



Chapter

2

Model Setting

1. 가중치 매개변수 최적화(optimization)
2. DNN 은닉층(Hidden layer)
3. 기울기 소실(vanishing gradient)
4. 과적합(overfitting) 방지

강의에 앞서서..

❖ 본 문서는 아래의 자료들을 활용하여 만들어 졌음을 알립니다

❖ 모두를 위한 딥러닝 강좌

- 네이버 Search & Clova AI 부분 리더 김성훈 교수님
- https://www.youtube.com/playlist?list=PLIMkM4tgfjnLSOjrEJN31gZATbcj_MpUm
- <https://www.edwith.org/boostcourse-dl-tensorflow/lecture/43739/>

❖ 스탠포드 대학 CNN 강좌

- Fei-Fei Li & Andrej Karpathy & Justin Johnson
- <http://cs231n.stanford.edu/slides/2020/>

CS231n: Convolutional Neural Networks for Visual Recognition

- This course, Prof. Fei-Fei Li & Justin Johnson & Serena Yeung
- Focusing on applications of deep learning to computer vision

강의에 앞서서..

❖ 밑바닥부터 시작하는 딥러닝

- <https://github.com/ExcelsiorCJH/DLFromScratch>

❖ DeepLearning Getting Started with TensorFlow

- <https://github.com/Jpub/TensorflowDeeplearning>

❖ TensorFlow Image Classification

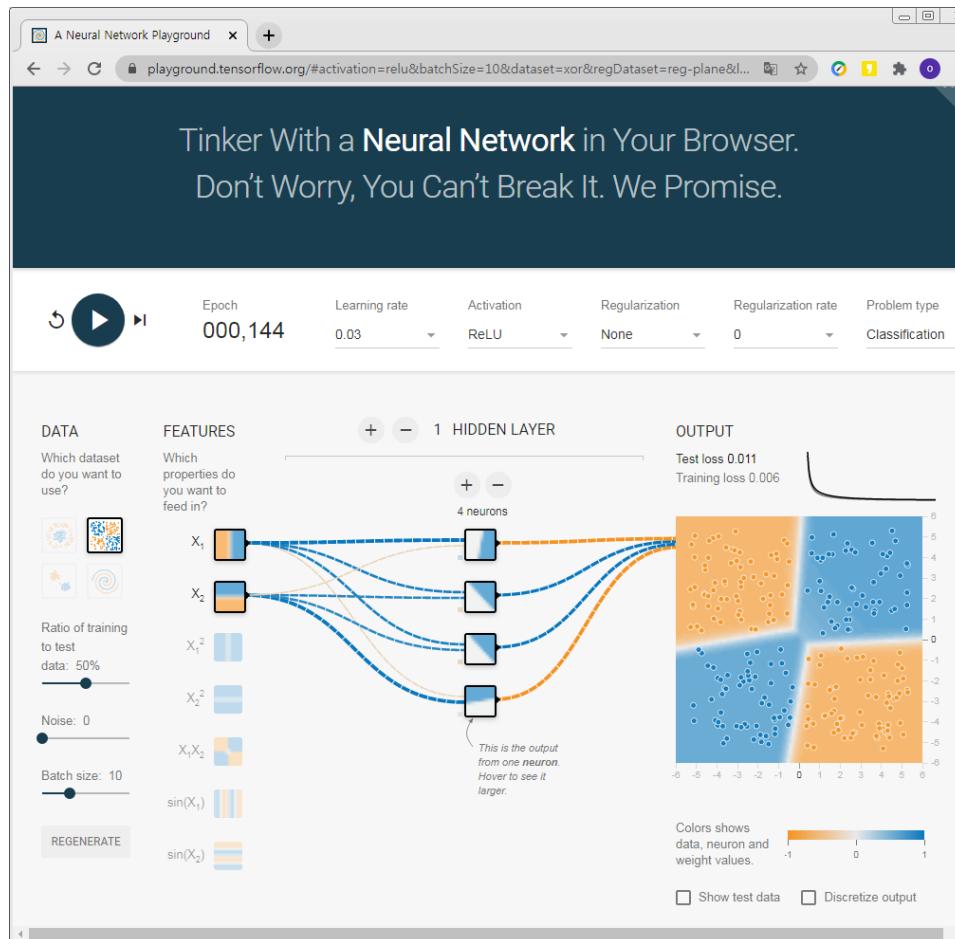
- <https://www.tensorflow.org/tutorials/images/classification>

❖ Hands-On Machine Learning

- <https://github.com/ExcelsiorCJH/Hands-On-ML>

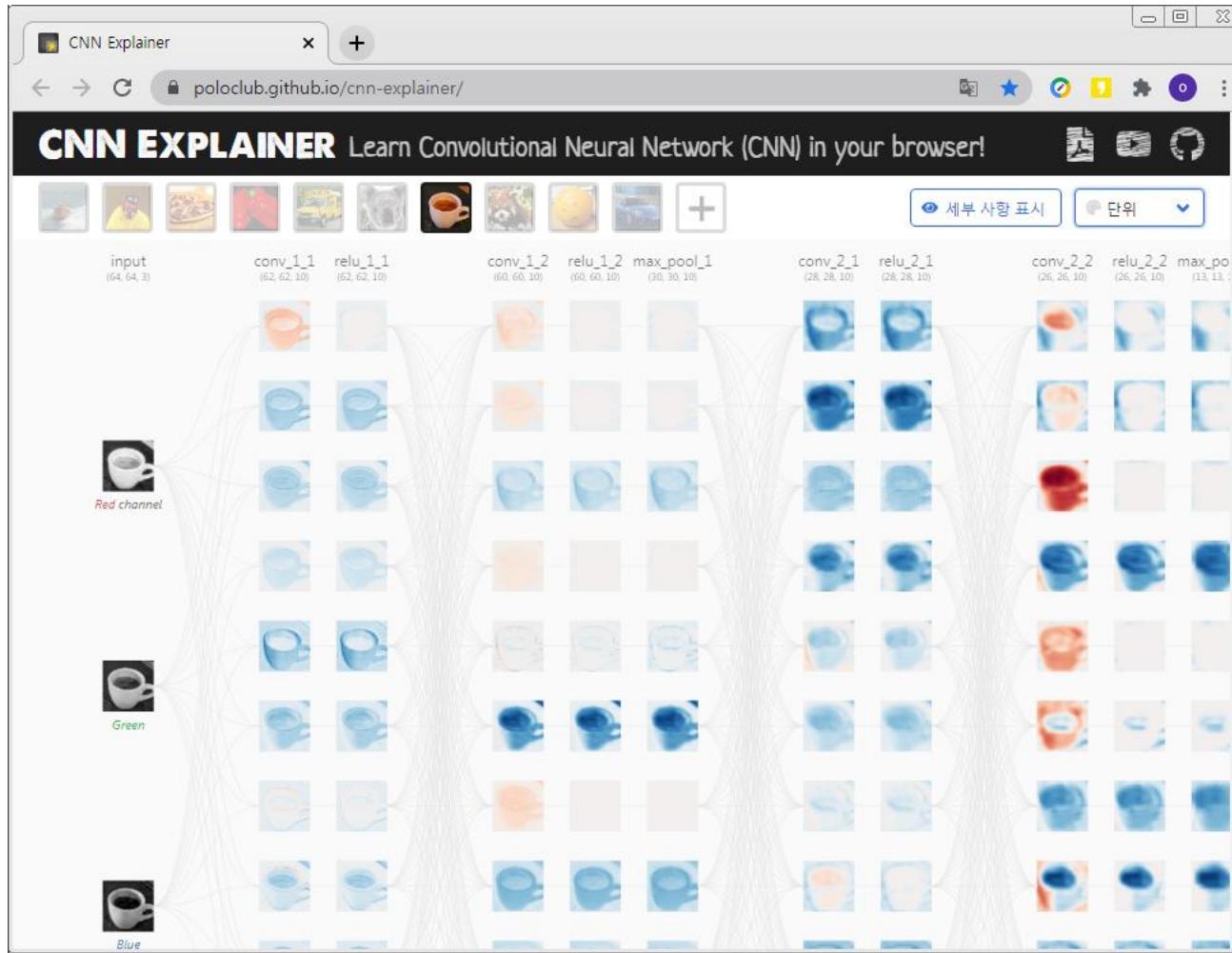
Neural Network Visualization

- ❖ 복수의 은닉층으로 구성된 이진 분류기
<https://playground.tensorflow.org/>



CNN explainer

❖ <https://poloclub.github.io/cnn-explainer/>



모델 설정 관련 기술들

❖ 가중치(w) 매개변수 최적화 방법(optimization)

- SGD(Stochastic Gradient Decent, 확률적 경사 하강법)
- Momentum
- AdaGrad (Adaptive Gradient)
- RMSProp(Root Mean Square Propagation)
- Adam(Adaptive Moment Estimation)

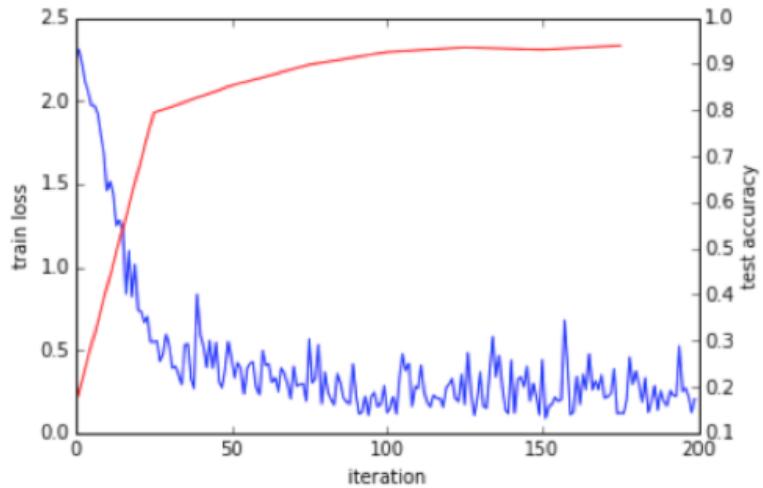
딥러닝 학습의 효율과 정확도를 높일 수 있다!

❖ 활성화 함수(Aactivation function)

❖ 가중치(w) 초기화(Weight Initialization)

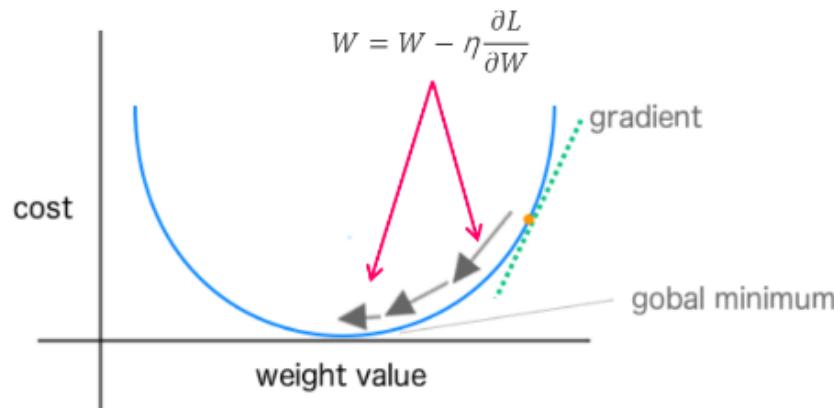
❖ 가중치 규제 (Weight Regularization)

❖ Drop out



가중치 매개변수 최적화(optimization)

- ✓ 신경망 학습의 목적은 **손실함수의 값을 최대한 낮추는** 파라미터(w, b)를 찾는 것
- ✓ 즉, 최적의 매개변수(파라미터)를 찾는 문제 → **최적화(optimization)**
- ✓ But, 신경망에서는 수식을 풀어 한번에 최적의 값을 찾을 수 있는 방법은 없음...
→ 층이 깊어질 수록 파라미터가 엄청나게 많아지므로
- ✓ 앞에서는 **확률적 경사 하강법(SGD)**을 이용해서 가중치 파라미터의 값을 여러 번 갱신하여 최적의 값을 구함



가중치 매개변수 최적화(optimization)

❖ Module: tf.keras.optimizers

- https://www.tensorflow.org/api_docs/python/tf/keras/optimizers

`class Adadelta`: Optimizer that implements the Adadelta algorithm.

`class Adagrad`: Optimizer that implements the Adagrad algorithm.

`class Adam`: Optimizer that implements the Adam algorithm.

`class Adamax`: Optimizer that implements the Adamax algorithm.

`class Ftrl`: Optimizer that implements the FTRL algorithm.

`class Nadam`: Optimizer that implements the NAdam algorithm.

`class Optimizer`: Updated base class for optimizers.

`class RMSprop`: Optimizer that implements the RMSprop algorithm.

`class SGD`: Stochastic gradient descent and momentum optimizer.

