Gradient-Based Learning

BOSTON UNIVERSITY
MACHINE INTELLIGENCE
COMMUNITY

Justin Chen, Brian Fakhoury

Sept. 19, 2017

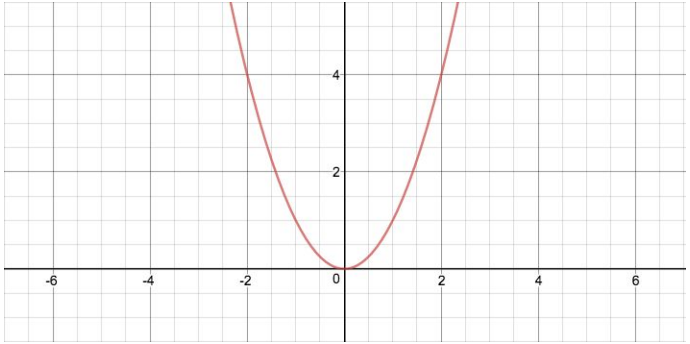
Univariate Functions

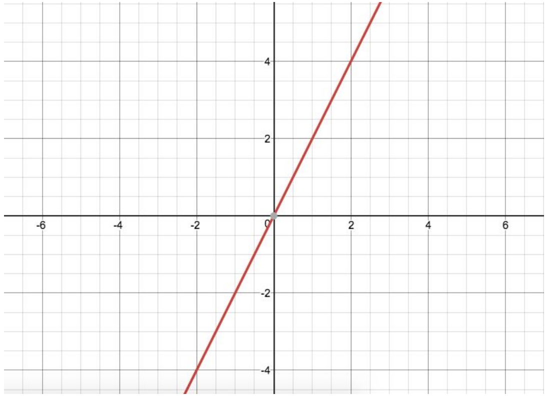
Univariate function

$$f(x) = x^2$$

Derivative

$$\frac{\delta f}{\delta x} = 2\text{x}$$





MACHINE INTELLIGENCE COMMUNITY

Multivariate functions

Multivariate function

Function of more than one variable/ **dimension**

$$f(\bar{x}) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 ...$$

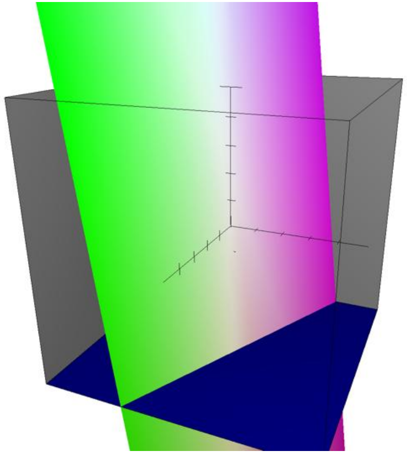
Multivariate derivative

Shows us slope of a function in all dimensions

$$\nabla f = \begin{bmatrix} \frac{\delta f}{\delta x_0} \\ \frac{\delta f}{\delta x_1} \\ \vdots \\ \frac{\delta f}{\delta x_n} \end{bmatrix}$$

$$f(x_0, x_1) = 4x_0 + 8x_1$$

$$\nabla f = [4,8]$$



MACHINE INTELLIGENCE COMMUNITY

Linear Equations in PYTORCH

Generate random matrix

```
>> import torch
>> r = torch.randn(2,3)

-0.2220  1.3369 -1.3627
 0.0863  0.8932 -0.6577
[torch.FloatTensor of size 2x3]
```

$$f(\bar{x}) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 ...$$

```
        >> import torch.nn as nn
        >> f = nn.Linear(2,1)
        >> [x.data for x in a.parameters()]
        [
           0.1685
weights { -0.0136
        [torch.FloatTensor of size 2x1]
        ,
          -0.5899
bias    { -0.8569
        [torch.FloatTensor of size 2]
        ]
```

Cost Functions

- Learning algorithms need to measure how wrong it is in order to improve the model
- **Cost function** measures **error** distance between **hypothesis** and **label** (correct answer)
- For example
  - Learning algorithm’s hypothesis: [0.1, 0.2, 0.3]
  - Correct answers: [0.5, 0.8, 0.9]
  - Errors: [0.4, 0.6, 0.6]
  - Least Square Error (LSE): $\frac{1}{2}[(0.1-0.5)^2+(0.2-0.8)^2+(0.3-0.9)^2]$

Different Cost Functions

- Common terms: **cost/ objective/ loss function**
- Tailored for the model
- Common cost functions:

Regression: **Least Square Error**

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

Classification: **Negative Log Likelihood**

$$J(\theta) = \sum_{i=1}^{m}(y^{(i)}log(h_\theta(x^{(i)})) + (1 - y^{(i)})log(1 - h_\theta(x^{(i)})))$$

Loss Functions in PYTORCH

Negative Log Likelihood

$$J(\theta) = \sum_{i=1}^{m}(y^{(i)}log(h_\theta(x^{(i)})) + (1 - y^{(i)})log(1 - h_\theta(x^{(i)})))$$

```
>> import torch.nn.functional as F
>> from torch.autograd import Variable
>> input = Variable(torch.randn(3, 5))
>> target = autograd.Variable(torch.LongTensor([1, 0, 4]))
>> loss = F.nll_loss(input, target)
Variable containing:
 2.2282
[torch.FloatTensor of size 1]
```

