

Lab 3: MLP

<삼성 AI 전문가 교육과정> 실습
서울대학교 바이오지능 연구실 (장병탁 교수)
최원석, 김윤성
2022.06.09

Biointelligence Laboratory
Dept. of Computer Science and Engineering
Seoul National University

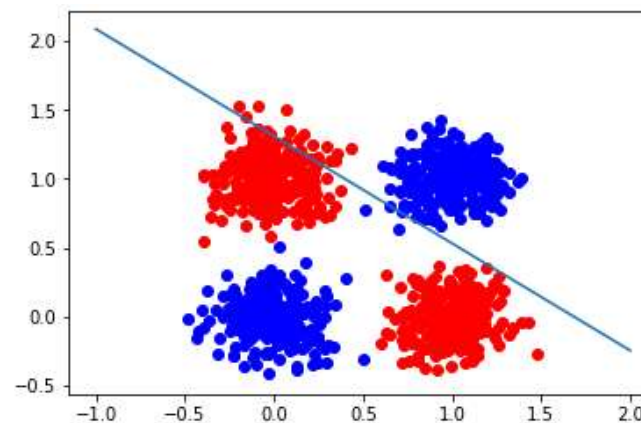
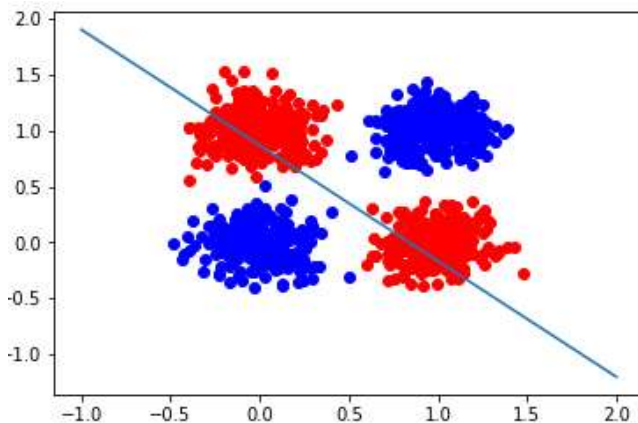


Multi-Layer Perceptron

Logic Gate Example (Prev.)

■ Single Perceptron Separator


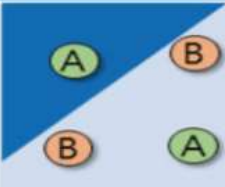
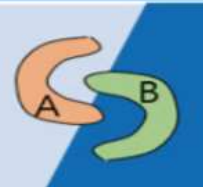

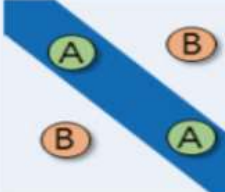




- 단일 퍼셉트론만으로는 XOR에 해당하는 함수를 근사할 수 없음
- $val = \text{Sigmoid}(ax + by + c) \rightarrow xy$ 평면상의 halfplane
- 파라미터(a, b, c)를 아무리 최적화해도 결과는 halfplane
- $val = \text{Sigmoid}(ax + by + c)$ 함수 구조 자체를 바꿔줘야 해결 가능



Multi Layer Perceptron

■ Multi-Layer Perceptron

- N-1번째 층의 출력을 입력으로 하는 N번째 층을 생성
- 표현할 수 있는 범위가 더 커짐

Structure	Regions	XOR	Meshed Regions
Single layer 	Halfplane bounded by hyperplane		
Two layers 	Convex Open or closed regions		
Three layers 	Arbitrary (limited by # of nodes)		

Multi Layer Perceptron

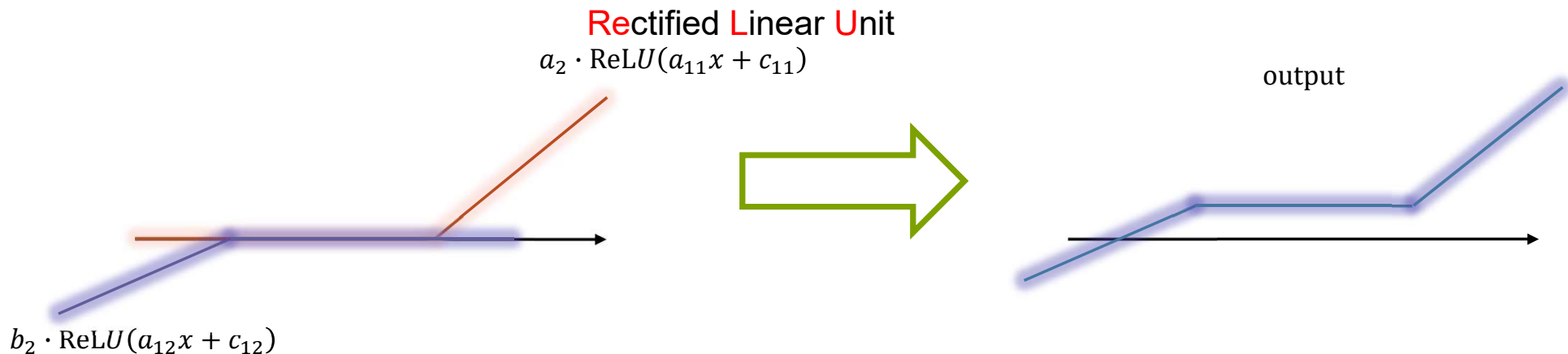
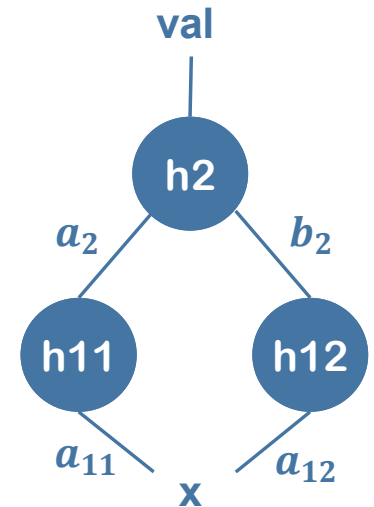
■ Representation power

