

# EE 148a [Winter 2026]

## Problem Set 1

**Due:** 01/27/26 at 11:59 p.m.

**Submit:** on [Gradescope](#)

Originally Prepared By: Armeet Singh Jatyani & Felipe Cruz

Welcome to EE 148a! Collaboration is allowed, just list the names of the people you collaborated in the header. If you choose to use L<sup>A</sup>T<sub>E</sub>X to type your submission, we recommend filling in this template.

**You do not need to upload your code as part of the submission. However, the coding problems can be a useful resource for your projects. The zip file for the coding problems is uploaded on Piazza in the Resources section.**

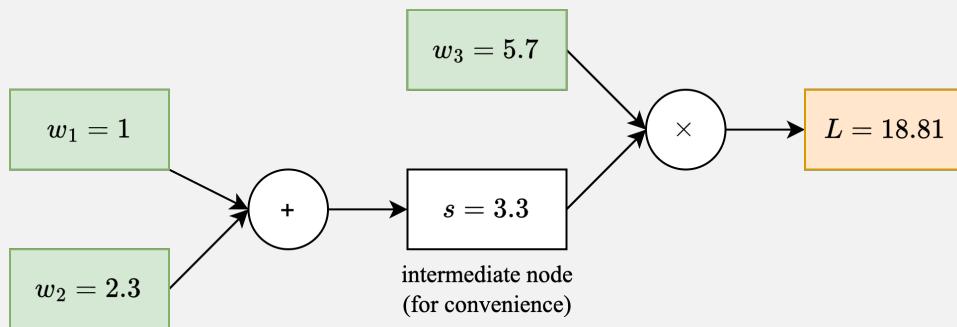
If you have any other questions, come to office hours or ask us on Piazza for a quick response.

### Problem 1: Gradient Calculation by Hand

[10 pts]

Let's consider a very simple expression  $L = (w_1 + w_2) \cdot w_3$ . In this expression, we can think of  $w_1, w_2, w_3$  as parameters we can change, and  $L$  as the result of this expression. If we told you  $L$  was a loss function and gave you the objective of minimizing loss, how would you choose your parameter weights  $w_i$ ? As covered in lecture, we can use *gradient descent* (we simplify it here)...

- Compute the rate of change of the result  $L$  with respect to each parameter  $w_i$ . Since our expression for  $L$  may have many variable parameters, we find the partial derivative denoted by  $\frac{\partial L}{\partial w_i}$ . We call this the *gradient*.
- Update our parameters via the rule  $w_i := w_i - \eta \frac{\partial L}{\partial w_i}$  for some small learning rate  $\eta$ . There are different strategies for picking a learning rate (see lecture). This step moves each parameter  $w_i$  in the direction opposite to the gradient.
- Repeat the above two steps until we converge to some  $L^*$ .



**Figure 1:** Expression diagram for  $L$ . We add intermediate node  $s$  to represent the result of  $w_1 + w_2$ . This is a deliberate choice that'll make more sense when we automate gradient backprop (next prob).

(a) Calculate  $\frac{\partial L}{\partial w_3}$ . (2 pts)

Please give an expression in terms of any of  $w_1, w_2, w_3, s, L$ . Then plug in the values from fig. 1 to get a value.

(b) Calculate  $\frac{\partial L}{\partial w_1}$  and  $\frac{\partial L}{\partial w_2}$ . *hint: use chain rule* (2 pts)

Your expressions must contain  $\frac{\partial s}{\partial w_1}$  or  $\frac{\partial s}{\partial w_2}$ . Evaluate using values from fig. 1.

(c) Update parameters (4 pts)

Now with learning rate  $\eta = 0.01$ , calculate the new parameter values  $w_1, w_2, w_3$  after one step. Notice that we calculate all the gradients first, then update all of the parameters. We will follow this order when we automate everything in problem 2.

(d) Analysis (2 pts)

Using the newly updated parameter values, calculate the new result value of  $L$  and compare it with the original value. What do you expect? What do you see? Did  $L$  increase or decrease?

## Problem 2: Autograd from Scratch

[0 pts]

In this problem, you will implement your very own *scalar autograd* (differentiation) library. If you're scared, don't be! You'll be writing just 100-150 lines of Python code in 2 files. This problem aims to introduce you to automatic differentiation and backpropagation by demonstrating how they work at the scalar (single number) level. More advanced libraries like PyTorch, JAX, and TensorFlow/Keras extend these concepts to the tensor/matrix/vector levels for automatic differentiation. This assignment is inspired and adapted from Andrej Karpathy's [micrograd](#) (1).

