hessQuik

Release 0.0.1

Elizabeth Newman and Lars Ruthotto

Feb 25, 2023

OVERVIEW

A lightweight package for fast, GPU-accelerated computation of gradients and Hessians of functions constructed via composition.

CHAPTER

ONE

STATEMENT OF NEED

Deep neural networks (DNNs) and other composition-based models have become a staple of data science, garnering state-of-the-art results and gaining widespread use in the scientific community, particularly as surrogate models to replace expensive computations. The unrivaled universality and success of DNNs is due, in part, to the convenience of automatic differentiation (AD) which enables users to compute derivatives of complex functions without an explicit formula. Despite being a powerful tool to compute first-order derivatives (gradients), AD encounters computational obstacles when computing second-order derivatives (Hessians).

Knowledge of second-order derivatives is paramount in many growing fields and can provide insight into the optimization problem solved to build a good model. Hessians are notoriously challenging to compute efficiently with AD and cumbersome to derive and debug analytically. Hence, many algorithms approximate Hessian information, resulting in suboptimal performance. To address these challenges, hessQuik computes Hessians analytically and efficiently with an implementation that is accelerated on GPUs.

1.1 Getting Started

Once `hessQuik` is installed, you can import as follows:

```
import hessQuik.activations as act
import hessQuik.layers as lay
import hessQuik.networks as net
```

You can construct a `hessQuik` network from layers as follows:

```
d = 10 # dimension of the input features
widths = [32, 64] # hidden channel dimensions
f = net.NN(lay.singleLayer(d, widths[0], act.antiTanhActivation()),
    lay.resnetLayer(widths[0], h=1.0, act.softplusActivation()),
    lay.singleLayer(widths[0], widths[1], act.quadraticActivation())
    )
```

You can obtain gradients and Hessians via:

```
nex = 20 # number of examples
x = torch.randn(nex, d)
fx, dfx, d2fx = f(x, do_gradient=True, do_Hessian=True)
```

That's it! You now have computed the value, gradient, and Hessian of the network f at the point x.

1.2 Installation

To install from PyPI, use the following from the command line:

```
pip install hessQuik
```

To install from Github, use the following from the command line:

```
python -m pip install git+https://github.com/elizabethnewman/hessQuik.git
```

The following are the `hessQuik` dependencies:

- `torch` (with the recommended version >= 1.10.0, but code will run with version >= 1.5.0)

The dependencies are installed automatically with `pip`.

1.3 Contributing

There are many ways to contribute to `hessQuik`.

1.3.1 Contributing Software

1. Fork the `hessQuik` repository
2. Clone your fork using the command:

   ```
   git clone https://github.com/<username>/hessQuik.git
   ```

3. Contribute to your forked repository.
4. Create a pull request.

If your code passes the necessary numerical tests and is well-documented, your changes and/or additions will be merged in the main `hessQuik` repository.

You can find examples of the tests used in each file and related unit tests the tests directory.

1.3.2 Reporting Issues

If you notice an issue with this repository, please report it using Github Issues. When reporting an implementation bug, include a small example that helps to reproduce the error. The issue will be addressed as quickly as possible

1.3.3 Seeking Support

If you have questions or need additional support, please open a Github Issue or send a direct email to elizabeth.newman@emory.edu.

1.4 Examples

1.4.1 Hermite Interpolation

Traditoinal polynomial interpolation seeks to find a polynomial to approximate an underlying function at given points and correspoonding function values. Hermite interpolation seeks a polynomial that additionally fits derivative values at the given points. Each given point requires more information, but fewer points are required to form a quality polynomial approximation.

`hessQuik` makes it easy to obtain first- and second-order derivative information for the inputs of a network, and hence is well-suited for fitting values and derivatives.

Check out this Google Colab notebook for Hermite interpolation to see `hessQuik` fit the *`hessQuik.utils.data.peaks()`* function using derivative information!

1.4.2 Testing New Layers

`hessQuik` provides tools to develop and test new layers. The package provides testing tools to ensure the derivatives are implemented correctly. Choosing the best implementation of a given layer requires taking timing and storage costs into account.

Check out this Google Colab notebook on testing layers to see various implementations of the *`hessQuik.layers.single_layer.singleLayer`* and testing methods!

1.5 hessQuik Functionality

1.5.1 hessQuik.activations

hessQuik Activation Function

class hessQuikActivationFunction(**args: Any*, ***kwargs: Any*)

Bases: `Module`

Base class for all hessQuik activation functions.

forward(*x: torch.Tensor*, *do_gradient: bool = False*, *do_Hessian: bool = False*, *forward_mode: bool = True*) → Tuple[torch.Tensor, Optional[torch.Tensor], Optional[torch.Tensor]]

Applies a pointwise activation function to the incoming data.

Parameters

- **x** (*torch.Tensor*) – input into the activation function. $(*)$ where $*$ means any shape.
- **do_gradient** (*bool, optional*) – If set to `True`, the gradient will be computed during the forward call. Default: `False`
- **do_Hessian** (*bool, optional*) – If set to `True`, the Hessian will be computed during the forward call. Default: `False`
- **forward_mode** (*bool, optional*) – If set to `False`, the derivatives will be computed in backward mode. Default: `True`

Returns

- **sigma** (*torch.Tensor*) - value of activation function at input x, same size as x

- **dsigma** (*torch.Tensor* or None) - first derivative of activation function at input x, same size as x
- **d2sigma** (*torch.Tensor* or None) - second derivative of activation function at input x, same size as x

backward(*do_Hessian: bool = False*) → Tuple[torch.Tensor, Optional[torch.Tensor]]

Computes derivatives of activation function evaluated at x in backward mode.

Calls self.compute_derivatives without inputs, stores necessary variables in self.ctx.

Inherited by all subclasses.

compute_derivatives(**args*, *do_Hessian: bool = False*) → Tuple[torch.Tensor, Optional[torch.Tensor]]

Parameters

- **args** (*torch.Tensor*) – variables needed to compute derivatives
- **do_Hessian** (*bool, optional*) – If set to True, the Hessian will be computed during the forward call. Default: False

Returns

- **dsigma** (*torch.Tensor* or None) - first derivative of activation function at input x, same size as x
- **d2sigma** (*torch.Tensor* or None) - second derivative of activation function at input x, same size as x

AntiTanh

class antiTanhActivation(**args: Any*, ***kwargs: Any*)

Bases: *hessQuikActivationFunction*

Applies the antiderivative of the hyperbolic tangent activation function to each entry of the incoming data.

Examples:

```
>>> import hessQuik.activations as act
>>> act_func = act.antiTanhActivation()
>>> x = torch.randn(10, 4)
>>> sigma, dsigma, d2sigma = act_func(x, do_gradient=True, do_Hessian=True)
```

forward(*x*, *do_gradient=False*, *do_Hessian=False*, *forward_mode=True*)

Activates each entry of incoming data via

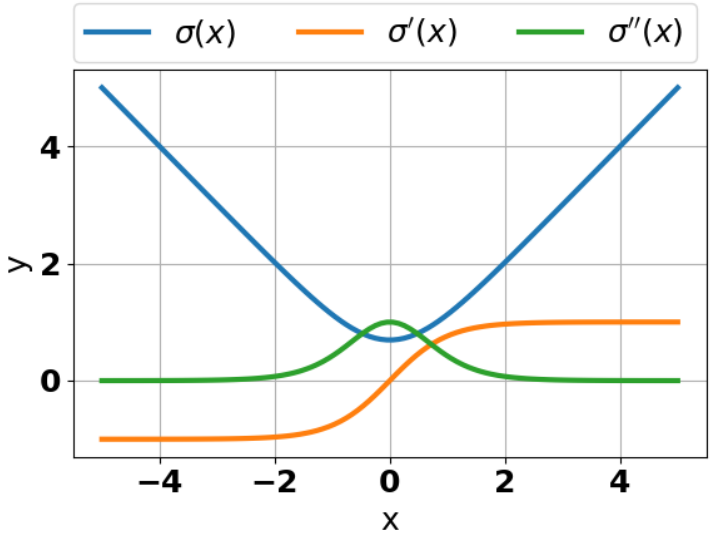
$$\sigma(x) = \ln(\cosh(x))$$

compute_derivatives(**args*, *do_Hessian=False*)

Computes the first and second derivatives of each entry of the incoming data via

$$\sigma'(x) = \tanh(x) \tag{1.1}$$

$$\sigma''(x) = 1 - \tanh^2(x) \tag{1.2}$$



Identity

class identityActivation(**args: Any*, ***kwargs: Any*)

Bases: *hessQuikActivationFunction*

Applies the identity activation function to each entry of the incoming data.

