Week 2: Optimization and Frameworks

Instructor: Ruixuan Wang
wangruix5@mail.sysu.edu.cn

School of Data and Computer Science
Sun Yat-Sen University

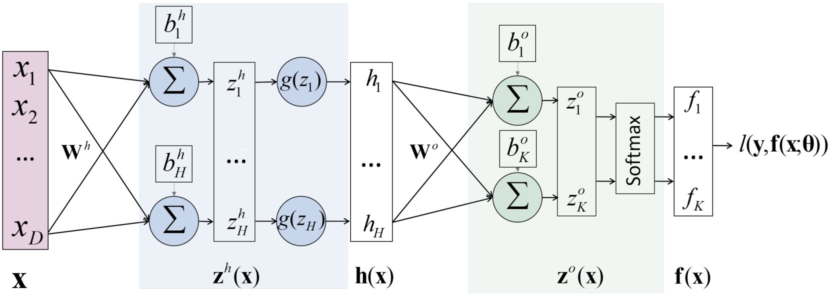
7 March, 2019

Derivatives: basic

- $L(\theta) = a$
- $L(\theta) = a\theta$
- $L(\theta) = au(\theta)$
- $L(\theta) = u(\theta) + v(\theta) - w(\theta)$
- $L(\theta) = \frac{u(\theta)}{v(\theta)}$
- $L(\theta) = u^n(\theta)$
- $L(\theta) = \frac{1}{u(\theta)}$
- $L(\theta) = \ln u(\theta)$
- $L(\theta) = e^{u(\theta)}$
- $L(\theta) = f(u(\theta))$

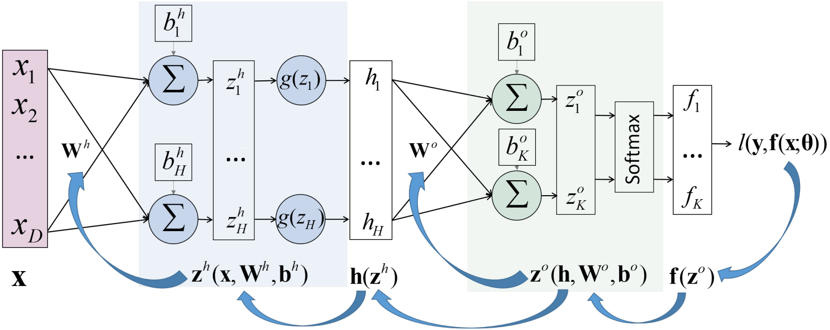
- $\frac{dL}{d\theta} = 0$
- $\frac{dL}{d\theta} = a$
- $\frac{dL}{d\theta} = a\frac{du}{d\theta}$
- $\frac{dL}{d\theta} = \frac{du}{d\theta} + \frac{dv}{d\theta} - \frac{dw}{d\theta}$
- $\frac{dL}{d\theta} = \frac{1}{v}\frac{du}{d\theta} - \frac{u}{v^2}\frac{dv}{d\theta}$
- $\frac{dL}{d\theta} = nu^{n-1}\frac{du}{d\theta}$
- $\frac{dL}{d\theta} = -\frac{1}{u^2}\frac{du}{d\theta}$
- $\frac{dL}{d\theta} = \frac{1}{u(\theta)}\frac{du}{d\theta}$
- $\frac{dL}{d\theta} = e^{u(\theta)}\frac{du}{d\theta}$
- $\frac{dL}{d\theta} = \frac{df}{du}\frac{du}{d\theta}$

Training a two-layer network classifier



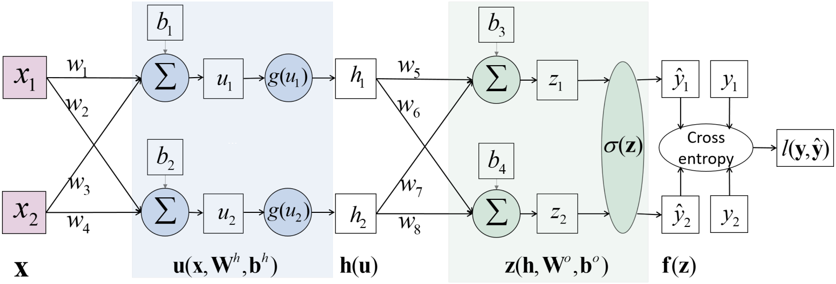
- Loss function $l(\mathbf{y}, \mathbf{f}(\mathbf{x}; \boldsymbol{\theta}))$ measures difference between prediction $\mathbf{f}(\mathbf{x}; \boldsymbol{\theta})$ and ideal output $\mathbf{y}$.
- Model parameters $\boldsymbol{\theta} = \{\mathbf{W}^h, \mathbf{b}^h, \mathbf{W}^o, \mathbf{b}^o\}$
- Training network: minimize loss with gradient descent to find best $\boldsymbol{\theta}^*$

Backpropagation: computation of gradient with chain rule



- Gradient of loss function over model parameters can be computed with chain rule.

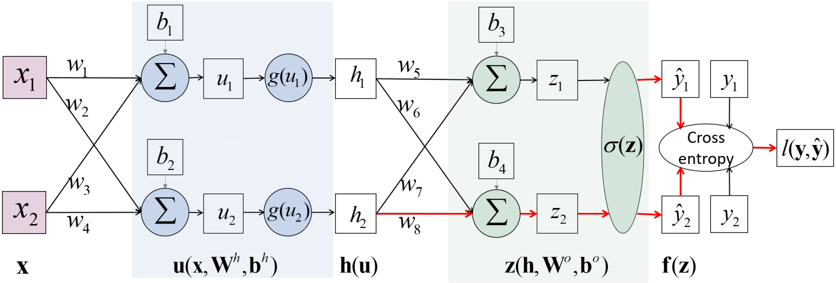
A simpler 2-layer network



- ReLU $g(u_i) = \max(0, u_i)$, Softmax $\sigma(\mathbf{z})$
- Cross entropy loss $l(\mathbf{y}, \hat{\mathbf{y}})$
- Model parameters $\boldsymbol{\theta} = [w_1, \ldots, w_8, b_1, \ldots, b_4]^{\mathrm{T}}$

$$\mathbf{W}^h = \begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix}, \mathbf{b}^h = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}, \mathbf{W}^o = \begin{bmatrix} w_5 & w_7 \\ w_6 & w_8 \end{bmatrix}, \mathbf{b}^o = \begin{bmatrix} b_3 \\ b_4 \end{bmatrix}$$

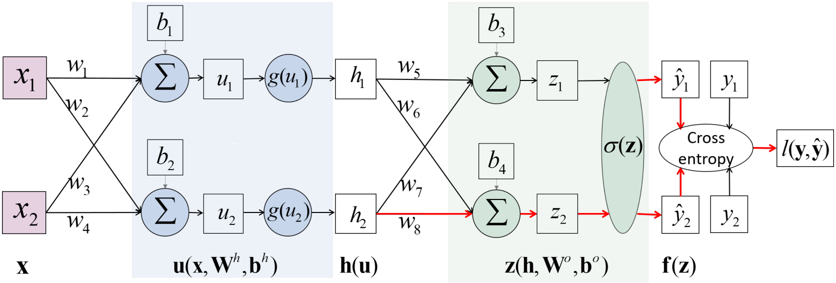
Backpropagation: one derivative example



- Backpropagation: computation of gradient with chain rule
- an example:

$$\frac{\partial l}{\partial w_8} = \frac{\partial l}{\partial z_2} \frac{\partial z_2}{\partial w_8}$$

Backpropagation: one derivative example (cont')



- Pre-softmax function $z_2 = h_1 w_6 + h_2 w_8 + b_4$, therefore

$$\frac{\partial z_2}{\partial w_8} = h_2$$

- To compute $\frac{\partial l}{\partial z_2}$, need to compute derivative of softmax $\frac{\partial \hat{y}_j}{\partial z_2}$

