JacrevFinite: Finite Difference approach to replacing PyTorch Autograd

Brendan Hendrata[1]

[1]A*STAR Bioinformatics Institute
*Contact: `bhendrat@andrew.cmu.edu`

30th June 2024

Abstract

JacrevFinite computes the Jacobian matrix of a given function, such as neural networks, with respect to its inputs using finite difference. It provides an alternative to symbolic computation for differentiation such as the torch.func.jacrev or torch.autograd.

Keywords: finite difference, automatic differentiation, PyTorch, autograd, neural network backpropagation

1. Introduction

The computation of derivatives is fundamental to numerous fields, particularly in machine learning, where gradients are essential for the optimization of neural networks. Traditional methods like symbolic differentiation and automatic differentiation are powerful and accurate, but can be limited by the complexity and non-differentiability of certain functions. This project introduces JacrevFinite, an alternative leveraging the finite difference method to compute the Jacobian matrix of functions, including neural networks, with respect to their inputs.

This project showcases the code developed for JacrevFinite, illustrating its methodology, implementation details, and practical applications. By implementing a finite difference approach, JacrevFinite enhances the versatility and applicability of gradient-based optimization in machine learning models. We compare its performance and accuracy against existing differentiation techniques, demonstrate its application in neural network training, and discuss the scenarios where it outperforms or complements other methods.

2. Function

```
JacrevFinite(*, function, num_args, wrapper=None, dim=None,
             override_dim_constraint=False, delta=1e-5, method='plus')
             (*args)
```

`JacrevFinite` computes the Jacobian of a given `function` with respect to the `args` at the specified index, `num_args`, using finite differences, providing a direct alternative to `torch.func.jacrev`.

Initialization and Parameters

The `JacrevFinite` class is initialized with the following parameters, which define the function to differentiate, the arguments to consider, and various optional settings to control the behavior of the finite difference computation.

```
class JacrevFinite:
    def __init__(self, *, function, num_args, wrapper=None, dim=None,
    delta=1e-5, override_dim_constraint=False, method='plus'):
        """
        Initialize JacrevFinite object.
        """
        assert isinstance(num_args, int), 'num_args must be int'
        assert isinstance(dim, int) or dim is None, 'dim must be int or None'
        assert isinstance(override_dim_constraint, bool),
        'override_dim_constraint must be bool'
        assert method in ['plus', 'minus'], 'method must be \'plus\' or
        \'minus\'

        self.function = function
        self.wrapper = wrapper
        self.num_args = num_args
        self.delta = delta
        self.dim = dim
        self.override = override_dim_constraint
        self.method = method
```

- **function (function)**: A Python function that takes one or more arguments and returns a single tensor. *Note: the function must have only one output.*
- **num_args (int)**: Index of the arguments to compute the Jacobian with respect to.

- **wrapper (function, optional)**: A function to convert `*args` into inputs for the main function, used when the main function cannot directly accept `*args`. The wrapper should return a list of transformed inputs. **Default: None**

- **dim (int, optional)**: Specifies the dimension to append batches over. If None, a singleton dimension at dimension 0 is added. **Default: None**

- **override_dim_constraint (bool, optional)**: Allows overriding the constraint that all input arguments must have the same number of dimensions. **Default: False**

- **delta (float, optional)**: Step size used for finite difference computations. The most stable delta values are between 1e-4 and 1e-5. **Default: 1e-5**

- **method (str, optional)**: Either 'plus' or 'minus'. Specifies whether delta should be added or subtracted for finite difference computations. Both methods should yield similar results but can be interchanged if accuracy is sub-par. **Default: 'plus'**

Returns

Returns the Jacobian of the function with respect to the arguments at index `num_args`.

3. How it works

`JacrevFinite` works by creating batch tensors with slight perturbations, by adding or subtracting delta, in the input values. It then calculates the differences in the function's output to approximate the Jacobian. Each step of this process is methodically handled by 4 specific functions within the class.

Mathematical Illustration

Consider a tensor $\mathbf{t}$ with values:

$$\mathbf{t} = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix}$$

The identity matrix scaled by $\delta = 0.00001$ is:

$$\delta\mathbf{I} = \delta \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0.00001 & 0 & 0 \\ 0 & 0.00001 & 0 \\ 0 & 0 & 0.00001 \end{bmatrix}$$

When added to the repeated tensor $\mathbf{t}$, it results in:

$$\mathbf{t}+\delta\mathbf{I} = \begin{bmatrix} t_1 & t_2 & t_3 \\ t_1 & t_2 & t_3 \\ t_1 & t_2 & t_3 \end{bmatrix} + \begin{bmatrix} 0.00001 & 0 & 0 \\ 0 & 0.00001 & 0 \\ 0 & 0 & 0.00001 \end{bmatrix} = \begin{bmatrix} t_1 + 0.00001 & t_2 & t_3 \\ t_1 & t_2 + 0.00001 & t_3 \\ t_1 & t_2 & t_3 + 0.00001 \end{bmatrix}$$

The output tensor from the function, $\mathbf{f}(\mathbf{t})$, can be computed for each perturbed input:

$$\mathbf{f}(\mathbf{t} + \delta\mathbf{I_i}) = \begin{bmatrix} f(t_1 + 0.00001, t_2, t_3) \\ f(t_1, t_2 + 0.00001, t_3) \\ f(t_1, t_2, t_3 + 0.00001) \end{bmatrix}$$

