# Neural Framework Documentation

## Contents

# Module `nf`

The neural framework package.

## Sub-modules

- nf.agents
- nf.experiment
- nf.problems

# Module `nf.agents`

The agents module provides an interface for defining neural agents along with some agent implementations.

## Sub-modules

- nf.agents.greedy_probing

- nf.agents.loss_with_goal_line_deviation
- nf.agents.mse
- nf.agents.sampling
- nf.agents.simulated_annealing
- nf.agents.util

## Classes

### Class `Agent`

```
class Agent(problem: nf.problems.Problem)
```

Abstract class representing a neural agent.

Constructor.

**Arguments** problem {Problem} – the instance of the problem that this agent will train on

#### Ancestors (in MRO)

- abc.ABC

#### Descendants

- nf.agents.GradientBasedAgent
- nf.agents.GradientFreeAgent

#### Methods

#### Method `compile`

```
def compile(self)
```

Set up the agent.

This method is called once before training and must be overridden

#### Method `fit`

```
def fit(self, X: tensorflow.python.framework.ops.Tensor, y: tensorflow.python.framework.ops.Tens
epochs: int, callbacks: List[tensorflow.python.keras.callbacks.Callback])
```

Abstract method to perform training. Akin to keras' fit method: the aim is to fit the model to the dataset.

Arguments

X {tf.Tensor} – input matrix y {tf.Tensor} – output samples epochs {int} – number of epochs to train for callbacks {typing.List[tf.keras.callbacks.Callback]} – list of keras callbacks to be registered

#### Method `train`

```
def train(self, epochs: int, metrics: List[str] = None) -> Dict[str, numpy.ndarray]
```

Train the agent for a specific number of epochs, logging the required metrics.

Arguments

epochs {int} – the number of epochs to train for

Keyword Arguments: metrics {typing.List[str]} – the epochs to log (need to be specified as part of the Problem instance) (default: {None})

Returns

**`typing.Dict[str, np.ndarray] -- dictionary of collected metrics indexed by their name`**

### Class `GradientBasedAgent`

```
class GradientBasedAgent(problem: nf.problems.Problem)
```

An abstract class representing an agent that uses derivatives (i.e. can train normally using the keras fit method).

Constructor.

**Arguments** problem {Problem} – the instance of the problem that this agent will train on

#### Ancestors (in MRO)

- nf.agents.Agent
- abc.ABC

#### Descendants

- nf.agents.loss_with_goal_line_deviation.LossWithGoalLineDeviation
- nf.agents.mse.MSE

### Class `GradientFreeAgent`

```
class GradientFreeAgent(problem: nf.problems.Problem, sampler: nf.agents.sampling.SamplingTechni
```

An abstract class representing a derivative-free agent. This means that the agent uses a sampling technique instead of relying on derivative information.

Constructor

**Arguments** problem {Problem} – the instance of the problem that this agent will train on sampler {SamplingTechnique} – the sampling technique to employ

#### Ancestors (in MRO)

- nf.agents.Agent
- abc.ABC

#### Descendants

- nf.agents.greedy_probing.GreedyProbing
- nf.agents.simulated_annealing.SimulatedAnnealing

#### Methods

##### Method `choose_best_weight_update`

```
def choose_best_weight_update(self, weight_samples: tensorflow.python.framework.ops.Tensor,
weight_history: tensorflow.python.framework.ops.Tensor, output_history: tensorflow.python.framew
X: tensorflow.python.framework.ops.Tensor, y: tensorflow.python.framework.ops.Tensor)
-> tensorflow.python.framework.ops.Tensor
```

Abstract method that decides which weight update to choose.

This is the only method that needs to be overridden by the agent implementation. Given a set of samples in weight space (produced by the sampling techniques) as well as other information (history of weights and outputs, as well as input samples and output targets), this function should return the new weight state that should be chosen. This method is called once per epoch.

