ONECLICK AI

드롭아웃과 Yolo

연준모

목차

1. 옵티나이저와 드롭아웃 옵티나이저?

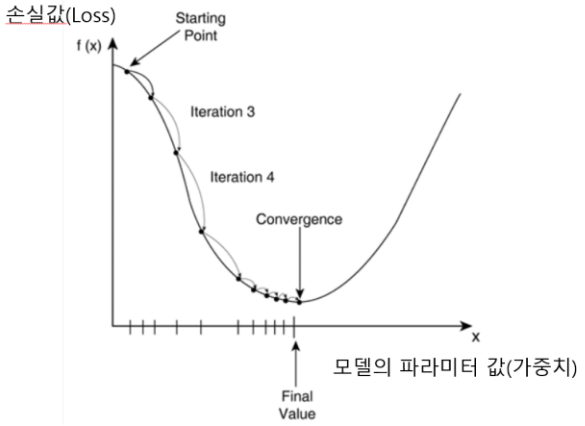
옵티나이저란?

손실함수 값을 최소화하는 최적의 파라미터를 찾는 알고리즘

일반적으로, 경사하강법을 기반으로 한다

4. 기본 개념 알고가기 경사하강법

ONECLICK AI



https://gihak111.github.io/ai/2025/09/16/Gradient_Descent_method_upload.html
https://arxiv.org/abs/1609.04747

오차를 최소화 하는 방향으로
가중치, 편향을 업데이트 한다

$$w_{new} = w_{old} - \eta \nabla J(w)$$

새로운 위치 = 현재 위치 + 움직인 방향과 거리

w_{new} : 업데이트 된 가중치. 다음 학습에서 사용된다.

w_{old} : 현재 가중치. 출발하는 값.

η : 학습률. 경사를 얼마나 많이 이동할 지 정한다.

$\nabla J(w)$: 현재 위치에서 기울기가 가장 가파른 방향.

여기에 사용되는 기울기를 역전파가 알려준다

1. 옵티나이저와 드롭아웃 확률적 경사 하강법 SGD?

Stochastic Gradient Descent SGD?
확률적 경사하강법

기본적인 경사 하강법은 전체 학습 데이터를 사용해서 한 번에 기울기를 계산

데이터가 매우 클 경우 계산량이 엄청나게 많아져서 느려 터진다.

=> 전체 데이터가 아닌, 무작위로 선택된 일부 데이터(미니배치 'mini batch')를 사용해 계산, 파라미터 업데이트 속도 증가, 노이즈가 있을 수 도 있는 데이터를 거름으로써 지역 최솟값에 빠질 위험을 줄여준다

$$W \leftarrow W - \eta \frac{\partial L_i(W)}{\partial W}$$

미니배치에 속하는 데이터 $x^{(i)}$와 정답 $y^{(i)}$에 대한 손실 함수를 $L_i(W)$ 라 할 때 식이 위와 같이 나온다

딥러닝은 전체 노드로, 손실 계산 + 가중치 업데이트만 미니 배치에 있는 노드만으로

1. 옵티나이저와 드롭아웃 모멘텀 Momentum?

Momentum?

'관성' 의 개념 도입

언덕 내려오는 공 점점 빨라지는 것 처럼

모멘텀은 이전의 기울기 방향을 현재 기울기 계산에 반영.

기울기가 같은 방향으로 계속 이동할 때 더 빠르게 학습, 기울기 방향이 자주 바뀌는 경우 진동을 줄여줌

모멘텀을 나타내는 변수 v와 모멘텀 계수 γ (보통 0.9와 같은 값을 사용)를 도입

1. 모멘텀 누적 v_t: 이전 스텝의 모멘텀γv_{t-1}과 현재 스텝의 기울기$\eta \nabla_W L(W)$를 합산

$$v_t = \gamma v_{t-1} + \eta \nabla_W L(W)$$

2. 가중치 업데이트: 누적된 모멘텀 값으로 가중치를 업데이트

$$W \leftarrow W - v_t$$

https://gihak111.github.io/ai/2025/10/14/momentum_upload.html

1. 옵티나이저와 드롭아웃 AdaGrad?

Adaptive Gradient? https://gihak111.github.io/ai/2025/10/14/Adagrad_upload.html

파라미터마다 서로 다른 학습률을 적용

언덕 내려오는 공 점점 빨라지는 것 처럼

학습 과정에서 변화가 적었던 파라미터는 더 큰 학습률로 업데이트하고,
변화가 많았던 파라미터는 작은 학습률로 업데이트

과거의 모든 기울기 제곱값을 합산하는 변수 G를 사용.

1. 기울기 제곱 누적 G_t: 현재 스텝의 기울기 제곱$\left(\nabla_W L(W)\right)^2$을 이전 값에 더한다

$$G_t \leftarrow G_{t-1} + \left(\nabla_W L(W)\right)^2$$

2. 가중치 업데이트: 학습률 η를 G_t의 제곱근으로 나누어 업데이트 강도를 조절. ϵ은 분모가 0이 되는 것을 방지하는 작은 값(e.g., 1e-8)

$$W \leftarrow W - \frac{\eta}{\sqrt{G_t + \epsilon}} \nabla_W L(W)$$

학습률이 급격히 감소하는 문제가 있다 해결 위해 RMSProp 가 있다

https://gihak111.github.io/ai/2025/10/16/RMSProp_upload.html

1. 옵티나이저와 드롭아웃 Adam?

