

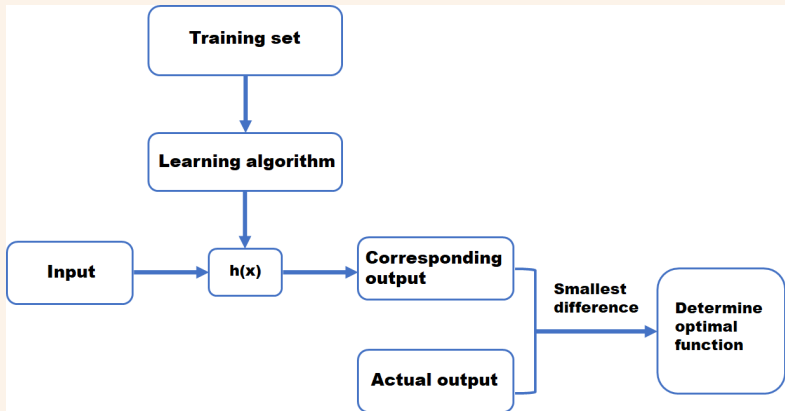
Gradient Descent in Machine Learning

MATH214 – Spring 2023

pgro

Motivation: Want machines to make more accurate predictions after fed with training sets.

Outline of gradient descent algorithms:



Why linear regression?

- Simple and proper way to get the information from training set
- Related to the linear algebra

Theorem

Hypothesis function:

$$h_{\theta}(x) := \theta_0 + \sum_{i=1}^m (\theta_i x_i)$$

*x represents several elements that affect final result,
and theta represents corresponding weight of each element.*

Target: Want to make prediction more accurate.

Mathematical understanding: Make the difference between predicted value and actual one as small as possible.

Method: Use a special function to achieve it.

So, we introduce a new function,

Theorem

Loss function:

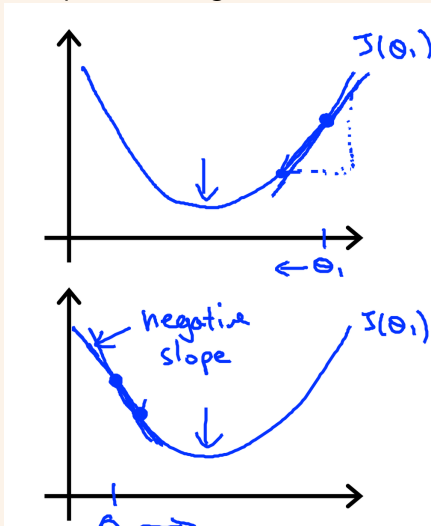
$$J(\theta_0, \theta_1, \dots, \theta_m) := \frac{1}{2n} \sum_{j=1}^n ((h_{\theta}(x_0^{(j)}, x_1^{(j)}, x_m^{(j)}) - y^{(j)})^2$$

n equals to the size of training set.

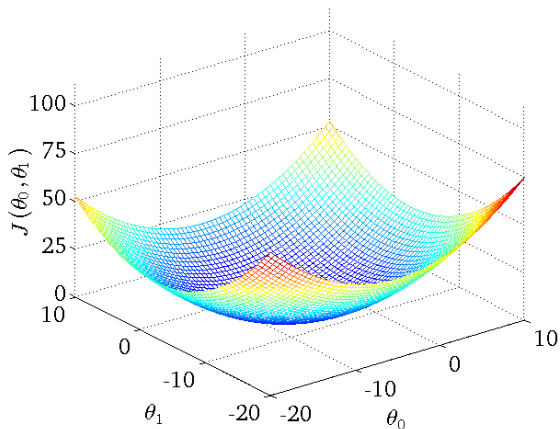
Some properties of loss function:(under the case of linear regression)

- convexity

One-parameter figure:



Two-parameters figure:



Convexity guarantees that we can find the global minimum.

Theorem

For θ_j ,

$$\theta_j \leftarrow \theta_j - \alpha \frac{d}{d\theta_j} J(\theta_0, \theta_1, \dots, \theta_m)$$

alpha represents the learning efficiency.

