NNGeometry

Oct 09, 2020

Contents

NNGeometry is a library built on top of PyTorch aiming at giving tools to easily manipulate and study properties of Fisher Information Matrices and tangent kernels.

You can start by looking at the quick start example below. Convinced? Then *install NNGeometry*, try the tutorials or explore the API reference.

> **Warning:** NNGeometry is under developement, as such it is possible that core components change when between versions.

CHAPTER 1

Tutorials

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

CHAPTER 3

In-depth

3.1 Quick example

With NNGeometry, you can easily manipulate $d \times d$ matrices and d vectors where d is the number of parameter of your neural network, for modern neural networks where d can be as big as 10^8. These matrices include for instance:

- The *Fisher Information Matrix* (FIM) used in statistics, in the natural gradient algorithm, or as an approximate of the Hessian matrix in some applications.
- *Posterior covariances* in Bayesian Deep Learning.

You can also compute finite *tangent kernels*.

A naive computation of the FIM would require storing $d \times d$ scalars in memory. This is prohibitively large for modern neural network architectures, and a line of research has focused at finding lower memory intensive approximations specific to neural networks, such as KFAC, EKFAC, low-rank approximations, etc. This library proposes a common interface for manipulating these different approximations, called *representations*.

Let us now illustrate this by computing the FIM using the KFAC representation.

```
>>> F_kfac = FIM(model=model,
                 loader=loader,
                 representation=PMatKFAC,
                 n_output=10,
                 variant='classif_logits',
                 device='cuda')
>>> print(F_kfac.trace())
```

Computing the FIM requires the following arguments:

- The `torch.nn.Module` `model` object is the PyTorch model used as our neural network.
- The `torch.utils.data.DataLoader` `loader` object is the dataloader that contains examples used for computing the FIM.
- The `object.PMatKFAC` `PMatKFAC` argument specifies which representation to use in order to store the FIM.

We will next define a vector in parameter space, by using the current value given by our model:

```
>>> v = PVector.from_model(model)
```

We can now compute the matrix-vector product Fv by simply calling:

```
>>> Fv = F_kfac.mv(v)
```

Note that switching from the `object.PMatKFAC` representation to any other representation such as `object.PMatDense` is as simple as passing `representation=PMatDense` when building the `F_kfac` object.

More notebook examples can be found at https://github.com/tfjgeorge/nngeometry/tree/master/examples

