Viper Core

Georg Wallmann, Sophia Mädler, Niklas Schmacke

Oct 13, 2021

CONTENTS:

Table of Contents

Welcome to the show

CHAPTER

ONE

COMMAND LINE ARGUMENTS

1.1 viper-split

Manipulate existing single cell hdf5 datasets.

```
usage: viper-split [-h] [-o OUTPUT OUTPUT] [-r] [-t THREADS] [-c] input_dataset
```

1.1.1 Positional Arguments

input_dataset input dataset which should be split

1.1.2 Named Arguments

-o, --output Output definition <name> <length>. For example -o test.h5 0.9 or or -o test.h5 1000. If the sum of all lengths is <= 1, it is interpretated as fraction. Else its used as absolute value

-r, --random shuffle single cells randomly

Default: False

-t, --threads number of threads

Default: 4

-c, --compression use lzf compression

Default: False

Manipulate existing single cell hdf5 datasets can be used for splitting, shuffleing and compression / decompression

Splitting with shuffle and compression:

```
viper-split single_cells.h5 -r -c -o train.h5 0.9 -o test.h5 0.05 -o validate.h5 0.05
```

Shuffle: viper-split single_cells.h5 -r -o single_cells.h5 1.0

Compression: viper-split single_cells.h5 -c -o single_cells.h5 1.0

Decompression: viper-split single_cells.h5 -o single_cells.h5 1.0

1.2 viper-stat

Scan directory for viper projects.

```
usage: viper-stat [-h] [-t THREADS] [-r RECURSION] [search_directory]
```

1.2.1 Positional Arguments

search_directory directory containing viper projects

1.2.2 Named Arguments

-t, --threads number of threads

Default: 8

-r, --recursion levels of recursion

Default: 5

CHAPTER

TWO

ML

2.1 datasets

class vipercore.ml.datasets.**HDF5SingleCellDataset**(**args: Any*, ***kwargs: Any*)

class vipercore.ml.datasets.**NPYSingleCellDataset**(**args: Any*, ***kwargs: Any*)

Summary line.

Extended description of function.

Parameters

- **arg1** (*int*) – Description of arg1
- **arg2** (*str*) – Description of arg2

Returns Description of return value

Return type bool

2.2 metrics

vipercore.ml.metrics.**auc**(*predictions*, *labels*)

area under curve of the receiver operator characteristic

vipercore.ml.metrics.**precision**(*predictions*, *labels*, *pos_label=0*)

precision for predictiong class pos_label

vipercore.ml.metrics.**precision_top_n**(*predictions*, *labels*, *pos_label=0*, *top_n=0.01*)

precision for the top_n percentage (0.01 = 1%) predictions

vipercore.ml.metrics.**recall**(*predictions*, *labels*, *pos_label=0*)

recall for predictiong class pos_label

vipercore.ml.metrics.**recall_top_n**(*predictions*, *labels*, *pos_label=0*, *top_n=0.01*)

recall for the top_n percentage (0.01 = 1%) predictions

2.3 models

class vipercore.ml.models.**GolgiCAE**(*cfg='B'*, *in_channels=5*, *out_channels=5*)

forward(*x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class vipercore.ml.models.**GolgiVAE**(*in_channels*, *out_channels*, *latent_dim*, *hidden_dims=None*, ***kwargs*)

decode(*z*)
Maps the given latent codes onto the image space. :param z: (Tensor) [B x D] :return: (Tensor) [B x C x H x W]

encode(*input*)
Encodes the input by passing through the encoder network and returns the latent codes. :param input: (Tensor) Input tensor to encoder [N x C x H x W] :return: (Tensor) List of latent codes

forward(*input*, ***kwargs*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

loss_function(*target*, *output*, **args*, ***kwargs*)

Computes the VAE loss function. KL(N(mu, sigma), N(0, 1)) = log

rac{1}{sigma} + rac{sigma^2 + mu^2}{2} - rac{1}{2}

param args

param kwargs

return

reparameterize(*mu*, *logvar*)
Reparameterization trick to sample from N(mu, var) from N(0,1). :param mu: (Tensor) Mean of the latent Gaussian [B x D] :param logvar: (Tensor) Standard deviation of the latent Gaussian [B x D] :return: (Tensor) [B x D]

class vipercore.ml.models.**GolgiVGG**(*cfg='B'*, *dimensions=196*, *in_channels=5*, *num_classes=2*)

forward(*x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

2.4 plmodels

class vipercore.ml.plmodels.**AEModel**(*model*, *hparams*)

configure_optimizers()

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple.

