# DL 복제

**` 딥러닝은 무엇인가요? 딥러닝과 머신러닝의 차이는 무엇이나요?`**

**## 답변 - type: answer**

A: 딥러닝이란 **여러 층을 가진 인공신경망(Artificial Neural Network, ANN)을 사용하여 머신러닝 학습을 수행하는 것**으로, 심층학습이라고도 부른다.
딥러닝은 엄밀히 말하자면 **머신러닝에 포함되는 개념이다.** 따라서 전통적인 머신러닝 기법과 딥러닝 기법의 차이를 설명하고자 한다.

**추가 설명**

### 딥러닝(Deep Learning) 추가 설명

**딥러닝(Deep Learning)이란 여러 층을 가진 인공신경망(Artificial Neural Network, ANN)을 사용하여 머신러닝 학습을 수행하는 것으로 심층학습이라고도 부릅니다.** 따라서 딥러닝은 머신러닝과 전혀 다른 개념이 아니라 머신러닝의 한 종류라고 할 수 있습니다.
기존의 머신러닝에서는 학습하려는 데이터의 여러 특징 중에서 어떤 특징을 추출할지를 사람이 직접 분석하고 판단해야만 했습니다.
하지만 **딥러닝에서는 기계가 자동으로 학습하려는 데이터에서 특징을 추출하여 학습하게 됩니다.**
**이처럼 딥러닝과 머신러닝의 가장 큰 차이점은 바로 기계의 자가 학습 여부로 볼 수 있습니다.**
따라서 딥러닝이란 기계가 자동으로 대규모 데이터에서 중요한 패턴 및 규칙을 학습하고, 이를 토대로 의사결정이나 예측 등을 수행하는 기술로 정의내릴 수 있습니다.

### 인공신경망(Artificial Neural Network, ANN)

**딥러닝에서 가장 기본이 되는 개념은 바로 신경망(Neural Network)입니다.**
신경망이란 인간의 뇌가 가지는 생물학적 특성 중 뉴런의 연결 구조를 가리키며, 이러한 신경망을 본떠 만든 네트워크 구조를 인공신경망(Artificial Neural Network, ANN)이라고 부릅니다.
**인간의 뇌에는 약 1,000억 개의 수많은 뉴런 즉 신경세포가 존재하며, 하나의 뉴런은 다른 뉴런에게서 신호를 받고 또 다른 뉴런에게 신호를 전달하는 단순한 역할만을 수행합니다. 하지만 인간의 뇌는 이러한 수많은 뉴런이 모여 만든 신호의 흐름을 기반으로 다양한 사고를 할 수 있게 되며, 이것을 컴퓨터로 구현하도록 노력한 것이 바로 인공신경망입니다.**

## 퍼셉트론(perceptron)

**퍼셉트론(perceptron)이란 1957년 미국의 심리학자 프랑크 로젠블라트(Frank Rosenblatt)에 의해 고안된 인공신경망 이론을 설명한 최초의 알고리즘이라고 할 수 있습니다.** 로젠블라트는 가장 간단한 퍼셉트론으로 입력층과 출력층만으로 구성된 단층 퍼셉트론(single layer perceptron)의 개념을 제안했습니다.

**단층 퍼셉트론(single layer perceptron)이 동작하는 방식은 다음과 같습니다.**

1. **각 노드의 입력치와 가중치를 서로 곱하여 모두 합한다.**
2. **이렇게 합한 값을 활성화 함수가 가지고 있는 임계치(선택의 기준이 되는 값)와 서로 비교한다.**
3. **만약 그 값이 임계치보다 크면 뉴런은 활성화되고, 만약 임계치보다 작으면 뉴런은 비활성화 된다.**

**이러한 단층 퍼셉트론에서 가중치와 임계치를 적절히 변경하면, 상황에 맞는 적절한 의사결정을 내릴 수 있게 됩니다.**
또한, 단층 퍼셉트론을 여러 개 조합하면 더욱 복잡한 문제도 판단할 수 있게 되며, 이를 다층 퍼셉트론(MultiLayer Perceptron, MLP)이라고 부릅니다.
다층 퍼셉트론은 단층 퍼셉트론을 사용해서는 풀지 못하는 비선형 문제까지도 풀 수 있습니다.
일반적으로 인공신경망이란 이와 같은 다층 퍼셉트론의 조합이라 할 수 있습니다.

다시 돌아와서, 머신러닝과 딥러닝의 가장 큰 차이점은 다음과 같다. **기존 머신러닝에서는 학습하려는 데이터의 여러 특징 중에서 어떤 특징을 추출할지 사람이 직접 분석하고 판단해야하는 반면, 딥러닝에서는 기계가 자동으로 학습하려는 데이터에서 특징을 추출하여 학습하게 된다.** 따라서 특징 추출에 사람이 개입(feature engineering)하면 머신러닝, 개입하지 않으면 딥러닝이다. 또한, 딥러닝은 머신러닝보다 큰 데이터셋과 긴 학습시간이 필요하다. 정형데이터는 주로 머신러닝, 비정형데이터는 주로 딥러닝 방식을 사용한다.

## 인공지능

**인공지능이란** 인간이 가지고 있는 인식, 판단 등의 지적 능력을 모델링하여 컴퓨터에서 구현하는 것이다. 머신러닝, 딥러닝 외에도 다양한 분야가 인공지능 내에 포함된다.

**머신러닝이란** 데이터를 기반으로 패턴을 학습하고 결과를 예측하는 알고리즘 기법이다. 머신러닝은 조건이 복잡하고 규칙이 다양한 경우에, 데이터를 기반으로 일정한/숨겨진 패턴을 찾아내서 문제를 해결한다. 머신러닝의 단점은 데이터에 매우 의존적이라는 것이다. 즉, 좋은 품질의 데이터를 갖추지 못하면 머신러닝 수행결과도 좋지 않다는 것이다.

머신러닝은 아래와 같이 분류된다.

**지도 학습 - [분류, 회귀, 추천시스템, 시각/음성 인지(DL), 텍스트분석/NLP(DL)]**

**비지도 학습 - [클러스터링, 차원축소, 강화학습]**

**`Cost Function과 Activation Function은 무엇인가요?`**

**## 답변 - type: answer**

**A: 1. cost function**

모델은 데이터에 대해 현재 예측을 얼마나 잘하고 있는지 알아야 학습 방향을 어느 방향으로, 얼마나 개선할지 판단할 수 있다.

이 때, 예측 값과 데이터 값의 차이에 대한 함수를 **cost function**(MSE, CrossEntropy 등) 이라고 한다.

**cost function**을 최소화함으로써 모델을 적절한 표현력을 갖추도록 학습시킬 수 있다.

추가로 Cost function (비용 함수)은 기계 학습 및 딥 러닝에서 모델이 얼마나 잘 수행하고 있는지를 평가하는 지표로 사용됩니다.
기본 정의- 비용 함수는 모델이 예측한 출력값과 실제 값(정답) 간의 차이를 수치적으로 표현한 함수입니다.
모델의 오차(error)를 측정하며, 이 값을 최소화하는 것이 학습의 목표입니다.

**2. activation function**

데이터를 예측하기 위해 선형 모델을 사용할 수 있다. 하지만 선형 모델의 경우 복잡한 데이터에 대해서는 적절한 예측을 못한다. 따라서 이를 처리하기 위해 비선형 모델이 필요하다.
선형 모델을 비선형 모델로 만들어주는 역할을 하는 함수가 바로 활성화 함수 activation function(Sigmoid, ReLU 등) 이다.
비선형 함수인 활성화 함수가 선형 함수와 결합됨으로써 선형 모델은 비선형 모델이 된다.
선형 모델은 깊게 쌓을 수 없다. 깊게 쌓아도 하나의 층을 잘 튜닝한 것과 다르지 않기 때문이다.
비선형 모델은 깊게 쌓을 수 있다. 선형으로 만들었다가 비선형으로 만드는 작업을 계속 반복할 수 있기 때문이다. 이로 인해 모델은 복잡한 데이터에 대해 더 표현력이 좋아질 수 있다.
활성화 함수는 입력 값에 대해 더 높게 혹은 더 낮게 만들 수 있기 때문에 활성화 함수라고 불린다.

Activation Function이란?
입력 신호의 총합을 출력신호로 변환하는 함수를 일반적으로 Activation Function이라고 합니다.
활성화(Activate)라는 이름에서 알 수 있듯이 활성화 함수란 입력 신호의 총합이 활성화를 일으키는지 정하는 역할을 한다

6가지 Activation Function

Sigmoid
특징- 출력 값을 0에서 1로 변경해줍니다.(Squashes number to range [0, 1])
가장 많이 사용되었던 활성화 함수라고 합니다.

단점들

Saturation(포화상태)
Sigmoid outputs are not zero-centered
Sigmoid 함수의 두 가지 문제점
우선 Saturation문제가 있습니다. Saturate라는 의미는 Sigmoid 함수의 출력 그래프를 보면 입력 신호의 총합이 크거나 작을 때 기울기가 0에 가까워지는 것을 볼 수 있습니다. 이렇듯 Activation Function의 구간에서 기울기(gradient)가 0에 가까워지는 현상을 Saturated라고 합니다. 이는 Vanishing Gradient문제를 야기합니다.
Sigmoid 문제점 II
두 번째 문제는 Sigmoid 함수가 zero-centerd 하지 않다는 점입니다.

tanh 특징
출력 값은 -1에서 1로 압축시켜줍니다.
zero-centerd 합니다(Nice!, sigmoid가 가졌던 두 번째 문제점을 해결해줍니다)

단점- 여전히 gradient가 죽는 구간이 있습니다. (양수/음수 구간 모두 존재합니다)

ReLU 특징
양의 값에서는 Saturated 되지 않습니다.
계산 효율이 뛰어납니다. sigmid/tanh보다 훨씬 빠릅니다.(6배 정도)
생물학적 타당성도 가장 높은 activation function이라고 합니다.

단점- zero-centerd가 아니라는 문제가 다시 발생했습니다.(non-zero centered)
또한 음수 영역에서 saturated 되는 문제가 다시 발생합니다.
ReLU가 DATA CLOUD로부터 떨어져 있을 때 Dead ReLU가 발생할 수 있습니다.
첫째, 초기화를 잘못한 경우입니다.
가중치 평면이 data cloud에서 멀리 떨어져 있는 경우입니다. 이런 경우 어떤 입력 데이터에서도 activate 되지 않습니다.
더 흔한 경우는 Learning rate가 높은 경우입니다.
가중치 파라미터 업데이트의 learning rate가 높은 경우 ReLU가 데이터의 manifold를 벗어나게 됩니다. 이런 일들은 학습과정에서 흔한 일이고 충분히 발생할 수 있다고 합니다.
그래서 학습이 잘 되다가 죽어버리게 된다고 합니다.

LeakyReLU 특징
ReLU와 유사하지만 negative regime(음의 영역)에서 더 이상 0이 아닙니다.
saturated 되지 않습니다
여전히 계산이 효율적이며 빠릅니다
더 이상 Dead ReLU현상이 없게 됩니다

ELU -ELU는 LU 패밀리입니다(ReLU, LeakyReLU, PReLU...)
하지만 ELU는 zero-mean에 가까운 출력 값을 보입니다.(그래프를 보시면 0을 기준으로 조금 스무스합니다)
앞선 LU패밀리가 zero-mean 출력 값을 갖지 못하는 것에 비해 상당한 이점을 가지고 있습니다.
하지만 Leaky ReLU와 비교해보면 ELU는 negative에서 "기울기"를 가지는 것 대신에 또 다시 "Saturated"되는 문제를 나타냅니다.
ELU는 이런 Saturation이 좀 더 noise에 강인할 수 있다고 주장을 합니다. 이런 deactivation이 좀 더 강인함을 줄 수 있다고 주장합니다. ELU의 논문에는 ELU가 왜 더 뛰어난지 잘 설명을 해준다고 합니다. (시간이 나면 한 번 볼 만하겠군요!)
ELU는 ReLU와 Leaky ReLU의 중간 정도라고 보시면 된다고 합니다. ELU는 Leaky ReLU처럼 zero-mean의 출력을 내지만 Saturation관점에서는 ReLU의 특성도 가지고 있습니다.
PReLU는 negative space에 기울기가 있다는 점에서 Leaky ReLU와 유사한 것을 알 수 있습니다. 다만 여기에서는 기울기가 alpha라는 파라미터로 결정이 됩니다.
alpha를 딱 정해놓는 것이 아니라 backprop으로 학습시키는 파라미터로 만든 것입니다.

Maxout
지금까지 본 활성화 함수들과는 조금 다르게 생겼습니다. 입력을 받아들이는 특정한 기본 형식을 미리 정의하지 않습니다.
대신에 w1에 x를 내적 한 값 + b1과 w2에 x를 내적 한 값 +b2의 최댓값을 사용합니다.
Maxout은 두 값 중 최댓값을 선택합니다.
Maxout은 ReLU와 Leaky ReLU의 좀 더 일반화된 형태입니다. 왜냐하면 Maxout은 두 개의 선형 함수를 취하기 때문입니다. (ReLU, Leaky ReLU를 보시면 선형 함수 두 개의 결합인 것을 알 수 있습니다)
Maxout 또한 선형이기 때문에 Saturation 되지 않으며 gradient가 죽지 않을 것입니다.
여기서 문제점은 뉴런당 파라미터의 수가 두 배가 된다는 것입니다. 이제는 W1과 W2 두 개의 파라미터를 가지고 있어야 합니다.
실제로 가장 많이 쓰는 것은 바로 ReLU입니다.
다만 ReLU를 사용하려면 learning rate을 아주 조심스럽게 결정해야 할 것입니다.

인공신경망에 대한 연구가 한계를 맞게된 첫 과제는 바로 XOR문제였다.

기존의 퍼셉트론은 AND와 OR문제는 해결할 수 있었지만 선형 분류기라는 한계에 의해 XOR과 같은 non-linear한 문제는 해결할 수 없었다.

그리고 이를 해결하기 위해 나온 개념이 hidden layer이다. 그러나 이 hidden layer도 무작정 쌓기만 한다고 해서 퍼셉트론을 선형분류기에서 비선형분류기로 바꿀 수 있는 것은 아니다.

