Backpropagation, SGD, and Adam

Deep Learning (ENGG*6600*07)

School of Engineering,
University of Guelph, ON, Canada

Course Instructor: Benyamin Ghojogh
Fall 2023

Gradient Descent

Gradient descent: introduction

- **Gradient descent** is one of the fundamental first-order methods.
- It was first suggested by Cauchy in 1874 [1] and Hadamard in 1908 [2] and its convergence was later analyzed in [3].
- Unconstrained optimization:
$$\underset{\boldsymbol{x}}{\text{minimize}} \quad f(\boldsymbol{x}). \tag{1}$$
- In numerical optimization for unconstrained optimization, we start with a random feasible initial point and iteratively update it by step $\Delta\boldsymbol{x}$:
$$\boldsymbol{x}^{(k+1)} := \boldsymbol{x}^{(k)} + \Delta\boldsymbol{x}. \tag{2}$$
- Continue until we converge to (or get sufficiently close to) the desired optimal point $\boldsymbol{x}^*$.
- The step $\Delta\boldsymbol{x}$ is also denoted by $\boldsymbol{p}$ in the literature, i.e., $\boldsymbol{p} := \Delta\boldsymbol{x}$.

Gradient descent: update

- Assume the gradient of function $f(\boldsymbol{x})$ is L-smooth where L is the Lipschitz constant. In gradient descent, the update at every iteration is (see my "Optimization Techniques" course for proof):

$$\Delta \boldsymbol{x} = -\frac{1}{L}\nabla f(\boldsymbol{x}^{(k)}) \implies \boldsymbol{x}^{(k+1)} := \boldsymbol{x}^{(k)} - \frac{1}{L}\nabla f(\boldsymbol{x}^{(k)}). \quad (3)$$

- The problem is that often we either do not know the Lipschitz constant L or it is hard to compute. Hence, rather than $\Delta \boldsymbol{x} = -\frac{1}{L}\nabla f(\boldsymbol{x}^{(k)})$, we use:

$$\Delta \boldsymbol{x} = -\eta\nabla f(\boldsymbol{x}^{(k)}), \text{ i.e., } \boldsymbol{x}^{(k+1)} := \boldsymbol{x}^{(k)} - \eta\nabla f(\boldsymbol{x}^{(k)}), \quad (4)$$

  where $\eta > 0$ is the **step size**, also called the **learning rate** in data science literature.
- If the optimization problem is maximization rather than minimization, the step should be $\Delta \boldsymbol{x} = \eta\nabla f(\boldsymbol{x}^{(k)})$ rather than Eq. (4). In that case, the name of method is **gradient ascent**.
- The learning rate can be found by **line search** (see my "Optimization Techniques" course for more information), which is used often in optimization and not in deep learning.

Gradient descent: series of solutions

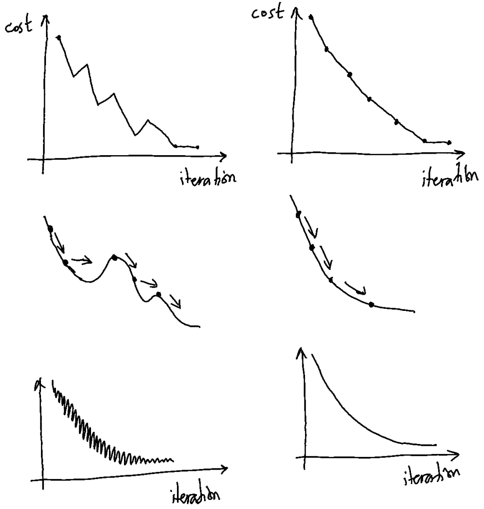
- For a convex function, the series of solutions converges to the optimal solution while the function value decreases iteratively until the local minimum:

$$\begin{aligned}&\{\boldsymbol{x}^{(0)}, \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \dots\} \to \boldsymbol{x}^*,\\ &f(\boldsymbol{x}^{(0)}) \geq f(\boldsymbol{x}^{(1)}) \geq f(\boldsymbol{x}^{(2)}) \geq \cdots \geq f(\boldsymbol{x}^*).\end{aligned}$$

- If the optimization problem is a convex problem, the solution is the global solution; otherwise, the solution is local.

Gradient descent: cost versus iterations

$$\{\boldsymbol{x}^{(0)}, \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \dots\} \to \boldsymbol{x}^*,$$
$$f(\boldsymbol{x}^{(0)}) \geq f(\boldsymbol{x}^{(1)}) \geq f(\boldsymbol{x}^{(2)}) \geq \cdots \geq f(\boldsymbol{x}^*).$$



Convergence criterion

Convergence criteria

- For all numerical optimization methods including gradient descent, there exist several methods for convergence criterion to stop updating the solution and terminate optimization.
- Some of them are:
  - Small norm of gradient:

    $$\|\nabla f(\boldsymbol{x}^{(k+1)})\|_2 \leq \epsilon,$$

    where ϵ is a small positive number.
    - ★ The reason for this criterion is the first-order optimality condition (recall that at the local optimum, we have $\|\nabla f(\boldsymbol{x}^*)\|_2 = 0$).
    - ★ If the function is not convex, this criterion has the risk of stopping at a saddle point.
  - Small change of cost function:

    $$|f(\boldsymbol{x}^{(k+1)}) - f(\boldsymbol{x}^{(k)})| \leq \epsilon.$$

  - Small change of gradient of function:

    $$|\nabla f(\boldsymbol{x}^{(k+1)}) - \nabla f(\boldsymbol{x}^{(k)})| \leq \epsilon.$$

