Stochastic Gradient Descent

COMP4551

Binhang Yuan

Empirical Risk

Define the Empirical Risk

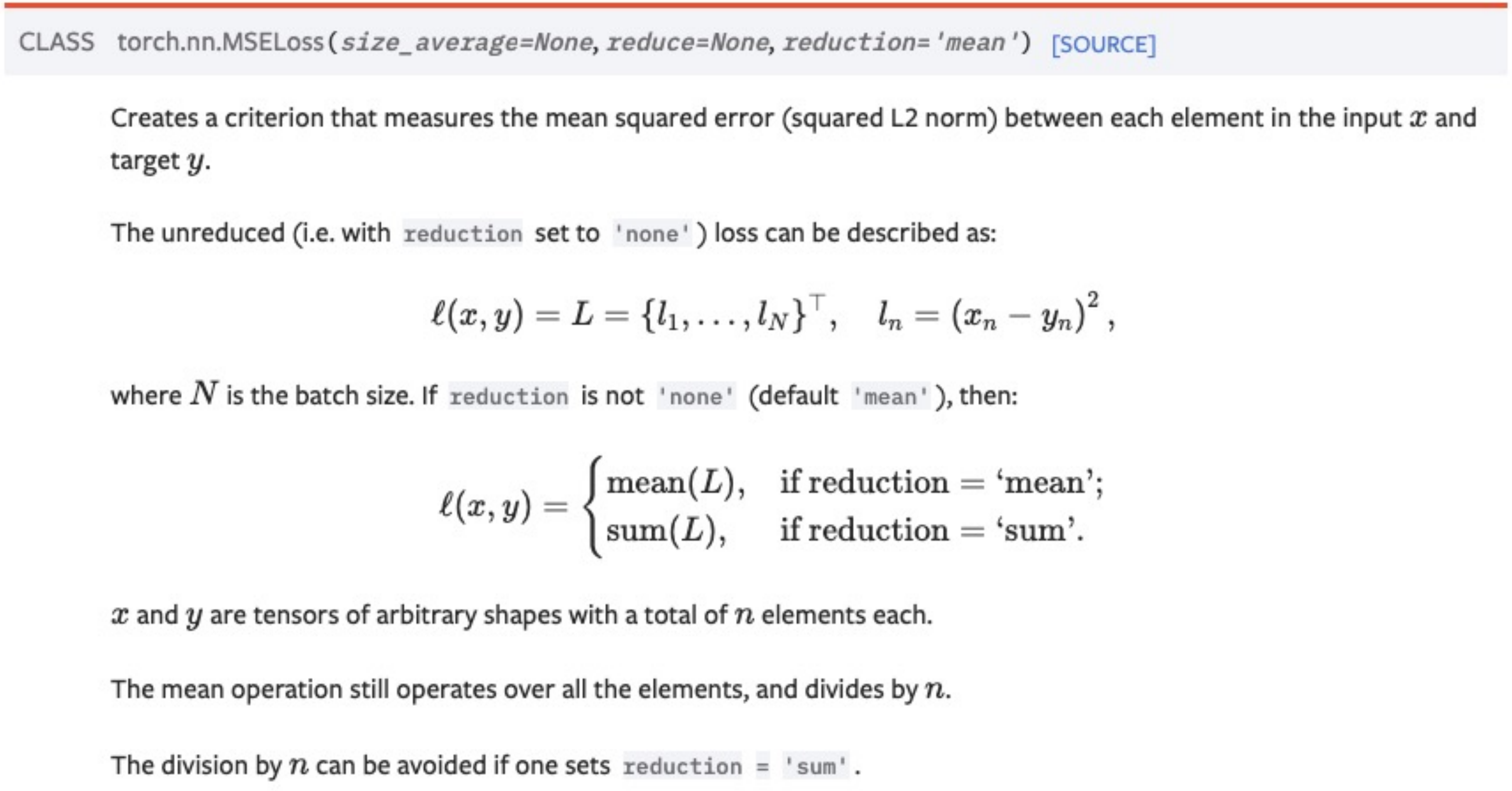
- Suppose we have:
  - a dataset $\mathcal{D} = \{(x_1, y_1), (x_1, y_2), \dots, (x_N, y_N)\}$, where
  - $x_i \in \mathcal{X}$ is the input and
  - $y_i \in \mathcal{Y}$ is the output.
  - Let $h: \mathcal{X} \rightarrow \mathcal{Y}$ be a hypothesized model (mapping from input to output) we are trying to evaluate, which is parameterized by $w \in \mathbb{R}^d$.
  - Let $L: \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R}_+$ be a non-negative loss function which measures how different two outputs are
- The emprical risk R is defined as:

$$R(h_w) = \frac{1}{N} \sum_{i=1}^{N} L(h_w(x_i), y_i)$$

Common Loss Functions

- Mean squared error loss (L2 loss).
- L1 loss.
- Negative log-likelihood loss.
- Cross entropy loss.
- KL divergence loss.

Mean Squared Error Loss



CLASS torch.nn.MSELoss(*size_average=None*, *reduce=None*, *reduction='mean'*) [SOURCE]

Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input x and target y.

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = (x_n - y_n)^2,$$

where N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then:

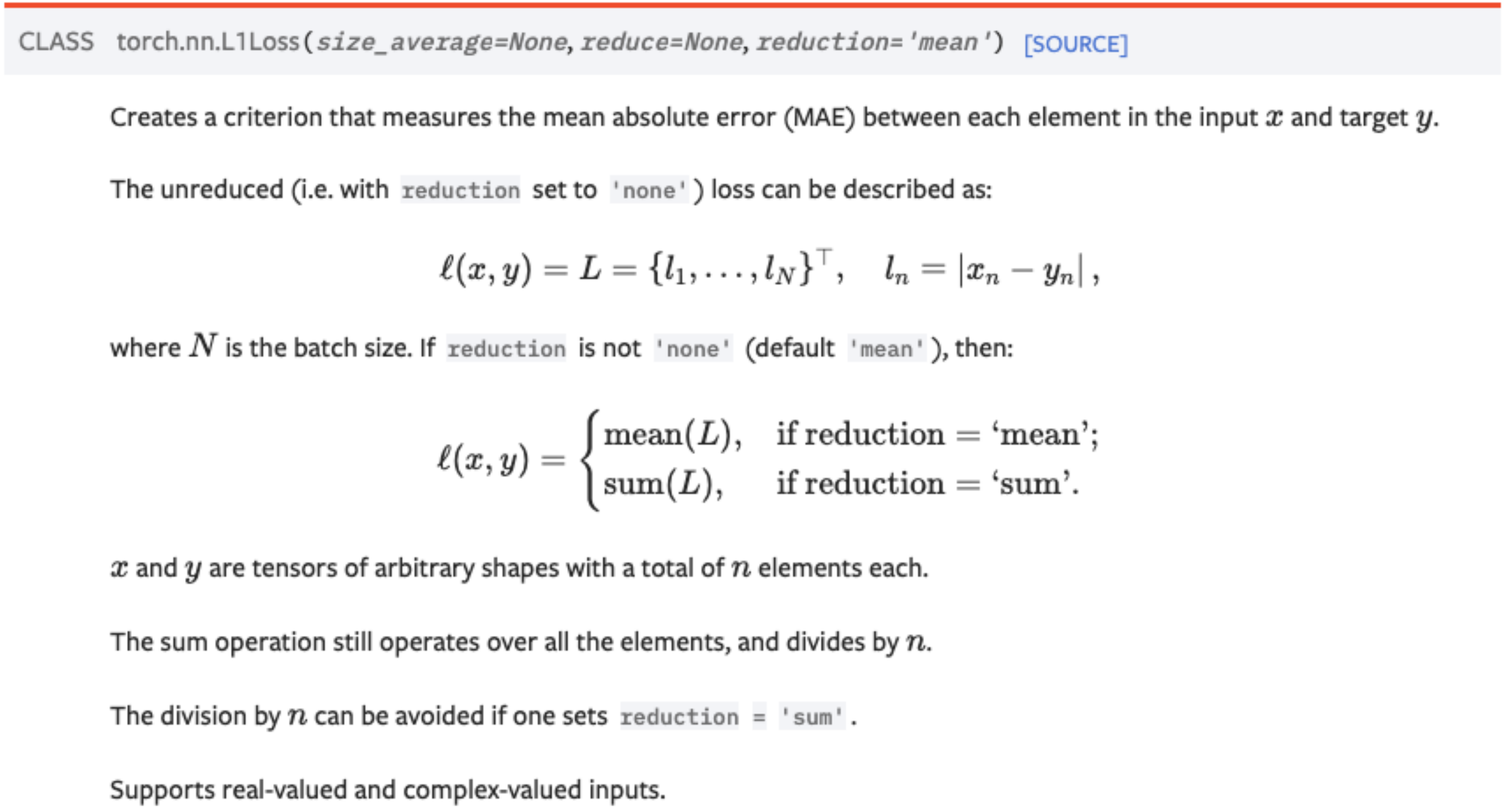
$$\ell(x, y) = \begin{cases} \operatorname{mean}(L), & \text{if reduction} = \text{`mean';} \\ \operatorname{sum}(L), & \text{if reduction} = \text{`sum'.} \end{cases}$$

x and y are tensors of arbitrary shapes with a total of n elements each.

The mean operation still operates over all the elements, and divides by n.

The division by n can be avoided if one sets `reduction = 'sum'`.

L1 Loss



CLASS torch.nn.L1Loss(*size_average=None*, *reduce=None*, *reduction='mean'*) [SOURCE]

Creates a criterion that measures the mean absolute error (MAE) between each element in the input x and target y.

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = \left| x_n - y_n \right|,$$

where N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then:

$$\ell(x, y) = \begin{cases} \operatorname{mean}(L), & \text{if reduction} = \text{`mean';} \\ \operatorname{sum}(L), & \text{if reduction} = \text{`sum'.} \end{cases}$$

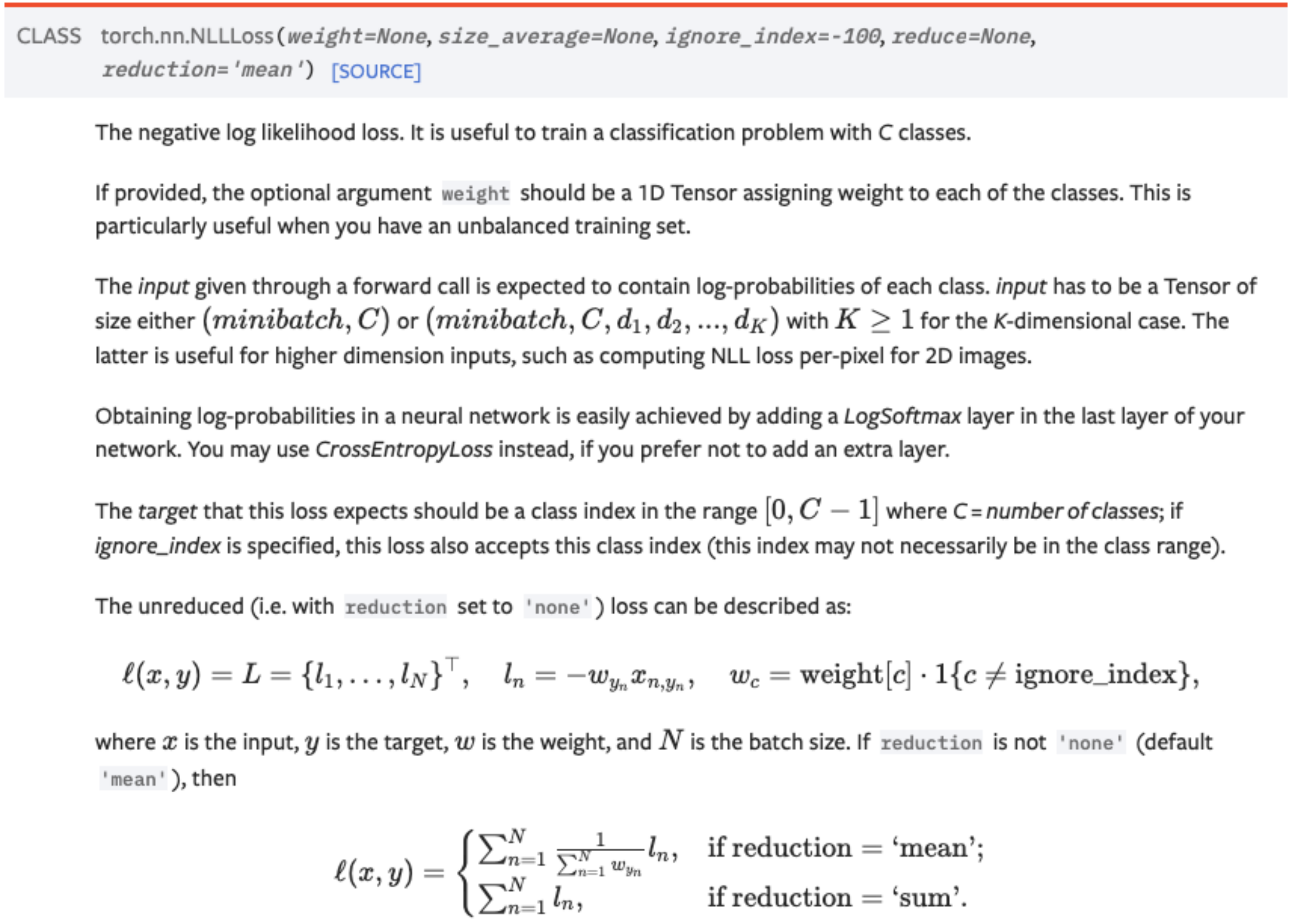
x and y are tensors of arbitrary shapes with a total of n elements each.