왜냐하면 선형 시스템이 아무리 깊어지더라도 f(ax+by)=af(x) + bf(y)의 성질 때문에 결국 하나의 layer로 깊은 layer를 구현할 수 있기 때문이다.

즉, linear한 연산을 갖는 layer를 수십개 쌓아도 결국 이는 하나의 linear 연산으로 나타낼 수 있다.

<활성화 함수>
이에 대한 해결책이 바로 활성화 함수(activation function)이다.
활성화 함수를 사용하면 입력값에 대한 출력값이 linear하게 나오지 않으므로 선형분류기를 비선형 시스템으로 만들 수 있다.
따라서 MLP(Multiple layer perceptron)는 단지 linear layer를 여러개 쌓는 개념이 아닌 활성화 함수를 이용한 non-linear 시스템을 여러 layer로 쌓는 개념이다.

활성화함수를 사용하면 왜 입력값에 대한 출력값을 비선형으로 만들 수 있는지는 함수의 생김새만 봐도 명확하다.

결론적으로 활성화 함수는 입력값을 non-linear한 방식으로 출력값을 도출하기 위해 사용한다.

이를 통해 linear system을 non-linear한 system으로 바꿀 수 있게 되는 것이다.
그러나, 이렇게 활성화 함수를 이용하여 비선형 시스템인 MLP를 이용하여 XOR는 해결될 수 있지만, MLP의 파라미터 개수가 점점 많아지면서 각각의 weight와 bias를 학습시키는 것이 매우 어려워 다시 한 번 침체기를 겪게되었다.
그리고 이를 해결한 알고리즘이 바로 역전파(Back Propagation)이다.

**`Tensorflow, PyTorch 특징과 차이가 뭘까요?`**

**## 답변 - type: answer**

**A: Tensorflow와 Pytorch의 가장 큰 차이점은 딥러닝을 구현하는 패러다임이 다르다는 것이다. Tensorflow는 Define-and-Run인 반면에, Pytorch는 Define-by-Run이다.**

**Define and Run (Tensorflow)은 코드를 직접 돌리는 환경인 세션을 만들고, placeholder를 선언하고 이것으로 계산 그래프를 만들고(Define), 코드를 실행하는 시점에 데이터를 넣어 실행하는(Run) 방식이다.** 이는 계산 그래프를 명확히 보여주면서 실행시점에 데이터만 바꿔줘도 되는 유연함을 장점으로 갖지만, 그 자체로 비직관적이다.

**Define by Run (PyTorch)은 선언과 동시에 데이터를 집어넣고 세션도 필요없이 돌리면 되기때문에 코드가 간결하고 난이도가 낮은 편이다.**

두 프레임워크 모두 계산 그래프를 정의하고 자동으로 그래디언트를 계산하는 기능이 있다. 하지만 **Tensorflow의 계산 그래프는 정적이고 Pytorch는 동적이다.**

즉 Tensorflow에서는 계산 그래프를 한 번 정의하고 나면 그래프에 들어가는 입력 데이터만 다르게 할 수 있을 뿐 같은 그래프만을 실행할 수 있다. 하지만 PyTorch는 각 순전파마다 새로운 계산 그래프를 정의하여 이용한다.

PyTorch는 Python을 위한 오픈소스 머신 러닝 라이브러리이다. Torch를 기반으로 하며, 자연어 처리와 같은 애플리케이션을 위해 사용된다. GPU사용이 가능하기 때문에 속도가 상당히 빠르다. 아직까지는 Tensorflow의 사용자가 많지만, 비직관적인 구조와 난이도 때문에, Pytorch의 사용자가 늘어나고 있는 추세이다. 이는 Facebook의 인공지능 연구팀이 개발했으며, Uber의 "Pyro"(확률론적 프로그래밍 언어)소프트웨어가 Pytorch를 기반으로 한다

Pytorch는 두 개의 높은 수준의 파이선 패키지 형태로 제공한다.
GPU로 가속화한 행렬 계산 예) NumPy
테이프 기반 자동 미분 시스템을 기반으로 구축된 심층 신경망
Facebook은 PyTorch와 Convolutional Architecture for Fast Feature Embedding (Caffe2)을 모두 운영하고 있지만 비호환성으로 인해 PyTorch 정의 모델을 Caffe2로 변환하거나 그 반대로 변환하는 것이 어렵다. 개신경망 교환(ONNX, Open Neural Network Exchange) 프로젝트는 Facebook과 Microsoft가 프레임워크 간 모델 전환을 위해 2017년 9월 만든 프로젝트다. Caffe2는 2018년 3월 말에 PyTorch으로 합병되었다.

**`Data Normalization은 무엇이고 왜 필요한가요?`**

**## 답변 - type: answer**

**A: Data Normalization(데이터 정규화)이란 feature들의 분포(scale)을 조절하여 균일하게 만드는 방법이다.** 데이터 정규화가 필요한 이유는 **데이터 feature 간 scale 차이가 심하게 날 때, 큰 범위를 가지는 feature(ex. 가격)가 작은 범위를 가지는 feature(ex. 나이)보다 더 강하게 모델에 반영될 수 있기 때문이다.**

**Regularization(정규화, 규제) 란 모델에 제약(penalty)를 주어 모델의 복잡성을 낮추고, 이를 통해 오버피팅을 방지하는 방법이다.** 제약을 사용하면 학습 정확도(train accuracy)는 조금 낮아질 수 있지만, 테스트 정확도(test accuracy)를 높일 수 있다. 정규화에는 Drop out, Early Stopping, Weight decay(Parameter Norm Penalty)와 같은 방법이 존재한다.

**Normalization, Standardization은 모두 데이터의 범위(scale)을 축소하는 방법이다. (re-scaling) 데이터의 범위 재조정이 필요한 이유는 데이터의 범위가 너무 넓은 곳에 퍼져있을 때(scale이 크다면), 데이터셋이 outlier를 지나치게 반영하여 오버피팅이 될 가능성이 높기 때문이다.** 두 방법은 scale 조절 방식에 차이가 존재한다.

**Normalization(정규화) 방법에는 Batch Normalization, Min-Max Normalization 등이 있다.**
**Batch Normalization:** 적용시키려는 레이어의 통계량, 분포를 정규화시키는 방법이다.
**Min-Max Normalization:** 모든 데이터 중에서 가장 작은 값을 0, 가장 큰 값을 1로 두고, 나머지 값들은 비율을 맞춰서 모두 0과 1 사이의 값으로 스케일링하는 방법이다. 모든 feature들의 스케일이 동일하지만, 이상치(outlier)를 잘 처리하지 못한다. 식은 아래와 같다.

**Standardization(표준화) 란 표준화 확률변수를 구하는 방법이다. 이는 z-score를 구하는 방법을 의미한다. z-score normalization이라 불리기도 한다.**
**Z-score:** 관측값이 평균 기준으로 얼마나 떨어져있는지 나타낼 때 사용한다. 각 데이터에서 데이터 전체의 평균을 빼고, 이를 표준편차로 나누는 방식이다. 이상치(outlier)를 잘 처리하지만, 정확히 동일한 척도로 정규화 된 데이터를 생성하지는 않는다.

**`알고있는 Activation Function에 대해 알려주세요. (Sigmoid, ReLU, LeakyReLU, Tanh 등)`**

**## 답변 - type: answer**

A: Sigmoid - sigmoid 함수는 s(z) = frac{1}{1 + e^{-z}로, 입력을 0~1 사이의 값으로 바꿔준다.
**입력 값이 크거나 작을 때 기울기가 0에 가까워지는 saturationsaturation(포화) 문제가 있다.** 이는 gradient vanishing 문제를 야기하므로 요즘에는 활성화 함수로서 잘 사용되지 않는다.
또한 값이 zero-centered 가 아니기 때문에 입력값의 부호에 그대로 영향을 받으므로 경사하강법 과정에서 정확한 방향으로 가지 못하고 지그재그로 움직이는 문제가 있다.

Tanh - tanh 함수는 입력을 -1~1 사이의 값으로 바꿔준다. sigmoid 함수와 마찬가지로 saturationsaturation(포화) 문제가 있다.

ReLU - ReLU 함수는 f(x) = max(0, x)으로, 입력이 양수면 그대로, 음수면 0을 출력한다. 계산 효율과 성능에서 뛰어난 성능을 보여 가장 많이 사용되는 활성화 함수이다. 양의 입력에 대해서는 saturation(포화) 문제가 발생하지 않는다. 음의 입력 값에 대해서는 어떤 업데이트도 되지 않는 Dead ReLU 문제가 발생한다.

**여기서 saturation(포화)란,** 학습에서 작은 기울기는 곧 학습 능력이 제한된다는 것을 의미하고 이를 일컬어 신경망에 포화(Saturation)가 발생했다고 한다. 때문에 포화가 일어나지 않게 하기 위해서는 입력 값을 작게 유지해야 한다.

Leaky ReLU - Leaky ReLU는 f(x) = max(0.01x, x)으로, ReLU 와 마찬가지로 좋은 성능을 유지하면서 음수 입력이 0이 아니게 됨에 따라 Dead ReLU 문제를 해결하였다.

`오버피팅일 경우 어떻게 대처해야 할까요?`

## 답변 - type: answer

A: Early Stopping - training loss는 계속 낮아지더라도 validation loss는 올라가는 시점을 overfitting으로 간주하여 학습을 종료하는 방법이다.

Parameter Norm Penalty / Weight Decay - 비용함수에 제곱을 더하거나(L_2 Regularization) 절댓값을 더해서(L_1 Regularization) weight의 크기에 페널티를 부과하는 방법을 말한다.

Data augmentation - 훈련 데이터의 개수가 적을 때, 데이터에 인위적으로 변화를 주어 훈련 데이터의 수를 늘리는 방법이다.

Noise robustness - 노이즈나 이상치같은 엉뚱한 데이터가 들어와도 흔들리지 않는(robust 한) 모델을 만들기 위해 input data나 weight에 일부러 노이즈를 주는 방법을 말한다.

Label smoothing - 모델이 Ground Truth를 정확하게 예측하지 않아도 되게 만들어 주어 정확하지 않은 학습 데이터셋에 치중되는 경향(overconfident)을 막아주는 방법이다.

**Dropout** - 각 계층 마다 일정 비율의 뉴런을 임의로 정해 drop 시키고 나머지 뉴런만 학습하도록 하는 방법을 말한다. 매 학습마다 drop 되는 뉴런이 달라지기 때문에 서로 다른 모델들을 앙상블 하는 것과 같은 효과가 있다. dropout은 **학습 시에만 적용**하고, 추론 시에는 적용하지 않는다.
**Batch normalization - 활성화함수의 활성화값 또는 출력값을 정규화하는 방법이다. 각 hidden layer에서 정규화를 하면서 입력분포가 일정하게 되고, 이에 따라 Learning rate을 크게 설정해도 괜찮아진다. 결과적으로 학습속도가 빨라지는 효과가 있다.**

**`하이퍼 파라미터는 무엇인가요?`**

**## 답변 - type: answer**

**A: 하이퍼 파라미터(Hyper-parameter)는 모델링할 때, 사용자가 직접 세팅해주는 값**을 뜻한다. **하이퍼 파라미터는 정해진 최적의 값이 없으며, 사용자의 선험적 지식을 기반으로 설정(휴리스틱)한다.** 예를들어 딥러닝의 하이퍼 파라미터에는 학습률, 배치 사이즈 등이 있고, 가중치는 학습 과정에서 바뀌는 값이며 이는 파라미터에 속한다. 하이퍼 파라미터 튜닝 기법에는 Manual Search, Grid Search, Random Search, Bayesian Optimization 등이 있다.

추가로 설명하면,

파라미터는 한국어로 매개변수입니다. 파라미터는 모델 내부에서 결정되는 변수입니다. 또한 그 값은 데이터로부터 결정됩니다. 무슨 말인지 예를 들어 설명해보겠습니다. 한 클래스에 속해 있는 학생들의 키에 대한 정규분포를 그린다고 합시다. 정규분포를 그리면 평균(μ)과 표준편차(σ) 값이 구해집니다. 여기서 평균과 표준편차는 파라미터(parameter)입니다. 파라미터는 데이터를 통해 구해지며 (They are estimated or learned from data), 모델 내부적으로 결정되는 값입니다. 사용자에 의해 조정되지 않습니다. (They are often not set manually by the practitioner)

선형 회귀의 계수도 마찬가지입니다. 수많은 데이터가 있고, 그 데이터에 대해 선형 회귀를 했을 때 계수가 결정됩니다. 이 계수는 사용자가 직접 설정하는 것이 아니라 모델링에 의해 자동으로 결정되는 값입니다. (They are required by the model when making predictions)

하이퍼 파라미터는 모델링할 때 사용자가 직접 세팅해주는 값을 뜻합니다. (They are often specified by the practitioner) learning rate나 서포트 벡터 머신에서의 C, sigma 값, KNN에서의 K값 등등 굉장히 많습니다. 머신러닝 모델을 쓸 때 사용자가 직접 세팅해야 하는 값은 상당히 많습니다. 그 모든 게 다 하이퍼 파라미터입니다. 하지만, 많은 사람들이 그런 값들을 조정할 때 그냥 '모델의 파라미터를 조정한다'라는 표현을 씁니다. 원칙적으로는 '모델의 하이퍼 파라미터를 조정한다'라고 해야 합니다.

하이퍼 파라미터는 정해진 최적의 값이 없습니다. 휴리스틱한 방법이나 경험 법칙(rules of thumb)에 의해 결정하는 경우가 많습니다. (They can often be set using heuristics) 베이지안 옵티미제이션과 같이 자동으로 하이퍼 파라미터를 선택해주는 라이브러리도 있긴 합니다.

파라미터와 하이퍼 파라미터를 구분하는 기준은 사용자가 직접 설정하느냐 아니냐입니다. 사용자가 직접 설정하면 하이퍼 파라미터, 모델 혹은 데이터에 의해 결정되면 파라미터입니다.

### 파라미터 vs 하이퍼 파라미터

파라미터와 하이퍼 파라미터를 구분하는 기준은 사용자가 직접 설정하느냐 아니냐이다. 사용자가 직접 설정하면 하이퍼 파라미터, 모델 혹은 데이터에 의해 결정되면 파라미터이다.