Optimizing Cost Function

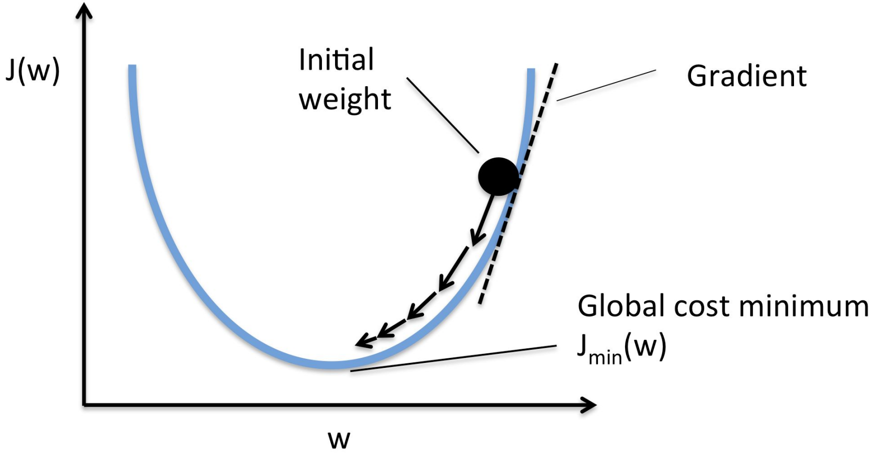
- Goal is to **minimize cost function** J
  - Compute derivative of J w.r.t. parameters θ

$$\nabla J = \frac{\delta J}{\delta \theta}$$

- Consider this simple cost function

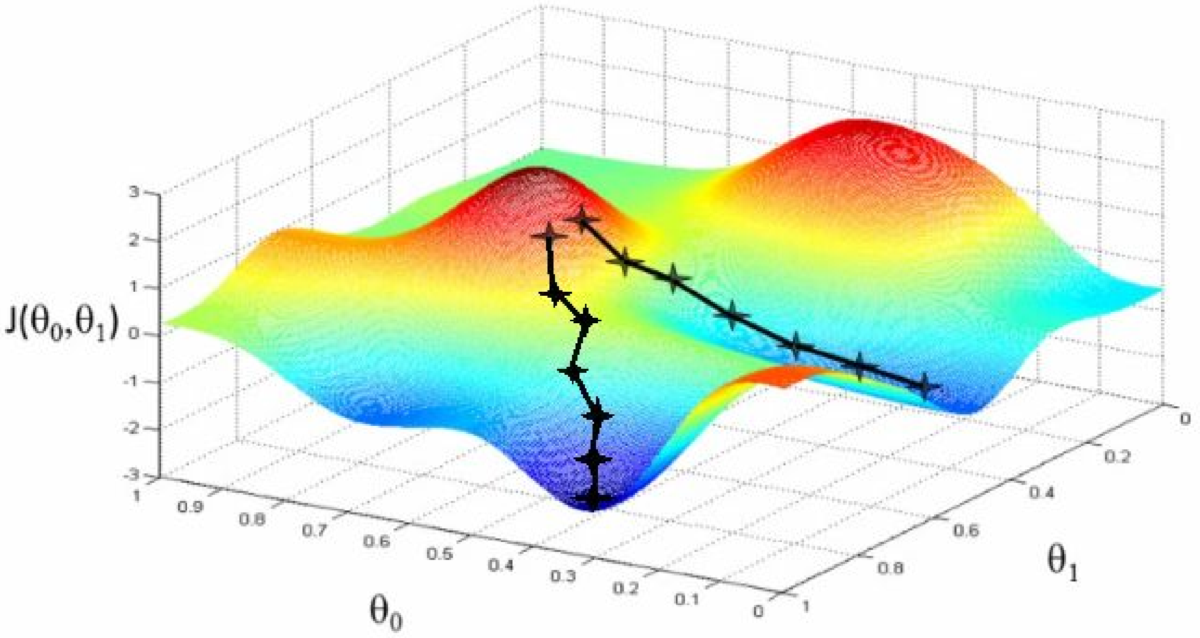
$$f(x) = x^2 \quad \longrightarrow \quad \frac{\delta f}{\delta x} = 2\text{x}$$

  - Solve derivative for 0
  - **Convex** functions have single **global minima**
  - Most **cost landscapes** are **non-convex** - contain many **local minima (exponentially many in the number of dimensions)**



https://sebastianraschka.com/faq/docs/closed-form-vs-gd.html

Non-convex Optimization



Gradient Descent

$$\theta_j := \theta_j - \alpha \overbrace{\frac{\partial}{\partial \theta_j} J(\theta)}^{\text{Also written } \nabla J(\theta)}$$

