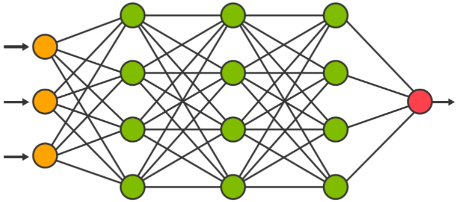
Introduction to Neural Networks

ML Instruction Team, Fall 2022

CE Department
Sharif University of Technology



Problem: Over-fitting in a Neural Network

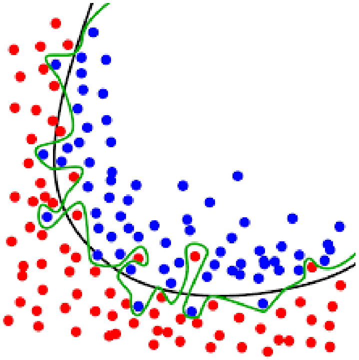


Figure: Over-fitting in a Neural Network source

Solution 1: L1/L2 regularization

- just like a classic regularizer
- sum the regularizer term for every layer weight

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

Solution 1: L1/L2 regularization

- L1/L2 regularizer functions (review)

$$L1 : R(w) = |w|$$
$$L2 : R(w) = w^2$$

- you can also combine the two different regularizers (Elastic net)

$$R(w) = \beta w^2 + |w|$$

Solution 2: Early Stopping

- Stop learning when the validation error is Minimum

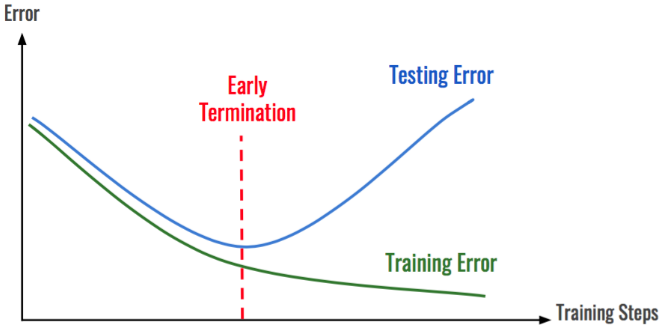


Figure: Early-Stopping diagramsource

Solution 3: Dropout

during training:

- at each iteration, every neuron has a probability P of being dropped out
  - meaning it will be ignored
- P := dropout rate

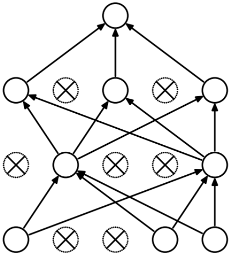


Figure: Behavior of dropout at training time. Source

Solution 3: Dropout

during testing:

- at each iteration, on average, 1-P percent of Neurons are active
  - less neurons results in bigger weights
- we need to normalize weights by 1-P

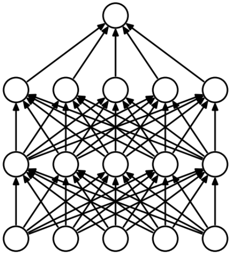


Figure: Behavior of dropout at testing time. Source

Solution 3: Dropout

practice tips:

- still overfitting -> increase dropout rate
- underfitting -> decrease dropout rate
- layersize -> dropout rate
- usually in practice, people will use dropout after the last hidden layer

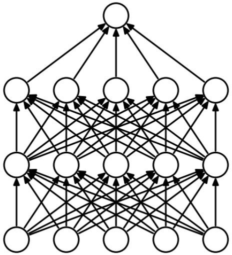


Figure: Behavior of dropout at testing time. Source

Problem: Vanishing/Exploding Gradients

Solution: Batch Norm layer

Regularization: Dropout

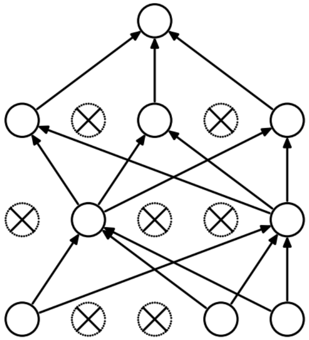


Figure: Source

- 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:

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

- 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

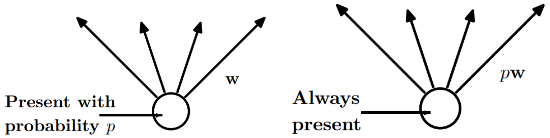
- We want to ”average out” the randomness at test time:

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

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

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

Regularization: Adding Noise

- We've seen a common approach for regularization thus far:
  - ▷ **Training**: Add some kind of randomness (z) :

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

  - ▷ **Testing**: Average out the randomness:

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

Regularization: Adding Noise

- We've seen a common approach for regularization thus far:
  - ▷ **Training**: Add some kind of randomness (z) :

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

  - ▷ **Testing**: Average out the randomness:

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

- Adding noise is another way to prevent a neural network from overfitting on the training data. In every iteration, a random noise is added to the outputs of the layer, preventing consequent layers from co-adapting too much to the outputs of this layer.

Batch Normalization

Input: $x : N \times D$ Learnable Parameters: $\gamma, \beta : D$

Output: $y : N \times D$ Intermediates: $\mu, \sigma : D$, $\hat{x} : N \times D$

$$
\begin{aligned}
\mu_j &= \text{(Running) average of values seen during training} \\
\sigma_j^2 &= \text{(Running) average of values seen during training} \\
\hat{x}_{i,j} &= \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}} \\
y_{i,j} &= \gamma_j \hat{x}_{i,j} + \beta_j
\end{aligned}
$$

Batch Normalization

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

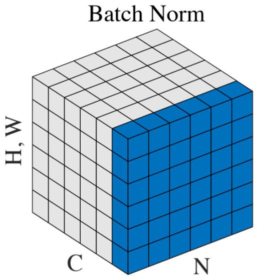


Figure: Batch normalization in CNNs Source.

- ▷ BN for FCNs: $x, y : N \times D \rightarrow \mu, \sigma, \gamma, \beta : 1 \times D$
- ▷ BN for CNNs: $x, y : N \times C \times H \times W \rightarrow \mu, \sigma, \gamma, \beta : 1 \times C \times 1 \times 1$
- ▷ In both cases: $y = \gamma(x - \mu)/\sigma + \beta$

Gradient Clipping

- What will happen in case of a large gradient value?
- The gradient descent will take us far away from our local position.

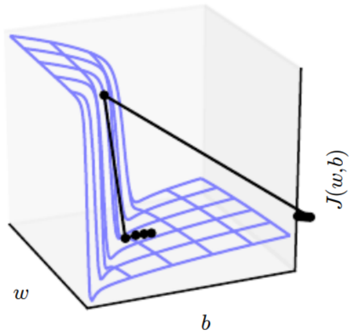


Figure: The problem of large gradient value [1].

Gradient Clipping

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

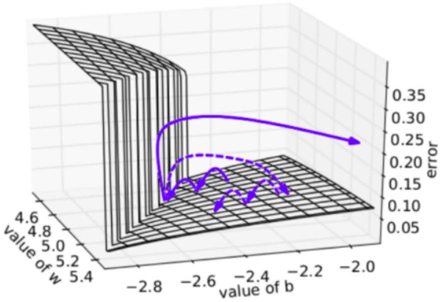
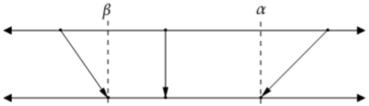


Figure: The effect of gradient clipping. Instead of solid line following dotted line will lead us to minimum, Source

