deephyp Documentation

Release 0.1.5

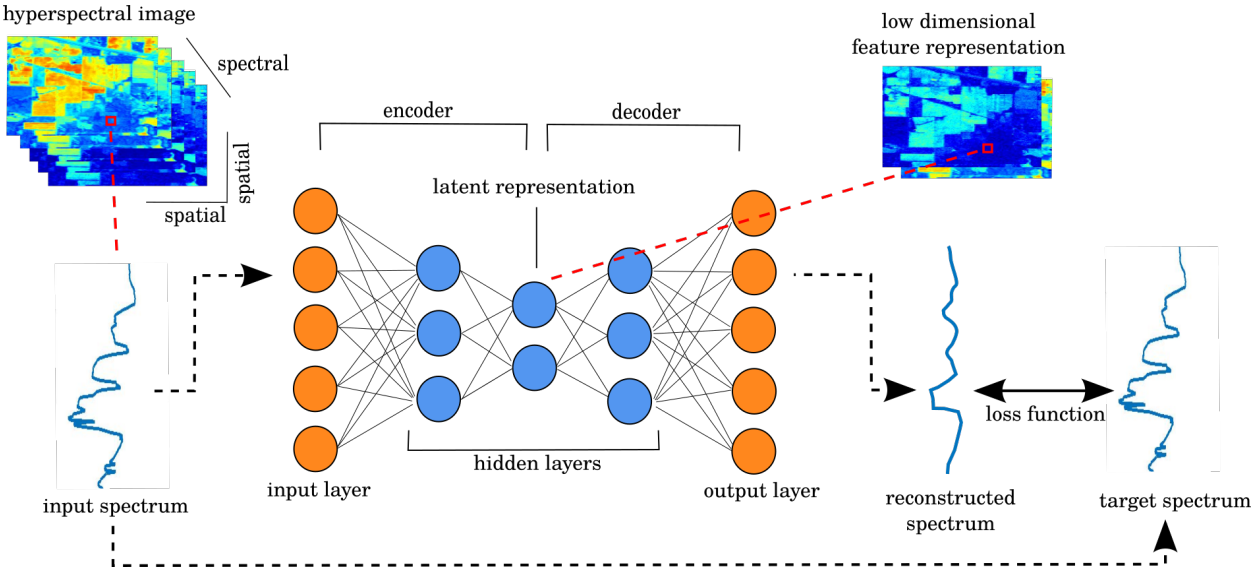
Lloyd Windrim

Oct 14, 2019

Contents:

deephyp is an open source python-based toolbox, built on tensorflow, for training and using unsupervised autoencoders and supervised deep learning classifiers for hyperspectral data.

Source code available on Github.



CHAPTER 1

Installation

The latest release of the toolbox can be installed from the command line using pip:

```
pip install deephyp
```

or to update:

```
pip install deephyp --upgrade
```

The software dependencies needed to run the toolbox are python 2 or python 3 (tested with version 2.7.15 and 3.5.2) with packages:

- tensorflow (tested with v1.14.0) - not yet compatible with tensorflow v2.0
- numpy (tested with v1.15.4)

Because deephyp is note yet compatible with tensorflow v2.0, you will have to install an older version of tensorflow:

```
pip install tensorflow==1.14
```

Or if you are using a gpu:

```
pip install tensorflow-gpu==1.14
```

If you want to use deephyp but you have tensorflow v2.0 installed, you can install deephyp in a virtual environment with tensorflow v1.14. See instructions on setting up a virtual environment here.

To import deephyp, in python write:

```
import deephyp
```

Source code available on github.

CHAPTER 2

How to cite

If you use the toolbox in your research, please cite: Windrim et al. Unsupervised Feature-Learning for Hyperspectral Data with Autoencoders. Remote Sensing 11.7 (2019): 864. This paper explains the spectral angle (SA), spectral information divergence (SID) and sum-of-squared errors (SSE) loss functions for training autoencoders.

If you use the cosine spectral angle (CSA) loss function in your research, please cite: Windrim et al. Unsupervised feature learning for illumination robustness. 2016 IEEE International Conference on Image Processing (ICIP).

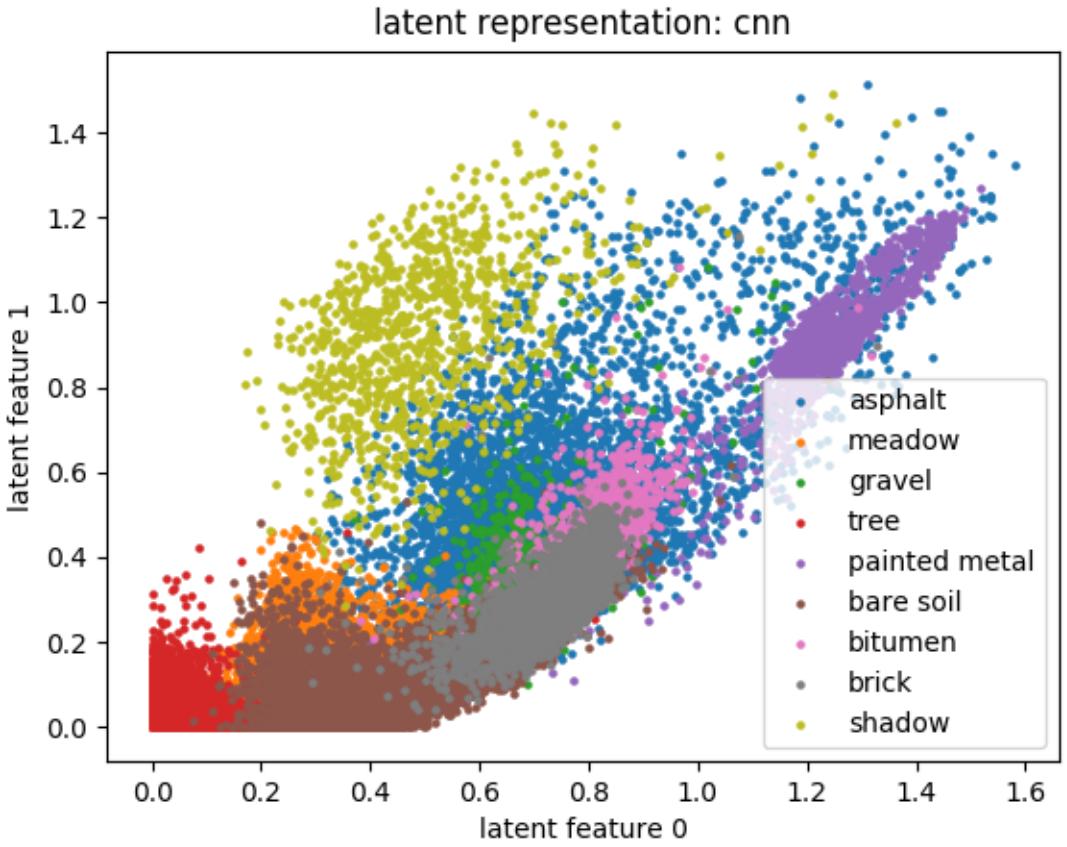
If you use the classification networks in your research, please cite: Windrim et al. Hyperspectral CNN Classification with Limited Training Samples. 2017 Proceedings of the British Machine Vision Conference (BMVC).

CHAPTER 3

Getting Started

3.1 Getting started with autoencoders

Autoencoders are unsupervised neural networks that are useful for a range of applications such as unsupervised feature learning and dimensionality reduction. Autoencoders are trained to learn the parameters for an *encoder* which maps the input data to a latent space and a *decoder* which reconstructs the input from the latent space. The latent space is often of a lower dimensionality then the input data, and can be thought of as a feature vector. The network is trained to minimise the reconstruction error between its decoded output and the input data (hence it is *unsupervised*). Once trained, the *encoder* can be used to map data to the latent space.



3.1.1 Download a hyperspectral dataset

Some hyperspectral datasets in a matlab file format (.mat) can be downloaded from here. To get started, download the 'Pavia University' dataset.

deephyp operates on hyperspectral data in numpy array format. The matlab file (.mat) you just downloaded can be read as a numpy array using the scipy.io.loadmat function:

```
import scipy.io
mat = scipy.io.loadmat( 'PaviaU.mat' )
img = mat[ 'paviaU' ]
```

where *img* is a numpy array. You are now ready to use the toolbox!

3.1.2 Overview

For both autoencoders and classifiers, the toolbox uses several key processes:

- data preparation
- data iterator
- building networks
- adding train operations

- training networks
- loading and testing a trained network

Each of these are elaborated on below:

3.1.3 Data preparation

A class within the toolbox from the *data* module called *HypImg* handles the hyperspectral dataset and all of its metadata. As mentioned earlier, the class accepts the hyperspectral data in numpy format, with shape [numRows x numCols x numBands] or [numSamples x numBands]. The networks in the toolbox operate in the spectral domain, not the spatial, so if a hypercube image is input with shape [numRows x numCols x numBands], it is reshaped to [numSamples x numBands], collapsing the spatial dimensions into a single dimension.

The Pavia Uni hyperspectral image can be passed to the *HypImg* class as follows:

```
from deephyp import data
hypData = data.HypImg( img )
```

It is also possible to pass class labels to *HypImg*, but if you are training an unsupervised autoencoder you do not need to do this.

Then the data can be pre-processed using a function of the *HypImg* class. For example, using the 'minmax' approach:

```
hypData.pre_process( 'minmax' )
```

The result is stored in the attribute *spectraPrep* attribute. Currently, only the 'minmax' approach is available, but additions will be made in future versions.

3.1.4 Data iterator

The *Iterator* class within the *data* module has methods for calling batches from the data that are used to train the network. A separate iterator object is made for the training and validation data. For example, an iterator object made from 200,000 pre-processed hyperspectral training samples with a batchsize of 1000 is defined by:

```
dataTrain = data.Iterator( dataSamples=hypData.spectraPrep[:200000, :],␣
↪targets=hypData.spectraPrep[:200000, :], batchSize=1000 )
```

Similarly, an iterator object made from 100 validation samples is defined as:

```
dataVal = data.Iterator( dataSamples=hypData.spectraPrep[200000:200100, :],␣
↪targets=hypData.spectraPrep[200000:200100, :] )
```

Because the batchsize is unspecified for the validation iterator, all 100 samples are used for each batch. For a typical unsupervised autoencoder, the targets that the network is learning to output are the same as the data samples being input into the network, as in the above iterator examples. When training a supervised classifier, the targets will be the ground truth class labels.

The data in any iterator can also be shuffled before it is used to train a network:

```
dataTrain.shuffle()
```

3.1.5 Building networks

The *autoencoder* module has classes for creating autoencoder neural networks:

```
from deephyp import autoencoder
```

There are currently two type of autoencoders that can be set up. A multi-layer perceptron (MLP) autoencoder has purely fully-connected (i.e. dense) layers:

```
net = autoencoder.mlp_1D_network( inputSize=hypData.numBands )
```

And a convolutional autoencoder has mostly convolutional layers, with a fully-connected layer used to map the final convolutional layer in the encoder to the latent vector:

```
net = autoencoder.cnn_1D_network( inputSize=hypData.numBands )
```

If not using config files to set up a network, then the input size of the data must be specified. This should be the number of spectral bands, which is stored in *hypData.numBands* for convenience.

Additional aspects of the network architecture can also be specified when initialising the *autoencoder* object. For the MLP autoencoder:

```
net = autoencoder.mlp_1D_network( inputSize=hypData.numBands, encoderSize=[50,30,10,
↪5], activationFunc='relu', weightInitOpt='truncated_normal', tiedWeights=[1,0,0,0],␣
↪skipConnect=False, activationFuncFinal='linear')
```

where the following components of the architecture can be specified:

- number of layers in the encoder (and decoder) - this is the length of the list 'encoderSize'
- number of neurons in each layer of the encoder - these are the values in the 'encoderSize' list. The last value in the list is the number of dimensions in the latent vector.
- the activation function which proceeds each layer and the function for the final decoder layer - activationFunc and activationFuncFinal
- the method of initialising network parameters (e.g. xavier improved) - 'weightInitOpt'
- which layers of the encoder to tie to the decoder, such that they share a set of parameters - these are the values in the list 'tiedWeights'
- whether the network uses skip connections between corresponding layers in the encoder and decoder - specified by the boolean argument skipConnect

Therefore, the above MLP autoencoder has four encoder layers (and four symmetric decoder layers), with five neurons in the latent layer. This network could be used to represent a hyperspectral image with five dimensions.