Derivative of softmax function

Softmax function

$$\hat{y}_j = \sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

$$\frac{\partial \hat{y}_j}{\partial z_i} = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \frac{dz_j}{dz_i} - \frac{e^{z_j}}{(\sum_{k=1}^{K} e^{z_k})^2} e^{z_i}$$

$$= \hat{y}_j \frac{dz_j}{dz_i} - \hat{y}_j \hat{y}_i$$

$$= \begin{cases} \hat{y}_i(1-\hat{y}_i), & \text{if } i = j \\ -\hat{y}_i \hat{y}_j, & \text{if } i \neq j \end{cases}$$

Derivative of softmax function

Softmax function

$$\hat{y}_j = \sigma(\mathbf{z})_j \quad = \quad \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

$$\begin{aligned}
\frac{\partial \hat{y}_j}{\partial z_i} &= \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}} \frac{dz_j}{dz_i} - \frac{e^{z_j}}{(\sum_{k=1}^K e^{z_k})^2} e^{z_i} \\
&= \hat{y}_j \frac{dz_j}{dz_i} - \hat{y}_j \hat{y}_i \\
&= \begin{cases} \hat{y}_i(1 - \hat{y}_i), & \text{if } i = j \\ -\hat{y}_i \hat{y}_j, & \text{if } i \neq j \end{cases}
\end{aligned}$$

Derivative of softmax function

Softmax function

$$\hat{y}_j = \sigma(\mathbf{z})_j \quad = \quad \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

$$\begin{aligned}
\frac{\partial \hat{y}_j}{\partial z_i} &= \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \frac{dz_j}{dz_i} - \frac{e^{z_j}}{(\sum_{k=1}^{K} e^{z_k})^2} e^{z_i} \\
&= \hat{y}_j \frac{dz_j}{dz_i} - \hat{y}_j \hat{y}_i \\
&= \begin{cases} \hat{y}_i(1 - \hat{y}_i), & \text{if } i = j \\ -\hat{y}_i \hat{y}_j, & \text{if } i \neq j \end{cases}
\end{aligned}$$

Derivative of loss function over pre-softmax

- Network output $\hat{\mathbf{y}} = \mathbf{f}(\mathbf{x};\boldsymbol{\theta}) = (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_K)$
- Ideal output $\mathbf{y} = (y_1, \ldots, y_K) = (0, \ldots, 1, \ldots, 0)$
- Cross entropy loss

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{k=1}^{K} y_k \log \hat{y}_k$$

- Derivative of l over z_i, with chain rule

$$\begin{aligned}
\frac{\partial l}{\partial z_i} &= -\sum_{k=1}^{K} y_k \frac{\partial \log \hat{y}_k}{\partial z_i} = -\sum_{k=1}^{K} y_k \frac{1}{\hat{y}_k}\frac{\partial \hat{y}_k}{\partial z_i} \\
&= -y_i \frac{1}{\hat{y}_i}\hat{y}_i(1-\hat{y}_i) - \sum_{k \neq i} y_k \frac{1}{\hat{y}_k}(-\hat{y}_k\hat{y}_i) \\
&= \hat{y}_i\left(\sum_{k=1}^{K} y_k\right) - y_i = \hat{y}_i - y_i
\end{aligned}$$

Derivative of loss function over pre-softmax

- Network output $\hat{\mathbf{y}} = \mathbf{f}(\mathbf{x}; \boldsymbol{\theta}) = (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_K)$
- Ideal output $\mathbf{y} = (y_1, \ldots, y_K) = (0, \ldots, 1, \ldots, 0)$
- Cross entropy loss

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{k=1}^{K} y_k \log \hat{y}_k$$

- Derivative of l over z_i, with chain rule

$$\begin{aligned}
\frac{\partial l}{\partial z_i} &= -\sum_{k=1}^{K} y_k \frac{\partial \log \hat{y}_k}{\partial z_i} = -\sum_{k=1}^{K} y_k \frac{1}{\hat{y}_k} \frac{\partial \hat{y}_k}{\partial z_i} \\
&= -y_i \frac{1}{\hat{y}_i} \hat{y}_i (1 - \hat{y}_i) - \sum_{k \neq i} y_k \frac{1}{\hat{y}_k} (-\hat{y}_k \hat{y}_i) \\
&= \hat{y}_i \left(\sum_{k=1}^{K} y_k\right) - y_i = \hat{y}_i - y_i
\end{aligned}$$

Derivative of loss function over pre-softmax

- Network output $\hat{\mathbf{y}} = \mathbf{f}(\mathbf{x}; \boldsymbol{\theta}) = (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_K)$
- Ideal output $\mathbf{y} = (y_1, \ldots, y_K) = (0, \ldots, 1, \ldots, 0)$
- Cross entropy loss

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{k=1}^{K} y_k \log \hat{y}_k$$

- Derivative of l over z_i, with chain rule

$$\begin{aligned}
\frac{\partial l}{\partial z_i} &= -\sum_{k=1}^{K} y_k \frac{\partial \log \hat{y}_k}{\partial z_i} = -\sum_{k=1}^{K} y_k \frac{1}{\hat{y}_k} \frac{\partial \hat{y}_k}{\partial z_i} \\
&= -y_i \frac{1}{\hat{y}_i} \hat{y}_i (1 - \hat{y}_i) - \sum_{k \neq i} y_k \frac{1}{\hat{y}_k} (-\hat{y}_k \hat{y}_i) \\
&= \hat{y}_i \left(\sum_{k=1}^{K} y_k\right) - y_i = \hat{y}_i - y_i
\end{aligned}$$

Derivative of loss function over pre-softmax

- Network output $\hat{\mathbf{y}} = \mathbf{f}(\mathbf{x}; \boldsymbol{\theta}) = (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_K)$
- Ideal output $\mathbf{y} = (y_1, \ldots, y_K) = (0, \ldots, 1, \ldots, 0)$
- Cross entropy loss

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{k=1}^{K} y_k \log \hat{y}_k$$

- Derivative of l over z_i, with chain rule

$$\begin{aligned}
\frac{\partial l}{\partial z_i} &= -\sum_{k=1}^{K} y_k \frac{\partial \log \hat{y}_k}{\partial z_i} = -\sum_{k=1}^{K} y_k \frac{1}{\hat{y}_k} \frac{\partial \hat{y}_k}{\partial z_i} \\
&= -y_i \frac{1}{\hat{y}_i} \hat{y}_i (1 - \hat{y}_i) - \sum_{k \neq i} y_k \frac{1}{\hat{y}_k} (-\hat{y}_k \hat{y}_i) \\
&= \hat{y}_i \left(\sum_{k=1}^{K} y_k\right) - y_i = \hat{y}_i - y_i
\end{aligned}$$

Derivative of loss function over weight parameters

From $\frac{\partial z_2}{\partial w_8} = h_2$ and $\frac{\partial l}{\partial z_i} = \hat{y}_i - y_i$

$$\frac{\partial l}{\partial w_8} = \frac{\partial l}{\partial z_2}\frac{\partial z_2}{\partial w_8} = h_2(\hat{y}_2 - y_2)$$

Similarly,

$$\frac{\partial l}{\partial w_7} = \frac{\partial l}{\partial z_1}\frac{\partial z_1}{\partial w_7} = h_2(\hat{y}_1 - y_1)$$

$$\frac{\partial l}{\partial b_4} = \frac{\partial l}{\partial z_2}\frac{\partial z_2}{\partial b_4} = \hat{y}_2 - y_2$$

Why called backpropagation?

- Propagation of prediction errors $\hat{y}_i - y_i$ in gradient!

