Deep Learning

04 Gradient Descent

Dr. Konda Reddy Mopuri
Dept. of Artificial Intelligence
IIT Hyderabad
Jan-May 2023

Loss/Risk

1. Learning: finding a good function f^* from a set of functions $\mathcal{F}$

Loss/Risk

1. Learning: finding a good function f^* from a set of functions $\mathcal{F}$
2. How to find the goodness of a function f?

Loss/Risk

1. Learning: finding a good function f^* from a set of functions $\mathcal{F}$
2. How to find the goodness of a function f?
3. Through a loss $l : \mathcal{F} \times \mathcal{L} \rightarrow \mathcal{R}$

Loss/Risk

1. Learning: finding a good function f^* from a set of functions $\mathcal{F}$
2. How to find the goodness of a function f?
3. Through a loss $l : \mathcal{F} \times \mathcal{L} \rightarrow \mathcal{R}$
4. Such that value of $l(f, z)$ increases with the *wrongness* of f on z: (measure of discripency between the expected and predicted)

Loss/Risk

1. Learning: finding a good function f^* from a set of functions $\mathcal{F}$
2. How to find the goodness of a function f?
3. Through a loss $l : \mathcal{F} \times \mathcal{L} \rightarrow \mathcal{R}$
4. Such that value of $l(f, z)$ increases with the *wrongness* of f on z: (measure of discripency between the expected and predicted)
5.
    - Regression: $l(f, (x, y)) = (f(x) - y)^2$
    - Classification: $l(f, (x, y)) = \mathbf{1}(f(x) \neq y)$
    - Density estimation: $l(q, z) = -log(q(z))$

Loss/Risk

1. Learning: finding a good function f^* from a set of functions $\mathcal{F}$
2. How to find the goodness of a function f?
3. Through a loss $l : \mathcal{F} \times \mathcal{L} \rightarrow \mathcal{R}$
4. Such that value of $l(f, z)$ increases with the *wrongness* of f on z: (measure of discripency between the expected and predicted)
5.
   - Regression: $l(f, (x, y)) = (f(x) - y)^2$
   - Classification: $l(f, (x, y)) = \mathbf{1}(f(x) \neq y)$
   - Density estimation: $l(q, z) = -log(q(z))$
6. Loss may have additional terms (from prior knowledge)

Expected Risk

1. We want f with small *expected (average) risk* $R(f) = \mathbb{E}_z(l(f, z))$

Expected Risk

1. We want f with small *expected (average) risk* $R(f) = \mathbb{E}_z(l(f, z))$
2. $f^* = \operatorname*{argmin}_{f \in \mathcal{F}} R(f)$

Expected Risk

1. We want f with small *expected (average) risk* $R(f) = \mathbb{E}_z(l(f,z))$
2. $f^* = \underset{f \in \mathcal{F}}{\operatorname{argmin}} R(f)$
3. This is unknown. However, if the training data $\mathcal{D} = \{z_1, \ldots, z_N\}$ is i.i.d. we can estimate the risk empirically (known as empirical risk),

   $\hat{R}(f;\mathcal{D}) = \hat{\mathbb{E}}_{\mathcal{D}}(l(f,z)) = \frac{1}{N}\sum_{i=1}^{N} l(f, z_n)$

Optimization

- How to find the model parameters that minimize the loss function?

$$w^* = \underset{w}{\operatorname{argmin}}\, L(w)$$

Optimization

- How to find the model parameters that minimize the loss function?

$$w^* = \underset{w}{\operatorname{argmin}}\, L(w)$$

- General and vast, but we will discuss within our context

Optimization

- Finding the parameters that minimize the training loss

$$W^*, \mathbf{b}^* = \underset{W, \mathbf{b}}{\operatorname{argmin}} \mathcal{L}(f(\cdot; W, \mathbf{b}); \mathcal{D})$$

Optimization

- Finding the parameters that minimize the training loss

$$W^*, \mathbf{b}^* = \underset{W, \mathbf{b}}{\operatorname{argmin}} \mathcal{L}(f(\cdot; W, \mathbf{b}); \mathcal{D})$$

- How do we find these optimal parameters?