The sum operation still operates over all the elements, and divides by n.

The division by n can be avoided if one sets `reduction = 'sum'`.

Supports real-valued and complex-valued inputs.

Negative Log-likelihood Loss



CLASS torch.nn.NLLLoss(*weight=None*, *size_average=None*, *ignore_index=-100*, *reduce=None*, *reduction='mean'*) [SOURCE]

The negative log likelihood loss. It is useful to train a classification problem with *C* classes.

If provided, the optional argument `weight` should be a 1D Tensor assigning weight to each of the classes. This is particularly useful when you have an unbalanced training set.

The *input* given through a forward call is expected to contain log-probabilities of each class. *input* has to be a Tensor of size either $(minibatch, C)$ or $(minibatch, C, d_1, d_2, ..., d_K)$ with $K \geq 1$ for the *K*-dimensional case. The latter is useful for higher dimension inputs, such as computing NLL loss per-pixel for 2D images.

Obtaining log-probabilities in a neural network is easily achieved by adding a *LogSoftmax* layer in the last layer of your network. You may use *CrossEntropyLoss* instead, if you prefer not to add an extra layer.

The *target* that this loss expects should be a class index in the range $[0, C-1]$ where C = *number of classes*; if *ignore_index* is specified, this loss also accepts this class index (this index may not necessarily be in the class range).

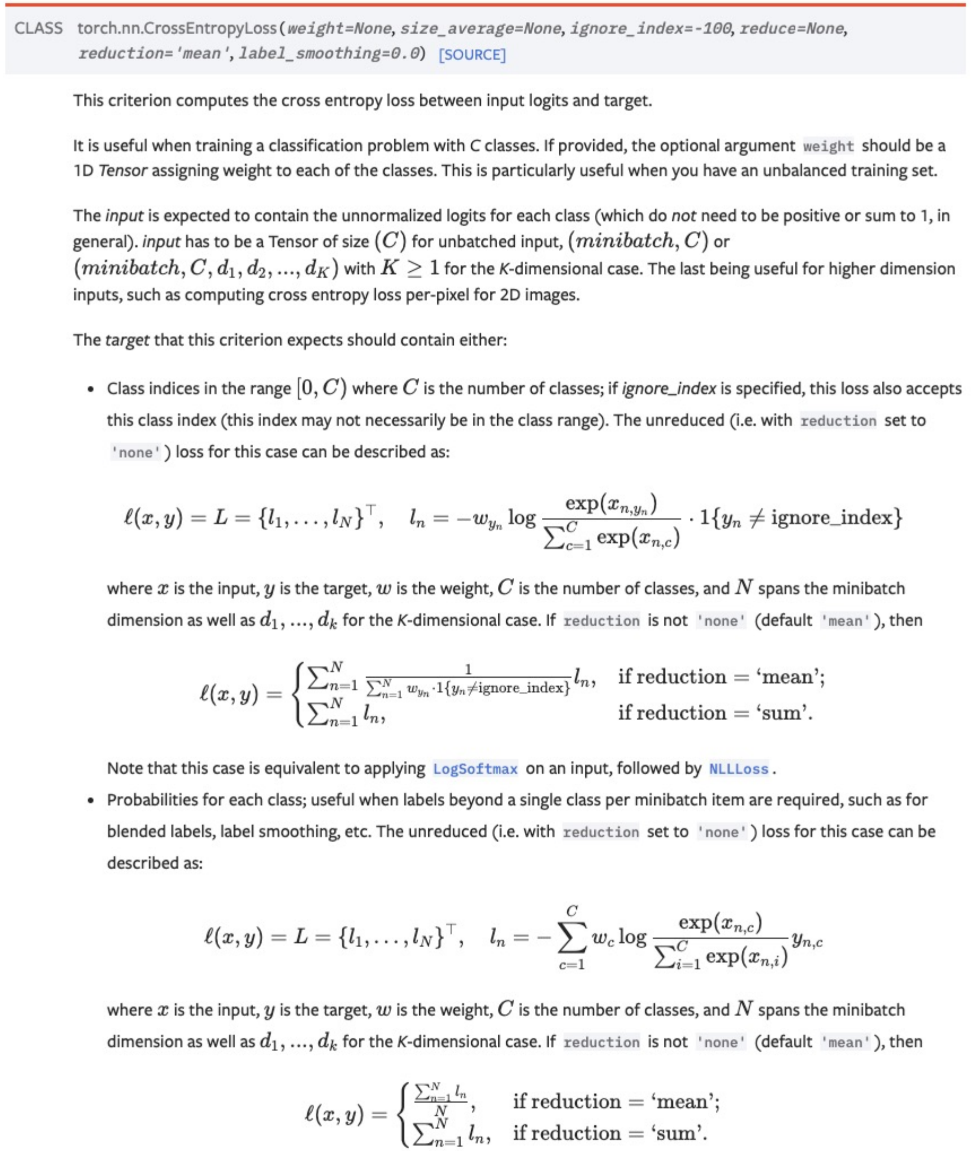
The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_{y_n} x_{n, y_n}, \quad w_c = \text{weight}[c] \cdot 1\{c \neq \text{ignore\_index}\},$$

where x is the input, y is the target, w is the weight, and N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then

$$\ell(x, y) = \begin{cases} \sum_{n=1}^N \frac{1}{\sum_{n=1}^N w_{y_n}} l_n, & \text{if reduction} = \text{'mean';} \\ \sum_{n=1}^N l_n, & \text{if reduction} = \text{'sum'.} \end{cases}$$

Cross Entropy Loss



CLASS torch.nn.CrossEntropyLoss(*weight=None*, *size_average=None*, *ignore_index=-100*, *reduce=None*, *reduction='mean'*, *label_smoothing=0.0*) [SOURCE]

This criterion computes the cross entropy loss between input logits and target.

It is useful when training a classification problem with C classes. If provided, the optional argument `weight` should be a 1D *Tensor* assigning weight to each of the classes. This is particularly useful when you have an unbalanced training set.

The *input* is expected to contain the unnormalized logits for each class (which do *not* need to be positive or sum to 1, in general). *input* has to be a Tensor of size (C) for unbatched input, $(minibatch, C)$ or $(minibatch, C, d_1, d_2, ..., d_K)$ with $K \geq 1$ for the *K*-dimensional case. The last being useful for higher dimension inputs, such as computing cross entropy loss per-pixel for 2D images.

The *target* that this criterion expects should contain either:

- Class indices in the range $[0, C)$ where C is the number of classes; if *ignore_index* is specified, this loss also accepts this class index (this index may not necessarily be in the class range). The unreduced (i.e. with `reduction` set to `'none'`) loss for this case can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_{y_n} \log \frac{\exp(x_{n,y_n})}{\sum_{c=1}^{C} \exp(x_{n,c})} \cdot 1\{y_n \neq \text{ignore\_index}\}$$

where x is the input, y is the target, w is the weight, C is the number of classes, and N spans the minibatch dimension as well as $d_1, ..., d_k$ for the *K*-dimensional case. If `reduction` is not `'none'` (default `'mean'`), then

$$\ell(x, y) = \begin{cases} \sum_{n=1}^{N} \frac{1}{\sum_{n=1}^{N} w_{y_n} \cdot 1\{y_n \neq \text{ignore\_index}\}} l_n, & \text{if reduction} = \text{`mean';} \\ \sum_{n=1}^{N} l_n, & \text{if reduction} = \text{`sum'.} \end{cases}$$

Note that this case is equivalent to applying `LogSoftmax` on an input, followed by `NLLLoss`.

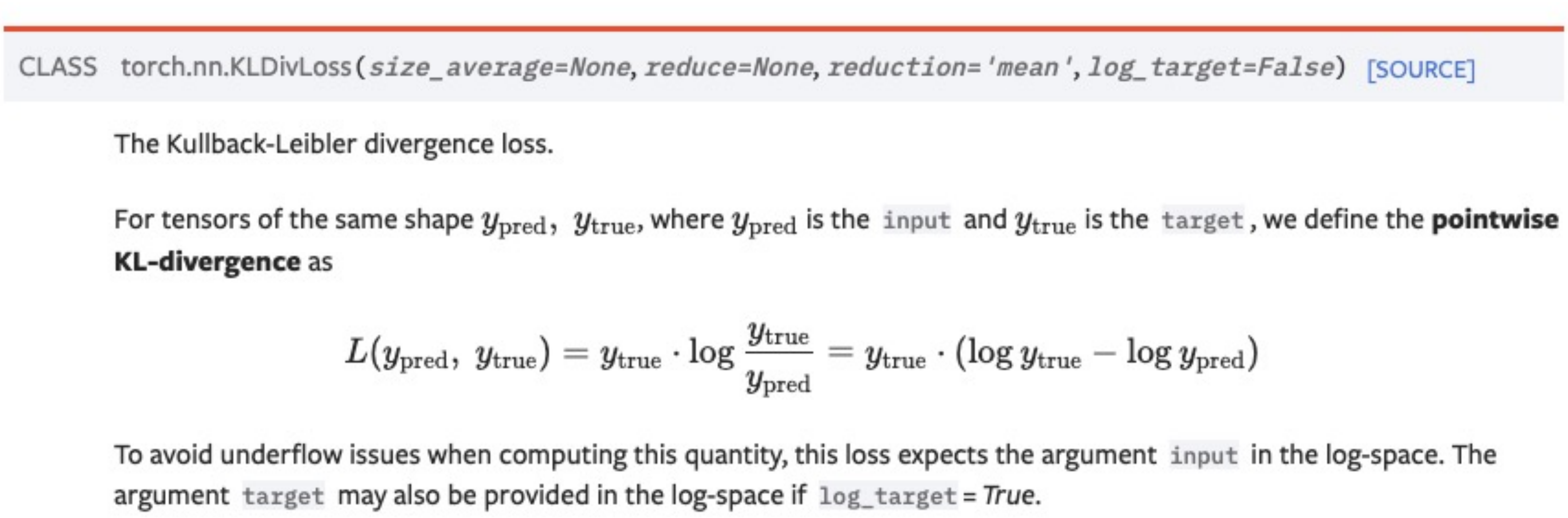
- Probabilities for each class; useful when labels beyond a single class per minibatch item are required, such as for blended labels, label smoothing, etc. The unreduced (i.e. with `reduction` set to `'none'`) loss for this case can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -\sum_{c=1}^{C} w_c \log \frac{\exp(x_{n,c})}{\sum_{i=1}^{C} \exp(x_{n,i})} y_{n,c}$$

where x is the input, y is the target, w is the weight, C is the number of classes, and N spans the minibatch dimension as well as $d_1, ..., d_k$ for the *K*-dimensional case. If `reduction` is not `'none'` (default `'mean'`), then

$$\ell(x, y) = \begin{cases} \frac{\sum_{n=1}^{N} l_n}{N}, & \text{if reduction} = \text{`mean';} \\ \sum_{n=1}^{N} l_n, & \text{if reduction} = \text{`sum'.} \end{cases}$$

KL Divergence Loss



CLASS torch.nn.KLDivLoss(*size_average=None*, *reduce=None*, *reduction='mean'*, *log_target=False*) [SOURCE]

The Kullback-Leibler divergence loss.

For tensors of the same shape $y_{\text{pred}},\ y_{\text{true}}$, where y_{pred} is the `input` and y_{true} is the `target`, we define the **pointwise KL-divergence** as

$$L(y_{\text{pred}},\ y_{\text{true}}) = y_{\text{true}} \cdot \log \frac{y_{\text{true}}}{y_{\text{pred}}} = y_{\text{true}} \cdot (\log y_{\text{true}} - \log y_{\text{pred}})$$

To avoid underflow issues when computing this quantity, this loss expects the argument `input` in the log-space. The argument `target` may also be provided in the log-space if `log_target` = *True*.

Computational Cost of the Emprical Risk

- The number of training examples N, the cost will be proportional to N.
- The cost to compute the loss function L.
- The cost to evaluate the hypothesis h_w.