Derivative of loss function over weight parameters

From $\frac{\partial z_2}{\partial w_8} = h_2$ and $\frac{\partial l}{\partial z_i} = \hat{y}_i - y_i$

$$\frac{\partial l}{\partial w_8} = \frac{\partial l}{\partial z_2}\frac{\partial z_2}{\partial w_8} = h_2(\hat{y}_2 - y_2)$$

Similarly,

$$\frac{\partial l}{\partial w_7} = \frac{\partial l}{\partial z_1}\frac{\partial z_1}{\partial w_7} = h_2(\hat{y}_1 - y_1)$$
$$\frac{\partial l}{\partial b_4} = \frac{\partial l}{\partial z_2}\frac{\partial z_2}{\partial b_4} = \hat{y}_2 - y_2$$

Why called backpropagation?

- Propagation of prediction errors $\hat{y}_i - y_i$ in gradient!

Derivative of loss function over weight parameters

From $\frac{\partial z_2}{\partial w_8} = h_2$ and $\frac{\partial l}{\partial z_i} = \hat{y}_i - y_i$

$$\frac{\partial l}{\partial w_8} = \frac{\partial l}{\partial z_2}\frac{\partial z_2}{\partial w_8} = h_2(\hat{y}_2 - y_2)$$

Similarly,

$$\frac{\partial l}{\partial w_7} = \frac{\partial l}{\partial z_1}\frac{\partial z_1}{\partial w_7} = h_2(\hat{y}_1 - y_1)$$
$$\frac{\partial l}{\partial b_4} = \frac{\partial l}{\partial z_2}\frac{\partial z_2}{\partial b_4} = \hat{y}_2 - y_2$$

Why called backpropagation?

- Propagation of prediction errors $\hat{y}_i - y_i$ in gradient!

Derivative of loss function over weight parameters

From $\frac{\partial z_2}{\partial w_8} = h_2$ and $\frac{\partial l}{\partial z_i} = \hat{y}_i - y_i$

$$\frac{\partial l}{\partial w_8} = \frac{\partial l}{\partial z_2}\frac{\partial z_2}{\partial w_8} = h_2(\hat{y}_2 - y_2)$$

Similarly,

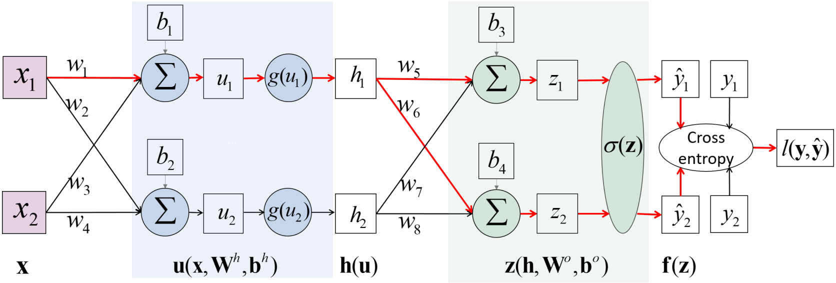
$$\frac{\partial l}{\partial w_7} = \frac{\partial l}{\partial z_1}\frac{\partial z_1}{\partial w_7} = h_2(\hat{y}_1 - y_1)$$

$$\frac{\partial l}{\partial b_4} = \frac{\partial l}{\partial z_2}\frac{\partial z_2}{\partial b_4} = \hat{y}_2 - y_2$$

Why called backpropagation?

- Propagation of prediction errors $\hat{y}_i - y_i$ in gradient!

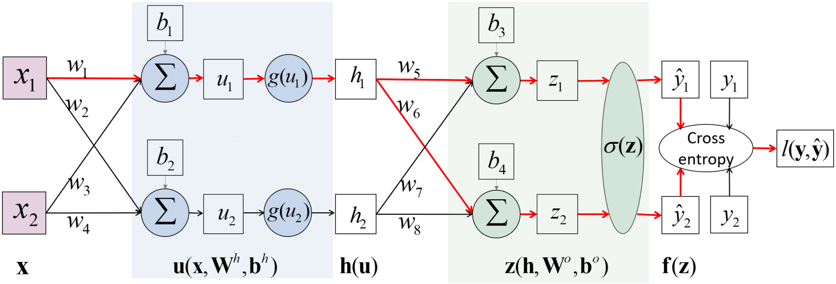
Backpropagation: another derivative



- another example:

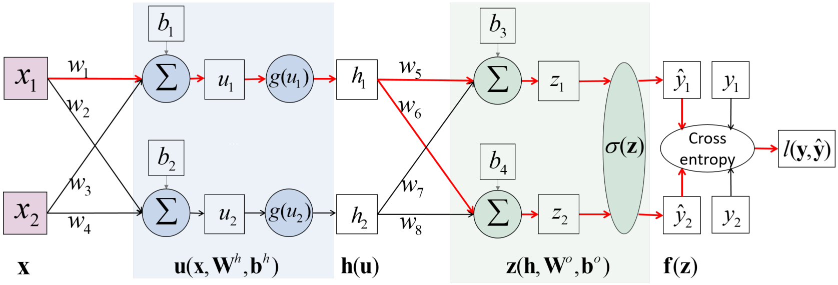
$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial h_1}\frac{\partial h_1}{\partial w_1} = \left(\sum_i \frac{\partial l}{\partial z_i} \cdot \frac{\partial z_i}{\partial h_1}\right) \cdot \left(\frac{dh_1}{du_1} \cdot \frac{\partial u_1}{\partial w_1}\right)$$

Backpropagation: another derivative (cont')



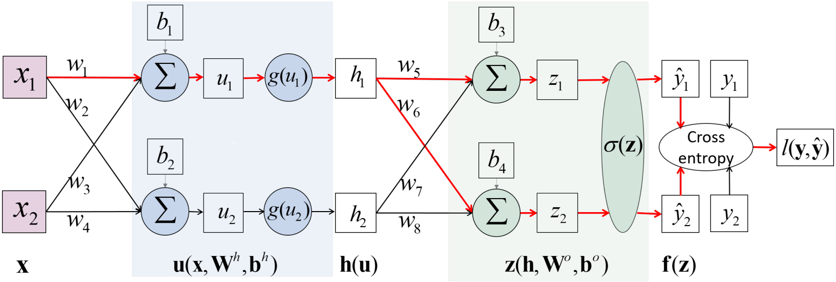
$$
\begin{aligned}
\frac{\partial u_1}{\partial w_1} &= \frac{\partial}{\partial w_1}(x_1 w_1 + x_2 w_2 + b_1) = x_1 \\
\frac{dh_1}{du_1} &= \frac{d}{du_1}(\max(0, u_1)) = 1_{u_1 > 0}
\end{aligned}
$$

Backpropagation: another derivative (cont')



$$
\begin{aligned}
\frac{\partial z_1}{\partial h_1} &= \frac{\partial}{\partial h_1}(h_1 w_5 + h_2 w_7 + b_3) = w_5 \\
\frac{\partial z_2}{\partial h_1} &= \frac{\partial}{\partial h_1}(h_1 w_6 + h_2 w_8 + b_4) = w_6
\end{aligned}
$$

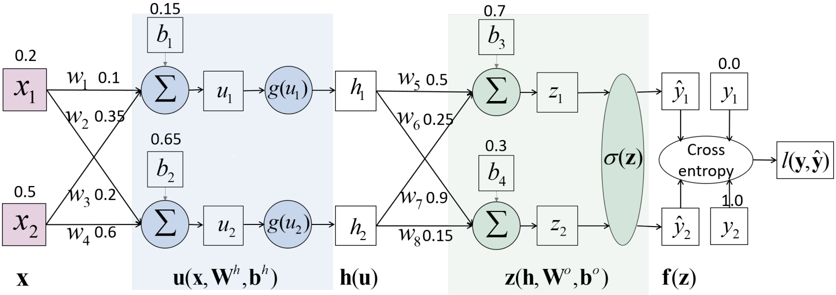
Backpropagation: another derivative (cont')



$$
\begin{aligned}
\frac{\partial l}{\partial w_1} &= \left(\sum_i \frac{\partial l}{\partial z_i} \cdot \frac{\partial z_i}{\partial h_1}\right) \cdot \left(\frac{d h_1}{d u_1} \cdot \frac{\partial u_1}{\partial w_1}\right) \\
&= \begin{cases} x_1\{w_5(\hat{y}_1 - y_1) + w_6(\hat{y}_2 - y_2)\}, & \text{if } u_1 > 0 \\ 0, & \text{if } u_1 \le 0 \end{cases}
\end{aligned}
$$

