SmulGrad: Building Automatic Differentiation from Scratch

A Hands-On Learning Assignment

Inspired by Stanford CS336 and Karpathy's micrograd

Contents

1 Overview

Automatic differentiation (autodiff) is the backbone of modern deep learning frameworks. Unlike symbolic differentiation (which manipulates mathematical expressions) or numerical differentiation (which approximates derivatives using finite differences), autodiff computes exact derivatives by decomposing computations into elementary operations and applying the chain rule systematically.

In **reverse-mode autodiff** (also called backpropagation), we:

1. Build a computational graph during the forward pass
2. Traverse this graph backwards to accumulate gradients using the chain rule

This approach is efficient when we have many inputs and few outputs—exactly the case in neural network training where we differentiate a scalar loss with respect to millions of parameters.

1.1 Learning Objectives

By completing this assignment, you will:

- Understand how computational graphs represent mathematical expressions
- Implement the reverse-mode autodiff algorithm from scratch
- Handle tensor operations and broadcasting in gradient computation
- Build a system capable of training neural networks

1.2 Assignment Structure

Part	Topic	Points
1	Scalar Values and Basic Operations	15
2	The Backward Pass	20
3	More Operations	15
4	Tensor Support	22
5	Matrix Operations	17
6	Neural Network Training	15
	Total	**104**

1.3 Getting Started

```
# Install dependencies
uv sync

# Run all tests (most will fail initially)
uv run pytest

# Run tests for a specific part
uv run pytest -k "part1"

# Run a single test
uv run pytest -k "TestValueCreation"
```

1.4 File Structure

```
smulgrad/
    engine.py            # Value and Tensor classes (Parts 1-5)
    nn.py                # Neural network utilities (Part 6)
tests/
    adapters.py          # Adapter functions connecting your code to
       tests
    cases/               # Test files (do not modify)
        test_part1_scalar.py
        test_part2_backward.py
        test_part3_ops.py
        test_part4_tensor.py
        test_part5_matrix.py
        test_part6_nn.py
        conftest.py      # Pytest fixtures
```

You will implement everything in the `smulgrad/` directory:

- `engine.py`: Your `Value` and `Tensor` classes (Parts 1–5)
- `nn.py`: Neural network utilities like `softmax`, `cross_entropy`, `SGD`, `Linear` (Part 6)

The tests import your code through adapter functions in `tests/adapters.py`. Import directly from your modules, e.g., `from smulgrad.engine import Value`.

2 Part 1: Scalar Values and Basic Operations

(15 points)

We begin by creating a `Value` class that wraps a scalar number and tracks operations performed on it. This is the fundamental building block of our autodiff system.

2.1 The Value Class

A `Value` object needs to store:

- `data`: The actual numerical value (a Python float)
- `grad`: The gradient of some output with respect to this value (initialized to 0.0)
- `_prev`: A set of `Value` objects that this value depends on (its "children" in the computational graph)
- `_op`: A string describing how this value was created (e.g., `'+'`, `'*'`, `'relu'`)

Problem (value_creation): Create a Value class that stores a scalar value (2 points)

Your `Value` class should support:

```python
v = Value(3.0)
print(v.data)  # 3.0
print(v.grad)  # 0.0
```

Deliverable: Implement the adapter `create_value` in `tests/adapters.py`. Run `uv run pytest -k TestValueCreation` to verify.

2.2 Addition

We want to be able to add `Value` objects together using Python's `+` operator. When we compute `c = a + b`, we need to:

1. Compute `c.data = a.data + b.data`
2. Record that `c` depends on `a` and `b` (store them in `c._prev`)
3. Record the operation (`c._op = '+'`)

Problem (value_add): Implement addition for Value objects (2 points)

Your implementation should support:

```python
a = Value(2.0)
b = Value(3.0)
c = a + b
print(c.data)  # 5.0
```

You should also handle adding a `Value` to a plain Python number:

```python
a = Value(2.0)
c = a + 3  # Should work
c = 3 + a  # Should also work (hint: __radd__)
```

Deliverable: Implement the adapter `run_add` in `tests/adapters.py`. Run `uv run pytest -k TestValueAdd` to verify.

