FIT3181 Deep Learning

Week 03: Stochastic Gradient Descent and Optimization for Deep Learning

Lecturer: Lim Chern Hong

Email: lim.chernhong@monash.edu

Department of Data Science and AI
Faculty of Information Technology, Monash University, Australia

Outline

- Revision of calculus.
- Computational graph and forward/backward propagations.
- Gradient descent and stochastic gradient descent.
- Backpropagation in feed-forward neural networks.
- Optimizers for deep learning.

- **Further reading recommendations**
  - Deep Learning – Chapter 8
  - Dive into Deep Learning – Chapter 11 (https://d2l.ai/chapter_optimization/index.html)
  - Ruder’s blog: https://ruder.io/optimizing-gradient-descent/index.html

A small detour to calculus

- Calculus = **mathematics of change** (very important for deep learning)
- Properties of derivative:
  - $f'(x) = \nabla f(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$
  - $(uv)' = u'v + uv'$
  - $\left(\frac{u}{v}\right)' = \frac{u'v - uv'}{v^2}$
  - $(e^u)' = u'e^u$
  - $(\log u)' = \frac{u'}{u}$
- Multi-variate function $f: \mathbb{R}^n \to \mathbb{R}$ with $y = f(x) = f(x_1, \ldots, x_n)$.
  - Gradient/derivative: $\frac{\partial f}{\partial x}(a) = \nabla_x f(a) = [\nabla_{x_1} f(a), \nabla_{x_2} f(a), \ldots, \nabla_{x_n} f(a)]$.
- Chain rule 🔗 :
  - $\frac{\partial u}{\partial x} = \frac{\partial u}{\partial v} \times \frac{\partial v}{\partial x}$

Example

- Consider the function $f(x, y, z) = \log(\exp(x) + \exp(y) + \exp(z))$. What are $f_x' = \nabla_x f$, $f_y' = \nabla_y f$, and $f_z' = \nabla_z f$?

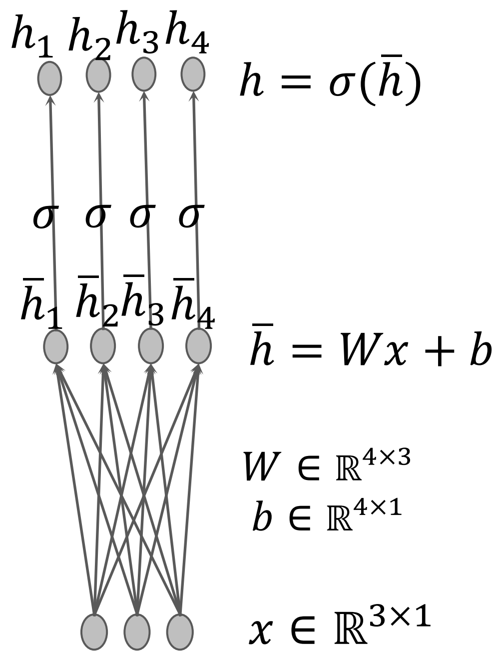
- $f(x, y, z) = \log(u)$ with $u = \exp(x) + \exp(y) + \exp(z)$.
- $f_x' = \frac{u_x'}{u} = \frac{\exp(x)}{\exp(x)+\exp(y)+\exp(z)}$.
- $f_y' = \frac{u_y'}{u} = \frac{\exp(y)}{\exp(x)+\exp(y)+\exp(z)}$.
- $f_z' = \frac{u_z'}{u} = \frac{\exp(z)}{\exp(x)+\exp(y)+\exp(z)}$.
- $\nabla f = [f_x', f_y', f_z'] = softmax([x, y, z])$.

Example (Forum discussion)

- $\bar{h} = Wx + b$ and $h = \sigma(\bar{h})$
  - $h = \sigma(Wx + b)$
  - σ is the **activation function**
- $\frac{\partial h}{\partial x} = \frac{\partial h}{\partial \bar{h}} \times \frac{\partial \bar{h}}{\partial x} = diag\left(\sigma'(\bar{h})\right) W$

- $$\frac{\partial h}{\partial \bar{h}} = \begin{bmatrix} \frac{\partial h_1}{\partial \bar{h}_1} & \frac{\partial h_1}{\partial \bar{h}_2} & \frac{\partial h_1}{\partial \bar{h}_3} & \frac{\partial h_1}{\partial \bar{h}_4} \\ \frac{\partial h_2}{\partial \bar{h}_1} & \frac{\partial h_2}{\partial \bar{h}_2} & \frac{\partial h_2}{\partial \bar{h}_3} & \frac{\partial h_2}{\partial \bar{h}_4} \\ \frac{\partial h_3}{\partial \bar{h}_1} & \frac{\partial h_3}{\partial \bar{h}_2} & \frac{\partial h_3}{\partial \bar{h}_3} & \frac{\partial h_3}{\partial \bar{h}_4} \\ \frac{\partial h_4}{\partial \bar{h}_1} & \frac{\partial h_4}{\partial \bar{h}_2} & \frac{\partial h_4}{\partial \bar{h}_3} & \frac{\partial h_4}{\partial \bar{h}_4} \end{bmatrix} = \begin{bmatrix} \sigma'(\bar{h}_1) & 0 & 0 & 0 \\ 0 & \sigma'(\bar{h}_2) & 0 & 0 \\ 0 & 0 & \sigma'(\bar{h}_3) & 0 \\ 0 & 0 & 0 & \sigma'(\bar{h}_4) \end{bmatrix} = diag(\sigma'(\bar{h}))$$

- $\frac{\partial \bar{h}}{\partial x} = W$



How to code with numpy

- $\bar{h} = Wx + b$ and $h = sigmoid(\bar{h})$
  - $h = sigmoid(Wx + b)$
  - $\sigma = sigmoid$ is the activation function
- $\frac{\partial h}{\partial x} = \frac{\partial h}{\partial \bar{h}} \times \frac{\partial \bar{h}}{\partial x} = diag\left(\sigma'(\bar{h})\right)W = diag(\sigma(\bar{h})[1-\sigma(\bar{h})])W = diag(h(1-h))W$

```
import numpy as np
```

```
x = np.array([[1],[-1],[1]])
x
```

```
array([[ 1],
       [-1],
       [ 1]])
```

```
W = np.array([[-1,1,1,],[1,-1,1,],[1,1,-1],[-1,-1,-1]])
W
```

```
array([[-1,  1,  1],
       [ 1, -1,  1],
       [ 1,  1, -1],
       [-1, -1, -1]])
```

```
b = np.ones((4,1))
b
```

```
array([[1.],
       [1.],
       [1.],
       [1.]])
```

Declare W, x, b

```
h_bar = W.dot(x) +b
h_bar
```

```
array([[0.],
       [4.],
       [0.],
       [0.]])
```

```
def sigmoid(x):
    return 1.0/(1+ np.exp(-x))
```

```
h = sigmoid(h_bar)
h
```

```
array([[0.5       ],
       [0.98201379],
       [0.5       ],
       [0.5       ]])
```

Forward propagation

```
v= h*(1-h)
v
```

```
array([[0.25      ],
       [0.01766271],
       [0.25      ],
       [0.25      ]])
```

```
D = np.diag(v[:,0])
D
```

```
array([[0.25      , 0.        , 0.        , 0.        ],
       [0.        , 0.01766271, 0.        , 0.        ],
       [0.        , 0.        , 0.25      , 0.        ],
       [0.        , 0.        , 0.        , 0.25      ]])
```

```
derivative = D.dot(W)
derivative
```

```
array([[-0.25      ,  0.25      ,  0.25      ],
       [ 0.01766271, -0.01766271,  0.01766271],
       [ 0.25      ,  0.25      , -0.25      ],
       [-0.25      , -0.25      , -0.25      ]])
```

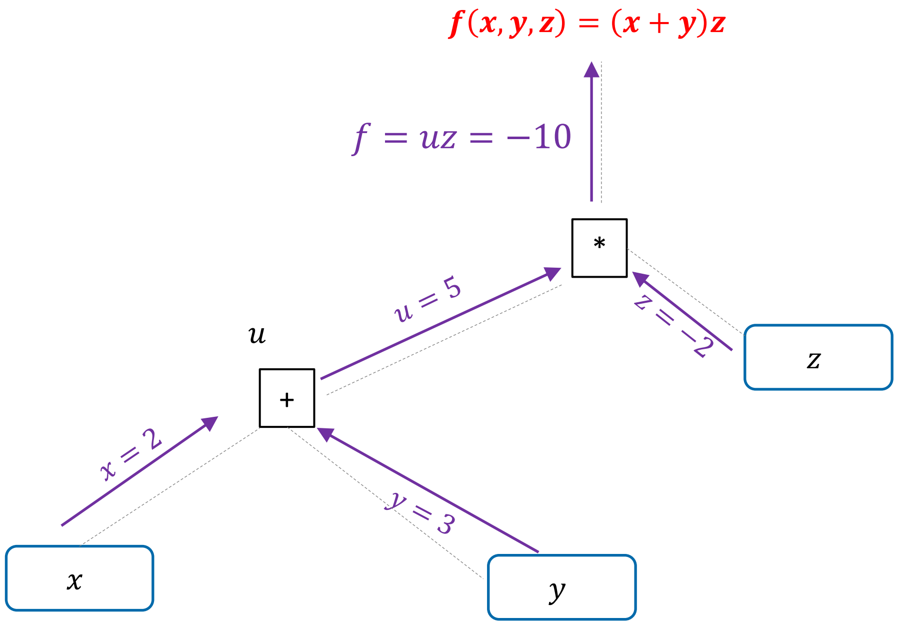
Backward propagation

Computational graph

Computational graph (used in TensorFlow)

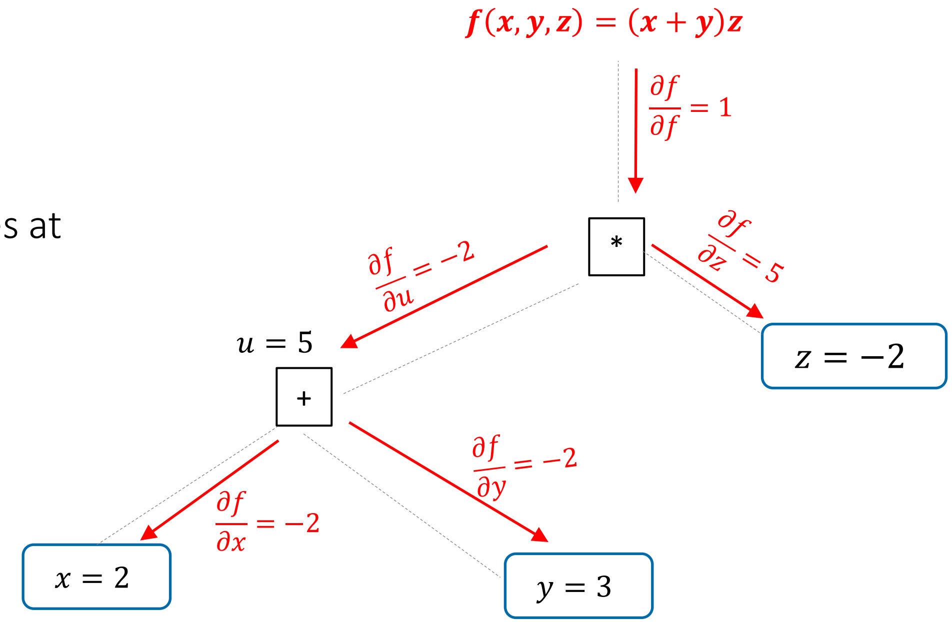
Problem: $f(x, y, z) = (x + y)z$
What are its partial derivatives, evaluated at $x = 2, y = 3, z = -2$?

