CS230: Deep Learning

Stanford University Lecture Notes

Compiled Study Notes

December 20, 2025

Contents

1 Logistic Regression as a Neural Network

Chapter 1. Deep Learning Introduction (Completed)

(이전 단원에서 우리는 딥러닝이 무엇인지, 그리고 데이터가 어떻게 새로운 석유가 되었는지 배웠습니다. 이제 그 거대한 딥러닝이라는 기계를 구성하는 가장 작은 부품을 뜯어볼 차례입니다.)

1.0.1 Logistic Regression as a Neural Network

☒ 연결 고리

거대한 빌딩도 벽돌 한 장에서 시작하듯, 아무리 복잡한 AI(LLM, AlphaGo 등)도 결국 '뉴런(Neuron)'이라는 작은 단위의 집합입니다. 이번 단원에서는 딥러닝의 가장 기초 단위인 로지스틱 회귀(Logistic Regression)를 하나의 신경망으로 해석하고, 그 내부 동작 원리를 완전히 해부합니다.

[☒ 단원 개요 (Overview)]

이 단원은 딥러닝 학습의 **'기초 체력'**을 다지는 구간입니다.

- **목표**: 이진 분류 문제를 신경망 구조(Input $\rightarrow$ Linear $\rightarrow$ Activation)로 모델링합니다.
- **핵심**: 계산 그래프를 통해 순전파(Forward)와 역전파(Backpropagation)를 유도합니다.
- **이유**: MSE 대신 **Binary Cross-Entropy**를 비용 함수로 쓰는 이유를 이해합니다.
- **구현**: Python(NumPy)을 사용하여 for-loop 없는 **벡터화(Vectorization)** 코드를 작성합니다.

1.0.2 핵심 용어 정리 (Terminology)

용어	설명 (한 줄 정의)
특징 벡터 (Feature Vector, x)	예측을 위해 입력되는 데이터의 정보들 (예: 이미지의 픽셀값)
가중치 (Weight, w)	입력 정보가 결과에 미치는 중요도 (클수록 중요한 정보)
편향 (Bias, b)	입력과 상관없이 기본적으로 가지는 성향 혹은 임계값
시그모이드 (Sigmoid, σ)	계산된 점수를 0과 1 사이의 확률로 변환하는 활성화 함수
교차 엔트로피 (Cross-Entropy)	확률 분포 간의 차이를 측정하는 비용 함수 (틀릴수록 값이 커짐)

1.0.3 Core Concepts: 신경망의 해부

1.0.4 1. 문제 정의: 이진 분류 (Binary Classification)

한 줄 요약: 질문에 대해 YES(1) 또는 NO(0)로 답하는 문제입니다.

[고양이 탐지기]

당신이 사진을 보고 "이것은 고양이입니까?"라는 질문에 답해야 한다고 상상해봅시다.

- 입력(x): 사진 속의 털, 귀 모양, 눈동자 색깔 등의 단서들.
- 출력($\hat{y}$): "고양이일 확률은 80

기술적 정의: n_x 차원의 입력 벡터 x가 주어졌을 때, 출력 y가 1일 확률 $\hat{y} = P(y = 1|x)$를 예측합니다.

1.0.5 2. 뉴런의 구조: 선형 결합과 활성화

로지스틱 회귀는 두 단계의 '생각 과정'을 거칩니다.

Step 1: 선형 결합 (Linear Function) - 점수 매기기

$$z = w^T x + b$$

- w (가중치): 각 단서가 얼마나 중요한지 결정합니다. (예: '뾰족한 귀'는 고양이 판별에 중요함 $\rightarrow$ 높은 w)
- b (편향): 입력이 0이어도 기본적으로 갖는 점수입니다. (예: "나는 동물을 좋아해서 일단 고양이로 보고 싶어" $\rightarrow$ 높은 b)

Step 2: 활성화 함수 (Sigmoid) - 확률로 변환

$$a = \sigma(z) = \frac{1}{1 + e^{-z}}$$

선형 결합의 결과 z는 $-\infty$에서 $+\infty$까지의 값을 가질 수 있습니다. 이를 $0 \sim 1$ 사이의 확률로 압축하는 과정입니다.

- z가 매우 크면 $\rightarrow a \approx 1$ (확신)
- z가 0이면 $\rightarrow a = 0.5$ (반반)
- z가 매우 작으면 $\rightarrow a \approx 0$ (아님)

1.0.6 3. 비용 함수 (Cost Function): 왜 MSE가 아닐까?

우리는 모델이 예측한 값(a)과 실제 정답(y)이 다르다면 모델을 '혼내줘야' 합니다. 그 벌점을 매기는 규칙이 비용 함수입니다.

[MSE(평균 제곱 오차)를 쓰면 안 되나요?]

선형 회귀에서는 MSE($\frac{1}{2}(\hat{y} - y)^2$)를 쓰지만, 로지스틱 회귀에서 이를 쓰면 비용 함수가 **울퉁불퉁한(Non-Convex)** 모양이 됩니다. 즉, 경사 하강법을 할 때, 진짜 최소점(Global Minimum)이 아닌 웅덩이(Local Optima)에 빠져 학습이 멈출 수 있습니다.

따라서 우리는 매끄러운 그릇 모양(Convex)을 보장하는 이진 교차 엔트로피(Log Loss)를 사용합니다.

$$L(a, y) = -(y \log(a) + (1 - y) \log(1 - a))$$

- 정답이 1($y = 1$)인데 예측을 0에 가깝게 하면($a \rightarrow 0$), $-\log(a)$ 때문에 비용이 무한대로 치솟습니다. (엄청난 벌점!)

1.0.7 Numerical Example: 손으로 풀어보는 로지스틱 회귀

수식을 이해하는 가장 좋은 방법은 숫자를 넣어보는 것입니다.

[시험 합격 예측 시나리오]

상황: 학생의 '공부 시간(x_1)'과 '수면 시간(x_2)'으로 '합격($y = 1$)'을 예측합니다.

- **데이터**: 공부 2시간($x_1 = 2$), 수면 5시간($x_2 = 5$). 실제 결과: 합격($y = 1$).
- **초기 파라미터**: $w_1 = 0.1, w_2 = -0.1, b = 0.0$ (초기화 상태)

Step 1: 선형 계산 (Forward)

$$z = (w_1 \cdot x_1) + (w_2 \cdot x_2) + b$$

$$z = (0.1 \cdot 2) + (-0.1 \cdot 5) + 0 = 0.2 - 0.5 = -0.3$$

Step 2: 활성화 (Sigmoid)

$$a = \frac{1}{1 + e^{-(-0.3)}} = \frac{1}{1 + 1.349} \approx 0.425$$

→ 모델은 합격 확률을 42.5%로 예측했습니다. (실제는 합격인데, 예측이 틀렸네요!)

Step 3: 비용 계산 (Loss)

$$L = -(1 \cdot \log(0.425) + 0 \cdot \log(\dots)) \approx -(-0.855) = 0.855$$

→ 벌점은 0.855입니다.

Step 4: 역전파 (Backward) - 학습의 핵심 우리는 정답($y = 1$)에 가까워지도록 w를 수정해야 합니다. 여기서 마법의 공식 $dz = a - y$가 등장합니다.

$$dz = a - y = 0.425 - 1 = -0.575$$

이제 가중치에 대한 기울기(dw)를 구합니다.

$$dw_1 = x_1 \cdot dz = 2 \cdot (-0.575) = -1.15$$

$$dw_2 = x_2 \cdot dz = 5 \cdot (-0.575) = -2.875$$

$$db = dz = -0.575$$

Step 5: 파라미터 업데이트 (Gradient Descent) 학습률 $\alpha = 0.1$이라고 가정합시다.

$$w_1 \leftarrow w_1 - \alpha \cdot dw_1 = 0.1 - 0.1(-1.15) = 0.1 + 0.115 = 0.215$$

$$w_2 \leftarrow w_2 - \alpha \cdot dw_2 = -0.1 - 0.1(-2.875) = -0.1 + 0.2875 = 0.1875$$

결과 해석: w_1 (공부 시간 중요도)이 0.1에서 0.215로 증가했습니다. 즉, 모델은 "공부를 많이 할수록 합격한다"는 것을 배웠습니다!

1.0.8 Python Implementation (Vectorization)

이론을 실제 코드로 옮겨봅시다. 여기서 중요한 것은 'for' 루프를 쓰지 않는 벡터화입니다.

Listing 1: Vectorized Logistic Regression

```
import numpy as np

def propagate(w, b, X, Y):
    """
␣␣␣␣Arguments:
␣␣␣␣w␣--␣□ □ □ ␣(n_x,␣1)
␣␣␣␣b␣--␣□ □ ␣(scalar)
␣␣␣␣X␣--␣□ □ ␣□ □ □ ␣(n_x,␣m)␣->␣□ m␣□ □ □ ␣□ □
␣␣␣␣Y␣--␣□ □ ␣□ □ □ ␣(1,␣m)
␣␣␣␣"""
    m = X.shape[1]

    # --- Forward Propagation □ □ □ () ---
    # □ □  □ □ □ □  □ □ m □ □ □ □  □  □ □  □ □  (Vectorization)
    Z = np.dot(w.T, X) + b
    A = 1 / (1 + np.exp(-Z)) # Sigmoid

    # Cost □ □
    cost = -1/m * np.sum(Y * np.log(A) + (1-Y) * np.log(1-A))

```

```
    # --- Backward Propagation □ □ □ () ---
    # □ □ □ □ □ □  □ □ □  □ □ : dZ = A - Y
    dZ = A - Y

    # Gradients □ □
    dw = 1/m * np.dot(X, dZ.T) # □ □  □ □ : (n, m) * (m, 1) = (n, 1)
    db = 1/m * np.sum(dZ)

    return dw, db, cost
```

[주의: Rank-1 Array Pitfall]

NumPy에서 'a = np.random.randn(5)'는 '(5,)' 형태를 가집니다. 이는 행 벡터도 열 벡터도 아니어서 전치('T')를 해도 모양이 바뀌지 않아 버그를 유발합니다. 반드시 'a = np.random.randn(5, 1)' 처럼 차원을 명시하십시오.

1.0.9 자주 묻는 질문 (FAQ)

Q1. 편향(Bias) b는 왜 필요한가요? 없으면 안 되나요?
A. 원점을 반드시 지나야 한다는 제약이 생깁니다. 예를 들어, 공부를 하나도 안 했어도($x = 0$) 합격할 확률이 0이 아닐 수 있습니다. b는 그래프를 좌우로 이동시켜 데이터에 더 잘 맞도록 해주는 '유연성'을 제공합니다.

Q2. 초기화할 때 w를 0으로 둬도 되나요?
A. **로지스틱 회귀에서는 가능합니다.** 비용 함수가 볼록(Convex)하기 때문에 어디서 시작하든 바닥(최적해)으로 굴러갑니다. 하지만 나중에 배울 심층 신경망(Deep Network)에서는 절대 0으로 초기화하면 안 됩니다(대칭성 문제).

Q3. 학습률(Learning Rate)이 너무 크면 어떻게 되나요?
A. 보폭이 너무 커서 최적점을 지나쳐 버리거나(Overshooting), 영원히 수렴하지 않고 발산할 수 있습니다. 반대로 너무 작으면 학습 속도가 너무 느려집니다.

[☒ 단원 요약 (Chapter Summary)]

1. 로지스틱 회귀는 딥러닝의 가장 작은 단위인 1-Layer Neural Network이다.
2. 구조: $z = w^T x + b$ (선형) $\rightarrow a = \sigma(z)$ (비선형 활성화).
3. 학습: 이진 교차 엔트로피를 최소화하는 방향으로 경사 하강법을 수행한다.
4. 구현: m개의 데이터를 'for'문 없이 처리하기 위해 Vectorization(행렬 연산)을 사용한다.

☒ 다음 단원 예고

축하합니다! 여러분은 이제 신경망의 '뇌세포' 하나를 완벽하게 만들 수 있습니다. 다음 장 [Chapter 3. Shallow Neural Networks]에서는 이 세포들을 옆으로 나란히 연결하고, 뒤로 층층이 쌓아서 더 복잡한 문제를 해결하는 은닉층(Hidden Layer)의 마법을 배워보겠습니다.

2 Binary Classification Cost Function

☒ 지난 시간 복습 및 연결

지난 시간, 우리는 딥러닝이라는 거대한 숲(Big Picture)을 보았습니다. 이제 현미경을 꺼내 들 시간입니다. 숲을 이루는 가장 작은 단위인 나무(뉴런) 하나를 완벽하게 해부해 봅시다. 로지스틱 회귀를 단순한 통계 기법이 아닌, '가장 얕은 신경망(Shallow Neural Network)'으로 이해하는 것이 딥러닝 마스터의 첫걸음입니다.

2.1 Overview

[핵심 요약]

이 단원은 딥러닝 모델의 최소 단위인 '단일 뉴런'의 작동 원리를 다룹니다.

- **목표:** 이진 분류 문제를 입력 $\rightarrow$ 선형 계산 $\rightarrow$ 활성화 $\rightarrow$ 출력의 신경망 구조로 재해석합니다.
- **핵심:** 계산 그래프를 통해 순전파(Forward)와 역전파(Backward)의 수학적 흐름을 이해합니다.
- **구현:** 'for-loop' 없이 행렬 연산(Vectorization)을 사용하여 효율적인 코드를 작성합니다.
- **이유:** MSE 대신 Binary Cross-Entropy를 비용 함수로 사용하는 이유를 볼록(Convex) 최적화 관점에서 배웁니다.

2.1.1 Essential Terminology

딥러닝 엔지니어들이 숨 쉬듯 사용하는 용어들입니다.

기호	용어	한 줄 정의
x	입력 (Input)	판단의 근거가 되는 데이터 (예: 이미지 픽셀값)
w	가중치 (Weight)	각 입력 정보의 중요도 (클수록 결과에 큰 영향)
b	편향 (Bias)	입력과 무관한 기본 성향 혹은 임계값
z	선형 결과	가중치와 입력을 곱하고 더한 1차 점수 ($w^T x + b$)
$\sigma(z)$	활성화 함수	점수(z)를 확률($0 \sim 1$)로 변환하는 필터 (Sigmoid)
$\hat{y}$	예측값 (Output)	모델이 추측한 정답 확률 (a라고도 씀)
L	손실 (Loss)	예측이 틀렸을 때 부과하는 벌점 (하나의 데이터)
J	비용 (Cost)	전체 데이터에 대한 손실의 평균 (전체 성적표)

2.1.2 Core Concepts: 뉴런의 해부

2.1.3 1. 신경망적 구조 (The Architecture)

로지스틱 회귀는 두 단계의 생각 과정을 거치는 단일 뉴런입니다.

Step 1: 선형 결합 (Linear Combination)

입력된 정보들을 중요도(w)에 따라 합산합니다.

$$z = w^T x + b$$

Step 2: 비선형 활성화 (Activation)
계산된 점수(z)는 $-\infty \sim \infty$ 범위를 가집니다. 이를 확률($0 \sim 1$)로 바꾸기 위해 시그모이드(Sigmoid) 함수를 통과시킵니다.

$$\hat{y} = a = \sigma(z) = \frac{1}{1+e^{-z}}$$

[대학 입학 사정관 비유]
이 뉴런을 '깐깐한 입학 사정관'이라고 상상해 봅시다.

1. **입력 (x):** 학생의 내신 성적(x_1), 수능 점수(x_2), 봉사 시간(x_3).
2. **가중치 (w):** 사정관의 평가 기준. (수능이 중요하면 w_2가 큼).
3. **편향 (b):** 학교의 관대함. (점수가 낮아도 일단 긍정적으로 보면 $b > 0$).
4. **선형 결합 (z):** $z = (x_1w_1 + x_2w_2 + x_3w_3) + b$. (학생의 총점 계산).
5. **활성화 (σ):** 총점이 1000점이든 -500점이든, 합격 확률은 0%에서 100% 사이여야 합니다. 시그모이드는 이 점수를 확률로 매핑합니다.

2.1.4 2. 비용 함수 (Cost Function): 왜 MSE가 아닌가?

우리는 모델이 정답을 맞히면 칭찬하고, 틀리면 벌점(Cost)을 줘야 합니다. 선형 회귀에서 쓰던 MSE(평균 제곱 오차)를 쓰면 안 될까요?
[MSE 사용 금지 경보]
로지스틱 회귀(Sigmoid 포함)에 MSE를 적용하면 비용 함수 그래프가 울퉁불퉁한 계란판 모양(Non-Convex)이 됩니다. 경사 하강법을 쓸 때, 가장 깊은 골짜기(Global Minimum)가 아닌 엉뚱한 웅덩이(Local Optima)에 빠져 학습이 멈출 수 있습니다.

그래서 우리는 매끄러운 그릇 모양(Convex)을 보장하는 로그 손실(Binary Cross-Entropy)을 사용합니다.

$$J(w,b) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log(\hat{y}^{(i)}) + (1-y^{(i)})\log(1-\hat{y}^{(i)})]$$

- 정답이 1인데 0이라고 예측하면? $-\log(0) = \infty$ (무한대의 벌점!)
- 틀릴수록 기하급수적으로 큰 페널티를 부여하여 빠르게 수정하게 만듭니다.

2.1.5 Numerical Example: 손으로 푸는 로지스틱 회귀

수식을 눈으로만 보면 이해되지 않습니다. 숫자를 넣어봅시다.
[야간 자율학습 도망자 잡기 시나리오]
상황: 선생님(모델)이 학생의 행동을 보고 '도망($y = 1$)' 갈지 예측합니다.

- **입력 x:** 가방을 쌈($x_1 = 1$), 눈치를 봄($x_2 = 5$, 매우 많이 봄).
- **가중치 w:** $w_1 = 2.0$ (가방 싸는 건 중요), $w_2 = 0.5$ (눈치는 덜 중요).
- **편향 b:** -3.0 (선생님은 기본적으로 학생을 믿음).

1. 순전파 (Forward): 예측하기
선형 점수 계산:

$$z = (1 \times 2.0) + (5 \times 0.5) + (-3.0) = 2.0 + 2.5 - 3.0 = 1.5$$

활성화(확률 변환):

$$a = \frac{1}{1+e^{-1.5}} \approx \frac{1}{1+0.223} \approx 0.817$$

→ 선생님은 이 학생이 도망갈 확률을 81.7%로 예측했습니다.

2. 역전파 (Backward): 학습하기

실제 결과: 학생이 도망갔습니다 ($y = 1$). 오차 계산 (Magic Step):

$$dz = a - y = 0.817 - 1 = -0.183$$

이 값은 "내가 0.183만큼 부족하게 예측했구나"라는 직관적인 오차입니다. 이제 w를 업데이트하기 위해 미분값(Gradient)을 구합니다.

$$dw_1 = x_1 \times dz = 1 \times (-0.183) = -0.183$$

$$dw_2 = x_2 \times dz = 5 \times (-0.183) = -0.915$$

3. 파라미터 업데이트 (경사 하강법)

학습률 $\alpha = 0.1$이라면:

$$w_1 \leftarrow 2.0 - 0.1(-0.183) = 2.0183$$

결과: w_1이 증가했습니다. 즉, "가방을 싸는 행동"이 도망에 더 중요한 단서라고 학습했습니다!

2.1.6 Implementation: Vectorization (벡터화)

이제 Python으로 구현합니다. m개의 데이터를 처리할 때 'for' 루프를 쓰면 느립니다. 'numpy'의 행렬 연산을 써야 합니다.

Listing 2: Vectorized Logistic Regression Unit

```
import numpy as np

class LogisticUnit:
    def __init__(self, input_dim):
        # □ w (input_dim, 1) □ □ □ □ □ □ □ □ □ □
        # □ □ □ □ □ □ □ □ □ □ 0 □ □ □ □ □ □ □ □ □ □ □ □ (Deep □ NN □ □ !)
        self.w = np.zeros((input_dim, 1))
        self.b = 0.0

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def propagate(self, X, Y):
        """
␣␣␣␣␣␣␣␣X:␣(n_x,␣m)␣□ □ ␣¬␣□ □ m␣□ □ □ □ ␣□ □ ␣□ □ □
␣␣␣␣␣␣␣␣Y:␣(1,␣m)␣□ □ ␣¬␣□ □ ␣□ □ □
␣␣␣␣␣␣␣␣"""
        m = X.shape[1] # □ □ □ □ □

        # 1. Forward Propagation □ ( □ □ □ m □ □ )
        # (n_x, 1).T @ (n_x, m) + scalar -> (1, m)
        Z = np.dot(self.w.T, X) + self.b
        A = self.sigmoid(Z)

        # □ □ □ □ (Binary Cross-Entropy)
        cost = -1/m * np.sum(Y * np.log(A) + (1-Y) * np.log(1-A))

        # 2. Backward Propagation □ □ (: Chain Rule)
        # dZ = A - Y □ □ □ □ ( □ □ □ □ □ □ ) -> □ □ □ □ □ □ □ □ □ !
        dZ = A - Y

        # Gradient □ □ (Vectorized)
        dw = 1/m * np.dot(X, dZ.T)
        db = 1/m * np.sum(dZ)

        return {"dw": dw, "db": db}, cost
```

2.1.7 FAQ: 초심자가 자주 묻는 질문

- **Q1. 편향(b)이 왜 필요한가요? 없으면 안 되나요?**
  A. 편향이 없으면 결정 경계가 무조건 원점(0,0)을 지나야 합니다. 예를 들어, 아무런 행동을 안 해도($x = 0$) 합격률이 50%가 넘을 수 있는데, b가 없으면 이를 표현할 수 없습니다. b는 그래프를 좌우로 움직이는 '유연성'을 줍니다.

- **Q2. 가중치 w를 0으로 초기화해도 학습이 되나요?**
  A. 로지스틱 회귀에서는 YES. 비용 함수가 밥그릇 모양(Convex)이라서 어디서 시작하든 바닥으로 굴러갑니다. 하지만 나중에 배울 다층 신경망에서는 절대 안 됩니다(대칭성 문제).

- **Q3. 학습률(Learning Rate)을 어떻게 정하나요?**
  A. 너무 크면 정답을 지나쳐 발산(Overshooting)하고, 너무 작으면 학습이 영원히 걸립니다. 보통 0.01, 0.001 등으로 시작해 비용(Cost) 그래프가 잘 내려가는지 보며 조정합니다.

☒ 다음 단계 (Next Step)

축하합니다! 여러분은 딥러닝의 가장 기본 부품인 '뉴런' 하나를 완벽하게 이해하고 구현했습니다.

다음 장 [Chapter 3. Shallow Neural Networks]에서는 이 뉴런들을 옆으로 나란히 배치하고, 뒤로 연결하여 은닉층(Hidden Layer)을 만드는 법을 배웁니다. 뉴런 하나로는 단순한 선형 분류만 가능하지만, 뉴런이 모이면 복잡한 비선형 문제도 해결할 수 있습니다.

[단원 요약 (Cheat Sheet)]

1. **구조:** 로지스틱 회귀 = $z = w^T x + b$ (선형) $\rightarrow \sigma(z)$ (비선형 활성화).
2. **학습:** Cross-Entropy 비용 함수를 최소화하는 방향으로 w, b를 업데이트.
3. **수학:** 역전파의 핵심 미분 값은 $dZ = A - Y$ (예측 - 정답).
4. **구현:** 'for-loop' 대신 'np.dot'을 활용한 Vectorization 필수.

3 Gradient Descent Optimization

☒ 지난 시간 복습 및 연결

우리는 지난 시간에 로지스틱 회귀의 '뇌 구조(Architecture)'를 만들고, 입력 신호를 흘려보내는 '순전파(Forward Propagation)'를 설계했습니다. 하지만 지금 이 신경망은 갓 태어난 아기와 같습니다. 세상에 대해 아무것도 모르죠(파라미터가 초기화된 상태). 이제 이 아이를 가르칠 시간입니다. 학습이란 "내가 얼마나 틀렸는지 확인하고(Cost), 고쳐 나가는(Gradient Descent) 과정"입니다.

3.1 Overview

[핵심 목표]

이 유닛은 머신러닝의 '엔진(Engine)'을 다룹니다. 차체(모델 구조)가 좋아도 엔진(학습 알고리즘)이 없으면 움직이지 않습니다.

- **구분:** 데이터 하나에 대한 오차(Loss)와 전체 성적표(Cost)를 구분합니다.
- **이유:** 왜 MSE 대신 Log Loss(Binary Cross-Entropy)를 써야 하는지 '지형(Topology)' 관점에서 이해합니다.
- **원리:** 산에서 내려오는 방법인 경사 하강법(Gradient Descent)의 원리를 배웁니다.
- **조절:** 학습률(Learning Rate)이 학습 속도에 미치는 영향을 분석합니다.

3.1.1 Essential Terminology

용어	기호	한 줄 핵심 요약
손실 함수 (Loss)	$L(\hat{y}, y)$	데이터 **샘플 1개**에 대한 오차 (작을수록 좋음)
비용 함수 (Cost)	$J(w, b)$	전체 학습 데이터(m개)에 대한 **Loss의 평균**
볼록성 (Convexity)	-	밥그릇처럼 매끄러운 모양 (최소점이 하나뿐인 안전한 지형)
기울기 (Gradient)	dw, db	현재 위치에서 가장 가파른 경사의 방향
학습률 (Learning Rate)	α	한 번 업데이트할 때 이동하는 보폭의 크기

3.1.2 Core Concepts: 학습의 매커니즘

3.1.3 1. Loss vs. Cost (오차의 정의)

한 줄 요약: Loss는 '쪽지시험 점수', Cost는 '학기말 평균 성적'입니다.
[시험 점수 비유]

- **Loss Function (L):** 1번 학생이 문제를 틀렸습니다. 이 학생 하나의 오차입니다.
- **Cost Function (J):** 우리 반 30명 전체의 평균 오차입니다. 선생님(모델)의 목표는 특정 학생만 잘 가르치는 게 아니라, 반 전체의 평균 성적(J)을 좋게 만드는 것입니다.

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

3.1.4 2. Why Log Loss? (MSE의 함정)

한 줄 요약: 로지스틱 회귀에서 MSE를 쓰면 함정이 많은 산이 되지만, Log Loss를 쓰면 매끄러운 밥그릇이 됩니다.

기술적 정의: 선형 회귀와 달리 Sigmoid 함수가 포함된 로지스틱 회귀에 MSE($\frac{1}{2}(\hat{y}-y)^2$)를 적용하면 비용 함수가 비볼록(Non-Convex) 형태가 됩니다. 이는 수많은 국소 최적해(Local Optima)를 만듭니다.

[MSE를 쓰면 안 되는 이유]

위 그림의 오른쪽(Non-Convex)을 보세요. 울퉁불퉁한 지형에서는 구슬을 굴렸을 때 가장 깊은 바닥(Global Minimum)이 아니라, 중간에 있는 작은 웅덩이(Local Minimum)에 갇혀버립니다. 학습이 망했다는 뜻입니다. 반면, 로그 손실(Log Loss)을 사용하면 왼쪽(Convex)처럼 매끄러운 그릇 모양이 되어, 어디서 시작하든 바닥으로 수렴합니다.

우리가 사용할 공식 (Binary Cross-Entropy):

$$L(\hat{y}, y) = -(y \log(\hat{y}) + (1-y) \log(1-\hat{y}))$$

- 정답(y)이 1일 때: 예측($\hat{y}$)이 1이면 비용 0, 0이면 비용 ∞.
- 틀렸을 때 무한대의 벌점을 주어 빠르게 고치도록 유도합니다.

3.1.5 3. Gradient Descent (경사 하강법)

한 줄 요약: 눈을 가린 채 산에서 가장 낮은 골짜기로 내려가는 방법입니다.

[안개 낀 산 하산하기]

당신은 짙은 안개가 낀 산 정상에 서 있습니다. 앞이 보이지 않습니다. 가장 낮은 곳(비용 최소화 지점)으로 가려면 어떻게 해야 할까요?

1. 발로 땅을 더듬어 경사가 가장 급하게 내려가는 방향을 찾습니다. (**Gradient 계산**)
2. 그 방향으로 한 발자국 내딛습니다. (**Update**)
3. 바닥에 도착할 때까지 반복합니다.

업데이트 공식:

$$w := w - \alpha \frac{\partial J}{\partial w}$$

$$b := b - \alpha \frac{\partial J}{\partial b}$$

- **빼기($-$)의 의미:** 기울기가 양수(오르막)라면 w를 줄여야(왼쪽으로 가야) 내려갈 수 있습니다. 반대 방향으로 가야 하므로 뺍니다.
- α **(Learning Rate):** 한 발자국의 크기입니다.

3.1.6 Practical Scenario: 학습률 α의 중요성

학습률(Learning Rate) α는 모델의 운명을 결정하는 가장 중요한 숫자(Hyperparameter)입니다.

- **Case A:** α**가 너무 작을 때 (0.00001)**
  개미처럼 기어갑니다. 해가 질 때까지(학습 종료까지) 산 중턱에도 못 갑니다. (수렴 속도 매우 느림)
- **Case B:** α**가 너무 클 때 (10.0)**
  거인의 점프입니다. 골짜기를 향해 뛰었는데 너무 멀리 뛰어서 반대편 산등성이에 처박힙니다. 오히려 더 높은 곳으로 올라갈 수도 있습니다. (**Overshooting / Divergence**)

3.1.7 Numerical Example: 비용 계산 해보기

[비용 함수 계산 실습]

상황: 고양이 사진($y = 1$)을 보여줬는데, 모델이 0.8(80%)로 예측했습니다.

$$y = 1, \quad \hat{y} = 0.8$$

1. 손실(Loss) 계산: 공식: $L = -(1 \cdot \log(0.8) + 0 \cdot \log(0.2))$

$$L = -\log(0.8) \approx -(-0.223) = 0.223$$

상황 변경: 만약 모델이 0.1(10%)로 잘못 예측했다면?

$$L = -\log(0.1) \approx -(-2.30) = 2.30$$

→ 예측이 틀릴수록 벌점(Loss)이 0.223에서 2.30으로 10배 넘게 커졌습니다! 이것이 Log Loss의 위력입니다.

3.1.8 Implementation (Python)

이론을 'numpy' 코드로 옮겨봅시다.

Listing 3: Cost Function and Optimization

```
import numpy as np

def compute_cost(A, Y):
    """
␣␣␣␣A:␣□ □ □ ␣(1,␣m),␣Y:␣□ □ □ ␣(1,␣m)
␣␣␣␣"""
    m = Y.shape[1]

    # log(0) □ □ □  □□  □□  □□  □(epsilon)□  □ □ □ □  □□  □ □ □ □ □ .
    epsilon = 1e-5

    # Binary Cross-Entropy □ □  (Element-wise multiplication)
    cost = -1/m * np.sum(Y * np.log(A + epsilon) + (1-Y) * np.log(1-A + epsilon))

    return float(np.squeeze(cost)) # □ □ □  □ □ □ □  □ □

def update_parameters(w, b, dw, db, learning_rate):
    """
␣␣␣␣□ □ ␣□ □ □ ␣□ □ □ □ ␣□ □
␣␣␣␣"""
    # □ □  □ □ □ □  □ □ □ (dw, db)□  □□  □ □ □ □  □□ alpha □ □
    w = w - learning_rate * dw
    b = b - learning_rate * db

    return w, b
```

3.1.9 FAQ: 초심자가 자주 묻는 질문

- **Q1. 경사 하강법 공식에서 왜 더하지 않고 빼나요?**
  A. 기울기(Gradient)는 함수가 '증가하는' 방향을 가리킵니다. 우리는 비용을 '줄여야' 하므로 기울기의 반대 방향으로 가야 합니다. 그래서 뺍니다.

- **Q2. 비용 함수가 0이 되면 좋은 건가요?**
  A. 이론적으로는 완벽하지만, 현실에서는 과적합(Overfitting)을 의심해야 합니다. 문제집 답을 달달 외운 상태일 수 있어서, 새로운 문제(Test Set)는 못 풀 수도 있습니다.

☒ 다음 단계 (Next Step)

이제 우리는 구조(Architecture)를 만들었고, 학습 방법(Optimizer)까지 장착했습니다. 다음 장에서는 이 모든 부품을 조립하여 실제 데이터를 입력받아 학습하고 예측하는 전체 모델(Full Model)을 완성하겠습니다.

[단원 요약 (Cheat Sheet)]

1. **Cost Function:** 전체 데이터의 오차 평균(J)을 최소화하는 것이 목표다.
2. **Log Loss:** 로지스틱 회귀에는 MSE 대신 Log Loss를 써야 Convex(볼록)해진다.
3. **Gradient Descent:** $w_{new} = w_{old} - \alpha \cdot dw$. 경사를 타고 내려가는 알고리즘.
4. **Learning Rate:** 너무 크면 발산, 너무 작으면 느리다. 적절한 튜닝이 필요하다.

4 Vectorization

☒ 지난 시간 복습 및 연결

우리는 지난 시간까지 딥러닝의 '이론적 토대(비용 함수, 경사 하강법)'를 완성했습니다. 이론적으로는 완벽합니다. 하지만 이 수식을 컴퓨터에게 그대로 주면 학습하는 데 수십 년이 걸릴지도 모릅니다. 이제 이 이론에 '제트 엔진'을 달아줄 시간입니다. 초심자와 전문가를 가르는 가장 결정적인 기술, 벡터화(Vectorization)를 배워봅시다.

4.1 Overview

[핵심 목표]

이 단원은 딥러닝 코드를 수백 배 빠르게 만드는 '최적화 기술'을 다룹니다.

- **개념:** 'for-loop'를 죄악시하고, 행렬 단위 연산(Vectorization)을 해야 하는 이유를 배웁니다.
- **원리:** CPU/GPU의 SIMD(병렬 처리) 아키텍처가 어떻게 연산을 가속하는지 이해합니다.
- **기술:** NumPy의 핵심 기능인 브로드캐스팅(Broadcasting)의 규칙과 위험성을 파악합니다.
- **검증:** 실제 코드로 100만 개의 데이터를 연산해보며 속도 차이를 눈으로 확인합니다.

4.1.1 Essential Terminology

용어	설명	한 줄 핵심 요약
Vectorization	벡터화	반복문 없이 데이터를 통째로(행렬로) 연산하는 기법
SIMD	Single Instruction, Multiple Data	명령어 하나로 여러 데이터를 동시에 처리하는 CPU 기술
Broadcasting	브로드캐스팅	모양이 다른 배열끼리 연산할 때 자동으로 크기를 맞춰주는 기능
NumPy	넘파이	파이썬의 느린 속도를 C언어 레벨 최적화로 극복한 수치 연산 라이브러리
Rank-1 Array	랭크-1 배열	'(5,)' 처럼 행도 열도 아닌 애매한 배열 (버그의 주범)

4.1.2 Core Concepts: 속도의 비밀

4.1.3 1. Vectorization (벡터화란 무엇인가?)

한 줄 요약: 하나씩 처리하지 말고, 트럭에 실어서 한 번에 옮기십시오.

[이사짐 옮기기 비유]

100만 개의 벽돌(데이터)을 옮겨야 합니다.

- **For-loop (Non-vectorized):** 인부가 벽돌을 손에 하나씩 들고 100만 번 왕복합니다. (파이썬이 데이터를 하나씩 꺼내서 처리함)
- **Vectorization:** 100만 개의 벽돌을 거대한 덤프트럭(행렬)에 싣고 단 한 번에 이동합니다. (NumPy가 데이터를 통째로 메모리에 올려 처리함)

기술적 정의: $z = w^T x + b$를 계산할 때, $w_1 x_1, w_2 x_2 \dots$를 순회하지 않고, w와 x 전체 벡터를 한 번에 내적(Dot Product)하는 것입니다.

4.1.4 2. Under the Hood: SIMD (하드웨어의 마법)

왜 NumPy('np.dot')가 'for'문보다 빠를까요? 단순히 C언어로 짜여서가 아닙니다. 컴퓨터 구조적인 이유가 있습니다.

- **SISD (Single Instruction, Single Data):** 일반적인 'for'문입니다. CPU가 "가져와", "곱해", "저장해"를 데이터 하나마다 반복합니다.
- **SIMD (Single Instruction, Multiple Data):** 최신 CPU는 "이 8개의 데이터를 동시에 곱해!"라는 명령을 내릴 수 있습니다. 벡터화는 이 병렬 처리 기능을 활용합니다.

4.1.5 3. Broadcasting (브로드캐스팅)

한 줄 요약: 작은 행렬을 큰 행렬 크기에 맞게 자동으로 '늘려서(Stretch)' 연산합니다.

- **상황:** (4×1) 행렬에 숫자 100(스칼라)을 더하고 싶습니다.
- **원칙:** 수학적으로는 불가능하지만, Python은 100을 자동으로 (4×1) 크기로 복사하여 더해줍니다.
- **예시:**

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + 100 \rightarrow \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + \begin{bmatrix} 100 \\ 100 \\ 100 \\ 100 \end{bmatrix} = \begin{bmatrix} 101 \\ 102 \\ 103 \\ 104 \end{bmatrix}$$

4.1.6 Implementation & Benchmark (성능 검증)

"백문이 불여일견"입니다. 100만 개의 데이터를 곱하는 시간을 직접 측정해 봅시다.
[For-loop vs Vectorization 속도 대결]

```python
import numpy as np
import time

# □□□ □□ : □ 100 □□ □□ □□
a = np.random.rand(1000000)
b = np.random.rand(1000000)

# --- 1. For-loop □□ ( □□ ) ---
c = 0
tic = time.time()
for i in range(1000000):
    c += a[i] * b[i]
toc = time.time()

print(f"For-loop:␣{c:.4f}")
print(f"Time:␣{1000␣*␣(toc␣-␣tic):.2f}␣ms") # □ 400~500ms □□

# --- 2. Vectorization □□ ( □□ ) ---
tic = time.time()
c_vec = np.dot(a, b) # SIMD □□ □□
toc = time.time()

print(f"Vectorized:␣{c_vec:.4f}")
print(f"Time:␣{1000␣*␣(toc␣-␣tic):.2f}␣ms") # □ 1~2ms □□
```

결과 분석: 벡터화 코드가 약 **300 500배** 더 빠릅니다. 딥러닝 모델 학습 시간이 1달 걸릴 것을 2시간으로 줄여주는 마법입니다.

4.1.7 Pitfalls & FAQ

[Rank-1 Array의 함정]

NumPy에서 'a = np.random.randn(5)'를 하면 모양(Shape)이 '(5,)'가 됩니다. 이것은 행 벡터도, 열 벡터도 아닌 애매한 상태라 전치(Transpose)가 안 됩니다.

- **나쁜 예:** 'a = np.random.randn(5)' → 버그 발생 위험 높음.
- **좋은 예:** 'a = np.random.randn(5, 1)' (열 벡터) 또는 '(1, 5)' (행 벡터)로 명시하십시오.
- **습관:** 코드 중간에 'assert(a.shape == (5, 1))'을 넣어 차원을 확인하십시오.

FAQ: 자주 묻는 질문

Q1. GPU는 언제 쓰나요?
A. NumPy는 기본적으로 CPU를 사용합니다. 나중에 배울 TensorFlow나 PyTorch는 이 벡터화 연산을 GPU(그래픽 카드)에서 수행하여, CPU보다 훨씬 더 많은 병렬 처리(수천 개의 코어)를 가능하게 합니다. 원리는 똑같습니다.

Q2. 모든 코드를 벡터화할 수 있나요?
A. 대부분의 수학 연산은 가능합니다. 하지만 복잡한 조건문('if-else')이 데이터마다 다르게 적용되어야 한다면 벡터화가 어려울 수 있습니다. 그럼에도 99%의 딥러닝 연산은 벡터화가 가능합니다.

☒ 다음 단계 (Next Step)

축하합니다! 여러분은 이제 **'고속 연산 엔진(Vectorization)'**을 장착했습니다.

지금까지는 뉴런이 딱 하나(로지스틱 회귀)뿐이었습니다. 다음 장 [Chapter 3. Shallow Neural Networks]에서는 이 강력한 엔진을 활용해 뉴런을 수백 개로 늘려보겠습니다. 이제 진짜 '신경망'다운 신경망을 만들 차례입니다.

[단원 요약 (Cheat Sheet)]

1. **Vectorization:** 'for'문은 죄악이다. 'np.dot' 등을 써서 행렬 단위로 계산하라.
2. **SIMD:** 벡터화는 CPU의 병렬 처리 명령어를 사용하여 속도를 수백 배 높인다.
3. **Broadcasting:** 차원이 달라도 NumPy가 알아서 맞춰주지만, 버그를 조심해야 한다.
4. **Shape Check:** '(n,)' 대신 '(n, 1)'을 사용하여 차원을 명시하는 습관을 들여라.

5 Broadcasting in NumPy

☒ 지난 시간 복습 및 연결

지난 강의에서 우리는 딥러닝 속도의 핵심인 벡터화(Vectorization)를 배웠습니다. 벡터화가 고속도로(Engine)라면, 오늘 배울 브로드캐스팅(Broadcasting)은 차선을 자유자재로 변경하는 유연함(Flexibility)입니다. 많은 학생들이 'np.dot'은 잘 쓰면서도, 모양이 다른 행렬끼리 연산할 때 발생하는 오류에는 속수무책입니다. 이 원리를 알아야 진정한 디버깅 마스터가 될 수 있습니다.

5.1 Overview

[핵심 목표]

이 단원은 서로 다른 모양(Shape)의 데이터를 오류 없이 연산하는 방법을 다룹니다.

- **개념:** NumPy가 작은 배열을 자동으로 확장(Stretch)하여 연산하는 규칙을 배웁니다.
- **구현:** 'for-loop' 없이 데이터 정규화(Normalization)를 수행하는 코드를 작성합니다.
- **원리:** 메모리 복사 없이 스트라이드(Strides) 조작을 통해 효율적으로 동작하는 내부 원리를 이해합니다.

5.1.1 Essential Terminology

용어	설명	한 줄 핵심 요약
Broadcasting	브로드캐스팅	모양이 다른 배열 간 연산 시, 작은 쪽을 자동으로 늘려주는 기능
Shape	형상	배열의 차원 크기 (예: $(4, 3)$은 4행 3열)
Normalization	정규화	데이터의 평균을 0, 분산을 1로 맞추는 전처리 과정
Keepdims	차원 유지	연산 후에도 차원(Rank)을 삭제하지 않고 유지하는 옵션
Strides	스트라이드	메모리 상에서 다음 요소로 넘어가기 위한 보폭 (Byte 단위)

5.1.2 Core Concepts: 브로드캐스팅의 마법

5.1.3 1. Broadcasting의 정의와 비유

한 줄 요약: 작은 행렬을 큰 행렬 크기에 맞춰 자동으로 '늘려서(Copy)' 연산합니다.

[식빵과 버터 비유]

식빵 100개(데이터 $m = 100$)에 버터(b)를 발라야 합니다.

- **For-loop:** 식빵을 하나 꺼내고, 버터를 바르고, 내려놓습니다. (100번 반복)
- **Broadcasting:** 마법을 부려 버터를 식빵 100개 길이만큼 순식간에 늘린(Stretch) 뒤, 한 번에 쾅 찍어버립니다.

5.1.4 2. General Broadcasting Rules (엄격한 규칙)

아무거나 다 늘려주지는 않습니다. NumPy는 뒤(오른쪽) 차원부터 비교하여 다음 조건 중 하나를 만족해야만 연산을 허용합니다.

1. **Equal:** 두 차원의 크기가 같다.
2. **One:** 둘 중 하나의 크기가 1이다. (이 경우 1인 쪽이 늘어남)

[colback=white, colframe=black, title=Rule Check Example] **Case 1: 가능 (Success)**
$A : (4, \mathbf{3})$
$B : (4, \mathbf{1}) \rightarrow$ 1이 3으로 확장됨.
결과: $(4, 3)$
Case 2: 불가능 (Fail - ValueError)
$A : (4, \mathbf{3})$
$B : (4, \mathbf{2}) \rightarrow$ 3과 2는 다르고, 둘 다 1이 아님.

5.1.5 3. Under the Hood: 가상 복사 (Virtual Copying)

"교수님, 데이터를 늘리면 메모리를 낭비하는 것 아닌가요?"
아닙니다. 이것이 브로드캐스팅 기술의 핵심입니다.

- **Physical (실제):** $b = [1, 2, 3]$ (메모리엔 딱 3개만 존재)
- **Logical (가상):** CPU에게는 마치 $[1, 1, 1, \dots], [2, 2, 2, \dots]$ 인 것처럼 주소를 속여서 알려줍니다.
- **Strides Manipulation:** 메모리를 실제로 복사하지 않고, 데이터 접근 보폭(Stride)을 0으로 설정하여 같은 값을 반복해서 읽게 만듭니다. 마치 홀로그램과 같습니다.

5.1.6 Numerical Example: 손으로 푸는 브로드캐스팅

[칼로리 계산 시나리오]
상황: 4가지 음식(행)의 영양소(열: 탄, 단, 지) 데이터가 있습니다. 각 음식의 총 칼로리가 100g당 얼마인지 더하고 싶습니다.
데이터 행렬 A (2개 음식 x 3개 영양소):

$$\begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \end{bmatrix}$$

조미료 B (각 영양소에 추가될 값, 1 x 3):

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

연산 과정 ($A + B$): 행렬 B의 행(Row) 차원이 1이므로, 행렬 A의 크기인 2로 확장됩니다.

$$\begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \end{bmatrix} + \begin{bmatrix} 1 & 2 & 3 \\ \mathbf{1} & \mathbf{2} & \mathbf{3} \leftarrow \text{(복사됨)} \end{bmatrix} = \begin{bmatrix} 11 & 22 & 33 \\ 41 & 52 & 63 \end{bmatrix}$$

5.1.7 Implementation: Data Normalization

브로드캐스팅이 가장 빛을 발하는 순간은 데이터를 전처리(Preprocessing) 할 때입니다. 입력 데이터의 평균을 0, 분산을 1로 만드는 정규화를 구현해봅시다.

Listing 4: Broadcasting Implementation

```
import numpy as np
import time

def normalization_demo():
    # □ □ □ : □ □ 4 □ □ (Feature), □ 100 □ □  □ □
    # Shape: (4, 1000000)
    X = np.random.rand(4, 1000000) * 100

    # 1. □ □ □  □ □ □ □  □ □
    # axis=1: □ (column) □ □ □ □  □ □  □ ( □ □  □ □ )
    # keepdims=True: (4,)□  □ □ □  (4, 1)□  □ □  -> □ □ □ □ □ □ □  □ □  □ □ !
    mu = np.mean(X, axis=1, keepdims=True)
    sigma = np.std(X, axis=1, keepdims=True)

```

```
    print(f"mu␣shape:␣{mu.shape}") # (4, 1) □□

    # 2. Broadcasting □□ □□ ()
    # (4, 1000000) - (4, 1) -> (4, 1)□ □ 100 □ □□□□ □□□
    # □□□□ □□□□ □□
    tic = time.time()
    X_norm = (X - mu) / sigma
    toc = time.time()

    print(f"Broadcasting␣time:␣{1000␣*␣(toc␣-␣tic):.2f}␣ms") # □□ □□

if __name__ == "__main__":
    normalization_demo()
```

5.1.8 FAQ Pitfalls

[주의: (m, 1) + (1, m) = (m, m)]

브로드캐스팅의 강력함이 독이 될 때가 있습니다.

- 벡터 A: $(5, 1)$
- 벡터 B: $(1, 5)$
- $A + B$: $(5, 5)$ 행렬이 되어버립니다.

두 벡터를 더해서 같은 크기의 벡터를 만들고 싶었다면, 반드시 두 벡터의 Shape이 일치하는지 'assert' 문으로 확인해야 합니다.

Q. `keepdims=True`를 안 쓰면 어떻게 되나요?

A. 'mu'의 shape이 '(4,)'가 됩니다. 이를 'Rank-1 Array'라고 합니다. 대부분의 경우 브로드캐스팅이 잘 되지만, 특정 상황에서 예상치 못한 차원 확장이 일어나 디버깅이 매우 어려워집니다. 명시적으로 '(4, 1)'을 유지하는 것이 안전합니다.

☒ 다음 단계 (Next Step)

이제 여러분은 데이터의 모양(Shape)을 자유자재로 다루는 기술까지 익혔습니다. 벡터화와 브로드캐스팅이라는 두 개의 무기를 손에 쥐었습니다.

다음 장 [Chapter 3. Shallow Neural Networks]에서는 드디어 로지스틱 회귀(뉴런 1개)를 넘어서, 은닉층(Hidden Layer)이 있는 진짜 신경망을 구축합니다. 여기서부터 딥러닝의 마법이 시작됩니다.