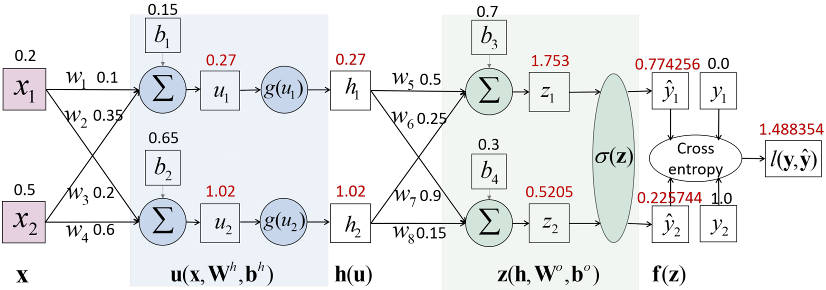
Backpropagation: computation example

- Suppose we have a single input $\mathbf{x} = [0.2, 0.5]^\mathrm{T}$
- Ideal (expected) output $\mathbf{y} = [0, 1]^\mathrm{T}$
- Model parameters $\boldsymbol{\theta}$ was initialized as follows



Backpropagation: computation example (cont')

- Forward computation first, with results in red

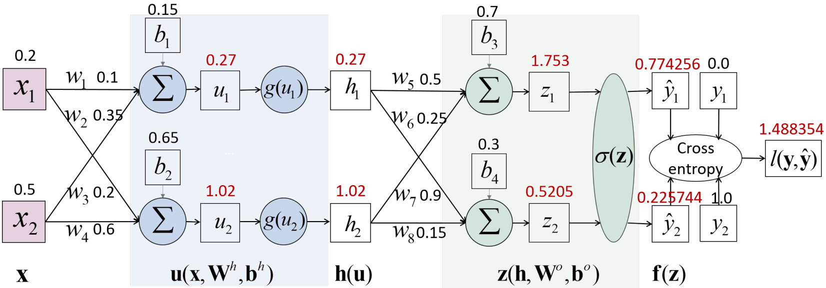


- Suppose learning rate $\eta = 0.5$, update with gradient descent,

$$
\begin{aligned}
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta \nabla L(\boldsymbol{\theta}_t) \\
\frac{\partial l}{\partial w_8} &= h_2(\hat{y}_2 - y_2) = 1.02 * (0.225744 - 1) = -0.789741 \\
w_8 &= 0.15 - 0.5 * (-0.789741) = 0.544871
\end{aligned}
$$

Backpropagation: computation example (cont')

- Forward computation first, with results in red

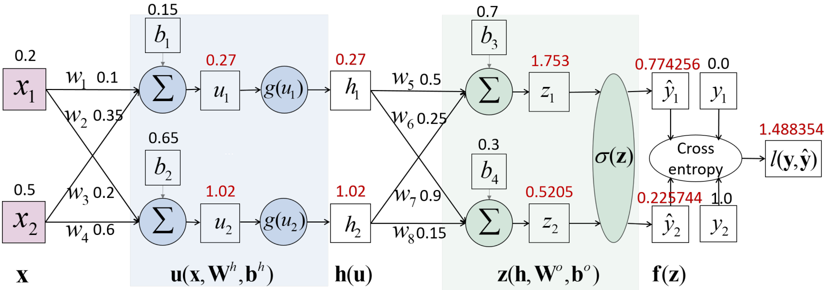


- Suppose learning rate $\eta = 0.5$, update with gradient descent,

$$\begin{aligned}
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta \nabla L(\boldsymbol{\theta}_t) \\
\frac{\partial l}{\partial w_8} &= h_2(\hat{y}_2 - y_2) = 1.02 * (0.225744 - 1) = -0.789741 \\
w_8 &= 0.15 - 0.5 * (-0.789741) = 0.544871
\end{aligned}$$

Backpropagation: computation example (cont')

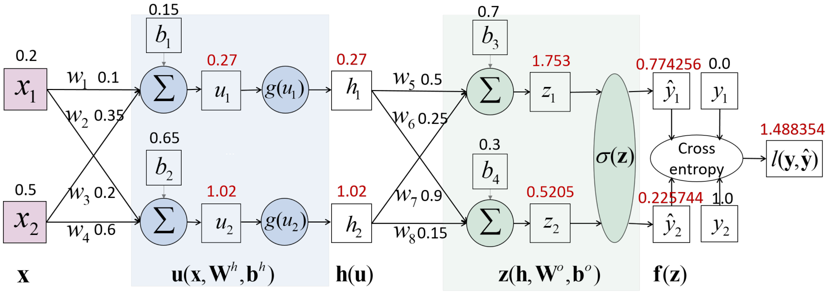
- Forward computation first, with results in red



- Suppose learning rate $\eta = 0.5$, update with gradient descent,

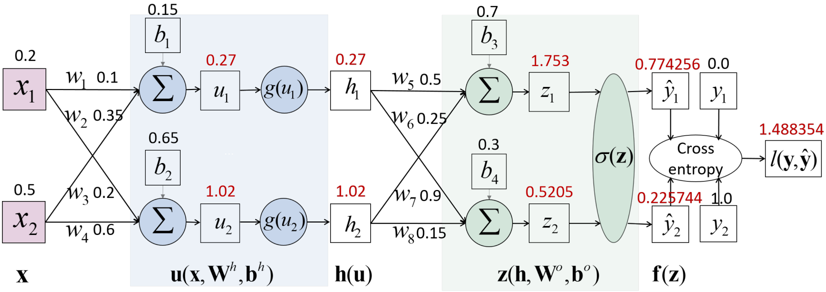
$$
\begin{aligned}
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta \nabla L(\boldsymbol{\theta}_t) \\
\frac{\partial l}{\partial w_8} &= h_2(\hat{y}_2 - y_2) = 1.02 * (0.225744 - 1) = -0.789741 \\
w_8 &= 0.15 - 0.5 * (-0.789741) = 0.544871
\end{aligned}
$$

Backpropagation: computation example (cont')



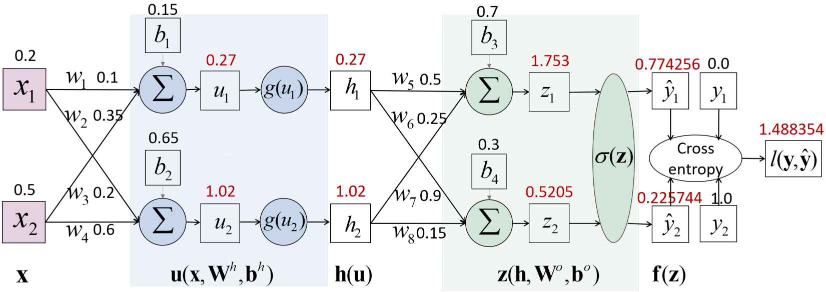
$$
\begin{aligned}
\frac{\partial l}{\partial w_1} &= x_1\{w_5(\hat{y}_1 - y_1) + w_6(\hat{y}_2 - y_2)\} \\
&= 0.2 * \{0.5 * (0.774256 - 0) + 0.25 * (0.225744 - 1.0)\} \\
&= 0.038713 \\
w_1 &= 0.1 - 0.5 * 0.038713 = 0.080644
\end{aligned}
$$

Backpropagation: computation example (cont')



$$
\begin{aligned}
\frac{\partial l}{\partial w_1} &= x_1\{w_5(\hat{y}_1 - y_1) + w_6(\hat{y}_2 - y_2)\} \\
&= 0.2 * \{0.5 * (0.774256 - 0) + 0.25 * (0.225744 - 1.0)\} \\
&= 0.038713 \\
w_1 &= 0.1 - 0.5 * 0.038713 = 0.080644
\end{aligned}
$$

Backpropagation: computation example (cont')



$$
\begin{aligned}
\frac{\partial l}{\partial w_1} &= x_1\{w_5(\hat{y}_1 - y_1) + w_6(\hat{y}_2 - y_2)\} \\
&= 0.2 * \{0.5 * (0.774256 - 0) + 0.25 * (0.225744 - 1.0)\} \\
&= 0.038713 \\
w_1 &= 0.1 - 0.5 * 0.038713 = 0.080644
\end{aligned}
$$

Above example used *single* data to update model parameters.

