Portfolio

이름 : 김정섭
Email : kjscop@naver.com
Github : https://github.com/jskim0406

[Github_Link]

Table of contents

0. Study Reference

(0) Study reference

1. Math

1) Mathematics for machine learning(Cambridge University Press.) (https://mml-book.github.io)
2) 데이터사이언스 스쿨(김도형) (https://datascienceschool.net/view-notebook/04358acdcf3347fc989c4cfc0ef6121c/)
3) Calculus(James Stewart) (https://www.amazon.com/Calculus-James-Stewart/dp/1285740629)
4) Linear Algebra(Gilbert Strang) (http://faculty.marshall.usc.edu/gareth-james/ISL/)
5) Linear Algebra(Khan Academy Open Course) (https://ko.khanacademy.org/math/linear-algebra)
6) “The Matrix Calculus You Need for Deep Learning” (2018, Terence Parr, Jeremy Howard)

2. Statistics

1) Mathematics for machine learning(Cambridge University Press.) (https://mml-book.github.io)
2) 데이터사이언스 스쿨(김도형) (https://datascienceschool.net/view-notebook/04358acdcf3347fc989c4cfc0ef6121c/)
3) Introduction to Statistical Learning with R(Gareth James) (http://faculty.marshall.usc.edu/gareth-james/ISL/)
4) 수리통계학(전명식, 송선주) (https://kyobobook.co.kr/product/detailViewKor.laf?mallGb=KOR&ejkGb=KOR&barcode=9791158080129)

3. Machine Learning

1) 데이터사이언스 스쿨(김도형) (https://datascienceschool.net/view-notebook/04358acdcf3347fc989c4cfc0ef6121c/)
2) Deep Learning from Scratch: Building with Pytorch from First Principles(Seth Weidman) (https://www.amazon.com/Deep-Learning-Scratch-Building-Principles/dp/1492041416)
3) Standalone Deep Learning(Idea Factory KAIST) (https://www.youtube.com/playlist?list=PLSAJwo7mw8jn8iaXwT4MqLbZnS-LJwnBd)
4) “Fast-Campus” online course

4. Paper

1) Neural Ordinary Differential Equation (NeurIPS 2018, Ricky T. Q. Chen)
2) Augmented Neural ODEs (NeurIPS 2019, Emilien Dupont)
3) Deep Residual Learning for Image Recognition (CVPR 2015, Kaiming He)
4) U-Net : Convolutional Network for Biomedical Image Segmentation (CVPR 2015, Olaf Ronneberger)
5) Learning Deep Features for Discriminative Localization(CVPR 2016, Bolei Zhou)
6) Grad CAM : Visual Explanation from Deep Networks via Gradient based Localization (ICCV 2017, RR Selvaraju)

1. Linear Algebra

(1) Linear combination (Image Morphing)

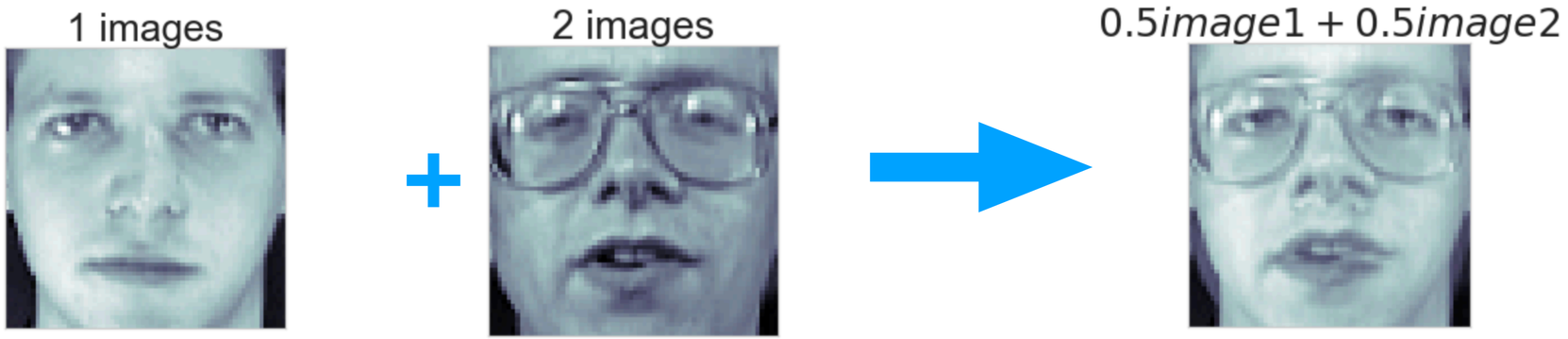
(2) Cosine similarity

(3) SVD (Singular Value Decomposition)

(4) PCA (Principal Component Analysis)

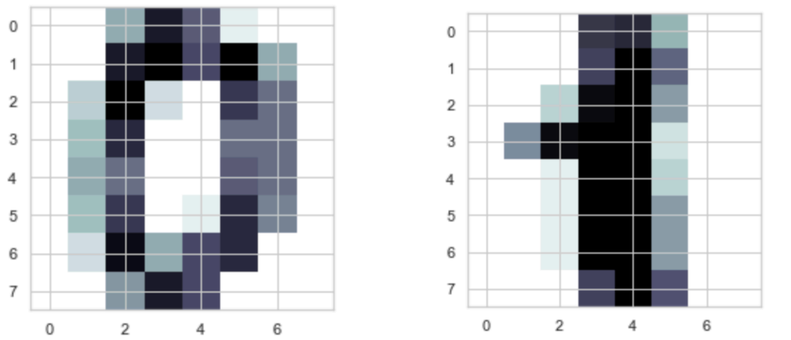
(1) Linear combination (Image Morphing)

Image data의 가중평균(선형결합)으로 새로운 이미지로 변형(Morphing)



(2) Cosine similarity

‘0’과 ‘8’의 코사인 유사도가 더 큼을 확인

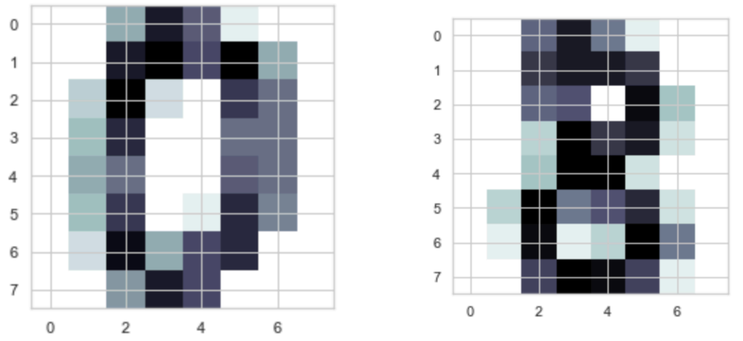


```
((_0.T@_1)/(np.linalg.norm(_0)*np.linalg.norm(_1)))[0][0]
```

```
0.5191023426414686
```

Cosine similarity

<



```
((_0.T@_8)/(np.linalg.norm(_0)*np.linalg.norm(_8)))[0][0]
```

```
0.7515122122359871
```

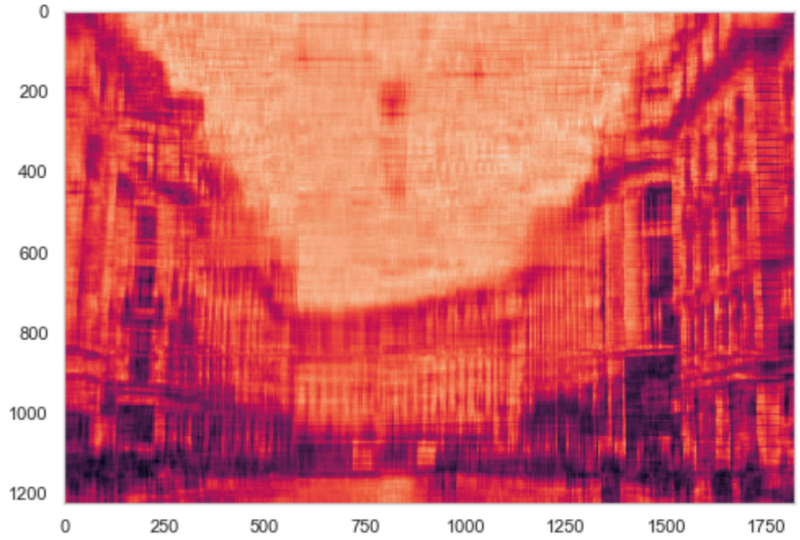
[Study_Link] [Code_Link]

(3) SVD (Singular Value Decomposition)

이미지 데이터를 Rank-1 Matrix로 분해(SVD)후 선형결합해 근사

$$A_{image} = U\Sigma V^T \approx \sum_{i=1}^{18} u_i \sigma_i v_i^T$$

```
plt.imshow(np.real(S[0]*U[:,:1]@VT[:1,:])
           +np.real(S[1]*U[:,1:2]@VT[1:2,:])
           +np.real(S[2]*U[:,2:3]@VT[2:3,:])
           +np.real(S[3]*U[:,3:4]@VT[3:4,:])
           +np.real(S[4]*U[:,4:5]@VT[4:5,:])
           +np.real(S[5]*U[:,5:6]@VT[5:6,:])
           +np.real(S[6]*U[:,6:7]@VT[6:7,:])
           +np.real(S[7]*U[:,7:8]@VT[7:8,:])
           +np.real(S[8]*U[:,8:9]@VT[8:9,:])
           +np.real(S[9]*U[:,9:10]@VT[9:10,:])
           +np.real(S[10]*U[:,10:11]@VT[10:11,:])
           +np.real(S[11]*U[:,11:12]@VT[11:12,:])
           +np.real(S[12]*U[:,12:13]@VT[12:13,:])
           +np.real(S[13]*U[:,13:14]@VT[13:14,:])
           +np.real(S[14]*U[:,14:15]@VT[14:15,:])
           +np.real(S[15]*U[:,15:16]@VT[15:16,:])
           +np.real(S[16]*U[:,16:17]@VT[16:17,:])
           +np.real(S[17]*U[:,17:18]@VT[17:18,:]))
plt.show()
```



[Study_Link] [Code_Link]

(4) PCA (Principal Component Analysis)

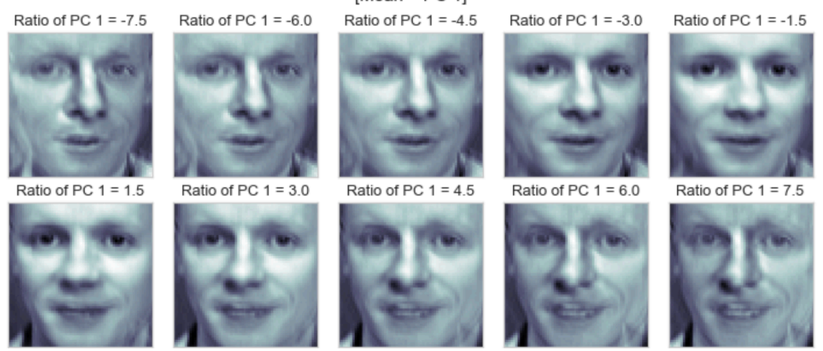
주성분 분석을 통해 주성분의 역할 확인

Original image data

PCA

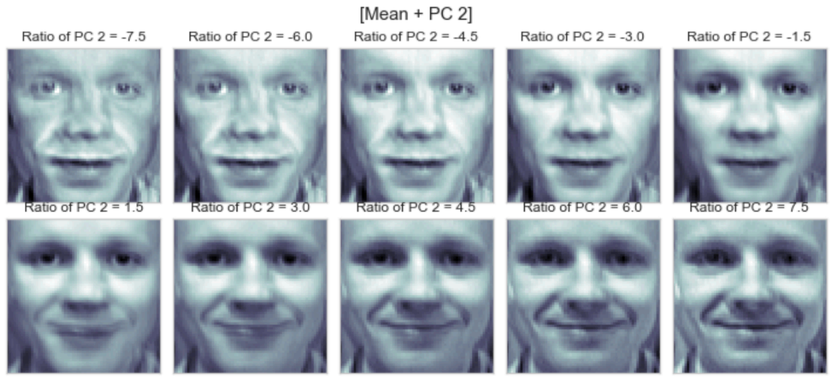
PCA

[평균값 + 주성분 1]



주성분 1의 역할 : 얼굴의 좌우 회전

[평균값 + 주성분 2]



주성분 2의 역할 : 얼굴의 표정 정보

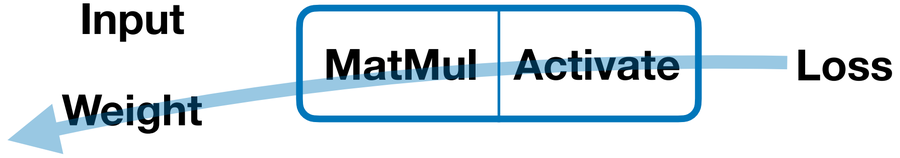
[Study_Link] [Code_Link]

2. Matrix Derivative

(5) Back Propagation

a. 1 Hidden_layer
b. 2 Hidden_layer
c. with Pytorch

(5) Back Propagation (1 Hidden Layer)



Single Layer 의 가장 단순한 Network의 Backpropagation 과정 이해

1 Hidden Layer Backpropagation

$X \in \mathbb{R}^{3\times 3}$, $W \in \mathbb{R}^{3\times 2}$

$N = XW \in \mathbb{R}^{3\times 2}$ $\xrightarrow{\frac{\partial N}{\partial X}}$ $S(XW) \in \mathbb{R}^{3\times 2}$ $\xrightarrow{\frac{\partial S}{\partial N}}$ $L = \Lambda(S(XW))$

$$\frac{\partial L}{\partial X} = \frac{\partial \Lambda}{\partial S} \cdot \frac{\partial S}{\partial N} \cdot \frac{\partial N}{\partial X} = \frac{\partial \Lambda}{\partial S} \cdot \frac{\partial \sigma(XW)}{\partial X}$$

① $\frac{\partial \Lambda}{\partial S_{ij}} = \begin{bmatrix} \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \end{bmatrix}$ $(\because \Lambda = S_{11}+S_{12}+S_{21}+S_{22}+S_{31}+S_{32})$ $*S_{11} = \sigma(XW_{11})$

$= \begin{bmatrix} \frac{\partial \Lambda}{\partial S_{11}} & \frac{\partial \Lambda}{\partial S_{12}} \\ \vdots & \end{bmatrix}$

② $\frac{\partial S}{\partial N} = \frac{\partial \sigma(XW)}{\partial (XW)} = \begin{bmatrix} \frac{\partial S_{11}}{\partial N} & \frac{\partial S_{12}}{\partial N} \\ \vdots & \end{bmatrix}$

③ $\frac{\partial N}{\partial X} = \frac{\partial (XW)}{\partial X} = \begin{bmatrix} \frac{\partial XW_{11}}{\partial X} & \frac{\partial XW_{12}}{\partial X} \\ \vdots & \end{bmatrix}$

1) $X_{11} \longrightarrow XW_{11} \longrightarrow \sigma(XW_{11})$

$\Rightarrow \frac{\partial \sigma(XW_{11})}{\partial X_{11}} = \frac{\partial \sigma(XW_{11})}{\partial XW_{11}} \frac{\partial XW_{11}}{\partial X_{11}} = \frac{\partial \sigma(XW_{11})}{\partial Z_{11}} \cdot W_{11}$

$* XW = \begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix} \begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \\ W_{31} & W_{32} \end{bmatrix} = \begin{bmatrix} X_{11}W_{11}+X_{12}W_{21}+X_{13}W_{31} & X_{11}W_{12}+X_{12}W_{22}+X_{13}W_{32} \\ X_{21}W_{11}+X_{22}W_{21}+X_{23}W_{31} & X_{21}W_{12}+X_{22}W_{22}+X_{23}W_{32} \\ X_{31}W_{11}+\cdots & X_{31}W_{12}+\cdots \end{bmatrix}$

$\sigma(XW) = \begin{bmatrix} \sigma(X_{11}W_{11}+X_{12}W_{21}+X_{13}W_{31}) & \sigma(X_{11}W_{12}+X_{12}W_{22}+X_{13}W_{32}) \\ \sigma(X_{21}W_{11}+X_{22}W_{21}+X_{23}W_{31}) & \sigma(X_{21}W_{12}+X_{22}W_{22}+X_{23}W_{32}) \\ \sigma(X_{31}W_{11}+\cdots) & \sigma(X_{31}W_{12}+\cdots) \end{bmatrix}$

2) $\frac{\partial \sigma(XW_{11})}{\partial X} = \begin{bmatrix} \frac{\partial \sigma(XW_{11})}{\partial X_{11}} & \frac{\partial \sigma(XW_{11})}{\partial X_{12}} & \frac{\partial \sigma(XW_{11})}{\partial X_{13}} \\ \frac{\partial \sigma(XW_{11})}{\partial X_{21}} & \frac{\partial \sigma(XW_{11})}{\partial X_{22}} & \frac{\partial \sigma(XW_{11})}{\partial X_{23}} \\ \frac{\partial \sigma(XW_{11})}{\partial X_{31}} & \frac{\partial \sigma(XW_{11})}{\partial X_{32}} & \frac{\partial \sigma(XW_{11})}{\partial X_{33}} \end{bmatrix} = \begin{bmatrix} \frac{\partial \sigma(XW_{11})}{\partial u} W_{11} & \frac{\partial \sigma(XW_{11})}{\partial u} W_{21} & \frac{\partial \sigma(XW_{11})}{\partial u} W_{31} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$

$\frac{\partial \sigma(XW_{12})}{\partial X} = \begin{bmatrix} \frac{\partial \sigma(XW_{12})}{\partial u} W_{12} & \frac{\partial \sigma(XW_{12})}{\partial u} W_{22} & \frac{\partial \sigma(XW_{12})}{\partial u} W_{32} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$

$\frac{\partial \sigma(XW_{21})}{\partial X} = \begin{bmatrix} 0 & 0 & 0 \\ \frac{\partial \sigma(XW_{21})}{\partial u} W_{11} & \frac{\partial \sigma(XW_{21})}{\partial u} W_{21} & \frac{\partial \sigma(XW_{21})}{\partial u} W_{31} \\ 0 & 0 & 0 \end{bmatrix}$

$\frac{\partial \sigma(XW_{22})}{\partial X} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \frac{\partial \sigma(XW_{22})}{\partial u} W_{12} & \frac{\partial \sigma(XW_{22})}{\partial u} W_{22} & \frac{\partial \sigma(XW_{22})}{\partial u} W_{32} \end{bmatrix}$

3) $\frac{\partial \sigma(XW)}{\partial X} = \frac{\partial S}{\partial N} \frac{\partial N}{\partial X} = \begin{bmatrix} \frac{\partial \sigma(XW_{11})}{\partial X} & \frac{\partial \sigma(XW_{12})}{\partial X} \\ \frac{\partial \sigma(XW_{11})}{\partial X} & \ddots \end{bmatrix}$

4) $\frac{\partial \Lambda}{\partial X} \Rightarrow \sum_{i=1}^{3}\sum_{j=1}^{2} \frac{\partial \sigma(XW_{ij})}{\partial X}$ $=$ $\begin{bmatrix} \frac{\partial \Lambda}{\partial X_{11}} & \frac{\partial \Lambda}{\partial X_{12}} & \frac{\partial \Lambda}{\partial X_{13}} \\ \frac{\partial \Lambda}{\partial X_{21}} & \frac{\partial \Lambda}{\partial X_{22}} & \frac{\partial \Lambda}{\partial X_{23}} \\ \frac{\partial \Lambda}{\partial X_{31}} & \frac{\partial \Lambda}{\partial X_{32}} & \frac{\partial \Lambda}{\partial X_{33}} \end{bmatrix}$

$* \Lambda = \in 6$

$\frac{\partial \Lambda}{\partial X} = \equiv \frac{\partial 6}{\partial X}$

$* \frac{\partial \Lambda}{\partial S} = \begin{bmatrix} \frac{\partial \Lambda}{\partial S_{11}} & \frac{\partial \Lambda}{\partial S_{12}} \\ \vdots & \end{bmatrix}$

$= \frac{\partial \Lambda}{\partial S} \frac{\partial S}{\partial X} = \frac{\partial \Lambda}{\partial S} W^T = \frac{\partial \Lambda}{\partial X}$

$= \begin{bmatrix} \frac{\partial \sigma(XW_{11})}{\partial u} & \frac{\partial \sigma(XW_{12})}{\partial u} \\ \frac{\partial \sigma(XW_{21})}{\partial u} & \frac{\partial \sigma(XW_{22})}{\partial u} \\ \vdots & \end{bmatrix} \begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \\ W_{31} & W_{32} \end{bmatrix}^T$

$= \frac{\partial \Lambda}{\partial S} \odot \frac{\partial S}{\partial N} * W^T$

$\Longrightarrow$ $\frac{\partial \Lambda}{\partial X} = \frac{\partial \Lambda}{\partial S} \odot \frac{\partial S}{\partial N} * W^T$ $(= \frac{\partial N}{\partial X})$

$* \frac{\partial S}{\partial N} = \frac{\partial \sigma(XW)}{\partial XW} = \begin{bmatrix} \frac{\partial \sigma(XW_{11})}{\partial XW_{11}} & \frac{\partial \sigma(XW_{12})}{\partial XW_{12}} \\ \vdots & \end{bmatrix}$

$X_{matrix} \xrightarrow{\frac{\partial (XW)}{\partial X}}$, $W_{matrix} \xrightarrow{\frac{\partial (XW)}{\partial W}}$ $N = XW \xrightarrow{\frac{\partial \sigma(XW)}{\partial XW}} \sigma(XW) \longrightarrow \Lambda(\sigma(XW))$

$\frac{\partial \sigma(XW)}{\partial X} = \frac{\partial \sigma(XW)}{\partial XW} \cdot \frac{\partial XW}{\partial X} = \frac{\partial \sigma(XW)}{\partial XW} \cdot W^T$ $\frac{\partial XW}{\partial X}$

$\frac{\partial \Lambda}{\partial X} = \frac{\partial \Lambda}{\partial \sigma(XW)} \cdot \frac{\partial \sigma(XW)}{\partial X}$

$\frac{\partial \sigma(XW)}{\partial W} = \frac{\partial \sigma(XW)}{\partial XW} \cdot \frac{\partial XW}{\partial W} = \frac{\partial \sigma(XW)}{\partial XW} \cdot X^T$

$* X = [x_1, x_2, x_3]$

$W = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$

$X \xrightarrow{W} W^T x \longrightarrow \sigma(W^T x)$

$\frac{\partial \sigma(W^T x)}{\partial x} = \frac{\partial \sigma(W^T x)}{\partial W^T x} \cdot \frac{\partial W^T x}{\partial x} = \frac{\partial \sigma(W^T x)}{\partial W^T x} \cdot (W^T)^T$

matrix Back Prop 생각 순서!

