EE 628 Deep Learning Fall 2019

Lecture 5

09/26/2019

Assistant Prof. Sergul Aydore

Department of Electrical and Computer Engineering

STEVENS INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

Overview

- Last lecture we covered
    - Multilayer Perceptron
    - Overfitting/underfitting
- Today, we will cover
    - Backpropagation
    - Optimization Algorithms

Forward Propagation, Backward Propagation, and Computational Graphs

- So far, we relied on backward function of the autograd module to compute the gradients

Forward Propagation, Backward Propagation, and Computational Graphs

- So far, we relied on backward function of the autograd module to compute the gradients
- The automatic calculation of gradients profoundly simplifies the implementation of deep learning algorithms

Forward Propagation, Backward Propagation, and Computational Graphs

- So far, we relied on backward function of the autograd module to compute the gradients
- The automatic calculation of gradients profoundly simplifies the implementation of deep learning algorithms
- Now, we will discuss some of the details of backward propagation

Forward Propagation, Backward Propagation, and Computational Graphs

- So far, we relied on backward function of the autograd module to compute the gradients
- The automatic calculation of gradients profoundly simplifies the implementation of deep learning algorithms
- Now, we will discuss some of the details of backward propagation
- To start, we will focus our exposition on a simple multilayer perceptron with
  - a single hidden layer and
  - $\ell 2$ norm regularization.

Really simple example

- We want
  - $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$
  - for $\mathrm{f(x, y, z) = (x + y)z}$
  - where $x = -2, y = 5, z = -4$
- Draw the computation graph

Reference: http://cs231n.stanford.edu/slides/2019/cs231n_2019_lecture04.pdf

Forward Propagation

- calculation and storage of intermediate variables from input layer to output layer.

Forward Propagation

- calculation and storage of intermediate variables from input layer to output layer.
- Let $\mathbf{x} \in R^d$ be the input example

Forward Propagation

- calculation and storage of intermediate variables from input layer to output layer.
- Let $\mathbf{x} \in R^d$ be the input example
- Intermediate variable is $\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x}$

Forward Propagation

- calculation and storage of intermediate variables from input layer to output layer.
- Let $\mathbf{x} \in R^d$ be the input example
- Intermediate variable is $\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x}$
- After inputting the intermediate variable into the activation function ϕ, we get $\mathbf{h} = \phi(\mathbf{z})$.

Forward Propagation

- calculation and storage of intermediate variables from input layer to output layer.
- Let $\mathbf{x} \in R^d$ be the input example
- Intermediate variable is $\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x}$
- After inputting the intermediate variable into the activation function ϕ, we get $\mathbf{h} = \phi(\mathbf{z})$.
- Let $\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}$ be the output variable

Forward Propagation

- calculation and storage of intermediate variables from input layer to output layer.
- Let $\mathbf{x} \in R^d$ be the input example
- Intermediate variable is $\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x}$
- After inputting the intermediate variable into the activation function ϕ, we get $\mathbf{h} = \phi(\mathbf{z})$.
- Let $\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}$ be the output variable
- Loss term for a single data example is $L = l(y, \mathbf{o})$

Forward Propagation

- calculation and storage of intermediate variables from input layer to output layer.
- Let $\mathbf{x} \in R^d$ be the input example
- Intermediate variable is $\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x}$
- After inputting the intermediate variable into the activation function ϕ, we get $\mathbf{h} = \phi(\mathbf{z})$.
- Let $\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}$ be the output variable
- Loss term for a single data example is $L = l(y, \mathbf{o})$
- The regularization term is $s = \frac{\lambda}{2}(||\mathbf{W}^{(1)}||_F^2 + ||\mathbf{W}^{(2)}||_F^2)$

Forward Propagation

- calculation and storage of intermediate variables from input layer to output layer.
- Let $\mathbf{x} \in R^d$ be the input example
- Intermediate variable is $\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x}$
- After inputting the intermediate variable into the activation function ϕ, we get $\mathbf{h} = \phi(\mathbf{z})$.
- Let $\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}$ be the output variable
- Loss term for a single data example is $L = l(y, \mathbf{o})$
- The regularization term is $s = \frac{\lambda}{2}(||\mathbf{W}^{(1)}||_F^2 + ||\mathbf{W}^{(2)}||_F^2)$
- Finally, the model's regularized loss (*objective function*) on a given data example is $J = L + s$

Computational Graph of Forward Propagation

- Plotting computational graphs helps us visualize the dependencies of operators and variables within the calculation

Computational Graph of Forward Propagation

- Plotting computational graphs helps us visualize the dependencies of operators and variables within the calculation

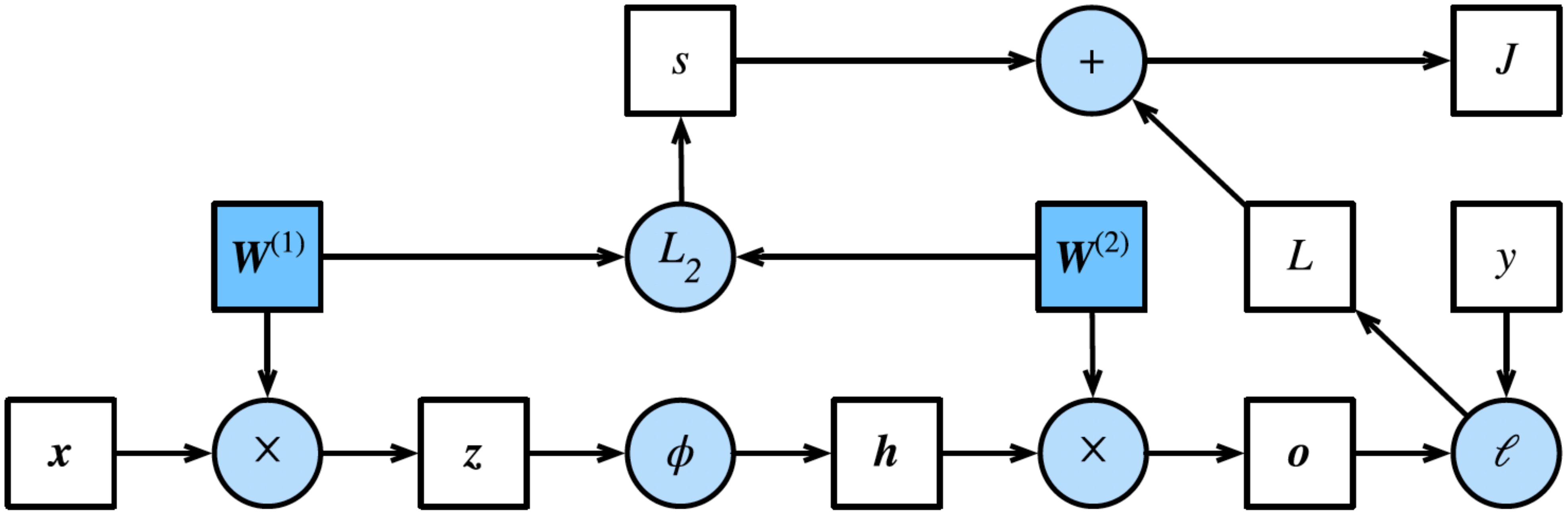


Fig. 6.7.1: Computational Graph

Backpropagation

- the method of calculating the gradient of neural network parameters

Backpropagation

- the method of calculating the gradient of neural network parameters
- in the order of the output layer to the input layer according to the **chain rule** in calculus.

Backpropagation

- the method of calculating the gradient of neural network parameters
- in the order of the output layer to the input layer according to the **chain rule** in calculus.
- What gradients do we need in our example?

Backpropagation

- $J = L + s$, then we have $\frac{\partial J}{\partial L} = ?$, $\frac{\partial J}{\partial s} = ?$

Backpropagation

- $J = L + s$, then we have $\frac{\partial J}{\partial L} =?$, $\frac{\partial J}{\partial s} =?$
- Next, we compute the gradient of the objective function with respect to the output layer $\frac{\partial J}{\partial \mathbf{o}} = prod\left(\frac{\partial J}{\partial L}, \frac{\partial L}{\partial \mathbf{o}}\right) = \frac{\partial L}{\partial \mathbf{o}}$ (what is the size?)

Backpropagation

- $J = L + s$, then we have $\frac{\partial J}{\partial L} =?$, $\frac{\partial J}{\partial s} =?$
- Next, we compute the gradient of the objective function with respect to the output layer $\frac{\partial J}{\partial \mathbf{o}} = prod\ (\frac{\partial J}{\partial L}, \frac{\partial L}{\partial \mathbf{o}}) = \frac{\partial L}{\partial \mathbf{o}}$ (what is the size?)
- Next, we calculate the gradients of the regularization term $\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)}$ and $\frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)}$.