Step 1:
a) construct computational graph
b) forward propagation
c) record value at each node



Reverse Auto-Diff (used in TensorFlow)

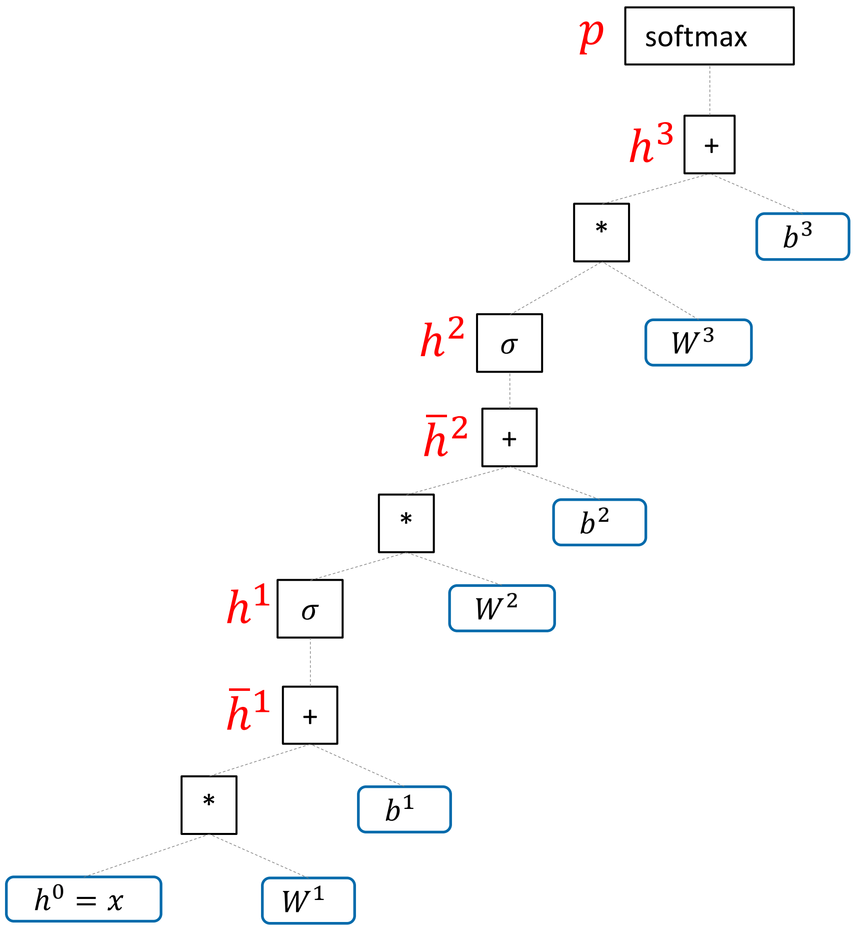
Step 2:
- a) traverse backward
- b) apply chain rule
- c) record differential values at each node

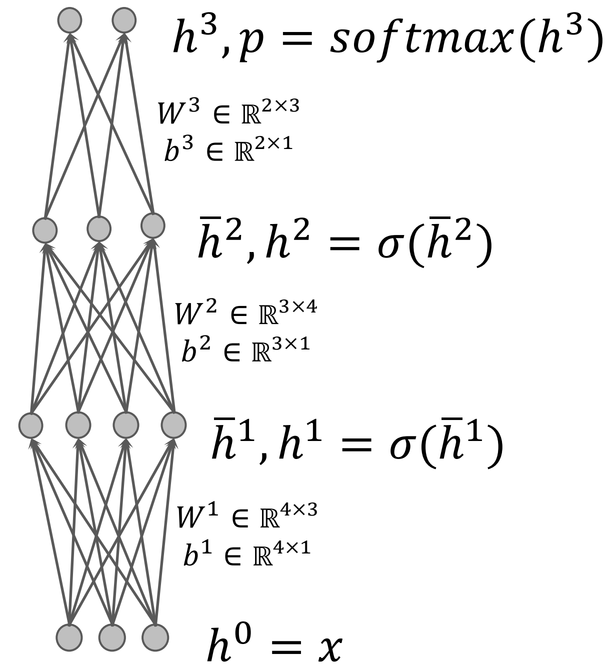


$$\frac{\partial f}{\partial f} = 1 \qquad \frac{\partial f}{\partial u} = \frac{\partial f}{\partial f}\frac{\partial f}{\partial u} = 1 * z = -2 \qquad \frac{\partial f}{\partial x} = \frac{\partial f}{\partial u}\frac{\partial u}{\partial x} = -2 * 1 = -2$$

$$\frac{\partial f}{\partial z} = \frac{\partial f}{\partial f}\frac{\partial f}{\partial z} = 1 * u = 5 \qquad \frac{\partial f}{\partial y} = \frac{\partial f}{\partial u}\frac{\partial u}{\partial y} = -2 * 1 = -2$$

Computational graph of feedforward nets (Forum discussion)





$h^0(x) = x$

for $k = 1$ **to** 2 **do**

$\quad \bar{h}^k = W^k h^{k-1}(x) + b^k$ //linear operation

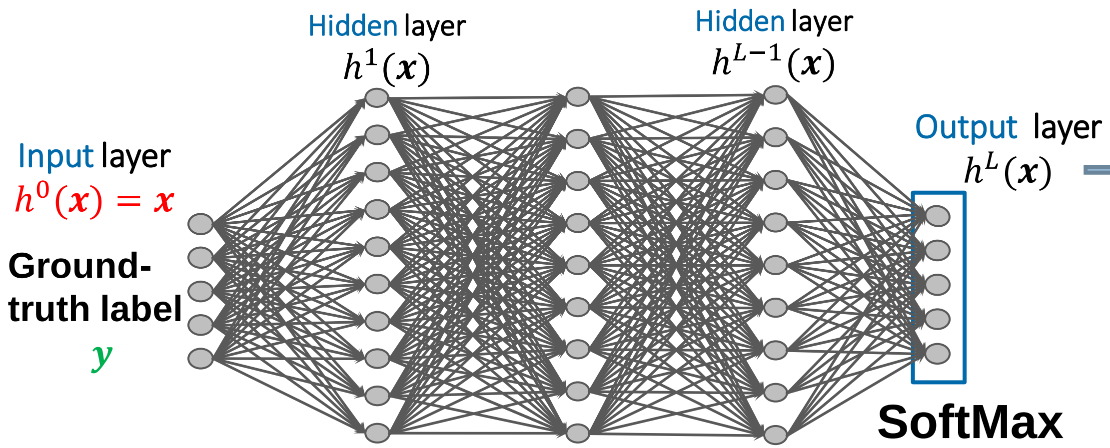
$\quad h^k(x) = \sigma\left(\bar{h}^k(x)\right)$ //activation

$h^3(x) = W^3 h^2(x) + b^2$

$p(x) = softmax(h^3(x))$ //prediction probabilities

Gradient descent and stochastic gradient descent

Recall optimization problem in deep learning (Forum discussion)



$\Longrightarrow p(x) = softmax(h^L(x))$

$\hat{y} = \mathrm{argmax}_{1 \le m \le M} p_m(x)$

$l(y, \hat{y}) = CE\left(1_y, p(x)\right) = -\log p_y(x)$

Training set $\longrightarrow$ **Loss function**

$D = \{(x_1, y_1), \dots (x_N, y_N)\}$

$$L(D;\theta) \coloneqq \frac{1}{N}\sum_{i=1}^{N} CE\left(1_{y_i}, p(x_i)\right) = -\frac{1}{N}\sum_{i=1}^{N} \log p_{y_i}(x_i)$$

- How to **solve** the **optimization problem** efficiently ($\theta \coloneqq \left\{\left(W^l, b^l\right)\right\}_{l=1}^{L}$)?
  - $\min_\theta L(D;\theta) \coloneqq -\frac{1}{N}\sum_{i=1}^{N} \log p_{y_i}(x_i) = -\frac{1}{N}\sum_{i=1}^{N} \log \frac{\exp\{h_{y_i}^L(x_i)\}}{\sum_{m=1}^{M} \exp\{h_m^L(x_i)\}}$
  - Generalize: $\min_\theta J(\theta) \coloneqq \frac{1}{N}\sum_{i=1}^{N} l(f(x_i;\theta), y_i)$

Optimization problem in ML and DL

- Most of optimization problems (OP) in machine learning (deep learning) has the following form:

$$\min_\theta J(\theta) = \underbrace{\Omega(\theta)} + \underbrace{\frac{1}{N}\sum_{i=1}^{N} l(y_i, f(x_i;\theta))}$$

Regularization term

- $\Omega(\theta) = \lambda \sum_k \sum_{i,j} \left(W_{i,j}^{\mathbf{k}}\right)^2 = \lambda \sum_k \left\|\boldsymbol{W}^{\mathbf{k}}\right\|_F^2$
- Encourage simple models
- Avoid overfitting

Empirical loss

- Work well on training set

- **Occam's Razor principle**: prefer simplest model that can well predict data.

How to efficiently solve this optimization problem?

N is the **training size** and might be very big (e.g., $N \approx 10^6$)

First-order iterative methods (gradient descent, steepest descent)

Use the **gradient** (first derivative) $\mathrm{g} = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ to update parameters

Second-order iterative methods (Newton and quasi Newton methods)

Use the **Hessian** matrix (second derivative) $\mathrm{H} = \nabla_{\boldsymbol{\theta}}^2 J(\boldsymbol{\theta})$ to update parameters

Gradient and Hessian matrix

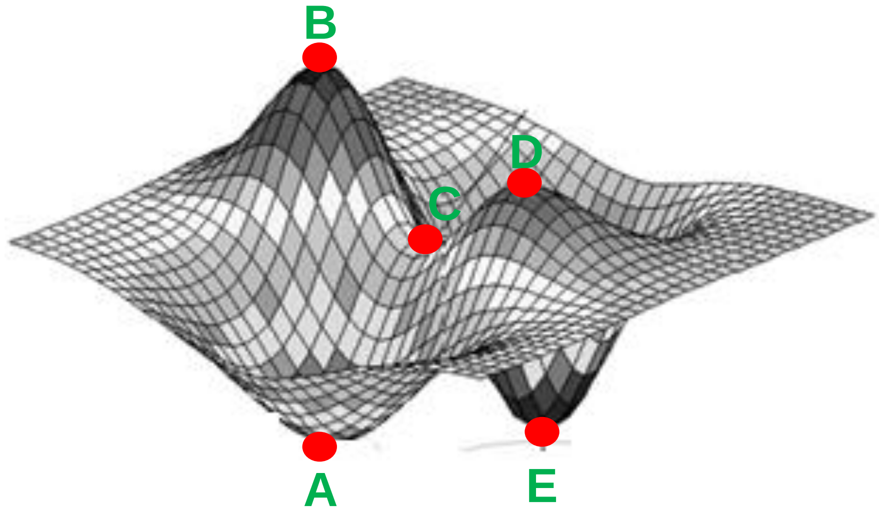
- Given an **objective function** $J(\theta)$ with $\theta = [\theta_1, \theta_2, \dots, \theta_P]$
  - For DL models
    - θ includes **weight matrices**, **filters**, and **biases** which are trainable model parameters.
    - P is the number of **trainable parameters** (P could be 20×10^6).
    - $J(\theta)$ is the loss function over a training set.

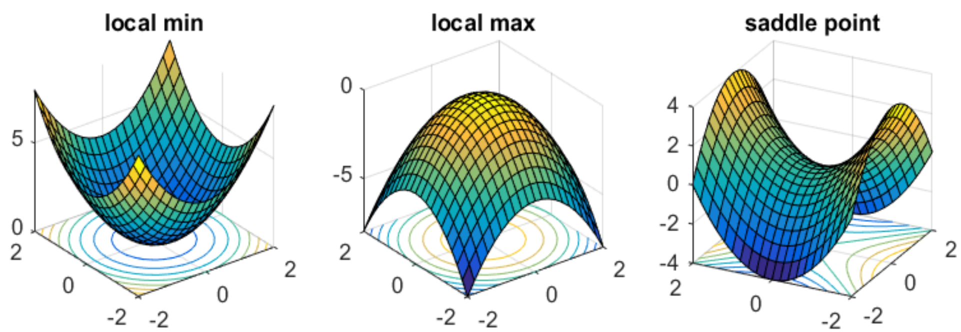
- Gradient $\mathrm{g} = \nabla J(\theta)$ is the **first order derivative** and defined as
  - $\nabla J(\theta) = g = \begin{bmatrix} \frac{\partial J}{\partial \theta_1}(\theta) \\ \dots \dots \dots . \\ \frac{\partial J}{\partial \theta_P}(\theta) \end{bmatrix}$

- Hessian matrix $H(\theta)$ is the **second order derivative** $\nabla^2 J(\theta)$ and defined as
  - $\nabla^2 J(\theta) = H(\theta) = \begin{bmatrix} \frac{\partial^2 J}{\partial \theta_1 \partial \theta_1}(\theta) & \dots & \dots & \frac{\partial^2 J}{\partial \theta_1 \partial \theta_j}(\theta) & \dots & \frac{\partial^2 J}{\partial \theta_1 \partial \theta_P}(\theta) \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \frac{\partial^2 J}{\partial \theta_i \partial \theta_1}(\theta) & \dots & \dots & \frac{\partial^2 J}{\partial \theta_i \partial \theta_j}(\theta) & \dots & \frac{\partial^2 J}{\partial \theta_i \partial \theta_P}(\theta) \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \frac{\partial^2 J}{\partial \theta_P \partial \theta_1}(\theta) & \dots & \dots & \frac{\partial^2 J}{\partial \theta_P \partial \theta_j}(\theta) & \dots & \frac{\partial^2 J}{\partial \theta_P \partial \theta_P}(\theta) \end{bmatrix}$

