PyTorch

Table of Contents

TORCH.OPTIM

`torch.optim` is a package implementing various optimization algorithms.

Most commonly used methods are already supported, and the interface is general enough, so that more sophisticated ones can also be easily integrated in the future.

How to use an optimizer

To use `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.

Constructing it

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

Example:

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

Per-parameter options

`Optimizer` s also support specifying per-parameter options. To do this, instead of passing an iterable of `Variable` s, pass in an iterable of `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.

• 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 `step()` method, that updates the parameters. It can be used in two ways:

`optimizer.step()`

This is a simplified version supported by most optimizers. The function can be called once the gradients are computed using e.g. `backward()` .

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

CLASS torch.optim.Optimizer(*params*, *defaults*) [SOURCE]

Base class for all optimizers.

• WARNING

Parameters need to be specified as collections that have a deterministic ordering that is consistent between runs. Examples of objects that don't satisfy those properties are sets and iterators over values of dictionaries.

Parameters

- **params** (*iterable*) – an iterable of `torch.Tensor` s or `dict` s. Specifies what Tensors should be optimized.
- **defaults** (*Dict*[*str*, *Any*]) – (dict): a dict containing default values of optimization options (used when a parameter group doesn't specify them).

`Optimizer.add_param_group`	Add a param group to the `Optimizer` s *param_groups*.
`Optimizer.load_state_dict`	Loads the optimizer state.
`Optimizer.state_dict`	Returns the state of the optimizer as a `dict`.
`Optimizer.step`	Performs a single optimization step (parameter update).
`Optimizer.zero_grad`	Resets the gradients of all optimized `torch.Tensor` s.

Algorithms

`Adadelta`	Implements Adadelta algorithm.
`Adagrad`	Implements Adagrad algorithm.
`Adam`	Implements Adam algorithm.
`AdamW`	Implements AdamW algorithm.
`SparseAdam`	SparseAdam implements a masked version of the Adam algorithm suitable for sparse gradients.
`Adamax`	Implements Adamax algorithm (a variant of Adam based on infinity norm).
`ASGD`	Implements Averaged Stochastic Gradient Descent.
`LBFGS`	Implements L-BFGS algorithm.
`NAdam`	Implements NAdam algorithm.
`RAdam`	Implements RAdam algorithm.

`RMSprop`	Implements RMSprop algorithm.
`Rprop`	Implements the resilient backpropagation algorithm.
`SGD`	Implements stochastic gradient descent (optionally with momentum).

Many of our algorithms have various implementations optimized for performance, readability and/or generality, so we attempt to default to the generally fastest implementation for the current device if no particular implementation has been specified by the user.

We have 3 major categories of implementations: for-loop, foreach (multi-tensor), and fused. The most straightforward implementations are for-loops over the parameters with big chunks of computation. For-looping is usually slower than our foreach implementations, which combine parameters into a multi-tensor and run the big chunks of computation all at once, thereby saving many sequential kernel calls. A few of our optimizers have even faster fused implementations, which fuse the big chunks of computation into one kernel. We can think of foreach implementations as fusing horizontally and fused implementations as fusing vertically on top of that.

In general, the performance ordering of the 3 implementations is fused > foreach > for-loop. So when applicable, we default to foreach over for-loop. Applicable means the foreach implementation is available, the user has not specified any implementation-specific kwargs (e.g., fused, foreach, differentiable), and all tensors are native and on CUDA. Note that while fused should be even faster than foreach, the implementations are newer and we would like to give them more bake-in time before flipping the switch everywhere. You are welcome to try them out though!

Below is a table showing the available and default implementations of each algorithm:

Algorithm	Default	Has foreach?	Has fused?
`Adadelta`	foreach	yes	no
`Adagrad`	foreach	yes	no
`Adam`	foreach	yes	yes
`AdamW`	foreach	yes	yes
`SparseAdam`	for-loop	no	no
`Adamax`	foreach	yes	no
`ASGD`	foreach	yes	no
`LBFGS`	for-loop	no	no
`NAdam`	foreach	yes	no
`RAdam`	foreach	yes	no
`RMSprop`	foreach	yes	no
`Rprop`	foreach	yes	no
`SGD`	foreach	yes	no

How to adjust learning rate

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

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

Example:

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
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:

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
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.

`lr_scheduler.LambdaLR`	Sets the learning rate of each parameter group to the initial lr times a given function.
`lr_scheduler.MultiplicativeLR`	Multiply the learning rate of each parameter group by the factor given in the specified function.
`lr_scheduler.StepLR`	Decays the learning rate of each parameter group by gamma every step_size epochs.
`lr_scheduler.MultiStepLR`	Decays the learning rate of each parameter group by gamma once the number of epoch reaches one of the milestones.
`lr_scheduler.ConstantLR`	Decays the learning rate of each parameter group by a small constant factor until the number of epoch reaches a pre-defined milestone: total_iters.
`lr_scheduler.LinearLR`	Decays the learning rate of each parameter group by linearly changing small multiplicative factor until the number of epoch reaches a pre-defined milestone: total_iters.
`lr_scheduler.ExponentialLR`	Decays the learning rate of each parameter group by gamma every epoch.
`lr_scheduler.PolynomialLR`	Decays the learning rate of each parameter group using a polynomial function in the given total_iters.
`lr_scheduler.CosineAnnealingLR`	Set the learning rate of each parameter group using a cosine annealing schedule, where η_{max} is set to the initial lr and T_{cur} is the number of epochs since the last restart in SGDR:
`lr_scheduler.ChainedScheduler`	Chains list of learning rate schedulers.

