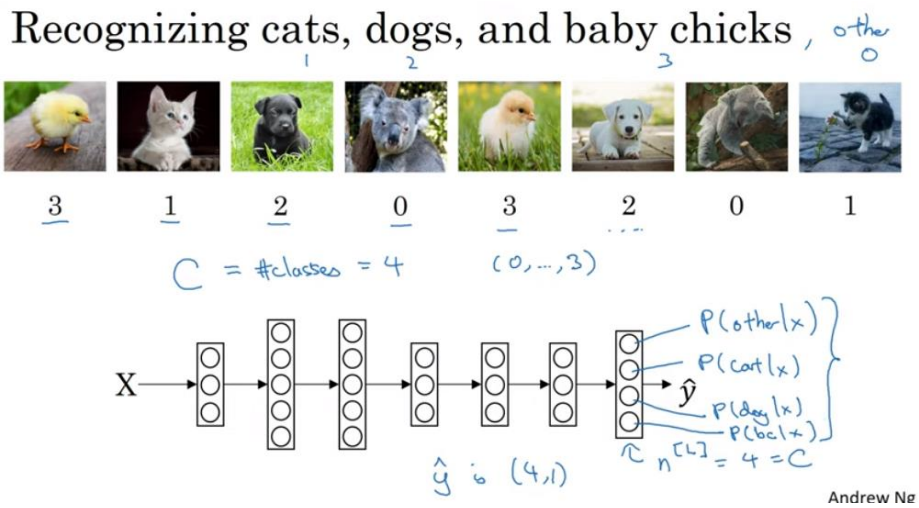
7. 다중 클래스 분류

<Softmax Regression>



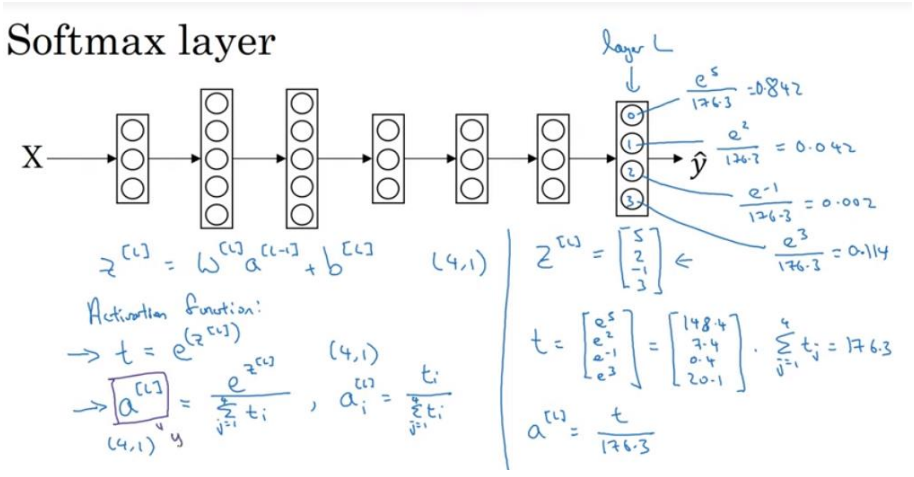
ex) 클래스가 1,2,3,0 일 때

- C : 입력값을 분류하는 클래스의 숫자 (여기서는 4개) / 0~C-1
- 출력층 L의 단위개수 : n = 4 = C

⇨ 각 클래스의 확률을 알고 싶은 것 (X가 주어졌을 때 기타 클래스가 나올 확률)