### Setup

Download the assignment `.zip` from Gradescope and extract it somewhere on your computer. Follow the project setup in `SETUP.md`. It'll guide you on how to install `graphviz` and `uv`, which we'll be using to manage your python environment. We've also made a setup video guide [here](#).

### Automatic Differentiation

Say we have an expression  $y = ax + b$ . We know that  $\frac{\partial y}{\partial x} = a$  or  $\frac{\partial y}{\partial a} = x$  or  $\frac{\partial y}{\partial b} = 1$ , but how do we calculate these partial derivatives automatically? What about far more complex expressions with nested structures?

The idea is to wrap constants and variables (to us they are all unit values) like  $a, b, x$  in a `Value` object/class (which you will implement), to exploit the modularity of both the "forward" and "backward" computation logic. So  $3.14 \rightarrow \text{Value}(3.14)$ . Then, we can construct complex expressions with these unit `Value`'s as building blocks. Under the hood, we will automatically construct a computation graph (DAG) as visualized in fig. 1. `Value` objects have methods that allow them to interact with other `Value`'s. For instance, you will implement functions for adding, subtracting, multiplying, and dividing any two `Value` objects as well as special transformations called *non-linearities* (ReLU, Sigmoid, tanh). For example,  $\text{Value}(3.14).\text{sigmoid}() = \sigma(3.14) = \frac{1}{1+e^{-3.14}}$ . Finally, you will implement the backbone of the autograd engine, the `backward()` method, which relies on the computation graph we constructed when piecing together values.

### Gradient Back-propagation

To find the gradient of the loss function  $L$  with respect to a parameter  $w$ , we use the chain rule.  $w_{next}$  is the node right after the current parameter  $w$ , so in fig. 1 if  $w = w_1$  then  $w_{next} = s$ .

$$\frac{\partial L}{\partial w} = \underbrace{\frac{\partial L}{\partial w_{next}}}_{\text{Gradient of next value.}} \cdot \underbrace{\frac{\partial w_{next}}{\partial w}}_{\text{Local gradient}}$$

Start by reading `grad/autograd.py`. Understand the `__init__`, `__repr__`, `parse`, and `__add__` functions in the `Value` class.

Implement the following functions to enable `Value` operations. We've done `__add__` and `__radd__` for you already, so read through those first. If you don't follow a similar order/structure, we cannot guarantee that your tests will pass. To understand what `__radd__` does, read the first response to [this thread](#). Run tests with `uv run tests.py` or `python tests.py` from the root directory after you implement each function! These functions are methods of the `Value` class, so `self` refers to the current object, and `other` refers to the other object. For example, Python maps `val_a + val_b` to the `__add__` function with `self=val_a` and `other=val_b`.

**CAREFUL:** When updating `self.grad` in these parts, use `self.grad +=` instead of `self.grad =` because the same `Value` can be used twice in an expression, so we should be accumulating gradients, not overwriting what's already in a `Value`'s `self.grad` field.

- (a) `__mul__` (`self * other`)
- (b) `__sub__` (`self - other`)

hint: addition and negation  
  `--neg--` is already implemented for you

- (c) `--pow--` (`self ** other`)
- (d) `--truediv--` (`self / other`) hint: multiplication and power

Now you'll implement some non-linear transformations, often called *nonlinearities*. Specifically, you'll implement the ReLU, sigmoid, and tanh transformations. We use these non-linear transformations later when building neural networks. Where possible, try to reuse code by calling the earlier functions. Some functions and derivatives can be represented in terms of themselves, or other previously handled cases. You may use the `math` library to compute values of `tanh(x)` or `exp(x)`.

- (e) `f_relu`
- (f) `f_sigmoid`
- (g) `f_tanh`

To the final part of this problem! Implement the following...

- (h) `backward`
- (i) Compare with hand-backpropagation

You will complete a few lines in `examples/ex_2i_backprop.py`. In problem 1, we did backpropagation by hand, for the expression  $L = (w_1 + w_2) \cdot w_3$ . Recreate this same expression from fig. 1 Run the script to generate `figs/ex_backprop_before.png` and `figs/ex_backprop_after.png`. Comment on whether the calculated gradients, updated parameters  $w_1, w_2, w_3$ , and new result value  $L$  match with your previous results in Problem 1.

You've successfully implemented an autograd engine that is capable of minimizing any expression that can be composed by your `Value` class (which supports basic operations, in addition to three non-linearities). Notice that any loss function  $L$  that can be defined in terms of `Value`'s can be minimized with our autograd library.

