3. Training Deep Networks

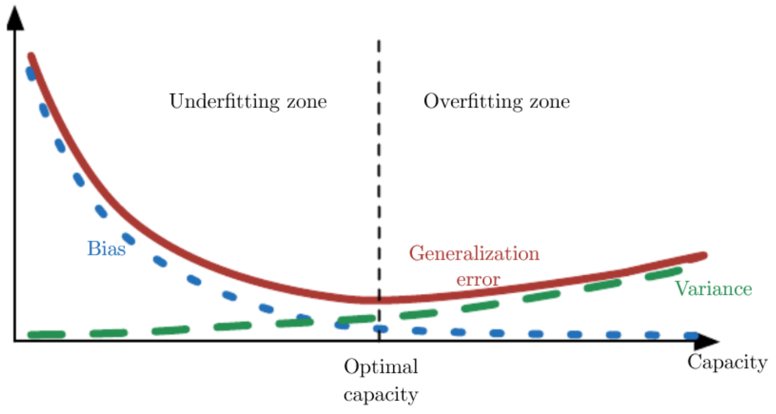
CS 5242 Neural Networks and Deep Learning

YOU, Yang

30.08.2022

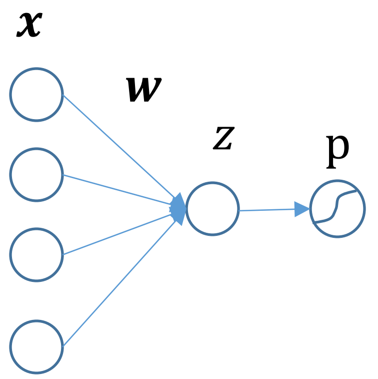
Recap

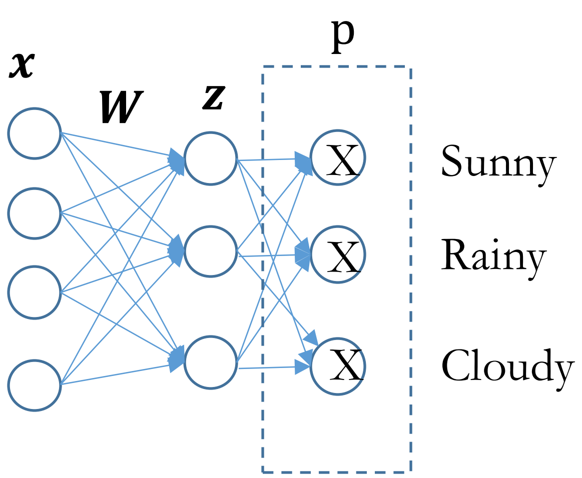
- Overfitting, underfitting, bias and variance
  - Polynomial regression
    - Hyper-parameter (degree M)
  - Dataset splitting
    - Training, validation, and test data
    - Tune the parameters to minimize training error
      - Overfit training data → evaluate on the validation data
    - Tune the hyper-parameters to minimize the validation error
      - Overfit validation data→ evaluate on the test data



Recap

- Perceptron model
- Binary classification model
  - Logistic function → probability
  - Binary cross-entropy loss
- Multi-class classification model
  - Softmax/multinomial regression
    - Softmax function → a vector of probabilities
  - Cross-entropy loss





Agenda

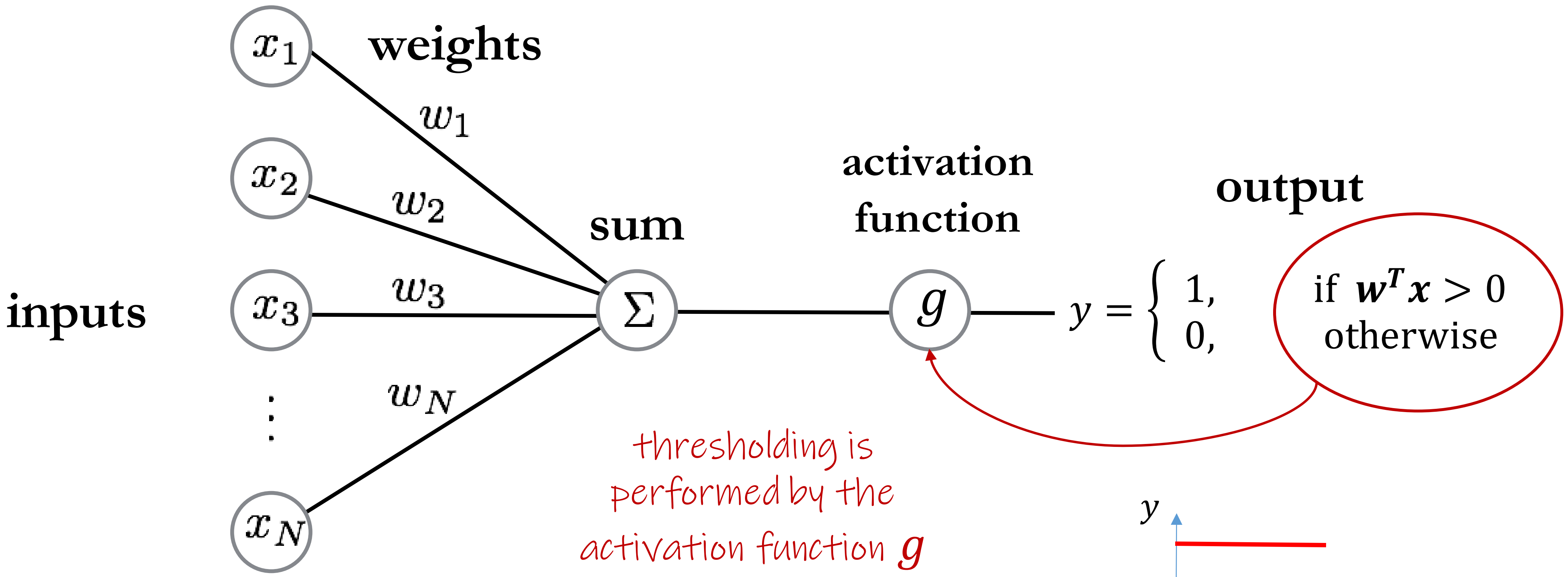
- Multi-layer Perceptrons
  - Activation functions & non-linear feature transformations
  - Going “deeper”
- Backpropagation algorithm
  - For computing the gradients of a deep neural network
- Advanced optimization algorithms
  - Stochastic gradient descent (SGD)
  - SGD+
- Training tricks
  - Data normalization
  - Parameter initialization

Multi-Layer Perceptrons (MLP)

Non-linear Activation Functions

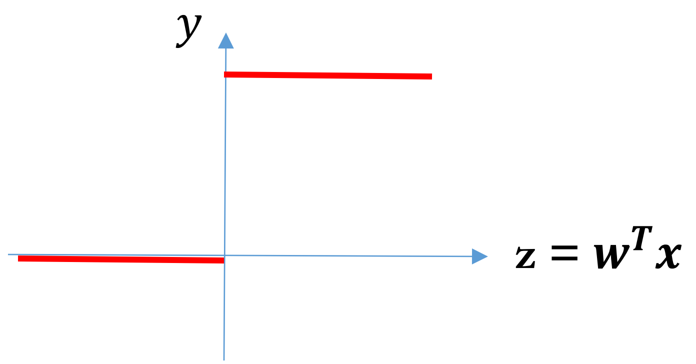
Definition of MLPs

The Perceptron



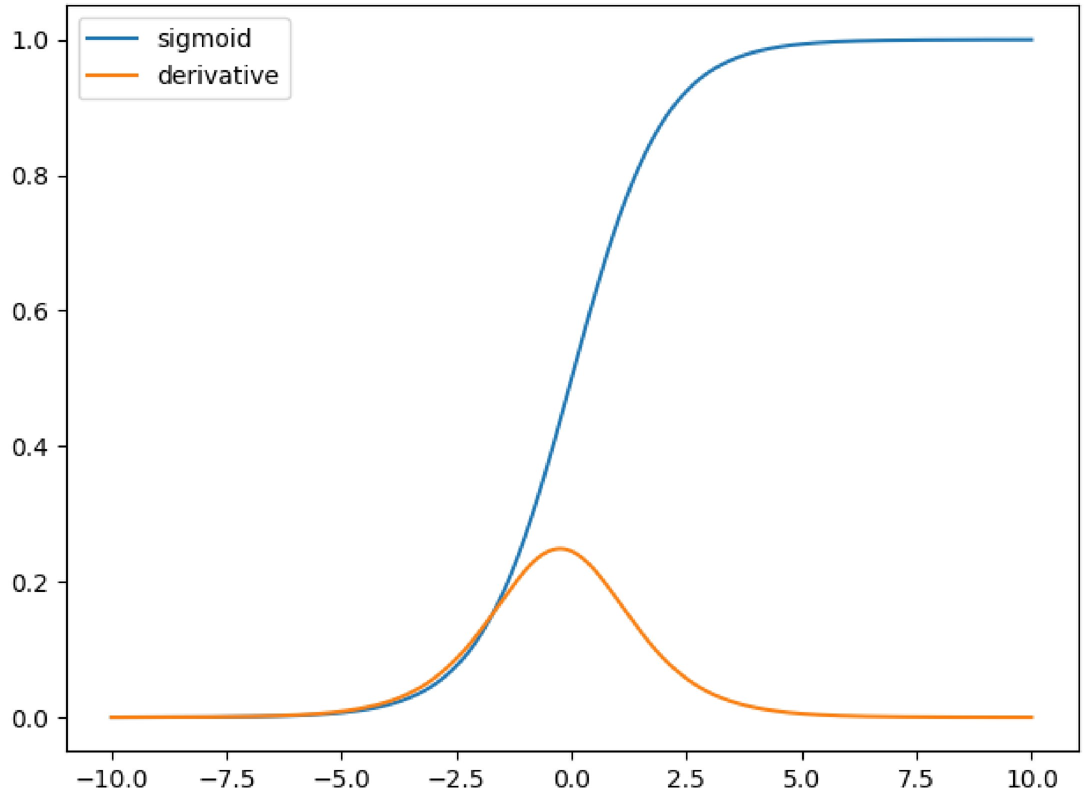
$$y = \begin{cases} 1, & \text{if } \boldsymbol{w}^T\boldsymbol{x} > 0 \\ 0, & \text{otherwise} \end{cases}$$

thresholding is performed by the activation function g



Logistic / Sigmoid Activation

We've already seen another type of activation function: Logistic or sigmoid function.



https://i.stack.imgur.com/inMoa.png

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

$$\frac{\partial \sigma}{\partial z} = \sigma(z)(1 - \sigma(z))$$

detailed derivation [here]

Activation functions $g()$

A linear function or mapping $f(x)$ has properties of

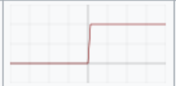
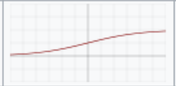
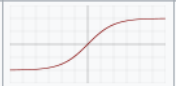
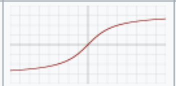
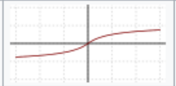
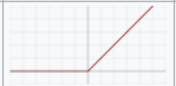
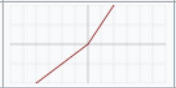
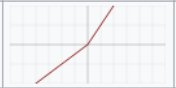
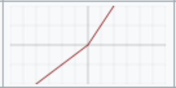
- *additivity* $f(x+y) = f(x) + f(y)$ and
- *homogeneity* $f(\alpha x) = \alpha f(x)$.

- Activation functions perform non-linear feature transformations

Q: Why do we need non-linear activation functions?

(1) Linear transformations alone are limited.

(2) Gives meaning to multi-layer perceptrons, because otherwise, linear layers would collapse (explain in later slides)

name	plot	equation	derivative	range
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$	$\{0, 1\}$
Logistic (a.k.a. Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$	$(0, 1)$
TanH		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$	$(-1, 1)$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$	$\left(-\frac{\pi}{2}, \frac{\pi}{2}\right)$
Softsign [7][8]		$f(x) = \frac{x}{1 + \lvert x \rvert}$	$f'(x) = \frac{1}{(1 + \lvert x \rvert)^2}$	$(-1, 1)$
Rectified linear unit (ReLU)[9]		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$[0, \infty)$
Leaky rectified linear unit (Leaky ReLU)[10]		$f(x) = \begin{cases} 0.01x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0.01 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$(-\infty, \infty)$
Parameteric rectified linear unit (PReLU)[11]		$f(\alpha, x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(\alpha, x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$(-\infty, \infty)$
Randomized leaky rectified linear unit (RReLU)[12]		$f(\alpha, x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$ [1]	$f'(\alpha, x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$(-\infty, \infty)$

Source from: https://en.wikipedia.org/wiki/Activation_function

Slide adapted from Wang Wei

Linear functions are limited

- Simple binary example:

$$y = \begin{cases} 1, & \text{if } \boldsymbol{w}^T\boldsymbol{x} > 0 \\ 0, & \text{otherwise} \end{cases}$$

"Training" algorithm

```
For each training example (x, y)
    Compute the prediction $\tilde{y}$
    Update $w_i = w_i + \alpha * (y - \tilde{y}) * x_i$
```

Possible weights for AND function:
$w_1 = 1, w_2 = 1, b = -1.5$

- **Multi-layer perceptrons with only linear functions are still single-layer perceptrons.**
- **Perceptrons are mainly termed as "linear classifiers" and can be used only for linear separable applications.**
- **XOR is not linearly separable.**
- **We need real multi-layer perceptrons.**

