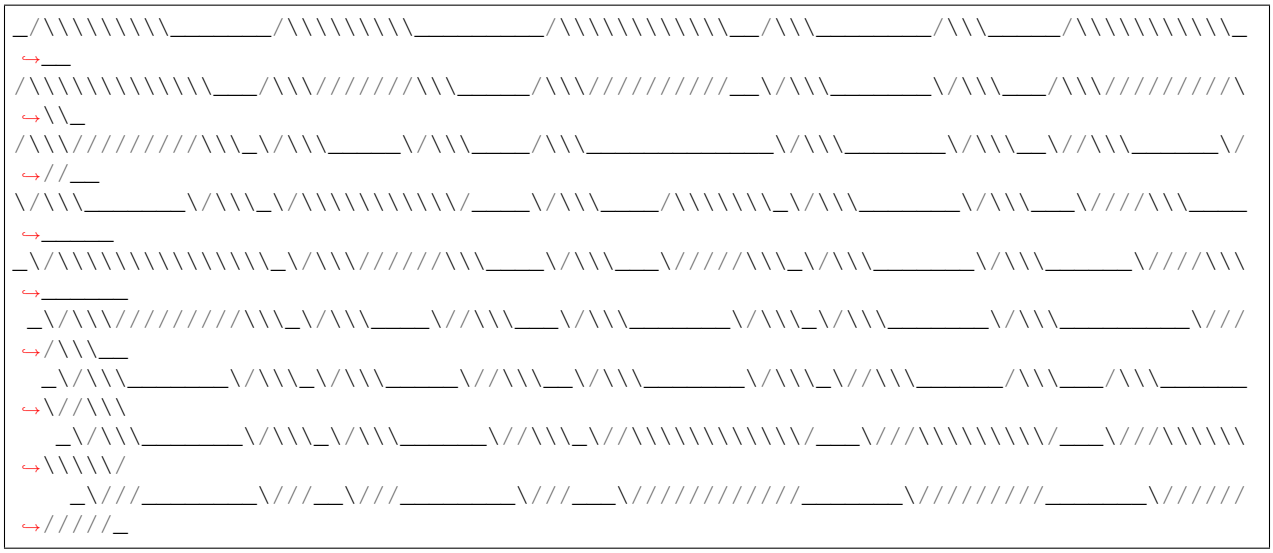
argus

Release 0.1.2

Ruslan Baikulov

Oct 17, 2020

NOTES



Argus is easy-to-use flexible library for training neural networks in PyTorch.

- GitHub repository: https://github.com/lRomul/argus
- License: MIT

CHAPTER

ONE

INSTALLATION

You can use pip to install argus:

```
pip install pytorch-argus
```

If you want to get the latest version of the code before it is released on PyPI you can install the library from GitHub:

```
pip install -U git+https://github.com/lRomul/argus.git
```

1.1 Quick start

Link to quick start jupyter notebook.

1.1.1 Simple example

Define a PyTorch model.

```
import torch
from torch import nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self, n_classes, p_dropout=0.5):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d(p=p_dropout)
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, n_classes)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return x
```

Define a *argus.model.Model* with `nn_module`, `optimizer`, `loss` attributes. Each value must be a class or function that returns object (`torch.nn.Module` for loss and nn_module, `torch.optim.Optimizer` for optimizer).

```
from argus import Model

class MnistModel(Model):
    nn_module = Net
    optimizer = torch.optim.SGD
    loss = torch.nn.CrossEntropyLoss
```

Create instance of `MnistModel` with specific parameters. Net will be initialized like `Net(n_classes=10, p_dropout=0.1)`. Same logic for optimizer `torch.optim.SGD(lr=0.01)`. Loss will be created without arguments `torch.nn.CrossEntropyLoss()`.

```
params = {
    'nn_module': {'n_classes': 10, 'p_dropout': 0.1},
    'optimizer': {'lr': 0.01},
    'device': 'cpu'
}

model = MnistModel(params)
```

