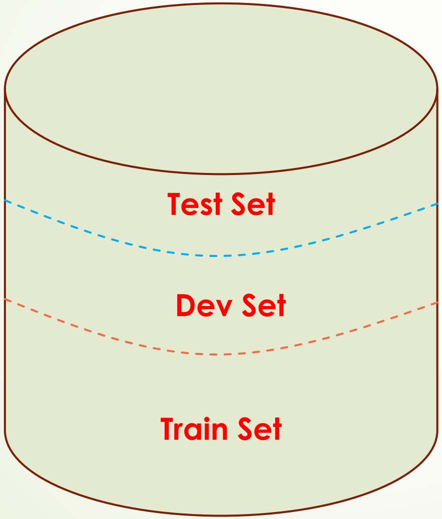
Introduction to Deep Learning
Chapter 3: Tuning DNN

Pao-Ann Hsiung

National Chung Cheng University

Contents

Training vs. Development vs. Test sets

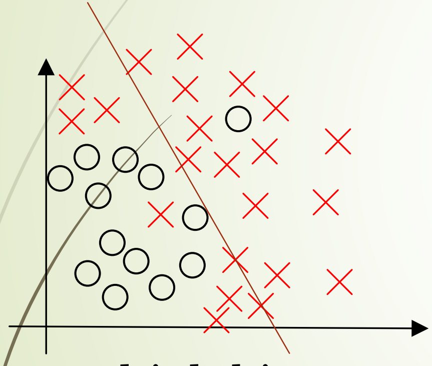


- Traditionally best practice:
  - Train : Test = **70:30**
  - Train : Dev : Test = **60:20:20**
- Modern big data era:
  - Total dataset size: 1,000,000
  - Dev set: big enough to evaluate different algorithm choices, say 10,000 more than enough
  - Test set: big enough to test accuracy, say 10,000 more than enough
  - Thus, train : dev : test = **98 : 1 : 1**
  - Or even, **99.5 : 0.4 : 0.1**

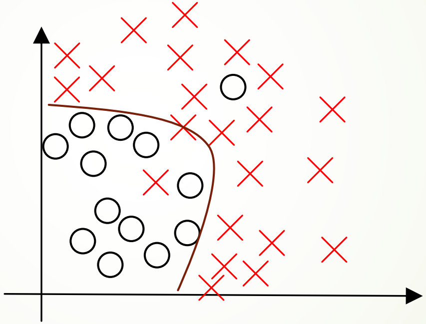
Mismatched train/test distribution

- Training set
  - cat pictures downloaded from webpages (high resolution, professional)
- Dev/test sets
  - cat pictures from your app (low resolution, blurry)
- The distributions are different for the above sets
- Make sure that the **dev** and **test** come from the SAME distribution because evaluations should matched while developing and testing.
- It is fine that the training set is different from dev/test sets.

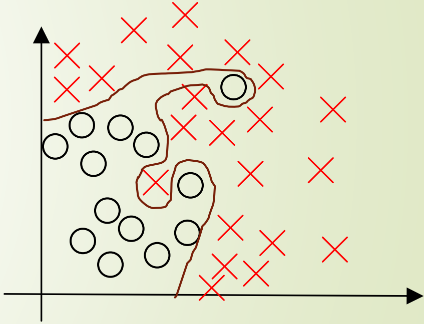
Bias vs. Variance



high bias



“just right”



high variance

Bias and Variance

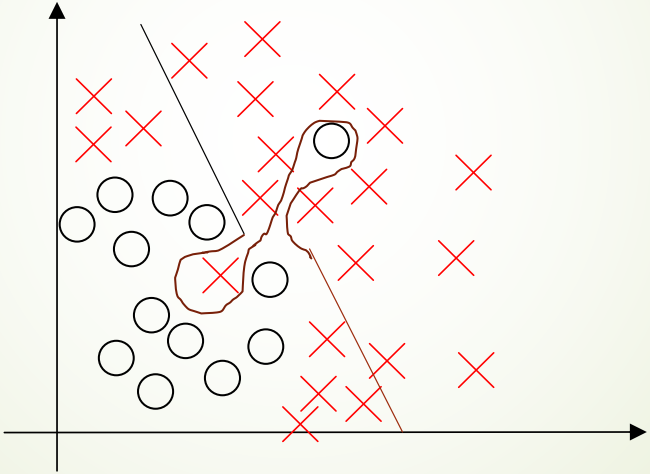
Classification Problem for Dogs:

✓

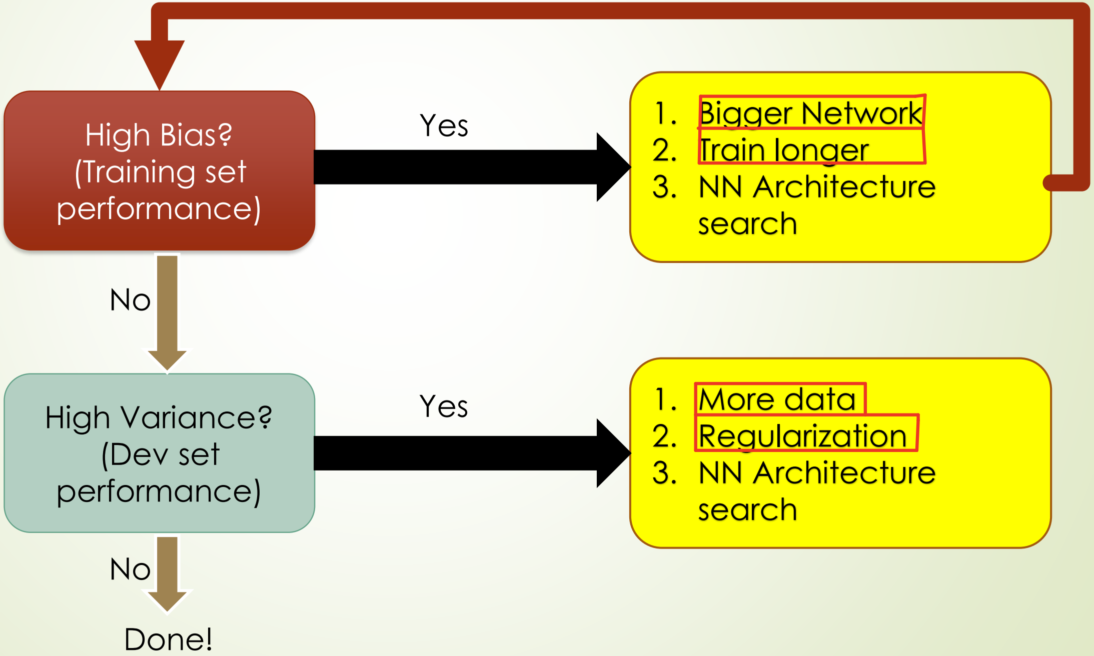
Error	Case 1	Case 2	Case 3	Case 4
Train set	1%	15%	15%	0.5%
Dev set	11%	16%	30%	1%
Variance	high		high	low
Bias		high	high	low

Note: Assuming human error is nearly 0%

High bias and high variance



How to solve the high bias/variance problem?



High Bias?
(Training set performance)

Yes

1. Bigger Network
2. Train longer
3. NN Architecture search

No

High Variance?
(Dev set performance)

Yes

1. More data
2. Regularization
3. NN Architecture search

No

Done!

Contents

- Regularization and Dropout
- Optimization
- Gradient Checking
- Momentum, RMSprop, Adam, Learning rate decay
- Hyperparameter tuning
- Batch Normalization
- Softmax Regression

Regularization

- Overfitting
  - Can be solved using Regularization or More Data
  - Sometimes it is difficult to get more data, so regularization could be a good

Regularization for Logistic Regression

- Cost function: $\min_{w,b} J(w,b)$

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

Used more often

- L2 regularization: $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$
- L1 regularization: $\|w\|_1 \; = \sum_{j=1}^{n_x} |w_j|$

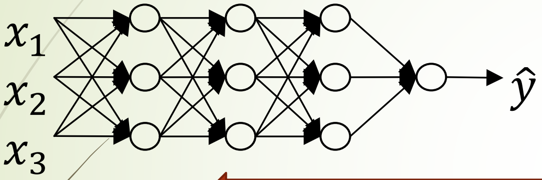
- λ is the **regularization parameter**
- In this course, we will use "lamd" to represent λ. (In Python, lambda is reserved keyword)

Regularization for Neural Networks

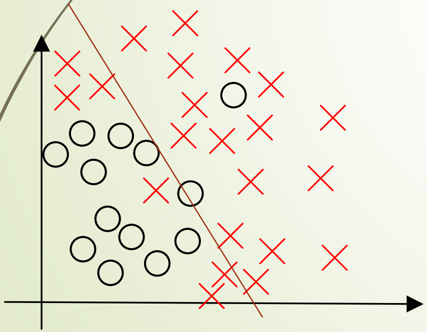
- $J(w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \boldsymbol{\frac{\lambda}{2m}\sum_{l=1}^{L}\|w^{[l]}\|_F^2}$
- **Frobenius** Norm: $\|w^{[l]}\|_F^2 = \sum_{i=1}^{n[l-1]}\sum_{j=1}^{n[l]} (w_{ij}^{[l]})^2$
- Back-propagation: $\frac{\partial J}{\partial W^{[l]}} = dW^{[l]} = \left(\frac{1}{m} dZ^{[l]} A^{[l]^T}\right) + \boldsymbol{\frac{\lambda}{m} W^{[l]}}$
- Weight updates: $W^{[l]} = W^{[l]} - \alpha dW^{[l]}$
- L2 normalization is also called “weight decay” because
  - $W^{[l]} = W^{[l]} - \alpha\left[\left(\frac{1}{m} dZ^{[l]} A^{[l]^T}\right) + \boldsymbol{\frac{\lambda}{m} W^{[l]}}\right]$
  - $= W^{[l]} - \frac{\alpha\lambda}{m} W^{[l]} - \alpha\left(\frac{1}{m} dZ^{[l]} A^{[l]^T}\right)$
  - $= \left(1 - \frac{\alpha\lambda}{m}\right) W^{[l]} - \alpha(\frac{1}{m} dZ^{[l]} A^{[l]^T})$

Less than 1

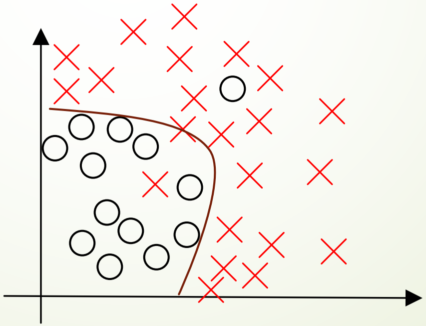
How does regularization prevent overfitting?



