KCC 2019 tutorial review

오토인코더 딥 러닝 모델 기초 이론 및 활용 실습

강연자: 최재영 교수님(한국외대)
발표자: 남궁 영

강연자 소개

2011 KAIST 전기및전자공학과 박사
2008-2012 토론토대학 연구원
2012-2013 펜실베니아대학 연구원
2013-2014 삼성전자 책임연구원
2014-2016 중원대학교 의료공학과 조교수
2016-현재 한국외국어대학교 컴퓨터.전자공학부 조교수
관심분야 : 딥 러닝 기반 안면인식, 머신러닝, 패턴인식, 영상처리

최재영 교수
(한국외대)

- ❖ Prof. Jae Young Choi
- ❖ Pattern Recognition and Machine Intelligence Lab. (PMI)
- ❖ Hankuk University of Foreign Studies

강의 소개

강연제목

오토인코더 딥 러닝 모델 기초 이론 및 활용 실습

강연요약

딥 러닝 분야의 대표적인 학습모델 중 하나인 오토인코더를 강의한다. 오토인코더 개발 배경, 개념, 설계방법, 응용분야 등에 대해 수강생을 이해를 돕는 것을 목표로 한다. 오토인코더와 관련된 핵심 논문들을 분석하여 관련 이론을 수강생에게 이해시키고 텐서플로우, 케라스 딥 러닝 프레임워크에 기반한 구현실습을 수행하여 수강생들이 현업에서 활용할 수 있도록 한다.

시간별 강의계획

시간	주제	주요내용
1	오토인코더 이론 I	- 오토인코더 이해를 위한 배경 지식(비지도학습, 차원축소, 신경망) 강의 - 기본 오토인코더 이론 강의 (비지도학습, 차원축소, 신경망) 강의
2	오토인코더 이론 II	- 심층 오토인코더 이론 강의 - 디노이징(Denoising) 오토인코더 이론 강의
3	오토인코더 실습	- 심층 오토인코더 실습 강의 - 디노이징 오토인코더 실습 강의

Content

- Background-Unsupervised Learning
  - Unsupervised learning
  - Supervised learning
  - Why not exploit unlabeled data?
  - Unsupervised learning
  - Examples of unsupervised learning
  - unsupervised learning

- Motivation of Autoencoder
  - Dimensionality reduction
  - Reconstruction issue
  - Principal component analysis
  - PCA details
  - PCA reconstruction
  - Summary of PCA
  - Limitation of PCA
  - Alternative perspective
  - Introduction of Autoencoder
  - Alternative model – Autoencoder
  - Low-dimensionaity embedding with examples

- Basic(simple) Autoencoder Theory
  - Basic Autoencoder
  - General autoencoder vs. Linear autoencoder

| 이론 I

- Deep Autoencoders
  - Deep(stacked) Autoencoder
  - Layerwise training vs. Whole network Training

- Denoising Autoencoder
  - Denoising autoencoder – Introduction
  - Applying autoencoders to eliminating noise
  - Denoising autoencoder (DAE)
  - Denoising autoencoder – Notations
  - Denoising autoencoder – Manifold perspective
  - Perspectives on denoising autoencoders
  - Denoising autoencoder – Result
  - Denoising autoencoder – Concluding remarks

| 이론 II

- Basic and Deep Autoencoder 실습 (Tensorflow)
  - Basic autoencoder
  - Deep autoencoder
  - Denoising autoencoder

| 실습

Background-
Unsupervised Learning

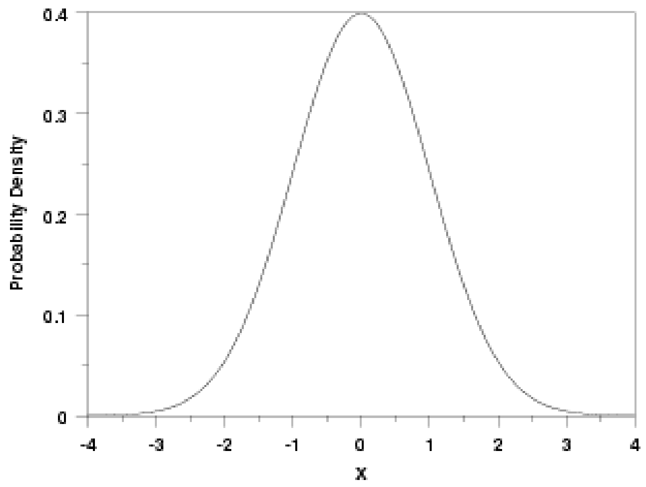
Unsupervised learning

❖ "Learning from unlabeled/unannotated data" (without supervision)

Training data $\{X_i\}_{i=1}^{n}$ ➡ **Learning algorithm** ➡ Prediction rule $\hat{f}_n$

❖ **What can we predict from unlabeled data?**

- Density estimation
- Groups or clusters in the data
- Low-dimensional structure
  - Principal Component Analysis (PCA)
  - Linear Discriminant Analysis(LDA)
  - Manifold learning(non-linear)



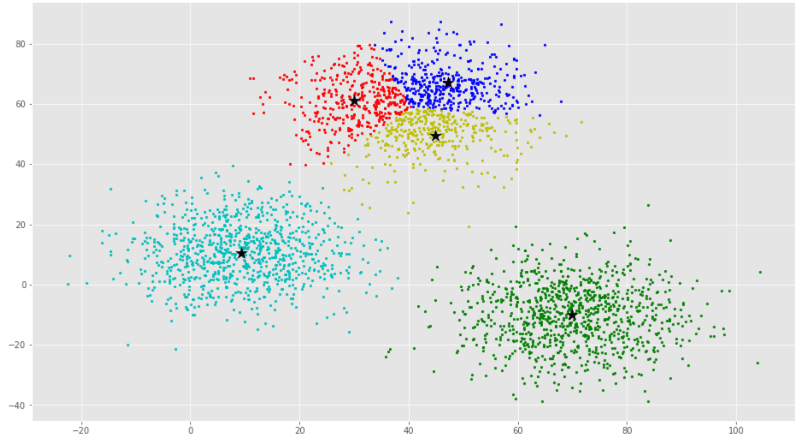
Unsupervised learning

❖ "Learning from unlabeled/unannotated data" (without supervision)

Training data $\{X_i\}_{i=1}^{n}$ → **Learning algorithm** → Prediction rule $\hat{f}_n$

❖ What can we predict from unlabeled data?

- Density estimation
- Groups or clusters in the data
- Low-dimensional structure
  - Principal Component Analysis (PCA)
  - Linear Discriminant Analysis(LDA)
  - Manifold learning(non-linear)



Unsupervised learning

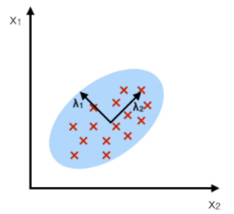
❖ “Learning from unlabeled/unannotated data” (without supervision)

Training data $\{X_i\}_{i=1}^{n}$ → **Learning algorithm** → Prediction rule $\hat{f}_n$

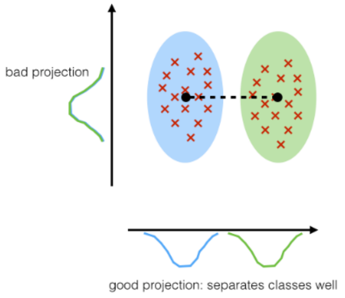
❖ What can we predict from unlabeled data?

- Density estimation
- Groups or clusters in the data
- Low-dimensional structure
  - Principal Component Analysis (PCA)
  - Linear Discriminant Analysis(LDA)
  - Manifold learning(non-linear)

PCA:
component axes that
maximize the variance



LDA:
maximizing the component
axes for class-separation



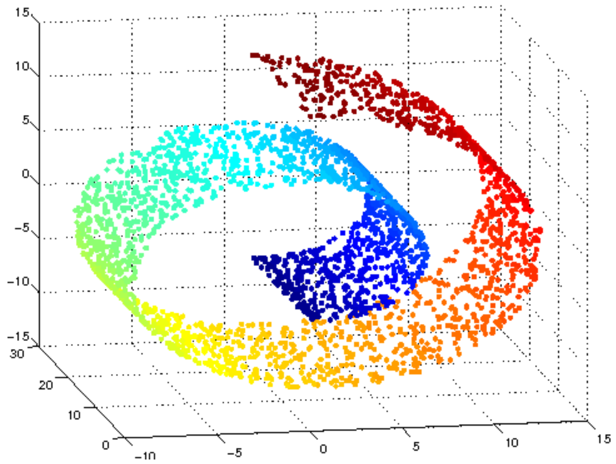
Unsupervised learning

❖ **"Learning from unlabeled/unannotated data" (without supervision)**

Training data $\{X_i\}_{i=1}^{n}$ → **Learning algorithm** → Prediction rule $\hat{f}_n$

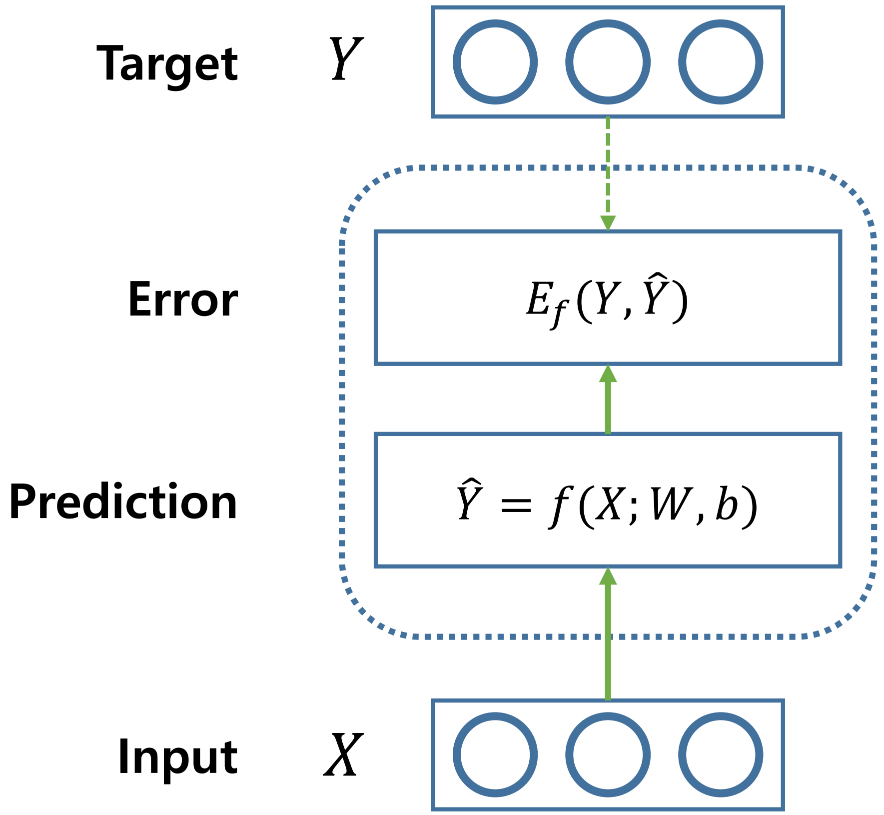
❖ **What can we predict from unlabeled data?**

- Density estimation
- Groups or clusters in the data
- Low-dimensional structure
  - Principal Component Analysis (PCA)
  - Linear Discriminant Analysis(LDA)
  - Manifold learning(non-linear)

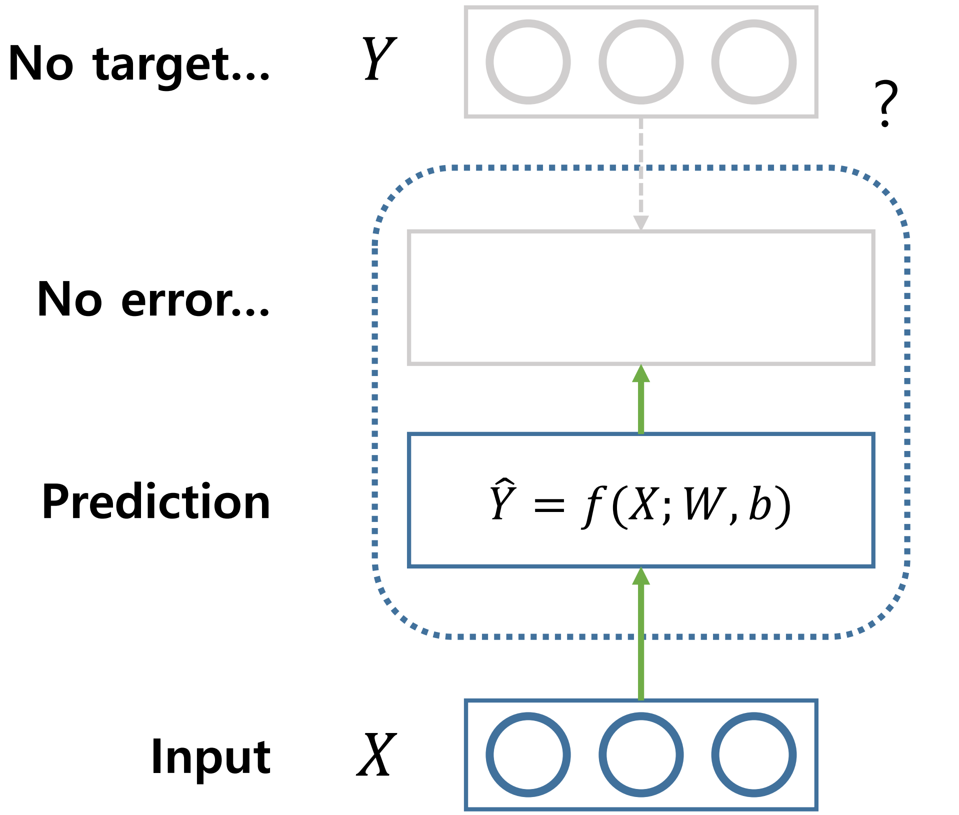


Supervised / Unsupervised learning

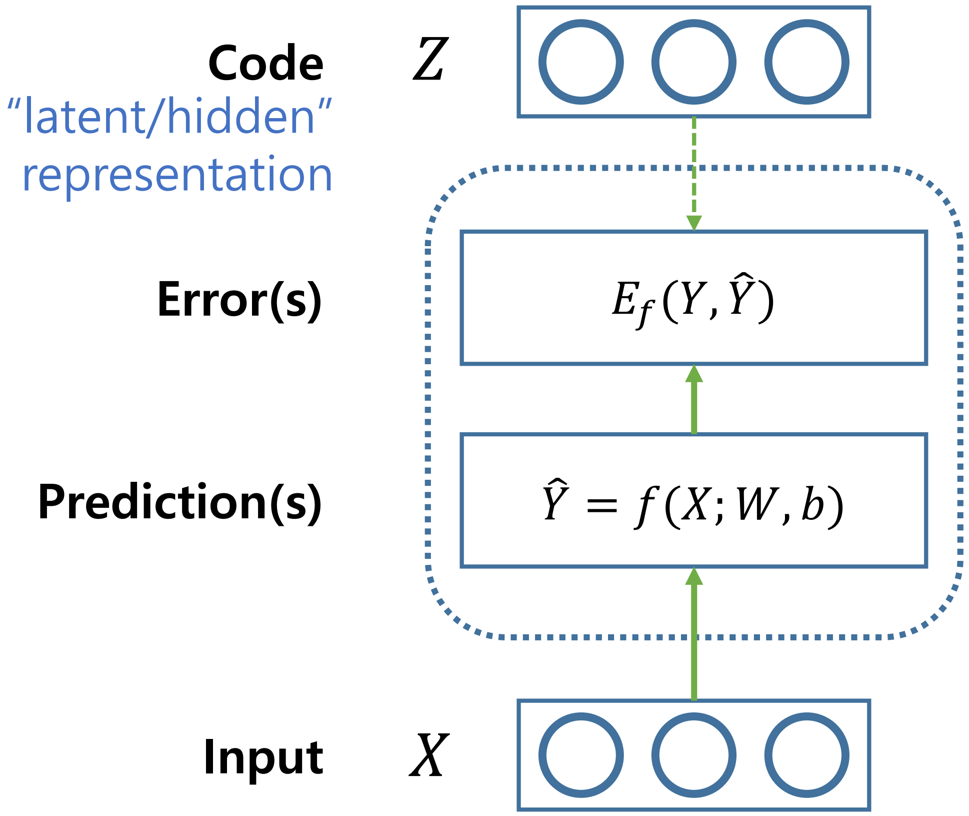
Supervised



Unsupervised



Unsupervised learning



- ❖ We want the **codes** to represent the **inputs** in the dataset.
- ❖ The **code** should be a compact representation of the inputs: low-dimensional and/or sparse.

