
AnalogVNN

Release 1.0.7

Vivswan Shah

Nov 22, 2023

CONTENTS

1	Table of contents	3
1.1	Install AnalogVNN	3
1.2	Cite AnalogVNN	4
1.3	Sample code	5
1.4	Tutorial	10
1.5	Inner Workings	13
1.6	Extra Analog Classes	14
1.7	API Reference	17
1.8	Changelog	98
2	Indices and tables	101
	Python Module Index	103
	Index	105

GitHub: <https://github.com/Vivswan/AnalogVNN>

AnalogVNN is a simulation framework built on PyTorch which can simulate the effects of analog components like optoelectronic noise, limited precision, and signal normalization present in photonics neural network accelerators. By following the same layer structure design present in PyTorch, the AnalogVNN framework allows users to convert most digital neural network models to their analog counterparts with just a few lines of code, taking full advantage of the open-source optimization, deep learning, and GPU acceleration libraries available through PyTorch.

TABLE OF CONTENTS

1.1 Install AnalogVNN

AnalogVNN is tested and supported on the following 64-bit systems:

- Python 3.7, 3.8, 3.9, 3.10, 3.11
- Windows 7 and later
- Ubuntu 16.04 and later, including WSL
- Red Hat Enterprise Linux 7 and later
- OpenSUSE 15.2 and later
- macOS 10.12 and later

1.1.1 Installation

Install `PyTorch` then:

- Pip:

```
# Current stable release for CPU and GPU
pip install analogvnn

# For additional optional features
pip install analogvnn[full]
```

OR

- AnalogVNN can be downloaded at ([GitHub](#)) or creating a fork of it.

1.1.2 Dependencies

Install the required dependencies:

- PyTorch
 - Manual installation required: <https://pytorch.org/>
- dataclasses
- scipy
- numpy

- networkx
- (optional) tensorboard
 - For using tensorboard to visualize the network, with class `analogvnn.utils.TensorboardModelLog.TensorboardModelLog`
- (optional) torchinfo
 - For adding summary to tensorboard by using `analogvnn.utils.TensorboardModelLog.TensorboardModelLog.add_summary()`
- (optional) graphviz
 - For saving and rendering forward and backward graphs using `analogvnn.graph.AcyclicDirectedGraph.AcyclicDirectedGraph.render()`
- (optional) python-graphviz
 - For saving and rendering forward and backward graphs using `analogvnn.graph.AcyclicDirectedGraph.AcyclicDirectedGraph.render()`

That's it, you are all set to simulate analog neural networks.

Head over to the [Tutorial](#) and look over the [Sample code](#).

1.2 Cite AnalogVNN

We would appreciate if you cite the following paper in your publications for which you used AnalogVNN:

DOI: [10.48550/arXiv.2210.10048](https://doi.org/10.48550/arXiv.2210.10048)

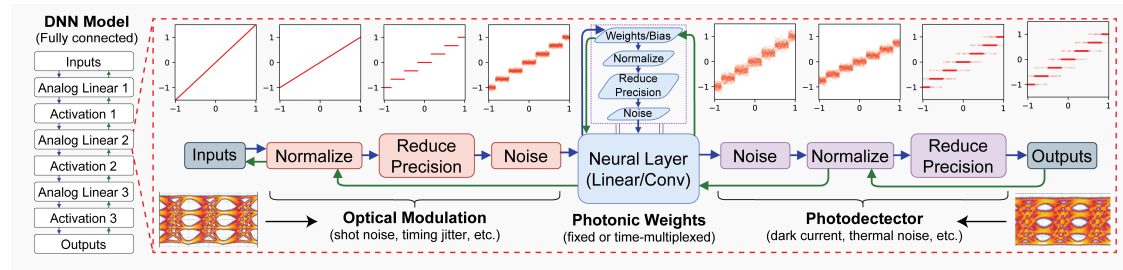
1.2.1 In BibTeX format

```
@article{shah2022analogvnn,  
  title={AnalogVNN: A fully modular framework for modeling and optimizing photonic_  
↪neural networks},  
  author={Shah, Vivswan and Youngblood, Nathan},  
  journal={arXiv preprint arXiv:2210.10048},  
  year={2022}  
}
```

1.2.2 In textual form

Vivswan Shah, and Nathan Youngblood. "AnalogVNN: A fully modular framework for modeling and optimizing photonic neural networks." arXiv preprint arXiv:2210.10048 (2022).

1.3 Sample code



Sample code and Sample code with logs for 3 layered linear photonic analog neural network with 4-bit precision, 0.5 Leakage and Clamp normalization:

```
import torch.backends.cudnn
import torchvision
from torch import optim, nn
from torch.utils.data import DataLoader
from torchvision.transforms import transforms

from analogvnn.nn.Linear import Linear
from analogvnn.nn.activation.Gaussian import GeLU
from analogvnn.nn.module.FullSequential import FullSequential
from analogvnn.nn.noise.GaussianNoise import GaussianNoise
from analogvnn.nn.normalize.Clamp import Clamp
from analogvnn.nn.precision.ReducePrecision import ReducePrecision
from analogvnn.parameter.PseudoParameter import PseudoParameter
from analogvnn.utils.is_cpu_cuda import is_cpu_cuda

def load_vision_dataset(dataset, path, batch_size, is_cuda=False, grayscale=True):
    """
    Loads a vision dataset with optional grayscale conversion and CUDA support.

    Args:
        dataset (Type[torchvision.datasets.VisionDataset]): the dataset class to use (e.
        ↳ g. torchvision.datasets.MNIST)
        path (str): the path to the dataset files
        batch_size (int): the batch size to use for the data loader
        is_cuda (bool): a flag indicating whether to use CUDA support (defaults to False)
        grayscale (bool): a flag indicating whether to convert the images to grayscale.
        ↳ (defaults to True)

    Returns:
        A tuple containing the train and test data loaders, the input shape, and a tuple
        ↳ of class labels.
    """
    dataset_kwargs = {
        'batch_size': batch_size,
        'shuffle': True
    }
```

(continues on next page)

(continued from previous page)

```

    }

    if is_cuda:
        cuda_kwargs = {
            'num_workers': 1,
            'pin_memory': True,
        }
        dataset_kwargs.update(cuda_kwargs)

    if grayscale:
        use_transform = transforms.Compose([
            transforms.Grayscale(),
            transforms.ToTensor(),
        ])
    else:
        use_transform = transforms.Compose([transforms.ToTensor()])

    train_set = dataset(root=path, train=True, download=True, transform=use_transform)
    test_set = dataset(root=path, train=False, download=True, transform=use_transform)
    train_loader = DataLoader(train_set, **dataset_kwargs)
    test_loader = DataLoader(test_set, **dataset_kwargs)

    zeroth_element = next(iter(test_loader))[0]
    input_shape = list(zeroth_element.shape)

    return train_loader, test_loader, input_shape, tuple(train_set.classes)

def cross_entropy_accuracy(output, target) -> float:
    """Cross Entropy accuracy function.

    Args:
        output (torch.Tensor): output of the model from passing inputs
        target (torch.Tensor): correct labels for the inputs

    Returns:
        float: accuracy from 0 to 1
    """

    _, preds = torch.max(output.data, 1)
    correct = (preds == target).sum().item()
    return correct / len(output)

class LinearModel(FullSequential):
    def __init__(self, activation_class, norm_class, precision_class, precision, noise_
    ↪class, leakage):
        """Initialise LinearModel with 3 Dense layers.

        Args:
            activation_class: Activation Class
            norm_class: Normalization Class

```

(continues on next page)

(continued from previous page)

```

        precision_class: Precision Class (ReducePrecision or
↪ StochasticReducePrecision)
        precision (int): precision of the weights and biases
        noise_class: Noise Class
        leakage (float): leakage is the probability that a reduced precision digital
↪ value (e.g., "1011") will
        acquire a different digital value (e.g., "1010" or "1100") after passing
↪ through the noise layer
        (i.e., the probability that the digital values transmitted and detected are
↪ different after passing through
        the analog channel).
        """

        super().__init__()

        self.activation_class = activation_class
        self.norm_class = norm_class
        self.precision_class = precision_class
        self.precision = precision
        self.noise_class = noise_class
        self.leakage = leakage

        self.all_layers = []
        self.all_layers.append(nn.Flatten(start_dim=1))
        self.add_layer(Linear(in_features=28 * 28, out_features=256))
        self.add_layer(Linear(in_features=256, out_features=128))
        self.add_layer(Linear(in_features=128, out_features=10))

        self.add_sequence(*self.all_layers)

    def add_layer(self, layer):
        """To add the analog layer.

        Args:
            layer (BaseLayer): digital layer module
        """

        self.all_layers.append(self.norm_class())
        self.all_layers.append(self.precision_class(precision=self.precision))
        self.all_layers.append(self.noise_class(leakage=self.leakage, precision=self.
↪ precision))
        self.all_layers.append(layer)
        self.all_layers.append(self.noise_class(leakage=self.leakage, precision=self.
↪ precision))
        self.all_layers.append(self.norm_class())
        self.all_layers.append(self.precision_class(precision=self.precision))
        self.all_layers.append(self.activation_class())
        self.activation_class.initialise_(layer.weight)

class WeightModel(FullSequential):
    def __init__(self, norm_class, precision_class, precision, noise_class, leakage):

```

(continues on next page)

(continued from previous page)

```

        """Initialize the WeightModel.

        Args:
            norm_class: Normalization Class
            precision_class: Precision Class (ReducePrecision or ↵
↵ StochasticReducePrecision)
            precision (int): precision of the weights and biases
            noise_class: Noise Class
            leakage (float): leakage is the probability that a reduced precision digital ↵
↵ value (e.g., "1011") will
            acquire a different digital value (e.g., "1010" or "1100") after passing ↵
↵ through the noise layer
            (i.e., the probability that the digital values transmitted and detected are ↵
↵ different after passing through
            the analog channel).
        """

        super().__init__()
        self.all_layers = []

        self.all_layers.append(norm_class())
        self.all_layers.append(precision_class(precision=precision))
        self.all_layers.append(noise_class(leakage=leakage, precision=precision))

        self.eval()
        self.add_sequence(*self.all_layers)

def run_linear3_model():
    """The main function to train photonics image classifier with 3 linear/dense nn for ↵
↵ MNIST dataset."""

    is_cpu_cuda.use_cuda_if_available()
    torch.backends.cudnn.benchmark = True
    torch.manual_seed(0)
    device, is_cuda = is_cpu_cuda.is_using_cuda
    print(f'Device: {device}')
    print()

    # Loading Data
    print('Loading Data...')
    train_loader, test_loader, input_shape, classes = load_vision_dataset(
        dataset=torchvision.datasets.MNIST,
        path='_data/',
        batch_size=128,
        is_cuda=is_cuda
    )

    # Creating Models
    print('Creating Models...')
    nn_model = LinearModel(
        activation_class=GeLU,

```

(continues on next page)

(continued from previous page)

```

        norm_class=Clamp,
        precision_class=ReducePrecision,
        precision=2 ** 4,
        noise_class=GaussianNoise,
        leakage=0.5
    )
    weight_model = WeightModel(
        norm_class=Clamp,
        precision_class=ReducePrecision,
        precision=2 ** 4,
        noise_class=GaussianNoise,
        leakage=0.5
    )

    # Parametrizing Parameters of the Models
    PseudoParameter.parametrize_module(nn_model, transformation=weight_model)

    # Setting Model Parameters
    nn_model.loss_function = nn.CrossEntropyLoss()
    nn_model.accuracy_function = cross_entropy_accuracy
    nn_model.optimizer = optim.Adam(params=nn_model.parameters())

    # Compile Model
    nn_model.compile(device=device)
    weight_model.compile(device=device)

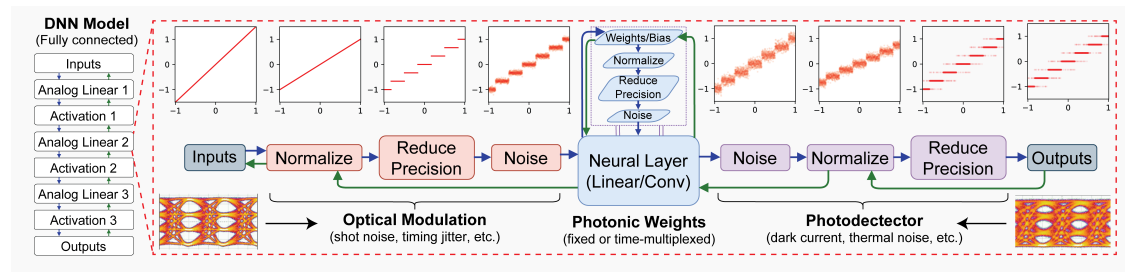
    # Training
    print('Starting Training...')
    for epoch in range(10):
        train_loss, train_accuracy = nn_model.train_on(train_loader, epoch=epoch)
        test_loss, test_accuracy = nn_model.test_on(test_loader, epoch=epoch)

        str_epoch = str(epoch + 1).zfill(1)
        print_str = f'({str_epoch})' \
                    f' Training Loss: {train_loss:.4f}, ' \
                    f' Training Accuracy: {100. * train_accuracy:.0f}%, ' \
                    f' Testing Loss: {test_loss:.4f}, ' \
                    f' Testing Accuracy: {100. * test_accuracy:.0f}%\n'
        print(print_str)
    print('Run Completed Successfully...')

if __name__ == '__main__':
    run_linear3_model()

```

1.4 Tutorial



To convert a digital model to its analog counterpart the following steps needs to be followed:

1. Adding the analog layers to the digital model. For example, to create the Photonic Linear Layer with *Reduce Precision*, *Normalization* and *Noise*:

1. Create the model similar to how you would create a digital model but using `analogvnn.nn.module.FullSequential.FullSequential` as superclass

```
class LinearModel(FullSequential):
    def __init__(self, activation_class, norm_class, precision_class, precision,
        ↪ noise_class, leakage):
        super().__init__()

        self.activation_class = activation_class
        self.norm_class = norm_class
        self.precision_class = precision_class
        self.precision = precision
        self.noise_class = noise_class
        self.leakage = leakage

        self.all_layers = []
        self.all_layers.append(nn.Flatten(start_dim=1))
        self.add_layer(Linear(in_features=28 * 28, out_features=256))
        self.add_layer(Linear(in_features=256, out_features=128))
        self.add_layer(Linear(in_features=128, out_features=10))

        self.add_sequence(*self.all_layers)
```

Note: `analogvnn.nn.module.Sequential.Sequential.add_sequence()` is used to create and set forward and backward graphs in AnalogVNN, more information in *Inner Workings*

2. To add the Reduce Precision, Normalization, and Noise before and after the main Linear layer, `add_layer` function is used.

```
def add_layer(self, layer):
    self.all_layers.append(self.norm_class())
    self.all_layers.append(self.precision_class(precision=self.precision))
    self.all_layers.append(self.noise_class(leakage=self.leakage,
        ↪ precision=self.precision))
    self.all_layers.append(layer)
    self.all_layers.append(self.noise_class(leakage=self.leakage,
        ↪ precision=self.precision))
    self.all_layers.append(self.norm_class())
```

(continues on next page)

(continued from previous page)

```

self.all_layers.append(self.precision_class(precision=self.precision))
self.all_layers.append(self.activation_class())
self.activation_class.initialise_(layer.weight)

```

2. Creating an Analog Parameters Model for analog parameters (analog weights and biases)

```

class WeightModel(FullSequential):
    def __init__(self, norm_class, precision_class, precision, noise_class,
        ↪leakage):
        super().__init__()
        self.all_layers = []

        self.all_layers.append(norm_class())
        self.all_layers.append(precision_class(precision=precision))
        self.all_layers.append(noise_class(leakage=leakage, precision=precision))

        self.eval()
        self.add_sequence(*self.all_layers)

```

Note: Since the WeightModel will only be used for converting the data to analog data to be used in the main LinearModel, we can use eval() to make sure the WeightModel is never been trained

3. Simply getting data and setting up the model as we will normally do in PyTorch with some minor changes for automatic evaluations

```

torch.backends.cudnn.benchmark = True
device, is_cuda = is_cpu_cuda.is_using_cuda
print(f"Device: {device}")
print()

# Loading Data
print(f"Loading Data...")
train_loader, test_loader, input_shape, classes = load_vision_dataset(
    dataset=torchvision.datasets.MNIST,
    path="_data/",
    batch_size=128,
    is_cuda=is_cuda
)

# Creating Models
print(f"Creating Models...")
nn_model = LinearModel(
    activation_class=GeLU,
    norm_class=Clamp,
    precision_class=ReducePrecision,
    precision=2 ** 4,
    noise_class=GaussianNoise,
    leakage=0.5
)
weight_model = WeightModel(
    norm_class=Clamp,
    precision_class=ReducePrecision,
    precision=2 ** 4,

```

(continues on next page)

(continued from previous page)

```
noise_class=GaussianNoise,
leakage=0.5
)

# Setting Model Parameters
nn_model.loss_function = nn.CrossEntropyLoss()
nn_model.accuracy_function = cross_entropy_accuracy
nn_model.compile(device=device)
weight_model.compile(device=device)
```

4. Using Analog Parameters Model to convert digital parameters to analog parameters using [*analogvnn.parameter.PseudoParameter.PseudoParameter.parametrize_module\(\)*](#)

```
PseudoParameter.parametrize_module(nn_model, transformation=weight_model)
```

5. Adding optimizer

```
nn_model.optimizer = optim.Adam(params=nn_model.parameters())
```

6. Then you are good to go to train and test the model

```
# Training
print(f"Starting Training...")
for epoch in range(10):
    train_loss, train_accuracy = nn_model.train_on(train_loader, epoch=epoch)
    test_loss, test_accuracy = nn_model.test_on(test_loader, epoch=epoch)

    str_epoch = str(epoch + 1).zfill(1)
    print_str = f'({str_epoch})' \
                f' Training Loss: {train_loss:.4f},' \
                f' Training Accuracy: {100. * train_accuracy:.0f}%, ' \
                f' Testing Loss: {test_loss:.4f},' \
                f' Testing Accuracy: {100. * test_accuracy:.0f}%\n'
    print(print_str)
print("Run Completed Successfully...")
```

Full Sample code for this process can be found at [*Sample code*](#)

1.5 Inner Workings

There are three major new classes in AnalogVNN, which are as follows

1.5.1 PseudoParameters

class: `analogvnn.parameter.PseudoParameter.PseudoParameter()`

`PseudoParameters` is a subclass of `Parameter` class of PyTorch.

`PseudoParameters` class lets you convert a digital parameter to analog parameter by converting the parameter of layer of `Parameter` class to `PseudoParameters`.

`PseudoParameters` requires a `ParameterizingModel` to parameterize the parameters (weights and biases) of the layer to get parameterized data

PyTorch's `ParameterizedParameters` vs AnalogVNN's `PseudoParameters`:

- Similarity (Forward or Parameterizing the data):

$$\text{Data} \rightarrow \text{ParameterizingModel} \rightarrow \text{Parameterized Data}$$
- Difference (Backward or Gradient Calculations):
 - `ParameterizedParameters`

$$\text{Parameterized Data} \rightarrow \text{ParameterizingModel} \rightarrow \text{Data}$$
 - `PseudoParameters`

$$\text{Parameterized Data} \rightarrow \text{Data}$$

So, by using `PseudoParameters` class the gradients of the parameter are calculated in such a way that the `ParameterizingModel` was never present.

To convert parameters of a layer or model to use `PseudoParameters`, then use:

```
PseudoParameters.parameterize(Model, "parameter_name", ↵
↵ transformation=ParameterizingModel)
```

OR

```
PseudoParameters.parametrize_module(Model, transformation=ParameterizingModel)
```

1.5.2 Forward and Backward Graphs

Graph class: `analogvnn.graph.ModelGraph.ModelGraph()`

Forward Graph class: `analogvnn.graph.ForwardGraph.ForwardGraph()`

Backward Graph class: `analogvnn.graph.BackwardGraph.BackwardGraph()`

Documentation Coming Soon...

1.6 Extra Analog Classes

Some extra layers which can be found in AnalogVNN are as follows:

1.6.1 Reduce Precision

Reduce Precision classes are used to reduce precision of an input to some given precision level

ReducePrecision

class: `analogvnn.nn.precision.ReducePrecision.ReducePrecision`

Reduce Precision uses the following function to reduce precision of the input value

$$RP(x) = \text{sign}(x * p) * \max(\lfloor |x * p| \rfloor, \lceil |x * p| - d \rceil) * \frac{1}{p}$$

where:

- x is the original number in full precision
- p is the analog precision of the input signal, output signal, or weights (p Natural Numbers, *Bit Precision* = $\log_2(p + 1)$)
- d is the divide parameter (0 ≤ d ≤ 1, default value = 0.5) which determines whether x is rounded to a discrete level higher or lower than the original value

StochasticReducePrecision

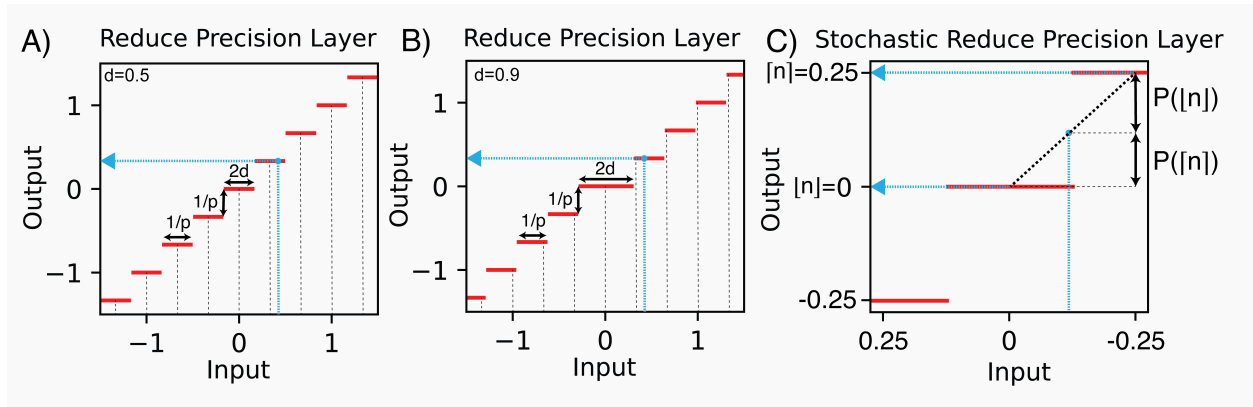
class: `analogvnn.nn.precision.StochasticReducePrecision.StochasticReducePrecision`

Reduce Precision uses the following probabilistic function to reduce precision of the input value

$$SRP(x) = \text{sign}(x * p) * f(|x * p|) * \frac{1}{p}$$
$$f(x) = \begin{cases} \lfloor x \rfloor & : r \leq 1 - |\lfloor x \rfloor - x| \\ \lceil x \rceil & : \text{otherwise} \end{cases}$$

where:

- r is a uniformly distributed random number between 0 and 1
- p is the analog precision (p Natural Numbers, *Bit Precision* = $\log_2(p + 1)$)
- f(x) is the stochastic rounding function



1.6.2 Normalization

LPNorm

class: `analogvnn.nn.normalize.LPNorm.LPNorm`

$$LPNorm(x) = \left[x_{ij..k} \rightarrow \frac{x_{ij..k}}{\sqrt[p]{\sum_{j..k} |x_{ij..k}|^p}} \right]$$

$$LPNormM(x) = \frac{LPNorm(x)}{\max(|LPNorm(x)|)}$$

where:

- x is the input weight matrix,
- $i, j \dots k$ are indexes of the matrix,
- p is a positive integer.

LPNormW

class: `analogvnn.nn.normalize.LPNorm.LPNormW`

$$LPNormW(x) = \frac{x}{\|x\|_p} = \frac{x}{\sqrt[p]{\sum |x|^p}}$$

$$LPNormWM(x) = \frac{LPNormW(x)}{\max(|LPNormW(x)|)}$$

where:

- x is the input weight matrix,
- p is a positive integer.

