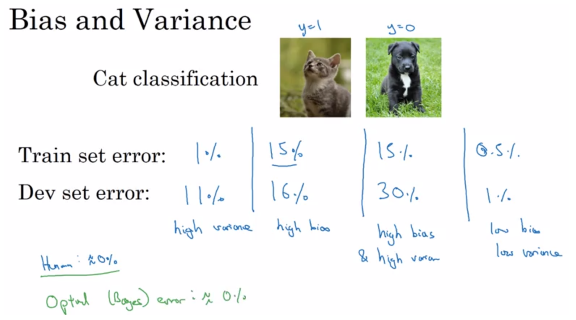
# deep Learning_c2_week1

**Created:** 19/02/01 11:01 **Updated:** 19/02/03 18:22

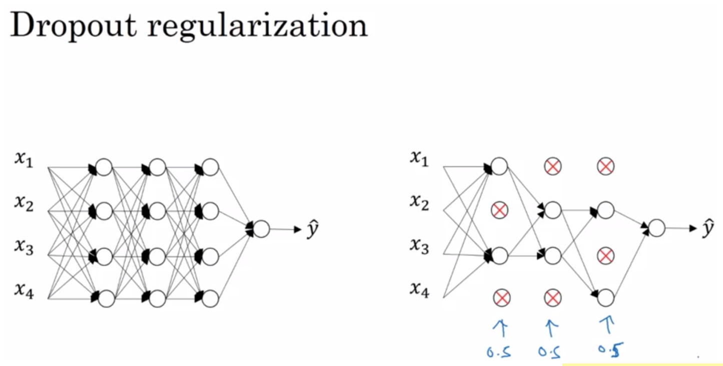
- ## Train/Dev/Test set
  - data set usually be split into 3 set
    - Training set
    - Hold-out cross validation set / Development or "dev" set
    - Test set
  - in traditional ML, the portion of them would be 60%/20%/20%
  - but when data set grow larger, we now could : ==> 98/1/1 or 99.5/0.25/0.25 (percentage)
  - Make sure the dev and test set are coming from the same distribution.
- ## bias & variance
  - 

    Bias and Variance

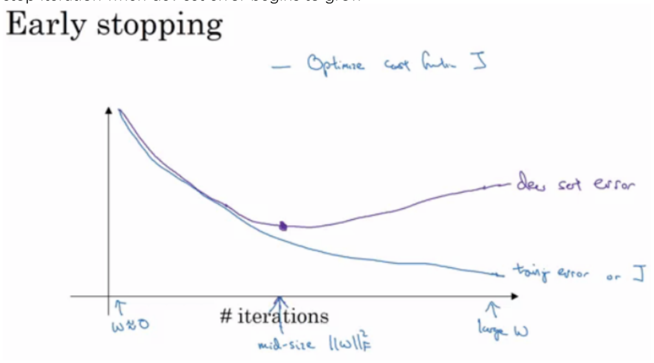
    Cat classification

    | | y=1 | y=0 | | |
    |---|---|---|---|---|
    | Train set error: | 1% | 15% | 15% | 0.5% |
    | Dev set error: | 11% | 16% | 30% | 1% |
    | | high variance | high bias | high bias & high variance | low bias low variance |

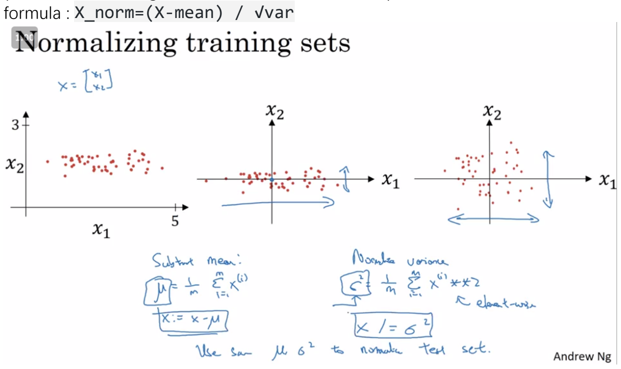
    Human : ≈0%

    Optimal (Bayes) error : ≈ 0%
  - These Assumptions came from that human has 0% error. If the problem isn't like that you'll need to use human error as **baseline**.
  - recipe for ML
    - If your algorithm has a high **bias**:
      - Try to make your NN bigger (size of hidden units, number of layers)
      - Try a different model that is suitable for your data.
      - Try to run it longer.
      - Different (advanced) optimization algorithms.
    - If your algorithm has a high **variance**:
      - More data.
      - Try regularization.
      - Try a different model that is suitable for your data.
- ## Regularization
  - L1 regularization
    - `||W|| = Sum(|w[i,j]|) # sum of absolute values of all w`
    - get **sparse** parameters (most of them would equal to 0)
    - can be used to compress parameters
  - **L2 regularization**
    - `||W||^2 = Sum(|w[i,j]|^2) # sum of all w squared`
    - regularized NN cost function: `J(w,b) = (1/m) * Sum(L(y(i),y'(i))) + (lambda/2m) * Sum((||W[l]||^2)`
      - caution: in python, *lambda* is reserved key word,
    - implementation tip: plot the cost function J as a function of the number of iterations of gradient descent
  - **Dropout regularization**
    - randomly eliminate some neurons. So, each time just train a small subset of whole neural network.
    -
    - 