2.3 Multiplication

Problem (value_mul): Implement multiplication for Value objects (2 points)

```python
a = Value(2.0)
b = Value(3.0)
c = a * b
print(c.data)  # 6.0

# Also support scalar multiplication
c = a * 3  # Should work
c = 3 * a  # Should also work
```

Deliverable: Implement the adapter `run_mul`. Run `uv run pytest -k TestValueMul` to verify.

2.4 Negation and Subtraction

Problem (value_neg_sub): Implement negation and subtraction (3 points)

Hint: Subtraction can be implemented in terms of addition and negation: `a - b = a + (-b)`

```python
a = Value(5.0)
b = Value(3.0)
print((-a).data)      # -5.0
print((a - b).data)   # 2.0
```

Deliverable: Implement the adapters `run_neg` and `run_sub`. Run `uv run pytest -k TestValueNegSub` to verify.

2.5 Power

Problem (value_pow): Implement raising a Value to a constant power (3 points)

```python
a = Value(2.0)
c = a ** 3
print(c.data)  # 8.0
```

Note: We only support constant exponents (Python int or float), not `Value` exponents.
Deliverable: Implement the adapter `run_pow`. Run `uv run pytest -k TestValuePow` to verify.

2.6 Division

Problem (value_div): Implement division (3 points)

Hint: Division can be implemented using multiplication and power: `a / b = a * (b ** -1)`

```python
a = Value(6.0)
b = Value(2.0)
print((a / b).data)  # 3.0
```

Deliverable: Implement the adapter `run_div`. Run `uv run pytest -k TestValueDiv` to verify.

3 Part 2: The Backward Pass

(20 points)

Now comes the core of autodiff: computing gradients via backpropagation. The key insight is that each operation knows how to compute the gradient with respect to its inputs, given the gradient with respect to its output.

3.1 Understanding the Chain Rule

For a function composition $y = f(g(x))$, the chain rule tells us:

$$\frac{dy}{dx} = \frac{dy}{dg} \cdot \frac{dg}{dx} \tag{1}$$

In our computational graph:

- Each node computes some value
- The "upstream gradient" is the gradient of the final output with respect to this node
- Each node must compute the "downstream gradient"—the gradient with respect to its inputs

For addition $c = a + b$:

$$\frac{\partial c}{\partial a} = 1 \qquad \frac{\partial c}{\partial b} = 1 \tag{2}$$

So: `a.grad += c.grad * 1`, `b.grad += c.grad * 1`

For multiplication $c = a \cdot b$:

$$\frac{\partial c}{\partial a} = b \qquad \frac{\partial c}{\partial b} = a \tag{3}$$

So: `a.grad += c.grad * b.data`, `b.grad += c.grad * a.data`

3.2 Local Backward Functions

Each `Value` needs a `_backward` function that propagates gradients to its children. This function should be set when the `Value` is created by an operation.

Problem (backward_add): Implement gradient computation for addition (3 points)

When you create `c = a + b`, you should also define `c._backward` such that calling it will add to `a.grad` and `b.grad`.
Deliverable: The test `test_backward_add` will verify your addition backward pass.

Problem (backward_mul): Implement gradient computation for multiplication (3 points)

For $c = a \cdot b$:

- `a.grad += c.grad * b.data`
- `b.grad += c.grad * a.data`

Deliverable: Run `uv run pytest -k TestBackwardMul` to verify.

Problem (backward_ops): Implement gradients for other operations (4 points)

Gradients for remaining operations:

$$\text{Negation:} \quad \frac{\partial(-a)}{\partial a} = -1 \tag{4}$$

$$\text{Power:} \quad \frac{\partial(a^n)}{\partial a} = n \cdot a^{n-1} \tag{5}$$

$$\text{Division:} \quad \frac{\partial(a/b)}{\partial a} = \frac{1}{b}, \quad \frac{\partial(a/b)}{\partial b} = -\frac{a}{b^2} \tag{6}$$

Deliverable: Run `uv run pytest -k TestBackwardOps` to verify.

