Development of the Pytorch Neuro Trigger

Tobias Jülg

07.09.2022

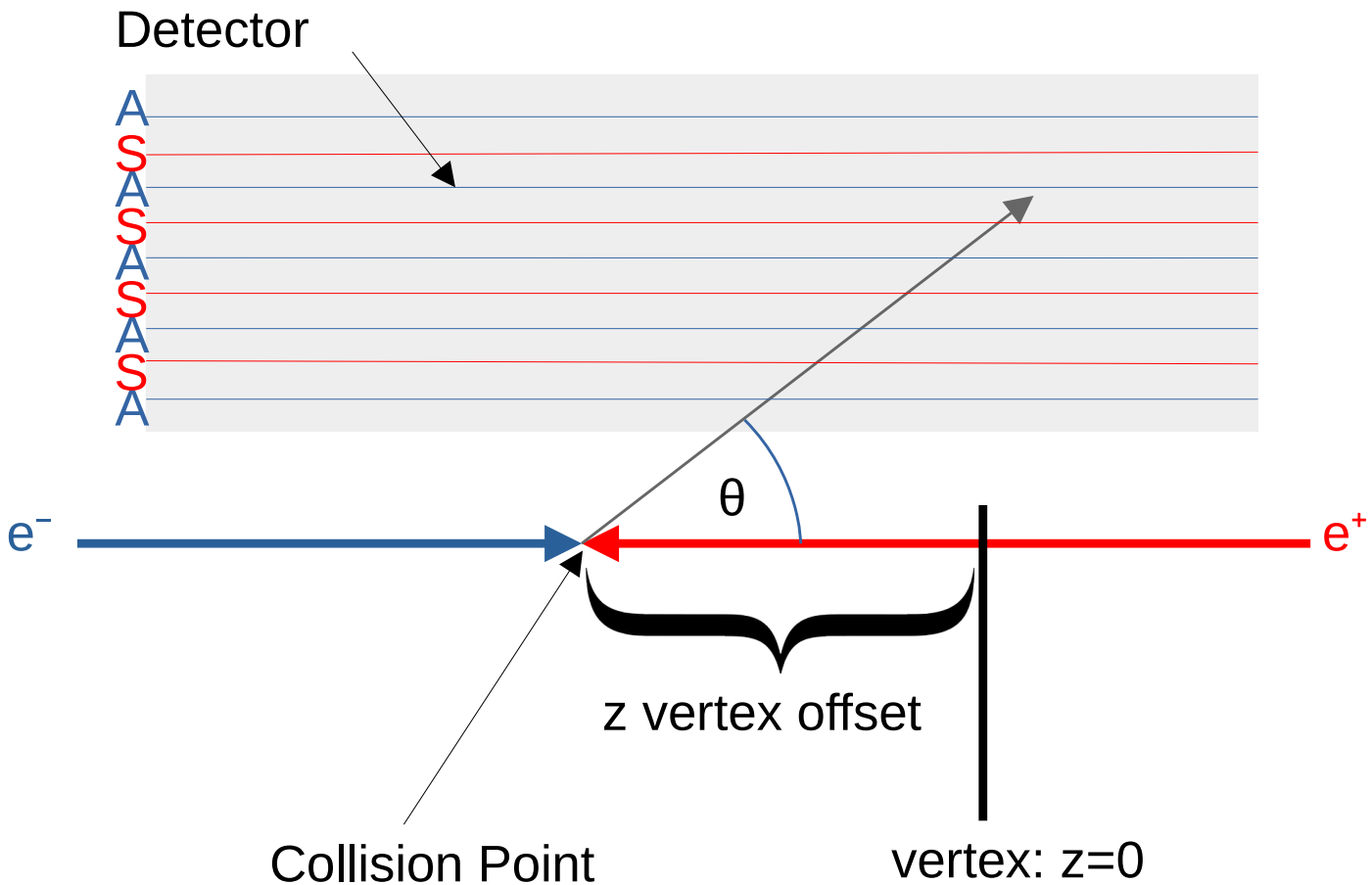
Max Planck Institute for Physics

Outline

Introduction

What do we want to predict?

- z-vertex
- θ angle
- Detector: 5 axial and 4 stereo superlayers



How to predict z and θ?

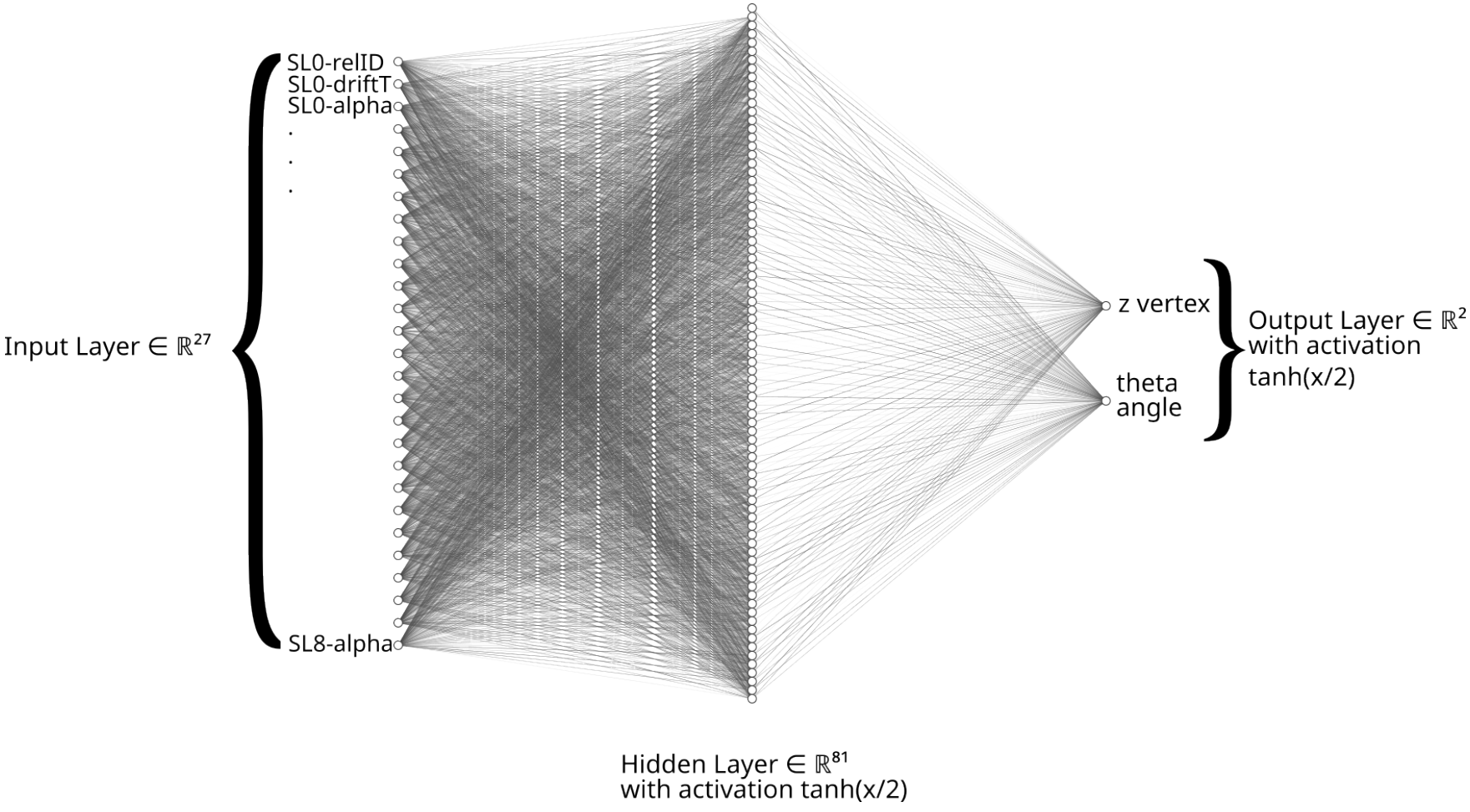
Neural Network for z and θ prediction of the e^+/e^- collision

- Input: 9 layers with
  - `SLx-relID`: Angle relative to 2D track ϕ
  - `SLx-driftT`: Drift time
  - `SLx-alpha`: Crossing angle α

  $\rightarrow \mathbb{R}^{27}$
- Output:
  - Collision's z-vertex
  - θ angle of the resulting partical

  $\rightarrow \mathbb{R}^2$
- One hidden layer with 81 neurons
- $\tanh(\frac{x}{2})$ as activation function

Network Architecture



Motivation

Motivation

Why was it neccassary to replace the old BASF2 FANN network training?

Motivation

Why was it neccassary to replace the old BASF2 FANN network training?

- FANN uses training and optimization algorithms from the 90's

Motivation

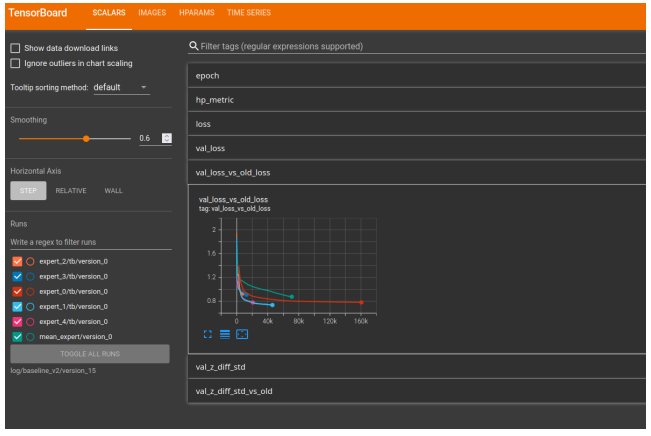
Why was it neccassary to replace the old BASF2 FANN network training?

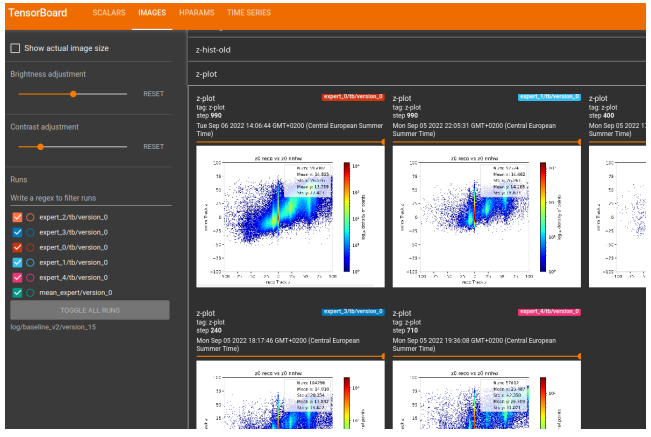
- FANN uses training and optimization algorithms from the 90's
- Improved algorithms in PyTorch
  - Stochastical gradient descent
  - Modern weight optimizer: Adam
  - L2 regularization / weight decay
  - Sophisticated random weight initialization strategies

Motivation

Why was it neccassary to replace the old BASF2 FANN network training?

- FANN uses training and optimization algorithms from the 90's
- Improved algorithms in PyTorch
  - Stochastical gradient descent
  - Modern weight optimizer: Adam
  - L2 regularization / weight decay
  - Sophisticated random weight initialization strategies
- Live graphical training supervision through tensorboard

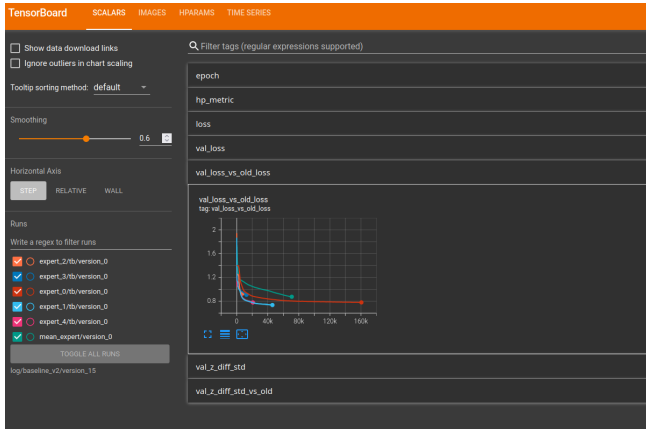


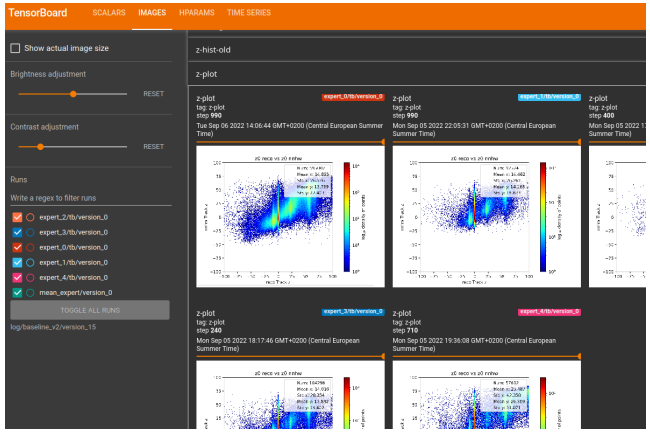


Motivation

Why was it neccassary to replace the old BASF2 FANN network training?

