Course 2

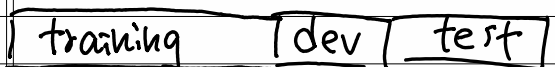
Hyperparameter Tuning

COURSE 2

- iterative process
  for hyperparameter tuning

idea → code → exp → idea

DATA



cross-validate

↑ for unbias testing

previous

70 / 30 or 60/20/20.

100 ~ 10000 samples

big data

1,000,000 data, 10,000 test might be enough

10.000

→ 98% / 1% / 1%

Mismatch train/test distribution

training
e.g cat on web

dev/test
cat on app

- try making dev/test with same distribut

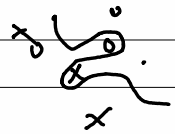
not having test set might be ok

BIAS & VARIANCE

- under fitting — high bias (not fitting well)
- just right
- over fitting — high variance (too flexible)

---

training set error:	1 %	15 %	15 %
dev set error :	11 %	16 %	30 %
	→ over fitting high variance	- under fit high bias	high bias var
			0.5 %
◁ high bias + high variance			1 %
			low bias var



RECIPE for ML

High bias → Bigger network
(training data problem) ↑ Train longer
(different NN architect)

↓

High variance → more data
(dev set problem) ↑ regularization
(different NN arch.)

* traditional method is difficult to improve bias AND variance!

high bias → more training data is not useful
high variance → regularization
more training data.

Regularization

Logistic Regre.. $w \in R^{n_x}$ $b \in R$

$\min_{w,b} J(w,b)$ λ: regularization para

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\|w\|_2^2 + \frac{\lambda}{2m} b$$

L_2 regu $\|w\|_2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$ (b: omit)

L_1 regularization $\|w\|_1$ $\frac{\lambda}{2m}\sum_{j=1}^{n_x} |w_j| = \frac{\lambda}{2m}\|w\|_1$

Neural Network

$$J(w^{[1]}, b^{[1]}, \ldots, w^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L} \|w^{[l]}\|^2$$

square norm

$$\|w^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l-1]}} \sum_{j=1}^{n^{[l]}} (w_{ij}^{[l]})^2 \quad w: (n^{[l]}, n^{[l-1]})$$

Frobenius form — Sum of square norm of matrix

$$dw^{[l]} = (\text{from back prop}) + \frac{\lambda}{m} w^{[l]}$$

$$w^{[l]} = w^{[l]} - \alpha\, dw^{[l]}$$

$$= w^{[l]} - \frac{\alpha\lambda}{m} w^{[l]} - \alpha\,(\text{from backprop})$$

weight decay

$$J(w^{[l]}, b^{[l]}) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\|w^{[l]}\|^2$$

$\lambda \uparrow \Rightarrow w \downarrow$ (bcz we penalize high cost)

$\Rightarrow$ simpler NN

Dropout regularization

- dropout random nodes in NN (hidden unit)

INVERTED DROPOUT

eg at layer 3,

keep-drop = 0.8

d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep-prop

a3 = np.multiply(a3, d3)

* ⇒ a3 /= keep-prop

e.g. total 50 units. shut off 10 units (20%)

$z^{[3]} = w^{[3]} \cdot a^{[3]} + b^{[3]}$

↑ bump up by 20%

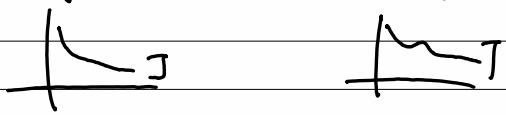
* no drop-out at test time

Why drop-out works?

- intuition → can't rely on one feature, so we spread out weights.
- keep-drop can vary in layers → shrink weights similar to L_2

- Computer Vision uses drop out bc high dimension

- Downside. J might looks weird in the plot

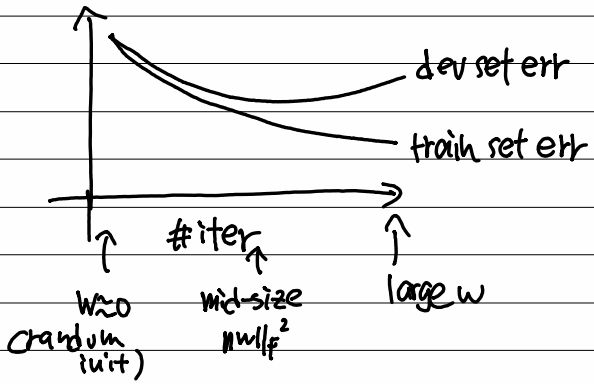


Regularization methods

Data augmentation

- flipping, random crop images. (flipping cats)
- adding distortion to characters etc

