

Overfitting,Dropout,Weight Decay,Batch Normalization

practice 인공지능,머신러닝 · 2024. 10. 16. 06:49

Summary

Dropout : 학습중에 일부 가중치를 확률적으로 0으로 만드는것

Dropout 쓰는이유: overfitting 막는데 도움

Overfitting: 학습데이터를 모델이 거의 외워버린 상태 => 새로운 테스트 데이터 대해 모델이 성능 낮음

Overfitting 막는 방법: 데이터 늘리기, Early stopping(validation data), Weight Decay(가중치 감소), Dropout, Batch Normalization

아래그림) dropout

인공지능은 여러가지 입력값 (feature 들)을 받아서 예측을 한다. 이때, 모델이 특정한 입력값에 과하게 의존하게 되면, 이후 예측에 문제가 생길 수 있다. 예를 들어서, 오렌지와 레몬을 분류하는 인공지능을 만들었다고 하자. 사진이 주어졌을 때, 둘중 하나로 예측하는 것이다.

인공지능은 여러가지 입력값 (feature 들)을 받아서 예측을 한다. 이때, 모델이 특정한 입력값에 과하게 의존하게 되면, 이후 예측에 문제가 생길 수 있다.

드롭아웃(Dropout)은 인공신경망(Artificial Neural Networks)에서 과적합(overfitting)을 방지하고 모델의 일반화 성능을 향상시키기 위해 사용되는 정규화 기법입니다. 드롭아웃은 훈련 과정에서 무작위로 뉴런(neuron)을 비활성화(즉, 값을 0으로 설정)하는 방법을 사용합니다. 이렇게 하면 모델이 일부 뉴런에 지나치게 의존하는 것을 방지하고, 다양한 뉴런 조합을 통해 학습하게 되어 더욱 강건한 모델이 됩니다.

드롭아웃의 주요 개념:

- 과적합 방지:** 모델이 훈련 데이터에 과도하게 맞춰지면 새로운 데이터에 대해 일반화 능력이 떨어질 수 있습니다. 드롭아웃은 뉴런 일부를 무작위로 꺼서 각 뉴런이 모든 입력에 대해 독립적으로 작동하도록 유도하여 과적합을 방지합니다.
- 훈련 시 적용:** 드롭아웃은 훈련 시에만 적용되며, 테스트나 추론 단계에서는 모든 뉴런이 활성화된 상태로 작동합니다. 이를 통해 테스트 시에는 완전한 모델을 사용하여 추론 성능을 극대화합니다.
- 드롭아웃 비율:** 드롭아웃 비율은 보통 0.5로 설정되며, 이는 각 학습 단계에서 절반의 뉴런을 무작위로 비활성화한다는 의미입니다. 하지만 특정 문제에 따라 비율을 조정할 수 있습니다.
- 앙상블 효과:** 드롭아웃은 여러 모델을 앙상블(ensemble)하는 것과 유사한 효과를냅니다. 모델이 훈련될 때 매번 다른 조합의 뉴런을 사용하게 되므로, 여러 모델이 동시에 학습되는 것처럼 작동합니다. 이는 모델의 일반화 성능을 높이는 데 기여합니다.

특정 뉴런에 과도하게 의존하게 되는 데이터의 예시로, 매우 단순하고 패턴이 명확한 데이터셋을 생각해 볼 수 있습니다. 특히, 이미지 분류 문제에서 특정한 패턴에 지나치게 의존하는 경우가 자주 발생합니다.

예시: 고양이와 강아지 이미지 분류

고양이와 강아지를 분류하는 신경망을 훈련시킨다고 가정해 봅시다. 훈련 데이터셋에서 고양이 이미지는 대부분 배경에 나무가 있고, 강아지 이미지는 대부분 배경에 실내 환경(예: 카펫)이 있는 경우가 있다고 하겠습니다.

이 경우, 신경망은 실제로 고양이나 강아지 자체의 특성이 아니라 **배경 정보**에 지나치게 의존하여 학습할 수 있습니다. 신경망의 특정 뉴런들은 고양이를 분류할 때 나무 배경에 주로 반응하고, 강아지를 분류할 때 실내 배경에 주로 반응하게 됩니다. 결국, 이 뉴런들은 고양이와 강아지를 구분하는 데 중요한 **주요 특징**보다는 배경 정보에 과도하게 의존하게 됩니다.

이로 인한 문제

이 모델이 테스트 데이터셋에서 **배경이 다른** 고양이와 강아지 이미지를 만나면, 모델이 제대로 분류하지 못할 가능성이 커집니다. 예를 들어, 고양이가 실내에서 찍힌 이미지를 입력하면, 모델은 배경에 익숙한 강아지 이미지라고 잘못 예측할 수 있습니다.

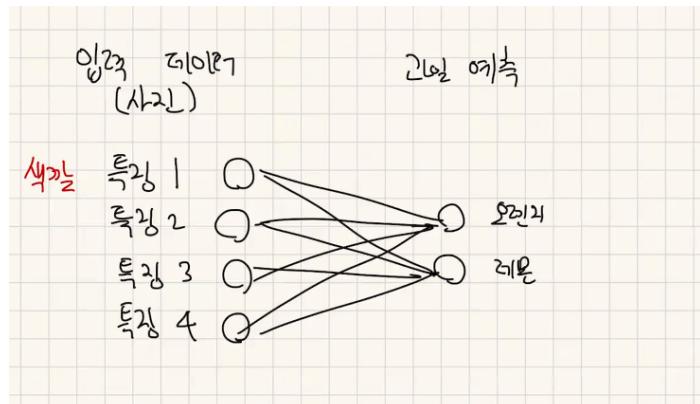
해결 방법으로서의 드롭아웃

이러한 과도한 의존을 방지하기 위해 드롭아웃을 적용하면, 훈련 중 일부 뉴런들이 무작위로 비활성화되므로, 특정 뉴런(예: 배경을 인식하는 뉴런)만으로 의존하지 않고 **다양한 뉴런이 고루 학습**하게 됩니다. 이는 모델이 고양이와 강아지를 분류할 때 배경이 아닌 중요한 시각적 특징(예: 귀 모양, 털 패턴 등)을 학습하도록 유도합니다.

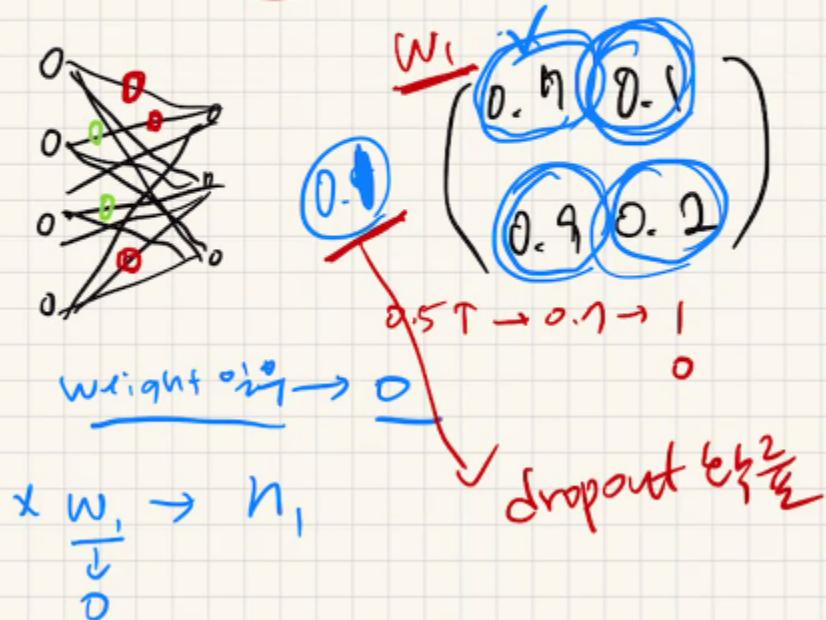
실제 상황 예시:

- 자율주행:** 자율주행차가 도로의 환경을 인식하는 모델이 있다면, 모델이 신호등이나 보행자 대신 특정 도로 배경(예: 도로 옆의 나무나 건물)에 지나치게 의존하는 경우가 있을 수 있습니다. 이 경우 배경 정보만 바뀌어도 모델의 성능이 급격히 떨어질 수 있습니다.
- 의료 영상:** 질병 진단을 위한 의료 영상 데이터에서, 특정 기계의 레이블이나 촬영 조건(조명, 배경)을 모델이 학습한다면, 실제 병변보다는 주변의 불필요한 정보에 의존할 가능성이 있습니다. 이는 진단의 정확도에 문제를 일으킬 수 있습니다.

드롭아웃은 이러한 특정 패턴에 과도하게 의존하지 않고, 중요한 특징을 골고루 학습하게 해주는 방법입니다.



Dropout : 학습 중에 일부 가중치는
작동하지 않음으로 만드는 것

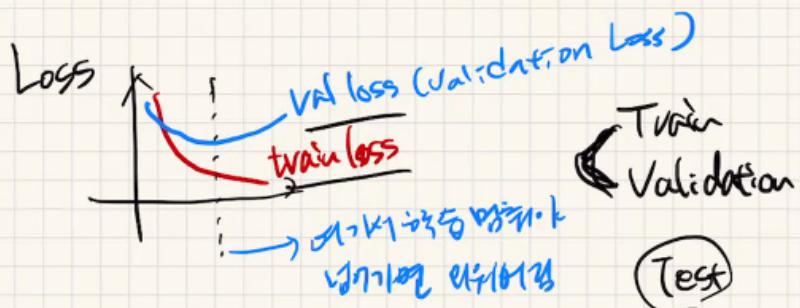
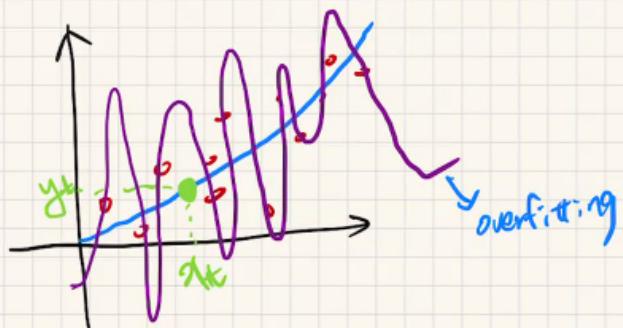


아래그림)

Overfitting: 학습데이터를 모델이 거의 외워버린 상태(불필요한 내용까지도) => 새로운 테스트 데이터 대해 모델이 성능 낮음

Overfitting 된 모델은 train(학습) 데이터에서는 매우 좋은 성과를 보이지만 (데이터들과 예측선 사이의 거리 0), 실제 새로운 (test, validation) 데이터(예측)에서는 빨간색 선 (선형회귀 모델) 보다 쓸모가 없다. 이는 Overfitting 된 모델이 train 데이터에서 불필요한 오차 등을 모두 학습 했기 때문이다.

