# Ch04. Optimization for Deep Learning

Hwanjo Yu

POSTECH

http://hwanjoyu.org

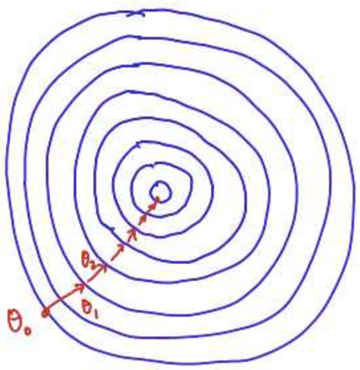
# Outline

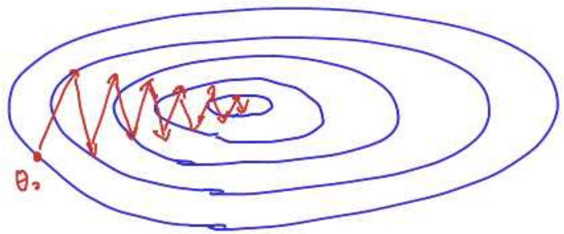
- Gradient descent
- Exponentially weighted average and bias correction
- Gradient descent with momentum
- AdaGrad
- RMSprop
- ADAM (the most popular one in practice)
- AMSGrad

# Gradient Descent

- Consider the objective function: $\mathcal{J}(\theta) = \mathbb{R}^D \rightarrow \mathbb{R}$.
- For instance, $\mathcal{J}(\theta) = \frac{1}{N}\sum_{n=1}^{N}\|f(\mathbf{x}_n;\theta) - y_n\|^2$ .
- Moves from the current values of parameters, $\theta_k$, in the opposite direction of the gradient of the objective function $\mathcal{J}(\theta)$ w.r.t. the parameters, evaluated at $\theta_k$:

$$\theta_{k+1} \leftarrow \theta_k - \alpha \left[\frac{\partial \mathcal{J}(\theta)}{\partial \theta}\right]_{\theta=\theta_k} .$$



# Batch Gradient Descent

- Training set $= \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_N, y_N)\} = (\mathbf{X}, \mathbf{y})$, where $\mathbf{x}_n \in \mathbb{R}^D$ and $y_n \in \mathbb{R}$.
- **Batch** $= (\mathbf{X}, \mathbf{y})$.
- Resort to entire training dataset to compute the gradient of the objective function.

$$\theta_{k+1} \leftarrow \theta_k - \alpha \left[ \frac{\partial \mathcal{J}(\theta; \mathbf{X}, \mathbf{y})}{\partial \theta} \right]_{\theta = \theta_k}.$$

- The accuracy of the parameter update is high but it can be slow.
- Intractable for datasets that do not fit in memory.
- Does not allow us to update the model *online* (with new examples on the fly).

# Mini-Batch Gradient Descent (SGD)

- **Mini-batch** of size $M = \left(\mathbf{X}^{\{m\}}, \mathbf{y}^{\{m\}}\right)$:

$$\mathbf{X}^{\{m\}} = \{\mathbf{x}_m, \dots, \mathbf{x}_{m+M-1}\},$$
$$\mathbf{y}^{\{m\}} = \{y_m, \dots, y_{m+M-1}\}.$$

- Consider the objective function that is the sum of errors evaluated on each mini-batch:

$$\mathcal{J}(\theta; \mathbf{X}, \mathbf{y}) = \frac{1}{N_M} \sum_{m=1}^{N_M} \mathcal{J}\left(\theta; \mathbf{X}^{\{m\}}, \mathbf{y}^{\{m\}}\right),$$

- where $\mathbf{X}^{\{m\}}$ and $\mathbf{y}^{\{m\}}$ are training examples in mini-batch $m$ and $N_M$ is the number of mini-batches.