Examples:

```
>>> import hessQuik.activations as act
>>> act_func = act.identityActivation()
>>> x = torch.randn(10, 4)
>>> sigma, dsigma, d2sigma = act_func(x, do_gradient=True, do_Hessian=True)
```

forward(*x*, *do_gradient=False*, *do_Hessian=False*, *forward_mode=True*)

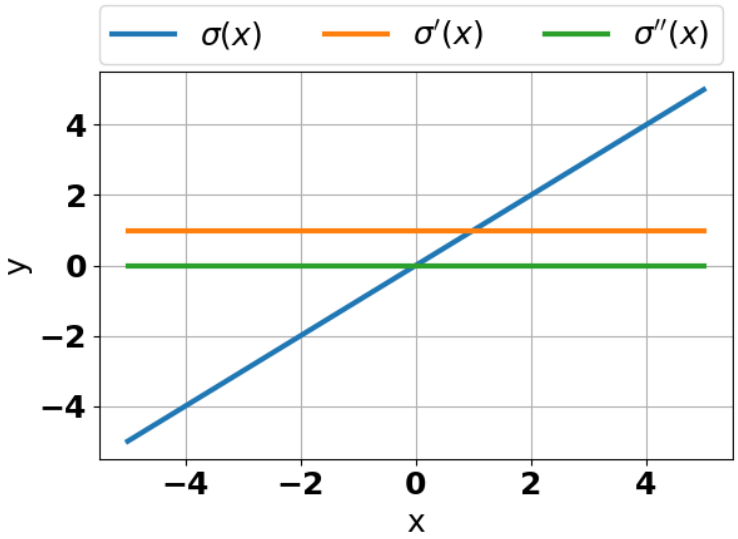
Activates each entry of incoming data via

$$\sigma(x) = x$$

compute_derivatives(**args*, *do_Hessian=False*)

Computes the first and second derivatives of each entry of the incoming data via

$$\sigma'(x) = 1 \tag{1.3}$$
$$\sigma''(x) = 0$$



Quadratic

class quadraticActivation(**args: Any*, ***kwargs: Any*)

Bases: *hessQuikActivationFunction*

Applies the quadratic activation function to each entry of the incoming data.

Examples:

```
>>> import hessQuik.activations as act
>>> act_func = act.quadraticActivation()
>>> x = torch.randn(10, 4)
>>> sigma, dsigma, d2sigma = act_func(x, do_gradient=True, do_Hessian=True)
```

forward(*x*, *do_gradient=False*, *do_Hessian=False*, *forward_mode=True*)

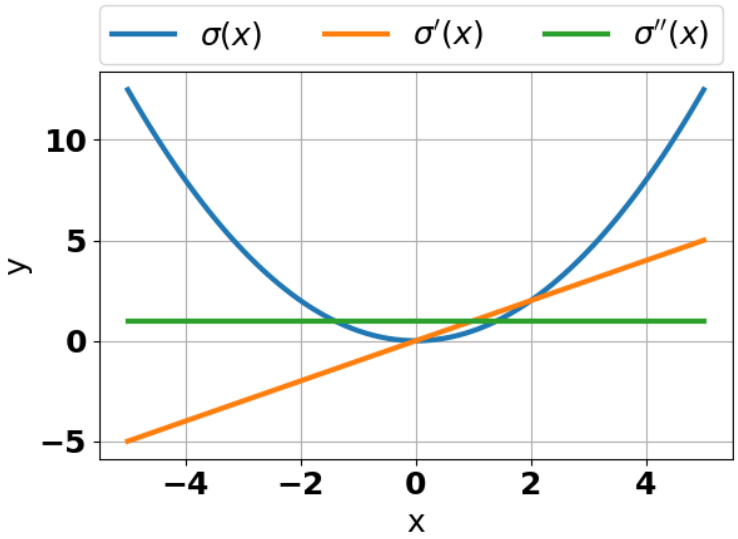
Activates each entry of incoming data via

$$\sigma(x) = \frac{1}{2}x^2$$

compute_derivatives(**args*, *do_Hessian=False*)

Computes the first and second derivatives of each entry of the incoming data via

$$\sigma'(x) = x \tag{1.5}$$
$$\sigma''(x) = 1 \tag{1.6}$$



Sigmoid

class sigmoidActivation(**args: Any*, ***kwargs: Any*)

Bases: *hessQuikActivationFunction*

Applies the sigmoid activation function to each entry of the incoming data.

Examples:

```
>>> import hessQuik.activations as act
>>> act_func = act.sigmoidActivation()
>>> x = torch.randn(10, 4)
>>> sigma, dsigma, d2sigma = act_func(x, do_gradient=True, do_Hessian=True)
```

forward(*x*, *do_gradient=False*, *do_Hessian=False*, *forward_mode=True*)

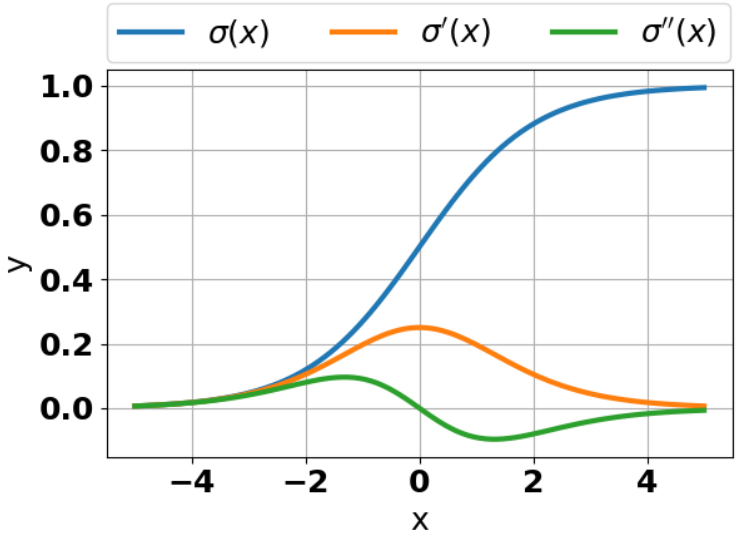
Activates each entry of incoming data via

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

compute_derivatives(**args*, *do_Hessian=False*)

Computes the first and second derivatives of each entry of the incoming data via

$$\sigma'(x) = \sigma(x)(1-\sigma(x)) \tag{1.7}$$
$$\sigma''(x) = \sigma'(x)(1-2*\sigma(x)) \tag{1.8}$$



Softplus

class softplusActivation(**args: Any*, ***kwargs: Any*)

Bases: *hessQuikActivationFunction*

Applies the softplus activation function to each entry of the incoming data.