Returns

Any of these 6 options.

- **Single optimizer**.
- **List or Tuple** of optimizers.
- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple lr_dict).
- **Dictionary**, with an `"optimizer"` key, and (optionally) a `"lr_scheduler"` key whose value is a single LR scheduler or lr_dict.
- **Tuple of dictionaries** as described above, with an optional `"frequency"` key.
- **None** - Fit will run without any optimizer.

Note: The lr_dict is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_dict = {
    'scheduler': lr_scheduler, # The LR scheduler instance (required)
    # The unit of the scheduler's step size, could also be 'step'
    'interval': 'epoch',
    'frequency': 1, # The frequency of the scheduler
    'monitor': 'val_loss', # Metric for `ReduceLROnPlateau` to monitor
    'strict': True, # Whether to crash the training if `monitor` is not found
    'name': None, # Custom name for `LearningRateMonitor` to use
}
```

Only the `"scheduler"` key is required, the rest will be set to the defaults above.

Note: The `frequency` value specified in a dict along with the `optimizer` key is an int corresponding to the number of sequential batches optimized with the specific optimizer. It should be given to none or to all of the optimizers. There is a difference between passing multiple optimizers in a list, and passing multiple optimizers in dictionaries with a frequency of 1: In the former case, all optimizers will operate on the given batch in each optimization step. In the latter, only one optimizer will operate on the given batch at every step. This is different from the `frequency` value specified in the lr_dict mentioned below.

```
def configure_optimizers(self):
    optimizer_one = torch.optim.SGD(self.model.parameters(), lr=0.01)
    optimizer_two = torch.optim.SGD(self.model.parameters(), lr=0.01)
    return [
        {'optimizer': optimizer_one, 'frequency': 5},
        {'optimizer': optimizer_two, 'frequency': 10},
    ]
```

In this example, the first optimizer will be used for the first 5 steps, the second optimizer for the next 10 steps and that cycle will continue. If an LR scheduler is specified for an optimizer using the `lr_scheduler` key in the above dict, the scheduler will only be updated when its optimizer is being used.

Examples:

```
# most cases
def configure_optimizers(self):
    return Adam(self.parameters(), lr=1e-3)

# multiple optimizer case (e.g.: GAN)
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    return gen_opt, dis_opt

# example with learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    dis_sch = CosineAnnealing(dis_opt, T_max=10)
    return [gen_opt, dis_opt], [dis_sch]

# example with step-based learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    gen_sch = {'scheduler': ExponentialLR(gen_opt, 0.99),
               'interval': 'step'}  # called after each training step
    dis_sch = CosineAnnealing(dis_opt, T_max=10) # called every epoch
    return [gen_opt, dis_opt], [gen_sch, dis_sch]