Adaptive Moment Estimation?

모멘텀과 RMSProp의 장점을 결합

기울기의 지수 이동 평균(1차 모멘텀)과 기울기 제곱의 지수 이동 평균(2차 모멘텀)을 함께 사용하여 파라미터를 업데이트

모멘텀을 위한 1차 모멘텀 추정값 m_t와 RMSProp을 위한 2차 모멘텀 추정값 v_t를 사용하며,
각각의 감쇠율로 β_1, β_2를 사용

1. 1차 모멘텀 계산 m_t:

$$m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1)\nabla_W L(W)$$

2. 2차 모멘텀 계산 v_t:

$$v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2)\left(\nabla_W L(W)\right)^2$$

https://gihak111.github.io/ai/2025/10/14/Adam_upload.html

1. 옵티나이저와 드롭아웃 Adam?

3. 편향 보정 (Bias Correction): 학습 초반에 m_t와 v_t가 0에 가깝게 추정되는 것을 보정

$$\widehat{m_t} = \frac{m_t}{1-\beta_1^t}$$

$$\widehat{v_t} = \frac{v_t}{1-\beta_2^t}$$

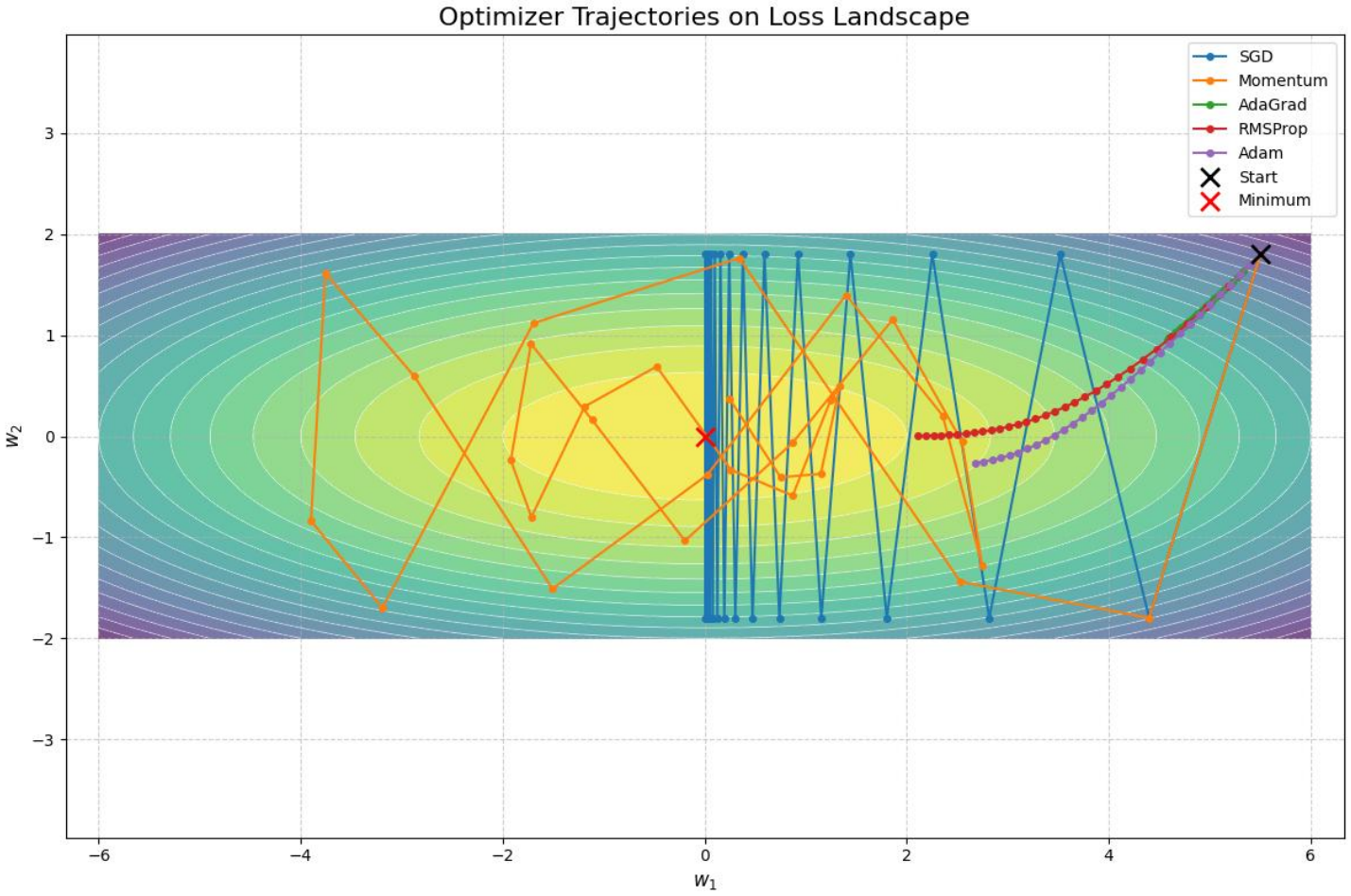
4. 가중치 업데이트: 보정된 모멘텀 값들을 사용하여 최종 업데이트를 수행

$$W \leftarrow W - \eta \frac{\widehat{m_t}}{\sqrt{\widehat{v_t}} + \epsilon}$$

Adam은 방향과 학습률을 모두 적응적으로 조절하여 대부분의 경우 좋은 성능을 보인다
그래서 요즘도 잘 쓰이는 그런거다 Adam, AdamW 이렇게 많이 쓰인다

1. 옵티나이저와 드롭아웃 비교?