Download MNIST dataset. Create validation and training PyTorch data loaders.

```
from torch.utils.data import DataLoader
from torchvision.transforms import Compose, ToTensor, Normalize
from torchvision.datasets import MNIST

data_transform = Compose([ToTensor(), Normalize((0.1307,), (0.3081,))])
train_mnist_dataset = MNIST(download=True, root="mnist_data",
                            transform=data_transform, train=True)
val_mnist_dataset = MNIST(download=False, root="mnist_data",
                          transform=data_transform, train=False)
train_loader = DataLoader(train_mnist_dataset,
                          batch_size=64, shuffle=True)
val_loader = DataLoader(val_mnist_dataset,
                        batch_size=128, shuffle=False)
```

Use callbacks and start train a model for 50 epochs.

```
from argus.callbacks import MonitorCheckpoint, EarlyStopping, ReduceLROnPlateau

callbacks = [
    MonitorCheckpoint(dir_path='mnist', monitor='val_accuracy', max_saves=3),
    EarlyStopping(monitor='val_accuracy', patience=9),
    ReduceLROnPlateau(monitor='val_accuracy', factor=0.5, patience=3)
]

model.fit(train_loader,
          val_loader=val_loader,
          num_epochs=50,
          metrics=['accuracy'],
          callbacks=callbacks)
```

Load model from checkpoint.

```
from pathlib import Path
from argus import load_model

del model
```

(continues on next page)

(continued from previous page)

```
model_path = Path("mnist/").glob("*.pth")
model_path = sorted(model_path)[-1]
print(f"Load model: {model_path}")
model = load_model(model_path)
print(model)
```

1.1.2 More flexibility

Argus can help you simplify the experiments with different architectures, losses, and optimizers. Let's define a *argus.model.Model* with two models via a dictionary. If you want to use PyTorch losses and optimizers it's not necessary to define them in argus model.

```
from torchvision.models import resnet18

class FlexModel(Model):
    nn_module = {
        'net': Net,
        'resnet18': resnet18
    }
```

Create a model instance. Parameters for nn_module is a tuple where the first element is a name, second is arguments. PyTorch losses and optimizers can be selected by a string with a class name.

```
params = {
    'nn_module': ('resnet18', {
        'pretrained': False,
        'num_classes': 1
    }),
    'optimizer': ('Adam', {'lr': 0.01}),
    'loss': 'CrossEntropyLoss',
    'device': 'cuda'
}

model = FlexModel(params)
```

Argus allows managing different combinations of your pipeline.

If you need for more flexibility you can:

- Override methods of *argus.model.Model*. For example *argus.model.Model.train_step()* and *argus.model.Model.val_step()*.
- Create custom *argus.callbacks.Callback*.
- Use custom `argus.metrics.Metric`.

1.2 Examples

You can find examples here.

1.2.1 Basic examples

- MNIST example.

```
python mnist.py --device cuda
```

- MNIST VAE example.

```
python mnist_vae.py --device cuda
```

- CIFAR example.

```
python cifar_simple.py --device cuda
```

1.2.2 Advanced examples

- CIFAR with DPP, mixed precision and gradient accumulation.

  Single GPU training:

```
python cifar_advanced.py --batch_size 256 --lr 0.001
```

  Single machine 2 GPUs distributed data parallel training:

```
./cifar_advanced.sh 2 --batch_size 128 --lr 0.0005
```

  DDP training with Apex mixed precision and gradient accumulation:

```
./cifar_advanced.sh 2 --batch_size 128 --lr 0.0005 --amp --iter_size 2
```

- Custom build methods for creation of model parts.