Examples:

```
>>> import hessQuik.activations as act
>>> act_func = act.softplusActivation()
>>> x = torch.randn(10, 4)
>>> sigma, dsigma, d2sigma = act_func(x, do_gradient=True, do_Hessian=True)
```

__init__(*beta: float = 1.0*, *threshold: float = 20.0*) → None

Parameters

- **beta** (*float*) – parameter affecting steepness of the softplus function. Default: 1.0
- **threshold** (*float*) – parameter for numerical stability. Uses identity function when $\beta x > threshold$

forward(*x*, *do_gradient=False*, *do_Hessian=False*, *forward_mode=True*)

Activates each entry of incoming data via

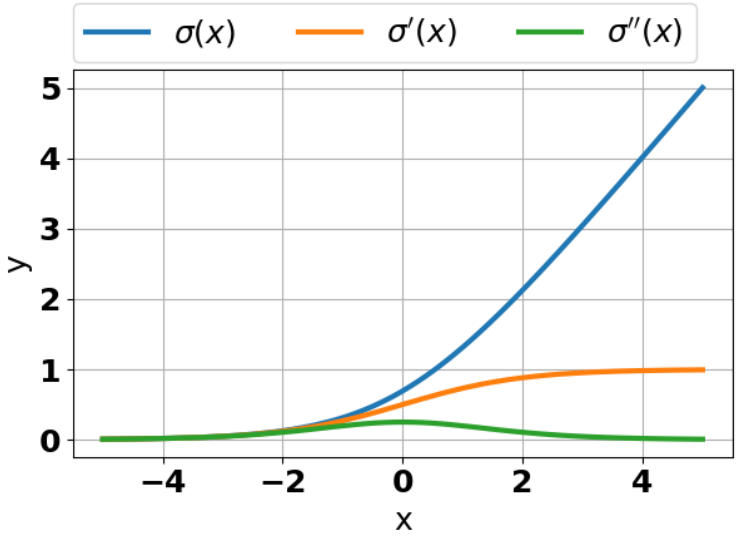
$$\sigma(x) = \frac{1}{\beta}\ln(1 + e^{\beta x})$$

compute_derivatives(**args*, *do_Hessian=False*)

Computes the first and second derivatives of each entry of the incoming data via

$$\sigma'(x) = \frac{1}{1 + e^{-\beta x}} \tag{1.9}$$

$$\sigma''(x) = \frac{\beta}{2\cosh(\beta x) + 2} \tag{1.10}$$



Tanh

class tanhActivation(**args: Any*, ***kwargs: Any*)

Bases: *hessQuikActivationFunction*

Applies the hyperbolic tangent activation function to each entry of the incoming data.

Examples:

```
>>> import hessQuik.activations as act
>>> act_func = act.tanhActivation()
>>> x = torch.randn(10, 4)
>>> sigma, dsigma, d2sigma = act_func(x, do_gradient=True, do_Hessian=True)
```

forward(*x*, *do_gradient=False*, *do_Hessian=False*, *forward_mode=True*)

Activates each entry of incoming data via

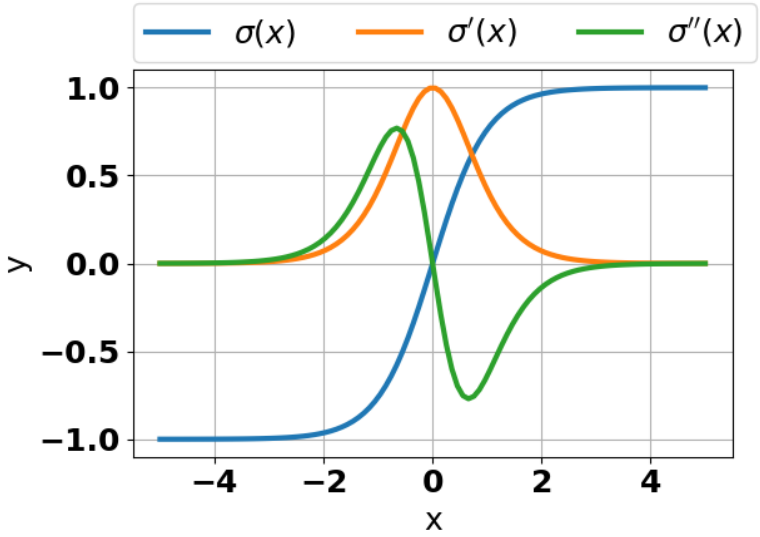
$$\sigma(x) = \tanh(x)$$

compute_derivatives(**args*, *do_Hessian=False*)

Computes the first and second derivatives of each entry of the incoming data via

$$\sigma'(x) = 1 - \tanh^2(x) \tag{1.11}$$

$$\sigma''(x) = -2\tanh(x)(1 - \tanh^2(x)) \tag{1.12}$$



1.5.2 hessQuik.layers

hessQuik Layer

class hessQuikLayer(**args: Any*, ***kwargs: Any*)

Bases: Module

Base class for all hessQuik layers.

dim_input() → int

Returns
dimension of input features

Return type
int

dim_output() → int

Returns
dimension of output features

Return type
int

forward(*u: torch.Tensor*, *do_gradient: bool = False*, *do_Hessian: bool = False*, *do_Laplacian: bool = False*, *forward_mode: bool = True*, *dudx: Optional[torch.Tensor] = None*, *d2ud2x: Optional[torch.Tensor] = None*, *v: Optional[torch.Tensor] = None*) → Tuple[torch.Tensor, Optional[torch.Tensor], Optional[torch.Tensor]]

Forward pass through the layer that maps input features u of size (n_s, n_{in}) to output features f of size (n_s, n_{out}) where n_s is the number of samples, n_{in} is the number of input features, and n_{out} is the number of output features.

The input features $u(x)$ is a function of the network input x of size (n_s, d) where d is the dimension of the network input.

Parameters

- **u** (*torch.Tensor*) – features from previous layer with shape (n_s, n_{in})
- **do_gradient** (*bool, optional*) – If set to `True`, the gradient will be computed during the forward call. Default: `False`
- **do_Hessian** (*bool, optional*) – If set to `True`, the Hessian will be computed during the forward call. Default: `False`
- **do_Laplacian** (*bool, optional*) – If set to `True`, the Laplacian will be computed during the forward call. Default: `False`
- **forward_mode** (*bool, optional*) – If set to `False`, the derivatives will be computed in backward mode. Default: `True`
- **dudx** (torch.Tensor or `None`) – if `forward_mode = True`, gradient of features from previous layer with respect to network input x with shape (n_s, d, n_{in})
- **d2ud2x** (torch.Tensor or `None`) – if `forward_mode = True`, Hessian of features from previous layer with respect to network input x with shape (n_s, d, d, n_{in})
- **v** (torch.Tensor or `None`) – if `forward_mode = True`, direction(s) to apply Jacobian and Hessian x with shape (d, k)

Returns

- **f** (*torch.Tensor*) - output features of layer with shape (n_s, n_{out})
- **dfdx** (*torch.Tensor* or `None`) - if `forward_mode = True`, gradient of output features with respect to network input x with shape (n_s, d, n_{out})
- **d2fd2x** (*torch.Tensor* or `None`) - if `forward_mode = True`, Hessian of output features with respect to network input x with shape (n_s, d, d, n_{out})

backward(*do_Hessian: bool = False*, *dgdf: Optional[torch.Tensor] = None*, *d2gd2f: Optional[torch.Tensor] = None*, *v: Optional[torch.Tensor] = None*) → Tuple[torch.Tensor, Optional[torch.Tensor]]

Backward pass through the layer that maps the gradient of the network g with respect to the output features f of size (n_s, n_{out}, m) to the gradient of the network g with respect to the input features u of size (n_s, n_{in}, m) where n_s is the number of samples, n_{in} is the number of input features,, n_{out} is the number of output features, and m is the number of network output features.

Parameters

- **do_Hessian** (*bool, optional*) – If set to `True`, the Hessian will be computed during the forward call. Default: `False`
- **dgdf** (*torch.Tensor*) – gradient of the subsequent layer features, $g(f)$, with respect to the layer outputs, f with shape (n_s, n_{out}, m).
- **d2gd2f** (torch.Tensor or `None`) – gradient of the subsequent layer features, $g(f)$, with respect to the layer outputs, f with shape $(n_s, n_{out}, n_{out}, m)$.
- **v** (torch.Tensor or `None`) – direction(s) to apply Jacobian transpose and Hessian x with shape (d, k)

Returns

- **dgdu** (*torch.Tensor* or `None`) - gradient of the network with respect to input features u with shape (n_s, n_{in}, m)
- **d2gd2u** (*torch.Tensor* or `None`) - Hessian of the network with respect to input features u with shape (n_s, n_{in}, n_{in}, m)

ICNN Layer

class ICNNLayer(**args: Any*, ***kwargs: Any*)

Bases: *hessQuikLayer*

Evaluate and compute derivatives of a single layer.

