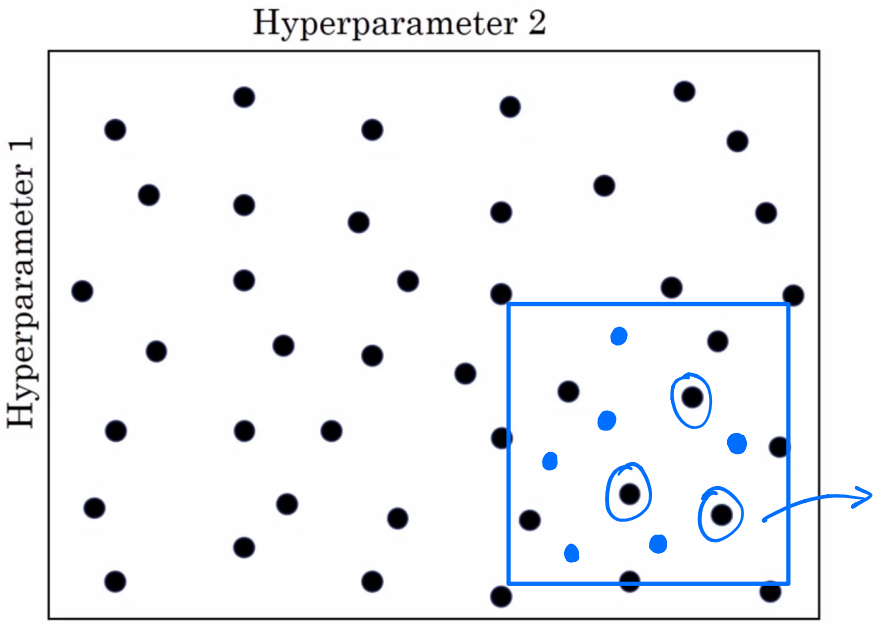
Tuning Hyperparameters

in order of tuning importance:

- → learning rate
- → mini-batch size
- → # of hidden units
- → # of layers
- → learning rate decay
- → β_1, β_2, ε

★ use random values not a grid to try out new hyperparameter values

coarse to fine:



→ Sample more densely adding more sample points

Appropriate scale for hyperparameters

$\alpha = 0.0001 \ldots 1$

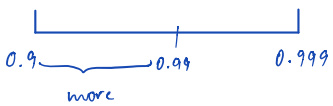
sample points on the log scale

$r = -4 * np.random.rand()$ ← $r \in [-4, 0]$

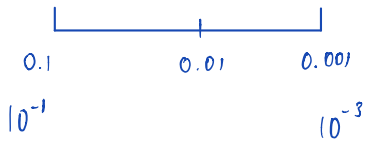
$\alpha = 10^r$ ← $10^{-4} \dots 10^0$

Hyperparameters for exponentially weighted averages

$\beta = 0.9 \dots 0.999$



$1-\beta = 0.1 \dots 0.001$



$r \in [-3, -1]$

$1-\beta = 10^r$

$\beta = 1 - 10^r$

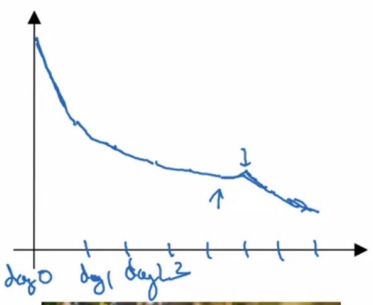
$\beta = 0.9 \rightarrow 0.9005$ no change

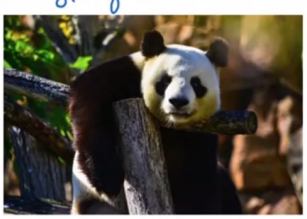
$\beta = 0.999 \rightarrow 0.9995$ huge impact

* more sensitive to small changes when β is close to 1

this makes us sample β more densely as it gets close to 1 $\left(\frac{1}{1-\beta}\right)$

Babysitting one model



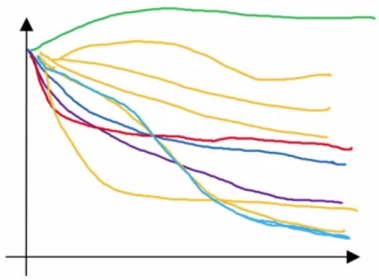


Panda ←

training one model and changing the values of the hyperparameters while running

computational power <

Training many models in parallel



Caviar ←

training alot at once with different values for hyperparameters

Andrew

Batch Normalization

Normalizing inputs to speed up learning

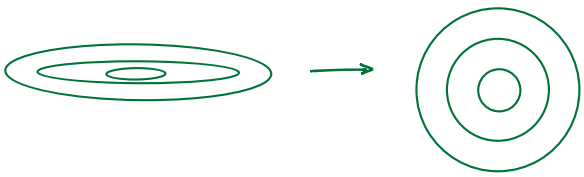
Linear Regression:

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

$$X = X - \mu$$

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

$$X = X / \sigma^2$$



* makes it faster to converge
* trains w and b more efficiently

Deep Neural Networks: batch normalization applies normalization to input units in the deep NNs runs the model much faster

Implementation:

for given intermediate values in NN $\underbrace{z^{(1)}, \ldots, z^{(m)}}_{z^{[l](i)}}$

$$\mu = \frac{1}{m} \sum z^{(i)}$$

$$\sigma = \frac{1}{m} \sum (z^{(i)} - \mu)^2$$

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

→ has mean 0 and variance 1

* we don't want hidden units to always have mean 0 and variance 1
we want a different distribution, so we do

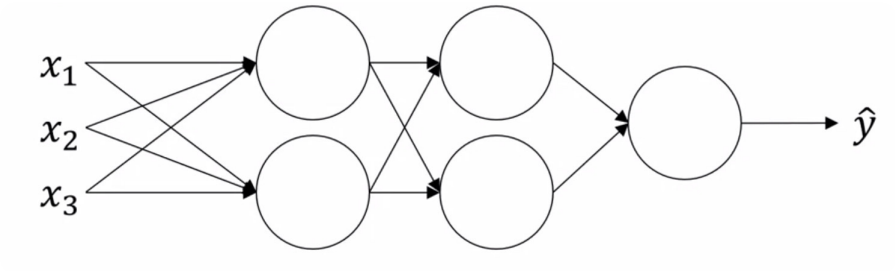
$$\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta$$

(we update them) learnable parameters of the model, it helps to set the mean of z to what we want

if $\gamma = \sqrt{\sigma^2 + \epsilon}$
and $\beta = \mu$
$\tilde{z}^{(i)} = z^{(i)}$
} for example

→ we use $\tilde{z}^{(i)}$ instead of $z^{(i)}$ for any layer l

Fitting Batch Norm to a Network



$X \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[\text{Batch Norm (BN)}]{\gamma^{[1]}, \beta^{[1]}} \tilde{z}^{[1]} \longrightarrow a^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \xrightarrow{w^{[2]}, b^{[2]}} z^{[2]} \xrightarrow[BN]{\gamma^{[2]}, \beta^{[2]}} \tilde{z}^{[2]} \longrightarrow a^{[2]}$

Parameters: $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]} \ldots w^{[L]}, b^{[L]}$

$\beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]} \ldots \beta^{[L]}, \gamma^{[L]}$

update:

$d\beta^{[l]}$

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

Working with mini-batches

$X^{\{1\}} \longrightarrow z^{[1]} \xrightarrow[BN]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \ldots \longrightarrow$ one step of gradient descent

$X^{\{2\}} \longrightarrow z^{[2]} \xrightarrow[BN]{\beta^{[2]}, \gamma^{[2]}} \tilde{z}^{[2]} \ldots$ (independent of the β and γ values of other mini batches)

$\vdots$

$X^{\{t\}}$

$z^{[l]} = w^{[l]} a^{[l-1]} + \cancel{b^{[l]}}$ — no use of b after batch norm because it is just a constant

$z^{[l]} = w^{[l]} a^{[l-1]}$

$z^{[l]}_{norm} \ldots$

$\tilde{z}^{[l]} = \gamma^{[l]} z^{[l]}_{norm} + \beta^{[l]}$

* any constant added will get cancelled out by the mean subtractions step

Implementing Gradient Descent

for $t=1 \ldots\ldots$ # of mini batches

compute forward prop.

in each hidden layer, use BN to replace $z^{[l]}$ with $\tilde{z}^{[l]}$

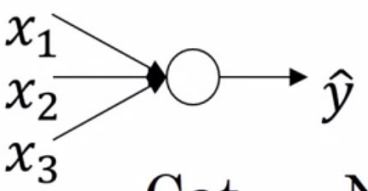
use backprop. to compute $dw^{[l]}$, ~~$db^{[l]}$~~, $d\beta^{[l]}$, $d\gamma^{[l]}$