Minimize the Emprical Risk

- Don't just want to calculate the empirical risk;
- Let $f\colon \mathbb{R}^d \rightarrow \mathbb{R}_+$ be the optimization object, which is formulated by the empirical risk;
- Let $\mathcal{D} = \{(x_1, y_1), (x_1, y_2), \dots, (x_N, y_N)\} = \{\xi_1, \xi_2, \dots, \xi_N\}$ be the training set;
- **<u>The training computation is solving the following optimization problem:</u>**

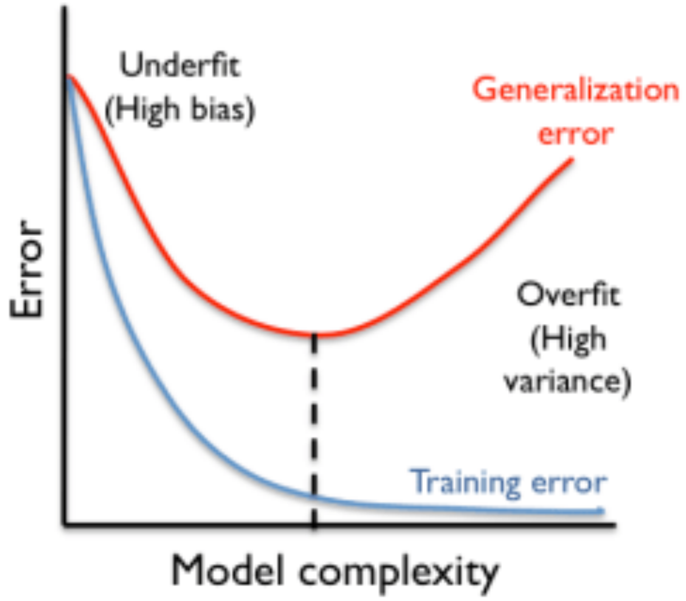
$$mimimize\colon\ R(h_w) = \frac{1}{N}\sum_{i=1}^{N} L(h_w(x_i), y_i) = f(w) = \frac{1}{N}\sum_{i=1}^{N} f(w; \xi_i)$$
$$over\ w \in \mathbb{R}^d$$

Generalization and Regularization

- Goal: reduce *generalization error* (test error) by constraining model complexity, often by discouraging very large weights.
- A very common approach is to modify the training objective:

$$mimimize: \; R(h_w) = \frac{1}{N}\sum_{i=1}^{N} f(\boldsymbol{w}; \xi_i) + \lambda\Omega(\mathbf{w})$$

$$over \; \boldsymbol{w} \in \mathbb{R}^d$$



- $\Omega(\mathrm{w})$ is a penalty term:
  - L2 norm: $\Omega(\boldsymbol{w}) = \|\boldsymbol{w}\|_2 = \left[\sum_{i=1}^{m} w_i^2\right]^{\frac{1}{2}}$
  - L1 norm: $\Omega(\boldsymbol{w}) = \|\boldsymbol{w}\|_1 = \sum_{i=1}^{m} |w_i|$
- λ is a hyperparameter to control regularization strength.

Gradient Descent

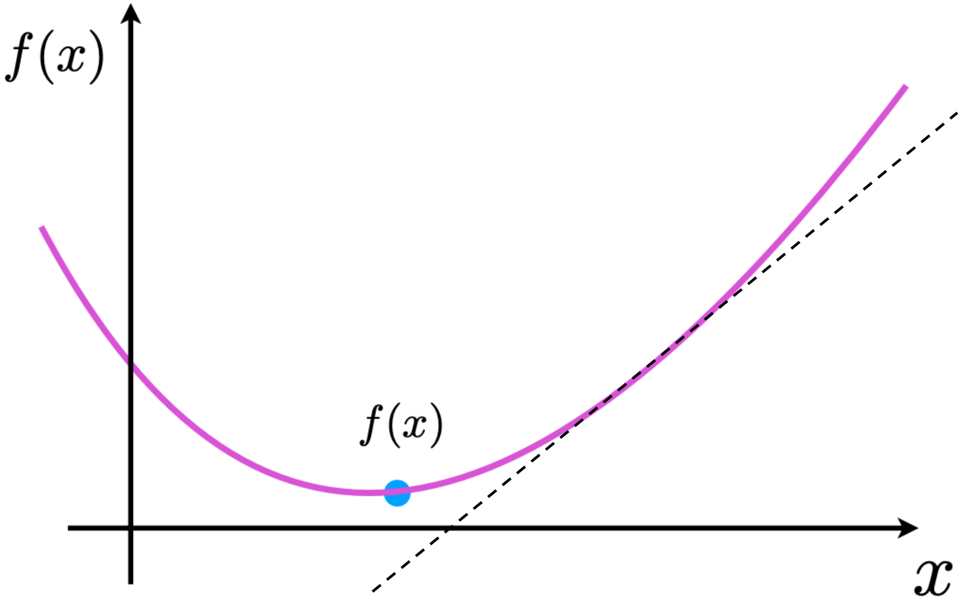
Gradient Descent Algorithm

- Suppose we have:
  - w_0 denotes the value of the initalized parameter;
  - w_t denotes the value of the parameter at iteration t;
  - $\alpha_t \in \mathbb{R}$ denotes the learning rate at iteration t;
  - ∇f denotes the gradient (*vector of partial derivatives*) of function f.
- **The gradient decent algorithm is defined by**:

$$w_{t+1} = w_t - \alpha_t \cdot \nabla f(w_t) = w_t - \alpha_t \cdot \frac{1}{N} \sum_{i=1}^{N} \nabla f(w_t; \xi_i)$$

Definition of a Derivative

- First, suppose we have:
  - $f: \mathbb{R} \to \mathbb{R};$
- Definition of a derivative:
  - $f'(x) = \frac{\partial f}{\partial x} = \lim_{\epsilon \to 0} \frac{f(x+\epsilon)-f(x)}{\epsilon}$

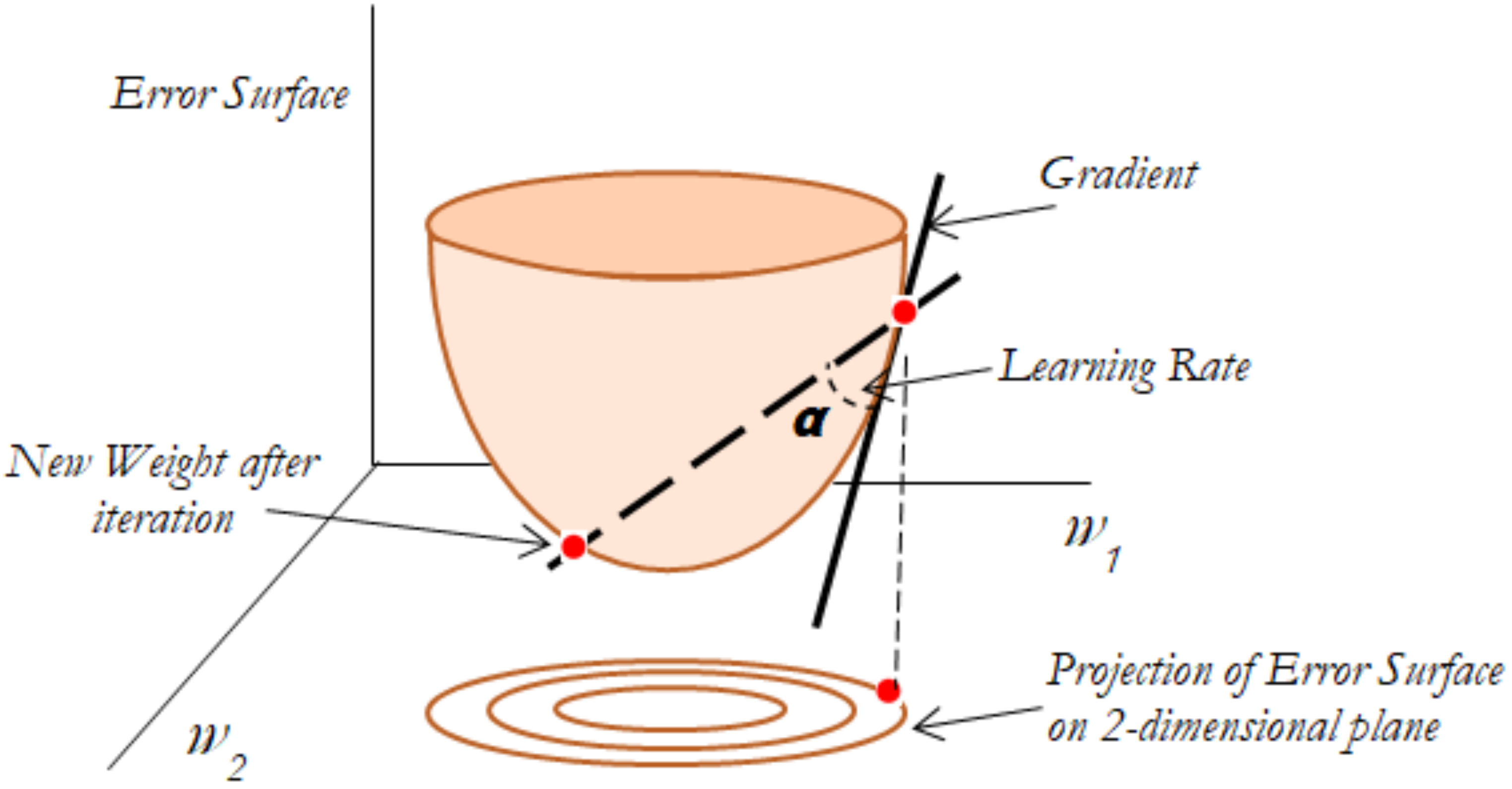


Definition of a Derivative

- Then, suppose we have:
  - $f: \mathbb{R}^d \to \mathbb{R}$;
- Definition of a derivative/gradient :
  - $\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \cdots \\ \frac{\partial f}{\partial x_d} \end{bmatrix} \in \mathbb{R}^d$
- Where:
  - $\frac{\partial f}{\partial x_i} = \lim_{\epsilon \to 0} \frac{f(x_1,x_2,\ldots,x_i+\epsilon,x_{i+1},\ldots,x_d) - f(x_1,x_2,\ldots,x_i,x_{i+1},\ldots,x_d)}{\epsilon} = \lim_{\epsilon \to 0} \frac{f(x+\epsilon e_i) - f(x)}{\epsilon}$

Why Does Gradient Descent Work?

- **Intuition:**
  - If the learning rate is small enough and the value of the gradient is nonzero;
  - Gradient descent decreases the value of the objective at each iteration;
  - Eventually, gradient descent comes close to a point where the gradient is zero.


Error Surface
Gradient
Learning Rate
α
New Weight after iteration
w_1
w_2
Projection of Error Surface on 2-dimensional plane

Stochastic Gradient Descent

Basic Idea

- Calculating the gradient over the whole dataset is computationally expensive!
  - LLM pretraining corpus can include trillions of tokens!
- How to reduce this cost?
  - Replace the full gradient (which is a sum) with ***a single gradient example***.
  - Iteratively by sampling a random example ξ_t uniformly from the training set and then updating the w_t.

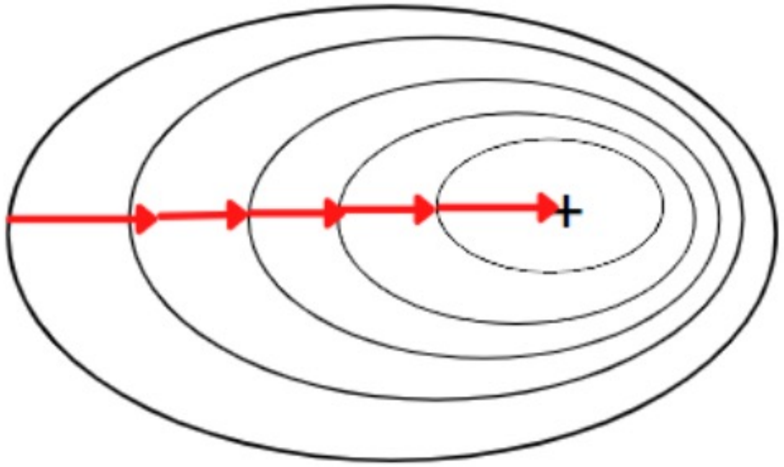
$$w_{t+1} = w_t - \alpha_t \cdot \cancel{\nabla f(w_t)}$$

$$\Big\downarrow$$

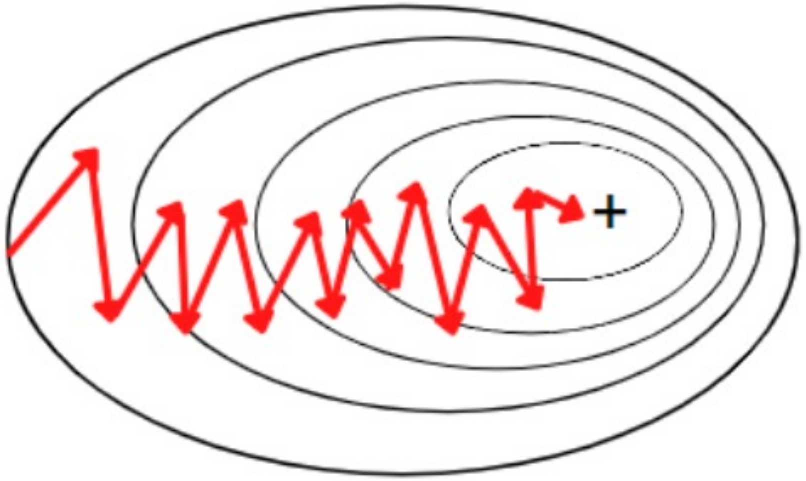
$$w_{t+1} = w_t - \alpha_t \cdot \nabla f(w_t; \xi_t)$$

Stochastic Gradient Descent

- Stochastic gradient descent won't necessarily decrease the total loss at every iteration!
  - But it runs much faster!
- Why is it fine to get an approximate solution for training?
  - In machine learning, generalization matters more than optimization.



