Homework 1 Part 1

Autograd and MLPs

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

Out: **Sept 6th 2020 12:00:00am EST**
Due: **Sept 27th 2020 11:59:59pm EST**

(Last updated: 9/13/20 11:41PM 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:**
  - **MyTorch**: An explanation of the library structure, Autograd, the local autograder, and how to submit to Autolab. You can get the starter code from Autolab or the course website.
  - **Autograd**: Part 1 one will cumulate in a functioning Autograd library, which will be autograded on Autolab. This is worth 40 points.
  - **MLPs**: Part 2 will be to use and expand your Autograd to complete an MLP library with Linear, ReLU, and BatchNorm layers. It will also have cross entropy loss and an SGD optimizer with momentum. All of these will be autograded. This is worth 60 points.
  - **MNIST**: Part 3 will be to use your MLP library to train on the MNIST dataset. This is worth 10 points.
  - **Appendix**: This contains information that you may (read: will) find useful.

- **Directions:**
  - We estimate this assignment may take up to 15 hours, with much of that time digesting concepts. So **start early**, and **come to office hours if you get stuck** to ensure things go as smoothly as possible.
  - 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 questions often rely on the completion of previous questions.
  - You are required to do this assignment using Python3. Other than for testing, 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.
  - Note that Autolab uses numpy v1.18.1

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 for each module. While Autograd may seem hard at first, the code is short and it will save a lot of future time and effort. We're hoping that you'll see its value in reducing your workload and also enhancing your understanding of DL in practice.

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

Training on MNIST is considered to the `print("Hello world!")` of DL. But this metaphor usually doesn't assume you've implemented autograd from scratch :) ☺.

Homework Structure

```
handout
├── autograder ............................................. Files for scoring your code locally
│   └── hw1_autograder
│       ├── runner.py ...................................... Files for running autograder tests
│       ├── test_mlp.py
│       ├── test_mnist.py
│       ├── test_autograd.py
│       └── data ........................................... [Question 3] MNIST Data Files
├── mytorch ................................................ MyTorch library
│   ├── nn ................................................. Neural Net-related files
│   │   ├── activations.py
│   │   ├── batchnorm.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
├── sandbox.py ............... Simple environment to test operations and autograd functions
├── create_tarball.sh ...................................... Script for generating Autolab submission
├── grade.sh ............................................... Script for running local autograder
└── hw1
    └── mnist.py ........................................... [Question 3] 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.

Also, see **Recitation 2 (Calculating Derivatives)**, which gives you more background and additional support in completing the homework. We'll release the slides early, with HW1.

0.1.1 Why autograd?

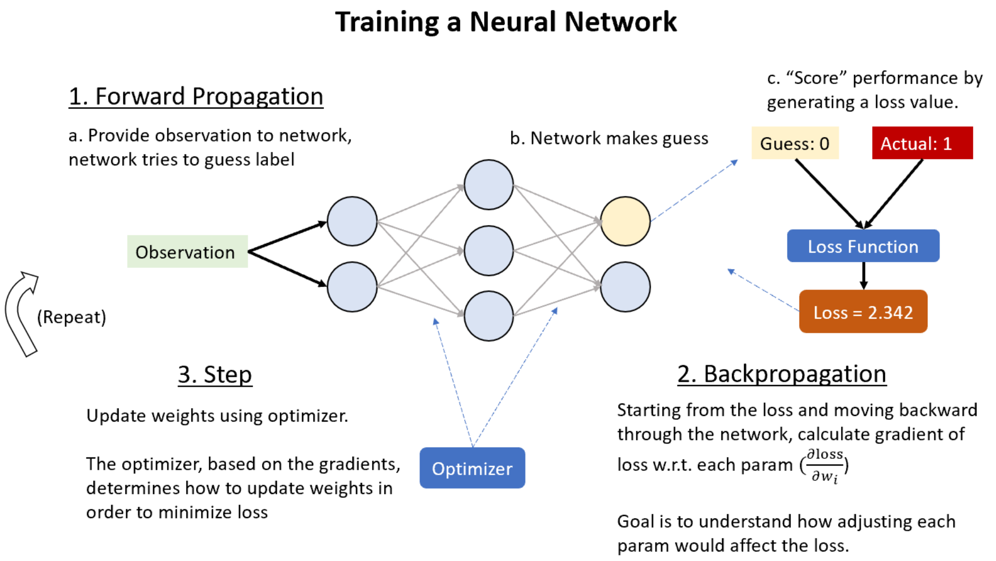
Autograd is a framework for automatic differentiation. It's used to automatically calculate the derivative (or for us, gradient) of **any** computable function[1]. This includes NNs, which are just big, big functions.

It turns out that **backpropagation is actually a special case of automatic differentiation**. They both do the same thing: calculate partial derivatives.

This equivalence is very convenient for us. Without autograd, you'd have to manually solve, implement, and re-implement derivative calculations for every individual network component (last year's homework). This also means that changing any part/setting of that component would require adding more special cases or possibly even re-implementing big chunks of the code.

But with autograd, you only need to implement derivatives for simple operations and functions, which by homework 2, you'll see will already be good enough to run most common DL components. The rest of the code will be typical **Object-Oriented Programing (OOP)**. Compared to last year's homework, this is much more flexible, easier to debug, and (eventually) less confusing.

0.1.2 Context: Training an NN


Training a Neural Network
1. Forward Propagation
a. Provide observation to network, network tries to guess label
b. Network makes guess
c. "Score" performance by generating a loss value.
Guess: 0
Actual: 1
Observation
Loss Function
Loss = 2.342
(Repeat)
3. Step
Update weights using optimizer.
The optimizer, based on the gradients, determines how to update weights in order to minimize loss
Optimizer
2. Backpropagation
Starting from the loss and moving backward through the network, calculate gradient of loss w.r.t. each param ($\frac{\partial \text{loss}}{\partial w_i}$)
Goal is to understand how adjusting each param would affect the loss.

Above is a typical training routine for an NN. Review it carefully, as it'll be important context for both explaining autograd and completing the assignment.

[1] See recitation 2 for explanation of the difference between derivatives/gradients

0.1.3 Context: Loss, Weights, and Derivatives

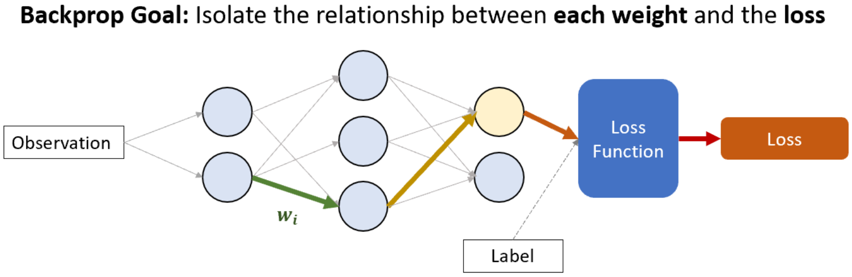
Important question: what are backprop and autograd doing? Why are they calculating gradients, and what are the gradients used for?

Recall these two calculus concepts:

1. A partial derivative $\frac{\partial y}{\partial x_i}$ measures how much the output y would change if we only change the input variable x_i. We assume the other variables are held constant.

2. The derivative can be interpreted as the slope of a function at some input point. If the slope is positive and you increase x_i, you should expect to move 'upward' (y should increase). Similarly, if the slope is negative, you should expect to move 'downward' (y should decrease).

In the forward pass, the final loss value depends on the input values, the network params, and the label value. But we're not interested in the input/label; what we're interested in is **how exactly each individual network param affected the loss**.


Backprop Goal: Isolate the relationship between **each weight** and the **loss**
Observation
w_i
Loss Function
Loss
Label

$$\frac{\partial \text{Loss}}{\partial w_i} = \frac{\partial \text{Loss}}{\partial \text{LossFunc}} \cdot \frac{\partial \text{LossFunc}}{\partial \text{Guess}} \cdot \frac{\partial \text{Guess}}{\partial w_j} \cdot \frac{\partial w_j}{\partial w_i}$$

The partial derivatives (or gradients; gradients are just the derivative transposed) that we are calculating in backprop track these relationships: **assuming the other params are held constant, how much would an increase in this param change the loss?** If we increase the param's value and the loss increases, that's probably bad (because we're trying to minimize loss). But if the loss decreases, that's probably good.

In short, the goal of backprop is to measure how much each param affects the loss, which is encoded in the gradients. We then use the gradients in the "Step" phase to actually adjust those params and minimize our *estimated* loss[2].

Note: This is where 'gradient descent' gets its name: we're trying to move around in the params' *gradient* space to *descend* to the lowest possible estimated loss.

Try to keep this goal in mind for the rest of the assignment: it should help contextualize things.

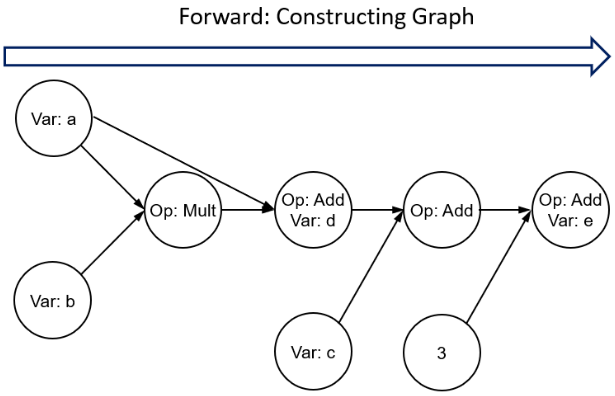
Autograd accomplishes this exact same thing, but it adds code to forward propagation in order to automate backprop. "Step" will be mostly unchanged, however, as it takes place after backprop/autograd anyway.

With this in mind, let's walk through what autograd is doing during Forward Propagation (autograd's **"forward pass"**) and Backpropagation (autograd's **"backward pass"**).

[2] Why *estimated* loss? Try to consider what a "true" loss for a real world problem might look like.

0.1.4 Forward Pass

During forward propagation, autograd automatically constructs a computational graph. It does this in the background, while the function is being run.



The computational graph tracks how elementary operations modify data throughout the function. Starting from the left, you can see how the input variables **a** and **b** are first multiplied together, resulting in a temporary output **Op:Mult**. Then, **a** is added to this temporary output, creating **d** (the middle node).

Nodes are added to the graph whenever an operation occurs. In practice, we do this by calling the `.apply()` method of the operation (which is a subclass of `autograd_engine.Function`). Calling `.apply()` on the subclass implicitly calls `Function.apply()`, which does the following:

```
1. Create a node object for the operation's output
2. Run the operation on the input(s) to get an output tensor
3. Store information on the node, which links it to the comp graph
4. Store the node on the output tensor
5. Return the output tensor
```

It sounds complicated, but remember that all of this is happening in the background; all the user sees is that an operation occurred on some tensors and they got the correct output tensor.

But what information is stored on the node, and how does it link the node to the comp graph?

The node stores two things: **the operation that created the node's data**[3] and **a record of the node's "parents"** (if it has any... ☹).

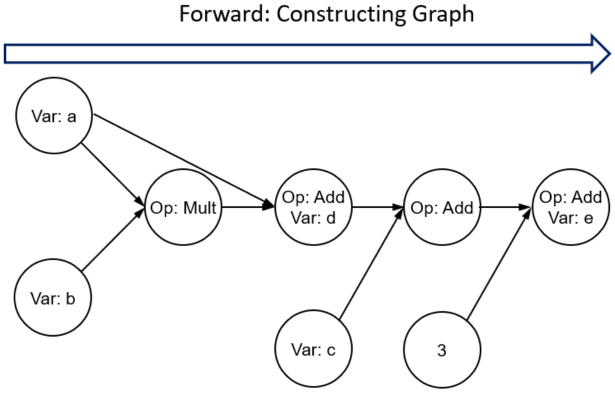
Recall that each tensor stores its own node. So when making the record of the current node's parents, we can usually just grab the nodes from the input tensors. But also recall that we *only create nodes when an operation is called.* `Op:Mult` was the very first operation, so its input tensors `a` and `b` didn't even have nodes initialized for them yet.

To solve this issue, `Function.apply()` also checks if any parents need to have their nodes created for them. If so, it creates the *appropriate type of node* for that parent (we'll introduce node types during backward), and then adds that node to its list of parents. Effectively, this connects the current node to its parent nodes in the graph.

To recap, whenever we have an operation on a tensor in the comp graph, we create a node, get the output of the operation, link the node to the graph, and store the node on the output tensor.

[3] In the code, we've already stored this for you.

Here's the same graph again, for reference:



Below is the code that created this graph:

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

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

And here's that code translated into symbolic math, just to make clear the inputs and outputs:

$$f(x, y, z) = ((x \cdot y + x) + z) + 3 \tag{1}$$

$$\text{Let } e = f(a, b, c) \tag{2}$$

We strongly recommend that you pause and try to recreate the above graph on pen and paper, just by looking at the code and the description on the previous page. Track what information each tensor and each node is storing. Ignore node types for now. It's ok to do this slowly, while asking clarifying questions on Piazza. This is confusing until you get it, and starting the code too early risks creating a debugging safari.

But why are we making this graph?

Remember: we're doing all of this to calculate gradients. Specifically, **the partial gradients of the loss w.r.t. each gradient-enabled tensor (any tensor with `requires_grad==True`, see above code)**. For us, our gradient-enabled tensors are `a` and `c`.

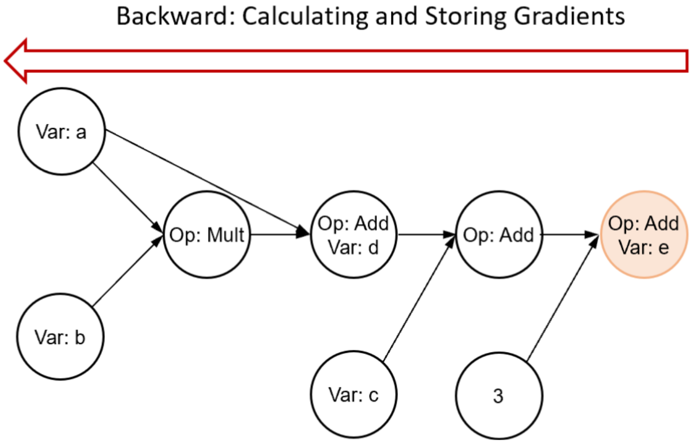
$$\text{Goal: } \frac{\partial e}{\partial a} \text{ and } \frac{\partial e}{\partial c}$$

Think of the graph as a trail of breadcrumbs that keeps track of where we've been. It's used to retrace our steps back through the graph during backprop.

By the way, this is where ”backpropagation” gets its name: both autograd/backprop traverse graphs *backwards* while calculating gradients.

Why backwards? Find out on the next page...

0.1.5 Backward Pass



In the backward pass, starting at the final node (`e`), autograd traverses the graph in reverse by performing a recursive Depth-First Search (DFS) .

Why a DFS? Because it turns out that every computable function can be decomposed into a Directed Acyclic Graph (DAG) . A reverse-order DFS on a DAG guarantees at least one valid path for traversing the entire graph in linear time[4].

Doing this in reverse is just much more efficient than doing it forwards. Pause and try to imagine why this is the case. For reference, the answer is here .

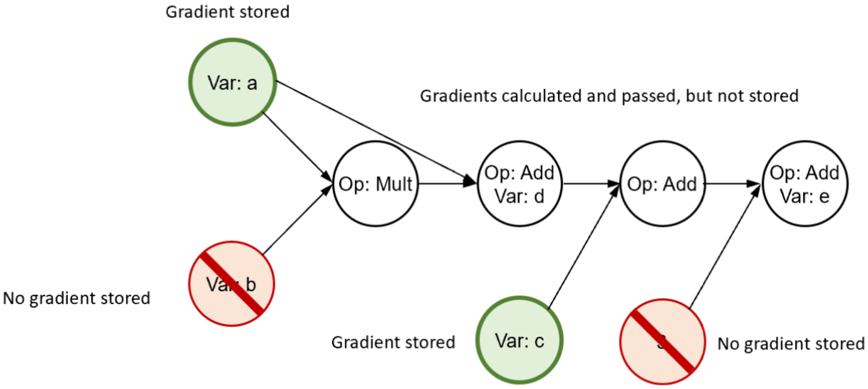
At each recursive call, Autograd calculates one gradient for **each** input.

$$\frac{\partial \text{ final}}{\partial \text{ input}_i} = \frac{\partial \text{ final}}{\partial \text{ output}} \frac{\partial \text{ output}}{\partial \text{ input}_i}$$

Each gradient is then passed onto its respective parent, but only if that parent is "gradient-enabled" (`requires_grad==True`). If the parent isn't, the parent does not get its own gradient/recursive call. Note: constants like the `3` node are *not* gradient-enabled by default.

Eventually, there will be a gradient-enabled node that has no more gradient-enabled parents. For nodes like these, all we do is store the gradient they just received in their tensor's `.grad`.

The rules may sound complicated, but the results are easy to understand visually:



Make sure you understand why every node in this graph does/does not have a stored gradient.

[4] This is because a reverse-order DFS on a DAG is essentially a Topological Sort

Let's walk through the backward pass step-by-step. Calculations will be simpler than usual here, as they're on scalars instead of matrices, but the main ideas are the same.

Step 1: Starting Traversal

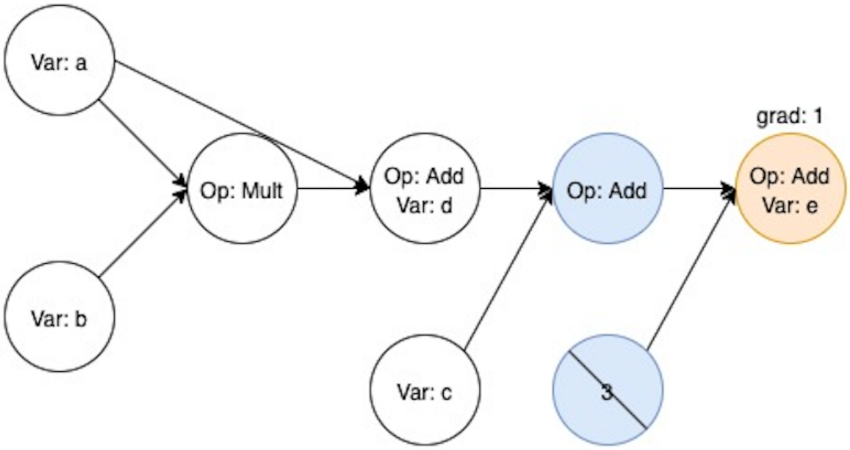
We begin this process by calling `e.backward()`. This method simply calculates the gradient of `e` w.r.t. itself:

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

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 the actual recursive DFS.

At each recursive call, we're given two objects.

The first object is a tensor containing the gradient of the final node w.r.t. our output (here, $\frac{\partial \mathrm{e}}{\partial \mathrm{e}}$). We'll use this to calculate our own gradient(s).

The second object "`grad_fn`" contains the current node object. The node object has a method `.apply()` that contains instructions on what to do with the given gradient.

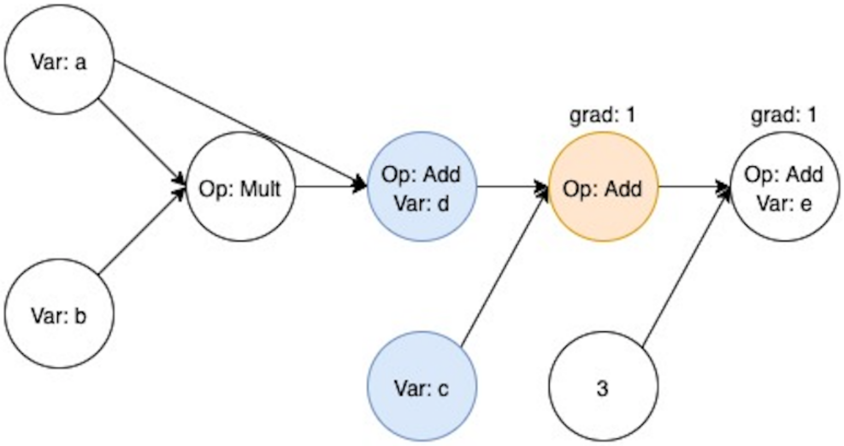
In this case, `grad_fn` is a `BackwardFunction` object. So calling `grad_fn.apply(grad_of_outputs)` actually calls the operation's gradient calculation method (`Add.backward()`). This calculates the gradients w.r.t. each input:

$$\frac{\partial e}{\partial \text{Op:Add}} = \frac{\partial e}{\partial e}\frac{\partial e}{\partial \text{Op:Add}} = [1]\,[1] = [1]$$
$$\frac{\partial e}{\partial [3]} = \frac{\partial e}{\partial e}\frac{\partial e}{\partial [3]} = [1]\,[1] = [1]$$

Notice how the given gradient (here, $\frac{\partial e}{\partial e}$) was used in both calculations.

We now pass the gradients onto their respective parents for their own recursive calls, but only if they have `requires_grad==True`. This means we only pass `Op:Add` its gradient, as `[3]` is a constant. No more work needs to be done for `[3]`, as it has no parents and no need to store a gradient.

Step 3



We're now at `Op:Add`'s recursive call. We're given $\frac{\partial \mathrm{e}}{\partial \mathrm{Op:Add}}$ and another `grad_fn` that's a `BackwardFunction` object again.

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

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

Same as before, we pass these gradients to our parents (`c` and `d`). But let's explain **c**'s recursive call first, as it's a bit different[5].

Step 4: Storing a gradient

This call is a bit different, as this node's tensor `c` is gradient-enabled (`requires_grad==True`) and has no parents (`is_leaf==True`).

As discussed before, this means that we have to store the gradient we just received. In NNs, tensors like `c` are usually network params.

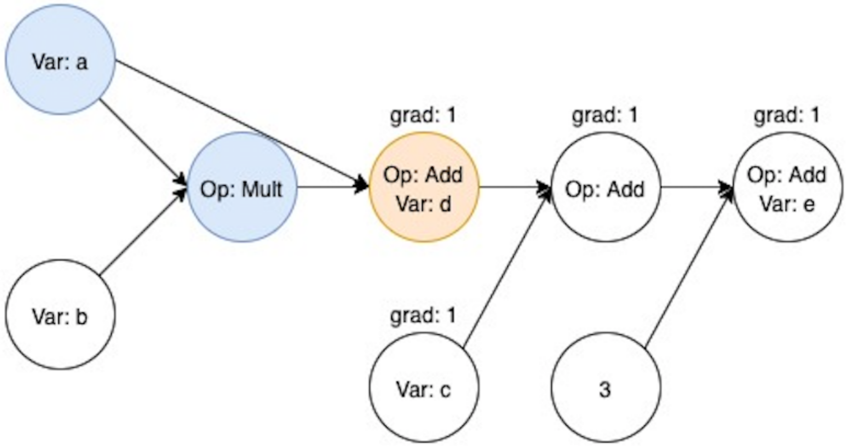
We can actually store gradients by calling the same `grad_fn.apply()` function, because this time `grad_fn` is an `AccumulateGrad` object. `AccumulateGrad.apply()` handles storing gradients.