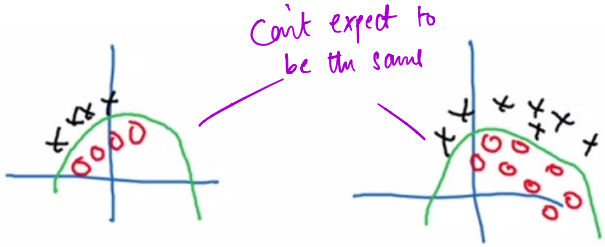
update parameters:

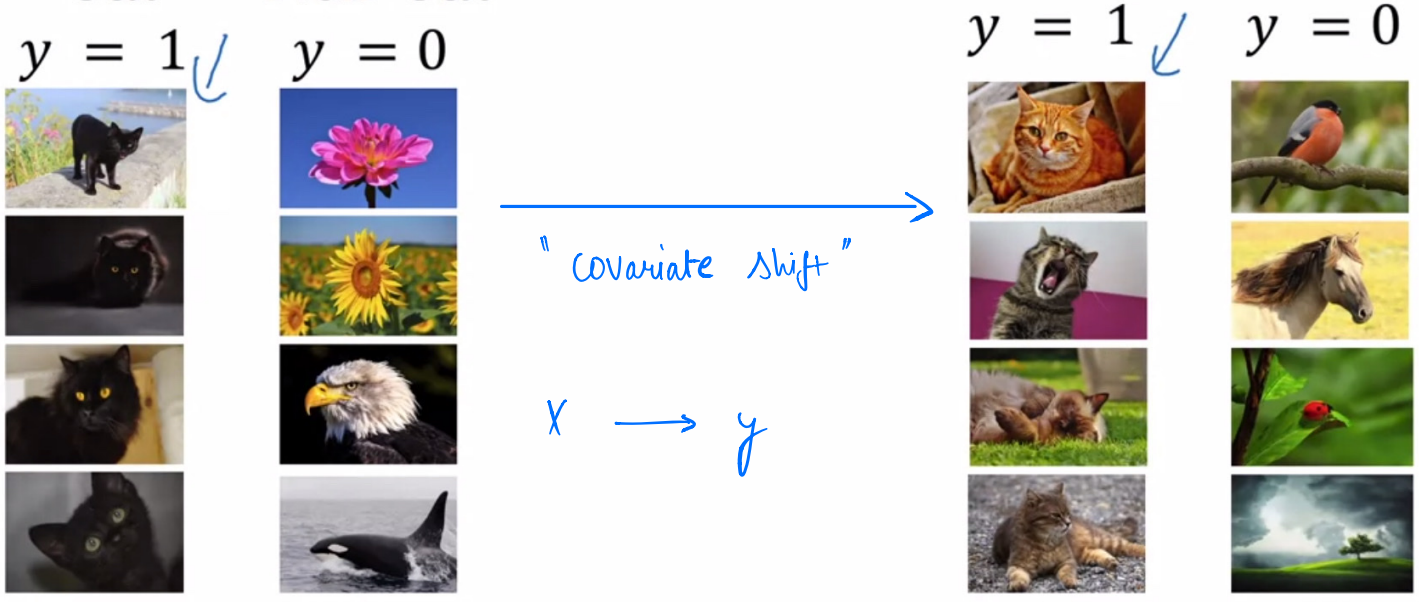
$$w^{[l]} = w^{[l]} - \alpha\, dw^{[l]}$$
$$\beta^{[l]} = \beta^{[l]} - \alpha\, d\beta^{[l]}$$
$$\gamma^{[l]} = \gamma^{[l]} - \alpha\, d\gamma^{[l]}$$

