《 _____ 미적 세특 자료 》

하이에듀

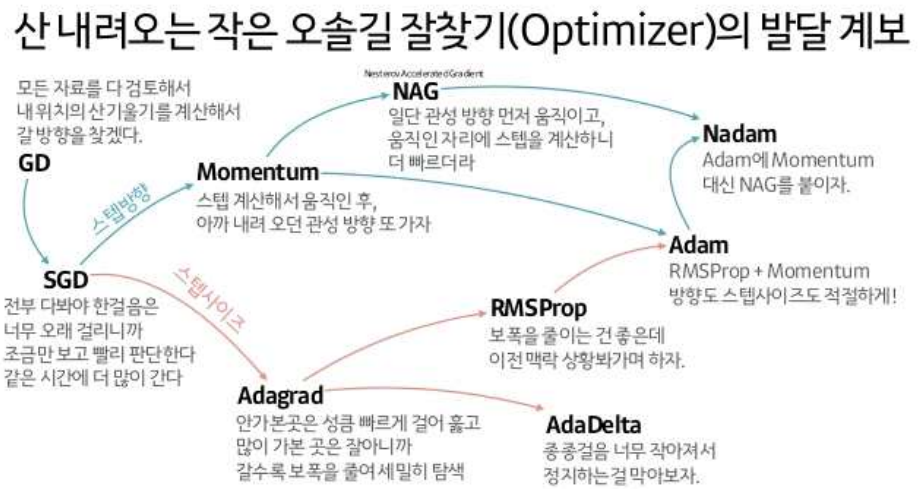
주제	인공신경망 설계에 필요한 미적분

요약

인공신경망은 인간의 신경세포를 모방해서 만든 인공지능 구현 방법 중 하나입니다. 딥러닝의 기반 기술이기도 합니다. 어떠한 데이터를 학습시킬 때 입력 노드에 따른 출력 노드의 목표값이 있고, 인공신경망은 노드 사이의 연결의 가중치를 조정함으로써 계속해서 오차값을 줄여나갑니다. 하지만 오차값을 줄이는 것은 쉽지 않기 때문에 오차함수를 미분하여 얻은 기울기를 이용해 올바른 가중치를 찾아가는 경사하강법을 사용합니다.

책 "신경망 첫걸음" 내용 일부를 첨부하겠습니다. 먼저 읽어보시는 걸 추천합니다. 시그모이드 함수와 경사하강법이 어떻게 활용되는지에 주목해주세요.

읽어보셨나요? 최적화 알고리즘의 종류 중에 경사하강법이 있습니다. 최적화 알고리즘은 각종 공학적 문제의 최적화 변수 값을 각 탐색 범위 내에서 조절함으로써 주어진 비용함수(cost function) 값을 최소화 또는 최대화하는 해를 찾아내는 컴퓨터 연산기법인데, 그 중 경사하강법은 미적분을 이용하는 것이 특징입니다. 사실 경사하강법(GD)도 종류가 많습니다.



링크 [다양한 최적화 알고리즘]에서 많은 AdaGrad, RMSProp, Adam 과 같은 더 많은 미적분 활용 최적화 알고리즘을 접할 수 있으니 단순히 세특에 '경사하강법'만 적는 것 보다는 풍부한 내용이 될 것입니다.

자료 1. 참고자료 [책 "신경망 첫걸음"]

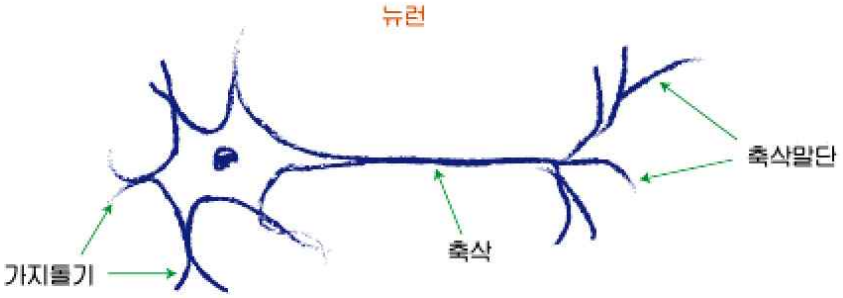
CHAPTER 6

대자연의 컴퓨터, 뉴런

앞에서 말했던 대로 과학자들은 동물 뇌의 신비한 능력에 주목하게 되었습니다. 심지어 뇌 크기가 매우 작은 한 마리 비둘기조차, 엄청난 자원을 가지고 빠른 속도로 연산하는 컴퓨터보다 훨씬 광범위한 능력을 가지고 있기 때문입니다.

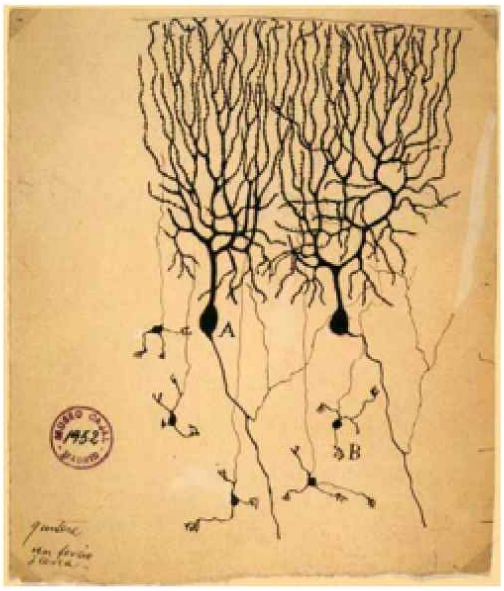
과학자들은 구조적 차이점에 주목하기 시작했습니다. 전통적인 컴퓨터는 데이터를 순차적으로 처리합니다. 또한 이러한 처리는 명확한 체계에 의해 정확히 수행됩니다. 냉철하고 에누리 없는 연산만이 존재할 뿐 어떤 흐릿함이나 애매함도 존재하지 않습니다. 반면 동물의 뇌는 비록 컴퓨터보다 느리게 동작하지만 신호를 순차적이 아니라 병렬적으로 처리하며, 그 처리 특성 중 하나는 바로 불명확성이었습니다.

생물학적 뇌의 기본 단위인 **뉴런(신경세포)**neuron을 보겠습니다.



종류에 상관없이 모든 뉴런은 한쪽 끝에서 다른 쪽 끝으로 전기신호를 전송합니다. 즉 가지돌기dendrite에서 축삭axon을 거쳐 축삭말단axon terminal까지 전송하는 것입니다. 이런 식으로 신호를 하나의 뉴런에서부터 다른 뉴런으로 계속해서 전달합니다. 우리의 몸은 이런 과정을 통해 빛, 소리 등 다양한 감각을 인지할 수 있게 됩니다. 감각 뉴런으로부터 전달된 신호는 신경계를 통해 우리의 뇌로 전달되는데, 뇌 역시 대부분 뉴런으로 구성되어 있습니다.

다음 그림은 1899년 스페인의 한 신경과학자가 스케치한 비둘기 뇌의 뉴런의 모습입니다. 가지돌기와 축삭말단 등 주요 부위를 볼 수 있습니다.



그렇다면, 흥미롭고 복잡한 업무를 수행하기 위해서는 몇 개의 뉴런이 필요할까요?

인간의 뇌는 약 1천억 개의 뉴런을 가지고 있습니다. 초파리는 겨우 10만 개 정도의 뉴런을 갖고 있지만 잘 날아다니고, 음식을 찾아 섭취하고, 위험을 피하는 등 상당히 복잡한 업무를 수행할 수 있습니다. 10만 개 정도라면 오늘날의 컴퓨터가 충분히 복제를 시도할 만한 수준입니다. 심지어 선충은 불과 302개의 뉴런을 갖고 있는데 이는 오늘날의 컴퓨터가 가질 수 있는 자원에 비하면 사실 미미한 숫자이지만, 그럼에도 컴퓨터보다 힘든 업무를 잘 수행합니다.

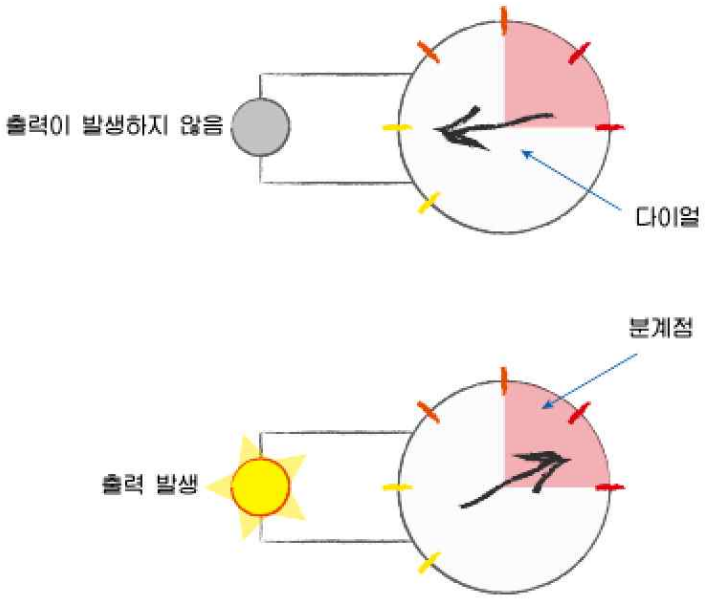
도대체 그 비밀은 무엇일까요? 도대체 생물학적 뇌는 컴퓨터보다 훨씬 느리고 상대적으로 적은 연산 자원을 가지고 있는데도, 어떻게 출중한 능력을 발휘하는 것일까요? 비록 의식과 같은 뇌의 기능은 여전히 미지의 영역이기는 하지만, 문제를 풀기 위해 계산을 수행하는 방법에 관해서라면 뉴런에 대한 연구가 상당히 진척되었습니다.

뉴런의 동작 원리를 알아보겠습니다. 뉴런은 전기 입력을 받아 또 다른 전기신호를 발생시킵니다. 이는 우리가 앞에서 살펴본 분류 또는 예측자에서 입력을 받아 어떤 처리를 해 결과를 출력하는 것과 매우 유사해 보입니다.

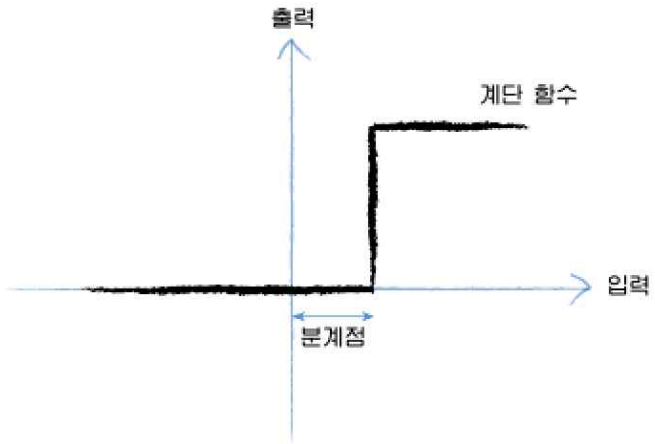
그렇다면 우리는 앞에서 했던 대로 뉴런을 선형함수로 표현할 수 있을까요? 좋은 시도이긴 하지만 그렇게 할 수 없습니다. 생물학적 뉴런에서는 입력을 단순히 선형함수로 처리해 출력을 만들어내지 않습니다. 다시 말해 생물학적 뉴런의 출력은 '출력 값 = (상수 × 입력 값) + (또 다른 상수)'와 같은 형태가 아니라는 것입니다.