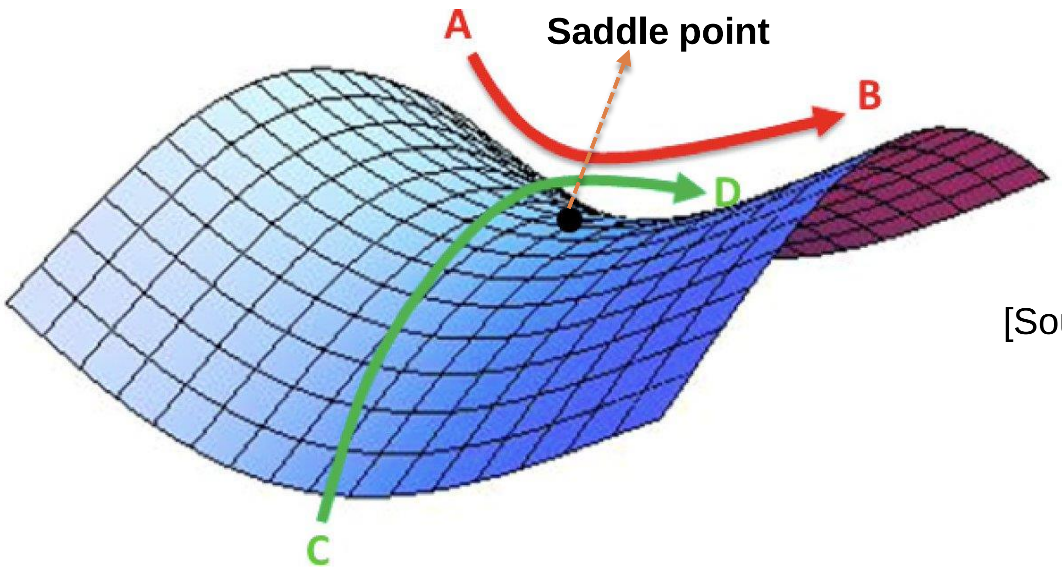
Local minima-maxima and saddle point

- Given an **objective function** $J(\theta)$ with $\theta = [\theta_1, \theta_2, \dots, \theta_P]$
  - θ is said to be a **critical point** if $\nabla J(\theta) = \mathbf{0}$ (vector $\mathbf{0}$)
- Let us denote the **set of eigenvalues** of Hessian matrix $\nabla^2 J(\theta) = H(\theta)$ by
  - $\lambda_1 \le \lambda_2 \le \cdots \le \lambda_P$
- **Local minima**
  - $\nabla J(\theta) = \mathbf{0}$ and $\nabla^2 J(\theta) = H(\theta) \succ 0$ (positive semi-definite matrix)
  - $0 \le \lambda_1 \le \lambda_2 \le \cdots \le \lambda_P$
- **Local maxima**
  - $\nabla J(\theta) = \mathbf{0}$ and $\nabla^2 J(\theta) = H(\theta) \prec 0$ (negative semi-definite matrix)
  - $\lambda_1 \le \lambda_2 \le \cdots \le \lambda_P \le 0$
- **Saddle point**
  - $\nabla J(\theta) = \mathbf{0}$ and $\nabla^2 J(\theta) = H(\theta) \prec 0$ (indefinite matrix)
  - $\lambda_1 \le \lambda_2 \le \cdots < 0 < \cdots \le \lambda_P$





More on saddle point (Forum discussion)



[Source: Internet]

$$f(x) = f(x_1, x_2) = x_1^2 - x_2^2$$

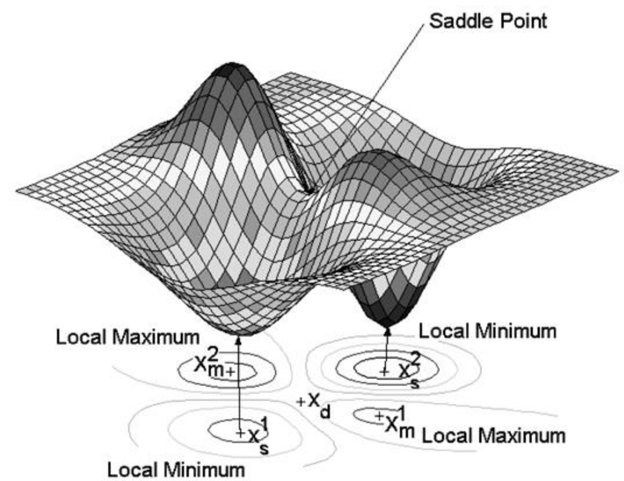
Gradient $g = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2x_1 \\ -2x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ → a **critical point** $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$.

Hessian matrix is $H = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & -2 \end{bmatrix}$

Two eigenvalues $\lambda_1 = -2 < 0 < 2 = \lambda_2$ → $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$ is a **saddle point**.

Numbers of local minima vs saddle points

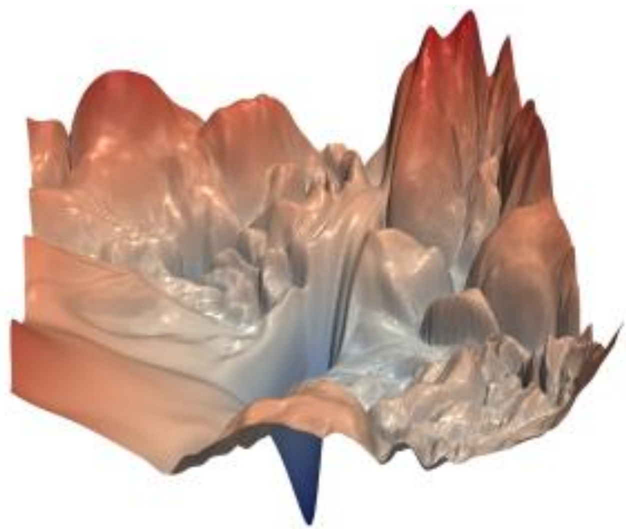
- We assume to pick randomly a training set
  - The Hessian matrix $H(\theta)$ is a random matrix with **random eigenvalues** $\lambda_1, \lambda_2, \dots, \lambda_P$
  - We assume that $\mathbb{P}(\lambda_1 \geq 0) = \mathbb{P}(\lambda_2 \geq 0) = \cdots = \mathbb{P}(\lambda_P \geq 0) = 0.5$



- Therefore, we have
  - $\mathbb{P}(minima) = \mathbb{P}(\lambda_1 \geq 0)\mathbb{P}(\lambda_2 \geq 0) \dots \mathbb{P}(\lambda_P \geq 0) = 0.5^P$
  - $\mathbb{P}(maxima) = \mathbb{P}(\lambda_1 \leq 0)\mathbb{P}(\lambda_2 \leq 0) \dots \mathbb{P}(\lambda_P \leq 0) = 0.5^P$
  - $\mathbb{P}(saddle\ point) = 1 - \mathbb{P}(minima) - \mathbb{P}(maxima) = 1 - 0.5^{P-1}$

- The ratio of **#local minima/maxima against #saddle points**
  - **#local-minima:#local-maxima:#saddle-point=**$1:1:(2^P - 2)$
  - Number of **saddle points** is even **exponentially much more** than that of **local minima/maxima**

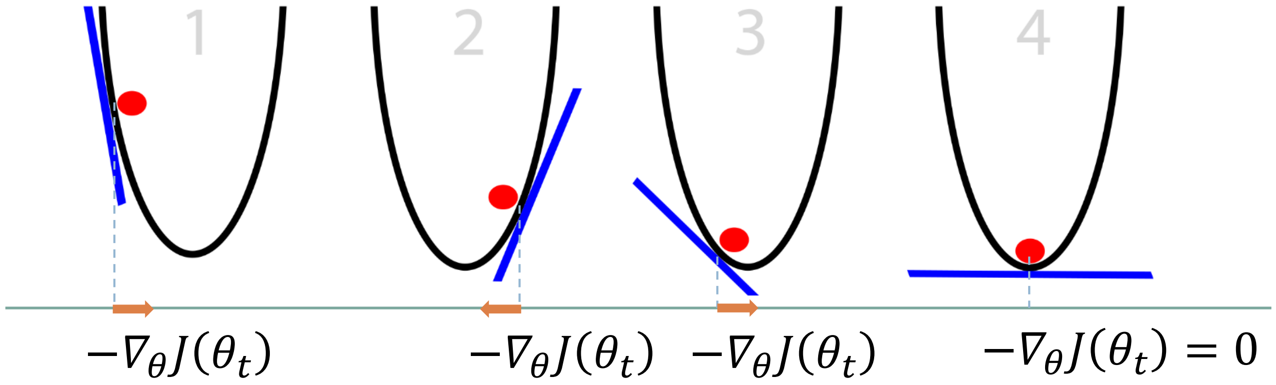
The loss surface of DL optimization problem



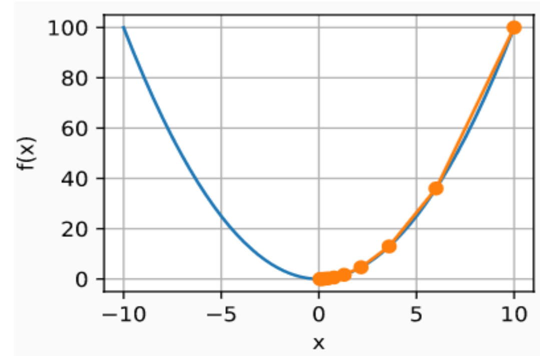
Loss surface of a ResNet without skip connection [Hao Li et al., NeurIPS 2017]

- The **optimization problem** in **deep learning:**
  - $\min_{\theta} J(\theta) \coloneqq L(D;\theta) \coloneqq \frac{1}{N}\sum_{i=1}^{N} \ell(f(x_i;\theta), y_i) = -\frac{1}{N}\sum_{n=1}^{N} \log \frac{\exp\{h_{y_i}^{L}(x_i)\}}{\sum_{m=1}^{M} \exp\{h_m^{L}(x_i)\}}$
- A very **complex** and **complicated** objective function
  - Highly **non-linear** and **non-convex** function
  - The **loss surface** is very **complex**
  - Many local minima points, but the number of saddle points is even **exponentially much more**

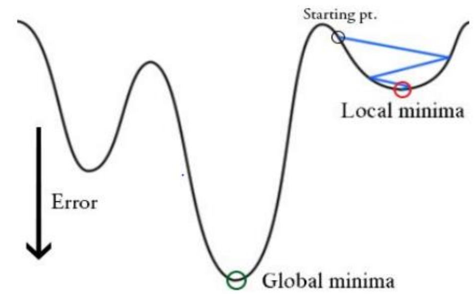
Gradient descend (Forum discussion)



- We need to solve
  - $\min_{\theta} J(\theta)$
- Follow to **the opposite side** of the current gradient
  - $\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta_t)$ where $\boldsymbol{\eta} > \mathbf{0}$ is the **learning rate**.
- Guarantee to converge to a **global minima** if $J(.)$ is **convex**.
- Get stuck in a **local minima** or **saddle points** if $J(.)$ is non-convex.

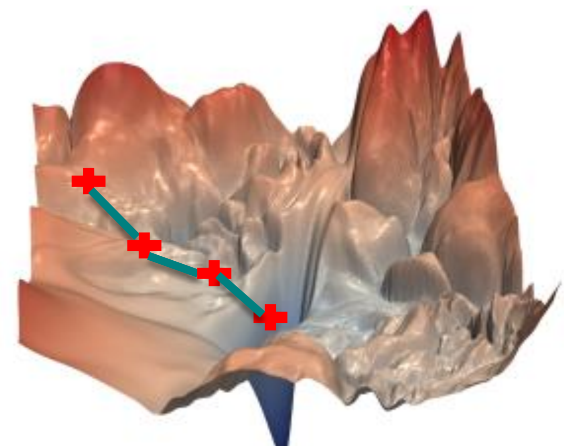


Convex case



(Source: www.cs.ubc.ca)

Non-convex case



DL case: easy to get stuck in saddle points

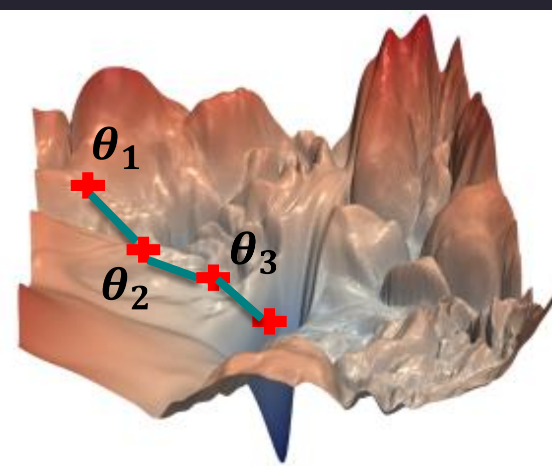
Gradient descend

Algorithm

- **Input**: objective function $\boldsymbol{J}(\boldsymbol{\theta})$
- **Output**: optimal solution $\boldsymbol{\theta}^*$