```
>> grad_fn.apply(grad_of_outputs)
>> c.grad
tensor(1.)
```

Again, we've re-used the command `grad_fn.apply()`, but its functionality changed depending on what `grad_fn` was.

[5] In practice, it won't matter what order you call the parents in

Step 5



Back to `d`'s call, you know the drill.

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

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

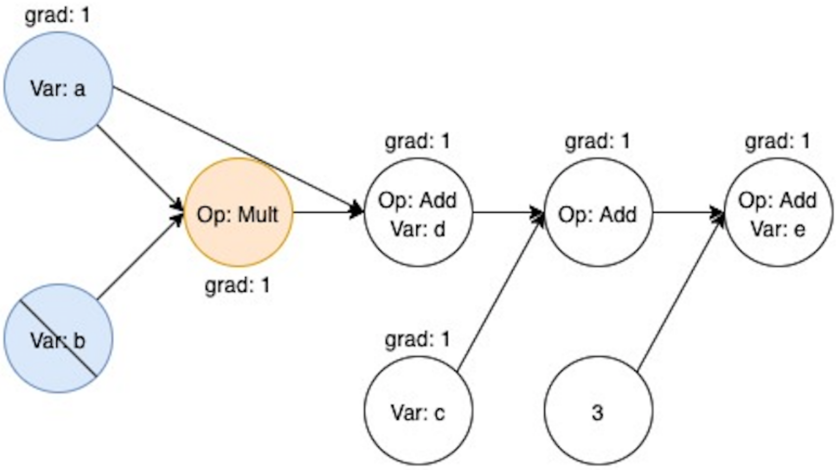
Step 6

We've stepped into the `a` node to store its gradient.

