Delve Documentation

Release 0.1.50

Delve Developers

Apr 01, 2023

GETTING STARTED

Delve is a Python package for analyzing the inference dynamics of your model.

Use Delve if you need a lightweight PyTorch extension that:

- Gives you insight into the inference dynamics of your architecture
- Allows you to optimize and adjust neural networks models to your dataset without much trial and error
- Allows you to analyze the eigenspaces your data at different stages of inference
- Provides you basic tooling for experiment logging

CHAPTER

ONE

MOTIVATION

Designing a deep neural network is a trial and error heavy process that mostly revolves around comparing performance metrics of different runs. One of the key issues with this development process is that the results of metrics not really propagate back easily to concrete design improvements. Delve provides you with spectral analysis tools that allow you to investigate the inference dynamic evolving in the model while training. This allows you to spot underutilized and unused layers. Mismatches between object size and neural architecture among other inefficiencies. These observations can be propagated back directly to design changes in the architecture even before the model has fully converged, allowing for a quicker and more guided design process.

This work is closely related to Maithra Raghu (Google Brain) et al's work on SVCCA:

- "Maithra Raghu on the differences between wide and deep networks", 2020 [YouTube]
- "SVCCA:Singular Vector Canonical Correlation Analysis for Deep Learning and Interpretability", 2017 [arXiv]

CHAPTER

TWO

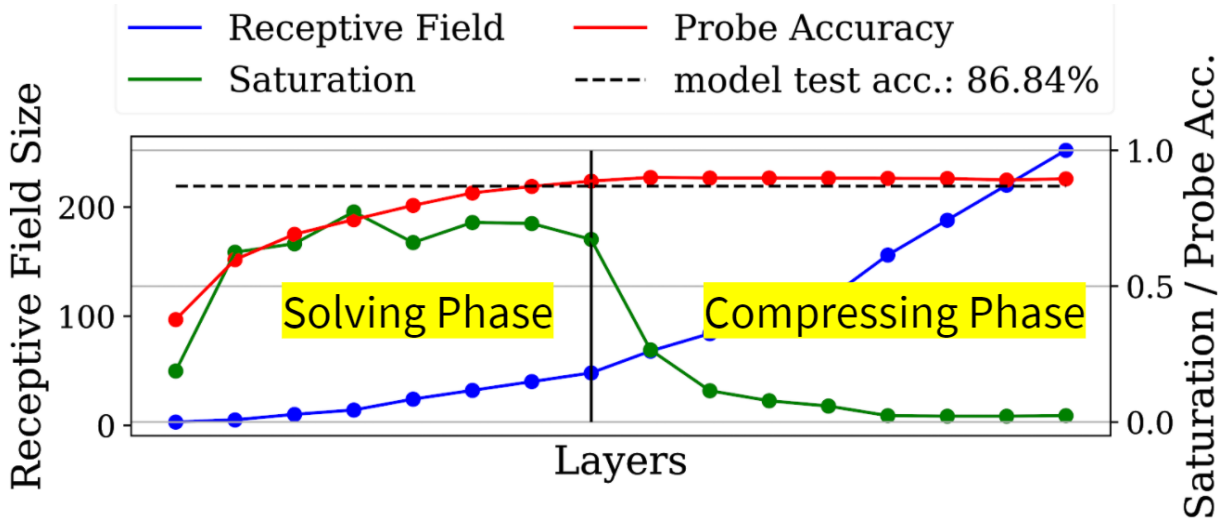
INSTALLATION

```
pip install delve
```

2.1 Using Layer Saturation to improve model performance

The saturation metric is the core feature of delve. By default saturation is a value between 0 and 1.0 computed for any convolutional, lstm or dense layer in the network. The saturation describes the percentage of eigendirections required for explaining 99% of the variance. Simply speaking, it tells you how much your data is "filling up" the individual layers inside your model.

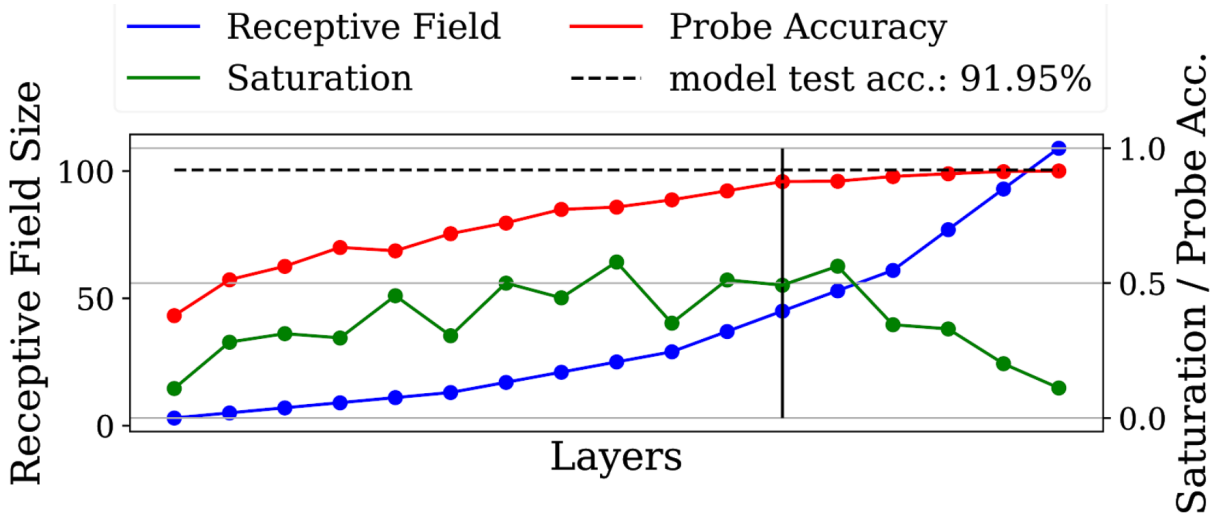
In the image below you can see how saturation portraits inefficiencies in your neural network. The depicted model is ResNet18 trained on 32 pixel images, which is way to small for a model with a receptive field exceeding 400 pixels in the final layers.