The Jacobian matrix is then approximated by computing the finite differences:

$$\mathbf{J} = \frac{\mathbf{f}(\mathbf{t} + \delta\mathbf{I}) - \mathbf{f}(\mathbf{t})}{\delta}$$

For example, if $\mathbf{f}(\mathbf{t})$ is the original function output:

$$\mathbf{f}(\mathbf{t}) = \begin{bmatrix} f(t_1, t_2, t_3) \\ f(t_1, t_2, t_3) \\ f(t_1, t_2, t_3) \end{bmatrix}$$

The finite difference approximation for each element in the Jacobian is:

$$\mathbf{J} \approx \begin{bmatrix} \frac{f(t_1+0.00001,t_2,t_3)-f(t_1,t_2,t_3)}{0.00001} & \frac{f(t_1,t_2+0.00001,t_3)-f(t_1,t_2,t_3)}{0.00001} & \frac{f(t_1,t_2,t_3+0.00001)-f(t_1,t_2,t_3)}{0.00001} \end{bmatrix}$$

Code Implementation Examples

Here are two examples of how to use `JacrevFinite` in practice:

Example 1: Simple Addition Function

Consider a simple function that adds two inputs together. We want to compute the Jacobian of this function with respect to the first input:

```
def f(x, y):
    return x + y

input1 = (1, 1)
input2 = [2, 3]

# Compute the Jacobian using JacrevFinite
jacobian = JacrevFinite(function=f, num_args=0)(input1, input2)
```

In this example, JacrevFinite computes the Jacobian with respect to the first input input1, treating the second input input2 as a constant during the differentiation.

Example 2: Neural Network Forward Pass

In more complex scenarios, such as computing the Jacobian for a neural network with multiple inputs, JacrevFinite can handle the computation effectively:

```
class Update():
    def forward(self, x, y, z):
        ...
        return output

input1 = torch.randn(2, 3)
input2 = torch.randn(2, 3)
input3 = torch.randn(2, 3)

function = Update()

# Compute the Jacobian using JacrevFinite
jacobian = JacrevFinite(function=function.forward, num_args=0)
(input1, input2, input3)
```

In this example, JacrevFinite computes the Jacobian of the forward pass of a neural network with respect to the first input input1, while treating input2 and input3 as constants.

4. Computational Workflow in JacrevFinite

The JacrevFinite class is designed to compute the Jacobian matrix of a given function using finite differences. The core of this functionality is implemented in the __call__ method, which orchestrates the process of perturbing the inputs, applying the function, and calculating the Jacobian. Below is the implementation of the __call__ method:

```
def __call__(self, *args):
    """
    Performs computation.

    Args:
        *args: Input arguments.

    Returns:
```

```
    Tensor: Jacobian matrix.
"""
assert self.num_args < len(args), 'invalid num_args'

# Converts inputs to a list of tensors
if len(args) == 1:
    self.inputs = args[0]
    if not isinstance(self.inputs, Tensor):
        self.inputs = torch.tensor(self.inputs)
    self.inputs = self.inputs.unsqueeze(0).to(torch.float64)
    self.inputs = list(self.inputs)

else:
    self.inputs = [inputs if isinstance(inputs, Tensor) else \
                   torch.tensor(inputs, dtype=torch.float64) for
                   inputs in args]

# Checks that all the tensors have the same number of dimensions
if self.override is False:
    first_dim = self.inputs[0].dim()
    for tensor in self.inputs:
        assert tensor.dim() == first_dim, f"Tensor {tensor} has a
        different number of dimensions: \
            {tensor.dim()} vs {first_dim}"

self.output_dim = self.get_outputdim()

# Forward passes
input1 = self.delta_forward() # changes self.inputs
input2 = self.wrapper_forward(input1)
output = self.func_forward(input2)
jacobian = self.jacobian_forward(output)

return jacobian
```

4.1 Method Explanation

The `__call__` method performs the following key steps:

- **Input Validation and Conversion:** The method first checks that the specified `num_args` index is valid relative to the number of input arguments. It then converts the input arguments to tensors if they are not already, and ensures all inputs have the same number of dimensions.

- **Dimension Consistency Check:** If the `override_dim_constraint` is set to `False`, the method checks that all input tensors have the same number of dimensions to maintain consistency during Jacobian computation.

- **Forward Passes:** The method proceeds through several forward passes:

  - `delta_forward`: Perturbs the inputs by adding or subtracting delta to create a batch of tensors.
  - `wrapper_forward`: (Optional) Applies a wrapper function to preprocess the inputs.
  - `func_forward`: Passes the processed inputs through the main function to obtain the output.
  - `jacobian_forward`: Computes the Jacobian matrix by calculating the finite differences of the output with respect to the perturbed inputs.

- **Output:** Finally, the computed Jacobian matrix is returned.

This method encapsulates the main computational workflow of the `JacrevFinite` class, ensuring that the Jacobian matrix is accurately and efficiently computed using finite differences.

