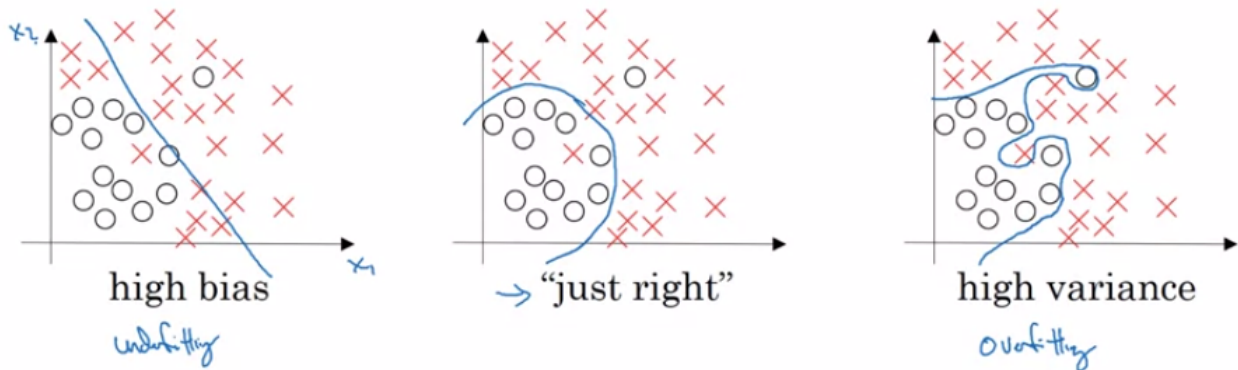


Notations

- $m = m_{train}$ - #train examples
- m_{test} - #test examples
- $n^{[l]}$ - number of hidden units on layer l
- $g^{[l]}$ - activation function on layer l
- $w^{[l]}, b^{[l]}$ - parameters on layer l
- $dW^{[l]}, dB^{[l]} \dots dA^{[l]}$ - derivatives on layer l
- $par^{[l](i)}$ - the value of example i parameter on layer l
- α - learning rate
- C - #classes

Bias and variance



High bias and variance

High bias \rightarrow

- Bigger network
- Train longer
- Better suited NN architectures

High variance \rightarrow

- More data
- Regularization
- More appropriate NN architecture

Regularization

$$\min_{w,b} J(w, b)$$

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

λ – regularization parameter

L1 regularization : $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$

L2 regularization : $\|w\|_1 = \sum_{j=1}^{n_x} |w_j|$

Regularization in neural network :

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

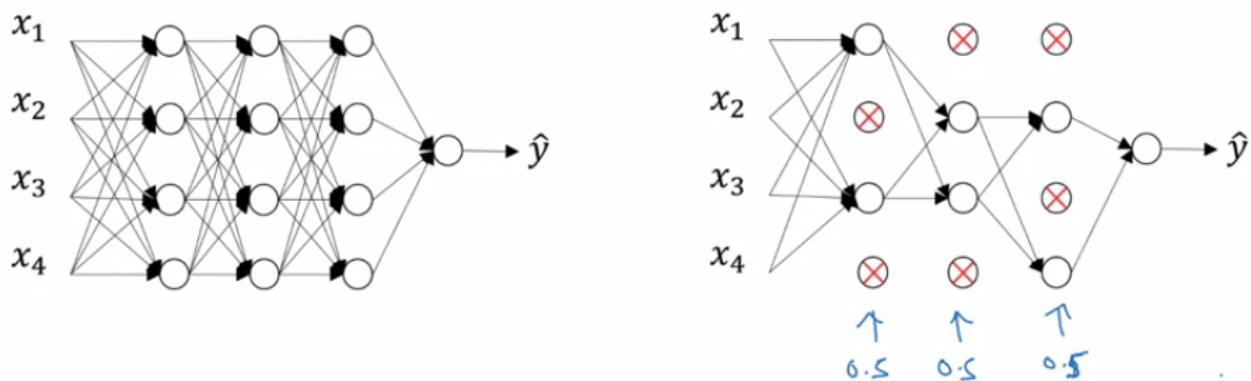
Frobenius norm : $\|w\|_F^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} (w_{i,j}^{[l]})^2$

Backprop with Regularization :

$$dW^{[l]} = (\text{from backprop}) + \frac{\lambda}{m} w^{[l]}$$

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

Dropout regularization



Dropout regularization

Train time : We will randomly eliminate each hidden unit on the hidden layer with a chance $1 - \text{keepprob}$.