3.3 Topological Sort and Full Backward

To compute all gradients, we need to:

1. Build a topological ordering of all nodes (children before parents)
2. Set the gradient of the output node to 1.0
3. Call `_backward()` on each node in reverse topological order

Problem (backward_full): Implement a backward() method that computes all gradients (5

```
a = Value(2.0)
b = Value(3.0)
c = a * b
d = c + a
d.backward()

print(a.grad)  # d(d)/d(a) = d(c+a)/da = dc/da + 1 = b + 1 = 4.0
print(b.grad)  # d(d)/d(b) = dc/db = a = 2.0
```

Key implementation detail: You need to traverse the graph in reverse topological order. Use depth-first search to build the ordering.
Deliverable: Implement the adapter `run_backward`. Run `uv run pytest -k TestBackwardFull` to verify.

3.4 Gradient Accumulation

When a value is used multiple times in a computation, its gradient should accumulate.

Problem (grad_accumulation): Ensure gradients accumulate correctly (3 points)

```
a = Value(3.0)
b = a + a  # a is used twice
b.backward()
print(a.grad)  # Should be 2.0, not 1.0
```

Deliverable: Run `uv run pytest -k TestGradAccumulation` to verify.

Problem (grad_of_grad): Second derivatives (bonus) (2 points)

A powerful property of autodiff is that gradients are themselves computations that can be differentiated. If your implementation is clean, second derivatives should work automatically.

```python
a = Value(2.0)
b = a * a  # b = a^2
b.backward()
# a.grad is now 2*a = 4.0
# If we could differentiate again, d(2a)/da = 2
```

Deliverable: Run `uv run pytest -k TestGradOfGrad` to verify (bonus).

4 Part 3: More Operations

(15 points)

Real neural networks need more than just arithmetic. Let's add activation functions and other operations.

4.1 ReLU

The Rectified Linear Unit (ReLU) is defined as:

$$\text{relu}(x) = \max(0, x) \tag{7}$$

Its gradient is:

$$\frac{d(\text{relu}(x))}{dx} = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

Problem (relu): Implement the ReLU activation function (3 points)

```python
a = Value(-3.0)
b = a.relu()
print(b.data)  # 0.0

c = Value(3.0)
d = c.relu()
print(d.data)  # 3.0
```

Deliverable: Implement the adapter `run_relu`. Run `uv run pytest -k TestReLU` to verify.

4.2 Exponential

Problem (exp): Implement the exponential function (3 points)

```python
a = Value(2.0)
b = a.exp()
print(b.data)  # e^2 ~ 7.389
```

Gradient: $\frac{d(e^x)}{dx} = e^x$

Deliverable: Implement the adapter `run_exp`. Run `uv run pytest -k TestExp` to verify.

4.3 Natural Logarithm

Problem (log): Implement the natural logarithm (3 points)

```python
a = Value(2.0)
b = a.log()
print(b.data)  # ln(2) ~ 0.693
```

Gradient: $\frac{d(\ln x)}{dx} = \frac{1}{x}$

Deliverable: Implement the adapter `run_log`. Run `uv run pytest -k TestLog` to verify.

4.4 Hyperbolic Tangent

Problem (tanh): Implement the hyperbolic tangent (3 points)

```python
a = Value(1.0)
b = a.tanh()
print(b.data)  # tanh(1) ~ 0.7616
```

Gradient: $\frac{d(\tanh x)}{dx} = 1 - \tanh^2(x)$

Note: You can implement tanh using exp: $\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$, but implementing it directly with its own backward is more numerically stable.

Deliverable: Implement the adapter `run_tanh`. Run `uv run pytest -k TestTanh` to verify.