```
>> grad_fn.apply(gradient_e_wrt_a)
>> a.grad
tensor(1.)
```

Now to step out and step into the `Op:Mult` node.

Step 7: Plot Twist!



`Op:Mult` is a bit different - thus far, our derivatives have only been over addition nodes. Now we have a new operation (multiplication) that has a different kind of derivative.

In general, for any $z = xy$, we know that $\frac{\partial z}{\partial x} = y$ and $\frac{\partial z}{\partial y} = x$. Here's the difference: notice that **the derivative of a product depends on the specific values of its inputs**. For $\frac{\partial z}{\partial x}$, we need to know y, and for $\frac{\partial z}{\partial y}$, we need to know x. Similarly, for `Op:Mult`'s gradients, we'll need to know `a` and `b`.

The problem is that we saw the inputs during forward, but now we're in backward. To fix this, we store any information we need in an object that gets passed to both the forward and backward methods. In short, **whenever we need to pass information between forward and backward, store it in the `ContextManager` object ("`ctx`") during forward, and retrieve the info during backward.**

$$\frac{\partial e}{\partial b} = \frac{\partial e}{\partial \text{ Op:Mul}} \frac{\partial \text{ Op:Mul}}{\partial b} = \left[1\right] a = \left[1\right] \left[1\right] = \left[1\right]$$
$$\frac{\partial e}{\partial a} = \frac{\partial e}{\partial \text{ Op:Mul}} \frac{\partial \text{ Op:Mul}}{\partial a} = \left[1\right] b = \left[1\right] \left[2\right] = \left[2\right]$$

Remember, `b` has `requires_grad==False`, so we don't pass its gradient. But for `a`...

Step 8: Plot Twist (again)!

We need to store this gradient, but remember that we already stored `a.grad=`$\left[1\right]$ in Step 6. What do we do with this new one?

Answer: We **"accumulate"** gradients with a `+=` operation. This is handled by `AccumulateGrad.apply()`.

```
>>> a.grad
tensor(1.)
>>> grad_fn.apply(gradient_e_wrt_a)
>>> a.grad
tensor(3.)
```

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

We're done! But before we finish: why `+=`? Why would we add these gradients? Find out on the next page...

0.1.6 Accumulating Gradients

Q: But why do we "accumulate" gradients with +=?

A: Because of the **generalized chain rule**.

Generalized Chain Rule

For a function f with scalar output y, where each of its M arguments are functions $(g_1, g_2, ..., g_M)$ of N variables $(x_1, x_2, ..., x_N)$:

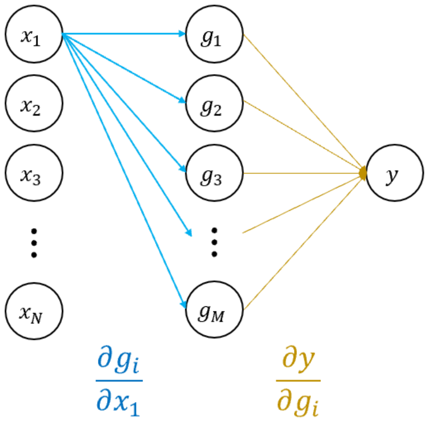
$$y = f(g_1(x_1, x_2, ..., x_N), g_2(x_1, x_2, ..., x_N)..., g_M(x_1, x_2, ..., x_N))$$

the partial derivative of its output w.r.t. **one input variable** x_i is:

$$\frac{\partial y}{\partial x_i} = \frac{\partial y}{\partial g_1}\frac{\partial g_1}{\partial x_i} + \frac{\partial y}{\partial g_2}\frac{\partial g_2}{\partial x_i} + ... + \frac{\partial y}{\partial g_M}\frac{\partial g_M}{\partial x_i}$$

Notice, we're adding products of partial derivatives. You can interpret this as **summing x_i's various "pathways" (chain rule expansions) to influencing the output y.**

To illustrate, let's say we want $\frac{\partial y}{\partial x_1}$:


x_1
x_2
x_3
x_N
g_1
g_2
g_3
g_M
y
$\frac{\partial g_i}{\partial x_1}$
$\frac{\partial y}{\partial g_i}$

$$\frac{\partial y}{\partial x_1} = \frac{\partial y}{\partial g_1} \cdot \frac{\partial g_1}{\partial x_1} + \cdots + \frac{\partial y}{\partial g_M} \cdot \frac{\partial g_M}{\partial x_1}$$

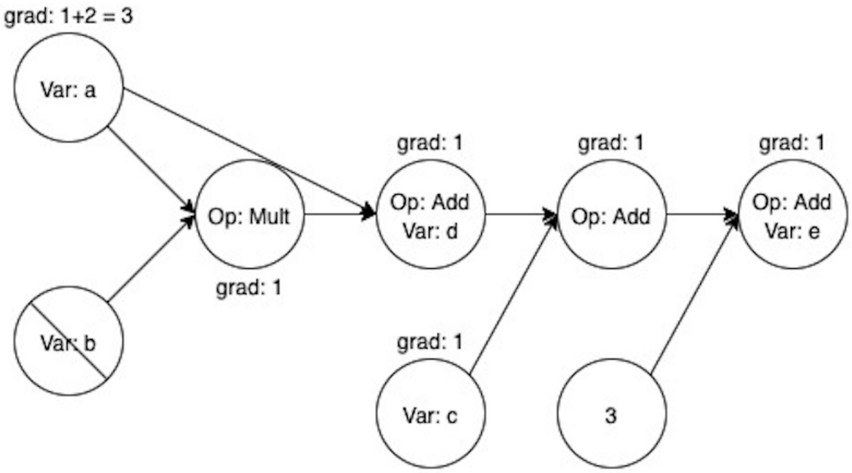
Notice that each summand is a single reverse-order "pathway" that leads from x_1 all the way to y. In autograd, every time a path finishes tracing back to the input, the term's product is stored with a `+=` until the other paths make it back. When all paths make it back to the node, all the terms have been summed, and its partial gradient is complete.

Autograd essentially does this, except it's not just w.r.t. one variable, it's **the partial w.r.t. ALL gradient-enabled tensors at the same time**. It does this because pathways often overlap; if we calculated each partial separately, we'd be re-calculating the same term multiple times, wasting computation.

Autograd's pretty good!

0.1.7 Conclusion: Code

We're done! Here were our results:



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

Forward:

```
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
```

Backward:

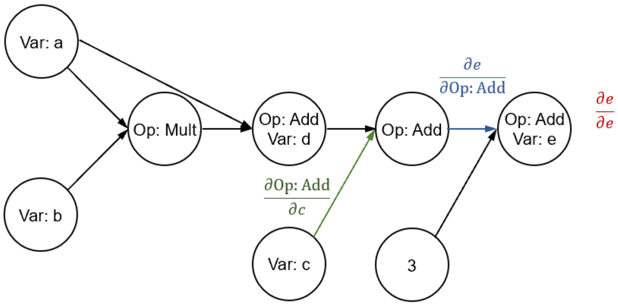
```
e.backward()

>>> print(a.grad)
tensor(3.)
>>> print(b.grad) # No result returned, empty gradient
>>> print(c.grad)
tensor(1.)
```

Nice.

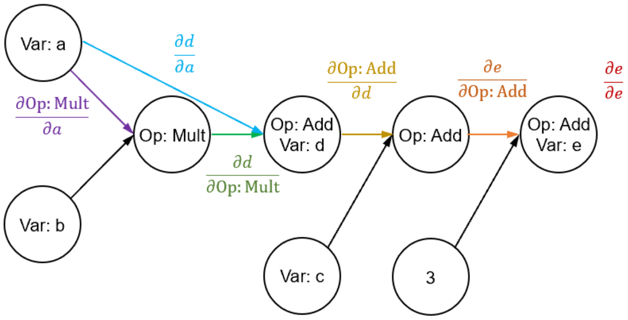
0.1.8 Conclusion: Symbolic Math

Here's the equivalent symbolic math for each of the two gradients we stored.

$$\begin{aligned}\frac{\partial e}{\partial c} &= \frac{\partial e}{\partial e} \cdot \frac{\partial e}{\partial \text{Op: Add}} \cdot \frac{\partial \text{Op: Add}}{\partial c} \\ &= [1] \cdot [1] \cdot [1] \\ &= [1]\end{aligned}$$


Var: a
Var: b
Op: Mult
Op: Add
Var: d
Op: Add
Op: Add
Var: e
Var: c
3
$\frac{\partial e}{\partial \text{Op: Add}}$
$\frac{\partial e}{\partial e}$
$\frac{\partial \text{Op: Add}}{\partial c}$

$$\begin{aligned}\frac{\partial e}{\partial a} = & \frac{\partial e}{\partial e} \cdot \frac{\partial e}{\partial \text{Op: Add}} \cdot \frac{\partial \text{Op: Add}}{\partial d} \cdot \frac{\partial d}{\partial a} \\ & + \frac{\partial e}{\partial e} \cdot \frac{\partial e}{\partial \text{Op: Add}} \cdot \frac{\partial \text{Op: Add}}{\partial d} \cdot \frac{\partial d}{\partial \text{Op: Mult}} \cdot \frac{\partial \text{Op: Mult}}{\partial a} \\ = & [1] \cdot [1] \cdot [1] \cdot [1] + [1] \cdot [1] \cdot [1] \cdot [1] \cdot [2] \\ = & [1] + [2] \\ = & [3]\end{aligned}$$


Var: a
Var: b
Op: Mult
Op: Add
Var: d
Op: Add
Op: Add
Var: e
Var: c
3
$\frac{\partial d}{\partial a}$
$\frac{\partial \text{Op: Mult}}{\partial a}$
$\frac{\partial d}{\partial \text{Op: Mult}}$
$\frac{\partial \text{Op: Add}}{\partial d}$
$\frac{\partial e}{\partial \text{Op: Add}}$
$\frac{\partial e}{\partial e}$

Notice how much overlap there is between paths; autograd only computed that overlap once.

Again, autograd's pretty good!

0.2 Autograd File Structure

It may be hard to keep track of all the autograd-specific code, so we've provided a quick guide below.

```
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[6]. This also includes loss functions.

```
functional.py
├── class Add ........................ [Given] Element-wise addition between tensors
├── class Sub ........................ Same as above; subtraction
├── class Mul ........................ Element-wise tensor product
├── class Div ........................ Element-wise tensor division
├── class Transpose .................. [Given] Transposing a tensor
├── class Reshape .................... [Given] Reshaping a tensor
├── class Log ........................ [Given] Element-wise log of a tensor
└── function cross_entropy() ......... Cross-entropy loss (loss function in comp graph)
```

Note: elementary operators are generally subclasses of `autograd_engine.Function`[7].

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[8].

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.

[6] See the actual Torch's `nn.functional` for ideas on what operations belong here

[7] In Python, subclasses indicate their parent in their declaration. i.e. `class Add(Function)`

[8] Again, see the actual `torch.Tensor` for ideas

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 2.6 (all problems before Problem 3: MNIST)

```
./grade.sh 1
```

If this doesn't work, converting line-endings may help:

```
sudo apt install dos2unix
dos2unix grade.sh
./grade.sh 1
```

If all else fails, you can run the autograder manually with this:

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

Note: as MNIST is not autograded, 100/110 is a "full" score for this section.

0.3.2 Running Local Autograder (MNIST only)

After completing all of Problem 3, use this autograder to test MNIST and generate plots.

```
./grade.sh m
```

You can also run it manually with this:

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

Note: If you're using WSL, plotting may not work unless you run VcXsrv or Xming; see here for instructions.

0.3.3 Running the Sandbox

We've provided `sandbox.py`: a script to test and easily debug basic operations and autograd. When you add your own new operators, write your own tests for these operations in the sandbox.

```
python3 sandbox.py
```

0.3.4 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.

To receive credit for problem 3, you must already have plots generated by the MNIST autograder. Make sure the plot image is in the hw1 folder. The plots will be included automatically the next time you generate a submission.

1 Implementing Autograd [Total: 40 points]

We'll start implementing autograd by programming some elementary operations.