How to update parameters if we have lots of (training) data?

- Training dataset $D = \{(\mathbf{x}_n, \mathbf{y}_n) | n = 1, \ldots, N\}$
- Loss function

$$L(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} l(\mathbf{y}_n, \mathbf{f}(\mathbf{x}_n; \boldsymbol{\theta}))$$

Above example used *single* data to update model parameters.

How to update parameters if we have lots of (training) data?

- Training dataset $D = \{(\mathbf{x}_n, \mathbf{y}_n) | n = 1, \ldots, N\}$
- Loss function

$$L(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} l(\mathbf{y}_n, \mathbf{f}(\mathbf{x}_n; \boldsymbol{\theta}))$$

Batch gradient descent

- Update model parameters using all data with gradient descent

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta \nabla L(\boldsymbol{\theta}_t) \\ &= \boldsymbol{\theta}_t - \frac{\eta}{N} \sum_{n=1}^{N} \nabla l(\mathbf{y}_n, \mathbf{f}(\mathbf{x}_n; \boldsymbol{\theta}_t))\end{aligned}$$

  where η is learning rate, $\nabla L(\boldsymbol{\theta}_t)$ is gradient of loss function L over current model parameters $\boldsymbol{\theta}_t$.
- Very slow, and intractable for big data to fit in memory
- Guarantee to find minimum of loss function

```
for iter in range(nb_epochs):
    params_grad = eval_grad(loss_func, dataset, params)
    params = params - learning_rate * params_grad
```

Batch gradient descent

- Update model parameters using all data with gradient descent

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta \nabla L(\boldsymbol{\theta}_t) \\ &= \boldsymbol{\theta}_t - \frac{\eta}{N} \sum_{n=1}^{N} \nabla l(\mathbf{y}_n, \mathbf{f}(\mathbf{x}_n; \boldsymbol{\theta}_t))\end{aligned}$$

  where η is learning rate, $\nabla L(\boldsymbol{\theta}_t)$ is gradient of loss function L over current model parameters $\boldsymbol{\theta}_t$.
- Very slow, and intractable for big data to fit in memory
- Guarantee to find minimum of loss function

```
for iter in range(nb_epochs):
    params_grad = eval_grad(loss_func, dataset, params)
    params = params - learning_rate * params_grad
```

This section is mainly from http://ruder.io/optimizing-gradient-descent/

Stochastic gradient descent

- Update model parameters using each single data $(\mathbf{x}_n, \mathbf{y}_n)$

$$
\begin{aligned}
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta \nabla L(\boldsymbol{\theta}_t) \\
&= \boldsymbol{\theta}_t - \eta \nabla l(\mathbf{y}_n, \mathbf{f}(\mathbf{x}_n; \boldsymbol{\theta}_t))
\end{aligned}
$$

- Very fast update, tractable for big data
- May jump to better local minimum
- Converge almost certainly to local minimum
- Convergence fluctuates

```
for i in range(nb_epochs):
    np.random.shuffle(dataset)
    for example in dataset:
        params_grad = eval_grad(loss_func, example, params)
        params = params - learning_rate * params_grad
```

Stochastic gradient descent

- Update model parameters using each single data $(\mathbf{x}_n, \mathbf{y}_n)$

$$
\begin{aligned}
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta \nabla L(\boldsymbol{\theta}_t) \\
&= \boldsymbol{\theta}_t - \eta \nabla l(\mathbf{y}_n, \mathbf{f}(\mathbf{x}_n; \boldsymbol{\theta}_t))
\end{aligned}
$$

- Very fast update, tractable for big data
- May jump to better local minimum
- Converge almost certainly to local minimum
- Convergence fluctuates

```
for i in range(nb_epochs):
    np.random.shuffle(dataset)
    for example in dataset:
        params_grad = eval_grad(loss_func, example, params)
        params = params - learning_rate * params_grad
```

Mini-batch gradient descent

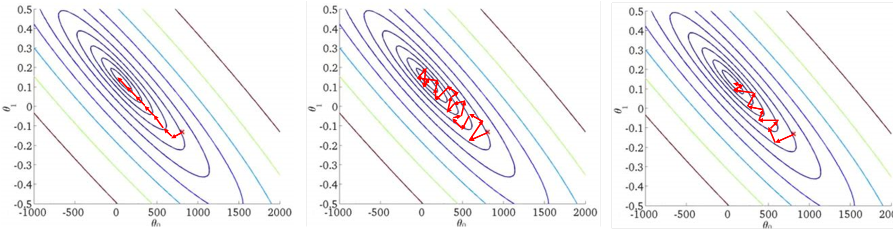
- Update model parameters using a mini-batch of data $S \subset D$

$$
\begin{aligned}
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta \nabla L(\boldsymbol{\theta}_t) \\
&= \boldsymbol{\theta}_t - \frac{\eta}{|S|} \sum_{(\mathbf{x}_n, \mathbf{y}_n) \in S} \nabla l(\mathbf{y}_n, \mathbf{f}(\mathbf{x}_n; \boldsymbol{\theta}_t))
\end{aligned}
$$

- In general $|S|$ ranges between tens and two/three hundreds
- Take the best of batch and stochastic gradient descents

```
for i in range(nb_epochs):
    np.random.shuffle(dataset)
    for batch in get_batches(dataset, batch_size=64):
        params_grad = eval_grad(loss_func, batch, params)
        params = params - learning_rate * params_grad
```

Mini-batch gradient descent (cont')

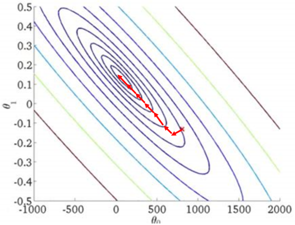


Batch gradient descent　　Stochastic gradient descent　　Mini-batch gradient descent

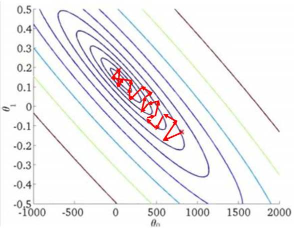
Challenges of mini-batch stochastic gradient

- adjust learning rate
- well approximate true gradient with mini-batch data

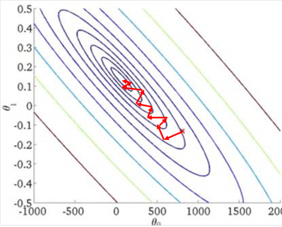
Mini-batch gradient descent (cont')



Batch gradient descent



Stochastic gradient descent



Mini-batch gradient descent

Challenges of mini-batch stochastic gradient

- adjust learning rate
- well approximate true gradient with mini-batch data

Momentum

- consider downhill directions from previous steps

$$
\begin{aligned}
\boldsymbol{v}_{t+1} &= \gamma \boldsymbol{v}_t + \eta \nabla L(\boldsymbol{\theta}_t) \\
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \boldsymbol{v}_{t+1}
\end{aligned}
$$