`lr_scheduler.SequentialLR`	Receives the list of schedulers that is expected to be called sequentially during optimization process and milestone points that provides exact intervals to reflect which scheduler is supposed to be called at a given epoch.
`lr_scheduler.ReduceLROnPlateau`	Reduce learning rate when a metric has stopped improving.
`lr_scheduler.CyclicLR`	Sets the learning rate of each parameter group according to cyclical learning rate policy (CLR).
`lr_scheduler.OneCycleLR`	Sets the learning rate of each parameter group according to the 1cycle learning rate policy.
`lr_scheduler.CosineAnnealingWarmRestarts`	Set the learning rate of each parameter group using a cosine annealing schedule, where η_{max} is set to the initial lr, T_{cur} is the number of epochs since the last restart and T_i is the number of epochs between two warm restarts in SGDR:

Weight Averaging (SWA and EMA)

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

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

EMA is a widely known technique to reduce the training time by reducing the number of weight updates needed. It is a variation of Polyak averaging, but using exponential weights instead of equal weights across iterations.

Constructing averaged models

The *AveragedModel* class serves to compute the weights of the SWA or EMA model.

You can create an SWA averaged model by running:

```
>>> averaged_model = AveragedModel(model)
```

EMA models are constructed by specifying the `multi_avg_fn` argument as follows:

```
>>> decay = 0.999
>>> averaged_model = AveragedModel(model, multi_avg_fn=get_ema_multi_avg_fn(decay))
```

Decay is a parameter between 0 and 1 that controls how fast the averaged parameters are decayed. If not provided to `get_ema_multi_avg_fn`, the default is 0.999.

`get_ema_multi_avg_fn` returns a function that applies the following EMA equation to the weights:

$$W_{t+1}^{\text{EMA}} = \alpha W_t^{\text{EMA}} + (1-\alpha) W_t^{\text{model}}$$

where alpha is the EMA decay.

Here the model `model` can be an arbitrary `torch.nn.Module` object. `averaged_model` will keep track of the running averages of the parameters of the `model`. To update these averages, you should use the `update_parameters()` function after the *optimizer.step*():

```
>>> averaged_model.update_parameters(model)
```

For SWA and EMA, this call is usually done right after the optimizer `step()`. In the case of SWA, this is usually skipped for some numbers of steps at the beginning of the training.

Custom averaging strategies

By default, `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` or `multi_avg_fn` parameters:

- `avg_fn` allows defining a function operating on each parameter tuple (averaged parameter, model parameter) and should return the new averaged parameter.
- `multi_avg_fn` allows defining more efficient operations acting on a tuple of parameter lists, (averaged parameter list, model parameter list), at the same time, for example using the `torch._foreach*` functions. This function must update the averaged parameters in-place.

In the following example `ema_model` computes an exponential moving average using the `avg_fn` parameter:

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

In the following example `ema_model` computes an exponential moving average using the more efficient `multi_avg_fn` parameter:

```
>>> ema_model = AveragedModel(model, multi_avg_fn=get_ema_multi_avg_fn(0.9))
```

SWA learning rate schedules

Typically, in SWA the learning rate is set to a high constant value. `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:

```
>>> 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

`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:

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

`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.

• WARNING

`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.

Putting it all together: SWA

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)
```

Putting it all together: EMA

In the example below, `ema_model` is the EMA model that accumulates the exponentially-decayed averages of the weights with a decay rate of 0.999. We train the model for a total of 300 epochs and start to collect EMA averages immediately.

```
>>> loader, optimizer, model, loss_fn = ...
>>> ema_model = torch.optim.swa_utils.AveragedModel(model, \
>>>             multi_avg_fn=torch.optim.swa_utils.get_ema_multi_avg_fn(0.999))
>>>
>>> for epoch in range(300):
>>>       for input, target in loader:
>>>           optimizer.zero_grad()
>>>           loss_fn(model(input), target).backward()
>>>           optimizer.step()
>>>           ema_model.update_parameters(model)
>>>
>>> # Update bn statistics for the ema_model at the end
>>> torch.optim.swa_utils.update_bn(loader, ema_model)
>>> # Use ema_model to make predictions on test data
>>> preds = ema_model(test_input)
```

‹ Previous Next ›

Docs

Access comprehensive developer documentation for PyTorch

View Docs >

Tutorials

Get in-depth tutorials for beginners and advanced developers

View Tutorials >

Resources

Find development resources and get your questions answered

View Resources >