ONECLICK AI



SGD와 Momentum은 발산해버림(너무 높은 학습율)

1. 옵티나이저와 드롭아웃 드롭아웃?

Dropout?

신경망의 과적합(overfitting)을 방지하기 위한 강력한 정규화(regularization) 기법

한 줄로 설명하자면, 훈련 중에 뉴런을 무작위로 끈다

신경망의 한 계층(layer)이 수행하는 연산은 활성화 함수까지 해서 이렇게 나온다

$$A^{(l)} = f\left(W^{(l)}\mathrm{x}^{(l-1)} + b^{(l)}\right)$$

입력 백터: $\mathrm{x}^{(l-1)}$

가중치 행렬: $W^{(1)}$

편향 벡터: $b^{(1)}$

활성화 함수: f

$W^{(l)}\mathrm{x}^{(l-1)}$ 이게 행렬곱셈인데,

이 연산이 바로 신경망을 통해 데이터의 특징이 변환되는 과정의 본질

하드마르곱 + 마스킹으로 구현

1. 옵티나이저와 드롭아웃 드롭아웃?

Dropout?

1. 마스크 벡터 생성

드롭아웃을 적용할 활성화 벡터 $x^{(l)}$와 동일한 차원의 마스크 벡터 $d^{(l)}$를 생성
이 벡터의 각 원소는 뉴런을 유지할 확률 p를 따르는 베르누이 분포에서 샘플링 된다
즉, 각 원소는 확률 p로 1이 되고, 확률 1-p로 0이 된다

베르누이 분포가 궁금하다면?

예시로, p=0.8이고 $x^{(l)}$가 4차원 벡터일 때, 마스크 $d^{(l)}$는 $\begin{pmatrix}1\\0\\1\\1\end{pmatrix}$ 와 같이 생성될 수 있다

살아남을 확률이 80%

2. 마스킹 적용

벡터 $x^{(l)}$에 마스크 벡터 $d^{(l)}$를 하드마르 곱⊙하여 드롭아웃이 적용된 새로운 활성화 벡터 $\widetilde{x^{(l)}}$를 만든다

$$\widetilde{a^{(l)}} = a^{(l)} \odot d^{(l)}$$

$$\begin{pmatrix}0.8\\0.2\\0.5\\0.9\end{pmatrix} \odot \begin{pmatrix}1\\0\\1\\1\end{pmatrix} = \begin{pmatrix}0.8\\0.0\\0.5\\0.9\end{pmatrix}$$

마스킹된 벡터 $\widetilde{x^{(l)}}$가 다음 계층의 입력으로 전달

일부러 죽은 노드를 만들어 낸다

1. 옵티나이저와 드롭아웃 드롭아웃?

Dropout?

? 과정 늘어났는데 그러면 시간 더 걸리고 느려지는거 아님?

위에서 설명한 하드마르 곱은 대각 행렬을 이용한 행렬 곱셈으로 완벽하게 치환할 수 있다.

$$d^{(l)} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots d_n \end{pmatrix} \quad \rightarrow \quad D^{(l)} = \begin{pmatrix} d_1 & 0 & \cdots & 0 \\ 0 & d_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & d_n \end{pmatrix}$$

이제 마스크 연산은 다음과 같이 표현된다

$$\mathrm{x}^{\widetilde{(l)}} = \mathrm{x}^{(l)} \odot d^{(l)} = D^{(l)}\mathrm{x}^{(l)}$$

$$\mathrm{x}^{(l+1)} = f\left(W^{(l+1)}\mathrm{x}^{\widetilde{(l)}} + b^{(l+1)}\right) = f\left(W^{(l+1)}\left(D^{(l)}\mathrm{x}^{(l)}\right) + b^{(l+1)}\right)$$

행렬곱셈은 결합법칙이 성립. 괄호의 위치를 바꿀 수 있다

$$\mathrm{x}^{(l+1)} = f\left(\left(W^{(l+1)}D^{(l)}\right)\mathrm{x}^{(l)} + b^{(l+1)}\right)$$

1. 옵티나이저와 드롭아웃 드롭아웃?

Dropout?

$W^{(l+1)}D^{(l)}$는 $W^{(l+1)}$의 열(column)들을 $D^{(l)}$의 대각 원소(0 또는 1)로 스케일링하는 것과 같다.

$D^{(l)}$의 대각 원소가 0인 위치에 해당하는 $W^{(l+1)}$의 열 벡터 전체가 0으로 바뀐다.

이는 드롭 아웃마냥 비활성 되는거다

결론적으로, 훈련 과정의 매 스텝마다 신경망은 무작위로 가중치 행렬의 일부 열들이 0으로 설정 더 얇은 부분 신경망을 학습하게 된다

? 근데 렐루는 0 되는게 문제라 하지 않았음?? 죽은 렐루라고??