Optimization

- Finding the parameters that minimize the training loss

$$W^*, \mathbf{b}^* = \operatorname*{argmin}_{W,\mathbf{b}} \mathcal{L}(f(\cdot; W, \mathbf{b}); \mathcal{D})$$

- How do we find these optimal parameters?
  - Closed form solution (e.g. linear regression)

Optimization

- Finding the parameters that minimize the training loss

$$W^*, \mathbf{b}^* = \underset{W,\mathbf{b}}{\operatorname{argmin}} \, \mathcal{L}(f(\cdot; W, \mathbf{b}); \mathcal{D})$$

- How do we find these optimal parameters?
  - Closed form solution (e.g. linear regression)
  - Ad-hoc recipes (e.g. Perceptron, K-NN classifier)

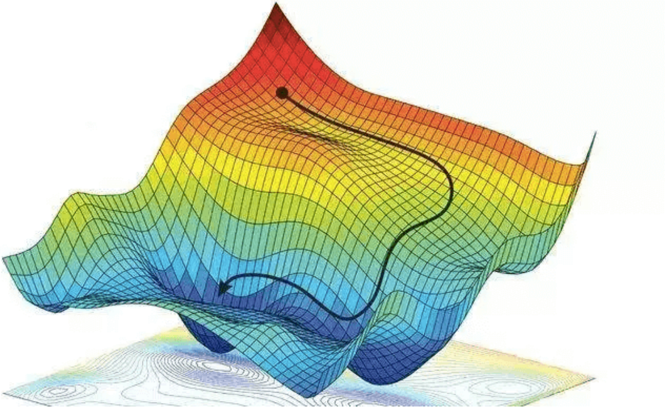
Optimization

- Finding the parameters that minimize the training loss

$$W^*, \mathbf{b}^* = \underset{W,\mathbf{b}}{\operatorname{argmin}} \mathcal{L}(f(\cdot; W, \mathbf{b}); \mathcal{D})$$

- How do we find these optimal parameters?
    - Closed form solution (e.g. linear regression)
    - Ad-hoc recipes (e.g. Perceptron, K-NN classifier)
    - What if the loss function can't be minimized analytically?

Loss surface



Source: Medium

Not-so-intelligent idea!

- Probe random directions

Not-so-intelligent idea!

- `Probe random directions`
- `Progress if you find a useful direction`

Not-so-intelligent idea!

- Probe random directions
- Progress if you find a useful direction
- Repeat

Not-so-intelligent idea!

- Probe random directions
- Progress if you find a useful direction
- Repeat
- Very ineffective!

A better looking one: Follow the slope!

- Sense the slope around the feet

A better looking one: Follow the slope!

- Sense the slope around the feet
- Identify the steepest direction, make a brief progress

A better looking one: Follow the slope!

- Sense the slope around the feet
- Identify the steepest direction, make a brief progress
- Repeat until convergence

A better looking one: Follow the slope!

- Sense the slope around the feet
- Identify the steepest direction, make a brief progress
- Repeat until convergence
- This is Gradient Descent!

Derivative and Gradient

- In 1D, derivative of a function gives the slope

$$\frac{\partial f}{\partial x} = \lim_{\delta \to 0} \frac{f(x+\delta) - f(x)}{h}$$

Derivative and Gradient

- In 1D, derivative of a function gives the slope

$$\frac{\partial f}{\partial x} = \lim_{\delta \to 0} \frac{f(x+\delta) - f(x)}{h}$$

- In higher dimensions, given a function

$$f : \mathcal{R}^D \to \mathcal{R}$$

gradient is the mapping

$$\begin{aligned} \nabla f : & \mathcal{R}^D \to \mathcal{R}^D \\ & x \to \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_D} \right) \end{aligned}$$

Derivative and Gradient

- In 1D, derivative of a function gives the slope

$$\frac{\partial f}{\partial x} = \lim_{\delta \to 0} \frac{f(x+\delta) - f(x)}{h}$$

- In higher dimensions, given a function

$$f : \mathcal{R}^D \to \mathcal{R}$$

gradient is the mapping

$$\begin{aligned} \nabla f : &\mathcal{R}^D \to \mathcal{R}^D \\ &x \to \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_D} \right) \end{aligned}$$

- ∇f vector gives the direction and rate of fastest increase for f.