과학자들의 관찰에 의하면 뉴런은 입력을 받았을 때 즉시 반응하지는 않습니다. 대신에 입력이 누적되어 어떤 수준으로 커진 경우에만 출력을 하게 됩니다. 즉 입력 값이 어떤 **분계점**threshold에 도달해야 출력이 발생하는 것입니다. 예를 들어 컵에 물을 채울 때 컵을 가득 채워야만 물이 넘치는 것을 연상하면 이해가 될 것

입니다. 뉴런이 미세한 잡음 신호 따위는 전달하지 않고 의미가 있는 신호만 전달하게 된다는 점에서도 직관적으로 이해가 될 것입니다. 다음 그림은 입력이 분계점을 통과할 정도로 충분히 커져야만 출력을 생성해낸다는 것을 다이얼을 돌리는 것에 비유해 표현한 그림입니다.

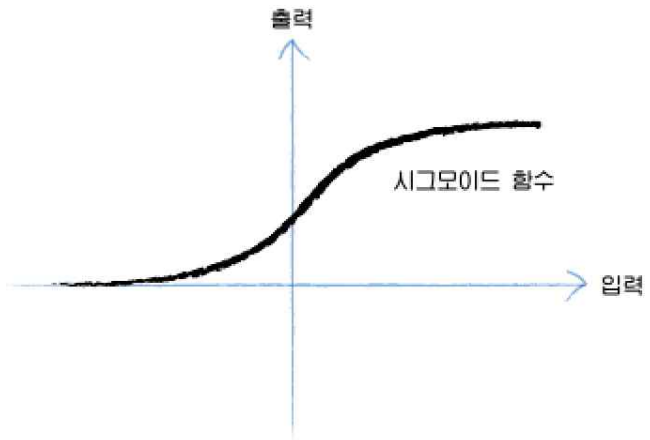


이처럼 입력 신호를 받아 특정 분계점을 넘어서는 경우에 출력 신호를 생성해주는 함수를 **활성화 함수**activation function라고 합니다. 수학적으로 다양한 활성화 함수가 존재합니다. 활성화 함수 중 가장 단순한 형태인 **계단 함수**step function는 다음과 같이 표현됩니다.



그래프에서 보듯이 입력 값이 작은 경우 출력 값은 0이 됩니다. 하지만 일단 입력 값이 분계점 이상이 되면 출력 값은 갑자기 올라가게 됩니다. 이와 같은 인공 뉴런의 반응은 우리 뇌에 있는 생물학적 뉴런의 반응과 유사한 면이 있습니다. 이처럼 입력 값이 분계점에 이르러 출력을 발생시키는 현상을 일컬어 뉴런이 **작동한다**fire라고 말합니다.

이번에는 계단 함수를 좀 더 개선해볼까요? 다음 그래프에 있는 S자 모양의 함수를 **시그모이드 함수**sigmoid function라고 합니다. 시그모이드 함수는 차갑고 딱딱한 계단 함수보다 부드러운 형태를 가지는데, 이는 보다 자연스럽고 현실에 가깝습니다. 가만히 들여다보면 우리가 살고 있는 대자연의 모든 것은 대부분 날카롭고 딱딱한 모서리를 가지지 않는다는 점을 알 수 있습니다.



우리는 앞으로 인공 신경망을 만들 때 이처럼 부드러운 S자 형태를 가지는 시그모이드 함수를 활용할 것입니다. 인공지능 연구자들은 시그모이드 외에 다른 함수들을 이용하기도 하지만, 시그모이드는 단순하면서도 전통적으로 많이 쓰여 온 함수이므로 인공 신경망을 처음 공부하는 우리에게는 안성맞춤이라고 할 수 있습니다.[1]

시그모이드 함수는 때로 **로지스틱 함수**logistic function라고 부르기도 하며, 수식으로는 다음과 같이 표현됩니다.

$$y = \frac{1}{1 + e^{-x}}$$

1 활성화 함수에는 시그모이드 함수 외에도 쌍곡탄젠트(tanh), ReLU, Leaky ReLU, Maxout, ELU 등 많은 종류가 있습니다. 활성화 함수는 2016년 현 시점에도 여전히 활발히 연구가 이루어지고 있는 분야이고, 시그모이드 함수는 전통적으로 가장 기본이라 할 수 있는 활성화 함수로서 학습 용도로 적절합니다. 다만 엄밀히 말하면 현업에서는 몇 가지 문제가 있어 더는 사용이 권장되지 않는 상황입니다. 현 시점에서는 ReLU 함수가 가장 많이 사용된다는 점도 참고하기 바랍니다.

의외로 간단하니 수식을 보고 너무 겁부터 먹을 필요는 없습니다. e라는 기호는 2.71828…이라는 값의 상수로서 수학이나 물리학 분야에서 흔히 사용되는 상수입니다.[2] 값의 마지막 부분을 …으로 표현한 이유는 소수점 이하 부분이 영원히 이어지기 때문입니다. 수학에서는 이런 특징을 가지는 숫자를 일컬어 초월수라고 부르는데요, 너무 깊게 생각할 것 없이 우리는 그냥 e는 2.71828이라고 간주하겠습니다. 입력 값 x에 마이너스를 붙이고 이를 e의 지수로 취합니다. 여기에 1을 더하면 $1 + e^{-x}$입니다. 이에 대해 역수를 취하면 위의 식이 나오는 것입니다. 어려울 것 없죠?

예를 들어 x의 값이 0인 경우 e^{-x}는 1이 되므로 $y = 1 / (1 + 1) = 1/2$이 될 것입니다.

우리가 시그모이드 함수를 활성화 함수로 사용하는 또 다른 이유는 바로 시그모이드 함수가 다른 함수들보다 계산이 매우 편리하다는 점인데, 이에 대해서는 조금 뒤에 알아보겠습니다.

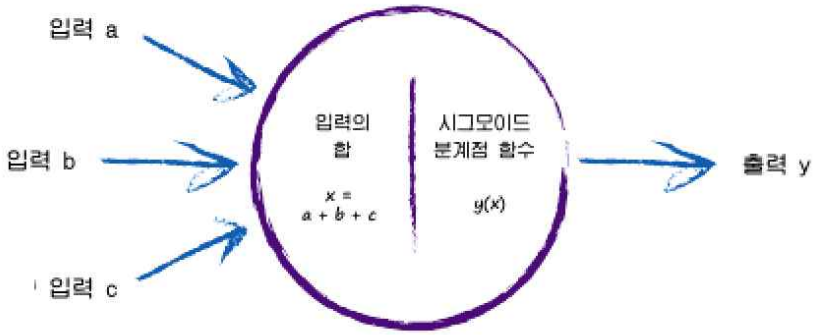
이제 뉴런 이야기로 다시 돌아가서 인공 뉴런을 어떻게 모델화할지 생각해보겠습니다.

여기서 인지해야 할 점은 실제 생물학적 뉴런은 한 개의 입력이 아니라, 여러 개의 입력을 받는다는 점입니다. 이미 우리는 불 논리 기계로 2개의 입력을 받아본 적이 있으므로 이러한 사실이 생소하지는 않습니다.

뉴런은 여러 개의 입력을 받아 어떻게 처리할까요? 뉴런에서는 각각의 입력을 더해줍니다. 그다음 이 합을 시그모이드 함수의 입력 값으로 전달합니다. 시그모이드 함수는 이 입력 값을 이용해 출력을 생성합니다. 이러한 과정 역시 생물학적 뉴런의 동작 원리를 반영하는 것입니다. 이러한 과정은 다음과 같은 그림으로 표현할 수 있습니다.

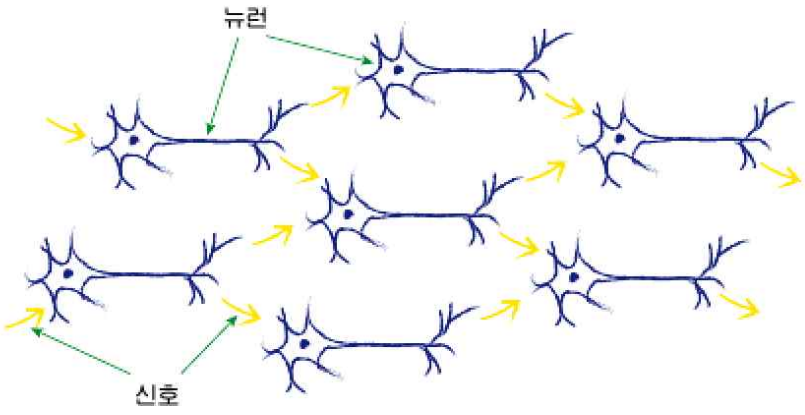
---

2 오일러의 수, 자연상수, 네이피어 상수 등으로 불리기도 합니다.



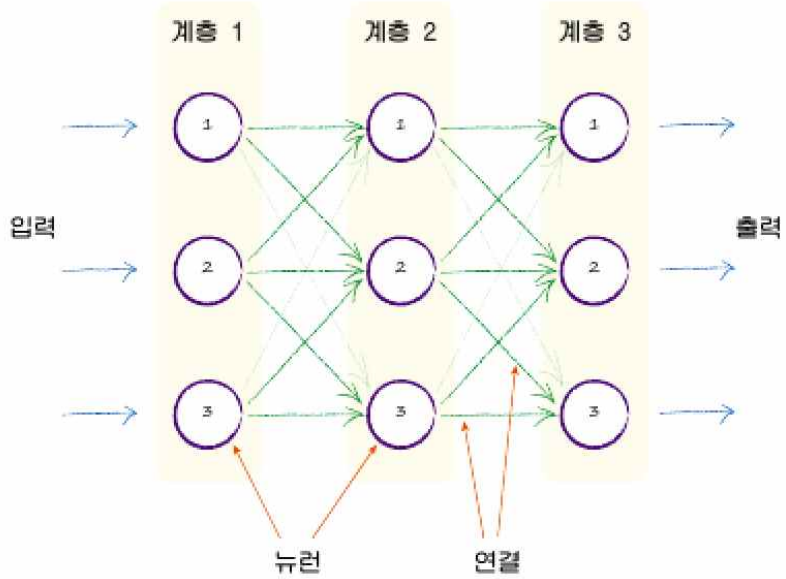
만약 입력 a, 입력 b, 입력 c의 합인 x가 분계점을 넘어설 정도로 충분히 크지 않다면 시그모이드 함수는 아무것도 출력하지 않게 됩니다. 반대로 x가 분계점을 넘으면 시그모이드 함수는 이 뉴런을 작동시킵니다. 흥미로운 점은 나머지 입력 값들이 매우 작다고 하더라도 단지 1개의 입력 값만 충분히 크다면 뉴런은 작동할 수 있다는 점입니다. 이런 점을 볼 때 뉴런은 뭔가 복잡하고 쉽게 잡히지 않을 것 같은 연산을 해낼 수 있을 것 같다는 느낌을 직관적으로 받게 됩니다.

전기신호는 가지돌기에 의해 수집된 다음, 하나로 결합되어 더 강한 전기신호가 됩니다. 이 신호가 분계점을 넘어설 정도로 충분히 크다면 뉴런은 축삭으로 신호를 발사해 축삭말단을 통해 다음 뉴런의 가지돌기로 전달하게 합니다. 이런 식으로 결합된 몇 개의 뉴런을 다음 그림에서 확인할 수 있습니다.



이 그림에서 눈여겨볼 점은 각각의 뉴런이 1개가 아니라 여러 개의 뉴런으로부터 입력을 받는다는 점입니다. 또한 각각의 뉴런은 한번 작동될 때 여러 개의 뉴런으로 신호를 전달하게 됩니다.

