Deep Learning with Tensorflow Keras

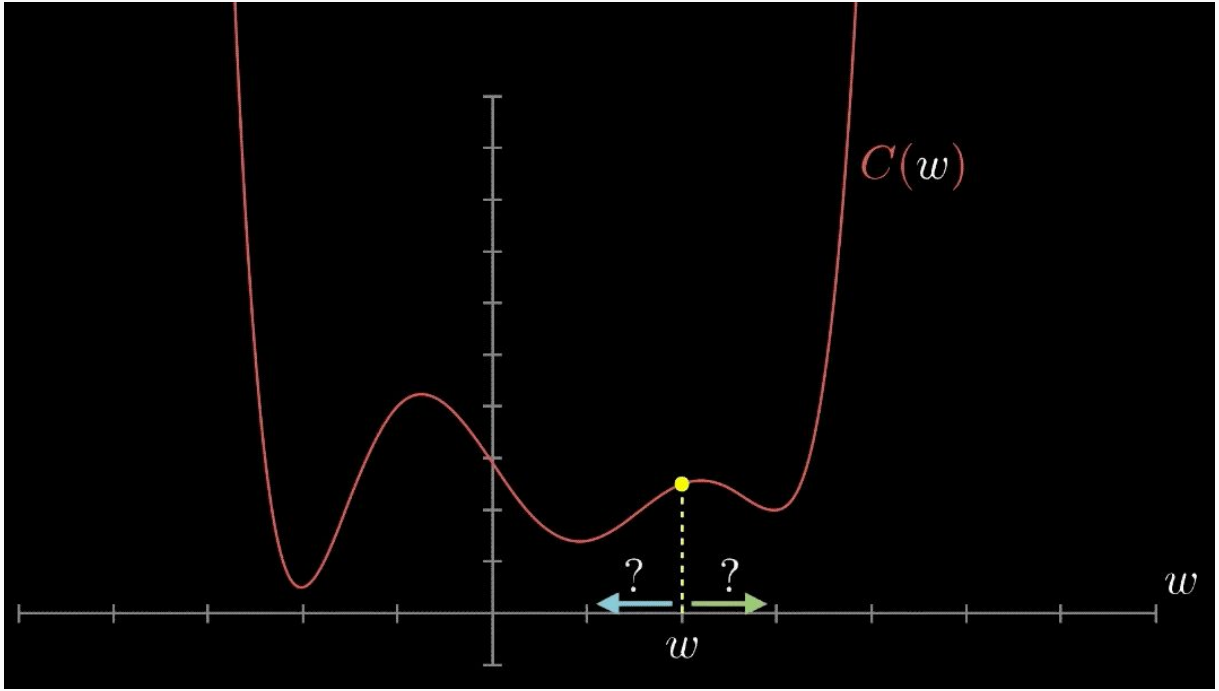
3강 딥러닝의 한계와 해결책

신유주

초기 딥러닝의 한계

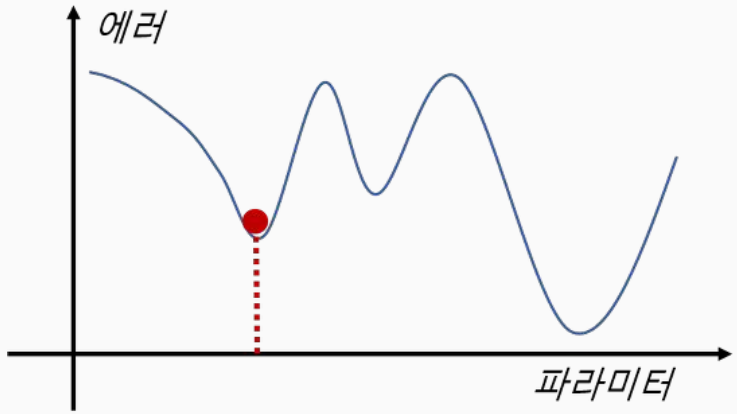
- 지역적 극소(local minima)로 학습될 수 있음
  - 손실함수의 global minimum이 아닌 다른 minima로 학습될 수 있음
- 경사 소실: 딥러닝 층이 깊어질수록 에러가 앞단까지 잘 전파되지 않음
- 데이터 부족: 데이터가 부족하면 딥러닝 파라미터가 제대로 학습되지 않음
- 느린 학습: DNN의 층이 많아지고 뉴런 수가 증가하면 계산량도 그만큼 늘어남(계산량 = 데이터 량 * 파라미터 수)
- 과적합(overfitting): 파라미터 수가 매우 많아 과적합되기 쉬움
- 초기화에 영향을 받음

경사하강법의 문제점



지역적 극소값(Local Minima)

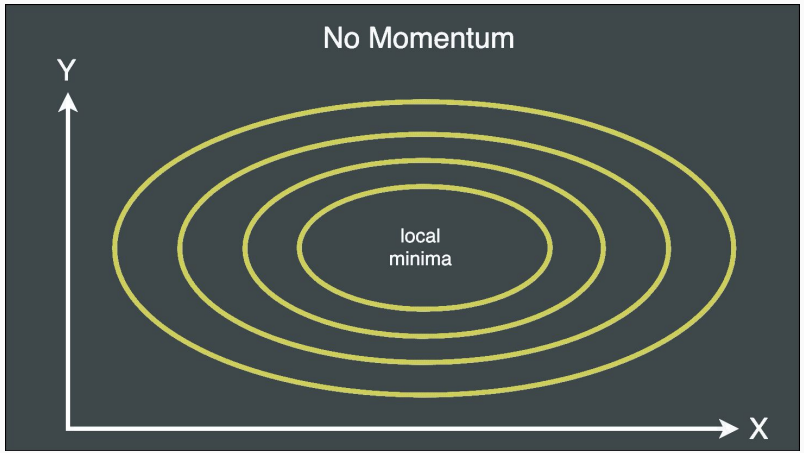
- SGD로 파라미터를 학습했을 때 손실함수의 지역극소에 갇힐 수 있음
- 이때 파라미터는 최적의 파라미터(전체 극소)가 아니기 때문에 성능저하가 일어남
- SGD를 변형해 지역적 극소를 탈출할 수 있도록 해야함
- Optimizer 수정을 통해 가능



SGD의 문제점

- local minima에 갇힐 수 있음
- 그래서 학습속도가 느림
  - Global optimum에 빠르게 도달하지 못함
- Zigzag로 움직이기 때문
  - SGD로 얻은 경사벡터의 방향은 최적의 방향이 아님