Overfitting : 학습 데이터는 성능이
기존 예상 범위를 벗어나는 경우
 ⇒ 시험 테스트 성능이 떨어짐
 성능이 좋지 않음

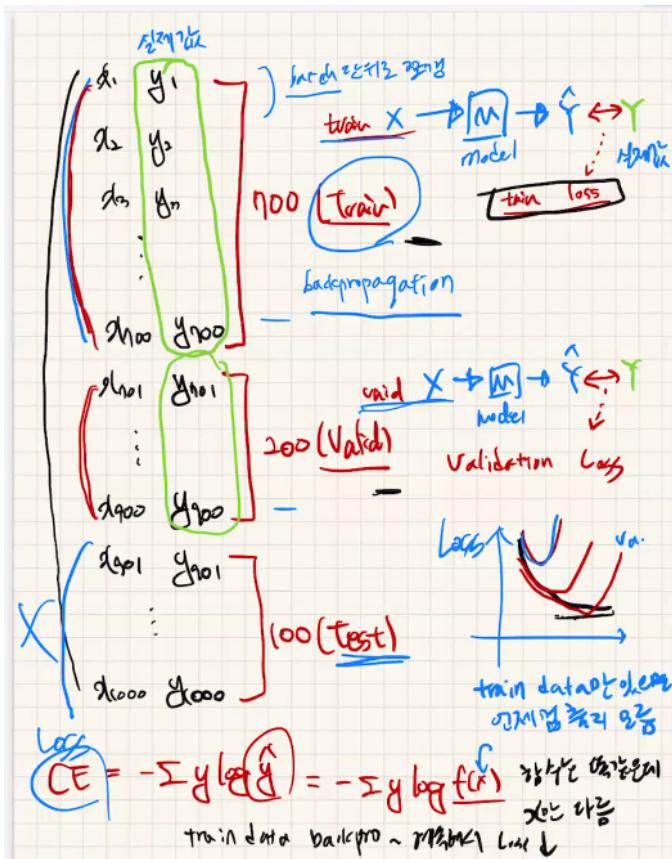


아래그림)

train data, validation data, test data

train data만 있으면 언제 멈출지 모르기 때문에 validation loss 함수가 꺾이는 지점에서 모델을
선정

test data는 최종적으로 성능평가.



아래그림)

데이터 늘리기 - overfitting 방지

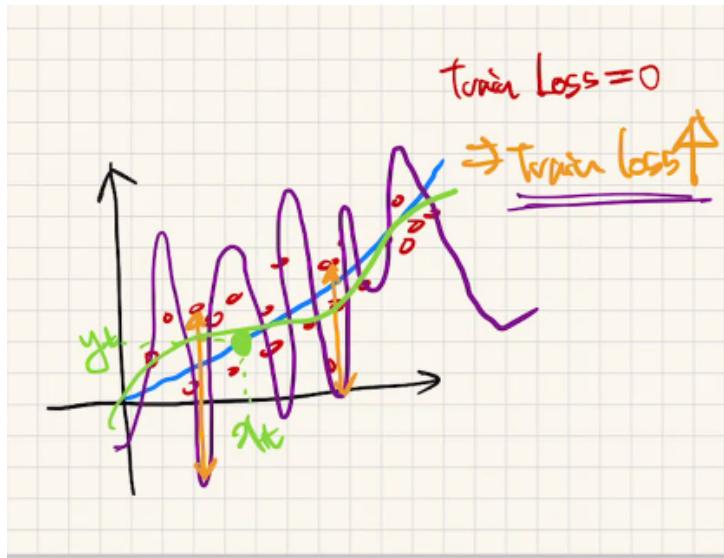
기존 overfitting 일때는 loss가 0인 보라색 그래프인데,

데이터를 늘리고 학습을 더 진행하면, train loss가 증가함

그러면 train loss 가 감소되는 방향으로 학습이 일어나서 초록색 그래프처럼 loss가 줄어드는 방향으로 그래프가 변해짐

진짜 함수인 파란색 함수랑 비슷해짐

(ex. 수학문제를 외우면 새로운 문제 못풀)



이미지에서 보여주는 그래프는 **과적합(overfitting)**의 개념을 설명하는 것 같고, 제가 설명한 **데이터 증강(data augmentation)**은 과적합을 방지하는 방법 중 하나입니다. 그림과 데이터 증강을 통한 과적합 방지의 관계를 좀 더 명확하게 설명해드리겠습니다.

1. 과적합이란?

- **과적합(overfitting)**은 모델이 훈련 데이터에 너무 집중하여, 새로운 데이터(테스트 데이터)에 대해서는 좋은 성능을 발휘하지 못하는 문제를 의미합니다. 훈련 데이터의 세부적인 패턴과 노이즈를 지나치게 학습하는 것이 원인입니다.
- 그림에서 **보라색 그래프**는 복잡한 모델이 훈련 데이터의 모든 작은 변동성까지 맞추려는 모습을 보여줍니다. 이 경우 **Train Loss(훈련 손실)**은 0에 가까워지지만, 테스트 데이터에서는 오히려 성능이 떨어질 수 있습니다.

2. 데이터 증강의 역할

- **데이터 증강(data augmentation)**은 인위적으로 데이터를 늘려 훈련 데이터를 다양화하는 방법입니다. 이미지나 텍스트 데이터를 약간 변형해 모델이 더 많은 패턴을 학습하도록 도와줍니다.
 - 예를 들어, 이미지 회전, 확대, 색상 변화 등으로 새로운 이미지를 만들어 데이터셋을 확장할 수 있습니다.
- 데이터 증강을 통해 훈련 데이터의 양과 다양성이 증가하면, 모델이 특정 데이터에만 과도하게 의존하는 경향을 줄일 수 있습니다. 즉, 모델이 다양한 변형된 데이터를 학습하게 되므로 **일반화 성능**이 향상됩니다.

3. 그림과 데이터 증강의 연관성

- 그림에서 보여지는 **보라색** 곡선은 과적합된 모델을 나타냅니다. 이 모델은 훈련 데이터의 세부적인 노이즈까지 학습하여, 새로운 데이터에 대해 좋은 성능을 발휘하지 못할 가능성이 큽니다.
- 초록색** 곡선은 데이터를 더 일반화하여 학습한 결과로, 모델이 더 단순한 패턴을 학습하고 과적합이 줄어든 모습을 보여줍니다.
- 데이터를 증강하면 모델이 다양한 패턴을 학습하게 되어, **초록색 그래프**처럼 더 일반화된 모델을 만들 수 있습니다. 데이터 증강은 모델이 훈련 데이터에 과도하게 의존하지 않도록 하여 과적합을 방지하는 데 효과적입니다.

요약

- 그림**은 훈련 데이터에 너무 맞춰진 모델(과적합된 모델)이 어떻게 손실을 줄이는지 보여주고, 일반화 성능이 부족한 상태에서 발생할 수 있는 문제를 설명합니다.
- 데이터 증강**은 데이터를 다양화하여 모델이 더 많은 변형된 데이터를 학습하게 만들고, 그 결과 과적합을 방지하여 새로운 데이터에 대한 예측 성능을 향상시키는 방법입니다

Weight Decay (L1,L2)

weight 크기 줄여야

출처:<https://sooho-kim.tistory.com/85>

I1-norm과 I2-norm에 대하여

해당 포스팅은 Dive into Deep Learning과 Mathematics for Machine Learning을 참고하여 정리한 글입니다. 이 영화를 알고 계시다면 저와 비슷한 세대임은 분명하겠군요. (학생 때 영화관에서 봤던 기억이 있었

sooho-kim.tistory.com

L_p-norm

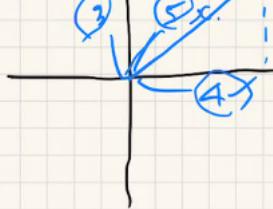
$$\|\underline{x}\|_p = \left(\sum_{i=1}^n |x_i|^p \right)^{\frac{1}{p}}$$

$$\|\underline{x}\|_1 = \sum |x_i|$$

$$\Rightarrow \|\underline{x}\|_2 = \left(\sum_{i=1}^n |x_i|^2 \right)^{\frac{1}{2}} = \sqrt{\sum_{i=1}^n |x_i|^2}$$

L₂-norm

$$\|\underline{x}\|_3$$



$$\checkmark \vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \sqrt{4^2 + 3^2} = 5$$

$$\|\vec{x}\|_3 = \sqrt{\sum |x_i|^2} = \sqrt{4^2 + 3^2} = 5$$

$$\vec{x} = (x_1 \underline{x_2} \underline{x_3} \underline{x_4 \dots}) = \sqrt{25} = 5$$

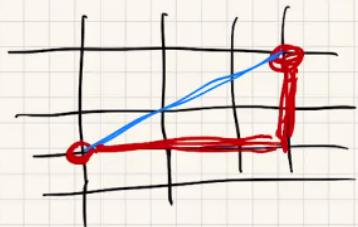
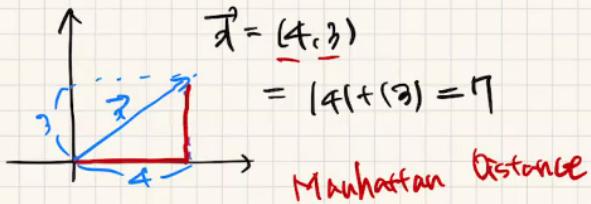
$$\checkmark \sqrt{\sum_{i=1}^n |x_i|^2} = \sqrt{x_1^2 + x_2^2 + x_3^2 + x_4^2 + \dots}$$

L_p Norm $\|x\|_p = \left(\sum_{i=1}^n |x_i|^p \right)^{\frac{1}{p}}$

$\|x\|_2 = \left(\sum_{i=1}^n |x_i|^2 \right)^{\frac{1}{2}}$

L₁ Norm $\|x\|_1 = \left(\sum_{i=1}^n |x_i| \right)^{\frac{1}{1}} = \sum_{i=1}^n |x_i|$

$= |x_1| + |x_2| + |x_3| + \dots$



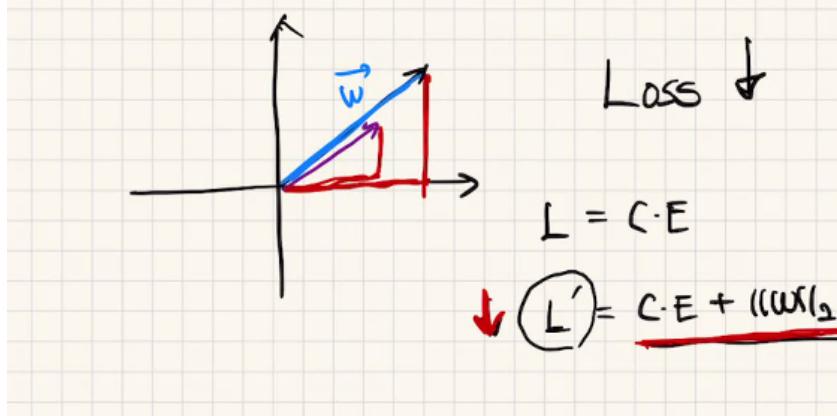
아래그림) 가중치 크기가 적어지면 함수가 복잡해지지 않아서 overfitting이 줄어든다.
L'은 새로운 L'으로 커지지만 loss를 줄어드는 방향으로 가서 overfitting이 줄어든다.