  - Reaching maximum desired number of iterations, denoted by

    $$k < \max_k.$$

Line-Search

Line-search

- We saw the step size of gradient descent requires knowledge of the Lipschitz constant for the smoothness of gradient. However, we may not know the exact Lipschitz constant. Hence, we can find the suitable step size η by a search which is named the **line-search**.
- In line-search of every optimization iteration, we start with $\eta = 1$ and if it does not satisfy:

$$f(\boldsymbol{x}^{(k)} + \Delta\boldsymbol{x}) - f(\boldsymbol{x}^{(k)}) < 0, \tag{5}$$

  with step $\Delta\boldsymbol{x} = -\eta\nabla f(\boldsymbol{x}^{(k)})$:

$$f(\boldsymbol{x}^{(k)} + \Delta\boldsymbol{x}) < f(\boldsymbol{x}^{(k)}) \implies f(\boldsymbol{x}^{(k)} - \eta\nabla f(\boldsymbol{x}^{(k)})) < f(\boldsymbol{x}^{(k)}), \tag{6}$$

  we halve it, $\eta \leftarrow \eta/2$.
- This halving step size is repeated until this equation is satisfied, i.e., until we have a decrease in the objective function. Note that this decrease will happen when the step size becomes small enough to satisfy (see my "Optimization Techniques" course for proof):

$$\eta < \frac{1}{L}. \tag{7}$$

- A more sophisticated line-search method is the **Armijo line-search** [4], also called the **backtracking line-search**. Another more sophisticated line-search is **Wolfe conditions** [5]. We will learn it later in the course. See my "Optimization Techniques" course for more information about these.

Gradient descent with line-search

The algorithm of gradient descent with line-search:

```
Initialize $\boldsymbol{x}^{(0)}$
for iteration $k = 0, 1, \dots$ do
    Initialize $\eta := 1$
    for iteration $\tau = 1, 2, \dots$ do
        Check line-search condition
        if not satisfied then
            $\eta \leftarrow \frac{1}{2} \times \eta$
        else
            $\boldsymbol{x}^{(k+1)} := \boldsymbol{x}^{(k)} - \eta \nabla f(\boldsymbol{x}^{(k)})$
            break the loop
    Check the convergence criterion
    if converged then
        return $\boldsymbol{x}^{(k+1)}$
```

As this algorithm shows, line-search has its own internal iterations inside every iteration of gradient descent.

Momentum

Gradient descent with momentum

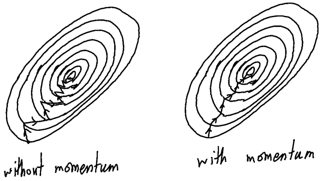
- Gradient descent and other first-order methods can have a momentum term. **Momentum**, proposed in [6], makes the change of solution $\Delta \boldsymbol{x}$ a little similar to the previous change of solution.
- Hence, the change adds a history of previous change to Eq. (4):

$$(\Delta \boldsymbol{x})^{(k)} := \alpha (\Delta \boldsymbol{x})^{(k-1)} - \eta^{(k)} \nabla f(\boldsymbol{x}^{(k)}), \tag{8}$$

  where $\alpha > 0$ is the momentum parameter which weights the importance of history compared to the descent direction.
- We use this $(\Delta \boldsymbol{x})^{(k)}$ in Eq. (2) for updating the solution:

$$\boldsymbol{x}^{(k+1)} := \boldsymbol{x}^{(k)} + (\Delta \boldsymbol{x})^{(k)}.$$

- Because of faithfulness to the track of previous updates, momentum reduces the amount of oscillation of updates in gradient descent optimization.



Steepest Descent

Steepest Descent

- **Steepest descent** is similar to gradient descent but there is a difference between them.
- In steepest descent, we move toward the negative gradient as much as possible to reach the smallest function value which can be achieved at every iteration.
- Hence, the step size at iteration k of steepest descent is calculated as [7]:

$$\eta^{(k)} := \arg\min_{\eta} f\big(\boldsymbol{x}^{(k)} - \eta \nabla f(\boldsymbol{x}^{(k)})\big), \tag{9}$$

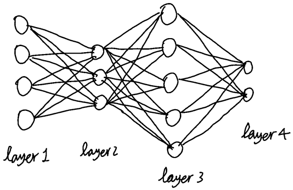
and then, the solution is updated using Eq. (4) as in gradient descent:

$$\boldsymbol{x}^{(k+1)} := \boldsymbol{x}^{(k)} - \eta \nabla f(\boldsymbol{x}^{(k)}).$$

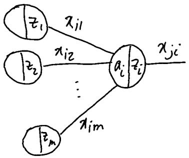
Backpropagation

Neural network

- Neural network:



- Every neuron in neural network:

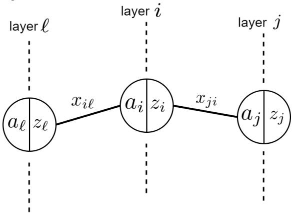


- Let x_{ji} denote the weight connecting neuron i to neuron j. Let a_i and z_i be the output of neuron i before and after applying its activation function $\sigma_i(.) : \mathbb{R} \to \mathbb{R}$, respectively.

$$a_i = \sum_{\ell=1}^{m} x_{i\ell} z_\ell, \qquad\qquad z_i := \sigma_i(a_i).$$

Backpropagation

- Consider three neurons in three layers of a network:



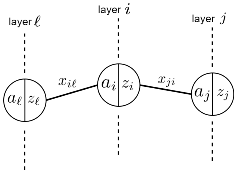
- We have $a_i = \sum_\ell x_{i\ell} z_\ell$ which sums over the neurons in layer ℓ. By chain rule, the gradient of error e w.r.t. to the weight between neurons ℓ and i is:

$$\frac{\partial e}{\partial x_{i\ell}} = \frac{\partial e}{\partial a_i} \times \frac{\partial a_i}{\partial x_{i\ell}} \overset{(a)}{=} \delta_i \times z_\ell, \tag{10}$$

where (a) is because $a_i = \sum_\ell x_{i\ell} z_\ell$ and we define:

$$\delta_i := \frac{\partial e}{\partial a_i}.$$

Backpropagation



- If layer i is the last layer, δ_i can be computed by derivative of error (loss function) w.r.t. the output.
- However, if i is one of the hidden layers, δ_i is computed by chain rule as:

$$\delta_i = \frac{\partial e}{\partial a_i} = \sum_j \Big(\frac{\partial e}{\partial a_j} \times \frac{\partial a_j}{\partial a_i}\Big) = \sum_j \Big(\delta_j \times \frac{\partial a_j}{\partial a_i}\Big). \qquad (11)$$

- The term $\partial a_j / \partial a_i$ is calculated by chain rule as:

$$\frac{\partial a_j}{\partial a_i} = \frac{\partial a_j}{\partial z_i} \times \frac{\partial z_i}{\partial a_i} \overset{(a)}{=} x_{ji}\, \sigma'(a_i), \qquad (12)$$

where (a) is because $a_j = \sum_i x_{ji} z_i$ and $z_i = \sigma(a_i)$ and $\sigma'(.)$ denotes the derivative of activation function. Putting Eq. (12) in Eq. (11) gives:

$$\delta_i = \sigma'(a_i) \sum_j (\delta_j\, x_{ji}).$$

Backpropagation

- We found:

$$\delta_i = \sigma'(a_i) \sum_j (\delta_j \, x_{ji}).$$

- Putting this equation in Eq. (10), $\frac{\partial e}{\partial x_{i\ell}} = \delta_i \times z_\ell$, gives:

$$\frac{\partial e}{\partial x_{i\ell}} = z_\ell \, \sigma'(a_i) \sum_j (\delta_j \, x_{ji}). \tag{13}$$

- **Backpropagation** uses the gradient in Eq. (13) for updating the weight $x_{i\ell}, \forall i, \ell$ by gradient descent:

$$x_{i\ell}^{(k+1)} := x_{i\ell}^{(k)} - \eta^{(k)} \frac{\partial e}{\partial x_{i\ell}}.$$

- This tunes the weights from last layer to the first layer for every iteration of optimization.
- Therefore, **backpropagation**, proposed in 1986 [6], is actually gradient descent with chain rule in derivatives because of having layers of parameters. It is the most well-known optimization method used for training neural networks.

Stochastic gradient methods

Stochastic gradient descent

- Assume we have a dataset of n data points, $\{\boldsymbol{a}_i \in \mathbb{R}^d\}_{i=1}^n$ and their labels $\{l_i \in \mathbb{R}\}_{i=1}^n$.
- Let the cost function $f(.)$ be decomposed into summation of n terms $\{f_i(\boldsymbol{x})\}_{i=1}^n$. Some well-known examples for the cost function terms are:
  - Least squares error: $f_i(\boldsymbol{x}) = 0.5(\boldsymbol{a}_i^\top \boldsymbol{x} - l_i)^2$,
  - Absolute error: $f_i(\boldsymbol{x}) = \boldsymbol{a}_i^\top \boldsymbol{x} - l_i$,
  - Hinge loss (for $l_i \in \{-1, 1\}$): $f_i(\boldsymbol{x}) = \max(0, 1 - l_i \boldsymbol{a}_i^\top \boldsymbol{x})$.
  - Logistic loss (for $l_i \in \{-1, 1\}$): $\log(\frac{1}{1+\exp(-l_i \boldsymbol{a}_i^\top \boldsymbol{x})})$.
- The optimization problem becomes:

$$\underset{\boldsymbol{x}}{\text{minimize}} \quad \frac{1}{n}\sum_{i=1}^{n} f_i(\boldsymbol{x}). \tag{14}$$

In this case, the full gradient is the average gradient, i.e:

$$\nabla f(\boldsymbol{x}) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\boldsymbol{x}), \tag{15}$$

so $\Delta \boldsymbol{x} = -\eta \nabla f(\boldsymbol{x}^{(k)})$, becomes $\Delta \boldsymbol{x} = -(\eta/n)\sum_{i=1}^n \nabla f_i(\boldsymbol{x}^{(k)})$. This is what gradient descent uses for updating the solution at every iteration.

Stochastic gradient descent

$$\nabla f(\boldsymbol{x}) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\boldsymbol{x}),$$

- Calculation of this full gradient is time-consuming and inefficient for large values of n, especially as it needs to be recalculated at every iteration.
- **Stochastic Gradient Descent (SGD)**, also called **stochastic gradient method**, approximates gradient descent stochastically and samples (i.e. bootstraps) one of the points at every iteration for updating the solution. Hence, it uses:

$$\boldsymbol{x}^{(k+1)} := \boldsymbol{x}^{(k)} - \eta^{(k)} \nabla f_i(\boldsymbol{x}^{(k)}), \tag{16}$$

  rather than Eq. (4), $\boldsymbol{x}^{(k+1)} := \boldsymbol{x}^{(k)} - \eta \nabla f(\boldsymbol{x}^{(k)})$.
- The idea of stochastic approximation was first proposed in 1951 [8]. It was first used for machine learning in 1998 [9].
- As Eq. (16) states, SGD often uses an adaptive step size which changes in every iteration. The step size can be decreasing because in initial iterations, where we are far away from the optimal solution, the step size can be large; however, it should be small in the last iterations which is supposed to be close to the optimal solution. Some well-known adaptations for the step size are:

$$\eta^{(k)} := \frac{1}{k}, \quad \eta^{(k)} := \frac{1}{\sqrt{k}}, \quad \eta^{(k)} := \eta. \tag{17}$$