$\frac{\partial XW_{11}}{\partial X_{11}} \longrightarrow \frac{\partial \sigma(XW_{11})}{\partial X_{11}} \longrightarrow \frac{\partial \sigma(XW_{11})}{\partial X} \longrightarrow \frac{\partial \Lambda}{\partial X}$

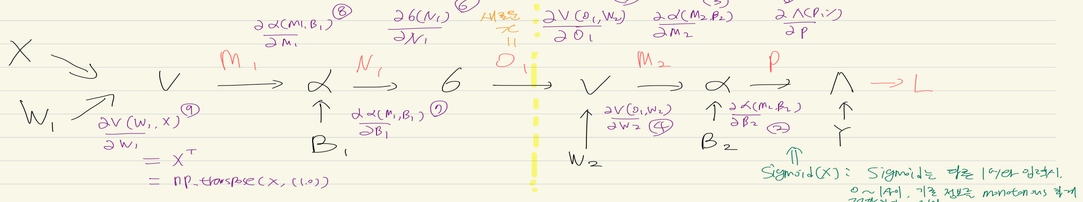
$\frac{\partial \Lambda}{\partial u} \odot \frac{\partial \sigma(XW)}{\partial u} * \frac{\partial XW}{\partial X}$

(5) Back Propagation (2 Hidden Layer)

2 Layer Network의 Backpropagation 과정 이해

2 Hidden Layer Backpropagation

[2 Layer Back prop]



① $\frac{\partial \Lambda(P,Y)}{\partial P} = \frac{1}{2} \times -2(Y-P) = \frac{1}{2} \times -2(\text{forward\_info}[\text{'y'}] - \text{forward\_info}[\text{'p'}])$ dL/dP

* $\Lambda = \Sigma (Y-P)^2$

② $\frac{\partial \alpha(M_2,B_2)}{\partial B_2} = I_{B_2} = np.ones\_like(\text{weights}[\text{'B2'}])$ $dP/dB2$

* $\alpha(M_2,B_2) = M_2 + B_2$

③ $\frac{\partial \alpha(M_2,B_2)}{\partial M_2} = I_{M_2} = np.ones\_like(\text{forward\_info}[\text{'M2'}])$ $dP/dM2$

④ $\frac{\partial V(O_1,W_2)}{\partial W_2} = O_1^T = np.transpose(\text{forward\_info}[\text{'O1'}], (1,0))$ $dM2/dW2$

* $V(O_1,W_2) = np.dot(O_1, W_2)$

⑤ $\frac{\partial V(O_1,W_2)}{\partial O_1} = W_2^T = np.transpose(\text{weights}[\text{'W2'}], (1,0))$ $dM2/dO1$

⑥ $\frac{\partial \sigma(N_1)}{\partial N_1} = deriv(sigmoid, N1) = Sigmoid(N1)(1 - Sigmoid(N1))$ $dO1/dN1$

(forward_info['N1'] / forward_info['N1'])

* $\sigma(N_1) = \sigma(M_1)$

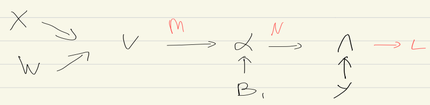
⑦ $\frac{\partial \alpha(M_1,B_1)}{\partial B_1} = I_{B_1} = np.ones\_like(\text{forward\_info}[\text{'B1'}])$ $dN1/dB1$

* $\alpha(M_1,B_1) = M_1 + B_1$

⑧ $\frac{\partial \alpha(M_1,B_1)}{\partial M_1} = I_{M_1} = np.ones\_like(\text{forward\_info}[\text{'M1'}])$ $dN1/dM1$

⑨ $\frac{\partial V(W_1,X)}{\partial W_1} = X^T = np.transpose(\text{forward\_info}[\text{'x'}], (1,0))$ $dM1/dW1$

[1 Layer Back prop]



$\Lambda = (Y-N)^2$ $\alpha = M + B_1$

$* \frac{\partial \Lambda}{W} = \frac{\partial \Lambda(N)}{\partial n} \cdot \frac{\partial \alpha(M,B_1)}{\partial m} \cdot \frac{\partial V(X,W)}{\partial W}$ $\quad * \frac{\partial \Lambda}{\partial B_1} = \frac{\partial \Lambda(N,Y)}{\partial N} \cdot \frac{\partial \alpha(M,B_1)}{\partial B_1}$

$\frac{\partial \Lambda}{\partial W} = X^T * \frac{\partial \Lambda(N)}{\partial n} \odot \frac{\partial \alpha(M,B_1)}{\partial m}$ $\quad = -2(Y-N) \odot I_B$

$\Rightarrow np.sum(\_\_\_, axis=0)$

① $dL/dW2 = \frac{\partial \Lambda}{\partial W2} = dM2/dW2 * dL/dP \odot dP/dM2 \Rightarrow np.dot(dM2/dW2, dL/dP)$

② $dL/dB2 = \frac{\partial \Lambda}{\partial B2} = dL/dP \odot dP/dB2 \Rightarrow np.sum(\_\_\_, axis=0)$

$* dL/dM2 = \frac{\partial \Lambda}{\partial M2} = dL/dP \odot dP/dM2_{=I_{M2}}$

$* dL/dO1 = dL/dM2 * dM2/dO1$

$* dL/dN1 = dL/dO1 \odot dO1/dN1$

$* dL/dM1 = dL/dN1 \odot dN1/dM1$

③ $dL/dW1 = dM1/dW1 * dL/dM1 \Rightarrow np.dot(dM1/dW1, dL/dM1)$

(X^T : "앞선 layer의 결과!")

④ $dL/dB1 = dL/dW1 * dN1/dB1 \Rightarrow np.sum(\_\_\_, axis=0)$

np.dot / ⊙ 곱셈

⇒ element wise 연산

행렬곱

⇒ np.dot은 scalar return!

[ ] [ ] <np.dot> [ ] [ ] <⊙>

<np.dot>

① Scalar끼리는 $\frac{\partial L}{\partial M_2}, \frac{\partial L}{\partial M_2}$

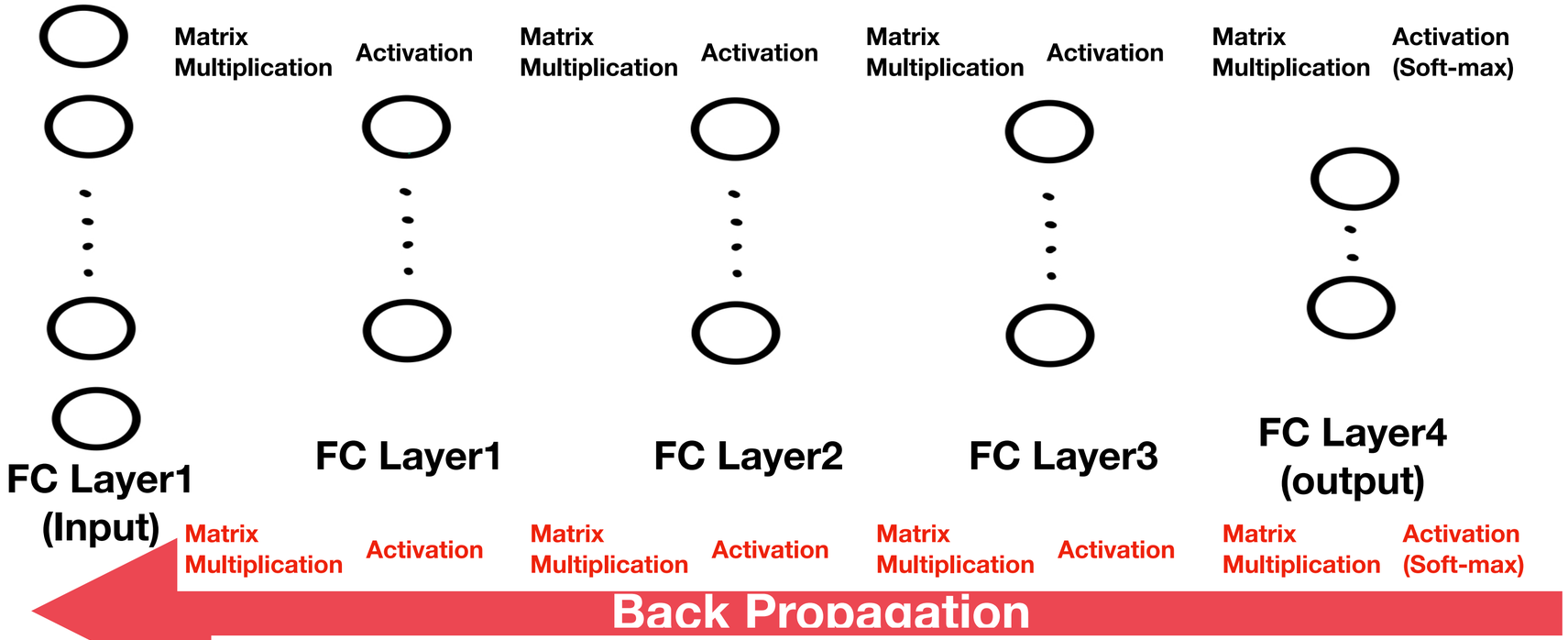
+

② Weight는 곱셈을 선호함 $\frac{\partial L}{\partial M_2} \times \frac{\partial M_2}{\partial W_2} = \frac{\partial L}{\partial W_2}$ ⇓ np.dot(W2, 결과)

(5) Back Propagation (with Pytorch)

1. Pytorch를 활용한 Backpropagation process

Pytorch의 Built-in-method(“grad_fn.next_functions” 등)를 활용해 Backpropagation 연산 과정 확인 (MNIST_digit_image dataset 대상)




```python
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(NeuralNet, self).__init__()
        self.relu = nn.ReLU()
        self.fc1 = nn.Linear(input_size, hidden_size, bias=False)
        self.fc2 = nn.Linear(hidden_size, hidden_size, bias=False)
        self.fc3 = nn.Linear(hidden_size, hidden_size, bias=False)
        self.fc4 = nn.Linear(hidden_size, num_classes, bias=False)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        out = self.relu(out)
        out = self.fc3(out)
        out = self.relu(out)
        out = self.fc4(out)
        return out
```

```python
params = list(model.parameters())

print(f'bias True 일때 parameter array 총 개수:{len(params)}')
for i in range(len(params)):
    print(f'parameter array shape: {params[i].shape}')
```

```
bias True 일때 parameter array 총 개수:4
parameter array shape: torch.Size([500, 784])
parameter array shape: torch.Size([500, 500])
parameter array shape: torch.Size([500, 500])
parameter array shape: torch.Size([10, 500])
```

```python
print("Loss part :              ", loss.grad_fn)
print("FC Layer 4(Activation) : ", loss.grad_fn.next_functions[0][0])
print("FC Layer 4 :             ",loss.grad_fn.next_functions[0][0].next_functions[0][0])
print("Fc Layer 3 :             ",loss.grad_fn.next_functions[0][0].next_functions[0][0].next_functions[0][0])
print("Fc Layer 3 :             ",loss.grad_fn.next_functions[0][0].next_functions[0][0].next_functions[0][0].next_functions[0][0])
print("Fc Layer 2 :             ",loss.grad_fn.next_functions[0][0].next_functions[0][0].next_functions[0][0].next_functions[0][0].next_functions[0][0])
print("Fc Layer 2 :             ",loss.grad_fn.next_functions[0][0].next_functions[0][0].next_functions[0][0].next_functions[0][0].next_functions[0][0].
print("Fc Layer 1 :             ",loss.grad_fn.next_functions[0][0].next_functions[0][0].next_functions[0][0].next_functions[0][0].next_functions[0][0].
print("Fc Layer 1(Input) :      ",loss.grad_fn.next_functions[0][0].next_functions[0][0].next_functions[0][0].next_functions[0][0].next_functions[0][0].
```

```
Loss part :              <NllLossBackward object at 0x7fc2449f7a20>
FC Layer 4(Activation) : <LogSoftmaxBackward object at 0x7fc2449f79e8>
FC Layer 4 :             <MmBackward object at 0x7fc2449f7a20>
Fc Layer 3 :             <ReluBackward0 object at 0x7fc2449f7a58>
Fc Layer 3 :             <MmBackward object at 0x7fc2449f79e8>
Fc Layer 2 :             <ReluBackward0 object at 0x7fc2449f7a90>
Fc Layer 2 :             <MmBackward object at 0x7fc2449f79e8>
Fc Layer 1 :             <ReluBackward0 object at 0x7fc2449f7a58>
Fc Layer 1(Input) :      <MmBackward object at 0x7fc2449f7a90>
```

[BackPropagation]

Input ← Loss

[Code_Link]

(5) Back Propagation (with pytorch)

2. Gradient descent process

```
gradient step:
    grad : -2.9021484806435183e-06
    layer1 weight : -4.344243279774673e-05, weight_updated : 0.0007414959436573554

    grad : -2.98062650472275e-06
    layer2 weight : 2.846606366802007e-05, weight_updated : 0.0006695874471915886

    grad : -1.4612485301768174e-06
    layer3 weight : -1.4833339264441747e-05, weight_updated : 0.0007128868501240504

    grad : 5.215406231884323e-12
    layer4 weight : 0.0006980535690672696, weight_updated : -5.820766091346741e-11
```

***1 iteration을 통해 weight 가 update 됨을 확인**

Forward

Backward

```
zero_grad:
    grad : 0.0
    layer1 weight : -4.3445343180792406e-05

    grad : 0.0
    layer2 weight : 2.8463089620345272e-05

    grad : 0.0
    layer3 weight : -1.4834816283837426e-05

    grad : 0.0
    layer4 weight : 0.0006980535108596087
```

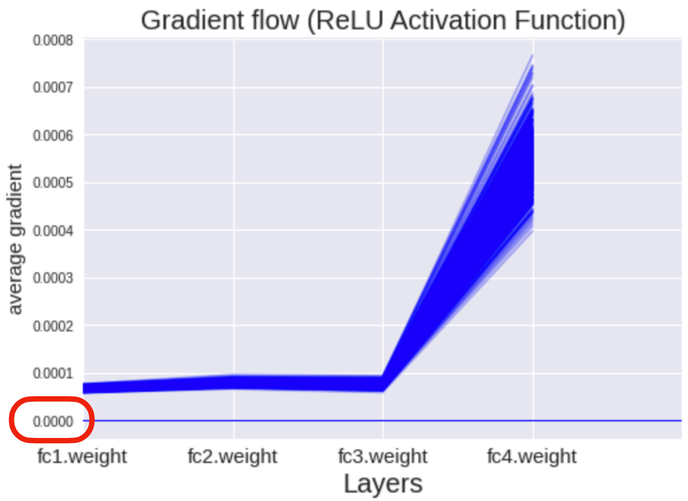
```
backward:
    grad : -2.9021484806435183e-06
    layer1 weight : -4.3445343180792406e-05

    grad : -2.98062650472275e-06
    layer2 weight : 2.8463089620345272e-05

    grad : -1.4612485301768174e-06
    layer3 weight : -1.4834816283837426e-05

    grad : 5.215406231884323e-12
    layer4 weight : 0.0006980535108596087
```

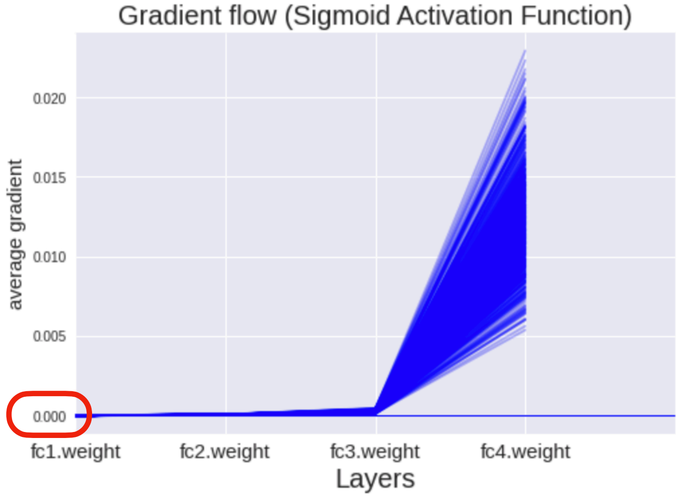
Gradient 계산

3. Gradient flow

각 Layer별 gradient(절대값, 평균)의 plot (# line = # batch)

Sigmoid activation fucntion 사용 시, ReLU와 비교해 backpropagation 과정에서 Gradient가 급격히 '0'에 가까워지는 현상(gradient vanishing) 확인





[Code_Link]

3. Machine Learning

(6) EDA (Exploratory Data Anlaysis) - “Google Playstore Data”

(7) Linear Regression - “Boston House Data”

(8) Linear Regression - “TOY PROJECT (“축구선수의 시장가치 예측과 SNS 지표의 기여도”)”

(9) Classification models practice

- **(a) Logistic Regression**
- **(b) LDA, QDA**
- **(c) Naive bayes**
- **(d) Decision tree**
- **(e) Ensemble (Random forest, Boosting)**
- **(f) SVM**

(6) EDA (Exploratory Data Analysis)

Google Play Store data EDA 분석

데이터 소개

Google Play Store Apps

저희가 사용한 data는 Google Play Store에서 수집된 데이터로 Kaggle에 올라와 있는 데이터를 사용

https://www.kaggle.com/lava18/google-play-store-apps

이 데이터는:

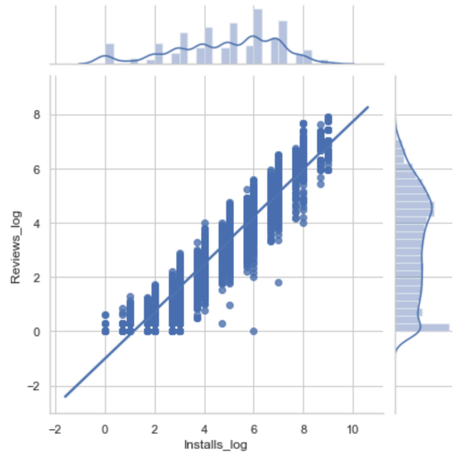
- 총 13개의 특징 칼럼
- 9660개의 unique한 값
- 총 10842개의 데이터
- csv 파일
- 1개를 제외한 12개의 특징 칼럼들은 모두 object 타입

데이터 전처리

1. 결측치 확인 Rating, Version data 결측치 존재
    - Rating : 1,473개 데이터 -> NaN값 데이터 존재 => 모두 0으로 처리 (사용자의 평가 유보 대상 서비스 판단, Install정보 기반)
        - Installs수(Rating 0 mean / 전체 mean / 전체 median) : (4,095, 1,546만, 10만)
    - Version : Current Ver 8개 / Android Ver 2개 => 모두 0으로 처리
2. 이상치 제거 2개 행(데이터) 이상치 제거
    - 총 rows 수 : 10841개 -> 10839개
3. 컬럼별 데이터타입 변경 Size, Reviews, Installs, Price, Rating : 숫자로 변경
4. 컬럼 추가
    - Log 값 적용한 컬럼 생성 : Installs_log, Reviews_log
    - 추가 이유 : 데이터의 증감 추세 유지 하 데이터 수치 너비를 좁혀 분석 용이성 제고

1. Reviews_log와 Installs_log 분석

```
1) log값 취한 데이터를 기준으로 상관성 분석
2) 목표하는 Installs 컬럼을 포함한 전체 특징들과 분석
```



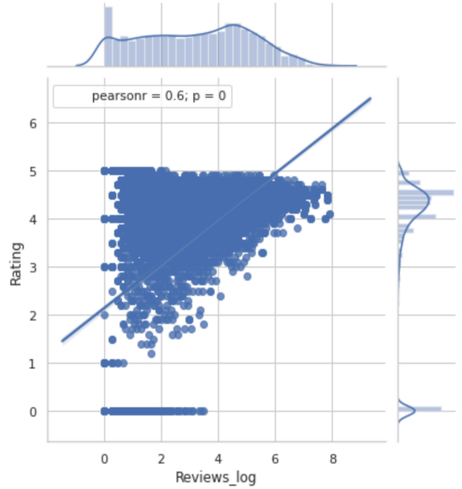

```
df.corr()['Reviews_log']
```

```
Rating           0.499426
Reviews          0.289243
Installs         0.314730
Price           -0.028723
Installs_log     0.956188
Reviews_log      1.000000
Name: Reviews_log, dtype: float64
```

2. Rating, Reviews_log 선형 분석

```
1) log값 취한 데이터를 기준으로 상관성 분석
2) Reviews와의 상관관계를 보이는 feature에 대한 추가 탐색
```

```
j = sns.jointplot(x="Reviews_log", y="Rating", data=df, kind='reg')
j.annotate(stats.pearsonr)
plt.show()
```



Conclusion

```
1) 높은 Installs과 Rating을 위해선 많은 Review 갯수가 중요하다는 관계 발견
2) Price : Price의 중요도는 상대적으로 낮은 것으로 확인됨 (상관계수 절대값 0.1 이상 feature 부재)
3) Category : 특정 Category에 치중된 상품 구성 (예상과 달리 Social의 비중이 적음)
4) Rating : 90% 가까이 4점 rating. 객관적인 평가가 이루어진다는 점에는 의문
```

[Code_Link]

(7) Linear Regression - Boston House Data

[회귀분석 대상 데이터 : Boston House dataset (from scikit-learn)]

1. Target data : MEDV, 1978년 보스턴 주택 가격 (506개 타운의 주택 가격 중앙값 (단위 1,000 달러))

2. Feature data(Columns) :

CRIM: 범죄율
INDUS: 비소매상업지역 면적 비율
NOX: 일산화질소 농도
RM: 주택당 방 수
LSTAT: 인구 중 하위 계층 비율
B: 인구 중 흑인 비율
PTRATIO: 학생/교사 비율
ZN: 25,000 평방피트를 초과 거주지역 비율
CHAS: 찰스강의 경계에 위치한 경우는 1, 아니면 0
AGE: 1940년 이전에 건축된 주택의 비율
RAD: 방사형 고속도로까지의 거리
DIS: 직업센터의 거리
TAX: 재산세율

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
501	0.06263	0.0	11.93	0.0	0.573	6.593	69.1	2.4786	1.0	273.0	21.0	391.99	9.67	22.4
502	0.04527	0.0	11.93	0.0	0.573	6.120	76.7	2.2875	1.0	273.0	21.0	396.90	9.08	20.6
503	0.06076	0.0	11.93	0.0	0.573	6.976	91.0	2.1675	1.0	273.0	21.0	396.90	5.64	23.9
504	0.10959	0.0	11.93	0.0	0.573	6.794	89.3	2.3889	1.0	273.0	21.0	393.45	6.48	22.0
505	0.04741	0.0	11.93	0.0	0.573	6.030	80.8	2.5050	1.0	273.0	21.0	396.90	7.88	11.9

506 rows × 14 columns

3. shape : (506,13 + 1)

(7) Linear Regression - Boston House Data

1. OLS regression analysis(Basic model)

$$MEDV = Const + w_1 CRIM + w_2 ZN + \ldots + w_{13} LSTAT$$

OLS Regression Results

Dep. Variable:	MEDV	R-squared:	0.741
Model:	OLS	Adj. R-squared:	0.734
Method:	Least Squares	F-statistic:	108.1
Date:	Wed, 15 Jul 2020	Prob (F-statistic):	6.72e-135
Time:	18:18:10	Log-Likelihood:	-1498.8
No. Observations:	506	AIC:	3026.
Df Residuals:	492	BIC:	3085.
Df Model:	13		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	36.4595	5.103	7.144	0.000	26.432	46.487
CRIM	-0.1080	0.033	-3.287	0.001	-0.173	-0.043
ZN	0.0464	0.014	3.382	0.001	0.019	0.073
INDUS	0.0206	0.061	0.334	0.738	-0.100	0.141
CHAS	2.6867	0.862	3.118	0.002	0.994	4.380
NOX	-17.7666	3.820	-4.651	0.000	-25.272	-10.262
RM	3.8099	0.418	9.116	0.000	2.989	4.631
AGE	0.0007	0.013	0.052	0.958	-0.025	0.027
DIS	-1.4756	0.199	-7.398	0.000	-1.867	-1.084
RAD	0.3060	0.066	4.613	0.000	0.176	0.436
TAX	-0.0123	0.004	-3.280	0.001	-0.020	-0.005
PTRATIO	-0.9527	0.131	-7.283	0.000	-1.210	-0.696
B	0.0093	0.003	3.467	0.001	0.004	0.015
LSTAT	-0.5248	0.051	-10.347	0.000	-0.624	-0.425

Omnibus:	178.041	Durbin-Watson:	1.078
Prob(Omnibus):	0.000	Jarque-Bera (JB):	783.126
Skew:	1.521	Prob(JB):	8.84e-171
Kurtosis:	8.281	Cond. No.	1.51e+04

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 1.51e+04. This might indicate that there are
strong multicollinearity or other numerical problems.

R-squared : 0.74
Condition number : 1.51e+04

[분석 결과]
1) 매우 높은 조건수를 확인
- 다중공선성 확인을 위해 추가적인 분석 필요
- 주요 변수들의 scaling으로 조건수 조정이 필요

Scaling

Lower Cond No.