Clamp

class: `analogvnn.nn.normalize.Clamp.Clamp`

$$Clamp_{pq}(x) = \begin{cases} q & : qx \\ x & : p \leq x \leq q \\ p & : px \end{cases}$$

where:

- p, q ($p < q$, Default value for photonics $p = 1$ and $q = 1$)

1.6.3 Noise

Leakage

We have defined an information loss parameter, “Error Probability” or “EP” or “Leakage”, as the probability that a reduced precision digital value (e.g., “1011”) will acquire a different digital value (e.g., “1010” or “1100”) after passing through the noise layer (i.e., the probability that the digital values transmitted and detected are different after passing through the analog channel). This is a similar concept to the bit error ratio (BER) used in digital communications, but for numbers with multiple bits of resolution. While SNR (signal-to-noise ratio) is inversely proportional to sigma, the standard deviation of the signal noise, EP is indirectly proportional to . However, we choose EP since it provides a more intuitive understanding of the effect of noise in an analog system from a digital perspective. It is also similar to the rate parameter used in PyTorch’s Dropout Layer [23], though different in function. EP is defined as follows:

$$leakage = 1 - \frac{\int_{q=a}^b \int_{p=-\infty}^{\infty} \text{sign}(\delta(RP(p) - RP(q))) * PDF_{N_{RP(q)}}(p) dp dq}{|b - a|}$$

$$leakage = 1 - \frac{\int_{q=a}^b \int_{p=\max(RP(q)-\frac{s}{2}, a)}^{\min(RP(q)+\frac{s}{2}, b)} PDF_{N_{RP(q)}}(p) dp dq}{|b - a|}$$

$$leakage = 1 - \frac{1}{size(R_{RP}(a, b)) - 1} * \sum_{q \in S_{RP}(s, b)} \int_{p=\max(p-\frac{s}{2}, a)}^{\min(q+\frac{s}{2}, b)} PDF_{N_{RP(q)}}(p) dp$$

$$leakage = 1 - \frac{1}{size(R_{RP}(a, b)) - 1} * \sum_{q \in S_{RP}(s, b)} [CDF_{N_q}(p)]_{\max(p-\frac{s}{2}, a)}^{\min(q+\frac{s}{2}, b)}$$

For noise distributions invariant to linear transformations (e.g., Uniform, Normal, Laplace, etc.), the EP equation is as follows:

$$leakage = 2 * CDF_{N_0} \left(-\frac{1}{2 * p} \right)$$

where:

- leakage is in the range [0, 1]
- δ is the Dirac Delta function
- RP is the Reduce Precision function (for the above equation, $d=0.5$)
- s is the step width of reduce precision function
- $R_{RP}(a, b)$ is $\{x[a, b] | RP(x) = x\}$
- PDF_x is the probability density function for the noise distribution, x
- CDF_x is the cumulative density function for the noise distribution, x

- N_x is the noise function around point x . (for Gaussian Noise, x = mean and for Poisson Noise, x = rate)
- a, b are the limits of the analog signal domain (for photonics $a = 1$ and $b = 1$)

GaussianNoise

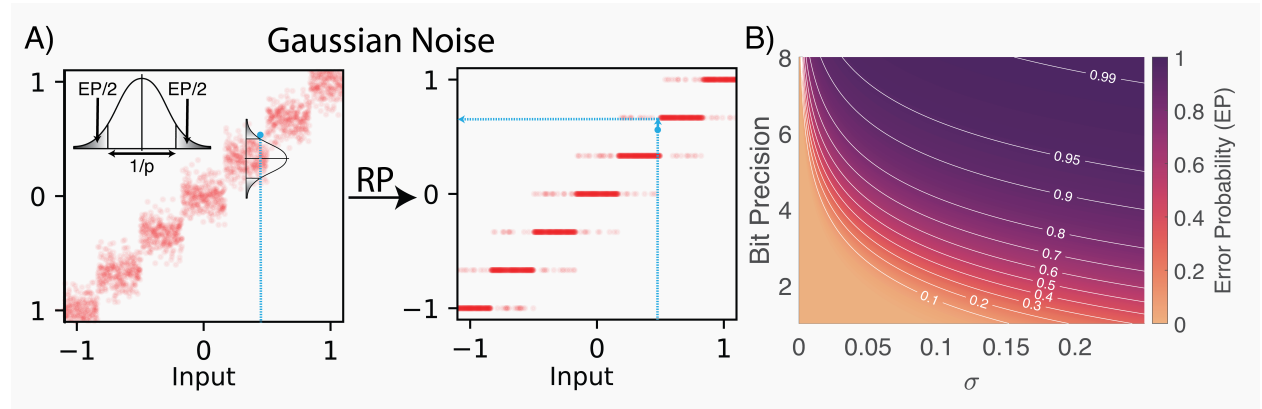
class: `analogvnn.nn.noise.GaussianNoise.GaussianNoise`

$$leakage = 1 - \operatorname{erf}\left(\frac{1}{2\sqrt{2} * \sigma * p}\right)$$

$$\sigma = \frac{1}{2\sqrt{2} * p * \operatorname{erf}^{-1}(1 - leakage)}$$

where:

- σ is the standard deviation of Gaussian Noise
- leakage is the error probability ($0 > leakage > 1$)
- erf is the Gauss Error Function
- p is precision



1.7 API Reference

This page contains auto-generated API reference documentation¹.

1.7.1 analogvnn

AnalogVNN: A fully modular framework for modeling and optimizing analog/photonic neural networks.

¹ Created with sphinx-autoapi

Subpackages

`analogvnn.backward`

Submodules

`analogvnn.backward.BackwardFunction`

Module Contents

Classes

<i>BackwardFunction</i>	The backward module that uses a function to compute the backward gradient.
-------------------------	--

```
class analogvnn.backward.BackwardFunction.BackwardFunction(backward_function:
                                                                analogvnn.utils.common_types.TENSOR_CALLABLE,
                                                                layer: torch.nn.Module = None)
```

Bases: *analogvnn.backward.BackwardModule.BackwardModule*, *abc.ABC*

The backward module that uses a function to compute the backward gradient.

Variables

_backward_function (*TENSOR_CALLABLE*) – The function used to compute the backward gradient.

property backward_function: `analogvnn.utils.common_types.TENSOR_CALLABLE`

The function used to compute the backward gradient.

Returns

The function used to compute the backward gradient.

Return type

`TENSOR_CALLABLE`

_backward_function: `analogvnn.utils.common_types.TENSOR_CALLABLE`

set_backward_function(backward_function: *analogvnn.utils.common_types.TENSOR_CALLABLE*) → *BackwardFunction*

Sets the function used to compute the backward gradient with.

Parameters

backward_function (*TENSOR_CALLABLE*) – The function used to compute the backward gradient with.

Returns

self.

Return type

BackwardFunction

backward(*grad_output: *torch.Tensor*, **grad_output_kward: *torch.Tensor*) → `analogvnn.utils.common_types.TENSORS`

Computes the backward gradient of inputs with respect to outputs using the backward function.

Parameters

- ***grad_output** (*Tensor*) – The gradients of the output of the layer.
- ****grad_output_kwarg** (*Tensor*) – The gradients of the output of the layer.

Returns

The gradients of the input of the layer.

Return type

TENSORS

Raises

NotImplementedError – If the backward function is not set.

`analogvnn.backward.BackwardIdentity`

Module Contents**Classes**

<i>BackwardIdentity</i>	The backward module that returns the output gradients as the input gradients.
-------------------------	---

class `analogvnn.backward.BackwardIdentity.BackwardIdentity`(*layer: torch.nn.Module = None*)

Bases: `analogvnn.backward.BackwardModule.BackwardModule`, `abc.ABC`

The backward module that returns the output gradients as the input gradients.

backward(**grad_output: torch.Tensor*, ***grad_output_kwarg: torch.Tensor*) → `analogvnn.utils.common_types.TENSORS`

Returns the output gradients as the input gradients.

Parameters

- ***grad_output** (*Tensor*) – The gradients of the output of the layer.
- ****grad_output_kwarg** (*Tensor*) – The gradients of the output of the layer.

Returns

The gradients of the input of the layer.

Return type

TENSORS

`analogvnn.backward.BackwardModule`

Module Contents**Classes**

<i>BackwardModule</i>	Base class for all backward modules.
-----------------------	--------------------------------------

```
class analogvnn.backward.BackwardModule.BackwardModule(layer: torch.nn.Module = None)
```

Bases: `abc.ABC`

Base class for all backward modules.

A backward module is a module that can be used to compute the backward gradient of a given function. It is used to compute the gradient of the input of a function with respect to the output of the function.

Variables

- `_layer` (*Optional*[`nn.Module`]) – The layer for which the backward gradient is computed.
- `_empty_holder_tensor` (*Tensor*) – A placeholder tensor which always requires gradient for backward gradient computation.
- `_autograd_backward` (*Type*[`AutogradBackward`]) – The autograd backward function.
- `_disable_autograd_backward` (*bool*) – If True the autograd backward function is disabled.

```
class AutogradBackward
```

Bases: `torch.autograd.Function`

Optimization and proper calculation of gradients when using the autograd engine.

```
static forward(ctx: Any, backward_module: BackwardModule, _: torch.Tensor, *args: torch.Tensor,
               **kwargs: torch.Tensor) → analogvnn.utils.common_types.TENSORS
```

Forward pass of the autograd function.

Parameters

- `ctx` – The context of the autograd function.
- `backward_module` (`BackwardModule`) – The backward module.
- `_` (*Tensor*) – placeholder tensor which always requires grad.
- `*args` (*Tensor*) – The arguments of the function.
- `**kwargs` (*Tensor*) – The keyword arguments of the function.

Returns

The output of the function.

Return type

TENSORS

```
static backward(ctx: Any, *grad_outputs: torch.Tensor) → Tuple[None, None,
                    analogvnn.utils.common_types.TENSORS]
```

Backward pass of the autograd function.

Parameters

- `ctx` – The context of the autograd function.
- `*grad_outputs` (*Tensor*) – The gradients of the output of the function.

Returns

The gradients of the input of the function.

Return type

TENSORS

```
property layer: Optional[torch.nn.Module]
```

Gets the layer for which the backward gradient is computed.

Returns

layer

Return type

`Optional`[`nn.Module`]

```

_layer: Optional[torch.nn.Module]
_empty_holder_tensor: torch.Tensor
_autograd_backward: Type[AutogradBackward]
_disable_autograd_backward: bool = False
__call__: Callable[Ellipsis, Any]

abstract forward(*inputs: torch.Tensor, **inputs_kwarg: torch.Tensor) →
    analogvnn.utils.common_types.TENSORS

```

Forward pass of the layer.

Parameters

- ***inputs** (*Tensor*) – The inputs of the layer.
- ****inputs_kwarg** (*Tensor*) – The keyword inputs of the layer.

Returns

The output of the layer.

Return type

TENSORS

Raises

NotImplementedError – If the forward pass is not implemented.

```

abstract backward(*grad_outputs: torch.Tensor, **grad_output_kwarg: torch.Tensor) →
    analogvnn.utils.common_types.TENSORS

```

Backward pass of the layer.

Parameters

- ***grad_outputs** (*Tensor*) – The gradients of the output of the layer.
- ****grad_output_kwarg** (*Tensor*) – The keyword gradients of the output of the layer.

Returns

The gradients of the input of the layer.

Return type

TENSORS

Raises

NotImplementedError – If the backward pass is not implemented.

```

_call_impl_forward(*args: torch.Tensor, **kwargs: torch.Tensor) →
    analogvnn.utils.common_types.TENSORS

```

Calls Forward pass of the layer.

Parameters

- ***inputs** (*Tensor*) – The inputs of the layer.
- ****inputs_kwarg** (*Tensor*) – The keyword inputs of the layer.

Returns

The output of the layer.

Return type

TENSORS

_call_impl_backward(*grad_output: *torch.Tensor*, **grad_output_kwarg: *torch.Tensor*) →
analogvnn.utils.common_types.TENSORS

Calls Backward pass of the layer.

Parameters

- ***grad_outputs** (*Tensor*) – The gradients of the output of the layer.
- ****grad_output_kwarg** (*Tensor*) – The keyword gradients of the output of the layer.

Returns

The gradients of the input of the layer.

Return type

TENSORS

auto_apply(*args: *torch.Tensor*, to_apply=True, **kwargs: *torch.Tensor*) →
analogvnn.utils.common_types.TENSORS

Applies the backward module to the given layer using the proper method.

Parameters

- ***args** (*Tensor*) – The inputs of the layer.
- **to_apply** (*bool*) – if True and is training then the AutogradBackward is applied,
- **applied.** (*otherwise the backward module is*) –
- ****kwargs** (*Tensor*) – The keyword inputs of the layer.

Returns

The output of the layer.

Return type

TENSORS

has_forward() → *bool*

Checks if the forward pass is implemented.

Returns

True if the forward pass is implemented, False otherwise.

Return type

bool

get_layer() → Optional[*torch.nn.Module*]

Gets the layer for which the backward gradient is computed.

Returns

layer

Return type

Optional[*nn.Module*]

set_layer(layer: Optional[*torch.nn.Module*]) → *BackwardModule*

Sets the layer for which the backward gradient is computed.

Parameters

layer (*nn.Module*) – The layer for which the backward gradient is computed.

Returns

self

Return type*BackwardModule***Raises**

- **ValueError** – If self is a subclass of nn.Module.
- **ValueError** – If the layer is already set.
- **ValueError** – If the layer is not an instance of nn.Module.

_set_autograd_backward()**static set_grad_of**(*tensor: torch.Tensor, grad: torch.Tensor*) → Optional[torch.Tensor]

Sets the gradient of the given tensor.

Parameters

- **tensor** (*Tensor*) – The tensor.
- **grad** (*Tensor*) – The gradient.

Returns

the gradient of the tensor.

Return type

Optional[Tensor]

__getattr__(*name: str*) → Any

Gets the attribute of the layer.

Parameters**name** (*str*) – The name of the attribute.**Returns**

The attribute of the layer.

Return type

Any

Raises**AttributeError** – If the attribute is not found.**analogvnn.backward.BackwardUsingForward****Module Contents****Classes***BackwardUsingForward*

The backward module that uses the forward function to compute the backward gradient.

class analogvnn.backward.BackwardUsingForward.**BackwardUsingForward**(*layer: torch.nn.Module = None*)

Bases: *analogvnn.backward.BackwardModule.BackwardModule*, *abc.ABC*

The backward module that uses the forward function to compute the backward gradient.

backward(*grad_output: *torch.Tensor*, **grad_output_kwarg: *torch.Tensor*) →
analogvnn.utils.common_types.TENSORS

Computes the backward gradient of inputs with respect to outputs using the forward function.

Parameters

- ***grad_output** (*Tensor*) – The gradients of the output of the layer.
- ****grad_output_kwarg** (*Tensor*) – The gradients of the output of the layer.

Returns

The gradients of the input of the layer.

Return type

TENSORS

analogvnn.fn

Additional functions for analogvnn.

Submodules

analogvnn.fn.dirac_delta

Module Contents

Functions

<i>gaussian_dirac_delta</i> (...)	Gaussian Dirac Delta function with standard deviation <i>std</i> .
-----------------------------------	--

analogvnn.fn.dirac_delta.**gaussian_dirac_delta**(x:
analogvnn.utils.common_types.TENSOR_OPERABLE,
std:
analogvnn.utils.common_types.TENSOR_OPERABLE =
0.001) →
analogvnn.utils.common_types.TENSOR_OPERABLE

Gaussian Dirac Delta function with standard deviation *std*.

Parameters

- **x** (*TENSOR_OPERABLE*) – Tensor
- **std** (*TENSOR_OPERABLE*) – standard deviation.

Returns

TENSOR_OPERABLE with the same shape as x, with values of the Gaussian Dirac Delta function.

Return type

TENSOR_OPERABLE

`analogvnn.fn.reduce_precision`

Module Contents

Functions

<code>reduce_precision(...)</code>	Takes x and reduces its precision to $precision$ by rounding to the nearest multiple of $precision$.
<code>stochastic_reduce_precision(...)</code>	Takes x and reduces its precision by rounding to the nearest multiple of $precision$ with stochastic scheme.

`analogvnn.fn.reduce_precision.reduce_precision(x:`
 `analogvnn.utils.common_types.TENSOR_OPERABLE,`
 `precision:`
 `analogvnn.utils.common_types.TENSOR_OPERABLE,`
 `divide:`
 `analogvnn.utils.common_types.TENSOR_OPERABLE)`
 $\rightarrow$
 `analogvnn.utils.common_types.TENSOR_OPERABLE`

Takes x and reduces its precision to $precision$ by rounding to the nearest multiple of $precision$.

Parameters

- **x** (`TENSOR_OPERABLE`) – Tensor
- **precision** (`TENSOR_OPERABLE`) – the precision of the quantization.
- **divide** (`TENSOR_OPERABLE`) – the number of bits to be reduced

Returns

`TENSOR_OPERABLE` with the same shape as x , but with values rounded to the nearest multiple of precision.

Return type

`TENSOR_OPERABLE`

`analogvnn.fn.reduce_precision.stochastic_reduce_precision(x:`
 `analogvnn.utils.common_types.TENSOR_OPERABLE,`
 `precision:`
 `analogvnn.utils.common_types.TENSOR_OPERABLE)`
 $\rightarrow$
 `analogvnn.utils.common_types.TENSOR_OPERABLE`

Takes x and reduces its precision by rounding to the nearest multiple of $precision$ with stochastic scheme.

Parameters

- **x** (`TENSOR_OPERABLE`) – Tensor
- **precision** (`TENSOR_OPERABLE`) – the precision of the quantization.

Returns

`TENSOR_OPERABLE` with the same shape as x , but with values rounded to the nearest multiple of precision.

Return type

`TENSOR_OPERABLE`

`analogvnn.fn.test`

Module Contents

Functions

<code>test(→ Tuple[float, float])</code>	Test the model on the test set.
--	---------------------------------

`analogvnn.fn.test.test(model: torch.nn.Module, test_loader: torch.utils.data.DataLoader, test_run: bool = False) → Tuple[float, float]`

Test the model on the test set.

Parameters

- **model** (*torch.nn.Module*) – the model to test.
- **test_loader** (*DataLoader*) – the test set.
- **test_run** (*bool*) – is it a test run.

Returns

the loss and accuracy of the model on the test set.

Return type

tuple

`analogvnn.fn.to_matrix`

Module Contents

Functions

<code>to_matrix(→ torch.Tensor)</code>	<i>to_matrix</i> takes a tensor and returns a matrix with the same values as the tensor.
--	--

`analogvnn.fn.to_matrix.to_matrix(tensor: torch.Tensor) → torch.Tensor`

to_matrix takes a tensor and returns a matrix with the same values as the tensor.

Parameters

tensor (*Tensor*) – Tensor

Returns

Tensor with the same values as the tensor, but with shape (1, -1).

Return type

Tensor

`analogvnn.fn.train`

Module Contents

Functions

<code>train(→ Tuple[float, float])</code>	Train the model on the train set.
---	-----------------------------------

`analogvnn.fn.train.train(model: torch.nn.Module, train_loader: torch.utils.data.DataLoader, epoch: Optional[int] = None, test_run: bool = False) → Tuple[float, float]`

Train the model on the train set.

Parameters

- **model** (*torch.nn.Module*) – the model to train.
- **train_loader** (*DataLoader*) – the train set.
- **epoch** (*int*) – the current epoch.
- **test_run** (*bool*) – is it a test run.

Returns

the loss and accuracy of the model on the train set.

Return type

tuple

`analogvnn.graph`

Submodules

`analogvnn.graph.AccumulateGrad`

Module Contents

Classes

<code><i>AccumulateGrad</i></code>	<code>AccumulateGrad</code> is a module that accumulates the gradients of the outputs of the module it is attached to.
------------------------------------	--

class `analogvnn.graph.AccumulateGrad.AccumulateGrad(module: Union[torch.nn.Module, Callable])`

`AccumulateGrad` is a module that accumulates the gradients of the outputs of the module it is attached to.

It has no parameters of its own.

Variables

- **module** (*nn.Module*) – Module to accumulate gradients for.
- **input_output_connections** (*Dict[str, Dict[str, Union[None, bool, int, str, GRAPH_NODE_TYPE]]]*) – input/output
- **connections.** –

input_output_connections: Dict[str, Dict[str, Union[None, bool, int, str, analogvnn.graph.GraphEnum.GRAPH_NODE_TYPE]]]

module: Union[torch.nn.Module, Callable]

grad
Alias for `__call__`.

__repr__()
Return a string representation of the module.

Returns
String representation of the module.

Return type
str

__call__ (*grad_outputs_args_kwargs: analogvnn.graph.ArgsKwargs.ArgsKwargs, forward_input_output_graph: Dict[analogvnn.graph.GraphEnum.GRAPH_NODE_TYPE, analogvnn.graph.ArgsKwargs.InputOutput]*) → *analogvnn.graph.ArgsKwargs.ArgsKwargs*
Calculate and Accumulate the output gradients of the module.

Parameters

- **grad_outputs_args_kwargs** (*ArgsKwargs*) – The output gradients from previous modules (predecessors).
- **forward_input_output_graph** (*Dict[GRAPH_NODE_TYPE, InputOutput]*) – The input and output from forward pass.

Returns
The output gradients.

Return type
ArgsKwargs

`analogvnn.graph.AcyclicDirectedGraph`

Module Contents

Classes

<i>AcyclicDirectedGraph</i>	The base class for all acyclic directed graphs.
-----------------------------	---

```
class analogvnn.graph.AcyclicDirectedGraph.AcyclicDirectedGraph(graph_state:
                                                                    analogvnn.graph.ModelGraphState.ModelGraphState
                                                                    = None)
```

Bases: `abc.ABC`

The base class for all acyclic directed graphs.

Variables

- **graph** (*nx.MultiDiGraph*) – The graph.
- **graph_state** (*ModelGraphState*) – The graph state.
- **_is_static** (*bool*) – If True, the graph is not changing during runtime and will be cached.

- **_static_graphs** (*Dict[GRAPH_NODE_TYPE, List[Tuple[GRAPH_NODE_TYPE, List[GRAPH_NODE_TYPE]]]*) – The static graphs.
- **INPUT** (*GraphEnum*) – GraphEnum.INPUT
- **OUTPUT** (*GraphEnum*) – GraphEnum.OUTPUT
- **STOP** (*GraphEnum*) – GraphEnum.STOP

graph: *networkx.MultiDiGraph*

graph_state: *analogvnn.graph.ModelGraphState.ModelGraphState*

_is_static: *bool*

_static_graphs: *Dict[analogvnn.graph.GraphEnum.GRAPH_NODE_TYPE, List[Tuple[analogvnn.graph.GraphEnum.GRAPH_NODE_TYPE, List[analogvnn.graph.GraphEnum.GRAPH_NODE_TYPE]]]*

INPUT

OUTPUT

STOP

save

Alias for render.

abstract __call__ (**args, **kwargs*)

Performs pass through the graph.

Parameters

- ***args** – Arguments
- ****kwargs** – Keyword arguments

Raises

NotImplementedError – since method is abstract

add_connection (**args: analogvnn.graph.GraphEnum.GRAPH_NODE_TYPE*)

Add a connection between nodes.

Parameters

***args** – The nodes.

Returns

self.

Return type

AcyclicDirectedGraph

add_edge (*u_of_edge: analogvnn.graph.GraphEnum.GRAPH_NODE_TYPE, v_of_edge: analogvnn.graph.GraphEnum.GRAPH_NODE_TYPE, in_arg: Union[None, int, bool] = None, in_kwarg: Union[None, str, bool] = None, out_arg: Union[None, int, bool] = None, out_kwarg: Union[None, str, bool] = None*)

Add an edge to the graph.

Parameters

- **u_of_edge** (*GRAPH_NODE_TYPE*) – The source node.
- **v_of_edge** (*GRAPH_NODE_TYPE*) – The target node.

- **in_arg** (*Union[None, int, bool]*) – The input argument.
- **in_kwarg** (*Union[None, str, bool]*) – The input keyword argument.
- **out_arg** (*Union[None, int, bool]*) – The output argument.
- **out_kwarg** (*Union[None, str, bool]*) – The output keyword argument.

Returns

self.

Return type

AcyclicDirectedGraph

```
static check_edge_parameters(in_arg: Union[None, int, bool], in_kwarg: Union[None, str, bool],
                             out_arg: Union[None, int, bool], out_kwarg: Union[None, str, bool]) →
    Dict[str, Union[None, int, str, bool]]
```

Check the edge's in and out parameters.

Parameters

- **in_arg** (*Union[None, int, bool]*) – The input argument.
- **in_kwarg** (*Union[None, str, bool]*) – The input keyword argument.
- **out_arg** (*Union[None, int, bool]*) – The output argument.
- **out_kwarg** (*Union[None, str, bool]*) – The output keyword argument.

Returns

Dict of valid edge's in and out parameters.

Return type

Dict[str, Union[None, int, str, bool]]

Raises

ValueError – If in and out parameters are invalid.

```
static _create_edge_label(in_arg: Union[None, int, bool] = None, in_kwarg: Union[None, str, bool] =
                           None, out_arg: Union[None, int, bool] = None, out_kwarg: Union[None,
                           str, bool] = None, **kwargs) → str
```

Create the edge's label.

Parameters

- **in_arg** (*Union[None, int, bool]*) – The input argument.
- **in_kwarg** (*Union[None, str, bool]*) – The input keyword argument.
- **out_arg** (*Union[None, int, bool]*) – The output argument.
- **out_kwarg** (*Union[None, str, bool]*) – The output keyword argument.