[단원 요약 (Cheat Sheet)]

1. **Broadcasting:** 작은 배열을 큰 배열에 맞춰 '가상으로 확장'하여 연산한다.
2. **Rule:** 차원을 오른쪽 끝부터 비교하여, 같거나 1이어야 한다.
3. **Memory:** 데이터를 실제로 복사하지 않으므로(Strides 조작) 메모리 효율적이다.
4. **Tip:** 'np.sum'이나 'np.mean' 사용 시 'keepdims=True'를 습관화하라.

6 Shallow Neural Networks

☒ 지난 시간 복습 및 연결

지금까지 우리는 로지스틱 회귀라는 '단일 뉴런(Single Neuron)'을 완벽하게 마스터했습니다. 하지만 뉴런 하나로는 단순한 선형 문제(직선으로 가르는 문제)밖에 해결하지 못합니다. 이제 이 뉴런들을 수직, 수평으로 연결하여 진정한 의미의 신경망을 구축할 시간입니다. 우리가 오늘 다룰 '얕은 신경망(Shallow Neural Network)'은 딥러닝이라는 거대한 마천루를 쌓기 위한 1층 기초 공사와 같습니다.

6.1 Overview

[핵심 목표]

이 단원은 로지스틱 회귀를 확장하여 2-Layer 신경망을 만드는 과정을 다룹니다.

- **개념:** 입력층과 출력층 사이에 있는 '은닉층(Hidden Layer)'의 역할과 정의를 이해합니다.
- **수학:** 층(Layer) 번호와 데이터 샘플 번호를 구분하는 표기법(Notation)을 익힙니다.
- **원리:** 왜 신경망에 비선형 활성화 함수(Tanh, ReLU)가 반드시 필요한지 증명합니다.
- **구현:** 행렬 연산을 통해 입력에서 출력까지 가는 순전파(Forward Propagation)를 구현합니다.

6.1.1 Essential Terminology

딥러닝 수학의 50%는 표기법을 제대로 아는 것에서 시작합니다.

표기	의미	예시 및 설명
$x = a^{[0]}$	입력층 (Input Layer)	원본 데이터. 가중치가 없으므로 0번째 층 취급.
$a^{[1]}$	은닉층 (Hidden Layer)	입력값을 변환하여 특징을 추출하는 중간 단계.
$a^{[2]} = \hat{y}$	출력층 (Output Layer)	최종 예측값 (예: 고양이일 확률).
$[l]$ (대괄호)	**층(Layer) 번호**	$W^{[1]}$ (1번 층의 가중치)
(i) (소괄호)	**데이터 샘플 번호**	$x^{(i)}$ (i번째 훈련 데이터)
$n^{[l]}$	l번째 층의 뉴런 개수	$n^{[1]} = 4$ (은닉층 뉴런이 4개)

6.1.2 Core Concepts: 은닉층의 해부

6.1.3 1. Architecture (구조: 1층에서 2층으로)

로지스틱 회귀가 '입력 $\rightarrow$ 출력'의 직행버스라면, 얕은 신경망은 중간에 환승 센터(은닉층)가 하나 있는 구조입니다.

- 입력층 ($Input$): x_1, x_2, x_3 (데이터 특성)
- 은닉층 ($Hidden$): 입력 정보를 섞고 비틀어서 새로운 정보를 만듭니다.
- 출력층 ($Output$): 은닉층의 정보를 종합하여 최종 결정을 내립니다.

[자동차 공장 조립 라인]

- Input (x): 철판, 유리, 고무 등 원자재.
- Hidden Layer ($a^{[1]}$): 공장 내부의 작업자들. 철판을 구부려 문짝을 만들고, 엔진을 조립합니다. 외부(사용자)에서는 이 과정이 보이지 않으므로 'Hidden(은닉)'이라고 합니다.
- Output Layer ($a^{[2]}$): 완성된 차를 검수하고 "출고 가능(1)" 혹은 "불량(0)" 판정을 내립니다.

6.1.4 2. 행렬 지옥 탈출 (Matrix Dimensions)

신경망 구현에서 가장 많이 틀리는 부분이 행렬의 크기(Dimension)입니다. 아래 표를 보며 반드시 차원을 맞추는 연습을 해야 합니다.

- $n^{[0]} = n_x$: 입력 특성 개수 (예: 3)
- $n^{[1]}$: 은닉층 뉴런 개수 (예: 4)
- m: 데이터 개수 (예: 100)

변수	Shape (행, 열)	암기 공식
$W^{[1]}$	$(n^{[1]}, n^{[0]})$	(은닉 뉴런 수, 입력 특성 수)
$b^{[1]}$	$(n^{[1]}, 1)$	(은닉 뉴런 수, 1)
$Z^{[1]}, A^{[1]}$	$(n^{[1]}, m)$	(은닉 뉴런 수, 데이터 개수)
$W^{[2]}$	$(1, n^{[1]})$	(출력 뉴런 수, 은닉 뉴런 수)

6.1.5 3. 비선형성(Non-linearity)의 필요성

질문: "교수님, 그냥 계산하기 편하게 선형 함수($y = ax + b$)만 계속 쌓으면 안 되나요?"
답변: 절대 안 됩니다. 비선형 활성화 함수(Sigmoid, Tanh, ReLU)가 없다면 신경망은 깊어질 의미가 없습니다.
증명:

$$Output = W_2(W_1x + b_1) + b_2 = (W_2W_1)x + (W_2b_1 + b_2) = W'x + b'$$

선형 함수끼리의 결합은 결국 또 다른 하나의 선형 함수가 됩니다. 100층을 쌓아도 수학적으로는 1층짜리 로지스틱 회귀와 똑같아집니다. 복잡한 곡선을 그리려면 비선형 함수가 필수입니다.

6.1.6 Mathematical Forward Propagation

입력 x가 신경망을 통과하는 과정을 수식으로 정리합니다.
Step 1: 입력 $\rightarrow$ 은닉층 (특징 추출)

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$
$$A^{[1]} = \tanh(Z^{[1]})$$

(참고: 은닉층에서는 Sigmoid보다 평균이 0인 Tanh가 학습 성능이 더 좋습니다.)
Step 2: 은닉층 $\rightarrow$ 출력층 (최종 예측)

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$
$$A^{[2]} = \sigma(Z^{[2]}) \quad (\hat{y})$$

6.1.7 Practical Scenario: 얼굴 인식

[얼굴 인식 AI의 사고 과정]

- Input (x): 이미지의 각 픽셀 밝기값 (단순한 숫자 나열).
- Hidden Layer ($a^{[1]}$): 픽셀들을 조합하여 '선(Line)', '모서리(Edge)', '눈 모양', '코 모양' 같은 특징(Feature)을 찾아냅니다.
- Output Layer ($a^{[2]}$): 찾아낸 눈, 코, 입의 특징을 종합하여 "이것은 철수의 얼굴이다(Probability)"라고 판단합니다.

6.1.8 Implementation (Python with NumPy)

Listing 5: Shallow Neural Network Forward Propagation

```python
import numpy as np

class ShallowNN:
    def __init__(self, n_x, n_h, n_y):
        np.random.seed(1)
        # □□□ □□□: □0 □□ □□ □□ □□□□ □! (Symmetry Breaking)
        self.W1 = np.random.randn(n_h, n_x) * 0.01
        self.b1 = np.zeros((n_h, 1))
        self.W2 = np.random.randn(n_y, n_h) * 0.01
        self.b2 = np.zeros((n_y, 1))

    def forward(self, X):
        """
␣␣␣␣␣␣␣␣X␣shape:␣(n_x,␣m)
␣␣␣␣␣␣␣␣"""
        # --- Layer 1 (Hidden) ---
        # Z1: (n_h, m)
        Z1 = np.dot(self.W1, X) + self.b1
        A1 = np.tanh(Z1) # □□□ □□□ □□ (Tanh)

        # --- Layer 2 (Output) ---
        # Z2: (n_y, m)
        Z2 = np.dot(self.W2, A1) + self.b2
        A2 = 1 / (1 + np.exp(-Z2)) # □□□ □□□ □□ (Sigmoid)

        return A2

# --- □□ □□ ---
if __name__ == "__main__":
    # □□ 3 □□, □□ 4 □□□ □□
    X = np.array([[1, 2, 3, 4],
                  [4, 5, 6, 7],
                  [7, 8, 9, 10]]) # shape (3, 4)

    # □□ (3) -> □□ (4) -> □□ (1)
    model = ShallowNN(n_x=3, n_h=4, n_y=1)
    output = model.forward(X)

    print("Output␣Shape:", output.shape) # (1, 4) □□
    print("Prediction:", output)
```

6.1.9 FAQ: 초심자가 자주 묻는 질문

- **Q1. 가중치 W를 왜 0으로 초기화하면 안 되나요?**
  A. W가 모두 0이면 은닉층의 모든 뉴런이 똑같은 계산을 하게 됩니다(대칭성 문제). 뉴런이 100개여도 사실상 1개인 것과 같습니다. 서로 다른 특징을 배우게 하려면 랜덤하게 깨뜨려야(Break Symmetry) 합니다.
- **Q2. 은닉층 개수는 어떻게 정하나요?**
  A. 하이퍼파라미터입니다. 정답은 없습니다. 문제의 복잡도에 따라 다르며, 실험을 통해 최적의 개수를 찾아야 합니다. 보통 입력 크기보다 약간 크게 잡는 것부터 시작합니다.

☒ 다음 단계 (Next Step)

이제 우리는 신경망의 뼈대를 세우고 신호(데이터)를 앞으로 보내는 법(Forward)을 알았습니다. 하지만 아직 학습은 하지 않았습니다. 다음 시간에는 예측값과 정답 사이의 오차를 구해서, 다시 뒤로 보내며 가중치를 수정하는 역전파(Backpropagatio 대해 다룹니다. 이것이 딥러닝 학습의 진정한 핵심입니다.

[단원 요약 (Cheat Sheet)]

1. **Hidden Layer:** 입력과 출력 사이에서 비선형적 특징을 추출하는 층.

2. **Notation:** $[l]$은 층 번호, (i)는 데이터 번호. 혼동 금지!
3. **Non-linearity:** 활성화 함수(Tanh, ReLU 등)가 없으면 신경망은 단순 선형 회귀와 같다.
4. **Dimension:** $W^{[1]}$의 크기는 $(n^{[1]}, n^{[0]})$이다. (행렬 크기 주의)

7 Activation Functions

☒ 지난 시간 복습 및 연결

지난 시간에 우리는 은닉층(Hidden Layer)을 추가하여 신경망의 깊이를 더했습니다. 그때 제가 "은닉층에는 Sigmoid보다 Tanh나 ReLU가 좋다"고 스쳐 지나가듯 말했습니다. "왜요? Sigmoid가 가장 유명하지 않나요?" 이 질문에 답하지 못하면 여러분은 매번 모델을 설계할 때마다 '선택 장애'에 시달릴 것입니다. 활성화 함수는 단순한 스위치가 아닙니다. 학습 신호(Gradient)를 살릴 수도, 죽일 수도 있는 생명 유지 장치입니다.

7.1 Overview

[핵심 목표]

이 단원은 딥러닝 모델의 성능을 결정짓는 '4대 활성화 함수'를 완벽하게 해부합니다.

- **비교:** Sigmoid, Tanh, ReLU, Leaky ReLU의 수식과 그래프 특징을 비교합니다.
- **원리:** 깊은 신경망에서 Sigmoid를 쓰면 학습이 멈추는 기울기 소실(Vanishing Gradient) 문제를 수학적으로 증명합니다.
- **전략:** 출력층과 은닉층에 각각 어떤 함수를 써야 하는지 Best Practice를 확립합니다.
- **구현:** NumPy를 사용하여 각 함수와 그 도함수(Derivative)를 효율적으로 코딩합니다.

7.1.1 Essential Terminology

함수명	범위	한 줄 특징
Sigmoid	$(0, 1)$	확률 표현에 최적. 하지만 깊어지면 학습 불가.
Tanh	$(-1, 1)$	Sigmoid의 확장판. 0 중심(Zero-centered)이라 학습이 더 빠름.
ReLU	$[0, \infty)$	**딥러닝의 표준.** 양수는 그대로, 음수는 차단. 연산 빠름.
Leaky ReLU	$(-\infty, \infty)$	ReLU의 변형. 음수일 때도 아주 약간의 기울기를 줌.

7.1.2 Core Concepts: The Big 4 Functions

7.1.3 1. Sigmoid Function (σ)

[Image of sigmoid function graph with equation]

- **수식:** $a = \frac{1}{1+e^{-z}}$
- **특징:** 0과 1 사이의 값으로 압축합니다. '확률' 개념과 잘 맞습니다.
- **치명적 단점:** 입력값(z)이 아주 크거나 작으면 기울기(미분값)가 0에 가까워집니다. 학습이 멈춥니다.
- **용도: 이진 분류의 출력층(Output Layer)**에만 씁니다. 은닉층엔 절대 쓰지 마세요.

7.1.4 2. Tanh (Hyperbolic Tangent)

- **수식:** $a = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$
- **특징:** Sigmoid를 위아래로 늘려 -1에서 1 사이 값을 갖게 했습니다. 평균이 0(Zero-centered)이므로 데이터의 중심을 잘 잡아주어 Sigmoid보다 학습 수렴이 빠릅니다.
- **용도:** 은닉층에서 Sigmoid보다 무조건 좋습니다. 하지만 여전히 기울기 소실 문제는 있습니다.

7.1.5 3. ReLU (Rectified Linear Unit) - The King

- **수식:** $a = \max(0, z)$
- **특징:** 단순 무식해 보이지만 가장 강력합니다.
  - $z > 0$: 기울기가 항상 1입니다. (신호가 약해지지 않음)
  - $z \leq 0$: 값을 0으로 차단합니다. (불필요한 신호 제거)
- **용도: 모든 은닉층의 기본값(Default)**입니다. 고민될 땐 무조건 ReLU를 쓰세요.

[전등 스위치 비유]

- **Sigmoid:** 조광기(Dimmer). 밝기를 0%에서 100%까지 미세하게 조절하지만, 너무 복잡합니다.
- **ReLU:** 똑딱 스위치. 켜지면 확실하게 켜지고(그대로 통과), 꺼지면 확실하게 꺼집니다(0). 단순함이 속도의 비결입니다.

7.1.6 Deep Dive: 왜 Sigmoid는 퇴출당했나?

이 부분은 딥러닝 역사에서 가장 중요한 전환점 중 하나인 기울기 소실 문제(Vanishing Gradient Problem)를 다룹니다.

7.1.7 The Vanishing Gradient Problem

역전파(Backpropagation)는 출력층의 오차를 입력층까지 전달하기 위해 미분값(기울기)을 계속 곱하는(Chain Rule) 과정입니다.

[수학적 증명: 0.25 vs 1.0]

Sigmoid 함수의 미분 최댓값은 $z = 0$일 때 0.25입니다. 만약 은닉층이 10개라고 가정해봅시다.

Case 1: Sigmoid 사용

$$Gradient \approx 0.25 \times 0.25 \times \cdots \times 0.25 = (0.25)^{10} \approx 0.0000009$$

$\rightarrow$ 입력층에 도달할 때쯤 기울기는 0이 되어 사라집니다. 앞단은 학습이 전혀 안 됩니다.

Case 2: ReLU 사용 (양수 구간)

$$Gradient = 1 \times 1 \times \cdots \times 1 = 1^{10} = 1$$

$\rightarrow$ 기울기가 줄어들지 않고 생생하게 입력층까지 전달됩니다. 이것이 100층짜리 딥러닝이 가능한 이유입니다.

7.1.8 Professor's Cheat Sheet (Best Practice)

실전에서 무엇을 쓸지 고민하지 마십시오. 이 규칙을 따르면 상위 10%입니다.

[colback=white, colframe=black, title=활성화 함수 선택 가이드]

- **출력층 (Output Layer):**
  - 이진 분류 (0 or 1): **Sigmoid**
  - 다중 분류 (Cat, Dog, Bird...): **Softmax**
  - 회귀 (집값 예측): **Linear** (활성화 함수 없음)
- **은닉층 (Hidden Layer):**
  - **기본 (Default): ReLU**
  - ReLU 성능이 아쉽거나 뉴런이 죽는 경우: **Leaky ReLU**
  - 데이터가 매우 적고 모델이 얕을 때: **Tanh**
  - **금지: Sigmoid** (절대 사용 금지)

7.1.9 Implementation (Python with NumPy)

함수값뿐만 아니라 역전파에 필요한 도함수(Derivative)까지 구현합니다.

Listing 6: Activation Functions Derivatives

```python
import numpy as np

class Activations:
    @staticmethod
    def sigmoid(z):
        """□ □ □ □ :␣0␣~␣1"""
        return 1 / (1 + np.exp(-z))

    @staticmethod
    def sigmoid_derivative(z):
        """Sigmoid␣□ □ :␣a␣*␣(1-a)"""
        s = 1 / (1 + np.exp(-z))
        return s * (1 - s)

    @staticmethod
    def relu(z):
        """□ □ □ □ :␣max(0,␣z)"""
        return np.maximum(0, z)

    @staticmethod
    def relu_derivative(z):
        """
␣␣␣␣␣␣␣␣ReLU␣□ □ :
␣␣␣␣␣␣␣␣z␣>␣0␣□ □ ␣1,␣z␣<=␣0␣□ □ ␣0
␣␣␣␣␣␣␣␣"""
        dZ = np.array(z, copy=True) # □ □  □ □
        dZ[z <= 0] = 0
        dZ[z > 0] = 1
        return dZ

    @staticmethod
    def tanh(z):
        """□ □ □ □ :␣-1␣~␣1"""
        return np.tanh(z) # NumPy □ □ □  □ □  □ □

    @staticmethod
    def tanh_derivative(z):
        """Tanh␣□ □ :␣1␣-␣a^2"""
        return 1 - np.power(np.tanh(z), 2)
```

7.1.10 FAQ Pitfalls

[ReLU의 미분 불가능 점 ($z = 0$)]

Q. 수학적으로 $z = 0$에서 ReLU는 미분이 불가능한데(뾰족점), 코딩은 어떻게 하나요?

A. 맞습니다. 하지만 컴퓨터 공학에서는 실용적으로 접근합니다. $z = 0$일 때 기울기를 그냥 0이나 1 중 하나로 정해버립니다. (보통 0으로 둠). z가 정확히 0.0000...이 될 확률은 매우 낮으므로 학습에 아무런 지장이 없습니다.

Q. Leaky ReLU는 언제 쓰나요?

A. ReLU를 썼는데 학습 중에 뉴런의 출력이 계속 0만 나와서 죽어버리는 현상(**Dying ReLU**)이 발생할 때 씁니다. 음수일 때 0.01 같은 작은 기울기를 주어 뉴런을 소생시킵니다.

☒ 다음 단계 (Next Step)

이제 우리는 뉴런의 구조(Layer)와 신호를 조절하는 스위치(Activation)까지 모두 갖췄습니다. 자동차로 치면 엔진과 변속기를 조립한 상태입니다.

다음 시간에는 이 자동차를 실제로 달리게 만드는 엔진 점화 과정, 즉 오차를 줄이기 위해 미분을 사용하는 '역전파(Backpropagation)'의 수식적 유도 과정을 아주 깊이 있게 파헤쳐 보겠습니다. 긴장하십시오. 이제 진짜 미분의 숲으로 들어갑니다.

[단원 요약 (Cheat Sheet)]

1. **Sigmoid:** 출력층(이진 분류)에만 사용. 은닉층 사용 시 기울기 소실 발생.
2. **ReLU:** 은닉층의 **Default**. 양수는 그대로(기울기 1), 음수는 0. 연산 빠름.
3. **Tanh:** Sigmoid보다 좋음(Zero-centered). 얕은 모델에 적합.
4. **Vanishing Gradient:** Sigmoid 미분값이 1보다 작아(≤ 0.25), 층이 깊어지면 학습 신호가 사라지는 현상.

8 Backpropagation

☒ 지난 시간 복습 및 연결

지난 시간에 우리는 신경망의 '구조(Layer)'와 '스위치(Activation Function)'를 장착했습니다. 이제 자동차는 완성되었습니다. 하지만 자동차를 앞으로 달리게만 해서는 운전을 배울 수 없습니다. 사고를 냈을 때(오차가 발생했을 때), 무엇이 잘못되었는지 파악하고 핸들을 돌리는 법(수정하는 법)을 배워야 합니다. 오늘 배울 역전파(Backpropagation)가 바로 그 과정입니다. 수학 기호가 쏟아지겠지만, 포기하지 마십시오. 이것이 딥러닝의 심장입니다.

8.1 Overview

[핵심 목표]

이 단원은 딥러닝 학습 메커니즘인 '순전파'와 '역전파'를 수식과 코드로 구현합니다.

- **순전파 (Forward):** 입력 X가 은닉층을 거쳐 출력 $\hat{y}$가 되는 과정을 행렬로 정의합니다.
- **역전파 (Backward):** 예측이 틀렸을 때, 비용 함수(Cost)의 기울기(Gradient)를 뒤쪽에서 앞쪽으로 계산합니다.
- **도구:** 미적분의 연쇄 법칙(Chain Rule)과 행렬의 전치(Transpose)가 왜 필요한지 이해합니다.
- **구현:** 차원(Dimension) 오류 없이 역전파 알고리즘을 코딩합니다.

8.1.1 Essential Terminology

기호	의미	비유 (역할)
$Z^{[l]}$	선형 출력 ($WX+b$)	뉴런이 받아들인 원시 점수
$A^{[l]}$	활성화 출력 ($g(Z)$)	점수를 확률/신호로 변환한 최종 리포트
$dZ^{[l]}$	오차항 ($\partial J/\partial Z$)	"얼마나 틀렸니?" (책임의 크기)
$dW^{[l]}$	가중치 기울기	"가중치를 얼마나 수정할까?"
$*$	**요소별 곱 (Element-wise)**	행렬 곱이 아니라, 같은 위치끼리 곱함

8.1.2 Core Concepts: 흐름의 이해

8.1.3 1. Forward Propagation (예측의 흐름)

데이터가 강물처럼 입력층에서 출력층으로 흐릅니다.

$$Input(X) \xrightarrow{W^{[1]},b^{[1]}} Hidden(A^{[1]}) \xrightarrow{W^{[2]},b^{[2]}} Output(A^{[2]})$$

이 과정은 직관적입니다. "입력받아서, 계산하고, 넘겨준다." 끝입니다.

8.1.4 2. Backward Propagation (학습의 흐름)

예측값($A^{[2]}$)과 실제값(Y)의 차이, 즉 비용(Cost)을 줄이기 위해 미분을 사용합니다. 문제는 우리가 수정하고 싶은 파라미터($W^{[1]}$)가 출력층에서 멀리 떨어져 있다는 것입니다.

[프로젝트 실패의 책임 소재 따지기 (The Blame Game)]

여러분이 팀장(출력층)이고 프로젝트가 실패(Error)했다고 가정해봅시다.

1. Step 1 (Output Layer): 먼저 최종 결과물($A^{[2]}$)을 보고 "얼마나 부족했는지($dZ^{[2]}$)" 파악합니다.

2. Step 2 (Hidden Layer): 팀장은 자신의 실패 원인을 분석하여, 중간 관리자(은닉층, $A^{[1]}$)에게 책임을 묻습니다. "네가 준 보고서가 잘못돼서 결과가 이렇게 됐잖아!" ($dZ^{[1]}$ 전파)
3. Step 3 (Parameters): 중간 관리자는 다시 자신의 업무 도구($W^{[1]}$)를 탓하며 수정합니다. "이 가중치가 문제였군, 고치자." ($dW^{[1]}$ 계산)

역전파는 이처럼 **오차(책임)를 뒤에서 앞으로 전달하며** 파라미터를 수정하는 과정입니다.

8.1.5 Deep Dive: The 6 Magic Equations

이 6개의 수식은 딥러닝 엔지니어의 구구단입니다. 연쇄 법칙(Chain Rule)에 의해 유도됩니다.

8.1.6 Phase 1: 출력층 (Layer 2) - 역전파의 시작

1. 오차 계산 ($dZ^{[2]}$) 가장 직관적인 수식입니다. 예측값과 정답의 차이입니다.

$$dZ^{[2]} = A^{[2]} - Y$$

(참고: Cross-Entropy와 Sigmoid 미분이 만나면 이렇게 깔끔하게 정리됩니다.)

2. 가중치 기울기 ($dW^{[2]}$) 오차($dZ^{[2]}$)에 입력값($A^{[1]}$)을 곱합니다.

$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$$

[왜 전치(T)를 하나요?]
행렬 곱셈의 차원을 맞추기 위해서입니다. $dZ^{[2]}$는 $(1, m)$, $A^{[1]}$은 $(n^{[1]}, m)$입니다. 곱하려면 $A^{[1]}$을 뒤집어야 $(1, m) \times (m, n^{[1]}) = (1, n^{[1]})$이 되어 $W^{[2]}$와 크기가 같아집니다.

3. 편향 기울기 ($db^{[2]}$) 오차들의 평균입니다.

$$db^{[2]} = \frac{1}{m} \sum_{rows} dZ^{[2]}$$

8.1.7 Phase 2: 은닉층 (Layer 1) - 핵심 구간

4. 은닉층 오차 ($dZ^{[1]}$) 여기가 가장 어렵습니다. 출력층의 오차를 가중치 비율만큼 가져오고(Linear), 활성화 함수의 미분값(Non-linear)을 곱합니다.

$$dZ^{[1]} = \underbrace{W^{[2]T} dZ^{[2]}}_{\text{오차 전파}} \underbrace{*}_{\text{요소별 곱}} \underbrace{g'^{[1]}(Z^{[1]})}_{\text{활성화 미분}}$$

- $W^{[2]T} dZ^{[2]}$: 출력층의 오차를 은닉층으로 역송신합니다.
- $*$: 행렬 곱이 아닙니다! Element-wise product입니다.
- $g'(Z^{[1]})$: 만약 Tanh를 썼다면 $(1 - A^{[1]2})$입니다.

5, 6. 파라미터 기울기 ($dW^{[1]}, db^{[1]}$) Layer 2와 동일한 패턴입니다.

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$$

$$db^{[1]} = \frac{1}{m} \sum dZ^{[1]}$$

8.1.8 Implementation (Python with NumPy)

수식을 코드로 옮길 때 가장 중요한 것은 차원(Shape) 확인입니다.

Listing 7: Full Backpropagation Implementation

```
import numpy as np

def backward_propagation(parameters, cache, X, Y):
    """
␣␣␣␣parameters:␣W1,␣b1,␣W2,␣b2
␣␣␣␣cache:␣Z1,␣A1,␣Z2,␣A2␣(Forward␣ □ □ □ ␣ □ □ □ ␣ □ )
␣␣␣␣X,␣Y:␣□ □ ␣□ □ □ ␣□ ␣□ □
␣␣␣␣"""
    m = X.shape[1] # □ □ □  □ □

    # 1. □ □ □ □  □  □ □  □ □
    W2 = parameters["W2"]
    A1 = cache["A1"]
    A2 = cache["A2"]

    # --- Layer 2 (Output) ---
    # □ □  1: dZ2 = A2 - Y
    dZ2 = A2 - Y

    # □ □  2: dW2 □ □ ( □  □ □ : dZ2 @ A1.T)
    dW2 = (1 / m) * np.dot(dZ2, A1.T)

    # □ □  3: db2 □ ( □ □  □ □ , keepdims □ □ )
    db2 = (1 / m) * np.sum(dZ2, axis=1, keepdims=True)

    # --- Layer 1 (Hidden) ---
    # □ □  4: dZ1 □ □  □ □ ( □ □ !)
    # g'(z) for Tanh = 1 - a^2
    # '*' □ □ □ □  □ □ □  □ (Element-wise)□ □  □ □
    dZ1 = np.dot(W2.T, dZ2) * (1 - np.power(A1, 2))

    # □ □  5: dW1
    dW1 = (1 / m) * np.dot(dZ1, X.T)

    # □ □  6: db1
    db1 = (1 / m) * np.sum(dZ1, axis=1, keepdims=True)

    grads = {"dW1": dW1, "db1": db1, "dW2": dW2, "db2": db2}
    return grads
```

8.1.9 Numerical Example: 계산 흐름 추적

[간단한 오차 역전파 예시]

상황: 정답 $y = 1$인데, 모델이 예측 $a^{[2]} = 0.8$을 내놓았습니다.

1. **오차 발생 ($dZ^{[2]}$):** $0.8 - 1.0 = -0.2$. (0.2만큼 부족함)
2. **은닉층 전달:** $W^{[2]}$가 0.5라고 가정합시다. 은닉층으로 오차를 보냅니다.

$$\text{전달된 오차} \approx 0.5 \times (-0.2) = -0.1$$

3. **활성화 미분 반영:** 만약 은닉층 활성화 미분값이 0.5라면?

$$dZ^{[1]} = -0.1 \times 0.5 = -0.05$$

4. **결론:** 은닉층의 오차는 -0.05입니다. 이 값을 줄이는 방향으로 $W^{[1]}$을 업데이트합니다.

8.1.10 FAQ Pitfalls

Q1. $dZ^{[1]}$ 구할 때 왜 행렬 곱(dot)이 아니라 요소별 곱(*) 인가요?
A. 연쇄 법칙 $\frac{\partial A}{\partial Z}$ 부분 때문입니다. 활성화 함수의 미분은 각 뉴런마다 개별적으로 적용됩니다. 행렬 전체를 섞는(Linear mixing) 과정이 아니므로 같은 위치의 원소끼리만 곱해야 합니다.

Q2. 전치(T)는 언제 하나요? 외워야 하나요?
A. 외우지 마세요. 차원(Dimensions)을 그려보면 됩니다. dW는 W와 모양이 같아야 합니다. (n, m)과 $(1, m)$을 곱해서 $(n, 1)$을 만들려면 뒤의 것을 뒤집어야 한다는 것이 자연스럽게 보입니다.

☒ 다음 단계 (Next Step)

고생하셨습니다. 여러분은 방금 딥러닝에서 가장 험난한 고개인 '역전파'를 넘었습니다. 이제 모델을 학습시킬 준비가 거의 다 되었습니다.

그런데, 학습을 시작할 때 가중치(W)를 처음에 어떻게 설정하느냐가 학습의 성패를 좌우한다는 사실을 아십니까? 다음 시간에는 [Practice] 세션으로, 랜덤 초기화(Random Initialization)의 중요성을 다루고, 왜 0으로 초기화하면 이 모든 역전파 알고리즘이 무용지물이 되는지 증명하겠습니다.

[단원 요약 (Cheat Sheet)]

1. **역전파:** 출력층의 오차(dZ)를 구해 입력층 방향으로 전파하며 기울기(dW)를 구한다.
2. **Chain Rule:** 층을 건너갈 때마다 미분값을 곱한다 (미분의 연쇄).
3. **Transpose:** 행렬 곱셈 시 차원을 맞추기 위해 전치 행렬(A^T)을 사용한다.
4. **Element-wise:** 활성화 함수의 미분값은 반드시 요소별 곱($*$)으로 계산한다.

9 Deep Neural Networks

☒ 지난 시간 복습 및 연결

우리는 지금까지 은닉층이 하나뿐인 '얕은 신경망'을 다뤘습니다. 하지만 현실 세계의 복잡한 문제(자율주행, 자연어 처리 등)를 풀기엔 뇌 용량이 부족합니다. 이제 우리는 은닉층을 2개, 3개, 아니 수백 개까지 쌓아 올릴 것입니다. 이것이 바로 여러분이 매일 듣는 '딥러닝(Deep Learning)'의 실체입니다. 단순히 층만 늘리는 게 아니라, 코드를 일반화(Generalization)하여 어떤 깊이의 모델도 만들 수 있는 건축가가 되어 봅시다.

9.1 Overview

[핵심 목표]

이 단원은 L개의 층을 가진 일반화된 심층 신경망(Deep MLP)을 설계하고 구현합니다.

- **표기법:** 층의 개수가 L개일 때의 파라미터($W^{[l]}, b^{[l]}$)와 활성화값($A^{[l]}$)을 정의합니다.
- **원리:** 딥러닝이 데이터를 계층적(Hierarchical)으로 이해하는 방식(점 $\rightarrow$ 선 $\rightarrow$ 면)을 배웁니다.
- **차원:** 각 층의 뉴런 개수($n^{[l]}$)만 보고도 가중치 행렬의 크기를 즉시 계산해냅니다.
- **구현:** 하드코딩(W1, W2...)을 버리고, 'for-loop'와 'Dictionary'를 이용해 유연한 코드를 작성합니다.

9.1.1 Essential Terminology: The Deep Notation

얕은 신경망에서 쓰던 표기법을 확장합니다. l은 현재 층 번호를 의미합니다.

기호	의미	설명
L	전체 층 수	입력층(0번)을 제외한 층의 개수.
$n^{[l]}$	l번째 층의 뉴런 수	$n^{[0]} = n_x$ (입력), $n^{[L]}$ (출력).
$g^{[l]}$	l번째 층의 활성화 함수	보통 은닉층은 ReLU, 출력층은 Sigmoid.
$A^{[l]}$	l번째 층의 출력	$A^{[l]} = g^{[l]}(Z^{[l]})$. 다음 층의 입력이 됨.

9.1.2 Core Concepts: 왜 깊게 쌓는가?

9.1.3 1. Hierarchical Representation (계층적 표현)

"교수님, 그냥 은닉층 1개에 뉴런 100만 개를 넣는 게(Wide), 10만 개씩 10층 쌓는 것(Deep)보다 낫지 않나요?"
아닙니다. 딥러닝의 힘은 '쪼개서 이해하기'에서 나옵니다.

[사람의 얼굴 인식 과정]

우리의 뇌나 딥러닝 모델은 복잡한 이미지를 한 번에 이해하지 않습니다.

1. **Layer 1 (Low-level):** 픽셀을 보고 가로선, 세로선 같은 경계(Edges)를 찾습니다.
2. **Layer 2 (Mid-level):** 선들을 조합해서 눈, 코, 귀 같은 부분(Parts)을 만듭니다.
3. **Layer 3 (High-level):** 부분들을 조합해서 사람 얼굴(Face) 전체를 인식합니다.

층을 깊게 쌓으면, 적은 파라미터로도 매우 복잡한 함수(사람 얼굴 등)를 효율적으로 표현할 수 있습니다.

9.1.4 2. Matrix Dimensions (차원 분석)

이 부분은 구현과 디버깅의 핵심입니다. 무조건 암기해야 합니다.
l번째 층의 가중치 $W^{[l]}$와 편향 $b^{[l]}$의 크기는 다음과 같습니다.

- $W^{[l]}$ **Shape:** $(n^{[l]}, n^{[l-1]}) \rightarrow$ (현재 층 뉴런 수, 이전 층 뉴런 수)
- $b^{[l]}$ **Shape:** $(n^{[l]}, 1)$
- $Z^{[l]}, A^{[l]}$ **Shape:** $(n^{[l]}, m) \rightarrow$ (현재 층 뉴런 수, 데이터 개수)

[차원 계산 퀴즈]

상황: 입력 특성 $n_x = 12288$ (이미지). Layer 1 뉴런: 20개. Layer 2 뉴런: 7개.
질문: $W^{[1]}$과 $W^{[2]}$의 크기는?

- $W^{[1]}$: $(n^{[1]}, n^{[0]}) = (20, 12288)$
- $W^{[2]}$: $(n^{[2]}, n^{[1]}) = (7, 20)$

9.1.5 Implementation: Building Deep Network

이제 L개의 층을 가진 신경망을 만듭니다. 'W1', 'W2' 변수를 따로 만들지 않고 'parameters['W' + str(l)]' 형태로 관리하는 것이 핵심입니다.

Listing 8: L-Layer Deep Neural Network Initialization Forward

```
import numpy as np

class DeepNN:
    def __init__(self, layer_dims):
        """
␣␣␣␣␣␣␣␣layer_dims:␣□ ␣□ □ ␣□ □ ␣□ □ ␣□ □ ␣□ □ □
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣□ :␣[12288,␣20,␣7,␣5,␣1]␣(4-Layer␣Network)
␣␣␣␣␣␣␣␣"""
        self.params = {}
        self.L = len(layer_dims) - 1 # □ □ □  □ □ □  □  □

        for l in range(1, self.L + 1):
            # He Initialization (ReLU □ □  □  □ □ !)
            # 0.01 □ □  np.sqrt(2 / □ □  □  □ □  □ )□  □ □
            self.params['W' + str(l)] = np.random.randn(layer_dims[l], layer_dims[l-1]) * np.sqrt(2/layer_dims[l-1])
            self.params['b' + str(l)] = np.zeros((layer_dims[l], 1))

            # □ □  □ □  □ □ □ (!)
            assert(self.params['W' + str(l)].shape == (layer_dims[l], layer_dims[l-1]))

    def forward(self, X):
        """
␣␣␣␣␣␣␣␣[Linear␣->␣ReLU]␣*␣(L-1)␣->␣[Linear␣->␣Sigmoid]␣*␣1
␣␣␣␣␣␣␣␣"""
        caches = []
        A = X
        L = self.L

        # 1. □ □ □  (1 ~ L-1): ReLU
        for l in range(1, L):
            A_prev = A
            W = self.params['W' + str(l)]
            b = self.params['b' + str(l)]

            # Linear
            Z = np.dot(W, A_prev) + b
            # Activation (ReLU)
            A = np.maximum(0, Z)

            # □ □ □ □  □ □  □ □  (W, b, A_prev, Z)
```

```
            caches.append((A_prev, W, b, Z))

        # 2. □ □ □  (L): Sigmoid □ □ ( □ □ )
        W = self.params['W' + str(L)]
        b = self.params['b' + str(L)]

        Z = np.dot(W, A) + b
        AL = 1 / (1 + np.exp(-Z)) # Sigmoid
        caches.append((A, W, b, Z))

        return AL, caches

# --- □ □  □ □  ---
if __name__ == "__main__":
    # □ □ [(3) -> □ □ (5) -> □ □ (3) -> □ □ (1)] □ □
    layers = [3, 5, 3, 1]
    model = DeepNN(layers)

    # □ □ □  □ □ □  (3 features, 4 samples)
    X = np.random.randn(3, 4)

    AL, _ = model.forward(X)
    print("Output Shape:", AL.shape) # (1, 4) □ □
```

9.1.6 FAQ Pitfalls

[파라미터 초기화: 0.01 vs He Initialization]

얕은 신경망에서는 '* 0.01'로 초기화해도 괜찮았습니다. 하지만 층이 깊어지면($L > 5$), 값이 계속 곱해지면서 신호가 사라지거나 폭발합니다(Vanishing/Exploding Gradient). 따라서 ReLU를 쓸 때는 He Initialization ('np.sqrt(2/n)')을 쓰는 것이 딥러닝의 표준(Standard)입니다.

Q. Cache 리스트는 왜 만드나요?

A. 순전파(Forward)가 끝나면 바로 역전파(Backward)를 해야 합니다. 역전파 수식을 보면 Z, A_{prev}, W 값이 필요합니다. 이미 계산한 값을 버리지 않고 'caches'에 저장해두면, 다시 계산할 필요 없이 효율적으로 역전파를 수행할 수 있습니다.

☒ 다음 단계 (Next Step)

이제 여러분은 어떤 깊이, 어떤 구조의 신경망도 만들 수 있는 설계 능력을 갖췄습니다. 'layers' 리스트에 숫자만 바꿔 넣으면 됩니다.

하지만 깊은 신경망을 학습시키는 것은 생각보다 까다롭습니다. 다음 시간에는 이 모델을 가지고 '고양이 vs 개' 이미지를 분류하는 실제 프로젝트를 수행하며, 학습 과정에서 발생하는 다양한 문제들을 해결해 보겠습니다.

[단원 요약 (Cheat Sheet)]

1. **Deep Learning:** 은닉층을 여러 개 쌓아 계층적 특징(Hierarchical Features)을 학습한다.
2. **Dimensions:** $W^{[l]}$의 크기는 $(n^{[l]}, n^{[l-1]})$이다. (현재 층, 이전 층)
3. **Implementation:** 'for-loop'를 사용하여 L번 반복하는 일반화된 코드를 작성한다.
4. **Initialization:** 깊은 망에서는 He Initialization을 사용하여 학습 불안정을 막는다.

10 Matrix Dimensions Initialization

☒ 지난 시간 복습 및 연결

우리는 이제 거대한 심층 신경망을 설계할 수 있는 건축가가 되었습니다. 하지만 설계도만 그렸을 뿐, 아직 착공도 하지 않았습니다. 건물을 올리기 전에 가장 먼저 해야 할 일은 무엇일까요? 설계도 검증(차원 확인)과 기초 공사(초기화)입니다. 이 두 가지를 소홀히 하면 코드를 실행하자마자 에러가 터지거나(Dimension Mismatch), 에러 메시지 하나 없이 학습이 전혀 안 되는(Bad Initialization) 침묵의 버그를 만나게 됩니다.

10.1 Overview

[핵심 목표]

이 단원은 디버깅 시간을 획기적으로 줄여주는 '차원 분석'과 학습 성공의 열쇠인 '파라미터 초기화'를 다룹니다.

- **분석:** L층 신경망의 파라미터(W, b)와 데이터(Z, A)의 형상(Shape)을 정확히 도출합니다.
- **이유:** 가중치를 0으로 초기화했을 때 발생하는 '대칭성 문제(Symmetry Problem)'를 증명합니다.
- **해결:** ReLU를 위한 표준 초기화 방법인 He Initialization의 원리와 코드를 익힙니다.

10.1.1 Essential Terminology

용어	설명	핵심 포인트
Shape	행렬의 차원 (행, 열)	디버깅의 90%는 Shape 맞추기입니다.
Symmetry Breaking	대칭성 파괴	뉴런들이 서로 다르게 학습되도록 초기값을 다르게 주는 것.
Zero Init	0으로 초기화	모든 뉴런이 똑같이 동작하게 만드는 **최악의 방법**.
He Init	He 초기화	ReLU 사용 시 분산을 유지해주는 **최고의 방법**.
Xavier Init	Xavier 초기화	Sigmoid/Tanh 사용 시 적합한 초기화 방법.

10.1.2 Core Concepts: 디버깅을 위한 헌법

10.1.3 1. Matrix Dimensions Rules (차원의 법칙)

코딩하다 헷갈릴 때마다 이 표를 보십시오. $n^{[l]}$은 현재 층의 뉴런 수, m은 데이터 개수입니다.

[colback=white, colframe=black, title=Shape Cheat Sheet]

- **파라미터 (학습 대상):**
  - $W^{[l]}$: $(n^{[l]}, n^{[l-1]}) \rightarrow$ (현재 층, 이전 층)
  - $b^{[l]}$: $(n^{[l]}, 1) \rightarrow$ 열 벡터 (Column Vector)
- **데이터 흐름 (Activations):**
  - $Z^{[l]}, A^{[l]}$: $(n^{[l]}, m)$
  - $dZ^{[l]}, dA^{[l]}$: $(n^{[l]}, m) \rightarrow$ 원래 데이터와 Shape 동일

10.1.4 2. Why Not Zero Initialization? (0 초기화의 저주)

"교수님, 로지스틱 회귀에선 0으로 해도 잘 됐잖아요?" 네, 하지만 신경망(은닉층이 있는 경우)에서는 절대 안 됩니다.
[복제 인간 군대 비유]

- **상황:** 모든 가중치 W를 0으로 초기화했습니다.
- **Forward:** 모든 은닉 뉴런이 입력값에 상관없이 똑같은 값(0)을 계산합니다.
- **Backward:** 모든 뉴런이 똑같은 오차(Gradient)를 보고받습니다.
- **Update:** 모든 뉴런이 똑같은 값으로 수정됩니다.
- **결과:** 뉴런이 100만 개여도, 결국 **뉴런 1개짜리 선형 모델**과 똑같이 행동합니다. 이를 대칭성(Symmetry) 문제라고 하며, 학습이 실패합니다.

10.1.5 3. Best Practice: He Initialization

그렇다면 랜덤하게(Random) 초기화하면 될까요? 너무 크면(x10) 기울기 소실이 오고, 너무 작으면(x0.0001) 신호가 죽어버립니다.

Kaiming He 박사가 제안한 He Initialization은 ReLU를 사용할 때 분산을 일정하게 유지해주는 마법의 공식입니다.

$$W^{[l]} \sim \text{Random} \times \sqrt{\frac{2}{n^{[l-1]}}}$$

- 이전 층의 뉴런 개수($n^{[l-1]}$)가 많을수록, 가중치를 더 작게 만들어 줍니다.
- $\sqrt{2}$는 ReLU가 음수 영역을 0으로 만들어 분산을 절반으로 깎아먹는 것을 보상해줍니다.

10.1.6 Implementation: Initialization Strategies

나쁜 예(Zero, Large Random)와 좋은 예(He)를 코드로 비교해봅시다.

Listing 9: Parameter Initialization Methods

```
import numpy as np

class Initializer:
    def __init__(self, layer_dims):
        self.layer_dims = layer_dims # □ : [1000, 100, 10]
        self.L = len(layer_dims) - 1

    def init_zeros(self):
        """
␣␣␣␣␣␣␣␣BAD:␣□ □ ␣□ □ □ □ ␣□ □ 0␣□ □ □ ␣->␣□ □ ␣□ □
␣␣␣␣␣␣␣␣"""
        params = {}
        for l in range(1, self.L + 1):
            params['W' + str(l)] = np.zeros((self.layer_dims[l], self.layer_dims[l-1]))
            params['b' + str(l)] = np.zeros((self.layer_dims[l], 1))
        return params

    def init_he(self):
        """
␣␣␣␣␣␣␣␣BEST:␣He␣Initialization␣(Standard␣for␣ReLU)
␣␣␣␣␣␣␣␣"""
        params = {}
        for l in range(1, self.L + 1):
            # 1. □ □  □ □
            n_curr = self.layer_dims[l]
            n_prev = self.layer_dims[l-1]

            # 2. He Initialization □ □  □ □
```

```
            # np.random.randn: □ □  0, □ □  □ 1 □ □ □ □
            # scaling: □ □ □  2/n_prev □  □ □ □
            scaling = np.sqrt(2 / n_prev)

            params['W' + str(l)] = np.random.randn(n_curr, n_prev) * scaling
            params['b' + str(l)] = np.zeros((n_curr, 1)) # □ □ □  □ □ □ 0 □ !

        return params

# --- □ □  ---
if __name__ == "__main__":
    dims = [1000, 100, 10]
    init = Initializer(dims)

    # He Init □ □  □ □
    params = init.init_he()
    W1 = params['W1']

    print("Shape Check:", W1.shape) # (100, 1000)
    print("Variance Check:", np.var(W1))
    print("Expected Variance:", 2/1000) # 0.002 □ □ □ □  □
```

10.1.7 FAQ Pitfalls

[편향(Bias) b는 0으로 해도 되나요?]

네, 됩니다! 대칭성 문제는 가중치 W에서 발생합니다. W가 이미 랜덤하게 섞여 있다면(Symmetry Broken), 편향 b가 모두 0이어도 뉴런들은 서로 다른 값을 출력하게 됩니다. 따라서 b는 편의상 'np.zeros'로 초기화하는 것이 일반적입니다.

Q. Xavier 초기화는 뭔가요?

A. Sigmoid나 Tanh 함수를 쓸 때 사용하는 초기화 방법입니다. 계수가 $\sqrt{1/n}$입니다. 하지만 요즘 딥러닝은 대부분 ReLU를 쓰기 때문에 He 초기화($\sqrt{2/n}$)가 더 많이 쓰입니다.

☒ 다음 단계 (Next Step)

이제 우리는 설계(Architecture), 기초 공사(Initialization), 자재 검수(Dimension Check)까지 완벽하게 마쳤습니다.

이제 남은 것은 건물을 짓는 것뿐입니다. 다음 시간에는 [Project] Building a Deep Neural Network Application을 통해, 우리가 만든 코드로 '고양이 vs 개' 이미지를 분류하는 인공지능을 완성하겠습니다. 여러분의 첫 번째 Deep Learning 프로젝트가 시작됩니다!

[단원 요약 (Cheat Sheet)]

1. **Dimensions:** W는 $(n^{[l]}, n^{[l-1]})$이다. 차원 확인이 디버깅의 시작이다.
2. **Zero Init:** W를 0으로 하면 학습이 안 된다. (대칭성 문제)
3. **He Init:** ReLU를 쓸 때는 'randn * sqrt(2/n)' 공식을 사용하라.
4. **Bias:** 편향 b는 0으로 초기화해도 안전하다.

11 Train/Dev/Test Split

☒ 지난 시간 복습 및 연결

지금까지 우리는 신경망이라는 '최고급 엔진'을 조립했습니다. 하지만 페라리 엔진을 트랙터에 달거나, 불순물이 섞인 연료를 넣으면 아무 소용이 없습니다. 이제부터는 엔진을 '어떻게 운용해야(Strategy)' 최고의 성능을 낼 수 있는지 배웁니다. 그 첫걸음은 데이터를 올바르게 나누는 것입니다. 많은 초심자가 데이터를 몽땅 털어 넣고 학습부터 시키지만, 이는 "채점 기준도 모른 채 시험 공부를 하는 것"과 같습니다.