2. OLS Regression Analysis(model with scaled features)

OLS Regression Results

Dep. Variable:	MEDV	R-squared:	0.741
Model:	OLS	Adj. R-squared:	0.734
Method:	Least Squares	F-statistic:	108.1
Date:	Wed, 15 Jul 2020	Prob (F-statistic):	6.72e-135
Time:	18:20:02	Log-Likelihood:	-1498.8
No. Observations:	506	AIC:	3026.
Df Residuals:	492	BIC:	3085.
Df Model:	13		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	22.3470	0.219	101.943	0.000	21.916	22.778
scale(CRIM)	-0.9281	0.282	-3.287	0.001	-1.483	-0.373
scale(ZN)	1.0816	0.320	3.382	0.001	0.453	1.710
scale(INDUS)	0.1409	0.421	0.334	0.738	-0.687	0.969
scale(NOX)	-2.0567	0.442	-4.651	0.000	-2.926	-1.188
scale(RM)	2.6742	0.293	9.116	0.000	2.098	3.251
scale(AGE)	0.0195	0.371	0.052	0.958	-0.710	0.749
scale(DIS)	-3.1040	0.420	-7.398	0.000	-3.928	-2.280
scale(RAD)	2.6622	0.577	4.613	0.000	1.528	3.796
scale(TAX)	-2.0768	0.633	-3.280	0.001	-3.321	-0.833
scale(PTRATIO)	-2.0606	0.283	-7.283	0.000	-2.617	-1.505
scale(B)	0.8493	0.245	3.467	0.001	0.368	1.331
scale(LSTAT)	-3.7436	0.362	-10.347	0.000	-4.454	-3.033
CHAS	2.6867	0.862	3.118	0.002	0.994	4.380

Omnibus:	178.041	Durbin-Watson:	1.078
Prob(Omnibus):	0.000	Jarque-Bera (JB):	783.126
Skew:	1.521	Prob(JB):	8.84e-171
Kurtosis:	8.281	Cond. No.	10.6

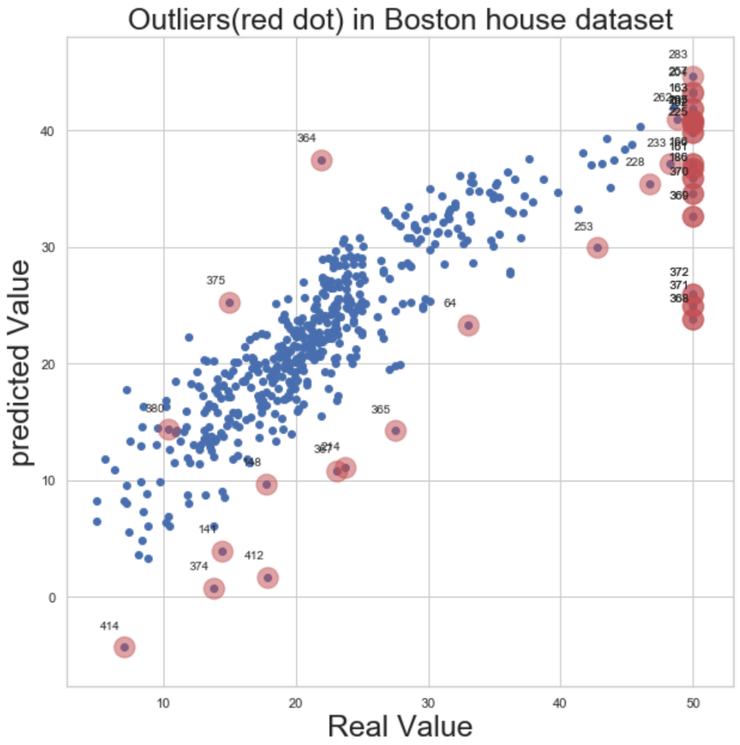
Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

R-squared : 0.74
Condition number : 10.6

[분석 결과]
1) 스케일링으로 조건수의 조정을 확인
- R-sqaured 값은 유지

(7) Linear Regression - Boston House Data

3. OLS Regression Analysis(model with outlier removed according to ‘Fox recommendation’)



Outliers Removed

OLS Regression Results

Dep. Variable:	MEDV	R-squared:	0.818
Model:	OLS	Adj. R-squared:	0.812
Method:	Least Squares	F-statistic:	126.9
Date:	Wed, 15 Jul 2020	Prob (F-statistic):	9.10e-127
Time:	18:38:44	Log-Likelihood:	-957.69
No. Observations:	380	AIC:	1943.
Df Residuals:	366	BIC:	1999.
Df Model:	13		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	21.4555	0.163	131.785	0.000	21.135	21.776
scale(CRIM)	-0.4351	0.236	-1.847	0.066	-0.898	0.028
scale(ZN)	0.9480	0.237	3.992	0.000	0.481	1.415
scale(INDUS)	-0.1444	0.294	-0.492	0.623	-0.722	0.433
scale(NOX)	-1.2558	0.328	-3.830	0.000	-1.901	-0.611
scale(RM)	2.6007	0.227	11.468	0.000	2.155	3.047
scale(AGE)	-0.7655	0.293	-2.616	0.009	-1.341	-0.190
scale(DIS)	-2.3313	0.314	-7.417	0.000	-2.949	-1.713
scale(RAD)	1.6630	0.393	4.228	0.000	0.889	2.436
scale(TAX)	-1.6769	0.406	-4.132	0.000	-2.475	-0.879
scale(PTRATIO)	-1.4405	0.196	-7.337	0.000	-1.827	-1.054
scale(B)	0.9273	0.188	4.939	0.000	0.558	1.297
scale(LSTAT)	-2.2301	0.299	-7.455	0.000	-2.818	-1.642
CHAS	1.1045	0.669	1.652	0.099	-0.210	2.419

Omnibus:	21.697	Durbin-Watson:	1.262
Prob(Omnibus):	0.000	Jarque-Bera (JB):	28.733
Skew:	0.465	Prob(JB):	5.76e-07
Kurtosis:	3.975	Cond. No.	10.7

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

The Number of Outliers : 55

R-squared : 0.818
Condition number : 10.7

[분석 결과]

1) Outlier의 제거를 통해 선형회귀 모델의 R-squared 상승(0.07 +) 확인
- 조건수는 유지

(7) Linear Regression - Boston House Data

4. OLS Regression Analysis(model with Polynomial model multicollinearity controlled)

OLS Regression Results

Dep. Variable:	MEDV	R-squared:	0.872
Model:	OLS	Adj. R-squared:	0.868
Method:	Least Squares	F-statistic:	199.9
Date:	Wed, 15 Jul 2020	Prob (F-statistic):	1.56e-185
Time:	18:47:46	Log-Likelihood:	317.45
No. Observations:	456	AIC:	-602.9
Df Residuals:	440	BIC:	-536.9
Df Model:	15		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	3.0338	0.007	433.880	0.000	3.020	3.048
scale(CRIM)	-0.3471	0.044	-7.976	0.000	-0.433	-0.262
scale(I(CRIM ** 2))	0.3075	0.071	4.331	0.000	0.168	0.447
scale(ZN)	-0.0465	0.022	-2.110	0.035	-0.090	-0.003
scale(I(ZN ** 2))	0.0440	0.020	2.206	0.028	0.005	0.083
scale(INDUS)	0.0037	0.012	0.323	0.747	-0.019	0.026
scale(NOX)	-0.0652	0.013	-5.001	0.000	-0.091	-0.040
scale(RM)	0.0999	0.011	9.195	0.000	0.079	0.121
scale(AGE)	-0.0273	0.011	-2.438	0.015	-0.049	-0.005
scale(np.log(DIS))	-0.1008	0.014	-7.368	0.000	-0.128	-0.074
scale(RAD)	0.1634	0.020	8.106	0.000	0.124	0.203
scale(TAX)	-0.0934	0.018	-5.153	0.000	-0.129	-0.058
scale(np.log(PTRATIO))	-0.0699	0.008	-8.872	0.000	-0.085	-0.054
scale(B)	0.0492	0.007	6.699	0.000	0.035	0.064
scale(np.log(LSTAT))	-0.1487	0.013	-11.074	0.000	-0.175	-0.122
CHAS	0.0659	0.026	2.580	0.010	0.016	0.116

Omnibus:	28.653	Durbin-Watson:	1.309
Prob(Omnibus):	0.000	Jarque-Bera (JB):	43.266
Skew:	0.465	Prob(JB):	4.03e-10
Kurtosis:	4.188	Cond. No.	35.2

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

R-squared : 0.872
Condition number : 35.2

[분석 결과]
1) 조건수가 다소 상승
- Polynomial 모델로서 다중공선성이 원인일 가능성이 높음

	VIF Factor	features
0	1.061624	CHAS
1	1.338325	scale(B)
2	1.478553	Intercept
3	1.780320	scale(np.log(PTRATIO))
4	2.596496	scale(RM)
5	3.748931	scale(AGE)
6	3.807459	scale(INDUS)
7	4.682812	scale(np.log(LSTAT))
8	5.071802	scale(NOX)
9	5.215025	scale(np.log(DIS))
10	9.107858	scale(TAX)
11	10.218588	scale(I(CRIM ** 2))
12	11.254736	scale(RAD)
13	11.751869	scale(I(ZN ** 2))
14	14.646056	scale(ZN)
15	21.260182	scale(CRIM)

각 feature별 다중공선성 영향 정도 확인
: 15개 중 6개 변수 선택해 모델 재구성

OLS Regression Results

Dep. Variable:	MEDV	R-squared:	0.836
Model:	OLS	Adj. R-squared:	0.834
Method:	Least Squares	F-statistic:	380.7
Date:	Sun, 19 Jul 2020	Prob (F-statistic):	1.42e-172
Time:	21:42:51	Log-Likelihood:	260.52
No. Observations:	456	AIC:	-507.0
Df Residuals:	449	BIC:	-478.2
Df Model:	6		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	3.0192	0.007	445.252	0.000	3.006	3.033
CHAS	0.0884	0.028	3.141	0.002	0.033	0.144
scale(B)	0.0558	0.008	6.989	0.000	0.040	0.072
scale(CRIM)	-0.1179	0.013	-9.120	0.000	-0.143	-0.092
scale(np.log(PTRATIO))	-0.0508	0.007	-6.936	0.000	-0.065	-0.036
scale(RM)	0.1153	0.011	10.828	0.000	0.094	0.136
scale(np.log(LSTAT))	-0.1570	0.011	-14.179	0.000	-0.179	-0.135

Omnibus:	29.141	Durbin-Watson:	1.113
Prob(Omnibus):	0.000	Jarque-Bera (JB):	42.637
Skew:	0.483	Prob(JB):	5.51e-10
Kurtosis:	4.145	Cond. No.	5.91

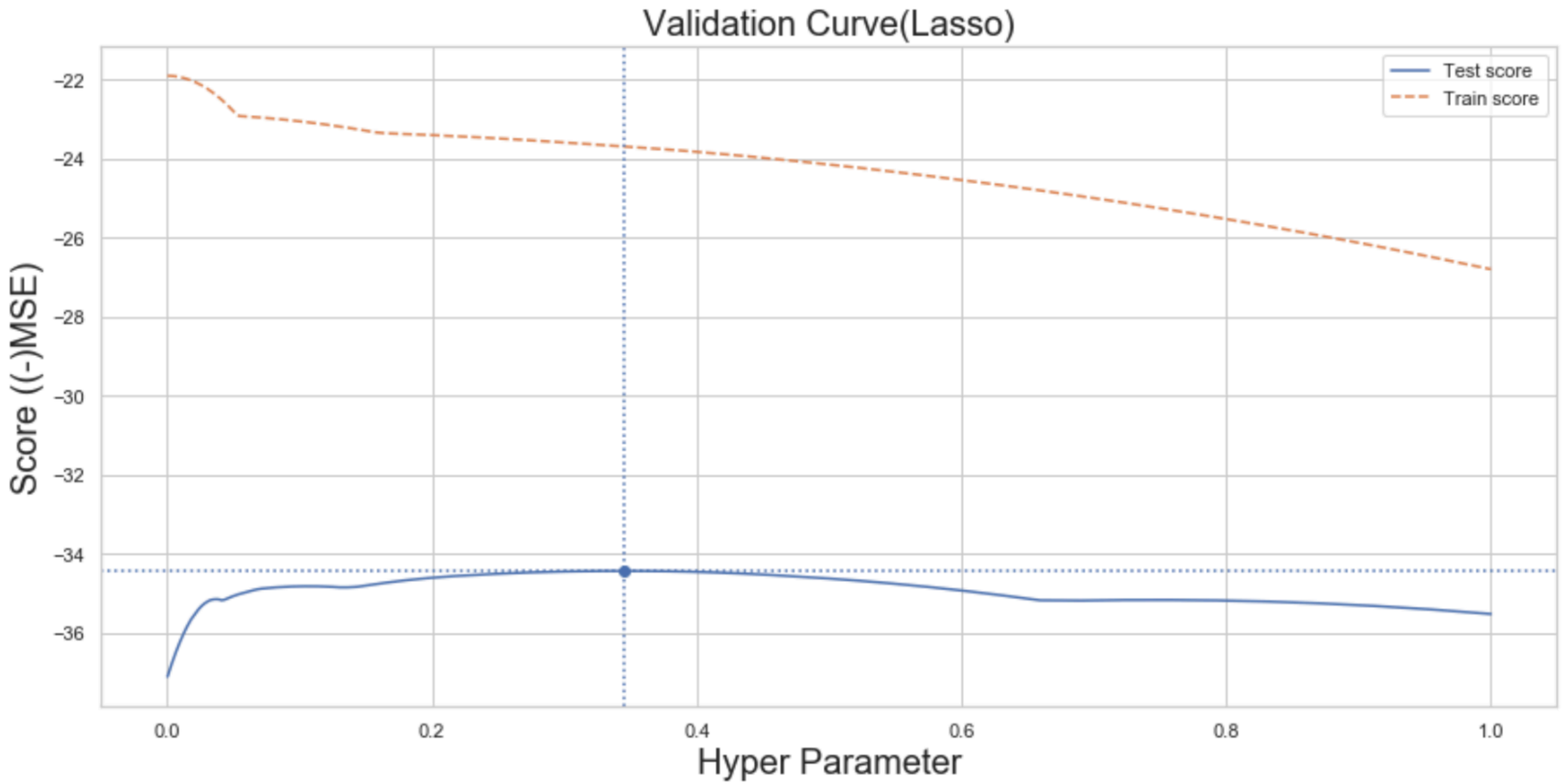
Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

R-squared : 0.836
Condition number : 5.91

[분석 결과]
1) 조건수 하락 및 성능 유지
- 절반도 안되는 변수로 성능은 유지하며,
조건수는 낮춰 보다 안정적인 성능 도출 가능함을 확인

(7) Linear Regression - Boston House Data

5. OLS Regression Analysis(Regularized model (Lasso))



Hyper Parameter 가 약 '0.345' 일 때, MSE 가 약 '-34.44' 로 최적의 Test 성능을 보임을 확인

(8) Linear Regression - TOY PROJECT (“축구선수의 시장가치 예측과 SNS 지표의 기여도”)

연구 결론 장표

DATA MAKETH VALUES

축구선수 시장가치 예측 시, SNS 지표의 기여도 확인

Model 1
- Domain based feature selection
- Attack & Mid-Field data

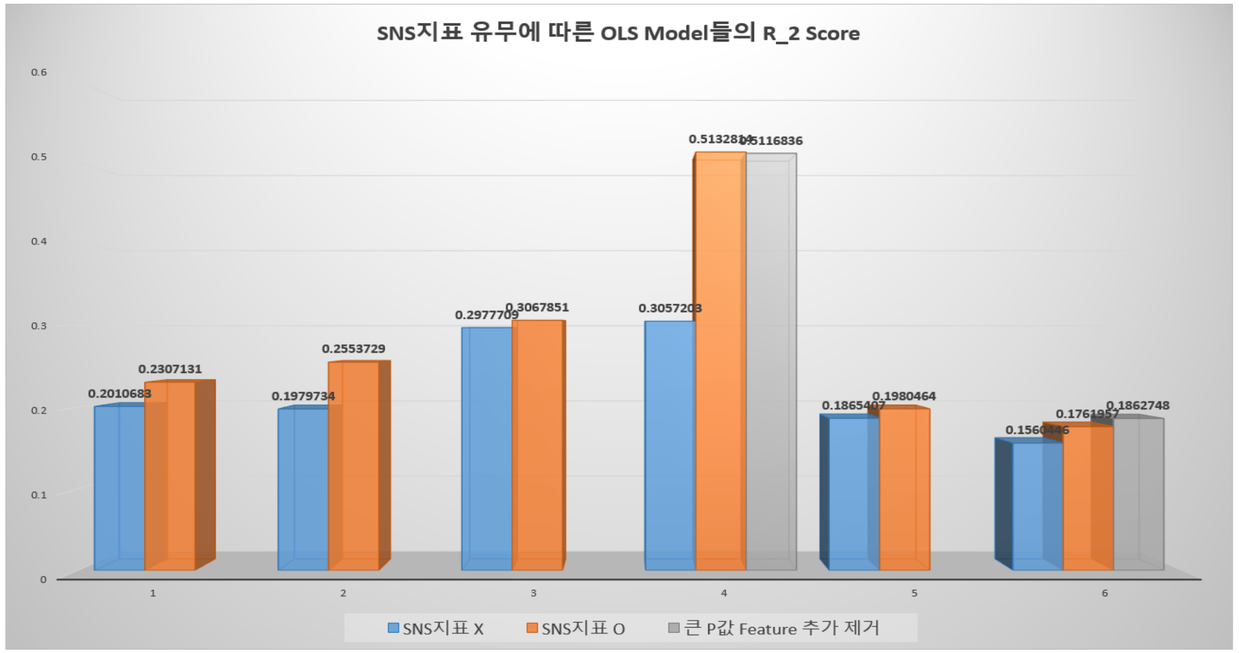
Model 2
- Domain based feature selection
- Attack data

Model 3
- P-value based Backward Elimination
- Attack & Mid-Field data

Model 4
- Domain based PCA & Backward elimination
- Attack Data

Model 5
- Domain based PCA & Backward elimination
- Attack + Mid-Field data

Model 6
- Explained variance based PCA & Backward elimination



+ 14.74% + 28.99% + 3.03% + 67.89% + 6.17% + 12.91%

6개 모델 모두 SNS feature 추가 시, R_sqaured 성능 향상 확인

[Code_Link]

(9) Classification models practice _ Logistic Regression

1. Statsmodels 패키지를 활용한 Logistic Regression practice - Iris data를 대상으로 품종 분류 실습

[Iris data set 개괄]

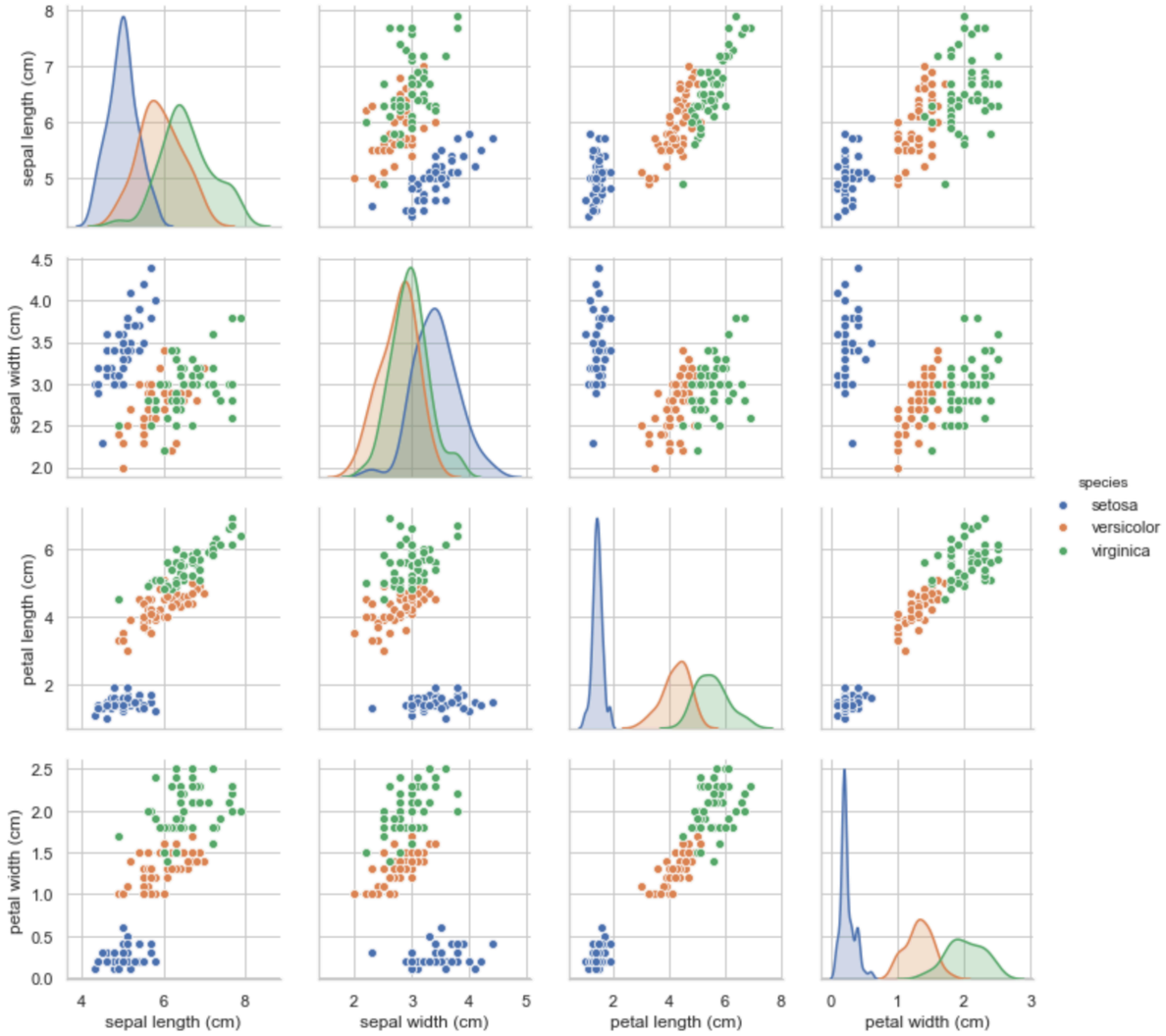
[dataset]

4 Feature
- Petal width
- Petal length
- Sepal width
- Sepal length

3 Label
- Setosa(0)
- Versicolor(1)
- Virginica(2)

```
**Data Set Characteristics:**

    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, predictive attributes and the class
    :Attribute Information:
        - sepal length in cm
        - sepal width in cm
        - petal length in cm
        - petal width in cm
        - class:
                - Iris-Setosa
                - Iris-Versicolour
                - Iris-Virginica

    :Summary Statistics:

    ============== ==== ==== ======= ===== ====================
                    Min  Max   Mean    SD   Class Correlation
    ============== ==== ==== ======= ===== ====================
    sepal length:   4.3  7.9   5.84   0.83    0.7826
    sepal width:    2.0  4.4   3.05   0.43   -0.4194
    petal length:   1.0  6.9   3.76   1.76    0.9490  (high!)
    petal width:    0.1  2.5   1.20   0.76    0.9565  (high!)
    ============== ==== ==== ======= ===== ====================

    :Missing Attribute Values: None
    :Class Distribution: 33.3% for each of 3 classes.
    :Creator: R.A. Fisher
    :Donor: Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)
    :Date: July, 1988
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species
145	6.7	3.0	5.2	2.3	virginica
146	6.3	2.5	5.0	1.9	virginica
147	6.5	3.0	5.2	2.0	virginica
148	6.2	3.4	5.4	2.3	virginica
149	5.9	3.0	5.1	1.8	virginica

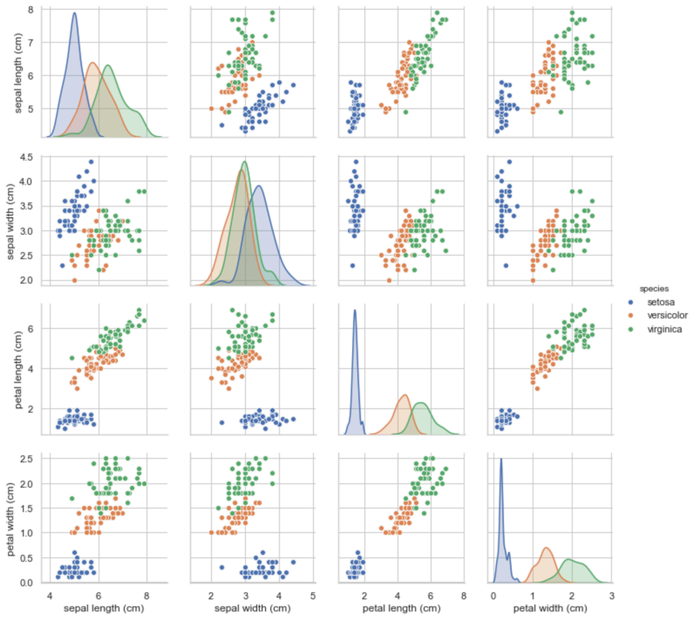


[Study_Link] [Code_Link]

(9) Classification models practice _ Logistic Regression

1. Statsmodels 패키지를 활용한 Logistic Regression practice - Iris data를 대상으로 품종 분류 실습

[Data]



[Logistic model]

4 Features
- Petal width
- Petal length
- Sepal width
- Sepal length

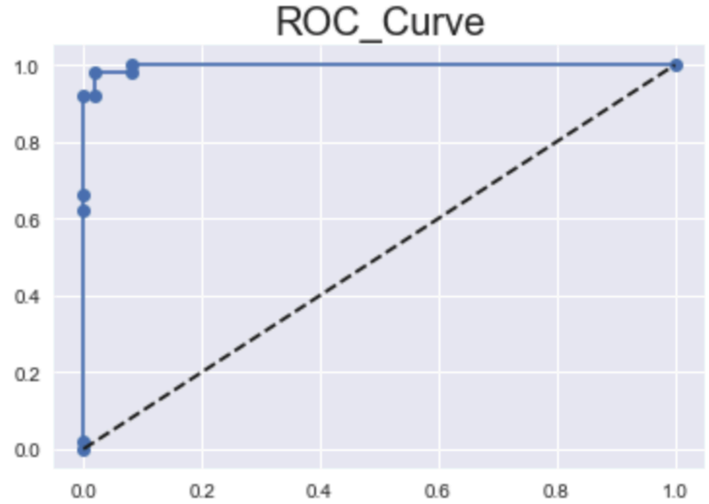
2 Labels
- Versicolor(0)
- Virginica(1)

Classification (Logistic model)

[Classification result]