■ Without (nonlinear) activation functions

$$\begin{aligned} & \blacksquare a_2(a_{11}x + c_{11}) + b_2(a_{12}x + c_{12}) + c_2 \\ & = (a_2a_{11} + b_2a_{12})x + (a_2c_{11} + b_2c_{12} + c_2) = a'x + c' \end{aligned}$$

■ With (nonlinear) activation functions

$$\blacksquare a_2 \cdot \text{ReLU}(a_{11}x + c_{11}) + b_2 \cdot \text{ReLU}(a_{12}x + c_{12}) + c_2 \neq a'x + c'$$



Single perceptron as affine transformation

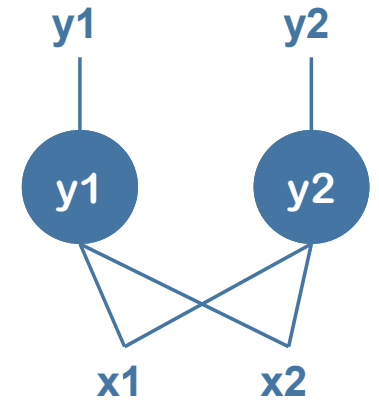
■ Affine transformation

- $\vec{y} = A\vec{x} + \vec{b}$

■ Single perceptron

- $y_1 = a_{11}x_1 + a_{12}x_2 + b_1$
- $y_2 = a_{21}x_1 + a_{22}x_2 + b_2$

⇒ $\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$



- Single perceptron을 통과하는 연산은 affine transformation과 동일
- 입력의 개수가 n , 출력의 개수가 m 인 임의의 single perceptron의 파라미터
 - 크기가 $m \times n$ 인 가중치 행렬
 - 길이가 m 인 bias vector

MLP 모델의 구성

■ `nn.Linear(in_features, out_features)`

- Weight matrix, bias vector를 포함한 단일 perceptron layer를 구현한 라이브러리
 - Input, output개수: `in_features`, `out_features`
- `nn.Module`을 inherit한 일종의 submodule
- Parameter를 가지므로 모델 초기화 시 선언 필요

■ `nn.functional.relu(input)`

- ReLU activation function

■ `torch.sigmoid(input)`

- 결과값 shaping을 위한 Sigmoid function
- 모델이 아닌 단순한 함수이므로 초기화 필요 없음

MLP implementation details

Model initialization

■ 모델 선언

```
class Separator(nn.Module):
    def __init__(self):
        super(Separator, self).__init__()
        self.fc1 = nn.Linear(2, 4)
        self.fc2 = nn.Linear(4, 1)

    def forward(self, x):
        x = self.fc1(x)
        x = nn.functional.relu(x)
        x = self.fc2(x)
        x = torch.sigmoid(x)
        return x
```

- `__init__(self)`
 - 2개의 fully connected(linear) layer를 선언
 - Weight initialization
 - 노드 개수: 2(x,y)→4(hidden) → 1(val_)
- `forward(self, x)`
 - `val_ = sigmoid(fc2(ReLU(fc1(x))))`

$$\begin{bmatrix} x \\ y \end{bmatrix} \xrightarrow{ReLU \cdot fc_1} ReLu \left(\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \right) = \begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix} \xrightarrow{\sigma \cdot fc_2} \sigma \left(\begin{bmatrix} w'_1 & w'_2 & w'_3 & w'_4 \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix} + b' \right) = val_-$$

Training phase details

■ 전체적인 순서

```
batch_size = 20
num_epochs = 10
num_workers = 4
optimizer = Adam(model.parameters(), lr=lr)
```

```
for epoch in range(num_epochs):
    total_loss = 0
    for x, val in dataloader:
        x = x.cuda()
        val = val.cuda()
        optimizer.zero_grad()
        val_ = model(x)
        loss = torch.sum(torch.pow(val - val_, 2))

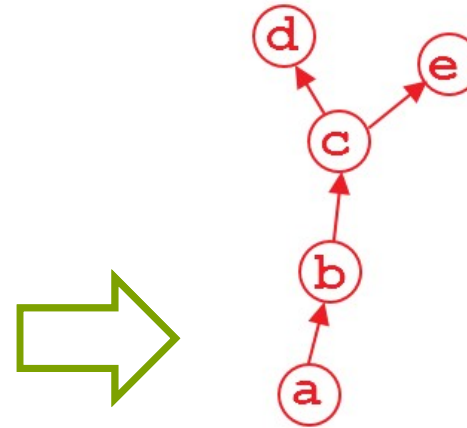
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print("Loss : {:.5f}".format(total_loss / len(DATASET)))
```

- Hyperparameter 선언
 - batch_size
 - num_epochs
 - learning_rate
- Optimizer 초기화
- 모델, 데이터셋 초기화
- for each epoch
 - optimizer.zero_grad()
 - **val_ = model(x)**
 - loss(val, val_)
 - loss.backward()
 - optimizer.step()

Graph construction

■ Graph construction

```
import torch
from torch.autograd import Variable
a = Variable(torch.rand(1, 4), requires_grad=True)
b = a**2
c = b*2
d = c.mean()
e = c.sum()
```



- 위와 같은 코드 실행 시, 오른쪽과 같은 그래프가 implicit하게 생성
 - `b.data` – 값 : 2.3
 - `b.grad_fn` – `b`의 미분에 이용될 함수 : `<MulBackward0>`
- `e.backward()` 실행 시 다음 인자가 계산됨
 - `b.grad` – 변수 e 을 b 로 미분한 미분값 : $\frac{\partial e}{\partial b}$