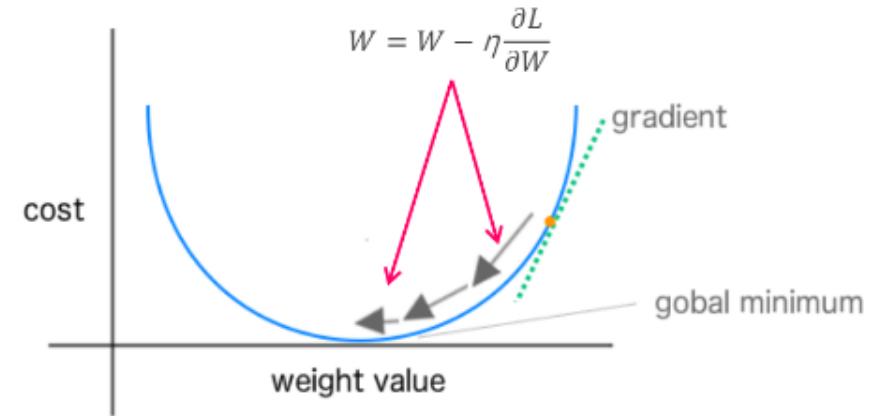
가중치 매개변수 최적화(optimization)

❖ SGD(Stochastic Gradient Decent, 확률적 경사하강법)

$$W \leftarrow W - \eta \frac{\partial L}{\partial W}$$

← *W*에 대한
손실함수(*L*)의 기울기

```
class SGD:  
  
    """확률적 경사 하강법 ( Stochastic Gradient Descent ) """  
  
    def __init__(self, lr=0.01):  
        self.lr = lr  
  
    def update(self, params, grads):  
        for key in params.keys():  
            params[key] -= self.lr * grads[key]
```

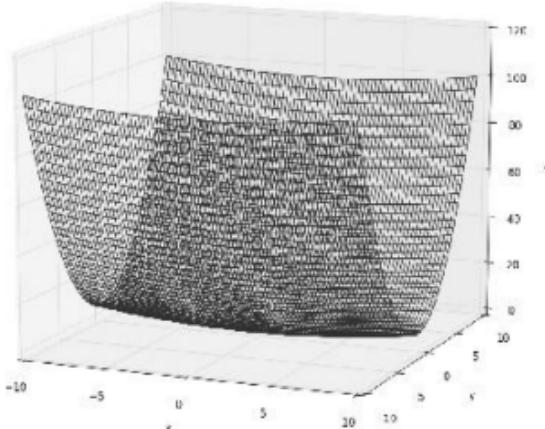


가중치 매개변수 최적화(optimization)

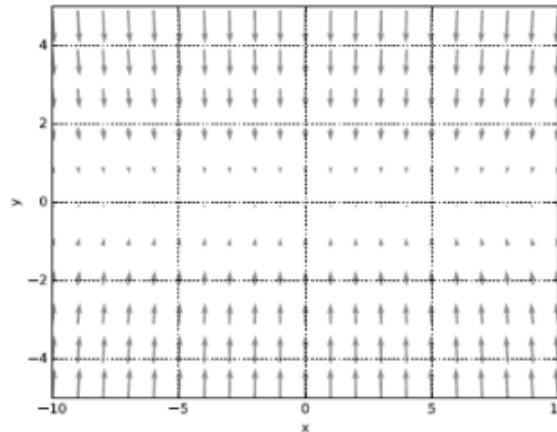
❖ SGD 단점

- ✓ 비등방성(anisotropy) 함수 즉, 방향에 따라 기울기가 달라지는 함수에서는 탐색 경로가 비효율적임

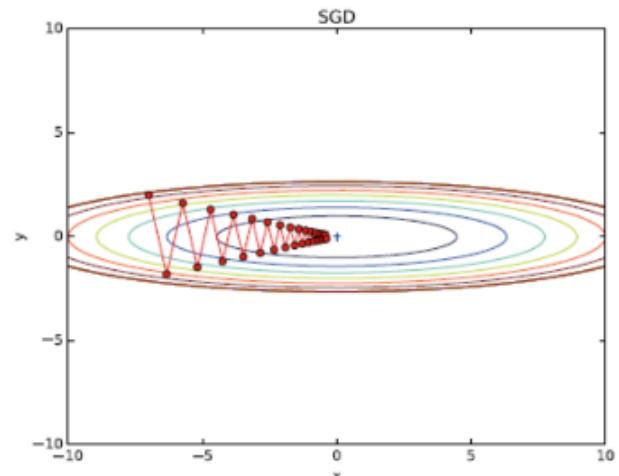
$$f(x, y) = \frac{1}{20}x^2 + y^2$$



그래프



기울기

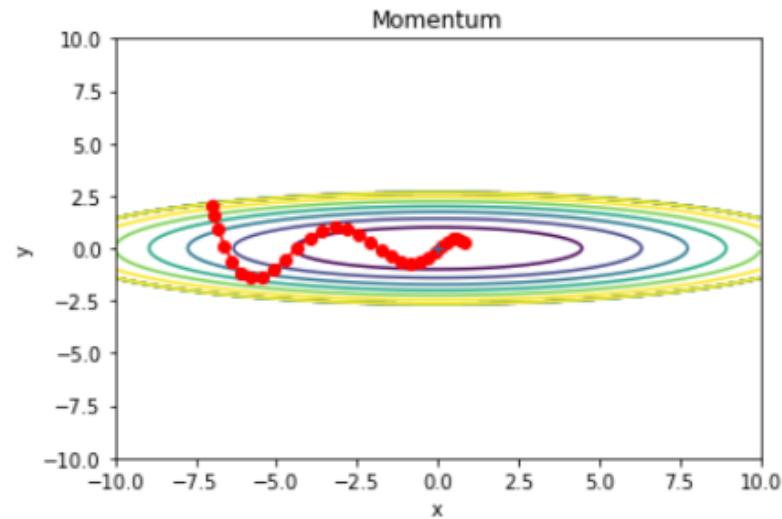


갱신 경로

가중치 매개변수 최적화(optimization)

❖ 모멘텀(Momentum)

- ✓ 모멘텀은 '운동량'을 뜻하는 단어로, 일종의 '관성'을 주는 방법
- ✓ 속도 v 라는 개념을 추가하여 기울기 방향으로 힘을 받아 물체가 가속
- ✓ 기울기(gradients)를 통해 이동하는 방향과는 별개로, 과거에 이동했던 방식을 기억하면서 그 방향으로 일정 정도를 추가적으로 이동하는 방식



가중치 매개변수 최적화(optimization)

❖ 모멘텀(Momentum)

- ✓ 모멘텀 소스코드

```
class Momentum:  
    def __init__(self, lr=0.01, momentum=0.9):  
        self.lr = lr  
        self.momentum = momentum  
        self.v = None  
  
    def update(self, params, grads):  
        if self.v is None:  
            self.v = {}  
            for key, val in params.items():  
                self.v[key] = np.zeros_like(val)  
  
        for key in params.keys():  
            self.v[key] = self.momentum*self.v[key] - self.lr*grads[key]  
            params[key] += self.v[key]  
  

$$v \leftarrow \alpha v - \eta \frac{\partial L}{\partial W}$$
  

$$W \leftarrow W + v$$

```

가중치 매개변수 최적화(optimization)

❖ AdaGrad (Adaptive Gradient)

- ✓ 신경망 학습에서는 learning rate(η) 값이 중요 → 너무 작으면 오래 걸리고, 너무 크면 발산
- ✓ AdaGrad는 처음에는 크게 학습하다가 조금씩 작게 학습하는 방식으로 learning rate를 줄여나감
 \rightarrow learning rate decay

```
class AdaGrad:

    def __init__(self, lr=0.01):
        self.lr = lr
        self.h = None

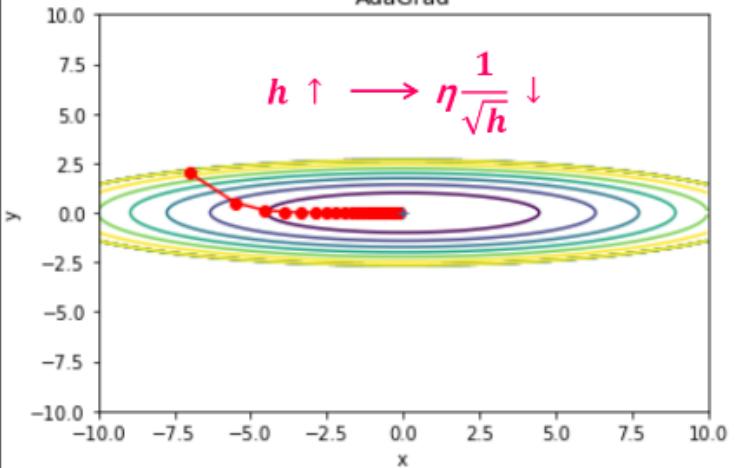
    def update(self, params, grads):
        if self.h is None:
            self.h = {}
            for key, val in params.items():
                self.h[key] = np.zeros_like(val)

        for key in params.keys():
            self.h[key] += grads[key] * grads[key]
            params[key] -= self.lr*grads[key]/(np.sqrt(self.h[key])+1e-7)
```

초기 $h = 0$

$$h \leftarrow h + \frac{\partial L}{\partial W} * \frac{\partial L}{\partial W}$$

$$W \leftarrow W - \eta * \frac{1}{\sqrt{h + \epsilon}} * \frac{\partial L}{\partial W}$$



가중치 매개변수 최적화(optimization)

❖ RMSProp(Root Mean Square Propagation)

- ✓ 딥러닝의 대가 제프리 힌튼(Geoffrey Hinton)이 제안한 방법
- ✓ AdaGrad의 단점인 학습속도가 급진적으로 단조 감소하는 방법을 보완함 → 오래 될 수록 거의 학습이 되지 않음
- ✓ AdaGrad에서 단순히 기울기(gradient, $\frac{\partial L}{\partial W}$)를 제곱하는 것이 아닌 이동평균(moving average)를 사용

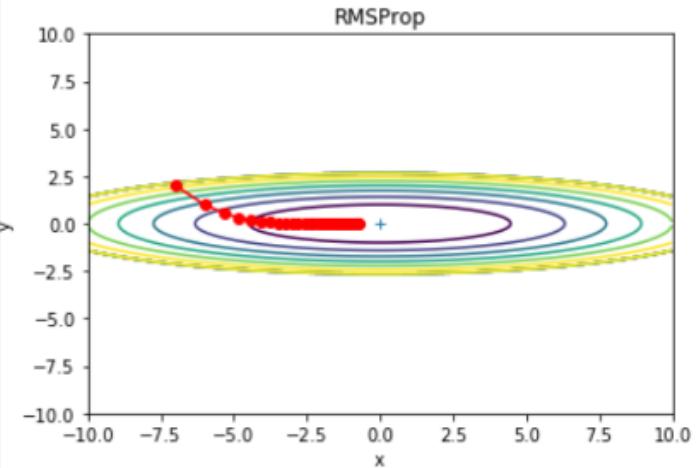
```
class RMSprop:
    def __init__(self, lr=0.01, decay_rate = 0.95):
        self.lr = lr
        self.decay_rate = decay_rate
        self.h = None

    def update(self, params, grads):
        if self.h is None:
            self.h = {}
            for key, val in params.items():
                self.h[key] = np.zeros_like(val)

        for key in params.keys():
            self.h[key] *= self.decay_rate
            self.h[key] += (1 - self.decay_rate) * grads[key] * grads[key]
            params[key] -= self.lr*grads[key]/(np.sqrt(self.h[key])+1e-7)
```

$$h \leftarrow \gamma h + (1 - \gamma) \frac{\partial L}{\partial W} * \frac{\partial L}{\partial W}$$

$$W \leftarrow W - \eta * \frac{1}{\sqrt{h + \epsilon}} * \frac{\partial L}{\partial W}$$



가중치 매개변수 최적화(optimization)

❖ Adam(Adaptive Moment Estimation)

- ✓ Adam은 RMSProp과 Momentum을 결합한 방법
- ✓ Adam에서는 m 과 v 가 초기값은 0이기 때문에 학습 초반부에서는 0에 가깝게 편향(bias) 되어있을 것이라고 판단하여 unbiased하게 작업을 해줌

$$\begin{aligned} m &\leftarrow m + (1 - \beta_1) \left(\frac{\partial L}{\partial W} - m \right) && \text{※ } \beta_1 \text{ 과 } \beta_2 \text{는 모멘텀용 계수로써,} \\ &= \beta_1 m + (1 - \beta_1) \frac{\partial L}{\partial W} && \text{※ } \beta_1 \text{은 } 0.9 \text{ } \beta_2 \text{는 } 0.999 \text{로 설정} \end{aligned}$$

Momentum 과
유사

$$\begin{aligned} v &\leftarrow v + (1 - \beta_2) \left(\frac{\partial L}{\partial W} * \frac{\partial L}{\partial W} - v \right) \\ &= \beta_2 v + (1 - \beta_2) \frac{\partial L}{\partial W} * \frac{\partial L}{\partial W} && \text{→ RMSProp 과
유사} \end{aligned}$$

$$W \leftarrow W - \eta * \boxed{\frac{\sqrt{1 - \beta_2^t}}{1 - \beta_1^t} * \frac{m}{\sqrt{v + \epsilon}}} \rightarrow \begin{aligned} \hat{m}_t &= \frac{m_t}{1 - \beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1 - \beta_2^t} \end{aligned}$$

가중치 매개변수 최적화(optimization)

❖ Adam(Adaptive Moment Estimation)

```
class Adam:

    def __init__(self, lr=0.001, beta1=0.9, beta2=0.999):
        self.lr = lr
        self.beta1 = beta1
        self.beta2 = beta2
        self.iter = 0
        self.m = None
        self.v = None

    def update(self, params, grads):
        if self.m is None:
            self.m, self.v = {}, {}
            for key, val in params.items():
                self.m[key] = np.zeros_like(val)
                self.v[key] = np.zeros_like(val)

        self.iter += 1
        lr_t = self.lr * np.sqrt(1.0 - self.beta2**self.iter) / (1.0 - self.beta1**self.iter)

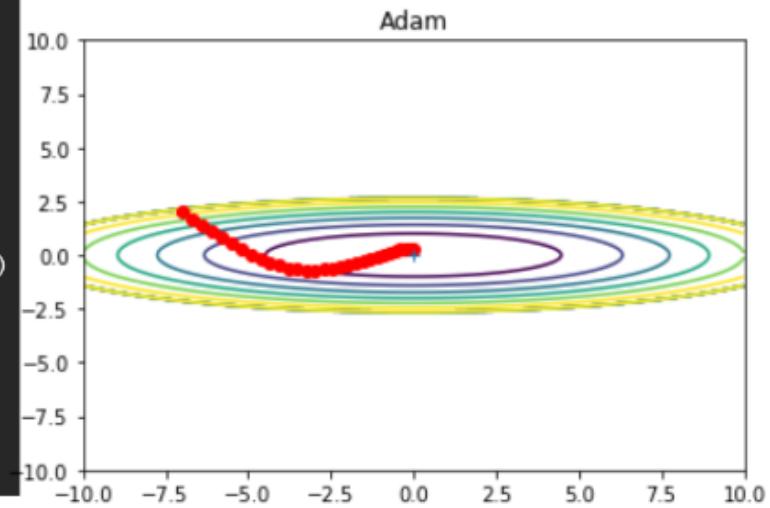
        for key in params.keys():
            self.m[key] += (1 - self.beta1) * (grads[key] - self.m[key])
            self.v[key] += (1 - self.beta2) * (grads[key]**2 - self.v[key])

            params[key] -= lr_t * self.m[key] / (np.sqrt(self.v[key]) + 1e-7)
```

$$m \leftarrow m + (1 - \beta_1) \left(\frac{\partial L}{\partial W} - m \right)$$