To visualize what this poorly chosen input resolution does to the inference, we trained logistic regressions on the output of every layer to solve the same task as the model. You can clearly see that only the first half of the model (at best) is improving the intermedia solutions of our logistic regression "probes". The layers following this are contributing nothing to the quality of the prediction! You also see that saturation is extremly low for this layers!

We call this a *tail* and it can be removed by either increasing the input resolution or (which is more economical) reducing the receptive field size to match the object size of your dataset.

We can do this by removing the first two downsampling layers, which quarters the growth of the receptive field of your network, which reduced not only the number of parameters but also makes more use of the available parameters, by



making more layers contribute effectivly!

For more details check our publication on this topics - Spectral Analysis of Latent Representations - Feature Space Saturation during Training - (Input) Size Matters for CNN Classifiers - Should you go deeper? Optimizing Convolutional Neural Networks without training - Go with the Flow: the distribution of information processing in multi-path networks (soon)

CHAPTER

THREE

DEMO

```
import torch
from delve import SaturationTracker
from torch.cuda import is_available
from torch.nn import CrossEntropyLoss
from torchvision.datasets import CIFAR10
from torchvision.transforms import ToTensor, Compose
from torch.utils.data.dataloader import DataLoader
from torch.optim import Adam
from torchvision.models.vgg import vgg16

# setup compute device
from tqdm import tqdm

if __name__ == "__main__":

  device = "cuda:0" if is_available() else "cpu"

  # Get some data
  train_data = CIFAR10(root="./tmp", train=True,
                       download=True, transform=Compose([ToTensor()]))
  test_data = CIFAR10(root="./tmp", train=False, download=True,␣
↪transform=Compose([ToTensor()]))

  train_loader = DataLoader(train_data, batch_size=1024,
                            shuffle=True, num_workers=6,
                            pin_memory=True)
  test_loader = DataLoader(test_data, batch_size=1024,
                           shuffle=False, num_workers=6,
                           pin_memory=True)

  # instantiate model
  model = vgg16(num_classes=10).to(device)

  # instantiate optimizer and loss
  optimizer = Adam(params=model.parameters())
  criterion = CrossEntropyLoss().to(device)

  # initialize delve
  tracker = SaturationTracker("my_experiment", save_to="plotcsv", modules=model,␣
↪device=device)
```

(continues on next page)

(continued from previous page)

```
  # begin training
  for epoch in range(10):
    model.train()
    for (images, labels) in tqdm(train_loader):
      images, labels = images.to(device), labels.to(device)
      prediction = model(images)
      optimizer.zero_grad(set_to_none=True)
      with torch.cuda.amp.autocast():
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)

        loss = criterion(outputs, labels)
      loss.backward()
      optimizer.step()

    total = 0
    test_loss = 0
    correct = 0
    model.eval()
    for (images, labels) in tqdm(test_loader):
      images, labels = images.to(device), labels.to(device)
      outputs = model(images)
      loss = criterion(outputs, labels)
      _, predicted = torch.max(outputs.data, 1)

      total += labels.size(0)
      correct += torch.sum((predicted == labels)).item()
      test_loss += loss.item()

    # add some additional metrics we want to keep track of
    tracker.add_scalar("accuracy", correct / total)
    tracker.add_scalar("loss", test_loss / total)

    # add saturation to the mix
    tracker.add_saturations()

  # close the tracker to finish training
  tracker.close()
```

CHAPTER

FOUR

SUPPORTED LAYERS

- Dense/Linear
- LSTM
- Convolutional

CHAPTER

FIVE

CITATION

If you use Delve in your publication, please cite:

```
@software{delve,
author       = {Justin Shenk and
                Mats L. Richter and
                Wolf Byttner and
                Michał Marcinkiewicz},
title        = {delve-team/delve: Latest},
month        = aug,
year         = 2021,
publisher    = {Zenodo},
version      = {v0.1.50},
doi          = {10.5281/zenodo.5233859},
url          = {https://doi.org/10.5281/zenodo.5233859}
}
```

5.1 Why this name, Delve?

delve (*verb*):

- reach inside a receptacle and search for something
- to carry on intensive and thorough research for data, information, or the like

5.1.1 Installation

Installing Delve

Delve require Python 3.6+ to be installed.

To install via pip:

```
pip install delve
```

To install the latest development version, clone the *GitHub* repository and use the setup script:

```
git clone https://github.com/delve-team/delve.git
cd delve
pip install .
```

Usage

Instantiate the `SaturationTracker` class where you define your PyTorch training loop, as in the example:

```
from torch import nn
from delve import SaturationTracker

...

model = nn.ModuleDict({
              'conv1': nn.Conv2d(1, 8, 3, padding=1),
              'linear1': nn.Linear(3, 1),
})


layers = [model.conv1, model.linear1]
stats = SaturationTracker('regression/h{}'.format(h),
   save_to="plotcsv",
   modules=layers,
   stats=["lsat"]
)

...

for _ in range(10):
   y_pred = model(x)
   loss = loss_fn(y_pred, y)
   optimizer.zero_grad()
   loss.backward()
   optimizer.step()

   stats.add_saturations()

stats.close()
```

This will hook into the layers in `layers` and log the statistics, in this case `lsat` (layer saturation). It will save images to `regression`.

