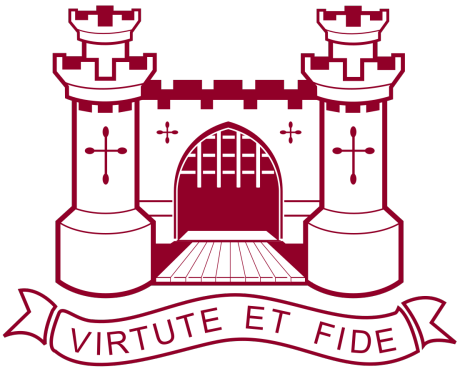
DeepDive: Deep Learning Library from scratch

Hills Road Sixth Form College

Minhal Valiya Peedikakkal

Computer Science NEA



September 2023

Contents

1 Analysis

1.1 Identification of the problem

The increasing complexity of deep learning libraries has presented significant challenges for new AI researchers and enthusiasts seeking to enhance these libraries for their own use. One major factor contributing to this difficulty is the use of performance optimization by library developers, which can lead to significant portions of the code being written in lower-level languages like C++. PyTorch, the most widely-used deep learning library, is a prime example of this, with 43% of its code base consisting of C++. As a result, researchers with expertise in higher-level programming languages like Python may find it challenging to contribute and customize these libraries. To address this issue, I will aim to develop a deep learning library in Python using 100% Python code with a focus on simplicity over complexity. This way it is easier for AI researchers and enthusiasts to understand and develop this library.

1.2 Clients

My primary client is Alex Davicenko, a programmer who is interested in learning deep learning. He is aware of the theory behind neural networks but is less proficient in its practical side. Although my primary client is not an AI researcher, he is another type of user at which this project is aimed: beginners. I interviewed my client to gain insight before building my project requirements

Interview with Client

.

ME: How do you think a fully implemented deep learning library in Python could help you and other new users?

ALEX DAVICENKO: A Python-only library would allow me to inspect all of the code and understand how everything fits together. Even if it sacrifices some performance, I think a simpler, more transparent design is more important for learning and experimentation. I could also easily modify and extend it since I am proficient in Python. This would really accelerate my ability to try new ideas with deep learning.

ME: What challenges have you faced when trying to use existing deep learning libraries like PyTorch?

ALEX DAVICENKO: I find existing libraries like PyTorch very complex and difficult to modify or extend. There is a lot of low-level C++ code that I don't understand. As a Python programmer without a background in C++, I struggle to contribute new features or optimizations.

ME: What are the main features you want to implement in the deep learning library?

ALEX DAVICENKO: Minimize package dependencies in the library itself. Do not use PyTorch except for testing purposes. Implement automatic differentiation in an extensible way with basic operators like addition, multiplication, and the dot product. Enable building neural networks intuitively through code using layers such as linear and potentially convolutional layers.

ME: Any extra features you would specifically like to see in the library?

ALEX DAVICENKO: Enable intuitive network construction using layers. Support adding custom layers and operators in Python without C++. As a challenge, you could accelerate training and inference of large models with GPU support.

ME: Any final things you would like to add?

ALEX DAVICENKO: Utilize object-oriented programming and standard naming conventions for methods and variables, similar to PyTorch, to facilitate code structure and make it easy to find for PyTorch users.

Interview Analysis

My client is looking for a deep learning library that is fully implemented in Python for simplicity, transparency, and ease of modification. Performance can be sacrificed for these goals. Enabling rapid experimentation with new ideas is also a priority.

Specific features the client wants:

- Minimal dependencies
- Automatic differentiation with extensibility
- Intuitive neural network construction using layers
- Ability to add custom layers and operators without C++
- Challenge: GPU Support for Large Models

My implementation approach should focus on the following.

- Object-oriented programming structure
- Utilizing naming conventions from PyTorch for familiarity
- Simplicity and transparency over optimization

This project will definitely challenge my code design skills. Moving forward, I need to balance ease of use and modification against performance, while enabling all the features listed by the client.

Client Hardware Specification

Knowing the client's hardware is useful for determining the appropriate types and sizes of models that can serve as examples for library usage. Here is the computer hardware specifications formatted as a table:

Component	**Details**
GPU	Model: GeForce RTX 3060 VRAM: 12 GB GDDR6 CUDA Cores: 3,584
CPU	Model: AMD Ryzen 7 5700X 8 cores 16 threads
Memory	RAM: 16 GB

Table 1: Computer Hardware Specifications

1.3 Existing similar software - PyTorch

PyTorch is the most popular and regarded as the best deep learning library (as of 2023), so I'll use it as a benchmark to showcase my project's advantages.

Installation

Installing PyTorch involves downloading a single package along with common dependencies, which is approximately 2.0GB in size. This can be a long process for new developers, especially considering the numerous dependencies required for GPU support. In contrast, my project aims to utilize lightweight packages like NumPy and CuPy (optional for GPU support), ensuring a simpler installation experience.

Usage

One of the important aspects of the library is its ease of use. Using PyTorch and observing its features has given me inspiration and ideas to how I should implement my library.

- **Automatic Differentiation:** The library must provide automatic differentiation capabilities. PyTorch's 'autograd' module provides this, enabling users to compute gradients easily.

```
import torch

# Define the input tensors with requires_grad=True to enable gradient
    computation
a = torch.tensor([2.0], requires_grad=True)
b = torch.tensor([3.0], requires_grad=True)

# Define the function y = x^2 + 3xz + z^2
y = a.pow(2) + 3 * a * b + b.pow(2)

# Compute the gradients of y with respect to x and z using PyTorch's
    autodiff
y.backward(torch.ones_like(y))

# Print the gradients of y with respect to x and z
print(a.grad)  # Output: tensor([7.])
print(b.grad)  # Output: tensor([10.])
```

- **Neural Networks:** Implementing common neural network functionality such as layers would be beneficial for building deep learning models. I will make an interface similar to PyTorch as shown below to construct neural networks. I should also make it easy for users to add new layers to the library.

```
import torch
import torch.nn as nn

# Define a simple neural network with one hidden layer
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(2, 3)
        self.fc2 = nn.Linear(3, 1)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Define the input tensor
x = torch.tensor([[0.0, 0.0], [0.0, 1.0], [1.0, 0.0], [1.0, 1.0]])

# Define the neural network
net = Net()

# input x into the net
y = net(x)
```

- **Data Utilities:** Providing utilities for data preprocessing, augmentation, and handling datasets can simplify the data pipeline.

```
import data_utils
p
dataset = data_utils.Dataset(data, labels)
dataloader = data_utils.DataLoader(dataset, batch_size=32, shuffle=True)

for inputs, targets in dataloader:
    # Training code here
```

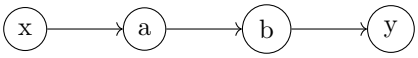
By incorporating such functionality, users will find it easier to work with my library and build with full potential

1.4 Algorithms Research

This section contains the algorithms I have researched and plan to use in my solution

Automatic Differentiation

Automatic differentiation (in short autodiff) is a technique used to numerically compute the exact gradient of a function. It exploits the fact that every computation, no matter how complex, executes a combination of elementary arithmetic operations e.g. addition and multiplication and elementary functions e.g. *sin, cos, exp, log* etc. The chain rule is applied to these operations allowing the gradient of complex function to be broken down and calculated independently and automatically



The graph above shows the data flow from the inputs x to the outputs y via variables a and b. Using the chain rule from calculus, we are able to calculate the gradient of y with respect to x

$$\frac{dy}{dx} = \frac{dy}{db} \cdot \frac{db}{da} \cdot \frac{da}{dx}$$

Automatic Differentiation has forward and reverse modes. The difference in both these methods are the order in which they are multiplied. In reverse mode autodiff, the flow of the gradients is reverse to the flow of the data [1]

$$\frac{dy}{dx} = \left(\frac{dy}{db} \cdot \frac{db}{da}\right) \cdot \frac{da}{dx} \quad \textbf{Reverse mode}$$

$$\frac{dy}{dx} = \frac{dy}{db} \cdot \left(\frac{db}{da} \cdot \frac{da}{dx}\right) \quad \textbf{Forward mode}$$

In neural networks, the input dimensionality is commonly much higher than that of the labels. As a result, reverse mode autodiff is computationally cheaper than forward mode for the application of neural networks [1]. Using the code snippet that shows the usage of PyTorch's autodiff, the process which automatic differentiation takes to find the gradients

$$a = 2.0$$

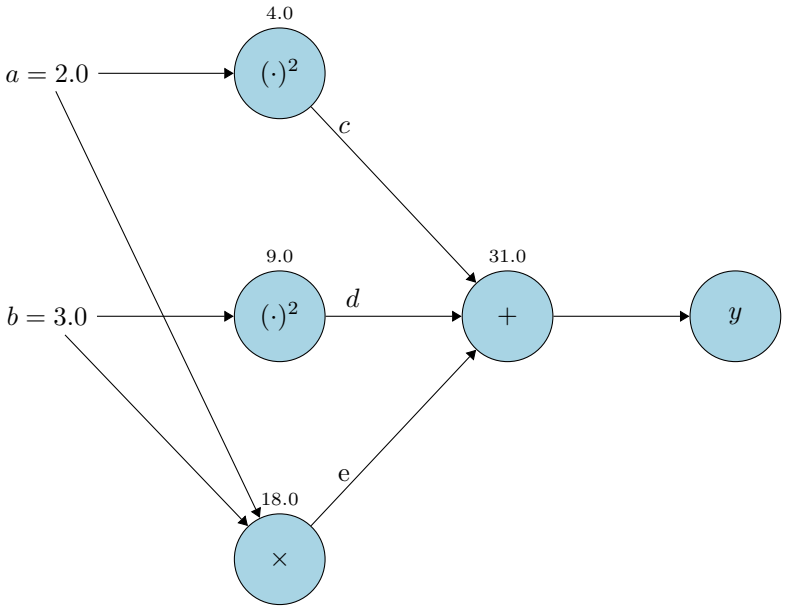
$$b = 3.0$$

$$c = a^2$$

$$d = b^2$$

$$e = 3ab$$

$$y = c + e + d$$



In reverse mode automatic differentiation, the forward pass involves evaluating the function and recording all intermediate values and dependencies without computing any derivatives. This creates a computational graph with nodes for each variable connected by edges representing dependencies, as depicted above.

After the forward pass completes, the reverse pass begins by calculating the "Adjoint" values. The Adjoint represents the partial derivative of the final output with respect to that intermediate variable. It is computed recursively starting from the output node and working backwards through the graph.

$$\bar{v}_i = \frac{\partial f}{\partial v_i} = \sum_{j\text{: child of } i} \bar{v}_j \frac{\partial v_j}{\partial v_i} \quad \textbf{Adjoint}$$

Specifically, for each node, the Adjoint is calculated by taking the sum of the Adjoints of all the node's children multiplied by the derivative of the child with respect to the current node. This effectively chains together the partial derivatives using the chain rule of calculus. So the Adjoint values propagate backwards from the output through the intermediate variables.

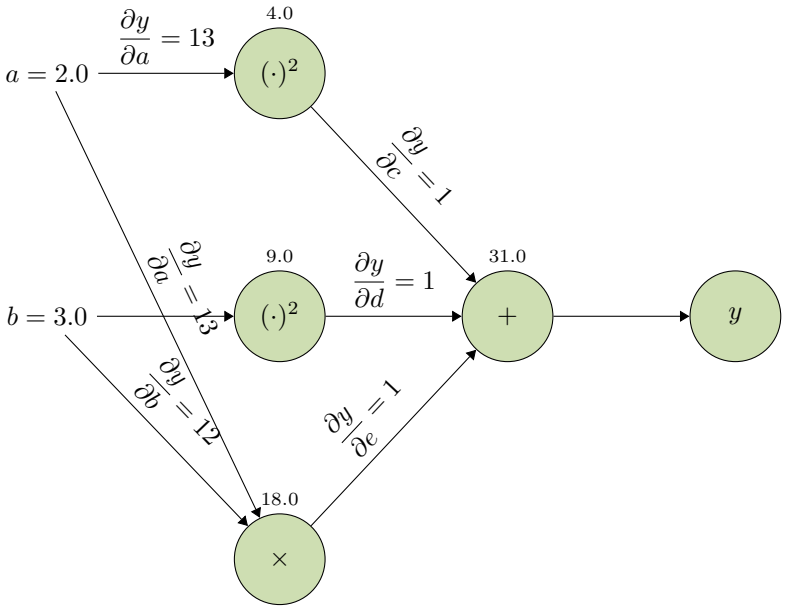


Figure 1: Computational Graph [2]

Once the Adjoints are computed for every node, they represent the partial derivatives of the final output with respect to each intermediate variable. The gradient can then be used for neural network optimization using gradient descent. Reverse mode approach is efficient for cases where the number of inputs is large compared to the number of outputs, this is the case for most supervised deep learning tasks (e.g. labels of an image have OOM smaller dimensions than the image it predicts) Reverse mode autodiff builds up a computational graph through the forward pass, then traverses it backwards to calculate gradients using the Adjoint, avoiding redundant derivative calculations. This allows efficient computation of gradients through long sequences of mathematical operations. This is how the library will calculate the gradient of the parameters of the network (with respect to loss etc) efficiently.

Neural Network Optimization Algorithms

Implementing both SGD and Adam optimization algorithms provides a solid foundation with 2 main benefits. SGD's simplicity with minimal hyperparameters accelerates initial testing during development. Its low memory requirements also allows scaling up network sizes. Meanwhile, Adam offers nice properties like fast initial convergence on some problems. Having both available gives users flexibility to choose the right optimizer for their needs and enables research into the tradeoffs. SGD serves as a simple, default option, while Adam provides an alternative for when its better convergence speed is needed.