After calculation, we find that, when j equals to 0,

$$\frac{d}{d\theta_j} J(\theta_0, \theta_1, \dots, \theta_m) = \frac{1}{n} \sum_{i=1}^n (h_{\theta}(x_0^{(i)}, x_1^{(i)}, \dots, x_m^{(i)}) - y^{(i)}),$$

when j is an integer larger than 0,

$$\frac{d}{d\theta_j} J(\theta_0, \theta_1, \dots, \theta_m) = \frac{1}{n} \sum_{i=1}^n (h_{\theta}(x_0^{(i)}, x_1^{(i)}, \dots, x_m^{(i)}) - y^{(i)}) x_j^{(i)},$$

repeat until convergence.

Why vectorizing?

- Simpler notations.
- Faster training speed in programming.

$$X = \begin{bmatrix} x^{(1)} \\ x^{(2)} \\ \vdots \\ x^{(n)} \end{bmatrix} = \begin{bmatrix} x_0^{(1)} & x_1^{(1)} & \cdots & x_m^{(1)} \\ x_0^{(2)} & x_1^{(2)} & \cdots & x_m^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_0^{(n)} & x_1^{(n)} & \cdots & x_m^{(n)} \end{bmatrix}, y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix}, \theta = \begin{bmatrix} \theta^{(1)} \\ \theta^{(2)} \\ \vdots \\ \theta^{(m)} \end{bmatrix}$$

$$\begin{aligned} \theta_j &:= \theta_j - \alpha \sum_{i=1}^n (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} \\ &= \theta_0 - \alpha \begin{bmatrix} x_j^{(1)} & x_j^{(2)} & \cdots & x_j^{(n)} \end{bmatrix} (h_{\theta}(x) - y) \\ &= \theta_0 - \alpha \begin{bmatrix} x_j^{(1)} & x_j^{(2)} & \cdots & x_j^{(n)} \end{bmatrix} (x \cdot \theta - y) \end{aligned}$$

Synthesizing all $j \in [i, n]$,

$$\begin{bmatrix} \theta^{(1)} \\ \theta^{(2)} \\ \vdots \\ \theta^{(m)} \end{bmatrix} := \begin{bmatrix} \theta^{(1)} \\ \theta^{(2)} \\ \vdots \\ \theta^{(m)} \end{bmatrix} - \alpha \begin{bmatrix} x_0^{(1)} & x_0^{(1)} & \cdots & x_0^{(n)} \\ x_1^{(1)} & x_1^{(1)} & \cdots & x_1^{(n)} \\ \vdots & \vdots & \ddots & \vdots \\ x_m^{(1)} & x_m^{(2)} & \cdots & x_m^{(n)} \end{bmatrix} (X \cdot \theta - y)$$

Theorem

Vectorized form of gradient descent in linear regression.

$$\theta := \theta - \alpha X^\top (X\theta - y)$$



Figure 1

- MNIST is a famous dataset used for image recognition and classification tasks. It consists of 70,000 (60,000 training+10,000 testing) handwritten digits in a 28x28 px grayscale image.
- Our project application is to use CNN model predict MNIST handwritten digits. In the application, we discover how Gradient Descent applies in DL.
- Project source code repo:
<https://github.com/openhe-hub/math214-project.git>

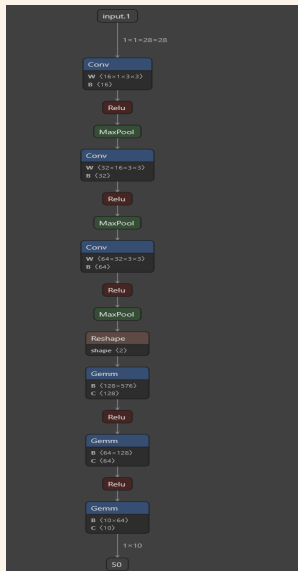


Figure 2 CNN Model (visualized by Netron)

- Our CNN Model:
3 *conv_layers* (Conv+Relu+MaxPool)
+1 *fc_layer* (Linear+Relu)
- Loss function: *CrossEntropy*
- Gradient Descent: *SGD*
- Other Parameters:
 - *epochs* = 30
 - *batch_size* = 64
 - *learning_rate* = 0.001