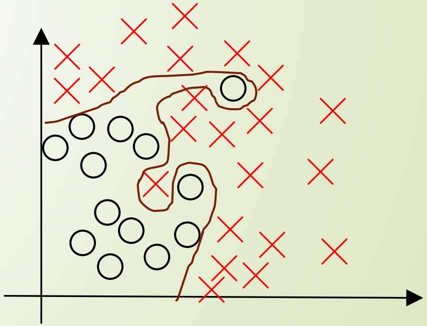
- $J(w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L} \|w^{[l]}\|_F^2$
- Weights **decay**, become very small, i.e. ➔ 0!
- Some nodes have lesser weights …
- Network becomes simpler, thus regularized!



high bias

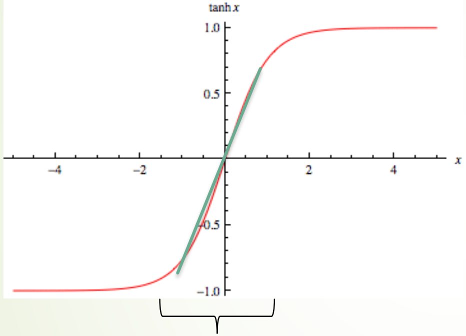


"just right"



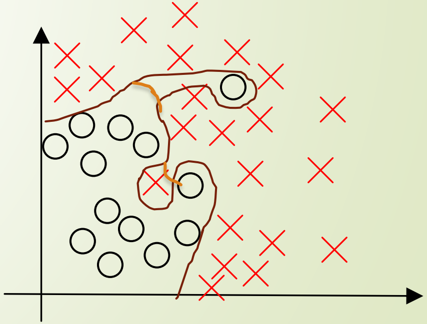
high variance

How does regularization prevent overfitting?



More linear

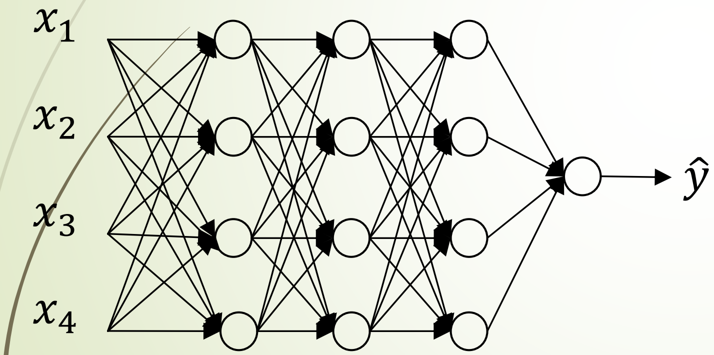
- $\lambda \uparrow \implies W^{[l]} \downarrow \implies z^{[l]} \downarrow$
- Activation values become more linear
- Each layer is more linear
- Full network is more linear

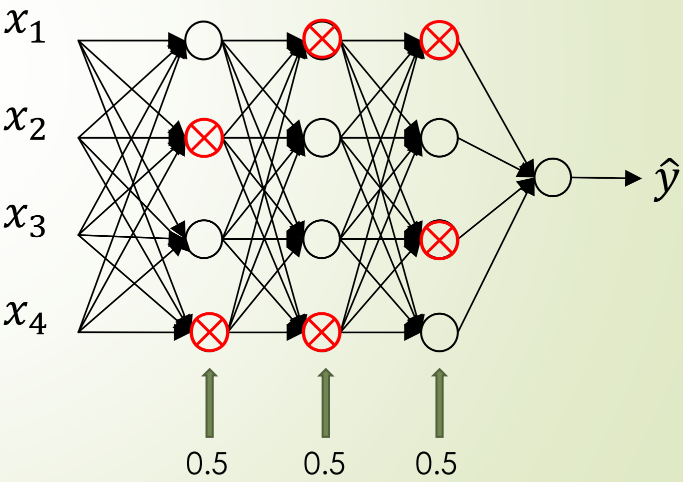


Dropout

減少node但權重不變

- Suppose dropout rate is 0.5, drop out 0.5 nodes in each layer for each sample
- For different samples, drop out different nodes in each layer.



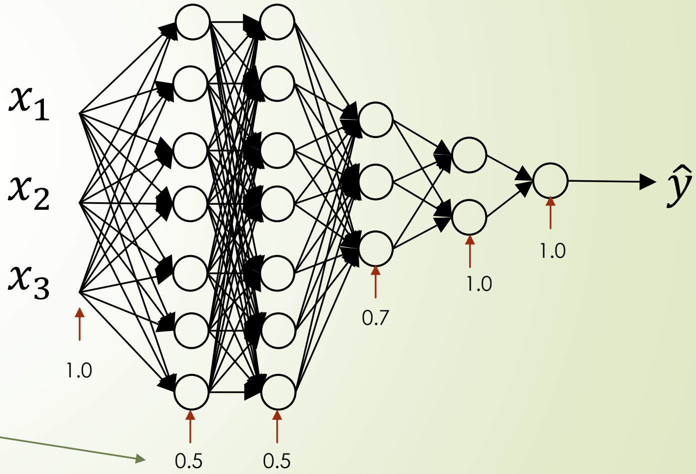


Inverted Dropout: most common version

- Suppose dropout is applied to layer 3
- keep_prob = 0.8 (probability a node will be kept)
- $d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep\_prob$
  - A vector to decide which nodes to dropout
- $a3 = np.multiply(a3, d3)$
- $a3 /= keep\_prob$
  - Pump up the activation values by keep_prob to maintain the expected values
- $z^{[4]} = w^{[4]}.a^{[3]} + b^{[4]}$
- Example: 100 units ➔ 20 units shut off
- Dropout different hidden units in different iterations

Some notes

- No dropout during test time
  - Would add noise during predictions is dropout is used during test time
- Why dropout works?
  - Regularizes the network
  - Reduces the dependence on some particular feature (input node)
    - Dropout spreads out the weights
- Can use different dropout keep_probs for different layers
- Cost function not well-defined because of the weights randomly changed