```python
# classifcation report

from sklearn.metrics import classification_report

print(classification_report(y_true, y_pred, target_names=['versicolor(0)', 'virginica(1)']))
```

```
                precision    recall  f1-score   support

versicolor(0)       0.98      0.98      0.98        50
virginica(1)        0.98      0.98      0.98        50

    accuracy                            0.98       100
   macro avg        0.98      0.98      0.98       100
weighted avg        0.98      0.98      0.98       100
```



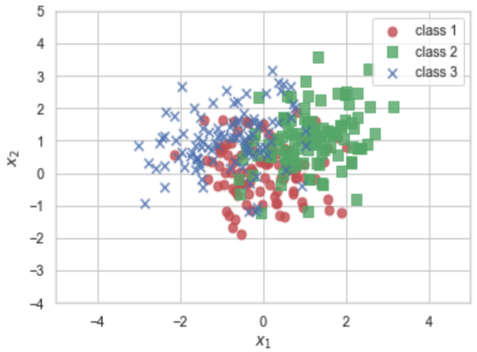
[Study_Link] [Code_Link]

(9) Classification models practice _ LDA, QDA

1. Scikit-learn 패키지를 활용한 QDA 실습 (Data from Multivariate Normal distribution random sampling)

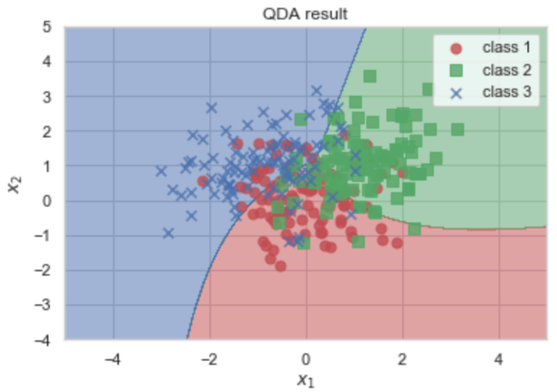
[Data]

```
N=100
rv1 = sp.stats.multivariate_normal([ 0, 0], [[0.7, 0.0], [0.0, 0.7]])
rv2 = sp.stats.multivariate_normal([ 1, 1], [[0.8, 0.2], [0.2, 0.8]])
rv3 = sp.stats.multivariate_normal([-1, 1], [[0.8, 0.2], [0.2, 0.8]])
np.random.seed(0)
X1 = rv1.rvs(N)
X2 = rv2.rvs(N)
X3 = rv3.rvs(N)
y1 = np.zeros(N)
y2 = np.ones(N)
y3 = 2 * np.ones(N)
X = np.vstack([X1, X2, X3])
y = np.hstack([y1, y2, y3])

plt.scatter(X1[:, 0], X1[:, 1], alpha=0.8, s=50, marker="o", color='r', label="class 1")
plt.scatter(X2[:, 0], X2[:, 1], alpha=0.8, s=50, marker="s", color='g', label="class 2")
plt.scatter(X3[:, 0], X3[:, 1], alpha=0.8, s=50, marker="x", color='b', label="class 3")
plt.xlim(-5, 5)
plt.ylim(-4, 5)
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")
plt.legend()
plt.show()
```



[QDA classification result]

```
# QDA를 이용한 판별경계선 시각화

x1min, x1max = -5, 5
x2min, x2max = -4, 5
XX1, XX2 = np.meshgrid(np.arange(x1min, x1max, (x1max-x1min)/1000),
                       np.arange(x2min, x2max, (x2max-x2min)/1000))
YY = np.reshape(qda.predict(np.array([XX1.ravel(), XX2.ravel()]).T), XX1.shape)
cmap = mpl.colors.ListedColormap(sns.color_palette(["r", "g", "b"]).as_hex())
plt.contourf(XX1, XX2, YY, cmap=cmap, alpha=0.5)
plt.scatter(X1[:, 0], X1[:, 1], alpha=0.8, s=50, marker="o", color='r', label="class 1")
plt.scatter(X2[:, 0], X2[:, 1], alpha=0.8, s=50, marker="s", color='g', label="class 2")
plt.scatter(X3[:, 0], X3[:, 1], alpha=0.8, s=50, marker="x", color='b', label="class 3")
plt.xlim(x1min, x1max)
plt.ylim(x2min, x2max)
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")
plt.title("QDA result")
plt.legend()
plt.show()
```

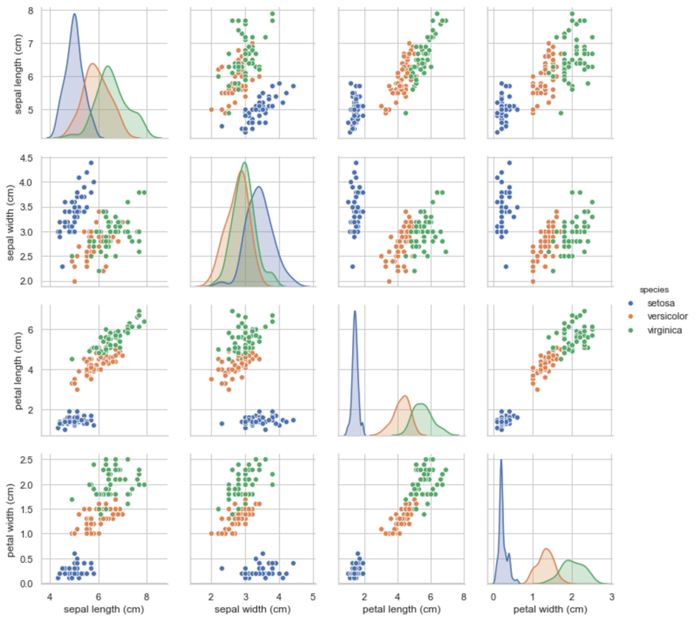


[Study_Link] [Code_Link]

(9) Classification models practice _ LDA, QDA

2. Scikit-learn 패키지를 활용한 LDA 실습 (Iris data)

[Data]



[LDA model]

4 Features
- Petal width
- Petal length
- Sepal width
- Sepal length

3 Labels
- Setosa(0)
- Versicolor(1)
- Virginica(2)

Classification (LDA model)

[Classification result]

```
y_pred = lda.predict(X)
y_true = y
confusion_matrix(y_true, y_pred)
```

```
array([[50,  0,  0],
       [ 0, 48,  2],
       [ 0,  1, 49]])
```

```
print(classification_report(y_true, y_pred, target_names=iris.target_names))
```

```
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        50
  versicolor       0.98      0.96      0.97        50
   virginica       0.96      0.98      0.97        50

    accuracy                           0.98       150
   macro avg       0.98      0.98      0.98       150
weighted avg       0.98      0.98      0.98       150
```

[Study_Link] [Code_Link]

(9) Classification models practice _ Naive Bayes

1. Scikit-learn 패키지를 활용한 Naive Bayes classification model 실습

- **Naver sentiment movie corpus v1.0 Dataset 활용** (dataset : https://github.com/e9t/nsmc)

[Dataset]

- All 200k reviews
- 150 (train), 50(test) reviews data
- 100 (positive review, rating 9-10), 100(negative review, rating 1-4) data

- 영화 리뷰데이터에 대한 Sentiment analysis(긍정 / 부정 판단)

```
3  pprint(data[:10])
```

```
[['9976970', '아 더빙.. 진짜 짜증나네요 목소리', '0'],
 ['3819312', '흠...포스터보고 초딩영화줄....오버연기조차 가볍지 않구나', '1'],
 ['10265843', '너무재밓었다그래서보는것을추천한다', '0'],
 ['9045019', '교도소 이야기구먼 ..솔직히 재미는 없다..평점 조정', '0'],
 ['6483659',
  '사이몬페그의 익살스런 연기가 돋보였던 영화!스파이더맨에서 늙어보이기만 했던 커스틴 던스트가 너무나도 이뻐보였다',
  '1'],
 ['5403919', '막 걸음마 뗀 3세부터 초등학교 1학년생인 8살용영화.ㅋㅋㅋ...별반개도 아까움.', '0'],
 ['7797314', '원작의 긴장감을 제대로 살려내지못했다.', '0'],
 ['9443947',
  '별 반개도 아깝다 욕나온다 이응경 길용우 연기생활이몇년인지..정말 발로해도 그것보단 낫겟다 납치.감금만반복반복..이드라마는 가족도없다 '
  '연기못하는사람만모엿네',
  '0'],
 ['7156791', '액션이 없는데도 재미 있는 몇안되는 영화', '1'],
 ['5912145', '왜케 평점이 낮은건데? 꽤 볼만한데.. 헐리우드식 화려함에만 너무 길들여져 있나?', '1']]
```

[Study_Link] [Code_Link]

(9) Classification models practice _ Naive Bayes

Classification Report (부정 : '0', 긍정 : '1')

1. 전처리 : CountVectorizer, 모델 : NB-multinomial

	precision	recall	f1-score	support
0	0.81	0.84	0.83	24827
1	0.84	0.81	0.82	25173
accuracy			0.83	50000
macro avg	0.83	0.83	0.83	50000
weighted avg	0.83	0.83	0.83	50000

2. 전처리 : CountVectorizer, 모델 : NB-multinomial
Konlpy 형태소 분석기 활용

	precision	recall	f1-score	support
0	0.85	0.86	0.85	24827
1	0.86	0.85	0.85	25173
accuracy			0.85	50000
macro avg	0.85	0.85	0.85	50000
weighted avg	0.85	0.85	0.85	50000

3. 전처리 : CountVectorizer, 모델 : NB-multinomial
Konlpy 형태소 분석기 활용
gram 범위 : 1-2 gram

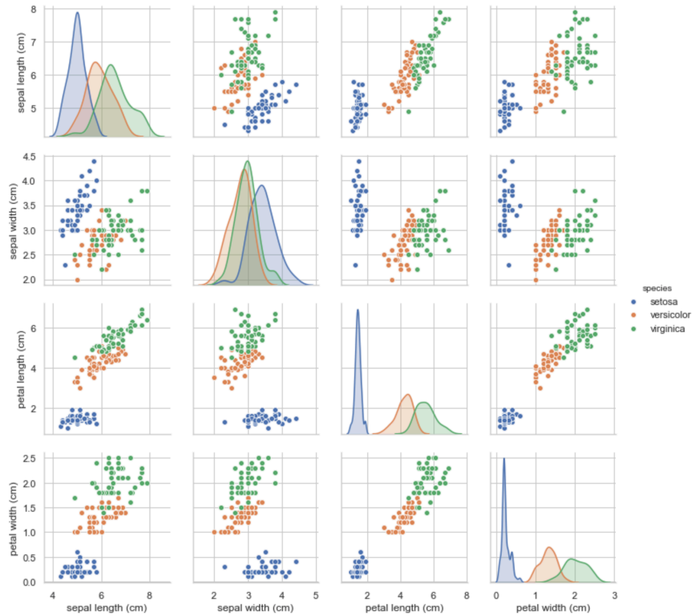
	precision	recall	f1-score	support
0	0.86	0.87	0.87	24827
1	0.87	0.86	0.87	25173
accuracy			0.87	50000
macro avg	0.87	0.87	0.87	50000
weighted avg	0.87	0.87	0.87	50000

[Study_Link] [Code_Link]

(9) Classification models practice _ Decision Tree

1. Scikit-learn 패키지를 활용한 Decision Tree 실습 (IRIS data)

[Data]
- Iris data set

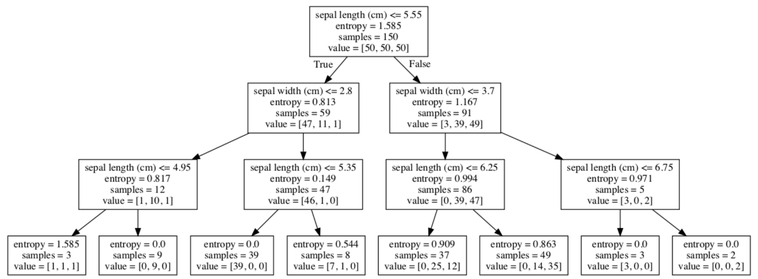


[Decision Tree model]

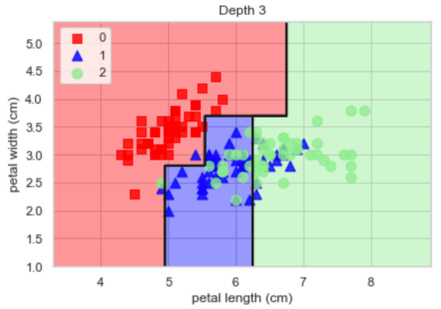
2 Features
- Sepal width
- Sepal length

3 Labels
- Setosa(0)
- Versicolor(1)
- Virginica(2)

Classification (Decision Tree model)

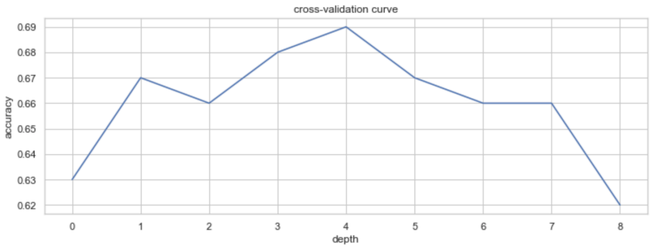


[Classification result]



	precision	recall	f1-score	support
0	0.94	1.00	0.97	50
1	0.74	0.68	0.71	50
2	0.73	0.74	0.73	50
accuracy			0.81	150
macro avg	0.80	0.81	0.80	150
weighted avg	0.80	0.81	0.80	150

[K-fold(k=5) Cross Validation Curve]



[Study_Link] [Code_Link]

(9) Classification models practice _ Ensemble Models

[Titanic data set 개괄]

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S

[Dataset]

10 Feature
- Age
- Sex
- Fare
- Embarked
- Etc

2 Label
- Survived(1)
- Dead(0)

[Preprocessing]

1. Missing value 처리
2. One-hot encoding

[대상 column]

1) ‘Sex’
2) ‘Embarked’
3) ‘Age’
4) ‘Fare(test_data)’

[Classification model]

1. Bagging Classifier
2. Random Forest model
3. Boosting(Gradient) model

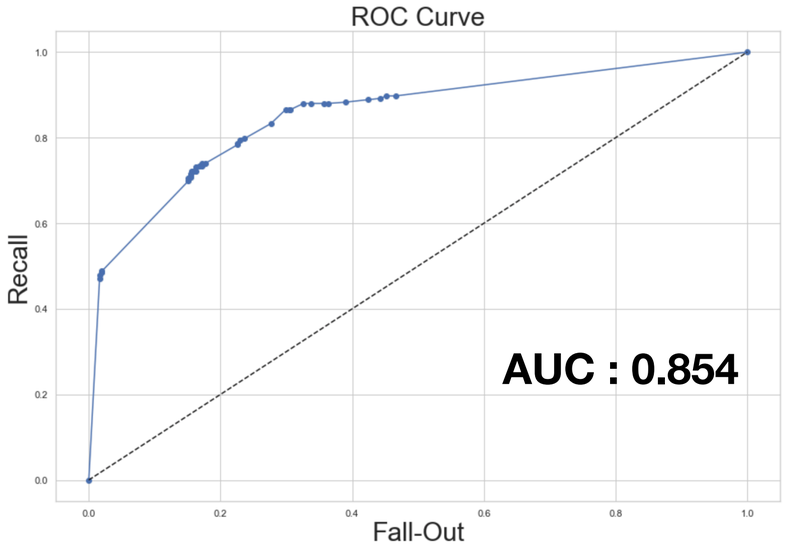
[Study_Link] [Code_Link]

(9) Classification models practice _ Ensemble Models

1. Bagging Classifier (Decision Tree based)

[Classification Result]

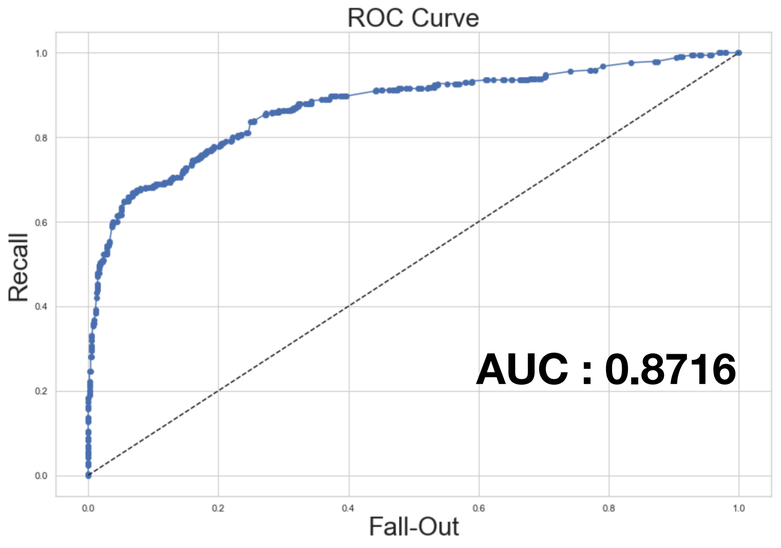
	precision	recall	f1-score	support
0	0.82	0.85	0.83	549
1	0.74	0.70	0.72	342
accuracy			0.79	891
macro avg	0.78	0.78	0.78	891
weighted avg	0.79	0.79	0.79	891



2. Random Forest

[Classification Result]

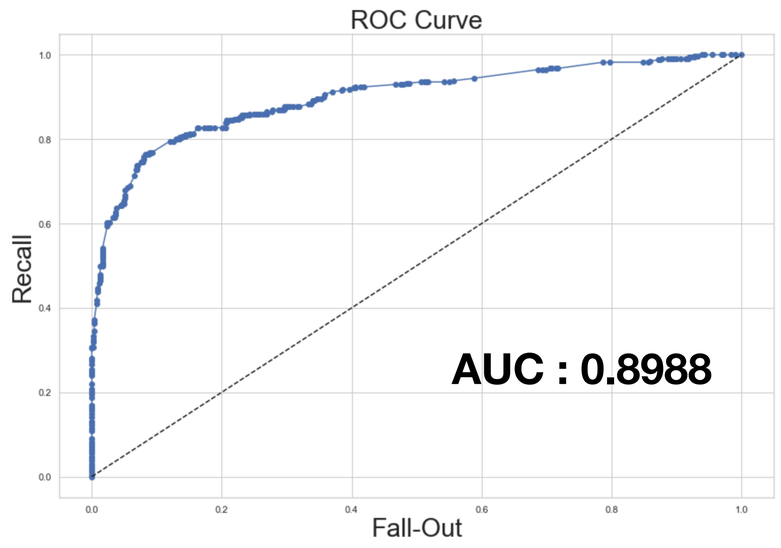
	precision	recall	f1-score	support
0	0.81	0.95	0.87	549
1	0.89	0.63	0.74	342
accuracy			0.83	891
macro avg	0.85	0.79	0.81	891
weighted avg	0.84	0.83	0.82	891



3. Gradient Boosting

[Classification Result]

	precision	recall	f1-score	support
0	0.83	0.95	0.88	549
1	0.89	0.68	0.77	342
accuracy			0.85	891
macro avg	0.86	0.81	0.83	891
weighted avg	0.85	0.85	0.84	891



[Study_Link] [Code_Link]

(9) Classification models practice _ SVM

[Data]
- Iris data set

[Support Vector Machine]

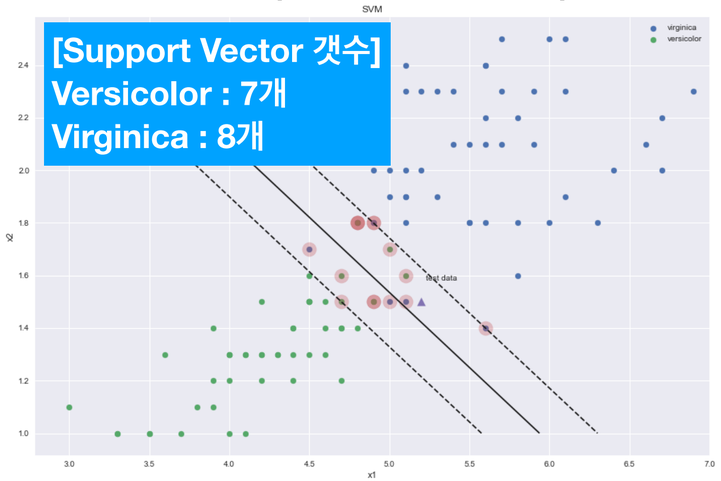
2 Features
- Petal width
- Petal length

3 Labels
- Versicolor(0)
- Virginica(1)

Classification (SVM)
- Large “C” coefficient
- Small “C” coefficient

[Classification result]

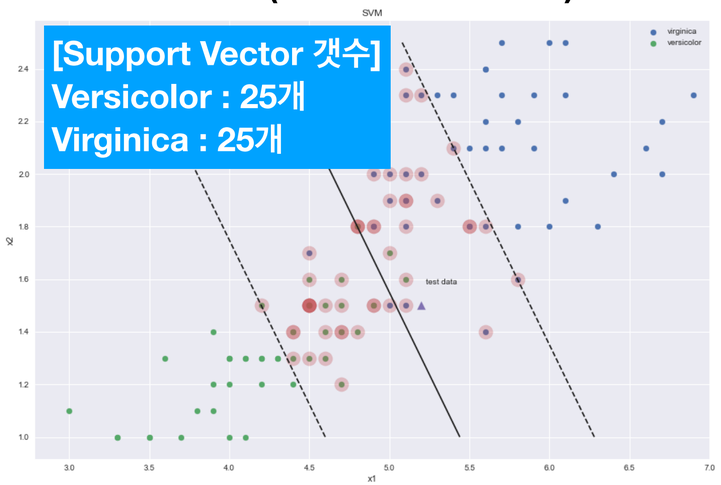
1. SVM (C coefficient = 10)



	precision	recall	f1-score	support
versicolor	0.96	0.94	0.95	50
virginica	0.94	0.96	0.95	50
accuracy			0.95	100
macro avg	0.95	0.95	0.95	100
weighted avg	0.95	0.95	0.95	100

보다 큰 C coefficient로 인해,
Slack Variable 허용 조건이 까다로워 짐
이로 인해, Margin 영역이 좁아진 것을 확인

2. SVM (C coefficient = 0.1)



	precision	recall	f1-score	support
versicolor	0.96	0.94	0.95	50
virginica	0.94	0.96	0.95	50
accuracy			0.95	100
macro avg	0.95	0.95	0.95	100
weighted avg	0.95	0.95	0.95	100

작은 C coefficient로 인해,
Slack Variable 허용 조건이 완화됨
이로 인해, Margin 영역이 넓게 형성됨을 확인

Outlier등 분산이 넓은 데이터에 대해
보다 안정적인 모델 형성 가능

[Study_Link] [Code_Link]

4. Deep Learning

(10) Learning Process 이해 (Linear model under different hyper parameters)

(11) MLP Implementation (from scratch)

(12) CNN

- **a. VGG13/19 model + CIFAR-10**

(13) U-Net Paper 구/현

(10) Learning Process 이해 (Linear model under Different Hyper parameters)

Learning Process 이해를 위한 실험

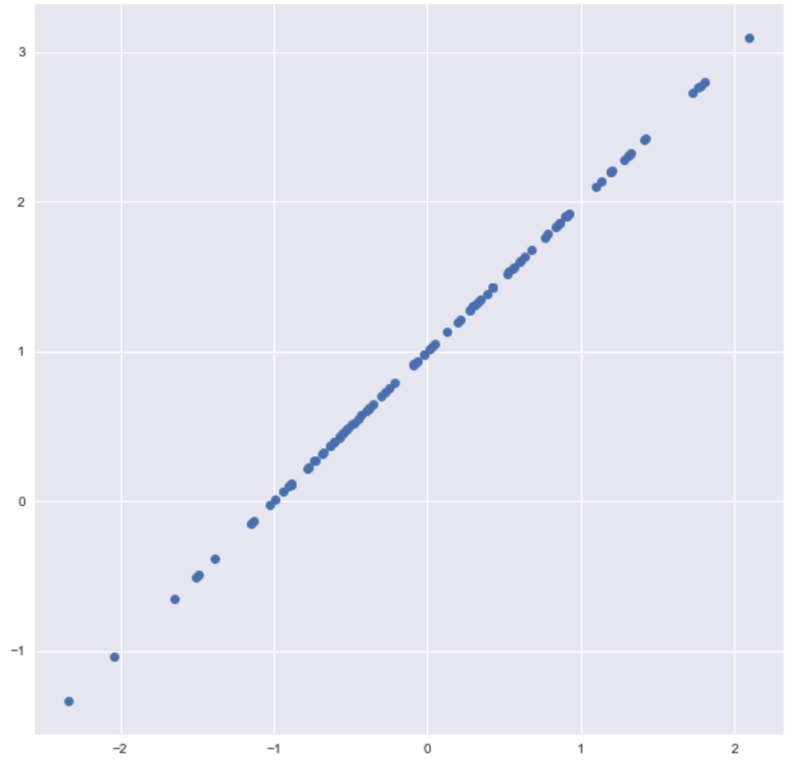
[Model and Data]
1) # data : 100
2) # features of data : 2 (weight : 1 + bias : 1)
3) Distribution of data : Normal Distribution (Randomly generation)
4) Model : Linear model (Y = wx + 1)

[실험 변수]
1) Learning rate
2) Batch size
3) Noise of data
4) Standard deviation (data)
5) Mean (data)

Dataset