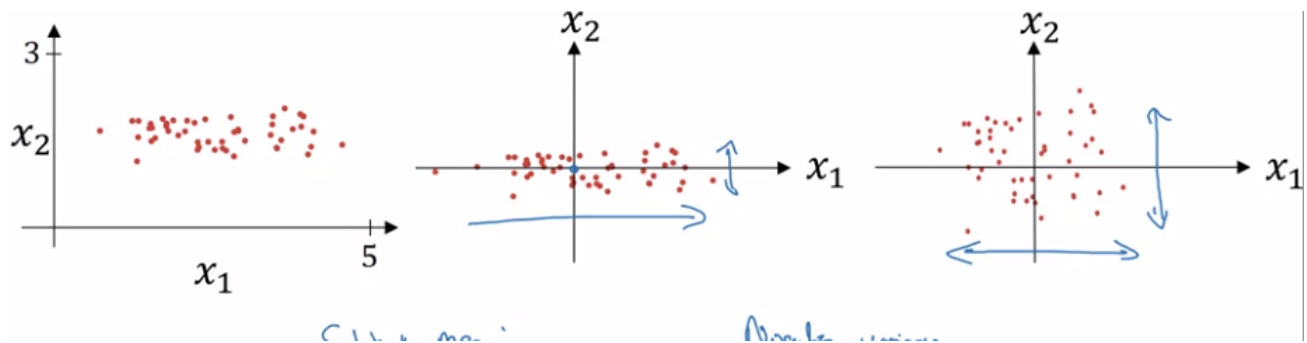
```

#forward_prop
d3 = np.random.rand(A3.shape[0], A3.shape[1]) < keep_prob
A3 = np.multiply(A3, d3) #A3 *= d3
A3 /= keep_prob
....
#back_prop
dA3 = np.multiply(dA3, d3) # dA3 *= d3
dA3 /= keep_prob

```

Test time : In the test time we do not apply dropout(do not randomly eliminate units) and do not keep the 1/keepprob in the calculations used in training.

Normalizing inputs



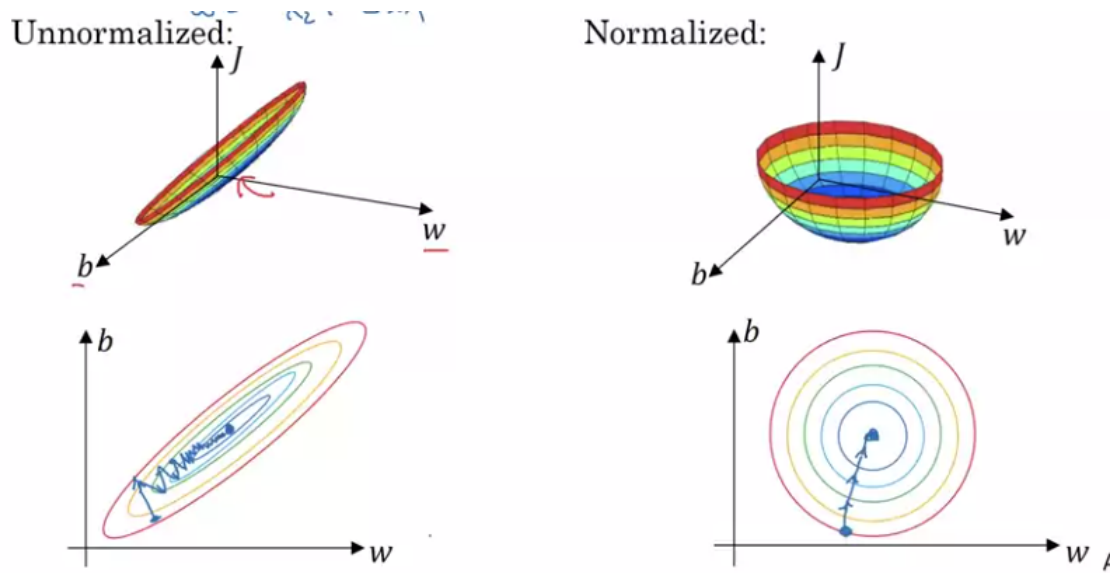
Normalizing inputs

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

$$x = x - \mu$$

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

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



Unnormalized and normalized

Weight Initialization

Relu : $w^{[l]} = \text{np.random.randn}(\text{shape}) * \left(\sqrt{\frac{2}{n^{[l-1]}}} \right)$