Advice:

1. We've provided `sandbox.py` (optional) to help you easily debug operations and the rest of autograd.
2. Don't change the names of existing classes/variables. This will confuse the Autograder.
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[9]. Also, see this recitation on debugging: Recitation 0F ↗
5. Read error messages closely, and feel free to read the autograder's code for context/clues.

1.1 Basic Operations [15 points]

Let's begin by defining how some elementary tensor operations behave in the computational graph.

1.1.1 Element-wise Subtraction [5 points]

In `nn/functional.py`, we've given you the `Add` class as an example. Now, complete `Sub`.

Then, in `tensor.py`, complete `Tensor.__sub__()`. This function defines what happens when you try to subtract two tensors like this: `tensor_a - tensor_b`[10]. By calling `F.Sub` here, we're connecting every "`-`" operation to the computational graph.

1.1.2 Element-wise Multiplication [5 points]

In `nn/functional.py`, complete `Mul`. Then create/complete `Tensor.__mul__()`

1.1.3 Element-wise Division [5 points]

In `nn/functional.py`, complete `Div`. Then create/complete `Tensor.__truediv__()`.

1.2 Autograd [25 points]

Now to implement the core Autograd engine. If you haven't yet, we encourage you to read Appendix A. Note that while implementing this, you may temporarily break the basic operations tests, so plan to debug.

- In `autograd_engine.py` finish `Function.apply()`. During the forward pass, this both runs the called operation AND adds node(s) onto the computational graph. This method is called whenever an operation occurs (usually in `tensor.py`). We've given you some starting code; see the intro section for hints on completing this.
- In `tensor.py`, implement the `Tensor.backward()` function. This should be very short - it kicks off the DFS. See step 1 of the backward example for what this is doing.
- In `autograd_engine.py`, implement the `backward(grad_fn, grad_of_output)` 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.

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

[10] See here for more info ↗

2 MLPs [Total: 60 points]

Now for the fun stuff - coding a trainable MLP.

This will involve coding network layers, an activation function, and an optimizer. Only after completing these tasks three will you be able to challenge the formidable MNIST 👹.

Note: from now on, you will need to implement your OWN operations/functions if you need them. We have given you some freebies, however.

A common question is whether an operation should account for multiple dimensions or other edge cases. A good rule of thumb is to implement what you need, and then modify the code later when needed. Also, avoid implementing any operations you don't need. Be sure to keep using the sandbox to develop/debug.

2.1 Linear Layer [10 points]

First, in `nn/sequential.py`, implement `Sequential.forward()`. This class simply passes input data through each layer held in `Sequential.layers` and returns the final result[11].

Next, in `nn/linear.py`, complete `Linear.forward()`. **This will require implementing matrix operation(s) in `nn/functional.py`.** Hint: what operations are in the formula below?

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

(Note: this formulation should be cleaner than the commonly used $Wx + b$.)

Note that you will have to modify `Add` to support tensors with different shapes. This is done with broadcasting ↗. While NumPy will handle broadcasting for you in forward, you'll need to implement the derivative of the broadcast during the backward.

For example, say we add a tensor A, shaped (3, 4), with B, shaped (4,). NumPy will turn B into (3, 4) implicitly in forward. However in backward, the gradient we calculate with respect to B will be shaped (3, 4); it needs to be reduced to shape (4,) before we return it.

To do this, we recommend you implement a function `unbroadcast(grad, shape)` in `functional.py`. In theory, this function should be able to unbroadcast gradients for ANY operation (not just Add, but also Sub, Mult, Etc). However, if you'd prefer to implement just a limited 2D unbroadcast for Add, that will work for now.

In `sandbox.py`, you can use `testbroadcast` to test your solution to unbroadcasting.

2.2 ReLU [5 points]

First, in `nn/functional.py`, create and complete a `ReLU(Function)` class. Similar to the elementary operations before, this class describes how the ReLU activation function works in the computational graph.

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

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

Then, in `nn/activations.py`, complete `ReLU.forward()` by calling the `functional.ReLU` function, just like you did with operations in `tensor.py`.

[11] `Sequential` can be used as a simple "model" or to group layers into a single module as part of a larger model. Useful for simplifying code.

2.3 Stochastic Gradient Descent (SGD) [10 points]

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

After gradients are calculated, optimizers like SGD are used to update trainable params in order to minimize loss.

Note that this class inherits from `optim.optimizer.Optimizer`. Also, make sure this code does NOT add to the computational graph.

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

2.4 Batch Normalization (BatchNorm) [10 points]

In `nn/batchnorm.py`, complete the `BatchNorm1d` class.

For a conceptual introduction to BatchNorm, see Appendix C.

BatchNorm (Ioffe and Szegedy 2015) uses the following equations for its forward function:

$$\mu_{\mathcal{B}} = \frac{1}{m} \sum_{i=1}^{m} \mathbf{x}_i \tag{3}$$

$$s_{\mathcal{B}}^2 = \frac{1}{m} \sum_{i=1}^{m} (\mathbf{x}_i - \mu_{\mathcal{B}})^2 \tag{4}$$

$$\hat{\mathbf{x}}_i = \frac{\mathbf{x}_i - \mu_{\mathcal{B}}}{\sqrt{s_{\mathcal{B}}^2 + \epsilon}} \tag{5}$$

$$\mathbf{y}_i = \gamma \hat{\mathbf{x}}_i + \beta \tag{6}$$

x_i is the input to the BatchNorm layer. Within the forward method, compute the sample mean ($\mu_{\mathcal{B}}$), sample variance ($s_{\mathcal{B}}^2$), and norm ($\hat{\mathbf{x}}_i$). Epsilon (ϵ) is used when calculating the norm to avoid dividing by zero. Lastly, return the final output (y_i).

You may need to implement operation(s) like `Sum` in `nn/functional.py` and `tensor.py`. Remember to use matrix operations instead of `for` loops; loops are too slow.

Also, you'll need to calculate the **running mean/variance too**:

$$\sigma_{\mathcal{B}}^2 = \frac{1}{m-1} \sum_{i=1}^{m} (\mathbf{x}_i - \mu_{\mathcal{B}})^2 \tag{7}$$

$$E[x] = (1 - \alpha) * E[x] + \alpha * \mu_{\mathcal{B}} \tag{8}$$

$$Var[x] = (1 - \alpha) * Var[x] + \alpha * \sigma_{\mathcal{B}}^2 \tag{9}$$

Note: The notation above is consistent with PyTorch's implementation of Batchnorm. The α above is actually $1 - \alpha$ in the original paper.

BatchNorm operates differently during training and eval. During training (`BatchNorm1d.is_train==True`), your forward method should calculate an unbiased estimate of the variance ($\sigma_{\mathcal{B}}^2$), and maintain a running average of the mean and variance. These running averages should be used during inference (`BatchNorm1d.is_train==False`) in place of $\mu_{\mathcal{B}}$ and $s_{\mathcal{B}}^2$.

2.5 Cross Entropy Loss [15 points]

For info on how to calculate Cross Entropy Loss, see Appendix C.

There are two ways that you can complete this question and receive full credit.

1. You can complete the `cross_entropy` function we provided in `functional.py`. In this case, autograd will handle the backward calculation for you.
2. You can create and complete your own subclass of Function, just like the operations earlier. This means you'll need to implement the derivative calculation, which we describe in the appendix. You'll also need to change `nn.loss.CrossEntropyLoss.forward()` to call this subclass.