The convolutional autoencoder has similar arguments for defining the network architecture, but without 'encoderSize' and with some additional arguments:

```
net = autoencoder.cnn_1D_network( inputSize=hypData.numBands, zDim=3,␣
↪encoderNumFilters=[10,10,10], encoderFilterSize=[20,10,10],  activationFunc='relu',␣
↪weightInitOpt='truncated_normal',  encoderStride=[1, 1, 1], padding='VALID',␣
↪tiedWeights=[0,0,0],  skipConnect=False, activationFuncFinal='linear' )
```

which are:

- number of layers in the encoder (and decoder) - this is the length of the list 'encodernumFilters'
- number of filters/kernels in each conv layer - these are the values in the 'encodernumFilters' list
- the size of the filters/kernels in each conv layer - these are the values in the 'encoderFilterSize' list
- the stride of the filters/kernels in each conv layer - these are the values in the 'encoderStride' list
- the number of dimensions in the latent vector - zDim

- the type of padding each conv layer uses - padding

Note that the convolutional autoencoder uses *deconvolutional* layers in the decoder, which can upsample the data from the latent layer to the output layer.

Instead of defining the network architecture by the initialisation arguments, a config.json file can be used:

```
net = autoencoder.mlp_1D_network( configFile='config.json') )
```

A config file is generated each time a network in the toolbox is trained, so you can use one from another network as a template for making a new one.

3.1.6 Adding training operations

Once a network has been created, a training operation can be added to it. It is possible to add multiple training operations to a network, so each op must be given a name:

```
net.add_train_op( name='experiment_1' )
```

When adding a train op, details about how the network will be trained with that op can also be specified. For example, a train op for an autoencoder which uses the cosine spectral angle (CSA) loss function, a learning rate of 0.001 with no decay, optimised with Adam and no weight decay can be defined by:

```
net.add_train_op( name='experiment_1', lossFunc='CSA', learning_rate=1e-3, method=
↪'Adam', wd_lambda=0.0 )
```

There are several loss functions that can be used to train an autoencoder with this toolbox, many of which were designed specifically for hyperspectral data:

- cosine spectral angle (CSA)
- spectral angle (SA)
- spectral information divergence (SID)
- sum-of-squared errors (SSE)

Note that when using the CSA, SA and SID loss functions it is expected that the reconstructed spectra have a different magnitude to the target spectra, but a similar shape. The SSE should produce a similar magnitude and shape. Also, since the SID contains *log* in its expression which is undefined for values <= *0*, it is best to use sigmoid as the activation function (including the final activation function) for networks trained with the SID loss. See the code examples for a demonstration.

The method for decaying the learning rate can also be customised. For example, to decay the learning rate exponentially every 100 steps (starting at 0.001):

```
net.add_train_op( name='experiment_1',learning_rate=1e-3, decay_steps=100, decay_
↪rate=0.9 )
```

A piecewise approach to decaying the learning rate can also be used. For example, to change the learning rate from 0.001 to 0.0001 after 100 steps, and then to 0.00001 after a further 200 steps:

```
net.add_train_op( name='experiment_1',learning_rate=1e-3, piecewise_bounds=[100,300],␣
↪piecewise_values=[1e-4,1e-5] )
```

3.1.7 Training networks

Once one or multiple training ops have been added to a network, they can be used to learn a model (or multiple models) for that network through training:

```
net.train( dataTrain=dataTrain, dataVal=dataVal, train_op_name='experiment_1', n_
↪epochs=100, save_addr=model_directory, visualiseRateTrain=5, visualiseRateVal=10,␣
↪save_epochs=[50,100])
```

The train method learns a model using one train op, therefore the train method should be called at least once for each train op that was added. The name of the train op must be specified, and the training and validation iterators created previously must be input. A path to a directory to save the model must also be specified. The example above will train a network for 100 epochs of the training dataset (that is, loop through the entire training dataset 100 times), and save the model at 50 and 100 epochs. The training loss will be displayed every 5 epochs, and the validation loss will be displayed every 10 epochs.

It is also possible to load a pre-trained model and continue to train it by passing the address of the epoch folder containing the model checkpoint as the save_addr argument. For example, if the directory for the model at epoch 50 (epoch_50 folder) was passed to save_addr in the example above, then the model would initialise with the epoch 50 parameters and be trained for an additional 50 epochs to reach 100, at which point the model would be saved in a folder called epoch_100 in the same directory as the epoch_50 folder.

The interface for training autoencoders and classifiers is the same.

3.1.8 Loading and testing a trained network

Once you have a trained network, it can be loaded and tested out on some hyperspectral data.

Open a new python script. To load a trained model on a new dataset, ensure the data has been pre-processed similarly using:

```
from deephyp import data
new_hypData = data.HypImg( new_img )
new_hypData.pre_process( 'minmax' )
```

Then set up the network. The network architecture must be the same as the one used to train the model being loaded. However, this is easy as the directory where models are saved should contain an automatically generated config.json file, which can be used to set up the network with the same architecture:

```
from deephyp import autoencoder
net = autoencoder.mlp_1D_network( configFile='model_directory/config.json' )
```

Once the architecture has been defined, add a model to the network. For example, adding the model that was saved at epoch 100:

```
net.add_model( addr='model_directory/epoch_100'), modelName='csa_100' )
```

Because multiple models can be added to a single network, the added model must be given a name. The name can be anything - the above model is named 'csa_100' because it was trained for 100 epochs using the cosine spectral angle loss function).

When the network is set up and a model has been added, hyperspectral data can be passed through it. To use a trained autoencoder to extract the latent vectors of some spectra:

```
dataZ = net.encoder( modelName='csa_100', dataSamples=new_hypData.spectraPrep )
```

Make sure to refer to the name of the model the network should use. The encoded hyperspectral data (*dataZ*) can also be decoded to get the reconstruction:

```
dataY = net.decoder(modelName='csa_100', dataZ=dataZ)
```

It is also possible to encode and decode in one step with:

```
dataY = net.encoder_decoder(modelName='csa_100', dataZ=new_hypData.spectraPrep)
```

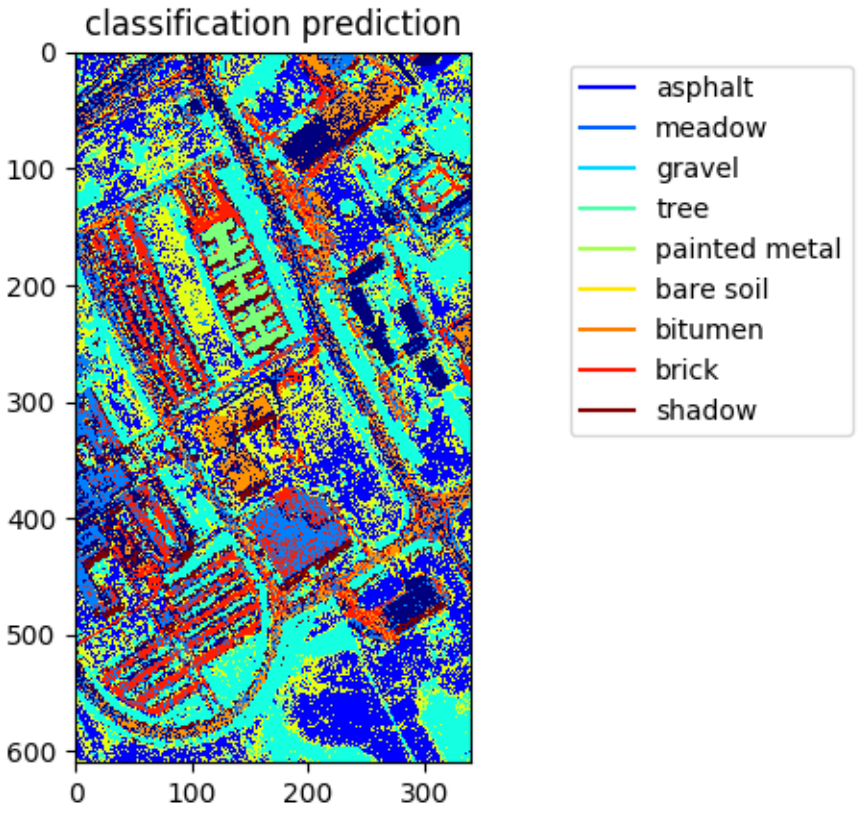
You can use numpy to reshape the latent vector *dataZ* so that it looks like an image again:

```
import numpy
imgZ = numpy.reshape( dataZ, (new_hypData.numRows, new_hypData.numCols, -1) )
```

Now you should have a basic idea of how to use the **deephyp** toolbox to train an autoencoder for hyperspectral data!

3.2 Getting started with classifiers

Classifiers are supervised neural networks that can be trained to automatically predict the class label of a data sample based on it spectral characteristics. Classifiers are trained to map input data samples to one-hot binary vectors which indicate the class of the data sample. The network requires data samples with class labels to train on (hence it is *supervised*). Once trained, a classifier can predict the class of new data samples.



3.2.1 Download a hyperspectral dataset

Some hyperspectral datasets in a matlab file format (.mat) can be downloaded from here. To get started, download the 'Pavia University' dataset and its ground truth labels.

deephyp operates on hyperspectral data in numpy array format. The matlab files (.mat) you just downloaded can be read as a numpy array using the scipy.io.loadmat function:

```
import scipy.io
mat = scipy.io.loadmat( 'PaviaU.mat' )
img = mat[ 'paviaU' ]
```

where *img* is a numpy array. Use the same function to read the ground truth class labels:

```
mat_gt = scipy.io.loadmat('PaviaU_gt.mat')
img_gt = mat_gt['paviaU_gt']
```

You are now ready to use the toolbox!

3.2.2 Overview

For both autoencoders and classifiers, the toolbox uses several key processes:

- data preparation
- data iterator
- building networks
- adding train operations
- training networks
- loading and testing a trained network

Each of these are elaborated on below:

3.2.3 Data preparation

A class within the toolbox from the *data* module called *HypImg* handles the hyperspectral dataset and all of its meta-data. As mentioned earlier, the class accepts the hyperspectral data in numpy format, with shape [numRows x numCols x numBands] or [numSamples x numBands]. The networks in the toolbox operate in the spectral domain, not the spatial, so if a hypercube image is input with shape [numRows x numCols x numBands], it is reshaped to [numSamples x numBands], collapsing the spatial dimensions into a single dimension.

The Pavia Uni hyperspectral image and labels can be passed to the *HypImg* class as follows:

```
from deephyp import data
hypData = data.HypImg( img, labels=img_gt )
```

Upon initialisation the *HypImg* object will automatically generate one-hot labels from the labels input, stored in the *labelsOnehot* attribute. Classes with a label <= 0 are considered a background class, and are not included in the *numClasses* attribute. Any samples with a zero label will appear as a row of zeros in *labelsOnehot*.

The data can be pre-processed using a function of the *HypImg* class. For example, using the 'minmax' approach:

```
hypData.pre_process( 'minmax' )
```

The result is stored in the *spectraPrep* attribute. Currently, only the 'minmax' approach is available, but additions will be made in future versions.

3.2.4 Data iterator

The *Iterator* class within the *data* module has methods for calling batches from the data that are used to train the network. A separate iterator object is made for the training and validation data.

Before setting up an *Iterator* object, establish which data samples from the hyperspectral image you will use to train and validate the network. For example, you can use the following code to get the indexes of 50 training samples and 15 validation samples per class, for each of the nine non-background classes in the Pavia Uni dataset:

```
import numpy as np
trainSamples = 50 # per class
valSamples = 15 # per class
train_indexes = []
for i in range(1,10):
    train_indexes += np.nonzero(hypData.labels == i)[0][:trainSamples].tolist()
val_indexes = []
for i in range(1,10):
    val_indexes += np.nonzero(hypData.labels ==␣
↪i)[0][trainSamples:trainSamples+valSamples].tolist()
```

Now, to build an iterator object for training from the pre-processed hyperspectral training samples and their labels, with a batchsize of 50, use:

```
dataTrain = data.Iterator( dataSamples=hypData.spectraPrep[train_indexes, :],␣
↪targets=hypData.labelsOnehot[train_indexes,:], batchSize=50 )
```

Since we are training a supervised classifier, the targets are the ground truth class labels.

