MagNet Documentation

Release 0.1

Vaisakh

Aug 30, 2018

Package Reference

CHAPTER 1

magnet

magnet.**eval**(**modules*)

A Context Manger that makes it easy to run computations in `eval` mode.

It sets modules in their `eval` mode and ensures that gradients are not computed.

This is a more wholesome option than torch.no_grad() since many Modules (BatchNorm, Dropout etc.) behave differently while training and testing.

Examples:

```
>>> import magnet as mag
```

```
>>> import magnet.nodes as mn
```

```
>>> import torch
```

```
>>> model = mn.Linear(10)
```

```
>>> x = torch.randn(4, 3)
```

```
>>> # Using eval() as context manager
>>> with mag.eval(model):
>>>     model(x)
```

```
>>> # Use as decorator
>>> @mag.eval(model)
>>> def foo():
>>>     return model(x)
>>> foo()
```

```
>>> # The modules can also be given at runtime by specifying no arguments
>>> @mag.eval
```

(continues on next page)

(continued from previous page)

```
>>> def foo(model):
>>>     return model(x)
>>> foo()
>>> # The method then takes modules from the arguments
>>> # to the decorated function.
```

CHAPTER 2

magnet.data

2.1 Data

class magnet.data.**Data**(*train*, *val=None*, *test=None*, *val_split=0.2*, ***kwargs*)

A container which holds the Training, Validation and Test Sets and provides DataLoaders on call.

This is a convenient abstraction which is used downstream with the Trainer and various debuggers.

It works in tandem with the custom Dataset, DataLoader and Sampler sub-classes that MagNet defines.

Parameters

- **train** (Dataset) – The training set
- **val** (Dataset) – The validation set. Default: None
- **test** (Dataset) – The test set. Default: None
- **val_split** (*float*) – The fraction of training data to hold out as validation if validation set is not given. Default: 0.2

Keyword Arguments

- **num_workers** (*int*) – how many subprocesses to use for data loading. 0 means that the data will be loaded in the main process. Default: 0
- **collate_fn** (*callable*) – merges a list of samples to form a mini-batch Default: pack_collate()
- **pin_memory** (*bool*) – If True, the data loader will copy tensors into CUDA pinned memory before returning them. Default: False
- **timeout** (*numeric*) – if positive, the timeout value for collecting a batch from workers. Should always be non-negative. Default: 0
- **worker_init_fn** (*callable*) – If not None, this will be called on each worker subprocess with the worker id (an int in [0, num_workers - 1]) as input, after seeding and before data loading. Default: None

- **transforms** (*list or callable*) – A list of transforms to be applied to each datapoint. Default: None
- **fetch_fn** (*callable*) – A function which is applied to each datapoint before collating. Default: None

__call__(*batch_size=1*, *shuffle=False*, *replace=False*, *probabilities=None*, *sample_space=None*, *mode='train'*)

Returns a MagNet DataLoader that iterates over the dataset.

Parameters

- **batch_size** (*int*) – How many samples per batch to load. Default: 1
- **shuffle** (*bool*) – Set to True to have the data reshuffled at every epoch. Default: False
- **replace** (*bool*) – If True every datapoint can be resampled per epoch. Default: False
- **probabilities** (*list or numpy.ndarray*) – An array of probabilities of drawing each member of the dataset. Default: None
- **sample_space** (*float or int or list*) – The fraction / length / indices of the sample to draw from. Default: None
- **mode** (*str*) – One of ['train', 'val', 'test']. Default: 'train'

2.2 Core Datasets

magnet.data.core.**MNIST**(*val_split=0.2*, *path=PosixPath('/home/docs/.data')*, ***kwargs*)

The MNIST Dataset.

Parameters

- **val_split** (*float*) – The fraction of training data to hold out as validation if validation set is not given. Default: 0.2
- **path** (*pathlib.Path or str*) – The path to save the dataset to. Default: Magnet Datapath

Keyword Arguments **()** – See Data for more details.

2.3 Transforms

magnet.data.transforms.**augmented_image_transforms**(*d=0*, *t=0*, *s=0*, *sh=0*, *ph=0*, *pv=0*, *resample=2*)

Returns a list of augmented transforms to be applied to natural images.

Parameters

- **d** (*sequence or float or int*) – Range of degrees to select from. Default: 0
- **t** (*tuple*) – Tuple of maximum absolute fraction for horizontal and vertical translations. Default: 0
- **s** (*tuple, optional*) – Scaling factor interval. Default: 0
- **sh** (*sequence or float or int, optional*) – Range of shear. Default: 0
- **ph** (*float*) – The probability of flipping the image horizontally. Default: 0

- **pv** (*float*) – The probability of flipping the image vertically. Default: 0
- **resample** (*int*) – An optional resampling filter. Default: 2

See `torchvision.transforms` for more details.

`magnet.data.transforms.`**`image_transforms`**(*augmentation=0*, *direction='horizontal'*)

Returns a list of transforms to be applied to natural images.

Parameters

- **augmentation** (*float*) – The percentage of augmentation to be applied. Default: 0
- **direction** (*str*) – The direction to flip the image at random. Default: `'horizontal'`

CHAPTER 3

Nodes

3.1 Node

class magnet.nodes.**Node**(**args*, ***kwargs*)

Abstract base class that defines MagNet's Node implementation.

A Node is a 'self-aware Module'. It can dynamically parametrize itself in runtime.

For instance, a Linear Node can infer the input features automatically when first called; a Conv Node can infer the dimensionality (1, 2, 3) of the input automatically.

MagNet's Nodes strive to help the developer as much as possible by finding the right hyperparameter values automatically. Ideally, the developer shouldn't need to define anything except the basic architecture and the inputs and outputs.

