APL 405: Machine Learning for Mechanics

Lecture 12: Parameter Optimization

by

Rajdip Nayek

Assistant Professor,
Applied Mechanics Department,
IIT Delhi

Instructor email: rajdipn@am.iitd.ac.in

Recap

- We looked at different types of loss functions for regression and classification
- **Learning** the parameters of a chosen parametric model often requires minimizing an appropriate loss funciton
- An ML engineer therefore needs to be familiar with some strategies to solve optimization problems
- In this lecture, we will introduce some ideas behind some of the optimization methods used in ML

Optimization in Machine Learning (ML)

- **Optimization** → Finding the minimum or maximum of an **objective function**

- A maximization problem can be formulated as a minimization problem

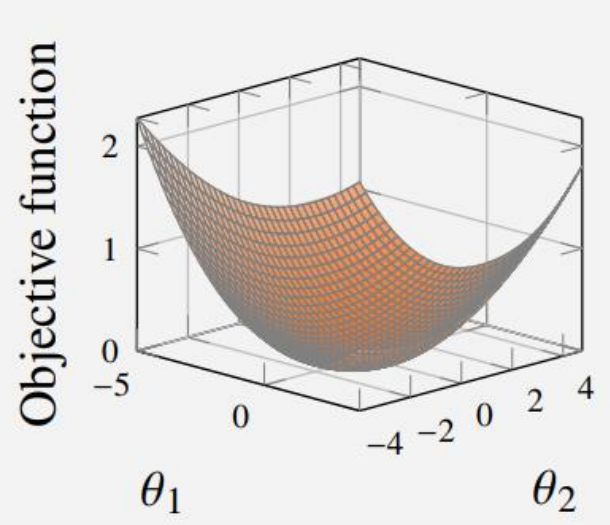
$$\widehat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} J(\boldsymbol{\theta}) = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} -J(\boldsymbol{\theta})$$

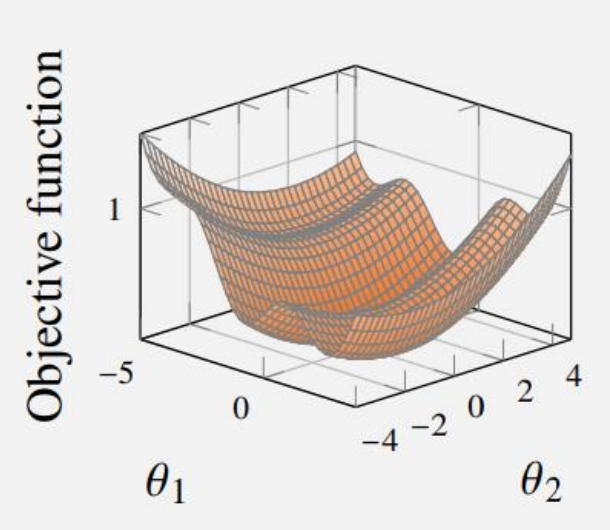
  So we will limit ourselves to minimization problems only

- Optimization in ML is primarily used in **two** ways:
  - **Training a model** by minimizing the cost function w.r.t. the model parameters
    - **Objective function** : $J(\boldsymbol{\theta})$
    - **Optimization variable**: $\boldsymbol{\theta}$
  - **Tuning hyperparameters**, such as the regularization parameter λ
    - **Objective function** : $E_{\text{hold-out}}$
    - **Optimization variable**: Hyperparameters (e.g. λ)

Convex functions

- An important class of objective functions are **convex functions**
- Optimization is *much easier* for convex objective functions, and it is a good idea to consider whether a non-convex optimization can be reformulated into a convex problem (but it is not always possible)
- Convex functions have unique minimum and no other local minima





- Examples of convex functions are cost functions for logistic regression and linear regression
- However, most problems in ML do not lead to convex functions

Gradient of a cost function

- **Training examples:** $\{(\boldsymbol{x}^{(1)}, t^{(1)}), (\boldsymbol{x}^{(2)}, t^{(2)}), \cdots, (\boldsymbol{x}^{(N)}, t^{(N)})\}$

- Let's say the chosen model be: $y = f_{\boldsymbol{\theta}}(\boldsymbol{x})$

- **Cost function** → Average over individual training losses

$$J(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} L\left(y^{(i)}, f_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})\right)$$

