scaling-spoon

Release 1.0

Kaan Oktay, Jonas Frey, Raffael Theiler

CONTENTS:

1.2 cifar100.py 1.3 configloader.py 1.4 autoencoder.py 1.5 fcview.py 1.6 interpolate.py 1.7 maps.py 1.8 network_stack.py 1.9 stackable_network.py 1.10 vgg_autoencoder.py 1.11 vgg.py 1.12 encodernet.py 1.13 cfg_to_network.py		Indic	es and tables
1.2 cifar100.py 1.3 configloader.py 1.4 autoencoder.py 1.5 fcview.py 1.6 interpolate.py 1.7 maps.py 1.8 network_stack.py 1.9 stackable_network.py 1.10 vgg_autoencoder.py 1.11 vgg.py 1.12 encodernet.py 1.13 cfg_to_network.py 1.14 layer_training_def.py		1.1	cifar10.py
1.3 configloader.py 1.4 autoencoder.py 1.5 fcview.py 1.6 interpolate.py 1.7 maps.py 1.8 network_stack.py 1.9 stackable_network.py 1.10 vgg_autoencoder.py 1.11 vgg.py 1.12 encodernet.py 1.13 cfg_to_network.py 1.14 layer_training_def.py		1.2	cifar100.py
1.5 fcview.py 1.6 interpolate.py 1.7 maps.py 1.8 network_stack.py 1.9 stackable_network.py 1.10 vgg_autoencoder.py 1.11 vgg.py 1.12 encodernet.py 1.13 cfg_to_network.py 1.14 layer_training_def.py		1.3	configloader.py
1.6 interpolate.py 1.7 maps.py 1.8 network_stack.py 1.9 stackable_network.py 1.10 vgg_autoencoder.py 1.11 vgg.py 1.12 encodernet.py 1.13 cfg_to_network.py 1.14 layer_training_def.py		1.4	autoencoder.py
1.7 maps.py . 1.8 network_stack.py . 1.9 stackable_network.py 1.10 vgg_autoencoder.py . 1.11 vgg.py . 1.12 encodernet.py . 1.13 cfg_to_network.py . 1.14 layer_training_def.py .		1.5	fcview.py
1.8 network_stack.py 1.9 stackable_network.py 1.10 vgg_autoencoder.py 1.11 vgg.py 1.12 encodernet.py 1.13 cfg_to_network.py 1.14 layer_training_def.py		1.6	interpolate.py
1.9 stackable_network.py 1.10 vgg_autoencoder.py 1.11 vgg.py 1.12 encodernet.py 1.13 cfg_to_network.py 1.14 layer_training_def.py		1.7	maps.py
1.10vgg_autoencoder.py1.11vgg.py1.12encodernet.py1.13cfg_to_network.py1.14layer_training_def.py		1.8	network_stack.py
1.11 vgg.py		1.9	stackable_network.py
1.12 encodernet.py		1.10	vgg_autoencoder.py
1.13 cfg_to_network.py		1.11	vgg.py
1.14 layer_training_def.py		1.12	encodernet.py
1.14 layer_training_def.py		1.13	cfg_to_network.py
Python Module Index		1.14	layer_training_def.py
Python Module Index			
	yt]	hon N	Module Index
Index	ıd	ex	

CHAPTER

ONE

INDICES AND TABLES

- genindex
- · modindex
- · search

1.1 cifar10.py

Dataset wrappers to use cifar10 in a semi-supervised fashion.

loaders.cifar10.	This class randomly permutes a set of indicies of the
CifarSubsetSampler(indices)	cifar10 dataset.
loaders.cifar10.	Provides the three dataloaders:
semi_supervised_cifar10()	

```
class loaders.cifar10.CifarSubsetSampler(indices)
```

Bases: torch.utils.data.sampler.Sampler

This class randomly permutes a set of indicies of the cifar10 dataset. We need this sampler to restrict the dataset class to only provide permuted batches of data from one one the strictly separated supervised and unsupervised sets.

Provides the three dataloaders:

- Testset Dataloader
- Training Set Dataloader (Supervised Dataset)
- Training Set Dataloader (Unsupervised Dataset)

It wraps the pytorch cifar10 dataset class.