4.5 Complex Expression Test

Problem (complex_expr): Test a complex expression combining multiple operations (3 poi

```python
a = Value(2.0)
b = Value(3.0)
c = a * b + a ** 2
d = c.relu() - b / a
e = d.tanh()
e.backward()
```

Deliverable: Run `uv run pytest -k TestComplexExpr` to verify.

5 Part 4: Tensor Support

(20 points)

Scalars are great for understanding, but real neural networks operate on tensors. Now we extend our system to support multi-dimensional arrays.

5.1 The Tensor Class

Create a `Tensor` class that wraps a numpy array instead of a scalar. The structure is similar to `Value`:

- `data`: A numpy ndarray
- `grad`: Gradient array (same shape as data), initialized to zeros
- `_prev`: Set of parent tensors
- `_op`: Operation string
- `_backward`: Backward function

Problem (tensor_creation): Create a Tensor class (2 points)

```python
t = Tensor([[1.0, 2.0], [3.0, 4.0]])
print(t.data.shape)  # (2, 2)
print(t.grad.shape)  # (2, 2)
```

Deliverable: Implement the adapter `create_tensor`. Run `uv run pytest -k TestTensorCreation` to verify.

5.2 Element-wise Operations

Problem (tensor_elementwise): Implement element-wise add and multiply for tensors (3 po

```python
a = Tensor([[1.0, 2.0], [3.0, 4.0]])
b = Tensor([[5.0, 6.0], [7.0, 8.0]])
c = a + b
print(c.data)  # [[6, 8], [10, 12]]

d = a * b
print(d.data)  # [[5, 12], [21, 32]]
```

The backward pass for element-wise operations:

- Addition: gradients pass through unchanged
- Multiplication: each element's gradient is multiplied by the corresponding element of the other tensor

Deliverable: Implement adapters `run_tensor_add` and `run_tensor_mul`. Run `uv run pytest -k TestTensorElementwise` to verify.

5.3 Backward Pass for Tensors

Just like `Value`, your `Tensor` class needs a `backward()` method to compute gradients through the computational graph.

Problem (tensor_backward): Implement backward() for Tensor (2 points)

The backward pass for tensors works identically to scalars:

1. Build a topological ordering of all tensors in the graph
2. Set the gradient of the output tensor to ones (same shape as data)
3. Call `_backward()` on each tensor in reverse topological order

```python
a = Tensor([[1.0, 2.0], [3.0, 4.0]])
b = Tensor([[5.0, 6.0], [7.0, 8.0]])
c = a + b
c.backward()  # Computes gradients for a and b
print(a.grad)  # [[1, 1], [1, 1]]
```

Note: For non-scalar outputs, `backward()` should initialize the output gradient to `np.ones_like(self.data)`.
Deliverable: Implement the adapter `run_tensor_backward`. Run `uv run pytest -k TestTensorBackward` to verify.

5.4 Broadcasting

NumPy broadcasting allows operations between arrays of different shapes. Your backward pass must handle this correctly by summing gradients along broadcasted dimensions.

Problem (tensor_broadcast): Handle broadcasting in backward pass (4 points)

```python
a = Tensor([[1.0, 2.0, 3.0]])  # Shape (1, 3)
b = Tensor([[1.0], [2.0]])     # Shape (2, 1)
c = a + b  # Shape (2, 3) due to broadcasting
c.backward()

# a.grad should have shape (1, 3) - sum along axis 0
# b.grad should have shape (2, 1) - sum along axis 1
```

Key insight: When numpy broadcasts a smaller array to a larger shape, the backward pass must sum the gradients along the broadcasted dimensions to get back to the original shape.
Deliverable: Run `uv run pytest -k TestTensorBroadcast` to verify.

5.5 Sum Reduction

Problem (tensor_sum): Implement sum reduction (3 points)

```python
a = Tensor([[1.0, 2.0], [3.0, 4.0]])
b = a.sum()  # Sum all elements
print(b.data)  # 10.0

c = a.sum(axis=0)  # Sum along axis 0
print(c.data)  # [4.0, 6.0]

d = a.sum(axis=1)  # Sum along axis 1
print(d.data)  # [3.0, 7.0]

e = a.sum(axis=0, keepdims=True)
print(e.data.shape)  # (1, 2)
```

Gradient: The gradient flows back to all summed elements equally.
Deliverable: Implement the adapter `run_tensor_sum`. Run `uv run pytest -k TestTensorSum` to verify.