- **Gradient of loss function** w.r.t. parameter $\boldsymbol{\theta}$

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \frac{1}{N}\sum_{i=1}^{N} L\left(y^{(i)}, f_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})\right)$$

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} L^{(i)}$$

Note: $\boldsymbol{\theta}$ would represent hyperparameters in the case of hyperparameter optimization

Gradient Descent

- **Gradient of loss function** w.r.t. parameter $\boldsymbol{\theta}$

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} L^{(i)}$$

  - $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ has the same dimension as $\boldsymbol{\theta}$

  - $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ describes the direction in which $J(\boldsymbol{\theta})$ increases. Therefore, $-\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ describes the direction in which $J(\boldsymbol{\theta})$ decreases

- Taking a (small) step in the direction of the negative gradient will reduce the value of cost function

$$J\left(\boldsymbol{\theta} - \gamma \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})\right) \leq J(\boldsymbol{\theta}) \quad \text{for some } \gamma > 0$$

- This suggests that if we have $\boldsymbol{\theta}^{(t)}$ and want to select $\boldsymbol{\theta}^{(t+1)}$ such that $J\left(\boldsymbol{\theta}^{(t+1)}\right) \leq J\left(\boldsymbol{\theta}^{(t)}\right)$, we should

Update $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \gamma \nabla_{\boldsymbol{\theta}} J\left(\boldsymbol{\theta}^{(t)}\right)$
with some $\gamma > 0$

γ is called the learning rate

GD is a local optimizer, there is no guarantee that it will find the global minimum

Batch gradient descent (BGD)

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} L^{(i)}$$

Update $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \gamma \nabla_{\boldsymbol{\theta}} J\left(\boldsymbol{\theta}^{(t)}\right)$
with some $\gamma > 0$

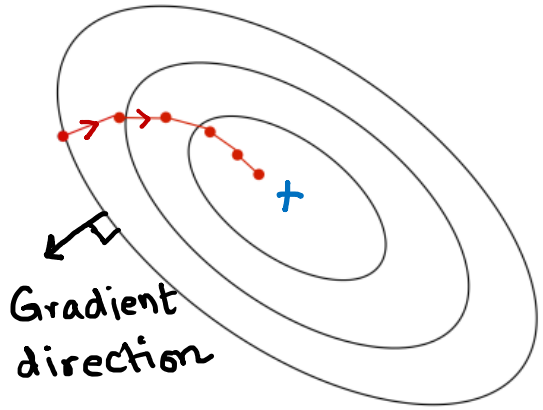
- Specify a learning rate, compute the total gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ by averaging over *all* individual loss function gradients for every training example, and then update the parameters $\boldsymbol{\theta}$

- The algorithm goes over the entire data once before updating the parameters

- This is known as **batch gradient descent (BGD)**, since we treat the entire training set as a batch

  - **Pros**: There is no approximation in gradient calculation. Each update step guarantees that the loss will decrease, if γ is small enough

  - **Cons**: However, Batch GD can be very time-consuming for a large datasets (very large N), due the summation over N datapoints



Batch gradient descent (Batch GD)

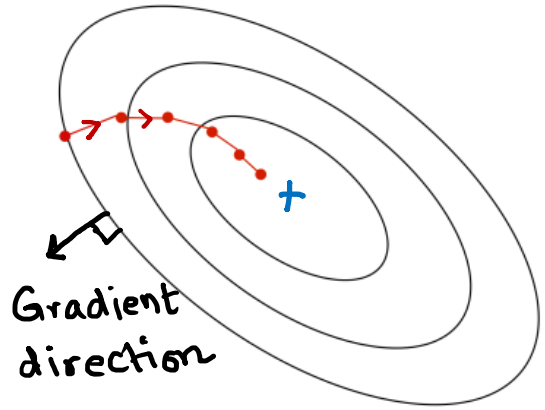
$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} L^{(i)}$$

Update $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \gamma \nabla_{\boldsymbol{\theta}} J\left(\boldsymbol{\theta}^{(t)}\right)$
with some $\gamma > 0$