Training phase details

■ 전체적인 순서

```
batch_size = 20
num_epochs = 10
num_workers = 4
optimizer = Adam(model.parameters(), lr=lr)
```

```
for epoch in range(num_epochs):
    total_loss = 0
    for x, val in dataloader:
        x = x.cuda()
        val = val.cuda()
        optimizer.zero_grad()
        val_ = model(x)
        loss = torch.sum(torch.pow(val - val_, 2))
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print("Loss : {:.5f}".format(total_loss / len(DATASET)))
```

- Hyperparameter 선언
 - batch_size
 - num_epochs
 - learning_rate
- Optimizer 선언
- for each epoch
 - optimizer.zero_grad()
 - val_ = model(x)
 - loss(val, val_)
 - loss.backward()
 - optimizer.step()

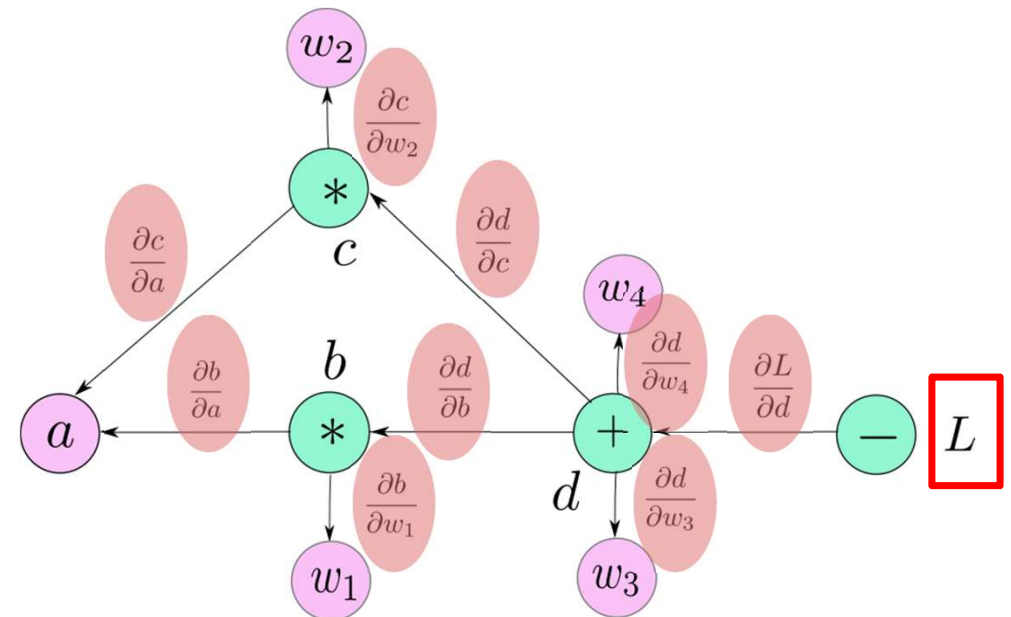
Loss function

■ Loss function

- 모델의 결과값과 실제 정답과의 차이
- 패널티
- L2 loss: $\|val - val_{-}\|_2^2$

■ `loss.backward()`

- 실행 시 loss와 연결된 모든 tensor의 `.grad` 값이 계산됨



Training phase details

■ 전체적인 순서

```
batch_size = 20
num_epochs = 10
num_workers = 4
optimizer = Adam(model.parameters(), lr=lr)
```

```
for epoch in range(num_epochs):
    total_loss = 0
    for x, val in dataloader:
        x = x.cuda()
        val = val.cuda()
        optimizer.zero_grad()
        val_ = model(x)
        loss = torch.sum(torch.pow(val - val_, 2))

        loss.backward()
        optimizer.step()
    total_loss += loss.item()
    print("Loss : {:.5f}".format(total_loss / len(DATASET)))
```

- Hyperparameter 선언
 - batch_size
 - num_epochs
 - learning_rate
- Optimizer 선언
- for each epoch
 - optimizer.zero_grad()
 - val_ = model(x)
 - loss(val, val_)
 - loss.backward()
 - optimizer.step()

Optimizer

■ Optimizer

- Variable의 .grad 값을 사용하여 Variable의 값을 변경(optimize)
 - learning_rate: 학습 가중치
 - params: 학습할 Variable 리스트
- Adam optimizer를 많이 사용
 - 이후 실습에서 다룰 예정

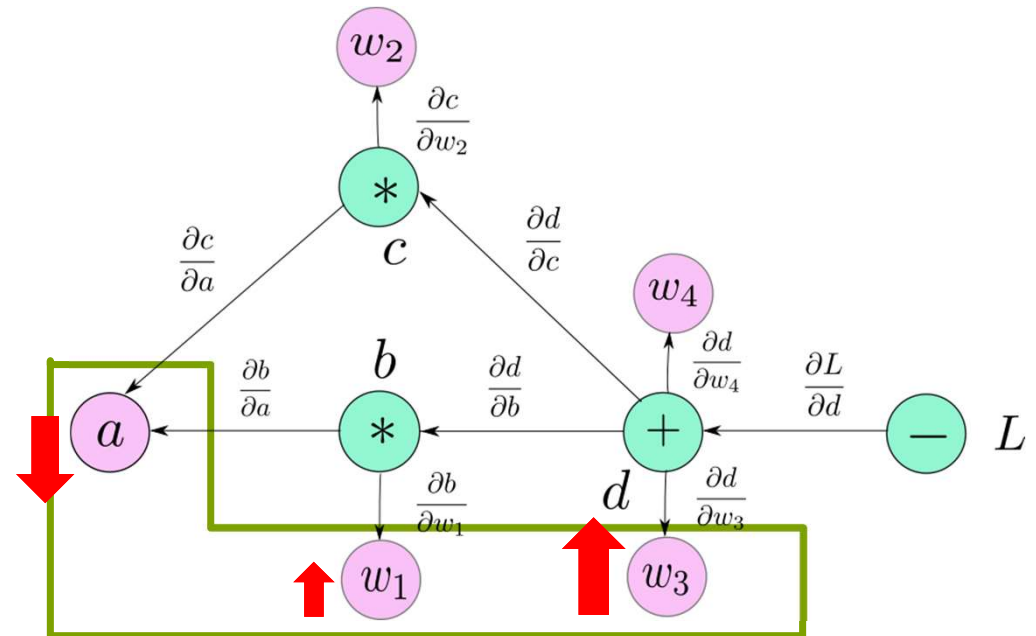
```
batch_size = 20
num_epochs = 10
num_workers = 4
optimizer = Adam(model.parameters(), lr=lr)
```

