

3주 2일차

딥러닝 기본 & 응용





강의 내용

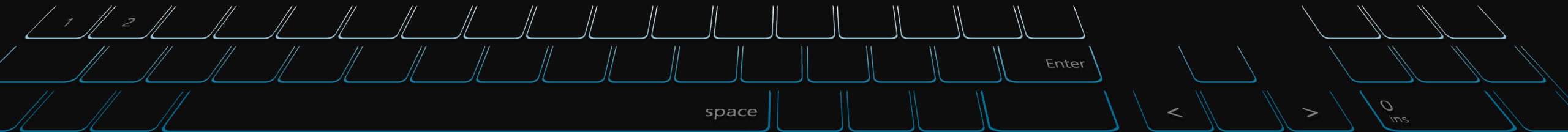
Contents of Lecture

기간	내용	실습
1주 1일차	Ipython 기초	문제 및 답안 코드 제공
1주 2일차	Numpy	문제 및 답안 코드 제공
1주 3일차	Pandas	문제 및 답안 코드 제공
1주 4일차	Matplotlib	문제 및 답안 코드 제공
1주 5일차	EDA	문제 및 답안 코드 제공
2주 1일차	통계 분포	문제 및 답안 코드 제공
2주 2일차	통계적 실험 & 유의성 검정	문제 및 답안 코드 제공
2주 3일차	인공지능 개념 & 예측모형	문제 및 답안 코드 제공
2주 4일차	회귀 및 분류	문제 및 답안 코드 제공
2주 5일차	분류 & 머신러닝 기본	문제 및 답안 코드 제공

기간	내용	과제
3주 1일차	머신러닝 응용 및 고급	문제 및 답안 코드 제공
3주 2일차	딥러닝 기본 & 응용	문제 및 답안 코드 제공
3주 3일차	고급 딥러닝	문제 및 답안 코드 제공
3주 4일차	알고보면 쓸모있는 신기한 기계학습	문제 및 답안 코드 제공
3주 5일차	MLOps란?	문제 및 답안 코드 제공
4주 1일차	MLOps 준비	문제 및 답안 코드 제공
4주 2일차	MLOps 구축 3단계 및 파이프라인 구축	문제 및 답안 코드 제공
4주 3일차	MLFlow 소개 및 활용	문제 및 답안 코드 제공
4주 4일차	AWS/Azure 배포	문제 및 답안 코드 제공
4주 5일차	Google 배포 및 Databricks 활용 & 최종 개별 프로젝트 발표	실습코드 제공 프로젝트 발표

DL 알고리즘 기본

- 1) 퍼셉트론, Keras, 모델 저장&복원, 콜백, 텐서보드 시각화,
- 2) 하이퍼 파라미터 튜닝 (은닉층, 은닉층의 뉴런, 학습률, 배치 크기 등)

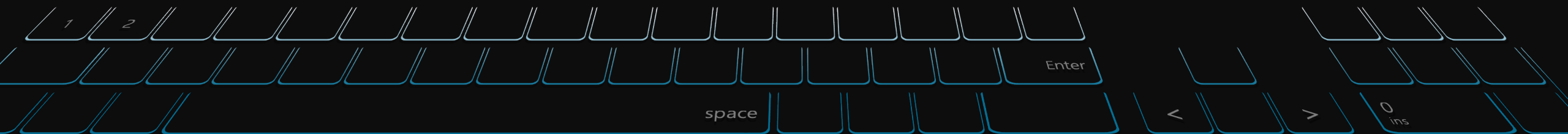




ML 알고리즘 기본 Contents

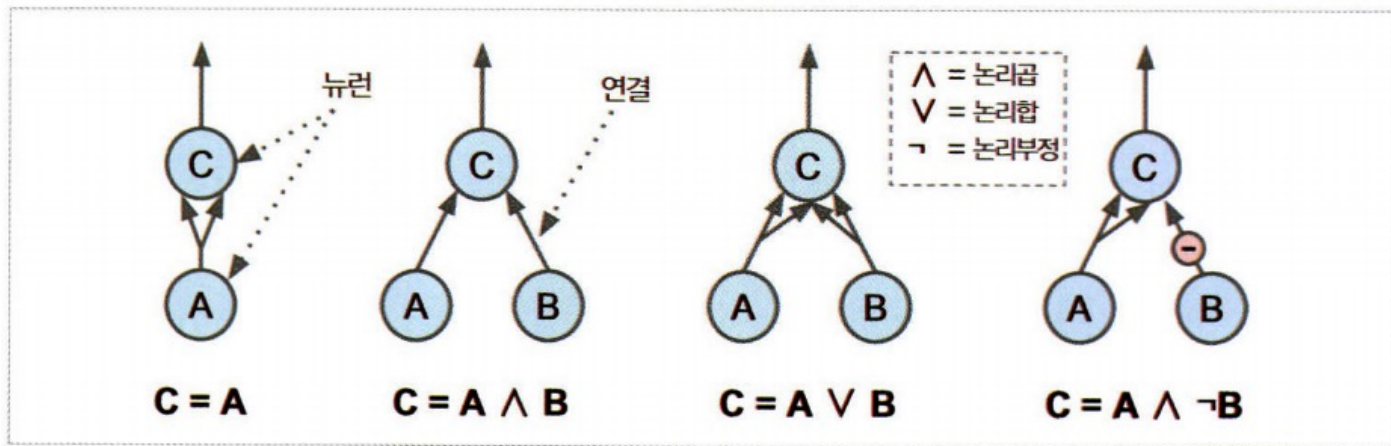
Machine Learning Algorithm Basic

1. 퍼셉트론
2. Keras API
 1. 학습 및 검증
 2. 예측
 3. Callback
 4. Tensorboard 시각화
 5. 하이퍼 파라미터 튜닝





- 생물학적 뉴런에서 착안한 매우 단순한 신경망 모델 인공 뉴런
- 하나 이상의 이진(on/off) 입력과 이진 출력 하나
- 입력이 일정 개수만큼 활성화되었을 때 출력을 내보낸다.





퍼셉트론

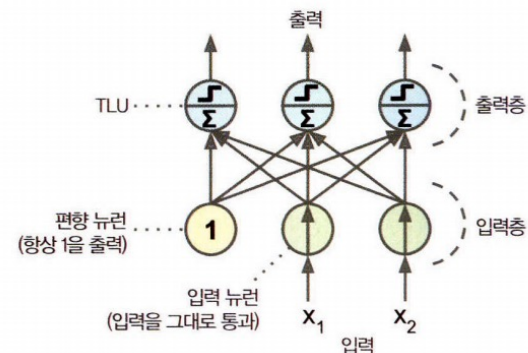
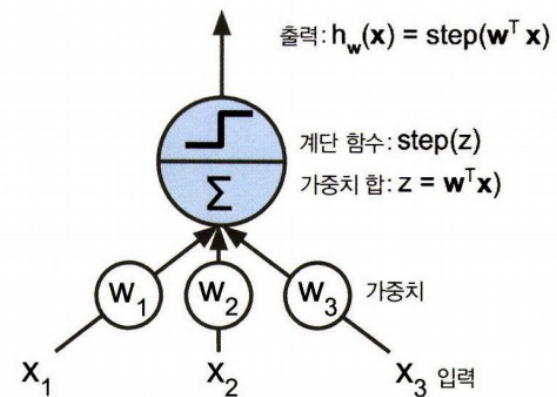
Perceptron

- 입력변수의 값들에 대한 가중합에 대한 활성화함수를 적용하여 최종 결과물 생성
- TLU(threshold logic unit) 또는 LTU(linear threshold unit)이라고 불림
- 퍼셉트론은 층이 하나뿐인 TLU로 구성
- 한 층에 있는 모든 뉴런이 이전 층의 모든 뉴런과 연결되어 있을 때 이를 완전 연결층(fully connected layer) 또는 밀집 층(dense layer) 라고 부름

계단함수

- 가장 많이 사용되는 계단함수: Heaviside 계단함수와 sign 함수

$$\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \quad \text{sgn}(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ 1 & \text{if } z > 0 \end{cases}$$





퍼셉트론

Perceptron

- 완전 연결 층의 출력 계산 방식

$$h_{W,b}(X) = \phi(XW + b)$$

- 퍼셉트론 학습 규칙 = **W**eight(가중치 업데이트)

$$w_{i,j}^{(next\ step)} = w_{i,j} + \eta(y_j - \hat{y}_j)x_j$$

- $w_{i,j}$ 는 i 번째 입력 뉴런과 j 번째 출력 뉴런 사이를 연결하는 가중치
- x_i 는 현재 훈련 샘플의 i 번째 뉴런의 입력값
- \hat{y}_j 는 현재 훈련 샘플의 j 번째 출력 뉴런의 출력값
- y_j 는 현재 훈련 샘플의 j 번째 출력 뉴런의 타깃값
- η 는 학습률

- 각 출력 뉴런의 결정 경계는 선형이므로 퍼셉트론 또한 복잡한 패턴을 학습하지 못 함
- 훈련 샘플이 선형적으로 구분될 수 있다면, 이 알고리즘이 정답에 수렴함

=> 퍼셉트론 수렴이론





퍼셉트론

Perceptron

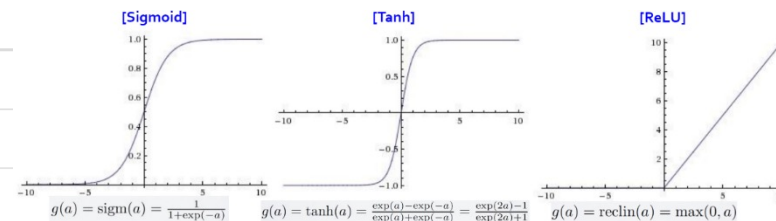
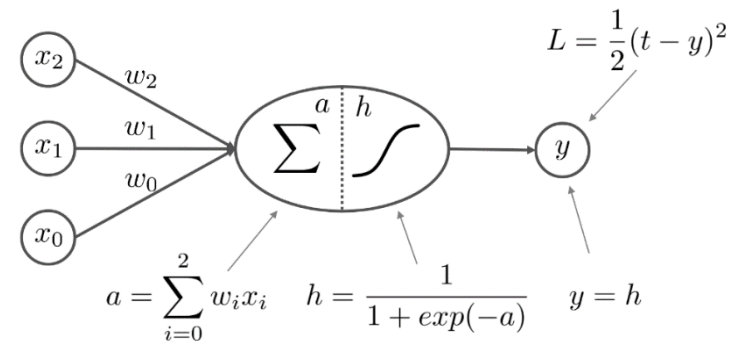
1. 입력 노드(input node)

- 우리가 알고 있는 개별 설명 변수가 각각의 입력 노드에 해당

2. 은닉 노드(hidden node)

- 개별 설명변수들의 값을 취합(선형 결합)하여 비선형 변환(활성화)을 수행
 - 활성화 함수의 역할
 - 각 노드가 이전 노드들로부터 전달받은 정보를 다음 노드에 얼마만큼 전달해 줄 것인가를 결정
 - 대표적인 활성화 함수

활성화 함수	설명
Sigmoid	가장 일반적으로 사용되는 활성화 함수, [0, 1]의 범위를 가지며 학습 속도가 상대적으로 느림
Tanh	활성화함수와 형태는 유사하나 [-1, 1]의 범위를 가져 학습 속도가 상대적으로 빠름
ReLU	학습속도가 매우 빠르며 상대적으로 계산이 쉬움(지수함수 형태를 사용하지 않는다)





퍼셉트론

Perceptron

3. 출력 노드(output node)

- 은닉 노드에서 생성된 값을 그대로 받아들임(다층 퍼셉트론에서는 여러 은닉 노드에서 정보 취합)

4. 목적함수

- 퍼셉트론의 목적: 주어진 학습 데이터의 입력 정보와 출력 정보의 관계를 잘 찾도록 가중치를 조절
- 현재 퍼셉트론의 결과물이 실제 정답에 얼마나 가까운지를 손실 함수(loss function)를 통해 측정
- 전체 데이터 셋에 대해서 현재 퍼셉트론이 얼마나 잘못하고 있는지는 비용 함수(cost function)을 사용



퍼셉트론

Perceptron

퍼셉트론

노트: 사이킷런 향후 버전에서 `max_iter` 와 `tol` 매개변수의 기본값이 바뀌기 때문에 경고를 피하기 위해 명시적으로 지정합니다.