Stochastic Gradient Descent

Stochastic gradient descent (SGD) is a basic neural network optimization algorithm used to minimize an objective function. It addresses the computational inefficiency of traditional gradient descent methods when dealing with large datasets hence making it useful for training neural networks. SGD processes training data in small batches instead of the entire dataset at once (batch training). Instead of using the entire dataset for each iteration, a small random batch is selected to calculate the gradient and update the model parameters. This random selection from the dataset introduces randomness into the optimization process, hence the term

"stochastic" in stochastic gradient descent. SGD can converge faster than batch training because it performs updates more frequently. The learning rate is a hyperparameter which specifies the size of the step taken by gradient descent. The steps taken by the SGD algorithm [3]:

1. Randomly shuffle the data set of size m
2. Select a learning rate α
3. Generate initial random parameter values (using Xavier initialization) θ as the starting point
4. Compute the gradient vector with respect to a random single training example or small batch $x^j, y^j = \nabla_\theta J(\theta; x^j; y^j)$
5. Update all parameters θ, compute $\theta_{i+1} = \theta_i - \alpha \times \nabla_\theta J(\theta; x^j; y^j)$
6. Repeat Step 4 and 5 until a local minimum is reached

Adaptive Moment Estimation (ADAM)

The Adam optimizer is a combination of the best properties of the Momentum (with SGD) and RMSProp algorithms. Momentum helps accelerate gradients in the right direction and dampens oscillations. RMSprop divides the learning rate by a running average of the magnitudes of recent gradients. Adam is relatively easy to configure where the default configuration parameters do well on most problems. The Adam optimizer calculates a running average of the gradients and the squared gradients for each parameter in the model. It then uses these averages to calculate the update for each parameter during training. It is an adaptive learning rate algorithm designed to improve training speeds in deep neural networks and reach convergence quickly. The steps taken by the Adam Algorithm [4]:

1. Select hyperparameters: learning rate α, 1st moment decay β_1, 2nd moment decay β_2, and small constant ϵ
2. Calculate the gradient of the loss function:

$$\frac{\partial \mathcal{J}}{\partial W^{[l]}}$$

3. Update exponential moving average of past gradients (1st moment estimate):

$$v_{dW^{[l]}} = \beta_1 v_{dW^{[l]}} + (1 - \beta_1) \frac{\partial \mathcal{J}}{\partial W^{[l]}}$$

4. Update exponential moving average of squared past gradients (2nd moment estimate):

$$s_{dW^{[l]}} = \beta_2 s_{dW^{[l]}} + (1 - \beta_2) (\frac{\partial \mathcal{J}}{\partial W^{[l]}})^2$$

5. Bias correct the 1st moment exponential moving average:

$$v_{dW^{[l]}}^{\text{corrected}} = \frac{v_{dW^{[l]}}}{1 - (\beta_1)^t}$$

6. Bias correct the 2nd moment exponential moving average:

$$s_{dW^{[l]}}^{\text{corrected}} = \frac{s_{dW^{[l]}}}{1 - (\beta_2)^t}$$

7. Update the weights using the bias corrected 1st and 2nd moment estimates:

$$W^{[l]} = W^{[l]} - \alpha \frac{v_{dW^{[l]}}^{\text{corrected}}}{\sqrt{s_{dW^{[l]}}^{\text{corrected}} + \varepsilon}}$$

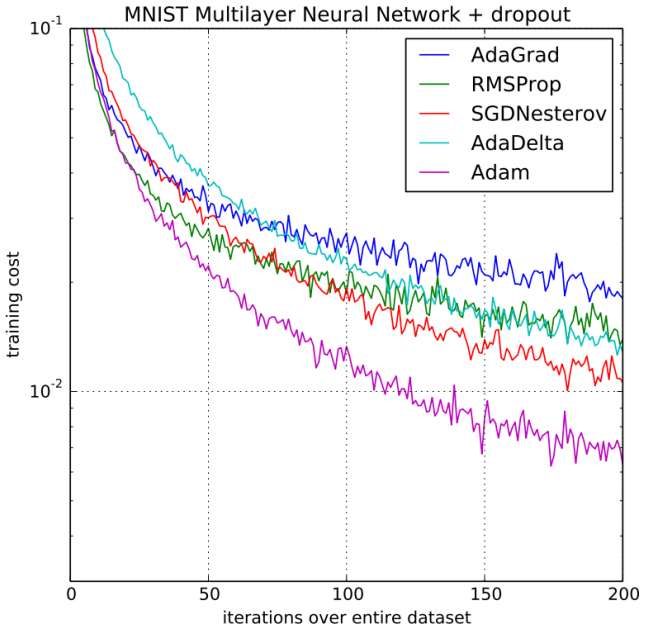


Figure 2: Adam performance compared to other optimizers [5]

2D Convolutions

One of the requirements which was understood from the interview Convolutions are a key component of convolutional neural networks (CNNs). In two dimensions, a convolution operates on an input image I and a kernel (filter) K to produce an output image O.

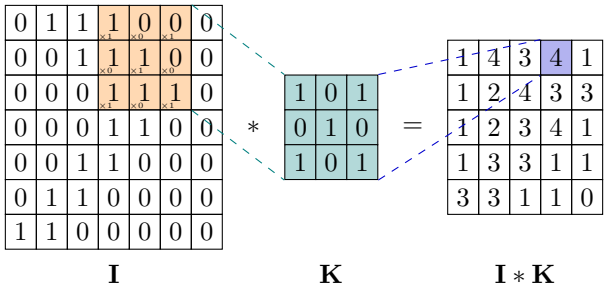
Mathematical Definition The 2D convolution operation is defined mathematically as:

$$O(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)$$

Where:

- I is the input image
- K is the kernel
- O is the output image
- i, j index the output image coordinates
- m, n index the kernel coordinates

Intuitively, the kernel slides across the image, computing the inner product between the kernel and local regions of input. This results in the output image.



Key Parameters Key parameters that define a 2D convolution include:

- Kernel size (height and width)
- Stride (vertical and horizontal offset between convolutions)
- Padding (pixels added to borders of input image)
- Input/output channels (filters applied)

Modifying these parameters changes the dimensionality of the output and spatial arrangement of outputs.

Implementations Efficient implementations utilize:

- Striding and padding to construct output
- Matrix multiplication to express computation
- GPU parallelization to apply filters simultaneously

which provide significant performance gains.

Regularization

Regularisation is a technique used to reduce overfitting in neural networks. Overfitting occurs when a model fits the training data too well but fails to generalise to new data. This happens when the network learns the noise in the training data in addition to the underlying patterns. Regularization penalizes complex models, thereby favoring simpler models that don't overfit. This improves the generalisation of the network. Some common regularization techniques are:

- **L1 regularisation**: Adds a penalty equal to the absolute value of the weights to the loss function. This forces sparse solutions by shrinking some weights to exactly 0.

$$\text{Loss} = \text{Loss} + \lambda \sum_{i=1}^{n} |w_i|$$

  Where λ controls the regularization strength.

- **L2 regularisation**: Adds a penalty equal to the square of the weights to the loss function. This causes the weights to prefer smaller non-zero values rather than exact zeros like L1. Also known as weight decay.

$$\text{Loss} = \text{Loss} + \lambda \sum_{i=1}^{n} w_i^2$$

Xavier Initialization

Xavier initialization is a method for setting the starting values for the weights in a neural network. The goal is to prevent vanishing or exploding gradients during training. It works by making sure that the signal that passes through each layer stays at a similar level. Here's how:

1. The initial weights $\mathbf{W}$ are randomly drawn from a distribution with zero mean.
2. The variance of the distribution is set to $\frac{1}{n_{\text{in}}}$.
3. n_{in} is the number of inputs going into that layer.

For example, say a layer has 300 input units and 100 output units. To initialise the weights $\mathbf{W}$ for that layer:

1. Set the mean of the distribution to 0.
2. Set the variance to $\frac{1}{300}$. Since there are 300 inputs, $n_{\text{in}} = 300$.
3. Randomly draw values for $\mathbf{W}$ from this distribution.

This results in the output activations having about the same variance as the inputs.

So Xavier initialization carefully scales the initial random weights based on the number of inputs. This helps to keep the signals and gradients flowing properly during training.

The Dataset Loader

The Dataset Loader (DataLoader) is responsible for transforming and batching a dataset in preparation for training. The DataLoader implemented in this library utilises two key algorithms:

Batching Algorithm This splits the dataset into batches of predefined size by striding through data examples sequentially. If the last batch ends up smaller than the batch size, it gets included as well. This allows batch iteration of the data set during training.

Shuffle Algorithm To introduce randomness into the data order, implementing the Fisher-Yates shuffle algorithm provides an efficient way to reorder examples. This iterates backward through the dataset, swapping each element with a randomly chosen preceding element. Doing this in-place rearranges the order effectively.

1.5 Objectives of the Deep Learning Library

Challenge objectives are written in *italics*

Automatic Differentiation

1. Tensor object can store n-dimensional values for parameters and gradients
2. Develop gradient functions for basic operations such as addition, subtraction, and matrix multiplication, as well as for common Deep Learning operations like logsoftmax, sigmoid, and ReLU.
3. Implement a backward function that computes the gradient for each intermediate variable in the computational graph. This function should be recursive to simplify understanding and usage.
4. Automatic Differentiation Testing
   - 4.1 Implement unit tests to validate correctness of automatic differentiation gradients.
   - 4.2 Compare the gradients computed by automatic differentiation with the gradients of PyTorch for a set of basic operations.
   - 4.3 Ensure absolute difference between automatic differentiation and PyTorch gradients is small (below threshold).

Easy-to-use Neural Network Functionality

1. Create a NeuralNet class which can be used to construct neural networks from individual layers and operations.
2. Create a class for Layers, which will be used to define a type of layer in the NeuralNet class. Implement common neural network layers (e.g. Linear Layers, 2D Convolutions Layers)
    2.1 Fully-connected layer (PyTorch Linear Layer)
    2.2 2D Convolution
    2.3 Normalization Layers
3. Activation Functions
    3.1 Include activation functions e.g. ReLU, sigmoid, softmax etc.
4. Optimizers for neural network training
    4.1 Stochastic Gradient Descent
    4.2 *Adaptive Moment Estimation (Adam)*
5. Layers should be initialized with random weights
    5.1 Xavier Initialization
6. Implement Regularization Techniques
    6.1 L1 Regularization
    6.2 L2 Regularization
    6.3 Layer Normalization
7. Model saving to disk and loading from files. Use an appropriate method to export model weights.
8. *GPU support for neural network training*

Data Utilities

1. *Ability to easily load large datasets into the Tensor format using external libraries for downloading the dataset (e.g. HF datasets?)*
2. *A dataloader which can split data into batches for training*
3. *Datasets can be shuffled*

2 Documented Design

2.1 Structuring a Python Library

```
PyPI_package_root
|   mnist.py
|   pyproject.toml
|   pytest.ini
|   README.md
|   test.py
|
\---deepdive
        __init__.py
        nn.py
        tensor.py
        utils.py
```

As this project will be compiled into a Python wheel for distribution through PyPI, following best practices for packaging in its structure is essential. This will enable building, installing, and using the library as any other Python package.

At the top level, a pyproject.toml file will define metadata and dependencies for the project. Inside a Python package initialized with a __init__.py file will reside the source code.

```toml
[build-system]
requires = ["setuptools", "wheel"]

[project]
name = "deepdive-ml"
version = "1.0.0"
description = "A deep learning library from scratch in python"
authors = [
{name = "Minhal Valiya Peedikakkal", email = "myemail@example.com"},
]
classifiers = [
"Intended Audience :: Developers",
"Programming Language :: Python :: 3",
"Programming Language :: Python :: 3.6",
"Programming Language :: Python :: 3.7",
"Programming Language :: Python :: 3.8",

]
dependencies = [
"numpy",
]
```

Listing 1: pyproject.toml

```python

# Import necessary modules or classes here
from .utils import check_cupy

# Define any global variables or constants here
CUPY_AVAILABLE = check_cupy()

# Optionally, include any initialisation code here
```

Listing 2: Example __init__.py file

The __init__.py file denotes a Python package directory. It can contain initialization code that gets executed upon import to set up the package environment. Key uses include:

- Importing key modules/classes for the package interface
- Defining global constants/variables
- Initializing the package by running setup functions

By defining what symbols are exposed in __init__.py, it controls the public API when users import the package. This initialization also helps set up any state needed before use.

The example project repository provided by the maintainers of PyPI (Python Packaging Authority) helped in making design choices in the project structure [6]

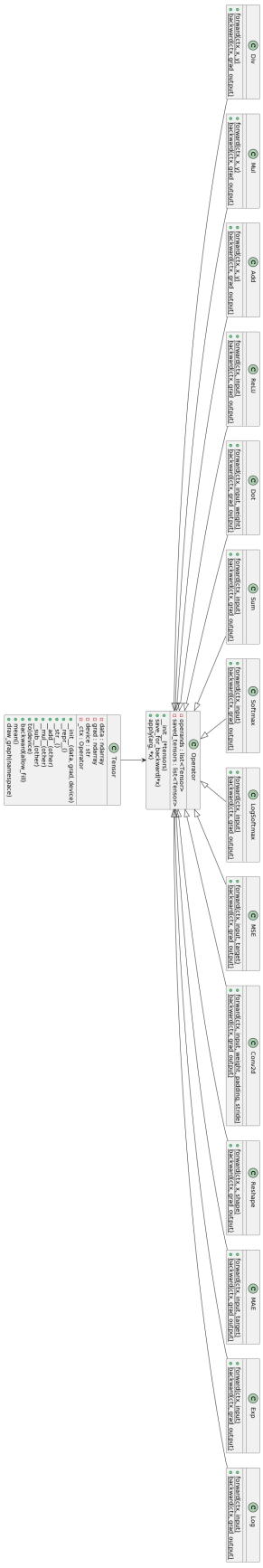


Figure 3: Tensor Class

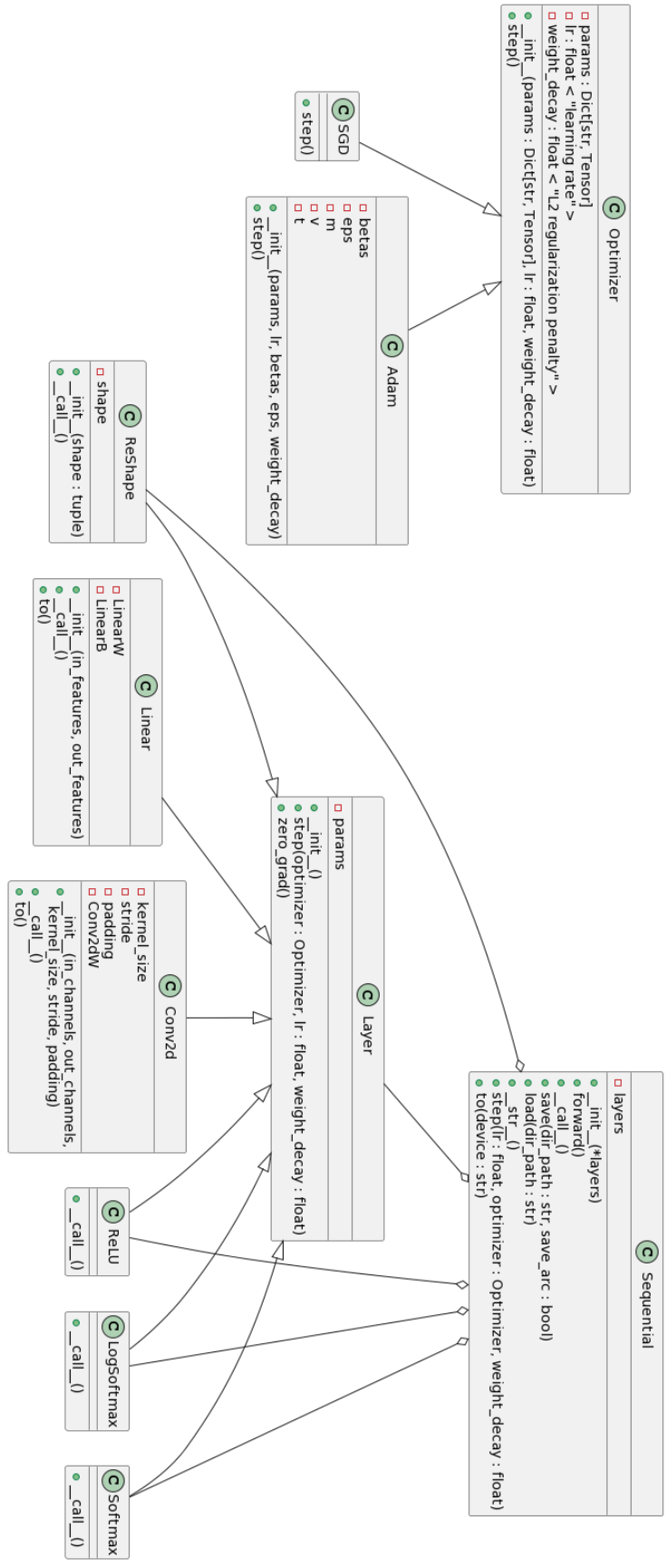


Figure 4: Neural Network Classes

2.2 Library Documentation

deepdive

Minhal Valiya Peedikakkal

Dec 29, 2023

CONTENTS:

CHAPTER

ONE

DEEPDIVE PACKAGE

1.1 Submodules

1.2 deepdive.dash module

class deepdive.dash.**Dash**

Bases: object

1.3 deepdive.datautils module

class deepdive.datautils.**DataLoader**(*dataset: DatasetDict*, *batch_size: int = 1*, *shuffle: bool = False*)

Bases: object

DataLoader for HuggingFace Datasets

1.3.1 Attributes

dataset (DatasetDict): HuggingFace Dataset bs (int): Batch Size shuffle (bool): Whether to shuffle the dataset batches (list): List of batches batch_index (int): Current batch index

1.3.2 Example:

```
>>> from deepdive.datautils import DataLoader
>>> from datasets import load_dataset
>>> mnist = load_dataset('mnist')
>>> dataloader = DataLoader(mnist['train'], batch_size=32)
>>> for i, (input, targets) in enumerate(dataloader):
...     print(input.shape, targets.shape) # (32, 28, 28) (32,)
```

get_batches()

Returns a list of batches from self.dataset

shuffle()

Uses Fisher-Yattes Shuffle Algorithm to shuffle the dataset

1.4 deepdive.nn module

class deepdive.nn.**Adam**(*params*, *lr=0.001*, *betas=(0.9, 0.999)*, *eps=1e-08*)

Bases: *Optimizer*

Adaptive Moment Estimation optimizer

1.4.1 Attributes

params (list): list of parameters to optimize lr (float): learning rate betas (tuple): coefficients used for computing running averages of gradient and its square eps (float): term added to the denominator to improve numerical stability

step()

Performs a single optimization step by updating the parameters in the direction of the gradient with the learning rate. More information can be found in the Analysis algorithms research.

class deepdive.nn.**Conv2d**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*)

Bases: *Layer*

2D Convolutional Neural Network Layer

1.4.2 Attributes

in_channels (int): number of input channels out_channels (int): number of output channels kernel_size (int): size of the kernel stride (int): stride of the kernel padding (int): padding of the kernel params (dict): dictionary of parameters.