- FANN uses training and optimization algorithms from the 90's
- Improved algorithms in PyTorch
  - Stochastical gradient descent
  - Modern weight optimizer: Adam
  - L2 regularization / weight decay
  - Sophisticated random weight initialization strategies
- Live graphical training supervision through tensorboard
- Modular and decoupled from BASF2

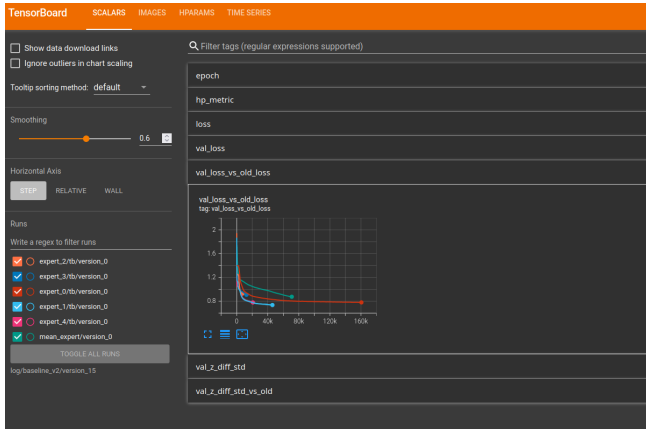


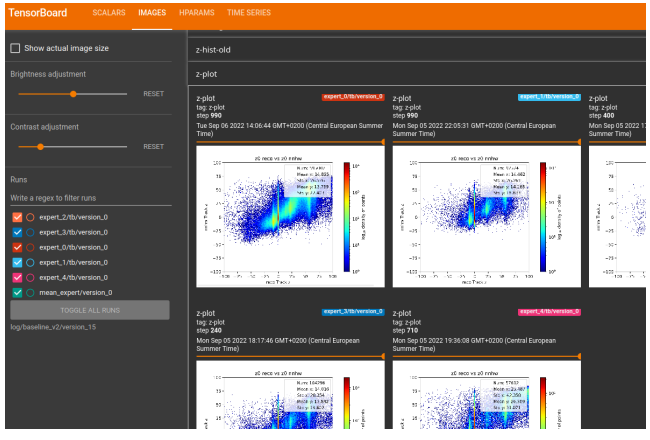


Motivation

Why was it neccassary to replace the old BASF2 FANN network training?

- FANN uses training and optimization algorithms from the 90's
- Improved algorithms in PyTorch
  - Stochastical gradient descent
  - Modern weight optimizer: Adam
  - L2 regularization / weight decay
  - Sophisticated random weight initialization strategies
- Live graphical training supervision through tensorboard
- Modular and decoupled from BASF2
- Flexible configuration which does not require code changes to test different hyperparameter settings

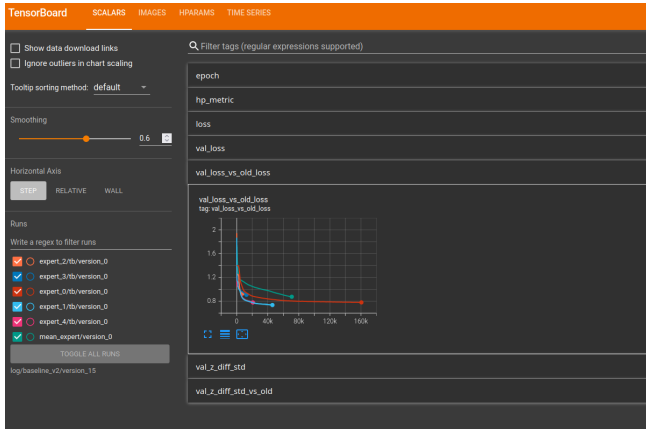


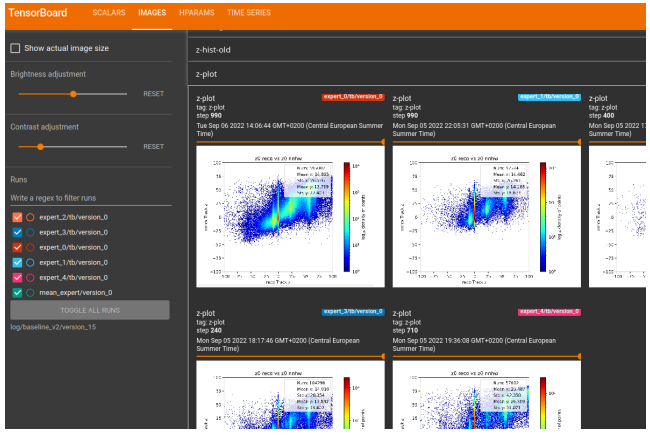


Motivation

Why was it neccassary to replace the old BASF2 FANN network training?

- FANN uses training and optimization algorithms from the 90's
- Improved algorithms in PyTorch
  - Stochastical gradient descent
  - Modern weight optimizer: Adam
  - L2 regularization / weight decay
  - Sophisticated random weight initialization strategies
- Live graphical training supervision through tensorboard
- Modular and decoupled from BASF2
- Flexible configuration which does not require code changes to test different hyperparameter settings
- Reproducability build-in through extensive logging





Configuration and Training

Folder Structure

```
nnt-pytorch
├── changelog.md
├── docs
│   ├── class_diagram.dia
│   ├── class_diagram.png
│   ├── class_diagram.svg
│   ├── norm_inf_bounds.png
│   ├── norm_non_inf_bounds.png
│   ├── uniform.png
│   └── zhist.png
├── README.md
├── requirements.txt
└── neuro_trigger
    ├── configs.py
    └── __init__.py
```

```
neuro_trigger
├── lightning
│   ├── __init__.py
│   ├── mean_tb_logger.py
│   └── pl_module.py
├── main.py
├── pytorch
│   ├── __init__.py
│   ├── dataset_filters.py
│   ├── dataset.py
│   └── model.py
├── tests
│   ├── __init__.py
│   ├── test_data.csv
│   ├── test_data_filter.csv
│   └── test.py
├── utils.py
└── visualize.py
```

Configure a Training

Trainings are configured in a python dictionary in `neuro_trigger/config.py`

- All training hyperparameter can be configured like learning rate, batch size, optimizer, network model, ... [1]
- Supports configurations inheritance
- Global hyperparameters vs per-expert hyperparameters
- Hierarchical: Flexible and extensible scheme

[1] Extensive list can be found in the README.md documentation

Configure a Training: Example

Base config to set default parameters:

```
"base": {
    "version": 1.0,
    "description": "Some desc.",
    "learning_rate": 1e-3,
    "batch_size": 2048,
    "weight_decay": 1e-6,
    "in_size": 27,
    "out_size": 2,
    "workers": 5,
    "epochs": 10,
    "model": "BaselineModel",
    "loss": "MSELoss",
    "optim": "Adam",
    "act": "tanh",
    "experts": [0, 1, 2, 3, 4],
    "compare_to": None,
    "load_pre_trained_weights":
        None,
}
```

Configure a Training: Example

Base config to set default parameters:

```
"base": {
    "version": 1.0,
    "description": "Some desc.",
    "learning_rate": 1e-3,
    "batch_size": 2048,
    "weight_decay": 1e-6,
    "in_size": 27,
    "out_size": 2,
    "workers": 5,
    "epochs": 10,
    "model": "BaselineModel",
    "loss": "MSELoss",
    "optim": "Adam",
    "act": "tanh",
    "experts": [0, 1, 2, 3, 4],
    "compare_to": None,
    "load_pre_trained_weights":
        None,
}
```

Advanced config which inherits from base:

```
"my_base_extension": {
    "extends": "base",
    "description": "My ext.",
    "act": "tanh/2",
    "epochs": 1000,
    "expert_3": {
        "batch_size": 128,
        "epochs": 2000,
    },
    "expert_4": {
        "batch_size": 16,
        "epochs": 4000,
    },
    "compare_to":
        "base/version_4",
}
```

Training CLI

To start a training the `neuro_trigger/main.py` CLI has to be used:

```
usage: main.py [-h] [-p] [-s] mode

Tool to start the neuro trigger training.

positional arguments:
  mode                  Config mode to use, must be defined
                        in config.py

optional arguments:
  -h, --help            show this help message and exit
  -p, --production      If the flag is not given code will
                        run in debug mode otherwise production
                            mode
  -s, --solo_expert     Whether one or several experts are used
                        for training
```

Start Training and Tensorboard

- Start a production training

```
source venv/bin/activate
export PYTHONPATH="$PYTHONPATH:."
python neuro_trigger/main.py my_base_extension -p
```

Start Training and Tensorboard

- Start a production training

```
source venv/bin/activate
export PYTHONPATH="$PYTHONPATH:."
python neuro_trigger/main.py my_base_extension -p
```

- View the log path

```
cat log/log.txt
```

Start Training and Tensorboard

- Start a production training

```
source venv/bin/activate
export PYTHONPATH="$PYTHONPATH:."
python neuro_trigger/main.py my_base_extension -p
```

- View the log path

```
cat log/log.txt
```

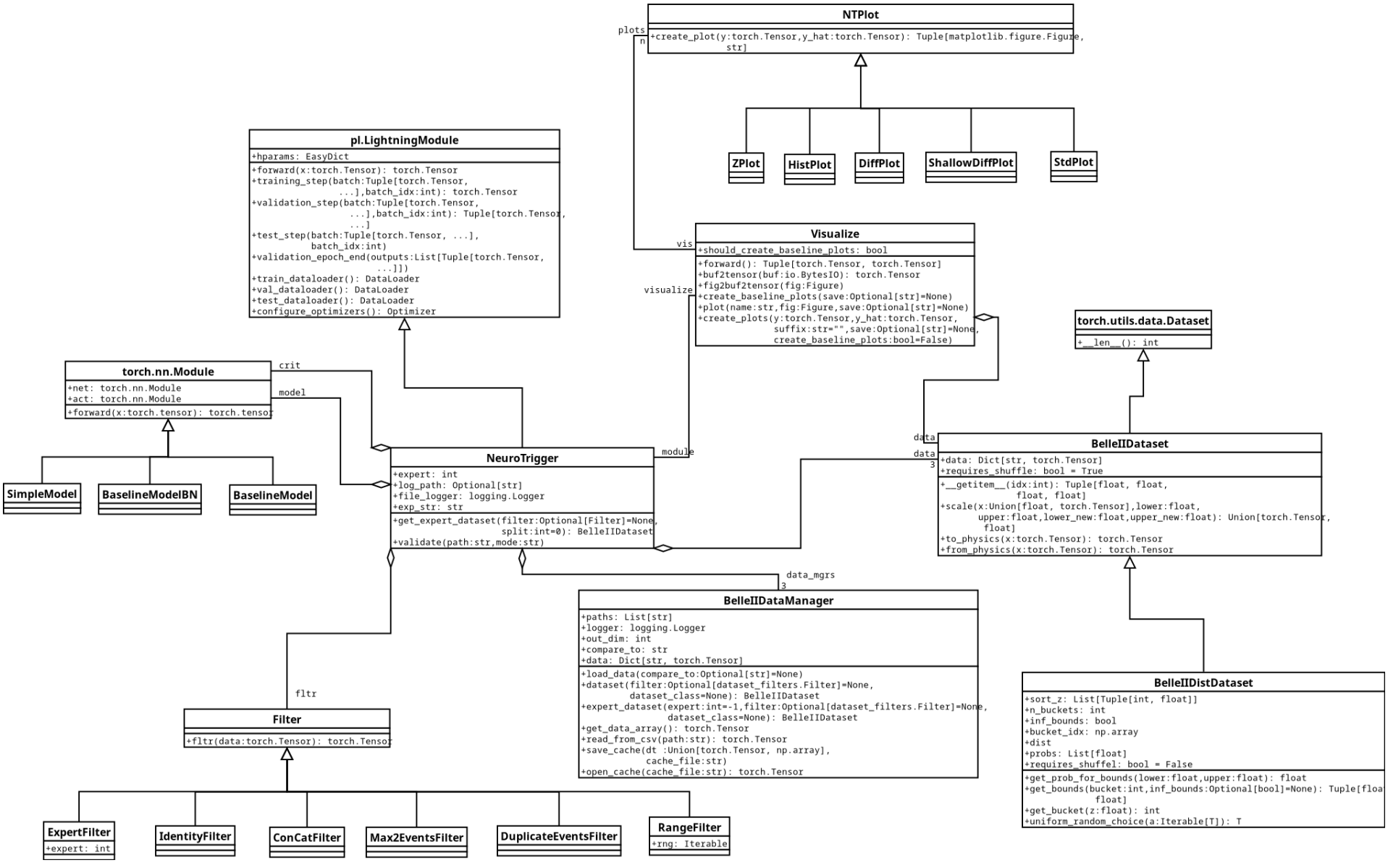
- Start Tensorboard

```
tensorboard --logdir=log/my_base_extension/
    version_0
```