Similarly, an iterator object for validation is defined with:

```
dataVal = data.Iterator( dataSamples=hypData.spectraPrep[val_indexes, :],␣
↪targets=hypData.labelsOnehot[val_indexes,:] )
```

Because the batchsize is unspecified for the validation iterator, all samples are used for each batch.

The data in any iterator can also be shuffled before it is used to train a network:

```
dataTrain.shuffle()
```

3.2.5 Building networks

The *classifier* module has a class for creating supervised classification neural networks:

```
from deephyp import classifier
```

There is currently one type of classifier that can be set up, which contains a combination of convolutional layers (at the start) and fully-connected layers (at the end).:

```
net = classifier.cnn_1D_network( inputSize=hypData.numBands, numClasses=hypData.
↪numClasses )
```

If not using config files to set up a network, then the input size of the data (which should be the number of spectral bands) and the number of classes must be specified. These are stored in *hypData.numBands* and *hypData.numClasses* for convenience.

Additional aspects of the network architecture can also be specified when initialising the *classifier* object:

```
net = classifier.cnn_1D_network( inputSize=hypData.numBands, numClasses=hypData.
↪numClasses, convFilterSize=[20,10,10], convNumFilters=[10,10,10], convStride = [1,1,
↪1], fcSize=[20,20], activationFunc='relu', weightInitOpt='truncated_normal',␣
↪padding='VALID' )
```

where the following components of the architecture can be specified:

- number of convolutional layers - this is the length of the list 'convNumFilters'
- number of filters/kernels in each conv layer - these are the values in the 'convNumFilters' list
- the size of the filters/kernels in each conv layer - these are the values in the 'convFilterSize' list
- the stride of the filters/kernels in each conv layer - these are the values in the 'convStride' list
- the type of padding each conv layer uses - padding
- number of fully-connected layers - this is the length of the list 'fcSize'
- number of neurons in each fully-connected layer - these are the values in the 'fcSize' list
- the activation function which proceeds each layer - activationFunc
- the method of initialising network parameters (e.g. xavier improved) - 'weightInitOpt'

Therefore, the above CNN classifier has three convolutional layers, two fully-connected layers and an output layer. The three convolutional layers each have 10 filters, with sizes 20, 10 and 10. The fully-connected layers both have 20 neurons.

Instead of defining the network architecture by the initialisation arguments, a config.json file can be used:

```
net = classifier.cnn_1D_network( configFile='config.json') )
```

A config file is generated each time a network in the toolbox is trained, so you can use one from another network as a template for making a new one.

3.2.6 Adding training operations

Once a network has been created, a training operation can be added to it. It is possible to add multiple training operations to a network, so each op must be given a name:

```
net.add_train_op( name='experiment_1' )
```

When adding a train op, details about how the network will be trained with that op can also be specified. For example, a train op for a classifier with a learning rate of 0.001 with no decay, optimised with Adam, class balancing and no weight decay can be defined by:

```
net.add_train_op( name='experiment_1', balance_classes=True, learning_rate=1e-3,␣
↪method='Adam', wd_lambda=0.0  )
```

Classification networks are trained using a cross-entropy loss function.

The method for decaying the learning rate can also be customised. For example, to decay the learning rate exponentially every 100 steps (starting at 0.001):

```
net.add_train_op( name='experiment_1',learning_rate=1e-3, decay_steps=100, decay_
↪rate=0.9 )
```

A piecewise approach to decaying the learning rate can also be used. For example, to change the learning rate from 0.001 to 0.0001 after 100 steps, and then to 0.00001 after a further 200 steps:

```
net.add_train_op( name='experiment_1',learning_rate=1e-3, piecewise_bounds=[100,300],␣
↪piecewise_values=[1e-4,1e-5] )
```

3.2.7 Training networks

Once one or multiple training ops have been added to a network, they can be used to learn a model (or multiple models) for that network through training:

```
net.train( dataTrain=dataTrain, dataVal=dataVal, train_op_name='experiment_1', n_
↪epochs=100, save_addr=model_directory, visualiseRateTrain=5, visualiseRateVal=10,␣
↪save_epochs=[50,100])
```

The train method learns a model using one train op, therefore the train method should be called at least once for each train op that was added. The name of the train op must be specified, and the training and validation iterators created previously must be input. A path to a directory to save the model must also be specified. The example above will train a network for 100 epochs of the training dataset (that is, loop through the entire training dataset 100 times), and save the model at 50 and 100 epochs. The training loss will be displayed every 5 epochs, and the validation loss will be displayed every 10 epochs.

It is also possible to load a pre-trained model and continue to train it by passing the address of the epoch folder containing the model checkpoint as the save_addr argument. For example, if the directory for the model at epoch 50 (epoch_50 folder) was passed to save_addr in the example above, then the model would initialise with the epoch 50 parameters and be trained for an additional 50 epochs to reach 100, at which point the model would be saved in a folder called epoch_100 in the same directory as the epoch_50 folder.

The interface for training autoencoders and classifiers is the same.

3.2.8 Loading and testing a trained network

Once you have a trained network, it can be loaded and tested out on some hyperspectral data.

Open a new python script. To load a trained model on a new dataset, ensure the data has been pre-processed similarly using:

```
from deephyp import data
new_hypData = data.HypImg( new_img )
new_hypData.pre_process( 'minmax' )
```

When doing inference, labels do not need to be input into *HypImg* (unless you want to use them for evaluation).

Set up the network. The network architecture must be the same as the one used to train the model being loaded. However, this is easy as the directory where models are saved should contain an automatically generated config.json file, which can be used to set up the network with the same architecture:

```
from deephyp import classifier
net = classifier.cnn_1D_network( configFile='model_directory/config.json' )
```

Once the architecture has been defined, add a model to the network. For example, adding the model that was saved at epoch 100:

```
net.add_model( addr='model_directory/epoch_100'), modelName='clf_100' )
```

Because multiple models can be added to a single network, the added model must be given a name. The name can be anything - the above model is named 'clf_100' because it is a classifier and was trained for 100 epochs).

When the network is set up and a model has been added, hyperspectral data can be passed through it. To use a trained classifier to predict the classification labels of some spectra:

```
dataPred = net.predict_labels( modelName='clf_100', dataSamples=new_hypData.
↪spectraPrep  )
```

Like-wise, to predict the classification scores for each class of some spectra:

```
dataScores = net.predict_scores( modelName='clf_100', dataSamples=new_hypData.
↪spectraPrep  )
```

To extract the features in the second last layer of the classifier network:

```
dataFeatures = net.predict_features( modelName='clf_100', dataSamples=new_hypData.
↪spectraPrep, layer=net.numLayers-1 )
```

You can use numpy to reshape the predicted labels (*dataPred*) so that they look like an image again:

```
imgPred = numpy.reshape( dataPred, ( new_hypData.numRows, new_hypData.numCols ) )
```

Now you should have a basic idea of how to use the **deephyp** toolbox to train a classifier for hyperspectral data!

CHAPTER 4

API Reference

4.1 deephyp.data

4.1.1 deephyp.data.HypImg

class deephyp.data.**HypImg**(*spectralInput*, *labels=None*, *wavelengths=None*, *bands=None*)

Class for handling data. Stores meta-data and contains attributes for pre-processing the data. If passed labels, samples with label zero are considered as a background class. This class is not included in numClasses and data samples with this label have a one-hot vector label of all zeros.

Parameters

- **spectralInput** (*np.array float*) – Spectral dataset. Shape can be [numRows x numCols x numBands] or [numSamples x numBands].
- **wavelengths** (*np.array float*) – Vector of wavelengths that spectralInput wavelengths lie within.
- **bands** (*np.array int*) – Wavelength indexes for each band of spectralInput. Shape [numBands].
- **labels** (*np.array int*) – Class labels for each spectral sample in spectralInput. Shape can be [numRows x numCols] or [numSamples].

spectra

Un-pre-processed spectral data with shape [numSamples x numBands].

Type np.array float

spectraCube

If data passed as image - un-pre-processed spectral datacube with shape [numSamples x numBands]. Else None.

Type np.array float

spectraPrep

Pre-processed spectral data with shape [numSamples x numBands].

Type np.array float

numSamples
The number of spectra.

Type int

numRows
If data passed as image - the number of image rows. Else None.

Type int

numCols
If data passed as image - the number of image columns. Else None.

Type int

wavelengths
If provided - vector of wavelengths that spectra wavelengths lie within. Else None.

Type np.array float

bands
If provided - wavelength indexes for each band of spectra with shape [numBands]. Else None.

Type np.array int

labels
If provided - class labels for each spectral sample with shape [numSamples]. Else None.

Type np.array int

labelsOnehot
If labels provided - the one-hot label vector for each sample. Samples with label zero (background class) have a one-hot vector of all zeros. Else None.

Type np.array int

pre_process(*method='minmax'*)
Pre-process data for input into the network. Stores in the spectraPrep attribute.

Parameters **method** (*str*) – Method of pre-processing. Current options: 'minmax'

4.1.2 deephyp.data.Iterator

class deephyp.data.**Iterator**(*dataSamples*, *targets*, *batchSize=None*)
Class for iterating through data, to train the network.

Parameters

- **dataSamples** (*np.array float*) – Data to be input into the network. Shape [numSamples x numBands].
- **targets** (*np.array int*) – Network output target of each dataSample. For classification, these are the class labels, and it could be the dataSamples for autoencoders. Shape [numSamples x arbitrary]
- **batchSize** (*int*) – Number of dataSamples per batch

dataSamples
Data to be input into the network. Shape [numSamples x numBands].

Type np.array float

targets
Network output target of each dataSample. For classification, these are the class labels, and it could be the dataSamples for autoencoders. Shape [numSamples x arbitrary]

Type np.array int

batchSize
Number of dataSamples per batch. If None - set to numSamples (i.e. whole dataset).

Type int

numSamples
The number of data samples.

Type int

currentBatch
A list of indexes specifying the data samples in the current batch. Shape [batchSize]

Type int list

get_batch(*idx*)
Returns a specified set of samples and targets.

Parameters **idx** (*int list*) – Indexes of samples (and targets) to return.

Returns

2-element tuple containing:

- (*np.array float*) - Batch of data samples at [idx] indexes. Shape [length(idx) x numBands].
- (*np.array int*) - Batch of targets at [idx] indexes. Shape [length(idx) x arbitrary].

Return type (tuple)

next_batch()
Return next batch of samples and targets (with batchSize number of samples). The currentBatch indexes are incremented. If end of dataset reached, the indexes wraps around to the beginning.

Returns

2-element tuple containing:

- (*np.array float*) - Batch of data samples at currentBatch indexes. Shape [batchSize x numBands].
- (*np.array int*) - Batch of targets at currentBatch indexes. Shape [batchSize x arbitrary].

Return type (tuple)

reset_batch()
Resets the current batch to the beginning.

shuffle()
Randomly permutes all dataSamples (and corresponding targets).

4.2 deephyp.autoencoder

4.2.1 deephyp.autoencoder.mlp_1D_network

class deephyp.autoencoder.**mlp_1D_network**(*configFile=None, inputSize=None, encoderSize=[50, 30, 10], activationFunc='sigmoid', tiedWeights=None, weightInitOpt='truncated_normal', weightStd=0.1, skipConnect=False, activationFuncFinal='linear'*)

Class for setting up a 1-D multi-layer perceptron (mlp) autoencoder network. Layers are all fully-connected (i.e. dense).