Gradient Descent

$$
\begin{aligned}
\mathcal{L}(w + \eta u) &= \mathcal{L}(w) + \eta u^T \nabla_w \mathcal{L}(w) + \frac{\eta^2}{2!} u^T \nabla^2 \mathcal{L}(w) u + \ldots \\
&\approx \mathcal{L}(w) + \eta u^T \nabla_w \mathcal{L}(w)
\end{aligned}
$$

- For $\mathcal{L}(w + \eta u)$ to be lesser than $\mathcal{L}(w)$, $u^T \nabla_w \mathcal{L}(w) < 0$

Gradient Descent

$$\begin{aligned}\mathcal{L}(w + \eta u) &= \mathcal{L}(w) + \eta u^T \nabla_w \mathcal{L}(w) + \frac{\eta^2}{2!} u^T \nabla^2 \mathcal{L}(w) u + \dots \\ &\approx \mathcal{L}(w) + \eta u^T \nabla_w \mathcal{L}(w)\end{aligned}$$

- For $\mathcal{L}(w + \eta u)$ to be lesser than $\mathcal{L}(w)$, $u^T \nabla_w \mathcal{L}(w) < 0$
- The difference would be least if u is in the opposite direction to $\nabla_w \mathcal{L}(w)$, the gradient

Gradient Descent

- Goal is to minimize the error (or loss): determine the parameters w that minimize the loss $\mathcal{L}(w)$

Gradient Descent

- Goal is to minimize the error (or loss): determine the parameters w that minimize the loss $\mathcal{L}(w)$
- Gradient points uphill $\rightarrow$ negative of gradient points downhill

Gradient Descent

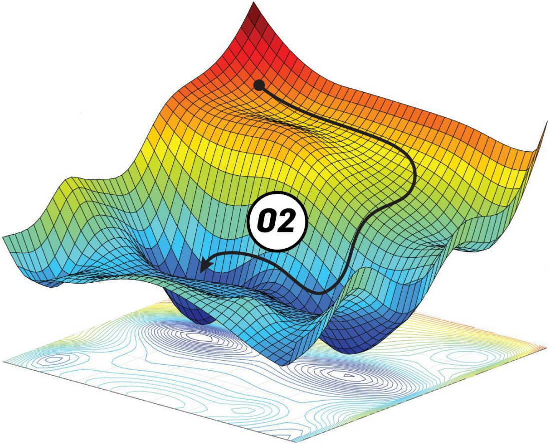


Figure credits:Ahmed Fawzy Gad

Gradient Descent

1. Start with an arbitrary initial parameter vector w_0

Gradient Descent

1. Start with an arbitrary initial parameter vector w_0
2. Repeatedly modify it via updating in small steps

Gradient Descent

1. Start with an arbitrary initial parameter vector w_0
2. Repeatedly modify it via updating in small steps
3. At each step, modify in the direction that produces steepest descent along the error surface

How to compute the gradient?

- Numerically, for each component of w using the derivative formula

$$\frac{\partial f}{\partial x} = \lim_{\delta \to 0} \frac{f(x+\delta) - f(x)}{\delta}$$

How to compute the gradient?

- Numerically, for each component of w using the derivative formula

$$\frac{\partial f}{\partial x} = \lim_{\delta \to 0} \frac{f(x+\delta) - f(x)}{\delta}$$

- Slow and approximate!

How to compute the gradient?

- Analytically, using calculus for computing the derivatives

$$L_i = \sum_{j \neq y_i} max\{0, s_j - s_{y_i} + 1\}$$

$$L = \frac{1}{N} \sum_i L_i + \sum_k w_k^2$$

$$s = f(x, W)$$

$$\nabla L_{iw}?$$

How to compute the gradient?

- Analytically, using calculus for computing the derivatives

$$L_i = \sum_{j \neq y_i} max\{0, s_j - s_{y_i} + 1\}$$

$$L = \frac{1}{N} \sum_i L_i + \sum_k w_k^2$$

$$s = f(x, W)$$

$$\nabla L_{iw}?$$

- Analytic way is fast, exact, but error-prone!

Batch Gradient Descent

for i in range(nb_epochs):
  ∇L_w = evaluate_gradient(L, $\mathcal{D}$, w)
  w = w - η * ∇L_w

Batch Gradient Descent

```
for i in range(nb_epochs):
  ∇L_w = evaluate_gradient(L, 𝒟, w)
  w = w - η * ∇L_w
```

$$\text{for } i \text{ in range(nb\_epochs):}$$
$$\nabla L_w = \text{evaluate\_gradient}(L, \mathcal{D}, w)$$
$$w = w - \eta * \nabla L_w$$

1. Guaranteed to converge to global minima in case of convex functions, and to a local minima in case of non-convex functions

Stochastic Gradient Descent (SGD)

1. Performs updates parameters for each training example
   $w = w - \eta \nabla_w \mathcal{L}(w, x^i, y^i)$

Stochastic Gradient Descent (SGD)