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

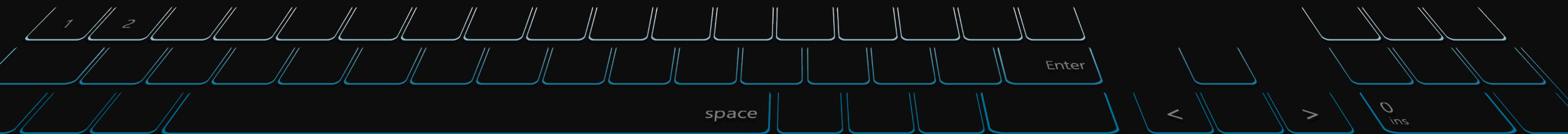
iris = load_iris()
X = iris.data[:, (2, 3)] # 꽃잎 길이, 꽃잎 너비
y = (iris.target == 0).astype(np.int)

per_clf = Perceptron(max_iter=1000, tol=1e-3, random_state=42)
per_clf.fit(X, y)

y_pred = per_clf.predict([[2, 0.5]])
```

```
y_pred
```

```
array([1])
```





퍼셉트론

Perceptron

```
In [4]: a = -per_clf.coef_[0][0] / per_clf.coef_[0][1]
b = -per_clf.intercept_ / per_clf.coef_[0][1]

axes = [0, 5, 0, 2]

x0, x1 = np.meshgrid(
    np.linspace(axes[0], axes[1], 500).reshape(-1, 1),
    np.linspace(axes[2], axes[3], 200).reshape(-1, 1),
)
X_new = np.c_[x0.ravel(), x1.ravel()]
y_predict = per_clf.predict(X_new)
zz = y_predict.reshape(x0.shape)

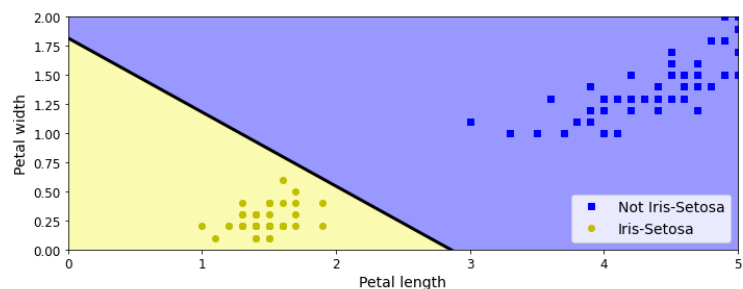
plt.figure(figsize=(10, 4))
plt.plot(X[y==0, 0], X[y==0, 1], "bs", label="Not Iris-Setosa")
plt.plot(X[y==1, 0], X[y==1, 1], "yo", label="Iris-Setosa")

plt.plot([axes[0], axes[1]], [a * axes[0] + b, a * axes[1] + b], "k-", linewidth=3)
from matplotlib.colors import ListedColormap
custom_cmap = ListedColormap(['#9999ff', '#fafab0'])

plt.contourf(x0, x1, zz, cmap=custom_cmap)
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.legend(loc="lower right", fontsize=14)
plt.axis(axes)

save_fig("perceptron_iris_plot")
plt.show()
```

그림 저장: perceptron_iris_plot

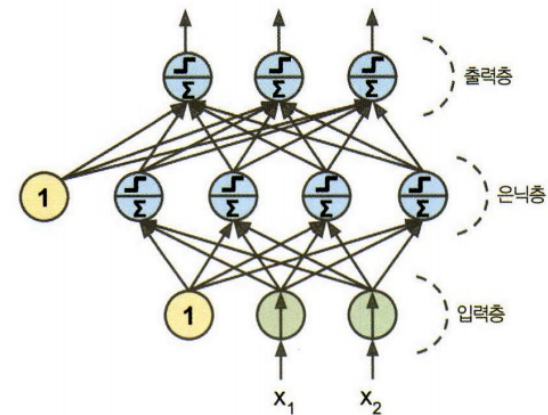




다층 퍼셉트론

Multi-layer Perceptron (MLP)

- 문제를 한꺼번에 풀지 말고 풀 수 있는 형태의 여러 개 문제로 나누어 풀자
- 입력층 하나와 은닉층이라 불리는 하나 이상의 TLU층과 마지막 출력층으로 구성
- 입력층과 가까운 층을 보통 하위 층, 출력에 가까운 층을 상위 층
- 은닉층을 여러 개 쌓아 올린 인공 신경망을 심층 신경망(deep neural network, **DNN**)이라고 한다.

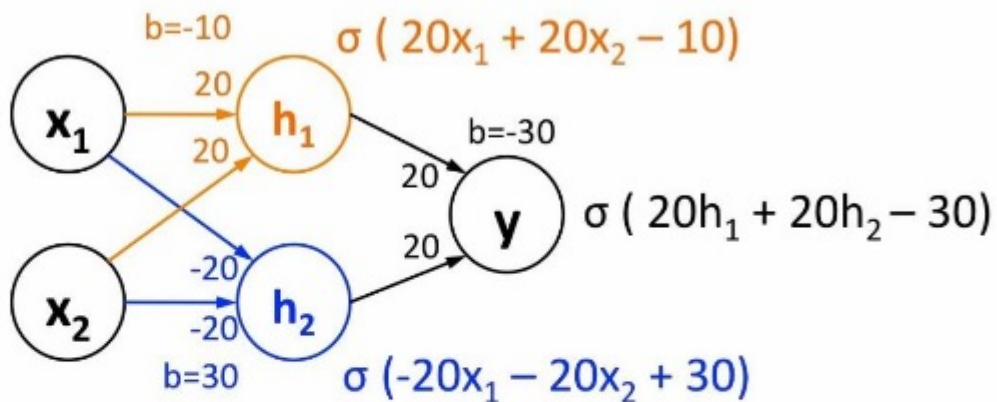
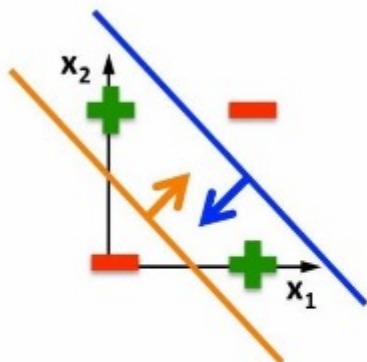




다층 퍼셉트론

Multi-layer Perceptron (MLP)

Linear classifiers
cannot solve this



$\sigma(20*0 + 20*0 - 10) \approx 0$	$\sigma(-20*0 - 20*0 + 30) \approx 1$	$\sigma(20*0 + 20*1 - 30) \approx 0$
$\sigma(20*1 + 20*1 - 10) \approx 1$	$\sigma(-20*1 - 20*1 + 30) \approx 0$	$\sigma(20*1 + 20*0 - 30) \approx 0$
$\sigma(20*0 + 20*1 - 10) \approx 1$	$\sigma(-20*0 - 20*1 + 30) \approx 1$	$\sigma(20*1 + 20*1 - 30) \approx 1$
$\sigma(20*1 + 20*0 - 10) \approx 1$	$\sigma(-20*1 - 20*0 + 30) \approx 1$	$\sigma(20*1 + 20*1 - 30) \approx 1$



다층 퍼셉트론

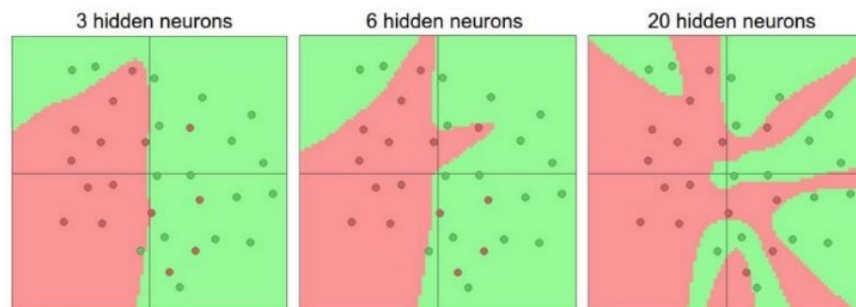
Multi-layer Perceptron (MLP)

- 다층 퍼셉트론의 예측력이 우수한 이유

구분	로지스틱 회귀분석	의사결정나무	인공신경망
선의 수	1개	제한 없음	사용자 지정 (은닉층 및 노드의 수)
선의 방향	제약 없음	축에 수직	제약 없음

- 은닉 노드의 역할

- 은닉노드의 수가 인공신경망 복잡도(complexity)를 결정
- 은닉노드가 많을수록 임의의 분류 경계면을 찾거나(분류 문제) 굴곡이 많은 함수를 추정(회귀 문제)할 수 있음





다층 퍼셉트론

Multi-layer Perceptron (MLP)

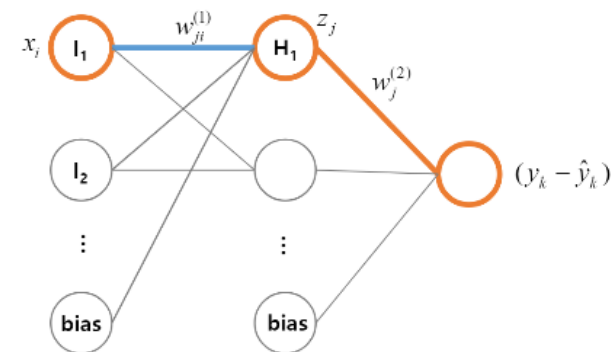
• 역전파 알고리즘 (Backpropagation Algorithm)

1. 각 훈련 샘플에 대해 역전파 알고리즘이 먼저 예측을 만든다.(정방향 계산)
2. 역방향으로 각 층을 거치면서 각 연결이 오차에 기여한 정도를 측정(역방향 계산)

- k번째 관측치의 오차 $Err_k = \frac{1}{2}(y_k - \hat{y}_k)^2$, $\hat{y}_k = \sum_{j=1}^{p+1} w_j^{(2)} g(\sum_{i=1}^{d+1} w_{ji}^{(1)} x_i)$

3. 오차가 감소하도록 가중치를 조정(경사 하강법)

- j번째 은닉 노드와 출력 노드를 연결하는 가중치 w_j 의 변화량
- J번째 은닉 노드와 i번째 입력 노드를 연결하는 가중치 w_{ji} 의 변화량



$$\frac{\partial Err_k}{\partial w_j^{(2)}} = \frac{\partial Err_k}{\partial \hat{y}_k} \cdot \frac{\partial \hat{y}_k}{\partial w_j^{(2)}} = (y_k - \hat{y}_k) \cdot z_j$$

$$\frac{\partial Err_k}{\partial w_{ji}^{(1)}} = \frac{\partial Err_k}{\partial \hat{y}_k} \cdot \frac{\partial \hat{y}_k}{\partial z_j} \cdot \frac{\partial z_j}{\partial h_j} \cdot \frac{\partial h_j}{\partial w_{ji}^{(1)}} = (y_k - \hat{y}_k) \cdot w_j^{(2)} \cdot z_j \cdot (1 - z_j) \cdot x_i$$

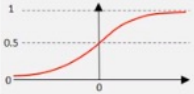
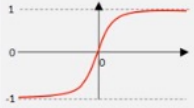



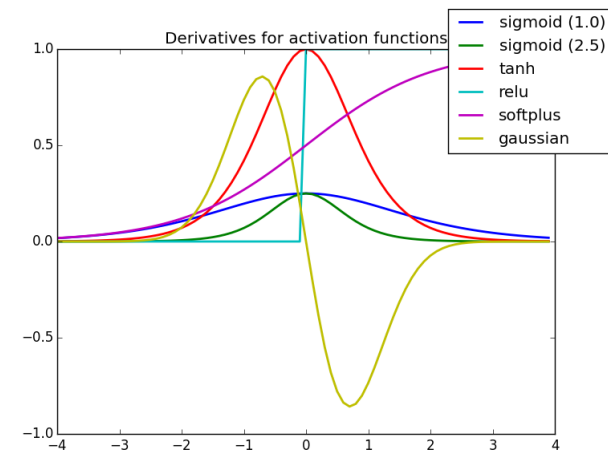


다층 퍼셉트론

Multi-layer Perceptron (MLP)

- 활성화 함수
 - Softplus
 - Softmax
 - ReLU
 - Tanh
 - Sigmoid

Name	Formula	Derivative	Graph	Range
sigmoid (logistic function)	$\sigma(a) = \frac{1}{1+e^{-a}}$	$\frac{\partial \sigma(a)}{\partial a} = \sigma(a)(1 - \sigma(a))$		(0,1)
TanH (hyperbolic tangent)	$\tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$	$\frac{\partial \tanh(a)}{\partial a} = \frac{4}{(e^a + e^{-a})^2}$		(-1,1)
ReLu (rectified linear unit)	$\text{relu}(a) = \max(0, a)$	$\frac{\partial \text{relu}(a)}{\partial a} = \begin{cases} 0, & \text{if } a \leq 0 \\ 1, & \text{if } a > 0 \end{cases}$		(0,∞)
softmax	$\sigma_i(\mathbf{a}) = \frac{e^{a_i}}{\sum_j e^{a_j}}$	$\frac{\partial \sigma_i(\mathbf{a})}{\partial a_j} = \sigma_i(\mathbf{a}) (\delta_{ij} - \sigma_j(\mathbf{a}))$ Where δ_{ij} is 1 if $i=j$, 0 otherwise	?	(0,1)

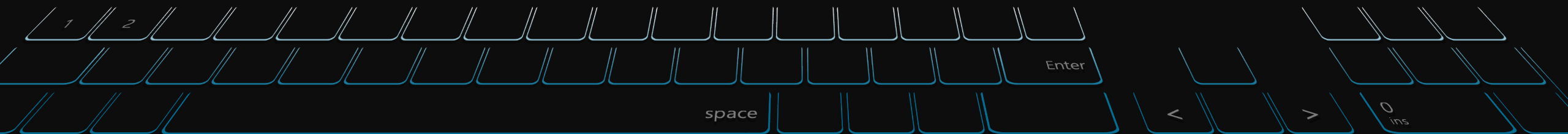




회귀를 위한 다층 퍼셉트론

Multi-layer Perceptron (MLP)

- 출력 뉴런에 활성화 함수를 사용하지 않고 어떤 범위의 값도 출력되도록 한다.
- 항상 양수여야 한다면 출력층에 ReLU 활성화 함수를 사용할 수 있다.
- 또는 softplus 활성화 함수를 사용하여 z 가 음수 일 때 0에 가까워지고 큰 양수 일수록 z 에 가깝게 할 수 있다.
- 어떤 범위 한의 값을 예측하고 싶다면 로지스틱 함수나 하이퍼볼릭 탄젠트 함수를 사용하고 레이블의 스케일을 적절한 범위로 조정할 수 있다.
- 훈련에 사용하는 손실 함수는 전형적으로 평균 제곱 오차(MSE)이다.
- 하지만, 훈련 세트에 이상치가 많다면 평균 절댓값 오차(MAE)를 사용할 수 있다.(또는 둘을 조합한 후버(Huber)손실 사용)





회귀를 위한 다층 퍼셉트론

Multi-layer Perceptron (MLP)

- 회귀 MLP의 전형적인 구조

하이퍼파라미터	일반적인 값
입력 뉴런 수	특성마다 하나
은닉층 수	문제에 따라 다름, 일반적으로 1에서 5 사이
은닉층의 뉴런 수	문제에 따라 다름, 일반적으로 10에서 100사이
출력 뉴런 수	예측 차원마다 하나
은닉층의 활성화 함수	ReLU(또는 SELU)
출력층의 활성화 함수	없음, 또는 (출력이 양수일 때) ReLU/softplus 나 (출력을 특정 범위로 제한할 때) logistic/tanh 사용
손실 함수	MSE나 (이상치가 있다면) MAE/Huber

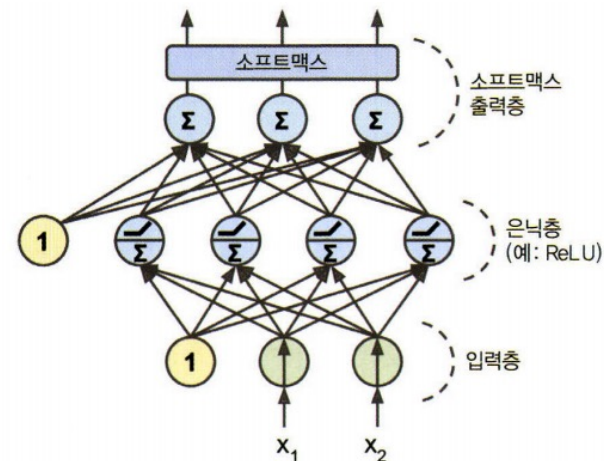




분류를 위한 다층 퍼셉트론

Multi-layer Perceptron (MLP)

- 이진 분류
 - 로지스틱 활성화 함수를 가진 출력 뉴런 하나 필요
 - 출력은 0과 1사이의 실수로 이를 양성 클래스에 대한 예측 확률로 해석
- 다중 레이블 이진 분류
 - 로지스틱 활성화 함수를 가진 출력 뉴런 여러 개로 다중 레이블 분류 가능
 - 각 샘플이 3개 이상의 클래스 중 한 클래스만 속하여야 한다면 출력층에 소프트 맥스 활성화 함수 사용
 - 모든 예측 확률을 0과 1사이로 만들고 더했을 때 1이 되도록 한다. 이를 다중 분류라고 함.
 - 손실 함수로는 일반적으로 크로스 엔트로피 손실(또는 로그 손실) 활용



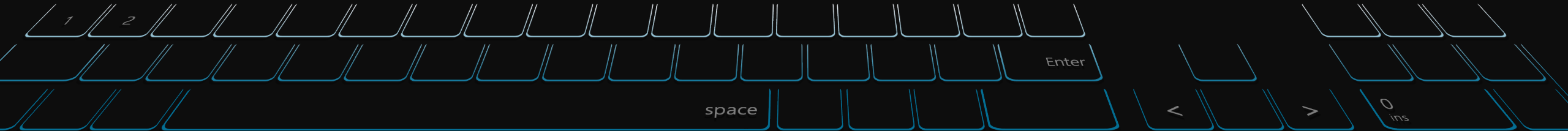


분류를 위한 다층 퍼셉트론

Multi-layer Perceptron (MLP)

- 분류 MLP의 전형적인 구조

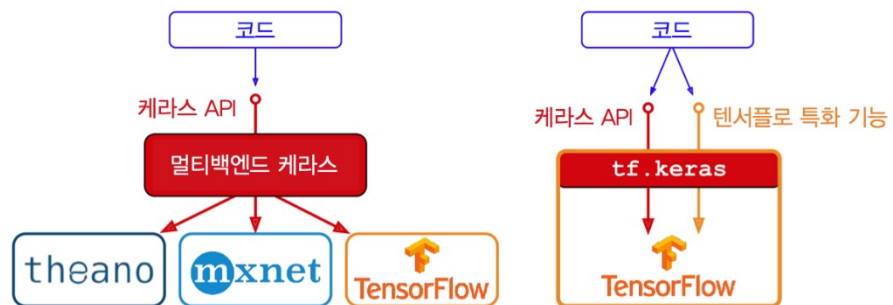
하이퍼파라미터	이진 분류	다중 레이블 분류	다중 분류
입력층과 은닉층	회귀와 동일	회귀와 동일	회귀와 동일
출력 뉴런 수	1개	레이블마다 1개	클래스마다 1개
출력층의 활성화 함수	로지스틱 함수	로지스틱 함수	소프트맥스 함수
손실 함수	크로스 엔트로피	크로스 엔트로피	크로스 엔트로피





Keras API

- Keras로 다층 퍼셉트론 구현하기



이미지 분류기 만들기 📌

먼저 텐서플로와 케라스를 임포트합니다.