Arguments

weight_samples {tf.Tensor} – the samples in weight space produced by the sampling technique weight_history {tf.Tensor} – the history of weight states output_history {tf.Tensor} – the history of predictions (output states) X {tf.Tensor} – the input matrix y {tf.Tensor} – the output targets

Returns

**tf.Tensor -- the new weight state chosen by the agent**

### Method `get_weights`

```
def get_weights(self) -> tensorflow.python.framework.ops.Tensor
```

Get the current weight state as a concatenated vector.

Returns

**tf.Tensor -- the weight vector**

### Method `predict_for_multiple_weights`

```
def predict_for_multiple_weights(self, weights: tensorflow.python.framework.ops.Tensor,
X: tensorflow.python.framework.ops.Tensor) -> tensorflow.python.framework.ops.Tensor
```

A batched version of the predict_for_weights function.

Batching is performed over the first dimension of the weights tensor.

Arguments

weights {tf.Tensor} – the weight matrix (collection of weight vectors) X {tf.Tensor} – the input samples

Returns

**tf.Tensor -- the outputs for each weight vector**

### Method `predict_for_weights`

```
def predict_for_weights(self, weights: tensorflow.python.framework.ops.Tensor,
X: tensorflow.python.framework.ops.Tensor) -> tensorflow.python.framework.ops.Tensor
```

Get the prediction output for the model given a specific weight state.

Note: this function does *not* change the weights back!

Arguments

weights {tf.Tensor} – the weight vector X {tf.Tensor} – the input matrix

Returns

**tf.Tensor -- the outputs (predictions)**

### Method `set_weights`

```
def set_weights(self, weights: tensorflow.python.framework.ops.Tensor)
```

Set the weight vector.

Internally, the weights are formed back into the shape required by keras.

Arguments

weights {tf.Tensor} – the weight vector

## Module `nf.agents.greedy_probing`

The greedy probing agent.

## Classes

### Class GreedyProbing

```
class GreedyProbing(problem: nf.problems.Problem, sampler: nf.agents.sampling.SamplingTechnique)
```

Implementation of the greedy probing agent.

Constructor

**Arguments** problem {Problem} – the instance of the problem that this agent will train on sampler {SamplingTechnique} – the sampling technique to employ

### Ancestors (in MRO)

- nf.agents.GradientFreeAgent
- nf.agents.Agent
- abc.ABC

# Module nf.agents.loss_with_goal_line_deviation

A gradient-based agent with a custom loss function that tries to minimise the distance to the goal line.

## Classes

### Class LossWithGoalLineDeviation

```
class LossWithGoalLineDeviation(problem: nf.problems.Problem, learning_rate: float = 0.01,
momentum: float = 0.0)
```

Implementation of a gradient-based agent with a custom loss function that tries to minimise the distance to the goal line.

Constructor.

**Arguments** problem {Problem} – the instance of the problem that this agent will train on

Keyword Arguments: learning_rate {float} – the learning rate (default: {0.01}) momentum {float} – the momentum (default: {0.0})

### Ancestors (in MRO)

- nf.agents.GradientBasedAgent
- nf.agents.Agent
- abc.ABC

# Module nf.agents.mse

The classical steepest gradient descent agent with mean squared eror.

## Classes

### Class MSE

```
class MSE(problem: nf.problems.Problem, learning_rate: float = 0.01, momentum: float = 0.0)
```

Implementation of the classical steepest gradient descent agent with mean squared eror.

Constructor.

**Arguments** problem {Problem} – the instance of the problem that this agent will train on

Keyword Arguments: learning_rate {float} – the learning rate (default: {0.01}) momentum {float} – the momentum (default: {0.0})

#### Ancestors (in MRO)

- nf.agents.GradientBasedAgent
- nf.agents.Agent
- abc.ABC

# Module `nf.agents.sampling`

Sampling techniques for gradient-free agents.

## Classes

### Class `ExhaustiveSamplingTechnique`

```
class ExhaustiveSamplingTechnique(sample_radius: float, uniform_radius: bool = True)
```

Implementation of an exhaustive sampling technique that obtains samples with a specific radius along all possible 45° directions in a weight space of arbitrary dimensionality.

The number of samples will be 3^N-1 where N is the dimensionality of the weight space.

