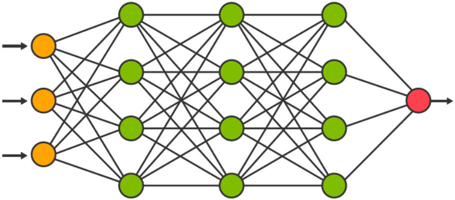
Introduction to Neural Networks

ML Instruction Team, Fall 2022

CE Department
Sharif University of Technology



Early Stopping

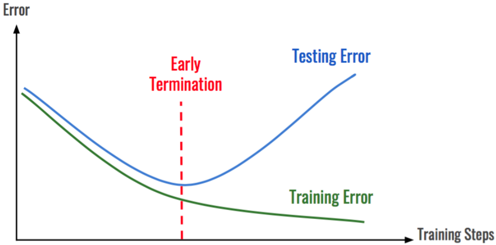


Figure: Early Stopping. Source

- Stop training when accuracy on the validation set decreases (or loss increases). Or keep track of the model parameters that worked best on validation set.

Regularization: Add term to loss

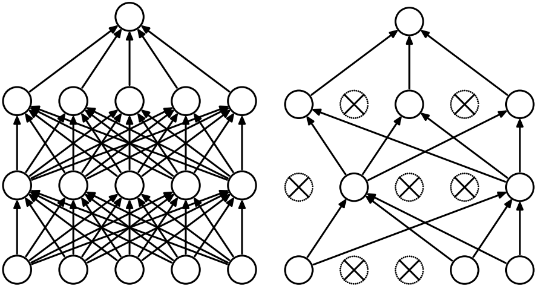
$$L = \frac{1}{N}\sum_{i=1}^{N} L(\phi(x_i), y_i) + \lambda R(W)$$

Common regularization terms:

L2 regularization	$R(W) = \sum_k \sum_l W_{k,l}^2$
L1 regularization	$R(W) = \sum_k \sum_l W_{k,l}^2$
Elastic net (L1 + L2)	$R(W) = \sum_k \sum_l \beta W_{k,l}^2 + \lvert W_{k,l}^2 \rvert$

Regularization: Dropout

- Randomly set some of neurons to zero in forward pass.



(a) Without using dropout

(b) After using dropout.

Figure: Behavior of dropout at training time. Source

Regularization: Dropout

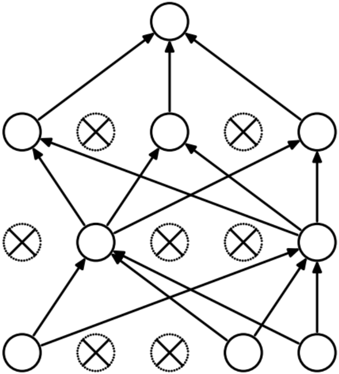


Figure: Source

- Dropout:
  - Prevents co-adaptation of features (forces network to have redundant representations).
  - Can be considered a large ensemble of models sharing parameters.

Dropout: Test Time

- Dropout makes output of network random!

$$y = f_W(x, z)$$

z: random mask
x: input of the layer
y: output of the layer

- We want to ”average out” the randomness at test time:

$$y = f(x) = \mathbb{E}_z[f(x, z)] = \sum_z p(z) f(x, z) dz$$

- Can we calculate the integral exactly?

Dropout: Test Time

- Dropout makes output of network random!

$$y = f_W(x, z)$$

z: random mask
x: input of the layer
y: output of the layer

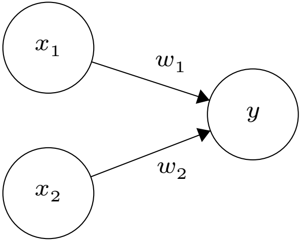
- We want to ”average out” the randomness at test time:

$$y = f(x) = \mathbb{E}_z[f(x, z)] = \sum_z p(z) f(x, z) dz$$

- Can we calculate the integral exactly?
- We need to approximate the integral.

Dropout: Test Time

- Consider a simple case:



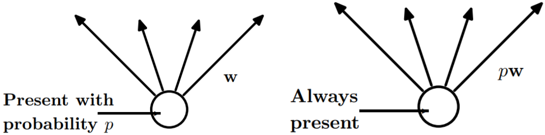
- At training time, each neuron is alive with probability of $p = 0.5$:

$$\begin{aligned}\mathbb{E}[y] =& \frac{1}{4}(w_1x_1 + w_2x_2) + \frac{1}{4}(w_1x_1 + 0) \\ &+ \frac{1}{4}(0 + w_2x_2) + \frac{1}{4}(0 + 0) \\ =& \frac{1}{2}(w_1x_1 + w_2x_2)\end{aligned}$$

- At test time:
  - **Multiply** by dropout rate.