4.2 delta_forward Method

The `delta_forward` method is a critical component of the `JacrevFinite` class, responsible for generating perturbed versions of the input tensors. These perturbations are used to approximate the Jacobian matrix through finite difference computations. The method operates through the following steps:

- **Specify Tensor to Append Delta:** The method begins by selecting the tensor on which delta will be applied, based on the `num_args` index.

```
tensor = self.inputs[self.num_args]
```

- **Add Singleton Dimension:** If the `dim` parameter is set to `None`, a singleton dimension is added at dimension 0 of the tensor. Otherwise, the specified dimension is used.

```
if self.dim is None:
    tensor = tensor.clone().unsqueeze(0)
    dim = 0
else:
    dim = self.dim
```

- **Ensure Correct Dimension Size:** The method checks that the size of the tensor along the specified dimension is 1, ensuring the tensor can be properly concatenated with the perturbed versions.

```
assert tensor.size(dim) == 1, 'wrong dimension to append batch over, size mus
```

- **Determine Number of Repetitions:** The number of repetitions and the number of dimensions in the tensor are determined to prepare for batch creation.

```
num_rep = tensor.view(-1).size(0)
num_dim = tensor.dim()
```

- **Define Repeat Dimensions:** The tensor is repeated `num_rep` times along the specified dimension to create multiple copies that will be perturbed.

```
repeat_dim = torch.ones(num_dim, dtype=int).tolist()
repeat_dim[dim] = num_rep
```

- **Define Reshape and Permute Dimensions:** The tensor shape is adjusted by removing the specified dimension and inserting it at the end. Permute dimensions are also configured for proper alignment.

```
reshape_dim = list(tensor.shape)
reshape_dim.pop(dim)
reshape_dim.insert(len(reshape_dim), num_rep)

permute_dim = range(num_dim)
permute_dim = [num if num < dim else num - 1 for num in permute_dim]
permute_dim[dim] = num_dim - 1
```

- **Repeat Tensor and Add Delta:** The tensor is repeated, and an identity matrix scaled by delta is created and reshaped to match the repeated tensor. Depending on the method (either 'plus' or 'minus'), the delta tensor is either added to or subtracted from the repeated tensor.

```
repeated_tensor = tensor.repeat(repeat_dim)

delta_tensor = torch.eye(num_rep, dtype=tensor.dtype, device=tensor.device) *
delta_tensor = delta_tensor.reshape(reshape_dim).permute(permute_dim)

if self.method == 'plus':
    append_tensor = repeated_tensor + delta_tensor
else:
    append_tensor = repeated_tensor - delta_tensor
```

This is in-line with the mathematical construct explained previously: Assuming tensor $\mathbf{t}$ with values:

$$\mathbf{t} = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix}$$

and $\delta = 0.00001$. The identity matrix scaled by δ is:

$$\delta\mathbf{I} = \delta \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0.00001 & 0 & 0 \\ 0 & 0.00001 & 0 \\ 0 & 0 & 0.00001 \end{bmatrix}$$

When added to the repeated tensor $\mathbf{t}$, the result is:

$$\mathbf{t} + \delta\mathbf{I} = \begin{bmatrix} t_1 & t_2 & t_3 \\ t_1 & t_2 & t_3 \\ t_1 & t_2 & t_3 \end{bmatrix} + \begin{bmatrix} 0.00001 & 0 & 0 \\ 0 & 0.00001 & 0 \\ 0 & 0 & 0.00001 \end{bmatrix}$$

$$= \begin{bmatrix} t_1 + 0.00001 & t_2 & t_3 \\ t_1 & t_2 + 0.00001 & t_3 \\ t_1 & t_2 & t_3 + 0.00001 \end{bmatrix}$$

- **Concatenate with Original Tensor:** The perturbed tensors are concatenated with the original tensor along the specified dimension to create a batch tensor. The batch size is then updated.

```
batch_tensor = torch.cat((tensor, append_tensor), dim=dim)
self.batch_size = batch_tensor.size(dim)
```

- **Replace Inputs with Batch Tensor:** The original list of inputs is updated to include the batch tensor. Other tensors are repeated to match the batch size, ensuring consistency across all inputs.

```
    inputs_copy = self.inputs.copy()
    inputs_copy.pop(self.num_args)

    new_inputs = []

    for input_tensor in inputs_copy:
        input_tensor = input_tensor.clone()

        if self.dim is None:
            input_tensor = input_tensor.unsqueeze(0)

        repeat_shape = [1] * input_tensor.dim()
        repeat_shape[dim] = self.batch_size
        repeated_tensor = input_tensor.repeat(*repeat_shape)

        new_inputs.append(repeated_tensor)

    new_inputs.insert(self.num_args, batch_tensor)

    self.new_dim = dim
    return new_inputs
```

4.3 wrapper_forward Method

The `wrapper_forward` method applies the wrapper function to the input list generated by the `delta_forward` method. This step is essential for scenarios where the main function cannot directly accept the perturbed inputs created by `delta_forward`. The wrapper function transforms these inputs into a format that the main function can process. The method operates as follows:

- **Apply Wrapper Function:** The method takes the list of inputs from `delta_forward` as its argument.

```
    def wrapper_forward(self, input1):
```

- **Check for Wrapper Function:** It checks whether a wrapper function has been provided. If no wrapper function is specified (`self.wrapper` is `None`), the method simply returns the original inputs.

```
    if self.wrapper is None:
        input2 = input1
```

- **Apply the Wrapper Function:** If a wrapper function is provided, it is applied to the inputs. The wrapper function is expected to take the inputs as arguments and return a list, tuple, or other iterable structure that is compatible with the main function.

```
    else:
        input2 = self.wrapper(*input1)
```

- **Return Transformed Inputs:** Finally, the method returns the transformed inputs, which will be used by the main function in the subsequent steps.

```
    return input2
```

The need for this method arises from the requirement to handle complex input transformations that cannot be directly managed by the main function. By using a wrapper function, we can preprocess the inputs to fit the expected format of the main function. This flexibility is particularly useful in scenarios where the main function expects inputs in a specific structure or when additional preprocessing is necessary before the main computation.