Returns

The edge's label.

Return type

str

```
compile(is_static: bool = True)
```

Compile the graph.

Parameters

is_static (*bool*) – If True, the graph will be compiled as a static graph.

Returns

The compiled graph.

Return type

AcyclicDirectedGraph

Raises

ValueError – If the graph is not acyclic.

static `_reindex_out_args`(*graph*: *networkx.MultiDiGraph*) → *networkx.MultiDiGraph*

Reindex the output arguments.

Parameters

graph (*nx.MultiDiGraph*) – The graph.

Returns

The graph with re-indexed output arguments.

Return type

nx.MultiDiGraph

_create_static_sub_graph(*from_node*: *analogvnn.graph.GraphEnum.GRAPH_NODE_TYPE*) →
 List[Tuple[*analogvnn.graph.GraphEnum.GRAPH_NODE_TYPE*,
 List[*analogvnn.graph.GraphEnum.GRAPH_NODE_TYPE*]]]

Create a static sub graph connected to the given node.

Parameters

from_node (*GRAPH_NODE_TYPE*) – The node.

Returns

The static sub graph.

Return type

List[Tuple[*GRAPH_NODE_TYPE*, List[*GRAPH_NODE_TYPE*]]]

parse_args_kwargs(*input_output_graph*: Dict[*analogvnn.graph.GraphEnum.GRAPH_NODE_TYPE*,
analogvnn.graph.ArgsKwargs.InputOutput], *module*:
analogvnn.graph.GraphEnum.GRAPH_NODE_TYPE, *predecessors*:
 List[*analogvnn.graph.GraphEnum.GRAPH_NODE_TYPE*]) →
analogvnn.graph.ArgsKwargs.ArgsKwargs

Parse the arguments and keyword arguments.

Parameters

- **input_output_graph** (Dict[*GRAPH_NODE_TYPE*, *InputOutput*]) – The input output graph.
- **module** (*GRAPH_NODE_TYPE*) – The module.
- **predecessors** (List[*GRAPH_NODE_TYPE*]) – The predecessors.

Returns

The arguments and keyword arguments.

Return type

ArgsKwargs

render(**args*, *real_label*: *bool* = *False*, ***kwargs*) → *str*

Save the source to file and render with the Graphviz engine.

Parameters

- ***args** – Arguments to pass to graphviz render function.

- **real_label** – If True, the real label will be used instead of the label.
- ****kwargs** – Keyword arguments to pass to graphviz render function.

Returns

The (possibly relative) path of the rendered file.

Return type

str

`analogvnn.graph.ArgsKwargs`

Module Contents**Classes**

<i>InputOutput</i>	Inputs and outputs of a module.
<i>ArgsKwargs</i>	The arguments.

Attributes

<i>ArgsKwargsInput</i>	ArgsKwargsInput is the input type for ArgsKwargs
<i>ArgsKwargsOutput</i>	ArgsKwargsOutput is the output type for ArgsKwargs

class `analogvnn.graph.ArgsKwargs.InputOutput`

Inputs and outputs of a module.

Variables

- **inputs** (*Optional*[[*ArgsKwargs*](#)]) – Inputs of a module.
- **outputs** (*Optional*[[*ArgsKwargs*](#)]) – Outputs of a module.

inputs: *Optional*[[*ArgsKwargs*](#)]

outputs: *Optional*[[*ArgsKwargs*](#)]

class `analogvnn.graph.ArgsKwargs.ArgsKwargs`(*args=None, kwargs=None*)

The arguments.

Variables

- **args** (*List*) – The arguments.
- **kwargs** (*Dict*) – The keyword arguments.

args: *List*

kwargs: *Dict*

is_empty()

Returns whether the ArgsKwargs object is empty.

__repr__()

Returns a string representation of the parameter.

classmethod `to_args_kwargs_object(outputs: ArgsKwargsInput) → ArgsKwargs`

Convert the output of a module to ArgsKwargs object.

Parameters

outputs – The output of a module

Returns

The ArgsKwargs object

Return type

ArgsKwargs

static `from_args_kwargs_object(outputs: ArgsKwargs) → ArgsKwargsOutput`

Convert ArgsKwargs to object or tuple or dict.

Parameters

outputs (*ArgsKwargs*) – ArgsKwargs object

Returns

object or tuple or dict

Return type

ArgsKwargsOutput

`analogvnn.graph.ArgsKwargs.ArgsKwargsInput`

ArgsKwargsInput is the input type for ArgsKwargs

`analogvnn.graph.ArgsKwargs.ArgsKwargsOutput`

ArgsKwargsOutput is the output type for ArgsKwargs

`analogvnn.graph.BackwardGraph`

Module Contents

Classes

BackwardGraph

The backward graph.

class `analogvnn.graph.BackwardGraph.BackwardGraph(graph_state:`

`analogvnn.graph.ModelGraphState.ModelGraphState
= None)`

Bases: `analogvnn.graph.AcyclicDirectedGraph.AcyclicDirectedGraph`

The backward graph.

__call__ (`gradient: analogvnn.utils.common_types.TENSORS = None`) →
`analogvnn.graph.ArgsKwargs.ArgsKwargsOutput`

Backward pass through the backward graph.

Parameters

gradient (*TENSORS*) – gradient of the loss function w.r.t. the output of the forward graph

Returns

gradient of the inputs function w.r.t. loss

Return type

ArgsKwargsOutput

compile(*is_static=True*)

Compile the graph.

Parameters

is_static (*bool*) – If True, the graph is not changing during runtime and will be cached.

Returns

self.

Return type

BackwardGraph

Raises

ValueError – If no forward pass has been performed yet.

from_forward(*forward_graph: Union[analogvnn.graph.AcyclicDirectedGraph, networkx.DiGraph]*) → *BackwardGraph*

Create a backward graph from inverting forward graph.

Parameters

forward_graph (*Union[AcyclicDirectedGraph, nx.DiGraph]*) – The forward graph.

Returns

self.

Return type

BackwardGraph

calculate(**args, **kwargs*) → *analogvnn.graph.ArgsKwargs.ArgsKwargsOutput*

Calculate the gradient of the whole graph w.r.t. loss.

Parameters

- ***args** – The gradients args of outputs.
- ****kwargs** – The gradients kwargs of outputs.

Returns

The gradient of the inputs function w.r.t. loss.

Return type

ArgsKwargsOutput

Raises

ValueError – If no forward pass has been performed yet.

_pass(*from_node: analogvnn.graph.GraphEnum.GRAPH_NODE_TYPE, *args, **kwargs*) → *Dict[analogvnn.graph.GraphEnum.GRAPH_NODE_TYPE, analogvnn.graph.ArgsKwargs.InputOutput]*

Perform the backward pass through the graph.

Parameters

- **from_node** (*GRAPH_NODE_TYPE*) – The node to start the backward pass from.
- ***args** – The gradients args of outputs.
- ****kwargs** – The gradients kwargs of outputs.

Returns

The input and output gradients of each node.

Return type

Dict[GRAPH_NODE_TYPE, InputOutput]

```
_calculate_gradients(module: Union[analogvnn.graph.AccumulateGrad.AccumulateGrad,
                                analogvnn.nn.module.Layer.Layer,
                                analogvnn.backward.BackwardModule.BackwardModule, Callable], grad_outputs:
                                analogvnn.graph.ArgsKwargs.InputOutput) →
                                analogvnn.graph.ArgsKwargs.ArgsKwargs
```

Calculate the gradient of a module w.r.t. outputs of the module using the output's gradients.

Parameters

- **module** (*Union[AccumulateGrad, Layer, BackwardModule, Callable]*) – The module to calculate the gradient of.
- **grad_outputs** (*InputOutput*) – The gradients of the output of the module.

Returns

The input gradients of the module.

Return type

ArgsKwargs

analogvnn.graph.ForwardGraph

Module Contents

Classes

<i>ForwardGraph</i>	The forward graph.
---------------------	--------------------

```
class analogvnn.graph.ForwardGraph.ForwardGraph(graph_state:
                                analogvnn.graph.ModelGraphState.ModelGraphState
                                = None)
```

Bases: *analogvnn.graph.AcyclicDirectedGraph.AcyclicDirectedGraph*

The forward graph.

```
__call__(inputs: analogvnn.utils.common_types.TENSORS, is_training: bool) →
                                analogvnn.graph.ArgsKwargs.ArgsKwargsOutput
```

Forward pass through the forward graph.

Parameters

- **inputs** (*TENSORS*) – Input to the graph
- **is_training** (*bool*) – Is training or not

Returns

Output of the graph

Return type

ArgsKwargsOutput

```
compile(is_static: bool = True)
```

Compile the graph.

Parameters

- **is_static** (*bool*) – If True, the graph is not changing during runtime and will be cached.

Returns

self.

Return type

ForwardGraph

Raises

ValueError – If no forward pass has been performed yet.

calculate(inputs: *analogvnn.utils.common_types.TENSORS*, is_training: *bool = True*, **kwargs) → *analogvnn.graph.ArgsKwargs.ArgsKwargsOutput*

Calculate the output of the graph.

Parameters

- **inputs** (*TENSORS*) – Input to the graph
- **is_training** (*bool*) – Is training or not
- ****kwargs** – Additional arguments

Returns

Output of the graph

Return type

ArgsKwargsOutput

_pass(from_node: *analogvnn.graph.GraphEnum.GraphEnum*, *inputs: *torch.Tensor*) → *Dict[analogvnn.graph.GraphEnum.GraphEnum, analogvnn.graph.ArgsKwargs.InputOutput]*

Perform the forward pass through the graph.

Parameters

- **from_node** (*GraphEnum*) – The node to start the forward pass from
- ***inputs** (*Tensor*) – Input to the graph

Returns

The input and output of each node

Return type

Dict[GraphEnum, InputOutput]

static _detach_tensor(tensor: *torch.Tensor*) → *torch.Tensor*

Detach the tensor from the autograd graph.

Parameters

tensor (*torch.Tensor*) – Tensor to detach

Returns

Detached tensor

Return type

torch.Tensor

`analogvnn.graph.GraphEnum`

Module Contents

Classes

<i>GraphEnum</i>	The graph enum for indicating input, output and stop.
------------------	---

Attributes

<i>GRAPH_NODE_TYPE</i>

class `analogvnn.graph.GraphEnum.GraphEnum`Bases: `enum.Enum`

The graph enum for indicating input, output and stop.

Variables

- **INPUT** (`GraphEnum`) – `GraphEnum.INPUT`
- **OUTPUT** (`GraphEnum`) – `GraphEnum.OUTPUT`
- **STOP** (`GraphEnum`) – `GraphEnum.STOP`

`INPUT = 'INPUT'``OUTPUT = 'OUTPUT'``STOP = 'STOP'``analogvnn.graph.GraphEnum.GRAPH_NODE_TYPE``analogvnn.graph.ModelGraph`

Module Contents

Classes

<i>ModelGraph</i>	Store model's graph.
-------------------	----------------------

class `analogvnn.graph.ModelGraph.ModelGraph`(*use_autograd_graph*: `bool = False`, *allow_loops*: `bool = False`)Bases: `analogvnn.graph.ModelGraphState.ModelGraphState`

Store model's graph.

Variables

- **forward_graph** (`ForwardGraph`) – store model's forward graph.

- **backward_graph** (*BackwardGraph*) – store model’s backward graph.

forward_graph: *analogvnn.graph.ForwardGraph.ForwardGraph*

backward_graph: *analogvnn.graph.BackwardGraph.BackwardGraph*

compile(*is_static: bool = True, auto_backward_graph: bool = False*) → *ModelGraph*

Compile the model graph.

Parameters

- **is_static** (*bool*) – If True, the model graph is static.
- **auto_backward_graph** (*bool*) – If True, the backward graph is automatically created.

Returns

self.

Return type

ModelGraph

analogvnn.graph.ModelGraphState

Module Contents

Classes

<i>ModelGraphState</i>	The state of a model graph.
------------------------	-----------------------------

class *analogvnn.graph.ModelGraphState.ModelGraphState*(*use_autograd_graph: bool = False, allow_loops=False*)

The state of a model graph.

Variables

- **allow_loops** (*bool*) – if True, the graph is allowed to contain loops.
- **forward_input_output_graph** (*Optional[Dict[GRAPH_NODE_TYPE, InputOutput]]*) – the input and output of the
- **pass.** (*forward*) –
- **use_autograd_graph** (*bool*) – if True, the autograd graph is used to calculate the gradients.
- **_loss** (*Tensor*) – the loss.
- **INPUT** (*GraphEnum*) – *GraphEnum.INPUT*
- **OUTPUT** (*GraphEnum*) – *GraphEnum.OUTPUT*
- **STOP** (*GraphEnum*) – *GraphEnum.STOP*

Properties:

input (*Tensor*): the input of the forward pass. output (*Tensor*): the output of the forward pass. loss (*Tensor*): the loss.

property inputs: Optional[[analogvnn.graph.ArgsKwargs.ArgsKwargs](#)]

Get the inputs.

Returns

the inputs.

Return type

[ArgsKwargs](#)

property outputs: Optional[[analogvnn.graph.ArgsKwargs.ArgsKwargs](#)]

Get the output.

Returns

the output.

Return type

[ArgsKwargs](#)

property loss

Get the loss.

Returns

the loss.

Return type

Tensor

allow_loops: bool

use_autograd_graph: bool

forward_input_output_graph:

Optional[Dict[[analogvnn.graph.GraphEnum.GRAPH_NODE_TYPE](#),
[analogvnn.graph.ArgsKwargs.InputOutput](#)]]

_loss: Optional[[torch.Tensor](#)]

INPUT

OUTPUT

STOP

ready_for_forward(*exception: bool = False*) → bool

Check if the state is ready for forward pass.

Parameters

exception (bool) – If True, an exception is raised if the state is not ready for forward pass.

Returns

True if the state is ready for forward pass.

Return type

bool

Raises

[RuntimeError](#) – If the state is not ready for forward pass and exception is True.

ready_for_backward(*exception: bool = False*) → bool

Check if the state is ready for backward pass.

Parameters

exception (*bool*) – if True, raise an exception if the state is not ready for backward pass.

Returns

True if the state is ready for backward pass.

Return type

bool

Raises

RuntimeError – if the state is not ready for backward pass and exception is True.

set_loss (*loss: Union[torch.Tensor, None]*) → *ModelGraphState*

Set the loss.

Parameters

loss (*Tensor*) – the loss.

Returns

self.

Return type

ModelGraphState

`analogvnn.graph.to_graph_viz_digraph`

Module Contents**Functions**

<code>to_graphviz_digraph</code> (→ <i>graphviz.Digraph</i>)	Returns a pygraphviz graph from a NetworkX graph N.
---	---

`analogvnn.graph.to_graph_viz_digraph.to_graphviz_digraph` (*from_graph: networkx.DiGraph*,
real_label: bool = False) →
graphviz.Digraph

Returns a pygraphviz graph from a NetworkX graph N.

Parameters

- **from_graph** (*networkx.DiGraph*) – the graph to convert.
- **real_label** (*bool*) – True to use the real label.

Returns

the converted graph.

Return type

graphviz.Digraph

Raises

ImportError – if graphviz (<https://pygraphviz.github.io/>) is not available.

`analogvnn.nn`

Subpackages

`analogvnn.nn.activation`

Submodules

`analogvnn.nn.activation.Activation`

Module Contents

Classes

<i><code>InitImplement</code></i>	Implements the initialisation of parameters using the activation function.
<i><code>Activation</code></i>	This class is base class for all activation functions.

class `analogvnn.nn.activation.Activation.InitImplement`

Implements the initialisation of parameters using the activation function.

static `initialise(tensor: torch.Tensor) → torch.Tensor`

Initialisation of tensor using xavier uniform initialisation.

Parameters**tensor** (*Tensor*) – the tensor to be initialized.**Returns**

the initialized tensor.

Return type

Tensor

static `initialise_(tensor: torch.Tensor) → torch.Tensor`

In-place initialisation of tensor using xavier uniform initialisation.

Parameters**tensor** (*Tensor*) – the tensor to be initialized.**Returns**

the initialized tensor.

Return type

Tensor

class `analogvnn.nn.activation.Activation.Activation`Bases: `analogvnn.nn.module.Layer.Layer`, `analogvnn.backward.BackwardModule.BackwardModule`, `InitImplement`, `abc.ABC`

This class is base class for all activation functions.

`analogvnn.nn.activation.BinaryStep`

Module Contents

Classes

<i>BinaryStep</i>	Implements the binary step activation function.
-------------------	---

class `analogvnn.nn.activation.BinaryStep.BinaryStep`

Bases: `analogvnn.nn.activation.Activation.Activation`

Implements the binary step activation function.

static forward(*x*: `torch.Tensor`) → `torch.Tensor`

Forward pass of the binary step activation function.

Parameters

x (*Tensor*) – the input tensor.

Returns

the output tensor.

Return type

`Tensor`

backward(*grad_output*: `Optional[torch.Tensor]`) → `Optional[torch.Tensor]`

Backward pass of the binary step activation function.

Parameters

grad_output (*Optional[Tensor]*) – the gradient of the output tensor.

Returns

the gradient of the input tensor.

Return type

`Optional[Tensor]`

`analogvnn.nn.activation.ELU`

Module Contents

Classes

<i>SELU</i>	Implements the scaled exponential linear unit (SELU) activation function.
<i>ELU</i>	Implements the exponential linear unit (ELU) activation function.

class `analogvnn.nn.activation.ELU.SELU(alpha: float = 1.0507, scale_factor: float = 1.0)`

Bases: `analogvnn.nn.activation.Activation.Activation`

Implements the scaled exponential linear unit (SELU) activation function.

Variables

- **alpha** (*nn.Parameter*) – the alpha parameter.
- **scale_factor** (*nn.Parameter*) – the scale factor parameter.

__constants__ = ['alpha', 'scale_factor']

alpha: *torch.nn.Parameter*

scale_factor: *torch.nn.Parameter*

forward(*x*: *torch.Tensor*) → *torch.Tensor*

Forward pass of the scaled exponential linear unit (SELU) activation function.

Parameters

x (*Tensor*) – the input tensor.

Returns

the output tensor.

Return type

Tensor

backward(*grad_output*: *Optional[torch.Tensor]*) → *Optional[torch.Tensor]*

Backward pass of the scaled exponential linear unit (SELU) activation function.

Parameters

grad_output (*Optional[Tensor]*) – the gradient of the output tensor.

Returns

the gradient of the input tensor.

Return type

Optional[Tensor]

static initialise(*tensor*: *torch.Tensor*) → *torch.Tensor*

Initialisation of tensor using xavier uniform, gain associated with SELU activation function.

Parameters

tensor (*Tensor*) – the tensor to be initialized.

Returns

the initialized tensor.

Return type

Tensor

static initialise_(*tensor*: *torch.Tensor*) → *torch.Tensor*

In-place initialisation of tensor using xavier uniform, gain associated with SELU activation function.

Parameters

tensor (*Tensor*) – the tensor to be initialized.

Returns

the initialized tensor.

Return type

Tensor

class *analogvnn.nn.activation.ELU.ELU*(*alpha*: *float* = 1.0507)

Bases: *SELU*

Implements the exponential linear unit (ELU) activation function.

Variables

- **alpha** (*nn.Parameter*) – 1.0507
- **scale_factor** (*nn.Parameter*) – 1.

`analogvnn.nn.activation.Gaussian`

Module Contents

Classes

<i>Gaussian</i>	Implements the Gaussian activation function.
<i>GeLU</i>	Implements the Gaussian error linear unit (GeLU) activation function.

class `analogvnn.nn.activation.Gaussian.Gaussian`

Bases: `analogvnn.nn.activation.Activation.Activation`

Implements the Gaussian activation function.

static forward(*x*: *torch.Tensor*) → *torch.Tensor*

Forward pass of the Gaussian activation function.

Parameters

x (*Tensor*) – the input tensor.

Returns

the output tensor.

Return type

Tensor

backward(*grad_output*: *Optional[torch.Tensor]*) → *Optional[torch.Tensor]*

Backward pass of the Gaussian activation function.

Parameters

grad_output (*Optional[Tensor]*) – the gradient of the output tensor.

Returns

the gradient of the input tensor.

Return type

Optional[Tensor]

class `analogvnn.nn.activation.Gaussian.GeLU`

Bases: `analogvnn.nn.activation.Activation.Activation`

Implements the Gaussian error linear unit (GeLU) activation function.

static forward(*x*: *torch.Tensor*) → *torch.Tensor*

Forward pass of the Gaussian error linear unit (GeLU) activation function.

Parameters

x (*Tensor*) – the input tensor.

Returns

the output tensor.

Return type

Tensor

backward(*grad_output*: *Optional[torch.Tensor]*) → *Optional[torch.Tensor]*

Backward pass of the Gaussian error linear unit (GeLU) activation function.

Parameters**grad_output** (*Optional[Tensor]*) – the gradient of the output tensor.**Returns**

the gradient of the input tensor.

Return type*Optional[Tensor]***analogvnn.nn.activation.Identity****Module Contents****Classes***Identity*

Implements the identity activation function.

class analogvnn.nn.activation.Identity.**Identity**(*name=None*)Bases: *analogvnn.nn.activation.Activation.Activation*

Implements the identity activation function.

Variables**name** (*str*) – the name of the activation function.**name**: *Optional[str]***extra_repr**() → *str*Extra `__repr__` of the identity activation function.**Returns**

the extra representation of the identity activation function.

Return type*str***static forward**(*x*: *torch.Tensor*) → *torch.Tensor*

Forward pass of the identity activation function.

Parameters**x** (*Tensor*) – the input tensor.**Returns**

the output tensor same as the input tensor.

Return type

Tensor

backward(*grad_output*: *Optional[torch.Tensor]*) → *Optional[torch.Tensor]*

Backward pass of the identity activation function.

Parameters

grad_output (*Optional[Tensor]*) – the gradient of the output tensor.

Returns

the gradient of the input tensor same as the gradient of the output tensor.

Return type

Optional[Tensor]

`analogvnn.nn.activation.ReLU`

Module Contents**Classes**

<i>PReLU</i>	Implements the parametric rectified linear unit (PReLU) activation function.
<i>ReLU</i>	Implements the rectified linear unit (ReLU) activation function.
<i>LeakyReLU</i>	Implements the leaky rectified linear unit (LeakyReLU) activation function.

class `analogvnn.nn.activation.ReLU.PReLU(alpha: float)`

Bases: `analogvnn.nn.activation.Activation.Activation`

Implements the parametric rectified linear unit (PReLU) activation function.

Variables

- **alpha** (*float*) – the slope of the negative part of the activation function.
- **_zero** (*Tensor*) – placeholder tensor of zero.

`__constants__` = ['alpha', '_zero']

alpha: `torch.nn.Parameter`

_zero: `torch.nn.Parameter`

forward(*x: torch.Tensor*) → `torch.Tensor`

Forward pass of the parametric rectified linear unit (PReLU) activation function.

Parameters

x (*Tensor*) – the input tensor.

Returns

the output tensor.

Return type

Tensor

backward(*grad_output: Optional[torch.Tensor]*) → `Optional[torch.Tensor]`

Backward pass of the parametric rectified linear unit (PReLU) activation function.

Parameters

grad_output (*Optional[Tensor]*) – the gradient of the output tensor.

Returns

the gradient of the input tensor.

Return type

Optional[Tensor]

static initialise(*tensor*: *torch.Tensor*) → *torch.Tensor*

Initialisation of tensor using kaiming uniform, gain associated with PReLU activation function.

Parameters

tensor (*Tensor*) – the tensor to be initialized.

Returns

the initialized tensor.

Return type

Tensor

static initialise_(*tensor*: *torch.Tensor*) → *torch.Tensor*

In-place initialisation of tensor using kaiming uniform, gain associated with PReLU activation function.

Parameters

tensor (*Tensor*) – the tensor to be initialized.

Returns

the initialized tensor.

Return type

Tensor

class analogvnn.nn.activation.ReLU.**ReLU**

Bases: *PReLU*

Implements the rectified linear unit (ReLU) activation function.

Variables

alpha (*float*) – 0

static initialise(*tensor*: *torch.Tensor*) → *torch.Tensor*

Initialisation of tensor using kaiming uniform, gain associated with ReLU activation function.

Parameters

tensor (*Tensor*) – the tensor to be initialized.

Returns

the initialized tensor.

Return type

Tensor

static initialise_(*tensor*: *torch.Tensor*) → *torch.Tensor*

In-place initialisation of tensor using kaiming uniform, gain associated with ReLU activation function.

Parameters

tensor (*Tensor*) – the tensor to be initialized.

Returns

the initialized tensor.