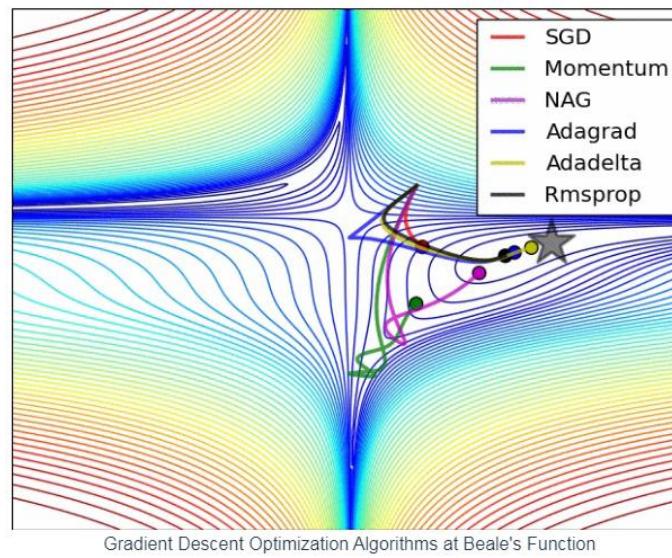
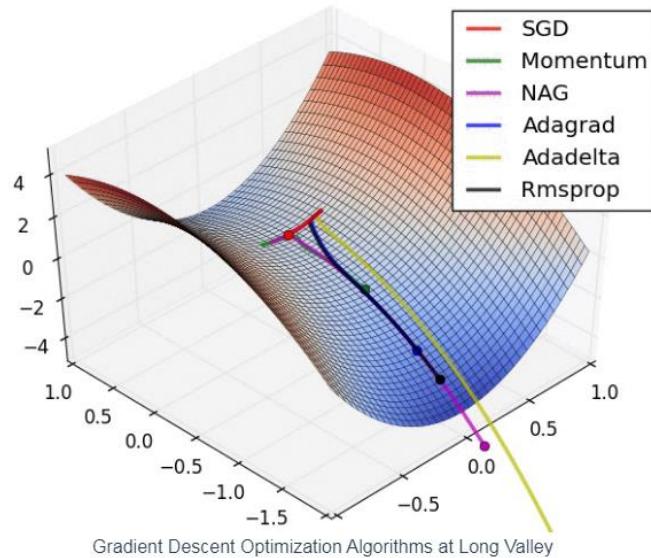
$$v \leftarrow v + (1 - \beta_2) \left(\frac{\partial L}{\partial W} * \frac{\partial L}{\partial W} - v \right)$$

$$W \leftarrow W - \eta * \frac{\sqrt{1 - \beta_2^t}}{1 - \beta_1^t} * \frac{m}{\sqrt{v + \epsilon}}$$



가중치 매개변수 최적화(optimization)

❖ Optimizer 비교



<http://shuuki4.github.io/deep%20learning/2016/05/20/Gradient-Descent-Algorithm-Overview.html>

가중치 매개변수 최적화(optimization)

❖ Optimizer 비교 (MNIST)

- ✓ 어떤 Optimizer가 좋다고 딱 골라 말할 수는 없음
- ✓ 상황에 따라 적절한 Optimizer를 선택 해야함
- ✓ But, 요즘 추세는 Adam을 많이 사용하는 듯함

Lab10_1_mnist_CNN_Keras-optimizer.ipynb

Hyper Parameters

```
learning_rate = 0.01
training_epochs = 5
batch_size = 100

tf.random.set_seed(777)
```

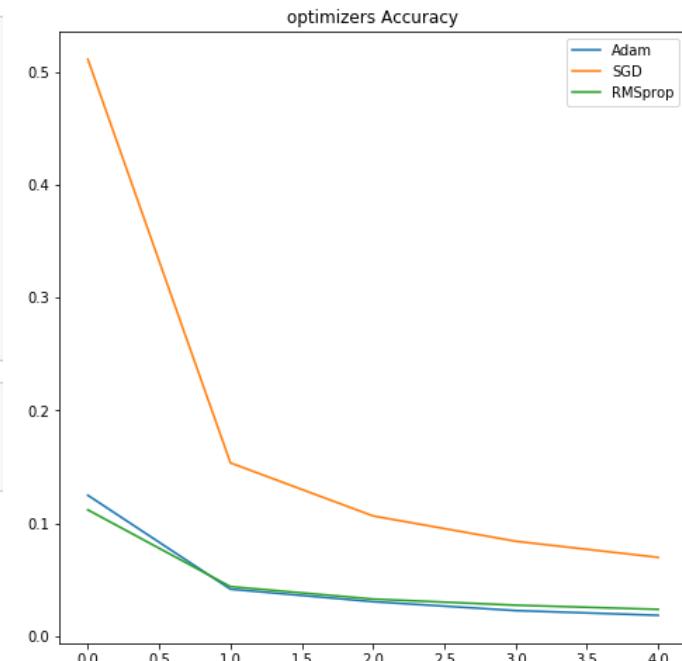
```
model1 = Sequential([
    Conv2D(32, 3, activation='relu', padding='same', input_shape=(28, 28, 1)),
    MaxPooling2D(),
    Conv2D(filters=64, kernel_size=3, activation='relu', padding='same'),
    MaxPooling2D(),

    Flatten(),
    Dense(512, activation=tf.nn.relu),
    Dropout(0.2),
    Dense(10, activation=tf.nn.softmax)
])

model1.compile(optimizer='adam',
               loss='sparse_categorical_crossentropy',
               metrics=['accuracy'])

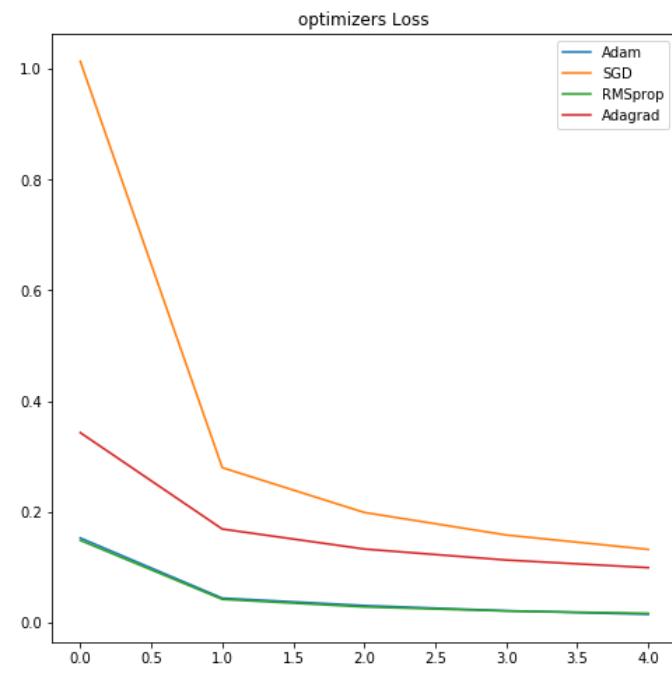
model2.compile(optimizer='SGD',
               loss='sparse_categorical_crossentropy',
               metrics=['accuracy'])

model3.compile(optimizer='RMSprop',
               loss='sparse_categorical_crossentropy',
               metrics=['accuracy'])
```



QUIZ!

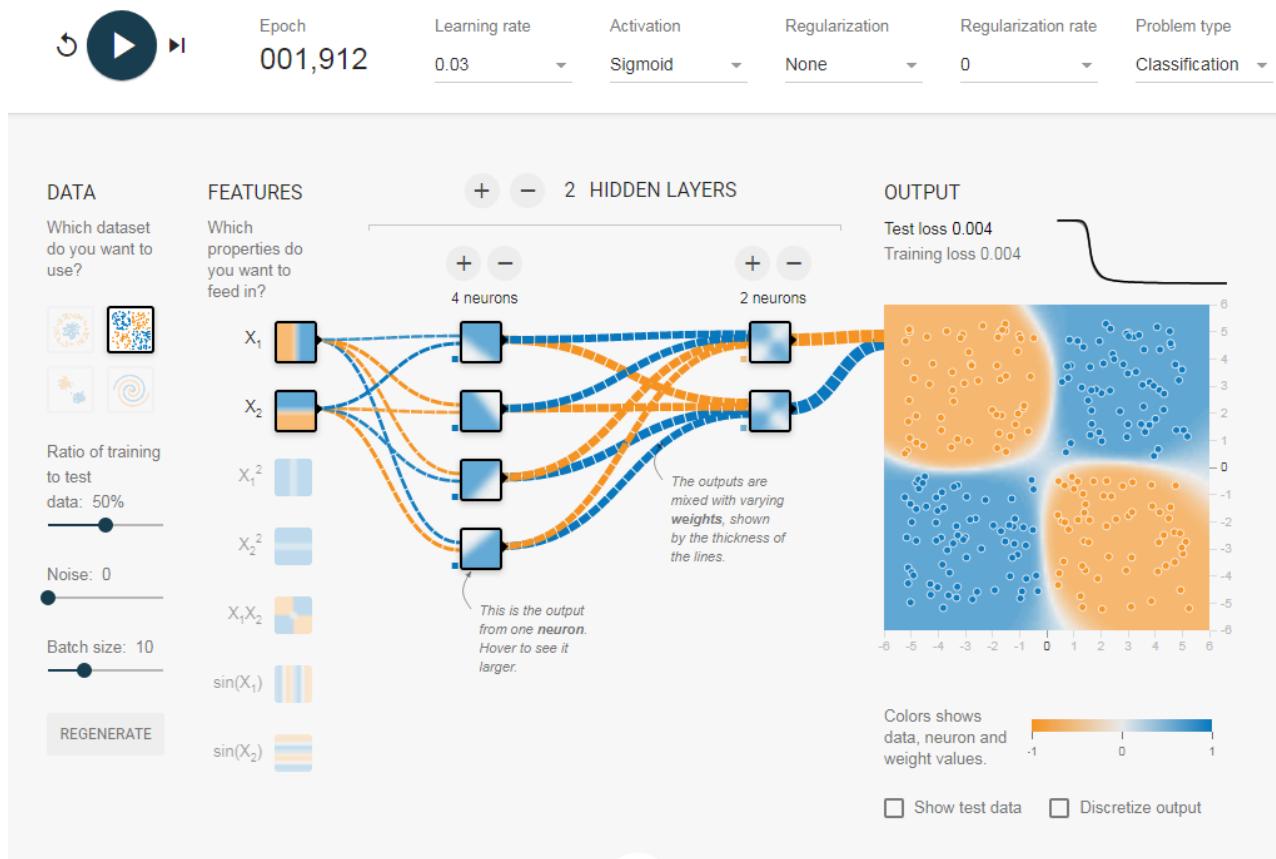
“Lab10_1_mnist_CNN_Keras-optimizer.ipynb” 에
optimizer “Adagrad” 방법을 추가하여
Loss 비교 그래프를 확인하세요



Hidden layer

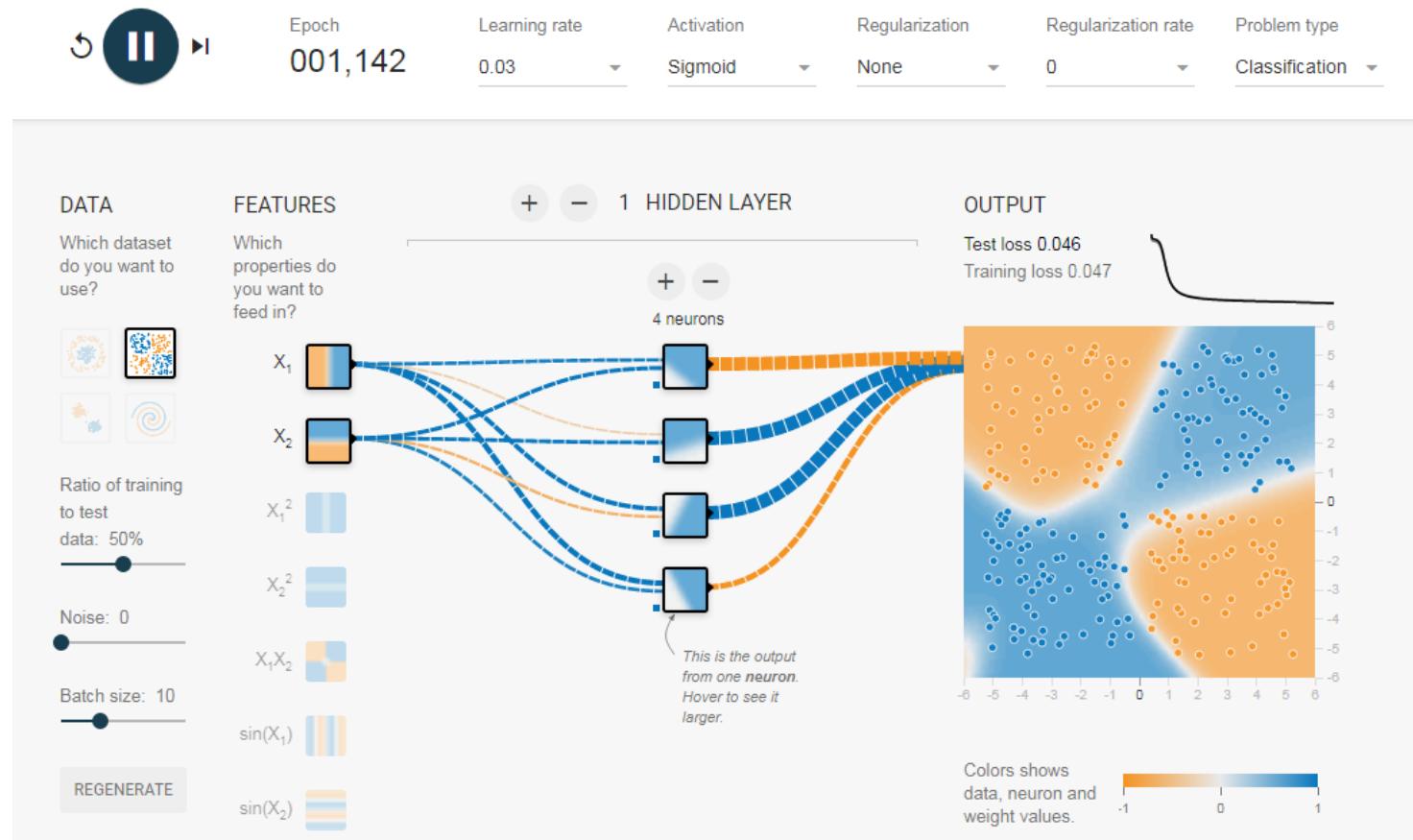
❖ Hidden layer, Hidden node, activation function