11.1 Overview

[핵심 목표]

이 단원은 성공적인 머신러닝 프로젝트의 나침반인 '데이터 분할 전략'을 다룹니다.

- **정의:** Train(학습), Dev(튜닝), Test(평가) 세트의 명확한 역할 차이를 이해합니다.
- **비율:** 빅데이터 시대(100만 개 이상)에 왜 98:1:1 비율을 사용하는지 통계적으로 설명합니다.
- **원칙:** Dev와 Test 세트가 반드시 동일한 분포(Same Distribution)여야 하는 이유를 배웁니다.
- **진단:** 데이터 분할 결과를 통해 모델의 과소적합/과대적합을 진단하는 표를 해석합니다.

11.1.1 Essential Terminology

데이터셋	역할	비유 (수험생)
Train Set	모델의 파라미터(W, b) 학습	**교과서** (평소 공부)
Dev Set	하이퍼파라미터 튜닝 & 모델 선택	**모의고사** (실력 점검 및 공부법 수정)
Test Set	최종 성능 평가 (학습/튜닝 관여 X)	**수능/본고사** (결과 번복 불가)

* Note: 과거에는 'Validation Set'이라고 불렀으나, Andrew Ng 교수는 'Dev Set'이라는 용어를 선호합니다.

11.1.2 Core Concepts: 전략적 분할

11.1.3 1. The Era Shift: 60/20/20 vs 98/1/1

데이터의 양(Size)에 따라 황금 비율은 달라집니다.

- **Traditional ML (Small Data):** 데이터가 1만 개 미만일 때.
  - 비율: **60% : 20% : 20%**
  - 이유: 평가용 데이터가 너무 적으면 통계적으로 신뢰할 수 없어서 20%나 떼어놔야 했습니다.
- **Deep Learning Era (Big Data):** 데이터가 100만 개 이상일 때.
  - 비율: **98% : 1% : 1%**
  - 이유: 100만 개의 1%면 1만 개입니다. 이 정도면 평가하기에 충분합니다. 나머지 98%를 학습(Train)에 몰아주어 성능을 극대화하는 것이 유리합니다.

11.1.4 2. The Golden Rule: Same Distribution

"Dev Set과 Test Set은 반드시 같은 과녁을 겨냥해야 한다."
[나쁜 예시 (Bad Example)]

- **Train:** 웹에서 크롤링한 고화질 고양이 사진 (20만 장)
- **Dev/Test:** 사용자가 폰으로 찍은 흐릿한 고양이 사진 (1만 장)

결과: 훈련 때는 99점(고화질 마스터)이지만, 실전에서는 0점입니다. **해결:** 모든 데이터를 섞어서(Shuffle) 나누거나, Dev/Test를 실제 목표(모바일 사진)로만 구성해야 합니다.

11.1.5 Deep Dive: 모델 진단 (Bias vs Variance)

데이터를 나누는 진짜 이유는 모델의 상태를 진단하기 위해서입니다.
성능 진단표 (Human Error ≈ 0% 가정)

Train Error	Dev Error	진단 (Diagnosis)	처방 (Action)
1%	11%	**High Variance** (과대적합)	데이터 추가, 정규화(Dropout), 모델 축소
15%	16%	**High Bias** (과소적합)	더 큰 모델(층 추가), 학습 시간 연장
15%	30%	**High Bias & Variance**	모델 구조 변경, 데이터 정제
0.5%	1%	**Low Bias & Low Variance**	**Ideal (성공!)**

- **Bias(편향) 문제:** Train Set조차 제대로 못 맞춤. (공부를 안 함)
- **Variance(분산) 문제:** Train은 잘 맞추는데 Dev는 못 맞춤. (교과서만 달달 외움, 응용 불가)

11.1.6 Implementation: Data Leakage 방지

가장 중요한 것은 Data Leakage(데이터 누수)를 막는 것입니다. 정규화(Normalization)를 할 때, 전체 데이터의 평균을 쓰면 안 됩니다. 오직 Train Set의 통계량만 사용해야 합니다.

Listing 10: Stratified Split Safe Normalization

```
import numpy as np
from sklearn.model_selection import train_test_split

def prepare_data(X, y):
    """
␣␣␣␣X:␣(m,␣n_x)␣features
␣␣␣␣y:␣(m,)␣labels
␣␣␣␣"""
    # 1. Stratified Split □□□( □□ □□□□□ □□)
    # Train(98%) vs Temp(2%)
    X_train, X_temp, y_train, y_temp = train_test_split(
        X, y, test_size=0.02, random_state=42, stratify=y
    )

    # □ Temp □□ □□ □□□ Dev(1%) vs Test(1%)
    X_dev, X_test, y_dev, y_test = train_test_split(
        X_temp, y_temp, test_size=0.5, random_state=42, stratify=y_temp
    )

    # 2. Data Leakage □□ □□□ □□(!)
    # □□ □X □□□, □□ X_train □□□ □□□□□□/ □□
    mean = np.mean(X_train, axis=0)
    std = np.std(X_train, axis=0)

    # □□□ □□□□□ Train, Dev, Test □□ □□
    X_train_norm = (X_train - mean) / (std + 1e-8)
    X_dev_norm = (X_dev - mean) / (std + 1e-8)
```

```
    X_test_norm = (X_test - mean) / (std + 1e-8)

    return X_train_norm, X_dev_norm, X_test_norm, y_train, y_dev, y_test
```

11.1.7 FAQ Pitfalls

Q1. Test Set 없이 Train/Dev만 쓰면 안 되나요?
A. 가능은 합니다만, 위험합니다. Dev Set을 보고 모델을 계속 수정하다 보면, 모델이 Dev Set에 과적합(Overfitting)됩니다. 마치 모의고사 답을 외워버린 것과 같습니다. 객관적인 최종 평가를 위해 Test Set은 한 번도 보지 않은 상태로 남겨둬야 합니다.

Q2. 시계열 데이터(주식)도 랜덤 셔플(Shuffle)해도 되나요?
A. 절대 안 됩니다. 미래 정보가 과거 학습 데이터에 섞여 들어가게 됩니다(Look-ahead Bias). 시계열 데이터는 시간 순서대로 잘라야 합니다. (예: 1 9월 Train, 10월 Dev, 11월 Test)

☒ 다음 단계 (Next Step)

데이터 세팅이 끝났습니다. 이제 여러분은 모델이 High Variance(과대적합) 상태인지, High Bias(과소적합) 상태인지 진단할 수 있습니다.

만약 진단 결과 모델이 High Variance(과대적합)라면 어떻게 해야 할까요? 데이터를 더 모으는 것이 좋겠지만, 돈과 시간이 듭니다. 다음 시간에는 데이터를 늘리지 않고도 과대적합을 해결하는 마법 같은 기법, [Regularization] (L2 Regularization Dropout)을 배우겠습니다.

[단원 요약 (Cheat Sheet)]

1. **Split:** 빅데이터 시대에는 98/1/1 비율이 대세다. Train에 집중하라.
2. **Distribution:** Dev와 Test는 반드시 같은 분포여야 한다.
3. **Leakage:** 정규화 시 평균(μ)과 분산(σ)은 오직 Train Set에서만 구한다.
4. **Diagnosis:** Train Error와 Dev Error의 차이가 크면 Variance(과대적합) 문제다.

12 Bias vs Variance

☒ 지난 시간 복습 및 연결

지난 시간에 우리는 데이터를 Train, Dev, Test로 나누는 전략을 세웠습니다. 이제 모델을 학습시켰고 성적표(Error Rate)를 받았습니다. 그런데 성적이 기대 이하입니다. 이때 "왜 성능이 안 나오지?"라고 막연해하면 안 됩니다. 머신러닝 엔지니어가 내릴 수 있는 진단은 딱 두 가지입니다. "공부를 덜 했거나(High Bias)" 아니면 "문제집만 달달 외웠거나(High Variance)". 이 두 가지 병명을 정확히 진단해야 올바른 약(Solution)을 쓸 수 있습니다.

12.1 Overview

[핵심 목표]

이 단원은 모델의 성능 저하 원인을 규명하는 '진단(Diagnosis)' 기술을 다룹니다.

- **개념:** 편향(Bias)과 분산(Variance)을 각각 과소적합(Underfitting)과 과대적합(Overfitting)으로 이해합니다.
- **기준:** Bayes Error(최적 오차)를 기준으로 Train Error와 Dev Error의 격차(Gap)를 분석합니다.
- **변화:** 딥러닝 시대에 Bias와 Variance를 동시에 줄이는 것이 가능해진 이유를 알아봅니다.
- **구현:** 학습 곡선(Learning Curve)을 그리고 자동으로 상태를 진단하는 Python 코드를 작성합니다.

12.1.1 Essential Terminology

용어	의미	비유 (학생)
High Bias	과소적합 (Underfitting)	공부를 대충 해서 교과서(Train) 내용도 모름.
High Variance	과대적합 (Overfitting)	교과서 답만 달달 외워서 응용 문제(Dev)는 다 틀림.
Bayes Error	이론적 최소 오차	인간도 틀릴 수밖에 없는 문제의 난이도 (한계치).
Avoidable Bias	Train Error - Bayes Error	우리가 노력으로 줄일 수 있는 편향.

12.1.2 Core Concepts: 과녁 맞추기 (Bullseye Analogy)

12.1.3 1. High Bias (Underfitting)

- 현상: 화살들이 정중앙에서 멀리 떨어져 있고, 자기들끼리는 뭉쳐 있습니다.
- 원인: 모델이 너무 단순해서(예: 직선) 데이터의 복잡한 패턴을 전혀 파악하지 못했습니다.

12.1.4 2. High Variance (Overfitting)

- 현상: 화살들의 평균 위치는 정중앙이지만, 사방으로 흩어져 있습니다.
- 원인: 훈련 데이터의 사소한 노이즈까지 과도하게 학습해서, 조금만 다른 데이터가 오면 예측이 널뜁니다.

12.1.5 3. The Diagnostic Recipe (진단 레시피)

숫자를 보고 진단하는 법입니다. Bayes Error(인간 수준 오차)가 0%라고 가정합니다.

증상별 처방전

Train Error	Dev Error	진단 (Diagnosis)	처방 (Prescription)
1%	11%	**High Variance**	데이터 추가, 정규화(L2/Dropout)
15%	16%	**High Bias**	더 큰 모델(은닉층 추가), 오래 학습
15%	30%	**High Bias & Variance**	모델 구조 변경, 데이터 정제
0.5%	1%	**Good Fit**	현재 상태 유지

12.1.6 Deep Dive: 딥러닝 시대의 트레이드오프

- **과거 (Traditional ML):** Bias를 줄이면 Variance가 늘어나는 '시소' 관계였습니다.
- **현재 (Deep Learning):**
  - **Bias 줄이기:** 네트워크를 더 크게 만듭니다. (데이터가 많다면 Variance를 거의 건드리지 않음)
  - **Variance 줄이기:** 데이터를 더 많이 모읍니다. (Bias를 거의 건드리지 않음)
- **결론:** 컴퓨팅 파워와 데이터만 충분하다면, Bias와 Variance를 동시에 잡을 수 있습니다.

12.1.7 Implementation: Auto-Diagnosis Tool

학습 기록(History)을 입력받아 자동으로 병명을 진단해주는 클래스를 만듭니다.

Listing 11: Model Diagnosis Class

```
import matplotlib.pyplot as plt

class ModelDiagnostician:
    def __init__(self, train_acc, dev_acc, human_acc=0.99):
        # □ □ □ (Accuracy)□ □ □ (Error)□ □ □
        self.train_err = 1.0 - train_acc
        self.dev_err = 1.0 - dev_acc
        self.human_err = 1.0 - human_acc

    def diagnose(self):
        print(f"Human Error: {self.human_err:.2%}")
        print(f"Train Error: {self.train_err:.2%}")
        print(f"Dev Error  : {self.dev_err:.2%}")
        print("-" * 30)

        # 1. Bias □ □  (□ Train □ Human □ □ )
        avoidable_bias = self.train_err - self.human_err

        # 2. Variance □ □  (□ Dev □ Train □ □ )
        variance = self.dev_err - self.train_err

        threshold = 0.02 # 2% □ □  □ □ □ □  □ □ □  □ □

        if avoidable_bias > threshold:
            print("[Diagnosis] High Bias (Underfitting)")
            print(">> Solution: Bigger Network, Train Longer (Epochs)")

        elif variance > threshold:
            print("[Diagnosis] High Variance (Overfitting)")
            print(">> Solution: More Data, Regularization (Dropout, L2)")

        else:
            print("[Diagnosis] Good Fit! Great Job.")

# --- □ □  □ □  ---
if __name__ == "__main__":
    # □ □ : □ □ □  □  □ □ □ (99%), □ □ □  □  □ (89%) -> High Variance
    train_accuracy = 0.99
    dev_accuracy = 0.89
```

```
40
41      doctor = ModelDiagnostician(train_accuracy, dev_accuracy)
42      doctor.diagnose()
```

12.1.8 FAQ Pitfalls

[Train Error가 높다고 무조건 High Bias인가요?]

아닙니다! 비교 대상(Bayes Error)이 중요합니다.

- **상황:** 흐릿한 옛날 문서 인식. 사람도 15% 틀림(Human Error = 15%).
- **결과:** 모델의 Train Error가 15%임.
- **진단:** 이것은 High Bias가 아닙니다. 이미 사람만큼 잘한 것입니다(Optimal). 이 경우엔 Bias를 줄이려 노력할 필요가 없습니다.

Q. High Bias와 High Variance가 동시에 높으면요?

A. 최악의 상황입니다. 모델이 정답도 못 맞추면서 예측값은 널뛰기를 합니다. 보통 모델 구조 자체가 데이터에 맞지 않거나, 데이터에 심각한 오류가 있을 때 발생합니다.

☒ 다음 단계 (Next Step)

진단 결과, 만약 여러분의 모델이 High Variance(과대적합) 판정을 받았다면 어떻게 해야 할까요? "데이터를 더 모으세요"라는 조언은 쉽지만, 현실에서는 돈과 시간이 듭니다. 데이터를 늘리지 않고도 과대적합을 치료하는 마법의 알약이 있습니다.

다음 시간에는 [Regularization] 유닛에서 L2 정규화와 Dropout이라는 강력한 치료법을 배워보겠습니다.

[단원 요약 (Cheat Sheet)]

1. **High Bias:** Train Error가 높다. → 모델을 키워라(Bigger Network).
2. **High Variance:** Dev Error가 Train Error보다 훨씬 높다. → 데이터를 모으거나 정규화(Regularization)하라.
3. **Reference:** 절대적인 수치가 아니라 Bayes Error(Human-level)와의 차이(Gap)를 봐야 한다.
4. **Priority:** 보통 Bias를 먼저 잡고, 그 다음 Variance를 잡는 순서로 진행한다.

13 Regularization (L1/L2)

☒ 지난 시간 복습 및 연결

지난 시간에 우리는 모델이 High Variance(과대적합)라는 병에 걸렸음을 진단했습니다. 모델이 학습 데이터에만 너무 집착해서(암기해서), 실전 문제(Dev Set)를 못 푸는 상황입니다. 이제 처방전을 쓸 차례입니다. 과대적합을 치료하는 가장 전통적이고 강력한 항생제는 바로 '정규화(Regularization)'입니다. 정규화는 모델에게 **"정답을 맞추되, 너무 꼼수(큰 가중치)는 쓰지 마라"**라고 제약(Penalty)을 거는 것입니다.

13.1 Overview

[핵심 목표]

이 단원은 모델의 복잡도를 억제하여 일반화 성능을 높이는 L2 정규화를 집중적으로 다룹니다.

- **개념:** 비용 함수 J에 가중치 크기($||W||^2$)에 비례하는 벌점을 추가합니다.
- **수학:** 역전파 과정에서 가중치가 스스로 줄어드는 Weight Decay(가중치 감쇠) 현상을 수식으로 증명합니다.
- **구현:** 정규화 항이 포함된 Forward 및 Backward 코드를 작성합니다.
- **비교:** L1 정규화(Lasso)와의 차이점(희소성)을 이해합니다.

13.1.1 Essential Terminology

용어	기호	핵심 의미
L2 Regularization	Ridge	가중치의 제곱합을 벌점으로 사용. $W \to 0$ (작아짐).
L1 Regularization	Lasso	가중치의 절댓값 합을 벌점으로 사용. $W = 0$ (사라짐/희소성).
Lambda	λ	정규화 강도. 클수록 모델이 단순해짐(Underfitting 위험).
Weight Decay	-	매 업데이트마다 가중치가 일정 비율씩 감소하는 현상.
Frobenius Norm	$\|\|W\|\|_F^2$	행렬의 모든 원소를 제곱해서 더한 값.

13.1.2 Core Concepts: 벌점 시스템

13.1.3 1. The Idea of Penalty

우리의 목표는 비용 J를 최소화하는 것입니다. 여기에 "가중치 W가 커지면 벌점을 주겠다"는 새로운 규칙을 추가합니다.

$$J_{regularized}(W,b) = \underbrace{J_{original}(W,b)}_{\text{오차 (Cross Entropy)}} + \underbrace{\frac{\lambda}{2m}\sum_{l}||W^{[l]}||_F^2}_{\text{벌점 (L2 Penalty)}}$$

- λ (Lambda): 벌점의 강도입니다. 하이퍼파라미터입니다.
- m: 데이터 개수.
- $2m$: 미분할 때 제곱(^{2})이 내려와서 2와 약분되라고 미리 2로 나눠둡니다. (수학적 편의)

13.1.4 2. L1 vs L2 (Which one to use?)

- **L2 (Standard):** 가중치를 0에 가깝게 만듭니다. 모든 특성을 골고루 사용하게 합니다. **딥러닝의 기본값(Default)**입니다.
- **L1 (Sparse):** 가중치를 완전히 0으로 만듭니다. 불필요한 특성을 제거(Feature Selection)하고 싶을 때 쓰지만, 미분이 까다로워 잘 안 씁니다.

13.1.5 Deep Dive: Why "Weight Decay"?

이 섹션은 L2 정규화를 왜 '가중치 감쇠'라고 부르는지 수학적으로 증명합니다.
역전파 수식 유도 비용 함수 J_{reg}를 W에 대해 미분해 봅시다.

$$\frac{\partial J_{reg}}{\partial W} = \frac{\partial J_{orig}}{\partial W} + \frac{\partial}{\partial W}\left(\frac{\lambda}{2m}W^2\right)$$

$$dW_{reg} = dW_{orig} + \frac{\lambda}{m}W$$

(분모의 2가 미분되면서 사라졌습니다!)
이제 경사 하강법 업데이트 식에 대입합니다.

$$W_{new} = W - \alpha \cdot dW_{reg}$$

$$W_{new} = W - \alpha\left(dW_{orig} + \frac{\lambda}{m}W\right)$$

이 식을 W로 묶으면 놀라운 결과가 나옵니다:

$$W_{new} = \underbrace{\left(1 - \frac{\alpha\lambda}{m}\right)}_{\text{Decay Factor } (<1)} W - \alpha \cdot dW_{orig}$$

결론: 매 업데이트마다 가중치 W는 원래 학습 방향($-\alpha dW$)으로 가기 전에, 자기 자신의 크기를 $(1 - \frac{\alpha\lambda}{m})$ 비율만큼 줄입니다. 즉, 가만히 있어도 스스로 감소(Decay)합니다.

13.1.6 Implementation: L2 Regularization

정규화는 Forward(비용 계산)와 Backward(기울기 계산) 양쪽에 모두 코드를 추가해야 합니다.

Listing 12: L2 Regularization Implementation

```
import numpy as np

class L2Regularizer:
    def __init__(self, lambd):
        self.lambd = lambd

    def compute_cost(self, cost_cross_entropy, parameters, m):
        """
␣␣␣␣␣␣␣␣J_total␣=␣J_cross_entropy␣+␣(lambda␣/␣2m)␣*␣sum(W^2)
␣␣␣␣␣␣␣␣"""
        L = len(parameters) // 2
        L2_cost = 0

        for l in range(1, L + 1):
            W = parameters['W' + str(l)]
            # Frobenius Norm □□ □□
            L2_cost += np.sum(np.square(W))

        L2_cost *= (self.lambd / (2 * m)) # 2□ □ m □□ □□ !

        return cost_cross_entropy + L2_cost
```

```
22
23      def backward(self, dW_orig, W, m):
24          """
25  ␣␣␣␣␣␣␣␣dW_reg␣=␣dW_orig␣+␣(lambda␣/␣m)␣*␣W
26  ␣␣␣␣␣␣␣␣"""
27          # □□□□ □□ □□□□ □□ □□□□□( □□m□□!)
28          dW_reg = dW_orig + ((self.lambd / m) * W)
29
30          return dW_reg
31
32  # --- □□ □□ ---
33  if __name__ == "__main__":
34      m = 1000
35      lambd = 0.7
36      reg = L2Regularizer(lambd)
37
38      # □□□ W (Weight)
39      W = np.array([[0.5, -0.2], [0.1, 0.8]])
40      dW_orig = np.array([[0.01, 0.02], [-0.01, 0.05]]) # □□ □□□
41
42      # □□□ □□
43      dW_final = reg.backward(dW_orig, W, m)
44
45      print("Original␣dW:\n", dW_orig)
46      print("Regularized␣dW:\n", dW_final)
47      # dW □□ W□□ □□□□ □□ □ □□ -> □W□□□0□ □□ □
```

13.1.7 FAQ Pitfalls

[편향(Bias) b는 정규화 안 하나요?]

보통 안 합니다. b는 함수의 모양(곡률)이 아니라 위치만 이동시킵니다. 따라서 모델의 복잡도에 큰 영향을 주지 않습니다. W만 정규화해도 충분합니다.

Q. Lambda(λ) 값은 어떻게 정하나요?

A. 하이퍼파라미터입니다. 여러 값을 시도해보고 Dev Set의 오차가 가장 낮은 값을 찾아야 합니다. 보통 0.01, 0.001처럼 로그 스케일로 탐색합니다.

☒ 다음 단계 (Next Step)

우리는 L2 정규화를 통해 가중치가 너무 커지는 것을 막아 과대적합을 억제했습니다.

하지만 때로는 더 과격한 방법이 필요할 때가 있습니다. 가중치를 줄이는 게 아니라, 아예 뉴런을 무작위로 꺼버리는(Shutdown) 방법입니다. "어떻게 뇌세포를 죽이는데 학습이 더 잘 되나요?" 다음 시간에는 딥러닝에서 가장 독특하고 강력한 정규화 기법인 [Regularization] Dropout (드롭아웃)을 배우겠습니다.

[단원 요약 (Cheat Sheet)]

1. **L2 Regularization:** 가중치 제곱합(W^2)을 비용 함수에 추가하여 큰 가중치에 벌점을 준다.
2. **Weight Decay:** 역전파 시 W가 매번 조금씩 0을 향해 줄어든다.
3. **Effect:** W가 작아지면 모델이 선형(Linear)에 가까워져 복잡도가 줄어든다. (과대적합 해결)
4. **Tip:** Cost 계산 시엔 $2m$, Gradient 계산 시엔 m으로 나눈다.

14 Dropout Regularization

☒ 지난 시간 복습 및 연결

지난 시간에 우리는 가중치(W)의 크기를 강제로 줄여버리는 L2 정규화(Weight Decay)를 배웠습니다. 오늘 배울 기법은 조금 더 '과격'합니다. 모델의 과대적합(Overfitting)을 막기 위해, 학습 과정에서 멀쩡한 뉴런들을 무작위로 '제거(Kill)'해버립니다. 바로 드롭아웃(Dropout)입니다. "뇌세포를 죽이는데 뇌가 더 똑똑해진다니?"라는 의문이 들겠지만, 이것이 현대 딥러닝에서 가장 강력한 정규화 기법입니다. 그 역설적인 원리를 파헤쳐 보겠습니다.

14.1 Overview

[핵심 목표]

이 단원은 신경망의 강건함(Robustness)을 높이는 드롭아웃의 원리와 구현을 다룹니다.

- **원리:** 드롭아웃이 어떻게 특정 뉴런에 대한 의존도(Co-adaptation)를 낮추는지 이해합니다.
- **수학:** 학습과 테스트 시의 출력값 차이를 보정하는 Inverted Dropout 기술을 익힙니다.
- **규칙:** 드롭아웃은 오직 학습(Training) 때만 켜고, 테스트(Test) 때는 끈다는 원칙을 명심합니다.
- **구현:** NumPy를 사용하여 마스크 행렬(Mask Matrix)을 만들고 적용해봅니다.

14.1.1 Essential Terminology

용어	변수	설명
Dropout	-	학습 시 뉴런을 무작위로 삭제(0으로 설정)하는 기법.
Keep Probability	`keep_prob`	뉴런을 **'살려둘'** 확률. (예: 0.8 = 20% 삭제).
Inverted Dropout	-	학습 시 값을 `keep_prob`로 나누어 스케일을 보정하는 표준 방식.
Ensemble	-	여러 모델의 예측을 평균 내는 것. 드롭아웃은 이 효과를 냄.

14.1.2 Core Concepts: 무작위 삭제의 미학

14.1.3 1. 직관적 해석 (Why does it work?)

[천재에게 의존하는 팀 프로젝트]

- **상황 (No Dropout):** 팀에 천재 한 명(특정 뉴런)이 있습니다. 다른 팀원들은 그 천재만 믿고 일을 안 합니다. 만약 천재가 결근하면(새로운 데이터), 프로젝트는 망합니다. (과대적합)
- **상황 (Dropout):** 매일 무작위로 팀원을 출근시키지 않습니다. 천재가 결근할 수도 있습니다.
- **결과:** 팀원들은 누구에게도 의존할 수 없으므로, 모두가 업무 전반을 익히게 됩니다. 결국 팀 전체가 강력하고 유연해집니다.

드롭아웃을 적용하면 뉴런들이 특정 입력(친구)에만 의존하지 않고, 가중치를 골고루 분산(Spread out)시키게 됩니다. 이는 L2 정규화와 비슷한 효과를 냅니다.

14.1.4 Deep Dive: Inverted Dropout (역 드롭아웃)

이 섹션은 면접 단골 질문인 "왜 학습 때 값을 나누나요?"에 대한 답입니다.

The Scale Problem `keep_prob = 0.5` (50% 삭제)라고 가정합시다.

1. 학습 단계 (Train): 뉴런의 절반이 0이 되므로, 다음 층으로 전달되는 합계($Z = \sum w_i a_i$)도 대략 **절반**으로 줄어듭니다.

2. 테스트 단계 (Test): 테스트 때는 드롭아웃을 끕니다(모든 뉴런 사용). Z 값이 학습 때보다 **2배 뻥튀기** 됩니다. 예측값이 완전히 달라집니다.

3. 해결책 (Inverted Dropout): 학습 단계에서 살아남은 뉴런의 값을 미리 **2배로 키워줍니다 ($A/ = 0.5$)**. 이렇게 하면 학습 때의 기댓값($E[A]$)이 테스트 때와 비슷하게 유지됩니다. 테스트 때는 아무런 연산도 할 필요가 없어집니다.

14.1.5 Implementation: Dropout Layer

가장 중요한 것은 'is$_t raining$'플래그입니다.테스트때는드롭아웃을적용하면안됩니다.

Listing 13: Inverted Dropout Implementation

```python
import numpy as np

class Dropout:
    def __init__(self, keep_prob=0.8):
        self.keep_prob = keep_prob
        self.mask = None # □ □ □ □  □ □ □  □ □

    def forward(self, A, is_training=True):
        """
␣␣␣␣␣␣␣␣A:␣Activation␣values
␣␣␣␣␣␣␣␣"""
        if is_training:
            # 1. □ □ □  □ □  (0 ~ 1 □ □  < keep_prob)
            # □ □ keep_prob □ □ □  True(1), □ □  False(0)
            D = np.random.rand(A.shape[0], A.shape[1])
            D = (D < self.keep_prob).astype(int)
            self.mask = D

            # 2. □ □  □ □  (Shut down)
            A = A * D

            # 3. □ □ □  □ □  (Inverted Dropout □ □ !)
            A = A / self.keep_prob

        return A

    def backward(self, dA):
        """
␣␣␣␣␣␣␣␣□ □ □ :␣□ □ ␣□ □ □ ␣□ □ □ □ ␣□ □ □ 0␣□
␣␣␣␣␣␣␣␣"""
        # 1. □ □ □  □ □
        dA = dA * self.mask

        # 2. □ □ □  □ □  □ □ □ ( □  □ □ □ □  □ □ □ □  □ □ □  □ )
        dA = dA / self.keep_prob

        return dA

# --- □ □  □ □  ---
if __name__ == "__main__":
    np.random.seed(1)
    A = np.ones((5, 3)) * 10 # □ □  □ □  □ 10 □ □

    dropout = Dropout(keep_prob=0.8)

    # Train Mode
    A_train = dropout.forward(A, is_training=True)
    print("Train␣Output:\n", A_train)
    # □ □ □  0, □ □ □ □  12.5 (10 / 0.8)□  □

```

```
    # Test Mode
    A_test = dropout.forward(A, is_training=False)
    print("\nTest Output:\n", A_test)
    # □□ □□□□ 10 □□
```

14.1.6 FAQ Pitfalls

[비용 함수(Cost Function) 진동 문제]

드롭아웃을 쓰면 매번 네트워크 구조가 무작위로 바뀝니다. 따라서 비용 함수 J가 매끄럽게 내려가지 않고 톱니바퀴처럼 진동할 수 있습니다. 디버깅할 때는 잠시 'keep$_prob = 1.0$'(드롭아웃끄기)으로설정하여J가잘내려가는지확인한후, 다시켜는것이좋

Q. 입력층(Input Layer)에도 드롭아웃을 쓰나요?

A. 보통은 안 씁니다. 원본 데이터(X)를 지워버리면 정보 손실이 너무 크기 때문입니다. 주로 파라미터가 많은 은닉층(FC Layer)에 사용합니다.

Q. `keep_prob`는 어떻게 정하나요?

A. 과대적합이 심할 것 같은 층(뉴런이 많은 층)은 낮게(0.5), 그렇지 않은 층은 높게(0.8 1.0) 설정합니다.

☒ 다음 단계 (Next Step)

이제 우리는 과대적합을 막는 두 가지 강력한 방패(L2, Dropout)를 얻었습니다. 하지만 모델 학습이 너무 느리다면 어떨까요? 아무리 좋은 모델도 학습에 1년이 걸린다면 무용지물입니다.

다음 시간에는 학습 속도를 비약적으로 높여주는 [Optimization] 기술로 넘어갑니다. 그 첫 번째 열쇠인 '입력 정규화(Input Normalization)'가 왜 경사 하강법의 속도를 높이는지 기하학적으로 살펴보겠습니다.

[단원 요약 (Cheat Sheet)]

1. **Dropout:** 학습 시 무작위로 뉴런을 끈다. (Ensemble 효과, 과대적합 방지)
2. **Inverted Dropout:** 학습 시 출력값을 `keep_prob`로 나눠주어 기댓값을 유지한다.
3. **Test Time:** 테스트 시에는 절대 드롭아웃을 쓰지 않는다.
4. **Caution:** Cost 그래프가 진동할 수 있으니 디버깅 시엔 끄고 확인한다.

15 Data Augmentation Early Stopping

☒ 지난 시간 복습 및 연결

우리는 L2 정규화와 드롭아웃이라는 강력한 수학적 기법으로 과대적합(Overfitting)을 억제했습니다. 오늘은 조금 더 '실용적이고(Practical)' '경제적인(Economical)' 접근법을 다룹니다. 데이터를 더 모으는 것은 비쌉니다. 하지만 가지고 있는 데이터를 변형해서 '공짜 데이터'를 만드는 기술(Augmentation)과, 학습을 가장 좋은 타이밍에 멈추는 '타임머신' 기술(Early Stopping)은 비용 대비 효과가 엄청납니다.

15.1 Overview

[핵심 목표]

이 단원은 모델의 일반화 성능을 높이는 가장 직관적이고 가성비 좋은 두 가지 기법을 다룹니다.

- **Data Augmentation:** 이미지를 변형하여 데이터셋을 뻥튀기하고, 모델에게 불변성(Invariance)을 가르칩니다.
- **Early Stopping:** 과대적합이 시작되기 직전에 학습을 멈추는 알고리즘을 구현합니다.
- **On-the-fly:** 디스크 용량을 아끼기 위해 학습 도중 실시간으로 데이터를 변형하는 파이프라인을 이해합니다.

15.1.1 Essential Terminology

용어	설명	비유
Data Augmentation	원본 데이터를 변형해 가짜 데이터를 생성	복사기로 문제집 복사하기 (근데 약간 비뚤게)
Early Stopping	성능 악화 시점에 학습 중단	박수 칠 때 떠나라
Patience	성능이 안 좋아져도 기다려주는 횟수	"한 번만 더 기회를 줄게"
On-the-fly	미리 저장하지 않고 필요할 때 즉석 생성	주문 들어오면 요리하기 (미리 해두면 상함)

15.1.2 Core Concepts: 공짜 점심은 있다

15.1.3 1. Data Augmentation (데이터 증강)

모델에게 "고양이는 뒤집어도, 어두워도, 잘려도 고양이다"라는 사실을 가르칩니다.

- **Mirroring (Flipping):** 거울처럼 좌우 반전. (숫자 인식 등 방향이 중요한 데이터엔 금지!)
- **Random Cropping:** 이미지의 일부분을 무작위로 잘라냄.
- **Rotation / Shearing:** 회전 및 비틀기.
- **Color Jittering:** 밝기, 채도 등에 노이즈 추가.

On-the-fly Generation (실시간 생성) "교수님, 변형된 이미지를 하드디스크에 저장해두고 써야 합니까?" **절대 아닙니다.** 1TB짜리 데이터셋을 10배 증강하면 10TB가 됩니다. 감당할 수 없습니다. **CPU가 학습 도중에 실시간으로 변형**해서 GPU에게 넘겨주는 방식을 사용합니다. 디스크 용량은 그대로 유지됩니다.

15.1.4 2. Early Stopping (조기 종료)

학습 곡선(Learning Curve)을 보다가, Train Error는 줄지만 Dev Error가 다시 올라가려는 순간(과대적합 시작점)에 멈춥니다.

- **원리:** 학습을 오래 하면 가중치 W가 점점 커져서 복잡한 패턴을 익히게 됩니다. Early Stopping은 W가 너무 커지기 전에 멈추므로, 수학적으로 L2 정규화와 유사한 효과를 냅니다.
- **단점 (Orthogonalization):** Andrew Ng 교수는 이 방식이 'Bias 줄이기'와 'Variance 줄이기'를 동시에 건드리기 때문에(직교화 위배), 튜닝이 복잡해질 수 있다고 지적합니다. 하지만 편해서 많이 씁니다.

15.1.5 Implementation: On-the-fly Pipeline

데이터 증강은 NumPy로 간단히, 조기 종료는 클래스로 구현하여 원리를 파악합니다.

Listing 14: Data Augmentation Early Stopping

```python
import numpy as np
import copy

class DataAugmentor:
    @staticmethod
    def random_flip(image, p=0.5):
        """□ □ □□ □ """
        if np.random.rand() < p:
            return np.fliplr(image)
        return image

    @staticmethod
    def random_crop(image, crop_size=(200, 200)):
        """□ □ □ □□ □ □□ □ □ """
        h, w, _ = image.shape
        top = np.random.randint(0, h - crop_size[0])
        left = np.random.randint(0, w - crop_size[1])
        return image[top:top+crop_size[0], left:left+crop_size[1], :]

class EarlyStopping:
    def __init__(self, patience=5, min_delta=0.0):
        self.patience = patience # □ □ □ □□ □ ()
        self.min_delta = min_delta # □ □ □□ □ □ □
        self.counter = 0
        self.best_loss = np.inf
        self.best_model = None
        self.stop = False

    def check(self, val_loss, model_params):
        if val_loss < (self.best_loss - self.min_delta):
            # □ □ □□ □ □ ! -> □ □ □□ □ □ □ □ □ □ □
            self.best_loss = val_loss
            self.counter = 0
            # □ □ : □ deepcopy □ □□ □ □ □ □ □ □ □ □ □ □ ( □ □ □□ □ □ )
            self.best_model = copy.deepcopy(model_params)
        else:
            # □ □ □□ □ □ □ □ / -> □ □ □ □□ □ □
            self.counter += 1
            print(f"EarlyStopping␣counter:␣{self.counter}/{self.patience}")
            if self.counter >= self.patience:
                self.stop = True

    def restore(self):
        return self.best_model
```

15.1.6 FAQ Pitfalls

[Deep Copy를 안 쓰면 생기는 일]

파이썬에서 'best$_{m}odel = model$'이라고쓰면, '$best_{m}odel$'은'$model$'을가리키는별명이될뿐입니다.학습이계속진행되어'$model$'이

Q. Data Augmentation을 Test Set에도 적용하나요?

A. 보통은 안 합니다. 하지만 성능을 극대화하기 위해 'Test Time Augmentation (TTA)'이라는 기법을 쓰기도 합니다. 테스트 이미지를 5가지로 변형해서 예측한 뒤 평균을 내는 것입니다. (대회용 테크닉)

☒ 다음 단계 (Next Step)

이제 우리는 과대적합을 막는 모든 무기(L2, Dropout, Augmentation, Early Stopping)를 갖췄습니다. 방어 준비는 끝났습니다.

이제 공격(학습 속도)을 강화할 차례입니다. 경사 하강법(Gradient Descent)은 너무 정직해서 느립니다. 다음 시간에는 [Optimization Algorithms]으로 넘어가서, 경사 하강법에 가속도를 붙이는 Momentum, 보폭을 조절하는 Adam 등 최신 최적화 기법을 배우겠습니다.

[단원 요약 (Cheat Sheet)]

1. **Augmentation:** 데이터를 변형해 양을 늘리고 불변성(Invariance)을 학습시킨다.
2. **On-the-fly:** 디스크 절약을 위해 학습 도중 실시간으로 변형한다.
3. **Early Stopping:** Dev Error가 오르기 시작하면 멈춘다. (과대적합 방지)
4. **Patience:** 일시적인 성능 저하를 견디기 위해 인내심(Patience) 값을 설정한다.

16 Mini-Batch Gradient Descent

☒ 지난 시간 복습 및 연결

지난 강의까지 우리는 과대적합을 막는 '방어 기술(Regularization)'을 익혔습니다. 이제부터는 모델의 학습 속도를 극한으로 끌어올리는 '가속 기술(Optimization)'을 다룹니다. 딥러닝의 연료는 데이터입니다. 데이터가 1000만 개라면, 기존 방식(Batch Gradient Descent)으로는 한 걸음 떼는 데 며칠이 걸립니다. 이를 해결하기 위해 데이터를 작은 덩어리로 쪼개서 학습하는 미니 배치(Mini-batch) 기법이 등장했습니다. 이는 현대 딥러닝의 사실상 표준(Standard)입니다.

16.1 Overview

[핵심 목표]

이 단원은 대용량 데이터를 효율적으로 학습시키는 미니 배치 경사 하강법을 다룹니다.

- **비교:** Batch(전체), Stochastic(1개), Mini-batch(덩어리)의 장단점을 비교합니다.
- **용어:** 헷갈리기 쉬운 Epoch(에폭)과 Iteration(반복)의 개념을 명확히 합니다.
- **원리:** 왜 배치 크기를 2^n (64, 128 등)으로 설정해야 하드웨어(CPU/GPU)가 좋아하는지 배웁니다.
- **구현:** 데이터를 무작위로 섞고(Shuffle) 나누는(Partition) 코드를 작성합니다.

16.1.1 Essential Terminology

용어	설명	예시 ($m = 1000$, Batch=100)
Epoch	전체 데이터(m)를 한 번 다 훑는 것.	책 1권을 1회독 함.
Iteration	파라미터를 한 번 업데이트하는 것.	문제 100개를 풀고 채점함.
Mini-batch	한 번의 Iteration에 쓰이는 데이터 묶음.	1 Epoch = 10 Iterations.

16.1.2 Core Concepts: 학습 방식의 스펙트럼

16.1.3 1. The Three Types of Gradient Descent

데이터셋 크기가 m일 때, 한 번의 업데이트에 몇 개의 데이터를 쓰는가?

- **Batch Gradient Descent (BGD):**
  - 데이터: 전체 m개 사용.
  - 특징: 안정적이지만 너무 느림. 메모리 부족 위험.
- **Stochastic Gradient Descent (SGD):**
  - 데이터: 딱 1개 사용.
  - 특징: 엄청 빠르지만 벡터화(병렬 처리) 이점이 없음. 진동이 심함.
- **Mini-batch Gradient Descent:**
  - 데이터: T개 사용 (예: 64, 128).
  - 특징: **Sweet Spot.** BGD의 안정성 + SGD의 속도 + 벡터화 효율성을 모두 잡음.

[산 내려오기 비유]

- **Batch:** 지도를 펼쳐 산 전체를 파악한 뒤, 정확하게 한 발자국 내딛습니다. (너무 신중함)
- **Stochastic:** 눈을 감고 발끝 감각만으로 미친 듯이 뛰어내려 갑니다. (빠르지만 비틀거림)
- **Mini-batch:** 100걸음 앞만 보고 방향을 잡아 내려갑니다. (적당히 빠르고 적당히 정확함)

16.1.4 2. Why Powers of 2? (2^n)

"교수님, 배치 크기를 100개나 50개로 하면 안 되나요?"

- 됩니다. 하지만 비효율적입니다.
- 컴퓨터 메모리(CPU/GPU)의 주소 체계와 캐시는 2진수 기반입니다.
- $32, 64, 128, 256, 512$ 등으로 설정하면 메모리 정렬(Alignment)이 딱 맞아떨어져 연산 속도가 최적화됩니다.

16.1.5 Implementation: Shuffle and Partition

데이터를 섞고(Shuffle) 자르는(Partition) 과정을 구현합니다. 가장 중요한 점은 X와 Y를 똑같은 순서로 섞어야 한다는 것입니다.

Listing 15: Random Mini-batches Generator

```python
import numpy as np
import math

def random_mini_batches(X, Y, mini_batch_size=64, seed=0):
    """
␣␣␣␣X:␣(n_x,␣m),␣Y:␣(1,␣m)
␣␣␣␣"""
    np.random.seed(seed)
    m = X.shape[1]  # □ □ □  □ □  □
    mini_batches = []

    # Step 1: Shuffle (X, Y)
    # 0 ~ m-1 □ □ □  □ □ □  □ □ □  □ □
    permutation = list(np.random.permutation(m))

    # □ □ : □ X □ Y □ □  □ □ □ □  □ □  □ ( □ □ )
    shuffled_X = X[:, permutation]
    shuffled_Y = Y[:, permutation].reshape((1, m))

    # Step 2: Partition □ □ □ ()
    # □  □  □ □ □  □ □
    num_complete_minibatches = math.floor(m / mini_batch_size)

    for k in range(0, num_complete_minibatches):
        begin = k * mini_batch_size
        end = (k + 1) * mini_batch_size

        mini_batch_X = shuffled_X[:, begin : end]
        mini_batch_Y = shuffled_Y[:, begin : end]
        mini_batches.append((mini_batch_X, mini_batch_Y))

    # Step 3: Handling the Remainder □ □ □ ( □ □ )
    # □ □ □ □  □  □ □ □ □ □ □ □  □ □  □  □ □ □  □ □ □  □ □
    if m % mini_batch_size != 0:
        begin = num_complete_minibatches * mini_batch_size
        # □ □ □  □  □ □
        mini_batch_X = shuffled_X[:, begin : ]
        mini_batch_Y = shuffled_Y[:, begin : ]
        mini_batches.append((mini_batch_X, mini_batch_Y))

    return mini_batches
```

16.1.6 FAQ Pitfalls

[Shuffle 시 주의사항]

시계열 데이터(주식, 날씨, 음성 등)처럼 순서(Time)가 중요한 데이터는 절대 섞으면 안 됩니다! 과거 데이터로 미래를 예측해야 하는데, 섞어버리면 미래를 보고 과거를 맞추는 꼴(Data Leakage)이 됩니다.

Q. 배치 크기가 너무 크면(8192 이상) 어떻게 되나요?

A. GPU 메모리 부족(OOM Error)이 발생할 수 있습니다. 또한, 모델이 너무 일반화된 패턴만 배워서 성능(Generalization)이 떨어지는 현상(Sharp Minima)이 발생할 수 있습니다. 보통 32 512 사이를 권장합니다.

☒ 다음 단계 (Next Step)

미니 배치를 쓰니 학습이 빨라졌습니다. 하지만 비용 함수 그래프를 확대해 보면 여전히 지그재그로 진동하며 내려갑니다.

이 진동을 줄이고, 내리막길에서 공이 굴러가듯 관성(Inertia)을 붙여 더 빠르게 내려가게 할 수는 없을까요? 다음 시간에는 단순한 경사 하강법을 넘어선 [Optimization] Momentum (모멘텀) 알고리즘에 대해 배우겠습니다.

[단원 요약 (Cheat Sheet)]

1. **Mini-batch GD:** 데이터를 작은 묶음으로 나누어 업데이트한다. (속도 + 안정성)
2. **Power of 2:** 배치 크기는 2^n (32, 64, 128...)이 하드웨어 효율적이다.
3. **Shuffle:** 매 에폭마다 데이터를 섞어주어야 학습이 골고루 된다.
4. **Last Batch:** 데이터가 나누어떨어지지 않을 때, 마지막 자투리 배치를 버리지 말고 처리해야 한다.

17 Momentum RMSprop

☒ 지난 시간 복습 및 연결

지난 시간에 우리는 데이터를 작은 덩어리로 쪼개 학습하는 미니 배치 경사 하강법을 배웠습니다. 속도는 빨라졌지만, 그래프를 보면 여전히 최적점을 향해 곧바로 가지 못하고 지그재그(Zigzag)로 진동(Oscillation)하며 내려갑니다. 이 진동을 줄이고, 최적해를 향해 '가속도(Acceleration)'를 붙일 수는 없을까요? 물리학의 관성을 이용한 Momentum과, 보폭을 자동으로 조절하는 RMSprop이 그 해답입니다.

17.1 Overview

[핵심 목표]

이 단원은 단순한 경사 하강법을 넘어선 고급 최적화 알고리즘 두 가지를 다룹니다.

- **기초:** 시계열 데이터의 트렌드를 추출하는 지수 가중 이동 평균(EWMA)을 이해합니다.
- **Momentum:** 과거의 기울기를 누적하여 관성을 만드는 원리를 배웁니다.
- **RMSprop:** 기울기의 크기(제곱)에 따라 학습 보폭을 조절하는 적응형 알고리즘을 익힙니다.
- **구현:** 두 알고리즘의 수식을 Python 코드로 옮기고 하이퍼파라미터(β)를 설정합니다.

17.1.1 Essential Terminology

용어	기호	설명
EWMA	v_t	지수 가중 이동 평균. 최근 데이터의 경향성을 나타냄.
Momentum	v	관성(속도). 과거의 진행 방향을 유지하려는 성질.
RMSprop	S	Root Mean Square Prop. 기울기 제곱을 이용해 보폭 조절.
Beta	β	과거 데이터를 얼마나 기억할지 결정하는 계수 (0.9 등).

17.1.2 Core Concepts: 가속의 원리

17.1.3 1. 지수 가중 이동 평균 (Exponentially Weighted Moving Average)

이 알고리즘들의 기초가 되는 수학입니다.

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

- $\beta = 0.9$: 최근 10일간의 평균 ($\frac{1}{1-0.9} = 10$)
- $\beta = 0.98$: 최근 50일간의 평균 ($\frac{1}{1-0.98} = 50$)
- β가 클수록 그래프가 부드러워지지만(Smoothing), 변화에 둔감해집니다(Latency).

17.1.4 2. Momentum (관성)

"공이 언덕을 굴러 내려갈 때 속도가 붙는 물리 법칙"
[얼음판 위의 쇠구슬]

- **SGD (일반 경사 하강법):** 마찰력이 무한대인 바닥. 힘(기울기)을 주면 움직이고, 안 주면 딱 멈춥니다. 방향이 바뀌면 즉시 꺾입니다 (지그재그).
- **Momentum:** 마찰력이 없는 얼음판. 힘을 주지 않아도 기존에 내려오던 속도(Velocity, v) 때문에 계속 미끄러져 내려갑니다. 이 관성이 진동을 상쇄하고 웅덩이(Local Minima)를 넘게 해줍니다.