Examples of unsupervised learning

❖ Linear decomposition of the inputs:

- principal Component Analysis and Singular Value Decomposition
- Independent Component Analysis [Bell & Sejnowski, 1995]
- **Sparse coding** [Olshausen & Field, 1997]
- …

❖ Fitting a distribution to the inputs:

- Mixtures of Gaussians
- Use of **Expectation-Maximaization algorithm** [Dempster et al, 1977]
- …

Unsupervised learning

❖ What's the point if we don't have labels?

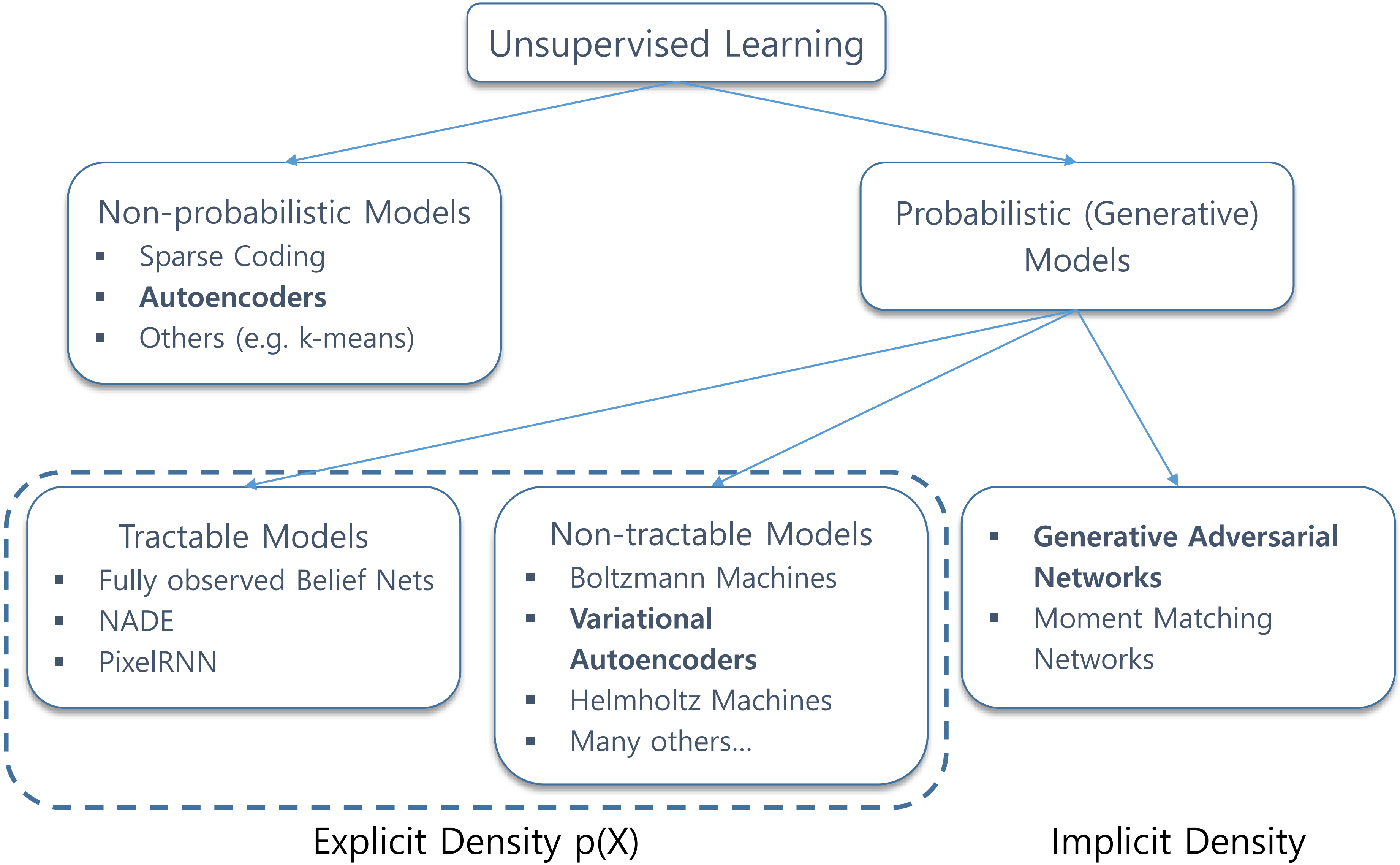
- As in the above example, unsupervised learning can often be a *precursor to supervised learning*, if we don't even know that the labels should be (e.g. disease subtypes)!

- Often vastly increases the amount of data available! Obtaining labelled data is not always:

- Can aid better *dimensionality reduction*, simplifying the work of other algorithms, allow for synthesizing new training data… and much more.

- Humans are essentially learning (mostly) unsupervised!

Unsupervised learning



Motivation of Autoencoder

Dimensionality reduction

- We now focus on the general unsupervised problem of dimensionality reduction – finding a way to appropriately compress out input into a useful "bottleneck" vector of smaller dimensionality (we often call this algorithm an encoder).

- Obvious application to supervised learning: *feeding the output of the bottleneck* into a simple classifier (e.g. k-NN, SVM, logistic regression…), perhaps fine-tuning the encoder as well.

- Fundamentally, dimensionality reduction (along with appropriate interpretability) is the essence of unsupervised learning – to compress data well, one must first understand it!

Dimensionality reduction

Dimensionality Reduction

Linear

- Principal Component Analysis (PCA)
- Linear Discriminant Analysis (LDA)
- etc..

Non-linear

- Autoencoders (AE)
- t-distributed stochastic neighbor embedding (t-SNE)
- Isomap
- Locally-linear embedding (LLE)
- etc..

Reconstruction issue

- In absence of any other information (that would be contained in labels), the best notion of "usefulness" for the bottleneck is our ability to reconstruct the input from it.

- Broadly speaking, we aim to specify two transformations:
  - The encoder ~ enc: $X \to Z$
  - The decoder ~ dec: $Z \to X$

  where X and Z are the input and code spaces, respectively (these are often simply $\mathbb{R}^n$ and $\mathbb{R}^m$ with n > m)

- Then we seek to find parameters of the encoder/decoder that minimize the reconstruction loss:

$$L(\vec{x}) = ||dec\big(enc(\vec{x})\big) - \vec{x}||^2$$

Principal component analysis

- Perhaps the simplest instance of this framework is the principal component analysis (PCA) algorithm.

- Encode by projecting the n-dimensional data onto a set of m orthogonal axes ($n \geq m$)

- To preserve the most information, always choose one of the axes to be the direction in which the dataset has the highest variance!

- Preserve m axes with highest variance.

PCA details

- Since projection onto orthogonal axes is a linear operation, the PCA encoder can be seen as simple matrix multiplication:

$$enc(\vec{X}) = W\vec{X}$$

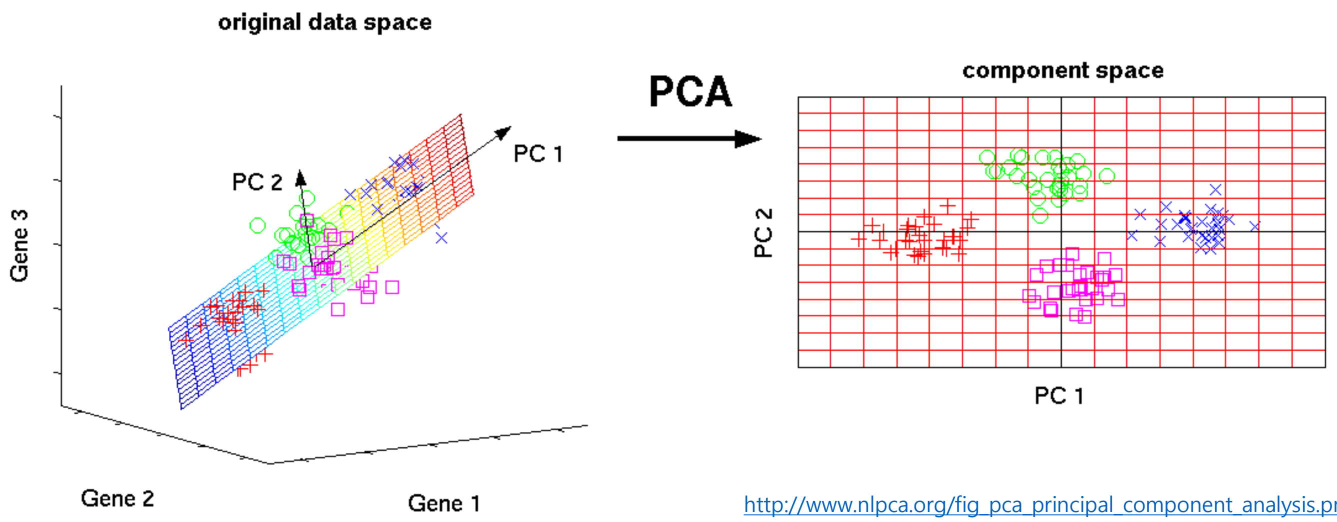
  where **W is of size** $m \times n$.

- As this is an orthogonal transformation, its inverse (along retained axes only) is its matrix's transpose:

$$dec(\vec{Z}) = W\vec{Z}$$

- We therefore seek to choose W to minimize $||\vec{X} - W^T W\vec{X}||^2$.
  Can solve this explicitly (suing eigenvalue analysis)!

Summary of PCA

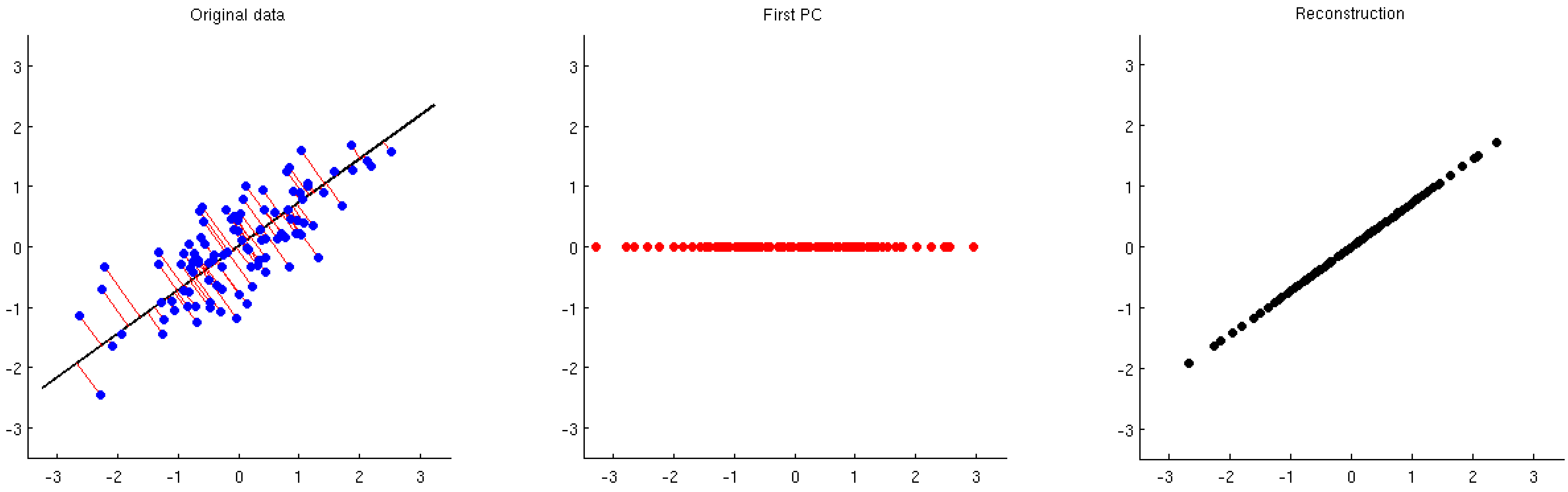


http://www.nlpca.org/fig_pca_principal_component_analysis.png

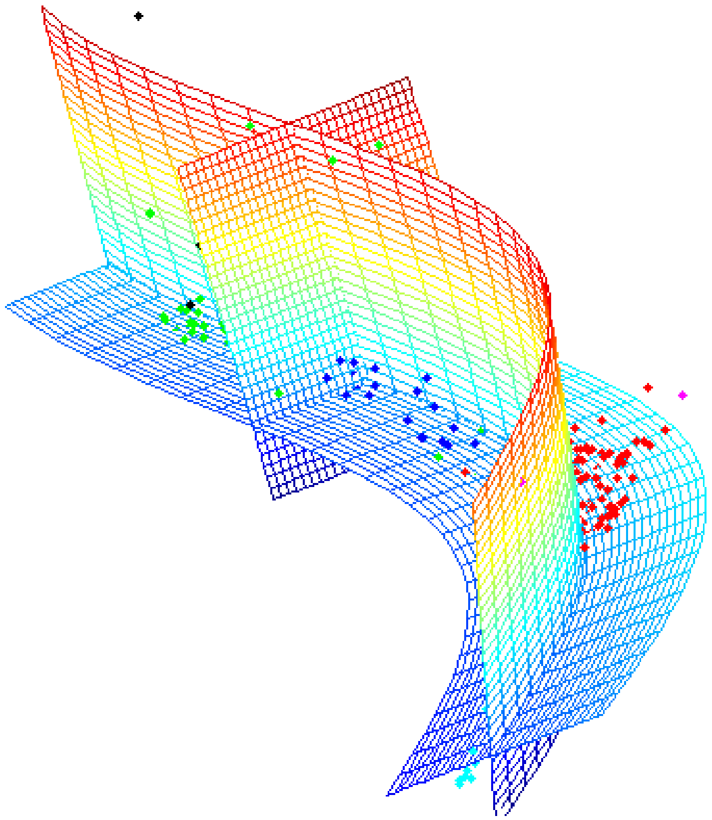
- **Finds k directions in which data has highest variance**
  - Principal directions (eigenvectors) W
- **Projecting inputs x on these vectors yields reduced dimension representation (&decorrealted)**
  - Principal components
  - $h = f_\theta(x) = W(x - \mu)\ with\ \theta = \{W, \mu\}$

- **Why mention PCA?**
  - Prototypical unsupervised representation learning algorithm
  - Related to autoencoders
  - Prototypical manifold modeling algorithm

PCA Reconstruction



Limitation of PCA



Linear model $\Rightarrow$ Incapable of capturing nonlinear manifolds!

Alternative perspective

- It should be simple to relate the operations of PCA to those of a two-layer fully-connected neural network without activations!

- This would allow us to work in exactly the same scenario, but train using backpropagation!

$$\vec{Z} = enc(\vec{X}) = W_1\vec{X} + \vec{b}_1$$
$$\vec{X} = dec(\vec{Z}) = W_2\vec{Z} + \vec{b}_2$$

- Once again, we optimize the reconstruction loss

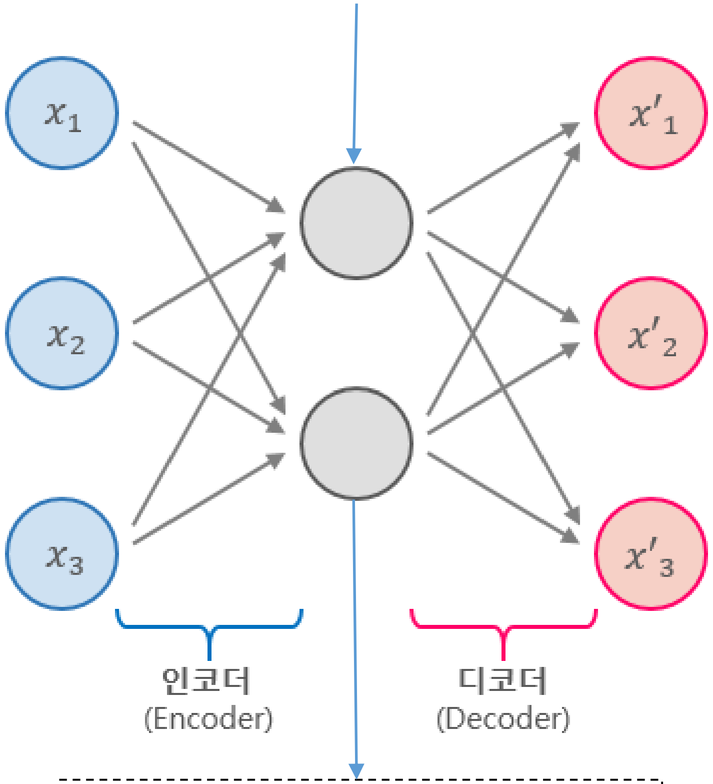
$$L(\vec{X}') = ||\vec{X}' - \vec{X}||^2$$

- We have just built out first autoencoder!