1.4.3 Notes

FOLLOW THIS RULE -> O = (W - K + 2P) / S + 1 Where: O = output height/length W = input height/length K = kernel size P = padding S = stride

class deepdive.nn.**Layer**

Bases: `object`

Base class for all layers

1.4.4 Attributes

params (dict): dictionary of parameters. The keys are the name of the layer followed by the paramter type (W, B) and the values are the parameter tensors.

1.4.5 Notes

Not all layers have both W and B parameters. For example, Conv2d layers only have W parameters.

step(*lr*, *optimizer*)

Calls the step function of the optimizer on the parameters of all the layers.

Parameters

- **lr** (*float*) – learning rate
- **optimizer** (Optimizer) – optimizer to use

class deepdive.nn.**Linear**(*in_features*, *out_features*)

Bases: *Layer*

Linear Neural Network Layer

1.4.6 Attributes

in_features (int): number of input features out_features (int): number of output features

to(*device*)

Moves the parameters of the layer to the specified device.

Parameters

device (*str (either "cpu" or "cuda")*) – device to move the parameters to

class deepdive.nn.**LogSoftmax**

Bases: object

Layer for applying the LogSoftmax activation function

class deepdive.nn.**Optimizer**(*params*, *lr=0.001*)

Bases: ABC

Base class for all optimizers

1.4.7 Attributes

params (list): list of parameters from the model lr (float): learning rate

abstract step()

Performs a single optimization step. Not implemented in base class. This function should be implemented in all subclasses.

class deepdive.nn.**ReLU**

Bases: object

Layer for applying the ReLU activation function

class deepdive.nn.**ReShape**(*shape: tuple*)

Bases: object

Layer for reshaping the input tensor to the specified shape

class deepdive.nn.**SGD**(*params*, *lr=0.001*)

Bases: *Optimizer*

Stochastic Gradient Descent optimizer

1.4.8 Attributes

params (list): list of parameters to optimize lr (float): learning rate

step()

Performs a single optimization step by updating the parameters in the direction of the gradient with the learning rate.

class deepdive.nn.**Sequential**(**layers*)

Bases: object

Sequential Neural Network Model. This class is used to combine multiple layers into a single model.

1.4.9 Attributes

layers (list): list of layers to combine

1.4.10 Example

```
>>> model = nn.Sequential(
...     nn.Conv2d(1, 4, 3, 1, 0),
...     nn.ReLU(),
...     nn.Conv2d(4, 8, 3, 1, 0),
...     nn.ReLU(),
...     nn.ReShape((-1, 8*24*24)),
...     nn.Linear(4608, 1024),
...     nn.ReLU(),
...     nn.Linear(1024, 10),
... )
>>> input = Tensor(np.random.randn(32, 28, 28))
>>> output = model.forward(input)
```

forward(*x*)

Performs a forward pass on the input tensor through all the layers and returns the output tensor.

Parameters
x (Tensor) – input tensor

Returns
output tensor

Return type
Tensor

save(*path*)

Saves the model parameters to the specified path.

Parameters
path (*str*) – path to save the model parameters to

step(*lr*, *optimizer*)

Calls the step function of the optimizer on the parameters of all the layers.

Parameters

- **lr** (*float*) – learning rate

- **optimizer** (Optimizer) – optimizer to use

to(*device*)

Moves the parameters of the model to the specified device.

Parameters
device (*str (either "cpu" or "cuda")*) – device to move the parameters to

Returns
self

Return type
Sequential

1.5 deepdive.tensor module

class deepdive.tensor.**Add**(**tensors*)

Bases: *Operator*

The Add class implements the addition operation for tensors and saves the context of the operation.

Note
The addition operation is defined as $f(x, y) = x + y$.

1.5.1 Example

```
>>> x = Tensor([1, 2, 3])
>>> y = Tensor([4, 5, 6])
>>> print(x.add(y))  # prints: [5, 7, 9]
```

static backward(*ctx*, *grad_output*)

Computes the backward pass of the addition operation.

Parameters

- **ctx** (Operator) – The context of the operation.
- **grad_output** (Tensor) – The gradient of the loss with respect to the current layer's output. Used to compute gradients for the layer's inputs (if needed) and weights.

Returns
The gradients of the addition operation.

Return type
tuple of *Tensor*

static forward(*ctx*, *x*, *y*)

Computes the forward pass of the addition operation and saves the context of the operation.

Parameters

- **ctx** (Operator) – The context of the operation.
- **x** (Tensor) – The first operand of the addition.
- **y** (Tensor) – The second operand of the addition.

Returns
The result of the addition.

Return type
Tensor

class deepdive.tensor.**Conv2d**(**tensors*)

Bases: *Operator*

Implements the 2D convolution operation for tensors.

This class is a subclass of Operator and defines the forward and backward methods for the 2D convolution operation.

Note
The 2D convolution operation is defined as $f(x) = \sum_m \sum_n x[m, n] * w[m, n]$.

1.5.2 Example

```
>>> input = Tensor(np.random.rand(1, 3, 32, 32))
>>> weight = Tensor(np.random.rand(16, 3, 5, 5))
>>> padding = 2
>>> stride = 1
>>> out = input.conv2d(weight, padding, stride)
>>> print(out.shape)  # Example output: (1, 16, 32, 32)
```

static backward(*ctx*, *grad_output*)

Computes the backward pass of the 2D convolution operation.

Parameters

- **ctx** (Operator) – The context of the operation.
- **grad_output** (Tensor) – The gradient of the loss with respect to the current layer's output. Used to compute gradients for the layer's inputs (if needed) and weights.

Returns
The gradients of the 2D convolution operation.

Return type
tuple of *Tensor*

static forward(*ctx*, *input*, *weight*, *padding*, *stride*)

Computes the forward pass of the 2D convolution operation and saves the context of the operation.

Parameters

- **ctx** (Operator) – The context of the operation.
- **input** (Tensor) – The input tensor.
- **weight** (Tensor) – The weight tensor.
- **padding** (*int*) – The padding of the operation.
- **stride** (*int*) – The stride of the operation.

Returns
The result of the 2D convolution operation.

Return type
Tensor

class deepdive.tensor.**Dot**(**tensors*)

Bases: *Operator*

The Dot class implements the dot product operation for tensors and saves the context of the operation.

Note
The dot product operation is defined as $f(x, y) = x \cdot y$.

1.5.3 Example

```
>>> x = Tensor([1, 2, 3])
>>> y = Tensor([4, 5, 6])
>>> print(x.dot(y))  # prints: 32
```

static backward(*ctx*, *grad_output*)

Computes the backward pass of the dot product operation.

Parameters

- **ctx** (Operator) – The context of the operation.
- **grad_output** (Tensor) – The gradient of the loss with respect to the current layer's output. Used to compute gradients for the layer's inputs (if needed) and weights.

Returns
The gradients of the dot product operation.

Return type
tuple of *Tensor*

static forward(*ctx*, *input*, *weight*)

Computes the forward pass of the dot product operation and saves the context of the operation.

Parameters

- **ctx** (Operator) – The context of the operation.
- **input** (Tensor) – The first operand of the dot product.
- **weight** (Tensor) – The second operand of the dot product.

Returns
The result of the dot product.

Return type
Tensor

class deepdive.tensor.**LogSoftmax**(**tensors*)

Bases: *Operator*

The LogSoftmax class implements the log softmax operation for tensors and saves the context of the operation.

note
The log softmax operation is defined as :math:`f(x) = log(

rac{e^{x_i}}{sum_{j=1}^n e^{x_j}})`.

```
>>> x = Tensor([1, 2, 3])
>>> print(x.logsoftmax())
```

static backward(*ctx*, *grad_output*)

Computes the backward pass of the log softmax operation.

Parameters

- **ctx** (Operator) – The context of the operation.
- **grad_output** (Tensor) – The gradient of the loss with respect to the current layer's output. Used to compute gradients for the layer's inputs (if needed) and weights.

Returns
The gradients of the log softmax operation.

Return type
tuple of *Tensor*

static forward(*ctx*, *input*)

Computes the forward pass of the log softmax operation.

Parameters

- **ctx** (Operator) – The context of the operation.
- **input** (Tensor) – The input tensor.

Returns
The result of the log softmax operation.

Return type
Tensor

class deepdive.tensor.**MAE**(**tensors*)

Bases: *Operator*

Implements the Mean Absolute Error (MAE) operation for tensors.

This class is a subclass of Operator and defines the forward and backward methods for computing the MAE between two tensors.

1.5.4 Example

```
>>> predictions = Tensor(np.array([3.0, -0.5, 2, 7]))
>>> targets = Tensor(np.array([2.5, 0.0, 2, 8]))
>>> error = predictions.mae(targets)
>>> print(error)  # Example output: Tensor([0.5, 0.5, 0, 1])
```

static backward(*ctx*, *grad_output*)

Computes the backward pass of the MAE operation.

Parameters

- **ctx** (Operator) – The context of the operation.
- **grad_output** (Tensor) – The gradient of the loss with respect to the current layer's output. Used to compute gradients for the layer's inputs (if needed) and weights.

Returns
The gradients of the MAE operation.

Return type
tuple of *Tensor*

static forward(*ctx*, *input*, *target*)

Computes the forward pass of the MAE operation and saves the context of the operation.

Parameters

- **ctx** (Operator) – The context of the operation.
- **input** (Tensor) – The first operand of the MAE operation.
- **target** (Tensor) – The second operand of the MAE operation.

Returns
The result of the MAE operation.

Return type
Tensor

class deepdive.tensor.**MSE**(**tensors*)

Bases: *Operator*

The MSE class implements the mean squared error operation for tensors and saves the context of the operation. Work In Progress

static backward(*ctx*, *grad_output*)

static forward(*ctx*, *input*, *target*)

class deepdive.tensor.**Mul**(**tensors*)

Bases: *Operator*

The Mul class implements the multiplication operation for tensors.

Note
The multiplication operation is defined as $f(x, y) = x * y$.

1.5.5 Example

```
>>> x = Tensor([1, 2, 3])
>>> y = Tensor([4, 5, 6])
>>> print(x.mul(y))  # prints: [4, 10, 18]
```

static backward(*ctx*, *grad_output*)

Computes the backward pass of the multiplication operation.

Parameters

- **ctx** (Operator) – The context of the operation.
- **grad_output** (Tensor) – The gradient of the loss with respect to the current layer's output. Used to compute gradients for the layer's inputs (if needed) and weights.

Returns
The gradients of the multiplication operation.

Return type
tuple of *Tensor*

static forward(*ctx*, *x*, *y*)

Computes the forward pass of the multiplication operation and saves the context of the operation.

Parameters

- **ctx** (Operator) – The context of the operation.
- **x** (Tensor) – The first operand of the multiplication.
- **y** (Tensor) – The second operand of the multiplication.

Returns
The result of the multiplication.

Return type
Tensor

class deepdive.tensor.**Operator**(**tensors*)

Bases: object

The Operator class is a base class that provides the functionality to store the context of each operation for backward propagation.

This class is intended to be subclassed by classes that implement specific tensor operations. The context stored by an Operator instance includes the inputs and outputs of the operation, which are used during the backward pass to compute gradients.

Note
This class does not implement the actual operations. Subclasses should override the *forward* and *backward* methods to implement the forward and backward passes of the operation.

1.5.6 Attributes

operands
[list of Tensor] The operands of the operation.

saved_tensors
[list of Tensor] The tensors saved by the *save_for_backward* method.

apply(*arg*, **x*)

Apply the operator.

This method applies the operator by:

1. Calling arg to generate a context
2. Calling arg.forward with the context and input tensors to compute

   the output

3. Creating a new Tensor instance to hold the result
4. Saving the context to the new Tensor instance for later use in

   backpropagation

Parameters

- **arg** (*object*) – The object that will be used to generate the context.
- **x** (Tensor) – The input tensors.

Returns
A new tensor containing the result of the operation and with the context saved in the *_ctx* attribute.

Return type
Tensor

save_for_backward(**x*)

Saves the input tensors for later use in the backward pass. This is done by extending the *saved_tensors* list.

Parameters
x (Tensor) – The tensors to save.

class deepdive.tensor.**ReLU**(**tensors*)

Bases: *Operator*

The ReLU class implements the ReLU operation for tensors and saves the context of the operation.

Note
The ReLU operation is defined as $f(x) = max(0, x)$.

1.5.7 Example

```
>>> x = Tensor([-1, 2, -3])
>>> print(x.relu())  # prints: [0, 2, 0]
```

static backward(*ctx*, *grad_output*)

Computes the backward pass of the ReLU operation.

Parameters

- **ctx** (Operator) – The context of the operation.
- **grad_output** (Tensor) – The gradient of the loss with respect to the current layer's output. Used to compute gradients for the layer's inputs (if needed) and weights.

Returns
The gradients of the ReLU operation.

Return type
tuple of *Tensor*

static forward(*ctx*, *input*)

Computes the forward pass of the ReLU operation and saves the context of the operation.

Parameters

- **ctx** (Operator) – The context of the operation.
- **input** (Tensor) – The input tensor.

Returns
The result of the ReLU operation.

Return type
Tensor

class deepdive.tensor.**Reshape**(**tensors*)

Bases: *Operator*

Implements the reshape operation for tensors.

This class is a subclass of Operator and defines the forward and backward methods for reshaping a tensor.

1.5.8 Example

```
>>> x = Tensor(np.array([[1, 2, 3], [4, 5, 6]]))
>>> y = x.reshape((3, 2))
>>> print(y)  # Example output: Tensor([[1, 2], [3, 4], [5, 6]])
```

static backward(*ctx*, *grad_output*)

Computes the backward pass of the reshape operation.

Parameters

- **ctx** (Operator) – The context of the operation.
- **grad_output** (Tensor) – The gradient of the loss with respect to the current layer's output. Used to compute gradients for the layer's inputs (if needed) and weights.

Returns
The gradients of the reshape operation.

Return type
tuple of *Tensor*

static forward(*ctx*, *x*, *shape: tuple*)

Computes the forward pass of the reshape operation and saves the context of the operation.

Parameters

- **ctx** (Operator) – The context of the operation.
- **x** (Tensor) – The input tensor.
- **shape** (*tuple*) – Reshape the tensor x to this shape.

Returns
The result of the reshape operation.

Return type
Tensor

class deepdive.tensor.**Sum**(**tensors*)

Bases: *Operator*

The Sum class implements the sum operation for tensors and saves the context of the operation.

Note
The sum operation is defined as $f(x) = \sum_{i=1}^{n} x_i$.