근데 죽은 렐루는 컨트롤이 안됨. 드롭아웃은 컨트롤 된다.

그리고 죽은 렐루는 한번 죽으면 영원히 업데이트 없지만,

드롭아웃은 다음 스텝의 학습에선 학습될 가능성 있음

1. 옵티나이저와 드롭아웃 드롭아웃?

Dropout?

드롭아웃의 문제점

훈련 시 뉴런의 p 비율만 사용했기 때문에, 활성화 값의 총 기대값이 p배만큼 작아진다.
이를 보정하기 위해 추론(모델을 실사용) 시에는 모든 뉴런을 사용하되,

가중치 행렬 전체에 p를 곱해줘야$W_{\text{test}} = p \cdot W_{\text{train}}$ 훈련 때와 스케일을 맞출 수 있다

이러한 불편함을 없앤 것이 바로 **역전파 드롭아웃**

원하는 총량을 유지하기 위해 활상화 비율로 나누어 준다

1. 옵티나이저와 드롭아웃 역전파 드롭아웃?

Dropout? 예시를 통해 이해해 보자

입력신호 x : [1, 5, 2, 8]
가중치 w : [0.1 , 0.2 , 0.3 , 0.4]
출력 z : 0.1 + 1 + 0.6 + 3.2 = 4.9
여기서, 4.9가 목표 출력이다

1. 일반 드롭아웃인 경우

1. 마스크 생성 : [1, 0, 1, 0]
2. 입력에 마스크 적용 : [1, 0, 2, 0]
3. 출력 계산 : (1 * 0.1) + (0 * 0.2) + (2 * 0.3) + (0 * 0.4) = 0.1 + 0 + 0.6 + 0 = 0.7

최종적으로, 0.7이라는 너무 많이 약해진 출력의 신호를 기준으로 학습한다

1. 옵티나이저와 드롭아웃 역전파 드롭아웃?

Dropout? 예시를 통해 이해해 보자

1. 일반 드롭아웃인 경우 예측시

훈련시 신호가 50%로 약해졌기 때문에 이를 보정해주기 위해 가중치에 p인 0.5를 곱한다

1. 가중치 스케일링 : w_{test}= w * p = [0.1, 0.2, 0.3, 0.4] * 0.5 = [0.05, 0.1, 0.15, 0.2]

2. 최종출력 계산 : z_{test}= (1 * 0.05) + (5 * 0.1) + (2 * 0.15) + (8 * 0.2)

$$z_{test} = 0.05 + 0.5 + 0.3 + 1.6 = 2.45$$

이러면, 목표치의 정확히 절반의 값이 나온다 이렇게, 강제로 스케일을 맞춰 일관성을 유지한다

강제로 맞춰도, 절반으로 나온다

1. 옵티나이저와 드롭아웃 역전파 드롭아웃?

Dropout? 예시를 통해 이해해 보자

2. 역전파 드롭아웃인 경우

같은 뉴런이 꺼졌다고 가정해보자

1. 마스크 생성 : [1, 0, 1, 0]
2. 입력에 마스크 적용 : [1, 0, 2, 0]
3. 살아남은 신호를 p로 나누어서 증폭 : $x_{inverted} = x_{masked}$ / p = [1, 0, 2, 0] / 0.5 = [2, 0, 4, 0]
4. 출력 계산 : $z_{inverted_train}$ = (2 * 0.1) + (0 * 0.2) + (4 * 0.3) + (0 * 0.4) = 0.2 + 0 + 1.2 + 0 = 1.4

1.4는 4.9와는 너무 다른 값이다. 하지만, 다른 마스크를 적용하면 또 다른 값이 나오는 등,
평균적인 기댓값이 4.9에 맞춰지도록 훈련이 진행된다

1. 옵티나이저와 드롭아웃 역전파 드롭아웃?

Dropout? 예시를 통해 이해해 보자

2. 일반 드롭아웃인 경우 예측시

훈련 과정에서 이미 보정을 해 주었기 때문에, 예측시에는 그냥 하면 된다

$$z_{test} = (1 * 0.1) + (5 * 0.2) + (2 * 0.3) + (8 * 0.4) = 4.9$$

이렇게 해서, 목표한 값 4.9가 그래도 출력되게 한다

1. 옵티나이저와 드롭아웃 비교

일반 드롭아웃과 역전파 드롭아웃 비교

구분	일반 드롭아웃	**역전파 드롭아웃 (Inverted Dropout)**
핵심 동작	훈련 시 신호를 약하게 하고, **예측 시 가중치를 보정**	훈련 시 약해진 신호를 **미리 증폭**
훈련 출력 (예시)	`0.7` (약해진 신호)	`1.4` (보정된 신호)
예측 출력 (예시)	`2.45` (보정된 결과)	`4.9` (원래 결과)
장점	개념이 단순함	**훈련/예측 모델이 동일하여 편리함 (현재 표준)**

2. Yolo 아키텍쳐 알아보기 어떻게 생겼을까?

Yolo와 함께 볼 것들?

실시간 객체 검출을 위한 모델
많은 버전이 있으며, 그 중에도 가장 최신인 Yolo v8을 기준으로 알아보자

이미지 내 객체의 위치와 종류를 한 번에 예측하는 단일 단계 검출기

분류와 지역화를 매우 빠른 속도로 한 번에 처리한다

막 로봇개가 사람 쫓아가고 하는거 그걸 이 yolo 모델로 만드는 거다

2. Yolo 아키텍쳐 알아보기 어떻게 생겼을까?

Yolo와 함께 볼 것들?