AND function

x1	x2	y
1	1	1
1	0	0
0	0	0
0	1	0

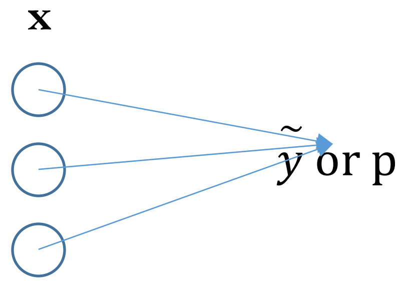
XOR function

x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0

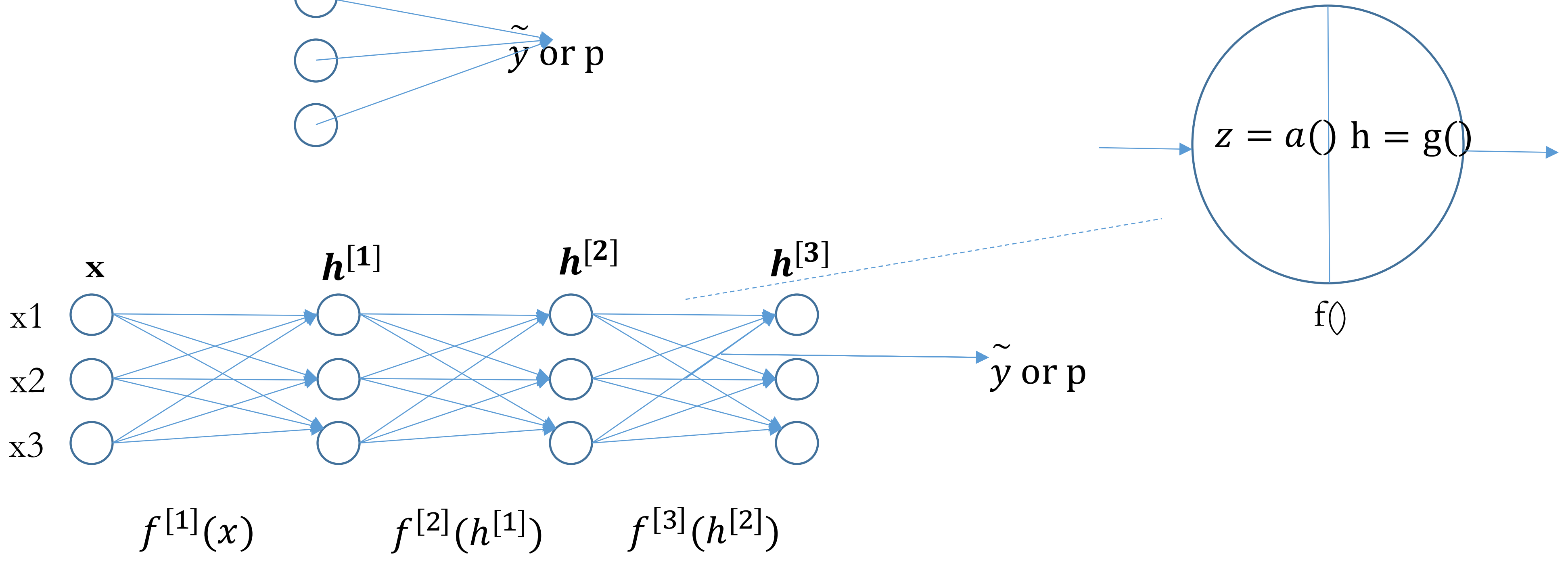
Slide adapted from Wang Wei

Multi-layer Perceptron

single layer perceptron (no hidden layers)



multi-layer perceptron



Slide credit: Wang Wei

Multi-layer Perceptron

A net with multiple layers that transform input features into hidden features and then make predictions

- At least one hidden layer
- i-th layer consists of a **linear/affine** transformation function

$$\boldsymbol{z}^{[i]} = a^{[i]}(\boldsymbol{h}^{[i-1]}) = W^{[i]}\boldsymbol{h}^{[\boldsymbol{i}-\mathbf{1}]} + \boldsymbol{b}^{[i]}$$

$$W^{[i]} \in R^{n_i \times n_{i-1}}, \boldsymbol{b}^{i} \in R^{n_i}$$

- $\boldsymbol{n_i}$ is the number of hidden units at the i-th layer and is a tunable hyper-parameter
- followed by a **non-linear activation** function

$$\boldsymbol{h}^{[i]} = g^{[i]}(\boldsymbol{z}^{[i]}), \in R^{n_i}$$

Slide credit: Wang Wei

A Linear MLP?

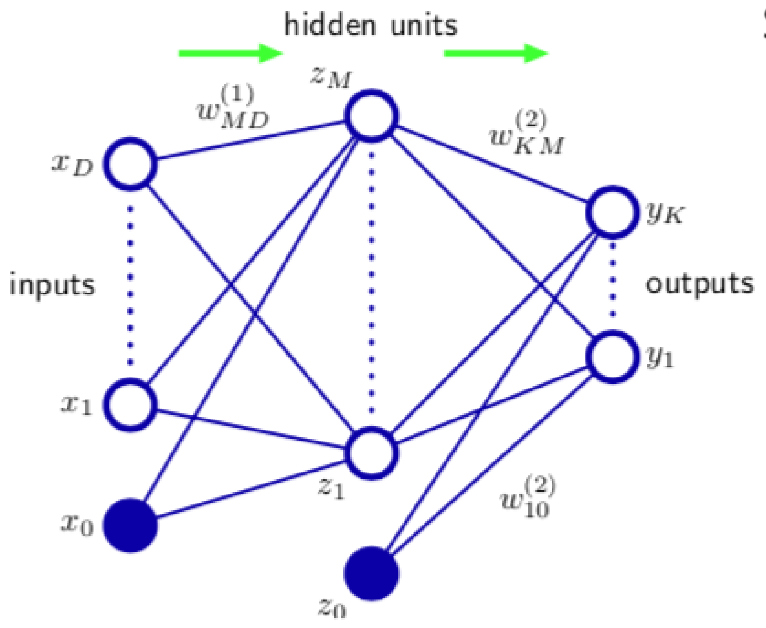


Figure: 5.1 from Bishop

Suppose we have a network with N hidden layers where activations h were linear

$$\begin{aligned}
y_k(\boldsymbol{x}, \boldsymbol{w}) &= \sigma\Big(\boldsymbol{w}^{(N)}_{n\cdot} \cdot h^{(N-1)} \cdots h^{(2)}\big(\boldsymbol{w}^{(2)}_{k\cdot} \cdot h^{(1)}(\boldsymbol{w}^{(1)}_{j\cdot} \cdot \boldsymbol{x})\big)\Big) \\
&= \sigma\Big(\boldsymbol{w}^{(N)}_{n\cdot} \cdot h^{(N-1)} \cdots h^{(2)}\big(\boldsymbol{w}^{(2)}_{k\cdot} \cdot \underbrace{\boldsymbol{w}^{(1)}_{j\cdot} \cdot h^{(1)}}_{\text{weighting outside of activation}} (\boldsymbol{x})\big)\Big) \\
&= \sigma\Big(\underbrace{\boldsymbol{w}^{(N)}_{n\cdot} \cdots \boldsymbol{w}^{(2)}_{k\cdot} \cdot \boldsymbol{w}^{(1)}_{j\cdot}}_{\boldsymbol{w}'} \cdot \underbrace{h^{(N-1)} \circ h^{(2)} \circ h^{(1)}}_{g}(\boldsymbol{x})\Big) \\
&= \sigma\big(\boldsymbol{w}' g(\boldsymbol{x})\big)
\end{aligned}$$

This result says that if all activation functions were linear, we can always define an equivalent collapsed network.

Because successive linear transformations together form yet another linear transformation, a multi-layered linear network is not so interesting. Therefore, we need non-linear activations.

Activation functions

- Logistic (Sigmoid, σ) VS ReLU

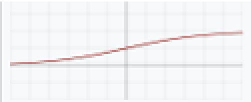
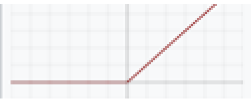
Logistic activation

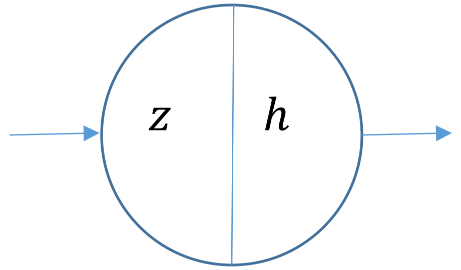
If $z_k$is large, e.g. >10, then h_k is near 1

If z_k is small, e.g. <-10, Then h is near 0

For both cases, $\frac{\partial h_k}{\partial z_k} \approx 0 \rightarrow$ gradient vanishing

à gradient vanishing

Logistic (a.k.a. Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1 - f(x))$	$(0, 1)$
Rectified linear unit (ReLU)[9]		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$[0, \infty)$



$$h^{[i]} = g^{[i]}(z^{[i]})$$

ReLU activation

If z_k is positive,$\frac{\partial h_k}{\partial z_k} = 1$, no gradient vanishing

If z_k is negative, $\frac{\partial h_k}{\partial z_k} = 0$ no gradients

better than sigmoid, since z_k has a larger working domain

Leaky ReLU (see previous slides) resolves the gradient vanishing problem for negative z_k

Slide credit: Wang Wei

Training MLP

- Cross-entropy loss for classification
- Squared Euclidean distance for regression
- GD algorithm
  - Compute $J(\boldsymbol{X}, \boldsymbol{Y})$
  - Compute gradient : $\frac{\partial J}{\partial \boldsymbol{W}^{[1]}}, \frac{\partial J}{\partial \boldsymbol{b}^{[1]}}, \frac{\partial J}{\partial \boldsymbol{W}^{[2]}}, \frac{\partial J}{\partial \boldsymbol{b}^{[2]}} \cdots$
  - Update: $\boldsymbol{W}^{[\boldsymbol{k}]} = \boldsymbol{W}^{[\boldsymbol{k}]} - \alpha \frac{\partial J}{\partial \boldsymbol{W}^{[k]}}, \boldsymbol{b}^{[\boldsymbol{k}]} = \boldsymbol{b}^{[\boldsymbol{k}]} - \alpha \frac{\partial J}{\partial \boldsymbol{b}^{[k]}}$

Slide credit: Wang Wei

Backpropagation

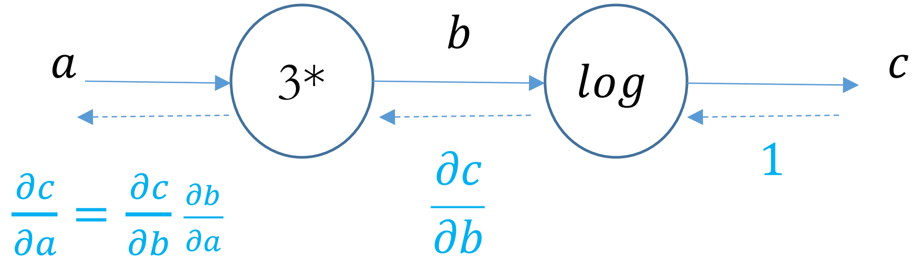
How to compute the gradient?

- GD algorithm needs the gradient of the loss w.r.t each parameter
- Linear regression model
  - $\frac{\partial L}{\partial \boldsymbol{w}} = (\tilde{y} - y)\boldsymbol{x}$
- Logistic regression model
  - $\frac{\partial L}{\partial \boldsymbol{w}} = (p - y)\boldsymbol{x}$
- Softmax regression model
  - $\frac{\partial L}{\partial W} = (\boldsymbol{p} - \boldsymbol{y})\boldsymbol{x}^T$
- How about MLP model with 10 hidden layers?
  - We need a **modular** way to compute the gradients

Slide credit: Wang Wei

Chain rule I

- $c = \log 3a$ Given $a = 1$, what is $\frac{\partial c}{\partial a}$?



a
$3*$
b
log
c
$\frac{\partial c}{\partial a} = \frac{\partial c}{\partial b}\frac{\partial b}{\partial a}$
$\frac{\partial c}{\partial b}$
1

- $b = f_1(a) = 3a,$ $c = f_2(b) = \log b \rightarrow c = \log 3a$
- $\frac{\partial c}{\partial a} = \frac{\partial c}{\partial b}\frac{\partial b}{\partial a} = f_2'(b) f_1'(a) = \frac{1}{b} * 3 = \frac{1}{3a} * 3 = \frac{3}{3} = 1$

Slide credit: Wang Wei

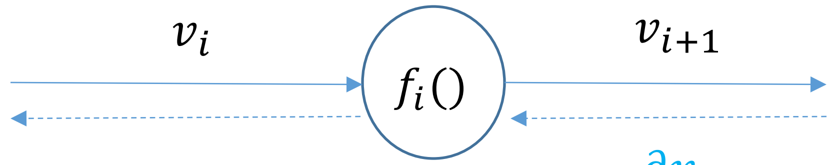
Chain rule I

- Basic chain rule
  - $v_2 = f_1(v_1), v_3 = f_2(v_2) \quad \dots \quad v_k = f_{k-1}(v_{k-1})$
  - v_i could be a scalar, vector, matrix, tensor
  - $\frac{\partial v_k}{\partial v_i} = mul(\frac{\partial v_k}{\partial v_{i+1}}, \frac{\partial v_{i+1}}{\partial v_i})$

General multiplication operation; to avoid the scalar/matrix/tensor transpose/ordering messiness …

Some special cases; shape-check to confirm!