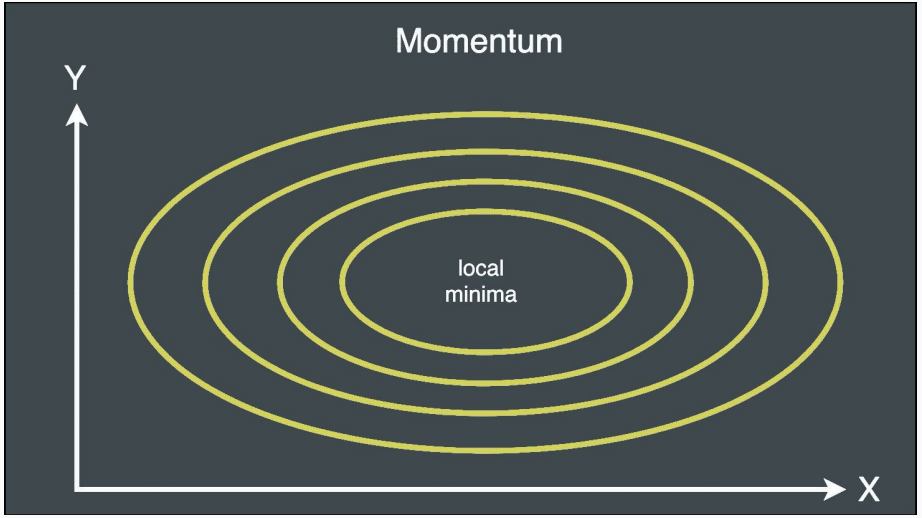
$$\theta = \theta - \eta \nabla L_\theta(\theta)$$



https://mlfromscratch.com/optimizers-explained/#/

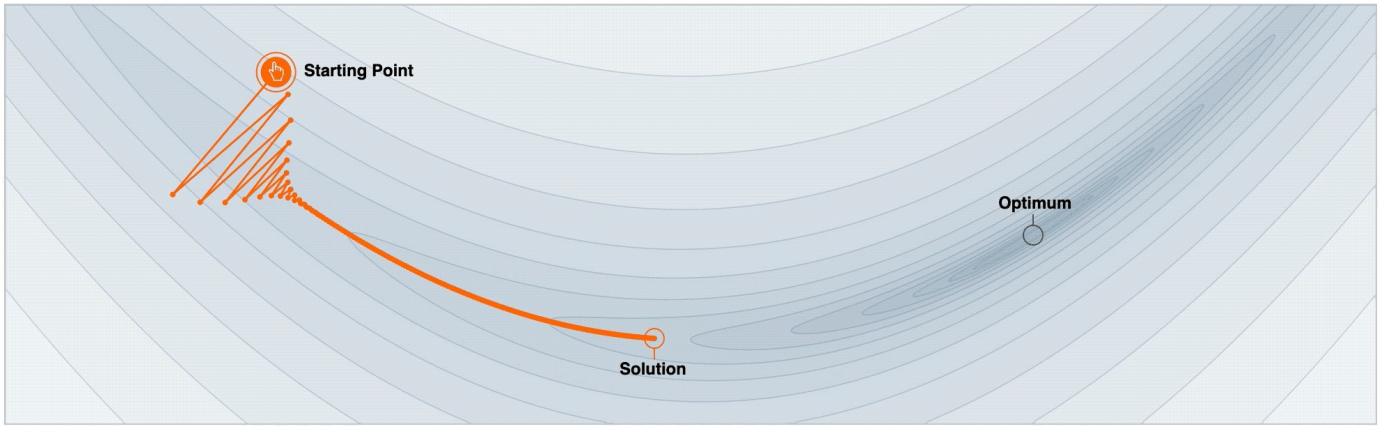
Momentum Optimizer

- 관성을 이용한 SGD의 변종
- 이전에 얻은 경사값의 일부를 다시 다음 경사값 에 더해줌



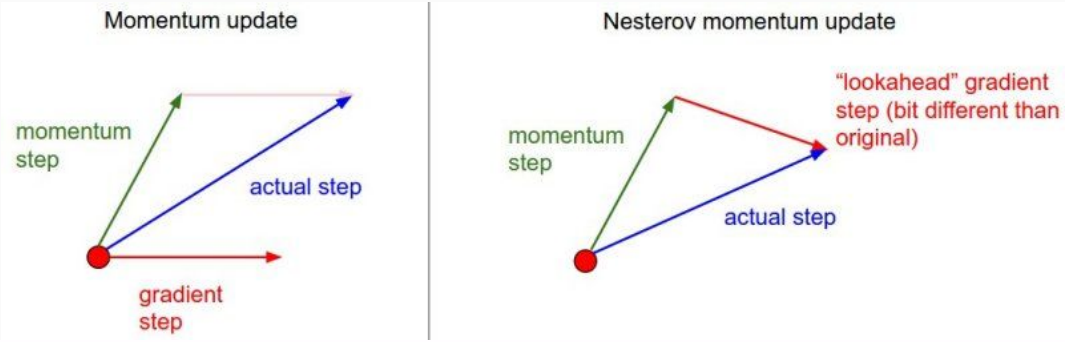
$$\Delta\theta_t = \eta\Delta\theta_{t-1} + \gamma\nabla_\theta L(\theta)$$
$$\theta_{t+1} = \theta_t - \Delta\theta_t$$

Momentum의 크기에 따른 변화



Nesterov Accelerated Gradient

- 모멘텀 경사값을 미리 더해본 후의 파라미터 값에 대한 경사값을 모멘텀에 다시 더해줌
- 현재 경사값은 사용하지 않음




```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9, nesterov=True)
```

AdaGrad(Adaptive Gradient)

- 학습이 부진한 변수는 learning rate를 크게 하여 더 많이 학습하게 하고
- 학습이 많이 된 변수는 learning rate를 줄여서 덜 변화하게 함
- 현 시점까지 그라디언트의 제곱값을 저장(G)
- 입실론은 임의의 작은 양수로 분모가 0이 되는 것을 방지함
- learning rate가 매우 작아질 수 있음

$$G_t = G_{t-1} + \left(\nabla_\theta L(\theta_t)\right)^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \nabla_\theta L(\theta_t)$$

RMSProp

- 너무 빨리 줄어드는 learning rate를 살리기 위해, 기존에 학습한 분량을 다 더하지 않고 지수 평균으로 대체

$$G_t = \gamma G_{t-1} + (1-\gamma)\left(\nabla_\theta L(\theta_t)\right)^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \nabla_\theta L(\theta_t)$$