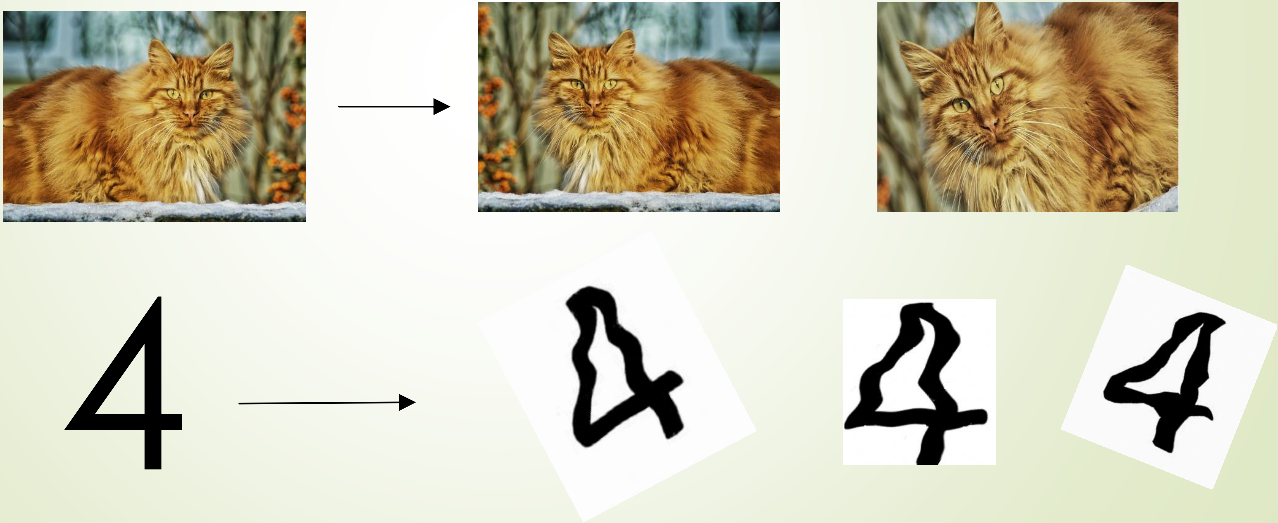


每層layer
的機率一樣

但權重
會改變

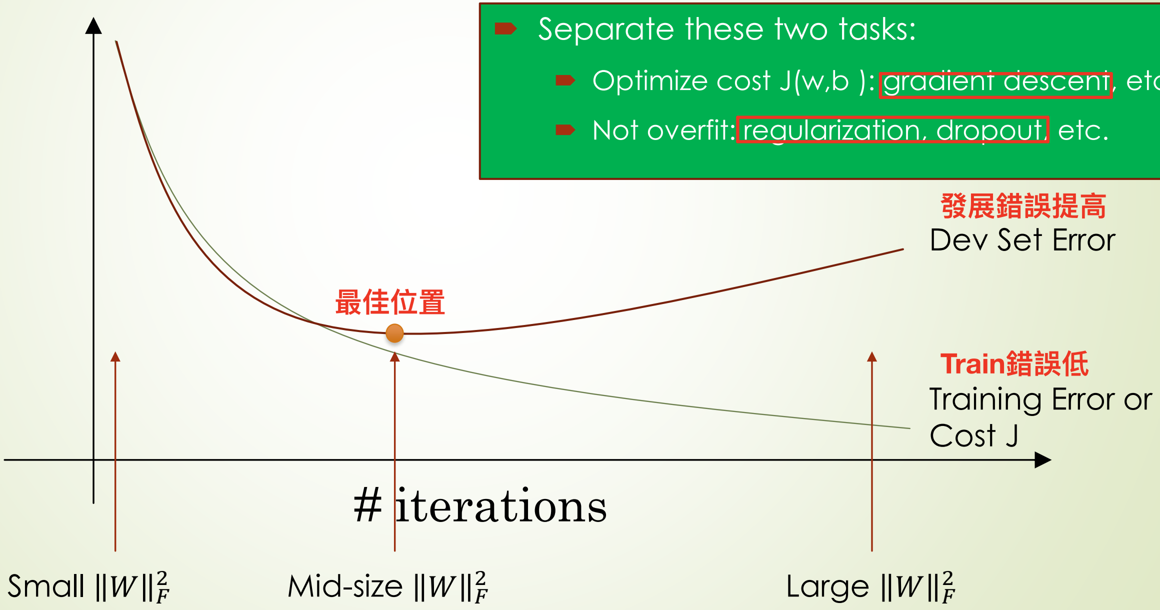
Other regularization methods: Data Augmentation

- Data Augmentation 做資料
  - Horizontal flipping or distortion of images



Other Regularization Methods: Early Stopping

- Separate these two tasks:
  - Optimize cost J(w,b): gradient descent, etc.
  - Not overfit: regularization, dropout, etc.



Contents

Normalizing Inputs
(similar to feature scaling)

- Normalizing inputs

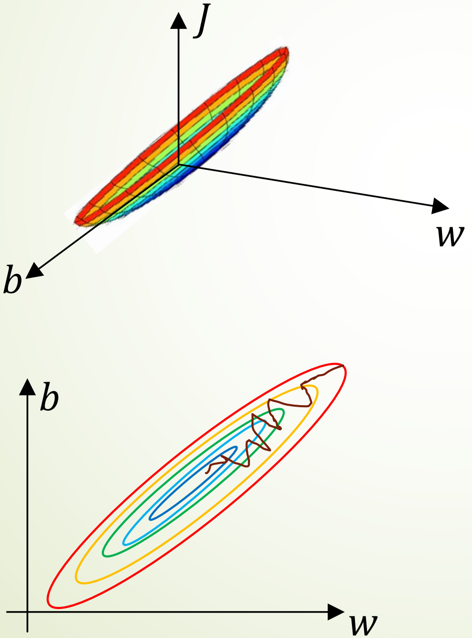
$$x_{j,std} = \frac{x_j - \mu_j}{\sigma_j}$$

where μ_j is the sample mean of the feature x_j and σ_j the standard deviation.

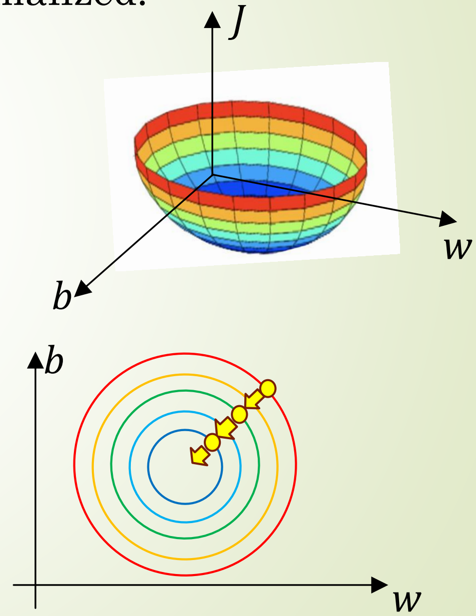
- After normalization, the inputs will have unit variance and centered around mean zero.

Why normalization?

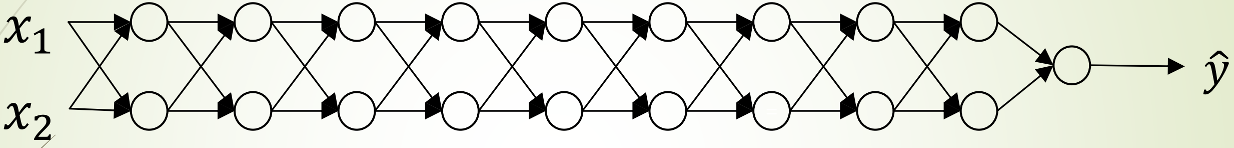
Unnormalized:



Normalized:



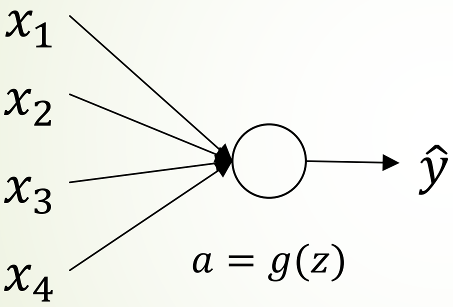
Vanishing/exploding Gradients



線性　　誤差

- Suppose g(z) = z, b = 0
- $y = W^{[1]}W^{[2]} \dots W^{[L]}x$
- $W^{[1]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix} \rightarrow \hat{y} = W^{[1]} \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{L-1} x \Rightarrow 1.5^L x \Rightarrow \hat{y} \Uparrow (\mathit{increases\ exponentially})$
- $W^{[1]} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \rightarrow \hat{y} = W^{[1]} \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}^{L-1} x \Rightarrow 0.5^L x \Rightarrow \hat{y} \Downarrow (\mathit{decreases\ exponentially})$
- Partial Solution: Careful choice of initial weights!

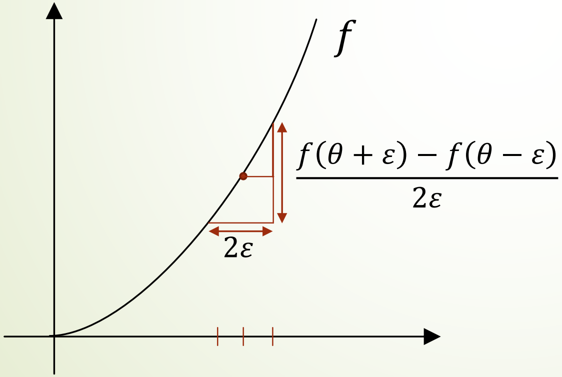
Weight Initialization for deep networks



- $z = \underline{w_1 x_1 + w_2 x_2 + \cdots + w_n x_n}$
- Large n ➔ smaller w_i 要讓值在1附近
- $\boxed{Var(w) = \frac{1}{n}}$
- ReLU Initialization
  - $W^{[l]} = np.random.randn(shape) * np.sqrt(\frac{2}{n^{[l-1]}})$ for ReLU
- Xavier Initialization
  - $W^{[l]} = np.random.randn(shape) * np.sqrt(\frac{1}{n^{[l-1]}})$ for tanh
- Other Initialization
  - $W^{[l]} = np.random.randn(shape) * np.sqrt(\frac{2}{n^{[l-1]} * n^{[l]}})$ for others

Contents

Numerical Approximation of Gradients



- $f(\theta) = \theta^3$
- $\frac{f(\theta+\varepsilon)-f(\theta-\varepsilon)}{2\varepsilon} \approx g(\theta)$
- Suppose $\theta = 1, \varepsilon = 0.01$
- $\frac{(1.01)^3-(0.99)^3}{2(0.01)} = \boxed{3.0001 \approx 3}$
- $\boxed{g(\theta) = 3\theta^2 = 3}$
- Approx. Error = 3.0001 – 3 = 0.0001
- For single sided difference, the result is 3.0301, approx. error = 0.0301
- Thus, double sided difference is much more accurate than single sided
- Will use this double sided difference for gradient checking

Gradient Checking

- Take $W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L]}$ and reshape them into a big vector θ
  - Concatenate all of them!
- Take $\mathrm{d}W^{[1]}, db^{[1]}, \dots, dW^{[L]}, db^{[L]}$ and reshape them into a big vector $\mathrm{d}\theta$
  - Concatenate all of them!
  - Same dimensions as the above
- Is $\mathrm{d}\theta$ the gradient (slope) of J?