이러한 생물학적 뉴런을 인공적으로 모델화하려면 어떻게 할까요? 바로 뉴런을 여러 **계층**layer에 걸쳐 위치시키고, 각각의 뉴런은 직전 계층과 직후 계층에 있는 모든 뉴런들과 상호 연결되어 있는 식으로 표현하면 될 것입니다. 이를 그림으로 표현하면 다음과 같습니다.



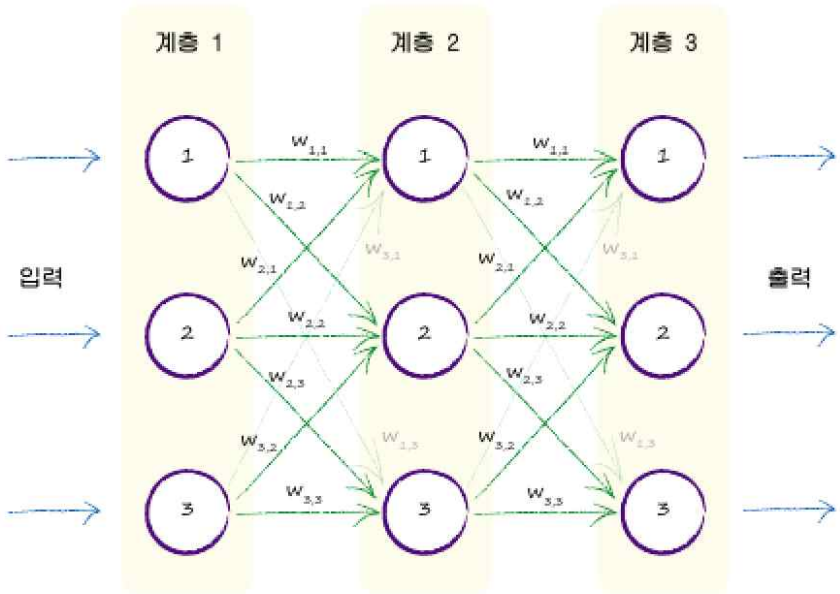
이 그림에는 3개의 계층이 있으며, 각각의 계층에는 뉴런이 3개씩 존재함을 확인할 수 있습니다. 이 각각의 인공 뉴런을 **노드**node라고 부릅니다. 각 노드는 직전 계층과 직후 계층에 존재하는 다른 모든 노드와 연결되어 있다는 점도 확인할 수 있습니다.

그렇다면 이러한 구조에서 과연 어떤 부분이 학습을 하는 것일까요? 학습 데이터를 통해 학습을 진행하게 되면 도대체 무엇을 조정해야 하는 것일까요? 우리

가 앞에서 살펴본 선형 분류자의 기울기처럼 우리가 업데이트해나가야 할 어떤 매개변수가 존재하는 것일까요?

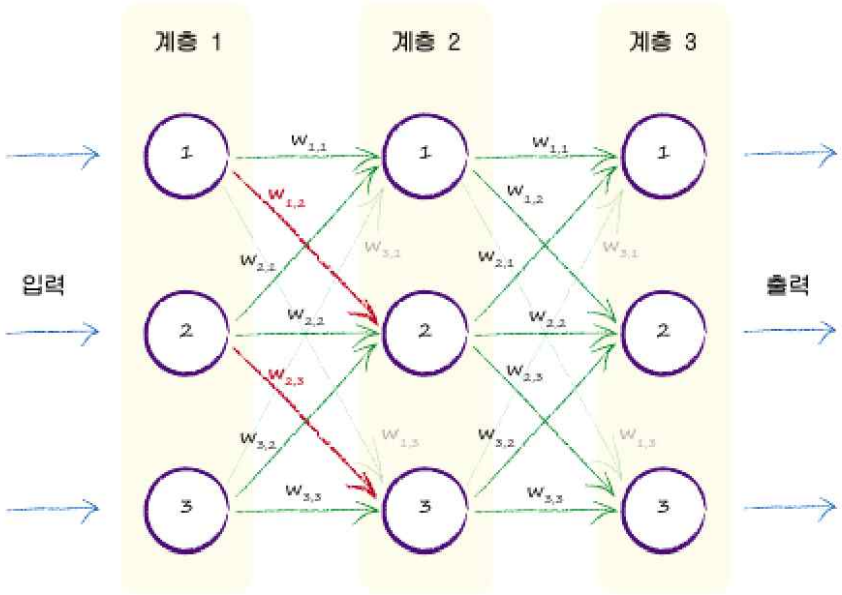
우선 분명한 것 한 가지는 노드 간 연결의 강도를 조정해나가야 한다는 점입니다. 하나의 노드 내에서 입력 값들의 합을 조정하거나 시그모이드 함수의 형태를 조정할 수도 있지만, 이는 단순히 노드 간 연결의 강도를 조정하는 것보다 훨씬 복잡한 작업입니다.

단순한 방법이 제대로 동작한다면 그것을 이용하면 됩니다. 다음 그림은 다시 한번 연결된 노드들을 표현합니다. 하지만 이번에는 각각의 연결에 적용할 **가중치**weight를 함께 표현했습니다. 낮은 가중치는 신호를 약화하며 높은 가중치는 신호를 강화합니다.



가중치 기호 옆에 있는 숫자들에 대해 설명하겠습니다. 예를 들어 $w_{2,3}$은 특정 계층의 노드 2에서 다음 계층의 노드 3으로 전달되는 신호와 관련된 가중치를

의미합니다. 또 $w_{1,2}$는 특정 계층의 노드 1에서 다음 계층의 노드 2로 전달되는 신호를 강화 또는 약화하는 역할을 하는 가중치입니다. 다음 그림에는 첫 번째 계층과 두 번째 계층 사이에서 $w_{1,2}$와 $w_{2,3}$을 강조해서 표시했습니다.



이쯤 되면 여러분은 왜 꼭 하나의 노드가 직전 계층과 직후 계층의 모든 노드와 연결되어야 하는지 의문을 가질 수 있습니다.[3] 사실 꼭 이렇게 모든 노드가 연결되어야 한다는 법은 없습니다. 더 창조적으로 연결할 수도 있습니다. 그럼에도 일반적으로 모든 노드 간에 연결을 하는 이유는, 이렇게 연결해야 프로그램으로 구현하기가 편리하며, 문제를 해결하기 위해 필요로 하는 절대적인 최솟값보다 연결이 몇 개 더 있다고 해서 별문제가 되는 것도 아니기 때문입니다. 꼭 필요한 연결이 아니라고 하면 그 연결은 실제로 학습 과정에서 자동적으로 가중치가 매우 낮아지게 마련입니다.

3 이처럼 모든 노드가 연결되는 것을 영어로는 FC(fully-connected)라고 합니다.

이게 무슨 말이냐고요? 다시 말해 우리의 네트워크가 가중치들의 값을 조정해 가며 출력 값을 개선해가는 과정에서, 일부 가중치들은 0에 가까운 값 또는 심지어는 0이 된다는 의미입니다. 가중치가 0이라는 것은 신호를 전달하지 않는다는 것이므로 결국 이러한 연결은 네트워크에 영향력을 행사할 수 없게 됩니다. 즉, 가중치가 0이라는 것은 신호에 0을 곱한다는 말이므로, 신호가 0이 되는 해당 연결은 사실상 끊어진 것이나 다름없게 됩니다.

핵심 정리

- 생물학적 뇌는 최첨단 컴퓨터에 비해 저장 능력과 연산 속도가 떨어져 보이지만, 하늘을 날고 음식을 찾아내고 언어를 학습하고 포식자로부터 도망가는 등 수준 높고 정교한 업무를 잘 수행합니다.
- 생물학적 뇌는 손상되었거나 완전하지 않은 신호에 대해 전통적인 컴퓨터 시스템에 비해 놀라울 정도로 탄력적인 반응을 합니다.
- 인공 신경망은 상호 연결된 뉴런으로 구성된 생물학적 뇌로부터 영감을 받아 구축되었습니다.

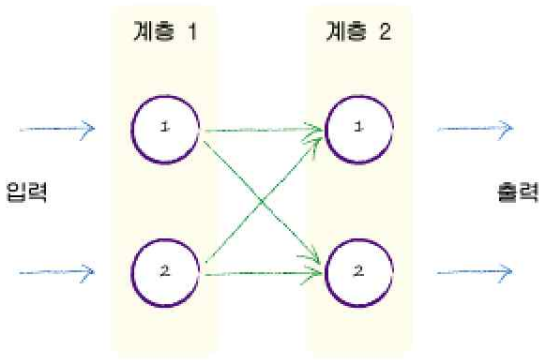
CHAPTER 7

신경망 내의 신호 따라가기

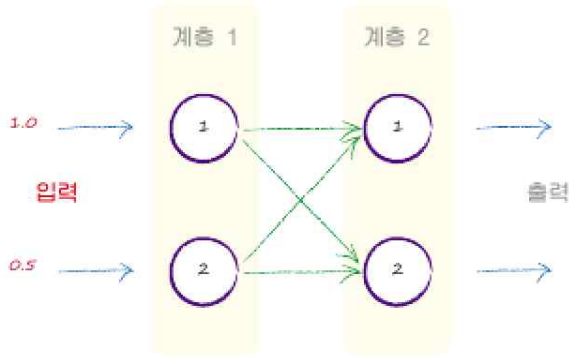
앞 장에서 우리는 뉴런이 3개 계층에 걸쳐 있으며 각각 직전 계층과 직후 계층의 다른 모든 뉴런과 연결되어 있는 그림을 살펴본 바 있습니다.

보기에는 멋진 그림이기는 한데, 신호가 입력되어 계층들을 거쳐서 출력까지 가는 과정에서 우리가 직접 이 신호를 계산해나갈 것을 생각하니 살짝 두려운 생각이 들기도 하는군요.

네, 어려운 작업이기는 합니다. 하지만 신경망을 단계별로 하나하나 그림으로 그려가면서 어떤 일이 벌어지고 있는지 확인해나갈 것이므로 너무 걱정할 필요는 없습니다. 물론 나중에는 컴퓨터를 이용해 이 작업을 할 것이지만, 여기서는 이해를 위해 단계별로 확인해보겠습니다. 더 쉽게 이해하기 위해 다음 그림처럼 2개 계층, 그리고 각 계층에는 2개 뉴런만 존재하는 단순한 신경망을 살펴보며 단계별로 과정을 이해해보겠습니다.



2개의 입력 값이 각각 1.0과 0.5라고 하겠습니다. 이를 그림으로 표현하면 다음과 같습니다.



앞에서 본 것처럼 각 노드는 각 입력 값의 합을 구한 후 활성화 함수를 통해 이를 출력합니다. 활성화 함수로는 앞에서 본 시그모이드 함수를 사용할 것입니다.

시그모이드 함수에서 x는 입력 값의 합을, y는 출력 값을 의미합니다.

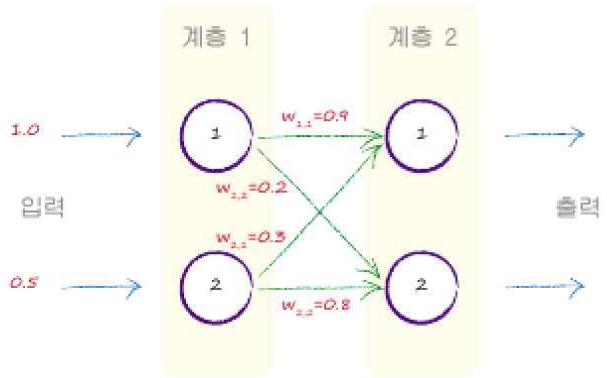
가중치는 어떻게 초기화해야 할까요? 일단 임의의 값으로 초기화해보겠습니다.