1.5.9 Example

```
>>> x = Tensor([1, 2, 3])
>>> print(x.sum())  # prints: 6
```

static backward(*ctx*, *grad_output*)

Computes the backward pass of the sum operation.

Parameters

- **ctx** (Operator) – The context of the operation.
- **grad_output** (Tensor) – The gradient of the loss with respect to the current layer's output. Used to compute gradients for the layer's inputs (if needed) and weights.

Returns
The gradients of the sum operation.

Return type
tuple of *Tensor*

static forward(*ctx*, *input*)

Computes the forward pass of the sum operation and saves the context of the operation.

Parameters

- **ctx** (Operator) – The context of the operation.
- **input** (Tensor) – The input tensor.

Returns
The result of the sum operation.

Return type
Tensor

class deepdive.tensor.**Tensor**(*data*, *grad=None*, *device=None*)

Bases: object

A class used to represent a Tensor.

This class provides methods for tensor operations and also keeps track of gradients for backpropagation.

1.5.10 Attributes

data
[numpy.ndarray or cupy.ndarray] The actual data of the tensor.

grad
[numpy.ndarray or cupy.ndarray, optional] The gradient of the tensor, used in backpropagation.

device
[str, optional] The device where the tensor is stored ("cuda" for GPU, None for CPU).

_ctx
[Context, optional] The context related to this tensor, used in backpropagation.

add(**x*)

backward(*allow_fill=True*)

Performs backpropagation from this tensor through the computation graph.

If this tensor is the result of an operation, this method will compute the gradient of this tensor with respect to the inputs of that operation, and recursively call backward on those inputs.

Parameters
allow_fill (*bool, optional*) – If True and if this tensor's grad attribute is None, a new gradient tensor will be created with the same shape as this tensor's data, filled with ones. This is useful for starting backpropagation from a scalar loss tensor.

Example

```
>>> x = Tensor([1, 2, 3])
>>> y = Tensor([4, 5, 6])
>>> z = x + y
>>> z.backward()
>>> print(x.grad)  # prints: array([1., 1., 1.])
```

conv2d(**x*)

dot(**x*)

logsoftmax(**x*)

mae(**x*)

mean()

Computes and returns the mean of the tensor.

The mean is computed by summing all elements in the tensor and dividing by the number of elements.

Returns

A new tensor containing the mean of the data of this tensor.

Return type

Tensor

Example

```
>>> x = Tensor([1, 2, 3, 4])
>>> print(x.mean())  # prints: 2.5
```

mse(**x*)

mul(**x*)

relu(**x*)

reshape(**x*)

sum(**x*)

to(*device*)

Moves the tensor to the specified device.

Parameters

device

[str] The name of the device to which the tensor should be moved. This should be “cuda” for GPU or “cpu” for CPU.

Returns

Tensor
A new tensor with the same data as this tensor, but located on the specified device.

Example

```
>>> x = Tensor([1, 2, 3])
>>> x = x.to("cuda")  # moves x to GPU
```

deepdive.tensor.**register**(*name*, *fxn*)

Adds a method to the Tensor class. The new method is a partial application of the *apply* method of the *fxn* object.

Parameters

- **name** (*str*) – The name of the new method.
- **fxn** (*object*) – The object whose *apply* method will be partially applied.

1.5.11 Example

If *fxn* is an instance of a class *Mul* with an *apply* method that multiplies its arguments:

```
register('mul', Mul())
t = Tensor([2, 3, 4])
t.mul(5)  # This will output a tensor with data [10, 15, 20]
```

1.6 deepdive.utils module

deepdive.utils.**check_cupy**() → bool

deepdive.utils.**import_cupy_else_numpy**()

Tries to import the cupy module, if not installed, imports numpy.

1.6.1 Returns

module
The imported module, either cupy or numpy.

1.6.2 Raises

ImportError
If neither cupy nor numpy can be imported.

1.7 Module contents

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

O

R

S

T

2.3 Algorithm Design

Backpropagation Algorithm

The backpropagation algorithm is the backbone of every Deep Learning library, enabling the computation of gradients for Tensors and aiding the optimization of neural networks.

Upon creation, each instance of the `Tensor` class is assigned an internal attribute, `_ctx`. This attribute, an instance of the `Operator` class, retains the information necessary to construct a computational graph. A tensor can then utilize its `_ctx` to identify its parent tensors, i.e., any nodes in the computational graph that precede it.

Consider the following example:

```
a = Tensor([1, 2, 3])
b = Tensor([4, 5, 6])
c = a.add(b)
print(f"c: {c.data} Operands: {c._ctx.operands}")

Output
c: [5 7 9] Operands: (Tensor([1 2 3]), Tensor([4 5 6]))
```

In this case, the `_ctx` of Tensor `c` contains an attribute `operands` which stores a tuple of Tensors that were involved in its creation. This feature is vital for backpropagating through the computational graph.

Invoking the `backward` method on a Tensor initiates a depth-first search (DFS) through the computational graph, recursively applying the following steps:

1. Assigns the `grad` attribute with an ndarray of 1s, identical in shape to the ndarray stored in the `data` attribute. This only occurs when `backward` is called for the first time.
2. Calculates the gradients with respect to inputs by invoking the `backward` method of the operator with the tensor's `_ctx` and the Tensor's `grad`.
3. Assigns the calculated gradient to each respective parent tensor in its `grad` attribute.
4. Calls `backward` on the parent tensors, repeating the above steps for each parent tensor until it reaches a Tensor without a `_ctx` attribute (i.e., no parent tensors).

One requirement of calling backward on a Tensor is that the Tensor must hold data which is scalar (This is the same in PyTorch). Continuing on from the last example, to find the gradient of the Tensor `c` with respect to its parent Tensors:

```
c.mean().backward()
print(a.grad, b.grad)

Output
[0.33333333 0.33333333 0.33333333] [0.33333333 0.33333333 0.33333333]
```

This is the full algorithm pseudocode

Algorithm 1 Backward function of Tensor class

procedure BACKWARD($allow\_fill$)
  if $self.\_ctx$ is None **then** ▷ Check if the tensor has a computation context
    return ▷ If not, return as no gradients need to be computed
  end if
  if $self.grad$ is None and $allow_fill$ **then**
    $self.grad \leftarrow$ ndarray of 1s with same shape as self.data
  end if
  $parent_grads \leftarrow self.\_ctx.backward(self.\_ctx, self.grad)$ ▷ Compute the gradients of the parent tensors using the backward method of the operation
  if $len(self.\_ctx.operands) == 1$ **then**
    $parent\_grads \leftarrow [parents\_grads]$ ▷ Ensure the gradients are in a list for consistent processing
  end if
  for each $tensor, gradient$ in $zip(self.\_ctx.operands, parent\_grads)$ **do** ▷ Iterate over each parent tensor and its corresponding gradient
    if $tensor.data.shape \neq gradient.shape$ **then**
      $gradient \leftarrow gradient.mean(axis = 0)$ ▷ Take mean of gradients over a axis if shapes mismatch
    end if
    if $tensor.grad$ is None **then**
      $tensor.grad \leftarrow gradient$ ▷ Assign the computed gradient to the parent tensor
    else
      $tensor.grad \leftarrow tensor.grad + gradient$ ▷ Accumulate gradients if tensor already has a gradient
    end if
    $tensor.backward(False)$ ▷ Recursively call backward on the parent tensors
  end for
end procedure

The provided library supports a wide range of operations, and new operations can be easily added by defining a new subclass of `Operator` and implementing the `forward` and `backward` methods. This modular design promotes code reusability and maintainability, making it relatively straightforward to extend the functionality of the Tensor class with new operators.

A requirement of the backward pass is that each operation needs to have a `backward` method defined, which computes the derivative of the operation with respect to its inputs. Here are the operators supported by the library and their derivatives:

- **Addition**:
  - Forward: $f(x, y) = x + y$
  - Backward: $\frac{\partial f}{\partial x} = 1, \frac{\partial f}{\partial y} = 1$
- **Multiplication (Hadamard Product)**:
  - Forward: $f(x, y) = x \cdot y$
  - Backward: $\frac{\partial f}{\partial x} = y, \frac{\partial f}{\partial y} = x$
- **ReLU**:
  - Forward: $f(x) = \max(0, x)$
  - Backward: $\frac{\partial f}{\partial x} = f(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$
- **Dot Product**:

- Forward: $f(x, y) = x \cdot y$
- Backward: $\frac{\partial f}{\partial x} = y^T$, $\frac{\partial f}{\partial y} = x$

- **Sum**:
  - Forward: $f(x) = \sum_{i=1}^{n} x_i$
  - Backward: $\frac{\partial f}{\partial x_i} = 1, \forall i$

- **LogSoftmax**:
  - Forward: $f(x) = \log\left(\frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}\right)$
  - Backward: $\frac{\partial f}{\partial x_i} = 1 - e^{x_i}$

- **Mean Squared Error (MSE)**:
  - Forward: $f(x, y) = (x - y)^2$
  - Backward: $\frac{\partial f}{\partial x} = 2(x - y)$, $\frac{\partial f}{\partial y} = -2(x - y)$

- **2D Convolution**:
  - Forward: $f(x, w) = \sum_m \sum_n x[m, n] \cdot w[m, n]$
  - Backward: The backward pass of a convolution operation is a complex process that involves 'flipping' the kernel and performing a full convolution. The exact derivative depends on the stride, padding, and other factors.

- **Division**:
  - Forward: $f(x, y) = x/y$
  - Backward: $\frac{\partial f}{\partial x} = 1/y$, $\frac{\partial f}{\partial y} = -x/y^2$

- **Mean Absolute Error (MAE)**:
  - Forward: $f(x, y) = |x - y|$
  - Backward: $\frac{\partial f}{\partial x} = \text{sign}(x - y)$, $\frac{\partial f}{\partial y} = -\text{sign}(x - y)$

- **Exponential**:
  - Forward: $f(x) = e^x$
  - Backward: $\frac{\partial f}{\partial x} = e^x$

- **Natural Logarithm**:
  - Forward: $f(x) = \log(x)$
  - Backward: $\frac{\partial f}{\partial x} = 1/x$

- **Reshape**:
  - Forward: $f(x, shape) = \text{reshape}(x, shape)$
  - Backward: $\frac{\partial f}{\partial x} = \text{reshape}(grad_output, original_shape)$

2D Convolutions

The 2D convolution operation is important for a deep learning library as it is used in CNN (Convolutional Neural Network) The forward pass of the Convolution Operation can be described as the pseudo code below

Algorithm 2 Forward Pass of Conv2D Operator

procedure CONV2DFORWARD($input, weight, padding, stride$)
  $N, C, W, H \leftarrow$ shape of $input$ ▷ N: batch size, C: number of channels, W: width, H: height
  $F, C, WW, HH \leftarrow$ shape of $weight$ ▷ F: number of filters, WW: filter width, HH: filter height
  assert $(H + 2 \times padding - HH) \mod stride = 0$ ▷ Check input height and width is compatible with the stride and padding
  assert $(W + 2 \times padding - WW) \mod stride = 0$
  $out\_h \leftarrow (H + 2 \times padding - HH) \div stride + 1$ ▷ Calculate the height and width of the output
  $out\_w \leftarrow (W + 2 \times padding - WW) \div stride + 1$
  $input\_pad \leftarrow$ pad $input$ with $padding$ on height and width
  $output \leftarrow$ initialize zero matrix of shape (N, F, out_h, out_w)
  for $n \leftarrow 0$ **to** $N - 1$ **do** ▷ Loop over each item in the batch
    for $f \leftarrow 0$ **to** $F - 1$ **do** ▷ Loop over each filter
      for $i \leftarrow 0$ **to** $out\_h - 1$ **do**
        for $j \leftarrow 0$ **to** $out\_w - 1$ **do**
          $input\_patch \leftarrow input\_pad[n, :, i \times stride : i \times stride + HH, j \times stride : j \times stride + WW]$ ▷ Extract a patch from the input
          $filter \leftarrow weight[f, :, :, :]$
          $output[n, f, i, j] \leftarrow \sum(input\_patch \times filter)$ ▷ Apply the filter to the input patch and store the result in the output
        end for
      end for
    end for
  end for
  return $output$
end procedure

Fisher-Yattes Shuffle Algorithm

A simple shuffling algorithm is needed to shuffle the batches in a dataset. This function will be used by the DataLoader. The shuffling algorithm that I will use can be described by the following pseudocode:

Algorithm 3 Fisher-Yates Shuffle

procedure FISHERYATESSHUFFLE(A) ▷ A is the array to shuffle
  for $i = \text{length}(A)$ **downto** 1 **do**
    $j \leftarrow \text{random}(1, i)$ ▷ Pick a random index from 1 to i
    $A[i], A[j] \leftarrow A[j], A[i]$ ▷ Swap elements at positions i and j
  end for
end procedure

Batching Algorithm

The DataLoader is used to take a dataset and split the data into batches of a user-defined size. I have developed an algorithm using modular arithmetic to create batches from the data and also the final batch can be of a size smaller than the batch size as the length of the dataset may not be a perfect multiple of the batch size

Algorithm 4 Get Batches

procedure GETBATCHES
  $num\_batches \leftarrow \lfloor \frac{len(dataset)}{bs} \rfloor$ ▷ Integer division
  $last\_bs \leftarrow len(dataset) \mod bs$ ▷ Modulo operation
  $batches \leftarrow []$
  for $i \in range(num\_batches)$ **do**
    $batches.append(dataset.select(range(i * bs, (i + 1) * bs)))$
  end for
  if $last\_bs \neq 0$ **then**
    $batches.append(dataset.select(range(num\_batches * bs, num\_batches * bs + last\_bs)))$
  end if
  return *batches*
end procedure

Softmax Jacobian

This function is used by backprop to calculate the derivatives of the output of a softmax functions w.r.t to its inputs The softmax function is defined as:

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}}$$

where $\mathbf{z}$ is the input vector of length N.

The Jacobian matrix of the softmax function is given by:

$$\text{Jacobian}(\text{softmax}(\mathbf{z}))_{ij} = \frac{\partial \text{softmax}(\mathbf{z})_i}{\partial \mathbf{z}_j} = \text{softmax}(\mathbf{z})_i \cdot (\delta_{ij} - \text{softmax}(\mathbf{z})_j)$$

Here, δ_{ij} is the Kronecker delta function, defined as:

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

Algorithm 5 Softmax Backward

procedure BACKWARD($ctx, grad\_output$)
  $output \leftarrow ctx.saved_tensors[0]$
  $d\_softmax \leftarrow output \odot (1 - output)$ ▷ Diagonal Jacobian / when i = j
  $jacobian \leftarrow -output[..., None] \otimes output[:, None, :]$ ▷ Off-diagonal Jacobian
  $jacobian[:, arange(output.shape[1]), arange(output.shape[1])] \leftarrow d\_softmax$ ▷ Insert diagonal
  $grad\_input \leftarrow grad\_output \times jacobian$ ▷ Matrix multiply
  return *grad_input*
end procedure

Modular Operator Design

The modular design of the library makes it straightforward to introduce new tensor operations by defining a new subclass of the `Operator` base class. This subclass encapsulates both the forward and backward computations for the new operation allowing code reusability and maintainability.

To add a new operator, one needs to create a new subclass of `Operator` and define the `forward` and `backward` static methods. The `forward` method computes the forward pass of the operation, while the `backward` method computes the backward pass, handling the gradients.

For example, consider the implementation of the `Mul` class, which implements the hadamard product operation:

```
class Mul(Operator):
    @staticmethod
    def forward(ctx, x, y):
        ctx.save_for_backward(x, y)
        return x*y

    @staticmethod
    def backward(ctx, grad_output):
        x, y = ctx.saved_tensors
        return y*grad_output, x*grad_output
```

In this example, the `forward` method computes the hadamard product of the input tensors `x` and `y`, while the `backward` method computes the gradients of the operation with respect to the inputs, using the saved tensors from the forward pass.