Gradient Checking

- For each i:
  - $d\theta_{approx}[i] = \frac{J(\theta_1,\theta_2,\ldots,\theta_i+\varepsilon,\ldots)-J(\theta_1,\theta_2,\ldots\theta_i-\varepsilon,\ldots)}{2\varepsilon} \quad \approx? \;\; d\theta[i] = \frac{\partial J}{\partial \theta_i}$
- Check $\frac{\left\|d\theta_{approx}-d\theta\right\|_2}{\left\|d\theta_{approx}\right\|_2+\left\|d\theta\right\|_2} \quad \approx$
  - $10^{-7} \quad \Longrightarrow \quad Great!$
  - $10^{-5} \quad \Longrightarrow \quad Check!$
  - $10^{-3} \quad \Longrightarrow \quad Worry!$
- (here, assume $\varepsilon = \; 10^{-7}$)

Gradient Checking Notes

- Don't use grad check in training
  - Use it only to DEBUG!
- If algorithm fails grad check, look at components to try to identify bug
  - $db^{[l]},\ or\ dW^{[l]}$ which ones differ during grad check
- Remember regularization
- Doesn't work with dropout

隨機重置activate function

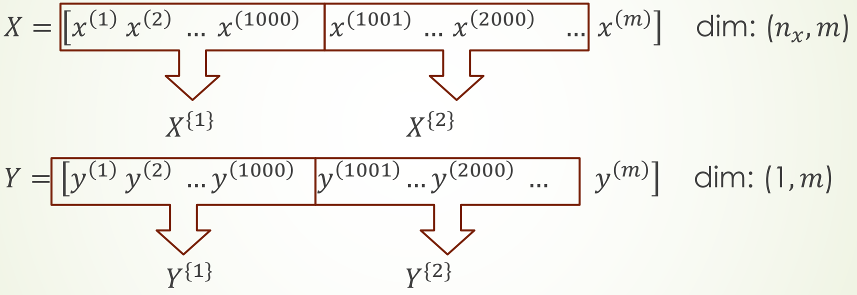
  - Turn off dropout, grad check, turn on dropout
- Run at random initialization; perhaps again after some training

Contents

- Regularization and Dropout
- Optimization
- Gradient Checking
- Momentum, RMSprop, Adam, Learning rate decay
- Hyperparameter tuning
- Batch Normalization
- Softmax Regression

Mini-batch gradient descent

- Vectorization allows you to efficiently compute on *m* examples



- $X = [x^{(1)}\ x^{(2)}\ \ldots\ x^{(1000)}\ x^{(1001)}\ \ldots\ x^{(2000)}\ \ldots\ x^{(m)}]$ dim: (n_x, m)
- $X^{\{1\}}$ $X^{\{2\}}$
- $Y = [y^{(1)}\ y^{(2)}\ \ldots y^{(1000)}\ y^{(1001)} \ldots y^{(2000)}\ \ldots\ y^{(m)}]$ dim: $(1, m)$
- $Y^{\{1\}}$ $Y^{\{2\}}$
- What if m = 5,000,000?
- Ans: 5,000 minibatches of 1,000 each!
- Mini-batch t: $X^{\{t\}}, Y^{\{t\}}$

Mini-batch gradient descent

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

- Forward propagation on $\boxed{X^{\{t\}}}$
  - $Z^{[1]} = W^{[1]}X^{\{t\}} + b^{\{l\}}$
  - $A^{[1]} = g^{[1]}(Z^{[1]})$
  - $\dots$
  - $A^{[L]} = g^{[L]}(Z^{[L]})$
- Compute cost: $J = \frac{1}{1000}\sum_{i=1}^{m} \mathcal{L}\underbrace{\left(\hat{y}^{(i)}, y^{(i)}\right)}_{} + \underbrace{\frac{\lambda}{2*1000}\sum_{l} \left\|W^{[l]}\right\|_F^2}_{}$

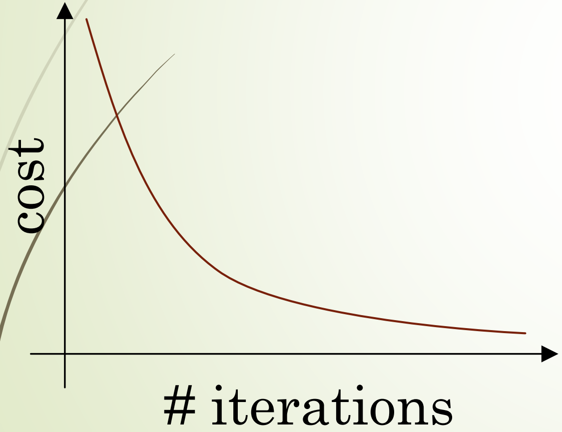
  預測與實際差異　Regulation
- Back propagation to compute gradients wrt $J^{\{t\}}$ using $(X^{\{t\}}, Y^{\{t\}})$
- Weight updates: $W^{[l]} = W^{[l]} - \alpha dW^{[l]},\ b^{[l]} = b^{[l]} - \alpha db^{[l]}$

}

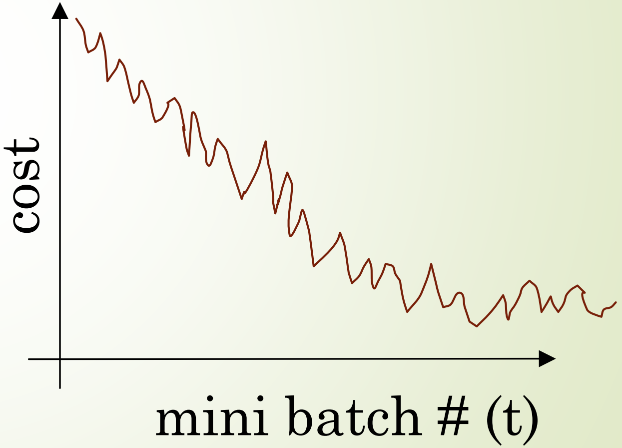
1 epoch = 5000 training inputs
Mini-batch GD much faster than GD!

Training with mini-batch gradient descent

Batch gradient descent



Mini-batch gradient descent



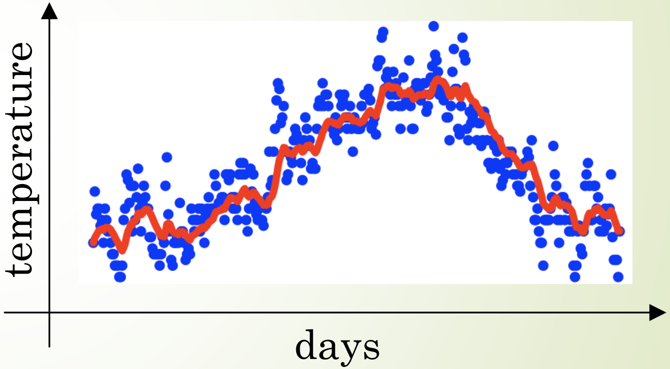
Choosing mini-batch size

Mini-batch size	$(X^{\{1\}}, Y^{\{1\}})$	**Gradient Descent**
m	(X, Y)	Batch
1	$(x^{(1)}, y^{(1)})$	Stochastic
$1 < size < m$		Mini-batch

- Small training set ($m \leq 2000$): Use batch gradient descent
- Typical mini-batch size: 64, 128, 256, 512 (powers of 2)
- Make sure mini-batch fits in CPU/GPU memory

Exponentially weighted averages

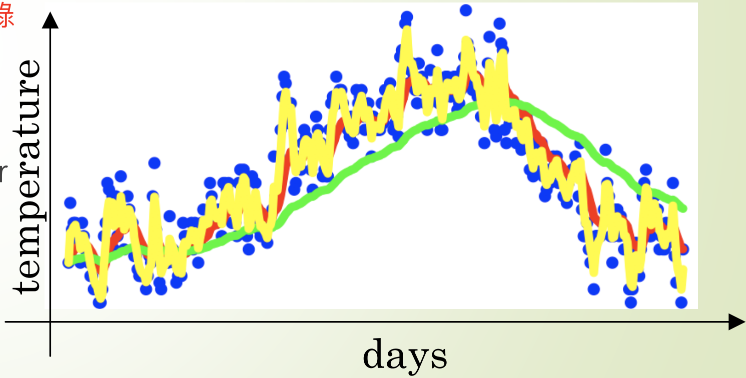
- $v_0 = 0$
- $v_1 = 0.9v_0 + 0.1\theta_1$
- $v_2 = 0.9v_1 + 0.1\theta_2$
- $v_3 = 0.9v_2 + 0.1\theta_3$
- …
- $\boldsymbol{v_t = 0.9v_{t-1} + 0.1\theta_t}$



Exponentially weighted averages

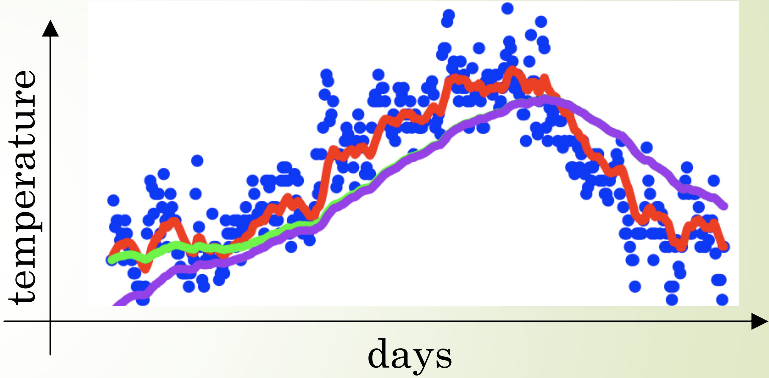
- $v_t = \beta v_{t-1} + (1-\beta)\theta_t$
- $\beta = 0.9: \approx 10\ days' temperature$ 紅
- $\beta = 0.98: \approx 50\ days' temperature$ 綠
- $\beta = 0.5: \approx 2\ days' temperature$ 黃

- v_t is approximately average over $\approx \frac{1}{1-\beta}$ days' temperature



Bias Correction in Exponentially Weighted Average

- $v_t = \beta v_{t-1} + (1-\beta)\theta_t$
- $v_0 = 0$
- $v_1 = 0.98 v_0 + 0.02\theta_1$
- $= 0.02\theta_1$
- $v_2 = 0.98 v_1 + 0.02\theta_2$
- $= 0.0196\theta_1 + 0.02\theta_2$



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

t=2, $1-\beta^t = 1-(0.98)^2 = 00396$

$$\frac{v_t}{0.0396} = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$$

Gradient Descent with Momentum

On iteration t:

Compute dW, db on the current mini-batch

舊值 再給權重　　新值 再給權重

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

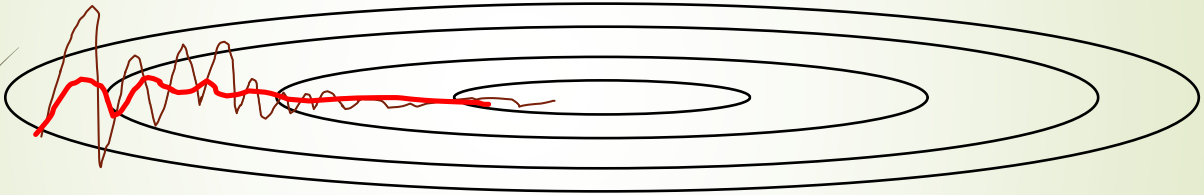
$$v_{db} = \beta v_{db} + (1-\beta)db$$

慢慢改變

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

Hyperparameters: α, β　　　$\beta = 0.9$

Gradient Descent with Momentum



- More smooth in the vertical direction due to weighted averaging
- Faster in the horizontal direction

RMSprop

On iteration t:

Compute dW, db on the current mini-batch

$$\text{Momentum}\left[\begin{aligned} s_{dW} &= \beta_2 s_{dW} + (1-\beta_2)dW^2 \\ s_{db} &= \beta_2 s_{db} + (1-\beta_2)db^2 \end{aligned}\right.$$

$$W = W - \alpha \boxed{\frac{dW}{\sqrt{s_{dW}} + \varepsilon}} \qquad b = b - \alpha \boxed{\frac{db}{\sqrt{s_{db}} + \varepsilon}}$$

$$\varepsilon = 10^{-8}$$

Adam Optimization Algorithm

Momentum+RMSprop

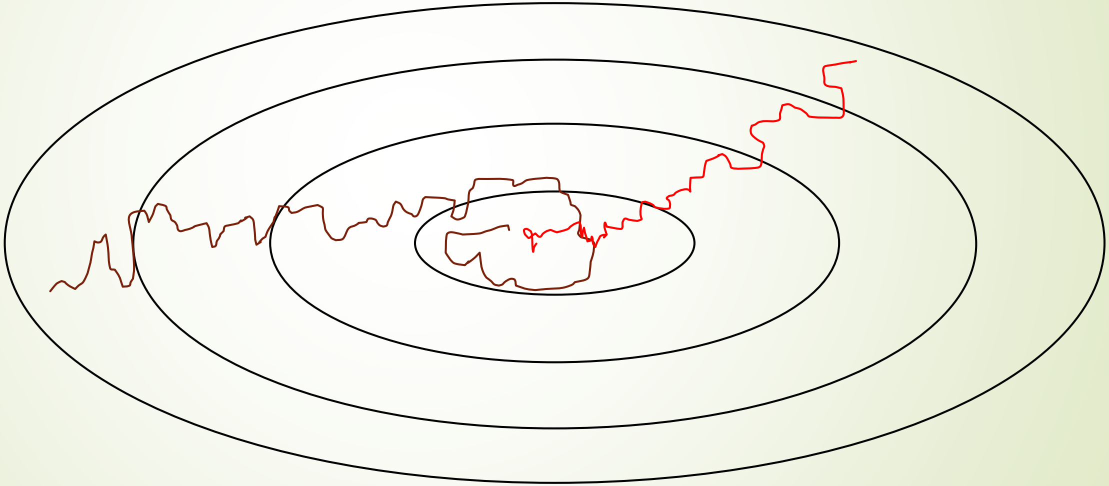
- $v_{dW} = 0, s_{dW} = 0, v_{db} = 0, s_{db} = 0$
- On iteration t:
- Compute dW, db using current mini-batch
- $v_{dW} = \beta_1 v_{dW} + (1-\beta_1)dW, \quad v_{db} = \beta_1 v_{db} + (1-\beta_1)db$ Momentum
- $s_{dW} = \beta_2 s_{dW} + (1-\beta_2)dW^2, \quad s_{db} = \beta_2 s_{db} + (1-\beta_2)db^2$ RMSprop
- 原本數 Bias
- $v_{dW}^{corrected} = v_{dW}/(1-\beta_1^t), \; v_{db}^{corrected} = \frac{v_{db}}{1-\beta_1^t}$
- $s_{dW}^{corrected} = s_{dW}/(1-\beta_2^t), \; s_{db}^{corrected} = \frac{s_{db}}{1-\beta_2^t}$
- $W = W - \alpha \boxed{\frac{v_{dW}^{corr}}{\sqrt{s_{dW}^{corr}+\varepsilon}}}, \; \mathrm{b} = \mathrm{b} - \alpha \boxed{\frac{v_{db}^{corr}}{\sqrt{s_{db}^{corr}+\varepsilon}}}$

Hyperparameters choice

- α: needs to be tuned 前面較快
- β_1: 0.9
- β_2: 0.999
- ε: 10^{-8}

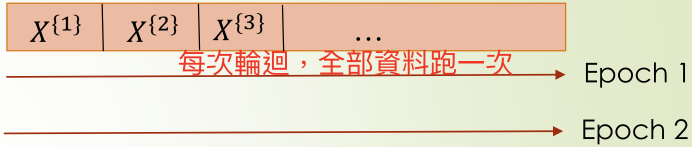
- Adam: Adaptive moment estimation

Learning rate decay

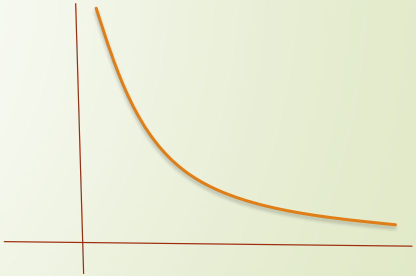


Learning rate decay

- 1 epoch = 1 pass through data
- $\alpha = \frac{1}{1 + decay{-}rate \times epoch{-}num}\alpha_0$

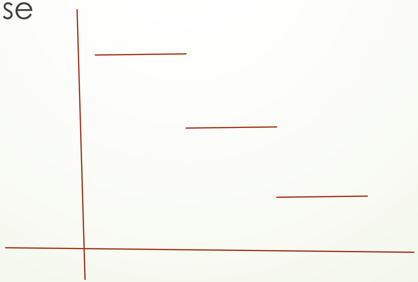


Epoch	α
1	0.10
2	0.67
3	0.50
4	0.40
…	…



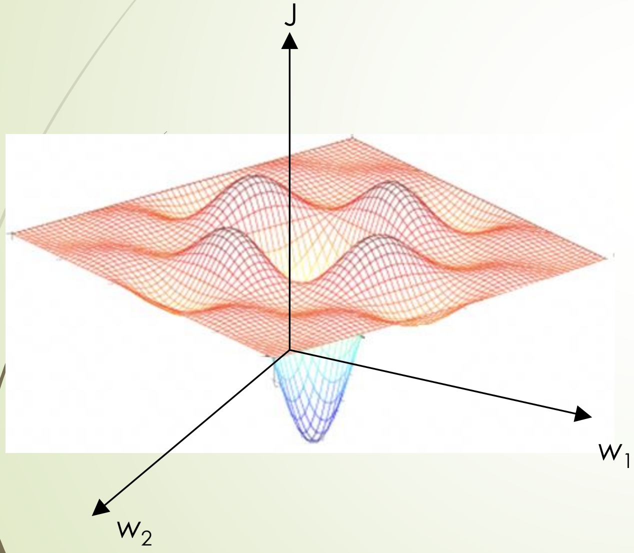
Other learning rate decay methods

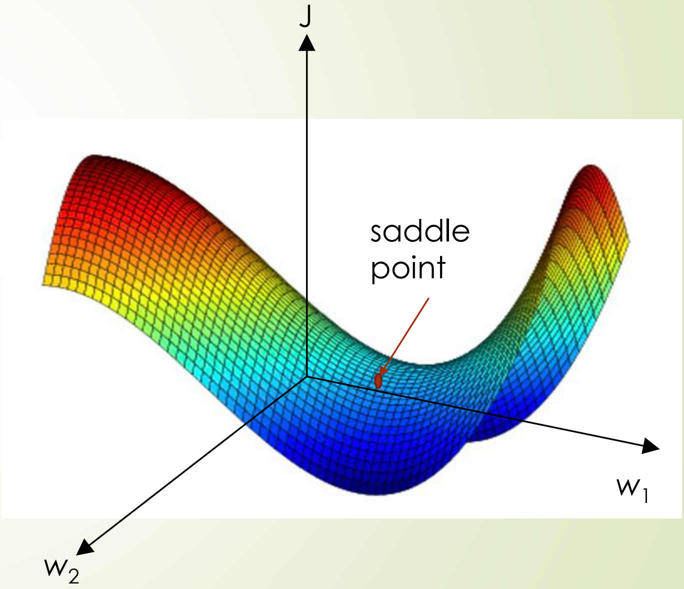
- $\alpha = 0.95^{epoch-num} \times \alpha_0$ (exponential decay)
- $\alpha = \frac{k}{\sqrt{epoch-num}} \times \alpha_0$ (k: constant)
- $\alpha = \frac{k}{\sqrt{t}}$ (t: mini-batch number)
- Discrete staircase



- Manual decay

Problem of Local Optima





Contents

- Regularization and Dropout
- Optimization
- Gradient Checking
- Momentum, RMSprop, Adam, Learning rate decay
- Hyperparameter tuning
- Batch Normalization
- Softmax Regression

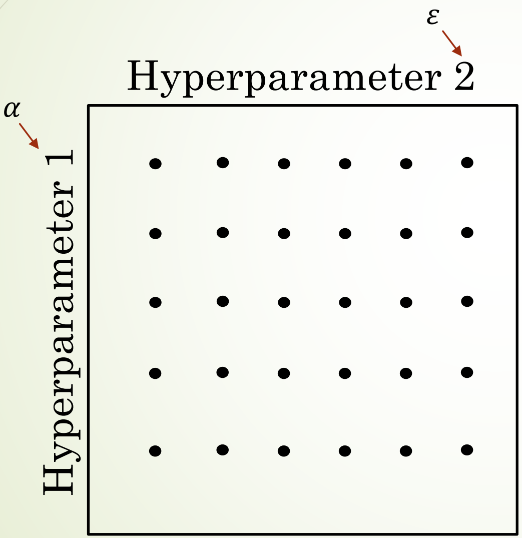
Hyperparameter Tuning

- 1 Learning rate: α
- 2 Momentum: β
  - RMPprop: $\beta_2 = 0.999$ (usually not tuned)
  - Adam: $\beta_1 = 0.9, \beta_2 = 0.999, \varepsilon = 10^{-8}$ (usually not tuned)
- 3 #layers
- 2 #hidden units
- 3 Learning rate decay
- 2 Mini-batch size