For example, consider a main function that requires inputs to be combined or reshaped in a particular manner. The wrapper function can handle these transformations, ensuring that the inputs are correctly formatted before being passed to the main function.

Example Usage:

Assume we have a main function that expects inputs as a concatenated tensor, but the original inputs are separate tensors. The wrapper function can be designed to concatenate these tensors before passing them to the main function:

```
def wrapper(*args):
    return torch.cat(args, dim=0)
```

By using `wrapper_forward`, we ensure that the inputs are correctly preprocessed, allowing the main function to operate on them without any issues.

```
# Example usage within JacrevFinite
jacrev_finite = JacrevFinite(function=main_function, num_args=0,
wrapper=wrapper)
```

4.4 func_forward Method

The func_forward method is responsible for applying the main function to the inputs processed by the wrapper_forward method. This method is crucial as it computes the actual output of the main function, which will be used in the finite difference calculations for the Jacobian matrix. The method operates as follows:

- **Apply Main Function:** The method takes the list, tuple, or iterable of inputs from wrapper_forward as its argument.

  ```
  def func_forward(self, input2):
  ```

- **Compute Output:** It applies the main function to the provided inputs using argument unpacking. This ensures that each element in the input list is passed as a separate argument to the main function.

  ```
  output = self.function(*input2)
  ```

- **Return Output:** Finally, the method returns the output of the main function, which is a tensor. This output is essential for the subsequent finite difference computations that approximate the Jacobian matrix.

  ```
  return output
  ```

The purpose of this method is to isolate the application of the main function from other steps in the Jacobian computation process. By doing so, it ensures that the inputs are correctly processed and that the main function's output is accurately obtained. This modular approach enhances code readability and maintainability.

The following example illustrates how the func_forward method fits into the overall workflow of JacrevFinite:

```
# Example usage within JacrevFinite
input1 = self.delta_forward()            # Step 1: Create perturbed inputs
input2 = self.wrapper_forward(input1)     # Step 2: Apply wrapper function
output = self.func_forward(input2)  # Step 3: Compute main function output
```

This method ensures that the main function receives the correctly processed inputs and returns the necessary output for further computations.

Detailed Example:

Assume we have a main function f that takes two arguments and returns their product. The func_forward method will apply this function to the processed inputs:

```
def f(x, y):
    return x * y

# In JacrevFinite class
self.function = f

# Processed inputs from wrapper_forward
input2 = [torch.tensor([2]), torch.tensor([3])]

# Compute output
output = self.func_forward(input2)
>>> output: tensor([6])
```

In this example, the `func_forward` method applies the main function `f` to the inputs `x` and `y`, and correctly computes their product. This output will then be used in the finite difference calculations for the Jacobian matrix.

4.5 jacobian_forward Method

The `jacobian_forward` method computes the Jacobian matrix by using the finite difference method. This method is essential for obtaining the derivatives of the output tensor with respect to the input tensor. The steps involved in this method are detailed as follows:

- **Compute Values for Reshape and Permutation:** The method begins by determining the shapes of the input and output tensors. These shapes are used to initialize the Jacobian matrix's shape and to set up the reshaping and permutation steps.

  ```
  input_delta_shape = list(self.inputs[self.num_args].shape)
  output_shape = self.output_dim
  jacobian_init = input_delta_shape + output_shape

  input_len = len(input_delta_shape)
  output_len = len(output_shape)
  ```

- **Determine Finite Difference Dimension:** The dimension over which the finite difference is computed is identified. This dimension corresponds to the batch size, which was set during the `delta_forward` method.

  ```
  batch_output_shape = list(output.shape)
  dim = batch_output_shape.index(self.batch_size)
  ```

- **Finite Difference Calculation:** The method calculates the finite difference to obtain the Jacobian. This is done by subtracting the reference output (obtained from the unperturbed input) from each of the perturbed outputs and dividing by delta.

  ```
  ref = output.select(dim, 0)
  output_transposed = output.transpose(0, dim)
  jacobian = (output_transposed[1:] - ref) / self.delta
  ```

  To visualize this, consider the output tensor $\mathbf{y}$ and the perturbed outputs $\mathbf{y}_i$ where $\mathbf{y}_i$ is the output with the i-th input perturbed by δ. The finite difference approximation for the Jacobian is:

  $$\mathbf{J}_{ij} = \frac{\partial y_i}{\partial x_j} \approx \frac{y_i(x_j + \delta) - y_i(x_j)}{\delta}$$

- **Reshape and Permute the Jacobian:** The computed Jacobian matrix is then reshaped and permuted to match the desired shape and order. This step ensures that the Jacobian has the correct dimensions and structure for further use.

  ```
  jacobian = jacobian.reshape(jacobian_init)
  permute_order = list(range(input_len, input_len + output_len)) +
  list(range(input_len))
  jacobian = jacobian.permute(*permute_order)
  ```

