Homework 1 Part 1

Autograd and MLPs

11-785: Introduction to Deep Learning (Fall 2020)

Out: **The Present**
Due: **The Future**

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 introduction to the library structure you will be creating, autograd, the local autograder, and the way you will submit your homework.
  - **A Simple Neural Network**: All of the problems in Part 1 will be graded on Autolab. You can download the starter code/mytorch folder structure from Autolab as well. This assignment has 100 points total.
  - **Appendix**: This contains information that you may 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.

Introduction

Starting from this assignment, you will be developing your own version of the popular deep learning library **PyTorch**. It will (cleverly) be called **"MyTorch"**.

A key part of MyTorch will be your implementation of **Autograd**, which is a library for Automatic Differentiation. This feature is new to this semester - previously, students manually programmed in symbolic derivatives. While Autograd will seem complicated at first, you'll see how much time and pain it'll save you in the future. We hope this will eventually make things easier and also enhance your understanding of DL in practice.

Your goal for this assignment is to develop all of the components necessary to train an MLP on the MNIST dataset.

Training on MNIST is considered the "Hello World!" of DL, although honestly speaking, most people don't implement Autograd to get there ☺.

Homework Structure

(Some less relevant files not shown)

```
handout
├── autograder ..... Files for scoring your code locally
│   └── hw1_autograder
│       ├── runner.py
│       ├── test_mlp.py
│       ├── test_mnist.py
│       ├── test_autograd.py
│       └── data ..... [Question 2.5] MNIST Data Files
├── mytorch ..... MyTorch library
│   ├── nn ..... Neural Net-related files
│   │   ├── activations.py
│   │   ├── functional.py
│   │   ├── linear.py
│   │   ├── loss.py
│   │   ├── module.py
│   │   └── sequential.py
│   ├── optim ..... Optimizer-related files
│   │   ├── optimizer.py
│   │   └── sgd.py
│   ├── autograd_engine.py ..... Autograd main code
│   └── tensor.py ..... Tensor object
├── create_tarball.sh ..... Script for generating Autolab submission
├── grade.sh ..... Script for running local autograder
└── hw1
    └── mnist.py ..... [Question 2.5] Running MLP on MNIST
```

Note: a prior background in Object-Oriented Programming (OOP) will be helpful, but is not required. If you're having difficulty navigating the code, please Google, post on Piazza, or come to TA hours.

Remember - you are building this library yourself. Mistakes early on can lead to a cascade of problems later, and **will** be difficult for others to debug. Make sure to read this document carefully, as it will influence future components of your work.

0.1 Autograd: An Introduction

Please make sure you understand Autograd before attempting the code. If you read from now until the start of the first question, you should be good to go.

This section of the handout will introduce Autograd and show you how it works.

0.1.1 Overview

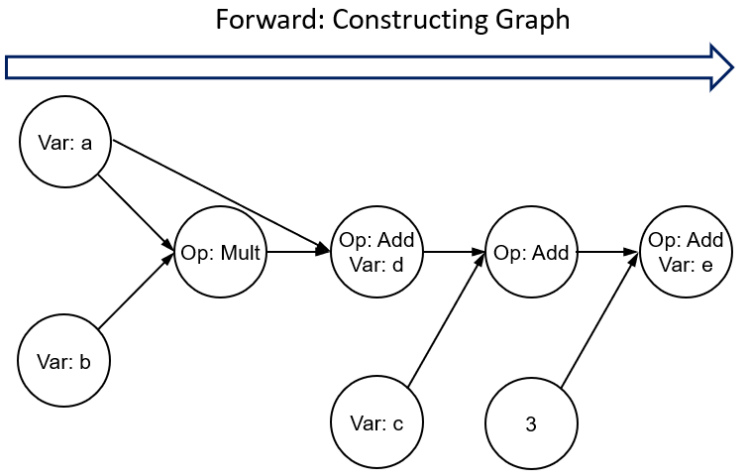
Autograd calculates the derivative of any function (not just NNs) by breaking it down into elementary components and applying the chain rule. (Remember - a single NN is just a big function!)

In modern DL, Autograd is used for training neural networks. In fact, **Backpropagation is actually a special case of automatic differentiation**. This is extremely convenient for us, as Autograd not only removes the need to write out complex derivatives, it's also faster, scalable, and numerically stable.

There are two main phases to Autograd: the forward pass and the backward pass.

0.1.2 The Forward Pass

The goal of the forward pass is to **track how variables are modified by each operation in the function**. This is done automatically while the function is being run. To do this, it constructs a **computational graph**.