- $W_{1,1} = 0.9$
- $W_{1,2} = 0.2$

- $W_{2,1} = 0.3$
- $W_{2,2} = 0.8$

이처럼 가중치의 초기 값을 임의의 값으로 지정하는 것은 나쁜 생각이 결코 아닙니다. 사실 우리는 앞에서 선형 분류자의 기울기도 임의의 값으로 초기화한 바 있습니다. 분류자가 예제들을 통해 학습해나가면서 임의의 값이 점점 개선되어가는 것을 확인했죠. 이는 인공 신경망에서도 동일하게 적용 가능합니다.

지금 우리가 보고 있는 작은 인공 신경망에는 겨우 4개의 가중치가 존재합니다. 2개의 계층 각각에 2개의 노드가 존재하니 가능한 조합이 총 4개가 되는 것이죠. 다음 그림에 이를 표현해봤습니다.

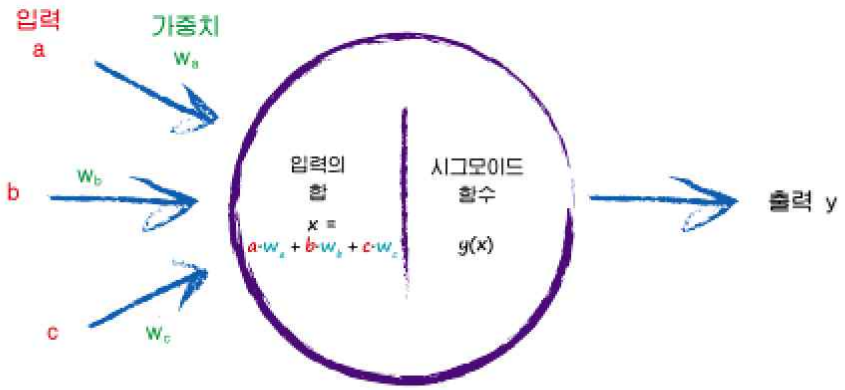


이제 계산을 시작해보겠습니다!

첫 번째 계층은 입력 계층이므로 입력 신호 값을 표현하는 것 외에는 어떤 작업도 필요하지 않습니다. 다시 말해서 입력 노드에서는 입력 값에 활성화 함수를 적용하지 않습니다. 이유를 묻는다면 사실 딱히 답이 있는 것은 아닙니다. 다만 역사적으로 그렇게 해왔다고 설명할 수밖에 없을 것 같습니다. 신경망의 첫 번째 계층은 입력 계층이며, 입력 계층이 하는 일은 그저 입력 값을 표시하는 것뿐입니다. 이렇게만 알면 됩니다.

입력 계층인 계층 1은 어떤 계산도 할 필요 없이 쉽게 끝났습니다.

다음으로 계층 2에서는 계산을 수행할 것입니다. 우리는 계층 2에 존재하는 각각의 노드로 들어오는 입력 값들의 합에 대해 작업을 해야 합니다. 시그모이드 함수에서 바로 x가 해당 노드로 들어온 입력 값들에 가중치를 곱한 값들의 합입니다. 다음 그림은 앞 장에서 봤던 시그모이드 함수 그림과 유사하지만, 들어오는 신호를 조정하기 위해 가중치가 추가되어 있는 것을 확인할 수 있습니다.



우선 계층 2의 노드 1을 보겠습니다. 입력 계층의 2개 노드가 모두 연결되어 있습니다. 입력 노드의 값은 각각 1.0과 0.5입니다. 첫 번째 노드로부터의 연결은 가중치가 0.9이며, 두 번째 노드로부터의 연결은 가중치가 0.3입니다. 따라서 입력 값들에 대해 가중치를 곱하고 합계를 낸 값은 다음과 같이 계산할 수 있습니다.

x = (노드 1의 출력 값 * 가중치) + (노드 2의 출력 값 * 가중치)
x = (1.0 * 0.9) + (0.5 * 0.3) = 0.9 + 0.15 = 1.05

이를 입력의 합이라고 부르며 이렇게 입력의 합을 구하는 것을 조합combine한다고 표현하기도 합니다. 우리가 신호를 조정하지 않았다면 1.0 + 0.5 = 1.5라는

단순합을 결과로 얻었을 것입니다. 하지만 신경망에서 학습을 하는 대상은 가중치입니다. 즉 점점 더 나은 결과를 얻기 위해 가중치의 값을 반복적으로 업데이트해가는 것이 바로 신경망의 학습이므로 가중치가 반드시 포함되어야 합니다.

이렇게 우리는 계층 2의 노드 1으로 들어온 입력의 합을 x = 1.05라고 계산했습니다. 이제 우리는 활성화 함수(시그모이드 함수)에 이 입력 값을 넣어줌으로써 최종적으로 이 노드의 출력 값을 계산할 준비가 되었습니다. 계산기를 이용해서 한번 계산해보기 바랍니다. 출력 값 y = 1 / (1 + 0.3499) = 1 / 1.3499 = 0.7408이 됩니다.

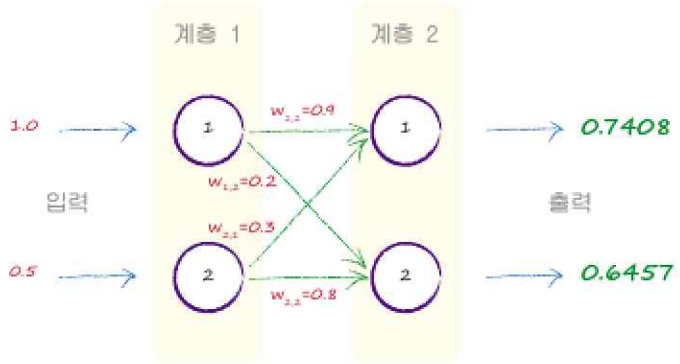
훌륭합니다! 이로써 우리는 2개 노드 중 1개 노드의 출력 값을 실제로 계산해낸 것입니다.

이제 남아 있는 계층 2의 노드 2에 대해 동일한 방식으로 출력 값을 알아내겠습니다. 우선 입력의 합 x를 다음과 같이 계산합니다.

x = (노드 1의 출력 값 * 가중치) + (노드 2의 출력 값 * 가중치)
x = (1.0 * 0.2) + (0.5 * 0.8) = 0.2 + 0.4 = 0.6

이 값을 시그모이드 활성화 함수에 입력 값으로 적용하면 y = 1 / (1 + 0.5488) = 1 / 1.5488 = 0.6457이라는 출력 값을 얻게 됩니다.

이제 우리가 계산한 출력 값을 포함해 그림으로 표현하면 다음과 같습니다.



우리는 지금까지 수작업을 통해 2개의 출력 값을 구했습니다. 매우 단순한 네트워크였기에 망정이지, 대형 네트워크에서 이러한 수작업은 사실 불가능에 가깝습니다. 이런 계산을 빠르고 지루하지 않게 처리할 수 있는 컴퓨터가 있어 다행입니다.

우리에게 컴퓨터가 있기는 하지만 2개보다 훨씬 더 많은 계층을 가지고 각각의 계층에는 4개, 8개, 심지어 수백 개의 노드가 있는 네트워크를 생각해보면, 컴퓨터 프로그램을 짠다고 해도 이런 각각의 계층에 존재하는 각 노드에 대해 계산을 수행하라고 일일이 명령하는 것조차도 엄청난 일이 될 것입니다. 지루하기도 하고, 또 하다 보면 실수도 하게 되겠죠.

다행히도 수많은 계층과 노드를 가지는 복잡한 신경망이라고 하더라도 매우 간결하게 출력 값을 구할 수 있는 수학적 방법이 존재합니다. 간결한 방법은 단지 인간뿐만 아니라 컴퓨터에게도 좋습니다. 인간 입장에서는 해석이 용이해서 좋고, 컴퓨터 입장에서는 훨씬 짧은 명령으로 효율적인 실행이 가능하기 때문입니다.

이처럼 간결한 접근 방법을 제공하는 것이 바로 **행렬**입니다. 다음 장에서 행렬에 대해 살펴보겠습니다.

CHAPTER 14

가중치의 진짜 업데이트

아직 우리는 인공 신경망에서 가중치를 어떻게 업데이트해야 하는가의 핵심적인 질문에 답하지 않았습니다. 지금까지의 과정은 이에 대한 답을 하기 위한 과정이었으며 이제 거의 다 왔습니다. 이 비밀을 풀기 위해 반드시 이해해야 할 핵심 아이디어 한 가지만 더 이해하면 됩니다!

지금까지는 네트워크의 각 계층에 걸쳐 역전파되는 오차를 구해봤습니다. 이처럼 오차를 구하는 이유는 인공 신경망이 보다 나은 답을 출력하게 하기 위해 가중치를 조정해가는 데 지침 역할을 하는 것이 오차이기 때문입니다. 이러한 과정은 이 책의 앞 부분에 나왔던 선형 분류자 예제에서부터 우리가 보아왔던 것입니다.

하지만 신경망에서 노드는 단순한 선형 분류자가 아닙니다. 노드는 입력되는 신호에 가중치를 적용한 후 이의 합을 구하고 다시 여기에 시그모이드 활성화 함수를 적용하는 식으로 좀 더 복잡한 구조를 가집니다. 그렇다면 이처럼 정교한 노드 사이를 연결하는 연결 노드의 가중치를 어떻게 업데이트해야 할까요? 우리는 왜 어떤 끝내주는 대수학 공식을 이용해 가중치를 단번에 구해낼 수 없는 것일까요?

가중치 계산

신경망에는 너무나도 많은 가중치의 조합이 존재합니다. 또한 신호가 여러 개의 계층을 타고 전파되어나갈 때 한 계층을 거칠 때마다 직전 계층의 출력 값이 다음 계층의 입력 값이 되므로 함수의 함수, 함수의 함수의 함수… 같은 식으로 수많은 함수의 조합이 필요하게 됩니다. 따라서 수학 연산의 과정이 너무 복잡하게 되므로 가중치를 한 방에 풀어주는 대수학을 활용할 수 없는 것입니다. 3개 노드로 구성되는 3개의 계층으로 이루어진 간단한 신경망이 있다고 생각해봅시다. 첫 번째 입력 노드와 두 번째 은닉 노드 간의 가중치를 얼마나 바꿔주면 세 번째 출력 노드의 결과 값이 0.5만큼 증가할까요? 이를 쉽게 구할 수는 없습니다. 운이 좋아 구했다고 하더라도 다른 출력 노드의 값을 개선하기 위해 다른 가중치의 값을 변경하는 순간 이 결과는 바로 달라지게 됩니다. 간단한 문제가 아닌 것입니다.

간단한 문제가 아니라는 것을 눈으로 확인하려면 다음의 무시무시한 공식을 보면 됩니다. 이 식은 3개 노드로 구성되는 3개의 계층으로 이루어진 신경망에서 출력 노드의 결과 값을 입력 노드와 가중치의 함수로 표현한 것입니다.

$$O_k = \frac{1}{1 + e^{-\sum_{j=1}^{3}\left(w_{j,k} \cdot \frac{1}{1+e^{-\sum_{i=1}^{3}(w_{i,j} \cdot x_i)}}\right)}}$$

x_i는 노드 i에서의 입력, $w_{i,j}$는 입력 노드 i와 은닉 노드 j를 연결하는 가중치, $w_{j,k}$는 은닉 노드 j와 출력 노드 k를 연결하는 가중치를 의미합니다. Σ(시그마)는 범위 내 모든 값에 대해 합을 구하라는 의미입니다.

네, 더 이상 진행하지 않는 것이 좋을 것 같습니다.

이런 문제를 풀어 영웅이 되기보다는 차라리 가중치들을 랜덤하게 추측해서 구해보는 것은 어떨까요?