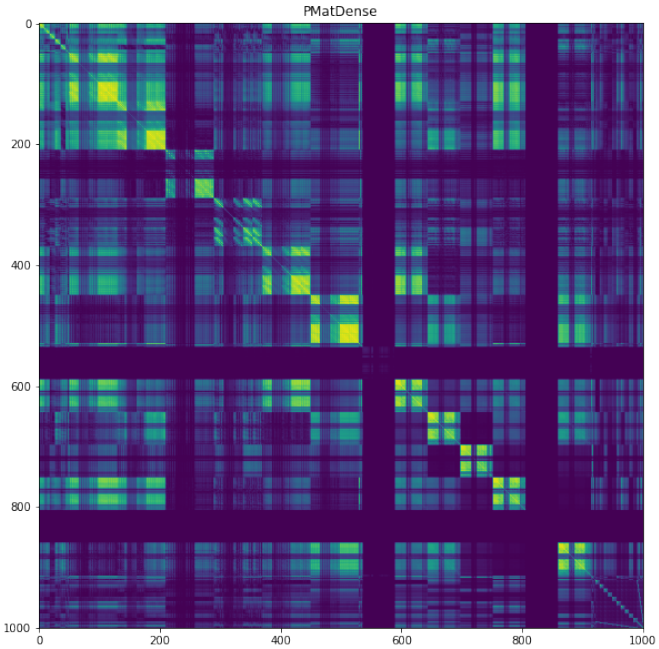
3.2 Installing NNGeometry

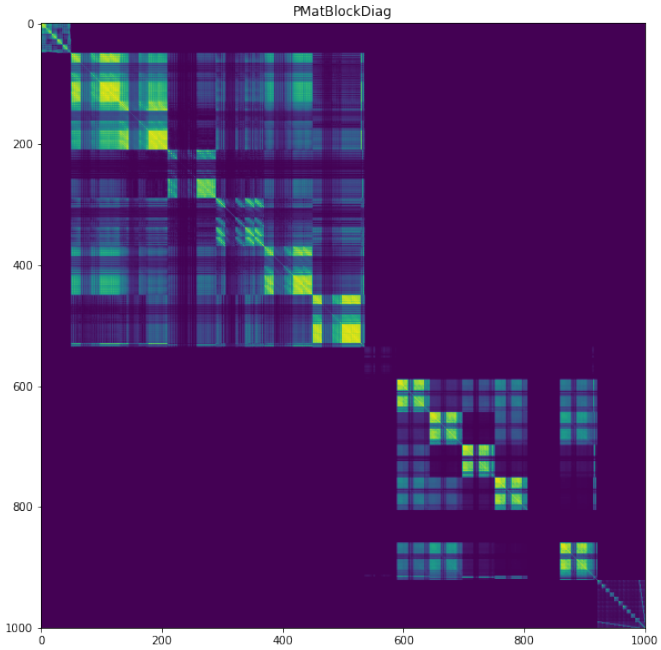
The development version of NNGeometry can be installed by directly cloning the `master` branch of the repository using `pip`:

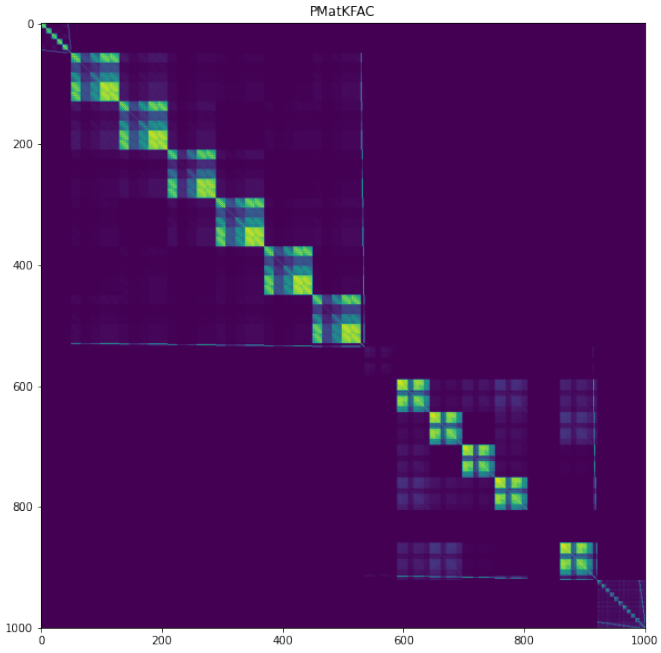
```
$ pip install git+git://github.com/tfjgeorge/nngeometry.git
```

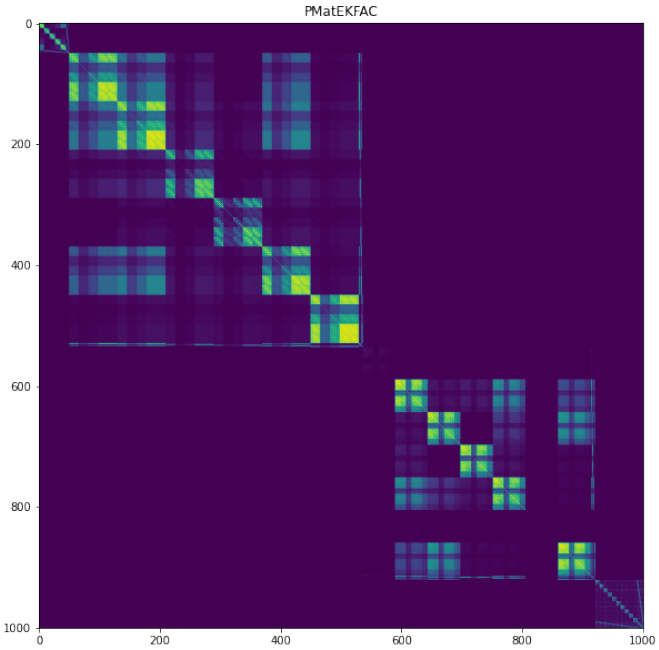
The only dependency is PyTorch.