■ optimizer.zero_grad()

- 변수들의 .grad 값을 0으로 초기화

■ optimizer.step()

- Params에 포함된 변수들의 값 변경
 - 변경 방법은 optimizer 종류 따라 다름



Training phase details

■ 전체적인 순서

```
batch_size = 20
num_epochs = 10
num_workers = 4
optimizer = Adam(model.parameters(), lr=lr)
```

```
for epoch in range(num_epochs):
    total_loss = 0
    for x, val in dataloader:
        x = x.cuda()
        val = val.cuda()
        optimizer.zero_grad()
        val_ = model(x)
        loss = torch.sum(torch.pow(val - val_, 2))

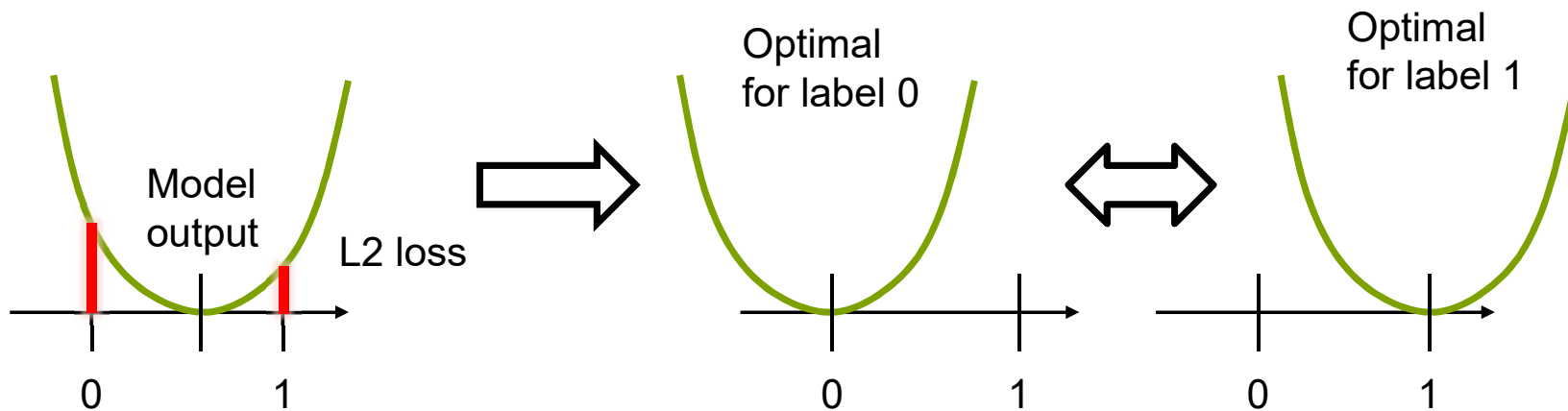
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print("Loss : {:.5f}".format(total_loss / len(DATASET)))
```

- Hyperparameter 선언
 - **batch_size**
 - **num_epochs**
 - **learning_rate**
- Optimizer 선언
- **for each epoch**
 - **optimizer.zero_grad()**
 - **val_ = model(x)**
 - **loss(val, val_)**
 - **loss.backward()**
 - **optimizer.step()**

Minibatch

■ Batch

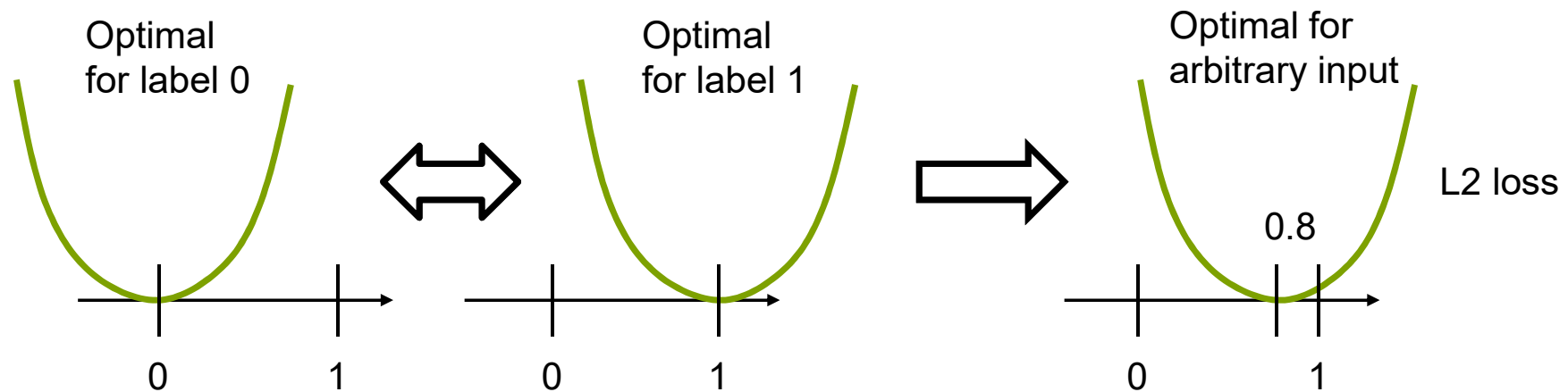
- Batch – 집단, 1회분
- 한번에 처리할 데이터 묶음
- 같은 입력 x 에 대해서, 80개의 샘플의 label이 1, 20개의 샘플 label이 0
- 임의의 x 를 하나씩 학습시킬 경우
 - 파라미터가 최적값이 0 또는 1이 되게끔 계속 진동함