The arguments passed to the constructor are stored in a _args attribute as a dictionary.

This is later modified by the build() method which get's automatically called on the first forward pass.

Keyword Arguments **name** (*str*) – Class Name

3.2 Core

class magnet.nodes.**Lambda**(*fn*, ***kwargs*)

Wraps a Node around any function.

Parameters **fn** (*callable*) – The function which gets called in the forward pass

Examples:

```
>>> import magnet.nodes as mn

>>> import torch
```

(continues on next page)

(continued from previous page)

```
>>> model = mn.Lambda(lambda x: x.mean())

>>> model(torch.arange(5, dtype=torch.float)).item()
2.0

>>> def subtract(x, y):
>>>     return x - y

>>> model = mn.Lambda(subtract)

>>> model(2 * torch.ones(1), torch.ones(1)).item()
1.0
```

class magnet.nodes.**Conv**(*c=None*, *k=3*, *p='half'*, *s=1*, *d=1*, *g=1*, *b=True*, *ic=None*, *act='relu'*, *bn=False*, ***kwargs*)

Applies a convolution over an input tensor.

Parameters

- **c** (*int*) – Number of channels produced by the convolution.
- **Default** – Inferred
- **k** (*int or tuple*) – Size of the convolving kernel. Default: 3
- **p** (*int, tuple or str*) – Zero-padding added to both sides
- **the input. Default** (*of*) – `'half'`
- **s** (*int or tuple*) – Stride of the convolution. Default: 1
- **d** (*int or tuple*) – Spacing between kernel elements. Default: 1
- **g** (*int*) – Number of blocked connections from input channels
- **output channels. Default** (*to*) – 1
- **b** (*bool*) – If `True`, adds a learnable bias to the output.
- **Default** – `True`
- **ic** (*int*) – Number of channels in the input image.
- **Default** – Inferred
- **act** (*str or None*) – The activation function to use.
- **Default** – `'relu'`

- `p` can be conveniently used for 'half', 'same' or 'double' padding to half, same or double the image size respectively. The arguments are accordingly inferred at runtime. For 'half' padding, the output channels (if not provided) are set to twice the input channels to make up for the lost information and vice-versa for the double padding. For 'same' padding, the output channels are kept equal to the input channels. In all three cases, the dilation is set to 1 and the stride is modified as required.
- `c` is inferred from the second dimension of the input tensor.
- `act` is set to 'relu' by default unlike the PyTorch implementation where activation functions need to be seperately defined. Take caution to manually set the activation to None, where needed.

Note: The dimensions (1, 2 or 3) of the convolutional kernels are inferred from the corresponding shape of the input tensor.

Note: One can also create multiple Nodes using the convinient multiplication (*) operation.

Multiplication with an integer n, gives n copies of the Node.

Multiplication with a list or tuple of integers, $(c_1, c_2, ..., c_n)$ gives n copies of the Node with $\subset$ set to c_i

Shape: - Input: $(N, C_{in}, *)$ where * is any non-zero number of trailing dimensions. - Output: $(N, C_{out}, *)$

Variables **layer** (*nn.Module*) – The Conv module built from torch.nn

Examples:

```
>>> import torch

>>> from torch import nn

>>> import magnet.nodes as mn
>>> from magnet.utils import summarize

>>> # A Conv layer with 32 channels and half padding
>>> model = mn.Conv(32)

>>> model(torch.randn(4, 16, 28, 28)).shape
torch.Size([4, 32, 14, 14])

>>> # Alternatively, the 32 in the constructor may be omitted
>>> # since it is inferred on runtime.

>>> # The same conv layer with 'double' padding
>>> model = mn.Conv(p='double')

>>> model(torch.randn(4, 16, 28, 28)).shape
torch.Size([4, 8, 56, 56])

>>> layers = mn.Conv() * 3
[Conv(), Conv(), Conv()]

>>> model = nn.Sequential(*layers)
>>> summarize(model)
+-------+------------+----------------------+
| Node  |   Shape    | Trainable Parameters |
+-------+------------+----------------------+
| input | 16, 28, 28 |          0           |
+-------+------------+----------------------+
| Conv  | 32, 14, 14 |        4,640         |
+-------+------------+----------------------+
| Conv  |  64, 7, 7  |        18,496        |
+-------+------------+----------------------+
| Conv  | 128, 4, 4  |        73,856        |
+-------+------------+----------------------+
Total Trainable Parameters: 96,992
```

class magnet.nodes.**Linear**(*o*, *b=True*, *flat=True*, *i=None*, *act='relu'*, *bn=False*, ***kwargs*)

Applies a linear transformation to the incoming tensor

Parameters

- **o** (*int, Required*) – Output dimensions
- **b** (*bool*) – Whether to include a bias term. Default: `True`
- **flat** (*bool*) – Whether to flatten out the input to 2 dimensions.
- **Default** – `True`
- **i** (*int*) – Input dimensions. Default: Inferred
- **act** (*str or None*) – The activation function to use.
- **Default** – `'relu'`
- **bn** (*bool*) – Whether to use Batch Normalization immediately after
- **layer. Default** (*the*) – `False`

- `flat` is used by default to flatten the input to a vector. This is useful, say in the case of CNNs where an 3-D image based output with multiple channels needs to be fed to several dense layers.
- `o` is inferred from the last dimension of the input tensor.
- `act` is set to 'relu' by default unlike the PyTorch implementation where activation functions need to be seperately defined. Take caution to manually set the activation to None, where needed.

Note: One can also create multiple Nodes using the convinient multiplication (*) operation.

Multiplication with an integer n, gives n copies of the Node.

