Stochastic Gradient Descent

COMP6211J

Binhang Yuan

Linear Algebra

Scalars

RELAXED
SYSTEM LAB

- Sample operations

$$c = a + b$$

$$c = a \cdot b$$

$$c = \sin a$$

- Length

$$|a| = \begin{cases} \text{a} & if \;\; a > 0 \\ -a & otherwise \end{cases}$$

$$|a + b| \leq |a| + |b|$$

$$|a \cdot b| = |a| \cdot |b|$$

Vector

RELAXED
SYSTEM LAB

- Vector in n-dimensions $\boldsymbol{a} \in \mathbb{R}^n$, $\mathbf{a}= \begin{bmatrix} a_1 \\ a_2 \\ \dots \\ a_n \end{bmatrix}$
- Sample operations:

$$\boldsymbol{c} = \boldsymbol{a} + \boldsymbol{b} \ \ where\ c_i = a_i + b_i$$

$$\boldsymbol{c} = \alpha \cdot \boldsymbol{b} \ \ where\ c_i = \alpha b_i$$

$$\boldsymbol{c} = \sin \boldsymbol{a} \ \ where\ c_i = \sin a_i$$

Vector

RELAXED
SYSTEM LAB

- Some properties of vector addition:
  - Commutative:

$$\boldsymbol{a} + \boldsymbol{b} = \boldsymbol{b} + \boldsymbol{a}, \qquad \boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^n$$

  - Associative:

$$(\boldsymbol{a} + \boldsymbol{b}) + \boldsymbol{c} = \boldsymbol{a} + (\boldsymbol{b} + \boldsymbol{c}), \qquad \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c} \in \mathbb{R}^n$$

  - Distibutive:

$$\alpha(\boldsymbol{a} + \boldsymbol{b}) = \alpha\boldsymbol{a} + \alpha\boldsymbol{b}, \qquad \boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^n$$

Vector

- p-norm

$$\|\boldsymbol{a}\|_p = \left[\sum_{i=1}^{m} a_i^p\right]^{\frac{1}{p}}$$

- p=1, Manhattan norm

$$\|\boldsymbol{a}\|_1 = \sum_{i=1}^{m} |a_i|$$

- p=2, Euclidean norm

$$\|\boldsymbol{a}\|_2 = \left[\sum_{i=1}^{m} a_i^2\right]^{\frac{1}{2}}$$

$$\|\boldsymbol{a}\| \geq 0 \; \forall \; \boldsymbol{a}$$

$$\|\boldsymbol{a} + \boldsymbol{b}\| \leq \|\boldsymbol{a}\| + \|\boldsymbol{b}\|$$

$$\|\alpha \cdot \boldsymbol{b}\| \leq |\alpha| \|\boldsymbol{b}\|$$

- p= ∞, infinity norm

$$\|\boldsymbol{a}\|_\infty = \max(|a_1|, |a_2|, \ldots, |a_m|)$$

Matrices

RELAXED
SYSTEM LAB

- Matrix in m, n-dimensions: $\boldsymbol{A} \in \mathbb{R}^{m \times n}$:

$$\mathbf{A} = \begin{bmatrix} A_{11} & \cdots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{m1} & \cdots & A_{mn} \end{bmatrix}$$

- Transpose of matrix:

$$\mathbf{A}^{\mathrm{T}} = \begin{bmatrix} A_{11} & \cdots & A_{m1} \\ \vdots & \ddots & \vdots \\ A_{1n} & \cdots & A_{mn} \end{bmatrix} \in \mathbb{R}^{n \times m}$$

$$(\boldsymbol{A} + \boldsymbol{B})^T = \boldsymbol{A}^T + \boldsymbol{B}^T, \forall \boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{m \times n}$$

Matrices

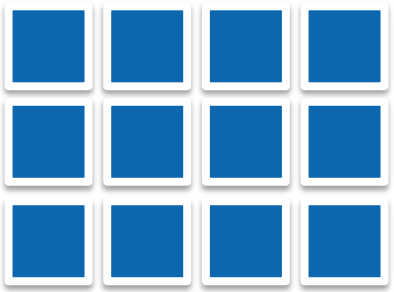
RELAXED
SYSTEM LAB

- Simple operations

$$\boldsymbol{C} = \boldsymbol{A} + \boldsymbol{B} \;\; where \; C_{ij} = A_{ij} + B_{ij}$$

$$\boldsymbol{C} = \alpha \cdot \boldsymbol{B} \;\; where \; C_{ij} = \alpha B_{ij}$$

$$\boldsymbol{C} = \sin \boldsymbol{A} \;\; where \; C_{ij} = \sin A_{ij}$$



Matrices

RELAXED
SYSTEM LAB

- Some properties of matrix addition:
  - Commutative:

$$\boldsymbol{A} + \boldsymbol{B} = \boldsymbol{B} + \boldsymbol{A}, \qquad \boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{m \times n}$$

  - Associative:

$$(\boldsymbol{A} + \boldsymbol{B}) + \boldsymbol{C} = \boldsymbol{A} + (\boldsymbol{B} + \boldsymbol{C}), \qquad \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C} \in \mathbb{R}^{m \times n}$$

  - Distibutive:

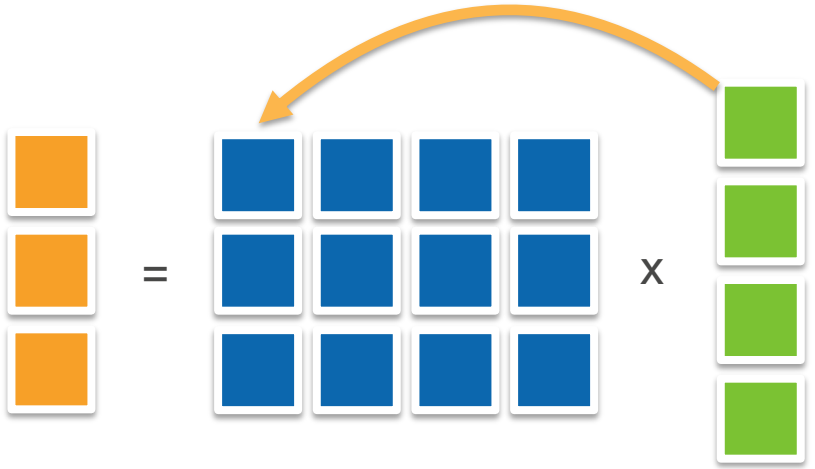
$$\alpha(\boldsymbol{A} + \boldsymbol{B}) = \alpha\boldsymbol{A} + \alpha\boldsymbol{B}, \qquad \boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{m \times n}$$

Matrices

RELAXED SYSTEM LAB

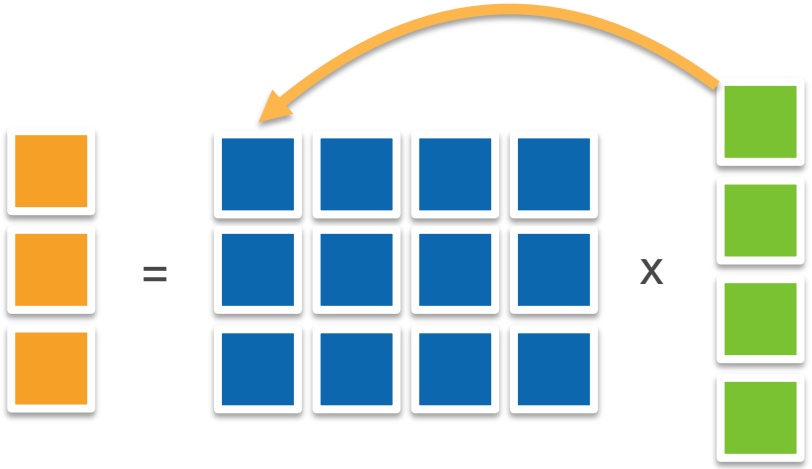
- Multiplications (matrix-vector), $\boldsymbol{c} = \boldsymbol{Ab}, \boldsymbol{c} \in \mathbb{R}^m, \boldsymbol{A} \in \mathbb{R}^{m \times n}, \boldsymbol{B} \in \mathbb{R}^n$

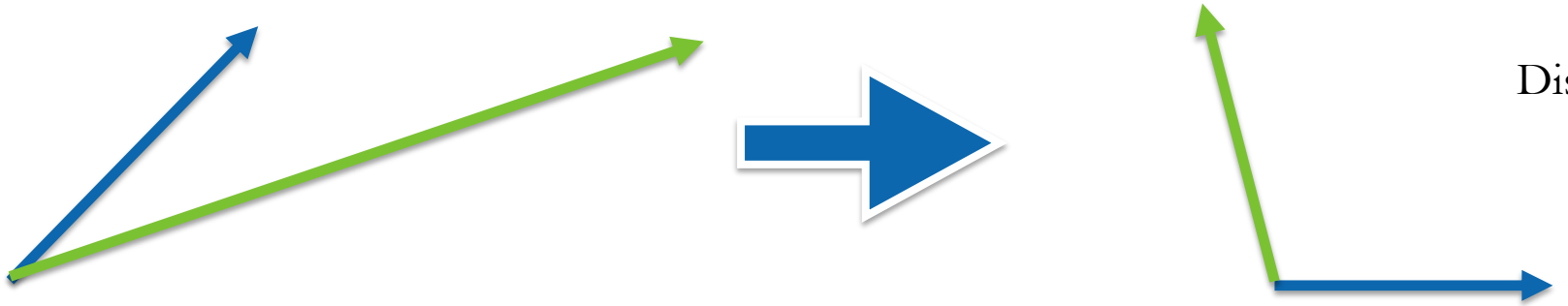
$$\begin{bmatrix} c_1 \\ \dots \\ c_m \end{bmatrix} = \boldsymbol{c} = \boldsymbol{Ab} = \begin{bmatrix} A_{11} & \cdots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{m1} & \cdots & A_{mn} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{bmatrix} \quad where\ c_i = \sum_{j=1}^{n} A_{ij} b_j$$



