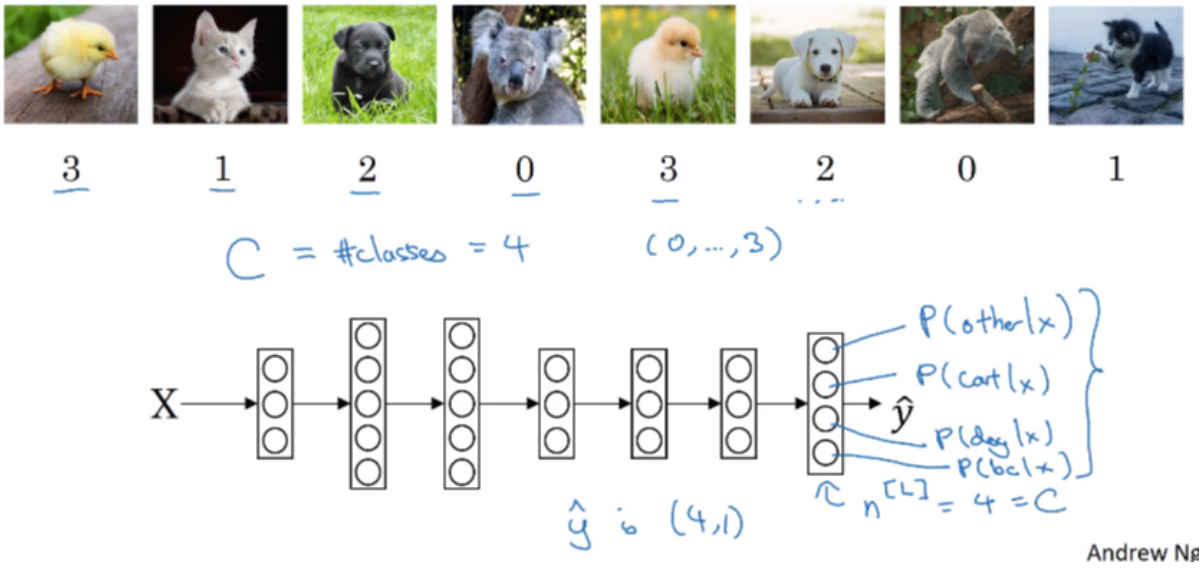
7. 다중 클래스 분류

1. Softmax Regression

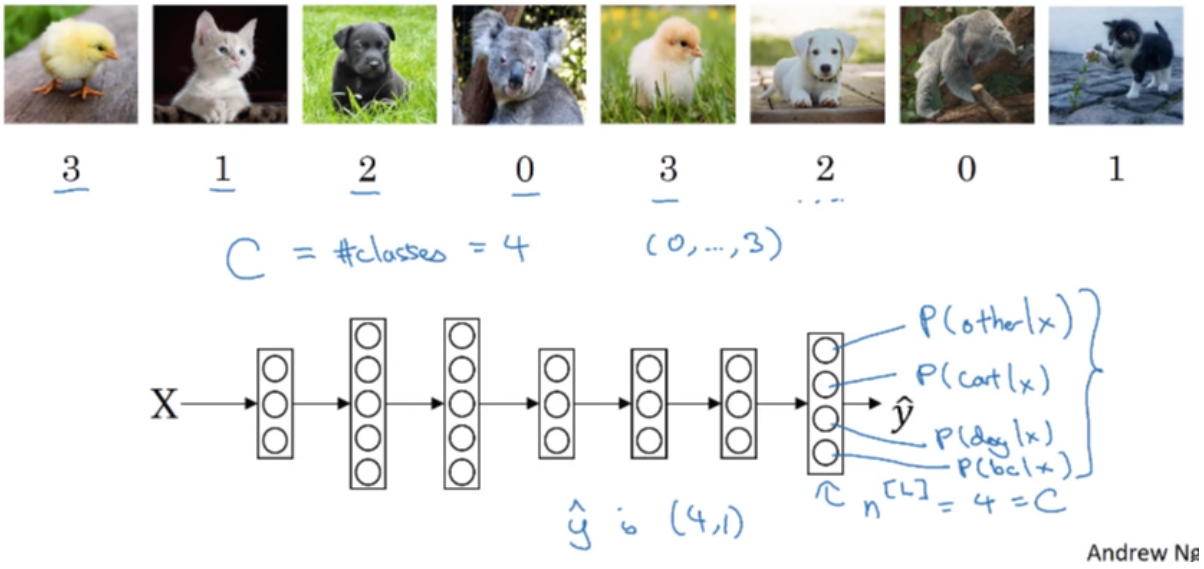
: 로지스틱 회귀를 일반화한 회귀

1. **Recognizing cats, dogs ,and baby chicks other**

- 고양이: 클래스 1
- 개: 클래스 2
- 병아리: 클래스 3
- 고양이, 개, 병아리 해당하지 않는 동물: 클래스 0



- 대문자 C: 클래스의 숫자를 나타내는데 사용 (입력값을 분류하는데 사용)
- 클래스에 숫자를 붙인다면 0에서 C-1까지 부여됨.



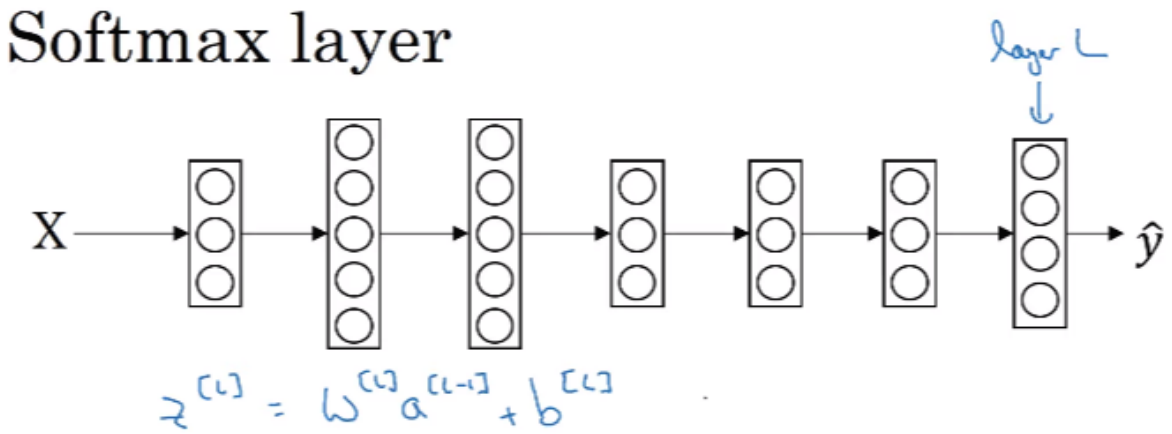
위와 같은 신경망을 하나 만들었다고 가정.

- 출력층에는 C개(이 경우 4개의 출력 단위가 있는 신경망),
  - 출력층 L의 단위 개수인 n은 4 (또는 일반적으로 C가 될 것)
- 첫 번째 단위에서 우리가 원하는 출력값: 입력값 X가 주어졌을 때 기타 클래스
- 두 번째 단위에서는 고양이의 확률
- 세 번째 단위에서는 개의 확률
- 네 번째 단위에서는 병아리의 확률이다(병아리는 BC로 줄여 필기).

→ **y^은** 출력값으로 **네 개의 확률값이 주어지므로 (4,1)차원의 벡터**가 될 것이다.

- **y^의 각 값들의 합은 1**이 되어야 한다.
- 이런 신경망을 얻기 위한 가장 표준적인 모델: **소프트맥스층을 사용하는 것**이다.

2. Softmax layer



- 신경망의 **최종층(L):** 평소처럼 층의 선형적인 부분인 **z^[L]을 계산 (**최종층인 L의 z값)
- z^[L]을 w^[L]과 이전 층의 활성화 함수(a^[L-1])를 곱한 뒤 그 최종층의 평향(b^[L])을 더하여 계산. => **z^[L] = w^[L]*a^[L-1]+b^[L]**
- z값을 계산하기 위해서 **소프트맥스 활성화 함수** 사용
  - 각 원소에 대해 계산할 때, **t=e^(z^[L])라는 임시 변수를 사용**