Backpropagation

- $J = L + s$, then we have $\frac{\partial J}{\partial L} = ?$, $\frac{\partial J}{\partial s} = ?$
- Next, we compute the gradient of the objective function with respect to the output layer $\frac{\partial J}{\partial \mathbf{o}} = prod\ (\frac{\partial J}{\partial L}, \frac{\partial L}{\partial \mathbf{o}}) = \frac{\partial L}{\partial \mathbf{o}}$ (what is the size?)
- Next, we calculate the gradients of the regularization term $\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)}$ and $\frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)}$.
- Now we are able to calculate the gradient $\frac{\partial J}{\partial \mathbf{W}^{(2)}} = prod\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial o}{\partial \mathbf{W}^{(2)}}\right) + prod\ \left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}}\right) = \frac{\partial J}{\partial \mathbf{o}} \mathbf{h}^T + \lambda \mathbf{W}^{(2)}$

Backpropagation

- $J = L + s$, then we have $\frac{\partial J}{\partial L} =?$, $\frac{\partial J}{\partial s} =?$
- Next, we compute the gradient of the objective function with respect to the output layer $\frac{\partial J}{\partial \mathbf{o}} = prod\ (\frac{\partial J}{\partial L}, \frac{\partial L}{\partial \mathbf{o}}) = \frac{\partial L}{\partial \mathbf{o}}$ (what is the size?)
- Next, we calculate the gradients of the regularization term $\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)}$ and $\frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)}$.
- Now we are able to calculate the gradient $\frac{\partial J}{\partial \mathbf{W}^{(2)}} = prod\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial o}{\partial \mathbf{W}^{(2)}}\right) + prod\ \left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}}\right) = \frac{\partial J}{\partial \mathbf{o}} \mathbf{h}^T + \lambda \mathbf{W}^{(2)}$
- To obtain the gradient with respect to $\mathbf{W}^{(1)}$, we need to continue backpropagation along the output layer to the hidden layer $\frac{\partial J}{\partial \mathbf{h}} = prod\ \left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{h}}\right) = W^{(2)T} \frac{\partial J}{\partial \mathbf{o}}$. Calculating gradient with respect to $\mathbf{z}$, requires derivative of the activation function $\frac{\partial J}{\partial z} = prod\ \left(\frac{\partial J}{\partial \mathbf{h}}, \frac{\partial \mathbf{h}}{\partial z}\right) = \frac{\partial J}{\partial \mathbf{h}} \odot \phi'(\mathbf{z})$.

Backpropagation

- $J = L + s$, then we have $\frac{\partial J}{\partial L} =?$, $\frac{\partial J}{\partial s} =?$
- Next, we compute the gradient of the objective function with respect to the output layer $\frac{\partial J}{\partial \mathbf{o}} = prod\ (\frac{\partial J}{\partial L}, \frac{\partial L}{\partial \mathbf{o}}) = \frac{\partial L}{\partial \mathbf{o}}$ (what is the size?)
- Next, we calculate the gradients of the regularization term $\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)}$ and $\frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)}$.
- Now we are able to calculate the gradient $\frac{\partial J}{\partial \mathbf{W}^{(2)}} = prod\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial o}{\partial \mathbf{W}^{(2)}}\right) + prod\ \left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}}\right) = \frac{\partial J}{\partial \mathbf{o}} \mathbf{h}^T + \lambda \mathbf{W}^{(2)}$
- To obtain the gradient with respect to $\mathbf{W}^{(1)}$, we need to continue backpropagation along the output layer to the hidden layer $\frac{\partial J}{\partial \mathbf{h}} = prod\ \left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{h}}\right) = W^{(2)T} \frac{\partial J}{\partial \mathbf{o}}$. Calculating gradient with respect to $\mathbf{z}$, requires derivative of the activation function $\frac{\partial J}{\partial z} = prod\ \left(\frac{\partial J}{\partial \mathbf{h}}, \frac{\partial \mathbf{h}}{\partial \mathbf{z}}\right) = \frac{\partial J}{\partial \mathbf{h}} \odot \phi'(\mathbf{z})$.
- Finally we obtain $\frac{\partial J}{\partial \mathbf{W}^{(1)}} = prod\left(\frac{\partial J}{\partial \mathbf{z}}, \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}}\right) + prod\ \left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(1)}}\right) = \frac{\partial J}{\partial \mathbf{z}} \mathbf{x}^T + \lambda \mathbf{W}^{(1)}$.

Training a Model

- forward and backward propagation depend on each other.

Training a Model

- forward and backward propagation depend on each other.
- for forward propagation, we traverse the compute graph in the direction of dependencies and compute all the variables on its path

Training a Model

- forward and backward propagation depend on each other.
- for forward propagation, we traverse the compute graph in the direction of dependencies and compute all the variables on its path
- These are then used for backpropagation where the compute order on the graph is reversed.

Training a Model

- forward and backward propagation depend on each other.
- for forward propagation, we traverse the compute graph in the direction of dependencies and compute all the variables on its path
- These are then used for backpropagation where the compute order on the graph is reversed.
- we need to retain the intermediate values until backpropagation is complete

Training a Model

- forward and backward propagation depend on each other.
- for forward propagation, we traverse the compute graph in the direction of dependencies and compute all the variables on its path
- These are then used for backpropagation where the compute order on the graph is reversed.
- we need to retain the intermediate values until backpropagation is complete
- This is also one of the reasons why backpropagation requires significantly more memory

Numerical Stability and Initialization

- Which nonlinearity function we use
- How we decide to initialize our parameters
- can play important role in convergence

Vanishing and Exploding Gradients

- Consider a deep neural network with d layers, input $\mathbf{x}$ and output $\mathbf{o}$.

Vanishing and Exploding Gradients

- Consider a deep neural network with d layers, input $\mathbf{x}$ and output $\mathbf{o}$.
- Each layer satisfies $\mathbf{h}^{t+1} = f_t(\mathbf{h}^t)$ and thus $\mathbf{o} = f_d \circ \ldots \circ f_1(\mathbf{x})$

Vanishing and Exploding Gradients

- Consider a deep neural network with d layers, input $\mathbf{x}$ and output $\mathbf{o}$.
- Each layer satisfies $\mathbf{h}^{t+1} = f_t(\mathbf{h}^t)$ and thus $\mathbf{o} = f_d \circ \ldots \circ f_1(\mathbf{x})$
- We can write the gradient of $\mathbf{o}$ with respect to any set of parameters $\mathbf{W}_t$ at layer t simply as

$$\partial_{\mathbf{W}_t}\mathbf{o} = \underbrace{\partial_{\mathbf{h}^{d-1}}\mathbf{h}^d}_{:=\mathbf{M}_d} \cdot \ldots \cdot \underbrace{\partial_{\mathbf{h}^t}\mathbf{h}^{t+1}}_{:=\mathbf{M}_t} \underbrace{\partial_{\mathbf{W}_t}\mathbf{h}^t}_{:=\mathbf{v}_t}.$$

Vanishing and Exploding Gradients

- Consider a deep neural network with d layers, input $\mathbf{x}$ and output $\mathbf{o}$.
- Each layer satisfies $\mathbf{h}^{t+1} = f_t(\mathbf{h}^t)$ and thus $\mathbf{o} = f_d \circ \ldots \circ f_1(\mathbf{x})$
- We can write the gradient of $\mathbf{o}$ with respect to any set of parameters $\mathbf{W}_t$ at layer t simply as

$$\partial_{\mathbf{W}_t}\mathbf{o} = \underbrace{\partial_{\mathbf{h}^{d-1}}\mathbf{h}^d}_{:=\mathbf{M}_d} \cdot \ldots \cdot \underbrace{\partial_{\mathbf{h}^t}\mathbf{h}^{t+1}}_{:=\mathbf{M}_t} \underbrace{\partial_{\mathbf{W}_t}\mathbf{h}^t}_{:=\mathbf{v}_t} .$$

- In other words, it is the product of $d - t$ matrices and a gradient vector

Vanishing and Exploding Gradients

- Consider a deep neural network with d layers, input $\mathbf{x}$ and output $\mathbf{o}$.
- Each layer satisfies $\mathbf{h}^{t+1} = f_t(\mathbf{h}^t)$ and thus $\mathbf{o} = f_d \circ \ldots \circ f_1(\mathbf{x})$
- We can write the gradient of $\mathbf{o}$ with respect to any set of parameters $\mathbf{W}_t$ at layer t simply as

$$\partial_{\mathbf{W}_t}\mathbf{o} = \underbrace{\partial_{\mathbf{h}^{d-1}}\mathbf{h}^d}_{:=\mathbf{M}_d} \cdot \ldots \cdot \underbrace{\partial_{\mathbf{h}^t}\mathbf{h}^{t+1}}_{:=\mathbf{M}_t} \underbrace{\partial_{\mathbf{W}_t}\mathbf{h}^t}_{:=\mathbf{v}_t}.$$

- In other words, it is the product of $d - t$ matrices and a gradient vector
- This product might be too large or too small!