1.2 cifar100.py

Dataset wrappers to use cifar100 in a semi-supervised fashion.

loaders.cifar100. CifarSubsetSampler(indices)	This class randomly permutes a set of indicies of the cifar100 dataset.
loaders.cifar100.	Provides the three dataloaders:
semi_supervised_cifar100()	

```
class loaders.cifar100.CifarSubsetSampler(indices)
```

Bases: torch.utils.data.sampler.Sampler

This class randomly permutes a set of indicies of the cifar 100 dataset. We need this sampler to restrict the dataset class to only provide permuted batches of data from one one the strictly separated supervised and unsupervised sets.

```
loaders.cifar100.semi_supervised_cifar100 (root, transformation_id: str, supervised_ratio=0.1, batch_size=128, download=False, augmentation=False, num_workers=(6, 6, 2)) \rightarrow Tuple[torch.utils.data.dataloader.DataLoader, torch.utils.data.dataloader.DataLoader, torch.utils.data.dataloader.DataLoader, torch.utils.data.dataloader.DataLoader]
```

Provides the three dataloaders:

- Testset Dataloader
- Training Set Dataloader (Supervised Dataset)
- Training Set Dataloader (Unsupervised Dataset)

It wraps the pytorch cifar 100 dataset class.

1.3 configloader.py

A yaml config loader.

loaders.ConfigLoader	This class implements different methods to load / write
	data from and to a yaml file.

```
class loaders.configloader.ConfigLoader
```

Bases: object

This class implements different methods to load / write data from and to a yaml file. It is used to serialize / load all the network hyperparameter.

```
from_file (env_path: str = 'yaml/env.yml', suppress_print=False) → load-
ers.configloader.ConfigLoader

from_string (data: str) → loaders.configloader.ConfigLoader

switch (key: str, options: Dict[str, Callable])

to_json()
```

1.4 autoencoder.py

First implementation of a horizontal (layerwise) training mechanism of a deep network. It is fundamentally flawed because the receptive field of this stack does not increase in depth and therefore it is not possible for the network to learn complex features in images.

This happens because the upstream maps the data back to the input size of this autoencoder to make it stackable.

Replaced by SidecarAutoencoder which trains a narrowing network that does not have these problems. We keep the code for reasoning / documentation of our mistakes.

```
modules.Autoencoder([color_channels, Implementation of an autoencoder used in our stack. dropout])
modules.SupervisedAutoencoder(color_channels) supervised extension to the autoencoder class.
```

Implementation of an autoencoder used in our stack. This is a horizontal module. It is connected to the upper layer via its upstream function.

```
calculate\_upstream(x)
```

computes the representation of x used in the next layer

```
class modules.autoencoder.SupervisedAutoencoder(color_channels: int)
```

Bases: modules.autoencoder.Autoencoder

A supervised extension to the autoencoder class. It extends autoencoder with a fully connected module linked to the autoencoder bottleneck.

1.5 fcview.py

```
modules.FCView()
                                             Pytorch view abstraction as nn.module
class modules.fcview.FCView
    Bases: torch.nn.modules.module.Module
    Pytorch view abstraction as nn.module
Experimental class not used in the final product
class modules.inheritance.Autoencoder(color_channels=3)
    Bases: object
    decoder = None
class modules.inheritance.StackableNetwork
    Bases: object
    abstract_method()
class modules.inheritance.SupervisedAutoencoder(color_channels)
    Bases: modules.inheritance.Autoencoder, modules.inheritance.StackableNetwork
    abstract_method()
    supervision = None
```

1.5. fcview.py 3

test = 1

1.6 interpolate.py

modules.Interpolate([scale_factor, mode])	Interpolate function used for inverse convolution in De-
	coders

class modules.interpolate.Interpolate(scale_factor=2, mode='nearest')

Bases: torch.nn.modules.module.Module

Interpolate function used for inverse convolution in Decoders

1.7 maps.py

The term map in this project is a nn.module that maps data from one horizontal layer to the next. It is the abstract concept of coupling such individually trained layers into one bigger (deeper) network.