  (z^[L]이 (4,1) 차원의 벡터인데, t는 z^[L]의 각 원소에 e를 취한 것이므로 결과도 (4,1)이 된다.)
  - 출력값인 a^[L]은 벡터 t와 동일
  - **합이 1이 되도록 모든 임시 변수값들의 합을 나눠서 정규화해야 함.**

  (즉 a^[L]은 z^[L]을 j가 1부터 4까지(원소가 4개이므로) t_j를 모두 더한 값으로 나눈다. 그러면 a^[L]은 (4,1)벡터이고 이 벡터의 i번째 원소(a^[L]_i)는 t_j를 t_i 값들의 합으로 나눈 것과 같다)

  - ex) (4,1) 벡터인 z^[L]이 [5, -2, 1, 3]라고 가정.

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

$$t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148 \cdot 4 \\ 7 \cdot 4 \\ 0 \cdot 4 \\ 20.1 \end{bmatrix} \cdot \sum_{j=1}^{4} t_j = 176.3$$

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

1. 원소별로 e를 취해 t를 구하기.

→ t는 e의 5승, e의 2승, e의 -1승, e의 3승 (계산하면 각각 148.4, 7.4, 0.4, 20.1)

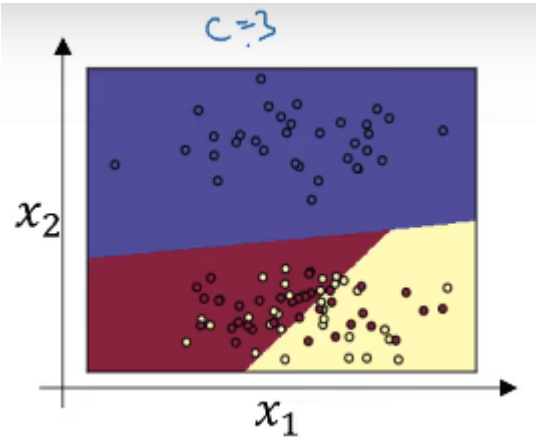
2. t에서 a^[L]으로 합이 1이 되도록 정규화시키기 위해 t의 4개의 합을 구하면 176.3.
3. a^[L]은 t를 176.3으로 나눈 것. 즉 예시에서 첫번째 노드의 값은 e^5을 176.3으로 나눈 0.842이다. 이런 z값을 얻었다면 클래스 0이 될 확률이 84.2%인 것이다.
4. 다음 노드의 출력값은 e^2을 176.3으로 나눈 값인 0.042이다. 다음은 e^(-1)을 176.3으로 나눠서 0.02이다. e^3을 176.3으로 나눠서 0.114값을 얻는다. 이는 11.4%의 확률로 클래스3(병아리)이 되는 것이다.
5. 이런 식으로 클래스 0.1.2.3이 될 확률을 구할 수 있다.
6. 이 신경망의 출력값 y^과도 같은 a^[L]은 (4,1)의 벡터가 되고, 그 안에는 계산한 숫자들(0.842, 0.042, 0.002, 0.114)이 들어가 있을 것. 이 알고리즘은 z^[L]이라는 벡터를 취해서 합이 1이 되는 4개의 확률 값을 내 놓는다.

→ z^[L]에서 a^[L]으로 되는 과정을 요약하자면 e를 취해서 임시 변수 t를 얻고 정규화한 이 과정을 소프트맥스 활성화 함수로 요약할 수 있다. 즉 **a^[L]은 z^[L] 벡터에 활성함수 g^[L]을 적용한 것**이다. 이 **활성화 함수 g의 특이한 점은 (4,1)벡터를 받아서 (4,1)벡터를 내놓는다는 것**이다. 이전에는 활성함수가 하나의 실수값을 받았다. (ex. 시그모이드나 Relu 활성화 함수 등은 실수를 받아서 실수를 출력)

→ **소프트맥스 활성화 함수의 특이한 점: 정규화를 하기위해서 입력값과 출력값이 모두 벡터라는 것**

3. **Softmax examples**

- 소프트맥스 분류로 할 수 있는 것을 보이기 위해 x_1과 x_2의 입력값이 있다고 가정. 이 값들은 바로 소프트맥스 층에 들어간다. 안에는 서너개의 노드가 있고, 출력값은 y^이다. 이
- z^[1]을 w^[1]과 x를 곱한 뒤 b^[1]을 더하여 계산하고 출력값 y^이자 a^[1]을 z^[1]에 소프트맥스 활성화함수를 적용시켜 얻는다.
- 예시1