- **Handle Negative Delta:** If the method specified for finite differences is ’minus’, the Jacobian matrix is negated to reflect this.

  ```
  if self.method == ’minus’:
      jacobian = torch.neg(jacobian)
  ```

- **Return Jacobian:** Finally, the computed Jacobian matrix is returned.

  ```
  return jacobian
  ```

The `jacobian_forward` method ensures accurate computation of the Jacobian matrix by leveraging finite difference approximations. The following example illustrates the overall workflow of `JacrevFinite`, showing how the `jacobian_forward` method fits into the sequence of operations:

```
# Example usage within JacrevFinite
input1 = self.delta_forward()            # Step 1: Create perturbed inputs
input2 = self.wrapper_forward(input1)     # Step 2: Apply wrapper function
output = self.func_forward(input2)  # Step 3: Compute main function output
jacobian = self.jacobian_forward(output) # Step 4: Compute Jacobian matrix
```

This detailed explanation covers the purpose and functionality of the `jacobian_forward` method, providing insights into the finite difference calculations and how the method fits into the overall process of computing the Jacobian matrix.

4.6 Overall Code

The `JacrevFinite` class is designed to compute the Jacobian matrix of a given function with respect to its inputs using the finite difference method. This class offers a practical and flexible alternative to symbolic and automatic differentiation methods. Below is the complete code for the `JacrevFinite` class, followed by detailed explanations of its methods.

```
import torch
from torch import Tensor

class JacrevFinite:
    def __init__(self, *, function, num_args, wrapper=None, dim=None, delta=1e-5,
    override_dim_constraint=False, method=’plus’):
        """
        Initialize JacrevFinite object.

        Args:
            function (callable): Function that takes one or more arguments and
                returns a single tensor.
            num_args (int): Index of the arguments to compute the Jacobian with
                respect to
            wrapper (callable, optional): Function to convert *args into inputs
                for main function, used when main function cannot directly accept
                *args.
                Wrapper should return list of transformed inputs. Default: None
            dim (int, optional): Specifies the dimension to append batches over.
                If None, a singleton dimension at dimension 0 is added.
                Must be a singleton dimension.
            delta (float, optional): Step size used for finite difference
                computations. Most stable at 1e-5 or 1e-4. Default: 1e-5
            override_dim_constraint (bool, optional): Overrides constraint that
                input arguments must have same number of dimensions.
                Default: False
            method (str, optional): Either 'plus' or 'minus'. Specifies whether
                delta should be added or subtracted for finite difference
                computations.
                Both methods should yield similar results but can be interchanged
                if accuracy is sub-par. Default: 'plus'

        Constraints:
            Inputs must have the same number of dimensions (.dim() must be equal)
            Function must only have one output

        Raises:
            AssertionError: If num_args is not an int.
            AssertionError: If dim is not an int or None.
            AssertionError: if override_dim_constraint is not bool.
            AssertionError: If method is not 'plus' or 'minus'
        """
        assert isinstance(num_args, int), 'num_args must be int'
        assert isinstance(dim, int) or dim is None, 'dim must be int or None'
        assert isinstance(override_dim_constraint, bool), \
        'override_dim_constraint must be bool'
        assert method in ['plus', 'minus'], 'method must be \'plus\' or \'minus\''

        self.function = function
        self.wrapper = wrapper
        self.num_args = num_args
        self.delta = delta
        self.dim = dim
        self.override = override_dim_constraint
        self.method = method

    def __call__(self, *args):
        """
        Performs computation.

        Args:
            *args: Input arguments.

        Returns:
            Tensor: Jacobian matrix.
        """
        assert self.num_args < len(args), 'invalid num_args'

        # Converts inputs to a list of tensors
    if len(args) == 1:
        self.inputs = args[0]
        if not isinstance(self.inputs, Tensor):
            self.inputs = torch.tensor(self.inputs)
        self.inputs = self.inputs.unsqueeze(0).to(torch.float64)
        self.inputs = list(self.inputs)

    else:
        self.inputs = [inputs if isinstance(inputs, Tensor) else \
                       torch.tensor(inputs, dtype=torch.float64) for
                       inputs in args]

    # Checks that all the tensors have the same number of dimensions
    if self.override is False:
        first_dim = self.inputs[0].dim()
        for tensor in self.inputs:
            assert tensor.dim() == first_dim, f"Tensor {tensor} has a \
            different number of dimensions: \
                {tensor.dim()} vs {first_dim}"

    self.output_dim = self.get_outputdim()

    # Forward passes
    input1 = self.delta_forward() # changes self.inputs
    input2 = self.wrapper_forward(input1)
    output = self.func_forward(input2)
    jacobian = self.jacobian_forward(output)

    return jacobian

def delta_forward(self):
    """
    Creates batch tensor by repeating input tensors and adding delta to 1
    element per repeated tensor.

    Returns:
        list: List of new inputs with the batch tensor included.
    """
    # Specifies which tensor to append delta over