Scalar learning rate

Individual weights

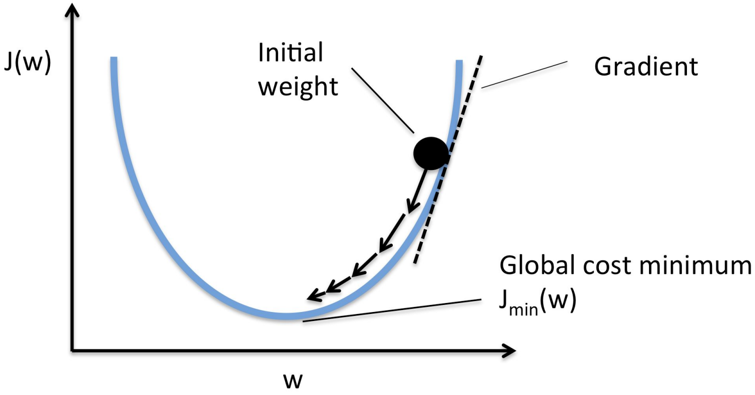
Cost/objective/loss function

Vector of weights

https://github.com/bumic/tutorials/blob/master/bumic-2/bumic-2.pdf

Gradient Descent

- Once we have a function for $\sigma J/\sigma W$, we can use it to iteratively move “downhill” and therefore minimize our cost function.
- This process of looking at a cost function and moving towards the best values for weights is gradient descent, and makes machine learning work beyond just 1 or 2 dimensions
  - If a deep neural network has 10 layers, it would take longer than the age of the universe to minimize the cost function with a linear method.

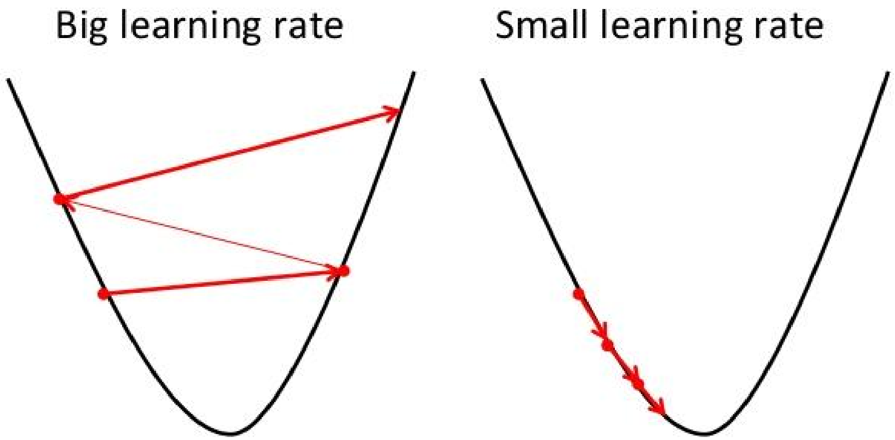


https://sebastianraschka.com/faq/docs/closed-form-vs-gd.html

Learning rate

- How fast should I learn?
  - Gradient points in direct of steepest descent, but does not indicate magnitude of step
- Multiply the learning rate to slow down how fast our network tries to compensate for a given piece of data
- Typical learning rates to try:

  0.1, 0.01, 0.001, 0.0001

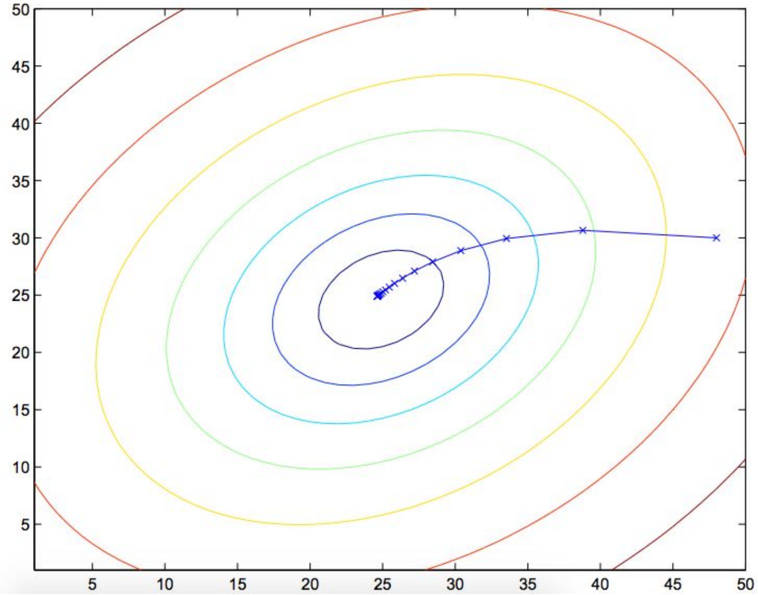


Batch Gradient Descent (GD)

$$Loop\,\{$$

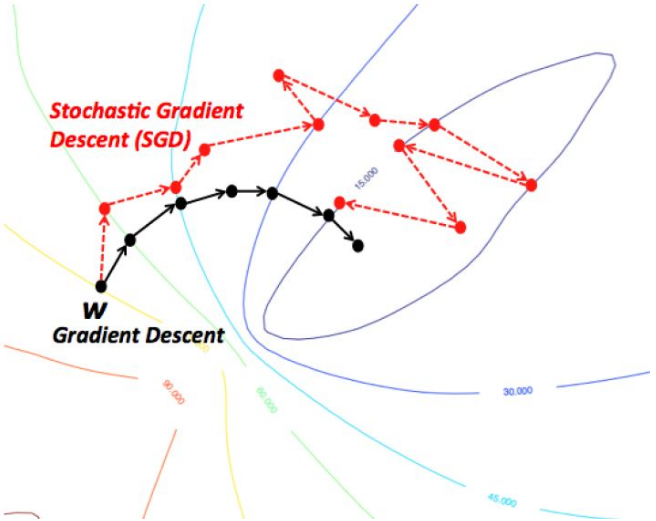
$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

$$\}$$



Stochastic Gradient Descent

- Calculate the gradient using one data sample at a time
  - Takes many iterations to go over entire data set
  - Each time the full data set is covered is an “epoch”
- Works better when there are many minima in a complex “manifold”
- Makes for as noisier descent, which can be useful for
  - Noise can be reduced with mini-batch
    - Use 8 or 16 samples at a time
  - Noise can be useful for “saddle points”