# example with optimizer frequencies
# see training procedure in `Improved Training of Wasserstein GANs`, Algorithm 1
# https://arxiv.org/abs/1704.00028
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    n_critic = 5
    return (
        {'optimizer': dis_opt, 'frequency': n_critic},
        {'optimizer': gen_opt, 'frequency': 1}
    )
```

Note: Some things to know:

- Lightning calls `.backward()` and `.step()` on each optimizer and learning rate scheduler as needed.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizers.
- If you use multiple optimizers, *training_step()* will have an additional `optimizer_idx` parameter.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.
- If you use multiple optimizers, gradients will be calculated only for the parameters of current optimizer at each training step.
- If you need to control how often those optimizers step or override the default `.step()` schedule, override the `optimizer_step()` hook.

forward(*x*)

Same as `torch.nn.Module.forward()`.

Parameters

- ***args** – Whatever you decide to pass into the forward method.
- ****kwargs** – Keyword arguments are also possible.

Returns Your model's output

on_train_epoch_end(*outputs*)

Called in the training loop at the very end of the epoch.

To access all batch outputs at the end of the epoch, either:

1. Implement *training_epoch_end* in the LightningModule OR
2. Cache data across steps on the attribute(s) of the *LightningModule* and access them in this hook

on_train_epoch_start()

Called in the training loop at the very beginning of the epoch.

on_validation_epoch_start()

Called in the validation loop at the very beginning of the epoch.

training_step(*batch*, *batch_idx*)

Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters

- **batch** (`Tensor` | (`Tensor`, ...) | [`Tensor`, ...]) – The output of your `DataLoader`. A tensor, tuple or list.
- **batch_idx** (*int*) – Integer displaying index of this batch
- **optimizer_idx** (*int*) – When using multiple optimizers, this argument will also be present.
- **hiddens** (`Tensor`) – Passed in if **:paramref:`~pytorch_lightning.core.lightning.LightningModule.truncated_** > 0.

Returns

Any of.

- `Tensor` - The loss tensor

- `dict` - A dictionary. Can include any keys, but must include the key `'loss'`
- `None` - Training will skip to the next batch

Note: Returning `None` is currently not supported for multi-GPU or TPU, or with 16-bit precision enabled.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

If you define multiple optimizers, this step will be called with an additional `optimizer_idx` parameter.

```
# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx, optimizer_idx):
    if optimizer_idx == 0:
        # do training_step with encoder
    if optimizer_idx == 1:
        # do training_step with decoder
```

If you add truncated back propagation through time you will also get an additional argument with the hidden states of the previous step.

```
# Truncated back-propagation through time
def training_step(self, batch, batch_idx, hiddens):
    # hiddens are the hidden states from the previous truncated backprop step
    ...
    out, hiddens = self.lstm(data, hiddens)
    ...
    return {'loss': loss, 'hiddens': hiddens}
```

Note: The loss value shown in the progress bar is smoothed (averaged) over the last values, so it differs from the actual loss returned in train/validation step.

class vipercore.ml.plmodels.**AutoEncoderModel**(***kwargs*)

configure_optimizers()

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple.

Returns

Any of these 6 options.

- **Single optimizer**.
- **List or Tuple** of optimizers.

- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple lr_dict).
- **Dictionary**, with an `"optimizer"` key, and (optionally) a `"lr_scheduler"` key whose value is a single LR scheduler or lr_dict.
- **Tuple of dictionaries** as described above, with an optional `"frequency"` key.
- **None** - Fit will run without any optimizer.

Note: The lr_dict is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_dict = {
    'scheduler': lr_scheduler, # The LR scheduler instance (required)
    # The unit of the scheduler's step size, could also be 'step'
    'interval': 'epoch',
    'frequency': 1, # The frequency of the scheduler
    'monitor': 'val_loss', # Metric for `ReduceLROnPlateau` to monitor
    'strict': True, # Whether to crash the training if `monitor` is not found
    'name': None, # Custom name for `LearningRateMonitor` to use
}
```

Only the `"scheduler"` key is required, the rest will be set to the defaults above.

Note: The `frequency` value specified in a dict along with the `optimizer` key is an int corresponding to the number of sequential batches optimized with the specific optimizer. It should be given to none or to all of the optimizers. There is a difference between passing multiple optimizers in a list, and passing multiple optimizers in dictionaries with a frequency of 1: In the former case, all optimizers will operate on the given batch in each optimization step. In the latter, only one optimizer will operate on the given batch at every step. This is different from the `frequency` value specified in the lr_dict mentioned below.

```
def configure_optimizers(self):
    optimizer_one = torch.optim.SGD(self.model.parameters(), lr=0.01)
    optimizer_two = torch.optim.SGD(self.model.parameters(), lr=0.01)
    return [
        {'optimizer': optimizer_one, 'frequency': 5},
        {'optimizer': optimizer_two, 'frequency': 10},
    ]
```

In this example, the first optimizer will be used for the first 5 steps, the second optimizer for the next 10 steps and that cycle will continue. If an LR scheduler is specified for an optimizer using the `lr_scheduler` key in the above dict, the scheduler will only be updated when its optimizer is being used.