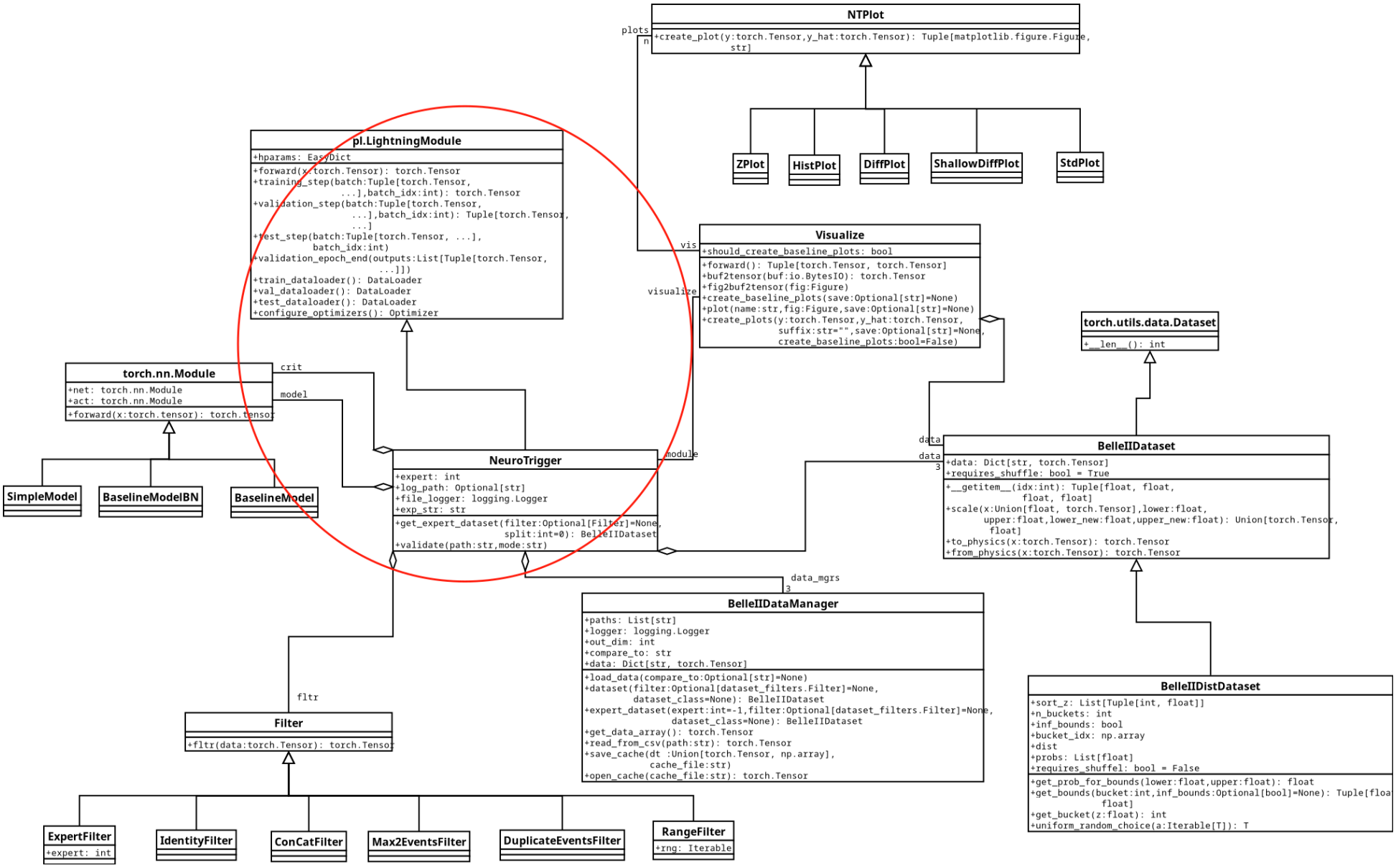
Live Demo!