Gradient Descent



Stochastic Gradient Descent

Mini-Batch Stochastic Gradient Descent

- Basic ideas:
  - To reduce the variance of stochastic gradients;
  - Split the training data into smaller batches;
  - Sampling batch (usually without replacement)
- Suppose we have:
  - B is the batch size;
  - Replace the single gradient with a batch of gradients:

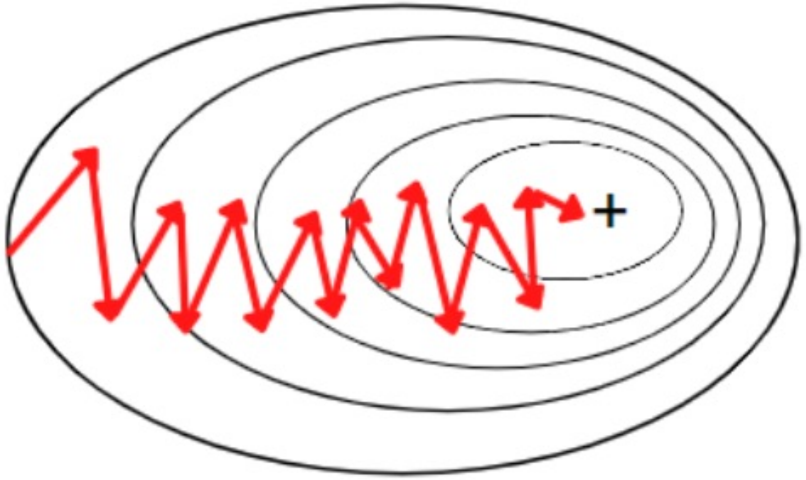
$$w_{t+1} = w_t - \alpha_t \cdot \cancel{\nabla f(w_t, \xi_t)}$$

⬇

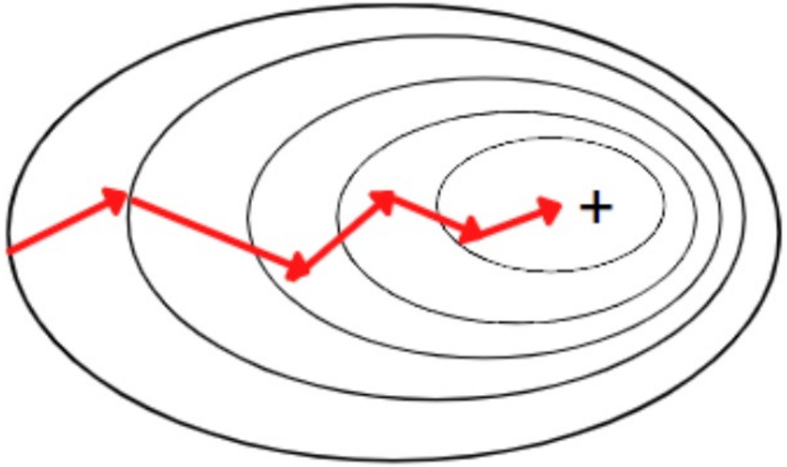
$$w_{t+1} = w_t - \alpha_t \cdot \frac{1}{\mathrm{B}} \sum_{i=1}^{B} \nabla f(w_t; \xi_i)$$

Mini-Batch Stochastic Gradient Descent

- Mini-batch stochastic gradient descent reduces the variance of stochastic gradients!



Stochastic Gradient Descent



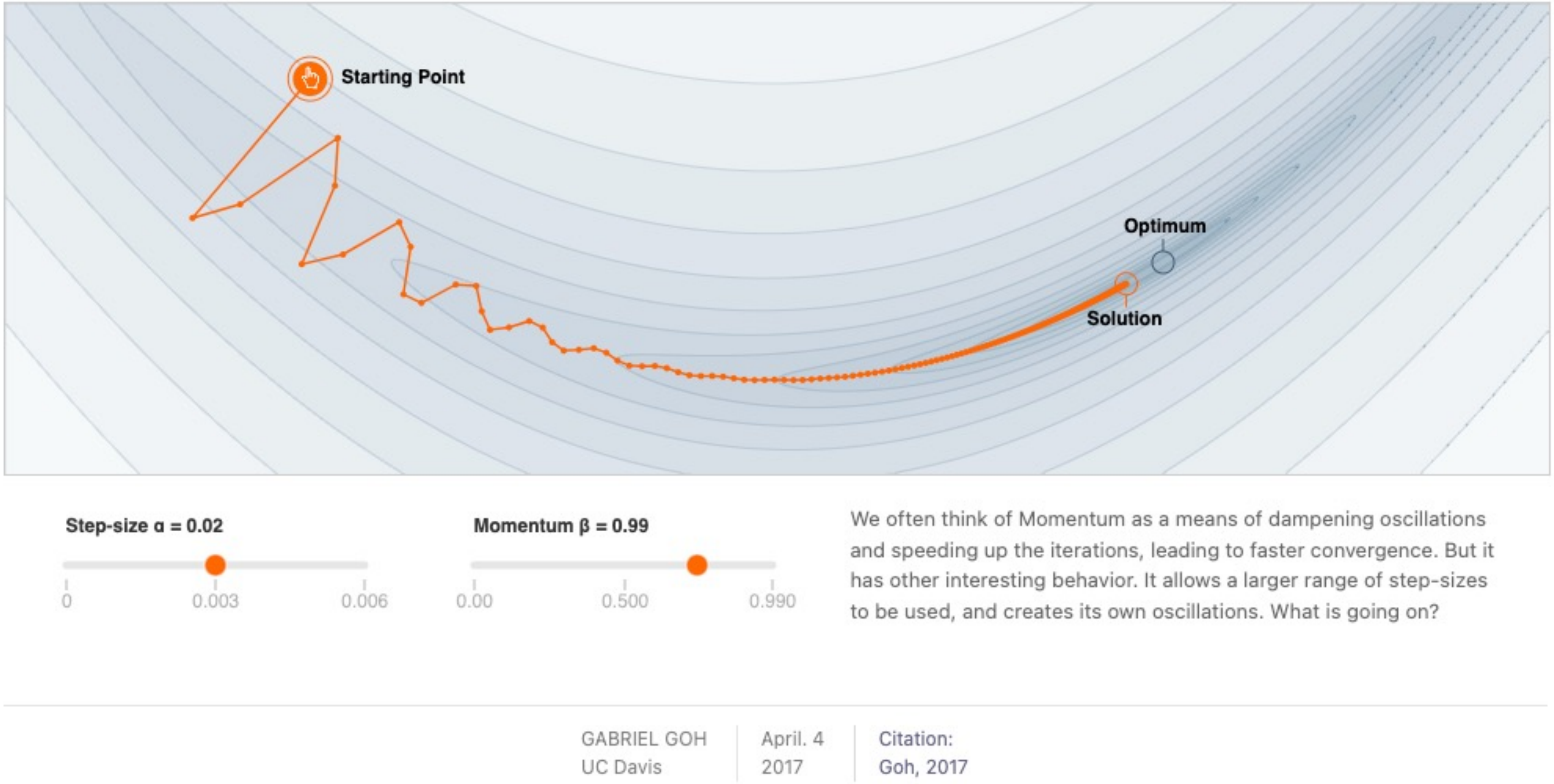
Mini-Batch Stochastic Gradient Descent

Acceleration of SGD 1: (Polyak's)Momentum

- Basic idea:
  - In SGD or mini-batch SGD the updates at each step is only based on current gradients, which can be unstable.
  - Momentum: exponentially weighted average of gradients.
  - The moving average method should be able to *denoise* the gradients computed at each step.
- Formal equation:

$$w_{t+1} = \underbrace{w_t - \alpha_t \cdot \frac{1}{\mathrm{B}} \sum_{i=1}^{B} \nabla f(w_t; \xi_i)}_{\text{Standard gradient step}} + \underbrace{\beta(w_t - w_{t-1})}_{\text{Momentum step}}$$

Why Momentum Really Works?



Step-size α = 0.02

Momentum β = 0.99

We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

GABRIEL GOH
UC Davis

April. 4
2017

Citation:
Goh, 2017

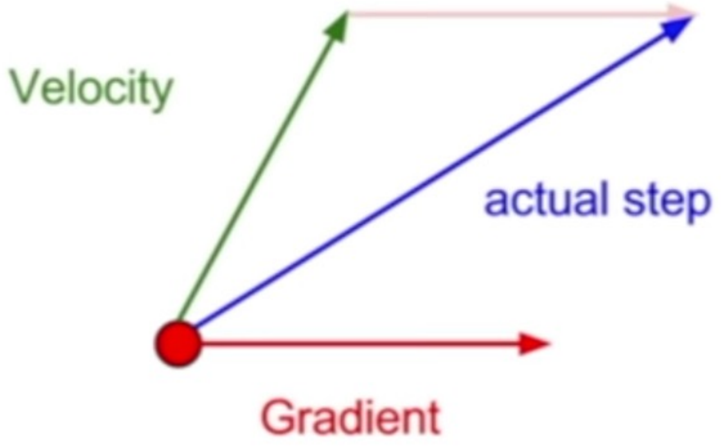
https://distill.pub/2017/momentum/

Acceleration of SGD 2: (Nesterov's) Momentum

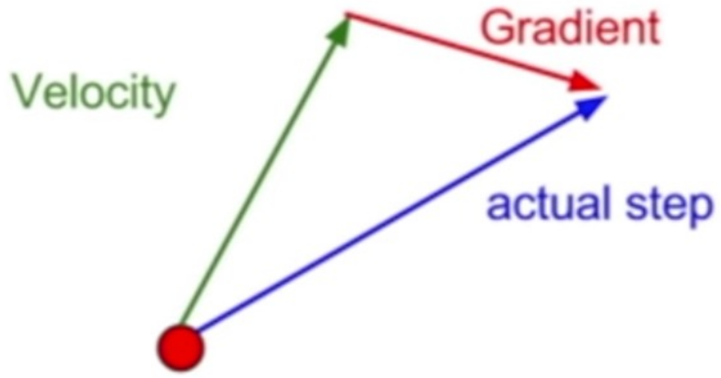
- Basic idea:
  - Polyak's momentum algorithm can fail to converge for some carefully built convex optimization problems.
  - Nesterov's Momentum: evaluates the gradient after applying momentum (at a point closer to the minimum point).
  - Works better for some cases in practice.
- Formal equation:

$$w_{t+1} = w_t - \underbrace{\alpha_t \cdot \frac{1}{\mathrm{B}} \sum_{i=1}^{B} \nabla f\left(w_t + \beta(w_t - w_{t-1}); \xi_i\right)}_{\text{Estimate gradient after applying momentum}} + \underbrace{\beta(w_t - w_{t-1})}_{\text{Momentum step}}$$