딥러닝에서 하이퍼 파라미터는 학습률, 배치 크기, 은닉층의 개수 등이 있고, 파라미터는 가중치, 편향 등이 있다.

### 용어 정리

**선험적 지식:** 경험하지 않아도 알 수 있는 것을 말한다.
**휴리스틱:** 체계적이면서 합리적인 판단이 굳이 필요하지 않은 상황에서 사람들이 빠르게 사용할 수 있도록, 보다 용이하게 구성된 간편추론의 방법이다. '대충 어림짐작하기', '눈대중으로 맞추기' 등의 방법을 일컫는다.

### `Weight Initialization 방법에 대해 말해주세요. 그리고 무엇을 많이 사용하나요?`

### ## 답변 - type: answer

### A: 초기 가중치 설정 (weight initialization)

딥러닝 학습에 있어 초기 가중치 설정은 매우 중요한 역활을 한다. 가중치를 잘못 설정할 경우 기울기 소실 문제나 표현력의 한계를 갖는 등 여러 문제를 야기하게 된다. 또한 딥러닝의 학습의 문제가 non-convex 이기 때문에 초기값을 잘못 설정할 경우 local minimum에 수렴할 가능성이 커지게 된다.

즉, 딥러닝에서 가중치를 잘 초기화하는 것은 기울기 소실이나 local minima 등의 문제를 야기할 수 있기 때문에 중요하다.

### LeCun Initialization - 딥러닝의 대가 LeCun 교수님이 제시한 초기화 방법으로 들어오는 노드 수에 대해 정규 분포와 균등 분포를 따르는 방법이 있다.

**Xavier Initialization - LeCun 방법과 비슷하지만 들어오는 노드 수와 나가는 노드 수에 의존하고, 적절한 상수값도 발견하여 사용한 방법이다. sigmoid 나 tanh 함수와는 좋은 결과를 보여주지만 ReLU 함수와 사용할 경우 0에 수렴하는 문제가 발생한다. 따라서 sigmoid 나 tanh 함수와 주로 많이 사용한다.**

**He Initialization - ReLU 와 함께 많이 사용되는 방법으로, LeCun 방법과 같지만 상수를 다르게 하였다. 들어오는 노드만 고려한다.**

초기값 설정을 잘못해 문제가 발생하는 경우들을 살펴보자.

1. 초기값을 모두 0으로 설정한 경우
   만약 데이터를 평균 0정도로 정규화시킨다면, 가중치를 0으로 초기화 시킨다는 생각은 꽤 합리적으로 보일 수 있다. 그러나 실제로 0으로 가중치를 초기화 한다면 모든 뉴런들이 같은 값을 나타낼 것이고, 역전파 과정에서 각 가중치의 update가 동일하게 이뤄질 것이다. 이러한 update는 학습을 진행 해도 계속해서 발생할 것이며, 결국 제대로 학습하기 어려울 것이다. 또한 이러한 동일한 update는 여러 층으로 나누는 의미를 상쇄시킨다.
2. 활성화 함수로 sigmoid 사용시 정규 분포 사용
   지난 포스트에서 확인했듯이 sigmoid함수는 input의 절대값이 조금이라도 커지게 되면 미분값이 소실되는 문제가 발생한다는 것을 확인했다.(Post 참고) 이 경우에 평균 0이고 표준편차가 1인 정규분포를 따르도록 가중치를 랜덤하게 초기화 한다고 가정하자. 이 경우에는 표준편차가 크기 때문에 학습을 반복할 수록 가중치 값들이 0,1 로 치우치는 문제 발생한다.(Gradient Vanishing) 이 경우 물론 Activation Function을 바꿈으로써 해결 할 수도 있겠지만, 가중치 초기화를 잘 설정함으로써도 어느정도 해결할 수 있다.
3. 2의 case에서 표준편차를 줄였을 경우
   2의 문제를 확인하고 표준편차가 커 |x|
   값이 커지면서 기울기가 소실되는 문제를 확인했기 때문에, 표준편차를 줄여서 |x| 값을 줄이려는 생각을 가지고 표준편차를 0.01로 설정한다고 가정하자. 이 경우에는 또다른 문제가 발생한다.
   이렇게 표준편차를 적게 하면 층이 깊어질 수록 가중치 값들이 중간 값인 0.5 부근에 몰리는 문제를 확인할 수 있을 것이다.

따라서 이렇게 가중치를 설정하는 것만으로도 학습의 큰영향을 끼친다는 것을 확인할 수 있었다. 그렇다면 더 나은 학습을 위해 가중치를 초기화하는 여러 방법들에 대해서 알아보도록 한다.

1. LeCun Initialization
   LeCun은 지난번 소개한 LeNet의 창시자이며, CNN을 세상에 도입한 사람이라 할 수 있다. 1998년 LeCun은 효과적인 역전파를 위한 논문에서 초기화 방법에 대해서 소개했는데 정규분포를 따르는 방법과 균등분포를 따르는 두가지 방법에 대해서 소개하였다.(LeCun 98, Efficient Backprop)

2. Xavier Initialization
   Xavier Initialization 혹은 Glorot Initialization라고도 불리는 초기화 방법은 이전 노드와 다음 노드의 개수에 의존하는 방법이다. Uniform 분포를 따르는 방법과 Normal분포를 따르는 두가지 방법이 사용된다.(Glorot & Bengio, AISTATS 2010) 구조는 LeCun의 초기화 방법과 유사하지만 다음 층의 노드 수도 사용하고, 많은 연구를 통해 가장 최적화된 상수값 또한 찾아냈다.

3.He Initialization
ReLU를 활성화 함수로 사용 시 Xavier 초기값 설정이 비효율적인 결과를 보이는 것을 확인했는데, 이런 경우 사용하는 초기화 방법을 He initialization이라고 한다. 이 방법 또한 정규분포와 균등분포 두가지 방법이 사용된다.

Bias 초기화
가중치 초기화 뿐만 아니라 편향(bias) 초기값 또한 초기값 설정 또한 중요하다.
보통의 경우에는 Bias는 0으로 초기화 하는 것이 일반적이다. ReLU의 경우 0.01과 같은 작은 값으로 b
를 초기화 하는 것이 좋다는 보고도 있지만 모든 경우는 아니라 일반적으로는 0으로 초기화 하는 것이 효율적이다.

Conclusion
다양한 종류의 초기화 방법에 대해서 알아 보았다. 초기값 설정이 학습과정에 매우 큰 영향을 끼칠 수 있기 때문에 초기화 방법 또한 신중히 선택해야 한다.

Sigmoid, tanh 경우 Xavier 초기화 방법이 효율적이다.
ReLU계의 활성화 함수 사용 시 He 초기화 방법이 효율적이다.
최근의 대부분의 모델에서는 He초기화를 주로 선택한다.
마지막으로, 대부분의 초기화 방법이 Normal Distribution과 Uniform Distribution을 따르는 두가지 방법이 있는데 이에대한 선택 기준에 대해서는 명확한 것이 없다. 하지만 He의 논문의 말을 인용하면,

최근의 Deep CNN 모델들은 주로 Gaussian Distribution을 따르는 가중치 초기화 방법을 사용한다.
따라서 Deep CNN의 경우 보통의 Gaussian 초기화 방법을 사용해 볼 수 있다.하지만 여러 초기화 방법들을 테스트하며 사용하는 것이 가장 좋은 방법일 것이다.

**`볼츠만 머신은 무엇인가요?`**

**## 답변 - type: answer**

**A:** 볼츠만 머신은 가시층(Visible Layer)와 은닉층(Hidden Layer), 총 두 개의 층으로 신경망을 구성하는 방법이다.

볼츠만 머신은 모든 뉴런이 연결되어 있는 완전 그래프 형태이며, 제한된 볼츠만 머신(RBM)에서는 같은 층의 뉴런들은 연결되어 있지 않은 모양이다.

기본적으로 단층구조이며, 확률 모델이다. 분류나 선형 회귀 분석 등에 사용될 수 있다.

특히 DBN(Deep Belief Network)에서는 RBM들을 쌓아올려, 각 볼츠만 머신을 순차적으로 학습시킨다.

Restricted Boltzmann Machine(이하 RBM)은 Generative Model이라고 하는데, ANN, DNN, CNN, RNN 등과 같은 Deterministic Model들과 약간 다른 목표를 갖고 있다.

Deterministic Model들이 타겟과 가설 간의 차이를 줄여서 오차를 줄이는 것이 목표라고 한다면, Generative Model들의 목표는 확률밀도함수를 모델링하는 것이다.

Boltzmann Machine
Boltzmann Machine은 이렇듯 확률분포(정확히는 확률질량함수 혹은 확률밀도함수)를 학습하기 위해 만들어졌다고 할 수 있다.

Boltzmann Machine이 가정하는 것은 "우리가 보고 있는 것들 외에도 보이지 않는 요소들까지 잘 포함시켜 학습할 수 있다면 확률분포를 좀 더 정확하게 알 수 있지 않을까?"라는 것이다.

Restricted Boltzmann Machine
그렇다면 우리의 관심사인 Restricted Boltzmann Machine(RBM)은 뭘까?

RBM은 Boltzmann Machine에서부터 파생되어 나온 것으로 visible unit과 hidden unit에는 내부적인 연결이 없고, visible unit과 hidden unit 간의 연결만이 남아있는 형태이다.

이렇게 RBM을 구성한 것은 여러가지 확률 계산과 관련된 실용적인 이유가 있다.

우선은 visible, hidden layer의 node 간의 내부적인 연결이 없어진 것은 사건 간의 독립성을 가정함으로써 확률분포의 결합을 쉽게 표현하기 위해서이다.

또, visible layer와 hidden layer만을 연결해줌으로써 visible layer의 데이터가 주어졌을 때 hidden layer의 데이터를 계산할 수 있도록 하거나 혹은 hidden layer의 데이터가 주어졌을 때 visible layer의 데이터를 계산할 수 있도록 하는 조건부 확률을 계산할 수 있게 하는 것이다.
즉, $p(h, v)$ 는 계산하기 어려운 반면에 $p(h|v)$ 나 $p(v|h)$ 는 그나마 조금더 계산이 수월하기 때문이다.

이런 이유를 모아 한 마디로 쉽게 말하면 Boltzmann Machine의 계산이 너무 복잡해지니까 이를 편하게 하기 위해 덜 엄격한 모델을 구성한 것으로 볼 수 있다.

Boltzmann Machine에서 RBM과 같은 형태를 구성하게 됨으로써 생기는 독특한 점은 RBM은 Feed-Forward Nerual Network(FFNN) 처럼 학습하게 된다는 점이다.

뒤에 더 설명하겠지만, RBM의 작동방식은 FFNN과 유사하게 forward propagation을 통해 hidden unit의 상태를 결정하고, 다시 hidden unit의 상태로부터 back propagation을 함으로써 visible unit의 상태를 재결정하게 된다.

RBM의 구조와 특성
위에서 서술했듯이 RBM이 무언가를 학습한다는 것은 확률분포를 습득하는 것이다.

**`TF, PyTorch 등을 사용할 때 디버깅 노하우는 무엇이 있나요?`**

**## 답변 - type: answer**

**A:** 오류가 발생하는 곳, 중요한 데이터가 바뀌는 지점을 디버깅 포인트로 두고, 확인하는 방법이 있다. 또, IDE에서 다양한 디버깅 extension을 지원하기 때문에 이를 잘 활용하면 좋은 인사이트를 얻을 수 있다. 예를들어, vs code의 jupyter extension을 사용하면 데이터 프레임, 변수값등을 보기 쉽게 정렬하여 확인할 수 있다.

디버깅 노하우도 중요하지만, 오류에 대한 대처방식을 익히면 좋다. 디버깅 하지 않고 오류에 대처할 수 있으므로, 디버깅 시간을 아껴준다. 예를들어, 딥러닝 학습을 위한 코드를 작성할 때, 가장 많이 발생하는 오류는 CUDA out of memory와 shape 오류이다. (개인적인 의견) out of memory와 같은 오류는 배치 사이즈를 줄인다거나, 입력 데이터의 사이즈를 줄이는 방식으로 해결할 수 있다. shape 오류는 디버깅을 통해서 현재 입력 데이터의 shape, type등을 확인하고, 함수의 파라미터가 요구하는 shape, type에 맞게 변형하는 과정이 필요하다.

추가적으로 딥러닝 디버깅 툴은 아니지만, logging tool로서 tensorboard, wandb 등이 매우 유용하게 사용될 수 있다.

여기서 디버그(Debug)는 프로그래밍 과정중에 발생하는 오류나 비정상적인 연산, 즉 버그를 찾고 수정하는 것이다. 이 과정을 디버깅(Debugging)이라 하기도 한다.

**`뉴럴넷의 가장 큰 단점은 무엇인가? 이를 위해 나온 One-Shot Learning은 무엇인가요?`**

**## 답변 - type: answer**

**A:** 사람은 처음 보는 물건 (새 레이블) 에 대해 조금만 봐도 다른 것과 이 물건을 구분해낼 수 있다. 하지만 뉴럴넷은 이 물건을 구분해내기 위해서는 이 물건에 대한 많은 데이터를 학습해야한다.

**One-shot Learning 은 뉴럴넷도 새로운 레이블을 지닌 데이터가 적을 때 (one-shot 에서는 한 개) 에도 모델이 좋은 성능을 내도록 사용되는 방법이다.**

이를 위해서는 기존에 다른 레이블의 많은 데이터를 학습하여 데이터의 특성을 잘 이해하는 pretrained 모델이 필요하다.

학습된 모델에 새로운 레이블의 데이터 하나 던져 주면 모델은 데이터의 특성에 대한 이해를 바탕으로 이 레이블에 대해서도 이해를 하게 된다.

**` 요즘 sigmoid 보다 ReLU를 많이 쓰는데 그 이유는 무엇인가요?`**

**## 답변 - type: answer**

**A:** sigmoid는 값이 큰 양수일수록 1에, 큰 음수일수록 0에 가까워진다. 반면 ReLU는 값이 양수이면 원래 값을 그대로 가져가고, 음수이면 0이다.

요즘 sigmoid보다 ReLU를 많이 쓰는 가장 큰 이유는 기울기 소실 문제(Gradient Vanishing) 때문이다. 기울기는 연쇄 법칙(Chain Rule)에 의해 국소적 미분값을 누적 곱을 시키는데, sigmoid의 경우 기울기가 항상 0과 1사이의 값이므로 이 값을 연쇄적으로 곱하게 되면 0에 수렴할 수 밖에 없다. 반면 ReLU는 값이 양수일 때, 기울기가 1이므로 연쇄 곱이 1보다 작아지는 것을 어느 정도 막아줄 수 있다.