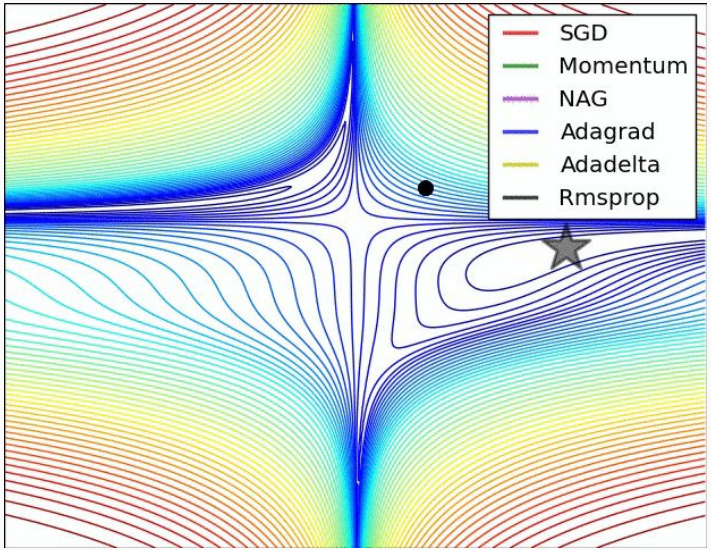
Adam

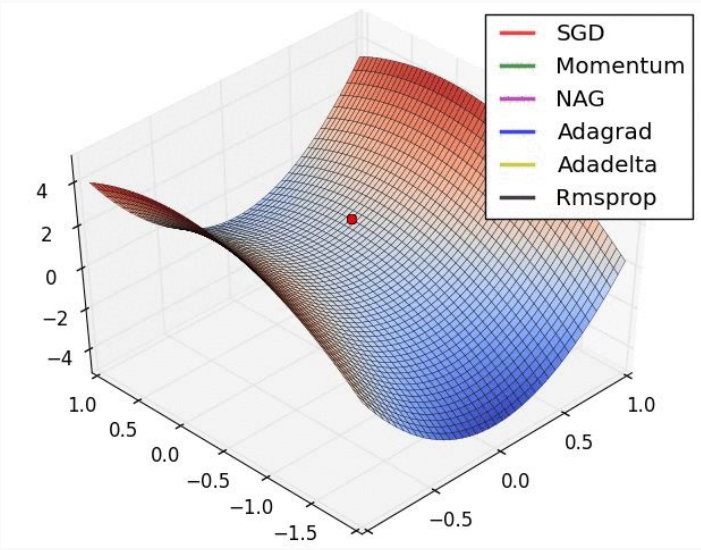
- =Momentum+AdaGrad+RMSProp
- m: momentum의 지수 평균(exponentially weighted average)
- v: 경사값 제곱의 지수 평균

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta L(\theta_t)$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\nabla_\theta L(\theta_t)^2$$

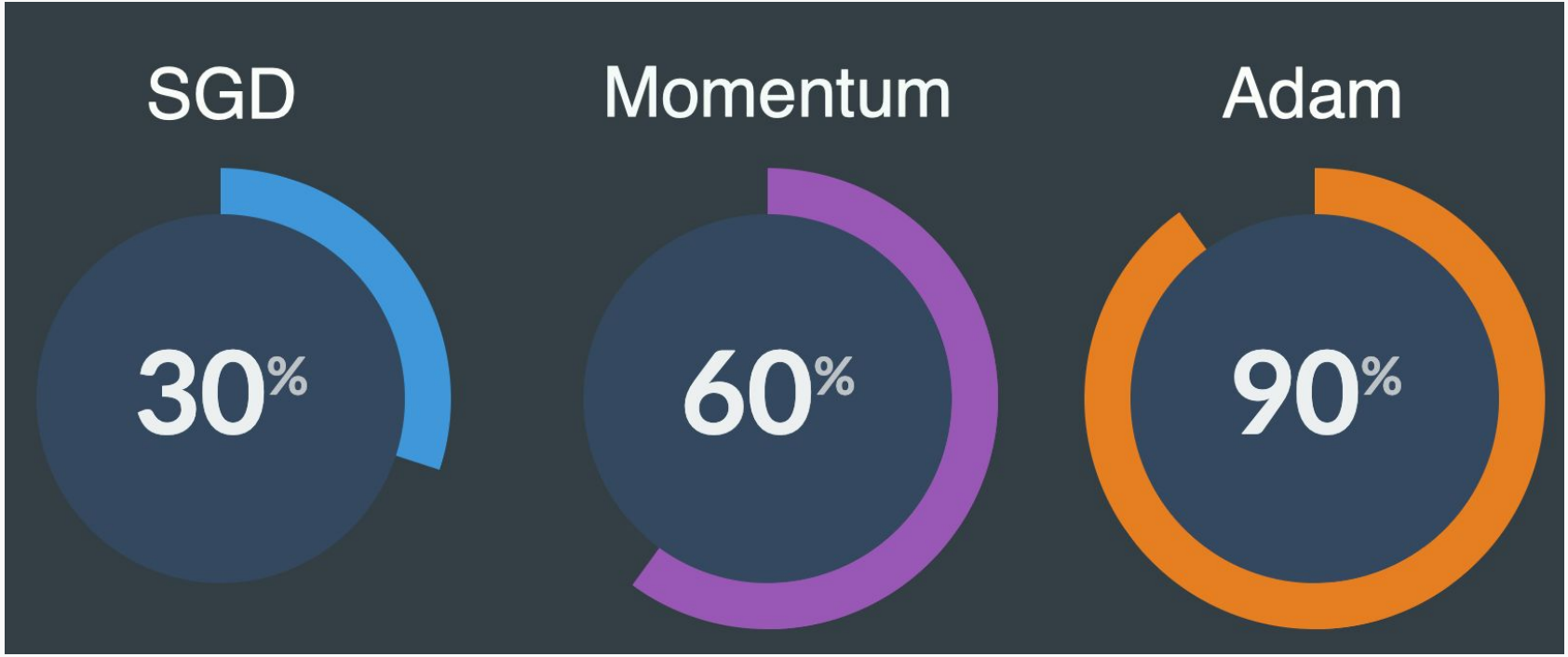
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}}\hat{m}_t$$

Optimizer 정리



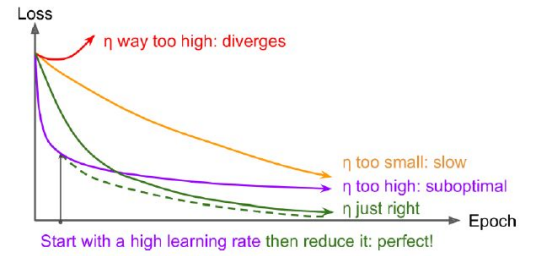


Optimizer 정리



Learning Rate Scheduling

- 큰 학습 속도로 시작해서 점점 줄여나가면 일정한 학습 속도로 학습시킬 때보다 더 좋은 성능의 DNN을 얻을 수 있음
- 지수함수, 1/x 다양한 함수 모양으로 줄일 수 있음
- 일정 epoch 도달 시 1/n을 하는 방법도 주로 사용됨
- Validation 데이터의 평가지표를 보고 학습 속도를 바꿀 수도 있음