```
In [9]: import tensorflow as tf
        from tensorflow import keras
```

```
In [10]: tf.__version__
```

```
Out[10]: '2.4.1'
```

```
In [11]: keras.__version__
```

```
Out[11]: '2.4.0'
```



Keras API

- Keras로 Sequence API 사용하여 Model 만들기
 - 활용 데이터



먼저 MNIST 데이터셋을 로드하겠습니다. 케라스는 `keras.datasets`에 널리 사용하는 데이터셋을 로드하기 위한 함수를 제공합니다. 이 데이터셋은 이미 훈련 세트와 테스트 세트로 나누어져 있습니다. 훈련 세트를 더 나누어 검증 세트를 만드는 것이 좋습니다:

```
fashion_mnist = keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()
```

훈련 세트는 60,000개의 흑백 이미지입니다. 각 이미지의 크기는 28x28 픽셀입니다:

```
X_train_full.shape
```

```
(60000, 28, 28)
```

각 픽셀의 강도는 바이트(0~255)로 표현됩니다:





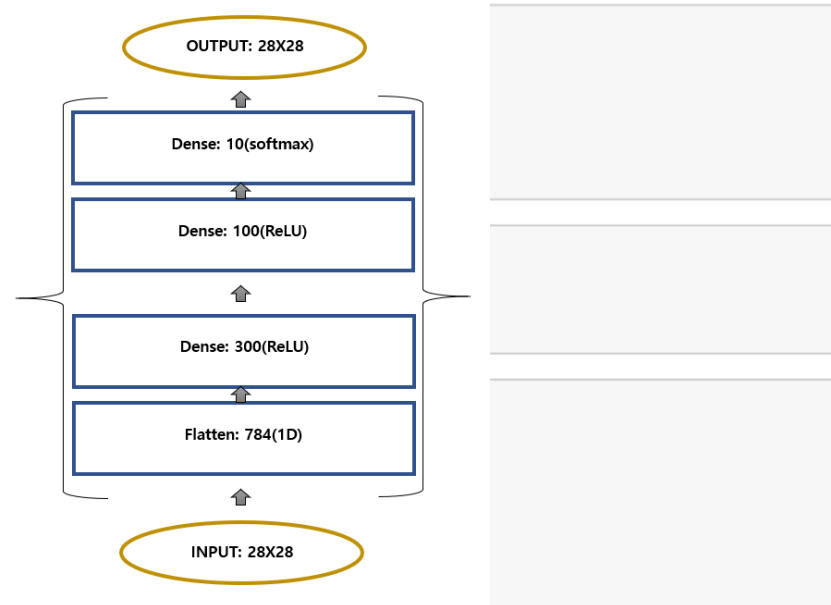
Keras API

- Keras로 Sequence API 사용하여 Model 만들기

```
model = keras.models.Sequential()  
model.add(keras.layers.Flatten(input_shape=[28, 28]))  
model.add(keras.layers.Dense(300, activation="relu"))  
model.add(keras.layers.Dense(100, activation="relu"))  
model.add(keras.layers.Dense(10, activation="softmax"))
```

```
keras.backend.clear_session()  
np.random.seed(42)  
tf.random.set_seed(42)
```

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





Keras API

- Keras로 Sequence API 사용하여 Model 만들기 (세부 설명)
 - 다음은 두개의 은닉층으로 이루어진 분류용 다층 퍼셉트론이다.
 - 첫 번째 라인은 Sequential 모델을 만든다. 이 모델은 가장 간단한 케라스의 신경망 모델이다. 순서대로 연결된 층을 일렬로 쌓아서 구성한다. 이를 시퀀스 API라고 부른다.
 - 그다음 첫 번째 층을 만들고 모델에 추가한다. Flatten 층은 기존의 28X28 이미지를 784의 1차원 배열로 변환한다. 이 층은 보다시피 어떤 모델 파라미터도 가지지 않고 간단한 전처리를 수행한다. 그리고 모델의 첫번째 층이므로 input_shape = [28,28]로 지정해준다. 이때 input_shape에는 배치 크기(가중치를 한 번 업데이트시킬 때마다 사용되는 샘플들의 묶음)를 제외하고 샘플의 크기만 써야 한다.
 - 그다음 뉴런 300개를 가진 Dense 은닉층을 추가한다. 이 층은 ReLU 활성화 함수를 사용한다. Dense층마다 각자 가중치 행렬을 관리한다. 이 행렬에는 층의 뉴런과 입력 사이의 모든 연결 가중치가 포함된다. 또한 (뉴런마다 하나씩 있는) 편향도 벡터로 관리한다. 이 층은 입력 데이터를 받으면 그 전 층의 입력 데이터와 가중치 행렬의 곱을 편향과 더해 활성화 함수를 통과하게 된다.
 - 다음 뉴런 100개를 가진 두 번째 Dense 은닉층을 추가한다. 역시 ReLU 활성화 함수를 사용한다.
 - 마지막으로 (클래스마다 하나씩) 뉴런 10개를 가진 Dense 출력층을 추가한다.





Keras API

- Keras로 Sequence API 사용하여 Model 만들기
 - 모든 층 구조 출력하기

```
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 300)	235500
dense_1 (Dense)	(None, 100)	30100
dense_2 (Dense)	(None, 10)	1010
Total params: 266,610		
Trainable params: 266,610		
Non-trainable params: 0		



Keras API

- Keras로 Sequence API 사용하여 Model 만들기
 - 모든 층 구조 출력하기

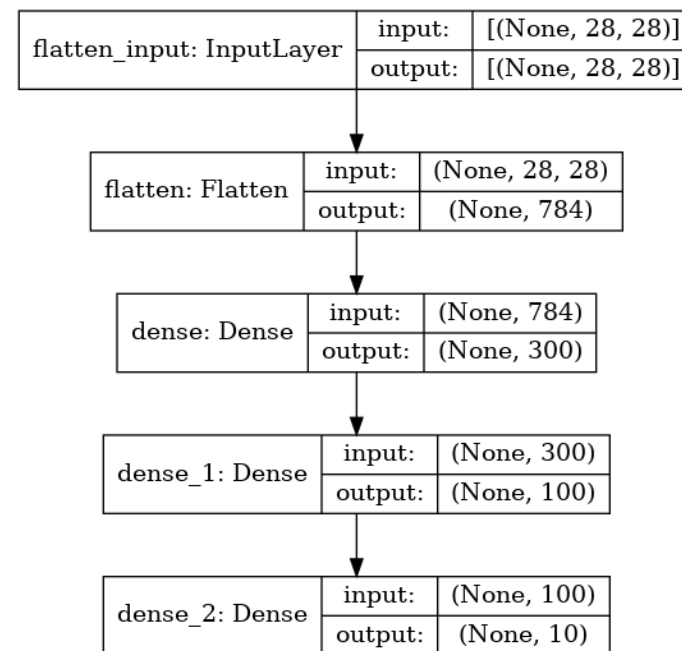
```
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 300)	235500
dense_1 (Dense)	(None, 100)	30100
dense_2 (Dense)	(None, 10)	1010
Total params: 266,610		
Trainable params: 266,610		
Non-trainable params: 0		

- 첫 번째 dense (Dense) 층에서 파라미터 개수가 235500개가 나온 이유는 784개의 입력에 대하여 300개의 가중치가 곱해지고 300개의 편향이 더해져 나온다. $(784 * 300 + 300)$

```
keras.utils.plot_model(model, "my_fashion_mnist_model.png", show_shapes=True)
```





- 실제 Weight / Bias 보는 법

```
hidden1 = model.layers[1]
hidden1.name

: 'dense'

: model.get_layer(hidden1.name) is hidden1

: True

: weights, biases = hidden1.get_weights()

: weights

: array([[ 0.02448617, -0.00877795, -0.02189048, ..., -0.02766046,
          0.03859074, -0.06889391],
        [ 0.00476504, -0.03105379, -0.0586676 , ...,  0.00602964,
        -0.02763776, -0.04165364],
        [-0.06189284, -0.06901957,  0.07102345, ..., -0.04238207,
          0.07121518, -0.07331658],
        ...,
        [-0.03048757,  0.02155137, -0.05400612, ..., -0.00113463,
          0.00228987,  0.05581069],
        [ 0.07061854, -0.06960931,  0.07038955, ..., -0.00384101,
          0.00034875,  0.02878432],
        [-0.06022581,  0.01577859, -0.02585464, ..., -0.00527829,
          0.00272203, -0.06793761]], dtype=float32)

: weights.shape

: (784, 300)
```

```
biases
```

```
: array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.] , dtype=float32)
```

```
biases.shape
```

```
: (300.)
```



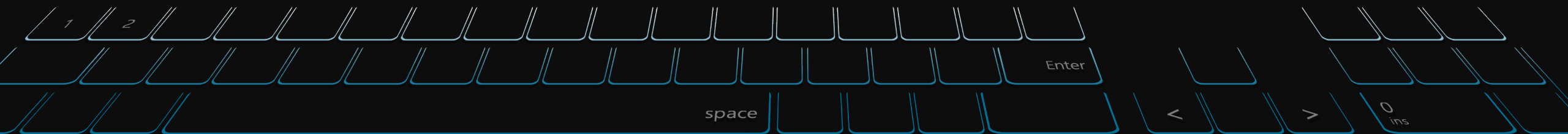
Keras API

- Keras로 Sequence API 사용하여 Model 만들기
 - Model Compile 하는 법 (손실 함수, 옵티마이저 지정)