Multiplication with a list or tuple of integers, $(o_1, o_2, ..., o_n)$ gives n copies of the Node with `o` set to o_i

---

Shape:

If `flat` is True

- Input: $(N, *)$ where $*$ means any number of trailing dimensions
- Output: $(N, *)$

Else

- Input: $(N, *, in\_features)$ where $*$ means any number of trailing dimensions
- Output: $(N, *, out\_features)$ where all but the last dimension are the same shape as the input.

Variables **layer** (*nn.Module*) – The Linear module built from torch.nn

Examples:

```
>>> import torch

>>> from torch import nn

>>> import magnet.nodes as mn
>>> from magnet.utils import summarize

>>> # A Linear mapping to 10-dimensional space
```

(continues on next page)

(continued from previous page)

```
>>> model = mn.Linear(10)

>>> model(torch.randn(64, 3, 28, 28)).shape
torch.Size([64, 10])

>>> # Don't flatten the input
>>> model = mn.Linear(10, flat=False)

>>> model(torch.randn(64, 3, 28, 28)).shape
torch.Size([64, 3, 28, 10])

>>> # Make a Deep Neural Network
>>> # Don't forget to turn the activation to None in the final layer
>>> layers = mn.Linear() * (10, 50) + [mn.Linear(10, act=None)]
[Linear(), Linear(), Linear()]

>>> model = nn.Sequential(*layers)
>>> summarize(model)
+------+---------+--------------------+-----------------------------------------
↪---------+
| Node |  Shape  |Trainable Parameters|                  Arguments
↪         |
+------+---------+--------------------+-----------------------------------------
↪---------+
|input |3, 28, 28|         0          |
↪         |
+------+---------+--------------------+-----------------------------------------
↪---------+
|Linear|   10    |       23,530       |bn=False, act=relu, i=2352, flat=True,
↪b=True, o=10 |
+------+---------+--------------------+-----------------------------------------
↪---------+
|Linear|   50    |        550         |bn=False, act=relu, i=10, flat=True,
↪b=True, o=50   |
+------+---------+--------------------+-----------------------------------------
↪---------+
|Linear|   10    |        510         |bn=False, act=None, i=50, flat=True,
↪b=True, o=10   |
+------+---------+--------------------+-----------------------------------------
↪---------+
Total Trainable Parameters: 24,590
```

class magnet.nodes.**RNN**(*h*, *n=1*, *b=False*, *bi=False*, *act='tanh'*, *d=0*, *batch_first=False*, *i=None*, ***kwargs*)

Applies a multi-layer RNN with to an input tensor.

Parameters

- **h** (*int, Required*) – The number of features in the hidden state *h*
- **n** (*int*) – Number of layers. Default: 1
- **b** (*bool*) – Whether to include a bias term. Default: `True`
- **bi** (*bool*) – If `True`, becomes a bidirectional RNN.
- **Default** – `False`
- **act** (*str or None*) – The activation function to use.

- **Default** – 'tanh'
- **d** (*int*) – The dropout probability of the outputs of each layer.
- **Default** – 0
- **batch_first** (*False*) – If True, then the input and output
- **are provided as `** (*tensors*) – False
- **i** (*int*) – Input dimensions. Default: Inferred

- i is inferred from the last dimension of the input tensor.

Note: One can also create multiple Nodes using the convinient multiplication (*) operation.

Multiplication with an integer n, gives n copies of the Node.

Multiplication with a list or tuple of integers, $(h_1, h_2, ..., h_n)$ gives n copies of the Node with h set to h_i

---

Variables **layer** (*nn.Module*) – The RNN module built from torch.nn

Examples:

```
>>> import torch

>>> from torch import nn

>>> import magnet.nodes as mn
>>> from magnet.utils import summarize

>>> # A recurrent layer with 32 hidden dimensions
>>> model = mn.RNN(32)

>>> model(torch.randn(7, 4, 300))[0].shape
torch.Size([7, 4, 32])

>>> # Attach a linear head
>>> model = nn.Sequential(model, mn.Linear(1000, act=None))
```

class magnet.nodes.**LSTM**(*h*, *n=1*, *b=False*, *bi=False*, *d=0*, *batch_first=False*, *i=None*, ***kwargs*)
Applies a multi-layer LSTM with to an input tensor.

See mn.RNN for more details

class magnet.nodes.**GRU**(*h*, *n=1*, *b=False*, *bi=False*, *d=0*, *batch_first=False*, *i=None*, ***kwargs*)
Applies a multi-layer GRU with to an input tensor.

See mn.RNN for more details

class magnet.nodes.**BatchNorm**(*e=1e-05*, *m=0.1*, *a=True*, *track=True*, *i=None*, ***kwargs*)
Applies Batch Normalization to the input tensor e=1e-05, m=0.1, a=True, track=True, i=None

Parameters

- **e** (*float*) – A small value added to the denominator
- **numerical stability. Default** (*for*) – 1e-5
- **m** (*float or None*) – The value used for the running_mean

- **running_var computation. Can be set to None for** (*and*) –
- **moving average** (*cumulative*) – 0.1
- **a** (*bool*) – Whether to have learnable affine parameters.
- **Default** – `True`
- **track** (*bool*) – Whether to track the running mean and variance.
- **Default** – `True`
- **i** (*int*) – Input channels. Default: Inferred

- `i` is inferred from the second dimension of the input tensor.

Note: The dimensions (1, 2 or 3) of the running mean and variance are inferred from the corresponding shape of the input tensor.

Note: One can also create multiple Nodes using the convinient multiplication (*) operation.

Multiplication with an integer n, gives n copies of the Node.