- Batch gradient descent treat the entire training set as a single batch
- Updates the parameter vector after each full pass (**epoch**) over the entire dataset


```
theta   = -1        # initialize parameter vector
eta     = 0.001     # learning rate
epochs  = 100       # number of passes over entire dataset
Ntr     = 10000     # number of training points
for i in range(epochs):
  dtheta = 0        # initialize increment to zero
  for x,t in zip(X,T):
    dtheta += grad_theta(theta, x, t)

  theta = theta - eta * dtheta / Ntr
```



Stochastic gradient descent (SGD)

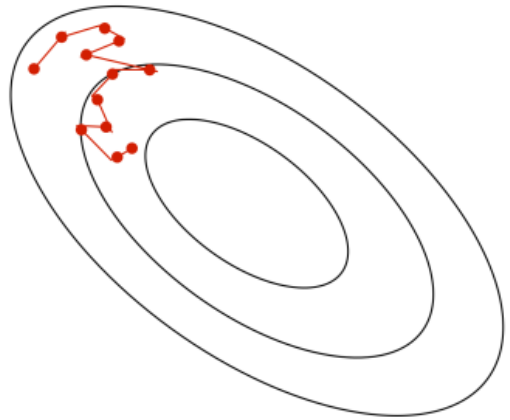
$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} L^{(i)}$$

Update $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \gamma \nabla_{\boldsymbol{\theta}} J\left(\boldsymbol{\theta}^{(t)}\right)$
with some $\gamma > 0$

- When N is very large, the summation can involve summing a many terms
- Also, it can be an issue to keep all data points in the computer memory at the same time
- Subsampling a small set from the full training set might be more useful
- In **SGD**, one random samples (without replacement) a training pair $\left(\boldsymbol{x}^{(i)}, y^{(i)}\right)$ from the full training dataset, and

Updates $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \gamma \nabla_{\boldsymbol{\theta}} L^{(i)}\left(\boldsymbol{\theta}^{(t)}\right)$
with some $\gamma > 0$

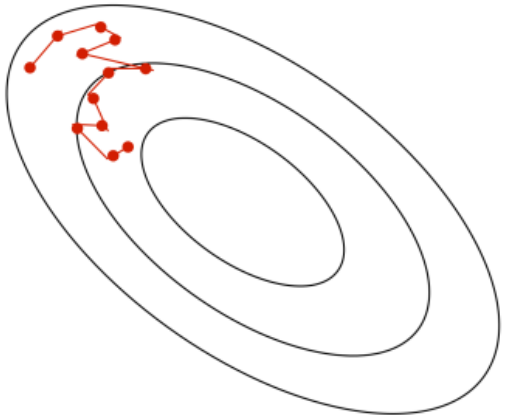
- **Pros**: SGD can make significant progress before it has even looked at the entire data!
- **Cons**: It uses an approximate estimate of gradient. So there is no guarantee that each step will decrease the loss



Stochastic gradient descent (SGD)

- We see many **fluctuations**. Why ? Because we are making greedy decisions
- Each data point is trying to push the parameters in a direction most favorable to it (without being aware of how the parameter update affects other points)
- A parameter update which is locally favorable to one point may harm other points (its almost as if the data points are competing with each other)
- There is no guarantee that each local greedy move will reduce the global error

- Can we reduce the oscillations by improving our stochastic estimates of the gradient (currently estimated from just 1 data point at a time)?
- Yes, let's look at mini-batch SGD