```
model.compile(loss="sparse_categorical_crossentropy",  
              optimizer="sgd",  
              metrics=["accuracy"])
```

위 코드는 다음과 같습니다:

```
model.compile(loss=keras.losses.sparse_categorical_crossentropy,  
              optimizer=keras.optimizers.SGD(),  
              metrics=[keras.metrics.sparse_categorical_accuracy])
```





Keras API

- Keras로 Sequence API 사용하여 Model 만들기
 - Model Compile 하는 법
 - 손실함수
 - 먼저 샘플들의 정답을 나타내는 레이블은 0에서 9까지의 정수로 이루어져 있다. 각 샘플들에 대하여 클래스들이 배타적(독립적)이므로 손실 함수를 "sparse_categorical_crossentropy"(다중 분류 손실 함수)를 사용한다.
 - 배타적이지 않는 경우(ex. One-Hot-Vector[0., 0., 0., 1., 0., 0., 0., 0., 0.])에 대하여 "categorical_crossentropy"를 사용한다.
 - 두 손실 함수는 계산 과정이 같아 실측값은 같게 나오기 때문에 클래스의 배타성 여부에 따라 편한 손실 함수를 선택하면 된다. 대신 다중 분류 (Multiclass classification) 문제를 해결하기 위해 출력층에 "softmax"를 넣어 줘야 한다.
 - 만약 (이진 레이블에 대한) 이진 분류를 수행한다면 출력층에 "softmax" 대신 "sigmoid"함수를 사용하고 "binary_crossentropy" 손실을 사용해야 한다.

	label	loss
binary_crossentropy	0 or 1	sigmoid cross entropy
categorical_crossentropy	[0, 1] or [1, 0]	softmax cross entropy
sparse_categorical_crossentropy	0 or 1	softmax cross entropy



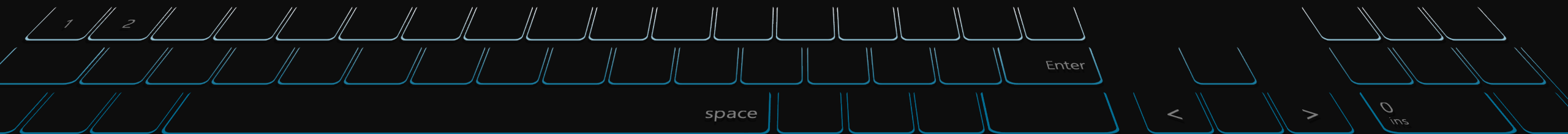


Keras API

- Keras로 Sequence API 사용하여 Model 만들기
 - Model 훈련 및 평가

```
history = model.fit(X_train, y_train, epochs=30,  
                    validation_data=(X_valid, y_valid))
```

```
Epoch 1/30  
1719/1719 [=====] - 7s 3ms/step - loss: 1.0187 - accuracy: 0.6807 - val_loss  
Epoch 2/30  
1719/1719 [=====] - 5s 3ms/step - loss: 0.5028 - accuracy: 0.8260 - val_loss  
Epoch 3/30  
1719/1719 [=====] - 5s 3ms/step - loss: 0.4485 - accuracy: 0.8423 - val_loss  
Epoch 4/30  
1719/1719 [=====] - 5s 3ms/step - loss: 0.4210 - accuracy: 0.8529 - val_loss  
Epoch 5/30  
1719/1719 [=====] - 5s 3ms/step - loss: 0.4061 - accuracy: 0.8576 - val_loss  
Epoch 6/30  
1719/1719 [=====] - 5s 3ms/step - loss: 0.3756 - accuracy: 0.8670 - val_loss  
Epoch 7/30  
1719/1719 [=====] - 5s 3ms/step - loss: 0.3654 - accuracy: 0.8711 - val_loss  
Epoch 8/30  
1719/1719 [=====] - 5s 3ms/step - loss: 0.3481 - accuracy: 0.8761 - val_loss  
Epoch 9/30  
1719/1719 [=====] - 5s 3ms/step - loss: 0.3486 - accuracy: 0.8757 - val_loss  
Epoch 10/30  
1719/1719 [=====] - 5s 3ms/step - loss: 0.3298 - accuracy: 0.8831 - val_loss
```



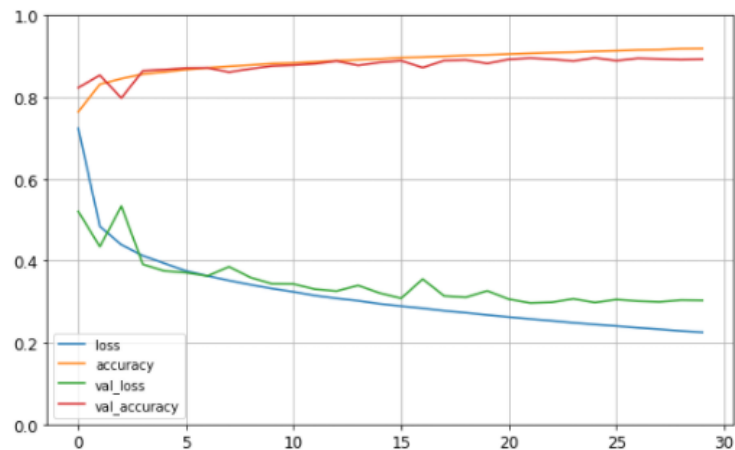


Keras API

- Keras로 Sequence API 사용하여 Model 만들기
 - Model 훈련 및 평가 (Train / Validation Dataset 기준 + Test Dataset 기준)

```
import pandas as pd  
  
pd.DataFrame(history.history).plot(figsize=(8, 5))  
plt.grid(True)  
plt.gca().set_ylim(0, 1)  
save_fig("keras_learning_curves_plot")  
plt.show()
```

그림 저장: keras_learning_curves_plot



```
: model.evaluate(X_test, y_test)
```

313/313 [=====] - 1s 2ms/step - loss: 0.3367 - accuracy: 0.8827

```
: [0.3366743326187134, 0.8827000260353088]
```



Keras API

- Keras로 Sequence API 사용하여 Model 만들기
 - Model을 사용한 예측

```
X_new = X_test[:3]
y_proba = model.predict(X_new)
y_proba.round(2)
```

```
array([[0. , 0. , 0. , 0. , 0. , 0.01, 0. , 0.03, 0. , 0.96],
       [0. , 0. , 0.99, 0. , 0.01, 0. , 0. , 0. , 0. , 0. ],
       [0. , 1. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ]],
      dtype=float32)
```

경고: `model.predict_classes(X_new)` 는 삭제될 예정입니다. 대신 `np.argmax(model.predict(X_new), axis=-1)` 를 사용하세요.

```
# y_pred = model.predict_classes(X_new)
y_pred = np.argmax(model.predict(X_new), axis=-1)
y_pred
```

```
array([9, 2, 1])
```

```
np.array(class_names)[y_pred]
```

```
array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')
```

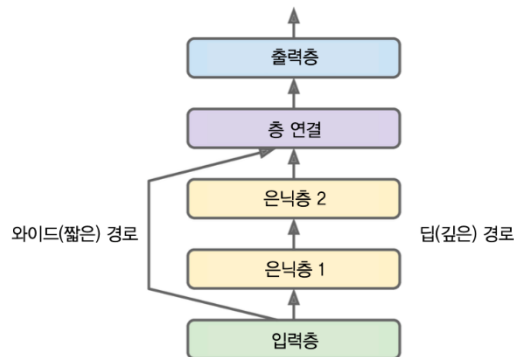
```
y_new = y_test[:3]
y_new
```

```
array([9, 2, 1], dtype=uint8)
```




Keras API

- Keras로 Sequence API 사용하여 Model 만들기
 - 함수형 API 기반 복잡한 Model 만들기



```
input_ = keras.layers.Input(shape=X_train.shape[1:])
hidden1 = keras.layers.Dense(30, activation="relu")(input_)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.concatenate([input_, hidden2])
output = keras.layers.Dense(1)(concat)
model = keras.models.Model(inputs=[input_], outputs=[output])
```

```
model.summary()
```

Model: "model"

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, 8)]	0	
dense_5 (Dense)	(None, 30)	270	input_1 [0] [0]
dense_6 (Dense)	(None, 30)	930	dense_5 [0] [0]
concatenate (Concatenate)	(None, 38)	0	input_1 [0] [0] dense_6 [0] [0]
dense_7 (Dense)	(None, 1)	39	concatenate [0] [0]
Total params: 1,239			
Trainable params: 1,239			
Non-trainable params: 0			

```
model.compile(loss="mean_squared_error", optimizer=keras.optimizers.SGD(lr=1e-3))
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
mse_test = model.evaluate(X_test, y_test)
y_pred = model.predict(X_new)
```

```
Epoch 1/20
363/363 [=====] - 1s 3ms/step - loss: 1.9731 - val_loss: 3.3940
Epoch 2/20
363/363 [=====] - 1s 3ms/step - loss: 0.7638 - val_loss: 0.9360
Epoch 3/20
363/363 [=====] - 1s 3ms/step - loss: 0.6045 - val_loss: 0.5649
Epoch 4/20
363/363 [=====] - 1s 3ms/step - loss: 0.5862 - val_loss: 0.5712
Epoch 5/20
363/363 [=====] - 1s 3ms/step - loss: 0.5452 - val_loss: 0.5045
Epoch 6/20
363/363 [=====] - 1s 3ms/step - loss: 0.5243 - val_loss: 0.4831
Epoch 7/20
363/363 [=====] - 1s 3ms/step - loss: 0.5185 - val_loss: 0.4639
Epoch 8/20
363/363 [=====] - 1s 3ms/step - loss: 0.4947 - val_loss: 0.4638
Epoch 9/20
363/363 [=====] - 1s 3ms/step - loss: 0.4782 - val_loss: 0.4421
Epoch 10/20
363/363 [=====] - 1s 3ms/step - loss: 0.4708 - val_loss: 0.4313
```



Keras API

- Keras로 Sequence API 사용하여 Model 만들기
 - 함수형 API 기반 복잡한 Model 만들기

```
input_ = keras.layers.Input(shape=X_train.shape[1:])
hidden1 = keras.layers.Dense(30, activation="relu")(input_)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.concatenate([input_, hidden2])
output = keras.layers.Dense(1)(concat)
model = keras.models.Model(inputs=[input_], outputs=[output])
```

```
model.summary()
```

Model: "model"

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, 8)]	0	
dense_5 (Dense)	(None, 30)	270	input_1[0][0]
dense_6 (Dense)	(None, 30)	930	dense_5[0][0]
concatenate (Concatenate)	(None, 38)	0	input_1[0][0] dense_6[0][0]
dense_7 (Dense)	(None, 1)	39	concatenate[0][0]
Total params: 1,239			
Trainable params: 1,239			
Non-trainable params: 0			

```
model.compile(loss="mean_squared_error", optimizer=keras.optimizers.SGD(lr=1e-3))
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
mse_test = model.evaluate(X_test, y_test)
y_pred = model.predict(X_new)
```

```
Epoch 1/20
363/363 [=====] - 1s 3ms/step - loss: 1.9731 - val_loss: 3.3940
Epoch 2/20
363/363 [=====] - 1s 3ms/step - loss: 0.7638 - val_loss: 0.9360
Epoch 3/20
363/363 [=====] - 1s 3ms/step - loss: 0.6045 - val_loss: 0.5649
Epoch 4/20
363/363 [=====] - 1s 3ms/step - loss: 0.5862 - val_loss: 0.5712
Epoch 5/20
363/363 [=====] - 1s 3ms/step - loss: 0.5452 - val_loss: 0.5045
Epoch 6/20
363/363 [=====] - 1s 3ms/step - loss: 0.5243 - val_loss: 0.4831
Epoch 7/20
363/363 [=====] - 1s 3ms/step - loss: 0.5185 - val_loss: 0.4639
Epoch 8/20
363/363 [=====] - 1s 3ms/step - loss: 0.4947 - val_loss: 0.4638
Epoch 9/20
363/363 [=====] - 1s 3ms/step - loss: 0.4782 - val_loss: 0.4421
Epoch 10/20
363/363 [=====] - 1s 3ms/step - loss: 0.4708 - val_loss: 0.4313
```



Keras API

- Model Save & Load
 - Model case
 - Weight case

```
model.save("my_keras_model.h5")
```

```
model = keras.models.load_model("my_keras_model.h5")
```

```
model.predict(X_new)
```

```
WARNING:tensorflow:7 out of the last 8 calls to <function Model.make_predict_function.<locals>.predic
```

```
array([[0.5400236],  
       [1.6505969],  
       [3.0098238]], dtype=float32)
```

```
model.save_weights("my_keras_weights.ckpt")
```

```
model.load_weights("my_keras_weights.ckpt")
```

```
<tensorflow.python.training.tracking.util.CheckpointLoadStatus at 0x7fe464c02d30>
```





Keras API

- Callback
 - 훈련이 장시간 지속되는 경우, 훈련 도중 일정 간격으로 체크포인트를 저장해야할 때 활용
 - Fit() 메서드의 callbacks 매개변수를 사용하여 케라스가 훈련의 시작이나 끝에 호출할 객체 리스트를 지정할 수 있음
 - ModelCheckpoint는 훈련하는 동안 일정한 간격으로 모델의 체크포인트를 저장함
 - Early stopping 설정 가능
 - 기본적으로 매 epoch의 끝에서 호출 됨

```
: model.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=1e-3))
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5", save_best_only=True)
history = model.fit(X_train, y_train, epochs=10,
                    validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb])
model = keras.models.load_model("my_keras_model.h5") # 최상의 모델로 불러옴
mse_test = model.evaluate(X_test, y_test)
```