Convergence Rate of Gradient Descent

- Consider a convex and differentiable function $f(.)$, with domain $\mathcal{D}$, whose gradient is L-smooth. Let f^* be the minimum of cost function and $\boldsymbol{x}^*$ be the minimizer. Starting from the initial point $\boldsymbol{x}^{(0)}$, after t iterations of the optimization algorithm, we will have the following.
- The convergence rate of gradient descent:

$$f(\boldsymbol{x}^{(t+1)}) - f^* \leq \frac{2L\|\boldsymbol{x}^{(0)} - \boldsymbol{x}^*\|_2^2}{t+1} = \mathcal{O}(\frac{1}{t}). \tag{18}$$

Convergence Rate of Stochastic Gradient Descent

- Consider a function $f(\boldsymbol{x}) = \sum_{i=1}^{n} f_i(\boldsymbol{x})$ and which is bounded below and each f_i is differentiable. Let the domain of function $f(.)$ be $\mathcal{D}$ and its gradient be L-smooth. Assume $\mathbb{E}[\|\nabla f_i(\boldsymbol{x}_k)\|_2^2 \,|\, \boldsymbol{x}_k] \leq \beta^2$ where β is a constant. Assume $\mathbb{E}[\|\nabla f_i(\boldsymbol{x}_k)\|_2^2 \,|\, \boldsymbol{x}_k] \leq \beta^2$ where β is a constant.
- Depending on the step size, the convergence rate of SGD is:

$$f(\boldsymbol{x}^{(t+1)}) - f^* \leq \mathcal{O}(\frac{1}{\log t}) \quad \text{if} \quad \eta^{(\tau)} = \frac{1}{\tau}, \tag{19}$$

$$f(\boldsymbol{x}^{(t+1)}) - f^* \leq \mathcal{O}(\frac{\log t}{\sqrt{t}}) \quad \text{if} \quad \eta^{(\tau)} = \frac{1}{\sqrt{\tau}}, \tag{20}$$

$$f(\boldsymbol{x}^{(t+1)}) - f^* \leq \mathcal{O}(\frac{1}{t} + \eta) \quad \text{if} \quad \eta^{(\tau)} = \eta, \tag{21}$$

  where τ denotes the iteration index.
- If the functions f_i's are μ-strongly convex, then the convergence rate of SGD is:

$$f(\boldsymbol{x}^{(t+1)}) - f^* \leq \mathcal{O}(\frac{1}{t}) \quad \text{if} \quad \eta^{(\tau)} = \frac{1}{\mu\tau}, \tag{22}$$

$$f(\boldsymbol{x}^{(t+1)}) - f^* \leq \mathcal{O}\big((1 - \frac{\mu}{L})^t + \eta\big) \quad \text{if} \quad \eta^{(\tau)} = \eta. \tag{23}$$

Analysis of convergence rates

- Recall Eqs. (21) and (23):

$$\text{convex or non-convex:} \quad \mathcal{O}(\frac{1}{t} + \eta) \quad \text{if} \quad \eta^{(\tau)} = \eta,$$
$$\text{strongly convex:} \quad \mathcal{O}\big((1 - \frac{\mu}{L})^t + \eta\big) \quad \text{if} \quad \eta^{(\tau)} = \eta.$$

- These equations show that with a fixed step size η, SGD converges sublinearly for a non-convex function and exponentially for a strongly convex function in the initial iterations.
- However, in the late iterations, it stagnates to a neighborhood around the optimal point and never reaches it. Hence, SGD has less accuracy than gradient descent (whose convergence rate is $\mathcal{O}(\frac{1}{t})$ as in Eq. (18)).
- The advantage of SGD over gradient descent is that its every iteration is much faster than every iteration of gradient descent because of less computations for gradient. This faster pacing of every iteration shows off more when n is huge.
- In summary, SGD has fast convergence to a low accurate optimal solution.
- It is noteworthy that the full gradient is not available in SGD to use for checking convergence, as discussed before. One can use other criteria or merely check the norm of gradient for the sampled point.
- SGD can be used with the line-search methods, too. SGD can also use a momentum term.

Mini-batch stochastic gradient descent

- Gradient descent uses the entire n data points and SGD uses one randomly sampled point at every iteration. For large datasets, gradient descent is very slow and intractable in every iteration while SGD will need a significant number of iterations to roughly cover all data. Besides, SGD has low accuracy in convergence to the optimal solution.
- We can have a middle case where we use a batch of b randomly sampled points at every iteration. This method is named the **mini-batch SGD** or the **hybrid deterministic-stochastic gradient** method. This batch-wise approach is wise for large datasets.
- Usually, before start of optimization, the n data points are randomly divided into $\lfloor n/b \rfloor$ batches of size b. This is equivalent to simple random sampling for sampling points into batches without replacement. We denote the dataset by $\mathcal{D}$ (where $|\mathcal{D}| = n$) and the i-th batch by $\mathcal{B}_i$ (where $|\mathcal{B}_i| = b$). The batches are disjoint:

$$\bigcup_{i=1}^{\lfloor n/b \rfloor} \mathcal{B}_i = \mathcal{D}, \tag{24}$$

$$\mathcal{B}_i \cap \mathcal{B}_j = \varnothing, \quad \forall i, j \in \{1, \dots, \lfloor n/b \rfloor\},\ i \neq j. \tag{25}$$

- Another less-used approach for making batches is to sample points for a batch during optimization. This is equivalent to bootstrapping for sampling points into batches with replacement. In this case, the batches are not disjoint anymore and Eqs. (24) and (25) do not hold.

Mini-batch stochastic gradient descent

Definition (Epoch)

In mini-batch SGD, when all $\lfloor n/b \rfloor$ batches of data are used for optimization once, an **epoch** is completed. After completion of an epoch, the next epoch is started and epochs are repeated until convergence of optimization.

