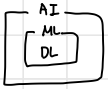
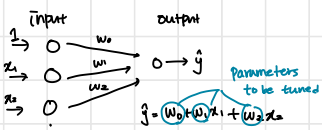
Ch1. Introduction



* DL (deep learning)
: Data → [ ] → [ ] … → [ ] → output
layer
hidden layer
mapping
→ $f \circ f_L \circ f_{L-1} \circ \cdots \circ f_1$: 합성함수

- f_L complex, f_1 simple
① FC-net (Fully connected) → 현재 많이 이용
② CNN (Convolutional neural networks)
③ RNN (Recurrent neural networks) : sequential data
- 발전 계기 : CNN
AlexNet (5 convolution + 3 FC) → VGGNet (deep CNN, 16-19 weight)
→ GoogleNet (22 layers) → Deep Residual Net (100-1000 layers)
- ML algorithm 모두 simple
- 큰 dataset이 필요 가능. (performance ↑)
- 한계
① not sample-efficient. ② adversarial attacks에 vulnerable
③ Suffer from hyperparameters.

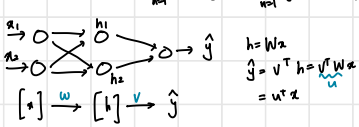
Ch2. Linear models for Regression



* Linear model
= weighted sum of features

$x(n) = \begin{bmatrix} 1 \\ x_1(n) \\ x_2(n) \end{bmatrix}$ $w = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}$ $\hat{y}(n) = w^T x(n)$

- error function: $J = \sum_{n=1}^{N} (y(n) - \hat{y}(n))^2 = \sum_{n=1}^{N} (y(n) - w^T x(n))^2$



$h = Wx$
$\hat{y} = v^T h = \underbrace{v^T W}_{u} x$
$= u^T x$

* Regression :
input x → [model $f_w(x)$] → output y

model $f_w(x) = E[y|x] = \int y\, p(y|x)\, dy$
→ linear model이라면 $f_w(x) = w^T x$

Ex. Suppose
$D = \{(x_1, y_1), (x_2, y_2), (x_3, y_3)\}$, $y_n = w_1 x_n + w_0$.
$y_1 = w_1 x_1 + w_0$ $e_1 = y_1 - w_1 x_1 - w_0$
$y_2 = w_1 x_2 + w_0$ ⇒ $e_2 = y_2 - w_1 x_2 - w_0$
$y_3 = w_1 x_3 + w_0$. $e_3 = y_1 - w_1 x_3 - w_0$

determine w_1 & w_0 ($e_1^2 + e_2^2 + e_3^2$을 최소화).

① = # equations, ② = # unknowns
- ① > ② : over-determined, under-complete 현실적인 regression
- ① = ② : Complete → Ridge
- ① < ② : over complete → LASSO

* Problem set-up : $f(x): x \to y$

Model
$f(x) = w_1 \phi_1(x) + w_0$

Error function
$J = \sum_{n=1}^{5} (y(n) - w_1 \phi_1(x(n)) - w_0)^2$
$\frac{\partial J}{\partial w_1} = 2 \sum_{n=1}^{5} (y(n) - w_1 \phi_1(x(n)))(-\phi_1(x(n))) = 0$
$w_1 \left[\sum_{n=1}^{3} \phi_1^2(x(n))\right] = \left[\sum_{n=1}^{3} y(n) \phi_1(x(n))\right]$

* Least square method.
$\sum_{n=1}^{N} (y(n) - w^T \phi(x(n)))^2 = \sum_{n=1}^{N} e(n)^2 = [e(1) \cdots e(N)] \begin{bmatrix} e(1) \\ \vdots \\ e(N) \end{bmatrix} = e^T e = \|e\|_2^2$

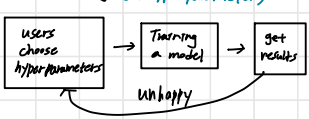
$J_{LS}(w) = \frac{1}{2N} \sum_{n=1}^{N} (y_n - w^T \phi(x(n)))^2 = \frac{1}{2N} \|y - \phi^T w\|_2^2$

$\phi^T = \begin{bmatrix} \phi_0(x_1) & \phi_1(x_1) & \cdots & \phi_M(x_1) \\ \phi_0(x_2) & \phi_1(x_2) & \cdots & \phi_M(x_2) \\ \vdots & & & \vdots \\ \phi_0(x_N) & \phi_1(x_N) & \cdots & \phi_M(x_N) \end{bmatrix}$ $(y - \phi^T w)^T (y - \phi^T w)$

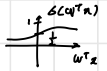
⇒ 최적화 : 미분 값 0 → $\frac{1}{2N} \|y - \phi^T w\|_2^2 = \frac{1}{2N} (y^T y - w^T \phi y - y^T \phi^T w + w^T \phi \phi^T w)$
$\frac{\partial}{\partial w} (\|y - \phi^T w\|_2^2) = \phi\phi^T w - \phi y = 0$
$\phi\phi^T w = \phi y$ ∴ $w_{LS} = (\phi\phi^T)^{-1} \phi y = \phi^{\dagger} y$

* Linear model (likelihood 사용 → randomness 추가)
$f(x(n)) = w^T x(n) + \varepsilon(n)$ & $\varepsilon(n) \sim N(0, \delta^2)$
$P(y|x) = N(w^T x, \delta^2) = \frac{1}{\sqrt{2\pi\delta^2}} e^{-\frac{1}{2\delta^2}(y - w^T x)^2}$
N samples (independent)
$\prod_{n=1}^{N} P(y(n) | w^T x(n)) \xrightarrow{\log} \sum_{n=1}^{N} \log P(y(n) | x(n))$

* overfitting을 방지하기 위해서는 weight vector $\|w\| < B$ 이어야 함
- Ridge Regression
$\min_w \frac{1}{2} \|y - \phi^T w\|_2^2 + \frac{\lambda}{2} \|w\|_2^2$ (regularizer, LS 항)
$w_{ridge} = (\Phi\Phi^T + \lambda I)^{-1} \Phi y$ (λ: hyperparameter)



* logistic model
$f_w(x) = p(y=1|x)$ $\hat{y}_n = \sigma(w^T x_n)$



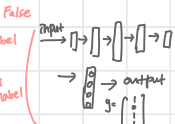
$p(y|x) \sim$ Bernoulli
$p(y|x) = p(y=1|x)^y \, p(y=0|x)^{1-y}$
$= p(y=1|x)^y [1 - p(y=1|x)]^{1-y}$
- objective : $\prod_{n=1}^{N} p(y(n)|x(n))$
- error function $J(w)$
① Gradient descent : $w^{(k+1)} \leftarrow w^{(k)} - \alpha \nabla J(w^{(k)})$
② Newton's method : $w^{(k+1)} \leftarrow w^{(k)} - \alpha H^{-1} \nabla J(w^{(k)})$
Hessian $= \nabla^2 J(w)$

* Softmax regression.
- multi class (ex. 3)
compute $p(y=1|x)$ $w_1^T x$
$p(y=2|x)$ $w_2^T x$ ⇒ $p(y=k|x) = e^{w_k^T x}$
$p(y=3|x)$ $w_3^T x$
- $p(y_n = k | x_n) = \text{Softmax}(\theta_k) = \frac{e^{w_k^T x_n}}{\sum_{j=1}^{K} e^{w_j^T x_n}}$

ex. model → $\hat{y} = \begin{bmatrix} 0.7 \\ 0.2 \\ 0.1 \end{bmatrix}$ $y = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ Tiger / lion / cat
p t
<target>
1 : true 0 : False

→ cross entropy $H(p, t) = -\sum p(x) \log f(x)$
$CE = -1 \log 0.7 - 0 \log 0.2 - 0 \log 0.1 = -\log 0.7 = 0.31$

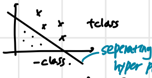
single label / multi label



Ch3. Perceptron

- Perceptron: single layer neural network (hidden layer X)
- Multilayer perceptron : multi layer extension of perceptron.
  = Fully connected Network.