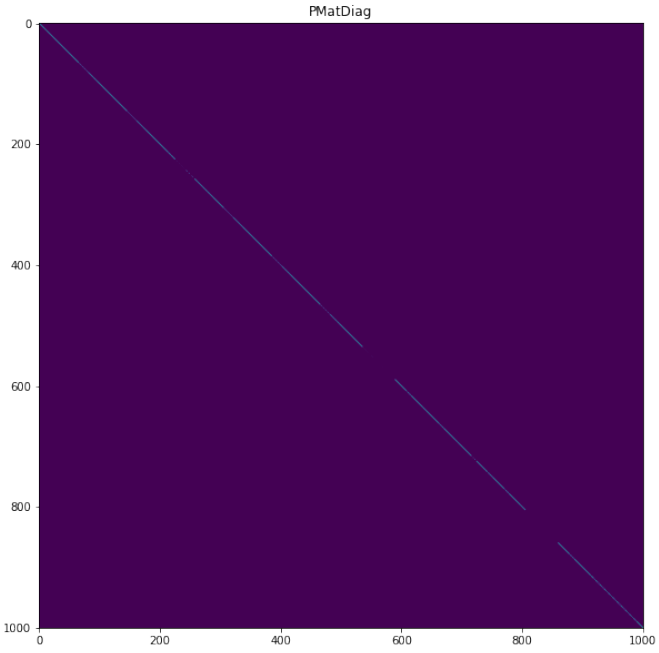
3.3 Parameter space representations

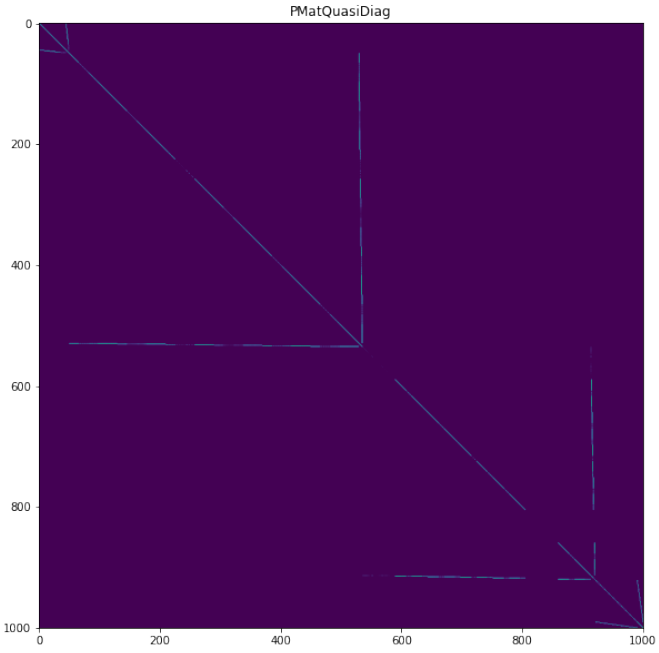



PMatBlockDiag
0
200
400
600
800
1000
0
200
400
600
800
1000


PMatKFAC
0
200
400
600
800
1000
0
200
400
600
800
1000


PMatEKFAC
0
200
400
600
800
1000
0
200
400
600
800
1000


PMatDiag
0
200
400
600
800
1000
0
200
400
600
800
1000



3.4 API Reference

Warning: This API reference is currently nothing but a dump of docstrings, ordered alphabetically.