Extensibility

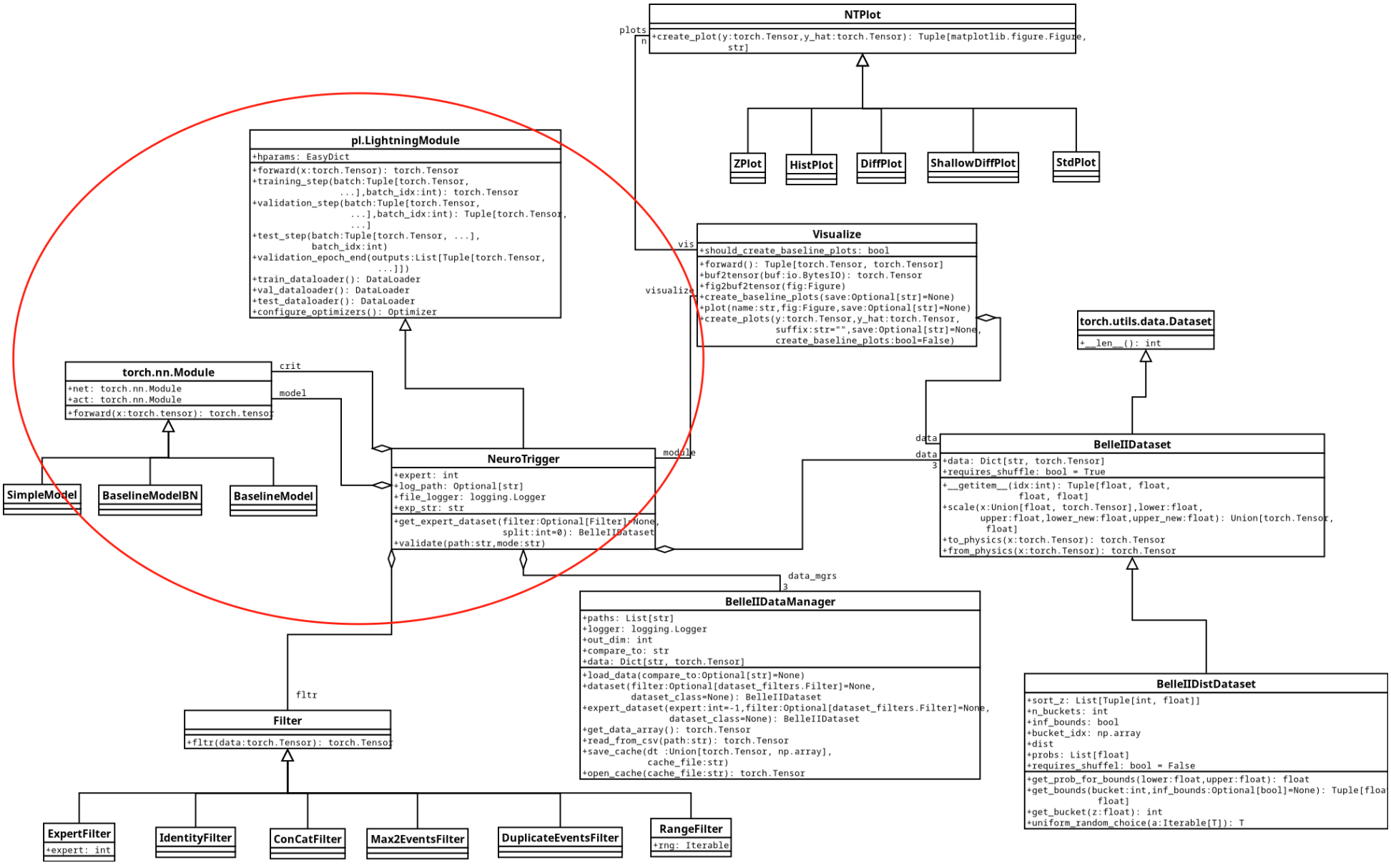
Software Architecture



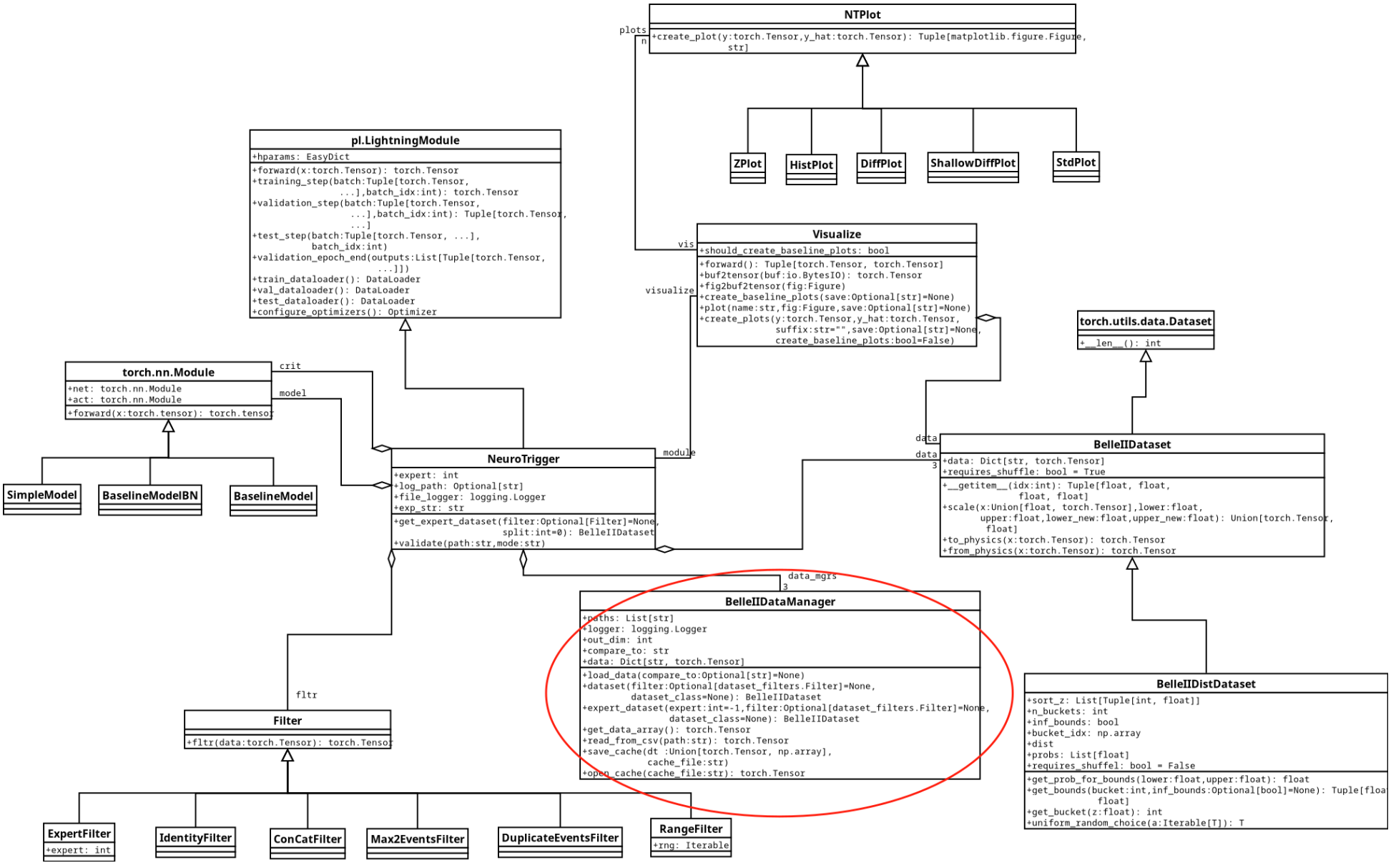
Software Architecture



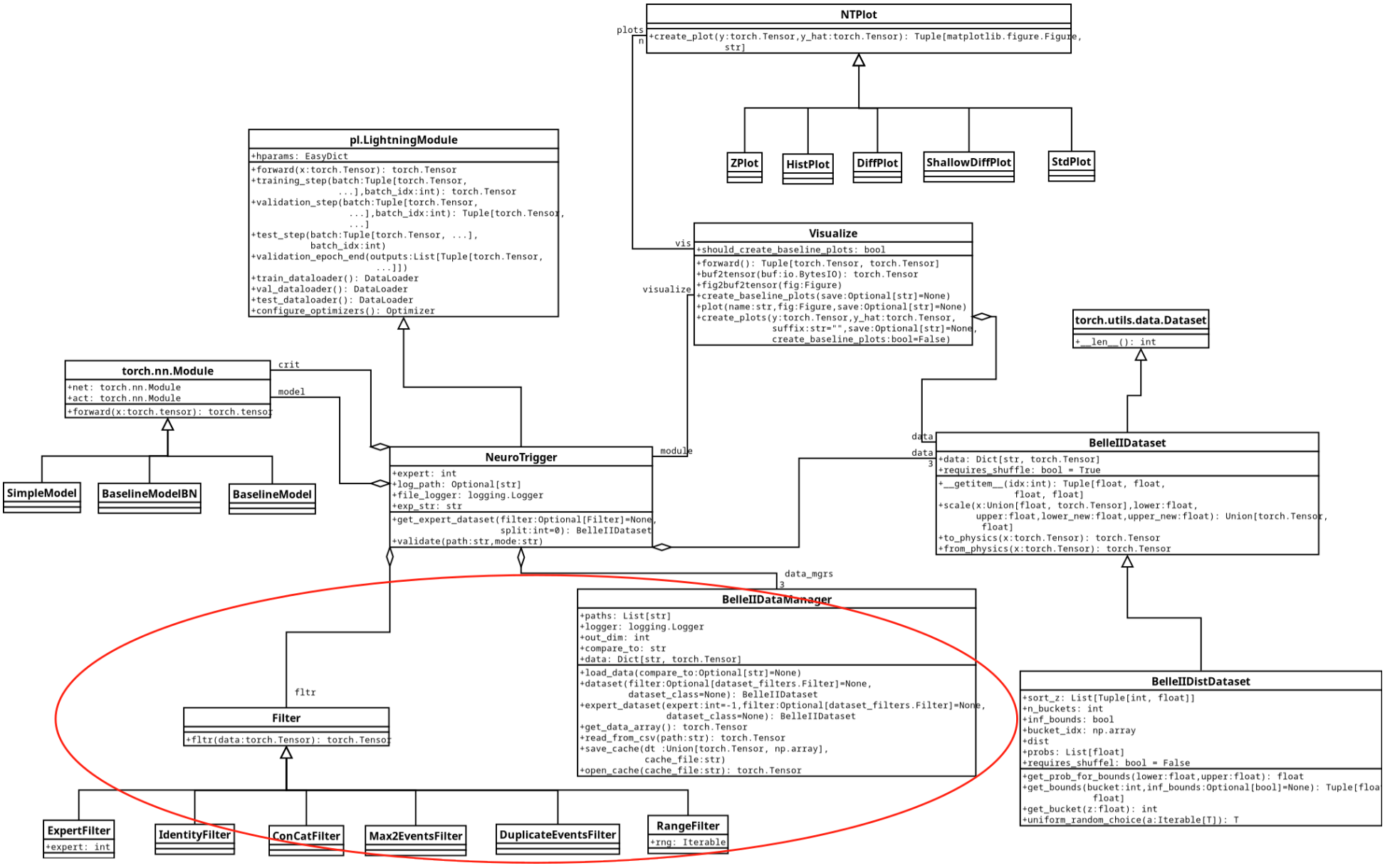
Software Architecture



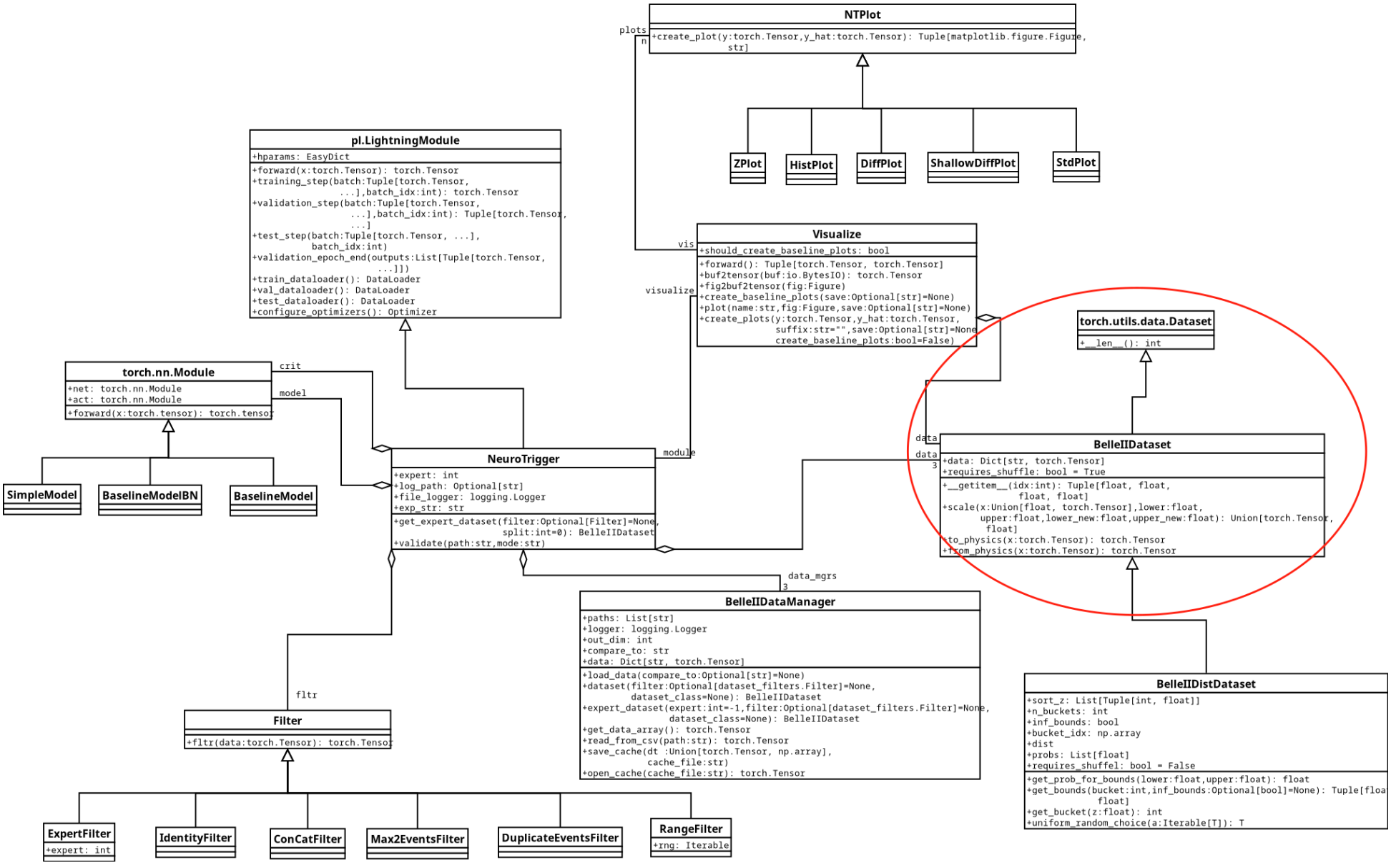
Software Architecture



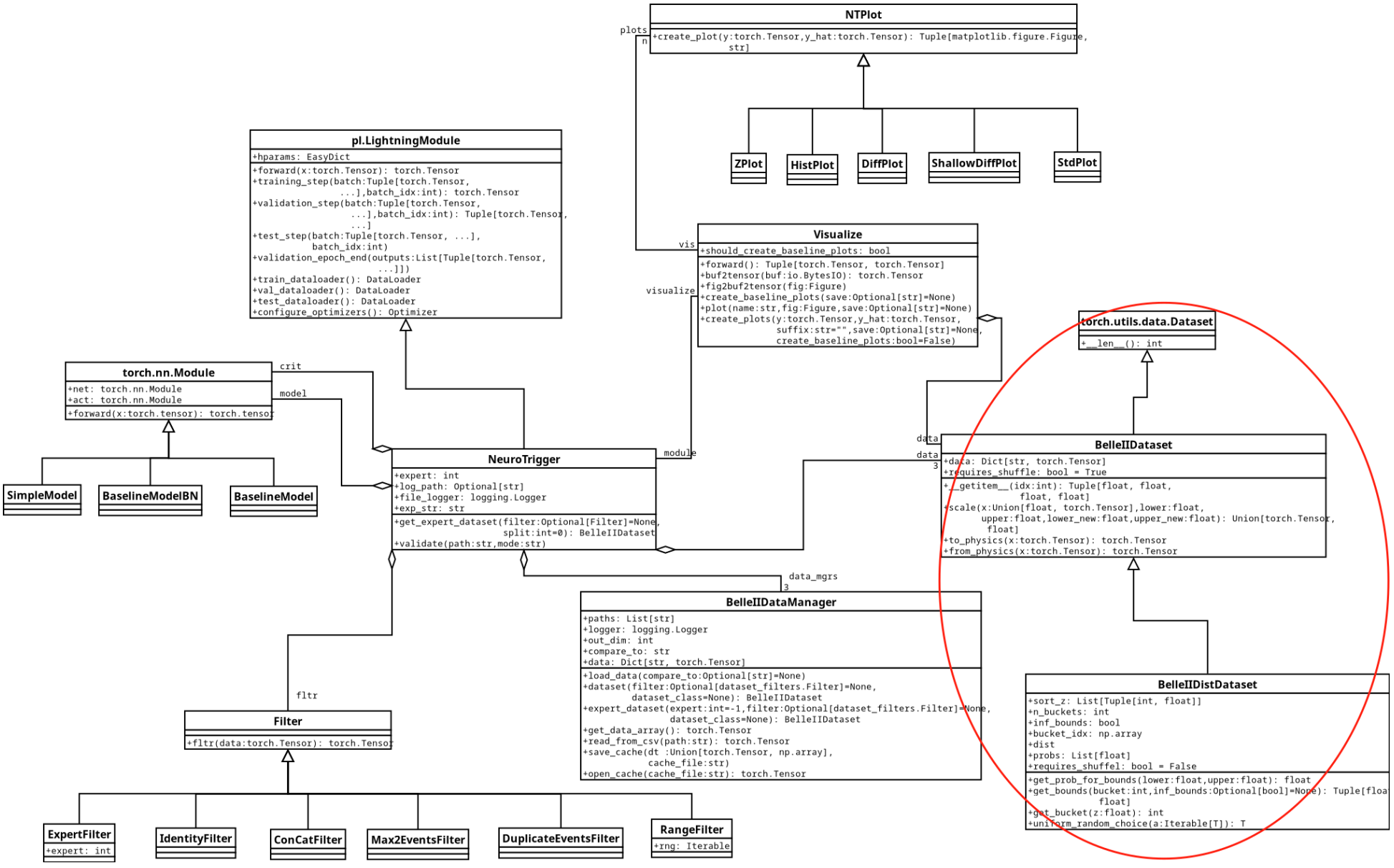
Software Architecture



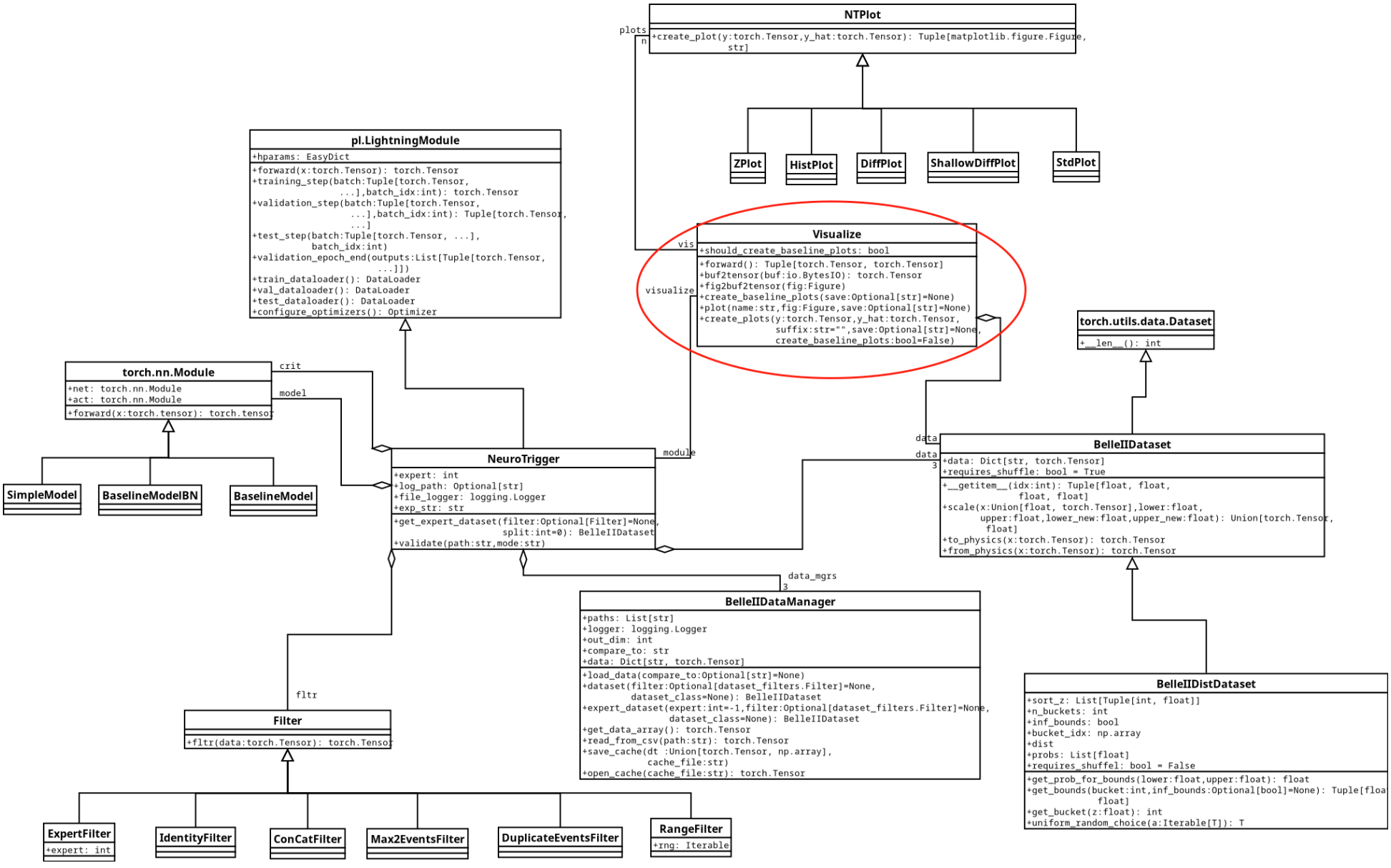
Software Architecture



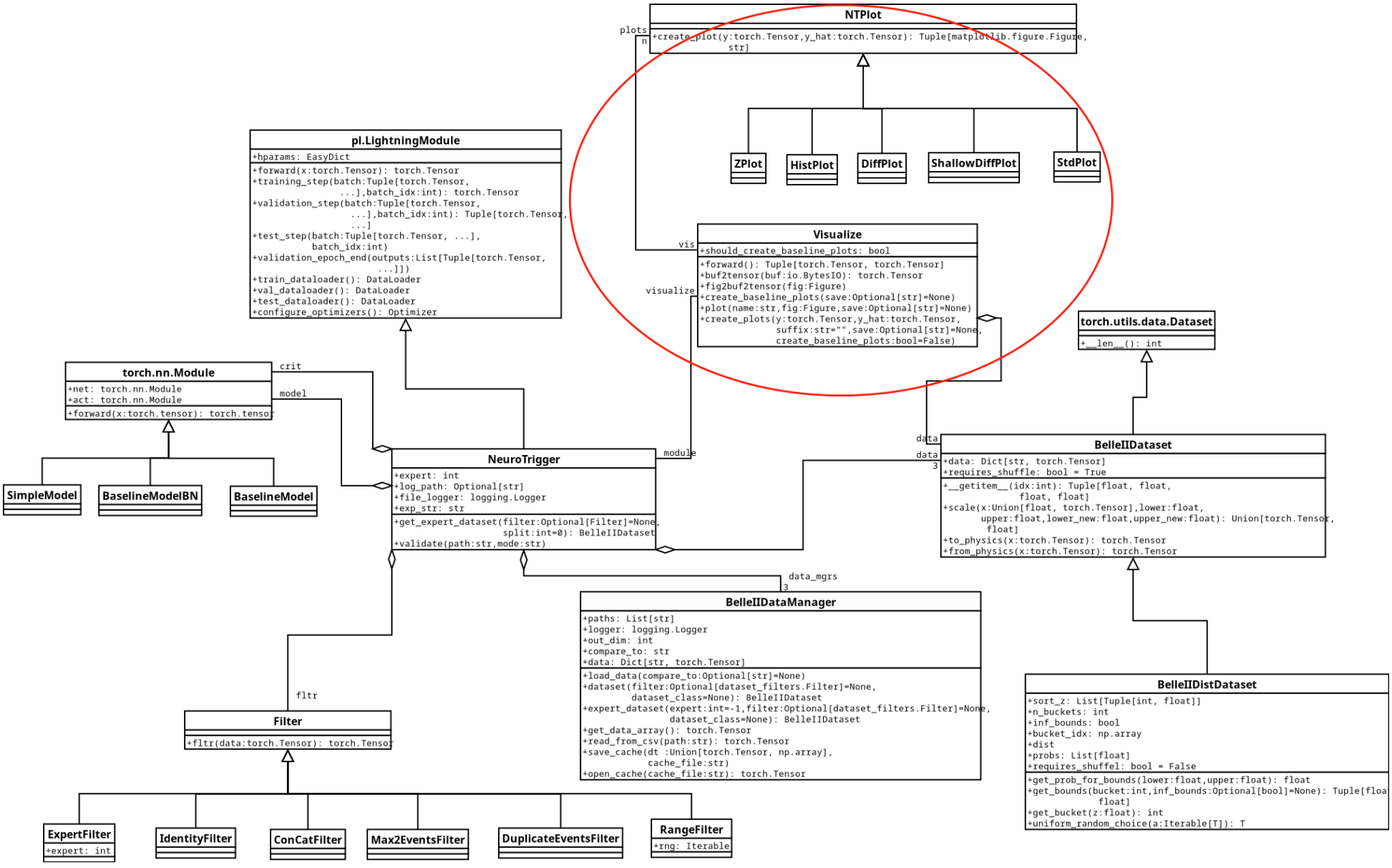
Software Architecture



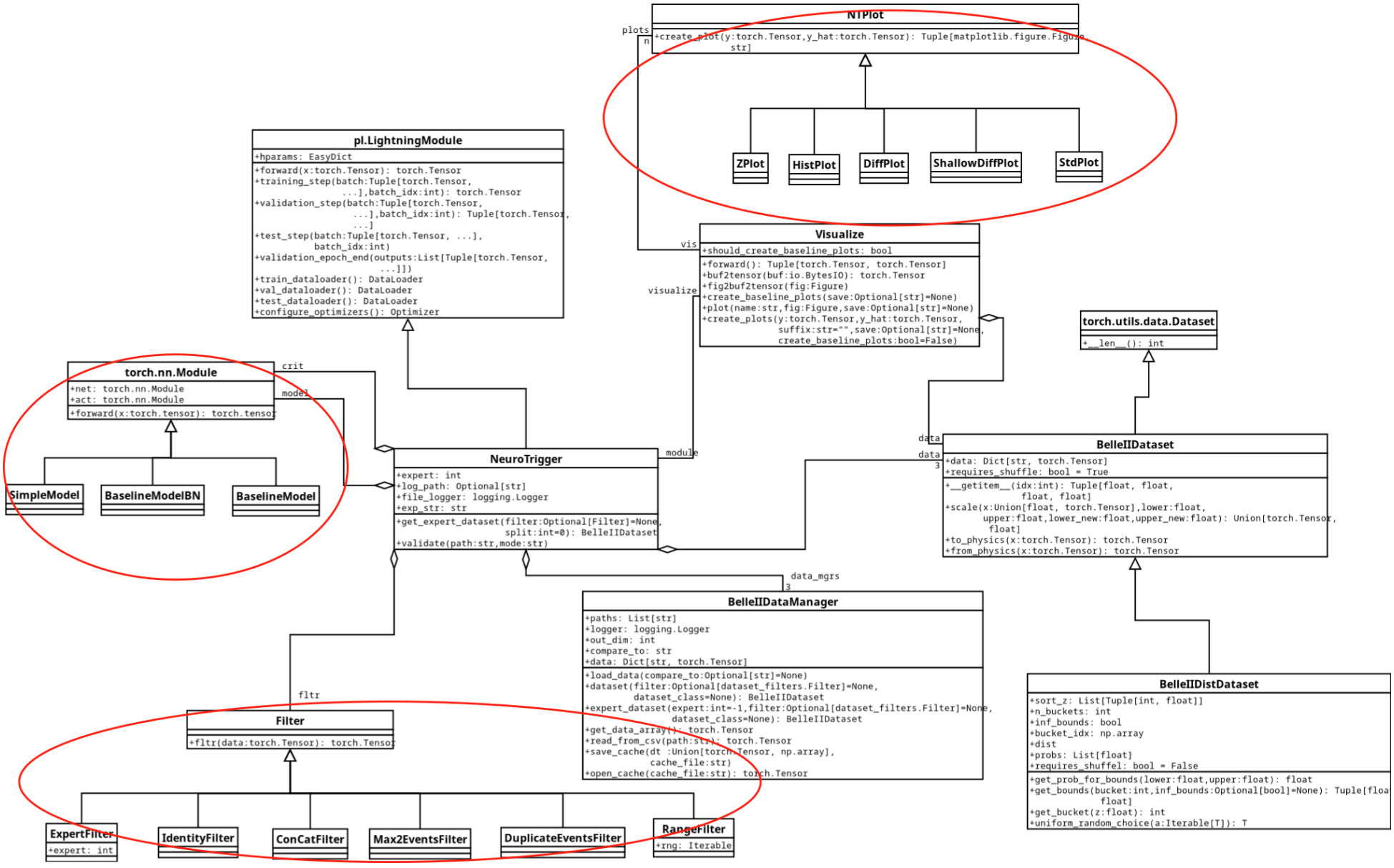
Software Architecture



Software Architecture

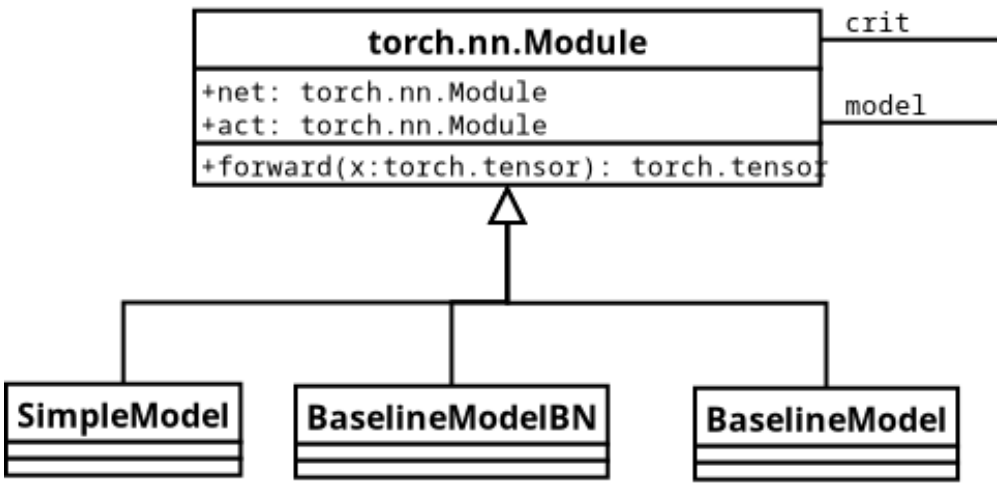


Software Architecture



New Network Models

- Network models are defined using PyTorch Modules
- Expected contstructor arguments:
  - `inp`: input vector size, usally 27
  - `out`: output vector size, 2 for z and θ, 1 for only z
  - `act`: Activation function (PyTorch Module)
- PyTorch Module needs to implement the `forward` function



New Network Models II

Example: Current two layer FC network

```
class BaselineModel(nn.Module):
```

New Network Models II

Example: Current two layer FC network

```