Examples:

```
>>> import torch, hessQuik.layers as lay
>>> f = lay.ICNNLayer(4, None, 7)
>>> x = torch.randn(10, 4)
>>> fx, dfdx, d2fd2x = f(x, do_gradient=True, do_Hessian=True)
>>> print(fx.shape, dfdx.shape, d2fd2x.shape)
torch.Size([10, 11]) torch.Size([10, 4, 11]) torch.Size([10, 4, 4, 11])
```

__init__(*input_dim: int*, *in_features: Optional[int]*, *out_features: int*, *act:* hessQuikActivationFunction = *torch.nn.Module*, *bias: bool = True*, *device=None*, *dtype=None*) → None

Parameters

- **input_dim** (*int*) – dimension of network inputs
- **in_features** (*int or``None``*) – number of input features. For first ICNN layer, set `in_features = None`
- **out_features** (*int*) – number of output features
- **act** (hessQuikActivationFunction) – activation function

Variables

- **K** – weight matrix for the network inputs of size (d, n_{out})
- **b** – bias vector of size $(n_{out},)$
- **L** – weight matrix for the input features of size (n_{in}, n_{out})
- **nonneg** – pointwise function to force l to have nonnegative weights. Default: `torch.nn.functional.softplus`

dim_input() → int

number of input features + dimension of network inputs

dim_output() → int

number of output features + dimension of network inputs

forward(*ux*, *do_gradient=False*, *do_Hessian=False*, *do_Laplacian=False*, *forward_mode=True*, *dudx=None*, *d2ud2x=None*, *v=None*)

Forward propagation through ICNN layer of the form

$$f(x) = \begin{bmatrix} \sigma\left(\begin{bmatrix} u(x) \\ x \end{bmatrix}\begin{bmatrix} L^+ \\ K \end{bmatrix} + b\right) \\ x \end{bmatrix}$$

Here, $u(x)$ is the input into the layer of size (n_s, n_{in}) which is a function of the input of the network, x of size (n_s, d). The output features, $f(x)$, are of size $(n_s, n_{out} + d)$. The notation $(\cdot)^+$ is a function that makes the weights of a matrix nonnegative.

As an example, for one sample, $n_s = 1$, the gradient with respect to $\begin{bmatrix} u & x \end{bmatrix}$ is of the form

$$\nabla_x f = \text{diag}\left(\sigma'\left(\begin{bmatrix} u(x) \\ x \end{bmatrix}\begin{bmatrix} L^+ \\ K \end{bmatrix} + b\right)\right)\begin{bmatrix} (L^+)^\top \\ K^\top \end{bmatrix}\begin{bmatrix} \nabla_x u \\ I \end{bmatrix}$$

where diag transforms a vector into the entries of a diagonal matrix and I is the $d \times d$ identity matrix.

backward(*do_Hessian=False*, *dgdf=None*, *d2gd2f=None*, *v=None*)

Backward propagation through ICNN layer of the form

$$f(u) = \begin{bmatrix} \sigma\left(\begin{bmatrix} u \\ x \end{bmatrix}\begin{bmatrix} L^+ \\ K \end{bmatrix} + b\right) \\ x \end{bmatrix}$$

Here, the network is g is a function of $f(u)$.

As an example, for one sample, $n_s = 1$, the gradient of the network with respect to u is of the form

$$\nabla_{[u,x]} g = \left(\sigma'\left(\begin{bmatrix} u \\ x \end{bmatrix}\begin{bmatrix} L^+ \\ K \end{bmatrix} + b\right) \odot \nabla_{[f,x]} g\right)\begin{bmatrix} (L^+)^\top \\ K^\top \end{bmatrix}$$

where $\odot$ denotes the pointwise product.

Quadratic ICNN Layer

class quadraticICNNLayer(**args: Any*, ***kwargs: Any*)

Bases: *hessQuikLayer*

Evaluate and compute derivatives of a ICNN quadratic layer.

Examples:

```
>>> import hessQuik.layers as lay
>>> f = lay.quadraticICNNLayer(4, None, 2)
>>> x = torch.randn(10, 4)
>>> fx, dfdx, d2fd2x = f(x, do_gradient=True, do_Hessian=True)
>>> print(fx.shape, dfdx.shape, d2fd2x.shape)
torch.Size([10, 1]) torch.Size([10, 4, 1]) torch.Size([10, 4, 4, 1])
```

__init__(*input_dim: int*, *in_features: Optional[int]*, *rank: int*, *device=None*, *dtype=None*) → None

Parameters

- **input_dim** (*int*) – dimension of network inputs
- **in_features** (int or None) – number of input features, n_{in}. For only ICNN quadratic layer, set `in_features = None`
- **rank** (*int*) – number of columns of quadratic matrix, r. In practice, $r < n_{in}$

Variables

- **v** – weight vector for network inputs of size $(d,)$
- **w** – weight vector for input features of size $(n_{in},)$
- **A** – weight matrix for quadratic term of size (d, r)
- **mu** – additive scalar bias
- **nonneg** – pointwise function to force l to have nonnegative weights. Default `torch.nn.functional.softplus`

dim_input() → int

number of input features + dimension of network inputs

dim_output() → int

scalar

forward(*ux*, *do_gradient=False*, *do_Hessian=False*, *do_Laplacian=False*, *forward_mode=True*, *dudx=None*, *d2ud2x=None*, *v=None*)

Forward propagation through ICNN layer of the form, for one sample $n_s = 1$,

$$f(x) = \begin{bmatrix} u(x) \\ x \end{bmatrix} \begin{bmatrix} w^+ \\ v \end{bmatrix} + \frac{1}{2} x A A^\top x^\top + \mu$$

Here, $u(x)$ is the input into the layer of size (n_s, n_{in}) which is a function of the input of the network, x of size (n_s, d). The output features, $f(x)$, are of size $(n_s, 1)$. The notation $(\cdot)^+$ is a function that makes the weights of a matrix nonnegative.

As an example, for one sample, $n_s = 1$, the gradient with respect to x is of the form

$$\nabla_x f = \begin{bmatrix} (w^+)^\top \\ v^\top \end{bmatrix} \begin{bmatrix} \nabla_x u \\ I \end{bmatrix} + x A A^\top$$

where I is the $d \times d$ identity matrix.

backward(*do_Hessian=False*, *dgdf=None*, *d2gd2f=None*, *v=None*)

Backward propagation through quadratic ICNN layer of the form, for one sample $n_s = 1$,

$$f\left(\begin{bmatrix} u & x \end{bmatrix}\right) = \begin{bmatrix} u \\ x \end{bmatrix} \begin{bmatrix} w^+ \\ v \end{bmatrix} + \frac{1}{2} x A A^\top x^\top + \mu$$

Here, the network is g is a function of $f(u)$.

The gradient of the layer with respect to $\begin{bmatrix} u & x \end{bmatrix}$ is of the form

$$\nabla_{[u,x]} f = \begin{bmatrix} (w^+)^\top & v^\top + x A A^\top \end{bmatrix}.$$

Quadratic Layer

class quadraticLayer(**args: Any*, ***kwargs: Any*)

Bases: *hessQuikLayer*

Evaluate and compute derivatives of a ICNN quadratic layer.

Examples:

```
>>> import hessQuik.layers as lay
>>> f = lay.quadraticLayer(4, 2)
>>> x = torch.randn(10, 4)
>>> fx, dfdx, d2fd2x = f(x, do_gradient=True, do_Hessian=True)
>>> print(fx.shape, dfdx.shape, d2fd2x.shape)
torch.Size([10, 1]) torch.Size([10, 4, 1]) torch.Size([10, 4, 4, 1])
```

__init__(*in_features: int*, *rank: int*, *device=None*, *dtype=None*) → None

Parameters

- **in_features** (*int*) – number of input features, n_{in}
- **rank** (*int*) – number of columns of quadratic matrix, r. In practice, $r < n_{in}$

Variables

- **v** – weight vector for network inputs of size $(d,)$
- **A** – weight matrix for quadratic term of size (d, r)
- **mu** – additive scalar bias

dim_input() → int

number of input features

dim_output() → int

scalar

forward(*u*, *do_gradient=False*, *do_Hessian=False*, *do_Laplacian=False*, *forward_mode=True*, *dudx=None*, *d2ud2x=None*, *v=None*)

Forward propagation through quadratic layer of the form, for one sample $n_s = 1$,

$$f(x) = u(x)v + \frac{1}{2}u(x)AA^\top u(x)^\top + \mu$$

Here, $u(x)$ is the input into the layer of size (n_s, n_{in}) which is a function of the input of the network, x. The output features, $f(x)$, are of size $(n_s, 1)$.