  - Reorder and transpose
  - $\frac{\partial v_k}{\partial v_{i+1}} \frac{\partial v_{i+1}}{\partial v_i}$ if all are scalars
  - $\frac{\partial v_{i+1}}{\partial v_i} \frac{\partial v_k}{\partial v_{i+1}}$ if v_k is a scalar, v_{i+1}, v_i are vectors



$\frac{\partial v_k}{\partial v_{i+1}}$

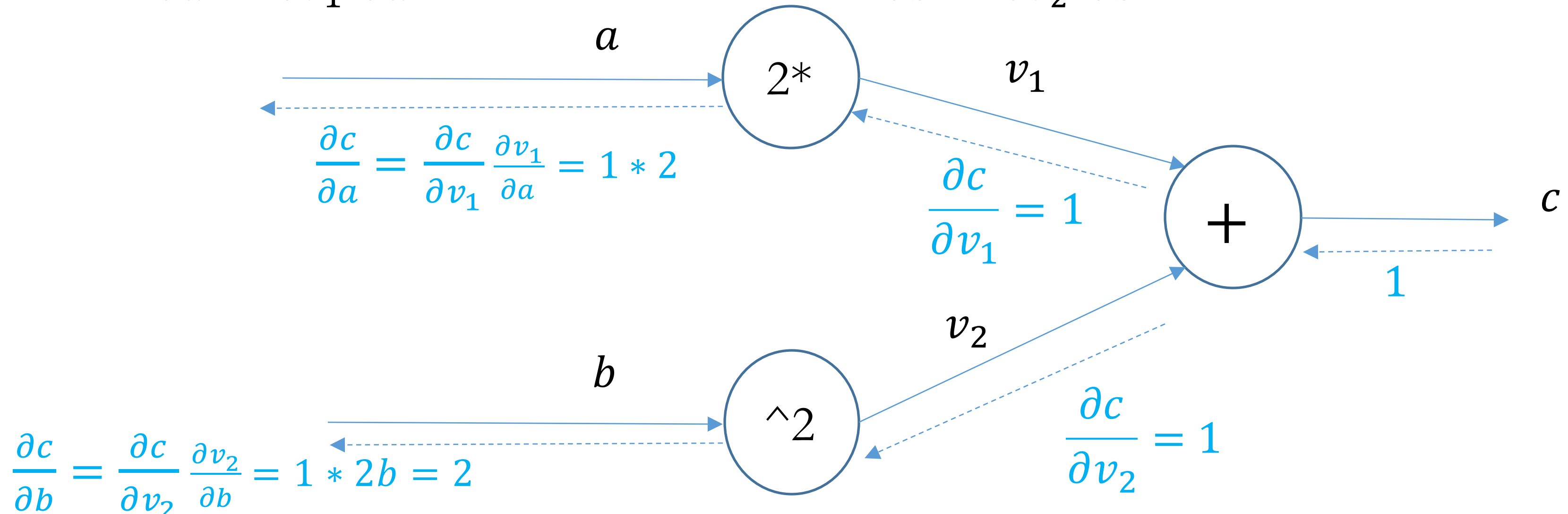
$\frac{\partial v_k}{\partial v_i} = mul(\frac{\partial v_k}{\partial v_{i+1}}, \frac{\partial v_{i+1}}{\partial v_i})$

$f_i'(v_i)$

Slide credit: Wang Wei

Chain rule II

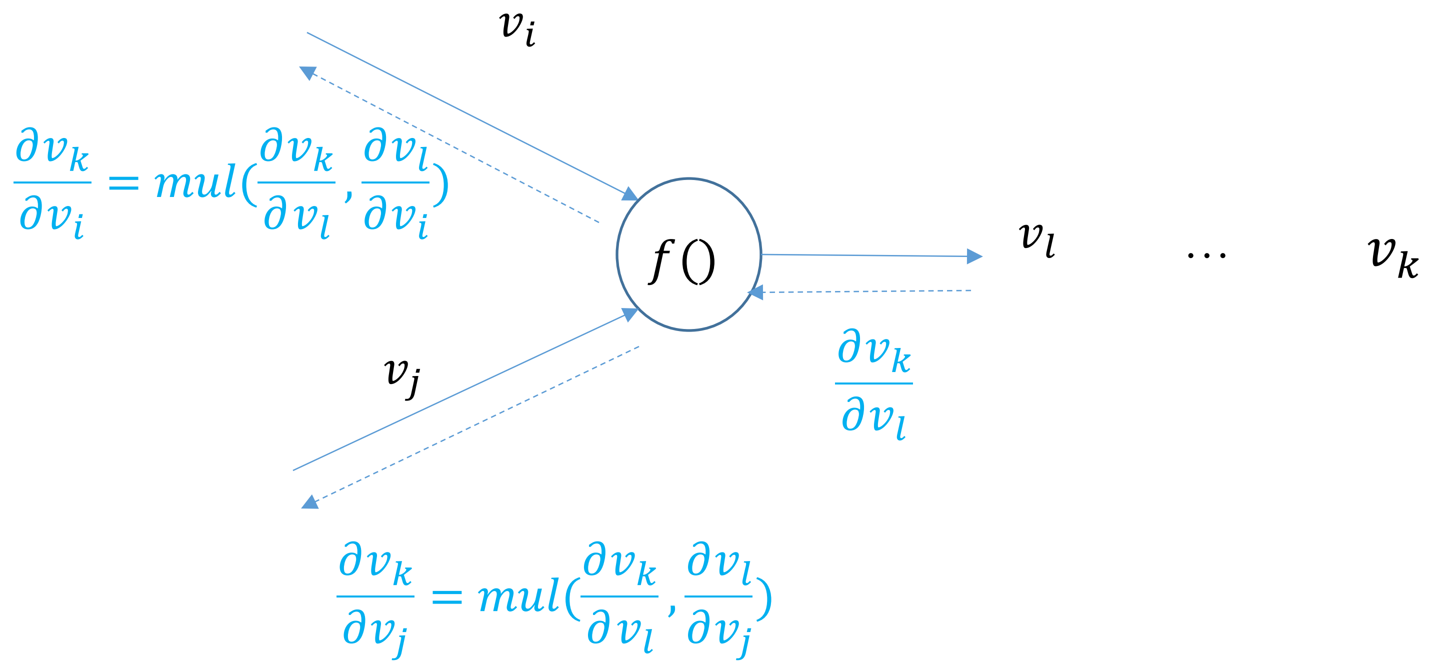
- $c = f(a, b) = 2a + b^2$
  - Given, $a = 1, b = 1$
  - $\frac{\partial c}{\partial a} = 2, \quad \frac{\partial c}{\partial b} = 2b = 2$
- $v_1 = f_1(a) = 2a, v_2 = f_2(b) = b^2, c = f_3(v_1, v_2) = v_1 + v_2$
  - $\frac{\partial c}{\partial a} = \frac{\partial c}{\partial v_1}\frac{\partial v_1}{\partial a} = 1 * 2 = 2 \qquad \frac{\partial c}{\partial b} = \frac{\partial c}{\partial v_2}\frac{\partial v_2}{\partial b} = 1 * 2b = 2$


a
$2*$
v_1
$\frac{\partial c}{\partial a} = \frac{\partial c}{\partial v_1}\frac{\partial v_1}{\partial a} = 1 * 2$
$\frac{\partial c}{\partial v_1} = 1$
$+$
c
1
v_2
b
$\hat{}2$
$\frac{\partial c}{\partial v_2} = 1$
$\frac{\partial c}{\partial b} = \frac{\partial c}{\partial v_2}\frac{\partial v_2}{\partial b} = 1 * 2b = 2$

Slide credit: Wang Wei

Chain rule II

- $v_l = f(v_i, v_j)$,
- v_k is the final output

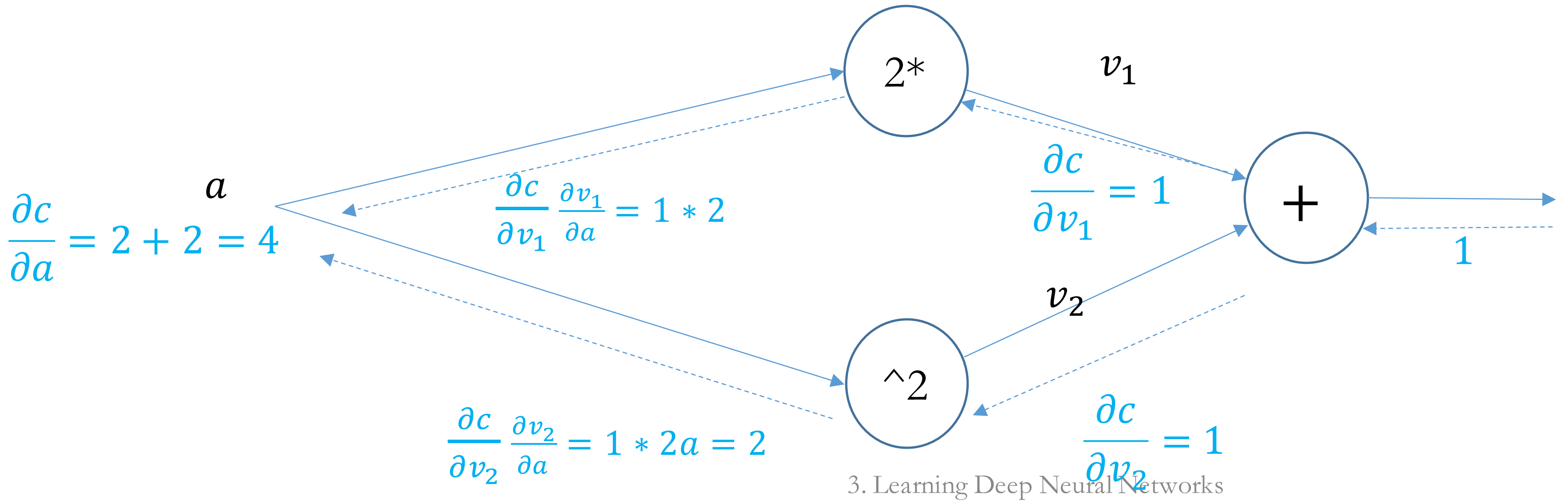


v_i
$\frac{\partial v_k}{\partial v_i} = mul(\frac{\partial v_k}{\partial v_l}, \frac{\partial v_l}{\partial v_i})$
$f()$
v_l
$\dots$
v_k
$\frac{\partial v_k}{\partial v_l}$
v_j
$\frac{\partial v_k}{\partial v_j} = mul(\frac{\partial v_k}{\partial v_l}, \frac{\partial v_l}{\partial v_j})$

Slide credit: Wang Wei

Chain rule III

- $c = 2a + a^2$
  - Given $a = 1, \frac{\partial c}{\partial a} = 2 + 2a = 4$
- $v_1 = f_1(a) = 2a, v_2 = f_2(a) = a^2, \mathrm{c} = f_3(v_1, v_2) = v_1 + v_2$
  - $\frac{\partial c}{\partial a} = \frac{\partial c}{\partial v_1}\frac{\partial v_1}{\partial a} + \frac{\partial c}{\partial v_2}\frac{\partial v_2}{\partial a} = 1 * 2 + 1 * 2a = 4$

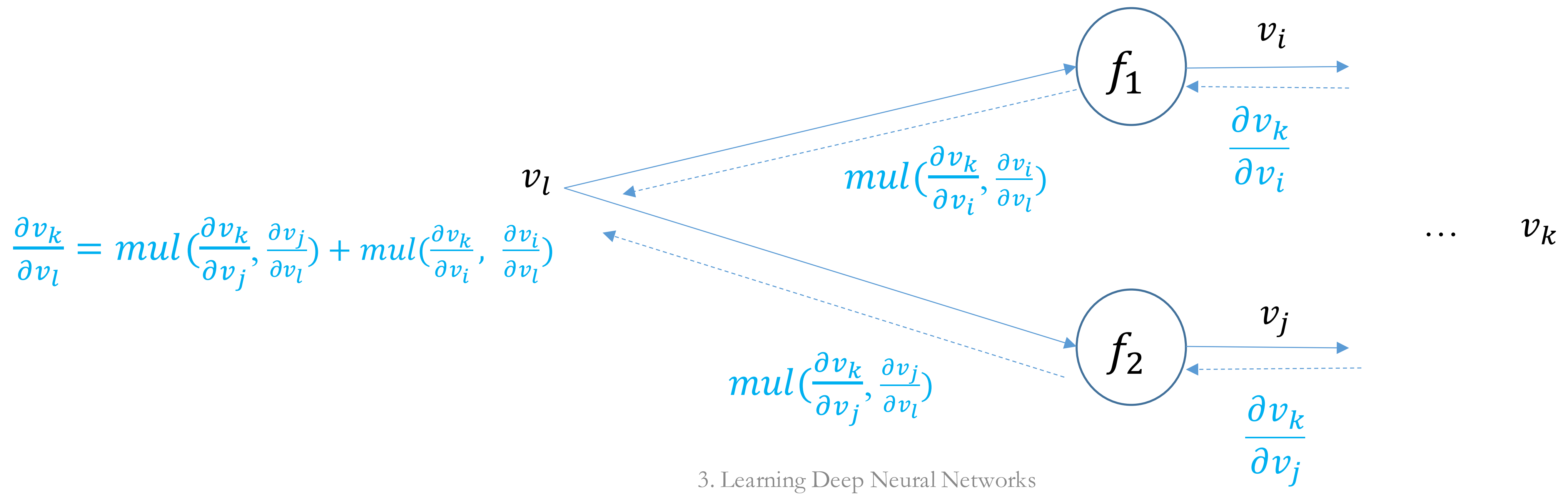


a
$\frac{\partial c}{\partial a} = 2 + 2 = 4$
$2*$
v_1
$\frac{\partial c}{\partial v_1}\frac{\partial v_1}{\partial a} = 1 * 2$
$\frac{\partial c}{\partial v_1} = 1$
$+$
c
1
v_2
$\hat{}2$
$\frac{\partial c}{\partial v_2}\frac{\partial v_2}{\partial a} = 1 * 2a = 2$
$\frac{\partial c}{\partial v_2} = 1$

Slide credit: Wang Wei

Chain rule III

- $v_i = f_1(v_l)$
- $v_j = f_2(v_l)$
- v_k is the final output



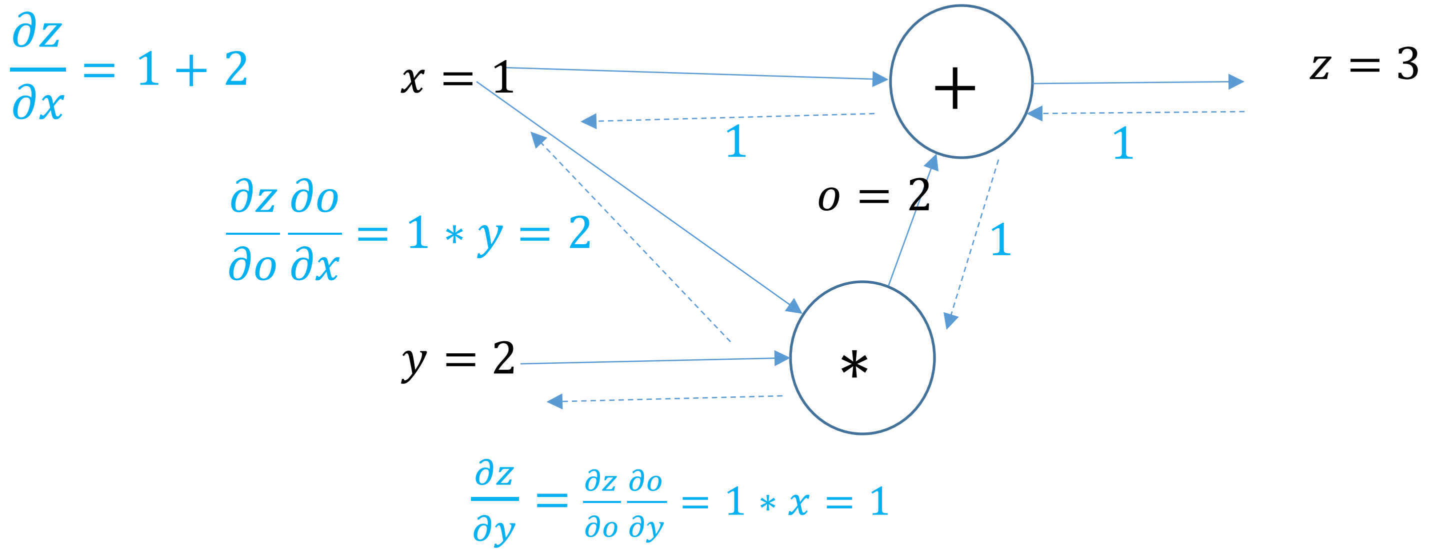
v_l
f_1
f_2
v_i
v_j
$\frac{\partial v_k}{\partial v_i}$
$\frac{\partial v_k}{\partial v_j}$
$mul(\frac{\partial v_k}{\partial v_i}, \frac{\partial v_i}{\partial v_l})$
$mul(\frac{\partial v_k}{\partial v_j}, \frac{\partial v_j}{\partial v_l})$
$\ldots$
v_k

$$\frac{\partial v_k}{\partial v_l} = mul(\frac{\partial v_k}{\partial v_j}, \frac{\partial v_j}{\partial v_l}) + mul(\frac{\partial v_k}{\partial v_i}, \frac{\partial v_i}{\partial v_l})$$

Slide credit: Wang Wei

Practice

- $z = f(x, y) = x * y + x$
- Given $x = 1, y = 2,$ compute $\frac{\partial z}{\partial x}, \frac{\partial z}{\partial y}$?