Early stopping



↓ breaks orthogonalization

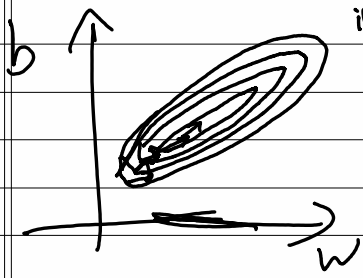
Normalizing Input.

subtract mean
normalize variance

$$x = x - \mu$$
$$x /= \sigma^2$$

* use same μ & σ^2 for testing / eval / train sets

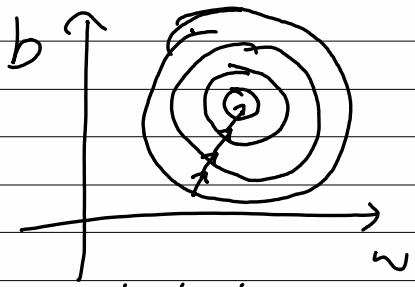
unnormalized



w, b might be in very diff range

require small learning rate for gradient descent

normalized



better!
safer to take larger step!

vanishing / exploding gradient

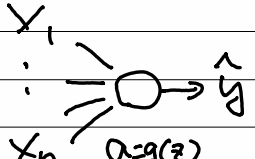
(assuming linear activation) back-prop

$\hat{y} = w^{[L]} \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{L-1} X$ - exploding

or

$\hat{y} = w^{[L]} \begin{pmatrix} 0.5 & 0 \\ 0 & 0.5 \end{pmatrix}^{L-1} X$ - vanishing

solution (initialization)



$z = w_1 x_1 + \dots + w_n x_n$ ~~$+b$~~

large n → small w_i

$Var(w_i) = \frac{1}{n}$ if ReLU, $Var(w_i) = \frac{2}{n}$

$w^{[l]} = np.random.rand(slope) \times np.sqrt(\frac{2}{n^{[l-1]}})$

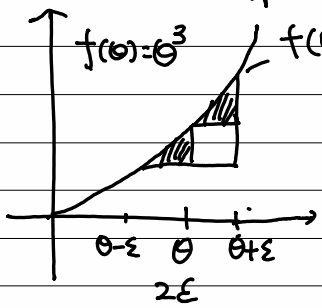
ReLU

$\tanh : \sqrt{\frac{1}{n^{[l-1]}}}$

or

Xavier initialization $\sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$

Numeric Approx of Gradients



$$\Delta \quad \frac{f(\theta+\varepsilon)-f(\theta-\varepsilon)}{2\varepsilon} \approx g(\theta)$$

$$\frac{(1.01)^3-(0.99)^3}{2\cdot 0.01}=3.0001\approx 3$$

app error $= 0.0001$ ✓

$$f'(\theta)=\lim_{\varepsilon\to 0}\frac{f(\theta+\varepsilon)-f(\theta-\varepsilon)}{2\varepsilon} \quad O(\varepsilon^2)$$

$$f'(\theta)\approx\lim_{\varepsilon\to 0}\frac{f(\theta+\varepsilon)-f(\theta)}{\varepsilon} \quad \text{err: } O(\varepsilon)$$

△ we use **two-sided ε** for gradient checking (more accurate) instead of one-sided checking

DEBUG TOOL

Gradient Checking (Grad Check)

$$J(W^{[1]}, b^{[1]} \dots W^{[n]} b^{[n]}) = J(\theta)$$

$$J(dW^{[1]}, db^{[1]} \dots dW^{[n]}, db^{[n]}) = J(d\theta)$$

for each i

$$d\theta_{approx}^{[i]} = \frac{J(\theta_1 \dots \theta_i+\varepsilon, \dots) - J(\theta_1 \dots \theta_i-\varepsilon \dots)}{2\varepsilon}$$

$$\approx d\theta^{[i]} = \frac{\partial J}{\partial \theta_i} \Rightarrow \text{check } d\theta_{approx} \overset{?}{\approx} d\theta \; ?$$

$$\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2} \approx$$

10^{-7} - great

10^{-5}

10^{-3} - worry! bug!