Vanishing Gradients

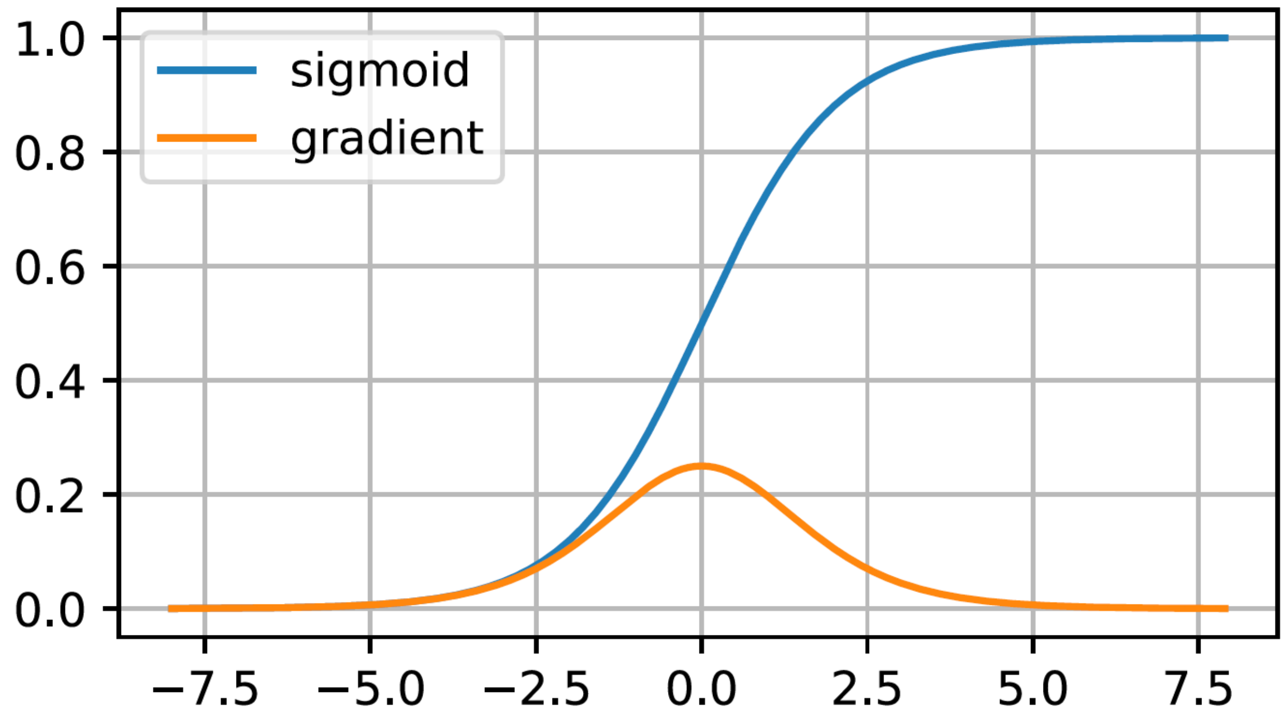
- Major problem for vanishing gradients is usually related to the activation function

Vanishing Gradients

- Major problem for vanishing gradients is usually related to the activation function
- Sigmoid function outputs 0 gradients unless we are in a perfect zone.

Vanishing Gradients

- Major problem for vanishing gradients is usually related to the activation function
- Sigmoid function outputs 0 gradients unless we are in a perfect zone.
- That is why ReLUs have become the default choice !



Exploding Gradients

- The opposite problem of vanishing gradients

Exploding Gradients

- The opposite problem of vanishing gradients
- Let’s draw 100 Gaussian random matrices and multiply them with some initial matrix 100 times.

Exploding Gradients

- The opposite problem of vanishing gradients
- Let’s draw 100 Gaussian random matrices and multiply them with some initial matrix 100 times.
- The matrix product explodes

```
A single matrix
[[ 2.2122064   0.7740038   1.0434405   1.1839255 ]
 [ 1.8917114  -1.2347414  -1.771029   -0.45138445]
 [ 0.57938355 -1.856082   -1.9768796  -0.20801921]
 [ 0.2444218  -0.03716067 -0.48774993 -0.02261727]]
<NDArray 4x4 @cpu(0)>
After multiplying 100 matrices
[[ 3.1575275e+20 -5.0052276e+19  2.0565092e+21 -2.3741922e+20]
 [-4.6332600e+20  7.3445046e+19 -3.0176513e+21  3.4838066e+20]
 [-5.8487235e+20  9.2711797e+19 -3.8092853e+21  4.3977330e+20]
 [-6.2947415e+19  9.9783660e+18 -4.0997977e+20  4.7331174e+19]]
<NDArray 4x4 @cpu(0)>
```

Symmetry

- Imagine what would happen if we initialized all of the parameters of some layer as $\mathbf{W}_l = c$ for some constant c.

Symmetry

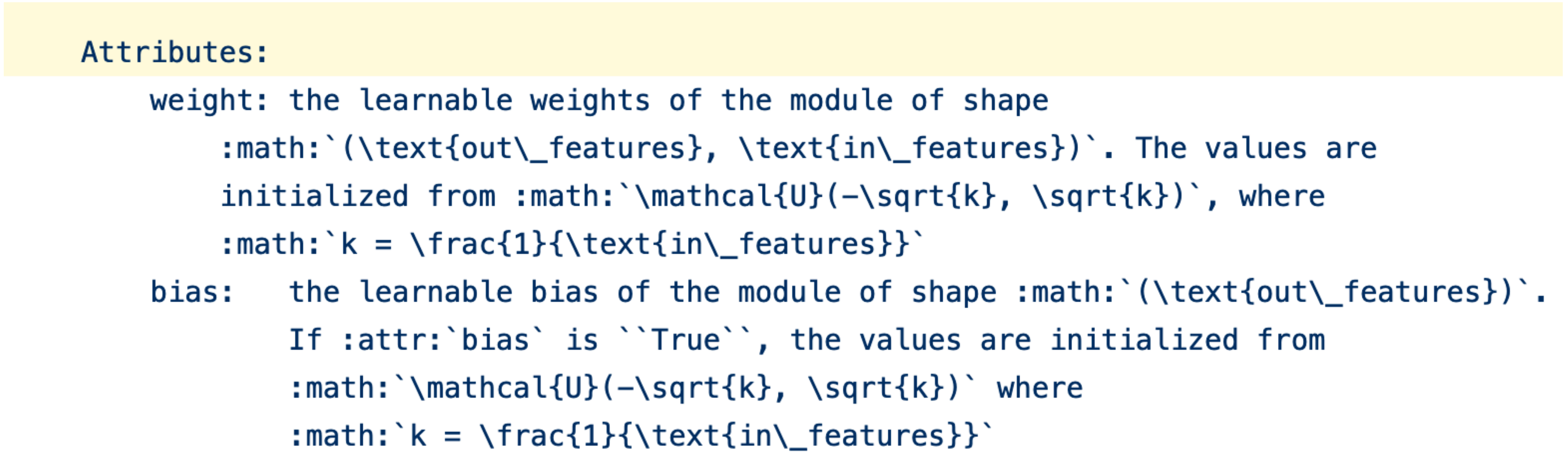
- Imagine what would happen if we initialized all of the parameters of some layer as $\mathbf{W}_l = c$ for some constant c.
- In this case, the gradients for all dimensions are identical

Symmetry

- Imagine what would happen if we initialized all of the parameters of some layer as $\mathbf{W}_l = c$ for some constant c.
- In this case, the gradients for all dimensions are identical
- SGD would never break symmetry but dropout regularization would.

