CUDA Automatic Mixed Precision examples

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 6)

Unknown directive type "currentmodule".

```
.. currentmodule:: torch.cuda.amp
```

Ordinarily, "automatic mixed precision training" means training with :class:`torch.cuda.amp.autocast` and :class:`torch.cuda.amp.GradScaler` together.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 8); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 8); *backlink*

Unknown interpreted text role "class".

Instances of :class:`torch.cuda.amp.autocast` enable autocasting for chosen regions. Autocasting automatically chooses the precision for GPU operations to improve performance while maintaining accuracy.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 11); *backlink*

Unknown interpreted text role "class".

Instances of :class:`torch.cuda.amp.GradScaler` help perform the steps of gradient scaling conveniently. Gradient scaling improves convergence for networks with `float16` gradients by minimizing gradient underflow, as explained :ref:`here<gradient-scaling>`.

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

Unknown interpreted text role "class".

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

Unknown interpreted text role "ref".

:class:`torch.cuda.amp.autocast` and :class:`torch.cuda.amp.GradScaler` are modular. In the samples below, each is used as its individual documentation suggests.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 19); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 19); *backlink*

Unknown interpreted text role "class".

(Samples here are illustrative. See the Automatic Mixed Precision recipe for a runnable walkthrough.)

Typical Mixed Precision Training

```
# Creates model and optimizer in default precision
model = Net().cuda()
optimizer = optim.SGD(model.parameters(), ...)

# Creates a GradScaler once at the beginning of training.
scaler = GradScaler()

for epoch in epochs:
    for input, target in data:
        optimizer.zero_grad()

        # Runs the forward pass with autocasting.
        with autocast():
            output = model(input)
            loss = loss_fn(output, target)

        # Scales loss.  Calls backward() on scaled loss to create scaled gradients.
        # Backward passes under autocast are not recommended.
        # Backward ops run in the same dtype autocast chose for corresponding forward ops.
        scaler.scale(loss).backward()

        # scaler.step() first unscales the gradients of the optimizer's assigned params.
        # If these gradients do not contain infs or NaNs, optimizer.step() is then called,
        # otherwise, optimizer.step() is skipped.
        scaler.step(optimizer)

        # Updates the scale for next iteration.
        scaler.update()
```

Working with Unscaled Gradients

All gradients produced by `scaler.scale(loss).backward()` are scaled. If you wish to modify or inspect the parameters' `.grad` attributes between `backward()` and `scaler.step(optimizer)`, you should unscale them first. For example, gradient clipping manipulates a set of gradients such that their global norm (see :func:`torch.nn.utils.clip_grad_norm_`) or maximum magnitude (see :func:`torch.nn.utils.clip_grad_value_`) is < = some user-imposed threshold. If you attempted to clip *without* unscaling, the gradients' norm/maximum magnitude would also be scaled, so your requested threshold (which was meant to be the threshold for *unscaled* gradients) would be invalid.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 67); *backlink***
>
> Unknown interpreted text role "func".

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 67); *backlink***
>
> Unknown interpreted text role "func".

`scaler.unscale_(optimizer)` unscales gradients held by `optimizer`'s assigned parameters. If your model or models contain other parameters that were assigned to another optimizer (say `optimizer2`), you may call `scaler.unscale_(optimizer2)`

separately to unscale those parameters' gradients as well.

Gradient clipping

Calling `scaler.unscale_(optimizer)` before clipping enables you to clip unscaled gradients as usual:

```
scaler = GradScaler()

for epoch in epochs:
    for input, target in data:
        optimizer.zero_grad()
        with autocast():
            output = model(input)
            loss = loss_fn(output, target)
        scaler.scale(loss).backward()

        # Unscales the gradients of optimizer's assigned params in-place
        scaler.unscale_(optimizer)

        # Since the gradients of optimizer's assigned params are unscaled, clips as usual:
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm)

        # optimizer's gradients are already unscaled, so scaler.step does not unscale them,
        # although it still skips optimizer.step() if the gradients contain infs or NaNs.
        scaler.step(optimizer)

        # Updates the scale for next iteration.
        scaler.update()
```