Update Rule 1. 속도 계산: $v = \beta v + (1 - \beta)dW$
2. 파라미터 업데이트: $W = W - \alpha v$
(α: 학습률, β: 보통 0.9 사용)

17.1.5 3. RMSprop (Root Mean Square Propagation)

"가파른 곳은 천천히, 완만한 곳은 빠르게"
제프리 힌튼 교수가 제안한 방법입니다. 기울기(dW)의 크기를 보고 보폭을 조절합니다.

- **원리:** 학습률 α를 $\sqrt{S}$로 나눠줍니다.
- **효과:**
  - 기울기가 큼(S 큼) $\rightarrow$ 분모가 커짐 $\rightarrow$ 업데이트 폭 감소 (진동 억제)
  - 기울기가 작음(S 작음) $\rightarrow$ 분모가 작아짐 $\rightarrow$ 업데이트 폭 증가 (가속)

Update Rule 1. 제곱 평균: $S = \beta_2 S + (1 - \beta_2)dW^2$ (요소별 제곱)
2. 파라미터 업데이트: $W = W - \alpha \frac{dW}{\sqrt{S} + \epsilon}$
(ϵ: 0으로 나누기 방지용, 10^{-8})

17.1.6 Implementation: Momentum & RMSprop

Listing 16: Optimization Algorithms Implementation

```
import numpy as np

def update_with_momentum(parameters, grads, v, beta, learning_rate):
    """
␣␣␣␣v:␣□ □ (Velocity)␣□ □ □ □ ␣□ □ □ □ (␣0)
␣␣␣␣beta:␣Momentum␣□ □ ␣□ □ (␣0.9)
␣␣␣␣"""
    L = len(parameters) // 2

    for l in range(1, L + 1):
        # 1. □ □ (v) □ □ □ □  □ □ ( □ □ )
        v["dW" + str(l)] = beta * v["dW" + str(l)] + (1 - beta) * grads["dW" + str(l)]
        v["db" + str(l)] = beta * v["db" + str(l)] + (1 - beta) * grads["db" + str(l)]

        # 2. □ □ □ □  □ □ □ □  (□ v □ □ )
        parameters["W" + str(l)] -= learning_rate * v["dW" + str(l)]
        parameters["b" + str(l)] -= learning_rate * v["db" + str(l)]

    return parameters, v

def update_with_rmsprop(parameters, grads, s, beta2, learning_rate, epsilon=1e-8):
    """
␣␣␣␣s:␣□ □ ␣□ □ (Squared␣Gradient)␣□ □ □ □
␣␣␣␣beta2:␣RMSprop␣□ □ ␣□ □ (␣0.999)
␣␣␣␣"""
    L = len(parameters) // 2
```

```
27
28      for l in range(1, L + 1):
29          # 1. □□ □□ (s) □□□□□ □□□□ ( □□ □□ !)
30          s["dW" + str(l)] = beta2 * s["dW" + str(l)] + (1 - beta2) * np.square(grads["dW" + str(l)])
31          s["db" + str(l)] = beta2 * s["db" + str(l)] + (1 - beta2) * np.square(grads["db" + str(l)])
32
33          # 2. □□□□ □□□□ □□□( □□□ )
34          # □□□ sqrt(s) + epsilon
35          parameters["W" + str(l)] -= learning_rate * (grads["dW" + str(l)] / (np.sqrt(s["dW" + str(l)]) + epsilon))
36          parameters["b" + str(l)] -= learning_rate * (grads["db" + str(l)] / (np.sqrt(s["db" + str(l)]) + epsilon))
37
38      return parameters, s
```

17.1.7 FAQ Pitfalls

[변수 초기화 실수]

v와 s는 학습 루프(Iteration)가 돌 때마다 초기화하면 안 됩니다! 그러면 관성이 사라집니다. 반드시 학습 시작 전(Epoch 0 이전)에 한 번만 0으로 초기화하고, 계속 값을 누적해가야 합니다.

Q. β(Momentum)와 β_2(RMSprop) 값은 튜닝해야 하나요?
A. 보통은 기본값($\beta = 0.9, \beta_2 = 0.999$)을 그대로 씁니다. 이 값들이 경험적으로 대부분의 문제에서 잘 작동합니다. 학습률(α) 튜닝이 훨씬 중요합니다.

☒ 다음 단계 (Next Step)

우리는 최고의 가속 엔진 두 개를 얻었습니다.

- Momentum: 관성을 이용하여 속도를 높임.
- RMSprop: 보폭을 조절하여 진동을 줄임.

"둘 다 쓰면 안 되나요?" 당연히 됩니다. 이 둘을 결합한 것이 바로 Adam (Adaptive Moment Estimation)입니다. 현재 딥러닝 세계를 지배하고 있는 Adam 알고리즘을 다음 시간에 완성하고, 최적화 단원을 마무리하겠습니다.

[단원 요약 (Cheat Sheet)]

1. **Momentum:** $v \leftarrow dW$. 과거의 속도를 유지하여 Local Minima 탈출 및 가속.
2. **RMSprop:** $S \leftarrow dW^2$. 기울기가 크면 학습률을 낮춰 진동을 방지.
3. **Math:** dW^2은 요소별 제곱이다. 나눗셈 시 ϵ을 더해 에러를 방지한다.
4. **Hyperparam:** $\beta = 0.9$, $\beta_2 = 0.999$가 국룰(Standard)이다.

18 Adam Optimizer

☒ 지난 시간 복습 및 연결

우리는 지난 두 강의를 통해 '관성'을 이용해 속도를 높이는 Momentum과, '보폭'을 조절해 진동을 줄이는 RMSprop을 배웠습니다. 그렇다면 자연스러운 질문이 생깁니다. "이 두 가지 장점을 모두 합칠 수는 없을까?" 그 해답이 바로 Adam (Adaptive Moment Estimation)입니다. Adam은 현재 딥러닝 학계와 현업에서 'Default Optimizer(기본 설정)'로 통합니다. 어떤 옵티마이저를 쓸지 고민될 때, 일단 Adam을 쓰면 80점 이상은 갑니다.

18.1 Overview

[핵심 목표]

이 단원은 현대 딥러닝의 표준인 Adam Optimizer의 내부 구조를 해부합니다.

- **통합:** Adam이 Momentum의 평균(1차)과 RMSprop의 분산(2차)을 어떻게 결합하는지 수식으로 이해합니다.
- **보정:** 학습 초기에 0으로 쏠리는 현상을 막기 위한 편향 보정(Bias Correction)을 익힙니다.
- **표준:** $\beta_1, \beta_2, \epsilon$ 등 하이퍼파라미터의 국룰(Standard Value)을 배웁니다.
- **구현:** Python으로 편향 보정이 포함된 전체 알고리즘을 밑바닥부터 구현합니다.

18.1.1 Essential Terminology

기호	표준값	역할
α	튜닝 필요	**학습률 (Learning Rate)**. 가장 중요함.
β_1	0.9	**Momentum 계수**. (기울기의 지수 평균)
β_2	0.999	**RMSprop 계수**. (기울기 제곱의 지수 평균)
ϵ	10^{-8}	**안정성 상수**. 0으로 나누기 방지.

18.1.2 Core Concepts: 최강의 융합

18.1.3 1. The Fusion Algorithm

Adam은 매 스텝(t)마다 다음 4단계를 수행합니다.

1. **Momentum (v):** 속도를 계산합니다. (1차 모멘트)
$$v_t = \beta_1 v_{t-1} + (1-\beta_1) dW$$
2. **RMSprop (s):** 가속도 제어(마찰력)를 계산합니다. (2차 모멘트)
$$s_t = \beta_2 s_{t-1} + (1-\beta_2) dW^2$$
3. **Bias Correction (핵심):** 초기 0으로 쏠린 값을 보정합니다.
$$v_t^{corr} = \frac{v_t}{1-\beta_1^t}, \quad s_t^{corr} = \frac{s_t}{1-\beta_2^t}$$
4. **Update:** 파라미터를 갱신합니다.
$$W = W - \alpha \frac{v_t^{corr}}{\sqrt{s_t^{corr}} + \epsilon}$$

18.1.4 Deep Dive: Bias Correction (편향 보정)

"교수님, 왜 굳이 $(1-\beta^t)$로 나눠주나요?" 이것은 Adam의 정교함을 보여주는 대목입니다.

초기값 0의 저주 우리는 $v_0 = 0$으로 시작합니다. 첫 번째 스텝($t = 1$)을 봅시다. ($\beta_1 = 0.9$ 가정)

$$v_1 = 0.9 \times 0 + 0.1 \times dW = 0.1dW$$

문제점: 실제 기울기(dW)의 **10분의 1(0.1)**밖에 반영되지 않습니다. 학습 초반에 거북이처럼 느려집니다.

해결책 (보정):

$$1 - \beta_1^1 = 1 - 0.9 = 0.1$$

$$v_1^{corr} = \frac{v_1}{0.1} = \frac{0.1dW}{0.1} = dW$$

결과: 보정 덕분에 초기에도 기울기를 100% 반영할 수 있습니다. t가 커지면 $\beta^t \to 0$이 되어, 분모가 1이 되므로 보정 효과는 자연스럽게 사라집니다.

18.1.5 Implementation: Adam from Scratch

Adam 구현 시 가장 중요한 것은 현재 반복 횟수 t를 추적하는 것입니다.

Listing 17: Adam Optimizer Implementation

```
import numpy as np

def update_parameters_with_adam(parameters, grads, v, s, t, learning_rate=0.01,
                                beta1=0.9, beta2=0.999, epsilon=1e-8):
    """
␣␣␣␣t:␣□ □ ␣iteration␣count␣□ □ (1␣□ □ □ □ ␣□ !)
␣␣␣␣v,␣s:␣□ □ ␣□ □ □ □ ␣□ □ □ ␣Momentum,␣RMSprop␣□ □
␣␣␣␣"""
    L = len(parameters) // 2
    v_corrected = {}
    s_corrected = {}

    for l in range(1, L + 1):
        # --- 1. Momentum (v) ---
        v["dW" + str(l)] = beta1 * v["dW" + str(l)] + (1 - beta1) * grads["dW" + str(l)]
        v["db" + str(l)] = beta1 * v["db" + str(l)] + (1 - beta1) * grads["db" + str(l)]

        # --- 2. Bias Correction (v) ---
        # 1 - beta^t □  □□
        v_corrected["dW" + str(l)] = v["dW" + str(l)] / (1 - np.power(beta1, t))
        v_corrected["db" + str(l)] = v["db" + str(l)] / (1 - np.power(beta1, t))

        # --- 3. RMSprop (s) ---
        # □ □ □  □ □ (square) □ □ !
        s["dW" + str(l)] = beta2 * s["dW" + str(l)] + (1 - beta2) * np.square(grads["dW" + str(l)])
        s["db" + str(l)] = beta2 * s["db" + str(l)] + (1 - beta2) * np.square(grads["db" + str(l)])

        # --- 4. Bias Correction (s) ---
        s_corrected["dW" + str(l)] = s["dW" + str(l)] / (1 - np.power(beta2, t))
        s_corrected["db" + str(l)] = s["db" + str(l)] / (1 - np.power(beta2, t))

        # --- 5. Update Parameters ---
        # □ □ : sqrt(s_corr) + epsilon
        parameters["W" + str(l)] -= learning_rate * (v_corrected["dW" + str(l)] / (np.sqrt(s_corrected["dW" + str(l)])
      + epsilon))
        parameters["b" + str(l)] -= learning_rate * (v_corrected["db" + str(l)] / (np.sqrt(s_corrected["db" + str(l)])
      + epsilon))

    return parameters, v, s
```

18.1.6 FAQ Pitfalls

[Iteration Count t 주의]

함수를 호출할 때 t는 반드시 1부터 시작해야 합니다. 만약 $t = 0$이면 $1 - \beta^0 = 1 - 1 = 0$이 되어 ZeroDivisionError가 발생합니다.

Q. Adam이 항상 최고인가요?

A. 대부분의 경우(CV, NLP, GAN) 그렇습니다. 하지만 아주 정교한 수렴이 필요할 때(SOTA 논문 등)는 일반 SGD+Momentum이 더 좋은 성능을 낼 때도 있습니다. 그래도 시작은 무조건 Adam을 추천합니다.

☒ 다음 단계 (Next Step)

이로써 우리는 최적화 알고리즘의 정점인 Adam을 정복했습니다. 이제 여러분은 어떤 모델이든 빠르고 안정적으로 학습시킬 수 있는 엔진을 갖췄습니다.

하지만 엔진 성능이 좋아도, 기어 변속(하이퍼파라미터 설정)을 잘못하면 차가 나가지 않습니다. α, β, 배치 크기, 은닉층 개수... 도대체 무엇부터 조절해야 할까요? 다음 시간에는 이 수많은 다이얼을 어떤 순서로 돌려야 하는지, [Hyperparameter Tuning]의 체계적인 전략(Random Search vs Grid Search)을 알려드리겠습니다.

[단원 요약 (Cheat Sheet)]

1. **Adam:** Momentum(속도) + RMSprop(가속도 제어) + Bias Correction(초기 보정).
2. **Standard Params:** α(튜닝), $\beta_1(0.9)$, $\beta_2(0.999)$, $\epsilon(10^{-8})$.
3. **Bias Correction:** 학습 초반에 파라미터 업데이트가 너무 작아지는 것을 막아준다.
4. **Memory:** v와 s를 따로 저장해야 하므로 일반 SGD보다 메모리를 더 쓴다.

19 Learning Rate Decay

☒ 지난 시간 복습 및 연결

지난 시간에 우리는 Adam Optimizer를 통해 최적화의 정점에 도달했습니다. 하지만 아주 미세한 문제가 하나 남았습니다. 학습 후반부가 되면 손실 함수(Cost)의 최저점 근처에서 모델이 안착하지 못하고 계속 맴도는 진동(Oscillation) 현상이 발생합니다. 주차장에서 시속 100km로 달리면 절대 주차 칸에 차를 넣을 수 없습니다. 목적지 근처에서는 속도를 줄여야 합니다. 이것이 바로 학습률 감쇠(Learning Rate Decay)가 필요한 이유입니다.

19.1 Overview

[핵심 목표]

이 단원은 학습 단계별로 학습률(α)을 조절하여 모델 성능을 극대화하는 기법을 다룹니다.

- **이유:** 고정된 학습률이 최저점 근처에서 수렴하지 못하는 이유를 기하학적으로 이해합니다.
- **전략:** 시간 기반(Inverse Time), 지수(Exponential), 계단식(Step) 감쇠의 수식적 차이를 파악합니다.
- **구현:** Python으로 스케줄러를 구현하고 에폭(Epoch)에 따른 변화를 그래프로 확인합니다.

19.1.1 Essential Terminology

용어	기호	설명
Learning Rate	α	한 번 업데이트할 때 이동하는 보폭의 크기.
Decay Rate	k	학습률을 얼마나 빨리 줄일지 결정하는 계수.
Epoch	t	전체 데이터를 한 번 학습한 횟수. (시간 단위)
Initial LR	α_0	학습 시작 시점의 초기 학습률.

19.1.2 Core Concepts: 속도 조절의 미학

19.1.3 1. The Parking Problem (왜 줄여야 하는가?)

[고속도로와 주차장 비유]

- **Early Stage (고속도로):** 최저점이 멀리 있습니다. 이때는 보폭이 커야(High α) 빨리 접근할 수 있습니다.
- **Late Stage (주차장):** 최저점 근처입니다. 이때도 보폭이 크다면 구멍을 지나쳐 버리고(Overshooting), 다시 돌아오려다 또 지나칩니다.
- **Solution:** 목적지에 가까워질수록 속도를 서서히 줄여서 정밀하게 주차(수렴)해야 합니다.

19.1.4 2. Decay Schedules (감쇠 전략)

시간(t, Epoch)이 지날수록 α를 줄이는 대표적인 공식들입니다.

- **1. Inverse Time Decay (시간 기반):**

$$\alpha = \frac{1}{1 + k \cdot t}\alpha_0$$

  가장 완만하게 줄어듭니다.

- **2. Exponential Decay (지수 감쇠):**

$$\alpha = k^t \cdot \alpha_0 \quad (k < 1, \text{예: } 0.95)$$

  빠르게 0으로 수렴합니다.

- **3. Step Decay (계단식 감쇠):** 10 에폭마다 절반으로 뚝 떨어뜨립니다. (ResNet 등 심층 모델에서 선호)

19.1.5 Deep Dive: Adaptive Methods와의 관계

"교수님, Adam이 알아서 학습률 조절해주지 않나요?"

- **Adam/RMSprop:** 파라미터마다 *개별적으로* 학습률을 조절(Adaptive)하지만, 전체적인 *글로벌 학습률*(α) 자체는 고정되어 있습니다.
- **결론:** Adam을 쓰더라도 Learning Rate Decay를 함께 적용하면, 최저점에서의 진동을 줄여 성능을 더 높일 수 있습니다. (SOTA 모델들의 필수 테크닉)

19.1.6 Implementation: Decay Scheduler

직접 스케줄러를 만들고 그래프를 그려봅시다.

Listing 18: Learning Rate Schedulers

```
import numpy as np
import matplotlib.pyplot as plt

class LRScheduler:
    def __init__(self, init_lr=1.0):
        self.init_lr = init_lr

    def inverse_time_decay(self, epoch, k=0.1):
        """lr = lr0 / (1 + kt)"""
        return self.init_lr / (1 + k * epoch)

    def exponential_decay(self, epoch, k=0.95):
        """lr = lr0 * k^t"""
        return self.init_lr * np.power(k, epoch)

    def step_decay(self, epoch, drop=0.5, interval=10):
        """□ □ □ □ (interval)□ □ □drop□ □ □ □ □ □ □ """
        exponent = np.floor((1 + epoch) / interval)
        return self.init_lr * np.power(drop, exponent)

# --- □ □ □ ---
if __name__ == "__main__":
    epochs = np.arange(0, 100)
    scheduler = LRScheduler(init_lr=1.0)

    lr1 = [scheduler.inverse_time_decay(e) for e in epochs]
    lr2 = [scheduler.exponential_decay(e) for e in epochs]
    lr3 = [scheduler.step_decay(e) for e in epochs]

    plt.figure(figsize=(10, 6))
    plt.plot(epochs, lr1, label='Inverse Time')
    plt.plot(epochs, lr2, label='Exponential')
    plt.plot(epochs, lr3, label='Step Decay')
    plt.title('Learning Rate Decay Schedules')
    plt.xlabel('Epochs')
    plt.ylabel('Learning Rate')
    plt.legend()
    plt.grid(True)
    plt.show()
```

19.1.7 FAQ Pitfalls

ReduceLROnPlateau (실전 꿀팁) 가장 실용적인 방법은 수식보다는 성능을 보고 줄이는 것입니다. **"지난 10 에폭 동안 Dev Error가 줄어들지 않았네? (Plateau)"** $\rightarrow$ **"이제 정밀 타격할 때다. 학습률을 1/10로 줄여라."** Keras나 PyTorch에서 'ReduceLROnPlateau' 콜백을 사용하면 됩니다.

Q. k(감쇠율)가 너무 크면 어떻게 되나요?

A. 학습률이 너무 빨리 0이 되어버립니다. 최저점에 도달하기도 전에 모델이 멈춰버리는 조기 수렴(Premature Convergence) 문제가 발생합니다.

⊠ 다음 단계 (Next Step)

이제 최적화 도구들은 모두 갖췄습니다. 하지만 하이퍼파라미터가 너무 많아졌습니다. ($\alpha, \beta_1, \beta_2, \epsilon, k, \lambda$, batch_size ...)

이 많은 다이얼을 어떤 순서로, 어떻게 맞춰야 할까요? 사람이 일일이 돌려보기엔 시간이 너무 부족합니다. 다음 시간에는 [Hyperparameter Tuning] 전략을 통해, 이 복잡한 퍼즐을 체계적으로 푸는 법을 배웁니다. Grid Search와 Random Search의 승부가 펼쳐집니다.

[단원 요약 (Cheat Sheet)]

1. **Need for Decay:** 고정 학습률은 최저점 근처에서 진동한다. 정밀한 수렴을 위해 줄여야 한다.
2. **Schedules:** Inverse Time(완만), Exponential(급격), Step(계단식) 등이 있다.
3. **Best Practice:** Dev Error가 정체될 때 줄이는 `ReduceLROnPlateau` 방식이 가장 효과적이다.
4. **With Adam:** Adam을 쓰더라도 Decay를 함께 쓰면 성능이 더 좋아진다.

20 Hyperparameter Tuning

☒ 지난 시간 복습 및 연결

우리는 지금까지 신경망을 만들고(Architecture), 학습시키고(Adam), 규제(Regularization)하는 모든 방법을 배웠습니다. 하지만 막상 여러분이 모델을 돌리려고 하면 거대한 벽에 부딪힙니다. "학습률은 0.01? 0.0001?", "배치 크기는 32? 64?" 수십 개의 다이얼을 무작위로 돌리는 것은 도박입니다. 우리는 체계적이고 과학적인 탐색 전략이 필요합니다.

20.1 Overview

[핵심 목표]

이 단원은 SOTA(State-of-the-art) 성능을 달성하기 위한 하이퍼파라미터 튜닝의 우선순위와 탐색 기법을 다룹니다.

- **우선순위:** 학습률(α)이 가장 중요하다는 계층 구조를 이해합니다.
- **전략:** 고차원 공간에서는 Random Search가 Grid Search보다 압도적으로 유리한 이유를 기하학적으로 증명합니다.
- **스케일:** 학습률 등을 탐색할 때 선형(Linear)이 아닌 로그 스케일(Log Scale)을 써야 하는 수학적 이유를 배웁니다.

20.1.1 Essential Terminology

기법	설명	비유
Grid Search	모든 조합을 격자무늬로 다 해보는 것.	자물쇠 번호를 0000부터 9999까지 다 돌려봄.
Random Search	무작위로 값을 찍어보는 것.	감으로 찍어서 맞춤 (고차원에서 유리).
Log Scale	자릿수 단위로 탐색 ($10^{-4}, 10^{-3} \dots$).	현미경 배율을 10배, 100배로 조절하며 관찰.
Coarse to Fine	넓게 훑고(Coarse), 좋은 곳을 집중 공략(Fine).	숲 전체를 스캔하고, 의심 가는 구역만 수색.

20.1.2 Core Concepts: 무엇이 중요한가?

20.1.3 1. Tuning Priority (우선순위 계급도)

모든 파라미터가 평등하지 않습니다. 앤드류 응 교수의 경험적 가이드라인입니다.

- **Tier 1 (King - 가장 중요):**
  - **Learning Rate (α)**: 이것이 틀리면 다른 걸 아무리 잘 맞춰도 소용없습니다.
- **Tier 2 (Queen - 중요):**
  - Momentum (β), Mini-batch Size, Hidden Units 개수.
- **Tier 3 (Pawn - 덜 중요):**
  - Layer 개수, Learning Rate Decay.
- **Do Not Touch (건드리지 마세요):**
  - Adam의 $\beta_1(0.9), \beta_2(0.999), \epsilon(10^{-8})$.

20.1.4 2. Grid Search vs Random Search

[보물 찾기]

지도상에 보물이 어디 있는지 모릅니다.

- **Grid Search:** 지도를 바둑판처럼 나누고 교차점만 팝니다. 만약 보물이 교차점 사이에 있다면? 영원히 못 찾습니다.
- **Random Search:** 지도를 무작위로 콕콕 찌릅니다. 같은 횟수를 시도하더라도, 중요한 파라미터에 대해 훨씬 더 다양한 값을 테스트해볼 수 있습니다.

20.1.5 3. Scale Matters: Log Scale Sampling

학습률 α를 0.0001에서 1 사이에서 찾는다고 합시다.

나쁜 방법 (Linear): 'np.random.rand()'

- 90%의 값이 $0.1 \sim 1$ 구간에 몰립니다.
- 정작 중요한 $0.0001 \sim 0.1$ 구간은 전체의 10%밖에 탐색하지 못합니다.

좋은 방법 (Log Scale):

- $10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}$ 각 구간을 공평하게 탐색해야 합니다.
- 지수(r)를 $-4 \sim 0$ 사이에서 뽑고, 10^r을 계산합니다.

20.1.6 Implementation: Scientific Search

올바른 스케일로 파라미터를 뽑는 함수를 구현합니다.

Listing 19: Hyperparameter Sampling Strategies

```
import numpy as np

class HyperparameterSearch:
    def sample_linear(self, low, high, num_samples):
        """
␣␣␣␣␣␣␣␣□ □ ␣□ □ ␣□ ,␣□ ␣□ ␣□ ␣□ □ □ □ (␣□ □ ␣□ □ ␣□ □ )
␣␣␣␣␣␣␣␣"""
        return np.random.uniform(low, high, num_samples)

    def sample_log_scale(self, low_exp, high_exp, num_samples):
        """
␣␣␣␣␣␣␣␣□ □ □ (alpha),␣□ □ □ □ □ (lambda)␣□ ␣□ □ □ □ (␣□ □ □ ␣□ □ )
␣␣␣␣␣␣␣␣□ □ :␣10^low_exp␣~␣10^high_exp
␣␣␣␣␣␣␣␣"""
        # 1. □ □ (r)□  □ □ □ □  □ □  (-4 ~ 0)
        r = np.random.uniform(low_exp, high_exp, num_samples)
        # 2. □ 10 □ □ □ □ □ □  □ □
        return 10  r

    def sample_beta(self, num_samples):
        """
␣␣␣␣␣␣␣␣Momentum␣Beta␣(0.9␣~␣0.999)
␣␣␣␣␣␣␣␣1-beta␣□ □ ␣□ □ ␣□ □ □ □ ␣□ □ ␣□ □ ␣□ □ !
␣␣␣␣␣␣␣␣"""
        # 1-beta □ □ : 0.1 ~ 0.001 (10^-1 ~ 10^-3)
        r = np.random.uniform(-3, -1, num_samples)
        return 1 - (10  r)

# --- □ □  ---
if __name__ == "__main__":
    searcher = HyperparameterSearch()

```

```
    # □ □ □  □ □ : 0.0001 ~ 1.0
    alphas = searcher.sample_log_scale(-4, 0, 5)
    print("Alphas:", np.round(alphas, 6))

    # □ □ □  □ □ : 0.9 ~ 0.999
    betas = searcher.sample_beta(5)
    print("Betas:", np.round(betas, 6))
```

20.1.7 FAQ Pitfalls

Coarse to Fine 전략 처음부터 100 Epoch씩 돌리며 완벽한 값을 찾으려 하지 마세요. 1. Coarse: 넓은 범위에서 5 10 Epoch만 짧게 돌려 대략적인 성능을 봅니다. 2. Zoom In: 성능이 좋은 영역을 발견하면 그 구간을 집중 확대합니다. 3. Fine: 좁은 영역에서 정밀하게 다시 Random Search를 수행합니다.

Q. β(Momentum)는 왜 $1-\beta$로 로그 샘플링하나요?

A. β는 1에 가까워질수록 민감해지기 때문입니다. $0.9 \to 0.9005$는 별 차이 없지만, $0.999 \to 0.9995$는 평균 기간이 1000일에서 2000일로 2배가 됩니다. 1에 가까운 값을 더 세밀하게 탐색해야 합니다.

☒ 다음 단계 (Next Step)

이제 최적의 파라미터까지 찾았습니다. 하지만 모델이 깊어질수록 "학습 속도가 느려지고 초기값에 너무 민감해지는 문제"가 발생합니다. 데이터 분포가 층을 지날 때마다 틀어지기 때문입니다.

이를 해결하기 위해 딥러닝 역사상 가장 위대한 발명 중 하나인 [Batch Normalization]이 등장했습니다. 다음 시간에 이 마법 같은 기법을 파헤쳐 보겠습니다.

[단원 요약 (Cheat Sheet)]

1. **Priority:** 학습률(α)이 1순위다.
2. **Strategy:** Grid Search보다는 Random Search가 효율적이다.
3. **Log Scale:** α나 λ는 자릿수 단위로 탐색해야 한다. $(10^{-4}, 10^{-3} \dots)$
4. **Process:** 넓게 훑고(Coarse), 좁게 파고들어라(Fine).

21 Batch Normalization

☒ 지난 시간 복습 및 연결

우리는 가중치 초기화와 하이퍼파라미터 튜닝을 통해 모델 성능을 높였습니다. 하지만 층이 깊어질수록 여전히 학습이 불안정하고, 학습률을 조금만 높여도 발산해버리는 문제가 발생합니다. 이유는 앞단 층의 파라미터가 바뀌면, 뒷단 층으로 넘어오는 데이터의 분포가 계속 바뀌기 때문입니다. 뒷단 층 입장에서는 계속 흔들리는 땅 위에서 균형을 잡으려는 것과 같습니다. 이를 해결하기 위해 2015년, "데이터 분포를 강제로 고정시키자"는 혁명적인 아이디어가 등장합니다. 바로 배치 정규화(Batch Normalization)입니다.

21.1 Overview

[핵심 목표]

이 단원은 딥러닝 역사상 가장 위대한 발명 중 하나인 배치 정규화의 원리와 구현을 다룹니다.

- **원인:** 학습을 방해하는 내부 공변량 변화(Internal Covariate Shift) 현상을 이해합니다.
- **알고리즘:** 미니 배치 단위로 평균/분산을 정규화하고, Scale(γ) & Shift(β) 파라미터로 복원하는 과정을 유도합니다.
- **차이:** 학습(Train) 때는 배치 통계량을, 추론(Test) 때는 이동 평균(Running Average)을 써야 함을 배웁니다.
- **구현:** 두 가지 모드를 지원하는 BN 클래스를 Python으로 구현합니다.

21.1.1 Essential Terminology

용어/기호	의미	역할
Batch Norm	배치 정규화	은닉층의 활성화 값을 정규 분포로 만듦.
Gamma (γ)	스케일 파라미터	정규화된 값의 분산을 조절 (학습 가능).
Beta (β)	시프트 파라미터	정규화된 값의 평균을 조절 (학습 가능).
Running Stats	이동 평균 통계량	테스트 시 사용하기 위해 학습 중 누적해둔 평균/분산.

21.1.2 Core Concepts: 흔들리는 땅 고정하기

21.1.3 1. Internal Covariate Shift (내부 공변량 변화)

[흔들리는 다리 위에서 걷기]

- **Without BN:** 앞사람(Layer 1)이 발을 구를 때마다 다리가 흔들립니다. 뒷사람(Layer 2)은 중심 잡느라 앞으로 나아갈 수가 없습니다. (학습 속도 저하)
- **With BN:** 각 층마다 "발판을 수평으로 고정(Normalize)"해줍니다. 뒷사람은 앞사람의 움직임에 상관없이 안정적으로 달릴 수 있습니다. (학습 속도 비약적 향상)

21.1.4 2. The Algorithm: Norm, Scale, Shift

은닉층의 값 Z에 대해 미니 배치 단위로 4단계를 수행합니다.

1. **Mean:** $\mu = \frac{1}{m}\sum z^{(i)}$
2. **Variance:** $\sigma^2 = \frac{1}{m}\sum (z^{(i)} - \mu)^2$

3. **Normalize:** $z_{norm} = \frac{z-\mu}{\sqrt{\sigma^2+\epsilon}}$ (ϵ: 0 나누기 방지)

4. **Scale & Shift (핵심):** $\tilde{z} = \gamma z_{norm} + \beta$

[왜 다시 γ, β로 망가뜨리나요?]

무조건 평균 0, 분산 1로 고정하면, 데이터가 Sigmoid의 선형 구간(가운데)에만 몰리게 되어 비선형성(표현력)을 잃게 됩니다. γ와 β를 학습 가능하게 두어, "필요하다면 원래 분포로 되돌릴 수 있는 자유"를 모델에게 주는 것입니다.

21.1.5 Deep Dive: Train vs Test Mode

배치 정규화 구현에서 가장 중요한 포인트입니다.

- **Training Mode:** 현재 들어온 미니 배치의 평균/분산을 계산해서 씁니다. 동시에 이 값들을 'running$_mean$', '$running_var$'에

- **Test Mode:** 테스트 때는 데이터가 1개만 들어올 수도 있습니다(분산 계산 불가). 따라서 학습 때 미리 저장해둔 'running$_mean$', '$running_var$'를가져와서정규화합니다.

21.1.6 Implementation: Batch Norm Class

Listing 20: Batch Normalization Implementation

```
import numpy as np

class BatchNorm:
    def __init__(self, n_features, momentum=0.9):
        self.gamma = np.ones((n_features, 1)) # □□□ 1 □□(□□)
        self.beta = np.zeros((n_features, 1)) # □□□ 0 □□(□□)

        # □□□ □□□ □□ □□□ (Running Stats)
        self.running_mean = np.zeros((n_features, 1))
        self.running_var = np.ones((n_features, 1))
        self.momentum = momentum
        self.epsilon = 1e-8

    def forward(self, Z, mode='train'):
        if mode == 'train':
            # 1. □□ □□ □□□ □□
            mu = np.mean(Z, axis=1, keepdims=True)
            var = np.var(Z, axis=1, keepdims=True)

            # 2. □□□
            Z_norm = (Z - mu) / np.sqrt(var + self.epsilon)

            # 3. Running Stats □□□□ □□(□□ □□)
            # □□□□ □□□□ □□□ □□□□
            self.running_mean = self.momentum * self.running_mean + (1 - self.momentum) * mu
            self.running_var = self.momentum * self.running_var + (1 - self.momentum) * var

        elif mode == 'test':
            # □□□ □□□ □□□□ □□□ □□
            Z_norm = (Z - self.running_mean) / np.sqrt(self.running_var + self.epsilon)

        # 4. Scale and Shift □□()
        out = self.gamma * Z_norm + self.beta
        return out
```

21.1.7 FAQ Pitfalls

Q. BN을 쓰면 왜 b(Bias)를 없애도 되나요?
A. $Z = WX + b$에서 평균 μ를 빼는 과정($Z - \mu$) 때문에 상수 b는 어차피 상쇄되어 사라집니다. 대신 BN의 β가 편향 역할을 대신합니다.

Q. BN은 어디에 넣나요? Activation 전? 후?
A. 원래 논문(Andrew Ng 스타일)은 Activation 전($Z \to BN \to A$)을 권장합니다. 하지만 최근에는 후($Z \to A \to BN$)에 넣는 경우도 많습니다. 둘 다 잘 동작합니다.

☒ 다음 단계 (Next Step)

이제 우리는 딥러닝 모델의 성능을 극한으로 끌어올리는 모든 도구(초기화, 정규화, 최적화, 배치 정규화)를 손에 넣었습니다.

지금까지는 '고양이 vs 개'처럼 답이 두 개인 이진 분류만 다뤘습니다. 하지만 세상에는 답이 여러 개인 문제가 더 많습니다. (숫자 0 9, 옷 종류 등) 다음 시간에는 여러 개의 클래스를 동시에 분류하는 [Softmax Regression]에 대해 다루겠습니다.

[단원 요약 (Cheat Sheet)]

1. **Batch Norm:** 각 층의 입력을 정규화하여 학습을 안정화하고 가속한다.
2. **Gamma/Beta:** 정규화로 잃어버린 표현력을 복구하기 위한 학습 파라미터.
3. **Train/Test:** 학습 시엔 배치 통계량, 테스트 시엔 이동 평균(Running Avg)을 쓴다.
4. **Effect:** 초기화에 덜 민감해지고, 높은 학습률을 쓸 수 있다.

22 Orthogonalization

☒ 지난 시간 복습 및 연결

우리는 지금까지 딥러닝 모델을 만들고(Build), 학습시키고(Train), 최적화(Optimize)하는 기술적인 방법론을 배웠습니다. 하지만 여러분이 현업에서 리더가 되어 팀을 이끌게 되면 이런 질문에 봉착합니다. "교수님, 성능이 안 나오는데 데이터를 더 모을까요, 층을 더 쌓을까요, 아니면 하이퍼파라미터를 튜닝할까요?" 이때 "일단 다 해봐"라고 말하면 프로젝트는 망합니다. 자원은 유한하기 때문입니다. 오늘 배울 직교화(Orthogonalization)는 복잡한 문제 상황에서 무엇부터 해결해야 할지 알려주는 나침반입니다.

22.1 Overview

[핵심 목표]

이 단원은 머신러닝 프로젝트를 효율적으로 이끄는 전략적 사고방식을 다룹니다.

- **개념:** 시스템의 변수들을 서로 독립적으로(Orthogonal) 제어하여 원하는 효과를 정밀 타격하는 원리를 배웁니다.
- **진단:** 프로젝트 성공을 위한 4단계 가정(Chain of Assumptions)을 확립합니다.
- **도구:** 각 문제(Bias, Variance 등)를 해결하기 위한 '직교화된 도구'를 매핑합니다.
- **주의:** 조기 종료(Early Stopping)가 왜 직교화 원칙에 위배되는지 분석합니다.

22.1.1 Essential Terminology

용어	의미	비유
Orthogonalization	변수 간 독립성 확보	핸들은 방향만, 페달은 속도만 조절함.
Chain of Assumptions	순차적 해결 단계	1단계(Train) $\rightarrow$ 2단계(Dev) $\rightarrow$ 3단계(Test).
Orthogonal Tool	한 가지 문제만 해결하는 도구	Bigger Network (Bias만 잡음).

22.1.2 Core Concepts: 한 번에 하나씩

22.1.3 1. What is Orthogonalization? (직교화란?)

수학적으로 두 벡터가 직교(90도)하면, 한 벡터를 조절해도 다른 벡터에는 영향을 주지 않습니다. 엔지니어링에서도 마찬가지입니다. "하나의 다이얼은 하나의 기능만 조절해야 한다"는 원칙입니다.

[자동차 운전 비유]

- **Orthogonal (좋은 설계):**
  - 핸들 $\rightarrow$ 방향 조절 (속도 영향 X)
  - 페달 $\rightarrow$ 속도 조절 (방향 영향 X)
  - 운전하기 쉽습니다. 원하는 대로 제어 가능합니다.
- **Non-Orthogonal (나쁜 설계):**
  - 핸들을 꺾을 때마다 브레이크가 걸리고 라디오 볼륨이 커진다면?
  - 운전이 불가능합니다. 튜닝도 마찬가지입니다.

22.1.4 2. The Chain of Assumptions (4단계 진단)

머신러닝 프로젝트는 다음 4단계를 순서대로 통과해야 합니다. 앞 단계가 해결되지 않으면 뒷 단계는 의미가 없습니다.

1. **Fit Training Set:** 훈련 데이터에서 인간 수준 성능을 낸다. (Bias 문제)
2. **Fit Dev Set:** 훈련된 모델이 검증 데이터에서도 잘한다. (Variance 문제)
3. **Fit Test Set:** 검증 데이터 성능이 테스트 데이터와 비슷하다. (Data Mismatch)
4. **Perform in Real World:** 실제 사용자가 만족한다. (Cost Function 오류)

22.1.5 Deep Dive: 도구 매핑 (Tool Mapping)

각 단계에서 문제가 발생했을 때, 다른 단계에 부작용을 주지 않고 해당 문제만 해결하는 직교화된 도구를 써야 합니다.
Dr. Ng's Prescription Table

문제 (Problem)	직교화된 도구 (Good)	비추천 도구 (Bad)
1. High Bias (Train 성능 낮음)	**Bigger Network** Adam Optimizer	Early Stopping (Bias/Var 둘 다 건드림)
2. High Variance (Dev 성능 낮음)	**More Data** Regularization (L2, Dropout)	Early Stopping
3. Test Mismatch	Bigger Dev Set	-
4. Real World Fail	Change Cost Function	-

[Early Stopping의 딜레마]

Early Stopping은 학습을 중간에 멈춥니다.

- 비용함수 J 최소화 중단 $\rightarrow$ Bias 악화
- 가중치 W 증가 억제 $\rightarrow$ Variance 개선

두 가지 효과가 섞여 있어(Coupled), 문제가 생겼을 때 원인을 파악하기 어렵게 만듭니다. 직교화 관점에서는 "Bias는 네트워크 크기로 잡고, Variance는 정규화로 잡는 것"이 더 명확합니다.

22.1.6 Implementation: Automated Strategist

직교화는 코드가 아니라 판단 로직입니다. 이를 자동화된 진단 클래스로 구현해봅시다.

Listing 21: ML Project Diagnosis Logic

```
class MLStrategist:
    def __init__(self, human_err, train_err, dev_err, test_err):
        self.human = human_err
        self.train = train_err
        self.dev = dev_err
        self.test = test_err
        self.threshold = 0.02 # 2% □□ □□□ □□

    def diagnose(self):
        print("---␣Diagnosis␣Report␣---")

        # 1. Bias Check
        avoidable_bias = self.train - self.human
        if avoidable_bias > self.threshold:
```

```
            print("[Problem] High Bias (Underfitting)")
            print("[Action] Increase Network Size, Train Longer.")
            return # □ □□ □□ □□ □□ □□ □□ (Orthogonal)

        # 2. Variance Check
        variance = self.dev - self.train
        if variance > self.threshold:
            print("[Problem] High Variance (Overfitting)")
            print("[Action] Get More Data, Add Regularization.")
            return

        # 3. Mismatch Check
        mismatch = self.test - self.dev
        if mismatch > self.threshold:
            print("[Problem] Overfitting to Dev Set")
            print("[Action] Collect More Dev/Test Data.")
            return

        print("[Result] Good Job! Ready for Deployment.")

# --- □ □ ---
if __name__ == "__main__":
    # □ □ □ □ : □ □ □ □ □ □ □
    strategist = MLStrategist(0.01, 0.15, 0.16, 0.17)
    strategist.diagnose()
```

22.1.7 FAQ Pitfalls

Q. Cost Function은 언제 바꾸나요?
A. 모델의 수치적 성능(Accuracy)은 좋은데, 실제 앱 사용자들의 불만(Real World)이 많을 때입니다. 예를 들어, 고양이 분류기가 야한 사진을 고양이로 분류했다면 정확도가 높아도 치명적입니다. 이때는 Cost Function에 '야한 사진 패널티'를 추가해야 합니다. (과녁 자체를 수정)

Q. Bias를 잡으려고 Regularization을 쓰면 안 되나요?
A. 비추천합니다. Regularization은 모델을 단순하게 만들어 Variance를 줄이는 도구입니다. 부작용으로 Bias가 약간 높아질 수 있습니다. Bias를 잡을 때는 그냥 더 큰 네트워크(Bigger Network)를 쓰는 것이 부작용 없는(Orthogonal) 해결책입니다.

☒ 다음 단계 (Next Step)

이제 우리는 문제 해결의 순서와 전략을 알았습니다. 그런데 모델 A는 정확도가 높은데 속도가 느리고, 모델 B는 정확도는 조금 낮은데 속도가 빠릅니다. 도대체 무엇을 선택해야 할까요?

다음 시간에는 애매모호한 상황을 숫자 하나로 정리하여 의사결정 속도를 높이는 [Single Number Evaluation Metric]에 대해 배우겠습니다.

[단원 요약 (Cheat Sheet)]

1. **Orthogonalization:** 변수들을 독립적으로 제어하여 튜닝을 명확하게 만드는 전략.
2. **Sequence:** Train Fit $\rightarrow$ Dev Fit $\rightarrow$ Test Fit 순서로 해결한다.
3. **Bias Tool:** Bigger Network, Adam Optimizer.
4. **Variance Tool:** More Data, Regularization.

23 Precision Recall

☒ 지난 시간 복습 및 연결

지난 시간에 우리는 프로젝트의 방향을 잡는 '직교화' 전략을 배웠습니다. 하지만 방향을 잡았다고 끝이 아닙니다. 팀원이 두 개의 모델을 들고 왔습니다. "A는 정밀도가 높은데 재현율이 낮고, B는 정밀도는 낮은데 재현율이 높습니다. 뭘 쓸까요?" 여기서 머뭇거리면 프로젝트가 멈춥니다. 앤드류 응 교수는 "평가 지표는 하나여야 한다(Single Number Metric)"고 강조합니다. 그래야 수많은 실험 결과를 한 줄로 세우고, 1등을 바로 뽑을 수 있기 때문입니다.

23.1 Overview

[핵심 목표]

이 단원은 불균형 데이터에서 모델 성능을 정확히 평가하는 F1 Score를 다룹니다.

- **정의:** 정밀도(Precision)와 재현율(Recall)의 개념을 오차 행렬을 통해 익힙니다.
- **이유:** 왜 단순 평균이 아닌 조화 평균(Harmonic Mean)을 써야 하는지 증명합니다.
- **관계:** 임계값 변화에 따라 두 지표가 반대로 움직이는 Trade-off 관계를 파악합니다.
- **구현:** Python으로 직접 지표를 계산하고 분석합니다.

23.1.1 Essential Terminology: Confusion Matrix

구분	예측: Positive (1)	예측: Negative (0)
실제: Positive (1)	**TP** (정답)	**FN** (놓침/미검출)
실제: Negative (0)	**FP** (오해/거짓 알람)	**TN** (정답)

23.1.2 Core Concepts: 두 마리 토끼 잡기

23.1.3 1. Precision (정밀도)

질문: "모델이 찾은 것 중에 진짜는 얼마나 되는가?"

$$P = \frac{TP}{TP + FP}$$

[깐깐한 미식가]

"나는 맛없는 건 절대 안 먹어." (FP 싫어함) 맛있는 것(TP)을 좀 놓치더라도, 내 입에 들어오는 건 무조건 맛있어야 합니다. **활용:** 스팸 메일 분류 (정상 메일을 스팸통에 넣으면 치명적임).

23.1.4 2. Recall (재현율)

질문: "실제 존재하는 것 중에 모델이 얼마나 찾았는가?"

$$R = \frac{TP}{TP + FN}$$

[그물망 어선]

"쓰레기가 좀 섞여도 좋으니, 물고기는 다 잡아라." (FN 싫어함) 잡동사니(FP)가 걸려도 괜찮지만, 물고기(TP)를 놓치면 안 됩니다. **활용:** 암 진단 (암 환자를 정상이라 하면 생명이 위험함), 도둑 탐지.

23.1.5 Deep Dive: Why Harmonic Mean? (F1 Score)

왜 두 지표를 그냥 더해서 2로 나누면(산술 평균) 안 될까요?

바보 모델의 함정 암 환자가 1%인 데이터가 있습니다. 어떤 모델이 "무조건 암이다(1)"라고 예측한다고 합시다. (Recall = 1.0, Precision $\approx$ 0.01)

1. 산술 평균 (Arithmetic Mean):

$$\frac{P+R}{2} = \frac{0.01+1.0}{2} \approx \mathbf{0.5}$$

$\rightarrow$ 말도 안 되는 모델에게 50점이나 줍니다. 과대평가입니다.

2. 조화 평균 (Harmonic Mean, F1 Score):

$$F1 = \frac{2}{\frac{1}{P}+\frac{1}{R}} = 2\frac{P \times R}{P+R}$$

$$2\frac{0.01 \times 1.0}{0.01+1.0} \approx \mathbf{0.019}$$

$\rightarrow$ 둘 중 하나라도 낮으면 점수를 확 깎아버립니다. 이것이 우리가 원하는 평가 방식입니다.

23.1.6 Implementation: Metric Calculation

'scikit-learn'을 쓰면 쉽지만, 원리 이해를 위해 'numpy'로 직접 구현해 봅시다.

Listing 22: Precision

```python
import numpy as np

def calculate_metrics(y_true, y_pred):
    """
    y_true: 실제 정답 (0 or 1)
    y_pred: 모델 예측 (0 or 1)
    """
    # 벡터 연산으로 TP, FP, FN 계산 (Vectorized)
    TP = np.sum((y_true == 1) & (y_pred == 1))
    FP = np.sum((y_true == 0) & (y_pred == 1))
    FN = np.sum((y_true == 1) & (y_pred == 0))

    # 분모 0 방지 작은 값
    epsilon = 1e-7

    # Precision & Recall
    precision = TP / (TP + FP + epsilon)
    recall = TP / (TP + FN + epsilon)

    # F1 Score (Harmonic Mean)
    f1 = 2 * (precision * recall) / (precision + recall + epsilon)

    return precision, recall, f1

# --- 테스트 ---
if __name__ == "__main__":
    # 실제 정답 (1) 총 2, 정상 (0) 총 8
    y_true = np.array([0, 1, 0, 0, 1, 0, 0, 0, 0, 0])

    # 모델 예측: 암을 1 맞추고 하나, 정상을 1 틀림
    y_pred = np.array([0, 1, 0, 0, 0, 1, 0, 0, 0, 0])

    p, r, f1 = calculate_metrics(y_true, y_pred)

    print(f"Precision: {p:.2f}") # 0.50 (2 중에 맞춘 1 개)
    print(f"Recall:    {r:.2f}") # 0.50 (실제 2 중 1 개)
    print(f"F1 Score:  {f1:.2f}") # 0.50
```

23.1.7 FAQ Pitfalls

Q. Accuracy(정확도)는 언제 쓰나요?
A. 데이터 클래스 비율이 50:50으로 균형 잡혀 있을 때만 씁니다. 불균형 데이터(99:1)에서는 무조건 0으로 찍어도 정확도가 99%가 나오므로 무의미합니다.

Q. Recall이 Precision보다 훨씬 중요한 경우는요?
A. F1 Score는 두 지표를 1:1로 봅니다. Recall을 더 중요하게 보고 싶다면 F2 Score(Recall에 가중치)를 쓰면 됩니다. 반대는 F0.5 Score입니다.