Learning Rate Scheduling

```
def exponential_decay(lr0, s):
    def exponential_decay_fn(epoch):
        return lr0 * 0.1**(epoch / s)
    return exponential_decay_fn

exponential_decay_fn = exponential_decay(lr0=0.01, s=20)

lr_scheduler = keras.callbacks.LearningRateScheduler(exponential_decay_fn)
history = model.fit(X_train_scaled, y_train, [...], callbacks=[lr_scheduler])
```

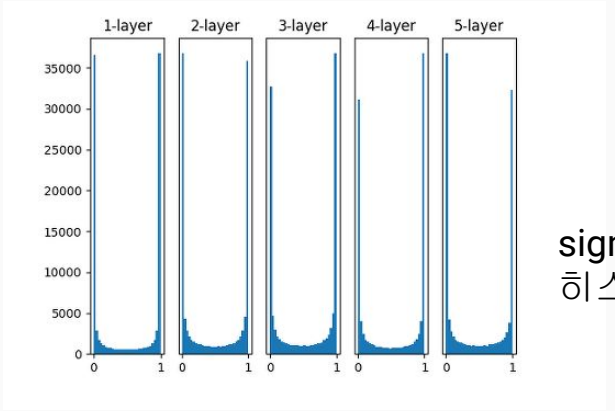
```
s = 20 * len(X_train) // 32 # number of steps in 20 epochs (batch size = 32)
learning_rate = keras.optimizers.schedules.ExponentialDecay(0.01, s, 0.1)
optimizer = keras.optimizers.SGD(learning_rate)
```

경사 소실(Vanishing Gradient)

- 층이 깊어지면 깊어질수록 손실함수에서 발생한 에러가 맨 아래층 가중치까지 전달되지 않음
- 역전파에서는 사용하는 연쇄법칙을 이용해 각 파라미터의 경사값을 구함
- 이때 0과 1사이의 값을 가지는 미분값을 여러번 곱해서 나중에가면 그 값이 0에 가까워지기 때문에 발생함
- 초기 가중치 설정과 활성화 함수에 영향을 받음

가중치 초기화(initialization)

- 예전엔 가중치 N(0,1)에서 샘플링을 해서 사용
- 하지만 활성화함수 중 sigmoid 함수는 0 근처에서 매우 낮은 경사값을 가짐
- 따라서 맨 아래 층에서 경사 소실이 일어날 가능성이 커짐
- sigmoid 함수 사용시 안정적 학습을 위해서 각 층의 출력값이 정규분포를 보여야 함
- 그래야 미분값이 0으로 수렴하지 않음
- 하지만 그냥 정규분포에서 뽑은 값으로 초기화하면 오른쪽 그림과 같은 출력값을 보임

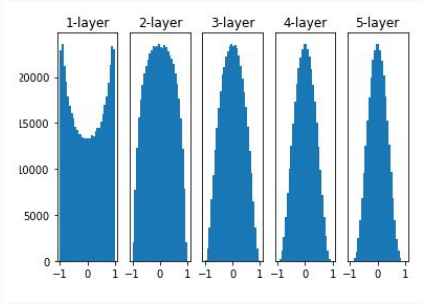


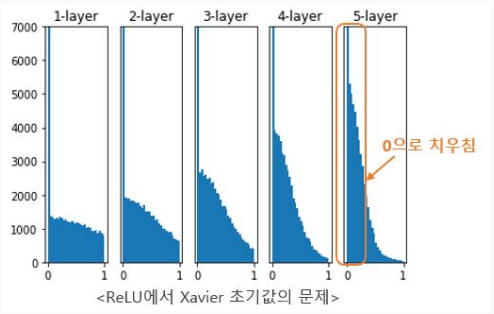
sigmoid값 히스토그램

Xavier(Glorot) Initialization

- 이를 막기위해 Xavier Initialization 등장
- 각 레이어의 출력에 대한 분산이 입력에 대한 분산과 같아야 하며,
  역전파에서 레이어를 통과하기 전과 후의 경사값의 분산이 동일해야 한다고 주장

- 평균이 0이고 표준편차 $\sigma = \sqrt{\frac{2}{n_{\text{inputs}}+n_{\text{outputs}}}}$ 인 정규분포
- 또는 $r = \sqrt{\frac{6}{n_{\text{inputs}}+n_{\text{outputs}}}}$ 일 때 $-r$과 $+r$ 사이의 균등분포
- 입력의 연결 개수와 출력의 연결 개수가 비슷할 경우 $\sigma = 1/\sqrt{n_{\text{inputs}}}$ 또는 $r = \sqrt{3}/\sqrt{n_{\text{inputs}}}$ 를 사용



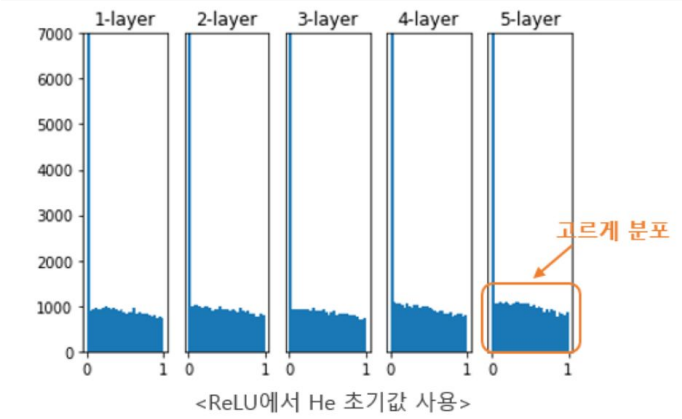


<ReLU에서 Xavier 초기값의 문제>

He Initialization

- Relu에 적합한 가중치 초기화

- 평균이 0이고 표준편차 $\sigma = \sqrt{2} \cdot \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$인 정규분포
- 또는 $r = \sqrt{2} \cdot \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$ 일 때 $-r$과 $+r$ 사이의 균등분포
- 입력의 연결 개수와 출력의 연결 개수가 비슷할 경우 $\sigma = \sqrt{2}/\sqrt{n_{\text{inputs}}}$ 또는 $r = \sqrt{2} \cdot \sqrt{3}/\sqrt{n_{\text{inputs}}}$ 를 사용



<ReLU에서 He 초기값 사용>

여러 가중치 초기화(initialization) 방법

Initialization	Activation functions	σ^2 (Normal)
Glorot	None, Tanh, Logistic, Softmax	$1 / fan_{avg}$
He	ReLU & variants	$2 / fan_{in}$
LeCun	SELU	$1 / fan_{in}$

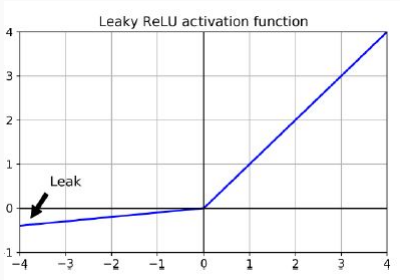
Keras default →

```
keras.layers.Dense(10, activation="relu", kernel_initializer="he_normal")
```

Relu의 문제점

- 경사 소실을 막기 위해 Relu를 사용해도 문제가 있음
- 음수 쪽의 경사값이 0이기 때문에 경사소실이 발생할 수 있음
- Leaky Relu등의 변형 사용으로 막을 수 있음
  - elu, selu, ...