Multiplication with a list or tuple of integers, $(i_1, i_2, ..., i_n)$ gives n copies of the Node with `i` set to i_i

---

Shape:

- Input: $(N, C, *)$ where $*$ means any number of trailing dimensions
- Output: $(N, C, *)$ (same shape as input)

Variables **layer** (*nn.Module*) – The BatchNorm module built from torch.nn

Examples:

```
>>> import torch

>>> from torch import nn

>>> import magnet.nodes as mn
>>> from magnet.utils import summarize

>>> # A Linear mapping to 10-dimensional space
>>> model = mn.Linear(10)

>>> model(torch.randn(64, 3, 28, 28)).shape
torch.Size([64, 10])

>>> # Don't flatten the input
>>> model = mn.Linear(10, flat=False)

>>> model(torch.randn(64, 3, 28, 28)).shape
torch.Size([64, 3, 28, 10])

>>> # Make a Deep Neural Network
>>> # Don't forget to turn the activation to None in the final layer
>>> layers = mn.Linear() * (10, 50) + [mn.Linear(10, act=None)]
```

(continues on next page)

(continued from previous page)

```
[Linear(), Linear(), Linear()]

>>> model = nn.Sequential(*layers)
>>> summarize(model)
+------+---------+--------------------+-----------------------------------------
↪---------+
| Node |  Shape  |Trainable Parameters|                   Arguments
↪         |
+------+---------+--------------------+-----------------------------------------
↪---------+
|input |3, 28, 28|         0          |
↪         |
+------+---------+--------------------+-----------------------------------------
↪---------+
|Linear|   10    |       23,530       |bn=False, act=relu, i=2352, flat=True,␣
↪b=True, o=10 |
+------+---------+--------------------+-----------------------------------------
↪---------+
|Linear|   50    |        550         |bn=False, act=relu, i=10, flat=True,␣
↪b=True, o=50   |
+------+---------+--------------------+-----------------------------------------
↪---------+
|Linear|   10    |        510         |bn=False, act=None, i=50, flat=True,␣
↪b=True, o=10   |
+------+---------+--------------------+-----------------------------------------
↪---------+
Total Trainable Parameters: 24,590
```

CHAPTER 4

magnet.training

4.1 Trainer

class magnet.training.**Trainer**(*models*, *optimizers*)

Abstract base class for training models.

The Trainer class makes it incredibly simple and convinient to train, monitor, debug and checkpoint entire Deep Learning projects.

Simply define your training loop by implementing the *optimize()* method.

Parameters

- **models** (list of nn.Module) – All the models that need to be trained
- **optimizers** (list of optim.Optimizer) – Any optimizers that are used

---

Note: If any model is in eval() model, the trainer is *set off*. This means that as per protocol, *all* models will not train.

Variables ***callbacks*** (*list*) – A list of callbacks attached to the trainer.

Take a look at *SupervisedTrainer* for an idea on how to extend this class.

optimize()

Defines the core optimization loop. This method is called on each iteration.

Two quick protocols that one needs to follow are:

1. **Do NOT** actually backpropagate or step() the optimizers if the trainer is not training. Use the is_training() method to find out. This is essential since this will ensure that the trainer behaves as expected when is_training() is False. Useful, for example, in cases like *callbacks.ColdStart*

2. Send a callback the signal 'gradient' with a keyword argument 'models' that is the list of models that accumulate a gradient. Usually, it's all the modules (self.modules).

Any callbacks that listen to this signal are interested in the gradient information (eg. `callbacks.Babysitter`).

train (*dataloader*, *epochs=1*, *callbacks=[]*, ***kwargs*)
Starts the training process.

Parameters

- **dataloader** (`DataLoader`) – The MagNet dataloader that iterates over the training set
- **epochs** (*float or int*) – The number of epochs to train for. Default: `1`
- **callbacks** (*list*) – Any callbacks to be attached. Default: []

Keyword Arguments **iterations** (*int*) – The number of iterations to train for

:keyword Overrides *epochs*.:

Note: PyTorch `DataLoader` s are not supported.

Ideally, encapsulate your dataset in the `Data` class.

---

mock (*path=None*)
A context manager that creates a temporary *'safe'* scope for training.

All impact to stateful objects (models, optimizers and the trainer itself) are forgotten once out of this scope.

This is very useful if you need to try out *what-if experiments*.

Parameters **path** (*pathlib.Path*) – The path to save temporary states into Default: {System temp directory}/.mock_trainer

epochs (*mode=None*)
The number of epochs completed.

Parameters **mode** (*str or None*) – If the mode is `'start'` or `'end'`, a boolean is returned signalling if it's the start or end of an epoch

register_parameter (*name*, *value*)
Use this to register *'stateful'* parameters that are serialized

4.2 SupervisedTrainer

class `magnet.training.`**SupervisedTrainer** (*model*, *optimizer='adam'*, *loss='cross_entropy'*, *metrics=[]*)
A simple trainer that implements a supervised approach where a simple model $\hat{y} = f(x)$ is trained to map $\hat{y}$ to ground-truth y according to some specified loss.

This is the training routine that most high-level deep learning frameworks implement.

Parameters

- **model** (`nn.Module`) – The model that needs to be trained
- **optimizer** (*str or optim.Optimzer*) – The optimizer used to train the model. Default: `'adam'`
- **loss** (str or `callable`) – A loss function that gives the objective to be minimized. Default: `'cross_entropy'`

- **metrics** (*list*) – Any other metrics that need to be monitored. Default: `[]`

- `optimizer` can be an actual `optim.Optimizer` instance or the name of a popular optimzizer (eg. `'adam'`).
- `loss` can be a function or the name of a popular loss function (eg. `'cross_entropy'`). It should accept 2 arguments ($\hat{y}$, y).
- `metrics` should contain a list of functions which accept 2 arguments ($\hat{y}$, y), like the loss function.