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 $\boldsymbol{g}_i < \beta$ :
    $\boldsymbol{g}_i \leftarrow \beta$
```



Gradient Clipping by value

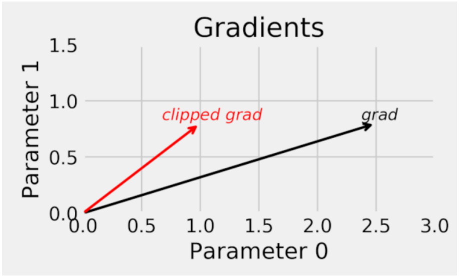


Figure: The effect of clipping by value. Source

- Clipping by value will change gradient direction.
- To preserve direction use clipping by norm.

Gradient Clipping by norm

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

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

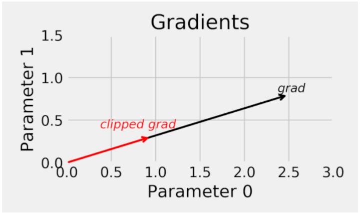


Figure: The effect of clipping by norm. source

v is the threshold for clipping which is a hyperparameter.

- Gradient clipping saves the direction of gradient and controls its norm.

Gradient Clipping

- Clipping by value will change the direction of the gradient, so it will send us to a bad neighborhood.
- Clipping by norm will preserve the direction and just control the value.
- So it is better to use clipping by norm.

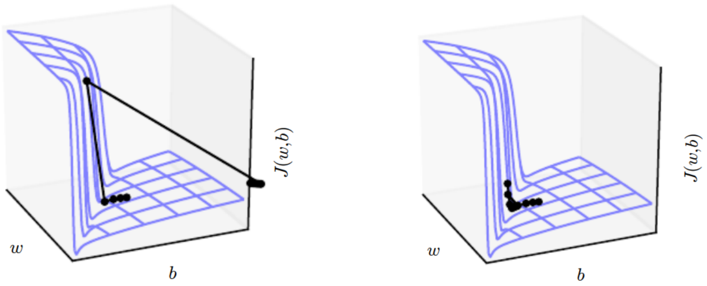


Figure: The ”cliffs” landscape (left) without gradient clipping and (right) with gradient clipping [1].

Weight Initialization

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

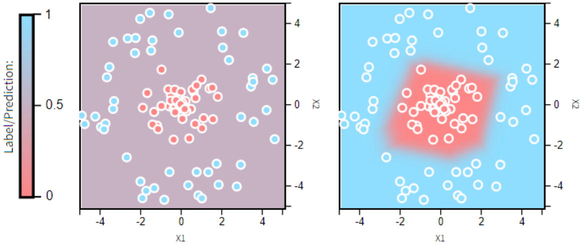


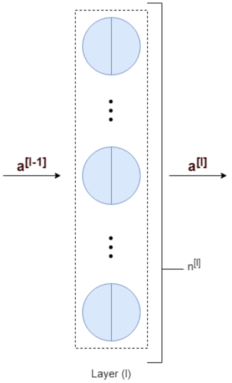
Figure: The output of a three layer network after about 600 epoch. (left) using a bad initialization method and (right) using an appropriate initialization [2].

Weight Initialization

Let's review some notations before we continue:

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

$$\begin{cases} \text{fan}_{\text{in}}^{[l]} = n^{[l-1]} \quad (\text{layer } l \text{ number of inputs}), \\ \text{fan}_{\text{out}}^{[l]} = n^{[l]} \quad (\text{layer } l \text{ number of outputs}), \\ \text{fan}_{\text{avg}}^{[l]} = \frac{n^{[l-1]}+n^{[l]}}{2} \end{cases}$$



Weight Initialization: Zero Initialization

Zero Initialization method:

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

- 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

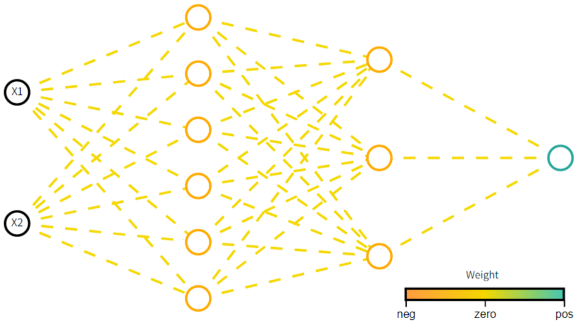


Figure: As we can see network has failed to break symmetry. There has been no improvement in weights after about 600 epochs of training [2].