```
N = 100
x_data = np.random.normal(0,1,size=(N,1))
y_data = x_data + 1

plt.style.use("seaborn")
fig, ax = plt.subplots(figsize=(10,10))
ax.plot(x_data,y_data,'bo')
print(x_data.shape,y_data.shape)
```

```
(100, 1) (100, 1)
```

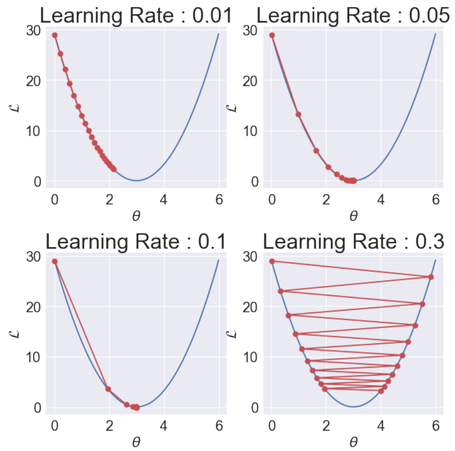


[Code_Link]

(10) Learning Process 이해 (Linear model under Different Hyper parameters)

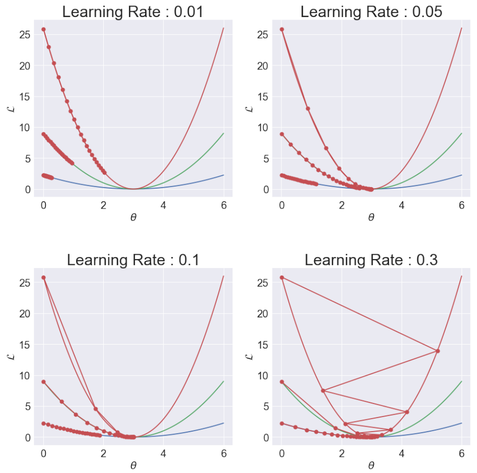
1. Learning rate

[학습 과정] : 동일 데이터, 학습률 변화



1) Learning rate와 학습속도는 비례
*단, Learning rate가 높아질 수록 parameter값이 발산 가능

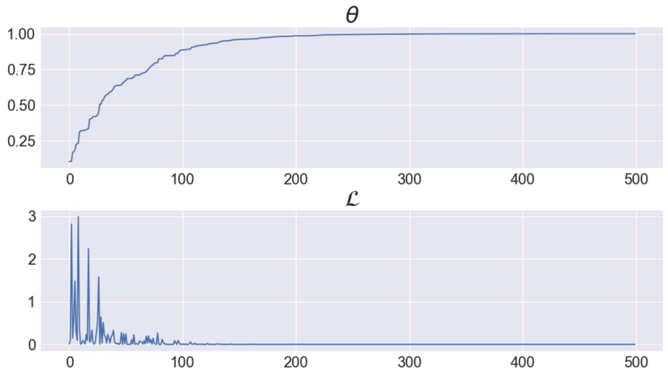
[학습 과정] : 다른 데이터, 학습률 변화



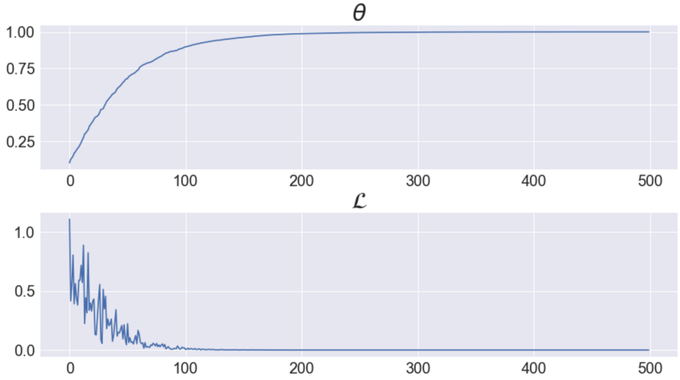
1) Data sample의 절대값이 클 수록, Learning rate에 더욱 민감

2. Batch size

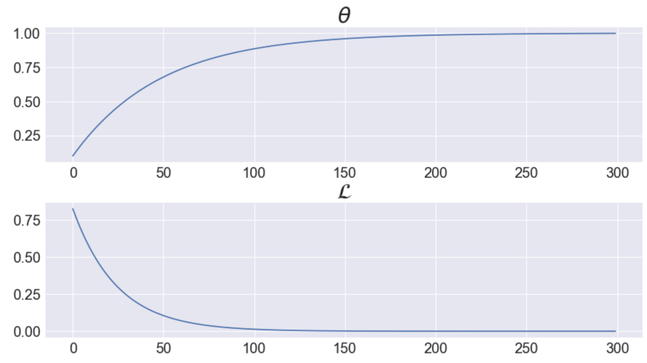
Batch size = 1 (Stochastic Gradient Descent)



Batch size = 8 (Mini-batch Gradient Descent)



Batch size = 100 (Full, Batch Gradient Descent)



[Batch size 확대의 영향]

1) Outlier의 효과가 Pooling 되어 보다 안정적으로 학습이 진행됨
2) Epoch당 연산 시간의 소요가 줄어듬

[Code Link]

(10) Learning Process 이해 (Linear model under Different Hyper parameters)

3. Noise of data

Question.3-12

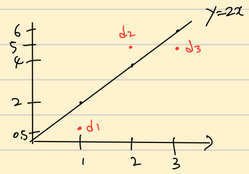
대부분의 dataset에는 noise가 섞여있다. $y = 2x$에서 이상적으로 만들어진 Dataset이 다음과 같을 때

$$D = \{(1,2), (2,4), (3,6)\}$$

noise에 의해 실제로 수집한 dataset이 다음과 같이 왜곡되었다고 가정하자.

$$D = \{(1,0.5), (2,5), (3,5)\}$$

θ가 2이었을때, 즉 predictor가 target function일때 각 data sample에 의해 θ가 update되는 값들을 구하고, noise가 학습에 미치는 영향을 분석하시오.



$\theta_{init} = 2$

$\theta_i = \theta_{i-1} + 2\alpha x_{i-1}(y_{i-1} - \theta_{i-1}x_{i-1})$

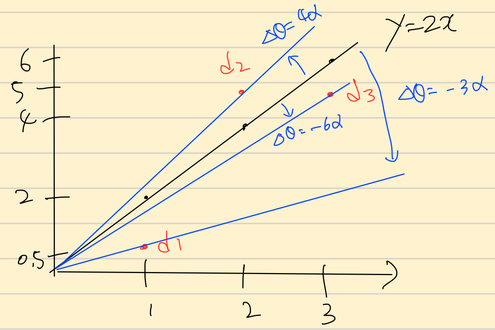
$Loss = (y_i - \theta x_i)^2$

Noise 이전 데이터에 대해 기존 Predictor($\theta=2$)는 완전한 성능을 보인다. 하지만 Noise로 인해 발생한 새로 데이터의 Loss가 반영, θ의 업데이트가 잘못된 상황이다. 각 Noise의 포함을 보기위해, update는 시키지 않고 Loss와 θ의 변화량을 관찰할 것이다.

Data point : (1, 0.5) $L = (0.5-2)^2 = 2.25$ $\Delta\theta = 2\alpha \cdot 1(0.5-2) = -3\alpha$

Data point : (2, 5) $L = (5-4)^2 = 1$ $\Delta\theta = 2\alpha \cdot 2(5-4) = 4\alpha$

Data point : (3, 5) $L = (5-6)^2 = 1$ $\Delta\theta = 2\alpha \cdot 3(5-6) = -6\alpha$



Noise

Data가 ↑로 왜곡되면 Predictor도 ↑로 왜곡되고,

Data가 ↓로 왜곡되면, Predictor도 ↓로 왜곡된다.

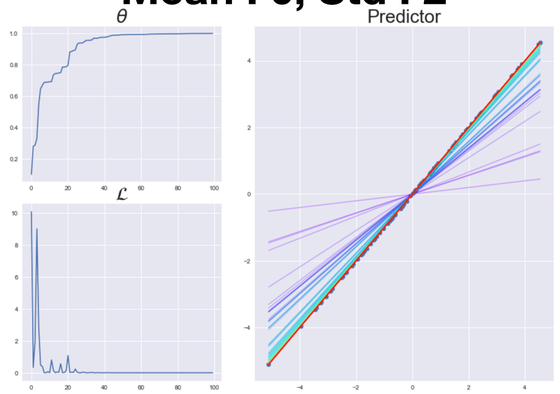
Sample data에 포함된 Noise에 가깝게 parameter의 학습이 진행됨

시사점 : 데이터의 분산 정도가 클 수록, 학습의 불안정성이 높아질 수 있음

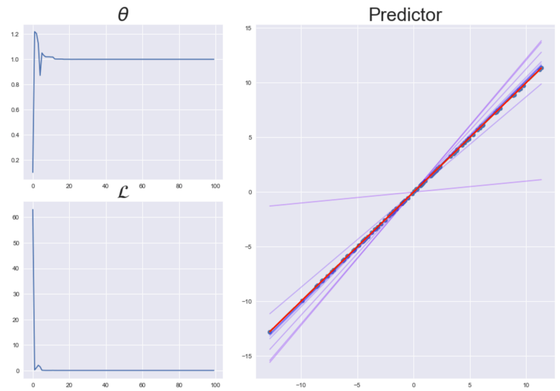
(10) Learning Process 이해 (Linear model under Different Hyper parameters)

4. Standard deviation (data)

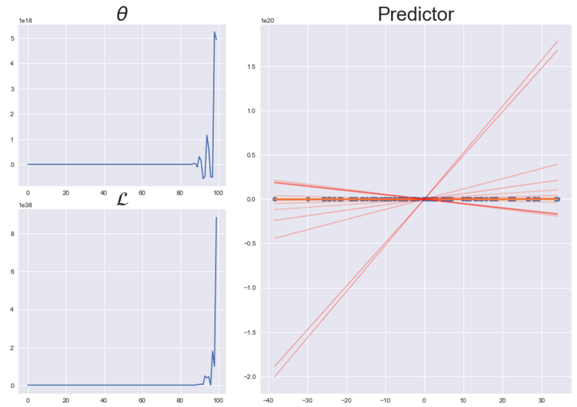
Mean : 0, Std : 2



Mean : 0, Std : 5



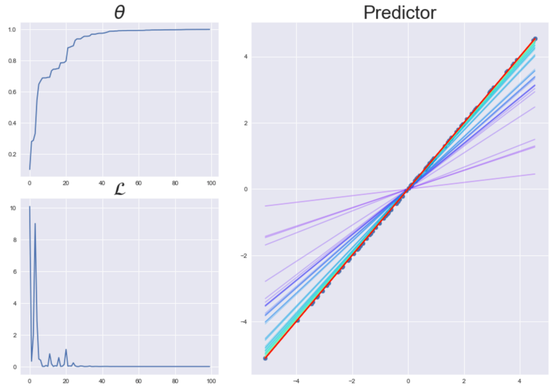
Mean : 0, Std : 15



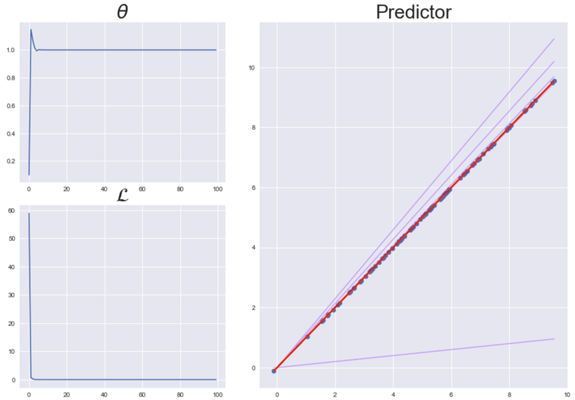
- 데이터의 표준편차가 클 수록, Parameter의 하습 보폭이 넓어짐 (Learning rate가 커지는 것과 같은 효과)
- 따라서, 표준편차가 큰 데이터 학습 시 Learning rate를 보다 작은 단위로 설정할 필요가 있음

5. Mean (data)

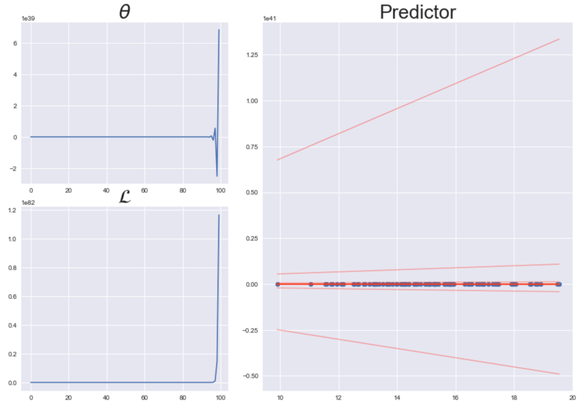
Mean : 0, Std : 2



Mean : 5, Std : 2



Mean : 15, Std : 2



- 데이터의 평균이 클 수록, Parameter의 하습 보폭이 넓어짐 (Learning rate가 커지는 것과 같은 효과)
- 따라서, |평균|이 큰 데이터 학습 시 Learning rate를 보다 작은 단위로 설정할 필요가 있음

[Code Link]

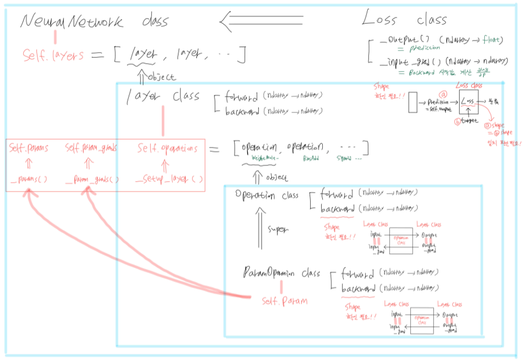
(11) MLP Implementation (from scratch)

Multi-Layer Perceptron 구현 (with Numpy)

[Reference]

1) Deep Learning from Scratch: Building with Pytorch from First Principles(Seth Weidman) (https://www.amazon.com/Deep-Learning-Scratch-Building-Principles/dp/1492041416)
2) Standalone Deep Learning(Idea Factory KAIST) (https://www.youtube.com/playlist?list=PLSAJwo7mw8jn8iaXwT4MqLbZnS-LJwnBd)

Code from scratch




```
class Operation:
    '''
    신경망 모델의 연산의 최상위 Abstracdt class
    '''
    def __init__(self):
        pass

    def forward(self, input_: np.ndarray, inference: bool=False)->np.ndarray:
        self.input_ = input_
        self.output = self._output(inference)
        return self.output

    def backward(self, output_grad: np.ndarray)->np.ndarray:
        '''
        self._input_grad() 호출함
        이때, 모양의 일치여부 확인 필요
        '''

        assert_same_shape(self.output, output_grad)

        self.input_grad = self._input_grad(output_grad)
        assert_same_shape(self.input_, self.input_grad)

        return self.input_grad

    def _output(self, inference: bool)->np.ndarray:
        '''
        Operation클래스의 concrete class(Subclasses of ParamOperation class)에서
        _output 메서드 구현해야 함
        '''
        raise NotImplementedError()

    def _input_grad(self, output_grad: np.ndarray)->np.ndarray:
        '''
        Operation을 구현한 모든 구상 클래스는 _input_grad 메서드를 구현해야 한다.
        '''
        raise NotImplementedError()
```

모듈화 “jskim_DNN”

- activations.py
- base.py
- dense.py
- dropout.py
- layers.py
- losses.py
- network.py
- optimizers.py
- train.py

```
import jskim_DNN
from jskim_DNN.layers import Dense
from jskim_DNN.losses import SoftmaxCrossEntropy, MeanSquaredError
from jskim_DNN.optimizers import Optimizer, SGD, SGDMomentum
from jskim_DNN.activations import Sigmoid, Tanh, Linear, ReLU
from jskim_DNN.network import NeuralNetwork
from jskim_DNN.train import Trainer
from jskim_DNN.utils.np_utils import softmax
from jskim_DNN.utils import mnist
```

MNIST_digit_image classification

Accuracy : 96.28%

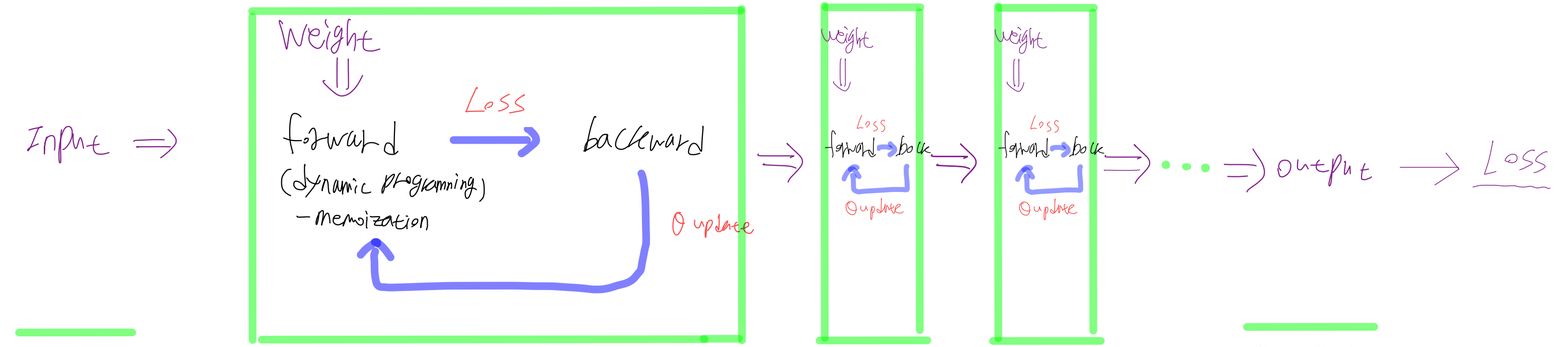
```
model = NeuralNetwork(
    layers=[Dense(neurons=89,
                  activation=Tanh(),
                  weight_init="glorot"),
            Dense(neurons=10,
                  activation=Linear(),
                  weight_init="glorot")],
            loss = SoftmaxCrossEntropy(),
seed=20190119)

optimizer = SGDMomentum(0.15, momentum=0.9, final_lr = 0.05, decay_type='linear')

trainer = Trainer(model, optimizer)
trainer.fit(X_train, train_labels, X_test, test_labels,
       epochs = 50,
       eval_every = 10,
       seed=20190119,
           batch_size=60,
           early_stopping=True)

calc_accuracy_model(model, X_test)
```

```
10에폭에서 검증 데이터에 대한 손실값: 0.373
20에폭에서 검증 데이터에 대한 손실값: 0.289
30에폭에서 검증 데이터에 대한 손실값: 0.271
40에폭에서 검증 데이터에 대한 손실값: 0.269