1.2.3 Kaggle solutions

- 1st place solution for Freesound Audio Tagging 2019 (mel-spectrograms, mixed precision)
- 14th place solution for TGS Salt Identification Challenge (segmentation, MeanTeacher)
- 50th place solution for Quick, Draw! Doodle Recognition Challenge (gradient accumulation, training on 50M images)
- 66th place solution for Kaggle Airbus Ship Detection Challenge (instance segmentation)
- Solution for Humpback Whale Identification (metric learning: arcface, center loss)
- Solution for VSB Power Line Fault Detection (1d conv)
- Solution for Bengali.AI Handwritten Grapheme Classification (EMA, mixed precision, CutMix)
- Solution for ALASKA2 Image Steganalysis competition (DDP, EMA, mixed precision, BitMix)

1.3 argus.model

class argus.model.**Model**(*params: dict*)

eval()
Set the nn_module into eval mode.

fit(*train_loader: Iterable*, *val_loader: Optional[Iterable] = None*, *num_epochs: int = 1*, *metrics: Optional[List[Union[argus.metrics.metric.Metric, str]]] = None*, *metrics_on_train: bool = False*, *callbacks: Optional[List[argus.callbacks.callback.Callback]] = None*, *val_callbacks: Optional[List[argus.callbacks.callback.Callback]] = None*)
Train the argus model.

The method attaches metrics and callbacks to the train and validation, and runs the training process.

Parameters

- **train_loader** (*Iterable*) – The train data loader.
- **val_loader** (*Iterable, optional*) – The validation data loader. Defaults to *None*.
- **num_epochs** (*int, optional*) – Number of training epochs to run. Defaults to 1.
- **metrics** (list of argus.metrics.Metric, optional) – List of metrics to evaluate. By default, the metrics are evaluated on the validation data (if any) only. Defaults to *None*.
- **metrics_on_train** (*bool, optional*) – Evaluate the metrics on train data as well. Defaults to False.
- **callbacks** (list of *argus.callbacks.Callback*, optional) – List of callbacks to be attached to the training process. Defaults to *None*.
- **val_callbacks** (list of *argus.callbacks.Callback*, optional) – List of callbacks to be attached to the validation process. Defaults to *None*.

get_lr() → Union[float, List[float]]
Get the learning rate from the optimizer.

It could be a single value or a list of values in the case of multiple parameter groups.

Returns The learning rate value or a list of individual parameter groups learning rate values.

Return type (float or a list of floats)

predict(*input*)
Make a prediction with the given input.

The prediction process consists of the input tensor transferring to the model device, forward pass of the nn_module in *eval* mode and application of the prediction_transform to the raw prediction output.

Parameters **input** (*torch.Tensor*) – The input tensor to predict with. It will be transferred to the model device. The user is responsible for ensuring that the input tensor shape and type match the model.

Returns

Predictions as the result of the prediction_transform application.

Return type torch.Tensor or other type

save(*file_path: Union[str, pathlib.Path]*)
Save the argus model into a file.

The argus model is saved as a dict:

```
{
    'model_name': Name of the argus model,
    'params': Argus model parameters dict,
    'nn_state_dict': torch nn_module.state_dict()
}
```

The *state_dict* is always transferred to cpu prior to saving.

Parameters **file_path** (*str*) – Path to the argus model file.

set_lr(*lr: Union[float, List[float]]*)
Set the learning rate for the optimizer.

The method allows setting individual learning rates for the optimizer parameter groups as well as setting even learning rate for all parameters.

Parameters **lr** (*number or list/tuple of numbers*) – The learning rate to set. If a single number is provided, all parameter groups learning rates are set to the same value. In order to set individual learning rates for each parameter group, a list or tuple of values with the corresponding length should be provided.

Raises

- **ValueError** – If *lr* is a list or tuple and its length is not equal to the number of parameter groups.
- **ValueError** – If *lr* type is not list, tuple, or number.
- **AttributeError** – If the model is not *train_ready* (i.e. not all attributes are set).

train()
Set the nn_module into train mode.

train_step(*batch*, *state: argus.engine.engine.State*) → dict
Perform a single train step.

The method is used by `argus.engine.Engine`. The train step includes input and target tensor transition to the model device, forward pass, loss evaluation, backward pass, and the train batch prediction preparation with a *prediction_transform*.

Parameters

- **(tuple of 2 torch.Tensors** (*batch*) – (input, target)): The input data and target tensors to process.
- **state** (`argus.engine.State`) – The argus model state.

Returns

The train step results:

```
{
    'prediction': The train batch predictions,
    'target': The train batch target data on the model device,
    'loss': Loss function value
}
```

Return type dict

val_step(*batch*, *state: argus.engine.engine.State*) → dict
Perform a single validation step.