5.1.2 Saturation

Saturation is a metric used for identifying the intrinsic dimensionality of features in a layer.

A visualization of how saturation changes over training and can be used to optimize network topology is provided at https://github.com/justinshenk/playground:

Covariance matrix of features is computed online:

$$Q(Z_l, Z_l) = \frac{\sum_{b=0}^{B} A_{l,b}^T A_{l,b}}{n} - (\bar{A}_l \bigotimes \bar{A}_l)$$

for B batches of layer output matrix A_l and n number of samples.

Note: For more information about how saturation is computed, read "Feature Space Saturation during Training".

5.1.3 Gallery

A gallery of examples

Extract layer saturation

Extract layer saturation with Delve.

```
import torch
from tqdm import trange

from delve import SaturationTracker


class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred


device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
torch.manual_seed(1)

for h in [3, 32]:
    # N is batch size; D_in is input dimension;
    # H is hidden dimension; D_out is output dimension.
    N, D_in, H, D_out = 64, 1000, h, 10

    # Create random Tensors to hold inputs and outputs
    x = torch.randn(N, D_in)
    y = torch.randn(N, D_out)
    x_test = torch.randn(N, D_in)
    y_test = torch.randn(N, D_out)

    # You can watch specific layers by handing them to delve as a list.
    # Also, you can hand over the entire Module-object to delve and let delve search for␣
↪recordable layers.
    model = TwoLayerNet(D_in, H, D_out)

    x, y, model = x.to(device), y.to(device), model.to(device)
    x_test, y_test = x_test.to(device), y_test.to(device)
```

(continues on next page)

(continued from previous page)

```
    layers = [model.linear1, model.linear2]
    stats = SaturationTracker('regression/h{}'.format(h),
                              save_to="plotcsv",
                              modules=layers,
                              device=device,
                              stats=["lsat", "lsat_eval"])

    loss_fn = torch.nn.MSELoss(reduction='sum')
    optimizer = torch.optim.SGD(model.parameters(), lr=1e-4, momentum=0.9)
    steps_iter = trange(2000, desc='steps', leave=True, position=0)
    steps_iter.write("{:^80}".format(
        "Regression - TwoLayerNet - Hidden layer size {}".format(h)))
    for step in steps_iter:
        # training step
        model.train()
        y_pred = model(x)
        loss = loss_fn(y_pred, y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # test step
        model.eval()
        y_pred = model(x_test)
        loss_test = loss_fn(y_pred, y_test)

        # update statistics
        steps_iter.set_description('loss=%g' % loss.item())
        stats.add_scalar("train-loss", loss.item())
        stats.add_scalar("test-loss", loss_test.item())

        stats.add_saturations()
    steps_iter.write('\n')
    stats.close()
    steps_iter.close()
```

Total running time of the script: (0 minutes 0.000 seconds)

5.1.4 Academic Gallery

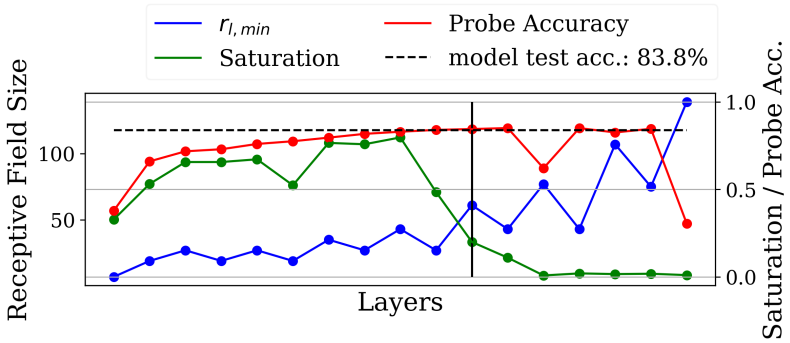
Delve has been used in several papers:

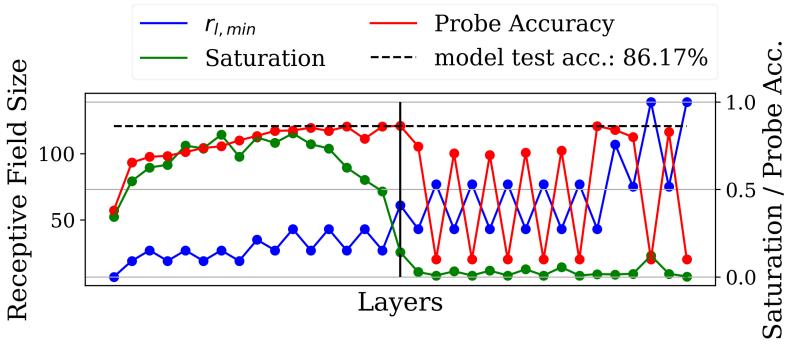
ResNet18 trained on Cifar10 for 30 epochs using the adam optimizer and a batch size of 64. Image from "Should You Go Deeper? Optimizing Convolutional Neural Networks without training".

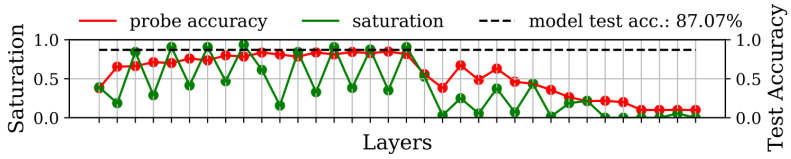
ResNet34 trained on Cifar10 for 30 epochs using the adam optimizer. Image from "Should You Go Deeper? Optimizing Convolutional Neural Networks without training".