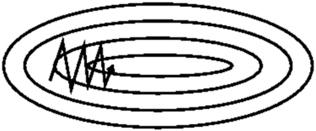
- $\gamma = 0.9$ or similar value
- $\boldsymbol{v}_{t+1} = \eta\{\nabla L(\boldsymbol{\theta}_t) + \gamma \nabla L(\boldsymbol{\theta}_{t-1}) + \gamma^2 \nabla L(\boldsymbol{\theta}_{t-2}) + \ldots\}$
- reduce oscillations in stochastic/mini-batch gradient descent

Momentum

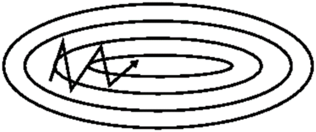
- consider downhill directions from previous steps

$$\begin{aligned}\boldsymbol{v}_{t+1} &= \gamma \boldsymbol{v}_t + \eta \nabla L(\boldsymbol{\theta}_t) \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \boldsymbol{v}_{t+1}\end{aligned}$$

- $\gamma = 0.9$ or similar value
- $\boldsymbol{v}_{t+1} = \eta\{\nabla L(\boldsymbol{\theta}_t) + \gamma \nabla L(\boldsymbol{\theta}_{t-1}) + \gamma^2 \nabla L(\boldsymbol{\theta}_{t-2}) + \ldots\}$
- reduce oscillations in stochastic/mini-batch gradient descent



(a) SGD without momentum



(b) SGD with momentum

Momentum (cont')

- Momentum may cause ball roll down too fast (red arrow)!
- Need a smarter ball knowing when to slow down before the hill slopes up again (blue arrow).

Nesterov accelerated gradient (NAG)

- Look ahead, i.e., consider downhill direction at future's approximate position $\boldsymbol{\theta}_t - \gamma \boldsymbol{v}_t$

$$\begin{aligned} \boldsymbol{v}_{t+1} &= \gamma \boldsymbol{v}_t + \eta \nabla L(\boldsymbol{\theta}_t - \gamma \boldsymbol{v}_t) \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \boldsymbol{v}_{t+1} \end{aligned}$$

- Avoid ball going down too fast with accumulated momentum

Nesterov accelerated gradient (NAG)

- Look ahead, i.e., consider downhill direction at future's approximate position $\boldsymbol{\theta}_t - \gamma \boldsymbol{v}_t$

$$
\begin{aligned}
\boldsymbol{v}_{t+1} &= \gamma \boldsymbol{v}_t + \eta \nabla L(\boldsymbol{\theta}_t - \gamma \boldsymbol{v}_t) \\
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \boldsymbol{v}_{t+1}
\end{aligned}
$$

- Avoid ball going down too fast with accumulated momentum

Nesterov accelerated gradient (NAG)

- Look ahead, i.e., consider downhill direction at future's approximate position $\boldsymbol{\theta}_t - \gamma \boldsymbol{v}_t$

$$
\begin{aligned}
\boldsymbol{v}_{t+1} &= \gamma \boldsymbol{v}_t + \eta \nabla L(\boldsymbol{\theta}_t - \gamma \boldsymbol{v}_t) \\
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \boldsymbol{v}_{t+1}
\end{aligned}
$$

- Avoid ball going down too fast with accumulated momentum

Momentum and NAG update gradient.
Can we update learning rate over iterations?

Adagrad

- Adaptively update learning rate for each parameter, with square root of sum of squares of its historical derivatives

$$
\begin{aligned}
g_{t+1,i} &= g_{t,i} + \left(\frac{\partial L(\boldsymbol{\theta}_t)}{\partial \theta_i}\right)^2 \\
\theta_{t+1,i} &= \theta_{t,i} - \frac{\eta}{\sqrt{g_{t+1,i}} + \epsilon} \frac{\partial L(\boldsymbol{\theta}_t)}{\partial \theta_i}
\end{aligned}
$$

- Larger update in history for θ_i will cause smaller learning rate $\eta/(\sqrt{g_{t+1,i}} + \epsilon)$ in updating θ_i.
- No need to manually tune learning rate.
- $g_{t+1,i}$ becomes larger over iterations, causing very small learning rate ultimately for each θ_i.

Adagrad

- Adaptively update learning rate for each parameter, with square root of sum of squares of its historical derivatives

$$
\begin{aligned}
g_{t+1,i} &= g_{t,i} + \left(\frac{\partial L(\boldsymbol{\theta}_t)}{\partial \theta_i}\right)^2 \\
\theta_{t+1,i} &= \theta_{t,i} - \frac{\eta}{\sqrt{g_{t+1,i}} + \epsilon} \frac{\partial L(\boldsymbol{\theta}_t)}{\partial \theta_i}
\end{aligned}
$$

- Larger update in history for θ_i will cause smaller learning rate $\eta/(\sqrt{g_{t+1,i}} + \epsilon)$ in updating θ_i.
- No need to manually tune learning rate.
- $g_{t+1,i}$ becomes larger over iterations, causing very small learning rate ultimately for each θ_i.

RMSprop (Root Mean Square propagation)

- Solve monotonically decreasing learning rate issue in Adagrad

$$
\begin{aligned}
g_{t+1,i} &= \gamma g_{t,i} + (1-\gamma)\left(\frac{\partial L(\boldsymbol{\theta}_t)}{\partial \theta_i}\right)^2 \\
\theta_{t+1,i} &= \theta_{t,i} - \frac{\eta}{\sqrt{g_{t+1,i}} + \epsilon} \frac{\partial L(\boldsymbol{\theta}_t)}{\partial \theta_i}
\end{aligned}
$$

- $\gamma = 0.9$ or similar value
- $g_{t+1,i}$ is exponential decaying average of squared derivatives
- No need to manually tune learning rate

RMSprop (Root Mean Square propagation)

- Solve monotonically decreasing learning rate issue in Adagrad

$$
\begin{aligned}
g_{t+1,i} &= \gamma g_{t,i} + (1-\gamma)\left(\frac{\partial L(\boldsymbol{\theta}_t)}{\partial \theta_i}\right)^2 \\
\theta_{t+1,i} &= \theta_{t,i} - \frac{\eta}{\sqrt{g_{t+1,i}} + \epsilon} \frac{\partial L(\boldsymbol{\theta}_t)}{\partial \theta_i}
\end{aligned}
$$

- $\gamma = 0.9$ or similar value
- $g_{t+1,i}$ is exponential decaying average of squared derivatives
- No need to manually tune learning rate

Adam (Adaptive Moment Estimation)

- Adds momentum and bias to RMSprop

$$
\begin{aligned}
m_{t+1,i} &= \beta_1 m_{t,i} + (1-\beta_1)\frac{\partial L(\boldsymbol{\theta}_t)}{\partial \theta_i} \\
g_{t+1,i} &= \beta_2 g_{t,i} + (1-\beta_2)\left(\frac{\partial L(\boldsymbol{\theta}_t)}{\partial \theta_i}\right)^2 \\
\hat{m}_{t+1,i} &= \frac{m_{t+1,i}}{1-\beta_1^{t+1}} \\
\hat{g}_{t+1,i} &= \frac{g_{t+1,i}}{1-\beta_2^{t+1}} \\
\theta_{t+1,i} &= \theta_{t,i} - \frac{\eta}{\sqrt{\hat{g}_{t+1,i}}+\epsilon}\hat{m}_{t+1,i}
\end{aligned}
$$

- $m_{t+1,i}$ and $g_{t+1,i}$ are first (mean) and second moment (uncentered variance) of derivative
- $m_{t+1,i}$ and $g_{t+1,i}$ are biased toward zero, so $\hat{m}_{t+1,i}$ and $\hat{g}_{t+1,i}$