1. Initialize parameters θ_0 randomly $\sim N(0, \sigma^2)$.
2. for t=1 to T
3. Compute gradients $\nabla_\theta J(\theta_t) = \frac{\partial J}{\partial \theta}(\theta_t)$
4. Update $\theta_{t+1} = \theta_t - \eta_t \nabla_\theta J(\theta_t)$
5. Return $\theta^* = \theta_{T+1}$

```python
import tensorflow as tf

weights = tf.Variable([tf.random.normal()])

while True:    # loop forever
    with tf.GradientTape() as g:
        loss = compute_loss(weights)
        gradient = g.gradient(loss, weights)

    weights = weights - lr * gradient
```



Define $f(x) = x^2$ and solve $\min_x f(x)$

```
%matplotlib inline
import numpy as np
import tensorflow as tf
from d2l import tensorflow as d2l


def f(x):  # Objective function
    return x**2

def f_grad(x):  # Gradient (derivative) of the objective function
    return 2 * x
```

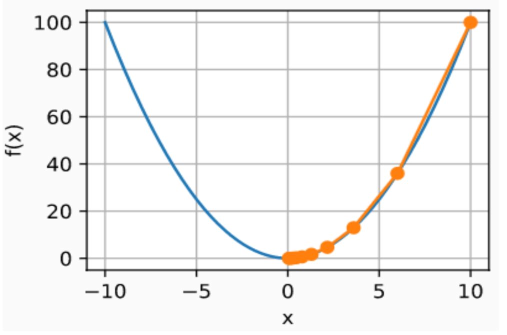
Gradient descent

```
def gd(eta, f_grad):
    x = 10.0
    results = [x]
    for i in range(10):
        x -= eta * f_grad(x)
        results.append(float(x))
    print(f'epoch 10, x: {x:f}')
    return results

results = gd(0.2, f_grad)
```

```
epoch 10, x: 0.060466
```

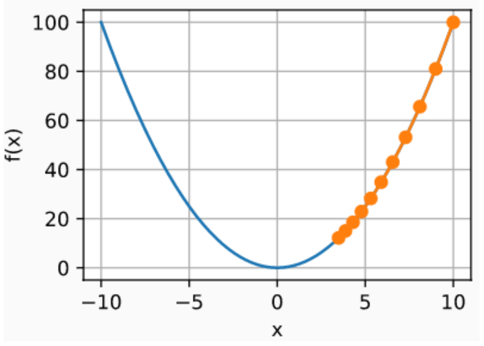
Show trace of gradient descent

```
def show_trace(results, f):
    n = max(abs(min(results)), abs(max(results)))
    f_line = tf.range(-n, n, 0.01)
    d2l.set_figsize()
    d2l.plot([f_line, results],
             [[f(x) for x in f_line], [f(x) for x in results]], 'x', 'f(x)',
             fmts=['-', '-o'])

show_trace(results, f)
```




```
show_trace(gd(0.05, f_grad), f)
```

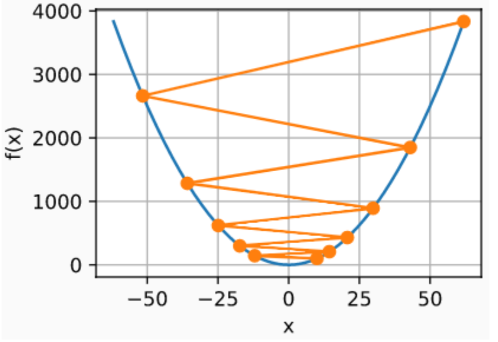
```
epoch 10, x: 3.486784
```



Good learning rate η

```
show_trace(gd(1.1, f_grad), f)
```

```
epoch 10, x: 61.917364
```



Too high rate η

[Source: Dive into Deep Learning]

Gradient descent with TF

Gradient descent for deep learning

- For **training deep nets**, we need to solve
  - $\min_{\theta} L(D;\theta) := \frac{1}{N}\sum_{i=1}^{N} l(x_i, y_i; \theta)$

  where $l(x_i, y_i; \theta) = -\log p(y = y_i | x_i) = -\log \frac{\exp\{h_{y_i}^L(x_i)\}}{\sum_{m=1}^{M} \exp\{h_m^L(x_i)\}}$ is the loss incurred by (x_i, y_i).
- Gradient descent update
  - $\theta_{t+1} = \theta_t - \eta \nabla_\theta L(D;\theta_t) = \theta_t - \frac{\eta}{N}\sum_{i=1}^{N} \nabla_\theta\, l(x_i, y_i; \theta_t)$ where $\boldsymbol{\eta} > \mathbf{0}$ is a learning rate.
  - To compute the gradient $\nabla_\theta L(D;\theta_t) = \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta\, l(x_i, y_i; \theta_t)$, we need to go through **all data points** in D → the **computational cost** is $O(N)$.
- This is very **computationally expensive** for big datasets ($N \approx 10^6$).
- How to **estimate the gradient** $\nabla_\theta L(D;\theta_t)$ more efficiently?

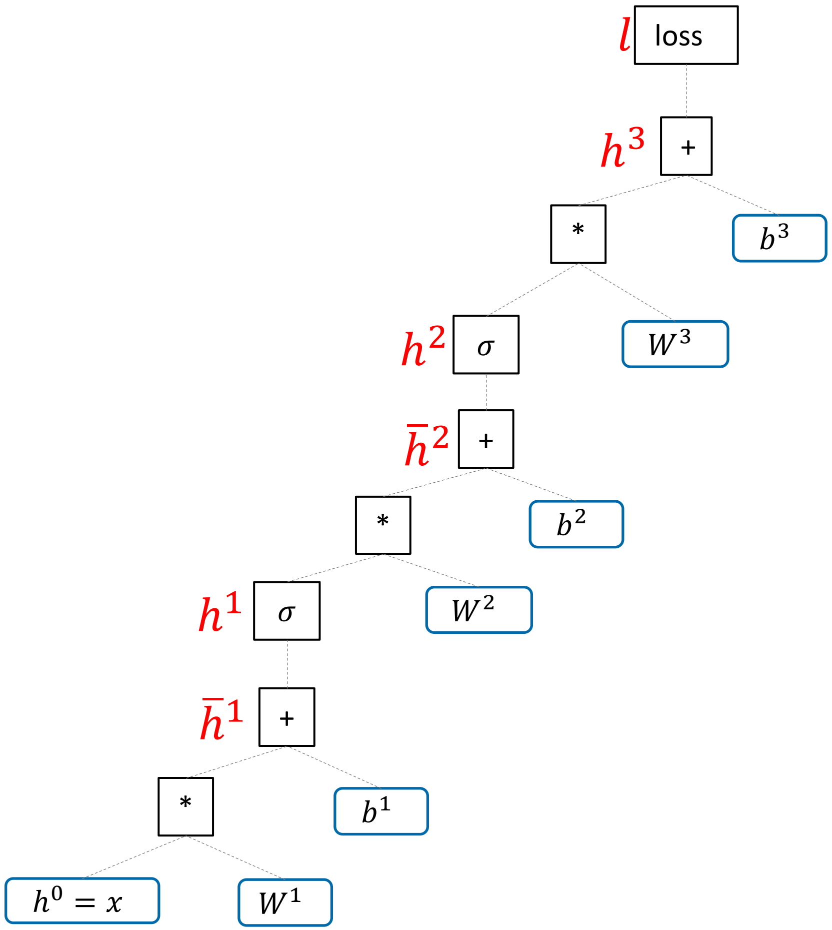
Stochastic gradient descent (Forum discussion)

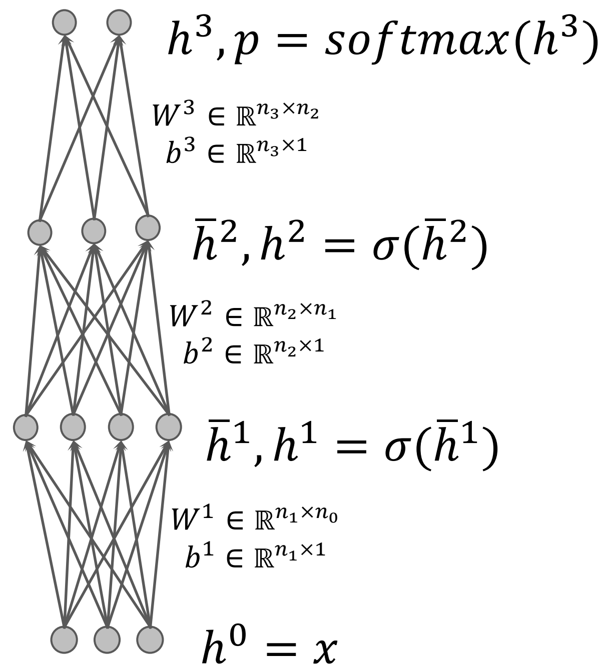
- The **optimization problem** in **deep learning** has the form
  - $\min_{\theta} L(D;\theta) := \frac{1}{N}\sum_{i=1}^{N} l(x_i, y_i; \theta)$
- Evaluation of the **full gradient** is **expensive**. We want to just **estimate** this gradient
  - Sample a mini-batch $\boldsymbol{i_1, i_2, \dots, i_b \sim Uni(\{1, 2, \dots, N\})}$ where b is the mini-batch (batch) size.
    - The batch size is usually $32, 64, 128, 256$, and so on.
  - Construct $\tilde{L}(\theta) := \frac{1}{b}\sum_{k=1}^{b} l(x_{i_k}, y_{i_k}; \theta)$ as the average loss of those in the current batch.
  - $E[\nabla_\theta \tilde{L}(\theta_t)] = \nabla_\theta L(D;\theta_t)$
    - $\nabla_\theta \tilde{L}(\theta_t) = \frac{1}{b}\sum_{k=1}^{b} \nabla_\theta l(x_{i_k}, y_{i_k}; \theta_t)$ is **unbiased** estimation of $\nabla_\theta L(D;\theta_t)$
    - $O(b)$ compares to $O(N)$.
- The update rule of SGD
  - $\theta_{t+1} = \theta_t - \eta_t \nabla_\theta \tilde{L}(\theta_t)$ with learning rate $\boldsymbol{\eta_t \propto O(\frac{1}{t})}$
  - We use $\nabla_\theta \tilde{L}(\theta_t)$ as an **unbiased estimate** of the full gradient $\nabla_\theta L(D;\theta)$
  - How to compute $\nabla_\theta \tilde{L}(\theta_t)$ efficiently for **deep networks**?

Q&A
When? Why? How? WHERE? WHEN? What? How? Why? WHAT? Why? When?
WHAT? HOW? WHEN? When? WHERE?
When? WHERE? WHAT? Why?
When?
WHO? WHAT?
WHEN?
WHERE? What?
Why? HOW? Where? Where? WHERE? When? What? WHERE? WHERE? When? HOW?
Why? When? What? When?
WHEN? WHEN? HOW? When? When? HOW? WHAT? Why?

Back propagation in feed-forward neural networks

Back propagation



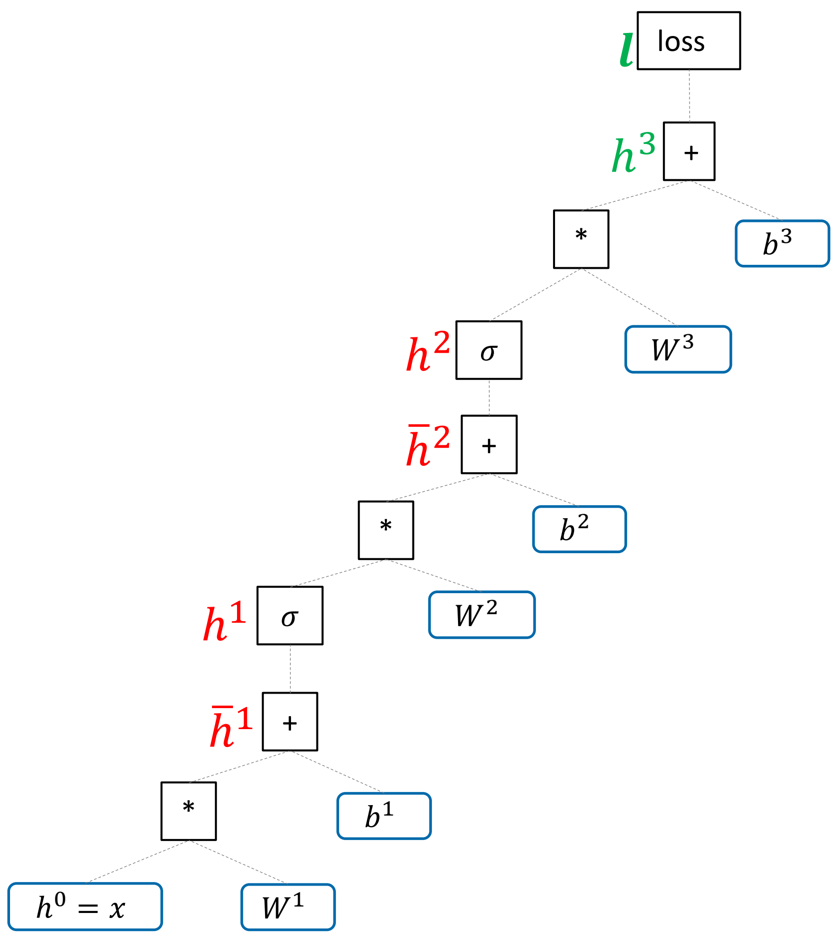


- Given a data point and label pair (x, y)
  - $l(x, y; \theta) = -\log \frac{\exp\{h_y^3(x)\}}{\sum_{m=1}^{M} \exp\{h_m^3(x)\}}$
- What are the derivatives?
  - $\nabla_{W^k} l(x, y; \theta)$ and $\nabla_{b^k} l(x, y; \theta)$ for $k = 1,2,3$?
  - Using **back propagation** to compute these derivatives conveniently.
- Update the model using SGD with the derivatives.