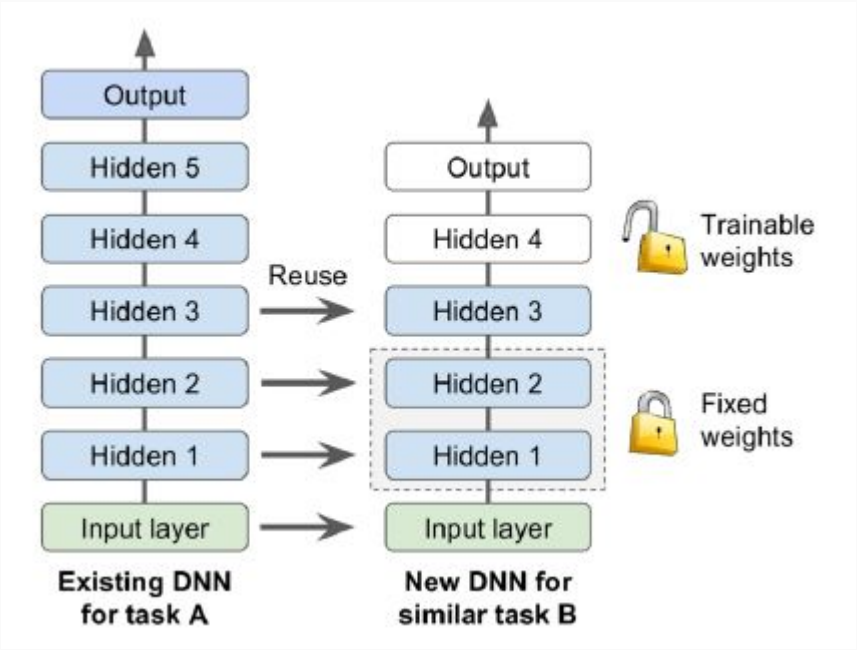
```
leaky_relu = keras.layers.LeakyReLU(alpha=0.2)
layer = keras.layers.Dense(10, activation=leaky_relu,
                           kernel_initializer="he_normal")
```



전이 학습(Transfer Learning)

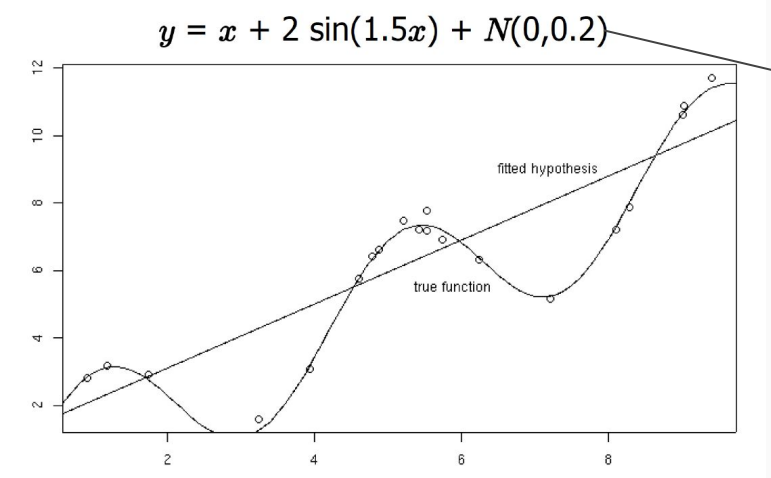
- 기존에 이미 훈련된 모델을 사용
- 유사한 종류의 일부 데이터로 그 모델의 일부만 훈련시킴
  - 개/고양이를 구분하도록 만든 DNN을 호랑이/사자를 분류하는 데 사용
- 아주 편리하게 남이 훈련시켜 놓은 모델을 빠르게 학습시켜 사용가능
- 하지만 적용하려는 데이터가 원래 모델을 학습시킨 데이터와 매우 다르거나 하려는 작업이 다르면 잘 작동하지 않을 수 있음

전이 학습(Transfer Learning)




```
model_A = keras.models.load_model("my_model_A.h5")
model_B_on_A = keras.models.Sequential(model_A.layers[:-1])
model_B_on_A.add(keras.layers.Dense(1, activation="sigmoid"))

for layer in model_B_on_A.layers[:-1]:
    layer.trainable = False

model_B_on_A.compile(loss="binary_crossentropy", optimizer="sgd",
                     metrics=["accuracy"])

history = model_B_on_A.fit(X_train_B, y_train_B, epochs=4,
                           validation_data=(X_valid_B, y_valid_B))

for layer in model_B_on_A.layers[:-1]:
    layer.trainable = True

optimizer = keras.optimizers.SGD(lr=1e-4) # the default lr is 1e-3
model_B_on_A.compile(loss="binary_crossentropy", optimizer=optimizer,
                     metrics=["accuracy"])
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=16,
                           validation_data=(X_valid_B, y_valid_B))
```