Note: A static `validate()` function is provided for the validation callback

---

Note: The `metrics` is of no use unless there is some callback (eg.‘‘callbacks.Monitor‘‘) to receive the metrics

---

Examples:

```
>>> import magnet as mag
>>> import magnet.nodes as mn

>>> from magnet.data import Data
>>> from magnet.training import callbacks, SupervisedTrainer

>>> data = Data.get('mnist')

>>> model = mn.Linear(10, act=None)
>>> model.build(x=next(data())[0])

>>> trainer = SupervisedTrainer(model)
>>> callbacks=[callbacks.Monitor(),
               callbacks.Validate(data(64, mode='val'), SupervisedTrainer.
↪validate)]
>>> trainer.train(data(64, shuffle=True), 1, callbacks)
```

`magnet.training.`**`finish_training`**(*path*, *names=None*)

A helper function for cleaning up the training logs and other checkpoints and retaining only the state_dicts of the trained models.

Parameters

- **path** (*pathlib.Path*) – The path where the trainer was checkpointed
- **names** (*list*) – The names of the models in the order given to the trainer. Default: None

- `names` can be used if the models themselves did not have names prior to training. The checkpoints default to an ordered naming scheme. If passed, the files are additionally renamed to these names.

Note: Does nothing / fails silently if the path does not exist.

Example:

```
>>> # Assume that we've defined two models - encoder and decoder,
>>> # and a suitable trainer. The models do not have a 'name' attribute.
```

(continues on next page)

(continued from previous page)

```
>>> trainer.save_state(checkpoint_path / 'my-trainer')

>>> # Suppose the checkpoint directory contains the following files:
>>> # my-trainer/
>>> #     models/
>>> #         0.pt
>>> #         1.pt
>>> #     callbacks/
>>> #         monitor/
>>> #         babysitter/
>>> #     state.p

>>> finish_training(path, names=['encoder', 'decoder'])

>>> # Now the directory contains these files:
>>> # encoder.pt
>>> # decoder.pt
```

CHAPTER 5

magnet.training.callbacks

5.1 CallbackQueue

class magnet.training.callbacks.**CallbackQueue**

A container for multiple callbacks that can be called in parallel.

If multiple callbacks need to be called together (as intended), they can be registered via this class.

Since callbacks need to be unique (by their name), this class also ensures that there are no duplicates.

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

Broadcasts a signal to all registered callbacks along with payload arguments.

Parameters **signal** (*object*) – Any object that is broadcast as a signal.

Note: Any other arguments will be sent as-is to the callbacks.

find(*name*)

Scans through the registered list and finds the callback with `name`.

If not found, returns None.

Raises `RuntimeError` – If multiple callbacks are found.

5.2 Monitor

class magnet.training.callbacks.**Monitor**(*frequency=10*, *show_progress=True*, ***kwargs*)

Allows easy monitoring of the training process.

Stores any metric / quantity broadcast using the `'write_stats'` signal.

Also adds a nice progress bar!

Parameters

- **frequency** (*int*) – Then number of times per epoch to flush the buffer. Default: 10
- **show_progress** (*bool*) – If `True`, adds a progress bar. Default: `True`

Keyword Arguments **name** (*str*) – Name of this callback. Default: `'monitor'`

- `frequency` is useful only if there are buffered metrics.

Examples:

```
>>> import torch

>>> import magnet as mag
>>> import magnet.nodes as mn

>>> from magnet.training import callbacks, SupervisedTrainer

>>> model = mn.Linear(10, act=None)
>>> with mag.eval(model): model(torch.randn(4, 1, 28, 28))

>>> trainer = SupervisedTrainer(model)

>>> callbacks = callbacks.CallbackQueue([callbacks.Monitor()])
>>> callbacks(signal='write_stats', trainer=trainer, key='loss', value=0.1)

>>> callbacks[0].history
{'loss': [{'val': 0.1}]}
```

__call__(*trainer*, *signal*, ***kwargs*)

Responds to the following signals:

- `'write_stats'`: Any keyword arguments will be passed to the `History.append()` method.
- `'on_training_start'`: To be called before start of training. Initializes the progress bar.
- `'on_batch_start'`: Called before the training loop. Updates the progress bar.
- `'on_batch_end'`: Called after the training loop. Flushes the history buffer if needed and sets the progress bar description.
- `'on_training_end'`: To be called after training. Closes the progress bar.
- `'load_state'`: Loads the state of this callback from `path`.
- `'save_state'`: Saves the state of this callback to `path`.

show(*metric=None*, *log=False*, *x_key='epochs'*, ***kwargs*)

Calls the corresponding `History.show()` method.

5.3 Validate

class magnet.training.callbacks.**Validate**(*dataloader*, *validate*, *frequency=10*, *batches=None*, *drop_last=False*, ***kwargs*)

Runs a validation function over a dataset during the course of training.

Most Machine Learning research uses a held out validation set as a proxy for the test set / real-life data. Hyper-parameters are usually tuned on the validation set.

Often, this is done during training in order to view the simultaneous learning on the validation set and catch any overfitting / underfitting.

This callback enables you to run a custom :py:meth‘validate‘ function over a `dataloader`.

Parameters

- **`dataloader`** (`DataLoader`) – DataLoader containing the validation set
- **`validate`** (*`bool`*) – A callable that does the validation
- **`frequency`** (*`int`*) – Then number of times per epoch to run the function. Default: 10
- **`batches`** (*`int or None`*) – The number of times / batches to call the validate function in each run. Default: `None`
- **`drop_last`** (*`bool`*) – If `True`, the last batch is not run. Default: `False`

Keyword Arguments **`name`** (*`str`*) – Name of this callback. Default: `'validate'`

- `validate` is a function which takes two arguments: (trainer, dataloader).
- `batches` defaults to a value which ensures that an epoch of the validation set matches an epoch of the training set.