↑

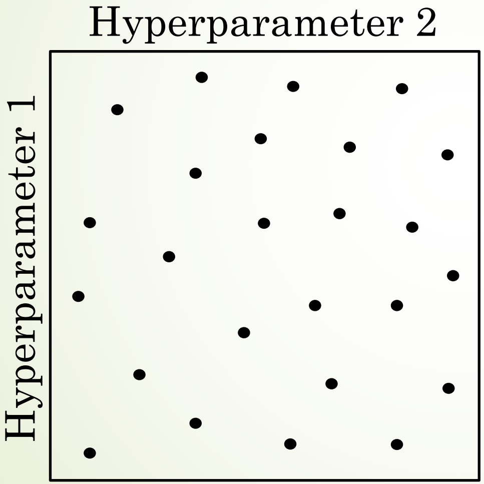
Tuning Order

Try random values: Don't use a grid

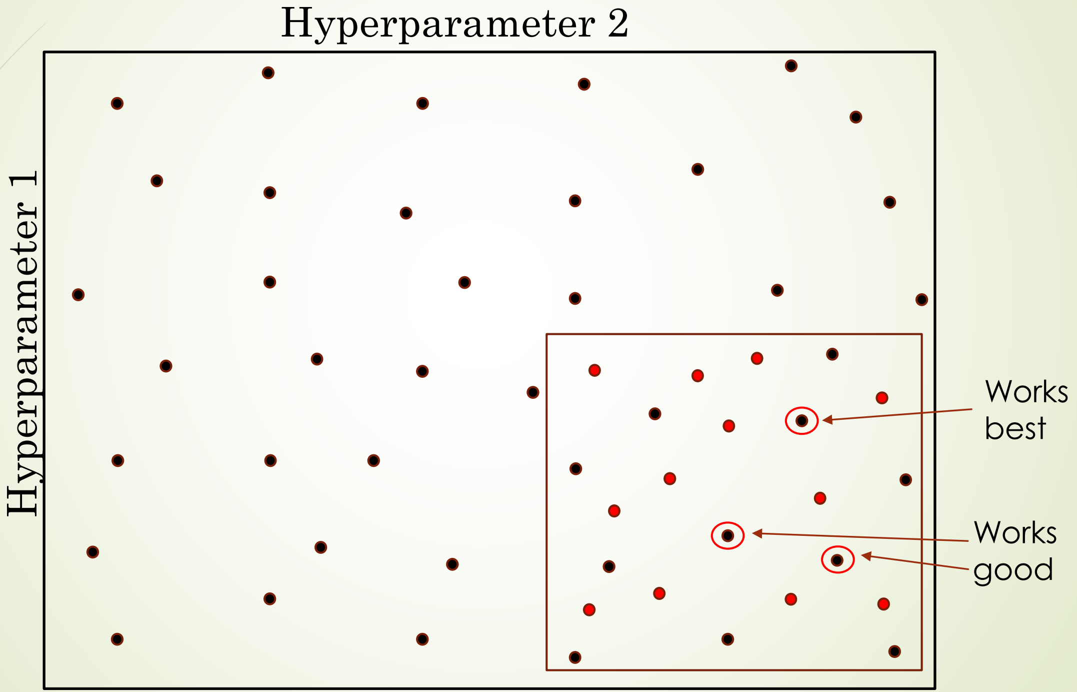


- Some parameters are more important than the others
- Take for example: α, ε
- α is more important than ε
- Grid search would result in searching through only 5 different important values (α)
- Thus, a **random search** would be better!

Try random values: Don't use a grid



Coarse to fine



Using an appropriate scale to pick hyperparameters

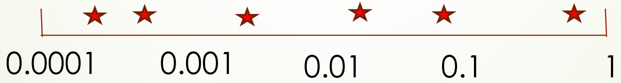
- Picking hyperparameters at random
- $n^{[l]} = 50, \dots, 100$

- Uniformly random sampling would be suffice for some hyperparameters, but not all!

Appropriate scale for hyperparameters

- $\alpha = 0.0001, \dots, 1$

- Random sampling results in 90% samples between 0.1 and 1 and only 10% samples between 0.0001 and 0.1. **This is not good!**
- Instead, take logarithmic scale for more uniform sampling



- $r = -4 \times np.random.randn()$ $\quad r \,\epsilon\, [-4,0]$
- $\alpha = 10^r$ $\quad \alpha \,\epsilon\, [10^{-4}, 10^0]$

$\alpha\epsilon[10^a, 10^b] \;\Rightarrow r \in [a, b]$

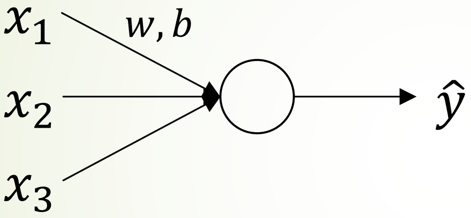
Hyperparameters for exponentially weighted averages

- $\beta = 0.9, \ldots, 0.999 \quad (10\ days\ to\ 1000\ days)$
- $1-\beta = 0.1, \ldots, 0.001 = \ 10^{-1} \ldots 10^{-3}$
- $1-\beta = \ 10^{r} \Longrightarrow \ \beta = 1-10^{r}, \ \Longrightarrow r\epsilon\,[-3,-1]$

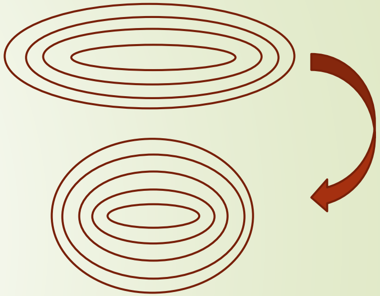
- Why not uniformly over 0.9 to 0.999?
- $\beta: 0.9000 \ \longrightarrow 0.9005 \ \ (approx.\,10\ samples)$
- $\beta: 0.999 \ \longrightarrow 0.9995 \ \ (approx.\,1000\ samples)$
- Thus, need to sample more densely in the range when β is close to 1 (or $1-\beta$ is close to 0).

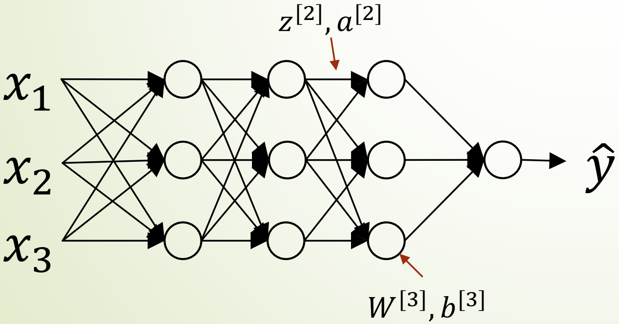
Contents

Normalizing inputs to speed up learning



- $\mu = \frac{1}{m}\sum_i x^{(i)}$
- $X = X - \mu$
- $\sigma^2 = \frac{1}{m}\sum_i x^{(i)2}$
- $X = \frac{X}{\sigma^2}$



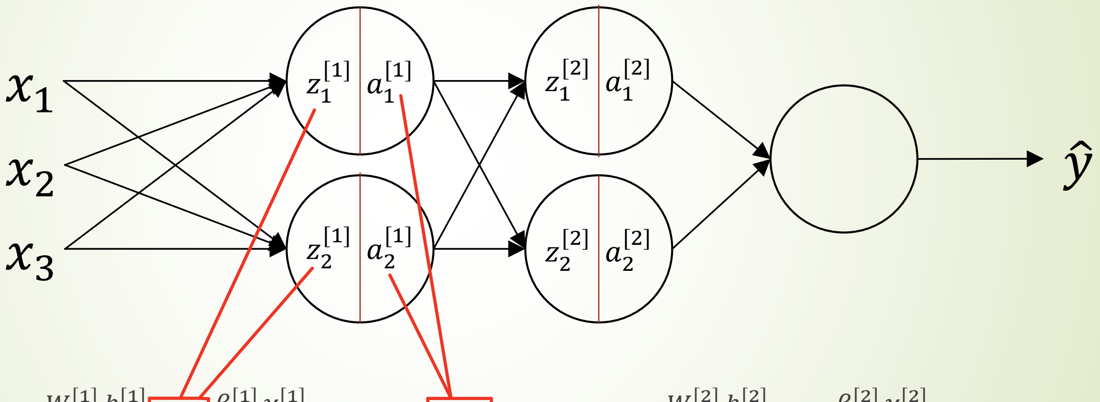


- Can we normalize $a^{[2]}$ so as to train $W^{[3]}, b^{[3]}$ faster?
- Usually, we normalize $Z^{[2]}$

Implementing Batch Norm

- Given some intermediate values in NN $z^{[l](i)}, z^{[l](2)}, \dots, z^{[l](m)}$
- $\mu = \frac{1}{m}\sum_i z^{[l](i)}$
- $\sigma^2 = \frac{1}{m}\sum_i z^{[l](i)2}$
- $z_{norm}^{(i)} = \frac{z^{[l](i)}-\mu}{\sqrt{\sigma^2+\varepsilon}}$
- $\tilde{z}^{[l](i)} = \gamma z_{norm}^{[l](i)} + \beta,$ 優化每一層的loss
  where $\boxed{\gamma, \beta}$ are learnable parameters to control mean and variance of the hidden unit values
- Use $\tilde{z}^{[l](i)}$ instead of $z^{[l](i)}$
- Note that if $\gamma = \sqrt{\sigma^2 + \varepsilon}$ and $\beta = \mu$, then $\tilde{z}^{[l](i)} = z^{[l](i)}$.