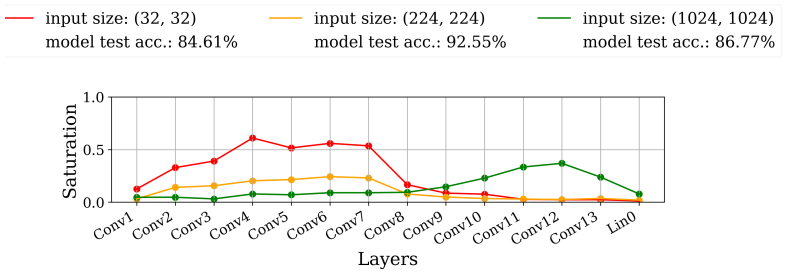
DenseNet18 trained on Food101 for 90 epochs using the stochastic gradient decent optimizer and a batch size of 128. Image from "Feature Space Saturation During Training".

VGG16 trained on 3 different resolutions for 30 epochs using the Adam-optimizer and a batch size of 32. You can see the shift in the inference process by observing the shift in high saturation values. Image from "(Input) Size Matters for Convolutional Neural Network Classifiers".


$r_{l,min}$
Saturation
Probe Accuracy
model test acc.: 83.8%
Receptive Field Size
100
50
Layers
1.0
0.5
0.0
Saturation / Probe Acc.


$r_{l,min}$
Saturation
Probe Accuracy
model test acc.: 86.17%
Receptive Field Size
100
50
Layers
1.0
0.5
0.0
Saturation / Probe Acc.


probe accuracy
saturation
model test acc.: 87.07%
Saturation
1.0
0.5
0.0
Layers
Test Accuracy

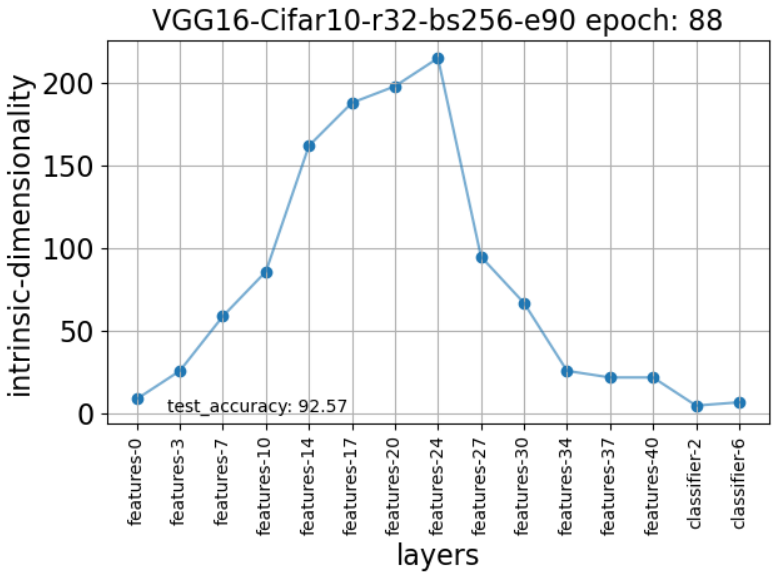

input size: (32, 32)
model test acc.: 84.61%
input size: (224, 224)
model test acc.: 92.55%
input size: (1024, 1024)
model test acc.: 86.77%
Saturation
1.0
0.5
0.0
Conv1
Conv2
Conv3
Conv4
Conv5
Conv6
Conv7
Conv8
Conv9
Conv10
Conv11
Conv12
Conv13
Lin0
Layers

5.1.5 Examples

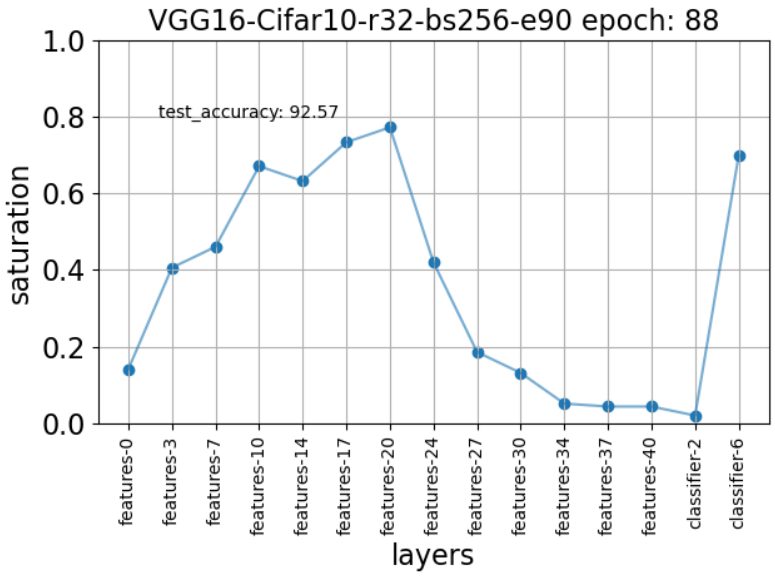
Delve allows the user to create plots and log records in various formats.

Plotting

Delve allows plotting results every epoch using `save_to="csvplot"`, which will create automated plots from the metrics recorded in the `stats` argument. The plots depict the layers generally in order of the forward pass.



Automatically generated plot of intrinsic dimensionality computed on the training set of Cifar10 on VGG16 at the 88th epoch of a 90 epoch of training.



Automatically generated plot of saturation computed on the training set of Cifar10 on VGG16 at the 88th epoch of a 90 epoch training.

Logging

Delve logs results with the `logging` package and shows progress with `tqdm`.

A simple example generated from a two-layer network trained on randomly generated data is provided in sphx_glr_gallery.