The method is used by `argus.engine.Engine`. The validation step includes input and target tensor transition to the model device, forward pass, loss evaluation, and the train batch prediction preparation with a *prediction_transform*.

Gradient calculations and the model weights update are omitted, which is the main difference with the *train_step()* method.

Parameters

- **(tuple of 2 torch.Tensors** (*batch*) – (input, target)): The input data and target tensors to process.
- **state** (`argus.engine.State`) – The argus model state.

Returns

The train step results:

```
{
    'prediction': The train batch predictions,
    'target': The train batch target data on the model device,
    'loss': Loss function value
}
```

Return type dict

validate(*val_loader: Optional[Iterable]*, *metrics: Optional[List[argus.metrics.metric.Metric]] = None*, *callbacks: Optional[List[argus.callbacks.callback.Callback]] = None*) → Dict[str, float]

Perform a validation.

Parameters

- **val_loader** (*Iterable*) – The validation data loader.
- **metrics** (list of `argus.metrics.Metric`, optional) – List of metrics to evaluate with the data. Defaults to *None*.
- **callbacks** (list of *argus.callbacks.Callback*, optional) – List of callbacks to be attached to the validation process. Defaults to *None*.

Returns The metrics dictionary.

Return type dict

1.3.1 Load argus model

`argus.model.`**load_model**(*file_path: Union[str, pathlib.Path]*, *nn_module=default*, *optimizer=default*, *loss=default*, *prediction_transform=default*, *device=default*, *change_params_func=Identity()*, *change_state_dict_func=Identity()*, *model_name=default*, ***kwargs*)

Load an argus model from a file.

The function allows loading an argus model, saved with *argus.model.Model.save()*. The model is always loaded in *eval* mode.

Parameters

- **file_path** (*str*) – Path to the file to load.
- **device** (str or `torch.device`, optional) – Device for the model. Defaults to None.

- **nn_module** (*dict, tuple or str, optional*) – Params of the nn_module to replace params in the state.
- **optimizer** (*dict, tuple or str, optional*) – Params of the optimizer to replace params in the state. Set to *None* if don't want to create optimizer in the loaded model.
- **loss** (*dict, tuple or str, optional*) – Params of the loss to replace params in the state. Set to *None* if don't want to create loss in the loaded model.
- **prediction_transform** (*dict, tuple or str, optional*) – Params of the prediction_transform to replace params in the state. Set to *None* if don't want to create prediction_transform in the loaded model.
- **change_params_func** (*function, optional*) – Function for modification of state params. Takes as input params from the loaded state, outputs params to model creation.
- **change_state_dict_func** (*function, optional*) – Function for modification of nn_module state dict. Takes as input state dict from the loaded state, outputs state dict to model creation.
- **model_name** (*str*) – Class name of *argus.model.Model*. By default uses name from loaded state.

Raises

- **ImportError** – If the model is not available in the scope. Often it means that it is not imported or defined.
- **FileNotFoundError** – If the file is not found by the *file_path*.

Returns Loaded argus model.

Return type *argus.model.Model*

1.4 argus.callbacks

All callbacks classes should inherit the base *argus.callbacks.Callback* class.

A callback may execute actions on the start and the end of the whole training process, each epoch or iteration, as well as any other custom events.

The actions should be specified within corresponding methods:

- `start`
- `complete`
- `epoch_start`
- `epoch_complete`
- `iteration_start`
- `iteration_complete`
- `catch_exception`

A simple custom callback which stops training after the specified time:

```
from time import time

from argus.engine import State
from argus.callbacks.callback import Callback


class TimerCallback(Callback):
    def __init__(self, time_limit: int):
        self.time_limit = time_limit
        self.start_time = 0

    def epoch_start(self, state: State):
        if state.epoch == 0:
            self.start_time = time()

    def iteration_complete(self, state: State):
        if time() - self.start_time > self.time_limit:
            state.stopped = True
            state.logger.info("Run out of time!")
```

class argus.callbacks.**Callback**
Base callback class.