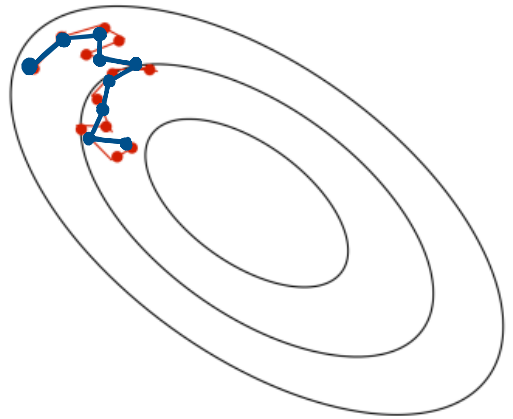


Mini-batch gradient descent

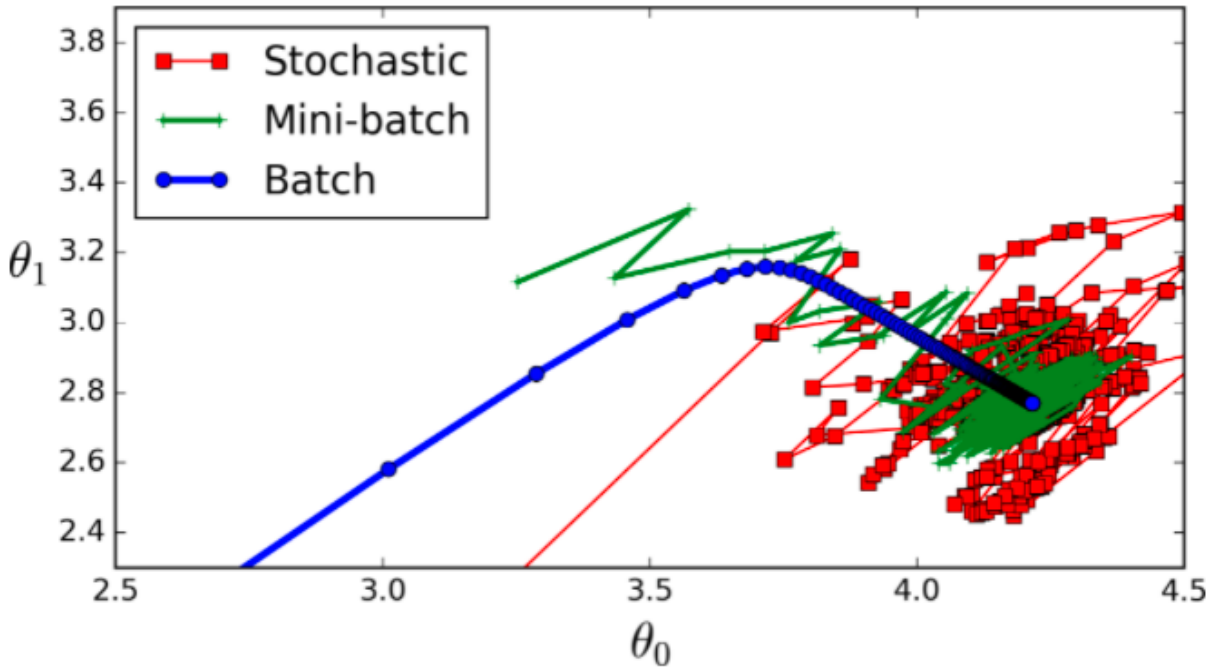
- Compute the gradients on a medium-sized set of training examples, called a ***mini-batch***
- Note that the algorithm updates the parameters after it sees a batch size B number of data points
- The stochastic estimates of gradients here are slightly better and less noisy
- Batch size $= 1$ leads to SGD! Typical batch sizes are 64, 128, 256

```
theta, eta, epochs  = -1, 0.001, 100
batch_size          = 64
num_points_seen     = 0
for i in range(epochs):
  dtheta = 0
  for x,t in zip(X,T):
    dtheta += grad_theta(theta, x, t)
    num_points_seen += 1

    if num_points_seen % batch_size == 0:
      # seen one mini-batch
      theta = theta - eta * dtheta / batch_size
      dtheta = 0 # reset gradients
```



Performance of mini-batch gradient descent



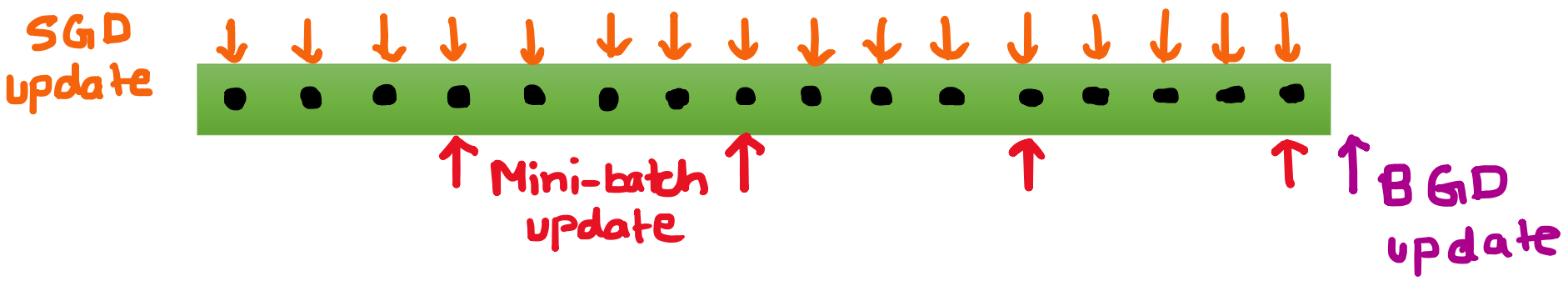
The mini-batch size *B* is a hyperparameter that needs to be set