Examples:

```
# most cases
def configure_optimizers(self):
    return Adam(self.parameters(), lr=1e-3)

# multiple optimizer case (e.g.: GAN)
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
```

(continues on next page)

(continued from previous page)

```
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    return gen_opt, dis_opt

# example with learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    dis_sch = CosineAnnealing(dis_opt, T_max=10)
    return [gen_opt, dis_opt], [dis_sch]

# example with step-based learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    gen_sch = {'scheduler': ExponentialLR(gen_opt, 0.99),
               'interval': 'step'}  # called after each training step
    dis_sch = CosineAnnealing(dis_opt, T_max=10) # called every epoch
    return [gen_opt, dis_opt], [gen_sch, dis_sch]

# example with optimizer frequencies
# see training procedure in `Improved Training of Wasserstein GANs`, Algorithm 1
# https://arxiv.org/abs/1704.00028
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    n_critic = 5
    return (
        {'optimizer': dis_opt, 'frequency': n_critic},
        {'optimizer': gen_opt, 'frequency': 1}
    )
```

Note: Some things to know:

- Lightning calls `.backward()` and `.step()` on each optimizer and learning rate scheduler as needed.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizers.
- If you use multiple optimizers, *training_step()* will have an additional `optimizer_idx` parameter.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.
- If you use multiple optimizers, gradients will be calculated only for the parameters of current optimizer at each training step.
- If you need to control how often those optimizers step or override the default `.step()` schedule, override the `optimizer_step()` hook.

---

forward(*x*)

Same as `torch.nn.Module.forward()`.

Parameters

- ***args** – Whatever you decide to pass into the forward method.
- ****kwargs** – Keyword arguments are also possible.

Returns Your model's output

on_train_epoch_end(*outputs*)
Called in the training loop at the very end of the epoch.

To access all batch outputs at the end of the epoch, either:

1. Implement *training_epoch_end* in the LightningModule OR
2. Cache data across steps on the attribute(s) of the *LightningModule* and access them in this hook

on_train_epoch_start()
Called in the training loop at the very beginning of the epoch.

on_validation_epoch_start()
Called in the validation loop at the very beginning of the epoch.

training_step(*batch*, *batch_idx*)
Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters

- **batch** (Tensor | (Tensor, ...) | [Tensor, ...]) – The output of your DataLoader. A tensor, tuple or list.
- **batch_idx** (*int*) – Integer displaying index of this batch
- **optimizer_idx** (*int*) – When using multiple optimizers, this argument will also be present.
- **hiddens** (Tensor) – Passed in if **:paramref:`~pytorch_lightning.core.lightning.LightningModule.truncated_** > 0.

Returns

Any of.

- Tensor - The loss tensor
- dict - A dictionary. Can include any keys, but must include the key 'loss'
- None - Training will skip to the next batch

Note: Returning None is currently not supported for multi-GPU or TPU, or with 16-bit precision enabled.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

If you define multiple optimizers, this step will be called with an additional optimizer_idx parameter.

```
# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx, optimizer_idx):
```

(continues on next page)

(continued from previous page)

```
    if optimizer_idx == 0:
        # do training_step with encoder
    if optimizer_idx == 1:
        # do training_step with decoder
```

If you add truncated back propagation through time you will also get an additional argument with the hidden states of the previous step.

```
# Truncated back-propagation through time
def training_step(self, batch, batch_idx, hiddens):
    # hiddens are the hidden states from the previous truncated backprop step
    ...
    out, hiddens = self.lstm(data, hiddens)
    ...
    return {'loss': loss, 'hiddens': hiddens}
```

Note: The loss value shown in the progress bar is smoothed (averaged) over the last values, so it differs from the actual loss returned in train/validation step.