Raises **TypeError** – Attribute is not callable.

1.4.1 Checkpoints

Callbacks for argus model saving.

class argus.callbacks.**Checkpoint**(*dir_path=''*, *file_format='model-{epoch:03d}-{train_loss:.6f}.pth'*, *max_saves=None*, *period=1*, *save_after_exception=False*)
Save the model with a given period.

In the simplest case, the callback can be used to save the model after each epoch.

Parameters

- **dir_path** (*str, optional*) – Directory to save checkpoints. The desired directory will be created if it does not exist. Defaults to ''.
- **file_format** (*str, optional*) – Model saving filename format. Any valid value names from the model State may be used. Defaults to 'model-{epoch:03d}-{train_loss:.6f}.pth'.
- **max_saves** (*int, optional*) – Number of last saved models to keep. Should be positive. If None - save all models. Defaults to None.
- **period** (*int, optional*) – Interval (number of epochs) between checkpoint saves. Defaults to 1.
- **save_after_exception** (*bool, optional*) – Save the model checkpoint after an exception occurs. Defaults to False.

save_model(*state: argus.engine.engine.State*, *file_path*)
Save model to file.

Override the method if you need custom checkpoint saving.

Parameters

- **state** (argus.engine.State) – State.
- **file_path** (*str*) – Checkpoint file path.

class argus.callbacks.**MonitorCheckpoint**(*dir_path=''*, *file_format='model-{epoch:03d}-{monitor:.6f}.pth'*, *max_saves=None*, *save_after_exception=False*, *monitor='val_loss'*, *better='auto'*)

Save the model checkpoints after a metric is improved.

The MonitorCheckpoint augments the simple Checkpoint with a metric monitoring. It saves the model after the defined metric is improved. It is possible to monitor loss values during training as well as any metric available in the model State.

Parameters

- **dir_path** (*str, optional*) – Directory to save checkpoints. The desired directory will be created if it does not exist. Defaults to ''.
- **file_format** (*str, optional*) – Model saving filename format. Any valid value names from the model State may be used. Defaults to 'model-{epoch:03d}-{monitor:.6f}.pth'.
- **max_saves** (*[type], optional*) – Number of last saved models to keep. Should be positive. If None - save all models. Defaults to None.
- **save_after_exception** (*bool, optional*) – Save the model checkpoint after an exception occurs. Defaults to False.
- **monitor** (*str, optional*) – Metric name to monitor. It should be prepended with *val_* for the metric value on validation data and *train_* for the metric value on the date from the train loader. A val_loader should be provided during the model fit to make it possible to monitor metrics start with *val_*. Defaults to *val_loss*.
- **better** (*str, optional*) – The metric improvement criterion. Should be 'min', 'max' or 'auto'. 'auto' means the criterion should be taken from the metric itself, which is appropriate behavior in most cases. Defaults to 'auto'.

1.4.2 Early stopping

A callback for argus model train stop after a metric has stopped improving.

class argus.callbacks.**EarlyStopping**(*monitor='val_loss'*, *patience=1*, *better='auto'*)

Stop the model training after its metric has stopped improving.

It is possible to monitor loss values during training as well as any metric available in the model State.

Parameters

- **monitor** (*str, optional*) – Metric name to monitor. It should be prepended with *val_* for the metric value on validation data and *train_* for the metric value on the date from the train loader. A val_loader should be provided during the model fit to make it possible to monitor metrics start with *val_*. Defaults to *val_loss*.
- **patience** (*int, optional*) – Number of training epochs without the metric improvement to stop training. Defaults to 1.
- **better** (*str, optional*) – The metric improvement criterion. Should be 'min', 'max' or 'auto'. 'auto' means the criterion should be taken from the metric itself, which is appropriate behavior in most cases. Defaults to 'auto'.

1.4.3 Learning rate schedulers

Callbacks for auto adjust the learning rate based on the number of epochs or other metrics measurements.

The learning rates schedulers allow implementing dynamic learning rate changing policy. These callbacks are wrappers of native PyTorch `torch.optim.lr_scheduler`.