- 은닉층, 은닉층 노드, 활성화 함수의 역할



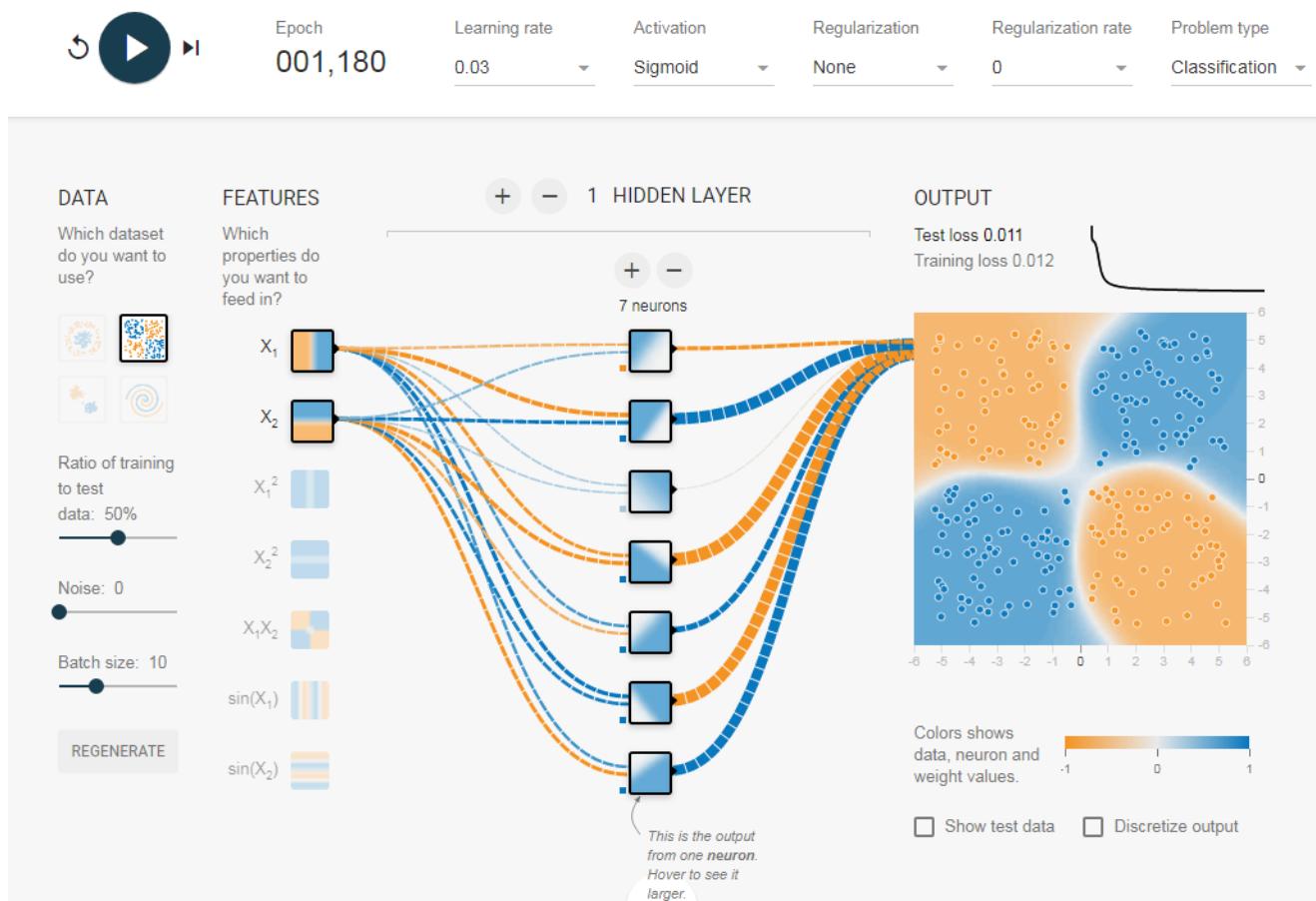
Hidden layer

- Hidden node 는 이전 노드의 특징 개수를 표현



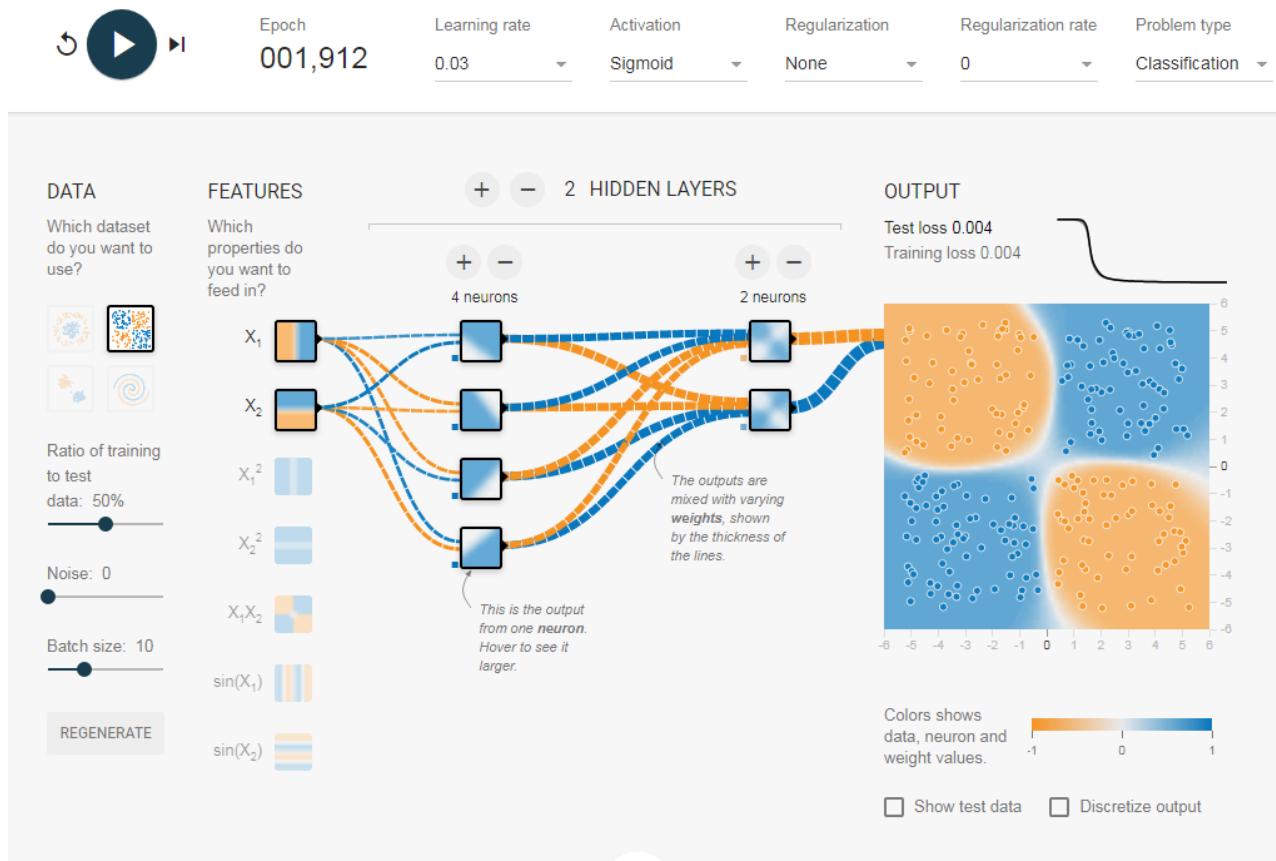
Hidden layer

- Hidden node 는 이전 노드의 특징 개수를 표현



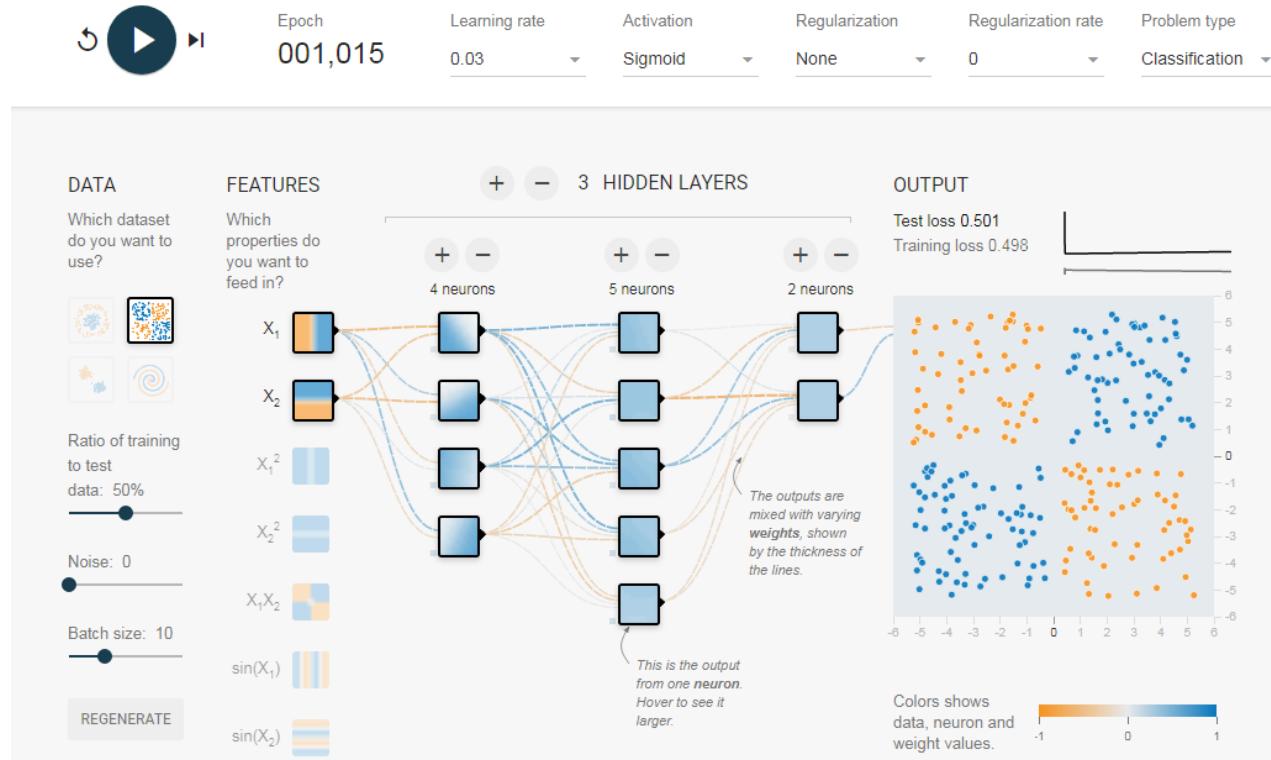
Hidden layer

- Hidden layer는 이전 노드의 특징에 기반한 분류기



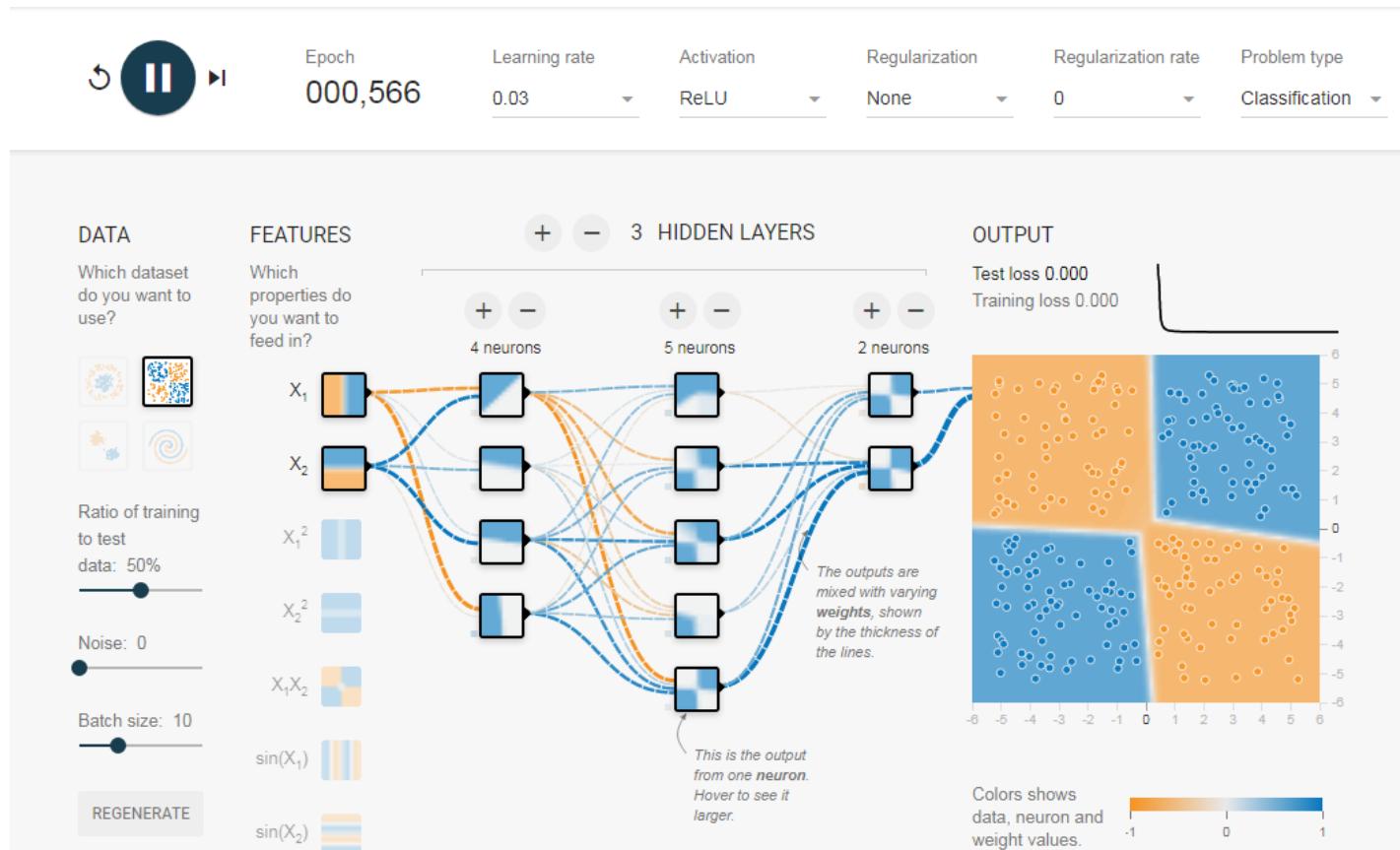
Hidden layer

- Activation 함수가 Sigmoid 인 경우 은닉층이 늘어날수록 학습이 안 되는 현상 발생
- 기울기 소실(vanishing gradient), 폭주(exploding)