class BaselineModel(nn.Module):
    def __init__(self,
                 inp: int = 27,
                 out: int = 2,
                 act: Optional[nn.Module]=None):
```

New Network Models II

Example: Current two layer FC network

```
class BaselineModel(nn.Module):
    def __init__(self,
                 inp: int = 27,
                 out: int = 2,
                 act: Optional[nn.Module]=None):
        super().__init__()
        act = act or nn.Tanh()
        self.net = nn.Sequential(
            nn.Linear(inp, 81),
            act,
            nn.Linear(81, out),
            act,
        )
```

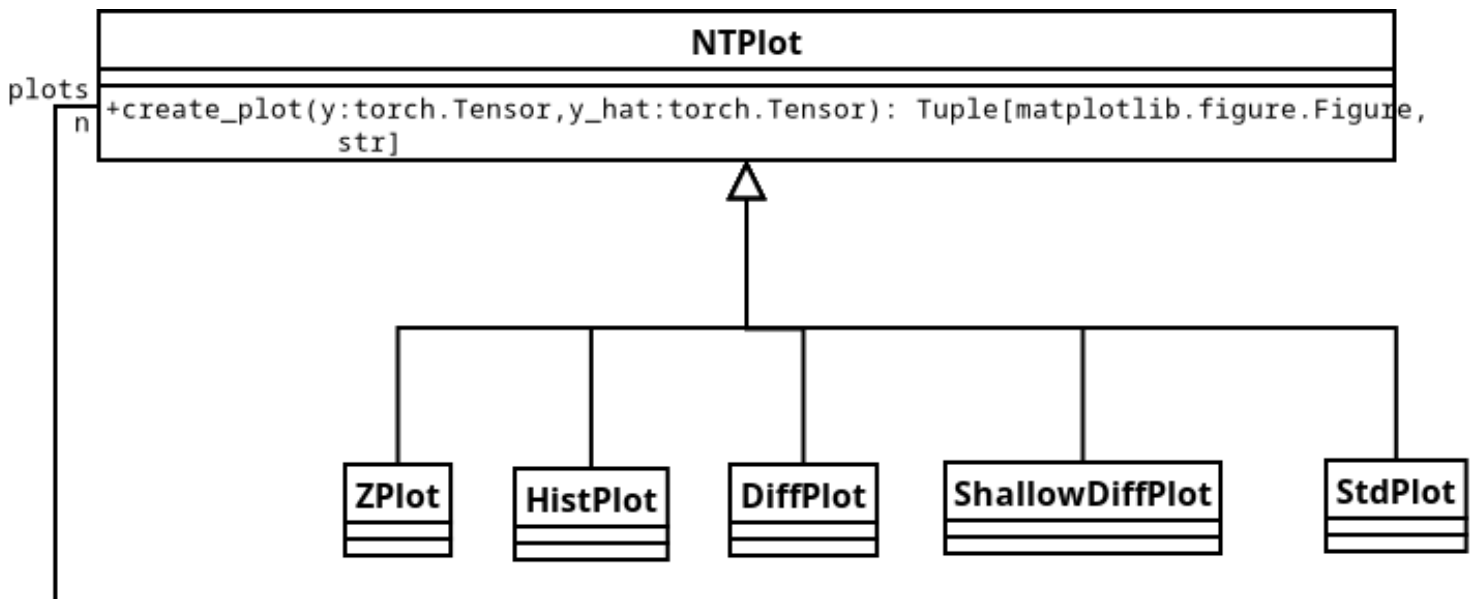
New Network Models II

Example: Current two layer FC network

```
class BaselineModel(nn.Module):
    def __init__(self,
                 inp: int = 27,
                 out: int = 2,
                 act: Optional[nn.Module]=None):
        super().__init__()
        act = act or nn.Tanh()
        self.net = nn.Sequential(
            nn.Linear(inp, 81),
            act,
            nn.Linear(81, out),
            act,
        )
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.net(x)
```