1. Conv는 CNN때 많이 해서 잘 알고 있으리라 믿는다

2. 배치 정규화
컨볼루션 연산을 거치며 숫자들의 분포가 한쪽으로 치우치거나 너무 커지는 것을 막아준다
각 레이어의 작업 결과를 "표준 규격"에 맞게 보정하여 다음 레이어가 작업을 더 쉽고 안정적으로 할 수 있게 돕는 역할이다

2. Yolo 아키텍쳐 알아보기 어떻게 생겼을까?

Yolo와 함께 볼 것들?

3. 활성화 함수 SiLU

컨볼루션이 찾아낸 패턴이 얼마나 중요한지를 판단하는 스위치 역할

탐지된 패턴이 중요하고 의미 있다면 스위치를 활짝 열어(강한 신호 전달),

별 의미 없다면 스위치를 닫아(약한 신호 전달) 불필요한
정보를 걸러낸다

$$SiLU(x) = x \cdot \sigma(x)$$

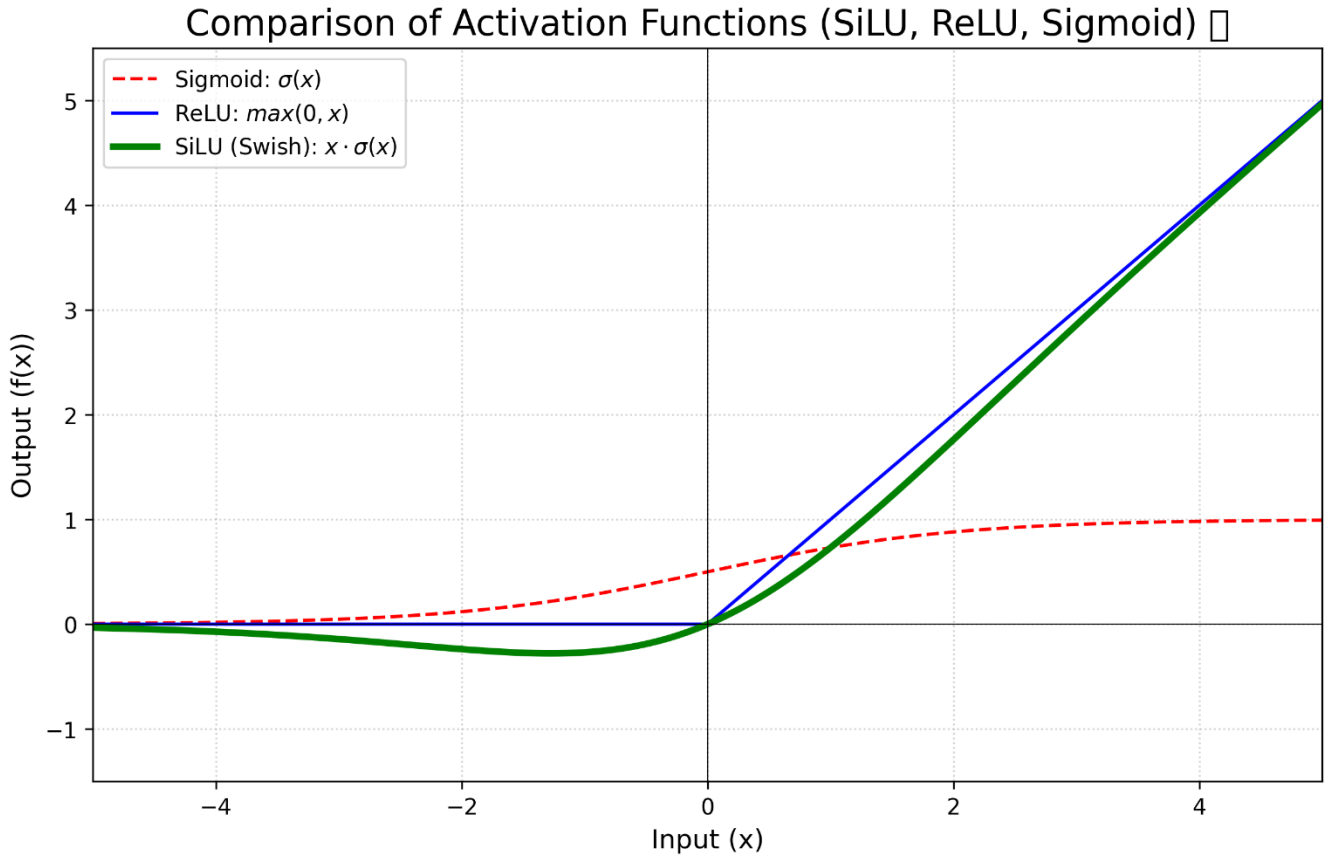
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

시그모이드를 통해서 입력값 x 를 0 ~ 1 사이로 정규화 (비선형성 부여)
값의 크기를 통해서 얼마나 값을 통과시킬지 스케일 값 역할을 하게 된다
입력값이 매우 크면 시그모이드의 출력이 커 연산되었을 때 값이 크게 나오고(문이 많이 열림)
입력값이 작으면 시그모이드의 연산값이 작게 된다(문이 덜 열림)

입력 백터를 그대로 유지 (선형성)

2. Yolo 아키텍쳐 알아보기 어떻게 생겼을까?

Yolo와 함께 볼 것들?