    - A most common technique to implement dropout is called "Inverted dropout".
      - python code:
      - `keep_prob = 0.8 # 0 <= keep_prob <= 1`
      - `l = 3 # this code is only for layer 3# the generated number that are less than 0.8 will be dropped. 80% stay, 20% dropped`
      - `d3 = np.random.rand(a[l].shape[0], a[l].shape[1]) < keep_prob`
      -
      - `a3 = np.multiply(a3,d3) # keep only the values in d3`
      -
      - `# increase a3 to not reduce the expected value of output# (ensures that the expected value of a3 remains the same) - to sol`
      - `a3 = a3 / keep_prob`
      - use *keep-prob* to control the probability of inversion.
    - intuition about how it work:
      - can't rely on any one feature, so have to spread out weights. ("don't put all the eggs in one basket")
    - downside:
      - can't plot J-iterations to debug

        - tips:
            - large layer size, smaller *keep-prob*
            - output layer,*keep-prob* equal to 1 (prediction should be stable)
            - Apply dropout both during forward and backward propagation.
    - data augementation
        - enlarge data set without getting brand new data
        - E. X.
            - in a computer vision data:
                - flip all your pictures horizontally
                - apply a random position and rotation to an image
    - early stopping
        - stop iteration when dev set error begins to grow
        - 

        - advantage
            - don't need to search a hyper parameter like in other regularization approaches (like `lambda` in L2 regularization).
        - disadvantage
            - simultaneously tries to minimize the cost function and not to overfit which contradicts the *orthogonalization* approach (will be discussed further).

- ## Normalization
    - (mathematical background: assume that all the sample in data set are **Independent and identically distributed,**
    - formula : `X_norm=(X-mean) / √var`
    - 
Normalizing training sets
$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$
3
$x_2$
$x_1$
5
$x_2$
$x_1$
$x_2$
$x_1$
Subtract mean:
$\boxed{\mu} = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$
$\boxed{x := x - \mu}$
Normalize variance
$\boxed{\sigma^2} = \frac{1}{m} \sum_{i=1}^{m} x^{(i)} ** 2$
element-wise
$\boxed{x \mathrel{/}= \sigma^2}$
Use same $\mu$ $\sigma^2$ to normalize test set.
Andrew Ng
    - why?
        - makes your inputs centered around 0
        - fast GD as  the shape of the cost function will be consistent (look more symmetric like circle in 2D example)

- ## Weight initialization for NN
    - why?
        - **exponential vanishing / exploding gradients** !!! (in deep NN with many layers)
            - when layers grow larger, each layer may output smaller or bigger Z[l].
            - if Z is too large or too small, the gradient would also be too large or too small (depend on which activation function you choose)
    - solution
        - intuition
            - In a single neuron (Perceptron model): `Z = w1x1 + w2x2 + ... + wnxn`
            - So if `n_x` is large we want `W`'s to be smaller to not explode the cost.
            - So lets say when we initialize `W`'s like this (better to use with `tanh` activation):

```
np.random.rand(shape) * np.sqrt(1/n[l-1])
```

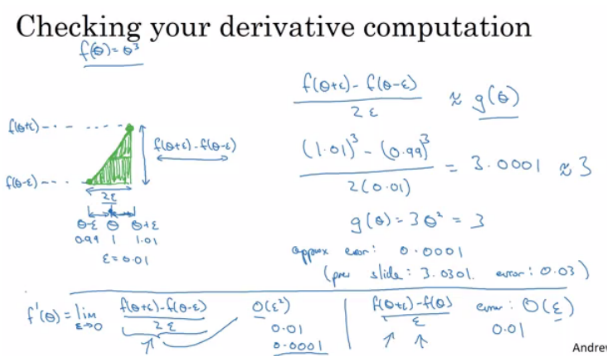
or variation of this (Bengio et al.):

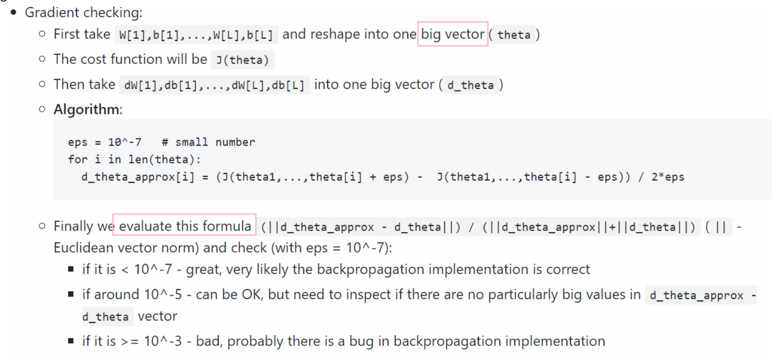
```
np.random.rand(shape) * np.sqrt(2/(n[l-1] + n[l]))
```