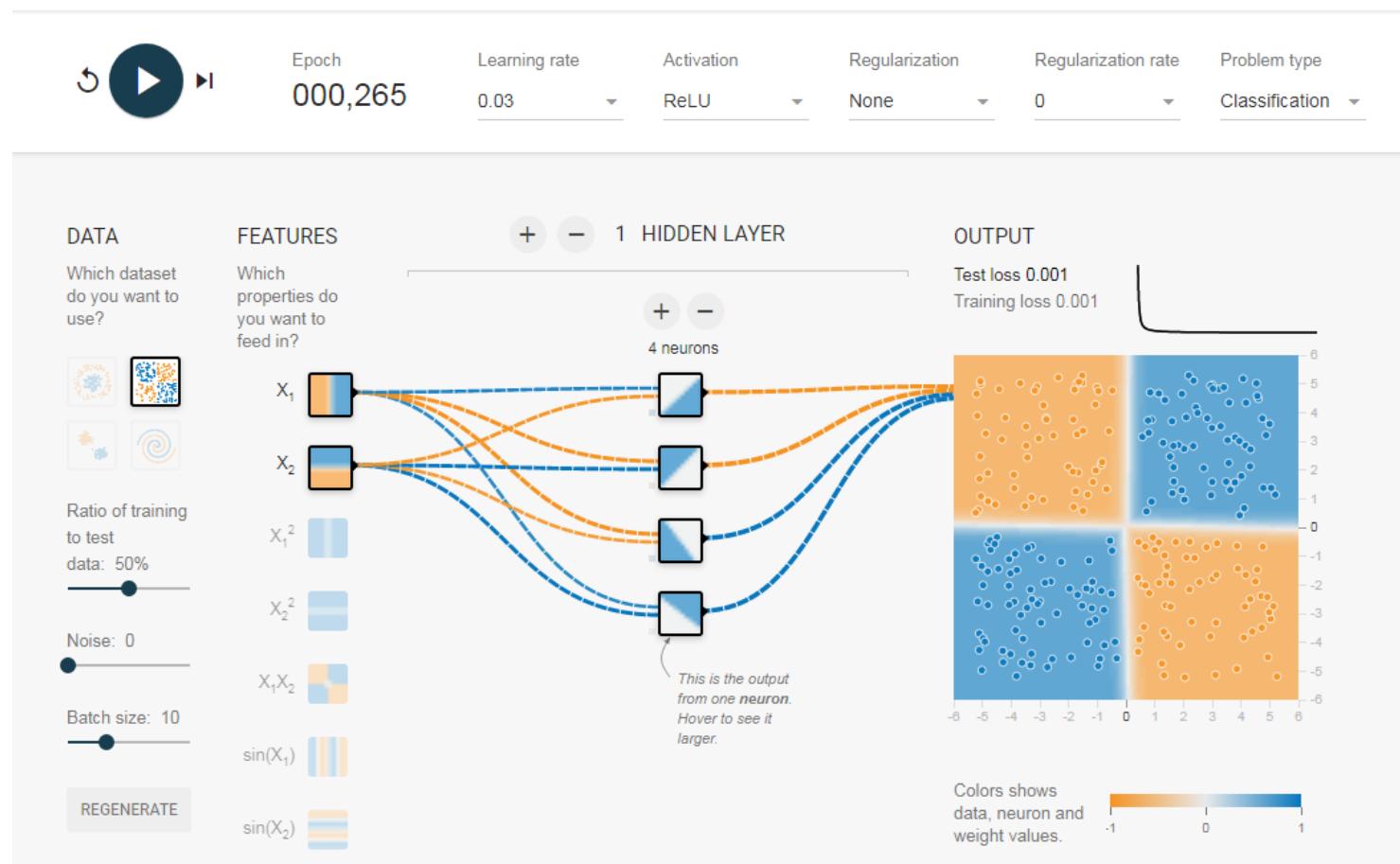
Hidden layer

❖ ReLU 함수는 학습이 진행됨



Hidden layer

❖ ReLU 함수



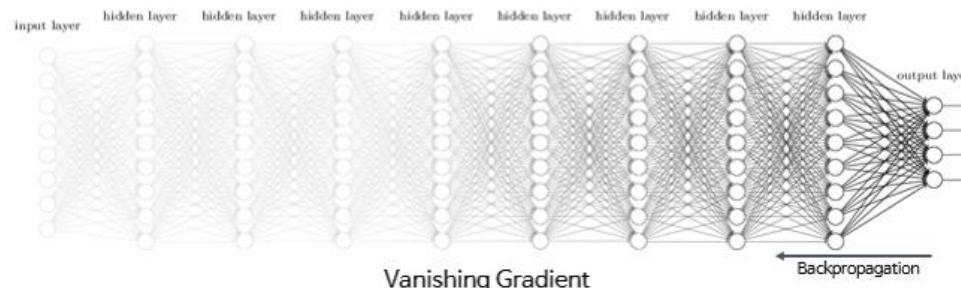
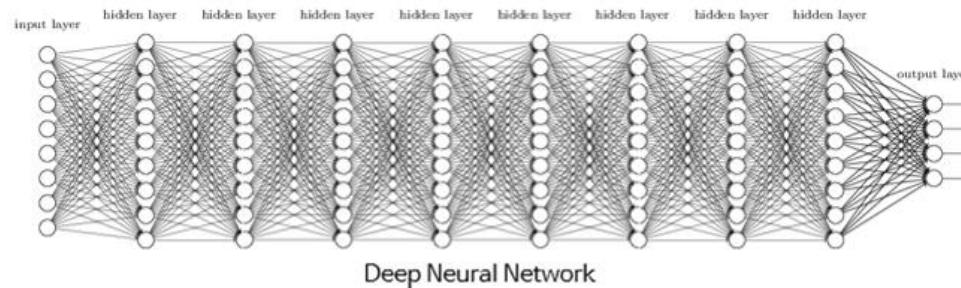
Deep Neural Network

❖ 기울기 소실(vanishing gradient)

- 심층신경망에서는 역전파 알고리즘이 입력 층으로 전달될 때 기울기가 점점 작아져서 가중치 매개변수가 수정이 안 되는 경우 발생

❖ 기울기 폭주(exploding gradient), 폭주(exploding)

- 역전파에서 기울기가 점점 커져 입력층으로 갈수록 가중치 매개변수가 기하급 수적으로 커지게 되는 경우

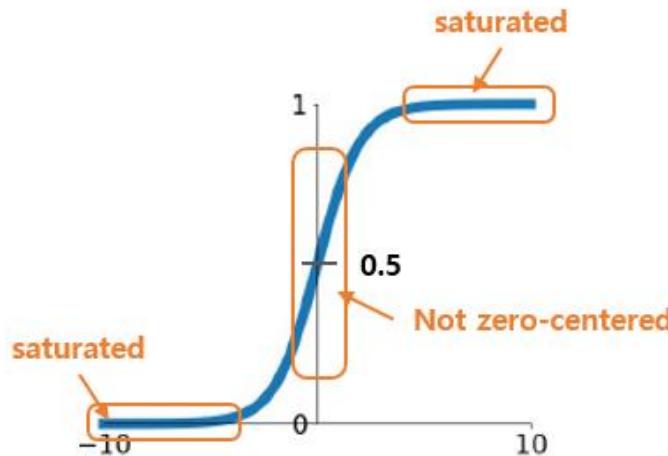


기울기 소실(vanishing gradient)

❖ Sigmoid 함수의 문제점

- 입력의 절대값이 크게 되면 0이나 1로 수렴하여 이러한 뉴런들은 기울기가 소멸(kill)되며 역전파가 진행됨에 따라 아래 층(layer)으로 신호 전달이 안됨
- 원점 중심(Not zero-centered)이 아니므로 평균이 0.5이며, 시그모이드 함수는 항상 양수를 출력하기 때문에 출력의 가중치 합이 입력의 가중치 합보다 커질 가능성이 높다(편향 이동(bias shift))
- 각 레이어를 지날 때마다 계속 커져 가장 높은 층에서는 활성화 함수의 출력이 0이나 1로 수렴하여 기울기 소실 문제가 일어남

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



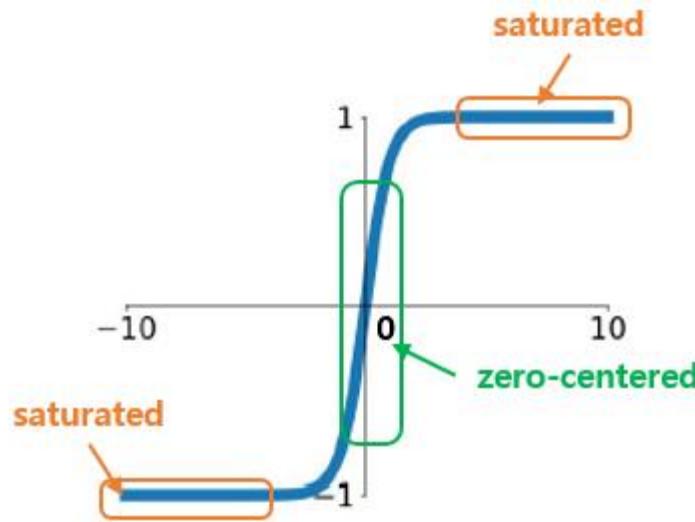
기울기 소실(vanishing gradient)

❖ 하이퍼볼릭 탄젠트 함수(tanh, hyperbolic tangent)

- 시그모이드 함수와 유사하나 입력값의 총합을 -1에서 1사이의 값으로 변환해 주며, 원점 중심(zero-centered)이기 때문에, 시그모이드와 달리 편향 이동이 일어나지 않으나 입력 값이 클 경우 기울기 소멸 현상 존재

$$\begin{aligned}\tanh(x) &= \frac{1 - e^{-x}}{1 + e^{-x}} \\ &= \frac{2}{1 + e^{-2x}} - 1\end{aligned}$$

$$\tanh(x) = 2\sigma(2x) - 1$$

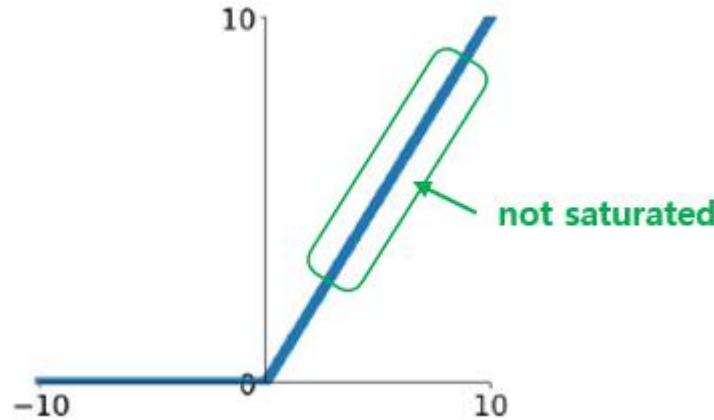


기울기 소실(vanishing gradient)

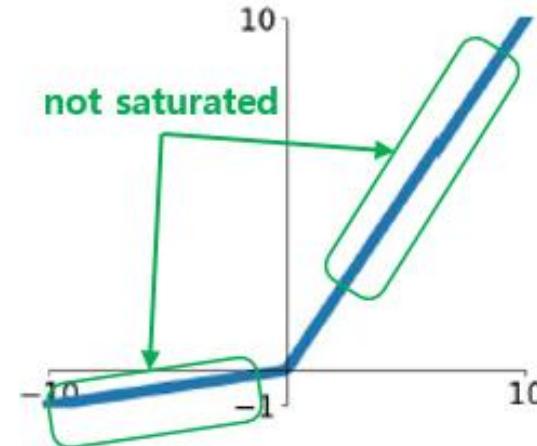
❖ ReLU

- sigmoid 계열과는 다른 활성화 함수
- sigmoid/tanh에 비해 계산속도 빠름
- LeakyReLU : 0일 경우 활성화 되지 않는 문제를 해결

$$\text{ReLU}(x) = \max(0, x)$$



$$\text{LeakyReLU}_\alpha(x) = \max(\alpha x, x) \quad \alpha = 0.01$$



가중치의 초기화(Weight Initialization)

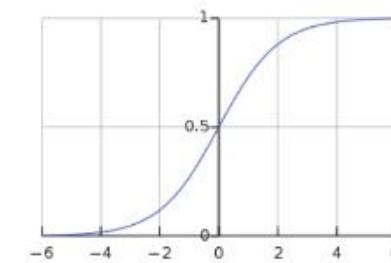
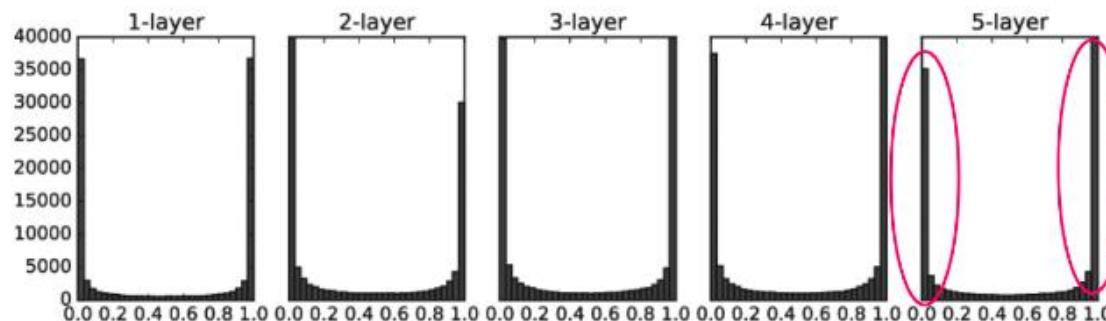
❖ 가중치의 초기값에 따라 학습 결과에 영향을 줄 수 있다.