works with momentum / RMS prop / Adam

Working of Batch Norm



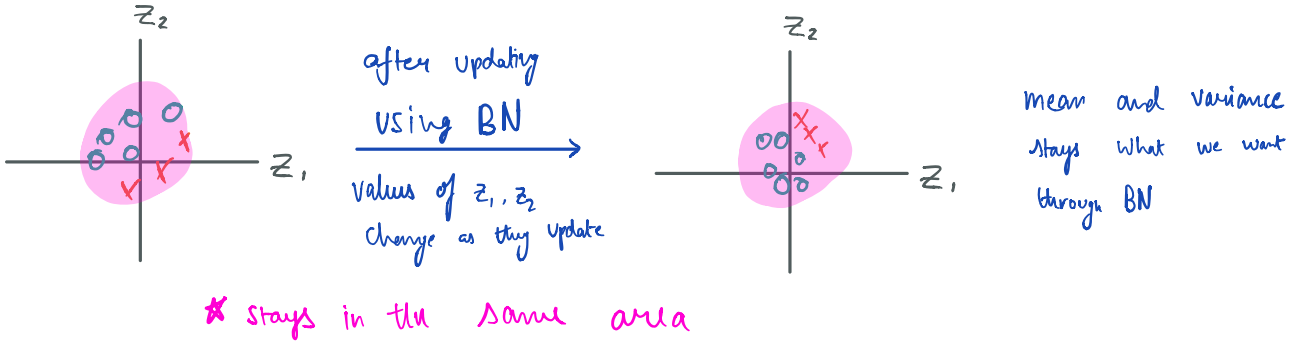




* if distribution of x changes, retraining of the model is needed

→ talking about hidden units, the values of the hidden units shifts around with each iteration of the gradient descent. } problem of covariate shift

→ batch norm reduces the amount of distribution of these hidden units



* stays in the same area

- reduces the problem of input values changing drastically
- causes stability in the change of these values, the change is less
- later layers have a more firm ground stand on, i.e, more stable activation values
- each minibatch has its own β and γ

Batch Norm as regularization

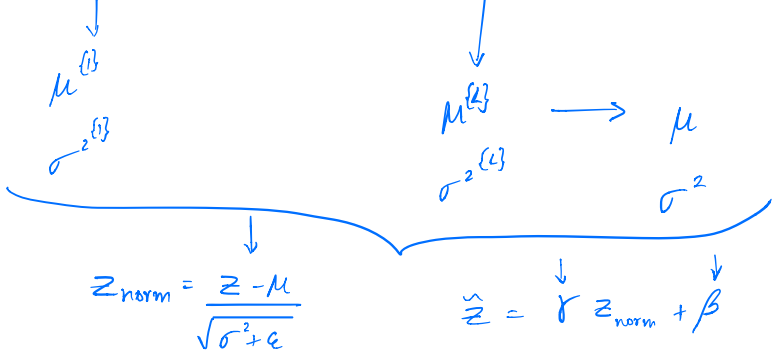
- Each mini-batch is scaled by the mean/variance computed on just that mini-batch.
- This adds some noise to the values $z^{[l]}$ within that minibatch. So similar to dropout, it adds some noise to each hidden layer's activations.
- This has a slight regularization effect.

Batch Norm at test time

- There are no mini-batches in the test set, so we have a separate estimate of μ, σ^2

μ, σ^2: estimate using exponentially weighted average (across mini-batches)

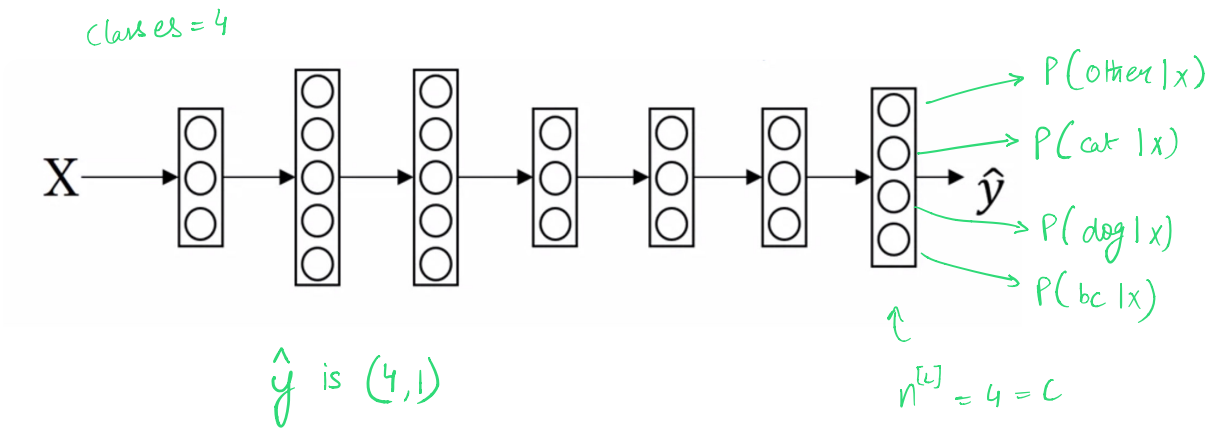
$X^{\{1\}}, X^{\{2\}}, X^{\{3\}}, \ldots, X^{\{L\}}$