Back propagation

From loss to h^3

$$f(x,y,z) = \log(\exp(x) + \exp(y) + \exp(z))$$
$$\nabla f = [f'_x, f'_y, f'_z] = softmax([x,y,z])$$



- $l(x, y; \theta) = -\log \frac{\exp\{h_y^3(x)\}}{\sum_{m=1}^{M} \exp\{h_m^3(x)\}} =$
  $-h_y^3(x) + \log[\sum_{m=1}^{M} \exp\{h_m^3(x)\}] =$
  $-\sum_{m=1}^{M} \mathbf{1}_{m=y} h_m^3 + \log[\sum_{m=1}^{M} \exp\{h_m^3\}]$

where $\mathbf{1}_{m=y} = \mathbf{1}$ if $\boldsymbol{m} = \boldsymbol{y}$ and $\mathbf{0}$ otherwise.

- $\frac{\partial l}{\partial h_m^3} = -\mathbf{1}_{m=y} + \frac{exp\{h_m^3\}}{\sum_{k=1}^{M} exp\{h_k^3\}}$
- $\boldsymbol{g^3} = \frac{\partial l}{\partial h^3} = -\mathbf{1}_y + \boldsymbol{softmax}(\boldsymbol{h^3}) =$
  $= \boldsymbol{p^T} - \mathbf{1}_y$

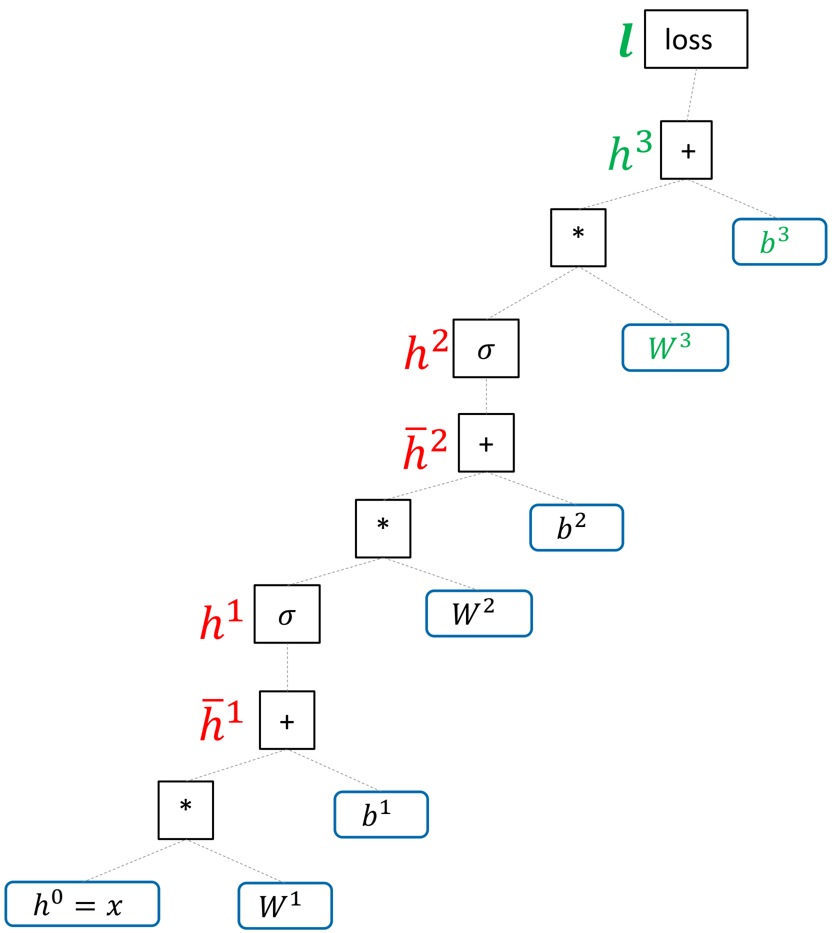
where $\mathbf{1}_y$ is the corresponding one-hot vector.

- g^3 has a shape $[1 \times n_3]$.

one hot vector of the true ground label, e.g. [0,0,1]

Back propagation

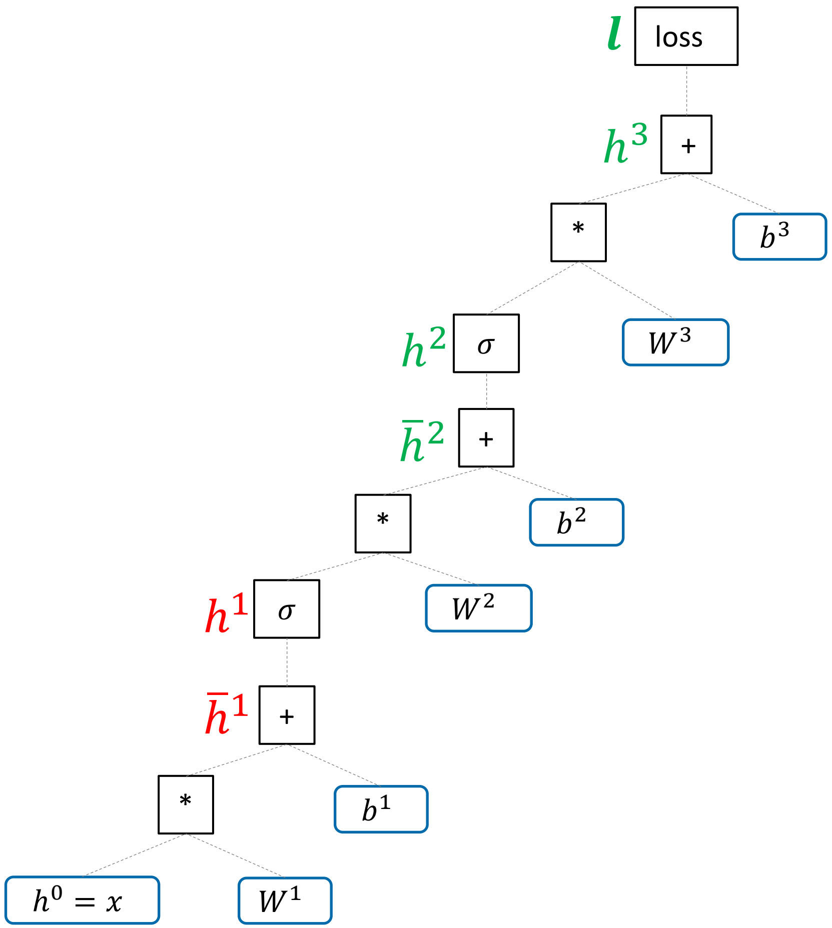
From loss to W^3, b^3



- $h^3 = W^3 h^2 + b^3$
- $$\frac{\partial l}{\partial W^3} = \frac{\partial l}{\partial h^3} \cdot \frac{\partial h^3}{\partial W^3} = \left(g^3\right)^T \left(h^2\right)^T$$
  - $[n_3 \times 1] \times [1 \times n_2] \rightarrow [n_3 \times n_2]$
- $$\frac{\partial l}{\partial b^3} = \frac{\partial l}{\partial h^3} \cdot \frac{\partial h^3}{\partial b^3} = g^3$$
  - $[1 \times n_3]$

Back propagation

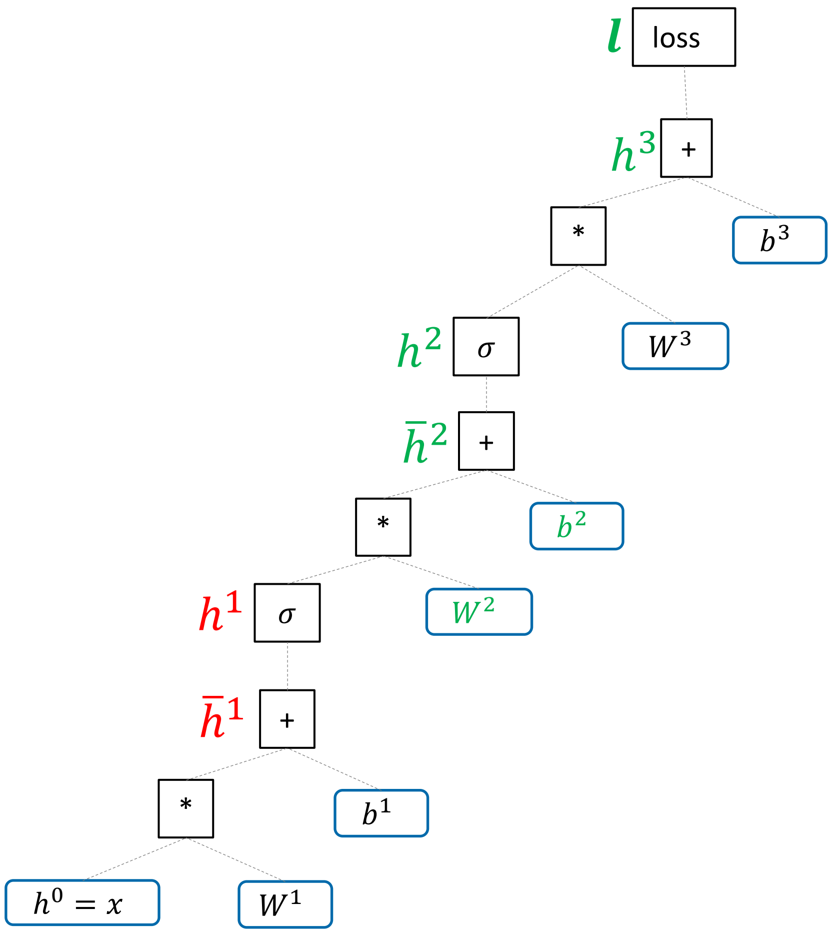
From loss to h^2 and $\bar{h}^2$



- $h^3 = W^3 h^2 + b^3$
- $\boldsymbol{g^2 = \frac{\partial l}{\partial h^2} = \frac{\partial l}{\partial h^3} \cdot \frac{\partial h^3}{\partial h^2} = g^3 W^3}$
  - $[1 \times n_3] \times [n_3 \times n_2] \rightarrow [1 \times n_2]$
- $\boldsymbol{h^2 = \sigma(\bar{h}^2)}$ (element-wise activation)
- $\boldsymbol{\frac{\partial h^2}{\partial \bar{h}^2} = diag(\sigma'(\bar{h}^2))}$
  - $\sigma'(\bar{h}^2)$ is **element-wise derivative** and $diag(u)$ is the **diagonal matrix** corresponding to the **vector $\boldsymbol{u}$** (the diagnose is u and others are zeros).
- $\boldsymbol{\bar{g}^2 = \frac{\partial l}{\partial \bar{h}^2} = \frac{\partial l}{\partial h^2} \cdot \frac{\partial h^2}{\partial \bar{h}^2} = g^2 diag(\sigma'(\bar{h}^2))}$
  - $[1 \times n_2] \times [n_2 \times n_2] \rightarrow [1 \times n_2]$