☒ 다음 단계 (Next Step)

이제 우리는 성능을 평가하는 단일 숫자(F1 Score)를 얻었습니다. 그런데 현실에서는 성능뿐만 아니라 제약 조건도 있습니다. "정확도는 높아야 하지만, 실행 시간은 10ms 이내여야 한다."

이런 복잡한 요구사항을 어떻게 단일 지표로 정리할까요? 다음 시간에는 [Satisficing and Optimizing Metrics]를 통해, '최적화해야 할 것'과 '만족시켜야 할 것'을 구분하는 전략을 배웁니다.

[단원 요약 (Cheat Sheet)]

1. **Need:** 불균형 데이터에서는 정확도 대신 Precision/Recall을 봐야 한다.
2. **Trade-off:** Precision과 Recall은 반비례 관계다. 둘 다 높은 게 최고다.
3. **F1 Score:** 조화 평균이다. 극단적인 값(하나만 높은 경우)에 페널티를 주어 균형을 잡는다.
4. **Single Number:** 지표를 하나로 합쳐야 빠른 의사결정이 가능하다.

24 F1 Score Optimizing Metrics

☒ 지난 시간 복습 및 연결

지난 시간에 우리는 정밀도와 재현율을 F1 Score로 합치는 법을 배웠습니다. 그런데 현실 문제는 더 복잡합니다. **모델 A: 정확도 99%, 응답시간 1.5초 모델 B: 정확도 98%, 응답시간 0.05초** A는 성능은 좋지만 1.5초나 걸려서 아무도 안 쓸 겁니다. 그렇다고 정확도와 시간을 평균 낼 수도 없습니다(단위가 다름). 이런 딜레마를 해결하기 위해 앤드류 응 교수는 "만족(Satisficing) 지표와 최적화(Optimizing) 지표의 분리"를 제안합니다.

24.1 Overview

[핵심 목표]

이 단원은 여러 개의 목표가 충돌할 때, 우선순위를 정하는 전략적 프레임워크를 다룹니다.

- **구분:** 평가 지표를 '최대한 좋게 할 것(Optimizing)'과 '기준만 넘기면 되는 것(Satisficing)'으로 나눕니다.
- **규칙:** N개의 지표가 있다면, 1개만 최적화하고 나머지 $N-1$개는 제약 조건으로 둡니다.
- **구현:** 여러 모델 후보 중 최적의 모델을 자동으로 선별하는 파이썬 코드를 작성합니다.

24.1.1 Essential Terminology

구분	정의	예시
Optimizing Metric	다다익선. 무한히 좋아질수록 좋은 단 하나의 지표.	정확도(Accuracy), F1 Score
Satisficing Metric	임계값(Threshold)만 넘으면 통과(Pass).	실행 시간(Latency) $\leq$ 100ms

24.1.2 Core Concepts: 올림픽 달리기

24.1.3 1. The Rule of N Metrics

고려해야 할 지표가 3개(정확도, 속도, 메모리)라면 어떻게 해야 할까요? **"3마리 토끼를 다 잡으려다가는 다 놓칩니다."**
[올림픽 달리기와 도핑 테스트]

- **Optimizing (달리기 기록):** 0.01초라도 빠르면 무조건 좋습니다. 금메달의 기준입니다.
- **Satisficing (도핑 테스트):** 약물이 "아주 조금 검출됨"이나 "전혀 검출 안 됨"이나 똑같이 통과(Pass)입니다. 기준치만 안 넘으면 됩니다. 더 깨끗하다고 가산점을 주진 않습니다.

24.1.4 2. Mathematical Formulation (수학적 정의)

이 전략은 머신러닝 문제를 '제약 조건이 있는 최적화 문제'로 바꿉니다.

$$\text{Maximize } \textbf{Accuracy}$$

$$\text{subject to } \textbf{Latency} \leq 100ms$$

$$\text{and } \textbf{Memory} \leq 500MB$$

- Accuracy: 목적 함수 (Objective Function) $\rightarrow$ Optimizing
- Latency, Memory: 제약 조건 (Constraints) $\rightarrow$ Satisficing

[가중치 합의 함정]

"그냥 Score $=$ Accuracy $- 0.5 \times$ Latency 처럼 합치면 안 되나요?" **비추천합니다.** 1. 단위가 다릅니다(Accuracy는 %, Latency는 ms). 2. 사용자 경험은 비선형적입니다. 100ms가 넘으면 '렉 걸림'을 느껴 바로 앱을 끕니다. 이를 선형 식으로 표현하기 어렵습니다. Satisficing(Cut-off)이 훨씬 자연스럽습니다.

24.1.5 Implementation: Model Selector

여러 모델의 성능표를 입력받아 자동으로 최적 모델을 뽑는 로직을 구현합니다.

Listing 23: Auto Model Selector Logic

```
class ModelSelector:
    def __init__(self, optimize_key, constraints):
        """
␣␣␣␣␣␣␣␣optimize_key:␣□ □ □ □ ␣□ □ ␣□ (:␣'accuracy')
␣␣␣␣␣␣␣␣constraints:␣□ □ □ {:␣□ □ □ (,␣□ □ □ )}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣□ :␣{'latency':␣(100,␣'lt')}␣->␣less␣than␣100
␣␣␣␣␣␣␣␣"""
        self.opt_key = optimize_key
        self.constraints = constraints

    def select(self, models):
        # 1. Satisficing □ □  (Filtering)
        valid_models = []
        for m in models:
            is_valid = True
            for key, (thresh, op) in self.constraints.items():
                val = m[key]
                if op == 'lt' and val > thresh: is_valid = False
                if op == 'gt' and val < thresh: is_valid = False

            if is_valid:
                valid_models.append(m)

        if not valid_models:
            print("No␣models␣satisfied␣constraints!")
            return None

        # 2. Optimizing □ □  (Sorting)
        # □ □  □ □  □ □ □  □ □
        best_model = sorted(valid_models, key=lambda x: x[self.opt_key], reverse=True)[0]
        return best_model

# --- □ □  ---
if __name__ == "__main__":
    candidates = [
        {'id': 'A', 'acc': 0.99, 'lat': 1500}, # □ □  □ , □ □  □ □
        {'id': 'B', 'acc': 0.98, 'lat': 90},   # □ □  □ □ , □ □  (Pass)
        {'id': 'C', 'acc': 0.90, 'lat': 50},   # □ □  □ □ , □ □  □ □  (Pass)
        {'id': 'D', 'acc': 0.985, 'lat': 110}, # □ □ □  □ □  (Fail)
    ]

    # □ □ : Latency < 100ms □  □  □ □ □  Accuracy □ □ □
    constraints = {'lat': (100, 'lt')}
    selector = ModelSelector('acc', constraints)

    best = selector.select(candidates)
    print(f"Best␣Model:␣{best['id']}␣(Acc:␣{best['acc']},␣Lat:␣{best['lat']})")
    # A, D □ □ . B(0.98) vs C(0.90) -> B □ □ .
```

24.1.6 FAQ Pitfalls

Q. 만족 지표(Satisficing)가 너무 빡빡하면 어떡하나요?

A. 만약 'Latency < 10ms'로 걸었는데 통과하는 모델이 하나도 없다면, 하드웨어를 바꾸거나 비즈니스 요구사항(임계값)을 완화해야 합니다.

Q. 최적화 지표를 2개로 하면 안 되나요?
A. 안 됩니다. "정확도도 높고 속도도 빠른 것"을 찾으려 하면, A(정확도짱)와 B(속도짱) 사이에서 결정을 못 내립니다. 결국 둘을 합친 단일 지표를 만들거나, 하나를 제약 조건으로 돌려야 합니다.

☒ 다음 단계 (Next Step)

이제 평가 기준(Metric)까지 완벽하게 세팅했습니다. 모델을 열심히 개선하고 있는데, 문득 의문이 듭니다. "도대체 이 모델은 어디까지 좋아질 수 있는 걸까? 100%가 가능한가?"

이 질문에 답하기 위해서는 비교 대상, 즉 기준점(Baseline)이 필요합니다. 다음 시간에는 [Human-level Performance]를 통해, 나의 모델이 현재 어느 수준인지, 그리고 얼마나 더 발전할 여지가 있는지(Bayes Error) 가늠하는 법을 배웁니다.

[단원 요약 (Cheat Sheet)]

1. **Optimizing:** 최대한 좋게 만들어야 하는 단 하나의 지표 (예: Accuracy).
2. **Satisficing:** 기준선(Threshold)만 넘으면 되는 지표들 (예: Latency, Cost).
3. **Process:** Satisficing 지표로 필터링(Cut)하고, Optimizing 지표로 줄 세운다(Rank).
4. **Rule:** N개의 지표가 있다면 1개는 Optimizing, $N-1$개는 Satisficing이다.

25 Error Analysis

☒ 지난 시간 복습 및 연결

우리는 최적의 모델을 선택하는 방법을 배웠습니다. 하지만 선택된 모델도 목표 성능(예: 99%)에는 도달하지 못했을 수 있습니다. 이때 많은 엔지니어들이 "직감"에 의존합니다. "개가 고양이처럼 보이네? 개 데이터를 더 모으자", "흐려서 그런가? 포토샵 전처리를 하자." 하지만 이것이 수개월을 낭비하는 최악의 결정이 될 수 있습니다. 우리는 직감이 아니라 '데이터'가 말하게 해야 합니다. 오늘 배울 에러 분석은 가장 효율적인 성능 향상 경로를 알려주는 나침반입니다.

25.1 Overview

[핵심 목표]

이 단원은 틀린 데이터를 직접 분석하여 프로젝트의 우선순위를 정하는 전략을 다룹니다.

- **철학:** "추측하지 말고 확인하라(Don't guess, look)." 오분류된 데이터를 직접 눈으로 확인합니다.
- **천장 분석:** 특정 문제를 해결했을 때 성능이 얼마나 오를지 상한선(Ceiling)을 계산합니다.
- **라벨 오류:** 정답 라벨 자체가 틀린 경우(Incorrect Label), 이를 수정해야 할지 판단하는 기준을 배웁니다.
- **구현:** 에러 리포트를 자동으로 생성하는 Python 클래스를 작성합니다.

25.1.1 Essential Terminology

용어	의미	활용
Ceiling Analysis	성능 향상의 최대치 분석	"이거 고치면 몇 점 오르지?" 계산
Misclassified	오분류된 데이터	모델이 틀린 것만 모아놓은 집합
Incorrect Label	잘못된 정답지	사람이 실수로 라벨링을 잘못한 경우

25.1.2 Core Concepts: 데이터가 말하게 하라

25.1.3 1. The Philosophy: Don't Guess, Look

모델 정확도가 90%입니다. (에러율 10%). 팀원이 제안합니다. "흐릿한(Blurry) 사진 때문에 틀리는 것 같아요. 흐림 제거 모델을 만듭시다! (예상 소요: 3개월)" 이 제안을 수락해야 할까요? 천장 분석(Ceiling Analysis)을 해보기 전엔 모릅니다.

25.1.4 2. Ceiling Analysis (천장 분석)

Dev Set에서 모델이 틀린 샘플 100개를 무작위로 뽑아 엑셀에 정리합니다.
분석 결과 시뮬레이션

에러 원인 (Category)	비율 (Count)	Ceiling (예상 향상)
흐릿함 (Blurry)	5%	$10\% \times 0.05 = \mathbf{0.5}\%$
배경 노이즈 (Noise)	60%	$10\% \times 0.60 = \mathbf{6.0}\%$
고양이 닮은 개	35%	$10\% \times 0.35 = \mathbf{3.5}\%$

결론: 흐림 제거 모델을 완벽하게 만들어도 성능은 고작 0.5% 오릅니다. 3개월을 낭비할 뻔했습니다. 우리는 '배경 노이즈' 문제(6.0% 향상 가능)에 집중해야 합니다.

25.1.5 3. Incorrectly Labeled Data (라벨 오류)

데이터를 보다 보니, 모델은 맞았는데 사람이 정답을 잘못 달아놓은 경우를 발견했습니다. 고쳐야 할까요?

- **Training Set:** 딥러닝은 무작위 오류(Random Error)에 강합니다. 데이터가 많다면 무시해도 됩니다. (단, 체계적 오류는 수정 필수)
- **Dev/Test Set:** 에러 분석 표에 'Incorrect Label' 열을 추가합니다.
  - 만약 이 비율이 전체 에러의 상당수라면(예: 에러의 30%), 반드시 고쳐야 합니다.
  - **주의:** Dev와 Test는 항상 동시에 고쳐야 분포가 유지됩니다.

25.1.6 Implementation: Error Report Generator

에러 분석을 도와주는 자동화 도구입니다.

Listing 24: Error Analysis Tool

```
import pandas as pd
import numpy as np

class ErrorAnalyzer:
    def __init__(self, y_true, y_pred):
        self.y_true = np.array(y_true)
        self.y_pred = np.array(y_pred)
        # □□ □□□ □□
        self.error_indices = np.where(self.y_true != self.y_pred)[0]
        self.total_errors = len(self.error_indices)
        self.total_samples = len(y_true)

    def generate_report(self, manual_tags):
        """
␣␣␣␣␣␣␣␣manual_tags:␣{index:␣['Blurry',␣'Noise'],␣...}
␣␣␣␣␣␣␣␣"""
        counts = {}
        for tags in manual_tags.values():
            for tag in tags:
                counts[tag] = counts.get(tag, 0) + 1

        df = pd.DataFrame(list(counts.items()), columns=['Category', 'Count'])

        # Ceiling Analysis □□
        # □□ □□□ □□ □□ □□ □□□
        df['Ceiling␣(%)'] = (df['Count'] / self.total_samples) * 100

        # □□
        df = df.sort_values(by='Count', ascending=False)
        return df

# --- □□ ---
if __name__ == "__main__":
    # □□ □□□ □ 1000 □ □□ □□□ 100 □□
    y_true = np.zeros(1000)
    y_pred = np.zeros(1000)
    y_pred[:100] = 1 # □ 100 □□

    analyzer = ErrorAnalyzer(y_true, y_pred)

    # □□□ □□ □□ □□□□□ □□ □□( □□ )
    tags = {
        0: ['Blurry'], 1: ['Noise'], 2: ['Blurry', 'Noise'],
        # ... □□ () ...
        99: ['Noise']
    }
```

```
    # □ □ (: Blurry □ 5, Noise □ 60)

    # □ □ □  □ □
    # df = analyzer.generate_report(tags)
    # print(df)
```

25.1.7 FAQ Pitfalls

[Dev Set만 고치면 안 되나요?]

절대 안 됩니다. Dev Set의 라벨만 고치고 Test Set을 그대로 두면, 두 데이터셋의 분포가 달라집니다(Distribution Mismatch). 평가의 신뢰도가 깨집니다. 귀찮더라도 Dev와 Test는 한 몸처럼 다뤄야 합니다.

Q. 몇 개나 분석해야 하나요?

A. 보통 100개 정도면 충분한 통계적 인사이트를 얻을 수 있습니다. 혼자서 100개 보는 데 1 2시간이면 됩니다. 팀원들과 100개씩 나눠서 보면 더 좋습니다.

☒ 다음 단계 (Next Step)

에러 분석을 통해 '무엇'을 고쳐야 할지 알게 되었습니다. 그런데 분석 결과, "훈련 데이터와 검증 데이터의 분포가 너무 다르다"는 결론이 나오면 어떻게 해야 할까요? (예: 훈련은 고화질, 검증은 저화질)

이것은 단순한 과대적합과는 다른 차원의 문제입니다. 다음 시간에는 [Training vs Dev/Test Distribution Mismatch] 문제를 진단하고 해결하는 고급 전략인 'Training-Dev Set'의 개념에 대해 배우겠습니다.

[단원 요약 (Cheat Sheet)]

1. **Don't Guess:** 직감이 아닌 데이터를 보고 결정하라.
2. **Ceiling Analysis:** 특정 문제를 해결했을 때 얻을 수 있는 최대 이익(ROI)을 계산하라.
3. **Incorrect Label:** 전체 에러 중 비중이 높다면 수정하라. 단, Dev/Test를 동시에 수정해야 한다.
4. **Spreadsheet:** 엑셀 등을 활용해 팀원과 에러 원인을 공유하라.

26 Train-Dev Set Data Mismatch

☒ 지난 시간 복습 및 연결

지난 시간에 우리는 에러 분석을 통해 '무엇을 고칠지' 우선순위를 정했습니다. 그런데 만약 여러분이 수집한 20만 장의 '고화질 웹 이미지'로 학습시켰는데, 정작 서비스할 '저화질 모바일 이미지'에서는 모델이 전혀 동작하지 않는다면 어떨까요? 이것은 단순한 과대적합(Variance) 문제가 아닙니다. 공부한 책과 시험 과목이 아예 다른 상황, 즉 데이터 불일치(Data Mismatch)입니다. 오늘은 이 까다로운 문제를 해결하는 비밀 무기인 'Train-Dev Set'을 배웁니다.

26.1 Overview

[핵심 목표]

이 단원은 학습 데이터와 실전 데이터의 분포가 다를 때 발생하는 문제를 해결합니다.

- **진단:** 과대적합(Variance)과 데이터 불일치(Mismatch)를 구분하기 위해 Train-Dev Set을 도입합니다.
- **논리:** Train, Train-Dev, Dev 에러 간의 격차(Gap)를 분석하여 모델 상태를 판별합니다.
- **해결:** 인공 데이터 합성(Data Synthesis)을 통해 학습 데이터를 실전 분포에 맞추는 법을 배웁니다.

26.1.1 Essential Terminology

데이터셋	구성 (Source)	역할
Train Set	웹 이미지 (20만 장)	모델 파라미터 학습
Train-Dev Set	**웹 이미지 (일부 떼어냄)**	**Variance 진단용 (학습 X)**
Dev Set	모바일 이미지 (5천 장)	타겟 성능 검증 및 Mismatch 진단
Test Set	모바일 이미지 (5천 장)	최종 성능 평가

26.1.2 Core Concepts: 새로운 진단 도구

26.1.3 1. The Trap of Standard Split (함정)

웹 이미지 20만 장 + 모바일 이미지 1만 장이 있습니다.

- **나쁜 방법:** 전부 섞어서(Shuffle) 나눈다. → Dev Set의 95%가 웹 이미지가 됨. 실전(모바일) 성능을 측정할 수 없음.
- **좋은 방법:** Dev/Test는 오직 모바일 이미지로만 구성한다. (Target 고정). Train에는 웹 이미지를 몰아준다.

26.1.4 2. Introducing "Train-Dev Set"

위의 '좋은 방법'을 쓰면 Train과 Dev의 분포가 달라집니다. 이때 에러가 높으면 "과대적합 때문인가? 아니면 데이터가 달라서인가?"를 알 수 없습니다. 이를 구분하기 위해 Train-Dev Set을 만듭니다.

- 정의: Train Set에서 무작위로 떼어낸 일부 데이터. 학습에는 쓰지 않음.
- 특징: Train Set과 분포가 완벽히 동일함.

26.1.5 Deep Dive: The Logic of Diagnosis

이 표가 오늘 강의의 핵심입니다. 에러의 차이(Gap)가 어디서 벌어지는지 봐야 합니다.
Mismatch 진단 로직

비교 대상	Gap의 의미	진단명 (Diagnosis)
Train vs **Human**	Avoidable Bias	**High Bias** (모델이 너무 단순함)
Train-Dev vs **Train**	Variance Gap	**High Variance** (과대적합)
Dev vs **Train-Dev**	**Mismatch Gap**	**Data Mismatch** (분포 차이)
Test vs **Dev**	Overfitting to Dev	Dev Set 과적합

[시나리오 분석]

상황: Train Error 1%.

- **Case A:** Train-Dev 9%, Dev 10%.
  - Train과 Train-Dev(같은 분포) 사이에서 에러가 폭증했습니다.
  - **진단:** High Variance (과대적합). 정규화가 필요합니다.
- **Case B:** Train-Dev 1.5%, Dev 10%.
  - 같은 분포(Train-Dev)에서는 잘하는데, 다른 분포(Dev)에 가니 못합니다.
  - **진단:** Data Mismatch. 데이터를 합성하거나 더 모아야 합니다.

26.1.6 Implementation: Automated Diagnosis

자동으로 데이터를 분할하고 문제를 진단하는 코드를 작성합니다.

Listing 25: Data Mismatch Diagnosis Tool

```
import numpy as np
from sklearn.model_selection import train_test_split

class MismatchDiagnostician:
    def __init__(self, X_web, y_web, X_mobile, y_mobile):
        # 1. Target(Dev/Test)□ □□□□ □□
        self.X_dev, self.X_test, self.y_dev, self.y_test = train_test_split(
            X_mobile, y_mobile, test_size=0.5, random_state=1
        )

        # 2. Train □ Set □ □□□ □□
        X_train_all, y_train_all = X_web, y_web

        # 3. □□ [] Train □□ Set Train-Dev □ Set □□□ □□ ( □□ )
        self.X_train, self.X_train_dev, self.y_train, self.y_train_dev = train_test_split(
            X_train_all, y_train_all, test_size=0.02, random_state=1
        )

    def diagnose(self, train_err, train_dev_err, dev_err):
        print("---␣Diagnosis␣Report␣---")
        variance_gap = train_dev_err - train_err
        mismatch_gap = dev_err - train_dev_err

        print(f"Variance␣Gap␣(TrainDev␣-␣Train):␣{variance_gap:.2%}")
        print(f"Mismatch␣Gap␣(Dev␣-␣TrainDev):␣␣␣{mismatch_gap:.2%}")

        if variance_gap > mismatch_gap:
            print(">>␣Diagnosis:␣High␣Variance␣(Overfitting)")
            print(">>␣Action:␣More␣Data,␣Regularization,␣Dropout")
```

```
        else:
            print(">> Diagnosis: Data Mismatch (Distribution Shift)")
            print(">> Action: Artificial Data Synthesis, Collect Target Data")

# --- □ □ ---
if __name__ == "__main__":
    # □ □ □ □ □ ( □ 5, □ □ □ □ 2)
    X_w, y_w = np.zeros((50000, 10)), np.zeros(50000)
    X_m, y_m = np.zeros((2000, 10)), np.zeros(2000)

    doctor = MismatchDiagnostician(X_w, y_w, X_m, y_m)

    # □ □ □ □ : Train 1%, Train-Dev 1.5%, Dev 10%
    doctor.diagnose(0.01, 0.015, 0.10)
```

26.1.7 FAQ Pitfalls

Q. Train-Dev Set을 Dev Set에서 떼어내면 안 되나요?
A. 절대 안 됩니다. 그러면 Train-Dev와 Dev의 분포가 같아집니다. Train-Dev는 반드시 Train Set의 부분집합이어야 "학습 데이터 분포에서의 일반화 성능"을 측정할 수 있습니다.

Q. Data Mismatch 해결책인 '데이터 합성'은 어떻게 하나요?
A. 깨끗한 음성(Train)에 자동차 소음(Noise)을 섞어서 시끄러운 음성(Dev와 비슷함)을 만드는 식입니다. 단, 너무 적은 종류의 소음만 반복해서 쓰면 모델이 그 소음 패턴에 과적합될 수 있으니 주의해야 합니다.

☒ 다음 단계 (Next Step)

이것으로 앤드류 응 교수의 '머신러닝 프로젝트 구조화 전략' 파트를 마칩니다. 여러분은 이제 단순히 코딩만 하는 엔지니어가 아니라, 프로젝트의 방향을 지휘하는 전략가(Strategist)가 되었습니다.

다음 시간부터는 다시 모델링의 세계로 돌아옵니다. 컴퓨터 비전(Computer Vision)의 혁명을 일으킨 [Convolutional Neural Networks (CNN)]의 기초부터 심화까지, 이미지 처리의 마법을 수학적으로 파헤쳐 보겠습니다.

[단원 요약 (Cheat Sheet)]

1. **Problem:** 학습 데이터(Web)와 실전 데이터(Mobile)의 분포가 다르면 성능이 떨어진다.
2. **Tool: Train-Dev Set** (Train과 분포는 같으나 학습엔 안 씀).
3. **Logic:** Train-Dev 에러가 낮고 Dev 에러가 높다면 **Data Mismatch**다.
4. **Action:** 인공 데이터 합성(Synthesis) 등을 통해 Train을 Dev스럽게 만들어라.

27 Transfer Learning Multi-task Learning

☒ 지난 시간 복습 및 연결

지금까지 우리는 모델을 밑바닥부터(Scratch) 학습시키는 법을 배웠습니다. 하지만 현실 세계의 엔지니어는 "아무것도 없는 상태에서 시작하지 않습니다." 책을 읽을 때마다 '가나다라'부터 다시 배우지 않듯, AI 모델도 남이 이미 학습해둔 지식(Knowledge)을 빌려와서 자신의 문제를 해결할 수 있습니다. 이것이 전이 학습(Transfer Learning)이며, 현대 딥러닝 성공의 90%는 여기에 기인합니다.

27.1 Overview

[핵심 목표]

이 단원은 '데이터 부족'을 해결하고 학습 효율을 극대화하는 두 가지 패러다임을 다룹니다.

- **Transfer Learning:** 대규모 데이터(ImageNet)로 학습된 모델을 가져와 내 문제(X-ray)에 적용하는 법을 배웁니다.
- **Fine-tuning:** 가중치를 고정(Freeze)하거나 미세 조정(Unfreeze)하는 단계별 전략을 익힙니다.
- **Multi-task Learning:** 하나의 모델이 여러 작업을 동시에 수행하며 지능을 높이는 원리를 이해합니다.
- **구현:** Keras를 활용해 Pre-trained Model을 로드하고 커스터마이징하는 코드를 작성합니다.

27.1.1 Essential Terminology

용어	의미	비유
Transfer Learning	지식 전이 (A $\rightarrow$ B)	영어 잘하는 사람이 불어도 빨리 배움.
Pre-trained Model	사전 학습된 모델 (Source)	이미 박사 학위를 받은 전문가.
Fine-tuning	미세 조정	전문가에게 우리 회사의 업무 매뉴얼만 가르침.
Multi-task Learning	동시 학습 (A & B)	수학과 물리를 동시에 배우면 시너지가 남.

27.1.2 Core Concepts: 지식의 재활용

27.1.3 1. Transfer Learning (전이 학습)

데이터가 풍부한 Task A(Source)에서 배운 지식을 데이터가 적은 Task B(Target)로 옮깁니다.

- **앞단 (Early Layers):** 엣지, 곡선, 질감 등 보편적인 특징을 배웁니다. (재활용 가능)
- **뒷단 (Later Layers):** 구체적인 사물(고양이, 자동차)을 배웁니다. (새로 학습 필요)

27.1.4 2. Multi-task Learning (다중 작업 학습)

하나의 신경망이 여러 작업(Task A, B, C)을 동시에 수행합니다.

- **Shared Layers:** 모든 작업에 공통적으로 필요한 저수준 특징(Low-level features)을 공유합니다.
- **효과:** 서로 다른 작업들이 일종의 노이즈(Regularization) 역할을 하여 과대적합을 막고 일반화 성능을 높입니다.
- **예시:** 자율주행 (표지판 인식 + 신호등 인식 + 보행자 감지).

27.1.5 Deep Dive: Fine-tuning Strategies

"데이터가 얼마나 있느냐"에 따라 전략이 달라집니다.
데이터 규모별 전략

1. **Small Data (데이터 매우 적음):**
   - 전략: Backbone 전체 고정 (Freeze).
   - 행동: 마지막 분류기(Head)만 떼어내고 새로 학습시킵니다.
2. **Medium Data (적당함):**
   - 전략: 앞단 일부 고정, 뒷단 일부 해제 (Fine-tuning).
   - 행동: 상위 층(Later Layers)의 가중치를 미세하게 업데이트합니다.
3. **Big Data (데이터 많음):**
   - 전략: 전체 재학습 (Retrain All).
   - 행동: 사전 학습된 가중치를 초기값(Initialization)으로만 쓰고 전체를 다 학습합니다.

27.1.6 Implementation: Transfer Learning with Keras

MobileNetV2를 가져와서 커스텀 분류기를 만드는 코드입니다.

Listing 26: Transfer Learning Pipeline

```
import tensorflow as tf
from tensorflow.keras import layers, models

def build_transfer_model(input_shape, num_classes):
    # 1. Load Pre-trained Model (MobileNetV2)
    # include_top=False: □ 1000 □□□ □□□□ (Head)□ □□
    # weights='imagenet': □□ ImageNet □□□□ □□□□ □□
    base_model = tf.keras.applications.MobileNetV2(
        input_shape=input_shape,
        include_top=False,
        weights='imagenet'
    )

    # 2. Freeze the Base Model □□ (!)
    # □□□ □ □ □□□ □□□□□ □□□□ □□□□ □□
    base_model.trainable = False

    # 3. Add Custom Head
    model = models.Sequential([
        base_model,
        layers.GlobalAveragePooling2D(), # □□□□ □□□□ □□
        layers.Dropout(0.2),             # □□□□ □□
        layers.Dense(num_classes, activation='softmax') # □ □□□ □□ □□□
    ])

    return model

# --- □□ □ Fine-tuning ---
if __name__ == "__main__":
    model = build_transfer_model((160, 160, 3), 10)

    # □□ 1: □ Head □□ (□ Base □□)
    model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['acc'])
    # model.fit(...)

    # □□ 2: Fine-tuning (□ Base □□□ □)
    base_model = model.layers[0]
    base_model.trainable = True

    # □□ □100□□ □□ □□, □□□ □□□ □□
    fine_tune_at = 100
```

```
    for layer in base_model.layers[:fine_tune_at]:
        layer.trainable = False

    # □□ : □□ □□ □□□ □□□□ □□ □□ (1/10 ~ 1/100) □□□ □
    model.compile(optimizer=tf.keras.optimizers.RMSprop(lr=1e-5),
                  loss='categorical_crossentropy', metrics=['acc'])
    # model.fit(...)
```

27.1.7 FAQ Pitfalls

[처음부터 Fine-tuning을 하면 안 되나요?]

절대 안 됩니다. 새로 붙인 Head(분류기)는 랜덤 초기화 상태라 엉뚱한 오차(Gradient)를 뿜어냅니다. Base Model을 고정하지 않으면, 이 큰 오차 때문에 잘 학습되어 있던 Base Model의 가중치가 다 망가져버립니다 (Catastrophic Forgetting). 반드시 Head를 먼저 학습시켜 안정화한 뒤, Base Model을 풀어야 합니다.

Q. 입력 이미지 크기가 달라도 되나요?

A. CNN의 특성상 가능은 하지만, 성능을 위해 사전 학습 모델이 사용했던 크기(예: 224x224)로 리사이징(Resize)해서 넣는 것을 권장합니다.

☒ 다음 단계 (Next Step)

이것으로 '머신러닝 프로젝트 구조화 전략' 파트를 마칩니다. 여러분은 이제 데이터를 다루는 전략부터, 남의 지식을 빌려오는 전략까지 모두 갖춘 전략가가 되었습니다.

다음 챕터부터는 딥러닝을 더욱 깊이 있게 만드는 [End-to-End Deep Learning]의 개념과, 이것이 전통적인 파이프라인 방식과 어떻게 다른지 비교 분석하며 시작해 보겠습니다. 수고하셨습니다.

[단원 요약 (Cheat Sheet)]

1. **Transfer Learning:** 빅데이터 모델(Source)을 소량 데이터 문제(Target)에 재활용한다.
2. **Freeze:** 초기 학습 시에는 Backbone을 고정하고 Head만 학습한다.
3. **Fine-tune:** 데이터가 충분하면 Backbone의 일부를 풀어 미세 조정한다. (Low LR 필수)
4. **Multi-task:** 여러 작업을 동시에 배우면 일반화 성능이 좋아진다.

28 End-to-End Learning

☒ 지난 시간 복습 및 연결

지난 시간까지 우리는 전이 학습과 다중 작업 학습을 통해 기존 지식을 활용하는 법을 배웠습니다. 이제 딥러닝이 가져온 가장 거대한 패러다임의 변화, 엔드투엔드(End-to-End) 딥러닝에 대해 이야기할 시간입니다. 과거에는 음성 인식을 위해 음향학, 음성학 등 수많은 파이프라인을 조립했습니다. 하지만 딥러닝은 이 모든 단계를 건너뛰고, "입력에서 출력으로 직행하는 고속도로"를 뚫어버렸습니다. 이것이 언제나 정답일까요?

28.1 Overview

[핵심 목표]

이 단원은 중간 단계를 생략하고 데이터만으로 학습하는 E2E 딥러닝의 장단점을 분석합니다.

- **정의:** 전통적인 파이프라인(Pipeline) 방식과 E2E 방식의 구조적 차이를 구분합니다.
- **장점:** 인간의 편향(Hand-designed components)을 제거하여 최적의 성능을 내는 원리를 이해합니다.
- **단점:** 왜 막대한 양의 데이터가 필요한지(Data Hungry), 왜 디버깅이 어려운지 파악합니다.
- **결정:** 언제 E2E를 쓰고, 언제 파이프라인을 써야 하는지 결정 기준을 세웁니다.

28.1.1 Essential Terminology

용어	의미	비유
End-to-End (E2E)	입력 $\rightarrow$ 출력으로 직행하는 단일 모델	직항 비행기 (중간 경유지 없음)
Pipeline	여러 모듈을 순차적으로 연결한 시스템	경유 비행기 (A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D)
Hand-engineered	사람이 직접 설계한 특징/규칙	수제작 부품 (장인의 손길)

28.1.2 Core Concepts: 직행 고속도로

28.1.3 1. Pipeline vs End-to-End

예시: 음성 인식 (Speech Recognition)

- **Traditional Pipeline:**

  $$\text{Audio} \rightarrow \text{MFCC(특징 추출)} \rightarrow \text{Phoneme(음소)} \rightarrow \text{Word} \rightarrow \text{Text}$$

  각 단계마다 전문가의 지식(음성학 등)이 필요합니다.

- **End-to-End Deep Learning:**

  $$\text{Audio} \rightarrow [\text{Deep Neural Network}] \rightarrow \text{Text}$$

  중간 단계(음소 등)를 명시적으로 가르치지 않습니다. 데이터만 충분하면 신경망이 알아서 최적의 내부 표현을 찾아냅니다.

28.1.4 2. The Key Idea: Let the Data Speak

전통적 방식에는 "음소를 먼저 찾아야 해"라는 인간의 가정(Bias)이 들어갑니다. 하지만 데이터가 충분하다면, 신경망은 음소보다 더 효율적인 자신만의 방식을 찾아낼 수 있습니다. E2E는 데이터가 스스로 최적의 처리 과정을 설계하도록 허용하는 것입니다.

28.1.5 Deep Dive: Pros & Cons

무조건 E2E가 좋은 것은 아닙니다. 명확한 트레이드오프가 존재합니다.

End-to-End 장단점 분석 **장점 (Pros):**

- **Data-driven:** 인간의 선입견에 갇히지 않고 데이터 패턴 자체를 학습하므로, 데이터가 많을수록 성능 상한선이 높습니다.
- **Simplicity:** 복잡한 파이프라인을 설계하고 유지보수할 필요가 없습니다. 신경망 하나만 관리하면 됩니다.

단점 (Cons):

- **Data Hungry:** (x, y) 쌍 데이터가 엄청나게 많이 필요합니다. (파이프라인은 각 모듈별로 적은 데이터로 학습 가능).
- **Black Box:** 왜 틀렸는지 설명하기 어렵습니다. (파이프라인은 "음소 인식에서 틀렸군" 하고 알 수 있음).

28.1.6 Example: Date Formatting

날짜 형식을 변환하는 간단한 예제로 E2E의 철학을 봅니다. Input: "20th Jan. 2023" $\rightarrow$ Output: "2023-01-20"

Listing 27: E2E Logic Overview

```
# Traditional Approach (Rule-based)
def parse_date(date_str):
    # □ □ □  □ □ (Regex) □ □  □ □
    # "Jan" -> 01, "February" -> 02 ...
    # □ □  □ □ , □ □  □ □  □  □ □ □
    pass

# End-to-End Approach
# □ □ □  □ □  □  □ □ □ , □ □ □ (x, y)□  □ □ □ □ □ .
x_data = ["25th␣Dec␣2022", "December␣25,␣2022", "12/25/22"]
y_data = ["2022-12-25",    "2022-12-25",        "2022-12-25"]

# □ □ □ □  □ 10 □ □  □ □ □ ,
# □ □ (Seq2Seq □ )□  "Dec□ " □ □ "12" □ □  □ □ □  □ □ □ □ □ .
# □ □ □  □ □  □ □ □ □  □ □ □ , □ □ □ □  □ □ □  □ □ □ □ .
model.fit(x_data, y_data)
```

28.1.7 FAQ Pitfalls

Q. 자율주행도 E2E로 하나요?

A. 초기에는 시도했지만(NVIDIA 등), 현재는 안전 때문에 파이프라인을 선호합니다. "이미지 $\rightarrow$ 핸들 조향"으로 바로 가면, 사고가 났을 때 왜 핸들을 꺾었는지 알 수 없어 디버깅과 책임 소재 파악이 불가능하기 때문입니다. (최근엔 다시 E2E 비중이 늘어나는 추세이긴 합니다.)

Q. 데이터가 적을 때 E2E를 쓰면 안 되나요?

A. 네, 성능이 처참할 수 있습니다. 데이터가 적을 때는 인간의 지식(Hand-engineered features)을 주입해주는 파이프라인 방식이 훨씬 효율적입니다.

☒ 다음 단계 (Next Step)

이것으로 앤드류 응 교수의 '머신러닝 프로젝트 구조화 전략(Structuring ML Projects)' 파트(Part 3)를 모두 마칩니다. 이제 여러분은 딥러닝의 기초 이론부터 성능 향상 기법, 그리고 프로젝트를 지휘하는 전략까지 모두 갖췄습니다.

다음 시간부터는 딥러닝의 꽃이자 가장 널리 쓰이는 분야인 [Part 4. Convolutional Neural Networks (CNN)]의 세계로 들어갑니다. 이미지를 처리하는 컴퓨터의 시각을 정복해보겠습니다. 기대하십시오.

[단원 요약 (Cheat Sheet)]

1. **E2E:** 중간 단계 없이 입력에서 출력으로 바로 매핑하는 딥러닝 방식.
2. **Pros:** 데이터가 충분하면 인간보다 더 효율적인 특징을 찾아낸다.
3. **Cons:** 막대한 양의 라벨링 데이터가 필요하다. 설명력이 부족하다.
4. **Decision:** 데이터 양이 적거나 안전/설명이 중요한 분야는 파이프라인을 쓴다.

29 CNN: Convolution Padding

☒ 지난 시간 복습 및 연결

우리는 지금까지 딥러닝 프로젝트를 전략적으로 지휘하는 법(Part 3)을 배웠습니다. 이제 딥러닝이 가장 눈부신 성과를 낸 컴퓨터 비전(Part 4)의 세계로 들어갑니다. 만약 고화질 이미지를 기존의 FC(Fully Connected) Layer에 넣으면 어떻게 될까요? 파라미터가 수십억 개로 폭발하여 계산이 불가능해집니다. 우리에겐 이미지의 지역적 특징을 효율적으로 추출하는 새로운 도구가 필요합니다. 바로 합성곱(Convolution)입니다.

29.1 Overview

[핵심 목표]

이 단원은 CNN을 구성하는 가장 기초적인 '레고 블록' 세 가지를 마스터합니다.

- **Convolution:** 필터를 슬라이딩하며 특징을 추출하는 기본 연산.
- **Padding:** 이미지 가장자리 정보 손실을 막고 크기를 유지하는 기법.
- **Strides:** 필터 이동 간격을 조절하여 출력 크기를 줄이는 기법.
- **Formula:** 입력 크기, 필터, 패딩, 스트라이드가 주어졌을 때 출력 크기를 계산하는 공식을 암기합니다.

29.1.1 Essential Terminology

용어	기호	설명
Filter / Kernel	$f \times f$	이미지를 훑는 작은 윈도우. 학습 대상 파라미터(W).
Padding	p	입력 이미지 테두리에 덧대는 가짜 픽셀(0).
Stride	s	필터가 한 번에 이동하는 칸 수(보폭).
Feature Map	-	합성곱 연산의 결과물(출력 이미지).

29.1.2 Core Concepts: CNN의 레고 블록

29.1.3 1. The Convolution Operation ($*$)

FC Layer가 이미지 전체를 한 번에 본다면, 합성곱은 작은 '손전등'으로 이미지를 훑는 것과 같습니다.

- 과정: 3×3 필터를 이미지 좌측 상단에 겹쳐 놓고, 겹치는 숫자끼리 곱해서 더합니다(내적). 그 결과값 하나가 출력의 픽셀 하나가 됩니다. 옆으로 한 칸씩 이동하며 반복합니다.
- 의미: 필터의 값에 따라 수직선, 수평선 같은 특정 패턴이 있는 위치를 찾아냅니다.

29.1.4 2. Padding (p)

합성곱을 하면 이미지가 점점 작아집니다 ($6 \times 6 \rightarrow 4 \times 4 \rightarrow \dots$). 또한 가장자리 픽셀은 필터가 덜 지나가서 정보가 소실됩니다. 이를 막기 위해 테두리에 0을 채웁니다.

- Valid Padding ($p = 0$): 패딩 없음. 크기가 줄어듦.
- Same Padding: 입력과 출력의 크기가 같아지도록 p를 설정함.

$$p = \frac{f-1}{2} \quad (\text{단, } s = 1)$$

29.1.5 3. Strides (s)

필터를 한 칸씩($s = 1$)이 아니라 두 칸씩($s = 2$) 듬성듬성 이동합니다.

- 효과: 출력 크기가 대략 $1/s$ 배로 줄어듭니다 (Downsampling).
- 용도: 계산량을 줄이거나 넓은 영역을 요약해서 볼 때 씁니다.

29.1.6 Deep Dive: The Golden Formula (만능 공식)

이 공식은 CNN 아키텍처를 설계하거나 논문을 읽을 때 필수입니다. 무조건 암기하십시오.
출력 크기 계산 공식 입력 크기가 $n \times n$ 일 때, 출력 크기는 다음과 같습니다.

$$n_{out} = \left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor$$

- n: 입력 크기
- p: 패딩 크기
- f: 필터 크기
- s: 스트라이드
- $\lfloor \cdot \rfloor$: 바닥 함수 (소수점 내림)

[계산 예제]

상황: 입력 7×7 ($n = 7$), 필터 3×3 ($f = 3$), 패딩 없음 ($p = 0$), 스트라이드 2 ($s = 2$).
계산:

$$n_{out} = \left\lfloor \frac{7 + 2(0) - 3}{2} + 1 \right\rfloor = \left\lfloor \frac{4}{2} + 1 \right\rfloor = \lfloor 3 \rfloor = 3$$

결과: 출력은 3×3 크기가 됩니다.

29.1.7 Implementation Perspective: 3D Volume

실제 이미지는 컬러(RGB)이므로 3차원($H \times W \times C$)입니다.

- Rule: 필터의 채널 수(깊이)는 입력의 채널 수와 항상 같아야 합니다.
- 예시: 입력이 $6 \times 6 \times$ **3** 이면, 필터는 $3 \times 3 \times$ **3** 이어야 합니다.
- 다중 필터: 만약 이런 필터를 10개 쓴다면? 출력은 $4 \times 4 \times$ **10** 이 됩니다.

[필터 개수 = 출력 채널 수]

CNN 층을 지난 뒤 데이터의 깊이(Depth)는 입력의 깊이가 아니라 필터의 개수에 의해 결정됩니다. 이것이 채널 수를 조절하는 핵심 메커니즘입니다.

☒ 다음 단계 (Next Step)

우리는 CNN의 기초 연산(Conv, Padding, Stride)을 마스터했습니다. 하지만 이것만으로는 부족합니다. 이미지의 크기를 더 과감하게 줄이면서도 중요한 정보(최대값)만 남기는 풀링(Pooling) 계층이 필요합니다.

다음 시간에는 Max Pooling과 Average Pooling에 대해 배우고, 드디어 이 모든 블록을 조립하여 첫 번째 완전한 CNN 모델(LeNet-5)을 만들어보겠습니다.

[단원 요약 (Cheat Sheet)]

1. **Convolution:** 필터를 슬라이딩하며 지역적 특징을 추출한다.
2. **Padding:** 가장자리 손실을 막고 크기를 유지한다 (Same Padding).
3. **Stride:** 이동 간격을 넓혀 크기를 줄인다.
4. **Formula:** $n_{out} = \lfloor \frac{n+2p-f}{s} + 1 \rfloor$.

30 Pooling Layers

☒ 지난 시간 복습 및 연결

지난 시간에 우리는 합성곱(Conv) 연산으로 이미지의 특징을 추출하는 법을 배웠습니다. 하지만 합성곱 층만 계속 쌓으면 연산량이 너무 많아지고, 모델이 이미지의 미세한 변화(1픽셀 이동 등)에 너무 민감해집니다. 우리는 "중요한 특징만 남기고, 크기는 줄여서 효율적으로" 처리하고 싶습니다. 이 두 마리 토끼를 잡는 기술이 바로 풀링(Pooling)입니다.

30.1 Overview

[핵심 목표]

이 단원은 CNN의 핵심 구성 요소인 풀링 층의 원리와 종류를 다룹니다.

- **다운샘플링:** 이미지 크기(n_H, n_W)를 줄여 계산량을 낮추는 원리를 이해합니다.
- **불변성:** 풀링이 어떻게 평행 이동에 대한 강건함(Invariance)을 제공하는지 배웁니다.
- **비교:** 가장 널리 쓰이는 Max Pooling과 과거에 쓰였던 Average Pooling을 비교합니다.
- **특징:** 풀링 층에는 학습 파라미터(W, b)가 없다는 점을 명심합니다.

30.1.1 Essential Terminology

용어	의미	핵심 역할
Max Pooling	영역 내 최댓값 선택	가장 강한 특징만 남김 (대세).
Average Pooling	영역 내 평균값 계산	정보를 부드럽게 요약함.
Invariant	불변성	입력이 조금 바뀌어도 출력은 변하지 않음.
Hyperparameters	f(필터 크기), s(스트라이드)	풀링 동작을 결정하는 설정값.

30.1.2 Core Concepts: 요약의 기술

30.1.3 1. Max Pooling (최대 풀링)

가장 널리 쓰이는 방식입니다. 필터 영역($f \times f$) 내에서 가장 큰 숫자 하나만 골라냅니다.

- 동작: 2×2 윈도우로 이미지를 훑으며 최댓값을 뽑습니다.
- 의미: "이 구역에 고양이 눈(특징)이 있는가?" $\rightarrow$ YES (높은 값). 정확히 어디(좌상단? 우하단?)에 있는지는 중요하지 않습니다. 존재 유무만 강조합니다.

30.1.4 2. Average Pooling (평균 풀링)

필터 영역 내의 숫자를 모두 더해 평균을 냅니다.

- 의미: 특징을 부드럽게(Smoothing) 만듭니다.
- 용도: 과거에는 많이 썼으나 최근에는 잘 안 씁니다. 단, 모델의 맨 마지막단(Global Average Pooling)에서는 여전히 유용합니다.

30.1.5 Deep Dive: Channel Independence

이 부분이 합성곱(Convolution)과 가장 헷갈리는 지점입니다.
채널 독립성의 법칙

- Convolution: 입력 채널(RGB 3개)을 모두 합쳐서(Sum) 하나의 숫자로 만듭니다. 채널 수가 변합니다.
- Pooling: 각 채널을 독립적으로(Independently) 처리합니다.
  - 입력: $32 \times 32 \times \mathbf{10}$
  - 풀링: $16 \times 16 \times \mathbf{10}$
  - **결과:** 높이/너비는 줄지만, **채널 수(깊이)는 그대로 유지**됩니다.

[파라미터 개수는 몇 개?]
풀링 층에는 학습해야 할 가중치(W)나 편향(b)이 있을까요? **정답: 0개입니다.** 풀링은 우리가 정해준 규칙(f, s, Max/Avg)대로만 계산하는 고정된 함수입니다. 역전파 때 업데이트될 대상이 없습니다.

30.1.6 Implementation: NumPy Pooling

원리 이해를 위해 4중 루프를 사용하여 직접 구현해 봅니다.

