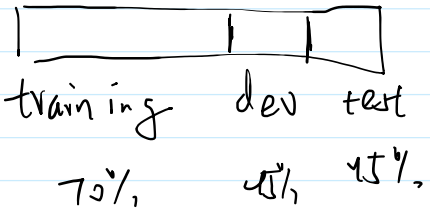
# Weak 1 - Setting of ML Application

Samstag, 18. Juli 2020 14:10

## Training/dev/test sets




Dev: to design/tune hyperparameters

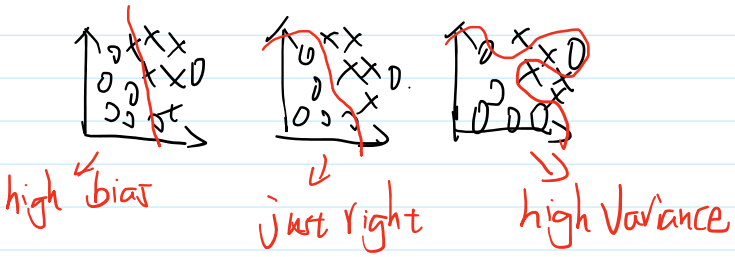
When dataset large enough -> 10,000: 98%, 1%, 1%

## Evaluation of Algorithms

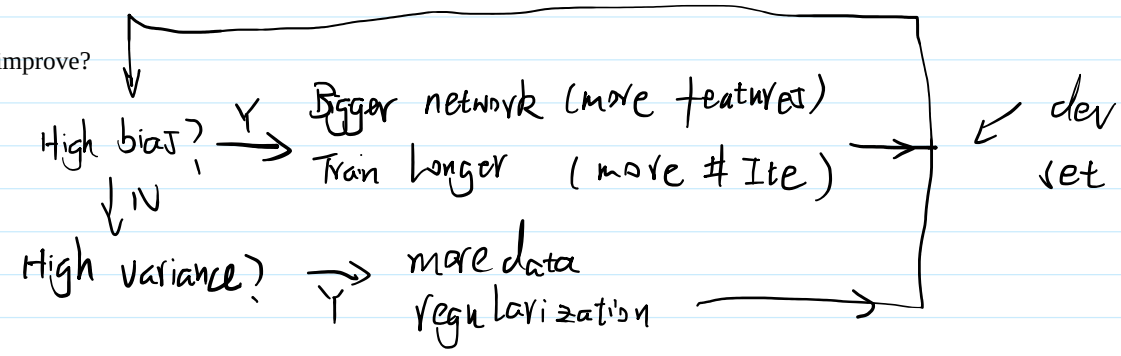
Bias and variance

High bias: under fitting
High variance: overfitting

| | High variance | High bias | both |
|---|---|---|---|
| Training set error: | 1% | 15% | 15% |
| Dev set error | 11% | 16% | 30% |




How to improve?




Among them, bigger network almost don't hurt variance and more data almost don't hurt bias.

## Regularization

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

Can be omited

$$\|w\|^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$$

$L_2$ Regularization

$L_1$ Regularization

$$\frac{\lambda}{2m}|w|$$

np.sum(np.square($w^{[l]}$)) for every layer $l$

$\lambda$ regularization parameter

For NN:

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

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

$$w^{[l]} = w^{[l]} - \alpha dw^{[l]} = w^{[l]} - \alpha\left[(\text{from Back prop}) + \frac{\lambda}{2m} w^{[l]}\right]$$

decrease w ← weight decay

Why regu reduces overfitting?

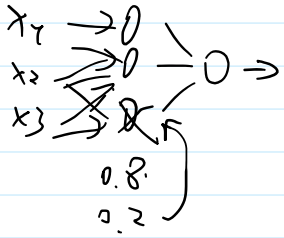
$$J(\;) = \frac{1}{m}\sum \dots + \frac{\lambda}{2m}(\;)$$

When lambda large, the goal of minimizing J() is becoming to minizine $\frac{\lambda}{2m}(\;) \Rightarrow w^{[l]} \approx 0$

Variance ↓ ← Less features ← A lot of hidden layer = 0

Dropout

Keep-prob = 0.8 -> 0.2 dropout




For example layer 3:

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

a3 = a3 * d3 Some units will be 0

A3 /= keep-prob # to keep the a3 and then z4 in the same scale dont decrease.