As long as your implementation is correct AND is correctly called by `nn.loss.CrossEntropyLoss.forward()`, you will receive full credit.

2.6 Momentum [10 points]

In `optim/sgd.py` modify `SGD.step()` to include "momentum". For a good explanation of momentum, see here ☐.

We will be using the following momentum update equation:

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

Note that you'll have to use `self.momentums`, which tracks the momentum of each parameter. Make sure this code does NOT add to the computational graph.

3 MNIST [10 points]

Finally, after all this, it's time to `print("Hello world!")`. But first, some concepts.

MNIST. Each observation in MNIST is a (28x28) grayscale image of a handwritten digit `[0-9]` that's been flattened into a 1-D array of floats between `[0,1]`. Your task is to identify which digit is in each image. You're also given the true label (`int` in `[0,9]`) for each observation.

Batching. In DL, instead of training on one observation at a time, we usually train on small, evenly-sized groups of points that we call "batches". Ideally (and generally in practice), this stabilizes training by decreasing the variation between individual data points.

During `forward()`, we put a single batch into a tensor and pass it through the model. This means we end up with a vector of losses: one loss value for each training point. We then aggregate this vector to create a single loss value (usually by averaging or summing). *Your implementation of XELoss already handles aggregation by averaging; just use this as is.*

Train/Validation/Test. Review this to understand your upcoming task. Today, we're only implementing training and validation routines. The autograder already has pre-split train/val data, and will also handle the plotting.

3.1 Initialize Objects

In `hw1/mnist.py`, complete `mnist()`. Initialize your criterion (`CrossEntropyLoss`), your optimizer (`SGD`), and your model (`Sequential`).

Set the learning rate of your optimizer to `lr=0.1`.

Use the following architecture:

```
    Linear(784, 20) -> BatchNorm1d(20) -> ReLU() -> Linear(20, 10)
```

Finally, call the `train()` method, which we'll implement below.

3.2 train()

Next, implement train(). Some notes:

1. We've preset batch_size=100 and num_epochs=3. Feel free to adjust while developing/testing.
2. Make sure to shuffle the data at the start of each epoch (hint: np.random.shuffle())
3. Make sure that input/label tensors are NOT gradient enabled.
4. For the sake of visualization, perform validation after every 100 batches and store the accuracy. Normally, people validate once per epoch, but we wanted to show a detailed curve.

Pseudocode:

```
def train():
    model.activate_train_mode()
    for each epoch:
        shuffle_train_data()
        batches = split_data_into_batches()
        for i, (batch_data, batch_labels) in enumerate(batches):
            optimizer.zero_grad() # clear any previous gradients
            out = forward_pass(batch_data)
            loss = criterion(out, batch_labels)
            loss.backward()
            optimizer.step() # update weights with new gradients
            if i is divisible by 100:
                accuracy = validate()
                store_validation_accuracy(accuracy)
                model.activate_train_mode()
    return validation_accuracies
```

(this is a typical routine; will become familiar throughout the semester)

3.3 validate()

Finally, implement validate(). Pseudocode again:

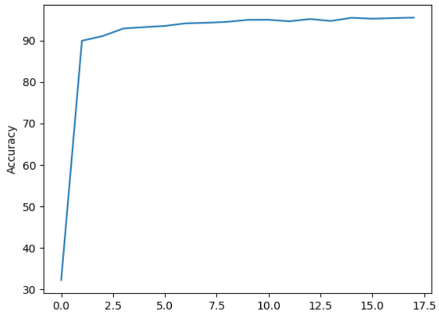
```
def validate():
    model.activate_eval_mode()
    batches = split_data_into_batches()
    for (batch_data, batch_labels) in batches:
        out = forward_pass(batch_data)
        batch_preds = get_idxs_of_largest_values_per_batch(out)
        num_correct = compare(batch_preds, batch_labels)
    accuracy = num_correct / len(val_data)
    return accuracy
```

Plotting and Submission

After completing the above methods, you can run the MNIST autograder (NOT the regular autograder; see Section 0.3) to test your script and generate a plot that shows the val accuracy at each epoch. The plot will be stored as ‘`validation_accuracy.png`’.

Note: Make sure this image is in the hw1 folder; it needs to be here to be included in the submission. Older versions of the handout may not automatically place them there.

The plot should look something like this:



Your plot doesn’t have to match this plot exactly, it can even train slower or end up a less accurate, that’s fine. It just needs to be clear that the network is learning. Your y axis may also be between `[0,1]` instead of `[0,100]`; that’s ok too. Don’t worry about axes/plot titles; the autograder intentionally only labels the y axis.

Note: the plot must be in your final submission to receive credit for Problem 3. When you run the autograder, the image should automatically be placed in the main folder. Then, running the submission generator will automatically place the image in submission. We’ll grade it manually afterwards.

You’re done! Implementing this is seriously no easy task. Very few people have implemented automatic differentiation from scratch. Congrats and great work. More not-easy tasks to come ☺.

References

Ioffe, Sergey and Christian Szegedy (2015). “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *CoRR* abs/1502.03167. arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