### Problem 3: Multi Layer Perceptron (MLP) from Scratch

[0 pts]

At a high level, neural networks are functions  $G_\theta : \mathbb{R}^n \rightarrow \mathbb{R}^m$ . This notation means that  $G$  is a function with parameters  $\theta$ , domain  $\mathbb{R}^n$ , and range  $\mathbb{R}^m$ . So  $G$  takes in an input vector with  $n$  *features* and outputs a vector with  $m$  *features*.  $G$  will be a very long expression with many different parameters, but since we'll compose it using only `Value`'s, we can backpropagate and tune parameters exactly how we did in Problem 1 and 2. You will be writing a very tiny neural network library in `grad/nn.py`, with a single type of model: the multi-layer perceptron (MLP).

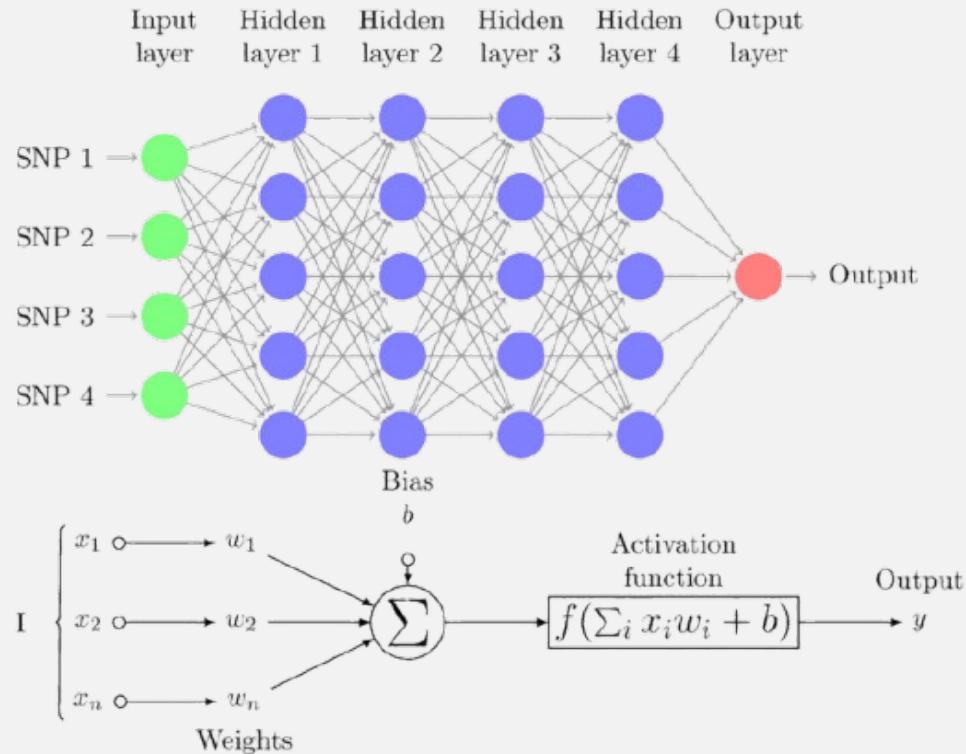


Figure 2

- (a) Implement class Perceptron
- (b) Implement class Layer
- (c) Implement class MLP
- (d) Train a PyTorch MLP on MNIST

Frameworks like PyTorch implement code for GPU/TPU accelerators and make things faster by leveraging parallelism. At their core, however, they implement the same functionality as our scalar autograd engine. To make training feasible, you will use PyTorch's `torch.nn` library, which exports an easy to use `Linear` layer: [read the docs here](#). We've filled in some boilerplate, but left the model choice (how many layers, hyperparameters, learning rate, activation functions, biases, augmentation techniques, etc.) up to you.

Train an MLP model that achieves 85% classification accuracy on Fashion MNIST! Do not use convolution (that'll come in a later assignment). You are limited to MLP `Linear` layers here. Please fill in all the TODO's in `examples/ex_3d_fashion_pytorch.ipynb`.

Include the final accuracy and loss vs. epochs plots in your submission.

**Problem 4: Nonlinearities, loss functions, convergence****[10 pts]**

1. Show that  $\tanh(x) + 1 = 2\sigma(2x)$  where  $\sigma$  is the sigmoid function. (2 pts)
2. Prove that function classes parametrized by both nonlinearities in (1) are identical, which is to say any function representable by a neural network with one activation function can be represented by a network with the other. Consider the role of bias terms. (3 pts)
3. Show that maximizing the log-likelihood function for a model where the outputs represent conditional class probabilities is equivalent to minimizing the cross-entropy loss. (3 pts)
4. Provide a justification for using these particular bias correction factors in Adam optimization. You may use words and/or derivations to support your explanation.