Constructor.

**Arguments** sample_radius {float} – the sample radius

Keyword Arguments: uniform_radius {bool} – whether or not to ensure that all samples have the same length (if False, the samples at 45° angles will be longer than the ones at 90° angles to any weight axis) (default: {True})

#### Ancestors (in MRO)

- nf.agents.sampling.SamplingTechnique
- abc.ABC

### Class `RandomSamplingGenerator`

```
class RandomSamplingGenerator(sample_radius: float, uniform_radius: bool = True)
```

Similar to RandomSamplingTechnique, except that samples are obtained on-demand using a generator.

This should be used when the number of samples needed is not known beforehand, and each sample is processed separately until reaching a termination condition.

Constructor.

**Arguments** sample_radius {float} – the sample radius num_samples {int} – the number of random samples to generate

Keyword Arguments: uniform_radius {bool} – whether the samples should be along the circumference of the sampling circle (True) or in the area of the sampling circle (False) (default: {True})

#### Ancestors (in MRO)

- nf.agents.sampling.SamplingTechnique
- abc.ABC

### Class `RandomSamplingTechnique`

```
class RandomSamplingTechnique(sample_radius: float, num_samples: int, uniform_radius: bool = Tru
```

Implementation of a random sampling technique that obtains samples within a specific radius in a weight space of arbitrary dimensionality.

Constructor.

**Arguments** sample_radius {float} – the sample radius num_samples {int} – the number of random samples to generate

Keyword Arguments: uniform_radius {bool} – whether the samples should be along the circumference of the sampling circle (True) or in the area of the sampling circle (False) (default: {True})

#### Ancestors (in MRO)

- nf.agents.sampling.SamplingTechnique
- abc.ABC

#### Class `SamplingTechnique`

```
class SamplingTechnique()
```

Abstract base class representing a sampling technique (for weight space).

#### Ancestors (in MRO)

- abc.ABC

#### Descendants

- nf.agents.sampling.ExhaustiveSamplingTechnique
- nf.agents.sampling.RandomSamplingGenerator
- nf.agents.sampling.RandomSamplingTechnique

#### Methods

#### Method `initialize`

```
def initialize(self, num_weights: int)
```

Initialize the sampling technique.

This method should be called once before training. Its purpose is to pre-compute certain values such that they do not have to be computed over and over again during training.

Arguments

num_weights {int} – the dimensionality of the weight space

# Module `nf.agents.simulated_annealing`

The simulated annealing agent.

## Classes

#### Class `SimulatedAnnealing`

```
class SimulatedAnnealing(problem: nf.problems.Problem, learning_rate: float,
max_attempts_per_iteration: int = 15, energy_coefficient: float = 10000.0,
temperature: float = 10000.0, cooling_rate: float = 0.05)
```

Implementation of the simulated annealing agent.

Constructor.

**Arguments** problem {Problem} – the instance of the problem that this agent will train on learning_rate {float} – the learning rate (essentially the sampling radius)

Keyword Arguments: max_attempts_per_iteration {int} – the maximum number of non-accepting samples to evaluate each iteration before falling back the best one seen thus far (default: {15}) energy_coefficient {float} – the energy coefficient (default: {10000.}) temperature {float} – the initial temperature (default: {10000.}) cooling_rate {float} – the cooling rate (default: {.05})

#### Ancestors (in MRO)

- nf.agents.GradientFreeAgent
- nf.agents.Agent
- abc.ABC

#### Methods

##### Method `cost`

```
def cost(self, current_output: tensorflow.python.framework.ops.Tensor,
target_output: tensorflow.python.framework.ops.Tensor) -> tensorflow.python.framework.ops.Tenso
```

A simple cost function.

Arguments

current_output {tf.Tensor} – the current output target_output {tf.Tensor} – the target

Returns

```
tf.Tensor -- the Euclidean distance between the current and target output
```

# Module `nf.agents.util`

Utilities for the agents module.