* linear classification



$f(x) = w^T x + b$ (w, b : parameter)

* error function

$J(w) = \sum_{x_n \in M} w^T x_n y_n$

$w = \arg\min_w \left[-\sum_{x_n \in M} w^T x_n y_n \right]$ apply gradient descent (소거)

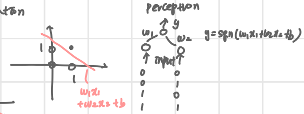
Gradient : $\nabla_w (w^T x_n y_n) = x_n y_n$

* Algorithm outline

① train sample x_n
② misclassified 되진 않았는지 check
→ 맞게 분류: do nothing
→ 잘못 분류 : update w ⇒ $w_{k+1} = w_k + \alpha x_n y_n$
(convergence 까지 반복)

* AND operation

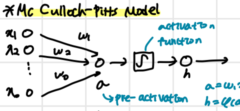
(0,1) → 0
(1,0) → 0
(0,0) → 0
(1,1) → 1



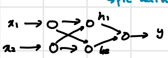
* XOR

→ 여러개의 decision boundary 필요

* Mc Culloch-Pitts Model



$a = w_1 x_1 + \cdots + w_D x_D + b$
$h = \varphi(a)$

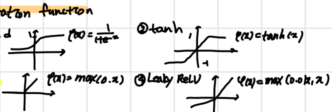


$h_1 = \varphi(w_{11} x_1 + w_{12} x_2)$
$h_2 = \varphi(w_{21} x_1 + w_{22} x_2)$
$y = v_1 h_1 + v_2 h_2 = v_1 \varphi(w_{11} x_1 + w_{12} x_2) + v_2 \varphi(w_{21} x_1 + w_{22} x_2)$
$= v_1 \varphi(w_1^T x) + v_2 \varphi(w_2^T x)$ → adaptive basis

linear model: $y = v_1 \varphi_1(x) + v_2 \varphi_2(x)$ → basis fixed.

* activation function

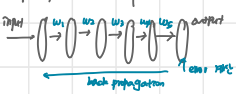
① sigmoid $f(x) = \frac{1}{1+e^{-x}}$
② tanh $\varphi(x) = \tanh(x)$
③ ReLU $f(x) = \max(0, x)$
④ Leaky ReLU $\varphi(x) = \max(0.01x, x)$



→ 원점에서는 미분불가 ⇒ subgradient
$\partial f_{x_0} = \{ a \mid a^T x \le f(x) \}$

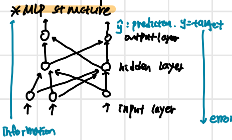
- ReLU의 장점
① cheap
② converge faster
③ sparsely activated

단점
① 뉴런이 죽을 수 있음 = dying ReLU problem
→ 해결 위해 leaky ReLU, ELU



⑤ softplus $\varphi(x) = \log(1+e^x)$

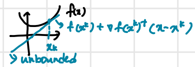
* MLP structure



#hidden layers
#hidden nodes ⊂ Hyperparameter
⇓
'Neural Architecture Search'

* back propagation

- gradient descent
$\min_x f(x)$

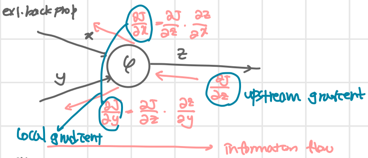


linear approximation
$\min\left[f(x^k) + \nabla f(x^k)^T (x - x^k) + \frac{1}{2\alpha} \|x - x^k\|_2^2 \right]$ (J)

$\frac{\partial J}{\partial x} = \nabla f(x^k) + \frac{1}{2\alpha}(2x - 2x^k) = \nabla f(x^k) + \frac{1}{\alpha}(x - x^k) = 0$

$\alpha \nabla f(x^k) + (x - x^k) = 0$
$\therefore x = x^k - \alpha \nabla f(x^k)$ ⇒ $x^{k+1} \leftarrow x^k - \alpha \nabla f(x^k)$ gradient를 구해서 update

ex1. back prop



ex2.
$x \xrightarrow{w_1} h \xrightarrow{w_2} y$

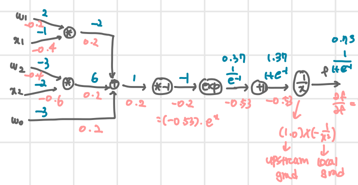
$\frac{\partial J}{\partial w_2}$: directly calculated

$\frac{\partial J}{\partial w_1}$: $\frac{\partial J}{\partial w_1}$ 은 경로를 이용해 도출. ⇒ chain rule 이용.

$\hat{h} = w_1 x$ → φ → $\hat{y} = w_2 h$
$\frac{\partial J}{\partial \hat{h}} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \hat{h}}$ ← $\frac{\partial J}{\partial \hat{y}}$
$= \frac{\partial J}{\partial \hat{y}} \cdot w_2 \varphi'(\hat{h})$. $\frac{\partial \hat{y}}{\partial \hat{h}} = \frac{\partial}{\partial \hat{h}}[w_2 \varphi(\hat{h})] = w_2 \varphi'(\hat{h})$

ex. Calculate Gradient by back prop.