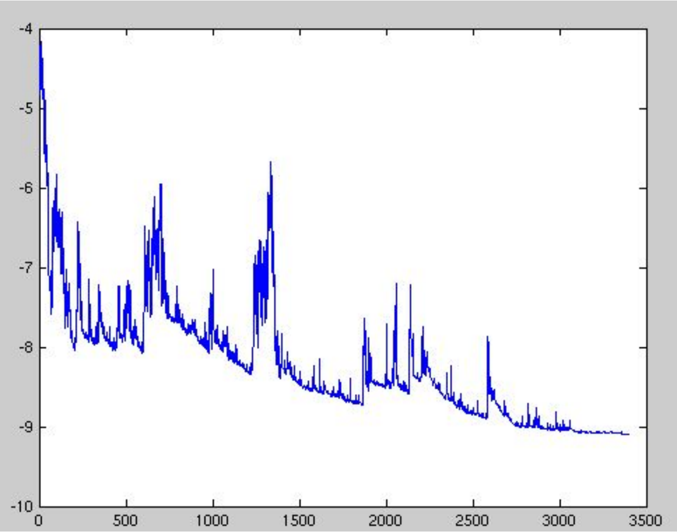
http://www.bogotobogo.com/python/scikit-learn/scikit-learn_batch-gradient-descent-versus-stochastic-gradient-descent.php

MACHINE INTELLIGENCE COMMUNITY

Stochastic Gradient Descent (SGD)

Size of dataset

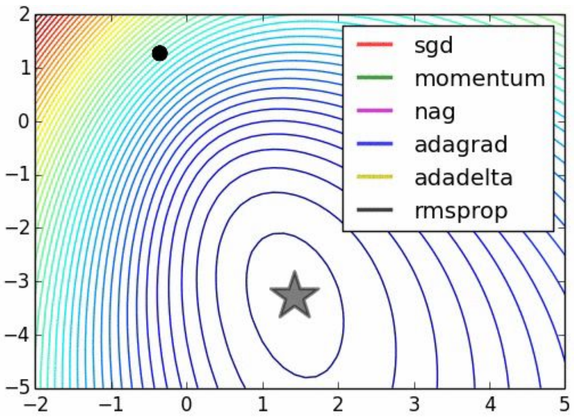
```
$Loop\{$
    $for\ i = 1\ to\ m\ \{$
        $\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$
    $\}$
$\}$
```



https://upload.wikimedia.org/wikipedia/commons/f/f3/Stogra.png

Different types of Gradient Descent

- The concept of gradient descent can be actualized in many different ways. Two popular, and simple, implementations of gradient descent are:
  - Batch gradient descent (which we have been describing so far)
  - Stochastic gradient descent
- A third variety is mini-batch gradient descent
  - Still simple, though more abstract in why it works
  - Between batch and gradient, select batches at a time (ex. 16 out of 100,000)
- For each method, there are multiple algorithms used to optimize the descent calculation

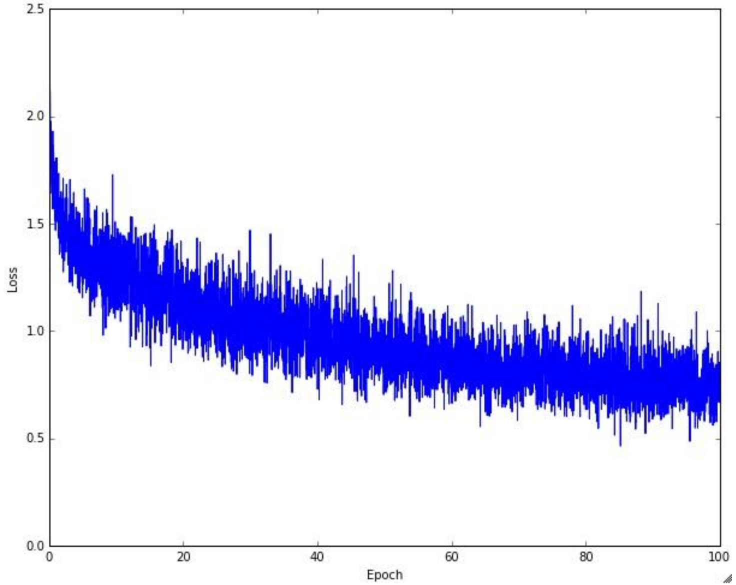


http://blog.hackerearth.com/3-types-gradient-descent-algorithms-small-large-data-sets