출력값 yhat은 (4,1) 벡터

4개의 확률값의 합= 1

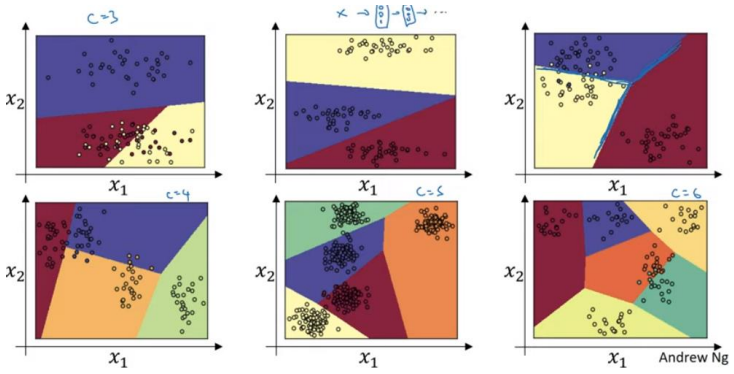


<u>소프트 맥스 층을 사용</u>

- 신경망의 최종층에서 z^[L] 구하려함.
- 소프트맥스 층의 활성 함수
- t=e^(z^([L]) 임시변수 사용
- t도 (4,1)벡터, a{L} 도 (4,1)벡터

$$a_i = \frac{e_{z_i}}{\sum_{j=1}^{C} e^{z_j}}$$

- a[L]=g[l](z[l])
- 소프트맥스 활성화 함수는 정규화를 하기 위해 입력 출력값이 다 벡터임



ex) c=3인 클래스

- 학습세트를 가져와서 세개의 선택지에 따라 분류하는 소프트맥스 학습
- 색깔은 출력값 나타냄
- 클래서가 2보다 큰 선형적 기준을 가진 로지스틱회귀의 일반적 기준
- 두 클래스 사이의 경계가 선형

Softmax 분류기 훈련시키기

ex) C=4, 임시변수: t

t를 합이 1이되도록 정규화

- 소프트맥스 : 하드맥스랑 반대되는 뜻

  (하드맥스 : z벡터를 받아와서 큰값에 1 넣고 나머지는 0으로 둠)

  소프트맥스는 부드러운 느낌으로 z를 확률로 대응

⇨ 두클래스만 다루는 로지스틱 회귀를 일반화함

c=2이면 로지스틱 회귀와 같아진다

하나만 계산해도 되므로 이는 로지스틱 회귀가 결괏값 계산하는 것과 같음

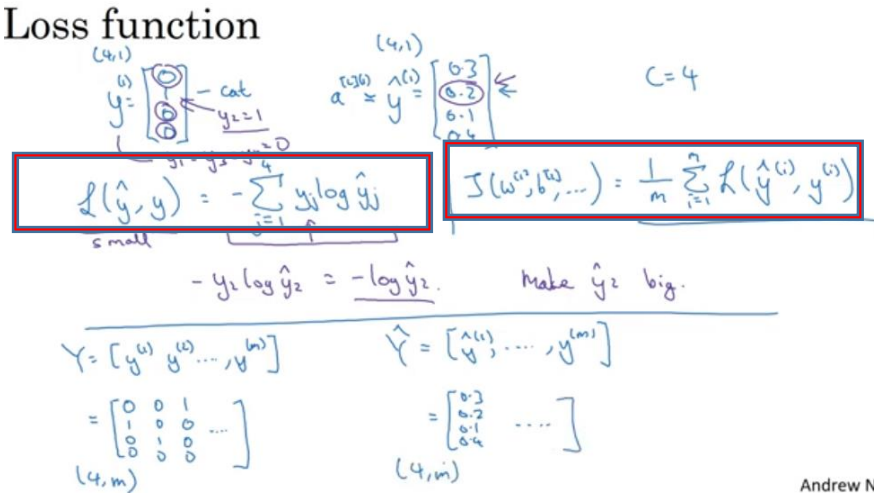
<손실함수>

y=[0,1,0,0] => cat

yhat=[0.3,0.2,0.1,0.4] 고양이일 확률은 20%

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

- 손실함수 :
- 손실함수의 값을 작게 만드려고 함 =>log(yhat2)를 크게 해야함
- 클래스가 뭐든 클래스에 대응하는 확률을 크게 만들어야함
- (최대우도 추정과 비슷)