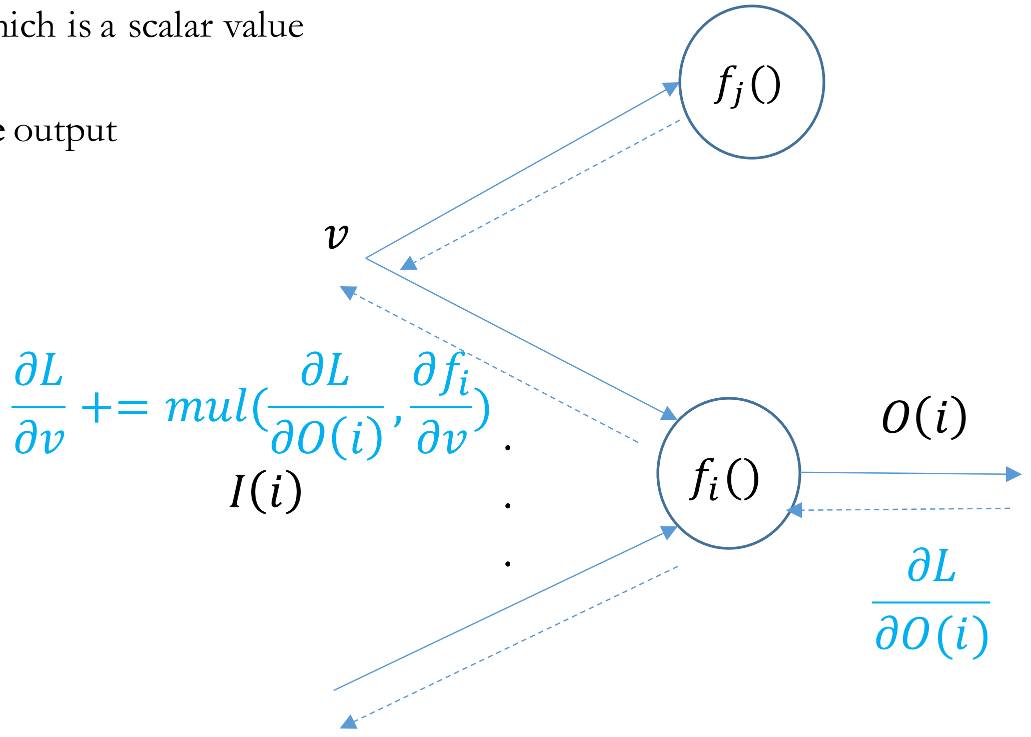


$\frac{\partial z}{\partial x} = 1 + 2$
$x = 1$
$+$
$z = 3$
1
1
$o = 2$
$\frac{\partial z}{\partial o}\frac{\partial o}{\partial x} = 1 * y = 2$
1
$y = 2$
$*$
$\frac{\partial z}{\partial y} = \frac{\partial z}{\partial o}\frac{\partial o}{\partial y} = 1 * x = 1$

Slide credit: Wang Wei

Backpropagation (BP)

- Computation graph
  - Denote all variables as $v_1, v_2, \dots$ sorted in topological order
    - The last variable is the loss (L) for neural network models, which is a scalar value
  - Denote all operations as $f_1, f_2, \dots f_k$
    - Each operation accepts multiple inputs and generate **a single** output
      - which may be used by multiple other operations
    - $I(i)$ denotes all the input variables to f_i
    - $O(i)$ denotes the single output variable of f_i
- Forward pass
  - For i = 1, 2, … k
    - Run operation $f_i: I(i) \rightarrow O(i)$
- Backward pass
  - Initialize $\frac{\partial L}{\partial v} = 0$ for all v
  - For i = k, … 1 Note the reverse order!
    - For each $v \in I(i)$ Compute $\frac{\partial L}{\partial v} += mul(\frac{\partial L}{\partial O(i)}, \frac{\partial f_i(I(i))}{\partial v})$



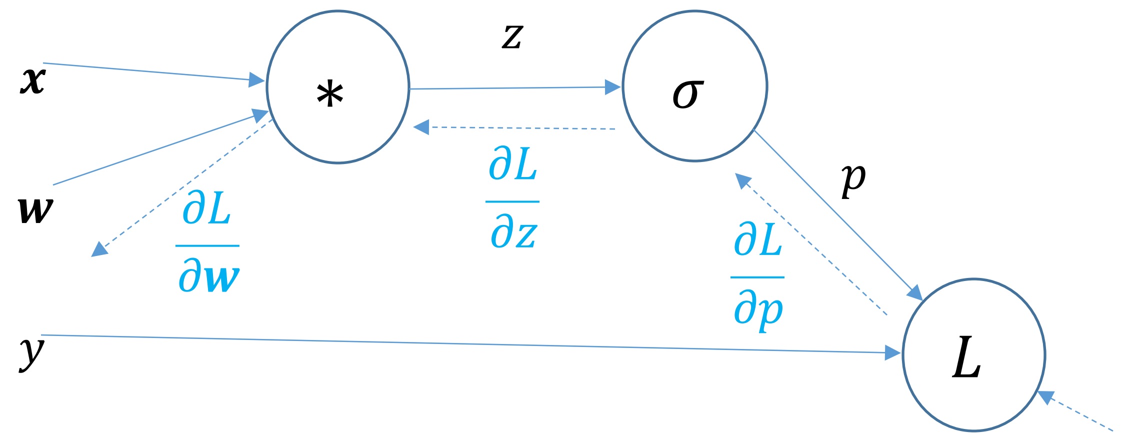
$f_j()$
v
$\frac{\partial L}{\partial v} += mul(\frac{\partial L}{\partial O(i)}, \frac{\partial f_i}{\partial v})$
$I(i)$
$f_i()$
$O(i)$
$\frac{\partial L}{\partial O(i)}$

Slide credit: Wang Wei

Logistic regression

- $z = \boldsymbol{w}^T \boldsymbol{x}, \boldsymbol{x} \in R^{n\times 1}, \boldsymbol{w} \in R^{n\times 1}$
- $p = \sigma(z)$
- $L(p, y) = -ylogp - (1-y)\log(1-p)$
- Dot
  - Forward$(\boldsymbol{x}, \boldsymbol{w})$: $\boldsymbol{w}^T \boldsymbol{x}$
  - Backward$(dz, \boldsymbol{x}, \boldsymbol{w})$: $dz\ \boldsymbol{x}$
- Logistic
  - Forward(z): $\sigma(z)$
  - Backward(dp, z) : $\mathrm{p} = \sigma(z);\ dp * p * (1-p) = (-\frac{y}{p} + \frac{1-y}{1-p}) * p * (1-p) = p - y$
- Binary-Cross-entropy
  - Forward(p, y): $-ylogp - (1-y)\log(1-p)$
  - Backward$([\mathrm{dL}], p, y)$: $-\frac{y}{p} + \frac{1-y}{1-p}$

dL is redundant since it is 1



$\boldsymbol{x}$
$\boldsymbol{w}$
y
$*$
z
σ
p
L
$\frac{\partial L}{\partial \boldsymbol{w}}$
$\frac{\partial L}{\partial z}$
$\frac{\partial L}{\partial p}$

$\frac{\partial L}{\partial L} = 1$

$$dz = \frac{\partial L}{\partial z}, \qquad dp = \frac{\partial L}{\partial p}$$

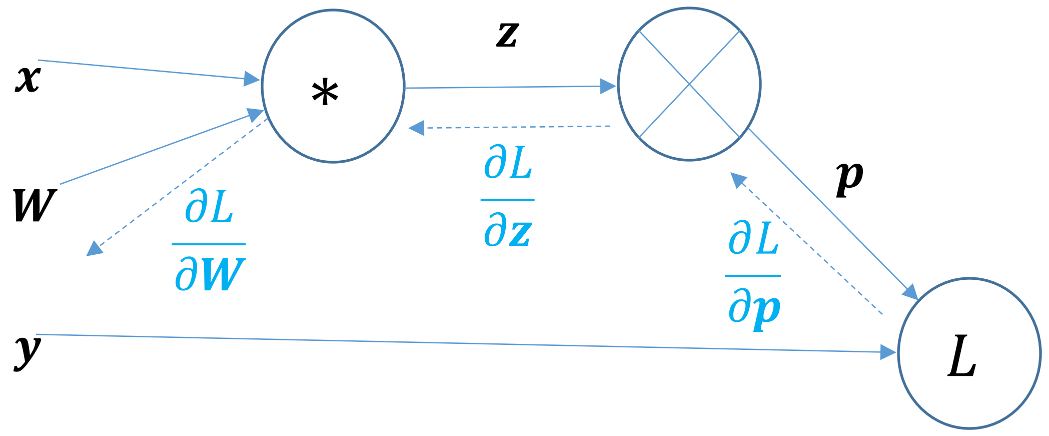
This operation can also be further decomposed; but since binary cross-entropy is used so commonly, we treat it as a whole stand-alone unit.

Slide credit: Wang Wei

Softmax regression

- $\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x},$ $\quad \boldsymbol{x} \in R^n, W \in R^{k*n}$
- $\boldsymbol{p} = softmax(\boldsymbol{z})$ $\quad \boldsymbol{p} \in R^k$
- $L(\boldsymbol{p}, \boldsymbol{y}) = \sum_{i=1} -y_i \log p_i$

- Matmul matrix multiplication
  - Forward$(\boldsymbol{x}, \boldsymbol{W})$: $\boldsymbol{W}\boldsymbol{x}$
  - Backward$(\boldsymbol{dz}, \boldsymbol{x}, \boldsymbol{W})$
- Softmax
  - Forward$(\boldsymbol{z})$: $\boldsymbol{p}$
  - Backward$(\boldsymbol{dp}, \boldsymbol{z})$
- Cross-entropy
  - Forward$(\boldsymbol{p}, \boldsymbol{y})$: $\sum_{i=1} -y_i \log p_i$
  - Backward$(\boldsymbol{p}, \boldsymbol{y})$

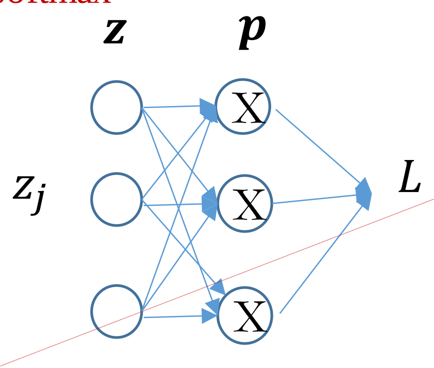


$\boldsymbol{x}$
$\boldsymbol{W}$
$\boldsymbol{y}$
$*$
$\boldsymbol{z}$
$\boldsymbol{p}$
L
$\frac{\partial L}{\partial \boldsymbol{W}}$
$\frac{\partial L}{\partial \boldsymbol{z}}$
$\frac{\partial L}{\partial \boldsymbol{p}}$

Slide credit: Wang Wei

Cleaner and computationally cheaper to consider the softmax and cross-entropy together!

$$\frac{\partial L}{\partial \mathbf{z}} = \underbrace{\frac{\partial L}{\partial \mathbf{p}}}_{\text{easy}}\underbrace{\frac{\partial \mathbf{p}}{\partial \mathbf{z}}}_{\text{messy matrix, since num \& denom are vectors}} \quad \boxed{= \mathbf{p} - \mathbf{y}}$$



$$\frac{\partial L}{\partial p_i} = -\frac{y_i}{p_i}, \qquad \frac{\partial L}{\partial \mathbf{p}} = -\frac{\mathbf{y}}{\mathbf{p}}$$

$$\frac{\partial L}{\partial z_j} = \sum_i \frac{\partial L}{\partial p_i}\frac{\partial p_i}{\partial z_j}$$

$$= \frac{\partial L}{\partial p_j}\frac{\partial p_j}{\partial z_j} + \sum_{i\neq j}\frac{\partial L}{\partial p_i}\frac{\partial p_i}{\partial z_j}$$

$$= -\frac{y_j}{p_j}(p_j - p_j^2) + \sum_{i\neq j} -\frac{y_i}{p_i}(-p_i p_j)$$

$$= -y_j(1-p_j) + \sum_{i\neq j} y_i p_j$$

$$= -y_j + y_j p_j + p_j \sum_{i\neq j} y_i \qquad \left(\sum y_i = 1\right)$$

$$= -y_j + y_j p_j + p_j(1 - y_j) = \boxed{p_j - y_j}$$

$$p_i = \frac{e^{z_i}}{\sum_k e^{z_k}}$$

$$h(x) = \frac{f(x)}{g(x)} \rightarrow h'(x) = \frac{f'(x)g(x) - f(x)g'(x)}{g^2(x)}$$

$$\frac{\partial \sum_k e^{z_k}}{\partial e^{z_j}} = \frac{\partial e^{z_j}}{\partial e^{z_j}} = e^{z_j}$$