Matrices

RELAXED
SYSTEM LAB





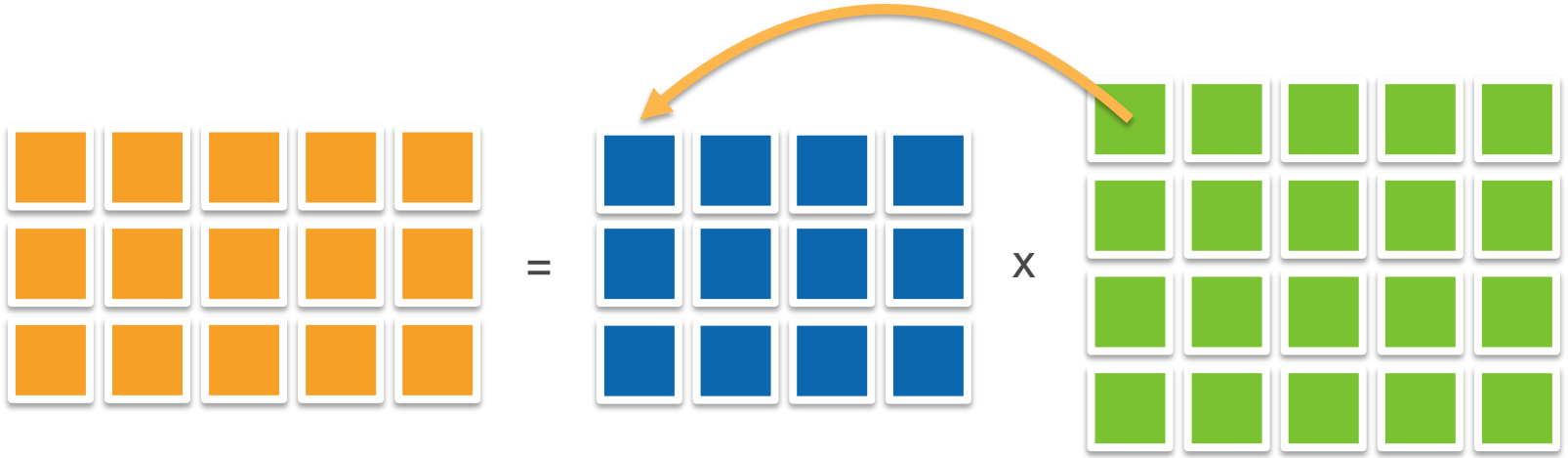
Distort the space.

Matrices

RELAXED SYSTEM LAB

- Multiplications (matrix-matrix) $\boldsymbol{C} = \boldsymbol{AB}, \boldsymbol{C} \in \mathbb{R}^{m \times p}, \boldsymbol{A} \in \mathbb{R}^{m \times n}, \boldsymbol{B} \in \mathbb{R}^{n \times p}$

$$\begin{bmatrix} C_{11} & \cdots & C_{1p} \\ \vdots & \ddots & \vdots \\ C_{m1} & \cdots & C_{mp} \end{bmatrix} = \boldsymbol{C} = \boldsymbol{AB} = \begin{bmatrix} A_{11} & \cdots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{m1} & \cdots & A_{mn} \end{bmatrix} \begin{bmatrix} B_{11} & \cdots & A_{1p} \\ \vdots & \ddots & \vdots \\ B_{n1} & \cdots & A_{np} \end{bmatrix} \text{where } C_{ik} = \sum_{j=1}^{n} A_{ij} B_{jk}$$



Matrices

RELAXED SYSTEM LAB

- Some properties of matrix multiplication:
  - Non-commutative!

$$\boldsymbol{AB} \neq \boldsymbol{BA}$$

  - Associative:

$$(\boldsymbol{AB})\boldsymbol{C} = \boldsymbol{A}(\boldsymbol{BC}), \qquad \forall \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$$

$$\alpha(\boldsymbol{AB}) = (\alpha\boldsymbol{A})\boldsymbol{B}, \qquad \forall \boldsymbol{A}, \boldsymbol{B}$$

  - Distibutive:

$$\boldsymbol{A}(\boldsymbol{B} + \boldsymbol{C}) = \boldsymbol{AB} + \boldsymbol{AC}, \qquad \forall \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$$

  - Transpose:

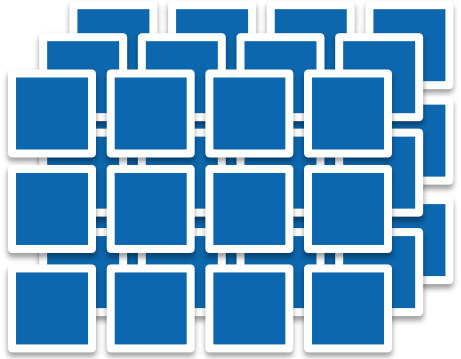
$$(\boldsymbol{AB})^T = \boldsymbol{B}^T \boldsymbol{A}^T$$

Tensor

RELAXED
SYSTEM LAB

- A tensor is a collection of numbers labelled by indices.

- The rank of a tensor is the number of indices required to specify an entry in the tensor:
  - A vector is a rank–1 tensor;
  - A matrix is a rank–2 tensor.



Tensor

RELAXED SYSTEM LAB

- Einstein summation convention:
  - Each index can appear at most twice in any term.
  - Repeated indices are implicitly summed over.
  - Each term must contain identical non-repeated indices.

$$M_{ij}v_j \equiv \sum_j M_{ij}v_j$$

Matrix multiplication between matrix M and vector v

$$M_{ij}u_jv_j$$

The index j appears three times in the first term.

$$T_{ijk}u_k + M_{ip}$$

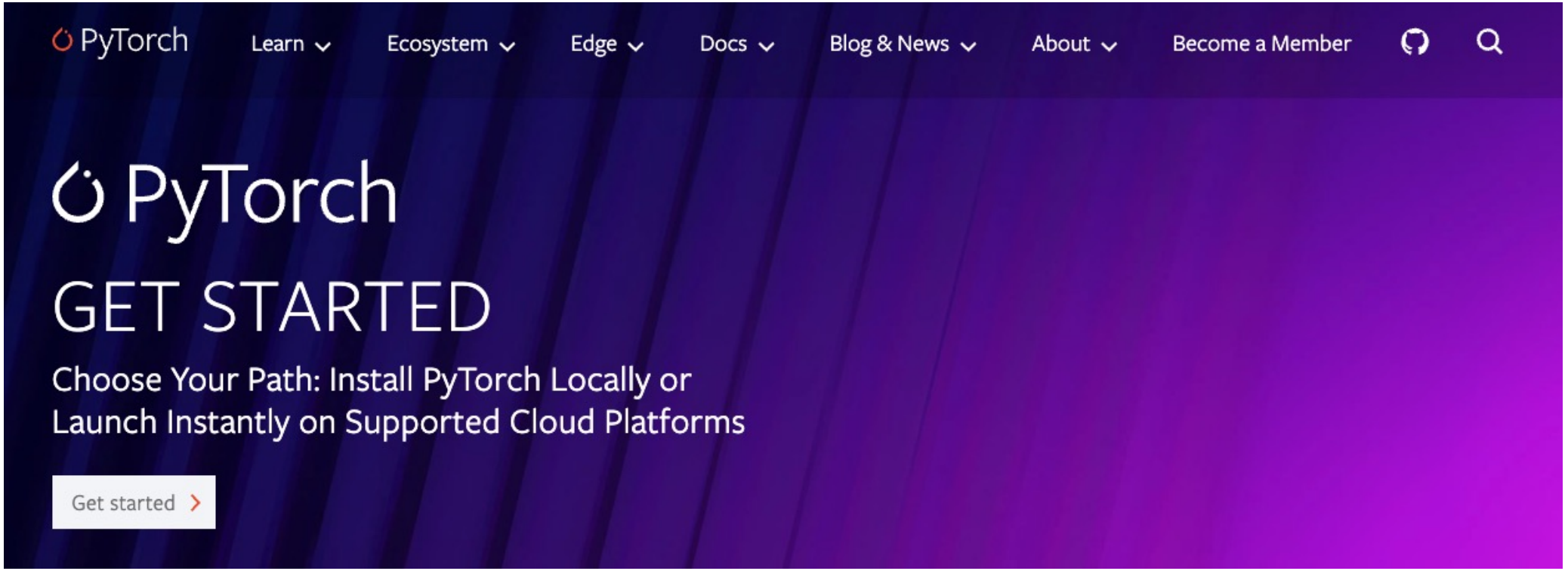
The first term contains the non-repeated index j whereas the second term contains p.

