Assignment

This module shows how to build the first version of MiniTorch using only simple values and functions. This covers key aspects of auto-differentiation: the key technique in the system. Then you will use your code to train a preliminary model.

Guides

- Derivatives
- Scalar
- Autodifferentiation
- Backpropagation

Task 1.1: Numerical Derivatives

Implement scalar numerical derivative calculation. This function will not be used in the main library but will be critical for testing the whole module.

> **Todo**
>
> Complete the following function in `minitorch/autodiff.py` and pass tests marked as `task1_1`.

```
minitorch.autodiff.central_difference(f: Any, *vals: Any, arg: int = 0, epsilon: float = 1e-06) -> Any
```

Computes an approximation to the derivative of `f` with respect to one arg.

See :doc: `derivative` or https://en.wikipedia.org/wiki/Finite_difference for more details.

Parameters:

- **f** – arbitrary function from n-scalar args to one value
- ***vals** – n-float values $x_0 \ldots x_{n-1}$
- **arg** – the number i of the arg to compute the derivative
- **epsilon** – a small constant

Returns:

- `Any` – An approximation of $f_i'(x_0, \ldots, x_{n-1})$

Task 1.2: Scalars

Implement the overridden mathematical functions required for the `minitorch.Scalar` class. Each of these requires wiring the internal Python operator to the correct `minitorch.Function.forward` call.

Read the example ScalarFunctions that we have implemented for guidelines. You may find it useful to reuse the operators from Module 0.

We have built a debugging tool for you to observe the workings of your expressions to see how the graph is built. You can run it in the *Autodiff Sandbox*. You can alter the expression at the top of the file and then run the code to create a graph in Streamlit:

```
streamlit run app.py -- 1
```

Todo

Complete the following functions in `minitorch/scalar_functions.py`.

minitorch.scalar_functions.Mul.forward(ctx: Context, a: float, b: float) -> float
staticmethod

minitorch.scalar_functions.Inv.forward(ctx: Context, a: float) -> float staticmethod

minitorch.scalar_functions.Neg.forward(ctx: Context, a: float) -> float staticmethod

minitorch.scalar_functions.Sigmoid.forward(ctx: Context, a: float) -> float
staticmethod

minitorch.scalar_functions.ReLU.forward(ctx: Context, a: float) -> float staticmethod

minitorch.scalar_functions.Exp.forward(ctx: Context, a: float) -> float staticmethod

minitorch.scalar_functions.LT.forward(ctx: Context, a: float, b: float) -> float
staticmethod

minitorch.scalar_functions.EQ.forward(ctx: Context, a: float, b: float) -> float
staticmethod

> **Todo**
>
> Complete the following function in `minitorch/scalar.py`, and pass tests marked as `task1_2`. See Python numerical overrides for the interface of these methods. All of these functions should return `minitorch.Scalar` arguments.

```
minitorch.scalar.Scalar.__lt__(b: ScalarLike) -> Scalar

minitorch.scalar.Scalar.__gt__(b: ScalarLike) -> Scalar

minitorch.scalar.Scalar.__sub__(b: ScalarLike) -> Scalar

minitorch.scalar.Scalar.__neg__() -> Scalar

minitorch.scalar.Scalar.__add__(b: ScalarLike) -> Scalar

minitorch.scalar.Scalar.log() -> Scalar

minitorch.scalar.Scalar.exp() -> Scalar

minitorch.scalar.Scalar.sigmoid() -> Scalar

minitorch.scalar.Scalar.relu() -> Scalar
```

Task 1.3: Chain Rule

Implement the chain_rule function in Scalar for functions of arbitrary arguments. This function should be able to backward process a function by passing it in a context and d and then collecting the local derivatives. It should then pair these with the right variables and return them. This function is also where we filter out constants that were used on the forward pass, but do not need derivatives.

> **Todo**
>
> Complete the following function in `minitorch/scalar.py`, and pass tests marked as `task1_3`.

```
minitorch.scalar.Scalar.chain_rule(d_output: Any) -> Iterable[Tuple[Variable, Any]]
```

Task 1.4: Backpropagation

Implement backpropagation. Each of these requires wiring the internal Python operator to the correct `minitorch.Function.backward` call.

Read the example ScalarFunctions that we have implemented for guidelines. Feel free to also consult differentiation rules if you forget how these identities work.

> **Todo**
>
> Complete the following functions in `minitorch/autodiff.py` and `minitorch/scalar.py`, and pass tests marked as `task1_4`.

`minitorch.topological_sort(variable: Variable) -> Iterable[Variable]`

Computes the topological order of the computation graph.

Parameters:

- **variable** (`Variable`) – The right-most variable

Returns:

- `Iterable[Variable]` – Non-constant Variables in topological order starting from the right.

`minitorch.backpropagate(variable: Variable, deriv: Any) -> None`

Runs backpropagation on the computation graph in order to compute derivatives for the leave nodes.

Parameters:

- **variable** (`Variable`) – The right-most variable
- **deriv** ( ) – Its derivative that we want to propagate backward to the leaves.

No return. Should write to its results to the derivative values of each leaf through `accumulate_derivative`.

`minitorch.scalar_functions.Mul.backward(ctx: Context, d_output: float) -> Tuple[float, float]` `staticmethod`

`minitorch.scalar_functions.Inv.backward(ctx: Context, d_output: float) -> float` `staticmethod`

`minitorch.scalar_functions.Neg.backward(ctx: Context, d_output: float) -> float` `staticmethod`

`minitorch.scalar_functions.Sigmoid.backward(ctx: Context, d_output: float) -> float` `staticmethod`

```
minitorch.scalar_functions.ReLU.backward(ctx: Context, d_output: float) -> float
```

staticmethod

```
minitorch.scalar_functions.Exp.backward(ctx: Context, d_output: float) -> float
```

staticmethod

Task 1.5: Training

If your code works, you should now be able to run the training script. Study the code in `project/run_scalar.py` carefully to understand what the neural network is doing.

You will also need Module code to implement the parameters `Network` and for `Linear`. You can modify the dataset and the module with the parameters at the bottom of the file. Start with this simple config:

```
PTS = 50
DATASET = minitorch.datasets["Simple"](PTS)
HIDDEN = 2
RATE = 0.5
```

You can then move up to something more complex, for instance:

```
PTS = 50
DATASET = minitorch.datasets["Xor"](PTS)

HIDDEN = 10
RATE = 0.5
```

If your code is successful, you should be able to run the full visualization:

```
streamlit run app.py -- 1
```

Todo

Train a scalar model for each of the 4 main datasets.

Add the output training logs and final images to your README file.