Recall from lecture III:

$$\bar{s}_{t+1} = \frac{s_{t+1}}{1 - \beta_1^t}$$
$$\bar{r}_{t+1} = \frac{r_{t+1}}{1 - \beta_2^t}$$

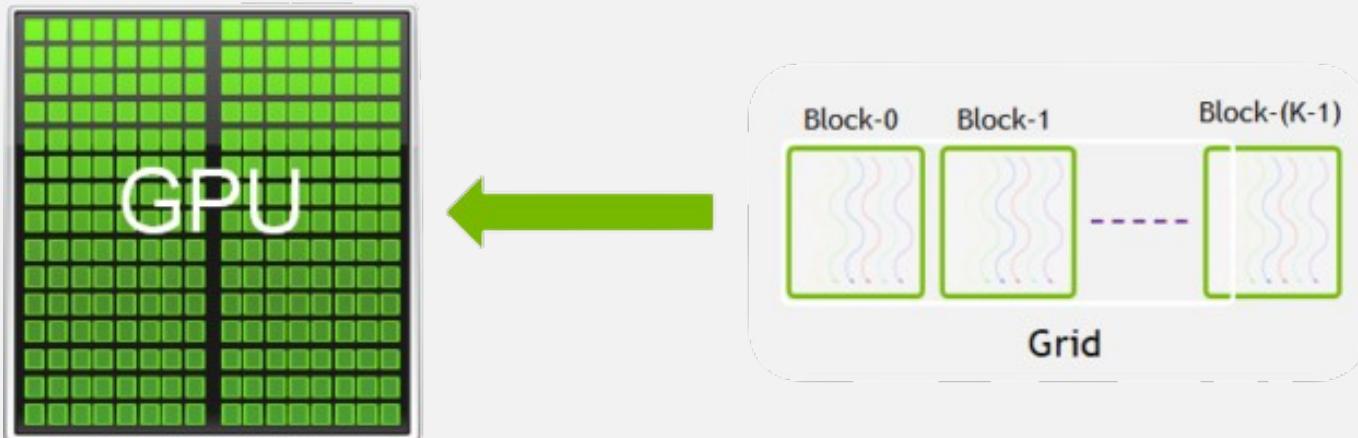
(2 pts)

As mentioned in Lectures I and III, GPUs have transformed neural network training by enabling massive task parallelization. This problem aims to introduce some fundamental concepts of GPU programming and its application to deep learning.

### Background & Definitions

A byproduct of their original purpose, GPUs are inherently optimized for massively parallel tasks, making them ideal for deep learning computations. Unlike CPUs, which typically have a single-to-double digit count of highly optimized cores, GPUs contain thousands of smaller cores optimized for high-throughput parallel processing. To gain a preliminary understanding of their importance and how they can be used to accelerate neural networks, we will briefly delve into Nvidia's CUDA platform, the most widely used platform for GPU programming.

1. **CUDA:** (*Compute Unified Device Architecture*) is a programming model developed for Nvidia's GPUs which allows users to write kernels and manage data transfer between the “host” (CPU) and “device” (GPU). The CUDA environment is C/C++-like\*, but we'll be using the Python library `pyCUDA` to interface with the APIs.
2. **Kernel:** A computation-performing function that runs on the GPU. Kernels are launched on the device and run on multiple threads simultaneously.
3. **Thread:** The fundamental unit of work within a kernel, which usually perform the same operation across different data.
4. **Block:** A group of threads that execute together on the GPU, logically unified by a self-contained shared memory pool. Blocks are isolated from one-another.
5. **Grid:** A virtualized collection of blocks that execute a kernel. Grids can be structured in 1 to 3 dimensions to represent parallel problems.



**Figure 3:** Visual representation of the CUDA hierarchy. (2)

\*Don't worry if you've never been exposed to C++, the syntax is essentially the same as C or Java, and you won't be doing anything too complicated. If your background is solely Python [here's a short guide with all the syntax you'll need](#). Additionally, a small section on interfacing with pointers is included in the Colab.

## Indexing Inside a Kernel

This hierarchy of **grid**, **block**, and **thread** forms the foundation of GPU programming, enabling organized parallel computations. But how can we use this for parallel processing?