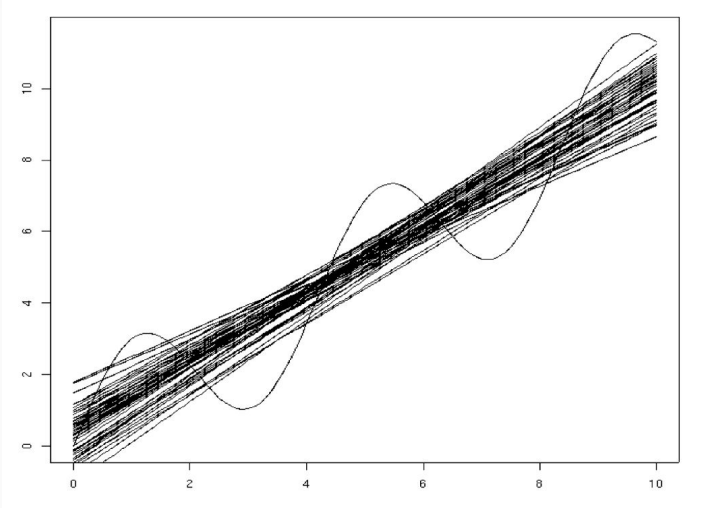
기계학습 모델의 Bias와 Variance



줄일 수 없는 오차

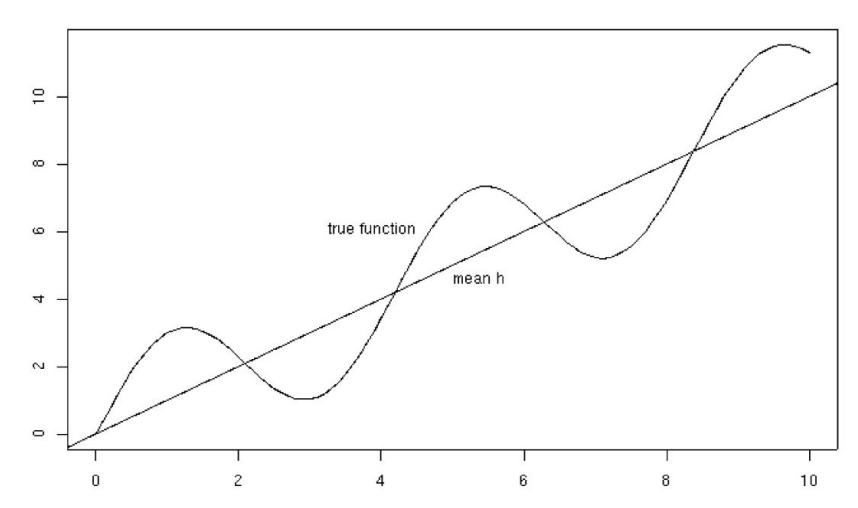
- 데이터는 y = x+2sin(1.5x)라는 식 f(x)에 줄일 수 없는 오차를 더한 값
- 여기서 만들어낸 데이터를 linear regression으로 학습시키는 상황

기계학습 모델의 Bias와 Variance



- 데이터를 여러 번 만들어내서 Linear Regression을 여러 번 해본 상황
- 여러 개 그어진 선들이 각 fitting된 모델을 뜻함

기계학습 모델의 Bias



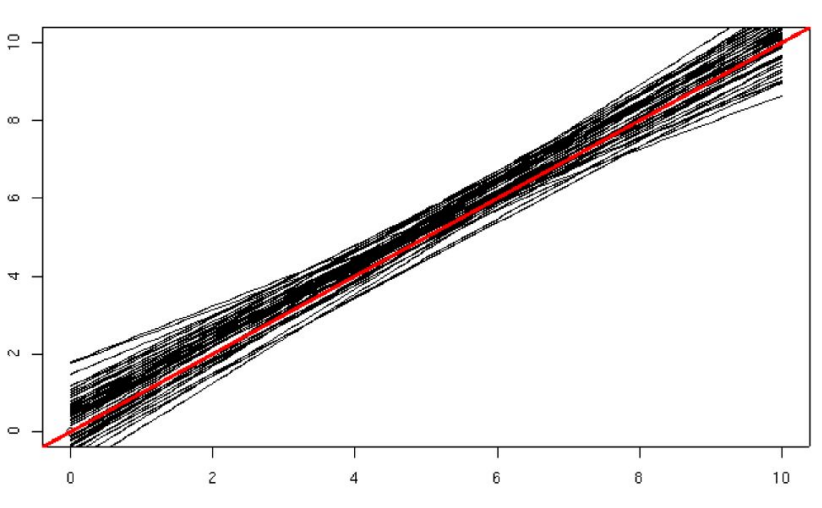
$y = x + 2\sin(1.5x)$

$$Bias[h(x)] = \overline{h(x)} - f(x)$$

모델

- 모든 '모델'을 평균을 내었을 때 다음과 같이 그려짐
- 여기서 Bias는 이 평균과 실제 함수와의 차이를 의미함

기계학습 모델의 Variance



$$Var[h(x)] = E_P\left[\left(h(x) - \overline{h(x)}\right)^2\right]$$

- Variance는 만들어진 여러 모델과 방금 구한 모델의 평균과의 차이의 제곱의 평균

MSE = Bias^2+Variance+Noise

Total Error = Bias^2 + Variance + Irreducible Error

$$E\left[\left(y - h(x)\right)^2\right]$$

$$= E[h(x)^2 - 2yh(x) + y^2]$$

$$= E[h(x)^2] + E[y^2] - 2E[y]E[h(x)]$$

$$= E\left[\left(h(x) - \overline{h(x)}\right)^2\right] + \overline{h(x)}^2 + E\left[\left(y - f(x)\right)^2\right] + f(x)^2 - 2f(x)\overline{h(x)}$$

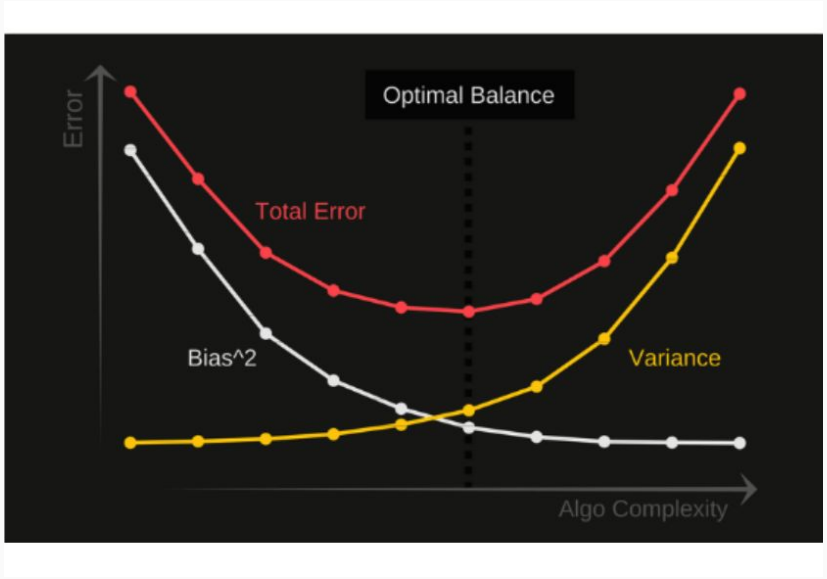
$$= E\left[\left(h(x) - \overline{h(x)}\right)^2\right] + \left(\overline{h(x)} - f(x)\right)^2 + E\left[\left(y - f(x)\right)^2\right]$$

Variance | Bias^2 | Noise

- 모델에서 구한 값과 실제 데이터와의 MSE는 위 식처럼 분해됨

모델의 구조 찾기

- 예측한 결과의 에러를 최소화할 수 있는 적당한 Bias와 Variance를 가진 모델을 찾아야 함
- 그러한 모델은 구조를 바꿔서 탐색해낼 수 있음
  - ex) DNN의 층수 바꾸기
- 하지만 여러 구조를 모두 하나씩 탐색하려면 너무나 오랜 시간이 걸림