Minibatch

■ Minibatch

- 전체 x 에 대한 loss를 모두 구한 뒤, 이를 평균내어 최종 loss를 구함
 - Optimal output인 0.8로 수렴
 - 학습하는데 너무 오랜 시간이 걸림
- 한번에 10개의 x 에 대한 loss를 구하고 평균내어 batch 단위의 loss를 구함
 - Minibatch



Minibatch

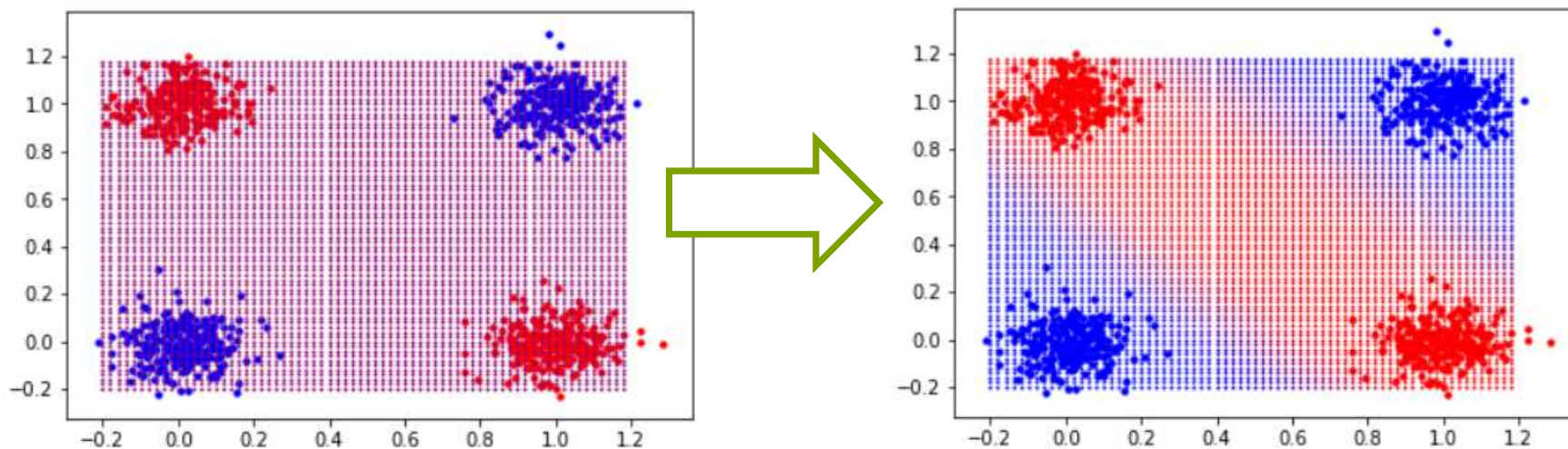
- Minibatch

- 크기가 커질수록
 - 안정적인 학습 가능
 - 학습 시간 대비 효율 감소
 - 많은 메모리 소모
- batch_size : 한번에 사용할 데이터 수
- iterations : batch_size 단위로 진행한 연산 수
 - $\text{Data size} = \text{batch_size} * \text{iterations}$
- epoch : 전체 training data 단위로 진행한 연산 수

■ Torch 구현 특징

- 모듈의 입력으로 들어오는 텐서의 **0번째 차원은 항상 batch size를 위해 할당**
- Batch size는 단순히 연산 횟수이므로, 입력값은 유동적으로 변할 수 있음
 - Ex) `fc = nn.Linear(30, 10)`, `b.shape: [5, 30]`, `c.shape: [100, 30]`
 - `fc(b).shape: [5, 10]` `fc(c).shape: [100, 30]`

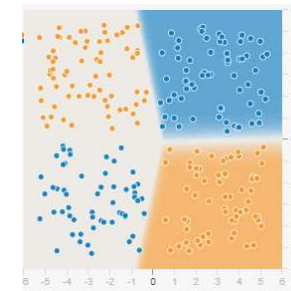
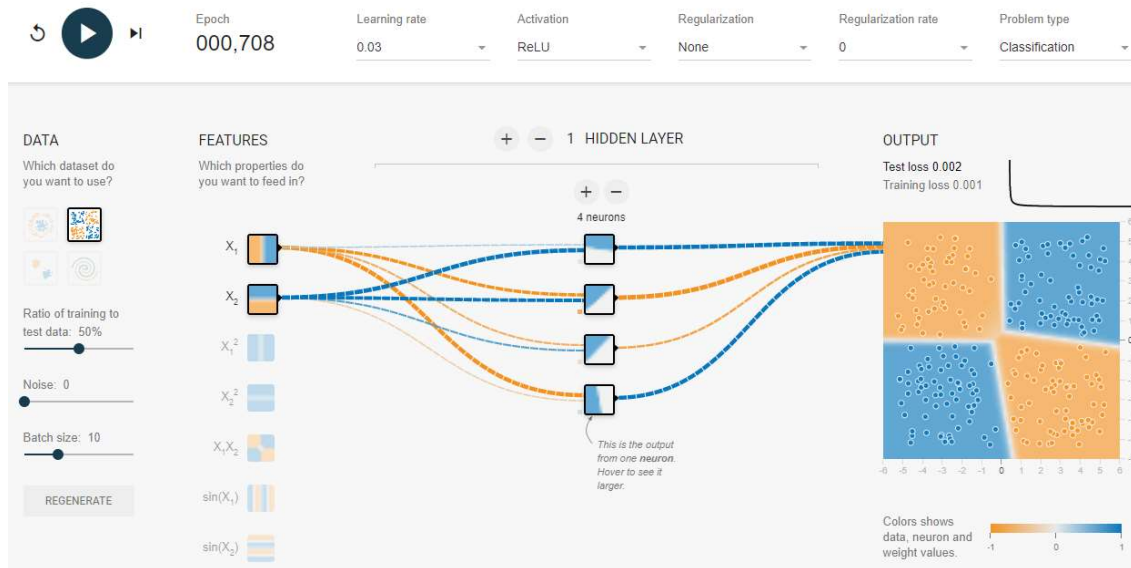
Results