Return type

Tensor

class `analogvnn.nn.activation.ReLU.LeakyReLU`

Bases: *PReLU*

Implements the leaky rectified linear unit (LeakyReLU) activation function.

Variables

alpha (*float*) – 0.01

`analogvnn.nn.activation.SiLU`

Module Contents

Classes

<i>SiLU</i>	Implements the SiLU activation function.
-------------	--

class `analogvnn.nn.activation.SiLU.SiLU`

Bases: *analogvnn.nn.activation.Activation.Activation*

Implements the SiLU activation function.

static forward(*x*: *torch.Tensor*) → *torch.Tensor*

Forward pass of the SiLU.

Parameters

x (*Tensor*) – the input tensor.

Returns

the output tensor.

Return type

Tensor

backward(*grad_output*: *Optional[torch.Tensor]*) → *Optional[torch.Tensor]*

Backward pass of the SiLU.

Parameters

grad_output (*Optional[Tensor]*) – the gradient of the output tensor.

Returns

the gradient of the input tensor.

Return type

Optional[Tensor]

`analogvnn.nn.activation.Sigmoid`

Module Contents

Classes

<i>Logistic</i>	Implements the logistic activation function.
<i>Sigmoid</i>	Implements the sigmoid activation function.

class analogvnn.nn.activation.Sigmoid.**Logistic**

Bases: *analogvnn.nn.activation.Activation.Activation*

Implements the logistic activation function.

static forward(*x*: *torch.Tensor*) → *torch.Tensor*

Forward pass of the logistic activation function.

Parameters

x (*Tensor*) – the input tensor.

Returns

the output tensor.

Return type

Tensor

backward(*grad_output*: *Optional[torch.Tensor]*) → *Optional[torch.Tensor]*

Backward pass of the logistic activation function.

Parameters

grad_output (*Optional[Tensor]*) – the gradient of the output tensor.

Returns

the gradient of the input tensor.

Return type

Optional[Tensor]

static initialise(*tensor*: *torch.Tensor*) → *torch.Tensor*

Initialisation of tensor using xavier uniform, gain associated with logistic activation function.

Parameters

tensor (*Tensor*) – the tensor to be initialized.

Returns

the initialized tensor.

Return type

Tensor

static initialise_(*tensor*: *torch.Tensor*) → *torch.Tensor*

In-place initialisation of tensor using xavier uniform, gain associated with logistic activation function.

Parameters

tensor (*Tensor*) – the tensor to be initialized.

Returns

the initialized tensor.

Return type

Tensor

class analogvnn.nn.activation.Sigmoid.**Sigmoid**

Bases: *Logistic*

Implements the sigmoid activation function.

`analogvnn.nn.activation.Tanh`

Module Contents

Classes

<i>Tanh</i>	Implements the tanh activation function.
-------------	--

class `analogvnn.nn.activation.Tanh.Tanh`

Bases: `analogvnn.nn.activation.Activation.Activation`

Implements the tanh activation function.

static forward(*x*: `torch.Tensor`) → `torch.Tensor`

Forward pass of the tanh activation function.

Parameters

x (*Tensor*) – the input tensor.

Returns

the output tensor.

Return type

`Tensor`

backward(*grad_output*: `Optional[torch.Tensor]`) → `Optional[torch.Tensor]`

Backward pass of the tanh activation function.

Parameters

grad_output (*Optional[Tensor]*) – the gradient of the output tensor.

Returns

the gradient of the input tensor.

Return type

`Optional[Tensor]`

static initialise(*tensor*: `torch.Tensor`) → `torch.Tensor`

Initialisation of tensor using xavier uniform, gain associated with tanh.

Parameters

tensor (*Tensor*) – the tensor to be initialized.

Returns

the initialized tensor.

Return type

`Tensor`

static initialise_(*tensor*: `torch.Tensor`) → `torch.Tensor`

In-place initialisation of tensor using xavier uniform, gain associated with tanh.

Parameters

tensor (*Tensor*) – the tensor to be initialized.

Returns

the initialized tensor.

Return type

`Tensor`

`analogvnn.nn.module`

Submodules

`analogvnn.nn.module.FullSequential`

Module Contents

Classes

<i>FullSequential</i>	A sequential model where backward graph is the reverse of forward graph.
-----------------------	--

class `analogvnn.nn.module.FullSequential.FullSequential`(*tensorboard_log_dir=None*,
device=is_cpu_cuda.device)

Bases: *analogvnn.nn.module.Sequential.Sequential*

A sequential model where backward graph is the reverse of forward graph.

compile(*device: Optional[torch.device] = None*, *layer_data: bool = True*)

Compile the model and add forward and backward graph.

Parameters

- **device** (*torch.device*) – The device to run the model on.
- **layer_data** (*bool*) – True if the data of the layers should be compiled.

Returns

self

Return type

FullSequential

`analogvnn.nn.module.Layer`

Module Contents

Classes

<i>Layer</i>	Base class for analog neural network modules.
--------------	---

class `analogvnn.nn.module.Layer.Layer`

Bases: *torch.nn.Module*

Base class for analog neural network modules.

Variables

- **_inputs** (*Union[None, ArgsKwargs]*) – Inputs of the layer.
- **_outputs** (*Union[None, Tensor, Sequence[Tensor]]*) – Outputs of the layer.
- **_backward_module** (*Optional[BackwardModule]*) – Backward module of the layer.

- **`_use_autograd_graph`** (*bool*) – If True, the autograd graph is used to calculate the gradients.
- **`call_super_init`** (*bool*) – If True, the super class `__init__` of `nn.Module` is called
- **`https`** – [//github.com/pytorch/pytorch/pull/91819](https://github.com/pytorch/pytorch/pull/91819)

property `use_autograd_graph`: *bool*

If True, the autograd graph is used to calculate the gradients.

Returns

`use_autograd_graph`.

Return type

bool

property `inputs`: `analogvnn.graph.ArgsKwargs.ArgsKwargsOutput`

Inputs of the layer.

Returns

`inputs`.

Return type

`ArgsKwargsOutput`

property `outputs`: `Union[None, torch.Tensor, Sequence[torch.Tensor]]`

Outputs of the layer.

Returns

`outputs`.

Return type

`Union[None, Tensor, Sequence[Tensor]]`

property `backward_function`: `Union[None, Callable, analogvnn.backward.BackwardModule.BackwardModule]`

Backward module of the layer.

Returns

`backward_function`.

Return type

`Union[None, Callable, BackwardModule]`

`_inputs`: `Union[None, analogvnn.graph.ArgsKwargs.ArgsKwargs]`

`_outputs`: `Union[None, torch.Tensor, Sequence[torch.Tensor]]`

`_backward_module`: `Optional[analogvnn.backward.BackwardModule.BackwardModule]`

`_use_autograd_graph`: *bool*

`call_super_init`: *bool* = True

`__call__` (**inputs*, ***kwargs*)

Calls the forward pass of neural network layer.

Parameters

- **`*inputs`** – Inputs of the forward pass.
- **`**kwargs`** – Keyword arguments of the forward pass.

set_backward_function(*backward_class*: Union[Callable, analogvnn.backward.BackwardModule.BackwardModule, Type[analogvnn.backward.BackwardModule.BackwardModule]]) → Layer

Sets the backward_function attribute.

Parameters

backward_class (Union[Callable, BackwardModule, Type[BackwardModule]])
– backward_function.

Returns

self.

Return type

Layer

Raises

TypeError – If backward_class is not a callable or BackwardModule.

named_registered_children(*memo*: Optional[Set[torch.nn.Module]] = None) → Iterator[Tuple[str, torch.nn.Module]]

Returns an iterator over immediate registered children modules.

Parameters

memo – a memo to store the set of modules already added to the result

Yields

(str, Module) – Tuple containing a name and child module

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

registered_children() → Iterator[torch.nn.Module]

Returns an iterator over immediate registered children modules.

Yields

nn.Module – a module in the network

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

_forward_wrapper(*function*: Callable) → Callable

Wrapper for the forward function.

Parameters

function (Callable) – Forward function.

Returns

Wrapped function.

Return type

Callable

_call_impl_forward(*args: torch.Tensor, **kwargs: torch.Tensor) → analogvnn.utils.common_types.TENSORS

Calls the forward pass of the layer.

Parameters

- ***args** – Inputs of the forward pass.

- ****kwargs** – Keyword arguments of the forward pass.

Returns

Outputs of the forward pass.

Return type

TENSORS

`analogvnn.nn.module.Model`

Module Contents

Classes

<i>Model</i>	Base class for analog neural network models.
--------------	--

class `analogvnn.nn.module.Model.Model`(*tensorboard_log_dir=None, device=is_cpu_cuda.device*)

Bases: `analogvnn.nn.module.Layer.Layer`, `analogvnn.backward.BackwardModule.BackwardModule`

Base class for analog neural network models.

Variables

- **_compiled** (*bool*) – True if the model is compiled.
- **tensorboard** (`TensorboardModelLog`) – The tensorboard logger of the model.
- **graphs** (`ModelGraph`) – The graph of the model.
- **forward_graph** (`ForwardGraph`) – The forward graph of the model.
- **backward_graph** (`BackwardGraph`) – The backward graph of the model.
- **optimizer** (*optim.Optimizer*) – The optimizer of the model.
- **loss_function** (*Optional[TENSOR_CALLABLE]*) – The loss function of the model.
- **accuracy_function** (*Optional[TENSOR_CALLABLE]*) – The accuracy function of the model.
- **device** (*torch.device*) – The device of the model.

property use_autograd_graph

Is the autograd graph used for the model.

Returns

If True, the autograd graph is used to calculate the gradients.

Return type

bool

__constants__ = ['device']

_compiled: *bool*

tensorboard: *Optional[analogvnn.utils.TensorboardModelLog.TensorboardModelLog]*

graphs: *analogvnn.graph.ModelGraph.ModelGraph*

forward_graph: `analogvnn.graph.ForwardGraph.ForwardGraph`

backward_graph: `analogvnn.graph.BackwardGraph.BackwardGraph`

optimizer: `Optional[torch.optim.Optimizer]`

loss_function: `Optional[analogvnn.utils.common_types.TENSOR_CALLABLE]`

accuracy_function: `Optional[analogvnn.utils.common_types.TENSOR_CALLABLE]`

device: `torch.device`

__call__(**args*, ***kwargs*)

Call the model.

Parameters

- ***args** – The arguments of the model.
- ****kwargs** – The keyword arguments of the model.

Returns

The output of the model.

Return type

TENSORS

Raises

`RuntimeError` – if the model is not compiled.

named_registered_children(*memo: Optional[Set[torch.nn.Module]] = None*) → `Iterator[Tuple[str, torch.nn.Module]]`

Returns an iterator over registered modules under self.

Parameters

memo – a memo to store the set of modules already added to the result

Yields

(*str*, *nn.Module*) – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, 1 will be returned only once.

compile(*device: Optional[torch.device] = None*, *layer_data: bool = True*)

Compile the model.

Parameters

- **device** (`torch.device`) – The device to run the model on.
- **layer_data** (`bool`) – If True, the layer data is logged.

Returns

The compiled model.

Return type

`Model`

forward(**inputs: torch.Tensor*) → `analogvnn.utils.common_types.TENSORS`

Forward pass of the model.

Parameters

***inputs** (`Tensor`) – The inputs of the model.

Returns

The output of the model.

Return type

TENSORS

backward(*inputs: *torch.Tensor*) → analogvnn.utils.common_types.TENSORS

Backward pass of the model.

Parameters

***inputs** (*Tensor*) – The inputs of the model.

Returns

The output of the model.

Return type

TENSORS

loss(output: *torch.Tensor*, target: *torch.Tensor*) → Tuple[*torch.Tensor*, *torch.Tensor*]

Calculate the loss of the model.

Parameters

- **output** (*Tensor*) – The output of the model.
- **target** (*Tensor*) – The target of the model.

Returns

The loss and the accuracy of the model.

Return type

Tuple[*Tensor*, *Tensor*]

Raises

ValueError – if loss_function is None.

train_on(train_loader: *torch.utils.data.DataLoader*, epoch: *int* = None, *args, **kwargs) → Tuple[*float*, *float*]

Train the model on the train_loader.

Parameters

- **train_loader** (*DataLoader*) – The train loader of the model.
- **epoch** (*int*) – The epoch of the model.
- ***args** – The arguments of the train function.
- ****kwargs** – The keyword arguments of the train function.

Returns

The loss and the accuracy of the model.

Return type

Tuple[*float*, *float*]

Raises

RuntimeError – if model is not compiled.

test_on(test_loader: *torch.utils.data.DataLoader*, epoch: *int* = None, *args, **kwargs) → Tuple[*float*, *float*]

Test the model on the test_loader.

Parameters

- **test_loader** (*DataLoader*) – The test loader of the model.

- **epoch** (*int*) – The epoch of the model.
- ***args** – The arguments of the test function.
- ****kwargs** – The keyword arguments of the test function.

Returns

The loss and the accuracy of the model.

Return type

Tuple[float, float]

Raises

RuntimeError – if model is not compiled.

fit(*train_loader*: *torch.utils.data.DataLoader*, *test_loader*: *torch.utils.data.DataLoader*, *epoch*: *int* = None)
→ Tuple[float, float, float, float]

Fit the model on the train_loader and test the model on the test_loader.

Parameters

- **train_loader** (*DataLoader*) – The train loader of the model.
- **test_loader** (*DataLoader*) – The test loader of the model.
- **epoch** (*int*) – The epoch of the model.

Returns

The train loss, the train accuracy, the test loss and the test accuracy of the model.

Return type

Tuple[float, float, float, float]

create_tensorboard(*log_dir*: *str*) → *analogvnn.utils.TensorboardModelLog.TensorboardModelLog*

Create a tensorboard.

Parameters

log_dir (*str*) – The log directory of the tensorboard.

Raises

ImportError – if tensorboard (<https://www.tensorflow.org/>) is not installed.

subscribe_tensorboard(*tensorboard*: *analogvnn.utils.TensorboardModelLog.TensorboardModelLog*)

Subscribe the model to the tensorboard.

Parameters

tensorboard (*TensorboardModelLog*) – The tensorboard of the model.

Returns

self.

Return type

Model

`analogvnn.nn.module.Sequential`

Module Contents

Classes

<i>Sequential</i>	Base class for all sequential models.
class <code>analogvnn.nn.module.Sequential.Sequential</code> (<i>tensorboard_log_dir=None</i> , <i>device=is_cpu_cuda.device</i>)	
Bases: <i>analogvnn.nn.module.Model.Model</i> , <i>torch.nn.Sequential</i>	
Base class for all sequential models.	
__call__ (*args, **kwargs)	
Call the model.	
Parameters	
• *args – The input. • **kwargs – The input.	
Returns	
The output of the model.	
Return type	
<i>torch.Tensor</i>	
compile (<i>device: Optional[torch.device] = None</i> , <i>layer_data: bool = True</i>)	
Compile the model and add forward graph.	
Parameters	
• device (<i>torch.device</i>) – The device to run the model on. • layer_data (<i>bool</i>) – True if the data of the layers should be compiled.	
Returns	
self	
Return type	
<i>Sequential</i>	
add_sequence (*args)	
Add a sequence of modules to the forward graph of model.	
Parameters	
*args (<i>nn.Module</i>) – The modules to add.	

`analogvnn.nn.noise`

Submodules

`analogvnn.nn.noise.GaussianNoise`

Module Contents

Classes

[*GaussianNoise*](#)

Implements the Gaussian noise function.

```
class analogvnn.nn.noise.GaussianNoise.GaussianNoise(std: Optional[float] = None, leakage: Optional[float] = None, precision: Optional[int] = None)
```

Bases: [*analogvnn.nn.noise.Noise.Noise*](#), [*analogvnn.backward.BackwardIdentity.BackwardIdentity*](#)

Implements the Gaussian noise function.

Variables

- **std** (*nn.Parameter*) – the standard deviation of the Gaussian noise.
- **leakage** (*nn.Parameter*) – the leakage of the Gaussian noise.
- **precision** (*nn.Parameter*) – the precision of the Gaussian noise.

property stddev: [`torch.Tensor`](#)

The standard deviation of the Gaussian noise.

Returns

the standard deviation of the Gaussian noise.

Return type

Tensor

property variance: [`torch.Tensor`](#)

The variance of the Gaussian noise.

Returns

the variance of the Gaussian noise.

Return type

Tensor

```
__constants__ = ['std', 'leakage', 'precision']
```

```
std: torch.nn.Parameter
```

```
leakage: torch.nn.Parameter
```

```
precision: torch.nn.Parameter
```

```
static calc_std(leakage: analogvnn.utils.common_types.TENSOR_OPERABLE, precision:  
                analogvnn.utils.common_types.TENSOR_OPERABLE) →  
                analogvnn.utils.common_types.TENSOR_OPERABLE
```

Calculate the standard deviation of the Gaussian noise.

Parameters

- **leakage** (*float*) – the leakage of the Gaussian noise.
- **precision** (*int*) – the precision of the Gaussian noise.

Returns

the standard deviation of the Gaussian noise.

Return type

float

```
static calc_precision(std: analogvnn.utils.common_types.TENSOR_OPERABLE, leakage:  
                      analogvnn.utils.common_types.TENSOR_OPERABLE) →  
                      analogvnn.utils.common_types.TENSOR_OPERABLE
```

Calculate the precision of the Gaussian noise.

Parameters

- **std** (*float*) – the standard deviation of the Gaussian noise.
- **leakage** (*float*) – the leakage of the Gaussian noise.

Returns

the precision of the Gaussian noise.

Return type

int

```
static calc_leakage(std: analogvnn.utils.common_types.TENSOR_OPERABLE, precision:  
                    analogvnn.utils.common_types.TENSOR_OPERABLE) →  
                    analogvnn.utils.common_types.TENSOR_OPERABLE
```

Calculate the leakage of the Gaussian noise.

Parameters

- **std** (*float*) – the standard deviation of the Gaussian noise.
- **precision** (*int*) – the precision of the Gaussian noise.

Returns

the leakage of the Gaussian noise.

Return type

float

```
pdf(x: torch.Tensor, mean: torch.Tensor = 0) → torch.Tensor
```

Calculate the probability density function of the Gaussian noise.

Parameters

- **x** (*Tensor*) – the input tensor.
- **mean** (*Tensor*) – the mean of the Gaussian noise.

Returns

the probability density function of the Gaussian noise.

Return type

Tensor

log_prob(*x*: *torch.Tensor*, *mean*: *torch.Tensor* = 0) → *torch.Tensor*

Calculate the log probability density function of the Gaussian noise.

Parameters

- **x** (*Tensor*) – the input tensor.
- **mean** (*Tensor*) – the mean of the Gaussian noise.

Returns

the log probability density function of the Gaussian noise.

Return type

Tensor

static static_cdf(*x*: *analogvnn.utils.common_types.TENSOR_OPERABLE*, *std*: *analogvnn.utils.common_types.TENSOR_OPERABLE*, *mean*: *analogvnn.utils.common_types.TENSOR_OPERABLE* = 0.0) → *analogvnn.utils.common_types.TENSOR_OPERABLE*

Calculate the cumulative distribution function of the Gaussian noise.

Parameters

- **x** (*TENSOR_OPERABLE*) – the input tensor.
- **std** (*TENSOR_OPERABLE*) – the standard deviation of the Gaussian noise.
- **mean** (*TENSOR_OPERABLE*) – the mean of the Gaussian noise.

Returns

the cumulative distribution function of the Gaussian noise.

Return type

TENSOR_OPERABLE

cdf(*x*: *torch.Tensor*, *mean*: *torch.Tensor* = 0) → *torch.Tensor*

Calculate the cumulative distribution function of the Gaussian noise.

Parameters

- **x** (*Tensor*) – the input tensor.
- **mean** (*Tensor*) – the mean of the Gaussian noise.

Returns

the cumulative distribution function of the Gaussian noise.

Return type

Tensor

forward(*x*: *torch.Tensor*) → *torch.Tensor*

Add the Gaussian noise to the input tensor.

Parameters**x** (*Tensor*) – the input tensor.**Returns**

the output tensor.

Return type

Tensor

extra_repr() → `str`

The extra representation of the Gaussian noise.

Returns

the extra representation of the Gaussian noise.

Return type

`str`

`analogvnn.nn.noise.LaplacianNoise`

Module Contents

Classes

`LaplacianNoise`

Implements the Laplacian noise function.

class `analogvnn.nn.noise.LaplacianNoise.LaplacianNoise`(*scale: Optional[float] = None, leakage: Optional[float] = None, precision: Optional[int] = None*)

Bases: `analogvnn.nn.noise.Noise.Noise`, `analogvnn.backward.BackwardIdentity.BackwardIdentity`

Implements the Laplacian noise function.

Variables

- **scale** (`nn.Parameter`) – the scale of the Laplacian noise.
- **leakage** (`nn.Parameter`) – the leakage of the Laplacian noise.
- **precision** (`nn.Parameter`) – the precision of the Laplacian noise.

property stddev: `torch.Tensor`

The standard deviation of the Laplacian noise.

Returns

the standard deviation of the Laplacian noise.

Return type

`Tensor`

property variance: `torch.Tensor`

The variance of the Laplacian noise.

Returns

the variance of the Laplacian noise.

Return type

`Tensor`

`__constants__` = ['scale', 'leakage', 'precision']

scale: `torch.nn.Parameter`

leakage: `torch.nn.Parameter`

precision: `torch.nn.Parameter`

static calc_scale(*leakage*: `analogvnn.utils.common_types.TENSOR_OPERABLE`, *precision*: `analogvnn.utils.common_types.TENSOR_OPERABLE`) → `analogvnn.utils.common_types.TENSOR_OPERABLE`

Calculate the scale of the Laplacian noise.

Parameters

- **leakage** (*float*) – the leakage of the Laplacian noise.
- **precision** (*int*) – the precision of the Laplacian noise.

Returns

the scale of the Laplacian noise.

Return type

float

static calc_precision(*scale*: `analogvnn.utils.common_types.TENSOR_OPERABLE`, *leakage*: `analogvnn.utils.common_types.TENSOR_OPERABLE`) → `analogvnn.utils.common_types.TENSOR_OPERABLE`

Calculate the precision of the Laplacian noise.

Parameters

- **scale** (*float*) – the scale of the Laplacian noise.
- **leakage** (*float*) – the leakage of the Laplacian noise.

Returns

the precision of the Laplacian noise.

Return type

int

static calc_leakage(*scale*: `analogvnn.utils.common_types.TENSOR_OPERABLE`, *precision*: `analogvnn.utils.common_types.TENSOR_OPERABLE`) → `torch.Tensor`

Calculate the leakage of the Laplacian noise.

Parameters

- **scale** (*float*) – the scale of the Laplacian noise.
- **precision** (*int*) – the precision of the Laplacian noise.

Returns

the leakage of the Laplacian noise.

Return type

float

pdf(*x*: `analogvnn.utils.common_types.TENSOR_OPERABLE`, *loc*: `analogvnn.utils.common_types.TENSOR_OPERABLE = 0`) → `torch.Tensor`

The probability density function of the Laplacian noise.

Parameters

- **x** (`TENSOR_OPERABLE`) – the input tensor.
- **loc** (`TENSOR_OPERABLE`) – the mean of the Laplacian noise.

Returns

the probability density function of the Laplacian noise.

Return type

Tensor

log_prob(*x*: *analogvnn.utils.common_types.TENSOR_OPERABLE*, *loc*:
analogvnn.utils.common_types.TENSOR_OPERABLE = 0) → *torch.Tensor*

The log probability density function of the Laplacian noise.

Parameters

- **x** (*TENSOR_OPERABLE*) – the input tensor.
- **loc** (*TENSOR_OPERABLE*) – the mean of the Laplacian noise.

Returns

the log probability density function of the Laplacian noise.

Return type

Tensor

static static_cdf(*x*: *analogvnn.utils.common_types.TENSOR_OPERABLE*, *scale*:
analogvnn.utils.common_types.TENSOR_OPERABLE, *loc*:
analogvnn.utils.common_types.TENSOR_OPERABLE = 0.0) →
analogvnn.utils.common_types.TENSOR_OPERABLE

The cumulative distribution function of the Laplacian noise.

Parameters

- **x** (*TENSOR_OPERABLE*) – the input tensor.
- **scale** (*TENSOR_OPERABLE*) – the scale of the Laplacian noise.
- **loc** (*TENSOR_OPERABLE*) – the mean of the Laplacian noise.

Returns

the cumulative distribution function of the Laplacian noise.

Return type*TENSOR_OPERABLE*

cdf(*x*: *torch.Tensor*, *loc*: *torch.Tensor* = 0) → *torch.Tensor*

The cumulative distribution function of the Laplacian noise.

Parameters

- **x** (*Tensor*) – the input tensor.
- **loc** (*Tensor*) – the mean of the Laplacian noise.