Tanh : $w^{[l]} = \text{np.random.randn}(\text{shape}) * \left(\sqrt{\frac{1}{n^{[l]}}} \right)$

Tanh xavier intuition : $w^{[l]} = \text{np.random.randn}(\text{shape}) * \left(\sqrt{\frac{2}{n^{[l]} + n^{[l-1]}}} \right)$

Gradient check for a neural network

Take $\underbrace{W^{[1]}, b^{[1]} \dots W^{[L]}, b^{[L]}}_{\text{concatenate}}$ and reshape into a big vector θ .

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

Take $\underbrace{dW^{[1]}, db^{[1]} \dots dW^{[L]}, db^{[L]}}_{\text{concatenate}}$ and reshape into a big vector $d\theta$.

$$J(\theta) = J(\theta_1 \dots \theta_L)$$

Algorithm :

$\varepsilon = 10^{-7}$

for each i:

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

$$d\theta_{\text{approx}} \approx d\theta?$$

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

$$\text{If } \frac{\|d\theta_{\text{approx}} - d\theta\|_2}{\|d\theta_{\text{approx}}\|_2 + \|d\theta\|_2} \approx 10^{-7} \rightarrow \text{great}$$

If $\frac{\|d\theta_{\text{approx}} - d\theta\|_2}{\|d\theta_{\text{approx}}\|_2 + \|d\theta\|_2} \approx 10^{-5} \rightarrow \text{maybe ok}$

If $\frac{\|d\theta_{\text{approx}} - d\theta\|_2}{\|d\theta_{\text{approx}}\|_2 + \|d\theta\|_2} \approx 10^{-3} \rightarrow \text{worry}$

Gradient checking notes :

- Don't use in training - only to debug
- If algorithm fails grad check, look at components to try to identify bug
- Remember regularization
- Does not work with dropout
- Run at random initialization; perhaps again after some training

Mini-batch gradient descent

$$X = [\underbrace{x^{(1)}, x^{(2)}, \dots, x^{(1000)}}_{X^{\{1\}}} | \underbrace{x^{(1001)}, x^{(1002)}, \dots, x^{(2000)}}_{X^{\{2\}}} | \dots | \dots x^{(m)}]$$

$$Y = [\underbrace{y^{(1)}, y^{(2)}, \dots, y^{(1000)}}_{Y^{\{1\}}} | \underbrace{y^{(1001)}, y^{(1002)}, \dots, y^{(2000)}}_{Y^{\{2\}}} | \dots | \dots y^{(m)}]$$

Mini-batch $t : X^{\{t\}}, Y^{\{t\}}$

Algorithm : Let $m = 5,000,000$ and size of mini-batch = 1000 \rightarrow

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

Forward prop on $X^{\{t\}}$.

$$z^{[t]} = w^{[t]} X^{\{t\}} + b^{[t]}$$

$$A^{[t]} = g^{[t]}(z^{[t]})$$

$$\vdots$$

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

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

Backprop to compute gradients w.r.t $J^{\{t\}}$ (using $(X^{\{t\}}, Y^{\{t\}})$)

$$w^{[t+1]} = w^{[t]} - \alpha dw^{[t]}, \quad b^{[t+1]} = b^{[t]} - \alpha db^{[t]}$$

}

avg $\frac{x}{m}$
(as if $m=1$)

Algorithm

Choosing your mini-batch size :

- If mini-batch size = m : Batch gradient descent $(X, Y) = (X^{\{1\}}, Y^{\{1\}}) \Rightarrow$ too long per iteration.

- If mini-batch size = 1 : Stochastic gradient descent
 $(X^{\{1\}}, Y^{\{1\}}) = (x^{(1)}, y^{(1)}) \dots (X^{\{m\}}, Y^{\{m\}}) = (x^{(m)}, y^{(m)}) \Rightarrow$ lose almost all your speed up from vectorization.
- In practice mini-batch size should be between $1 \dots m \Rightarrow$ Fastest learning.