$L' = L + \|w\|_p$

Weight Decay

$\|w\|_1$ L₁-Regularization

$\|w\|_2$ L₂-Regularization



아래그림) L1(Lasso Regularization), L2(Ridge Regularization)

모델이 학습과정에서 불필요한 특징, 입력값에 의존하는것이 문제가됨

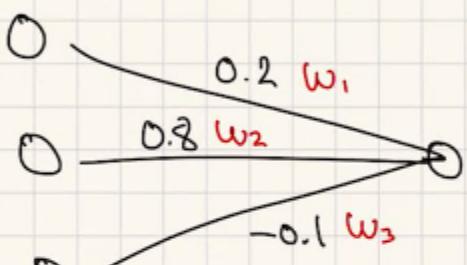
L1(Lasso Regularization), L2(Ridge Regularization)를 적용하게 되면 모델이 정말 중요한 가중치만 남기고

나머지는 0으로 유지하게 된다. 가중치를 필터링. 결과적으로 모델이 단순화된다.

불필요한 특징 (입력값)을
제거하는 것이 원리가 됨.

B로 가중치 w_i 를 0 시작

→ 확률을 통해 증감.

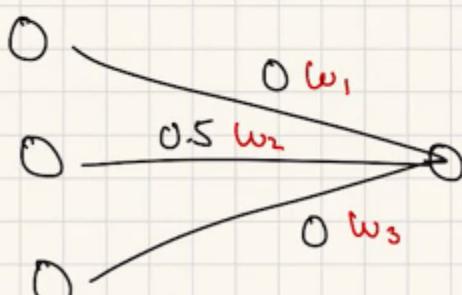


$$L1 = 0.2 + 0.8 + 0.1 = 1.1$$
$$L2 = \sqrt{0.2^2 + 0.8^2 + (-0.1)^2}$$
$$\approx 0.89$$

(L1, L2)

가장 큰 가중치의 penalty.

\Rightarrow 중요한 종료하는 가중치는 $L1$, 다른거는 0으로
유지된다.



$$L1 = 0.5$$

$$L2 = 0.5$$

$\sum_{i=1}^n \|w_i\| = \text{penalty}$

$L2$ 모든 가중치의 제곱의 합의
제곱을 쓰운것.

$$\sqrt{\sum_{i=1}^n w_i^2} = \text{penalty}$$

이미지에서 보여주는 L1 노름과 L2 노름은 정규화(regularization) 기법으로, 과적합을 방지 하는 데 매우 중요한 역할을 합니다. 이 두 가지 방법은 모델이 너무 복잡해지는 것을 막아, 훈련 데이터에 지나치게 맞춰지는 **과적합(overfitting)**을 방지합니다.

1. L1 노름 (L1 Norm)

- **L1 정규화(라쏘 정규화, Lasso Regularization)**라고도 불립니다.
- 수식:
- L1 정규화는 모델의 파라미터(가중치)의 절댓값을 합한 값에 페널티를 부과하는 방식입니다. 즉, 모델의 파라미터 값이 0에 가까워지도록 유도합니다.
- **효과:** L1 노름은 **희소성(sparsity)**을 만들어냅니다. 이는 **일부 불필요한 가중치들을 0으로 만들어**, 모델이 불필요한 특성에 의존하지 않게 합니다. 이로 인해 모델이 단순해지며, 과적합이 방지됩니다.
 - **L1 정규화**는 특히 데이터에서 중요한 특성만 남기고 나머지 특성의 영향을 줄이는 효과가 있어, 특징 선택(feature selection)에 자주 사용됩니다.

$$\text{수식: } \|\mathbf{x}\|_1 = \sum_{i=1}^n |x_i|$$

2. L2 노름 (L2 Norm)

- **L2 정규화(릿지 정규화, Ridge Regularization)**라고도 불립니다.
- 수식:
-
- L2 정규화는 모델의 파라미터(가중치)의 제곱을 합한 값에 페널티를 부과합니다. L2 정규화는 가중치들이 지나치게 크거나 작아지지 않도록 제어하여, 모델이 너무 복잡해지지 않도록 합니다.
- **효과:** L2 노름은 **가중치의 크기를 제한**하여 모델이 모든 특성에 대해 고르게 학습하도록 돕습니다. 모델이 과도하게 특정 특성에 의존하지 않게 하여, 과적합을 방지합니다.
 - L2 정규화는 파라미터 값이 0으로 수렴하지 않지만, 각 파라미터가 작은 값을 가지도록 만들어 모델이 더 일반화됩니다.

$$\text{수식: } \|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^n x_i^2}$$

과적합 방지와 L1, L2 노름

- 과적합은 모델이 훈련 데이터의 모든 세부사항과 노이즈까지 학습할 때 발생합니다. L1 또는 L2 노름을 사용한 정규화는 모델이 불필요하게 복잡해지는 것을 막아, 모델이 새로운 데이터에 대해 더 잘 일반화될 수 있도록 합니다.
- **L1 정규화**는 불필요한 특성을 제거하는 경향이 있고, **L2 정규화**는 가중치를 고르게 만들어 모델을 단순화하는 효과가 있습니다. 이로 인해 두 정규화 방법 모두 과적합을 줄이는 데 효과적입니다.

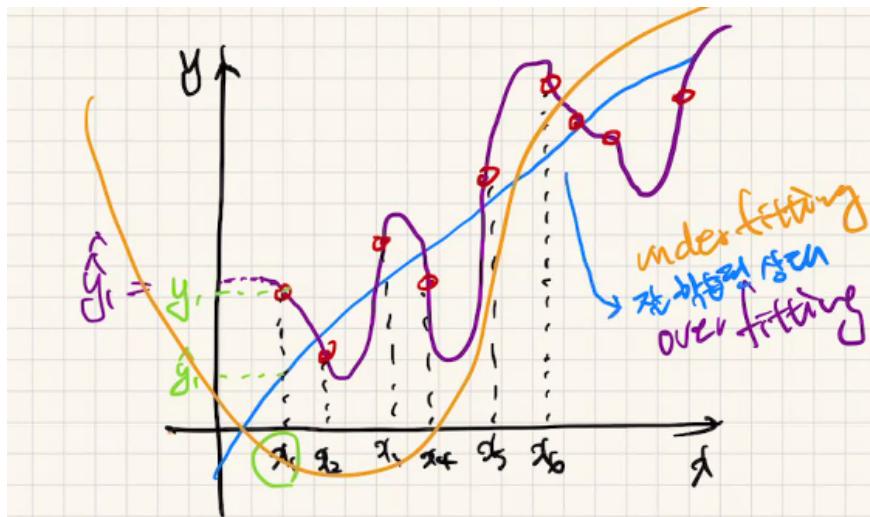
따라서, L1 및 L2 정규화는 모델의 복잡성을 줄이고, 과적합을 방지하는 주요 기법입니다. 두 노름 모두 모델이 특정한 데이터 포인트나 특성에 지나치게 의존하지 않도록 하여, 모델이 더 일반화될 수 있게 도와줍니다.

아래그림)

초기상태는 underfitting,

너무 overfitting(학습이 많이해서 거의 외운상태) 하면 train loss가 거의 0 됨(loss가 적다고 좋 은게 아님)

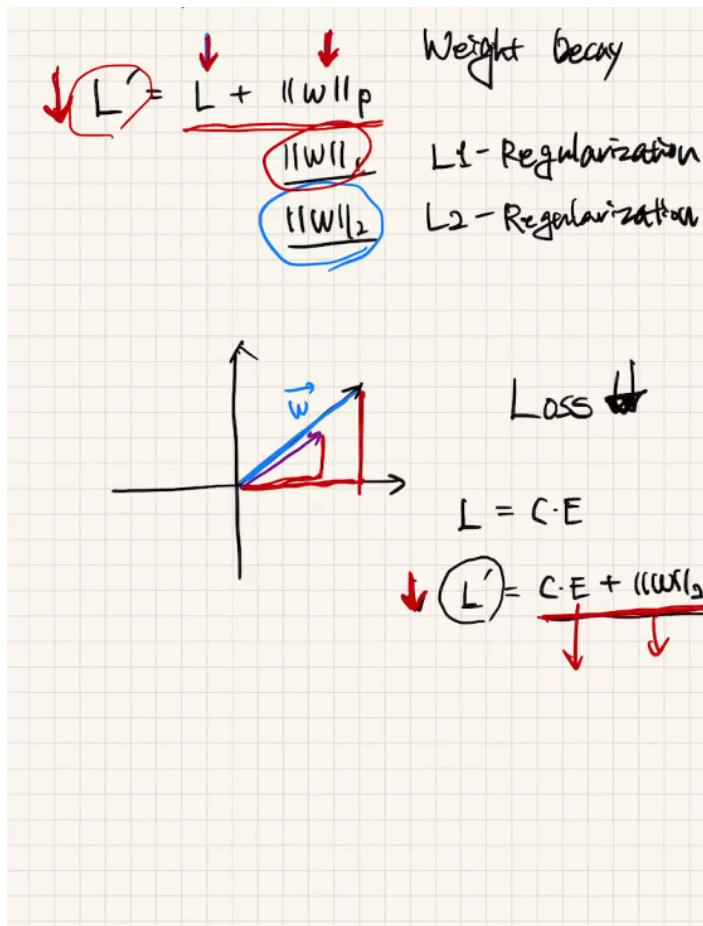
파란색선은 학습이 잘된상태



$$\text{Loss} = \frac{1}{n} \sum (y - \hat{y})^2$$

II(반복) Loss = $\frac{1}{n} \left[\underbrace{(y_1 - \hat{y}_1)^2}_{\text{0}} + \underbrace{(y_2 - \hat{y}_2)^2}_{\text{0}} + \dots \right]$

IV(최종) Loss = $\frac{1}{n} \left(\underbrace{(y_1 - \hat{y}_1)^2}_{\text{0}} + \underbrace{(y_2 - \hat{y}_2)^2}_{\text{0}} + \dots \right)$
 $= 0$

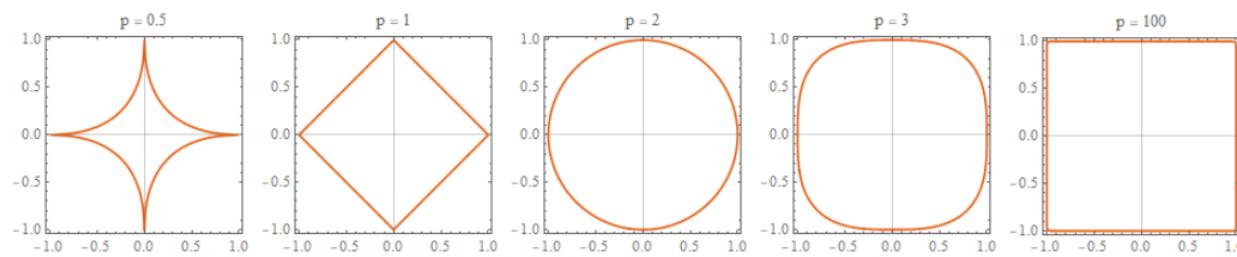


2. l_p -norm

일반적인 식으로 표현하는 l_p -norm은 다음과 같은 식으로 표현됩니다.

$$\|x\|_p = \left(\sum_{i=1}^n |x_i|^p \right)^{\frac{1}{p}}$$

이 식에서 $p=1$ 라면, l_1 -norm, $p=2$ 라면, l_2 -norm이 됩니다.