Back propagation

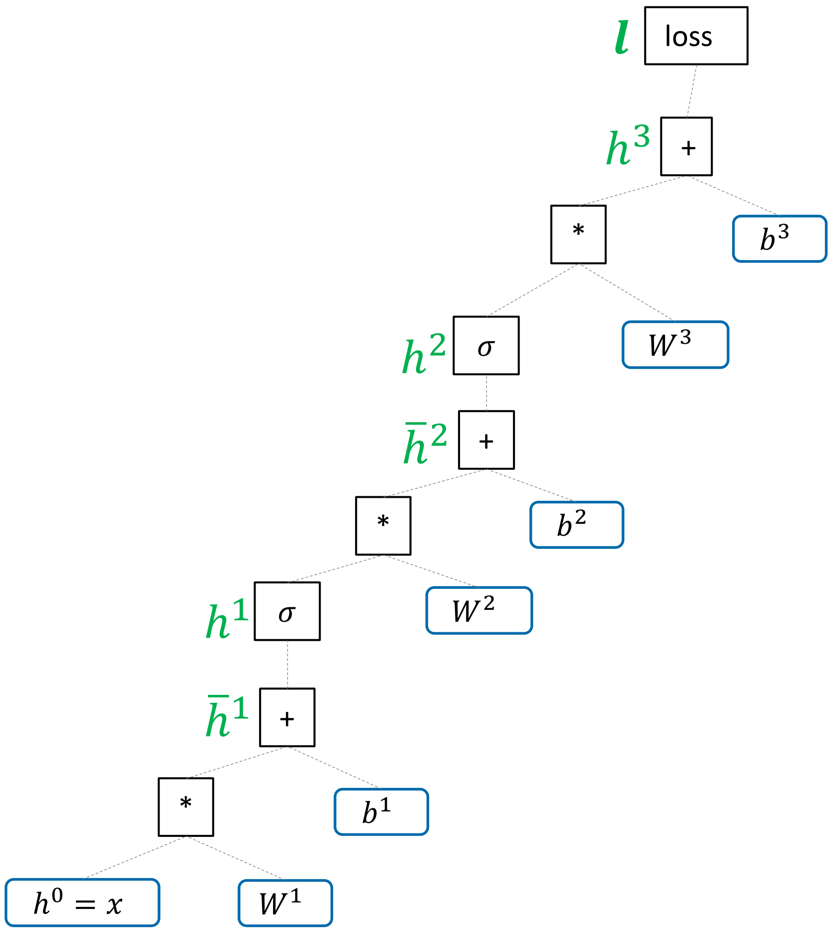
From loss to W^2 and b^2



- $\bar{h}^2 = W^2 h^1 + b^2$
- $\dfrac{\partial l}{\partial W^2} = \dfrac{\partial l}{\partial \bar{h}^2} \cdot \dfrac{\partial \bar{h}^2}{\partial W^2} = (\bar{g}^2)^T (h^1)^T$
  - $[n_2 \times 1] \times [1 \times n_1] \rightarrow [n_2 \times n_1]$
- $\dfrac{\partial l}{\partial b^2} = \dfrac{\partial l}{\partial \bar{h}^2} \cdot \dfrac{\partial \bar{h}^2}{\partial b^2} = \bar{g}^2$
  - $[1 \times n_2]$

Back propagation

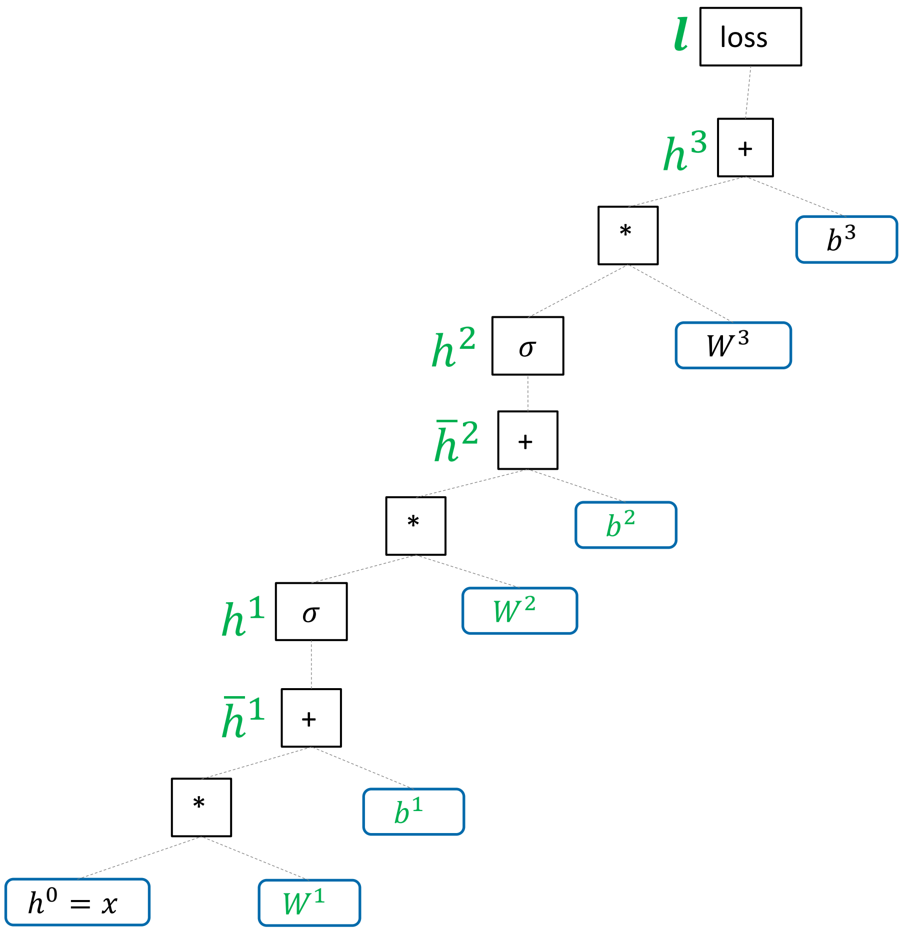
From loss to h^1 and $\bar{h}^1$



- $\bar{h}^2 = W^2 h^1 + b^2$
- $\boldsymbol{g^1} = \frac{\partial l}{\partial h^1} = \frac{\partial l}{\partial \bar{h}^2} \cdot \frac{\partial \bar{h}^2}{\partial h^1} = \bar{g}^2 W^2$
  - $[1 \times n_2] \times [n_2 \times n_1] \to [1 \times n_1]$
- $\boldsymbol{h^1 = \sigma(\bar{h}^1)}$ (element-wise activation)
- $\frac{\partial h^1}{\partial \bar{h}^1} = \boldsymbol{diag(\sigma'(\bar{h}^1))}$
  - $\sigma'(\bar{h}^1)$ is **element-wise derivative** and $diag(u)$ is the **diagonal matri**x corresponding to the **vector** $\boldsymbol{u}$ (the diagnose is u and others are zeros).
- $\bar{g}^1 = \frac{\partial l}{\partial \bar{h}^1} = \frac{\partial l}{\partial h^1} \cdot \frac{\partial h^1}{\partial \bar{h}^1} = \boldsymbol{g^1 diag(\sigma'(\bar{h}^1))}$
  - $[1 \times n_1] \times [n_1 \times n_1] \to [1 \times n_1]$

Back propagation

From loss to W^1 and b^1



- $\bar{h}^1 = W^1 h^0 + b^1 \quad (h^0 = x)$
- $\frac{\partial \boldsymbol{l}}{\partial \boldsymbol{W}^1} = \frac{\partial \boldsymbol{l}}{\partial \bar{\boldsymbol{h}}^1} \cdot \frac{\partial \bar{\boldsymbol{h}}^1}{\partial \boldsymbol{W}^1} = (\bar{\boldsymbol{g}}^1)^T (\boldsymbol{h}^0)^T$
  - $[n_1 \times 1] \times [1 \times d] \rightarrow [n_1 \times d]$
- $\frac{\partial \boldsymbol{l}}{\partial \boldsymbol{b}^1} = \frac{\partial \boldsymbol{l}}{\partial \bar{\boldsymbol{h}}^1} \cdot \frac{\partial \bar{\boldsymbol{h}}^1}{\partial \boldsymbol{b}^1} = \bar{\boldsymbol{g}}^1$
  - $[1 \times n_1]$

Exercise: How to compute $\frac{\partial l}{\partial x}$?

SGD for deep learning

b = 32 //batch size

iter_per_epoch = N/b //epoch means one round going through all data points

n_epoch = 50 //number of epochs

for epoch=1 to n_epoch do

  for i=1 to iter_per_epoch do

    Sample a minibatch $B = \left\{\left(x_{i_j}, y_{i_j}\right)\right\}_{j=1}^{b}$ from the training set

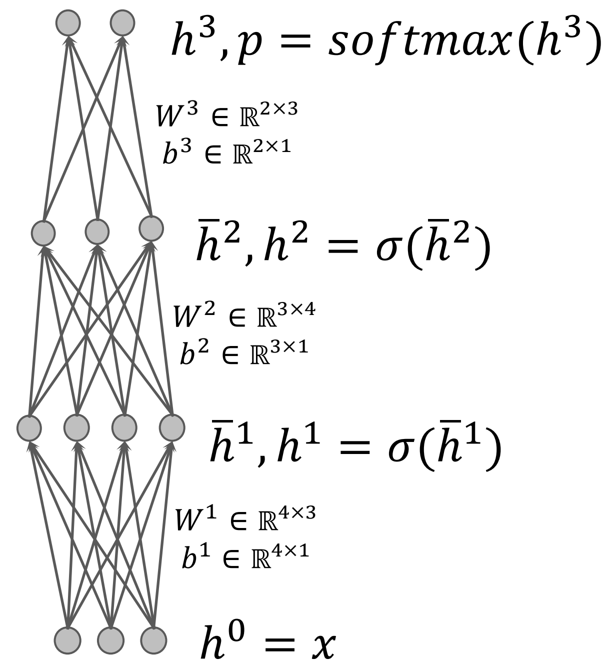
    Do forward propagation for B

    Do back propagation to compute $\left(\frac{\partial l}{\partial W^k}, \frac{\partial l}{\partial b^k}\right)_{k=1}^{L}$

    for k=1 to L do

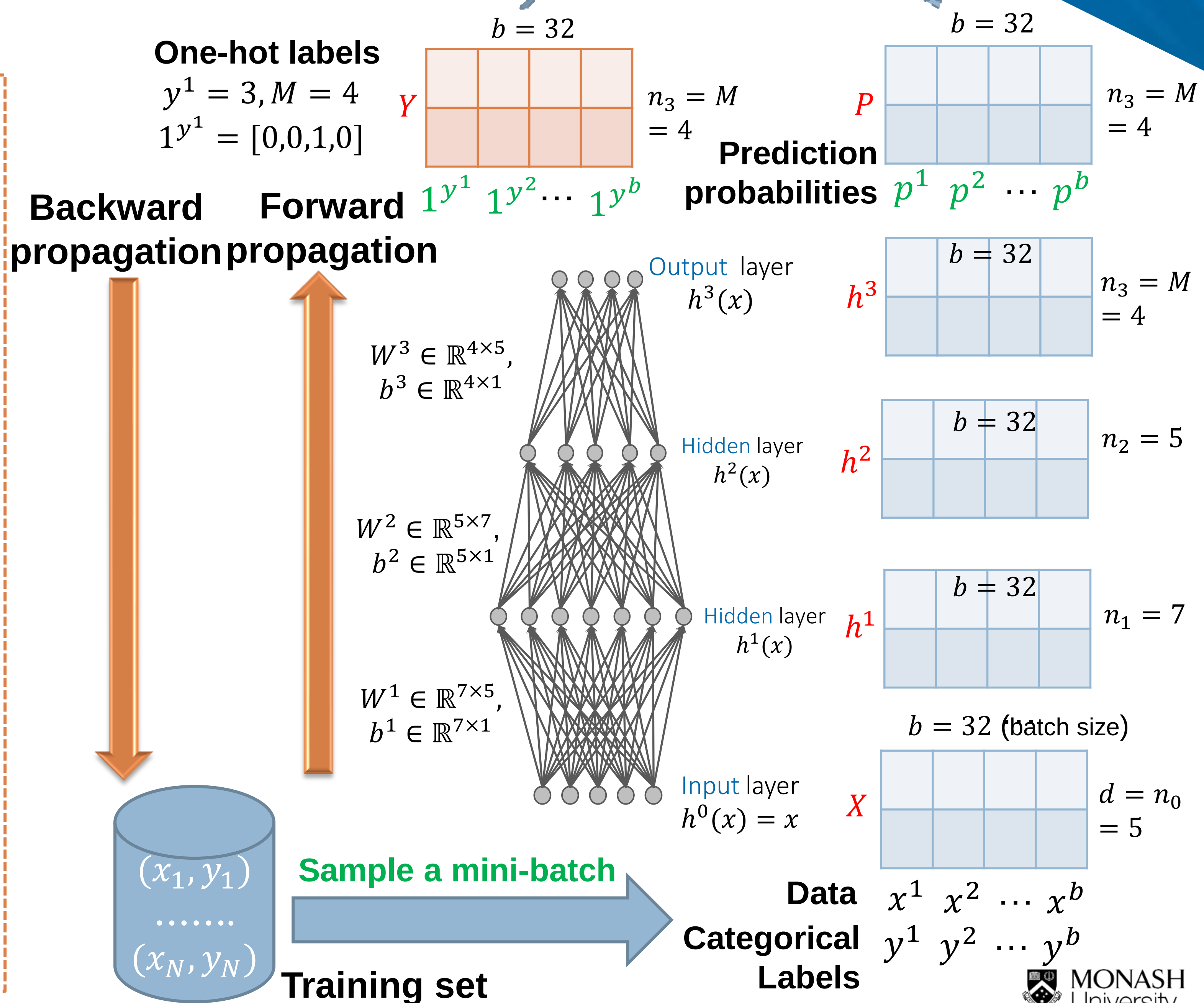
      $W_k = W_k - \eta \frac{\partial l}{\partial W^k}$