Mini-Batch SGD

Size of dataset

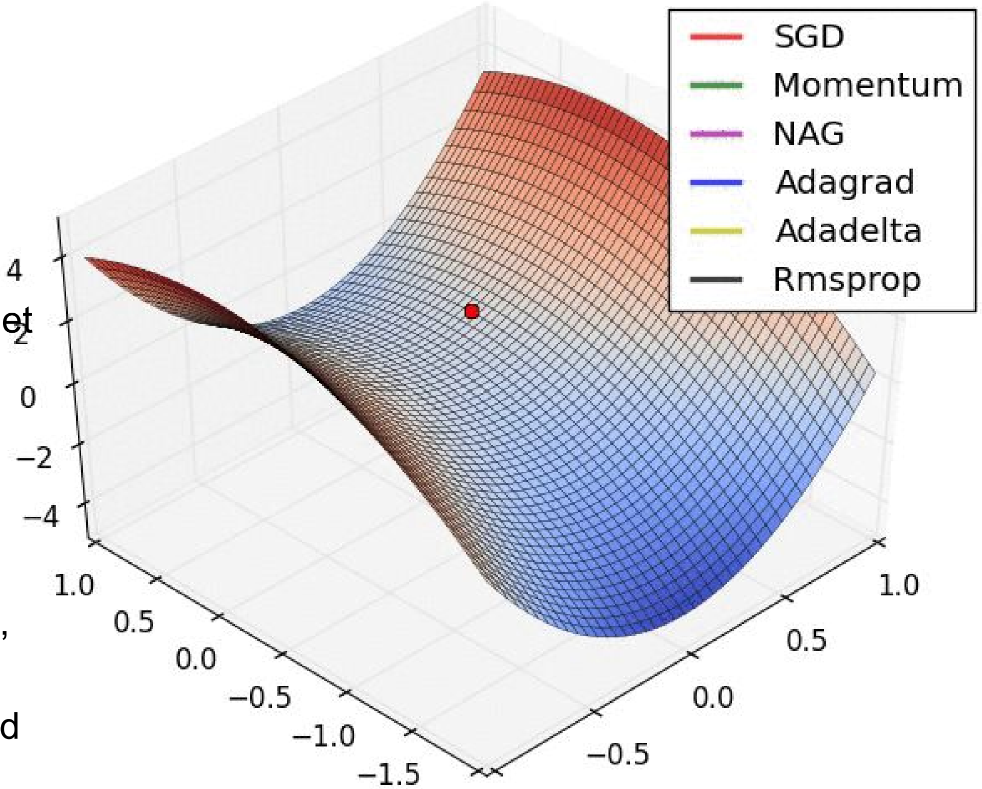
```
$Loop\{$
    $for\ i = 1\ to\ m\ \{$
        $\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$
    $\}$
$\}$
```



http://cs231n.github.io/neural-networks-3/

Saddle points

- Can be a big issue for gradient descent.
  - If you are right on the saddle, the gradient does not always help you get off
- Can exist above one dimension
- Many solutions to this problem exist
  - A simple one is by just adding noise, you can force the algorithm to randomly “catch on” to the downward slope



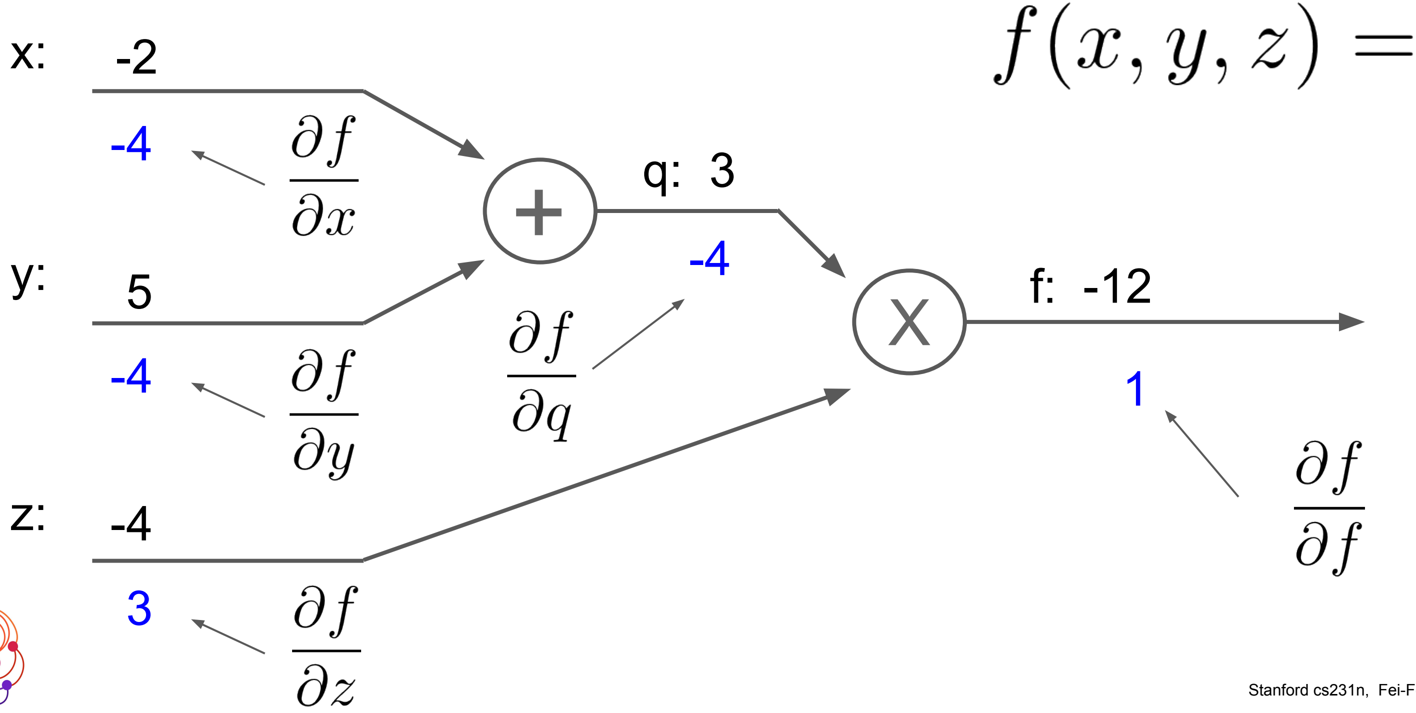
http://cs231n.github.io/neural-networks-3/