Polyak's Momentum vs. Nesterov's Momentum

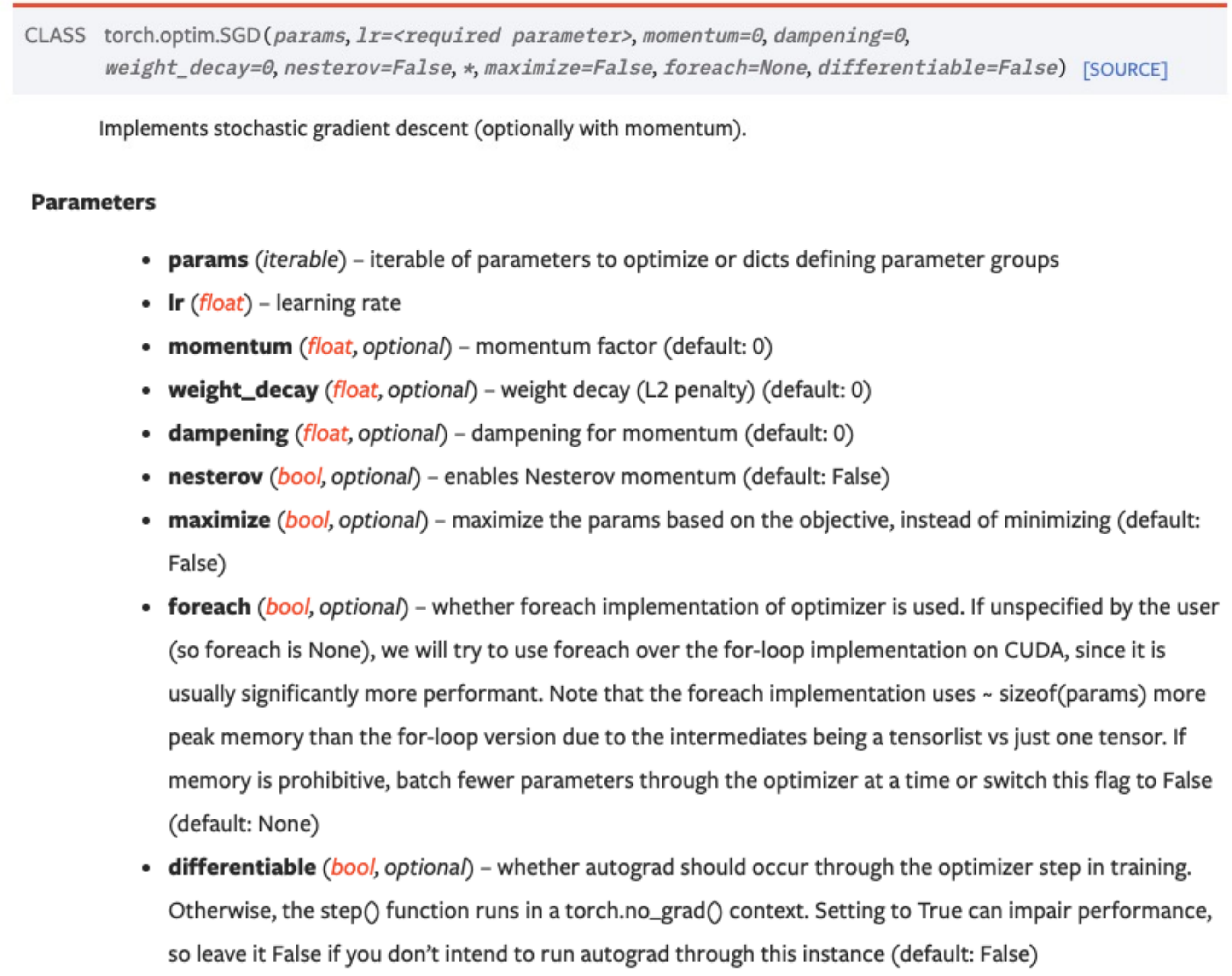


Polyak's Momentum



Nesterov's Momentum

SGD in PyTorch

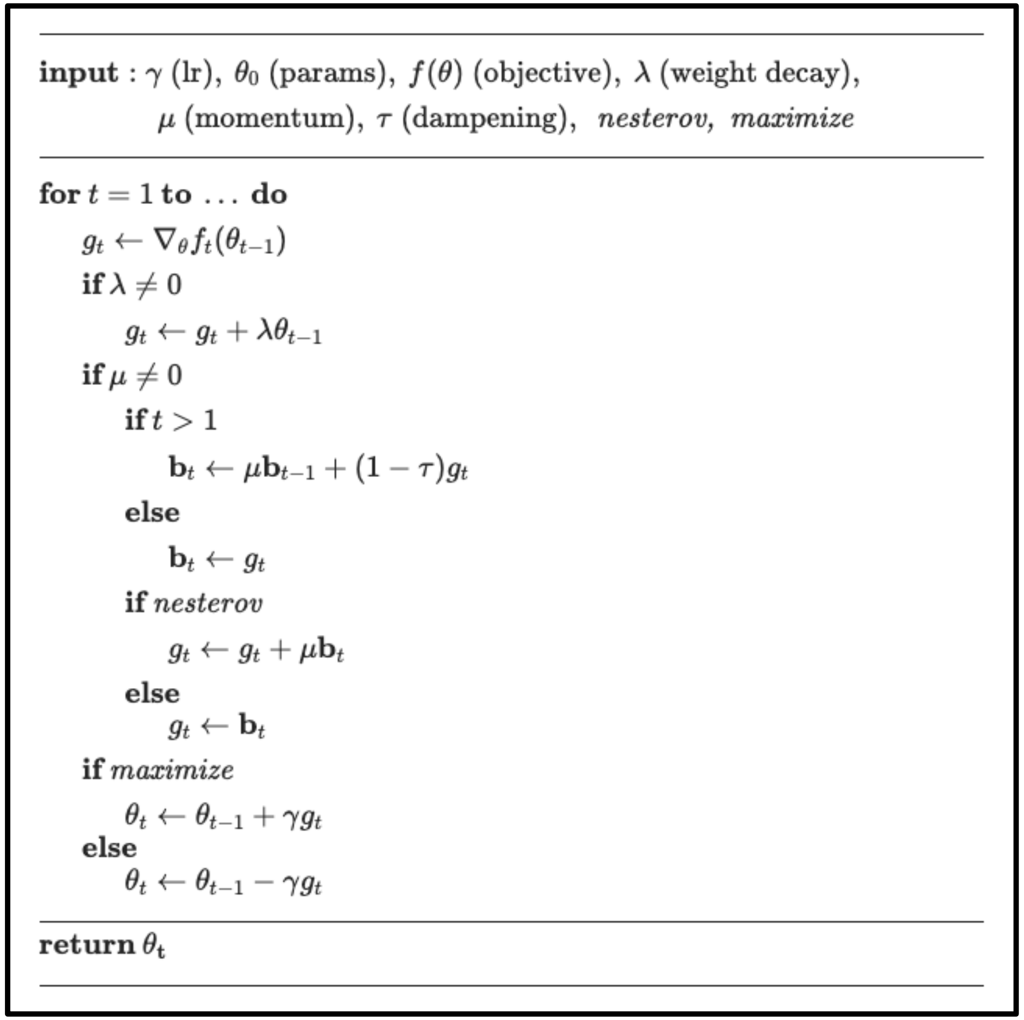


CLASS torch.optim.SGD(*params*, *lr=<required parameter>*, *momentum=0*, *dampening=0*, *weight_decay=0*, *nesterov=False*, *, *maximize=False*, *foreach=None*, *differentiable=False*) [SOURCE]

Implements stochastic gradient descent (optionally with momentum).

Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float*) – learning rate
- **momentum** (*float, optional*) – momentum factor (default: 0)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)
- **dampening** (*float, optional*) – dampening for momentum (default: 0)
- **nesterov** (*bool, optional*) – enables Nesterov momentum (default: False)
- **maximize** (*bool, optional*) – maximize the params based on the objective, instead of minimizing (default: False)
- **foreach** (*bool, optional*) – whether foreach implementation of optimizer is used. If unspecified by the user (so foreach is None), we will try to use foreach over the for-loop implementation on CUDA, since it is usually significantly more performant. Note that the foreach implementation uses ~ sizeof(params) more peak memory than the for-loop version due to the intermediates being a tensorlist vs just one tensor. If memory is prohibitive, batch fewer parameters through the optimizer at a time or switch this flag to False (default: None)
- **differentiable** (*bool, optional*) – whether autograd should occur through the optimizer step in training. Otherwise, the step() function runs in a torch.no_grad() context. Setting to True can impair performance, so leave it False if you don't intend to run autograd through this instance (default: False)



```
input : $\gamma$ (lr), $\theta_0$ (params), $f(\theta)$ (objective), $\lambda$ (weight decay),
        $\mu$ (momentum), $\tau$ (dampening), $nesterov,\ maximize$

for $t = 1$ to $\ldots$ do
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
    if $\lambda \neq 0$
        $g_t \leftarrow g_t + \lambda\theta_{t-1}$
    if $\mu \neq 0$
        if $t > 1$
            $\mathbf{b}_t \leftarrow \mu\mathbf{b}_{t-1} + (1-\tau)g_t$
        else
            $\mathbf{b}_t \leftarrow g_t$
        if $nesterov$
            $g_t \leftarrow g_t + \mu\mathbf{b}_t$
        else
            $g_t \leftarrow \mathbf{b}_t$
    if $maximize$
        $\theta_t \leftarrow \theta_{t-1} + \gamma g_t$
    else
        $\theta_t \leftarrow \theta_{t-1} - \gamma g_t$

return $\theta_t$
```

(Approximated) Second Order Method

Definition of Second-Order Derivative

- Suppose we have:
  - $f: \mathbb{R} \to \mathbb{R}$;
- Definition of a derivative:
  - $f'(x) = \frac{\partial f}{\partial x} = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}$;
- Definition of second-order derivative:
  - $f''(x) = \frac{\partial^2 f}{\partial x^2} = \lim_{\epsilon \to 0} \frac{f'(x+\epsilon) - f'(x)}{\epsilon}$;
- Represent the **local curvature**: how the slope of the function changes.

Definition of Second-Order Derivative

- Suppose we have:
  - $f: \mathbb{R}^d \to \mathbb{R}$;
- Definition of a gradient:
  - $\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \cdots \\ \frac{\partial f}{\partial x_d} \end{bmatrix} \in \mathbb{R}^d$

- Second-order derivative **Hessian matrix**:
  - $\nabla^2 f(x) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_p} \\ \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_p} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_p \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_p^2} \end{bmatrix} \in \mathbb{R}^{d \times d}$

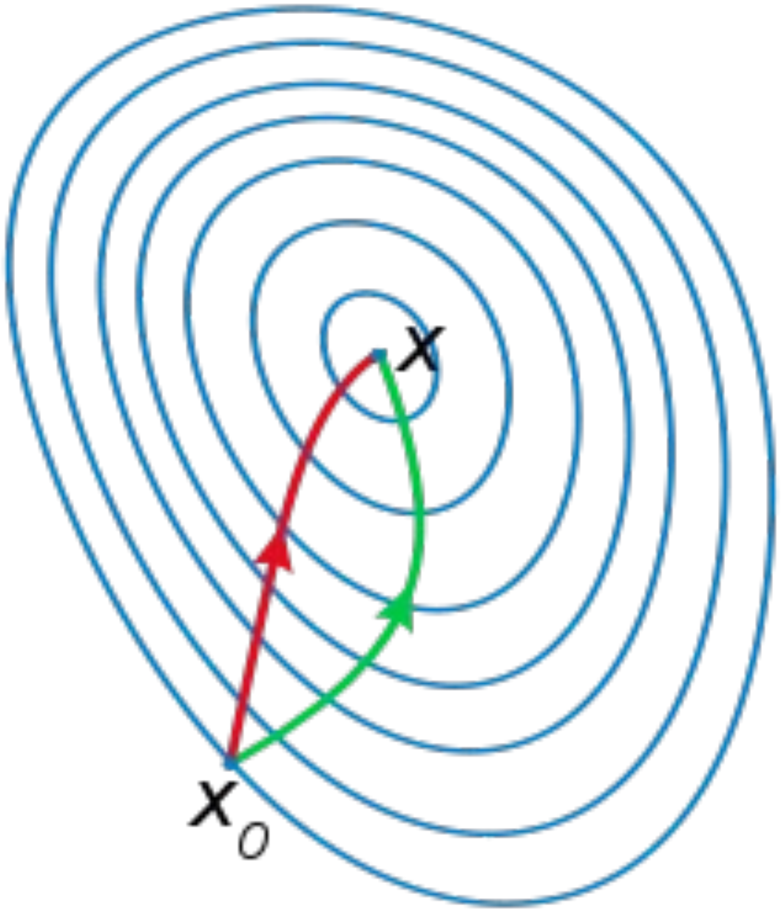
Netown’s Method

- Suppose we have:
  - w_0 denotes the value of the initalized parameter;
  - w_t denotes the value of the parameter at iteration t;
  - ∇f denotes the gradient of function f;
  - $\nabla^2 f$ denotes the Hessian matrix of function f.
- **<u>The Newton's method is defined by</u>**:

$$w_{t+1} = w_t - \left(\nabla^2 f(w_t)\right)^{-1} \nabla f(w_t)$$

Netwon’s Method

- Benefits:
  - Superlinear (quadratic) convergence rate!
- Problem:
  - To compute the Hessian matrix is too computationally expensive!
  - Even storing the Hessian matrix is impossible for most ML models.



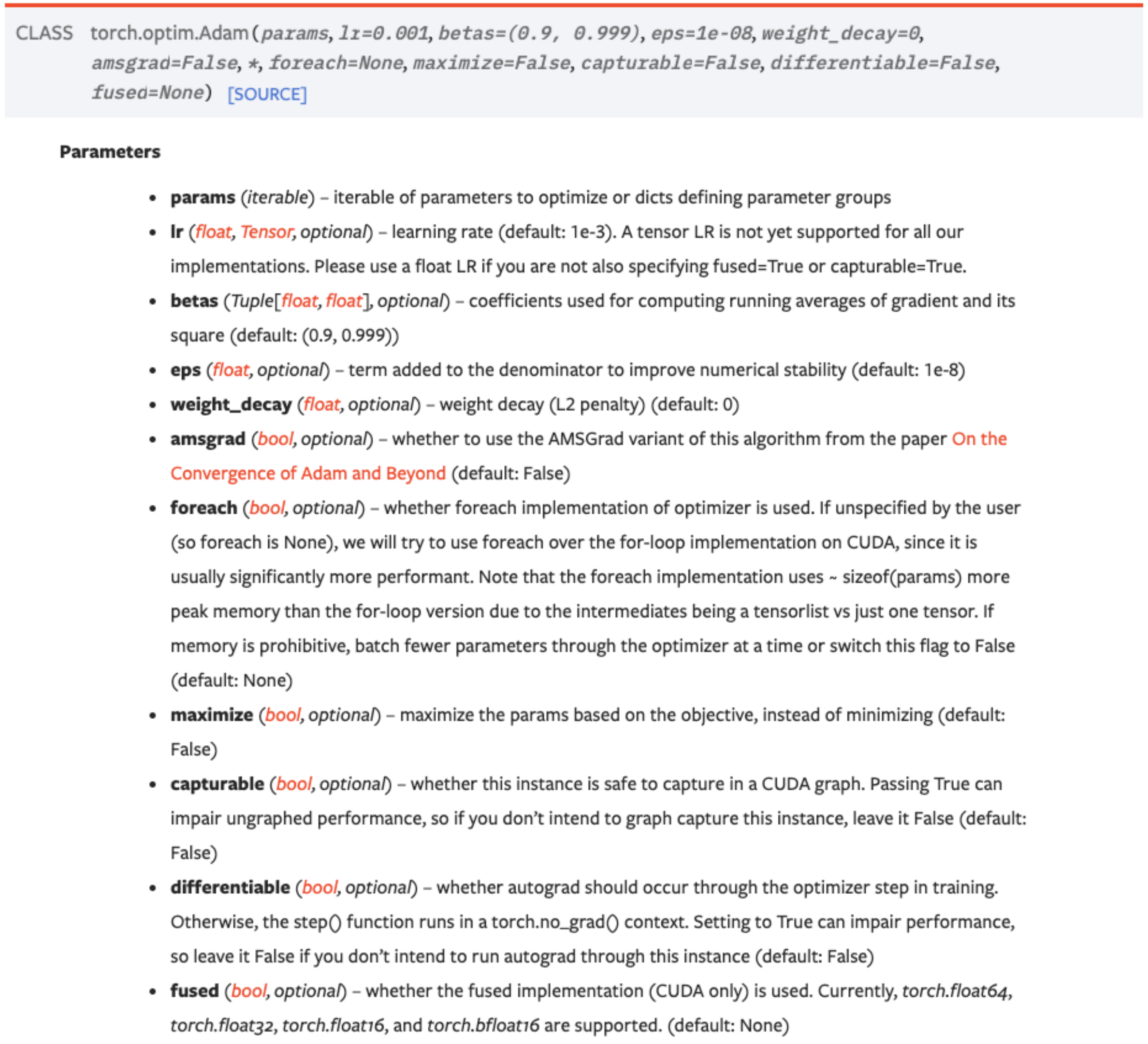
Gradient descent (green) v.s. Newton's method (red): Newton's method uses curvature information to take a more direct route.