## Functions

### Function `get_distance_to_line`

```
def get_distance_to_line(point: tensorflow.python.framework.ops.Tensor,
a: tensorflow.python.framework.ops.Tensor, b: tensorflow.python.framework.ops.Tensor)
-> tensorflow.python.framework.ops.Tensor
```

Utility function to get the shortest distance from a point to the line passing through a and b.

Arguments

point {tf.Tensor} – the point a {tf.Tensor} – one point on the line b {tf.Tensor} – another point on the line

Returns

```
tf.Tensor -- the distance
```

### Function `get_num_weights`

```
def get_num_weights(model: tensorflow.python.keras.engine.training.Model) ->
int
```

Utility function to determine the number of weights in a keras model.

Arguments

model {tf.keras.Model} – the keras model

Returns

`int -- the number of weights`

#### Function `scale_to_length`

```
def scale_to_length(x: tensorflow.python.framework.ops.Tensor, length: float)
-> tensorflow.python.framework.ops.Tensor
```

Utility function to scale vector(s) to a certain length.

Arguments

x {tf.Tensor} – the vector (or collection of vectors in form of a matrix) length {float} – the length

Returns

`tf.Tensor -- the scaled vector(s)`

# Module `nf.experiment`

The experiment module.

## Sub-modules

- nf.experiment.metrics
- nf.experiment.visualisations

## Classes

#### Class `Experiment`

```
class Experiment(agents: Dict[str, nf.agents.Agent])
```

Class representing an experiment that can be run.

When it is run, the experment will coordinate the training of the agents. The experiment will also manage the collection and aggregation of metrics, and ensure data is passed to the visualisations to update them in real time.

Constructor.

**Arguments** agents {typing.Dict[str, Agent]} – the list of agents that will be run for this experiment

#### Methods

#### Method `run`

```
def run(self, doc: bokeh.document.document.Document, visualisations: Sequence[nf.experiment.visu
epoch_batches: int = 100, epoch_batch_size: int = 50, cols: int = 2, title: str = 'Experiment')
```

Run the experiment.

Arguments

doc {Document} – the bokeh Document visualisations {typing.Sequence[Visualisation]} – the visualisations to show in real time

Keyword Arguments: epoch_batches {int} – the number of batches of training to perform for each agent (default: {100}) epoch_batch_size {int} – the number of epochs to train for per training batch (default: {50}) cols {int} – the number of columns to display the visualisations in (default: {2}) title {str} – optional title of the web page (default: {"Experiment"})

#### Method `run_server`

```
def run_server(self, *args, port: int = 5000, **kwargs)
```

Start the bokeh server with the experiment (*args and **kwargs are passed on to the Experiment.run() method).

Keyword Arguments: port {int} – the port to run the server on (default: {5000})

# Module `nf.experiment.metrics`

Metrics for experiments.

## Classes

### Class `Metric`

```
class Metric(name: str, dimensions: tuple = None)
```

Class representing a metric and meta-information about it.

A metric is a tensor of arbitrary rank. However, when visualising a metric, we want a time series of scalar values. This means that we store some index into the dimensionality of the metric that will obtain a scalar value. For example, a metric of shape [5, 6, 7] could be indexed by [0, 0, 0] which would obtain the first element.

Constructor.

**Arguments** name {str} – the name of the metric

Keyword Arguments: dimensions {tuple} – the index into the dimensionality of the metric (see note above) (default: {None})

#### Static methods

##### Method `from_string`

```
def from_string(description: str) -> nf.experiment.metrics.Metric
```

Create an instance of the Metric class from a string.

The string will be of the form "name:a:b:c" where "name" is the name of the metric and a, b, c consitute the index.

Arguments

description {str} – the string representation of the metric

Returns

```
Metric -- the Metric instance
```

#### Methods

##### Method `select`

```
def select(self, metrics: typing.Dict[str, np.ndarray]) -> numpy.ndarray
```

Obtain the data for this metric using the index.

Arguments

metrics {typing.Dict[str, np.ndarray]} – a dictionary of metrics

Returns

```
np.ndarray -- the metric
```

# Module `nf.experiment.visualisations`

Visualisations for experiments.