Introduction of Autoencoder

Bottleneck hidden layer

Input layer



output layer
reconstruct input

Autoencoders

= auto-associators
= diabolo networks
= sandglass-shaped net

- Code
- [Latent variable]
- Feature
- Hidden representation

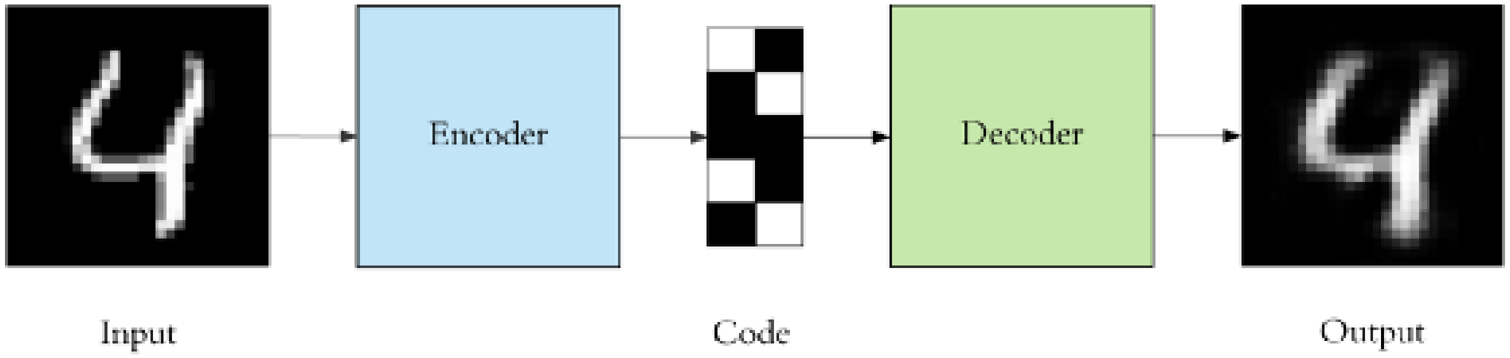
image | https://excelsior-cjh.tistory.com/187

| http://videolectures.net/deeplearning2015_Vincent_autoencoders/?q=Vincent%20autoencoder

Introduction of Autoencoder

❖ **Encoder**: To produce the low-dimensional embedding or code

❖ **Decoder**: To invert low-dimensional embedding to reconstruct the input

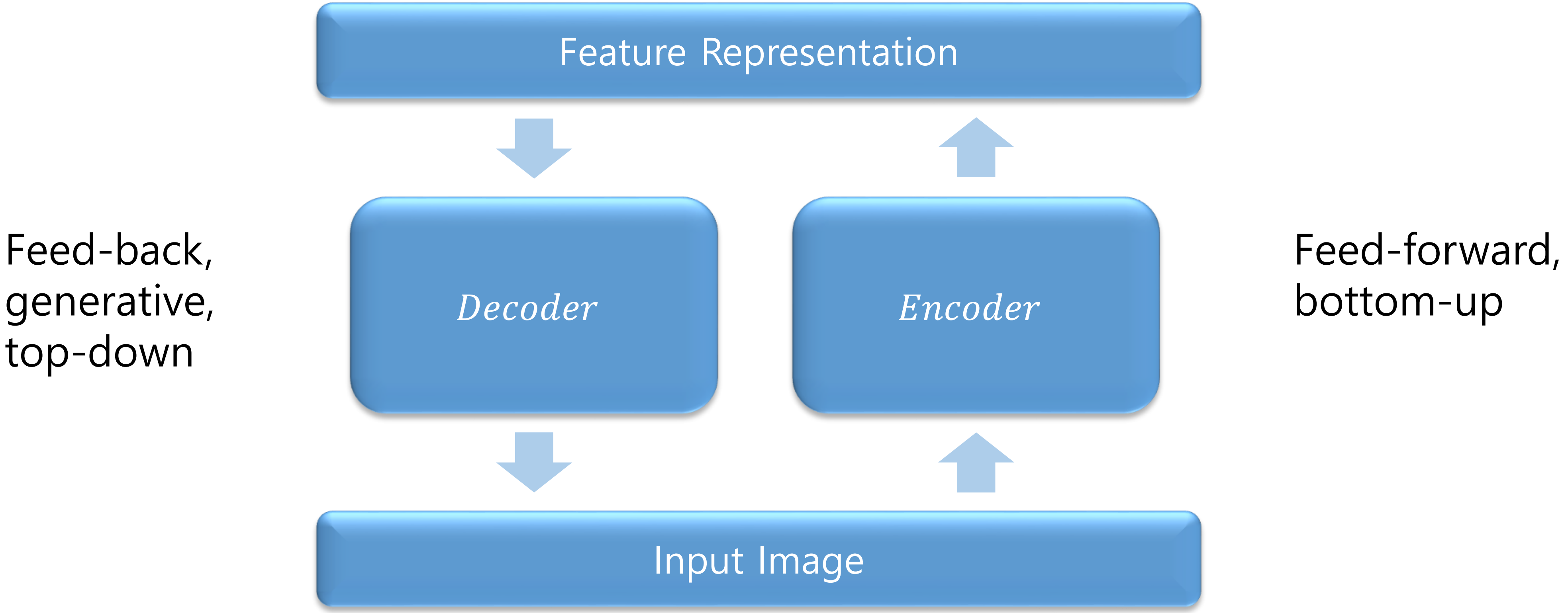


Embedding

image | https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798

Basic(simple) Autoencoder Theory

Basic autoencoder

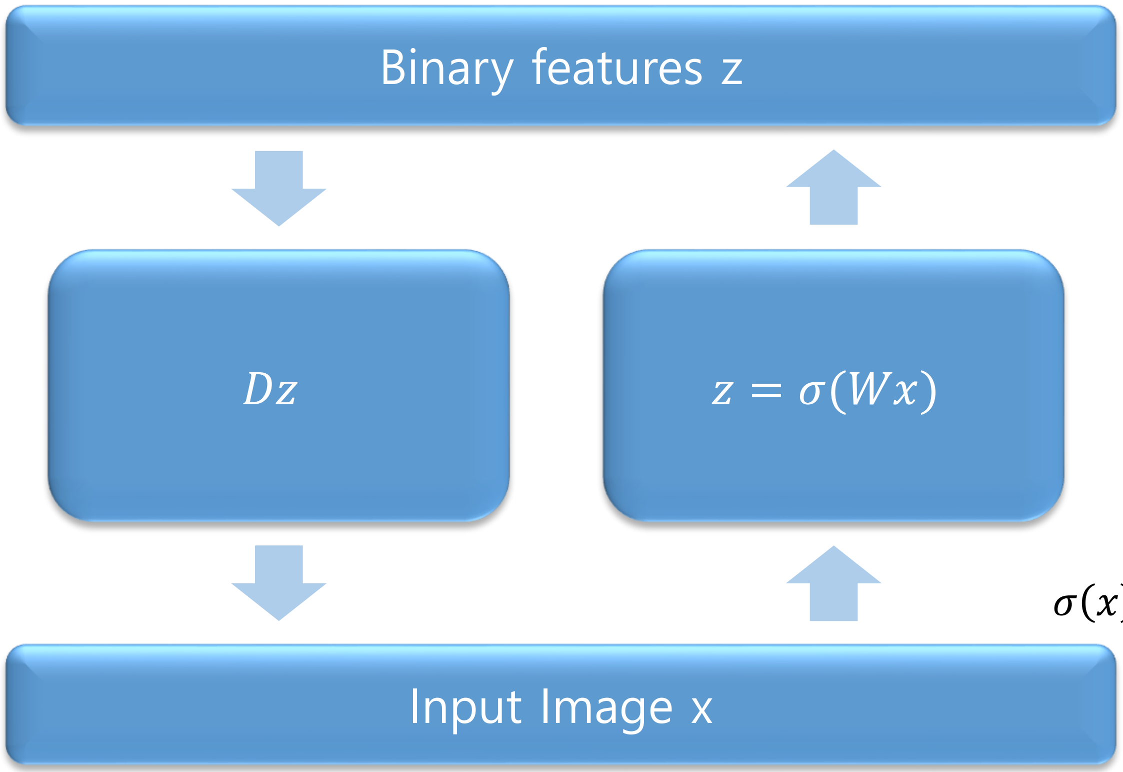


- **Details of what goes insider the encoder and decoder matter!**
- **need constraints to avoid learning an identity.**

Basic autoencoder

Decoder filters D

Linear function

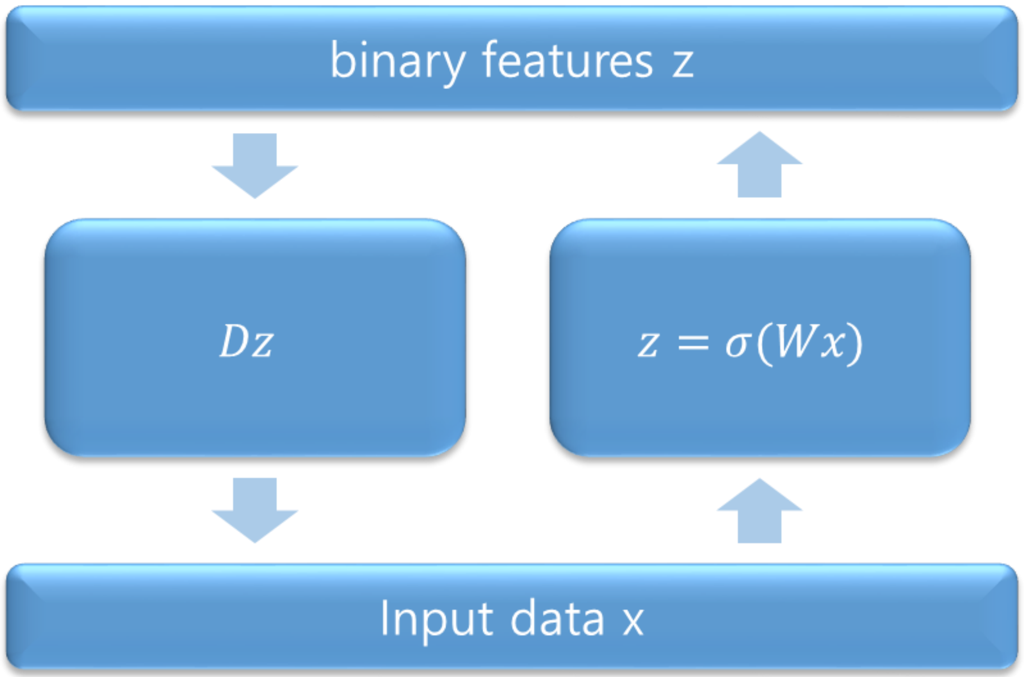


Encoder filters W.

Sigmoid function

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

Basic autoencoder

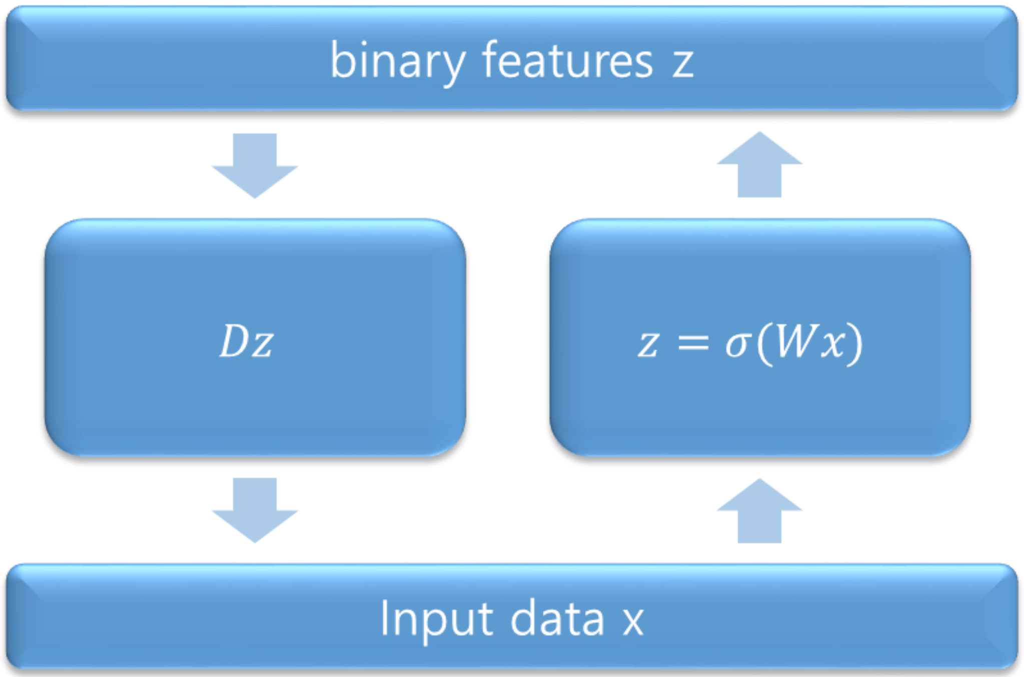


- An autoencoder with D inputs, D outputs, and K hidden units, with K<D.
- Given an input x, its reconstruction is given by:

$$y_j(x,W,D) = \underbrace{\sum_{k=1}^{K} D_{jk}}_{\text{Decoder}}\, \underbrace{\sigma\left(\sum_{i=1}^{D} W_{ki}x_i\right)}_{\text{Encoder}}, i = 1, \dots, D$$

$$y_j = \sum_{k=1}^{K} D_{jk}\, z_k \qquad z_k = \sigma\left(\sum_{i=1}^{D} W_{ki}x_i\right)$$

Basic autoencoder

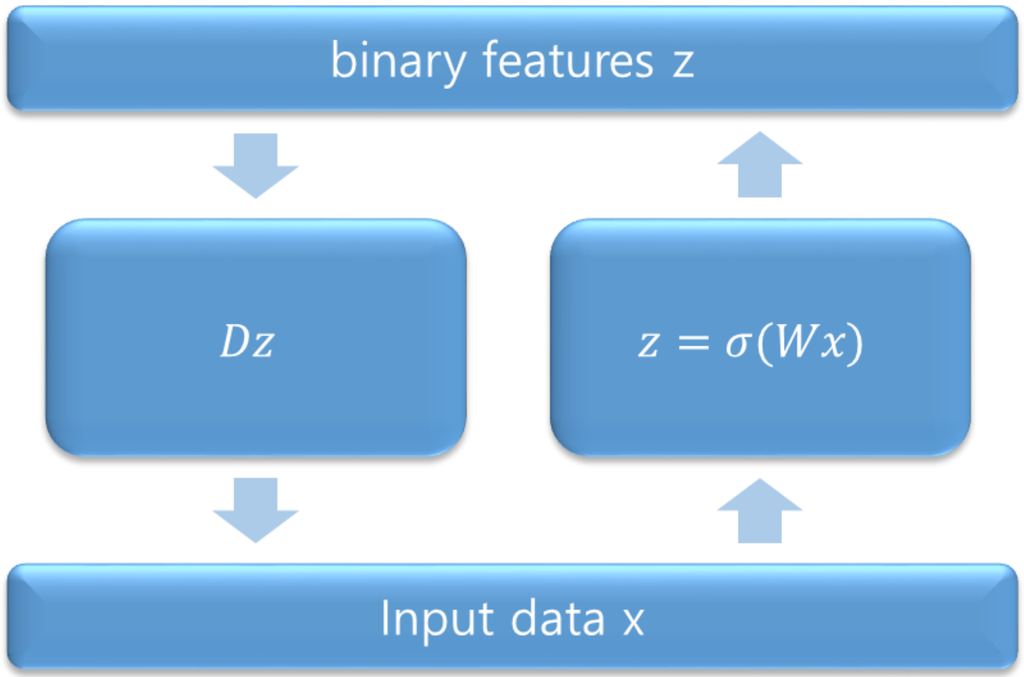


- An autoencoder with D inputs, D outputs, and K hidden units, with K<D.