Listing 28: Max Average Pooling Implementation

```
import numpy as np

class Pooling:
    def __init__(self, f=2, s=2, mode='max'):
        self.f = f
        self.s = s
        self.mode = mode

    def forward(self, A_prev):
        """
␣␣␣␣␣␣␣␣A_prev:␣(m,␣n_H,␣n_W,␣n_C)
␣␣␣␣␣␣␣␣"""
        (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape

        # □□ □□ □□ □□( □□ )
        n_H = int((n_H_prev - self.f) / self.s) + 1
        n_W = int((n_W_prev - self.f) / self.s) + 1
        n_C = n_C_prev # □□ □□ □□□!

        A = np.zeros((m, n_H, n_W, n_C))

        for i in range(m):              # □□□ □□
            for h in range(n_H):        # □□ □□
                for w in range(n_W):    # □□ □□
                    for c in range(n_C):# □□ □□□()

                        # □□□ □□□□
                        vert_start = h * self.s
                        vert_end   = vert_start + self.f
                        horiz_start= w * self.s
                        horiz_end  = horiz_start + self.f

                        slice_A = A_prev[i, vert_start:vert_end, horiz_start:horiz_end, c]

                        if self.mode == 'max':
                            A[i, h, w, c] = np.max(slice_A)
                        elif self.mode == 'average':
                            A[i, h, w, c] = np.mean(slice_A)
        return A

# --- □□ ---
if __name__ == "__main__":
    # 4x4 □□□, □□ □1
    img = np.array([[[[1],[3],[2],[1]],
```

```
                    [[2],[9],[1],[1]],
                    [[1],[3],[2],[3]],
                    [[5],[6],[1],[2]]]]) # shape (1,4,4,1)

pool = Pooling(f=2, s=2, mode='max')
out = pool.forward(img)

print("Input:\n", img[0,:,:,0])
print("Max␣Pool␣Output:\n", out[0,:,:,0])
# □ □ : [[9, 2], [6, 3]]
```

30.1.7 FAQ Pitfalls

Q. 풀링을 하면 정보가 사라지는데 괜찮나요?
A. 네, 그게 목적입니다. 불필요한 배경이나 노이즈는 버리고, "여기 특징이 있다!"(Max Value)라는 핵심 정보만 남겨서 다음 층으로 전달하는 것이 CNN의 추상화 과정입니다.

Q. $f = 3, s = 2$ 같은 건 언제 쓰나요?
A. 보통은 $f = 2, s = 2$ (크기 절반 축소)가 국룰입니다. 하지만 겹치는 영역을 두고 싶을 때(Overlapping Pooling) $f = 3, s = 2$를 쓰기도 합니다 (예: AlexNet).

☒ 다음 단계 (Next Step)

이제 우리는 CNN을 만드는 3대 요소인 Convolution, ReLU(Activation), Pooling을 모두 배웠습니다. 레고 블록이 다 모였습니다.

이제 이것들을 어떻게 조립해야 할까요? 다음 시간에는 이 블록들을 결합하여 숫자 필기체(MNIST)를 인식하는 전설적인 CNN 아키텍처, [LeNet-5 Example]을 통해 첫 번째 완전한 CNN 모델을 구축해보겠습니다.

[단원 요약 (Cheat Sheet)]

1. **Max Pooling:** 영역 내 최댓값만 남겨 특징을 강조한다. (가장 많이 씀)
2. **Dimension:** 가로/세로는 줄어들지만, 채널 수(n_C)는 유지된다.
3. **Parameters:** 학습할 가중치(W)가 없다. (Parameter-free)
4. **Effect:** 연산량을 줄이고, 이동 불변성(Invariance)을 얻는다.

31 Classic Networks (LeNet, AlexNet, VGG)

☒ 지난 시간 복습 및 연결

우리는 CNN의 레고 블록(Convolution, Pooling, Activation)을 마스터했습니다. 이제 이 블록들을 조립하여 거대한 성(Model)을 쌓을 시간입니다. 딥러닝 역사에는 "이 모델 이전과 이후로 세상이 나뉘었다"고 평가받는 전설적인 아키텍처들이 있습니다. 오늘은 그 역사의 시작점인 LeNet, 딥러닝 붐을 일으킨 AlexNet, 그리고 깊은 신경망의 표준을 제시한 VGG를 해부합니다.

31.1 Overview

[핵심 목표]

이 단원은 현대 컴퓨터 비전 모델의 조상이 되는 세 가지 고전 모델을 다룹니다.

- **LeNet-5 (1998):** CNN의 기본 구조(Conv-Pool 반복)를 정립한 선구자.
- **AlexNet (2012):** ReLU, Dropout 등을 도입하여 딥러닝 붐을 일으킨 주인공.
- **VGG-16 (2014):** 3×3 필터만으로 깊이를 쌓는 '단순함의 미학'을 증명한 표준 모델.
- **패턴:** 채널은 늘리고(↑), 크기는 줄이는(↓) 공통적인 설계 패턴을 익힙니다.

31.1.1 Model Summary Table

특징	LeNet-5 (1998)	AlexNet (2012)	VGG-16 (2014)
입력	32×32 (Gray)	227×227 (RGB)	224×224 (RGB)
필터	5×5	$11 \times 11, 5 \times 5$	Only 3×3
활성화	Sigmoid / Tanh	ReLU	ReLU
풀링	Average Pooling	Max Pooling	Max Pooling
파라미터	약 6만 개	약 6,000만 개	약 1억 3,800만 개

31.1.2 Core Concepts: 전설들의 계보

31.1.3 1. LeNet-5: The Pioneer

얀 르쿤(Yann LeCun) 교수가 은행 수표의 손글씨 숫자(MNIST) 인식을 위해 개발했습니다.

- 구조: Conv $\rightarrow$ Pool $\rightarrow$ Conv $\rightarrow$ Pool $\rightarrow$ FC ...
- 의의: 이미지의 크기는 줄이고($32 \rightarrow 28 \rightarrow 14 \ldots$), 채널 수는 늘리는($1 \rightarrow 6 \rightarrow 16$) 패턴을 처음 정립했습니다.

31.1.4 2. AlexNet: The Game Changer

2012년 ImageNet 대회 우승작. 딥러닝 시대를 연 장본인입니다.

- ReLU: Sigmoid의 기울기 소실 문제를 해결했습니다.
- Dropout: 과대적합을 막기 위해 FC 층에 드롭아웃을 적용했습니다.
- Multi-GPU: 당시 GPU 성능 한계로 모델을 두 개로 쪼개 학습했습니다.

31.1.5 3. VGG-16: The Standardizer

옥스퍼드 대학 VGG 팀이 개발했습니다. "화려한 기교(11x11 필터 등)는 필요 없다. 깊이(Depth)만이 정답이다"를 증명했습니다.

Why 3×3 filters? VGG는 모든 층에서 3×3 필터만 씁니다. 왜 큰 필터 한 번 대신 작은 필터를 여러 번 쓸까요?

1. 파라미터 효율 (Parameter Efficiency)

- 5×5 필터 1개: $25 \times C^2$ 파라미터.
- 3×3 필터 2개: $2 \times (9 \times C^2) = 18 \times C^2$ 파라미터.
- 결론: 같은 영역(Receptive Field)을 보면서도 파라미터 수를 28% 절약합니다.

2. 비선형성 (Non-linearity)

- 층이 두 개라는 것은 ReLU를 두 번 통과한다는 뜻입니다.
- 더 복잡하고 정교한 함수를 학습할 수 있습니다.

31.1.6 Implementation: Load VGG16 with Keras

최신 프레임워크에서는 이 거대한 모델을 단 한 줄로 불러올 수 있습니다. 전이 학습의 기초가 됩니다.

Listing 29: Loading VGG16

```
from tensorflow.keras.applications import VGG16

# ImageNet □ □ □ □  □ □  VGG16 □ □  □ □
# include_top=True: □ □ □  FC □ □ □ □ (1000 □ □ □ )□ □  □ □
model = VGG16(weights='imagenet', include_top=True)

# □ □  □ □  □ □
model.summary()

# □ □  □ □  □ □ ( □ □ ):
# Block 1: Conv(3x3) -> Conv(3x3) -> MaxPool
# Block 2: Conv(3x3) -> Conv(3x3) -> MaxPool
# ... □ □ ( □ : 64 -> 128 -> 256 -> 512)
```

31.1.7 FAQ Pitfalls

Q. VGG-16의 단점은 없나요?

A. 너무 무겁습니다. 파라미터 수가 1억 3,800만 개나 되어 메모리를 엄청나게 잡아먹습니다. 또한 16층, 19층까지는 괜찮았지만, 그 이상 쌓으면 다시 기울기 소실 문제로 학습이 안 됩니다.

Q. 1×1 Convolution은 뭔가요? (Inception 예고)

A. 필터 크기가 1×1인 합성곱입니다. 공간적 정보(H, W)는 건드리지 않고, 채널 수(C)를 줄이거나 늘리는 역할을 합니다. 연산량을 줄이는 '병목(Bottleneck)' 기법의 핵심입니다.

☒ 다음 단계 (Next Step)

VGG는 훌륭했지만, 층이 20개를 넘어가면 성능이 오히려 떨어지는 현상이 발견되었습니다. (Degradation Problem) 인간의 뇌는 수백 층의 깊이도 처리합니다. 딥러닝도 100층, 1000층을 쌓을 수 없을까요?

다음 시간에는 딥러닝 역사상 가장 중요한 발명 중 하나인 '잔차 연결(Residual Connection)'을 도입하여 152층을 쌓은 괴물, [ResNet (Residual Networks)]을 분석하겠습니다.

[단원 요약 (Cheat Sheet)]

1. **LeNet-5:** CNN의 조상. Conv-Pool 패턴의 시초.

2. **AlexNet:** ReLU와 Dropout으로 딥러닝 성능을 입증함.
3. **VGG-16:** 3×3 필터만 사용하여 깊이를 쌓음. (단순함, 파라미터 효율)
4. **Limit:** VGG조차도 너무 깊어지면 학습이 안 되는 한계가 있음.

32 ResNet (Skip Connections)

☒ 지난 시간 복습 및 연결

지난 시간에 배운 VGG-16은 16개 층으로 뛰어난 성능을 보였습니다. 그렇다면 질문이 생깁니다. "층을 100개, 1000개로 늘리면 성능이 더 좋아지지 않을까요?" 이론적으로는 그래야 합니다. 하지만 실제로는 층이 20개를 넘어가면 성능이 급격히 떨어지는 퇴보(Degradation) 현상이 발생했습니다. 깊은 망을 학습시키는 것 자체가 너무 어려웠던 것입니다. 이 난제를 해결하고 딥러닝 역사를 새로 쓴 모델이 바로 ResNet입니다. 핵심은 "지름길(Shortcut)"을 뚫어주는 것입니다.

32.1 Overview

[핵심 목표]

이 단원은 152층 이상의 초심층 신경망 학습을 가능하게 한 ResNet의 핵심 원리를 다룹니다.

- **문제:** 망이 깊어질수록 학습이 안 되는 이유(기울기 소실 등)를 이해합니다.
- **해결:** 입력 x를 출력에 더해주는 Skip Connection 구조를 파악합니다.
- **원리:** 잔차 블록이 어떻게 항등 매핑($H(x) = x$)을 쉽게 학습하는지 수식으로 증명합니다.
- **구현:** Keras로 Identity Block과 Convolutional Block을 구현합니다.

32.1.1 Essential Terminology

용어	의미	핵심
Skip Connection	입력을 몇 층 건너뛰어 출력에 더하는 연결선	지름길 (Shortcut)
Residual (잔차)	학습해야 할 차이 ($F(x)$)	$H(x) - x$
Identity Mapping	입력을 그대로 출력하는 것 ($H(x) = x$)	기본은 한다.

32.1.2 Core Concepts: 지름길의 마법

32.1.3 1. The Degradation Problem (성능 저하)

일반적인 네트워크(Plain Network)에 층을 계속 추가하면, 학습 데이터에 대한 에러조차 높아집니다. 기울기(Gradient)가 입력층까지 도달하지 못하고 사라져버리기 때문입니다.

32.1.4 2. Skip Connection (잔차 연결)

ResNet의 아이디어는 간단합니다. "정보가 흐르는 고속도로를 뚫어주자."

- **Main Path:** 합성곱 층을 통과하여 $F(x)$를 계산합니다.
- **Shortcut:** 입력 x를 그대로 가져와서 더합니다.
- **Output:** $H(x) = F(x) + x$

[교과서 암기 비유]

- **Plain:** "백지상태에서 교과서 전체($H(x)$)를 다 외워라." (어렵다)
- **ResNet:** "너는 이미 x만큼 알고 있으니, 교과서 내용과 네 지식의 차이($F(x)$)만 추가로 공부해라." (쉽다)

32.1.5 Deep Dive: Why does it work?

Gradient Superhighway 역전파 시 미분값이 전달되는 과정을 봅시다.

$$H(x) = F(x) + x$$

$$\frac{\partial H}{\partial x} = \frac{\partial F(x)}{\partial x} + \mathbf{1}$$

의미: 복잡한 합성곱 경로(F)의 미분값이 0이 되어도(Vanishing), 지름길 경로($+1$)가 살아있습니다. 기울기가 소실되지 않고 네트워크 앞단까지 그대로 전달됩니다.

32.1.6 3. Dimension Matching (차원 일치)

$F(x)$와 x를 더하려면 두 행렬의 크기가 같아야 합니다.

- **Identity Block:** 입출력 크기가 같을 때. 그냥 더함.
- **Convolutional Block:** 크기가 다를 때(Pooling 등). x에도 1×1 Conv를 적용해 크기를 맞춰준 뒤 더함.

32.1.7 Implementation: ResNet Block

Keras Functional API를 사용한 구현입니다.

Listing 30: ResNet Identity Block

```
from tensorflow.keras import layers, models

def identity_block(X, f, filters):
    """
␣␣␣␣X:␣□ □ ␣□ □
␣␣␣␣f:␣□ □ ␣□ □ ␣□ □ □ ()
␣␣␣␣filters:␣□ □ ␣□ □ ␣□ □ □ ␣[F1,␣F2,␣F3]
␣␣␣␣"""
    F1, F2, F3 = filters
    X_shortcut = X # □ □  □ □  □ □ □ □ ()

    # --- Main Path □ □ (3 Conv □ ) ---
    # 1. 1x1 Conv □ □ ( □ □ □ □ □ /)
    X = layers.Conv2D(filters=F1, kernel_size=(1, 1), padding='valid')(X)
    X = layers.BatchNormalization()(X)
    X = layers.Activation('relu')(X)

    # 2. fxf Conv □ □ ( □ □ )
    X = layers.Conv2D(filters=F2, kernel_size=(f, f), padding='same')(X)
    X = layers.BatchNormalization()(X)
    X = layers.Activation('relu')(X)

    # 3. 1x1 Conv □ □ ( □ □ )
    X = layers.Conv2D(filters=F3, kernel_size=(1, 1), padding='valid')(X)
    X = layers.BatchNormalization()(X)

    # --- Skip Connection □ □ () ---
    # Main Path □ □ □  □ □ □ (Original X)□  □ □
    X = layers.Add()([X, X_shortcut])

    # □ □  □ □  ReLU □ □  □ □ (!)
    X = layers.Activation('relu')(X)

    return X
```

32.1.8 FAQ Pitfalls

[ReLU 위치 주의]

Skip Connection을 더하는 'Add()' 연산은 마지막 ReLU 이전에 수행되어야 합니다. (X + Shortcut) $\rightarrow$ ReLU. 순서가 바뀌면 성능이 떨어집니다.

Q. ResNet-50의 'Bottleneck' 구조가 뭔가요?

A. 1×1, 3×3, 1×1 순서로 쌓은 블록입니다. 1×1로 채널을 줄였다가(압축), 연산하고, 다시 늘립니다. 연산량을 줄이면서 깊이를 늘리기 위한 테크닉입니다.

☒ 다음 단계 (Next Step)

ResNet을 통해 우리는 깊이에 대한 두려움을 극복했습니다. 그런데 비슷한 시기에 구글에서는 깊이가 아니라 "너비(Width)"와 "다양성"에 집중한 모델을 내놓았습니다. 1×1, 3×3, 5×5 필터를 한 층에서 동시에 쓴다면 어떨까요?

다음 시간에는 Network within a Network라고 불리는 1×1 Convolution의 마법과, 이를 활용한 [Inception Network]에 대해 알아보겠습니다.

[단원 요약 (Cheat Sheet)]

1. **Problem:** 너무 깊으면 학습이 안 된다 (Degradation).
2. **Solution:** Skip Connection ($H(x) = F(x) + x$).
3. **Math:** 미분 시 $+1$ 항이 생겨 기울기 소실을 막는다.
4. **Block:** 차원이 같으면 Identity Block, 다르면 Conv Block을 쓴다.

33 Inception Network

☒ 지난 시간 복습 및 연결

지난 시간에 우리는 ResNet을 통해 신경망을 "깊게(Deep)" 쌓는 법을 배웠습니다. 그런데 구글 연구팀은 전혀 다른 질문을 던졌습니다. "필터 크기를 꼭 하나만 골라야 하나? 1×1, 3×3, 5×5를 다 쓰면 안 되나?" 이 단순하고 무식해 보이는 아이디어에서 출발하여, 연산 효율성을 극대화한 아키텍처가 바로 Inception Network입니다. 그리고 이를 가능하게 만든 숨은 공신은 1×1 Convolution입니다.

33.1 Overview

[핵심 목표]

이 단원은 CNN의 효율성 혁명을 이끈 Inception 구조와 핵심 기술을 다룹니다.

- **1x1 Conv:** 채널 수를 조절하여 연산량을 줄이는 Network in Network 개념을 이해합니다.
- **Inception Module:** 다양한 크기의 필터를 병렬로 수행하고 합치는 구조를 파악합니다.
- **Bottleneck:** 1×1 합성곱을 통해 연산 비용을 1/10 수준으로 줄이는 원리를 증명합니다.
- **구현:** 복잡한 분기(Branch) 구조를 Keras로 구현합니다.

33.1.1 Essential Terminology

용어	의미	핵심 역할
1×1 **Convolution**	$1\times1\times C$ 필터 연산	채널 수 조절 (Dimensionality Reduction).
Bottleneck Layer	입력을 압축하는 층	연산량을 획기적으로 줄임.
Inception Module	병렬 연산 블록	다양한 스케일의 특징을 동시에 추출함.
Concatenate	이어 붙이기	병렬로 나온 결과들을 채널 축으로 합침.

33.1.2 Core Concepts: 수도꼭지를 잠가라

33.1.3 1. The Magic of 1×1 Convolution

"교수님, 1×1 필터면 그냥 숫자 하나 곱하는 거 아닙니까?" 2D 이미지에서는 그렇습니다. 하지만 입체적인 볼륨($H\times W\times C$)에서는 다릅니다.

- 연산: $1\times1\times$ **192** (입력 채널 수) 크기의 필터가 입력을 훑습니다.
- 효과: 채널 방향으로 FC Layer를 적용하는 것과 같습니다. 필터 개수를 32개로 설정하면, 출력 채널은 32개가 됩니다. (192 $\rightarrow$ 32 압축)

33.1.4 2. Bottleneck Layer (비용 절감의 핵심)

이 섹션이 오늘 강의의 하이라이트입니다. 숫자로 증명합니다.

Computational Cost Analysis **상황:** 입력 $28\times28\times192$ $\rightarrow$ 출력 $28\times28\times32$ (using 5×5 conv).

1. Naive Approach (그냥 5×5 사용):

$$\text{Cost} = (28\times28\times32)\times(5\times5\times192) \approx \textbf{120,000,000} \text{ (1.2억)}$$

2. Bottleneck Approach (1×1로 줄이고 5×5 사용): (1) 1×1로 192ch $\rightarrow$ 16ch 압축 (중간 단계)

$$(28\times28\times16)\times(1\times1\times192) \approx 2,400,000$$

(2) 5×5로 16ch $\rightarrow$ 32ch 확장 (최종 단계)

$$(28 \times 28 \times 32) \times (5 \times 5 \times 16) \approx 10,000,000$$

총합: $2.4M + 10M = \mathbf{12.4M}$
결과: 연산량이 약 1/10로 줄었습니다. 성능 저하는 거의 없습니다.

33.1.5 Deep Dive: Inception Module Architecture

구글은 고민했습니다. "3×3을 쓸까, 5×5를 쓸까?" 결론은 "그냥 다 하자(Do them all)"였습니다.

- 구조: 1×1, 3×3, 5×5, Max Pooling을 병렬로 수행합니다.
- 병합: 나온 결과들의 크기(28×28)를 맞추고(Padding='same'), 채널 축으로 이어 붙입니다(Concatenate).
- 최적화: 3×3과 5×5 앞에는 반드시 1×1 병목 층을 두어 연산량을 줄입니다.

33.1.6 Implementation: Inception Block with Keras

Keras Functional API를 사용해야 복잡한 분기(Branch) 구조를 짤 수 있습니다.

Listing 31: Inception Module Implementation

```
from tensorflow.keras import layers

def inception_module(x, filters):
    """
␣␣␣␣x:␣□ □ ␣□ □
␣␣␣␣filters:␣□ ␣□ □ □ ␣□ □ ␣□ □ ␣□ □ □ □
␣␣␣␣"""

    # Branch 1: 1x1 Conv
    path1 = layers.Conv2D(filters['f1x1'], (1, 1), padding='same', activation='relu')(x)

    # Branch 2: 1x1 (Reduce) -> 3x3
    path2 = layers.Conv2D(filters['f3x3_reduce'], (1, 1), padding='same', activation='relu')(x)
    path2 = layers.Conv2D(filters['f3x3'], (3, 3), padding='same', activation='relu')(path2)

    # Branch 3: 1x1 (Reduce) -> 5x5
    path3 = layers.Conv2D(filters['f5x5_reduce'], (1, 1), padding='same', activation='relu')(x)
    path3 = layers.Conv2D(filters['f5x5'], (5, 5), padding='same', activation='relu')(path3)

    # Branch 4: MaxPool -> 1x1
    path4 = layers.MaxPooling2D((3, 3), strides=(1, 1), padding='same')(x)
    path4 = layers.Conv2D(filters['pool_proj'], (1, 1), padding='same', activation='relu')(path4)

    # □ □  (Concatenate)
    output = layers.concatenate([path1, path2, path3, path4], axis=3)

    return output
```

33.1.7 FAQ Pitfalls

Q. 정보를 압축했다가 늘려도 손실이 없나요?
A. 네, 괜찮습니다. 이미지 데이터에는 중복성(Redundancy)이 많기 때문입니다. 1×1 층은 필요한 핵심 정보만 압축(Linear Combination)해서 다음 층에 넘겨주는 역할을 합니다.

Q. Max Pooling 뒤에 왜 1×1 Conv를 붙이나요?
A. 풀링은 채널 수를 줄이지 못합니다. 인셉션 모듈을 거칠 때마다 채널이 계속 늘어나는 것을 막기 위해, 풀링 뒤에 1×1을 붙여 채널 수를 강제로 줄여줍니다(Projection).

☒ 다음 단계 (Next Step)

우리는 이제 이미지를 분류(Classification)하는 최고의 아키텍처들을 모두 섭렵했습니다 (VGG, ResNet, Inception).

하지만 현실 세계에서는 이미지에 무엇이 있는지만 아는 것으로는 부족합니다. "그 물체가 어디에 있는지(위치)"도 알아야 합니다. 다음 시간에는 컴퓨터 비전의 꽃, [Object Detection]으로 넘어갑니다. 그중에서도 실시간 객체 탐지의 혁명, YOLO (You Only Look Once) 알고리즘을 향한 여정을 시작해 보겠습니다.

[단원 요약 (Cheat Sheet)]

1. 1×1 **Conv:** 채널 수를 줄여 연산량을 아끼는 핵심 도구.
2. **Inception:** 여러 필터를 병렬로 사용하여 다양한 특징을 동시에 잡는다.
3. **Bottleneck:** 큰 필터 앞에 1×1을 두어 입력을 압축한다. (비용 1/10 절감)
4. **Concatenate:** 병렬 연산 결과를 채널 축으로 합친다.

34 Object Detection: Sliding Window

☒ 지난 시간 복습 및 연결

우리는 지금까지 "이 이미지가 고양이인가?"를 맞추는 분류(Classification) 문제를 풀었습니다. 하지만 자율주행차라면 어떨까요? "전방에 차가 있다"는 것만으로는 부족합니다. "전방 50m 왼쪽 차선에 있다"는 위치 정보가 필요하며, 동시에 보행자, 신호등도 찾아야 합니다. 이것이 객체 탐지(Object Detection)입니다. 오늘은 그 시초인 슬라이딩 윈도우 알고리즘과, 이를 획기적으로 가속화한 합성곱 구현법을 배웁니다.

34.1 Overview

[핵심 목표]

이 단원은 객체 탐지의 기본 개념과 속도 문제를 해결하는 핵심 기술을 다룹니다.

- **개념:** 분류(Classification)와 탐지(Detection)의 차이를 이해합니다.
- **고전:** 윈도우를 이동시키며 찾는 슬라이딩 윈도우 방식의 한계를 파악합니다.
- **혁신:** FC 층을 Conv 층으로 변환하여, 단 한 번의 연산으로 모든 윈도우를 처리하는 기술을 익힙니다.

34.1.1 Essential Terminology

용어	질문	출력 예시
Classification	무엇인가?	Cat (1)
Localization	무엇이고 어디에 있는가? (단일 객체)	Cat, b_x, b_y, b_h, b_w
Detection	무엇들이 각각 어디에 있는가? (다중 객체)	Cat(x,y..), Dog(x,y..)

34.1.2 Core Concepts: 찾을 때까지 뒤진다

34.1.3 1. Sliding Windows Algorithm (Basic)

가장 원시적인 방법입니다. 1. 이미지 왼쪽 상단부터 작은 윈도우를 잘라냅니다. 2. 잘라낸 이미지를 ConvNet에 넣어 예측합니다. 3. 옆으로 한 칸 이동(Stride)해서 반복합니다. 4. 다 끝나면 윈도우 크기를 키워서 다시 처음부터 합니다.

[치명적 단점: 속도]

이 방식은 계산 비용이 폭발합니다. 작은 스트라이드 $\rightarrow$ 수만 번 ConvNet 실행 $\rightarrow$ 너무 느림. 큰 스트라이드 $\rightarrow$ 듬성듬성 봄 $\rightarrow$ 정확도 하락.

34.1.4 Deep Dive: Convolutional Implementation

"어떻게 하면 for-loop 없이 한 번에 처리할까?" 이 섹션이 오늘 강의의 하이라이트입니다.

34.1.5 1. Turning FC into Conv layers

전통적인 CNN의 마지막은 평탄화(Flatten) 후 FC 층이었습니다. 이를 1×1 Conv 층으로 바꿉니다.

- 기존: $5 \times 5 \times 16 \xrightarrow{Flatten} 400 \xrightarrow{FC} 400$
- 변환: $5 \times 5 \times 16 \xrightarrow{Conv(5\times5,400)} \mathbf{1 \times 1 \times 400}$

수학적으로 값은 완벽히 동일하지만, 이제는 공간적 위치 정보를 유지할 수 있게 되었습니다.

34.1.6 2. Running on the Whole Image

이제 이미지를 자르지 않고 통째로 넣습니다.

연산 공유의 마법 학습 모델 입력이 14×14라고 가정합시다. 테스트 때 16×16 이미지를 통째로 넣으면 어떻게 될까요?

- 기존: 4번 잘라서 4번 실행.
- Conv 방식: 16×16 이미지가 네트워크를 통과하면, 최종 출력이 2×2 크기로 나옵니다.
- 해석: $(0, 0)$은 좌상단 윈도우 결과, $(0, 1)$은 우상단 윈도우 결과입니다.
- 결과: 공통 영역의 연산을 공유하므로 속도가 수십 배 빨라집니다.

34.1.7 Implementation: Fully Convolutional Network

Keras를 이용해 FC 층을 Conv 층으로 대체하는 모델을 만듭니다.

Listing 32: Fully Convolutional Model

```
import tensorflow as tf
from tensorflow.keras import layers, models

def create_fcn_model(input_shape, num_classes):
    inputs = layers.Input(shape=input_shape)

    # Feature Extractor
    x = layers.Conv2D(16, (5, 5), activation='relu')(inputs)
    x = layers.MaxPooling2D((2, 2), strides=2)(x)
    x = layers.Conv2D(32, (5, 5), activation='relu')(x)
    x = layers.MaxPooling2D((2, 2), strides=2)(x)

    # --- □□ □□ □□ ---
    # Flatten □□ Conv2D □□
    # □□□ □□□ □□□ 5□□ x5 □□□ □, 5x5 □□ □□
    x = layers.Conv2D(256, (5, 5), activation='relu')(x)

    # □□□ □□□ (1x1 Conv)
    outputs = layers.Conv2D(num_classes, (1, 1), activation='softmax')(x)

    model = models.Model(inputs, outputs)
    return model

# --- □□ ---
if __name__ == "__main__":
    # 1. □□□ □□ ( □□□ )
    train_model = create_fcn_model((28, 28, 3), 4)
    print(train_model.output_shape) # (None, 1, 1, 4)

    # 2. □□□□ □ ( □□□ □□□ □□ )
    test_input = tf.random.normal((1, 32, 32, 3))
    test_output = train_model(test_input)
    print(test_output.shape) # (1, 2, 2, 4) -> □ 4 □□□ □□ □□ □□
```

34.1.8 FAQ Pitfalls

Q. 이렇게 하면 바운딩 박스 위치가 정확한가요?

A. 아니요. 윈도우가 고정된 간격(Stride)으로만 움직이기 때문에, 객체가 윈도우 사이에 걸쳐 있거나 크기가 안 맞으면 정확히 잡아내지 못합니다. 이 문제를 해결하기 위해 다음 시간에 배울 YOLO가 필요합니다.

Q. 입력 이미지 크기가 계속 바뀌어도 되나요?

A. 네, Fully Convolutional Network는 고정된 크기의 FC 층이 없으므로 입력 크기에 제한이 없습니다. 입력이 커지면 출력 맵($H \times W$)도 커질 뿐입니다.

☒ 다음 단계 (Next Step)

우리는 슬라이딩 윈도우를 빠르게 만드는 법을 배웠습니다. 하지만 여전히 "정확한 박스 위치"를 잡지 못하는 한계가 있습니다.

객체가 어디에 있든 정확하게 박스를 쳐주고(Regression), 심지어 하나의 셀에서 여러 객체를 동시에 찾아내는 실시간 탐지의 끝판왕. 다음 시간에는 [YOLO (You Only Look Once)] 알고리즘을 통해 IOU와 Non-max Suppression의 개념을 정복하겠습니다.

[단원 요약 (Cheat Sheet)]

1. **Detection:** 무엇이(What) 어디에(Where) 있는지 찾는 문제.
2. **Sliding Window:** 윈도우를 이동하며 찾음. 너무 느림.
3. **Conv Implementation:** FC 층을 Conv 층으로 바꾸면 연산을 공유할 수 있다.
4. **Result:** 큰 이미지를 한 번만 통과시키면(One pass) 모든 윈도우 결과가 나온다.

35 YOLO Algorithm

☒ 지난 시간 복습 및 연결

지난 시간에 배운 슬라이딩 윈도우는 합성곱 구현으로 속도는 빨라졌지만, 여전히 "박스 위치가 부정확하다"는 한계가 있었습니다. 윈도우가 고정된 간격으로만 움직이기 때문입니다. "객체의 중심을 찾고, 그 중심을 기준으로 박스 크기를 예측하면 어떨까?" 이 아이디어로 탄생한 것이 YOLO입니다. 이름처럼 이미지를 단 한 번만 보고(Look Once), 모든 객체의 위치와 종류를 동시에 찾아내는 혁신적인 알고리즘입니다.

35.1 Overview

[핵심 목표]

이 단원은 실시간 객체 탐지의 표준인 YOLO 알고리즘의 원리를 완벽히 이해합니다.

- **그리드:** 이미지를 격자로 나누고, 객체 중심점이 속한 셀이 책임을 지는 구조를 배웁니다.
- **IoU:** 두 박스가 얼마나 겹치는지를 측정하는 평가 지표를 수학적으로 정의합니다.
- **NMS:** 중복된 박스를 제거하는 비최대 억제(Non-max Suppression) 알고리즘을 익힙니다.
- **앵커:** 겹친 물체를 분리하는 앵커 박스(Anchor Box) 개념을 파악합니다.

35.1.1 Essential Terminology

용어	약어	설명
Grid Cell	-	이미지를 $S \times S$로 나눈 작은 구역.
IoU	Intersection over Union	교집합 영역 / 합집합 영역. (일치도)
NMS	Non-max Suppression	가장 확실한 박스 하나만 남기고 나머지는 지움.
Anchor Box	-	미리 정의된 박스 모양. (길쭉한 사람, 넓은 차 등)

35.1.2 Core Concepts: 단 한 번의 추론

35.1.3 1. Bounding Box Predictions (그리드 시스템)

YOLO는 이미지를 $S \times S$ 그리드(보통 19×19)로 나눕니다. 각 셀은 다음 벡터 y를 예측합니다.

$$y = [p_c, b_x, b_y, b_h, b_w, c_1, c_2, \dots]^T$$

- p_c: 객체가 있을 확률 (Confidence).
- b_x, b_y: 박스 중심 좌표 (셀 내 상대 위치, 0 1).
- b_h, b_w: 박스 높이/너비 (전체 이미지 대비 비율).
- c_i: 클래스 확률 (차, 사람 등).

35.1.4 2. IoU (Intersection over Union)

모델이 예측한 박스가 정답과 얼마나 비슷한지 평가하는 척도입니다. IoU 수식

$$\text{IoU} = \frac{\text{교집합 영역 (Intersection)}}{\text{합집합 영역 (Union)}}$$

- 보통 IoU ≥ 0.5 이면 "올바른 탐지"로 간주합니다.
- 1이면 완벽하게 일치, 0이면 전혀 겹치지 않음.

35.1.5 3. Anchor Boxes (겹친 물체 해결)

한 셀의 중심에 사람과 차가 겹쳐 있다면? 기존 벡터로는 하나만 예측 가능합니다. 이를 위해 미리 정의된 모양(앵커)을 사용합니다.

$$y = [\text{Anchor 1}, \text{Anchor 2}]$$

- Anchor 1 (세로로 긴): 사람 담당.
- Anchor 2 (가로로 넓은): 자동차 담당.

35.1.6 Deep Dive: Non-max Suppression (NMS)

YOLO는 하나의 객체에 대해 여러 셀이 "내가 찾았다!"며 박스를 칠 수 있습니다. 중복을 제거해야 합니다.

1. Filter: $p_c < 0.6$ 인 박스(확신 없는 것)는 모두 버립니다.
2. Select: 남은 박스 중 p_c가 가장 높은 것을 선택합니다 (Best Box).
3. Suppress: 선택된 박스와 IoU ≥ 0.5 인(많이 겹친) 다른 박스들은 "같은 물체를 중복 탐지한 것"으로 보고 지웁니다.
4. Repeat: 박스가 다 정리될 때까지 반복합니다.

35.1.7 Implementation: IoU Calculation

IoU 계산은 객체 탐지 성능 평가와 NMS 구현의 핵심입니다.

Listing 33: IoU Calculation Function

```
def calculate_iou(box1, box2):
    """
␣␣␣␣box:␣(x1,␣y1,␣x2,␣y2)□ □ ␣□ □ □ (,␣□ □ □ )
␣␣␣␣"""
    (b1_x1, b1_y1, b1_x2, b1_y2) = box1
    (b2_x1, b2_y1, b2_x2, b2_y2) = box2

    # 1. □ □ □ (Intersection) □ □  □ □
    xi1 = max(b1_x1, b2_x1)
    yi1 = max(b1_y1, b2_y1)
    xi2 = min(b1_x2, b2_x2)
    yi2 = min(b1_y2, b2_y2)

    # □ □  □ □ □ ( 0)
    inter_area = max(0, xi2 - xi1) * max(0, yi2 - yi1)

    # 2. □ □ □ (Union) □ □  □ □
    b1_area = (b1_x2 - b1_x1) * (b1_y2 - b1_y1)
    b2_area = (b2_x2 - b2_x1) * (b2_y2 - b2_y1)
    union_area = b1_area + b2_area - inter_area

    # 3. IoU
    return inter_area / (union_area + 1e-6)

# --- □ □ □  ---
if __name__ == "__main__":
    box_a = (1, 1, 3, 3) # □ □  4
    box_b = (2, 2, 4, 4) # □ □  4, □ □ □  1
    # □ □ □  = 4 + 4 - 1 = 7
    # IoU = 1/7 = 0.1428...

    print(f"IoU:␣{calculate_iou(box_a,␣box_b):.4f}")
```

35.1.8 FAQ Pitfalls

[좌표계 주의]

YOLO의 출력 b_x, b_y는 그리드 셀 내부에서의 상대 위치(0 1)입니다. 실제 이미지 위에 박스를 그리려면, 셀의 위치(인덱스)를 더하고 이미지 크기를 곱해주는 변환 과정이 필요합니다.

Q. 앵커 박스 크기는 어떻게 정하나요?

A. 보통 훈련 데이터에 있는 객체들의 실제 박스 크기를 모아서 K-Means 클러스터링을 돌립니다. 가장 빈번하게 등장하는 대표적인 모양 5 9개를 선정하여 사용합니다 (YOLO v2부터 적용).

☒ 다음 단계 (Next Step)

이것으로 컴퓨터 비전(CNN) 파트를 마칩니다. 이제 여러분은 정지된 이미지에서 사물을 분류하고 위치까지 찾아내는 기술을 습득했습니다.

하지만 세상은 멈춰 있지 않습니다. 유튜브 영상, 음성 인식, 주가 예측, 번역 등은 시간의 흐름(Sequence)이 있는 데이터입니다. 다음 시간부터는 [Part 5. Sequence Models]의 세계로 떠납니다. 시계열 데이터를 처리하는 가장 기본적인 신경망, RNN (Recurrent Neural Networks)에 대해 알아보겠습니다.

[단원 요약 (Cheat Sheet)]

1. **YOLO:** 그리드 셀마다 바운딩 박스를 회귀(Regression)로 직접 예측한다.
2. **IoU:** 교집합/합집합. 박스 위치 정확도의 척도이자 NMS의 기준.
3. **NMS:** 중복된 박스를 제거하여 객체당 하나의 박스만 남긴다.
4. **Anchor:** 다양한 비율의 객체를 잡기 위해 미리 정의된 박스 모양을 쓴다.

36 Siamese Networks

☒ 지난 시간 복습 및 연결

지난 시간까지 우리는 YOLO를 통해 객체의 위치를 찾는 법을 배웠습니다. 오늘은 그보다 더 까다로운 문제인 "이 사람이 누구인가?"를 구별하는 얼굴 인식에 도전합니다. 우리는 스마트폰을 살 때 얼굴을 한 번만 등록합니다. 그런데 딥러닝은 보통 수천 장의 데이터가 필요합니다. 어떻게 단 한 장의 사진만으로 주인을 알아볼까요? 기존 상식을 깨는 One-shot Learning과 Siamese Network의 비밀을 파헤쳐 봅니다.

36.1 Overview

[핵심 목표]

이 단원은 데이터가 극도로 적은 상황에서 작동하는 얼굴 인식 시스템의 원리를 다룹니다.

- **One-shot Learning:** 단 한 장의 데이터로 학습하는 문제를 '분류'가 아닌 '유사도 측정'으로 풉니다.
- **Siamese Network:** 두 이미지를 같은 네트워크에 통과시켜 거리(Distance)를 계산하는 구조를 배웁니다.
- **Triplet Loss:** Anchor, Positive, Negative 세 장의 사진을 이용한 학습 방법을 수학적으로 유도합니다.

36.1.1 Essential Terminology

용어	의미	핵심 역할
One-shot Learning	한 번만 보고 배우기	데이터가 적은 문제 해결.
Siamese Network	샴 네트워크	두 입력이 같은 가중치(W)를 공유함.
Encoding	$f(x)$	이미지를 128차원 등의 숫자 벡터로 변환.
Triplet Loss	세 쌍 손실 함수	A-P는 가깝게, A-N은 멀게 만듦.

36.1.2 Core Concepts: 분류가 아니라 비교다

36.1.3 1. The Challenge (One-shot Learning)

직원 1,000명의 출입 통제 시스템을 만든다고 가정합시다.

- Softmax (실패): 1,000개 클래스로 분류. 신입 사원이 오면 네트워크 전체를 재학습해야 합니다. (확장성 0점)
- Similarity (성공): 두 사진을 비교하여 거리(Distance) d를 출력합니다.
  - $d(\text{img1}, \text{img2}) \leq \tau$: 같은 사람 (문 열림)
  - $d(\text{img1}, \text{img2}) > \tau$: 다른 사람 (거부)

  신입 사원이 오면 사진만 DB에 추가하면 됩니다. 재학습이 필요 없습니다.

36.1.4 2. Siamese Network (샴 네트워크)

[Image of Siamese network architecture with two shared CNNs feeding into distance calculation]
두 개의 똑같은 네트워크가 머리(가중치)를 공유합니다.

1. 두 이미지 $x^{(1)}, x^{(2)}$를 각각 CNN에 넣습니다.
2. 마지막 FC 층에서 나온 벡터(인코딩) $f(x^{(1)}), f(x^{(2)})$를 얻습니다.

3. 두 벡터 사이의 유클리드 거리를 계산합니다.

$$d(x^{(1)}, x^{(2)}) = ||f(x^{(1)}) - f(x^{(2)})||^2$$

36.1.5 Deep Dive: Triplet Loss (트리플렛 손실)

샴 네트워크를 어떻게 학습시킬까요? 세 장의 사진을 한 세트(Triplet)로 묶어 학습합니다.

- Anchor (A): 기준이 되는 내 사진.
- Positive (P): 나와 같은 사람의 다른 사진.
- Negative (N): 나와 다른 사람(영희)의 사진.

Loss Function Derivation 우리의 목표는 다음과 같습니다.

$$||f(A) - f(P)||^2 \leq ||f(A) - f(N)||^2$$

(A와 P 사이의 거리가 A와 N 사이의 거리보다 작아야 한다.)
하지만 신경망이 $f(x) = 0$ (모든 출력을 0으로)으로 학습해버리면, $0 \leq 0$이 되어버립니다(Trivial Solution). 이를 막기 위해 마진(α)을 둡니다.

$$||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 + \alpha \leq 0$$

최종 손실 함수 (ReLU 형태):

$$L(A, P, N) = \max(0, ||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 + \alpha)$$

36.1.6 Implementation: Triplet Loss

TensorFlow/Keras 스타일의 손실 함수 구현입니다.

Listing 34: Triplet Loss Function

```
import tensorflow as tf

def triplet_loss(y_true, y_pred, alpha=0.2):
    """
␣␣␣␣y_pred:␣[Anchor,␣Positive,␣Negative]␣□ □ □ □ ␣□ □ □ ␣□ □
␣␣␣␣"""
    # 1. □ □ □  □ □  □ □  □ □ ( □ □ □  □ 1/3)
    total_len = y_pred.shape[1]
    emb_size = total_len // 3

    anchor = y_pred[:, 0:emb_size]
    positive = y_pred[:, emb_size:2*emb_size]
    negative = y_pred[:, 2*emb_size:]

    # 2. □ □  □ □  □ □ □ ()
    # axis=-1: □ □ □  □  □ □ □  □
    pos_dist = tf.reduce_sum(tf.square(anchor - positive), axis=-1)
    neg_dist = tf.reduce_sum(tf.square(anchor - negative), axis=-1)

    # 3. Loss □ □  (Margin □ □ )
    basic_loss = pos_dist - neg_dist + alpha

    # 4. max(0, loss)
    loss = tf.maximum(basic_loss, 0.0)

    return tf.reduce_mean(loss)
```

36.1.7 FAQ Pitfalls

[Hard Triplet Mining (학습 데이터 선정)]

랜덤하게 A, P, N을 고르면 학습이 잘 안 됩니다. 왜냐하면 대부분의 경우 랜덤한 두 사람(A, N)은 이미 충분히 다르게 생겼기 때문입니다. ($Loss = 0$) 학습 효율을 위해 "A와 N이 꽤 닮은 경우(Hard Triplet)"를 골라내어 학습시켜야 합니다.

Q. 얼굴 말고 다른 거에도 쓸 수 있나요?

A. 네! 서명 인식, 지문 인식, 심지어 이미지 검색(쇼핑몰에서 비슷한 옷 찾기) 등 "유사한 것을 찾는" 모든 분야에 쓰입니다.

☒ 다음 단계 (Next Step)

얼굴 인식은 CNN이 추출한 '내용(Content)'을 비교하는 기술이었습니다. 그렇다면 CNN이 추출한 '화풍(Style)'만 따로 떼어낼 수도 있을까요?

다음 시간에는 반 고흐의 화풍을 내 사진에 입히는 예술적인 AI, [Neural Style Transfer]에 대해 알아봅니다. CNN의 깊은 층이 무엇을 보고 있는지 시각적으로 확인할 수 있는 흥미로운 시간이 될 것입니다.

[단원 요약 (Cheat Sheet)]

1. **Similarity:** 분류가 아닌 거리 측정 문제로 접근한다.
2. **Siamese Network:** 가중치를 공유하는 쌍둥이 네트워크.
3. **Triplet Loss:** (A, P, N) 구조. A-P는 당기고, A-N은 민다.
4. **Margin α:** 모델이 모든 출력을 0으로 만드는 꼼수를 방지한다.

37 One-Shot Learning

☒ 지난 시간 복습 및 연결

지난 시간에 배운 슬라이딩 윈도우는 합성곱 구현으로 속도는 빨라졌지만, 여전히 "박스 위치가 부정확하다"는 한계가 있었습니다. 윈도우가 고정된 간격으로만 움직이기 때문입니다. "객체의 중심을 찾고, 그 중심을 기준으로 박스 크기를 예측하면 어떨까?" 이 아이디어로 탄생한 것이 YOLO입니다. 이름처럼 이미지를 단 한 번만 보고(Look Once), 모든 객체의 위치와 종류를 동시에 찾아내는 혁신적인 알고리즘입니다.

37.1 Overview

[핵심 목표]

이 단원은 실시간 객체 탐지의 표준인 YOLO 알고리즘의 원리를 완벽히 이해합니다.

- **그리드:** 이미지를 격자로 나누고, 객체 중심점이 속한 셀이 책임을 지는 구조를 배웁니다.
- **IoU:** 두 박스가 얼마나 겹치는지를 측정하는 평가 지표를 수학적으로 정의합니다.
- **NMS:** 중복된 박스를 제거하는 비최대 억제(Non-max Suppression) 알고리즘을 익힙니다.
- **앵커:** 겹친 물체를 분리하는 앵커 박스(Anchor Box) 개념을 파악합니다.

37.1.1 Essential Terminology

용어	약어	설명
Grid Cell	-	이미지를 $S \times S$로 나눈 작은 구역.
IoU	Intersection over Union	교집합 영역 / 합집합 영역. (일치도)
NMS	Non-max Suppression	가장 확실한 박스 하나만 남기고 나머지는 지움.
Anchor Box	-	미리 정의된 박스 모양. (길쭉한 사람, 넓은 차 등)

37.1.2 Core Concepts: 단 한 번의 추론

37.1.3 1. Bounding Box Predictions (그리드 시스템)

YOLO는 이미지를 $S \times S$ 그리드(보통 19×19)로 나눕니다. 각 셀은 다음 벡터 y를 예측합니다.

$$y = [p_c, b_x, b_y, b_h, b_w, c_1, c_2, \dots]^T$$

- p_c: 객체가 있을 확률 (Confidence).
- b_x, b_y: 박스 중심 좌표 (셀 내 상대 위치, 0 1).
- b_h, b_w: 박스 높이/너비 (전체 이미지 대비 비율).
- c_i: 클래스 확률 (차, 사람 등).

37.1.4 2. IoU (Intersection over Union)

모델이 예측한 박스가 정답과 얼마나 비슷한지 평가하는 척도입니다. IoU 수식

$$\text{IoU} = \frac{\text{교집합 영역 (Intersection)}}{\text{합집합 영역 (Union)}}$$

- 보통 IoU ≥ 0.5 이면 "올바른 탐지"로 간주합니다.
- 1이면 완벽하게 일치, 0이면 전혀 겹치지 않음.

37.1.5 3. Anchor Boxes (겹친 물체 해결)

한 셀의 중심에 사람과 차가 겹쳐 있다면? 기존 벡터로는 하나만 예측 가능합니다. 이를 위해 미리 정의된 모양(앵커)을 사용합니다.

$$y = [\text{Anchor 1}, \text{Anchor 2}]$$

- Anchor 1 (세로로 긴): 사람 담당.
- Anchor 2 (가로로 넓은): 자동차 담당.