- **Large batches:** converge in fewer parameter updates because each stochastic gradient is less noisy
- **Small batches:** perform more parameter updates because each one requires less computation

Things to remember

- N is the total number of training examples
- B is the mini batch size
- 1 epoch = one pass over the entire data
- 1 iteration = one update step of the parameters

Algorithm	Batch size	Number of iterations in 1 epoch
Batch GD	N	1
SGD	1	N
Mini-batch GD	B	$\sim N/B$

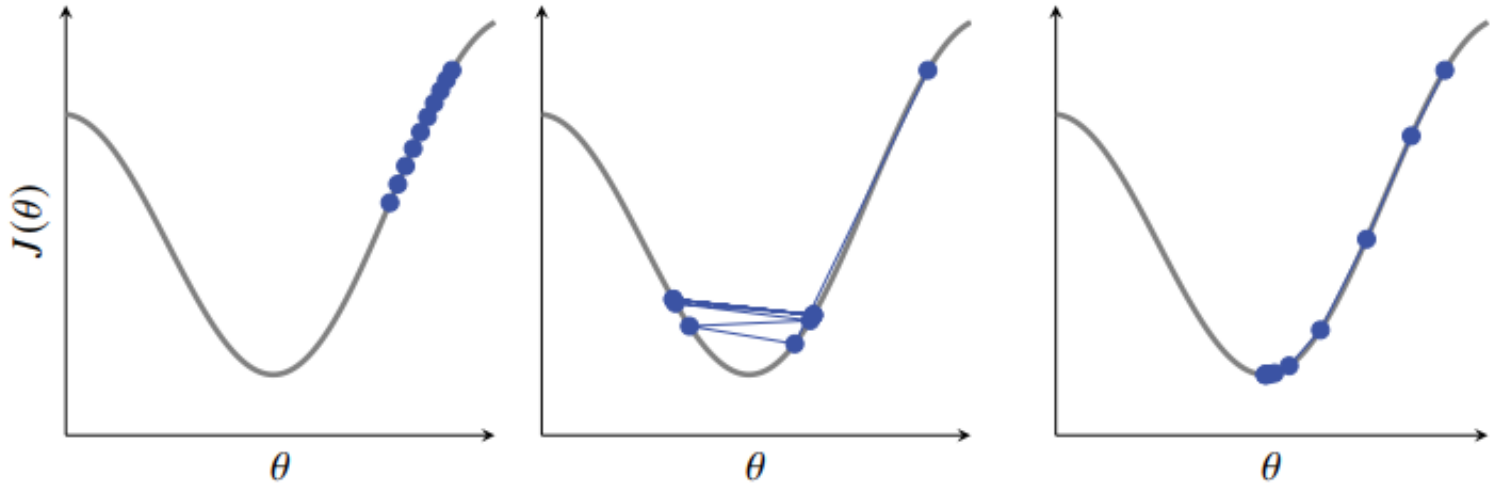


Learning rate

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} L^{(i)}$$

Update $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \gamma \nabla_{\boldsymbol{\theta}} J\left(\boldsymbol{\theta}^{(t)}\right)$
with some $\gamma > 0$

- Learning rate γ determines how big the $\boldsymbol{\theta}$-step to take at each iteration
- In practice we do not know what learning rate γ to choose



(a) Low learning rate $\gamma = 0.05$ **(b)** High learning rate $\gamma = 1.2$ **(c)** Good learning rate $\gamma = 0.3$

- γ is usually selected by the user, or it could be viewed as a hyperparameter