3.4.1 Generators

The spirit of NNGeometry is that you do not directly manipulate Generator objects, these can be considered as a backend that you do not need to worry about once instantiated. You instead instantiate concrete representations such as *PMatDense* or *PMatKFAC* and directly call linear algebra operations on these concrete representations.

class nngeometry.generator.jacobian.**Jacobian**(*model*, *loader*, *function*, *n_output=1*, *centering=False*, *layer_collection=None*)

Computes jacobians $\mathbf{J}_{ijk} = \frac{\partial f(x_j)_i}{\delta \mathbf{w}_k}$, FIM matrices $\mathbf{F}_{k,k'} = \frac{1}{n}\sum_{i,j}\mathbf{J}_{ijk}\mathbf{J}_{ijk'}$ and NTK matrices $\mathbf{K}_{iji'j'} = \sum_k \mathbf{J}_{ijk}\mathbf{J}_{ijk'}$.

This generator is written in pure PyTorch and exploits some tricks in order to make computations more efficient.

Parameters

- **layer_collection** (*layercollection.LayerCollection*) –
- **model** (Pytorch *nn.Module*) –
- **loader** (Pytorch *utils.data.DataLoader*) –
- **function** (*python function*) – A function $f(X, Y, Z)$ where X, Y, Z are mini-batchs returned by the dataloader (Note that in some cases Y, Z are not required)

- **n_output** (*integer*) – How many output is there for each example of your function. E.g. in 10 class classification this would probably be 10.

3.4.2 Layer collection

Layer collections describe the structure of parameters that will be differentiated. We need the LayerCollection object in order to be able to map components of different objects together. As an example, when performing a matrix-vector product using a block diagonal representation, we need to make sure that elements of the vector corresponding to parameters from layer 1 are multiplied with the diagonal block also corresponding to parameters from layer 1, and so on.

Typical use cases include:

- All parameters of your network: for this you can simply use the constructor *nngeometry.layercollection.LayerCollection.from_model()*
- Only parameters from some layers of your network. In this case you need to

1. instantiate a new LayerCollection object
2. add your layers one at a time using *nngeometry.layercollection.LayerCollection.add_layer_from_model()*

class nngeometry.layercollection.**AbstractLayer**

class nngeometry.layercollection.**BatchNorm1dLayer**(*num_features*)

numel()

class nngeometry.layercollection.**BatchNorm2dLayer**(*num_features*)

numel()

class nngeometry.layercollection.**Conv2dLayer**(*in_channels*, *out_channels*, *kernel_size*, *bias=True*)

numel()

class nngeometry.layercollection.**GroupNormLayer**(*num_groups*, *num_channels*)

numel()

class nngeometry.layercollection.**LayerCollection**(*layers=None*)

This class describes a set or subset of layers, that can be used in order to instantiate `nngeometry.object.PVector` or `nngeometry.object.PSpaceDense` objects

Parameters **layers** –

add_layer(*name*, *layer*)

add_layer_from_model(*model*, *module*)

Add a layer by specifying the module corresponding to this layer (e.g. torch.nn.Linear or torch.nn.BatchNorm1d)

Parameters

- **model** – The model defining the neural network
- **module** – The layer to be added

from_model(*ignore_unsupported_layers=False*)
Constructs a new LayerCollection object by using all parameters of the model passed as argument.

Parameters

- **model** (*nn.Module*) – The PyTorch model
- **ignore_unsupported_layers** – If false, will raise an error

when model contains layers that are not supported yet. If true, will silently ignore the layer :type ignore_unsupported_layers: bool

get_layerid_module_maps(*model*)

numel()
Total number of scalar parameters in this LayerCollection object

Returns number of scalar parameters

Return type int

parameters(*layerid_to_module*)

class nngeometry.layercollection.**LinearLayer**(*in_features*, *out_features*, *bias=True*)

numel()

class nngeometry.layercollection.**Parameter**(**size*)

numel()

3.4.3 Metrics

Metrics introduction

nngeometry.metrics.**FIM**(*model*, *loader*, *representation*, *n_output*, *variant='classif_logits'*, *device='cpu'*, *function=None*, *layer_collection=None*)
Helper that creates a matrix computing the Fisher Information Matrix using closed form expressions for the expectation y|x as described in (Pascanu and Bengio, 2013)