The gradient with respect to x is of the form

$$\nabla_x f = (v^\top + uAA^\top)\nabla_x u$$

backward(*do_Hessian=False*, *dgdf=None*, *d2gd2f=None*, *v=None*)

Backward propagation through quadratic ICNN layer of the form, for one sample $n_s = 1$,

$$f(u) = uv + \frac{1}{2}uAA^\top u^\top + \mu$$

Here, the network is g is a function of $f(u)$.

The gradient of the layer with respect to u is of the form

$$\nabla_u f = v^\top + uAA^\top.$$

Residual Layer

class resnetLayer(**args: Any*, ***kwargs: Any*)

Bases: *hessQuikLayer*

Evaluate and compute derivatives of a residual layer.

Examples:

```
>>> import hessQuik.layers as lay
>>> f = lay.resnetLayer(4, h=0.25)
>>> x = torch.randn(10, 4)
>>> fx, dfdx, d2fd2x = f(x, do_gradient=True, do_Hessian=True)
>>> print(fx.shape, dfdx.shape, d2fd2x.shape)
torch.Size([10, 4]) torch.Size([10, 4, 4]) torch.Size([10, 4, 4, 4])
```

__init__(*width: int*, *h: float = 1.0*, *act:* hessQuikActivationFunction *= torch.nn.Module*, *bias: bool = True*, *device=None*, *dtype=None*) → None

Parameters

- **width** (*int*) – number of input and output features, w
- **h** (*float*) – step size, $h > 0$
- **act** (hessQuikActivationFunction) – activation function
- **bias** (*bool*) – additive bias flag

Variables
layer – singleLayer with w input features and w output features

dim_input() → int
width

dim_output() → int
width

forward(*u*, *do_gradient=False*, *do_Hessian=False*, *do_Laplacian=False*, *forward_mode=True*, *dudx=None*, *d2ud2x=None*, *v=None*)
Forward propagation through resnet layer of the form

$$f(x) = u(x) + h \cdot singleLayer(u(x))$$

Here, $u(x)$ is the input into the layer of size (n_s, w) which is a function of the input of the network, x. The output features, $f(x)$, are of size (n_s, w).

As an example, for one sample, $n_s = 1$, the gradient with respect to x is of the form

$$\nabla_x f = I + h \nabla_x singleLayer(u(x))$$

where I denotes the $w \times w$ identity matrix.

backward(*do_Hessian=False*, *dgdf=None*, *d2gd2f=None*, *v=None*)
Backward propagation through single layer of the form

$$f(u) = u + h \cdot singleLayer(u)$$

Here, the network is g is a function of $f(u)$.

As an example, for one sample, $n_s = 1$, the gradient of the network with respect to u is of the form

$$\nabla_u g = \nabla_f g + h \cdot \nabla_u singleLayer(u)$$

where $\odot$ denotes the pointwise product.

Single Layer

class singleLayer(**args: Any*, ***kwargs: Any*)
Bases: *hessQuikLayer*

Evaluate and compute derivatives of a single layer.

Examples:

```
>>> import hessQuik.layers as lay
>>> f = lay.singleLayer(4, 7)
>>> x = torch.randn(10, 4)
>>> fx, dfdx, d2fd2x = f(x, do_gradient=True, do_Hessian=True)
>>> print(fx.shape, dfdx.shape, d2fd2x.shape)
torch.Size([10, 7]) torch.Size([10, 4, 7]) torch.Size([10, 4, 4, 7])
```

__init__(*in_features: int*, *out_features: int*, *act:* hessQuikActivationFunction = *torch.nn.Module*, *bias: bool* = *True*, *device=None*, *dtype=None*) → None

Parameters

- **in_features** (*int*) – number of input features, n_{in}
- **out_features** (*int*) – number of output features, n_{out}
- **act** (hessQuikActivationFunction) – activation function
- **bias** (*bool*) – additive bias flag

Variables

- **K** – weight matrix of size (n_{in}, n_{out})
- **b** – bias vector of size $(n_{out},)$

dim_input() → int

number of input features

dim_output()

number of output features

forward(*u*, *do_gradient=False*, *do_Hessian=False*, *do_Laplacian=False*, *forward_mode=True*, *dudx=None*, *d2ud2x=None*, *v=None*)

Forward propagation through single layer of the form

$$f(x) = \sigma(u(x)K + b)$$

Here, $u(x)$ is the input into the layer of size (n_s, n_{in}) which is a function of the input of the network, x. The output features, $f(x)$, are of size (n_s, n_{out}).

As an example, for one sample, $n_s = 1$, the gradient with respect to x is of the form

$$\nabla_x f = \text{diag}(\sigma'(u(x)K + b))K^\top \nabla_x u$$

where diag transforms a vector into the entries of a diagonal matrix.

backward(*do_Hessian=False*, *dgdf=None*, *d2gd2f=None*, *v=None*)

Backward propagation through single layer of the form

$$f(u) = \sigma(uK + b)$$

Here, the network is g is a function of $f(u)$.

As an example, for one sample, $n_s = 1$, the gradient of the network with respect to u is of the form

$$\nabla_u g = (\sigma'(uK + b) \odot \nabla_f g)K^\top$$

where $\odot$ denotes the pointwise product.

1.5.3 hessQuik.networks

hessQuik Network

class NN(**args: Any*, ***kwargs: Any*)

Bases: `Sequential`

Wrapper for hessQuik networks built upon torch.nn.Sequential.

__init__(**args*)

Parameters
args – sequence of hessQuik layers to be concatenated

dim_input() → int

Number of network input features

dim_output()

Number of network output features

setup_forward_mode(***kwargs*)

Setup forward or backward mode.

If `kwargs` does not include a `forward_mode` key, then the heuristic is to use `forward_mode = True` if $n_{in} < n_{out}$ where n_{in} is the number of input features and n_{out} is the number of output features.

There are three possible options once `forward_mode` is a key of `kwargs`:

- If `forward_mode = True`, then the network computes derivatives during forward propagation.
- If `forward_mode = False`, then the network calls the backward routine to compute derivatives after forward propagating.
- If `forward_mode = None`, then the network will compute derivatives in backward mode, but will not call the backward routine. This enables concatenation of networks, not just layers.

forward(*x: torch.Tensor*, *do_gradient: bool = False*, *do_Hessian: bool = False*, *do_Laplacian: bool = False*, *dudx: Optional[torch.Tensor] = None*, *d2ud2x: Optional[torch.Tensor] = None*, *v: Optional[torch.Tensor] = None*, ***kwargs*) → Tuple[torch.Tensor, Optional[torch.Tensor], Optional[torch.Tensor]]

Forward propagate through network and compute derivatives

Parameters

- **x** (*torch.Tensor*) – input into network of shape (n_s, d) where n_s is the number of samples and d is the number of input features
- **do_gradient** (*bool, optional*) – If set to `True`, the gradient will be computed during the forward call. Default: `False`
- **do_Hessian** (*bool, optional*) – If set to `True`, the Hessian will be computed during the forward call. Default: `False`
- **dudx** (torch.Tensor or `None`) – if `forward_mode = True`, gradient of features from previous layer with respect to network input x with shape (n_s, d, n_{in})
- **d2ud2x** (torch.Tensor or `None`) – if `forward_mode = True`, Hessian of features from previous layer with respect to network input x with shape (n_s, d, d, n_{in})
- **kwargs** – additional options, such as `forward_mode` as a user input

Returns

- **f** (*torch.Tensor*) - output features of network with shape (n_s, m) where m is the number of network output features
- **dfdx** (*torch.Tensor* or `None`) - if `forward_mode = True`, gradient of output features with respect to network input x with shape (n_s, d, m)
- **d2fd2x** (*torch.Tensor* or `None`) - if `forward_mode = True`, Hessian of output features with respect to network input x with shape (n_s, d, d, m)

backward(*do_Hessian: bool = False*, *dgdf: Optional[torch.Tensor] = None*, *d2gd2f: Optional[torch.Tensor] = None*, *v: Optional[torch.Tensor] = None*) → Tuple[torch.Tensor, Optional[torch.Tensor]]

Compute derivatives using backward propagation. This method is called during the forward pass if `forward_mode = False`.