? 렐루랑 생긴거 비슷한데 이거 왜 쓰는거임?

동적 연산:
단순히 0이냐 그냥 출력이냐를 비교하는 렐루와 다르게 동적 연산을 진행

죽은 노드 값 완화:
ReLU는 0이하 값이 들어오면 역전파 때 죽어버리는 문제가 있지만, SiLU는 0에 수렴하지 0이 아니기 때문에 그럴일이 없다

2. Yolo 아키텍쳐 알아보기 어떻게 생겼을까?

Yolo?

다음과 같은 3개의 구조를 가진다

1. 백본 : 픽셀을 의미로 바꾸는 특징 추출기
간단하게 표현하면, CNN이다

2. 넥 : 흩어진 정보를 종합하는 융합기
여러 정보를 전부 종합한다

3. 헤드 : 최종 예측기
여러 정보를 전부 종합한다

2. Yolo 아키텍쳐 알아보기 어떻게 생겼을까?

Yolo?

1. 백본(Backbone)

CNN에서 했던, 특징 추출 과정을 그대로 따라간다
여기서는, C2f 모듈을 수십번 반복해서 이미지의 특징을 점진적으로 추출해 낸다

C2f 모듈 구성

1. Conv

2. 배치 정규화

3. 활성화 함수 SiLU

2. Yolo 아키텍쳐 알아보기 어떻게 생겼을까?

Yolo?

2. 넥(Neck)

백본을 통과하면서 이런 정보가 만들어진다
1. 깊은 층의 정보 : 해상도는 낮지만, 이건 사람이다 같은 의미 정보가 많이 있다
2. 얕은 층의 정보 : 해상도는 높지만, 여기에 모서리가 있다 같은 위치 정보만 있다

넥은 이 두 종류의 정보를 효과적으로 융합, 이 위치에는 사람이 있다 라는 종합적 정보를 만들어 낸다

여기서는 PANet을 사용한다
1. 하향식 경로 : 먼저 깊은 층의 의미 정보를 얕은 층으로 전달한다.
   사람을 찾아봐 같은 지시를 내리는 것과 같다
2. 상향식 경로 : 그 후, 지시를 받은 얕은 층이 가진 정확한 위치 정보를 다시 깊은 층으로 전달.
   사람은 이 픽셀 주변에 있다고 알려준다

이렇게 양방향 정보 교환을 통해서 넥은 크고 작은 모든 객체를 정확히 탐지한다

2. Yolo 아키텍쳐 알아보기 어떻게 생겼을까?

Yolo?

3. 헤드(Head)

넥에서 완벽하게 만들어진2가지의 특징을 맵을 박아서 우리가 원하는 형태의 출력을 한다(분류)

2 가지의 특징을 가지는데,

1. 앵커 프리

과거는 미리 여러 크기의 예상 사각형(앵커)를 정해두고,

3번 예상 사각형을 약간 늘리면 객체에 맞겠순 이라고 예측하였다

Yolov8은 예상 사각형 없이 객체의 중심은 여기고,
너비는 이만큼, 높이는 이만큼이라고 위치와 크기를 직접 예측한다. 훨신 직관적이고 효율적이다

2. Yolo 아키텍쳐 알아보기 어떻게 생겼을까?

Yolo?

3. 헤드(Head)

2. 분리형 헤드

헤드 안에는 두 개의 역할이 있다

1. 위치 역할 : 오직 객체의 위치(사각형 좌표)를 예측하는 것에만 특화되어 있다
2. 종류 역할 : 오직 객체의 종류를 예측하는 데만 특화되어 있다

이렇게, 역할을 분리하면 더 잘 예측해서 위치와 종류 모두에 대한 예측 정확도가 크게 향상된다

2. Yolo 아키텍쳐 알아보기 어떻게 생겼을까?

다음과 같은 아키텍쳐라 하고 해 보자

욜로모델 자체가 반복적으로 돌아가야 하는 부분이 엄청나게 많기 때문에,
숫자 대입해서 진행하는 예제가 쉽지 않다.

따라서, 대략적인 크기를 보며 어떤 흐름으로 작동되는지만 봐 보자.

+ c2f에서 Batch Norm, SiLU는 빼고 진행하겠다.

https://gihak111.github.io/ai/2025/10/16/AI_Architecture_Yolo_upload.html

3. 수식으로 Yolo 열어보기 1. 순전파

순전파 1. 백본

입력 이미지 X_{input}은 (640, 640, 3) 크기의 행렬이라고 하자

백본의 입력인 Conv의 첫 번째 레이어는 Conv 레이어 이다.

1. Stem Layer (Conv): 첫 번째 컨볼루션 레이어는 큰 보폭(Stride=2)으로 연산을 수행한다

   입력: X_{input} (640, 640, 3)

   연산: $Y = \mathrm{Conv}(X, W, b)$

   출력: 크기는 절반으로 줄고 채널은 증가한 특징 맵 F_0 (320, 320, 64)가 생성된다.