Many different maps are collected in this module. It one can check in the config yaml to see which of these maps is used in a specific training instance.

modules.RandomMap(in_shape, out_shape)	This map randomly permutes the input and maps it to the output shape.
modules.SidecarMap(main_network_layer)	This map takes any set of non-trainable layer from a
	bigger main network and uses it as a map function.
modules.ConvMap(in_shape, out_shape)	Trainable map that is a one layer convolution to map
	from input to output dimension.
modules.DecoderMap(trained_net)	This map takes the last layer of a horizontal autoencoder
	and uses it as an upstream mapping initialization.

class modules.maps.ConvMap(in_shape: Tuple[int, int, int], out_shape: Tuple[int, int, int])
 Bases: torch.nn.modules.module.Module

Trainable map that is a one layer convolution to map from input to output dimension.

 $\verb"class" modules.maps.DecoderMap" (trained_net: modules.autoencoder.Autoencoder)$

Bases: torch.nn.modules.module.Module

This map takes the last layer of a horizontal autoencoder and uses it as an upstream mapping initialization. No scaling is needed since this layer has the exact dimensions we need by design.

Other than that, it is similar to ConvMap.

class modules.maps.RandomMap(in_shape: Tuple[int, int, int], out_shape: Tuple[int, int, int])
 Bases: torch.nn.modules.module.Module

This map randomly permutes the input and maps it to the output shape.

class modules.maps.SidecarMap (main_network_layer: List[torch.nn.modules.module.Module])
 Bases: torch.nn.modules.module.Module

This map takes any set of non-trainable layer from a bigger main network and uses it as a map function.

Mainly used for maxpool layers.

1.8 network_stack.py

modules.NetworkStack(networks[,])	A network stack is responsible to use a set of layers and
	maps to:

Bases: torch.nn.modules.module.Module

A network stack is responsible to use a set of layers and maps to:

- · Compute the upstream to this layer
- Train that one layer horizontally

The class is initialized with a reference to all the previous layers

upwards(x)

1.9 stackable_network.py

Interface class that is used in classes that implement the specification necessary to be used as a horizontal training layer.

modules.StackableNetwork()	Interface, not a Class! Do not implement anything here Classes that inherit from StackableNetwork are re-
	quired, (in python by convention, not enforced) to provide an implementation of these functions.

class modules.stackable_network.StackableNetwork

Bases: object

Interface, not a Class! Do not implement anything here Classes that inherit from StackableNetwork are required, (in python by convention, not enforced) to provide an implementation of these functions.

calculate_upstream (previous_network)

Calculate the output to the point where the map function will be applied after.

Make sure those layers appear in parameters()

1.10 vgg_autoencoder.py

This module contains all autoencoders required to train a VGG network horizontally.

modules.SidecarAutoencoder()	This autoencoder is a "sidecar" to a deeper network.
modules.SupervisedSidecarAutoencoder(.)The supervision extension of the sidecar autoencoder.

```
class modules.vgg_autoencoder.SidecarAutoencoder (main_network_layer:
```

List[torch.nn.modules.module.Module], img_size: int, channels: Tuple[int, int], dropout: float, kernel_size: int, encoder_type: str)

Bases: torch.nn.modules.module.Module

This autoencoder is a "sidecar" to a deeper network. It takes a slice of the main network (main_network_layer) and trains this layer isolated from the other parts of the main network.

Different autoencoder types are supported (A-C).

```
bottleneck_size() \rightarrow int calculate_upstream(x)
```

class modules.vgg_autoencoder.SupervisedSidecarAutoencoder (main_network_layer:

List[torch.nn.modules.module.Module],
img_size: int, channels: Tuple[int,
int], dropout: float,
kernel_size: int,
encoder_type: int,
num_classes: int)

Bases: modules.vgg_autoencoder.SidecarAutoencoder

The supervision extension of the sidecar autoencoder.

1.11 vgg.py

This is the main vertical network of this project. We use / modified the VGG implementation from https://github.com/pytorch/vision/blob/master/torchvision/models/vgg.py

modules.VGG(num_classes, dropout, img_size, ...) VGG CNN Implementation with minor changes such that we can extract all layers to train them individually.

 $Bases: \verb|torch.nn.modules.module|. Module|$

VGG CNN Implementation with minor changes such that we can extract all layers to train them individually.