Adaptive Moment Estimation (Adam)

- Suppose we have:
  - w_0 denotes the value of the initalized parameter;
  - w_t denotes the value of the parameter at iteration t;
  - ∇f denotes the gradient of function f;
  - m_t denotes the first order moment;
  - v_t denotes the second order moment;
  - β_1, β_2 denotes two hyper-parameters;

- **Adam is defined by**:
  - $m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla f(w_t)$
  - $v_t = \beta_2 v_{t-1} + (1 - \beta_2)\left(\nabla f(w_t)\right)^2$
  - $\widehat{m_t} = \frac{m_t}{1-\beta_1^t}$
  - $\widehat{v_t} = \frac{v_t}{1-\beta_2^t}$
  - $w_{t+1} = w_t - \alpha_t \frac{\widehat{m_t}}{\sqrt{\widehat{v_t}}+\epsilon}$

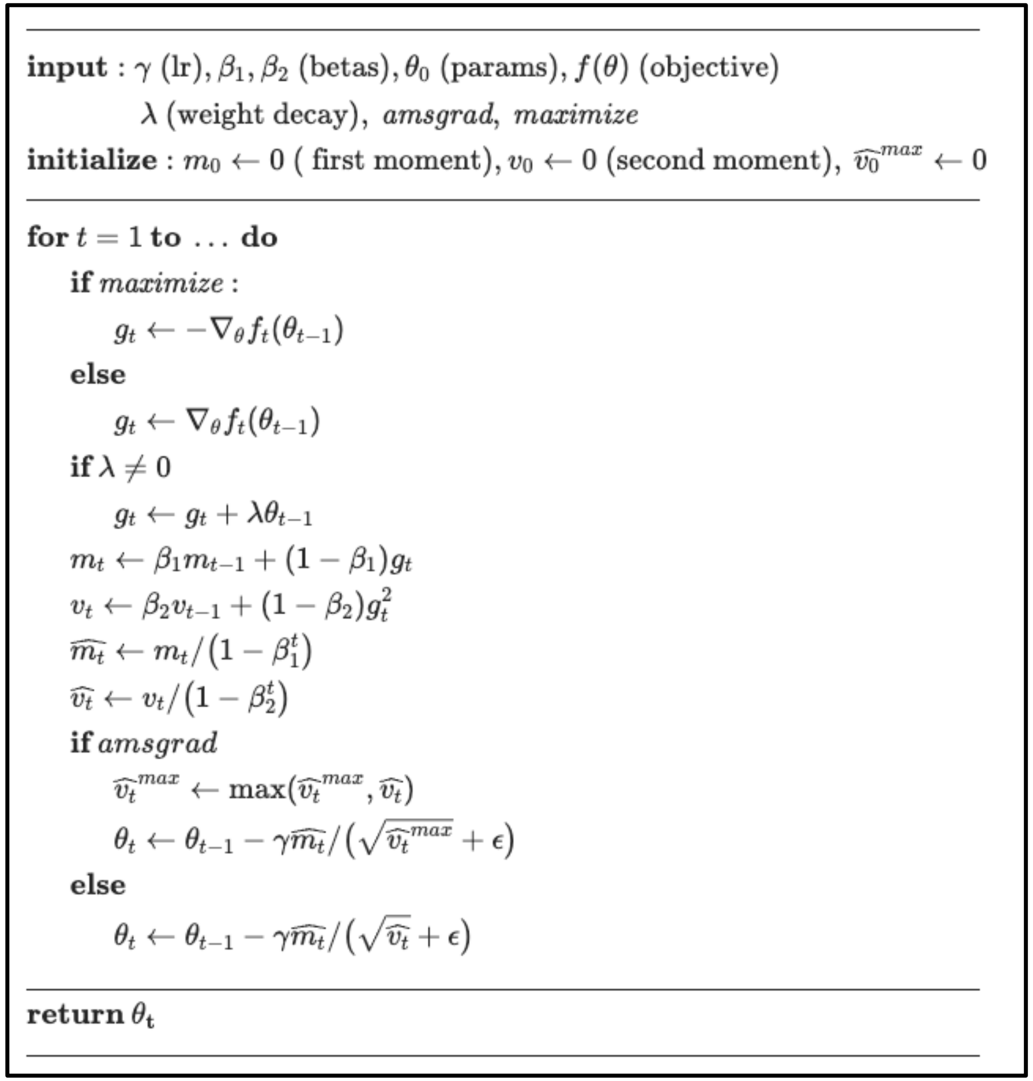
Adam in PyTorch



CLASS torch.optim.Adam(*params*, *lr=0.001*, *betas=(0.9, 0.999)*, *eps=1e-08*, *weight_decay=0*, *amsgrad=False*, ***, *foreach=None*, *maximize=False*, *capturable=False*, *differentiable=False*, *fused=None*) [SOURCE]

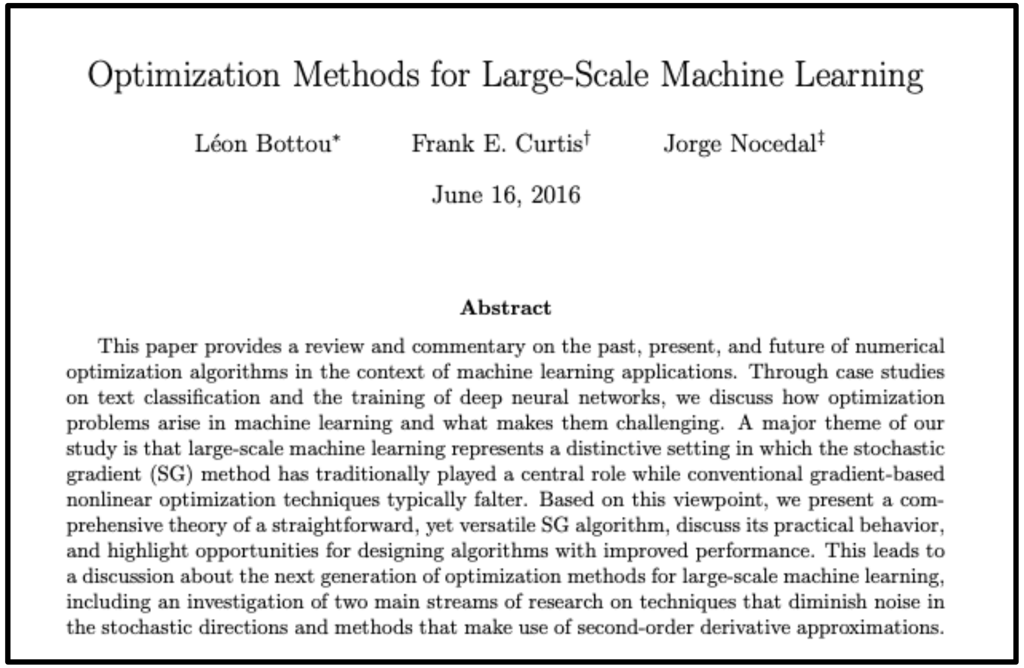
Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float*, *Tensor*, *optional*) – learning rate (default: 1e-3). A tensor LR is not yet supported for all our implementations. Please use a float LR if you are not also specifying fused=True or capturable=True.
- **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** (*bool*, *optional*) – whether to use the AMSGrad variant of this algorithm from the paper On the Convergence of Adam and Beyond (default: False)
- **foreach** (*bool*, *optional*) – whether foreach implementation of optimizer is used. If unspecified by the user (so foreach is None), we will try to use foreach over the for-loop implementation on CUDA, since it is usually significantly more performant. Note that the foreach implementation uses ~ sizeof(params) more peak memory than the for-loop version due to the intermediates being a tensorlist vs just one tensor. If memory is prohibitive, batch fewer parameters through the optimizer at a time or switch this flag to False (default: None)
- **maximize** (*bool*, *optional*) – maximize the params based on the objective, instead of minimizing (default: False)
- **capturable** (*bool*, *optional*) – whether this instance is safe to capture in a CUDA graph. Passing True can impair ungraphed performance, so if you don't intend to graph capture this instance, leave it False (default: False)
- **differentiable** (*bool*, *optional*) – whether autograd should occur through the optimizer step in training. Otherwise, the step() function runs in a torch.no_grad() context. Setting to True can impair performance, so leave it False if you don't intend to run autograd through this instance (default: False)
- **fused** (*bool*, *optional*) – whether the fused implementation (CUDA only) is used. Currently, *torch.float64*, *torch.float32*, *torch.float16*, and *torch.bfloat16* are supported. (default: None)



```
input : $\gamma$ (lr), $\beta_1, \beta_2$ (betas), $\theta_0$ (params), $f(\theta)$ (objective)
        $\lambda$ (weight decay), $amsgrad$, $maximize$
initialize : $m_0 \leftarrow 0$ ( first moment), $v_0 \leftarrow 0$ (second moment), $\widehat{v_0}^{max} \leftarrow 0$

for $t = 1$ to ... do
    if $maximize$ :
        $g_t \leftarrow -\nabla_{\theta} f_t(\theta_{t-1})$
    else
        $g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$
    if $\lambda \neq 0$
        $g_t \leftarrow g_t + \lambda\theta_{t-1}$
    $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
    $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
    $\widehat{m_t} \leftarrow m_t / (1 - \beta_1^t)$
    $\widehat{v_t} \leftarrow v_t / (1 - \beta_2^t)$
    if $amsgrad$
        $\widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t})$
        $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / (\sqrt{\widehat{v_t}^{max}} + \epsilon)$
    else
        $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / (\sqrt{\widehat{v_t}} + \epsilon)$

return $\theta_t$
```

Further Reading (Optional)



Optimization Methods for Large-Scale Machine Learning

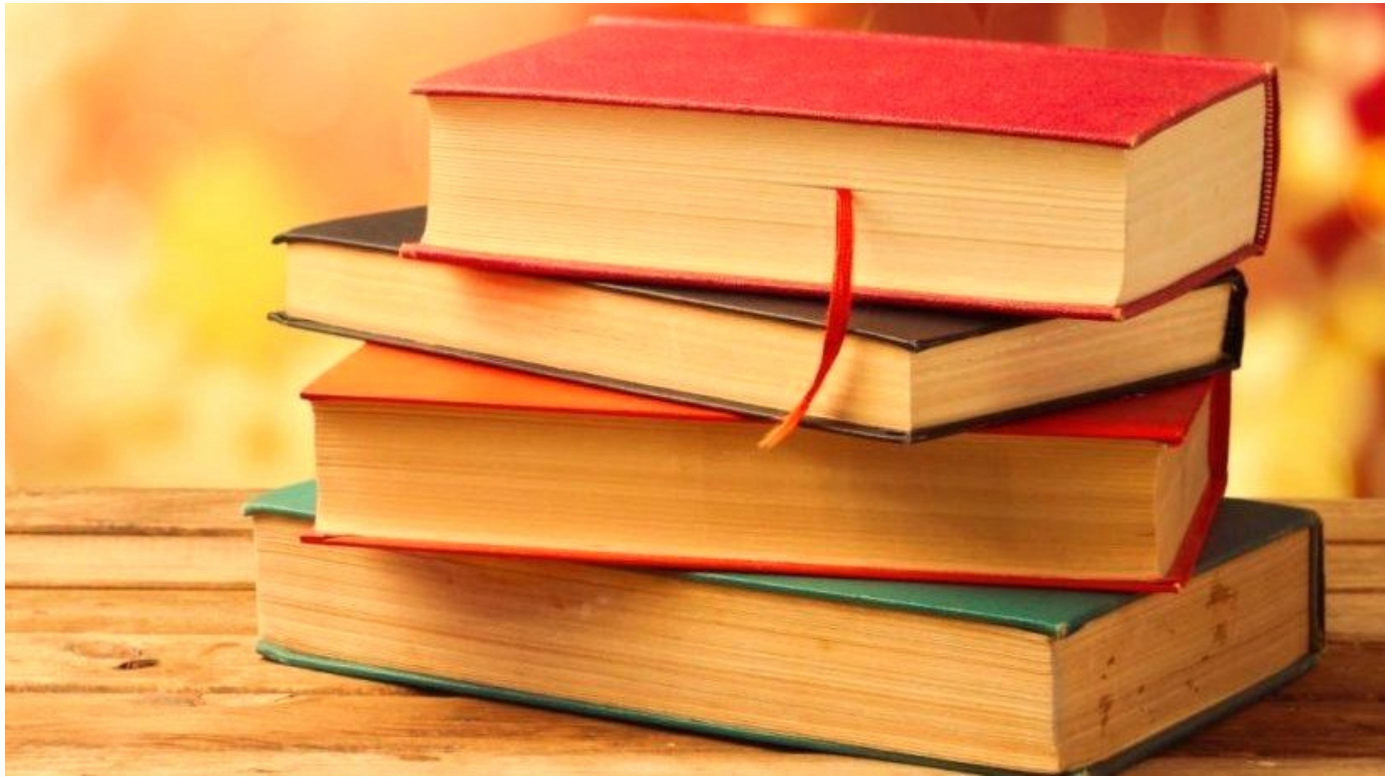
Léon Bottou* Frank E. Curtis† Jorge Nocedal‡