$f(w, x) = \frac{1}{1 + e^{-(w_1 x_1 + w_2 x_2 + w_0)}}$



Ch4. Optimization

- error function
$J(\theta) = \frac{1}{2N} \sum_{n=1}^{N} \| f(x_n; \theta) - y_n \|^2$ | θ: model parameters | square loss : $J(\theta) = \sum_{n=1}^{N} (y_n - f_\theta(x_n))^2$

→ want : $\theta = \arg\min_\theta J(\theta)$, error를 최소화하는 θ

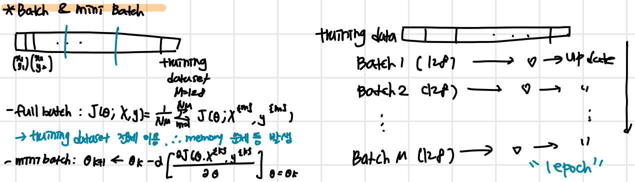
* Gradient Descent → iteration 반복해야 함, 그래서 gradient 계산

$\theta_{k+1} \leftarrow \theta_k - \alpha \left[\frac{\partial J(\theta)}{\partial \theta} \right]_{\theta = \theta_k}$ (current gradient) $= \theta_{k+1} \leftarrow \theta_k + \alpha_k v_k$ (stepsize, direction)

일반적으로 $\langle \nabla J(\theta) \cdot v \rangle < 0$

Gradient = steepest descent

* Batch & mini Batch



- full batch : $J(\theta; X, y) = \frac{1}{N_M} \sum_{m=1}^{N_M} J(\theta; x^{(m)}, y^{(m)})$
→ training dataset 전체 이용. ∴ memory 문제 등 발생
- mini batch : $\theta_{k+1} \leftarrow \theta_k - \alpha \left[\frac{\partial J(\theta, x^{(k)}, y^{(k)})}{\partial \theta} \right]_{\theta = \theta_k}$
→ training Data를 여러개의 batch로 나누고 특정 batch만 이용해서 gradient 추출
if batch size가 1이면 sample by sample 사용.
∴ error optimization 필요

* Saddle point
: 어떤 point에서 보는 방향에 따라 min or max 바뀜 (3 dimension 이상일 때)

* Stochastic approximation = SGD

$J(\theta) = \sum_{n=1}^{N} (y_n - f_\theta(x_n))^2 \approx E[(y - f_\theta(x))^2]$

- $\theta \leftarrow \theta - \alpha \nabla_\theta ((y_n - f_\theta(x_n))^2)$: batch size에 관련.

$J(\theta) = \sum_{n=1}^{N_1} (y_n - f_\theta(x_n))^2$ mini batch
$n = 1, 2, \cdots N_1, N_1+1, \cdots N_2, N_2+1, \cdots N_3, \cdots N$ (M, M, M, M)
$\Rightarrow J(\theta) = \sum_{n=1}^{N_1} (y_n - f_\theta(x_n))^2 + \sum_{n=N_1+1}^{N_2} (y_n - f_\theta(x_n))^2 + \cdots + \sum_{n=N_{M-1}+1}^{N} (y_n - f_\theta(x_n))^2$

mini-batch Gradient Descent = SGD
→ problem: error (경향이 noisy 함)
step size만 잘 정하면 좋다 여겨짐.

- learning rate를 적절하게 정하기는 어려움
① too small : 속도 느림
② too big : fluctuation 심함.
Sol) moving average 사용.

* Exponentially Weighted Moving average
moving average θ_t : $V_t = \beta V_{t-1} + (1-\beta)\theta_t$
(V_{t-1}, θ_t) convex com.
↑ previous MA ↑ current value

ex. $V_0 = 0$
$V_1 = \beta V_0 + (1-\beta)\theta_1 = (1-\beta)\theta_1$
$V_2 = \beta V_1 + (1-\beta)\theta_2 = \beta(1-\beta)\theta_1 + (1-\beta)\theta_2 = (1-\beta)(\beta\theta_1 + \theta_2)$
$V_3 = \beta V_2 + (1-\beta)\theta_3 = (1-\beta)(\beta^2\theta_1 + \beta\theta_2 + \theta_3)$
⋮
∴ $V_t = (1-\beta)(\beta^{t-1}\theta_1 + \beta^{t-2}\theta_2 + \cdots + \theta_t)$ ⇒ approximate average over $\frac{1}{1-\beta}$ samples
ex. $\beta = 0.9$ 이면 $\frac{1}{1-\beta} = 10$

* Gradient Descent with Momentum
$\theta_{t+1} = \theta_t - \alpha(\nabla J(\theta_t))$, α = step size
→ moving average를 사용하여 gradient update
$V_{t+1} = \beta V_t + (1-\beta)[\nabla J(\theta_t)]$
$\theta_{t+1} = \theta_t - \alpha V_{t+1}$

* Rprop (Resilient Back prop)
- Manhattan - learning rule
→ gradient의 부호에만 영향을 받아 update
if $\nabla < 0$, $\theta_i \leftarrow \theta_i + \Delta_0$
$\nabla > 0$, $\theta_i \leftarrow \theta_i - \Delta_0$
- Manhattan - learning rule에서 확장하여 진행

$$\Delta_i^{(t)} = \begin{cases} \eta^+ \Delta_i^{(t-1)} & \text{if } \frac{\partial J}{\partial\theta_i^{(t)}}\frac{\partial J}{\partial\theta_i^{(t-1)}} > 0 \\ \eta^- \Delta_i^{(t-1)} & \text{if } \frac{\partial J}{\partial\theta_i^{(t)}}\frac{\partial J}{\partial\theta_i^{(t-1)}} < 0 \\ \Delta_i^{(t-1)} & \text{else.} \end{cases}$$

* Ad Grad : Adaptive Gradient
$V_i^{(t+1)} = \frac{\alpha}{\sqrt{r_i^{(t)}+\epsilon}} \nabla J(\theta_i^{(t)})$
→ iteration이 늘면 learning rate ↓

* RMS Prop : Rprop + SGD
→ Adaptive한 learning rate를 weight에 따라 적용
$V_{t+1} = \frac{\nabla J(\theta_t)}{\sqrt{r_{t+1}+\epsilon}}$, $\theta_{t+1} = \theta_t - \alpha V_{t+1}$

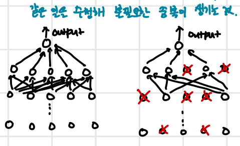
* SGD → GD with momentum → RMS prop
(exponentially weight moving average) (MA → squared gradient)

* ADAM Optimization
= momentum + bias correction + RMS prop
gradient의 MA, squared gradient의 MA를 구해서 각각 bias correction 해주고
gradient의 MA를 squared gradient의 MA로 나눔.
$\frac{\alpha}{\sqrt{r_t^{bc}}}$ ⟺ adaptive learning rate (α: constant)

ex. η = decay rate. Ω = epoch number, α_0 = constant.
① $\alpha = \frac{1}{1+\eta\Omega}\alpha_0$ $\Omega=1$ → $\alpha = \frac{1}{1+\eta}\alpha_0$ | epoch가 늘수록
② $\alpha = 0.95^{\Omega}\alpha_0$ $\Omega=2$ → $\alpha = \frac{1}{1+2\eta}\alpha_0$ ↓ learning rate 감소
③ $\alpha = \frac{k}{\sqrt{\Omega}}\alpha_0$
④ $\alpha = \frac{k}{\sqrt{t}}\alpha_0$

Ch5. DropOut

: co-adaptation을 해결하기 위한 방법
↳ 같은 layer에서 2개 이상의 node가 input, output, parameter가 동일하게 같은 일을 수행해 불필요한 중복이 생기는 것.