      $b_k = b_k - \eta \frac{\partial l}{\partial b^k}$



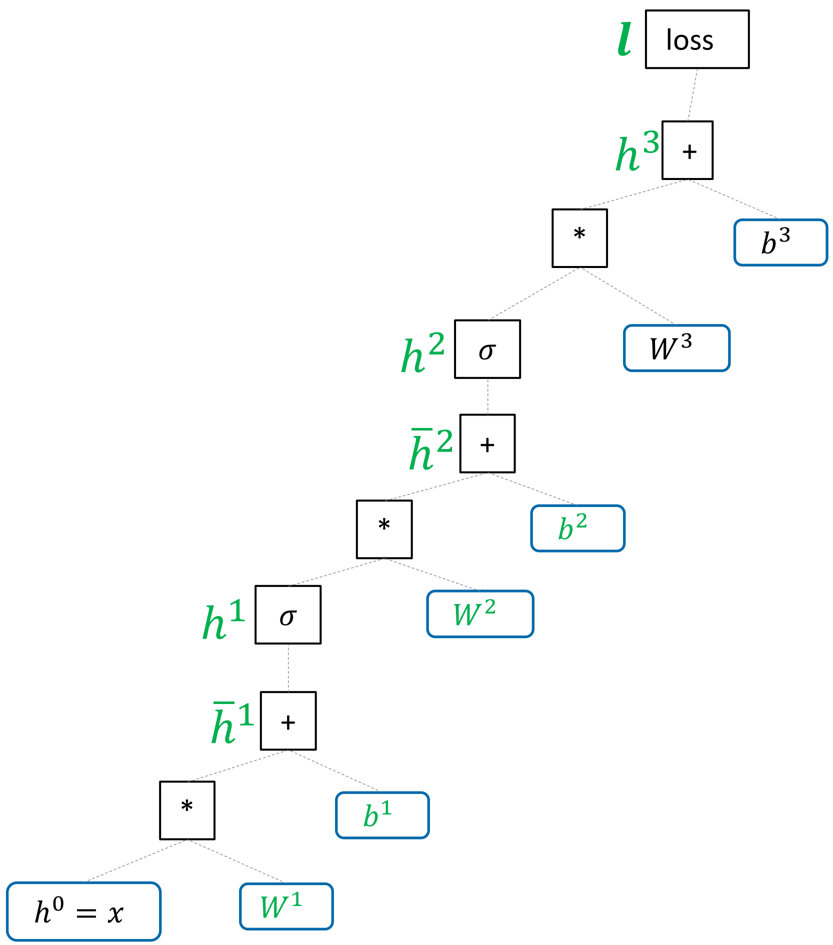
Mini-batch feed-forward

$$batch\_loss = \frac{1}{b}\sum_{i=1}^{b} CE(1^{y_i}, p^i)$$



- Input
  - Tensor X: $[n_0 = d, b]$ (b is the batch size)
- Hidden layer 1
  - Tensor $[n_1, b]$
- ……………
- Output layer
  - Tensor P: $[n_L = M, b]$
- The loss of the batch
  - $\frac{1}{b}\sum_{i=1}^{b} CE(1^{y_i}, p^i) = -\frac{1}{b}\sum_{i=1}^{b} \log p_{y_i}^{i}$
  - Update **weight matrices** and **biases** to **minimize** the batch loss using back **propagation**.

Why does deep learning need GPU and TPU?



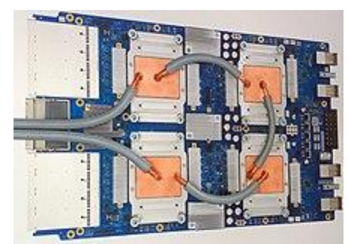
- Let consider

$$\frac{\partial l}{\partial W^1} = \frac{\partial l}{\partial h^3} \cdot \frac{\partial h^3}{\partial h^2} \cdot \frac{\partial h^2}{\partial \bar{h}^2} \cdot \frac{\partial \bar{h}^2}{\partial h^1} \cdot \frac{\partial h^1}{\partial \bar{h}^1} \cdot \frac{\partial \bar{h}^1}{\partial W^1}$$

$$= \left[(p - 1_y) W^3 diag\left(\sigma'(\bar{h}^2)\right) W^2 diag\left(\sigma'(\bar{h}^1)\right) \right]^T (h^0)^T$$

- For a really deep net, this **back propagation** requires many **matrix multiplications**
  - We need specific hardware that can parallel and significantly speed up matrix multiplication operation
  - **GPU** (Graphic Processing Unit) and **TPU** (Tensor Processing Unit)

GPU (Source: HelloTech)

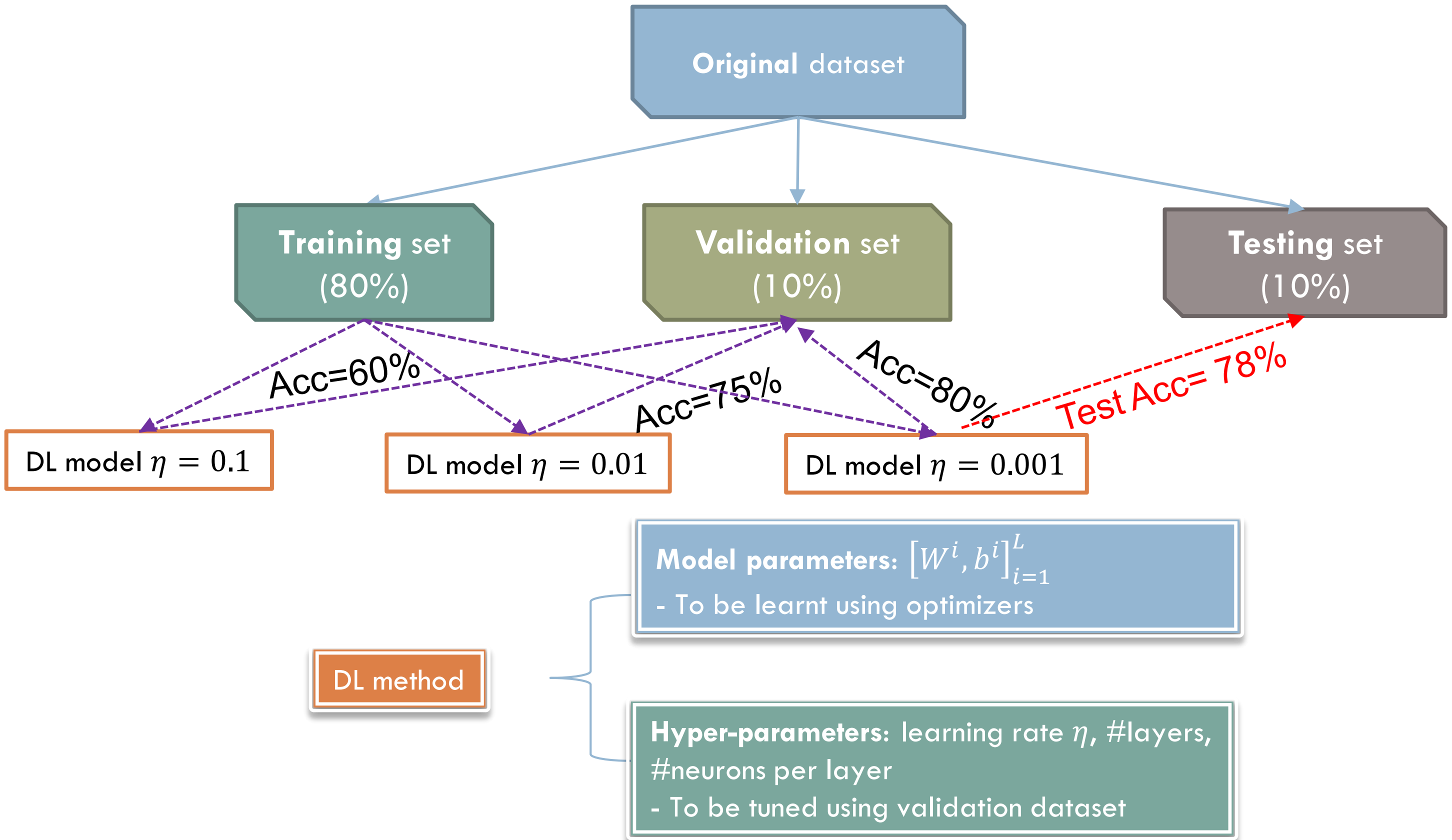


TPU (Source: Wikepedia)

Deep learning pipeline

Tuning hyper-parameters

- We want to train our DL model on a **training set** such that the **trained model** can predict well **unseen data** in a **separate testing set**.

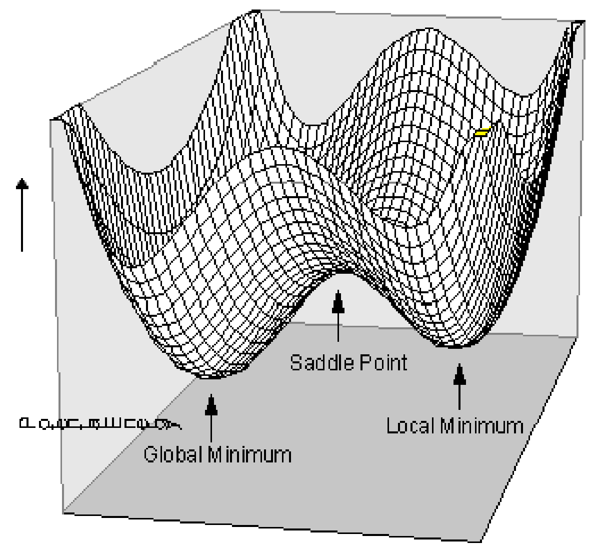



Q&A
When? Why? How? WHERE? What? How? Why? WHAT? Why? When?
WHAT? WHEN? HOW? WHEN? When? WHERE?
When? WHERE? WHAT? Why? When? WHAT?
WHO? WHEN?
WHERE? What? HOW?
Why? HOW? Where? Where? WHERE? When? What? What? WHERE? WHERE? When? When?
Why? WHEN? WHEN? HOW? When? When? HOW? WHAT? Why?

Optimizers for deep learning

Challenges of optimization for Deep Learning (Forum discussion)

- The **optimization problem** in **deep learning:**
  - $\min_{\theta} J(\theta) := L(D;\theta) := \frac{1}{N}\sum_{i=1}^{N} l(x_i, y_i; \theta) = -\frac{1}{N}\sum_{n=1}^{N} \log \frac{\exp\{h_{y_i}^{L}(x_i)\}}{\sum_{m=1}^{M} \exp\{h_m^{L}(x_i)\}}$
- A very **complex** and **complicated** objective function
  - Highly **non-linear** and **non-convex** function
  - The **loss surface** is very **complex**
  - Many local minima points, but the number of saddle points is even **exponentially much more**
- Need **efficient optimizers** to solve
  - SGD with momentum, Adagrad, Adadelta, RMSProp, Adam, and Nadam
  - They are **built-in optimizers** of TF.



(Source: Jan Jakubik)

SGD and SGD with momentum

(Source: DL book, Ch. 8)

Algorithm 8.1 Stochastic gradient descent (SGD) update at training iteration k

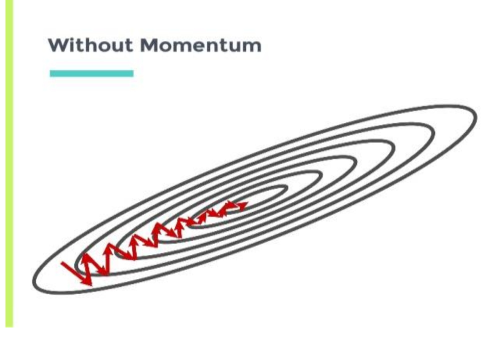
Require: Learning rate ϵ_k.
Require: Initial parameter $\boldsymbol{\theta}$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m}\nabla_{\boldsymbol{\theta}}\sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon\hat{\boldsymbol{g}}$
end while

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

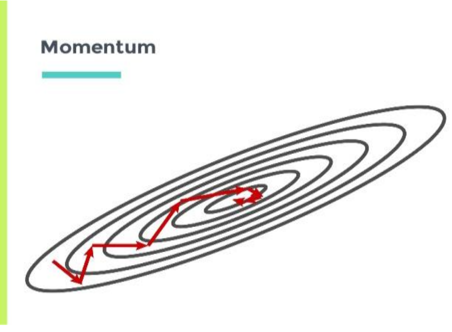
Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}}\sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha\boldsymbol{v} - \epsilon\boldsymbol{g}$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

- SGD uses only the **gradient of the mini-batch** to update the model
- It is fast at first several epochs and becomes **much slower** later.