Dropout: Test Time

- At test time neurons are always present and its output is multiplied by dropout probability:



(a) Without using dropout

(b) After using dropout.

Figure: Behavior of dropout at test time. Source

Batch Normalization

- Input: $\boldsymbol{x} : N \times D$
- Output: $\boldsymbol{y} : N \times D$
- Learnable Parameters: $\boldsymbol{\gamma}, \boldsymbol{\beta} : 1 \times D$
- Intermediates: $\boldsymbol{\mu}_B, \boldsymbol{\mu}, \boldsymbol{\sigma}_B^2, \boldsymbol{\sigma}^2 : 1 \times D$
- Intermediates: $\hat{\boldsymbol{x}} : N \times D$
- Hyper-parameters: m (momentum)

Batch Normalization: Training Time

- Input: $\boldsymbol{x} : N \times D$
- Output: $\boldsymbol{y} : N \times D$
- Learnable Parameters: $\boldsymbol{\gamma}, \boldsymbol{\beta} : 1 \times D$
- Intermediates: $\boldsymbol{\mu}_B, \boldsymbol{\mu}, \boldsymbol{\sigma}_B^2, \boldsymbol{\sigma}^2 : 1 \times D$
- Intermediates: $\hat{\boldsymbol{x}} : N \times D$
- Hyper-parameters: m (momentum)

$$\boldsymbol{\mu}_B = \frac{1}{N_B} \sum_{i=1}^{N_B} \boldsymbol{x}^{(i)}$$

$$\boldsymbol{\sigma}_B^2 = \frac{1}{N_B} \sum_{i=1}^{N_B} (\boldsymbol{x}^{(i)} - \boldsymbol{\mu}_B)$$

$$\boldsymbol{\mu} = m\boldsymbol{\mu} + (1 - m)\boldsymbol{\mu}_B \qquad \text{(Running average)}$$

$$\boldsymbol{\sigma}^2 = m\boldsymbol{\sigma}^2 + (1 - m)\boldsymbol{\sigma}_B^2 \qquad \text{(Running average)}$$

$$\hat{\boldsymbol{x}}^{(i)} = \frac{\boldsymbol{x}^{(i)} - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B^2 + \epsilon}}$$

$$\boldsymbol{y}^{(i)} = \boldsymbol{\gamma}\hat{\boldsymbol{x}}^{(i)} + \boldsymbol{\beta}$$

Batch Normalization: Test Time

- Input: $\boldsymbol{x} : N \times D$
- Output: $\boldsymbol{y} : N \times D$
- Learnable Parameters: $\boldsymbol{\gamma}, \boldsymbol{\beta} : 1 \times D$
- Intermediates: $\boldsymbol{\mu}_B, \boldsymbol{\mu}, \boldsymbol{\sigma}_B^2, \boldsymbol{\sigma}^2 : 1 \times D$
- Intermediates: $\hat{\boldsymbol{x}} : N \times D$
- Hyper-parameters: m (momentum)

$$\hat{\boldsymbol{x}}^{(i)} = \frac{\boldsymbol{x}^{(i)} - \boldsymbol{\mu}}{\sqrt{\boldsymbol{\sigma}^2 + \epsilon}}$$

$$\boldsymbol{y}^{(i)} = \boldsymbol{\gamma}\hat{\boldsymbol{x}}^{(i)} + \boldsymbol{\beta}$$

Batch Normalization

- Batch normalization is done along with **C** axis in convolutional networks:

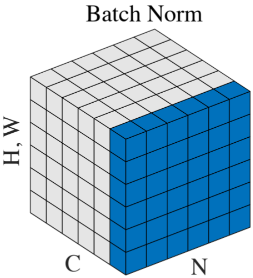


Figure: Batch normalization in CNNs Source.