- Mini-batch gradient descent updates parameters:

$$\theta_{k+1} \leftarrow \theta_k - \alpha \left[ \frac{\partial \mathcal{J}\left(\theta; \mathbf{X}^{\{k\}}, \mathbf{y}^{\{k\}}\right)}{\partial \theta} \right]_{\theta=\theta_k}.$$

- Or using a batch of size 1,

$$\theta_{t+1} \leftarrow \theta_t - \alpha \left[ \frac{\partial \mathcal{J}(\theta; \mathbf{x}_t, y_t)}{\partial \theta} \right]_{\theta=\theta_t}.$$

# Stochastic Gradient Descent (SGD)

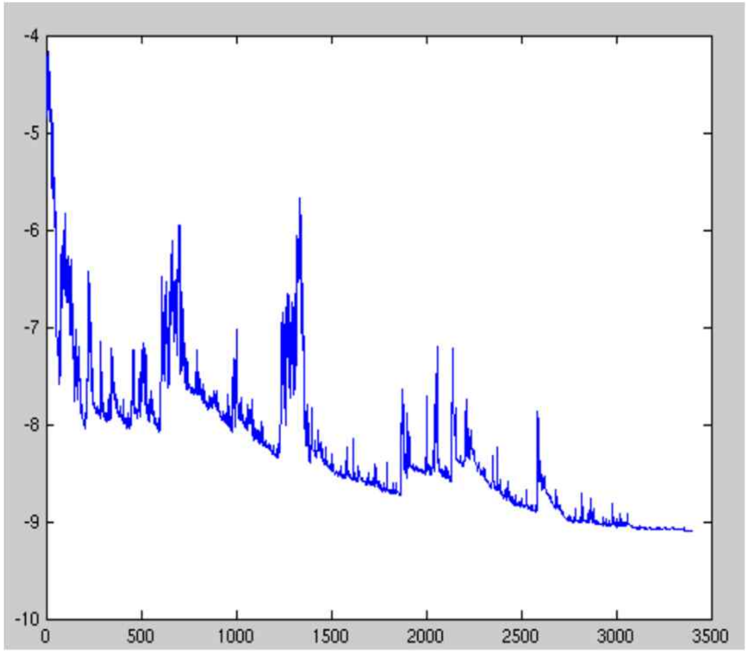



Figure: Fluctuation in the objective values as gradient steps for SGD.

[Figure source: Wikipedia]

# Challenges

Vanilla gradient descent provides a few challenges to be addressed:

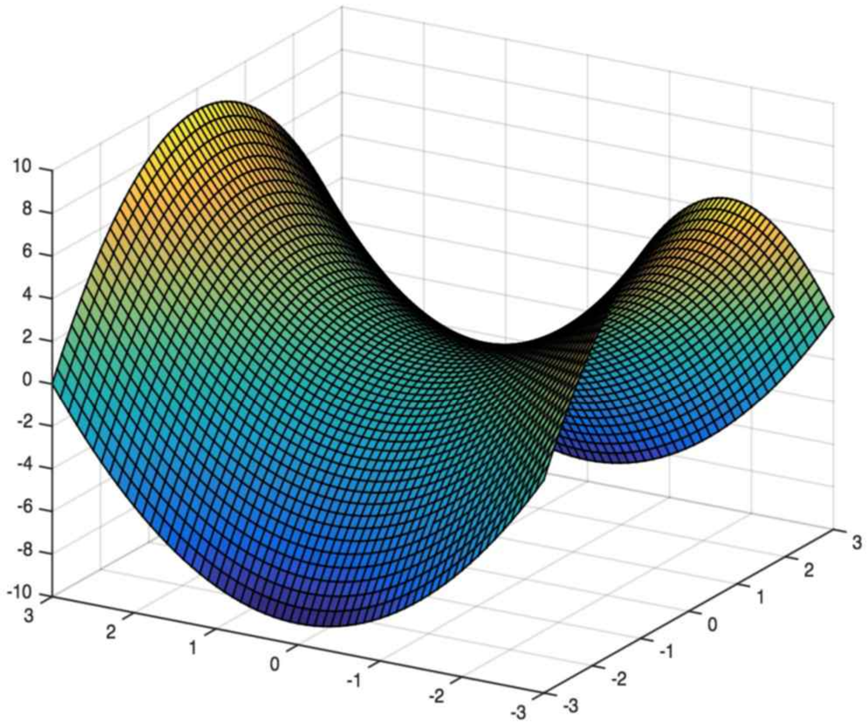
- Choosing a proper step size (learning rate) is difficult.
  - Small step size yields slow convergence.
  - Large step size hinders convergence, causing fluctuation in the objective function.
- Learning rate decay schedule has to be defined in advance and is unable to adapt a dataset's characteristics.
- Avoiding getting trapped in local minima or saddle points[1] is a key challenge for the minimization of highly non-convex objective functions (common for deep neural networks).