Returns

the cumulative distribution function of the Laplacian noise.

Return type

Tensor

forward(*x*: *torch.Tensor*) → *torch.Tensor*

Add Laplacian noise to the input tensor.

Parameters

x (*Tensor*) – the input tensor.

Returns

the output tensor with Laplacian noise.

Return type

Tensor

extra_repr() → str

The extra representation of the Laplacian noise.

Returns

the extra representation of the Laplacian noise.

Return type

str

`analogvnn.nn.noise.Noise`**Module Contents****Classes***Noise*

This class is base class for all noise functions.

class `analogvnn.nn.noise.Noise.Noise`Bases: `analogvnn.nn.module.Layer.Layer`

This class is base class for all noise functions.

`analogvnn.nn.noise.PoissonNoise`**Module Contents****Classes***PoissonNoise*

Implements the Poisson noise function.

class `analogvnn.nn.noise.PoissonNoise.PoissonNoise`(*scale: Optional[float] = None, max_leakage:**Optional[float] = None, precision: Optional[int] = None)*Bases: `analogvnn.nn.noise.Noise.Noise`, `analogvnn.backward.BackwardIdentity`,
`BackwardIdentity`

Implements the Poisson noise function.

Variables

- **scale** (*nn.Parameter*) – the scale of the Poisson noise function.
- **max_leakage** (*nn.Parameter*) – the maximum leakage of the Poisson noise.
- **precision** (*nn.Parameter*) – the precision of the Poisson noise.

property **leakage:** float

The leakage of the Poisson noise.

Returns

the leakage of the Poisson noise.

Return type

float

property rate_factor: `torch.Tensor`

The rate factor of the Poisson noise.

Returns

the rate factor of the Poisson noise.

Return type

Tensor

`__constants__ = ['scale', 'max_leakage', 'precision']`

`scale:` `torch.nn.Parameter`

`max_leakage:` `torch.nn.Parameter`

`precision:` `torch.nn.Parameter`

static `calc_scale`(*max_leakage: analogvnn.utils.common_types.TENSOR_OPERABLE*, *precision: analogvnn.utils.common_types.TENSOR_OPERABLE*, *max_check=10000*) → `analogvnn.utils.common_types.TENSOR_OPERABLE`

Calculates the scale using the maximum leakage and the precision.

Parameters

- **max_leakage** (*TENSOR_OPERABLE*) – the maximum leakage of the Poisson noise.
- **precision** (*TENSOR_OPERABLE*) – the precision of the Poisson noise.
- **max_check** (*int*) – the maximum value to check for the scale.

Returns

the scale of the Poisson noise function.

Return type

`TENSOR_OPERABLE`

static `calc_precision`(*scale: analogvnn.utils.common_types.TENSOR_OPERABLE*, *max_leakage: analogvnn.utils.common_types.TENSOR_OPERABLE*, *max_check=2**16*) → `analogvnn.utils.common_types.TENSOR_OPERABLE`

Calculates the precision using the scale and the maximum leakage.

Parameters

- **scale** (*TENSOR_OPERABLE*) – the scale of the Poisson noise function.
- **max_leakage** (*TENSOR_OPERABLE*) – the maximum leakage of the Poisson noise.
- **max_check** (*int*) – the maximum value to check for the precision.

Returns

the precision of the Poisson noise.

Return type

`TENSOR_OPERABLE`

```
static calc_max_leakage(scale: analogvnn.utils.common_types.TENSOR_OPERABLE, precision:
                        analogvnn.utils.common_types.TENSOR_OPERABLE) →
                        analogvnn.utils.common_types.TENSOR_OPERABLE
```

Calculates the maximum leakage using the scale and the precision.

Parameters

- **scale** (*TENSOR_OPERABLE*) – the scale of the Poisson noise function.
- **precision** (*TENSOR_OPERABLE*) – the precision of the Poisson noise.

Returns

the maximum leakage of the Poisson noise.

Return type

TENSOR_OPERABLE

```
static static_cdf(x: analogvnn.utils.common_types.TENSOR_OPERABLE, rate:
                  analogvnn.utils.common_types.TENSOR_OPERABLE, scale_factor:
                  analogvnn.utils.common_types.TENSOR_OPERABLE) →
                  analogvnn.utils.common_types.TENSOR_OPERABLE
```

Calculates the cumulative distribution function of the Poisson noise.

Parameters

- **x** (*TENSOR_OPERABLE*) – the input of the Poisson noise.
- **rate** (*TENSOR_OPERABLE*) – the rate of the Poisson noise.
- **scale_factor** (*TENSOR_OPERABLE*) – the scale factor of the Poisson noise.

Returns

the cumulative distribution function of the Poisson noise.

Return type

TENSOR_OPERABLE

```
static staticmethod_leakage(scale: analogvnn.utils.common_types.TENSOR_OPERABLE, precision:
                             analogvnn.utils.common_types.TENSOR_OPERABLE) →
                             analogvnn.utils.common_types.TENSOR_OPERABLE
```

Calculates the leakage of the Poisson noise using the scale and the precision.

Parameters

- **scale** (*TENSOR_OPERABLE*) – the scale of the Poisson noise function.
- **precision** (*TENSOR_OPERABLE*) – the precision of the Poisson noise.

Returns

the leakage of the Poisson noise.

Return type

TENSOR_OPERABLE

```
pdf(x: torch.Tensor, rate: torch.Tensor) → torch.Tensor
```

Calculates the probability density function of the Poisson noise.

Parameters

- **x** (*Tensor*) – the input of the Poisson noise.
- **rate** (*Tensor*) – the rate of the Poisson noise.

Returns

the probability density function of the Poisson noise.

Return type

Tensor

log_prob(*x*: *torch.Tensor*, *rate*: *torch.Tensor*) → *torch.Tensor*

Calculates the log probability of the Poisson noise.

Parameters

- **x** (*Tensor*) – the input of the Poisson noise.
- **rate** (*Tensor*) – the rate of the Poisson noise.

Returns

the log probability of the Poisson noise.

Return type

Tensor

cdf(*x*: *torch.Tensor*, *rate*: *torch.Tensor*) → *torch.Tensor*

Calculates the cumulative distribution function of the Poisson noise.

Parameters

- **x** (*Tensor*) – the input of the Poisson noise.
- **rate** (*Tensor*) – the rate of the Poisson noise.

Returns

the cumulative distribution function of the Poisson noise.

Return type

Tensor

forward(*x*: *torch.Tensor*) → *torch.Tensor*

Adds the Poisson noise to the input.

Parameters

x (*Tensor*) – the input of the Poisson noise.

Returns

the output of the Poisson noise.

Return type

Tensor

extra_repr() → *str*

Returns the extra representation of the Poisson noise.

Returns

the extra representation of the Poisson noise.

Return type

str

`analogvnn.nn.noise.UniformNoise`

Module Contents

Classes

<i>UniformNoise</i>	Implements the uniform noise function.
---------------------	--

class `analogvnn.nn.noise.UniformNoise.UniformNoise`(*low*: *Optional[float] = None*, *high*: *Optional[float] = None*, *leakage*: *Optional[float] = None*, *precision*: *Optional[int] = None*)

Bases: `analogvnn.nn.noise.Noise.Noise`, `analogvnn.backward.BackwardIdentity.BackwardIdentity`

Implements the uniform noise function.

Variables

- **low** (*nn.Parameter*) – the lower bound of the uniform noise.
- **high** (*nn.Parameter*) – the upper bound of the uniform noise.
- **leakage** (*nn.Parameter*) – the leakage of the uniform noise.
- **precision** (*nn.Parameter*) – the precision of the uniform noise.

property mean: `torch.Tensor`

The mean of the uniform noise.

Returns

the mean of the uniform noise.

Return type

Tensor

property stddev: `torch.Tensor`

The standard deviation of the uniform noise.

Returns

the standard deviation of the uniform noise.

Return type

Tensor

property variance: `torch.Tensor`

The variance of the uniform noise.

Returns

the variance of the uniform noise.

Return type

Tensor

__constants__ = ['low', 'high', 'leakage', 'precision']

low: `torch.nn.Parameter`

high: `torch.nn.Parameter`

leakage: `torch.nn.Parameter`

precision: `torch.nn.Parameter`

static calc_high_low(*leakage: analogvnn.utils.common_types.TENSOR_OPERABLE, precision: analogvnn.utils.common_types.TENSOR_OPERABLE*) →
Tuple[analogvnn.utils.common_types.TENSOR_OPERABLE,
analogvnn.utils.common_types.TENSOR_OPERABLE]

Calculate the high and low from leakage and precision.

Parameters

- **leakage** (*TENSOR_OPERABLE*) – the leakage of the uniform noise.
- **precision** (*TENSOR_OPERABLE*) – the precision of the uniform noise.

Returns

the high and low of the uniform noise.

Return type

Tuple[TENSOR_OPERABLE, TENSOR_OPERABLE]

static calc_leakage(*low: analogvnn.utils.common_types.TENSOR_OPERABLE, high: analogvnn.utils.common_types.TENSOR_OPERABLE, precision: analogvnn.utils.common_types.TENSOR_OPERABLE*) →
analogvnn.utils.common_types.TENSOR_OPERABLE

Calculate the leakage from low, high and precision.

Parameters

- **low** (*TENSOR_OPERABLE*) – the lower bound of the uniform noise.
- **high** (*TENSOR_OPERABLE*) – the upper bound of the uniform noise.
- **precision** (*TENSOR_OPERABLE*) – the precision of the uniform noise.

Returns

the leakage of the uniform noise.

Return type

TENSOR_OPERABLE

static calc_precision(*low: analogvnn.utils.common_types.TENSOR_OPERABLE, high: analogvnn.utils.common_types.TENSOR_OPERABLE, leakage: analogvnn.utils.common_types.TENSOR_OPERABLE*) →
analogvnn.utils.common_types.TENSOR_OPERABLE

Calculate the precision from low, high and leakage.

Parameters

- **low** (*TENSOR_OPERABLE*) – the lower bound of the uniform noise.
- **high** (*TENSOR_OPERABLE*) – the upper bound of the uniform noise.
- **leakage** (*TENSOR_OPERABLE*) – the leakage of the uniform noise.

Returns

the precision of the uniform noise.

Return type

TENSOR_OPERABLE

pdf(*x*: *torch.Tensor*) → *torch.Tensor*

The probability density function of the uniform noise.

Parameters

x (*Tensor*) – the input tensor.

Returns

the probability density function of the uniform noise.

Return type

Tensor

log_prob(*x*: *torch.Tensor*) → *torch.Tensor*

The log probability density function of the uniform noise.

Parameters

x (*Tensor*) – the input tensor.

Returns

the log probability density function of the uniform noise.

Return type

Tensor

cdf(*x*: *analogvnn.utils.common_types.TENSOR_OPERABLE*) → *analogvnn.utils.common_types.TENSOR_OPERABLE*

The cumulative distribution function of the uniform noise.

Parameters

x (*TENSOR_OPERABLE*) – the input tensor.

Returns

the cumulative distribution function of the uniform noise.

Return type

TENSOR_OPERABLE

forward(*x*: *torch.Tensor*) → *torch.Tensor*

Add the uniform noise to the input tensor.

Parameters

x (*Tensor*) – the input tensor.

Returns

the output tensor.

Return type

Tensor

extra_repr() → *str*

The extra representation of the uniform noise.

Returns

the extra representation of the uniform noise.

Return type

str

`analogvnn.nn.normalize`

Submodules

`analogvnn.nn.normalize.Clamp`

Module Contents

Classes

<i>Clamp</i>	Implements the clamp normalization function with range [-1, 1].
<i>Clamp01</i>	Implements the clamp normalization function with range [0, 1].

class `analogvnn.nn.normalize.Clamp.Clamp`

Bases: *[analogvnn.nn.normalize.Normalize.Normalize](#)*, *[analogvnn.backward.BackwardIdentity.BackwardIdentity](#)*

Implements the clamp normalization function with range [-1, 1].

static forward(*x*: *torch.Tensor*)

Forward pass of the clamp normalization function with range [-1, 1].

Parameters

x (*Tensor*) – the input tensor.

Returns

the output tensor.

Return type

Tensor

backward(*grad_output*: *Optional[torch.Tensor]*) → *Optional[torch.Tensor]*

Backward pass of the clamp normalization function with range [-1, 1].

Parameters

grad_output (*Optional[Tensor]*) – the gradient of the output tensor.

Returns

the gradient of the input tensor.

Return type

Optional[Tensor]

class `analogvnn.nn.normalize.Clamp.Clamp01`

Bases: *[analogvnn.nn.normalize.Normalize.Normalize](#)*, *[analogvnn.backward.BackwardIdentity.BackwardIdentity](#)*

Implements the clamp normalization function with range [0, 1].

static forward(*x*: *torch.Tensor*)

Forward pass of the clamp normalization function with range [0, 1].

Parameters

x (*Tensor*) – the input tensor.

Returns

the output tensor.

Return type

Tensor

backward(*grad_output*: *Optional[torch.Tensor]*) → *Optional[torch.Tensor]*

Backward pass of the clamp normalization function with range [0, 1].

Parameters

grad_output (*Optional[Tensor]*) – the gradient of the output tensor.

Returns

the gradient of the input tensor.

Return type

Optional[Tensor]

`analogvnn.nn.normalize.LPNorm`

Module Contents**Classes**

<i>LPNorm</i>	Implements the row-wise Lp normalization function.
<i>LPNormW</i>	Implements the whole matrix Lp normalization function.
<i>L1Norm</i>	Implements the row-wise L1 normalization function.
<i>L2Norm</i>	Implements the row-wise L2 normalization function.
<i>L1NormW</i>	Implements the whole matrix L1 normalization function.
<i>L2NormW</i>	Implements the whole matrix L2 normalization function.
<i>L1NormM</i>	Implements the row-wise L1 normalization function with maximum absolute value of 1.
<i>L2NormM</i>	Implements the row-wise L2 normalization function with maximum absolute value of 1.
<i>L1NormWM</i>	Implements the whole matrix L1 normalization function with maximum absolute value of 1.
<i>L2NormWM</i>	Implements the whole matrix L2 normalization function with maximum absolute value of 1.

class `analogvnn.nn.normalize.LPNorm.LPNorm`(*p*: *int*, *make_max_1*=*False*)

Bases: `analogvnn.nn.normalize.Normalize.Normalize`, `analogvnn.backward.BackwardIdentity.BackwardIdentity`

Implements the row-wise Lp normalization function.

Variables

- **p** (*int*) – the pth power of the Lp norm.
- **make_max_1** (*bool*) – if True, the maximum absolute value of the output tensor will be 1.

`__constants__` = ['p', 'make_max_1']

p: torch.nn.Parameter

make_max_1: torch.nn.Parameter

forward(x: torch.Tensor) → torch.Tensor

Forward pass of row-wise Lp normalization function.

Parameters

x (Tensor) – the input tensor.

Returns

the output tensor.

Return type

Tensor

class analogvnn.nn.normalize.LPNorm.LPNormW(p: int, make_max_1=False)

Bases: LPNorm

Implements the whole matrix Lp normalization function.

forward(x: torch.Tensor) → torch.Tensor

Forward pass of whole matrix Lp normalization function.

Parameters

x (Tensor) – the input tensor.

Returns

the output tensor.

Return type

Tensor

class analogvnn.nn.normalize.LPNorm.L1Norm

Bases: LPNorm

Implements the row-wise L1 normalization function.

class analogvnn.nn.normalize.LPNorm.L2Norm

Bases: LPNorm

Implements the row-wise L2 normalization function.

class analogvnn.nn.normalize.LPNorm.L1NormW

Bases: LPNormW

Implements the whole matrix L1 normalization function.

class analogvnn.nn.normalize.LPNorm.L2NormW

Bases: LPNormW

Implements the whole matrix L2 normalization function.

class analogvnn.nn.normalize.LPNorm.L1NormM

Bases: LPNorm

Implements the row-wise L1 normalization function with maximum absolute value of 1.

class analogvnn.nn.normalize.LPNorm.L2NormM

Bases: LPNorm

Implements the row-wise L2 normalization function with maximum absolute value of 1.

class analogvnn.nn.normalize.LPNorm.L1NormWM

Bases: *LPNormW*

Implements the whole matrix L1 normalization function with maximum absolute value of 1.

class analogvnn.nn.normalize.LPNorm.L2NormWM

Bases: *LPNormW*

Implements the whole matrix L2 normalization function with maximum absolute value of 1.

analogvnn.nn.normalize.Normalize

Module Contents

Classes

Normalize

This class is base class for all normalization functions.

class analogvnn.nn.normalize.Normalize.Normalize

Bases: *analogvnn.nn.module.Layer.Layer*

This class is base class for all normalization functions.

analogvnn.nn.precision

Submodules

analogvnn.nn.precision.Precision

Module Contents

Classes

Precision

This class is base class for all precision functions.

class analogvnn.nn.precision.Precision.Precision

Bases: *analogvnn.nn.module.Layer.Layer*

This class is base class for all precision functions.

`analogvnn.nn.precision.ReducePrecision`

Module Contents

Classes

<i>ReducePrecision</i>	Implements the reduce precision function.
------------------------	---

class `analogvnn.nn.precision.ReducePrecision.ReducePrecision`(*precision: int = None, divide: float = 0.5*)

Bases: `analogvnn.nn.precision.Precision.Precision`, `analogvnn.backward.BackwardIdentity.BackwardIdentity`

Implements the reduce precision function.

Variables

- **precision** (`nn.Parameter`) – the precision of the output tensor.
- **divide** (`nn.Parameter`) – the rounding value that is if divide is 0.5, then 0.6 will be rounded to 1.0 and 0.4 will be rounded to 0.0.

property `precision_width: torch.Tensor`

The precision width.

Returns

the precision width

Return type

Tensor

property `bit_precision: torch.Tensor`

The bit precision of the ReducePrecision module.

Returns

the bit precision of the ReducePrecision module.

Return type

Tensor

`__constants__ = ['precision', 'divide']`

precision: `torch.nn.Parameter`

divide: `torch.nn.Parameter`

static `convert_to_precision`(*bit_precision: analogvnn.utils.common_types.TENSOR_OPERABLE*) → `analogvnn.utils.common_types.TENSOR_OPERABLE`

Convert the bit precision to the precision.

Parameters

bit_precision (`TENSOR_OPERABLE`) – the bit precision.

Returns

the precision.

Return type

`TENSOR_OPERABLE`

extra_repr() → str

The extra `__repr__` string of the ReducePrecision module.

Returns

string

Return type

str

forward(x: *torch.Tensor*) → torch.Tensor

Forward function of the ReducePrecision module.

Parameters

x (*Tensor*) – the input tensor.

Returns

the output tensor.

Return type

Tensor

`analogvnn.nn.precision.StochasticReducePrecision`

Module Contents

Classes

StochasticReducePrecision

Implements the stochastic reduce precision function.

class `analogvnn.nn.precision.StochasticReducePrecision.StochasticReducePrecision`(precision: *int* = 8)

Bases: *analogvnn.nn.precision.Precision.Precision*, *analogvnn.backward.BackwardIdentity.BackwardIdentity*

Implements the stochastic reduce precision function.

Variables

precision (*nn.Parameter*) – the precision of the output tensor.

property precision_width: *torch.Tensor*

The precision width.

Returns

the precision width

Return type

Tensor

property bit_precision: *torch.Tensor*

The bit precision of the ReducePrecision module.

Returns

the bit precision of the ReducePrecision module.

Return type

Tensor

```
__constants__ = ['precision']
```

```
precision: torch.nn.Parameter
```

```
static convert_to_precision(bit_precision: analogvnn.utils.common_types.TENSOR_OPERABLE) →  
    analogvnn.utils.common_types.TENSOR_OPERABLE
```

Convert the bit precision to the precision.

Parameters

bit_precision (*TENSOR_OPERABLE*) – the bit precision.

Returns

the precision.

Return type

TENSOR_OPERABLE

```
extra_repr() → str
```

The extra `__repr__` string of the StochasticReducePrecision module.

Returns

string

Return type

str

```
forward(x: torch.Tensor) → torch.Tensor
```

Forward function of the StochasticReducePrecision module.

Parameters

x (*Tensor*) – input tensor.

Returns

output tensor.

Return type

Tensor

Submodules

`analogvnn.nn.Linear`

Module Contents

Classes

<i>LinearBackpropagation</i>	The backpropagation module of a linear layer.
<i>Linear</i>	A linear layer.

```
class analogvnn.nn.Linear.LinearBackpropagation(layer: torch.nn.Module = None)
```

Bases: [*analogvnn.backward.BackwardModule.BackwardModule*](#)

The backpropagation module of a linear layer.

forward(*x*: *torch.Tensor*)

Forward pass of the linear layer.

Parameters

x (*Tensor*) – The input of the linear layer.

Returns

The output of the linear layer.

Return type

Tensor

backward(*grad_output*: *Optional[torch.Tensor]*) → *Optional[torch.Tensor]*

Backward pass of the linear layer.

Parameters

grad_output (*Optional[Tensor]*) – The gradient of the output.

Returns

The gradient of the input.

Return type

Optional[Tensor]

class analogvnn.nn.Linear.**Linear**(*in_features*: *int*, *out_features*: *int*, *bias*: *bool* = *True*)

Bases: *analogvnn.nn.module.Layer.Layer*

A linear layer.

Variables

- **in_features** (*int*) – The number of input features.
- **out_features** (*int*) – The number of output features.
- **weight** (*nn.Parameter*) – The weight of the layer.
- **bias** (*nn.Parameter*) – The bias of the layer.

__constants__ = ['in_features', 'out_features']

in_features: *int*

out_features: *int*

weight: *torch.nn.Parameter*

bias: *Optional[torch.nn.Parameter]*

reset_parameters()

Reset the parameters of the layer.

extra_repr() → *str*

Extra representation of the linear layer.

Returns

The extra representation of the linear layer.

Return type

str

`analogvnn.parameter`

Submodules

`analogvnn.parameter.PseudoParameter`

Module Contents

Classes

PseudoParameter

A parameterized parameter which acts like a normal parameter during gradient updates.

class `analogvnn.parameter.PseudoParameter.PseudoParameter`(*data=None, requires_grad=True, transformation=None*)

Bases: `torch.nn.Module`

A parameterized parameter which acts like a normal parameter during gradient updates.

PyTorch's ParameterizedParameters vs AnalogVNN's PseudoParameters:

- **Similarity (Forward or Parameterizing the data):**
 - > Data -> ParameterizingModel -> Parameterized Data
- **Difference (Backward or Gradient Calculations):** - ParameterizedParameters
 - > Parameterized Data -> ParameterizingModel -> Data
 - PseudoParameters > Parameterized Data -> Data

Variables

- **`_transformation`** (*Callable*) – the transformation.
- **`_transformed`** (*nn.Parameter*) – the transformed parameter.

Properties:

`grad` (Tensor): the gradient of the parameter. `module` (PseudoParameterModule): the module that wraps the parameter and the transformation. `transformation` (Callable): the transformation.

property transformation

Returns the transformation.

Returns

the transformation.

Return type

Callable

`_transformation:` Callable

`_transformed:` `torch.nn.Parameter`

forward

Alias for `__call__`

_call_impl

Alias for `__call__`

right_inverse

Alias for `set_original_data`.

static identity(*x: Any*) → *Any*

The identity function.

Parameters

x (*Any*) – the input tensor.

Returns

the input tensor.

Return type

Any

__call__(*args, **kwargs)

Transforms the parameter.

Parameters

- ***args** – additional arguments.
- ****kwargs** – additional keyword arguments.

Returns

the transformed parameter.

Return type

`nn.Parameter`

Raises

RuntimeError – if the transformation callable fails.

set_original_data(*data: torch.Tensor*) → *PseudoParameter*

Set data to the original parameter.

Parameters

data (*Tensor*) – the data to set.

Returns

`self`.

Return type

PseudoParameter

__repr__()

Returns a string representation of the parameter.

Returns

the string representation.

Return type

`str`

set_transformation(*transformation*) → *PseudoParameter*

Sets the transformation.

Parameters

transformation (*Callable*) – the transformation.

Returns

self.

Return type

PseudoParameter

static substitute_member(*tensor_from: Any, tensor_to: Any, property_name: str, setter: bool = True*)

Substitutes a member of a tensor as property of another tensor.

Parameters

- **tensor_from** (*Any*) – the tensor property to substitute.
- **tensor_to** (*Any*) – the tensor property to substitute to.
- **property_name** (*str*) – the name of the property.
- **setter** (*bool*) – whether to substitute the setter.

classmethod parameterize(*module: torch.nn.Module, param_name: str, transformation: Callable*) → *PseudoParameter*

Parameterizes a parameter.

Parameters

- **module** (*nn.Module*) – the module.
- **param_name** (*str*) – the name of the parameter.
- **transformation** (*Callable*) – the transformation to apply.