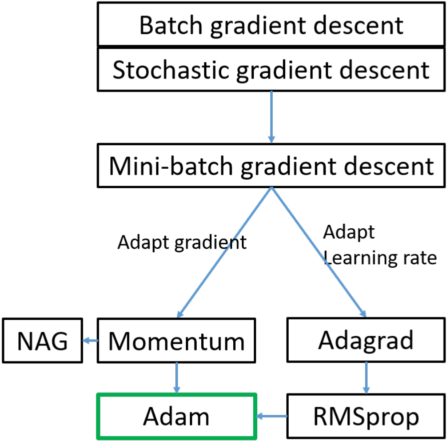
Adam (Adaptive Moment Estimation)

- Adds momentum and bias to RMSprop

$$
\begin{aligned}
m_{t+1,i} &= \beta_1 m_{t,i} + (1-\beta_1)\frac{\partial L(\boldsymbol{\theta}_t)}{\partial \theta_i} \\
g_{t+1,i} &= \beta_2 g_{t,i} + (1-\beta_2)\left(\frac{\partial L(\boldsymbol{\theta}_t)}{\partial \theta_i}\right)^2 \\
\hat{m}_{t+1,i} &= \frac{m_{t+1,i}}{1-\beta_1^{t+1}} \\
\hat{g}_{t+1,i} &= \frac{g_{t+1,i}}{1-\beta_2^{t+1}} \\
\theta_{t+1,i} &= \theta_{t,i} - \frac{\eta}{\sqrt{\hat{g}_{t+1,i}}+\epsilon}\hat{m}_{t+1,i}
\end{aligned}
$$

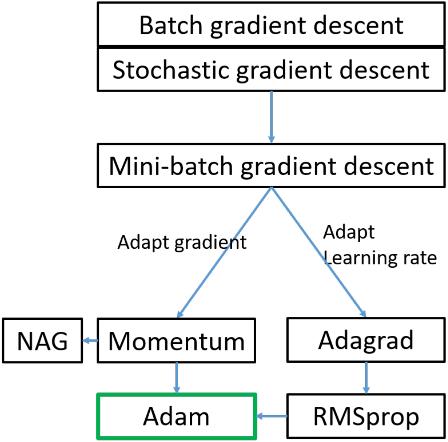
- $m_{t+1,i}$ and $g_{t+1,i}$ are first (mean) and second moment (uncentered variance) of derivative
- $m_{t+1,i}$ and $g_{t+1,i}$ are biased toward zero, so $\hat{m}_{t+1,i}$ and $\hat{g}_{t+1,i}$

Gradient descent: summary



- However, actually vanilla SGD or mini-batch gradient with simple decreasing learning rate schedule works well!

Gradient descent: summary



- However, actually vanilla SGD or mini-batch gradient with simple decreasing learning rate schedule works well!

Hyper-parameter tuning

The above is about finding network's weight parameters.

We also need to determine

- parameters in gradient descent, e.g., learning rate
- number of network layers and number of neurons per layer

These are called *hyper-parameters*!

Hyper-parameter tuning

The above is about finding network's weight parameters.

We also need to determine

- parameters in gradient descent, e.g., learning rate
- number of network layers and number of neurons per layer

These are called *hyper-parameters*!

Hyper-parameter tuning (cont')

How to choose from multiple sets of hyper-parameter values?

- Rule: choose the set with which the trained network model has better *generalization* performance!

Generalization ability

How well does the trained model work for unseen data?

Hyper-parameter tuning (cont')

How to choose from multiple sets of hyper-parameter values?

- Rule: choose the set with which the trained network model has better *generalization* performance!

Generalization ability

How well does the trained model work for unseen data?

Hyper-parameter tuning (cont')

How to choose from multiple sets of hyper-parameter values?

- Rule: choose the set with which the trained network model has better *generalization* performance!

Generalization ability

How well does the trained model work for unseen data?

举一反三

Data set for hyper-parameter tuning

Whole data set divided:

- Training set (e.g., 60%): used to train model parameter
- Validation set (e.g., 20%): to find optimal hyper-parameters
- Test set (e.g., 20%): to evaluate final model's performance

Training set

Validation set

Test set

- Choose the set of hyper-parameters with which the trained model has the best performance on the validation set.

Data set for hyper-parameter tuning

Whole data set divided:

- Training set (e.g., 60%): used to train model parameter
- Validation set (e.g., 20%): to find optimal hyper-parameters
- Test set (e.g., 20%): to evaluate final model's performance

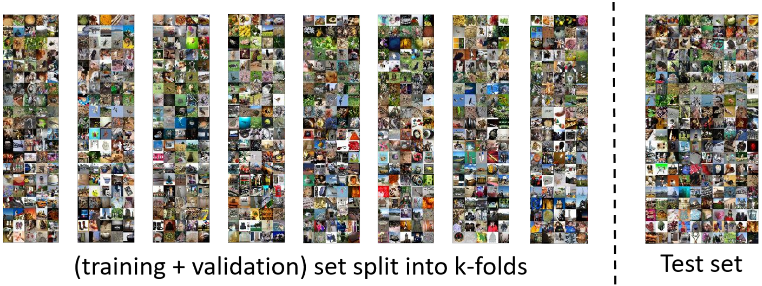
Training set Validation set Test set

- Choose the set of hyper-parameters with which the trained model has the best performance on the validation set.

Data set for hyper-parameter tuning (cont')

When whole set is small: K-fold cross validation

- for each run, use one unique subset as validation set, the other (K-1) subsets as training set
- average performance over K runs



Hyper-parameter tuning: grid search

Where are the multiple sets of hyper-parameters from?

Grid search

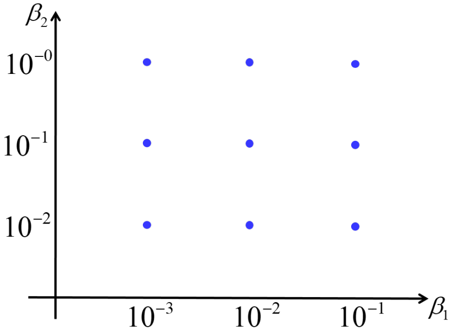
- exhaustive searching in hyper-parameter space
- often in log scale
- only for small number of hyper-parameters

Hyper-parameter tuning: grid search

Where are the multiple sets of hyper-parameters from?

Grid search

- exhaustive searching in hyper-parameter space
- often in log scale
- only for small number of hyper-parameters

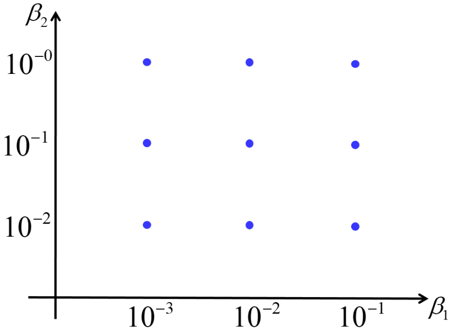


Hyper-parameter tuning: grid search

Where are the multiple sets of hyper-parameters from?