1. Performs updates parameters for each training example
   $w = w - \eta \nabla_w \mathcal{L}(w, x^i, y^i)$
2. In case of large datasets, Batch GD computes redundant gradients for similar examples for each parameter update

Stochastic Gradient Descent (SGD)

1. Performs updates parameters for each training example
   $w = w - \eta \nabla_w \mathcal{L}(w, x^i, y^i)$
2. In case of large datasets, Batch GD computes redundant gradients for similar examples for each parameter update
3. SGD does away with redundancy and generally faster and can be used to learn online

Stochastic Gradient Descent (SGD)

1. However, frequent updates with a high variance cause the objective function to fluctuate heavily

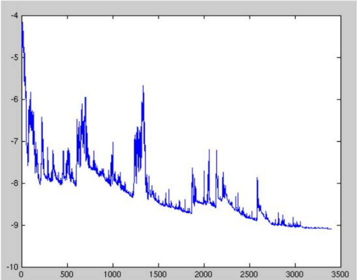


Figure credits: Wikipedia

Stochastic Gradient Descent (SGD)

1. SGD's fluctuations enable it to jump to new and potentially better local minima

Stochastic Gradient Descent (SGD)

1. SGD's fluctuations enable it to jump to new and potentially better local minima
2. This complicates the convergence, as it overshoots

Stochastic Gradient Descent (SGD)

1. SGD's fluctuations enable it to jump to new and potentially better local minima
2. This complicates the convergence, as it overshoots
3. However, if the learning rate is slowly decreased, we can show similar convergence to Batch GD

Stochastic Gradient Descent (SGD)

for i in range(nb_epochs):
  np.random.shuffle($\mathcal{D}$)
  for $x_i \in \mathcal{D}$:
    ∇L_w = evaluate_gradient(L, x_i, w)
    $w = w - \eta * \nabla L_w$

Mini-batch Gradient Descent

1. Takes the best of both worlds, updates the parameters for every mini-batch of n samples
$w = w - \eta \nabla_w \mathcal{L}(w, x^{i:i+n}, y^{i:i+n})$

Mini-batch Gradient Descent

1. Takes the best of both worlds, updates the parameters for every mini-batch of n samples
   $w = w - \eta \nabla_w \mathcal{L}(w, x^{i:i+n}, y^{i:i+n})$
2.
   - Reduces the variance of the parameter updates, which can lead to more stable convergence
   - Can make use of highly optimized matrix optimizations

Mini-batch Gradient Descent

1. Takes the best of both worlds, updates the parameters for every mini-batch of n samples
   $w = w - \eta \nabla_w \mathcal{L}(w, x^{i:i+n}, y^{i:i+n})$
2.
   - Reduces the variance of the parameter updates, which can lead to more stable convergence
   - Can make use of highly optimized matrix optimizations
3. Common mini-batch sizes vary from 32 to 1024, depending on the application

Mini-batch Gradient Descent

1. Takes the best of both worlds, updates the parameters for every mini-batch of n samples
   $w = w - \eta \nabla_w \mathcal{L}(w, x^{i:i+n}, y^{i:i+n})$
2.
   - Reduces the variance of the parameter updates, which can lead to more stable convergence
   - Can make use of highly optimized matrix optimizations
3. Common mini-batch sizes vary from 32 to 1024, depending on the application
4. This is the algorithm of choice while training DNNs (also, incorrectly referred to as SGD in general)

Mini-batch Gradient Descent

for i in range(nb_epochs):
  np.random.shuffle($\mathcal{D}$)
  for batch in get_batches($\mathcal{D}$, batch_size = 128):
   ∇L_w = evaluate_gradient(L, batch, w)
    $w = w - \eta * \nabla L_w$

Some challenges

1. Choosing a proper learning rate

Some challenges

1. Choosing a proper learning rate
   - Learning rate schedules try to adjust it during the training

Some challenges

1. Choosing a proper learning rate
   - Learning rate schedules try to adjust it during the training
   - However, these schedules are defined in advance and hence unable to adapt to the task at hand

Some challenges

1. Choosing a proper learning rate
   - Learning rate schedules try to adjust it during the training
   - However, these schedules are defined in advance and hence unable to adapt to the task at hand
2. Same learning rate applies to all the parameters

Some challenges

1. Choosing a proper learning rate
   - Learning rate schedules try to adjust it during the training
   - However, these schedules are defined in advance and hence unable to adapt to the task at hand
2. Same learning rate applies to all the parameters
3. Avoiding numerous sub-optimal local minima