5.6 Mean Reduction

Problem (tensor_mean): Implement mean reduction (2 points)

```python
a = Tensor([[1.0, 2.0], [3.0, 4.0]])
b = a.mean()
print(b.data)  # 2.5
```

Gradient: $\frac{\partial(\text{mean})}{\partial x_i} = \frac{1}{n}$ for each element.
Deliverable: Implement the adapter `run_tensor_mean`. Run `uv run pytest -k TestTensorMean` to verify.

5.7 Max Reduction

Problem (tensor_max): Implement max reduction (3 points)

```python
a = Tensor([[1.0, 4.0], [3.0, 2.0]])
b = a.max()
print(b.data)  # 4.0
```

Gradient: The gradient only flows to the maximum element(s). If multiple elements share the maximum value, the gradient should be split equally among them.
Deliverable: Implement the adapter `run_tensor_max`. Run `uv run pytest -k TestTensorMax` to verify.

5.8 Reshape and Transpose

Problem (tensor_reshape): Implement reshape operation (2 points)

```python
a = Tensor([[1.0, 2.0], [3.0, 4.0]])
b = a.reshape((4,))
print(b.data)  # [1.0, 2.0, 3.0, 4.0]
```

Gradient: Reshape the gradient back to the original shape.
Deliverable: Implement the adapter `run_tensor_reshape`. Run `uv run pytest -k TestTensorReshape` to verify.

Problem (tensor_transpose): Implement transpose operation (1 points)

```python
a = Tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
b = a.T  # or a.transpose()
print(b.data.shape)  # (3, 2)
```

Gradient: Transpose the gradient.
Deliverable: Implement the adapter `run_tensor_transpose`. Run `uv run pytest -k TestTensorTranspose` to verify.

6 Part 5: Matrix Operations

(15 points)

Matrix multiplication is the workhorse of neural networks. This section focuses on getting the gradients right for matrix operations.

6.1 Matrix Multiplication

For matrices $Y = XW$ where X is (n, m) and W is (m, d):

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} W^T \tag{9}$$

$$\frac{\partial L}{\partial W} = X^T \frac{\partial L}{\partial Y} \tag{10}$$

Problem (matmul): Implement matrix multiplication with correct gradients (5 points)

```python
X = Tensor([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])  # (3, 2)
W = Tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])    # (2, 3)
Y = X @ W  # or matmul(X, W)
print(Y.data.shape)  # (3, 3)

Y.sum().backward()
# Check X.grad and W.grad
```

Deliverable: Implement the adapter `run_matmul`. Run `uv run pytest -k TestMatmul` to verify.

6.2 Batched Matrix Multiplication

Problem (batched_matmul): Handle batched matrix multiplication (4 points)

```python
# Batch of 2 matrices
X = Tensor(np.random.randn(2, 3, 4))  # (batch, n, m)
W = Tensor(np.random.randn(2, 4, 5))  # (batch, m, d)
Y = X @ W  # (batch, n, d) = (2, 3, 5)
```

Deliverable: Run `uv run pytest -k TestBatchedMatmul` to verify.

6.3 Matrix-Vector Products

Problem (matvec): Handle matrix-vector products (3 points)

```python
A = Tensor([[1.0, 2.0], [3.0, 4.0]])  # (2, 2)
x = Tensor([1.0, 2.0])                 # (2,)
y = A @ x                              # (2,)
print(y.data)  # [5.0, 11.0]
```

Deliverable: Run `uv run pytest -k TestMatvec` to verify.

6.4 Tensor Activation Functions

Just like the `Value` class, your `Tensor` class needs element-wise activation functions. These apply the operation to each element independently.