- We can determine **the network parameters W and D** by minimizing the reconstruction error:

$$E(W,D) = \frac{1}{2}\sum_{n=1}^{N} ||y(x_n, W, D) - x_n||^2$$

Basic autoencoder



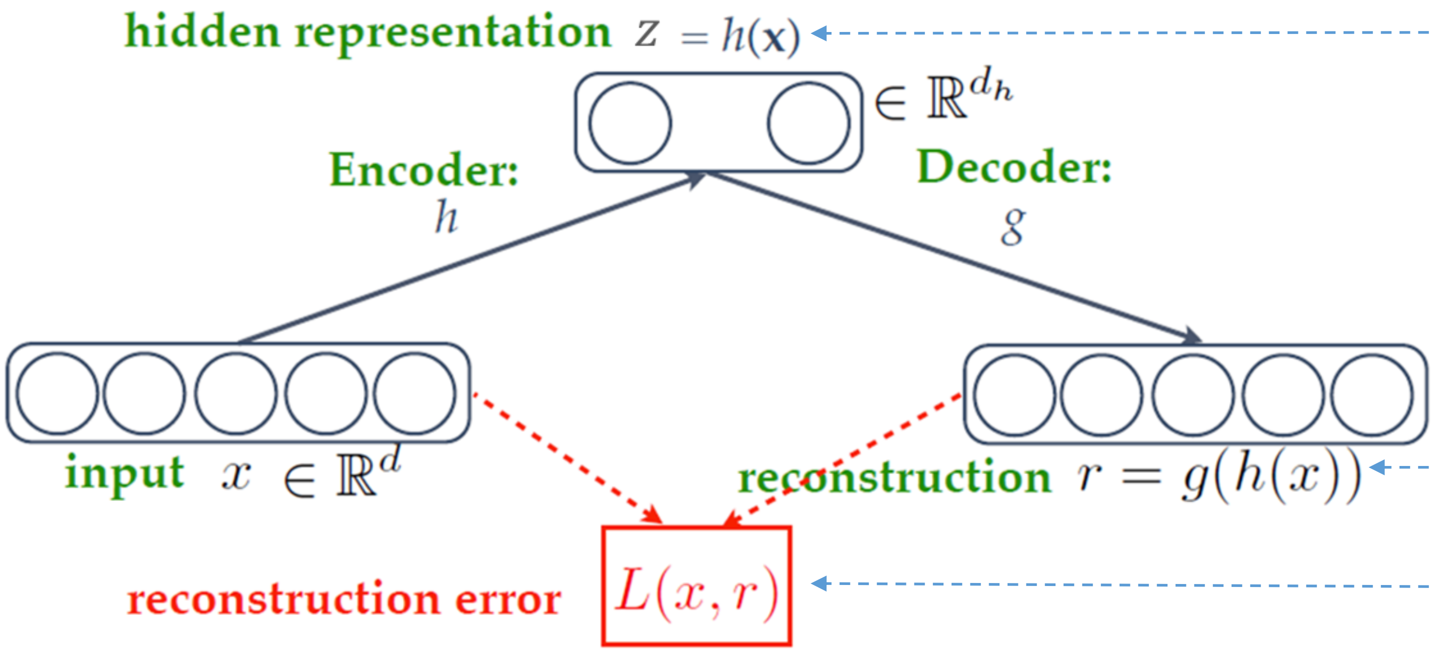
- If the hidden and output layers and linear, it will learn hidden units that are a linear function of the data and minimize the squared error.

- The K hidden units will span the same space as the first k principal components. The weight vectors may not be orthogonal.

- With nonlinear hidden units, we have a nonlinear generalization of PCA.

General Autoencoder vs. Linear autoencoder

General Autoencoder



Minimize

$$\mathcal{J}_{\text{AE}} = \sum_{x \in D} L(x, g(h(x)))$$

Linear Autoencoder

$h(x) = W_e x + b_e$

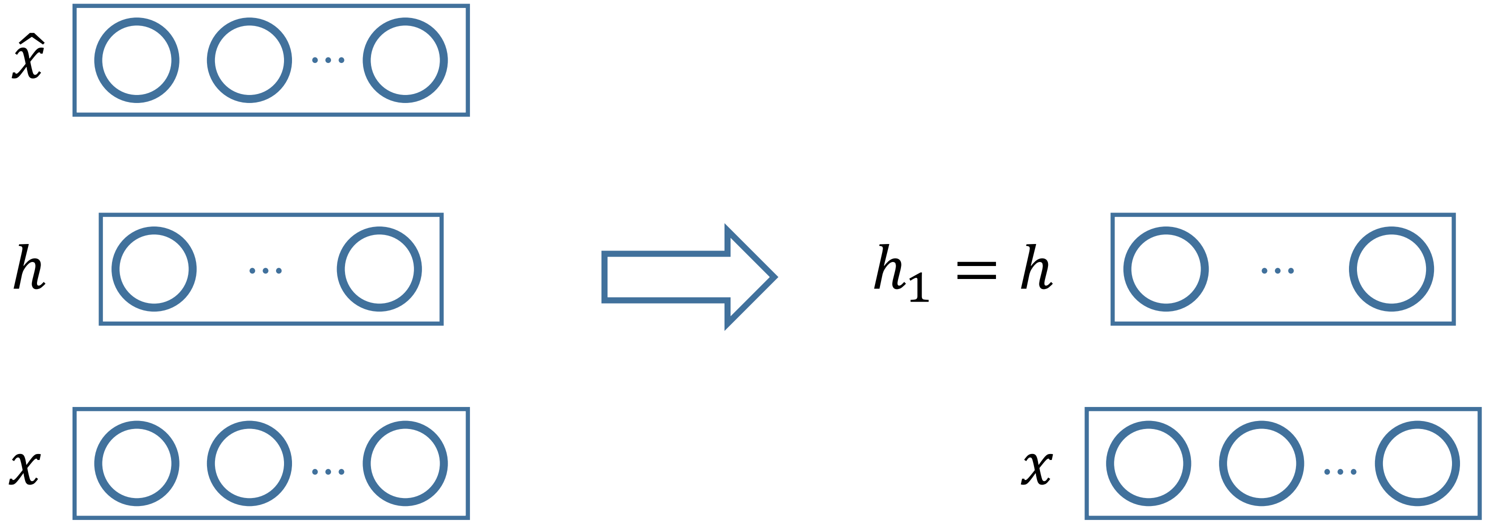
$g(h(x)) = W_d z + b_d$

$||x - y||^2$ or cross-entropy

Hidden layer가 1개이고 layer 간
fully-connected로 연결된 구조

Deep Autoencoders

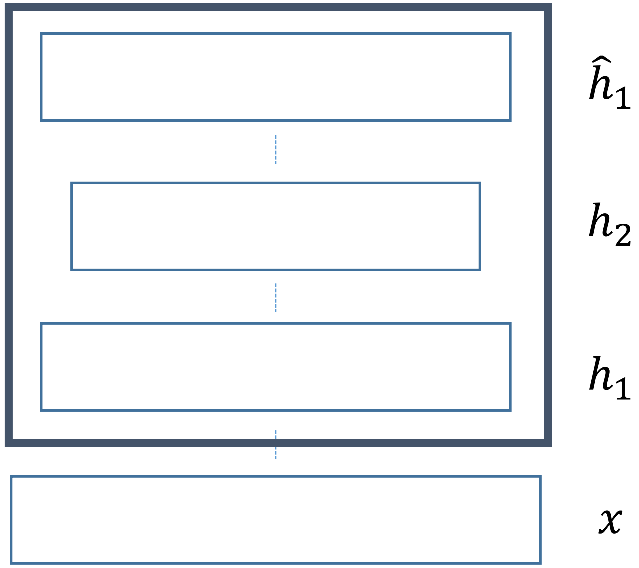
Deep (stacked) autoencoder



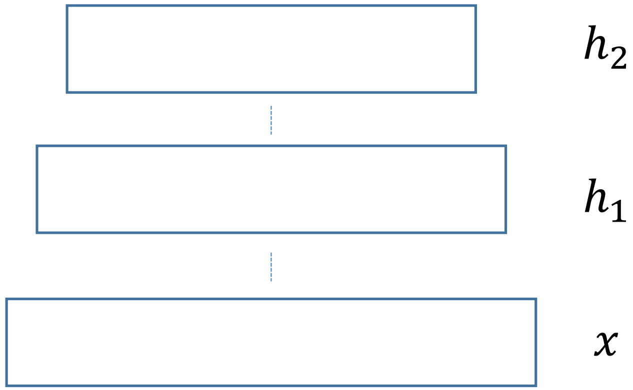
Remove top layer

- The simple autoencoder can be stacked to produce a **deep autoencoder**
- The top layer of the simple autoencoder is removed while the bottom and the middle layers remain intact

Deep (stacked) autoencoder



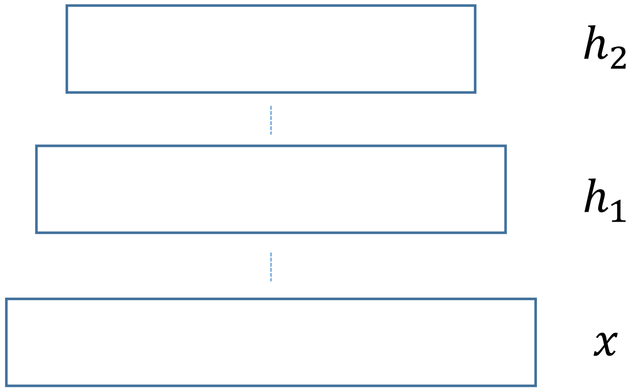
New autoencoder



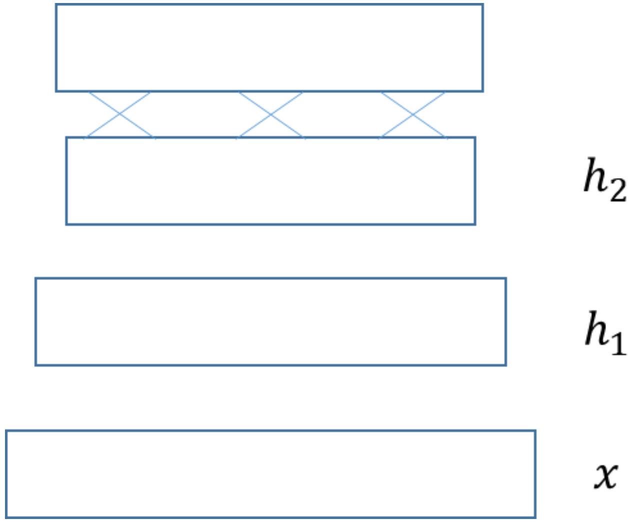
Remove top layer

- The three layers, h_1, h_2, and $\hat{h}_1$, can be regarded as another simple autoencoder on its own right and is trained likewise.
  In particular, given an original input $x^{(i)}$, h_1 is now regarded as the input for the newly created simple autoencoder $h_1 - h_2 - \hat{h}_1$
- Remove the top layer $\hat{h}_1$ and keep h_2. We now have a three-layer neural network consisting of x, h_1, h_2
- The connection matrix between h_1 and h_2 is the one gotten from the simple autoencoder $h_1 - h_2 - \hat{h}_1$
- The connection matrix between x and h_1 is the one gotten from the simple autoencoder $-h_2 - \hat{x}_1$

Deep (stacked) autoencoder



keep adding new layers



Classifier

Everything was done using only the input x. Since we have no need for the label y, what we have done so far can be dubbed **unsupervised learning**

After this deep autoencoder is constructed, we can now put it to used for classification task

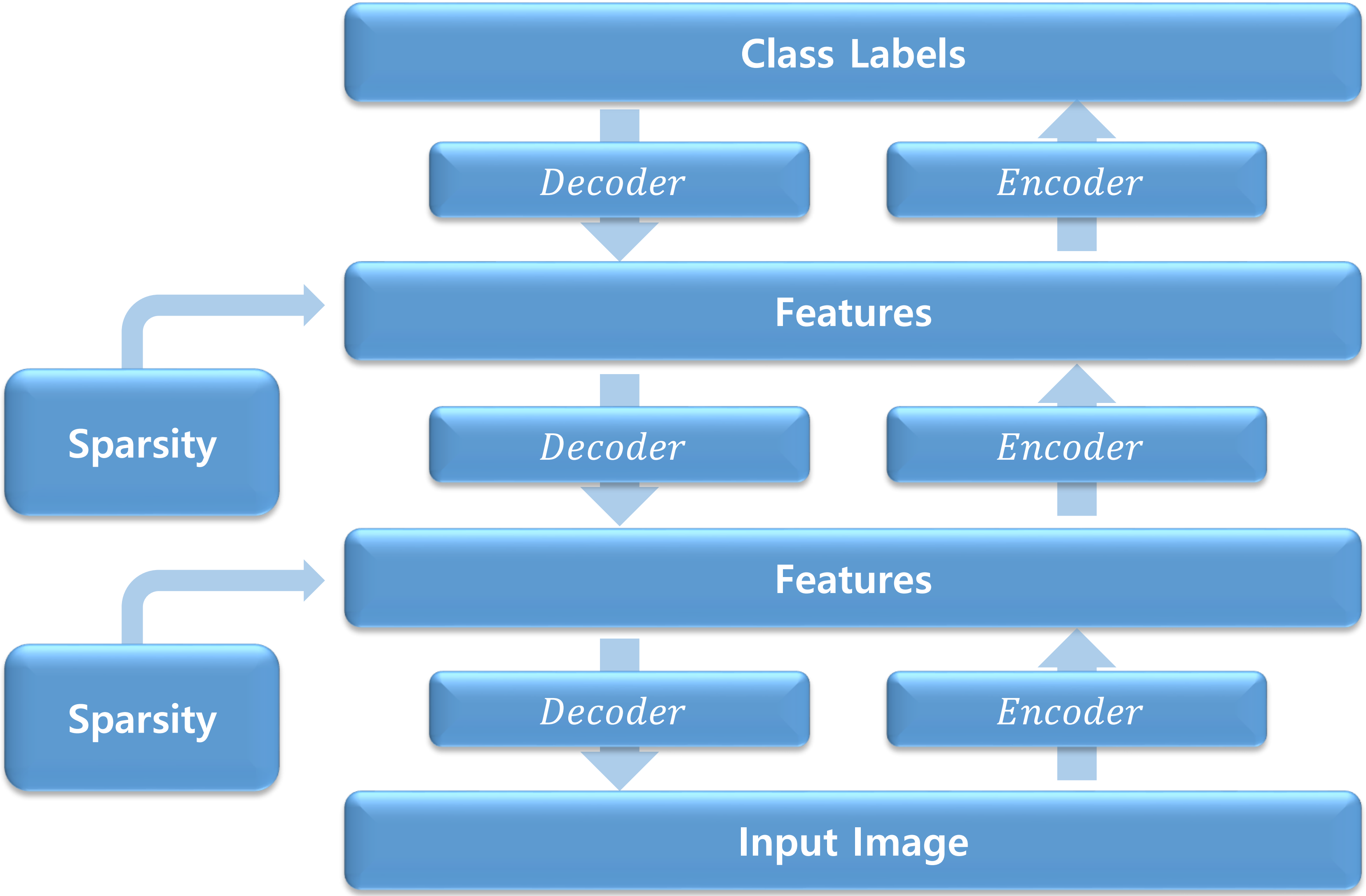
Softmax layer put on top of the deep autoencoder given

Deep (stacked) autoencoder

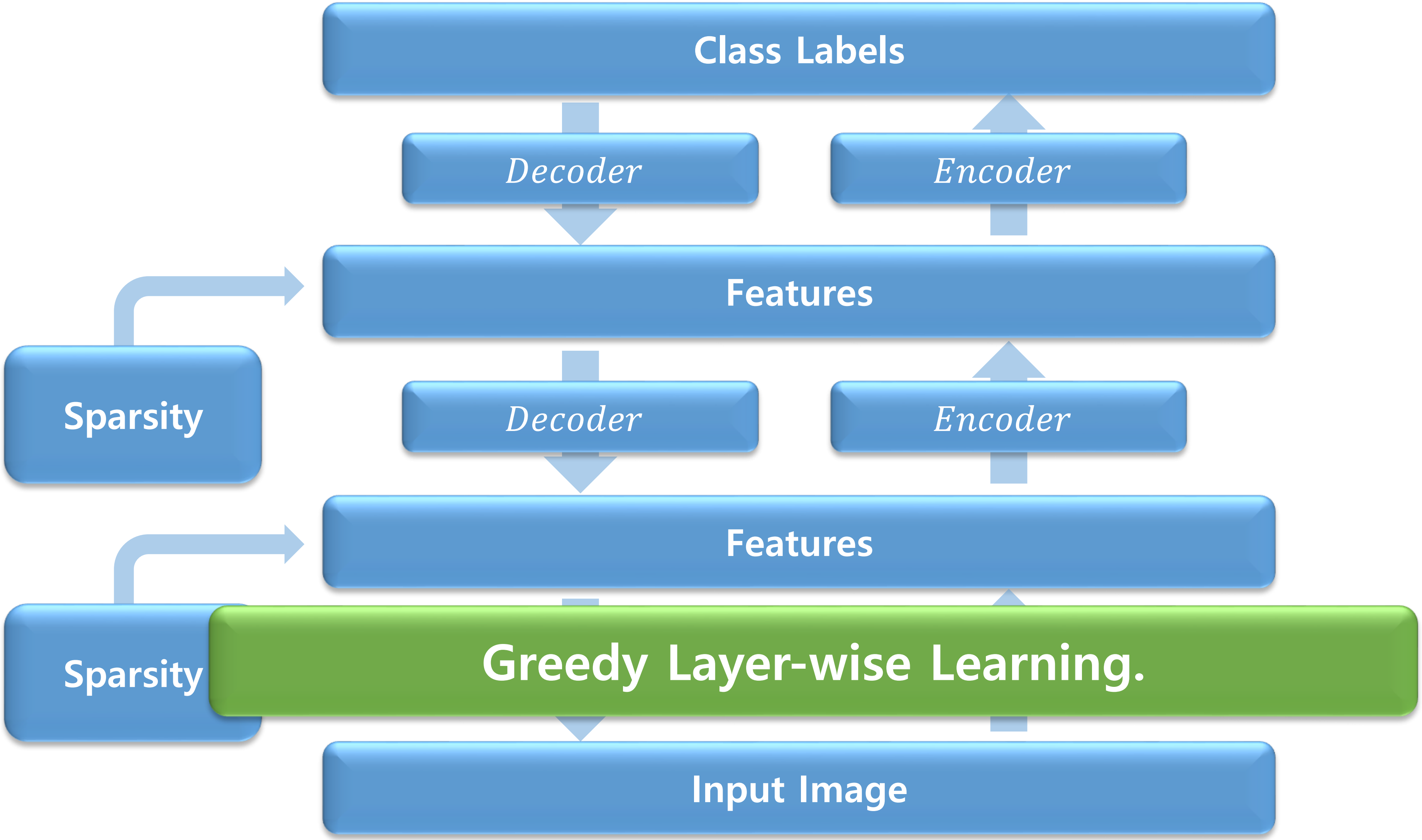
- If the deep autoencoder is trained in such a way to act as a good encoder, the information loss should be small enough through this compression process so that reasonable variations in the input data do not result in adverse effects.