Tensors in PyTorch

PyTorch

RELAXED
SYSTEM LAB

- End-to-end machine learning framework developed by Meta (https://pytorch.org/)



Tensor Broadcasting

RELAXED SYSTEM LAB

- Tensor broadcasting:
  - Perform an operation between tensors that have similarities in their shapes;
  - if a PyTorch operation supports broadcast, then its Tensor arguments can be automatically expanded to be of equal sizes (without making copies of the data).
- Two tensors are "broadcastable" if the following rules hold:
  - Each tensor must have at least one dimension (no empty tensors).
  - Comparing the dimension sizes of the two tensors, going ***from last to first***:
    - Each dimension must be equal, or
    - One of the dimensions must be of size 1, or
    - The dimension does not exist in one of the tensors.
- If two tensors x, y are "broadcastable", the resulting tensor size is calculated as:
  - If the number of dimensions of x and y are not equal, prepend 1 to the dimensions of the tensor with fewer dimensions to make them equal length.
  - Then, for each dimension size, the resulting dimension size is the max of the sizes of x and y along that dimension.

```
x=torch.empty(5,3,4,1)
y=torch.empty(  3,1,1)
```

```
x=torch.empty(5,2,4,1)
y=torch.empty(  3,1,1)
```

```
x=torch.empty(5,1,4,1)
y=torch.empty(  3,1,1)
(x+y).size()

torch.Size([5, 3, 4, 1])
```

Math & Logic with PyTorch Tensors

RELAXED SYSTEM LAB

- Examples allow broadcasting.

Code	Output		
a = torch.ones(4, 3, 2) # 3rd & 2nd dims identical to a, dim 1 absent b = a * torch.rand(3, 2) print(b) # 3rd dim = 1, 2nd dim identical to a c = a * torch.rand(3, 1) print(c) # 3rd dim identical to a, 2nd dim = 1 d = a * torch.rand(1, 2) print(d)	tensor([[[0.6493, 0.2633], [0.4762, 0.0548], [0.2024, 0.5731]], [[0.6493, 0.2633], [0.4762, 0.0548], [0.2024, 0.5731]], [[0.6493, 0.2633], [0.4762, 0.0548], [0.2024, 0.5731]], [[0.6493, 0.2633], [0.4762, 0.0548], [0.2024, 0.5731]]])	tensor([[[0.7191, 0.7191], [0.4067, 0.4067], [0.7301, 0.7301]], [[0.7191, 0.7191], [0.4067, 0.4067], [0.7301, 0.7301]], [[0.7191, 0.7191], [0.4067, 0.4067], [0.7301, 0.7301]], [[0.7191, 0.7191], [0.4067, 0.4067], [0.7301, 0.7301]]])	tensor([[[0.6276, 0.7357], [0.6276, 0.7357], [0.6276, 0.7357]], [[0.6276, 0.7357], [0.6276, 0.7357], [0.6276, 0.7357]], [[0.6276, 0.7357], [0.6276, 0.7357], [0.6276, 0.7357]], [[0.6276, 0.7357], [0.6276, 0.7357], [0.6276, 0.7357]]])

Math & Logic with PyTorch Tensors

RELAXED SYSTEM LAB

- Examples attempt at broadcasting that will fail.

Code

```
a = torch.ones(4, 3, 2)

b = a * torch.rand(4, 3)    # dimensions must match last-to-first

c = a * torch.rand(2, 3) # both 3rd & 2nd dims different

d = a * torch.rand((0, ))   # can't broadcast with an empty tensor
```

Einstein Notation in PyTorch

RELAXED SYSTEM LAB

Code

```
a = torch.rand(2,3)
b = torch.rand(3,4)
c = torch.einsum("ik,kj->ij", a, b)
```

$$c_{ij} = \sum_{k=0}^{k<3} a_{ik} b_{kj}$$

- Einstein summation in PyTorch:
  - *free index*: index on the right-hand side (e.g., i, j in the above example).
  - *summation index*: index only on the left-hand side, index to be summed over (e.g., k in the above example).
- Execution:
  - Repeated indices among different input operands are multiplied.
  - And then summed over.
  - The indices on the output side can be permutated.
  - If the right-hand side is ignored, the indices that appear only once on the left-hand side will be placed on the right-hand side by default.

TORCH.EINSUM

torch.einsum(*equation*, **operands*) → Tensor [SOURCE]

Sums the product of the elements of the input `operands` along dimensions specified using a notation based on the Einstein summation convention.

Einsum allows computing many common multi-dimensional linear algebraic array operations by representing them in a short-hand format based on the Einstein summation convention, given by `equation`. The details of this format are described below, but the general idea is to label every dimension of the input `operands` with some subscript and define which subscripts are part of the output. The output is then computed by summing the product of the elements of the `operands` along the dimensions whose subscripts are not part of the output. For example, matrix multiplication can be computed using einsum as *torch.einsum("ij,jk->ik", A, B)*. Here, j is the summation subscript and i and k the output subscripts (see section below for more details on why).

Equation:

The `equation` string specifies the subscripts (letters in [a-zA-Z]) for each dimension of the input `operands` in the same order as the dimensions, separating subscripts for each operand by a comma (','), e.g. *'ij,jk'* specify subscripts for two 2D operands. The dimensions labeled with the same subscript must be broadcastable, that is, their size must either match or be *1*. The exception is if a subscript is repeated for the same input operand, in which case the dimensions labeled with this subscript for this operand must match in size and the operand will be replaced by its diagonal along these dimensions. The subscripts that appear exactly once in the `equation` will be part of the output, sorted in increasing alphabetical order. The output is computed by multiplying the input `operands` element-wise, with their dimensions aligned based on the subscripts, and then summing out the dimensions whose subscripts are not part of the output.

Optionally, the output subscripts can be explicitly defined by adding an arrow ('->') at the end of the equation followed by the subscripts for the output. For instance, the following equation computes the transpose of a matrix multiplication: 'ij,jk->ki'. The output subscripts must appear at least once for some input operand and at most once for the output.

Ellipsis ('...') can be used in place of subscripts to broadcast the dimensions covered by the ellipsis. Each input operand may contain at most one ellipsis which will cover the dimensions not covered by subscripts, e.g. for an input operand with 5 dimensions, the ellipsis in the equation *'ab...c'* cover the third and fourth dimensions. The ellipsis does not need to cover the same number of dimensions across the `operands` but the 'shape' of the ellipsis (the size of the dimensions covered by them) must broadcast together. If the output is not explicitly defined with the arrow ('->') notation, the ellipsis will come first in the output (left-most dimensions), before the subscript labels that appear exactly once for the input operands. e.g. the following equation implements batch matrix multiplication *'...ij,...jk'*.

A few final notes: the equation may contain whitespaces between the different elements (subscripts, ellipsis, arrow and comma) but something like '...' is not valid. An empty string '' is valid for scalar operands.

RELAXED
SYSTEM LAB

Empirical Risk

Define the Empirical Risk

RELAXED
SYSTEM LAB

- Suppose we have:
  - a dataset $\mathcal{D} = \{(x_1, y_1), (x_1, y_2), \ldots, (x_N, y_N)\}$, where
  - $x_i \in \mathcal{X}$ is the input and
  - $y_i \in \mathcal{Y}$ is the output.
  - Let $h: \mathcal{X} \to \mathcal{Y}$ be a hypothesized model (mapping from input to output) we are trying to evaluate, which is parameterized by $w \in \mathbb{R}^d$.
  - Let $L: \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_+$ 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

RELAXED
SYSTEM LAB

- Mean squared error loss.
- L1 Loss.
- Negative log-likelihood loss.
- Cross entropy loss.
- KL divergence loss.

Mean Squared Error Loss

RELAXED
SYSTEM LAB

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:

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

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

RELAXED
SYSTEM LAB

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:

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

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

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

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

RELAXED SYSTEM LAB

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

RELAXED SYSTEM LAB

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

RELAXED
SYSTEM LAB

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_{pred}, y_{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

RELAXED
SYSTEM LAB

- Don't just want to calculate the empirical risk;
- Let $f\colon \mathbb{R}^d \to \mathbb{R}_+$ 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;
- **The training computation is solving the following optimization problem:**

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

Gradient Descent

Gradient Descent Algorithm

RELAXED SYSTEM LAB

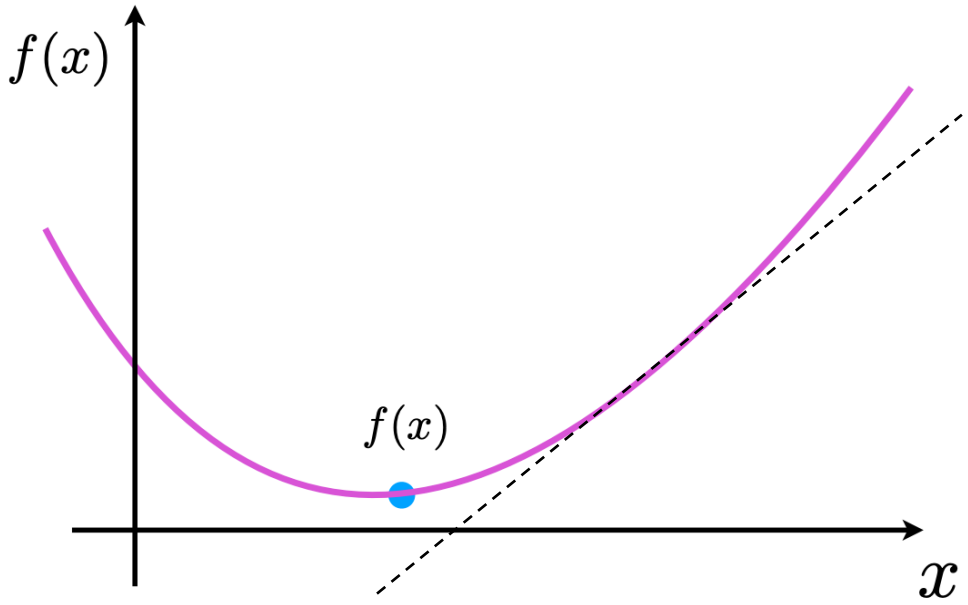
- Suppose we have:
  - w_0 denotes the value of the initalized parameter;
  - w_t denotes the value of the parameter at iteration t;
  - $\alpha_t \in \mathbb{R}$ denotes the learning rate at iteration t;
  - ∇f denotes the gradient (*vector of partial derivatives*) of function f.
- **The gradient decent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