June 16, 2016

Abstract

This paper provides a review and commentary on the past, present, and future of numerical optimization algorithms in the context of machine learning applications. Through case studies on text classification and the training of deep neural networks, we discuss how optimization problems arise in machine learning and what makes them challenging. A major theme of our study is that large-scale machine learning represents a distinctive setting in which the stochastic gradient (SG) method has traditionally played a central role while conventional gradient-based nonlinear optimization techniques typically falter. Based on this viewpoint, we present a comprehensive theory of a straightforward, yet versatile SG algorithm, discuss its practical behavior, and highlight opportunities for designing algorithms with improved performance. This leads to a discussion about the next generation of optimization methods for large-scale machine learning, including an investigation of two main streams of research on techniques that diminish noise in the stochastic directions and methods that make use of second-order derivative approximations.

https://arxiv.org/abs/1606.04838



PyTorch Practice

https://pytorch.org/tutorials/beginner/basics/optimization_tutorial.html

The Frist Training Script

Load necessary packages

```
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor
import matplotlib
from matplotlib import pyplot
```

The Frist Training Script

Define some hyper-parameters

```
optimizer_name = "SGD"
learning_rate = 1e-3
momentum = 0.5
batch_size = 16
epochs = 10
```

The Frist Training Script

Define the data loaders

```
training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)

train_dataloader = DataLoader(training_data, batch_size=batch_size)
test_dataloader = DataLoader(test_data, batch_size=batch_size)
```

The Frist Training Script

Define the Model

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits

model = NeuralNetwork()
```

The Frist Training Script

Define the loss function and optimizer

```
# Define the loss function
loss_fn = nn.CrossEntropyLoss()

# Define the optimizer
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate,
momentum=momentum)
```

The Frist Training Script

Define the training function

```
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    log = {'loss': [], '#samples': []}
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * batch_size + len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
            log['loss'].append(loss)
            log['#samples'].append(current)
    return log
```

The Frist Training Script

Define the test function

```
def test_loop(dataloader, model, loss_fn):
    model.eval()
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()
    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")
```

The Frist Training Script

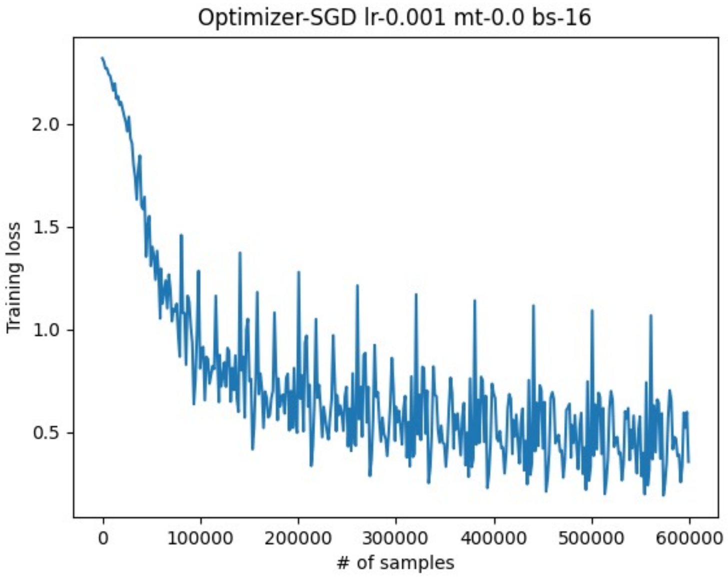
Put them together and visualize the results

```
log = {'loss': [], '#samples': []}
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    epoch_log = train_loop(train_dataloader, model, loss_fn, optimizer)
    log['loss'].extend(epoch_log['loss'])
    log['#samples'].extend([ v+60000*t for v in epoch_log['#samples']])
    test_loop(test_dataloader, model, loss_fn)

print("Done!")

pyplot.plot(log['#samples'], log['loss']);
pyplot.xlabel("# of samples");
pyplot.ylabel("Training loss");