우리가 어려운 문제에 봉착했을 때는 이런 생각이 도움이 될 수도 있습니다. 이러한 접근 방식을 **무차별 대입**brute force 방법이라고 합니다. 무차별 대입 방법은 비밀번호를 크래킹cracking하는 데에 사용되기도 합니다. 만약 비밀번호를 짧고 단순한 영어 단어로 설정한다면 무차별 대입 방법에 뚫릴 가능성이 상당히 높습니다. 각 가중치가 −1와 1 사이의 값으로 설정되어 있으며 1,000가지 값 중 하나라고 상상해보겠습니다. 예를 들어 0.501, −0.203, 0.999 등으로 말입니다. 그러면 3개의 노드를 가지는 3개의 계층으로 구성된 신경망에는 총 18개의 가중치가 존재하므로 우리는 총 18,000가지의 경우에 대해 테스트해보면 되는 것입니다. 조금 더 현실적으로 각각의 계층이 500개의 노드를 가진다고 하면 총 가중치의 개수는 50만 개가 될 것이며 우리는 총 5억 가지의 가능성에 대해 가중치를 테스트하게 될 것입니다. 하나의 조합을 연산하는 데 1초가 걸린다고 치고 계산해보면 단 하나의 학습 데이터를 학습하고 가중치를 업데이트하는 데 무려 16년이 걸린다는 말이 됩니다. 학습 데이터가 1,000개라면 16,000년이 걸린다는 뜻이죠!

이를 통해 현실적으로는 제대로 된 신경망에서 무차별 대입이 전혀 실용적이지 못하다는 사실을 알게 되었습니다. 신경망에 계층을 늘리거나, 노드를 늘리거나, 또는 가능한 가중치 값의 개수를 늘리면 상황은 더 어려워질 것입니다.

경사 하강법

그렇다면 가중치를 어떻게 구해야 할까요? 이 문제는 오랜 시간 수학자들을 괴롭혀 왔는데 1960~1970년대에 이르러서야 실용적인 해법이 등장하게 되었습니다. 누가 이러한 해법을 만들어냈는지에 대해서는 다양한 의견이 존재하지만, 우리에게 중요한 점은 이 해법으로 인해 인공 신경망이 폭발적으로 발전하게 되

었다는 것입니다.

그래서 이처럼 어려운 문제를 어떻게 해결한 것일까요? 믿을 수 있을지 모르겠지만 여러분은 이미 이를 스스로 할 수 있는 도구를 가지고 있습니다. 우리는 앞에서 이에 대해 공부한 바 있습니다. 이제 이에 대해 살펴보겠습니다.

우선 현실적으로 **비관주의**를 받아들이는 것부터 시작하겠습니다.

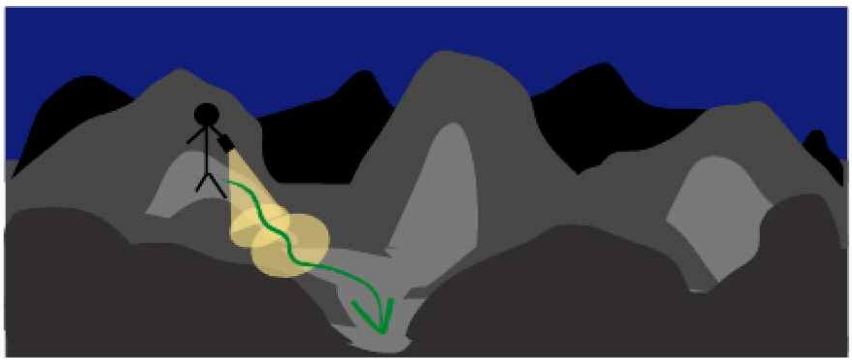
앞에서 본 대로 신경망에서 모든 가중치들이 결과 값을 내는 과정을 수식으로 표현하는 것은 너무 복잡합니다. 가중치의 조합이 너무 많기 때문에 최적의 조합을 찾기 위해 하나씩 테스트해보는 것 자체도 어렵습니다.

비관주의를 받아들여야 할 또 다른 이유가 있습니다. 신경망을 학습시킬 때 학습 데이터가 충분하지 않은 경우가 있습니다. 때로는 학습 데이터 자체에 오류가 있는 경우도 존재합니다. 또 경우에 따라서는 신경망 자체가 문제를 해결하기에 충분한 수의 계층이나 노드를 갖지 않기도 합니다.

이러한 사실은 우리가 이러한 한계점을 명확히 인식하고 현실적인 접근법을 취해야 한다는 점을 의미합니다. 현실적으로 접근하게 되면 비록 수학적으로는 정확하지는 않을지 모르겠지만, 그릇된 이상적인 가정에 기반을 두지는 않을 테니 결과적으로는 더 나은 결과를 얻을 수 있게 될 것입니다.

이제 우리가 등산을 갔다가 불행히도 어두운 밤에 험난한 산속에서 조난을 당했다고 상상해보겠습니다. 커다란 골짜기들이 있고 언덕 중간중간에는 돌출 부위나 틈이 존재합니다. 어둡기 때문에 무엇도 제대로 보이지 않습니다. 우리는 언덕의 한 부분에 서 있고 언덕 아래로 내려가야 하는 상황입니다. 지도는 없지만 다행히 작은 손전등이 하나 있습니다. 하지만 이 손전등은 너무나 작고 성능이 떨어져서 지형 전체를 보기는커녕 불과 몇 미터 앞도 살피기 어려운 수준입니다. 이런 상황에서 우리는 어떻게 행동할까요? 아마 손전등을 이용해 발 주위를 살펴볼 것입니다. 손전등을 이용해 발 주변의 사방을 둘러본 후에 아래로 내려가는 쪽으로 한걸음을 내딛게 될 것입니다. 우리는 비록 지도도 없고 이 장소에

와본 적도 없지만 이런 식으로 한 걸음 한 걸음씩 느리게 언덕을 내려오게 될 것입니다.



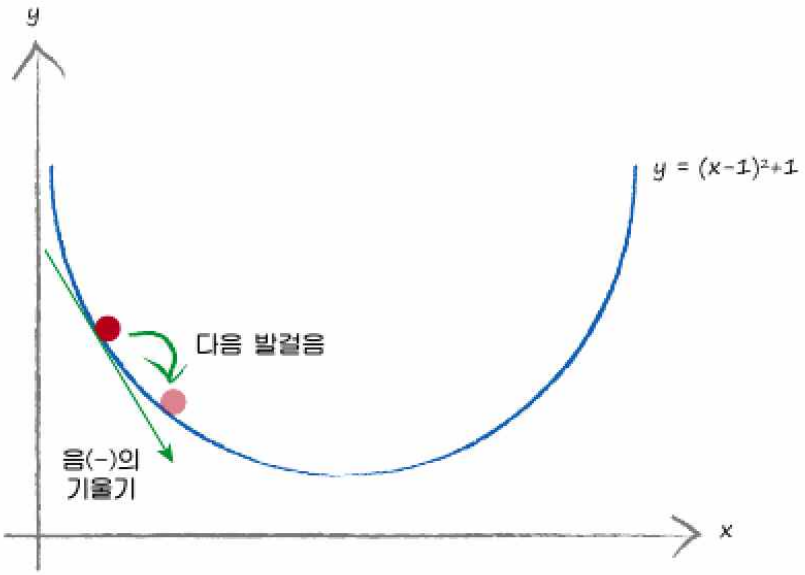
수학에서는 이러한 접근 방식을 **경사 하강법**gradient descent이라고 부릅니다. 우리는 한 걸음을 내딛고 나서 다시 한번 어느 방향이 우리의 목적지(산의 최저점)로 가는 가장 빠른 지름길인지를 파악하기 위해 주변을 살피게 됩니다. 그러고 나서 그 방향으로 다시 한 걸음을 내딛습니다. 구사일생으로 목적지에 도달할 때까지 이 과정을 수없이 반복하게 됩니다. 여기서 경사란 지형의 **기울기**를 의미합니다. 즉 우리는 매 상황에서 경사가 가장 급한 방향으로 한 걸음씩 내딛는 과정을 반복하게 됩니다.

이와 같은 복잡한 지형을 수학에서의 함수라고 상상해보겠습니다. 우리는 복잡한 함수에 대한 이해가 없더라도 경사 하강법을 이용해 최저점을 찾을 수는 있습니다. 함수가 너무 복잡해 대수학을 통해서는 최저점을 찾아내기 어렵다면 경사 하강법을 대신 이용할 수 있는 것입니다. 맞습니다. 경사 하강법에서는 정답에 접근하는 방식으로 단계적 접근 방식을 취하면서 우리의 위치를 조금씩 개선해나가므로 정확한 해답을 얻지 못할 수도 있습니다. 하지만 답을 전혀 얻지 못하는 것보다는 훨씬 낫습니다. 실제 최저점을 목표로 해서 만족스러운 최종 정확도에 이르는 순간까지 아주 작은 한 걸음일지라도 계속해서 답을 다듬어가게 됩니다.

경사 하강법과 신경망의 연결 고리는 무엇일까요? 복잡하고 어려운 함수가 신경망의 오차라고 한다면, 최저점을 찾기 위해 아래로 내려가는 것은 오차를 최소화해나가는 과정이라고 할 수 있겠습니다. 우리는 신경망의 결과 값을 개선해 나갑니다. 이것이 바로 우리가 원하는 것입니다!

경사 하강법의 아이디어를 이해하기 위해 아주 간단한 예제를 보겠습니다.

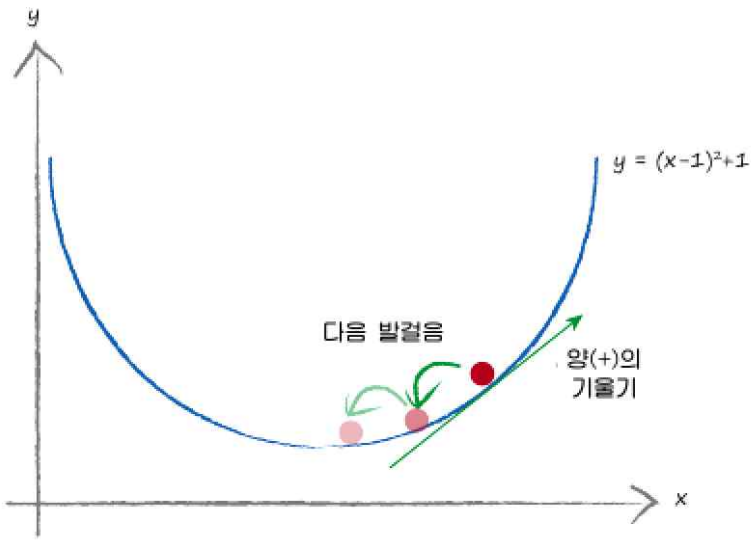
다음 그래프는 함수 $y = (x-1)^2 + 1$을 나타냅니다. 만약 이 함수에서 y가 오차라면 우리는 이 y를 최소화하는 x를 찾고자 할 것입니다. 이 함수가 복잡하고 어려운 함수라고 가정하고 여기에 경사 하강법을 적용해보겠습니다.



위의 그래프에서 빨간색 점이 시작 위치라고 가정하겠습니다. 우리는 이 시작점부터 경사 하강법을 적용하고자 합니다. 마치 등산객처럼 먼저 우리가 서 있는 주위를 둘러보고 어느 쪽으로 가야 아래쪽으로 향할지를 확인하게 됩니다. 화살표로 표시된 기울기는 이 경우 음(-)의 기울기임을 볼 수 있습니다. 우리는 내려가기를 원하므로 x 축 방향으로 오른쪽으로 이동하겠습니다. 즉 x 값을 조금 증가합니다. 이것이 바로 등산객의 첫 번째 발걸음이 됩니다. 보다시피 우리의

위치가 조금 개선되어 최저점에 좀 더 가까운 위치로 이동한 것을 확인할 수 있습니다.