1) 입력값 x_1과 x_2에 대하여 이 결정 기준을 나타내는 c=3의 클래스를 가진 소프트맥스 층을 사용

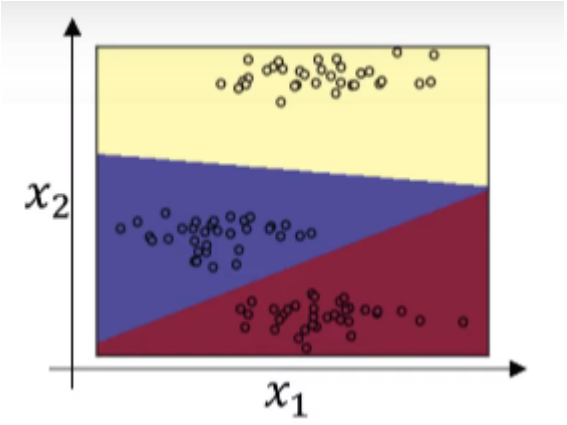
2) **선형적인 기준**에 따라 데이터가 3개의 클래스로 구분.

3) 학습 세트를 가져와서 비용 함수와 세 개의 선택지에 따라 분류하는 소프트맥스 함수를 학습시키는 것.

4) 색깔은 **소프트맥스 분류 함수에 따라 출력값을 나타낸 것**이고, **입력값은 가장 높은 확률의 출력값**에 따라 색을 입힌 것.

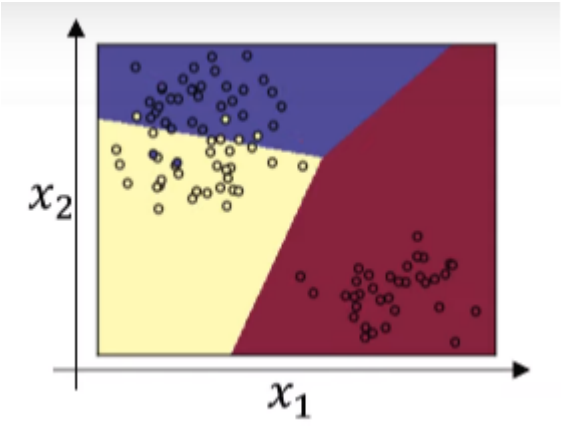
5) 선형기준을 가지고 있는 로지스틱 회귀의 일반적인 형태. 하지만 클래스는 0,1 또는 0,1,2가 될 수 있다.

- 예시2



→ 소프트맥스 분류 함수가 나타낸 또 다른 경우. 3개의 클래스에 따라 데이터를 학습시킴.

- 예시3

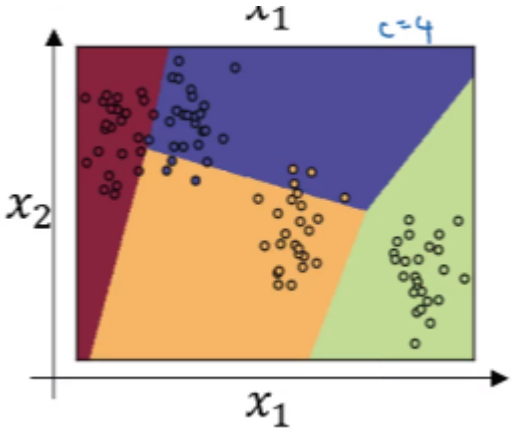


1) 또 다른 분류.

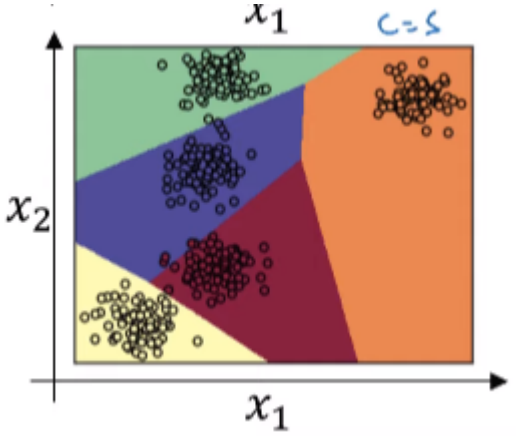
2) 얻을 수 있는 직관: 두 클래스 사이의 경계가 선형이라는 것.