- BN for FCNs: $\boldsymbol{x}, \boldsymbol{y} : N \times D$ $\rightarrow \boldsymbol{\mu}, \boldsymbol{\sigma}, \boldsymbol{\gamma}, \boldsymbol{\beta} : 1 \times D$
- BN for CNNs: $\boldsymbol{x}, \boldsymbol{y} : N \times C \times H \times W$ $\rightarrow \boldsymbol{\mu}, \boldsymbol{\sigma}, \boldsymbol{\gamma}, \boldsymbol{\beta} : 1 \times C \times 1 \times 1$
- In both cases: $\boldsymbol{y} = \boldsymbol{\gamma}(\boldsymbol{x} - \boldsymbol{\mu}) / \sqrt{\boldsymbol{\sigma}^2 + \epsilon} + \boldsymbol{\beta}$

Gradient Clipping

- In case of a large or small gradient, what will happen?

Gradient Clipping

- In case of a large or small gradient, what will happen?
- Gradient descent either won’t change our position or will send us far away.

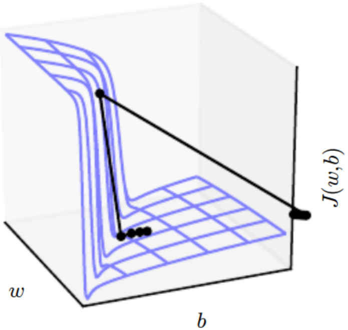


Figure: The problem of large gradient value [1].

Gradient Clipping

- Solve this problem simply by clipping gradient
- Two approaches to do so:
  - Clipping by value
  - Clipping by norm

Gradient Clipping by value

- Set a max (α) and min (β) threshold value
- For each index of gradient $\boldsymbol{g}_i$ if it is lower or greater than your threshold clip it:

```
if $\boldsymbol{g}_i > \alpha$ :
    $\boldsymbol{g}_i \leftarrow \alpha$
else if $\boldsymbol{g}_i < \beta$ :
    $\boldsymbol{g}_i \leftarrow \beta$
```

- Clipping by value will not save gradient direction but still works well in practice.
- To preserve direction use clipping by norm.

Gradient Clipping by norm

- Clip the norm $\|\boldsymbol{g}\|$ of the gradient $\boldsymbol{g}$ before updating parameters:

$$\text{if } \|\boldsymbol{g}\| > v:$$
$$\boldsymbol{g} \leftarrow \frac{\boldsymbol{g}}{\|\boldsymbol{g}\|} v$$

  v is the threshold for clipping which is a hyperparameter.
- Gradient clipping saves the direction of gradient and controls its norm.

Gradient Clipping

- The effect of gradient clipping:

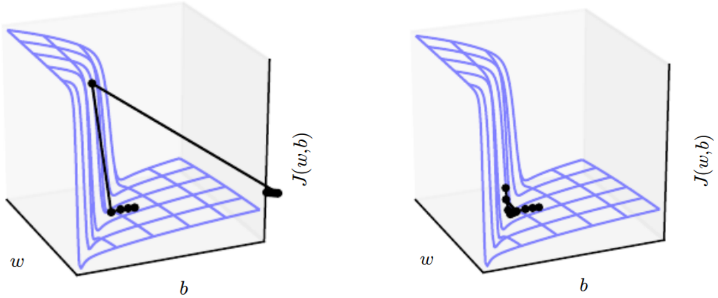


Figure: The ”cliffs” landscape (left) without gradient clipping and (right) with gradient clipping [1].

Weight Initialization

- Is initialization really necessary?
- What are the impacts of initialization?

Weight Initialization

- Is initialization really necessary?
- What are the impacts of initialization?
- A bad initialization may increase convergence time or even make optimization diverge.
- How to initialize?
  - Zero initialization
  - Random initialization

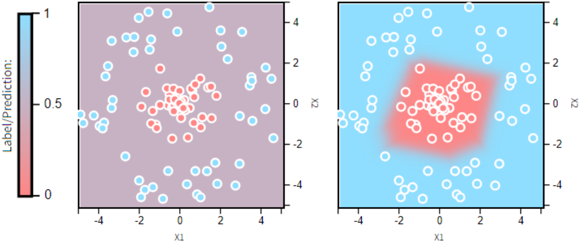
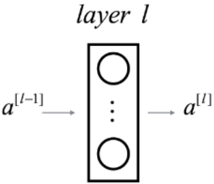


Figure: The output of a three layer network after about 600 epoch. (left) using a bad initialization method and (right) using an appropriate initialization [2].