---

validation_step(*batch*, *batch_idx*)

Operates on a single batch of data from the validation set. In this step you'd might generate examples or calculate anything of interest like accuracy.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    val_outs.append(out)
validation_epoch_end(val_outs)
```

Parameters

- **batch** (Tensor | (Tensor, ...) | [Tensor, ...]) – The output of your DataLoader. A tensor, tuple or list.
- **batch_idx** (*int*) – The index of this batch
- **dataloader_idx** (*int*) – The index of the dataloader that produced this batch (only if multiple val dataloaders used)

Returns

Any of.

- Any object or value
- None - Validation will skip to the next batch

```
# pseudocode of order
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    if defined('validation_step_end'):
```

(continues on next page)

(continued from previous page)

```
        out = validation_step_end(out)
    val_outs.append(out)
val_outs = validation_epoch_end(val_outs)
```

```
# if you have one val dataloader:
def validation_step(self, batch, batch_idx)

# if you have multiple val dataloaders:
def validation_step(self, batch, batch_idx, dataloader_idx)
```

Examples:

```
# CASE 1: A single validation dataset
def validation_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    val_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'val_loss': loss, 'val_acc': val_acc})
```

If you pass in multiple val dataloaders, *validation_step()* will have an additional argument.

```
# CASE 2: multiple validation dataloaders
def validation_step(self, batch, batch_idx, dataloader_idx):
    # dataloader_idx tells you which dataset this is.
```

Note: If you don't need to validate you don't need to implement this method.

Note: When the *validation_step()* is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of validation, the model goes back to training mode and gradients are enabled.

class vipercore.ml.plmodels.**GeneralModel**(*model*, *hparams*)

configure_optimizers()

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need

one. But in the case of GANs or similar you might have multiple.

Returns

Any of these 6 options.

- **Single optimizer**.
- **List or Tuple** of optimizers.
- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple lr_dict).
- **Dictionary**, with an `"optimizer"` key, and (optionally) a `"lr_scheduler"` key whose value is a single LR scheduler or lr_dict.
- **Tuple of dictionaries** as described above, with an optional `"frequency"` key.
- **None** - Fit will run without any optimizer.

Note: The lr_dict is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_dict = {
    'scheduler': lr_scheduler, # The LR scheduler instance (required)
    # The unit of the scheduler's step size, could also be 'step'
    'interval': 'epoch',
    'frequency': 1, # The frequency of the scheduler
    'monitor': 'val_loss', # Metric for `ReduceLROnPlateau` to monitor
    'strict': True, # Whether to crash the training if `monitor` is not found
    'name': None, # Custom name for `LearningRateMonitor` to use
}
```

Only the `"scheduler"` key is required, the rest will be set to the defaults above.

Note: The `frequency` value specified in a dict along with the `optimizer` key is an int corresponding to the number of sequential batches optimized with the specific optimizer. It should be given to none or to all of the optimizers. There is a difference between passing multiple optimizers in a list, and passing multiple optimizers in dictionaries with a frequency of 1: In the former case, all optimizers will operate on the given batch in each optimization step. In the latter, only one optimizer will operate on the given batch at every step. This is different from the `frequency` value specified in the lr_dict mentioned below.

```
def configure_optimizers(self):
    optimizer_one = torch.optim.SGD(self.model.parameters(), lr=0.01)
    optimizer_two = torch.optim.SGD(self.model.parameters(), lr=0.01)
    return [
        {'optimizer': optimizer_one, 'frequency': 5},
        {'optimizer': optimizer_two, 'frequency': 10},
    ]
```

In this example, the first optimizer will be used for the first 5 steps, the second optimizer for the next 10 steps and that cycle will continue. If an LR scheduler is specified for an optimizer using the `lr_scheduler` key in the above dict, the scheduler will only be updated when its optimizer is being used.

