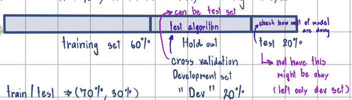
Train / Dev / Test Sets

Applied ML → # layers, # hidden unit, learning rate, ...

Data



train / test → (70%, 30%)

more big data = less "dev set" and "test set"

Mismatched train / test distribution

different picture (quality, camera, type) make sure come from same distribution

1) Train picture from webpage, Dev/test sets from users

Bias / Variance

1) high bias → underfitting

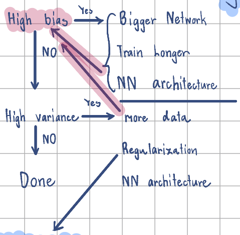
2) high variance → overfitting

e.g.

Train set error →	1%	15%	compare to human error: human ≈ 0% (bad) high bias; human ≈ 15% (good) just right
Dev set error →	11%	16%	

high variance

Basic recipe for machine learning



Regularization

call lamb — regularization parameter — (optional)

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

L_2 regularization → $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$ | $w : (n^{[l-1]}, n^{[l]})$

L_1 regularization → $\frac{\lambda}{m}\sum_{i=1}^{n_x} |w| = \frac{\lambda}{m}\|w\|_1$

$$\sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} (w_{ij}^{[l]})^2$$

Frobenius norm → $\|\cdot\|_2^2$ → $\|\cdot\|_F^2$

Weight decay → $w^{[l]} = w^{[l]} - \alpha[(\text{from backup}) + \frac{\lambda}{m}w^{[l]}]$

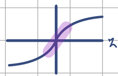
$w^{[l]} = (1 - \alpha\frac{\lambda}{m})\, w^{[l]} - \alpha$ (from backup)

↳ shrink a little each step before the normal gradient descent

Why Regularization Reduces Overfitting

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

$w^{[l]} \approx 0$



example: tanh $g(z) = \tanh(z)$

$\lambda \uparrow$ $w^{[l]} \downarrow$ $z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$

Every layer ≈ linear

Dropout regularization

→ it is hyperparameter. It chosen propotion of neurons and dropout randomly deactivates (randomly set some of them to zero)

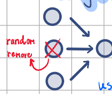
e.g. Illustrate with layer $l = 3$

d_3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob (can be 0.2, ..., 0.8; kept 0.8 chances 0.2)

a_3 = np.multiply(a3, d3) # a3 *= d3

a3 /= keep_prop Inverted dropout → made expected value a3 remain the same

Understanding Dropout



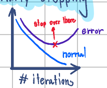
set dropout to lower than another layer because that layer can be overfitting more than another layer

used in computer vision (big data input)

Other Regularization

1) Data augmentation → e.g. flip image in horizon axis (edit image just a little) get more data (is expensive), edit text or transfor (for a little)

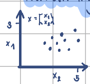
2. Early stopping → 2 times to know where to stop



Optimize cost function J
e.g. Gradient Descent, ..., etc.

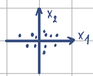
Not overfit
e.g. Regularization, ..., etc.

Normalizing Inputs



1. subtract mean

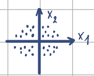
$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$ | $x = x - \mu$



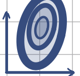
2. Normalize varion

$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} {}^{**2}$

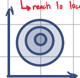
$x /= \sigma^2$



unnormalize



normalize



It made cost function more symmetric

↳ reach to local minima more easier

vanishing / exploding gradients

vanishing / exploding gradients ⇒ big (or small (very very)) slope made training difficult

1) vanishing ⇒ early layers learn extremely slowly

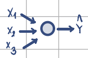
2) exploding ⇒ weights (w, b) become stable ($\hat{y}$ very big) (l ⇒ large layer)

$W^{[l]} >$ Identity matrix e.g. 1.5^{l} (explode)

$W^{[l]} <$ Identity matrix e.g. 0.9^{l} or 0.01^{l} (vanish)

weight initialization for deep networks

example



$z = w_1 x + \ldots + w_n x_n + b$ (→ not blowup or small; b: ignore it for this time)

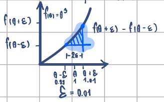
large n → smaller W_i

xavier initialization ≈ $\frac{1}{n}$ (tanh, sigmoid) $\sqrt{\frac{6}{\text{fan\_in} + \text{fan\_out}}}$

he initialization ≈ $\frac{2}{n}$ (ReLU) $\sigma = \sqrt{\frac{2}{\text{fan\_in}}}$