다만, ReLU는 값이 음수이면, 기울기가 0이기 때문에 일부 뉴런이 죽을 수 있다는 단점이 존재한다. 이를 보완한 활성화 함수로 Leaky ReLU가 있다.

**`Non-Linearity라는 말의 의미와 그 필요성은 무엇인가요?`**

**## 답변 - type: answer**

**A:** 비선형(non-linearity)의 뜻을 알기 위해서는 우선 선형(linearity)가 무엇인지 알아야 한다. 어떤 모델이 선형적(linearity)라고 한다면 그 모델은 변수 x_1, x_2, ... , x_n과 가중치 w_1, w_2, ... , w_n으로 y = w_1 * x_1 + w_2 * x_2 + ... + w_n * x_n으로 표현할 수 있으며, 가산성(Additreivityly)과 동차성(Homogeneity)을 만족해야 한다. 즉, 선형성(linearity) 또는 선형(linear)은 직선처럼 똑바른 도형, 또는 그와 비슷한 성질을 갖는 대상이라는 뜻으로, 이러한 성질을 갖고 있는 변환 등에 대하여 쓰는 용어이다. 함수의 경우, 어떠한 함수가 진행하는 모양이 '직선'이라는 의미로 사용된다. 이러한 개념은 수학, 물리학 등에서 많이 사용된다. 다른 말로 1차라고도 한다. (단어 '1차' 자체는, '선형'을 의미하지 않는 경우도 많다.)

가산성: 임의의 수 x, y에 대해 f(x+y) = f(x) + f(y)가 성립

동차성: 임의의 수 x, alpha에 대해 f(alpha x) = alpha f(x)가 성립

이를 만족하지 못하는 모델을 비선형 관계에 있는 모델이라고 한다.

딥러닝에서 이런 비선형 관계는 활성화 함수(activation function)을 도입함으로써 표현할 수 있다. 그럼 비선형 관계 즉, 활성화 함수가 왜 필요할까? 바로 활성화 함수를 사용해 여러 층을 쌓아서 더 복잡한 표현을 하기 위해서이다. 활성화 함수가 h(x) = cx인 선형 함수라고 생각해보자. 이 때 n개의 층을 쌓았다고 할 때, 모델은 y = h^n(x) = c^nx로 나타낼 수 있다. c^n=k라는 상수로 치환하면 결국 1층을 가진 신경망과 동일하다. 그렇기 때문에 비선형인 활성화 함수가 필요한 것이다.

**`ReLU로 어떻게 곡선 함수를 근사하나요?`**

## 답변 - type: answer
A: ReLU는 양수일 때 인 선형 함수와 음수일 때 인 선형 함수 두 개를 결합된 형태이다. 그렇지만 ReLU는 선형 함수가 갖는 가산성을 만족하지 못하기 때문에 비선형 함수로 볼 수 있다.
ReLU를 여러 개 결합하면, 특정 지점에서 특정 각도만큼 선형 함수를 구부릴 수 있다. 이 성질을 이용하여 곡선 함수 뿐만 아니라 모든 함수에 근사를 할 수 있게 된다.

**`ReLU의 문제점은 무엇인가요?`**

## 답변 - type: answer
A: ReLU의 가장 큰 문제점은 바로 죽은 뉴런(Dead Neurons)이다. ReLU는 결과값이 음수인 경우 모두 0으로 취급하는데, back propagation시 기울기에 0이 곱해져 해당 부분의 뉴런은 죽고 그 이후의 뉴런 모두 죽게 된다. 이를 해결한 Leaky ReLU는 값이 음수일 때 조금의 음의 기울기를 갖도록 하여 뉴런이 조금이라도 기울기를 갖도록 한다. 또 다른 방법으로는 입력값에 아주 조금의 편향(bias)를 주어 ReLU를 왼쪽으로 조금 이동시키는 방법이 있다. 이렇게 되면 입력값은 모두 양수이므로 뉴런이 모두 활성화가 되어 뉴런이 죽지 않는다.

두 번째 문제는 편향 이동(Bias Shift)이다. ReLU는 항상 0이상의 값을 출력하기 때문에 활성화값의 평균이 0보다 커 zero-centered하지 않다. 활성화값이 zero-centered되지 않으면 가중치 업데이트가 동일한 방향으로만 업데이트가 돼서 학습 속도가 느려질 수가 있다.

이를 해결하기 위해 배치 정규화(Batch Normalization)을 사용하거나 zero-centered된 ELU, SeLU와 같은 활성화 함수를 사용한다.

**`편향(bias)는 왜 있는 것일까요?`**

## 답변 - type: answer

A: 편향(bias)는 활성화 함수가 왼쪽 혹은 오른쪽으로 이동한다. 가중치(weight)는 활성화 함수의 가파른 정도 즉, 기울기를 조절하는 반면, 편향(bias)는 **활성화 함수를 움직임으로써 데이터에 더 잘 맞도록 한다.**

**`Gradient Descent에 대해서 쉽게 설명한다면 어떻게 설명할 수 있나요?`**

## 답변 - type: answer

A: gradient descent 방법은 steepest descent 방법이라고도 불리는데, 함수 값이 낮아지는 방향으로 독립 변수 값을 변형시켜가면서 최종적으로는 최소 함수 값을 갖도록 하는 독립 변수 값을 찾는 방법이다.

gradient descent의 목적과 사용 이유:
gradient descent는 함수의 최소값을 찾는 문제에서 활용된다.

함수의 최소, 최댓값을 찾으려면 "미분계수가 0인 지점을 찾으면 되지 않느냐?"라고 물을 수 있는데,

미분계수가 0인 지점을 찾는 방식이 아닌 gradient descent를 이용해 함수의 최소값을 찾는 주된 이유는

우리가 주로 실제 분석에서 맞딱드리게 되는 함수들은 닫힌 형태(closed form)가 아니거나 함수의 형태가 복잡해 (가령, 비선형함수) 미분계수와 그 근을 계산하기 어려운 경우가 많고,
실제 미분계수를 계산하는 과정을 컴퓨터로 구현하는 것에 비해 gradient descent는 컴퓨터로 비교적 쉽게 구현할 수 있기 때문이다.
추가적으로,

데이터 양이 매우 큰 경우 gradient descent와 같은 iterative한 방법을 통해 해를 구하면 계산량 측면에서 더 효율적으로 해를 구할 수 있다.

딥러닝에서는 Loss function을 최소화시키기 위해 파라미터에 대해 Loss function을 미분하여 그 기울기값(gradient)을 구하고, 경사가 하강하는 방향으로 파라미터 값을 점진적으로 찾기위해 사용된다.Gradient Descent의 문제점으로는 크게 두 가지가 있다.

첫 번째로 적절한 step size(learning rate)가 필요하다. step size가 큰 경우 한 번 이동하는 거리가 커지므로 빠르게 수렴할 수 있다는 장점이 있다. 하지만, step size를 너무 크게 설정해버리면 최소값을 계산하도록 수렴하지 못하고 함수 값이 계속 커지는 방향으로 최적화가 진행될 수 있다.

한편 step size가 너무 작은 경우 발산하지는 않을 수 있지만 최적의 x를 구하는데 소요되는 시간이 오래 걸린다는 단점이 있다.

두 번째로 local minima 문제이다. gradient descent 알고리즘을 시작하는 위치는 매번 랜덤하기 때문에 어떤 경우에는 local minima에 빠져 계속 헤어나오지 못하는 경우도 생긴다.

**`왜 꼭 Gradient를 써야 할까? 그 그래프에서 가로축과 세로축 각각은 무엇인가? 실제 상황에서는 그 그래프가 어떻게 그려질까요?`**

**## 답변 - type: answer**

**A:** Gradient가 양수이면 올라가는 방향이며 음수이면 내려가는 방향이다. 실제 상황에서는 Gradient 그래프가 0을 중심으로 진동하는 모양이 될 것이다.

**`GD 중에 때때로 Loss가 증가하는 이유는 무엇인가요?`**

**## 답변 - type: answer**

**A:** minima에 들어갔다가 나오는 경우일 것이다. 실제로 사용되는 GD에서는 local minima 문제를 피하기 위해 Momentum 등의 개념을 도입한 RMSprop, Adam 등의 optimization 전략을 사용한다.

각 optimization 전략에 따라 gradient가 양수인 방향으로도 parameter update step을 가져가는 경우가 생길 수 있으며, 이 경우에는 Loss가 일시적으로 증가할 수 있다.

**`Back Propagation에 대해서 쉽게 설명 한다면?`**

**## 답변 - type: answer**

**A:** 역전파 알고리즘은 Loss에 대한 입력값의 기울기(미분값)를 출력층 layer에서부터 계산하여 거꾸로 전파시키는 것이다.

이렇게 거꾸로 전파시켜서 최종적으로 출력층에서의 output값에 대한 입력층에서의 input data의 기울기 값을 구할 수 있다.

이 과정에서 chain rule이 이용된다.

출력층 바로 전 layer에서부터 기울기(미분값)을 계산하고 이를 점점 거꾸로 전파시키면서 전 layer들에서의 기울기와 서로 곱하는 형식으로 나아가면 최종적으로 출력층의 output에 대한 입력층에서의 input의 기울기(미분값)을 구할 수가 있다.

역전파 알고리즘이 해결한 문제가 바로 파라미터가 매우 많고 layer가 여러개 있을때 가중치w와 b를 학습시키기 어려웠다는 문제이다.

이는 역전파 알고리즘으로 각 layer에서 기울기 값을 구하고 그 기울기 값을 이용하여 Gradient descent 방법으로 가중치w와 b를 update시키면서 해결되었다.

**`Local Minima 문제에도 불구하고 딥러닝이 잘 되는 이유는 무엇인가요?`**

**## 답변 - type: answer**

**A:** local minima 문제가 사실은 고차원(High Dimensional)의 공간에서는 발생하기 힘든, 매우 희귀한 경우이기 때문이다. 실제 딥러닝 모델에서는 weight가 수도없이 많으며, 그 수많은 weight가 모두 local minima에 빠져야 weight update가 정지되기 때문에 local minima는 큰 문제가 되지 않는다.

Local Minima 문제에도 불구하고 딥러닝이 잘 되는, 더 구체적인 이유- 고차원의 공간에서 모든 축의 방향으로 오목한 형태가 형성될 확률은 거의 0에 가깝다. 따라서, 고차원의 공간에서 대부분의 critical point는 local minima가 아니라 saddle point다. 그리고, 고차원의 공간에서 설령 local minima가 발생한다 하더라도 이는 global minimum이거나 또는 global minimum과 거의 유사한 수준의 에러 값을 갖는다. 왜냐하면, critical point에 포함된 위로 볼록인 방향 축의 비율이 크면 클수록 높은 에러를 가지기 때문이다.(실험적 결과) local minima는 위로 볼록인 경우가 하나도 없는 경우이기 때문에 결과적으로 매우 낮은 에러를 갖게 될 것이다.

Critical point, Saddle point, Local minimum- 주요 포인트들:
critical point: 일차 미분이 0인 지점이다. (local/global)minima, (local/global)maxima, saddle point를 가리킴
local minimum: 모든 방향에서 극소값을 만족하는 점
global minimum: 모든 방향에서 극소값을 만족하는 점 중에 가장 값이 작은 점(정답)
saddle point: 어느 방향에서 보면 극대값이지만 다른 방향에서 보면 극소값이 되는 점`

**`GD(Gradient Descent)가 Local Minima 문제를 피하는 방법은 무엇인가요?`**

**## 답변 - type: answer**
**A:** Local minima 문제를 피하는 방법으로는 Momentum, Nesterov Accelerated Gradient(NAG), Adagrad, Adadelta, RMSprop, Adam 등이 있다.

SGD는 Stochastic Gradient Descent으로, 하나 혹은 여러개의 데이터를 확인한 후에 어느 방향으로 갈 지 정하는 가장 기초적인 방식이다.

Momentum이란 관성을 의미하며, 이전 gradient의 방향성을 담고있는 momentum 인자를 통해 흐르던 방향을 어느 정도 유지시켜 local minima에 빠지지 않게 만든다. 즉, 관성을 이용하여, 학습 속도를 더 빠르게 하고, 변곡점을 잘 넘어갈 수 있도록 해주는 역할을 수행한다.

Nesterov Accelerated Gradient(NAG) 는 모멘텀과 비슷한 역할을 수행하는 Look-ahead gradient 인자를 포함하여, a 라는 accumulate gradient가 gradient를 감소시키는 역할을 한다. 모멘텀과 다른 점은, 미리 한 스텝을 옮겨가본 후에 어느 방향으로 갈지 정한다는 것이다.

Adagrad란 뉴럴넷의 파라미터가 많이 바뀌었는지 적게 바뀌었는지 확인하고, 적게 변한건 더 크게 변하게 하고, 크게 변한건 더 작게 변화시키는 방법이다. Adagrad는 sum of gradient s`Quares(G_t)를 사용하는데, 이는 그래디언트가 얼만큼 변했는지를 제곱해서 더하는 것이므로 계속 커진다는 문제가 발생한다. G_t가 계속 커지면 분모가 점점 무한대에 가까워지게 되어, W 업데이트가 되지 않게 되어, 뒤로 갈수록 학습이 점점 안되는 문제점이 발생한다.

Adadelta는 Exponential Moving Average(EMA)를 사용하여, Adagrad의 G_t가 계속 커지는 현상을 막을 수 있다. EMA는 현재 타임스텝으로부터 윈도우 사이즈만큼의 파라미터 변화(그래디언트 제곱의 변화)를 반영하는 역할을 하는데, 이전의 값을 모두 저장하는 것이 아닌, 이전 변화량에 특정 비율을 곱해 더한 인자를 따로 두는 방식이다. Adadelta는 learning rate가 없다.