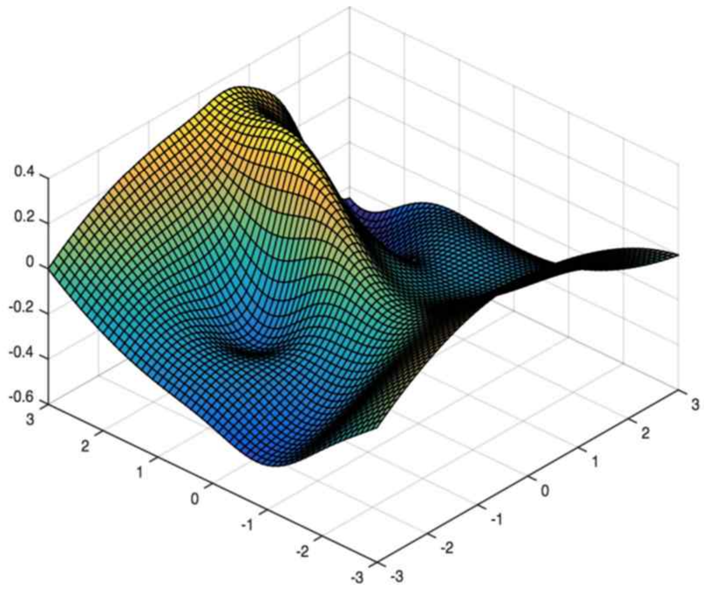
Need techniques to:

- accelerate the vanilla gradient descent;
- help it out of local minima.

[1]Yann N., Dauphin et al. (2014), "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization," *Preprint arXiv:1406.2572.*

# Saddle Points



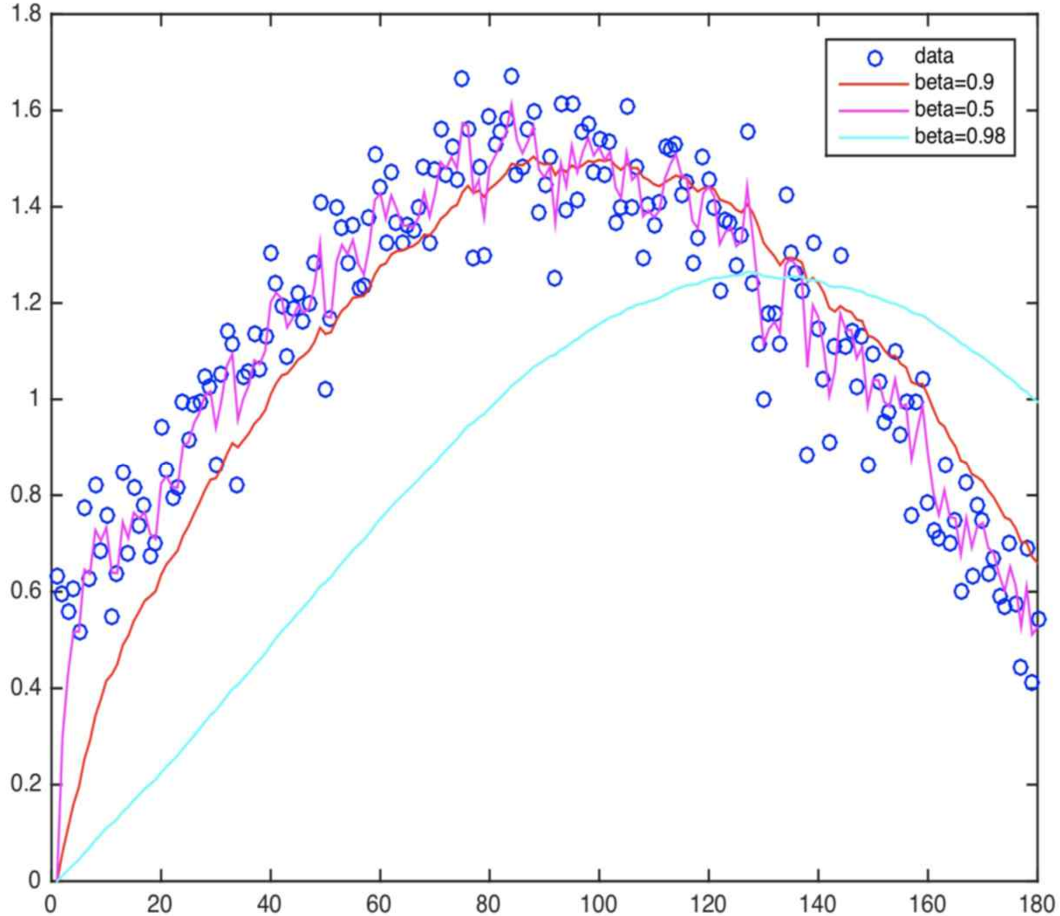
# Exponentially Weighted Moving Average

- Suppose that we are given $\theta_1, \theta_2, \theta_3, \dots,$
- Moving average of $\theta_t$ is calculated as