Parameters

- **do_Hessian** (*bool, optional*) – If set to `True`, the Hessian will be computed during the forward call. Default: `False`
- **dgdf** (*torch.Tensor*) – gradient of the subsequent layer features, $g(f)$, with respect to the layer outputs, f with shape (n_s, n_{out}, m).
- **d2gd2f** (torch.Tensor or `None`) – gradient of the subsequent layer features, $g(f)$, with respect to the layer outputs, f with shape $(n_s, n_{out}, n_{out}, m)$.

Returns

- **dgdf** (*torch.Tensor* or `None`) - gradient of the network with respect to input features x with shape (n_s, d, m)
- **d2gd2f** (*torch.Tensor* or `None`) - Hessian of the network with respect to input features u with shape (n_s, d, d, m)

class NNPytorchAD(**args: Any*, ***kwargs: Any*)

Bases: `Module`

Compute the derivatives of a network using Pytorch's automatic differentiation.

The implementation follows that of CP Flow.

__init__(*net:* NN)

Create wrapper around hessQuik network.

Parameters
net (*hessQuik.networks.NN*) – hessQuik network

forward(*x: torch.Tensor*, *do_gradient: bool = False*, *do_Hessian: bool = False*, ***kwargs*) →
Tuple[torch.Tensor, Optional[torch.Tensor], Optional[torch.Tensor]]

Forward propagate through the hessQuik network without computing derivatives. Then, use automatic differentiation to compute derivatives using `torch.autograd.grad`.

class NNPytorchHessian(**args: Any*, ***kwargs: Any*)

Bases: `Module`

Compute the derivatives of a network using Pytorch's Hessian functional.

__init__(*net*)

Create wrapper around hessQuik network.

Parameters
net (*hessQuik.networks.NN*) – hessQuik network

forward(*x: torch.Tensor*, *do_gradient: bool = False*, *do_Hessian: bool = False*, ***kwargs*) → Tuple[torch.Tensor, Optional[torch.Tensor], Optional[torch.Tensor]]

Forward propagate through the hessQuik network without computing derivatives. Then, use automatic differentiation to compute derivatives using `torch.autograd.functional.hessian`.

Fully Connected Network

class fullyConnectedNN(**args: Any*, ***kwargs: Any*)

Bases: *NN*

Fully-connected network where every layer is a single layer. Let $u_0 = x$ be the input into the network. The construction is of the form

$$u_1 = \sigma(K_1 u_0 + b_1) \tag{1.13}$$
$$u_2 = \sigma(K_2 u_1 + b_2) \tag{1.14}$$
$$\vdots \tag{1.15}$$
$$u_\ell = \sigma(K_\ell u_{\ell-1} + b_\ell) \tag{1.16}$$

where ℓ is the number of layers. Each vector of features u_i is of size (n_s, n_i) where n_s is the number of samples and n_i is the dimension or width of the hidden features on layer i. Users choose the widths of the network and the activation function σ.

ICNN Network

class ICNN(**args: Any*, ***kwargs: Any*)

Bases: *NN*

Input Convex Neural Networks (ICNN) were proposed in the paper Input Convex Neural Networks by Amos, Xu, and Kolter. The network is constructed such that it is convex with respect to the network inputs x. It is constructed via

$$u_1 = \sigma(K_1 x + b_1) \tag{1.17}$$
$$u_2 = \sigma(L_2^+ u_1 + K_2 x + b_2) \tag{1.18}$$
$$\vdots \tag{1.19}$$
$$u_\ell = \sigma(L_\ell^+ u_{\ell-1} + K_\ell x + b_\ell) \tag{1.20}$$

where ℓ is the number of layers. Here, $(\cdot)^+$ is a function that forces the matrix to have only nonnegative entries. The activation function σ must be convex and non-decreasing.

Residual Neural Network

class resnetNN(**args: Any*, ***kwargs: Any*)

Bases: *NN*

Residual neural networks (ResNet) were popularized in the paper Deep Residual Learning for Image Recognition by He et al. Here, every layer is a single layer plus a skip connection. Let u_0 be the input into the ResNet. The construction is of the form

$$u_1 = u_0 + h\sigma(K_1 u_0 + b_1) \tag{1.21}$$
$$u_2 = u_1 + h\sigma(K_2 x + b_2) \tag{1.22}$$
$$\vdots \tag{1.23}$$
$$u_\ell = u_{\ell-1} + h\sigma(K_\ell u_{\ell-1} + b_\ell) \tag{1.24}$$

where ℓ is the number of layers, called the depth of the network. Each vector of features u_i is of size (n_s, w) where n_s is the number of samples and w is the width of the network. Users choose the width and depth of the network and the activation function σ.

1.5.4 hessQuik.utils

Peaks Data

peaks(*y: torch.Tensor*, *do_gradient: bool = False*, *do_Hessian: bool = False*) → Tuple[torch.Tensor, Optional[torch.Tensor], Optional[torch.Tensor]]

Generate data from the MATLAB 2D peaks function

Examples:

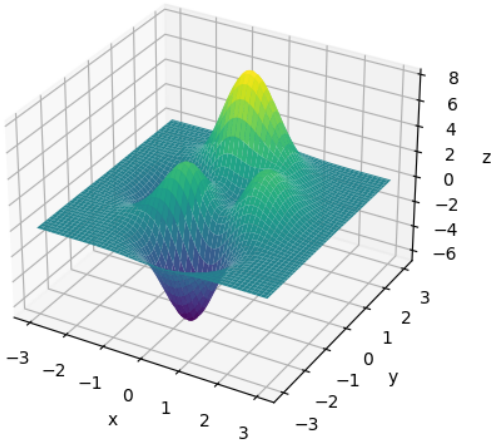
```
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits import mplot3d
x, y = torch.linspace(-3, 3, 100), torch.linspace(-3, 3, 100)
grid_x, grid_y = torch.meshgrid(x, y)
grid_xy = torch.concat((grid_x.reshape(-1, 1), grid_y.reshape(-1, 1)), dim=1)
grid_z, *_ = peaks(grid_xy)
fig = plt.figure()
ax = plt.axes(projection='3d')
surf = ax.plot_surface(grid_x, grid_y, grid_z.reshape(grid_x.shape), cmap=cm.
↪viridis)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
plt.show()
```

Parameters

- **y** (*torch.Tensor*) – (x, y) coordinates with shape $(n_s, 2)$ where n_s is the number of samples
- **do_gradient** (*bool, optional*) – If set to `True`, the gradient will be computed during the forward call. Default: `False`
- **do_Hessian** (*bool, optional*) – If set to `True`, the Hessian will be computed during the forward call. Default: `False`

Returns

- **f** (*torch.Tensor*) - value of peaks function at each coordinate with shape $(n_s, 1)$
- **dfdx** (*torch.Tensor* or `None`) - value of gradient at each coordinate with shape $(n_s, 2)$
- **d2fd2x** (*torch.Tensor* or `None`) - value of Hessian at each coordinate with shape $(n_s, 4)$



Input Derivative Checks

input_derivative_check(*f: Union[torch.nn.Module, Callable]*, *x: torch.Tensor*, *do_Hessian: bool = False*, *forward_mode: bool = True*, *num_test: int = 15*, *base: float = 2.0*, *tol: float = 0.1*, *verbose: float = False*) → Tuple[Optional[bool], Optional[bool]]

Taylor approximation test to verify derivatives. Form the approximation by perturbing the input x in the direction p with step size $h > 0$ via

$$f(x + hp) \approx f(x) + h\nabla f(x)^\top p + \frac{1}{2} p^\top \nabla^2 f(x) p$$

As $h \downarrow 0^+$, the error between the approximation and the true value will decrease. The rate of decrease indicates the accuracy of the derivative computation. For details, see Chapter 5 of Computational Methods for Electromagnetics by Eldad Haber.