**`찾은 해가 Global Minimum인지 아닌지 알 수 있는 방법은 무엇인가요?`**

## 답변 - type: answer
A: Gradient Descent 방식에서 local minima에 도달함은 증명되어있으나, global minima에 도달하는 것은 보장되지 않았다. 또한, 현재 지점이 global minima인지도 알 수 없다. 딥러닝에서 다루는 문제가 convexity를 만족하지 않기 때문이다. 대신, local minima를 찾는다면, 그 지점이 곧 global minima일 가능성이 크다. Local Minima 문제에도 불구하고 딥러닝이 잘 되는 이유는?에서 언급했듯, saddle point가 아닌 완전한 local minimum이 발생하는 경우는 희귀하다. 따라서 모든 방향에서 아래로 볼록인 local minima를 발견한다면, 그 지점이 바로 global minima일 가능성이 높다.

### ` Training 세트와 Test 세트를 분리하는 이유는 무엇인가요?`

## 답변 - type: answer
A: 모델은 데이터에 대해 예측값을 만들고 정답과 비교하며 업데이트되면서 학습이 된다. 그런데 학습 데이터에 대해서는 좋은 성능을 낸다 하더라도 본 적 없는 데이터에 대해서는 잘 대응하지 못하는 오버피팅 문제가 생긴다면 좋은 모델이 아니다.

이를 막기 위해 학습된 모델이 처음 보는 데이터에도 강건하게 성능을 내는지 판단하기 위한 수단으로 test 세트를 따로 만든다.

모델이 새로운 샘플에 얼마나 잘 일반화될지 아는 유일한 방법은 새로운 샘플에 실제로 적용하여 generalization error(일반화 오차, 새로운 샘플에 대한 오류 비율)를 확인하는 것이다. 하지만 모델을 실제 서비스에 바로 적용하기엔 리스크가 너무 크다.
이 때문에 training data를 training set과 test set 두 개로 나누는 것이다. 이름 그대로 training set을 이용해 모델을 트레이닝시키고 test set을 이용해 모델을 테스트하여 generalization error의 estimation(추정값)을 얻는다. 이 추정값으로 모델이 새로운 샘플에 얼마나 잘 작동할지를 판단한다.

### `Validation 세트가 따로 있는 이유는 무엇인가요?`

## 답변 - type: answer
A: 모델을 학습시키고 test 데이터를 통해 모델의 일반화 성능을 파악하고, 다시 모델에 새로운 시도를 하고 test 데이터를 통해 모델의 성능을 파악한다고 생각해보자.

이 경우, 모델은 결국 test 데이터에도 오버피팅이 되어 다시 처음 보는 데이터를 주면 좋은 성능을 보장할 수 없게 된다.

이 문제를 막기 위해 validation 세트를 사용한다. validation 세트를 통해 모델의 성능을 평가하고 하이퍼파라미터 등을 수정하는 것이다.

즉, train 데이터로 모델을 학습시키고 valid 데이터로 학습된 모델의 성능 평가를 하고 더 좋은 방향으로 모델을 수정한다. 그리고 최종적으로 만들어진 모델로 test 데이터를 통해 최종 성능을 평가한다.

validation set은 training set과는 별도로 하이퍼파라미터를 튜닝하는데 사용하는 데이터셋이다.
linear 모델과 multinomial 모델 중 어떤 것을 선택할지 갈등하고 있다고 하자. training set으로 training 시키고 test set을 사용해 generalization error를 비교해보니 linear 모델이 더 좋았다. 그래서 linear 모델을 사용하기로 결정했고, 이제 overfitting을 피하기 위해 regularization을 적용하려고 한다. 하이퍼파라미터 튜닝으로 최적의 하이퍼파라미터를 찾았고 generalization error 추정값이 5%인 모델을 완성했다. 비로소 이 모델을 실제 서비스에 투입하였다. 하지만 성능이 예상만큼 좋지 않았고 generalization error가 15%나 되었다. 왜 그럴까?

generalization error를 동일한 test set에서 여러 번 측정하였으므로 모델의 하이퍼파라미터가 test set에 최적화되었기 때문이다. 이 문제에 대한 일반적인 해결 방법은 validation set이라고 부르는 두 번째 holdout set를 만드는 것이다. training set을 사용해 다양한 하이퍼파라미터로 여러 모델을 training 시키고, validation set에서 최상의 성능을 내는 모델과 하이퍼파라미터를 선택한다. 최상의 모델을 찾았으면 generlization error의 추정값을 얻기 위해 test set으로 단 한번의 최종 테스트를 한다.

cross-validation은 또 뭐야?
데이터셋이 충분하지 않은 경우 validation set에 데이터를 나눠주는 것도 매우 아쉽다. 하지만 cross-validation을 사용한다면 validation set을 나누지 않아도 된다.
training set을 여러 subset으로 나누고, 각 모델을 이 subset의 조합으로 training시키고 나머지 부분으로 test한다. 특정 데이터에만 하이퍼파라미터를 튜닝한 것이 아니고 모든 데이터를 대상으로 하이퍼파라미터를 튜닝한 것이기 때문에 실제 서비스와의 generalization error의 추정값과 generalization error가 유사할 것으로 기대된다.

데이터들을 나눌 때 클래스 비율이 보장되도록 나눌 수도 있고, 랜덤으로 나눌 수도 있으며, 이미지와 같은 경우 같은 사람의 이미지가 여러 장 있다면 training/test 로 각기 다른 곳에 들어가지 않도록 그룹화하는 경우도 있다.

**`Test 세트가 오염되었다는 말의 뜻은 무엇인가요?`**

**## 답변 - type: answer**
**A:** test 데이터는 한 번도 학습에서 본 적 없는 데이터여야 한다. 그런데 train 데이터가 test 데이터와 흡사하거나 포함되기까지한다면 test 데이터는 더이상 학습된 모델의 성능 평가를 객관적으로 하지 못한다.

이렇듯 test 데이터가 train 데이터와 유사하거나 포함된 경우에 test 세트가 오염되었다고 말한다.

데이터 누수는 훈련 데이터에 타겟에 대한 정보가 포함됐지만, 그 정보를 실제 예측에서는 사용할 수 없는 경우 발생한다. 이는 훈련 데이터 셋, 심지어 검증 데이터 셋에서도 높은 성능을 이끌어낼 수 있는 반면, 모델이 실제 배포됐을 때 제대로 동작하지 않을 것이다.

다시 말해, 누수가 일어나면 모델을 실제로 사용해서 의사 결정을 시작하기 전까지는 문제점을 찾지 못할 가능성이 높다. 하지만 그 이후에 모델이 매우 부정확하다는 사실을 깨닫게 된다.
데이터 누수에는 2가지 주요 유형이 있다. 타겟 누수(target leakage)와 훈련-테스트 오염(train-test contamination)이 바로 그것이다.

Target leakage
타겟 누수(Target leakage)는 예측 시점에서 사용할 수 없는 데이터가 데이터 셋에 포함되어 있을 때 발생한다. 단순히 어떤 특성이 좋은 예측을 위해 필요한가만 고민하는 것뿐만 아니라, 해당 특성을 사용할 수 있게 되는 타이밍이나 시간 순서의 관점에서 타겟 누수에 대해 생각해 볼 필요가 있다.

Train-Test Contamination
또 다른 유형의 누수는 검증 데이터와 훈련 데이터를 제대로 구별하지 않았을 때 발생한다.

검증 데이터셋 검사는 이전에 고려하지 않았던 데이터들에 대해서 모델의 예측 결과를 측정하기 위한 것이다. 하지만 검증 데이터가 전처리에 영향을 준다면 이 과정이 미묘하게 손상될 수 있다.

예를 들어 train_test_split 함수를 호출하기 전 전처리를 실행한다고 가정해보자. 최종 결과는 어떨까? 모델의 검증 점수가 높으므로 신뢰도가 높지만, 실제로 배포할 때에는 성능이 저하된다.

결국 검증 데이터나 테스트 데이터의 데이터를 예측 방법에 통합했으므로, 새로운 데이터에 일반화할 수 없는 경우에도 특정 데이터에 대해서도 잘 작동할 수도 있다. 더 복잡한 특성 공학을 수행하면 이 문제가 더욱 미묘하고 위험해진다.

만약 검증 데이터가 train-test split 함수로 생성됐다면 검증 데이터를 모든 fitting에서 제외하고, 전처리 단계의 fitting에 포함시켜야 한다. 이는 scikit-learn의 파이프라인을 이용하면 더 쉽다. 교차 검증을 사용할 때는 파이프 라인 내에서 전처리를 수행하는 것이 훨씬 중요하다.

Conclusion
많은 데이터 과학 응용 프로그램에서 데이터 누수때문에 몇억원의 손해가 발생할 수 있다고 생각해보자. 훈련 / 검증 데이터를 주의 깊게 관찰해서 분리하면 훈련-테스트 오염을 방지할 수 있다. 파이프라인이 이러한 분리를 할 때 유용하다.

**`Regularization이란 무엇인가요?`**

**## 답변 - type: answer**

**A:** 모델의 오버피팅을 막고 처음 보는 데이터에도 잘 예측하도록 만드는 방법을 Regularization(일반화)라고 한다.
대표적인 방법으로 Dropout, L1, L2 Regularization 등이 존재한다.

**`Batch Normalization의 효과는 무엇인가요?`**

**## 답변 - type: answer**

**A:** 배치 정규화(Batch Normalization)은 학습 시 미니배치 단위로 입력의 분포가 평균이 0, 분산이 1이 되도록 정규화한다. 더불어 gamma로 스케일과 beta로 이동 변환을 수행한다. 이렇게 배치 정규화를 사용하면 다음과 같은 효과를 얻을 수 있다.
장점 1: 기울기 소실/폭발 문제가 해결되어 큰 학습률을 설정할 수 있어 학습속도가 빨라진다.
장점 2: 항상 입력을 정규화시키기 때문에 가중치 초깃값에 크게 의존하지 않아도 된다.
장점 3 :자체적인 규제(Regularization) 효과가 있어 Dropout이나 Weight Decay와 같은 규제 방법을 사용하지 않아도 된다.

배치 정규화 (Batch Normalization) 를 더 자세히 살펴보면, 신경망에서 학습시 Gradient 기반의 방법들은 파라미터 값의 작은 변화가 신경망 출력에 얼마나 영향을 미칠 것인가를 기반으로 파라미터 값을 학습시키게 된다. 만약 파라미터 값의 변화가 신경망 결과의 매우 작은 변화를 미치게 될 경우 파라미터를 효과적으로 학습 시킬 수 없게 된다. Gradient 라는 것이 결국 미분값 즉 변화량을 의미하는데 이 변화량이 매우 작아지거나(Vanishing) 커진다면(Exploding) 신경망을 효과적으로 학습시키지 못하고, Error rate 가 낮아지지 않고 수렴해버리는 문제가 발생하게 된다.

그래서 이러한 문제를 해결하기 위해서 Sigmoid 나 tanh 등의 활성화 함수들은 매우 비선형적인 방식으로 입력 값을 매우 작은 출력 값의 범위로 squash 해버리는데, 가령 sigmoid는 실수 범위의 수를 [0, 1]로 맵핑해버린다. 이렇게 출력의 범위를 설정할 경우, 매우 넓은 입력 값의 범위가 극도로 작은 범위의 결과 값으로 매핑된다. 이러한 현상은 비선형성 레이어들이 여러개 있을 때 더욱 더 효과를 발휘하여(?) 학습이 악화된다. 첫 레이어의 입력 값에 대해 매우 큰 변화량이 있더라도 결과 값의 변화량은 극소가 되어 버리는 것이다. 그래서 이러한 문제점을 해결하기 위해 활성화 함수로 자주 쓰이는 것이 ReLU(Rectified Linear Unit) 이다. 또한 아래와 같은 방법들도 존재한다.

Change activation function : 활성화 함수 중 Sigmoid 에서 이 문제가 발생하기 때문에 ReLU 를 사용
Careful initialization : 가중치 초기화를 잘 하는 것을 의미
Small learning rate : Gradient Exploding 문제를 해결하기 위해 learning rate 값을 작게 설정함

위와 같은 트릭을 이용하여 문제를 해결하는 것도 좋지만, 이러한 간접적인 방법 보다는 "학습하는 과정 자체를 전체적으로 안정화"하여 학습 속도를 가속 시킬 수 있는 근본적인 방법인 "배치 정규화(Batch Normalization)"를 사용하는 것이 좋다. 이는 위와 마찬가지로 Gradient Vanishing / Gradient Exploding이 일어나는 문제를 방지하기 위한 아이디어이다.

**`Dropout의 효과는 무엇인가요?`**

**## 답변 - type: answer**
**A:** 드롭아웃(Dropout)은 설정된 확률 p만큼 은닉층(hidden layer)에 있는 뉴런을 무작위로 제거하는 방법으로, 오버피팅을 방지하기 위한 방법 중 하나이다. (정확히는 출력을 0으로 만들어 더이상의 전파가 되지 않도록 한다.) 드롭아웃(Dropout)은 학습 때마다 무작위로 뉴런을 제거하므로 매번 다른 모델을 학습시키는 것으로 해석할 수 있다. 그리고 추론 시 출력에 제거 확률 p를 곱함으로써 앙상블 학습에서 여러 모델의 평균을 내는 효과를 얻을 수 있다.

**`BN 적용해서 학습 이후 실제 사용시에 주의할 점은 무엇인가요? 코드로는 무엇인가요?`**

**## 답변 - type: answer**
**A:** 학습 과정에서는 미니 배치의 평균과 분산을 계산하여 배치 정규화를 적용하지만, 추론 시에는 학습 데이터 전체에 대한 평균과 분산을 계산하여 적용을 해야 한다. 왜냐하면 사용자가 설정한 배치의 크기에 따라 추론 결과가 변할 수도 있기 때문이다.

Motivation: Deep learning의 속도를 어떻게 더 빠르게 만들 수 있을까?
Deep learning이 잘 동작하고, 뛰어난 성능을 보인다는 것은 이제 누구나 알고 있다. 그러나 여전히 deep learning은 굉장히 시간이 오래 걸리는 작업이고, 그만큼 computation power도 많이 필요로 한다. 그 동안의 연구 결과를 보면, converge한 것처럼 보이더라도 더 많이 돌리게 된다면 더 좋은 결과로 수렴한다는 것을 알 수 있는 만큼, deep neural network의 train 속도를 높이는 것은 전체적인 성능 향상에 도움이 될 것이다.