Parameters

- **configFile** (*str*) – Optional way of setting up the network. All other inputs can be ignored (will be overwritten). Pass the address of the .json config file.
- **inputSize** (*int*) – Number of dimensions of input data (i.e. number of spectral bands). Value must be input if not using a config file.
- **encoderSize** (*int list*) – Number of nodes at each layer of the encoder. List length is number of encoder layers.
- **activationFunc** (*str*) – Activation function for all layers except the last one. Current options: ['sigmoid', 'relu', 'linear'].
- **tiedWeights** (*binary list or None*) – Specifies whether or not to tie weights at each layer: - 1: tied weights of specific encoder layer to corresponding decoder weights - 0: do not tie weights of specific layer - None: sets all layers to 0
- **weightInitOpt** (*string*) – Method of weight initialisation. Current options: ['gaussian', 'truncated_normal', 'xavier', 'xavier_improved'].
- **weightStd** (*float*) – Used by 'gaussian' and 'truncated_normal' weight initialisation methods.
- **skipConnect** (*boolean*) – Whether to use skip connections throughout the network.
- **activationFuncFinal** (*str*) – Activation function for final layer. Current options: ['sigmoid', 'relu', 'linear'].

inputSize
Number of dimensions of input data (i.e. number of spectral bands).

Type int

activationFunc
Activation function for all layers except the last one.

Type str

tiedWeights
Whether (1) or not (0) the weights of an encoder layer are tied to a decoder layer.

Type binary list

skipConnect
Whether the network uses skip connections between corresponding encoder and decoder layers.

Type boolean

weightInitOpt
Method of weight initialisation.

Type string

weightStd
Parameter for 'gaussian' and 'truncated_normal' weight initialisation methods.

Type float

activationFuncFinal
Activation function for final layer.

Type str

encoderSize
Number of inputs and number of nodes at each layer of the encoder.

Type int list

decoderSize
Number of nodes at each layer of the decoder and number of outputs.

Type int list

z
Latent representation of data. Accessible through the *encoder* class function, requiring a trained model.

Type tensor

y_recon
Reconstructed output of network. Accessible through the *decoder* and *encoder_decoder* class functions, requiring a trained model.

Type tensor

train_ops
Dictionary of names of train and loss ops (suffixed with _train and _loss) added to the network using the *add_train_op* class function. The name (without suffix) is passed to the *train* class function to train the network with the referenced train and loss op.

Type dict

modelsAddrs
Dictionary of model names added to the network using the *add_model* class function. The names reference models which can be used by the *encoder*, *decoder* and *encoder_decoder* class functions.

Type dict

add_model(*addr*, *modelName*)
Loads a saved set of model parameters for the network.

Parameters

- **addr** (*str*) – Address of the directory containing the checkpoint files.
- **modelName** (*str*) – Name of the model (to refer to it later in-case of multiple models for a given network).

add_train_op(*name*, *lossFunc='CSA'*, *learning_rate=0.001*, *decay_steps=None*, *decay_rate=None*, *piecewise_bounds=None*, *piecewise_values=None*, *method='Adam'*, *wd_lambda=0.0*)
Constructs a loss op and training op from a specific loss function and optimiser. User gives the ops a name, and the train op and loss opp are stored in a dictionary (train_ops) under that name.

Parameters

- **name** (*str*) – Name of the training op (to refer to it later in-case of multiple training ops).

- **lossFunc** (*str*) – Reconstruction loss function.
- **learning_rate** (*float*) – Controls the degree to which the weights are updated during training.
- **decay_steps** (*int*) – Epoch frequency at which to decay the learning rate.
- **decay_rate** (*float*) – Fraction at which to decay the learning rate.
- **piecewise_bounds** (*int list*) – Epoch step intervals for decaying the learning rate. Alternative to decay steps.
- **piecewise_values** (*float list*) – Rate at which to decay the learning rate at the piecewise_bounds.
- **method** (*str*) – Optimisation method.
- **wd_lambda** (*float*) – Scalar to control weighting of weight decay in loss.

decoder (*modelName*, *dataZ*)
Extract the reconstruction of some dataSamples from their latent representation encoding using a trained model.

Parameters

- **modelName** (*str*) – Name of the model to use (previously added with add_model()).
- **dataZ** (*np.array*) – Latent representation of data samples to reconstruct using the network. Shape [numSamples x arbitrary].

Returns Reconstructed data (y_recon attribute). Shape [numSamples x arbitrary].

Return type (np.array)

encoder (*modelName*, *dataSamples*)
Extract the latent variable of some dataSamples using a trained model.

Parameters

- **modelName** (*str*) – Name of the model to use (previously added with add_model()).
- **dataSample** (*np.array*) – Shape [numSamples x inputSize].

Returns Latent representation z of dataSamples. Shape [numSamples x arbitrary].

Return type (np.array)

encoder_decoder (*modelName*, *dataSamples*)
Extract the reconstruction of some dataSamples using a trained model.

Parameters

- **modelName** (*str*) – Name of the model to use (previously added with add_model()).
- **dataSample** (*np.array*) – Data samples to reconstruct using the network. Shape [numSamples x inputSize].

Returns Reconstructed data (y_recon attribute). Shape [numSamples x arbitrary].

Return type (np.array)

train (*dataTrain, dataVal, train_op_name, n_epochs, save_addr, visualiseRateTrain=0, visualiseRateVal=0, save_epochs=[1000]*)
Calls network_ops function to train a network.

Parameters

- **dataTrain** (*obj*) – Iterator object for training data.

- **dataVal** (*obj*) – Iterator object for validation data.
- **train_op_name** (*str*) – Name of training op created.
- **n_epochs** (*int*) – Number of loops through dataset to train for.
- **save_addr** (*str*) – Address of a directory to save checkpoints for desired epochs, or address of saved checkpoint. If address is for an epoch and contains a previously saved checkpoint, then the network will start training from there. Otherwise it will be trained from scratch.
- **visualiseRateTrain** (*int*) – Epoch rate at which to print training loss in console.
- **visualiseRateVal** (*int*) – Epoch rate at which to print validation loss in console.
- **save_epochs** (*int list*) – Epochs to save checkpoints at.

4.2.2 deephyp.autoencoder.cnn_1D_network

class deephyp.autoencoder.**cnn_1D_network**(*configFile=None, inputSize=None, zDim=5, encoderNumFilters=[10, 10, 10], encoderFilterSize=[20, 10, 10], activationFunc='sigmoid', tiedWeights=None, weightInitOpt='truncated_normal', weightStd=0.1, skipConnect=False, padding='VALID', encoderStride=[1, 1, 1], activationFuncFinal='linear'*)

Class for setting up a 1-D convolutional autoencoder network. Builds a network with an encoder containing convolutional layers followed by a single fully-connected layer to map from the final convolutional layer in the encoder to the latent layer. The decoder contains a single fully-connected layer and then several deconvolutional layers which reconstruct the spectra in the output.

Parameters

- **configFile** (*str*) – Optional way of setting up the network. All other inputs can be ignored (will be overwritten). Pass the address of the .json config file.
- **inputSize** (*int*) – Number of dimensions of input data (i.e. number of spectral bands). Value must be input if not using a config file.
- **zDim** (*int*) – Dimensionality of latent vector.
- **encoderNumFilters** (*int list*) – Number of filters at each layer of the encoder. List length is number of convolutional encoder layers. Note that there is a single mlp layer after the last convolutional layer.
- **encoderFilterSize** (*int list*) – Size of filter at each layer of the encoder. List length is number of encoder layers.
- **activationFunc** (*str*) – Activation function for all layers except the last one. Current options: [‘sigmoid’, ‘relu’, ‘linear’].
- **tiedWeights** (*binary list or None*) – Specifies whether or not to tie weights at each layer: - 1: tied weights of specific encoder layer to corresponding decoder weights - 0: do not tie weights of specific layer - None: sets all layers to 0
- **weightInitOpt** (*string*) – Method of weight initialisation. Current options: [‘gaussian’, ‘truncated_normal’, ‘xavier’, ‘xavier_improved’].
- **weightStd** (*float*) – Used by ‘gaussian’ and ‘truncated_normal’ weight initialisation methods.
- **skipConnect** (*boolean*) – Whether to use skip connections throughout the network.

- **padding** (*str*) – Type of padding used. Current options: ['VALID', 'SAME'].
- **encoderStride** (*int list*) – Stride at each convolutional encoder layer.
- **activationFuncFinal** (*str*) – Activation function for final layer. Current options: ['sigmoid', 'relu', 'linear'].

inputSize
Number of dimensions of input data (i.e. number of spectral bands).

Type int

activationFunc
Activation function for all layers except the last one.

Type str

tiedWeights
Whether (1) or not (0) the weights of an encoder layer are tied to a decoder layer.

Type binary list

skipConnect
Whether the network uses skip connections between corresponding encoder and decoder layers.

Type boolean

weightInitOpt
Method of weight initialisation.

Type string

weightStd
Parameter for 'gaussian' and 'truncated_normal' weight initialisation methods.

Type float

activationFuncFinal
Activation function for final layer.

Type str

encoderNumFilters
Number of filters at each layer of the encoder. List length is number of convolutional encoder layers. Note that there is a single mlp layer after the last convolutional layer.

Type int list

encoderFilterSize
Size of filter at each layer of the encoder. List length is number of encoder layers.

Type int list

encoderStride
Stride at each convolutional encoder layer.

Type int list

decoderNumFilters

Type int list

decoderFilterSize

Type int list

decoderStride

Type int list

zDim
Dimensionality of latent vector.

Type int

padding
Type of padding used. Current options: ['VALID', 'SAME'].

Type str

z
Latent representation of data. Accessible through the *encoder* class function, requiring a trained model.

Type tensor

y_recon
Reconstructed output of network. Accessible through the *decoder* and *encoder_decoder* class functions, requiring a trained model.

Type tensor

train_ops
Dictionary of names of train and loss ops (suffixed with _train and _loss) added to the network using the *add_train_op* class function. The name (without suffix) is passed to the *train* class function to train the network with the referenced train and loss op.

Type dict

modelsAddrs
Dictionary of model names added to the network using the *add_model* class function. The names reference models which can be used by the *encoder*, *decoder* and *encoder_decoder* class functions.

Type dict

add_model(*addr*, *modelName*)
Loads a saved set of model parameters for the network.

Parameters

- **addr** (*str*) – Address of the directory containing the checkpoint files.
- **modelName** (*str*) – Name of the model (to refer to it later in-case of multiple models for a given network).

add_train_op(*name*, *lossFunc='SSE'*, *learning_rate=0.001*, *decay_steps=None*, *decay_rate=None*, *piecewise_bounds=None*, *piecewise_values=None*, *method='Adam'*, *wd_lambda=0.0*)
Constructs a loss op and training op from a specific loss function and optimiser. User gives the ops a name, and the train op and loss opp are stored in a dictionary (train_ops) under that name.

Parameters

- **name** (*str*) – Name of the training op (to refer to it later in-case of multiple training ops).
- **lossFunc** (*str*) – Reconstruction loss function.
- **learning_rate** (*float*) – Controls the degree to which the weights are updated during training.
- **decay_steps** (*int*) – Epoch frequency at which to decay the learning rate.
- **decay_rate** (*float*) – Fraction at which to decay the learning rate.

- **piecewise_bounds** (*int list*) – Epoch step intervals for decaying the learning rate. Alternative to decay steps.
- **piecewise_values** (*float list*) – Rate at which to decay the learning rate at the piecewise_bounds.
- **method** (*str*) – Optimisation method.
- **wd_lambda** (*float*) – Scalar to control weighting of weight decay in loss.

decoder (*modelName*, *dataZ*)
Extract the reconstruction of some dataSamples from their latent representation encoding using a trained model.

Parameters

- **modelName** (*str*) – Name of the model to use (previously added with add_model()).
- **dataZ** (*np.array*) – Latent representation of data samples to reconstruct using the network. Shape [numSamples x arbitrary].

Returns Reconstructed data (y_recon attribute). Shape [numSamples x arbitrary].

Return type (np.array)

encoder (*modelName*, *dataSamples*)
Extract the latent variable of some dataSamples using a trained model.

Parameters

- **modelName** (*str*) – Name of the model to use (previously added with add_model()).
- **dataSample** (*np.array*) – Shape [numSamples x inputSize].

Returns Latent representation z of dataSamples. Shape [numSamples x arbitrary].