Parameter Initialization

- One way of addressing the issues raised above is through careful initialization of the weight vectors
- PyTorch uses uniform initialization by default for Linear layers



```
Attributes:
    weight: the learnable weights of the module of shape
        :math:`(\text{out\_features}, \text{in\_features})`. The values are
        initialized from :math:`\mathcal{U}(-\sqrt{k}, \sqrt{k})`, where
        :math:`k = \frac{1}{\text{in\_features}}`
    bias:   the learnable bias of the module of shape :math:`(\text{out\_features})`.
            If :attr:`bias` is ``True``, the values are initialized from
            :math:`\mathcal{U}(-\sqrt{k}, \sqrt{k})` where
            :math:`k = \frac{1}{\text{in\_features}}`
```

Parameter Initialization

- One way of addressing the issues raised above is through careful initialization of the weight vectors

Xavier Initialization

- Let’s look at the scale distributions of the activations of the hidden unit h_i: $h_i = \sum_{j=1}^{n_{in}} W_{ij} x_j$

Xavier Initialization

- Let's look at the scale distributions of the activations of the hidden unit h_i: $h_i = \sum_{j=1}^{n_{in}} W_{ij} x_j$
- Assume that W_{ij} are i.i.d and have zero mean and variance σ^2

Xavier Initialization

- Let's look at the scale distributions of the activations of the hidden unit h_i: $h_i = \sum_{j=1}^{n_{in}} W_{ij} x_j$
- Assume that W_{ij} are i.i.d and have zero mean and variance σ^2
- Also assume that the data is zero mean with variance γ^2

Xavier Initialization

- Let's look at the scale distributions of the activations of the hidden unit h_i: $h_i = \sum_{j=1}^{n_{in}} W_{ij} x_j$
- Assume that W_{ij} are i.i.d and have zero mean and variance σ^2
- Also assume that the data is zero mean with variance γ^2
- In this case, the mean and variance of h_i is:

$$\mathrm{E}[h_i] = \sum_{j=1}^{n_{in}} \mathrm{E}[W_{ij} x_j] = 0 \qquad \mathrm{E}[h_i^2] = \sum_{j=1}^{n_{in}} \mathrm{E}[W_{ij}^2 x_j^2] = \sum_{j=1}^{n_{in}} \mathrm{E}[W_{ij}^2] \mathrm{E}[x_j^2] = n_{in} \sigma^2 \gamma^2$$

Xavier Initialization

- Let's look at the scale distributions of the activations of the hidden unit h_i: $h_i = \sum_{j=1}^{n_{in}} W_{ij} x_j$
- Assume that W_{ij} are i.i.d and have zero mean and variance σ^2
- Also assume that the data is zero mean with variance γ^2
- In this case, the mean and variance of h_i is:

$$\mathrm{E}[h_i] = \sum_{j=1}^{n_{in}} \mathrm{E}[W_{ij} x_j] = 0 \qquad \mathrm{E}[h_i^2] = \sum_{j=1}^{n_{in}} \mathrm{E}[W_{ij}^2 x_j^2] = \sum_{j=1}^{n_{in}} \mathrm{E}[W_{ij}^2]\mathrm{E}[x_j^2] = n_{in}\sigma^2\gamma^2$$

- To keep the variance fixed, we need $n_{in}\sigma^2 = 1$.

Xavier Initialization

- Let's look at the scale distributions of the activations of the hidden unit h_i: $h_i = \sum_{j=1}^{n_{in}} W_{ij} x_j$
- Assume that W_{ij} are i.i.d and have zero mean and variance σ^2
- Also assume that the data is zero mean with variance γ^2
- In this case, the mean and variance of h_i is:

$$\mathrm{E}[h_i] = \sum_{j=1}^{n_{in}} \mathrm{E}[W_{ij} x_j] = 0 \qquad \mathrm{E}[h_i^2] = \sum_{j=1}^{n_{in}} \mathrm{E}[W_{ij}^2 x_j^2] = \sum_{j=1}^{n_{in}} \mathrm{E}[W_{ij}^2]\mathrm{E}[x_j^2] = n_{in}\sigma^2\gamma^2$$

- To keep the variance fixed, we need $n_{in}\sigma^2 = 1$.
- Similarly, we need $n_{out}\sigma^2 = 1$ for backpropagation which leaves us in a dilemma.

Xavier Initialization

- Let's look at the scale distributions of the activations of the hidden unit h_i: $h_i = \sum_{j=1}^{n_{in}} W_{ij} x_j$
- Assume that W_{ij} are i.i.d and have zero mean and variance σ^2
- Also assume that the data is zero mean with variance γ^2
- In this case, the mean and variance of h_i is:

$$\mathrm{E}[h_i] = \sum_{j=1}^{n_{in}} \mathrm{E}[W_{ij} x_j] = 0 \qquad \mathrm{E}[h_i^2] = \sum_{j=1}^{n_{in}} \mathrm{E}[W_{ij}^2 x_j^2] = \sum_{j=1}^{n_{in}} \mathrm{E}[W_{ij}^2]\mathrm{E}[x_j^2] = n_{in}\sigma^2\gamma^2$$

- To keep the variance fixed, we need $n_{in}\sigma^2 = 1$.
- Similarly, we need $n_{out}\sigma^2 = 1$ for backpropagation which leaves us in a dilemma.
- Xavier initialization simply tries to satisfy: $\frac{1}{2}(n_{in} + n_{out})\sigma^2 = 1$

Optimization Algorithms

- Optimization Algorithms are important for deep learning

Optimization Algorithms

- Optimization Algorithms are important for deep learning
- The performance of the training algorithm directly affects the model's training efficiency

Optimization Algorithms

- Optimization Algorithms are important for deep learning
- The performance of the training algorithm directly affects the model’s training efficiency
- Understanding the principles of different optimization algorithms will help us tune the parameters in a targeted manner

Optimization Algorithms

- Optimization Algorithms are important for deep learning
- The performance of the training algorithm directly affects the model's training efficiency
- Understanding the principles of different optimization algorithms will help us tune the parameters in a targeted manner
- Now, it is time to explore common deep learning algorithms in depth

Optimization Algorithms

- Optimization Algorithms are important for deep learning
- The performance of the training algorithm directly affects the model’s training efficiency
- Understanding the principles of different optimization algorithms will help us tune the parameters in a targeted manner
- Now, it is time to explore common deep learning algorithms in depth
- Almost all problems arising in deep learning are nonconvex

Optimization Algorithms

- Optimization Algorithms are important for deep learning
- The performance of the training algorithm directly affects the model’s training efficiency
- Understanding the principles of different optimization algorithms will help us tune the parameters in a targeted manner
- Now, it is time to explore common deep learning algorithms in depth
- Almost all problems arising in deep learning are nonconvex
- However, analysis of the algorithms in the context of convex problems can be very instructive.

Optimization and Deep Learning

- We usually define a loss function first in deep learning

Optimization and Deep Learning

- We usually define a loss function first in deep learning
- Then, we use an optimization algorithm in an attempt to minimize it

Optimization and Deep Learning

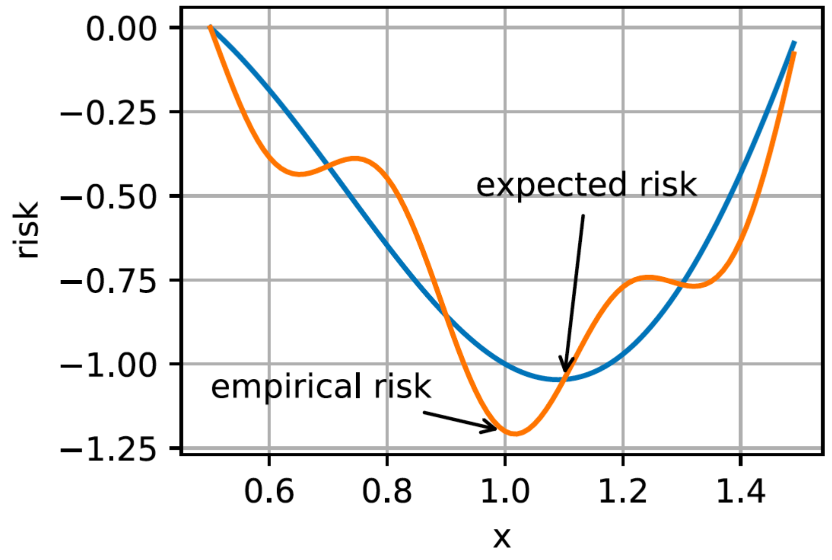
- We usually define a loss function first in deep learning
- Then, we use an optimization algorithm in an attempt to minimize it
- By tradition most optimization algorithms are concerned with minimization

Optimization and Deep Learning

- We usually define a loss function first in deep learning
- Then, we use an optimization algorithm in an attempt to minimize it
- By tradition most optimization algorithms are concerned with minimization
- The goal of optimization is to reduce training error but the goal is deep learning (statistical inference in general)