다음 그래프에서는 다른 지점에서 시작한 경우를 살펴보겠습니다.

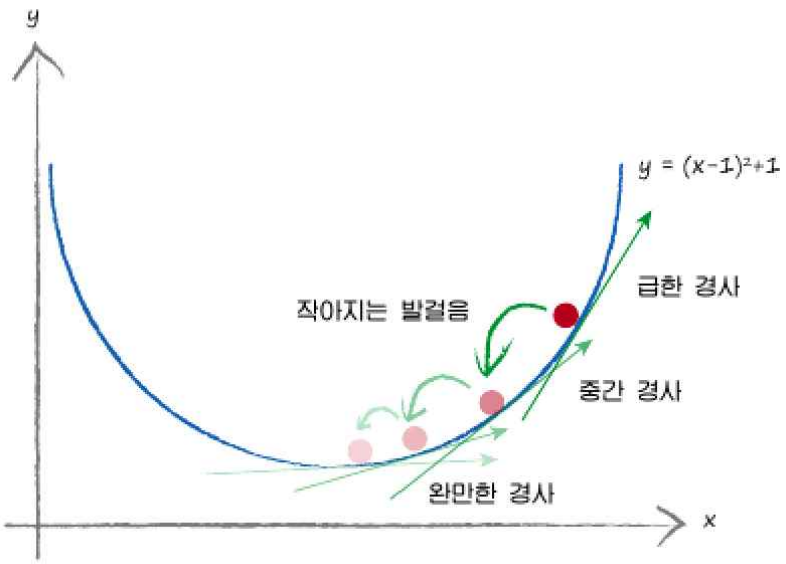


이번에는 우리의 발 아래의 기울기가 양의 기울기이므로 우리는 왼쪽으로 이동하게 됩니다. 다시 말해 x의 값을 약간 감소합니다. 이번에도 우리는 실제 최저점 방향으로 보다 근접하게 되고, 우리의 위치가 개선되었다는 점을 확인할 수 있습니다. 이러한 과정을 계속 반복하다 보면 더 이상 개선의 정도가 너무 작아 무시할 수준 정도가 되어, 최저점에 도달하는 행복한 순간을 맞이하게 될 것입니다.

여기에서 한 가지 주의해야 할 점은 내딛는 걸음의 크기를 잘 조정해야 한다는 것입니다. 걸음 폭이 너무 크면 최저점을 단숨에 지나치게 되며(오버슈팅) 영원히 최저점에 도달하지 못하고 양쪽을 왔다 갔다 하는 결과가 되기 때문입니다. 예를 들어 최저점으로부터 0.5미터 거리까지 근접했는데 한 걸음이 2미터

라면, 우리의 다음 걸음은 최저점을 지나치게 될 것이며 최저점에 도달할 방법이 영원히 없을 것입니다. 만약 발걸음의 크기를 기울기의 크기에 비례하도록 조정해줄 수 있다면 목적지에 거의 도달했을 즈음에는 보다 작은 발걸음을 내디딜 것입니다. 여기에서 가정하는 것은 우리가 최저점에 가까워질수록 기울기는 완만해진다는 사실입니다. 이는 대부분의 매끄럽고 연속적인 함수에서 옳은 가정입니다. 하지만 보통 수학자들이 **불연속**discontinuity이라고 부르는 지그재그 형태의 복잡한 함수에 대해서는 올바른 가정이 아닐 것입니다.

다음 그림은 함수의 기울기가 완만해짐에 따라 발걸음을 조정해나가는 것을 표현합니다. 기울기가 완만해진다는 것은 최저점에 가까워진다는 것을 의미하므로 좋은 신호라고 할 수 있습니다.

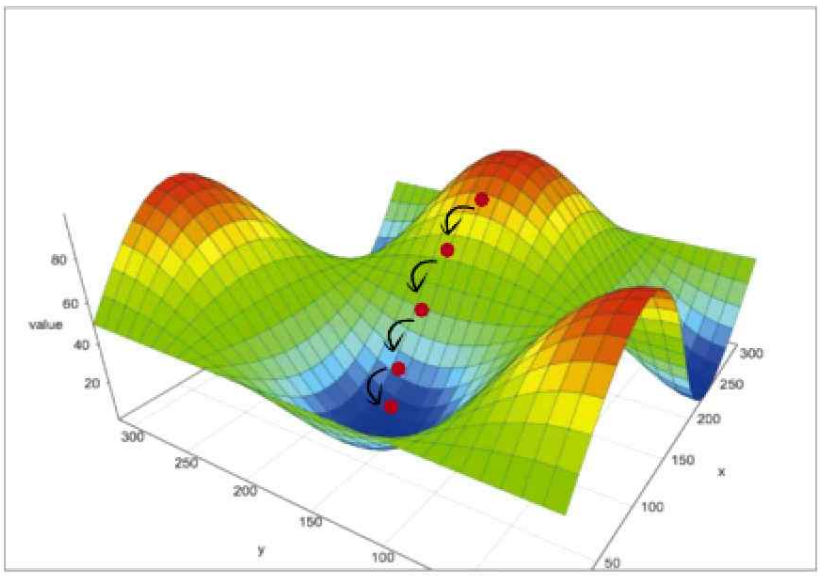


x를 기울기와 반대 방향으로 증가하는 것을 눈치채셨나요? 양의 기울기라면 x 값을 감소하고, 음의 기울기이면 x 값을 증가합니다. 그래프를 보면 명확하지만, 이 사실을 잊고 길을 잃기 쉽습니다.

경사 하강법을 사용할 때 우리는 $y = (x - 1)^2 + 1$이 복잡하고 어려운 함수라고 가정했기 때문에 대수학을 이용해 한 번에 최저점을 구하지 않았습니다. 비록 우리가 수학적 접근을 통해 기울기의 값을 정확히 구한 것은 아니지만 우리는 올바른 방향을 예측해 그 방향으로 이동할 수 있었습니다.

이러한 접근 방법이 빛을 발하는 경우는 바로 많은 매개변수를 가지는 함수에 대응해야 하는 경우입니다. 즉 y의 값이 x라는 한 변수에 의해 결정되는 경우가 아니라 a, b, c, d, e, f 등 여러 변수에 의해 결정되는 경우입니다. 생각해보면 신경망의 결과 값 및 오차 값은 바로 이처럼 많은 가중치 매개변수에 의해 결정됩니다. 보통 수백 개가 넘습니다.

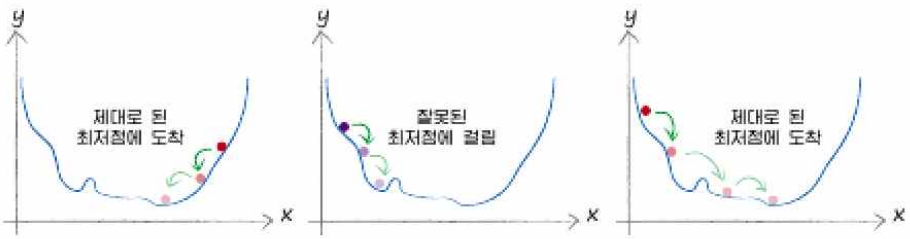
다음 그림은 2개의 매개변수에 의해 결정되는 조금 더 복잡한 함수의 예를 들어 경사 하강법을 보여줍니다. 2개의 매개변수를 가지므로 함수의 결과 값은 높이로 표시되어 3차원의 형태를 띠게 됩니다.



이 3차원 표면을 보면, 경사 하강법을 이용해 진행할 때 우리 의도대로 화살표로 표시된 곳이 아니라 엉뚱한 계곡으로 진행될 수도 있지 않을까라는 의문이 생길 수 있습니다. 좀 더 일반적으로 풀어서 이야기하면, 복잡한 함수는 1개의 계곡이 아니라 여러 개의 계곡을 가질 수 있으므로 경사 하강법이 때로는 잘못된 계곡에 빠지고 여기에서 벗어나지 못하는 경우가 있지 않을까라는 의문 말입니다. 잘못된 계곡이란 진정한 최저점이 아닌 계곡을 의미합니다. 그렇습니다. 이런 일은 분명히 발생할 수 있습니다.[1]

이처럼 잘못된 계곡에 빠지는 것을 피하려면 각각 다른 출발점에서 시작해 여러 번 학습하는 방법을 취할 수 있습니다. 출발점이 다르다는 것은 매개변수의 초기 값을 다르게 준다는 것입니다. 즉 인공 신경망에서는 가중치의 초기 값을 다르게 준다는 것을 의미합니다.

다음 그림은 경사 하강법에 의해 최저점으로 접근하는 세 가지 경우를 보여줍니다. 가운데 그림의 경우 잘못된 계곡에 빠진 것을 확인하기 바랍니다.



중간 정리

- **경사 하강법**은 함수의 최저점을 구하기 위한 좋은 접근 방법입니다. 특히 함수가 매우 복잡하고 어려워 대수학을 이용해 수학적 접근 방식으로 풀기 어려울 때도 잘 동작합니다.
- 매개변수가 많아서 다른 접근 방법들이 실패하거나 현실적이지 못한 경우에도 경사 하강법은 잘 동작합니다.
- 데이터가 불완전하거나 함수가 완벽하게 표현되지 못하거나 잘못된 발걸음을 내디딘 경우에도 경사 하강법은 **탄력적으로** 대응합니다.

1 제대로 된 최저점을 global minimum, 잘못된(국소적) 최저점을 local minimum이라고 부르기도 합니다.

여러 가지 오차함수

신경망에서는 수많은 가중치라는 매개변수를 가지는 복잡하고 어려운 함수에 의해 그 결과 값을 얻습니다. 경사 하강법을 신경망에 이용해도 될까요? 물론입니다. 필요한 것은 **오차함수**error function를 올바르게 선택하는 일입니다.

신경망의 결과 함수output function 자체는 오차함수가 아닙니다. 하지만 오차는 학습 목표 값과 실제 값 간의 차이를 의미하므로 우리는 결과 함수를 쉽게 오차함수로 변환할 수 있습니다.

여기에서 주의해야 할 점이 있습니다. 다음 표에는 3개의 출력 노드에 대한 목표 값, 실제 값과 함께, 오차함수로 쓸 후보가 세 가지 있습니다.

실제 결과 값	목표 값	오차 (목표 값 − 실제 값)	오차 |목표 값 − 실제 값|	오차 (목표 값 − 실제 값)^2
0.4	0.5	0.1	0.1	0.01
0.8	0.7	−0.1	0.1	0.01
1.0	1.0	0	0	0
합		0	0.2	0.02

첫 번째 후보의 경우에서 오차함수는 단순히 **(목표 값 − 실제 값)**입니다. 언뜻 보면 충분히 합리적인 오차함수로 보이지만 전체 노드의 오차를 구하기 위해 합을 구해보면 그 값이 0인 것을 확인할 수 있습니다!

어떻게 된 일일까요? 첫 번째 노드와 두 번째 노드는 각각 오차를 가지고 있으므로 이 신경망이 완벽하게 학습되지 않았다는 것은 분명한 사실입니다. 그런데 전체 노드의 오차의 합은 이 신경망에 오차가 없다고 이야기하고 있습니다. 이런 결과가 나오는 이유는 양의 오차와 음의 오차가 서로를 상쇄했기 때문입니다. 이처럼 완벽하게 서로를 상쇄해서 오차의 합이 0으로 나오는 경우는 좀 극단적인 사례이기는 하지만, 그렇지 않더라도 오차가 서로 상쇄된다는 점에서, 이는 전체 오차를 구하는 합리적인 방법이 아니라는 사실을 깨달을 수 있습니다.