Return type (np.array)

encoder_decoder (*modelName*, *dataSamples*)
Extract the reconstruction of some dataSamples using a trained model.

Parameters

- **modelName** (*str*) – Name of the model to use (previously added with add_model()).
- **dataSample** (*np.array*) – Data samples to reconstruct using the network. Shape [numSamples x inputSize].

Returns Reconstructed data (y_recon attribute). Shape [numSamples x arbitrary].

Return type (np.array)

train (*dataTrain, dataVal, train_op_name, n_epochs, save_addr, visualiseRateTrain=0, visualiseRateVal=0, save_epochs=[1000]*)
Calls network_ops function to train a network.

Parameters

- **dataTrain** (*obj*) – Iterator object for training data.
- **dataVal** (*obj*) – Iterator object for validation data.
- **train_op_name** (*str*) – Name of training op created.
- **n_epochs** (*int*) – Number of loops through dataset to train for.

- **save_addr** (*str*) – Address of a directory to save checkpoints for desired epochs, or address of saved checkpoint. If address is for an epoch and contains a previously saved checkpoint, then the network will start training from there. Otherwise it will be trained from scratch.
- **visualiseRateTrain** (*int*) – Epoch rate at which to print training loss in console.
- **visualiseRateVal** (*int*) – Epoch rate at which to print validation loss in console.
- **save_epochs** (*int list*) – Epochs to save checkpoints at.

4.3 deephyp.classifier

4.3.1 deephyp.classifier.cnn_1D_network

class deephyp.classifier.**cnn_1D_network**(*configFile=None, inputSize=None, numClasses=None, convFilterSize=[20, 10, 10], convNumFilters=[10, 10, 10], convStride=[1, 1, 1], fcSize=[20, 20], activationFunc='relu', weightInitOpt='truncated_normal', weightStd=0.1, padding='VALID'*)

Class for setting up a 1-D convolutional neural network (cnn) for classification. Contains several convolutional layers followed by several fully-connected layers. The network outputs scores for each class, for a given set of input data samples.

Parameters

- **configFile** (*str*) – Optional way of setting up the network. All other inputs can be ignored (will be overwritten). Pass the address of the .json config file.
- **inputSize** (*int*) – Number of dimensions of input data (i.e. number of spectral bands). Value must be input if not using a config file.
- **numClasses** (*int*) – Number of labelled classes in the dataset (not including the zero class).
- **convFilterSize** (*int list*) – Size of filter at each convolutional layer. List length is number of convolutional layers.
- **convNumFilters** (*int list*) – Number of filters at each convolutional layer of the network. List length is number of convolutional layers.
- **convStride** (*int list*) – Stride at each convolutional layer. List length is number of convolutional layers. fcSize (int list): Number of nodes at each fully-connected (i.e. dense) layer of the encoder. List length is number of fully-connected layers.
- **activationFunc** (*str*) – Activation function for all layers except the last one. Current options: ['sigmoid', 'relu', 'linear'].
- **weightInitOpt** (*string*) – Method of weight initialisation. Current options: ['gaussian', 'truncated_normal', 'xavier', 'xavier_improved'].
- **weightStd** (*float*) – Used by 'gaussian' and 'truncated_normal' weight initialisation methods.
- **padding** (*str*) – Type of padding used. Current options: ['VALID', 'SAME'].

inputSize

Number of dimensions of input data (i.e. number of spectral bands).

Type int

activationFunc
Activation function for all layers except the last one.

Type str

weightInitOpt
Method of weight initialisation.

Type string

weightStd
Parameter for 'gaussian' and 'truncated_normal' weight initialisation methods.

Type float

convFilterSize
Size of filter at each convolutional layer. List length is number of convolutional layers.

Type int list

convNumFilters
Number of filters at each convolutional layer of the network. List length is number of convolutional layers.

Type int list

convStride
Stride at each convolutional layer. List length is number of convolutional layers. padding (str): Type of padding used. Current options: ['VALID', 'SAME'].

Type int list

fcSize
Number of nodes at each fully-connected (i.e. dense) layer of the encoder. List length is number of fully-connected layers.

Type int list

numLayers
Total number of layers (convolutional and fully-connected).

Type int

y_pred
Output of network - class scores with shape [numSamples x numClasses]. Accessible through the *predict_scores* class functions, requiring a trained model.

Type tensor

train_ops
Dictionary of names of train and loss ops (suffixed with _train and _loss) added to the network using the *add_train_op* class function. The name (without suffix) is passed to the *train* class function to train the network with the referenced train and loss op.

Type dict

modelsAddrs
Dictionary of model names added to the network using the *add_model* class function. The names reference models which can be used by the *predict_scores*, *predict_labels* and *predict_features* class functions.

Type dict

add_model(*addr*, *modelName*)
Loads a saved set of model parameters for the network.

Parameters

- **addr** (*str*) – Address of the directory containing the checkpoint files.
- **modelName** (*str*) – Name of the model (to refer to it later in-case of multiple models for a given network).

add_train_op(*name*, *balance_classes=True*, *learning_rate=0.001*, *decay_steps=None*, *decay_rate=None*, *piecewise_bounds=None*, *piecewise_values=None*, *method='Adam'*, *wd_lambda=0.0*)

Constructs a loss op and training op from a specific loss function and optimiser. User gives the ops a name, and the train op and loss opp are stored in a dictionary (train_ops) under that name.

Parameters

- **name** (*str*) – Name of the training op (to refer to it later in-case of multiple training ops).
- **balance_classes** (*boolean*) – Weight the samples during training so that the contribtion to the loss of each class is balanced by the number of samples the class has (in a given batch).
- **learning_rate** (*float*) – Controls the degree to which the weights are updated during training.
- **decay_steps** (*int*) – Epoch frequency at which to decay the learning rate.
- **decay_rate** (*float*) – Fraction at which to decay the learning rate.
- **piecewise_bounds** (*int list*) – Epoch step intervals for decaying the learning rate. Alternative to decay steps.
- **piecewise_values** (*float list*) – Rate at which to decay the learning rate at the piecewise_bounds.
- **method** (*str*) – Optimisation method.
- **wd_lambda** (*float*) – Scalar to control weighting of weight decay in loss.

predict_features(*modelName*, *dataSamples*, *layer*)

Extract the predicted feature values at a particular layer of the network.

Parameters

- **modelName** (*str*) – Name of the model to use (previously added with add_model())
- **dataSamples** (*np.array*) – Shape [numSamples x inputSize]
- **layer** (*int*) – Layer at which to extract features. Must be between 1 and numLayers inclusive.

Returns Values of neurons at layer. Shape [numSamples x numNeurons] if fully-connected layer and [numSamples x convDim1 x convDim2] if convolutional layer.

Return type (np.array)

predict_labels(*modelName*, *dataSamples*)

Extract the predicted classification labels of some dataSamples using a trained model.

Parameters

- **modelName** (*str*) – Name of the model to use (previously added with add_model())
- **dataSamples** (*array*) – Shape [numSamples x inputSize]

Returns Predicted classification labels of dataSamples. Shape [numSamples].

Return type (np.array)

predict_scores(*modelName*, *dataSamples*, *useSoftmax=True*)
Extract the predicted classification scores of some dataSamples using a trained model.

Parameters

- **modelName** (*str*) – Name of the model to use (previously added with add_model()).
- **dataSamples** (*np.array*) – Shape [numSamples x inputSize].
- **useSoftmax** (*boolean*) – Pass predicted scores output by network through a softmax function.

Returns Predicted classification scores of dataSamples. Shape [numSamples x numClasses].

Return type (np.array)

train(*dataTrain, dataVal, train_op_name, n_epochs, save_addr, visualiseRateTrain=0, visualiseRateVal=0, save_epochs=[1000]*)
Calls network_ops function to train a network.

Parameters

- **dataTrain** (*obj*) – Iterator object for training data.
- **dataVal** (*obj*) – Iterator object for validation data.
- **train_op_name** (*str*) – Name of training op created.
- **n_epochs** (*int*) – Number of loops through dataset to train for.
- **save_addr** (*str*) – Address of a directory to save checkpoints for desired epochs, or address of saved checkpoint. If address is for an epoch and contains a previously saved checkpoint, then the network will start training from there. Otherwise it will be trained from scratch.
- **visualiseRateTrain** (*int*) – Epoch rate at which to print training loss in console.
- **visualiseRateVal** (*int*) – Epoch rate at which to print validation loss in console.
- **save_epochs** (*int list*) – Epochs to save checkpoints at.

CHAPTER 5

Code Examples

Each example has a block of code for training and a block of code for testing. These should be run as separate scripts.

5.1 autoencoder examples

Each example has a block of code for training and a block of code for testing. These should be run as separate scripts.

5.1.1 train and test a basic MLP

The code block directly below will train an MLP (or dense) autoencoder on the Pavia Uni hyperspectral dataset. Make sure you have a folder in your directory called 'models'. Once trained, look at the next code block to test out the trained autoencoder. If you have already downloaded the Pavia Uni dataset (e.g. from another example) you can comment out that step.

The network has three encoder and three decoder layers, with 50 neurons in the first layer, 30 in the second and 10 in the third (the latent layer). A model is trained with 200,000 spectral samples and 100 validation samples with a batch size of 1000 samples. Training lasts for 100 epochs, with a learning rate of 0.001, the Adam optimiser and cosine spectral angle (CSA) reconstruction loss function. The train loss and validation loss are displayed every 10 epochs. Models are saved at 50 and 100 epochs. The models are saved in the models/test_ae_mlp folder.

```
import deephyp.data
import deephyp.autoencoder

import scipy.io
import os
import shutil
try:
    from urllib import urlretrieve # python2
except:
    from urllib.request import urlretrieve # python3

```

(continues on next page)

(continued from previous page)

```
# download dataset (if already downloaded, comment this out)
urlretrieve( 'http://www.ehu.eus/ccwintco/uploads/e/ee/PaviaU.mat', os.path.join(os.
↪getcwd(),'PaviaU.mat') )

# read data into numpy array
mat = scipy.io.loadmat( 'PaviaU.mat' )
img = mat[ 'paviaU' ]

# create a hyperspectral dataset object from the numpy array
hypData = deephyp.data.HypImg( img )

# pre-process data to make the model easier to train
hypData.pre_process( 'minmax' )

# create data iterator objects for training and validation using the pre-processed␣
↪data
trainSamples = 200000
valSamples = 100
dataTrain = deephyp.data.Iterator( dataSamples=hypData.spectraPrep[:trainSamples, :],
                          targets=hypData.spectraPrep[:trainSamples, :],␣
↪batchSize=1000 )
dataVal = deephyp.data.Iterator( dataSamples=hypData.
↪spectraPrep[trainSamples:trainSamples+valSamples, :],
                        targets=hypData.
↪spectraPrep[trainSamples:trainSamples+valSamples, :] )

# shuffle training data
dataTrain.shuffle()

# setup a fully-connected autoencoder neural network with 3 encoder layers
net = deephyp.autoencoder.mlp_1D_network( inputSize=hypData.numBands,␣
↪encoderSize=[50,30,10], activationFunc='relu',
                                    weightInitOpt='truncated_normal', tiedWeights=None,
↪ skipConnect=False )

# setup a training operation for the network
net.add_train_op( name='csa', lossFunc='CSA', learning_rate=1e-3, decay_steps=None,␣
↪decay_rate=None,
                  method='Adam', wd_lambda=0.0 )

# create a directory to save the learnt model
model_dir = os.path.join('models','test_ae_mlp')
if os.path.exists(model_dir):
    # if directory already exists, delete it
    shutil.rmtree(model_dir)
os.mkdir(model_dir)

# train the network for 100 epochs, saving the model at epoch 50 and 100
net.train(dataTrain=dataTrain, dataVal=dataVal, train_op_name='csa', n_epochs=100,␣
↪save_addr=model_dir,
          visualiseRateTrain=10, visualiseRateVal=10, save_epochs=[50,100])
```

The code below will test a trained MLP (or dense) autoencoder on the Pavia Uni hyperspectral dataset. Make sure you have a folder in your directory called 'results'. The network can be trained using the above code block. Run the testing code block as a separate script to the training code block.