RELAXED
SYSTEM LAB

- 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

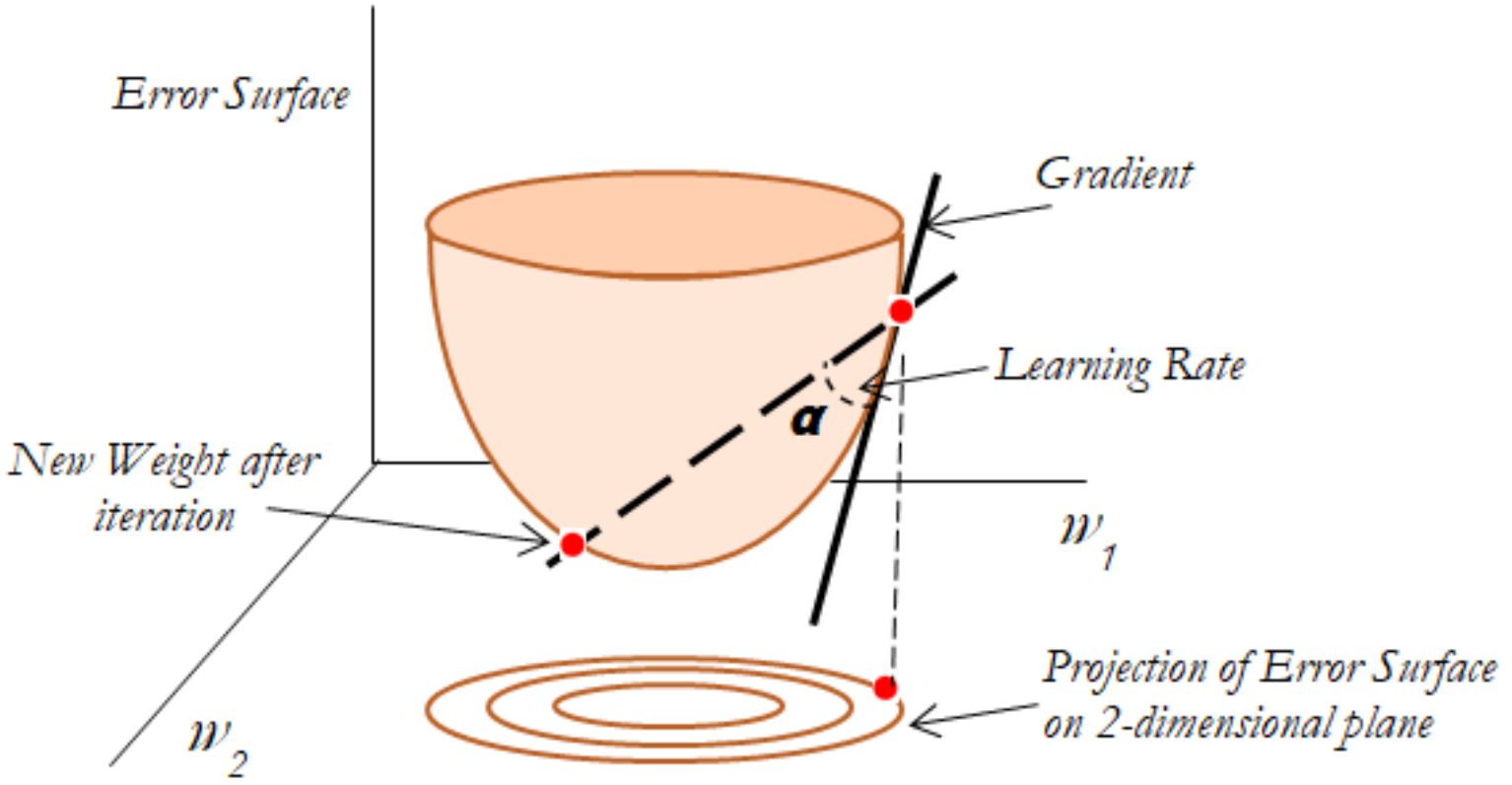
RELAXED
SYSTEM LAB

- Then, suppose we have:
  - $f: \mathbb{R}^d \rightarrow \mathbb{R}$;
- 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?

RELAXED SYSTEM LAB

- **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.



Stochastic Gradient Descent

Basic Idea

RELAXED
SYSTEM LAB

- 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.

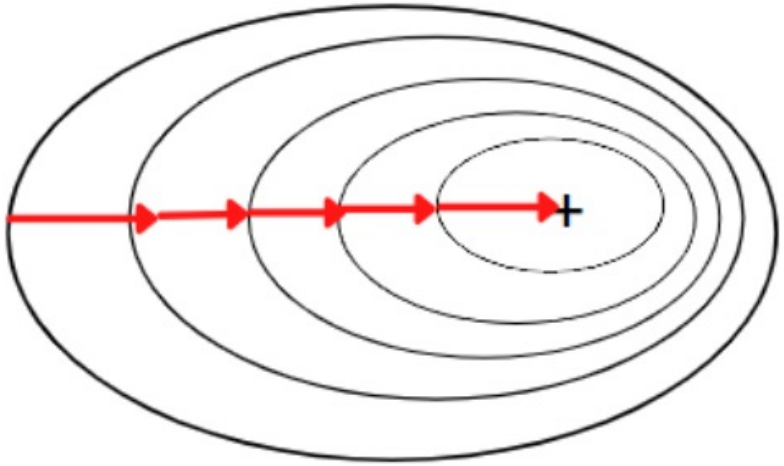
$$w_{t+1} = w_t - \alpha_t \cdot \nabla f(w_t)$$

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

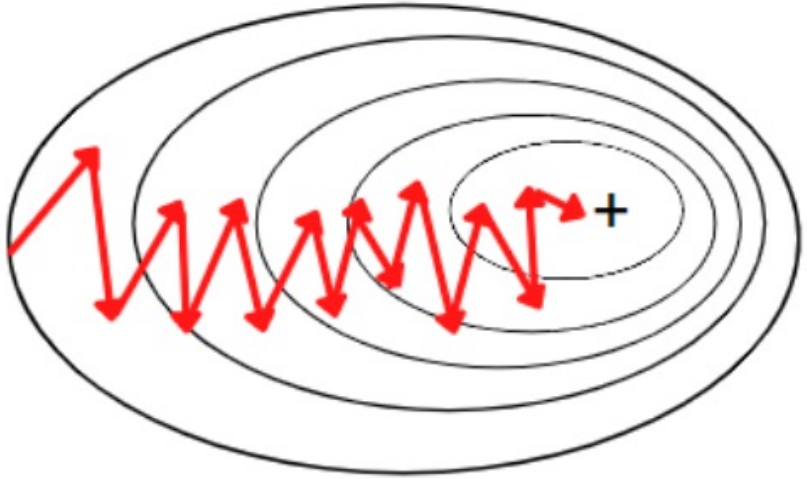
Stochastic Gradient Descent

RELAXED SYSTEM LAB

- 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

RELAXED SYSTEM LAB

- 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:

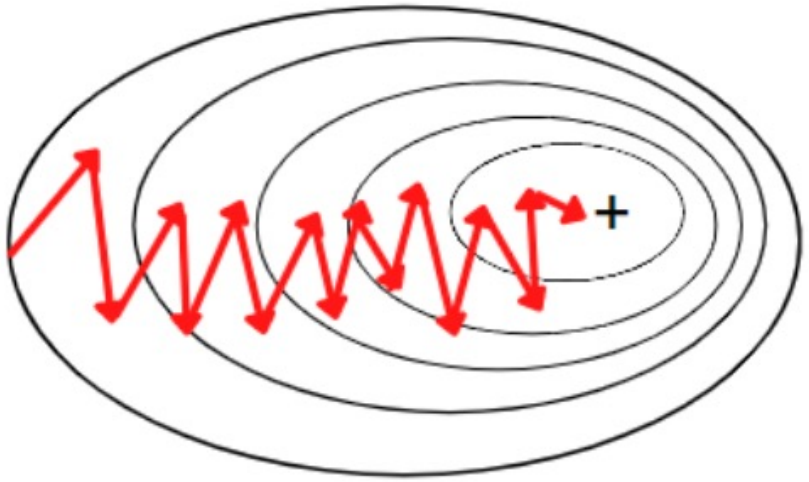
$$w_{t+1} = w_t - \alpha_t \cdot \nabla f(w_t; \xi_t)$$

