU` in `python/needle/ops.py`. **Note that in this one case it's acceptable to access the `.realize_cached_data()` call on the output tensor, since the ReLU function is not twice differentiable anyway**.

Then,

3. Fill out the `nn_epoch` method in the `apps/simple_ml.py` file.

Again, you can use your solution in Homework 0 for the `nn_epoch` function as a starting point. Note that unlike in Homework 0, the inputs `W1` and `W2` are `Tensors`. Inputs `X` and `y` however are still numpy arrays - you should iterate over mini-batches of the numpy arrays `X` and `y` as you did in Homework 0, and then cast each `X_batch` as a `Tensor`, and one hot encode `y_batch` and cast as a `Tensor`. While last time we derived the backpropagation equations for this two-layer ReLU network directly, this time we will be using our auto-differentiation engine to compute the gradients generically by calling the `.backward()` method of the `Tensor` class. For each mini-batch, after calling `.backward()`, you should compute the updated values for `W1` and `W2` in `numpy`, and then create new `Tensors` for `W1` and `W2` with these `numpy` values. Your solution should return the final `W1` and `W2` `Tensors`.

```
!python3 -m pytest -l -k "nn_epoch_ndl"
```

```
!python3 -m mugrade submit 'YOUR_GRADER_KEY_HERE' -k "nn_epoch_ndl"
```