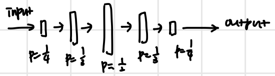


- 매번 batch마다 다른 node 없앰 (training 시)
- test 시에는 모든 node 사용 (dropout X)) rescaling

<regular Net>
$E_N = \frac{1}{2}(y - \sum_j p_j w_j x_j)^2$
$\frac{\partial E_N}{\partial w_i} = -y p_i x_i + w_i p_i^2 x_i^2 + \sum_{j\neq i} w_j p_i p_j x_i x_j$

<drop out net>
$E_D = \frac{1}{2}(y - \sum_j \delta_j w_j x_j)^2$
$\frac{\partial E_D}{\partial w_i} = -y\delta_i x_i + w_i \delta_i^2 x_i^2 + \sum_{j\neq i} w_j \delta_i \delta_j x_i x_j$

* Inverted Dropout (training ⇒ dropout, testing ⇒ rescaling) 이를 test시에 하지 않고 training에서 rescaling 함.

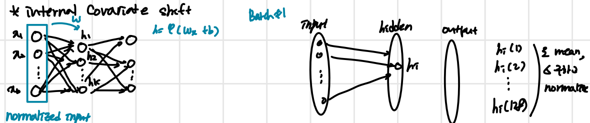


dropout rate ∈ hyper parameter
⇒ FC net에서 많이 사용

Ch6. Normalization

≡ Standardization
$X = \{x_1, x_2, \cdots, x_N\}$ ⟶ $Z = \{z_1, \cdots, z_N\}$ mean = 0, variance = 1
$z_n = \frac{x_n - \mu_x}{\sigma_x}$ ⇒ $\mu_z = 0$, $\sigma_z = 1$

* internal Covariate shift



⇒ batch - normalization

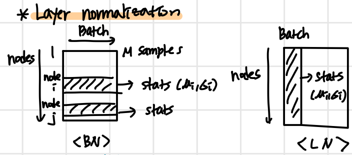
- input을 normalize 해도 hidden layer에서 다시 normalize 필요.
- input distribution 에서는 Training = test 지만 covariance shift 에서는 성립 X
- ① pre activation 와 ② activation 두 곳 중 pre-activation에서 normalize 함.
→ ⓐ → [∫] → ⓗ →
pre-activation activation

* Batch Normalization
⇒ M sized mini batch 각각에 적용됨.
① Normalize pre-activation a_i : $\tilde{z}_i = \frac{a_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$ → 모든 배치 같은 값. $\mu_i = \frac{1}{M}\sum_{m=1}^{M} a_{i,m}$, $\sigma_i^2 = \sum_{m=1}^{M}(a_{i,m} - \mu_i)^2$
② Scale & shift : $z_i = \gamma_i \tilde{z}_i + \beta_i$ → (γ_i, β_i)도 back-prop 과정에서 나오는 parameter

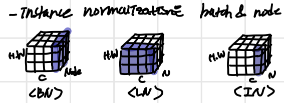
ⓧ $\xrightarrow{w^{(1)}, b^{(1)}}$ ⓐ$^{(1)}$ → [∫] → ⓗ$^{(1)}$ $\xrightarrow{w^{(2)}, b^{(2)}}$ ⓐ$^{(2)}$ → [∫] → ⓗ$^{(2)}$ → ...
input activation function hidden layer

BN
ⓧ $w^{(1)}, b^{(1)}$ ⓐ$^{(1)}$ $\xrightarrow{\gamma^{(1)}, \beta^{(1)}}$ ⓩ$^{(1)}$ → [∫] → ⓗ$^{(1)}$ ⟶ ⓐ$^{(2)}$
← back prop : γ, β update

* Layer normalization

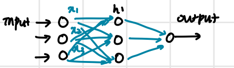


→ RNN에는 잘 맞지 않음 → RNN의 경우 많이 사용.
- mean과 variance가 모든 node들에 대해 동일히 적용.
- BN은 node 별로 μ, σ 구하고, LN은 batch 별로 μ, σ 구함.
- Instance normalization은 batch & node 전부 고려. 특정 channel 만 사용



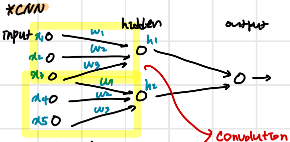
Ch7. CNN (Convolutional Neural Network)

* FC-nets



$h_1 = f(x_1, x_2, x_3)$
$= \varphi(w_{11}x_1 + w_{12}x_2 + w_{13}x_3)$

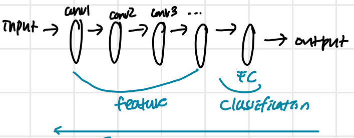
* CNN



$h_1 = \varphi(w_1x_1 + w_2x_2 + w_3x_3 + w_0)$
$h_2 = \varphi(w_1x_3 + w_2x_4 + w_3x_5 + w_0)$

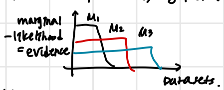
- parameter sharing

→ convolution
⇒ cross-correlation between $\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$ & $\begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$



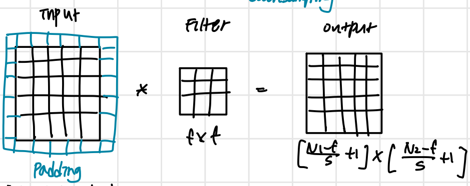
* Bayesian Model comparison

- model selection : $p(D|M_i) = \int p(D|w, M_i)\, p(w|M_i)\, dw$



* Convolution

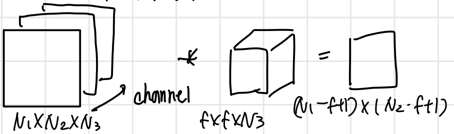
- input layer → feature maps → pooling → convolution → pooling → features
  ↳ downsampling



$\left[\frac{N_1 - f}{s} + 1\right] \times \left[\frac{N_2 - f}{s} + 1\right]$

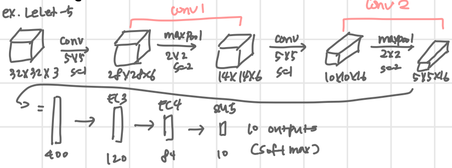
p : padding p=1
$(N_1 \times N_2)$

- convolution over volume



- channel 의 길이를 조절할 때 1×1 convolution 적용.
- Max pooling : channel 길이 유지

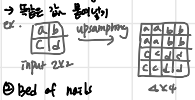
ex. LeNet-5



* Upsampling

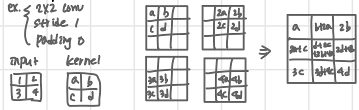
↳ Unpooling & Transposed Convolution

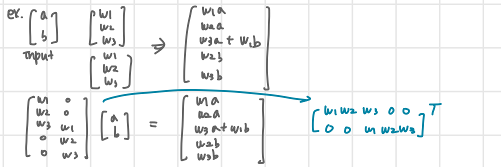
① Nearest Neighbor
→ 똑같은 값 몰아넣기
ex.