- Therefore, the weights of the deep autoencoder should be good initial weights for the classification problem at hand. This is one of the reasons why deep autoencoder works so effectively.

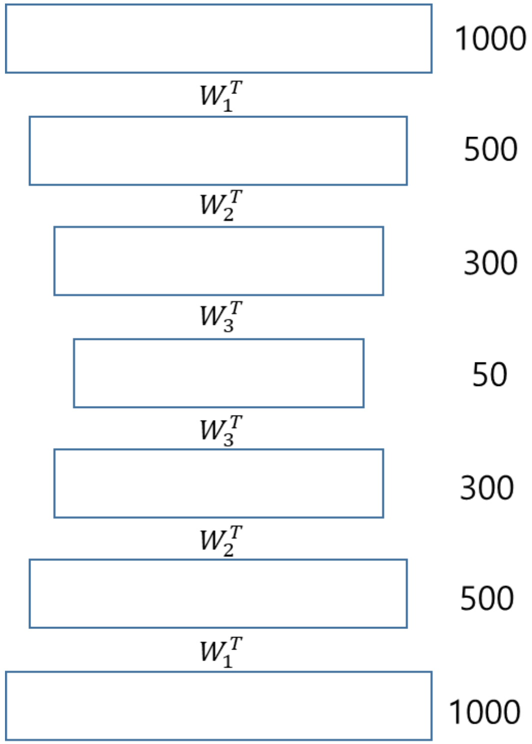
Deep autoencoder



Deep autoencoder



Layerwise training vs. Whole network training



Stack them all at once

- Goal and the construction of the autoencoder is only a preparatory step (Layerwise training).
- There is another way to train autoencoders. In here, instead of building up layer-by-layer, we put up the whole structure at one and train the whole network (find good connection weights) by minimizing the $L^2 - error$ between the input x at the bottom layer and the output $\hat{x}$ at the top layer.
- This is a feasible problem because nowadays training deep neural networks can be routinely done.

Denoising Autoencoder

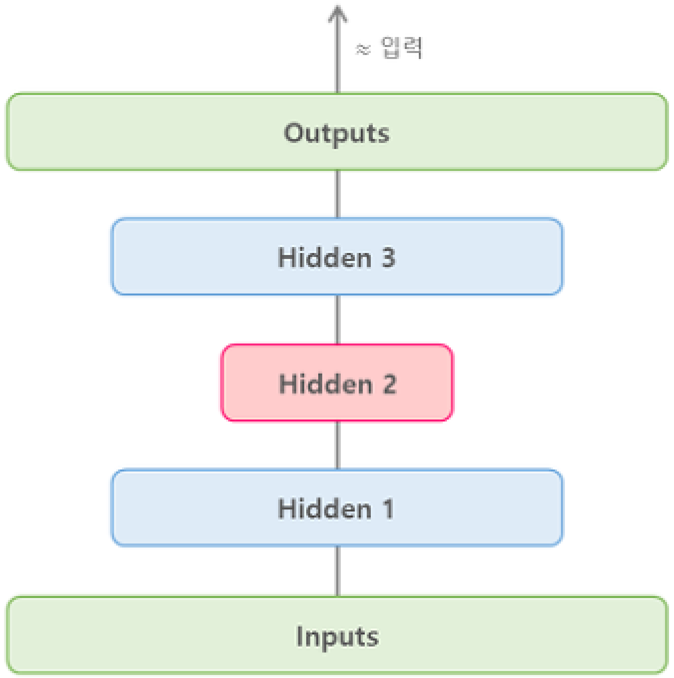
Denoising autoencoder - Introduction

- Unsupervised learning based on the idea of making the learned representations robust to partial corruption of the input pattern
- This approach can be used to train autoencoders, and theses denoising autoencoders can be stacked to initialize deep architectures
- The algorithm can be motivated from a manifold learning
- Showing the surprising advantage of corrupting the input of autoencoders on a pattern classification benchmark suite

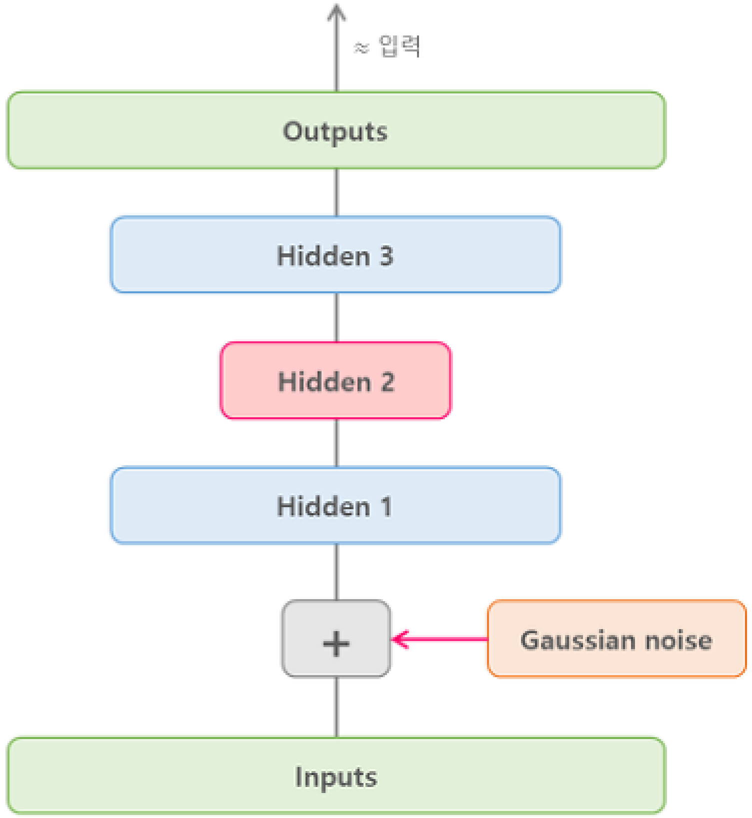
Applying autoencoders to eliminating noise

- **This comes from several motivations:**
  - Real-word data is often noisy;
  - Combatting overfitting the training data;
  - Learning more robust representations!

Denoising qutoencoder (DAE)



< stacked autoencoder >



< denoising autoencoder >

image | https://excelsior-cjh.tistory.com/187(modified)

Denoising autoencoder - Notations

- Autoencoder takes an input vector $\boldsymbol{x} \in [\mathbf{0},\mathbf{1}]^{\boldsymbol{d}}$, and first maps it to a hidden representation $\boldsymbol{y} \in [\mathbf{0},\mathbf{1}]^{\boldsymbol{d}'}$ through a deterministic mapping $\boldsymbol{y} = \boldsymbol{f_\theta}(\boldsymbol{x}) = \boldsymbol{s}(\boldsymbol{W_x} + \boldsymbol{b})$, parameterized by $\boldsymbol{\theta} = \{\boldsymbol{W},\boldsymbol{b}\}$. $\boldsymbol{W}$ is a $\boldsymbol{d} \times \boldsymbol{d}'$ weight matrix and $\boldsymbol{b}$ is a bias vector.
- The resulting latent representation $\boldsymbol{y}$ is then mapped back to a "reconstructed" vector $\boldsymbol{z} \in [\mathbf{0},\mathbf{1}]^{\boldsymbol{d}}$ in input space $\boldsymbol{z} = \boldsymbol{g_{\theta'}}(\boldsymbol{y}) = \boldsymbol{s}(\boldsymbol{W}'\boldsymbol{y} + \boldsymbol{b}')$ with $\boldsymbol{\theta}' = \{\boldsymbol{W}',\boldsymbol{b}'\}$.
- Each training $\boldsymbol{x}^{(\boldsymbol{i})}$ is thus mapped to a corresponding $\boldsymbol{y}^{(\boldsymbol{i})}$ and a reconstruction $\boldsymbol{z}^{(\boldsymbol{i})}$.
- The parameters of this model are optimized to minimize the $\boldsymbol{average\,reconstruction\,error}$:

$$\theta^*\theta'^* = \underset{\theta,\theta'}{\operatorname{argmin}} \frac{1}{n}\sum_{i=1}^{n} L\left(x^{(i)}, z^{(i)}\right)$$

$$= \underset{\theta,\theta'}{\operatorname{argmin}} \frac{1}{n}\sum_{i=1}^{n} L\left(x^{(i)}, g_{\theta'}\left(f_\theta\left(x^{(i)}\right)\right)\right)$$

Where $\boldsymbol{L}$ is a loss function such as the traditional $\boldsymbol{squared\,error}$ $\boldsymbol{L}(\boldsymbol{x},\boldsymbol{z}) = ||\boldsymbol{x} - \boldsymbol{z}||^2$

Denoising autoencoder

❖ **An alternative loss, suggested by the interpretation of x and z as either bit vectors or vectors of bit probabilities (Bernoullis) is the** $\boldsymbol{reconstruction}$ "$\boldsymbol{cross - entropy}$“:

$$L(x,z) = H(\beta_x || \beta_z)$$

$$= -\sum_{k=1}^{d} [x_k \log z_k + (1 - x_k) \log(1 - z_k)]$$

Denoising autoencoder

- To test our hypothesis and enforce robustness to partially destroyed inputs we modify the basic autoencoder we just described.

- This is done by first corrupting the initial input x to get a partially destroyed version x by means of a stochastic mapping $\tilde{x} \sim q_D(\tilde{x}|x)$.

- Parameterized by the desired proportion v of "destruction": for each input x, a fixed number vd of components are chosen at random, and their value is forced to 0, while the others are left untouched.

Denoising autoencoder

- The corrupted input $\tilde{x}$ is then mapped, as with the basic autoencoder, to a hidden representation $y = f_\theta(\tilde{x}) = s(W\tilde{x} + b)$

- Reconstruct a $z = g_{\theta'}(y) = s(W'y + b')$

- As before the parameters are trained to minimize the average reconstruction error $L_{IH}(x, z) = H(\beta_x || \beta_z)$ over a training set, i.e. to have z as close as possible to the uncorrupted input x.

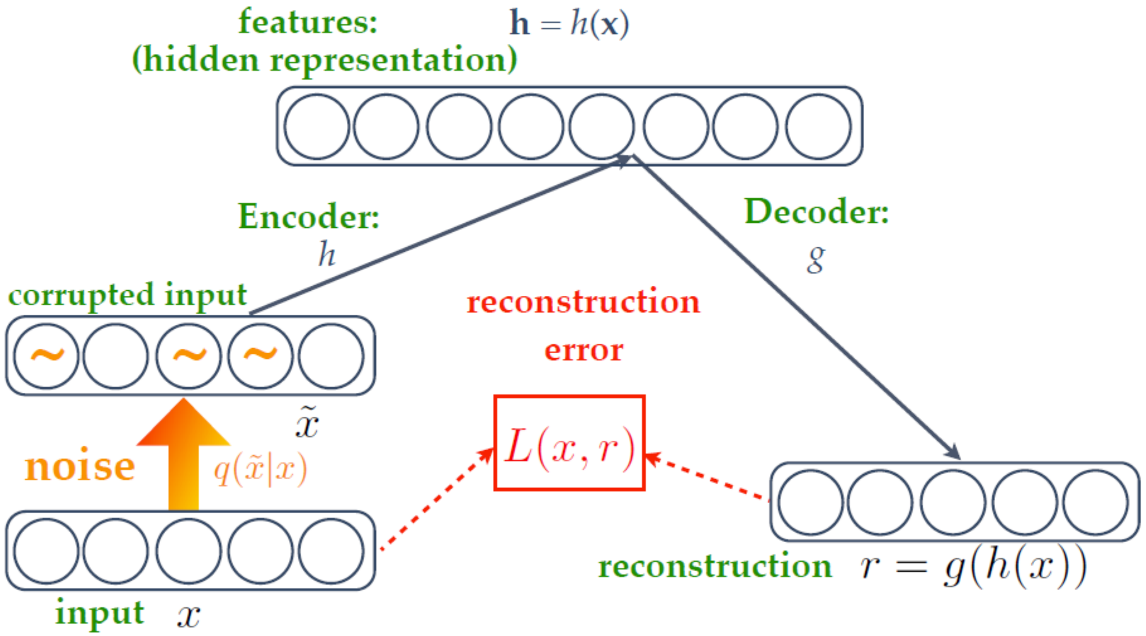
- But the key difference is that z is now a deterministic function of $\tilde{x}$ rather than x.

