# torch.optim

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)optim.rst`, line 4)

Unknown directive type "automodule".

```
.. automodule:: torch.optim
```

## How to use an optimizer

To use :mod:`torch.optim` you have to construct an optimizer object, that will hold the current state and will update the parameters based on the computed gradients.

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)optim.rst`, line 9); *backlink*

Unknown interpreted text role "mod".

### Constructing it

To construct an :class:`Optimizer` you have to give it an iterable containing the parameters (all should be :class:`~torch.autograd.Variable` s) to optimize. Then, you can specify optimizer-specific options such as the learning rate, weight decay, etc.

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)optim.rst`, line 15); *backlink*

Unknown interpreted text role "class".

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)optim.rst`, line 15); *backlink*

Unknown interpreted text role "class".

**Note**

If you need to move a model to GPU via `.cuda()`, please do so before constructing optimizers for it. Parameters of a model after `.cuda()` will be different objects with those before the call.

In general, you should make sure that optimized parameters live in consistent locations when optimizers are constructed and used.

Example:

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
optimizer = optim.Adam([var1, var2], lr=0.0001)
```

### Per-parameter options

:class:`Optimizer` s also support specifying per-parameter options. To do this, instead of passing an iterable of :class:`~torch.autograd.Variable` s, pass in an iterable of :class:`dict` s. Each of them will define a separate parameter group, and should contain a `params` key, containing a list of parameters belonging to it. Other keys should match the keyword arguments accepted by the optimizers, and will be used as optimization options for this group.

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)optim.rst`, line 36); *backlink*

Unknown interpreted text role "class".

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)optim.rst`, line 36); *backlink*

Unknown interpreted text role "class".

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master) (docs) (source)optim.rst`, line 36); *backlink*

Unknown interpreted text role "class".

**Note**

You can still pass options as keyword arguments. They will be used as defaults, in the groups that didn't override them. This is useful when you only want to vary a single option, while keeping all others consistent between parameter groups.

For example, this is very useful when one wants to specify per-layer learning rates:

```
optim.SGD([
                {'params': model.base.parameters()},
                {'params': model.classifier.parameters(), 'lr': 1e-3}
            ], lr=1e-2, momentum=0.9)
```

This means that `model.base`'s parameters will use the default learning rate of `1e-2`, `model.classifier`'s parameters will use a learning rate of `1e-3`, and a momentum of `0.9` will be used for all parameters.

### Taking an optimization step

All optimizers implement a :func:`~Optimizer.step` method, that updates the parameters. It can be used in two ways:

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master) (docs) (source)optim.rst`, line 65); *backlink*

Unknown interpreted text role "func".

**`optimizer.step()`**

This is a simplified version supported by most optimizers. The function can be called once the gradients are computed using e.g. :func:`~torch.autograd.Variable.backward`.

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master) (docs) (source)optim.rst`, line 71); *backlink*

Unknown interpreted text role "func".

Example:

```
for input, target in dataset:
    optimizer.zero_grad()
    output = model(input)
    loss = loss_fn(output, target)
    loss.backward()
    optimizer.step()
```

**`optimizer.step(closure)`**

Some optimization algorithms such as Conjugate Gradient and LBFGS need to reevaluate the function multiple times, so you have to pass in a closure that allows them to recompute your model. The closure should clear the gradients, compute the loss, and return it.

Example:

```
for input, target in dataset:
    def closure():
        optimizer.zero_grad()
        output = model(input)
        loss = loss_fn(output, target)
        loss.backward()
        return loss
    optimizer.step(closure)
```

## Base class

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master) (docs) (source)optim.rst`, line 108)

Unknown directive type "autoclass".

```
.. autoclass:: Optimizer
```

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master) (docs) (source)optim.rst`, line 110)

Unknown directive type "autosummary".

```
.. autosummary::
    :toctree: generated
    :nosignatures:

    Optimizer.add_param_group
    Optimizer.load_state_dict
    Optimizer.state_dict
    Optimizer.step
    Optimizer.zero_grad
```

## Algorithms

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master) (docs) (source)optim.rst`, line 123)

Unknown directive type "autosummary".