   이 F_0가 다음 레이어의 입력으로 그대로 전달된다.

순전파 1. 백본

2. C2f 모듈 및 다운샘플링: C2f 모듈은 특징을 풍부하게 만들고,
중간의 다운샘플링 컨볼루션은 특징 맵의 크기를 계속 줄여나간다.

과정 1: F_0 (320, 320, 64)가 C2f와 다운샘플링 Conv를 거쳐 F_1 (80, 80, 256) 으로 변환된다.
이 F_1은 나중에 넥(Neck)에서 재사용하기 위해 저장된다.

과정 2: F_1이 다음 C2f와 다운샘플링 Conv를 거쳐 F_2 (40, 40, 512) 로 변환된다.
이 F_2 또한 저장된다.

과정 3: F_2가 마지막 C2f와 다운샘플링 Conv를 거쳐 F_3 (20, 20, 1024) 로 변환된다.

순전파 2. 넥

넥은 백본에서 저장해 둔 다양한 크기의 특징 맵(F_1, F_2, F_3)을 받아 정보를 융합

하향식 경로 (Top-down): 깊은 층의 의미 정보
("이것은 자동차 형태이다")를 얕은 층으로 전달하여 특징을 풍부하게 만든다.

상향식 경로 (Bottom-up): 의미 정보가 보강된 얕은 층의 정확한 위치 정보
("자동차는 이 픽셀 주변에 있다")를 다시 깊은 층으로 전달해 예측 정확도를 높인다.

3. 수식으로 Yolo 열어보기 1. 순전파

순전파 2. 넥

1. 하향식 경로 (Top-down Path)

1. 백본의 가장 깊은 특징 맵 F_3 (20x20)를 업샘플링(Upsample)하여 해상도를 두 배로 높인 F_{3_up} (40x40)을 만든다.
2. 이 F_{3_up}을 백본의 중간 특징 맵 F_2 (40x40)와 채널 축을 기준으로 결합(Concatenate)한다. 두 정보가 합쳐져 채널의 깊이가 매우 깊어진다.

$$F_{fused1} = \text{Concat}\left(\left[F_{3\_up}, F_2\right]\right) \rightarrow (40,40,1536)$$

3. 깊고 복잡해진 특징 맵 F_{fused1}을 C2f 모듈로 처리하여 정보를 효과적으로 압축하고 정제한다. 이를 통해 중간 특징 맵 N_1 (40x40)을 생성한다.
4. 이 과정을 한 번 더 반복(N_1 을 업샘플링하여 F_1 과 결합)하여 더 높은 해상도의 중간 특징 맵 N_2 (80x80)를 생성한다.

순전파 2. 넥

2. 상향식 경로 (Bottom-up Path)

이제 하향식 경로에서 만들어진 특징 맵들을 활용하여 반대 방향으로 정보를 전달한다.

1. 가장 해상도가 높은 N_2 (80x80)를 다운샘플링(Downsample)하여 해상도를 절반으로 줄인 N_{2_down} (40x40)을 만든다.
2. N_{2_down} 을 하향식 경로의 중간 특징 맵 N_1 (40x40)과 결합한다.

$$F_{fused2} = \text{Concat}([N_{2\_down}, N_1])$$

3. F_{fused2} 를 다시 C2f 모듈로 처리하여, 최종적으로 헤드(Head)로 전달될 3개의 특징 맵 중 하나인 P_{medium} (40x40)을 생성한다.
4. 이 과정을 반복하여 최종적으로 P_{small} (80x80), P_{medium} (40x40), P_{large} (20x20) 세 가지 크기의 특징 맵을 완성하며, 이것들이 헤드로 전달될 최종 입력 데이터가 된다.

순전파 3. 헤드

헤드는 넥에서 완성된 3개의 특징 맵($P_{small}, P_{medium}, P_{large}$) 각각에 대해 독립적으로 예측을 수행한다.
예시로 앞서 계산한 값 중 P_{medium} (40x40) 특징 맵을 기준으로 하자.

분리형 헤드 (Decoupled Head): 예측의 정확도를 높이기 위해, 하나의 네트워크가 아닌 위치 예측과 종류 예측을 위한 두 개의 분리된 경로로 특징 맵을 전달한다.

위치 예측 경로: 바운딩 박스(x, y, w, h) 예측에 특화된 Conv 레이어들을 통과한다.

$$Y_{bbox\_feat} = \mathrm{Conv}_{bbox}(P_{medium})$$

종류 예측 경로: 객체의 종류(Class) 예측에 특화된 Conv 레이어들을 통과한다.

$$Y_{cls\_feat} = \mathrm{Conv}_{cls}(P_{medium})$$

최종 출력: 각 경로는 (40x40) 크기의 최종 특징 맵을 출력한다.
이 맵의 각 격자 셀(grid cell)은 해당 위치에서의 예측값을 담고 있다.
예를 들어, (i, j) 위치의 셀은 바운딩 박스 정보(4개 값)와 클래스 확률(80개 값)을 출력한다.
이는 각 셀의 특징 벡터에 대해 선형 변환을 수행하는 것과 같다.