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

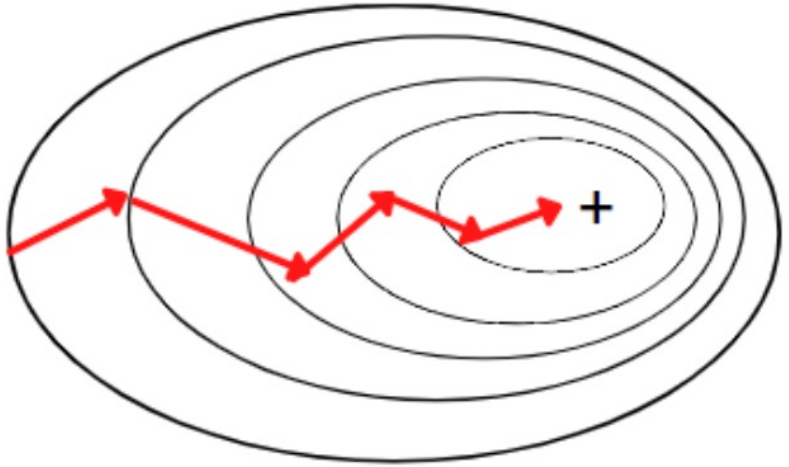
Mini-Batch Stochastic Gradient Descent

RELAXED SYSTEM LAB

- 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

RELAXED SYSTEM LAB

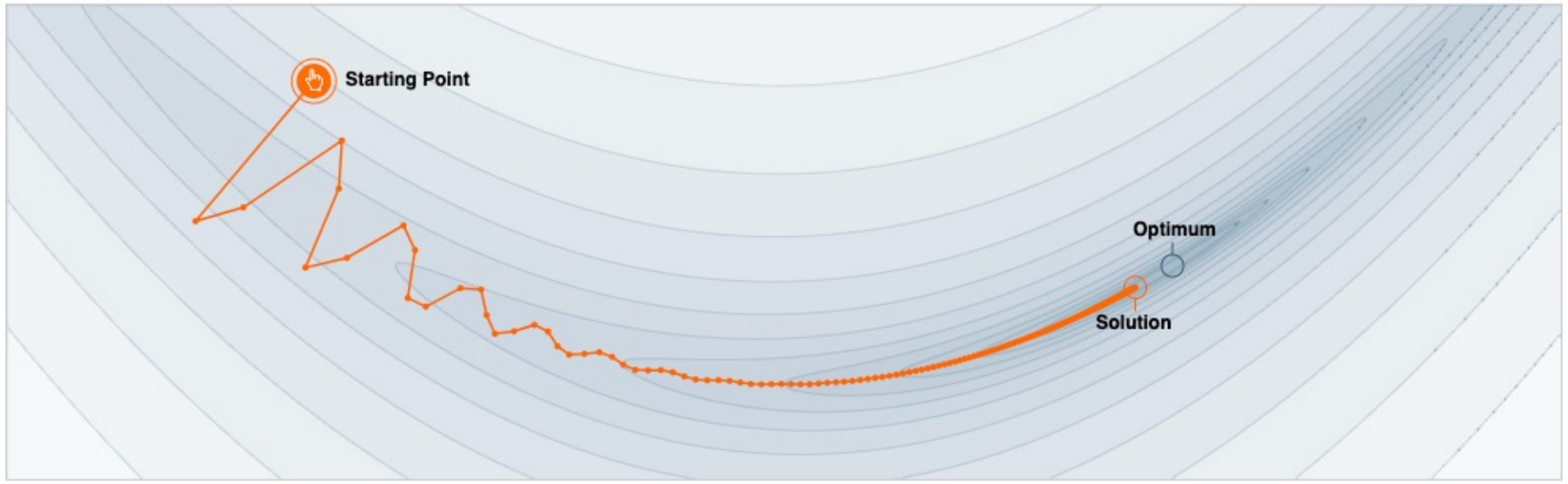
- Basic idea:
  - In SGD or mini-batch SGD the updates at each step is only based on current gradients, which can be unstable.
  - Momentum: exponentially weighted average of gradients.
  - The moving average method should be able to *denoise* the gradients computed at each step.
- Formal equation:

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

Standard gradient step Momentum step

Why Momentum Really Works?

RELAXED
SYSTEM LAB



Step-size α = 0.02

0 0.003 0.006

Momentum β = 0.99

0.00 0.500 0.990

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

RELAXED SYSTEM LAB

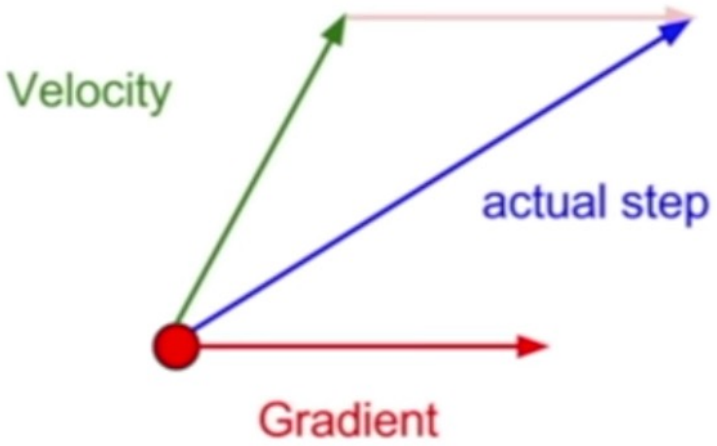
- Basic idea:
  - Polyak's momentum algorithm can fail to converge for some carefully built convex optimization problems.
  - Nesterov's Momentum: evaluates the gradient after applying momentum (at a point closer to the minimum point).
  - Works better for some cases in practice.
- Formal equation:

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

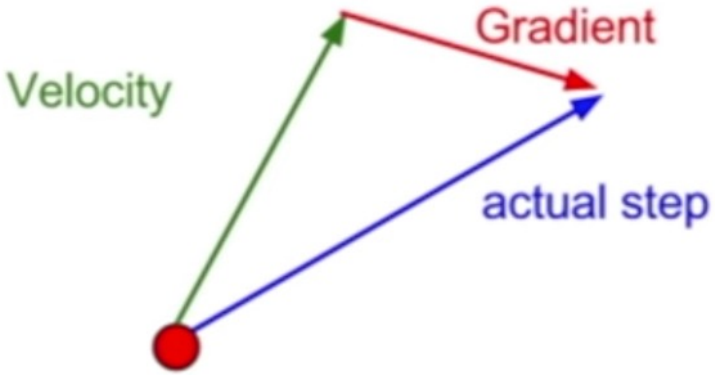
Estimate gradient after applying momentum — Momentum step

Polyak's Momentum vs. Nesterov's Momentum

RELAXED SYSTEM LAB



Polyak's Momentum



Nesterov's Momentum

SGD in PyTorch

RELAXED SYSTEM LAB

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 : γ (lr), θ_0 (params), $f(\theta)$ (objective), λ (weight decay), μ (momentum), τ (dampening), *nesterov*, *maximize*

for $t = 1$ **to** ... **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 θ_t

(Approximated) Second Order Method

Definition of Second-Order Derivative

RELAXED SYSTEM LAB

- 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

RELAXED SYSTEM LAB

- 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

RELAXED SYSTEM LAB

- 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

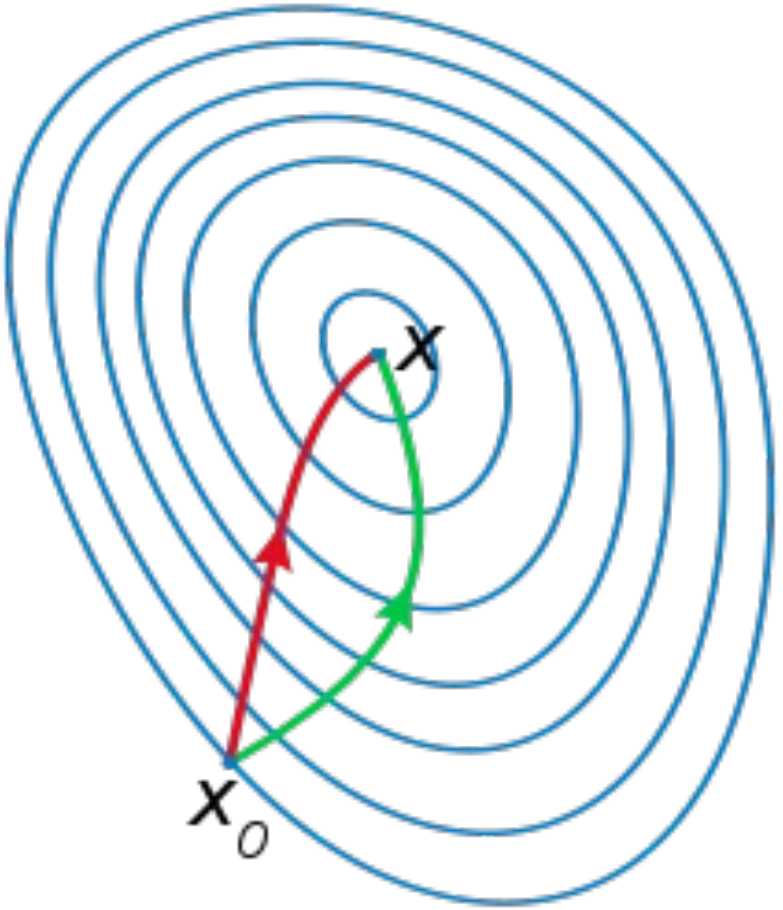
RELAXED
SYSTEM LAB

- 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.
- **The Newton's method is defined by**:

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

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)