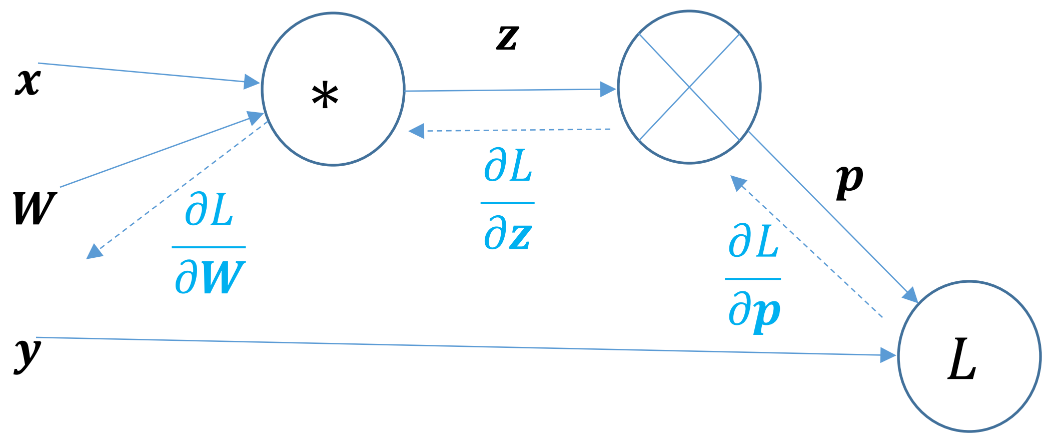
$$\frac{\partial p_i}{\partial z_j} = \begin{cases} i = j, \dfrac{e^{z_j}\sum_k e^{z_k} - e^{z_j}e^{z_j}}{\left(\sum_k e^{z_k}\right)^2} = p_j - p_j^2 \\ i \neq j, \dfrac{0\sum_k e^{z_k} - e^{z_i}e^{z_j}}{\left(\sum_k e^{z_k}\right)^2} = -p_i p_j \end{cases}$$

Refer to this website for the derivation process

Slide credit: Wang Wei

Softmax regression

- $\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x},$ $\quad\quad \boldsymbol{x} \in R^n, W \in R^{k*n}$
- $\boldsymbol{p} = softmax(\boldsymbol{z}) \quad \boldsymbol{p} \in R^k$
- $L(\boldsymbol{p}, \boldsymbol{y}) = \sum_{i=1} -y_i \log p_i$



$\boldsymbol{x}$
$\boldsymbol{W}$
$\boldsymbol{y}$
$*$
$\boldsymbol{z}$
$\boldsymbol{p}$
L
$\frac{\partial L}{\partial \boldsymbol{W}}$
$\frac{\partial L}{\partial \boldsymbol{z}}$
$\frac{\partial L}{\partial \boldsymbol{p}}$

L is a complex node

- Matmul
  - Forward($\boldsymbol{x}, \boldsymbol{W}$): $\boldsymbol{W}\boldsymbol{x}$
  - Backward($d\boldsymbol{z}, \boldsymbol{x}, \boldsymbol{W}$): $d\boldsymbol{z}\ \boldsymbol{x}^T$
- Softmax-Cross-Entropy
  - Forward($\boldsymbol{z}$): $\boldsymbol{p}$; $\sum_{i=1} -y_i \log p_i$
  - Backward($\boldsymbol{z}, \boldsymbol{y}$): $\boldsymbol{p} - \boldsymbol{y}$

Slide credit: Wang Wei

BP for Multiple examples

So far, we've considered backpropagation for a single example.
How should we handle all of our training samples?

- Approach I (individual)
  - Backpropagation for each example separately
  - Average the gradients across all examples
- Approach II (vectorized)
  - Use matrix (one row per example) in the BP
  - Average the loss
  - Compute gradient based on averaged loss

$$\begin{tikzcd}
x_1 \arrow[r, "BP"] & \partial L_1/\partial w \arrow[dr] & \\
x_2 \arrow[r, "BP"] & \partial L_2/\partial w \arrow[r] & \text{avg} \\
x_3 \arrow[r, "BP"] & \partial L_3/\partial w \arrow[ur] &
\end{tikzcd}$$

$$\begin{tikzcd}
x_1 \arrow[dr] & & & \\
x_2 \arrow[r] & \mathbf{X} \arrow[r, "BP"] & \mathbf{L} \arrow[r] & \partial L_{avg}/\partial w \\
x_3 \arrow[ur] & & &
\end{tikzcd}$$

Which approach is better and why?

Slide credit: Wang Wei

BP operations

These are all derived via chain rule

- Add_bias e.g. Z = XW + b
  - $\boldsymbol{A}$ is a matrix; b is a row vector
  - Forward($\boldsymbol{A}, b$): $\boldsymbol{C} = A + b$
  - Backward($d\boldsymbol{C}, \boldsymbol{A}, b$): $d\boldsymbol{A} = d\boldsymbol{C}$, $db = \mathbf{1}^T\, d\boldsymbol{C}$
- Array and scalar multiplication
  - $\boldsymbol{v}$ is an array, k is a scalar
  - Forward($\boldsymbol{v}, k$): $c = k\, v$
  - Backward($d\boldsymbol{c}, \boldsymbol{v}, k$):$d\boldsymbol{v} = k\, d\boldsymbol{c}$

Slide credit: Wang Wei

BP operations

- Matmul matrix multiplication operation
  - $\boldsymbol{A} \in R^{m*k}, \boldsymbol{B} \in R^{k*n}$ (including matrix with a single column or row)
  - Forward$(\boldsymbol{A}, \boldsymbol{B})$: $\boldsymbol{C} = AB \in R^{m*n}$
  - Backward$(\boldsymbol{dC}, \boldsymbol{A}, \boldsymbol{B})$: $d\boldsymbol{A} = d\boldsymbol{C}\, \boldsymbol{B}^{\boldsymbol{T}}, d\boldsymbol{B} = \boldsymbol{A}^{\boldsymbol{T}} d\boldsymbol{C}$
- Logistic operation
  - a is an array of any shape
  - Forward(a): $b = \sigma(a)$
  - Backward(db, a): $da = db * \sigma(a) * (1 - \sigma(a))$
- Softmax-Cross-entropy operation
  - $Z \text{ and } Y \in R^{m*k}$ are matrix of the same shape; each row of a (or b) sums to 1
  - Forward $(\boldsymbol{Z}, \boldsymbol{Y})$: $\boldsymbol{P} = softmax(\boldsymbol{Z}); L = sum\ (-\boldsymbol{Y} log \boldsymbol{P})/m$
  - Backward$(\boldsymbol{Z}, \boldsymbol{Y})$: $d\boldsymbol{Z} = (\boldsymbol{P} - \boldsymbol{Y})/\mathrm{m}$

Slide credit: Wang Wei

Stochastic Gradient Descent

Gradient descent algorithm

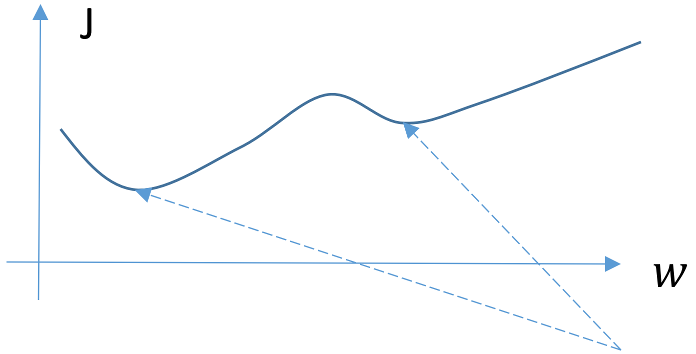
- Define the network architecture
- Select a proper loss function
- Initialize all model parameters (usually randomly)
- Repeat
    - Forward to compute the average loss across all training examples
    - Backward to compute the gradient of the loss w.r.t each parameter
        - $\frac{\partial J}{\partial w}, w \in \Phi$; Φ denotes all parameters of the model
    - Update: $w = w - \alpha \frac{\partial J}{\partial w}, w \in \Phi$

Slide credit: Wang Wei

Problem of GD: Local optimum

- For any $w \in \Phi$, $\frac{\partial J}{\partial w} = 0$, local minimum
- This case happens very rarely since:
  - Gradient should be 0
  - Local minimum / u-shape
  - Requires this to be true for all **w** in the network;

this happens when there is a u shape in the curve.



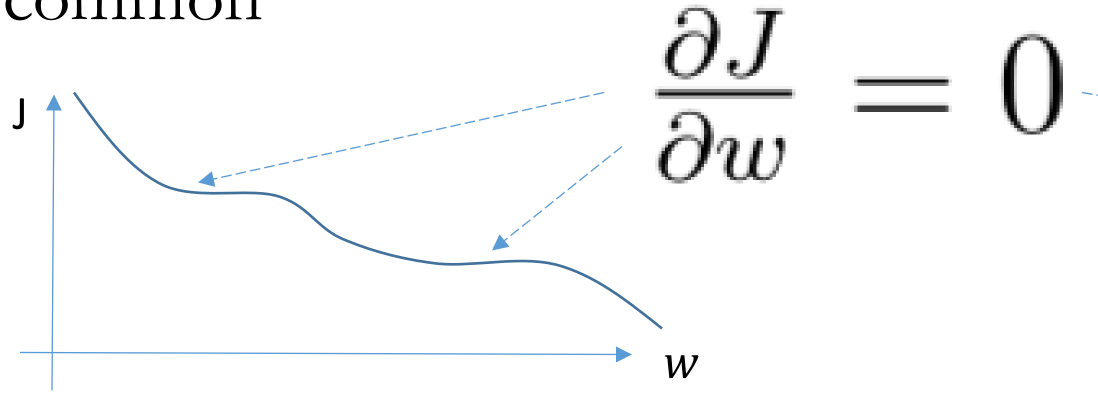
$$w = w - \alpha \frac{\partial J}{\partial w} = w - 0$$

get stuck at local optimum

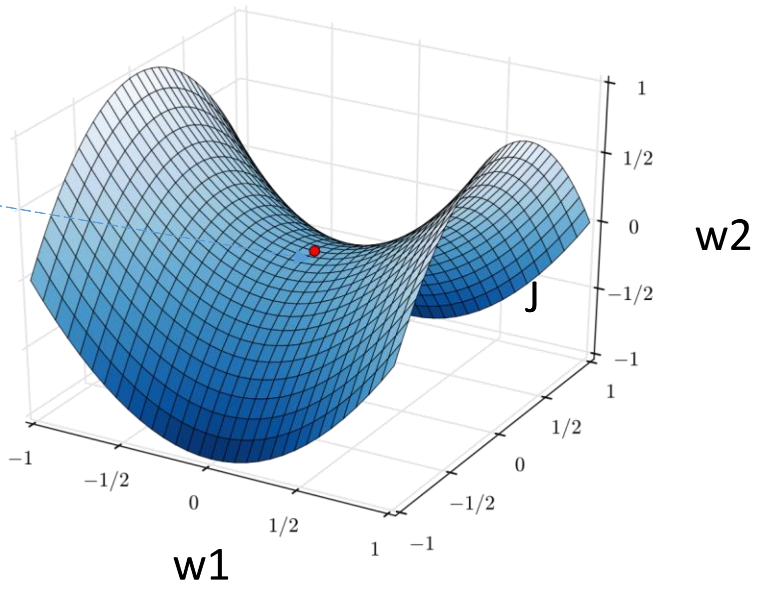
Slide credit: Wang Wei

Problem of GD: Saddle points [link]

- For any $w \in \Phi$, $\frac{\partial J}{\partial w} = 0$, but is not a local optimum
- More common



$$\frac{\partial J}{\partial w} = 0$$



from: https://en.wikipedia.org/wiki/Saddle_point

These two problems second problem highlights the difficulty of applying gradient descent to non-convex functions. We simply *don't* find the global optimum. Sometimes, we may not even arrive at a local optimum.

Slide credit: Wang Wei

Problem of GD: Efficiency

- GD has to load all training samples into memory to do BP
  - High memory cost
  - High computation cost per iteration

We can resolve this by making the gradient descent ***stochastic***.

Slide credit: Wang Wei

Stochastic gradient descent (SGD)

- Randomly initialize the parameters
- Repeat until converge
  - Randomly pick a (single) training sample
    - Compute the loss $J^{(i)}$
    - Compute the derivative of $J^{(i)}$ w.r.t each parameter w
    - Update all parameters $\mathrm{w} = \mathrm{w} - \alpha \frac{\partial J^{(i)}}{\partial w}$
- Advantage
  - More efficient in terms of memory and speed per iteration
  - $\frac{\partial J^{(i)}}{\partial w}$ may not be zero although $\frac{\partial J}{\partial w}$=0, $\frac{\partial J^{(i)}}{\partial w}$ may not be zero although $\frac{\partial J^{(i-1)}}{\partial w}$ =0
- Disadvantage
  - Not moving in the optimal direction for every step due to the stochastic process

SGD solves our problem with local optimum and saddle points. How?

We are effectively minimizing the loss for that single example.