Examples:

```
# most cases
def configure_optimizers(self):
    return Adam(self.parameters(), lr=1e-3)

# multiple optimizer case (e.g.: GAN)
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    return gen_opt, dis_opt

# example with learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    dis_sch = CosineAnnealing(dis_opt, T_max=10)
    return [gen_opt, dis_opt], [dis_sch]

# example with step-based learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    gen_sch = {'scheduler': ExponentialLR(gen_opt, 0.99),
               'interval': 'step'}  # called after each training step
    dis_sch = CosineAnnealing(dis_opt, T_max=10) # called every epoch
    return [gen_opt, dis_opt], [gen_sch, dis_sch]

# example with optimizer frequencies
# see training procedure in `Improved Training of Wasserstein GANs`, Algorithm 1
# https://arxiv.org/abs/1704.00028
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    n_critic = 5
    return (
        {'optimizer': dis_opt, 'frequency': n_critic},
        {'optimizer': gen_opt, 'frequency': 1}
    )
```

Note: Some things to know:

- Lightning calls `.backward()` and `.step()` on each optimizer and learning rate scheduler as needed.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizers.
- If you use multiple optimizers, *training_step()* will have an additional `optimizer_idx` parameter.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.
- If you use multiple optimizers, gradients will be calculated only for the parameters of current optimizer at each training step.
- If you need to control how often those optimizers step or override the default `.step()` schedule, override the `optimizer_step()` hook.

forward(*x*)
Same as `torch.nn.Module.forward()`.

Parameters

- ***args** – Whatever you decide to pass into the forward method.
- ****kwargs** – Keyword arguments are also possible.

Returns Your model's output

on_train_epoch_end(*outputs*)
Called in the training loop at the very end of the epoch.

To access all batch outputs at the end of the epoch, either:

1. Implement *training_epoch_end* in the LightningModule OR
2. Cache data across steps on the attribute(s) of the *LightningModule* and access them in this hook

on_train_epoch_start()
Called in the training loop at the very beginning of the epoch.

on_train_start()
Called at the beginning of training after sanity check.

on_validation_epoch_end()
Called in the validation loop at the very end of the epoch.

on_validation_epoch_start()
Called in the validation loop at the very beginning of the epoch.

training_step(*batch*, *batch_idx*)
Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters

- **batch** (`Tensor` | (`Tensor`, ...) | [`Tensor`, ...]) – The output of your `DataLoader`. A tensor, tuple or list.
- **batch_idx** (*int*) – Integer displaying index of this batch
- **optimizer_idx** (*int*) – When using multiple optimizers, this argument will also be present.
- **hiddens** (`Tensor`) – Passed in if **:paramref:`~pytorch_lightning.core.lightning.LightningModule.truncated_** > 0.

Returns

Any of.

- `Tensor` - The loss tensor
- `dict` - A dictionary. Can include any keys, but must include the key `'loss'`
- `None` - Training will skip to the next batch

Note: Returning `None` is currently not supported for multi-GPU or TPU, or with 16-bit precision enabled.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

If you define multiple optimizers, this step will be called with an additional `optimizer_idx` parameter.

```
# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx, optimizer_idx):
    if optimizer_idx == 0:
        # do training_step with encoder
    if optimizer_idx == 1:
        # do training_step with decoder
```

If you add truncated back propagation through time you will also get an additional argument with the hidden states of the previous step.

```
# Truncated back-propagation through time
def training_step(self, batch, batch_idx, hiddens):
    # hiddens are the hidden states from the previous truncated backprop step
    ...
    out, hiddens = self.lstm(data, hiddens)
    ...
    return {'loss': loss, 'hiddens': hiddens}
```

Note: The loss value shown in the progress bar is smoothed (averaged) over the last values, so it differs from the actual loss returned in train/validation step.

---

validation_step(*batch*, *batch_idx*)

Operates on a single batch of data from the validation set. In this step you'd might generate examples or calculate anything of interest like accuracy.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    val_outs.append(out)
validation_epoch_end(val_outs)
```

Parameters

- **batch** (`Tensor` | (`Tensor`, ...) | [`Tensor`, ...]) – The output of your `DataLoader`. A tensor, tuple or list.
- **batch_idx** (*int*) – The index of this batch
- **dataloader_idx** (*int*) – The index of the dataloader that produced this batch (only if multiple val dataloaders used)

Returns

Any of.

- Any object or value

- None - Validation will skip to the next batch

```
# pseudocode of order
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    if defined('validation_step_end'):
        out = validation_step_end(out)
    val_outs.append(out)
val_outs = validation_epoch_end(val_outs)
```

```
# if you have one val dataloader:
def validation_step(self, batch, batch_idx)

# if you have multiple val dataloaders:
def validation_step(self, batch, batch_idx, dataloader_idx)
```

Examples:

```
# CASE 1: A single validation dataset
def validation_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    val_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'val_loss': loss, 'val_acc': val_acc})
```

If you pass in multiple val dataloaders, *validation_step()* will have an additional argument.