RELAXED SYSTEM LAB

- 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

RELAXED SYSTEM LAB

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 : γ (lr), β_1, β_2 (betas), θ_0 (params), $f(\theta)$ (objective)
λ (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 $\lambda \neq 0$
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
  $\widehat{m_t} \leftarrow m_t / \left(1-\beta_1^t\right)$
  $\widehat{v_t} \leftarrow v_t / \left(1-\beta_2^t\right)$
  if *amsgrad*
    $\widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t})$
    $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \left(\sqrt{\widehat{v_t}^{max}} + \epsilon\right)$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \left(\sqrt{\widehat{v_t}} + \epsilon\right)$

return θ_t

Further Reading (Optional)

RELAXED SYSTEM LAB

Optimization Methods for Large-Scale Machine Learning

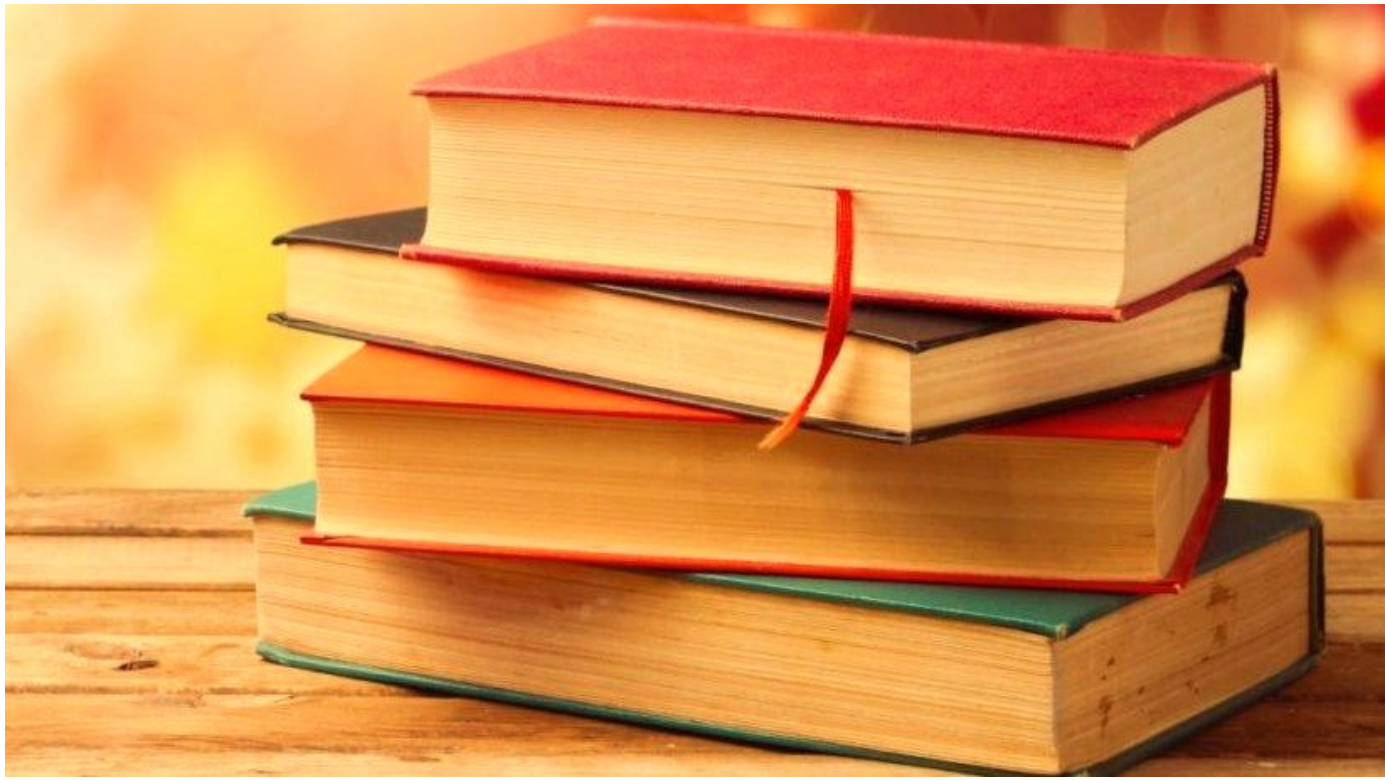
Léon Bottou* Frank E. Curtis† Jorge Nocedal‡