보통 Deep learning을 train할 때에는 stochastic gradient descent (SGD) method를 사용한다. SGD의 속도를 높이는 가장 naive한 방법은 learning rate를 높이는 것이지만, 높은 learning rate는 보통 gradient vanishing 혹은 gradient exploding problem을 야기한다는 문제가 있다.

Gradient vanishing은 backpropagation algorithm에서 아래 layer로 내려갈수록, 현재 parameter의 gradient를 계산했을 때 앞에서 받은 미분 값들이 곱해지면서 그 값이 거의 없어지는 (vanish하는) 현상을 의미한다. Gradient exploding은 learning rate가 너무 높아 diverge하는 현상을 말한다. Learning rate의 값이 크면 이 두 가지 현상이 발생할 확률이 높기 때문에 우리는 보통 작은 learning rate를 고르게 된다. 그러나 우리는 이미 일반적으로 learning rate의 값이 diverge하지 않을 정도로 크면 gradient method의 converge 속도가 향상된다는 것을 알고 있다. 어떤 논문이 던지는 질문은 다음과 같다. 자연스럽게 나오는 궁금증은 Gradient vanishing/exploding problem이 발생하지 않도록 하면서 learning rate 값을 크게 설정할 수 있는 neural network model을 design할 수 있는가?

Internal Covariate Shift: learning rate의 값이 작아지는 이유
Gradient vanishing problem이 발생하는 이유에 대해서는 여러가지 설명이 가능하지만 (exploding은 그냥 우리가 값을 작게 설정하여 해결할 수 있다) 이 논문에서는 internal covariate shift라는 개념을 제안한다

어떠한 논문에서는 단순히 train/test input data의 distribution이 변하는 것 뿐 아니라, 각각의 layer들의 input distribution이 training 과정에서 일정하지 않기 때문에 문제가 발생한다고 주장하며, 이렇게 각각의 layer들의 input distribution이 consistent하지 않은 현상을 internal convariate shift라고 정의한다. 이 논문에서 이것이 문제가 된다고 주장하는 이유는, 각각의 layer parameter들은 현재 layer에 들어오는 input data 뿐만 아니라 다른 model parameter들에도 영향을 받기 때문이라고한다. 즉, gradient vanishing problem이 발생하는 이유를 backpropagation 과정에서 아래로 내려갈수록 이전 gradient들의 영향이 더 커져서 지금 parameter가 거의 update되지 않는다고 설명하는 것과 같은 맥락이다.

기존에는 이런 현상을 방지하기 위하여 ReLU neuron을 사용하거나 (Nair & Hinton, 2010), cafeful initialization을 사용하거나 (Bengio & Glorot, 2010; Saxe et al., 2013), leanring rate를 작게 취하는 등의 전략을 사용했지만, 그런 방법이 아닌 다른 방법을 통해 internal covariate shift 문제가 해결이 된다면 더 높은 learning rate를 선택하여 learning 속도를 빠르게하는 것이 가능할 것이다.

Navie approach: Whitening
따라서 이 논문의 목표는 internal covariate shift를 줄이는 것이다. 그렇다면 internal covariate shift는 어떻게 줄일 수 있을까? 이 논문에서는 엄청 간단하게 input distribution을 zero mean, unit variance를 가지는 normal distribution으로 normalize 시키는 것으로 문제를 해결하며, 이를 whitening이라한다 (LeCun 1998, Wiesler & Ney 2011). 주어진 column data Xin R^{dtimes n}에 대해 whitening transform은 다음과 같다.

hat{X} = Cov(X)^{-1/2} X, Cov(X) = E[( X - E[X] ) ( X - E[X] )^top ].

그러나 이런 naive한 approach에서는 크게 두 가지 문제점들이 발생하게 된다.

multi variate normal distribution으로 normalize를 하려면 inverse의 square root를 계산해야 하기 때문에 필요한 계산량이 많다.
mean과 variance 세팅은 어떻게 할 것인가? 전체 데이터를 기준으로 mean/variance를 training마다 계산하면 계산량이 많이 필요하다.
따라서 이 논문에서는 이런 문제점들을 해결할 수 있으면서, 동시에 everywhere differentiable하여 backpropagation algorithm을 적용하는 데에 큰 문제가 없는 간단한 simplification을 제안한다.

Batch Noramlization Transform
앞서 제시된 문제점들을 해결하기 위하여 이 논문에서는 두 가지 approach를 제안한다.

각 차원들이 서로 independent하다고 가정하고 각 차원 별로 따로 estimate를 하고 그 대신 표현형을 더 풍성하게 해 줄 linear transform도 함께 learning한다
전체 데이터에 대해 mean/variance를 계산하는 대신 지금 계산하고 있는 batch에 대해서만 mean/variance를 구한 다음 inference를 할 때에만 real mean/variance를 계산한다
먼저 naive approach에서 covariance matrix의 inverse square root를 계산해야했던 이유는 모든 feature들이 서로 correlated되었다고 가정했기 때문이지만, 각각이 independent하다고 가정함으로써, 단순 scalar 계산만으로 normalization이 가능해진다. 이를 수식으로 표현하면 다음과 같다.

d dimensional data x = (x^{(1)}, x^{(2)}, ldots, x^{(d)})에 대해 각각의 차원 k 마다 다음과 같은 식을 계산하여 hat x를 계산한다

hat{x}^{(k)} = frac{x^{(k)} - E[x^{(k)}]}{sqrt{Var[x^{(k)}]}}.

그러나 이렇게 correlation을 무시하고 learning하는 경우 각각의 관계가 중요한 경우 제대로 되지 못한 training을 하게 될 수도 있으므로 이를 방지하기 위한 linear transform을 각각의 dimension $k$마다 learning해준다. 이 transform은 scaling과 shifting을 포함한다.

y^{(k)} = gamma hat{x}^{(k)} + beta.

이때 parameter gamma, beta는 neural network를 train하면서 마치 weight를 update하듯 같이 update하는 model parameter이다.

두 번째로, 전체 데이터의 expectation을 계산하는 대신 주어진 mini-batch의 sample mean/variance를 계산하여 대입한다.

이제 앞서 설명한 두 가지 simplification을 적용하여 다음과 같은 batch normalization transform이라는 것을 정의할 수 있다.

Train/Inference with BN network
앞에서 batch normalization transform을 각각의 layer input을 normalization하는데에 사용할 것이라는 설명을 했었다. 다시말해서 BN network는 기존 network에서 각각의 layer input 앞에 batch normalization layer라는 layer를 추가한 것과 구조가 동일하다.

주의해야할 점 하나는 train 과정에서는 mini-batch의 sample mean/variance를 사용하여 BN transform을 계산하였지만, inference를 할 때에도 같은 규칙을 적용하게 되면 mini-batch 세팅에 따라 inference가 변할 수도 있기 때문에 각각의 test example마다 deterministic한 결과를 얻기 위하여 sample mean/variance 대신 그 동안 저장해둔 sample mean/variance들을 사용하여 unbiased mean/variance estimator를 계산하여 이를 BN transform에 이용한다.

BN network의 장점
저자들이 주장하는 BN network의 장점은 크게 두 가지이다.

더 큰 learning rate를 쓸 수 있다. internal covariate shift를 감소시키고, parameter scaling에도 영향을 받지 않고, 더 큰 weight가 더 작은 gradient를 유도하기 때문에 parameter growth가 안정화되는 효과가 있다.
Training 과정에서 mini-batch를 어떻게 설정하느냐에 따라 같은 sample에 대해 다른 결과가 나온다. 따라서 더 general한 model을 learning하는 효과가 있고, drop out, l2 regularization 등에 대한 의존도가 떨어진다.
어떠한 논문을 살펴보면 BN transform이 scale invariant하고, 큰 weight에 대해 작은 gradient가 유도되기 때문에 paramter growth를 안정화시키는 효과가 있다는 언급이 있다. 또한 regularization효과를 더 강화하기 위하여 매 mini-batch마다 training data를 shuffling하여 input으로 넣는데, 이때 한 mini-batch 안에서는 같은 데이터가 중복으로 나오지 않도록 shuffling하여 대입한다.

BN 네트워크 성능 accelerating하기
BN을 추가하는 것 만으로 성능 개선이 엄청나게 일어나는 것은 아니며 다음과 같은 parameter tunning이 추가로 필요하다고 한다. 1. learning rate 값을 키운다 (0.0075 -> 0.045, 5) 2. drop out을 제거한다 (BN이 regularization 효과가 있기 때문이라고 한다) 3. l2 weight regularization을 줄인다 (BN이 regularization 효과가 있기 때문이라고 한다) 4. learning rate decay를 accelerate한다 (6배 더 빠르게 가속한다) 5. local response normalization을 제거한다 (BN에는 적합하지 않다고 한다) 6. training example의 per-batch shuffling을 추가한다 (BN이 regularization 효과를 증폭시키기 위함이다) 7. photometric distortion을 줄인다 (BN이 속도가 더 빠르고 더 적은 train example을 보게 되기 때문에 실제 데이터에 더 집중한다고 한다)

이런 parameter tunning이 추가로 이루어지고 나면, 기존 neural network보다 ImageNet에서 훨씬 좋은 성능을 내는 neural network를 구성할 수 있다고 한다.

**`GAN에서 Generator 쪽에도 BN을 적용해도 될까요?**

**## 답변 - type: answer**
**A:** 일반적으로 GAN에서는 생성기(Generator)의 출력층(Output Layer)에만 BN(Batch Normalization)을 적용하지 않는다. 왜냐하면 생성기가 만든 이미지가 BN을 지나면 실제 이미지와는 값의 범위가 달라지기 때문이다.

DCGAN이 대단한 이유
GAN의 불안정성
Ian Goodfellow가 GAN을 발표한 이후로 많은 분야에 GAN이 적용하여 연구가 되었지만 그 때마다 항상 불안정한 구조로 인한 문제가 따라붙었습니다. 그래서 GAN을 다룬 논문들을 보면 하나같이 이 부분에 대한 어려움에 대하여 얘기를 하고 있고, NIPS 2016에서 Tutorial이나 workshop 세션에서도 큰 주제가 되었던 부분이 바로 이 "GAN의 안정화"였습니다.

Minimax 혹은 saddle problem을 풀어야하는 GAN은 어쩔 수 없이 태생적으로 불안정할 수 밖에 없는데요 앞선 글에서 살펴보았듯이 이론적으로는 fixed solution으로 수렴하는 것이 보장되어 있지만, 실제 적용에서는 이론적 가정이 깨지면서 생기는 불안정한 구조적 단점을 보이곤 했습니다.

DCGAN은 이름에서 알 수 있듯이 Convolutional 구조를 GAN에 녹여 넣었습니다. 아시다시피 지도학습(Supervised Learning)에 Convolutional Neural Network (CNN)을 이용한 것은 Computer vision application에 큰 반향을 일으켰죠. 그에 반해 비지도 학습(Unsupervised Learning)에 CNN을 이용하는 것은 아주 적은 관심을 받았는데요
DCGAN paper는 지도학습에서의 CNN의 성공과 비지도 학습 간의 격차를 줄이는 데에 큰 역할을 하였습니다.....만

사실 위와 같은 역사적 중요성과 같은 추상적이고 서사적?인 영향력을 굳이 말할 필요도 없이 그냥 결과가 엄.청.납니다.

일단 DCGAN에서 제시한 가이드 라인대로 GAN 구조를 짜면 상당히 안정적으로 학습이 되는 것을 볼 수 있는데다, 생성 모델(Generative Model)이 보여줄 수 있는 역할은 거진 다 보여주었습니다.

대부분의 상황에서 언제나 안정적으로 학습이 되는 Convolutional GAN 구조(DCGAN)를 제안하였다는 점
마치 word2vec과 같이 DCGAN으로 학습된 Generator가 벡터 산술 연산이 가능한 성질을 갖고 이것으로 semantic 수준에서의 sample generation을 해볼 수 있다는 점
DCGAN이 학습한 filter들을 시각화하여 보여주고 특정 filter들이 이미지의 특정 물체를 학습했다는 것을 보여주었다는 점
이렇게 학습된 Discriminator가 다른 비지도 학습 알고리즘들과 비교하여 비등한 이미지 분류 성능을 보였다는 점

이렇게 네 가지로 요약할 수 있겠습니다.
그 중 상당히 재미있다고 그리고 대단하다고 생각한 부분들을 강조해두었는데요. 테크니컬한 구조에 대한 설명도 좋지만 학습된 네트워크를 분석한 결과들이 더 흥미롭기 때문에 구조에 대한 부분은 가볍게 언급하고 분석에 대한 내용을 중점적으로 소개해보도록 하겠습니다.

DCGAN이 해결하고자 하는 문제
생성 영상의 해상도
일단 DCGAN이 처음으로 Convolution을 GAN에 넣으려 시도한 논문은 아닙니다. 이전에도 다른 연구자들이 Convolution을 GAN 구조에 넣으려 시도했으나 별로 좋은 결과를 얻지는 못했는데요 자연스럽게 굳이 Convolution을 왜 넣으려고 하는가? 하는 의문이 들 수 있습니다.

현재 아는 바로는 Scale up의 문제 때문에 그런 것으로 알고 있습니다. DCGAN의 이전에 고해상도? 혹은 높은 품질의 이미지 생성이 가능하기라도 한 GAN 모델으로는 LAPGANs(Denton et al.,2015)가 유일했습니다. 그러다 최소한 한정된 이미지 종류에 대해서긴 하지만 (예를 들자면, 침실 이미지들) DCGAN이 최초로 상당히 높은 품질의 영상을 single shot으로 만들어 내는 것에 성공하였습니다.

이전에 있던 여러 생성 모델들의 결과를 보면, 사실 MNIST 숫자에 대한 결과들이나 질감 합성(texture synthesis) 등에 한정적으로 사용이 된 경우가 아니면 현실로부터 얻을 수 있는 자연적인 영상을 생성하는 것에 성공한 모델은 거의 없다고 보시면 됩니다. Ian Goodfellow가 제시한 초기 GAN도 생성된 이미지를 보면 사실 해석을 하기엔 뭔가 좀 그런 영상들을 만들어내었지 정말 사람이 보아도 그럴듯한 이미지를 만들어 내지는 못했습니다. LAPGAN은 조금 더 그럴듯한 영상을 만들기는 했지만 구조로부터 기인하는 노이즈로 인해 물체들이 마치 흔들린 듯한 형체로 생성되는 문제가 여전히 남아있었습니다.