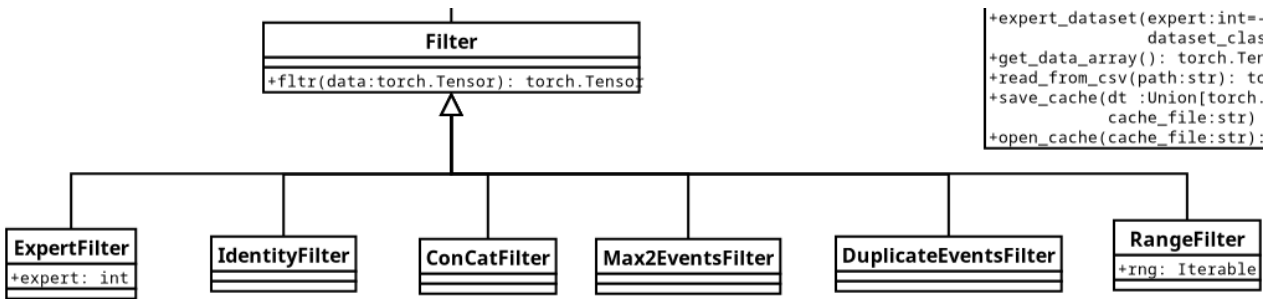
New Plots and Graphs

- Subclass from `visualize.NTPlot`
- Override the `create_plot` method
  - Gets prediction and ground truth (neuro and reco z)
  - Must return a matplotlib figure and a figure name



New Dataset filters

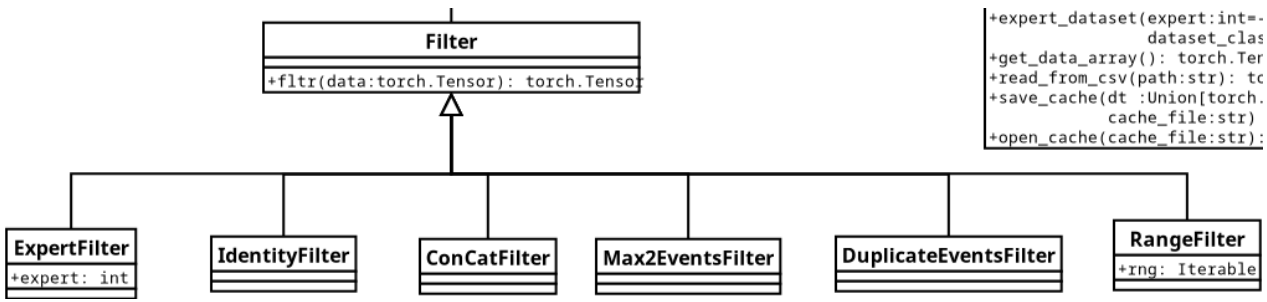
- Subclass from `dataset_filters.Filter`
- Override the `fltr` method
  - Gets data dictionary[2]
  - Must return boolean array with entries set to one that should be kept



[2] Keys and dimensions: `x` $\in \mathbb{R}^{N\times 27}$, `y` $\in \mathbb{R}^{N\times 2}$, `expert` $\in \mathbb{R}^{N}$, `y_hat_old` $\in \mathbb{R}^{N}$, ...

New Dataset filters

- Subclass from `dataset_filters.Filter`
- Override the `fltr` method
  - Gets data dictionary[2]
  - Must return boolean array with entries set to one that should be kept

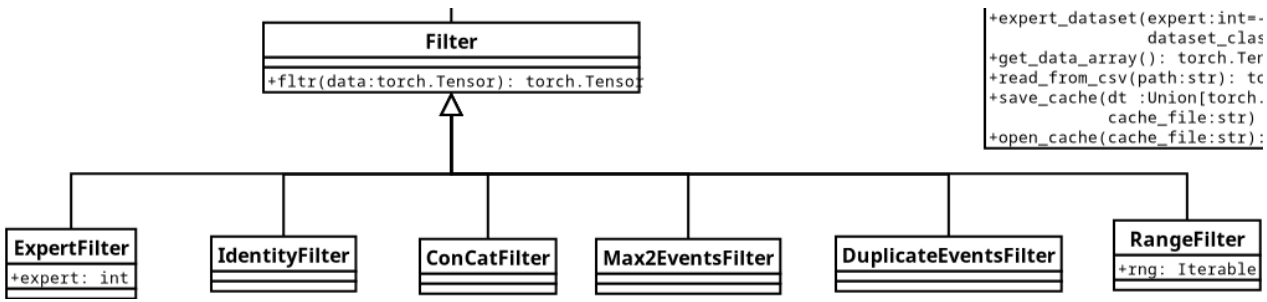


Example: Keep only positive z values

[2] Keys and dimensions: $\mathtt{x} \in \mathbb{R}^{N \times 27}$, $\mathtt{y} \in \mathbb{R}^{N \times 2}$, $\mathtt{expert} \in \mathbb{R}^{N}$, $\mathtt{y\_hat\_old} \in \mathbb{R}^{N}$, ...

New Dataset filters

- Subclass from `dataset_filters.Filter`
- Override the `fltr` method
  - Gets data dictionary[2]
  - Must return boolean array with entries set to one that should be kept



Example: Keep only positive z values

```
class PositiveZFilter(Filter):
    def fltr(self, data: Dict[str, torch.Tensor]) -> torch.Tensor:
        z_data = data['y'][:,0]
        return z_data > 0
```

[2] Keys and dimensions: `x` $\in \mathbb{R}^{N\times 27}$, `y` $\in \mathbb{R}^{N\times 2}$, `expert` $\in \mathbb{R}^{N}$, `y_hat_old` $\in \mathbb{R}^{N}$, ...

Dataset filters in Configuration

Use the new filter in your config (filter needs to be defined in `dataset_filters.py`):

```
"some_config": {
    # ...
    "filter": "dataset_filters.PositiveZFilter()",
}
```

Dataset filters in Configuration

Use the new filter in your config (filter needs to be defined in `dataset_filters.py`):

```
"some_config": {
    # ...
    "filter": "dataset_filters.PositiveZFilter()",
}
```

Combine several filters with the `ConCatFilter`:

```
"some_config": {
    # ...
    "filter": "dataset_filters.ConCatFilter([
                    dataset_filters.PositiveZFilter(),
                    dataset_filters.RangeFilter(range(100))
                ])",
}
```

Dataset filters in Configuration

Use the new filter in your config (filter needs to be defined in `dataset_filters.py`):

```
"some_config": {
    # ...
    "filter": "dataset_filters.PositiveZFilter()",
}
```

Combine several filters with the `ConCatFilter`:

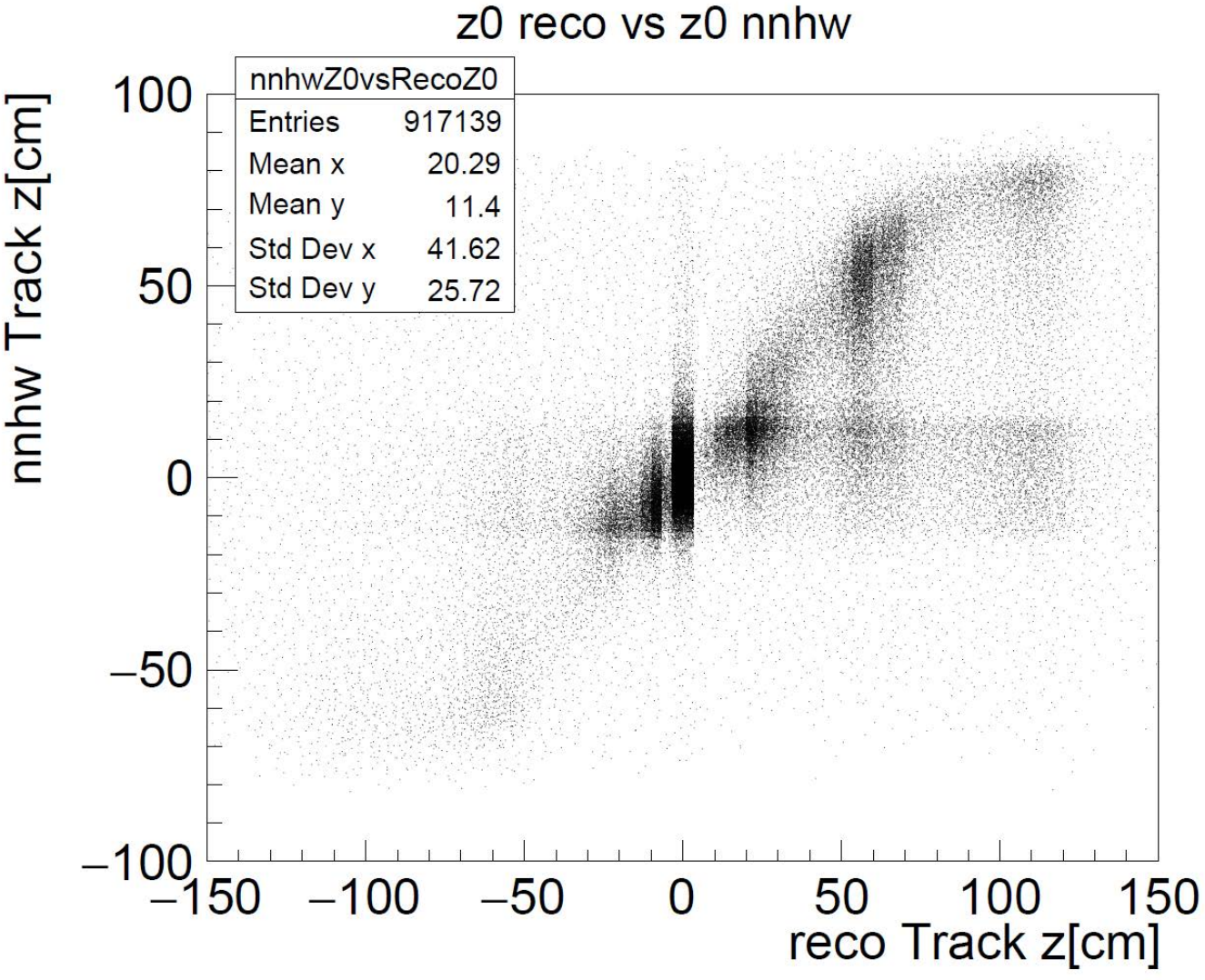
```
"some_config": {
    # ...
    "filter": "dataset_filters.ConCatFilter([
                    dataset_filters.PositiveZFilter(),
                    dataset_filters.RangeFilter(range(100))
                ])",
}
```