Layers are trained via sidecar autoencoders. The pooling layers become upstream maps.

 $\label{lem:conduction} \begin{tabular}{ll} \begin{tabular}{ll} \textbf{get_trainable_modules} () &\rightarrow List[Tuple[List[torch.nn.modules.module.Module]], Tuple[int, int], \\ &\quad int, List[torch.nn.modules.module.Module]]] \end{tabular}$

 $\begin{tabular}{ll} \textbf{make_modules} (cfg, img_size, batch_norm: bool) &\to Tuple[List[torch.nn.modules.module.Module], \\ List[Tuple[List[torch.nn.modules.module.Module], & Tuple[int, int], \\ List[torch.nn.modules.module.Module]]]] \end{tabular}$

1.12 encodernet.py

This module combines all the custom network modules and defines the schedule how to train and evaluate our networks.

networks.encodernet.

Main Class of this project.

AutoencoderNet(gcfg, rcfg)

Bases: object

Main Class of this project. Groups all the functions to run a network but is not responsible to assemble the network configuration / hyperparameters. Most of the functions accept a LayerTrainingDefinition or a ConfigLoader to access the training parameters.

This function implements a voting scheme over all layers. It uses a layer-wise softmax prediction. The final decision is taken as argmax over the mean of the individual distributions.

measure_time (t_start)

plot_img (real_imgs, dc_imgs, epoch)

test (epoch: int, global_epoch: int, config: networks.layer_training_def.LayerTrainingDefinition, plot_every_n_epochs=1)

This method is the default test method for a horizontal autoencoder network block. It operates on a given config of type LayerTrainingDefinition. The method freezes the current layer and computes the test accuracy.

test_vgg_classifier(epoch: int, global_epoch: int, config: networks.layer_training_def.LayerTrainingDefinition, plot_every_n_epochs=1)

The vgg network has its own classification layer with a cost function that differs from the usual autoencoder network block. Therefore it has its own test method.

 $to_img(x)$

train (epoch: int, global_epoch: int, config: networks.layer_training_def.LayerTrainingDefinition)

This method is the default train method for a horizontal autoencoder network block. It takes care of dataset iteration and loss calculation / optimization. It operates on a given config of type LayerTrainingDefinition. This code represents what needs to be done in one epoch of training.

train_test()

This function defines the overall training schedule over the total number of epochs. It trains the layer one by one:

- First layer 0 is trained for n epochs
- Then layer 1 is trained for n epochs
- etc...

It is called from the main script after class initialization to start the training.

train_vgg_classifier (epoch: int, global_epoch: int, config: networks.layer_training_def.LayerTrainingDefinition) net-

The vgg network has its own classification layer with a cost function that differs from the usual autoencoder network block. Therefore it has its own training method.

visualize(epoch=0)

Visualize gradients generated with color guided back propagation

wave_train_test()

This function defines the overall training schedule over the total number of epochs. It trains the layer in a wave-like pattern:

- Layer 0 is trained for (total_epochs / waves) epochs
- Layer 1 is trained for (total epochs / waves) epochs

after the final layer is trained, we return back to layer 0.

This method converges much quicker than strict sequential training.

It is called from the main script after class initialization to start the training.

```
networks.encodernet.ensure_dir(path: str)
networks.encodernet.save_layer(layer: torch.nn.modules.module.Module, path: str)
```

1.13 cfg_to_network.py

Module to map from configuration files to classes of our network.

```
networks.cfg_to_network(gcfg, rcfg)
                                                      This is the main network assembly function.
networks.cfg_to_network.cfg_to_network(gcfg:
                                                                   loaders.configloader.ConfigLoader,
                                                              loaders.configloader.ConfigLoader)
                                                    List[networks.layer training def.LayerTrainingDefinition]
     This is the main network assembly function. It takes a config from a configloader (a yaml file). Assembles a
     network stack as list of LayerTrainingDefintions.
     Returns: List[LayerTrainingDefinition]
networks.cfg_to_network.load_layer(layer: torch.nn.modules.module.Module, path: str)
networks.cfq_to_network.vgg_sidecar_layer(vgg:
                                                                 modules.vgg.VGG, index:
                                                                                                 int,
                                                                    float,
                                                                           kernel_size:
                                                        dropout:
                                                                                           int,
                                                                                                 en-
                                                        coder_type:
                                                                      int, num classes:
                                                                                           int)
                                                        torch.nn.modules.module.Module
     helper function used in cfg to network to assemble a SupervisedSidecarAutoencoder
```