$$\mathbf{v}_t = \beta \mathbf{v}_{t-1} + (1-\beta)\theta_t, \qquad 1 > \beta > 0$$

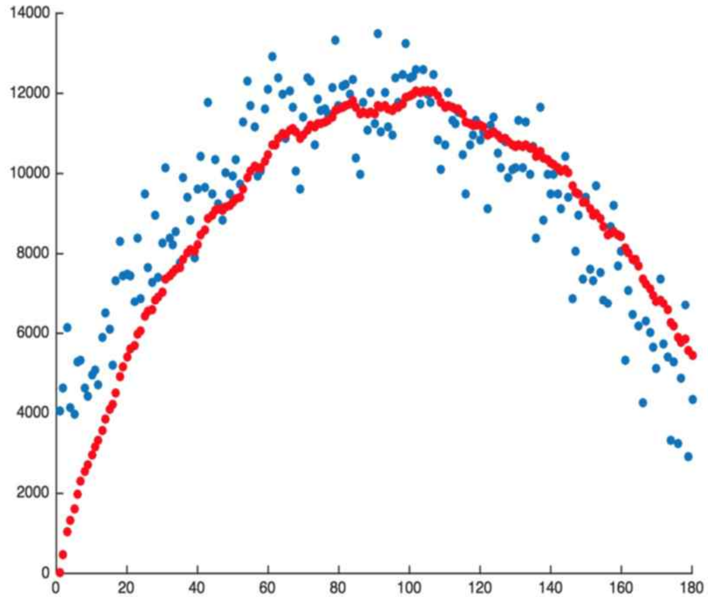
- Start with $\mathbf{v}_0 = 0$.
  - $\mathbf{v}_1 = \beta \mathbf{v}_0 + (1-\beta)\theta_1 = (1-\beta)\theta_1,$
  - $\mathbf{v}_2 = \beta \mathbf{v}_1 + (1-\beta)\theta_2 = \beta(1-\beta)\theta_1 + (1-\beta)\theta_2 = (1-\beta)(\beta\theta_1 + \theta_2),$
  - $\mathbf{v}_3 = \beta \mathbf{v}_2 + (1-\beta)\theta_3 = \beta(1-\beta)(\beta\theta_1 + \theta_2) + (1-\beta)\theta_3 = (1-\beta)(\beta^2\theta_1 + \beta\theta_2 + \theta_3).$
- $\mathbf{v}_t = (1-\beta)(\beta^{t-1}\theta_1 + \beta^{t-2}\theta_2 + \cdots + \theta_t).$
- which approximately average over $\frac{1}{1-\beta}$ samples.

# Exponentially Weighted Moving Average

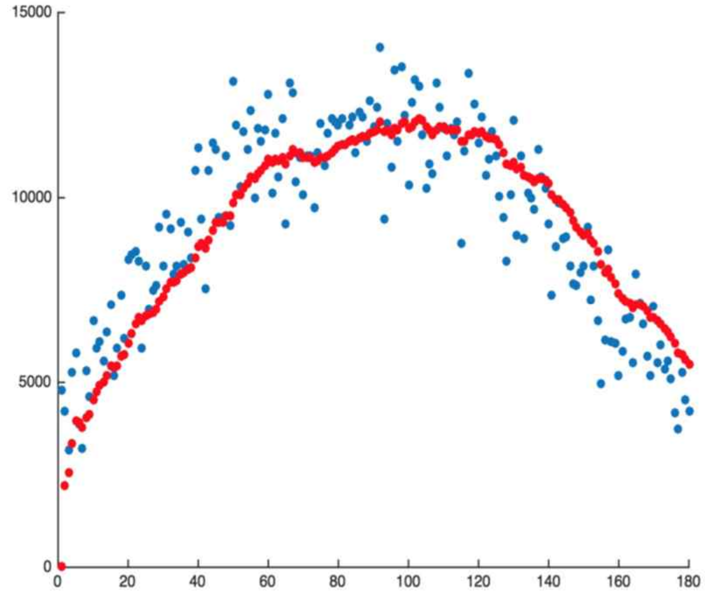


# Bias Correction

- Use $\frac{\mathbf{v}_t}{1-\beta^t}$ instead of $\mathbf{v}_t$ after computing $\mathbf{v}_t = \beta\mathbf{v}_{t-1} + (1-\beta)\theta_t$. (useful during initial phase)