Returns

the parameterized parameter.

Return type

PseudoParameter

classmethod parametrize_module(*module: torch.nn.Module, transformation: Callable, requires_grad: bool = True*)

Parametrize all parameters of a module.

Parameters

- **module** (*nn.Module*) – the module parameters to parametrize.
- **transformation** (*Callable*) – the transformation.
- **requires_grad** (*bool*) – if True, only parametrized parameters that require gradients.

analogvnn.utils

Submodules

analogvnn.utils.TensorboardModelLog

Module Contents**Classes**

TensorboardModelLog

Tensorboard model log.

```
class analogvnn.utils.TensorboardModelLog.TensorboardModelLog(model:
    analogvnn.nn.module.Model.Model,
    log_dir: str)
```

Tensorboard model log.

Variables

- **model** (*nn.Module*) – the model to log.
- **tensorboard** (*SummaryWriter*) – the tensorboard.
- **layer_data** (*bool*) – whether to log the layer data.
- **_log_record** (*Dict[str, bool]*) – the log record.

model: *torch.nn.Module*

tensorboard: *Optional[torch.utils.tensorboard.SummaryWriter]*

layer_data: *bool*

_log_record: *Dict[str, bool]*

__exit__

Close the tensorboard.

set_log_dir(*log_dir: str*) → *TensorboardModelLog*

Set the log directory.

Parameters

log_dir (*str*) – the log directory.

Returns

self.

Return type

TensorboardModelLog

Raises

ValueError – if the log directory is invalid.

_add_layer_data(*epoch: int = None*)

Add the layer data to the tensorboard.

Parameters

epoch (*int*) – the epoch to add the data for.

on_compile(*layer_data: bool = True*)

Called when the model is compiled.

Parameters

layer_data (*bool*) – whether to log the layer data.

add_graph(*train_loader: torch.utils.data.DataLoader, model: Optional[torch.nn.Module] = None,*
input_size: Optional[Sequence[int]] = None) → *TensorboardModelLog*

Add the model graph to the tensorboard.

Parameters

- **train_loader** (*DataLoader*) – the train loader.
- **model** (*Optional[nn.Module]*) – the model to log.

- **input_size** (*Optional[Sequence[int]]*) – the input size.

Returns

self.

Return type

TensorboardModelLog

add_summary(*input_size: Optional[Sequence[int]] = None, train_loader: Optional[torch.utils.data.DataLoader] = None, model: Optional[torch.nn.Module] = None, *args, **kwargs*) → *Tuple[str, str]*

Add the model summary to the tensorboard.

Parameters

- **input_size** (*Optional[Sequence[int]]*) – the input size.
- **train_loader** (*Optional[DataLoader]*) – the train loader.
- **model** (*nn.Module*) – the model to log.
- ***args** – the arguments to `torchinfo.summary`.
- ****kwargs** – the keyword arguments to `torchinfo.summary`.

Returns

the model `__repr__` and the model summary.

Return type

Tuple[str, str]

register_training(*epoch: int, train_loss: float, train_accuracy: float*) → *TensorboardModelLog*

Register the training data.

Parameters

- **epoch** (*int*) – the epoch.
- **train_loss** (*float*) – the training loss.
- **train_accuracy** (*float*) – the training accuracy.

Returns

self.

Return type

TensorboardModelLog

register_testing(*epoch: int, test_loss: float, test_accuracy: float*) → *TensorboardModelLog*

Register the testing data.

Parameters

- **epoch** (*int*) – the epoch.
- **test_loss** (*float*) – the test loss.
- **test_accuracy** (*float*) – the test accuracy.

Returns

self.

Return type

TensorboardModelLog

close(*args, **kwargs)

Close the tensorboard.

Parameters

- ***args** – ignored.
- ****kwargs** – ignored.

__enter__()

Enter the TensorboardModelLog context.

Returns

self.

Return type

TensorboardModelLog

analogvnn.utils.common_types

Module Contents

analogvnn.utils.common_types.**TENSORS**

TENSORS is a type alias for a tensor or a sequence of tensors.

analogvnn.utils.common_types.**TENSOR_OPERABLE**

TENSOR_OPERABLE is a type alias for types that can be operated on by a tensor.

analogvnn.utils.common_types.**TENSOR_CALLABLE**

TENSOR_CALLABLE is a type alias for a function that takes a *TENSOR_OPERABLE* and returns a *TENSOR_OPERABLE*.

analogvnn.utils.get_model_summaries

Module Contents

Functions

<i>get_model_summaries</i> (→ Tuple[str, str])	Creates the model summaries.
--	------------------------------

analogvnn.utils.get_model_summaries.**get_model_summaries**(*model: Optional[torch.nn.Module]*,
input_size: Optional[Sequence[int]] = None, *train_loader:*
torch.utils.data.DataLoader = None, *args,
 **kwargs) → Tuple[str, str]

Creates the model summaries.

Parameters

- **train_loader** (*DataLoader*) – the train loader.
- **model** (*nn.Module*) – the model to log.
- **input_size** (*Optional[Sequence[int]]*) – the input size.

- ***args** – the arguments to `torchinfo.summary`.
- ****kwargs** – the keyword arguments to `torchinfo.summary`.

Returns

the model `__repr__` and the model summary.

Return type

`Tuple[str, str]`

Raises

- **ImportError** – if `torchinfo` (<https://github.com/tyleryep/torchinfo>) is not installed.
- **ValueError** – if the `input_size` and `train_loader` are `None`.

`analogvnn.utils.is_cpu_cuda`

Module Contents**Classes**

<code>CPUCuda</code>	<code>CPUCuda</code> is a class that can be used to get, check and set the device.
----------------------	--

Attributes

<code>is_cpu_cuda</code>	The <code>CPUCuda</code> instance.
--------------------------	------------------------------------

class `analogvnn.utils.is_cpu_cuda.CPUCuda`

`CPUCuda` is a class that can be used to get, check and set the device.

Variables

- **_device** (`torch.device`) – The device.
- **device_name** (`str`) – The name of the device.

property device: `torch.device`

Get the device.

Returns

the device.

Return type

`torch.device`

property is_cpu: `bool`

Check if the device is cpu.

Returns

True if the device is cpu, False otherwise.

Return type

`bool`

property is_cuda: `bool`

Check if the device is cuda.

Returns

True if the device is cuda, False otherwise.

Return type

`bool`

property is_using_cuda: `Tuple[torch.device, bool]`

Check if the device is cuda.

Returns

the device and True if the device is cuda, False otherwise.

Return type

`tuple`

_device: `torch.device`

device_name: `str`

use_cpu() → *CPU_cuda*

Use cpu.

Returns

self

Return type

CPU_cuda

use_cuda_if_available() → *CPU_cuda*

Use cuda if available.

Returns

self

Return type

CPU_cuda

set_device(device_name: Union[str, torch.device]) → *CPU_cuda*

Set the device to the given device name.

Parameters

device_name (*Union[str, torch.device]*) – the device name.

Returns

self

Return type

CPU_cuda

get_module_device(module) → `torch.device`

Get the device of the given module.

Parameters

module (*torch.nn.Module*) – the module.

Returns

the device of the module.

Return type

`torch.device`

`analogvnn.utils.is_cpu_cuda.is_cpu_cuda:` *CPUCuda*

The CPUCuda instance.

Type
CPUCuda

`analogvnn.utils.render_autograd_graph`

Module Contents

Classes

<i>AutoGradDot</i>	Stores and manages Graphviz representation of PyTorch autograd graph.
--------------------	---

Functions

<i>size_to_str</i> (size)	Convert a tensor size to a string.
<i>make_autograd_obj_from_outputs</i> (→ AutoGrad-Dot)	Compile Graphviz representation of PyTorch autograd graph from output tensors.
<i>make_autograd_obj_from_module</i> (→ AutoGrad-Dot)	Compile Graphviz representation of PyTorch autograd graph from forward pass.
<i>get_autograd_dot_from_trace</i> (→ graphviz.Digraph)	Produces graphs of torch.jit.trace outputs.
<i>get_autograd_dot_from_outputs</i> (→ graphviz.Digraph)	Runs and make Graphviz representation of PyTorch autograd graph from output tensors.
<i>get_autograd_dot_from_module</i> (→ graphviz.Digraph)	Runs and make Graphviz representation of PyTorch autograd graph from forward pass.
<i>save_autograd_graph_from_outputs</i> (→ str)	Save Graphviz representation of PyTorch autograd graph from output tensors.
<i>save_autograd_graph_from_module</i> (→ str)	Save Graphviz representation of PyTorch autograd graph from forward pass.
<i>save_autograd_graph_from_trace</i> (→ str)	Save Graphviz representation of PyTorch autograd graph from trace.

`analogvnn.utils.render_autograd_graph.size_to_str`(*size*)

Convert a tensor size to a string.

Parameters

size (*torch.Size*) – the size to convert.

Returns

the string representation of the size.

Return type

str

class `analogvnn.utils.render_autograd_graph.AutoGradDot`

Stores and manages Graphviz representation of PyTorch autograd graph.

Variables

- **dot** (*graphviz.Digraph*) – Graphviz representation of the autograd graph.
- **_module** (*nn.Module*) – The module to be traced.
- **_inputs** (*List[[Tensor](#)]*) – The inputs to the module.
- **_inputs_kwargs** (*Dict[[str](#), [Tensor](#)]*) – The keyword arguments to the module.
- **_outputs** (*Sequence[[Tensor](#)]*) – The outputs of the module.
- **param_map** (*Dict[[int](#), [str](#)]*) – A map from parameter values to their names.
- **_seen** (*set*) – A set of nodes that have already been added to the graph.
- **show_attrs** (*bool*) – whether to display non-tensor attributes of backward nodes (Requires PyTorch version ≥ 1.9)
- **show_saved** (*bool*) – whether to display saved tensor nodes that are not by custom autograd functions. Saved tensor nodes for custom functions, if present, are always displayed. (Requires PyTorch version ≥ 1.9)
- **max_attr_chars** (*int*) – if `show_attrs` is *True*, sets max number of characters to display for any given attribute.
- **_called** (*bool*) – the module has been called.

property inputs: [Sequence\[\[torch.Tensor\]\(#\)\]](#)

The arg inputs to the module.

Returns

the arg inputs to the module.

Return type

[Sequence\[\[Tensor\]\(#\)\]](#)

property inputs_kwargs: [Dict\[\[str\]\(#\), \[torch.Tensor\]\(#\)\]](#)

The keyword inputs to the module.

Parameters

- **Dict[[str](#)** – the keyword inputs to the module.
- **Tensor]** – the keyword inputs to the module.

property outputs: [Optional\[Sequence\[\[torch.Tensor\]\(#\)\]\]](#)

The outputs of the module.

Returns

the outputs of the module.

Return type

[Optional\[Sequence\[\[Tensor\]\(#\)\]\]](#)

property module: [torch.nn.Module](#)

The module.

Returns

the module to be traced.

Return type

[nn.Module](#)

property ignore_tensor: Dict[int, bool]

The tensor ignored from the dot graphs.

Returns

the ignore tensor dict.

Return type

Dict[int, bool]

dot: graphviz.Digraph

_module: torch.nn.Module

_inputs: Sequence[torch.Tensor]

_inputs_kwargs: Dict[str, torch.Tensor]

_outputs: Sequence[torch.Tensor]

param_map: dict

_seen: set

show_attrs: bool

show_saved: bool

max_attr_chars: int

_called: bool = False

_ignore_tensor: Dict[int, bool]

__post_init__()

Create the graphviz graph.

Raises

ImportError – if graphviz (<https://pygraphviz.github.io/>) is not available.

reset_params()

Reset the param_map and _seen.

Returns

self.

Return type

AutoGradDot

add_ignore_tensor(tensor: torch.Tensor)

Add a tensor to the ignore tensor dict.

Parameters

tensor (*Tensor*) – the tensor to ignore.

Returns

self.

Return type

AutoGradDot

del_ignore_tensor(*tensor*: *torch.Tensor*)

Delete a tensor from the ignore tensor dict.

Parameters

tensor (*Tensor*) – the tensor to delete.

Returns

self.

Return type

AutoGradDot

get_tensor_name(*tensor*: *torch.Tensor*, *name*: *Optional[str] = None*) → *Tuple[str, str]*

Get the name of the tensor.

Parameters

- **tensor** (*Tensor*) – the tensor.
- **name** (*Optional[str]*) – the name of the tensor. Defaults to None.

Returns

the name and size of the tensor.

Return type

Tuple[str, str]

add_tensor(*tensor*: *torch.Tensor*, *name*: *Optional[str] = None*, *_attributes=None*, ***kwargs*)

Add a tensor to the graph.

Parameters

- **tensor** (*Tensor*) – the tensor.
- **name** (*Optional[str]*) – the name of the tensor. Defaults to None.
- **_attributes** (*Optional[Dict[str, str]]*) – the attributes of the tensor. Defaults to None.
- ****kwargs** – the attributes of the dot.node function.

Returns

self.

Return type

AutoGradDot

add_fn(*fn*: *Any*, *_attributes=None*, ***kwargs*)

Add a function to the graph.

Parameters

- **fn** (*Any*) – the function.
- **_attributes** (*Optional[Dict[str, str]]*) – the attributes of the function. Defaults to None.
- ****kwargs** – the attributes of the dot.node function.

Returns

self.

Return type

AutoGradDot

add_edge(*u: Any, v: Any, label: Optional[str] = None, _attributes=None, **kwargs*)

Add an edge to the graph.

Parameters

- **u** (*Any*) – tail node.
- **v** (*Any*) – head node.
- **label** (*Optional[str]*) – the label of the edge. Defaults to None.
- **_attributes** (*Optional[Dict[str, str]]*) – the attributes of the edge. Defaults to None.
- ****kwargs** – the attributes of the dot.edge function.

Returns

self.

Return type

AutoGradDot

add_seen(*item: Any*)

Add an item to the seen set.

Parameters

item (*Any*) – the item.

Returns

self.

Return type

AutoGradDot

is_seen(*item: Any*) → bool

Check if the item is in the seen set.

Parameters

item (*Any*) – the item.

Returns

True if the item is in the seen set.

Return type

bool

`analogvnn.utils.render_autograd_graph.make_autograd_obj_from_outputs`(*outputs:*

*Union[torch.Tensor,
Sequence[torch.Tensor]],
named_params:
Union[Dict[str, Any],
Iterator[Tuple[str,
torch.nn.Parameter]]],
additional_params:
Optional[dict] = None,
show_attrs: bool = True,
show_saved: bool = True,
max_attr_chars: int = 50)
→ *AutoGradDot**

Compile Graphviz representation of PyTorch autograd graph from output tensors.

Parameters

- **outputs** (*Union[[Tensor](#), [Sequence\[\[Tensor\]\(#\)\]](#)]*) – output tensor(s) of forward pass
- **named_params** (*Union[[Dict\[str, Any\]](#), [Iterator\[Tuple\[str, \[Parameter\]\(#\)\]\]\]](#)*) – dict of params to label nodes with
- **additional_params** (*dict*) – dict of additional params to label nodes with
- **show_attrs** (*bool*) – whether to display non-tensor attributes of backward nodes (Requires PyTorch version ≥ 1.9)
- **show_saved** (*bool*) – whether to display saved tensor nodes that are not by custom autograd functions. Saved tensor nodes for custom functions, if present, are always displayed. (Requires PyTorch version ≥ 1.9)
- **max_attr_chars** (*int*) – if `show_attrs` is *True*, sets max number of characters to display for any given attribute.

Returns

graphviz representation of autograd graph

Return type

[AutoGradDot](#)

```
analogvnn.utils.render_autograd_graph.make_autograd_obj_from_module(module: torch.nn.Module,
                                                                    *args: torch.Tensor,
                                                                    additional_params:
                                                                    Optional[dict] = None,
                                                                    show_attrs: bool = True,
                                                                    show_saved: bool = True,
                                                                    max_attr_chars: int = 50,
                                                                    from_forward: bool =
                                                                    False, **kwargs:
                                                                    torch.Tensor) →
                                                                    AutoGradDot
```

Compile Graphviz representation of PyTorch autograd graph from forward pass.

Parameters

- **module** (*nn.Module*) – PyTorch model
- ***args** (*Tensor*) – input to the model
- **additional_params** (*dict*) – dict of additional params to label nodes with
- **show_attrs** (*bool*) – whether to display non-tensor attributes of backward nodes (Requires PyTorch version ≥ 1.9)
- **show_saved** (*bool*) – whether to display saved tensor nodes that are not by custom autograd functions. Saved tensor nodes for custom functions, if present, are always displayed. (Requires PyTorch version ≥ 1.9)
- **max_attr_chars** (*int*) – if `show_attrs` is *True*, sets max number of characters to display for any given attribute.
- **from_forward** (*bool*) – if *True* then use autograd graph otherwise analogvnn graph
- ****kwargs** (*Tensor*) – input to the model

Returns

graphviz representation of autograd graph

Return type

[AutoGradDot](#)

`analogvnn.utils.render_autograd_graph.get_autograd_dot_from_trace(trace) → graphviz.Digraph`

Produces graphs of `torch.jit.trace` outputs.

Example: `>>> trace, = torch.jit.trace(model, args=(x,)) >>> dot = get_autograd_dot_from_trace(trace)`

Parameters

trace (`torch.jit.trace`) – the trace object to visualize.

Returns

the resulting graph.

Return type

`graphviz.Digraph`

`analogvnn.utils.render_autograd_graph.get_autograd_dot_from_outputs(outputs:`

`Union[torch.Tensor,
Sequence[torch.Tensor]],
named_params:
Union[Dict[str, Any],
Iterator[Tuple[str,
torch.nn.Parameter]]],
additional_params:
Optional[dict] = None,
show_attrs: bool = True,
show_saved: bool = True,
max_attr_chars: int = 50)
→ graphviz.Digraph`

Runs and make Graphviz representation of PyTorch autograd graph from output tensors.

Parameters

- **outputs** (`Union[Tensor, Sequence[Tensor]]`) – output tensor(s) of forward pass
- **named_params** (`Union[Dict[str, Any], Iterator[Tuple[str, Parameter]]]`) – dict of params to label nodes with
- **additional_params** (`dict`) – dict of additional params to label nodes with
- **show_attrs** (`bool`) – whether to display non-tensor attributes of backward nodes (Requires PyTorch version `>= 1.9`)
- **show_saved** (`bool`) – whether to display saved tensor nodes that are not by custom autograd functions. Saved tensor nodes for custom functions, if present, are always displayed. (Requires PyTorch version `>= 1.9`)
- **max_attr_chars** (`int`) – if `show_attrs` is `True`, sets max number of characters to display for any given attribute.

Returns

`graphviz` representation of autograd graph

Return type

`Digraph`

```
analogvnn.utils.render_autograd_graph.get_autograd_dot_from_module(module: torch.nn.Module,
                                                                    *args: torch.Tensor,
                                                                    additional_params:
                                                                    Optional[dict] = None,
                                                                    show_attrs: bool = True,
                                                                    show_saved: bool = True,
                                                                    max_attr_chars: int = 50,
                                                                    from_forward: bool = False,
                                                                    **kwargs: torch.Tensor) →
                                                                    graphviz.Digraph
```

Runs and make Graphviz representation of PyTorch autograd graph from forward pass.

Parameters

- **module** (*nn.Module*) – PyTorch model
- ***args** (*Tensor*) – input to the model
- **additional_params** (*dict*) – dict of additional params to label nodes with
- **show_attrs** (*bool*) – whether to display non-tensor attributes of backward nodes (Requires PyTorch version >= 1.9)
- **show_saved** (*bool*) – whether to display saved tensor nodes that are not by custom autograd functions. Saved tensor nodes for custom functions, if present, are always displayed. (Requires PyTorch version >= 1.9)
- **max_attr_chars** (*int*) – if show_attrs is *True*, sets max number of characters to display for any given attribute.
- **from_forward** (*bool*) – if *True* then use autograd graph otherwise analogvnn graph
- ****kwargs** (*Tensor*) – input to the model

Returns

graphviz representation of autograd graph

Return type

Digraph

```
analogvnn.utils.render_autograd_graph.save_autograd_graph_from_outputs(filename: Union[str,
                                                                 pathlib.Path], outputs:
                                                                 Union[torch.Tensor, Sequence[torch.Tensor]],
                                                                 named_params:
                                                                 Union[Dict[str, Any],
                                                                 Iterator[Tuple[str,
                                                                 torch.nn.Parameter]]],
                                                                 additional_params:
                                                                 Optional[dict] = None,
                                                                 show_attrs: bool =
                                                                 True, show_saved: bool =
                                                                 True,
                                                                 max_attr_chars: int =
                                                                 50) → str
```

Save Graphviz representation of PyTorch autograd graph from output tensors.

Parameters

- **filename** (*Union[str, Path]*) – filename to save the graph to

- **outputs** (*Union[[Tensor](#), [Sequence\[\[Tensor\]\(#\)\]](#)]*) – output tensor(s) of forward pass
- **named_params** (*Union[[Dict\[\[str\]\(#\), \[Any\]\(#\)\]](#), [Iterator\[Tuple\[\[str\]\(#\), \[Parameter\]\(#\)\]\]\]](#)*) – dict of params to label nodes with
- **additional_params** (*dict*) – dict of additional params to label nodes with
- **show_attrs** (*bool*) – whether to display non-tensor attributes of backward nodes (Requires PyTorch version ≥ 1.9)
- **show_saved** (*bool*) – whether to display saved tensor nodes that are not by custom autograd functions. Saved tensor nodes for custom functions, if present, are always displayed. (Requires PyTorch version ≥ 1.9)
- **max_attr_chars** (*int*) – if `show_attrs` is *True*, sets max number of characters to display for any given attribute.

Returns

The (possibly relative) path of the rendered file.

Return type

[str](#)

```
analogvnn.utils.render_autograd_graph.save_autograd_graph_from_module(filename: Union[str,
                                                                                   pathlib.Path], module:
                                                                                   torch.nn.Module, *args:
                                                                                   torch.Tensor,
                                                                                   additional_params:
                                                                                   Optional[dict] = None,
                                                                                   show_attrs: bool = True,
                                                                                   show_saved: bool =
                                                                                   True, max_attr_chars:
                                                                                   int = 50, from_forward:
                                                                                   bool = False, **kwargs:
                                                                                   torch.Tensor) → str
```

Save Graphviz representation of PyTorch autograd graph from forward pass.

Parameters

- **filename** (*Union[[str](#), [Path](#)]*) – filename to save the graph to
- **module** (*nn.Module*) – PyTorch model
- ***args** (*Tensor*) – input to the model
- **additional_params** (*dict*) – dict of additional params to label nodes with
- **show_attrs** (*bool*) – whether to display non-tensor attributes of backward nodes (Requires PyTorch version ≥ 1.9)
- **show_saved** (*bool*) – whether to display saved tensor nodes that are not by custom autograd functions. Saved tensor nodes for custom functions, if present, are always displayed. (Requires PyTorch version ≥ 1.9)
- **max_attr_chars** (*int*) – if `show_attrs` is *True*, sets max number of characters to display for any given attribute.
- **from_forward** (*bool*) – if *True* then use autograd graph otherwise analogvnn graph
- ****kwargs** (*Tensor*) – input to the model

Returns

The (possibly relative) path of the rendered file.

Return type`str`

`analogvnn.utils.render_autograd_graph.save_autograd_graph_from_trace(filename: Union[str, pathlib.Path], trace) → str`

Save Graphviz representation of PyTorch autograd graph from trace.

Parameters

- **filename** (Union[`str`, `Path`]) – filename to save the graph to
- **trace** (`torch.jit.trace`) – the trace object to visualize.

Returns

The (possibly relative) path of the rendered file.

Return type`str`

`analogvnn.utils.to_tensor_parameter`

Module Contents**Functions**

<code>to_float_tensor(→</code>	<code>Tuple[Union[torch.Tensor,</code>	Converts the given arguments to <code>torch.Tensor</code> of type
<code>None], ...)</code>		<code>torch.float32</code> .
<code>to_nongrad_parameter(→</code>	<code>Tu-</code>	Converts the given arguments to <code>nn.Parameter</code> of type
<code>ple[Union[torch.nn.Parameter, ...]</code>		<code>torch.float32</code> .

`analogvnn.utils.to_tensor_parameter.to_float_tensor(*args) → Tuple[Union[torch.Tensor, None], Ellipsis]`

Converts the given arguments to `torch.Tensor` of type `torch.float32`.

The returned tensors are not trainable.

Parameters

***args** – the arguments to convert.

Returns

the converted arguments.

Return type`tuple`

`analogvnn.utils.to_tensor_parameter.to_nongrad_parameter(*args) → Tuple[Union[torch.nn.Parameter, None], Ellipsis]`

Converts the given arguments to `nn.Parameter` of type `torch.float32`.