            - Setting initialization part inside sqrt to `2/n[l-1]` for `ReLU` is better:

```
np.random.rand(shape) * np.sqrt(2/n[l-1])
```

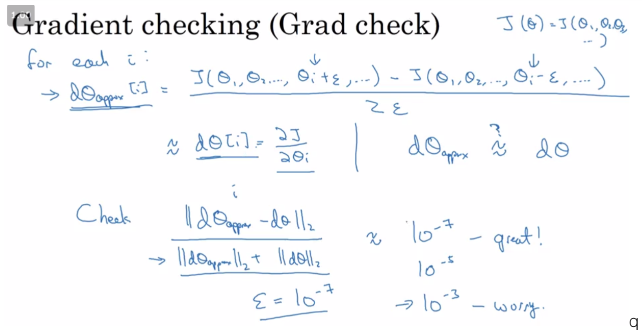
- Numerical approximation of derivative (gradient checking)

- 

- process
  - 

    - Gradient checking:
      - First take `W[1],b[1],...,W[L],b[L]` and reshape into one big vector ( `theta` )
      - The cost function will be `J(theta)`
      - Then take `dW[1],db[1],...,dW[L],db[L]` into one big vector ( `d_theta` )
      - **Algorithm**:

        ```
        eps = 10^-7    # small number
        for i in len(theta):
          d_theta_approx[i] = (J(theta1,...,theta[i] + eps) -  J(theta1,...,theta[i] - eps)) / 2*eps
        ```

      - Finally we evaluate this formula `(||d_theta_approx - d_theta||) / (||d_theta_approx||+||d_theta||)` ( `||` - Euclidean vector norm) and check (with eps = 10^-7):
        - if it is < 10^-7 - great, very likely the backpropagation implementation is correct
        - if around 10^-5 - can be OK, but need to inspect if there are no particularly big values in `d_theta_approx - d_theta` vector
        - if it is >= 10^-3 - bad, probably there is a bug in backpropagation implementation
  - 
Gradient checking (Grad check)
J(θ) = J(θ₁, θ₂, ...)
for each i:
→ dθ_approx[i] = J(θ₁, θ₂, ..., θᵢ + ε, ...) − J(θ₁, θ₂, ..., θᵢ − ε, ...) / 2ε
≈ dθ[i] = ∂J/∂θᵢ
dθ_approx ≈? dθ
Check ‖dθ_approx − dθ‖₂ / ‖dθ_approx‖₂ + ‖dθ‖₂
≈ 10⁻⁷ − great!
10⁻⁵
→ 10⁻³ − worry.
ε = 10⁻⁷
q
- notes
  - only for debugging
  - Gradient checking doesn't work with dropout because J is not consistent.
    - You can first turn off dropout (set `keep_prob = 1.0`), run gradient checking and then turn on dropout again.

---

coding!!!

- Initialization
  - randomly initialize $W$ to break symmetry
  - initializing with overly large random numbers slows down the optimization.
  - use "He-Initialization" for relu

related code:

```
np.zeros(shape)
np.sqrt()
np.random.randn(shape)
```

- L2 regularization
  - Weights end up smaller ("weight decay")
  - formula
    - $$J_{regularized} = \underbrace{-\frac{1}{m} \sum\limits_{i = 1}^{m} \large{(}\small y^{(i)}\log\left(a^{[L](i)}\right) + (1-y^{(i)})\log\left(1- a^{[L](i)}\right) \large{)} }_\text{cross-entropy cost} + \underbrace{\frac{1}{m} \frac{\lambda}{2} \sum\limits_l\sum\limits_k\sum\limits_j W_{k,j}^{[l]2} }_\text{L2 regularization cost}$$
    - cost function
    - BP $\left(\frac{d}{dW} \left( \frac{1}{2}\frac{\lambda}{m} W^2\right) = \frac{\lambda}{m} W\right)$.

related code:

```
np.sum(np.square(W))
```

- dropout
  - divide each dropout layer by keep_prob to keep the same expected value for the activations.

related code:

```
    D1 = np.random.rand(A1.shape[0],A1.shape[1])                # Step 1: initialize matrix D1 = np.random.rand(..., ...)
    D1 = D1<keep_prob                                           # Step 2: convert entries of D1 to 0 or 1 (using keep_prob as the threshold)
    A1 = A1*D1                                                  # Step 3: shut down some neurons of A1
    A1 = A1/keep_prob                                           # Step 4: scale the value of neurons that haven't been shut down
```