Theorem

SGD Iteration Formula: $w_{t+1} = w_t - \eta \nabla L(w_t)$

Algorithm 1: SGD algorithm for training a CNN

Data: Input data x and labels y

Result: Trained CNN model

```
1 Function Train():  
2   for  $epoch = 1$  to  $epochs$  do  
3     for  $batch\_i = (x_i, y_i)$  in dataset do  
4       Set gradients of network parameters to zero;  
5       Forward propagation:  $\hat{y} = \text{forward}(x_i)$ ;  
6       Compute loss function:  $loss = L(\hat{y}, y_i)$ ;  
7       Backward propagation:  $loss.backward()$ ;  
8       Update parameters using SGD:  $w = w - \eta \nabla L(w)$ ;  
9     end  
10  end
```

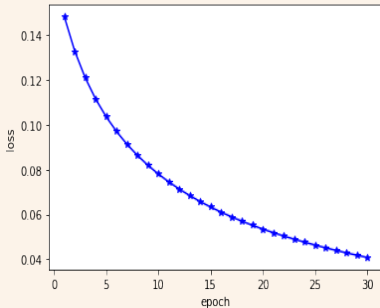


Figure 3 SGD epoch-loss plot

The epoch-loss plot shows how the model's loss function changes over time, as it updates the model parameters using small batches of training data. The plot typically displays a downward trend, with rapid loss reductions in initial epochs, and slower reductions as the algorithm converges.

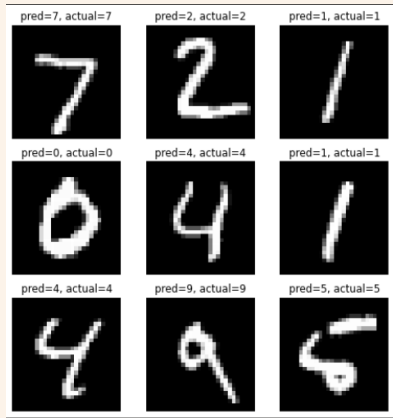
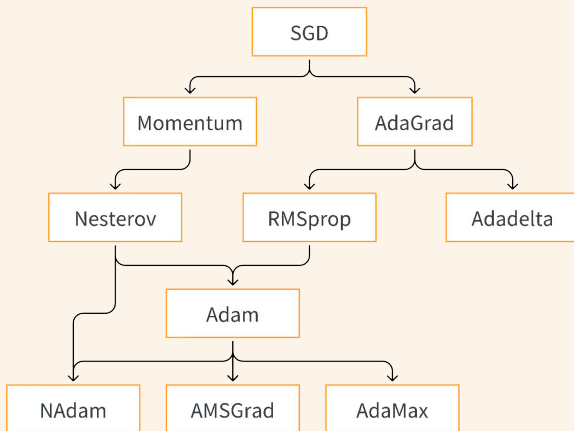


Figure 4 Testing set prediction result

- Result: The total loss is about 0.0413, and the accuracy is about 98.67%. Algorithms (SGD) based on Gradient Descent works well in our CNN model.
- Improvement: Consider adjusting parameters like *learning_rate*, *batch_size*. Also, we may change different GD algorithms like *Adam*.

Motivation: Wants to improve learning efficiency (always!), reduce fluctuation, avoid local minima, etc.

Improved gradient descent algorithms:



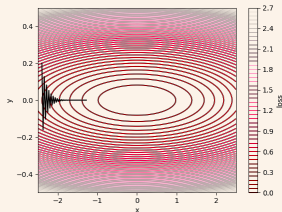
Theorem

For θ_j ,

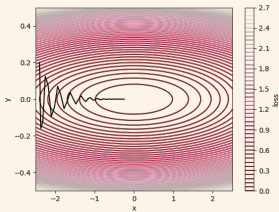
$$v_j \leftarrow \eta v_j - \alpha \nabla_{\theta_j} J(\theta)$$

$$\theta_j \leftarrow \theta_j - v_j$$

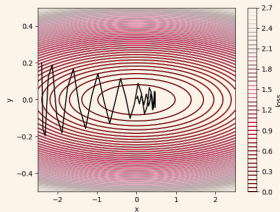
Illustration: Keep accumulating momentum (velocity) when rolling down a slope.