Grad Check Implementation

- Don't use in training
- look at component to try to identify the bug
- remember regularization
- doesn't work with dropout
  - keep-prop = 1 for debug
- run at random initialization

mini-batch gradient descent

split training set into batches

$X^{\{t\}}, Y^{\{t\}}$

}

batch v.s. mini-batch
(all) (small batches)

△ notation

$X^{(i)}$

$X^{[l]}$

$X^{\{t\}}$ – curly bracket.

training set batch

EXAMPLE

5,000,000 data → 5000 mini-batch, each has 1000 data.

for t = 1 to 5000

forward prop on $X^{\{t\}}$

cost $J^{\{t\}} = \frac{1}{1000} \sum L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 \cdot 1000} \|w^{[l]}\|_F^2$

backprop using $J^{\{t\}}$

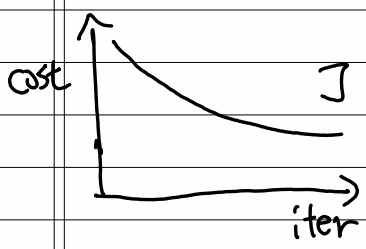
$w^{[l]} = w^{[l]} - \alpha dw^{[l]}$ $b^{[l]} = b^{[l]} - \alpha db^{[l]}$

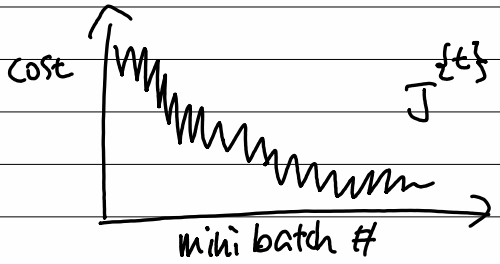
1 epoch – a single pass of 'All' data

BATCH – 1 step descent

MINIBATCH – 5000 step descent.

WHY MINI-BATCH WORK?





noisy, but trend down
bc? batch might be difficult

mini-batch size

size = m → batch gradient descent
size = 1 → stochastic gradient descent.

BATCH - too long per iteration
STOCHASTIC - lose the benefit of vectorization

GUIDE:

small traing set:
(~2000) use batch gradient

typical size
64 ~ 512
make sure mini-batch fits CPU/GPU memory.

Exponentially Weighted Average

moving

$$V_t = \beta V_{t-1} + (1-\beta)\theta_t$$

V_t is approx average over $\frac{1}{1-\beta}$ samples

$\beta = 0.9$

e.g. $\beta = 0.9 \rightarrow$ 10 samples

$\beta = 0.98 \rightarrow$ 50 samples

$V_{100} = 0.1\theta_{100} + 0.9 V_{99}$

$V_{99} = 0.1\theta_{99} + 0.9 V_{98}$

$V_{98} = 0.1\theta_{98} + 0.9 V_{97}$

$\vdots$

$V_{100} = 0.1\theta_{100} + 0.9 V_{99}$

$= 0.1\theta_{100} + 0.9(0.1\theta_{99} + 0.9 V_{98})$

$= 0.1\theta_{100} + 0.1 \cdot 0.9\,\theta_{99} + 0.9^2(0.1\theta_{98} + 0.9 V_{97})$

$= 0.1\theta_{100} + 0.1 \cdot 0.9\,\theta_{99} + 0.1(0.9)^2\theta_{98} + (0.9)^3 V_{97} + \dots$

$(0.9)^{10} \approx 0.35 \approx \frac{1}{e}$ after $\frac{1}{e}$ it drops faster.