Optimization and Deep Learning

- We usually define a loss function first in deep learning
- Then, we use an optimization algorithm in an attempt to minimize it
- By tradition most optimization algorithms are concerned with minimization
- The goal of optimization is to reduce training error but the goal is deep learning (statistical inference in general)

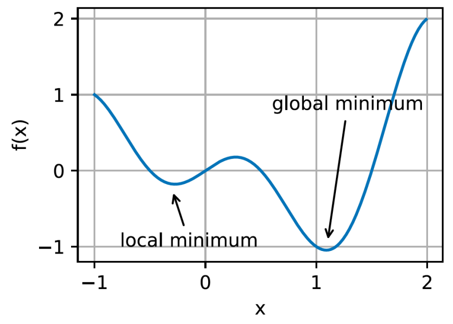


Optimization Challenges in Deep Learning

- We are going to focus specifically on the performance of the optimization algorithm in minimizing the objective function

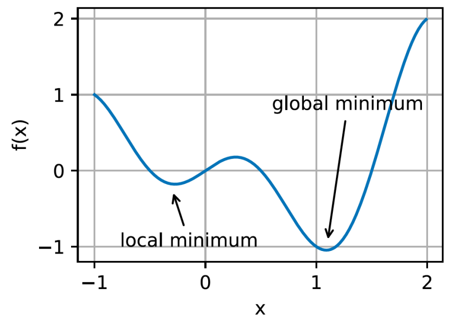
Optimization Challenges in Deep Learning

- We are going to focus specifically on the performance of the optimization algorithm in minimizing the objective function
- **Local Minima**: The objective functions of deep learning models usually has many local minima.
    - Only some degree of noise might knock the parameter out of the local minimum.
    - This is one of the beneficial properties of SGD.

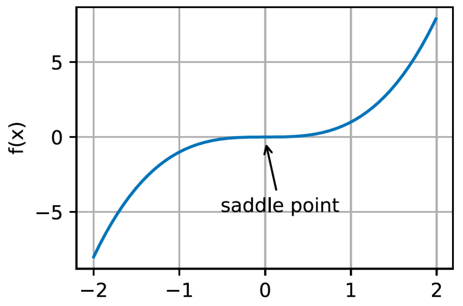


Optimization Challenges in Deep Learning

- We are going to focus specifically on the performance of the optimization algorithm in minimizing the objective function
- **Local Minima**: The objective functions of deep learning models usually has many local minima.
  - Only some degree of noise might knock the parameter out of the local minimum.
  - This is one of the beneficial properties of SGD.

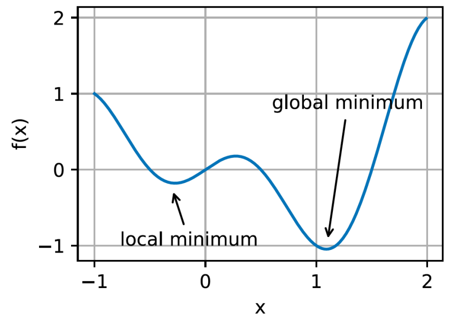


- **Saddle Points:** Any location where all gradients vanish but which is neither a global or a local minimum.
  - Optimization might stall at the point even though it is not a minimum.

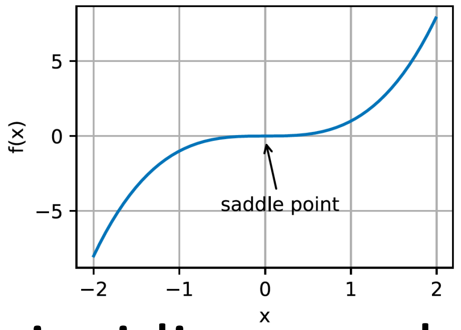


Optimization Challenges in Deep Learning

- We are going to focus specifically on the performance of the optimization algorithm in minimizing the objective function
- **Local Minima**: The objective functions of deep learning models usually has many local minima.
  - Only some degree of noise might knock the parameter out of the local minimum.
  - This is one of the beneficial properties of SGD.



- **Saddle Points:** Any location where all gradients vanish but which is neither a global or a local minimum.
  - Optimization might stall at the point even though it is not a minimum.



- Vanishing Gradients: Probably, the most insidious problem

Convexity

- It is much easier to design and test algorithms in the context of convexity.

Convexity

- It is much easier to design and test algorithms in the context of convexity.
- Optimization problems in DL still exhibit some properties of convex problems near local minima

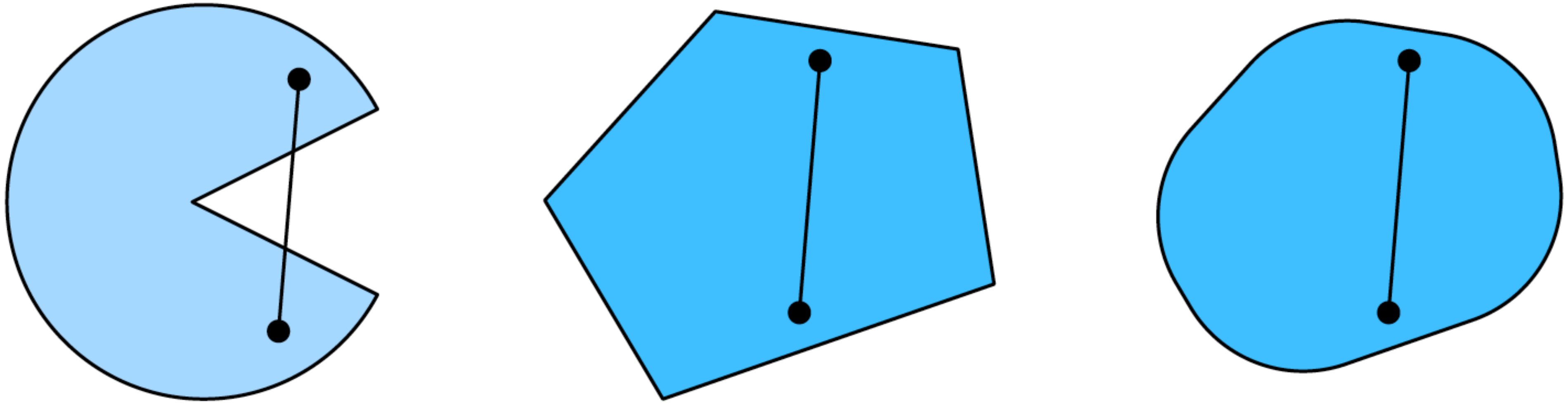
Convexity

- It is much easier to design and test algorithms in the context of convexity.
- Optimization problems in DL still exhibit some properties of convex problems near local minima
- **Sets**: A set X in a vector space is convex if for any $a, b \in X$ the line segment connecting a and b is also in X. Mathematically, for all $\lambda \in [0, 1]$ we have

$$\lambda a + (1 - \lambda) b \in X \text{ whenever } a, b \in X$$

Convexity

- It is much easier to design and test algorithms in the context of convexity.
- Optimization problems in DL still exhibit some properties of convex problems near local minima
- **Sets**: A set X in a vector space is convex if for any $a, b \in X$ the line segment connecting a and b is also in X. Mathematically, for all $\lambda \in [0, 1]$ we have



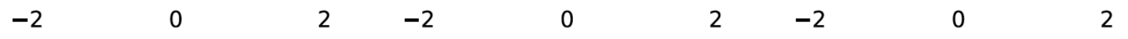
Convexity

- Say X and Y are convex sets, then $X \cap Y$ is ?

−2 0 2 −2 0 2 −2 0 2

Convexity

- Say X and Y are convex sets, then $X \cap Y$ is ?
- Say X and Y are convex sets, then $X \cup Y$ is ?