```
.. autosummary::
    :toctree: generated
    :nosignatures:

    Adadelta
    Adagrad
    Adam
    AdamW
    SparseAdam
    Adamax
    ASGD
    LBFGS
    NAdam
    RAdam
    RMSprop
    Rprop
    SGD
```

## How to adjust learning rate

:mod:\`torch.optim.lr_scheduler\` provides several methods to adjust the learning rate based on the number of epochs. :class:\`torch.optim.lr_scheduler.ReduceLROnPlateau\` allows dynamic learning rate reducing based on some validation measurements.

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master) (docs) (source)optim.rst`, line 144); *backlink*

Unknown interpreted text role "mod".

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master) (docs) (source)optim.rst`, line 144); *backlink*

Unknown interpreted text role "class".

Learning rate scheduling should be applied after optimizer's update; e.g., you should write your code this way:

Example:

```
model = [Parameter(torch.randn(2, 2, requires_grad=True))]
optimizer = SGD(model, 0.1)
scheduler = ExponentialLR(optimizer, gamma=0.9)

for epoch in range(20):
    for input, target in dataset:
        optimizer.zero_grad()
        output = model(input)
        loss = loss_fn(output, target)
        loss.backward()
```

```
        optimizer.step()
    scheduler.step()
```

Most learning rate schedulers can be called back-to-back (also referred to as chaining schedulers). The result is that each scheduler is applied one after the other on the learning rate obtained by the one preceding it.

Example:

```
model = [Parameter(torch.randn(2, 2, requires_grad=True))]
optimizer = SGD(model, 0.1)
scheduler1 = ExponentialLR(optimizer, gamma=0.9)
scheduler2 = MultiStepLR(optimizer, milestones=[30,80], gamma=0.1)

for epoch in range(20):
    for input, target in dataset:
        optimizer.zero_grad()
        output = model(input)
        loss = loss_fn(output, target)
        loss.backward()
        optimizer.step()
    scheduler1.step()
    scheduler2.step()
```

In many places in the documentation, we will use the following template to refer to schedulers algorithms.

```
>>> scheduler = ...
>>> for epoch in range(100):
>>>     train(...)
>>>     validate(...)
>>>     scheduler.step()
```

**Warning**

Prior to PyTorch 1.1.0, the learning rate scheduler was expected to be called before the optimizer's update; 1.1.0 changed this behavior in a BC-breaking way. If you use the learning rate scheduler (calling `scheduler.step()`) before the optimizer's update (calling `optimizer.step()`), this will skip the first value of the learning rate schedule. If you are unable to reproduce results after upgrading to PyTorch 1.1.0, please check if you are calling `scheduler.step()` at the wrong time.

**System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master) (docs) (source)optim.rst`, line 205)**

Unknown directive type "autosummary".

```
.. autosummary::
    :toctree: generated
    :nosignatures:

    lr_scheduler.LambdaLR
    lr_scheduler.MultiplicativeLR
    lr_scheduler.StepLR
    lr_scheduler.MultiStepLR
    lr_scheduler.ConstantLR
    lr_scheduler.LinearLR
    lr_scheduler.ExponentialLR
    lr_scheduler.CosineAnnealingLR
    lr_scheduler.ChainedScheduler
    lr_scheduler.SequentialLR
    lr_scheduler.ReduceLROnPlateau
    lr_scheduler.CyclicLR
    lr_scheduler.OneCycleLR
    lr_scheduler.CosineAnnealingWarmRestarts
```

## Stochastic Weight Averaging

:mod:`torch.optim.swa_utils` implements Stochastic Weight Averaging (SWA). In particular, :class:`torch.optim.swa_utils.AveragedModel` class implements SWA models, :class:`torch.optim.swa_utils.SWALR` implements the SWA learning rate scheduler and :func:`torch.optim.swa_utils.update_bn` is a utility function used to update SWA batch normalization statistics at the end of training.

**System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master) (docs) (source)optim.rst`, line 227); *backlink***

Unknown interpreted text role "mod".

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master) (docs) (source)optim.rst`, line 227); *backlink*

Unknown interpreted text role "class".

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master) (docs) (source)optim.rst`, line 227); *backlink*

Unknown interpreted text role "class".

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master) (docs) (source)optim.rst`, line 227); *backlink*

Unknown interpreted text role "func".

SWA has been proposed in Averaging Weights Leads to Wider Optima and Better Generalization.

## Constructing averaged models

*AveragedModel* class serves to compute the weights of the SWA model. You can create an averaged model by running:

```
>>> swa_model = AveragedModel(model)
```

Here the model `model` can be an arbitrary :class:`torch.nn.Module` object. `swa_model` will keep track of the running averages of the parameters of the `model`. To update these averages, you can use the :func:`update_parameters` function:

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master) (docs) (source)optim.rst`, line 245); *backlink*

Unknown interpreted text role "class".

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master) (docs) (source)optim.rst`, line 245); *backlink*

Unknown interpreted text role "func".

```
>>> swa_model.update_parameters(model)
```

## SWA learning rate schedules

Typically, in SWA the learning rate is set to a high constant value. :class:`SWALR` is a learning rate scheduler that anneals the learning rate to a fixed value, and then keeps it constant. For example, the following code creates a scheduler that linearly anneals the learning rate from its initial value to 0.05 in 5 epochs within each parameter group:

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master) (docs) (source)optim.rst`, line 255); *backlink*

Unknown interpreted text role "class".

```
>>> swa_scheduler = torch.optim.swa_utils.SWALR(optimizer, \
>>>         anneal_strategy="linear", anneal_epochs=5, swa_lr=0.05)
```

You can also use cosine annealing to a fixed value instead of linear annealing by setting `anneal_strategy="cos"`.

## Taking care of batch normalization

:func:`update_bn` is a utility function that allows to compute the batchnorm statistics for the SWA model on a given dataloader `loader` at the end of training:

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master) (docs) (source)optim.rst`, line 270); *backlink*

Unknown interpreted text role "func".

```
>>> torch.optim.swa_utils.update_bn(loader, swa_model)
```

:func:`update_bn` applies the `swa_model` to every element in the dataloader and computes the activation statistics for each batch normalization layer in the model.

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)optim.rst`, line 275); *backlink*

Unknown interpreted text role "func".

**Warning**

:func:`update_bn` assumes that each batch in the dataloader `loader` is either a tensors or a list of tensors where the first element is the tensor that the network `swa_model` should be applied to. If your dataloader has a different structure, you can update the batch normalization statistics of the `swa_model` by doing a forward pass with the `swa_model` on each element of the dataset.

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)optim.rst`, line 279); *backlink*

Unknown interpreted text role "func".

### Custom averaging strategies

By default, :class:`torch.optim.swa_utils.AveragedModel` computes a running equal average of the parameters that you provide, but you can also use custom averaging functions with the `avg_fn` parameter. In the following example `ema_model` computes an exponential moving average.

**System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\(pytorch-master)(docs)(source)optim.rst`, line 288); *backlink*

Unknown interpreted text role "class".

Example:

```
>>> ema_avg = lambda averaged_model_parameter, model_parameter, num_averaged:\
>>>         0.1 * averaged_model_parameter + 0.9 * model_parameter
>>> ema_model = torch.optim.swa_utils.AveragedModel(model, avg_fn=ema_avg)
```

### Putting it all together

In the example below, `swa_model` is the SWA model that accumulates the averages of the weights. We train the model for a total of 300 epochs and we switch to the SWA learning rate schedule and start to collect SWA averages of the parameters at epoch 160:

```
>>> loader, optimizer, model, loss_fn = ...
>>> swa_model = torch.optim.swa_utils.AveragedModel(model)
>>> scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=300)
>>> swa_start = 160
>>> swa_scheduler = SWALR(optimizer, swa_lr=0.05)
>>>
>>> for epoch in range(300):
>>>       for input, target in loader:
>>>           optimizer.zero_grad()
>>>           loss_fn(model(input), target).backward()
>>>           optimizer.step()
>>>       if epoch > swa_start:
>>>           swa_model.update_parameters(model)
>>>           swa_scheduler.step()
>>>       else:
>>>           scheduler.step()
>>>
>>> # Update bn statistics for the swa_model at the end
>>> torch.optim.swa_utils.update_bn(loader, swa_model)
>>> # Use swa_model to make predictions on test data
>>> preds = swa_model(test_input)
```