Adding Batch Norm to a network



- $X \xrightarrow{W^{[1]},b^{[1]}} \boxed{Z^{[1]}} \xrightarrow{\beta^{[1]},\gamma^{[1]}} \tilde{Z}^{[1]} \longrightarrow \boxed{a^{[1]}} = g^{[1]}(\tilde{Z}^{[1]}) \xrightarrow{W^{[2]},b^{[2]}} Z^{[2]} \xrightarrow{\beta^{[2]},\gamma^{[2]}} \tilde{Z}^{[2]} \longrightarrow a^{[2]}$
- $\beta^{[l]} = \beta^{[l]} - \alpha d\beta^{[l]}$
- Can use a single function `tf.nn.batch_normalization()` to do Batch Norm in Tensorflow.

Working with mini-batches

- $X^{\{1\}} \xrightarrow{W^{[1]},b^{[1]}} Z^{[1]} \xrightarrow{\beta^{[1]},\gamma^{[1]}} \tilde{Z}^{[1]} \longrightarrow a^{[1]} = g^{[1]}(\tilde{Z}^{[1]}) \xrightarrow{W^{[2]},b^{[2]}} Z^{[2]} \xrightarrow{\beta^{[2]},\gamma^{[2]}} \tilde{Z}^{[2]} \longrightarrow a^{[2]}$
- $X^{\{2\}} \xrightarrow{W^{[1]},b^{[1]}} Z^{[1]} \xrightarrow{\beta^{[1]},\gamma^{[1]}} \tilde{Z}^{[1]} \longrightarrow a^{[1]} = g^{[1]}(\tilde{Z}^{[1]}) \xrightarrow{W^{[2]},b^{[2]}} Z^{[2]} \xrightarrow{\beta^{[2]},\gamma^{[2]}} \tilde{Z}^{[2]} \longrightarrow a^{[2]}$
- $X^{\{3\}} \xrightarrow{W^{[1]},b^{[1]}} Z^{[1]} \xrightarrow{\beta^{[1]},\gamma^{[1]}} \tilde{Z}^{[1]} \longrightarrow a^{[1]} = g^{[1]}(\tilde{Z}^{[1]}) \xrightarrow{W^{[2]},b^{[2]}} Z^{[2]} \xrightarrow{\beta^{[2]},\gamma^{[2]}} \tilde{Z}^{[2]} \longrightarrow a^{[2]}$
- … for all mini-batches

- Parameters: $W^{[l]}, b^{[l]}, \beta^{[l]}, \gamma^{[l]}$
- $Z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$, since we subtract the mean for batch norm, the parameter $b^{[l]}$ is not required.
- $\tilde{Z}^{[l]} = \gamma^{[l]}Z_{norm}^{[l]} + \beta^{[l]}$, so $\beta^{[l]}$ will now act as the bias.

Implementing Gradient Descent

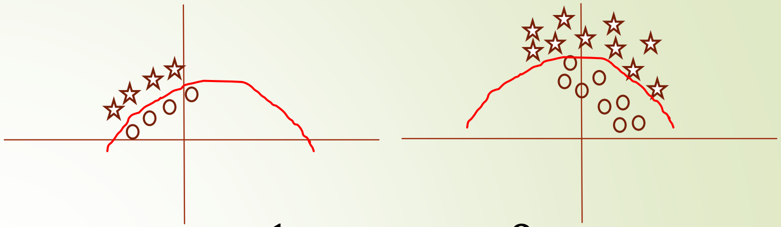
- for $t = 1 \; \ldots numMiniBatches$
    - Compute forward propagation on $X^{\{t\}}$
        - In each hidden layer, use BN to replace $Z^{[l]}$ with $\tilde{Z}^{[l]}$.
    - Use back propagation to compute $dW^{[l]}, \cancel{db^{[l]}}, d\beta^{[l]}, d\gamma^{[l]}$
    - Update parameters: $W^{[l]} := W^{[l]} - \alpha dW^{[l]},\; \beta^{[l]} := \beta^{[l]} - \alpha d\beta^{[l]},\; \gamma^{[l]} := \gamma^{[l]} - \alpha d\gamma^{[l]}$
        - (Can also use GD with **Momentum, RMSprop, or Adam**!)

Why does Batch Norm work?

- Feature normalization can speed up learning
  - Thus, normalizing hidden values can also speed up learning

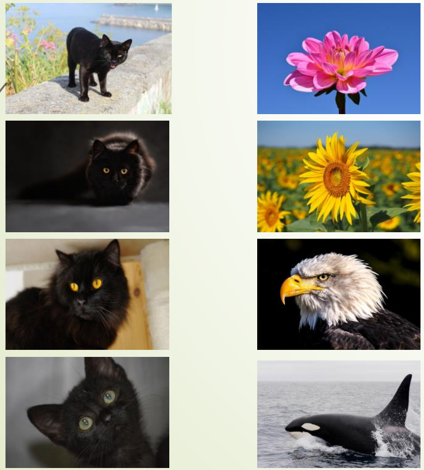
Covariate Shift

改變平均數/學到新的東西



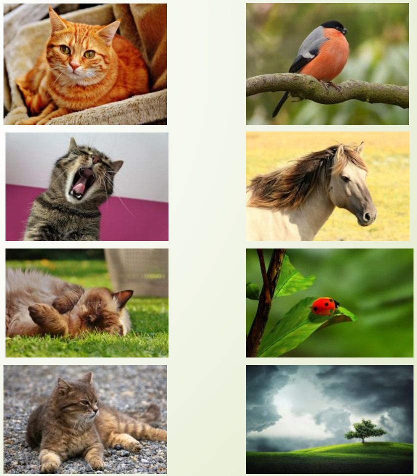
Cat $y = 1$

Non-Cat $y = 0$

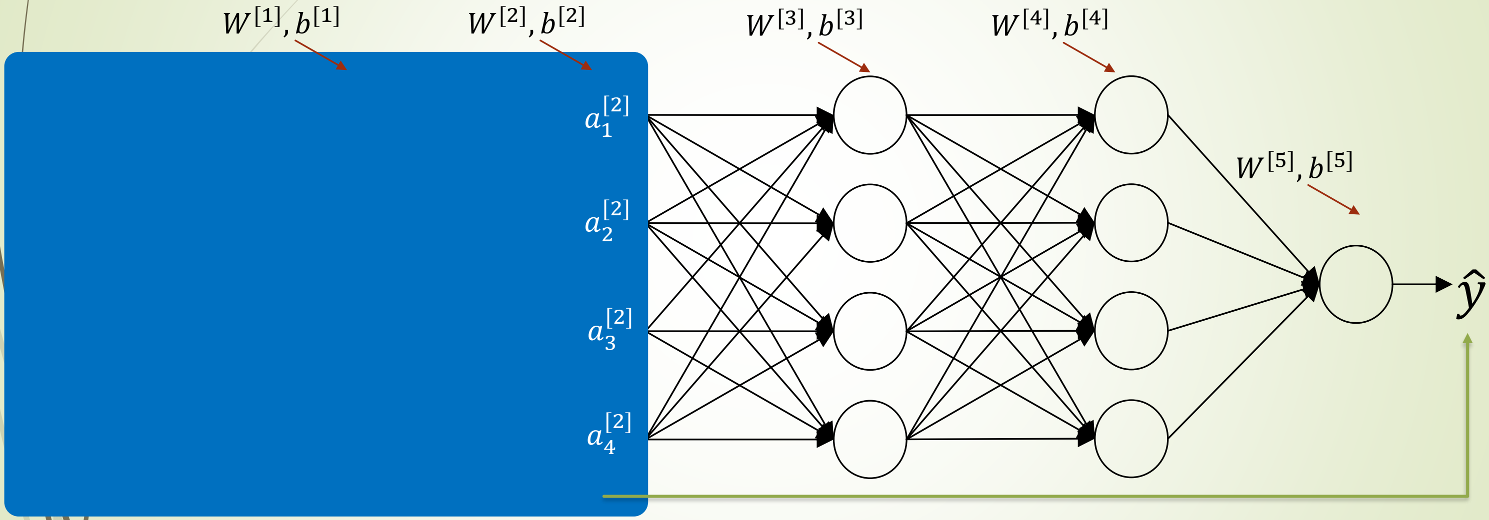


$y = 1$

$y = 0$



Why this is a problem with NN?



- Batch Normalization de-couples consecutive hidden layers by scaling the hidden values to have constant mean and variance.