- SGD with momentum uses a **velocity vector $\boldsymbol{v}$** which **stores the past gradients** together with the **current gradient** to speed up SGD
  - α is a hyper-parameter that indicates how quickly the contributions of previous gradients. In practice, this is usually set to 0.5, 0.9, and 0.99.
  - The momentum primarily solves 2 problems: **poor conditioning** of the Hessian matrix and **variance** in the stochastic gradient.



(Source: Sebastian Ruder)



SGD with Nesterov Momentum

Not in assessment

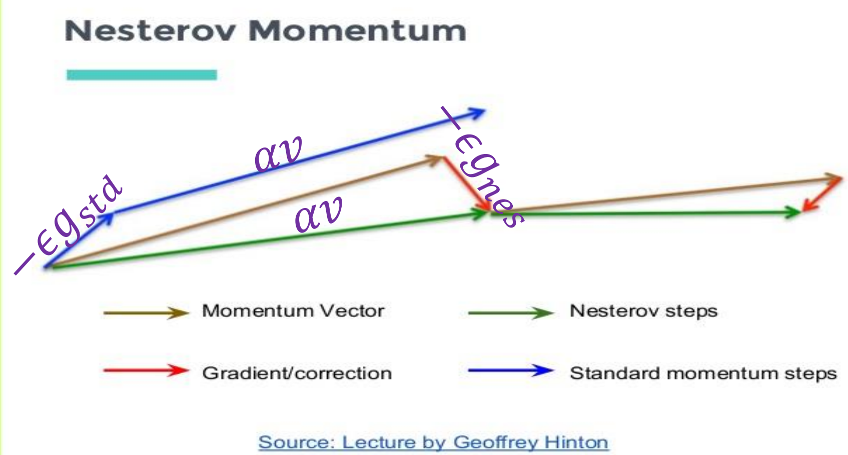
Algorithm 8.3 Stochastic gradient descent (SGD) with Nesterov momentum

---

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
while stopping criterion not met **do**
  Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding labels $\boldsymbol{y}^{(i)}$.
  Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
  Compute gradient (at interim point): $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$
  Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

(Source: DL book, Ch. 8)

- The only difference between **Nesterov momentum** and **standard momentum** is how the gradient is computed.
  - With Nesterov momentum the gradient is computed after the velocity is applied
- For **convex batch gradient**, Nesterov momentum **improves** the convergence rate from $\mathcal{O}(1/k)$ to $\mathcal{O}(1/k^2)$.
- Unfortunately, this result **does not** hold for stochastic gradient.



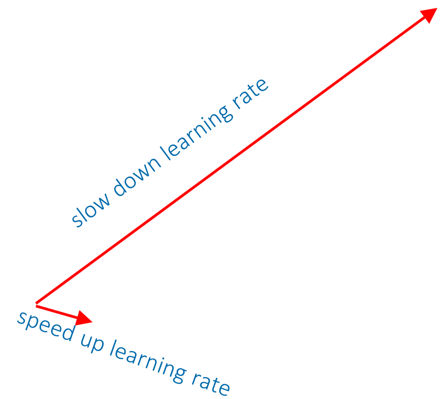
(Source: lecture of Goeffrey Hinton)

AdaGrad

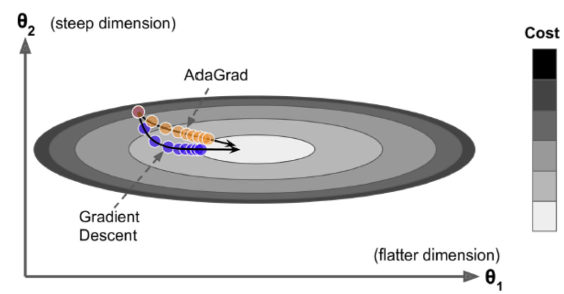
Not in assessment

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, perhaps 10^{-7}, for numerical stability
Initialize gradient accumulation variable $\boldsymbol{r} = \boldsymbol{0}$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$
Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end while



- Learning rates are scaled by the square root of the cumulative sum of squared gradients
- Direction with large partial derivatives
  - Thus, rapid decrease in their learning rates
- Direction with small partial derivatives
  - Hence relatively small decrease in their learning rates
- Weakness: always decrease the learning rate!
  - Excellent for convex problems
  - But not so good for DL (with non-convex problems)



(Source: Hands. On, Ch. 11)

Not in assessment

RMSProp

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ.
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers.
Initialize accumulation variables $\boldsymbol{r} = 0$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$
Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
end while

(Source: DL book, Ch. 8)

- A modification of AdaGrad to work better for **non-convex** setting.
- Instead of cumulative sum, use exponential moving/smoothing average.
- RMSProp has been shown to be an effective and practical optimization algorithm for DNN.
  - Currently one of the go-to optimization methods being employed routinely by DL applications.

Adam

Not in assessment

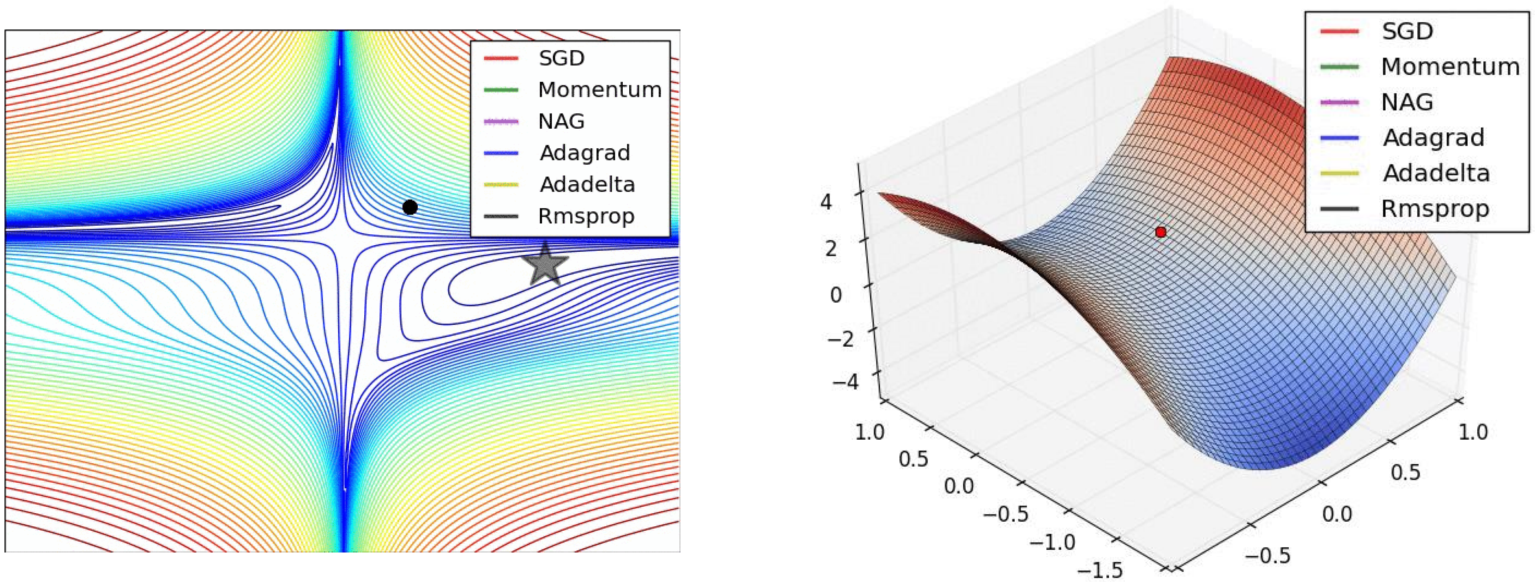
- The best variant that essentially combines RMSProp with momentum

- Suggested default values: $\eta = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$ and $\epsilon = 10^{-8}$.

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization. (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
  Initialize time step $t = 0$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1) \boldsymbol{g}$
    Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2) \boldsymbol{g} \odot \boldsymbol{g}$
    Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
    Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
    Compute update: $\Delta \boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
  end while

Visual comparison of all optimizers



[Source: Sebastian Ruder]

TensorFlow optimizers

- TensorFlow pre-implemented methods for **TF 1.x**:
  - `tf.train.GradientDescentOptimizer` (Standard SGD)
  - `tf.train.MomentumOptimizer` (SGD with momentum)
  - `tf.train.AdaGradOptimizer` (AdaGrad)
  - `tf.train.RMSPropOptimizer` (RMSProp)
  - `tf.train.AdamOptimizer` (Adam)

```
optimizer_names = ["Nadam", "Adam", "Adadelta", "Adagrad", "RMSprop", "SGD"]
optimizer_list = [keras.optimizers.Nadam(learning_rate=0.001), keras.optimizers.Adam(learning_rate=0.001), keras.optimizers.Adadelta(learning_rate=0.001),
                  keras.optimizers.Adagrad(learning_rate=0.001), keras.optimizers.RMSprop(learning_rate=0.001), keras.optimizers.SGD(learning_rate=0.001)]
best_acc = 0
best_i = -1
for i in range(len(optimizer_list)):
    print("*Evaluating with {}\n".format(str(optimizer_names[i])))
    dnn_model.compile(optimizer=optimizer_list[i], loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    dnn_model.fit(x=X_train, y=y_train, batch_size=32, epochs=30, validation_data=(X_valid, y_valid), verbose=0)
    acc = dnn_model.evaluate(X_test, y_test)[1]
    print("The test accuracy is {}\n".format(acc))
    if acc > best_acc:
        best_acc = acc
        best_i = i
print("The best accuracy is {} with {}".format(best_acc, optimizer_names[best_i]))
```

For TF 2.x

Optimizers in deep learning

Optimizers

First order methods

Second order methods (Newton method)

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma H^{-1}\boldsymbol{g}$$

- **Impossible** for DL

Adaptive learning rate based on gradients

Standard Mini-batched SGD

Adagrad

$$\boldsymbol{g} \leftarrow \nabla J(\boldsymbol{\theta})$$
$$\boldsymbol{\gamma} \leftarrow \boldsymbol{\gamma} + \boldsymbol{g} \odot \boldsymbol{g}$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\eta}{\epsilon + \sqrt{\boldsymbol{\gamma}}} \odot \boldsymbol{g}$$

RMSProp

$$\boldsymbol{g} \leftarrow \nabla J(\boldsymbol{\theta})$$
$$\boldsymbol{\gamma} \leftarrow \beta\boldsymbol{\gamma} + (1-\beta)\boldsymbol{g} \odot \boldsymbol{g}$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\eta}{\sqrt{\epsilon + \boldsymbol{\gamma}}} \odot \boldsymbol{g}$$

Adam

$$t \leftarrow t + 1$$
$$\boldsymbol{g} \leftarrow \nabla J(\boldsymbol{\theta})$$
$$\boldsymbol{s} \leftarrow \beta_1 \boldsymbol{s} + (1-\beta_1)\boldsymbol{g}$$
$$\boldsymbol{\gamma} \leftarrow \beta_2 \boldsymbol{\gamma} + (1-\beta_2)\boldsymbol{g} \odot \boldsymbol{g}$$
$$\boldsymbol{s} \leftarrow \frac{\boldsymbol{s}}{1-\beta_1^t}$$
$$\boldsymbol{\gamma} \leftarrow \frac{\boldsymbol{\gamma}}{1-\beta_2^t}$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\frac{\boldsymbol{s}}{\sqrt{\epsilon + \boldsymbol{\gamma}}}$$

SGD with momentum

$$\boldsymbol{v} \leftarrow \alpha\boldsymbol{v} - \eta\nabla J(\boldsymbol{\theta})$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$$

SGD with Nesterov momentum

$$\boldsymbol{v} \leftarrow \alpha\boldsymbol{v} - \eta\nabla J(\boldsymbol{\theta} + \alpha\boldsymbol{v})$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$$

Not in assessment

Summary

- Optimization problem in DL and ML
  - Regularization term + Empirical loss term
- Gradient descent
- Stochastic gradient descent
- Backward propagation
- Other optimizers in DL
  - SGD with momentum, Adagrad, RMSProp, and Adam
- First order methods and second order methods

Thanks for your attention!

Mini-batch feed-forward

- Input
  - Tensor X: $[n_0 = d, b]$ (b is the batch size)
- Hidden layer 1
  - Tensor $[n_1, b]$
- ....
- Output layer
  - Tensor P: $[n_L = M, b]$
- The loss of the batch
  - $\frac{1}{b}\sum_{i=1}^{b} CE(1^{y_i}, p^i) = -\frac{1}{b}\sum_{i=1}^{b} \log p^i_{y_i}$