The returned parameters are not trainable.

Parameters

***args** – the arguments to convert.

Returns

the converted arguments.

Return type
tuple

Package Contents

```
analogvnn.__package__ = 'analogvnn'
analogvnn.__author__ = 'Vivswan Shah (vivswanshah@pitt.edu)'
analogvnn.__version__
```

1.8 Changelog

1.8.1 1.0.7

- Fixed GeLU backward function equation.

1.8.2 1.0.6

- `Model` is subclass of `BackwardModule` for additional functionality.
- Using `inspect.isclass` to check if `backward_class` is a class in `Linear.set_backward_function`.
- Repr using `self.__class__.__name__` in all classes.

1.8.3 1.0.5 (Patches for Pytorch 2.0.1)

- Removed unnecessary `PseudoParameter.grad` property.
- Patch for Pytorch 2.0.1, add filtering inputs in `BackwardGraph._calculate_gradients`.

1.8.4 1.0.4

- Combined `PseudoParameter` and `PseudoParameterModule` for better visibility.
 - BugFix: fixed save and load of `state_dict` of `PseudoParameter` and transformation module.
- Removed redundant class `analogvnn.parameter.Parameter`.

1.8.5 1.0.3

- Added support for no loss function in `Model` class.
 - If no loss function is provided, the `Model` object will use outputs for gradient computation.
- Added support for multiple loss outputs from loss function.

1.8.6 1.0.2

- Bugfix: removed `graph` from `Layer` class.
 - `graph` was causing issues with nested `Model` objects.
 - Now `_use_autograd_graph` is directly set while compiling the `Model` object.

1.8.7 1.0.1 (Patches for Pytorch 2.0.0)

- added `grad.setter` to `PseudoParameterModule` class.

1.8.8 1.0.0

- Public release.

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

a

- `analogvnn`, 17
- `analogvnn.backward`, 18
- `analogvnn.backward.BackwardFunction`, 18
- `analogvnn.backward.BackwardIdentity`, 19
- `analogvnn.backward.BackwardModule`, 19
- `analogvnn.backward.BackwardUsingForward`, 23
- `analogvnn.fn`, 24
- `analogvnn.fn.dirac_delta`, 24
- `analogvnn.fn.reduce_precision`, 25
- `analogvnn.fn.test`, 26
- `analogvnn.fn.to_matrix`, 26
- `analogvnn.fn.train`, 27
- `analogvnn.graph`, 27
- `analogvnn.graph.AccumulateGrad`, 27
- `analogvnn.graph.AcyclicDirectedGraph`, 28
- `analogvnn.graph.ArgsKwargs`, 32
- `analogvnn.graph.BackwardGraph`, 33
- `analogvnn.graph.ForwardGraph`, 35
- `analogvnn.graph.GraphEnum`, 37
- `analogvnn.graph.ModelGraph`, 37
- `analogvnn.graph.ModelGraphState`, 38
- `analogvnn.graph.to_graph_viz_digraph`, 40
- `analogvnn.nn`, 41
- `analogvnn.nn.activation`, 41
- `analogvnn.nn.activation.Activation`, 41
- `analogvnn.nn.activation.BinaryStep`, 42
- `analogvnn.nn.activation.ELU`, 42
- `analogvnn.nn.activation.Gaussian`, 44
- `analogvnn.nn.activation.Identity`, 45
- `analogvnn.nn.activation.ReLU`, 46
- `analogvnn.nn.activation.Sigmoid`, 48
- `analogvnn.nn.activation.SiLU`, 48
- `analogvnn.nn.activation.Tanh`, 50
- `analogvnn.nn.Linear`, 78
- `analogvnn.nn.module`, 51
- `analogvnn.nn.module.FullSequential`, 51
- `analogvnn.nn.module.Layer`, 51
- `analogvnn.nn.module.Model`, 54
- `analogvnn.nn.module.Sequential`, 58
- `analogvnn.nn.noise`, 59
- `analogvnn.nn.noise.GaussianNoise`, 59
- `analogvnn.nn.noise.LaplacianNoise`, 62
- `analogvnn.nn.noise.Noise`, 65
- `analogvnn.nn.noise.PoissonNoise`, 65
- `analogvnn.nn.noise.UniformNoise`, 69
- `analogvnn.nn.normalize`, 72
- `analogvnn.nn.normalize.Clamp`, 72
- `analogvnn.nn.normalize.LPNorm`, 73
- `analogvnn.nn.normalize.Normalize`, 75
- `analogvnn.nn.precision`, 75
- `analogvnn.nn.precision.Precision`, 75
- `analogvnn.nn.precision.ReducePrecision`, 76
- `analogvnn.nn.precision.StochasticReducePrecision`, 77
- `analogvnn.parameter`, 80
- `analogvnn.parameter.PseudoParameter`, 80
- `analogvnn.utils`, 82
- `analogvnn.utils.common_types`, 85
- `analogvnn.utils.get_model_summaries`, 85
- `analogvnn.utils.is_cpu_cuda`, 86
- `analogvnn.utils.render_autograd_graph`, 88
- `analogvnn.utils.TensorboardModelLog`, 82
- `analogvnn.utils.to_tensor_parameter`, 97

Symbols

__author__ (in module *analogvnn*), 98
 __call__ (*analogvnn.backward.BackwardModule.BackwardModule* attribute), 21
 __call__() (*analogvnn.graph.AccumulateGrad.AccumulateGrad* method), 28
 __call__() (*analogvnn.graph.AcyclicDirectedGraph.AcyclicDirectedGraph* method), 29
 __call__() (*analogvnn.graph.BackwardGraph.BackwardGraph* method), 33
 __call__() (*analogvnn.graph.ForwardGraph.ForwardGraph* method), 35
 __call__() (*analogvnn.nn.module.Layer.Layer* method), 52
 __call__() (*analogvnn.nn.module.Model.Model* method), 55
 __call__() (*analogvnn.nn.module.Sequential.Sequential* method), 58
 __call__() (*analogvnn.parameter.PseudoParameter.PseudoParameter* method), 81
 __constants__ (*analogvnn.nn.Linear.Linear* attribute), 79
 __constants__ (*analogvnn.nn.activation.ELU.SELU* attribute), 43
 __constants__ (*analogvnn.nn.activation.ReLU.PReLU* attribute), 46
 __constants__ (*analogvnn.nn.module.Model.Model* attribute), 54
 __constants__ (*analogvnn.nn.noise.GaussianNoise.GaussianNoise* attribute), 59
 __constants__ (*analogvnn.nn.noise.LaplacianNoise.LaplacianNoise* attribute), 62
 __constants__ (*analogvnn.nn.noise.PoissonNoise.PoissonNoise* attribute), 66
 __constants__ (*analogvnn.nn.noise.UniformNoise.UniformNoise* attribute), 69
 __constants__ (*analogvnn.nn.normalize.LPNorm.LPNorm* attribute), 73
 __constants__ (*analogvnn.nn.precision.ReducePrecision.ReducePrecision* attribute), 76
 __constants__ (*analogvnn.nn.precision.StochasticReducer.StochasticReducer* attribute), 77
 __enter__() (*analogvnn.utils.TensorboardModelLog.TensorboardModelLog* method), 85
 __exit__() (*analogvnn.utils.TensorboardModelLog.TensorboardModelLog* attribute), 83
 __getattr__() (*analogvnn.backward.BackwardModule.BackwardModule* method), 23
 __package__ (in module *analogvnn*), 98
 __post_init__() (*analogvnn.utils.render_autograd_graph.AutoGradDot* method), 90
 __repr__() (*analogvnn.graph.AccumulateGrad.AccumulateGrad* method), 28
 __repr__() (*analogvnn.graph.ArgsKwargs.ArgsKwargs* method), 32
 __repr__() (*analogvnn.parameter.PseudoParameter.PseudoParameter* method), 81
 __version__ (in module *analogvnn*), 98
 _add_layer_data() (*analogvnn.utils.TensorboardModelLog.TensorboardModelLog* method), 83
 _autograd_backward (*analogvnn.backward.BackwardModule.BackwardModule* attribute), 21
 _backward_function (*analogvnn.backward.BackwardFunction.BackwardFunction* attribute), 18
 _backward_module (*analogvnn.nn.module.Layer.Layer* attribute), 52
 _calculate_gradients() (*analogvnn.graph.BackwardGraph.BackwardGraph* method), 34
 _call_impl (*analogvnn.parameter.PseudoParameter.PseudoParameter* attribute), 80
 _call_impl_backward() (*analogvnn.backward.BackwardModule.BackwardModule* method), 21
 _call_impl_forward() (*analogvnn.backward.BackwardModule.BackwardModule* method), 21
 _call_impl_forward() (*analogvnn.nn.module.Layer.Layer* method), 53
 _called (*analogvnn.utils.render_autograd_graph.AutoGradDot* attribute), 90
 _compiled (*analogvnn.nn.module.Model.Model* attribute), 54

_create_edge_label() (analogvnn.graph.AcyclicDirectedGraph.AcyclicDirectedGraph attribute), 80
 static method), 30
 _create_static_sub_graph() (analogvnn.graph.AcyclicDirectedGraph.AcyclicDirectedGraph attribute), 80
 method), 31
 _detach_tensor() (analogvnn.graph.ForwardGraph.ForwardGraph attribute), 52
 static method), 36
 _device (analogvnn.utils.is_cpu_cuda.CPUCuda attribute), 87
 _disable_autograd_backward (analogvnn.backward.BackwardModule.BackwardModule attribute), 21
 _empty_holder_tensor (analogvnn.backward.BackwardModule.BackwardModule attribute), 55
 attribute), 21
 _forward_wrapper() (analogvnn.nn.module.Layer.Layer attribute), 41
 method), 53
 _ignore_tensor (analogvnn.utils.render_autograd_graph.AutoGradDot attribute), 90
 attribute), 90
 _inputs (analogvnn.nn.module.Layer.Layer attribute), 52
 attribute), 90
 _inputs_kwargs (analogvnn.utils.render_autograd_graph.AutoGradDot attribute), 90
 attribute), 90
 _is_static (analogvnn.graph.AcyclicDirectedGraph.AcyclicDirectedGraph attribute), 29
 attribute), 29
 _layer (analogvnn.backward.BackwardModule.BackwardModule attribute), 20
 attribute), 20
 _log_record (analogvnn.utils.TensorboardModelLog.TensorboardModelLog attribute), 83
 attribute), 83
 _loss (analogvnn.graph.ModelGraphState.ModelGraphState attribute), 39
 attribute), 39
 _module (analogvnn.utils.render_autograd_graph.AutoGradDot attribute), 90
 attribute), 90
 _outputs (analogvnn.nn.module.Layer.Layer attribute), 52
 attribute), 90
 _outputs (analogvnn.utils.render_autograd_graph.AutoGradDot attribute), 90
 attribute), 90
 _pass() (analogvnn.graph.BackwardGraph.BackwardGraph attribute), 34
 method), 34
 _pass() (analogvnn.graph.ForwardGraph.ForwardGraph attribute), 36
 method), 36
 _reindex_out_args() (analogvnn.graph.AcyclicDirectedGraph.AcyclicDirectedGraph attribute), 31
 static method), 31
 _seen (analogvnn.utils.render_autograd_graph.AutoGradDot attribute), 90
 attribute), 90
 _set_autograd_backward() (analogvnn.backward.BackwardModule.BackwardModule attribute), 23
 method), 23
 _static_graphs (analogvnn.graph.AcyclicDirectedGraph.AcyclicDirectedGraph attribute), 29
 attribute), 29
 _transformation (analogvnn.parameter.PseudoParameter.PseudoParameter attribute), 80
 attribute), 80
 _transformed (analogvnn.parameter.PseudoParameter.PseudoParameter attribute), 80
 attribute), 80
 _zero (analogvnn.nn.activation.ReLU.PReLU attribute), 46
 attribute), 46

A

AccumulateGrad (class in analogvnn.graph.AccumulateGrad), 27
 accuracy_function (analogvnn.nn.module.Model.Model attribute), 55
 Activation (class in analogvnn.nn.activation.Activation), 41
 AcyclicDirectedGraph (class in analogvnn.graph.AcyclicDirectedGraph), 28
 add_connection() (analogvnn.graph.AcyclicDirectedGraph.AcyclicDirectedGraph attribute), 29
 method), 29
 add_edge() (analogvnn.graph.AcyclicDirectedGraph.AcyclicDirectedGraph attribute), 29
 method), 29
 AddEdgeOp (analogvnn.utils.render_autograd_graph.AutoGradDot attribute), 91
 method), 91
 add_directed_graph (analogvnn.utils.render_autograd_graph.AutoGradDot attribute), 91
 method), 91
 add_graph() (analogvnn.utils.TensorboardModelLog.TensorboardModelLog attribute), 83
 method), 83
 add_holder_tensor() (analogvnn.utils.render_autograd_graph.AutoGradDot attribute), 90
 method), 90
 add_seen() (analogvnn.utils.render_autograd_graph.AutoGradDot attribute), 92
 method), 92
 add_sequence() (analogvnn.nn.module.Sequential.Sequential attribute), 58
 method), 58
 add_summary() (analogvnn.utils.TensorboardModelLog.TensorboardModelLog attribute), 84
 method), 84
 add_tensor() (analogvnn.utils.render_autograd_graph.AutoGradDot attribute), 91
 method), 91
 allow_loops (analogvnn.graph.ModelGraphState.ModelGraphState attribute), 39
 attribute), 39
 alpha (analogvnn.nn.activation.ELU.SELU attribute), 43
 attribute), 43
 alpha (analogvnn.nn.activation.ReLU.PReLU attribute), 46
 attribute), 46
 analogvnn module, 17
 analogvnn.backward module, 18
 analogvnn.backward.BackwardFunction module, 18
 analogvnn.backward.BackwardIdentity module, 19

analogvnn.backward.BackwardModule	analogvnn.nn.activation.SiLU
module, 19	module, 48
analogvnn.backward.BackwardUsingForward	analogvnn.nn.activation.Tanh
module, 23	module, 50
analogvnn.fn	analogvnn.nn.Linear
module, 24	module, 78
analogvnn.fn.dirac_delta	analogvnn.nn.module
module, 24	module, 51
analogvnn.fn.reduce_precision	analogvnn.nn.module.FullSequential
module, 25	module, 51
analogvnn.fn.test	analogvnn.nn.module.Layer
module, 26	module, 51
analogvnn.fn.to_matrix	analogvnn.nn.module.Model
module, 26	module, 54
analogvnn.fn.train	analogvnn.nn.module.Sequential
module, 27	module, 58
analogvnn.graph	analogvnn.nn.noise
module, 27	module, 59
analogvnn.graph.AccumulateGrad	analogvnn.nn.noise.GaussianNoise
module, 27	module, 59
analogvnn.graph.AcyclicDirectedGraph	analogvnn.nn.noise.LaplacianNoise
module, 28	module, 62
analogvnn.graph.ArgsKwargs	analogvnn.nn.noise.Noise
module, 32	module, 65
analogvnn.graph.BackwardGraph	analogvnn.nn.noise.PoissonNoise
module, 33	module, 65
analogvnn.graph.ForwardGraph	analogvnn.nn.noise.UniformNoise
module, 35	module, 69
analogvnn.graph.GraphEnum	analogvnn.nn.normalize
module, 37	module, 72
analogvnn.graph.ModelGraph	analogvnn.nn.normalize.Clamp
module, 37	module, 72
analogvnn.graph.ModelGraphState	analogvnn.nn.normalize.LPNorm
module, 38	module, 73
analogvnn.graph.to_graph_viz_digraph	analogvnn.nn.normalize.Normalize
module, 40	module, 75
analogvnn.nn	analogvnn.nn.precision
module, 41	module, 75
analogvnn.nn.activation	analogvnn.nn.precision.Precision
module, 41	module, 75
analogvnn.nn.activation.Activation	analogvnn.nn.precision.ReducePrecision
module, 41	module, 76
analogvnn.nn.activation.BinaryStep	analogvnn.nn.precision.StochasticReducePrecision
module, 42	module, 77
analogvnn.nn.activation.ELU	analogvnn.parameter
module, 42	module, 80
analogvnn.nn.activation.Gaussian	analogvnn.parameter.PseudoParameter
module, 44	module, 80
analogvnn.nn.activation.Identity	analogvnn.utils
module, 45	module, 82
analogvnn.nn.activation.ReLU	analogvnn.utils.common_types
module, 46	module, 85
analogvnn.nn.activation.Sigmoid	analogvnn.utils.get_model_summaries
module, 48	module, 85

analogvnn.utils.is_cpu_cuda
 module, 86
 analogvnn.utils.render_autograd_graph
 module, 88
 analogvnn.utils.TensorboardModelLog
 module, 82
 analogvnn.utils.to_tensor_parameter
 module, 97
 args (analogvnn.graph.ArgsKwargs.ArgsKwargs attribute), 32
 ArgsKwargs (class in analogvnn.graph.ArgsKwargs), 32
 ArgsKwargsInput (in module analogvnn.graph.ArgsKwargs), 33
 ArgsKwargsOutput (in module analogvnn.graph.ArgsKwargs), 33
 auto_apply() (analogvnn.backward.BackwardModule.BackwardModule method), 22
 AutoGradDot (class in analogvnn.utils.render_autograd_graph), 88
B
 backward() (analogvnn.backward.BackwardFunction.BackwardFunction method), 18
 backward() (analogvnn.backward.BackwardIdentity.BackwardIdentity method), 19
 backward() (analogvnn.backward.BackwardModule.BackwardModule method), 21
 backward() (analogvnn.backward.BackwardModule.BackwardModule static method), 20
 backward() (analogvnn.backward.BackwardUsingForward.BackwardUsingForward method), 23
 backward() (analogvnn.nn.activation.BinaryStep.BinaryStep method), 42
 backward() (analogvnn.nn.activation.ELU.SELU method), 43
 backward() (analogvnn.nn.activation.Gaussian.Gaussian method), 44
 backward() (analogvnn.nn.activation.Gaussian.GeLU method), 45
 backward() (analogvnn.nn.activation.Identity.Identity method), 45
 backward() (analogvnn.nn.activation.ReLU.PReLU method), 46
 backward() (analogvnn.nn.activation.Sigmoid.Logistic method), 49
 backward() (analogvnn.nn.activation.SiLU.SiLU method), 48
 backward() (analogvnn.nn.activation.Tanh.Tanh method), 50
 backward() (analogvnn.nn.Linear.LinearBackpropagation method), 79
 backward() (analogvnn.nn.module.Model.Model method), 56
 backward() (analogvnn.nn.normalize.Clamp.Clamp method), 72
 backward() (analogvnn.nn.normalize.Clamp.Clamp01 method), 73
 backward_function (analogvnn.backward.BackwardFunction.BackwardFunction property), 18
 backward_function (analogvnn.nn.module.Layer.Layer property), 52
 backward_graph (analogvnn.graph.ModelGraph.ModelGraph attribute), 38
 backward_graph (analogvnn.nn.module.Model.Model attribute), 55
 BackwardFunction (class in analogvnn.backward.BackwardFunction), 18
 BackwardGraph (class in analogvnn.graph.BackwardGraph), 33
 BackwardIdentity (class in analogvnn.backward.BackwardIdentity), 19
 BackwardModule (class in analogvnn.backward.BackwardModule), 19
 BackwardModule.AutoGradBackward (class in analogvnn.backward.BackwardModule), 20
 BackwardUsingForward (class in analogvnn.backward.BackwardUsingForward), 23
 bias (analogvnn.nn.module.Linear attribute), 79
 BinaryStep (class in analogvnn.nn.activation.BinaryStep), 42
 bit_precision (analogvnn.nn.precision.ReducePrecision.ReducePrecision property), 76
 bit_precision (analogvnn.nn.precision.StochasticReducePrecision.Stochastic property), 77
C
 calc_high_low() (analogvnn.nn.noise.UniformNoise.UniformNoise static method), 70
 calc_leakage() (analogvnn.nn.noise.GaussianNoise.GaussianNoise static method), 60
 calc_leakage() (analogvnn.nn.noise.LaplacianNoise.LaplacianNoise static method), 63
 calc_leakage() (analogvnn.nn.noise.UniformNoise.UniformNoise static method), 70
 calc_max_leakage() (analogvnn.nn.noise.PoissonNoise.PoissonNoise static method), 66
 calc_precision() (analogvnn.nn.noise.GaussianNoise.GaussianNoise static method), 60
 calc_precision() (analogvnn.nn.noise.LaplacianNoise.LaplacianNoise static method), 63
 calc_precision() (analogvnn.nn.noise.PoissonNoise.PoissonNoise static method), 66

`calc_precision()` (*analogvnn.nn.noise.UniformNoise.UniformNoise* static method), 70
`calc_scale()` (*analogvnn.nn.noise.LaplacianNoise.LaplacianNoise* static method), 63
`calc_scale()` (*analogvnn.nn.noise.PoissonNoise.PoissonNoise* static method), 66
`calc_std()` (*analogvnn.nn.noise.GaussianNoise.GaussianNoise* static method), 59
`calculate()` (*analogvnn.graph.BackwardGraph.BackwardGraph* method), 34
`calculate()` (*analogvnn.graph.ForwardGraph.ForwardGraph* method), 36
`call_super_init` (*analogvnn.nn.module.Layer.Layer* attribute), 52
`cdf()` (*analogvnn.nn.noise.GaussianNoise.GaussianNoise* method), 61
`cdf()` (*analogvnn.nn.noise.LaplacianNoise.LaplacianNoise* method), 64
`cdf()` (*analogvnn.nn.noise.PoissonNoise.PoissonNoise* method), 68
`cdf()` (*analogvnn.nn.noise.UniformNoise.UniformNoise* method), 71
`check_edge_parameters()` (*analogvnn.graph.AcyclicDirectedGraph.AcyclicDirectedGraph* static method), 30
`Clamp` (class in *analogvnn.nn.normalize.Clamp*), 72
`Clamp01` (class in *analogvnn.nn.normalize.Clamp*), 72
`close()` (*analogvnn.utils.TensorboardModelLog.TensorboardModelLog* method), 84
`compile()` (*analogvnn.graph.AcyclicDirectedGraph.AcyclicDirectedGraph* method), 30
`compile()` (*analogvnn.graph.BackwardGraph.BackwardGraph* method), 34
`compile()` (*analogvnn.graph.ForwardGraph.ForwardGraph* method), 35
`compile()` (*analogvnn.graph.ModelGraph.ModelGraph* method), 38
`compile()` (*analogvnn.nn.module.FullSequential.FullSequential* method), 51
`compile()` (*analogvnn.nn.module.Model.Model* method), 55
`compile()` (*analogvnn.nn.module.Sequential.Sequential* method), 58
`convert_to_precision()` (*analogvnn.nn.precision.ReducePrecision.ReducePrecision* static method), 76
`convert_to_precision()` (*analogvnn.nn.precision.StochasticReducePrecision.StochasticReducePrecision* static method), 78
`CPUcuda` (class in *analogvnn.utils.is_cpu_cuda*), 86
`create_tensorboard()` (*analogvnn.nn.module.Model.Model* method), 57
`del_ignore_tensor()` (*analogvnn.utils.render_autograd_graph.AutoGradDot* method), 90
`device` (*analogvnn.nn.module.Model.Model* attribute), 55
`device` (*analogvnn.utils.is_cpu_cuda.CPUcuda* property), 86
`device_name` (*analogvnn.utils.is_cpu_cuda.CPUcuda* attribute), 87
`divide` (*analogvnn.nn.precision.ReducePrecision.ReducePrecision* attribute), 76
`dot` (*analogvnn.utils.render_autograd_graph.AutoGradDot* attribute), 90

E

`ELU` (class in *analogvnn.nn.activation.ELU*), 43
`extra_repr()` (*analogvnn.nn.activation.Identity.Identity* method), 45
`extra_repr()` (*analogvnn.nn.Linear.Linear* method), 79
`extra_repr()` (*analogvnn.nn.noise.GaussianNoise.GaussianNoise* method), 61
`extra_repr()` (*analogvnn.nn.noise.LaplacianNoise.LaplacianNoise* method), 65
`extra_repr()` (*analogvnn.nn.noise.PoissonNoise.PoissonNoise* method), 68
`extra_repr()` (*analogvnn.nn.noise.UniformNoise.UniformNoise* method), 71
`extra_repr()` (*analogvnn.nn.precision.ReducePrecision.ReducePrecision* method), 76
`extra_repr()` (*analogvnn.nn.precision.StochasticReducePrecision.StochasticReducePrecision* method), 78