② Bed of nails
→ 0으로 채우기
③ max unpooling
→ 어디서 max 값이 나타났는지 기억해두고 나머지는 0

- Transposed Convolution

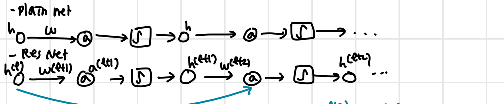




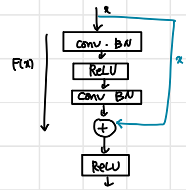
- filter의 개수를 늘리면 다양한 feature들 얻을 수 있음

* ResNet

- plain net
- Res Net



skip connection ⇒ $h^{(l+2)} = \varphi(w^{(l+2)} h^{(l+1)} + b^{(l+2)} + h^{(l)})$



$H(x) = F(x) + x$
$F(x) = H(x) - x$
↳ residual function

- gradient $\frac{\partial J}{\partial x} = \frac{\partial J}{\partial H}\frac{\partial F}{\partial x} + \frac{\partial J}{\partial H}$

* Inception Nets

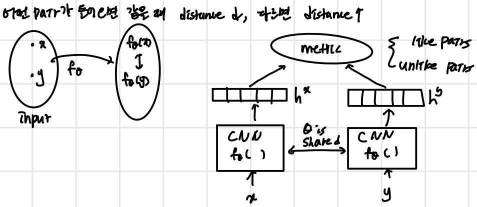
: loss를 여러개로 나누어 계산하고 나중에 total loss 계산
- v1, v3, v4 ...

* object Detection

= classification + Localization
- R-CNN : object가 어디있는지 region을 propose 함
  → 그 region에 feature가 있는지 없는지 판단
  단점 : 시간 ↑ (object를 detect하기까지)
- fast R-CNN : 하나의 CNN만 사용 ⇒ speed는 빨라짐 but overhead존재
- faster R-CNN : Region proposal ⇒ overhead 줄임. 속도 ⇑
- YOLO : grid로 나누고 나중에 합침. 속도 ↑↑. data를 한번에 처리

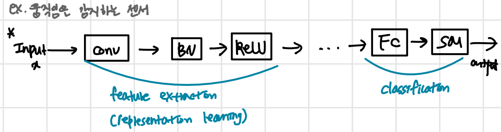
* CNN with siamese Architecture

⇒ Detection & Verification : 같은지 아닌지만 판단 ⇒ 무엇인지는 판단 X
- 어떤 pair가 들어오면 같을 때 distance ↓, 다르면 distance ↑



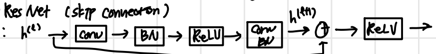
* CNN for timeseries classification

ex. 움직임을 감지하는 센서



- pre-trained CNNs

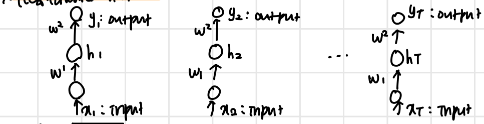
① VGG
② Inception Net (1×1, 3×3, 5×5, maxpooling) → 덜 deep하게 감
③ ResNet (skip connection)
:



- image classification
- semantic segmentation (upsampling)
- object detection (YOLO)
- verification (siamese Architecture)
- time series classification

Ch8. RNN (Recurrent Neural Network)

***Feed forward network**



: $[x_n] \rightarrow$ [model FF] $\rightarrow [y_n]$

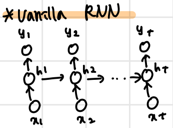
if input sequence (ex. text, video, music, …)

: $[x_1], [x_2], \cdots [x_T] \rightarrow$ [model] $\rightarrow [y_1], [y_2], \cdots [y_T]$

⇒ 각각을 independent 하게 처리하면 안됨 (dependency 고려 필요)

∴ hidden transition 필요

***Vanilla RNN**

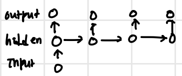


$y_t = \varphi(W_{yh} h_t + b_y)$

$h_t = \rho(W_{hx} x_t + W_{hh} h_{t-1} + b_h)$

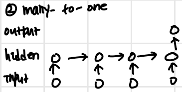
→ variable x를 handling 할 수 있어야함

① One-to-many



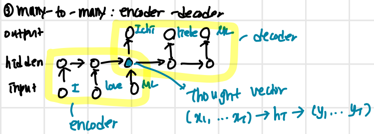
ex. music

② many-to-one



ex. video

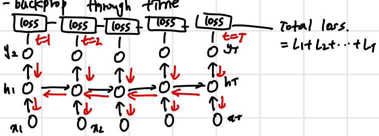
③ many-to-many : encoder-decoder



$(x_1, \cdots x_T) \rightarrow h_T \rightarrow (y_1 \cdots y_T)$

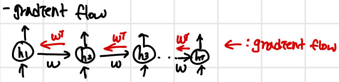
→ seq-to-seq learning

- backprop through time



Total loss $= L_1 + L_2 + \cdots + L_T$

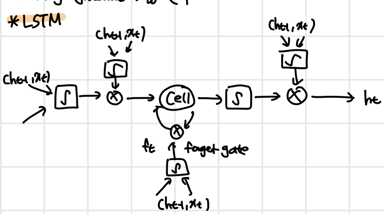
- gradient flow



exploding gradients : $W^t > 1$

vanishing gradients : $W^t < 1$

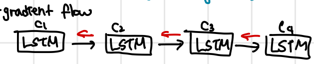
***LSTM**



- forget gate : 0과 1 사이의 값.
  - if forget gate = 0 : cell의 과거 전부 지움
  - if forget gate = 1 : 과거의 값 전부 기억

→ input . output . forget gate는 sigmoid 함수에 들어가기 때문에 항상 0과 1 사이의 값 가짐

- gradient flow



***GRU**

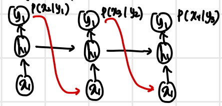
. cell 없이 update gate z_t 와 reset gate r_t 로만 이루어짐

- if $z_t = 1$: $h_t = g_t$ (h_{t-1} 무시)
- if $z_t = 0$: $h_t = h_{t-1}$ (g_t 무시)

$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot g_t$

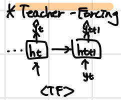
- natural language의 경우 GRU 많이 사용. 이외의 sequential data는 LSTM 많이 사용

***Generative RNN**



: 앞의 output을 다음 input으로 넣기

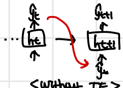
***Teacher-Forcing**



<TF>

: $\hat{y}_t$가 아닌 y_t(true 값) input

→ training은 잘 되지만 test시 문제



<without TF>

: 예측값($\hat{y}_t$)를 input

→ $\hat{y}_t$가 정확하지 않으므로 training 문제

⇒ scheduled sampling 필요.

***scheduled sampling.**

: training 시 true 값인 y_t를 input으로 넣음 (TF) → ε_i의 확률로.
점점 ε_i가 줄어들면서 $\hat{y}_t$을 사용하게 됨. (free-learning)

Ch9. Attention & Transformer model

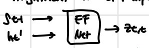
***Attention in Encoder-Decoder**

: encoder-decoder 에서 output을 예측하는 매 시점마다 encoder의 전체 input을 참고하는것
이때 같은 비율로 참고하는것이 아닌 연관성이 높은 부분에 비중을 줌

- 최종적인 $\text{context vector} = \sum_{t'} \alpha_{t,t'} h_{t'}$ → convex sum, $\sum \alpha_t$ ↵
- $\alpha_{t,t'}$: output y_t가 input 중 어디에 더 attention 해야하는지의 정도

⇒ $\alpha_{t,t'} = \dfrac{e^{z_{t,t'}}}{\sum_{k=1}^{T'} e^{z_{t,k}}}$

- Alignment model : input의 어느 부분에 더 집중 해야하는지 판단



$\rightarrow z_{t,t'}$ (S_t: hidden node in decoder, h_t: encoder의 hidden node, $z_{t,t'}$: score)

softmax에 넣기 이전 나이 ↵

→ input sequence 가 바뀌면 alignment 도 바뀜

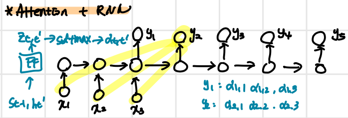
***Visual Attention**

: 어떤 image가 들어오면 문장이 output이 됨.