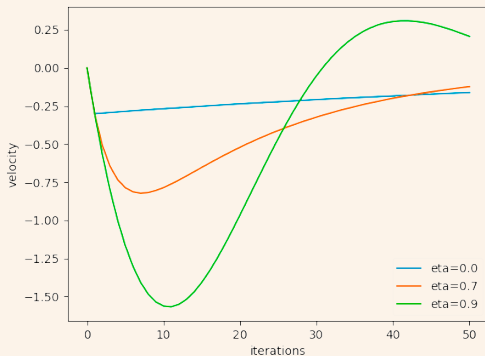
$\eta = 0$



$\eta = 0.7$



$\eta = 0.9$



Descent velocity comparison.

Advantages of GD with momentum:

- 1 Accelerate the correct direction, and decelerate the wrong direction.
- 2 Reduce fluctuation in the “valley area”.

Theorem

For θ_j ,

$$g_{t,j} = \nabla_{\theta_{t,j}} J(\theta)$$

$$G_{t,j} = G_{t-1,j} + g_{t,j}^2$$

$$\theta_{t+1,j} = \theta_{t,j} - \frac{\alpha}{\sqrt{G_{t,j} + \epsilon}} g_{t,j}$$

Illustration:

- Automatically adapt the learning rate to the gradient.

img/comparison of GD vs adam, rmsprop, adagrad, momentum

gradient descent

momentum

adagrad

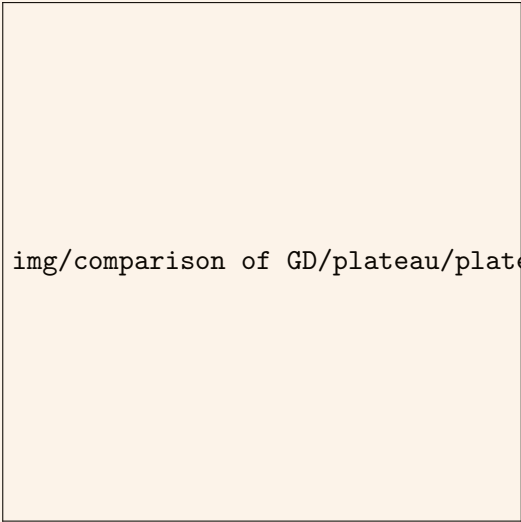


img/comparison of GD/saddle/saddle.png

Gradient Descent in Saddle.

Advantages of AdaGrad:

- 1 Eliminate fluctuation.



img/comparison of GD/plateau/plateau.png

Gradient Descent in Plateau.

Disadvantages of AdaGrad:

- 1 Decrease slow if gradients are high.

Theorem

$$\mathbf{g}_t = \nabla_{\theta} J(\theta) = \begin{bmatrix} \nabla_{\theta_{t,1}} J(\theta) \\ \nabla_{\theta_{t,2}} J(\theta) \\ \vdots \\ \nabla_{\theta_{t,n}} J(\theta) \end{bmatrix}$$

$$G_t = \sum_{\tau=1}^t \mathbf{g}_{\tau} \mathbf{g}_{\tau}^T = \sum_{\tau=1}^t \begin{bmatrix} \nabla_{\theta_{\tau,1}}^2 J(\theta) & * & \cdots & * \\ * & \nabla_{\theta_{\tau,2}}^2 J(\theta) & \cdots & * \\ \vdots & \vdots & \ddots & \vdots \\ * & * & \cdots & \nabla_{\theta_{\tau,n}}^2 J(\theta) \end{bmatrix}$$

$$\theta_{t+1} = \theta_t - \alpha (\text{diag}(G_t) + \epsilon I_n)^{-\frac{1}{2}} \mathbf{g}_t$$

Theorem

Thank you!