Convexity

- Say X and Y are convex sets, then $X \cap Y$ is ?
- Say X and Y are convex sets, then $X \cup Y$ is ?
- Typically, the problems in DL are defined on convex domains
  - For instance R^d is a convex set
  - In some cases we work with variables of bounded length, such as balls of radius r

$$\{\mathbf{x} | \mathbf{x} \in \mathbb{R}^d \text{ and } \|\mathbf{x}\|_2 \leq r\}$$

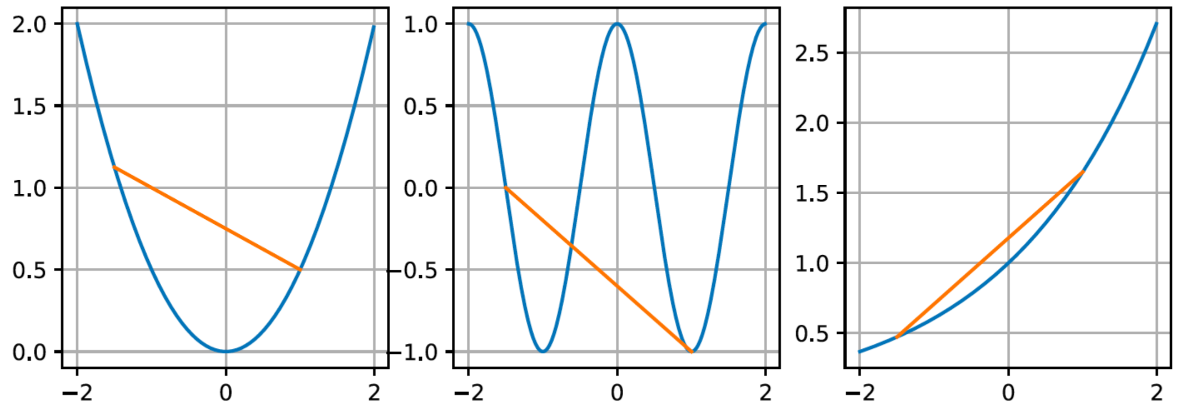
Convexity

- Say X and Y are convex sets, then $X \cap Y$ is ?
- Say X and Y are convex sets, then $X \cup Y$ is ?
- Typically, the problems in DL are defined on convex domains
  - For instance R^d is a convex set
  - In some cases we work with variables of bounded length, such as balls of radius r

$$\{\mathbf{x} | \mathbf{x} \in \mathbb{R}^d \text{ and } \|\mathbf{x}\|_2 \leq r\}$$

- **Functions**: Given a convex set X, a function defined on it $f: X \to R$ is convex if for all $x, x' \in X$ and for all $\lambda \in [0, 1]$ we have

$$\lambda f(x) + (1 - \lambda) f(x') \geq f(\lambda x + (1 - \lambda) x')$$



Convexity

- Jensen's Inequality: It amounts to generalization of the definition of convexity. $\sum_i \alpha_i f(x_i) \geq f\left(\sum_i \alpha_i x_i\right)$ and $\mathrm{E}_x[f(x)] \geq f\left(\mathrm{E}_x[x]\right)$

Convexity

- Jensen's Inequality: It amounts to generalization of the definition of convexity. $\sum_i \alpha_i f(x_i) \geq f\left(\sum_i \alpha_i x_i\right)$ and $\mathrm{E}_x[f(x)] \geq f\left(\mathrm{E}_x[x]\right)$
- In other words, the expectation of a convex function is larger than the convex function of an expectation.

Convexity

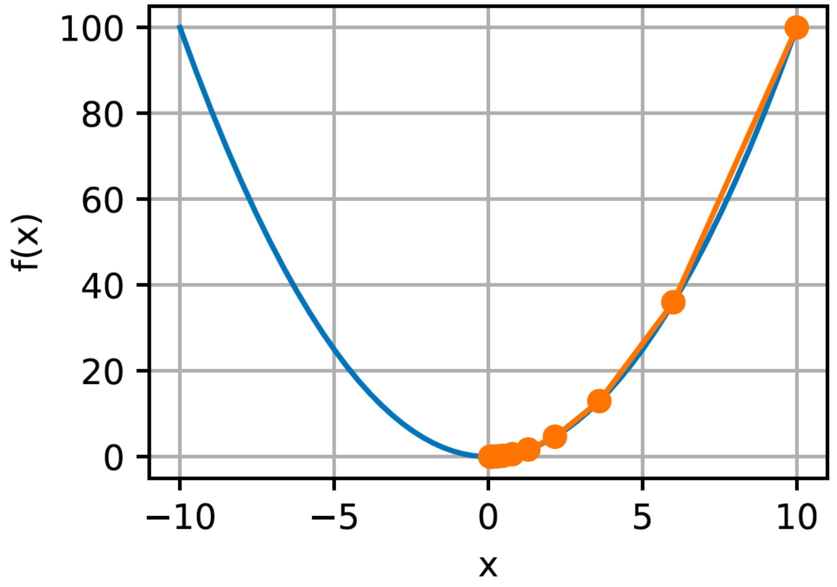
- Jensen's Inequality: It amounts to generalization of the definition of convexity. $\sum_i \alpha_i f(x_i) \geq f\left(\sum_i \alpha_i x_i\right)$ and $\mathrm{E}_x[f(x)] \geq f\left(\mathrm{E}_x[x]\right)$
- In other words, the expectation of a convex function is larger than the convex function of an expectation.
- Can you prove this?

Gradient Descent

- Let’s start with an example in one dimension to explain why the gradient descent algorithm may reduce the value of the objective function.
  - Prove
  - Hint: Use Taylor’s series expansion around $x + \epsilon$ and then replace ϵ with $-\eta f'(x)$.

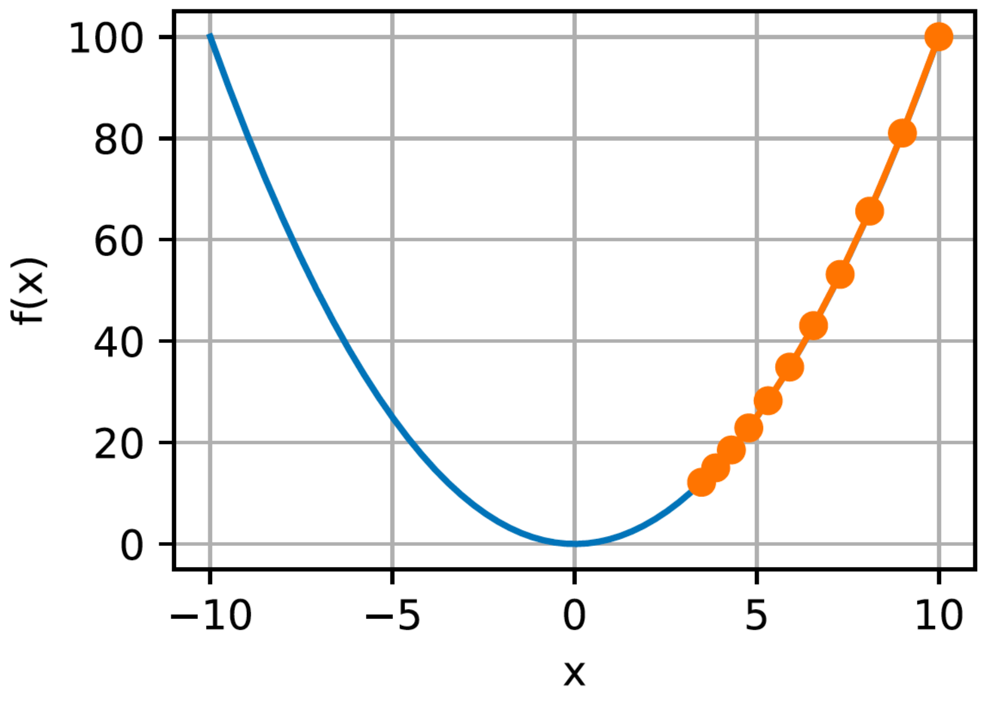
Gradient Descent

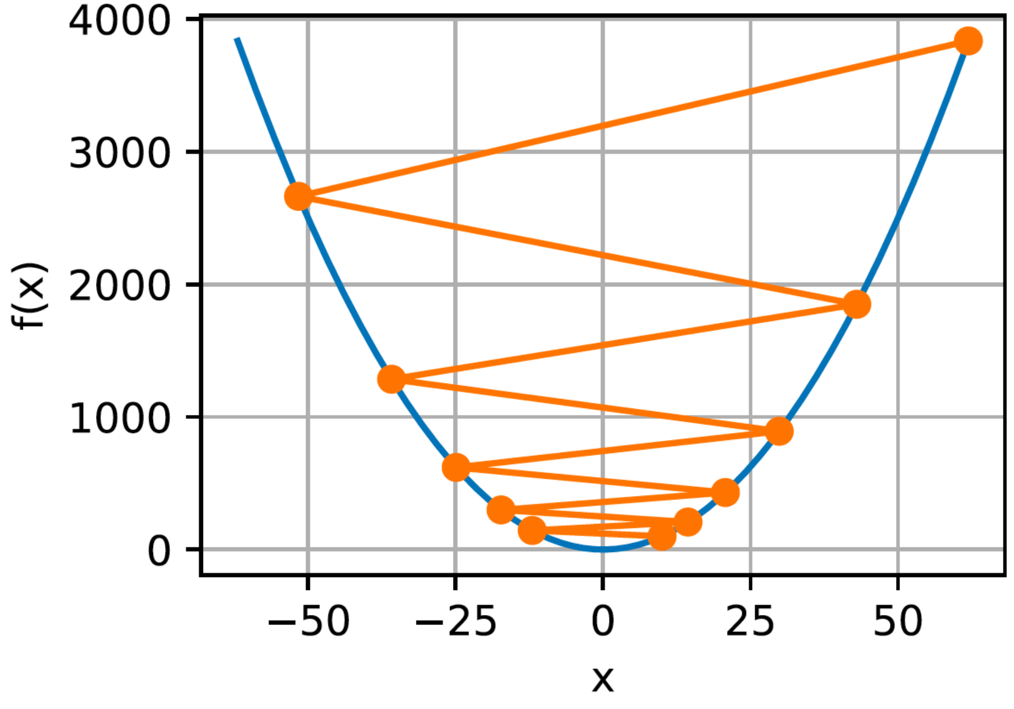
- Let’s start with an example in one dimension to explain why the gradient descent algorithm may reduce the value of the objective function.
  - Prove
  - Hint: Use Taylor’s series expansion around $x + \epsilon$ and then replace ϵ with $-\eta f'(x)$.