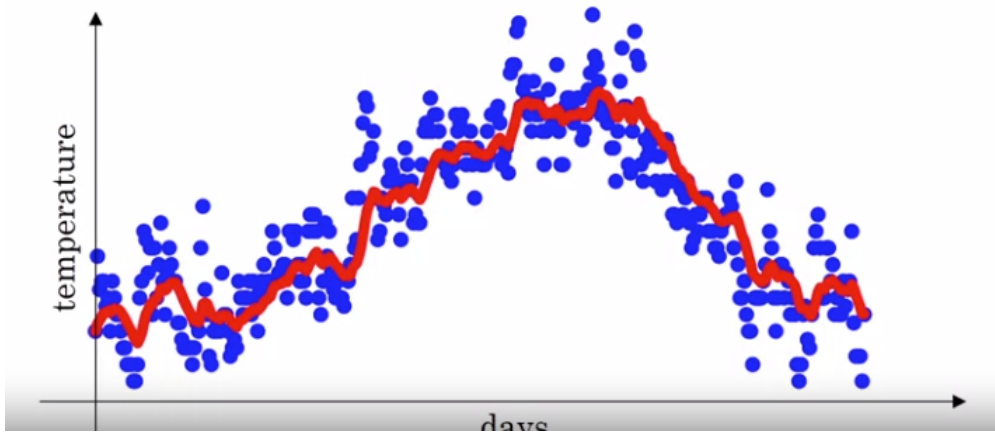
Typical mini-batch size : 64, 128, 256, 512, 1024.

If size of training set is small ($m \leq 2500$) \rightarrow better use batch gradient descent.

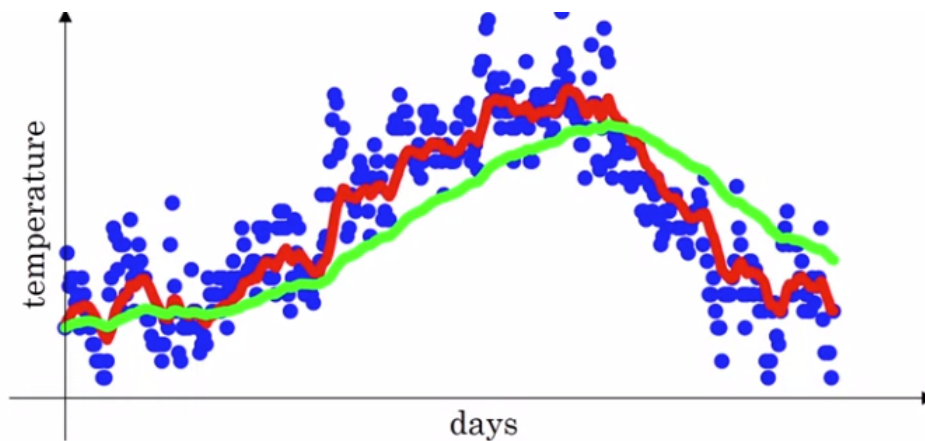
Exponentially weighted averages

θ_t - temperature on day t .

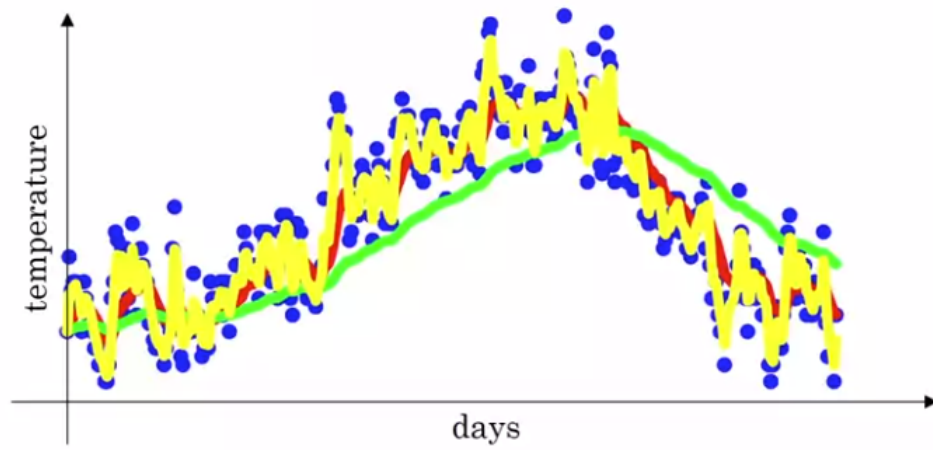
$$v_t = \beta v_{t-1} + (1 - \beta) \theta_t$$