$(0.98)^{50} \approx \frac{1}{e}$

Bias Correction in EWA

bias because $V_0 = 0$ $\quad v_t = \beta v_{t-1} + (1-\beta)\theta_t$

incorrect v in the beginning of series

$V_0 = 0$

$v_1 = 0.9 v_0 + 0.1\theta_1 = 0.1\theta_1$

$v_2 = 0.9 v_1 + 0.1\theta_2$

$= 0.09\theta_1 + 0.1\theta_2$

$\vdots$

Correction

$\frac{v_t}{1-\beta^t}$, correct in the beginning

$t \uparrow$, $\beta^t \sim 0$, correction does not affect v_t.

Gradient Descent with Momentum

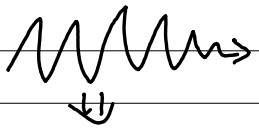
Compute EWA of gradient, then update

on iter t

compute dw db on current mini-batch

$V_{dw} = \beta V_{dw} + (1-\beta) dw$

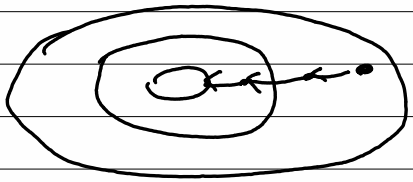
$V_{db} = \beta V_{db} + (1-\beta) db$



$W = W - \alpha V_{dw}$

$B = B - \alpha V_{db}$.

dampen oscillation overshoot.



ball rolling.

$V_{dw} = \beta V_{dw} + (1-\beta) dw$

$V_{db} = \beta V_{db} + (1-\beta) db$ acceleration

velocity

friction

hyper parameters

α - learning rate

β - 0.9 usually works

RMSprop (root mean square prop)

On iter t.

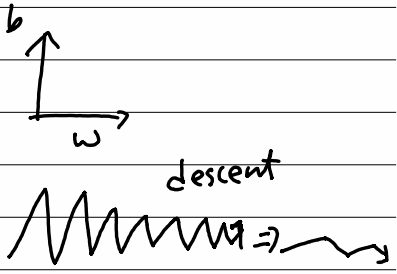
compute dw, db on mini-batch

$S_{dw} = \beta S_{dw} + (1-\beta) dw^2$ - element-wise square

$S_{db} = \beta S_{db} + (1-\beta) db^2$

$w = w - \alpha \frac{dw}{\sqrt{S_{dw}} + \varepsilon}$ (ε: avoid divided by 0)

$b = b - \alpha \frac{db}{\sqrt{S_{db}} + \varepsilon}$



S_{dw} ↓ w ↑

S_{db} ↑ b ↓.

Adam Optimization Algorithm

adaptive moment estimation

momentum + RMSdrop

$V_{dw}=0 \quad S_{dw}=0 \quad V_{db}=0, \; S_{db}=0$

On iter t.

compute dw, db of mini-batch

momentum (

$V_{dw} = \beta_1 V_{dw} + (1-\beta_1) dw \quad V_{db} = \beta_1 V_{db} + (1-\beta_1) db$

RMSprop)

$S_{dw} = \beta_2 S_{dw} + \beta_2 dw^2 \quad S_{db} = \beta_2 S_{db} + (1-\beta_2) db^2$

bias correction

$$V_{dw}^{corrected} = \frac{V_{dw}}{(1-\beta_1^t)} \qquad V_{db}^{corr} = \frac{V_{db}}{(1-\beta_1^t)}$$

$$S_{dw}^{corrected} = \frac{S_{dw}}{(1-\beta_2^t)} \qquad S_{db}^{corr} = \frac{S_{db}}{(1-\beta_2^t)}$$

RMSprop

$$w = w - \alpha \frac{V_{dw}^{corr}}{\sqrt{S_{dw}^{corr}} + \varepsilon} \qquad b = b - \alpha \frac{V_{db}^{corr}}{\sqrt{S_{db}^{corr}} + \varepsilon}$$

hyperparameter α - tune

β_1 : 0.9 → dw - first momentum

β_2 : 0.999 → dw^2 - second momentum

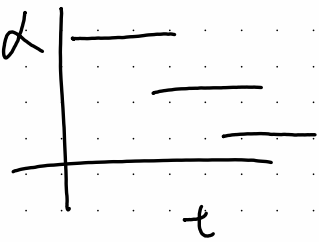
ε : 10^{-8}

Learning Rate Decay

slowly reduce α (avoid overshoot when close to solution)

$$\alpha = \frac{1}{1 + \text{decay rate} \times \text{epoch num}} \alpha_0 \Rightarrow \begin{matrix}\text{para.} \\ \alpha_0 \\ \text{decay rate}\end{matrix}$$