  For instance, if the training set has 80 datapoints and the validation set has 20 and the batch size is 1 for both, an epoch consists of 80 iterations for the training set and 20 for the validation set.

  If the validate function is run 10 times(`frequency`) per epoch of the training set, then `batches` must be 2.

`__call__`(*trainer*, *signal*, ***kwargs*)
Responds to the following signals:

- `'on_training_start'`: To be called before start of training. Automatically finds the number of batches per run.
- `'on_batch_end'`: Called after the training loop. Calls the `validate` function.
- `'on_training_end'`: To be called after training. If `drop_last`, calls the `validate` function.
- `'load_state'`: Loads the state of this callback from `path`.
- `'save_state'`: Saves the state of this callback to `path`.

5.4 Checkpoint

`class` `magnet.training.callbacks.`**`Checkpoint`**(*path*, *interval='5 m'*, ***kwargs*)
Serializes stateful objects during the training process.

For many practical Deep Learning projects, training takes many hours, even days.

As such, it is only natural that you’d want to save the progress every once in a while.

This callback saves the models, optimizers, schedulers and the trainer itself periodically and automatically loads from those states if found.

Parameters

- **`path`** (*`pathlib.Path`*) – The root path to save to
- **`interval`** (*`str`*) – The time between checkpoints. Default: ‘5 m’

Keyword Arguments **`name`** (*`str`*) – Name of this callback. Default: `'checkpoint'`

- `interval` should be a string of the form `'{duration} {unit}'`. Valid units are: `'us'` (microseconds), `'ms'` (milliseconds), `'s'` (seconds), `'m'` (minutes)', `'h'` (hours), `'d'` (days).

__call__(*trainer*, *signal*, ***kwargs*)
Responds to the following signals:

- `'on_training_start'`: To be called before start of training. Creates the path if it doesn't exist and loads from it if it does. Also sets the starting time.
- `'on_batch_end'`: Called after the training loop. Checkpoints if the interval is crossed and resets the clock.
- `'on_training_end'`: To be called after training. Checkpoints one last time.
- `'load_state'`: Loads the state of this callback from `path`.
- `'save_state'`: Saves the state of this callback to `path`.

5.5 ColdStart

class magnet.training.callbacks.**ColdStart**(*epochs=0.1*, ***kwargs*)
Starts the trainer in `eval` mode for a few iterations.

Sometimes, you may want to find out how the model performs prior to any training. This callback freezes the training initially.

Parameters **epochs** (*float*) – The number of epochs to freeze the trainer. Default: 0.1

Keyword Arguments **name** (*str*) – Name of this callback. Default: `'coldstart'`

__call__(*trainer*, *signal*, ***kwargs*)
Responds to the following signals:

- `'on_training_start'`: To be called before start of training. Sets the models in `eval` mode.
- `'on_batch_end'`: Called after the training loop. If the `epochs` is exhausted, unfreezes the trainer and removes this callback from the queue.

5.6 LRScheduler

class magnet.training.callbacks.**LRScheduler**(*scheduler*, ***kwargs*)
A helper callback to add in optimizer schedulers.

Parameters **scheduler** (`LRScheduler`) – The scheduler.

Keyword Arguments **name** (*str*) – Name of this callback. Default: `'lr_scheduler'`

__call__(*trainer*, *signal*, ***kwargs*)
Responds to the following signals:

- `'on_batch_start'`: Called before the training loop. If it is the start of an epoch, steps the scheduler.

CHAPTER 6

magnet.training.history

CHAPTER 7

magnet.training.utils

magnet.training.utils.**load_object**(*path*, ***kwargs*)
A convinience method to unpickle a file.

Parameters **path** (*pathlib.Path*) – The path to the pickle file

Keyword Arguments **default** (*object*) – A default value to be returned if the file does not exist. Default: `None`

Raises `RuntimeError` – If a default keyword argument is not provided and the file is not found.

magnet.training.utils.**load_state**(*module*, *path*, *alternative_name=None*)
Loads the state_dict of a PyTorch object from a specified path.

This is a more robust version of the of the PyTorch way in the sense that the device mapping is automatically handled.

Parameters

- **module** (*object*) – Any PyTorch object that has a state_dict
- **path** (*pathlib.Path*) – The path to folder containing the state_dict file
- **alternative_name** (*str or None*) – A fallback name for the file if the module object does not have a name attribute. Default: `None`

Raises `RuntimeError` – If no `alternative_name` is provided and the module does not have a name.

Note: If you already know the file name, set `alternative_name` to that.

This is just a convinience method that assumes that the file name will be the same as the name of the module (if there is one).

---

magnet.training.utils.**save_object**(*obj*, *path*)
A convinience method to pickle an object.

Parameters

- **obj** (*object*) – The object to pickle
- **path** (*pathlib.Path*) – The path to save to

Note: If the `path` does not exists, it is created.

`magnet.training.utils.`**save_state**(*module*, *path*, *alternative_name=None*)

Saves the state_dict of a PyTorch object to a specified path.

Parameters

- **module** (*object*) – Any PyTorch object that has a state_dict
- **path** (*pathlib.Path*) – The path to a folder to save the state_dict to
- **alternative_name** (*str or None*) – A fallback name for the file if the module object does not have a name attribute. Default: `None`

Raises `RuntimeError` – If no `alternative_name` is provided and the module does not have a name.

CHAPTER 8

Debugging

magnet.debug.**overfit**(*trainer*, *data*, *batch_size*, *epochs=1*, *metric='loss'*, ***kwargs*)

Runs training on small samples of the dataset in order to overfit.

If you can't overfit a small sample, you can't model the data well.

This debugger tries to overfit on multple small samples of the data. The sample size and batch sizes are varied and the training is done for a fixed number of epochs.