After defining the new operator class, it can be easily registered with the `Tensor` class using the `register` function:

```
register('mul', Mul)
```

This function adds a method named `mul` to the `Tensor` class, which is a partial application of the `apply` method of the `Mul` operator. As a result, any instance of the `Tensor` class will now have a `mul` method that applies the `Mul` operator to the input tensors.

The modular design and the use of the `register` function make it straightforward to extend the functionality of the `Tensor` class with new operators. New operations can be added without modifying the core implementation of the `Tensor` class, promoting code reusability and maintainability. Moreover, the use of static methods for `forward` and `backward` allows for efficient computation and memory management, as these methods do not require instantiation of the operator class.

Overall, the provided library demonstrates a well-structured and modular approach for adding tensor operations, making it relatively easy to extend the functionality of the `Tensor` class with new operators.

Debugging the Library

The provided library includes a useful `draw_graph` function that can aid in the debugging and visualization of computational graphs involving tensor operations. This function leverages the graphviz library to generate a visual representation of the computation graph, depicting the tensors and the operations performed on them.

The computational graph shown in the attached image illustrates the power of the `draw_graph` function. Each node in the graph represents a tensor, displaying its shape, gradient status, and a unique identifier. The edges between the nodes represent the operations applied to the tensors, with the operation type indicated by the label on the edge.

By visualizing the computational graph, developers can gain valuable insights into the flow of tensor operations and the dependencies between tensors. This can be particularly useful when debugging complex models or investigating issues related to backpropagation and gradient computation.

Some potential use cases for the `draw_graph` function include:

1. **Verifying Computational Graph Structure**: The visual representation of the computational graph allows developers to verify that the operations are being applied in the correct order and that the tensors are being transformed as expected.

2. **Identifying Gradient Flow Issues**: By examining the gradient status of tensors in the graph, developers can quickly identify tensors that may not be receiving gradients during backpropagation, which can help pinpoint potential issues in the model architecture or the implementation of specific operations.

3. **Tracing Tensor Shapes**: The graph displays the shape of each tensor, making it easier to trace the transformation of tensor shapes throughout the computational graph. This can be valuable when

debugging shape-related issues or ensuring that operations are compatible with the expected tensor shapes.

4. **Understanding Model Complexity**: The visual representation of the computational graph can provide insights into the complexity of a model, helping developers identify potential bottlenecks or areas for optimization.

5. **Debugging Gradient Computations**: By inspecting the gradients of tensors involved in specific operations, developers can verify the correctness of the gradient computations performed during back-propagation.

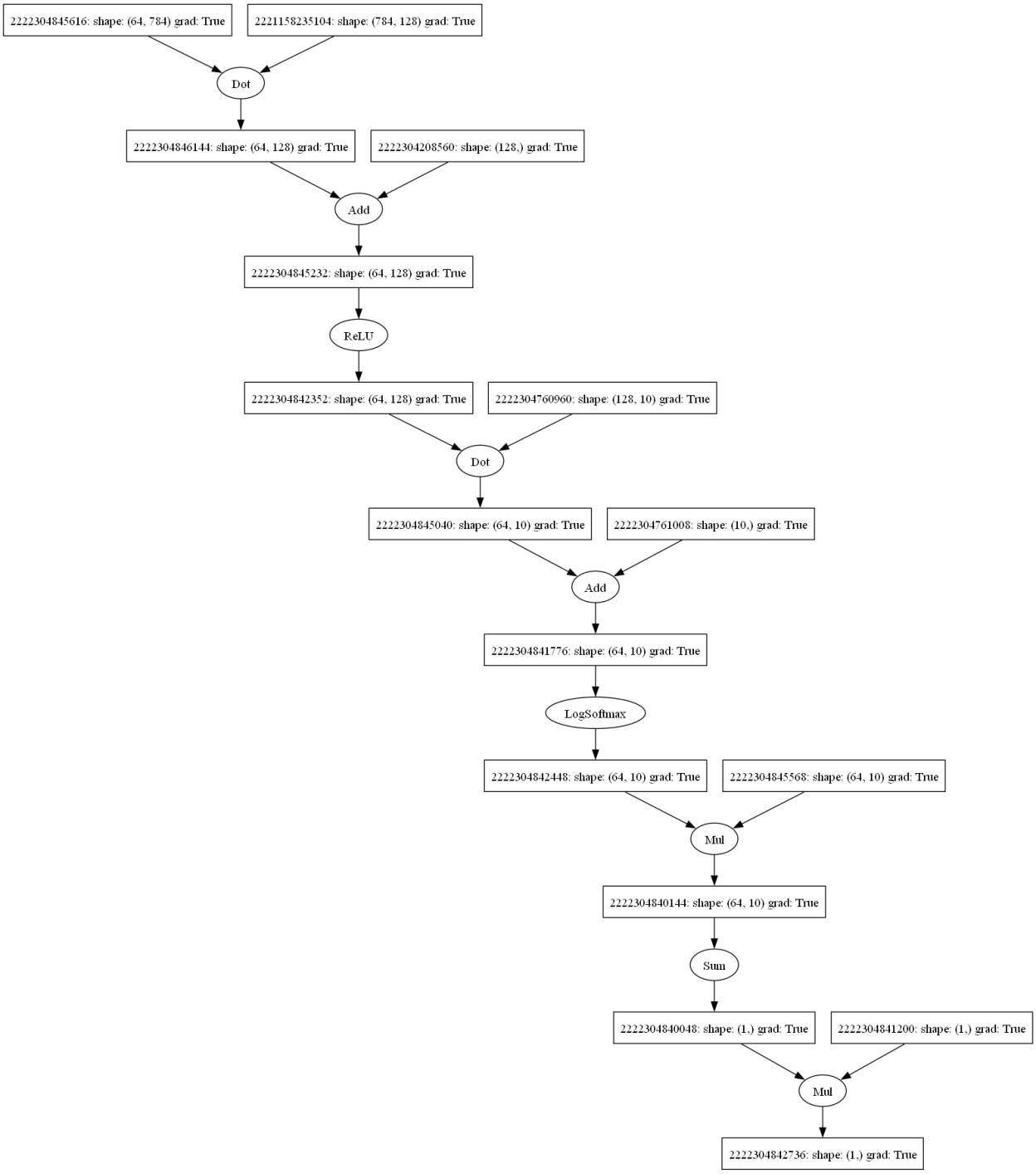


Figure 5: Convolutional Neural Network Computational Graph produced by draw_graph

3 Technical Solution

3.1 Advanced Coding Skills Pointer

Table 2: Advanced Coding Skills Pointer

Coding Skill	**Details**	**Page Number**
Object-oriented programming	Classes like `Tensor`, `Operator`, `Layer`, `Sequential` encapsulate related data and behaviors.	53, 55
Operator overloading	Overloaded methods like `__repr__`, `__str__` and operators like `*`, `+` for `Tensor` class.	53
Context managers	`Operator` class uses contexts and a registry to manage operations like `mul`, `add`, etc.	55
Functional Programming	Utilized partial function application and functions like `map`.	55, 71
Duck typing	Methods like `forward`, `backward` assume the presence of certain methods and attributes.	55, 53
Modular design	Components split into different modules and files like `tensor.py`, `nn.py`, etc.	-
Advanced NumPy usage	Methods like `conv2d`, `dot`, `softmax` show advanced NumPy functionalities.	58, 57
Type annotations	Type annotations for method inputs and outputs for type safety.	-
Advanced data structure usage	Model parameters in nested dicts, usage of tuples and lists.	61, 62, 63
Model loading and saving	Created a custom file format `.arc` and folder structure for saving and loading neural network models seamlessly	66

3.2 Code

deepdive/tensor.py

```python
from .utils import import_cupy_else_numpy
import numpy as np
import cupy
from functools import partialmethod
from itertools import product
from typing import Union, Optional, List, Tuple
import graphviz


ndarray = Union[np.ndarray, cupy.ndarray]


class Tensor:
    #--

    def __init__(
        self,
        data: Union[ndarray, List[Union[float, int]], int, float],
        grad: Optional[ndarray] = None,
        device: Optional[str] = None
    ):
        self.grad = grad
        self.device = device
        if device == "cuda":
            global np
            np = import_cupy_else_numpy()
        self.data = np.array(data)
        self._ctx: Operator = None

    def __repr__(self):
        return f"Tensor({self.data})"

    def __str__(self):
        return f"Tensor shape: {self.data.shape}\nGradient: {self.grad}\nDevice: {self.device}"

    def __add__(self, other: 'Tensor'):
        return self.add(other)

    def __mul__(self, other: 'Tensor'):
        return self.mul(other)

    def __sub__(self, other: 'Tensor'):
        return self.add(other.mul(-1))

    def to(self, device: str):
        #--
        return Tensor(self.data, device=device)

    def backward(self, allow_fill: bool = True) -> None:
        #--
        if self._ctx is None:
            return

        if self.grad is None and allow_fill:
            ##

            assert self.data.size == 1
            self.grad = np.ones_like(self.data)

        assert (self.grad is not None)
```

```
    parent_grads = self._ctx.backward(
        self._ctx, self.grad)  ##

    if len(self._ctx.operands) == 1:
        parent_grads = np.expand_dims(parent_grads, axis=0)

    for tensor, gradient in zip(self._ctx.operands, parent_grads):
        if tensor.data.shape != gradient.shape:
            ##

            gradient = gradient.mean(axis=0)

        if tensor.grad is None:
            tensor.grad = gradient
        else:
            tensor.grad += gradient

        tensor.backward(allow_fill=False)

def mean(self) -> 'Tensor':
    #--
    div = Tensor(np.array([1/self.data.size]))
    return self.sum().mul(div)

def draw_graph(self, namespace: Optional[dict] = None) -> None:
    #--
    ##

    def get_label(tensor: Tensor):
        return f'shape: {tensor.data.shape} grad: {True if tensor.grad is not None else False}'

    def build_graph(tensor, dot: graphviz.Digraph):
        if tensor in dot:
            return
        ##

        name = None
        if namespace is not None:
            for var_name, var_value in namespace.items():
                if var_value is tensor:
                    name = var_name
                    break

        ##

        label = name if name is not None else str(id(tensor))

        dot.node(str(id(tensor)),
                 f'{label}: {get_label(tensor)}', shape='box')
        if tensor._ctx is not None:
            dot.node(str(id(tensor._ctx)), type(tensor._ctx).__name__)
            dot.edge(str(id(tensor._ctx)), str(id(tensor)))
            for operand in tensor._ctx.operands:
                if isinstance(operand, Tensor):
                    dot.edge(str(id(operand)), str(id(tensor._ctx)))
                    build_graph(operand, dot)
        else:
            return
```

```
        dot = graphviz.Digraph()
        build_graph(self, dot)
        dot.render("graph", format="png", cleanup=True)


class Operator:
    #--

    def __init__(self, *tensors: Tensor) -> None:
        self.operands = tensors
        self.saved_tensors: List[Tensor] = []

    def save_for_backward(self, *x: Tensor):
        #--
        self.saved_tensors.extend(x)

    def apply(self, arg: 'Operator', *x: tuple) -> Tensor:
        #--
        ctx = arg(self, *x)
        ret = Tensor(arg.forward(ctx, self.data, *
                     [t.data if isinstance(t, Tensor) else t for t in x]))
        ret._ctx = ctx
        return ret


def register(name, fxn):
    #--
    setattr(Tensor, name, partialmethod(fxn.apply, fxn))


class Div(Operator):
    #--
    @staticmethod
    def forward(ctx, x, y) -> ndarray:
        #--
        ctx.save_for_backward(x, y)
        return x / y

    @staticmethod
    def backward(ctx, grad_output) -> Tuple[ndarray, ndarray]:
        #--
        x, y = ctx.saved_tensors
        return grad_output / y, -x * grad_output / y**2


register('div', Div)


class Mul(Operator):
    #--
    @staticmethod
    def forward(ctx, x, y) -> ndarray:
        #--
        ctx.save_for_backward(x, y)
        return x*y

    @staticmethod
    def backward(ctx, grad_output) -> Tuple[ndarray, ndarray]:
        #--
        x, y = ctx.saved_tensors
```

```
        return y*grad_output, x*grad_output


register('mul', Mul)


class Add(Operator):
    #--
    @staticmethod
    def forward(ctx, x, y) -> ndarray:
        #--
        ctx.save_for_backward(x, y)
        return x+y

    @staticmethod
    def backward(ctx, grad_output) -> Tuple[ndarray, ndarray]:
        #--
        x, y = ctx.saved_tensors
        return grad_output, grad_output


register('add', Add)


class ReLU(Operator):
    #--
    @staticmethod
    def forward(ctx: Operator, input: ndarray) -> ndarray:
        #--
        ctx.save_for_backward(input)
        return np.maximum(input, 0)

    @staticmethod
    def backward(ctx: Operator, grad_output: ndarray) -> ndarray:
        #--
        input, = ctx.saved_tensors
        grad_input = grad_output.copy()
        grad_input[input < 0] = 0
        return grad_input


register('relu', ReLU)


class Dot(Operator):
    #--
    @staticmethod
    def forward(ctx: Operator, input: ndarray, weight: ndarray) -> ndarray:
        #--
        ctx.save_for_backward(input, weight)
        return input.dot(weight)

    @staticmethod
    def backward(ctx: Operator, grad_output: ndarray) -> Tuple[ndarray, ndarray]:
        #--
        input, weight = ctx.saved_tensors
        grad_input = grad_output.dot(weight.T)
        grad_weight = input.T.dot(grad_output)
        return grad_input, grad_weight
```

```
register('dot', Dot)


class Sum(Operator):
    #--
    @staticmethod
    def forward(ctx: Operator, input: ndarray) -> ndarray:
        #--
        ctx.save_for_backward(input)
        return np.array([input.sum()])

    @staticmethod
    def backward(ctx: Operator, grad_output: ndarray) -> ndarray:
        #--
        input, = ctx.saved_tensors
        return grad_output * np.ones_like(input)


register('sum', Sum)


class Softmax(Operator):
    #--
    @staticmethod
    def forward(ctx: Operator, input: ndarray) -> ndarray:
        #--
        exp = np.exp(input - np.max(input, axis=-1, keepdims=True))
        output = exp / exp.sum(axis=-1, keepdims=True)
        ctx.save_for_backward(output)
        return output

    @staticmethod
    def backward(ctx: Operator, grad_output: ndarray) -> ndarray:
        #--
        output, = ctx.saved_tensors
        d_softmax = output * (1 - output)  ##

        jacobian = -output[..., None] * output[:, None, :]  ##

        jacobian[:, np.arange(output.shape[1]), np.arange(
            output.shape[1])] = d_softmax  ##

        return np.einsum('ij,ijk->ik', grad_output, jacobian)


register('softmax', Softmax)


class LogSoftmax(Operator):
    #--
    @staticmethod
    def forward(ctx: Operator, input: ndarray) -> ndarray:
        #--
        def logsumexp(x):
            c = x.max(axis=1)
            return c + np.log(np.exp(x-c.reshape((-1, 1))).sum(axis=1))
        output = input - logsumexp(input).reshape((-1, 1))
        ctx.save_for_backward(output)
        return output
```

```
    @staticmethod
    def backward(ctx: Operator, grad_output: ndarray) -> Tuple[ndarray, ndarray]:
        #--
        output, = ctx.saved_tensors
        return grad_output - np.exp(output)*grad_output.sum(axis=1).reshape((-1, 1))


register('logsoftmax', LogSoftmax)


class MSE(Operator):
    #--
    @staticmethod
    def forward(ctx: Operator, input: ndarray, target: ndarray) -> ndarray:
        #--

        ctx.save_for_backward(input, target)
        return np.array((input - target)**2)

    @staticmethod
    def backward(ctx: Operator, grad_output: ndarray) -> Tuple[ndarray, ndarray]:
        input, target = ctx.saved_tensors
        grad_input = grad_output * 2*(input - target)
        grad_target = grad_output * -2*(input - target)
        return grad_input, grad_target


register('mse', MSE)


class Conv2d(Operator):
    ##

    #--
    @staticmethod
    def forward(ctx: Operator, input: ndarray, weight: ndarray, padding: int, stride: int) -> ndarray:
        #--
        ctx.save_for_backward(input, weight, padding, stride)

        N, C, W, H = input.shape
        F, C, WW, HH = weight.shape

        assert (H + 2 * padding - HH) % stride == 0
        assert (W + 2 * padding - WW) % stride == 0

        out_h = (H + 2 * padding - HH) // stride + 1
        out_w = (W + 2 * padding - WW) // stride + 1

        input_pad = np.pad(
            input, ((0, 0), (0, 0), (padding, padding), (padding, padding)), 'constant')
        output = np.zeros((N, F, out_h, out_w))
        for n in range(N):
            for f in range(F):
                for i in range(out_h):
                    for j in range(out_w):
                        output[n, f, i, j] = np.sum(
                            input_pad[n, :, i * stride:i * stride + HH, j *
                                      stride:j * stride + WW] * weight[f, :, :, :]
                        )
```

```
        return output

    @staticmethod
    def backward(ctx: Operator, grad_output: ndarray) -> Tuple[ndarray, ndarray]:
        #--
        input, weight, padding, stride = ctx.saved_tensors

        N, C, H, W = input.shape
        F, _, HH, WW = weight.shape
        out_h = (H + 2 * padding - HH) // stride + 1
        out_w = (W + 2 * padding - WW) // stride + 1
        grad_weight = np.zeros_like(weight)
        grad_input = np.zeros_like(input)

        input_pad = np.pad(
            input, ((0, 0), (0, 0), (padding, padding), (padding, padding)), 'constant')

        for n in range(N):
            for f in range(F):
                for i in range(out_h):
                    for j in range(out_w):
                        input_patch = input_pad[n, :, i *
                                                stride:i*stride+HH, j*stride:j*stride+WW]
                        grad_weight[f] += input_patch * grad_output[n, f, i, j]

        return grad_input, grad_weight


register('conv2d', Conv2d)


class Reshape(Operator):
    #--
    @staticmethod
    def forward(ctx: Operator, x: ndarray, shape: tuple) -> ndarray:
        #--
        ctx.operands = (ctx.operands[0],)
        ctx.save_for_backward(x.shape)
        return x.reshape(shape)

    @staticmethod
    def backward(ctx: Operator, grad_output: ndarray) -> ndarray:
        #--
        original_shape, = ctx.saved_tensors
        return grad_output.reshape(original_shape)


register('reshape', Reshape)


class MAE(Operator):
    #--
    @staticmethod
    def forward(ctx: Operator, input: ndarray, target: ndarray) -> ndarray:
        #--
        ctx.save_for_backward(input, target)
        return np.abs(input - target)

    @staticmethod
```

```
    def backward(ctx: Operator, grad_output: ndarray) -> Tuple[ndarray, ndarray]:
        #--
        input, target = ctx.saved_tensors
        grad_input = grad_output * np.sign(input - target)
        grad_target = grad_output * -np.sign(input - target)
        return grad_input, grad_target


register('mae', MAE)

class Sigmoid(Operator):
    #--
    @staticmethod
    def forward(ctx: Operator, input: ndarray) -> ndarray:
        #--
        output = 1 / (1 + np.exp(-input))
        ctx.save_for_backward(output)
        return output

    @staticmethod
    def backward(ctx: Operator, grad_output: ndarray) -> ndarray:
        #--
        output, = ctx.saved_tensors
        return grad_output * output * (1 - output)

register('sigmoid', Sigmoid)
class Exp(Operator):
    #--
    @staticmethod
    def forward(ctx: Operator, input: ndarray) -> ndarray:
        #--
        ctx.save_for_backward(input)
        return np.exp(input)

    @staticmethod
    def backward(ctx: Operator, grad_output: ndarray) -> ndarray:
        #--
        input, = ctx.saved_tensors
        return grad_output * np.exp(input)


register('exp', Exp)


class Log(Operator):
    #--
    @staticmethod
    def forward(ctx: Operator, input: ndarray) -> ndarray:
        #--
        ctx.save_for_backward(input)
        return np.log(input)

    @staticmethod
    def backward(ctx: Operator, grad_output: ndarray) -> ndarray:
        #--
        input, = ctx.saved_tensors
        return grad_output / input


register('log', Log)
```

```
class LayerNorm(Operator):
    #--

    @staticmethod
    def forward(ctx: Operator, input: ndarray, gamma: ndarray, beta: ndarray, epsilon: float = 1e-5) ->
    ↪  ndarray:
        #--
        ctx.save_for_backward(input, gamma, beta, epsilon)
        mean = np.mean(input, axis=-1, keepdims=True)
        variance = np.var(input, axis=-1, keepdims=True)
        output = (input - mean) / np.sqrt(variance + epsilon)
        output = output * gamma + beta
        return output

    @staticmethod
    def backward(ctx: Operator, grad_output: ndarray) -> Tuple[ndarray, ndarray, ndarray]:
        #--
        input, gamma, beta, epsilon = ctx.saved_tensors
        mean = np.mean(input, axis=-1, keepdims=True)
        variance = np.var(input, axis=-1, keepdims=True)
        grad_input = grad_output * gamma / np.sqrt(variance + epsilon)
        grad_gamma = np.sum(grad_output * (input - mean) / np.sqrt(variance + epsilon), axis=-1,
        ↪  keepdims=True)
        grad_beta = np.sum(grad_output, axis=-1, keepdims=True)
        return grad_input, grad_gamma, grad_beta

register('layernorm', LayerNorm)
```