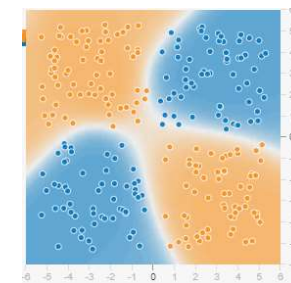
Tips: Neural network playground

■ TensorFlow neural network playground

- <https://playground.tensorflow.org>
- Model structure, activation function 등을 변경했을 때의 effect를 시각적으로 확인하고 이해하는데 도움이 됨



2 neurons



Sigmoid

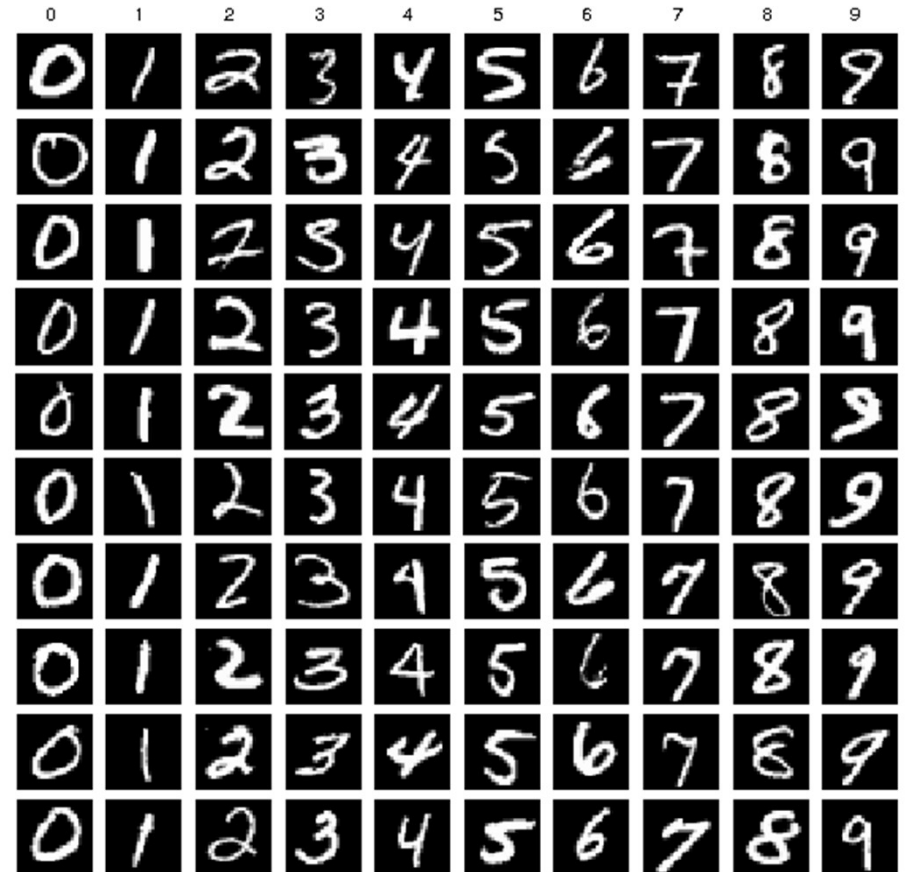
Image classification - MNIST

■ MNIST dataset

- 가장 유명한 입문용 데이터셋
- 0~9까지의 필기체 이미지(28 x 28)
 - 각 이미지에 맞는 숫자(label)

■ Torch mnist

- torchvision.datasets에 여러 유명한 기본 데이터셋이 미리 구현되어 있음
 - <https://pytorch.org/docs/stable/torchvision/datasets.html>



MNIST dataset

■ Loading dataset

- Import: *from torchvision.datasets import MNIST*
- `MNIST(root, train, transform, download)`
 - **root**: 데이터셋이 저장된(저장할) 루트 경로
 - **train**: True일 때 학습 데이터셋을, False일 때 테스트 데이터셋을 반환
 - **transform**: 원본 이미지를 변환시키는 전처리 과정
 - 여기서는 이미지를 tensor로만 바꿔주는 `ToTensor()` 함수만 사용
 - **download**: True일 때 root 경로에 데이터셋이 없으면 직접 다운로드함

```
transforms = Compose([
    ToTensor(),
])

trainset = MNIST('/content/gdrive/My Drive/MNIST_models/', train=True, transform=transforms, download=True)
testset = MNIST('/content/gdrive/My Drive/MNIST_models/', train=False, transform=transforms, download=True)
```