The nodes of the graph are either **values**, **elementary operations**, or **both**.

Starting from the left, you can see how the input variables **a** and **b** are first multiplied together. Then, **a** is added to this product, which results in the variable **d** (middle node, which is both an operation and value).

The above graph was actually created by this "function"[1]:

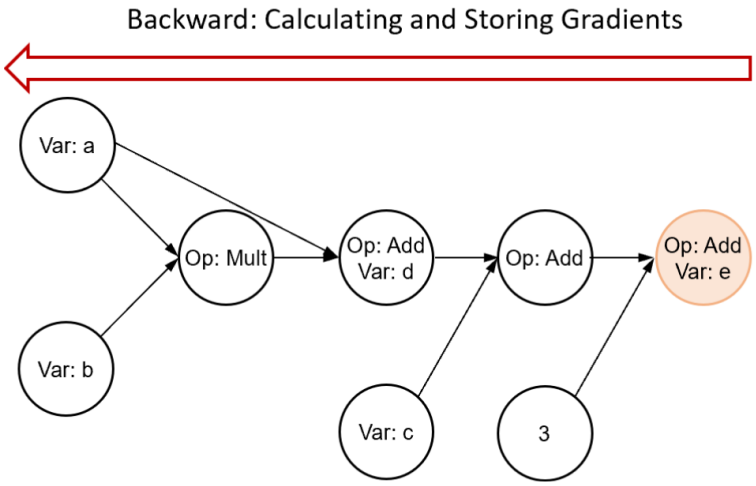
```
import torch

a = torch.tensor(1., requires_grad=True)
b = torch.tensor(2.)
c = torch.tensor(3., requires_grad=True)

d = a + a * b
e = d + c + 3
```

Notice how individual variables, operations, and variable assignments are tracked by the graph.

[1]Note, for simplicity, this example will use scalars instead of matrices. But the same principles apply.

0.1.3 The Backward Pass

After the function is run and the graph is constructed, we can now calculate the actual derivatives.



During backward, Autograd essentially retraces its steps, going in reverse from the final node in the graph (here, **e**)[2]. It does this with a recursive Depth-First Search (DFS).

At each recursive call, Autograd calculates the gradient(s): partial(s) of the final node w.r.t. the input(s) of the current node.

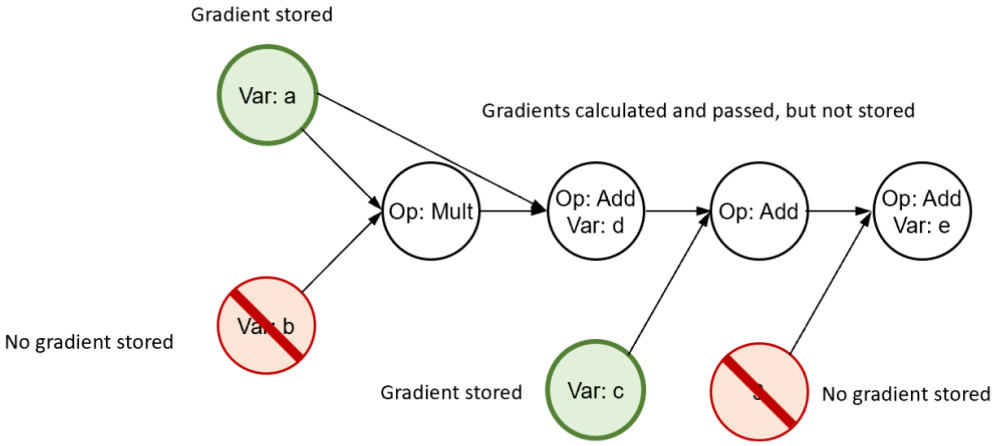
$$\frac{\partial \text{ final}}{\partial \text{ input of current}} = \frac{\partial \text{ final}}{\partial \text{ output of current}} \cdot \frac{\partial \text{ output of current}}{\partial \text{ input of current}}$$

If the current `node` is a tensor with no parents (`is_leaf=True`) and has `requires_grad=True`, the gradient will be stored in `node.grad`.

From the previous page:

```
a = torch.tensor(1., requires_grad=True)
b = torch.tensor(2.)
c = torch.tensor(3., requires_grad=True)
```

b will not have a gradient stored in `b.grad` after `backward()`. The **3** node won't either, as it's a constant and not a variable Tensor[3].