$\beta = 0.9$ (red)



$\beta = 0.98$ (green)



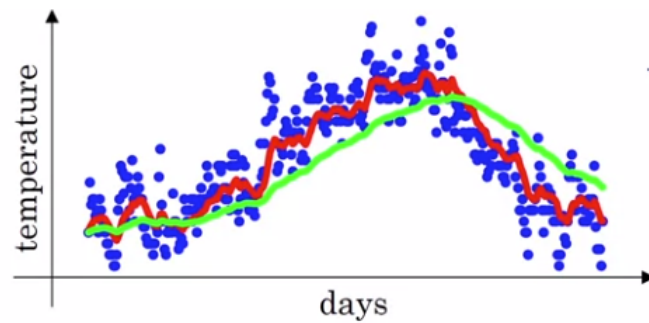
$\beta = 0.5$ (yellow)

$$v_{100} = (1 - \beta)\theta_{100} + (1 - \beta)^2\theta_{99} + \dots + (1 - \beta)^{100}\theta_1$$

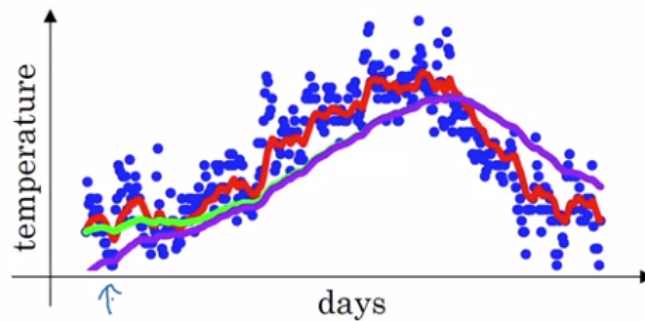
$$\beta = 0.9 \rightarrow (0.9)^{10} \approx \frac{1}{e}, \beta = 0.98 \rightarrow (0.98)^{50} \approx \frac{1}{e}$$

$\beta^{\frac{1}{1-\beta}} \approx \frac{1}{e}$ — we mainly take into account this $\frac{1}{1-\beta}$ number of the last days.

Bias correction in exponentially weighted averages



With bias correction (green & red)



Without bias correction (purple)

$\beta = 0.98$

$$v_0 = 0, v_1 = \underbrace{0.98v_0}_{0} + 0.02\theta_1 \approx 0$$

$$v_1, v_2 \dots v_k \approx 0$$

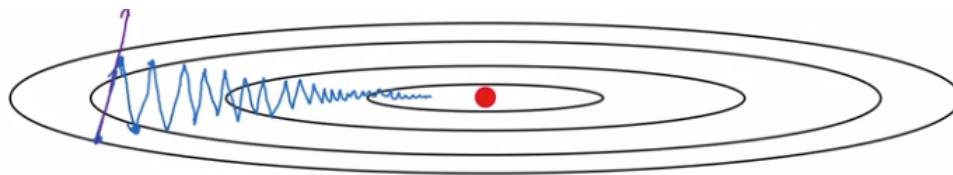
$$\text{Bias correction : } \frac{v_t}{1 - \beta^t}$$

$$t = 2$$

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

$$\frac{v_2}{0.0396} = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396} \approx 0.49\theta_1 + 0.5\theta_2 \not\approx 0$$

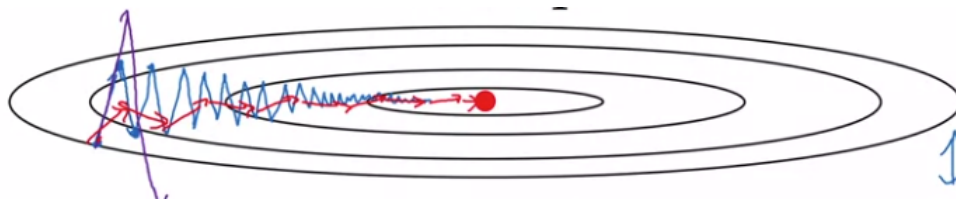
Gradient descent with momentum



Gradient descent (blue)

Want slower learning vertically, faster learning horizontally.