3) 그래프를 보면 노란색과 빨간색 사이에도 선형 경계가 그려져 있고 보라색과 빨간색, 빨간색과 노란색 사이에도 선형결정 경계가 그려져 있다. 하지만 다른 선형 함수를 사용해서 공간을 세 개의 클래스로 나눌 수도 있다.

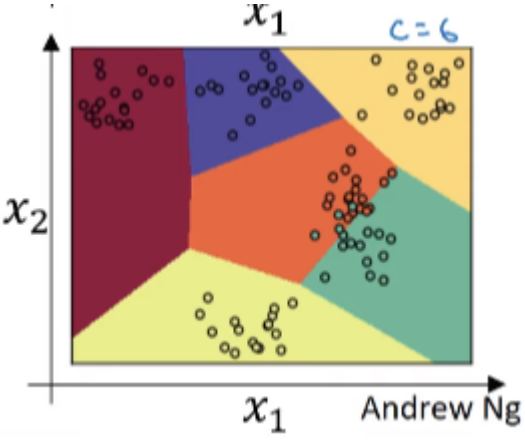
- C = 4인 경우



- C=5인 경우



- C = 6인 경우



→ 은닉 유닛이 여러개인, 더 깊은 신경망을 다룬다면 여러 클래스를 분류하기 위해 더 복잡하고 비선형의 경계도 볼 수 있다.

2. Softmax 분류기 훈련시키기

1. **Understanding softmax**

$$z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix} \quad t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix}$$

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

- C=4개의 클래스 → z^[L]은 (4,1)벡터.
- 원소 단위로 e를 취해 임시 변수 t를 얻음.
- **활성화 함수인 g^[L]은 소프트맥스 활성화함수**. 이 함수는 t를 합이 1이 되도록 정규화 시켰고 이는 a^[L]과 같다. 여기서 z^[L]의 가장 큰 원소가 5였고 가장 큰 확률도 첫번째 확률이다.
- 소프트맥스: 하드맥스와 반대.
    - **하드맥스는** z벡터를 받아와서 z의 원소를 살펴보고 가장 큰 값이 있는 곳에 1을 나머지에는 0을 갖는 벡터로 대응시킨다.

$$\text{“half max”} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

- **소프트맥스 회귀나 활성화 함수가 두 클래스만 다루는 로지스틱 회귀를 일반화**했다는 점이 중요

- 만약 **소프트맥스에서 C=2라면 결국 로지스틱 회귀와 같아진다.**

→ C=2에서 소프트맥스를 적용했을 때, 출력층 a^[L]은 C=2에서 출력값 두 개를 모아둔 것이다. 0.842와 0.158이라고 하면 이 두 숫자의 합은 항상 1이 된다. (합이 항상 1이므로 둘 다 계산하면 번거로우므로 하나만 계산해도 된다.) 그러면 그 숫자를 계산하는 방식이 로지스틱 회귀가 하나의 출력값을 계산하는 방식과 같다.

2. **Loss function**

- ex) 한 샘플이 목표로 하는 출력값이 관측을 기반으로 0,1,0,0이라고 하자.
  - 이 벡터는 클래스가 1이므로 고양이를 의미.
  - 신경망의 출력값 y^ = a^[L] (y^은 합이 1인 확률로 구성된 벡터) 이 샘플에 대해서는 고양이일 확률이 20%에 불과하므로 신경망이 잘 작동하지는 않을 것

$$y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad a^{[L]} = \hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 04 \end{bmatrix}$$

  - **신경망을 학습시키기 위한 손실함수**는?
    - 소프트맥스 분류에서 주로 사용하는 손실함수는 **j=1부터 4까지(일반적으로는 1부터 C까지) y_j*log(y^_j)합의 음수값**이다. 위의 식에서 y_1=y_3=y_4=0이고 y_2만 유일하게 1의 값을 갖는다. 이 합을 구할 때 y_j가 0이면 고려해주지 않아도 되므로 유일하게 남는 항은 **-y_2*log(y^-2)**.