Weight Initialization

Let's review some notations before we continue:

$$\begin{cases} n^{[l]} := \text{layer } l \text{ neurons number}, \\ W^{[l]} := \text{layer } l \text{ weights}, \\ b^{[l]} := \text{layer } l \text{ biases}, \\ a^{[l]} := \text{layer } l \text{ outputs} \end{cases}$$



Weight Initialization: Zero Initialization

Zero Initialization method:

$$\begin{cases} W^{[l]} = 0, \\ b^{[l]} = 0 \end{cases}$$

- Simple but perform very poorly. (why?)

Weight Initialization: Zero Initialization

Zero Initialization method:

$$\begin{cases} W^{[l]} = 0, \\ b^{[l]} = 0 \end{cases}$$

- Simple but perform very poorly. (why?)
- Zero initialization will lead each neuron to learn the same feature
- This problem is known as network failing to break symmetry
- In fact any constant initialization suffers from this problem.

Weight Initialization: Zero Initialization

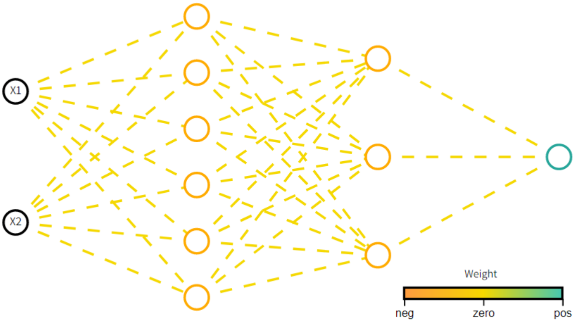


Figure: As we can see network has failed to break symmetry. There has been no improvement in weights after about 600 epochs of training [2].

- We need to break symmetry. How? using randomness.

Weight Initialization: Random Initialization

Simple Random Initialization:

$$\begin{cases} W^{[l]} \sim \mathcal{N}\left(\mu = 0, \sigma^2\right), \\ b^{[l]} = 0 \end{cases}$$

Weight Initialization: Random Initialization

Simple Random Initialization:

$$\begin{cases} W^{[l]} \sim \mathcal{N}\left(\mu = 0, \sigma^2\right), \\ b^{[l]} = 0 \end{cases}$$

- It depends on standard deviation (σ) value
- If it choose carefully, will perform well for small networks
- One can use $\sigma = 0.01$ as a best practice.
- But still has problems with deeper networks.
- Too small/large value for σ will lead to vanishing/exploding gradient problem.

Weight Initialization: Random Initialization

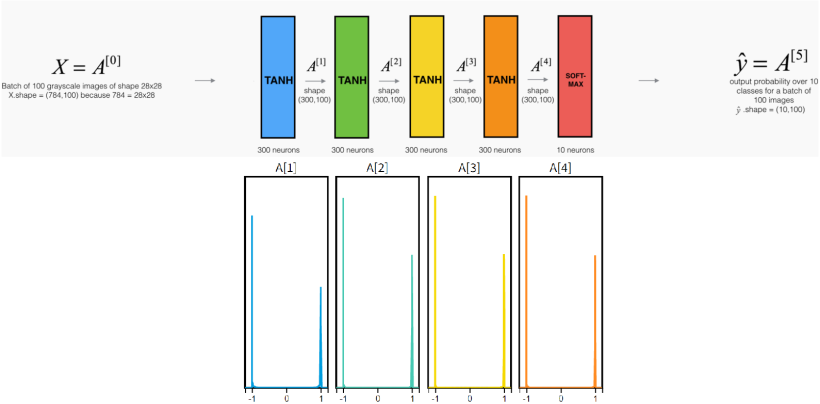

$X = A^{[0]}$
Batch of 100 grayscale images of shape 28x28
X.shape = (784,100) because 784 = 28x28
TANH
TANH
TANH
TANH
SOFT-MAX
$A^{[1]}$
$A^{[2]}$
$A^{[3]}$
$A^{[4]}$
shape (300,100)
300 neurons
10 neurons
$\hat{y} = A^{[5]}$
output probability over 10 classes for a batch of 100 images
$\hat{y}$ .shape = (10,100)
A[1]
A[2]
A[3]
A[4]
-1
0
1

Figure: The problem of normal initialization. On the top, you can see the model architecture, and on the bottom, you can see the density of each layer's output. Model has trained on MNIST dataset for 4 epoch. Weights are initialized randomly from $\mathcal{N}(0, 1)$ [2].

Weight Initialization: Random Initialization

- How to have a better random initialization?
- We need to follow these rules:
    - keep the mean of the activations zero.
    - keep the variance of the activations same across every layer.
- How to do so?