37.1.6 Deep Dive: Non-max Suppression (NMS)

YOLO는 하나의 객체에 대해 여러 셀이 "내가 찾았다!"며 박스를 칠 수 있습니다. 중복을 제거해야 합니다.

1. Filter: $p_c < 0.6$ 인 박스(확신 없는 것)는 모두 버립니다.
2. Select: 남은 박스 중 p_c가 가장 높은 것을 선택합니다 (Best Box).
3. Suppress: 선택된 박스와 IoU ≥ 0.5 인(많이 겹친) 다른 박스들은 "같은 물체를 중복 탐지한 것"으로 보고 지웁니다.
4. Repeat: 박스가 다 정리될 때까지 반복합니다.

37.1.7 Implementation: IoU Calculation

IoU 계산은 객체 탐지 성능 평가와 NMS 구현의 핵심입니다.

Listing 35: IoU Calculation Function

```
def calculate_iou(box1, box2):
    """
␣␣␣␣box:␣(x1,␣y1,␣x2,␣y2)□ □ ␣□ □ □ (,␣□ □ □ )
␣␣␣␣"""
    (b1_x1, b1_y1, b1_x2, b1_y2) = box1
    (b2_x1, b2_y1, b2_x2, b2_y2) = box2

    # 1. □ □ □ (Intersection) □ □  □ □
    xi1 = max(b1_x1, b2_x1)
    yi1 = max(b1_y1, b2_y1)
    xi2 = min(b1_x2, b2_x2)
    yi2 = min(b1_y2, b2_y2)

    # □ □  □ □ □ ( 0)
    inter_area = max(0, xi2 - xi1) * max(0, yi2 - yi1)

    # 2. □ □ □ (Union) □ □  □ □
    b1_area = (b1_x2 - b1_x1) * (b1_y2 - b1_y1)
    b2_area = (b2_x2 - b2_x1) * (b2_y2 - b2_y1)
    union_area = b1_area + b2_area - inter_area

    # 3. IoU
    return inter_area / (union_area + 1e-6)

# --- □ □ □  ---
if __name__ == "__main__":
    box_a = (1, 1, 3, 3) # □ □  4
    box_b = (2, 2, 4, 4) # □ □  4, □ □ □  1
    # □ □ □  = 4 + 4 - 1 = 7
    # IoU = 1/7 = 0.1428...

    print(f"IoU:␣{calculate_iou(box_a,␣box_b):.4f}")
```

37.1.8 FAQ Pitfalls

[좌표계 주의]

YOLO의 출력 b_x, b_y는 그리드 셀 내부에서의 상대 위치(0 1)입니다. 실제 이미지 위에 박스를 그리려면, 셀의 위치(인덱스)를 더하고 이미지 크기를 곱해주는 변환 과정이 필요합니다.

Q. 앵커 박스 크기는 어떻게 정하나요?

A. 보통 훈련 데이터에 있는 객체들의 실제 박스 크기를 모아서 K-Means 클러스터링을 돌립니다. 가장 빈번하게 등장하는 대표적인 모양 5 9개를 선정하여 사용합니다 (YOLO v2부터 적용).

☒ 다음 단계 (Next Step)

이것으로 컴퓨터 비전(CNN) 파트를 마칩니다. 이제 여러분은 정지된 이미지에서 사물을 분류하고 위치까지 찾아내는 기술을 습득했습니다.

하지만 세상은 멈춰 있지 않습니다. 유튜브 영상, 음성 인식, 주가 예측, 번역 등은 시간의 흐름(Sequence)이 있는 데이터입니다. 다음 시간부터는 [Part 5. Sequence Models]의 세계로 떠납니다. 시계열 데이터를 처리하는 가장 기본적인 신경망, RNN (Recurrent Neural Networks)에 대해 알아보겠습니다.

[단원 요약 (Cheat Sheet)]

1. **YOLO:** 그리드 셀마다 바운딩 박스를 회귀(Regression)로 직접 예측한다.
2. **IoU:** 교집합/합집합. 박스 위치 정확도의 척도이자 NMS의 기준.
3. **NMS:** 중복된 박스를 제거하여 객체당 하나의 박스만 남긴다.
4. **Anchor:** 다양한 비율의 객체를 잡기 위해 미리 정의된 박스 모양을 쓴다.

38 Neural Style Transfer

☒ 지난 시간 복습 및 연결

지난 시간에 우리는 샴 네트워크를 통해 CNN이 이미지의 특징을 벡터로 압축하는 법을 배웠습니다. 그렇다면, 이 특징을 분리해서 조작할 수는 없을까요? 이미지의 '내용(Content)'은 내 사진인데, '화풍(Style)'은 반 고흐의 그림처럼 만들 수 있다면 어떨까요? 이것이 바로 AI 예술의 시초, Neural Style Transfer(NST)입니다.

38.1 Overview

[핵심 목표]

이 단원은 CNN의 특성을 활용하여 두 이미지를 합성하는 NST 알고리즘을 다룹니다.

- **전환:** 가중치(W)가 아닌 입력 이미지(G)의 픽셀을 학습한다는 차이점을 이해합니다.
- **콘텐츠:** 깊은 층의 활성화 맵을 비교하여 내용의 유사성을 측정합니다.
- **스타일:** 그람 행렬(Gram Matrix)을 통해 이미지의 질감과 상관관계를 수치화합니다.
- **통합:** $J(G) = \alpha J_{content} + \beta J_{style}$을 최소화하여 예술 작품을 생성합니다.

38.1.1 Essential Terminology

이미지	기호	역할
Content Image	C	내용의 기준 (예: 내 사진).
Style Image	S	화풍의 기준 (예: 고흐 그림).
Generated Image	G	우리가 만들 결과물 (처음엔 노이즈).
Gram Matrix	$G_{kk'}$	특성맵 채널 간의 상관관계를 나타내는 행렬.

38.1.2 Core Concepts: 무엇을 학습하는가?

38.1.3 1. The Big Picture

NST의 가장 큰 특징은 학습의 대상이 다르다는 것입니다.

- **기존 CNN 학습:** 이미지 고정 $\rightarrow$ 가중치 W 업데이트.
- **NST 학습:** 가중치 W 고정(Pre-trained) $\rightarrow$ 생성 이미지 G의 픽셀값 업데이트.

38.1.4 2. Content Cost Function ($J_{content}$)

"이미지 G가 이미지 C와 비슷한 내용을 담고 있는가?"

- 원리: CNN의 깊은 층(Deep Layer)은 사물의 배치나 구조 같은 고차원 정보를 담고 있습니다.
- 수식: 특정 층 l에서 두 이미지의 활성화 맵($a^{[l]}$) 간의 차이(MSE)를 계산합니다.

$$J_{content}(C, G) = \frac{1}{2} \| a^{[l](C)} - a^{[l](G)} \|^2$$

38.1.5 3. Style Cost Function (J_{style}) - [핵심]

"이미지 G가 이미지 S의 화풍(질감)을 담고 있는가?" 스타일은 "어디에 있는가"가 아니라 "무엇이 함께 나타나는가(Correlation)"입

[그람 행렬의 직관]

어떤 층에 두 개의 필터(채널)가 있다고 가정합시다.

- 필터 A: 수직선을 찾음.
- 필터 B: 주황색을 찾음.

이 두 필터가 이미지의 같은 위치에서 동시에 활성화된다면(상관관계 높음), "이 화풍은 수직선과 주황색이 함께 다니는 스타일이다"라고 정의할 수 있습니다. 이것을 수치화한 것이 그람 행렬(Gram Matrix)입니다.

Gram Matrix ($G^{[l]}$)

$$G^{[l]}_{kk'} = \sum_i \sum_j a^{[l]}_{i,j,k} \cdot a^{[l]}_{i,j,k'}$$

(채널 k와 채널 k'의 활성화 맵 내적)

스타일 비용은 두 이미지의 그람 행렬 차이입니다.

$$J^{[l]}_{style}(S, G) = \|G^{[l](S)} - G^{[l](G)}\|^2$$

38.1.6 Implementation: Optimization Loop

TensorFlow/Keras를 사용한 구현의 핵심 구조입니다.

Listing 36: Neural Style Transfer Logic

```
import tensorflow as tf

# 1. □□ □□ (VGG19, □□□ □□)
vgg = tf.keras.applications.VGG19(include_top=False, weights='imagenet')
vgg.trainable = False

# 2. □□ □□□(G) □□□ □□( □□!)
# □□□□ □□ □□□□ □□□□□ Content □□□□ □□
generated_image = tf.Variable(content_image)

# 3. □□□ □□
optimizer = tf.optimizers.Adam(learning_rate=0.02)

@tf.function
def train_step(image):
    with tf.GradientTape() as tape:
        # □□ □□ -> □□□ □ □□
        outputs = get_vgg_layers(image)

        # Loss □□
        loss_c = calculate_content_loss(outputs, content_targets)
        loss_s = calculate_style_loss(outputs, style_targets) # Gram Matrix □□

        total_loss = alpha * loss_c + beta * loss_s

    # □□: □□□□ □□ □□'□□□□'□□ □□□ □□
    grad = tape.gradient(total_loss, image)

    # □□□ □□ □□□□
    optimizer.apply_gradients([(grad, image)])

    # □□□ □□□ (0~1 □□ □□)
    image.assign(tf.clip_by_value(image, 0.0, 1.0))
```

38.1.7 FAQ Pitfalls

[왜 그람 행렬인가요?]

그람 행렬은 공간 정보(Spatial Information)를 합쳐버립니다($\sum_{i,j}$). 즉, "어디에" 있는지는 무시하고 "어떤 특징들이 통계적으로 공존하는가"만 남기기 때문에 스타일(질감, 패턴)을 표현하기에 적합합니다.

Q. α와 β는 어떻게 정하나요?

A. 취향입니다. α(콘텐츠)를 높이면 원본 사진과 비슷해지고, β(스타일)를 높이면 그림 화풍이 강해집니다. 보통 β를 훨씬 크게 설정합니다(숫자 스케일 차이 때문).

☒ 다음 단계 (Next Step)

우리는 CNN을 이용해 이미지를 분류하고, 탐지하고, 심지어 예술 작품으로 변환하는 방법까지 배웠습니다. 이로써 컴퓨터 비전(Computer Vision) 파트를 마무리합니다.

이제 시각 정보가 아닌, 시간의 흐름이 있는 데이터(Sequence Data)의 세계로 넘어갑니다. 다음 시간부터는 음성, 언어, 주가 등 연속적인 데이터를 처리하는 [Part 5. Sequence Models]를 시작하며, 그 첫 번째 주자인 RNN (Recurrent Neural Networks)에 대해 알아보겠습니다.

[단원 요약 (Cheat Sheet)]

1. **NST:** 사전 학습된 CNN을 이용해 콘텐츠와 스타일을 합성한다.
2. **Learning:** 가중치는 고정하고, 입력 이미지의 픽셀을 학습(업데이트)한다.
3. **Content:** 깊은 층의 활성화 맵 차이를 줄인다.
4. **Style:** 그람 행렬(Gram Matrix)의 차이를 줄여 질감 상관관계를 모방한다.

39 RNN: Sequence Models

☒ 지난 시간 복습 및 연결

우리는 지금까지 고정된 크기의 이미지($H \times W$)를 처리하는 CNN을 다뤘습니다. 하지만 세상의 많은 데이터는 '순서(Sequence)'와 '시간(Time)'을 가지고 있습니다. "나는 프랑스에 가서..."라는 말을 들으면, 뒤에 "프랑스어"라는 말이 나올 확률이 높다는 것을 우리는 압니다. 이는 앞선 단어들의 문맥(Context)을 기억하기 때문입니다. 기존 신경망은 이 '기억' 능력이 없습니다. 오늘은 기억을 가진 신경망, RNN의 세계로 들어갑니다.

39.1 Overview

[핵심 목표]

이 단원은 시계열 데이터를 처리하는 RNN의 기본 원리와 학습 알고리즘을 다룹니다.

- **구조:** 은닉 상태(Hidden State)를 통해 과거 정보를 현재로 전달하는 루프 구조를 이해합니다.
- **수식:** $a^{\langle t \rangle} = g(W_{aa}a^{\langle t-1 \rangle} + W_{ax}x^{\langle t \rangle})$ 공식을 마스터합니다.
- **공유:** 모든 시간 단계에서 파라미터(W)를 공유하여 일반화 성능을 높이는 원리를 파악합니다.
- **학습:** 시간을 거슬러 올라가는 역전파, BPTT의 개념을 익힙니다.

39.1.1 Essential Terminology

기호	용어	설명
$x^{\langle t \rangle}$	Input	시간 t에서의 입력 (예: t번째 단어).
$a^{\langle t \rangle}$	Hidden State	시간 t에서의 은닉 상태. (기억)
$y^{\langle t \rangle}$	Output	시간 t에서의 출력 (예: 다음 단어 예측).
W_{aa}	Weight (Hidden)	과거 기억을 현재로 가져오는 가중치.
W_{ax}	Weight (Input)	현재 입력을 받아들이는 가중치.

39.1.2 Core Concepts: 순환의 마법

39.1.3 1. RNN Architecture (Unrolled)

RNN은 자신을 가리키는 화살표(Loop)를 가집니다. 이를 시간축으로 펼치면 다음과 같습니다.

- 입력: 매 시점 t마다 $x^{\langle t \rangle}$가 들어옵니다.
- 전달: 이전 시점의 기억 $a^{\langle t-1 \rangle}$이 현재 시점 t로 전달됩니다.
- 출력: 두 정보를 합쳐서 $y^{\langle t \rangle}$를 출력하고, 다음 시점 $t+1$로 기억 $a^{\langle t \rangle}$를 넘깁니다.

39.1.4 2. Forward Propagation Formulas

RNN의 핵심은 "현재 입력과 과거 기억을 섞어서 새로운 기억을 만든다"는 것입니다.
은닉 상태 업데이트 (기억 갱신)

$$a^{\langle t \rangle} = \tanh(W_{aa}a^{\langle t-1 \rangle} + W_{ax}x^{\langle t \rangle} + b_a)$$

- $W_{aa}a^{\langle t-1 \rangle}$: 과거의 기억 반영.
- $W_{ax}x^{\langle t \rangle}$: 현재의 정보 반영.
- tanh: 값을 -1 1 사이로 압축하여 폭발 방지 (주로 사용).

출력 계산

$$\hat{y}^{\langle t \rangle} = \text{softmax}(W_{ya}a^{\langle t \rangle} + b_y)$$

- 현재의 기억($a^{\langle t \rangle}$)을 바탕으로 예측을 수행합니다.

Key Point (Parameter Sharing): $t = 1$이든 $t = 100$이든, W_{aa}, W_{ax}, W_{ya}는 모두 똑같은 행렬을 재사용합니다. 이것이 RNN이 길이 제한 없이 문장을 처리할 수 있는 비결입니다.

39.1.5 Deep Dive: Backpropagation Through Time (BPTT)

RNN의 학습은 시간을 거슬러 올라갑니다.

- Loss: 전체 손실 L은 각 시간 단계별 손실의 합입니다. $L = \sum L^{\langle t \rangle}$.
- Gradient: $t = 100$ 시점의 오차를 수정하려면, $t = 99, 98, \ldots, 1$ 시점의 상태까지 영향을 미쳐야 합니다.
- Problem: 미분값이 계속 곱해지면서(W_{aa}^{100}), 값이 0으로 사라지거나(Vanishing) 무한대로 커지는(Exploding) 문제가 발생합니다. 이로 인해 기본 RNN은 장기 의존성(Long-term Dependency)을 학습하기 어렵습니다.

39.1.6 Implementation: RNN Step Loop

NumPy로 RNN의 내부 동작을 구현해 봅니다.

Listing 37: RNN Forward Pass Implementation

```
import numpy as np

def rnn_cell_forward(xt, a_prev, parameters):
    """
␣␣␣␣□ □ ␣□ □ ␣□ □ ␣(t)␣□ □
␣␣␣␣"""
    Wax = parameters["Wax"]
    Waa = parameters["Waa"]
    Wya = parameters["Wya"]
    ba = parameters["ba"]
    by = parameters["by"]

    # 1. □ □  □ □  □ □  □ □  (Tanh □ □ )
    # a_next = tanh(Waa*a_prev + Wax*xt + ba)
    a_next = np.tanh(np.dot(Waa, a_prev) + np.dot(Wax, xt) + ba)

    # 2. □ □  □ □  □ □  (Softmax □ □ )
    yt_pred = softmax(np.dot(Wya, a_next) + by)

    return a_next, yt_pred

def rnn_forward(x, a0, parameters):
    """
␣␣␣␣□ □ ␣□ □ □ ␣□ □ ␣(Time␣Loop)
␣␣␣␣"""
    n_x, m, T_x = x.shape  # T_x: □ □  □ □  (Timesteps)
```

```
    n_y, n_a = parameters["Wya"].shape

    a = np.zeros((n_a, m, T_x))      # □ □  □ □  □ □ □
    y_pred = np.zeros((n_y, m, T_x)) # □ □  □ □  □ □ □

    a_next = a0 # □ □  □ □

    # □ □  □ □ □ □  □ □  (□ RNN □ □ )
    for t in range(T_x):
        xt = x[:, :, t] # □ □ t □ □

        # □  □ □ □ □
        a_next, yt_pred = rnn_cell_forward(xt, a_next, parameters)

        # □ □
        a[:, :, t] = a_next
        y_pred[:, :, t] = yt_pred

    return a, y_pred

def softmax(x):
    e_x = np.exp(x - np.max(x))
    return e_x / e_x.sum(axis=0)
```

39.1.7 FAQ Pitfalls

Q. 왜 ReLU 대신 Tanh를 쓰나요?
A. RNN은 같은 가중치를 수십, 수백 번 반복해서 곱합니다. ReLU를 쓰면 값이 계속 커져서 발산(Exploding)하기 쉽습니다. Tanh는 값을 -1 1 사이로 묶어두어(Bounding) 안정적인 학습을 돕습니다.

Q. RNN은 병렬 처리가 안 되나요?
A. 네, 구조적으로 어렵습니다. t 시점의 계산을 하려면 반드시 $t-1$ 시점의 결과가 나와야 하기 때문입니다(Sequential). 이것이 트랜스포머(Transformer)가 등장하게 된 배경 중 하나입니다.

☒ 다음 단계 (Next Step)

기본 RNN은 이론적으로 훌륭하지만, 문장이 조금만 길어져도(10단어 이상) 앞부분 내용을 까먹는 기울기 소실 문제가 있습니다.

이를 해결하기 위해 딥러닝 연구자들은 "기억을 얼마나 오래 유지할지 스스로 결정하는 게이트(Gate)"를 만들었습니다. 다음 시간에는 현대 NLP의 근간이 된 [GRU (Gated Recurrent Unit)]와 [LSTM (Long Short-Term Memory)]에 대해 다루겠습니다.

[단원 요약 (Cheat Sheet)]

1. **Structure:** 입력(x) + 이전 기억(a) $\rightarrow$ 새 기억 $\rightarrow$ 출력(y).
2. **Sharing:** 모든 시점에서 동일한 파라미터(W_{ax}, W_{aa})를 쓴다.
3. **Formula:** $a^{\langle t \rangle} = \tanh(W_{aa}a^{\langle t-1 \rangle} + W_{ax}x^{\langle t \rangle} + b_a)$.
4. **Limit:** 긴 시퀀스에서는 기울기가 소실되어(Vanishing Gradient) 초기 기억을 잃는다.

40 RNN Architectures

☒ 지난 시간 복습 및 연결

지난 시간에 우리는 RNN의 기본 구조(Basic RNN Cell)를 배웠습니다. 당시에는 입력 길이(T_x)와 출력 길이(T_y)가 같은 경우만 가정했습니다. 하지만 현실은 다릅니다. "I love you"(3단어)를 번역하면 "Je t'aime"(2단어)가 됩니다. 음악 생성은 장르 하나(1)를 주면 곡 전체(수백)를 만듭니다. RNN의 강력함은 바로 이 입출력 구조의 유연성(Flexibility)에서 나옵니다.

40.1 Overview

[핵심 목표]

이 단원은 입력(T_x)과 출력(T_y)의 조합에 따른 5가지 RNN 아키텍처를 분류하고 이해합니다.

- **One-to-Many:** 하나의 입력으로 시퀀스를 생성 (음악, 캡셔닝).
- **Many-to-One:** 시퀀스를 하나의 결과로 요약 (감성 분석).
- **Many-to-Many (Same):** 입력마다 즉시 출력 (개체명 인식).
- **Many-to-Many (Diff):** 다 듣고 말하기 (기계 번역, Encoder-Decoder).

40.1.1 Essential Terminology

구조	입력 : 출력	대표 예시
One-to-One	1 : 1	일반적인 신경망 (이미지 분류).
One-to-Many	1 : N	음악 생성, 이미지 캡셔닝.
Many-to-One	N : 1	감성 분석 (리뷰 $\rightarrow$ 별점).
Many-to-Many	N : N	개체명 인식 (NER).
Seq2Seq	N : M	기계 번역 (번역은 문장 끝까지 들어야 함).

40.1.2 Core Concepts: 구조의 다양성

40.1.3 1. One-to-Many ($T_x = 1, T_y > 1$)

하나의 씨앗(Seed) 정보를 주면 긴 시퀀스를 생성합니다.

- 동작: 첫 타임 스텝에서 입력 x를 받습니다. 그 이후에는 전 단계의 출력 $\hat{y}^{\langle t-1 \rangle}$을 다음 단계의 입력으로 재사용합니다 (Auto-regressive).
- 예시: 음악 생성 (장르 $\rightarrow$ 멜로디), 이미지 캡셔닝 (이미지 $\rightarrow$ "고양이가 잔다").

40.1.4 2. Many-to-One ($T_x > 1, T_y = 1$)

긴 시퀀스를 읽어서 하나의 결론을 내립니다.

- 동작: 시퀀스를 끝까지 읽으며 은닉 상태(a)를 업데이트합니다. 출력은 마지막 타임 스텝에서만 나옵니다.
- 예시: 영화 리뷰 감성 분석 (텍스트 $\rightarrow$ 긍정/부정).

40.1.5 Deep Dive: Many-to-Many (The Tricky Part)

Many-to-Many는 다시 두 가지로 나뉩니다. 이 차이가 매우 중요합니다.

40.1.6 1. Synced Many-to-Many ($T_x = T_y$)

입력과 출력의 길이가 같고, 타이밍이 일치합니다.

- 동작: 입력이 들어올 때마다 즉시 출력을 뱉습니다.
- 예시: 비디오 프레임 분류, 개체명 인식(NER).

40.1.7 2. Asynchronous Many-to-Many ($T_x \neq T_y$) - Seq2Seq

입력과 출력의 길이가 다릅니다. 기계 번역의 표준인 Encoder-Decoder 구조입니다.

- Encoder (Many-to-One): 입력 문장 전체를 읽어서 하나의 문맥 벡터(Context Vector)로 압축합니다.
- Decoder (One-to-Many): 압축된 벡터를 바탕으로 번역 문장을 생성합니다.
- 이유: "나는 너를 사랑해"를 번역하려면 문장 끝까지 들어봐야 어순을 맞출 수 있기 때문입니다.

40.1.8 Implementation: Encoder-Decoder Structure

Keras를 이용해 Seq2Seq 모델의 개념적 구조를 구현해 봅니다.

Listing 38: Encoder-Decoder Implementation

```
from tensorflow.keras.layers import Input, LSTM, Dense
from tensorflow.keras.models import Model

def build_seq2seq(n_x, n_y, Tx, Ty, n_a):
    """
␣␣␣␣n_x,␣n_y:␣□ □ □ /␣□ □ ␣□ □ ␣□ □
␣␣␣␣Tx,␣Ty:␣□ □ □ /␣□ □ □ ␣□ □
␣␣␣␣n_a:␣□ □ ␣□ □ ␣□ □
␣␣␣␣"""

    # --- 1. Encoder □ □ ( □ □ ) ---
    encoder_inputs = Input(shape=(Tx, n_x))

    # return_state=True: □ □ □  □ □ (h, c)□  □ □ □ □
    # return_sequences=False: □ □  □ □ □  □ □  □ □  □ □ □ ( □ □ )
    encoder = LSTM(n_a, return_state=True)

    _, state_h, state_c = encoder(encoder_inputs)

    # □  'encoder_states□ ' □ □  □ □ □  □ □  □ □ □  □ □  □ □ ''
    encoder_states = [state_h, state_c]

    # --- 2. Decoder □ □ ( □ □ ) ---
    decoder_inputs = Input(shape=(Ty, n_y))

    # □ Encoder □ □ □  □ Decoder □ □  □ □ (initial_state)□  □ □ !
    decoder_lstm = LSTM(n_a, return_sequences=True, return_state=True)
    decoder_outputs, _, _ = decoder_lstm(decoder_inputs,
                                         initial_state=encoder_states)

    # □ □  □ □  (Softmax)
    decoder_dense = Dense(n_y, activation='softmax')
    decoder_outputs = decoder_dense(decoder_outputs)

    # □ □  □ □
    model = Model([encoder_inputs, decoder_inputs], decoder_outputs)
    return model
```

40.1.9 FAQ Pitfalls

Q. 입력 문장이 너무 길면 어떻게 되나요?
A. Encoder가 문장의 모든 정보를 하나의 벡터에 욱여넣어야 하므로, 문장이 길어지면(예: 50단어 이상) 정보 손실이 발생해 성능이 떨어집니다. 이를 해결하기 위해 나중에 Attention Mechanism이 등장합니다.

Q. 이미지 캡셔닝은 어떤 구조인가요?
A. One-to-Many입니다. 입력은 이미지(CNN을 통과한 1개의 벡터), 출력은 텍스트 시퀀스(RNN)입니다. CNN이 Encoder, RNN이 Decoder 역할을 하는 셈입니다.

☒ 다음 단계 (Next Step)

RNN은 이렇게 다양한 구조로 변신할 수 있습니다. 하지만 어떤 구조를 쓰든, 시퀀스가 길어지면 앞의 내용을 잊어버리는 '기울기 소실'이라는 고질병은 여전합니다.

다음 시간에는 이 문제를 해결하기 위해 "기억을 저장하는 금고(Cell State)"와 "문의 열쇠(Gate)"를 도입한 현대적 RNN의 표준, [GRU]와 [LSTM]을 해부하겠습니다.

[단원 요약 (Cheat Sheet)]

1. **Flexibility:** T_x와 T_y가 달라도 처리할 수 있다.
2. **Many-to-One:** 감성 분석 (정보 요약).
3. **Many-to-Many:** 개체명 인식 (즉시 출력) vs 기계 번역 (다 듣고 출력).
4. **Seq2Seq:** Encoder가 압축하고 Decoder가 생성한다.

41 GRU (Gated Recurrent Unit)

☒ 지난 시간 복습 및 연결

지난 시간에 배운 기본 RNN은 이론적으로는 완벽해 보입니다. 과거 정보를 현재로 전달하니까요. 하지만 실제로는 문장이 10단어만 넘어가도 앞부분 내용을 까맣게 잊어버립니다. "The cat, which ate ..., was full." RNN은 중간 수식어가 길어지면 주어 'cat(단수)'을 잊어버리고 동사 'was'를 예측하지 못합니다. 이것이 바로 기울기 소실(Vanishing Gradient) 문제입니다. 오늘은 이 난제를 해결한 GRU를 배웁니다.

41.1 Overview

[핵심 목표]

이 단원은 RNN의 장기 의존성 문제를 해결하는 게이트(Gate) 메커니즘을 다룹니다.

- **원인:** 역전파 시 기울기가 지수적으로 작아져서 초기 기억이 사라지는 수학적 원리를 이해합니다.
- **해결:** 정보를 "얼마나 유지하고 버릴지" 결정하는 게이트(Gate) 개념을 도입합니다.
- **모델:** LSTM의 간소화 버전인 GRU의 구조와 수식을 마스터합니다.

41.1.1 Essential Terminology

용어	의미	역할
Vanishing Gradient	기울기 소실	깊은 신경망 학습을 방해하는 주범.
Gate (Γ)	문지기 (0 1)	정보 흐름을 제어하는 시그모이드 함수.
Update Gate (Γ_u)	업데이트 게이트	과거 기억을 얼마나 유지할지 결정.
Reset Gate (Γ_r)	리셋 게이트	과거 기억을 얼마나 무시할지 결정.

41.1.2 Core Concepts: 기억의 보존

41.1.3 1. The Problem: Vanishing Gradients

RNN에서 $t = 100$ 시점의 오차를 $t = 1$ 시점까지 전파하려면 가중치 행렬 W를 100번 곱해야 합니다.

[복사본의 복사본]

문서를 복사하고, 그 복사본을 또 복사하는 과정을 100번 반복한다고 상상해 보세요. 조금이라도 흐릿해지면($W < 1$), 100번째 복사본은 백지가 되어버립니다. 반대로 진해지면($W > 1$), 검은색 잉크 덩어리가 됩니다(Exploding).

41.1.4 2. The Solution: Gated Recurrent Unit (GRU)

핵심 아이디어는 "기억을 위한 전용 금고(Memory Cell)를 만들고 문지기를 세우자"는 것입니다.

GRU의 핵심 수식

$$c^{\langle t \rangle} = \Gamma_u \cdot \tilde{c}^{\langle t \rangle} + (1 - \Gamma_u) \cdot c^{\langle t-1 \rangle}$$

- Γ_u (Update Gate): 0이면 문을 닫습니다.
- $\Gamma_u \approx 0$ 일 때: $c^{\langle t \rangle} \approx c^{\langle t-1 \rangle}$.
- 의미: 과거의 기억($c^{\langle t-1 \rangle}$)이 아무런 변형 없이(행렬 곱셈 없이) 그대로 복사되어 다음으로 넘어갑니다. 기울기 소실 없이 고속도로처럼 전달됩니다.

41.1.5 Deep Dive: GRU Gates Detail

GRU는 두 개의 게이트를 사용합니다.

1. Update Gate (Γ_u): "이 기억을 계속 가져갈까?" (단수/복수 정보 유지)

$$\Gamma_u = \sigma(W_u[c^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_u)$$

2. Reset Gate (Γ_r): "이제 과거는 잊을까?" (문침표가 나오면 리셋)

$$\Gamma_r = \sigma(W_r[c^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_r)$$

3. Candidate Memory ($\tilde{c}$): 새로운 기억 후보

$$\tilde{c}^{\langle t \rangle} = \tanh(W_c[\Gamma_r \cdot c^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_c)$$

41.1.6 Implementation: GRU Cell

Listing 39: GRU Cell Forward Implementation

```python
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def gru_cell_forward(xt, a_prev, parameters):
    """
␣␣␣␣xt:␣□ □ ␣□ □
␣␣␣␣a_prev:␣□ □ ␣□ □ ␣□ □ ␣(□ □ GRU␣Memory␣Cell␣□ c␣□ □ )
␣␣␣␣"""
    # □ □ □ □  □ □
    W_u = parameters["Wu"] # Update Gate Weights
    W_r = parameters["Wr"] # Reset Gate Weights
    W_c = parameters["Wc"] # Candidate Memory Weights

    # □ □  □ □  (Concatenate)
    concat = np.concatenate((a_prev, xt), axis=0)

    # 1. Update Gate (Gamma_u)
    Gamma_u = sigmoid(np.dot(W_u, concat) + parameters["bu"])

    # 2. Reset Gate (Gamma_r)
    Gamma_r = sigmoid(np.dot(W_r, concat) + parameters["br"])

    # 3. Candidate Memory (c_tilde)
    # □ □  □ □ □  Reset Gate □ □
    concat_reset = np.concatenate((Gamma_r * a_prev, xt), axis=0)
    c_candidate = np.tanh(np.dot(W_c, concat_reset) + parameters["bc"])

    # 4. Final Memory Update (The Magic Formula)
    # □ u □ □ 0 □ □  □ □  □ □ , □ □ 1 □  □ □ □ □  □ □ □
    c_next = Gamma_u * c_candidate + (1 - Gamma_u) * a_prev

    return c_next
```

41.1.7 FAQ Pitfalls

[Gradient Clipping (기울기 자르기)]

기울기 소실의 반대는 기울기 폭발(Exploding Gradient)입니다. 숫자가 너무 커져서 'NaN'이 뜹니다. 해결책은 간단합니다. 기울기 벡터의 크기(Norm)가 특정 값(예: 5)을 넘으면 강제로 줄여버리는 Gradient Clipping을 사용합니다.

Q. GRU와 LSTM 중 뭐가 더 좋나요?

A. 정답은 없습니다. GRU는 구조가 단순해서(게이트 2개) 연산이 빠르고 데이터가 적을 때 유리합니다. LSTM은 구조가 복잡하지만(게이트 3개) 표현력이 더 좋습니다. 보통 LSTM을 기본으로 쓰고, 속도가 중요하면 GRU를 씁니다.

☒ 다음 단계 (Next Step)

GRU는 훌륭하고 단순합니다. 하지만 때로는 더 섬세한 제어가 필요합니다. GRU보다 조금 더 복잡하지만, 더 강력하고 널리 쓰이는 형님이 있습니다.

다음 시간에는 GRU의 확장판이자 RNN의 사실상 표준(De facto Standard), [LSTM (Long Short-Term Memory)]에 대해 다루겠습니다. 3개의 게이트(Forget, Input, Output)가 어떻게 기억을 수술하듯 정교하게 다루는지 알아보겠습니다.

[단원 요약 (Cheat Sheet)]

1. **Vanishing Gradient:** 시퀀스가 길어지면 초기 정보가 사라지는 문제.
2. **Gate:** 시그모이드를 사용해 정보의 통과 여부(0 1)를 결정하는 밸브.
3. **GRU:** Update/Reset 게이트를 사용해 기억을 보존하거나 갱신한다.
4. **Key:** $\Gamma_u = 0$일 때 과거 기억이 손실 없이 그대로 전달된다.

42 LSTM (Long Short-Term Memory)

☒ 지난 시간 복습 및 연결

지난 시간에 배운 GRU는 '기울기 소실'을 해결하기 위해 게이트를 도입한 훌륭한 모델이었습니다. 하지만 GRU는 2014년에 나온 모델이고, 그보다 훨씬 전인 1997년에 제안되어 지금까지 시퀀스 모델의 제왕(Standard)으로 군림하고 있는 모델이 있습니다. 바로 LSTM입니다. GRU가 2개의 게이트로 효율성을 추구했다면, LSTM은 3개의 게이트로 기억을 더욱 정교하게 제어합니다. 앤드류 응 교수님은 말합니다. "무엇을 쓸지 모르겠다면, 일단 LSTM부터 시작하십시오."

42.1 Overview

[핵심 목표]

이 단원은 시퀀스 모델의 업계 표준인 LSTM의 구조와 동작 원리를 파헤칩니다.

- **구조:** Cell State(c)와 Hidden State(a)가 분리된 이중 트랙 구조를 이해합니다.
- **게이트:** Forget, Update, Output이라는 3개의 게이트 역할을 파악합니다.
- **수식:** 과거를 잊고(Γ_f) 새로운 기억을 더하는(Γ_u) 독립적 제어 공식을 익힙니다.
- **비교:** GRU와 LSTM의 장단점을 비교하고 선택 기준을 세웁니다.

42.1.1 Essential Terminology

용어	기호	역할
Cell State	$c^{\langle t \rangle}$	장기 기억 고속도로. 내부에서만 순환하며 정보를 보존함.
Hidden State	$a^{\langle t \rangle}$	단기 상태 및 출력. Cell State를 가공하여 외부로 내보냄.
Forget Gate	Γ_f	과거의 기억을 삭제하는 비율 (0 1).
Update Gate	Γ_u	새로운 기억을 저장하는 비율 (0 1).
Output Gate	Γ_o	현재 상태를 다음 층으로 내보내는 비율 (0 1).

42.1.2 Core Concepts: 기억의 정교한 제어

42.1.3 1. The Key Difference: c and a

GRU는 기억(c)과 출력(a)이 통합되어 있었습니다. 반면 LSTM은 이를 엄격히 분리합니다.

- Cell State ($c^{\langle t \rangle}$): 기억의 핵심입니다. 기울기가 소실되지 않도록 보호받는 경로입니다.
- Hidden State ($a^{\langle t \rangle}$): Cell State에 tanh를 씌우고 Output Gate를 통과시켜 만든, "지금 당장 필요한 정보"입니다.

42.1.4 2. The Three Gates (3개의 문지기)

LSTM은 기억을 관리하기 위해 3단계 검문을 수행합니다.

[LSTM의 기억 관리 시스템]

- Forget Gate (Γ_f): [쓰레기통] "이전 기억 중 쓸모없는 건 버려라." (예: 문단이 바뀌었으니 이전 주제 삭제)
- Update Gate (Γ_u): [기록장] "새로운 정보 중 중요한 것만 적어라." (예: 새로운 주어 등록)
- Output Gate (Γ_o): [스피커] "지금 당장 필요한 정보만 말해라." (예: 다음 단어 예측에 필요한 정보만 발설)

42.1.5 Deep Dive: LSTM Equations

이 수식들이 LSTM의 작동 원리를 보여주는 지도입니다.

42.1.6 1. Gate Calculation

이전 상태($a^{\langle t-1 \rangle}$)와 현재 입력($x^{\langle t \rangle}$)을 보고 게이트를 얼마나 열지 결정합니다.

$$\Gamma_f = \sigma(W_f[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_f)$$

$$\Gamma_u = \sigma(W_u[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_u)$$

$$\Gamma_o = \sigma(W_o[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_o)$$

42.1.7 2. Memory Update (핵심)

Cell State Update

$$c^{\langle t \rangle} = \underbrace{\Gamma_f \cdot c^{\langle t-1 \rangle}}_{\text{과거 기억 삭제}} + \underbrace{\Gamma_u \cdot \tilde{c}^{\langle t \rangle}}_{\text{새 기억 추가}}$$

- GRU와의 차이: GRU는 Γ_u 하나로 과거와 현재의 비율을 시소처럼 조절했습니다 ($1 - \Gamma_u$).
- LSTM: Γ_f와 Γ_u가 독립적입니다. 과거를 기억하면서($\Gamma_f = 1$) 동시에 새로운 중요한 정보도 추가($\Gamma_u = 1$)할 수 있어 표현력이 더 풍부합니다.

42.1.8 3. Output Generation

$$a^{\langle t \rangle} = \Gamma_o \cdot \tanh(c^{\langle t \rangle})$$

42.1.9 Comparison: GRU vs LSTM

특징	GRU	LSTM
게이트 수	2개 (Update, Reset)	3개 (Forget, Update, Output)
복잡도	단순함 (Simpler)	복잡함 (Powerful)
데이터 양	적을 때 유리	많을 때 유리 (대용량 학습)
위상	경량화 모델	업계 표준 (Default Choice)

42.1.10 Implementation: LSTM Cell

Listing 40: LSTM Cell Forward Implementation

```
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def lstm_cell_forward(xt, a_prev, c_prev, parameters):
    """
    xt: □ □ □ □
    a_prev: □ □ □ □ □ □ □ □ ()
    c_prev: □ □ □ □ □ □ □ □ ()
    """
    # □ □ □ □ □ □ (Wf, Wu, Wc, Wo □ )
```

```
    concat = np.concatenate((a_prev, xt), axis=0)

    # 1. Gates □ □
    ft = sigmoid(np.dot(parameters["Wf"], concat) + parameters["bf"]) # Forget
    it = sigmoid(np.dot(parameters["Wu"], concat) + parameters["bu"]) # Update(Input)
    ot = sigmoid(np.dot(parameters["Wo"], concat) + parameters["bo"]) # Output

    # 2. □ □ □ □ □ □ □ □ (Candidate)
    c_tilde = np.tanh(np.dot(parameters["Wc"], concat) + parameters["bc"])

    # 3. Cell State □ □ □ □ □ □ (!)
    # □ □ □ □ □ □ □ □ □ (ft), □ □ □ □ □ □ (it)
    c_next = ft * c_prev + it * c_tilde

    # 4. Hidden State □ □ □ □ □ □ ()
    a_next = ot * np.tanh(c_next)

    return a_next, c_next
```

42.1.11 FAQ Pitfalls

Q. Peephole Connection이란 뭔가요?
A. 기본 LSTM에서 게이트(Γ)는 $a^{\langle t-1 \rangle}$와 $x^{\langle t \rangle}$만 봅니다. Peephole 변형은 게이트가 Cell State($c^{\langle t-1 \rangle}$)도 훔쳐보고(Peep) 결정을 내리게 합니다. "기억통이 얼마나 찼는지 보고 문을 열지 말지 정한다"는 개념입니다.

Q. 왜 LSTM이 기울기 소실에 강한가요?
A. $c^{\langle t \rangle} = c^{\langle t-1 \rangle} + \dots$ 형태의 덧셈 연산 때문입니다. 역전파 시 덧셈은 기울기를 그대로($\times 1$) 전달하는 특성이 있어, 깊은 시간까지 오차가 잘 전달됩니다. (ResNet과 유사 원리)

☒ 다음 단계 (Next Step)

이제 우리는 시퀀스 데이터를 처리하는 가장 강력한 엔진(LSTM)을 만들었습니다. 이제 이 엔진에 넣을 연료(데이터)를 가공할 차례입니다.

컴퓨터는 '사과', '바나나'를 이해하지 못합니다. 숫자로 바꿔야 하는데, 단순히 $1, 2$로 바꾸면 단어 간 의미 관계가 사라집니다. 다음 시간에는 단어의 의미를 벡터 공간에 매핑하여 "왕 - 남자 + 여자 = 여왕"이라는 연산을 가능하게 하는 마법, [Word Embedding (Word2Vec, GloVe)]에 대해 다루겠습니다.

[단원 요약 (Cheat Sheet)]

1. **LSTM:** 3개의 게이트로 기억을 관리하는 시퀀스 모델의 표준.
2. **Structure:** 장기 기억(c)과 단기 상태(a)를 분리하여 운영한다.
3. **Gates:** Forget(삭제), Update(추가), Output(출력).
4. **Strategy:** 일단 LSTM을 기본으로 쓰고, 경량화가 필요하면 GRU를 고려하라.

43 Word Embeddings

☒ 지난 시간 복습 및 연결

지난 시간 우리는 시퀀스 데이터를 처리하는 강력한 엔진인 LSTM을 만들었습니다. 이제 이 엔진에 연료를 넣을 차례입니다. 컴퓨터는 '사과'나 '왕'이라는 글자를 이해하지 못합니다. 오직 숫자만 이해하죠. 그렇다면 단어를 어떻게 숫자로 바꿔야 할까요? 단순히 번호를 매기면(1번 사과, 2번 배) 엉뚱한 수학적 관계가 생깁니다. 우리는 단어의 의미(Meaning)를 숫자 속에 담고 싶습니다. 오늘은 NLP의 혁명, 단어 임베딩(Word Embedding)을 배웁니다.

43.1 Overview

[핵심 목표]

이 단원은 컴퓨터가 인간의 언어를 이해하는 척하게 만든 핵심 기술을 다룹니다.

- **One-hot:** 전통적인 방식의 희소성(Sparsity)과 의미 부재 문제를 이해합니다.
- **Embedding:** 단어를 실수 벡터(Dense Vector)로 표현하여 의미를 담는 원리를 파악합니다.
- **Analogy:** 벡터 연산을 통해 "남자 - 여자 = 왕 - 여왕" 관계를 도출해봅니다.
- **Matrix:** 임베딩 층이 실제로는 거대한 룩업 테이블(Lookup Table)임을 이해합니다.

43.1.1 Essential Terminology

용어	형태	특징
One-hot Encoding	$[0, 0, 1, 0, \dots]$	희소함. 단어 간 거리가 모두 같음 (관계 없음).
Word Embedding	$[0.1, -0.5, 0.9, \dots]$	밀집함. 비슷한 단어끼리 거리가 가까움.
Embedding Matrix	$E \in \mathbb{R}^{V \times D}$	모든 단어의 벡터를 담고 있는 행렬.

43.1.2 Core Concepts: 숫자에 의미를 담다

43.1.3 1. The Old Way: One-hot Encoding

단어 사전 크기가 10,000개라면 10,000차원 벡터를 만듭니다.

- Man (5391번): $[0, \dots, 1(\text{5391번째}), \dots, 0]$
- Woman (9853번): $[0, \dots, 1(\text{9853번째}), \dots, 0]$
- 문제점: 두 벡터의 내적(Dot Product)은 0입니다. 컴퓨터 입장에서 Man은 Woman과도 다르고 Apple과도 다릅니다. 유사성을 알 수 없습니다.

43.1.4 2. The New Way: Word Embedding

단어를 훨씬 작은 차원(예: 300차원)의 실수 벡터(Real-valued Vector)로 표현합니다. 각 차원은 우리가 알 수 없는(혹은 추상적인) '특징(Feature)'을 나타냅니다.

[가상의 특징표 (Featurized Representation)]

Feature	Man	Woman	King	Queen
Gender	-1	1	-0.95	0.97
Royal	0.01	0.02	0.93	0.95
Age	0.03	0.02	0.70	0.60

이제 "Man"과 "Woman" 벡터를 비교하면, Gender를 제외한 나머지 수치들이 매우 비슷합니다. 내적을 하면 값이 크게 나옵니다. 유사성을 계산할 수 있습니다.

43.1.5 3. Analogy Reasoning (유추 추론)

단어 임베딩의 가장 유명한 예시입니다.

$$e_{Man} - e_{Woman} \approx \begin{bmatrix} -2 \\ 0 \\ 0 \end{bmatrix} \quad \text{(Gender 차이만 남음)}$$

$$e_{King} - e_{Queen} \approx \begin{bmatrix} -2 \\ 0 \\ 0 \end{bmatrix} \quad \text{(Gender 차이만 남음)}$$

따라서 다음 등식이 성립합니다.

$$e_{Man} - e_{Woman} \approx e_{King} - e_{Queen}$$

$$e_{King} - e_{Man} + e_{Woman} \approx e_{Queen}$$

컴퓨터에게 "왕 - 남자 + 여자"를 계산하라면, 놀랍게도 "여왕"에 해당하는 벡터를 찾아냅니다.

43.1.6 Deep Dive: Implementation Details

43.1.7 Embedding Layer as a Lookup Table

수학적으로는 원-핫 벡터 O_j와 임베딩 행렬 E를 곱하는 것($E \cdot O_j$)입니다. 하지만 O_j는 하나만 1이고 나머지가 0이므로, 이는 결국 행렬 E의 j번째 열(Column)을 꺼내오는 것과 같습니다. 딥러닝 프레임워크는 이를 행렬 곱셈이 아닌, 인덱스 접근(Lookup)으로 구현하여 속도를 높입니다.

43.1.8 Implementation: Keras Embedding

Listing 41: Embedding Layer Usage

```
import tensorflow as tf
from tensorflow.keras.layers import Embedding

# □□
vocab_size = 10000  # □□ □□ □□ (V)
embedding_dim = 300 # □□□ □□ (D)
input_length = 10   # □□ □□ □□ (T)

# □□ □□
model = tf.keras.Sequential()

# Embedding Layer
# □ □□ (10000, 300) □□□ □□ □E□□□□.
# □□ □□□□ □□ □□□□, □□□□ □□ □□□''□□□□□□.
model.add(Embedding(input_dim=vocab_size,
```

```
                    output_dim=embedding_dim,
                    input_length=input_length))

# □□ □□
model.summary()

# --- □□ □□□□ □□ ---
# "I love you" -> [1, 539, 23, 0, 0...] □□ ( □□□□□ □□ )
# □□ shape: (Batch, 10)
# □□ shape: (Batch, 10, 300) -> □ □□□□ □□ 300 □□□□ □□
```

43.1.9 FAQ Pitfalls

Q. 임베딩 값은 어떻게 학습하나요?
A. 두 가지 방법이 있습니다. 1. End-to-End: 내 문제(예: 감성 분석)를 풀면서 임베딩 층도 같이 학습시킵니다. 데이터가 많아야 합니다. 2. Pre-trained: Word2Vec이나 GloVe처럼 위키피디아 같은 방대한 텍스트로 미리 학습된 임베딩 행렬을 가져와서 씁니다. (전이 학습, 추천!)

Q. 임베딩 차원(300 등)은 어떻게 정하나요?
A. 하이퍼파라미터입니다. 보통 50, 100, 300 등을 많이 씁니다. 차원이 클수록 더 정교한 의미를 담을 수 있지만, 메모리와 계산량이 늘어납니다.

☒ 다음 단계 (Next Step)

이제 우리는 단어를 벡터로 바꾸는 '개념'을 알았습니다. 그렇다면 이 벡터 값(숫자들)은 도대체 어떻게 구하는 걸까요? 사람이 일일이 입력할 수는 없습니다.