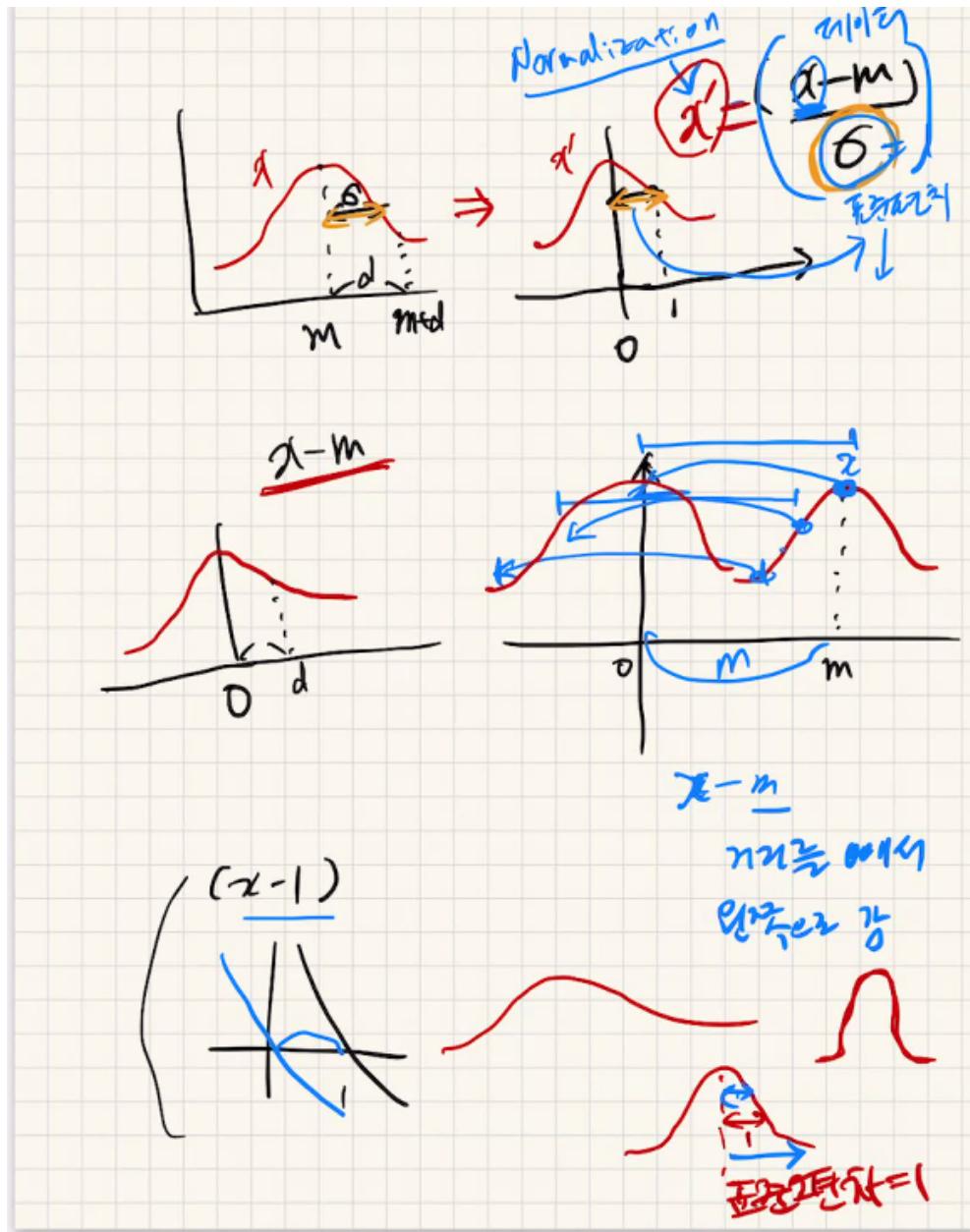


Batch Normalization

Batch
 $\vec{x}_1 = (\vec{x}_1^1, \vec{x}_1^2, \vec{x}_1^3, \vec{x}_1^4, \dots)$ 평균
 $\vec{x}_2 = (\vec{x}_2^1, \vec{x}_2^2, \vec{x}_2^3, \vec{x}_2^4, \vec{x}_2^5, \dots)$
 $\vec{x}_3 = (\vec{x}_3^1, \vec{x}_3^2, \vec{x}_3^3, \vec{x}_3^4, \vec{x}_3^5, \dots)$
 $\vec{x}_4 = (\vec{x}_4^1, \vec{x}_4^2, \vec{x}_4^3, \vec{x}_4^4, \vec{x}_4^5, \dots)$
 ↓
 평균, 표준편차
 평균을 기준으로
 m 평균
 n 표준편차
 평균을 기준으로 표준화
 $m = \frac{\vec{x}_1 + \vec{x}_2 + \vec{x}_3 + \vec{x}_4}{n}$
 표준편차 = $\sqrt{\frac{(\vec{x}_1 - m)^2 + (\vec{x}_2 - m)^2 + (\vec{x}_3 - m)^2 + (\vec{x}_4 - m)^2}{n}}$
분산 → 표준편차²
 $\frac{\vec{x} - m}{\sigma}$
 평균을 기준으로 표준화
 평균을 기준으로 표준화
 평균을 기준으로 표준화
 평균을 기준으로 표준화

normalization

m (평균)= 0 , 표준편차=1로 만들어준다.(원래 표준편자는 1이 같거나 아니었는데 1로만듬)
 batch normalization

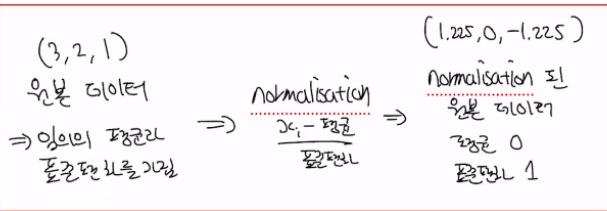


- 평균 0 으로 만들기
 $3, 2, 1 \rightarrow 평균 2$

평균 2

분산: $\frac{(3-2)^2 + (2-2)^2 + (1-2)^2}{3} = \frac{1+0+1}{3} = \frac{2}{3}$

표준편차: $\sqrt{\frac{2}{3}}$



만약 원본 데이터의 표준편수가 0이 나온다면

Normalisation 공식 $\frac{x_i - 평균}{표준편차}$ 에서, 0으로 나누기 힘 (오류)

따라서 분산이 아주 작은 값을 대해서 0으로 나누는 것 방지

- normalisation 된 데이터

3, 2, 1

$$\Rightarrow \frac{3-2}{\sqrt{\frac{2}{3}}}, \frac{2-2}{\sqrt{\frac{2}{3}}}, \frac{1-2}{\sqrt{\frac{2}{3}}} \quad (\text{식: } \frac{x_i - 평균}{표준편차})$$

$$\Rightarrow \frac{1}{\sqrt{\frac{2}{3}}}, \frac{0}{\sqrt{\frac{2}{3}}}, \frac{-1}{\sqrt{\frac{2}{3}}}$$

$$\Rightarrow 1.225, 0, -1.225 \quad (\text{근사값})$$

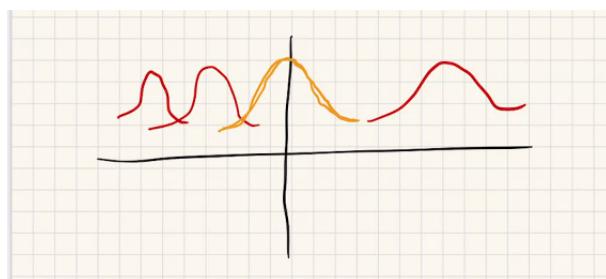
평균 0

$$\text{분산 } \frac{(1.225-0)^2 + (0-0)^2 + (-1.225-0)^2}{3}$$

$$\text{표준편차} \sqrt{\frac{(1.225-0)^2 + (0-0)^2 + (-1.225-0)^2}{3}} \\ = \sqrt{\frac{1.5+0+1.5}{3}} = \sqrt{\frac{3}{3}} = 1$$

batch normalization

- 데이터의 batch를 normalization 한다.
- 정리를 하니 overfitting 이 된다.



분산 = 편차의 ^2 의 평균

루트 -> 편차의 평균 -> 표준적인 평균 = 평균편차 = 표준편차

$m(\text{평균}) = 0$, 표준편차 = 1

$\vec{x}_1 = (x_1^1, x_1^2, x_1^3, x_1^4, x_1^5 \dots)$ 차원
 $\vec{x}_2 = (x_2^1, x_2^2, x_2^3, x_2^4, x_2^5 \dots)$
 $\vec{x}_3 = (x_3^1, x_3^2, x_3^3, x_3^4, x_3^5 \dots)$
 $\vec{x}_4 = (x_4^1, x_4^2, x_4^3, x_4^4, x_4^5 \dots)$

Batch \downarrow 평균, 표준편차

평균 $m = \frac{x_1 + x_2 + x_3 + x_4}{n}$ 평균을 기준으로
 표준편차 $s = \sqrt{\frac{(x_1 - m)^2 + (x_2 - m)^2 + (x_3 - m)^2 + (x_4 - m)^2}{n}}$ 표준편차는 평균과 차이의 제곱근의 제곱근

표준편차 $s = \sqrt{\frac{(x_1 - m)^2 + (x_2 - m)^2 + (x_3 - m)^2 + (x_4 - m)^2}{n}}$

분산 \rightarrow 표준편차² $s^2 = \frac{(x_1 - m)^2 + (x_2 - m)^2 + (x_3 - m)^2 + (x_4 - m)^2}{n}$ 표준편차의 제곱

표준편차 $s = \sqrt{\frac{(x_1 - m)^2 + (x_2 - m)^2 + (x_3 - m)^2 + (x_4 - m)^2}{n}}$ 표준편차

표준편차는 평균과 차이의 제곱근의 제곱근

$m(\text{평균})=0, \text{표준편차}=1$

Code SUMMARY

성씨를 보고 국적을 맞추는 Multi class classification (class 는 국가종류)

mlp모델을 학습시켜서 해결

초기화

#Data 다운받는것

```
# 만약 코랩에서 실행하는 경우 아래 코드를 실행하여 전처리된 라이트 버전의 데이터를 다운로드하세요.
```

!mkdir data

#Data 다운받

는것

```
!wget https://git.io/JtaFp -O data/download.py
!wget https://git.io/Jtabe -O data/get-all-data.sh
!chmod 755 data/get-all-data.sh
%cd data
!./get-all-data.sh
%cd ..
```

The screenshot shows the Google Colab interface with two panes. The left pane displays the Python code for the `SurnameDataset` class and its methods. The right pane shows the output of the code execution, including a DataFrame and the generated `SurnameVectorizer` object.