The network is setup using the config file output during training. Then the 100 epoch model is added (named 'csa_100'). The model is used to encode a latent representation of the Pavia Uni data, and reconstruct it from the latent representation. A figure of the latent vector for a 'meadow' spectral sample and the reconstruction is saved in the results folder.

```
import deephyp.data
import deephyp.autoencoder

import scipy.io
import matplotlib.pyplot as plt
import os
import numpy as np


 # read data into numpy array
 mat = scipy.io.loadmat( 'PaviaU.mat' )
 img = mat[ 'paviaU' ]

 # create a hyperspectral dataset object from the numpy array
 hypData = deephyp.data.HypImg( img )

 # pre-process data to make the model easier to train
 hypData.pre_process( 'minmax' )

 # setup a network from a config file
 net = deephyp.autoencoder.mlp_1D_network( configFile=os.path.join('models','test_ae_
↪mlp','config.json') )

 # assign previously trained parameters to the network, and name model
 net.add_model( addr=os.path.join('models','test_ae_mlp','epoch_100'), modelName='csa_
↪100' )

 # feed forward hyperspectral dataset through encoder (get latent encoding)
 dataZ = net.encoder( modelName='csa_100', dataSamples=hypData.spectraPrep )

 # feed forward latent encoding through decoder (get reconstruction)
 dataY = net.decoder(modelName='csa_100', dataZ=dataZ)


 #--------- visualisation ----------------------------------------

 # reshape latent encoding to original image dimensions
 imgZ = np.reshape(dataZ, (hypData.numRows, hypData.numCols, -1))

 # reshape reconstructed output of decoder
 imgY = np.reshape(dataY, (hypData.numRows, hypData.numCols, -1))

 # reshape pre-processed input
 imgX = np.reshape(hypData.spectraPrep, (hypData.numRows, hypData.numCols, -1))

 # visualise latent image using 3 out of the 10 dimensions
 colourImg = imgZ.copy()
 colourImg = colourImg[ :,:,np.argsort(-np.std(np.std(colourImg, axis=0),␣
↪axis=0))[:3] ]
 colourImg /= np.max(np.max(colourImg, axis=0), axis=0)

 # save a latent image (using 3 out of the 10 dimensions)
 plt.imsave(os.path.join('results', 'test_mlp_latentImg.png'), colourImg)
```

(continues on next page)

(continued from previous page)

```
# save plot of latent vector of 'meadow' spectra
fig = plt.figure()
plt.plot(imgZ[576, 210, :])
plt.xlabel('latent dimension')
plt.ylabel('latent value')
plt.title('meadow spectra')
plt.savefig(os.path.join('results', 'test_mlp_latentVector.png'))

# save plot comparing pre-processed 'meadow' spectra input with decoder␣
↪reconstruction
fig = plt.figure()
ax = plt.subplot(111)
ax.plot(range(hypData.numBands),imgX[576, 210, :],label='pre-processed input')
ax.plot(range(hypData.numBands),imgY[576, 210, :],label='reconstruction')
plt.xlabel('band')
plt.ylabel('value')
plt.title('meadow spectra')
ax.legend()
plt.savefig(os.path.join('results', 'test_mlp_InputVsReconstruct.png'))
```

5.1.2 train multiple models for an MLP

The code block directly below will train several different models for a given MLP (or dense) autoencoder architecture on the Pavia Uni hyperspectral dataset. Each model is trained with a different reconstruction loss function. Make sure you have a folder in your directory called 'models'. Once trained, look at the next code block to test out the trained autoencoder. If you have already downloaded the Pavia Uni dataset (e.g. from another example) you can comment out that step.

The network has four encoder and four decoder layers, with 50 neurons in the first layer, 30 in the second, 10 in the third and 3 in the fourth layer (the latent layer). Models are trained with 200,000 spectral samples and 100 validation samples with a batch size of 1000 samples. Training lasts for 100 epochs, with a learning rate of 0.001 and the Adam optimiser. Three different models are trained, each with a different reconstruction loss function: the sum-of-squared errors (SSE), cosine spectral angle (CSA) and spectral angle (SA). The train loss and validation loss are displayed every 10 epochs. Models are saved at 50 and 100 epochs. The models are saved in the models/test_ae_mlp_adv_csa, models/test_ae_mlp_adv_sa and models/test_ae_mlp_adv_sse folders. Note that all of these models use the same network object.

```
import deephyp.data
import deephyp.autoencoder

import scipy.io
import os
import shutil
try:
    from urllib import urlretrieve # python2
except:
    from urllib.request import urlretrieve # python3


# download dataset (if already downloaded, comment this out)
urlretrieve( 'http://www.ehu.eus/ccwintco/uploads/e/ee/PaviaU.mat', os.path.join(os.
↪getcwd(),'PaviaU.mat') )
```

(continues on next page)

(continued from previous page)

```
# read data into numpy array
mat = scipy.io.loadmat( 'PaviaU.mat' )
img = mat[ 'paviaU' ]

# create a hyperspectral dataset object from the numpy array
hypData = deephyp.data.HypImg( img )

# pre-process data to make the model easier to train
hypData.pre_process( 'minmax' )

# create data iterator objects for training and validation using the pre-processed␣
↪data
trainSamples = 200000
valSamples = 100
dataTrain = deephyp.data.Iterator( dataSamples=hypData.spectraPrep[:trainSamples, :],
                          targets=hypData.spectraPrep[:trainSamples, :],␣
↪batchSize=1000 )
dataVal = deephyp.data.Iterator( dataSamples=hypData.
↪spectraPrep[trainSamples:trainSamples+valSamples, :],
                        targets=hypData.
↪spectraPrep[trainSamples:trainSamples+valSamples, :] )

# shuffle training data
dataTrain.shuffle()

# setup a fully-connected autoencoder neural network with 3 encoder layers
net = deephyp.autoencoder.mlp_1D_network( inputSize=hypData.numBands,␣
↪encoderSize=[50,30,10,3], activationFunc='relu',
                                    weightInitOpt='truncated_normal', tiedWeights=None,
↪ skipConnect=False )

# setup multiple training operations for the network (with different loss functions)
net.add_train_op(name='sse', lossFunc='SSE', learning_rate=1e-3, decay_steps=None,␣
↪decay_rate=None,
                 method='Adam', wd_lambda=0.0)

net.add_train_op( name='csa', lossFunc='CSA', learning_rate=1e-3, decay_steps=None,␣
↪decay_rate=None,
                  method='Adam', wd_lambda=0.0 )

net.add_train_op(name='sa', lossFunc='SA', learning_rate=1e-3, decay_steps=None,␣
↪decay_rate=None,
                 method='Adam', wd_lambda=0.0)


# create directories to save the learnt models
for method in ['sse','csa','sa']:
    model_dir = os.path.join('models','test_ae_mlp_adv_%s'%(method))
    if os.path.exists(model_dir):
        # if directory already exists, delete it
        shutil.rmtree(model_dir)
    os.mkdir(model_dir)

    # train a model for each training op
    dataTrain.reset_batch()
    net.train(dataTrain=dataTrain, dataVal=dataVal, train_op_name=method, n_
↪epochs=100, save_addr=model_dir,
```

(continues on next page)

(continued from previous page)

```
                visualiseRateTrain=10, visualiseRateVal=10, save_epochs=[50, 100])
```

The code below will test the MLP (or dense) autoencoder models trained in the above code block, on the Pavia Uni hyperspectral dataset. Make sure you have a folder in your directory called 'results'. Run the testing code block as a separate script to the training code block. The code block below downloads the Pavia Uni ground truth labels.

The network is setup using the config file output during training. Because all three models use the same network, the network can be setup from just one of the config files. Each of the three trained models are added to the network. The models are each used to encode a latent representation of the Pavia Uni data and a scatter plot figure of the samples in two of the three latent dimensions are shown for each model. The two latent features with the greatest standard deviation of the data samples are used for the scatter plot.

```
import deephyp.data
import deephyp.autoencoder

import scipy.io
import matplotlib.pyplot as plt
import os
import numpy as np
try:
    from urllib import urlretrieve # python2
except:
    from urllib.request import urlretrieve # python3


 # read data into numpy array
 mat = scipy.io.loadmat( 'PaviaU.mat' )
 img = mat[ 'paviaU' ]

 # create a hyperspectral dataset object from the numpy array
 hypData = deephyp.data.HypImg( img )

 # pre-process data to make the model easier to train
 hypData.pre_process( 'minmax' )

 # setup a network from a config file
 net = deephyp.autoencoder.mlp_1D_network( configFile=os.path.join('models','test_ae_
↪mlp_adv_sse','config.json') )

 # assign previously trained parameters to the network, and name each model
 net.add_model( addr=os.path.join('models','test_ae_mlp_adv_sse','epoch_100'),␣
↪modelName='sse_100' )
 net.add_model(addr=os.path.join('models', 'test_ae_mlp_adv_csa', 'epoch_100'),␣
↪modelName='csa_100')
 net.add_model(addr=os.path.join('models', 'test_ae_mlp_adv_sa', 'epoch_100'),␣
↪modelName='sa_100')

 # feed forward hyperspectral dataset through each encoder model (get latent encoding)
 dataZ_sse = net.encoder( modelName='sse_100', dataSamples=hypData.spectraPrep )
 dataZ_csa = net.encoder(modelName='csa_100', dataSamples=hypData.spectraPrep)
 dataZ_sa = net.encoder(modelName='sa_100', dataSamples=hypData.spectraPrep)

 # feed forward latent encoding through each decoder model (get reconstruction)
 dataY_sse = net.decoder(modelName='sse_100', dataZ=dataZ_sse)
 dataY_csa = net.decoder(modelName='csa_100', dataZ=dataZ_csa)
 dataY_sa = net.decoder(modelName='sa_100', dataZ=dataZ_sa)
```

(continues on next page)

(continued from previous page)

```
#--------- visualisation ----------------------------------------

# download dataset ground truth pixel labels (if already downloaded, comment this␣
↪out).
urlretrieve( 'http://www.ehu.eus/ccwintco/uploads/5/50/PaviaU_gt.mat',
                        os.path.join(os.getcwd(), 'PaviaU_gt.mat') )

# read labels into numpy array
mat_gt = scipy.io.loadmat( 'PaviaU_gt.mat' )
img_gt = mat_gt['paviaU_gt']
gt = np.reshape( img_gt , ( -1 ) )

method = ['sse','csa','sa']

dataZ_collection = [dataZ_sse, dataZ_csa, dataZ_sa]
for j,dataZ in enumerate(dataZ_collection):

    # save a scatter plot image of 2 of 3 latent dimensions
    idx = np.argsort(-np.std(dataZ, axis=0))
    fig, ax = plt.subplots()
    for i,gt_class in enumerate(['asphalt', 'meadow', 'gravel','tree','painted metal
↪','bare soil','bitumen','brick','shadow']):
        ax.scatter(dataZ[gt == i+1, idx[0]], dataZ[gt == i+1, idx[1]], c='C%i'%i,s=5,
↪label=gt_class)
    ax.legend()
    plt.title('latent representation: %s'%(method[j]))
    plt.xlabel('latent feature %i' % (idx[0]))
    plt.ylabel('latent feature %i' % (idx[1]))
    plt.savefig(os.path.join('results', 'test_mlp_scatter_%s.png'%(method[j])))
```

5.1.3 train an MLP with the SID loss function

The code block directly below will train an MLP (or dense) autoencoder on the Pavia Uni hyperspectral dataset. Make sure you have a folder in your directory called 'models'. Once trained, look at the next code block to test out the trained autoencoder. If you have already downloaded the Pavia Uni dataset (e.g. from another example) you can comment out that step.

The network has three encoder and three decoder layers, with 50 neurons in the first layer, 30 in the second and 10 in the third (the latent layer). A model is trained with 200,000 spectral samples and 100 validation samples with a batch size of 1000 samples. Training lasts for 100 epochs, with a learning rate of 0.001, the Adam optimiser and spectral information divergence (SID) reconstruction loss function. The train loss and validation loss are displayed every 10 epochs. Models are saved at 50 and 100 epochs. The models are saved in the models/test_ae_mlp_sid folder.