deepdive/nn.py

```
from .tensor import Tensor
from .utils import import_cupy_else_numpy
from abc import ABC, abstractmethod
import numpy as np
from typing import List, Tuple, Optional, Union, Dict, TypedDict
import os


class Optimizer(ABC):
    #--

    def __init__(
            self,
            params: Optional[dict] = None,
            lr: float = 1e-3,
            weight_decay: Optional[float] = 0.0
        ):
        self.params = params
        self.lr = lr
        self.weight_decay = weight_decay

    @abstractmethod
    def step(self):
        #--
        pass


class SGD(Optimizer):
    #--
```

```
    def step(self):
        #--
        for p in self.params:
            if self.weight_decay != 0:
                p.grad += self.weight_decay * p.data
            p.data -= self.lr * p.grad


class Adam(Optimizer):
    #--

    def __init__(
            self,
            params: Optional[dict] = None,
            lr: float = 1e-3, betas: Tuple[float, float] = (0.9, 0.999),
            eps: float = 1e-8,
            weight_decay: Optional[float] = 0.0
        ):
        super().__init__(params, lr, weight_decay)
        self.betas = betas
        self.eps = eps
        self.m = [np.zeros_like(p.data) for p in self.params.values()]
        self.v = [np.zeros_like(p.data) for p in self.params.values()]
        self.t = 0

    def step(self):
        #--
        self.t += 1
        for i, p in enumerate(self.params):
            if self.weight_decay != 0:
                p.grad += self.weight_decay * p.data
            self.m[i] = self.betas[0] * self.m[i] +                (1 - self.betas[0]) * p.grad
            self.v[i] = self.betas[1] * self.v[i] +                (1 - self.betas[1]) * p.grad**2
            m_hat = self.m[i] / (1 - self.betas[0]**self.t)
            v_hat = self.v[i] / (1 - self.betas[1]**self.t)
            p.data -= self.lr * m_hat / (np.sqrt(v_hat) + self.eps)


class Layer:
    #--

    def __init__(self):
        self.params: Dict[str, Tensor] = {}

    def step(self, optimizer: Optimizer, lr: float = 1e-3, weight_decay: Optional[float] = 0.0):
        #--
        for p in self.params.values():
            if p.grad.shape != p.data.shape:
                ##

                p.grad = p.grad.mean(axis=0)
        optim = optimizer(self.params.values(), lr, weight_decay)
        optim.step()
        ##

        self.zero_grad()

    def zero_grad(self):
        #--
```

```
        for p in self.params.values():
            p.grad.fill(0)


class Linear(Layer):
    #--

    def __init__(self, in_features: int, out_features: int):
        super().__init__()

        w = np.random.uniform(
            low=-1.,
            high=1.,
            size=(in_features, out_features)
        ) / np.sqrt(in_features * out_features)

        b = np.zeros(
            shape=(out_features,)
        )
        self.params = {
            "LinearW": Tensor(w),
            "LinearB": Tensor(b)
        }

    def __str__(self) -> str:
        return f"Linear({self.params['LinearW'].data.shape[0]},{self.params['LinearW'].data.shape[1]})"

    def __call__(self, x: Tensor) -> Tensor:
        ##

        #--
        x = x.dot(self.params["LinearW"])
        x = x.add(self.params["LinearB"])
        return x

    def to(self, device: str):
        #--
        for name, p in self.params.items():
            self.params[name] = p.to(device)
        return self


class Conv2d(Layer):
    #--

    def __init__(
        self,
        in_channels: int,
        out_channels: int,
        kernel_size: int,
        stride: int = 1,
        padding: int = 0
    ):
        super().__init__()
        self.kernel_size = kernel_size
        self.stride = stride
        self.padding = padding
        w = np.random.uniform(
            low=-1.,
            high=1.,
```

```
            size=(out_channels, in_channels, kernel_size, kernel_size)
        ) / np.sqrt(in_channels*out_channels*kernel_size*kernel_size)
        self.params["Conv2dW"] = Tensor(w)

    def __call__(self, x: Tensor) -> Tensor:
        #--
        output = x.conv2d(self.params["Conv2dW"], self.padding, self.stride)
        return output

    def __str__(self) -> str:
        return
        ↪  f"Conv2d({self.params['Conv2dW'].data.shape[1]},{self.params['Conv2dW'].data.shape[0]},{self.kernel_
        ↪  {self.stride},{self.padding})"

    def to(self, device: str):
        #--
        for name, p in self.params.items():
            self.params[name] = p.to(device)
        return self

class LayerNorm(Layer):
    #--

    def __init__(self, normalized_shape: int, eps: float = 1e-5):
        super().__init__()
        self.normalized_shape = normalized_shape
        self.eps = eps
        self.weight = np.ones(shape=(normalized_shape,))
        self.bias = np.zeros(shape=(normalized_shape,))
        self.params = {
            "LayerNormGamma": Tensor(self.weight),
            "LayerNormBeta": Tensor(self.bias)
        }

    def __call__(self, x: Tensor) -> Tensor:
        #--
        return x.layernorm(self.params["LayerNormGamma"], self.params["LayerNormBeta"], self.eps)

    def __str__(self) -> str:
        return f"LayerNorm({self.normalized_shape})"

    def to(self, device: str):
        #--
        for name, p in self.params.items():
            self.params[name] = p.to(device)
        return self


class ReShape():
    #--

    def __init__(self, shape: tuple) -> None:
        self.shape = shape

    def __call__(self, x: Tensor) -> Tensor:
        #--
        return x.reshape(self.shape)

    def __str__(self) -> str:
        return f"ReShape{self.shape}"
```

```
class ReLU():
    #--

    def __call__(self, x: Tensor) -> Tensor:
        #--
        return x.relu()

    def __str__(self) -> str:
        return "ReLU"


class LogSoftmax():
    #--

    def __call__(self, x: Tensor) -> Tensor:
        #--
        return x.logsoftmax()

    def __str__(self) -> str:
        return "LogSoftmax"

class Softmax():
    #--

    def __call__(self, x: Tensor) -> Tensor:
        #--
        return x.softmax()

    def __str__(self) -> str:
        return "Softmax"
class Sequential:
    #--

    def __init__(self, *layers: Layer):
        self.layers = layers

    def forward(self, x: Tensor):
        #--
        for layer in self.layers:
            x = layer(x)
        return x

    def __call__(self, x: Tensor) -> Tensor:
        #--
        return self.forward(x)

    def save(self, dir_path: str, save_arc: bool = True):
        #--
        if not os.path.exists(dir_path):
            os.makedirs(dir_path)

        for i, layer in enumerate(self.layers):
            if hasattr(layer, 'params'):
                for name, p in layer.params.items():
                    np.save(os.path.join(
                        dir_path, f"layer_{i}_{name}.npy"), p.data)
        ##
```

```
        if save_arc:
            with open(os.path.join(dir_path, "model.arc"), "w") as f:
                f.write(str(self))

    def load(self, dir_path: str):
        #--
        for i, layer in enumerate(self.layers):
            if hasattr(layer, 'params'):
                for name in layer.params.keys():
                    layer.params[name] = np.load(
                        os.path.join(
                            dir_path,
                            f"layer_{i}_{name}.npy"
                        ),
                        allow_pickle=True
                    )

    def __str__(self) -> str:
        return "\n".join([str(l) for l in self.layers])

    def step(self, lr, optimizer: Optimizer, weight_decay: Optional[float] = 0.0):
        #--
        for layer in self.layers:
            if isinstance(layer, Layer):
                layer.step(lr=lr, weight_decay=weight_decay,
                           optimizer=optimizer)

    def to(self, device: str):
        #--
        for layer in self.layers:
            if isinstance(layer, Layer):
                layer.to(device)
        global np
        if device == "cuda":
            np = import_cupy_else_numpy()
        else:
            np = np
        return self

def load_from_arc(model_dir: str):
    #--
    layers = []
    with open(os.path.join(model_dir, "model.arc"), "r") as f:
        for line in f:
            line = line.strip()
            if line.startswith("Linear"):
                in_features, out_features = map(int, line[7:-1].split(","))
                layers.append(Linear(in_features, out_features))
            elif line.startswith("Conv2d"):
                in_channels, out_channels, kernel_size, stride, padding = map(
                    int, line[7:-1].split(","))
                layers.append(Conv2d(
                    in_channels, out_channels, kernel_size, stride, padding))
            elif line.startswith("ReShape"):
                shape = tuple(map(int, line[7:-1].split(",")))
                layers.append(ReShape(shape))
            elif line == "ReLU":
                layers.append(ReLU())
            elif line == "LogSoftmax":
                layers.append(LogSoftmax())
```

```
            elif line == "Softmax":
                layers.append(Softmax())
    model = Sequential(*layers)
    model.load(model_dir)
    return model
```

deepdive/datautils.py

```
from .tensor import Tensor
from datasets import DatasetDict, Dataset
import random
import numpy as np
##

##

##

##

##



class DataLoader:
    #--

    def __init__(self, dataset: DatasetDict, batch_size: int = 1, shuffle: bool = False) -> None:
        self.dataset = dataset
        self.bs = batch_size
        if shuffle:
            self.shuffle()
        self.batches = self.get_batches()
        self.batch_index = 0

    def get_batches(self):
        #--
        num_batches, last_bs = divmod(len(self.dataset), self.bs)
        batches = [self.dataset.select(
            range(i*self.bs, (i+1)*self.bs)) for i in range(num_batches)]
        if last_bs != 0:
            batches.append(self.dataset.select(
                range(num_batches*self.bs, num_batches*self.bs + last_bs)))
        return batches

    def shuffle(self):
        ##

        #--
        arr = self.dataset.to_list()
        for i in range(len(arr) - 1, 0, -1):
            j = random.randint(0, i)
            arr[i], arr[j] = arr[j], arr[i]

        self.dataset = Dataset.from_list(arr)

    def __iter__(self):
        return self

    def __len__(self):
```

```
        #--
        return len(self.batches)

    def __next__(self):
        #--
        if self.batch_index >= len(self.batches):
            raise StopIteration
        batch = self.batches[self.batch_index]
        self.batch_index += 1
        return_values = [batch[feature] for feature in batch.features.keys()]
        return_values = [np.array(return_value)
                         for return_value in return_values]
        return *return_values,
```