- In mini-batch SGD, if the k-th iteration of optimization is using the k'-th batch, the update of solution is done as:

$$\boldsymbol{x}^{(k+1)} := \boldsymbol{x}^{(k)} - \eta^{(k)} \frac{1}{b} \sum_{i \in \mathcal{B}_{k'}} \nabla f_i(\boldsymbol{x}^{(k)}). \tag{26}$$

- The scale factor $1/b$ is sometimes dropped for simplicity.
- Mini-batch SGD is used significantly in machine learning, especially in neural networks [9, 10].
- Because of dividing data into batches, mini-batch SGD can be solved on parallel servers as a distributed optimization method.

Mini-batch stochastic gradient descent

Theorem (Convergence rates for mini-batch SGD)

Consider a function $f(\boldsymbol{x}) = \sum_{i=1}^{n} f_i(\boldsymbol{x})$ which is bounded below and each f_i is differentiable. Let the domain of function $f(.)$ be $\mathcal{D}$ and its gradient be L-smooth and assume $\eta^{(k)} = \eta$ is fixed. The batch-wise gradient is an approximation to the full gradient with some error e_t for the t-th iteration:

$$\frac{1}{b} \sum_{i \in \mathcal{B}_{t'}} \nabla f_i(\boldsymbol{x}^{(t)}) = \nabla f(\boldsymbol{x}^{(t)}) + e_t. \tag{27}$$

The convergence rate of mini-batch SGD for non-convex and convex functions are:

$$\mathcal{O}\big(\frac{1}{t} + \|e_t\|_2^2\big), \tag{28}$$

where t denotes the iteration index. If the functions f_i's are μ-strongly convex, then the convergence rate of mini-batch SGD is:

$$\mathcal{O}\big((1 - \frac{\mu}{L})^t + \|e_t\|_2^2\big). \tag{29}$$

Therefore, the convergence rate of mini-batch gets closer to that of gradient descent, $\mathcal{O}(1/t)$, if the batch size increases.

Mini-batch stochastic gradient descent

- If we sample the batches without replacement (i.e., sampling batches by simple random sampling before start of optimization) or with replacement (i.e., bootstrapping during optimization), the expected error is [11, Proposition 3]:

$$\mathbb{E}[\|e_t\|_2^2] = (1 - \frac{b}{n})\frac{\sigma^2}{b}, \tag{30}$$

$$\mathbb{E}[\|e_t\|_2^2] = \frac{\sigma^2}{b}, \tag{31}$$

  respectively, where σ^2 is the variance of whole dataset.

- According to Eqs. (30) and (31), the accuracy of SGD by sampling without and with replacement increases by $b \to n$ and $b \to \infty$, respectively.
- However, this increase makes every iteration slower so there is trade-off between accuracy and speed.

Adaptive Learning Rate

Adaptive Gradient (AdaGrad)

- We can adapt the learning rate in stochastic gradient methods. Three most well-known methods for adapting the learning rate are AdaGrad, RMSProp, and Adam.
- **Adaptive Gradient (AdaGrad)** method, proposed in 2011 [12], updates the solution iteratively as:

$$\boldsymbol{x}^{(k+1)} := \boldsymbol{x}^{(k)} - \eta^{(k)} \boldsymbol{G}^{-1} \nabla f_i(\boldsymbol{x}^{(k)}), \tag{32}$$

where $\boldsymbol{G}$ is a $(d \times d)$ diagonal matrix whose (j, j)-th element is:

$$\boldsymbol{G}(j, j) := \sqrt{\varepsilon + \sum_{\tau=0}^{k} \big(\nabla_j f_{i_\tau}(\boldsymbol{x}^{(\tau)})\big)^2}, \tag{33}$$

where $\varepsilon \geq 0$ is for stability (making $\boldsymbol{G}$ full rank), i_τ is the randomly sampled point (from $\{1, \dots, n\}$) at iteration τ, and $\nabla_j f_{i_\tau}(.)$ is the partial derivative of $f_{i_\tau}(.)$ w.r.t. its j-th element (note that $f_{i_\tau}(.)$ is d-dimensional).
- Putting Eq. (33) in Eq. (32) can simplify AdaGrad to:

$$\boldsymbol{x}_j^{(k+1)} := \boldsymbol{x}_j^{(k)} - \frac{\eta^{(k)}}{\sqrt{\varepsilon + \sum_{\tau=0}^{k} \big(\nabla_j f_{i_\tau}(\boldsymbol{x}^{(\tau)})\big)^2}} \nabla f_j(\boldsymbol{x}_j^{(k)}). \tag{34}$$

- AdaGrad keeps a history of the sampled points and it takes derivative for them to use. During the iterations so far, if a dimension has changed significantly, it dampens the learning rate for that dimension (see the inverse in Eq. (32)); hence, it gives more weight for changing the dimensions which have not changed noticeably. In this way, all dimensions will have a fair chance to change.

Root Mean Square Propagation (RMSProp)

- **Root Mean Square Propagation (RMSProp)** was first proposed in 2012 [13] which is unpublished.
- It is an improved version of **Rprop (resilient backpropagation)**, proposed in 1992 [14], which uses the sign of gradient in optimization.
- Inspired by momentum in Eq. (8):

$$(\Delta \boldsymbol{x})^{(k)} := \alpha(\Delta \boldsymbol{x})^{(k-1)} - \eta^{(k)} \nabla f(\boldsymbol{x}^{(k)}),$$

  it updates a scalar variable v as [15]:

$$v^{(k+1)} := \gamma v^{(k)} + (1-\gamma)\|\nabla f_i(\boldsymbol{x}^{(k)})\|_2^2, \qquad (35)$$

  where $\gamma \in [0,1]$ is the forgetting factor (e.g., $\gamma = 0.9$). Then, it uses this v to weight the learning rate:

$$\boldsymbol{x}^{(k+1)} := \boldsymbol{x}^{(k)} - \frac{\eta^{(k)}}{\sqrt{\varepsilon + v^{(k+1)}}} \nabla f_j(\boldsymbol{x}_j^{(k)}), \qquad (36)$$

  where $\epsilon \geq 0$ is for stability not to have division by zero.
- Comparing Eqs. (34) and (36) shows that RMSProp has a similar form to AdaGrad.