Parameters

- **model** (*torch.nn.Module*) – The model that contains all parameters of the function
- **loader** (*torch.utils.data.DataLoader*) – DataLoader for computing expectation over the input space
- **representation** (*class*) – The parameter matrix representation that will be used to store the matrix
- **n_output** (*int*) – Number of outputs of the model
- **variants** (*string 'classif_logits' or 'regression', optional*) – (default='classif_logits') Variant to use depending on how you interpret your function. Possible choices are:
  - 'classif_logits' when using logits for classification
  - 'regression' when using a gaussian regression model
- **device** (*string, optional (default='cpu')*) – Target device for the returned matrix

- **function** (*function, optional (default=None)*) – An optional function if different from *model(input)*. If it is different from None, it will override the device parameter.
- **layer_collection** (layercollection.LayerCollection, *optional*) – (default=None) An optional layer collection

nngeometry.metrics.**FIM_MonteCarlo**(*model*, *loader*, *representation*, *variant='classif_logits'*, *trials=1*, *device='cpu'*, *function=None*, *layer_collection=None*)

Helper that creates a matrix computing the Fisher Information Matrix using a Monte-Carlo estimate of $\mathbb{E}_{y|x}$ with *trials* samples per example

Parameters

- **model** (*torch.nn.Module*) – The model that contains all parameters of the function
- **loader** (*torch.utils.data.DataLoader*) – DataLoader for computing expectation over the input space
- **representation** (*class*) – The parameter matrix representation that will be used to store the matrix
- **variants** (*string 'classif_logits' or 'regression', optional*) – (default='classif_logits') Variant to use depending on how you interpret your function. Possible choices are:
  - 'classif_logits' when using logits for classification
  - 'regression' when using a gaussian regression model
- **trials** (*int, optional (default=1)*) – Number of trials for Monte Carlo sampling
- **device** (*string, optional (default='cpu')*) – Target device for the returned matrix
- **function** (*function, optional (default=None)*) – An optional function if different from *model(input)*. If it is different from None, it will override the device parameter.
- **layer_collection** (layercollection.LayerCollection, *optional*) – (default=None) An optional layer collection

3.4.4 Parameter space matrix representations

All parameter space matrix representations inherit from *nngeometry.object.pspace.PMatAbstract*. This abstract class defines all method that can be used with all representations (with some exceptions!). *nngeometry.object.pspace.PMatAbstract* cannot be instantiated, you instead have to choose one of the concrete representations below.

class nngeometry.object.pspace.**PMatAbstract**(*generator*, *data=None*)

A $d \times d$ matrix in parameter space. This abstract class defines common methods used in concrete representations.

Parameters

- **generator** (*nngeometry.generator.jacobian.Jacobian*) – The generator
- **data** – if None, uses the generator to populate the matrix data.

get_diag()

Computes and returns the diagonal elements of this matrix.

Returns a PyTorch Tensor

size(*dim=None*)
Size of the matrix as a tuple, regardless of the actual size in memory.

Parameters **dim** (*int or None*) – dimension

```
>>> M.size()
(1254, 1254)
>>> M.size(0)
1254
```

solve(*v*, *regul*)
Solves Fx = v in x

Parameters

- **regul** (PVector) – Tikhonov regularization
- **v** – v

vTMv(*v*)
Computes the quadratic form defined by M in v, namely the product $v^\top M v$

Parameters **v** (*object.vector.PVector*) – vector v

3.4.5 Concrete representations

NNGeometry allows to switch between representations easily. With each representation comes a tradeof between accuracy and memory/computational cost. If testing a new algorithm, we recommend testing on a small network using the most accurate representation that fits in memory (typically *nngeometry.object.pspace.PMatDense*), then switch to a larger scale experiment, and to a lower memory representation.

class nngeometry.object.pspace.**PMatBlockDiag**(*generator*, *data=None*)

get_diag()
Computes and returns the diagonal elements of this matrix.