[2]In DL, the final node is usually the loss value
[3]See Appendix A for clear explanation of when/how gradients are calculated

Let's walk through an example of the backward pass.

Step 1: Starting Traversal

We begin by calling `.backward()` on the final node (tensor)[4]. Calling `e.backward()` tells Autograd to compute the approximate "full gradient"[5] of **e** with respect to each variable that affected it.

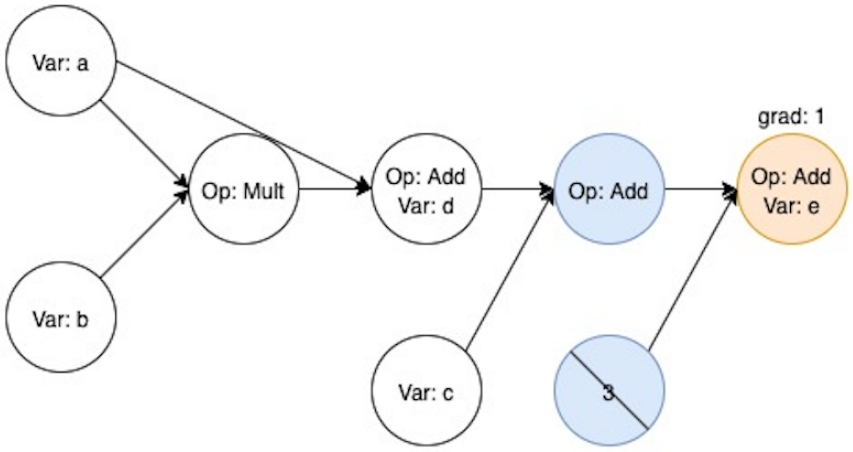
First, `e.backward()` calculates the gradient of itself w.r.t. itself:

$$\frac{\partial e}{\partial e} = [1]$$

Remember that the gradient of a tensor w.r.t. itself is just a tensor of the same shape, filled with 1's.

It then passes (without storing) its gradient to the graph traversal method: `autograd_engine.backward()`.

Step 2: First Recursive Call



We're now in `autograd_engine.backward()`, which performs a recursive DFS (see prev page for why).

Here, the output is **e**. We also know that **addition, like many other operations, always happens between two elements**, so we know there's two inputs: `Op:Add` and **3**.

Let's do the `Op:Add` node first.

$$\frac{\partial e}{\partial \text{Op:Add}} = \frac{\partial e}{\partial e} \cdot \frac{\partial e}{\partial \text{Op:Add}} = [1] \cdot [1] = [1]$$

We end up with a gradient of `[1]`, which we pass on to `Op:Add` for its own recursive call.

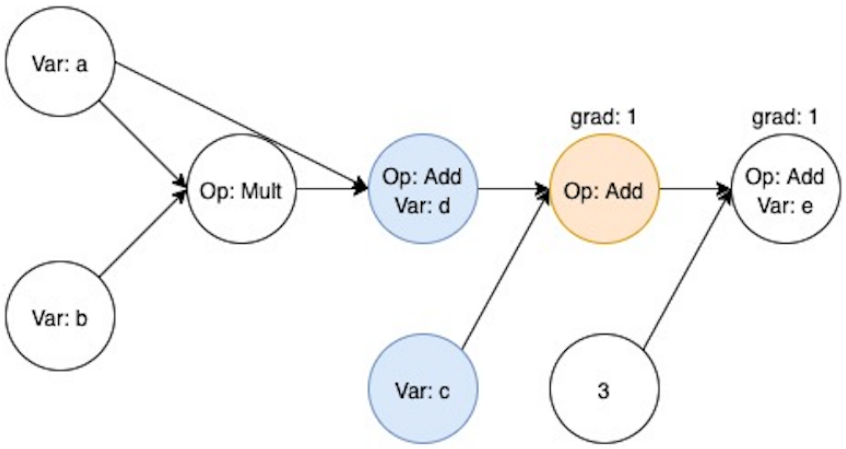
Now for the `3` node. Because it's a leaf node that doesn't store gradients, there's technically no point in calculating anything here.

In your own code, if you can find a way to check if a gradient computation is needed[6], this may speed up your training, as gradient computations are usually expensive. Otherwise you can keep things simple, and just always calculate the gradient, even if it's never stored.

[4]The real Torch only allows calling `.backward()` on scalar tensors
[5]Note: technically speaking, gradients != derivatives; see recitation 2 "Computing Derivatives"
[6]Hint: Appendix A

Also, note that the calculations in this example are simple because they're on scalars; for matrices, things get more technical, although the same principles apply.