- encoder : CNN 사용 , decoder : RNN 사용

→ attention을 통해 image에서 찾고자 하는 부분만 집중

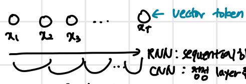
***Attention + RNN**



$y_1 : \alpha_{1,1}\ \alpha_{1,2}\ , \alpha_{1,3}$

$y_2 : \alpha_{2,1}\ \alpha_{2,2}\ . \alpha_{2,3}$

***Transformer models.**



RNN : sequential하게 dependence. (한 방향)

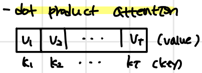
CNN : 점점 layer를 쌓아감 (작은 면적 → 큰 면적)

- sequence를 다룰때 RNN. CNN이 아닌 attention만 사용

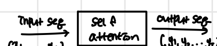
***Vanilla transformer**

: encoder + decoder = attention + positional encoding + feedforward net
↳ encoder. decoder 각각

- dot product attention



query $q_i \rightarrow \sum_j w_{ij} v_j, \ w_{ij} = \dfrac{e^{q_i^T k_j}}{\sum_j e^{q_i^T k_j}}$

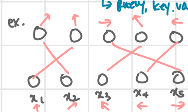


$y_i = \sum_j w_{ij} x_j$

ex $y_2 = w_{21} x_1 + w_{22} x_2 + w_{23} x_3, \ w_{21} = \dfrac{e^{x_2^T x_1}}{e^{x_2^T x_1} + e^{x_2^T x_2} + e^{x_2^T x_3}}$

↳ query, key, value가 same sequence로부터 나옴.

ex.



- Query : 단어에 대한 가중치
- Key : 단어가 Query와 얼마나 연관되었는지 비교하는 가중치
- Value : 의미에 대한 가중치

- scaled Dot-product attention

N = seq length.

$$y_i = \sum_{n=1}^{N} \left[\frac{e^{\frac{q_i k_n^T}{\sqrt{D_k}}}}{\sum_{l=1}^{N} e^{\frac{q_i k_l^T}{\sqrt{D_k}}}} \right] v_n$$

softmax $\left(\left(\frac{q_i k_n^T}{\sqrt{D_k}} \right)_{n=1:N} \right)$

- multi-head attention

$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$

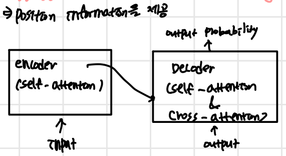
- encoder : self attention layers.
- decoder : self-attention layers. input은 masked
  ↳ 타깃 이후의 단어는 보지 않고 예측하기 위함
- Positional encoding
  ① 각각의 time step에 대해 unique한 encoding을 함
  ② 2개의 time step에 대한 distance는 consistence 해야 함
  ③ 더 긴 sequence에 대해 쉽게 generalize 될 수 있어야 함
  ④ deterministic

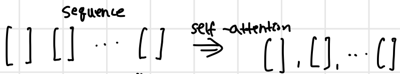
ex.
0 : 0 0 0 0
1 : 0 0 0 1
2 : 0 0 1 0
3 : 0 0 1 1
4 : 0 1 0 0
5 : 0 1 0 1

← 점점 천천히 변함 ∴ positional vector + embedding vector ⇒ sinusoidal

⇒ position information을 제공

-





⇒ need to consider "context"

자기 자신을 query로 보면 나머지가 key. value가 됨.

$x_1, x_2, \cdots, x_T \rightarrow$ [self-attention] $\rightarrow y_1, y_2, \cdots y_T$

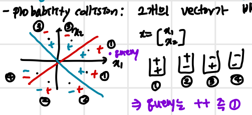
∴ 입력한 문장의 각 단어를 처리하면서 문장의 다른 위치에 있는 단어들으로부터 힌트를 받아 현재 타깃의 위치를 더 잘 encoding 할 수 있게 함

* Transformer vs Reformer

- Transformer : long sequence X. dot product attention needs complexity
- Reformer : LSH attention. Reversible layers. chunking
  ↳ dot product attention 대신으로 complexity ↓
  ↳ memory space ↓

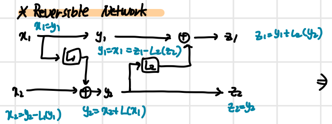
* LSH attention (Locality Sensitive Hashing)

- Hamming space: $\begin{bmatrix}1\\0\\1\end{bmatrix} \begin{bmatrix}0\\1\\1\end{bmatrix}, \cdots$
  ⇒ high dimension data를 0.1을 이용한 low dimension으로 바꿀 수 있음
- probability collision: 2개의 vector가 비슷할수록 ↑



⇒ query는 ++ 즉 ① bucket에 해당하는 것들만 사용

* Reversible Network

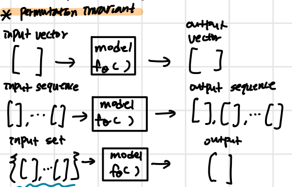


⇒ 거꾸로 recover 가능

* Linformer

: Random projection을 통해 sequence ↓

* permutation invariant



↳ set 안에 있는 것들의 순서가 바뀌더라도 같은 set이 있다면 같은 output 나와야 함

∴ permutation invariant 필요

* Deep sets

- 기본적인 permutation invariant
  : $f(X) \approx \rho\left(\sum_{x \in X} \phi(x)\right)$

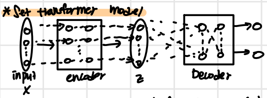
* Permutation Invariance and Equivariance

- permutation invariant: set 안에 있는 것들의 순서가 바뀌어도 response에 영향X
- permutation equivariant: input에 permutation이 있으면 output도 permutation
  → 순서도 똑같이 바뀌어야 함

ex. $N=3$, $\theta_{\lambda} = \left(\begin{bmatrix} \lambda & 0 & 0 \\ 0 & \lambda & 0 \\ 0 & 0 & \lambda \end{bmatrix} + \gamma \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \right) \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$

$= \begin{bmatrix} \lambda x_1 \\ \lambda x_2 \\ \lambda x_3 \end{bmatrix} + \begin{bmatrix} \gamma(x_1+x_2+x_3) \\ \gamma(x_1+x_2+x_3) \\ \gamma(x_1+x_2+x_3) \end{bmatrix}$