Denoising autoencoder

❖ **The objective function minimized by stochastic gradient descent becomes:**

$$\underset{\theta,\theta'}{\operatorname{argmin}}\, \mathbb{E}_{q^0(X,\tilde{X})}[L_{IH}(X, g_{\theta'}\left(f_\theta\left(\tilde{X}\right)\right))]$$

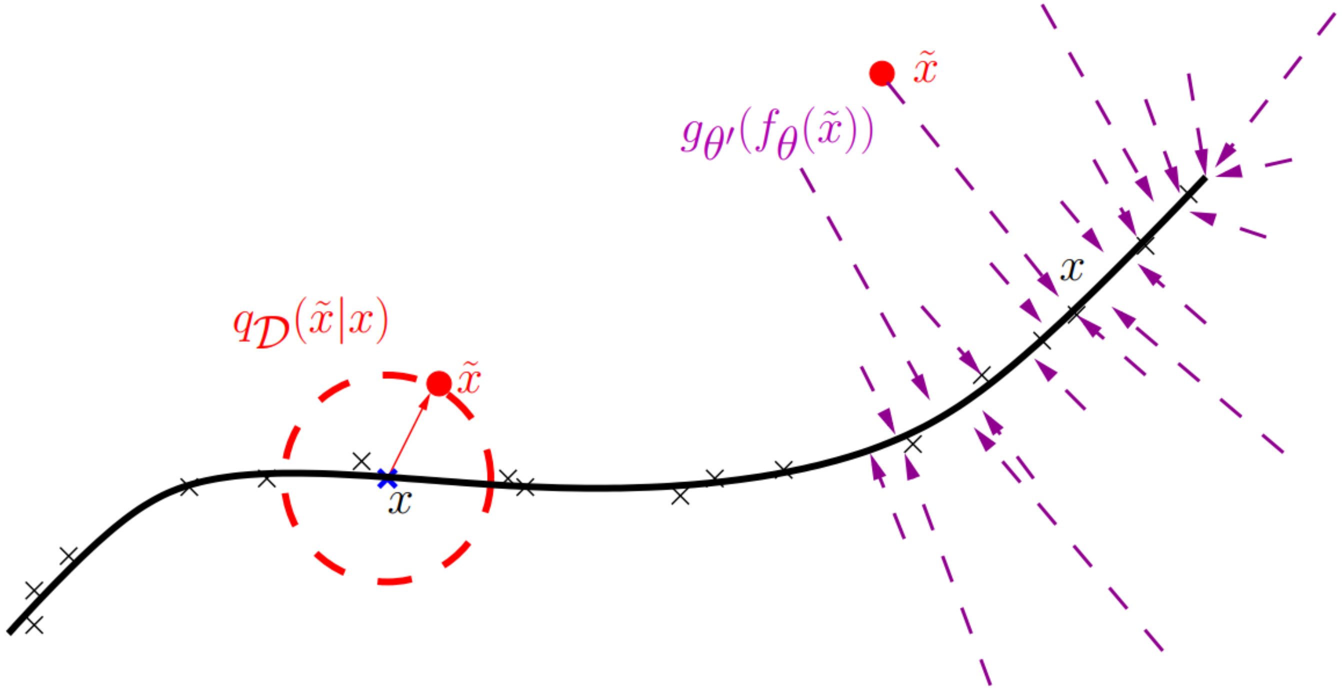
Denoising autoencoder



- Clean input $x \in [0, 1]^d$ is partially destroyed, yielding corrupted input: $\tilde{x} \sim q_D(\tilde{x}|x)$.
- $\tilde{x}$ is mapped to hidden representation $y = f_\theta(\tilde{x})$.
- From y we reconstruct a $z = g_{\theta'}(y)$.
- Train parameters to minimize the cross-entropy “reconstruction error” $L_{IH}(x, z) = H(\beta_x || \beta_z)$, where β_x denotes multivariate Bernoulli distribution with parameter x.

| http://videolectures.net/deeplearning2015_vincent_autoencoders/?q=vomcemt%20autoencoder

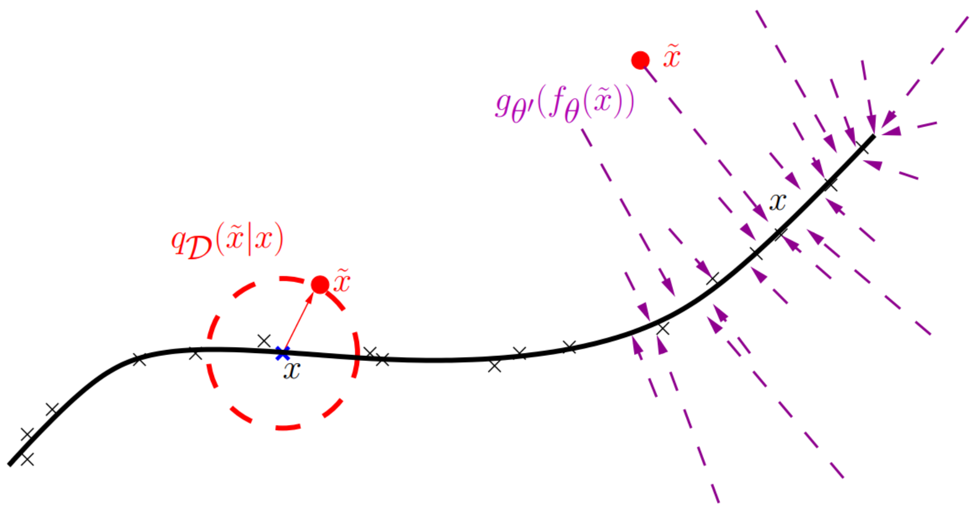
Denoising autoencoder – manifold perspective



- **Manifold learning perspective. Suppose training data (X) concentrate near a low-dimensional manifold. Corrupted examples (·) obtained by applying corruption process $q_d(\tilde{X}|X)$ will lie farther from the manifold. The model learns with $p(\tilde{X}|X)$ to "project them back" onto the manifold.**

Perspectives on denoising autoencoders

Manifold learning perspective



❖ **Denoising autoencoder can be seen as a way to learn a manifold:**

- Suppose training data (X) concentrate near a low-dimensional manifold.
- Corrupted examples $(\cdot)$ are obtained by applying corruption process $\boldsymbol{q_d}(\widetilde{\boldsymbol{X}}|\boldsymbol{X})$ and will lie farther from the manifold.
- The model learns with $\boldsymbol{p}(\widetilde{\boldsymbol{X}}|\boldsymbol{X})$ to "project them back" onto the manifold.
- Intermediate representation Y can be interpreted as a coordinate system for points on the manifold.

Denoising autoencoder - Result

Dataset	SVM_{rbf}	SVM_{poly}	DBN-1	SAA-3	DBN-3	SdA-3 (ν)
basic	**3.03±0.15**	3.69±0.17	3.94±0.17	3.46±0.16	3.11±0.15	**2.80±0.14** (10%)
rot	11.11±0.28	15.42±0.32	14.69±0.31	**10.30±0.27**	**10.30±0.27**	**10.29±0.27** (10%)
bg-rand	14.58±0.31	16.62±0.33	9.80±0.26	11.28±0.28	**6.73±0.22**	10.38±0.27 (40%)
bg-img	22.61±0.37	24.01±0.37	**16.15±0.32**	23.00±0.37	**16.31±0.32**	**16.68±0.33** (25%)
rot-bg-img	55.18±0.44	56.41±0.43	52.21±0.44	51.93±0.44	47.39±0.44	**44.49±0.44** (25%)
rect	**2.15±0.13**	**2.15±0.13**	4.71±0.19	2.41±0.13	2.60±0.14	**1.99±0.12** (10%)
rect-img	24.04±0.37	24.05±0.37	23.69±0.37	24.05±0.37	22.50±0.37	**21.59±0.36** (25%)
convex	19.13±0.34	19.82±0.35	19.92±0.35	**18.41±0.34**	**18.63±0.34**	**19.06±0.34** (10%)

- Note that SAA-3(basic autoencoder) is equibalent to a SdA-3(denoising autoencoder) with $v = 0\%$ destruction
- As can be seen in the table, the corruption+denoising training works remarkably well as an initialization step
- In most cases yields significantly better classification performance than basic autoencoder stacking with no noise.

Denoising autoencoder – Concluding remarks

- Unsupervised initialization of layers with an explicit denoising criterion helps to capture interesting structure in the input distribution
- This is turn leads to intermediate representations much better suited for subsequent learning tasks such as supervised classification
- Robustness to corruption in the representations they learn, possibly because of their stochastic nature which introduces noise in the representation during training

Autoencoder 구현실습

Development Environment

- ❖ conda, python
- ❖ tensorflow, sklearn, argparse, numpy, matplotlib, pillow
- ❖ jupyter notebook

Basic and Deep Autoencoder 실습(Tensorflow)

Content

❖ Basic autoencoder

- 오토인코더의 기본 구조
- 필요한 모듈 호출 및 데이터셋 로드
- 오토 인코더 모델 하이퍼파라미터 설정
- 오토인코더 모델 구성 (인코더/디코더)
- 모델 학습시키기 (손실 함수와 최적화 함수 설정)
- 학습 실행 결과
- 모델 테스트 해보기(matplotlib을 이용한 이미지 출력)

❖ Deep autoencoder

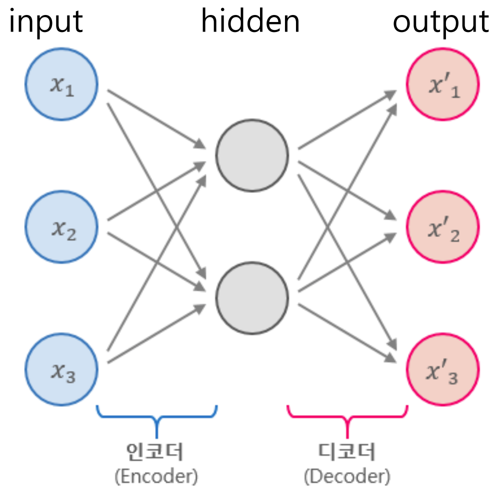
- 필요 라이브러리 설치 및 코드 다운로드
- 오토인코더 구조
- 모델 구조와 하이퍼파라미터 설정
- layer함수
- 인코더/디코더
- 손실 함수 정의 및 신경망 학습 알고리즘
- 모델 학습시키기
- 프로그램 실행 및 테스트 – 결과 및 손실값 변화

❖ Denoising autoencoder

- corrupt_input 함수
- 손상된 이미지 생성
- loss function
- 프로그램 실행하기
- 프로그램 실행하여 테스트하기
- 디노이징 오토인코더 결과 에시

Basic autoencoder

❖ 오토인코더의 기본 구조



- 입력값과 출력값을 같게 하는 신경망이며, 가운데 **은닉 계층의 노드 수가 입력값보다 적다**.
  (데이터 압출, 노이즈 제거에 효과적)
- 입력을 히든 레이어로 인코딩하고 이를 디코딩하여 만들어진 출력 값을 **원래의 입력값과 비슷해지도록 만드는 가중치**를 찾아내는 것이 핵심이다.
  (입력값이 압축되므로 입력에서 출력으로의 손실 없는 완벽한 복사는 일어날 수 없다.)
- **변이형** 오토인코더, **잡음 제거** 오토인코더 등 다양한 방식이 있다.

picture | https://excelsior-cjh.tistory.com/187

Basic autoencoder

❖ 필요한 모듈 호출 및 데이터셋 로드 (1)

- numpy – 행렬 조작과 연산에 필수라 할 수 있는 수치 해석용 파이썬 라이브러리
- matplotlib – 시각화를 위해 그래프를 쉽게 그릴 수 있도록 해주는 파이썬 라이브러리

- 밑에 두 줄처럼 텐서플로에 내장된 MNIST 모듈을 통해 MNIST 데이터셋을 다운받고 사용할 수 있다.

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("./mnist/data/", one_hot=True)
```

- MNIST 데이터셋 – 손으로 쓴 숫자들의 이미지를 모아 놓은 데이터셋으로, 0부터 9까지의 숫자를 28X28픽셀 크기의 이미지로 구성해 놓음. 전 처리까지 잘 되어있어 머신러닝계의 Hello, World!라고도 볼 수 있다.
  (총 70,000장 – train data: 60,000장, test data: 10,000장)

- one hot encoding – 데이터가 가질 수 있는 값들을 일렬로 나열한 배열을 만들고, 표현하려는 값을 뜻하는 인덱스의 원소만 1로 표기하고 나머지 원소는 모두 0으로 채우는 label 표기법

Basic autoencoder

❖ 필요한 모듈 호출 및 데이터셋 로드 (2)

- 프로젝트 폴더에 mnist 폴더가 생긴 것을 확인할 수 있다.

› 내 PC › 로컬 디스크 (C:) › TF_Projects › mnist › data

이름	수정한 날짜	유형	크기
t10k-images-idx3-ubyte	2019-07-17 오후...	압축(GZ) 파일	1,611KB
t10k-labels-idx1-ubyte	2019-07-17 오후...	압축(GZ) 파일	5KB
train-images-idx3-ubyte	2019-07-17 오후...	압축(GZ) 파일	9,681KB
train-labels-idx1-ubyte	2019-07-17 오후...	압축(GZ) 파일	29KB

- mnist 숫자 이미지

Basic autoencoder

❖ 오토인코더 모델 하이퍼 파라미터 설정

```
learning_rate = 0.01
training_epoch = 20
batch_size = 100
# 신경망 레이어 구성 옵션
n_hidden = 256  # 히든 레이어의 뉴런 갯수
n_input = 28*28  # 입력값 크기 - 이미지 픽셀 수
```

- learning_rate – 최적화 함수에서 사용할 학습률
- training_epoch – 전체 데이터를 학습할 총 횟수
- batch_size – 미니배치로 한 번에 학습할 데이터(이미지)의 개수

Basic autoencoder

❖ 오토인코더 모델 구성

```
#########
# 신경망 모델 구성
#####
X = tf.placeholder(tf.float32, [None, n_input])
```

- 입력 X의 플레이스홀더를 설정해준다.
- 오토인코더 모델은 **비지도 학습**이므로 y(ground truth)값이 없다.
- X코드에서 텐서의 첫 번째 차원에 None이 있는 자리는 **한번에 학습시킬 mnist 이미지의 개수**를 지정하는 값이 들어간다. 즉, 배치 크기를 지정하는 자리.
- None은 크기가 정해지지 않았음을 의미한다. 한번에 학습할 개수를 계속 바꿔가면서 실험해보려는 경우에 None으로 넣어주면 텐서플로가 알아서 계산한다.

Basic autoencoder

❖ 오토인코더 모델 구성 – 인코더 만들기

```
W_encode = tf.Variable(tf.random_normal([n_input, n_hidden]))
b_encode = tf.Variable(tf.random_normal([n_hidden]))
# sigmoid 함수를 이용해 신경망 레이어를 구성
# sigmoid(X * W + b)
# 인코더 레이어 구성
encoder = tf.nn.sigmoid(
                tf.add(tf.matmul(X, W_encode), b_encode))
```

인코더 모델: $\sigma(\mathrm{X} * W_{encode} + b_{encode})$

- 인코더 레이어의 가중치(weight)와 편향(bias) 변수를 원하는 뉴런의 개수만큼 설정하고 **정규분포를 띄는 무작위 수로 초기화** 한다.
- n_hidde개(256개)의 뉴런을 가진 은닉층을 만든다. output의 크기(은닉 층의 뉴런 개수)를 입력값(뉴런 784개)보다 적은 크기로 만들어 정보를 압축하여 특성을 뽑아낸다.
- 활성화 함수로 sigmoid 사용

Basic autoencoder

❖ 오토인코더 모델 구성 – 디코더 만들기

```
W_decode = tf.Variable(tf.random_normal([n_hidden, n_input]))
b_decode = tf.Variable(tf.random_normal([n_input]))
# 디코더 레이어 구성
# 이 디코더가 최종 모델
decoder = tf.nn.sigmoid(
                tf.add(tf.matmul(encoder, W_decode), b_decode))
```

디코더(최종 모델): $\sigma(\text{encoder} * W_{decode} + b_{decode})$

- 디코더도 인코더와 같은 구성이지만 **입력값을 은닉층의 크기**로, **출력값을 입력층의 크기**로 만들어 입력과 똑같은 output을 만들어 내도록 한다.
- 활성화 함수로 sigmoid를 사용하여 최종 모델을 구성한다(decoder가 최종 출력 값이 됨).

Basic autoencoder

❖ 모델 학습시키기 – 손실 함수와 최적화 함수 설정

```
cost = tf.reduce_mean(tf.pow(X - decoder, 2))
optimizer = tf.train.RMSPropOptimizer(learning_rate).minimize(cost)
```

- 입력 값인 X를 평가를 위한 실측값(ground truth)으로 사용하고, 디코더가 내보낸 결과 값 decode와의 차이의 제곱을 **손실값**으로 설정한다.
- **비용(cost)**은 tf.reduce_mean 함수를 사용하여 모든 데이터에 대한 손실 값의 평균을 내어 구한다. (MSE, Mean Square Error)
- 텐서플로가 기본적으로 제공하는 tf.train.RMSPropOptimizer 최적화 함수를 사용하여 가중치와 편향값을 변경해가면서 **손실값을 최소화하는 최적화된 가중치와 편향값**을 찾아주도록 한다.
- https://www.tensorflow.org/api_docs/python/tf/train

Basic autoencoder

❖ **모델 학습시키기**

```
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

total_batch = int(mnist.train.num_examples / batch_size)

for epoch in range(training_epoch):
    total_cost = 0
    for i in range(total_batch):
        batch_xs, batch_ys = mnist.train.next_batch(batch_size)
        _, cost_val = sess.run([optimizer, cost],
                               feed_dict={X: batch_xs})
        total_cost += cost_val
    print('Epoch: ', '%04d' % (epoch + 1),
          'Avg. cost = {:.4f}'.format(total_cost / total_batch))

print('최적화 완료!')
```

- 정의한 변수들을 **초기화**해주고, 학습 데이터의 총 개수인 mnist.train.num_examples를 앞서 설정했던 batch_size(미니배치 크기 = 100)로 나누어 **미니 배치가 총 몇 개**인지를 구한다. (mnist.train을 사용하면 학습 데이터를, mnist.test를 사용하면 테스트 데이터를 사용할 수 있다.)
- 학습 데이터 전체를 학습하는 일을 총 training_epoch만큼 반복한다.
- mnist.train.next_batch(batch_size)함수를 이용해 학습할 데이터를 배치 크기만큼 가져와서 feed_dict매개변수에 입력값이자 평가를 위한 실측값 X에 사용할 데이터를 넣어준다.
- sess.run을 이용하여 최적화시키고 손실값을 저장한 다음, 한 세대의 학습이 끝나면 학습한 세대의 평균 손실값을 출력한다.

Basic autoencoder

❖ 학습 실행 결과

- epoch이 늘어남에 따라 손실값이 성공적으로 점점 잘 줄어드는 것을 확인할 수 있다.