Numerical Approximations of Gradients (use in gradient checking)

check you derivative computation



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

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

$$g(\theta) = 3\theta^3 = 3$$

approx error ⇒ 0.0001

Gradient Checking

$J(\theta) = J(\theta_1, \theta_2, \theta_3, \ldots)$ and $dw^{[l]}, db^{[l]}$

take all of $W^{[l]}, b^{[l]}$ and reshape a big vector θ

for each i: $d\theta_{aprox}[i] = \frac{J(\theta_1, \theta_2, \ldots, \theta_i + \varepsilon, \ldots) - J(\theta_1, \theta_2, \ldots, \theta_i - \varepsilon, \ldots)}{2\varepsilon}$

$\approx d\theta[i] = \frac{\partial J}{\partial \theta_i}$ | $d\theta_{aprox} \approx d\theta$

check $\frac{\| d\theta_{aprox} - d\theta \|_2}{\| d\theta_{aprox} \|_2 + \| d\theta \|_2} \approx 10^{-7}$ (If you declare $\varepsilon = 10^{-7}$ result will be around ε)

$10^{-5}, 10^{-3}$ Bad

Gradient checking implementation notes

- don't use in training (only for debug). (→ made training slow)
- if algorithm fails grad check, look at components to identify bug.

look at $d\theta_{aprox}[i]$, $d\theta[i]$

- Remember regularization
- Doesn't work with dropout Instead turn off dropout use grag check to double check algorithm is correct without it, and then turn on drop out
- Run at random initialization; perhaps again after some training

w, b close to 0 but sometimes bigger maybe implementation back prop is correct only when w, b close to 0, but it gives more accurate when w, b become large

Mini Batch Gradient Descent (Hyperparameters)

Batch vs mini-batch

Vectorization ⇒ allows you to efficiently compute

$X_{(n_x, m)} = [x^{(1)}, x^{(2)}, \ldots x^{(m)}]$ ($X^{\{1\}}_{(n_x, 1000)}$, $X^{\{2\}}_{(n_x, 1000)}$, … $X^{\{5000\}}_{(n_x, 1000)}$) | $Y_{(1, m)} = [y^{(1)}, y^{(2)}, \ldots, y^{(m)}]$ ($Y^{\{1\}}_{(1, 1000)}$, … $Y^{\{5000\}}_{(1, 1000)}$)

$m = 5{,}000{,}000$ | mini-batches of 1,000 each = 5000

mini-batches ⇒ $X^{\{t\}}, Y^{\{t\}}$

symbols

$x^{(i)}$ ⇒ i training examples

$Z^{[l]}$ ⇒ Z value at l layer NN

$X^{\{t\}}, Y^{\{t\}}$ ⇒ mini-batches

mini-batch gradient descent

for $t = 1, \ldots, 5000$ {

1) Forward Prop on $X^{\{t\}}$

$$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$$

$$A^{[l]} = g^{[l]}(Z^{[l]})$$

⋮

2) Compute Cost ⇒ $J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{l} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 \cdot 1000} \sum_{l=1} \| W^{[l]} \|_F^2$

3) Backprop to compute gradients $J^{\{t\}}$ ($X^{\{t\}}, Y^{\{t\}}$)

4) $W^{[l]} = W^{[l]} - \alpha \, dW^{[l]}$

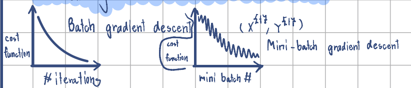
$b^{[l]} = b^{[l]} - \alpha \, db^{[l]}$ }

1 epoch = one full pass through the training set

e.g 60,000 examples, mini_batch size = 1,000

1 epoch = 60 steps

Understanding Mini Batch Gradient Descent



Choosing your mini-batch size

size = m | $(X^{\{t\}}, Y^{\{t\}}) = (X, Y)$ Too long per iteration

size = 1 | every examples is mini-batch lose speed for vectorize

In practice value in between 1 - m

small training set ($m \leq 2000$) ⇒ use batch

Typical mini-batch size ⇒ 64, 128, 256, 512 (2^6, 2^7, 2^8, 2^9; 2 made run fast)