  →이 항들을 모두 합할 때 j가 2인 경우를 제외하고는 모두 0이기 때문이다. y_2는 1이니 결국 -log (y^_2)가 된다.

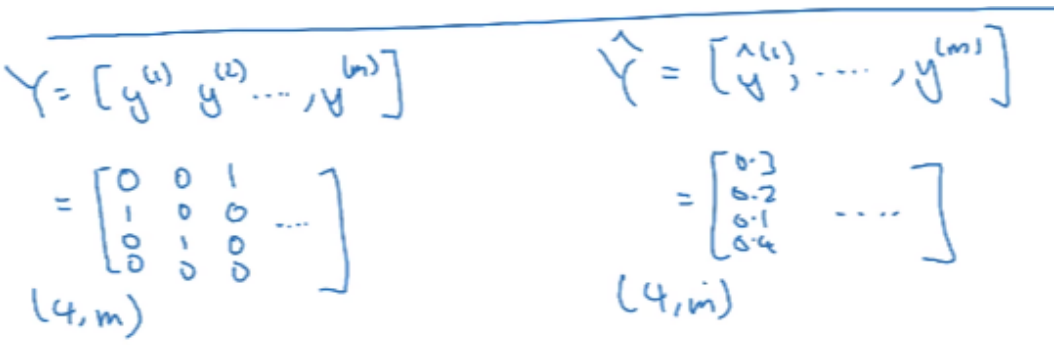
$$L(\hat{y}, y) = -\sum_{j-1}^{4} y_j \log y_j$$

- 학습 알고리즘이 **경사하강법을 이용해서 이 손실 함수의 값을 작게 만들려고 할 것**이다. **결국 -log (y^_2)의 값을 작게 만드는 것이다. 이는 y^_2의 값을 가능한 한 크게 만들어야 한다는 의미**이다.

→ 즉 일반적으로 손실함수는 훈련세트에서 관측에 따른 클래스가 무엇이든 간에 그 클래스에 대응하는 확률을 가능한 한 크게 만드는 것

- **전체 훈련 세트에 대해 비용함수 J는?**
  - 매개변수를 설정할 때 비용함수는 전체 훈련 세트에서 학습 알고리즘의 예측에 대한 손실함수를 합하는 것이다(훈련샘플들에 대해서).
  - 비용함수를 최소로 하기 위해 경사하강법을 써야 한다.
- **구현**
  - C=4이고 (4,1) 벡터인 상황에서 **y^도 (4,1) 벡터.**
  - Y 행렬: y^[1], y^[2]부터 y^[m]이 된다. 위에서 했던 샘플이 첫번째 훈련샘플이라고 한다면 첫번째 열은 0 1 0 0 이 된다. 두번째 샘플은 개가 되고 세번째 샘플은 기타이며 ... 계속한다. → **Y는 결국 (4,m) 차원의 행렬**이 된다.
  - y^은 y^(1)부터 y^(m)을 수평하게 쌓은 것.

→ 이는 첫번째 훈련샘플에 대한 출력값이므로 y^(1)이 된다. 그러면 Y는 0.3 0.2 0.1 0.4를 갖게 된다. Y도 (4,m)차원이 된다.

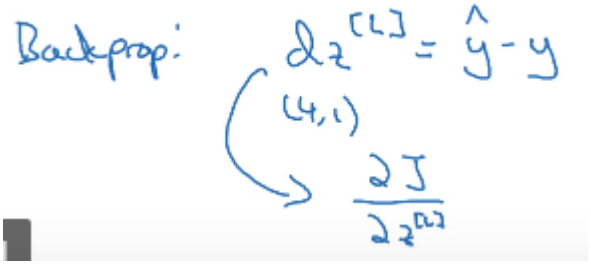


3. **Gradient descent with softmax**

- 마지막 출력층이 z^[L]을 계산.

→ **(C,1) 차원**이 되고(예시에서는 (4,1)차원) 여기에 소프트맥스 활성화 함수를 취해서 a^[L] 또는 y^을 얻는다. 그 값을 이용해서 손실함수를 계산할 수 있다.