MNIST classifier implementation

■ Simple modification

```
class MNISTDNN(nn.Module):
    def __init__(self, IMG_SIZE=28):
        super(MNISTDNN, self).__init__()
        self.fc1 = nn.Linear(IMG_SIZE*IMG_SIZE, 32)
        self.BN1 = torch.nn.BatchNorm1d(32)
        self.fc2 = nn.Linear(32, 10)

    def forward(self, x):
        x = self.fc1(x)
        x = F.relu(x)
        x = self.BN1(x)
        x = self.fc2(x)
        x = torch.softmax(x, dim=-1)
        return x
```

```
if __name__ == '__main__':
    model = MNISTDNN(IMG_SIZE).cuda()

    model_parameters = filter(lambda p: p.requires_grad, model.parameters())
    num_params = sum([np.prod(p.size()) for p in model_parameters])
    print("number of parameters : {}".format(num_params))

    optimizer = Adam(model_parameters(), lr=LEARNING_RATE)
    loss_fn = nn.CrossEntropyLoss()

    for epoch in range(NUM_EPOCHES):
        tot_loss = 0.0

        for x, y in train_loader:
            optimizer.zero_grad()
            x = x.cuda().view(-1, IMG_SIZE*IMG_SIZE)
            y_ = model(x)
            loss = loss_fn(y_, y.cuda())
            loss.backward()
            tot_loss += loss.item()
            optimizer.step()

        print("Epoch {}, Loss(train) : {}".format(epoch+1, tot_loss))
        if epoch % 2 == 1:
            x, y = next(iter(test_loader))
            x = x.cuda().view(-1, IMG_SIZE*IMG_SIZE)
            y_ = model(x)
            _, argmax = torch.max(y_, dim=-1)
            test_acc = compute_acc(argmax, y.numpy())

            print("Acc(test) : {}".format(test_acc))

    torch.save(model.state_dict(), "/content/gdrive/My Drive/MNIST_models/DNN.pt")

    model_test = MNISTDNN(IMG_SIZE).cuda()
    model_test.load_state_dict(torch.load("/content/gdrive/My Drive/MNIST_models/DNN.pt"))
    model_test.eval()
    x, y = next(iter(test_loader))
    x = x.cuda().view(-1, IMG_SIZE*IMG_SIZE)
    y_ = model_test(x)
    _, argmax = torch.max(y_, dim=-1)
    test_acc = compute_acc(argmax, y.numpy())

    print("Acc(test) : {}".format(test_acc))
```


MNIST training

■ Simple modifications

- Batch normalization 추가
- Loss function: L2 loss \rightarrow cross entropy
- 전처리: 2d tensor인 x 를 1d tensor로 변환
- Model saving / loading
- Validation 과정

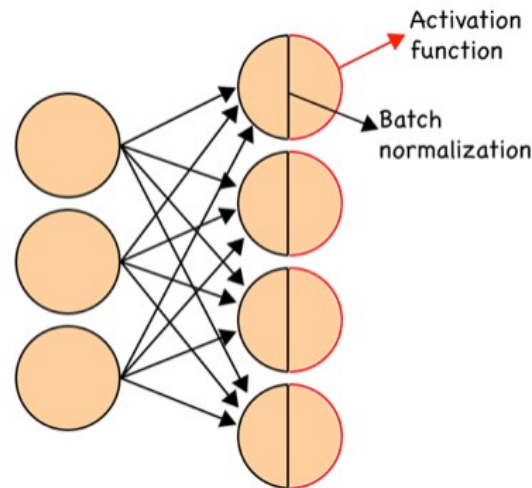
Batch normalization

■ Batch normalization

- 최근 가장 널리 쓰이는 regularization 기법

- <https://arxiv.org/abs/1502.03167>

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots x_m\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{ mini-batch mean}$$
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{ mini-batch variance}$$
$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{ normalize}$$
$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{ scale and shift}$$


$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$
$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i)$$

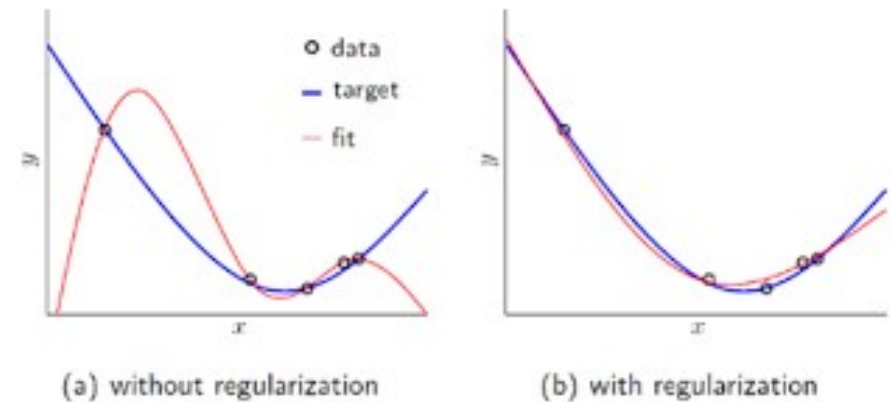
출처: <http://mohammadpz.github.io>

- Batch로 들어오는 입력을 정규분포 $N(\beta, \gamma)$ 로 표준화
- γ, β : learning parameter

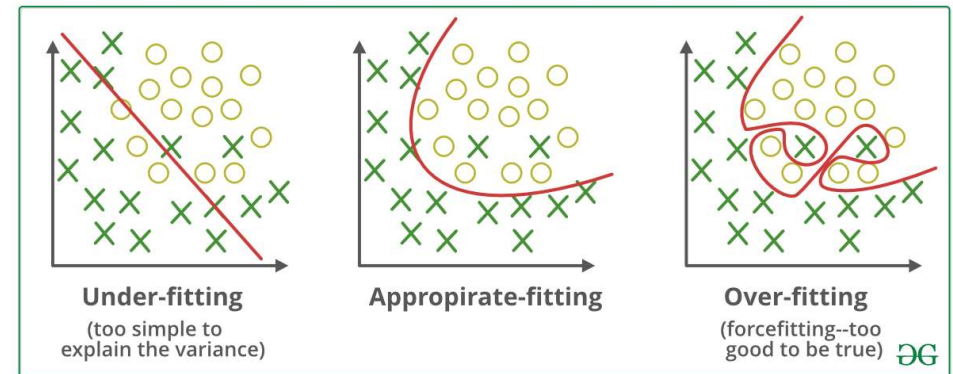
Batch normalization

■ Regularization effect

- 다항식의 예시에서, 과하게 요동치는 그래프는 다항식의 계수가 큰 값, 큰 분산을 가지는 경우에 발생
- Training data의 noise값까지 모두 학습
 - Noise가 바뀐 test data에서 성능이 낮아지게 됨
- Regularizer: coefficient의 값의 범위를 제한시켜 overfitting 방지



출처: <https://m.blog.naver.com/laonple/220527647084>



출처: <https://medium.com/analytics-vidhya/regularization-in-machine-learning-and-deep-learning-f5fa06a3e58a>