순전파 3. 헤드

최종 출력: 각 경로는 (40x40) 크기의 최종 특징 맵을 출력한다.
이 맵의 각 격자 셀(grid cell)은 해당 위치에서의 예측값을 담고 있다.
예를 들어, (i, j) 위치의 셀은 바운딩 박스 정보(4개 값)와 클래스 확률(80개 값)을 출력한다.
이는 각 셀의 특징 벡터에 대해 선형 변환을 수행하는 것과 같다.

$$Y_{pred} = W \cdot X_{cell\_feat} + b$$

손실값은 손실함수 활용해서 구하면 된다. Yolov8에서는 특수한 방법으로 계산한다

https://gihak111.github.io/ai/2025/10/16/Yolo_Loss_Function_upload.html

3. 수식으로 Yolo 열어보기 2. 역전파

역전파 1. 손실값

손실 함수 L을 통해 최종 예측값 Y_{pred}와 정답 Y_{true}간의 오차를 계산한다.

이 손실을 최종 예측값으로 미분한 $\frac{\partial L}{\partial Y_{pred}}$ 가 역전파의 시작점이 되는 최초의 오차 신호이다.

역전파 2. 헤드

헤드의 최종 연산은 선형 변환 $Y = WX + b$와 같다. 이 식을 기반으로 역전파를 수행한다.

1. 가중치(W)에 대한 기울기 계산

   목표: 손실에 대한 가중치의 영향력, 즉 $\frac{\partial L}{\partial W}$ 를 구한다.

   연쇄 법칙: $\frac{\partial L}{\partial W} = \frac{\partial L}{\partial Y} \cdot \frac{\partial Y}{\partial W}$

   미분: 순전파 식 $Y = WX + b$를 W에 대해 미분하면 $\frac{\partial Y}{\partial W} = X^T$ 이다.

   결과: $\frac{\partial L}{\partial W} = \frac{\partial L}{\partial Y} \cdot X^T$

   가중치의 기울기는 '출력 오차'에 그 출력을 만드는 데 사용된 '입력 데이터'를 곱한 값이다.

   이 기울기는 가중치 업데이트에 직접 사용된다.

역전파 2. 헤드

2. 이전 레이어 입력(X)으로 오차 전파:

목표: 이전 레이어(넥)로 전달할 오차 신호, 즉 $\frac{\partial L}{\partial X}$ 를 구한다.

연쇄 법칙: $\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} \cdot \frac{\partial Y}{\partial X}$

미분: 순전파 식 $Y = WX + b$를 X에 대해 미분하면 $\frac{\partial Y}{\partial X} = W^T$ 이다.

결과: $\frac{\partial L}{\partial X} = W^T \cdot \frac{\partial L}{\partial Y}$

이전 레이어로 전달되는 오차는 '출력 오차'에 '가중치 행렬의 전치'를 곱한 값이다.

이는 각 가중치가 출력 오차에 기여한 만큼을 역으로 계산하여 입력 오차를 재구성하는 과정이다.

역전파 3. 넥(Neck)과 백본(Backbone)에서의 역전파

$\frac{\partial L}{\partial X}$ 가 거슬러 올라간다

Concatenate의 역전파:

순전파 때 채널 축으로 합쳐졌던 두 특징 맵 A, B가 있었다면,
역전파 시에는 들어온 기울기 텐서를 합쳐지기 전의 채널 크기에 맞게 그대로 분할(Slice)하여 각각의 경로로 전달한다

Convolution의 역전파:

헤드와 마찬가지로, 들어온 출력 오차를 이용해 가중치에 대한 기울기($\frac{\partial L}{\partial W}$)와
입력에 대한 기울기($\frac{\partial L}{\partial X}$)를 계산하여 각각 가중치 업데이트와
이전 레이어로의 오차 전파에 사용한다.

역전파 3. 넥(Neck)과 백본(Backbone)에서의 역전파

$\frac{\partial L}{\partial X}$ 가 거슬러 올라간다

활성화 함수(SiLU)의 역전파: 생략했지만, 집고 넘어가자

활성화 함수를 $f(x)$라 하면, 출력 오차에 활성화 함수의 도함수(미분값) $f'(x)$을 곱하여 오차를 전달한다.

이는 특정 뉴런이 활성화된 정도에 따라 오차를 조절하여 전달하는 역할을 한다.

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial f(x)} \cdot f'(x)$$

가중치 업데이트

모든 레이어에서 계산된 가중치 기울기($\frac{\partial L}{\partial W}$)를 사용하여

경사 하강법(Gradient Descent)으로 모든 가중치를 업데이트한다. (η는 학습률)

$$W_{new} = W_{old} - \eta \cdot \frac{\partial L}{\partial W}$$

이 순전파와 역전파 과정을 수없이 반복하며,

모델은 오차를 줄이는 방향으로 점차적으로 개선되어
더 정확한 예측을 학습하게 된다.

가중치 업데이트

모든 레이어에서 계산된 가중치 기울기($\frac{\partial L}{\partial W}$)를 사용하여

경사 하강법(Gradient Descent)으로 모든 가중치를 업데이트한다. (η는 학습률)

$$W_{new} = W_{old} - \eta \cdot \frac{\partial L}{\partial W}$$

이 순전파와 역전파 과정을 수없이 반복하며,

모델은 오차를 줄이는 방향으로 점차적으로 개선되어
더 정확한 예측을 학습하게 된다.

4. Numpy 코드

https://huggingface.co/gihakkk/Yolo_test

감사합니다