Since the SID loss contains log in its expression which is undefined for values <= 0, it is best to use sigmoid as the activation function (including the final activation function) for networks trained with the SID loss.

```
import deephyp.data
import deephyp.autoencoder

import scipy.io
import os
import shutil
try:
```

(continues on next page)

(continued from previous page)

```
    from urllib import urlretrieve # python2
except:
    from urllib.request import urlretrieve # python3


# download dataset (if already downloaded, comment this out)
urlretrieve( 'http://www.ehu.eus/ccwintco/uploads/e/ee/PaviaU.mat', os.path.join(os.
↪getcwd(),'PaviaU.mat') )

# read data into numpy array
mat = scipy.io.loadmat( 'PaviaU.mat' )
img = mat[ 'paviaU' ]

# create a hyperspectral dataset object from the numpy array
hypData = deephyp.data.HypImg( img )

# pre-process data to make the model easier to train
hypData.pre_process( 'minmax' )

# create data iterator objects for training and validation using the pre-processed␣
↪data
trainSamples = 200000
valSamples = 100
dataTrain = deephyp.data.Iterator( dataSamples=hypData.spectraPrep[:trainSamples, :],
                          targets=hypData.spectraPrep[:trainSamples, :],␣
↪batchSize=1000 )
dataVal = deephyp.data.Iterator( dataSamples=hypData.
↪spectraPrep[trainSamples:trainSamples+valSamples, :],
                        targets=hypData.
↪spectraPrep[trainSamples:trainSamples+valSamples, :] )

# shuffle training data
dataTrain.shuffle()

# setup a fully-connected autoencoder neural network with 3 encoder layers
net = deephyp.autoencoder.mlp_1D_network( inputSize=hypData.numBands,␣
↪encoderSize=[50,30,10], activationFunc='sigmoid',
                                  weightInitOpt='truncated_normal', tiedWeights=None,
↪ skipConnect=False,
                                  activationFuncFinal='sigmoid' )

# setup a training operation for the network
net.add_train_op( name='sid', lossFunc='SID', learning_rate=1e-3, decay_steps=None,␣
↪decay_rate=None,
                  method='Adam', wd_lambda=0.0 )

# create a directory to save the learnt model
model_dir = os.path.join('models','test_ae_mlp_sid')
if os.path.exists(model_dir):
    # if directory already exists, delete it
    shutil.rmtree(model_dir)
os.mkdir(model_dir)

# train the network for 100 epochs, saving the model at epoch 50 and 100
net.train(dataTrain=dataTrain, dataVal=dataVal, train_op_name='sid', n_epochs=100,␣
↪save_addr=model_dir,
          visualiseRateTrain=10, visualiseRateVal=10, save_epochs=[100])
```

The code below will test a trained MLP (or dense) autoencoder on the Pavia Uni hyperspectral dataset. Make sure you have a folder in your directory called 'results'. The network can be trained using the above code block. Run the testing code block as a separate script to the training code block.

The network is setup using the config file output during training. Then the 100 epoch model is added (named 'sid_100'). The model is used to encode a latent representation of the Pavia Uni data and a scatter plot figure of the samples in two of the ten latent dimensions are shown for each model. The two latent features with the greatest standard deviation of the data samples are used for the scatter plot.

```
import deephyp.data
import deephyp.autoencoder

import scipy.io
import matplotlib.pyplot as plt
import os
import numpy as np
try:
    from urllib import urlretrieve # python2
except:
    from urllib.request import urlretrieve # python3


 # read data into numpy array
 mat = scipy.io.loadmat( 'PaviaU.mat' )
 img = mat[ 'paviaU' ]

 # create a hyperspectral dataset object from the numpy array
 hypData = deephyp.data.HypImg( img )

 # pre-process data to make the model easier to train
 hypData.pre_process( 'minmax' )

 # setup a network from a config file
 net = deephyp.autoencoder.mlp_1D_network( configFile=os.path.join('models','test_ae_
↪mlp_sid','config.json') )

 # assign previously trained parameters to the network, and name model
 net.add_model( addr=os.path.join('models','test_ae_mlp_sid','epoch_100'), modelName=
↪'sid_100' )

 # feed forward hyperspectral dataset through encoder (get latent encoding)
 dataZ = net.encoder( modelName='sid_100', dataSamples=hypData.spectraPrep )

 # feed forward latent encoding through decoder (get reconstruction)
 dataY = net.decoder(modelName='sid_100', dataZ=dataZ)


 #--------- visualisation ----------------------------------------

 # download dataset ground truth pixel labels (if already downloaded, comment this␣
↪out)
 urlretrieve( 'http://www.ehu.eus/ccwintco/uploads/5/50/PaviaU_gt.mat',
                    os.path.join(os.getcwd(), 'PaviaU_gt.mat') )

 # read labels into numpy array
 mat_gt = scipy.io.loadmat( 'PaviaU_gt.mat' )
 img_gt = mat_gt['paviaU_gt']
 gt = np.reshape( img_gt , ( -1 ) )
```

(continues on next page)

(continued from previous page)

```
# save a scatter plot image of 2 of 3 latent dimensions
idx = np.argsort(-np.std(dataZ, axis=0))
fig, ax = plt.subplots()
for i,gt_class in enumerate(['asphalt', 'meadow', 'gravel','tree','painted metal',
↪'bare soil','bitumen','brick','shadow']):
    ax.scatter(dataZ[gt == i+1, idx[0]], dataZ[gt == i+1, idx[1]], c='C%i'%i,s=5,
↪label=gt_class)
ax.legend()
plt.title('latent representation: sid')
plt.xlabel('latent feature %i' % (idx[0]))
plt.ylabel('latent feature %i' % (idx[1]))
plt.savefig(os.path.join('results', 'test_mlp_scatter_sid.png'))
```

5.1.4 train and compare an MLP and CNN autoencoder

The code block directly below will train an MLP (or dense) autoencoder and a CNN autoencoder on the Pavia Uni hyperspectral dataset. Make sure you have a folder in your directory called 'models'. Once trained, look at the next code block to test out the trained autoencoders. If you have already downloaded the Pavia Uni dataset (e.g. from another example) you can comment out that step.

The MLP network has four encoder and four decoder layers, with 50 neurons in the first layer, 30 in the second, 10 in the third and 3 in the fourth layer (the latent layer) of the encocer. The CNN network has an encoder with three convolutional layers and a fully-connected layer joining the output of the third convolutional layer to the latent layer. The decoder has a fully-connected layer joining the latent layer to the deconvolutional layers followed by three deconvolutional layers. The first convolutional layer has 10 filters of size 20, with the second and third both having 10 filters of size 10. All convolutional layers have a stride of 1. The decoder is symmetric. The CNN latent layer has 3 neurons (to have the same dimensionality as the MLP).

Both the mlp and cnn models are trained with 200,000 spectral samples and 100 validation samples with a batch size of 1000 samples, with a learning rate of 0.001 and the Adam optimiser. The MLP is trained for 100 epochs and the CNN is trained for 10 epochs. Both networks are trained with the cosine spectral angle (CSA) reconstruction loss function. The train loss and validation loss are visualised every 10 epochs, except for the CNN training loss which is visualised every 1 epoch. The MLP is saved at 100 epochs and the CNN is saved at 10 epochs. The models are saved in the models/test_ae_comparison_mlp and models/test_ae_comparison_cnn folders.

```
import deephyp.data
import deephyp.autoencoder

import scipy.io
import os
import shutil
try:
    from urllib import urlretrieve # python2
except:
    from urllib.request import urlretrieve # python3


# download dataset (if already downloaded, comment this out)
urlretrieve( 'http://www.ehu.eus/ccwintco/uploads/e/ee/PaviaU.mat', os.path.join(os.
↪getcwd(),'PaviaU.mat') )

# read data into numpy array
```

(continues on next page)

(continued from previous page)

```
mat = scipy.io.loadmat( 'PaviaU.mat' )
img = mat[ 'paviaU' ]

# create a hyperspectral dataset object from the numpy array
hypData = deephyp.data.HypImg( img )

# pre-process data to make the model easier to train
hypData.pre_process( 'minmax' )

# create data iterator objects for training and validation using the pre-processed␣
↪data
trainSamples = 200000
valSamples = 100
dataTrain = deephyp.data.Iterator( dataSamples=hypData.spectraPrep[:trainSamples, :],
                          targets=hypData.spectraPrep[:trainSamples, :],␣
↪batchSize=1000 )
dataVal = deephyp.data.Iterator( dataSamples=hypData.
↪spectraPrep[trainSamples:trainSamples+valSamples, :],
                        targets=hypData.
↪spectraPrep[trainSamples:trainSamples+valSamples, :] )

# shuffle training data
dataTrain.shuffle()

# setup a fully-connected autoencoder neural network with 3 encoder layers
net_mlp = deephyp.autoencoder.mlp_1D_network( inputSize=hypData.numBands,␣
↪encoderSize=[50,30,10,3], activationFunc='relu',
                                    weightInitOpt='truncated_normal', tiedWeights=None,
↪ skipConnect=False )

# setup a convolutional autoencoder neural network with 3 conv encoder layers
net_cnn = deephyp.autoencoder.cnn_1D_network( inputSize=hypData.numBands, zDim=3,␣
↪encoderNumFilters=[10,10,10] ,
                                    encoderFilterSize=[20,10,10], activationFunc='relu',
↪ weightInitOpt='truncated_normal',
                                    encoderStride=[1, 1, 1], tiedWeights=None,␣
↪skipConnect=False )

# setup a training operation for each network (using the same loss function)
net_mlp.add_train_op(name='csa', lossFunc='CSA', learning_rate=1e-3, decay_
↪steps=None, decay_rate=None,
                 method='Adam', wd_lambda=0.0)

net_cnn.add_train_op( name='csa', lossFunc='CSA', learning_rate=1e-3, decay_
↪steps=None, decay_rate=None,
                  method='Adam', wd_lambda=0.0 )



# create directories to save the learnt models
model_dirs = []
for method in ['mlp','cnn']:
    model_dir = os.path.join('models','test_ae_comparison_%s'%(method))
    if os.path.exists(model_dir):
        # if directory already exists, delete it
        shutil.rmtree(model_dir)
    os.mkdir(model_dir)
```

(continues on next page)

(continued from previous page)

```
    model_dirs.append( model_dir )

# train the mlp model (100 epochs)
dataTrain.reset_batch()
net_mlp.train(dataTrain=dataTrain, dataVal=dataVal, train_op_name='csa', n_
↪epochs=100, save_addr=model_dirs[0],
          visualiseRateTrain=10, visualiseRateVal=10, save_epochs=[100])

# train the cnn model (takes longer, so only 10 epochs)
dataTrain.reset_batch()
net_cnn.train(dataTrain=dataTrain, dataVal=dataVal, train_op_name='csa', n_epochs=10,
↪ save_addr=model_dirs[1],
          visualiseRateTrain=1, visualiseRateVal=10, save_epochs=[10])
```

The code below will test the MLP (or dense) and CNN autoencoder models trained in the above code block, on the Pavia Uni hyperspectral dataset. Make sure you have a folder in your directory called 'results'. Run the testing code block as a separate script to the training code block. The code block below downloads the Pavia Uni ground truth labels.