Batch Norm at test time

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

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

$$z_{\text{norm}}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

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

- During training, μ, σ^2 are computed over each mini-batch
- During testing, μ, σ^2 are estimated using exponentially weighted average (across mini-batch)
  - $\mu^{\{1\}[l]}, \mu^{\{2\}[l]}, \mu^{\{3\}[l]}, \ldots$ take weighted average
  - $\sigma^{2\{1\}[l]}, \sigma^{2\{2\}[l]}, \sigma^{2\{3\}[l]}, \ldots$ take weighted average

Contents

- Regularization and Dropout
- Optimization
- Gradient Checking
- Momentum, RMSprop, Adam, Learning rate decay
- Hyperparameter tuning
- Batch Normalization
- Softmax Regression

Recognizing cats, dogs, and baby chicks

3

1

2

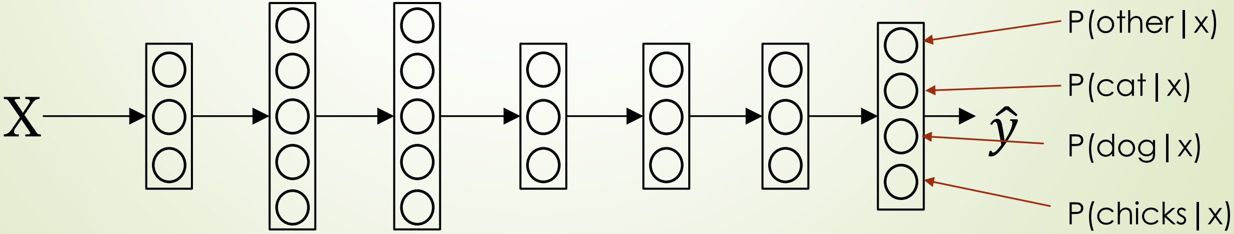
0

3

2

0

1



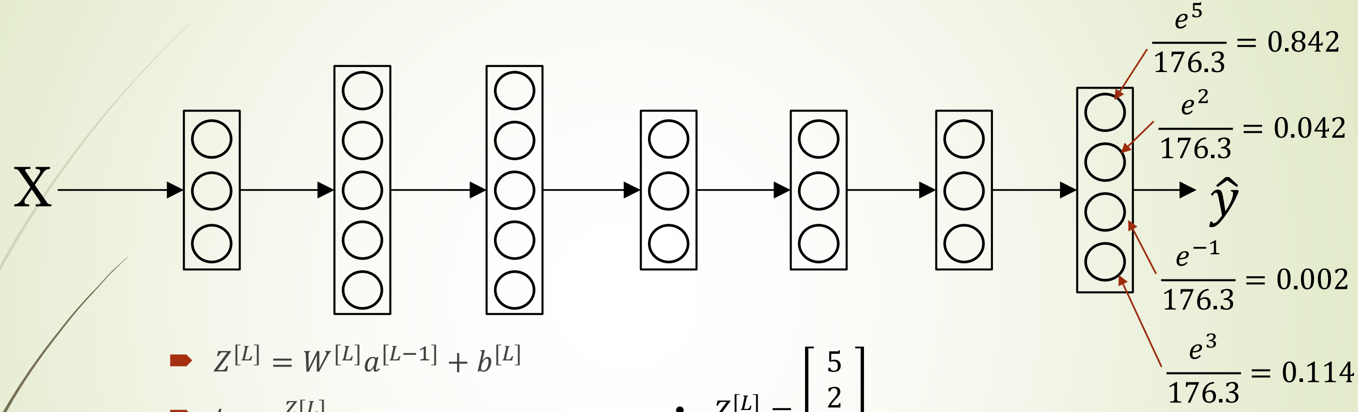
P(other | x)

P(cat | x)

P(dog | x)

P(chicks | x)

Softmax Layer



$\frac{e^5}{176.3} = 0.842$

$\frac{e^2}{176.3} = 0.042$

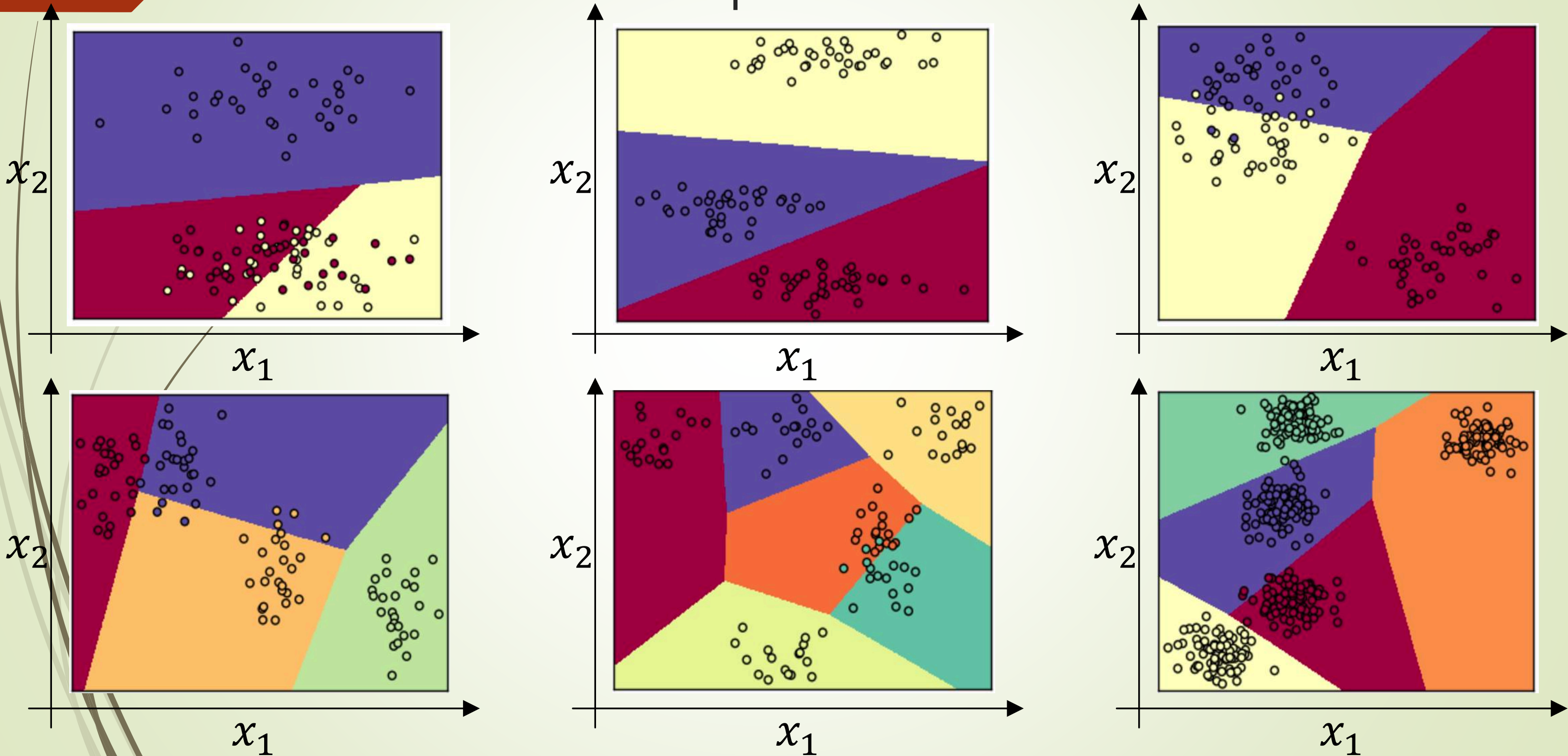
$\frac{e^{-1}}{176.3} = 0.002$

$\frac{e^3}{176.3} = 0.114$

- $Z^{[L]} = W^{[L]}a^{[L-1]} + b^{[L]}$
- $t = e^{Z[L]}$
- $a^{[L]} = \frac{e^{Z^{[L]}}}{\sum_{i=1}^{n_L} t_i}$

- $Z^{[L]} = \begin{bmatrix}5\\2\\-1\\3\end{bmatrix}$
- $t = \begin{bmatrix}e^5\\e^2\\e^{-1}\\e^3\end{bmatrix} = \begin{bmatrix}148.4\\7.4\\0.4\\20.1\end{bmatrix}$ $\sum_{j=1}^{4} t_j = 176.3$
- $a^{[L]} = \frac{t}{176.3}$

Softmax examples



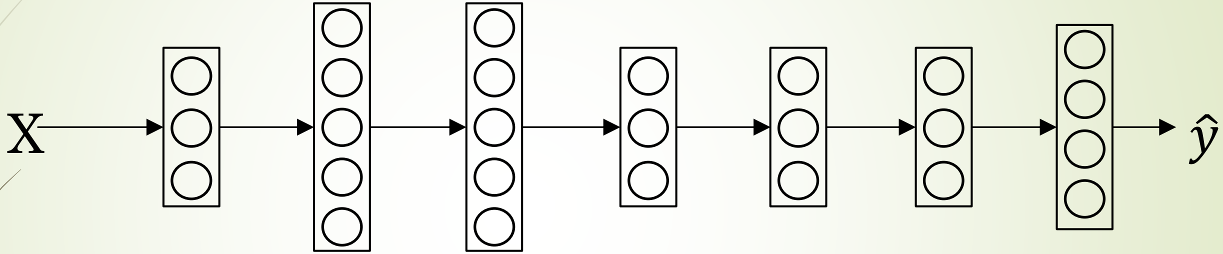
Understanding softmax

- $Z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$ $\qquad t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix}$
- $a^{[L]} = g^{[L]}\left(z^{[L]}\right) = \begin{bmatrix} e^5/(e^5 + e^2 + e^{-1} + e^3) \\ e^2/(e^5 + e^2 + e^{-1} + e^3) \\ e^{-1}/(e^5 + e^2 + e^{-1} + e^3) \\ e^3/(e^5 + e^2 + e^{-1} + e^3) \end{bmatrix} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$
- Softmax because each class has a probability, whereas hardmax would give 1 to the class with highest probability and 0 to the rest $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$.
- If C = 2, softmax reduces to logistic regression.

Loss function for training NN with softmax

- Loss function: $\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{n_l} y_j \log \hat{y}_j$
- Suppose $y^{(i)} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$, $\mathcal{L}(\hat{y}, y) = \underline{-y_2 \log \hat{y}_2 = -\log \hat{y}_2}$, thus to make loss $\mathcal{L}$ small, need to make probability $\hat{y}_2$ big!
- Cost function: $\mathcal{J}\left(W^{[1]}, b^{[1]}, \dots\right) = \boxed{\frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})}$

Gradient Descent with Softmax



- Backprop: $dZ^{[L]} = \hat{y} - y$
- In the programming frameworks, backprop is done automatically, no need to implement.

References

- Coursera online courses