Using the idea with exponentially weighted averages.



Gradient descent with momentum (red)

Algorithm :

On iteration t :

Compute dW, db on the current mini-batch

$$v_{dW} = \beta v_{dW} + (1 - \beta)dW \quad (\text{or batch})$$

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

$$W = W - \alpha v_{dW}, \quad b = b - \alpha v_{db}$$

Algorithm

RMSprop

We have the same problem with gradient descent. Want slower learning vertically, faster learning horizontally.

Algorithm :

On iteration t :

Compute dW, dB on the current mini-batch(or batch)

$$S_{dW} = \beta S_{dW} + (1 - \beta) \underbrace{dW^2}_{\text{elemnt-wise}}$$

$$S_{db} = \beta S_{db} + (1 - \beta) \underbrace{db^2}_{\text{elemnt-wise}}$$

$$w = w - \alpha \frac{dW}{\sqrt{S_{dW} + \varepsilon}}$$

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

Adam optimization

Adam optimization = Gradient descent with momentum + RMSprop

Algorithm :

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

On iteration t :

Compute dW, dB on the current mini-batch(or batch)

$$V_{dW} = \beta_1 V_{dW} + (1 - \beta_1) dW, V_{db} = \beta_1 V_{db} + (1 - \beta_1) db \leftarrow \text{"moment"} \quad \beta_1$$

$$S_{dW} = \beta_2 S_{dW} + (1 - \beta_2) dW^2, S_{db} = \beta_2 S_{db} + (1 - \beta_2) db^2 \leftarrow \text{"RMSprop"} \quad \beta_2$$

$$V_{dW}^{corrected} = \frac{V_{dW}}{1 - \beta_1^t}, V_{db}^{corrected} = \frac{V_{db}}{1 - \beta_1^t}$$

$$S_{dW}^{corrected} = \frac{S_{dW}}{1 - \beta_2^t}, S_{db}^{corrected} = \frac{S_{db}}{1 - \beta_2^t}$$

$$W = W - \alpha \frac{V_{dW}^{corrected}}{\sqrt{S_{dW}^{corrected} + \varepsilon}}, b = b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected} + \varepsilon}}$$

More about Adam optimization : <https://arxiv.org/pdf/1412.6980.pdf>

Learning rate decay

1 epoch = 1 pass through data

$$\alpha = \frac{1}{1 + \text{decay_rate} * \text{epoch_num}}$$

<i>epoch_num</i>	<i>learning_rate</i>
1	0.1
2	0.067
3	0.05
4	0.04
...	...

Other learning rate decay methods :

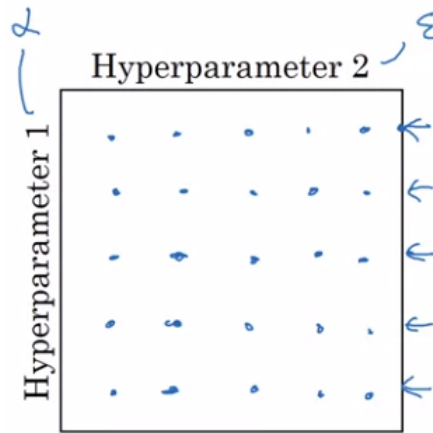
$$\alpha = 0.95^{\text{epoch_num}} \cdot \alpha_0$$

$$\alpha = \frac{k}{\sqrt{\text{epoch_num}}} \cdot \alpha_0$$

$$\alpha = \frac{k}{\sqrt{t}} \alpha_0$$

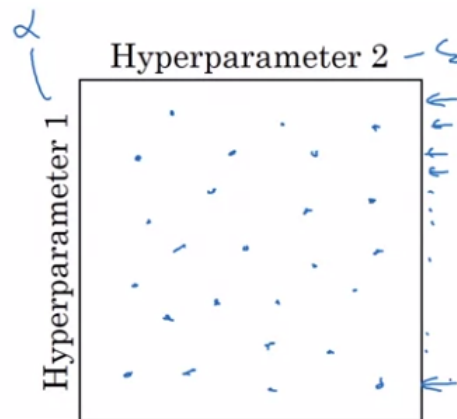
Tunning process

We can iterate over all values(grid).