Code (Left Pane):

```
class SurnameDataset(Dataset):
    def __init__(self, surname_df, vectorizer):
        frequencies = [count for _, count in sorted(surname_df.items(), key=lambda item: item[1], reverse=True)]
        self.class_weights = 1.0 / torch.tensor(frequencies, dtype=torch.float32)
        # a = SurnameDataset(df, vectorizer)
        # dataframe을 불러와서 surname dataset instance를 만든다.

    @classmethod
    def load_dataset_and_make_vectorizer(cls, surname_csv):
        # 내부에서는 자기함수를 쓸수있어서 cls 를 쓴다.
        """데이터셋을 로드하고 새로운 SurnameVectorizer 객체를 만들고"""
        review_csv(str): 데이터셋의 위치
        반환값:
        SurnameDataset의 인스턴스
        """
        surname_df = pd.read_csv(surname_csv)
        train_surname_df = surname_df[surname_df['split']=='train']
        return cls(surname_df, SurnameVectorizer.from_dataframe(train_surname_df))
        #class를 호출해서 instance 를 만듬

    @classmethod
    def load_dataset_and_load_vectorizer(cls, surname_csv, vectorizer_filepath):...
```

Output (Right Pane):

```
178  class SurnameDataset(Dataset):
179      def __init__(self, surname_df, vectorizer):
180          frequencies = [count for _, count in sorted(surname_df.items(), key=lambda item: item[1], reverse=True)]
181          self.class_weights = 1.0 / torch.tensor(frequencies, dtype=torch.float32)
182          # a = SurnameDataset(df, vectorizer)
183          # dataframe을 불러와서 surname dataset instance를 만든다.
184
185          @classmethod
186          def load_dataset_and_make_vectorizer(cls, surname_csv):
187              """내부에서는 자기함수를 쓸수있어서 cls 를 쓴다.
188                  데이터셋을 로드하고 새로운 SurnameVectorizer 객체를 만들고"""
189              review_csv (str): 데이터셋의 위치
190              반환값:
191              SurnameDataset의 인스턴스
192              """
193              surname_df = pd.read_csv(surname_csv)
194              train_surname_df = surname_df[surname_df['split']=='train']
195              return cls(surname_df, SurnameVectorizer.from_dataframe(train_surname_df))
196              #class를 호출해서 instance 를 만듬
197
198              @classmethod
199              def load_dataset_and_load_vectorizer(cls, surname_csv, vectorizer_filepath):...
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
```

Dataframe Output:

	nationality	nationality_index	split	surname
0	Arabic	15	train	Totalh
1	Arabic	15	train	Aboud
2	Arabic	15	train	Fakhoury

The screenshot shows the Google Colab interface with two panes. The left pane displays the Python code for the `SurnameDataset` class and its methods. The right pane shows the output of the code execution, including a detailed explanation of the `save_vectorizer` method.

Code (Left Pane):

```
class SurnameDataset(Dataset):
    def __init__(self, surname_df, vectorizer):
        review_csv(str): 데이터셋의 위치
        반환값:
        SurnameDataset의 인스턴스
        """
        surname_df = pd.read_csv(surname_csv)
        train_surname_df = surname_df[surname_df['split']=='train']
        return cls(surname_df, SurnameVectorizer.from_dataframe(train_surname_df))
        #class를 호출해서 instance 를 만듬

    @classmethod
    def load_dataset_and_load_vectorizer(cls, surname_csv, vectorizer_filepath):...
```

Output (Right Pane):

```
178  class SurnameDataset(Dataset):
179      def __init__(self, surname_df, vectorizer):
180          review_csv (str): 데이터셋의 위치
181          반환값:
182          SurnameDataset의 인스턴스
183          """
184          surname_df = pd.read_csv(surname_csv)
185          train_surname_df = surname_df[surname_df['split']=='train']
186          return cls(surname_df, SurnameVectorizer.from_dataframe(train_surname_df))
187          #class를 호출해서 instance 를 만듬
188
189          @classmethod
190          def load_dataset_and_load_vectorizer(cls, surname_csv, vectorizer_filepath):...
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
```

Dataframe Output:

	nationality	nationality_index	split	surname
0	Arabic	15	train	Totalh
1	Arabic	15	train	Aboud
2	Arabic	15	train	Fakhoury

```

from argparse import Namespace
...
class SurnameDataset(Dataset):
    def __init__(self, surname_df, vectorizer):
        ...
        self.surname_df = surname_df
        self.vectorizer = vectorizer
        ...
        self.train_df = self.surname_df[self.surname_df['split']=='train']
        self.train_size = len(self.train_df)
        ...
        self.val_df = self.surname_df[self.surname_df['split']=='val']
        self.validation_size = len(self.val_df)
        ...
        self.test_df = self.surname_df[self.surname_df['split']=='test']

```

Annotations on the right side of the code:

- Red circle: `# a = SurnameDataset(df, vectorizer)`
- Red arrow: `# dataframe을 불러와서 surname dataset instance를 만든다.`
- Red circle: `# a.surname_df -> surname_df`
- Red arrow: `# self.surname_df = surname_df`
- Red circle: `def __init__(self, surname_df, vectorizer):`
- Red arrow: `""" #__init__은 class 생성될 때 호출되는 함수 매개변수: surname_df (pandas.DataFrame): 데이터셋 vectorizer (SurnameVectorizer): SurnameVectorizer 객체 내부(서자)에 무조건 들어가 있는 것"""`
- Red circle: `self.surname_df = surname_df #내부에서만`
- Red arrow: `self._vectorizer = vectorizer`
- Red circle: `self.train_df = self.surname_df[self.surname_df['split']=='train']`
- Red arrow: `self.train_size = len(self.train_df)`
- Red circle: `self.val_df = self.surname_df[self.surname_df['split']=='val']`
- Red arrow: `self.validation_size = len(self.val_df)`
- Red circle: `self.test_df = self.surname_df[self.surname_df['split']=='test']`
- Red arrow: `self.test_size = len(self.test_df)`

```

from argparse import Namespace
...
class SurnameDataset(Dataset):
    def __init__(self, surname_df, vectorizer):
        ...
        self.validation_size = len(self.val_df)
        ...
        self.test_df = self.surname_df[self.surname_df['split']=='test']
        self.test_size = len(self.test_df)
        ...
        self._lookup_dict = {'train': (self.train_df, self.train_size),
                            'val': (self.val_df, self.validation_size),
                            'test': (self.test_df, self.test_size)}
        ...
        self.set_split('train')
        ...
        class_counts = surname_df.nationality.value_counts().to_dict()
        ...
        def sort_key(item):
            return self._vectorizer.nationality_vocab.lookup_token(item[0])
        ...
        sorted_counts = sorted(class_counts.items(), key=sort_key)
        frequencies = [count for _, count in sorted_counts]
        self._class_weights = 1.0 / torch.tensor(frequencies, dtype=torch.float32)
        ...
        # a = SurnameDataset(df, vectorizer)

```

Annotations on the right side of the code:

- Red circle: `def __init__(self, surname_df, vectorizer):`
- Red arrow: `""" 함수 init 을 만들어서 이 함수가 실행 """`
- Red circle: `self.validation_size = len(self.val_df)`
- Red arrow: `self.validation_size = len(self.val_df)`
- Red circle: `self.test_df = self.surname_df[self.surname_df['split']=='test']`
- Red arrow: `self.test_df = self.surname_df[self.surname_df['split']=='test']`
- Red circle: `self.test_size = len(self.test_df)`
- Red arrow: `self.test_size = len(self.test_df)`
- Red circle: `self._lookup_dict = {'train': (self.train_df, self.train_size),`
- Red arrow: `'val': (self.val_df, self.validation_size),`
- Red circle: `'test': (self.test_df, self.test_size)}`
- Red arrow: `'test': (self.test_df, self.test_size)}`
- Red circle: `self.set_split('train')`
- Red arrow: `self.set_split('train')`
- Red circle: `# 클래스 가중치`
- Red arrow: `class_counts = surname_df.nationality.value_counts().to_dict()`
- Red circle: `def sort_key(item):`
- Red arrow: `return self._vectorizer.nationality_vocab.lookup_token(item[0])`
- Red circle: `sorted_counts = sorted(class_counts.items(), key=sort_key)`
- Red arrow: `sorted_counts = sorted(class_counts.items(), key=sort_key)`
- Red circle: `frequencies = [count for _, count in sorted_counts]`
- Red arrow: `frequencies = [count for _, count in sorted_counts]`
- Red circle: `self._class_weights = 1.0 / torch.tensor(frequencies, dtype=torch.float32)`
- Red arrow: `self._class_weights = 1.0 / torch.tensor(frequencies, dtype=torch.float32)`
- Red circle: `# a = SurnameDataset(df, vectorizer)`
- Red arrow: `# a = SurnameDataset(df, vectorizer)`

Rectified Linear Unit 렉티파
ReLU는 입력값이 0보다 작거나 출력하는 유닛인데,

보기 도구 세이프서치

나전

ReLU 함수를 쓸까?
· 1. 계산 효율성 · 2. Gradient(기울기) 소실
네트워크에 더 ...

ChatGPT Click here to ask ChatGPT

AI Tools - Become Superhuman Today!

```

from argparse import Namespace
...
class SurnameClassifier(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        ...
        super(SurnameClassifier, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)
        ...
        fully_connected_layer를 지정하고 MLP의 정방향 계산
        """
        def forward(self, x_in, apply_softmax=False):
            ...
            개별 변수:
            x_in (torch.Tensor): 입력 데이터 텐서
            x_in.shape는 (batch, input_dim)입니다.
            apply_softmax (bool): 소프트맥스 활성화 함수를 위한 플래그
            ...
            크로스-엔트로피 손실을 사용하려면 False로 지정해야 합니다.
            반환 값:
            결과 텐서. tensor.shape은 (batch, output_dim)입니다.
            ...
            intermediate_vector = F.relu(self.fc1(x_in))
            prediction_vector = self.fc2(intermediate_vector)
            ...
            if apply_softmax:
                prediction_vector = F.softmax(prediction_vector, dim=1)

```

```

from argparse import Namespace
class SurnameDataset(Dataset):
    def __init__(self, input_dim, hidden_dim, output_dim):
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x_in, apply_softmax=False):
        intermediate_vector = F.relu(self.fc1(x_in))
        prediction_vector = self.fc2(intermediate_vector)
        if apply_softmax:
            prediction_vector = F.softmax(prediction_vector, dim=1)
        return prediction_vector

```

```

from argparse import Namespace
class SurnameDataset(Dataset):
    def __init__(self, input_dim, hidden_dim, output_dim):
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x_in, apply_softmax=False):
        intermediate_vector = F.relu(self.fc1(x_in))
        prediction_vector = self.fc2(intermediate_vector)
        if apply_softmax:
            prediction_vector = F.softmax(prediction_vector, dim=1)
        return prediction_vector

```

```

class SurnameClassifier(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(SurnameClassifier, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x_in, apply_softmax=False):
        intermediate_vector = F.relu(self.fc1(x_in))
        prediction_vector = self.fc2(intermediate_vector)
        if apply_softmax:
            prediction_vector = F.softmax(prediction_vector, dim=1)
        return prediction_vector