5.1.6 Reference

SaturationTracker

`SaturationTracker` provides a hook for PyTorch and extracts metrics during model training.

class delve.**SaturationTracker**(*savefile: str, save_to: ~typing.Union[str, ~delve.writers.AbstractWriter], modules: ~torch.nn.modules.module.Module, layer_filter: ~typing.Callable[[~typing.Dict[str, ~torch.nn.modules.module.Module]], ~typing.Dict[str, ~torch.nn.modules.module.Module]] = <function SaturationTracker.<lambda>>, writer_args: ~typing.Optional[~typing.Dict[str, ~typing.Any]] = None, log_interval=1, max_samples=None, stats: list = ['lsat'], layerwise_sat: bool = True, reset_covariance: bool = True, average_sat: bool = False, ignore_layer_names: ~typing.List[str] = [], include_conv: bool = True, conv_method: str = 'channelwise', timeseries_method: str = 'last_timestep', sat_threshold: str = 0.99, nosave=False, verbose: bool = False, device='cuda:0', initial_epoch: int = 0, interpolation_strategy: ~typing.Optional[str] = None, interpolation_downsampling: int = 32*)

Takes PyTorch module and records layer saturation,
intrinsic dimensionality and other scalars.

Parameters

- **savefile** (*str*) – destination for summaries
- **(str** (*save_to*) –

Specify one or multiple save strategies.
You can use preimplemented save strategies or inherit from the AbstractWriter in order to implement your own preferred saving strategy.

pre-existing saving strategies are:

csv
[stores all stats in a csv-file with one] row for each epoch.

plot
[produces plots from intrinsic dimensionality] and / or layer saturation

tensorboard : saves all stats to tensorboard print : print all metrics on console

as soon as they are logged

npy
[creates a folder-structure with npy-files] containing the logged values. This is the only save strategy that can save the full covariance matrix. This strategy is useful if you want to reproduce intrinsic dimensionality and saturation values with other thresholds without re-evaluating model checkpoints.

- `List[Union[str` –

Specify one or multiple save strategies.
You can use preimplemented save strategies or inherit from the AbstractWriter in order to implement your own preferred saving strategy.

pre-existing saving strategies are:

csv
[stores all stats in a csv-file with one] row for each epoch.

plot
[produces plots from intrinsic dimensionality] and / or layer saturation

tensorboard : saves all stats to tensorboard print : print all metrics on console

as soon as they are logged

npy
[creates a folder-structure with npy-files] containing the logged values. This is the only save strategy that can save the full covariance matrix. This strategy is useful if you want to reproduce intrinsic dimensionality and saturation values with other thresholds without re-evaluating model checkpoints.

- `delve.writers.AbstractWriter]]` –

Specify one or multiple save strategies.
You can use preimplemented save strategies or inherit from the AbstractWriter in order to implement your own preferred saving strategy.

pre-existing saving strategies are:

csv
[stores all stats in a csv-file with one] row for each epoch.

plot
[produces plots from intrinsic dimensionality] and / or layer saturation

tensorboard : saves all stats to tensorboard print : print all metrics on console

as soon as they are logged

npy
[creates a folder-structure with npy-files] containing the logged values. This is the only save strategy that can save the full covariance matrix. This strategy is useful if you want to reproduce intrinsic dimensionality and saturation values with other thresholds without re-evaluating model checkpoints.

- **modules** (*torch modules or list of modules*) – layer-containing object. Per default, only Conv2D, Linear and LSTM-Cells are recorded
- **layer_filter** (*func*) – A filter function that is used to avoid layers from being tracked. This is function receiving a dictionary as input and returning it with undesired entries

removed. Default: Identity function. The dictionary contains string keys mapping to torch.nn.Module objects.

- **writers_args** (*dict*) – contains additional arguments passed over to the writers. This is only used, when a writer is initialized through a string-key.
- **log_interval** (*int*) – distances between two batches used for updating the covariance matrix. Default value is 1, which means that all data is used for computing intrinsic dimensionality and saturation. Increasing the log interval is usefull on very large datasets to reduce numeric instability.
- **max_samples** (*int*) – (optional) the covariance matrix in each layer will halt updating itself when max_samples are reached. Usecase is similar to log-interval, when datasets are very large.
- **stats** (*list of str*) – list of stats to compute

  supported stats are:
  idim : intrinsic dimensionality lsat : layer saturation (intrinsic dimensionality divided by feature space dimensionality) cov : the covariance-matrix (only saveable using the 'npy' save strategy) det : the determinant of the covariance matrix (also known as generalized variance) trc : the trace of the covariance matrix, generally a more useful metric than det for determining

  the total variance of the data than the determinant. However note that this does not take the correlation between features into account. On the other hand, in most cases the determinent will be zero, since there will be very strongly correlated features, so trace might be the better option.

  dtrc : the trace of the diagonalmatrix, another way of measuring the dispersion of the data. lsat : layer saturation (intrinsic dimensionality

  divided by feature space dimensionality)

  embed : samples embedded in the eigenspace of dimension 2
- **layerwise_sat** (*bool*) – whether or not to include layerwise saturation when saving
- **reset_covariance** (*bool*) – True by default, resets the covariance every time the stats are computed. Disabling this option will strongly bias covariance since the gradient will influence the model. We recommend computing saturation at the end of training and testing.
- **include_conv** – setting to False includes only linear layers
- **conv_method** (*str*) –

  how to subsample convolutional layers. Default is
  channelwise, which means that the each position of the filter tensor is considered a datapoint, effectivly yielding a data matrix of shape (height*width*batch_size, num_filters)

  supported methods are:

  channelwise
  [treats every depth vector of the tensor as a] datapoint, effectivly reshaping the data tensor from shape (batch_size, height, width, channel) into (batch_size*height*width, channel).