Weight Initialization: Random Initialization

- How to have a better random initialization?
- We need to follow these rules:
  - keep the mean of the activations zero.
  - keep the variance of the activations same across every layer.
- How to do so?

Xavier Random Initialization:

$$\begin{cases} W^{[l]} \sim \mathcal{N}\left(\mu = 0, \sigma^2 = \frac{1}{n^{[l]}}\right), \\ b^{[l]} = 0 \end{cases}$$

(this method works fine for *tanh*, and you can read about why it works at [2].)

Weight Initialization: Random Initialization

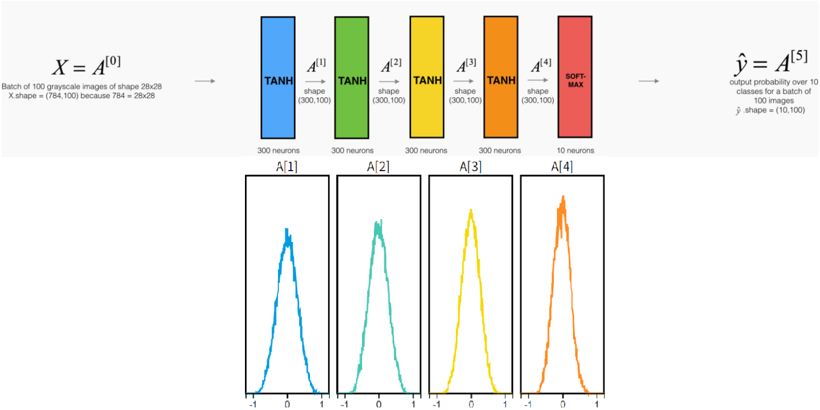


Figure: Vanishing gradient is no longer problem using Xavier initialization. Model has trained on MNIST dataset for 4 epoch. [2].

Weight Initialization

- We discussed weight initialization in previous slides.
- A good initialization will help the model with the vanishing/exploding gradient problem.
- Xavier method works well with *tanh* activation function.
  - If you use $ReLU$ activation use He initialization:

He Initialization:

$$\begin{cases} W^{[l]} \sim \mathcal{N}\left(\mu = 0, \sigma^2 = \frac{2}{n^{[l]}}\right), \\ b^{[l]} = 0 \end{cases}$$

Various GD types

- So far you got familiar with gradient-based optimization
- If $\boldsymbol{g}$ is the gradient of cost w.r.t parameters $\boldsymbol{\theta}$, then we will update parameters with this simple rule:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \boldsymbol{g}$$

- But there is one question here, how to compute $\boldsymbol{g}$?
- Based on how we calculate $\boldsymbol{g}$ we will have different types of gradient descent:
  - Batch Gradient Descent
  - Stochastic Gradient Descent
  - Mini-Batch Gradient Descent

Various GD types

Review before continue:

Training cost function ($\mathcal{J}$) over a dataset usually is the average of loss function ($\mathcal{L}$) on entire training set, so for a dataset $\mathcal{D} = \{d_i\}_{i=1}^{n}$ we have:

$$\mathcal{J}(\mathcal{D}) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(d_i; \boldsymbol{\theta})$$

Various GD types: Batch Gradient Descent

- In this type we use entire training set to calculate gradient

Batch Gradient Descent:

$$\boldsymbol{g} = \frac{1}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} \mathcal{L}(d_i, \boldsymbol{\theta})$$

- This really needs huge computation and so is slow for large training sets.

Various GD types: Batch Gradient Descent

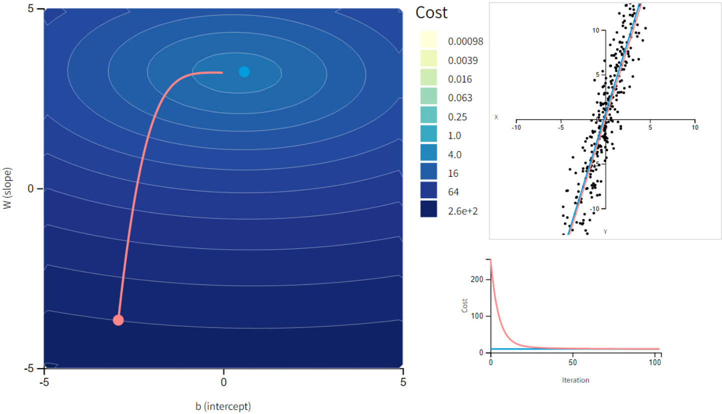


Figure: Optimization of parameters using BGD. Movement is very smooth [3].