title = "Optimizer-{} lr-{} mt-{} bs-{}".format(optimizer_name, learning_rate, momentum,
batch_size)
pyplot.title(title);
pyplot.savefig('./'+title+'.png')
```

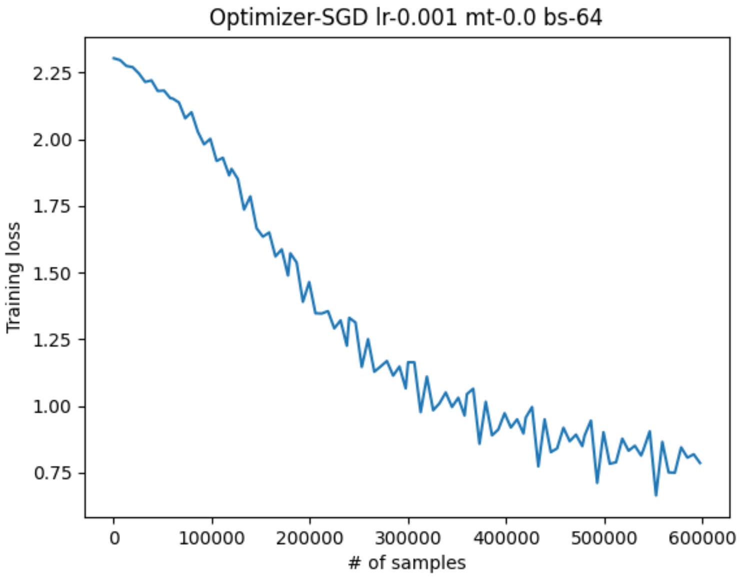
RELAXED
SYSTEM LAB

Demo & Results

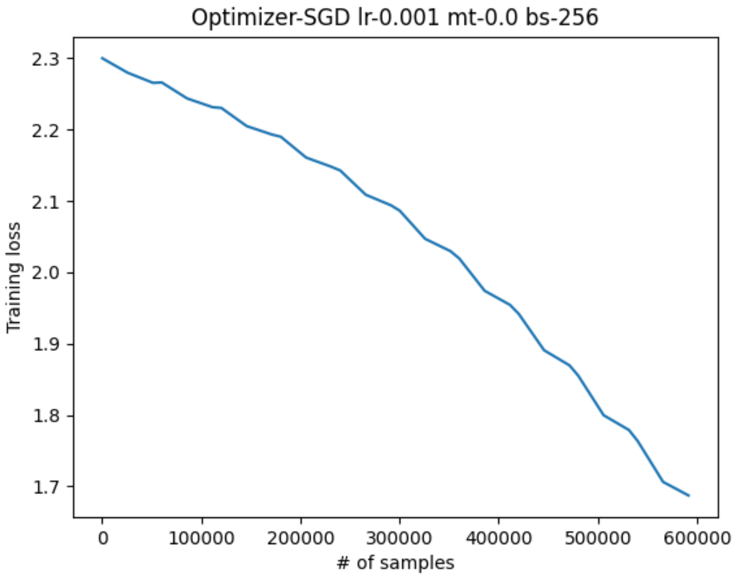
SGD: Batch Size



Batch size: 16

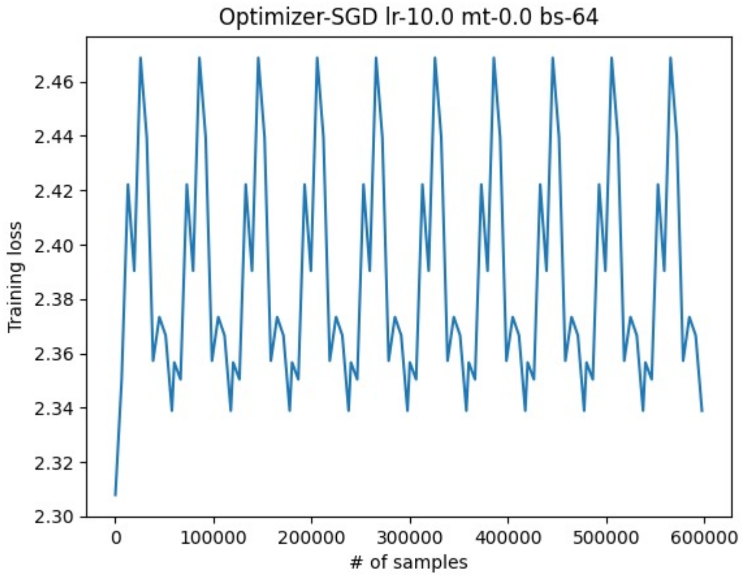


Batch size: 64

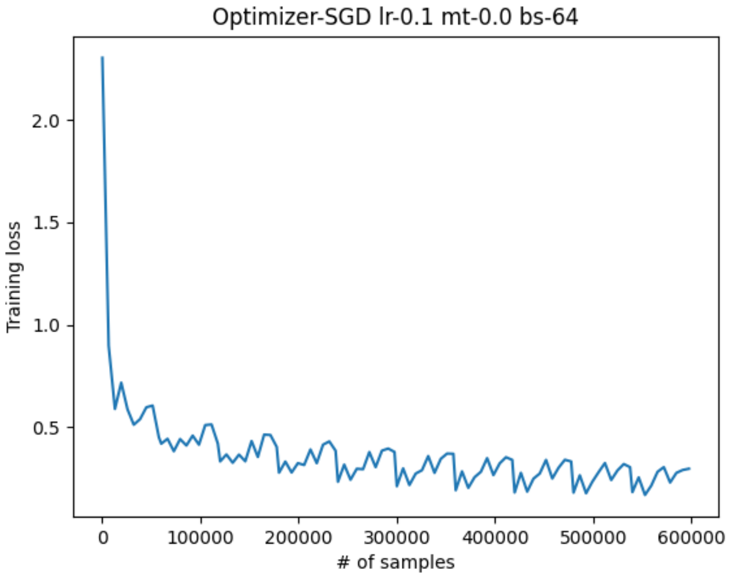


Batch size: 256

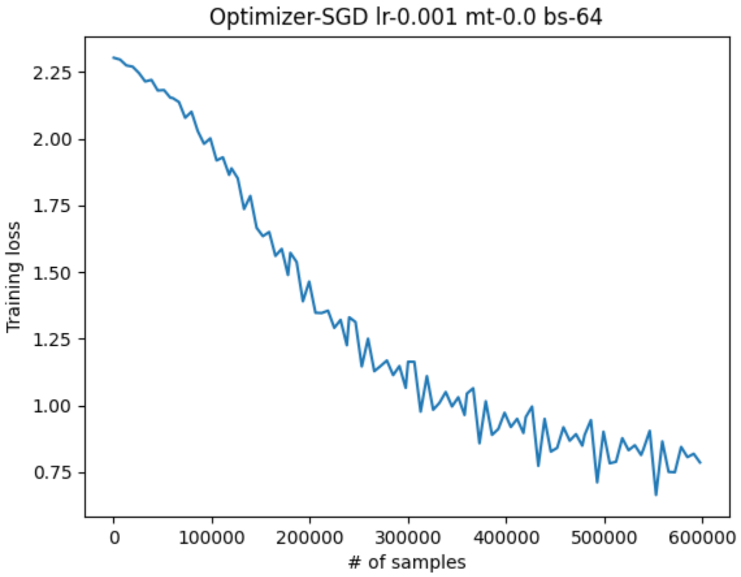
SGD: Learning Rate



<u>Learning rate</u>:10

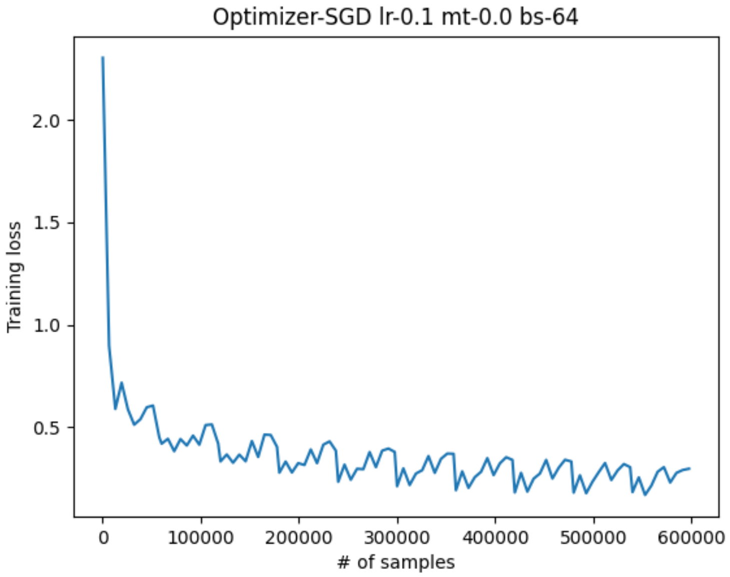


<u>Learning rate</u>: 0.1

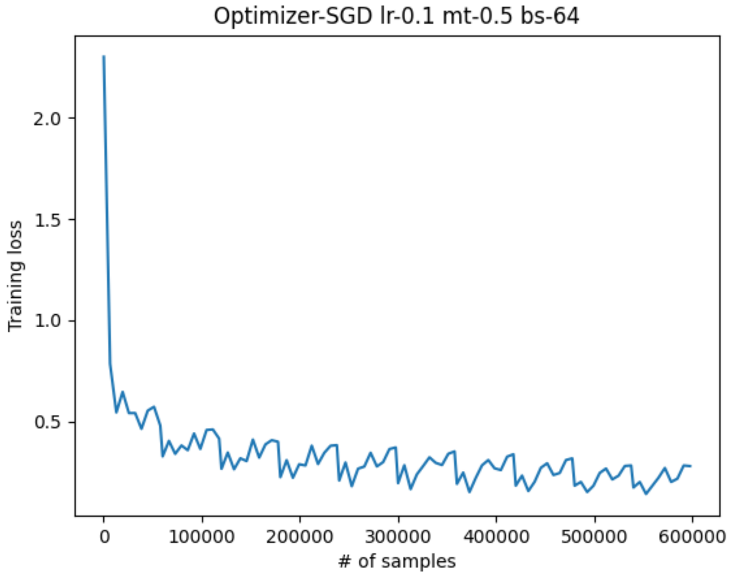


<u>Learning rate</u>: 0.001

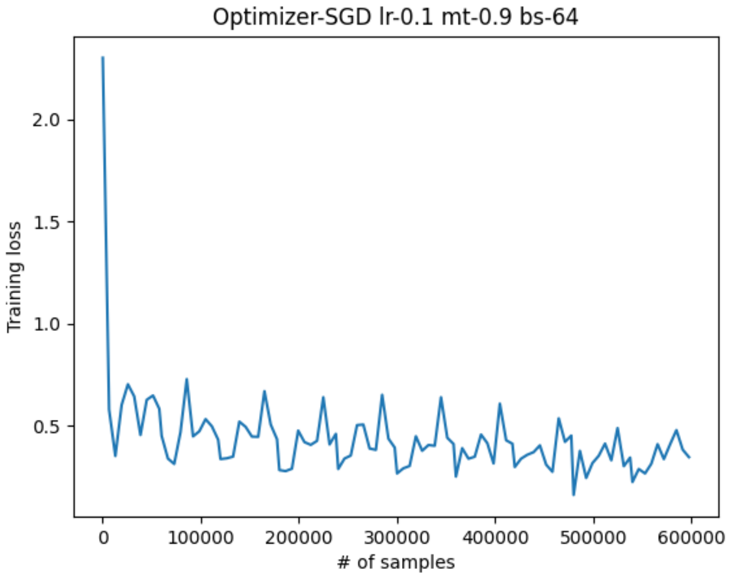
SGD: Momentum



Momentum: 0

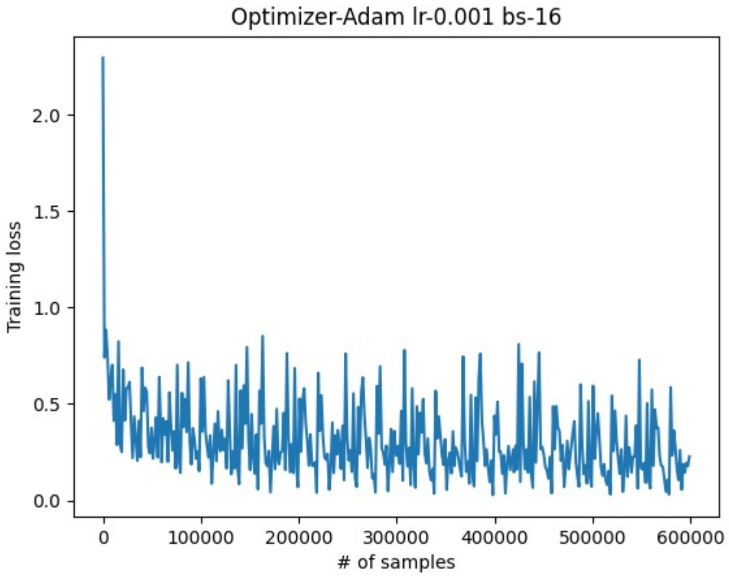


Momentum: 0.5

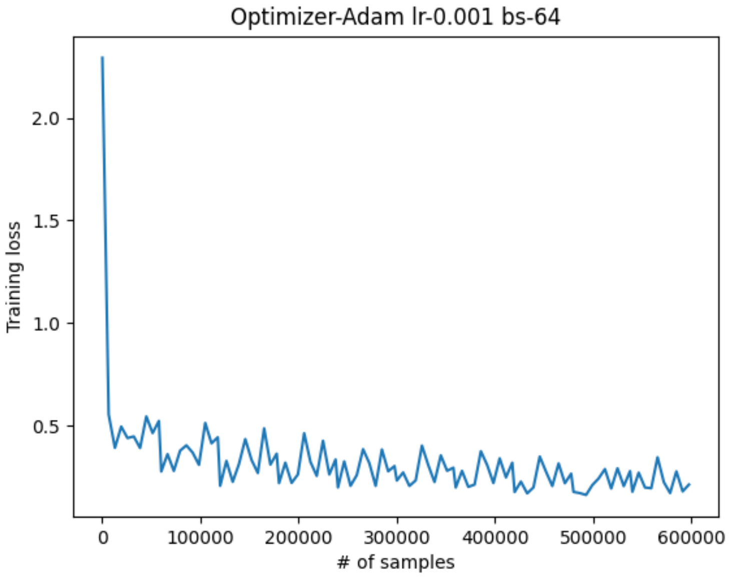


Momentum: 0.9

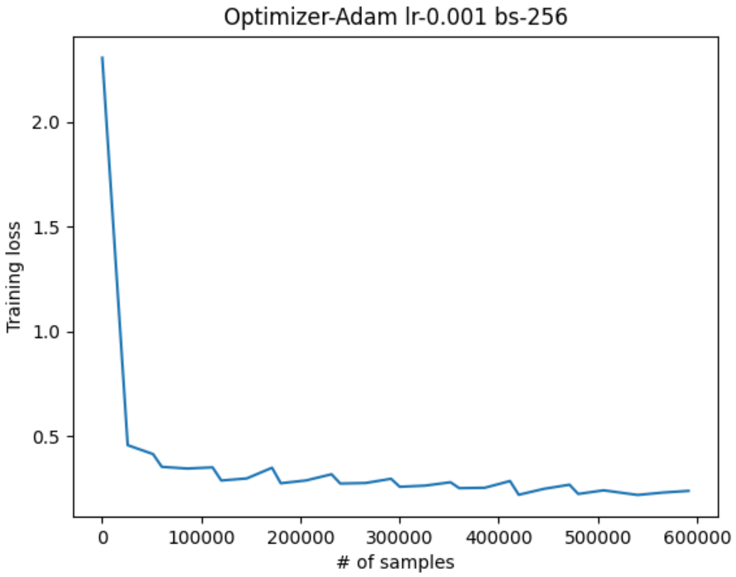
Adam: Batch Size


Optimizer-Adam lr-0.001 bs-16

<u>Batch size</u>: 16

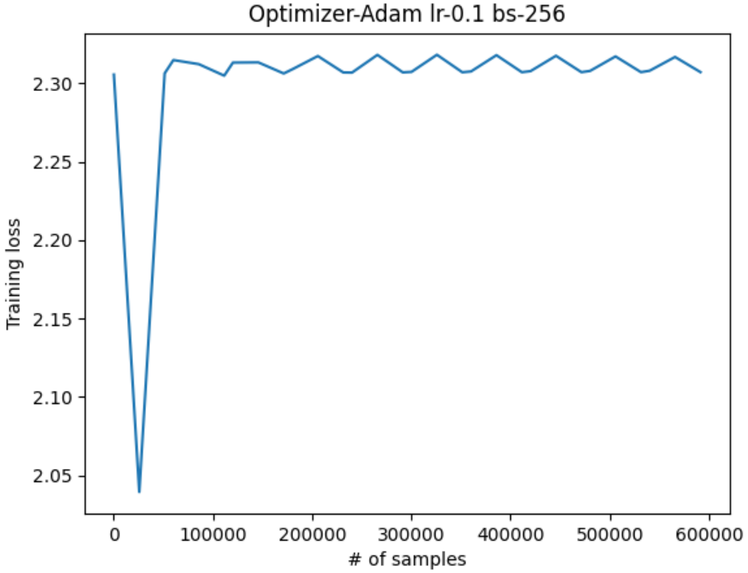

Optimizer-Adam lr-0.001 bs-64

<u>Batch size</u>: 64

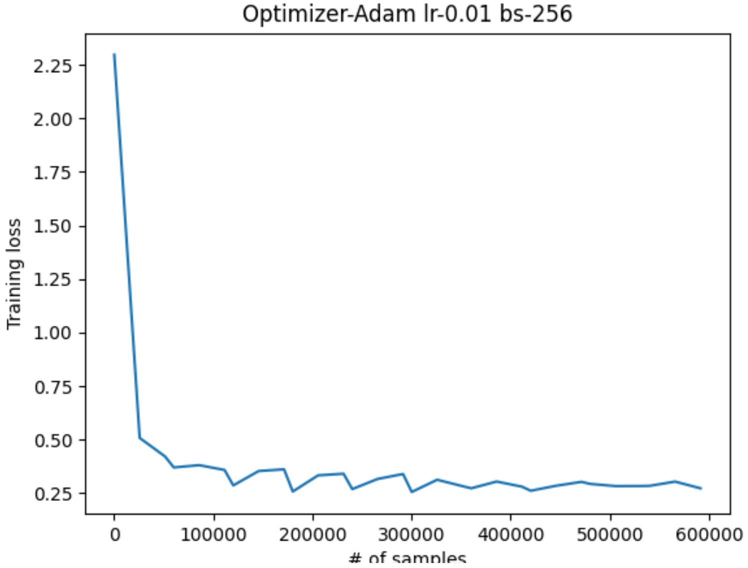

Optimizer-Adam lr-0.001 bs-256

<u>Batch size</u>: 256

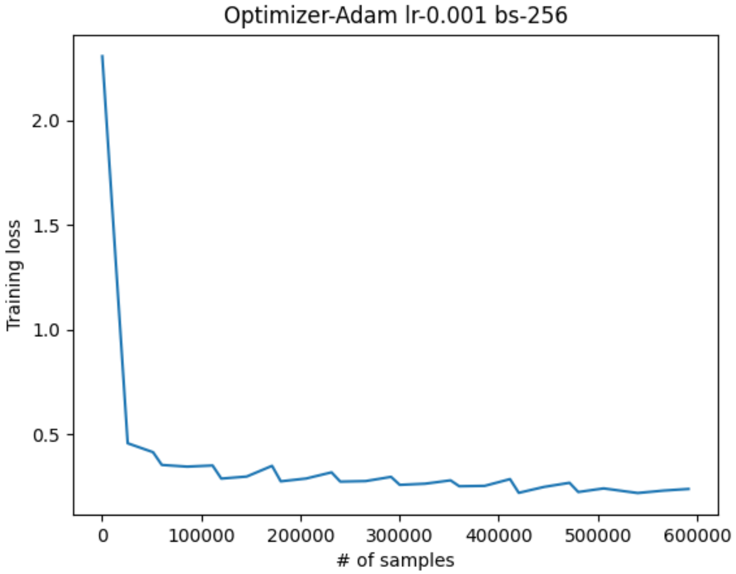
Adam: Learning Rate


Optimizer-Adam lr-0.1 bs-256

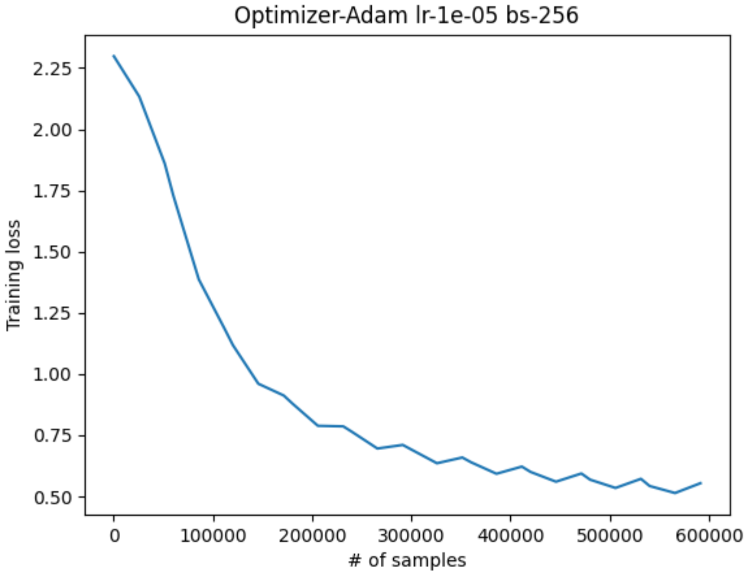
LR:0.1


Optimizer-Adam lr-0.01 bs-256

LR:0.01


Optimizer-Adam lr-0.001 bs-256

LR:0.001


Optimizer-Adam lr-1e-05 bs-256

LR:0.00001

References

- https://www.ruder.io/optimizing-gradient-descent/
- https://www.cs.cornell.edu/courses/cs4787/2023fa/lectures/lecture6.html
- https://www.cs.cornell.edu/courses/cs4787/2023fa/lectures/lecture7.html
- https://mitliagkas.github.io/ift6085-2019/ift-6085-lecture-6-notes.pdf
- https://akyrillidis.github.io/comp414-514/schedule/
- https://towardsdatascience.com/stochastic-gradient-descent-with-momentum-a84097641a5d
- https://www.linkedin.com/pulse/demystifying-optimization-techniques-machine-learning-cabaleiro/