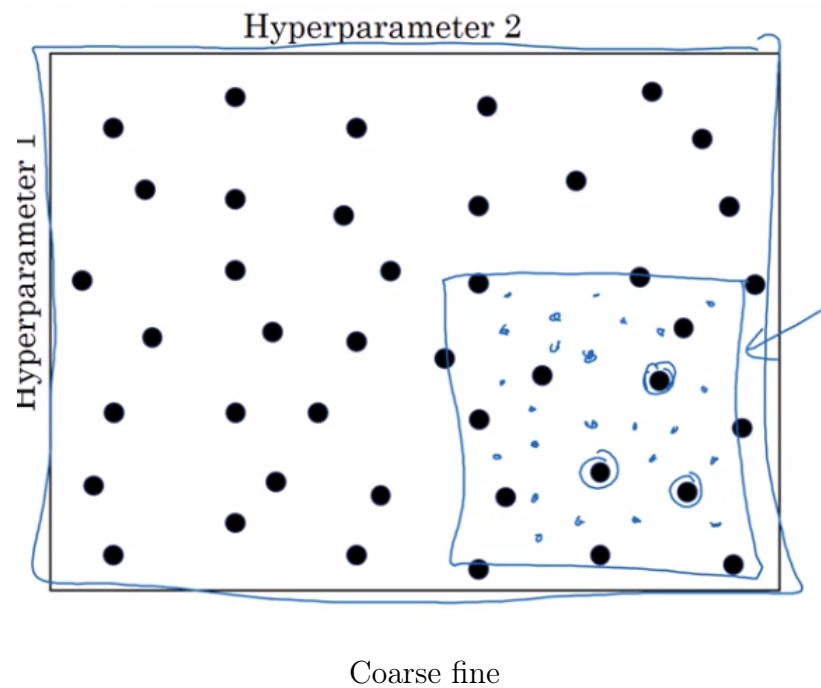
Grid 5×5

But better is choose points at random.



25 random points

Or maybe again at random, but to then focus more resources on searching within this blue square if you're suspecting that the best setting, the hyperparameters, may be in this region. So after doing a coarse sample of this entire square, that tells you to then focus on a smaller square.



Using an appropriate scale to pick hyperparameters

Example 1 :

Pick a random value from $[0.0001; 1]$

$$a = 0.0001 = 10^{-4}, b = 1 = 10^0$$

$$r = -4 * \text{np.random.rand}() \in [-4; 0]$$

$$10^r \in [0.0001; 1]$$

Example 2 :

Pick a random value β from $[0.9; 0.999]$. Similar to pick a random $1 - \beta$ from $[0.001; 0.1]$