Problem (tensor_activations): Implement activation functions for Tensor (2 points)

Implement the following activation functions for your Tensor class:

- `relu()`: Element-wise ReLU: $\max(0, x)$
- `exp()`: Element-wise exponential: e^x
- `log()`: Element-wise natural logarithm: $\ln(x)$
- `tanh()`: Element-wise hyperbolic tangent

```python
a = Tensor([[-1.0, 2.0], [3.0, -4.0]])
b = a.relu()
print(b.data)  # [[0, 2], [3, 0]]

c = Tensor([[1.0, 2.0]])
d = c.exp()
print(d.data)  # [[2.718..., 7.389...]]
```

The backward passes are the same as for `Value`, but applied element-wise:

- ReLU: gradient is 1 where input > 0, else 0
- exp: gradient is e^x
- log: gradient is $1/x$
- tanh: gradient is $1 - \tanh^2(x)$

Deliverable: Implement adapters `run_tensor_relu`, `run_tensor_exp`, and `run_tensor_log`. Run `uv run pytest -k TestTensorActivations` to verify.

6.5 Complex Matrix Expression

Problem (matrix_chain): Test a chain of matrix operations (3 points)

```python
X = Tensor(...)  # (batch, in_features)
W1 = Tensor(...) # (in_features, hidden)
W2 = Tensor(...) # (hidden, out_features)

hidden = (X @ W1).relu()
output = hidden @ W2
loss = output.sum()
loss.backward()
```

Deliverable: Run `uv run pytest -k TestMatrixChain` to verify.

7 Part 6: Neural Network Training

(15 points)

Now let's put everything together to train an actual neural network. In this section, you will implement the core components needed for training: layers, activation functions, loss functions, and optimizers.

7.1 Linear Layer

A linear (fully connected) layer performs an affine transformation on its input:

$$y = xW + b \tag{11}$$

where $x \in \mathbb{R}^{n \times d_{in}}$ is the input, $W \in \mathbb{R}^{d_{in} \times d_{out}}$ is the weight matrix, and $b \in \mathbb{R}^{d_{out}}$ is the bias vector.

Problem (linear_layer): Implement a linear (fully connected) layer (2 points)

Create a `Linear` class with:

- `__init__(self, in_features, out_features)`: Initialize weight matrix W with small random values (e.g., scaled by 0.01) and bias b with zeros. Both should be `Tensor` objects.
- `__call__(self, x)`: Compute $xW + b$ using your Tensor operations.
- `parameters(self)`: Return a list containing the weight and bias tensors.

Deliverable: Implement the adapter `run_linear_layer`. Run `uv run pytest -k TestLinearLayer` to verify.

7.2 Softmax

The softmax function converts raw scores (logits) into a probability distribution:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{12}$$

For numerical stability, subtract the maximum value before exponentiating:

$$\text{softmax}(x_i) = \frac{e^{x_i - \max(x)}}{\sum_j e^{x_j - \max(x)}} \tag{13}$$

This prevents overflow when x contains large values.

Problem (softmax): Implement the softmax function (3 points)

Implement a `softmax(x, axis=-1)` function that:

1. Subtracts the maximum value along the specified axis (with `keepdims=True`)
2. Applies the exponential function element-wise
3. Divides by the sum along the specified axis (with `keepdims=True`)

Use only your Tensor operations (`max`, `exp`, `sum`, subtraction, division).
Deliverable: Implement the adapter `run_softmax`. Run `uv run pytest -k TestSoftmax` to verify.