This usually gives an insight on what to expect from the actual training.

Parameters

- **trainer** (*magnet.trainer.Trainer*) – The Trainer object
- **data** (magnet.data.Data) – The data object used for training
- **batch_size** (*int*) – The intended batch size
- **epochs** (*float*) – The expected epochs for convergence for 1% of the data. Default: `1`
- **metric** (*str*) – The metric to plot. Default: `'loss'`

Note: The maximum sample size is 1% of the size of the dataset.

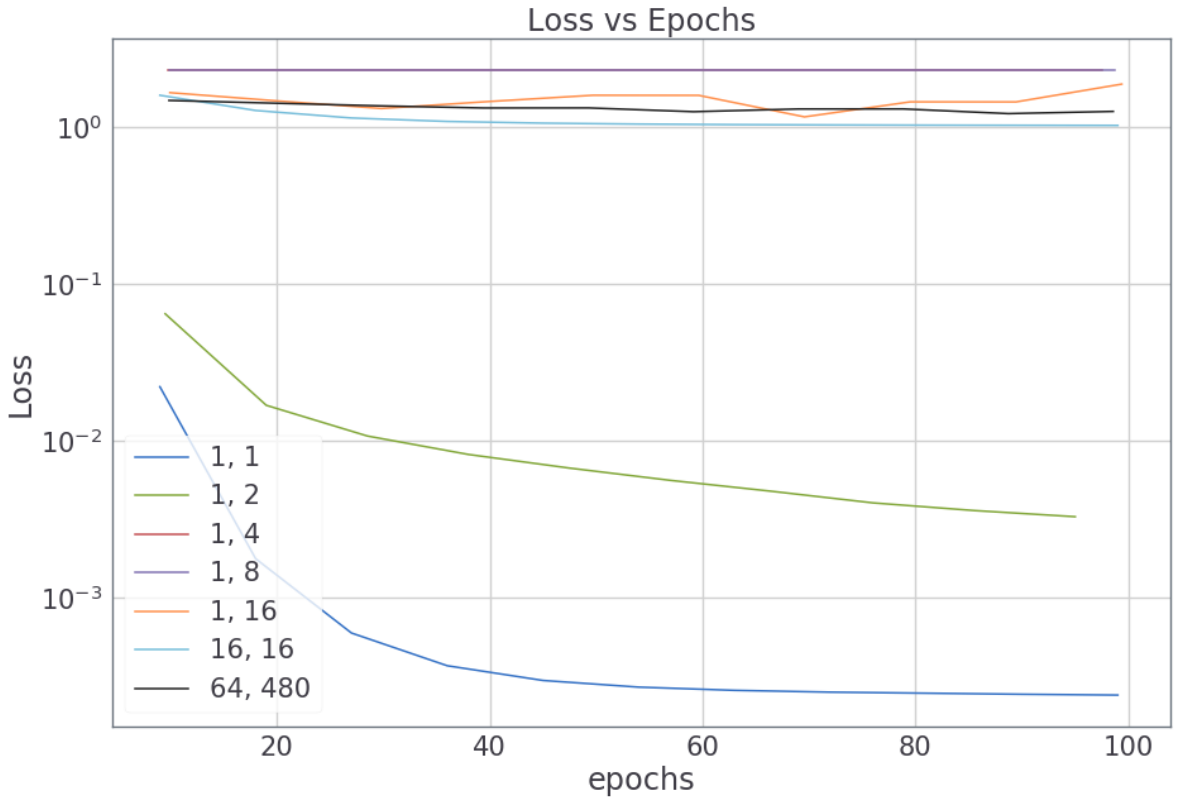
Examples:

```
>>> import magnet as mag
>>> import magnet.nodes as mn
>>> import magnet.debug as mdb

>>> from magnet.data import Data
>>> from magnet.training import SupervisedTrainer

>>> data = Data.get('mnist')

>>> model = mn.Linear(10)
```

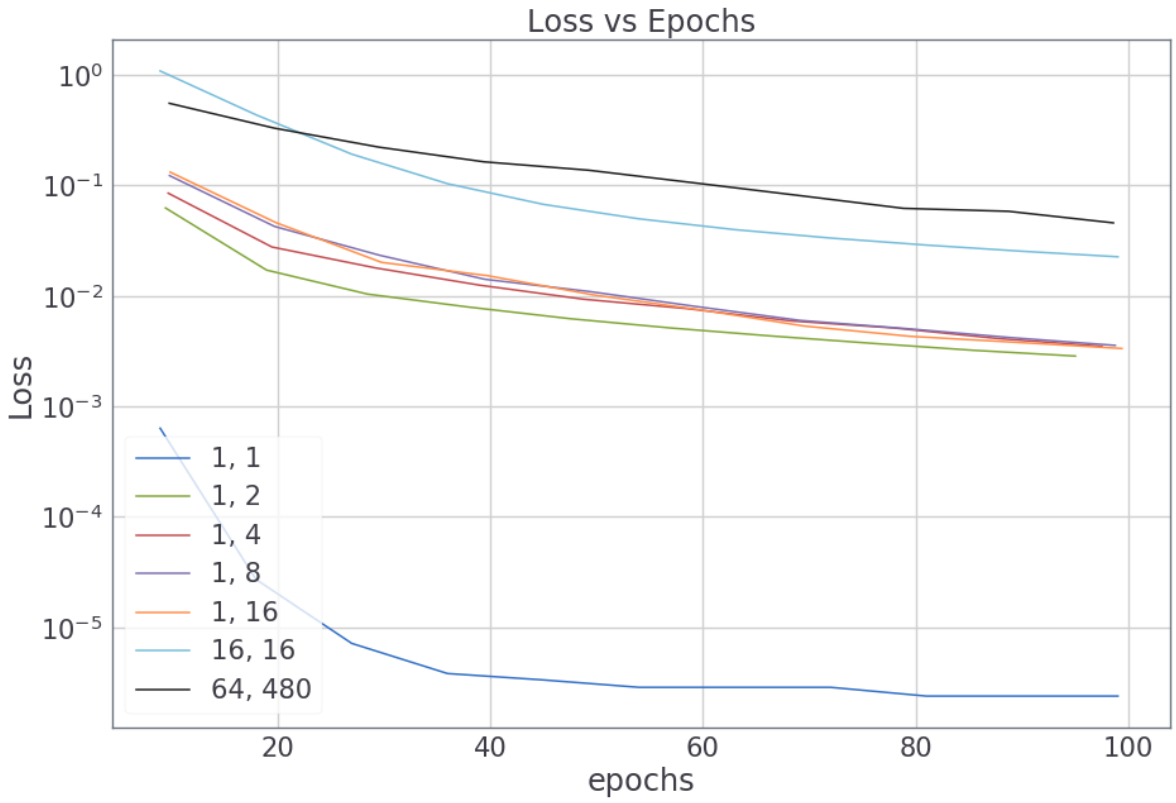
(continues on next page)