Step 3



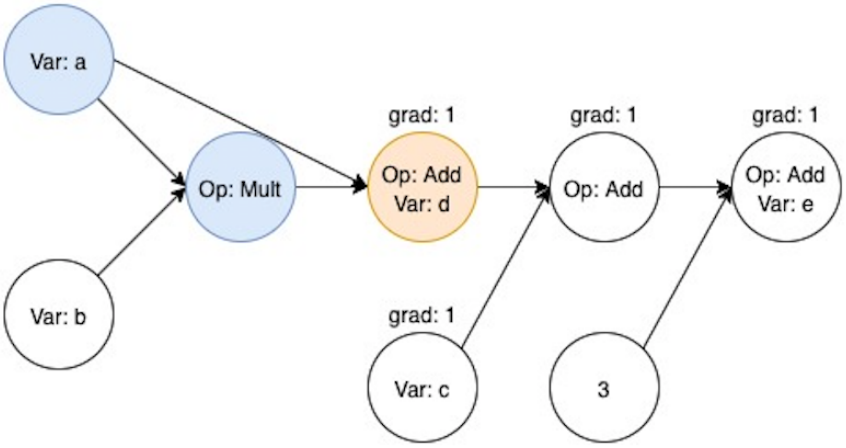
More of the same, except we compute and store the gradient of **c**.

$$\frac{\partial e}{\partial d} = \frac{\partial e}{\partial \text{ current}} \cdot \frac{\partial \text{ current}}{\partial d} = [1] \cdot [1] = [1]$$

$$\frac{\partial e}{\partial c} = \frac{\partial e}{\partial \text{ current}} \cdot \frac{\partial \text{ current}}{\partial c} = [1] \cdot [1] = [1]$$

Tip: The first factor of each gradient is always passed to us from the prev node.

Step 4

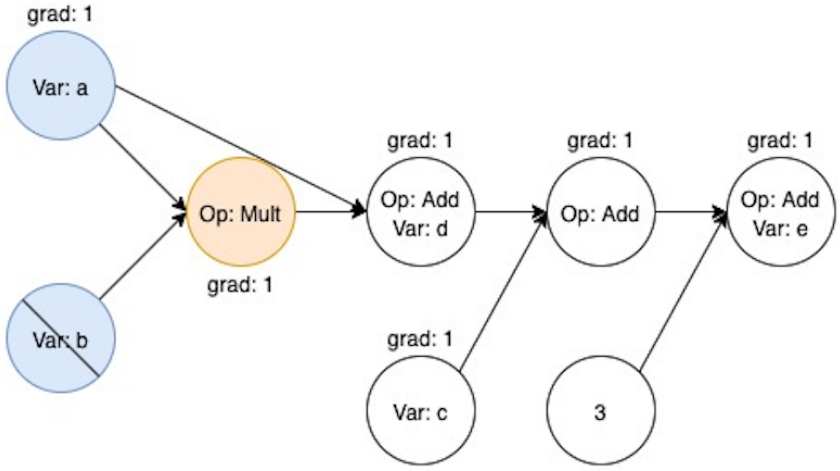


Same, but now storing the gradient of **a**.

$$\frac{\partial e}{\partial a} = [1]$$

$$\frac{\partial e}{\partial \text{ Op:Mult}} = [1]$$

Step 5: Plot Twist



This is a bit different - we have a new operation (multiplication) that behaves differently from addition.

In general, for any $z = x \cdot y$, we know that $\frac{\partial z}{\partial x} = y$ and $\frac{\partial z}{\partial y} = x$.

Here's the difference: the value of multiplication's derivative depends on its original inputs. This means that we need to store them during the forward pass, so that we can access them in the backward pass.

In the assignment, this is done with the **ContextManager** object, which is passed in as a variable **ctx** during both forward and backward.

We don't need to compute the gradient of `b` here, so for `a`:

$$\frac{\partial e}{\partial a} = \frac{\partial e}{\partial \text{ Op:Mult}} \cdot \frac{\partial \text{ Op:Mult}}{\partial a} = [1] \cdot b = [1] \cdot [2] = [2]$$

But wait, remember that we already stored a gradient for `a` in Step 4 (`a.grad=`$[1]$). What do we do with this new one?

Answer: We **"accumulate"** it with a `+=` operation[7].