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

RELAXED SYSTEM LAB

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

RELAXED SYSTEM LAB

Define some hyper-parameters

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

The Frist Training Script

RELAXED
SYSTEM LAB

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

RELAXED SYSTEM LAB

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

RELAXED SYSTEM LAB

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

RELAXED SYSTEM LAB

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

RELAXED SYSTEM LAB

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

RELAXED
SYSTEM LAB

Put them together and visualize the results

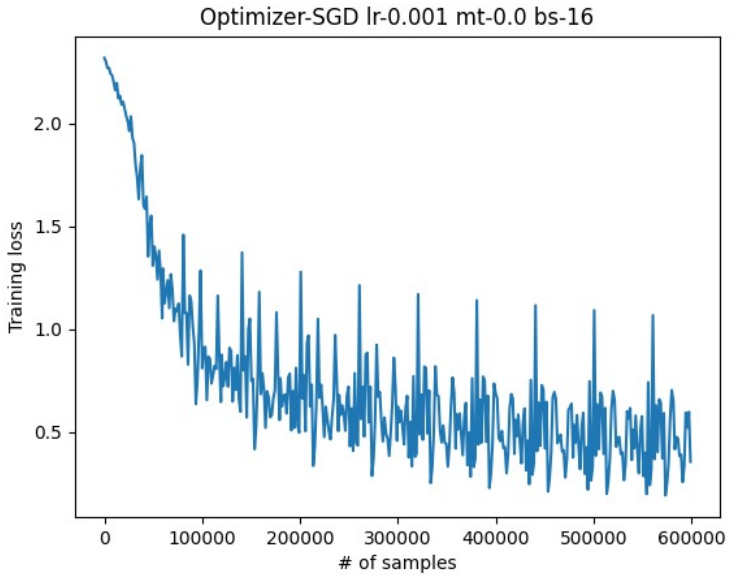
```
log = {'loss': [], '#samples': []}
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    epoch_log = train_loop(train_dataloader, model, loss_fn, optimizer)
    log['loss'].extend(epoch_log['loss'])
    log['#samples'].extend([ v+60000*t for v in epoch_log['#samples']])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

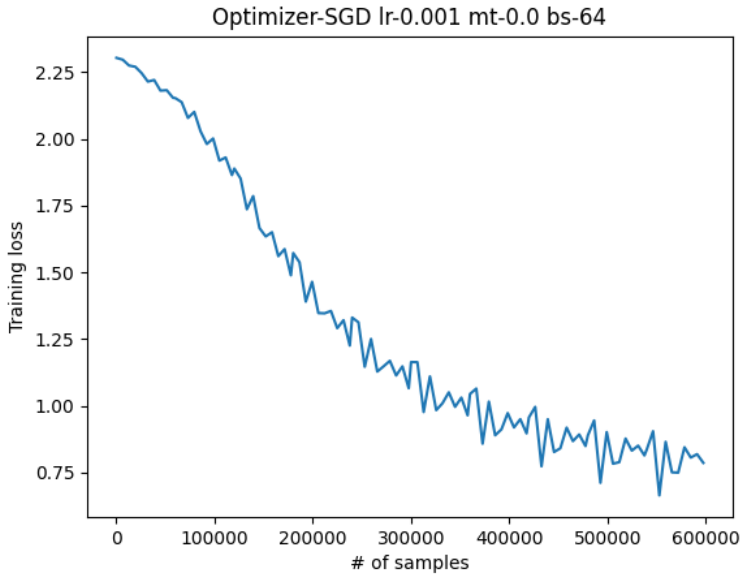
RELAXED
SYSTEM LAB

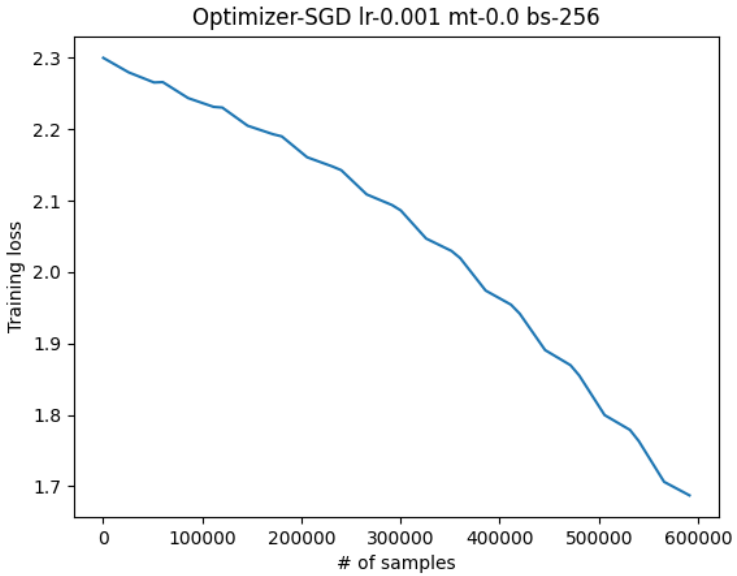
Demo & Results