Every thread has a unique identifier, which can be determined by its position within its block, and the position of its block within the grid. By using these identifiers, threads can independently process data in a manner that corresponds to their specific task in parallel. The CUDA runtime provides these through the variables below, which are made available within any kernel that you write.

- **threadIdx.x**, **threadIdx.y**, **threadIdx.z**: A thread's index within its block in the x, y, and z dimension.
- **blockIdx.x**, **blockIdx.y**, **blockIdx.z**: A block's index within the grid along respective dimension.
- **blockDim.x**, **blockDim.y**, **blockDim.z**: # total threads in each block along respective dimension.
- **gridDim.x**, **gridDim.y**, **gridDim.z**: # blocks in grid across respective dimension.

When launching a kernel with pyCUDA, we specify both the block size, **block**, and the grid size, **grid** we want to use to ensure we sufficiently cover the size of the input data. We will stick to a block size of  $16 \times 16 \times 1$ , and define our grid size based on the block size as and the size of the data.

Using these variables, we can map locations in memory to physical hardware cores. As this problem will deal with 2D structures only, you'll only need to use the following lines of code.

```
int idx = threadIdx.x+blockDim.x*blockIdx.x;
int idy = threadIdx.y+blockDim.y*blockIdx.y;
```

There are of course, tons of additional systems-level and hardware-level details, but for this problem the only other detail you need to know is that GPUs have their global memory (VRAM), which is larger yet slower to access than the shared memory available to a given block.

## General CUDA Workflow

Now, let's cover the typical framework to follow once you've established that a problem can be solved in parallel with a guided example. Create a copy of the Google Colab environment for this project [linked here](#) and follow along until you reach the next section.

## Matrix Multiplication Kernel

Now, it's your turn to dive deeper.

Matrix operations are a necessity at every step of training a deep learning model, from data augmentation to neuron activations and backpropagation. By definition, linear operators are distributive, and thus most of the matrix operations used behind the scenes of deep learning are known as *embarrassingly parallel* problems, since individual operations can be computed independently and brought together after.

Your task is to implement a kernel to compute the matrix multiplication

$$\begin{aligned}C &= AB \\ \dim(A) &= n \times m \\ \dim(B) &= m \times p\end{aligned}$$

Hint, recall the dot product definition of matrix multiplication

$$C_{ij} = \sum_{k=1}^m A_{ik} \cdot B_{kj}$$

### Tasks

1. Write down how long it takes for your `doublify` kernel to execute for  $n = 2^{12}$ .
2. Copy the benchmarks graph you made for your `matmul` kernel, and describe the runtime trends you see.
3. Explain (don't implement) 2 nontrivial\* ways your `matmul` kernel could be optimized for speed. This can be either software or hardware-dependent.

**Problem 6: Representation power****[20 pts]**

1. Show that adding layers to a neural network without non-linear activation functions does not increase its expressive power. (4 pts)
2. Give an example where adding layers to a deep network without non-linearities could reduce the expressive capacity of the network. (3 pts)
3. Design a one-layer perceptron that fits the logical functions **AND** and **OR** for 2D inputs. Why can't a single layer fit **XOR**? (5 pts)
4. Show that a neural network with a one-dimensional input  $x$  and one-dimensional output  $y$ , with ReLU activations of depth  $D$  and widths  $W_d$ ,  $d = 1, \dots, D$  can represent piecewise linear functions  $f$  with at most  $2^D \prod_{d=1}^D W_d$  linear pieces.

Essentially, if we define  $\kappa(f)$  as the smallest number of linear pieces in  $f$ , show

$$\kappa(y) \leq 2^D \prod_{d=1}^D W_d$$

Here are some hints.

- a. We can always represent a piecewise linear function with  $N$  pieces as a one-layer network with width  $N$  (case  $D = 1$ )
- b. Proceed by induction, keeping in mind that
  - i. The property is invariant to scale (the number of linear pieces of  $f$ ,  $\ell_f$  needs remains the same regardless of scaling  $f$ )
  - ii. The number of linear pieces in the sum of two functions is at most the sum of the number of linear pieces in each individual function.
  - iii. Applying a ReLU to a function at most doubles the number of linear pieces needed

(8 pts)

## References

- [1] Andrej Karpathy and Baptiste Pesquet. karpathy/micrograd, Aug 2024. URL <https://github.com/karpathy/micrograd/>.
- [2] Pradeep Gupta. Cuda refresher: The cuda programming model, Jun 2023. URL <https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/>.