```
: model.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=1e-3))
early_stopping_cb = keras.callbacks.EarlyStopping(patience=10,
                                                  restore_best_weights=True)
history = model.fit(X_train, y_train, epochs=100,
                    validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb, early_stopping_cb])
mse_test = model.evaluate(X_test, y_test)
```



Keras API

- Tensorboard : Interactive 시각화 도구
 - 훈련하는 동안의 학습곡선 생성
 - 여러 실행 간 학습 곡선을 비교
 - 계산 그래프 시각화와 훈련 통계 분석 수행
 - 모델이 생성한 이미지를 확인
 - 3D에 투영된 복잡한 다차원 데이터 시각화
 - 자동 클러스터링 등
- 활용 시 알아 둘 내용
 - 각각의 이진 데이터 레코드 : Summary
 - 텐서보드 서버는 로그 디렉토리를 모니터링 및 변경사항을 읽어서 그래프를 업데이트함
 - 텐서보드 서버가 루트 로그 디렉토리를 가리키고, 프로그램은 실행될 때 마다, 다른 서브 디렉토리에 이벤트를 기록함





Keras API

- Tensorboard 사용법

```
: root_logdir = os.path.join(os.getcwd(), "my_logs")
```

```
: def get_run_logdir():  
:     import time  
:     run_id = time.strftime("run_%Y_%m_%d-%H_%M_%S")  
:     return os.path.join(root_logdir, run_id)  
  
run_logdir = get_run_logdir()  
run_logdir
```

```
: './my_logs/run_2021_02_18-03_57_36'
```

```
: keras.backend.clear_session()  
np.random.seed(42)  
tf.random.set_seed(42)
```

```
: model = keras.models.Sequential([  
:     keras.layers.Dense(30, activation="relu", input_shape=[8]),  
:     keras.layers.Dense(30, activation="relu"),  
:     keras.layers.Dense(1)  
: ])  
model.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=1e-3))
```

```
: tensorboard_cb = keras.callbacks.TensorBoard(run_logdir)  
history = model.fit(X_train, y_train, epochs=30,  
:                 validation_data=(X_valid, y_valid),  
:                 callbacks=[checkpoint_cb, tensorboard_cb])
```





Keras API

- Tensorboard 사용법

```
: root_logdir = os.path.join(os.getcwd(), "my_logs")
```

```
: def get_run_logdir():  
    import time  
    run_id = time.strftime("run_%Y_%m_%d-%H_%M_%S")  
    return os.path.join(root_logdir, run_id)  
  
run_logdir = get_run_logdir()  
run_logdir
```

```
: './my_logs/run_2021_02_18-03_57_36'
```

```
: keras.backend.clear_session()  
np.random.seed(42)  
tf.random.set_seed(42)
```

```
: model = keras.models.Sequential([  
    keras.layers.Dense(30, activation="relu", input_shape=[8]),  
    keras.layers.Dense(30, activation="relu"),  
    keras.layers.Dense(1)  
])  
model.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=1e-3))
```

```
: tensorboard_cb = keras.callbacks.TensorBoard(run_logdir)  
history = model.fit(X_train, y_train, epochs=30,  
                    validation_data=(X_valid, y_valid),  
                    callbacks=[checkpoint_cb, tensorboard_cb])
```

텐서보드 서버를 실행하는 한 가지 방법은 터미널에서 직접 실행하는 것입니다. 터미널을 열고 텐서보드가 설치된 가상 환경을 활성화합니다. 그다음 노트북 디렉토리로 이동하여 다음 명령을 입력하세요:

```
$ tensorboard --logdir=./my_logs --port=6006
```

그다음 웹 브라우저를 열고 localhost:6006에 접속하면 텐서보드를 사용할 수 있습니다. 사용이 끝나면 터미널에서 Ctrl-C를 눌러 텐서보드 서버를 종료하세요.

또는 다음처럼 텐서보드의 주피터 확장을 사용할 수 있습니다(이 명령은 텐서보드가 로컬 컴퓨터에 설치되어 있어야 합니다):

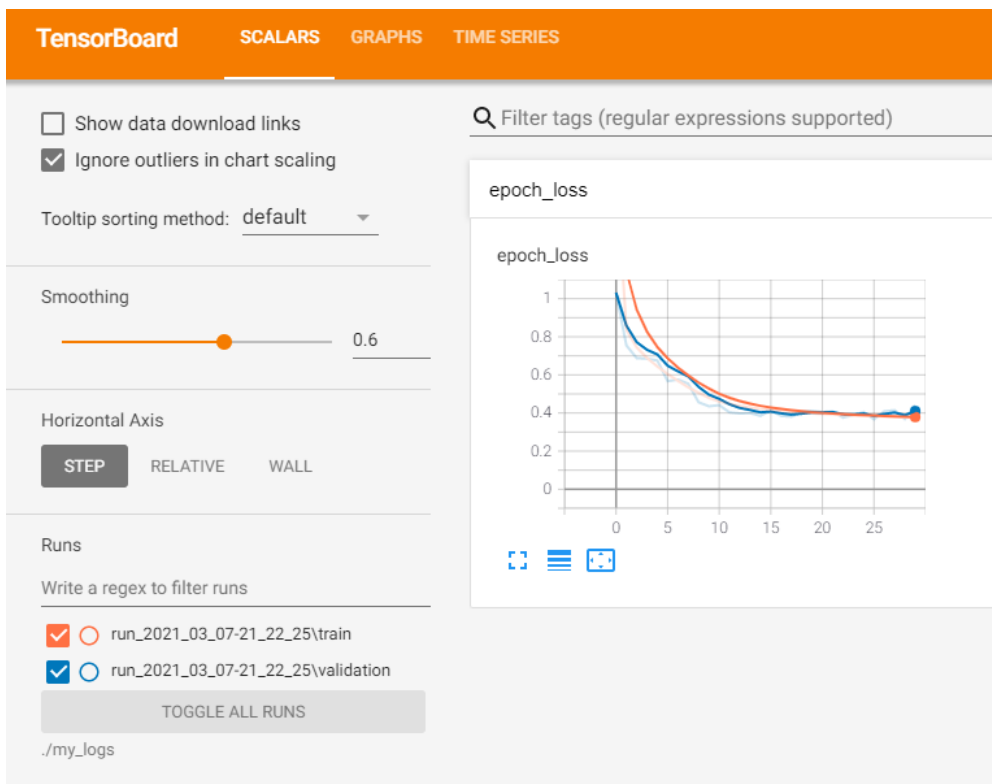
```
%load_ext tensorboard  
%tensorboard --logdir=./my_logs --port=6006
```





Keras API

- Tensorboard 화면 예시





Keras API

- 신경망 Hyperparameter Tuning
 - 신경망의 단점
 - 유연성, 조정할 하이퍼파라미터가 많기 때문
 - 최적 Hyperparameter Tuning 방법
 - 많은 Hyperparameter 조합을 시도하여서 검증세트에서 가장 좋은 점수를 내는 경우를 선택
 - GridSearchCV | RandomizedSearchCV 기반 Tuning 진행
 - 케라스 model을 scikit-learn estimator로 인식되도록 변경해야함.

```
def build_model(n_hidden=1, n_neurons=30, learning_rate=3e-3, input_shape=[8]):  
    model = keras.models.Sequential()  
    model.add(keras.layers.InputLayer(input_shape=input_shape))  
    for layer in range(n_hidden):  
        model.add(keras.layers.Dense(n_neurons, activation="relu"))  
    model.add(keras.layers.Dense(1))  
    optimizer = keras.optimizers.SGD(lr=learning_rate)  
    model.compile(loss="mse", optimizer=optimizer)  
    return model
```

```
keras_reg = keras.wrappers.scikit_learn.KerasRegressor(build_model)
```

```
keras_reg.fit(X_train, y_train, epochs=100,  
             validation_data=(X_valid, y_valid),  
             callbacks=[keras.callbacks.EarlyStopping(patience=10)])
```



Keras API

- 신경망 Hyperparameter Tuning
 - 신경망의 단점
 - 유연성, 조정할 하이퍼파라미터가 많기 때문
 - 최적 Hyperparameter Tuning 방법
 - 많은 Hyperparameter 조합을 시도하여서 검증세트에서 가장 좋은 점수를 내는 경우를 선택
 - GridSearchCV | RandomizedSearchCV 기반 Tuning 진행
 - 케라스 model을 scikit-learn estimator로 인식되도록 변경해야함.

```
def build_model(n_hidden=1, n_neurons=30, learning_rate=3e-3, input_shape=[8]):  
    model = keras.models.Sequential()  
    model.add(keras.layers.InputLayer(input_shape=input_shape))  
    for layer in range(n_hidden):  
        model.add(keras.layers.Dense(n_neurons, activation="relu"))  
    model.add(keras.layers.Dense(1))  
    optimizer = keras.optimizers.SGD(lr=learning_rate)  
    model.compile(loss="mse", optimizer=optimizer)  
    return model
```

```
keras_reg = keras.wrappers.scikit_learn.KerasRegressor(build_model)
```

```
keras_reg.fit(X_train, y_train, epochs=100,  
             validation_data=(X_valid, y_valid),  
             callbacks=[keras.callbacks.EarlyStopping(patience=10)])
```

```
from scipy.stats import reciprocal  
from sklearn.model_selection import RandomizedSearchCV
```

```
param_distributions = {  
    "n_hidden": [0, 1, 2, 3],  
    "n_neurons": np.arange(1, 100).tolist(),  
    "learning_rate": reciprocal(3e-4, 3e-2).rvs(1000).tolist(),  
}
```

```
rnd_search_cv = RandomizedSearchCV(keras_reg, param_distributions, n_iter=10, cv=3, verbose=2)  
rnd_search_cv.fit(X_train, y_train, epochs=100,  
                 validation_data=(X_valid, y_valid),  
                 callbacks=[keras.callbacks.EarlyStopping(patience=10)])
```



Keras API

- 신경망 Hyperparameter Tuning 최종 결과 확인

```
rnd_search_cv.best_params_
```

```
{'n_neurons': 74, 'n_hidden': 3, 'learning_rate': 0.005803602934201024}
```

```
rnd_search_cv.best_score_
```

```
-0.3242675264676412
```

```
rnd_search_cv.best_estimator_
```

```
<tensorflow.python.keras.wrappers.scikit_learn.KerasRegressor at 0x7fe3c2171790>
```

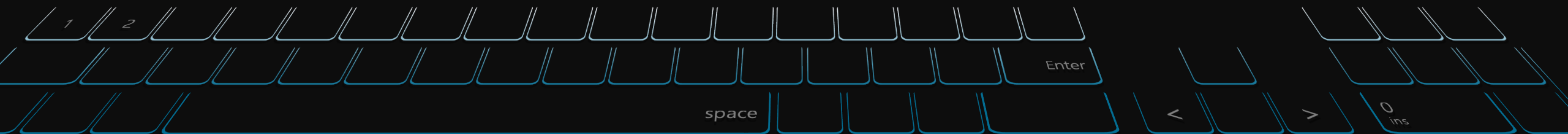
```
rnd_search_cv.score(X_test, y_test)
```