Various GD types: Stochastic Gradient Descent

- Instead of calculating exact gradient, we can estimate it using our data
- This is exactly what SGD does, it estimates gradient using only single data point

Stochastic Gradient Descent:

$$\hat{\boldsymbol{g}} = \nabla_{\boldsymbol{\theta}} \mathcal{L}(d_i, \boldsymbol{\theta})$$

- As we use an approximation of gradient, instead of gently decreasing, the cost function will bounce up and down and decrease only on average.
- This method is really computationally efficient cause we only need to calculate gradient for one point per iteration.

Various GD types: Stochastic Gradient Descent

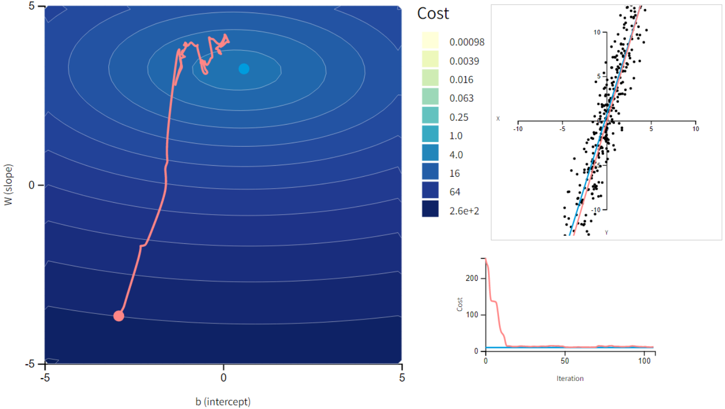


Figure: Optimization of parameters using SGD. As we expect, the movement is not that smooth [3].

Various GD types: Mini-Batch Gradient Descent

- In this method we still use estimation idea But use a batch of data instead of one point.

Mini-Batch Gradient Descent:

$$\hat{\boldsymbol{g}} = \frac{1}{|\mathcal{B}|} \sum_{d \in \mathcal{B}} \nabla_{\boldsymbol{\theta}} \mathcal{L}(d, \boldsymbol{\theta}), \quad \mathcal{B} \subset \mathcal{D}$$

- A better estimation than SGD
- With this way we can get a performance boost from hardware optimization, especially when using GPUs.
- Batch size ($|\mathcal{B}|$) is a hyperparameter you need to tune.

Various GD types: Mini-Batch Gradient Descent

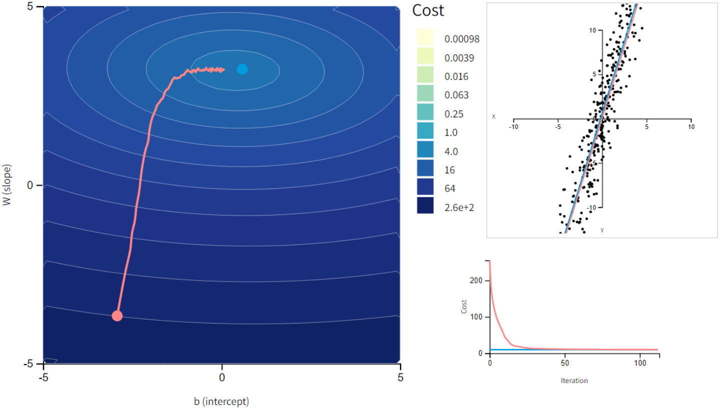


Figure: Optimization of parameters using MBGD. The movement is much smother than SGD and behave like BGD [3].

Various GD types

- So we got familiar with different types of GD.
  - ✓ Batch Gradient Descent (BGD)
  - ✓ Stochastic Gradient Descent (SGD)
  - ✓ Mini-Batch Gradient Descent (MBGD)

- The most recommended one is MBGD, because it is computational efficient.
- Choosing the right batch size is important to ensure convergence of the cost function and parameter values, and to the generalization of your model.

Thank You!

Any Question?

References

I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*.
MIT Press, 2016.
http://www.deeplearningbook.org.

K. Katanforoosh and D. Kunin, “Initializing neural networks,” 2019.
https://www.deeplearning.ai/ai-notes/initialization/.

K. Katanforoosh and D. Kunin, “Parameter optimization in neural networks,” 2019.
https://www.deeplearning.ai/ai-notes/optimization/.