- $\alpha = 0.95^{\text{epoch num}} \cdot \alpha_0$ - exponential decay
- $\alpha = \frac{k}{\sqrt{\text{epoch num}}} \cdot \alpha_0$ or $\frac{k}{\sqrt{t}} \cdot \alpha_0$

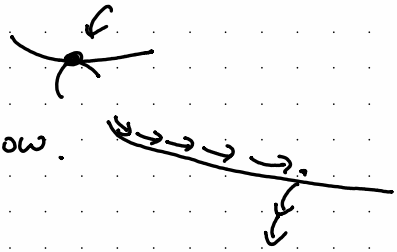


discrete decay

- manual decay (manually update...)

Local optima

most local optima happens at saddle point



- unlikely to stuck in bad local
- plateaus can make learning slow.

Hyperparameter tuning

$\boxed{\alpha}$. β. $\beta_1, \beta_2, \varepsilon$. ~~&~~ α

- # layers
- # hidden units — second important
- learning rate decay — 3rd
- mini-patch

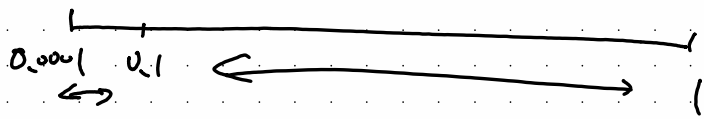
- use "random" search, not grid

- coarse to fine

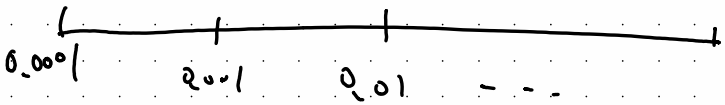
appropriate scale to pick hyper-parameter.

$n^{[l]}$ ⌊ uniform sample works

α.



in log scale



$r = -4 * np.random.rand() \leftarrow r \in [-4, 0]$

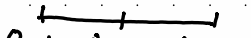
$\alpha = 10^r$

$\in [10^{-4} \dots 10^0]$

Hyperparameter for EWA (exponentially weighted averages)

$\beta = 0.9 \dots 0.999$

$1-\beta = 0.1 \dots 0.001$



$r \in [-3, -1]$

$1-\beta = 10^r$

$\beta = 1 - 10^r$

$\frac{1}{1-\beta}$ ← when $\beta \to 1$.

Small changes have big impact.

?

$\beta = 1 - 10^{(-r-1)}$

Hyperparameter tuning

Babysitting one model ← large model / fewer resource — panda

Training many models in parallel — caviar.

Batch Normalization (single layer)

can we normalize $a^{[l]}$ so that train $W^{[l+1]}$, $b^{[l+1]}$ faster

$z^{(1)} \dots z^{(m)}$

$z^{[l](i)}$

$$\mu = \frac{1}{m}\sum_i z^{(i)}$$

$$\sigma^2 = \frac{1}{m}\sum_i (z_i - \mu)^2$$

$$z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

$$\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta$$

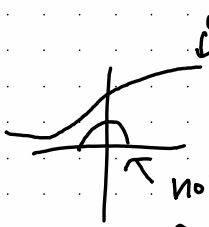
γ, β → learnable hyperparameter

if $\gamma = \sqrt{\sigma^2 + \varepsilon}$

$\beta = \mu$

then $\tilde{z}^{(i)} = z^{(i)}$

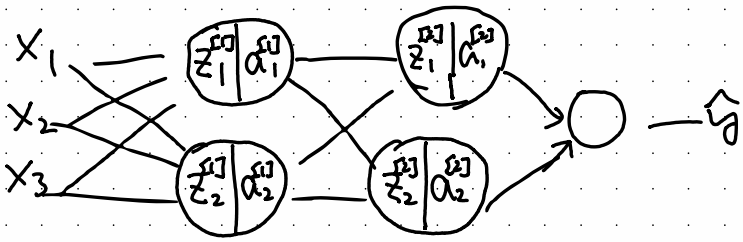
⇒ use $\tilde{z}^{(i)}$ instead of $z^{(i)}$



avoids value squeeze in small area

(by spreading the value)