    tensor = self.inputs[self.num_args]

    if self.dim is None:
        tensor = tensor.clone().unsqueeze(0)  # Add new singleton dimension
        dim = 0  # The dimension along which to concatenate
    else:
        dim = self.dim  # Use the specified dimension

    assert tensor.size(dim) == 1, ’wrong dimension to append batch over, \
    size must = 1’

    num_rep = tensor.view(-1).size(0) # Number of repetitions
    num_dim = tensor.dim() # Number of dimensions in tensor

    # Repeat_dim (num_rep times over dim)
    repeat_dim = torch.ones(num_dim, dtype=int).tolist()
    repeat_dim[dim] = num_rep

    # Reshape_dim (move dim to last value and multiply by appended size)
    reshape_dim = list(tensor.shape)
    reshape_dim.pop(dim)
    reshape_dim.insert(len(reshape_dim), num_rep)

    # Permute_dim (change order of dimensions to move dim to last value)
    permute_dim = range(num_dim)
    permute_dim = [num if num<dim else num-1 for num in
    permute_dim]
    permute_dim[dim] = num_dim-1

    # Operations to add delta onto every single element:

    # Repeat tensor num_rep times over dim
    repeated_tensor = tensor.repeat(repeat_dim)

    # Create identity matrix of size (num_rep x num_rep) multiplied by delta
    then reshape to fit repeated_tensor
    delta_tensor = torch.eye(num_rep, dtype =tensor.dtype, \
    device=tensor.device)*self.delta

    delta_tensor = delta_tensor.reshape(reshape_dim).permute(permute_dim)

    # Add or minus the tensors together
    if self.method == ’plus’:
        append_tensor = repeated_tensor + delta_tensor
    else:
        append_tensor = repeated_tensor - delta_tensor
    # Concatenate with original tensor
    batch_tensor = torch.cat((tensor, append_tensor), dim=dim)
    self.batch_size = batch_tensor.size(dim)

    # Replace inputs with batch_tensor and ensure all tensors have same batch
    size:
    inputs_copy = self.inputs.copy()
    inputs_copy.pop(self.num_args)

    new_inputs = []

    # Repeating other tensors to ensure same batch size
    for input_tensor in inputs_copy:
        input_tensor = input_tensor.clone()

        if self.dim is None:
            input_tensor = input_tensor.unsqueeze(0)

        repeat_shape = [1] * input_tensor.dim()
        repeat_shape[dim] = self.batch_size
        repeated_tensor = input_tensor.repeat(*repeat_shape)

        new_inputs.append(repeated_tensor)

    new_inputs.insert(self.num_args, batch_tensor)

    self.new_dim = dim
    return new_inputs

def wrapper_forward(self, input1):
    """
    Apply the wrapper function to input1.

    Args:
        input1 (list): Input list from delta_forward.

    Returns:
        list/tuple/iterable: Output after applying the wrapper.
    """
    if self.wrapper is None:
        input2 = input1
    else:
        input2 = self.wrapper(*input1)

        return input2

    def func_forward(self, input2):
        """
        Apply the function to input2.

        Args:
            input2 (list/tuple/iterable): Input list from wrapper_forward.

        Returns:
            Tensor: Output of the function.
        """
        output = self.function(*input2)
        return output

    def jacobian_forward(self, output):
        """
        Computes the Jacobian matrix.

        Args:
            output (Tensor): Output from func_forward.

        Returns:
            Tensor: Computed Jacobian matrix.
        """
        # Compute values for reshape and permutation
        input_delta_shape = list(self.inputs[self.num_args].shape)
        output_shape = self.output_dim
        jacobian_init = input_delta_shape + output_shape

        input_len = len(input_delta_shape)
        output_len = len(output_shape)

        # Determine over which dimension to do finite difference (subtract
        and divide delta)
        batch_output_shape = list(output.shape)
        dim = batch_output_shape.index(self.batch_size)

        # Finite difference to obtain Jacobian
        ref = output.select(dim,0)
        output_transposed = output.transpose(0, dim)
        jacobian = (output_transposed[1:] - ref)/self.delta

        # Reshape and permute the Jacobian to the desired shape
        jacobian = jacobian.reshape(jacobian_init)
        permute_order = list(range(input_len, input_len + output_len)) +
        list(range(input_len))
        jacobian = jacobian.permute(*permute_order)

        # For negative delta instance
        if self.method == 'minus':
            jacobian = torch.neg(jacobian)
        return jacobian

    def get_outputdim(self):
        """
        Gets output dimensions for a single batch.
        Used to determine dimensions of Jacobian matrix

        Returns:
            list: The output dimensions.
        """
        inputs = self.wrapper_forward(self.inputs)
        output = self.func_forward(inputs)
        output_dim = list(output.shape)

        return output_dim
```

The `JacrevFinite` class comprises several methods that work together to compute the Jacobian matrix:

- `delta_forward`: Creates batch tensors with slight perturbations.
- `wrapper_forward`: Applies a wrapper function to preprocess the inputs.
- `func_forward`: Computes the output of the main function using the processed inputs.
- `jacobian_forward`: Computes the Jacobian matrix using finite differences.

Each of these methods plays a crucial role in ensuring accurate and efficient computation of the Jacobian matrix, providing a robust alternative to traditional differentiation techniques.

5. Errors and Discrepancies

5.1 Delta Instability

The accuracy of finite difference methods in computing derivatives is highly sensitive to the choice of the delta value. This section examines the impact of varying delta on the accuracy of the Jacobian matrix computations performed by `JacrevFinite`.