`scaler` records that `scaler.unscale_(optimizer)` was already called for this optimizer this iteration, so `scaler.step(optimizer)` knows not to redundantly unscale gradients before (internally) calling `optimizer.step()`.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst`, line 112)

Unknown directive type "currentmodule".

```
.. currentmodule:: torch.cuda.amp.GradScaler
```

Warning

:meth:`unscale_<unscale_>` should only be called once per optimizer per :meth:`step<step>` call, and only after all gradients for that optimizer's assigned parameters have been accumulated. Calling :meth:`unscale_<unscale_>` twice for a given optimizer between each :meth:`step<step>` triggers a RuntimeError.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst`, line 115); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst`, line 115); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst`, line 115); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst`, line 115); *backlink*

Unknown interpreted text role "meth".

Working with Scaled Gradients

Gradient accumulation

Gradient accumulation adds gradients over an effective batch of size `batch_per_iter * iters_to_accumulate` (`* num_procs` if distributed). The scale should be calibrated for the effective batch, which means inf/NaN checking, step skipping if inf/NaN grads are found, and scale updates should occur at effective-batch granularity. Also, grads should remain scaled, and the scale factor should remain constant, while grads for a given effective batch are accumulated. If grads are unscaled (or the scale factor changes) before accumulation is complete, the next backward pass will add scaled grads to unscaled grads (or grads scaled by a different factor) after which it's impossible to recover the accumulated unscaled grads :meth:`step<step>` must apply.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst`, line 126); *backlink*
>
> Unknown interpreted text role "meth".

Therefore, if you want to :meth:`unscale_<unscale_>` grads (e.g., to allow clipping unscaled grads), call :meth:`unscale_<unscale_>` just before :meth:`step<step>`, after all (scaled) grads for the upcoming :meth:`step<step>` have been accumulated. Also, only call :meth:`update<update>` at the end of iterations where you called :meth:`step<step>` for a full effective batch:

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst`, line 134); *backlink*
>
> Unknown interpreted text role "meth".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst`, line 134); *backlink*
>
> Unknown interpreted text role "meth".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst`, line 134); *backlink*
>
> Unknown interpreted text role "meth".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst`, line 134); *backlink*
>
> Unknown interpreted text role "meth".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst`, line 134); *backlink*
>
> Unknown interpreted text role "meth".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst`, line 134); *backlink*
>
> Unknown interpreted text role "meth".

```
scaler = GradScaler()

for epoch in epochs:
    for i, (input, target) in enumerate(data):
        with autocast():
            output = model(input)
            loss = loss_fn(output, target)
            loss = loss / iters_to_accumulate
```

```
        # Accumulates scaled gradients.
        scaler.scale(loss).backward()

        if (i + 1) % iters_to_accumulate == 0:
            # may unscale_ here if desired (e.g., to allow clipping unscaled gradients)

            scaler.step(optimizer)
            scaler.update()
            optimizer.zero_grad()
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 158)

Unknown directive type "currentmodule".

```
.. currentmodule:: torch.cuda.amp
```

Gradient penalty

A gradient penalty implementation commonly creates gradients using :func:`torch.autograd.grad`, combines them to create the penalty value, and adds the penalty value to the loss.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 163); *backlink*

Unknown interpreted text role "func".

Here's an ordinary example of an L2 penalty without gradient scaling or autocasting:

```
for epoch in epochs:
    for input, target in data:
        optimizer.zero_grad()
        output = model(input)
        loss = loss_fn(output, target)

        # Creates gradients
        grad_params = torch.autograd.grad(outputs=loss,
                                          inputs=model.parameters(),
                                          create_graph=True)

        # Computes the penalty term and adds it to the loss
        grad_norm = 0
        for grad in grad_params:
            grad_norm += grad.pow(2).sum()
        grad_norm = grad_norm.sqrt()
        loss = loss + grad_norm

        loss.backward()

        # clip gradients here, if desired

        optimizer.step()
```

To implement a gradient penalty *with* gradient scaling, the `outputs` Tensor(s) passed to :func:`torch.autograd.grad` should be scaled. The resulting gradients will therefore be scaled, and should be unscaled before being combined to create the penalty value.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 193); *backlink*

Unknown interpreted text role "func".

Also, the penalty term computation is part of the forward pass, and therefore should be inside an :class:`autocast` context.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 198); *backlink*

Unknown interpreted text role "class".

Here's how that looks for the same L2 penalty:

```
scaler = GradScaler()

for epoch in epochs:
    for input, target in data:
        optimizer.zero_grad()
        with autocast():
            output = model(input)
            loss = loss_fn(output, target)

        # Scales the loss for autograd.grad's backward pass, producing scaled_grad_params
        scaled_grad_params = torch.autograd.grad(outputs=scaler.scale(loss),
                                                  inputs=model.parameters(),
                                                  create_graph=True)

        # Creates unscaled grad_params before computing the penalty. scaled_grad_params are
        # not owned by any optimizer, so ordinary division is used instead of scaler.unscale_:
        inv_scale = 1./scaler.get_scale()
        grad_params = [p * inv_scale for p in scaled_grad_params]

        # Computes the penalty term and adds it to the loss
        with autocast():
            grad_norm = 0
            for grad in grad_params:
                grad_norm += grad.pow(2).sum()
            grad_norm = grad_norm.sqrt()
            loss = loss + grad_norm

        # Applies scaling to the backward call as usual.
        # Accumulates leaf gradients that are correctly scaled.
        scaler.scale(loss).backward()

        # may unscale_ here if desired (e.g., to allow clipping unscaled gradients)

        # step() and update() proceed as usual.
        scaler.step(optimizer)
        scaler.update()
```

Working with Multiple Models, Losses, and Optimizers

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst`, line 244)

Unknown directive type "currentmodule".

```
.. currentmodule:: torch.cuda.amp.GradScaler
```

If your network has multiple losses, you must call :meth:`scaler.scale<scale>` on each of them individually. If your network has multiple optimizers, you may call :meth:`scaler.unscale_<unscale_>` on any of them individually, and you must call :meth:`scaler.step<step>` on each of them individually.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst`, line 246); ***backlink***

Unknown interpreted text role "meth".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst`, line 246); ***backlink***

Unknown interpreted text role "meth".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst`, line 246); ***backlink***

Unknown interpreted text role "meth".

However, :meth:`scaler.update<update>` should only be called once, after all optimizers used this iteration have been stepped:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 250); backlink

Unknown interpreted text role "meth".

```
scaler = torch.cuda.amp.GradScaler()

for epoch in epochs:
    for input, target in data:
        optimizer0.zero_grad()
        optimizer1.zero_grad()
        with autocast():
            output0 = model0(input)
            output1 = model1(input)
            loss0 = loss_fn(2 * output0 + 3 * output1, target)
            loss1 = loss_fn(3 * output0 - 5 * output1, target)

        # (retain_graph here is unrelated to amp, it's present because in this
        # example, both backward() calls share some sections of graph.)
        scaler.scale(loss0).backward(retain_graph=True)
        scaler.scale(loss1).backward()

        # You can choose which optimizers receive explicit unscaling, if you
        # want to inspect or modify the gradients of the params they own.
        scaler.unscale_(optimizer0)

        scaler.step(optimizer0)
        scaler.step(optimizer1)

        scaler.update()
```

Each optimizer checks its gradients for infs/NaNs and makes an independent decision whether or not to skip the step. This may result in one optimizer skipping the step while the other one does not. Since step skipping occurs rarely (every several hundred iterations) this should not impede convergence. If you observe poor convergence after adding gradient scaling to a multiple-optimizer model, please report a bug.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 285)

Unknown directive type "currentmodule".

```
.. currentmodule:: torch.cuda.amp
```

Working with Multiple GPUs

The issues described here only affect :class:`autocast`. :class:`GradScaler`'s usage is unchanged.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 292); backlink

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 292); backlink

Unknown interpreted text role "class".

DataParallel in a single process

Even if :class:`torch.nn.DataParallel` spawns threads to run the forward pass on each device. The autocast state is propagated in each one and the following will work:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 299); backlink

Unknown interpreted text role "class".