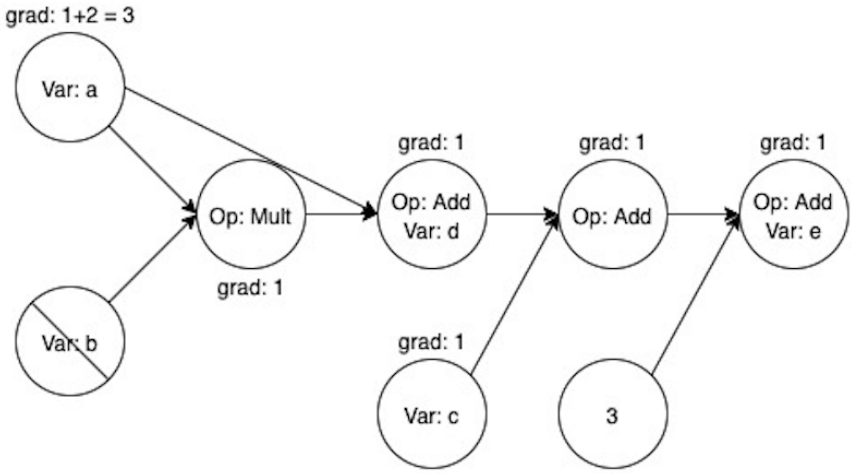
```
a.grad += [2]
>>> a.grad
[3]
```

Now **a**'s stored gradient is $= 3$.

[7] For more details, refer to the derivative recitation.

0.1.4 Concluding the Example

We're done! Here are our results:



Let's verify that this is correct using the real Torch.

Input:

```
a = torch.tensor(1., requires_grad=True)
b = torch.tensor(2.)
c = torch.tensor(3., requires_grad=True)

d = a + a * b
e = d + c + 3

e.backward()

print(a.grad)
print(c.grad)
```

Output:

```
>>> print(a.grad)
tensor(3.)
>>> print(b.grad) # No stored gradient
>>> print(c.grad)
tensor(1.)
```

Nice.

0.2 Autograd File Structure

It may be hard to keep track of all the Autograd-specific files/classes/functions, so we'll briefly describe where they are and what they do.

```
mytorch
├── nn
│   └── functional.py
├── autograd_engine.py
└── tensor.py
```

0.2.1 `functional.py`

This file contains the forward/backward behavior of operations in the computational graph. In other words, **any operation that affects the network's final loss value should have an implementation here**[8]. This also includes loss functions.

```
functional.py
├── class Add ..... Node in graph; addition between two tensors
├── class Sub ..... Same as above; subtraction
├── class Mult ..... Element-wise tensor product. Note: "Mult" not "Mul"
├── class Div ..... Element-wise tensor division
└── function cross_entropy() ..... Cross-entropy loss (outputs tensor you'd do .backward() on)
```

Note: each elementary operation (node in graph) is a subclass of `autograd_engine.Function`[9].

0.2.2 `autograd_engine.py`

```
autograd_engine.py
├── function backward() ..... The recursive DFS
├── class Function ..... Base class for functions in functional.py
├── class AccumulateGrad ..... Used in backward(). Represents nodes that accumulate gradients
├── class ContextManager ..... Arg "ctx" in functions. Passes data between forward/backward
└── class BackwardFunction ..... Used in backward(). Represents intermediate nodes
```

In this file you'll only need to implement `backward()` and `Function.apply()`. But you'll want to read the other classes' code, as you'll be extensively working with them.

0.2.3 `tensor.py`

```
tensor.py
└── class Tensor ..... Wrapper for np.array, allows interaction with MyTorch
```

Contains only the class representing a Tensor. Most operations on Tensors will likely need to be defined as a class method here[10].

You'll probably want to **override operators**; here's a good explanation on what this means.

Also, pay close attention to the class variables defined in `__init__()`. Appendix A covers these in detail; especially before starting problem 1.2, we highly encourage you to read it.

[8] See the actual Torch's `nn.functional` for ideas on what operations belong here.
[9] In Python, subclasses indicate their parent in their declaration. i.e. `class Add(Function)`
[10] Again, see the actual `torch.Tensor` for ideas on what to implement

0.3 Running/Submitting Code

This section covers how to test code locally and how to create the final submission.

Note that there are two different local autograders. We'll explain them both here.

0.3.1 Running Local Autograder (Before MNIST)

Run the command below to calculate scores for Problems 1.1 to 2.4 (all problems before Problem 2.5: MNIST)

```
.grade.sh 1
```

(Make sure the command includes the "`1`")

If the script fails for some reason, you can run the autograder manually with this:

```
python3 ./autograder/hw1_autograder/runner.py
```

Note: as MNIST is not Autograded, 90/100 is a "full" score for this section.