1.14 layer training def.py

Wrappers / Definitions needed in our network.

networks.LayerTrainingDefinition([])	This class is a wrapper for all objects that are needed in
	a layer training process.
LayerType	Enum that describes different layer types.

```
class networks.layer_training_def.LayerTrainingDefinition(layer_type:
                                                                          works.layer_training_def.LayerType
                                                                          = None, layer name:
                                                                         str
                                                                                 =
                                                                                         None,
                                                                          num_epochs:
                                                                                        int =
                                                                          0, pretraining_store:
                                                                          str = None, pretrain-
                                                                          ing load: str = None,
                                                                          model_base_path: str
                                                                          = None, stack: mod-
                                                                          ules.network_stack.NetworkStack
                                                                          = None, upstream:
                                                                          torch.nn.modules.module.Module
                                                                                        model:
                                                                               None,
                                                                          torch.nn.modules.module.Module
                                                                             None,
                                                                                     tp_alpha:
                                                                          torch.nn.parameter.Parameter
                                                                          = None,
                                                                                    optimizer:
                                                                          torch.optim.optimizer.Optimizer
                                                                                         None,
                                                                          ae_loss_function:
                                                                          str = None)
     Bases: object
```

This class is a wrapper for all objects that are needed in a layer training process.

The property stack contains a reference to all the previous layers such that the upstream (representation of x before this layer) can be calculated.

```
ae_loss_function = None
    layer_name = None
    layer_type = None
    model = None
    model_base_path = None
    num_epochs = 0
    optimizer = None
    pretraining_load = None
    pretraining_store = None
    stack = None
    tp_alpha = None
    upstream = None
class networks.layer_training_def.LayerType
    Bases: enum. Enum
```

Enum that describes different layer types.

- VGGlinear : the last layer of our VGG network
- · Stack: any other layer trained by a supervised autoencoder

Stack = 1

VGGlinear = 0

PYTHON MODULE INDEX

```
loaders.cifar10,1
loaders.cifar100,1
loaders.configloader,2
m
modules.autoencoder, 2
modules.fcview, 3
modules.inheritance, 3
modules.interpolate, 4
modules.maps, 4
modules.network_stack,4
modules.stackable_network,5
modules.vgg, 6
modules.vgg_autoencoder,5
networks.cfg_to_network,8
networks.encodernet, 6
networks.layer_training_def, 8
```

12 Python Module Index

INDEX

Α	E
abstract_method() (mod- ules.inheritance.StackableNetwork method), 3	<pre>ensure_dir() (in module networks.encodernet), 8</pre>
abstract_method() (mod- ules.inheritance.SupervisedAutoencoder method), 3	<pre>FCView (class in modules.fcview), 3 from_file() (loaders.configloader.ConfigLoader</pre>
ae_loss_function (net- works.layer_training_def.LayerTrainingDefinition attribute), 9	from_string() (loaders.configloader.ConfigLoader method), 2
Autoencoder (class in modules.autoencoder), 3	G
Autoencoder (class in modules.inheritance), 3 AutoencoderNet (class in networks.encodernet), 7	<pre>get_trainable_modules() (modules.vgg.VGG</pre>
В	1
bottleneck_size() (mod- ules.vgg_autoencoder.SidecarAutoencoder	Interpolate (class in modules.interpolate), 4
method), 6	L
C	layer_name (networks.layer_training_def.LayerTrainingDefinition attribute), 9
calculate_upstream() (mod- ules.autoencoder.Autoencoder method),	layer_type (networks.layer_training_def.LayerTrainingDefinition attribute), 9
calculate_upstream() (mod- ules.stackable_network.StackableNetwork	LayerTrainingDefinition (class in net- works.layer_training_def), 8 LayerType (class in networks.layer_training_def), 9
<pre>method), 5 calculate_upstream() (mod-</pre>	<pre>load_layer() (in module networks.cfg_to_network),</pre>
ules.vgg_autoencoder.SidecarAutoencoder	8 loaders.cifar10 (module), 1
method), 6	loaders.cifar100 (module), 1
cfg_to_network() (in module net- works.cfg_to_network), 8	loaders.configloader (module), 2
CifarSubsetSampler (class in loaders.cifar10), 1	M
CifarSubsetSampler (class in loaders.cifar100), 2 ConfigLoader (class in loaders.configloader), 2	majority_vote() (net-
ConvMap (class in modules.maps), 4	works.encodernet.AutoencoderNet method), 7
D	make_modules() (modules.vgg.VGG method), 6
${\tt decoder} \ \textit{(modules.inheritance.} \\ \textit{Autoencoder attribute)}, \\ 3$	measure_time() (net- works.encodernet.AutoencoderNet method),
DecoderMap (class in modules.maps), 4	model (networks.layer_training_def.LayerTrainingDefinition attribute), 9