Slide credit: Wang Wei

Mini-batch SGD

- Randomly initialize the parameters
- Repeat until converge
  - Randomly pick **b** training samples (called a mini-batch)
    - Compute the loss $J = \sum_{i=1}^{b} J^i / b$
    - Compute the derivative of J w.r.t each parameter w
    - Update each parameter $w = w - \alpha \frac{\partial J}{\partial w}$
- The derivative direction is more stable than SGD
  - Averaged over b samples → converges faster in terms of # iterations
- Efficiency
  - R for convergence rate (in proportion to inverse of total number iterations)
  - T for wall time per iteration (e.g. hours)

$$R_{GD} > R_{mini-SGD} > R_{SGD}$$
$$T_{GD} > T_{mini-SGD} > T_{SGD}$$

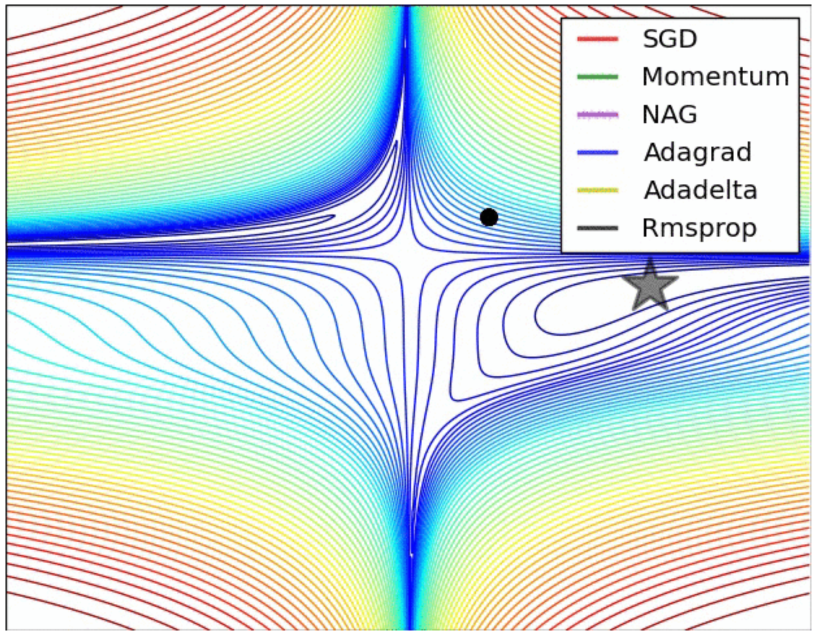
Slide credit: Wang Wei

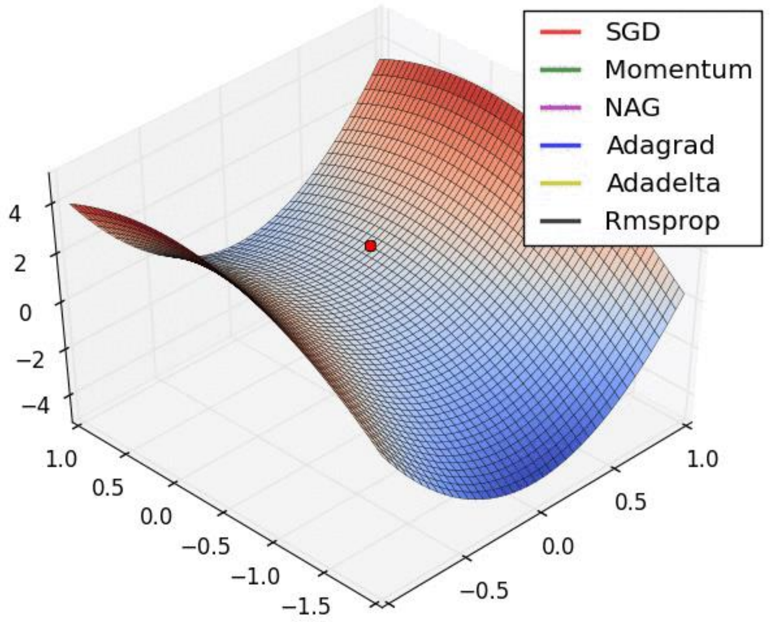
Batch Size: Small or Big?

- larger batches are more accurate at estimating the gradient, but returns are less than linear
  - In the beginning: 2X batch size, 2X fewer iterations
  - After that: 2X batch size, kX fewer iterations (k<2)
- multicore architectures are underutilized by extremely small batches with some absolute minimum batch size, below which there is a fixed time cost
- if all batch samples are processed in parallel (as is the case for GPUs), then the amount of memory scales w/ batch size, so one is limited by GPU memory
- some hardware have better runtime with specific array sizes, e.g. for GPUs, powers of 2 are usually optimal
- small batches act as regularizers with noisy approximated gradients;
- training with a small batch size may require smaller learning rates to maintain stability due to the high variance in the estimated gradients; this trade-off may slow down the overall learning speed

The Evolution of Gradient Descent

- Many variants of gradient descent [link]

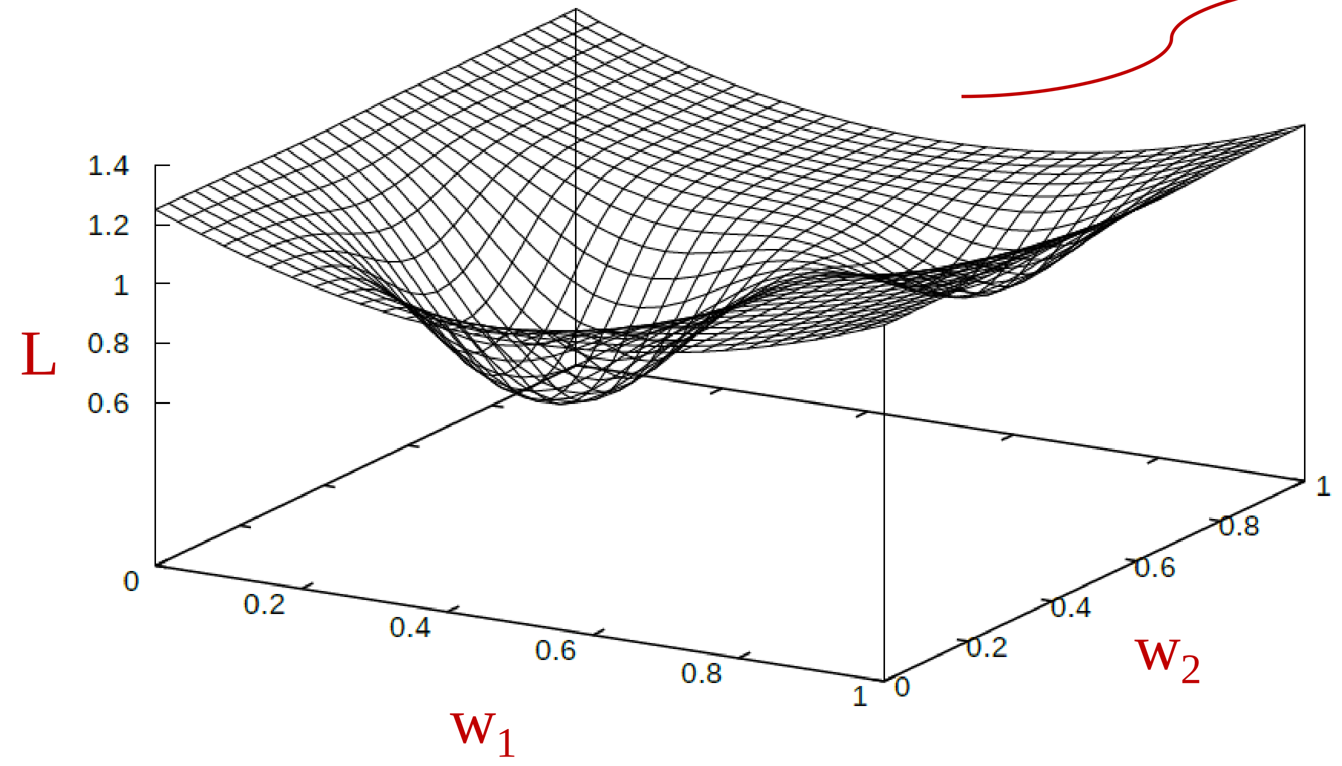


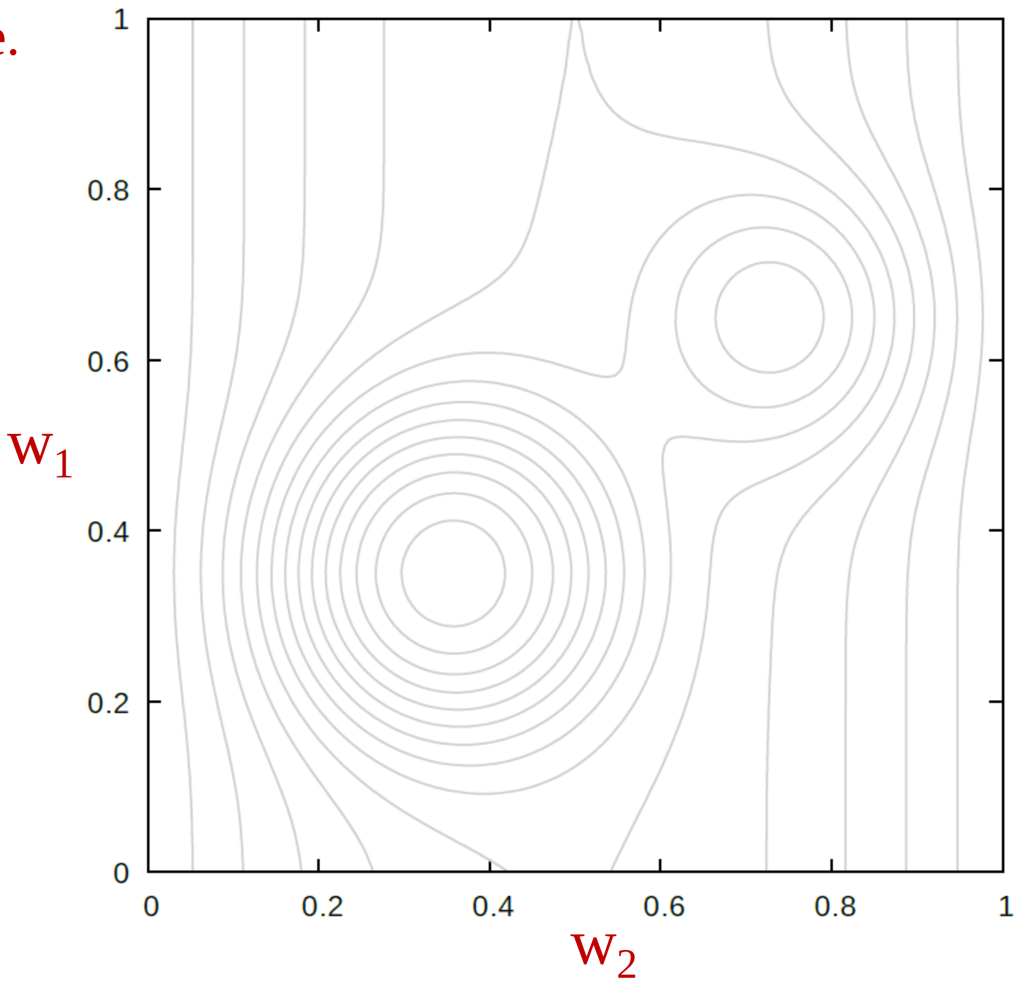


Adapted from Wang Wei

Loss contour

Flatten: all values on a given contour has the same loss value.





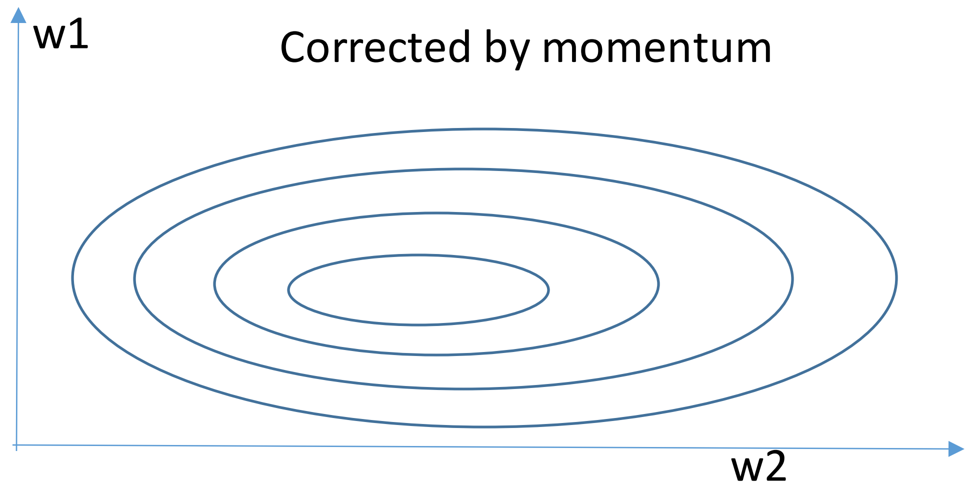
Consider the loss wrt 2 weights.

Images from: https://fleuret.org/ee559/

Mini-batch SGD with momentum

- Use history movement to correct or accelerate current movement

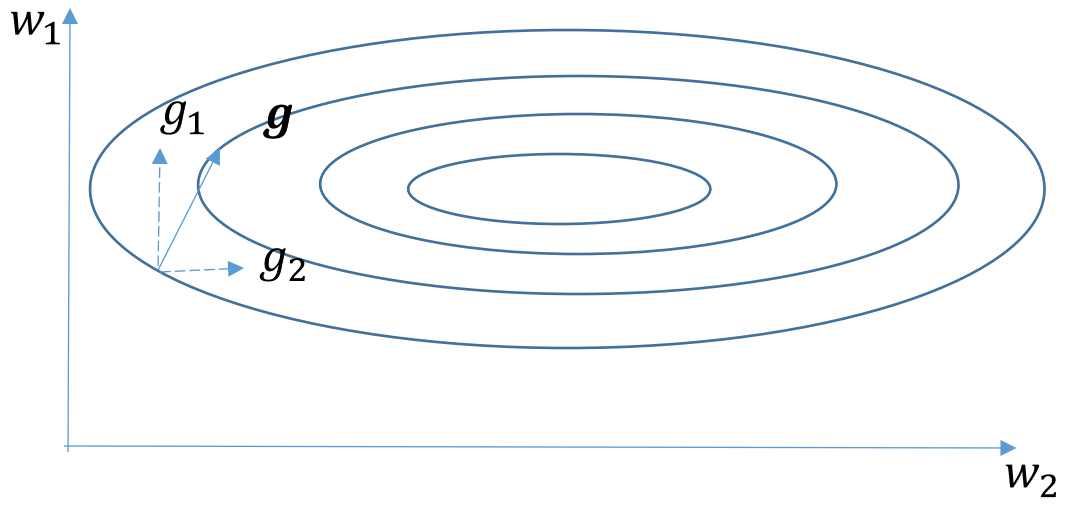
- Exponential average
  - For every parameter w, we maintain a v for it (initialized as 0)
  - $v = \beta v + g$ (e.g., $\beta = 0.9$)   accumulation
  - $\mathrm{w} = \mathrm{w} - \alpha v$



Slide credit: Wang Wei

RMSprop

- Problem
  - Moving along g_2 would decrease the loss faster, but g_2 is smaller than g_1
- Solution
  - For each parameter w, maintain a variable s (initialized as 0) to rescale the gradient g
  - $s = \beta s + (1 - \beta) g^2$ $(\beta = 0.9)$
  - $w = w - \alpha g / \sqrt{s + \epsilon}$ 10-7



moving average

Rescales learning rate to remove effect of gradient "size"
Makes a balance between different directions

Slide credit: Wang Wei

Diederik P. Kingma and Jimmy Lei Ba. Adam : A method for stochastic optimization. ICLR 2015.