(a) moving average




(b) with bias correction

# Gradient Descent with Momentum

Ning Qian (1999), "On the momentum term in gradient descent learning algorithms,“ Neural Networks.

- Recall gradient descent:

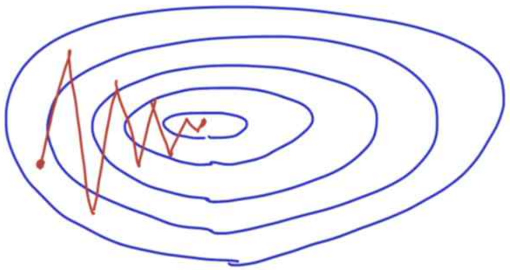
$$\theta_{t+1} = \theta_t - \alpha[\nabla\mathcal{J}(\theta_t)],$$

  - where $\alpha$ is the step size.

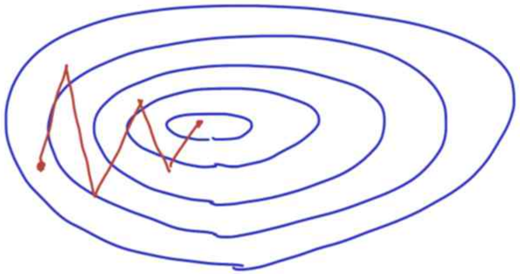
- <u>Gradient descent with momentum</u> uses moving averages of gradients to update parameters.

$$\mathbf{v}_{t+1} = \beta\mathbf{v}_t + (1-\beta)[\nabla\mathcal{J}(\theta_t)],$$
$$\theta_{t+1} = \theta_t - \alpha\mathbf{v}_{t+1}.$$

  - When gradients keep pointing in the same direction, this will increase the size of the steps taken towards the minimum.
  - When the gradient keeps changing direction, momentum will smooth out the variations.



(a) SGD without momentum



(a) SGD with momentum

# AdaGrad: Adaptive Gradient

John Duchi, Elad Hazan, and Yoram Singer (2011), "Adaptive subgradient methods for online learning and stochastic optimization," JMLR.

- A different step size for every parameter $\theta_i$ at every time step $t$.

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \frac{\alpha}{\sqrt{G_{i,i}^{(t)} + \epsilon}} \nabla\mathcal{J}\left(\theta_i^{(t)}\right),$$

- where $G_{i,i}^{(t)} = \sum_{j=1}^{t} \left(\nabla\mathcal{J}\left(\theta_i^{(j)}\right)\right)^2$ contains the sum of squares of the gradients $\theta_i$ w.r.t. up to time step $t$.
- $G_{i,i}^{(t)}$ represents the diagonal entries of the matrix $\mathbf{G}^{(t)}$ which is calculated as

$$\mathbf{G}^{(t)} = \mathrm{diag}\left(\sum_{j=1}^{t} \left[\nabla\mathcal{J}\left(\theta^{(j)}\right)\right]\left[\nabla\mathcal{J}\left(\theta^{(j)}\right)\right]^{\mathrm{T}}\right).$$

- In practice, we update each parameter $\theta_i$:

$$r_i^{(t)} = r_i^{(t-1)} + \left(\nabla\mathcal{J}\left(\theta_i^{(t)}\right)\right)^2,$$
$$\theta_i^{(t+1)} = \theta_i^{(t)} - \frac{\alpha}{\sqrt{r_i^{(t)} + \epsilon}} \nabla\mathcal{J}\left(\theta_i^{(t)}\right).$$

- Steps get smaller and smaller over the course of training.
- Convex case: Makes sense.
- Non-convex case: Can get stuck on saddle points.

# RMSProp

T. Tieleman and G. Hinton (2012), "Lecture 6.5 - RMSProp, COURSERA: Neural Networks for Machine Learning".

Mathew D. Zeiler (2012), "ADADELTA: An adaptive learning rate method," Preprint arXiv:1212.5701.