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

Weight Initialization: Random Initialization

- To use randomness in our initialization we can use uniform or normal distribution:

General Uniform Initialization:

$$\begin{cases} W^{[l]} \sim U(-r, +r), \\ b^{[l]} = 0 \end{cases}$$

General Normal Initialization:

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

- But this is really crucial to choose r or σ properly.

Weight Initialization: Random Initialization

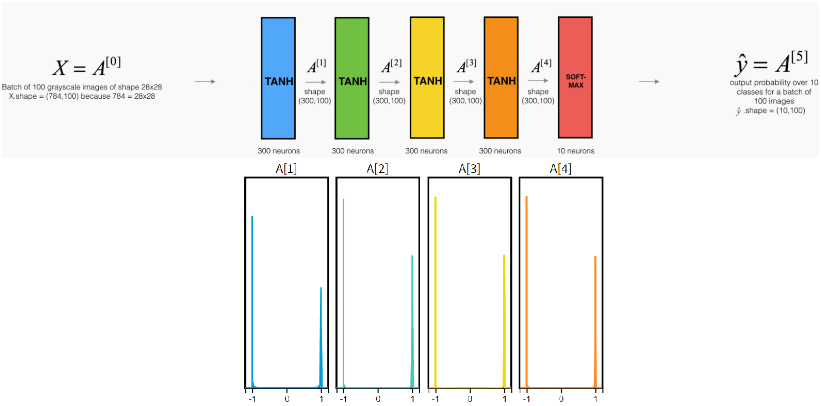


Figure: Uniform initialization problem. On the top, you can see the model architecture, and on the bottom, you can see the density of each layer's output. Model has trained on MNIST dataset for 4 epoch. Weights are initialized randomly from $U(\frac{-1}{\sqrt{n^{[l-1]}}}, \frac{1}{\sqrt{n^{[l-1]}}})$ [2].

Weight Initialization: Random Initialization

- How to choose r or σ?
- We need to follow these rules:
  - keep the mean of the activations zero.
  - keep the variance of the activations same across every layer.

Xavier Initialization:

- For Uniform distribution use:

$$r = \sqrt{\frac{3}{\text{fan}_{\text{avg}}}}$$

- For Normal distribution use:

$$\sigma^2 = \frac{1}{\text{fan}_{\text{avg}}}$$

(You can read about why this method works at [2].)

Weight Initialization: Xavier Initialization

- Xavier initialization works well on Tanh, Logestic or Sigmoid activation function.

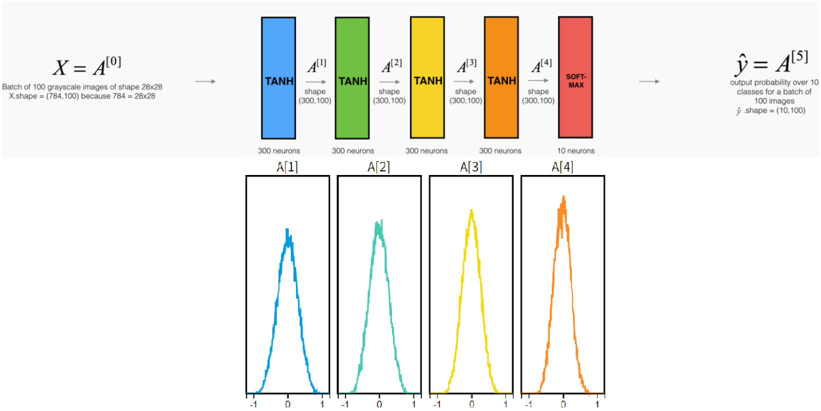


Figure: Vanishing gradient is no longer problem using Xavier initialization. Model has trained on MNIST dataset for 4 epoch. [2].

Weight Initialization: He Initialization

- Different method has proposed for different activation functions.

He Initialization:

- For Normal distribution:

$$\sigma^2 = \frac{2}{n^{[l]}}$$

- For Uniform distribution:

$$r = \sqrt{3\sigma^2}$$

- He initialization works well on ReLU and its variants.

Various GD types

- So far you got familiar with gradient-based optimization.
- If $\boldsymbol{g} = \nabla_{\boldsymbol{\theta}} \mathcal{J}$, then we will update parameters with this simple rule:

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

- 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

Recap:

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:

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

For example:

$$H(p,q) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{k} y_j^{(i)}\log(p(y_j^{(i)}))$$

$$\text{MSE}(y,\hat{y}) = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$$

$$\text{MAE}(y,\hat{y}) = \frac{1}{m}\sum_{i=1}^{m}|(y_i - \hat{y}_i)|$$

Various GD types: Batch Gradient Descent

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

Batch Gradient:

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

- Using this method with very large training set:
  - ▷ Your data can be too large to process in your memory.
  - ▷ It requires a lot of processing to compute gradient for all samples.
- Using exact gradient may lead us to local minima.
- Moving noisy may help us get out of this local minimas.

Various GD types: Batch Gradient Descent

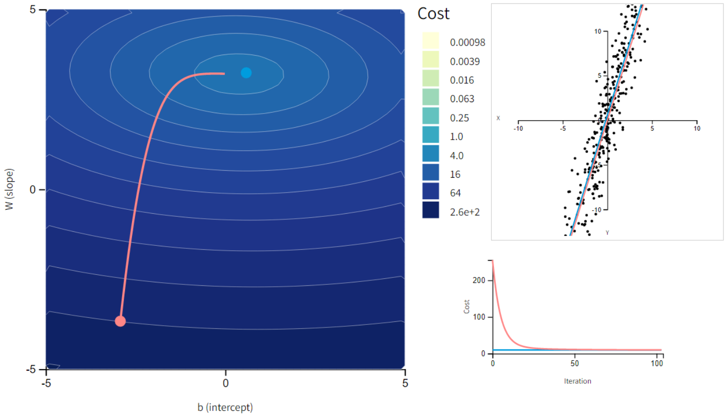


Figure: Optimization of parameters using BGD. Movement is very smooth [3].

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:

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

- 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

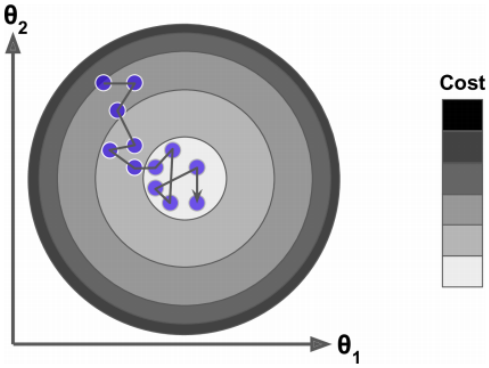


Figure: Optimization of parameters using SGD. As we expect, the movement is not that smooth [3].

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:

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

- This is 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

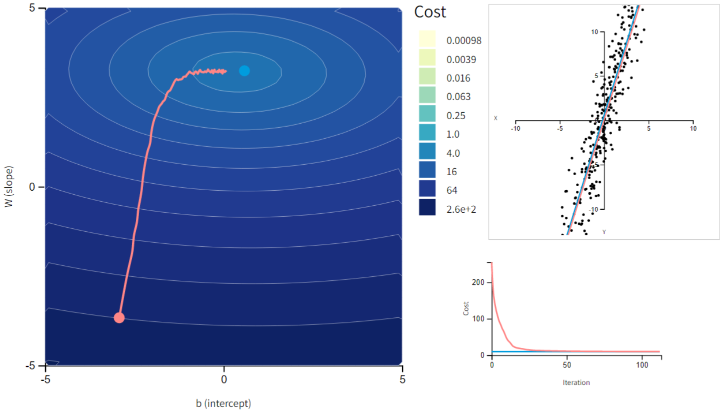


Figure: Optimization of parameters using MBGD. The movement is much smother than SGD and behave like BGD [3].

Various GD types

- Now that we know what a batch is, we can define epoch and iteration:
  - ▷ One Epoch is when an entire dataset is passed forward and backward through the network only once.
  - ▷ One Iteration is when a batch is passed forward and backward through the network.

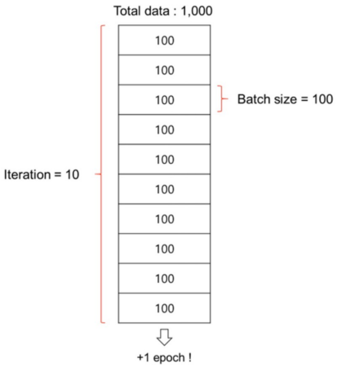


Figure: Epoch vs Iteration, source.

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/.