F

`fit()` (*analogvnn.nn.module.Model.Model* method), 57
`forward` (*analogvnn.parameter.PseudoParameter.PseudoParameter* attribute), 80
`forward()` (*analogvnn.backward.BackwardModule.BackwardModule* method), 21
`forward()` (*analogvnn.backward.BackwardModule.BackwardModule.Auto* static method), 20
`forward()` (*analogvnn.nn.activation.BinaryStep.BinaryStep* static method), 42
`forward()` (*analogvnn.nn.activation.ELU.SELU* method), 43
`forward()` (*analogvnn.nn.activation.Gaussian.Gaussian* static method), 44
`forward()` (*analogvnn.nn.activation.Gaussian.GeLU* static method), 44
`forward()` (*analogvnn.nn.activation.Identity.Identity* static method), 45
`forward()` (*analogvnn.nn.activation.ReLU.PReLU* method), 46

`forward()` (*analogvnn.nn.activation.Sigmoid.Logistic static method*), 49
`forward()` (*analogvnn.nn.activation.SiLU.SiLU static method*), 48
`forward()` (*analogvnn.nn.activation.Tanh.Tanh static method*), 50
`forward()` (*analogvnn.nn.Linear.LinearBackpropagation method*), 78
`forward()` (*analogvnn.nn.module.Model.Model method*), 55
`forward()` (*analogvnn.nn.noise.GaussianNoise.GaussianNoise method*), 61
`forward()` (*analogvnn.nn.noise.LaplacianNoise.LaplacianNoise method*), 64
`forward()` (*analogvnn.nn.noise.PoissonNoise.PoissonNoise method*), 68
`forward()` (*analogvnn.nn.noise.UniformNoise.UniformNoise method*), 71
`forward()` (*analogvnn.nn.normalize.Clamp.Clamp static method*), 72
`forward()` (*analogvnn.nn.normalize.Clamp.Clamp01 static method*), 72
`forward()` (*analogvnn.nn.normalize.LPNorm.LPNorm method*), 74
`forward()` (*analogvnn.nn.normalize.LPNorm.LPNormW method*), 74
`forward()` (*analogvnn.nn.precision.ReducePrecision.ReducePrecision method*), 77
`forward()` (*analogvnn.nn.precision.StochasticReducePrecision.StochasticReducePrecision method*), 78
`forward_graph` (*analogvnn.graph.ModelGraph.ModelGraph attribute*), 38
`forward_graph` (*analogvnn.nn.module.Model.Model attribute*), 54
`forward_input_output_graph` (*analogvnn.graph.ModelGraphState.ModelGraphState attribute*), 39
ForwardGraph (class in *analogvnn.graph.ForwardGraph*), 35
`from_args_kwargs_object()` (*analogvnn.graph.ArgsKwargs.ArgsKwargs static method*), 33
`from_forward()` (*analogvnn.graph.BackwardGraph.BackwardGraph method*), 34
FullSequential (class in *analogvnn.nn.module.FullSequential*), 51

G

Gaussian (class in *analogvnn.nn.activation.Gaussian*), 44
`gaussian_dirac_delta()` (in *module analogvnn.fn.dirac_delta*), 24
GaussianNoise (class in *analogvnn.nn.noise.GaussianNoise*), 59
GeLU (class in *analogvnn.nn.activation.Gaussian*), 44
`get_autograd_dot_from_module()` (in *module analogvnn.utils.render_autograd_graph*), 94
`get_autograd_dot_from_outputs()` (in *module analogvnn.utils.render_autograd_graph*), 94
`get_autograd_dot_from_trace()` (in *module analogvnn.utils.render_autograd_graph*), 93
`get_layer()` (*analogvnn.backward.BackwardModule.BackwardModule method*), 22
`get_model_summaries()` (in *module analogvnn.utils.get_model_summaries*), 85
`get_module_device()` (*analogvnn.utils.is_cpu_cuda.CPUCuda method*), 87
`get_tensor_name()` (*analogvnn.utils.render_autograd_graph.AutoGrad method*), 91
grad (*analogvnn.graph.AccumulateGrad.AccumulateGrad attribute*), 28
graph (*analogvnn.graph.AcyclicDirectedGraph.AcyclicDirectedGraph attribute*), 29
GRAPH_NODE_TYPE (in *module analogvnn.graph.GraphEnum*), 37
graph_state (*analogvnn.graph.AcyclicDirectedGraph.AcyclicDirectedGraph attribute*), 29
GraphEnum (class in *analogvnn.graph.GraphEnum*), 37
graphs (*analogvnn.nn.module.Model.Model attribute*), 54

H

`has_forward()` (*analogvnn.backward.BackwardModule.BackwardModule method*), 22
high (*analogvnn.nn.noise.UniformNoise.UniformNoise attribute*), 69

I

Identity (class in *analogvnn.nn.activation.Identity*), 45
`identity()` (*analogvnn.parameter.PseudoParameter.PseudoParameter static method*), 81
`ignore_tensor` (*analogvnn.utils.render_autograd_graph.AutoGradDot property*), 89
in_features (*analogvnn.nn.Linear.Linear attribute*), 79
`initialise()` (*analogvnn.nn.activation.Activation.InitImplement static method*), 41
`initialise()` (*analogvnn.nn.activation.ELU.SELU static method*), 43
`initialise()` (*analogvnn.nn.activation.ReLU.PReLU static method*), 47
`initialise()` (*analogvnn.nn.activation.ReLU.ReLU static method*), 47
`initialise()` (*analogvnn.nn.activation.Sigmoid.Logistic static method*), 49
`initialise()` (*analogvnn.nn.activation.Tanh.Tanh static method*), 50

initialise_() (*analogvnn.nn.activation.Activation.InitImplement static method*), 41
initialise_() (*analogvnn.nn.activation.ELU.SELU static method*), 43
initialise_() (*analogvnn.nn.activation.ReLU.PReLU static method*), 47
initialise_() (*analogvnn.nn.activation.ReLU.ReLU static method*), 47
initialise_() (*analogvnn.nn.activation.Sigmoid.LogisticLaplacianNoise static method*), 49
initialise_() (*analogvnn.nn.activation.Tanh.Tanh static method*), 50
InitImplement (class in *analogvnn.nn.activation.Activation*), 41
INPUT (*analogvnn.graph.AcyclicDirectedGraph.AcyclicDirectedGraph attribute*), 29
INPUT (*analogvnn.graph.GraphEnum.GraphEnum attribute*), 37
INPUT (*analogvnn.graph.ModelGraphState.ModelGraphState attribute*), 39
input_output_connections (*analogvnn.graph.AccumulateGrad.AccumulateGrad attribute*), 28
InputOutput (class in *analogvnn.graph.ArgsKwargs*), 32
inputs (*analogvnn.graph.ArgsKwargs.InputOutput attribute*), 32
inputs (*analogvnn.graph.ModelGraphState.ModelGraphState property*), 38
inputs (*analogvnn.nn.module.Layer.Layer property*), 52
inputs (*analogvnn.utils.render_autograd_graph.AutoGradDot property*), 89
inputs_kwargs (*analogvnn.utils.render_autograd_graph.AutoGradDot property*), 89
is_cpu (*analogvnn.utils.is_cpu_cuda.CPUCuda property*), 86
is_cpu_cuda (in module *analogvnn.utils.is_cpu_cuda*), 87
is_cuda (*analogvnn.utils.is_cpu_cuda.CPUCuda property*), 86
is_empty() (*analogvnn.graph.ArgsKwargs.ArgsKwargs method*), 32
is_seen() (*analogvnn.utils.render_autograd_graph.AutoGradDot method*), 92
is_using_cuda (*analogvnn.utils.is_cpu_cuda.CPUCuda property*), 87

K
kwargs (*analogvnn.graph.ArgsKwargs.ArgsKwargs attribute*), 32

L
L1Norm (class in *analogvnn.nn.normalize.LPNorm*), 74
L1NormM (class in *analogvnn.nn.normalize.LPNorm*), 74
L1NormW (class in *analogvnn.nn.normalize.LPNorm*), 74
L2Norm (class in *analogvnn.nn.normalize.LPNorm*), 74
L2NormM (class in *analogvnn.nn.normalize.LPNorm*), 74
L2NormW (class in *analogvnn.nn.normalize.LPNorm*), 74
L2NormWM (class in *analogvnn.nn.normalize.LPNorm*), 75
LaplacianNoise (class in *analogvnn.nn.noise.LaplacianNoise*), 62
layer (*analogvnn.backward.BackwardModule.BackwardModule property*), 20
Layer (class in *analogvnn.nn.module.Layer*), 51
layer_data (*analogvnn.utils.TensorboardModelLog.TensorboardModelLog attribute*), 83
leakage (*analogvnn.nn.noise.GaussianNoise.GaussianNoise attribute*), 59
leakage (*analogvnn.nn.noise.LaplacianNoise.LaplacianNoise attribute*), 62
leakage (*analogvnn.nn.noise.PoissonNoise.PoissonNoise property*), 65
leakage (*analogvnn.nn.noise.UniformNoise.UniformNoise attribute*), 69
LeakyReLU (class in *analogvnn.nn.activation.ReLU*), 47
Linear (class in *analogvnn.nn.Linear*), 79
LinearBackpropagation (class in *analogvnn.nn.Linear*), 78
log_prob() (*analogvnn.nn.noise.GaussianNoise.GaussianNoise method*), 61
log_prob() (*analogvnn.nn.noise.LaplacianNoise.LaplacianNoise method*), 64
log_prob() (*analogvnn.nn.noise.PoissonNoise.PoissonNoise method*), 68
log_prob() (*analogvnn.nn.noise.UniformNoise.UniformNoise method*), 71
Logistic (class in *analogvnn.nn.activation.Sigmoid*), 48
loss (*analogvnn.graph.ModelGraphState.ModelGraphState property*), 39
loss() (*analogvnn.nn.module.Model.Model method*), 56
loss_function (*analogvnn.nn.module.Model.Model attribute*), 55
low (*analogvnn.nn.noise.UniformNoise.UniformNoise attribute*), 69
LPNorm (class in *analogvnn.nn.normalize.LPNorm*), 73
LPNormW (class in *analogvnn.nn.normalize.LPNorm*), 74

M
make_autograd_obj_from_module() (in module *analogvnn.utils.render_autograd_graph*), 93
make_autograd_obj_from_outputs() (in module *analogvnn.utils.render_autograd_graph*), 92
make_max_1 (*analogvnn.nn.normalize.LPNorm.LPNorm attribute*), 74

`max_attr_chars` (*analogvnn.utils.render_autograd_graph.AutoGradDot* attribute), 90
`max_leakage` (*analogvnn.nn.noise.PoissonNoise.PoissonNoise* attribute), 66
`mean` (*analogvnn.nn.noise.UniformNoise.UniformNoise* property), 69
`model` (*analogvnn.utils.TensorboardModelLog.TensorboardModelLog* attribute), 83
`Model` (class in *analogvnn.nn.module.Model*), 54
`ModelGraph` (class in *analogvnn.graph.ModelGraph*), 37
`ModelGraphState` (class in *analogvnn.graph.ModelGraphState*), 38
`module`
 `analogvnn`, 17
 `analogvnn.backward`, 18
 `analogvnn.backward.BackwardFunction`, 18
 `analogvnn.backward.BackwardIdentity`, 19
 `analogvnn.backward.BackwardModule`, 19
 `analogvnn.backward.BackwardUsingForward`, 23
 `analogvnn.fn`, 24
 `analogvnn.fn.dirac_delta`, 24
 `analogvnn.fn.reduce_precision`, 25
 `analogvnn.fn.test`, 26
 `analogvnn.fn.to_matrix`, 26
 `analogvnn.fn.train`, 27
 `analogvnn.graph`, 27
 `analogvnn.graph.AccumulateGrad`, 27
 `analogvnn.graph.AcyclicDirectedGraph`, 28
 `analogvnn.graph.ArgsKwargs`, 32
 `analogvnn.graph.BackwardGraph`, 33
 `analogvnn.graph.ForwardGraph`, 35
 `analogvnn.graph.GraphEnum`, 37
 `analogvnn.graph.ModelGraph`, 37
 `analogvnn.graph.ModelGraphState`, 38
 `analogvnn.graph.to_graph_viz_digraph`, 40
 `analogvnn.nn`, 41
 `analogvnn.nn.activation`, 41
 `analogvnn.nn.activation.Activation`, 41
 `analogvnn.nn.activation.BinaryStep`, 42
 `analogvnn.nn.activation.ELU`, 42
 `analogvnn.nn.activation.Gaussian`, 44
 `analogvnn.nn.activation.Identity`, 45
 `analogvnn.nn.activation.ReLU`, 46
 `analogvnn.nn.activation.Sigmoid`, 48
 `analogvnn.nn.activation.SiLU`, 48
 `analogvnn.nn.activation.Tanh`, 50
 `analogvnn.nn.Linear`, 78
 `analogvnn.nn.module`, 51
 `analogvnn.nn.module.FullSequential`, 51
 `analogvnn.nn.module.Layer`, 51
 `analogvnn.nn.module.Model`, 54
 `analogvnn.nn.module.Sequential`, 58
 `analogvnn.nn.noise`, 59
 `analogvnn.nn.noise.GaussianNoise`, 59
 `analogvnn.nn.noise.LaplacianNoise`, 62
 `analogvnn.nn.noise.Noise`, 65
 `analogvnn.nn.noise.PoissonNoise`, 65
 `analogvnn.nn.noise.UniformNoise`, 69
 `analogvnn.nn.normalize`, 72
 `analogvnn.nn.normalize.Clamp`, 72
 `analogvnn.nn.normalize.LPNorm`, 73
 `analogvnn.nn.normalize.Normalize`, 75
 `analogvnn.nn.precision`, 75
 `analogvnn.nn.precision.Precision`, 75
 `analogvnn.nn.precision.ReducePrecision`, 76
 `analogvnn.nn.precision.StochasticReducePrecision`, 77
 `analogvnn.parameter`, 80
 `analogvnn.parameter.PseudoParameter`, 80
 `analogvnn.utils`, 82
 `analogvnn.utils.common_types`, 85
 `analogvnn.utils.get_model_summaries`, 85
 `analogvnn.utils.is_cpu_cuda`, 86
 `analogvnn.utils.render_autograd_graph`, 88
 `analogvnn.utils.TensorboardModelLog`, 82
 `analogvnn.utils.to_tensor_parameter`, 97
`module` (*analogvnn.graph.AccumulateGrad.AccumulateGrad* attribute), 28
`module` (*analogvnn.utils.render_autograd_graph.AutoGradDot* property), 89

N

`name` (*analogvnn.nn.activation.Identity.Identity* attribute), 45
`named_registered_children()`
 (*analogvnn.nn.module.Layer.Layer* method), 53
`named_registered_children()`
 (*analogvnn.nn.module.Model.Model* method), 55
`Noise` (class in *analogvnn.nn.noise.Noise*), 65
`Normalize` (class in *analogvnn.nn.normalize.Normalize*), 75

O

`on_compile()` (*analogvnn.utils.TensorboardModelLog.TensorboardModelLog* method), 83
`optimizer` (*analogvnn.nn.module.Model.Model* attribute), 55
`out_features` (*analogvnn.nn.Linear.Linear* attribute), 79
`OUTPUT` (*analogvnn.graph.AcyclicDirectedGraph.AcyclicDirectedGraph* attribute), 29
`OUTPUT` (*analogvnn.graph.GraphEnum.GraphEnum* attribute), 37

OUTPUT (*analogvnn.graph.ModelGraphState.ModelGraphState* attribute), 39
 outputs (*analogvnn.graph.ArgsKwargs.InputOutput* attribute), 32
 outputs (*analogvnn.graph.ModelGraphState.ModelGraphState* property), 39
 outputs (*analogvnn.nn.module.Layer.Layer* property), 52
 outputs (*analogvnn.utils.render_autograd_graph.AutoGradDot* property), 89
P
 p (*analogvnn.nn.normalize.LPNorm.LPNorm* attribute), 73
 param_map (*analogvnn.utils.render_autograd_graph.AutoGradDot* attribute), 90
 parameterize() (*analogvnn.parameter.PseudoParameter.PseudoParameter* class method), 82
 parametrize_module() (*analogvnn.parameter.PseudoParameter.PseudoParameter* method), 84
 parametrize_module() (*analogvnn.parameter.PseudoParameter.PseudoParameter* class method), 82
 parse_args_kwargs() (*analogvnn.graph.AcyclicDirectedGraph.AcyclicDirectedGraph* method), 31
 pdf() (*analogvnn.nn.noise.GaussianNoise.GaussianNoise* method), 60
 pdf() (*analogvnn.nn.noise.LaplacianNoise.LaplacianNoise* method), 63
 pdf() (*analogvnn.nn.noise.PoissonNoise.PoissonNoise* method), 67
 pdf() (*analogvnn.nn.noise.UniformNoise.UniformNoise* method), 70
 PoissonNoise (class in *analogvnn.nn.noise.PoissonNoise*), 65
 precision (*analogvnn.nn.noise.GaussianNoise.GaussianNoise* attribute), 59
 precision (*analogvnn.nn.noise.LaplacianNoise.LaplacianNoise* attribute), 62
 precision (*analogvnn.nn.noise.PoissonNoise.PoissonNoise* attribute), 66
 precision (*analogvnn.nn.noise.UniformNoise.UniformNoise* attribute), 70
 precision (*analogvnn.nn.precision.ReducePrecision.ReducePrecision* attribute), 76
 precision (*analogvnn.nn.precision.StochasticReducePrecision.StochasticReducePrecision* attribute), 78
 Precision (class in *analogvnn.nn.precision.Precision*), 75
 precision_width (*analogvnn.nn.precision.ReducePrecision.ReducePrecision* property), 76
 precision_width (*analogvnn.nn.precision.StochasticReducePrecision.StochasticReducePrecision* property), 77
 PReLU (class in *analogvnn.nn.activation.ReLU*), 46
 PseudoParameter (class in *analogvnn.parameter.PseudoParameter*), 80
R
 rate_factor (*analogvnn.nn.noise.PoissonNoise.PoissonNoise* property), 66
 ready_for_backward() (*analogvnn.graph.ModelGraphState.ModelGraphState* method), 39
 ready_for_forward() (*analogvnn.graph.ModelGraphState.ModelGraphState* method), 39
 reduce_precision() (in module *analogvnn.fn.reduce_precision*), 25
 ReducePrecision (class in *analogvnn.nn.precision.ReducePrecision*), 76
 register_testing() (*analogvnn.utils.TensorboardModelLog.TensorboardModelLog* method), 84
 register_training() (*analogvnn.utils.TensorboardModelLog.TensorboardModelLog* method), 84
 registered_children() (*analogvnn.nn.module.Layer.Layer* method), 53
 ReLU (class in *analogvnn.nn.activation.ReLU*), 47
 render() (*analogvnn.graph.AcyclicDirectedGraph.AcyclicDirectedGraph* method), 31
 reset_parameters() (*analogvnn.nn.Linear.Linear* method), 79
 reset_params() (*analogvnn.utils.render_autograd_graph.AutoGradDot* method), 90
 right_inverse (*analogvnn.parameter.PseudoParameter.PseudoParameter* attribute), 81
S
 save (*analogvnn.graph.AcyclicDirectedGraph.AcyclicDirectedGraph* attribute), 29
 save_autograd_graph_from_module() (in module *analogvnn.utils.render_autograd_graph*), 96
 save_autograd_graph_from_outputs() (in module *analogvnn.utils.render_autograd_graph*), 95
 save_autograd_graph_from_trace() (in module *analogvnn.utils.render_autograd_graph*), 97
 scale (*analogvnn.nn.noise.LaplacianNoise.LaplacianNoise* attribute), 62
 scale (*analogvnn.nn.noise.PoissonNoise.PoissonNoise* attribute), 66
 scale_factor (*analogvnn.nn.activation.ELU.SELU* attribute), 42
 Sequential (class in *analogvnn.nn.module.Sequential*), 58

set_backward_function() (analogvnn.backward.BackwardFunction.BackwardFunction attribute), 29
 (analogvnn.backward.BackwardFunction.BackwardFunction method), 18
 set_backward_function() (analogvnn.nn.module.Layer.Layer method), 52
 set_device() (analogvnn.utils.is_cpu_cuda.CPUCuda method), 87
 set_grad_of() (analogvnn.backward.BackwardModule.BackwardModule static method), 23
 set_layer() (analogvnn.backward.BackwardModule.BackwardModule static method), 22
 set_log_dir() (analogvnn.utils.TensorboardModelLog.TensorboardModelLog method), 83
 set_loss() (analogvnn.graph.ModelGraphState.ModelGraphState method), 40
 set_original_data() (analogvnn.parameter.PseudoParameter.PseudoParameter method), 81
 set_transformation() (analogvnn.parameter.PseudoParameter.PseudoParameter method), 81
 show_attrs (analogvnn.utils.render_autograd_graph.AutoGradDot attribute), 90
 show_saved (analogvnn.utils.render_autograd_graph.AutoGradDot attribute), 90
 Sigmoid (class in analogvnn.nn.activation.Sigmoid), 49
 SiLU (class in analogvnn.nn.activation.SiLU), 48
 size_to_str() (in module analogvnn.utils.render_autograd_graph), 88
 static_cdf() (analogvnn.nn.noise.GaussianNoise.GaussianNoise static method), 61
 static_cdf() (analogvnn.nn.noise.LaplacianNoise.LaplacianNoise static method), 64
 static_cdf() (analogvnn.nn.noise.PoissonNoise.PoissonNoise static method), 67
 staticmethod_leakage() (analogvnn.nn.noise.PoissonNoise.PoissonNoise static method), 67
 std (analogvnn.nn.noise.GaussianNoise.GaussianNoise attribute), 59
 stddev (analogvnn.nn.noise.GaussianNoise.GaussianNoise property), 59
 stddev (analogvnn.nn.noise.LaplacianNoise.LaplacianNoise property), 62
 stddev (analogvnn.nn.noise.UniformNoise.UniformNoise property), 69
 stochastic_reduce_precision() (in module analogvnn.fn.reduce_precision), 25
 StochasticReducePrecision (class in analogvnn.nn.precision.StochasticReducePrecision), 77
 STOP (analogvnn.graph.AcyclicDirectedGraph.AcyclicDirectedGraph attribute), 29
 STOP (analogvnn.graph.GraphEnum.GraphEnum attribute), 37
 STOP (analogvnn.graph.ModelGraphState.ModelGraphState attribute), 39
 subscribe_tensorboard() (analogvnn.nn.module.Model.Model method), 57
 subscribe_tensorboard() (analogvnn.parameter.PseudoParameter.PseudoParameter static method), 82
 Tanh (class in analogvnn.nn.activation.Tanh), 50
 TENSOR_CALLABLE (in module analogvnn.utils.common_types), 85
 TENSOR_OPERABLE (in module analogvnn.utils.common_types), 85
 tensorboard (analogvnn.nn.module.Model.Model attribute), 54
 tensorboard (analogvnn.utils.TensorboardModelLog.TensorboardModelLog attribute), 83
 TensorboardModelLog (class in analogvnn.utils.TensorboardModelLog), 83
 TENSORS (in module analogvnn.utils.common_types), 85
 test() (in module analogvnn.fn.test), 26
 test_on() (analogvnn.nn.module.Model.Model method), 56
 to_args_kwargs_object() (analogvnn.graph.ArgsKwargs.ArgsKwargs class method), 32
 to_float_tensor() (in module analogvnn.utils.to_tensor_parameter), 97
 to_graphviz_digraph() (in module analogvnn.graph.to_graph_viz_digraph), 40
 to_matrix() (in module analogvnn.fn.to_matrix), 26
 to_nongrad_parameter() (in module analogvnn.utils.to_tensor_parameter), 97
 train() (in module analogvnn.fn.train), 27
 train_on() (analogvnn.nn.module.Model.Model method), 56
 transformation (analogvnn.parameter.PseudoParameter.PseudoParameter property), 80
U
 UniformNoise (class in analogvnn.nn.noise.UniformNoise), 69
 use_autograd_graph (analogvnn.graph.ModelGraphState.ModelGraphState attribute), 39
 use_autograd_graph (analogvnn.nn.module.Layer.Layer property), 52

`use_autograd_graph` (*analogvnn.nn.module.Model.Model*
property), [54](#)

`use_cpu()` (*analogvnn.utils.is_cpu_cuda.CPUCuda*
method), [87](#)

`use_cuda_if_available()`
(*analogvnn.utils.is_cpu_cuda.CPUCuda*
method), [87](#)

V

`variance` (*analogvnn.nn.noise.GaussianNoise.GaussianNoise*
property), [59](#)

`variance` (*analogvnn.nn.noise.LaplacianNoise.LaplacianNoise*
property), [62](#)

`variance` (*analogvnn.nn.noise.UniformNoise.UniformNoise*
property), [69](#)

W

`weight` (*analogvnn.nn.Linear.Linear attribute*), [79](#)