Returns a PyTorch Tensor

solve(*vs*, *regul=1e-08*)
Solves Fx = v in x

Parameters

- **regul** (PVector) – Tikhonov regularization
- **v** – v

vTMv(*vector*)
Computes the quadratic form defined by M in v, namely the product $v^\top M v$

Parameters **v** (*object.vector.PVector*) – vector v

class nngeometry.object.pspace.**PMatDense**(*generator*, *data=None*)

get_diag()
Computes and returns the diagonal elements of this matrix.

Returns a PyTorch Tensor

solve(*v*, *regul=1e-08*, *impl='solve'*)
solves v = Ax in x

vTMv(*v*)
Computes the quadratic form defined by M in v, namely the product $v^\top Mv$

Parameters **v** (*object.vector.PVector*) – vector v

class nngeometry.object.pspace.**PMatDiag**(*generator=None*, *data=None*)

get_diag()
Computes and returns the diagonal elements of this matrix.

Returns a PyTorch Tensor

solve(*v*, *regul=1e-08*)
solves v = Ax in x

vTMv(*v*)
Computes the quadratic form defined by M in v, namely the product $v^\top Mv$

Parameters **v** (*object.vector.PVector*) – vector v

class nngeometry.object.pspace.**PMatEKFAC**(*generator*, *data=None*)
EKFAC representation from *George, Laurent et al., Fast Approximate Natural Gradient Descent in a Kronecker-factored Eigenbasis, NIPS 2018*

get_dense_tensor(*split_weight_bias=True*)

- split_weight_bias (bool): if True then the parameters are ordered in

the same way as in the dense or blockdiag representation, but it involves more operations. Otherwise the coefficients corresponding to the bias are mixed between coefficients of the weight matrix

get_diag(*v*)
Computes and returns the diagonal elements of this matrix.

Returns a PyTorch Tensor

solve(*vs*, *regul=1e-08*)
Solves Fx = v in x

Parameters

- **regul** (PVector) – Tikhonov regularization
- **v** – v

update_diag()
Will update the diagonal in the KFE (aka the approximate eigenvalues) using current values of the model's parameters

vTMv(*vector*)
Computes the quadratic form defined by M in v, namely the product $v^\top Mv$

Parameters **v** (*object.vector.PVector*) – vector v

class nngeometry.object.pspace.**PMatImplicit**(*generator*)
PMatImplicit is a very special representation, since no elements of the matrix is ever computed, but instead various linear algebra operations are performed implicitely using efficient tricks.

The computations are done exactly, meaning that there is no approximation involved. This is useful for networks too big to fit in memory.

get_diag()
Computes and returns the diagonal elements of this matrix.

Returns a PyTorch Tensor

solve(*v*)
Solves Fx = v in x

Parameters

- **regul** (PVector) – Tikhonov regularization
- **v** – v

vTMv(*v*)
Computes the quadratic form defined by M in v, namely the product $v^\top M v$

Parameters **v** (*object.vector.PVector*) – vector v

class nngeometry.object.pspace.**PMatKFAC**(*generator*, *data=None*)

get_dense_tensor(*split_weight_bias=True*)

- split_weight_bias (bool): if True then the parameters are ordered in

the same way as in the dense or blockdiag representation, but it involves more operations. Otherwise the coefficients corresponding to the bias are mixed between coefficients of the weight matrix

get_diag(*split_weight_bias=True*)

- split_weight_bias (bool): if True then the parameters are ordered in

the same way as in the dense or blockdiag representation, but it involves more operations. Otherwise the coefficients corresponding to the bias are mixed between coefficients of the weight matrix

solve(*vs*, *regul=1e-08*, *use_pi=True*)
Solves Fx = v in x

Parameters

- **regul** (PVector) – Tikhonov regularization
- **v** – v

vTMv(*vector*)
Computes the quadratic form defined by M in v, namely the product $v^\top M v$

Parameters **v** (*object.vector.PVector*) – vector v

class nngeometry.object.pspace.**PMatLowRank**(*generator*, *data=None*)

get_diag()
Computes and returns the diagonal elements of this matrix.