Examples:

```
>>> from hessQuik.layers import singleLayer
>>> torch.set_default_dtype(torch.float64)  # use double precision to check␣
↪implementations
>>> x = torch.randn(10, 4)
>>> f = singleLayer(4, 7, act=act.softplusActivation())
>>> input_derivative_check(f, x, do_Hessian=True, verbose=True, forward_mode=True)
    h                           E0                          E1                          E2
    1.00 x 2^(00)               1.62 x 2^(-02)              1.70 x 2^(-07)              1.
↪02 x 2^(-12)
    1.00 x 2^(-01)              1.63 x 2^(-03)              1.70 x 2^(-09)              1.
↪06 x 2^(-15)
    1.00 x 2^(-02)              1.63 x 2^(-04)              1.69 x 2^(-11)              1.
↪08 x 2^(-18)
    1.00 x 2^(-03)              1.63 x 2^(-05)              1.69 x 2^(-13)              1.
↪09 x 2^(-21)
    1.00 x 2^(-04)              1.63 x 2^(-06)              1.69 x 2^(-15)              1.
↪09 x 2^(-24)
```

(continues on next page)

(continued from previous page)

```
    1.00 x 2^(-05)              1.63 x 2^(-07)          1.69 x 2^(-17)          1.
↪10 x 2^(-27)
    1.00 x 2^(-06)              1.63 x 2^(-08)          1.69 x 2^(-19)          1.
↪10 x 2^(-30)
    1.00 x 2^(-07)              1.63 x 2^(-09)          1.69 x 2^(-21)          1.
↪10 x 2^(-33)
    1.00 x 2^(-08)              1.63 x 2^(-10)          1.69 x 2^(-23)          1.
↪10 x 2^(-36)
    1.00 x 2^(-09)              1.63 x 2^(-11)          1.69 x 2^(-25)          1.
↪10 x 2^(-39)
    1.00 x 2^(-10)              1.63 x 2^(-12)          1.69 x 2^(-27)          1.
↪10 x 2^(-42)
    1.00 x 2^(-11)              1.63 x 2^(-13)          1.69 x 2^(-29)          1.
↪10 x 2^(-45)
    1.00 x 2^(-12)              1.63 x 2^(-14)          1.69 x 2^(-31)          1.
↪15 x 2^(-48)
    1.00 x 2^(-13)              1.63 x 2^(-15)          1.69 x 2^(-33)          1.
↪33 x 2^(-50)
    1.00 x 2^(-14)              1.63 x 2^(-16)          1.69 x 2^(-35)          1.
↪70 x 2^(-51)
   Gradient PASSED!
   Hessian PASSED!
```

Parameters

- **f** (*torch.nn.Module or Callable*) – callable function that returns value, gradient, and Hessian
- **x** (*torch.Tensor*) – input data
- **do_Hessian** (*bool, optional*) – If set to `True`, the Hessian will be computed during the forward call. Default: `False`
- **forward_mode** (*bool, optional*) – If set to `False`, the derivatives will be computed in backward mode. Default: `True`
- **num_test** (*int*) – number of perturbations
- **base** (*float*) – step size $h = base^k$
- **tol** (*float*) – small tolerance to account for numerical errors when computing the order of approximation
- **verbose** (*bool*) – printout flag

Returns

- **grad_check** (*bool*) - if `True`, gradient check passes
- **hess_check** (*bool, optional*) - if `True`, Hessian check passes

input_derivative_check_finite_difference(*f: Callable*, *x: torch.Tensor*, *do_Hessian: bool = False*, *forward_mode: bool = True*, *eps: float = 0.0001*, *atol: float = 1e-05*, *rtol: float = 0.001*, *verbose: bool = False*) → Tuple[Optional[bool], Optional[bool]]

Finite difference test to verify derivatives. Form the approximation by perturbing each entry in the input in the

unit direction with step size $\varepsilon > 0$:

$$\widetilde{\nabla f}_i = \frac{f(x_i + \varepsilon) - f(x_i - \varepsilon)}{2\varepsilon}$$

where $x_i \pm \varepsilon$ means add or subtract ε from the i-th entry of the input x, but leave the other entries unchanged. The notation $\widetilde{(\cdot)}$ indicates the finite difference approximation.

Examples:

```
>>> from hessQuik.layers import singleLayer
>>> torch.set_default_dtype(torch.float64)  # use double precision to check␣
↪implementations
>>> x = torch.randn(10, 4)
>>> f = singleLayer(4, 7, act=act.tanhActivation())
>>> input_derivative_check_finite_difference(f, x, do_Hessian=True, verbose=True,␣
↪forward_mode=True)
    Gradient Finite Difference: Error = 8.1720e-10, Relative Error = 2.5602e-10
    Gradient PASSED!
    Hessian Finite Difference: Error = 4.5324e-08, Relative Error = 4.4598e-08
    Hessian PASSED!
```

Parameters

- **f** (*Callable*) – callable function that returns value, gradient, and Hessian
- **x** (*torch.Tensor*) – input data
- **do_Hessian** (*bool, optional*) – If set to `True`, the Hessian will be computed during the forward call. Default: `False`
- **forward_mode** (*bool, optional*) – If set to `False`, the derivatives will be computed in backward mode. Default: `True`
- **eps** (*float*) – step size. Default: 1e-4
- **atol** (*float*) – absolute tolerance, e.g., $\|\nabla f - \widetilde{\nabla f}\| < atol$. Default: 1e-5
- **rtol** (*float*) – relative tolerance, e.g., $\|\nabla f - \widetilde{\nabla f}\| / \|\nabla f\| < rtol$. Default: 1e-3
- **verbose** (*bool*) – printout flag

Returns

- **grad_check** (*bool*) - if `True`, gradient check passes
- **hess_check** (*bool, optional*) - if `True`, Hessian check passes

Network Weights Derivative Check

network_derivative_check(*f: torch.nn.Module*, *x: torch.Tensor*, *do_Hessian: bool = False*, *forward_mode: bool = True*, *num_test: int = 15*, *base: float = 2.0*, *tol: float = 0.1*, *verbose: bool = False*) → Optional[bool]

Taylor approximation test to verify derivatives. Form the approximation by perturbing the network weights θ in the direction p with step size $h > 0$ via

$$\Phi(\theta + hp) \approx \Phi(\theta) + h\nabla_\theta \Phi(\theta)^\top p$$

where Φ is the objective function and θ are the network weights. This test uses the loss

$$\Phi(\theta) = \frac{1}{2}\|f_\theta(x)\|^2 + \frac{1}{2}\|\nabla f_\theta(x)\|^2 + \frac{1}{2}\|\nabla^2 f_\theta(x)\|^2$$

to validate network gradient computation after computing derivatives of the input features of the network f_θ.

As $h \downarrow 0^+$, the error between the approximation and the true value will decrease. The rate of decrease indicates the accuracy of the derivative computation. For details, see Chapter 5 of Computational Methods for Electromagnetics by Eldad Haber.

Examples:

```
>>> import hessQuik.activations as act, hessQuik.layers as lay, hessQuik.networks␣
↪as net
>>> torch.set_default_dtype(torch.float64)  # use double precision to check␣
↪implementations
>>> x = torch.randn(10, 4)
>>> width, depth = 8, 3
>>> f = net.NN(lay.singleLayer(4, width, act=act.tanhActivation()), net.
↪resnetNN(width, depth, h=1.0, act=act.tanhActivation()), lay.singleLayer(width, 1,
↪ act=act.identityActivation()))
>>> network_derivative_check(f, x, do_Hessian=True, verbose=True, forward_mode=True)
    h                           E0                          E1
    1.00 x 2^(00)               1.97 x 2^(02)               1.05 x 2^(01)
    1.00 x 2^(-01)              1.14 x 2^(02)               1.71 x 2^(-02)
    1.00 x 2^(-02)              1.20 x 2^(01)               1.50 x 2^(-04)
    1.00 x 2^(-03)              1.22 x 2^(00)               1.40 x 2^(-06)
    1.00 x 2^(-04)              1.24 x 2^(-01)              1.35 x 2^(-08)
    1.00 x 2^(-05)              1.24 x 2^(-02)              1.32 x 2^(-10)
    1.00 x 2^(-06)              1.24 x 2^(-03)              1.31 x 2^(-12)
    1.00 x 2^(-07)              1.24 x 2^(-04)              1.30 x 2^(-14)
    1.00 x 2^(-08)              1.25 x 2^(-05)              1.30 x 2^(-16)
    1.00 x 2^(-09)              1.25 x 2^(-06)              1.30 x 2^(-18)
    1.00 x 2^(-10)              1.25 x 2^(-07)              1.30 x 2^(-20)
    1.00 x 2^(-11)              1.25 x 2^(-08)              1.30 x 2^(-22)
    1.00 x 2^(-12)              1.25 x 2^(-09)              1.30 x 2^(-24)
    1.00 x 2^(-13)              1.25 x 2^(-10)              1.30 x 2^(-26)
    1.00 x 2^(-14)              1.25 x 2^(-11)              1.30 x 2^(-28)
    Gradient PASSED!
```