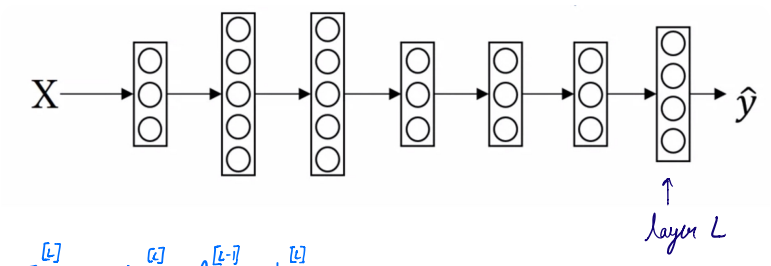
- We keep a running average of μ and σ^2 across mini-batches

Softmax Regression : it is a type of an activation function that helps transform z

- C = # classes
- $n^{[L]} = C$ → the last layer in the NN



Softmax Layer



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

Activation function:

$$t = e^{z^{[l]}} \qquad (4,1)$$

$$a^{[l]} = \frac{e^{z^{[l]}}}{\sum_{i=1}^{4} t_i} \qquad (4,1)$$

i^{th} element: $a_i^{[l]} = \frac{t_i}{\sum_{i=1}^{4} t_i}$

example:

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

$$t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}$$

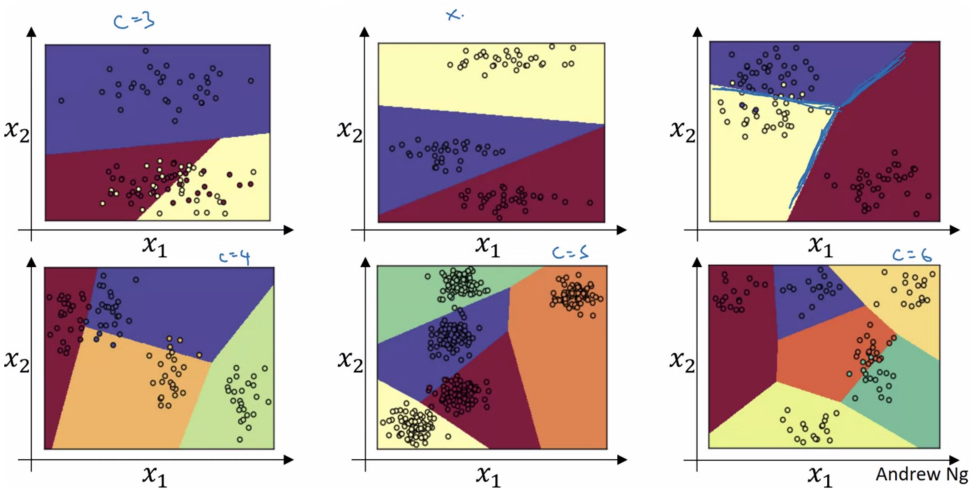
$$\sum_{i=1}^{4} t_i = 176.3$$

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

$$a^{[l]} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$$

"hard max" → $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ ← biggest element gets 1, rest get 0

Graph examples: linear functions are separating



Training Softmax Classifier

- Softmax regression generalizes logistic regression to C classes
  - → if C=2, essentially reduces to logistic regression

Loss & Cost Function

$$y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \rightarrow \text{cat} \qquad a^{[L]} = \hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix} \rightarrow \text{doing badly}$$

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

$$-y_2 \log \hat{y}_2 = -\log \hat{y}_2$$

making this small, making y_2 big

$$J(w^{[1]}, b^{[1]}, \ldots) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

$$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & \cdots & y^{(m)} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & \cdots \\ 1 & 0 & 0 & \\ 0 & 1 & 0 & \\ 0 & 0 & 0 & \end{bmatrix} \quad (4, m)$$

$$\hat{Y} = \begin{bmatrix} \hat{y}^{(1)} & \hat{y}^{(2)} & \cdots & \hat{y}^{(m)} \end{bmatrix} = \begin{bmatrix} 0.3 & \cdots \\ 0.2 & \\ 0.1 & \\ 0.4 & \end{bmatrix} \quad (4, m)$$

Gradient Descent

Back prop:

$$dz^{[L]} = \hat{y} - y \qquad (4, 1)$$

→ same dimensions for rest of the derivatives

Deep learning frameworks

- Caffe/Caffe2
- CNTK
- DL4J
- Keras
- Lasagne
- mxnet
- PaddlePaddle
- TensorFlow
- Theano
- Torch

Choosing deep learning frameworks

- Ease of programming (development and deployment)
- Running speed
- Truly open (open source with good governance)