50에폭에서 손실값이 증가했다. 마지막으로 측정한 손실값은 40 에폭까지 학습된 모델에서 계산된 0.269이다.
모델 검증을 위한 정확도: 96.28%
```

[Code_Link]

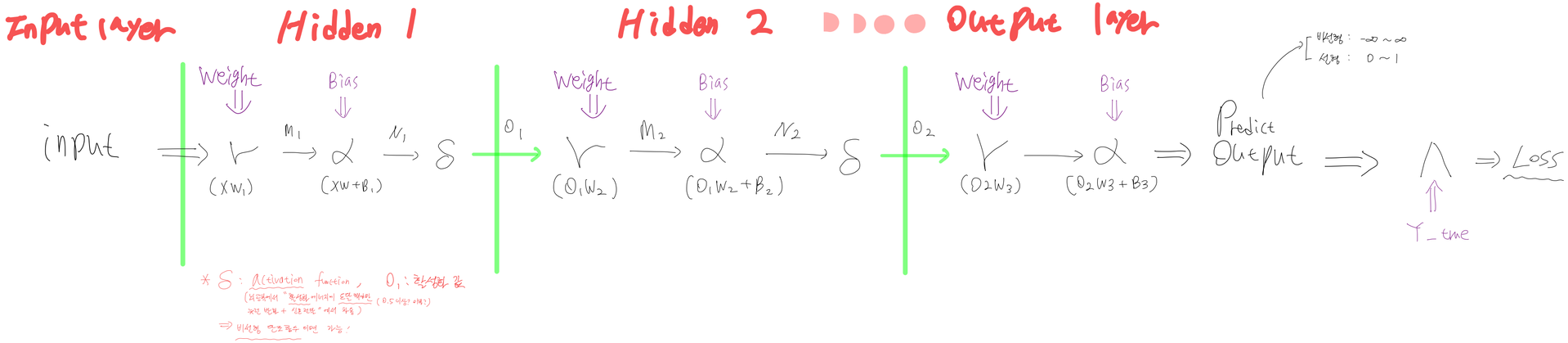
(11) MLP Implementation (from scratch)

[Deep Learning from scratch] * layer 정의 : 선형 → 비선형 (비선형 연산으로 끝나는 구조)



- "선형 → 비선형" ⟹ 1. 비 선형적 관계 (feature – 데이터) 학습
  2. layer 반복으로 새로 조합된 feature – 데이터 간 관계 학습

- 비선형 연산 X ⟹ 출력값 크기 $-\infty \sim \infty$
  〃 O ⟹ $0 \sim 1$



* δ : Activation function, O_1 : 활성화 값
(보통에서 "비선형 레이어의 모든 벡터는 양의 방향 + 상호적응" 여러 차원)
⇒ 비선형 연산조합수 이면 가능 :

[Code Link]

(11) MLP Implementation (from scratch)

신경망 구성요소 / : Operation

[코드]

- ① Abstract → ② Concrete
  (추상화 선언) (구현) (input 관련)

① Abstract
1) 구조 operation class

2) 시그니처 input ⇄ output (forward / backward)

- forward ( )
- backward ( )
- _output ( ) — abstract
- _input_grad ( ) — abstract

_input, output, input_grad

* assert same shape ?
  - input shape == input_grad shape
  - output shape == output_grad shape

parameter 관련

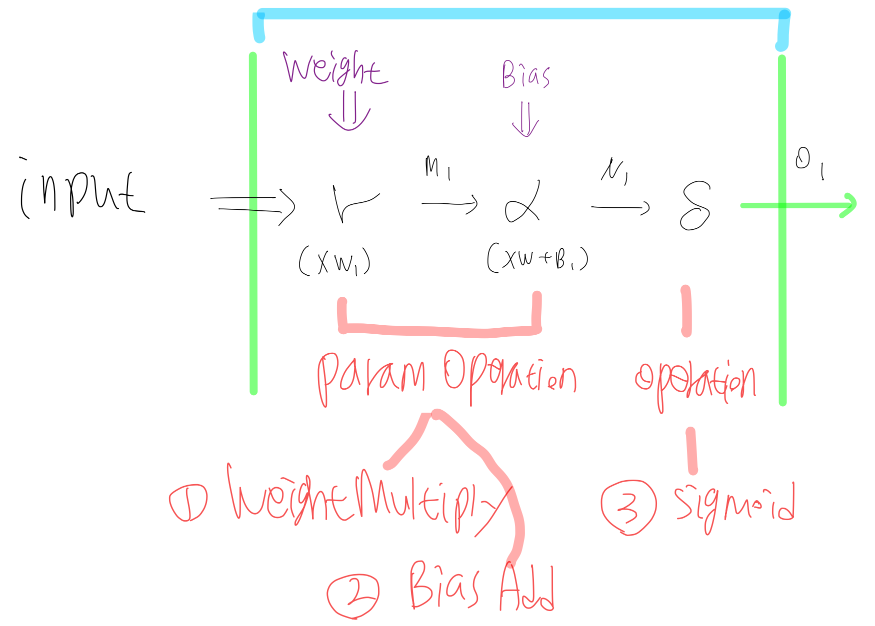
1) 구조 param Operation class (operation class)

2) 시그니처 input ⇄ output, ↓↑ weight

- forward ( ) ⇐ operation 상속
- backward ( ) ⇐ 〃 + method overriding
- _param_grad ( ) — abstract
- _input_grad ( ) — abstract

* assert same shape ?
  - input shape == input_grad shape
  - output shape == output_grad shape

[class operation 의 구조]



③ class Sigmoid (Operation)

weight, Bias, input → X W ⇄ +B₁ ⇄ σ( ) ⇄

$\frac{\partial \sigma(XW+B_1)}{\partial B_1}$ = Sigmoid(○)(1−Sig(○))

- __init__( ) ⇐ super( ).__init__( )
- _Output( ) ⇒ Sigmoid (_input)
- _input_grad ( ) ⇒ Output_grad ⊙ [Sigmoid(○)(1−Sig(○))]

② Concrete

operation의 Concrete
1. XW_1
2. $XW_1 + B_1$
3. $\sigma(XW_1 + B_1)$

① class WeightMultiply (param operation)

weight, input → X W ⇄ (ⓐ, ⓑ)

- __init__( ) ⇐ super( ).__init__(W)
- _output ( ) ⇒ XW
- _input_grad (ⓐ) ⇒ dot(output_grad, $\frac{\partial XW}{\partial X}$) ($W^T$)
- _param_grad (ⓑ) ⇒ dot(output_grad, $\frac{\partial XW}{\partial X}$) ⇒ $\frac{\partial XW}{\partial X}$. output_grad (X^T)
  연산 시 행렬의 형태는 'X^T'로 순서 변환.

② class BiasAdd (param operation)

Weight, Bias, input → X W ⇄ +B₁ ⇄

$\frac{\partial (XW_1+B_1)}{\partial XW_1} = I_{XW_1}$ ⓐ, $\frac{\partial (XW_1+B_1)}{\partial B_1} = I_{B_1}$ ⓑ

- __init__ ( ) ⇐ super( ).__init__(B)
  * assert B.shape[0] == 1
  (B = [—] 형태여야 함)
- _output ( ) ⇒ input + Self.param
  (Scalar) ([—])
- _input_grad (ⓐ) ⇒ Output_grad ⊙ $\frac{\partial (XW_1+B_1)}{\partial XW_1}$ ($I_{XW_1} = I_{input}$)
- _param_grad (ⓑ) ⇒ output_grad ⊙ $\frac{\partial (XW_1+B_1)}{\partial B_1}$ ($I_{B_1} = I_{param}$) ⇒ np.sum(___ , axis=0).reshape(1, B[1])
  "[param operation 의 BiasAdd의 backward return ( ) 이해]" 참고

[Code Link]

(11) MLP Implementation (from scratch)

신경망 구성요소 2 : layer

Abstract class

① class Layer(object):

init 공간생성 — 1) layer 제반, 연산 정보 공간 init (생성) : "neurons" (= 뉴런의 갯수), 첫 층 (0 layer) 여부, 연산 정보 저장 (params [ ], param_grads [ ]), 연산 순서 저장 (operations [ ])

_setup_layer (= 공간 채우라는 trigger) — 2) 1)의 공간 채우는 Abstract function
* Concrete class 에서 위의 공간 채우는 코드 구현.

forward() — 3) 1), 2)로 시행된 layer들은 순서대로 forward 하는 함수

(1) 첫 레이어? (0 layer?) ⇒ setup_layer() ; operations, weights 초기값 채워짐

(2) forward (operation 객체의 forward function)
self._input = input_*
for operations
input_ = operations의 원소.forward(input_)

*self._input은 forward 연산에서 사용되기 때문에, 초기값은 input_으로 저장해 준다. (self._input은 forward(input_)로 통해 계속 갱신된다.)

최종 모델에서 이 return 값이 prediction 이 됨

(3) forward 결과 return
* 갱신된 input_은 결국 output이 되어 return 된다. (forward 연산 결과)

backward() — 4) 1), 2)로 시행된 layer들은 역순으로 forward 하는 함수

(1) assert_same_shape

(2) backward (operation 객체 내 backward function)
for reversed(operations) [W, B, 6, ...]
output_grad = reversed된 operations의 원소.backward(output_grad)

최종 input_grad return — (3) backward 결과 return
* 갱신된 output_grad는 결국 input_grad가 되어 return 된다.

최종 param_grad 호출 — (4) __param_grads() 로 operation 객체 내 모든 param_grad 호출 (param_grads 내 저장됨)
ParamOperation 내 backward function에서 그때 그때 param_grad 연산됨
*개별 연산 코드는 Concrete class (W, B)에서 구현

_param_grads() — 5) ParamOperation 클래스 객체에서 for 로 param_grad 값 호출 (param_grads 에 저장)

_params() — 6) // param 값 호출 (params 에 저장)

Layer의 sequence를 concrete implementation!

Concrete class

② class Dense(Layer)

1) layer sequence (Dense) 구성을 위한 공간 생성 — super().__init__(neurons)
= layer 구성을 위한 공간
+ activation function 입력 받아 저장.

2) _setup_layer Concrete implementation

(1) shuffle seed 지정 (W, B 무작위 추출 위함)
(2) W, B 초기값 생성 ⟹ operations 에 저장 "sequence" [W₁, B₁, 6,]
Shape은 ○○ 생각
+ 초기값은 normal distribution 에서.

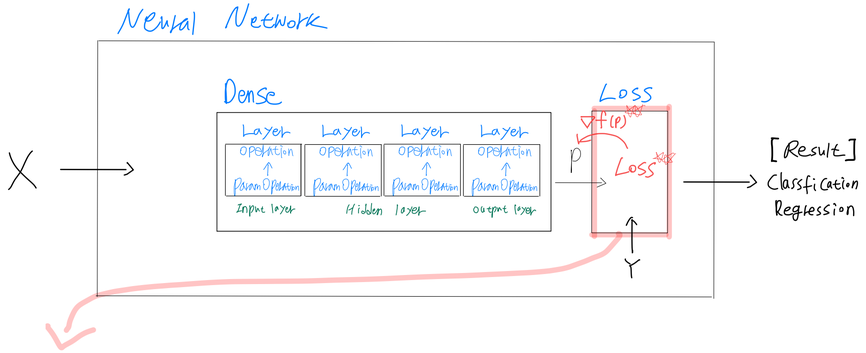
* W shape, B shape 주의

X → XW → XW + b → O₁ → O₁W
3,5 — 3,5 5,neurons — 3,neurons 1,neurons — 3,neurons — 3,neurons neurons,neurons

[Code Link]

(11) MLP Implementation (from scratch)

신경망 구성요소 3 : Loss



① class Loss :

1) 생성자 ⇐ pass

2) forward ★ Loss 값 (float) Return ! ★최종 output을 바탕으로 loss_val return

- parameter : prediction (ndarray) , target (ndarray) ★prediction과 target의 shape 일치여부 확인!

```
loss_val = self._output()
return loss_val
```

4) _output Abstract

★ Loss 구하는 function으로 Concrete 필요!

3) backward ★ Loss_gradient (ndarray) Return ! ★ 첫 gradient 계산해 최종 input_grad을 return 해 Output layer에 전달!

- parameter : X

```
self.input_grad = self._input_grad()
return self.input_grad
```

★prediction과 input_grad shape 일치여부 확인!

5) _input_grad() Abstract

★ _output (Loss 구하는 function) 바탕으로 gradient 계산하는 Concrete 필요!

② class MeanSquaredError (Loss) :

1) 생성자 ⇐ super() + pass

2) _output

$loss = MSE \left(\frac{1}{n} \Sigma (P-Y)^2 \right)$

return loss

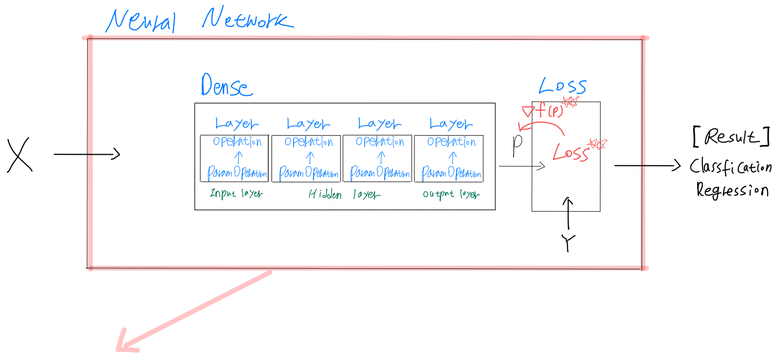
3) _input_grad

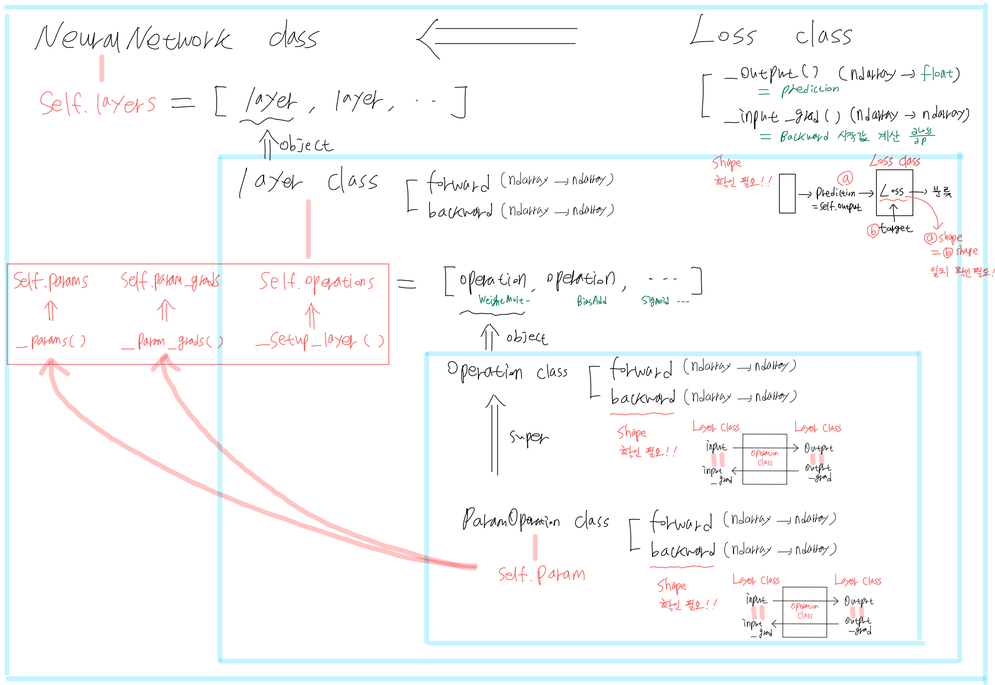
return $\frac{\partial MSE}{\partial P} = \frac{2}{n}(P-Y)$

[Code Link]

(11) MLP Implementation (from scratch)

신경망 구성요소 4 : Neural Network





① Class NeuralNetwork

1) 생성자 ⟹ layers : List[Layer], loss : Loss, seed : float = 1

NN ⇐ Loss
Layer

NN을 이루는 sequence를 받음

* seed 설정

2) forward

- parameter : X_batch

X_batch → NN

```
X_out = X_batch
for layer in self.layers:
    X_out = layer.forward(X_out)

return X_out
```

3) backward

- parameter : loss_grad

```
grad = loss_grad
for △ in ○
    grad = △(grad)
```

4) train_batch

- parameter : X_batch, y_batch

```
# forward
predictions = self.forward(X_batch)
loss = self.loss.forward(predictions, y_batch)

# backward
self.backward(self.loss.backward())
return loss
```

최초의 input_grad 계산 ← loss 값

5) params * NN의 parameter 값들을 받음

```
for layer in self.layers:
    yield from layer.params
```

6) param_grads * NN의 param_grad 값들을 받음

```
for layer in self.layers:
    yield from layer.param_grads
```

[Code Link]

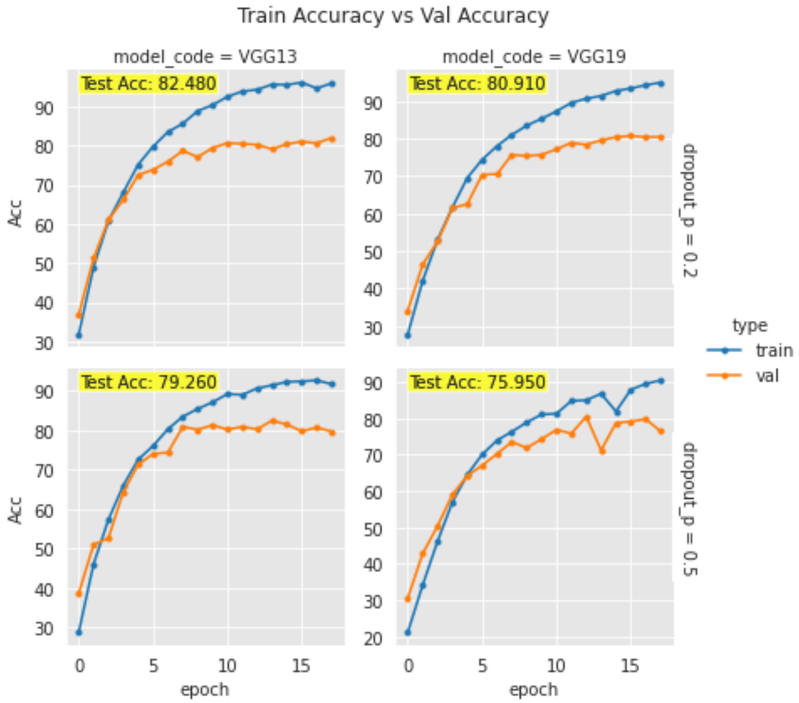
(12) CNN (VGG13 / 19 + CIFAR-10)

1. VGG 13/19 구조

```
# M : maxpooling 단계, int : out channel 숫자
# M 은 모두 5개 ==>> 전체 width = height = 32 니까, 5번의 maxpooling 통해 receptive field = 1 (32/2/2/2/2/2 = 1) 로 만들어, receptive field = 전체 image로 만들겠다.
# 최종적으로 뽑아낸 복잡한 feature들로 receptive field를 전체 이미지 범위로 놓고, 모든 이미지에 이 feature map을 만들어내겠다. 는 것.
cfg = {
    'VGG11': [64, 'M', 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, 'M'],
    'VGG13': [64, 64, 'M', 128, 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, 'M'],
    'VGG16': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 512, 512, 512, 'M', 512, 512, 512, 'M'],
    'VGG19': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 256, 'M', 512, 512, 512, 512, 'M', 512, 512, 512, 512, 'M'],
}
```

구조 참고 : (https://github.com/kuangliu/pytorch-cifar/blob/master/models/vgg.py)

2. Accuracy(CIFAR-10)



[Code_Link]

(13) U-Net Paper 구현

U-Net: Convolutional Networks for Biomedical Image Segmentation

Olaf Ronneberger, Philipp Fischer, and Thomas Brox

Computer Science Department and BIOSS Centre for Biological Signalling Studies,
University of Freiburg, Germany
ronneber@informatik.uni-freiburg.de,
WWW home page: http://lmb.informatik.uni-freiburg.de/

Abstract. There is large consent that successful training of deep networks requires many thousand annotated training samples. In this paper, we present a network and training strategy that relies on the strong use of data augmentation to use the available annotated samples more

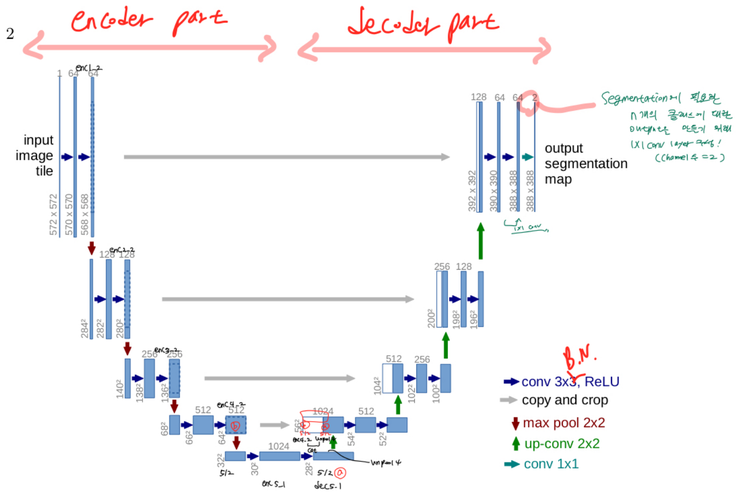


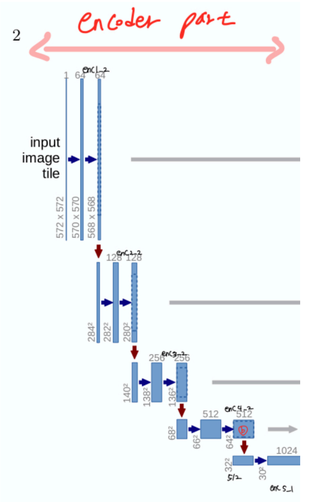
Fig. 1. U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

Layer 구조

[Conv - BatchNorm - ReLU]

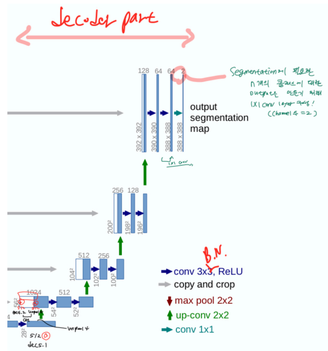
```
def CBR2d(in_channels, out_channels, kernel_size=3, padding=1, stride=1, bias=True):
    layers=[]
    layers += [nn.Conv2d(in_channels=in_channels, out_channels=out_channels,
                         kernel_size=kernel_size, padding=padding, stride=stride, bias=bias)]
    layers += [nn.BatchNorm2d(num_features=out_channels)]
    layers += [nn.ReLU()]

    cbr = nn.Sequential(*layers)

    return cbr
```

1. Encoder part




```
# Encoder part
# first stage
self.enc1_1 = CBR2d(in_channels=1, out_channels=64)
self.enc1_2 = CBR2d(in_channels=64, out_channels=64)

self.pool1 = nn.MaxPool2d(kernel_size=2)

# second stage
self.enc2_1 = CBR2d(in_channels=64, out_channels=128)
self.enc2_2 = CBR2d(in_channels=128, out_channels=128)

self.pool2 = nn.MaxPool2d(kernel_size=2)

# third stage
self.enc3_1 = CBR2d(in_channels=128, out_channels=256)
self.enc3_2 = CBR2d(in_channels=256, out_channels=256)

self.pool3 = nn.MaxPool2d(kernel_size=2)

# fourth stage
self.enc4_1 = CBR2d(in_channels=256, out_channels=512)
self.enc4_2 = CBR2d(in_channels=512, out_channels=512)

self.pool4 = nn.MaxPool2d(kernel_size=2)  # 32, 32, 512 output

# fifth stage
self.enc5_1 = CBR2d(in_channels=512, out_channels=1024) # encoder part 마무리
```

2. Decoder Part

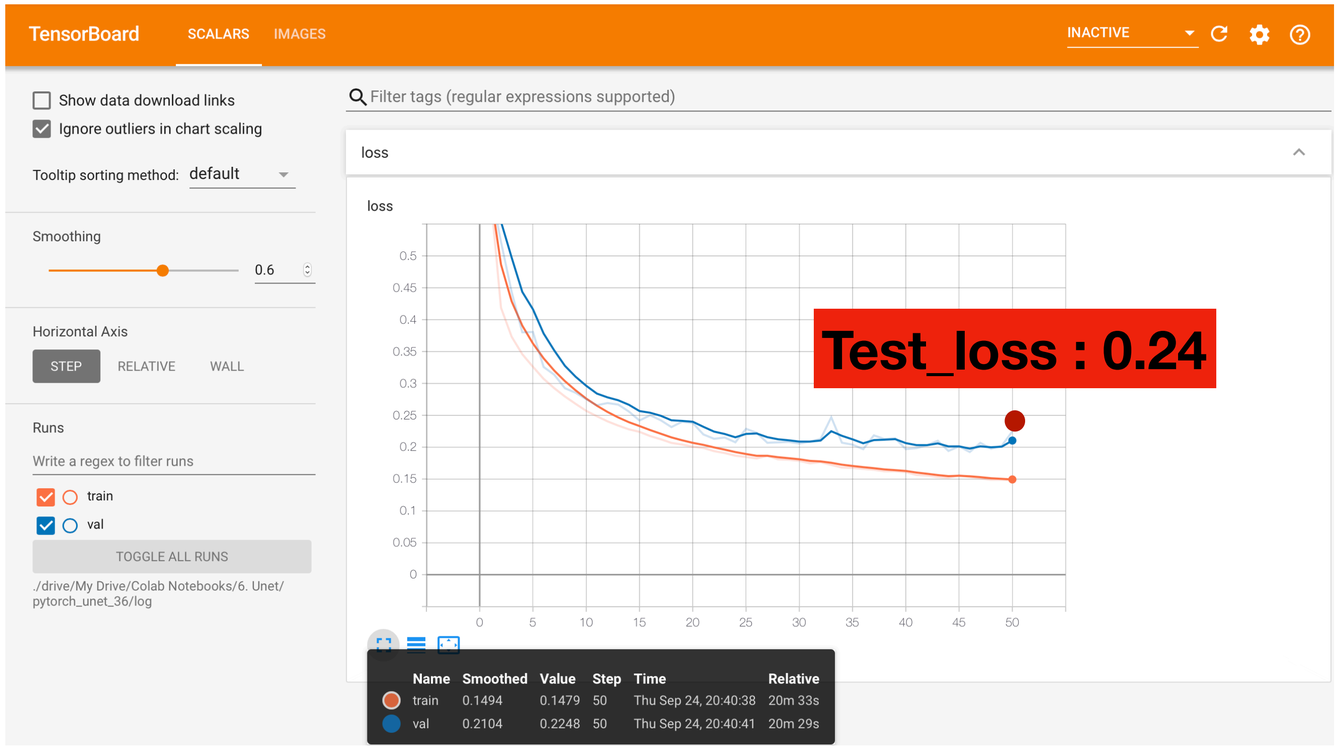


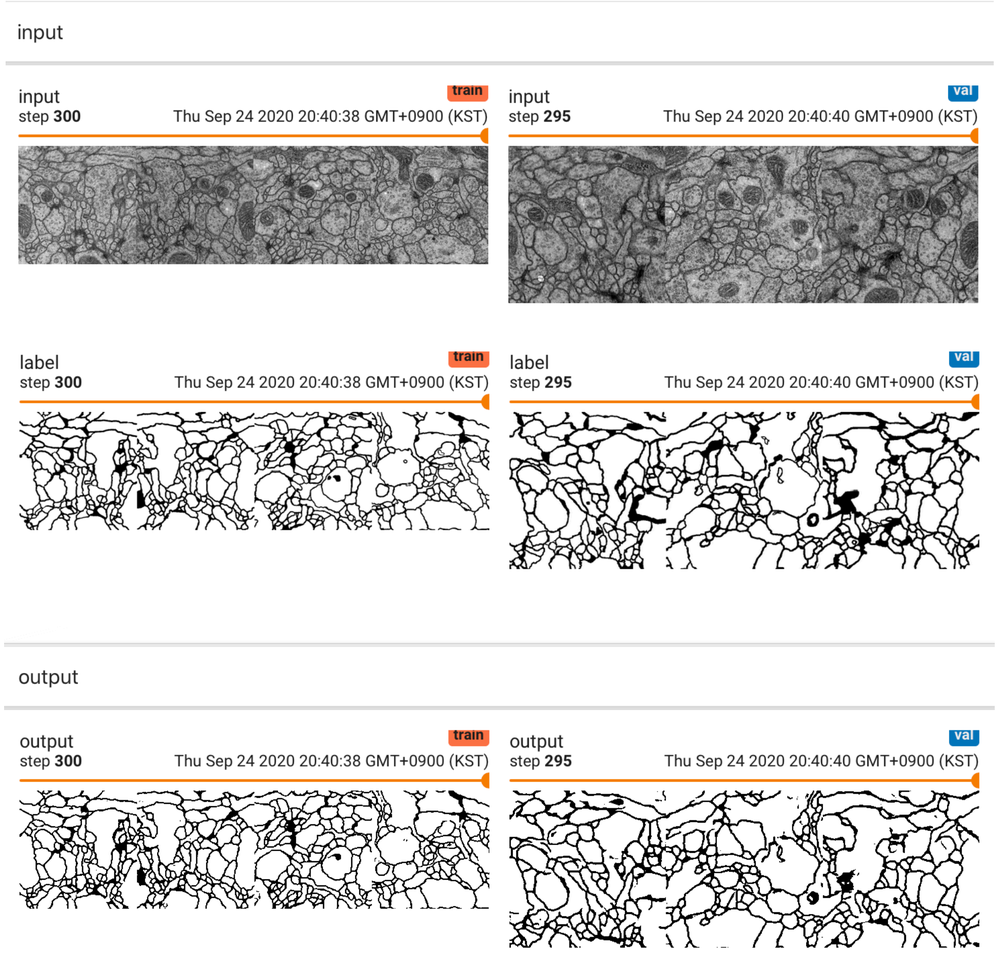