- RMSProp = Rprop + SGD
- Adaptive individual learning rate for each weight.
- Instead of accumulating all past squared gradients, the moving average is used to scale the step size.
- Update parameters $\theta_t$ by

$$\mathbf{r}_{t+1} = \beta \mathbf{r}_t + (1-\beta)[\nabla \mathcal{J}(\theta_t)]^2 \quad \text{(element-wise square)},$$

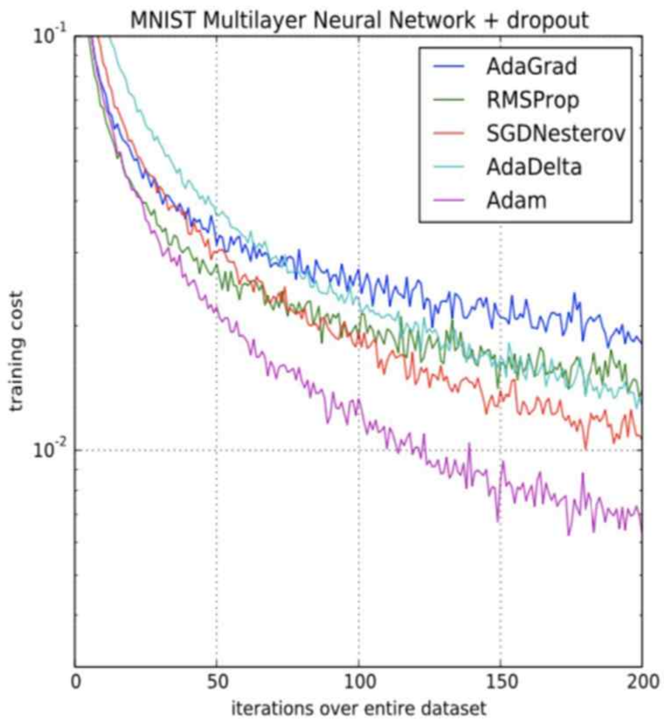
$$\theta_{t+1} = \theta_t - \alpha \frac{\nabla \mathcal{J}(\theta_t)}{\sqrt{\mathbf{r}_{t+1} + \epsilon}} \quad \text{(element-wise division)}.$$

- Intension: Larger oscillation produces larger $[\nabla \mathcal{J}(\theta_t)]^2$ thus smaller update.
- $\epsilon$ is to prevent the division-by-zero (e.g., $10^{-8}$).

# ADAM

Diederik P. Kingma and Jimmy Lei Ba (2015), "ADAM: A method for stochastic optimization," ICLR.

- Uses estimations of first and second moments of gradient to adapt the learning rate for each weight of the neural network.
- Adaptive individual learning rate for each weight.
- ADAM = momentum + RMSProp + bias correction.
  - $\mathbf{v}_t = \beta_1 \mathbf{v}_{t-1} + (1-\beta_1)[\nabla \mathcal{J}(\theta_{t-1})]$,
  - $\mathbf{r}_t = \beta_2 \mathbf{r}_{t-1} + (1-\beta_2)[\nabla \mathcal{J}(\theta_{t-1})]^2$ (element-wise square),
  - $\mathbf{v}_t^{bc} = \frac{\mathbf{v}_t}{1-\beta_1^t}$,
  - $\mathbf{r}_t^{bc} = \frac{\mathbf{r}_t}{1-\beta_2^t}$,
  - $\theta_t = \theta_{t-1} - \alpha \frac{\mathbf{v}_t^{bc}}{\sqrt{\mathbf{r}_t^{bc} + \epsilon}}$ (element-wise division).



# Learning Rate Decay

- Slowly reduce the step size $\alpha$.
- One epoch = one pass through whole training examples.
- Strategies ($\eta =$ decay rate; $\Omega =$ epoch number; $t =$ batch number)

$$\alpha = \frac{1}{1 + \eta\Omega}\alpha_0,$$
$$\alpha = 0.95^{\Omega}\alpha_0,$$
$$\alpha = \frac{k}{\sqrt{\Omega}}\alpha_0,$$
$$\alpha = \frac{k}{\sqrt{t}}\alpha_0,$$

- Or, $\alpha$ is manually set such that the value of constant is decreasing in a stepwise fashion.