CNNs are Black-box Methods
생성 영상의 품질에 대한 문제 외에도 Neural Network는 항상 black-box method일뿐이라는 지적을 피할 수가 없죠. 이를 해결하기 위해 Zeiler가 2014년에 deconvolution을 이용해서 convolution filter들이 네트워크에서 어떤 역할을 하고 있는지 보여주려한 시도 등이 있었지만 여전히 남아있는 문제들이 있습니다.

Measure for sample evaluation
특히나 생성 모델(Generative Model)의 성능을 판단하는 기준이 모호하기 때문에 실제 네트워크가 잘했는지 못했는지를 정량적으로 얘기하기가 쉽지 않다는 것이 같이 결부되면서 문제가 심화됩니다.

아무래도 정답이 있는 것이 아니다보니 생성한 이미지가 '얼마나' 정확한지 혹은 그럴듯한지를 얘기하기가 쉽지 않을뿐더러 그에 맞는 측정 방법이 마땅치도 않습니다. 주관적인 판단으로 얘기를 할 수는 있겠지만 말 그대로 이는 선호도 조사와 다를바가 없지요.

이 때문에 아직도 이 부분은 많은 연구가 필요한 부분으로 남아있는 것 같습니다. 다만 이런 측정 방법이 없는 문제를 우회적으로나마 해결하기 위해 GAN의 학습이 끝난 이후 얻은 Discriminator 혹은 Generator의 feature(weight)들을 이용하여 데이터를 transform한 후 K-mean clustering 등을 통해 분류한 결과가 얼마나 좋은 지로 간접적으로 얘기하는 방법을 택하는 경우도 많이 사용되는 것 같습니다.

기존의 GAN 논문에서는 Parzen window based likelihood를 이용하여 네트워크가 생성한 이미지와 원래 이미지가 얼마나 유사한지를 비교하는 방식을 사용했는데, 최근 NIPS 2016 Tutorial report에서도 언급되었듯 더이상 이런 방식을 사용하지는 않는 것 같습니다. 이에 관한 얘기는 Theis의 2015년 논문 "A note on the evaluation of generative models"를 보시면 될 듯 합니다.

**`SGD, RMSprop, Adam에 대해서 아는대로 설명한다면 무엇인가요?`**

**## 답변 - type: answer**

**A:** SGD - Loss Function을 계산할 때 전체 train set을 사용하는 것을 Batch Gradient Descent 라고 한다. 그러나 이렇게 계산을 할 경우 한번 step을 내딛을 때 전체 데이터에 대해 Loss Function을 계산해야 하므로 너무 많은 계산량이 필요하다.
이를 방지하기 위해 보통은 Stochastic Gradient Descent(SGD)라는 방법을 사용한다.
이 방법에서는 loss function을 계산할 때 전체 데이터(batch) 대신 데이터 한 개 또는 일부 조그마한 데이터의 모음(mini-batch)에 대해서만 loss function을 계산한다.
데이터 한 개를 사용하는 경우를 Stochastic Gradient Descent(SGD), 데이터의 일부(mini-batch)를 사용하는 경우를 mini-batch Stochastic Gradient Descent(mini-batch SGD)라고 하지만 오늘날의 딥러닝에서 일반적으로 통용되는 SGD는 mini-batch SGD 이다.

이 방법은 batch gradient descent 보다 다소 부정확할 수는 있지만, 훨씬 계산 속도가 빠르기 때문에 같은 시간에 더 많은 step을 갈 수 있으며 여러 번 반복할 경우 보통 batch의 결과와 유사한 결과로 수렴한다.

또한, SGD를 사용할 경우 Batch Gradient Descent에서 빠질 local minima에 빠지지 않고 더 좋은 방향으로 수렴할 가능성도 있다.

RMSprop - RMSProp은 딥러닝의 대가 제프리 힌톤이 제안한 방법으로서, Adagrad의 단점을 해결하기 위한 방법이다.

Adagrad의 식에서 gradient의 제곱값을 더해나가면서 구한 $G_t$부분을 합이 아니라 지수평균으로 바꾸어서 대체한 방법이다.

이렇게 대체를 할 경우 Adagrad처럼 G_t가 무한정 커지지는 않으면서 최근 변화량의 변수간 상대적인 크기 차이는 유지할 수 있다.

Adam - Adam(Adaptive Moment Estimation)은 RMSProp과 Momentum 방식을 합친 것 같은 알고리즘이다.
이 방식에서는 Momentum 방식과 유사하게 지금까지 계산해온 기울기의 지수평균을 저장하며, RMSProp과 유사하게 기울기의 제곱값의 지수평균을 저장한다.
다만, Adam에서는 m과 v가 처음에 0으로 초기화되어 있기 때문에 학습의 초반부에서는 m_t, v_t가 0에 가깝게 bias 되어있을 것이라고 판단하여 이를 unbiased 하게 만들어주는 작업을 거친다.

m_t, v_t의 식을 ∑(시그마) 형태로 펼친 후 양변에 expectation을 씌워서 정리해보면, 다음과 같은 보정을 통해 unbiased 된 expectation을 얻을 수 있다.

이 보정된 expectation들을 가지고 gradient가 들어갈 자리에 widehat{m_t}, G_t가 들어갈 자리에 widehat{v_t}를 넣어 계산을 진행한다.

**`SGD에서 Stochastic의 의미는 무엇인가요?`**

## 답변 - type: answer
A: SGD는 Loss Function을 계산할 때 전체 train dataset을 사용하는 Batch Gradient Descent와 다르게 일부 조그마한 데이터의 모음(mini-batch)에 대해서만 loss function을 계산한다.

Stochastic은 mini-batch가 전체 train dataset에서 무작위로 선택된다는 것을 의미한다.

**`미니배치를 작게 할때의 장단점은 무엇인가요?`**

## 답변 - type: answer
A: 장점 - 한 iteration의 계산량이 적어지기 때문에 step 당 속도가 빨라진다.
적은 Graphic Ram으로도 학습이 가능하다.

단점 - 데이터 전체의 경향을 반영하기 힘들다. 업데이트를 항상 좋은 방향으로 하지만은 않는다.

그런데 미니 배치는 왜 쓸까?

딥러닝에서 한번의 iteration을 위해 들어가는 인풋데이터는 보통 batch라고 하여 수십수백개의 데이터를 한그룹으로 사용하게 됩니다. 그렇다면 mini-batch는 한번의 iteration에 인풋 데이터로 한개를 쓰는 경우와 전체 데이터셋을 쓰는 두 경우(양극단)에 비해 어떤 장점이 있길래 이렇게 당연한 듯이 쓰이는 걸까요. 당연한 말이지만 mini-batch는 두가지 방법의 장점을 모두 얻기 위한(서로의 단점을 보완) 타협점입니다, 아래에서는 두가지 방법의 장단점에 대해 알아보고 왜 mini-batch를 사용하는지 정리해보겠습니다.

(forward + backpropagation+업데이트를 거치는 한번의 과정을 iteration이라고 합니다.)

**데이터를 한개 쓰는 경우**

**장점:**

1)iteration 한번 수행하는데 소요되는 시간이 매우 짧습니다. cost function의 최적의 값을 찾아가는 과정을 한걸음한걸음 minimum을 향해 걸어가는 것으로 생각한다면 매우 빠르게 걸을 수 있습니다.

**단점:**

1)데이터 전체의 경향을 반영하기가 힘듭니다, 그래서 업데이트를 꼭 좋은 방향으로만 하지않습니다. 현재 학습을 진행하는 데이터 한개에 대해서는 cost function의 값이 줄어들더라도 이로 인해 다른 데이터에 대해서는 cost가 증가 할 수 있기 때문입니다. 결국 많이 헤매게 됩니다.

2)하드웨어입장에서 비효율적입니다, 현재 딥러닝에 GPU를 많이 쓰는 이유는 그 강력한 병렬연산능력 때문입니다. 그런데 한번에 데이터 한개만 학습에 사용한다면 그 병렬연산을 안쓰는 것이나 마찬가지입니다. 학습을 위해서 갈 길이 먼데 매우 아까운 낭비가 될 것입니다.

**전체데이터를 쓰는 경우**

**장점:**

1)전체 데이터를 반영하여 한걸음한걸음을 내딛습니다, 즉 정말로 cost function의 값을 줄이는 양질의 이동을 하게됩니다.

**단점:**

1)데이터셋의 크기가 커질 경우 iteration을 한번 수행하는데 소요되는 시간이 매우 깁니다, 최적의 위치를 찾아가기 위해서는 최소한으로 수행해야 하는 iteration이 존재하기에 학습시간이 매우 길어집니다. 이를 보완하기 위해서 learning rate를 높이려고 해봐도 쉽지 않습니다. 보통 학습을 진행 할 때 learning rate를 너무 크게 잡으면 local minimum만 왔다갔다하거나 minimum에 들어가지 못하는 shooting 현상이 생기기 때문입니다.

2)하드웨어입장에서 부담스럽습니다. 데이터셋이 커질 경우 그 데이터를 메모리에 올려야 될 뿐만 아니라 그 데이터의 전처리한 결과나 레이어를 거친 아웃풋 등도 수시로 메모리를 드나듭니다. 즉매우 큰 메모리용량이 필요하게 됩니다.

결국 데이터를 한개 쓰면 빠르지만 너무 헤매고, 전체를 쓰면 정확하지만 너무 느립니다. 즉 적당히 빠르고 적당히 정확한 길을 찾기 위해 mini-batch를 사용합니다. 적게는 수십개부터 많게는 수백개의 데이터를 한 그룹으로하여 처리함으로써 iteration 한번 수행하는데 소요되는 시간을 최대한 줄이면서 전체 데이터를 최대한 반영합니다. 동시에 보통은 가능한 한도 내에서 batch 크기를 최대한 크게 잡아 하드웨어에 부담을 주지 않는 선에서 하드웨어를 최대한 활용합니다. 이와 같은 이유로 거의 당연하게 mini-batch를 사용하게 됩니다.

**`모멘텀의 수식을 적어 본다면 무엇인가요?`**

**## 답변 - type: answer**

**A:** Momentum 방식은 말 그대로 Gradient Descent를 통해 이동하는 과정에 일종의 관성을 주는 것이다.

현재 Gradient를 통해 이동하는 방향과는 별개로, 과거에 이동했던 방식을 기억하면서 그 방향으로 일정 정도를 추가적으로 이동하는 방식이다.

Neural network의 weight을 조절하는 과정에는 보통 'Gradient Descent' 라는 방법을 사용한다. 이는 네트워크의 parameter들을 theta라고 했을 때, 네트워크에서 내놓는 결과값과 실제 결과값 사이의 차이를 정의하는 함수 Loss function J(theta)의 값을 최소화하기 위해 기울기(gradient) nabla_{theta} J(theta)를 이용하는 방법이다. Gradient Descent에서는 theta에 대해 gradient의 반대 방향으로 일정 크기만큼 이동해내는 것을 반복하여 Loss function J(theta)의 값을 최소화하는 heta의 값을 찾는다. 한 iteration에서의 변화 식은 다음과 같다.

이 때 eta는 미리 정해진 걸음의 크기 'step size'로서, 보통 0.01~0.001 정도의 적당한 크기를 사용한다.

이 때 Loss Function을 계산할 때 전체 train set을 사용하는 것을 Batch Gradient Descent 라고 한다. 그러나 이렇게 계산을 할 경우 한번 step을 내딛을 때 전체 데이터에 대해 Loss Function을 계산해야 하므로 너무 많은 계산량이 필요하다. 이를 방지하기 위해 보통은 Stochastic Gradient Descent (SGD) 라는 방법을 사용한다. 이 방법에서는 loss function을 계산할 때 전체 데이터(batch) 대신 일부 조그마한 데이터의 모음(mini-batch)에 대해서만 loss function을 계산한다. 이 방법은 batch gradient descent 보다 다소 부정확할 수는 있지만, 훨씬 계산 속도가 빠르기 때문에 같은 시간에 더 많은 step을 갈 수 있으며 여러 번 반복할 경우 보통 batch의 결과와 유사한 결과로 수렴한다. 또한, SGD를 사용할 경우 Batch Gradient Descent에서 빠질 local minima에 빠지지 않고 더 좋은 방향으로 수렴할 가능성도 있다.

보통 Neural Network를 트레이닝할 때는 이 SGD를 이용한다. 그러나 단순한 SGD를 이용하여 네트워크를 학습시키는 것에는 한계가 있다.

**`간단한 MNIST 분류기를 MLP+CPU 버전으로 numpy로 만든다면 몇줄일까요?`**

**## 답변 - type: answer**
**A:** 2-layer 신경망을 구현한다고 했을 때, 100줄 이내로 만들 수 있다.

**`어느 정도 돌아가는 녀석을 작성하기까지 몇시간 정도 걸릴까요?`**

**## 답변 - type: answer**

**A:** 간단한 MNIST 분류기를 MLP+CPU 버전으로 numpy로 만든 경우, 15 에폭 기준 0.9 이상의 정확도가 나온다고 한다. 이 100줄 가량의 코드를 작성하는데 걸리는 시간은 사람마다 다르겠지만, 구조를 정확히 알고있다면 오래걸려도 30분 내에는 작성할 수 있을 것이라 생각한다. 그러나 pretrain되지 않은 모델의 경우, 학습시간이 꽤 오래걸린다고 한다.