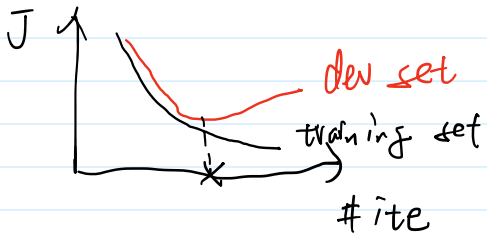
Dropout only used for training set, not test set because it will bring in random result.

Different dropout (keep-prob) can be used on different layers. If one layer has many units, keep-pro can be smaller.

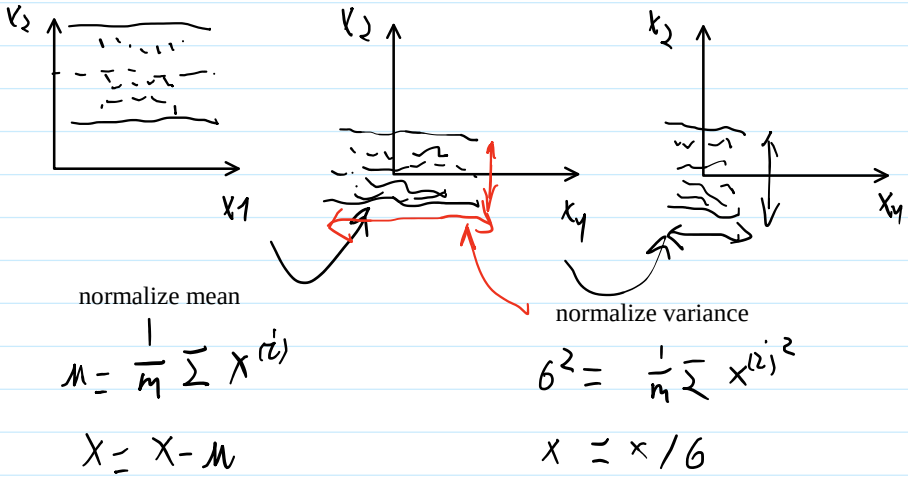
For computer vision application, dropout is usually default because the input size is so big (so many features, Xn large), so the training example m are always relative too small -> high variance.

## Other Regularization techniques

1. More input data -> data augmentation
   a. Horizontal mirror images
   b. Random rotations
   c. Strong distortion
2. Early stop



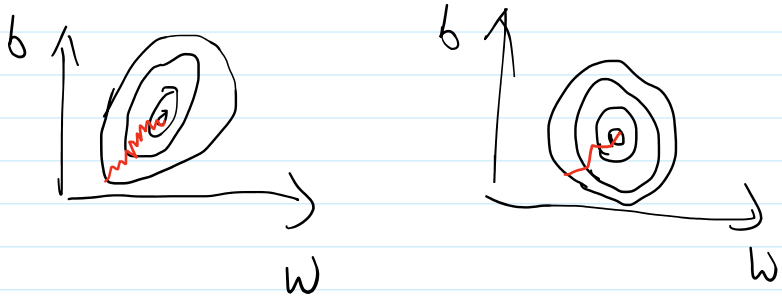

## Data Normalization




use same parameters to normalize train and test set

## When Normalization required?

when data on similar scale, learning is faster, which allows a bigger learning rate.




## Vanishing / Exploding gradients

when $W^{[l]} > I$ / $< I$

I(identical metrix), activation and gradients will increased or decrease expotentially, which makes learning hard.

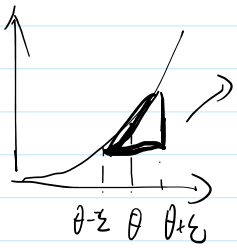
To solve Vanishing / Exploding gradients problem:

Random initialize weight w for layer l by:

$W^{[l]} = np.random.randn(shape) * np.sqrt\left(\frac{2}{n^{[l-1]}}\right)$ if activation = ReLU

$\sqrt{\frac{1}{n^{[l-1]}}}$ if activation = tanh

## Gradient Checking -- using mathematical relationship tp calculate gradient




gradient at $\theta = \frac{J(\theta+\varepsilon) - J(\theta-\varepsilon)}{2\varepsilon} \approx g(\theta)$

Implementation:

1. Reshape all parameters (w1, b1, w2, b2, …..) into vectors and concatenating them into one vector. → $J(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}) = J(\theta)$
2. Do the previous to all dw and db to get d_theta
3. For each item from $\theta$

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

should $\approx d\theta^{[i]} = \frac{\partial J}{\partial \theta_i}$

check $\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2} \approx 10^{-7}$ by $\varepsilon = 10^{-7}$ ⇒ great!