두 번째 후보의 경우에는 바로 이 문제점을 없애기 위해 **절댓값**absolute value을 취합니다. 절댓값이란 **| 목표 값 – 실제 값 |**의 형태로 표기하며, 연산 결과가 음수일 경우 양으로 부호를 바꿔주는 역할을 합니다. 이렇게 하면 오차 간에 상쇄되는 일은 없기 때문에 잘 동작합니다. 하지만 이 방법이 그다지 선호되지 않는 이유는, 최저점 근처에 가면 기울기가 연속적이지 않으며 이로 인해 경사 하강법이 잘 작동하지 않을 수 있기 때문입니다. 예를 들어 V자 형태의 계곡에 빠지게 되면 최저점에 가지 못하고 그 주위를 계속 맴도는 경우가 생길 수 있습니다. 최저점 근처에서 기울기가 작아지지 않기 때문에 우리의 발걸음도 작아지지 않아 결국은 오버슈팅하게 될 가능성이 높다는 것입니다.

세 번째 후보는 **(목표 값 – 실제 값)^2** 같은 식으로 두 값의 차이에 제곱을 합니다. 이런 방식을 **제곱오차**squared error 방식이라고 합니다. 이 제곱오차 방식이 절댓값 방식보다 선호되며 많이 사용되는데 그 이유는 다음과 같습니다.

- 오차함수로 제곱오차 방식을 사용하면 경사 하강법의 기울기를 구하는 대수학이 간단해집니다.
- 오차함수가 부드럽고 연속적이므로 경사 하강법이 잘 동작하게 됩니다. 값이 갑자기 상승하거나 빈 틈이 존재하지 않게 됩니다.
- 최저점에 접근함에 따라 경사가 점점 작아지므로 목표물을 오버슈팅할 가능성이 작아집니다.

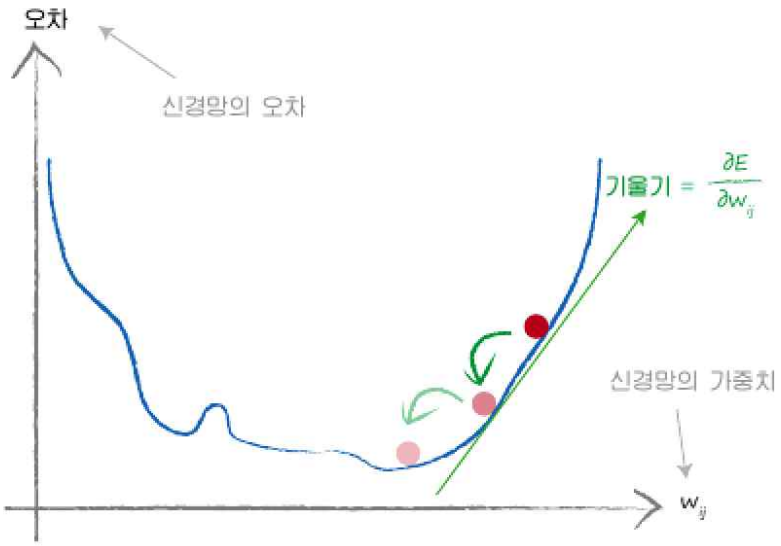
또 다른 후보도 존재할까요? 물론 우리는 더 복잡하고 흥미로운 오차함수를 만들어낼 수도 있습니다. 경우에 따라 어떤 것은 잘 동작하지 않고 어떤 것은 특정 문제에 대해 잘 동작할 것입니다. 하지만 더 이상 일을 복잡하게 만들 가치는 없을 듯합니다.

이제 거의 다 왔습니다!

미분으로 오차함수 구하기

경사 하강법을 사용하려면 가중치에 대한 오차함수의 기울기를 구해야 합니다. 이 작업을 하기 위해서는 **미분**calculus이 필요합니다. 혹시 미분이라는 개념이 생소하면 부록에서 미분 기초에 대해 읽어보기 바랍니다. 미분은 어떤 하나의 변화가 다른 것의 변화에 어떤 영향을 주는지를 구하는 수학적 접근 방법입니다. 예를 들어 스프링에 힘을 얼마나 주느냐에 따라 스프링의 길이가 어떻게 달라지는가 하는 것입니다. 우리의 관심은 신경망 내에서 오차함수가 가중치에 의해 얼마나 영향을 받는가입니다. 다르게 표현하면 '오차는 가중치의 변화에 얼마나 민감한가'라고 할 수도 있겠습니다.

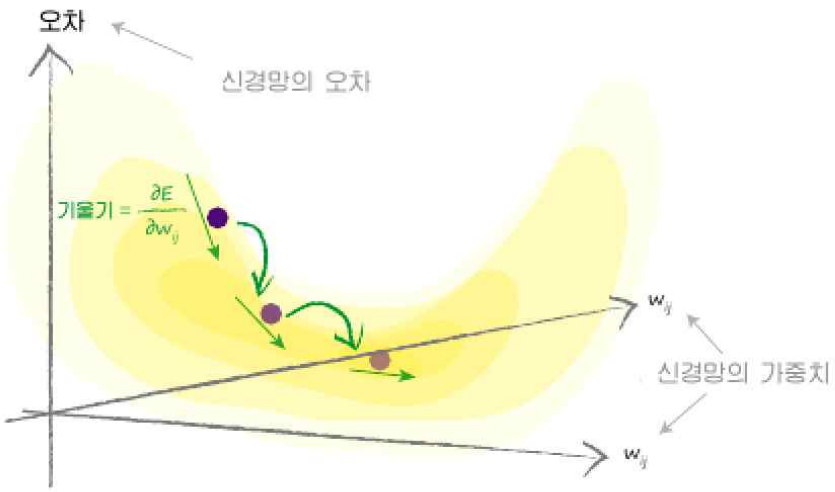
다음 그래프를 보면서 설명하겠습니다.



이 그래프는 앞에서 봤던 것과 거의 유사합니다. 우리가 지금 하는 일도 앞에서 했던 것과 전혀 다르지 않습니다. 다만 이번에 우리가 최소화하고자 하는 함수는 신경망의 오차입니다. 다듬어가고자 하는 매개변수는 가중치입니다. 이 그림

에서는 편의를 위해 1개의 가중치만 표시했지만 사실 신경망에는 수많은 가중치가 있다는 사실을 기억하기 바랍니다.

다음 그래프에는 2개의 가중치가 존재하므로 오차함수는 3차원의 표면으로 표현됩니다. 2개의 가중치의 변화에 따라 3차원의 표면은 변화하게 됩니다. 마치 계곡이 있는 산맥 지형에서 오차를 최소화하려 시도하는 것처럼 보일 것입니다.



많은 매개변수를 가지는 함수는 오차를 시각화하기는 어렵지만, 그 경우에도 경사 하강법을 이용해 최저점을 찾아가는 방식은 여전히 동일하다는 점을 기억하기 바랍니다.

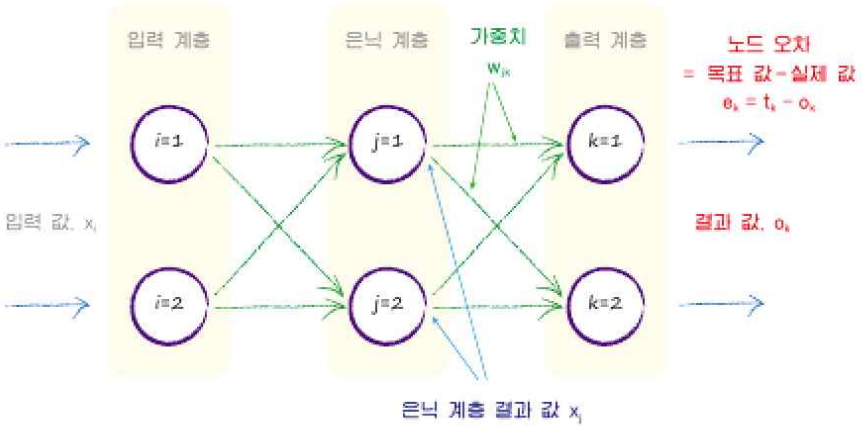
우리가 구하고자 하는 바를 수식으로 표현하면 다음과 같습니다.

$$\frac{\partial E}{\partial w_{jk}}$$

즉, 가중치 w_{jk}의 값이 변화함에 따라 오차 E의 값이 얼마만큼 변하는지입니다. 다시 말해 이 값은 최저점의 방향으로 감소하기 원하는 오차함수의 기울기입니다.

우선 은닉 계층(j)과 최종 출력 계층(k) 사이에 존재하는 연결 가중치에 집중

해서 보겠습니다. 다음 그림에 이 부분을 강조해 표시해뒀습니다. 입력 계층과 은닉 계층 사이의 연결 가중치에 대해서는 조금 후에 살펴보겠습니다.



우리는 앞으로 미분을 할 때 각각의 기호의 의미를 잊지 않기 위해 이 그림을 계속 언급할 것입니다. 미분의 각 단계는 결코 어렵지 않으며 상세하게 살펴볼 것입니다. 게다가 우리는 필요한 모든 개념을 이미 학습했습니다.

우선 오차함수를 전개해보겠습니다. 오차함수는 n개의 노드에 대해 목표 값과 실제 값의 차를 구해 이를 제곱한 다음에 모두 더한 합입니다.[2]

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} \Sigma_n (t_n - o_n)^2$$

단지 오차함수인 E를 풀어 쓴 것뿐입니다.

노드 n에서의 결과 값 o_n은 오직 이와 연결되는 연결 노드로부터만 영향을 받습니다. 다시 말해 노드 k의 결과 값인 o_k는 그와 연결되는 가중치 w_{jk}에 의해서만 영향을 받는다는 뜻입니다.

2 수식에서 t는 target(목표)을 뜻하고, o는 output(결과)을 뜻합니다.

다른 관점에서 보면 노드 k의 결과 값은 가중치 w_{jb}에는 영향을 받지 않는다고 말할 수도 있습니다(k와 b는 같지 않다고 가정). 가중치 w_{jb}는 노드 k와 연결되지 않기 때문입니다. 즉 가중치 w_{jb}는 출력 노드 b와 연결되지 k와는 연결되지 않습니다.

이 의미는 이 합으로부터 w_{jk}의 연결 노드인 o_k 외의 모든 o_n을 제거할 수 있다는 뜻입니다. 성가신 합 기호(Σ) 기호를 통째로 제거할 수 있게 되었습니다! 기억해둘 만한 좋은 방법입니다.

이제 여러분은 오차함수를 구할 때 처음부터 모든 출력 노드에 걸쳐 합을 구하지 않아도 된다는 것을 깨달았습니다. 그 이유는 노드의 결과 값은 오직 연결된 가중치에 의해서만 영향을 받기 때문이라는 것도 알고 있습니다. 사실 많은 책에서 이 사실에 대해 명확히 설명을 안 해주고 그저 오차함수라고 애매하게 표현하는 것을 많이 보곤 했습니다.

이제 수식이 한층 간결해졌습니다.

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} (t_k - o_k)^2$$

이제 미분을 해볼까요? 다시 말하지만 미분이라는 개념이 생소하다면 부록을 참고하기 바랍니다.

t_k는 상수이므로 w_{jk}의 값이 변하더라도 불변입니다. 다시 말해 t_k는 w_{jk}의 함수가 아닙니다. 목표 값이 가중치에 영향을 받는다는 점은 말이 안 되기 때문입니다. 이제 우리가 처리할 것은 w_{jk}에 영향을 받는 o_k 부분뿐입니다.

연쇄 법칙을 이용해 미분 작업을 몇 개의 조각으로 나누어보겠습니다. 연쇄 법칙에 대해서는 부록을 참고하기 바랍니다.