```

vectorize는 단어를 vector화

참고로 one hot vector는 하나만 1이고,

multil hot vector는 여러개가 1

The screenshot shows two side-by-side code editors in Google Colab. On the left, a DataFrame named 'df' is displayed with columns: nationality, nationality_index, split, and surname. A red arrow points from the 'surname' column to the right-hand code editor. The right-hand code editor contains a Python class definition for 'SurnameDataset'. The 'set_split' method is highlighted, showing its implementation for selecting a split ('train', 'val', or 'test') based on the input string. The code uses a lookup dictionary to map nationalities to target sizes.

```
from argparse import Namespace
class SurnameDataset(Dataset):
    def __init__(self):
        # ...
    @classmethod
    def load_dataset_and_load_vectorizer(cls, surname_vectorizer_filepath):
        # ...
    @staticmethod
    def load_vectorizer_only(vectorizer_filepath):
        # ...
    def save_vectorizer(self, vectorizer_filepath):
        # ...
    def get_vectorizer(self):
        # ...
    def set_split(self, split="train"):
        """데이터프레임에 있는 열을 사용해 분할 세트를 선택합니다"""
        self._target_split = split
        self._target_df, self._target_size = self._lookup_dict[split]
    def __len__(self):
        return self._target_size
    def __getitem__(self, index):
        """파이토치 데이터셋의 주요 진입 메서드"""
        # ...
    # ...
# ...
def len(df.nationality.unique()):
    array(['Arabic', 'Chinese', 'Czech', 'Dutch', 'English', 'French',
           'German', 'Greek', 'Irish', 'Italian', 'Japanese', 'Korean',
           'Polish', 'Portuguese', 'Russian', 'Scottish', 'Spanish',
           'Vietnamese'], dtype=object)
```

This screenshot shows the same Google Colab environment as the previous one. It displays the same DataFrame 'df' and the same Python code for the 'SurnameDataset' class. The 'set_split' method is again highlighted. The code remains identical to the first screenshot, demonstrating the implementation of vectorization and splitting logic.

```
from argparse import Namespace
class SurnameDataset(Dataset):
    def __init__(self):
        # ...
    @classmethod
    def load_dataset_and_load_vectorizer(cls, surname_vectorizer_filepath):
        # ...
    @staticmethod
    def load_vectorizer_only(vectorizer_filepath):
        # ...
    def save_vectorizer(self, vectorizer_filepath):
        # ...
    def get_vectorizer(self):
        # ...
    def set_split(self, split="train"):
        """데이터프레임에 있는 열을 사용해 분할 세트를 선택합니다"""
        self._target_split = split
        self._target_df, self._target_size = self._lookup_dict[split]
    def __len__(self):
        return self._target_size
    def __getitem__(self, index):
        """파이토치 데이터셋의 주요 진입 메서드"""
        # ...
    # ...
# ...
def len(df.nationality.unique()):
    array(['Arabic', 'Chinese', 'Czech', 'Dutch', 'English', 'French',
           'German', 'Greek', 'Irish', 'Italian', 'Japanese', 'Korean',
           'Polish', 'Portuguese', 'Russian', 'Scottish', 'Spanish',
           'Vietnamese'], dtype=object)
```

from argparse import Namespace

```

4-2_Classifying_Surname
파일 수정 보기 삽입 런타임 도구 텍스트 도구
+ 코드 + 텍스트
[ ] # 정확도를 계산합니다
acc_t = compute_accuracy(y_pred, batch_dict['y_surname'])
running_acc += (acc_t - running_acc) / (batch_index + 1)

# 진행 바 업데이트
train_bar.set_postfix({'loss': running_loss, 'acc': running_acc}, refresh=True)
train_bar.update()

train_state['train_loss'].append(running_loss)
train_state['train_acc'].append(running_acc)

# 검증 세트에 대한 순회
# 검증 세트와 배치 제너레이터 준비, 순회와 정확도를 0으로 설정
dataset.set_split('val')
batch_generator = generate_batches(dataset,
                                   batch_size=args.batch_size,
                                   device=args.device)

running_loss = 0.
running_acc = 0.
classifier.eval()

for batch_index, batch_dict in enumerate(batch_generator):
    # 단계 1. 출력을 계산합니다
    y_pred = classifier(batch_dict['x_surname'])

    # 단계 2. 손실을 계산합니다
    loss = loss_func(y_pred, batch_dict['y_nationality'])
    loss_t = loss.to('cpu').item()
    running_loss += (loss_t - running_loss) / (batch_index + 1)

```

모델 예측값
surname
classifier
label
모델 예측값이 label과 비교해서 loss가 적게 되도록 classifier weight가 예측값이 변형되면서 loss가 점점 적어짐

from argparse import Namespace

```

4-2_Classifying_Surnames_with_an_MLP.ipynb
파일 수정 보기 삽입 런타임 도구 모드 변경사 환경 지정자로 Colab AI
+ 코드 + 텍스트
# 정확도를 계산합니다
acc_t = compute_accuracy(y_pred, batch_dict['y_surname'])
running_acc += (acc_t - running_acc) / (batch_index + 1)

# 진행 바 업데이트
train_bar.set_postfix({'loss': running_loss, 'acc': running_acc}, refresh=True)
train_bar.update()

train_state['train_loss'].append(running_loss)
train_state['train_acc'].append(running_acc)

# 검증 세트에 대한 순회
# 검증 세트와 배치 제너레이터 준비, 순회와 정확도를 0으로 설정
dataset.set_split('val')
batch_generator = generate_batches(dataset,
                                   batch_size=args.batch_size,
                                   device=args.device)

running_loss = 0.
running_acc = 0.
classifier.eval()

for batch_index, batch_dict in enumerate(batch_generator):
    # 단계 1. 출력을 계산합니다
    y_pred = classifier(batch_dict['x_surname'])

    # 단계 2. 손실을 계산합니다
    loss = loss_func(y_pred, batch_dict['y_nationality'])
    loss_t = loss.to('cpu').item()
    running_loss += (loss_t - running_loss) / (batch_index + 1)

```

from argparse import Namespace

```

class Vocabulary(Namespace):
    def __init__(self):
        self._token_to_idx = {}
        self._idx_to_token = []
        self._unk_index = None
        self._size = 0

    def add_many(self, tokens):
        indices = [self.add_token(token) for token in tokens]
        return indices

    def lookup_token(self, token):
        if token in self._token_to_idx:
            return self._token_to_idx[token]
        else:
            if self._unk_index is None:
                self._unk_index = len(self._token_to_idx)
            return self._unk_index

    def lookup_index(self, index):
        if index in self._idx_to_token:
            return self._idx_to_token[index]
        else:
            return None

```



```

# 성화도를 계산합니다
acc_t = compute_accuracy(y_pred, batch_dict)
running_acc += (acc_t - running_acc) / (batch_index + 1)

# 진행 바 업데이트
train_bar.set_postfix(loss=running_loss, acc=running_acc,
epoch=epoch_index)
train_bar.update()

train_state['train_loss'].append(running_loss)
train_state['train_acc'].append(running_acc)

# 검증 세트에 대한 순회

# 검증 세트와 배치 제너레이터 준비, 손실과 정확도를 0으로 설정
dataset.set_split('val')
batch_generator = generate_batches(dataset,
batch_size=args.batch_size,
device=args.device)

running_loss = 0.
running_acc = 0.
classifier.eval()

for batch_index, batch_dict in enumerate(batch_generator):

    # 단계 1. 출력을 계산합니다
    y_pred = classifier(batch_dict['x_surname'])

    # 단계 2. 손실을 계산합니다
    loss = loss_func(y_pred, batch_dict['y_nationality'])
    loss_t = loss.item()
    running_loss += (loss_t - running_loss) / (batch_index + 1)

```

이 함수는 파일 헤더에서 정의된 `__getitem__` 메서드를 사용합니다.

매개 변수:

- `index (int)`: 데이터 포인트의 인덱스
- 반환 값:
 - 데이터 포인트의 특성 (`x_surname`)과 레이블 (`y_nationality`)로 이루어진 딕셔너리

```python
row = self.\_target\_df.loc[index]
surname\_vector = \
self.\_vectorizer.vectorize(row.surname)
# multihot vector도 surname\_vector에 넣는다.
nationality\_index = \
self.\_vectorizer.nationality\_vocab.lookup\_token(
row.nationality)
# 단어가 넣어서 몇번인덱스인지 알아서 nationality\_index에 넣는다.
return {'x\_surname': surname\_vector,
'y\_nationality': nationality\_index}
```
`def get_num_batches(self, batch_size):`
 배치 크기가 주어지면 데이터셋으로 만들 수 있는 배치 개수를 반환합니다

두 개를 비교할 때 CrossEntropyLoss 사용 모델 예측값 label은 loss 계산 시 onehot vector 가나가 아니라서

모델 예측값이 label과 비교해서 loss 계산 loss 가 적게 되도록 classifier weight 가 update되고 예측값이 변경되면서 ↪ 가 저자 저자 저자 저자

원래 weight w 기울기 $\frac{\partial L}{\partial w}$ 미분값은 learning rate : 얼마 만큼 갑자

File: 4-2_Classifying_Surnames_with_an_MLP.ipynb

```

optimizer = optim.Adam(classifier.parameters(), lr=args.learning_rate)
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer=optimizer, # lr는 scheduler에 넣어야 한다.
                                                mode='min', factor=0.5, # lr은 0.5로 줄어든다.
                                                patience=1)

train_state = make_train_state(args)

epoch_bar = tqdm.notebook.tqdm(desc='training routine',
                                total=args.num_epochs,
                                position=0)

dataset.set_split('train')
train_bar = tqdm.notebook.tqdm(desc='split=train',
                                total=dataset.get_num_batches(args.b),
                                position=1,
                                leave=True)

dataset.set_split('val')
val_bar = tqdm.notebook.tqdm(desc='split=val',
                                total=dataset.get_num_batches(args.b),
                                position=1,
                                leave=True)

try:
    for epoch_index in range(args.num_epochs):
        train_state['epoch_index'] = epoch_index

        # 훈련 세트와 배치 제너레이터 준비, 손실과 정확도를 0으로 설정
        dataset.set_split('train')
        batch_generator = generate_batches(dataset,
                                            batch_size=args.batch_size,
                                            device=args.device)

        running_loss = 0.0
        running_acc = 0.0
        classifier.train()

        for batch_index, batch_dict in enumerate(batch_generator):
            # 훈련 과정은 5단계로 이루어집니다

            # 단계 1. 그레이디언트를 0으로 초기화합니다
            optimizer.zero_grad()

            # 단계 2. 출력을 계산합니다
            y_pred = classifier(batch_dict['x_surname'])

            # 단계 3. 손실을 계산합니다
            loss = loss_func(y_pred, batch_dict['y_nationality'])
            loss_t = loss.item()
            running_loss += (loss_t - running_loss) / (batch_index + 1)

            # 단계 4. 손실을 사용해 그레이디언트를 계산합니다
            loss.backward()

            # 단계 5. 옵티マイ저로 가중치를 업데이트합니다
            optimizer.step()

```

training routine: 100% 100/100 [03:41<00:00, 1.98s/it]

split=train: 99% 119/120 [03:41<00:02, 2.75s/it, acc=51.8, epoch=9]

split=val: 96% 24/25 [03:41<00:00, 21.07s/it, acc=45.4, epoch=9]