컴퓨터가 방대한 텍스트를 읽으면서 "단어의 의미는 그 주변 단어들에 의해 결정된다"는 원리를 이용해 스스로 학습하게 해야 합니다. 다음 시간에는 임베딩 학습 알고리즘의 양대 산맥, [Word2Vec (Skip-gram, CBOW)]과 [GloVe]에 대해 알아보겠습니다.

[단원 요약 (Cheat Sheet)]

1. **One-hot:** 단어 간 유사성을 표현하지 못하는 희소 벡터.
2. **Embedding:** 단어 간 유사성을 내적(거리)으로 계산할 수 있는 밀집 벡터.
3. **Analogy:** 벡터 연산으로 단어 관계(성별, 시제 등)를 추론 가능함.
4. **Layer:** 임베딩 층은 거대한 룩업 테이블(Lookup Table)이다.

44 Word2Vec GloVe

☒ 지난 시간 복습 및 연결

지난 시간에 우리는 단어를 벡터로 바꾸는 임베딩의 개념을 배웠습니다. '왕'과 '남자'가 가깝다는 것을 알았죠. 그렇다면 이 마법 같은 벡터 값들은 어떻게 구할까요? 언어학자 존 퍼스는 말했습니다. "단어의 의미는 그 단어의 친구들(주변 단어)을 보면 알 수 있다." 오늘 배울 알고리즘들은 이 철학을 구현한 것입니다. 방대한 텍스트를 읽으며 단어의 관계를 파악하고 의미를 학습하는 Word2Vec과 GloVe에 대해 알아봅니다.

44.1 Overview

[핵심 목표]

이 단원은 단어 임베딩을 학습하는 대표적인 두 가지 알고리즘을 다룹니다.

- **Skip-gram:** 중심 단어로 주변 단어를 예측하며 학습하는 방식을 이해합니다.
- **Negative Sampling:** 거대한 Softmax 연산을 피하고 속도를 높이는 최적화 기법을 배웁니다.
- **CBOW:** 주변 단어로 중심 단어를 예측하는 방식과 Skip-gram의 차이를 비교합니다.
- **GloVe:** 전체 말뭉치의 동시 등장 행렬을 활용하는 통계적 방법을 파악합니다.

44.1.1 Essential Terminology

알고리즘	방식	특징
Skip-gram	중심 $\rightarrow$ 주변 예측	희귀 단어 학습에 유리함 (널리 쓰임).
CBOW	주변 $\rightarrow$ 중심 예측	학습 속도가 빠름.
Negative Sampling	이진 분류 문제로 변환	10만 개 클래스 분류를 O/X 문제로 바꿈.
GloVe	행렬 분해 (Matrix Factorization)	전체 통계 정보를 직접 활용함.

44.1.2 Core Concepts: 친구를 보면 너를 안다

44.1.3 1. Word2Vec: Skip-gram Model

우리는 라벨이 없는 텍스트를 지도 학습(Supervised Learning) 문제로 바꿉니다. **문장:** "I want a glass of **orange** juice to drink."

- Input (Context): orange (중심 단어)
- Target: juice (주변 단어)
- Task: "orange가 나왔을 때, 주변에 juice가 나올 확률은?"

44.1.4 2. The Problem with Softmax

$$P(t|c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{V} e^{\theta_j^T e_c}}$$

분모의 합($\sum$)을 구하려면 사전(Vocabulary)에 있는 10만 개 단어를 다 계산해야 합니다. 학습 한 번 할 때마다 10만 번 연산? 너무 느립니다.

44.1.5 3. The Solution: Negative Sampling

구글 연구팀은 문제를 바꿨습니다. "10만 개 중 하나 맞추기" → "이게 진짜 쌍(Pair)이냐 가짜냐(Binary)?"

- Positive Pair (진짜): (orange, juice) → Label 1
- Negative Pair (가짜): (orange, king), (orange, book) ... → Label 0

이제 1개의 진짜와 k개의 가짜(보통 5 20개)만 계산하면 됩니다. 연산량이 1/10000로 줄어듭니다. 이것이 Word2Vec의 핵심입니다.

44.1.6 GloVe (Global Vectors)

Word2Vec은 윈도우를 슬라이딩하며 국소적(Local) 정보만 봅니다. GloVe는 전체 통계(Global Statistics)를 봅니다.
Co-occurrence Matrix (동시 등장 행렬)

$$X_{ij} = \text{단어 } i \text{ 주변에 단어 } j\text{가 등장한 횟수}$$

GloVe의 목표는 임베딩 벡터의 내적이 이 횟수의 로그값과 비슷해지는 것입니다.

$$w_i^T w_j + b_i + b_j \approx \log(X_{ij})$$

44.1.7 Implementation: Gensim Word2Vec

실무에서는 직접 구현하기보다 최적화된 라이브러리 'Gensim'을 씁니다.

Listing 42: Word2Vec Training with Gensim

```
from gensim.models import Word2Vec

# 1. □□ □□□ □□□□□(□□ □□□)
sentences = [
    ['I', 'love', 'machine', 'learning'],
    ['deep', 'learning', 'is', 'fun'],
    ['machine', 'learning', 'is', 'hard'],
    ['orange', 'juice', 'is', 'delicious']
]

# 2. □□ □□
# vector_size: □□□ □□ (100~300)
# window: □□ □□ □□ (5)
# sg: 1=Skip-gram, 0=CBOW
model = Word2Vec(sentences, vector_size=10, window=2, min_count=1, sg=1)

# 3. □□ □□
vector = model.wv['machine']
print(f"Vector:\n{vector}")

# 4. □□□ □□ □□()
similar = model.wv.most_similar('deep')
print(f"Similar to 'deep': {similar}")
```

44.1.8 FAQ Pitfalls

[데이터 양이 중요합니다]

위의 예제처럼 문장 4개로는 아무것도 못 배웁니다. Word2Vec이 제대로 작동하려면 위키피디아 전체 같은 대용량 텍스트가 필요합니다. 실무에서는 보통 구글이나 페이스북이 미리 학습해둔 모델(Pre-trained)을 다운받아 씁니다.

Q. Word2Vec과 GloVe 중 뭐가 더 좋나요?

A. 비슷합니다. 어떤 데이터셋에서는 Word2Vec이, 어떤 곳에서는 GloVe가 낫습니다. 보통은 둘 다 써보고 성능 좋은 걸 택하거나, 최근에는 BERT 같은 문맥 기반 임베딩을 씁니다.

☒ 다음 단계 (Next Step)

이제 우리는 단어 하나하나를 의미 있는 벡터로 바꾸는 법을 알았습니다. 하지만 "I ate an apple"과 "An apple was eaten by me"는 단어 순서가 다르지만 의미는 같습니다. 반면 RNN은 길이가 길어지면 앞을 잊어버리는 문제가 여전합니다.

다음 시간에는 입력 시퀀스 전체를 보고 번역하거나 요약하는 [Seq2Seq] 모델과, 긴 문장에서도 중요한 단어에 집중하게 만드는 [Attention Mechanism]을 배우겠습니다. 이것이 현대 AI의 정점인 Transformer로 가는 마지막 관문입니다.

[단원 요약 (Cheat Sheet)]

1. **Skip-gram:** 중심 단어로 주변 단어를 예측한다. (희귀 단어에 강함)
2. **Negative Sampling:** 전체 Softmax 대신 O/X 문제로 바꿔 속도를 높인다.
3. **GloVe:** 전체 말뭉치의 통계(동시 등장 횟수)를 직접 활용한다.
4. **Tip:** 데이터가 적을 땐 Pre-trained 모델을 써라.

45 Bias in Embeddings

☒ 지난 시간 복습 및 연결

지난 시간에 배운 Word2Vec은 놀라웠습니다. 하지만 2016년, 연구자들은 충격적인 사실을 발견했습니다. "남자에게 프로그래머가 있다면, 여자에게는 누가 있는가?" 모델의 대답은 "가정주부(Homemaker)"였습니다. 이는 모델의 잘못이 아닙니다. 모델은 인간이 쓴 텍스트의 편향(Gender Bias)을 그대로 학습했을 뿐입니다. 오늘은 AI 윤리의 핵심, 임베딩 편향 제거를 배웁니다.

45.1 Overview

[핵심 목표]

이 단원은 학습된 임베딩 벡터에서 사회적 편향을 수학적으로 제거하는 알고리즘을 다룹니다.

- **식별:** 편향 방향(Gender Axis)을 벡터 공간에서 찾아냅니다.
- **중화 (Neutralize):** 의사, 간호사 등 중립 단어에서 편향 성분을 제거합니다 (투영).
- **균형화 (Equalize):** 할머니, 할아버지 등 성별 단어가 중립 단어와 같은 거리를 갖도록 조정합니다.

45.1.1 Essential Terminology

용어	의미	예시
Bias Axis	성별을 나타내는 벡터 방향.	$\vec{he} - \vec{she}$ 의 평균.
Neutral Words	성별과 무관해야 하는 단어.	Doctor, Nurse, Teacher.
Gender Specific	성별이 정의상 필요한 단어.	Grandfather, Queen, Girl.

45.1.2 Core Concepts: 편향의 기하학

45.1.3 1. Identifying Bias (편향 식별)

임베딩 공간에서 성별 편향은 다음과 같이 나타납니다.

$$Man : Woman :: King : Queen \quad \text{(적절함)}$$

$$Man : Woman :: Doctor : \textbf{Nurse} \quad \text{(부적절함 - 고정관념)}$$

45.1.4 2. The Debiasing Algorithm (3단계)

45.1.5 Step 1: Identify Bias Direction (축 찾기)

성별을 나타내는 단어 쌍들의 차이를 평균 내어 성별 축(Gender Axis) g를 정의합니다.

$$g \approx \text{Average}(\vec{he} - \vec{she}, \vec{male} - \vec{female}, \dots)$$

45.1.6 Step 2: Neutralize (중화)

"Doctor" 같은 중립 단어는 성별 축 성분을 가져서는 안 됩니다. Action: 단어 벡터 w를 성별 축 g에 투영(Project)하여 편향 성분을 뺍니다.

중화 공식 (Projection)

$$w_{debiased} = w - \frac{w \cdot g}{\|g\|^2} g$$

벡터 w에서 성별 축 방향의 그림자(성분)를 제거하여, 성별 축과 수직이 되게 만듭니다.

45.1.7 Step 3: Equalize (균형화)

"Grandmother"와 "Grandfather"는 성별 정보를 가져야 하므로 중화하면 안 됩니다. 하지만 "Babysitter"와의 거리는 공평해야 합니다. Action: 두 단어가 중립 축에서 등거리(Equidistant)에 위치하도록 미세 조정합니다.

45.1.8 Implementation: Neutralize Function

벡터 투영을 통해 편향을 제거하는 함수입니다.

Listing 43: Neutralize Implementation

```
import numpy as np

def neutralize(curr_embedding, g):
    """
␣␣␣␣curr_embedding:␣□ □ □ ␣□ □ ␣□ □ ␣□ (:␣doctor)
␣␣␣␣g:␣□ □ ␣□ □ ␣□ □ ␣(bias␣axis)
␣␣␣␣"""
    # 1. □ □  □ □  □ □  (Projection)
    # (w . g / ||g||^2) * g
    norm_sq = np.linalg.norm(g)  2
    projection = (np.dot(curr_embedding, g) / norm_sq) * g

    # 2. □ □ □ □  □ □  □ □
    debiased_embedding = curr_embedding - projection

    return debiased_embedding

# --- □ □  ---
if __name__ == "__main__":
    g = np.array([1.0, 0.0]) # □ □ x □ □  □ □ □ □  □ □
    doctor = np.array([2.0, 4.0]) # □ □  □ □ (2.0)□  □ □ □

    # □ □
    doctor_debiased = neutralize(doctor, g)

    print(f"Original:␣{doctor}")
    print(f"Debiased:␣{doctor_debiased}")
    # □ □ : [0.0, 4.0] □ □ ( □ □ □  □ 0 □ )
```

45.1.9 FAQ Pitfalls

[모든 단어를 중화하면 안 됩니다!]

"King", "Queen", "Actor", "Actress" 같은 단어는 성별 정보가 그 단어의 핵심 의미입니다. 이런 단어들을 중화해버리면 "King"과 "Queen"이 똑같은 단어가 되어버립니다. 따라서 성별 정의 단어(Gender Definition Words)를 분류하는 작업이 선행되어야 합니다.

Q. 이 방법으로 모든 편향이 사라지나요?

A. 완벽하진 않습니다. 'Hard Debiasing'은 벡터의 방향만 수정하지만, 단어들이 뭉쳐 있는 클러스터링 구조 등 간접적인 편향은 남을 수 있습니다. 하지만 매우 효과적인 첫걸음입니다.

☒ 다음 단계 (Next Step)

이제 우리는 단어를 윤리적으로 올바른 벡터로 표현하는 법까지 배웠습니다. 준비는 끝났습니다.

이제 단어들을 조합해 문장을 통째로 번역하거나 요약하는 거대한 모델을 만들어 봅시다. 다음 시간에는 입력 시퀀스를 압축했다가 다시 풀어내는 [Seq2Seq Model]과, 긴 문장 처리에 필수적인 [Beam Search] 알고리즘에 대해 다루겠습니다.

[단원 요약 (Cheat Sheet)]

1. **Bias:** 학습 데이터의 사회적 편향이 임베딩 벡터에 반영된다.

2. **Neutralize:** 중립 단어(직업 등)를 성별 축에 수직이 되도록 투영한다.
3. **Equalize:** 성별 단어(할머니/할아버지)가 중립 단어와 등거리에 있게 한다.
4. **Ethics:** AI 모델 배포 전 편향 제거는 필수적인 전처리 과정이다.

46 Sequence-to-Sequence Models

☒ 지난 시간 복습 및 연결

지난 시간까지 우리는 단어를 벡터로 바꾸는 법(Embedding)을 배웠습니다. 이제 단어들을 조립해 문장을 만들고, 번역하는 시스템을 만들 차례입니다. 그런데 문제가 있습니다. 입력: "I love you" (3단어) → 출력: "Je t'aime" (2단어). 길이가 다릅니다. 기존 RNN은 이를 처리할 수 없습니다. 이를 해결하기 위해 구글이 제안한 혁명적인 아키텍처, Seq2Seq (Encoder-Decoder)를 배웁니다.

46.1 Overview

[핵심 목표]

이 단원은 입력과 출력의 길이가 다른 시퀀스 변환 모델을 다룹니다.

- **구조:** 입력 시퀀스를 압축하는 인코더와, 이를 풀어내는 디코더 구조를 이해합니다.
- **압축:** 인코더의 마지막 은닉 상태가 문장 전체의 의미를 담은 문맥 벡터(Context Vector)가 됨을 파악합니다.
- **전달:** 문맥 벡터가 디코더의 초기 상태로 전달되는 흐름을 코드로 구현합니다.
- **한계:** 문장이 길어지면 정보가 손실되는 병목(Bottleneck) 현상을 인지합니다.

46.1.1 Essential Terminology

용어	역할	비유
Encoder	입력 문장을 읽고 이해함.	문장을 읽는 번역가.
Decoder	이해한 내용을 바탕으로 문장을 생성함.	번역문을 쓰는 번역가.
Context Vector	인코더가 넘겨주는 핵심 정보 (마지막 h).	머릿속에 정리된 문장의 의미.

46.1.2 Core Concepts: 다 듣고 말하기

46.1.3 1. The Architecture: Encoder-Decoder

Seq2Seq는 두 개의 RNN을 이어 붙인 구조입니다.

- Encoder: 문장 $x^{\langle 1 \rangle} \dots x^{\langle T_x \rangle}$를 순서대로 읽습니다. 출력을 내지 않고 은닉 상태(h)만 업데이트합니다. 마지막 상태 $h^{\langle T_x \rangle}$에 모든 정보를 압축합니다.
- Decoder: 인코더가 준 문맥 벡터를 자신의 초기 상태($h_{dec}^{\langle 0 \rangle}$)로 받습니다. 이를 바탕으로 번역문을 생성합니다.

[번역가의 작업 방식]

- Read (Encoder): 문장을 끝까지 다 읽습니다. 중간에 번역하지 않습니다. 머릿속에 핵심 의미(Context)를 정리합니다.
- Write (Decoder): 머릿속 의미를 바탕으로 프랑스어 문장을 처음부터 써 내려갑니다.

46.1.4 2. Training Trick: Teacher Forcing

학습할 때는 디코더가 실수로 엉뚱한 단어를 예측했더라도, 다음 스텝 입력으로는 정답 단어(Ground Truth)를 넣어줍니다. 이를 교사 강요(Teacher Forcing)라고 합니다. (초반 학습 안정화용)

46.1.5 Implementation: Keras Functional API

인코더의 상태를 디코더로 넘겨주는 핵심 코드를 작성합니다.

Listing 44: Seq2Seq Model Definition

```python
import tensorflow as tf
from tensorflow.keras.layers import Input, LSTM, Dense, Embedding
from tensorflow.keras.models import Model

def build_seq2seq(vocab_size, embedding_dim, latent_dim):

    # --- 1. Encoder ---
    enc_inputs = Input(shape=(None,))
    enc_emb = Embedding(vocab_size, embedding_dim)(enc_inputs)

    # return_state=True: □ □ □  □ □  □ □ (h)□  □  □ □ (c)□  □ □
    encoder_lstm = LSTM(latent_dim, return_state=True)
    _, state_h, state_c = encoder_lstm(enc_emb)

    # □ □  □ □  (Context Vector): □ □ □  □ □  □ □ □  □ □ □  □
    encoder_states = [state_h, state_c]

    # --- 2. Decoder ---
    dec_inputs = Input(shape=(None,))
    dec_emb = Embedding(vocab_size, embedding_dim)(dec_inputs)

    # return_sequences=True: □ □  □ □  □ □  □ □  □ □ □ ( □ □ )
    decoder_lstm = LSTM(latent_dim, return_sequences=True, return_state=True)

    # □ □ : □ initial_state □ □ □ □  □ □ □ □  □ □ !
    dec_outputs, _, _ = decoder_lstm(dec_emb, initial_state=encoder_states)

    decoder_dense = Dense(vocab_size, activation='softmax')
    dec_outputs = decoder_dense(dec_outputs)

    # □ □  □ □
    model = Model([enc_inputs, dec_inputs], dec_outputs)
    return model
```

46.1.6 FAQ Pitfalls

[The Bottleneck Problem (병목 현상)]

Seq2Seq의 치명적 단점은 인코더가 문장이 길든 짧든 고정된 크기의 벡터 하나에 모든 정보를 우겨 넣어야 한다는 점입니다. 문장이 길어지면(50단어 이상) 정보 손실이 발생해 번역 품질이 급격히 떨어집니다. 이를 해결하기 위해 Attention Mechanism이 등장했습니다.

Q. 추론(Inference) 때는 어떻게 하나요?

A. 학습 때와 달리 정답을 모르므로, 디코더가 방금 예측한 단어를 다음 스텝의 입력으로 넣어주는 루프(Loop)를 직접 구현해야 합니다. (Auto-regressive)

☒ 다음 단계 (Next Step)

이제 모델을 만들었습니다. 그런데 디코더가 단어를 생성할 때, 매 순간 가장 확률 높은 단어 하나만(Greedy) 고르면 최고의 문장이 될까요? "The" 뒤에 "nice"가 올 확률이 높다고 골랐는데, 나중에 보니 "The huge building..."이 더 좋은 번역일 수도 있습니다.

다음 시간에는 번역 품질을 결정하는 결정적 탐색 알고리즘, [Beam Search]에 대해 알아보고, Seq2Seq의 병목 문제를 해결하는 [Attention Mechanism]으로 넘어가겠습니다.

[단원 요약 (Cheat Sheet)]

1. **Structure:** Encoder가 압축하고 Decoder가 푼다.

2. **Context:** 인코더의 마지막 상태가 문장의 핵심 의미를 담은 매개체다.
3. **Transfer:** h_{enc}를 h_{dec}의 'initial$_state$'로전달한다.
3. **Limit:** 긴 문장을 하나의 벡터로 압축하면 정보 손실이 발생한다.

47 Beam Search

☒ 지난 시간 복습 및 연결

우리는 Seq2Seq 모델로 번역기를 만들었습니다. 그런데 막상 돌려보니 엉뚱한 문장이 나옵니다. 이유는 우리가 '당장의 최선(Greedy)'만 선택했기 때문입니다. 1등만 뽑아서 문장을 이었더니 전체 문맥은 엉망이 된 것이죠. 인생과 마찬가지로, "당장의 최선이 결과적인 최선은 아닐 수" 있습니다. 여러 가능성을 동시에 탐색하는 빔 서치(Beam Search)가 필요합니다.

47.1 Overview

[핵심 목표]

이 단원은 텍스트 생성 모델의 성능을 결정짓는 탐색 알고리즘을 다룹니다.

- **Greedy:** 매 순간 1등만 뽑는 방식의 한계(Local Optima)를 이해합니다.
- **Beam:** 상위 B개의 가능성을 살려두며 탐색하는 원리를 파악합니다.
- **Refinement:** 로그 우도(Log Likelihood)와 길이 정규화(Length Normalization)를 통해 수학적 안정성을 확보합니다.

47.1.1 Essential Terminology

용어	의미	특징
Greedy Search	매번 확률 1등 단어만 선택.	빠르지만 최적해 보장 못함.
Beam Search	매번 상위 B개를 유지하며 탐색.	느리지만 더 좋은 문장 생성.
Beam Width (B)	유지할 후보 개수 (보통 3 10).	$B = 1$이면 Greedy와 같음.

47.1.2 Core Concepts: 여러 길을 동시에 가라

47.1.3 1. The Problem: Greedy Search

매 스텝에서 확률이 가장 높은 단어 1개만 고르고 다음으로 넘어갑니다.

[미로 찾기]

갈림길에서 무조건 "출구와 가까워 보이는 쪽"으로만 가는 것과 같습니다. 가다 보니 막다른 길일 수 있지만, 되돌아갈 수 없습니다(No Backtracking).

47.1.4 2. The Solution: Beam Search

매 순간 상위 B개의 가능성(Hypothesis)을 유지합니다. ($B = 3$ 가정)

- Step 1: 첫 단어 후보 중 상위 3개를 뽑습니다. (예: "in", "the", "a")
- Step 2: 살아남은 3개의 단어 각각에 대해, 다음 단어 확률을 계산합니다.
- Step 3: 총 30,000개(3 $\times$ 단어장) 조합 중 다시 상위 3등까지만 남기고 나머지는 버립니다.
- Repeat: 문장이 끝날 때까지 반복합니다.

47.1.5 Deep Dive: Refinements (정밀화)

단순 확률 곱셈에는 수학적 문제가 있습니다. 이를 보정해야 합니다.

47.1.6 1. Log Likelihood (로그 우도)

확률(0.1, 0.05...)을 수십 번 곱하면 값이 $0.0000\ldots$ 이 되어 컴퓨터가 0으로 인식해버립니다 (Underflow). 해결책은 로그(log)를 취하는 것입니다. 곱셈이 덧셈으로 변합니다.

Score Function

$$\sum_{t=1}^{T_y} \log P(y^{\langle t \rangle} | y^{\langle 1 \ldots t-1 \rangle}, x)$$

로그 합을 최대화하는 것은 확률 곱을 최대화하는 것과 수학적으로 동일합니다.

47.1.7 2. Length Normalization (길이 정규화)

로그 확률(음수)을 계속 더하면, 문장이 길어질수록 점수는 계속 낮아집니다. 모델이 "짧은 문장"을 과도하게 선호하게 됩니다. 해결책은 점수를 문장 길이(T_y)로 나누어 평균을 내는 것입니다.

Normalized Score

$$\text{Score} = \frac{1}{T_y^{\alpha}} \sum \log P(\ldots)$$

- α: 보정 계수 (보통 0.7).
- 긴 문장에 대한 페널티를 완화해줍니다.

47.1.8 FAQ Pitfalls

Q. B를 무조건 크게 하면 좋은가요?
A. 아닙니다. B가 크면 더 좋은 문장을 찾을 확률은 높지만, 계산량과 메모리가 B배로 늘어납니다. 실무에서는 보통 $B = 3 \sim 10$ 정도를 쓰며, 아주 정밀한 번역이 필요할 때만 더 키웁니다.

Q. 빔 서치는 최적해(Global Optima)를 보장하나요?
A. 아니요. B가 무한대(BFS)가 아닌 이상, 휴리스틱 탐색이므로 완벽한 최적해를 보장하진 않습니다. 하지만 Greedy보다는 훨씬 낫습니다.

☒ 다음 단계 (Next Step)

빔 서치로 문장 생성 능력은 좋아졌습니다. 하지만 여전히 근본적인 문제가 남았습니다. Seq2Seq의 인코더는 아무리 긴 문장이라도 하나의 고정된 벡터(Context Vector)에 욱여넣어야 한다는 '병목(Bottleneck)' 현상입니다.

"번역할 때 문장 전체를 외우고 하는 사람은 없습니다. 필요한 단어를 그때그때 다시 보면서(Attention) 하죠." 다음 시간에는 딥러닝 역사상 가장 위대한 발명 중 하나인 [Attention Mechanism]을 통해 이 병목을 부수고 Transformer로 가는 문을 열겠습니다.

[단원 요약 (Cheat Sheet)]

1. **Greedy:** 빠르지만 시야가 좁아 최적 문장을 놓친다.
2. **Beam Search:** B개의 유망주를 끝까지 키운다.
3. **Log:** 언더플로우 방지를 위해 확률 곱 대신 로그 합을 쓴다.
4. **Normalize:** 짧은 문장 편향을 막기 위해 길이로 나눈다.

48 Bleu Score

☒ 지난 시간 복습 및 연결

지난 시간에 빔 서치로 최적의 문장을 생성하는 법을 배웠습니다. 그런데 근본적인 질문이 남습니다. "번역이 잘 됐는지 컴퓨터가 어떻게 채점할까?" 이미지 분류는 정답이 하나(고양이)지만, 번역은 정답이 여러 개입니다. ("나는 학교에 간다", "학교에 가는 중이다" 등) 사람이 일일이 채점하기엔 너무 비쌉니다. 이를 해결하기 위해 IBM이 제안한 자동화된 점수, BLEU Score를 배웁니다.

48.1 Overview

[핵심 목표]

이 단원은 기계 번역 평가의 표준인 BLEU Score의 원리를 다룹니다.

- **다중 정답:** 여러 개의 참고 문장(Reference)을 기준으로 평가합니다.
- **정밀도:** 단순 일치가 아닌 클리핑된 정밀도(Clipped Precision)를 사용해 "the the the" 문제를 해결합니다.
- **N-gram:** 단어 묶음을 비교하여 문맥과 어순의 정확성을 평가합니다.
- **페널티:** 너무 짧은 문장에 페널티(BP)를 주어 꼼수를 방지합니다.

48.1.1 Essential Terminology

용어	의미	역할
Reference	정답 문장(들).	채점 기준. (사람이 번역한 것)
Candidate	모델이 생성한 문장.	채점 대상.
Modified Precision	빈도수 제한 정밀도.	중복 단어 남발 방지.
Brevity Penalty	길이 불이익.	정보 누락(짧은 문장) 방지.

48.1.2 Core Concepts: 정밀도의 함정

48.1.3 1. The Problem with Standard Precision

Candidate: "the the the the the" (5단어) **Reference:** "The cat is on the mat."

- 단순 정밀도: 5개 모두 정답에 있음("the"). $\rightarrow$ 5/5 = 100점? (말도 안 됨)
- 해결책 (Clipping): 정답 문장에 "the"가 최대 몇 번 나오는지 셉니다. (2번). 분자를 2로 제한합니다. $\rightarrow$ 2/5 = 40점.

48.1.4 2. N-gram Analysis (어순 평가)

단어만 다 있다고 문장이 아닙니다. "The cat the mat on is" (단어는 맞지만 엉터리) 이를 잡기 위해 N-gram(단어 묶음)을 봅니다.

- Unigram (1-gram): 단어 존재 여부 (내용 충실도).
- Bigram (2-gram): "The cat", "cat is" ... (어순/유창성).

48.1.5 3. Brevity Penalty (BP)

정밀도 기반 평가는 "짧게 말하면 유리하다"는 약점이 있습니다. **Cand:** "The cat" (2단어 다 맞음. 정밀도 100하지만 정보가 누락되었습니다. 예측 문장이 정답보다 짧으면 점수를 깎습니다.

48.1.6 Deep Dive: The Formula

BLEU Formula

$$\text{BLEU} = BP \times \exp\left(\frac{1}{4}\sum_{n=1}^{4} p_n\right)$$

- p_n: n-gram의 보정된 정밀도.
- BP: Brevity Penalty (길이 페널티).
- $\exp(\dots)$: 기하 평균(Geometric Mean)을 구하는 방식.

48.1.7 Implementation: NLTK Library

파이썬 NLTK 라이브러리로 쉽게 계산할 수 있습니다.

Listing 45: BLEU Score Calculation

```
from nltk.translate.bleu_score import sentence_bleu

# 1. □□□ □□
# □ Reference □□ □□ □ □□ □□□□( □□□ )
references = [
    ['the', 'cat', 'is', 'on', 'the', 'mat'],       # Ref 1
    ['there', 'is', 'a', 'cat', 'on', 'the', 'mat'] # Ref 2
]

# □ Candidate □□□ □□□
candidate = ['the', 'cat', 'is', 'on', 'the', 'mat']

# 2. BLEU □□
# weights: 1-gram ~ 4-gram □□□ □□( □□□□ 0.25)
score = sentence_bleu(references, candidate, weights=(0.25, 0.25, 0.25, 0.25))

print(f"BLEU Score: {score:.4f}")
```

48.1.8 FAQ Pitfalls

Q. BLEU 점수가 높으면 완벽한 번역인가요?
A. 아니요. BLEU는 단어 일치도만 봅니다. 의미는 통하는데 단어가 다른 의역(Paraphrasing)이나, 미묘한 뉘앙스 차이는 잡아내지 못합니다. 하지만 "최소한 이 정도는 한다"는 지표로는 가장 훌륭합니다.

Q. 만약 4-gram이 하나도 안 맞으면요?
A. $p_4 = 0$이 되면 기하 평균 특성상 전체 점수가 0이 되어버립니다. 이를 막기 위해 아주 작은 값(ϵ)을 더해주는 Smoothing 기법을 씁니다.

☒ 다음 단계 (Next Step)

우리는 빔 서치로 문장을 만들고, BLEU로 평가까지 했습니다. 하지만 Seq2Seq에는 여전히 "긴 문장의 병목 현상"이라는 거대한 벽이 남아 있습니다. 인코더가 100단어를 벡터 하나에 압축하는 것은 무리입니다.

"번역할 때 문장 전체를 외우지 않고, 필요한 단어만 그때그때 다시 보면서(Attend) 번역할 수는 없을까?" 다음 시간, 딥러닝 역사상 가장 위대한 발명 중 하나인 [Attention Mechanism]을 통해 이 벽을 부수겠습니다. 이것이 바로 Transformer와 GPT로 가는 마지막 열쇠입니다.

[단원 요약 (Cheat Sheet)]

1. **Modified Precision:** 정답에 등장하는 횟수만큼만 인정하여 중복 꼼수를 막는다.
2. **N-gram:** 1 4단어 묶음을 비교하여 문맥과 유창성을 평가한다.
3. **BP:** 문장이 짧으면 점수를 깎아 정보 누락을 방지한다.
4. **Standard:** 기계 번역 성능 평가의 업계 표준이다.

49 Attention Mechanism

☒ 지난 시간 복습 및 연결

지난 시간에 배운 Seq2Seq 모델은 혁신적이었지만, 문장이 길어지면 성능이 급격히 떨어지는 '병목(Bottleneck)' 현상을 겪었습니다. 인코더가 100단어짜리 긴 문장을 고정된 크기의 벡터 하나에 억지로 압축해야 했기 때문입니다.

생각해 봅시다. 여러분이 긴 문장을 번역할 때, 문장 전체를 한 번 읽고 외운 다음 눈을 감고 번역합니까? 아닙니다. "지금 번역하려는 단어와 관련된 원문 부분을 그때그때 다시 쳐다보면서(Attend)" 번역합니다. 이 과정을 구현한 것이 바로 어텐션(Attention)입니다.

49.1 Overview

[핵심 목표]

이 단원은 긴 시퀀스 정보 손실을 해결하고 성능을 극대화하는 어텐션 메커니즘을 다룹니다.

- **동적 문맥:** 매 타임 스텝마다 다르게 계산되는 문맥 벡터 $c^{\langle t \rangle}$의 개념을 이해합니다.
- **가중치:** 인코더의 특정 단어에 얼마나 집중할지 나타내는 어텐션 가중치(α)를 정의합니다.
- **구조:** 인코더의 모든 은닉 상태를 가중 합(Weighted Sum)하여 정보를 취합하는 원리를 배웁니다.
- **학습:** 가중치를 결정하는 정렬 모델(Alignment Model)도 역전파로 함께 학습됨을 파악합니다.

49.1.1 Essential Terminology

용어	의미	비유
Attention Weight	$\alpha^{\langle t,t' \rangle}$: t번째 출력 시 t'번째 입력의 중요도.	돋보기로 비추는 빛의 세기.
Context Vector	$c^{\langle t \rangle}$: 인코더 상태들의 가중 평균.	번역 중인 단어의 연관 참고 자료.
Alignment Score	단어 간의 유사도/관련성 점수.	두 단어가 얼마나 '찰떡궁합'인가.

49.1.2 Core Concepts: 필요한 곳만 쳐다보기

49.1.3 1. The Intuition (직관)

기존 Seq2Seq는 인코더의 마지막 정보 하나만 디코더에게 던져줍니다. 반면, 어텐션 모델은 디코더가 단어를 생성할 때마다 인코더의 모든 은닉 상태($a^{\langle 1 \rangle}, a^{\langle 2 \rangle}, \dots$)를 다 참고합니다. 단, 중요한 것은 진하게(높은 가중치), 안 중요한 것은 흐리게 봅니다.

[번역가의 힐끔거리기]

- 기존: 문장을 한 번 읽고 책을 덮은 뒤 암기해서 번역함. (병목 발생)
- 어텐션: 번역문을 쓰면서 원문에서 지금 단어와 관련된 부분(예: 주어, 목적어)을 계속 힐끔힐끔 다시 보며 번역함.

49.1.4 2. The Architecture: Context Vector $c^{\langle t \rangle}$

문맥 벡터 $c^{\langle t \rangle}$는 인코더의 모든 은닉 상태 $a^{\langle t' \rangle}$의 가중 평균입니다.

Context Vector 공식

$$c^{\langle t \rangle} = \sum_{t'} \alpha^{\langle t,t' \rangle} a^{\langle t' \rangle}$$

- $\alpha^{\langle t,t' \rangle}$: 디코더 t 시점에 인코더 t' 시점을 보는 비중.
- $\sum_{t'} \alpha^{\langle t,t' \rangle} = 1$ (확률값).

49.1.5 Deep Dive: Computing Attention (수학적 원리)

그렇다면 가장 중요한 α (가중치)는 누가 정할까요? 사람이 정하는 게 아니라, 이조차 작은 신경망(Alignment Model)이 학습합니다.

1. **Alignment Score ($e^{\langle t,t' \rangle}$):** 디코더의 이전 상태 $s^{\langle t-1 \rangle}$와 인코더 상태 $a^{\langle t' \rangle}$가 얼마나 어울리는지 점수를 매깁니다.
$$e^{\langle t,t' \rangle} = \text{dense\_layer}([s^{\langle t-1 \rangle}, a^{\langle t' \rangle}])$$
2. **Softmax (Weights):** 점수를 확률로 변환합니다.
$$\alpha^{\langle t,t' \rangle} = \frac{\exp(e^{\langle t,t' \rangle})}{\sum_{k=1}^{T_x} \exp(e^{\langle t,k \rangle})}$$

49.1.6 Implementation: Bahdanau Attention

TensorFlow/Keras의 서브클래싱 방식으로 구현한 예시입니다.

Listing 46: Bahdanau Attention Layer

```python
import tensorflow as tf
from tensorflow.keras.layers import Layer, Dense

class BahdanauAttention(Layer):
    def __init__(self, units):
        super().__init__()
        self.W1 = Dense(units) # Decoder state weight
        self.W2 = Dense(units) # Encoder state weight
        self.V = Dense(1)      # Score weight

    def call(self, query, values):
        # query: □ □ □  □ □  □ □  (Batch, hidden)
        # values: □ □ □  □ □  □ □ □  (Batch, Tx, hidden)

        query_with_time = tf.expand_dims(query, 1)

        # 1. □ □  □ □  (Additive Attention)
        score = self.V(tf.nn.tanh(self.W1(query_with_time) + self.W2(values)))

        # 2. □ □ □  □ □  (Softmax)
        attention_weights = tf.nn.softmax(score, axis=1)

        # 3. □ □  □ □  □ □  (Weighted Sum)
        context_vector = attention_weights * values
        context_vector = tf.reduce_sum(context_vector, axis=1)

        return context_vector, attention_weights
```

49.1.7 FAQ Pitfalls

Q. 어텐션을 쓰면 모델이 무거워지지 않나요?
A. 네, 인코더의 모든 타임 스텝을 매번 연산해야 하므로 $T_x \times T_y$의 연산량이 추가됩니다. 하지만 성능 향상 폭이 워낙 커서 감수할 만한 비용입니다.

Q. 어텐션은 번역 말고 어디에 쓰이나요?
A. 이미지 캡셔닝에서도 쓰입니다. 단어를 생성할 때마다 이미지의 특정 구역을 쳐다보는 식이죠.

☒ 다음 단계 (Next Step)

어텐션은 RNN의 한계를 부수고 성능을 극대화했습니다. 하지만 사람들은 깨달았습니다. "어텐션이 이렇게 강력하다면, 굳이 느린 순차 방식(RNN)을 써야 할까? 그냥 어텐션만 쓰면 안 되나?"

여기서 딥러닝 역사를 바꾼 논문, "Attention Is All You Need"가 등장합니다. 다음 시간, RNN과 CNN을 모두 버리고 오직 어텐션만으로 무장한 현대 AI의 심장, [Transformer Network]에 대해 알아보겠습니다.

[단원 요약 (Cheat Sheet)]

1. **Problem:** Seq2Seq 인코더의 고정 크기 벡터 병목 현상.
2. **Solution:** 디코더가 매 순간 인코더의 모든 부분을 연관성에 따라 참고한다.
3. **Formula:** $c^{\langle t \rangle} = \sum \alpha^{\langle t,t' \rangle} a^{\langle t' \rangle}$ (가중 평균).
4. **Benefit:** 긴 문장 번역 성능 비약적 상승 + 시각적 해석 가능.

50 Self-Attention Transformers

☒ 지난 시간 복습 및 연결

지난 시간에 배운 '어텐션'이 RNN이라는 엔진의 성능을 높여주는 보조 장치였다면, 오늘 배울 트랜스포머(Transformer)는 그 엔진 자체를 통째로 갈아치운 혁명입니다. 구글의 2017년 논문 "Attention Is All You Need"는 순차적 처리(RNN)를 버리고 오직 어텐션만으로 문장을 이해할 수 있음을 증명했습니다. 이것이 바로 ChatGPT와 모든 거대 언어 모델(LLM)의 탄생 지점입니다.

50.1 Overview

[핵심 목표]

이 단원은 현대 AI 시스템의 중추인 트랜스포머 아키텍처의 핵심 원리를 다룹니다.

- **Self-Attention:** 문장 내 단어들이 서로 어떤 관계를 맺고 있는지 스스로 학습하는 원리를 배웁니다.
- **Q, K, V:** 정보 검색의 관점에서 어텐션을 계산하는 세 가지 구성 요소(Query, Key, Value)를 파악합니다.
- **Multi-Head:** 여러 개의 어텐션을 병렬로 수행하여 다각적인 문맥을 추출하는 이유를 학습합니다.
- **병렬성:** RNN과 달리 문장 전체를 동시에 처리함으로써 얻는 계산 효율성을 이해합니다.

50.1.1 Essential Terminology: The QKV Framework

용어	의미	역할
Query (Q)	질문	"지금 내가 찾고 싶은 정보는 무엇인가?"
Key (K)	인덱스/색인	"내가 가진 정보의 키워드는 무엇인가?"
Value (V)	내용	"키워드에 해당하는 실제 값은 무엇인가?"
Scaled Dot-Product	유사도 계산	Q와 K를 곱해 점수를 내고 루트 차원으로 나눔.

50.1.2 Core Concepts: 자기 자신을 돌아보기

50.1.3 1. Self-Attention: 대명사 해결

RNN은 단어를 순서대로 읽었지만, 트랜스포머는 문장 안의 모든 단어 쌍 사이의 관계를 한 번에 계산합니다.

["it"은 누구인가?]

"The animal didn't cross the street because it was too tired."

Self-Attention은 "it"이라는 단어를 처리할 때 문장의 모든 단어를 훑습니다. 모델은 "animal"과의 어텐션 점수를 가장 높게 주어, "it = animal"임을 스스로 깨닫게 됩니다.

50.1.4 2. Multi-Head Attention

하나의 어텐션만으로는 복잡한 문맥을 다 담기 어렵습니다. 트랜스포머는 여러 개의 '헤드'를 병렬로 운영합니다.

- Head 1: 주어와 동사의 관계에 집중
- Head 2: 형용사와 명사의 관계에 집중
- Head 3: 대명사와 선행사의 관계에 집중

이 다양한 관점들을 나중에 하나로 합쳐(Concatenate) 풍부한 이해력을 갖게 됩니다.

50.1.5 Deep Dive: The Attention Math

트랜스포머의 심장인 Scaled Dot-Product Attention의 수식입니다.

Scaled Dot-Product Attention

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

- QK^T: Query와 Key 사이의 유사도 점수.
- $\sqrt{d_k}$: 차원이 커질 때 점수가 너무 커져 Softmax 기울기가 소실되는 것을 방지하는 Scaling 계수.
- V: 최종적으로 가중치를 곱해 정보를 취합할 실제 값.

50.1.6 Implementation: Conceptual Logic

TensorFlow를 이용한 어텐션의 핵심 로직입니다.

Listing 47: Scaled Dot-Product Attention

```python
import tensorflow as tf

def scaled_dot_product_attention(q, k, v, mask=None):
    # 1. □ □ □  Score □ □
    matmul_qk = tf.matmul(q, k, transpose_b=True)

    # 2. Scaling □ □ ( □ □ □ □  □ □ □ )
    dk = tf.cast(tf.shape(k)[-1], tf.float32)
    scaled_logits = matmul_qk / tf.math.sqrt(dk)

    # 3. □ Softmax □ □  □ □ □  □ □
    attention_weights = tf.nn.softmax(scaled_logits, axis=-1)

    # 4. □ Value □ □  □ □ □  □ □  □ □
    output = tf.matmul(attention_weights, v)

    return output, attention_weights
```

50.1.7 Modern Trends: Beyond Transformers

트랜스포머 이후 AI의 패러다임은 크게 확장되었습니다.

- **LLM (거대 언어 모델):** GPT, Llama 등 트랜스포머 블록을 수백 개 쌓아 인간 수준의 추론을 수행합니다.
- **ViT (Vision Transformer):** 이미지도 패치로 나누어 어텐션을 적용합니다. 이제 컴퓨터 비전에서도 트랜스포머가 대세입니다.
- **Efficiency:** 문장이 길어질 때 계산량이 N^2으로 폭발하는 문제를 해결하기 위한 FlashAttention 등이 활발히 연구됩니다.

☒ Course Conclusion

[최종 요약]

1. **Self-Attention:** 단어 간의 유기적 관계를 한 번에 파악하는 혁신적 방식.
2. **No More RNN:** 병렬 처리가 가능해져 모델의 거대화가 가능해짐.
3. **Versatility:** 텍스트를 넘어 이미지, 오디오 등 모든 도메인으로 확장 중.

축하합니다! Seq2Seq에서 시작해 어텐션을 거쳐, 세상을 바꾸고 있는 트랜스포머의 원리까지 모두 마스터하셨습니다. 여러분은 이제 현대 딥러닝의 정점에 서 있습니다.

51 BERT vs GPT

☒ 지난 시간 복습 및 연결

우리는 지난 시간에 Self-Attention이라는 트랜스포머의 핵심 엔진을 배웠습니다. 오늘은 이 엔진들을 조립해서 어떻게 BERT나 GPT 같은 거대 모델의 뼈대가 되는 트랜스포머 아키텍처(Transformer Architecture)가 완성되는지 해부해 보겠습니다. 트랜스포머는 크게 정보를 읽어 들이는 인코더(Encoder)와 정보를 생성하는 디코더(Decoder) 두 부분으로 나뉩니다.

51.1 Overview

[핵심 목표]

이 단원은 현대 AI 시스템의 설계도인 트랜스포머의 구조를 완벽히 이해합니다.

- **전체 구조:** 인코더와 디코더의 연결 메커니즘을 파악합니다.
- **순서 인식:** RNN 없이 순서를 파악하는 Positional Encoding을 배웁니다.
- **안정성:** 깊은 층을 쌓기 위한 Add & Norm (잔차 연결 및 정규화) 장치를 확인합니다.
- **철학의 차이:** 인코더 기반의 BERT와 디코더 기반의 GPT 차이를 이해합니다.

51.1.1 Core Components: 설계도 해부

51.1.2 1. Positional Encoding (위치 정보 주입)

트랜스포머는 단어를 한꺼번에 병렬로 입력받기 때문에, "I love you"와 "You love me"를 구분하지 못합니다.

위치 신호 주입 각 단어 벡터에 고유한 위치 값을 더해줍니다. 주기 함수인 Sine과 Cosine을 활용하여 고차원 공간에서 단어의 상대적/절대적 위치를 입힙니다.

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

51.1.3 2. Encoder Block (인코더: 이해의 영역)

인코더는 입력 문장을 통째로 보고 문맥을 파악합니다.

- **Multi-Head Self-Attention:** 단어 사이의 유기적 관계를 계산합니다.
- **Add & Norm:** 잔차 연결로 기울기 소실을 막고 Layer Norm으로 학습을 돕습니다.
- **BERT:** 이 인코더를 쌓아 만든 모델로, 문맥을 양방향(Bi-directional)으로 이해합니다.

51.1.4 3. Decoder Block (디코더: 생성의 영역)

디코더는 분석된 정보를 바탕으로 단어를 하나씩 생성합니다.

- **Masked Self-Attention:** 단어 생성 시 미래 단어를 미리 보고 정답을 유추하지 못하도록 마스킹을 적용합니다.
- **Encoder-Decoder Attention:** 단어를 뱉을 때마다 인코더가 분석한 원문 정보를 다시 참고합니다.
- **GPT:** 이 디코더를 쌓아 만든 모델로, 다음 단어 예측(Auto-regressive)에 특화되어 있습니다.

51.1.5 Deep Dive: BERT vs GPT

같은 트랜스포머 뿌리를 두지만, 부품 선택에 따라 성격이 완전히 달라집니다.

특징	BERT	GPT
기반 구조	트랜스포머 인코더	트랜스포머 디코더
학습 방향	양방향 (Bi-directional)	단방향 (Left-to-Right)
주요 목적	문장 분류, 질의응답	문장 생성, 대화, 창작
비유	빈칸 채우기 잘하는 모범생	이야기를 지어내는 소설가

51.1.6 Implementation: Hugging Face Transformers

현대 개발자들은 사전 학습된 가중치를 불러와서 미세 조정(Fine-tuning)하여 사용합니다.

Listing 48: Loading Pre-trained Transformers

```
from transformers import BertModel, GPT2Model

# BERT: □□ □□ □ □□ □□ □
bert = BertModel.from_pretrained('bert-base-uncased')

# GPT: □□□ □□ □ □□ □□ □
gpt = GPT2Model.from_pretrained('gpt2')
```

☒ Summary & Next Step

1. **Transformer:** RNN을 대체한 완벽한 병렬 처리 아키텍처.
2. **Positional Encoding:** 어텐션에 '순서'라는 생명력을 불어넣는 장치.
3. **Functional Split:** 이해를 원하면 인코더(BERT), 생성을 원하면 디코더(GPT).

이제 여러분은 현대 인공지능이 인간의 언어를 어떻게 처리하는지 그 밑바닥 설계도를 보셨습니다. 이 구조는 이제 언어를 넘어 이미지(ViT), 음성(Whisper) 등으로 무한히 확장되고 있습니다.

[생각해볼 거리]
마스킹(Masking)이 없다면 디코더는 왜 학습이 불가능할까요? 잔차 연결(Residual Connection)이 깊은 트랜스포머 층에서 왜 필수적일까요? 궁금한 점이 있다면 언제든 질문해 주세요!