To illustrate this, we compare the finite difference method implemented in `JacrevFinite` with PyTorch's automatic differentiation (`autograd`). We evaluate the mean and maximum errors between the Jacobians obtained from these two methods as delta varies. The following code demonstrates the setup and computation:

```
import torch
import torch.nn as nn
from torch.func import jacrev
import matplotlib.pyplot as plt
from jacrev_finite import JacrevFinite

# Define a simple neural network model
class Network(nn.Module):
    def __init__(self, n_input, n_output, n_hidden):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(n_input, n_hidden),
            nn.ReLU(),
            nn.Linear(n_hidden, n_hidden),
            nn.ReLU(),
            nn.Linear(n_hidden, n_hidden),
            nn.ReLU(),
            nn.Linear(n_hidden, n_output)
        )

    def forward(self, x):
        return self.layers(x)

# Functions to compute mean and max errors
def meanerror(a, b):
    return a.sub(b).abs().mean().item()

def abserror(a, b):
    return a.sub(b).abs().max().item()

# Function to obtain errors for various delta values
def obtain_error(net):
    input = torch.randn((100, 5), dtype=torch.float64)
    deltas = [1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-6, 1e-7, 1e-8, 1e-9, 1e-10]

    auto = jacrev(func=net, argnums=0)(input)

    mean_errors = []
    max_errors = []

    for delta in deltas:
        finite = JacrevFinite(function=net, num_args=0, delta=delta)(input)
        mean_errors.append(meanerror(auto, finite))
        max_errors.append(abserror(auto, finite))

    return deltas, mean_errors, max_errors

# Initialize the network and compute errors
net = Network(5, 5, 128).double()
deltas, mean_errors, max_errors = obtain_error(net)

# Plot the results
plt.figure()
plt.plot(deltas, mean_errors, label='Mean Error', color='blue')
plt.plot(deltas, max_errors, label='Max Error', color='red')
plt.legend()
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Delta')
plt.ylabel('Error')
plt.title('Error vs Delta for Finite Difference Method')
plt.show()
```

The first graph illustrates the relationship between the chosen delta value and the errors in the Jacobian computations. The x-axis represents delta on a logarithmic scale, and the y-axis represents the error (both mean and maximum) also on a logarithmic scale. The mean error is shown in blue, and the maximum error is shown in red.

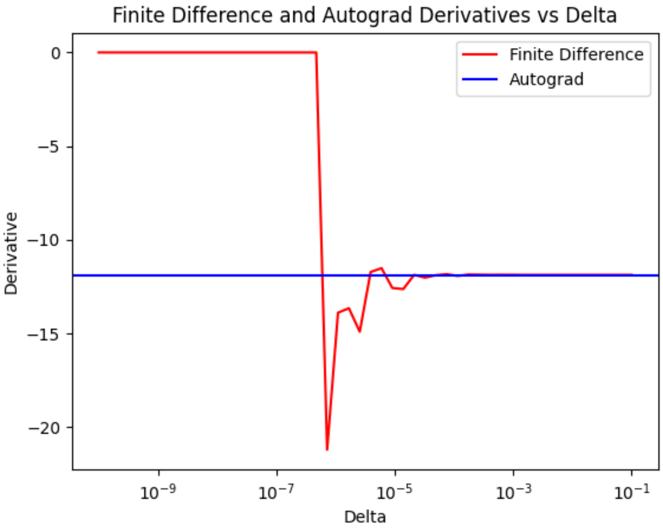

Figure 1: Finite Difference Method and Autograd Derivatives vs. Delta

The second graph presents the error percentage as a function of delta, comparing the errors obtained using `JacrevFinite` and PyTorch's `autograd`.

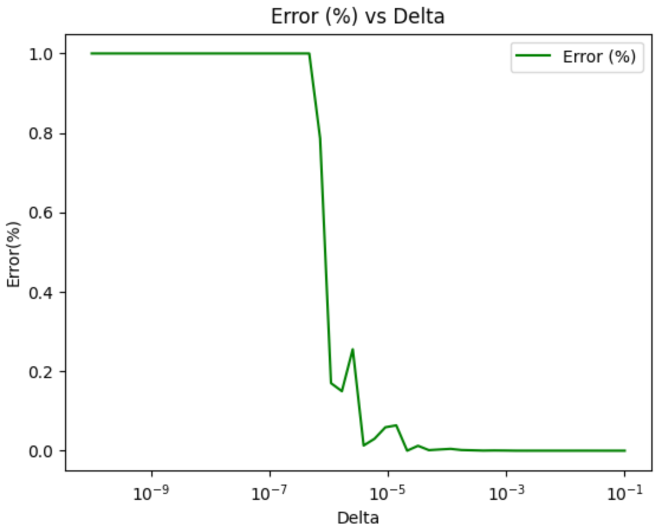

Figure 2: Error (%) vs. Delta for Autograd vs. Finite Difference Methods

As seen from the graphs, the accuracy of the finite difference method is maximized around $\delta = 1e-4$ to $1e-5$. Choosing delta values outside this range can lead to significant errors due to either too coarse or too fine perturbations, demonstrating the instability of the finite difference method concerning delta.

5.2 Computational Efficiency

In addition to accuracy, computational efficiency is a crucial factor when comparing different methods for computing derivatives. We evaluated the time taken by the finite difference method implemented in `JacrevFinite` compared to PyTorch's automatic differentiation (`autograd`) and `jacrev`. The following functions were tested to measure execution times:

- **Function 1:** A simple linear function `two(x)`.
- **Function 2:** A function `f(x, y)` that sums its inputs.
- **Function 3:** A function `g(x, y)` that multiplies its inputs, tested with a wrapper for input transformations.