```
162/162 [=====] - 0s 2ms/step - loss: 0.3104
```

```
-0.31039538979530334
```

```
model = rnd_search_cv.best_estimator_.model  
model
```

```
<tensorflow.python.keras.engine.sequential.Sequential at 0x7fe3c2171af0>
```



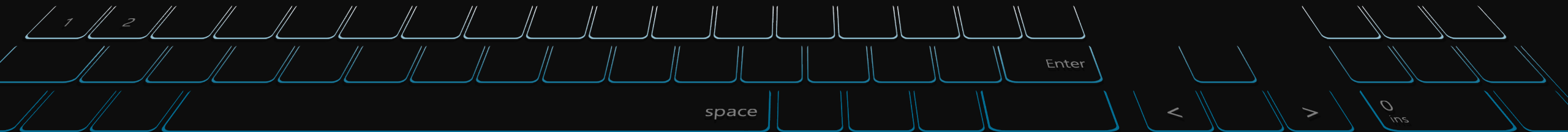


Keras API

- Hyperparameter Tuning Tool 공유
 - [\[Link\] Hyperparameter optimization tool](#)

Q&A and Break Time

질의응답 및 휴식 시간 (10분)



DL 알고리즘 응용

1) Gradient 소실 & 폭주, 고속 Optimizer, 규제 (Regularizer)

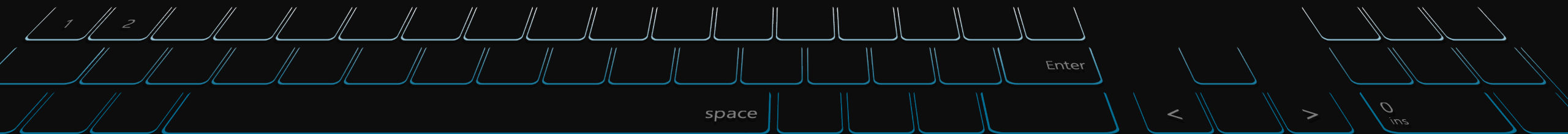




DL 알고리즘 응용 Contents

Deep Learning Algorithm Advanced

1. Gradient 소실 & 폭주
2. 고속 Optimizer
3. 규제 (Regularizer)

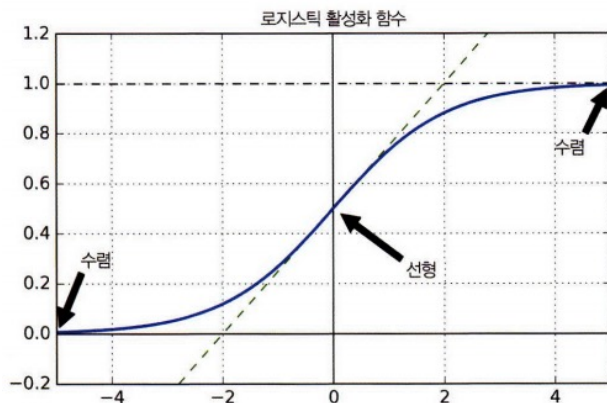




그래디언트 소실과 폭주

Vanishing & Exploding Gradient

- 그래디언트 소실(Vanishing Gradient): 알고리즘이 하위층으로 진행될수록 그래디언트가 점점 작아지는 경우
- 그래디언트 폭주(Exploding Gradient): 그래디언트가 점점 커져서 여러 층이 비정상적으로 큰 가중치를 갱신되어 알고리즘이 발산하는 경우
- 로지스틱 활성화 함수의 경우 입력이 커지면 0이나 1로 수렴해서 기울기가 0에 가까워짐
- 역전파가 될 때 사실상 신경망으로 전파할 그래디언트가 거의 없고 조금 있는 그래디언트는 최상위층에서부터 역전파가 진행되면서 점차 약해져서 실제로 아래쪽 층에는 아무것도 도달하지 않는다.





그래디언트 소실과 폭주

Vanishing & Exploding Gradient

글로럿과 He 초기화

- 글로럿과 벤지오가 불안정한 그래디언트 문제를 완화하는 방법 제안
- 각 층의 출력에 대한 분산이 입력에 대한 분산과 같아야 한다고 주장
- 역방향에서 층을 통과하기 전과 후의 그래디언트 분산이 동일
- $fan_{avg} = (fan_{in} + fan_{out})$ 층의 입력 개수는 팬인 출력 개수는 팬아웃

글로럿 초기화

평균이 0이고 분산이 $\sigma^2 = \frac{1}{fan_{avg}}$ 인 정규 분포

또는 $r = \sqrt{\frac{3}{fan_{avg}}}$ 일 때 -r과 +r사이의 균등 분포

르쿤 초기화

평균이 0이고 분산이 $\sigma^2 = \frac{1}{fan_{in}}$ 인 정규 분포

또는 $r = \sqrt{\frac{3}{fan_{in}}}$ 일 때 -r과 +r사이의 균등 분포

- 글로럿 초기화를 하면 훈련 속도가 증가

초기화 전략	활성화 함수	σ^2 (정규분포)
글로럿	활성화 함수 없음, tanh, logistic, softmax	$1/fan_{avg}$
He	ReLU 함수와 그 변종들	$2/fan_{in}$
르쿤	SELU	$1/fan_{in}$

- 케라스는 기본적으로 균등 분포의 글로럿 초기화를 사용





그래디언트 소실과 폭주

Vanishing & Exploding Gradient

수렴하지 않는 활성화 함수

- ReLU, ELU, SELU 등

[\[블로그\] 활성화 함수 10개 비교 자료](#)



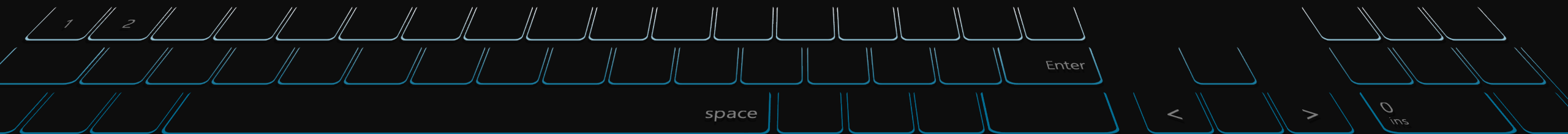


그래디언트 소실과 폭주

Vanishing & Exploding Gradient

배치 정규화

- 훈련하는 동안의 그래디언트 소실이나 폭주문제를 해결하기 위한 방법
- 세르게이 이오페, 치리슈티안 세게지가 제안
- 각 층에서 활성화 함수를 통과하기 전이나 후에 모델에 연산을 하나 추가
- 입력을 원점에 맞추고 정규화한 다음, 각 층에서 두 개의 새로운 파라미터로 결과값의 스케일을 조정하고 이동
- 신경망의 첫 번째 층으로 배치 정규화를 추가하면 훈련세트를 표준화할 필요가 없다.





그래디언트 소실과 폭주

Vanishing & Exploding Gradient

배치 정규화 알고리즘

배치 정규화

1.

$$\mu_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$$

2.

$$\sigma_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} (\mathbf{x}^{(i)} - \mu_B)^2$$

3.

$$\hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

4.

$$\mathbf{z}^{(i)} = \gamma \otimes \hat{\mathbf{x}}^{(i)} + \beta$$

μ_B 는 미니배치 B에 대해 평가한 입력의 평균 벡터(입력마다 하나의 평균)

σ_B 도 미니배치에 대해 평가한 입력의 표준편차 벡터

m_B 는 미니배치에 있는 샘플 수

$\hat{\mathbf{x}}^{(i)}$ 는 평균이 0이고 정규화된 샘플 i의 입력

γ 는 층의 출력 스케일 파라미터 벡터(입력마다 하나의 스케일 파라미터가 존재)

\otimes 는 원소별 곱셈(각 입력은 해당되는 출력 스케일 파라미터와 곱해진다.)

β 는 층의 출력 이동(오프셋) 파라미터 벡터(입력마다 하나의 스케일 파라미터 존재). 각 입력은 해당 파라미터만큼 이동

ε 은 분모가 0이 되는 것을 막기 위한 작은 숫자(전형적으로 10^{-5}). 안전을 위한 항(smoothing term) 이라고 함.

$\mathbf{z}^{(i)}$ 는 배치 정규화 연산의 출력. 즉 입력의 스케일을 조정하고 이동시킨 것



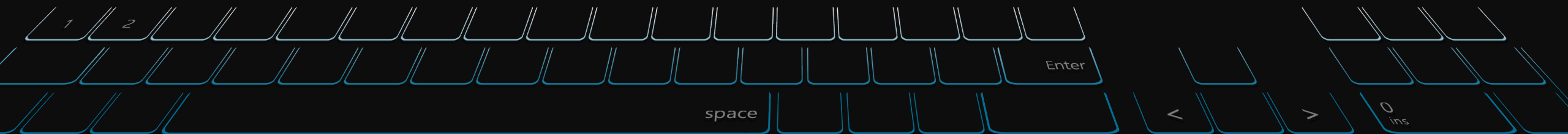


그래디언트 소실과 폭주

Vanishing & Exploding Gradient

배치 정규화

- 테스트 시,
 - 훈련이 끝난 후 전체 훈련 세트를 신경망에 통과시켜,
 - 배치 정규화 층의 각 입력에 대한 평균과 표준편차를 계산하여 예측할 때,
 - 배치 입력 평균과 표준 편차로 이 '최종' 입력 평균과 표준편차를 대신 사용하는 방법이 있지만,
 - 대부분 층의 입력 평균과 표준편차의 이동 평균을 사용해 훈련하는 동안 최종 통계를 추정
- 케라스의 BatchNormalization 층은 이를 자동으로 수행





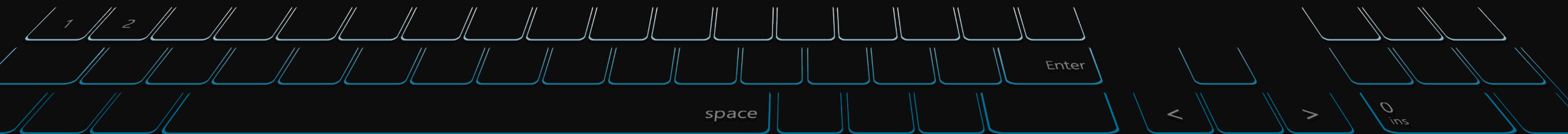
그래디언트 소실과 폭주

Vanishing & Exploding Gradient

배치 정규화

1. γ (출력 스케일 벡터)와 β (출력 이동 벡터)는 일반적인 역전파를 통해 학습
2. μ (최종 입력 평균 벡터)와 σ (최종 입력 표준편차 벡터)는 지수 이동 평균을 사용하여 추정
3. μ 와 σ 는 훈련하는 동안 추정되지만 훈련이 끝난 후에 사용

- 이미지넷 분류 작업에서 큰 성과
- 그래디언트 소실 문제가 크게 감소하여 tanh나 로지스틱 함수와 같은 수렴성을 가진 활성화 함수와도 사용 가능
- 가중치 초기화에 네트워크가 훨씬 덜 민감
- 규제와도 같은 역할을 하여 다른 규제 기법의 필요성 감소
- 단점
 - 모델의 복잡도가 커짐
 - 실행 시간 증가(대신 이전 층의 가중치를 바꾸어 이전 층과 배치 정규화 층이 합쳐진 결과를 내어 실행 시간을 단축시킬 수 있음)





그래디언트 소실과 폭주

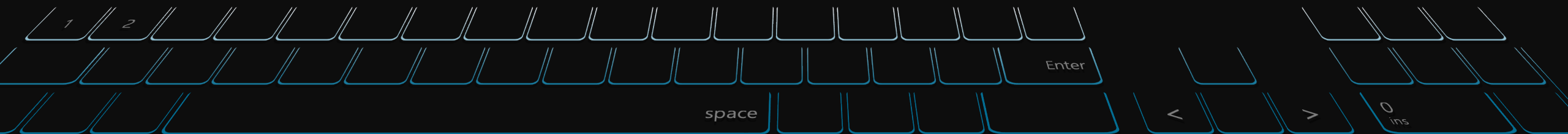
Vanishing & Exploding Gradient

그래디언트 클리핑

- 역전파될 때 일정 임계값을 넘어서지 못하게 그래디언트를 자르는 방법
- 케라스에서 구현하려면 옵티마이저를 만들 때 clipvalue와 clipnorm 매개변수를 지정