* Set transformer model



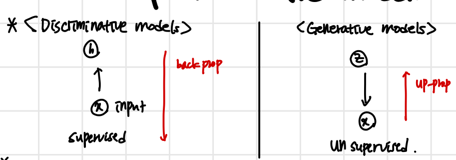
- permutation equivariant Attention modules (SAB & ISAB)
  SAB ↳ set attention block
  ISAB ↳ induced set attention block
  : query의 숫자가 더 적음 → self attention (auto encoder처럼)
- Pooling by Multihead attention
- encoder : stacks of SAB or ISAB
- set transformer는 permutation invariant 함

∴ SAB를 통해 encoder에서 permutation equivalent 한 layer를 만듦
→ multi head attention을 통한 pooling
→ aggregation → set을 잘 반영한 feature vector를 만듦 (permutation-invariant)
→ Decoder를 거쳐 target으로 mapping 됨

* Set transformer examples

① Meta Anomaly Detection
② Point cloud Classification
  ↳ 3차원의 Dataset으로 나타난 것을 classification

Ch10. Deep Generative models



* Density Estimation

$p_{model}(x;\theta) \approx p_{data}(x)$
(likelihood) (empirical distribution)

- Prescribed models
  ① Fit $p_\theta(x)$ (model distribution) to $p_{data}(x)$ (empirical distribution)
  → parameterized Distribution likelihood (확률 분포에 의한 관점)
  ② assign probability to every x in data distribution
  ex. Variational autoencoder (VAE)
- Implicit models
  ① learn Generative network, $z \rightarrow G_\theta(\ )$
  → likelihood free (randomness의 관점)
  ex. GAN (generative adversarial network)
- choose KL-divergence

$$\hat{\theta} = \arg\min_\theta D_{KL}[P_{data}(x) \| P_\theta(x)]$$

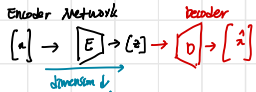
$$D_{KL}[P_{data}(x) \| P_\theta(x)] = \int P_{data}(x) \log \frac{P_{data}(x)}{P_\theta(x)} dx = \int P_{data}(x) (\log P_{data} - \log P_\theta) dx$$

$$\Rightarrow \arg\min_\theta \left[-\int P_{data}(x) \log P_\theta(x) dx \right] \Rightarrow \arg\max_\theta \left[\int P_{data}(x) \log P_\theta(x) dx \right]$$

↳ $\int \frac{1}{N} \sum_{n=1}^{N} \delta(x - x_n) \log P_\theta(x) dx$

* Autoencoder

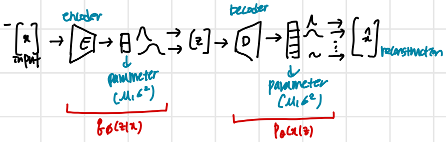
- Encoder Network



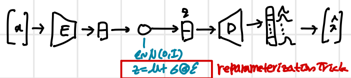
- How to train E? Given $\{x_1, \cdots x_N\}$ error : $\sum_x \| x - \hat{x} \|^2$
- ex. Google Brain
- limitation: 주어진 값을 복원하는 것이 주 목적이기 때문에 generator로 사용하기에는 부족함
  latent space에 있는 것들로 무의미한 결과값 가질 수 있음

* Variational Autoencoder

: probabilistic extension of autoencoder



* Train VAE with Reparameterization Trick



- Consider a random variable z : $z \sim N(\mu, \sigma^2)$

$\epsilon \sim N(0,1)$ $\quad E[z] = \mu + 0 = \mu$

$z = \mu + \sigma\epsilon$. $\quad Var(z) = \sigma^2$

* Training VAE – maximizing ELBO (Evidence lower bound)

- Variational lower-bound
: log-likelihood

$\log p_\theta(x) = \log \int p_\theta(x,z)\,dz = \log \int q_\phi(z|x) \frac{p_\theta(x,z)}{q_\phi(z|x)}\,dz \geq \int q_\phi(z|x) \log\left[\frac{p_\theta(x,z)}{q_\phi(z|x)}\right]dx$

↳ Jensen's Inequality

$= E_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{KL}[q_\phi(z|x) \| p(z)]$

↳ Reconstruction ↳ penalty

* Noisy gradients

- $\nabla_\phi E_q[\log p_\theta(x|z)] = \nabla_\phi \int q_\phi(z|x) \log p_\theta(x|z)\,dz$

$= \int \log p_\theta(x|z)\, \nabla_\phi q_\phi(z|x) \frac{q_\phi(z|x)}{q_\phi(z|x)}\,dz = \int q_\phi(z|x) \log p_\theta(x|z) \nabla_\phi \log q_\phi(z|x)\,dz$

$= E_q[\log p_\theta(x|z) \nabla_\phi \log q_\phi(z|x)]$

* Regularized Autoencoder (RAE)

- No noise injection
- deterministic autoencoder + explicit regularization for the decoder
- loss for RAE : $J_{RAE} = J_{Rec} + \beta J_z^{RAE} + \lambda J_{REG}$

  최소화   ↳ Explicit regularizer for the decoder

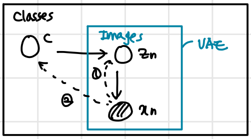
- example of J_{REG}

① Tikhonov regularization : $J_{REG} = \frac{1}{2}\|\theta\|_2^2$ (weight decay on decoder θ)

② Gradient penalty : $J_{REG} = \|\nabla D_\theta(E_\phi(x))\|_2^2$

③ Spectral normalization

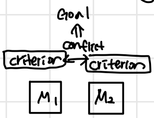
* Neural Statistician



① Standard inference network (VAE와 동일)
② static network (permutation-invariant model)

Ch11. Generative Adversarial Networks (GAN)

* Adversarial Training



- 2개 이상의 각각의 충돌되는 criterion을 갖는 모델들이 equilibrium을 이룰 때 Goal을 달성함.

* Training GAN

- Generator : $\min_G [E_{z \sim p(z)}[\log(1 - D(G(z;\theta)))]]$
- Discriminator : Generator의 성능을 알 수 있음. $x \to$ D $\to P(y=1|x)$ $y=1$: true, $y=0$: false

Data와 둘 거가 결정함

$\to \max_D [E_{x \sim p_d(x)}[\log D(x;\phi)] + E_{z \sim p(z)}[\log(1 - D(G(z;\theta)))]]$

* Generating images by GAN

- DC GAN : using 100 dimensional random noise vector
- progressive Growing of GANs : speed up + 덜 noisy 함
- GAN for single image super-resolution : upscaling을 사용해 더 선명함

* GAN의 한계

① Non convergence

② model collapsing

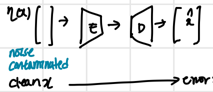
: Generator 입장에서는 Discriminator를 속이기만 하면 됨

→ 여러개의 mode 중 Discriminator를 잘 속이는 mode만 사용함 (다른 모드 사용X)

③ diminished gradient

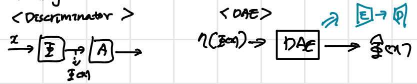
: Discriminator가 너무 뛰어나면 generator의 gradient가 사라짐