0.3.2 Running Local Autograder (MNIST only)

To test Problem 2.5 locally and generate plots for submission, run the following:

```
.grade.sh m
```

You can also run it manually with this:

```
python3 ./autograder/hw1_autograder/test_mnist.py
```

0.3.3 Submitting to Autolab

Note: You can submit to Autolab even if you're not finished yet. You should do this early and often, as it guarantees you a minimum grade and helps avoid last-minute problems with Autolab.

Run this script to gather the needed files into a `handin.tar` file:

```
./create_tarball.sh
```

You can now upload `handin.tar` to Autolab.

Note that, for the final submission, you'll need plots that were generated by the MNIST autograder.

---

1 Problem 1: Implementing Autograd [Total: 40 points]

It's finally time to implement Autograd. We'll start by implementing some elementary operations.

Advice:

1. The names of classes and class variables matter. The **Autograder** will look for specific ones when determining your score. Don't rename existing classes/variables, read error messages closely, and consider checking the autograder code itself to see what it's looking for.
2. Try to keep your code concise and object-oriented.
3. Use existing NumPy functions, especially if they replace loops with vector operations.
4. Debuggers like pdb are usually simpler and more reliable than `print` statements[11]. Also, see these recitations on debugging: Recitations 0F and 0G
5. **For how to run the Autograder and generate submissions, read Section 0.3.**

1.1 Basic Operations [15 points]

- In `nn/functional.py`, implement the `forward` and `backward` class methods for **element-wise subtraction, multiplication, and division**. The forward function simply performs the operation and returns the output. The backwards function takes as input the derivative of the loss with respect to the output, and returns the derivatives of the loss with respect to the inputs.
- The addition operation has been implemented for you as an example.

1.1.1 Subtract [5 points]

In `nn/functional.py`, complete the `Sub` class.

1.1.2 Multiply [5 points]

In `nn/functional.py`, complete the `Mult` class.

1.1.3 Divide [5 points]

In `nn/functional.py`, complete the `Div` class.

1.2 Autograd [25 points]

Now to implement the core Autograd engine. Autograd test1 will be the example we went through. Test2 will be the same, but more stringent in terms of what gradients you store/accumulate. If you haven't yet, we encourage you to read Appendix A.

- In `tensor.py`, implement the `Tensor.backward()` function. This should be short - see step 1 of the example for what this is doing.
- In `autograd_engine.py`, implement the `backward(grad_fn, grad_of_outputs)` function. This is the DFS backward pass. Think about what objects are being passed, and the base case(s) of the recursion. This code should also be short.
- Also in `autograd_engine.py`, finish the `.apply()` function of the `Function` class. Remember that this is the super-class that functions in `nn/functional.py` are inheriting from. We've provided some starter code that creates the `BackwardFunction` class. You'll need to populate the `backward_function.next_functions` list with the parent nodes of this node.

We will test your autograd implementation using the four basic operations from the previous section.

[11] For an easy breakpoint, this oneliner is great: `import pdb; pdb.set_trace()`

2 Problem 2: MLPs [Total: 60 points]

We're now nearing our main objective of this homework: training an MLP on the MNIST dataset. But first, we need to implement an actual MLP.

To do this, you'll need to code a Linear Layer, an activation function, and an optimizer. Only after completing these tasks three will you be able to challenge the formidable MNIST.

2.1 Linear and ReLU [15 points]

In `nn/linear.py`, complete the `Linear` class. This represents a linear layer in an MLP. Doing so will require additional work in `nn/functional.py`[12]. It will also require you to implement the forward method in `nn/sequential.py`. This should give you 5 points.

$$\text{Linear(x)} = xW^T + b$$

(Often notated $Wx + b$; this formulation works cleaner in Torch).

Then, complete the ReLU object in `nn/activations.py`. This will require completing the forward/backward of the ReLU function in `nn/functional.py`. This should give you the remaining 10 points.

$$\text{ReLU(z)} = \begin{cases} z & z > 0 \\ 0 & z \leq 0 \end{cases}$$

$$\text{ReLU}'\text{(z)} = \begin{cases} 1 & z > 0 \\ 0 & z \leq 0 \end{cases}$$

2.2 Stochastic Gradient Descent (SGD) [10 points]

In `optim/sgd.py`, complete the `SGD.step` function.

