ADAM

CLASS `torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False, *, maximize=False)` [SOURCE]

Implements Adam algorithm.

$$
\begin{aligned}
&\textbf{input} : \gamma \text{ (lr)}, \beta_1, \beta_2 \text{ (betas)}, \theta_0 \text{ (params)}, f(\theta) \text{ (objective)} \\
&\qquad \lambda \text{ (weight decay)}, \ amsgrad, \ maximize \\
&\textbf{initialize} : m_0 \leftarrow 0 \text{ (first moment)}, v_0 \leftarrow 0 \text{ (second moment)}, \ \widehat{v_0}^{max} \leftarrow 0 \\
&\textbf{for } t = 1 \textbf{ to } \ldots \textbf{ do} \\
&\quad \textbf{if } maximize : \\
&\qquad g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1}) \\
&\quad \textbf{else} \\
&\qquad g_t \leftarrow \nabla_\theta f_t(\theta_{t-1}) \\
&\quad \textbf{if } \lambda \neq 0 \\
&\qquad g_t \leftarrow g_t + \lambda \theta_{t-1} \\
&\quad m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t \\
&\quad v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \\
&\quad \widehat{m_t} \leftarrow m_t / \big(1 - \beta_1^t\big) \\
&\quad \widehat{v_t} \leftarrow v_t / \big(1 - \beta_2^t\big) \\
&\quad \textbf{if } amsgrad \\
&\qquad \widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t}) \\
&\qquad \theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \big(\sqrt{\widehat{v_t}^{max}} + \epsilon\big) \\
&\quad \textbf{else} \\
&\qquad \theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \big(\sqrt{\widehat{v_t}} + \epsilon\big) \\
&\textbf{return } \theta_t
\end{aligned}
$$

For further details regarding the algorithm we refer to Adam: A Method for Stochastic Optimization.

Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float*, *optional*) – learning rate (default: 1e-3)
- **betas** (*Tuple*[*float*, *float*], *optional*) – coefficients used for computing running averages of gradient and its square (default: (0.9, 0.999))
- **eps** (*float*, *optional*) – term added to the denominator to improve numerical stability (default: 1e-8)
- **weight_decay** (*float*, *optional*) – weight decay (L2 penalty) (default: 0)
- **amsgrad** (*boolean*, *optional*) – whether to use the AMSGrad variant of this algorithm from the paper On the Convergence of Adam and Beyond (default: False)
- **maximize** (*bool*, *optional*) – maximize the params based on the objective, instead of minimizing (default: False)

`add_param_group(param_group)`

Add a param group to the `Optimizer` s *param_groups*.

This can be useful when fine tuning a pre-trained network as frozen layers can be made trainable and added to the `Optimizer` as training progresses.

Parameters

param_group (*dict*) – Specifies what Tensors should be optimized along with group specific optimization options.

`load_state_dict(state_dict)`

Loads the optimizer state.

Parameters

state_dict (*dict*) – optimizer state. Should be an object returned from a call to `state_dict()` .

`state_dict()`

Returns the state of the optimizer as a `dict` .

It contains two entries:

- state - a dict holding current optimization state. Its content

  differs between optimizer classes.

- param_groups - a list containing all parameter groups where each

  parameter group is a dict

`step(closure=None)` [SOURCE]

Performs a single optimization step.

Parameters

closure (*callable*, *optional*) – A closure that reevaluates the model and returns the loss.

`zero_grad(set_to_none=False)`

Sets the gradients of all optimized `torch.Tensor` s to zero.

Parameters

set_to_none (*bool*) – instead of setting to zero, set the grads to None. This will in general have lower memory footprint, and can modestly improve performance. However, it changes certain behaviors. For example: 1. When the user tries to access a gradient and perform manual ops on it, a None attribute or a Tensor full of 0s will behave differently. 2. If the user requests `zero_grad(set_to_none=True)` followed by a backward pass, `.grad` s are guaranteed to be None for params that did not receive a gradient. 3. `torch.optim` optimizers have a different behavior if the gradient is 0 or None (in one case it does the step with a gradient of 0 and in the other it skips the step altogether).

‹ Previous Next ›