Currently, the following schedulers are available (see PyTorch documentation by the links provided for details on the schedulers algorithms themself):

LambdaLR

class argus.callbacks.**LambdaLR**(*lr_lambda*, *step_on_iteration=False*)
LambdaLR scheduler.

Multiply learning rate by a factor computed with a given function. The function should take int value number of epochs as the only argument.

Parameters

- **lr_lambda** (*function or list of functions*) – Lambda function for the learning rate factor computation.
- **step_on_iteration** (*bool*) – Step on each training iteration rather than each epoch. Defaults to False.

PyTorch docs on `torch.optim.lr_scheduler.LambdaLR`.

StepLR

class argus.callbacks.**StepLR**(*step_size*, *gamma=0.1*, *step_on_iteration=False*)
StepLR scheduler.

Multiply learning rate by a given factor with a given period.

Parameters

- **step_size** (*int*) – Period of learning rate update in epochs.
- **gamma** (*float, optional*) – Multiplicative factor. Defaults to 0.1.
- **step_on_iteration** (*bool*) – Step on each training iteration rather than each epoch. Defaults to False.

PyTorch docs on `torch.optim.lr_scheduler.StepLR`.

MultiStepLR

class argus.callbacks.**MultiStepLR**(*milestones*, *gamma=0.1*, *step_on_iteration=False*)
MultiStepLR scheduler.

Multiply learning rate by a given factor on each epoch from a given list.

Parameters

- **milestones** (*list of ints*) – List of epochs number to perform lr step.
- **gamma** (*float, optional*) – Multiplicative factor. Defaults to 0.1.
- **step_on_iteration** (*bool*) – Step on each training iteration rather than each epoch. Defaults to False.

PyTorch docs on torch.optim.lr_scheduler.MultiStepLR.

ExponentialLR

class argus.callbacks.**ExponentialLR**(*gamma*, *step_on_iteration=False*)
MultiStepLR scheduler.

Multiply learning rate by a given factor on each epoch.

Parameters

- **gamma** (*float, optional*) – Multiplicative factor. Defaults to 0.1.
- **step_on_iteration** (*bool*) – Step on each training iteration rather than each epoch. Defaults to False.

PyTorch docs on torch.optim.lr_scheduler.ExponentialLR.

CosineAnnealingLR

class argus.callbacks.**CosineAnnealingLR**(*T_max*, *eta_min=0*, *step_on_iteration=False*)
CosineAnnealingLR scheduler.

Set the learning rate of each parameter group using a cosine annealing schedule.

Parameters

- **T_max** (*int*) – Max number of epochs or iterations.
- **eta_min** (*float, optional*) – Min learning rate. Defaults to 0.
- **step_on_iteration** (*bool*) – Step on each training iteration rather than each epoch. Defaults to False.

PyTorch docs on torch.optim.lr_scheduler.CosineAnnealingLR.

ReduceLROnPlateau

class argus.callbacks.**ReduceLROnPlateau**(*monitor='val_loss'*, *better='auto'*, *factor=0.1*, *patience=10*, *verbose=False*, *threshold=0.0001*, *threshold_mode='rel'*, *cooldown=0*, *min_lr=0*, *eps=1e-08*)
ReduceLROnPlateau scheduler.

Reduce learning rate when a metric has stopped improving.

Parameters

- **monitor** (*str, optional*) – Metric name to monitor. It should be prepended with *val_* for the metric value on validation data and *train_* for the metric value on the date from the train loader. A val_loader should be provided during the model fit to make it possible to monitor metrics start with *val_*. Defaults to *val_loss*.
- **better** (*str, optional*) – The metric improvement criterion. Should be 'min', 'max' or 'auto'. 'auto' means the criterion should be taken from the metric itself, which is appropriate behavior in most cases. Defaults to 'auto'.
- **factor** (*float, optional*) – Multiplicative factor. Defaults to 0.1.
- **patience** (*int, optional*) – Number of training epochs without the metric improvement to update the learning rate. Defaults to 10.