Parameters

- **f** (*torch.nn.Module*) – callable function that returns value, gradient, and Hessian
- **x** (*torch.Tensor*) – input data
- **do_Hessian** (*bool, optional*) – If set to `True`, the Hessian will be computed during the forward call. Default: `False`
- **forward_mode** (*bool, optional*) – If set to `False`, the derivatives will be computed in backward mode. Default: `True`
- **num_test** (*int*) – number of perturbations
- **base** (*float*) – step size $h = base^k$
- **tol** (*float*) – small tolerance to account for numerical errors when computing the order of approximation

- **verbose** (*bool*) – printout flag

Returns

- **grad_check** (*bool*) - if `True`, gradient check passes

Timing Functions

setup_device_and_gradient(*f: NN*, *network_wrapper: str = 'hessQuik'*, *device: str = 'cpu'*) → torch.nn.Module

Setup network with correct wrapper and device

setup_resnet(*in_features: int*, *out_features: int*, *width: int = 16*, *depth: int = 4*) → *NN*

Setup resnet architecture for timing tests

setup_fully_connected(*in_features: int*, *out_features: int*, *width: int = 16*, *depth: int = 4*) → *NN*

Setup fully-connected architecture for timing tests

setup_icnn(*in_features: int*, *out_features: int*, *width: int = 16*, *depth: int = 4*) → *NN*

Setup ICNN architecture for timing tests.

Requires scalar output.

setup_network(*in_features: int*, *out_features: int*, *width: int*, *depth: int*, *network_type: str = 'resnet'*, *network_wrapper: str = 'hessQuik'*, *device: str = 'cpu'*)

Wrapper to setup network.

timing_test_cpu(*f: Union[NN, torch.nn.Module]*, *x: torch.Tensor*, *num_trials: int = 10*, *clear_memory: bool = True*) → torch.Tensor

Timing test for one architecture on CPU.

Test is run `num_trials` times and the timing for each trial is returned.

The timing includes one dry run for the first iteration that is not returned.

Memory is cleared after the completion of all trials.

timing_test_gpu(*f: Union[NN, torch.nn.Module]*, *x: torch.Tensor*, *num_trials: int = 10*, *clear_memory: bool = True*)

Timing test for one architecture on CPU.

Test is run `num_trials` times and the timing for each trial is returned.

Each trial includes a `torch.cuda.synchonize` call.

The timing includes one dry run for the first iteration that is not returned.

Memory is cleared after the completion of all trials.

timing_test(*in_feature_range: torch.Tensor*, *out_feature_range: torch.Tensor*, *nex: int = 10*, *num_trials: int = 10*, *width: int = 16*, *depth: int = 4*, *network_wrapper: str = 'hessQuik'*, *network_type: str = 'resnet'*, *device: str = 'cpu'*, *clear_memory: bool = True*) → dict

Parameters

- **in_feature_range** (*torch.Tensor*) – available input feature dimensions
- **out_feature_range** (*torch.Tensor*) – available output feature dimensions
- **nex** (*int, optional*) – number of examples for network input. Default: 10
- **num_trials** (*int, optional*) – number of trials per input-output feature combination. Default: 10

- **width** (*int, optional*) – width of network. Default: 16
- **depth** (*int, optional*) – depth of network. Default: 4
- **network_wrapper** (*str, optional*) – type of network wrapper. Default: 'hessQuik'. Options: 'hessQuik', 'PytorchAD', 'PytorchHessian'
- **network_type** (*str, optional*) – network architecture. Default: 'resnet'. Options: 'resnet', 'fully_connected', 'icnn'
- **device** (*str, optional*) – device for testing. Default: 'cpu'
- **clear_memory** (*bool, optional*) – flag to clear memory after each set of trials

Returns

dictionary containing keys

- **'timing_trials'** (*torch.Tensor*) - time (in seconds) for each trial and each architecture
- **'timing_trials_mean'** (*torch.Tensor*) - average over trials for each architecture
- **'timing_trials_std'** (*torch.Tensor*) - standard deviation over trials for each architecture
- **'in_feature_range'** (*torch.Tensor*) - available input feature dimensions
- **'out_feature_range'** (*torch.Tensor*) - available output feature dimensions
- **'nex'** (*int*) - number of samples for the input data
- **'num_trials'** (*int*) - number of trials per architecture

Training Functions

train_one_epoch(*f: torch.nn.Module*, *x: torch.Tensor*, *y: torch.Tensor*, *optimizer: torch.optim.Optimizer*, *batch_size: int = 5*, *do_gradient: bool = False*, *do_Hessian: bool = False*, *loss_weights: Union[tuple, list] = (1.0, 1.0, 1.0)*)

Training mean-square loss for one epoch where the loss function is

$$L(\theta) = \frac{1}{2N}\left(w_0\|f_\theta(x)\|^2 + w_1\|\nabla f_\theta(x)\|^2 + w_2\|\nabla^2 f_\theta(x)\|^2\right)$$

where f_{theta} is the network θ are the network weights, and N is the number of training samples. The loss corresponding to the function value, gradient, and Hessian each can have different weights, w_0, w_1, and w_2, respectively.

Parameters

- **f** (*torch.nn.Module*) – hessQuik neural network to train
- **x** (*torch.Tensor*) – training data of shape $(N, *)$ where $*$ can be any shapw
- **y** (*torch.Tensor*) – target data of shape $(N, *)$
- **optimizer** (*torch.optim.Optimizer*) – method for updating the network weights
- **batch_size** (*int*) – size of mini-batches for stochatistic training. Default: 5
- **do_gradient** (*bool, optional*) – If set to `True`, the gradient will be computed during the forward call. Default: `False`
- **do_Hessian** (*bool, optional*) – If set to `True`, the Hessian will be computed during the forward call. Default: `False`
- **loss_weights** (*tuple or list*) – weight for each term in the loss function

Returns
tuple containing the overall running loss and the running loss for each term in the loss function

test(*f: torch.nn.Module*, *x: torch.Tensor*, *y: torch.Tensor*, *do_gradient: bool = False*, *do_Hessian: bool = False*, *loss_weights: Union[tuple, list] = (1.0, 1.0, 1.0)*)

Evaluate mean-squared loss function without training

See *hessQuik.utils.training.train_one_epoch()* for details.

print_headers(*do_gradient: bool = True*, *do_Hessian: bool = False*, *verbose: bool = False*, *loss_weights: Union[tuple, list] = (1.0, 1.0, 1.0)*)

Print headers for nice training

Additional Utilities

module_getattr(*obj: torch.nn.Module*, *names: Tuple*)

Get specific attribute of module at any level

module_setattr(*obj: torch.nn.Module*, *names: Tuple*, *val: torch.Tensor*)

Set specific attribute of module at any level

extract_data(*net: torch.nn.Module*, *attr: str = 'data'*)

Extract data stored in specific attribute and store as 1D array

insert_data(*net: torch.nn.Module*, *theta: torch.Tensor*) → None

Insert 1D array of data into specific attribute

convert_to_base(*a: tuple*, *b: float = 2.0*) → tuple

Convert tuple of floats to a base-exponent pair for nice printouts.

See use in, e.g., *hessQuik.utils.input_derivative_check.input_derivative_check()*.

1.6 hessQuik Derivations

Coming Soon!

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

h

INDEX

Symbols

A

B

C

D

E

F

H

I

M

N

P

Q

R

S

T