↳ make sure fit in cpu/gpu memory

Exponentially weighted averages

to know it learn this below

optimization algorithm faster than gradient descent

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

e.g $\beta = 0.9 \approx 10$ days temp

$\beta = 0.98 \approx 50$ days temp

$\beta = 0.5 \approx 2$ days temp

V_t as approximately average over

$\approx \frac{1}{1-\beta}$ day temp

Understanding exponentially weighted averages

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

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

$0.9\,V_{98} + 0.1\,\theta_{98}$

$0.9\,V_{97} + 0.1\,\theta_{97}$

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

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

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

$\approx \frac{1}{1-\beta}$ $\varepsilon = 1-\beta$

$\frac{1}{\varepsilon} = 0.02$

$0.9^{10} \approx 0.35 \approx \frac{1}{e}$ $\frac{(1-\varepsilon)^{\frac{1}{\varepsilon}}}{0.9} = \frac{1}{e}$

in contrast $0.98^{50} \approx \frac{1}{e}$

Bias correction of Exponentially Weighted Averages

$\frac{V_t}{1-\beta^t}$

$t = 2$; $1-\beta^t = 1-(0.98)^2 = 0.0396$

$\frac{V_2}{0.0396} = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$

Gradient Descent with momentum

example

take a bit of step

slower learn

faster learn

with momentum

on iteration t:

Compute dw, db on current mini-batch

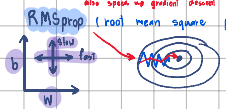
$V_{dw} = \beta V_{dw} + (1-\beta)\,dw$ (friting, velocity, acceleration)

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

imagine as bowl down from hill

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

$b = b - \alpha V_{db}$



On iteration t:

Compute dw, db on gradient mini-batch

$S_{dw} = \beta * S_{dw} + (1-\beta)\,dw^2$ small — element wise squaring operations

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

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

add ε to be sure not value to divide by very small number

Adam Optimization Algorithm (just taking 'momentum' + RMSprop)

$V_{dw} = 0$, $S_{dw} = 0$, $V_{db} = 0$, $V_{db} = 0$

on iteration t:

Compute dw, db on gradient mini-batch

momentum $V_{dw} = \beta_1 V_{dw} + (1-\beta_1)\,dw$; $V_{db} = \beta_1 V_{db} + (1-\beta_1)\,db$

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

$V_{dw}^{\text{corrected}} = V_{dw} / (1-\beta_1^t)$; $V_{db}^{\text{corrected}} = V_{db} / (1-\beta_1^t)$

$S_{dw}^{\text{corrected}} = S_{dw} / (1-\beta_2^t)$; $S_{db}^{\text{corrected}} = S_{db} / (1-\beta_2^t)$

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

Hyperparameter

stuck to find local minima, just reduce α make it easier to reach local minima

learning rate decay (reduces learning rate overtime)

1 epoch = 1 pass through data

$\alpha = \frac{1}{1 + \text{decay\_rate} * \text{epoch\_num}}\,\alpha_0$

Epoch	α
1	0.1
2	0.67
3	0.5
4	0.4

other methods

$\alpha = 0.95^{\text{epoch\_num}} \cdot \alpha_0$

$\alpha = \frac{k}{\sqrt{\text{epoch\_num}}} \cdot \alpha_0$ or $\frac{k}{\sqrt{t}} \cdot \alpha_0$

or just manual decay

Tuning Process

Hyperparameters

α most important

$\beta \approx 0.9$

$0.9 \left\{ \begin{matrix} 0.99 \\ 0.999 \end{matrix} \right\} 10^{-5}$

$\beta_1, \beta_2, \varepsilon$

layers

hidden units

learning rate decay

mini-batch size

Using an approciate scale to pick hyperparameters

1) random e.g $n^{[l]} = 50, \dots, 100$ picking random value

\# layers $L: 2 - 4$

$\alpha = 0.0001, \dots, 1$ ($\log_{10} 0.0001$ … $\log_{10} 1$)

can be $\in [-4, 0]$

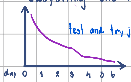
$r = -4 * \text{np.random.rand}()$

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

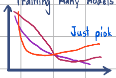
$\alpha = 10^{r}$

Hyperparameter Tuning in Practice

Babysitting one model (Pandora)



Training many models in parallel (Caviar)



Normalizing activations in a network (Batch Normalization)

Normalizing inputs to speed up learning

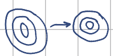
mean: $\mu = \frac{1}{M} \sum_i (X)^i$

$X = X - \mu$

compute varient: $\sigma^2 = \frac{1}{M} \sum_i X^{(i)2}$

$X = X / \sigma^2$

More round (easier to optimize)



normalize $a^{[l]}$ so $W^{[l]}, b^{[l]}$ more fast

Implementing Batch Norm

→ give some intermediate value in NN $z^{(1)}, \dots, z^{(n)}$

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

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

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

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

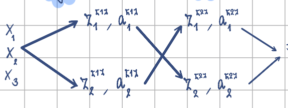
$\beta = \mu$

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

γ, β: learn about parameters of your model

use instead $z^{(i)}$

Fitting Batch Norm into Neural Networks



$X \xrightarrow{W^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[\text{Batch Norm (BN)}]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \rightarrow a^{[1]} = g^{[1]}(\tilde{z}^{[1]})$

e.g. tf.nn.batch-normalization()