  mean
  [applies global average pooling on] each feature map

  max
  [applies global max pooling on] each feature map

median
[applies global median pooling on] each feature map

flatten
[flattenes the entire feature map to a vector,] reshaping the data tensor into a data matrix of shape (batch_size, height*width*channel). This strategy for dealing with convolutions is extremly memory intensive and will likely cause memory and performance problems for any non toy-problem

- **timeseries_method** (*str*) –

  how to subsample timeseries methods. Default
  is last_timestep.

  supported methods are:
  timestepwise : stacks each sample timestep-by-timestep last_timestep : selects the last timestep's output

- **nosave** (*bool*) – If True, disables saving artifacts (images), default is False
- **verbose** (*bool*) – print saturation for every layer during training
- **sat_threshold** (*float*) – threshold used to determine the number of eigendirections belonging to the latent space. In effect, this is the threshold determining the the intrinsic dimensionality. Default value is 0.99 (99% of the explained variance), which is a compromise between a good and interpretable approximation. From experience the threshold should be between 0.97 and 0.9995 for meaningfull results.
- **verbose** – Change verbosity level (default is 0)
- **device** (*str*) – Device to do the computations on. Default is cuda:0. Generally it is recommended to do the computations on the gpu in order to get maximum performance. Using the cpu is generally slower but it lets delve use regular RAM instead of the generally more limited VRAM of the GPU. Not having delve run on the same device as the network causes slight performance decrease due to copying memory between devices during each forward pass. Delve can handle models distributed on multiple GPUs, however delve itself will always run on a single device.
- **initial_epoch** (*int*) – The initial epoch to start with. Default is 0, which corresponds to a new run. If initial_epoch != 0 the writers will look for save states that they can resume. If set to zero, all existing states will be overwritten. If set to a lower epoch than actually recorded the behavior of the writers is undefined and may result in crashes, loss of data or corrupted data.
- **interpolation_strategy** (*str*) – Default is None (disabled). If set to a string key accepted by the model-argument of torch.nn.functional.interpolate, the feature map will be resized to match the interpolated size. This is useful if you work with large resolutions and want to save up on computation time. is done if the resolution is smaller.
- **interpolation_downsampling** (*int*) – Default is 32. The target resolution if downsampling is enabled.

API Pages

SaturationTracker(savefile, save_to, ...[, ...])	Takes PyTorch module and records layer saturation,

delve.SaturationTracker

class delve.**SaturationTracker**(*savefile: str, save_to: ~typing.Union[str, ~delve.writers.AbstractWriter], modules: ~torch.nn.modules.module.Module, layer_filter: ~typing.Callable[[~typing.Dict[str, ~torch.nn.modules.module.Module]], ~typing.Dict[str, ~torch.nn.modules.module.Module]] = <function SaturationTracker.<lambda>>, writer_args: ~typing.Optional[~typing.Dict[str, ~typing.Any]] = None, log_interval=1, max_samples=None, stats: list = ['lsat'], layerwise_sat: bool = True, reset_covariance: bool = True, average_sat: bool = False, ignore_layer_names: ~typing.List[str] = [], include_conv: bool = True, conv_method: str = 'channelwise', timeseries_method: str = 'last_timestep', sat_threshold: str = 0.99, nosave=False, verbose: bool = False, device='cuda:0', initial_epoch: int = 0, interpolation_strategy: ~typing.Optional[str] = None, interpolation_downsampling: int = 32*)

Takes PyTorch module and records layer saturation,
intrinsic dimensionality and other scalars.

Parameters

- **savefile** (*str*) – destination for summaries
- **(str** (*save_to*) –

  Specify one or multiple save strategies.
  You can use preimplemented save strategies or inherit from the AbstractWriter in order to implement your own preferred saving strategy.

  pre-existing saving strategies are:

  csv
  [stores all stats in a csv-file with one] row for each epoch.

  plot
  [produces plots from intrinsic dimensionality] and / or layer saturation

  tensorboard : saves all stats to tensorboard print : print all metrics on console

  as soon as they are logged

  npy
  [creates a folder-structure with npy-files] containing the logged values. This is the only save strategy that can save the full covariance matrix. This strategy is useful if you want to reproduce intrinsic dimensionality and saturation values with other thresholds without re-evaluating model checkpoints.

- **List[Union[str** –

  Specify one or multiple save strategies.
  You can use preimplemented save strategies or inherit from the AbstractWriter in order to implement your own preferred saving strategy.

pre-existing saving strategies are:

csv
[stores all stats in a csv-file with one] row for each epoch.

plot
[produces plots from intrinsic dimensionality] and / or layer saturation

tensorboard : saves all stats to tensorboard print : print all metrics on console

as soon as they are logged

npy
[creates a folder-structure with npy-files] containing the logged values. This is the only save strategy that can save the full covariance matrix. This strategy is useful if you want to reproduce intrinsic dimensionality and saturation values with other thresholds without re-evaluating model checkpoints.

- **delve.writers.AbstractWriter]]** –

Specify one or multiple save strategies.
You can use preimplemented save strategies or inherit from the AbstractWriter in order to implement your own preferred saving strategy.

pre-existing saving strategies are:

csv
[stores all stats in a csv-file with one] row for each epoch.

plot
[produces plots from intrinsic dimensionality] and / or layer saturation

tensorboard : saves all stats to tensorboard print : print all metrics on console

as soon as they are logged

npy
[creates a folder-structure with npy-files] containing the logged values. This is the only save strategy that can save the full covariance matrix. This strategy is useful if you want to reproduce intrinsic dimensionality and saturation values with other thresholds without re-evaluating model checkpoints.