- ✓ 초기값을 0 또는 동일한 값으로 설정하게 되면 오차역전파시 모든 가중치의 값이 똑같이 갱신이 됨
→ 갱신을 해도 여전히 같기 때문에 학습의 의미가 없음
- ✓ 따라서, 이러한 문제를 해결하기 위해서는 초기값을 동일하지 않도록 **랜덤하게** 설정해야 함
- ✓ 특히, 가중치의 값이 클 경우 Overfitting이 일어날 확률이 높기 때문에 가중치를 가능한 작게 만들어 줘야함
→ *weight decay*

가중치의 초기화(Weight Initialization)

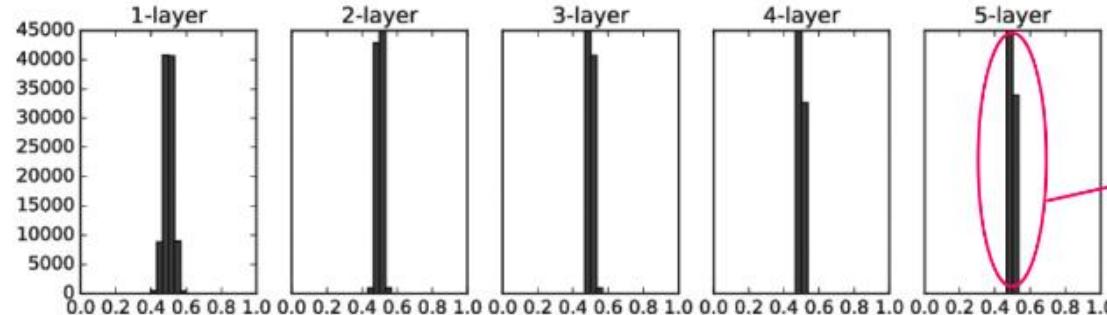
❖ 은닉층 활성화 함수의 출력값 분포

- ✓ 가중치를 표준편차가 1인 정규분포로 초기화할 때의 각 층의 활성화 값 분포



기울기 소실
(gradient vanishing)

- ✓ 가중치를 표준편차가 0.01인 정규분포로 초기화할 때의 각 층의 활성화 값 분포

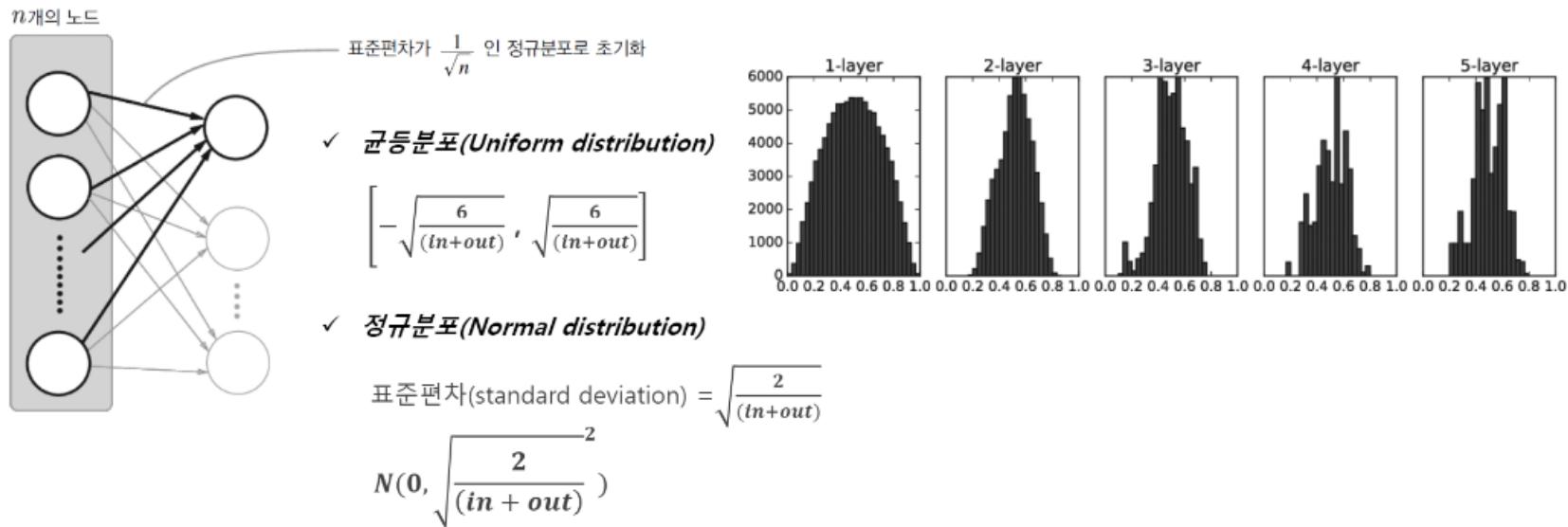


가중치가 거의 같은
값이므로 의미 없음

가중치의 초기화(Weight Initialization)

❖ Xavier 초기

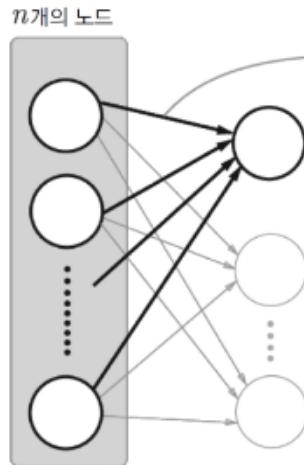
- Sigmoid 계열 함수(sigmoid/tanh) 에 적합한 초기값
- Xavier Glorot 이 제안
- 각 층의 활성화 값들을 광범위하게 분포시킬 목적
- 이전 노드의 개수가 N일 경우 $\frac{1}{\sqrt{n}}$ 이 되도록 설정



가중치의 초기화(Weight Initialization)

❖ He 초기값

- ✓ ReLU에 특화된 가중치 초기값 설정 방법
- ✓ 카이밍 히(Kaiming He)의 이름을 따 **He 초기값**이라고 함
- ✓ **He 초기값**: 초기값의 표준편차가 $\frac{2}{\sqrt{n}}$ 이 되도록 설정



표준편차가 $\frac{2}{\sqrt{n}}$ 인 정규분포로 초기화

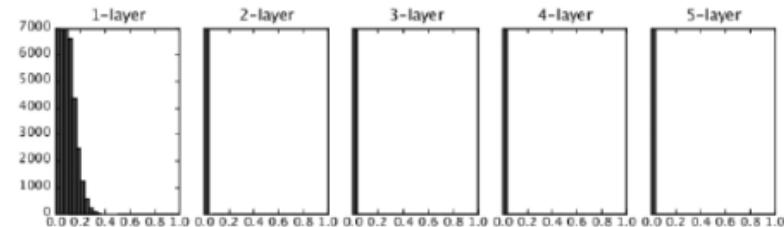
✓ **균등분포(Uniform distribution)**

$$\left[-\sqrt{\frac{6}{in}}, \sqrt{\frac{6}{in}} \right]$$

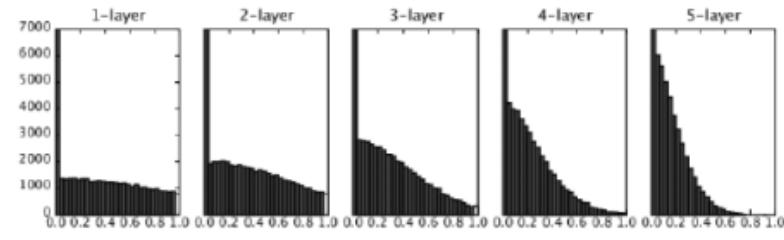
✓ **정규분포(Normal distribution)**

$$\text{표준편차(standard deviation)} = \sqrt{\frac{2}{in}}$$

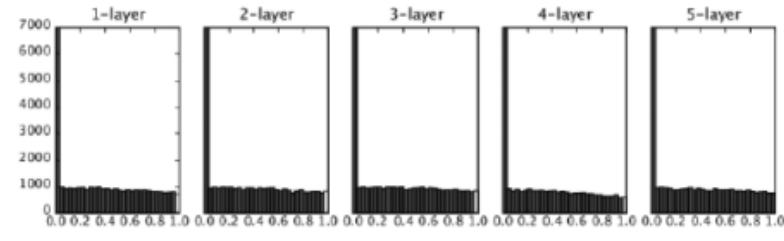
$$N(0, \sqrt{\frac{2}{in}})$$



표준편차가 0.01인 정규분포를 가중치 초기값으로 사용한 경우



Xavier 초기값을 사용한 경우



He 초기값을 사용한 경우

가중치의 초기화(Weight Initialization)

❖ Keras에서 제공되는 가중치 초기화

- 임의 값으로 초기화(random_uniform)
- Xavier 초기화(glorot_uniform, glorot_normal) , default
- He 초기화(he_uniform, he_normal)

```
tf.keras.layers.Conv2D(  
    filters, kernel_size, strides=(1, 1), padding='valid', data_format=None,  
    dilation_rate=(1, 1), activation=None, use_bias=True,  
    kernel_initializer='glorot_uniform', bias_initializer='zeros',  
    kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None,  
    kernel_constraint=None, bias_constraint=None, **kwargs  
)
```

가중치의 초기화(Weight Initialization)

(1) SGD Optimizer, activation='sigmoid', kernel_initializer='random_uniform'

- (1) SGD Optimizer, activation='sigmoid', kernel_initializer='random_uniform'

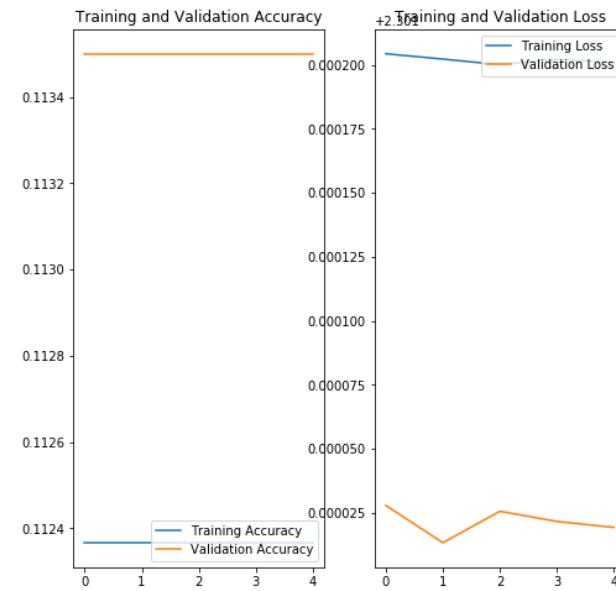
```
model1 = Sequential([
    Conv2D(32, 3, activation='sigmoid', padding='same', input_shape=(28, 28, 1),
           kernel_initializer='random_uniform'),
    MaxPooling2D(),
    Conv2D(filters=64, kernel_size=3, activation='sigmoid', padding='same'),
    MaxPooling2D(),

    Flatten(),
    Dense(512, activation=tf.nn.relu),
    Dropout(0.2),
    Dense(10, activation=tf.nn.softmax)
])

model1.compile(optimizer='SGD',
               loss='sparse_categorical_crossentropy',
               metrics=['accuracy'])

history11 = model1.fit(train_images, train_labels, validation_data=(test_images, test_labels),
                       batch_size=batch_size, epochs=training_epochs)
```

Train on 60000 samples, validate on 10000 samples
Epoch 1/5
60000/60000 [=====] - 34s 569us/sample - loss: 2.3012 - accuracy: 0.1124 - val_loss:
2.3010 - val_accuracy: 0.1135
Epoch 2/5
60000/60000 [=====] - 33s 558us/sample - loss: 2.3012 - accuracy: 0.1124 - val_loss:
2.3010 - val_accuracy: 0.1135
Epoch 3/5
60000/60000 [=====] - 34s 562us/sample - loss: 2.3012 - accuracy: 0.1124 - val_loss:
2.3010 - val_accuracy: 0.1135
Epoch 4/5
60000/60000 [=====] - 34s 564us/sample - loss: 2.3012 - accuracy: 0.1124 - val_loss:
2.3010 - val_accuracy: 0.1135
Epoch 5/5
60000/60000 [=====] - 33s 554us/sample - loss: 2.3012 - accuracy: 0.1124 - val_loss:
2.3010 - val_accuracy: 0.1135



가중치의 초기화(Weight Initialization)

(2)SGD Optimizer, activation='sigmoid', kernel_initializer='glorot_normal'

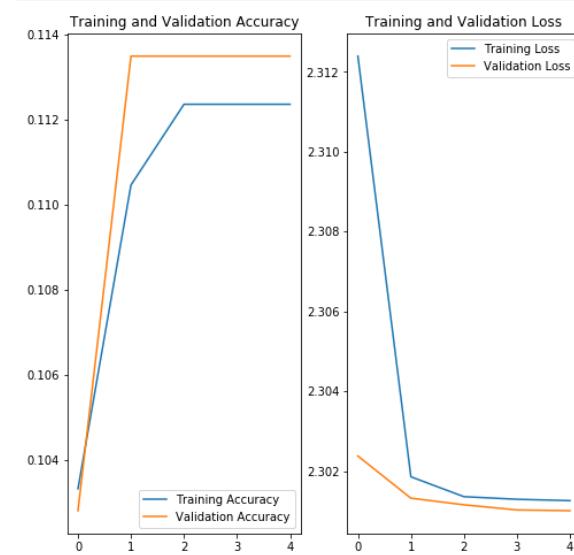
```
model2 = Sequential([
    Conv2D(32, 3, activation='sigmoid', padding='same', kernel_initializer='glorot_normal', input_shape=(28, 28, 1)),
    MaxPooling2D(),
    Conv2D(filters=64, kernel_size=3, activation='sigmoid', padding='same'),
    MaxPooling2D(),

    Flatten(),
    Dense(512, activation=tf.nn.relu),
    Dropout(0.2),
    Dense(10, activation=tf.nn.softmax)
])
```

```
model2.compile(optimizer='SGD',
               loss='sparse_categorical_crossentropy',
               metrics=['accuracy'])
```

```
history22 = model2.fit(train_images, train_labels, validation_data=(test_images, test_labels), batch_size=batch_size,
                       epochs=5, verbose=1)
```

Train on 60000 samples, validate on 10000 samples
Epoch 1/5
60000/60000 [=====] - 35s 580us/sample - loss: 2.3124 - accuracy: 0.1033 - val_loss:
2.3024 - val_accuracy: 0.1028
Epoch 2/5
60000/60000 [=====] - 35s 585us/sample - loss: 2.3019 - accuracy: 0.1105 - val_loss:
2.3013 - val_accuracy: 0.1135
Epoch 3/5
60000/60000 [=====] - 35s 579us/sample - loss: 2.3014 - accuracy: 0.1124 - val_loss:
2.3012 - val_accuracy: 0.1135
Epoch 4/5
60000/60000 [=====] - 35s 580us/sample - loss: 2.3013 - accuracy: 0.1124 - val_loss:
2.3010 - val_accuracy: 0.1135
Epoch 5/5
60000/60000 [=====] - 35s 582us/sample - loss: 2.3013 - accuracy: 0.1124 - val_loss:
2.3010 - val_accuracy: 0.1135