전체 훈련세트에 대한 비용함수

J= sum(loss function) / m

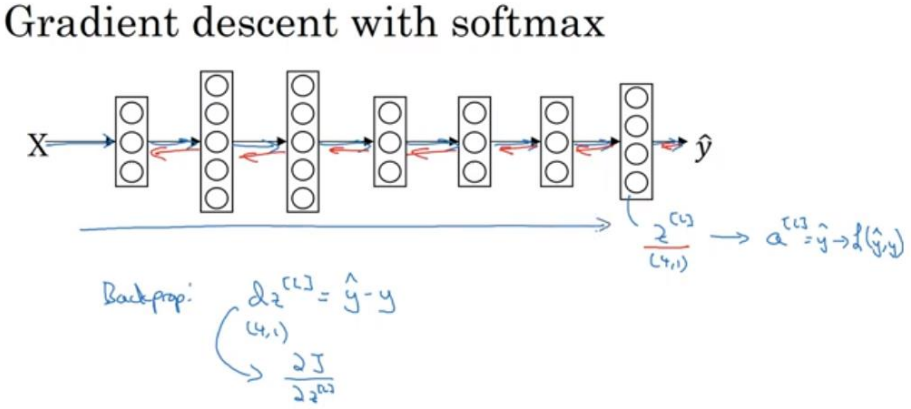
- 경사하강법 써서 비용함수를 최소로 해야함

Y=[y(1),y(2),..y(m)] => (4,m) 행렬

Yhat=[yhat(1),...yhat(m)] => (4,m) 행렬

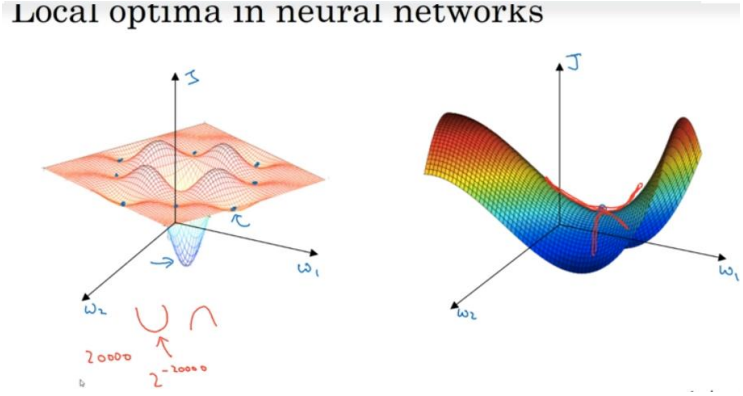
z[L] => a[L] or yhat 구하고, loss 구함

- 역방향 전파나 경사하강법은 **dz[L]=yhat-y**



8. 프로그래밍 프레임워크 소개

<지역 최적값 문제>



- 지역최적값이 많아보임
- 경사가 0인 점은 대부분 지역 최적값이 아니라, 비용함수의 경사가 0인경우가 **안장점**
- 낮은 차원의 공간에서 얻은 직관이 높은 차원에서는 적용되지 않을 수 있음