7.3 Cross-Entropy Loss

Cross-entropy measures the difference between two probability distributions. For classification with one-hot encoded targets t and predicted probabilities p:

$$L = -\sum_i t_i \log(p_i) \tag{14}$$

When combined with softmax, the gradient has a remarkably simple form. For logits z and one-hot targets t:

$$\frac{\partial L}{\partial z} = \text{softmax}(z) - t \tag{15}$$

Problem (cross_entropy): Implement cross-entropy loss (3 points)

Implement `softmax_cross_entropy(logits, targets)` that:

1. Takes raw logits (not probabilities) and one-hot encoded targets
2. Computes the cross-entropy loss using the numerically stable formula:

$$L = -\sum_i t_i (z_i - \log \sum_j e^{z_j}) \tag{16}$$

which simplifies to: $L = \log \sum_j e^{z_j} - \sum_i t_i z_i$

3. Returns a scalar loss (mean over the batch)

Hint: For the backward pass, you can either derive it through your existing operations, or implement a custom backward that uses the elegant gradient formula above.
Deliverable: Implement the adapter `run_cross_entropy`. Run `uv run pytest -k TestCrossEntropy` to verify.

7.4 SGD Optimizer

Stochastic Gradient Descent (SGD) updates parameters in the direction that reduces the loss:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L \tag{17}$$

where η is the learning rate and $\nabla_\theta L$ is the gradient of the loss with respect to parameters θ.

Problem (sgd): Implement stochastic gradient descent (2 points)

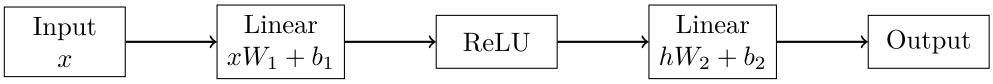
Create an `SGD` class with:

- `__init__(self, parameters, lr=0.01)`: Store the list of parameter tensors and learning rate.
- `step(self)`: Update each parameter's `data` by subtracting $\eta \times$ grad.
- `zero_grad(self)`: Reset all parameter gradients to zero (important before each backward pass).

Note: Modify `p.data` directly (a numpy array), not the Tensor object itself.
Deliverable: Implement the adapter `run_sgd_step`. Run `uv run pytest -k TestSGD` to verify.

7.5 MLP Training Loop

A Multi-Layer Perceptron (MLP) stacks linear layers with non-linear activations:



The training loop repeats:

1. **Forward pass**: Compute predictions from inputs
2. **Loss computation**: Compare predictions to targets
3. **Backward pass**: Compute gradients via `loss.backward()`
4. **Parameter update**: Apply SGD step
5. **Zero gradients**: Reset gradients for next iteration

Problem (mlp_training): Train a simple MLP on synthetic data (3 points)

Implement an `MLP` class that:

- Takes `input_size`, `hidden_size`, and `output_size` as constructor arguments
- Creates two `Linear` layers: input→hidden and hidden→output
- In `__call__`: applies first linear, then ReLU, then second linear
- In `parameters()`: returns all parameters from both layers

The test will create synthetic classification data and verify that your MLP can be trained (loss decreases over iterations).
Deliverable: Implement the adapter `create_mlp`, `run_mlp`, and `get_mlp_parameters`. Run `uv run pytest -k TestMLP` to verify.

7.6 Interactive Demo

Once your MLP is working, run the interactive demo to watch your neural network learn in real-time:

```
uv run python tests/cases/demo_mlp.py
```

This will train your MLP on a concentric circles dataset and show:

- Real-time visualization of the decision boundary evolving
- Loss curve showing training progress
- Final performance summary

This is the reward for all your hard work—seeing your autodiff engine actually train a neural network!

7.7 Numerical Gradient Checking

Gradient checking verifies your analytical gradients by comparing them to numerical approximations. Using the central difference formula:

$$\frac{\partial f}{\partial x_i} \approx \frac{f(x + h \cdot e_i) - f(x - h \cdot e_i)}{2h} \tag{18}$$

where e_i is a unit vector in the i-th direction and h is a small value (e.g., 10^{-5}).