Adam

Adam offers fast convergence, but sometimes not to good solutions [link]

- Combines momentum and RMSProp
  - $v = \beta_1 v + (1-\beta_1) g$ (e.g. β_1=0.9) moving average of 1st order gradient
  - $s = \beta_2 s + (1-\beta_2) g^2$ (e.g. β_2=0.999) moving average of 2nd order gradient
- $w = w - \alpha \frac{\hat{v}}{\sqrt{\hat{s}+\epsilon}}$
- $\hat{v}$ and $\hat{s}$ are bias-corrected for t-th iteration
  - $\hat{v} = \frac{v}{1-\beta_1^t}$, $\hat{s} = \frac{s}{1-\beta_2^t}$
  - mainly for the first few iterations;
    when t gets large, the denominator goes to 1

If $\beta_1 = 0.9$,

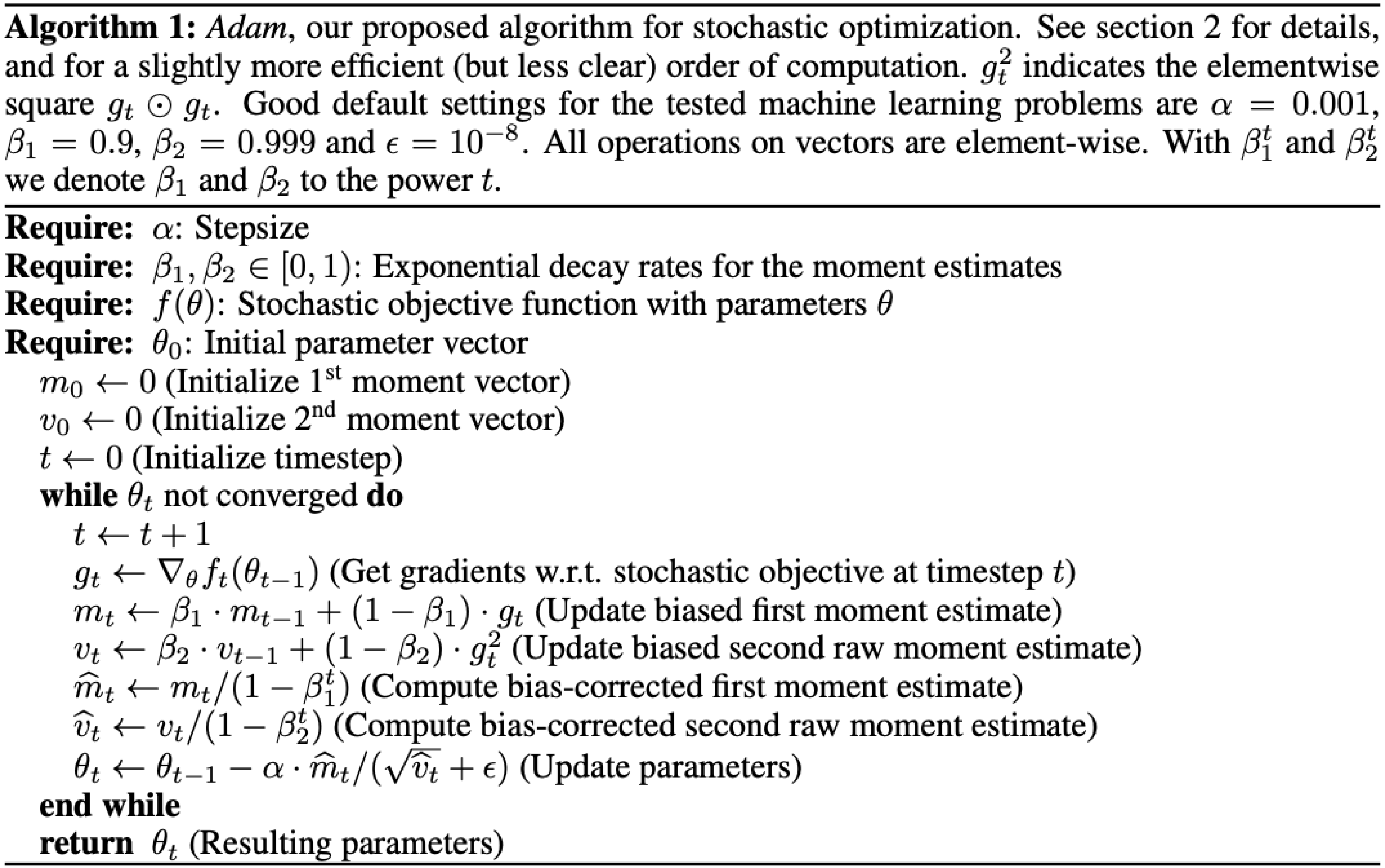
$$\begin{aligned} v_1 &= \beta_1\ v_0 + (1-\beta_1) g_1 \\ &= 0.9 v_0 + 0.1 g_1 \\ &= 0 + 0.1 g_1 = 0.1 g_1 \end{aligned}$$

$g_1 = 0.2$, then $v_1 = 0.02$

$\hat{v} = 0.02 / (1-0.9) = 0.02/0.1 = 0.2$

Adapted from Wang Wei

Adam



Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

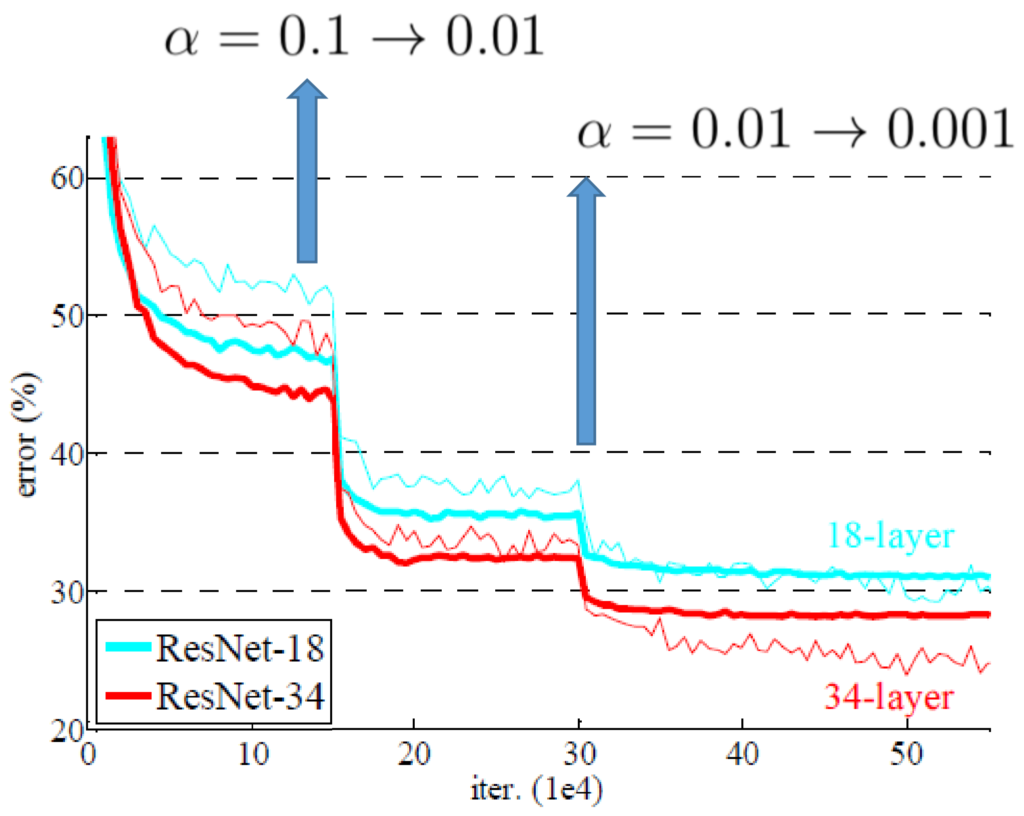
```
Require: $\alpha$: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters $\theta$
Require: $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while $\theta_t$ not converged do
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return $\theta_t$ (Resulting parameters)
```

Screenshot from Adam paper: https://arxiv.org/pdf/1412.6980.pdf

http://cs231n.github.io/neural-networks-3/

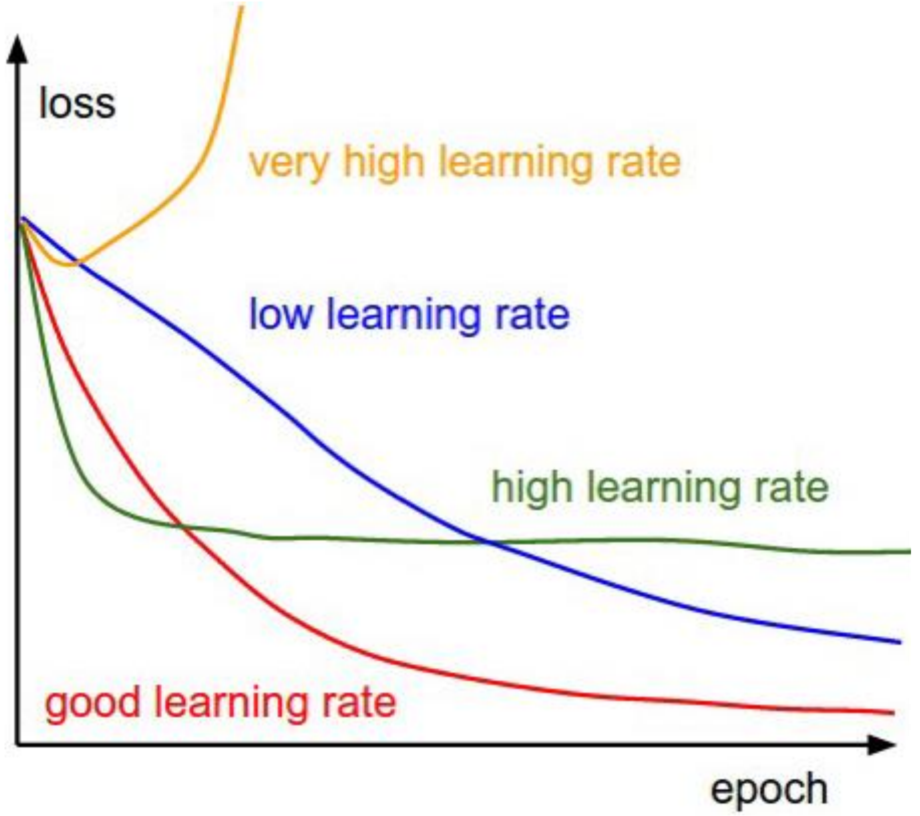
Learning rate

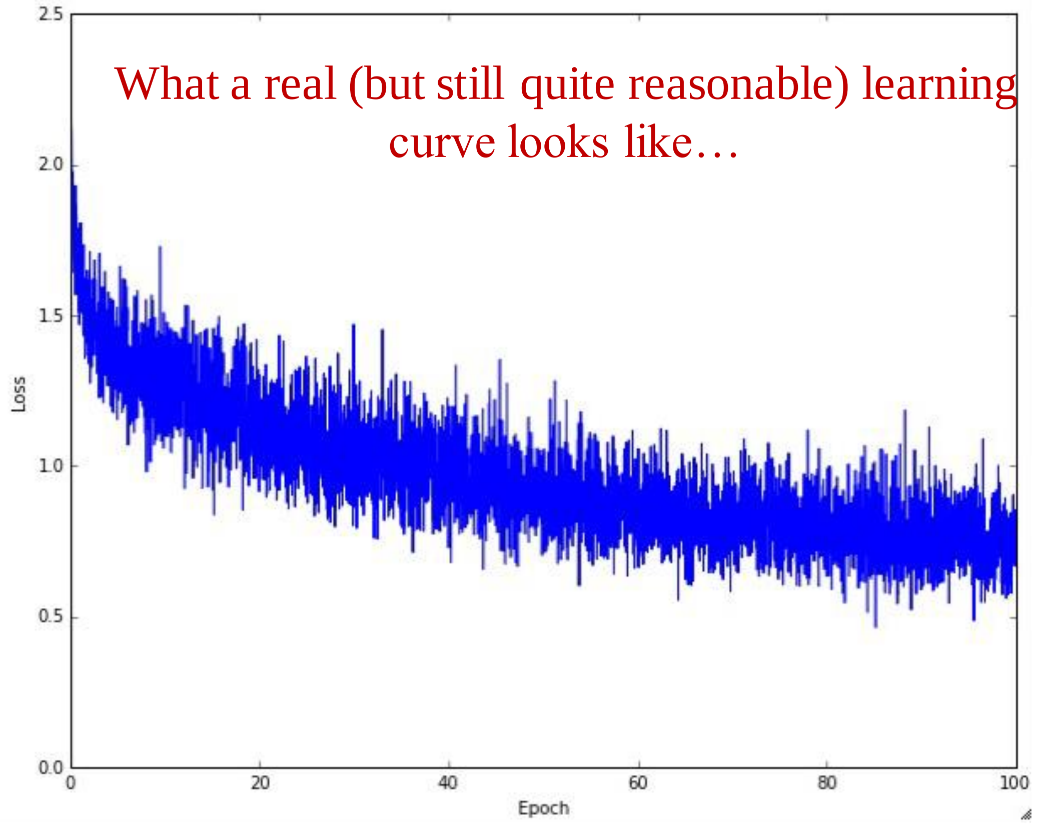
- Typical starting values: 0.1, 0.01
- Decay learning rate
  - Step
  - 1/t
  - Exponential, αe^{-kt}
- important for convergence!
  - If it is too large, w will oscillate around the optimal position.
  - If it is too small, it would take many iterations to reach the optimal position.
  - Initialize it with a large value and then decrease it gradually