## Sub-modules

- nf.experiment.visualisations.histogram
- nf.experiment.visualisations.scatter2d

## Classes

### Class `Visualisation`

```
class Visualisation(required_metrics: List[str])
```

Abstract base class for a visualisation.

Constructor.

**Arguments** required_metrics {typing.List[str]} – the metrics required by the initialisation (this is how the visualisation registers for receiving specific data)

#### Ancestors (in MRO)

- abc.ABC

#### Descendants

- nf.experiment.visualisations.scatter2d.Scatter2D

#### Methods

##### Method `plot`

```
def plot(self, metrics: List[Dict[str, numpy.ndarray]], plot: bokeh.plotting.figure.Figure,
doc: bokeh.document.document.Document)
```

Abstract method to update the plot with new data.

Arguments

metrics {typing.List[typing.Dict[str, np.ndarray]]} – the relevant metrics plot {Figure} – reference to the plot doc {Document} – the enclosing bokeh document

##### Method `setup`

```
def setup(self) -> Tuple[bokeh.plotting.figure.Figure, List[bokeh.models.glyphs.Line]]
```

An abstract method that will be called once to setup the visualisation with bokeh.

Returns

**`typing.Tuple[Figure, typing.List[Line]] -- the plot` and list of artists representing the agents on the**

# Module `nf.experiment.visualisations.histogram`

A histogram plot.

## Classes

### Class `Histogram`

```
class Histogram(metric: str, title: str = None)
```

Class implementing the histogram visualisation.

Histograms show the progression of one metric over time (epochs).

Constructor.

**Arguments** metric {str} – the name of the metric to visualise

Keyword Arguments: title {str} – optional title of the graph (default: {None})

#### Ancestors (in MRO)

- nf.experiment.visualisations.scatter2d.Scatter2D
- nf.experiment.visualisations.Visualisation
- abc.ABC

# Module `nf.experiment.visualisations.scatter2d`

A two-dimensional scatter plot.

## Classes

### Class `Scatter2D`

```
class Scatter2D(x: str, y: str = None, title: str = None, x_title: str = None,
y_title: str = None)
```

A two-dimensional scatter plot visualisation.

Constructor.

**Arguments** x {str} – the x-axis metric (specified using the metric syntax)

Keyword Arguments: y {str} – the y-axis metric (if unspecified, the y axis will be the next dimension of the x-axis metric) (default: {None}) title {str} – optional title of the graph (default: {None}) x_title {str} – optional x-axis title (default: {None}) y_title {str} – optional y-axis title (default: {None})

#### Raises

**ValueError** if the metrics are not specified correctly

#### Ancestors (in MRO)

- nf.experiment.visualisations.Visualisation
- abc.ABC

#### Descendants

- nf.experiment.visualisations.histogram.Histogram

# Module `nf.problems`

Module for the definition of neural problems, providing some problem implementations too.

## Sub-modules

- nf.problems.shallow_problem
- nf.problems.simple_problem
- nf.problems.stripe_problem
- nf.problems.util

## Classes

### Class `Problem`

```
class Problem(X: numpy.ndarray, y: numpy.ndarray, model: tensorflow.python.keras.engine.training
```

Abstract base class representing a neural problem.

Constructor.

**Arguments** X {np.ndarray} – the input matrix y {np.ndarray} – the output targets model {tf.keras.Model} – the keras model

#### Ancestors (in MRO)

- abc.ABC

#### Descendants

- nf.problems.shallow_problem.ShallowProblem
- nf.problems.simple_problem.SimpleProblem
- nf.problems.stripe_problem.StripeProblem

#### Instance variables

**Variable `X`** Get the input matrix.

Returns

`np.ndarray -- the input matrix`

**Variable `model`** Get the keras model.

Returns

`np.ndarray -- the keras model`

**Variable `y`** Get the output targets.

Returns

`np.ndarray -- the output targets`

#### Static methods

##### `Method metric`

```
def metric(func: Callable) -> Callable
```

Decorator function that should be used within a problem implementation to define metrics as functions.