(continued from previous page)

```
>>> with mag.eval(model): model(next(data())[0])

>>> trainer = SupervisedTrainer(model)

>>> mdb.overfit(trainer, data, batch_size=64)
```




```
>>> # Oops! Looks like there was something wrong.
>>> # Loss does not considerable decrease for samples sizes >= 4.
>>> # Of course, the activation was 'relu'.
>>> model = mn.Linear(10, act=None)
>>> with mag.eval(model): model(next(data())[0])

>>> trainer = SupervisedTrainer(model)

>>> mdb.overfit(trainer, data, batch_size=64)
>>> # Should be much better now.
```



`magnet.debug.`**`check_flow`**(*trainer*, *data*)

Checks if any trainable parameter is not receiving gradients.

Super useful for large architectures that use the .detach() function.

Parameters

- **`trainer`** (*`magnet.trainer.Trainer`*) – The Trainer object
- **`data`** (`magnet.data.Data`) – The data object used for training

`class` `magnet.debug.`**`Babysitter`**(*frequency=10*, ***kwargs*)

A callback which monitors the mean relative gradients for all parameters.

Parameters **`frequency`** (*`int`*) – Then number of times per epoch to monitor. Default: 10

Keyword Arguments **`name`** (*`str`*) – Name of this callback. Default: `'babysitter'`

`magnet.debug.`**`shape`**(*debug=True*)

The shapes of every tensor is printed out if a module is called within this context manager.

Useful for debugging the flow of tensors through layers and finding the values of various hyperparameters.

Parameters **`debug`** (*`bool or str`*) – If `str`, only the tensor with this name is tracked. If `True`, all tensors are tracked. Else, nothing is tracked.

CHAPTER 9

magnet.utils

magnet.utils.**summarize**(*module*, *x*, *parameters='trainable'*, *arguments=False*, *batch=False*, *max_width=120*)

Prints a pretty picture of how a one-input one output sequential model works.

Similar to `Model.summarize` found in Keras.

Parameters

- **module** (`nn.Module`) – The module to summarize
- **x** (`torch.Tensor`) – A sample tensor sent as input to the `module`.
- **parameters** (*`str or True`*) – Which kind of parameters to enumerate. Default: `'trainable'`
- **arguments** (*`bool`*) – Whether to show the arguments to a node. Default: `False`
- **batch** (*`bool`*) – Whether to show the batch dimension in the shape. Default: `False`
- **max_width** (*`int`*) – The maximum width of the table. Default: `120`

- `parameters` is one of ['trainable', 'non-trainable', 'all', True].

  'trainable' parameters are the ones which require gradients and can be optimized by SGD.

  Setting this to `True` will print both types as a tuple.

CHAPTER 10

magnet.utils.images

CHAPTER 11

magnet.utils.plot

CHAPTER 12

magnet.utils.varseq

magnet.utils.varseq.**pack**(*sequences*, *lengths=None*)
Packs a list of variable length Tensors

Parameters

- **sequences** (*list or torch.Tensor*) – The list of Tensors to pack
- **lengths** (*list*) – list of lengths of each tensor. Default: None

Note: If sequences is a tensor, lengths needs to be provided.

Note: The packed sequence that is returned has a convinient *unpack()* method as well as shape and order attributes. The order attribute stores the sorting order which should be used for unpacking.

Shapes: sequences should be a list of Tensors of size L x *, where L is the length of a sequence and * is any number of trailing dimensions, including zero.

magnet.utils.varseq.**unpack**(*sequence*, *as_list=False*)
Unpacks a PackedSequence object.

Parameters

- **sequence** (PackedSequence) – The tensor to unpack.
- **as_list** (*bool*) – If True, returns a list of tensors. Default: False

Note: The sequence should have an order attribute that stores the sorting order.

magnet.utils.varseq.**sort**(*sequences*, *order*, *dim=0*)
Sorts a tensor in a certain order along a certain dimension.

Parameters

- **sequences** (*torch.Tensor*) – The tensor to sort
- **order** (*numpy.ndarray*) – The sorting order
- **dim** (*int*) – The dimension to sort. Default `0`

magnet.utils.varseq.**unsort**(*sequences*, *order*, *dim=0*)

Unsorts a tensor in a certain order along a certain dimension.

Parameters

- **sequences** (*torch.Tensor*) – The tensor to unsort
- **order** (*numpy.ndarray*) – The sorting order
- **dim** (*int*) – The dimension to unsort. Default `0`

CHAPTER 13

Indices and tables

- genindex
- modindex

Python Module Index

m

Index

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

V