가중치의 초기화(Weight Initialization)

(3)SGD Optimizer, activation='relu', kernel_initializer='random_uniform'

```
model3 = Sequential([
    Conv2D(32, 3, activation='relu', kernel_initializer='random_uniform', padding='same',
           input_shape=(28, 28, 1)),
    MaxPooling2D(),
    Conv2D(filters=64, kernel_size=3, activation='relu', padding='same'),
    MaxPooling2D(),

    Flatten(),
    Dense(512, activation=tf.nn.relu),
    Dropout(0.2),
    Dense(10, activation=tf.nn.softmax)
])

model3.compile(optimizer='SGD',
               loss='sparse_categorical_crossentropy',
               metrics=['accuracy'])

history33 = model3.fit(train_images, train_labels, validation_data=(test_images, test_labels),
                       batch_size=batch_size, epochs=training_epochs)
```

Train on 60000 samples, validate on 10000 samples

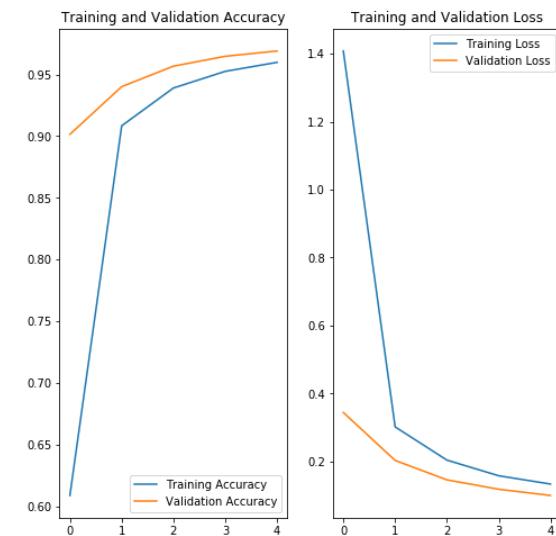
Epoch 1/5
60000/60000 [=====] - 35s 585us/sample - loss: 1.4077 - accuracy: 0.6087 - val_loss:
0.3446 - val_accuracy: 0.9015

Epoch 2/5
60000/60000 [=====] - 34s 566us/sample - loss: 0.3022 - accuracy: 0.9085 - val_loss:
0.2035 - val_accuracy: 0.9402

Epoch 3/5
60000/60000 [=====] - 34s 571us/sample - loss: 0.2045 - accuracy: 0.9391 - val_loss:
0.1464 - val_accuracy: 0.9568

Epoch 4/5
60000/60000 [=====] - 35s 585us/sample - loss: 0.1585 - accuracy: 0.9526 - val_loss:
0.1188 - val_accuracy: 0.9648

Epoch 5/5
60000/60000 [=====] - 35s 579us/sample - loss: 0.1341 - accuracy: 0.9599 - val_loss:
0.1005 - val_accuracy: 0.9691



가중치의 초기화(Weight Initialization)

(4)SGD Optimizer, activation='relu', kernel_initializer='he_uniform'

```

model4 = Sequential([
    Conv2D(32, 3, activation='relu', kernel_initializer='he_uniform', padding='same',
           input_shape=(28, 28, 1)),
    MaxPooling2D(),
    Conv2D(filters=64, kernel_size=3, activation='relu', padding='same'),
    MaxPooling2D(),

    Flatten(),
    Dense(512, activation=tf.nn.relu),
    Dropout(0.2),
    Dense(10, activation=tf.nn.softmax)
])

model4.compile(optimizer='SGD',
               loss='sparse_categorical_crossentropy',
               metrics=['accuracy'])

history44 = model4.fit(train_images, train_labels, validation_data=(test_images, test_labels),
                       batch_size=batch_size, epochs=training_epochs)

```

Train on 60000 samples, validate on 10000 samples

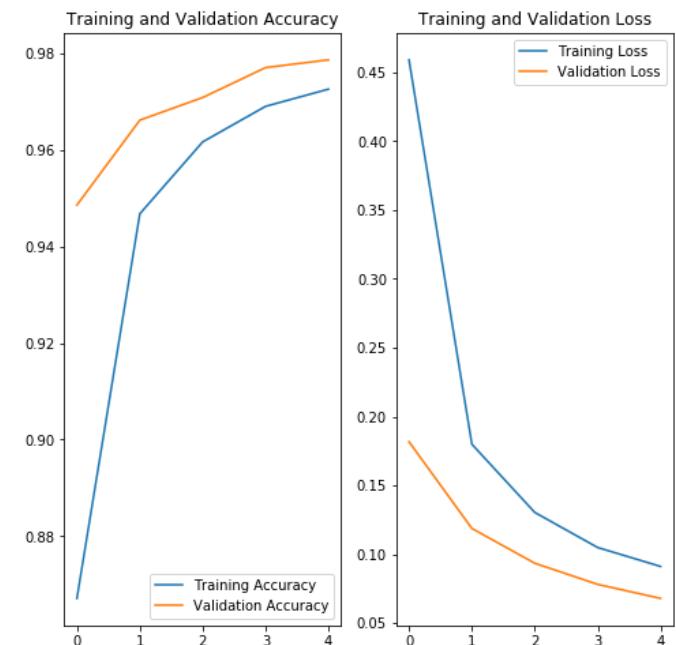
Epoch 1/5
60000/60000 [=====] - 36s 593us/sample - loss: 0.4591 - accuracy: 0.8671 - val_loss:
0.1817 - val_accuracy: 0.9486

Epoch 2/5
60000/60000 [=====] - 35s 591us/sample - loss: 0.1799 - accuracy: 0.9468 - val_loss:
0.1188 - val_accuracy: 0.9662

Epoch 3/5
60000/60000 [=====] - 35s 582us/sample - loss: 0.1304 - accuracy: 0.9617 - val_loss:
0.0935 - val_accuracy: 0.9709

Epoch 4/5
60000/60000 [=====] - 35s 580us/sample - loss: 0.1049 - accuracy: 0.9691 - val_loss:
0.0782 - val_accuracy: 0.9771

Epoch 5/5
60000/60000 [=====] - 34s 572us/sample - loss: 0.0912 - accuracy: 0.9726 - val_loss:
0.0680 - val_accuracy: 0.9787



가중치의 초기화(Weight Initialization)

(5) Adam Optimizer, activation='relu', kernel_initializer='he_uniform'

```
model5 = Sequential([
    Conv2D(32, 3, activation='relu', kernel_initializer='he_uniform', padding='same',
           input_shape=(28, 28, 1)),
    MaxPooling2D(),
    Conv2D(filters=64, kernel_size=3, activation='relu', padding='same'),
    MaxPooling2D(),

    Flatten(),
    Dense(512, activation=tf.nn.relu),
    Dropout(0.2),
    Dense(10, activation=tf.nn.softmax)
])

model5.compile(optimizer='Adam',
               loss='sparse_categorical_crossentropy',
               metrics=['accuracy'])

history55 = model5.fit(train_images, train_labels, validation_data=(test_images, test_labels),
                       batch_size=batch_size, epochs=training_epochs)
```

Train on 60000 samples, validate on 10000 samples

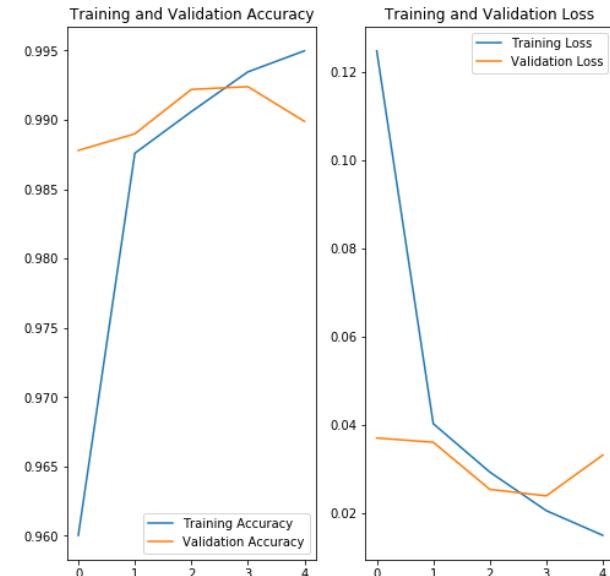
Epoch 1/5
60000/60000 [=====] - 37s 611us/sample - loss: 0.1247 - accuracy: 0.9600 - val_loss:
0.0370 - val_accuracy: 0.9878

Epoch 2/5
60000/60000 [=====] - 36s 596us/sample - loss: 0.0402 - accuracy: 0.9876 - val_loss:
0.0360 - val_accuracy: 0.9890

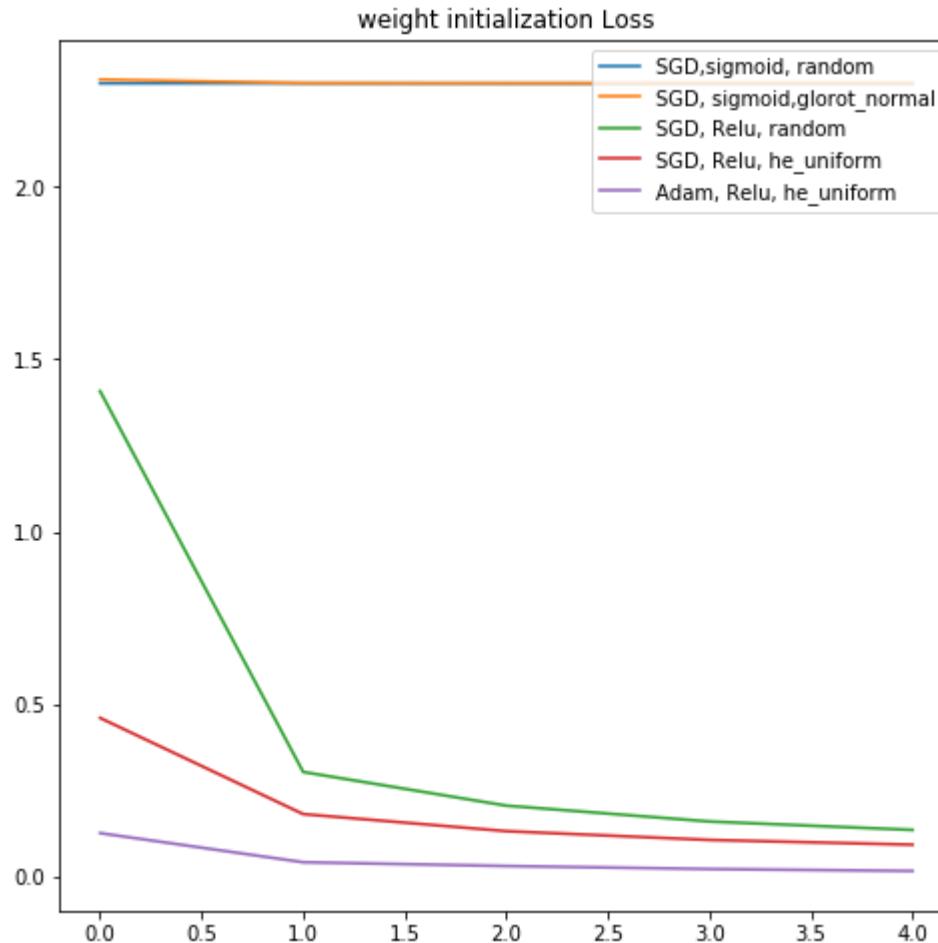
Epoch 3/5
60000/60000 [=====] - 35s 592us/sample - loss: 0.0292 - accuracy: 0.9906 - val_loss:
0.0253 - val_accuracy: 0.9922

Epoch 4/5
60000/60000 [=====] - 36s 593us/sample - loss: 0.0205 - accuracy: 0.9935 - val_loss:
0.0239 - val_accuracy: 0.9924

Epoch 5/5
60000/60000 [=====] - 36s 602us/sample - loss: 0.0149 - accuracy: 0.9950 - val_loss:
0.0331 - val_accuracy: 0.9989s: 0.0150 - accu

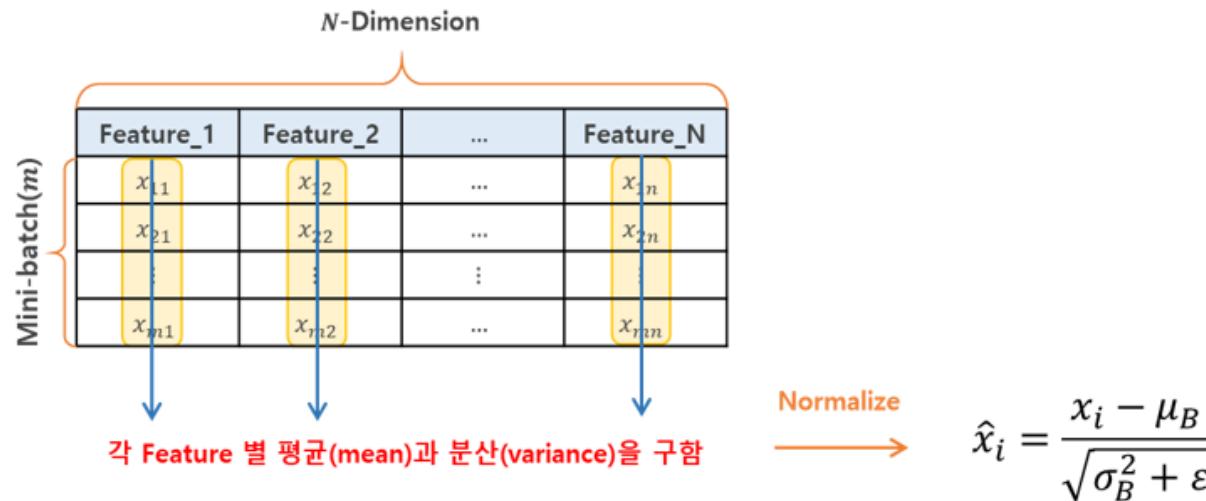


가중치의 초기화(Weight Initialization)