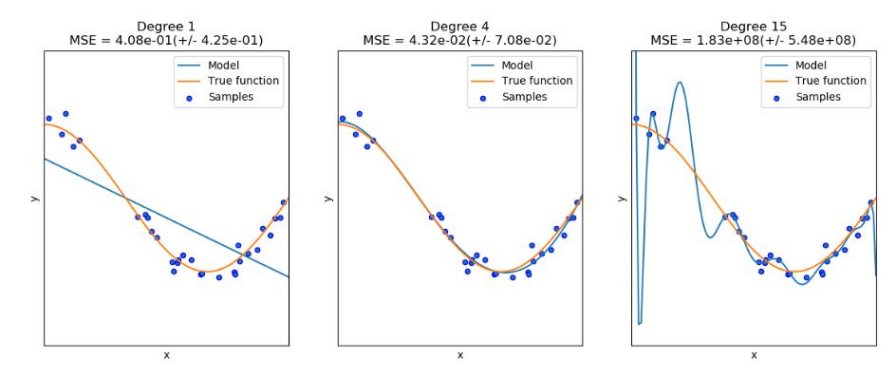


Bias-Variance Tradeoff

- Bias, Variance를 둘 다 줄이는 것이 이상적
- 즉, 우리가 하고싶은건 f(X)와 같은 모델을 찾는 것이지만 현실적으로 불가능
  - 모든 기계학습 모델을 사용해 결과를 분석하기엔 시간과 노력이 너무 많이 필요
  - 애초에 f(X)가 뭔지도 모르고 만약 안다면 모델링을 하는 이유가 없음
- 보통 어떤 모델을 선택해도 MSE는 바뀌지 않고 Bias, Variance 값만 바뀜
  - **Bias-Variance Tradeoff**
- 이미 가진 모델에서 모델 에러를 최소한으로 줄이는 방법이 필요
  - Overfitting이 일어난 모델의 훈련을 일부러 방해
  - Underfitting이 일어난 모델의 복잡도를 올리기

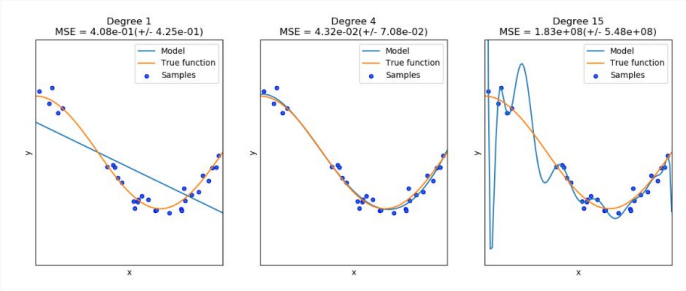
모델의 복잡도와 Bias, Variance

- 우리는 수학적 모델을 이용해 데이터 안에 있는 어떤 패턴을 알아내려고 함
- 하지만 데이터에는 항상 오류가 포함되어 있음
- 따라서 이러한 에러를 피해 그 안에 있는 일반적인 패턴을 잡아야 함



모델의 복잡도와 Bias, Variance

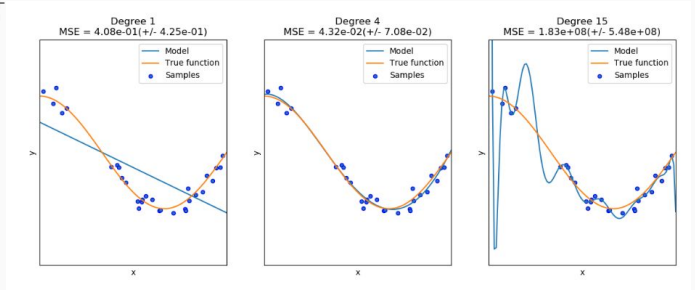
- Bias: 모델이 편향된 정도
  - 데이터의 모든 정보를 얼마나 고려하는지에 대한 정도
- Variance: 모델이 각 데이터에 얼마나 민감하게 반응하는지 나타내는 정도
  - 새로운 데이터에 대해 모델이 변화하는 정도
- 모델이 너무 단순하면 약간 복잡한 패턴도 잡아내지 못함
- 반대로 너무 복잡하면 오류를 패턴으로 인식함



- f(X) = sin(X)
- y = f(X) + error

모델의 복잡도와 Bias, Variance

- 왼쪽 모델은 선형 모델로 데이터의 모든 위치 정보를 고려하지 못함
- 하지만 새로운 데이터가 들어와도 이 직선은 크게 변하지 않음
  - 즉, Bias는 높지만 Variance는 낮은 상태
- 오른쪽 모델은 고차 함수 모델로 주어진 데이터를 매우 잘 맞추고 있음
- 하지만 새로운 데이터가 들어오면 모델 자체가 완전히 다른 형태로 변함
  - 또한, 데이터 자체의 에러도 학습해 overfitting이 발생함
  - 이 경우는 Bias는 낮고 Variance가 높음

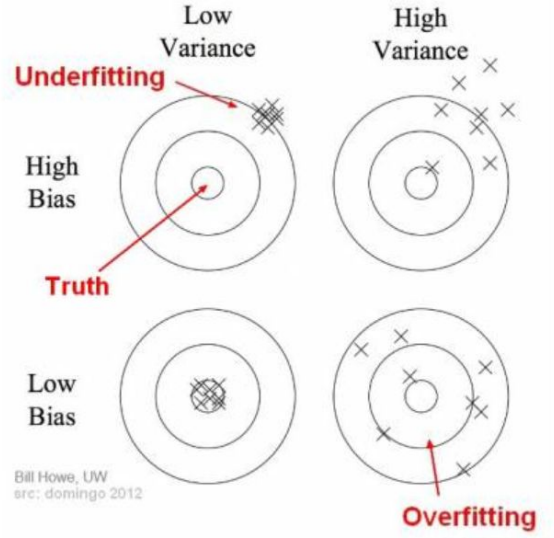


딥러닝의 모델의 Bias와 Variance

- 딥러닝에서 학습되는 파라미터 수가 많아질수록 다양한 훈련 데이터셋에 대해 잘 작동
- 그러나 Variance가 높아져 테스트 데이터셋에 대해 잘 작동하지 않음
  - 이는 훈련 데이터에 있는 오류까지 모두 모델링해버렸기 때문
- 하지만 우리가 알고 싶은 것은 그 데이터에 있는 규칙을 나타낸 진짜 함수임
- 파라미터가 많은 딥러닝은 Overfitting에 취약하고, 이를 해결하기 위해 Regularization을 사용하거나 데이터 수를 늘림
- Regularization은 Bias를 조금 증가시키는 대신 Variance를 크게 감소시켜 Overfitting을 방지함

Bias, Variance, Overfitting 정리

- 중앙: 실제 f(X)
- X표시: 모델 결과
- 중앙에서 멀어짐 = Bias 커짐
  - 애초에 모델이 실제 f(X)를 모델링하지 못함
  - Underfitting 발생
    - 모델 층 수 늘리기 등 파라미터를 늘려보기
- 산발적이다 = Variance 커짐
  - 모델이 너무 유연해서 모델 결과가 일정하지 못함
  - Overfitting 발생
    - 학습을 일부러 방해하는 **Regularizer** 필요