- **verbose** (*bool, optional*) – Print info on each update to stdout. Defaults to False.
- **threshold** (*float, optional*) – Threshold for considering the changes significant. Defaults to 1e-4.
- **threshold_mode** (*str, optional*) – Should be 'rel', 'abs'. Defaults to 'rel'.
- **cooldown** (*int, optional*) – Number of epochs to wait before resuming normal operation after lr has been updated. Defaults to 0.
- **min_lr** (*float or list of floats, optional*) – Min learning rate. Defaults to 0.
- **eps** (*float, optional*) – Min significant learning rate update. Defaults to 1e-8.

PyTorch docs on `torch.optim.lr_scheduler.ReduceLROnPlateau`.

CyclicLR

class argus.callbacks.**CyclicLR**(*base_lr*, *max_lr*, *step_size_up=2000*, *step_size_down=None*, *mode='triangular'*, *gamma=1.0*, *scale_fn=None*, *scale_mode='cycle'*, *cycle_momentum=True*, *base_momentum=0.8*, *max_momentum=0.9*, *step_on_iteration=True*)

CyclicLR scheduler.

Sets the learning rate of each parameter group according to cyclical learning rate policy.

Parameters

- **base_lr** (*float or list of floats*) – Initial learning rate.
- **max_lr** (*float or list of floats*) – Max learning rate.
- **step_size_up** (*int, optional*) – Increase phase duration in epochs or iterations. Defaults to 2000.
- **step_size_down** (*int, optional*) – Decrease phase duration in epochs or iterations. Defaults to None.
- **mode** (*str, optional*) – Should be 'triangular', 'triangular2' or 'exp_range'. Defaults to 'triangular'.
- **gamma** (*float, optional*) – Constant for the 'exp_range' policy. Defaults to 1.
- **scale_fn** (*function, optional*) – Custom scaling policy function. Defaults to None.
- **scale_mode** (*str, optional*) – Should be 'cycle' or 'iterations'. Defaults to 'cycle'.
- **cycle_momentum** (*bool, optional*) – Momentum is cycled inversely to learning rate between 'base_momentum' and 'max_momentum'. Defaults to True.
- **base_momentum** (*float or list of floats, optional*) – Lower momentum boundaries in the cycle for each parameter group. Defaults to 0.8.
- **max_momentum** (*float or list of floats, optional*) – Upper momentum boundaries in the cycle for each parameter group. Defaults to 0.9.
- **step_on_iteration** (*bool*) – Step on each training iteration rather than each epoch. Defaults to True.

PyTorch docs on `torch.optim.lr_scheduler.CyclicLR`.

CosineAnnealingWarmRestarts

class argus.callbacks.**CosineAnnealingWarmRestarts**(*T_0*, *T_mult=1*, *eta_min=0*, *step_on_iteration=False*)

CosineAnnealingLR scheduler.

Set the learning rate of each parameter group using a cosine annealing schedule with a warm restart.

Parameters

- **T_0** (*int*) – Number of epochs or iterations for the first restart.
- **T_mult** (*int*) – T increase factor after a restart.
- **eta_min** (*float, optional*) – Min learning rate. Defaults to 0.
- **step_on_iteration** (*bool*) – Step on each training iteration rather than each epoch. Defaults to False.

PyTorch docs on torch.optim.lr_scheduler.CosineAnnealingWarmRestarts.

MultiplicativeLR

class argus.callbacks.**MultiplicativeLR**(*lr_lambda*, *step_on_iteration=False*)

MultiplicativeLR scheduler.

Multiply the learning rate of each parameter group by the factor given in the specified function.

Parameters

- **lr_lambda** (*function or list*) – A function which computes a multiplicative factor given an integer parameter epoch, or a list of such functions, one for each group in optimizer.param_groups.
- **step_on_iteration** (*bool*) – Step on each training iteration rather than each epoch. Defaults to False.

PyTorch docs on torch.optim.lr_scheduler.MultiplicativeLR.