Batch norm in a NN



$$X \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[\text{BatchNorm}]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \rightarrow a^{[1]} = g^{[1]}(\tilde{z}^{[1]})$$

$$\xrightarrow{w^{[2]}\ b^{[2]}} z^{[2]} \xrightarrow[BN]{\beta^{[2]}\ \gamma^{[2]}} \tilde{z}^{[2]} \rightarrow a^{[2]} \rightarrow \dots$$

Parameters:

$$w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}, \dots w^{[l]}, b^{[l]}$$
$$\beta^{[1]}, \gamma^{[1]} \dots \quad \beta^{[l]}, \gamma^{[l]}$$

this β is different from β in Momentum ADAM etc

(tf.nn.batch-normalization

WORKING with MINI-BATCH

$X^{\{1\}}$ ——

$X^{\{2\}}$ ——

parameter: $w^{[l]}, b^{[l]}, \beta^{[l]}, \gamma^{[l]}$

$(n^{[l]}, 1) \quad (n^{[l]}, 1)$

$$\rightarrow z^{[l]} = w^{[l]} a^{[l-1]} + \cancel{b^{[l]}}$$

Batch-Normalization ↗ eliminate

$$\Rightarrow z^{[l]} = w^{[l]} a^{[l-1]}$$

$$\tilde{z}^{[l]} = \gamma^{[l]} z^{[l]}_{norm} + \beta^{[l]}$$

IMPLEMENTATION

for $t=1 \ldots$ #MINI BATCH
compute forward prop on $X^{\{t\}}$
in each layer, use BN to replace $z^{[l]}$ with $\tilde{z}^{[l]}$
use backprop to compute $dw^{[l]}$, ~~$db^{[l]}$~~, $d\beta^{[l]}$, $d\gamma^{[l]}$
update para $w^{[l]} = w^{[l]} - \alpha dw^{[l]}$

$$\beta^{[l]} = \beta^{[l]} - \alpha d\beta^{[l]}$$

$\vdots$

work with momentum, RMSprop, ADAM etc

Batch norm work bc?

- learning on shifting of distribution
  'covariance shift' e.g. training on black cat ↓ hard to generalize to yellow cat.

△ Batch norm reduce "cov shift"

△ Batch norm as regularization
- mean/var of each mini-batch differ
- add noise to value $z^{[l]}$ ($\tilde{z}^{[l]}$)
- similar to dropout, it add noise to each hidden layer's activation
→ slight regulation

Batch norm at test time

μ, σ^2 for testing: exponentially weighted average across all mini-batch

softmax regression

$P(\text{class}_i | x)$

$n^{[L]} = \#\text{class}$

e.g. 4 class

$z^{[L]} = w^{[L]} a^{[L-1]} + b^{[L]}$ $(4,1)$

Activation Function

$t = e^{(z^{[L]})}$

$a^{[L]} = \frac{e^{z^{[L]}}}{\sum_{j=1}^{4} t_i}$, $a_i^{[L]} = \frac{t_i}{\sum_{j=1}^{4} t_i}$

hard max

$\begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix} \rightarrow \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$

$a^{[L]} = g^{[L]}(z^{[L]})$

$(4,1)$ $(4,1)$

softmax activation

vector → vector.

others output real number.

$C=2$ → soft max reduced to logistic regression

Loss function

$C=4$. $y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ $\hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix}$

$L(y, \hat{y}) = -\sum_{j=1}^{4} y_j \log \hat{y}_j$

e.g. $-y_2 \log \hat{y}_2$

reduce loss → $\hat{y}_2 \uparrow$

m sample

maximum liklihood estimation

$J(w^{[1]}, b^{[1]}, \dots) = \frac{1}{m} \sum L(\hat{y}^{(i)}, y^{(i)})$

Deep Learning Frameworks

TensorFlow

$(w-5)^2$, $w-5=0$

$J(w) = w^2 - 10w + 25$

& only need to calculate forward prop.

Gradient Tape for back prop