가장 좋은 모델을 사용해 테스트 세트의 손실과 정확도를 계산합니다
classifier.load_state_dict(torch.load(train_state['model_filename']))
classifier = classifier.to(args.device)

File: 4-2_Classifying_Surnames_with_an_MLP.ipynb

```

# 훈련 세트와 배치 제너레이터 준비, 손실과 정확도를 0으로 설정
dataset.set_split('train')
batch_generator = generate_batches(dataset,
                                    batch_size=args.batch_size,
                                    device=args.device)

running_loss = 0.0
running_acc = 0.0
classifier.train()

for batch_index, batch_dict in enumerate(batch_generator):
    # 훈련 과정은 5단계로 이루어집니다

    # 단계 1. 그레이디언트를 0으로 초기화합니다
    optimizer.zero_grad()

    # 단계 2. 출력을 계산합니다
    y_pred = classifier(batch_dict['x_surname'])

    # 단계 3. 손실을 계산합니다
    loss = loss_func(y_pred, batch_dict['y_nationality'])
    loss_t = loss.item()
    running_loss += (loss_t - running_loss) / (batch_index + 1)

    # 단계 4. 손실을 사용해 그레이디언트를 계산합니다
    loss.backward()

    # 단계 5. 옵티マイ저로 가중치를 업데이트합니다
    optimizer.step()

```

학습 5단계

단계 1. 그레이디언트를 0으로 초기화합니다
optimizer.zero_grad()

단계 2. 출력을 계산합니다
y_pred = classifier(batch_dict['x_surname'])

단계 3. 손실을 계산합니다
loss = loss_func(y_pred, batch_dict['y_nationality'])
loss_t = loss.item()
running_loss += (loss_t - running_loss) / (batch_index + 1)

단계 4. 손실을 사용해 그레이디언트를 계산합니다
loss.backward()

단계 5. 옵티マイ저로 가중치를 업데이트합니다
optimizer.step()

File: 4-2_Classifying_Surnames_with_an_MLP.ipynb

```

batch_size=args.batch_size,
device=args.device)

running_loss = 0.0
running_acc = 0.0
classifier.train()

for batch_index, batch_dict in enumerate(batch_generator):
    # 훈련 과정은 5단계로 이루어집니다

    # 단계 1. 그레이디언트를 0으로 초기화합니다
    optimizer.zero_grad()

    # 단계 2. 출력을 계산합니다
    y_pred = classifier(batch_dict['x_surname'])

    # 단계 3. 손실을 계산합니다
    loss = loss_func(y_pred, batch_dict['y_nationality'])
    loss_t = loss.item()
    running_loss += (loss_t - running_loss) / (batch_index + 1)

    # 단계 4. 손실을 사용해 그레이디언트를 계산합니다
    loss.backward()

    # 단계 5. 옵티マイ저로 가중치를 업데이트합니다
    optimizer.step()

```

예측값 나오게하는 단계

모델 예측값이 label과 비교할 때 CrossEntropyLoss 사용 label은 onehot 형태로 되어 있으므로 loss 계산 시 onehot loss가 적용되도록 classifier weight 가 update되고 예측값이 변환되면서 loss가 점점 적어짐

원래 weight w \rightarrow $w - \frac{\partial L}{\partial w}$ gradient
learning rate : 얼마나 많이 갈지

4-2_Classifying_Surnames_with_an_MLP.ipynb

```

running_loss = 0.0
running_acc = 0.0
classifier.train()

for batch_index, batch_dict in enumerate(batch_generator):
    # 춘현 과정은 5단계로 이루어집니다

    # 단계 1. 그레이디언트를 0으로 초기화합니다
    optimizer.zero_grad()

    # 단계 2. 출력을 계산합니다
    y_pred = classifier(batch_dict['x_surname'])

    # 단계 3. 손실을 계산합니다
    loss = loss_func(y_pred, batch_dict['y_nationality'])
    loss_t = loss.item()
    running_loss += (loss_t - running_loss) / (batch_index + 1)

    # 단계 4. 손실을 사용해 그레이디언트를 계산합니다
    loss.backward()

    # 단계 5. 옵티마이저로 가중치를 업데이트합니다
    optimizer.step()

    # 정확도를 계산합니다
    acc_t = compute_accuracy(y_pred, batch_dict['y_nationality'])
    running_acc += (acc_t - running_acc) / (batch_index + 1)

```

4-2_Classifying_Surnames_with_an_MLP.ipynb

```

# 단계 1. 그레이디언트를 0으로 초기화합니다
optimizer.zero_grad()

# 단계 2. 출력을 계산합니다
y_pred = classifier(batch_dict['x_surname'])

# 단계 3. 손실을 계산합니다
loss = loss_func(y_pred, batch_dict['y_nationality'])
loss_t = loss.item()
running_loss += (loss_t - running_loss) / (batch_index + 1)

# 단계 4. 손실을 사용해 그레이디언트를 계산합니다
loss.backward()

# 단계 5. 옵티마이저로 가중치를 업데이트합니다
optimizer.step() #weight 업데이트

# 진행 바 업데이트
train_bar.set_postfix(loss=running_loss, acc=running_acc,
epoch=epoch_index)
train_bar.update()

train_state['train_loss'].append(running_loss)
train_state['train_acc'].append(running_acc)

```

4-2_Classifying_Surnames_with_an_MLP.ipynb

```

optimizer.step() #weight 업데이트
# 진행 바 업데이트
train_bar.set_postfix(loss=running_loss, acc=running_acc,
epoch=epoch_index)
train_bar.update()

train_state['train_loss'].append(running_loss)
train_state['train_acc'].append(running_acc)

# 검증 세트에 대한 순회

# 검증 세트와 배치 제너레이터 준비, 손실과 정확도를 0으로 설정
dataset.set_split('val')
batch_generator = generate_batches(dataset,
batch_size=args.batch_size,
device=args.device)
running_loss = 0.
running_acc = 0.
classifier.eval()

for batch_index, batch_dict in enumerate(batch_generator):
    # 단계 1. 출력을 계산합니다
    y_pred = classifier(batch_dict['x_surname'])

```

검증 세트와 배치 제너레이터 준비, 손실과 정확도를 0으로 설정
dataset.set_split('val')
batch_generator = generate_batches(dataset,
batch_size=args.batch_size,
device=args.device)
running_loss = 0.
running_acc = 0.
classifier.eval()
for batch_index, batch_dict in enumerate(batch_generator):
단계 1. 출력을 계산합니다
y_pred = classifier(batch_dict['x_surname'])
단계 2. 손실을 계산합니다
loss = loss_fn(y_pred, batch_dict['y_nationality'])
loss_t = loss.to("cpu").item()
running_loss += (loss_t - running_loss) / (batch_index + 1)
단계 3. 정확도를 계산합니다
acc_t = compute_accuracy(y_pred, batch_dict['y_nationality'])
running_acc += (acc_t - running_acc) / (batch_index + 1)
val_bar.set_postfix(x=(loss=running_loss, acc=running_acc,
epoch=epoch_index))
val_bar.update()
train_state['val_loss'].append(running_loss)
train_state['val_acc'].append(running_acc)
train_state = update_train_state(args=args, model=classifier,
train_state=train_state)

train 상태 저장용도
early_stopping_step: 0,
early_stopping_best_val': 1e8,
learning_rate: args.learning_rate,
epoch_index: 0,
train_loss: [],
train_acc: [],
val_loss: [],
val_acc: [],
test_loss: -1,
test_acc: -1,
model_filename: args.model_state_file}
def update_train_state(args, model, train_state):
''' 훈련 상태를 업데이트합니다.
Components:
- 조기 종료: 과대 적합 방지
- 모델 체크포인트: 더 나은 모델을 저장합니다
:param args: 메인 매개변수
:param model: 훈련할 모델
:param train_state: 훈련 상태를 담은 딕셔너리

적어도 한 번 모델을 저장합니다
if train_state['epoch_index'] == 0:
torch.save(model.state_dict(), train_state['model_filename'])
train_state['stop_early'] = False
성능이 향상되면 모델을 저장합니다
elif train_state['epoch_index'] > 1:
loss_tm1, loss_t = train_state['val_loss'][-2:]
손실이 나빠지면
if loss_t >= train_state['early_stopping_best_val']:
조기 종료 단계 업데이트
train_state['early_stopping_step'] += 1
손실이 감소하면
else:
최상의 모델 저장
if loss_t < train_state['early_stopping_best_val']: # 모델이 좋아질때만 저장하고싶다. loss가 이전값보다 적으면 저장하겠다
torch.save(model.state_dict(), train_state['model_filename'])
조기 종료 단계 재설정
train_state['early_stopping_step'] = 0
조기 종료 여부 확인
train_state['stop_early'] = w # w는 있어서 한줄내릴려고 쓴거
train_state['early_stopping_step'] >= args.early_stopping_criteria #early_stopping_criteria = 5, 5번보다 여려번 연속으로 loss가 개선되지 않으면, 종료하겠다.
return train_state
def compute_accuracy(y_pred, y_target):
_, y_pred_indices = y_pred.max(dim=1)

4-2_Classifying_Surnames_with_an_MLP.ipynb

```

loss = loss_func(y_pred, batch_dict['y_nationality'])
loss_t = loss.to("cpu").item()
running_loss += (loss_t - running_loss) / (batch_index + 1)

# 단계 3. 정확도를 계산합니다
acc_t = compute_accuracy(y_pred, batch_dict['y_nationality'])
running_acc += (acc_t - running_acc) / (batch_index + 1)
val_bar.set_postfix(loss=running_loss, acc=running_acc,
epoch=epoch_index)
val_bar.update()

train_state['val_loss'].append(running_loss) #중간중간 loss와 accuracy를 넣는다. 이걸보고 중간에 저장할지 학습을 끝낼지 결정
train_state['val_acc'].append(running_acc)

train_state = update_train_state(args=args, model=classifier, #update_train_state는 학습을 끝낼지 결정
train_state=train_state)

scheduler.step(train_state['val_loss'][-1])

if train_state['stop_early']:
    break

train_bar.n = 0
val_bar.n = 0
epoch_bar.update()
except KeyboardInterrupt:
    print("Exiting loop")

```

training routine: 100% 100/100 [03:41<00:00, 1.98s/it]

4-2_Classifying_Surnames_with_an_MLP.ipynb

```

def predict_nationality.surname(classifier, vectorizer):
    """새로운 성씨로 국적 예측하기

    매개변수:
        surname (str): 분류할 성씨
        classifier (SurnameClassifier): 분류기 객체
        vectorizer (SurnameVectorizer): SurnameVectorizer 객체

    반환값:
        가장 가능성이 높은 국적과 확률로 구성된 딕셔너리

    """
    vectorized_surname = vectorizer.vectorize(surname)
    vectorized_surname = torch.tensor(vectorized_surname).view(1, -1)
    result = classifier(vectorized_surname, apply_softmax=True)

    probability_values, indices = result.max(dim=1)
    index = indices.item()

    predicted_nationality = vectorizer.nationality_vocab.lookup_index(index)
    probability_value = probability_values.item()

    return {'nationality': predicted_nationality, 'probability': probability_value}

