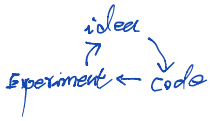
- Setting up your ML application
  - Train / dev / test sets

    Applied ML is a highly iterative process

    # layers, # hidden units, learning rates, activation functions ...

    Not having a test set might be Okay (only dev set (CV))

  - Bias / Variance

  - Basic "recipe" for machine learning

    High bias? → • Bigger Network
    • Train longer

    High Variance? → • More data
    • Regularization
    • NN architecture search

- Regularizing your Neural Network

  - Regularization

- Logistic Regression:

  L2 Regularization

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

  L1 Regularization

  $$J(w,b) = \cdots\cdots + \frac{\lambda}{2m}\|w\|_1$$

  w will be sparse (a few 0s in w)

  λ: Regularization Parameter

- Neural Network

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

$$\|w^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l-1]}}\sum_{j=1}^{n^{[l]}} (w_{ij}^{[l]})^2 \quad \text{—— Fobenius Norm of Matrix}$$

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

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

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

"weight decay"

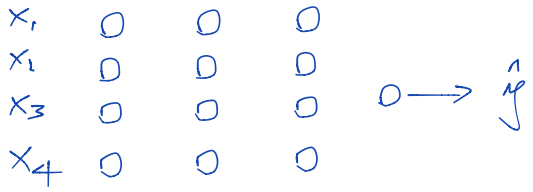
- Why regularization Reduces Overfitting

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

Reducing the impact of some of the hidden unit,

by adding penalize term. $\frac{\lambda}{2m}\sum_{l=1}^{L}\|w^{[l]}\|_F^2$

- Dropout Regularization



- Inverted Dropout

  Illustrate with $L=3$. keep-prob = 0.8

  $d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep\text{-}prob$

  $a3 = a3 * d3$

  $a3 = a3 / keep\text{-}prob$ (ensure expected value be ~~the~~ same)

- Making predictions at test time

  No dropout for testing. otherwise output will be noisy (random)

- Understanding Dropout

  Intuition: Can't rely on any one feature, so have to spread out weights
  → shrinking weights

  keep-prob: different layers can have diffent keep-prob values, usually
  the large neuron layer may have Smaller keep-prob.

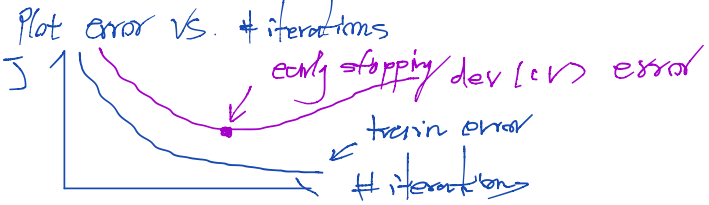
  Dropout is frequently used in Computer Vision

Downside: J is less well defined

- Other Regularization Method
  - Data augmentation
    - flip the image to get more training examples
    - Random Crops, distortions, rotations of images

    ⇒ cheap way to get more data

  - Early stopping



Downside: Early stopping Couples [optimize cost func J, Not overfit] ⇒ orthogonalization

Cannot work on these two aspect individually

Make things more completed

- Setting up your optimization Problem
  - Normalizing Inputs
    - Normalizing training sets

- Substract mean

$\mu = \frac{1}{m}\sum_i^m x^{(i)}$ $\quad x := x - \mu$

- Normalize variance

$\sigma^2 = \frac{1}{m}\sum_i^m (x^{(i)})^2$ [element-wise, already substracted mean]

$x = \frac{x}{\sigma}$

- Use same μ, σ to normalize test set

- Why normalize Inputs

: Unnormalized might have stracked/elongated Cost function
normalized: guarantees all input features are in about same scale → might have small learn rate

- Vanishing / exploding gradients

An intuitive example.

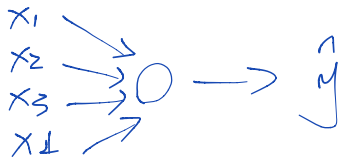
for a large deep NN, is each $w^{[L]}$ is very large than diag(1),
then output approx. $\hat{y} = w^{[L]}\begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{L-1} x \longrightarrow \hat{y}$ explod

if $w^{[L]}$ are all very small than diag(1), $\longrightarrow \hat{y}$ Vanish

Gradient descent may be very slow

- Weight Initialization for Deep Networks

- Single neuron example



$z = w_1 x_1 + w_2 x_2 + \cdots w_n x_n + b$

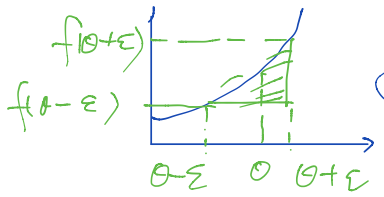
large n → smaller w_i

Initialize w_i so that $Var(w_i) = \begin{cases} \frac{1}{n} & \text{if tanh} \\ \frac{2}{n} & \text{if ReLU} \end{cases}$

$w^{[l]} = np.random.randn(shape) * np.sqrt\left(\begin{matrix}\frac{1}{n^{[l-1]}} \leftarrow \text{tanh} \\ \frac{2}{n^{[l-1]}} \leftarrow \text{ReLU}\end{matrix}\right)$

(according to published paper)

- Numerical Approximation of gradients
  - check your derivative Computation



$g(\theta) = \frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon} + O(\varepsilon^2)$

Sidenote: $f'(\theta) = \lim_{\varepsilon \to 0} \frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon} = \lim_{\varepsilon \to 0} \frac{f(\theta+2\varepsilon) - f(\theta)}{\varepsilon}$

$\varepsilon < 1 \quad f'(\theta) = \frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon} + O(\varepsilon^2)$

$f'(\theta) = \frac{f(\theta+\varepsilon) - f(\theta)}{\varepsilon} + O(\varepsilon)$

1st implementation yields smaller estimation error

Gradient checking

- Gradient checking for a neural network
  - Take $W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L]}$ and reshape into a big vector θ

    $J(W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L]}) = J(\theta)$
  - Take $dW^{[1]}, db^{[1]}, \dots, dW^{[L]}, db^{[L]}$ and reshape into a big vector $d\theta$

- Gradient checking

  for each i:

  $$d\theta_{approx}[i] = \frac{J(\theta_1, \theta_2, \dots, \theta_i + \varepsilon, \dots) - J(\theta_1, \theta_2, \dots \theta_i - \varepsilon, \dots)}{2\varepsilon}$$

  check $d\theta_{approx} \overset{?}{\approx} d\theta$ (from backprop)

  $$\frac{\| d\theta_{approx} - d\theta \|_2}{\| d\theta_{approx} \|_2 + \| d\theta \|_2} < 10^{-7} \text{ (e.g.)} \Rightarrow \text{ok!}$$

- Grad checking Implementation Notes

  - Don't use in training — only to debug

    make sure backprop works as expected

  - If algorithm fails grad check, look at component to identify bug

  - Remember regularization term in $d\theta$ from J

- Grand check doesn't work with Dropout
  difficult to calculate J with Dropout.
  Recommendation: turn off dropout, do Grad check,
  make sure algorithm works, then turn on
  Dropout, hope for the best!

- Run at random initialization; perhaps again after some training