<pre>model_base_path</pre>	SupervisedAutoencoder (class in mod- n ules.autoencoder), 3
<pre>attribute), 9 modules.autoencoder (module), 2</pre>	SupervisedAutoencoder (class in mod- ules.inheritance), 3
modules.fcview (module), 3	SupervisedSidecarAutoencoder (class in mod-
modules.inheritance (module), 3	ules.vgg_autoencoder), 6
modules.interpolate (module), 4	supervision (modules.inheritance.SupervisedAutoencoder
modules.maps (module), 4	attribute), 3
modules.network_stack (module), 4	switch() (loaders.configloader.ConfigLoader
modules.stackable_network (module), 5	method), 2
modules.vgg (module), 6	_
modules.vgg_autoencoder (module), 5	T
N	test (modules.inheritance.SupervisedAutoencoder at-
networks.cfg_to_network(module),8	tribute), 3
networks.encodernet (module), 6	test() (networks.encodernet.AutoencoderNet method),
networks.layer_training_def(module), 8	test_vgg_classifier() (net-
NetworkStack (class in modules.network_stack), 5	works.encodernet.AutoencoderNet method),
num_epochs (networks.layer_training_def.LayerTraining	
attribute), 9	to_img() (networks.encodernet.AutoencoderNet
	method), 7
0	to_json() (loaders.configloader.ConfigLoader
optimizer(networks.layer_training_def.LayerTrainingI	-
attribute), 9	$\verb tp_alpha (networks. layer_training_def. LayerTrainingDefinition) $
P	attribute), 9
	train() (networks.encodernet.AutoencoderNet method), 7
plot_img() (networks.encodernet.AutoencoderNet	train_test() (networks.encodernet.AutoencoderNet
method), 7	method), 7
<pre>pretraining_load</pre>	
attribute), 9	works.encodernet.AutoencoderNet method),
pretraining_store (net-	7
works.layer_training_def.LayerTrainingDefinitio	n
attribute), 9	Ü
R	upstream(networks.layer_training_def.LayerTrainingDefinition attribute), 9
RandomMap (class in modules.maps), 4	upwards() (modules.network_stack.NetworkStack
S	method), 5
	V
save_layer() (in module networks.encodernet), 8	
semi_supervised_cifar10() (in module load-	VGG (class in modules.vgg), 6
ers.cifar10), 1	vgg_sidecar_layer() (in module net-
semi_supervised_cifar100() (in module load-	works.cfg_to_network), 8
ers.cifar100), 2 SidecarAutoencoder (class in mod-	VGGlinear (networks.layer_training_def.LayerType
ules.vgg_autoencoder), 5	<pre>attribute), 9 visualize() (networks.encodernet.AutoencoderNet</pre>
SidecarMap (class in modules.maps), 4	method), 7
stack (networks.layer_training_def.LayerTrainingDefinit	· · · · · · · · · · · · · · · · · · ·
attribute), 9	W
Stack (networks.layer_training_def.LayerType at-	<pre>wave_train_test()</pre>
tribute), 9	works.encodernet.AutoencoderNet method),
StackableNetwork (class in modules.inheritance), 3	8
StackableNetwork (class in mod-	· ·
ules.stackable_network), 5	

14 Index