```

new_surname = input("분류하려는 성씨를 입력하세요: ")
classifier = classifier.to("cpu")
prediction = predict_nationality(new_surname, classifier, vectorizer)

두개를 비교할 때
CrossEntropyLoss 사용 label은 loss 계
모델 예측값
나라가 하나라
label
학습
new
data로
모델 예측값이 label과 비교해서 loss가 적게 되도록 classifier weight update되고 예측값이 변경되면서 loop가 지속된다

4-2_Classifying_Surnames_with_an_MLP.ipynb

```

[22] 얼마나 많은 예측을 보고 싶나요? 5
최상위 5개 예측:
Kim → Korean (p=0.37)
Kim → Japanese (p=0.17)
Kim → Arabic (p=0.11)
Kim → German (p=0.08)
Kim → Czech (p=0.07)

print(len(dataset.train_df))
print(len(dataset.val_df))
print(len(dataset.test_df))

7680
1640
1660

```

def __init__(self, surname_df, vectorizer):
 매개변수:
 surname_df (pandas.DataFrame): 데이터셋
 vectorizer (SurnameVectorizer):
 SurnameVectorizer 객체
 """
 self.surname_df = surname_df #내부에서만
 self._vectorizer = vectorizer

 self.train_df = self.surname_df[self.surname_df.
split=='train']
 self.train_size = len(self.train_df)

 self.val_df = self.surname_df[self.surname_df.
split=='val']
 self.validation_size = len(self.val_df)

 self.test_df = self.surname_df[self.surname_df.
split=='test']
 self.test_size = len(self.test_df)

 self._lookup_dict = {'train': (self.train_df, self.
train_size),
'val': (self.val_df, self.
validation_size),
'test': (self.test_df, self.
test_size)}

두개를 비교할때
CrossEntropyLoss 사용 label은 loss계
모델예측값
나라가하나라
label
학습
test
새로운
data로

train, validation 끝나고 새로운 data로 하는 test

train : 학습하는 전체data
validation
test

모델예측값이 label과 비교해서 |
loss가 적게되도록 classifier weight
update되고 예측값이 변경되면
loss가 저조로 짜이지

```

 4-2_Classifying_Surnames_with_an_MLP.ipynb ☆
 파일 수정 보기 삽입 런타임 도구 도움말 모든 변경사항 히든 저작권 Colab AI
 + 코드 + 텍스트 T4 디스크 RAM Colab AI
 [16] split=train: 99% 119/120 [03:28<00:04, 4.11s/it, acc=51.8, epoch=99, loss=
 3분 split=val: 96% 24/25 [03:28<00:03, 3.73s/it, acc=45.4, epoch=99, loss=
 2b # 가장 좋은 모델을 사용해 테스트 세트의 손실과 정확도를 계산합니다
 classifier.load_state_dict(torch.load(train_state['model_filename']))
 classifier.to(args.device)
 dataset.class_weights = dataset.class_weights.to(args.device)
 loss_func = nn.CrossEntropyLoss(dataset.class_weights)

 dataset.set_split('test')
 batch_generator = generate_batches(dataset,
 batch_size=args.batch_size,
 device=args.device)

 running_loss = 0.
 running_acc = 0.
 classifier.eval()

 for batch_index, batch_dict in enumerate(batch_generator):
 # 출력을 계산합니다
 y_pred = classifier(batch_dict['x_surname'])

 # 손실을 계산합니다
 loss = loss_func(y_pred, batch_dict['y_nationality'])
 loss_t = loss.item()
 running_loss += (loss_t - running_loss) / (batch_index + 1)

 # 정확도를 계산합니다
 acc_t = compute_accuracy(y_pred, batch_dict['y_nationality'])
 running_acc += (acc_t - running_acc) / (batch_index + 1)

 # 정확도를 계산합니다
 acc_t = compute_accuracy(y_pred, batch_dict['y_nationality'])
 running_acc += (acc_t - running_acc) / (batch_index + 1)

```

두개를 비교할때
CrossEntropyLoss 사용 label은 loss계
모델예측값
나라가하나라
label
학습
test
새로운
data로

train : 학습하는 전체data
validation
test

모델예측값이 label과 비교해서 |
loss가 적게되도록 classifier weight
update되고 예측값이 변경되면
loss가 저조로 짜이지

```

 4-2_Classifying_Surnames_with_an_MLP.ipynb ☆
 파일 수정 보기 삽입 런타임 도구 도움말 모든 변경사항 히든 저작권 Colab AI
 + 코드 + 텍스트 T4 디스크 RAM Colab AI
 [x] 주론
 def predict_nationality(surname, classifier, vectorizer):
 """새로운 성씨로 국적 예측하기"""
 하나 data입력시, lee, kim같은것
 모델이 어떻게 예측하는지 보는것
 매개변수:
 surname (str): 분류할 성씨
 classifier (SurnameClassifier): 분류기 객체
 vectorizer (SurnameVectorizer): SurnameVectorizer 객체
 반환값:
 가장 가능성이 높은 국적과 확률로 구성된 딕셔너리
 """
 vectorized_surname = vectorizer.vectorize(surname)
 vectorized_surname = torch.tensor(vectorized_surname).view(1, -1)
 result = classifier(vectorized_surname, apply_softmax=True)

 probability_values, indices = result.max(dim=1)
 index = indices.item()

 predicted_nationality = vectorizer.nationality_vocab.lookup_index(index)
 probability_value = probability_values.item()

 return {'nationality': predicted_nationality, 'probability': probability_value}

 [20] new_surname = input("분류하려는 성씨를 입력하세요: ")
 classifier = classifier.to("cpu")
 prediction = predict_nationality(new_surname, classifier, vectorizer)
 print("{} > {} (p={:0.2f})".format(new_surname,
 prediction['nationality'],
 prediction['probability']),

```

두개를 비교할때
CrossEntropyLoss 사용 label은 loss계
모델예측값
나라가하나라
label
학습
test
새로운
data로

train : 학습하는 전체data
validation
test

모델예측값이 label과 비교해서 |
loss가 적게되도록 classifier weight
update되고 예측값이 변경되면
loss가 저조로 짜이지

```

 4-2_Classifying_Surnames_with_an_MLP.ipynb ☆
 파일 수정 보기 삽입 런타임 도구 도움말 모든 변경사항 히든 저작권 Colab AI
 + 코드 + 텍스트 T4 디스크 RAM Colab AI
 [11] k = int(input("얼마나 많은 예측을 보고 싶나요? "))
 if k > len(vectorizer.nationality_vocab):
 print("한국 전체 국적 개수보다 큼 값을 입력했습니다. 모든 국적에 대한 예측을 반환합니다")
 k = len(vectorizer.nationality_vocab)

 predictions = predict_topk_nationality(new_surname, classifier, vectorizer, k=k)

 print("최상위 {}개 예측:".format(k))
 print("=====")
 for prediction in predictions:
 print("{} > {} (p={:0.2f})".format(new_surname,
 prediction['nationality'],
 prediction['probability']))

 분류하려는 성씨를 입력하세요: Kim
 얼마나 많은 예측을 보고 싶나요? 5
 최상위 5개 예측:
 Kim > Korean (p=0.37)
 Kim > Japanese (p=0.17)
 Kim > Arabic (p=0.11)
 Kim > German (p=0.08)
 Kim > Czech (p=0.07)

 [23] print(len(dataset.train_df))
 print(len(dataset.val_df))
 print(len(dataset.test_df))

 7680
 1640
 1660

```

```

4-2_Classifying_Surnames_with_an_MLP.ipynb ☆
파일 수정 보기 삽입 런타임 도구 도움말 모든 변경사항 히든 저작권
+ 코드 + 텍스트 T4 RAM 디스크 Colab AI
11 ① k = int(input("얼마나 많은 예측을 보고 싶나요? "))
  if k > len(vectorizer.nationality_vocab):
    print("모든 전체 국적 개수보다 큰 값을 입력했습니다. 모든 국적에 대한 예측을 반환합니다")
    k = len(vectorizer.nationality_vocab)

  predictions = predict_topk_nationality(new_surname, classifier, vectorizer, k=k)

  print("최상위 {}개 예측:".format(k))
  print("=====")
  for prediction in predictions:
    print("{} > {} (p={:.2f})".format(prediction['nationality'], prediction['probability']))

분류하려는 성씨를 입력하세요: Kim
얼마나 많은 예측을 보고 싶나요? 5
최상위 5개 예측:
Kim > Korean (p=0.37)
Kim > Japanese (p=0.17)
Kim > Arabic (p=0.11)
Kim > German (p=0.08)
Kim > Czech (p=0.07)

[28] print(len(dataset.train_df))
print(len(dataset.val_df))
print(len(dataset.test_df))

7680
1640
1660

```

두개를 비교할때
CrossEntropyLoss 사용 label은 loss계
모델예측값
나라가 하나라
18개다라
학습
label
test
새로운
data로
train : 학습하는 전체data
validation
test
모델예측값이 label과 비교해서
loss가 적게 되도록 classifier weight
update되고 예측값이 변경되면
loss가 저조로 진입지

원본 코드

https://colab.research.google.com/drive/13lcks_OU47911ewNURTMU9pTrm88MH6a#scrollTo=gGifMv5ONhSw

출처 파이토치로 배우는 자연어 처리

좋아요 6

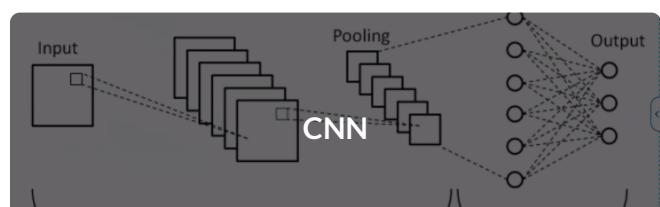
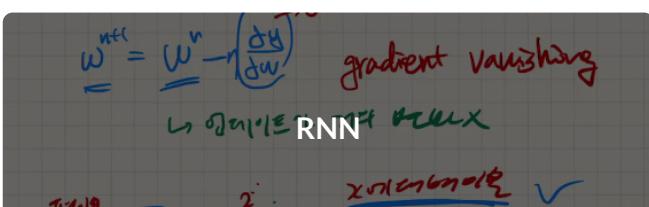
구독하기

'practice_인공지능,머신러닝' 카테고리의 다른 글

RNN (0)	2024.10.16
CNN (1)	2024.10.16
CNN (0)	2024.10.14
감독학습 vs 비지도학습 (Supervised vs Unsupervised): 분류, 군집화 (0)	2024.10.13
데이터 전처리 (Data Pre-processing) (0)	2024.10.13

관련글

관련글 더보기





자연어(NLP)

네이쳐2024 님의 블로그입니다.

[구독하기 +](#)

댓글 3



익명 [답글](#)

비밀댓글입니다.

7시간 전



익명 [답글](#)

비밀댓글입니다.

5시간 전



머니마스터843

포스팅 덕분에 많은 인사이트를 얻었어요! 저희 블로그에도 다양한 정보가 있습니다.

1시간 전 · [답글](#)



이름

비밀번호

내용을 입력하세요.



등록