```
# Decoder part
# fifth stage
self.dec5_1 = CBR2d(in_channels=1024, out_channels=512)

self.unpool4 = nn.ConvTranspose2d(in_channels=512, out_channels=512,
                                  kernel_size=2, stride=2, padding=0, bias=True)
# fourth stage
self.dec4_2 = CBR2d(in_channels=2*512, out_channels=512) # skip-connection이 있으므로, in_channels 수 주의 필요
self.dec4_1 = CBR2d(in_channels=512, out_channels=256)

self.unpool3 = nn.ConvTranspose2d(in_channels=256, out_channels=256,
                                  kernel_size=2, stride=2, padding=0, bias=True)
# third stage
self.dec3_2 = CBR2d(in_channels=2*256, out_channels=256) # skip-connection이 있으므로, in_channels 수 주의 필요
self.dec3_1 = CBR2d(in_channels=256, out_channels=128)

self.unpool2 = nn.ConvTranspose2d(in_channels=128, out_channels=128,
                                  kernel_size=2, stride=2, padding=0, bias=True)
# second stage
self.dec2_2 = CBR2d(in_channels=2*128, out_channels=128) # skip-connection이 있으므로, in_channels 수 주의 필요
self.dec2_1 = CBR2d(in_channels=128, out_channels=64)

self.unpool1 = nn.ConvTranspose2d(in_channels=64, out_channels=64,
                                  kernel_size=2, stride=2, padding=0, bias=True)
# first stage
self.dec1_2 = CBR2d(in_channels=2*64, out_channels=64) # skip-connection이 있으므로, in_channels 수 주의 필요
self.dec1_1 = CBR2d(in_channels=64, out_channels=64)