Arguments

func {typing.Callable} – a function that calculates a metric

Returns

`typing.Callable -- the decorated function`

### Methods

#### Method `evaluate_metrics`

```
def evaluate_metrics(self, metrics: List[str] = None) -> Dict[str, numpy.ndarray]
```

Evaluate a subset of metrics.

Keyword Arguments: metrics {typing.List[str]} – the list of metrics to evaluate (None means all metrics) (default: {None})

Returns

**`typing.Dict[str, np.ndarray] -- the evaluated metrics as a map from metric name to value(s)`**

# Module `nf.problems.shallow_problem`

The shallow excitation problem (see Figure 7 in "../../progress/main.pdf").

## Classes

### Class `ShallowProblem`

```
class ShallowProblem()
```

Implementation of the shallow excitation problem (see Figure 7 in "../../progress/main.pdf").

Constructor.

**Arguments** X {np.ndarray} – the input matrix y {np.ndarray} – the output targets model {tf.keras.Model} – the keras model

#### Ancestors (in MRO)

- nf.problems.Problem
- abc.ABC

#### Methods

##### Method `loss`

```
def loss(self)
```

##### Method `output`

```
def output(self)
```

##### Method `weights`

```
def weights(self)
```

# Module `nf.problems.simple_problem`

A simple neural problem that likely has no suboptimal local minima; can be used for testing that agents work under normal circumstances.

## Classes

### Class `SimpleProblem`

```
class SimpleProblem()
```

Implementation a simple neural problem that likely has no suboptimal local minima; can be used for testing that agents work under normal circumstances.

Constructor.

**Arguments** X {np.ndarray} – the input matrix y {np.ndarray} – the output targets model {tf.keras.Model} – the keras model

#### Ancestors (in MRO)

- nf.problems.Problem
- abc.ABC

#### Methods

##### Method `loss`

```
def loss(self)
```

##### Method `output`

```
def output(self)
```

##### Method `weights`

```
def weights(self)
```

# Module `nf.problems.stripe_problem`

The RBF stripe problem.

## Functions

### Function `rbf`

```
def rbf(x: tensorflow.python.framework.ops.Tensor) -> tensorflow.python.framework.ops.Tensor
```

The radial basis activation function $e^{(-x}2)$.

Arguments

x {tf.Tensor} – the excitation

Returns

**`tf.Tensor -- the activation`**

## Classes

### Class `StripeProblem`

```
class StripeProblem()
```

Implementation of the RBF stripe problem.

Constructor.

**Arguments** X {np.ndarray} – the input matrix y {np.ndarray} – the output targets model {tf.keras.Model} – the keras model

#### Ancestors (in MRO)

- nf.problems.Problem
- abc.ABC

#### Methods

##### Method `loss`

```
def loss(self)
```

##### Method `output`

```
def output(self)
```

##### Method `weights`

```
def weights(self)
```

# Module `nf.problems.util`

Utilities for the problems module.

## Classes

### Class `DenseWithFixedBias`

```
class DenseWithFixedBias(num_outputs: int, bias: float, kernel: tensorflow.python.framework.ops
```

A fully-conected keras layer with a fixed bias term.

Constructor.

**Arguments** num_outputs {int} – number of output units bias {float} – value of the bias kernel {tf.Tensor} – the initial weights

#### Ancestors (in MRO)

- tensorflow.python.keras.engine.base_layer.Layer
- tensorflow.python.module.module.Module
- tensorflow.python.training.tracking.tracking.AutoTrackable
- tensorflow.python.training.tracking.base.Trackable

#### Methods

##### Method `build`

```
def build(self, input_shape: tensorflow.python.framework.tensor_shape.TensorShape)
```

Build the layer for keras.

Arguments

input_shape {tf.TensorShape} – the output shape of the previous layer

##### Method `call`

```
def call(self, input: tensorflow.python.framework.ops.Tensor) -> tensorflow.python.framework.ops
```

Perform the forward pass.

Arguments

input {tf.Tensor} – the previous layer's output

Returns

**`tf.Tensor -- the result`**

---

Generated by *pdoc* 0.8.1 (https://pdoc3.github.io).