OneCycleLR

class argus.callbacks.**OneCycleLR**(*max_lr*, *total_steps=None*, *epochs=None*, *steps_per_epoch=None*, *pct_start=0.3*, *anneal_strategy='cos'*, *cycle_momentum=True*, *base_momentum=0.85*, *max_momentum=0.95*, *div_factor=25.0*, *final_div_factor=10000.0*)

OneCycleLR scheduler.

Sets the learning rate of each parameter group according to the 1cycle learning rate policy. The 1cycle policy anneals the learning rate from an initial learning rate to some maximum learning rate and then from that maximum learning rate to some minimum learning rate much lower than the initial learning rate.

Parameters

- **max_lr** (*float or list*) – Upper learning rate boundaries in the cycle for each parameter group.
- **total_steps** (*int*) – The total number of steps in the cycle. Note that if a value is not provided here, then it must be inferred by providing a value for epochs and steps_per_epoch. Defaults to None.

- **epochs** (*int*) – The number of epochs to train for. This is used along with steps_per_epoch in order to infer the total number of steps in the cycle if a value for total_steps is not provided. Defaults to None.
- **steps_per_epoch** (*int*) – The number of steps per epoch to train for. This is used along with epochs in order to infer the total number of steps in the cycle if a value for total_steps is not provided. Defaults to None.
- **pct_start** (*float*) – The percentage of the cycle (in number of steps) spent increasing the learning rate. Defaults to 0.3.
- **anneal_strategy** (*str*) – {'cos', 'linear'} Specifies the annealing strategy: "cos" for cosine annealing, "linear" for linear annealing. Defaults to 'cos'.
- **cycle_momentum** (*bool*) – If `True`, momentum is cycled inversely to learning rate between 'base_momentum' and 'max_momentum'. Defaults to True.
- **base_momentum** (*float or list*) – Lower momentum boundaries in the cycle for each parameter group. Note that momentum is cycled inversely to learning rate; at the peak of a cycle, momentum is 'base_momentum' and learning rate is 'max_lr'. Defaults to 0.85.
- **max_momentum** (*float or list*) – Upper momentum boundaries in the cycle for each parameter group. Functionally, it defines the cycle amplitude (max_momentum - base_momentum). Note that momentum is cycled inversely to learning rate; at the start of a cycle, momentum is 'max_momentum' and learning rate is 'base_lr' Defaults to 0.95.
- **div_factor** (*float*) – Determines the initial learning rate via initial_lr = max_lr/div_factor Defaults to 25.
- **final_div_factor** (*float*) – Determines the minimum learning rate via min_lr = initial_lr/final_div_factor Defaults to 1e4.

PyTorch docs on `torch.optim.lr_scheduler.OneCycleLR`.

1.4.4 Logging

Callbacks for logging argus model training process.

class argus.callbacks.**LoggingToFile**(*file_path*, *create_dir=True*, *formatter='[%(asctime)s][%(levelname)s]: %(message)s'*, *append=False*)

Write the argus model training progress into a file.

It adds a standard Python logger to log all losses and metrics values during training. The logger is used to output other messages, like info from callbacks and errors.

Parameters

- **file_path** (*str*) – Path to the logging file.
- **create_dir** (*bool, optional*) – Create the directory for the logging file if it does not exist. Defaults to True.
- **formatter** (*str, optional*) – Standard Python logging formatter to format the log messages. Defaults to '%(asctime)s %(levelname)s %(message)s'.
- **append** (*bool, optional*) – Append the log file if it already exists or rewrite it. Defaults to False.

class argus.callbacks.**LoggingToCSV**(*file_path*, *create_dir=True*, *separator=','*, *write_header=True*, *append=False*)

Write the argus model training progress into a CSV file.

It logs all losses and metrics values during training into a .csv file for for further analysis or visualization.

Parameters

- **file_path** (*str*) – Path to the .csv logging file.
- **create_dir** (*bool, optional*) – Create the directory for the logging file if it does not exist. Defaults to True.
- **separator** (*str, optional*) – Values separator character to use. Defaults to ‘,’.
- **write_header** (*bool, optional*) – Write the column headers. Defaults to True.
- **append** (*bool, optional*) – Append the log file if it already exists or rewrite it. Defaults to False.

INDEX