```
model = MyModel()
```

```
dp_model = nn.DataParallel(model)

# Sets autocast in the main thread
with autocast():
    # dp_model's internal threads will autocast.
    output = dp_model(input)
    # loss_fn also autocast
    loss = loss_fn(output)
```

DistributedDataParallel, one GPU per process

:class:`torch.nn.parallel.DistributedDataParallel`'s documentation recommends one GPU per process for best performance. In this case, `DistributedDataParallel` does not spawn threads internally, so usages of :class:`autocast` and :class:`GradScaler` are not affected.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 315); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 315); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 315); *backlink*

Unknown interpreted text role "class".

DistributedDataParallel, multiple GPUs per process

Here :class:`torch.nn.parallel.DistributedDataParallel` may spawn a side thread to run the forward pass on each device, like :class:`torch.nn.DataParallel`. :ref:`The fix is the same<amp-dataparallel>`: apply autocast as part of your model's `forward` method to ensure it's enabled in side threads.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 322); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 322); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 322); *backlink*

Unknown interpreted text role "ref".

Autocast and Custom Autograd Functions

If your network uses :ref:`custom autograd functions<extending-autograd>` (subclasses of :class:`torch.autograd.Function`), changes are required for autocast compatibility if any function

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 331); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 331); *backlink*

Unknown interpreted text role "class".

- takes multiple floating-point Tensor inputs,
- wraps any autocastable op (see the :ref:`Autocast Op Reference<autocast-op-reference>`), or

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 336); *backlink*

Unknown interpreted text role "ref".

- requires a particular `dtype` (for example, if it wraps CUDA extensions that were only compiled for `dtype`).

In all cases, if you're importing the function and can't alter its definition, a safe fallback is to disable autocast and force execution in `float32` (or `dtype`) at any points of use where errors occur:

```
with autocast():
    ...
    with autocast(enabled=False):
        output = imported_function(input1.float(), input2.float())
```

If you're the function's author (or can alter its definition) a better solution is to use the :func:`torch.cuda.amp.custom_fwd` and :func:`torch.cuda.amp.custom_bwd` decorators as shown in the relevant case below.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 349); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 349); *backlink*

Unknown interpreted text role "func".

Functions with multiple inputs or autocastable ops

Apply :func:`custom_fwd<custom_fwd>` and :func:`custom_bwd<custom_bwd>` (with no arguments) to `forward` and `backward` respectively. These ensure `forward` executes with the current autocast state and `backward` executes with the same autocast state as `forward` (which can prevent type mismatch errors):

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 356); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 356); *backlink*

Unknown interpreted text role "func".

```
class MyMM(torch.autograd.Function):
    @staticmethod
    @custom_fwd
    def forward(ctx, a, b):
        ctx.save_for_backward(a, b)
        return a.mm(b)
    @staticmethod
    @custom_bwd
    def backward(ctx, grad):
        a, b = ctx.saved_tensors
        return grad.mm(b.t()), a.t().mm(grad)
```

Now `MyMM` can be invoked anywhere, without disabling autocast or manually casting inputs:

```
mymm = MyMM.apply

with autocast():
    output = mymm(input1, input2)
```

Functions that need a particular `dtype`

Consider a custom function that requires `torch.float32` inputs. Apply :func:`custom_fwd(cast_inputs=torch.float32) <custom_fwd>` to `forward` and :func:`custom_bwd<custom_bwd>` (with no arguments) to `backward`. If `forward` runs in an autocast-enabled region, the decorators cast floating-point CUDA Tensor inputs to `float32`, and locally disable autocast during `forward` and `backward`:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 382); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]amp_examples.rst, line 382); *backlink*

Unknown interpreted text role "func".

```
class MyFloat32Func(torch.autograd.Function):
    @staticmethod
    @custom_fwd(cast_inputs=torch.float32)
    def forward(ctx, input):
        ctx.save_for_backward(input)
        ...
        return fwd_output
    @staticmethod
    @custom_bwd
    def backward(ctx, grad):
        ...
```

Now `MyFloat32Func` can be invoked anywhere, without manually disabling autocast or casting inputs:

```
func = MyFloat32Func.apply

with autocast():
    # func will run in float32, regardless of the surrounding autocast state
    output = func(input)
```