```
Epoch: 0001 Avg. cost = 0.2015
Epoch: 0002 Avg. cost = 0.0653
Epoch: 0003 Avg. cost = 0.0533
Epoch: 0004 Avg. cost = 0.0480
Epoch: 0005 Avg. cost = 0.0433
Epoch: 0006 Avg. cost = 0.0414
Epoch: 0007 Avg. cost = 0.0399
Epoch: 0008 Avg. cost = 0.0387
Epoch: 0009 Avg. cost = 0.0378
Epoch: 0010 Avg. cost = 0.0372
Epoch: 0011 Avg. cost = 0.0368
Epoch: 0012 Avg. cost = 0.0363
Epoch: 0013 Avg. cost = 0.0349
Epoch: 0014 Avg. cost = 0.0343
Epoch: 0015 Avg. cost = 0.0335
Epoch: 0016 Avg. cost = 0.0333
Epoch: 0017 Avg. cost = 0.0331
Epoch: 0018 Avg. cost = 0.0330
Epoch: 0019 Avg. cost = 0.0329
Epoch: 0020 Avg. cost = 0.0327
최적화 완료!
```

Basic autoencoder

❖ 모델 테스트 해보기

```
sample_size = 10

samples = sess.run(decoder,
                   feed_dict={X: mnist.test.images[:sample_size]})
```

- 먼저 앞에서부터 총 10개의 테스트 데이터를 가져와 feed_dict로 입력값 X에 넣어주고 디코더를 이용해 출력값으로 만든다.

Basic autoencoder

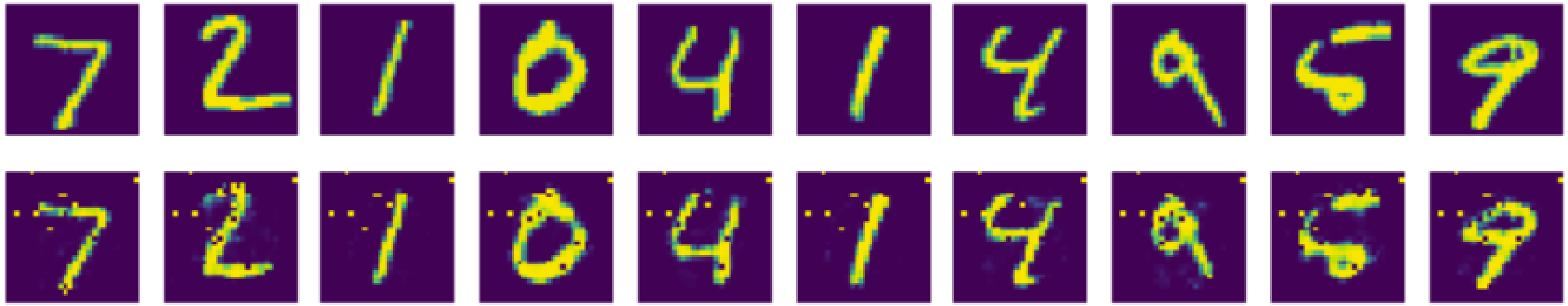
❖ 모델 테스트 해보기 – matplotlib을 이용한 이미지 출력

```
fig, ax = plt.subplots(2, sample_size, figsize=(sample_size, 2))

for i in range(sample_size):
    ax[0][i].set_axis_off()
    ax[1][i].set_axis_off()
    ax[0][i].imshow(np.reshape(mnist.test.images[i], (28, 28)))
    ax[1][i].imshow(np.reshape(samples[i], (28, 28)))

plt.show()
```

- numpy모듈을 이용해 mnist 데이터를 28x28크기의 이미지 데이터로 재구성한 뒤, matplotlib의 imshow 함수를 이용해 그래프에 이미지로 출력한다.
- 위쪽에는 입력 값의 이미지를, 아래쪽에는 오토인코더 신경망으로 생성한 이미지를 출력한다.

Basic autoencoder

❖ 모델 테스트 해보기 – 출력 결과



- 입력을 압축하여 약간의 노이즈가 껴 있지만 원본과 거의 유사하게 이미지를 생성해 낸 것을 확인할 수 있다.

Deep autoencoder

❖ 필요 라이브러리 설치 및 코드 다운로드

- 다음 명령어를 입력하여 실습에 필요한 라이브러리들을 설치한다.

관리자: 명령 프롬프트

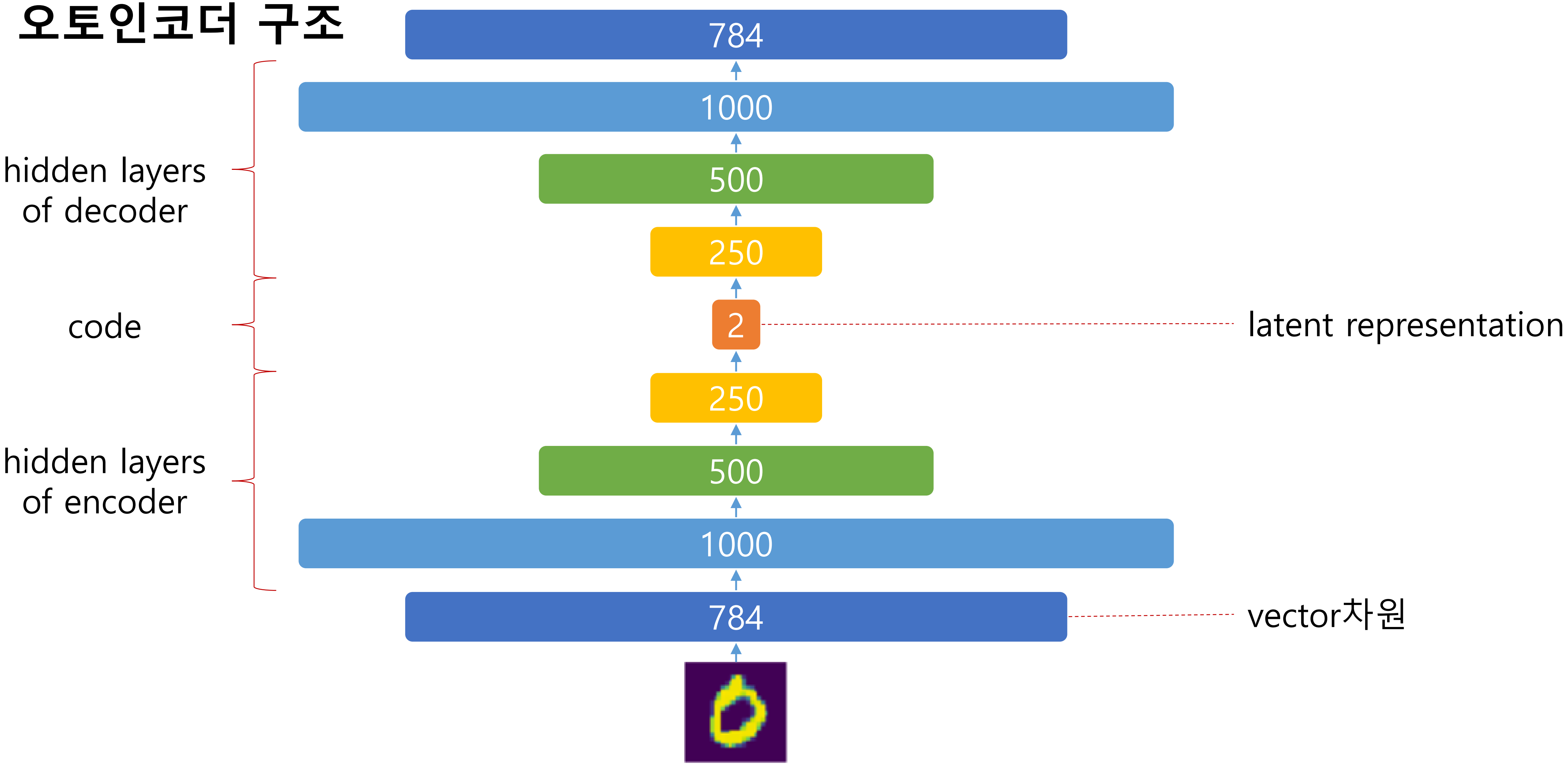
```
(tensorflow_dae) C:\TF_Projects>pip install sklearn argparse
```

- 구글 드라이브에서 C에 다운받고 풀어주기
  http://drive.google.com/open?id=1bRI8_Y-_LjAG6EjFIN03GFmaeVBb2mDR
- 그 폴더에 쥬피터 노트북 들어가기
- 오른쪽과 같이 to do로 비워져 있는 부분 코드 채워넣기
- 저장은 필수!

```
def encoder(x, n_code, phase_train):
    with tf.variable_scope("encoder"):
#################### to do ##############################




#####################################################
```

Deep autoencoder

❖ 오토인코더 구조



Deep autoencoder

❖ 모델 구조와 하이퍼파라미터 설정 – autoencoder_mnist.py

```
# Architecture
n_encoder_hidden_1 = 1000
n_encoder_hidden_2 = 500
n_encoder_hidden_3 = 250
n_decoder_hidden_1 = 250
n_decoder_hidden_2 = 500
n_decoder_hidden_3 = 1000

# Parameters
learning_rate = 0.01
training_epochs = 1000
batch_size = 100
```

Deep autoencoder

❖ layer 함수 – autoencoder_mnist.py

```
def layer(input, weight_shape, bias_shape, phase_train):
    weight_init = tf.random_normal_initializer(stddev=(1.0/weight_shape[0])**0.5)
    bias_init = tf.constant_initializer(value=0)
    W = tf.get_variable("W", weight_shape,
                        initializer=weight_init)
    b = tf.get_variable("b", bias_shape,
                        initializer=bias_init)
    logits = tf.matmul(input, W) + b
    return tf.nn.sigmoid(layer_batch_norm(logits, weight_shape[1], phase_train))
```

- 인자로 받은 가중치와 편향을 초기화한다.
- 모델은 WX + b 를 사용하고 활성화 함수로 sigmoid를 사용하여 나오는 출력값을 반환한다.

Deep autoencoder

❖ 인코더 구현하기 – autoencoder_mnist.py

```
def encoder(x, n_code, phase_train):
    with tf.variable_scope("encoder"):
################# to do #########################
        with tf.variable_scope("hidden_1"):
            hidden_1 = layer(x, [784, n_encoder_hidden_1], [n_encoder_hidden_1], phase_train)
        with tf.variable_scope("hidden_2"):
            hidden_2 = layer(hidden_1, [n_encoder_hidden_1, n_encoder_hidden_2], [n_encoder_hidden_2], phase_train)
        with tf.variable_scope("hidden_3"):
            hidden_3 = layer(hidden_2, [n_encoder_hidden_2, n_encoder_hidden_3], [n_encoder_hidden_3], phase_train)
        with tf.variable_scope("code"):
            code = layer(hidden_3, [n_encoder_hidden_3, n_code], [n_code], phase_train)

    return code
##################################################
```

- layer(input, weight, bias, phase_train)
- 784 ⟶ 1000 ⟶ 500 ⟶ 250 ⟶ 2

Deep autoencoder

❖ 디코더 구현하기 – autoencoder_mnist.py

```
def decoder(code, n_code, phase_train):
    with tf.variable_scope("decoder"):
################ to do #########################
        with tf.variable_scope("hidden_1"):
            hidden_1 = layer(code, [n_code, n_decoder_hidden_1], [n_decoder_hidden_1], phase_train)
        with tf.variable_scope("hidden_2"):
            hidden_2 = layer(hidden_1, [n_decoder_hidden_1, n_decoder_hidden_2], [n_decoder_hidden_2], phase_train)
        with tf.variable_scope("hidden_3"):
            hidden_3 = layer(hidden_2, [n_decoder_hidden_2, n_decoder_hidden_3], [n_decoder_hidden_3], phase_train)
        with tf.variable_scope("code"):
            output = layer(hidden_3, [n_decoder_hidden_3, 784], [784], phase_train)

    return output
################################################
```

- layer(input, weight, bias, phase_train)
- 2 ⟶ 250 ⟶ 500 ⟶ 1000 ⟶ 784

Deep autoencoder

❖ 손실 함수 정의 및 신경망 학습 알고리즘 – autoencoder_mnist.py

```python
def loss(output, x):
    with tf.variable_scope("training"):
################# to do ##########################
        l2 = tf.sqrt(tf.reduce_sum(tf.square(tf.subtract(output, x)), 1))
        train_loss = tf.reduce_mean(l2)
##################################################
        train_summary_op = tf.summary.scalar("train_cost", train_loss)
        return train_loss, train_summary_op

def training(cost, global_step):
    optimizer = tf.train.AdamOptimizer(learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-08,
                                       use_locking=False, name='Adam')
    train_op = optimizer.minimize(cost, global_step=global_step)
    return train_op
```

- 손실값은 입력값과 재구성한 출력값 사이의 거리를 L2 norm으로 계산한다.
- L2 norm 수식: $||I - O|| = \sqrt{\sum_i (I_i - O_i)^2}$
- 최종 손실값(train_loss)을 생성하기 위해 **미니배치 전체에 걸쳐 이 함수의 평균을 구한다.**
- Adam 최적화 알고리즘을 사용하고, 최종 손실값을 최소화하는 방향으로 신경망을 학습시킨다.

Deep autoencoder

❖ 모델 학습시키기 – autoencoder_mnist.py

```python
# Training cycle
for epoch in range(training_epochs):

    avg_cost = 0.
    total_batch = int(mnist.train.num_examples/batch_size)
    # Loop over all batches
    for i in range(total_batch):
        minibatch_x, minibatch_y = mnist.train.next_batch(batch_size)
        # Fit training using batch data
        _, new_cost, train_summary = sess.run([train_op, cost, train_summary_op],feed_dict={x: minibatch_x, phase_train: True})
        train_writer.add_summary(train_summary, sess.run(global_step))
        # Compute average loss
        avg_cost += new_cost/total_batch

    # Display logs per epoch step
    if epoch % display_step == 0:
        print("Epoch:", '%04d' % (epoch+1), "cost =", "{:.9f}".format(avg_cost))

        saver.save(sess, "mnist_autoencoder_hidden=" + n_code + "_logs/model-checkpoint-" + '%04d' % (epoch+1), global_step=global_step)
```

- 학습의 최종 목적은 cost를 최소화 시키는 것.
- 1 epoch 당 평균 최종 손실값을 출력한다.
- phase_train: True면 학습, False면 테스트
- 각 epoch 당 코드층(인코더 마지막 층) 모델의 체크포인트 저장도구(saver)를 사용한다.

Deep autoencoder

❖ 프로그램 실행하기 – autoencoder_mnist.py

- 명령줄 파라미터를 받아들여 실행하기 때문에 명령 프롬프트에서 코드를 실행해야한다.
- Window + R키를 눌러 cmd를 관리자 권한으로 또 하나를 실행한다.
- 아래와 같이 입력하여 가상환경을 실행시키고 프로그램 파일이 있는 폴더로 다시 이동한다.
- 다음과 같은 입력으로 실행하여 오토인코더 모델을 학습시킨다.