The execution times for each method were measured using the `timeit` module in Python. The results are summarized below:

```
Time for jacrev (two): 1.974651 seconds
Time for grad (two): 0.009607 seconds
Time for JacrevFinite (two): 0.034684 seconds

Time for jacrev (f): 0.021442 seconds
Time for grad (f): 0.004371 seconds
Time for JacrevFinite (f): 0.013784 seconds

Time for jacrev (g with wrapper): 0.023883 seconds
Time for JacrevFinite (g with wrapper): 0.015642 seconds
```

These results indicate that `JacrevFinite` generally performs better than `jacrev` in terms of computational efficiency, particularly when using the wrapper function for more complex input transformations. However, `autograd` remains the fastest method for simpler operations.

5.3 Accuracy

To ensure the accuracy of the `JacrevFinite` method, we compared its results against PyTorch's `jacrev` function. Various test cases were evaluated, including simple functions, more complex network forward passes, and varying input dimensions.

Test Case 1: Simple Function

For a simple multiplication function `function(x, y)`, we compared the Jacobian matrices computed by `JacrevFinite` and `jacrev`. The following code snippet illustrates the process:

```
input1 = torch.randn((100, 100), dtype=torch.float64)
input2 = torch.randn((100, 100), dtype=torch.float64)

jacobian_auto = jacrev(func=function, argnums=0)(input1, input2)
jacobian_finite = JacrevFinite(function=function, num_args=0)(input1, input2)

assertTensorEqual(jacobian_auto, jacobian_finite)
```

Test Case 2: Function with Different Dimensions

We tested the accuracy of `JacrevFinite` when appending over different dimensions and methods:

```
input3 = torch.randn((64, 1, 64), dtype=torch.float64)
input4 = torch.randn((64, 1, 64), dtype=torch.float64)

jacobian_auto1 = jacrev(func=function, argnums=0)(input3, input4)
jacobian_finite1 = JacrevFinite(function=function, num_args=0)(input3, input4)
jacobian_finite2 = JacrevFinite(function=function, num_args=0, dim=1, method='minus')(input3,

assertTensorEqual(jacobian_finite1, jacobian_finite2)
assertTensorEqual(jacobian_auto1, jacobian_finite1)
```

Test Case 3: Network Forward Passes

For a more complex scenario, we compared the Jacobian matrices for network forward passes using a neural network. The code below demonstrates this test:

```
net = Network(5, 5, 128).double()

input6 = torch.randn((20, 5), dtype=torch.float64)

jacobian_auto2 = jacrev(func=net, argnums=0)(input6)
jacobian_finite3 = JacrevFinite(function=net, num_args=0)(input6)

assertTensorEqual(jacobian_auto2, jacobian_finite3)
```

We also tested the network with larger input dimensions:

```
net = Network(2, 2, 256).double()

input7 = torch.randn((8, 1, 16, 2), dtype=torch.float64)

jacobian_auto3 = jacrev(func=net, argnums=0)(input7)
jacobian_finite4 = JacrevFinite(function=net, num_args=0, dim=1)(input7)

assertTensorEqual(jacobian_auto3, jacobian_finite4)
```

Results

The output of these tests confirms the accuracy of `JacrevFinite`, as the results closely match those obtained using PyTorch's `jacrev`. The specific results are shown below:

```
True
Error:
mean error: 4.309479631012399e-16, max error: 9.29114563064104e-11

True
Error:
mean error: 1.3039380170592267e-15, max error: 8.881784197001252e-11

True
Error:
mean error: 1.0865026644743119e-15, max error: 9.900658071160251e-11

True
Error:
mean error: 9.729684497912846e-14, max error: 6.905432822779112e-12

True
Error:
mean error: 1.4154864256284452e-14, max error: 8.50340284297424e-12
```

These results demonstrate that `JacrevFinite` provides highly accurate Jacobian matrix computations, with errors well within acceptable tolerances when compared to PyTorch's `jacrev`. This accuracy is maintained across various functions, input dimensions, and network architectures, ensuring reliable performance in diverse scenarios.

Accuracy Testing Code

Below is the code used to test the accuracy of `JacrevFinite`:

```
def assertTensorEqual(a, b, abs_tol=1e-9, mean_tol=1e-9):
    mean = a.sub(b).abs().mean().item()
    max = a.sub(b).abs().max().item()
    isEqual = (max < abs_tol and mean < mean_tol)
    print(isEqual)
    print(f"Error:\nmean error: {mean}, max error: {max}\n")
```

6. Conclusion

In this report, we introduced `JacrevFinite`, a finite difference method for computing the Jacobian matrix of functions, offering a practical alternative to symbolic and automatic differentiation methods in PyTorch. Through various test cases, we demonstrated the accuracy and computational efficiency of `JacrevFinite`, showing that it provides reliable and efficient Jacobian computations across different scenarios. The comparisons with PyTorch's `jacrev` and `autograd` functions confirmed that `JacrevFinite` is both accurate and versatile, making it a valuable tool for gradient-based optimization in machine learning models.

Source code

Source code can be found here: https://github.com/schrodingerslemur/jacrev_finite/blob/main/JacrevFinite.py.

Contributions

Dr Lee Hwee Kwan, **Dr Liu Wei**, and **Dr Park Sojeong**