tests.py

```
import numpy as np
import torch
from deepdive.tensor import Tensor


def test_mul():
    x_init = np.random.randn(3, 4).astype(np.float32)
    y_init = np.random.randn(3, 4).astype(np.float32)
    X, x = Tensor(x_init), torch.tensor(x_init, requires_grad=True)
    Y, y = Tensor(y_init), torch.tensor(y_init, requires_grad=True)

    Z = X.mul(Y)
    Z.mean().backward()
    z = x.mul(y)
    z.mean().backward()

    np.testing.assert_allclose(Z.data, z.detach().numpy(), atol=1e-6)
    np.testing.assert_allclose(X.grad, x.grad, atol=1e-6)
    np.testing.assert_allclose(Y.grad, y.grad, atol=1e-6)


def test_add():
    x_init = np.random.randn(3, 4).astype(np.float32)
    y_init = np.random.randn(3, 4).astype(np.float32)
    X, x = Tensor(x_init), torch.tensor(x_init, requires_grad=True)
    Y, y = Tensor(y_init), torch.tensor(y_init, requires_grad=True)

    Z = X.add(Y)
    Z.mean().backward()
    z = x.add(y)
    z.mean().backward()

    np.testing.assert_allclose(Z.data, z.detach().numpy(), atol=1e-6)
    np.testing.assert_allclose(X.grad, x.grad, atol=1e-6)
    np.testing.assert_allclose(Y.grad, y.grad, atol=1e-6)


def test_conv2d():
    x_init = np.random.randn(1, 1, 28, 28).astype(np.float32)
    w_init = np.random.randn(1, 1, 3, 3).astype(np.float32)
    X, x = Tensor(x_init), torch.tensor(x_init, requires_grad=True)
    Y, y = Tensor(w_init), torch.tensor(w_init, requires_grad=True)

    Z = X.conv2d(Y, 1, 1)
```

```
    Z.mean().backward()
    z = torch.nn.functional.conv2d(x, y, padding=1, stride=1)
    z.mean().backward()

    np.testing.assert_allclose(Z.data, z.detach().numpy(), atol=1e-5)
    np.testing.assert_allclose(Y.grad, y.grad, atol=1e-5)


def test_sum():
    x_init = np.random.randn(3, 4).astype(np.float32)
    X, x = Tensor(x_init), torch.tensor(x_init, requires_grad=True)

    Z = X.sum()
    Z.mean().backward()
    z = x.sum()
    z.mean().backward()

    np.testing.assert_allclose(Z.data, z.detach().numpy(), atol=1e-6)
    np.testing.assert_allclose(X.grad, x.grad, atol=1e-6)


def test_relu():
    x_init = np.random.randn(3, 4).astype(np.float32)
    X, x = Tensor(x_init), torch.tensor(x_init, requires_grad=True)

    Z = X.relu()
    Z.mean().backward()
    z = torch.nn.functional.relu(x)
    z.mean().backward()

    np.testing.assert_allclose(Z.data, z.detach().numpy(), atol=1e-6)
    np.testing.assert_allclose(X.grad, x.grad, atol=1e-6)


def test_dot():
    x_init = np.random.randn(3, 4).astype(np.float32)
    y_init = np.random.randn(4, 5).astype(np.float32)
    X, x = Tensor(x_init), torch.tensor(x_init, requires_grad=True)
    Y, y = Tensor(y_init), torch.tensor(y_init, requires_grad=True)

    Z = X.dot(Y)
    Z.mean().backward()
    z = x.matmul(y)
    z.mean().backward()

    np.testing.assert_allclose(Z.data, z.detach().numpy(), atol=1e-6)
    np.testing.assert_allclose(X.grad, x.grad, atol=1e-6)
    np.testing.assert_allclose(Y.grad, y.grad, atol=1e-6)


def test_logsoftmax():
    x_init = np.random.randn(3, 4).astype(np.float32)
    X, x = Tensor(x_init), torch.tensor(x_init, requires_grad=True)

    Z = X.logsoftmax()
    Z.mean().backward()
    z = torch.nn.functional.log_softmax(x, dim=-1)
    z.mean().backward()

    np.testing.assert_allclose(Z.data, z.detach().numpy(), atol=1e-6)
```

```
    np.testing.assert_allclose(X.grad, x.grad, atol=1e-6)


def test_softmax():
    x_init = np.random.randn(1, 10).astype(np.float32)
    X, x = Tensor(x_init), torch.tensor(x_init, requires_grad=True)

    Z = X.softmax()
    Z.mean().backward()
    z = torch.nn.functional.softmax(x, dim=-1)
    z.mean().backward()

    np.testing.assert_allclose(Z.data, z.detach().numpy(), atol=1e-6)
    np.testing.assert_allclose(X.grad, x.grad, atol=1e-6)


def test_exponential():
    x_init = np.random.randn(3, 4).astype(np.float32)
    X, x = Tensor(x_init), torch.tensor(x_init, requires_grad=True)

    Z = X.exp()
    Z.mean().backward()
    z = torch.exp(x)
    z.mean().backward()

    np.testing.assert_allclose(Z.data, z.detach().numpy(), atol=1e-6)
    np.testing.assert_allclose(X.grad, x.grad, atol=1e-6)


def test_log():
    x_init = np.random.rand(3, 4).astype(np.float32)
    X, x = Tensor(x_init), torch.tensor(x_init, requires_grad=True)

    Z = X.log()
    Z.mean().backward()
    z = torch.log(x)
    z.mean().backward()

    np.testing.assert_allclose(Z.data, z.detach().numpy(), atol=1e-6)
    np.testing.assert_allclose(X.grad, x.grad, atol=1e-6)

def test_sigmoid():
    x_init = np.random.randn(3, 4).astype(np.float32)
    X, x = Tensor(x_init), torch.tensor(x_init, requires_grad=True)

    Z = X.sigmoid()
    Z.mean().backward()
    z = torch.sigmoid(x)
    z.mean().backward()

    np.testing.assert_allclose(Z.data, z.detach().numpy(), atol=1e-6)
    np.testing.assert_allclose(X.grad, x.grad, atol=1e-6)

def test_layernorm():
    x_init = np.random.randn(3, 4).astype(np.float32)
    X, x = Tensor(x_init), torch.tensor(x_init, requires_grad=True)

    Z = X.layernorm()
    Z.mean().backward()
    z = torch.nn.functional.layer_norm(x, x.size()[1:])
```

```
    z.mean().backward()

    np.testing.assert_allclose(Z.data, z.detach().numpy(), atol=1e-6)
    np.testing.assert_allclose(X.grad, x.grad, atol=1e-6)
```

mnist.py

```
import numpy as np
from deepdive.tensor import Tensor
import deepdive.nn as nn
from tqdm import trange
from datasets import load_dataset
from deepdive.dash import Dash

mnist = load_dataset('mnist')


def convert_to_np(example):
    example['np_image'] = np.asarray(example['image'])
    return example


mnist = mnist.map(convert_to_np)

X_train, Y_train = np.asarray(
    mnist['train']['np_image']), np.asarray(mnist['train']['label'])
X_test, Y_test = np.asarray(
    mnist['test']['np_image']), np.asarray(mnist['test']['label'])

lr = 0.01
BS = 64

model = nn.Sequential(
    nn.Linear(784, 128),
    nn.LayerNorm(128),
    nn.ReLU(),
    nn.Linear(128, 10),
    nn.LayerNorm(10),
    nn.LogSoftmax()
)

##


for i in (t := trange(1000)):
    samp = np.random.randint(0, X_train.shape[0], size=(BS))
    x = Tensor(X_train[samp].reshape((-1, 28*28)))
    Y = Y_train[samp]
    y = np.zeros((len(samp), 10), np.float32)
    y[range(y.shape[0]), Y] = -1.0
    y = Tensor(y)
    output = model.forward(x)
    x = output.mul(y)
    x = x.mean()
    x.backward()

    loss = x.data
    cat = np.argmax(output.data, axis=1)
    accuracy = (cat == Y).mean()
    ##
```

```
    t.set_description(f"loss {loss} accuracy {accuracy}")
    ##

    ##

    model.step(lr=lr, optimizer=nn.SGD)

##

model.save('mnist_model')
```

CNNmnist.py

```
from deepdive.tensor import Tensor
import deepdive.nn as nn
from deepdive.datautils import DataLoader
import numpy as np
from datasets import load_dataset, Dataset
from tqdm import trange

mnist: Dataset = load_dataset('mnist')
dataloader = DataLoader(mnist["train"], batch_size=32)

model: nn.Sequential = nn.Sequential(
    nn.Conv2d(1, 4, 3, 1, 0),
    nn.ReLU(),
    nn.Conv2d(4, 8, 3, 1, 0),
    nn.ReLU(),
    nn.ReShape((-1, 8*24*24)),
    nn.Linear(4608, 1024),
    nn.ReLU(),
    nn.Linear(1024, 10),
    nn.LogSoftmax()
)


def train(num_epochs: int = 10):
    for i, (inp, targets) in enumerate(dataloader):
        inp = Tensor(np.expand_dims(inp, axis=1))
        targets = Tensor(np.eye(10)[targets])
        output = model.forward(inp)
        loss = output.mul(targets).mean()
        loss.backward()
        if i == 0:
            loss.draw_graph(locals())
        model.step(lr=0.01, weight_decay=0.0001, optimizer=nn.SGD)
        print(f"loss {loss.data}")
        if i == num_epochs:
            break


def test_accuracy(num_tests=100):
    accuracy = []
    for i in range(num_tests):
        inp, targets = mnist["test"][i]['image'], mnist["test"][i]['label']
        inp = np.array(inp)
        inp = Tensor(np.expand_dims(np.expand_dims(inp, axis=0), axis=0))
        output = model.forward(inp)
        prediction = np.argmax(output.data)
```

```
        accuracy.append(prediction == targets)
    accuracy = np.array(accuracy)
    accuracy = accuracy.mean()
    return accuracy


train(10)
print(test_accuracy())
```

deepdive/dash.py

```
##

import plotext as plt


class Dash:
    def __init__(self, num_metrics: int) -> None:
        self.num_metrics = num_metrics
        self.metrics = [[] for _ in range(num_metrics)]
        self.t = 0

    def update(self, *args) -> None:
        for i, arg in enumerate(args):
            self.metrics[i].append(arg)
        self.t += 1
        self.plot()

    def plot(self) -> None:
        plt.clf()
        for i in range(self.num_metrics):
            plt.plot(self.metrics[i], label=f"metric {i}")
        plt.title(f"Training Metrics")
        plt.xlabel("Time")
        plt.ylabel("Value")
        plt.show()
```

deepdive/utils.py

```
import importlib
import warnings


def import_cupy_else_numpy():
    #--
    try:
        return importlib.import_module("cupy")
    except ImportError:
        warnings.warn(
            "CuPy not installed, falling back to NumPy", stacklevel=2)
        return importlib.import_module("numpy")


def check_cupy() -> bool:
    try:
        importlib.import_module('cupy')
        return True
    except ImportError:
        return False
```

4 Testing

Automatic Differentiation		
Objective Number	Test or Implementation	Pass/-Fail
1	deepdive/tensor.py page no. 53 The `data` and `grad` attributes of the `Tensor` class are numpy ndarrays used for storing the contents of the tensor	Pass
2	deepdive/tensor.py this is demonstrated by every class that is inherited by the `Operator` class e.g. Mul, ReLU, LogSoftmax. This can be seen throughout tensor.py	Pass
3	deepdive/tensor.py	Pass
4.1	tests.py	Pass
4.2	tests.py	Pass
4.3	tests.py	Pass

Neural Network Functionality		
Objective Number	Test or Implementation	Pass/-Fail
1	deepdive/nn.py page no. 65 To follow PyTorchs naming convention, this is called `Sequential`	Pass
2.1	deepdive/nn.py page no. 63 Implemented as a Layer called `Linear`	Pass
2.2	deepdive/nn.py page no. 63 Implemented as a Layer called `Conv2d`	Pass
2.3	deepdive/nn.py page no. 61 This is implemented as a layer called `LayerNorm`	Pass
3.1	deepdive/nn.py page no. 65 Many Activation functions are implemented in this file, they can be distinguished as they are classes that do not inherit from Layer (because they dont have trainable parameters)	Pass
4.1	deepdive/nn.py page no. 61 Implemented as `SGD`	Pass
4.2	deepdive/nn.py page no. 62 Implemented as `Adam`	Pass
5.1	deepdive/nn.py page no. 63 Xavier Initialization can be seen to be used when weights are initialized in the `Linear` or `Conv2d` layers	Pass
6.1	deepdive/nn.py This can be seen implemented in each optimizer as weight decay	Pass
6.2	deepdive/nn.py This can be seen implemented in each optimizer as weight decay	Pass
6.3	deepdive/nn.py This is implemented as a layer called `LayerNorm`	Pass
7	deepdive/nn.py page no. 66 this is implemented as methods `save`, `load` and `load_from_arc` which are all part of the `Sequential` class	Pass
8	deepdive/nn.py page no. 53 GPU support is added using cupy, you can move tensors onto the GPU using the `to` method of the `Tensor` class	Pass

Data utilities		
Objective Number	Test or Implementation	Pass/-Fail
1	deepdive/datautils.py page no. 40 usage of this is shown in the upcoming testing jupyter notebook	Pass
2	deepdive/datautils.py page no. 40-41 this is implemented in the `get_batches` method in the `DataLoader` class	Pass
2	deepdive/datautils.py page no. 40 this is implemented in the `shuffle` method in the `DataLoader` class	Pass

Testing

March 12, 2024

1 Testing the library

1.0.1 Importing the library

Note that if the library is installed via pip than it can be imported as shown below

`from deepdive.tensor import Tensor`

no need for dd

[1]:
```
from dd.deepdive.tensor import Tensor
import dd.deepdive.nn as nn
import numpy as np
```

[2]:
```
from datasets import load_dataset
from tqdm import trange
from dd.deepdive.datautils import DataLoader
```

1.1 Testing Completeness of objectives

1.1.1 Main Objective 1: Automatic Differentiation

- **Tensor object can store n-dimensional values for parameters and gradients**
  - Tensor can be initialized given a list of values
  - Tensor can be initialized with random values given a shape

[5]:
```
# Tensor creation from numpy nd array
a_tensor = Tensor(np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]))
# Initialize gradient with ones of same shape as data
a_tensor.grad = np.ones_like(a_tensor.data)
# Now with random values of given shape
b_tensor = Tensor(np.random.rand(3, 3))
print(a_tensor)
print(a_tensor.data)
```

```
Tensor shape: (10,)
Gradient: None
Device: None
[ 1  2  3  4  5  6  7  8  9 10]
```