```
E:₩DenoisingAE>activate tensorflow_dae

(tensorflow_dae) E:₩DenoisingAE>python autoencoder_mnist.py 2
```

- .py 뒤에 붙은 2는 인코더의 마지막 출력으로 나오는 뉴런의 개수로 정해준 것이다. (code)

Deep autoencoder

❖ 실행 결과

```
Epoch: 0001 cost = 11.684345055
Validation Loss: 9.885018
Epoch: 0002 cost = 9.505442881
Validation Loss: 8.773004
Epoch: 0003 cost = 8.445695796
Validation Loss: 8.347751
Epoch: 0004 cost = 7.869748344
Validation Loss: 7.4505854
Epoch: 0005 cost = 7.405649343
Validation Loss: 7.3392873
Epoch: 0006 cost = 7.133315698
Validation Loss: 6.988289
Epoch: 0007 cost = 6.926423908
Validation Loss: 6.995893
```

- 다음과 같이 epoch 당 모델의 checkpoint가 저장된 폴더가 생성된다.

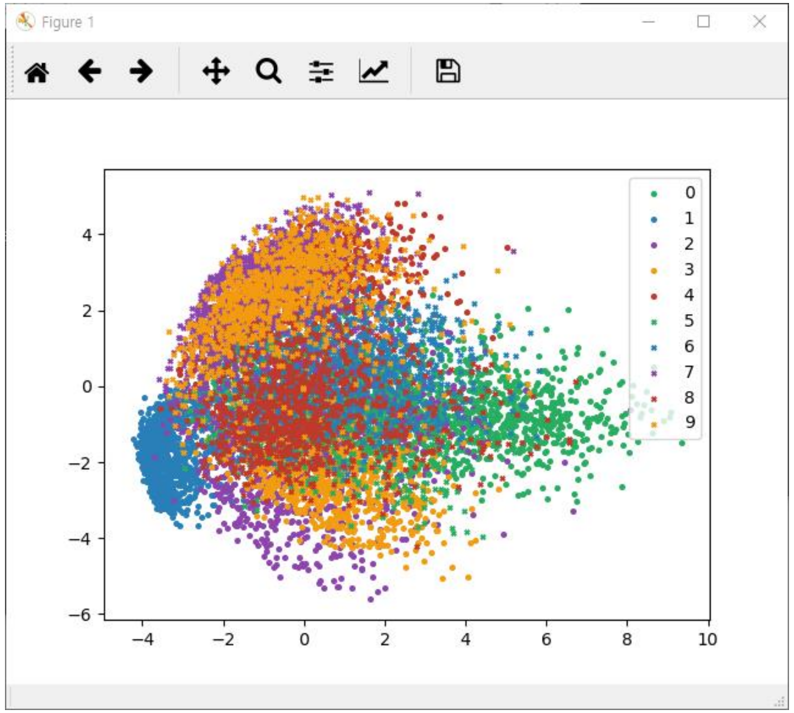
› 내 PC › 로컬 디스크 (E:) › DenoisingAE ›

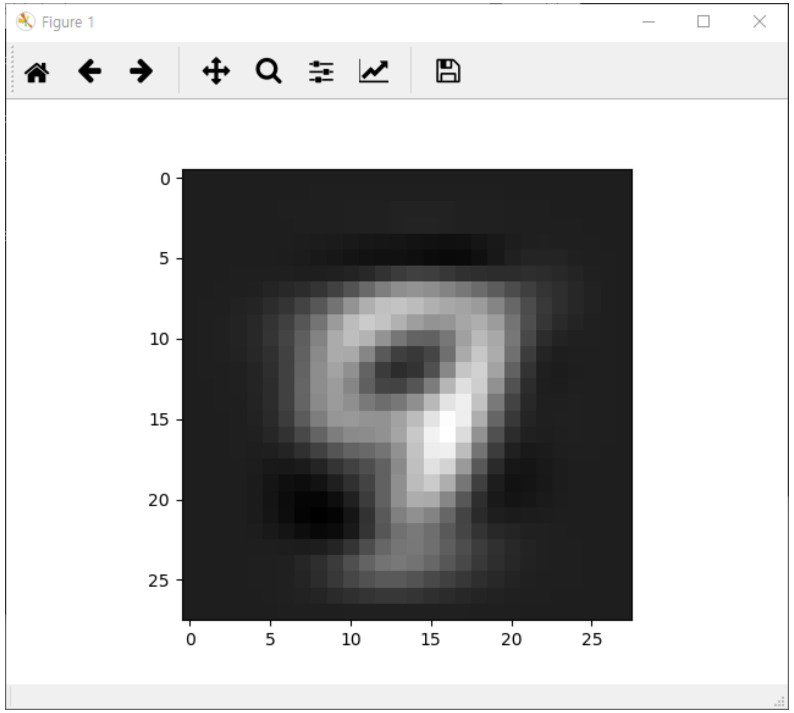
이름	수정한 날짜	유형	크기
.idea	2019-07-17 오후...	파일 폴더	
__pycache__	2019-07-17 오후...	파일 폴더	
data	2019-07-17 오후...	파일 폴더	
mnist_autoencoder_hidden=2_logs	2019-07-17 오후...	파일 폴더	
ae_vs_pca	2019-07-16 오후...	JetBrains PyCharm	3KB
autoencoder_mnist	2019-07-17 오후...	JetBrains PyCharm	8KB
denoising_autoencoder_mnist	2019-07-16 오후...	JetBrains PyCharm	8KB
dn_ae_test	2019-07-16 오후...	JetBrains PyCharm	3KB
input_data	2019-07-16 오후...	JetBrains PyCharm	8KB

Deep autoencoder

❖ 프로그램 실행하여 테스트하기

- python ae_vs_pca.py [모델 체크포인트 절대 경로]





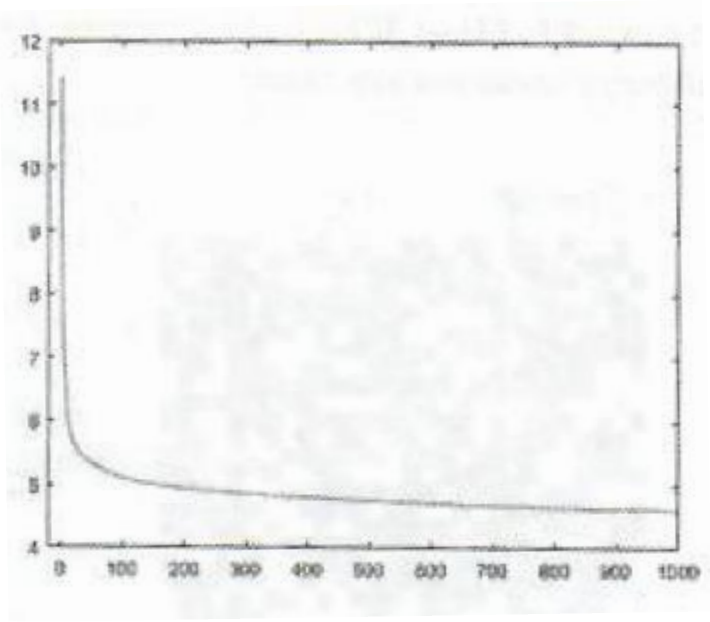
Deep autoencoder

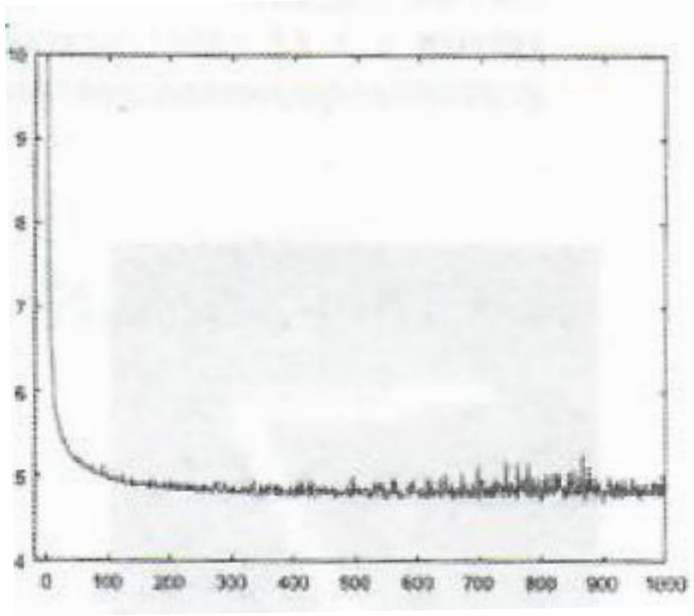
❖ 실행 결과 – 손실값 변화

- 손실값이 성공적으로 점점 줄어드는 것을 확인할 수 있다.

```
Epoch: 0001 cost = 11.684345055
Validation Loss: 9.885018
Epoch: 0002 cost = 9.505442881
Validation Loss: 8.773004
Epoch: 0003 cost = 8.445695796
Validation Loss: 8.347751
Epoch: 0004 cost = 7.869748344
Validation Loss: 7.4505854
Epoch: 0005 cost = 7.405649343
Validation Loss: 7.3392873
Epoch: 0006 cost = 7.133315698
Validation Loss: 6.988289
Epoch: 0007 cost = 6.926423908
Validation Loss: 6.995893
```

- 총 1000 epoch의 학습 기간에 걸친 손실 값 변화를 시각화하면 아래와 같다.
- 학습과 검증 비용 모두 점점 평평해질 때가지 감소한다.





Deep autoencoder

❖ 실행 결과

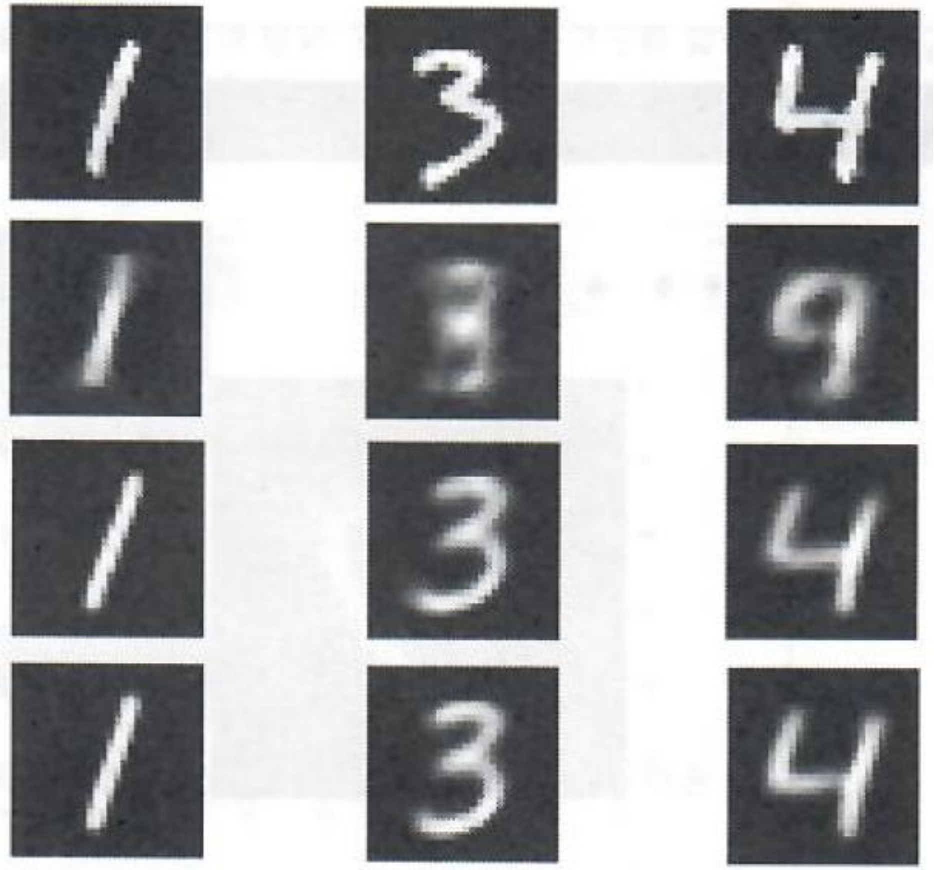
- 각 5, 100, 200 epoch마다 테스트 이미지를 재구성한 이미지들을 비교해본다.

원본 입력

5 epoch 후 재구성

100 epoch 후 재구성

200 epoch 후 재구성



Denoising autoencoder

❖ corrupt_input 함수 – denoising_autoencoder_mnist.py

- 원래 input 이미지에 노이즈를 곱하여 손상된 이미지를 생성한다.
- 0~2 사이 정수 (0, 1, 2)를 tf.random_uniform 함수를 사용하여 균일 분포로 입력값 X 크기만큼 초기화해 준다.

```
def corrupt_input(x):
    ##################### to do ##########################
    corrupting_matrix = tf.random_uniform(shape=tf.shape(x), minval=0, maxval=2, dtype=tf.int32)
    return x * tf.cast(corrupting_matrix, tf.float32)
    ######################################################
```

Denoising autoencoder

❖ 손상된 이미지 생성 – denoising_autoencoder_mnist.py

- Corrupt가 placeholder로 생성이 되는데, 1이면 corrupt_input이 c_x가 되고, 0이면 x가 c_x가 된다.
  즉, 1이면 입력을 손상시키고, 0이면 손상시키지 않는다.

- 인코더 신경망에 대한 입력이 X 대신 손상된 c_x가 된다.

```
##################### to do ##########################
corrupt = tf.placeholder(tf.float32)

c_x = (corrupt_input(x) * corrupt) + (x * (1 - corrupt))

code = encoder(c_x, int(n_code), phase_train)
####################################################
```

Denoising autoencoder

❖ loss function – denoising_autoencoder_mnist.py

- 기본 오토인코더와 손실 함수는 동일하다.
- 손실값을 구할 때 c_x와 outpu의 차이값이 아닌 원본 이미지의 x와 output의 차이로 정의하였기 때문에 손상 이미지가 아닌 원본 이미지와 유사하게 재구성하도록 학습시킨다.

```
def loss(output, x):
    with tf.variable_scope("training"):
        l2 = tf.sqrt(tf.reduce_sum(tf.square(tf.subtract(output, x)), 1))
        train_loss = tf.reduce_mean(l2)
        train_summary_op = tf.summary.scalar("train_cost", train_loss)
        return train_loss, train_summary_op
```

```
c_x = (corrupt_input(x) * corrupt) + (x * (1 - corrupt))

code = encoder(c_x, int(n_code), phase_train)
#######################################################

output = decoder(code, int(n_code), phase_train)

cost, train_summary_op = loss(output, x)
```

Denoising autoencoder

❖ 프로그램 실행하기 - denoising_autoencoder_mnist.py

- 다음과 같은 입력으로 실행하여 디노이징 오토인코더 모델을 학습시킨다.

관리자: 명령 프롬프트

```
(tensorflow_dae) C:₩practice>python denoising_autoencoder_mnist.py2
```

- .py 뒤에 붙은 '2'는 인코더의 마지막 출력으로 나오는 뉴런의 개수로 정해준 것이다. (encode)

Denoising autoencoder

❖ 프로그램 실행하여 테스트하기

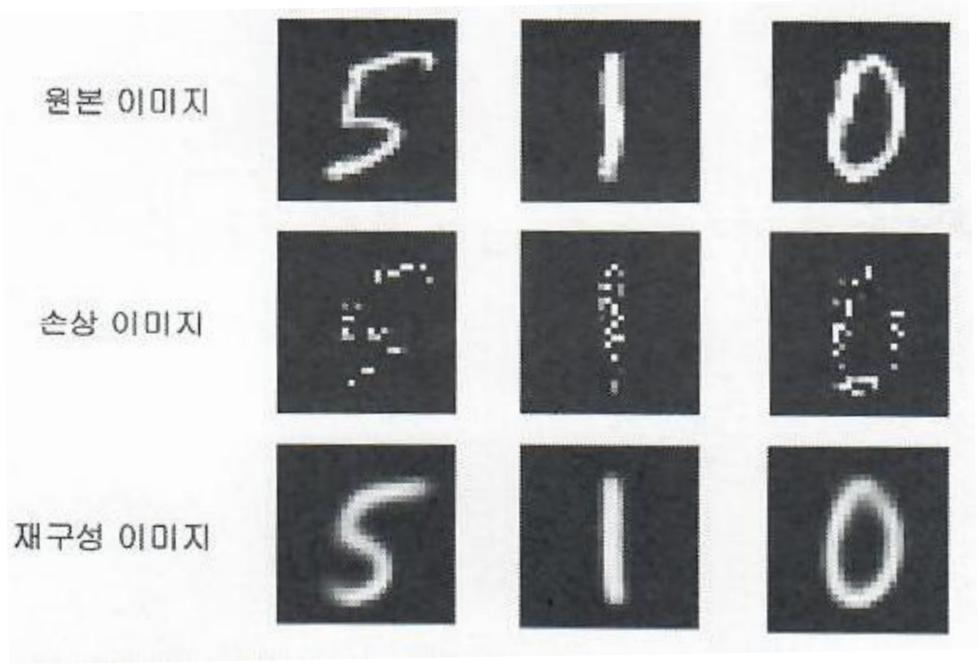
- python dn_ae_test.py [모델 체크포인트 절대 경로]

관리자: 명령 프롬프트

```
(tensorflow_dae) C:₩practice>python dn_ae_test.py C:₩practice₩autoencoder1-2₩model-checkpoint-0980-539000
```

Denoising autoencoder

❖ 디노이징 오토인코더 결과 예시



- 데이터셋에 손상 작업을 적용하고 손상되지 않은 원본 이미지들을 재구성한 이미지 비교
- 빠진 픽셀들을 잘 채워주는 것을 확인할 수 있다.