$$0.1 = 10^{-1}, 0.001 = 10^{-3} \rightarrow r \in [-1; -3]$$

$$r = -2 * \text{np.random.rand}() \in [-2; 0]$$

$$r = -2 * \text{np.random.rand}() - 1 \in [-3; -1]$$

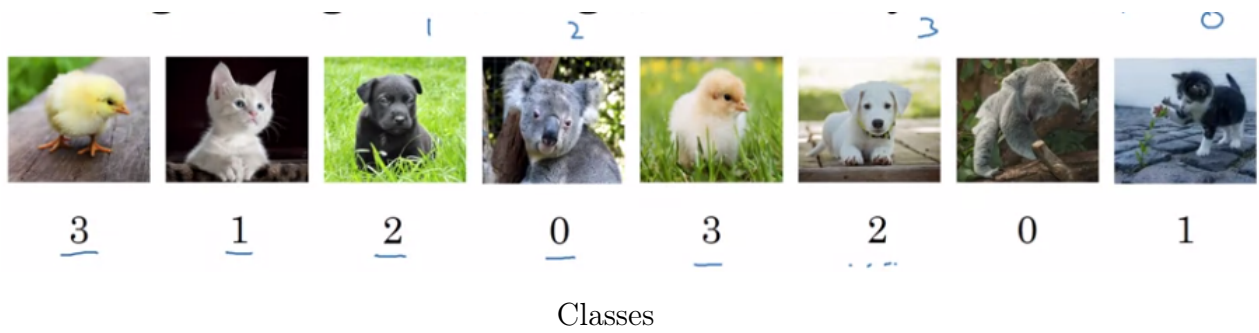
$$\beta = 1 - 10^r$$

Batch normalization

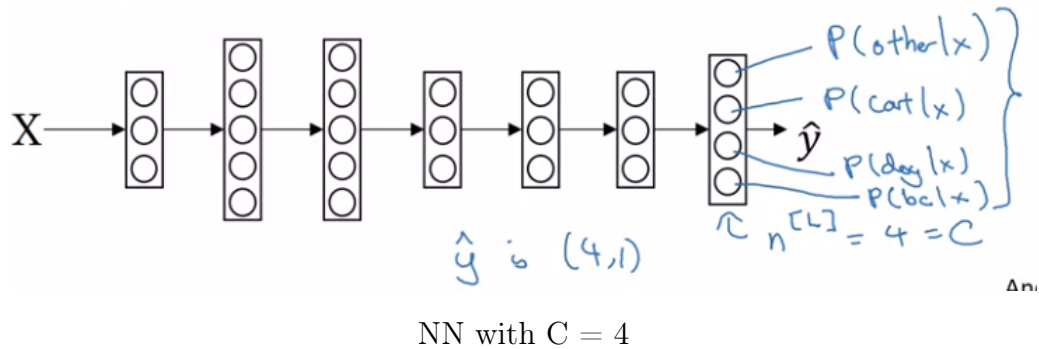
#TODO

Softmax regression

Want to recognize cats, dogs, baby chicks.



$C = \text{\#classes}, [0; C - 1]$



Activation function on softmax layer :

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

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

$$\underbrace{a^{[L]}}_{(4 \times 1)} = \frac{e^{(Z^{[L]})}}{\sum_{j=1}^4 t_j}$$

$$Z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$$

$$t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}, \quad \sum_{j=1}^4 t_j = 176.3$$

$$a^{[L]} = \frac{t}{176.3}$$

Example

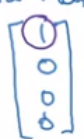
Training a softmax classifier

Hard max :

$$a^{[L]} = g^{[L]}(z^{[L]}) = \begin{bmatrix} e^5/(e^5 + e^2 + e^{-1} + e^3) \\ e^2/(e^5 + e^2 + e^{-1} + e^3) \\ e^{-1}/(e^5 + e^2 + e^{-1} + e^3) \\ e^3/(e^5 + e^2 + e^{-1} + e^3) \end{bmatrix} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$$

"soft max"

"hard max"



Hard max

Loss function :

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

Y - hard max, \hat{Y} - soft max.

(4,1) (4,1)

$y^{(1)} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ - cat $y_2 = 1$
 $y_1 = y_3 = y_4 = 0$

$a^{(1)} = \hat{y}^{(1)} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix}$ $C=4$

$\mathcal{L}(\hat{y}, y) = - \sum_{j=1}^4 y_j \log \hat{y}_j$ $J(w^{(1)}, b^{(1)}, \dots) = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$

small $-y_2 \log \hat{y}_2 = -\log \hat{y}_2$ make \hat{y}_2 big.

$Y = [y^{(1)} \ y^{(2)} \ \dots \ y^{(n)}]$ $\hat{Y} = [\hat{y}^{(1)} \ \dots \ \hat{y}^{(n)}]$

$= \begin{bmatrix} 0 & 0 & 1 & \dots \\ 1 & 0 & 0 & \dots \\ 0 & 1 & 0 & \dots \\ 0 & 0 & 0 & \dots \end{bmatrix}$ $= \begin{bmatrix} 0.3 & \dots \\ 0.2 & \dots \\ 0.1 & \dots \\ 0.4 & \dots \end{bmatrix}$

(4,m) (4,m)

Andrew

Training a softmax

$$\underbrace{dZ^{[L]}}_{(4 \times 1)} = \hat{Y} - Y \text{ - back-prop softmax.}$$