Model saving & loading

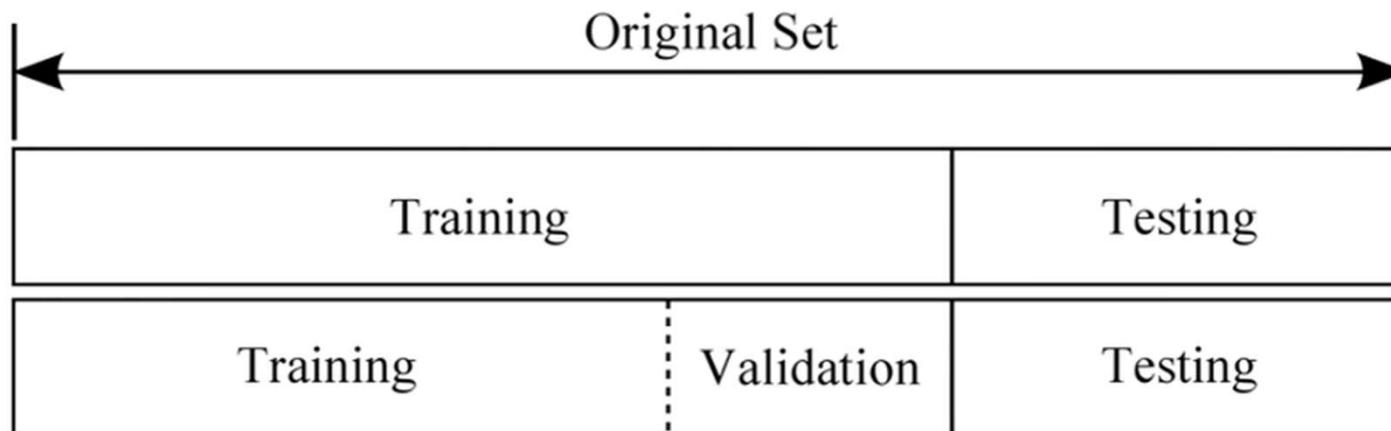
■ 2가지 저장 방법

- 모델 전체(serialized object)를 디스크에 저장
 - Save: torch.save(*model*, *dir*)
 - Load: torch.load(*dir*)
- 학습된 파라미터만을 디스크에 저장 (recommended)
 - 대상: model.state_dict()
 - Save: torch.save(*model.state_dict()*, *dir*)
 - Load: model.load_state_dict(*torch.load(dir)*)
 - 로딩 대상으로 지정할 model과, 로딩할 state_dict의 구조는 동일해야 함

Validation set

■ Validation 과정

- 보통 학습 기간을 지나치게 오래 두면 overfitting 등의 문제가 발생
- Test set은 모델의 최종 평가만을 위한 데이터셋, 학습 과정에 관여해선 안됨
- 학습하지 않은 데이터에 대한 성능평가가 필요하므로, training set 역시 X
- 별개의 데이터셋인 validation set을 둬
 - Validation set의 성능이 떨어지면 overfitting → 학습 중단



Results

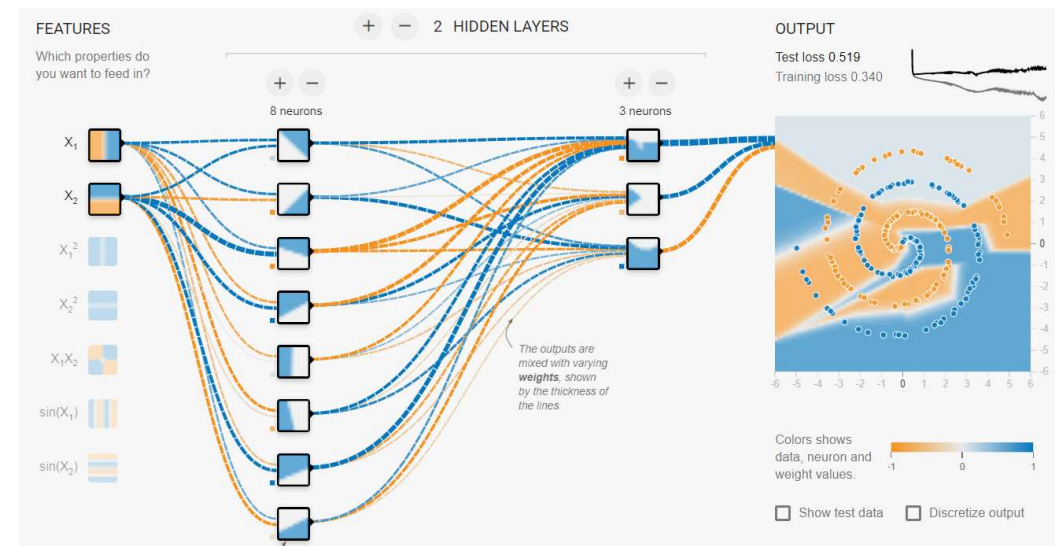
■ 학습 결과

- 약 97% 정도의 accuracy
- 학습 파라미터: **25514개**
 - 약 100MB

```
Epoch 25, Loss(train) : 347.5135814118800  
Epoch 26, Loss(train) : 347.42033445835114  
Acc(test) : 0.96484375  
Epoch 27, Loss(train) : 347.1093920469284  
Epoch 28, Loss(train) : 347.1259067058563  
Acc(test) : 0.9609375  
Epoch 29, Loss(train) : 347.12869024276733  
Epoch 30, Loss(train) : 347.00972402095795  
Acc(test) : 0.97265625  
Acc(test) : 0.97265625
```

■ 2층 MLP의 표현력

- 더 복잡한 데이터셋에 대해서
 - 그냥 쓰기엔 표현력이 부족
 - 층을 더 쌓으면 메모리가 커짐



Codes

- 실습 코드 Github 주소

- https://github.com/yskim5892/AI_Expert_2022