Learning Rate in Gradient Descent

- Which one has large learning rate and which one has small?





Multivariate Gradient Descent

- Let us consider a situation where $x \in R^d$

Multivariate Gradient Descent

- Let us consider a situation where $x \in R^d$
- The objective function $f: R^d \rightarrow R$ maps vectors into scalars

Multivariate Gradient Descent

- Let us consider a situation where $x \in R^d$
- The objective function $f : R^d \to R$ maps vectors into scalars
- Correspondingly, its gradient is a vector of consisting d partial derivatives

$$\nabla f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_d}\right]^\top.$$

Multivariate Gradient Descent

- Let us consider a situation where $x \in R^d$
- The objective function $f: R^d \to R$ maps vectors into scalars
- Correspondingly, its gradient is a vector of consisting d partial derivatives

$$\nabla f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_d}\right]^\top.$$

- As before in the univariate case, we can use the corresponding Taylor approximation for multivariate functions

$$f(\mathbf{x} + \epsilon) = f(\mathbf{x}) + \epsilon^\top \nabla f(\mathbf{x}) + O(\|\epsilon\|^2).$$

Multivariate Gradient Descent

- Let us consider a situation where $x \in R^d$
- The objective function $f\colon R^d \to R$ maps vectors into scalars
- Correspondingly, its gradient is a vector of consisting d partial derivatives

$$\nabla f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_d}\right]^\top.$$

- As before in the univariate case, we can use the corresponding Taylor approximation for multivariate functions

$$f(\mathbf{x} + \epsilon) = f(\mathbf{x}) + \epsilon^\top \nabla f(\mathbf{x}) + O(\|\epsilon\|^2).$$

- Choosing a positive learning rate yields the gradient descent algorithm

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f(\mathbf{x}).$$

Multivariate Gradient Descent

- Let $f(\mathbf{x}) = \mathrm{x}_1^2 + 2\mathrm{x}_2^2$ be the objective function

Multivariate Gradient Descent

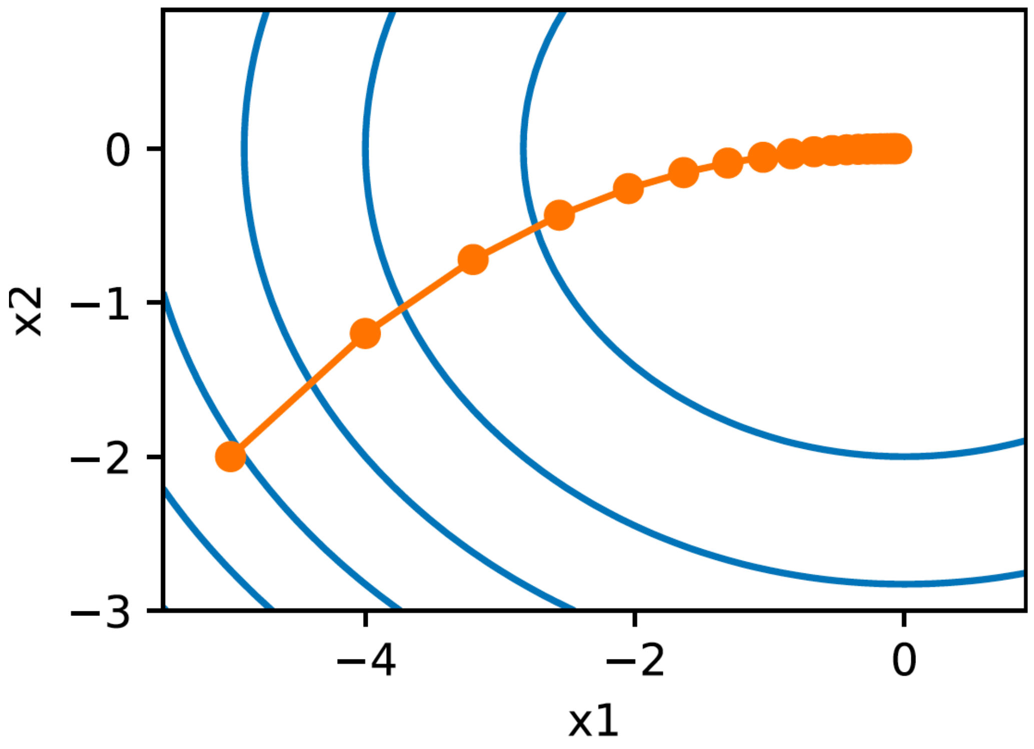
- Let $f(\mathbf{x}) = \mathrm{x}_1^2 + 2\mathrm{x}_2^2$ be the objective function
- What is the gradient?

Multivariate Gradient Descent

- Let $f(\mathbf{x}) = \mathrm{x}_1^2 + 2\mathrm{x}_2^2$ be the objective function
- What is the gradient?
- If the initial position is $[-5, -2]$, what is the next position?

Multivariate Gradient Descent

- Let $f(\mathbf{x}) = x_1^2 + 2x_2^2$ be the objective function
- What is the gradient?
- If the initial position is $[-5, -2]$, what is the next position?



Adaptive Methods

- Getting the learning rate η just right is tricky

Adaptive Methods

- Getting the learning rate η just right is tricky
- What if we determine η automatically?

Adaptive Methods

- Getting the learning rate η just right is tricky
- What if we determine η automatically?
- Second order methods that also look at the value of *curvature* can help.

Adaptive Methods

- Getting the learning rate η just right is tricky
- What if we determine η automatically?
- Second order methods that also look at the value of *curvature* can help.
- They cannot be applied directly to DL due to computational cost

Adaptive Methods

- Getting the learning rate η just right is tricky
- What if we determine η automatically?
- Second order methods that also look at the value of *curvature* can help.
- They cannot be applied directly to DL due to computational cost
- But they provide useful intuition into how to design advanced optimization algorithms

Newton’s method

- Reviewing the Taylor expansion of f there is no need to stop after the first term. In fact, we can write it as

$$f(\mathbf{x}+\epsilon) = f(\mathbf{x}) + \epsilon^\top \nabla f(\mathbf{x}) + \frac{1}{2}\epsilon^\top \nabla\nabla^\top f(\mathbf{x})\epsilon + O(\|\epsilon\|^3)$$

Newton’s method

- Reviewing the Taylor expansion of f there is no need to stop after the first term. In fact, we can write it as

$$f(\mathbf{x}+\epsilon) = f(\mathbf{x}) + \epsilon^\top \nabla f(\mathbf{x}) + \frac{1}{2}\epsilon^\top \nabla\nabla^\top f(\mathbf{x})\epsilon + O(\|\epsilon\|^3)$$

- We define $H_f := \Delta\Delta^T f(\mathbf{x})$ to be the Hessian of f
  - Easy to compute for small d and simple problems
  - For DL, H_f maybe too large and expensive to compute via backprop

Newton’s method

- Reviewing the Taylor expansion of f there is no need to stop after the first term. In fact, we can write it as

$$f(\mathbf{x}+\epsilon) = f(\mathbf{x}) + \epsilon^\top \nabla f(\mathbf{x}) + \frac{1}{2}\epsilon^\top \nabla\nabla^\top f(\mathbf{x})\epsilon + O(\|\epsilon\|^3)$$

- We define $H_f \coloneqq \Delta\Delta^T f(\mathbf{x})$ to be the Hessian of f
  - Easy to compute for small d and simple problems
  - For DL, H_f maybe too large and expensive to compute via backprop
- What is the best ϵ?