Adaptive Moment Estimation (Adam)

- **Adam (Adaptive Moment Estimation) optimizer** [16] improves over RMSProp by adding a momentum term.
- It updates the scalar v and the vector $\boldsymbol{m} \in \mathbb{R}^d$ as:

$$\boldsymbol{m}^{(k+1)} := \gamma_1 \boldsymbol{m}^{(k)} + (1-\gamma_1)\nabla f_i(\boldsymbol{x}^{(k)}), \tag{37}$$

$$v^{(k+1)} := \gamma_2 v^{(k)} + (1-\gamma_2)\|\nabla f_i(\boldsymbol{x}^{(k)})\|_2^2, \tag{38}$$

  where $\gamma_1, \gamma_2 \in [0,1]$. It normalizes these variables as:

$$\widehat{\boldsymbol{m}}^{(k+1)} := \frac{1}{1-\gamma_1^k}\boldsymbol{m}^{(k+1)}, \quad \widehat{v}^{(k+1)} := \frac{1}{1-\gamma_2^k} v^{(k+1)}.$$

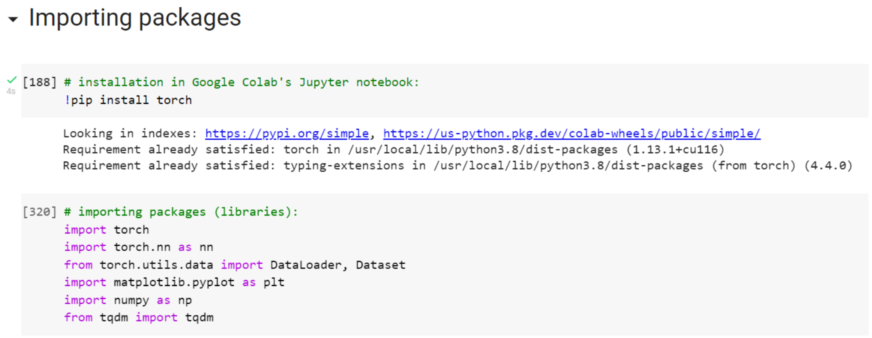
- Then, it updates the solution as:

$$\boldsymbol{x}^{(k+1)} := \boldsymbol{x}^{(k)} - \frac{\eta^{(k)}}{\sqrt{\varepsilon + \widehat{v}^{(k+1)}}}\widehat{\boldsymbol{m}}^{(k+1)}, \tag{39}$$

  which is stochastic gradient descent with momentum while using RMSProp.
- The Adam optimizer is one of the mostly used optimizers in **neural networks**.

Coding a Neural Network

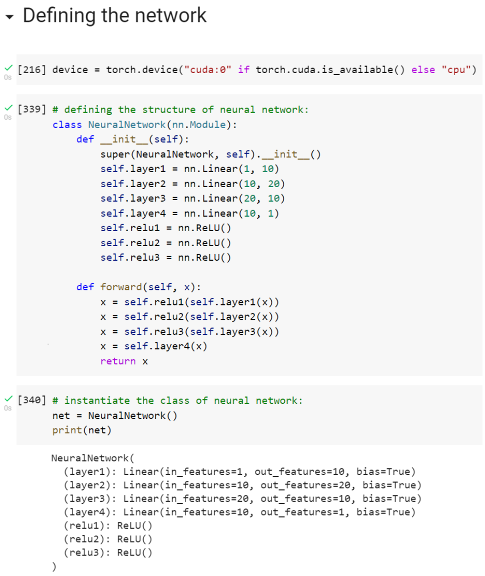
Neural network: importing packages


Importing packages

```
[188] # installation in Google Colab's Jupyter notebook:
      !pip install torch

      Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
      Requirement already satisfied: torch in /usr/local/lib/python3.8/dist-packages (1.13.1+cu116)
      Requirement already satisfied: typing-extensions in /usr/local/lib/python3.8/dist-packages (from torch) (4.4.0)
```

```
[320] # importing packages (libraries):
      import torch
      import torch.nn as nn
      from torch.utils.data import DataLoader, Dataset
      import matplotlib.pyplot as plt
      import numpy as np
      from tqdm import tqdm
```

Neural network: defining the network



Defining the network

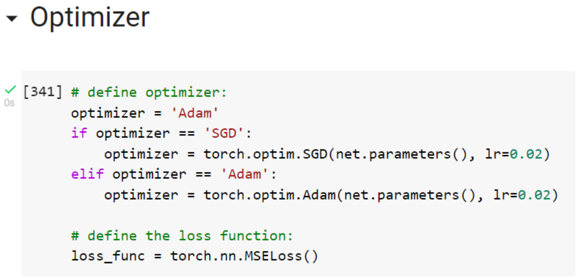
```
[216] device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

```
[339] # defining the structure of neural network:
      class NeuralNetwork(nn.Module):
          def __init__(self):
              super(NeuralNetwork, self).__init__()
              self.layer1 = nn.Linear(1, 10)
              self.layer2 = nn.Linear(10, 20)
              self.layer3 = nn.Linear(20, 10)
              self.layer4 = nn.Linear(10, 1)
              self.relu1 = nn.ReLU()
              self.relu2 = nn.ReLU()
              self.relu3 = nn.ReLU()

          def forward(self, x):
              x = self.relu1(self.layer1(x))
              x = self.relu2(self.layer2(x))
              x = self.relu3(self.layer3(x))
              x = self.layer4(x)
              return x
```