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 (some node object or None)
 ├── grad (Tensor)
 └── is_parameter (boolean)
```

A.1 `requires_grad` and `is_leaf`

The combination of these two variables determine the tensor's node type. Specifically, during `Function.apply()`, you'll be using these params to determine what node to create for the parent.

is_leaf (default: `True`) indicates whether this tensor is a "**Leaf Tensor**". Leaf Tensors are defined as not having any gradient-enabled parents. In short, any node that has `requires_grad=False` is a Leaf Tensor[12].

requires_grad[13] (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`. This node is a gradient-enabled node that has no parents. The `.apply()` of this node handles accumulating gradients in the tensor's `.grad`.

2. **BackwardFunction** node. Has `is_leaf=False` and `requires_grad=True`. This is an intermediate node, where gradients are calculated and passed, but not stored. The `.apply()` of this node calculates the gradient(s) w.r.t. the inputs and returns them.

3. **Constant** node (Store a `None` in the parent list). 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. This would be something like input data.

Remember, during `Function.apply()`, we're creating and storing these nodes in (`Tensor.grad_fn`).

Note: if any of a node's parents `requires_grad`, this node will also `require_grad`.

This is so that autograd knows to pass gradients onto gradient-enabled parents. For example, see the diagram in Section 0.1.4. `Op:Mult` would have `requires_grad==True`, because at least one of his parents (`a`) requires grad. But if all parents aren't gradient enabled, a child would have `requires_grad=False` and `C.is_leaf=True`.

A.2 `is_parameter`

Indicates whether this tensor contains parameters of a network. For example, `Linear.weight` is a tensor where `is_parameter` should be `True`. NOTE: if `is_parameter` is `True`, `requires_grad` and `is_leaf` must also be `True`.

[12] It's impossible for a tensor to have both `requires_grad=False` and `is_leaf=False`, hence 3 possible node types.

[13] 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[14]. 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[15].

We’ll also need another tensor containing the **true labels** for each observation in the batch. This is a `long`[16] tensor of size $(N,)$, where each entry contains an index of the correct label.

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**:

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

Remember, the above formula is for a single observation. You need to do this for all observations in the batch. Also, **don’t directly implement the above, as it’s numerically unstable. Read section B.1 for how to implement it.)**

Softmax scales the values in a 1D vector into “probabilities” that sum up to 1. We then scale the values 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).

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

For the numerator, you’re summing the values at the correct indices. The N in the denominator is the `batch_size`. Essentially, for a batch of outputs, we’re getting our average confidence in the correct answer.

That’s it! Calling `NLLLoss(LogSoftmax(logits), labels)` will give you your final loss.
Note that it’s averaged across the batch.

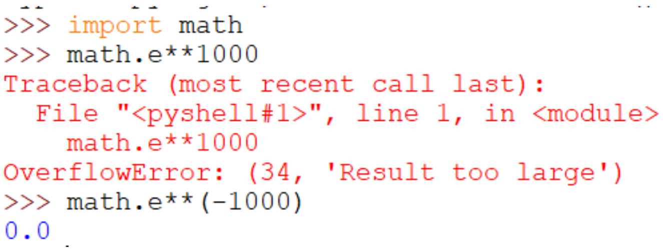
[14] It’s quite popular and commonly used in many different applications.

[15] During val/test, the index of the **maximum value** in the row is usually returned as your label.

[16] The official Torch uses long; for us, it should be ok to use int

B.1 Stabilizing LogSoftmax with the LogSumExp Trick

When implementing `LogSoftmax`, you'll need the **LogSumExp** trick. This technique 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 XE Loss - Derivation and Derivative

This section contains conceptual background and the derivative of XE Loss (which you may find useful).

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.

Simply put, it tells us how far our model is from the true distribution P. It does this by measuring how much information we receive when receiving new observations from Q.

Cross-Entropy is fully minimized 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.

For example, consider a coin toss. Even if we know the probability of heads (Bernoulli parameter: p) ahead of time, we don't know what the result will be until we observed it. The greater p is, the more certain we are about the outcome and the less information we expect to receive upon observation.

When calculating XELoss, we first use the softmax function to normalize our network's outputs before calculating the loss. The 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.

Next, we use Cross-Entropy as our objective function. 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:

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

NOTE: Implementing the above directly may not give you the correct result. Remember, you averaged over the batch during `Softmax()` by dividing by N.

This derivative is pleasingly simple and elegant. Remember, this is the derivative of softmax with cross-entropy divergence with respect to the input. What this is telling us is that when $y_j = 1$, the gradient is negative; thus the opposite direction of the gradient is positive.

In short, it is telling us to increase the probability mass of that specific output through the softmax.

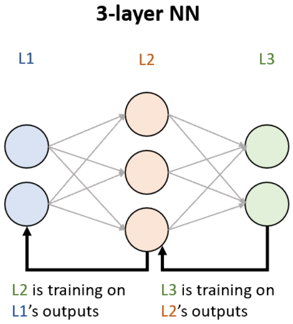
C BatchNorm

Batch Normalization ("BatchNorm") is a wildly successful technique for improving the speed and quality of learning in NNs. It does this by attempting to address an issue called **internal covariate shift**.

We encourage you to read the original paper ; it's written very clearly, and walks you through the math and reasoning behind each decision.

C.1 Motivation: Internal Covariate Shift

Internal covariate shift happens while training an NN.


3-layer NN
L1
L2
L3
L2 is training on L1's outputs
L3 is training on L2's outputs

In an MLP, each layer is training based on the activations of previous layer(s). But remember - **previous layers are ALSO training, changing their weights and outputs all the time**. This means that the later layers are working with frequently shifting information, leading to less stable and significantly slower training. This is especially true at the beginning of training, when parameters are changing a lot.

That's internal covariate shift. It's like if your boss AND your boss's boss joined the company on the same day that you did. And they also have the same level of experience that you do. Now you'll never get into FAANG...

C.2 Intro to BatchNorm

BatchNorm essentially introduces normalization/whitening *between* layers to help mitigate this problem. Specifically, a BN layer aims to linearly transform the output of the previous layer s.t. across the entire dataset, each neuron's output has `mean=0` and `variance=1` AND is linearly decorrelated with the other neurons' outputs.

By 'linearly decorrelated', we mean that for a layer l with m units, individual unit activities $\mathbf{x} = \{\mathbf{x}^{(k)}, \dots, \mathbf{x}^{(d)}\}$ are independent of each other – $\{\mathbf{x}^{(1)} \perp \dots \mathbf{x}^{(k)} \dots \perp \mathbf{x}^{(d)}\}$. Note that we consider the unit activities to be random variables.

In short, we want to make sure that normalization/whitening for a single neuron's output is happening consistently across the entire dataset. In truth, this is not computationally feasible (you'd have to feed in the entire dataset at once), nor is it always fully differentiable. So instead, we maintain "running estimates" of the dataset's mean/variance and update them as we see more observations.

How do we do this? Remember that we're training on batches[17] - small groups of observations usually sized 16, 32, 64, etc. **Since each batch contains a random subsample of the dataset, we assume that each batch is somewhat representative of the entire dataset**. Based on this assumption, we can use their means and variances to update our running estimates.

[17] Technically, *mini-batches.* "Batch" actually refers to the entire dataset. But colloquially and even in many papers, "batch" means "mini-batch".

C.3 Implementing BatchNorm

This section gives more detail/explanation about the implementation of BatchNorm. Read it if you need any clarifications.

Given this setup, consider $\mu_{\mathcal{B}}$ to be the mean and $\sigma_{\mathcal{B}}^2$ the variance of a unit's activity over the batch $\mathcal{B}$[18]. For a training set $\mathcal{X}$ with n examples, we partition it into n/m batches $\mathcal{B}$ of size m. For an arbitrary unit k, we compute the batch statistics $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}^2$ and normalize as follows:

$$u_{\mathcal{B}}^{(k)} \leftarrow \frac{1}{m} \sum_{i=1}^{m} \mathbf{x}_i^{(k)} \tag{10}$$

$$(\sigma_{\mathcal{B}}^2)^{(k)} \leftarrow \frac{1}{m} \sum_{i=1}^{m} \left(\mathbf{x}_i^{(k)} - \mu_{\mathcal{B}}^{(k)} \right)^2 \tag{11}$$

$$\hat{\mathbf{x}}_i \leftarrow \frac{\mathbf{x}_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \tag{12}$$

Note that we add $\epsilon = 1e - 5$ to $\sigma_{\mathcal{B}}^2$ in order to avoid dividing by zero.

A significant issue posed by simply normalizing individual unit activity across batches is that it limits the set of possible network representations. In order to avoid this, we introduce a set of trainable parameters ($\gamma^{(k)}$ and $\beta^{(k)}$) that learn to make the BatchNorm transformation into an identity transformation.

To do this, these per-unit learnable parameters $\gamma^{(k)}$ and $\beta^{(k)}$ rescale and reshift the normalized unit activity. Thus the output of the BatchNorm transformation for a data example, $\mathbf{y}_i$ is:

$$\mathbf{y}_i \leftarrow \gamma \hat{\mathbf{x}}_i + \beta$$

Training Statistics

$$E[x] = (1 - \alpha) * E[x] + \alpha * \mu_{\mathcal{B}} \tag{13}$$

$$Var[x] = (1 - \alpha) * Var[x] + \alpha * \sigma_{\mathcal{B}}^2 \tag{14}$$

Note: The notation above is consistent with PyTorch's implementation of Batchnorm. The α above is actually $1 - \alpha$ in the original paper.

This is the running mean $E[x]$ and running variance $Var[x]$ we talked about. We need to calculate them during training time when we have access to training data, so that we can use them to estimate the true mean and variance across the entire dataset.

If you didn't do this and recalculated running means/variances during test time, you'd end up wiping the data out (mean will be itself, var will be inf) because you're typically only shown one example at a time during test. This is why we use the running mean and variance.

[18] Again, technically 'mini-batch'