Working with mini-batches

→ process in every mini-batch

$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$ → if using batch normalization you can remove it or make it always zero

Shapes: $z^{[l]} = [n(l), m]$

$\beta^{[l]} = [n(l), m]$

$\gamma^{[l]} = [n(l), m]$

Implementing gradient descent

for $t = 1$ (num MiniBatches)

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

In each hidden layer, use Batch Norm to replace $z^{[l]}$ with $\tilde{z}^{[l]}$

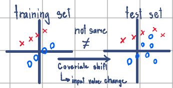
use backprop to compute $dW^{[l]}$, ~~$db^{[l]}$~~, $d\beta^{[l]}$, $d\gamma^{[l]}$

update parameters $W^{[l]} = W^{[l]} - \alpha\, dW^{[l]}$

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

$\gamma^{[l]} = \dots$

Why does batch norm work



- Each mini-batch is scaled by the mean/variance on just that mini-batch
- It adds some noise to the values $z^{[l]}$
- This has a slight regularization effect

Batch Norm At Test Time

μ, σ^2 → activate using exponentially weighted averages (cross-mini-batch)

$X^{\{1\}}, X^{\{2\}}, X^{\{3\}}, \dots$ → $\mu^{\{1\}[l]}, \mu^{\{2\}[l]}, \mu^{\{3\}[l]}$ ⇝ μ

$z_{normalize} = \frac{z - \mu}{\sqrt{\sigma^2 + \varepsilon}}$; $\tilde{z} = \gamma z^{(i)}_{normalize} + \beta$

Softmax Regression (recognize multiclass)

e.g. cats (1), dogs (2), other 0

$C = \#$ classes $= 3$ (0, …, 2) — output layer = 3

$X \rightarrow$ o o $\rightarrow$ o o o o $\rightarrow \hat{y}$ (3,1) — layer L (last layer)

Softmax layer

$z^{[L]} = W^{[L]} a^{[L-1]} + b^{[L]}$

Activation function

$t = e^{(z^{[L]})}$ (3,1)

$a^{[L]}_{(3,1)} = \frac{e^{(z^{[L]})}}{\sum_{j=1}^{3} t_i}$, $a_i^{[L]} = \frac{t_i}{\sum_{j=1}^{3} t_i}$

$z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \end{bmatrix}$, $t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \end{bmatrix}$, $\sum_{j=1}^{3} t_j = 156.2$

$a^{[L]} = \frac{t}{156.2} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \end{bmatrix}$

Training Softmax Classifier

Loss function e.g. $y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ → cat, $\hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix}$ → not doing well

$\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{4} y_j \log \hat{y}_j = -y_2 \log \hat{y}_2 = -\log \hat{y}_2$ ($y_2 = 1$)

$J(W^{[1]}, b^{[1]}, \dots) = \frac{1}{M} \sum_{i=1}^{M} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$

$Y = [y^{(1)}, y^{(2)}, \dots, y^{(m)}] = \begin{bmatrix} 0 & 0 & 1 & \dots \\ 1 & 0 & 0 & \\ 0 & 1 & 0 & \\ 0 & 0 & 0 & \end{bmatrix}$

$\hat{Y} = [\hat{y}^{(1)}, \dots, \hat{y}^{(m)}] = \begin{bmatrix} 0.3 & \dots \\ 0.2 & \\ 0.1 & \\ 0.4 & \end{bmatrix}$

Gradient Descent

Backprop: $dz^{[L]}_{(4,1)} = \hat{y} - y$ $\left(\frac{\partial J}{\partial z^{[L]}}\right)$

The problem of local optima

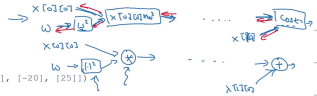
Tensorflow

Motivate problem

cost function

$J(w) = w^2 - 10w + 25 = (w-5)^2$ $\quad w=5$

Code example




```
import numpy as np
import tensorflow as tf

coefficients = np.array([[1], [-20], [25]])

w = tf.Variable([0],dtype=tf.float32)
x = tf.placeholder(tf.float32, [3,1])
cost = x[0][0]*w**2 + x[1][0]*w + x[2][0]     # (w-5)**2
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
init = tf.global_variables_initializer()
session = tf.Session()
session.run(init)
print(session.run(w))

for i in range(1000):
    session.run(train, feed_dict={x:coefficients})
print(session.run(w))
```

```
with tf.Session() as session:
    session.run(init)
    print(session.run(w))
```

Andrew Ng