$10^{-3}$ ⇒ worry!

- Dont use gradient check in training, only to debug model.
- Gradient check can't be used with drop off

## Take away from the programming assignment:

- Initialization correctly helps the NN model to converge faster and get a lower training error.
  - Initialize w and b as zeros for each layer leads to symmetry fail, meaning the cost will not decrease and model will predict 0 for each examples, because each layer will learn the same thing. So at last it is just a linear classifier such as logistic regression.
  - Set the weights randomly can break the symmetry. small random values performs better than larger.
  - He initialization / Xavier initialization: uses a scaling factor for the weights sqrt(1./layers_dims[l-1]) (Xavier) ; He initialization would use `sqrt(2./layers_dims[l-1])` and He initialization works well for networks with ReLU activations. -> help to solve vanishing / exploding gradients problem.

```
Def initialize(layers_dims):
    parameters = {}
        L = len(layers_dims) - 1 # integer representing the number of layers

        for l in range(1, L + 1):
            ### START CODE HERE ### (≈ 2 lines of code)
            parameters['W' + str(l)] = np.random.randn(layers_dims[l], layers_dims[l-1]) * np.sqrt(2./layers_dims[l-1])
            parameters['b' + str(l)] = np.zeros(shape=(layers_dims[l], 1))
            ### END CODE HERE ###'

        return parameters
```

- Regularization: L2 rege or dropout
  - L2 regularization: apply the lambda term on cost function and gradients (dW1, dW2, dW3…..)

```
def compute_cost_with_regularization(A3, Y, parameters, lambd):
    """
    Arguments:
    A3 -- post-activation, output of forward propagation, of shape (output size, number of examples)
    Y -- "true" labels vector, of shape (output size, number of examples)
    parameters -- dictionary containing parameters of the model

    Returns:
    cost
    """
    m = Y.shape[1]
    W1 = parameters["W1"]
    W2 = parameters["W2"]
    W3 = parameters["W3"]

    cross_entropy_cost = compute_cost(A3, Y) # This gives you the cross-entropy part of the cost
    L2_regularization_cost = 1/m * lambd/2 * (np.sum(np.square(W1)) + np.sum(np.square(W2)) + np.sum(np.square(W3)))
    cost = cross_entropy_cost + L2_regularization_cost

    return cost
```

  - dropout
    - in forward propagation

```
D1 = np.random.rand(A1.shape[0], A1.shape[1])  # Step 1: initialize matrix D1 = np.random.rand(..., ...)
D1 = (D1 < keep_prob).astype(int)              # Step 2: convert entries of D1 to 0 or 1 (using keep_prob as 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
```

    - in backward:

```
shut down the same neurons, by applying the same mask  D[1]]  to dA1.
divide dA1 by keep_prob again
```

    - only apply drop out on training set because we don't want randomness in the prediction

- Gradient Check
  - mathematically calculate gradient and then compare it with the result from the backpropagation.
  - Before applying gradient check you have to be sure that the J cost function is computed in a right way, because the gradient check based on that.
  - derivative:

$$\frac{\partial J}{\partial \theta} = \lim_{\varepsilon \to 0} \frac{J(\theta + \varepsilon) - J(\theta - \varepsilon)}{2\varepsilon}$$

For each i in num_parameters:

- To compute `J_plus[i]`:
  1. Set $\theta^+$ to `np.copy(parameters_values)`
  2. Set $\theta_i^+$ to $\theta_i^+ + \varepsilon$
  3. Calculate $J_i^+$ using to `forward_propagation_n(x, y, vector_to_dictionary(`$\theta^+$`))`.
- To compute `J_minus[i]`: do the same thing with $\theta^-$
- Compute $gradapprox[i] = \frac{J_i^+ - J_i^-}{2\varepsilon}$

Thus, you get a vector gradapprox, where gradapprox[i] is an approximation of the gradient with respect to `parameter_values[i]`. You can now compare this gradapprox vector to the gradients vector from backpropagation. Just like for the 1D case (Steps 1', 2', 3'), compute:

$$difference = \frac{\| grad - gradapprox \|_2}{\| grad \|_2 + \| gradapprox \|_2}$$