$\rightarrow$ Only positive z values **and** only the first one hundred samples

Feed-down Problem and Reweighting

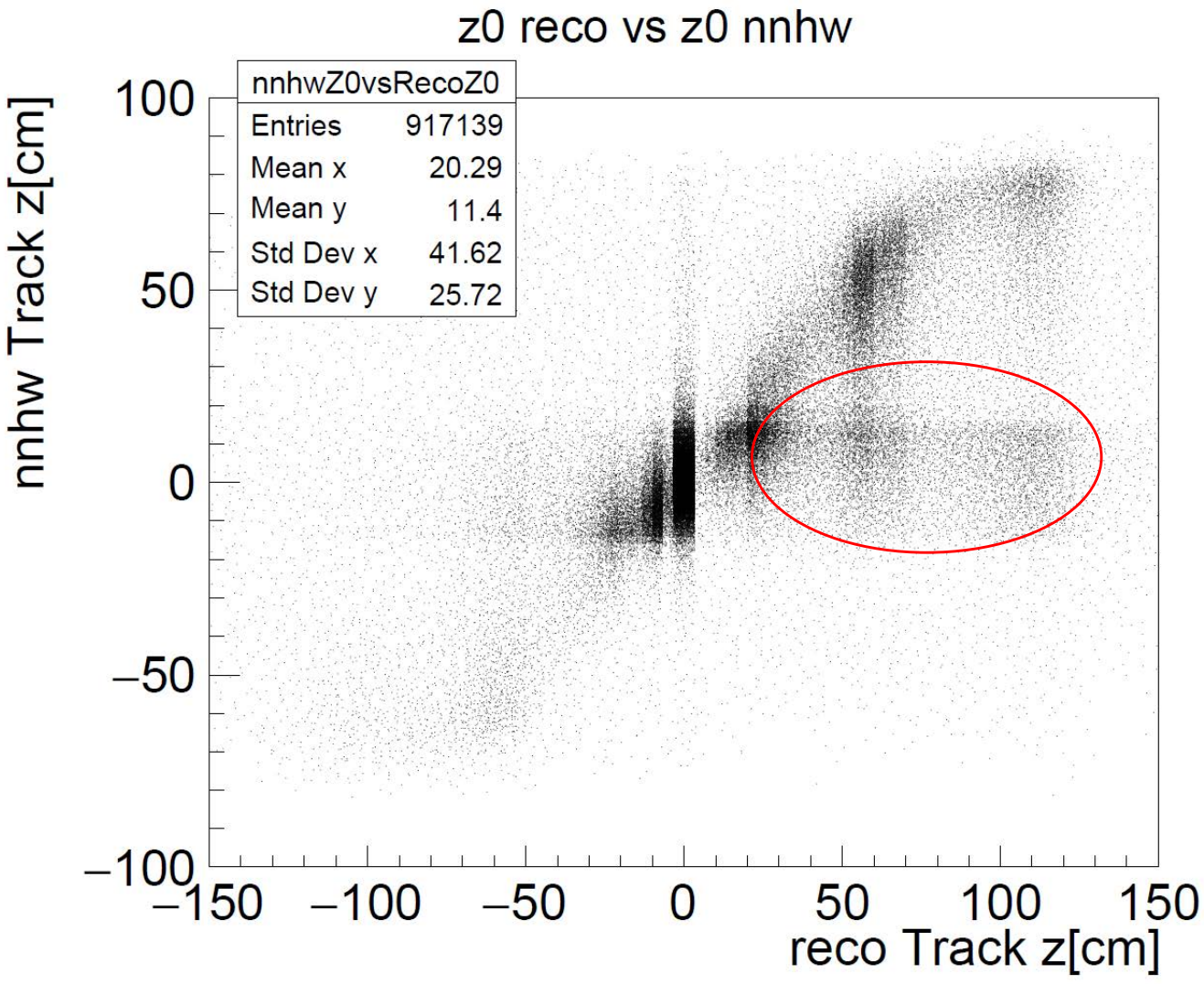
The Feed-Down Problem

Feed-Down in the pre-PyTorch training:



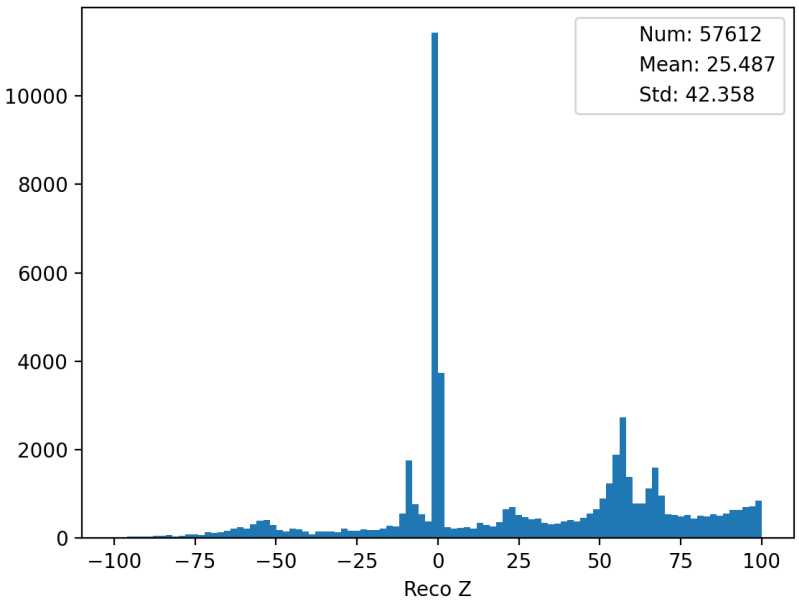
The Feed-Down Problem

Feed-Down in the pre-PyTorch training:



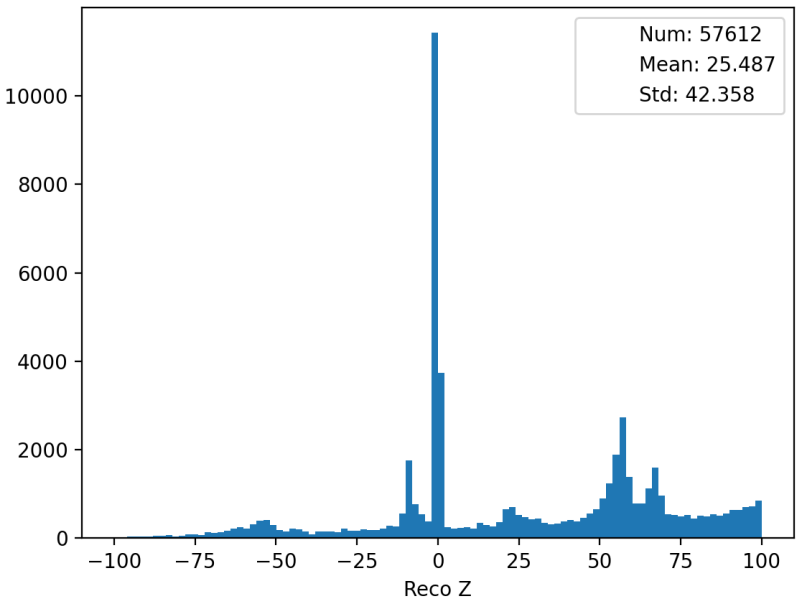
Reweighting: Root Problem & Idea

- A lot of data at $\hat{z} = 0$



Reweighting: Root Problem & Idea

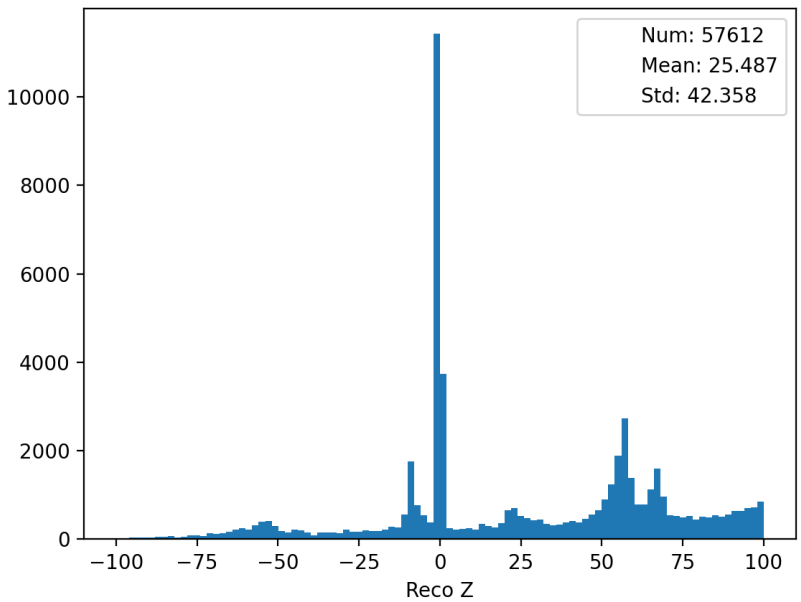
- A lot of data at $\hat{z} = 0$
- Bias towards predicting $z = 0$



Reweighting: Root Problem & Idea

- A lot of data at $\hat{z} = 0$
- Bias towards predicting $z = 0$
- Idea: Present the network with more data of the edge regions by duplicating them

$\rightarrow$ Upsampling



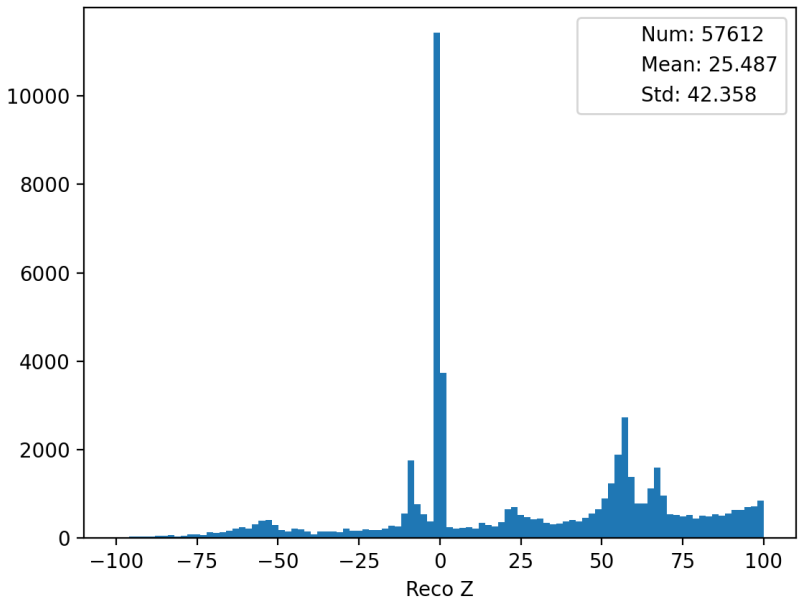
Reweighting: Root Problem & Idea

- A lot of data at $\hat{z} = 0$
- Bias towards predicting $z = 0$
- Idea: Present the network with more data of the edge regions by duplicating them

$\rightarrow$ Upsampling

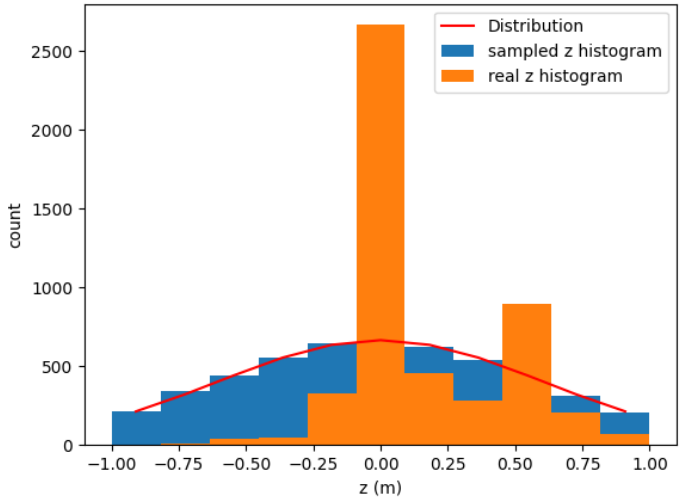
- Try to keep a similar distribution and avoid uniform distribution

$\rightarrow$ Area with more data is also more important



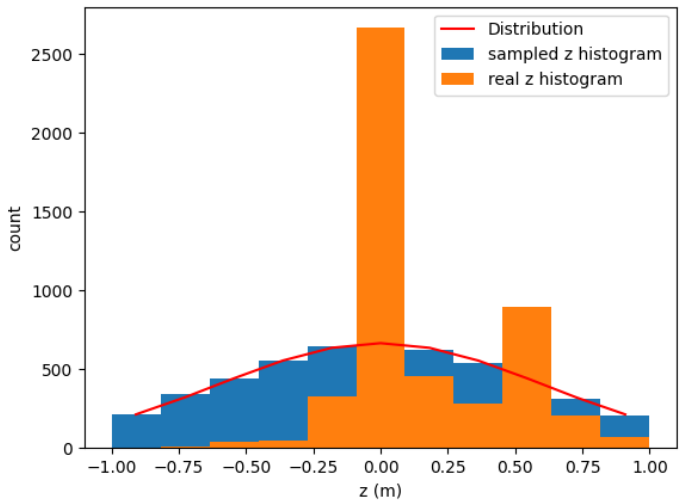
Reweighting: Algorithm

- Choose a continus probability density function $f(x)$ over z
  e.g. a normal distribution with certain mean and variance



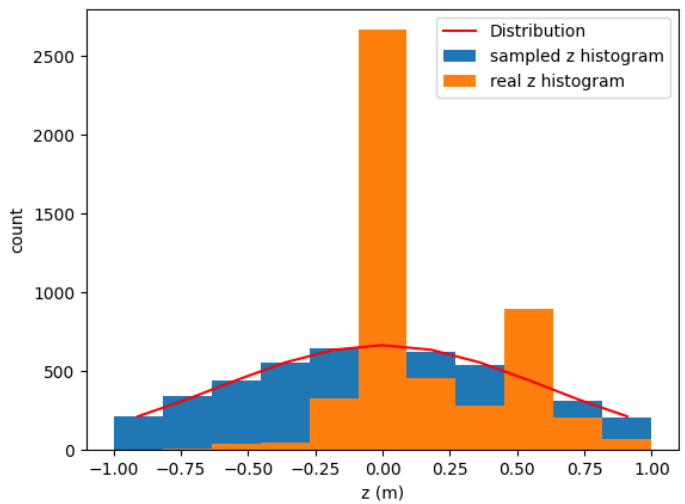
Reweighting: Algorithm

- Choose a continus probability density function $f(x)$ over z
  e.g. a normal distribution with certain mean and variance
- Divide the z range into n equally size buckets $b_i = [b_{i_l}, b_{i_u}] \; \forall i \in [n]$