Grid search

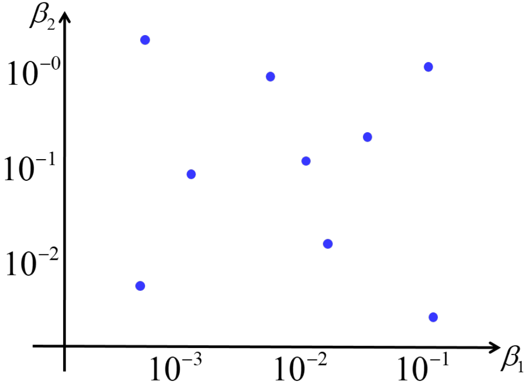
- exhaustive searching in hyper-parameter space
- often in log scale
- only for small number of hyper-parameters



Hyper-parameter tuning: random search

Random search

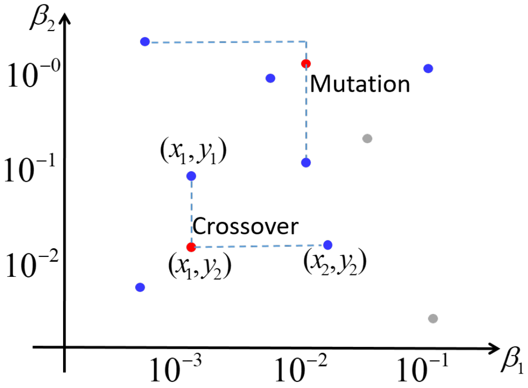
- randomly sample multiple times in hyper-parameter space
- better than grid method if hyper-parameter number is higher



Hyper-parameter tuning: evolutionary method

Evolutionary optimization

- repeat: replace worst-performing hyper-parameter sets (gray) with new sets (red) through crossover and mutation



Google used the method to find better network structures; see Jaderberg et al., 2017,"Population Based Training of Neural Networks"

Deep learning frameworks/platforms

In practice, how do we get a real deep learning system?

- construct the structure of a neural network
- minimize the loss function with training dataset

Deep learning frameworks/platforms

In practice, how do we get a real deep learning system?

- construct the structure of a neural network
- minimize the loss function with training dataset

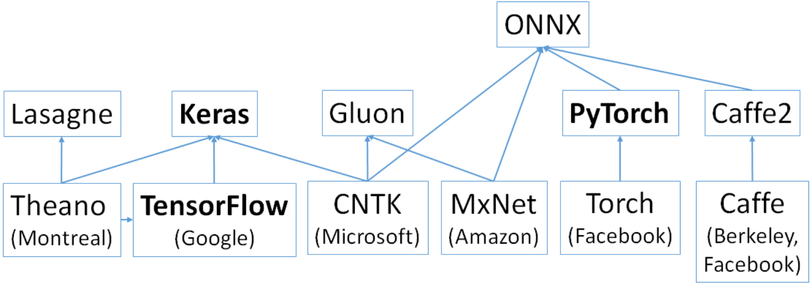
Such functions are provided by deep learning frameworks/platforms (including libraries, packages, toolkits)!

Deep learning frameworks/platforms

In practice, how do we get a real deep learning system?

- construct the structure of a neural network
- minimize the loss function with training dataset

Such functions are provided by deep learning frameworks/platforms (including libraries, packages, toolkits)!



It is simple to construct a network

Construct a two-layer network and run to get gradient

- PyTorch coding is more natural and easy to learn

TensorFlow

```python
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

with tf.device('/gpu:0'):
    x = tf.placeholder(tf.float32)
    y = tf.placeholder(tf.float32)
    z = tf.placeholder(tf.float32)

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

PyTorch

```python
import torch
from torch.autograd import Variable

N, D = 3, 4

x = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)
y = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)
z = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()

print(x.grad.data)
print(y.grad.data)
print(z.grad.data)
```

code from Stanford CS231n Lecture 8

It is simple to train a network

Construct a two-layer network and train it over 50 iterations

- Keras is as simple as PyTorch, but less flexible

Keras

```
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.optimizers import SGD

N, D, H = 64, 1000, 100

model = Sequential()
model.add(Dense(input_dim=D, output_dim=H))
model.add(Activation('relu'))
model.add(Dense(input_dim=H, output_dim=D))

optimizer = SGD(lr=1e0)
model.compile(loss='mean_squared_error',
              optimizer=optimizer)

x = np.random.randn(N, D)
y = np.random.randn(N, D)
history = model.fit(x, y, nb_epoch=50,
                    batch_size=N, verbose=0)
```

PyTorch

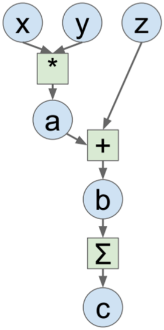
```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
          torch.nn.Linear(D_in, H),
          torch.nn.ReLU(),
          torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()

    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
```

Computational graph - TensorFlow

Static computational graph

- define-then-run: define graph (function) once, then compute
- more efficient, e.g., save memory

Computational Graphs



Create forward computational graph

TensorFlow

```python
# Basic computational graph
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

a = x * y
b = a + z
c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

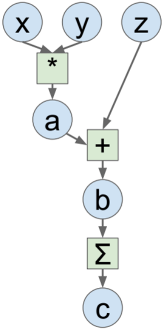
code and figure from Stanford CS231n Lecture 8

Computational graph - PyTorch

Dynamic computational graph

- define-by-run: construct graph (function) when computing
- more flexible, e.g., handle flexible inputs and outputs
- easier to construct complex graph and debug code!

Computational Graphs



Forward pass looks just like numpy

PyTorch

```
import torch
from torch.autograd import Variable

N, D = 3, 4

x = Variable(torch.randn(N, D),
             requires_grad=True)
y = Variable(torch.randn(N, D),
             requires_grad=True)
z = Variable(torch.randn(N, D),
             requires_grad=True)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()

print(x.grad.data)
print(y.grad.data)
print(z.grad.data)
```

TensorFlow vs PyTorch: more

- Both provide gradient computation with auto-differentiation
- TensorFlow: easy product deployment; more online codes
- PyTorch: easy data loading, parallel processing
- For developing products: TensorFlow is winner!
- For doing research: PyTorch/Keras is winner

from https://awni.github.io/pytorch-tensorflow/

TensorFlow vs PyTorch: more

- Both provide gradient computation with auto-differentiation
- TensorFlow: easy product deployment; more online codes
- PyTorch: easy data loading, parallel processing
- For developing products: TensorFlow is winner!
- For doing research: PyTorch/Keras is winner

from https://awni.github.io/pytorch-tensorflow/

Summary

- Backpropagation: gradient descent with chain rule
- Mini-batch gradient descent and variants
- Hyper-parameter tuning: validation, searching
- deep learning frameworks: TensorFlow or PyTorch?

Further reading:

- Sections 8.1, 8.3, 8.5, in textbook "Deep learning", http://www.deeplearningbook.org/
- http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture8.pdf
- Luo et al., Adaptive gradient methods with dynamic bound of learning rate. To appear in ICLR, 2019.

About paper summary

Where to find good papers:

- http://www.arxiv-sanity.com
- International Conference on Machine Learning (ICML)
- Neural Information Processing Systems (NIPS)
- International Conference on Learning Representations (ICLR)
- Association for the Advancement of Artificial Intelligence (AAAI)
- International Conference on Computer Vision and Pattern Recognition (CVPR)
- International Conference on Computer Vision (ICCV)
- European Conference on Computer Vision (ECCV)

Suggestion:

- Team members discuss together what to write
- One/two to write, the other two/one to criticize

About paper summary

Where to find good papers:

- http://www.arxiv-sanity.com
- International Conference on Machine Learning (ICML)
- Neural Information Processing Systems (NIPS)
- International Conference on Learning Representations (ICLR)
- Association for the Advancement of Artificial Intelligence (AAAI)
- International Conference on Computer Vision and Pattern Recognition (CVPR)
- International Conference on Computer Vision (ICCV)
- European Conference on Computer Vision (ECCV)

Suggestion:

- Team members discuss together what to write
- One/two to write, the other two/one to criticize

About the course project

Contests/challenges can be from:

- https://grand-challenge.org/challenges/ (those in 2018/2019)
- http://cvpr2019.thecvf.com/program/workshops
- https://www.kaggle.com/competitions
- http://rrc.cvc.uab.es/

Note:

- Choose those in 2018/2019
- Some deadlines are soon
- Need not choose active contests/challenges