This function is called after the gradients are generated. Optimizers use gradients to determine how to update network params in order to minimize loss.

Note that this class inherits from `optim.optimizer.Optimizer`. Also, you won't need to change the init method until you start momentum. Finally, make sure this code does not add to the computational graph.

$$W^k = W^{k-1} - \eta \nabla_W Loss(W^{k-1})$$

2.3 Cross Entropy Loss [15 points]

In `nn/functional.py`, complete the `CrossEntropyLoss` class. This will require calling and completing the `nn.functional.cross_entropy()` function.

Note that `nn.functional.cross_entropy()` does not need to extend `Function`, nor does it need to have a backward and forward function (although you may do that). It can be a standalone function that uses other functions that implement backward and forward.

For more info on Cross Entropy Loss, see Appendix B.

[12] being intentionally vague here ☺

2.4 Momentum [10 points]

Now go back to `optim/sgd.py` and add momentum. We will be using the following momentum update equation:

$$\nabla W^k = \beta \nabla W^{k-1} - \eta \nabla_W Loss(W^{k-1})$$
$$W^k = W^{k-1} + \nabla W^k$$

as discussed in lecture. One momentum value will have to be stored in the SGD for each parameter, so the init function will change. Make sure this code does not add to the computational graph.

2.5 MNIST [10 points]

Finally, you will test your code on the MNIST dataset.

In `hw1/mnist.py`, in the `mnist` function create an instance a criterion (`CrossEntropyLoss`) and an optimizer (`SGD`). Also create your `Sequential` model here, using `Linear` and `ReLU` layers.

Next, implement `train` and `validate`. Train should return a list containing the validation accuracy after every epoch.

Running the MNIST testing script will generate and save the file `'validation_accuracy.png'` in your handout folder.

Congratulations, you're done! Implementing this is no easy task; great work. More to come ☺.

For generating the final submission, see Section 0.3.3 on this handout.

Appendix

A Tensor Attributes

These are the class variables defined in `Tensor.__init__()`:

```
class Tensor
├── data (np.array)
├── requires_grad (boolean)
├── is_leaf (boolean)
├── grad_fn (autograd_engine.BackwardFunction)
├── grad (Tensor)
└── is_parameter (boolean)
```

A.1 `requires_grad` and `is_leaf`

These variables are for determining whether to calculate/store gradients during the Autograd backward phase.

is_leaf (default: `True`) indicates whether this tensor is a "**Leaf Tensor**". A Leaf Tensor is a tensor node on the computational graph that has no parents. **Any node that has `requires_grad=False` is a Leaf Tensor**[13].

requires_grad[14] (default: `False`) indicates whether gradients need to be calculated for this tensor.

The combination of these variables determines the tensor's role in the computational graph:

1. **AccumulateGrad** node. Has `is_leaf=True` and `requires_grad=True`. User-created input vector, need to calculate and store gradients in the `.grad` variable.
2. **BackwardFunction** node. Has `is_leaf=False` and `requires_grad=True`. This tensor is the output of an intermediate node. Gradients are calculated/passed, but not stored. When a function's `.apply()` method is called, a `autograd_engine.BackwardFunction` object is created and stored in the output tensor's `grad_fn` variable.
3. **Constant** node (Use `None`; no object for this). Has `is_leaf=True` and `requires_grad=False`. This means this Tensor is a user-created tensor with no parents, but does not require gradient storage.

For an intermediate node, if any of its parents `requires_grad`, then this node will also `require_grad`.

For example, consider a tensor `C` created by `C = A + B`. If both `A` and `B` had `requires_grad=False`, `C` would also have `requires_grad=False` and `C.is_leaf = True`.

If `A` or `B` had `requires_grad=True`, we would have `C.requires_grad=True` and `C.is_leaf = False`. PyTorch does not store gradients for intermediary tensors.

A.2 `grad_fn` and `grad`

Set during `autograd_engine.backward()`. See above.

A.3 `is_parameter`

Indicates whether this tensor represents parameters of a network. For example, `nn.linear.Linear.weight` is a tensor where `is_parameter` should be `True`. In addition, if `is_parameter` is `True`, `requires_grad` and `is_leaf` should also both be `True`.

[13] It's impossible for a tensor to have both `requires_grad=False` and `is_leaf=False`, hence 3 possible node types.
[14] Official description of requires_grad here

B Cross Entropy Loss ("XE Loss")

For quick info, the official Torch doc is very good. But we'll go in depth here.

Let's begin with a broad, coder-friendly definition of XE Loss.