- **modules** (*torch modules or list of modules*) – layer-containing object. Per default, only Conv2D, Linear and LSTM-Cells are recorded
- **layer_filter** (*func*) – A filter function that is used to avoid layers from being tracked. This is function receiving a dictionary as input and returning it with undesired entries removed. Default: Identity function. The dictionary contains string keys mapping to torch.nn.Module objects.
- **writers_args** (*dict*) – contains additional arguments passed over to the writers. This is only used, when a writer is initialized through a string-key.
- **log_interval** (*int*) – distances between two batches used for updating the covariance matrix. Default value is 1, which means that all data is used for computing intrinsic dimensionality and saturation. Increasing the log interval is usefull on very large datasets to reduce numeric instability.
- **max_samples** (*int*) – (optional) the covariance matrix in each layer will halt updating itself when max_samples are reached. Usecase is similar to log-interval, when datasets are very large.

- **stats** (*list of str*) – list of stats to compute

  supported stats are:
  idim : intrinsic dimensionality lsat : layer saturation (intrinsic dimensionality divided by feature space dimensionality) cov : the covariance-matrix (only saveable using the 'npy' save strategy) det : the determinant of the covariance matrix (also known as generalized variance) trc : the trace of the covariance matrix, generally a more useful metric than det for determining

  the total variance of the data than the determinant. However note that this does not take the correlation between features into account. On the other hand, in most cases the determinent will be zero, since there will be very strongly correlated features, so trace might be the better option.

  dtrc : the trace of the diagonalmatrix, another way of measuring the dispersion of the data. lsat : layer saturation (intrinsic dimensionality

  divided by feature space dimensionality)

  embed : samples embedded in the eigenspace of dimension 2

- **layerwise_sat** (*bool*) – whether or not to include layerwise saturation when saving
- **reset_covariance** (*bool*) – True by default, resets the covariance every time the stats are computed. Disabling this option will strongly bias covariance since the gradient will influence the model. We recommend computing saturation at the end of training and testing.
- **include_conv** – setting to False includes only linear layers
- **conv_method** (*str*) –

  how to subsample convolutional layers. Default is
  channelwise, which means that the each position of the filter tensor is considered a datapoint, effectivly yielding a data matrix of shape (height*width*batch_size, num_filters)

  supported methods are:

  channelwise
  [treats every depth vector of the tensor as a] datapoint, effectivly reshaping the data tensor from shape (batch_size, height, width, channel) into (batch_size*height*width, channel).

  mean
  [applies global average pooling on] each feature map

  max
  [applies global max pooling on] each feature map

  median
  [applies global median pooling on] each feature map

  flatten
  [flattenes the entire feature map to a vector,] reshaping the data tensor into a data matrix of shape (batch_size, height*width*channel). This strategy for dealing with convolutions is extremly memory intensive and will likely cause memory and performance problems for any non toy-problem

- **timeseries_method** (*str*) –

  how to subsample timeseries methods. Default
  is last_timestep.

supported methods are:
timestepwise : stacks each sample timestep-by-timestep last_timestep : selects the last timestep's output

- **nosave** (*bool*) – If True, disables saving artifacts (images), default is False
- **verbose** (*bool*) – print saturation for every layer during training
- **sat_threshold** (*float*) – threshold used to determine the number of eigendirections belonging to the latent space. In effect, this is the threshold determining the the intrinsic dimensionality. Default value is 0.99 (99% of the explained variance), which is a compromise between a good and interpretable approximation. From experience the threshold should be between 0.97 and 0.9995 for meaningfull results.
- **verbose** – Change verbosity level (default is 0)
- **device** (*str*) – Device to do the computations on. Default is cuda:0. Generally it is recommended to do the computations on the gpu in order to get maximum performance. Using the cpu is generally slower but it lets delve use regular RAM instead of the generally more limited VRAM of the GPU. Not having delve run on the same device as the network causes slight performance decrease due to copying memory between devices during each forward pass. Delve can handle models distributed on multiple GPUs, however delve itself will always run on a single device.
- **initial_epoch** (*int*) – The initial epoch to start with. Default is 0, which corresponds to a new run. If initial_epoch != 0 the writers will look for save states that they can resume. If set to zero, all existing states will be overwritten. If set to a lower epoch than actually recorded the behavior of the writers is undefined and may result in crashes, loss of data or corrupted data.
- **interpolation_strategy** (*str*) – Default is None (disabled). If set to a string key accepted by the model-argument of torch.nn.functional.interpolate, the feature map will be resized to match the interpolated size. This is useful if you work with large resolutions and want to save up on computation time. is done if the resolution is smaller.
- **interpolation_downsampling** (*int*) – Default is 32. The target resolution if downsampling is enabled.

5.1.7 Support for Delve

Bugs

Bugs, issues and improvement requests can be logged in Github Issues.

Community

Community support is provided via Gitter. Just ask a question there.

5.1.8 Contributing to Delve

(Contribution guidelines largely copied from geopandas)

Overview

Contributions to Delve are very welcome. They are likely to be accepted more quickly if they follow these guidelines.

At this stage of Delve development, the priorities are to define a simple, usable, and stable API and to have clean, maintainable, readable code. Performance matters, but not at the expense of those goals.

In general, Delve follows the conventions of the pandas project where applicable.

In particular, when submitting a pull request:

- All existing tests should pass. Please make sure that the test suite passes, both locally and on GitHub Actions. Status on GitHub Actions will be visible on a pull request.
- New functionality should include tests. Please write reasonable tests for your code and make sure that they pass on your pull request.
- Classes, methods, functions, etc. should have docstrings. The first line of a docstring should be a standalone summary. Parameters and return values should be ducumented explicitly.
- Delve supports python 3 (3.6+). Use modern python idioms when possible.
- Follow PEP 8 when possible.
- Imports should be grouped with standard library imports first, 3rd-party libraries next, and Delve imports third. Within each grouping, imports should be alphabetized. Always use absolute imports when possible, and explicit relative imports for local imports when necessary in tests.