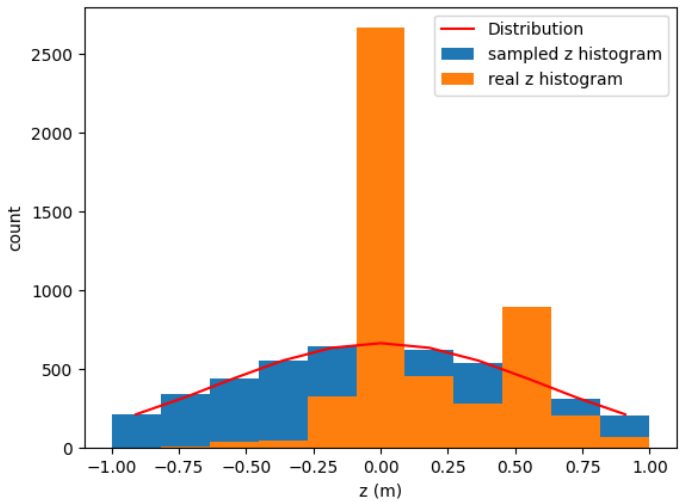
Reweighting: Algorithm

- Choose a continus probability density function $f(x)$ over z
  e.g. a normal distribution with certain mean and variance
- Divide the z range into n equally size buckets $b_i = [b_{i_l}, b_{i_u}] \; \forall i \in [n]$
- Bucket probability given by CDF: $p_i = \int_{b_{i_l}}^{b_{i_u}} f(x)dx$



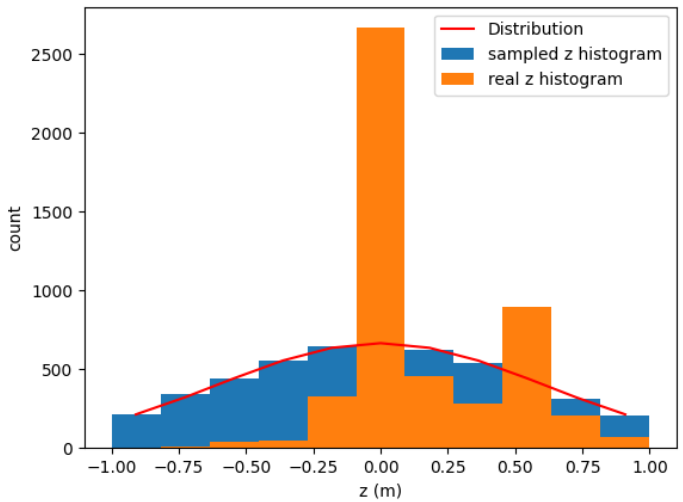
Reweighting: Algorithm

- Choose a continus probability density function $f(x)$ over z
  e.g. a normal distribution with certain mean and variance
- Divide the z range into n equally size buckets $b_i = [b_{i_l}, b_{i_u}] \; \forall i \in [n]$
- Bucket probability given by CDF: $p_i = \int_{b_{i_l}}^{b_{i_u}} f(x)dx$
- To sample with an item:



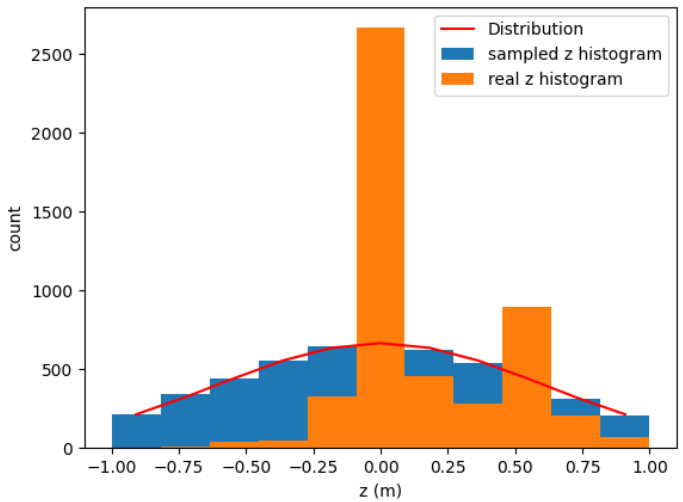
Reweighting: Algorithm

- Choose a continus probability density function $f(x)$ over z
  e.g. a normal distribution with certain mean and variance
- Divide the z range into n equally size buckets $b_i = [b_{i_l}, b_{i_u}] \; \forall i \in [n]$
- Bucket probability given by CDF: $p_i = \int_{b_{i_l}}^{b_{i_u}} f(x)dx$
- To sample with an item:
  - Pick a according the probabilities p_i



Reweighting: Algorithm

- Choose a continus probability density function $f(x)$ over z
  e.g. a normal distribution with certain mean and variance
- Divide the z range into n equally size buckets $b_i = [b_{i_l}, b_{i_u}] \; \forall i \in [n]$
- Bucket probability given by CDF: $p_i = \int_{b_{i_l}}^{b_{i_u}} f(x)dx$
- To sample with an item:
  - Pick a according the probabilities p_i
  - Within the bucket, sample uniform randomly



Reweighting: Configuration

- Reweightings are configured with a hierarchical syntax
- Example:
  - $n = 11$ buckets
  - Normal distribution with $\mu = 0$ and $\sigma = 0.4$

Reweighting: Configuration

- Reweightings are configured with a hierarchical syntax
- Example:
  - $n = 11$ buckets
  - Normal distribution with $\mu = 0$ and $\sigma = 0.4$

```
"some_config": {
    # ...




}
```

Reweighting: Configuration

- Reweightings are configured with a hierarchical syntax
- Example:
  - $n = 11$ buckets
  - Normal distribution with $\mu = 0$ and $\sigma = 0.4$

```
"some_config": {
    # ...
    "dist": {




    },
}
```

Reweighting: Configuration

- Reweightings are configured with a hierarchical syntax
- Example:
  - $n = 11$ buckets
  - Normal distribution with $\mu = 0$ and $\sigma = 0.4$

```
"some_config": {
    # ...
    "dist": {
        "n_buckets": 11,




    },
}
```

Reweighting: Configuration

- Reweightings are configured with a hierarchical syntax
- Example:
  - $n = 11$ buckets
  - Normal distribution with $\mu = 0$ and $\sigma = 0.4$

```
"some_config": {
    # ...
    "dist": {
        "n_buckets": 11,
        "norm": {
            "mean": 0,
            "std": 0.4,
        },

    },
}
```

Reweighting: Configuration

- Reweightings are configured with a hierarchical syntax
- Example:
  - $n = 11$ buckets
  - Normal distribution with $\mu = 0$ and $\sigma = 0.4$

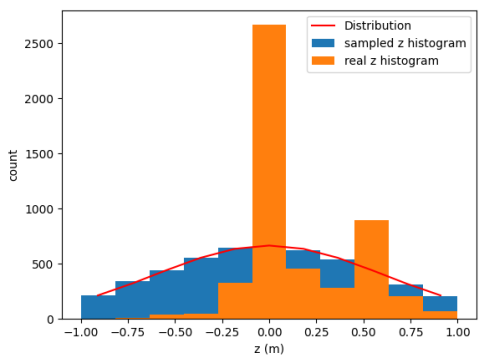
```
"some_config": {
    # ...
    "dist": {
        "n_buckets": 11,
        "norm": {
            "mean": 0,
            "std": 0.4,
        },
        "inf_bounds": False,
    },
}
```

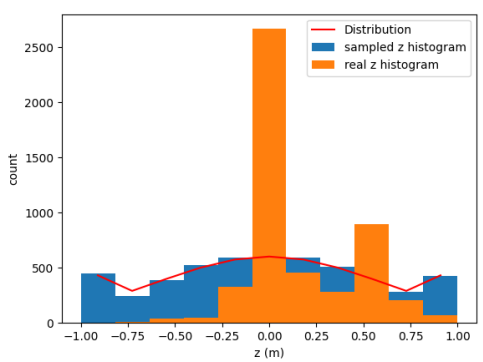
Reweighting: Configuration

- Reweightings are configured with a hierarchical syntax
- Example:
  - $n = 11$ buckets
  - Normal distribution with $\mu = 0$ and $\sigma = 0.4$

```
"some_config": {
    # ...
    "dist": {
        "n_buckets": 11,
        "norm": {
            "mean": 0,
            "std": 0.4,
        },
        "inf_bounds": False,
    },
}
```

Infinite bounds vs non finite bounds:





Results

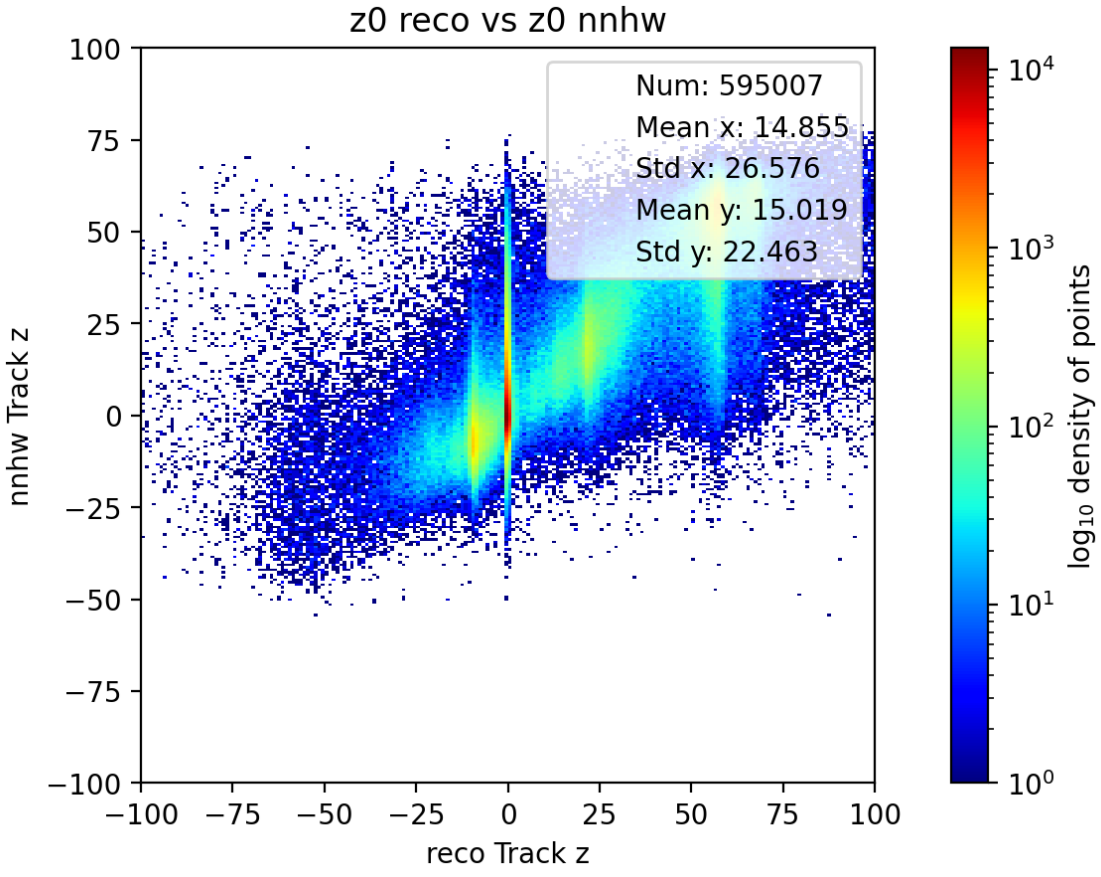
Numeric Results

Table 1: Validation loss vs Old loss

	Baseline	Tanh	Tanh BN	Softsign	Reweight	Events Filter
Expert 0	0.766	0.742	0.709	0.737	1.159	1.016
Expert 1	0.736	0.703	0.696	0.710	1.074	1.047
Expert 2	0.928	0.851	0.830	0.831	1.475	1.406
Expert 3	0.908	0.780	0.760	0.770	1.559	1.067
Expert 4	0.779	0.784	0.683	0.692	1.256	1.259
Mean	0.823	0.772	0.735	0.748	1.305	1.159

Z Plot

Neuro z vs reco z for expert 0 of the current Trigger:



Conclusion

- Interestingly, the first PyTorch training gave the highest gain, further optimization with regards to the feed-down problem even worsened outcome

Conclusion

- Interestingly, the first PyTorch training gave the highest gain, further optimization with regards to the feed-down problem even worsened outcome

$\rightarrow$ Most gain through modern training techniques

Conclusion

- Interestingly, the first PyTorch training gave the highest gain, further optimization with regards to the feed-down problem even worsened outcome

$\rightarrow$ Most gain through modern training techniques

- Further Improvements
  - Open for potential for different network architectures
  - Large hyperparameter optimization should be done
  - Use more and more diverse data to train!
  - Overall plots: Cumbersome to look at all experts

Questions?