```
# CASE 2: multiple validation dataloaders
def validation_step(self, batch, batch_idx, dataloader_idx):
    # dataloader_idx tells you which dataset this is.
```

Note: If you don't need to validate you don't need to implement this method.

Note: When the *validation_step()* is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of validation, the model goes back to training mode and gradients are

enabled.

---

class vipercore.ml.plmodels.**MultilabelSupervisedModel**(***kwargs*)

configure_optimizers()

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple.

Returns

Any of these 6 options.

- **Single optimizer**.
- **List or Tuple** of optimizers.
- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple lr_dict).
- **Dictionary**, with an `"optimizer"` key, and (optionally) a `"lr_scheduler"` key whose value is a single LR scheduler or lr_dict.
- **Tuple of dictionaries** as described above, with an optional `"frequency"` key.
- **None** - Fit will run without any optimizer.

Note: The lr_dict is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_dict = {
    'scheduler': lr_scheduler, # The LR scheduler instance (required)
    # The unit of the scheduler's step size, could also be 'step'
    'interval': 'epoch',
    'frequency': 1, # The frequency of the scheduler
    'monitor': 'val_loss', # Metric for `ReduceLROnPlateau` to monitor
    'strict': True, # Whether to crash the training if `monitor` is not found
    'name': None, # Custom name for `LearningRateMonitor` to use
}
```

Only the `"scheduler"` key is required, the rest will be set to the defaults above.

Note: The `frequency` value specified in a dict along with the `optimizer` key is an int corresponding to the number of sequential batches optimized with the specific optimizer. It should be given to none or to all of the optimizers. There is a difference between passing multiple optimizers in a list, and passing multiple optimizers in dictionaries with a frequency of 1: In the former case, all optimizers will operate on the given batch in each optimization step. In the latter, only one optimizer will operate on the given batch at every step. This is different from the `frequency` value specified in the lr_dict mentioned below.

```
def configure_optimizers(self):
    optimizer_one = torch.optim.SGD(self.model.parameters(), lr=0.01)
    optimizer_two = torch.optim.SGD(self.model.parameters(), lr=0.01)
    return [
        {'optimizer': optimizer_one, 'frequency': 5},
        {'optimizer': optimizer_two, 'frequency': 10},
    ]
```

In this example, the first optimizer will be used for the first 5 steps, the second optimizer for the next 10 steps and that cycle will continue. If an LR scheduler is specified for an optimizer using the `lr_scheduler` key in the above dict, the scheduler will only be updated when its optimizer is being used.

Examples:

```
# most cases
def configure_optimizers(self):
    return Adam(self.parameters(), lr=1e-3)

# multiple optimizer case (e.g.: GAN)
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    return gen_opt, dis_opt

# example with learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    dis_sch = CosineAnnealing(dis_opt, T_max=10)
    return [gen_opt, dis_opt], [dis_sch]

# example with step-based learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    gen_sch = {'scheduler': ExponentialLR(gen_opt, 0.99),
               'interval': 'step'}  # called after each training step
    dis_sch = CosineAnnealing(dis_opt, T_max=10) # called every epoch
    return [gen_opt, dis_opt], [gen_sch, dis_sch]

# example with optimizer frequencies
# see training procedure in `Improved Training of Wasserstein GANs`, Algorithm 1
# https://arxiv.org/abs/1704.00028
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    n_critic = 5
    return (
        {'optimizer': dis_opt, 'frequency': n_critic},
        {'optimizer': gen_opt, 'frequency': 1}
    )
```

Note: Some things to know:

- Lightning calls `.backward()` and `.step()` on each optimizer and learning rate scheduler as needed.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizers.
- If you use multiple optimizers, *`training_step()`* will have an additional `optimizer_idx` parameter.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.
- If you use multiple optimizers, gradients will be calculated only for the parameters of current optimizer

at each training step.