If you're trying to predict what class an input belongs to, XE Loss is a great loss function[15]. For a single training example, XE Loss essentially measures the "incorrectness" (**"divergence"**) of your confidence in the true label. The higher your loss, the more incorrect your confidence was.

To use XE Loss, the output of your network should be a float tensor of size (N, C), where N is batch size and C is the number of possible classes. We call this output a tensor of **"logits"**. The logits represent your unnormalized confidence that an observation belongs to each label. The label with the "highest confidence" (largest value in row) is usually your "prediction" for that observation[16].

We'll also need another tensor containing the **true labels** for each observation in the batch. This is a `long`[17] tensor of size $(N,)$.

There are essentially two steps to calculate XE Loss, which we'll cover below.

NOTE: Directly implementing the below formulas with loops will be too slow. Convert to matrix operations and/or use NumPy functions.

Step 1: `LogSoftmax()`

First, it applies a `LogSoftmax()` to the logits. For a **single observation**:

$$\text{LogSoftmax}(x_n) = \log \frac{e^{x_n}}{\sum_{c=0}^{C-1} e^{x_c}}$$

(Note: don't directly implement the above. Read section B.1 for why)

Softmax scales the values in a 1D vector into "probabilities" that sum up to 1. We then normalize by applying the `log`. The resulting vector p_n contains floats that are ≤ 0.

Step 2: `NLLLoss()`

Second, it calculates the Negative Log-Likelihood Loss (`NLLLoss()`) using the tensor from the previous step (P) and the true label tensor (L).

$$\text{NLLLoss}(P, L) = -\frac{\sum_{n=0}^{N-1} P_{n,l_n}}{N}$$

Essentially, you're getting the average (across the batch) of the values at the correct indices. The resulting scalar will be ≤ 0, so we take its negative.

This is your final loss. Note that it's averaged across the batch (this is generally a good thing; why?).

[15] As you'll see, it's popular and commonly used in a variety of different areas.
[16] During val/test, the idx of the **maximum value** is usually returned as your label.
[17] The official Torch uses long; for us, it should be ok to use int

B.1 Stabilizing LogSoftmax() with the LogSumExp Trick

The LogSumExp trick is used to prevent numerical underflow and overflow which can occur when the exponent is very large or very small. For example:

```
>>> import math
>>> math.e**1000
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    math.e**1000
OverflowError: (34, 'Result too large')
>>> math.e**(-1000)
0.0
```

As you can see, for exponents that are too large, Python throws an overflow error, and for exponents that are too small, it rounds down to zero.

We can avoid these errors by using the LogSumExp trick:

$$\log \sum_{n=0}^{N} e^{x_n} = a + \log \sum_{n=0}^{N} e^{x_n - a}$$

You can read proofs of its equivalence here and here

B.2 The Derivative of XE Loss

Cross-entropy comes from information theory, where it is defined as the expected information quantified as $\log \frac{1}{q}$ of some subjective distribution Q over an objective distribution P. It quantifies how much information we (our model Q) receive when we observe true outcomes from Q – it tells us how far our model is from the true distribution P. We minimize Cross-Entropy when $P = Q$. The value of cross-entropy in this case is known simply as the entropy.

$$H = \sum_{x \in X} P(x) \log \frac{1}{P(x)} = - \sum_{x \in X} P(x) \log P(x)$$

This is the irreducible information we receive when we observe the outcome of a random process. Consider a coin toss. Even if we know the Bernoulli parameter p (the probability of heads) ahead of time, we will never have absolute certainty about the outcome until we actually observe the toss. The greater p is, the more certain we are about the outcome and the less information we expect to receive upon observation.
We can normalize our network output using softmax and then use Cross-Entropy as an objective function. Our softmax outputs represent Q, our subjective distribution. We will denote each softmax output as $\hat{y}_j$ and represent the true distribution P with output labels $y_j = 1$ when the label is for each output. We let $y_j = 1$ when the label is j and $y_j = 0$ otherwise. The result is a degenerate distribution that will aim to estimate P when averaged over the training set.
Note that when we take the partial derivative of CrossEntropyLoss, we get the following result:

$$\frac{\partial L(\hat{y}, y)}{\partial x_j} = \hat{y}_j - y_j$$

This is a very simple and elegant expression for the derivative of softmax with cross-entropy divergence with respect to its input. What this is telling us is that when $y_j = 1$, the gradient is negative and thus the opposite direction of the gradient is positive: it is telling us to increase the probability mass of that specific output through the softmax.