- 안정지대에서 미분값이 오랫동안 0이므로 학습을 오래 지연시킴

충분히 큰 신경망을 학습시킨다면 지역 최적값에 갇힐 일이 잘 없음

안정지대에서 학습 속도가 느려지므로 문제 => 모멘텀, RMSprop, Adam 등의 알고리즘의 도움

Adam과 같은 최적화 알고리즘

<Tensorflow>

J(w)=w^2-10w+25=(w-5)^2

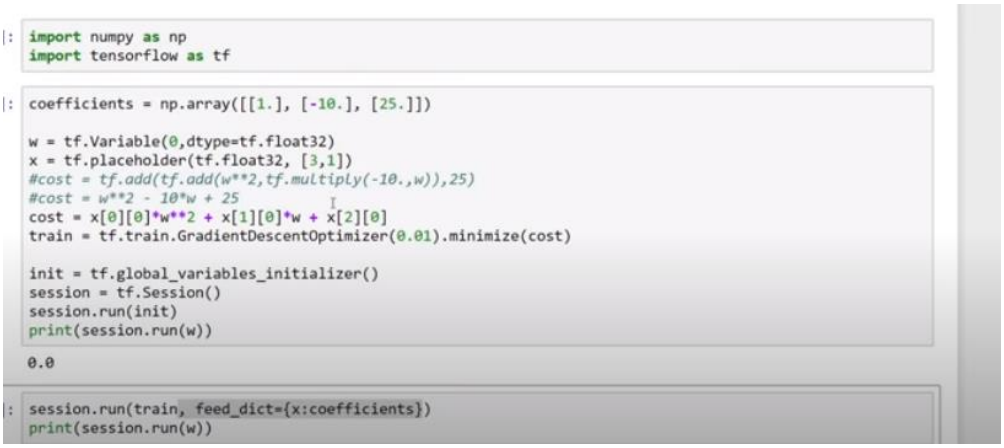
w=5 => J minimize

```
import numpy as np
import tensorflow as tf

w=tf.Variable(0,dtype=tf.float32)

#cost= tf.add(tf.add(w**2,multiply(-10.,w)),25)
cost= w**2-10*w+25

train=tf.train.GradientDescentOptimizer(0,01).minimize(cost)
init=tf.global_variables_initializer()

session=tf.Session() #=> 텐서플로우 세션 시작
session.run(init) #=> 전역변수 초기화
sess.run(w) #=> 변수dml 값을 알 수 있도록
session.run(train) #=> 경사하강법을 한단계 시행
print(session.run(w)) #=> w가 0.1

for I in range(1000):
session.run(train)
print(session.run(w)) #=> 4.99
```

1. w는 최적화하고 싶은 변수로 정의

2. 비용함수 정의

3. 텐서플로우는 자동으로 미분 계산 (정방향 전파만 구현해도 되는 이유)



```
import numpy as np
import tensorflow as tf

coefficients = np.array([[1.], [-10.], [25.]])

w = tf.Variable(0,dtype=tf.float32)
x = tf.placeholder(tf.float32, [3,1])
#cost = tf.add(tf.add(w**2,tf.multiply(-10.,w)),25)
#cost = w**2 - 10*w + 25
cost = x[0][0]*w**2 + x[1][0]*w + x[2][0]
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

init = tf.global_variables_initializer()
session = tf.Session()
session.run(init)
print(session.run(w))
```

```
0.0
```

```
session.run(train, feed_dict={x:coefficients})
print(session.run(w))
```

#x를 플레이스 홀더로 정의하면

x=tf.placeholder(tf.float32,[3,1])

cost=x[0][0]*w^2+x[1][0]*w+x[2][0]

x=>이차함수의 계수 조정

session.run(train,feed_dict={x:coefficients})

- 플레이스홀더 : 값을 나중에 넣는 변수
- 학습 데이터를 비용함수에서 쉽게 얻을 수 있음
- 각 학습에서 미니배치 경사하강법을 쓴다면 서로다른 미니배치를 넣어야함

```
import numpy as np
import tensorflow as tf

coefficients = np.array([[1], [-20], [25]])

w = tf.Variable([0],dtype=tf.float32)
x = tf.placeholder(tf.float32, [3,1])
cost = x[0][0]*w**2 + x[1][0]*w + x[2][0]     # (w-5)**2
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
init = tf.global_variables_initializer()
session = tf.Session()
session.run(init)
print(session.run(w))

for i in range(1000):
    session.run(train, feed_dict={x:coefficients})
print(session.run(w))
```

```
with tf.Session() as session:
    session.run(init)
    print(session.run(w))
```

- 비용함수를 명시하면 미분을 계산하고 비용을 최소화하는 것을 찾음(역방향 함수는 이미 구현됨)

⇨ 계산 그래프를 그리도록 하는 것!