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial o_k} \cdot \frac{\partial o_k}{\partial w_{jk}}$$

이렇게 함으로써 각 항별로 공략이 가능합니다. 첫 번째 항의 미분은 단순히 제곱의 미분이므로 다음과 같이 구할 수 있습니다. $E = (t_k - o_k)^2$였음을 떠올려 봅시다.

$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) \cdot \frac{\partial o_k}{\partial w_{jk}}$$

두 번째 항은 조금 더 생각을 요하기는 하지만 별로 어려울 것 없습니다. o_k는 노드 k의 결과 값으로 입력 신호의 가중치 합에 시그모이드 함수를 적용한 것이라는 사실을 기억할 겁니다. 이를 적용하면 다음과 같이 쓸 수 있습니다.

$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) \cdot \frac{\partial}{\partial w_{jk}} sigmoid\left(\sum_j w_{jk} \cdot o_j\right)$$

o_j는 최종 출력 계층의 출력 값이 아니라, 직전 은닉 계층의 노드로부터의 출력 값입니다.

시그모이드 함수의 미분 값은 무엇일까요? 미분에 대해서는 부록에서 그 원리를 살펴보므로 여기에서는 이미 널리 알려진 결과를 바로 사용하겠습니다. 시그모이드 함수의 미분 값은 다음과 같습니다.

$$\frac{\partial}{\partial x} sigmoid(x) = sigmoid(x)(1 - sigmoid(x))$$

많은 함수가 미분을 하면 보기만 해도 겁이 나는 복잡한 수식으로 변신하고는 합니다. 하지만 시그모이드는 미분한 결과도 식이 매우 간단하며 따라서 활용하기도 편리하다는 장점을 가지고 있습니다. 바로 이런 점이 시그모이드가 신경망

의 활성화 함수로 인기 있는 이유 중 하나입니다.

이제 이 결과를 적용해보겠습니다.

$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) \cdot sigmoid(\Sigma_j w_{jk} \cdot o_j)(1 - sigmoid(\Sigma_j w_{jk} \cdot o_j)) \cdot \frac{\partial}{\partial w_{jk}}(\Sigma_j w_{jk} \cdot o_j)$$

$$= -2(t_k - o_k) \cdot sigmoid(\Sigma_j w_{jk} \cdot o_j)(1 - sigmoid(\Sigma_j w_{jk} \cdot o_j)) \cdot o_j$$

시그모이드 항까지는 이해가 되는데, 그 뒤에 붙은 항은 뭘까요? 이는 시그모이드의 미분 값에 연쇄 법칙이 적용된 결과입니다. 간단히 말해 시그모이드 함수 내의 표현식도 한 번 더 미분되어야 하기 때문입니다. 물론 그 답은 o_j라는 점을 우리는 알고 있습니다.[3]

최종 수식을 적어보기 전에 맨 앞의 2는 제거하겠습니다. 우리는 언덕에서 내려갈 수 있는 오차함수의 기울기 방향에 대해서만 관심이 있으므로 상수는 제거해도 무방합니다. 이 상수가 2든 3이든 100이든 우리가 하나의 상수만 사용하는 한 상관없습니다. 그러므로 간소화를 위해 그냥 2를 없애겠습니다.

이제 '최종 수식'을 적어보겠습니다. 이 수식은 오차함수의 기울기를 표현함으로써 가중치 w_{jk}를 조정해나갈 수 있게 해줍니다.

$$\frac{\partial E}{\partial w_{jk}} = -(t_k - o_k) \cdot sigmoid(\Sigma_j w_{jk} \cdot o_j)(1 - sigmoid(\Sigma_j w_{jk} \cdot o_j)) \cdot o_j$$

우리가 드디어 해냈습니다!

바로 이 식이 우리가 그토록 찾아왔던 마법의 수식입니다. 신경망을 학습시키는

3 연쇄 법칙에 따라 f(g(x))를 미분하면 f'(g(x)) g'(x)가 되고 여기서 f는 시그모이드 함수, g는 $\Sigma\ w_{jk}\ o_j$이기 때문입니다. 여기서 노드 j의 결과 o_j는 앞에서 봤던 것처럼 오직 이와 연결되는 가중치 w_{jk}에 의해서만 영향을 받으므로 합 기호를 뗄 수 있고, 미분을 취하면 o_j만 남습니다.

핵심인 것이죠.

수식을 다시 한번 살펴보겠습니다. 각각의 항을 다른 색상으로 구분했습니다. 첫 번째 항은 우리가 잘 알고 있는 (목표 값 − 실제 값) 오차입니다. 두 번째 항의 시그모이드 안에 있는 합 기호는 사실 최종 계층의 노드로 들어오는 입력 신호에 불과하므로 우리는 이를 더 간단히 i_k라고 표기할 수도 있습니다. 이는 활성화 함수가 적용되기 전의 신호를 의미할 뿐입니다. 마지막 항은 이전 은닉 계층의 노드 j의 결과 값입니다. 수식을 이렇게 나누어 살펴봄으로써 실제로 기울기에 영향을 주는 요소들(궁극적으로 가중치를 최적의 값으로 업데이트하는 요소들)에 대해 감을 잡을 수 있을 것입니다.

최종적으로 단 한 가지 작업이 남아 있습니다. 위의 수식은 은닉 계층과 출력 계층 사이에 있는 가중치를 업데이트하기 위한 것입니다. 이제 입력 계층과 은닉 계층 사이에 있는 가중치들에 대해서도 유사하게 오차 기울기를 찾아보겠습니다.

이를 위해 복잡한 대수학을 할 필요가 없다는 사실은 이미 알고 있습니다. 방금 한 것과 같이 단순하게 이들 가중치에 대한 수식을 만들어보겠습니다. 따라서 이번에는 다음과 같이 진행하겠습니다.

- 이번에는 첫 번째 부분의 오차인 (목표 값 − 실제 값)이 은닉 계층에서 재조합된 역전파 오류가 되게 됩니다. 이 값을 e_j라고 부르겠습니다.
- 두 번째 시그모이드 부분은 동일하지만, 합 부분이 은닉 계층의 노드 j로 들어오는 입력 값에 가중치를 적용한 결과가 됩니다. 이 입력 값을 i_j라고 표기하겠습니다.
- 마지막 부분은 첫 번째 계층의 노드 o_i의 결과 값이 됩니다.

따라서 두 번째 최종 수식, 즉 입력 계층과 은닉 계층 사이의 가중치에 대한 오차함수의 기울기는 다음과 같이 표현할 수 있습니다.

$$\frac{\partial E}{\partial w_{ij}} = -(e_j) \cdot sigmoid(\Sigma_i w_{ij} \cdot o_i)(1 - sigmoid(\Sigma_i w_{ij} \cdot o_i)) \cdot o_i$$

이제 우리는 기울기에 대한 모든 중요한 마법 수식을 구했습니다. 학습 데이터를 이용해 학습을 진행할 때마다 우리는 이를 이용해 가중치를 업데이트해나갈 수 있게 된 것입니다.

학습률

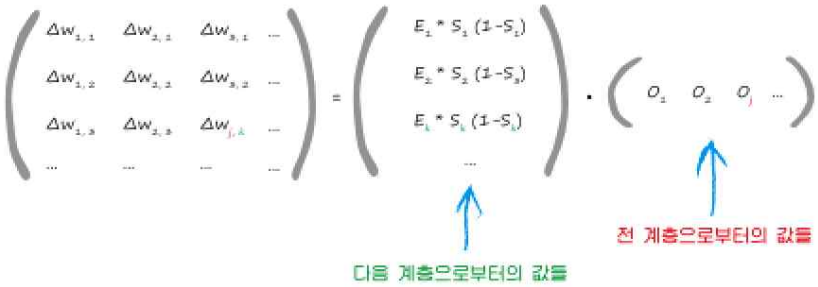
가중치는 기울기와 반대 방향으로 진행된다는 점을 기억하기 바랍니다. 그리고 학습률 인자를 적용함으로써 변화의 정도를 조정하겠습니다. 학습률은 문제에 따라 조금씩 다르게 튜닝해야 합니다. 이는 앞에서 우리가 선형 분류자를 학습시킬 때에도 적합하지 않은 학습 데이터에서 받는 나쁜 영향을 줄이고 가중치가 최저점 근처에서 오버슈팅하는 것을 방지하기 위해 사용한 바 있습니다. 수학적으로는 다음과 같이 표현합니다.

$$new\ w_{jk} = old\ w_{jk} - \alpha \cdot \frac{\partial E}{\partial w_{jk}}$$

업데이트된 새로운 가중치는 방금 구한 오차 기울기에 상수를 곱한 값을 원래 가중치에서 빼줌으로써 구할 수 있다는 뜻입니다. 빼는 이유는 양의 기울기일 경우에는 가중치를 줄이고, 음의 기울기일 경우에는 가중치를 늘리기 위함입니다. 여기서 상수 α는 오버슈팅을 방지하기 위해 변화의 강도를 조정하는 역할을 하며, 이를 바로 **학습률**이라고 합니다.

이 수식은 은닉 계층과 출력 계층 사이의 가중치뿐만 아니라, 입력 계층과 은닉 계층 사이의 가중치에도 동일하게 적용됩니다. 앞에서 구한 두 가지 오차 기울기 수식을 경우에 따라 다르게 사용하면 됩니다.

이러한 계산을 행렬곱으로 어떻게 표현할지 보겠습니다. 가중치 변화 행렬의 각 원소들이 어떻게 구성되는지 적어보겠습니다.



여기에서는 학습률 α를 생략했지만, 학습률은 그저 상수일 뿐 우리가 행렬곱을 수행하는 데에는 아무런 영향을 주지 않기 때문에 문제없습니다.

가중치 변화의 행렬은 한 계층의 노드 j와 다음 계층의 노드 k를 연결하는 가중치 w_{jk}를 조정하는 역할을 합니다. 앞의 수식에서 앞 항은 다음 계층(노드 k)으로부터의 값들을 이용하며, 뒤 항은 전 계층(노드 j)으로부터의 값들을 이용한다는 것을 볼 수 있습니다.

앞의 그림에서 뒤 항은 행이 하나인 행렬로 바로 직전 계층 o_j로부터의 결과 값의 전치 행렬입니다. 수식에서 행렬곱이 이루어지는 하나의 예시로 색상을 구분해 표시해뒀습니다.

따라서 이러한 가중치 업데이트 행렬은 다음과 같이 표현할 수 있습니다. 이제 우리는 컴퓨터를 이용해 이를 돌리기만 하면 될 것 같습니다.

$$\Delta w_{jk} = \alpha \cdot E_k \cdot O_k(1 - O_k) \cdot O_j^T$$

사실 그렇게 복잡한 것은 아닙니다. 사실상 출력 노드들로 이루어진 행렬이기 때문에 수식에서 시그모이드는 뺐습니다.

이게 전부입니다! 수고하셨습니다!

핵심 정리

- 신경망의 오차는 가중치의 함수입니다.
- 신경망을 개선한다는 것은 가중치의 변화를 통해 오차를 줄인다는 뜻입니다.
- 최적의 가중치를 직접 찾는 것은 매우 어렵습니다. 이를 대체하는 접근 방법은 작은 발걸음으로 오차함수를 줄여가면서 반복적으로 가중치를 개선해가는 방법입니다. 각 발걸음은 현재 위치에서 볼 때 가장 급격히 낮아지는 경사의 방향으로 취해집니다. 이런 방법을 **경사 하강법**이라고 합니다.
- 오차 기울기는 미분을 이용해 계산할 수 있으며 알고 보면 별로 어렵지 않습니다.