Problem (gradient_check): Implement gradient checking (2 points)

Implement `check_gradients(f, input, epsilon=1e-5, tolerance=1e-4)` that:

1. Computes analytical gradients by calling `f(input)` and `backward()`
2. Perturbs each element of the input tensor by $\pm\epsilon$ and computes numerical gradient
3. Compares analytical and numerical gradients using relative error:

$$\text{rel\_error} = \frac{|g_{analytical} - g_{numerical}|}{\max(|g_{analytical}|, |g_{numerical}|) + \epsilon} \tag{19}$$

4. Returns `True` if all relative errors are below the tolerance

Hint: Flatten the tensor to iterate over elements, modify `data` directly when perturbing, and remember to restore the original value after each perturbation.
Deliverable: Implement the adapter `run_gradient_check`. Run `uv run pytest -k TestGradientCheck` to verify.

8 Grading Summary

9 Bonus Challenges

Once you've completed the main assignment, try these extensions:

Bonus 1: More Optimizers **5 points**

Implement Adam optimizer with momentum and adaptive learning rates.

Bonus 2: Convolutional Operations **10 points**

Implement 2D convolution with correct gradients.

Bonus 3: MNIST Training **5 points**

Train your network on real MNIST data and achieve >95% accuracy.

Bonus 4: Visualization **3 points**

Implement a function to visualize computational graphs using graphviz.

10 Tips and Common Pitfalls

1. **Gradient accumulation**: When a value is used multiple times, gradients accumulate. Initialize gradients to 0 and use `+=` in backward.
2. **Topological order**: Process nodes in reverse topological order during backward. A node's backward should only be called after all its consumers have been processed.
3. **Broadcasting**: When shapes differ, numpy broadcasts. Your backward must "unbroadcast" by summing along the broadcasted dimensions.
4. **Numerical stability**: In softmax and cross-entropy, subtract the maximum value before exponentiating to prevent overflow.
5. **In-place operations**: Be careful with in-place numpy operations—they can corrupt gradient computation. Always create new arrays.
6. **Zero gradients**: Remember to zero gradients before each backward pass, or they will accumulate across iterations.

11 References

- Karpathy's micrograd: https://github.com/karpathy/micrograd
- PyTorch autograd documentation: https://pytorch.org/docs/stable/autograd.html
- Stanford CS231n backpropagation notes: https://cs231n.github.io/optimization-2/
- Baydin et al., "Automatic Differentiation in Machine Learning: a Survey"

Good luck, and enjoy building your own autodiff engine!

Test	Description	Points
Part 1: Scalars		**15**
test_value_creation	Value class creation	2
test_value_add	Addition	2
test_value_mul	Multiplication	2
test_value_neg_sub	Negation and subtraction	3
test_value_pow	Power operation	3
test_value_div	Division	3
Part 2: Backward		**20**
test_backward_add	Addition backward	3
test_backward_mul	Multiplication backward	3
test_backward_ops	Other operations backward	4
test_backward_full	Full backward pass	5
test_grad_accumulation	Gradient accumulation	3
test_grad_of_grad	Second derivatives (bonus)	2
Part 3: Operations		**15**
test_relu	ReLU activation	3
test_exp	Exponential	3
test_log	Natural logarithm	3
test_tanh	Hyperbolic tangent	3
test_complex_expr	Complex expressions	3
Part 4: Tensors		**22**
test_tensor_creation	Tensor creation	2
test_tensor_elementwise	Element-wise operations	3
test_tensor_backward	Backward pass	2
test_tensor_broadcast	Broadcasting	4
test_tensor_sum	Sum reduction	3
test_tensor_mean	Mean reduction	2
test_tensor_max	Max reduction	3
test_tensor_reshape	Reshape	2
test_tensor_transpose	Transpose	1
Part 5: Matrix Ops		**17**
test_matmul	Matrix multiplication	5
test_batched_matmul	Batched matmul	4
test_matvec	Matrix-vector products	3
test_tensor_activations	Tensor activations	2
test_matrix_chain	Chain of operations	3
Part 6: Neural Networks		**15**
test_linear_layer	Linear layer	2
test_softmax	Softmax function	3
test_cross_entropy	Cross-entropy loss	3
test_sgd	SGD optimizer	2
test_mlp_training	MLP training	3
test_gradient_check	Gradient checking	2
Total		**104**