Backpropagation (BP)

- Method for calculating the error contribution of a single computation unit
  - We can use this information to update weights (parameters), and get closer to a model with less loss
- Neurons (computational units) that contributed more to the error will be changed more by BP
  - Scaled by the learning rate, how much should the weight be adjusted

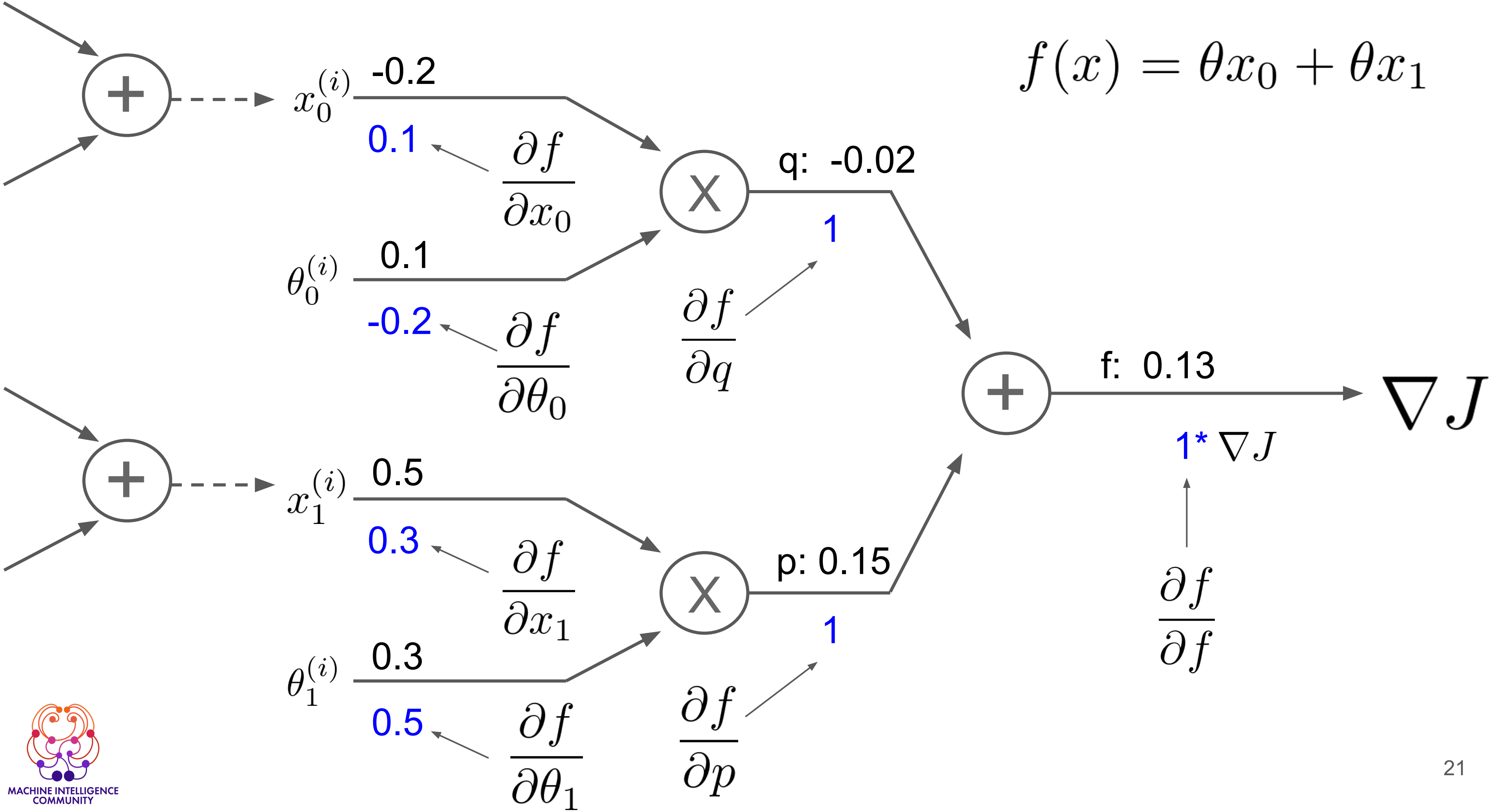
Derivatives of Computation Graph

$$f(x, y, z) = (x + y)z$$

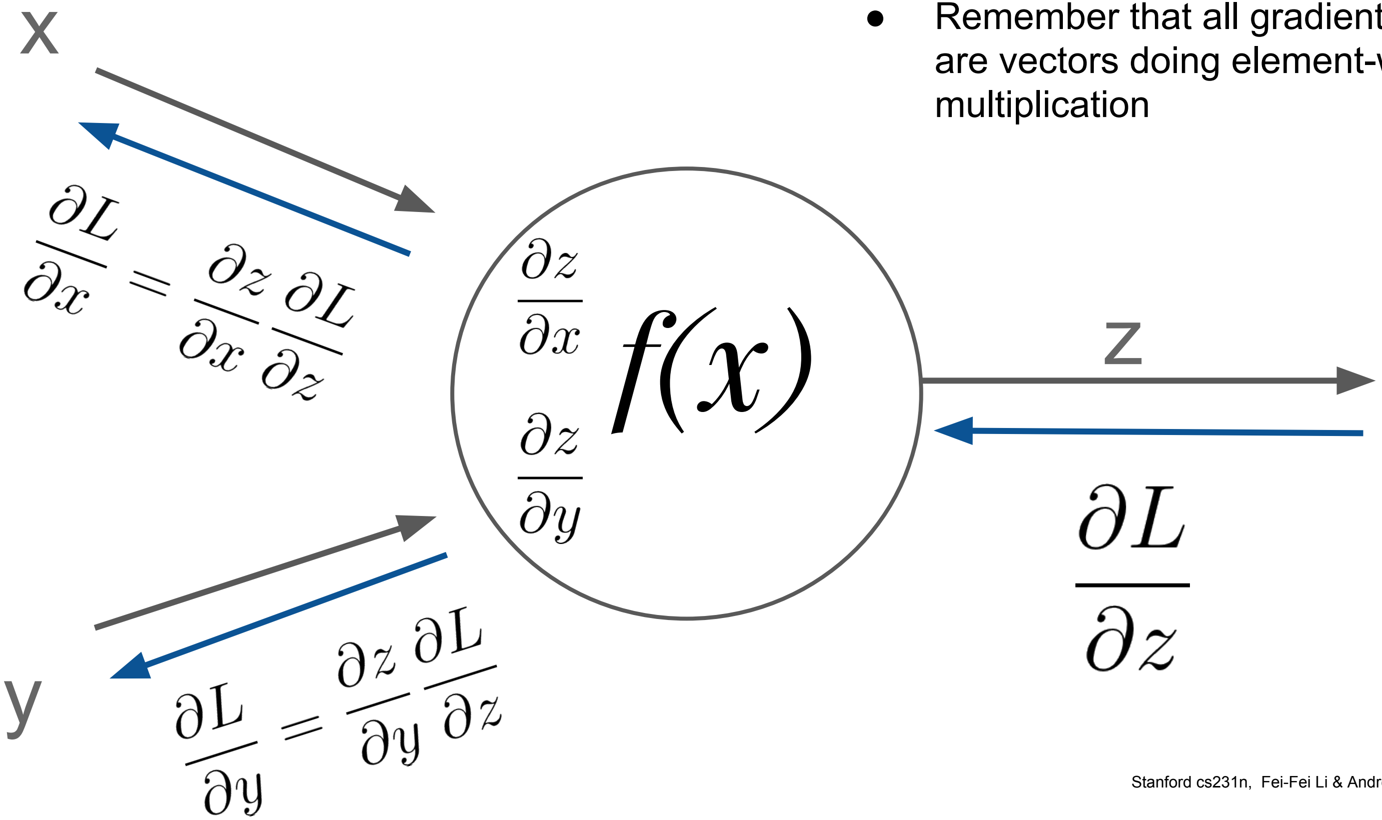

x: -2
-4
$\frac{\partial f}{\partial x}$
y: 5
-4
$\frac{\partial f}{\partial y}$
z: -4
3
$\frac{\partial f}{\partial z}$
+
q: 3
-4
$\frac{\partial f}{\partial q}$
X
f: -12
1
$\frac{\partial f}{\partial f}$

MACHINE INTELLIGENCE COMMUNITY

$$f(x) = \theta x_0 + \theta x_1$$


+
$x_0^{(i)}$
-0.2
0.1
$\frac{\partial f}{\partial x_0}$
$\theta_0^{(i)}$
0.1
-0.2
$\frac{\partial f}{\partial \theta_0}$
X
q: -0.02
1
$\frac{\partial f}{\partial q}$
+
f: 0.13
∇J
1* ∇J
$\frac{\partial f}{\partial f}$
+
$x_1^{(i)}$
0.5
0.3
$\frac{\partial f}{\partial x_1}$
$\theta_1^{(i)}$
0.3
0.5
$\frac{\partial f}{\partial \theta_1}$
X
p: 0.15
1
$\frac{\partial f}{\partial p}$