**`Back Propagation은 몇줄인가요?`**

## 답변 - type: answer
A: 참고 코드(경사하강법 적용) 기준으로 10줄이면 구현할 수 있다. gradient_descent 함수를 각 레이어별로 적용하면 미분값을 적용시킬 수 있다.

```
# 가중치 매개변수의 기울기를 구함 def numerical_gradient(f, x): h = 1e-4 gr
ad = np.zeros_like(x) # x와 형상이 같은 배열을 생성 for idx in range(x.siz
e): tmp_val = x[idx] # f(x+h) 계산 x[idx] = tmp_val + h fxh1 = f(x) # f(x-
h) 계산 x[idx] = tmp_val - h fxh2 = f(x) grad[idx] = (fxh1 - fxh2) / (2 *
h) x[idx] = tmp_val # 값 복원 return grad for key in ('W1', 'b1', 'W2', 'b
2'): network.params[key] -= learning_rate * grad[key]
```

### `CNN으로 바꾼다면 얼마나 추가될까요?`

## 답변 - type: answer
A: filter의 수, 크기, padding, stride 등에 대한 내용과 pooling layer등 레이어에 관한 정의가 추가되므로 약 50줄 정도 추가된다.
MLP 버전과 CNN 버전의 참고코드는 아래와 같다.

MLP 버전

신경망에는 적응 가능한 가중치와 편향이 있고, 이 가중치와 편향을 훈련 데이터에 적응하도록 조정하는 과정을 '학습'이라 한다. 신경망 학습은 다음과 같이 4단계로 수행한다. 1단계 - 미니배치 훈련 데이터 중 일부를 무작위로 가져온다. 이렇게 선별한 데이터를 미니배치라 하며, 그 미니배치의 손실함수 값을 줄이는 것이 목표이다. 2단계 - 기울기 산출 미니배치의 손실 함수 값을 줄이기 위해 각 가중치 매개변수의 기울기를 구한다. 기울기는 손실 함수의 값을 가장 작게 하는 방향을 제시한다. 3단계 - 매개변수 갱신 가중치 매개변수를 기울기 방향으로 아주 조금 갱신한다. 4단계 - 반복 1~3단계를 반복한다. 데이터를 무작위로 선정하기 때문에 확률적 경사 하강법stochastic gradient descent, SGD라고 부른다. ''' import sys import os import numpy as np sys.path.append(os.pardir) from common.functions import sigmoid, softmax, cross_entropy_error from common.gradient import numerical_gradient class TwoLayerNet: """ params : 신경망의 매개변수를 보관하는 딕셔너리 변수. params['W1']은 1번째 층의 가중치, params['b1']은 1번째 층의 편향. params['W2']은 2번째 층의 가중치, params['b2']은 2번째 층의 편향. grad : 기울기를 보관하는 딕셔너리 변수 (numerical_gradient의 반환값) grads['W1']은 1번째 층의 가중치의 기울기, grads['b1']은 1번째 층의 편향의 기울기. grads['W2']은 2번째 층의 가중치의 기울기, grads['b2']은 2번째 층의 편향의 기울기. """ # 초기화를 수행한다. def __init__(self, input_size, hidden_size, output_size, weight_init_std=0.01): # 가중치 초기화 self.params = {} self.params['W1'] = weight_init_std * \ np.random.randn(input_size, hidden_size) self.params['b1'] = np.zeros(hidden_size) self.params['W2'] = weight_init_std * \ np.random.randn(hidden_size, output_size) self.params['b2'] = np.zeros(output_size) # 예측(추론)을 수행한다. def predict(self, x): W1, W2 = self.params['W1'], self.params['W2'] b1, b2 = self.params['b1'], self.params['b2'] a1 = np.dot(x, W1) + b1 z1 = sigmoid(a1) a2 = np.dot(z1, W2) + b2 y = softmax(a2) return y # 손실 함수의 값을 구한다. # x : 입력데이터, t : 정답 레이블 def loss(self, x, t): y = self.predict(x) return cross_entropy_error(y, t) # 정확도를 구한다. def accuracy(self, x, t): y = self.predict(x) y = np.argmax(y, axis=1) t = np.argmax(t, axis=1) accuracy = np.sum(y == t) / float(x.shape[0]) return accuracy # 가중치 매개변수의 기울기를 구한다. def numerical_gradient(self, x, t): loss_W = lambda W: self.loss(x, t) grads = {} grads['W1'] = numerical_gradient(loss_W, self.params['W1']) grads['b1'] = numerical_gradient(loss_W, self.params['b1']) grads['W2'] = numerical_gradient(loss_W, self.params['W2']) grads['b2'] = numerical_gradient(loss_W, self.params['b2']) return grads if __name__ == '__main__': net = TwoLayerNet(input_size=784, hidden_size=100, output_size=10) print(net.params['W1'].shape) # (784, 100) print(net.params['b1'].shape) # (100,) print(net.params['W2'].shape) # (100, 10) print(net.params['b2'].shape) # (10,) x = np.random.rand(100, 784) # 더미 입력 데이터(100장 분량) t = np.random.rand(100, 10) # 더미 정답 레이블(100장 분량) grads = net.numerical_gradient(x, t) # 기울기 계산 # 주의 : 실행하는데 아주 오래걸림 print(grads['W1'].shape) # (784, 100)

```
print(grads['b1'].shape) # (100,) print(grads['W2'].shape) # (100, 10)
print(grads['b2'].shape) # (10,)
```

CNN 버전

```
import sys import os sys.path.append(os.pardir) # 부모 디렉터리의 파일을 가져올 수 있도록 설정 import numpy as np import matplotlib.pyplot as plt from collections import OrderedDict from common.layers import * from common.gradient import numerical_gradient from dataset.mnist import load_mnist from common.trainer import Trainer class SimpleConvNet: """ 다음과 같은 CNN을 구성한다. → Conv → ReLU → Pooling → Affine → ReLU → Affine → Softmax → 전체 구현은 simple_convnet.py 참고 """ def __init__(self, input_dim=(1, 28, 28), conv_param={'filter_num': 30, 'filter_size': 5, 'pad': 0, 'stride': 1}, hidden_size=100, output_size=10, weight_init_std=0.01): # 초기화 인수로 주어진 하이퍼파라미터를 딕셔너리에서 꺼내고 출력 크기를 계산한다. filter_num = conv_param['filter_num'] filter_size = conv_param['filter_size'] filter_pad = conv_param['pad'] filter_stride = conv_param['stride'] input_size = input_dim[1] conv_output_size = (input_size - filter_size + 2*filter_pad) / \ filter_stride + 1 pool_output_size = int(filter_num * (conv_output_size/2) * (conv_output_size/2)) # 1층의 합성곱 계층과 2, 3층의 완전연결 계층의 가중치와 편향 생성 self.params = {} self.params['W1'] = weight_init_std * \ np.random.randn(filter_num, input_dim[0], filter_size, filter_size) self.params['b1'] = np.zeros(filter_num) self.params['W2'] = weight_init_std * \ np.random.randn(pool_output_size, hidden_size) self.params['b2'] = np.zeros(hidden_size) self.params['W3'] = weight_init_std * \ np.random.randn(hidden_size, output_size) self.params['b3'] = np.zeros(output_size) # CNN을 구성하는 계층을 생성 self.layers = OrderedDict self.layers['Conv1'] = Convolution(self.params['W1'], self.params['b1'], conv_param['stride'], conv_param['pad']) self.layers['Relu1'] = Relu self.layers['Pool1'] = Pooling(pool_h=2, pool_w=2, stride=2) self.layers['Affine1'] = Affine(self.params['W2'], self.params['b2']) self.layers['Relu2'] = Relu self.layers['Affine2'] = Affine(self.params['W3'], self.params['b3']) self.last_layer = SoftmaxWithLoss def predict(self, x): """추론을 수행""" for layer in self.layers.values: x = layer.forward(x) return x def loss(self, x, t): """손실함수 값 계산""" y = self.predict(x) return self.last_layer.forward(y, t) def accuracy(self, x, t, batch_size=100): if t.ndim != 1: t = np.argmax(t, axis=1) acc = 0.0 for i in range(int(x.shape[0] / batch_size)): tx = x[i*batch_size: (i+1)*batch_size] tt = t[i*batch_size:(i+1)*batch_size] y = self.predict(tx) y = np.argmax(y, axis=1) acc += np.sum(y == tt) return acc / x.shape[0] def gradient(self, x, t): """오차역전파법으로 기울기를 구함""" # 순전파 self.loss(x, t) # 역전파 dout = 1 dout = self.last_layer.backward(dout) layers = list(self.layers.values) layers.reverse for layer in layers: dout = layer.backward(dout) # 결과 저장 grads = {} grads['W1'] = self.layers['Conv1'].dW grads['b1'] = self.layers['Conv1'].db grads['W2'] = self.layers['Affine1'].dW grads['b2'] = self.layers['Affine1'].db grads['W3'] = self.layers['Affine2'].dW grads['b3'] = self.layers['Affine2'].db return grads # 본 신경망으로 실제 MNIST 데이터셋을 학습하는 코드는 train_convnet.py
```

```
참고 # 데이터 읽기 (x_train, t_train), (x_test, t_test) =
load_mnist(flatten=False) # 시간이 오래 걸릴 경우 데이터를 줄인다. x_train,
t_train = x_train[:5000], t_train[:5000] x_test, t_test = x_test[:1000],
t_test[:1000] max_epochs = 20 network = SimpleConvNet(input_dim=(1, 28,
28), conv_param={'filter_num': 30, 'filter_size': 5, 'pad': 0, 'stride':
1}, hidden_size=100, output_size=10, weight_init_std=0.01) trainer =
Trainer(network, x_train, t_train, x_test, t_test, epochs=max_epochs,
mini_batch_size=100, optimizer='Adam', optimizer_param={'lr': 0.001},
evaluate_sample_num_per_epoch=1000) trainer.train # 그래프 그리기 markers
= {'train': 'o', 'test': 's'} x = np.arange(max_epochs) plt.plot(x,
trainer.train_acc_list, marker='o', label='train', markevery=2)
plt.plot(x, trainer.test_acc_list, marker='s', label='test', markevery=2)
plt.xlabel("epochs") plt.ylabel("accuracy") plt.ylim(0, 1.0)
plt.legend(loc='lower right') plt.show """ ================ Final Test
Accuracy ================ test acc:0.959 전체로 학습했을 경우 약 98%까지 가
능 """
```

**`간단한 MNIST 분류기를 TF, PyTorch 등으로 작성하는데 몇시간이 필요한가요?`**
**## 답변 - type: answer**
**A:** TF 나 Pytorch 를 몇 번 사용해본 사람이라면 도큐먼트 참고도 하고 적당히 구글링도 하면, MNIST 분류기의 데이터 다운로드, 데이터셋, 데이터로더, 모델 세팅, 학습, 추론 를 구현하는데 2시간이 걸리지 않을 것이라 생각한다.

강력한 성능을 내는 모델도 이러한 프레임워크를 사용하면 빠른 시간 내에 구현해낼 수 있음에 감사하고, 추상화가 잘 된 함수들일지라도 안에서는 어떤 동작을 하는지 알고 사용해야한다.

**`CNN이 아닌 MLP로 해도 잘 될까요?`**
**## 답변 - type: answer**
**A:** Convolution 레이어는 receptive field 를 통해 이미지의 위치 정보까지 고려할 수 있다는 장점이 있다.
반면 MLP 는 모두 Fully connected 구조이므로 이미지의 특징을 이해하는데 픽셀마다 위치를 고려할 수 없게된다.
따라서 MNIST 분류기에서 MLP 를 사용하면 CNN 을 사용했을 때보다 성능이 낮다.

**`마지막 레이어 부분에 대해서 설명 한다면 무엇인가요?`**
**## 답변 - type: answer**
**A:** MNIST 분류기는 Convolution 레이어를 깊게 쌓으며 숫자 이미지의 작은 특징부터 큰 특징까지 파악한다.
마지막 레이어, Fully connected 레이어는 이미지 데이터의 특징을 취합하여 10개의 숫자 중 적절한 숫자로 분류하는 역할을 한다.
만약 더 많은 레이블에 대해 분류해야 한다면 마지막 레이어의 out dimension 을 그에 맞게 설정하면 된다.

**`학습은 BCE loss로 하되 상황을 MSE loss로 보고 싶다면 어떻게 하나요?`**
**## 답변 - type: answer**
**A:** train 과정에서 criterion 은 BinaryCrossEntropy 를 사용하고, valid 데이터를 이용한 valid loss 를 구하는 과정에서는 MeanSquaredLoss 를 사용한다.

**`딥러닝할 때 GPU를 쓰면 좋은 이유는 무엇인가요?`**
**## 답변 - type: answer**
**A:** GPU(Graphics Processing Unit)은 부동 소수점 연산을 수행하는 많은 코어가 있어 수 많은 연산을 병렬처리할 수 있다. 또한 CPU보다 더 큰 메모리 대역폭을 가지고 있기 때문에 큰 데이터를 더 효율적으로 빠르게 처리할 수 있다.

메모리 대역폭(Memory Bandwidth)란 메모리가 처리할 수 있는 초당 데이터양을 뜻한다.

**`GPU를 두개 다 쓰고 싶다. 방법은 무엇인가요?`**

**## 답변 - type: answer**
**A:** Pytorch의 경우 torch.nn.DataParallel을 사용하여 여러 개의 GPU를 사용할 수 있다.
torch.device를 cuda로 설정한다.
nn.DataParallel을 사용하여 모델을 감싼다.
모델을 model.to(device)를 사용하여 GPU로 보낸다.

**Via a string:**

```
>>>torch.device('cuda:0')device(type='cuda', index=0) >>>torch.device('cpu')device(type='cpu') >>>torch.device('mps')device(type='mps') >>>torch.device('cuda')# current cuda devicedevice(type='cuda')
```

**to use GPUs with PyTorch**

```
#참고로 모델은 아무거나 예:model = SimpleModel) device = torch.device("cud
a") # torch.device를 cuda로 설정한다. model = nn.DataParallel(model) # nn.
DataParallel을 사용하여 모델을 감싼다. model = model.to(device) # 모델을 mo
del.to(device)를 사용하여 GPU로 보낸다.
```

copy all your tensors to the GPU:

```
mytensor = my_tensor.to(device)
```

**` 학습시 필요한 GPU 메모리는 어떻게 계산하나요?`**

**## 답변 - type: answer**

**A:** Pytorch를 기준으로 볼 때 something.to('cuda')로 변환하는 모든 것들을 생각해보면 된다. 보통 GPU로 올리는 것은 모델과 데이터셋이므로, (모델의 크기 + 데이터의 크기 × 배치 크기)로 학습시 필요한 메모리 크기를 계산할 수 있다.