# 1 x 1 conv layer (class output 산출 위함)
self.fc = nn.Conv2d(in_channels=64, out_channels=1, kernel_size=1, stride=1, padding=0, bias=True)
```

[Code Link]

(13) U-Net Paper 구현

3. Result





[Code_Link]

5. Paper Study

(14) Neural Ordinary Differential Equations (NeurIPS 2018, Ricky T. Q. Chen)

참고자료

1) Neural Ordinary Differential Equations (NeurIPS 2018, Ricky T. Q. Chen)
2) Augmented Neural ODEs (NeurIPS 2019, Emilien Dupont)
3) “Neural ODEs: breakdown of another deep learning breakthrough” (https://towardsdatascience.com/neural-odes-breakdown-of-another-deep-learning-breakthrough-3e78c7213795)
4) “Neural Ordinary Differential Equations” (https://msurtsukov.github.io/Neural-ODE/

(15) Grad CAM : Visual Explanation from Deep Networks via Gradient based Localization (ICCV 2017, RR Selvaraju)

참고자료

1) “Grad CAM : Visual Explanation from Deep Networks via Gradient based Localization” (ICCV 2017, RR Selvaraju)
2) “Learning Deep Features for Discriminative Localization” (CVPR 2016, Bolei Zhou)
3) 천우진 - CNN Localization 2 (CAM, grad CAM, PDA) (Youtube - “KoreaUniv DSBA” channel) (https://www.youtube.com/watch?v=aGIEVeaKLgY&t=3271s)
4) “Grad-CAM” (https://www.secmem.org/blog/2020/01/17/gradcam/)

(14) Paper_study : “Neural Ordinary Differential Equations(NeurIPS 2018)”

Neural Ordinary Differential Equations

Ricky T. Q. Chen*, Yulia Rubanova*, Jesse Bettencourt*, David Duvenaud
University of Toronto, Vector Institute
{rtqichen, rubanova, jessebett, duvenaud}@cs.toronto.edu

Abstract

We introduce a new family of deep neural network models. Instead of specifying a discrete sequence of hidden layers, we parameterize the derivative of the hidden state using a neural network. The output of the network is computed using a black-box differential equation solver. These continuous-depth models have constant memory cost, adapt their evaluation strategy to each input, and can explicitly trade numerical precision for speed. We demonstrate these properties in continuous-depth residual networks and continuous-time latent variable models. We also construct continuous normalizing flows, a generative model that can train by maximum likelihood, without partitioning or ordering the data dimensions. For training, we show how to scalably backpropagate through any ODE solver, without access to its internal operations. This allows end-to-end training of ODEs within larger models.

1 Introduction

Models such as residual networks, recurrent neural network decoders, and normalizing flows build complicated transformations by composing a sequence of transformations to a hidden state:

$$\mathbf{h}_{t+1} = \mathbf{h}_t + f(\mathbf{h}_t, \theta_t) \quad (1)$$

where $t \in \{0 \dots T\}$ and $\mathbf{h}_t \in \mathbb{R}^D$. These iterative updates can be seen as an Euler discretization of a continuous transformation (Lu et al., 2017; Haber and Ruthotto, 2017; Ruthotto and Haber, 2018).

What happens as we add more layers and take smaller steps? In the limit, we parameterize the continuous dynamics of hidden units using an ordinary differential equation (ODE) specified by a neural network:

$$\frac{d\mathbf{h}(t)}{dt} = f(\mathbf{h}(t), t, \theta) \quad (2)$$

Starting from the input layer $\mathbf{h}(0)$, we can define the output layer $\mathbf{h}(T)$ to be the solution to this ODE initial value problem at some time T. This value can be computed by a black-box differential equation solver, which evaluates the hidden unit dynamics f wherever necessary to determine the solution with the desired accuracy. Figure 1 contrasts these two approaches.

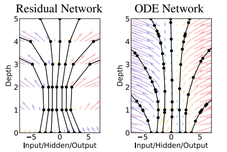


Figure 1: *Left:* A Residual network defines a discrete sequence of finite transformations. *Right:* A ODE network defines a vector field, which continuously transforms the state. *Both:* Circles represent evaluation locations.

Defining and evaluating models using ODE solvers has several benefits:

Memory efficiency In Section 2, we show how to compute gradients of a scalar-valued loss with respect to all inputs of any ODE solver, *without backpropagating through the operations of the solver*. Not storing any intermediate quantities of the forward pass allows us to train our models with constant memory cost as a function of depth, a major bottleneck of training deep models.

32nd Conference on Neural Information Processing Systems (NeurIPS 2018), Montréal, Canada.

Adaptive computation Euler's method is perhaps the simplest method for solving ODEs. There have since been more than 120 years of development of efficient and accurate ODE solvers (Runge, 1895; Kutta, 1901; Hairer et al., 1987). Modern ODE solvers provide guarantees about the growth of approximation error, monitor the level of error, and adapt their evaluation strategy on the fly to achieve the requested level of accuracy. This allows the cost of evaluating a model to scale with problem complexity. After training, accuracy can be reduced for real-time or low-power applications.

Scalable and invertible normalizing flows An unexpected side-benefit of continuous transformations is that the change of variables formula becomes easier to compute. In Section 4, we derive this result and use it to construct a new class of invertible density models that avoids the single-unit bottleneck of normalizing flows, and can be trained directly by maximum likelihood.

Continuous time-series models Unlike recurrent neural networks, which require discretizing observation and emission intervals, continuously-defined dynamics can naturally incorporate data which arrives at arbitrary times. In Section 5, we construct and demonstrate such a model.

2 Reverse-mode automatic differentiation of ODE solutions

The main technical difficulty in training continuous-depth networks is performing reverse-mode differentiation (also known as backpropagation) through the ODE solver. Differentiating through the operations of the forward pass is straightforward, but incurs a high memory cost and introduces additional numerical error.

We treat the ODE solver as a black box, and compute gradients using the *adjoint sensitivity method* (Pontryagin et al., 1962). This approach computes gradients by solving a second, augmented ODE backwards in time, and is applicable to all ODE solvers. This approach scales linearly with problem size, has low memory cost, and explicitly controls numerical error.

Consider optimizing a scalar-valued loss function $L()$, whose input is the result of an ODE solver:

$$L(\mathbf{z}(t_1)) = L\left(\mathbf{z}(t_0) + \int_{t_0}^{t_1} f(\mathbf{z}(t), t, \theta)dt\right) = L\left(\text{ODESolve}(\mathbf{z}(t_0), f, t_0, t_1, \theta)\right) \quad (3)$$

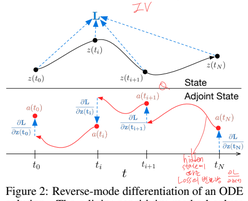


Figure 2: Reverse-mode differentiation of an ODE solution. The adjoint sensitivity method solves an augmented ODE backwards in time. The augmented system contains both the original state and the sensitivity of the loss with respect to the state. If the loss depends directly on the state at multiple observation times, the adjoint state must be updated in the direction of the partial derivative of the loss with respect to each observation.

To optimize L, we require gradients with respect to θ. The first step is to determining how the gradient of the loss depends on the hidden state $\mathbf{z}(t)$ at each instant. This quantity is called the *adjoint* $\mathbf{a}(t) = \partial L / \partial \mathbf{z}(t)$. Its dynamics are given by another ODE, which can be thought of as the instantaneous analog of the chain rule:

$$\frac{d\mathbf{a}(t)}{dt} = -\mathbf{a}(t)^T \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}} \quad (4)$$

We can compute $\partial L / \partial \mathbf{z}(t_0)$ by another call to an ODE solver. This solver must run backwards, starting from the initial value of $\partial L / \partial \mathbf{z}(t_1)$. One complication is that solving this ODE requires the knowing value of $\mathbf{z}(t)$ along its entire trajectory. However, we can simply recompute $\mathbf{z}(t)$ backwards in time together with the adjoint, starting from its final value $\mathbf{z}(t_1)$.

Computing the gradients with respect to the parameters θ requires evaluating a third integral, which depends on both $\mathbf{z}(t)$ and $\mathbf{a}(t)$:

$$\frac{dL}{d\theta} = -\int_{t_1}^{t_0} \mathbf{a}(t)^T \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \theta} dt \quad (5)$$

The vector-Jacobian products $\mathbf{a}(t)^T \frac{\partial f}{\partial \mathbf{z}}$ and $\mathbf{a}(t)^T \frac{\partial f}{\partial \theta}$ in (4) and (5) can be efficiently evaluated by automatic differentiation, at a time cost similar to that of evaluating f. All integrals for solving $\mathbf{z}$, $\mathbf{a}$ and $\frac{\partial L}{\partial \theta}$ can be computed in a single call to an ODE solver, which concatenates the original state, the adjoint, and the other partial derivatives into a single vector. Algorithm 1 shows how to construct the necessary dynamics, and call an ODE solver to compute all gradients at once.

Algorithm 1 Reverse-mode derivative of an ODE initial value problem

Input: dynamics parameters θ, start time t_0, stop time t_1, final state $\mathbf{z}(t_1)$, loss gradient $\partial L / \partial \mathbf{z}(t_1)$

$s_0 = [\mathbf{z}(t_1), \frac{\partial L}{\partial \mathbf{z}(t_1)}, \mathbf{0}_{|\theta|}]$ ▷ Define initial augmented state

def aug_dynamics$([\mathbf{z}(t), \mathbf{a}(t), \cdot], t, \theta)$: ▷ Define dynamics on augmented state

return $[f(\mathbf{z}(t), t, \theta), -\mathbf{a}(t)^T \frac{\partial f}{\partial \mathbf{z}}, -\mathbf{a}(t)^T \frac{\partial f}{\partial \theta}]$ ▷ Compute vector-Jacobian products

$[\mathbf{z}(t_0), \frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}] = \text{ODESolve}(s_0, \text{aug\_dynamics}, t_1, t_0, \theta)$ ▷ Solve reverse-time ODE

return $\frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}$ ▷ Return gradients

Most ODE solvers have the option to output the state $\mathbf{z}(t)$ at multiple times. When the loss depends on these intermediate states, the reverse-mode derivative must be broken into a sequence of separate solves, one between each consecutive pair of output times (Figure 2). At each observation, the adjoint must be adjusted in the direction of the corresponding partial derivative $\partial L / \partial \mathbf{z}(t_i)$.

The results above extend those of Stapor et al. (2018, section 2.4.2). An extended version of Algorithm 1 including derivatives w.r.t. t_0 and t_1 can be found in Appendix C. Detailed derivations are provided in Appendix B. Appendix D provides Python code which computes all derivatives for `scipy.integrate.odeint` by extending the `autograd` automatic differentiation package. This code also supports all higher-order derivatives. We have since released a PyTorch (Paszke et al., 2017) implementation, including GPU-based implementations of several standard ODE solvers at `github.com/rtqichen/torchdiffeq`.

3 Replacing residual networks with ODEs for supervised learning

In this section, we experimentally investigate the training of neural ODEs for supervised learning.

Software To solve ODE initial value problems numerically, we use the implicit Adams method implemented in LSODE and VODE and interfaced through the `scipy.integrate` package. Being an implicit method, it has better guarantees than explicit methods such as Runge-Kutta but requires solving a nonlinear optimization problem at every step. This setup makes direct backpropagation through the integrator difficult. We implement the adjoint sensitivity method in Python's `autograd` framework (Maclaurin et al., 2015). For the experiments in this section, we evaluated the hidden state dynamics and their derivatives on the GPU using Tensorflow, which were then called from the Fortran ODE solvers, which were called from Python `autograd` code.

Model Architectures We experiment with a small residual network which downsamples the input twice then applies 6 standard residual blocks He et al. (2016b), which are replaced by an ODESolve module in the ODE-Net variant. We also test a network with the same architecture but where gradients are backpropagated directly through a Runge-Kutta integrator, referred to as RK-Net. Table 1 shows test error, number of parameters, and memory cost. L denotes the number of layers in the ResNet, and $\tilde{L}$ is the number of function evaluations that the ODE solver requests in a single forward pass, which can be interpreted as an implicit number of layers. We find that ODE-Nets and RK-Nets can achieve around the same performance as the ResNet.

Table 1: Performance on MNIST. †From LeCun et al. (1998).

	Test Error	# Params	Memory	Time
1-Layer MLP†	1.60%	0.24 M	-	-
ResNet	0.41%	0.60 M	$\mathcal{O}(L)$	$\mathcal{O}(L)$
RK-Net	0.47%	0.22 M	$\mathcal{O}(L)$	$\mathcal{O}(\tilde{L})$
ODE-Net	0.42%	0.22 M	$\mathcal{O}(1)$	$\mathcal{O}(\tilde{L})$

Error Control in ODE-Nets ODE solvers can approximately ensure that the output is within a given tolerance of the true solution. Changing this tolerance changes the behavior of the network. We first verify that error can indeed be controlled in Figure 3a. The time spent by the forward call is proportional to the number of function evaluations (Figure 3b), so tuning the tolerance gives us a

- **2018 NeurIPS “Best Paper” 수상**
- **2018년 최초 제안된 후, 현재까지 다양한 후속 연구가 진행되고 있음**
- **본 논문에서 제안된 최초 모델은 Data의 dimension 경직성 등의 여러 한계가 있음**
- **그럼에도 불구하고, 기존의 이산적인(discrete) layer 구조에서 벗어난 새로운 Neural Network를 제시함으로써 다양한 발전가능성을 열어준 논문으로 평가됨**

(14) Paper_study : “Neural Ordinary Differential Equations(NeurIPS 2018)”

1. ‘Neural ODE’ 란 무엇인가?

Neural ODE는 Neural Network의 sequence of hidden states를 “continuous transformation”으로 본다. 그리고 이 “continuous transformation”을 systems of ODEs 로 설명한다.

2. ‘Neural ODE’는 왜 이러한 Network를 설계하고, ODE로 풀어냈을까?

ODE Solver(특히, Euler’s method)를 활용해 forward, backward process를 아주 간단히 풀어낼 수 있기 때문이다. 이는 결국 Backward pass에 대한 새로운 시각을 시각을 제시하기 위함이다.

첫째로, hidden states의 discrete 구조가 깊어짐에 따른 memory cost의 상승, accuracy의 감소에 대처할 수 있다.
둘째로, continuous sequential data를 보다 잘 학습할 수 있다. discretization을 위한 sequential data의 resampling(interpolation)등을 거치지 않는다.
셋째로, 전통적이면서 간결한 ODE solver(Euler’s method)를 활용하면, 실시간 혹은 computational environment가 다소 제한되는 상황에서도 정확한 학습이 가능하다.

기존의 residual network도 hidden states간의 transformation 과정을 residual(변화량)의 학습으로 보아, Euler’s method로 풀어낸 것으로 이해할 수 있다. 하지만, 이는 학습과정을 discretization 했다는 점에서 Neural ODE와는 다르다.

3. ‘Neural ODE’에서 가장 새로운 부분은 무엇인가?

Hidden states는 transformation을 Euler Method로 수치해석적으로 바라본 것은 새로운 것이 아니다. 이미 Residual Network에서 보여준 바 있다.
이 “transformation의 sequence를 continuous하게” 바라봤다는 점과 “Adjoint state method로 backward과정을 ODE solver로 forward pass와 ‘동일한 과정’으로 풀어냈다는 점”이 새로운 것이다.

Input과 parameter에 대한 gradient를 구하는 것은 Neural Network model의 numerical optimization에는 필수적인 과정이다. 물론, Euler’s method에 기반한 forward 과정은 무수한 “+” 연산으로 볼볼 수 있기 때문에, 기존의 chain-rule에 근거한 back-propagation으로도 충분히 학습은 가능하다. 따라서, “Adjoint state method”로 새롭게 backward과정을 디자인 하는 것이 꼭 필요한 것인가에 대한 의문을 가질 수 있다.

하지만, 기존의 chain-rule에 근거한 back-propagation은 hidden layer가 깊어질 수록, memory cost와 error 발생을 상대적으로 피하기 어렵다는 단점을 갖는다. 심지어 논문의 실험 결과, 새로운 backward 과정인 “Adjoint state method”를 활용해 학습하면, computational efficiency도 얻을 수 있음을 확인할 수 있다.*

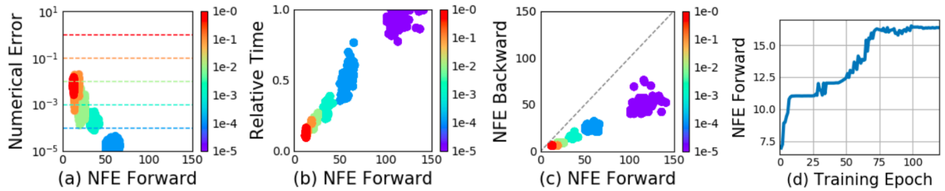


Figure 3: Statistics of a trained ODE-Net. (NFE = number of function evaluations.)

*Figure 3의 (c)를 보면, backward시, # of functions evaluations가 ODE-Net(Neural ODE)이 절반에 가까운 것을 확인할 수 있다.

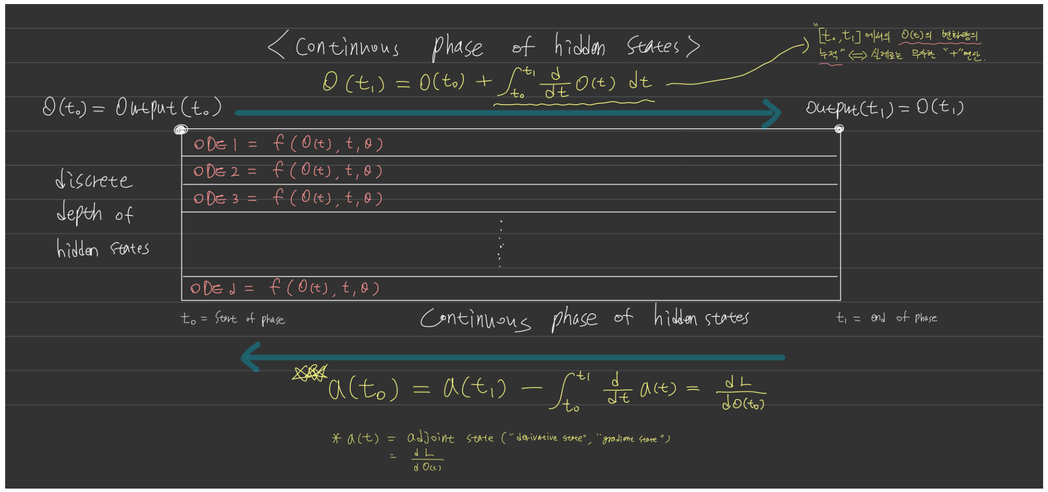
(14) Paper_study : “Neural Ordinary Differential Equations(NeurIPS 2018)”

4. ‘2’의 ‘memory cost 의 감소’는 어떻게 가능한가?

핵심은 "기존의 Backpropagation 없이”,
Loss에 대한 Input과 parameter의 gradient를 구하는 데 있다.

Neural ODE로 학습하면, 전통적인 forward / backpropagation에서 이루어지는 중간 매개변수, gradients들의 “기억”이 필요하지 않다.

Neural ODE의 forward pass, reverse-mode(“backprop”) 는 아래의 그림과 같다.



5. Follow-up study

“Augmented Neural ODEs(NeurIPS 2019)”

- 2018년 제안된 ‘Neural ODE’ 의 Network는 homeomorphism(위동형사상, 전단사 mapping구조) 이기에, input space의 topology를 그대로 유지하는 특징을 갖는다.
- 따라서, 아래와 같은 동심원 구조를 나타내는 function을 표현할 수 없는 한계를 갖는다.

4 Functions Neural ODEs cannot represent

We now introduce classes of functions in arbitrary dimension d which NODEs cannot represent. Let $0 < r_1 < r_2 < r_3$ and let $g : \mathbb{R}^d \to \mathbb{R}$ be a function such that

$$\begin{cases} g(\mathbf{x}) = -1 & \text{if } \|\mathbf{x}\| \le r_1 \\ g(\mathbf{x}) = 1 & \text{if } r_2 \le \|\mathbf{x}\| \le r_3, \end{cases}$$

where $\|\cdot\|$ is the Euclidean norm. An illustration of this function for $d = 2$ is shown in Fig. 4. The function maps all points inside the blue sphere to -1 and all points in the red annulus to 1.

Proposition 2. *Neural ODEs cannot represent* $g(\mathbf{x})$.

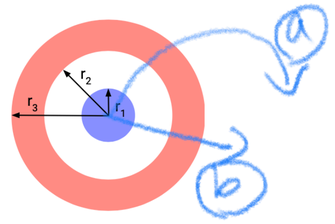


Figure 4: Diagram of $g(\mathbf{x})$ for $d = 2$.

- ‘Neural ODE’는 ‘b’와 같은 mapping만 가능하기에, 위의 함수(g(x))를 표현할 수 없다.

따라서, 본 연구에서 Augmented Neural ODE로 g(x)를 표현할 수 있는 개선된 모델을 제안했다.

5 Augmented Neural ODEs

Motivated by our theory and experiments, we introduce Augmented Neural ODEs (ANODEs) which provide a simple solution to the problems we have discussed. We augment the space on which we learn and solve the ODE from $\mathbb{R}^d$ to $\mathbb{R}^{d+p}$, allowing the ODE flow to lift points into the additional dimensions to avoid trajectories intersecting each other. Letting $\mathbf{a}(t) \in \mathbb{R}^p$ denote a point in the augmented part of the space, we can formulate the augmented ODE problem as

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{bmatrix}\mathbf{h}(t)\\ \mathbf{a}(t)\end{bmatrix} = \mathbf{f}\left(\begin{bmatrix}\mathbf{h}(t)\\ \mathbf{a}(t)\end{bmatrix}, t\right), \qquad \begin{bmatrix}\mathbf{h}(0)\\ \mathbf{a}(0)\end{bmatrix} = \begin{bmatrix}\mathbf{x}\\ \mathbf{0}\end{bmatrix}$$

i.e. we concatenate every data point $\mathbf{x}$ with a vector of zeros and solve the ODE on this augmented space. We hypothesize that this will also make the learned (augmented) $\mathbf{f}$ smoother, giving rise to simpler flows that the ODE solver can compute in fewer steps. In the following sections, we verify this behavior experimentally and show both on toy and image datasets that ANODEs achieve lower losses, better generalization and lower computational cost than regular NODEs.

- Augmented Neural ODE는 기존의 hidden state에 a(t)라는 가상의 새로운 input space를 더해 ODE를 더 넓은 차원에서 학습할 수 있도록 하였다. 이를 통해 보다 공간적으로 복잡한 function을 표현할 수 있다.
- Augmented Neural ODE는 기존의 Neural ODE보다 안정적이고, computational cost, loss가 낮은 결과를 보였다.

(15) Paper_study : “Grad CAM : Visual Explanations from Deep networks via Gradient based Localization”

1. ‘Grad-CAM’이란 무엇인가?

“Gradient-weighted CAM”의 줄임말이다. CNN구조 아래, 각 feature map의 gradient를 활용해 input data의 localization을 위한 CAM을 얻는 방법이다.
최초에 제안된 CAM에 비해, GAP에 의존하지 않아, CAM을 얻기에 구조적으로 보다 유연한 것이 특징이다. 더불어, Grad-CAM 논문에서는 Grad-CAM 방식에 “Guided back-propagation”을 적용한 Guided Grad-CAM을 제안해 CAM 결과물의 visualization을 위한 제언도 하였다.

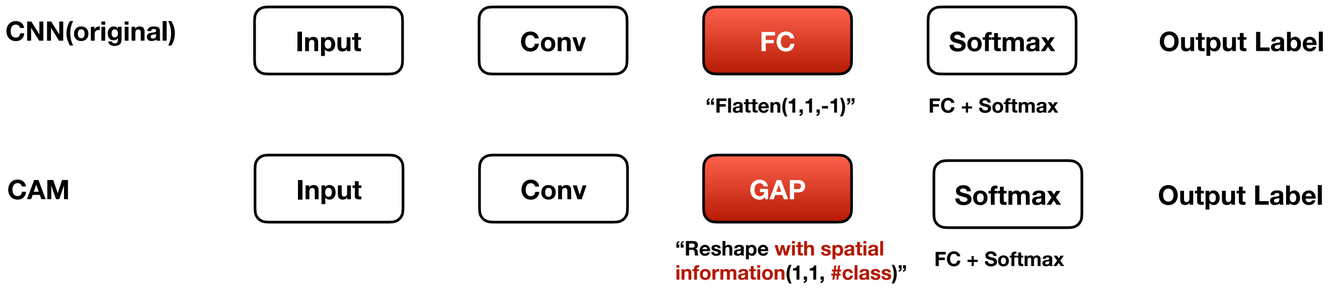
1-1. 그렇다면, “Localization”은 무엇인가?

주로 image data의 object detection task에서 등장하는 개념이다. 다양한 object가 mix된 data에서 개별 object의 ‘위치’ 정보를 출력하는 역할을 한다.

1-2. 그렇다면, “CAM”은 무엇인가?

“Class Activation Map”의 줄임말이다. “Learning Deep Features for Discriminative Localization(2015)”에서 제안된 개념이다. CNN구조에서 마지막 Convolution layer에서 얻어낸 feature map에 가중치를 곱해 얻어낸다. Model이 class를 판단하는 데 중요하게 고려한 feature map 혹은 object를 CAM 결과물을 통해 확인할 수 있다. 이 과정에서 “GAP”연산을 활용한다.
CAM을 얻는 과정을 자세히 살펴보자.

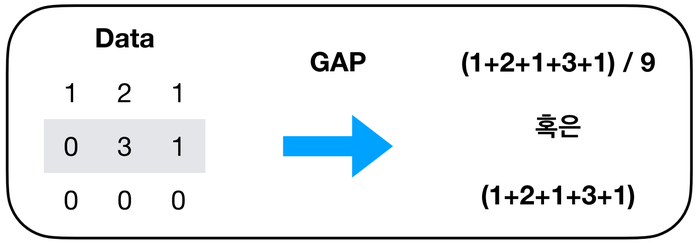
CAM을 얻기 위해선, 기존의 CNN구조에 1가지 변형이 적용된다. FC layer가 GAP로 대체되는 변형이 그것이다.



기존 CNN의 FC layer를 GAP layer로 대체했다.

1-2-1. “GAP”란 무엇인가?

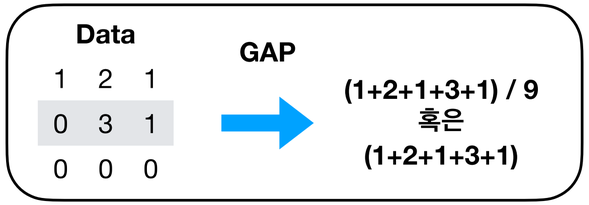
“Global Average Pooling”의 줄임말이다.
“Network In Network(2014)” 논문에서 제안된 개념이다. Pooling 방법 중 하나이다.
예를 들면, (3,3) data를 (1,1) shape로 축약(요약)하는 방법이다.



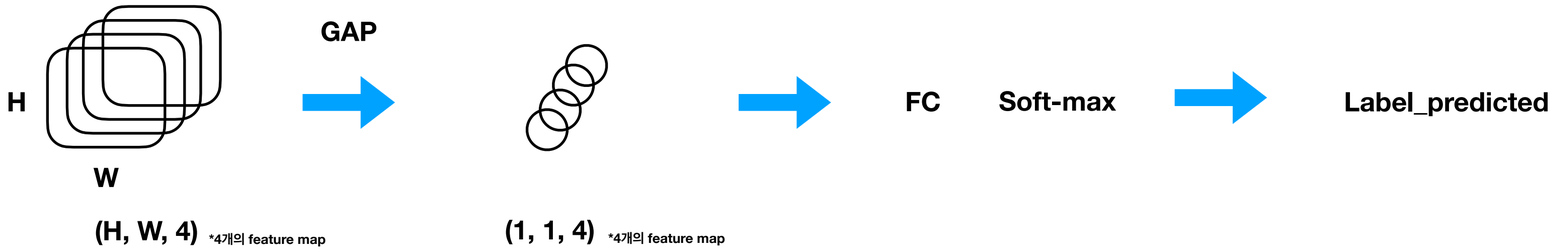
(15) Paper_study : “Grad CAM : Visual Explanations from Deep networks via Gradient based Localization”

1-2-1. “GAP”란 무엇인가?

“Global Average Pooling”의 줄임말이다.
“Network In Network(2014)” 논문에서 제안된 개념이다. Pooling 방법 중 하나이다.
예를 들면, (3,3) data를 (1,1) shape로 축약(요약)하는 방법이다.

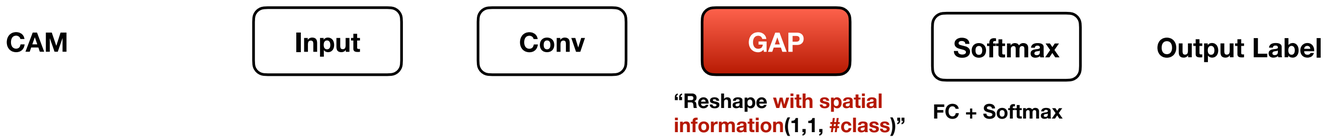


3-dimension의 image data에 대입해보자.



위와 같이, GAP를 통해 각 feature map이 (H,W) shape에서 spatial information을 요약한 (1,1) shape로 요약된다.
이를 통해, GAP는 기존 CNN의 FC layer 보다 spatial information을 잘 보존하면서, classification을 위한 soft-max layer에 input을 제공한다.

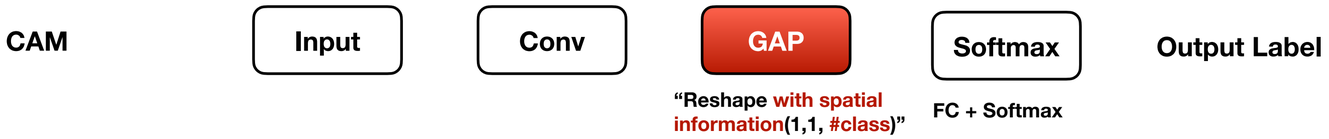
1-2-2. GAP와 CAM구조를 이해했다면, ‘CAM’을 계산하는 process를 살펴보자.



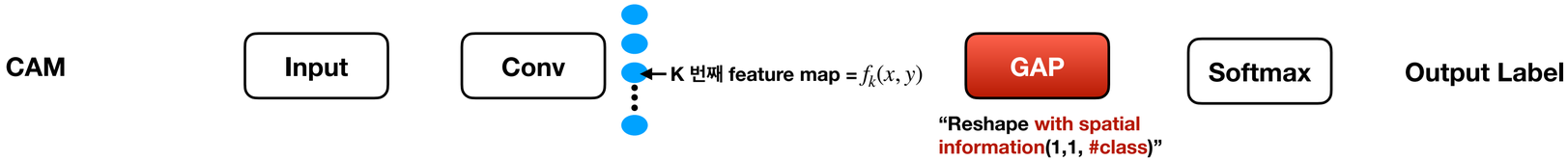
CAM은 위와 같은 구조를 갖는다.
각 단계를 수식으로 generalize해 CAM연산 process를 살펴보자.
먼저, 모든 convolution layer들을 거친 마지막 feature map을 $f_k(x, y)$ 라고 하자.

(15) Paper_study : “Grad CAM : Visual Explanations from Deep networks via Gradient based Localization”

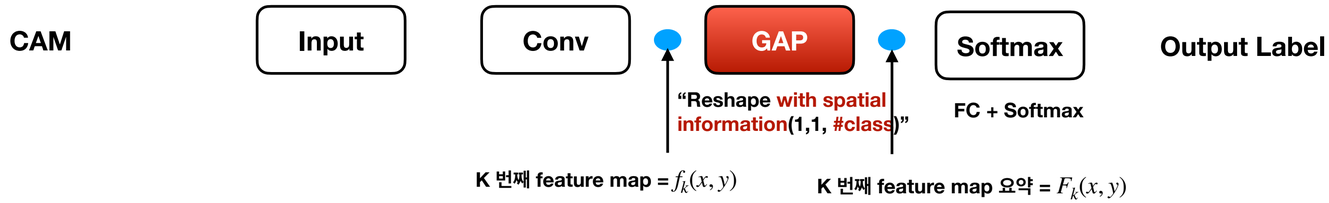
1-2-2. GAP와 CAM구조를 이해했다면, ‘CAM’을 계산하는 process를 살펴보자.



CAM은 위와 같은 구조를 갖는다.
각 단계를 수식으로 generalize해 CAM연산 process를 살펴보자.
먼저, 아래와 같이 모든 convolution layer들을 거친 마지막 feature map을 $f_k(x, y)$ 라고 하자.



그리고 Convolution layer로 얻은 feature map을 GAP 연산한다. 아래와 같이 이는 $F_k(x, y)$ = GAP($f_k(x, y)$) 로 표현한다.



GAP 연산으로 얻은 값($F_k(x, y)$)을 Weight 적용해 각 class별 score를 계산한다. (주의 : soft-max 입력 전 단계로, soft-max input 값을 계산하는 단계이다.)
이를 S_c 라고 하자. 이 score(S_c)는 클래스 ‘c’에 대한 각 feature map 의 중요도(영향도)를 나타낸다.

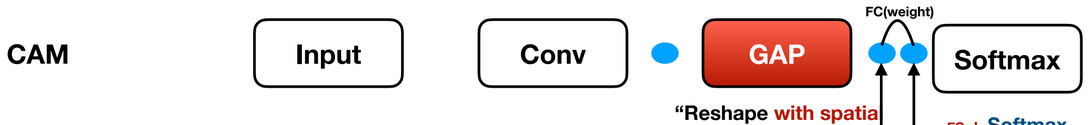
S_c = 클래스 c 에 대한 ‘k번째 feature map'의 중요도 = $\sum_k w_k^c F_k = \sum_k w_k^c \sum_{x,y} f_k(x, y) = \sum_{x,y} M_c(x, y)$

$* M_c(x, y) = \sum_k w_k^c f_k(x, y)$ = class c 에 대한 feature map의 activation 값
(x,y 에 대한 summation(upscaling)으로 ‘Map’을 생성함)

(15) Paper_study : “Grad CAM : Visual Explanations from Deep networks via Gradient based Localization”

1-2-2. GAP와 CAM구조를 이해했다면, ‘CAM’을 계산하는 process를 살펴보자.

GAP 연산으로 얻은 값($F_k(x, y)$)을 Weight 적용해 각 class별 score를 계산한다. (주의 : soft-max 입력 전 단계로, soft-max input 값을 계산하는 단계이다.)
이를 S_c 라고 하자. 이 score(S_c)는 클래스 ‘c’에 대한 각 feature map 의 중요도(영향도)를 나타낸다.



K 번째 feature map 요약 = $F_k(x, y)$

$$* M_c(x, y) = \sum_k w_k^c f_k(x, y)$$ = class c 에 대한 feature map의 activation 값
(x,y 에 대한 summation(upscaling)으로 ‘Map’을 생성함)

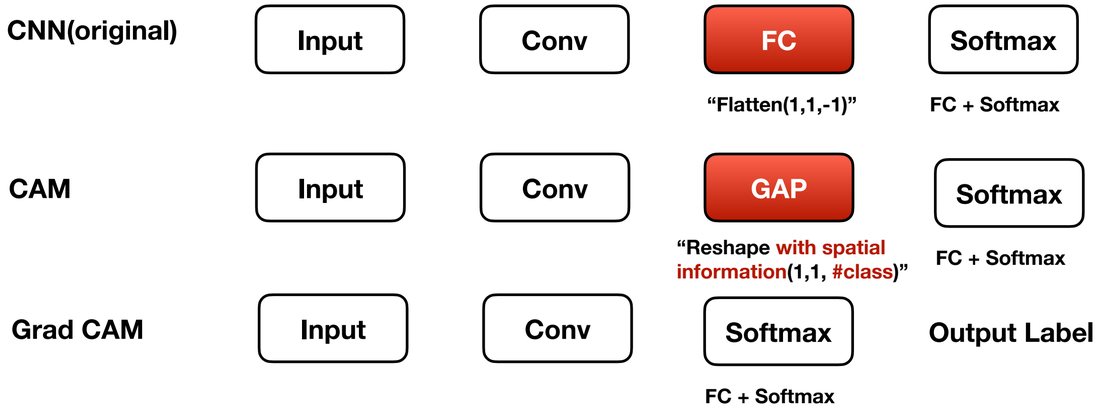
S_c = 클래스 c 에 대한 ‘k번째 feature map'의 중요도 = $\sum_k w_k^c F_k = \sum_k w_k^c \sum_{x,y} f_k(x, y) = \sum_{x,y} M_c(x, y)$

이렇게 얻은 S_c (score) 를 soft-max function에 입력해 확률 값으로 변환 뒤, classification을 위한 output label을 얻는다.

그리고 S_c (score) 를 원래의 input image의 size로 upscaling하여 CAM 결과 값을 얻어낸다.

2. ‘CAM’을 알아봤다. 그럼 다시 처음으로 돌아가, ‘Grad-CAM’이란 무엇인가?

앞서 살펴본 CAM 구조의 GAP layer 에 의존하지 않고, 모든 convolution layer 에 CAM 결과물을 얻어낼 수 있는 방법이다.
각 convolution layer output에 대한 gradient 를 활용하기 때문에, ‘Gradient Weighted CAM, Grad-CAM’ 이라 한다.



왼쪽과 같이, Grad CAM은 GAP를 거치지 않고 관심을 갖는 Conv layer에서 얻은 feature map($A_{i,j}$) 을 그대로 활용해 CAM을 구한다. 대신, 이 과정에서 해당 feature map에 대한 gradient를 활용한다.

(15) Paper_study : “Grad CAM : Visual Explanations from Deep networks via Gradient based Localization”

2. ‘CAM’을 알아봤다. 그럼 다시 처음으로 돌아가, ‘Grad-CAM’이란 무엇인가?

앞서 살펴본 CAM 구조의 GAP layer 에 의존하지 않고, 모든 convolution layer 에 CAM 결과물을 얻어낼 수 있는 방법이다.
각 convolution layer output에 대한 gradient 를 활용하기 때문에, ‘Gradient Weighted CAM, Grad-CAM’ 이라 한다.

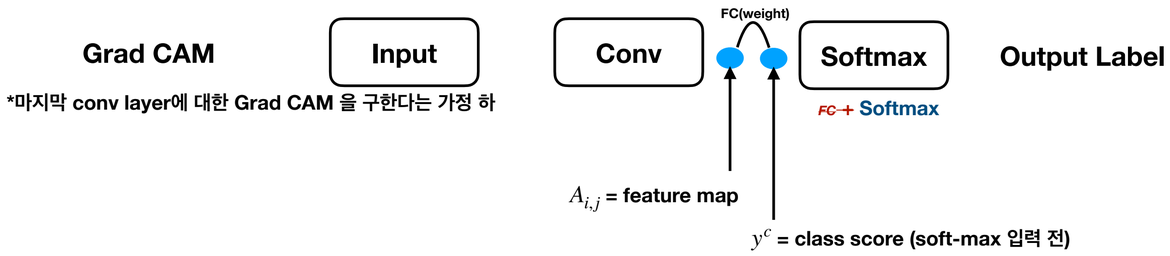
CNN(original) Input → Conv → FC → Softmax → Output Label
“Flatten(1,1,-1)” / FC + Softmax

CAM Input → Conv → GAP → Softmax → Output Label
“Reshape with spatial information(1,1, #class)” / FC + Softmax

Grad CAM Input → Conv → Softmax → Output Label
FC + Softmax

왼쪽과 같이, Grad CAM은 GAP를 거치지 않고 관심을 갖는 Conv layer에서 얻은 feature map($A_{i,j}$) 을 그대로 활용해 CAM을 구한다. 대신, 이 과정에서 해당 feature map에 대한 gradient를 활용한다.

2-1. Grad CAM 을 계산하는 과정은 ?



*마지막 conv layer에 대한 Grad CAM 을 구한다는 가정 하

먼저, 왼쪽의 구조에서 $A_{i,j}$, y^c 를 활용해 α_k^c 를 구한다.

α_k^c 는 “Neuron importance, importance weight"라고 불리며, k번째 feature map이 c class 에 미치는 중요도, 영향도를 나타낸다.

$$\alpha_k^c = \frac{1}{Z}\sum_{i,j}\frac{\partial y^c}{\partial A_{i,j}^k} = gradient$$

그리고 α_k^c (gradient, 변화정도)를 feature map 에 곱한 뒤, ReLU를 통과시켜 Grad CAM을 얻어낸다.

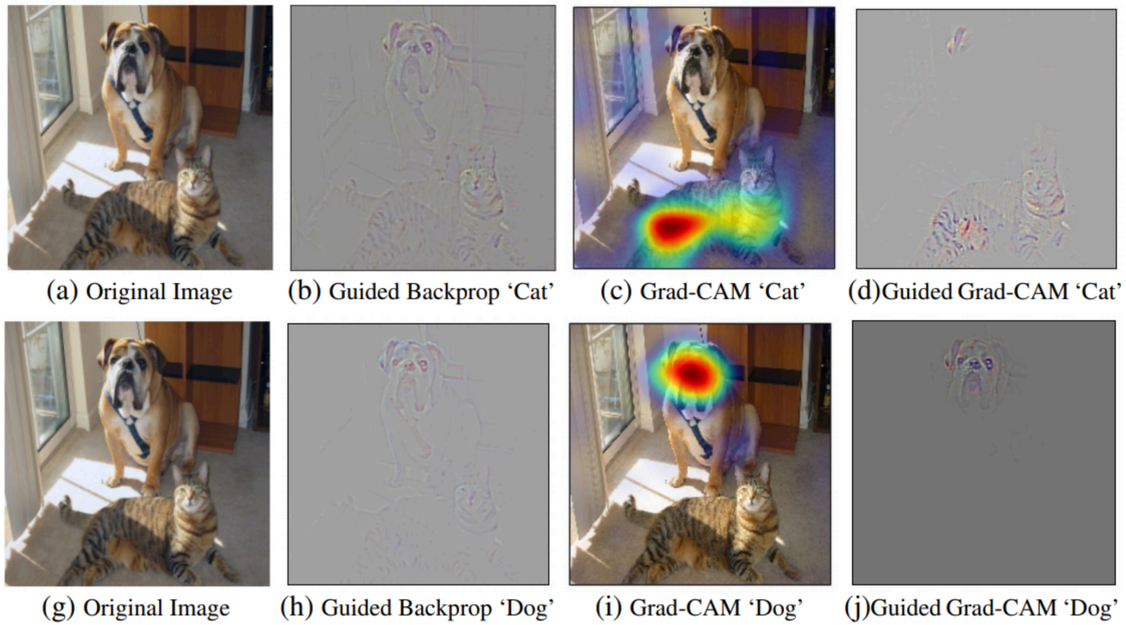
*ReLU를 입히는 이유는 CAM 결과물 자체가 각 feature map의 class에 대한 positive한 영향도만 필요로 하기 때문이다.

$$GradCAM = scaling(ReLU(\sum_k \alpha_k A^k))$$

(15) Paper_study : “Grad CAM : Visual Explanations from Deep networks via Gradient based Localization”

3. ‘Guided Grad CAM’이란 무엇인가?

Grad CAM은 class와 관련된 부분을 찾아내는 Localization에는 좋은 성능을 보인다. 하지만, Bilinear upscaling의 영향으로 class와 관련된 부분의 ‘detail’을 잘 잡아내지는 못하는 단점이 존재한다.
이를 “guided back-propagation”값을 Hadamard 연산해 보완하는 방법이 바로 ‘Guided Grad CAM’이다.



출처 : “Grad CAM : Visual Explanations from Deep networks via Gradient based Localization”

왼쪽의 그림과 같이, guided back-propagation을 적용한 (d), (j) 는 class 에 대한 detail을 보다 극명히 잡아내는 것을 볼 수 있다.

3-1. ‘guided back-propagation’이란 무엇인가?

ReLU를 예로 들면, forward pass는 ReLU(max(x,0))와 동일하나, backward pass에서 차이가 있다. Guided backprop은 Backprop시, input과 gradient 값을 모두 양수인 경우에만 backward로 흘려준다. 이를 통해, 더 적은 수의 gradient를 사용함으로써 보다 깔끔하고 극명한 Grad CAM 결과물을 시각화 할 수 있다.