SGD: Batch Size

RELAXED
SYSTEM LAB





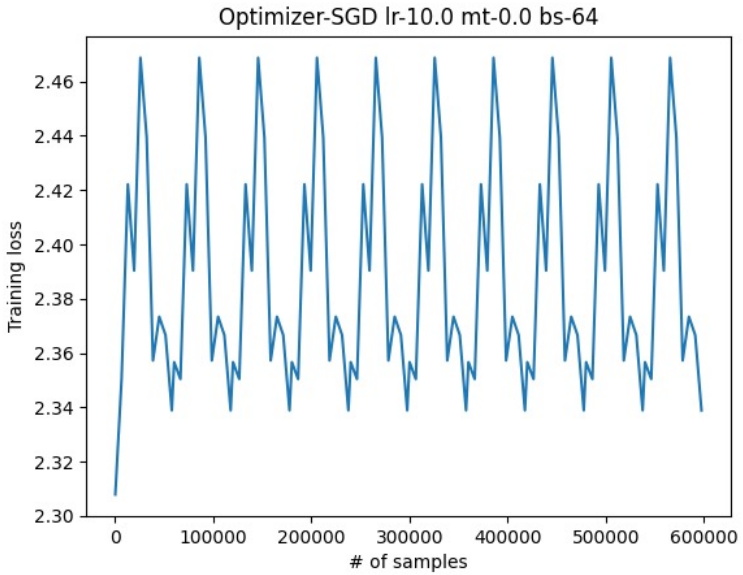


Batch size: 16

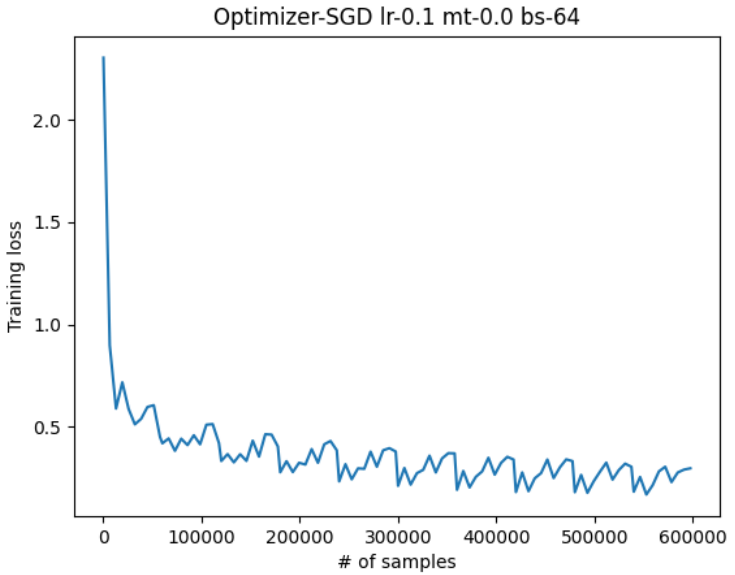
Batch size: 64

Batch size: 256

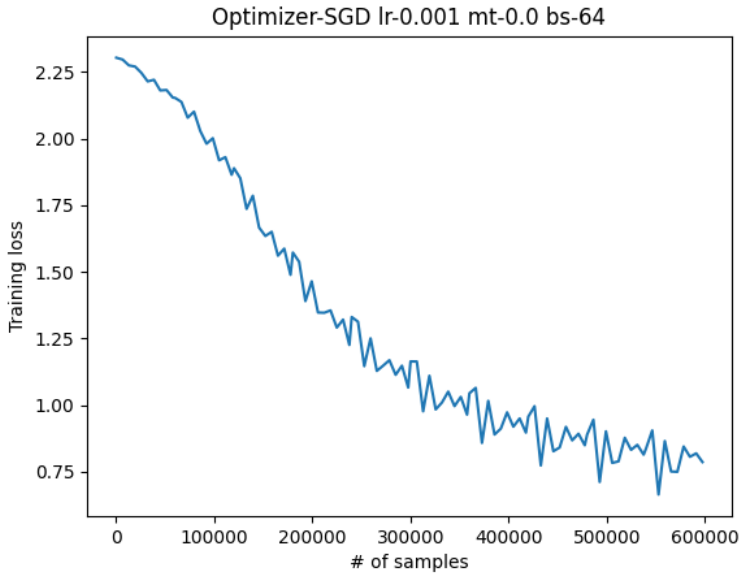
SGD: Learning Rate



Learning rate:10



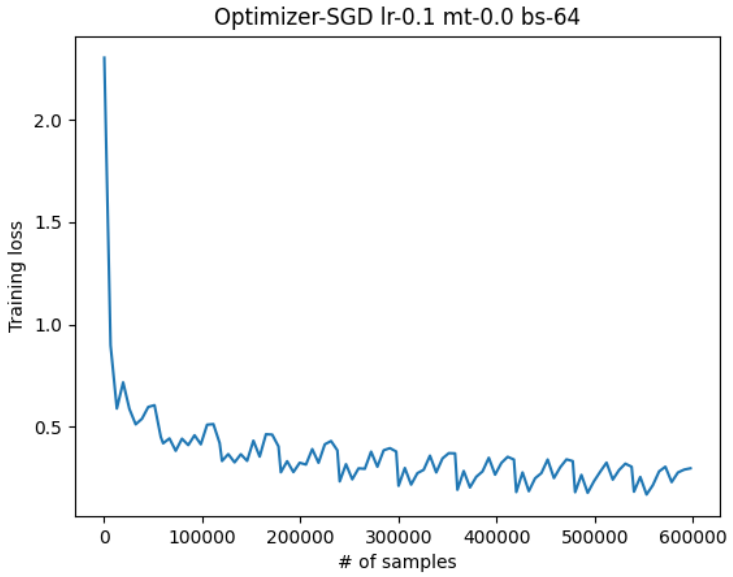
Learning rate: 0.1



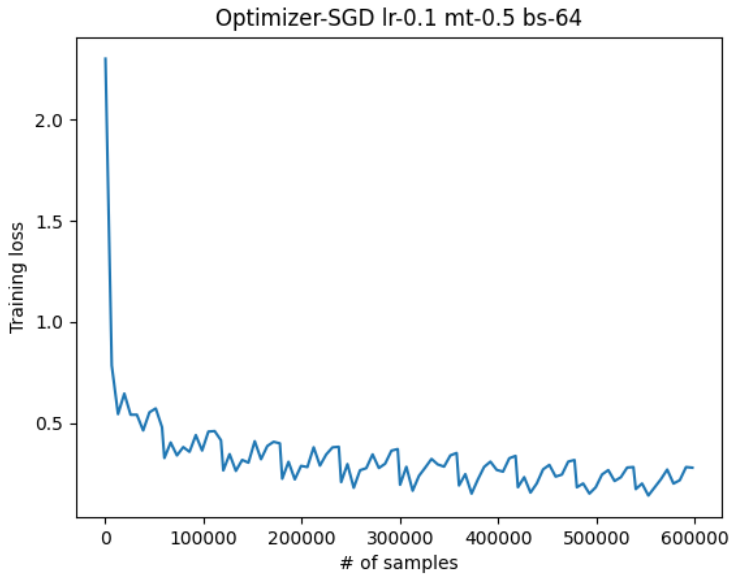
Learning rate: 0.001

SGD: Momentum

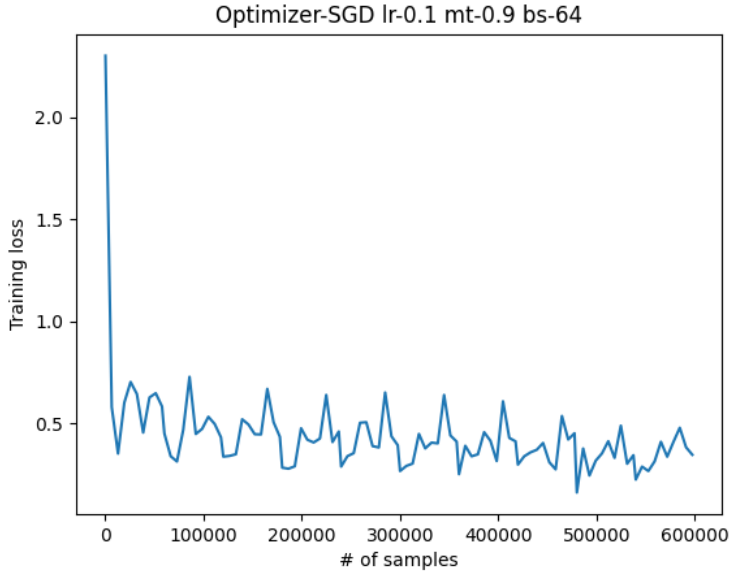
RELAXED
SYSTEM LAB



Momentum: 0



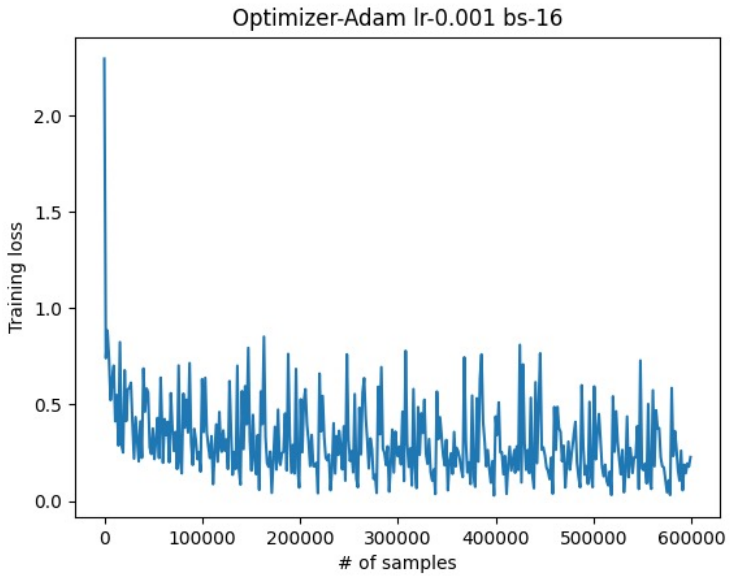
Momentum: 0.5



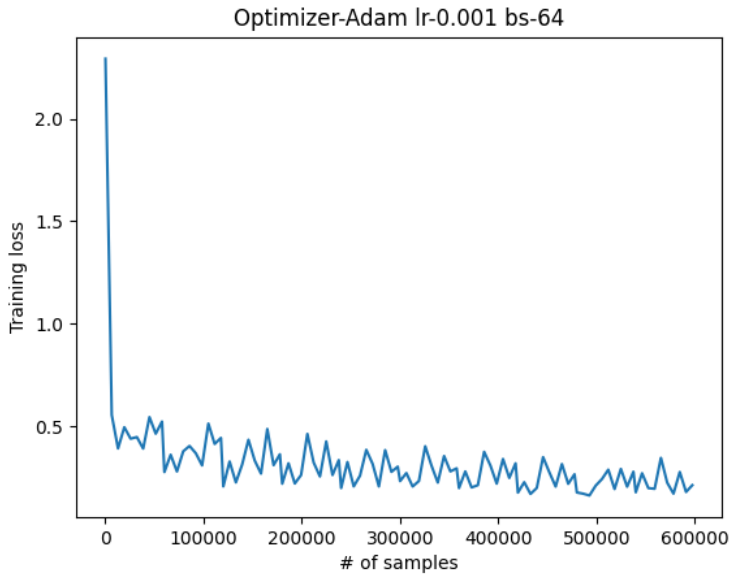
Momentum: 0.9

Adam: Batch Size

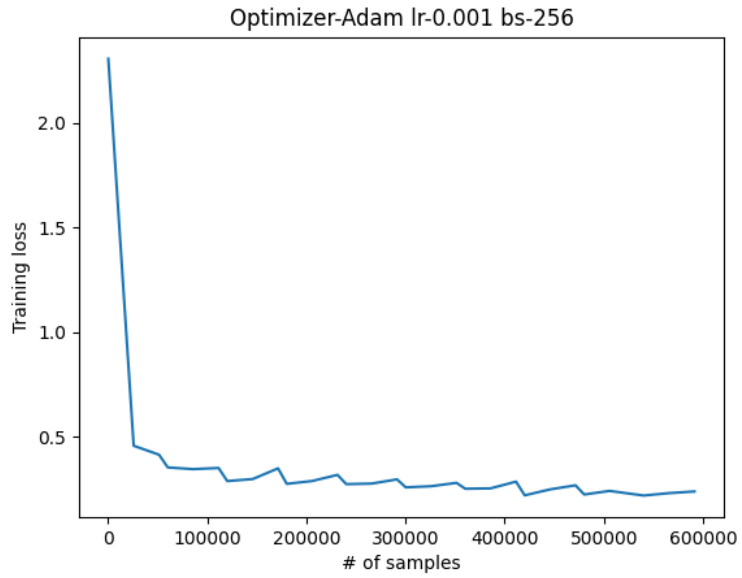
RELAXED
SYSTEM LAB



Batch size: 16



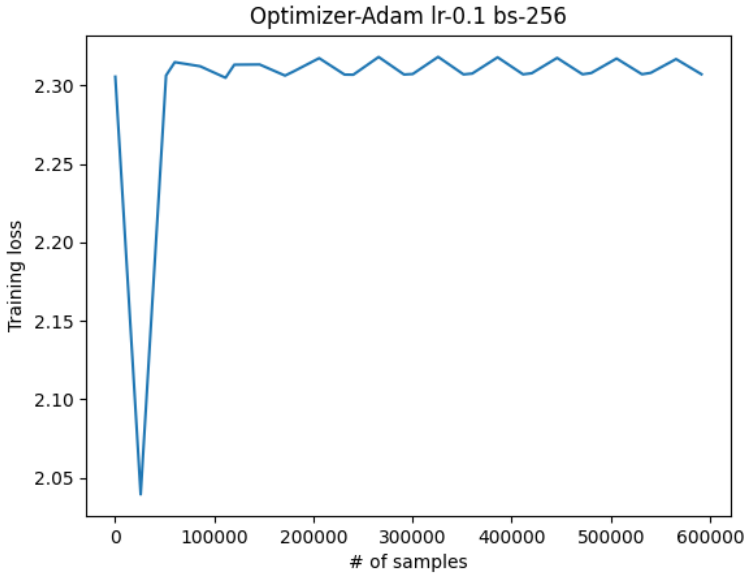
Batch size: 64



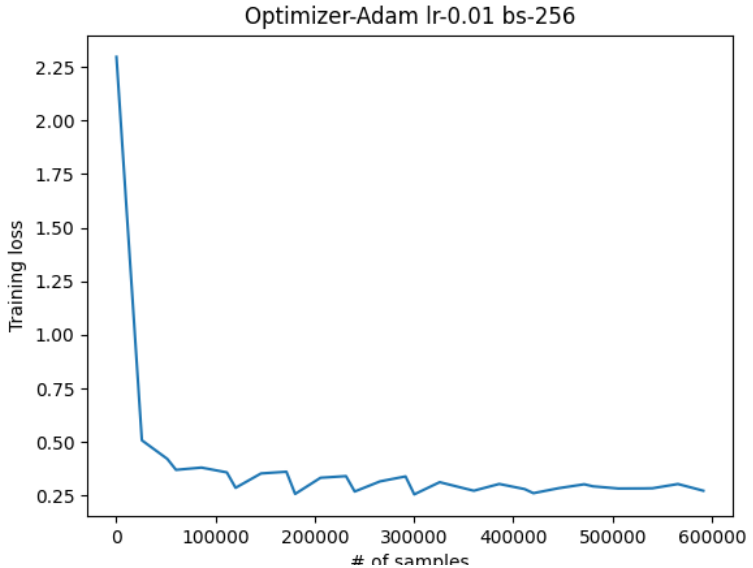
Batch size: 256

Adam: Learning Rate

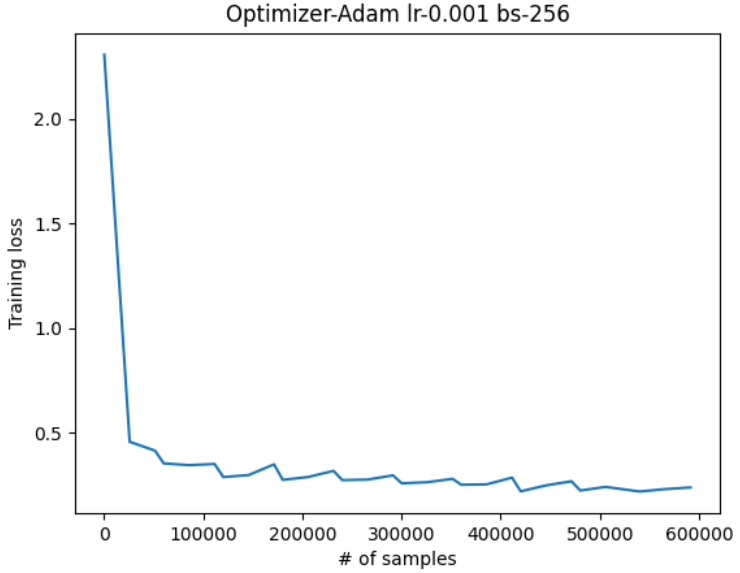
RELAXED SYSTEM LAB



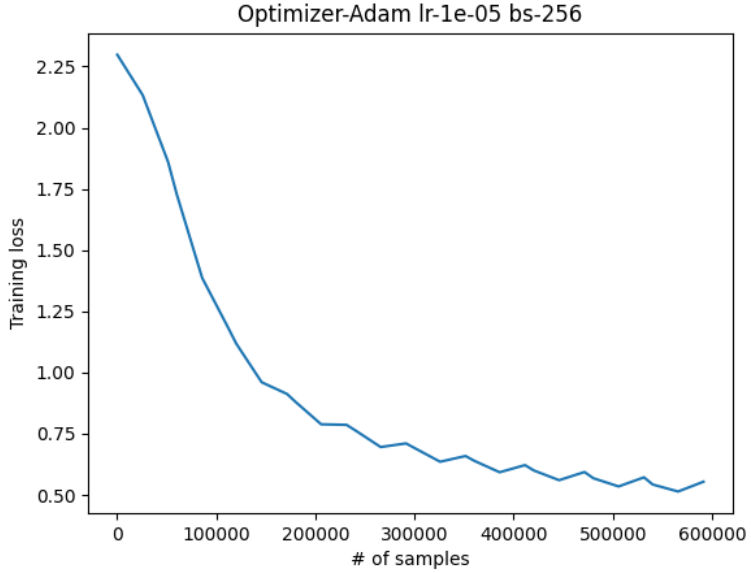
LR:0.1



LR:0.01



LR:0.001



LR:0.00001

References

- https://d2l.ai/
- https://www.dr-qubit.org/teaching/summation_delta.pdf
- https://pytorch.org/tutorials/beginner/introyt/tensors_deeper_tutorial.html
- 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/