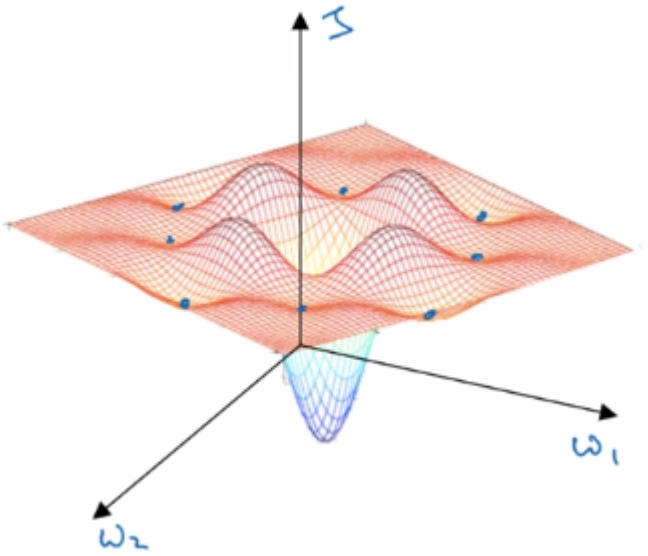
- dz^[L]: 비용함수 z^[L]에 대해 편미분한 것
- 신경망 전체에 대한 미분을 역방향전파로 구하기



8. 프로그래밍 프레임워크 소개

1. 지역 최적값 문제

1. **Local optima in neural networks**

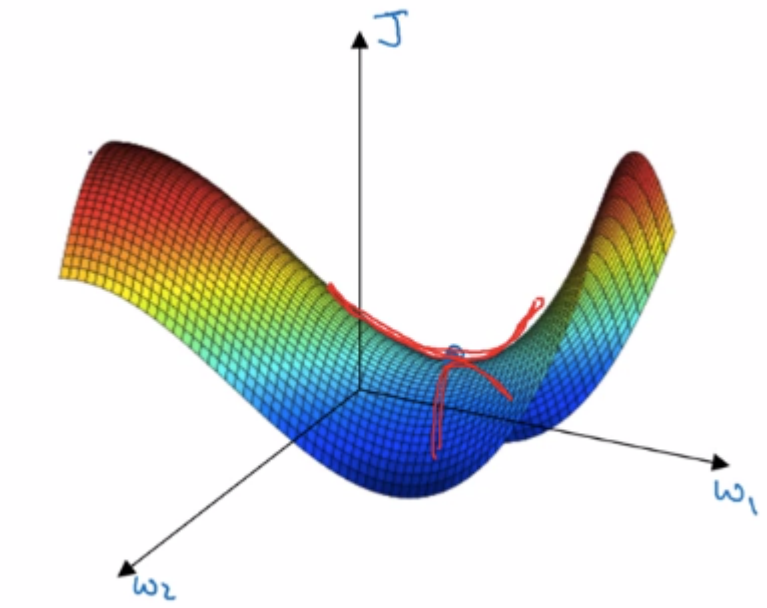


- w_1과 w_2라는 매개변수를 최적화한다고 할 때, 이 면적의 높이가 비용함수.
- 위의 그림에서는 지역 최적값이 많음.
- 경사하강법 등의 알고리즘이 전역 최적값에 도달하기 전에 지역 최적값에 갇혀버리기 쉽다.

→ 2차원에서 그림을 그린다면 서로 다른 지역 최적값이 많은 그림을 쉽게 접할 수 있다.