```
[340] # instantiate the class of neural network:
      net = NeuralNetwork()
      print(net)
```

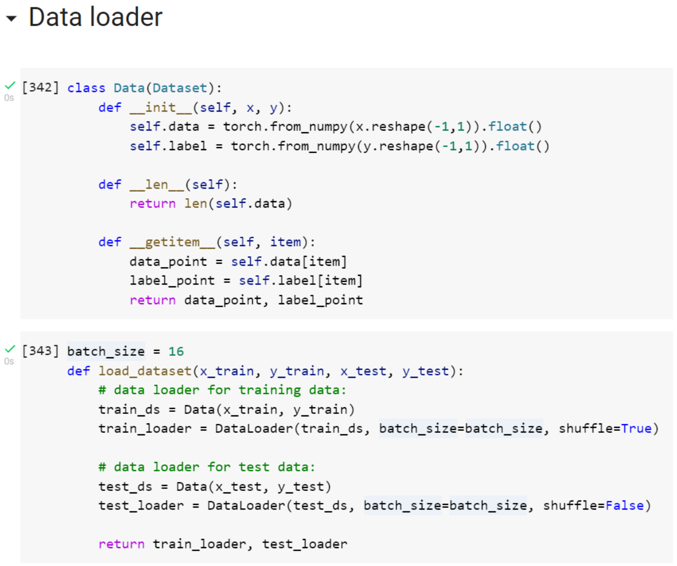
```
NeuralNetwork(
  (layer1): Linear(in_features=1, out_features=10, bias=True)
  (layer2): Linear(in_features=10, out_features=20, bias=True)
  (layer3): Linear(in_features=20, out_features=10, bias=True)
  (layer4): Linear(in_features=10, out_features=1, bias=True)
  (relu1): ReLU()
  (relu2): ReLU()
  (relu3): ReLU()
)
```

Neural network: optimizer



```
# define optimizer:
optimizer = 'Adam'
if optimizer == 'SGD':
    optimizer = torch.optim.SGD(net.parameters(), lr=0.02)
elif optimizer == 'Adam':
    optimizer = torch.optim.Adam(net.parameters(), lr=0.02)

# define the loss function:
loss_func = torch.nn.MSELoss()
```

Neural network: data loader

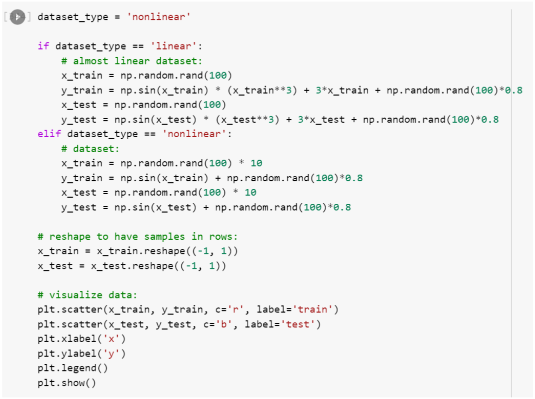


▾ Data loader

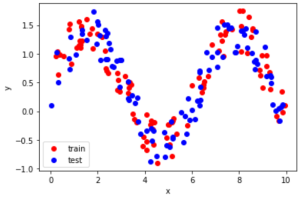
```
✓ [342] class Data(Dataset):
0s          def __init__(self, x, y):
                self.data = torch.from_numpy(x.reshape(-1,1)).float()
                self.label = torch.from_numpy(y.reshape(-1,1)).float()

            def __len__(self):
                return len(self.data)

            def __getitem__(self, item):
                data_point = self.data[item]
                label_point = self.label[item]
                return data_point, label_point
```

```
✓ [343] batch_size = 16
0s      def load_dataset(x_train, y_train, x_test, y_test):
            # data loader for training data:
            train_ds = Data(x_train, y_train)
            train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=True)

            # data loader for test data:
            test_ds = Data(x_test, y_test)
            test_loader = DataLoader(test_ds, batch_size=batch_size, shuffle=False)

            return train_loader, test_loader
```

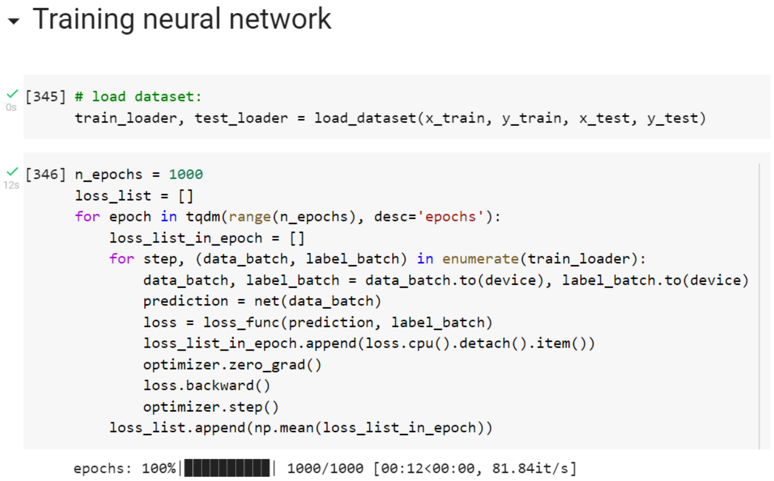
Neural network: dataset