From [2]

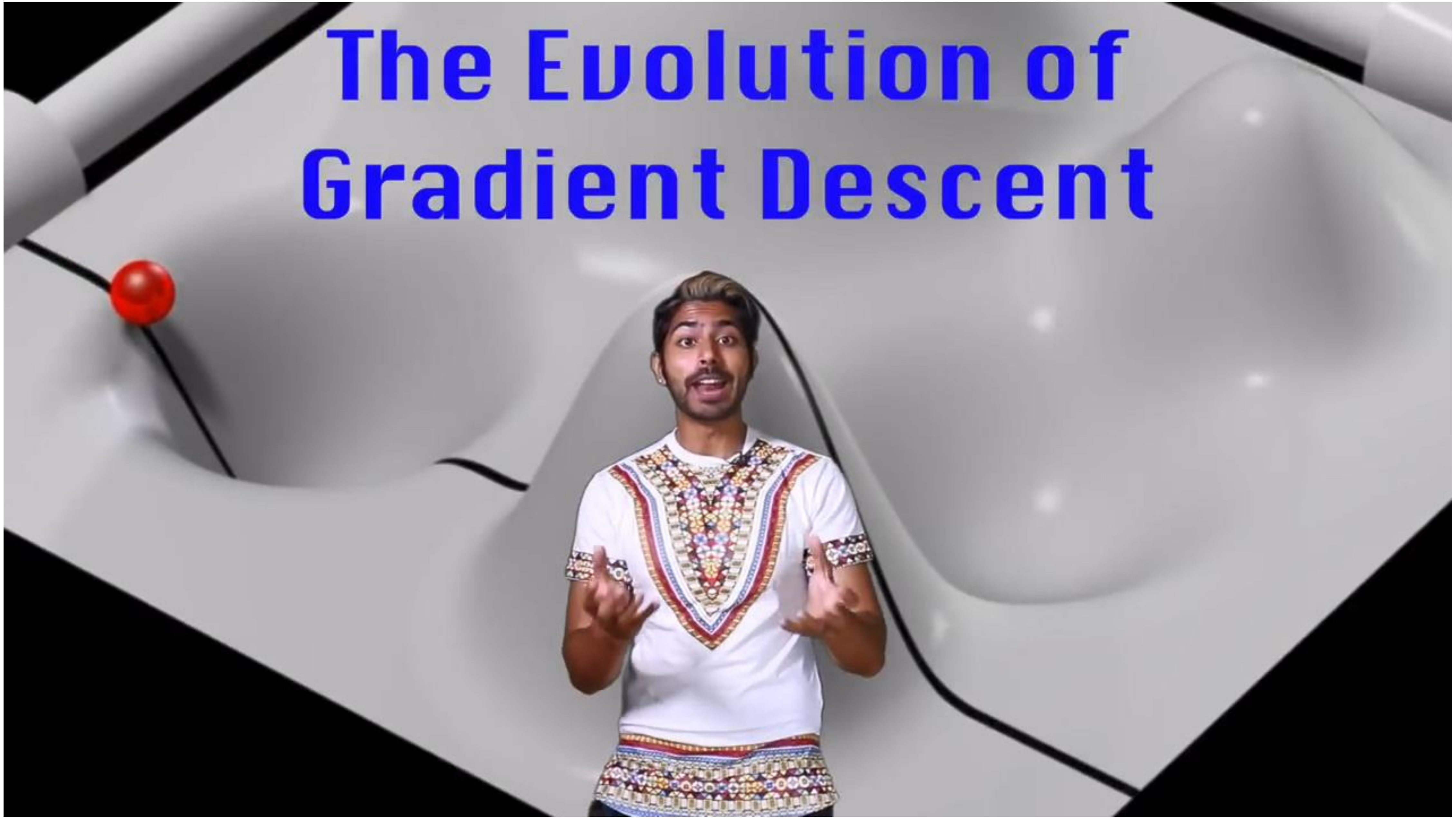
Impact of Learning Rate





Why does the loss increase with a very high learning rate?

http://cs231n.github.io/neural-networks-3/



https://www.youtube.com/watch?v=nhqo0u1a6fw

Tricks of the Trade

Random initialization, data normalization

Regularization (an early look).

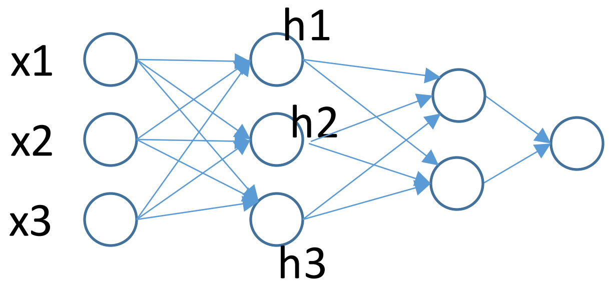
Random parameter initialization

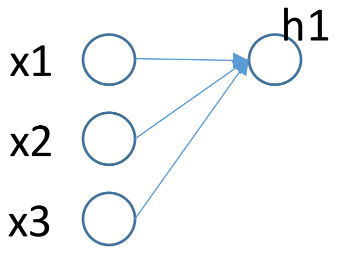
Why is random initialization so important?

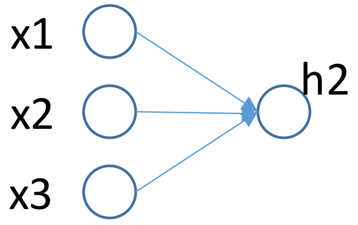
- Suppose that we had all elements of W were initialized as the same value (e.g. 0 or 1)
    - → all hidden units are the same
    - → derivatives of all hidden units are the same
    - → derivatives of weights connecting to xi are the same
    - → weights connecting to x_i have the same update

? problems

1. If all neurons in one layer are the same, then only one neuron is enough and all others are redundant → a very simple model
2. W's elements are always the same → redundant parameters







Slide credit: Wang Wei

Random Initializations Breaks Symmetry

- Initial parameters should break symmetry between different units.
- Two hidden units with the same activation function connected to the same inputs must have different initial parameters.
- If they were the same, then a deterministic learning algorithm, applied to a deterministic cost and model will update both units in the same way.
- If we want each unit to compute a different function (which we do!), then we should use random initialization.

https://towardsdatascience.com/neural-network-breaking-the-symmetry-e04f963395dd

How to Randomly Initialize?

- Weight matrix (W)
  - Gaussian, N(0, 0.01)
  - Uniform, U(-0.05, 0.05)
  - Glorot/Xavier
    - **Gaussian N(0, sqrt(2/(fan_in + fan_out))**
  - He/MSRA
    - **Gaussian N(0, sqrt(2/fan_in))**
  - Too small variance: **z** vanishing
  - Too large variance: **z** exploding

$z = XW + b, W \in \boldsymbol{R}^{n \times k}$

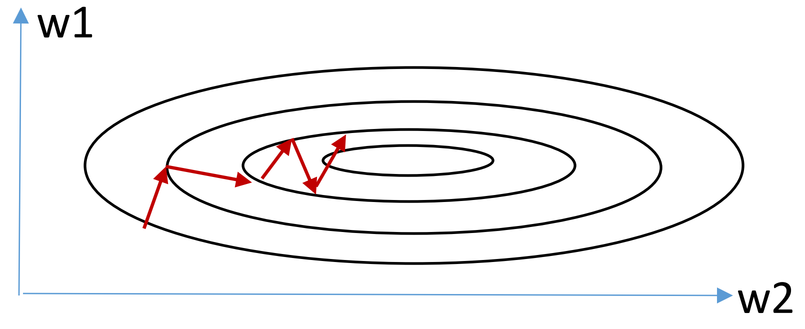
n: fan_in k: fan_out

General idea is to keep the variance of the random values within some reasonable range.
Detailed explanation given [here]

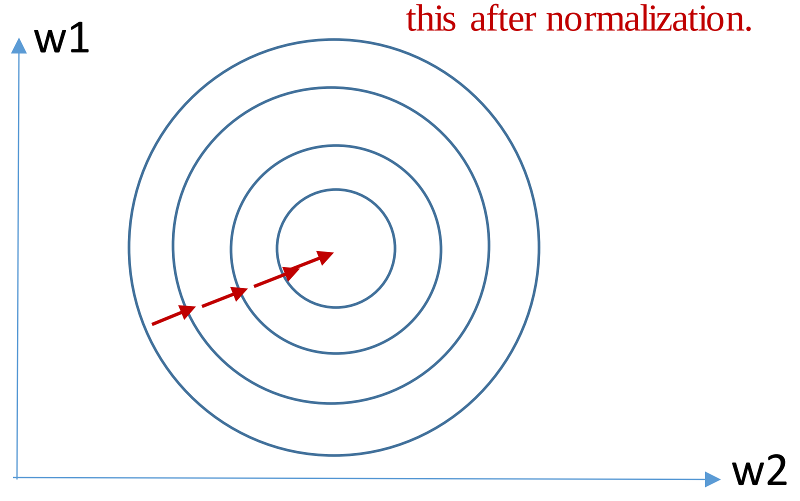
Slide credit: Wang Wei

Data normalization

- The features are at difference scale
  - E.g., attributes of a house varies in scale
  - The contours are ellipses
  - The gradient descent algorithm moves along the opposite of the gradient direction
  - **Not the optimal direction**
- Normalize the attributes into similar scale
  - [-1, 1], [0, 1]
  - $x_i = \frac{x_i - u_i}{\sigma_i}, x_i = \frac{x_i}{\max(x_i)}$
    - u_i, σ_i, max are computed over i-th feature of all samples from the training dataset



** this is just a conceptual visualization; it doesn't mean your loss contour looks like this after normalization.



Slide credit: Wang Wei

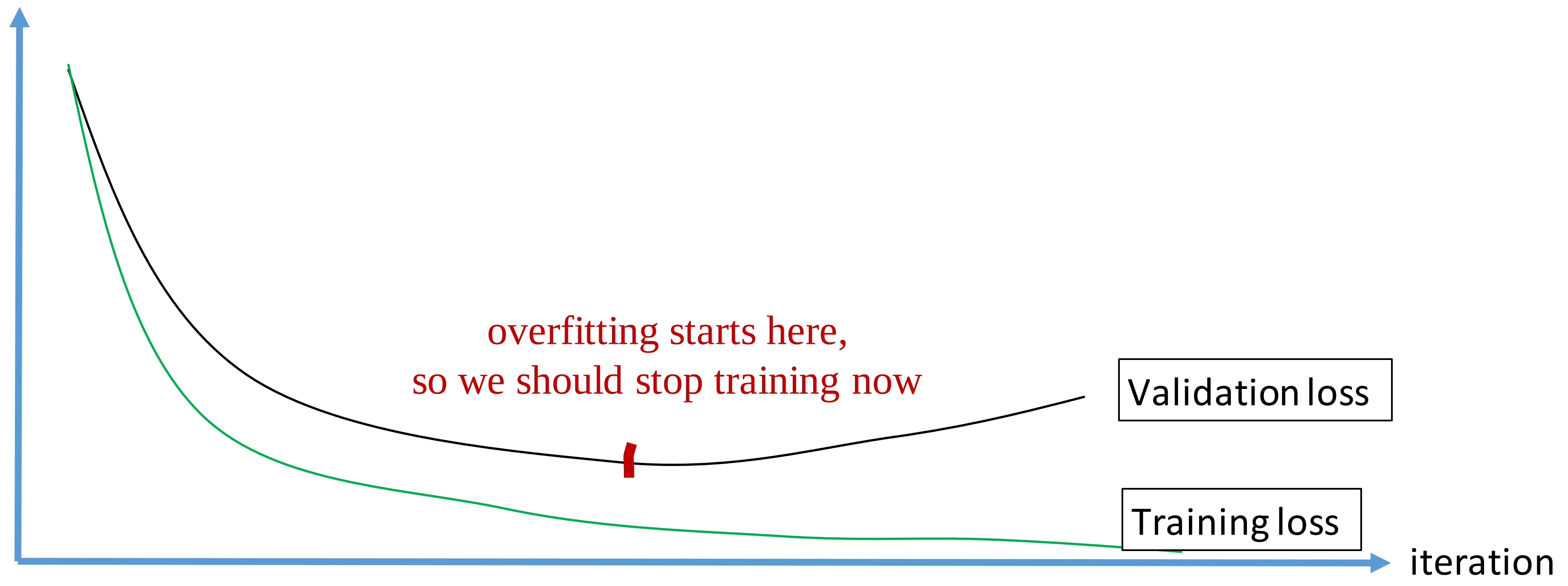
Regularization

** we will discuss this in much more details in subsequent lectures as well as additional regularization techniques

- $J_{reg} = J + \frac{\lambda}{2} ||\Phi||^2$, where Φ is the flatten vector of all parameters
- By minimizing the objective,
  - The parameters will be regularized/constrained to be small values
  - reduce overfitting
- For any $w \in \Phi$
  - $\frac{\partial J_{reg}}{\partial w} = \frac{\partial J}{\partial w} + \lambda w$

Slide credit: Wang Wei

Regularization --- early stopping



Slide credit: Wang Wei

Summary

- Backpropagation algorithm
  - A modular way to compute the gradient of the loss w.r.t parameters
  - Based on chain rules and matrix calculus
- Advanced gradient descent algorithms
  - SGD, mini-batch SGD
  - Momentum, RMSProp, Adam
- Some training tricks
  - Parameter initialization
  - Data normalization
  - Regularization and early stopping

Slide credit: Wang Wei