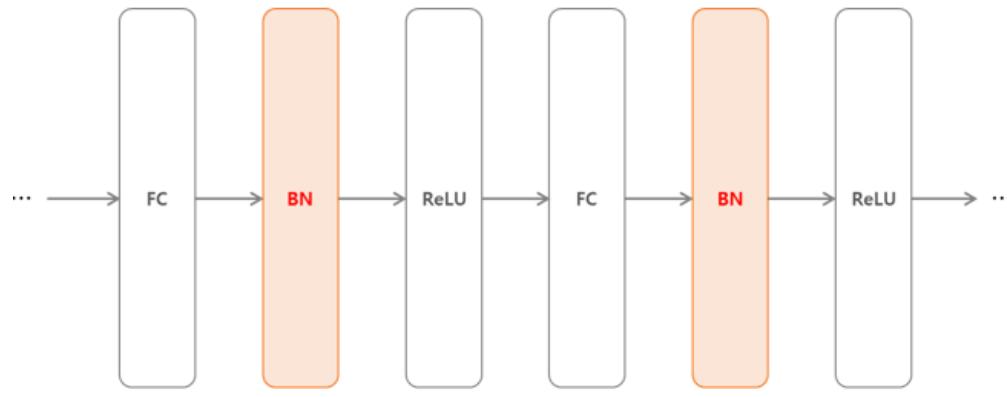
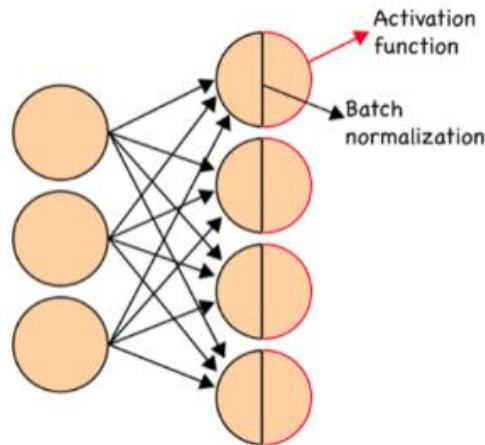
배치 정규화(BN, Batch Normalization)

- 강제적으로 각 층의 활성화 함수의 출력값 분포가 골고루 분포되도록 하는 방법
- 각 층에서의 활성화 함수 출력값이 정규분포(normal distribution)를 이루도록 하는 방법
- 학습하는 동안 이전 레이어에서의 가중치 매개변수가 변함에 따라 활성화 함수 출력값의 분포가 변화하는 **내부 공변량 변화(Internal Covariate Shift)** 문제를 줄이는 방법
- 미니배치(mini-batch)의 데이터에서 각 feature(특성)별 평균(mean)과 분산(variance)을 구한 뒤 정규화(normalize)



배치 정규화(BN, Batch Normalization)

- Fully Connected(FC)나 Convolutional layer 바로 다음, 활성화 함수를 통과하기 전에 배치 정규화(BN)레이어를 삽입하여 사용



배치 정규화(BN, Batch Normalization)

❖ 배치정규화의 장점

- tanh나 sigmoid 같은 활성화 함수에 대해 그래디언트 소실(vanishing gradient)문제가 감소.
- 가중치 초기화에 덜 민감
- 학습률(learning rate)를 크게 잡아도 gradient descent가 잘 수렴
- 오버피팅을 억제, BN이 마치 Regularization 역할을 하기 때문에 드롭아웃(Dropout)과 같은 규제기법에 대한 필요성이 감소
- BN로 인한 규제는 효과가 크지 않기 때문에 드롭아웃을 함께 사용하는 것이 좋다

배치 정규화(BN, Batch Normalization)

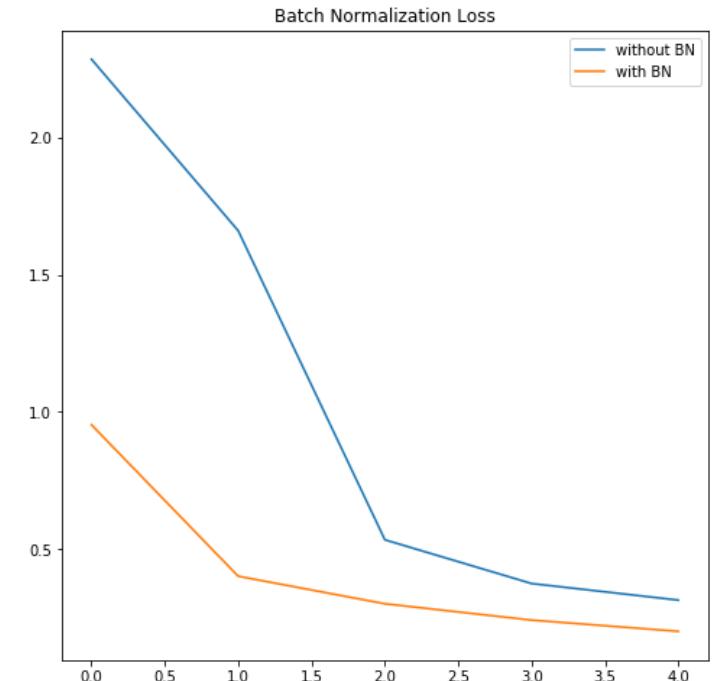
❖ 배치정규화 예

```
model2 = Sequential([
    Conv2D(32, 3, padding='same', input_shape=(28, 28, 1)),
    BatchNormalization(),
    Activation('sigmoid'),
    MaxPooling2D(),
    Conv2D(filters=64, kernel_size=3, padding='same'),
    BatchNormalization(),
    Activation('sigmoid'),
    MaxPooling2D(),

    Flatten(),
    Dense(512, activation=tf.nn.relu),
    Dropout(0.2),
    Dense(10, activation=tf.nn.softmax)
])

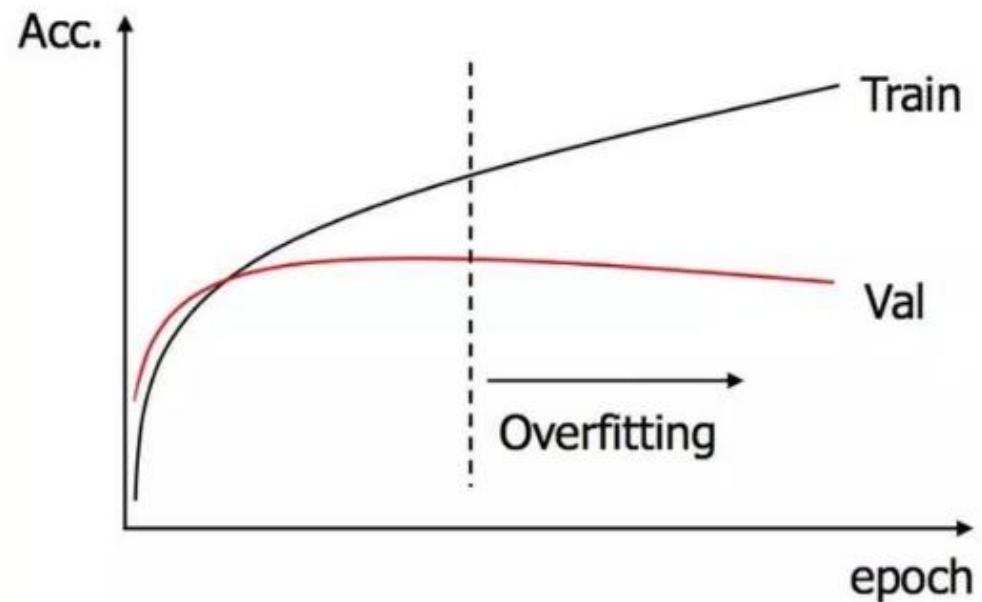
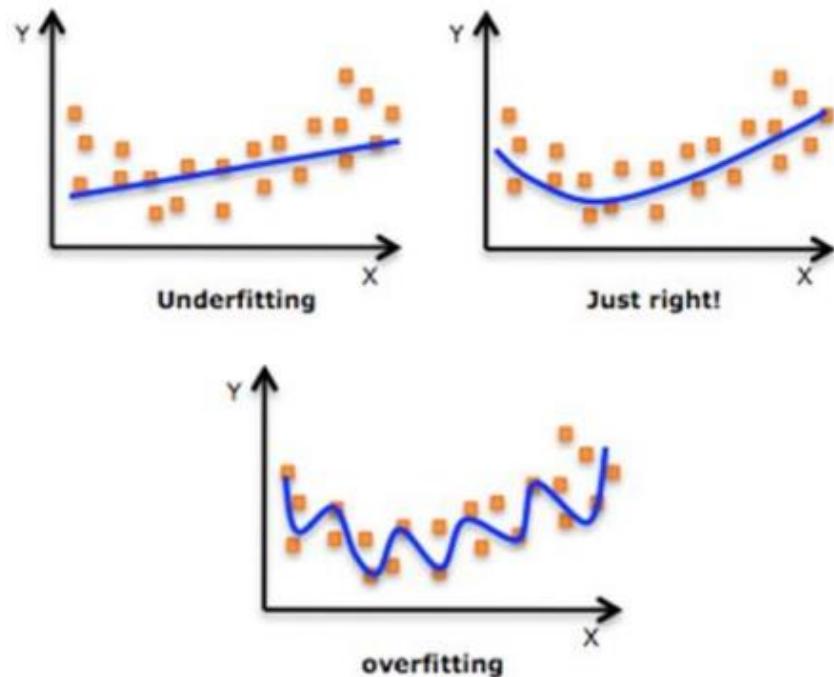
model2.compile(optimizer='SGD',
               loss='sparse_categorical_crossentropy',
               metrics=['accuracy'])

history2 = model2.fit(train_images, train_labels, validation_data=(test_images, test_labels),
                      batch_size=batch_size, epochs=training_epochs)
```



Overfitting

- ✓ 훈련 데이터(training set)에만 적합(fit)되어 그 외의 데이터(test set 등)에는 제대로 대응하지 못하는 상태
- ✓ 매개 변수가 많고 표현력이 높은 모델 → Complexity가 높은 모델
- ✓ 훈련 데이터가 적을 경우



Weight Regularization

❖ 가중치 감소 (Weight decay)

- ✓ 학습 과정에서 큰 가중치에 대해서는 큰 페널티를 부과하여 오버피팅을 억제하는 방법
- ✓ L1 보다 L2 가 더 많이 쓰임

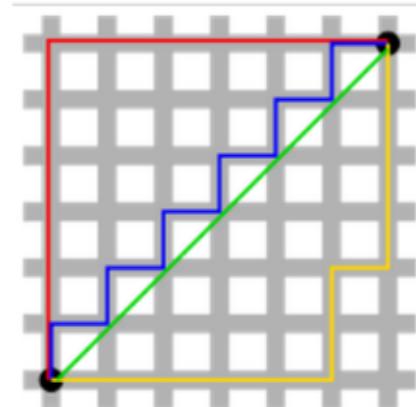
NOTE_ L2 법칙은 각 원소의 제곱들을 더한 것에 해당합니다. 가중치 $\mathbf{W} = (w_1, w_2, \dots, w_n)$ 이 있다면, L2 법칙에서는 $\sqrt{w_1^2 + w_2^2 + \dots + w_n^2}$ 으로 계산할 수 있습니다. L2 법칙 외에 L1 법칙과 L^∞ 법칙도 있습니다. L1 법칙은 절댓값의 합, 즉 $|w_1| + |w_2| + \dots + |w_n|$ 에 해당합니다. L^∞ 법칙은 Max 법칙이라고도 하며, 각 원소의 절댓값 중 가장 큰 것에 해당합니다. 정규화 항으로 L2 법칙, L1 법칙, L^∞ 법칙 중 어떤 것도 사용할 수 있습니다. 각자 특징이 있는데, 이 책에서는 일반적으로 자주 쓰는 L2 법칙만 구현합니다.

L1 regularization on least squares:

$$L = \arg \min_{\mathbf{w}} \sum_j \left(t(\mathbf{x}_j) - \sum_i w_i h_i(\mathbf{x}_j) \right)^2 + \lambda \sum_{i=1}^k |w_i|$$

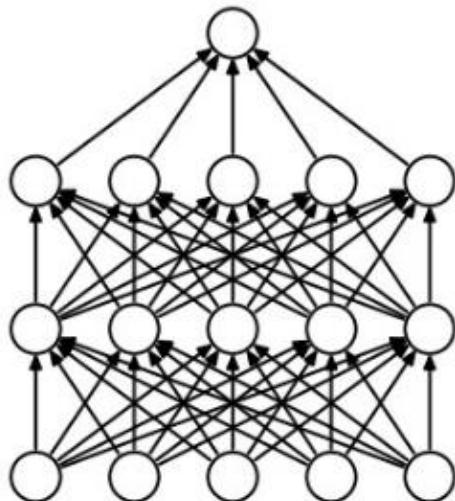
L2 regularization on least squares:

$$L = \arg \min_{\mathbf{w}} \sum_j \left(t(\mathbf{x}_j) - \sum_i w_i h_i(\mathbf{x}_j) \right)^2 + \lambda \sum_{i=1}^k w_i^2$$

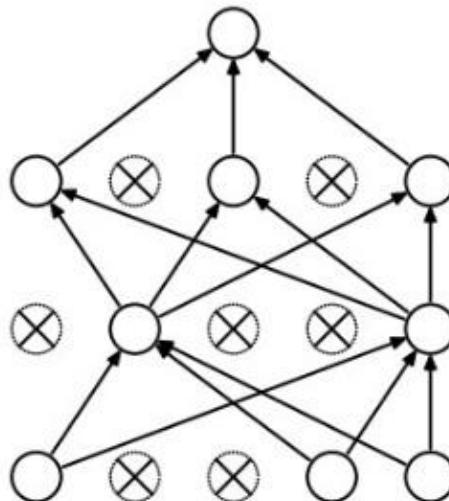


Drop Out

- 과적합을 막기 위해 학습 동안에만 레이어의 모든 노드에서 나가는 activation을 특정 확률로 지우는 기능
- 검증시에는 모든 activation을 다시 살림
- CNN에서는 드롭아웃 레이어를 Fully connected layer 뒤에 두지만 상황에 따라서는 max pooling 계층 뒤에 놓기도 함
- tf.keras.layers.Dropout(.2)
 - .2 : 제거 할 입력 단위의 비율(20%)



(a) Standard Neural Net



(b) After applying dropout.

```
model2 = Sequential([
    Conv2D(32, 3, padding='same', input_shape=(28, 28, 1)),
    BatchNormalization(),
    Activation('sigmoid'),
    MaxPooling2D(),
    Conv2D(filters=64, kernel_size=3, padding='same'),
    BatchNormalization(),
    Activation('sigmoid'),
    MaxPooling2D(),
    Flatten(),
    Dense(512, activation=tf.nn.relu),
    Dropout(0.2),
    Dense(10, activation=tf.nn.softmax)
])
```