Seven Steps for Contributing

There are seven basic steps to contributing to *Delve*:

1) Fork the *Delve* git repository
2) Create a development environment
3) Install *Delve* dependencies
4) Make a `development` build of *Delve*
5) Make changes to code and add tests
6) Update the documentation
7) Submit a Pull Request

Each of these 7 steps is detailed below.

1) Forking the *Delve* repository using Git

To the new user, working with Git is one of the more daunting aspects of contributing to *Delve*. It can very quickly become overwhelming, but sticking to the guidelines below will help keep the process straightforward and mostly trouble free. As always, if you are having difficulties please feel free to ask for help.

The code is hosted on GitHub. To contribute you will need to sign up for a free GitHub account. We use Git for version control to allow many people to work together on the project.

Some great resources for learning Git:

- Software Carpentry's Git Tutorial
- Atlassian
- the GitHub help pages.
- Matthew Brett's Pydagogue.

Getting started with Git

GitHub has instructions for installing git, setting up your SSH key, and configuring git. All these steps need to be completed before you can work seamlessly between your local repository and GitHub.

Forking

You will need your own fork to work on the code. Go to the Delve project page and hit the `Fork` button. You will want to clone your fork to your machine:

```
git clone git@github.com:your-user-name/delve.git delve-yourname
cd delve-yourname
git remote add upstream git://github.com/delve-team/delve.git
```

This creates the directory *delve-yourname* and connects your repository to the upstream (main project) *Delve* repository.

The testing suite will run automatically on Travis-CI once your pull request is submitted. However, if you wish to run the test suite on a branch prior to submitting the pull request, then Travis-CI needs to be hooked up to your GitHub repository. Instructions for doing so are here.

Creating a branch

You want your master branch to reflect only production-ready code, so create a feature branch for making your changes. For example:

```
git branch shiny-new-feature
git checkout shiny-new-feature
```

The above can be simplified to:

```
git checkout -b shiny-new-feature
```

This changes your working directory to the shiny-new-feature branch. Keep any changes in this branch specific to one bug or feature so it is clear what the branch brings to *delve*. You can have many shiny-new-features and switch in between them using the git checkout command.

To update this branch, you need to retrieve the changes from the master branch:

```
git fetch upstream
git rebase upstream/master
```

This will replay your commits on top of the latest Delve git master. If this leads to merge conflicts, you must resolve these before submitting your pull request. If you have uncommitted changes, you will need to `stash` them prior to updating. This will effectively store your changes and they can be reapplied after updating.

2) Creating a development environment

A development environment is a virtual space where you can keep an independent installation of *Delve*. This makes it easy to keep both a stable version of python in one place you use for work, and a development version (which you may break while playing with code) in another.

An easy way to create a *Delve* development environment is as follows:

- Install either Anaconda or miniconda
- Make sure that you have *cloned the repository*
- `cd` to the *delve* source directory

Tell conda to create a new environment, named `delve_dev`, or any other name you would like for this environment, by running:

```
conda create -n delve_dev
```

For a python 3 environment:

```
conda create -n delve_dev python=3.8
```

This will create the new environment, and not touch any of your existing environments, nor any existing python installation.

To work in this environment, Windows users should `activate` it as follows:

```
activate delve_dev
```

Mac OSX and Linux users should use:

```
source activate delve_dev
```

You will then see a confirmation message to indicate you are in the new development environment.

To view your environments:

```
conda info -e
```

To return to you home root environment:

```
deactivate
```

See the full conda docs here.

At this point you can easily do a *development* install, as detailed in the next sections.

3) Installing Dependencies

To run *Delve* in an development environment, you must first install *Delve*'s dependencies. We suggest doing so using the following commands (executed after your development environment has been activated):

```
pip install -r requirements/reqirements.txt
```

This should install all necessary dependencies.

Next activate pre-commit hooks by running:

```
pre-commit install
```

4) Making a development build

Once dependencies are in place, make an in-place build by navigating to the git clone of the *delve* repository and running:

```
python setup.py develop
```

5) Making changes and writing tests

Delve is serious about testing and strongly encourages contributors to embrace test-driven development (TDD). This development process "relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test." So, before actually writing any code, you should write your tests. Often the test can be taken from the original GitHub issue. However, it is always worth considering additional use cases and writing corresponding tests.

Adding tests is one of the most common requests after code is pushed to *delve*. Therefore, it is worth getting in the habit of writing tests ahead of time so this is never an issue.

delve uses the pytest testing system and the convenient extensions in numpy.testing.

Writing tests

All tests should go into the `tests` directory. This folder contains many current examples of tests, and we suggest looking to these for inspiration.

Running the test suite

The tests can then be run directly inside your Git clone (without having to install *Delve*) by typing:

```
pytest
```

6) Updating the Documentation

Delve documentation resides in the *doc* folder. Changes to the docs are make by modifying the appropriate file in the *source* folder within *doc*. *Delve* docs us reStructuredText syntax, which is explained here and the docstrings follow the Numpy Docstring standard.

Once you have made your changes, you can build the docs by navigating to the *doc* folder and typing:

```
make html
```

The resulting html pages will be located in *doc/build/html*.

7) Submitting a Pull Request

Once you've made changes and pushed them to your forked repository, you then submit a pull request to have them integrated into the *Delve* code base.

You can find a pull request (or PR) tutorial in the GitHub's Help Docs.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

S