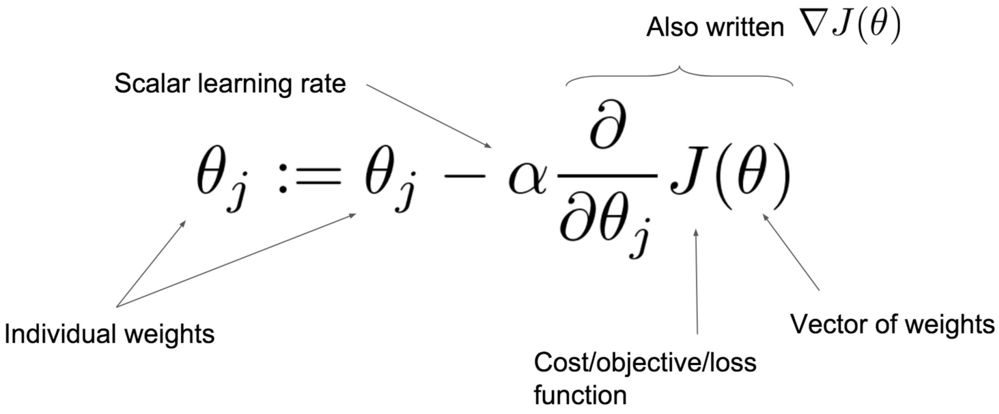
Local Gradient


x
y
z
$f(x)$
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
$\frac{\partial L}{\partial z}$
$\frac{\partial L}{\partial x} = \frac{\partial z}{\partial x}\frac{\partial L}{\partial z}$
$\frac{\partial L}{\partial y} = \frac{\partial z}{\partial y}\frac{\partial L}{\partial z}$

- Remember that all gradients are vectors doing element-wise multiplication

Gradient Descent + Backpropagation

- Now, we can adjust a weight based off our gradient descent calculations!
- θ_j can be updated as its old weight minus the change in cost function in respect to itself



$$\theta_j := \theta_j - \alpha \overbrace{\frac{\partial}{\partial \theta_j} J(\theta)}^{\text{Also written } \nabla J(\theta)}$$

Scalar learning rate

Individual weights

Cost/objective/loss function

Vector of weights

https://github.com/bumic/tutorials/blob/master/bumic-2/bumic-2.pdf

SGD + BP with PYTORCH

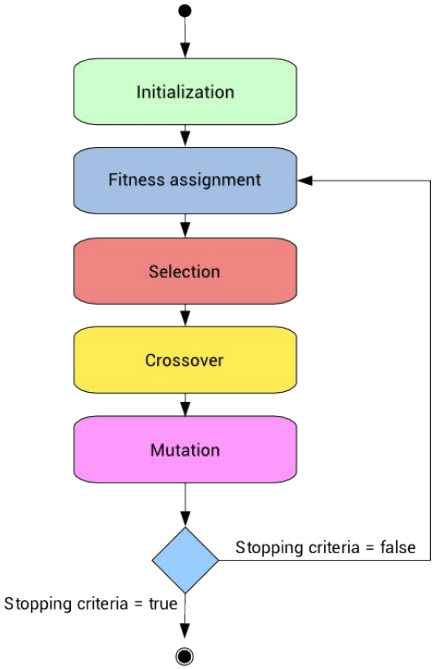
```
>> optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
>> optimizer.zero_grad()
>> loss = F.nll_loss(input, target)
>> loss.backward()
>> optimizer.step()
```

1. Define optimizer
2. **Clear local gradients**
3. Compute loss
4. Backpropagate error
5. Apply gradients to parameters

http://pytorch.org/docs/master/optim.html?highlight=sgd#torch.optim.SGD

Anything else besides Gradient Descent?

- Yes, but GD is fairly easy and effective
- Another cool option is using evolution models, or “genetic algorithms”
  - Make random mutations to your model to produce many new generation models
  - Test each one's’ “fitness”
  - Kill off the bad performers, evolve the good ones (feature selection)



https://www.neuraldesigner.com/blog/genetic_algorithms_for_feature_selection

MACHINE INTELLIGENCE COMMUNITY

References & Further Reading

[1] Ruder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747 (2016).

[2] Sun, Xu, et al. "meProp: Sparsified Back Propagation for Accelerated Deep Learning with Reduced Overfitting." arXiv preprint arXiv:1706.06197 (2017).

[3] Kingma, Diederik, and Jimmy Ba. "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).

[4] Zeiler, Matthew D. "ADADELTA: an adaptive learning rate method." arXiv preprint arXiv:1212.5701 (2012).

[5] Du, Simon S., et al. "Gradient Descent Can Take Exponential Time to Escape Saddle Points." arXiv preprint arXiv:1705.10412 (2017).

[6] Dean, Jeffrey, et al. "Large scale distributed deep networks." Advances in neural information processing systems. 2012.

[7] https://www.youtube.com/watch?v=i94OvYb6noo

[8] https://www.youtube.com/watch?v=5u4G23_OohI

Shoutout to our sponsors

Boston University Software Application and Innovation Lab

Boston University SPARK!

Upcoming Events

MIC Paper signup: https://goo.gl/iAm6TL
BUMIC Projects signup: https://goo.gl/GmP9oK

MIT MIC reading group:

Paper: Learning from Simulated and Unsupervised Images through Adversarial Training
Location: MIT 56-154 (building 56, room 154)
Date: 9.21.17 Time: 5 PM

BU MIC reading group:

Paper: SimRank Computation on Uncertain Graphs
Location: BU Hariri Seminar Room
Date: 9.22.17 Time: 7 PM

Next workshop:

Location: BU Hariri Seminar Room
Date: 9.26.17 Time: 7 PM