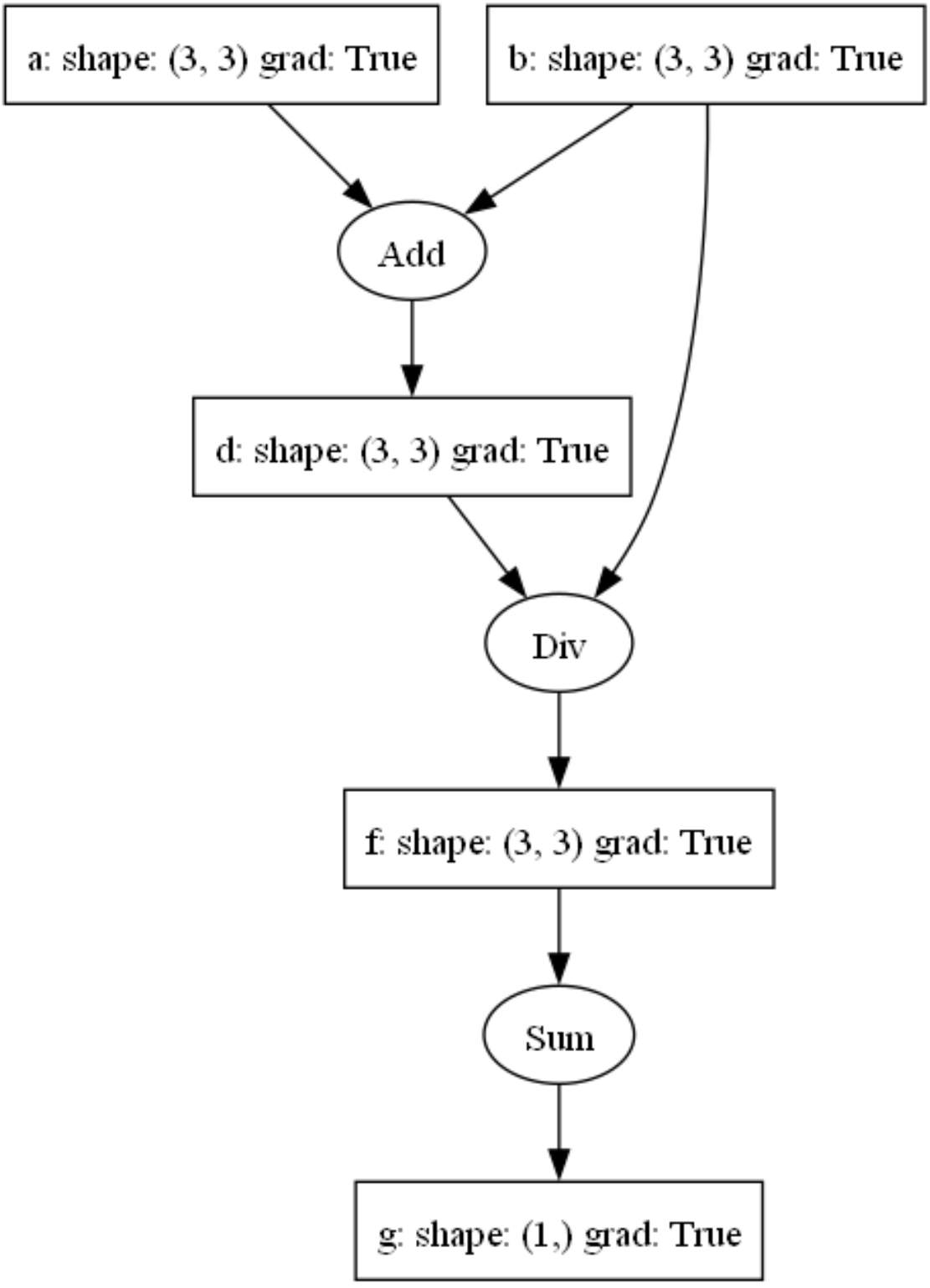
- **Develop gradient functions for basic operations such as addition, subtraction, and matrix multiplication, as well as for common Deep Learning operations like logsoftmax, sigmoid, and ReLU.**
  - These functions have been implemented in `tensor.py` and are tested against results from pytorch in `tests.py`
- **Implement a backward function that computes the gradient for each intermediate variable in the computational graph. This function should be recursive to simplify understanding and usage.**
  - The backward function has been implemented in `tensor.py` and is tested against results from pytorch in `tests.py`

Here I will showcase usage of automatic differentiation using the library. I will also use the draw_graph function to visualize the computational graph for the operations.

[4]:
```
import random

# Create random tensors
a = Tensor(np.random.rand(3, 3))
b = Tensor(np.random.rand(3, 3))
c = Tensor(np.random.rand(3, 3))

# Perform operations
d = a + b
e = b * c
f = d.div(b)
g = f.sum()
g.backward()
g.draw_graph(locals())
```

[5]:
```
from IPython.display import Image
Image(filename='graph.png')
```

[5]:


a: shape: (3, 3) grad: True
b: shape: (3, 3) grad: True
Add
d: shape: (3, 3) grad: True
Div
f: shape: (3, 3) grad: True
Sum
g: shape: (1,) grad: True

1.1.2 Main Objective 2: Neural Network Functionality

- **Create a NeuralNet class which can be used to construct neural networks from individual layers and operations**
  - To make the library easy to learn, I used the same naming convention as PyTorch (as requested by the client), therefore the class is called Sequential instead of NeuralNet
  - All objectives can be shown to be complete by constructing a Sequential object containing the layers that the client requested. Activation functions are also included in model construction

Below is a demonstration of training a simple neural network using the library. The network is trained to learn to recognize handwritten digits. The network uses the *ReLU* activation function. The network is trained using the *stochastic gradient descent* optimizer

[7]:
```
mnist = load_dataset('mnist')
```

[ ]:
```
def convert_to_np(example):
    example['np_image'] = np.asarray(example['image'])
    return example

mnist = mnist.map(convert_to_np)

X_train, Y_train = np.asarray(
    mnist['train']['np_image']), np.asarray(mnist['train']['label'])
X_test, Y_test = np.asarray(
    mnist['test']['np_image']), np.asarray(mnist['test']['label'])

lr = 0.01
BS = 64

model = nn.Sequential(
    nn.Linear(784, 128),
    nn.LayerNorm(128),
    nn.ReLU(),
    nn.Linear(128, 10),
    nn.LayerNorm(10),
    nn.LogSoftmax()
)
```

4.6 - **GPU support for neural network training** (optional)

To enable GPU support, the library uses the `cupy` library. The library is a drop-in replacement for numpy and can be used to perform operations on the GPU. The library is not installed by default, but can be installed using pip. The library is not required for the library to function, but if it is installed, the library will be able to place tensors on the GPU and perform operations on the GPU. Below is a demonstration of how a model can be simply moved to the GPU for training

[ ]:
```
model = model.to("cuda")
```

[ ]:
```python
for i in (t := trange(1000)):
    samp = np.random.randint(0, X_train.shape[0], size=(BS))
    x = Tensor(X_train[samp].reshape((-1, 28*28)))
    Y = Y_train[samp]
    y = np.zeros((len(samp), 10), np.float32)
    y[range(y.shape[0]), Y] = -1.0
    y = Tensor(y)
    output = model.forward(x)
    x = output.mul(y)
    x = x.mean()
    x.backward()

    loss = x.data
    cat = np.argmax(output.data, axis=1)
    accuracy = (cat == Y).mean()
    t.set_description(f"loss {loss} accuracy {accuracy}")
    # SGD
    model.step(lr=lr, optimizer=nn.SGD)
```

- 4.5 - *Model saving to disk and loading from files. Use an appropriate method to export model weights.*

For this objective, I have implemented the `save` and `load` methods in the Sequential class. The methods are used to save the model to disk and load the model from disk respectively. The methods use the `numpy` inbuilt save method which is used to save a numpy ndarray into a binary file. A custom `.arc` file is also created to store the model architecture. The `.arc` file is a simple text file that contains the model architecture in a human-readable format. The `load` method reads the `.arc` file and uses the information to construct the model. The `load` method then reads the weights from the binary file and assigns them to the model. The methods are demonstrated below

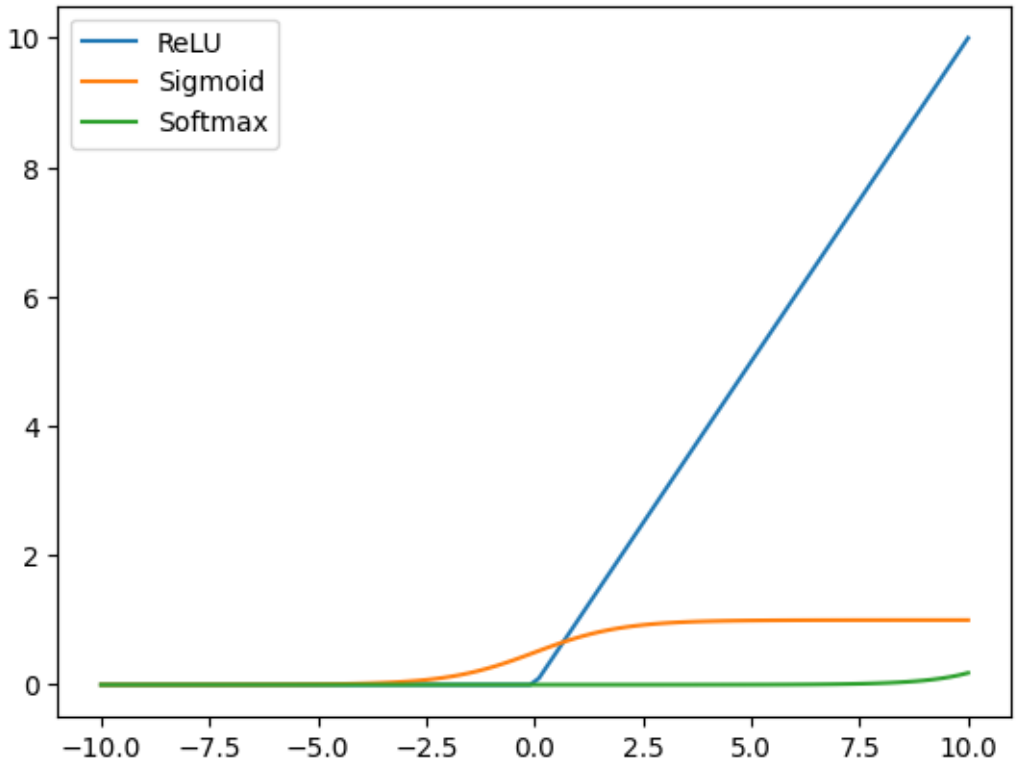
[ ]:
```python
# Model Saving
model.save('mnist_model', save_arc = True)
```

[9]:
```python
# Model Loading
model = nn.load_from_arc('mnist_model')

# Model Inference
image_tensor = Tensor(np.array(mnist['test'][0]['image']).reshape(1, 784))
logits = model.forward(image_tensor)
prediction = logits.data.argmax()
display(mnist['test'][0]['image'])
print(f"Prediction: {prediction}")
```

```
Prediction: 7
```

- **Activation Functions**:
  - ReLU
  - Sigmoid
  - softmax

[6]:
```
import matplotlib.pyplot as plt

x = np.linspace(-10, 10, 100)
y_relu = Tensor(x).relu().data
y_sigmoid = Tensor(x).sigmoid().data
y_softmax = Tensor(x).softmax().data
plt.plot(x, y_relu, label='ReLU')
plt.plot(x, y_sigmoid, label='Sigmoid')
plt.plot(x, y_softmax, label='Softmax')
plt.legend()
plt.show()
```



1.1.3 Main objective 3: Data Utilities

- **Create a DataLoader class which can be used to load data from disk and provide it to the model in batches**

- The library contains a DataLoader class which can be used to load data from a Huggingface dataset and provide it to the model in batches. The DataLoader class is demonstrated below
- Datasets can be shuffled
- Dataloader can split the dataset into batches

[17]:
```
mnist = load_dataset('mnist')
print(mnist)
dataloader = DataLoader(mnist["train"], batch_size = 32, shuffle = True)

for (batch_input, batch_target) in dataloader:
    print(batch_input.shape, batch_target.shape)
    break
```

```
DatasetDict({
    train: Dataset({
        features: ['image', 'label'],
        num_rows: 60000
    })
    test: Dataset({
        features: ['image', 'label'],
        num_rows: 10000
    })
})
(32,) (32,)
```

Testing the correctness of the computed gradients

This is a description of how objective **4** was reached in the library. To test if the gradients calculated from backpropagation is correct, I made tests for each operator by applying the operator onto two tensors and comparing the result with the same operator applied to the tensors using the PyTorch library. Knowing that the results from PyTorch will be correct, I can verify if the gradients for my library are being computated correctly. Here is an example of a test from `tests.py` of the `Mul` operator

```
def test_mul():
    x_init = np.random.randn(3, 4).astype(np.float32)
    y_init = np.random.randn(3, 4).astype(np.float32)
    X, x = Tensor(x_init), torch.tensor(x_init, requires_grad=True)
    Y, y = Tensor(y_init), torch.tensor(y_init, requires_grad=True)

    Z = X.mul(Y)
    Z.mean().backward()
    z = x.mul(y)
    z.mean().backward()

    np.testing.assert_allclose(Z.data, z.detach().numpy(), atol=1e-6)
    np.testing.assert_allclose(X.grad, x.grad, atol=1e-6)
    np.testing.assert_allclose(Y.grad, y.grad, atol=1e-6)
```

The atol parameter stands for "absolute tolerance". It's used in functions like `numpy.testing.assert_allclose` to specify the absolute tolerance within which the two arrays being compared should be considered "close".
If the absolute difference is greater than atol for any pair of elements, `numpy.testing.assert_allclose` will raise an `AssertionError`.

5 Evaluation

5.1 Client Feedback

I requested my client to use my library and review it. We had a conversation about the completeness of the project and I also requested he rate how well he thinks each project objective has been met. Here is the outline of the conversation

ME: After using the library, what are your immediate thoughts on it?

ALEX DAVICENKO: The modularity of the library is very well thought, and better than I had imagined it would be. The ability to add `Operators` to the tensor class so easily by defining its forward and backward in one class is an amazing feature. I love the simplicity of model construction, saving and loading. Following the naming conventions of PyTorch has also proved to make the code easier to write, as I can trust that the names of the methods which I am trying to use are familiar.

ME: Could you elaborate more on the modularity and extensibility aspects that impressed you?

ALEX DAVICENKO: The ability to easily define and register new operators for the Tensor class is a remarkable feature that sets your library apart. This flexibility allows for seamless integration of custom operations, enabling users to tailor the library to their specific needs.

ME: I see. And what do you think about the model construction process?

ALEX DAVICENKO: The model construction, saving, and loading process is simple. the introduction of the `.arc` file type is an ingenious solution that streamlines the process of loading pre-trained models. The ability to load models in a single line of code, without the need to reconstruct the model architecture beforehand, is a game-changer. This feature also promotes code readability which is great.

ME: That's great to hear. How would you rate the overall completeness and functionality of the library?

ALEX DAVICENKO: You have covered a wide range of functionalities, from basic tensor operations to advanced layers like convolutional and layer normalization layers. The inclusion of optimizers such as SGD and Adam further enhances the library's capabilities, making it a comprehensive solution for deep learning tasks.

ME: Thank you for the feedback. Do you have any final thoughts or remarks?

ALEX DAVICENKO: I can confidently say that you have met and exceeded the project objectives, delivering a library that is both user-friendly and highly capable.

ME: What improvements do you think would make this library a better product and how can this make it better?

ALEX DAVICENKO: One potential improvement could be to move away from relying heavily on NumPy ndarrays for a significant portion of its functionality. Although this approach allows for major performance gains, it would be desirable to have the ndarray functionality implemented natively in Python as part of the library itself. This would not only enhance the library's modularity and self-sufficiency but also provide greater control over the implementation details, allowing for further optimizations and customizations if needed. Considering the libraries goal is not performance but instead simplicity, I feel this is a potentially good improvement

Important takeaways from this interview:

- Modular design and ease of adding new operators to the Tensor class
- Simplicity of model construction, saving, and loading
- Adherence to PyTorch naming conventions for familiarity
- Introduction of `.arc` file type for streamlined model loading
- Comprehensive functionality, including tensor operations and advanced layers
- Inclusion of optimizers like SGD and Adam

- Potential improvement: Implement ndarray functionality natively in Python for better modularity and customization

w

Bibliography

[1] Marc Peter Deisenroth, A Aldo Faisal, and Cheng Soon Ong. *Mathematics for Machine Learning*. Cambridge University Press, 2020.

[2] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey, 2015.

[3] Jonathon Price, Alfred Wong, Tiancheng Yuan, Joshua Mathews, and Taiwo Olorunniwo. Stochastic gradient descent - cornell university computational optimization open textbook - optimization wiki.

[4] Akash Ajagekar. Adam - cornell university computational optimization open textbook - optimization wiki.

[5] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 12 2014.

[6] Python Packaging Authority. A sample python package project, 07 2023.