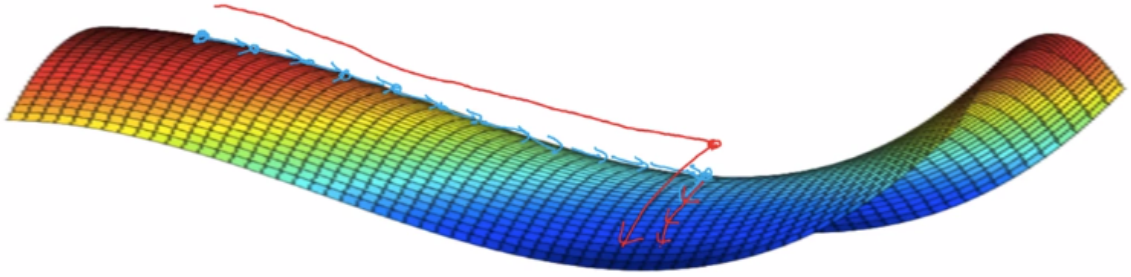
- **경사가 0인 점은 대부분 지역 최적값이 아니라 비용함수가 0인 경우 대개 안장점이다. (경사가 0일 점 중에 하나인 것). 사실 고차원 함수에서 경사가 0이면 각 방향에서 볼록함수나 오목함수가 되기 마련**이다.
- ex) 20,000 차원의 공간에서 지역 최적값이 되기 위해서는 20,000개의 방향이 모두 U(아래로 볼록 함수)와 같은 모양으로 생겨야 한다. 그러나 그런 일이 일어날 확률을 매우 낮다. (확률 적으로 2^(-20,000)정도)
    - **대신 어떤 방향에서는 위로 볼록한 형태가, 어떤 방향에서는 아래로 볼록한 형태가 주로 발생**할 것.

- 따라서 **고차원 공간에서는 지역 최적값보다 안장점이 되기 쉽다**. (안장점이라고 불리는 이유: 모양이 말에 얹는 안장과 비슷하기 때문)



→ **낮은 차원의 공간에서 얻었던 직관이 학습 알고리즘이 높은 차원에서 돌아갈 때 적용되지 않을 수도 있다**. 20,000개의 매개변수가 있을 때 J는 20,000 차원의 벡터에 대한 함수이고 지역 최적값보다 안장점을 훨씬 많이 볼 수 있을 것이다.

2. **Problem of plateaus**



- 진짜 문제점: **안정지대가 학습을 아주 지연시킬 수 있다는 것**
  - **안정 지대: 미분값이 아주 오랫동안 0에 가깝게 유지되는 지역**.
  - 만약 그림의 왼쪽 표시에 위치한다고 하면 경사하강법에 따라 면을 따라서 아래로 움직인다. **여기서 경사가 거의 0에 가까울테니 면이 거의 평평할 것**.

→ 아주 오랜 시간이 걸리게된다. 왼쪽이나 오른쪽에 무작위로 작은 변화가 주어지면 빨간색 화살표처럼 알고리즘이 안정지대를 벗어날 수 있다. 하지만 하늘색의 마지막 지점에 도달하기 전에 아주 긴 시간동안 경사를 탄 후에야 안정지대를 벗어날 수 있다.

- **충분히 큰 신경망을 학습시킨다면 지역최적값에 갇힐 일이 잘 없다. (여러 매개 변수와 비용함수 J가 상대적으로 고차원에서 정의되는 경우)**
- **안정지대는 학습속도가 매우 느려지기 때문에 문제이다. 여기서는 모멘텀이나 RMSprop, Adam 등 이런 알고리즘의 도움을 받을 수 있다. 이런 경우에는 Adam과 같은 최적화 알고리즘이 안정지대 내에서 움직이거나 벗어나는 속도를 올릴 수 있다.**

2. Tensorflow

: 딥러닝 프레임워크

1. Motivating problem

- 최소화하고 싶은 비용함수 J가 있다고 가정 → J(w) = w^2-10w+25
- J(w)=(w-5)^2와 같다.

→이 식을 최소로 하는 w는 5.

- 실습

```
[11] import numpy as np
     import tensorflow as tf
```

```
[16] import tensorflow.compat.v1 as tf

     tf.disable_v2_behavior()
```

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

     init = tf.compat.v1.global_variables_initializer()
     session = tf.Session()
     session.run(init)
     print(session.run(w))
```

```
0.0
```

```
session.run(train)
print(session.run(w))
```

```
0.099999994
```

```
[22] for i in range(1000):
       session.run(train)
     print(session.run(w))
```

```
4.9999886
```

```
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.compat.v1.global_variables_initializer()
session = tf.Session()
session.run(init)
print(session.run(w))
```

```
0.0
```

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

```
0.099999994
```

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

```
4.9999886
```

- coefficients의 -10을 -20으로 바꾸면: **w가 10과 아주 가까워짐**

```
coefficients = np.array([[1.],[-20.], [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.compat.v1.global_variables_initializer()
session = tf.Session()
session.run(init)
print(session.run(w))
```

```
0.0
```

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

```
0.19999999
```

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

```
9.999977
```