```
optimizer = keras.optimizer.SGD(clipvalue=1.0)  
model.compile(loss='mse', optimizer=optimizer)
```

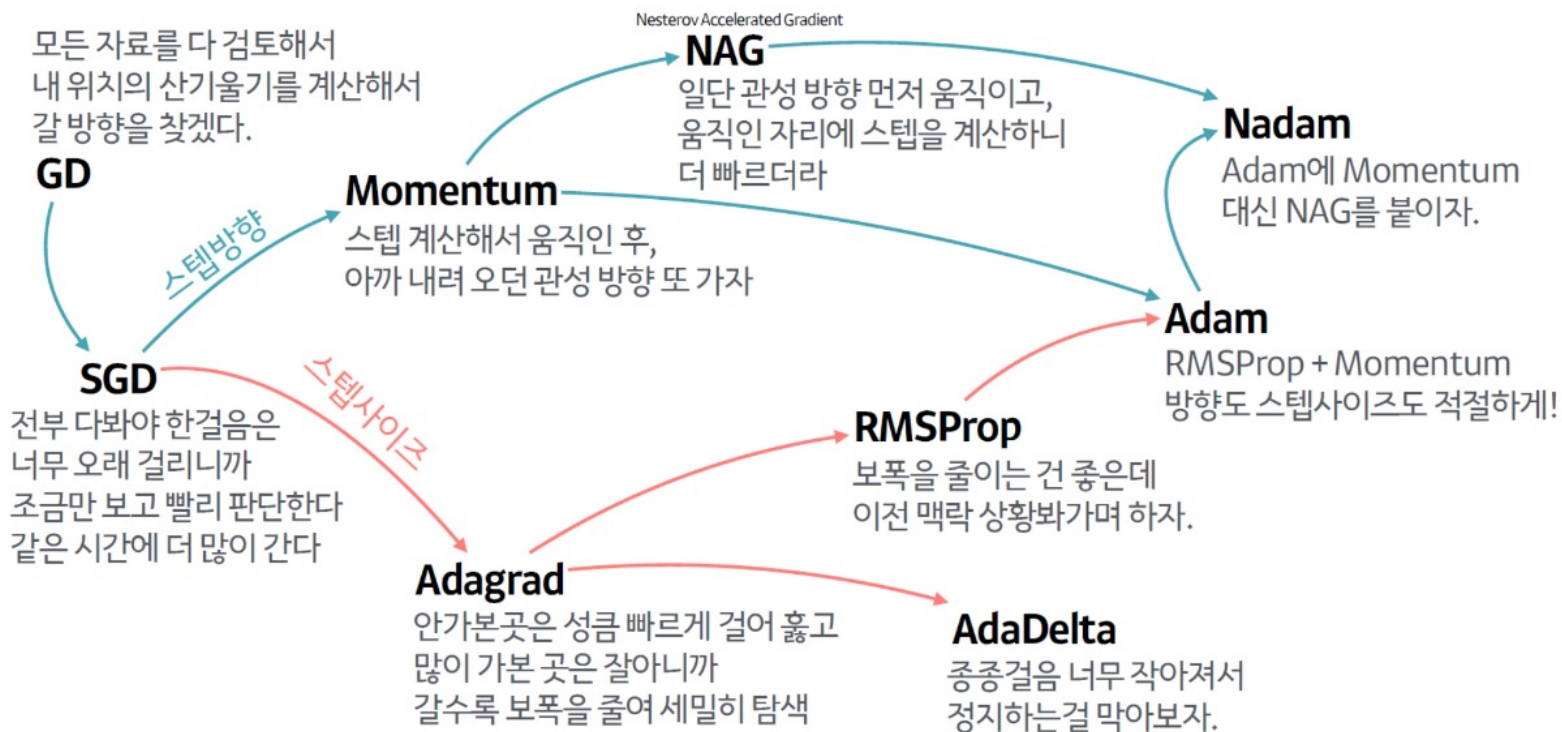
- 위의 경우 그래디언트 벡터의 모든 원소를 -1.0과 1.0 사이로 클리핑
 - 그래디언트 벡터의 방향을 바꿀 수 있다.
 - 그래디언트 클리핑이 그래디언트 벡터의 방향을 바꾸지 못하게 하려면 clipnorm을 지정하여 노름으로 클리핑





고속 옵티마이저

Blueprint





고속 옵티마이저

모멘텀 최적화 (Momentum Optimizer)

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

- 표준적인 경사 하강법은 경사면을 따라 일정한 크기의 스텝으로 조금씩 내려가는 반면 모멘텀 최적화는 처음에는 느리게 출발하지만 종단속도에 도달할 때까지는 빠르게 가속

- 경사 하강법

$$\theta \leftarrow -\eta \nabla_{\theta} J(\theta)$$

- 모멘텀 알고리즘

$$\mathbf{m} \leftarrow \beta \mathbf{m} - \eta \nabla_{\theta} J(\theta)$$

$$\theta \leftarrow \theta + \mathbf{m}$$

- 모멘텀 최적화는 이전 그래디언트가 얼마였는지가 상당히 중요
- 현재 그래디언트를 (학습률 η 를 곱한 후) 모멘텀 벡터 \mathbf{m} 에 더하고 이 값을 빼는 방식으로 가중치를 갱신
- 일종의 마찰저항을 표현하고 모멘텀이 너무 커지는 것을 막기 위해 모멘텀이라는 새로운 하이퍼파라미터 β 사용(일반적인 모멘텀 값은 0.9)
- 더 바르게 평편한 지역을 탈출하게 도움
- 경사 하강법이 **가파른 경사를 꽤 빠르게 내려가지만 좁고 긴 골짜기에서는 오랜 시간이 걸린다.**
모멘텀 최적화는 골짜기를 따라 **바닥(최적점)에 도달할 때까지 점점 더 빠르게** 내려간다.
- 배치 정규화를 사용하지 않는 심층 신경망에서 상위층은 종종 스케일이 매우 다른 입력을 받게 되는데 이런 경우 사용하면 도움이 된다.
- 또한 지역 최적점을 건너뛰도록 하는 데도 도움





고속 옵티마이저

네스테로프 가속 경사 (Nesterov Accelerated Gradient, NAG)

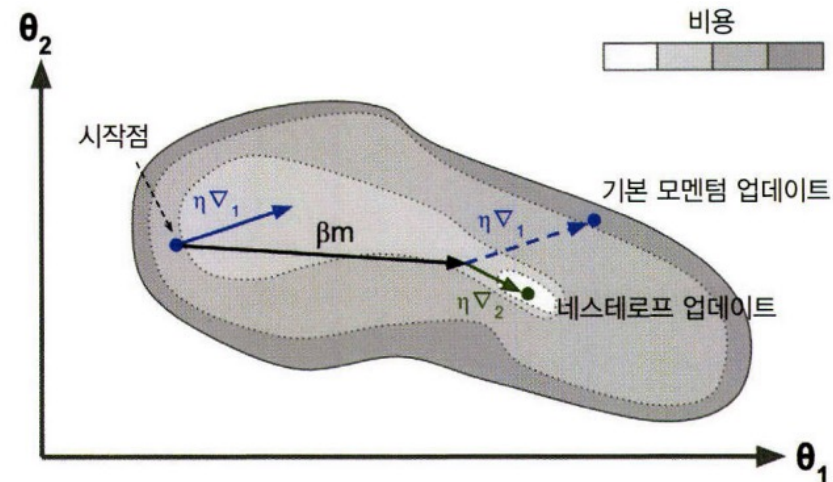
```
optimizer = keras.optimizer.SGD(lr=0.001, momentum=0.9, nesterov=True)
```

- 현재 위치가 θ 가 아니라 모멘텀 방향으로 조금 앞선 $\theta + \beta \mathbf{m}$ 에서 비용 함수의 그래디언트를 계산하는 것
- 네스테로프 가속 경사 알고리즘

$$\mathbf{m} \leftarrow \beta \mathbf{m} - \eta \nabla_{\theta} J(\theta + \beta \mathbf{m})$$

$$\theta \leftarrow \theta + \mathbf{m}$$

- 원래 위치에서의 그래디언트를 사용하는 것보다 그 방향으로 조금 더 나아가서 측정한 그래디언트를 사용하는 것이 더 정확할 것
- 진동을 감소시키고 수렴을 빠르게 만들어 준다.
- 일반적으로 기본 모멘텀 최적화보다 훈련 속도가 빠르다.





고속 옵티마이저

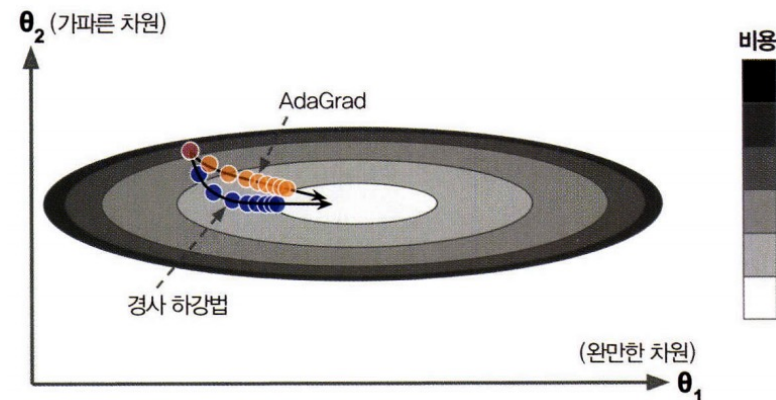
AdaGrad

- 가장 가파른 차원을 따라 그래디언트 벡터의 스케일 θ_{β} 감소
- AdaGrad 알고리즘

$$\mathbf{s} \leftarrow \mathbf{s} + \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$$

$$\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{\mathbf{s} + \epsilon}$$

- 첫 번째 단계로 그래디언트 제곱을 벡터 \mathbf{s} 에 누적(각 원소 \mathbf{s}_i 마다 파라미터 θ_i 에 대한 비용 함수의 편미분을 제곱하여 누적)
- 두 번째 단계로 그래디언트 벡터를 $\sqrt{\mathbf{s} + \epsilon}$ 으로 나누어 스케일을 조정 (\oslash 기호는 원소별 나눗셈을 나타내고 ϵ 은 0으로 나누는 것을 막기 위한 값으로, 일반적으로 10^{-10})
- 이 알고리즘은 학습률을 감소시키지만 경사가 완만한 차원보다 가파른 차원에 대해 더 빠르게 감소하며 이를 적응적 학습률(adaptive learning)이라고 부르며, 전역 최적점 방향으로 더 곧장 가도록 갱신되는 데 도움
- 학습률 하이퍼파라미터 η 를 덜 튜닝해도 되는 점이 또 하나의 장점
- 간단한 2차 방정식 문제에 대해서는 잘 작동하지만 신경망을 훈련할 때 너무 일찍 멈추는 경우가 있음.
학습률이 너무 감소되어 전역 최적점에 도착하기전에 알고리즘이 멈춤.
- 케라스에 Adagrad 옵티마이저가 있지만 **심층 신경망에 사용하지 말 것**





고속 옵티마이저

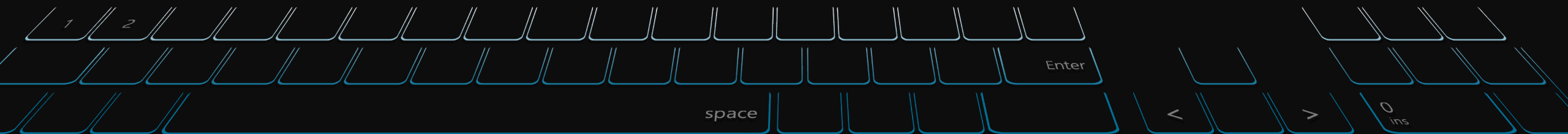
RMSProp `optimizer = keras.optimizer.RMSprop(lr=0.001, rho=0.9)`

- 가장 최근 반복에서 비롯된 그래디언트만 누적함으로써 AdaGrad가 전역 최적점에 도착하기 전에 알고리즘이 멈추는 문제를 해결
- 지수 감소를 사용
- RMSProp 알고리즘

$$\mathbf{s} \leftarrow \beta \mathbf{s} + (1 - \beta) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$$

$$\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{\mathbf{s} + \varepsilon}$$

- 보통 감쇠율 β 는 0.9로 설정
- 기본값이 잘 작동하는 경우가 많으므로 튜닝할 필요는 전혀 없음
- rho 매개변수는 β
- AdaGrad보다 훨씬 더 성능이 좋다.





고속 옵티마이저

Adam `optimizer = keras.optimizer.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)`

- 적응적 모멘트 추정(adaptive moment estimation)을 의미
- 모멘텀 최적화와 RMSProp의 아이디어를 합친 것
- 모멘텀 최적화처럼 지난 그래디언트의 지수 감소 평균을 따르고 RMSProp처럼 지난 그래디언트 제곱의 지수 감소된 평균을 따른다.
- Adam 알고리즘

$$\mathbf{m} \leftarrow \beta_1 \mathbf{m} + (1 - \beta_1) \nabla_{\theta} J(\theta)$$

$$\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$$

$$\hat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1 - \beta_1^t}$$

$$\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \beta_2^t}$$

$$\theta \leftarrow \theta + \eta \hat{\mathbf{m}} \oslash \sqrt{\hat{\mathbf{s}} + \epsilon}$$

- t는 (1부터 시작하는) 반복 횟수
- 단계 1,2,5는 모멘텀 최적화, RMSProp과 비슷
- 단계 3,4에서 m, s는 0으로 초기화되기 때문에 훈련 초기에 0쪽으로 치우치게 된다. 그래서 이 두 단계가 훈련 초기에 m, s의 값을 증폭시키는 데 도움을 준다.
- 반복 초기에는 m, s를 증폭시켜주지만 반복이 많이 진행되면 단계 3, 4의 분모는 1에 가까워져 거의 증폭되지 않는다.
- 모멘텀 감쇠 하이퍼파라미터 β_1 은 보통 0.9로, 스케일 감쇠 하이퍼파라미터 β_2 는 0.999로 초기화
- Adam이 적응적 학습률 알고리즘이기 때문에 학습률 하이퍼파라미터 η 를 튜닝할 필요가 적다.(기본값 0.001을 일반적으로 사용)



고속 옵티마이저

Adamax

- Adam은 시간에 따라 감소된 그래디언트의 l2 노름으로 파라미터 업데이트의 스케일을 낮춘다.(l2 노름은 제곱 합의 제곱근)
- Adamax는 l2 노름을 l ∞ 노름으로 바꾼다
- Adamax 알고리즘

$$\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1 - \beta_1) \nabla_{\theta} J(\theta)$$

$$\mathbf{s} \leftarrow \max(\beta_2 \mathbf{s}, \nabla_{\theta} J(\theta))$$

$$\hat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1 - \beta_1^t}$$

$$\theta \leftarrow \theta + \eta \hat{\mathbf{m}} \oslash \sqrt{\mathbf{s} + \epsilon}$$

- Adamax가 Adam보다 더 안정적
- 하지만 실제로 데이터셋에 따라 다르고 일반적으로 Adam의 성능이 더 낫다.





고속 옵티마이저

Nadam

- Adam 옵티마이저에 네스테로프 기법을 더한 것
- Adam보다 조금 더 빠르게 수렴
- 일반적으로 Nadam이 Adam보다 성능이 좋았지만 이따금 RMSProp이 나을 때도 있다.

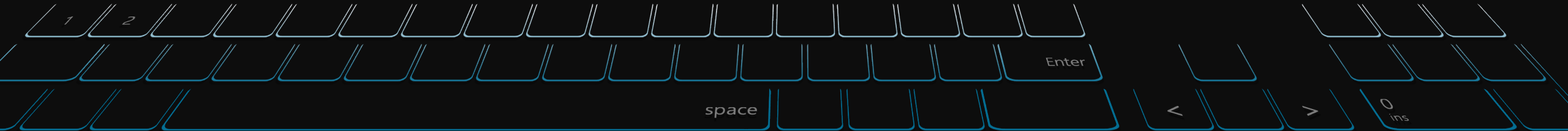


고속 옵티마이저

최종 옵티마이저 비교

옵티마이저 비교(*=나쁨, *=보통, **=좋음)

클래스	수렴 속도	수렴 품질
SGD	*	***
Momentum	**	***
Nesterov	**	***
AdaGrad	***	* (너무 일찍 멈춤)
RMSProp	***	** 또는 ***
Adam	***	** 또는 ***
Nadam	***	** 또는 ***
AdaMax	***	** 또는 ***

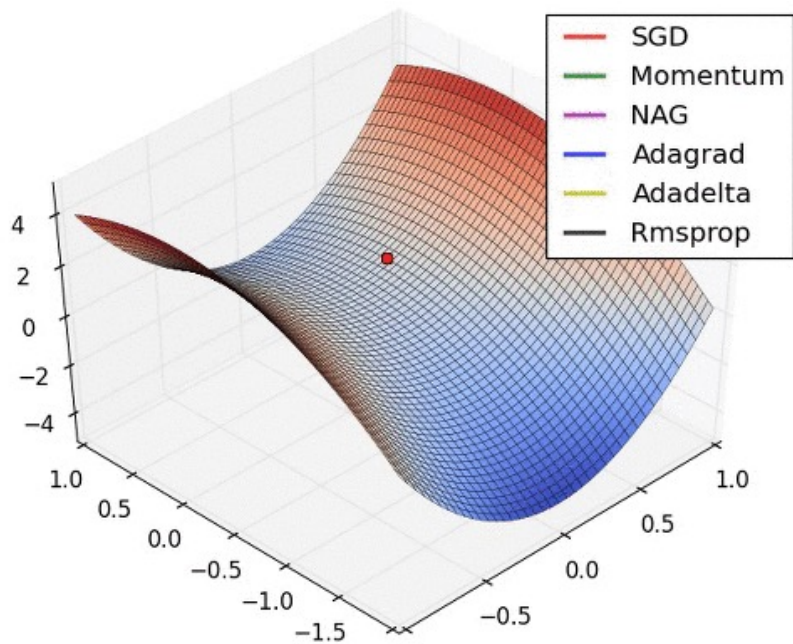




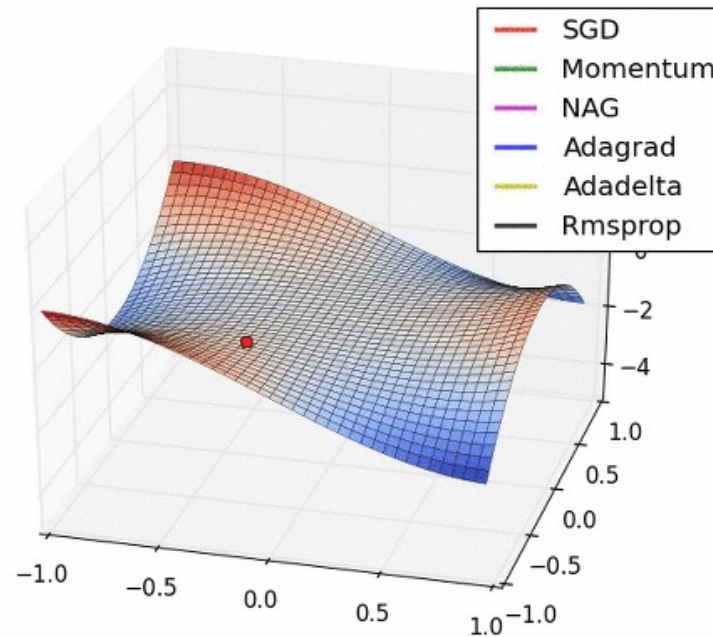
고속 옵티마이저

최종 옵티마이저 비교

협곡에서 옵티마이저별 경로



말안장점에서 옵티마이저별 경로

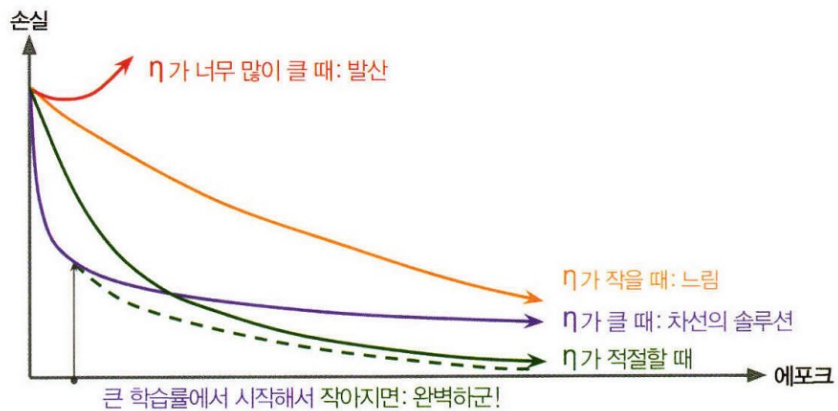




고속 옵티마이저

학습률 스케줄링

- 학습률을 너무 크게 잡으면 발산할 수 있으며, 너무 작게 잡으면 최적점에 수렴하는데 오랜 시간이 걸린다.



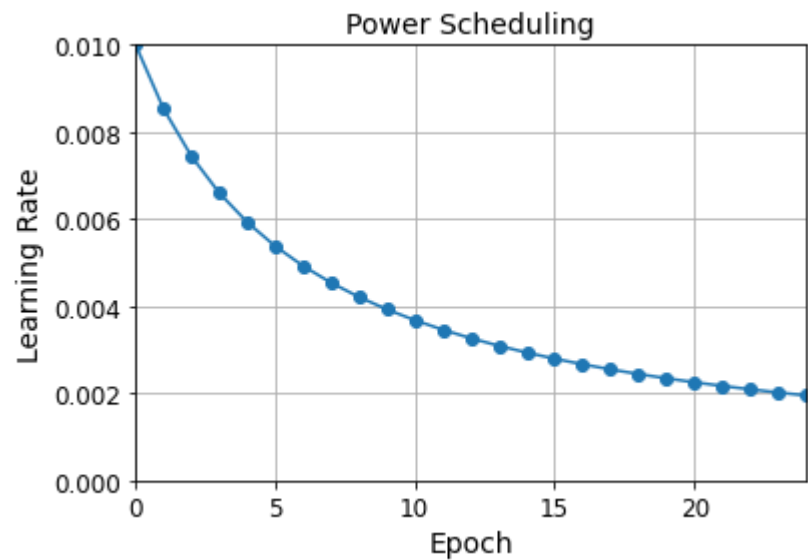
- 큰 학습률로 시작하고 학습 속도가 느려질 때 학습률을 낮추면 최적의 고정 학습률 보다 좋은 솔루션을 더 빨리 발견할 수 있다.
- 이를 학습 스케줄이라고 한다



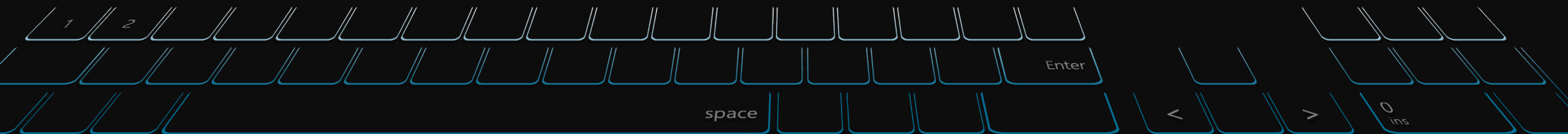
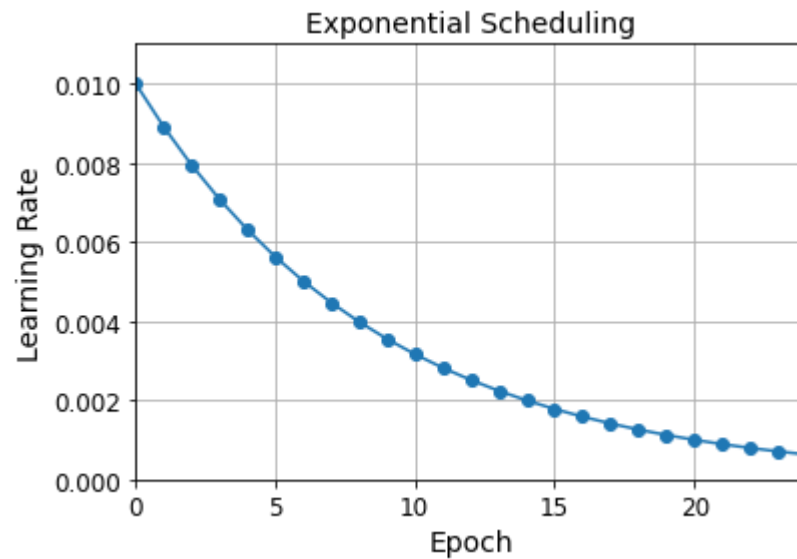
고속 옵티마이저

학습률 스케줄링 기법

1. 거듭제곱 기반 스케줄링(Power Scheduling)



2. 지수 기반 스케줄링 (Exponential Scheduling)

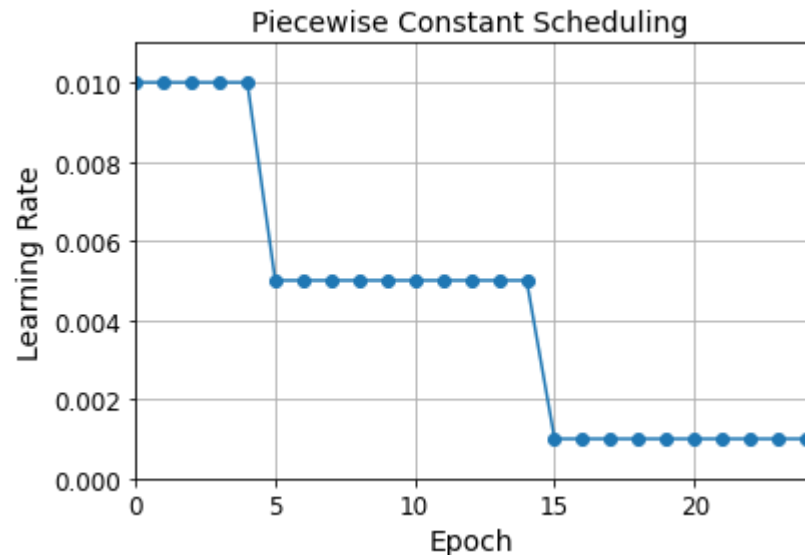




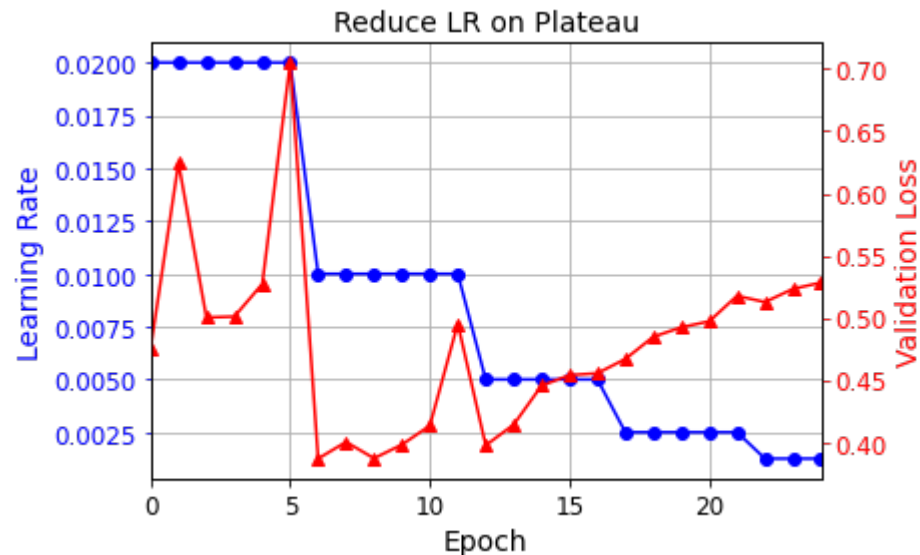
고속 옵티마이저

학습률 스케줄링 기법

3. 구간별 고정 스케줄링 (Piecewise Constant Scheduling)



4. 성능 기반 스케줄링 (Performance Scheduling)



5. 1 사이클 스케줄링 (1cycle Scheduling)

- 1 사이클은 훈련 절반 동안 초기에 지정한 학습률 n_0 을 선형적으로 n_1 까지 증가시킵니다. 그 다음 나머지 절반 동안은 증가되었던 n_1 을 n_0 으로 선형적으로 되돌립니다. 그리고 마지막 몇 번의 에포크 동안의 학습률은 소수점 몇 째 자리까지 선형적으로 줄입니다.



L1, L2 규제

- 다음은 규제 강도를 0.01을 사용하여 L2 규제를 적용하는 방법 예시

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

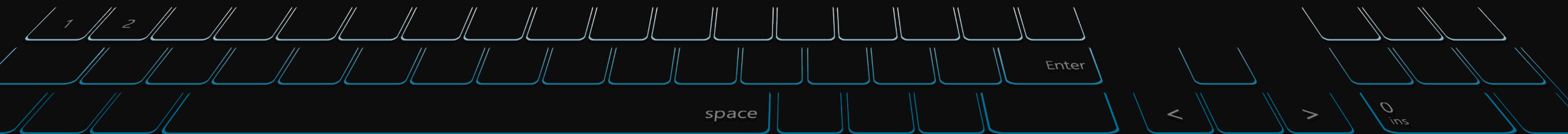
- L1 규제를 사용하고 싶다면 `keras.regularizers.l1()`을 사용하면 되고,
L1, L2 두가지가 모두 필요하면 `keras.regularizers.l1_l2()`를 사용하면 된다.



Dropout Rate

- Neural Network에서 가장 인기 있는 규제 기법 중 하나
- 매 Training 스텝에서 네트워크를 구성하는 각 뉴런은 임시적으로 드롭아웃될 확률 p 를 가진다.
(즉, 입력 뉴런은 포함되고, 출력 뉴런은 제외된다.)
다시 말해서 드롭아웃된 뉴런은 완전히 없는 셈치고 학습을 진행
- Test set에서는 드롭아웃이 적용되지 않다.
- 매 훈련 스텝마다 네트워크의 모양은 다르다.
즉, 매 훈련 스텝마다 서로 다른 Neural Network를 사용한 셈이 되는 것이다.
- 앙상블해서 훈련시킨 효과를 가지게 된다.