Newton's method

- Reviewing the Taylor expansion of f there is no need to stop after the first term. In fact, we can write it as

$$f(\mathbf{x}+\epsilon) = f(\mathbf{x}) + \epsilon^\top \nabla f(\mathbf{x}) + \frac{1}{2}\epsilon^\top \nabla\nabla^\top f(\mathbf{x})\epsilon + O(\|\epsilon\|^3)$$

- We define $H_f \coloneqq \Delta\Delta^T f(\mathbf{x})$ to be the Hessian of f
    - Easy to compute for small d and simple problems
    - For DL, H_f maybe too large and expensive to compute via backprop
- What is the best ϵ?
- Consider $f(x) = \frac{1}{2}x^2$, what is the gradient descent rule using Newton's method?

Newton’s method

- Reviewing the Taylor expansion of f there is no need to stop after the first term. In fact, we can write it as

$$f(\mathbf{x}+\epsilon) = f(\mathbf{x}) + \epsilon^\top \nabla f(\mathbf{x}) + \frac{1}{2}\epsilon^\top \nabla\nabla^\top f(\mathbf{x})\epsilon + O(\|\epsilon\|^3)$$

- We define $H_f := \Delta\Delta^T f(\mathbf{x})$ to be the Hessian of f
  - Easy to compute for small d and simple problems
  - For DL, H_f maybe too large and expensive to compute via backprop
- What is the best ϵ?
- Consider $f(x) = \frac{1}{2}x^2$, what is the gradient descent rule using Newton’s method?
- Preconditioning: computing the inverse of Hessian is expensive. So only use the diagonal entries of Hessian

Stochastic Gradient Descent (SGD)

- In DL, the objective function is often the average of losses per example in the training data set

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

Stochastic Gradient Descent (SGD)

- In DL, the objective function is often the average of losses per example in the training data set

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

- The gradient of the objective function at **x** is computed as $\nabla f(\boldsymbol{x}) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\boldsymbol{x}).$

Stochastic Gradient Descent (SGD)

- In DL, the objective function is often the average of losses per example in the training data set

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

- The gradient of the objective function at **x** is computed as $\nabla f(\boldsymbol{x}) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\boldsymbol{x}).$
- Computation cost of each update is $\mathcal{O}(n)$

Stochastic Gradient Descent (SGD)

- In DL, the objective function is often the average of losses per example in the training data set

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

- The gradient of the objective function at **x** is computed as $\nabla f(\boldsymbol{x}) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\boldsymbol{x}).$
- Computation cost of each update is $\mathcal{O}(n)$
- SGD reduces the computational cost at each iteration
    - At each iteration of SGD, we uniformly sample an index $i \in \{1, \dots, n\}$ for data instances at random
    - Compute the gradient $\nabla f_i(\mathbf{x})$ to update $\mathbf{x}$

Stochastic Gradient Descent (SGD)

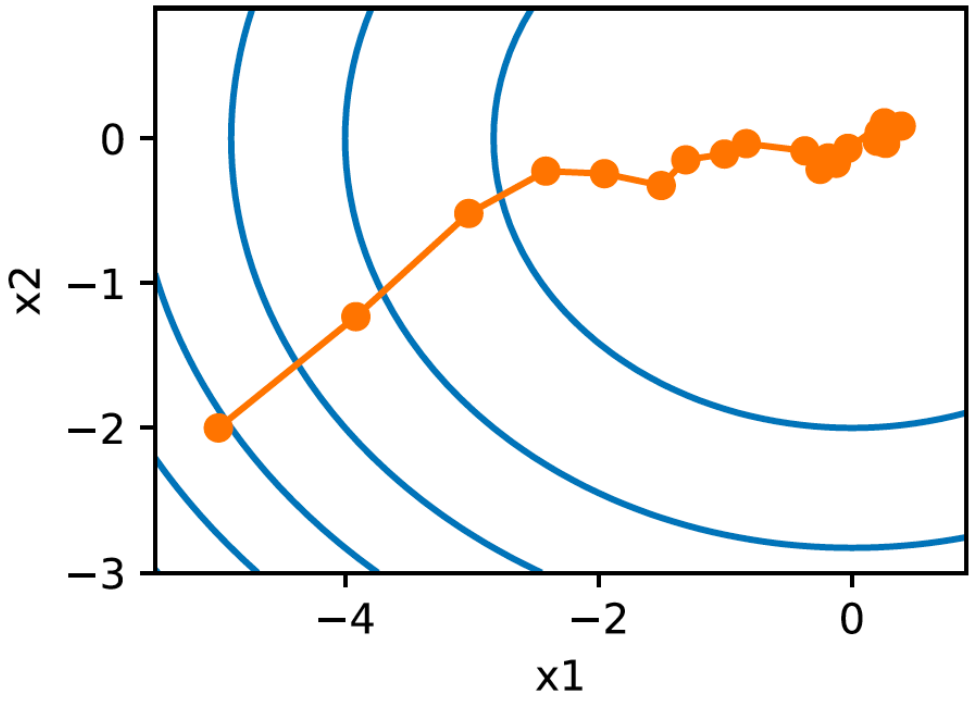
- What is the computation cost per update?

Stochastic Gradient Descent (SGD)

- What is the computation cost per update?
- Is the stochastic gradient $\nabla f_i(\mathbf{x})$ unbiased estimate of $\nabla f(\mathbf{x})$?

Stochastic Gradient Descent (SGD)

- What is the computation cost per update?
- Is the stochastic gradient $\nabla f_i(\mathbf{x})$ unbiased estimate of $\nabla f(\mathbf{x})$?
- The trajectory of SGD looks more noisy that GD



Mini-batch Stochastic Gradient Descent

- We can perform random uniform sampling for each iteration to form a minibatch and the use this minibatch to compute the gradient

Mini-batch Stochastic Gradient Descent

- We can perform random uniform sampling for each iteration to form a minibatch and the use this minibatch to compute the gradient
- In each subsequent time step $t > 0$, mini-batch SGD uses random uniform sampling to get a mini-batch $\mathcal{B}_t$ made up of example indices from the training set

Mini-batch Stochastic Gradient Descent

- We can perform random uniform sampling for each iteration to form a minibatch and the use this minibatch to compute the gradient
- In each subsequent time step $t > 0$, mini-batch SGD uses random uniform sampling to get a mini-batch $\mathcal{B}_t$ made up of example indices from the training set
- Then we compute the gradient $\mathbf{g}_t$ of the objective function at $\mathbf{x}_{t-1}$ with mini-batch $\mathcal{B}_t$ at time step t:

$$\boldsymbol{g}_t \leftarrow \nabla f_{\mathcal{B}_t}(\boldsymbol{x}_{t-1}) = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \nabla f_i(\boldsymbol{x}_{t-1})$$

Mini-batch Stochastic Gradient Descent

- We can perform random uniform sampling for each iteration to form a minibatch and the use this minibatch to compute the gradient
- In each subsequent time step $t > 0$, mini-batch SGD uses random uniform sampling to get a mini-batch $\mathcal{B}_t$ made up of example indices from the training set
- Then we compute the gradient $\mathbf{g}_t$ of the objective function at $\mathbf{x}_{t-1}$ with mini-batch $\mathcal{B}_t$ at time step t:

$$\boldsymbol{g}_t \leftarrow \nabla f_{\mathcal{B}_t}(\boldsymbol{x}_{t-1}) = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \nabla f_i(\boldsymbol{x}_{t-1})$$

- Given the learning rate η_t, the iteration of the mini-batch SGD is:

$$\boldsymbol{x}_t \leftarrow \boldsymbol{x}_{t-1} - \eta_t \boldsymbol{g}_t.$$

Mini-batch Stochastic Gradient Descent

- We can perform random uniform sampling for each iteration to form a minibatch and the use this minibatch to compute the gradient
- In each subsequent time step $t > 0$, mini-batch SGD uses random uniform sampling to get a mini-batch $\mathcal{B}_t$ made up of example indices from the training set
- Then we compute the gradient $\mathbf{g}_t$ of the objective function at $\mathbf{x}_{t-1}$ with mini-batch $\mathcal{B}_t$ at time step t:

$$\boldsymbol{g}_t \leftarrow \nabla f_{\mathcal{B}_t}(\boldsymbol{x}_{t-1}) = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \nabla f_i(\boldsymbol{x}_{t-1})$$

- Given the learning rate η_t, the iteration of the mini-batch SGD is:

$$\boldsymbol{x}_t \leftarrow \boldsymbol{x}_{t-1} - \eta_t \boldsymbol{g}_t.$$

- SGD can self-decay itself by using $\eta_t = \eta t^{\alpha}$ (usually $\alpha = -1$ or $\alpha = -0.5$), $\eta_t = \eta \alpha^t$ (e.g. $\alpha = 0.95$)