- If you need to control how often those optimizers step or override the default `.step()` schedule, override the `optimizer_step()` hook.

---

forward(*x*)
Same as `torch.nn.Module.forward()`.

Parameters

- ***args** – Whatever you decide to pass into the forward method.
- ****kwargs** – Keyword arguments are also possible.

Returns Your model's output

on_train_epoch_end(*outputs*)
Called in the training loop at the very end of the epoch.

To access all batch outputs at the end of the epoch, either:

1. Implement *training_epoch_end* in the LightningModule OR
2. Cache data across steps on the attribute(s) of the *LightningModule* and access them in this hook

on_train_epoch_start()
Called in the training loop at the very beginning of the epoch.

on_train_start()
Called at the beginning of training after sanity check.

on_validation_epoch_end()
Called in the validation loop at the very end of the epoch.

on_validation_epoch_start()
Called in the validation loop at the very beginning of the epoch.

training_step(*batch*, *batch_idx*)
Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters

- **batch** (`Tensor` | (`Tensor`, ...) | [`Tensor`, ...]) – The output of your `DataLoader`. A tensor, tuple or list.
- **batch_idx** (*int*) – Integer displaying index of this batch
- **optimizer_idx** (*int*) – When using multiple optimizers, this argument will also be present.
- **hiddens** (`Tensor`) – Passed in if **:paramref:`~pytorch_lightning.core.lightning.LightningModule.truncated_** > 0.

Returns

Any of.

- `Tensor` - The loss tensor
- `dict` - A dictionary. Can include any keys, but must include the key `'loss'`
- `None` - Training will skip to the next batch

Note: Returning `None` is currently not supported for multi-GPU or TPU, or with 16-bit precision enabled.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

If you define multiple optimizers, this step will be called with an additional `optimizer_idx` parameter.

```
# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx, optimizer_idx):
    if optimizer_idx == 0:
        # do training_step with encoder
    if optimizer_idx == 1:
        # do training_step with decoder
```

If you add truncated back propagation through time you will also get an additional argument with the hidden states of the previous step.

```
# Truncated back-propagation through time
def training_step(self, batch, batch_idx, hiddens):
    # hiddens are the hidden states from the previous truncated backprop step
    ...
    out, hiddens = self.lstm(data, hiddens)
    ...
    return {'loss': loss, 'hiddens': hiddens}
```

Note: The loss value shown in the progress bar is smoothed (averaged) over the last values, so it differs from the actual loss returned in train/validation step.

validation_step(*batch*, *batch_idx*)

Operates on a single batch of data from the validation set. In this step you'd might generate examples or calculate anything of interest like accuracy.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    val_outs.append(out)
validation_epoch_end(val_outs)
```

Parameters

- **batch** (`Tensor` | (`Tensor`, ...) | [`Tensor`, ...]) – The output of your `DataLoader`. A tensor, tuple or list.
- **batch_idx** (*int*) – The index of this batch

- **dataloader_idx** (*int*) – The index of the dataloader that produced this batch (only if multiple val dataloaders used)

Returns

Any of.

- Any object or value
- None - Validation will skip to the next batch

```
# pseudocode of order
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    if defined('validation_step_end'):
        out = validation_step_end(out)
    val_outs.append(out)
val_outs = validation_epoch_end(val_outs)
```

```
# if you have one val dataloader:
def validation_step(self, batch, batch_idx)

# if you have multiple val dataloaders:
def validation_step(self, batch, batch_idx, dataloader_idx)
```

Examples:

```
# CASE 1: A single validation dataset
def validation_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    val_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'val_loss': loss, 'val_acc': val_acc})
```

If you pass in multiple val dataloaders, *validation_step()* will have an additional argument.

```
# CASE 2: multiple validation dataloaders
def validation_step(self, batch, batch_idx, dataloader_idx):
    # dataloader_idx tells you which dataset this is.
```

Note: If you don't need to validate you don't need to implement this method.

Note: When the *validation_step()* is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of validation, the model goes back to training mode and gradients are enabled.

PYTHON MODULE INDEX

V

INDEX

P

R

T

V