Returns a PyTorch Tensor

solve(*v*)
Solves Fx = v in x

Parameters

- **regul** (PVector) – Tikhonov regularization
- **v** – v

vTMv(*v*)
Computes the quadratic form defined by M in v, namely the product $v^\top Mv$

Parameters **v** (*object.vector.PVector*) – vector v

class nngeometry.object.pspace.**PMatQuasiDiag**(*generator*, *data=None*)
Quasidiagonal approximation as decribed in Ollivier, Riemannian metrics for neural networks I: feedforward networks, Information and Inference: A Journal of the IMA, 2015

get_diag()
Computes and returns the diagonal elements of this matrix.

Returns a PyTorch Tensor

solve(*vs*, *regul=1e-08*)
Solves Fx = v in x

Parameters

- **regul** (PVector) – Tikhonov regularization
- **v** – v

vTMv(*vs*)
Computes the quadratic form defined by M in v, namely the product $v^\top Mv$

Parameters **v** (*object.vector.PVector*) – vector v

nngeometry.object.pspace.**bdot**(*A*, *B*)
batched dot product

3.4.6 Vector representations

In NNGeometry, vectors are not just a bunch of scalars, but they have a semantic meaning.

- *nngeometry.object.vector.PVector* objects are vectors living in the parameter space of a neural network model. An example of such vector is $\delta\mathbf{w}$ in the EWC penalty $\delta\mathbf{w}^\top F\delta\mathbf{w}$.
- *nngeometry.object.vector.FVector* objects are vectors living in the function space of a neural network model. An example of such vector is $\mathbf{f} = (f(x_1), \ldots, f(x_n))^\top$ where f is a neural network and $x_1, \ldots, x_n$ are examples from a training dataset.

class nngeometry.object.vector.**FVector**(*vector_repr=None*)
Bases: object

A vector in function space

get_flat_representation()

class nngeometry.object.vector.**PVector**(*layer_collection*, *vector_repr=None*, *dict_repr=None*)
Bases: object

A vector in parameter space

Param

clone()
Returns a clone of the current object

copy_to_model(*model*)
Updates *model* parameter values with the current PVector

Note. This is an inplace operation

detach()
Detachs the current PVector from the computation graph

static from_model(*model*)
Creates a PVector using the current values of the given model

static from_model_grad(*model*)
Creates a PVector using the current values of the *.grad* fields of parameters of the given model

get_dict_representation()

get_flat_representation()
Returns a Pytorch 1d tensor of the flatten vector.

> **Warning:** The ordering in which the parameters are flattened can seem to be arbitrary. It is in fact the same ordering as specified by the `layercollection.LayerCollection` object.

Returns a Pytorch Tensor

norm(*p=2*)
Computes the Lp norm of the PVector

nngeometry.object.vector.**random_fvector**(*n_samples*, *n_output=1*, *device=None*)

nngeometry.object.vector.**random_pvector**(*layer_collection*, *device=None*)
Returns a random `nngeometry.object.PVector` object using the structure defined by the *layer_collection* parameter, with each components drawn from a normal distribution with mean 0 and standard deviation 1.

The returned *PVector* will internally use a flat representation.

Parameters **layer_collection** – The *nngeometry.layercollection.LayerCollection* describing the structure of the random pvector

nngeometry.object.vector.**random_pvector_dict**(*layer_collection*, *device=None*)
Returns a random `nngeometry.object.PVector` object using the structure defined by the *layer_collection* parameter, with each components drawn from a normal distribution with mean 0 and standard deviation 1.

The returned *PVector* will internally use a dict representation.

Parameters **layer_collection** – The *nngeometry.layercollection.LayerCollection* describing the structure of the random pvector

CHAPTER 4

Quick start

Python Module Index

n

Index

J

L

N

P

R

S

U

V