```
dataset_type = 'nonlinear'

if dataset_type == 'linear':
    # almost linear dataset:
    x_train = np.random.rand(100)
    y_train = np.sin(x_train) * (x_train**3) + 3*x_train + np.random.rand(100)*0.8
    x_test = np.random.rand(100)
    y_test = np.sin(x_test) * (x_test**3) + 3*x_test + np.random.rand(100)*0.8
elif dataset_type == 'nonlinear':
    # dataset:
    x_train = np.random.rand(100) * 10
    y_train = np.sin(x_train) + np.random.rand(100)*0.8
    x_test = np.random.rand(100) * 10
    y_test = np.sin(x_test) + np.random.rand(100)*0.8

# reshape to have samples in rows:
x_train = x_train.reshape((-1, 1))
x_test = x_test.reshape((-1, 1))

# visualize data:
plt.scatter(x_train, y_train, c='r', label='train')
plt.scatter(x_test, y_test, c='b', label='test')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```



Neural network: training



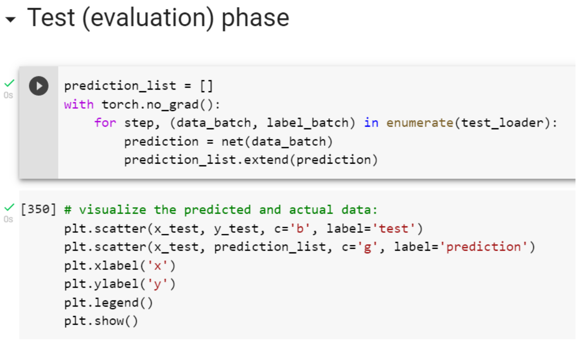
▾ Training neural network

```
✓  [345] # load dataset:
0s       train_loader, test_loader = load_dataset(x_train, y_train, x_test, y_test)
```

```
✓  [346] n_epochs = 1000
12s      loss_list = []
         for epoch in tqdm(range(n_epochs), desc='epochs'):
             loss_list_in_epoch = []
             for step, (data_batch, label_batch) in enumerate(train_loader):
                 data_batch, label_batch = data_batch.to(device), label_batch.to(device)
                 prediction = net(data_batch)
                 loss = loss_func(prediction, label_batch)
                 loss_list_in_epoch.append(loss.cpu().detach().item())
                 optimizer.zero_grad()
                 loss.backward()
                 optimizer.step()
             loss_list.append(np.mean(loss_list_in_epoch))
```

```
epochs: 100%|██████████| 1000/1000 [00:12<00:00, 81.84it/s]
```

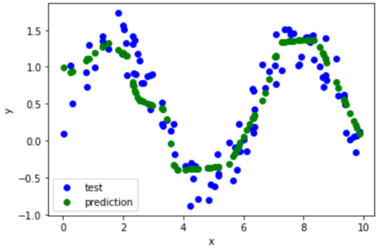
Neural network: test (evaluation)



Test (evaluation) phase

```
prediction_list = []
with torch.no_grad():
    for step, (data_batch, label_batch) in enumerate(test_loader):
        prediction = net(data_batch)
        prediction_list.extend(prediction)
```

```
[350] # visualize the predicted and actual data:
      plt.scatter(x_test, y_test, c='b', label='test')
      plt.scatter(x_test, prediction_list, c='g', label='prediction')
      plt.xlabel('x')
      plt.ylabel('y')
      plt.legend()
      plt.show()
```



Acknowledgement

- Some slides of this slide deck are inspired by the lectures of Prof. Kimon Fountoulakis at the University of Waterloo.
- Some slides of this slide deck are inspired by the lectures of Prof. Stephen Boyd at the Stanford University.
- Our tutorial also has the materials of this slide deck: [17]
- See my "Optimization Techniques" course on my YouTube channel for more information about first-order optimization including these methods.

References

[1] C. Lemaréchal, "Cauchy and the gradient method," *Doc Math Extra*, vol. 251, no. 254, p. 10, 2012.

[2] J. Hadamard, *Mémoire sur le problème d'analyse relatif à l'équilibre des plaques élastiques encastrées*, vol. 33.
Imprimerie nationale, 1908.

[3] H. B. Curry, "The method of steepest descent for non-linear minimization problems," *Quarterly of Applied Mathematics*, vol. 2, no. 3, pp. 258–261, 1944.

[4] L. Armijo, "Minimization of functions having Lipschitz continuous first partial derivatives," *Pacific Journal of mathematics*, vol. 16, no. 1, pp. 1–3, 1966.

[5] P. Wolfe, "Convergence conditions for ascent methods," *SIAM review*, vol. 11, no. 2, pp. 226–235, 1969.

[6] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[7] E. K. Chong and S. H. Zak, *An introduction to optimization*.
John Wiley & Sons, 2004.

[8] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

References (cont.)

[9] L. Bottou *et al.*, "Online learning and stochastic approximations," *On-line learning in neural networks*, vol. 17, no. 9, p. 142, 1998.

[10] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[11] B. Ghojogh, H. Nekoei, A. Ghojogh, F. Karray, and M. Crowley, "Sampling algorithms, from survey sampling to Monte Carlo methods: Tutorial and literature review," *arXiv preprint arXiv:2011.00901*, 2020.

[12] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of machine learning research*, vol. 12, no. 7, 2011.

[13] T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012.

[14] M. Riedmiller and H. Braun, "Rprop-a fast adaptive learning algorithm," in *Proceedings of the International Symposium on Computer and Information Science VII*, 1992.

[15] G. Hinton, N. Srivastava, and K. Swersky, "Neural networks for machine learning lecture 6a overview of mini-batch gradient descent," tech. rep., Department of Computer Science, University of Toronto, 2012.

[16] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

References (cont.)

[17] B. Ghojogh, A. Ghodsi, F. Karray, and M. Crowley, “KKT conditions, first-order and second-order optimization, and distributed optimization: Tutorial and survey,” *arXiv preprint arXiv:2110.01858*, 2021.