* Denoising Autoencoder



* GAN trained with Denoising Feature matching

- Denoising Autoencoder를 사용함



*	VAE	GAN
	likelihood-based	likelihood-free, Adversarial training
training	안정	어려움
Image	little blurry	sharp looking

* Semi-Supervised learning

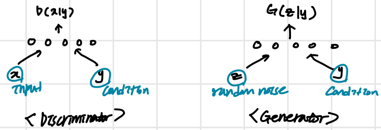
- labeled example이 적고 많은 unlabeled example이 있을 때 사용
- loss: $L_{supervised} + L_{unsupervised}$

  ↳ GAN 사용

* Conditional GAN

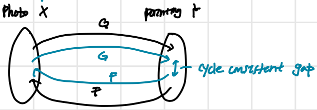
- generate fake sample with a condition y

→ generator를 control 할 수 있음



- ex: unpaired images to image translation
  (사진을 넣으면 특정 화가풍의 그림으로 바꿔줌)

→ Dataset이 pair되어있지 않아도 만들어 줘야함



: 2개의 GAN을 사용

* Adversarially learned Inference

- Inference Net $\xrightarrow{(x,\hat{z})}$ Discriminator $\xleftarrow{(\hat{x},z)}$ Generation Net
  <Encoder> <Decoder>
- 기존 GAN는 Decoder만 존재.

Ch12. Bayesian Optimization

*** Black box optimization**

: black box function이라면 어떤 function인지 알 수 없고 특정한 query에 대한 값만 알 수 있음 (이 값도 대부분 noisy)

*** Regret**

- instantaneous regret : best와 chosen의 차이
- cumulative regret
- simple regret
- cumulative regret의 linear보다 아래에 위치하면 no regret

*** Hyper parameter**

- 설정해야 하는 parameter (user가 설정) → 학습 전에 미리 설정

ex. learning rate, batch size, kernel size, # of layer...

- model parameter

: Data를 통한 training을 거쳐서 설정되는 것

ex. weight

- Hyper parameter를 찾는 법

① grid search

② random search

③ bayesian optimization

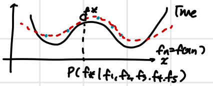
(↓ efficient ↑)

*** Auto ML**

- Feature processing
- Model / Algorithm selection
- Hyperparameter tuning

} 자동화

*** Gaussian Process Regression**



- 장점

① good uncertainty estimates

② choice of reasonable kernel can approximate broad class of f(x)

- 단점

① Not scalable

② limited by kernels

- kernel 고르는 법

① Gaussian kernel (squared exponential kernel)

② Matérn kernel

GP 이외에도 random forest, Neural network 사용 가능

- $\{(x_1, y_1), \cdots, (x_n, y_n)\}$

posterior mean $\bar{\mu}(x) = [k(x, x_1) \cdots k(x, x_N)] [K + \sigma^2 I]^{-1} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}$

↳ $\sum_{n=1}^{N} d_n k(x, x_n)$ ↳ $\begin{bmatrix} d_1 \\ \vdots \\ d_N \end{bmatrix}$

⇒ variance가 줄어듦 (uncertainty ↓)

*** Random forest**

- 빠르고 병렬학습 가능
- Poor extrapolation

*** Neural Network**

- size의 limitation 존재

*** Categorical / Integer-valued variable**

- Categorical : One-hot encoding
- Integer-valued : Rounding after optimizing the acquisition function

*** Acquisition function**

: 현재까지의 확률적 추정 결과를 바탕으로 최적의 input을 찾는데 가장 유용한 다음 input 후보를 추천하는 함수

① Probability of Improvement (PI)

② Expected Improvement (EI)

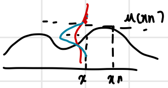
③ GP Upper Confidence Bound (GP-UCB)

] 가장 많이 사용 → gap이 크면 클수록 변화

① PI : $u(x, y : \xi) = \mathbb{I}(y > \alpha)$

→ y가 α보다 크면 1, 아니면 0

$PI(x) = P(f(x) \geq f(x_n))$



- 실제로는 slack variable ξ가 추가됨

→ Too big ξ: slow tuning, Too small: too narrow

② EI

$EI(x) = \int \max(y - y^*, 0) N(y | \bar{\mu}, \sigma^2) dy$

- PI와 마찬가지로 ξ 추가

$\therefore EI(x) = \begin{cases} (\mu(x) - \mu(x^*) - \xi)\Phi(z) + \sigma(x)\phi(z) & \text{if } \sigma(x) > 0 \\ 0 & \text{if } \sigma(x) = 0 \end{cases}$

$z = \begin{cases} \frac{\mu(x) - \mu(x^*) - \xi}{\sigma(x)} & \text{if } \sigma(x) > 0 \\ 0 & \text{if } \sigma(x) = 0 \end{cases}$

Ch13. Neural Process

- want : functional distribution을 찾는것

① Gaussian Process (GP)

: Nonparametric Bayesian method

→ data-efficient (데이터 수가 적어도 됨)

→ inference 필요

② Deep neural Network (DNN)

→ Data 많이 필요

→ GP보다 inference 측면에서 유리

③ Neural process

: Neural Network based probabilistic model

*** Generative Query Network**

: 서로 다른 view에서 본 사진 data가 들어오면 새로운 view (query)가 주어졌을 때 image를 generate함

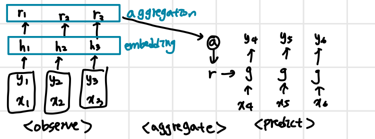
*** Conditional Neural process (CNP)**

: NN과 GP의 장점을 합침

	NN	GP	CNP
Can fit more than 1 function at test time	X	o	o
cheap	o	X	o

① GP처럼 stochastic process의 flexible

② NN처럼 data로부터 직접 gradient descent를 통해 train



ex. One-shot classification

- model : $Q_\theta(f(T) | O, T) \approx P(f(T) | O, T)$

O: Context points, T: target inputs

이때 Q_θ는 permutation invariant

↳ input의 순서와 관계없이 output 출력

- Architecture

: $r_n = h_\theta(x_n, y_n)$, $r = a(r_1, \ldots, r_N)$, $\phi_n = g_\theta(x_n, r)$

↳ NN의 output

⇒ ① regression : $\phi_n(\mu_n, \sigma_n^2)$

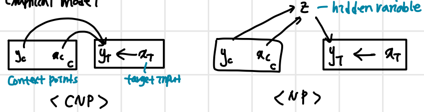
② classification : class probabilities 의 logits 가 output

*** Neural process**

: CNP의 확장

	NN	GP	CNP	NP
Can fit more than 1 function at test time	X	o	o	o
cheap	o	X	o	o
coherent function sampling	X	o	X	o

- Graphical model



① encoder : $r_n = h_\theta(x_n, y_n)$

② Aggregator : $r = a(r_1, r_2, \cdots r_N)$ 이 Gaussian RV z의 μ, σ 를 parameterize

↳ CNP와 추가된 부분

③ Conditional Decoder : $\phi_n = g_\theta(x_n, z)$