```
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")
])
```





규제

Regularizer

Max Norm 규제

```
layer = keras.layers.Dense(100, activation="selu", kernel_initializer="lecun_normal",  
                             kernel_constraint=keras.constraints.max_norm(1.))
```

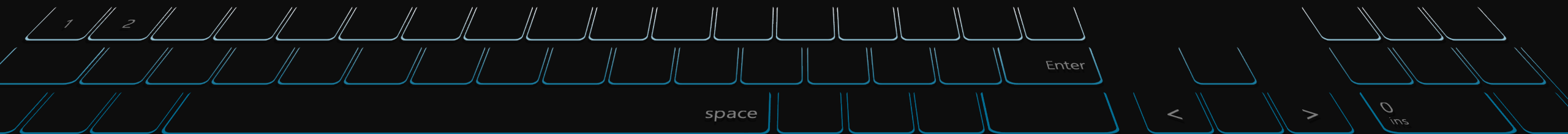
- 각 뉴런에 대해 입력의 연결 가중치 w 가 다음과 같도록 제한

$$\| \mathbf{w} \|_2 \leq r$$

- 전체 Loss 함수에 규제 항을 추가하지 않고, 매 훈련 스텝이 끝날때마다 w 의 norm을 계산하여 다음과 같이 스케일을 조정한다.

$$\mathbf{w} \leftarrow \mathbf{w} \frac{r}{\| \mathbf{w} \|_2}$$

- r 은 하이퍼 파라미터이며, r 을 줄이면 줄일수록 w 에 더 작은 값이 곱해지어 가중치가 줄어들 것이고, 그렇게 되면 Overfitting을 감소시키는데 도움이 된다.



Q&A

질의응답 (10분)



끝. 감사합니다.

수업 듣느라 수고하셨습니다.