The networks are setup using the config files output during training. Each of the models are added to their respective networks. The models are each used to encode a latent representation of the Pavia Uni data and a scatter plot figure of the samples in two of the three latent dimensions are shown for each model. The two latent features with the greatest standard deviation of the data samples are used for the scatter plot.

```
import deephyp.data
import deephyp.autoencoder

import scipy.io
import matplotlib.pyplot as plt
import os
import numpy as np
try:
    from urllib import urlretrieve # python2
except:
    from urllib.request import urlretrieve # python3


# read data into numpy array
mat = scipy.io.loadmat( 'PaviaU.mat' )
img = mat[ 'paviaU' ]

# create a hyperspectral dataset object from the numpy array
hypData = deephyp.data.HypImg( img )

# pre-process data to make the model easier to train
hypData.pre_process( 'minmax' )

# setup each network from the config files
net_mlp = deephyp.autoencoder.mlp_1D_network( configFile=os.path.join('models','test_
↪ae_comparison_mlp','config.json') )
net_cnn = deephyp.autoencoder.cnn_1D_network(configFile=os.path.join('models', 'test_
↪ae_comparison_cnn', 'config.json'))

# assign previously trained parameters to the network, and name each model
net_mlp.add_model( addr=os.path.join('models','test_ae_comparison_mlp','epoch_100'),␣
↪modelName='mlp_100' )
```

(continues on next page)

(continued from previous page)

```
net_cnn.add_model(addr=os.path.join('models', 'test_ae_comparison_cnn', 'epoch_10'),␣
↪modelName='cnn_10')


# feed forward hyperspectral dataset through each encoder model (get latent encoding)
dataZ_mlp = net_mlp.encoder( modelName='mlp_100', dataSamples=hypData.spectraPrep )
dataZ_cnn = net_cnn.encoder(modelName='cnn_10', dataSamples=hypData.spectraPrep)


# feed forward latent encoding through each decoder model (get reconstruction)
dataY_mlp = net_mlp.decoder(modelName='mlp_100', dataZ=dataZ_mlp)
dataY_cnn = net_cnn.decoder(modelName='cnn_10', dataZ=dataZ_cnn)



#--------- visualisation ----------------------------------------

# download dataset ground truth pixel labels (if already downloaded, comment this␣
↪out)
urlretrieve( 'http://www.ehu.eus/ccwintco/uploads/5/50/PaviaU_gt.mat',
                  os.path.join(os.getcwd(), 'PaviaU_gt.mat') )

# read labels into numpy array
mat_gt = scipy.io.loadmat( 'PaviaU_gt.mat' )
img_gt = mat_gt['paviaU_gt']
gt = np.reshape( img_gt , ( -1 ) )

method = ['mlp','cnn']

dataZ_collection = [dataZ_mlp, dataZ_cnn]
for j,dataZ in enumerate(dataZ_collection):

    # save a scatter plot image of 2 of 3 latent dimensions
    idx = np.argsort(-np.std(dataZ, axis=0))
    fig, ax = plt.subplots()
    for i,gt_class in enumerate(['asphalt', 'meadow', 'gravel','tree','painted metal
↪','bare soil','bitumen','brick','shadow']):
        ax.scatter(dataZ[gt == i+1, idx[0]], dataZ[gt == i+1, idx[1]], c='C%i'%i,s=5,
↪label=gt_class)
    ax.legend()
    plt.xlabel('latent feature %i'%(idx[0]))
    plt.ylabel('latent feature %i' % (idx[1]))
    plt.title('latent representation: %s'%(method[j]))
    plt.savefig(os.path.join('results', 'test_comparison_%s.png'%(method[j])))


# reshape reconstruction to original image dimensions
imgY_mlp = np.reshape(dataY_mlp, (hypData.numRows, hypData.numCols, -1))
imgY_cnn = np.reshape(dataY_cnn, (hypData.numRows, hypData.numCols, -1))
imgX = np.reshape(hypData.spectraPrep, (hypData.numRows, hypData.numCols, -1))

# save plot comparing pre-processed 'meadow' spectra input with decoder␣
↪reconstruction
fig = plt.figure()
ax = plt.subplot(111)
ax.plot(range(hypData.numBands),imgX[576, 210, :],label='pre-processed input')
ax.plot(range(hypData.numBands),imgY_mlp[576, 210, :],label='mlp reconstruction')
```

(continues on next page)

(continued from previous page)

```
ax.plot(range(hypData.numBands), imgY_cnn[576, 210, :], label='cnn reconstruction')
plt.xlabel('band')
plt.ylabel('value')
plt.title('meadow spectra')
ax.legend()
plt.savefig(os.path.join('results', 'test_reconstruct_comparison.png'))
```

5.2 classifier examples

Each example has a block of code for training and a block of code for testing. These should be run as separate scripts.

5.2.1 train and test a CNN classifier

The code block directly below will train CNN classifier on the Pavia Uni hyperspectral dataset. Make sure you have a folder in your directory called 'models'. Once trained, look at the next code block to test out the trained classifier. If you have already downloaded the Pavia Uni dataset and ground truth dataset (e.g. from another example) you can comment out that step.

The CNN classification network has three convolutional layers and three fully-connected layers (including the output layer). The first convolutional layer has 10 filters of size 20, with the second and third both having 10 filters of size 10. All convolutional layers have a stride of 1. The first two fully-connected layers both have 20 neurons and the final fully-connected layer has 9 neurons (because there are 9 classes). A ReLU activation function is used.

The CNN model is trained on 50 samples per each of the 9 classes (not including the background class, which has a label of zero). 15 samples per class are used for validation, with a batch size of 50. The network is trained for 1000 epochs using the cross-entropy loss function with class balancing (even though the number of samples per class is already balanced). Both the train and validation loss are visualised every 10 epochs and models are saved at epochs 100 and 1000. The models are saved in the models/test_clf_cnn folders.

```
import deephyp.data
import deephyp.classifier

import scipy.io
import os
import shutil
import numpy as np
try:
    from urllib import urlretrieve # python2
except:
    from urllib.request import urlretrieve # python3


 # download dataset and ground truth (if already downloaded, comment this out)
 urlretrieve( 'http://www.ehu.eus/ccwintco/uploads/e/ee/PaviaU.mat', os.path.join(os.
↪getcwd(),'PaviaU.mat') )
 urlretrieve('http://www.ehu.eus/ccwintco/uploads/5/50/PaviaU_gt.mat', os.path.
↪join(os.getcwd(), 'PaviaU_gt.mat') )

 # read data into numpy array
 mat = scipy.io.loadmat('PaviaU.mat')
 img = mat['paviaU']

 # read labels into numpy array
```

(continues on next page)

(continued from previous page)

```
mat_gt = scipy.io.loadmat('PaviaU_gt.mat')
img_gt = mat_gt['paviaU_gt']

# create a hyperspectral dataset object from the numpy array
hypData = deephyp.data.HypImg( img, labels=img_gt )

# pre-process data to make the model easier to train
hypData.pre_process( 'minmax' )

# get indices for training and validation data
trainSamples = 50 # per class
valSamples = 15 # per class
train_indices = []
for i in range(1,10):
    train_indices += np.nonzero(hypData.labels == i)[0][:trainSamples].tolist()
val_indices = []
for i in range(1,10):
    val_indices += np.nonzero(hypData.labels ==␣
↪i)[0][trainSamples:trainSamples+valSamples].tolist()

# create data iterator objects for training and validation using the pre-processed␣
↪data
dataTrain = deephyp.data.Iterator( dataSamples=hypData.spectraPrep[train_indices, :],
                          targets=hypData.labelsOnehot[train_indices,:],␣
↪batchSize=50 )
dataVal = deephyp.data.Iterator( dataSamples=hypData.spectraPrep[val_indices, :],
                        targets=hypData.labelsOnehot[val_indices,:] )

# shuffle training data
dataTrain.shuffle()

# setup a cnn classifier with 3 convolutional layers and 2 fully-connected layers
net = deephyp.classifier.cnn_1D_network( inputSize=hypData.numBands, numClasses=9,␣
↪convFilterSize=[20,10,10],
              convNumFilters=[10,10,10], convStride = [1,1,1], fcSize=[20,20],␣
↪activationFunc='relu',
              weightInitOpt='truncated_normal', weightStd=0.1, padding='VALID' )

# setup a training operation
net.add_train_op('basic50',balance_classes=True)

# create a directory to save the learnt model
model_dir = os.path.join('models', 'test_clf_cnn')
if os.path.exists(model_dir):
    # if directory already exists, delete it
    shutil.rmtree(model_dir)
os.mkdir(model_dir)

# train the network for 1000 epochs, saving the model at epoch 100 and 1000
net.train(dataTrain=dataTrain, dataVal=dataVal, train_op_name='basic50', n_
↪epochs=1000, save_addr=model_dir,
          visualiseRateTrain=10, visualiseRateVal=10, save_epochs=[100,1000])
```

The code below will test the CNN classifier model trained in the above code block, on the Pavia Uni hyperspectral dataset. Make sure you have a folder in your directory called 'results'. Run the testing code block as a separate script to the training code block.

The network is setup using the config file output during training. The model is added to the network (with the name 'basic_model'). Pavia Uni data samples from the entire image are passed through the network, which predicts labels and class labels and scores for each sample. Figures are saved showing the predicted class labels for the image with and without the background class masked out, as well as showing the ground truth labels.

```
import deephyp.data
import deephyp.classifier

import scipy.io
import pylab as pl
import os
import numpy as np


 # read data into numpy array
 mat = scipy.io.loadmat('PaviaU.mat')
 img = mat['paviaU']

 # create a hyperspectral dataset object from the numpy array
 hypData = deephyp.data.HypImg( img )

 # pre-process data to make the model easier to train
 hypData.pre_process('minmax')


 # setup a fully-connected autoencoder neural network with 3 encoder layers
 net = deephyp.classifier.cnn_1D_network(configFile=os.path.join('models','test_clf_
↪cnn','config.json'))

 # assign previously trained parameters to the network, and name model
 net.add_model( addr=os.path.join('models','test_clf_cnn','epoch_1000'), modelName=
↪'basic_model' )

 # feed forward hyperspectral dataset through the model to predict class labels and␣
↪scores for each sample
 data_pred = net.predict_labels( modelName='basic_model', dataSamples=hypData.
↪spectraPrep  )
 data_scores = net.predict_scores( modelName='basic_model', dataSamples=hypData.
↪spectraPrep  )

 # extract features at second last layer
 data_features = net.predict_features(modelName='basic_model', dataSamples=hypData.
↪spectraPrep, layer=net.numLayers-1)

 #--------- visualisation ----------------------------------------

 # reshape predicted labels to an image
 img_pred = np.reshape(data_pred, (hypData.numRows, hypData.numCols))

 # read labels into numpy array
 mat_gt = scipy.io.loadmat('PaviaU_gt.mat')
 img_gt = mat_gt['paviaU_gt']


 class_names = ['asphalt', 'meadow', 'gravel','tree','painted metal','bare soil',
↪'bitumen','brick','shadow']
 cmap = pl.cm.jet
```

(continues on next page)

(continued from previous page)

```
# save ground truth figure
pl.figure()
for entry in pl.unique(img_gt):
    colour = cmap(entry*255/(np.max(img_gt) - 0))
    pl.plot(0, 0, "-", c=colour, label=(['background']+class_names)[entry])
pl.imshow(img_gt,cmap=cmap)
pl.legend(bbox_to_anchor=(2, 1))
pl.title('ground truth labels')
pl.savefig(os.path.join('results', 'test_classification_gt.png'))

# save predicted classes figure
pl.figure()
for entry in pl.unique(img_pred):
    colour = cmap(entry*255/(np.max(img_pred) - 0))
    pl.plot(0, 0, "-", c=colour, label=class_names[entry-1])
pl.imshow(img_pred,cmap=cmap)
pl.legend(bbox_to_anchor=(2, 1))
pl.title('classification prediction')
pl.savefig(os.path.join('results', 'test_classification_pred.png'))

# save predicted classes figure with background masked out
img_pred[img_gt==0] = 0
pl.figure()
for entry in pl.unique(img_pred):
    colour = cmap(entry*255/(np.max(img_pred) - 0))
    pl.plot(0, 0, "-", c=colour, label=(['background']+class_names)[entry])
pl.imshow(img_pred,cmap=cmap)
pl.legend(bbox_to_anchor=(2, 1))
pl.title('classification prediction with background masked')
pl.savefig(os.path.join('results', 'test_classification_pred_bkgrd.png'))
```

CHAPTER 6

Related Publications

Some links to publications on deep learning for hyperspectral data:

- autoencoders: ICIP 2016, TIP 2017, Remote Sensing 2019
- CNNs for classification using data augmentation: BMVC 2017
- pre-training CNNs: TGRS 2018
- PhD thesis

CHAPTER 7

Contact

If you have any positive or negative feedback about *deephyp*, bugs to report or requests for new functionality in future versions, please get in contact via email:

lloydwindrim@gmail.com

Lloyd Windrim Ph.D

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Index

Y

Z