정규화 기법(Regularization)

- 과적합(Overfitting)을 막기 위해 학습을 방해하는 모든 기법 통칭
- 가중치가 너무 훈련 데이터에 과도하게 학습되는 현상을 방지

- 학습되는 가중치에 제한을 거는 정규화는 일반적인 방법
- Data augmentation을 통해 데이터 수를 늘려서도 방지 가능

L1, L2 정규화

- 손실함수에 모든 파라미터 크기에 대한 항을 추가함
- 이 항에 의해 파라미터 크기가 제한됨
- L1은 절대값을 사용
- L2는 제곱값을 사용

L1 regularization (**Lasso** regression)

$$L(x,y) \equiv \sum_{i=1}^{n}(y_i - h_\theta(x_i))^2 + \lambda \sum_{i=1}^{n}|\theta_i|$$

L2 regularization (**Ridge** regression)

$$L(x,y) \equiv \sum_{i=1}^{n}(y_i - h_\theta(x_i))^2 + \lambda \sum_{i=1}^{n}\theta_i^2$$

L1, L2 정규화

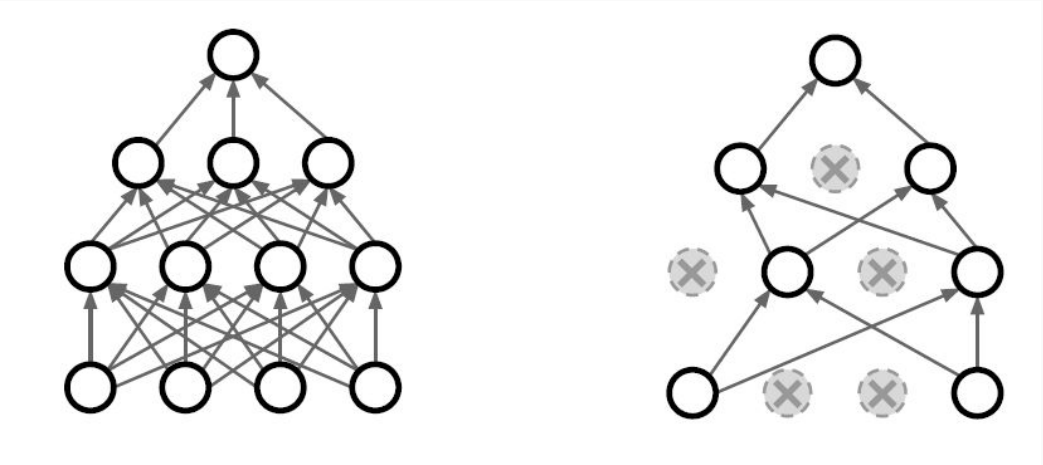
- L1은 파라미터를 Sparse하게 만드는 효과가 있음
  - Sparse하다 = 일부만 값을 가지고 나머지는 모두 0
- L2는 파라미터가 학습될 때마다 작아지도록 만듦

```
layer = keras.layers.Dense(100, activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))
```

```
from functools import partial

RegularizedDense = partial(keras.layers.Dense,
                           activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))

model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    RegularizedDense(300),
    RegularizedDense(100),
    RegularizedDense(10, activation="softmax",
                     kernel_initializer="glorot_uniform")
])
```

Dropout

- 순방향 전파 때 노드 일부를 지워버린 후 남은 노드에 대해서만 역전파를 진행해 파라미터를 업데이트함
- 여러 모델을 한꺼번에 학습하고 그 결과를 얻는 효과 (Ensemble model)
- 학습 이후 model.predict()시 원래 출력값에 지울 확률만큼 곱해서 사용



Dropout

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(10, activation="softmax")
])
```

Batch Normalization

- DNN 각 층에서 나오는 값들을 정규화(평균=0, 분산=1)해서 그 다음층에 넣어줌
- 각 층은 정규화된 값을 받아 다시 늘리고(scaling) 이동시킴(shifting)
  - 여기서 파라미터가 더 추가됨
- 훈련을 더 빠르게 하고 성능도 높아짐
- Regularization인 이유는 scaling과 shifting을 통해 데이터에 노이즈를 일부러 넣어 학습을 방해하기 때문

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$$

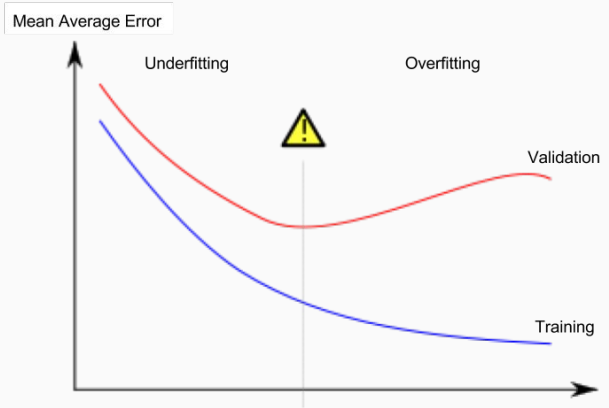
Batch Normalization

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(10, activation="softmax")
])
```

```
>>> model.summary()
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_3 (Flatten)          (None, 784)               0
_________________________________________________________________
batch_normalization_v2 (Batc (None, 784)               3136
_________________________________________________________________
dense_50 (Dense)             (None, 300)               235500
_________________________________________________________________
batch_normalization_v2_1 (Ba (None, 300)               1200
_________________________________________________________________
dense_51 (Dense)             (None, 100)               30100
_________________________________________________________________
batch_normalization_v2_2 (Ba (None, 100)               400
_________________________________________________________________
dense_52 (Dense)             (None, 10)                1010
=================================================================
Total params: 271,346
Trainable params: 268,978
Non-trainable params: 2,368
```

Early Stopping

- Validation loss나 평가지표를 보다가 올라간다 싶을 때 훈련을 멈추는 방법
- 훈련 횟수를 줄여 overfitting을 방지함



딥러닝의 한계 극복

- 지역적 극소(local minima)로 학습될 수 있음
  - Optimizer 선택을 통해 global minima를 탐색할 수 있도록 만들어줌
- 경사 소실
  - Saturation이 없는 활성화 함수(ex. Relu)를 이용해 경사값을 유지시켜 줌/가중치 초기화
- 데이터 부족
  - Transfer Learning을 사용해 원래 네트워크 일부만 훈련시킴
- 느린 학습
  - 좋은 GPU 사용하거나 파라미터 수가 적은 효과적인 네트워크 사용
- 과적합(overfitting)
  - Regularization, Early stopping, dropout, batch normalization 등의 방법으로 학습