

# Deep Learning

조희철

2018년 3월 4일

## 차례

<b>1</b>	<b>Language Model</b>	<b>3</b>
<b>2</b>	<b>Chain Rule</b>	<b>8</b>
<b>3</b>	<b>Deep Neural Net</b>	<b>11</b>
<b>4</b>	<b>Back Propagation</b>	<b>12</b>
4.1	A Neural Network in 11 Lines . . . . .	12
4.2	Linear Classifier . . . . .	13
<b>5</b>	<b>Tensorflow &amp; Network</b>	<b>15</b>
5.1	Tensorflow . . . . .	15
5.2	Network 설계 . . . . .	18
<b>6</b>	<b>Convolution Neural Network</b>	<b>24</b>
6.1	im2col . . . . .	24
6.2	VGG의 이해 . . . . .	26
6.3	Image File Loading & Save . . . . .	29
<b>7</b>	<b>Recurrent Neural Network</b>	<b>31</b>
7.1	RNN Toy Model: i am trask blog . . . . .	31
7.2	DENNY BRITZ Language Model . . . . .	34
7.3	Minimal Character-Level Vanilla RNN Model by Andrej Karpathy . . . . .	36
7.4	LSTM(Long Short Term Memory) Networks . . . . .	38
7.5	RNN with TensorFlow . . . . .	41
7.6	Character-Level Vanilla RNN Model by Andrej Karpathy . . . . .	44
7.7	Image Captioning with RNNs(CS231n) . . . . .	45
7.8	Neural Machine Translation . . . . .	46
<b>8</b>	<b>Reinforcement Learning</b>	<b>47</b>
8.1	Q-Table, Q-Network . . . . .	47
8.2	DQN . . . . .	49
8.3	Policy Gradient . . . . .	50

<b>9 Generative Adversarial Networks</b>	<b>51</b>
9.1 Variational Auto Encoder . . . . .	51
9.2 Vanilla GAN . . . . .	51
9.3 DCGAN . . . . .	52
9.4 Texture Synthesis(2015) . . . . .	58
9.5 Neural Style(2017) . . . . .	59
9.6 Deep Photo Style Transfer . . . . .	63
<b>10 Miscellaneous</b>	<b>65</b>
10.1 Optimization Methods . . . . .	65
10.2 Batch Normalization . . . . .	66
10.3 Naive Bayes Spam Filter . . . . .	68
10.4 Support Vector Machine . . . . .	68
10.5 Redistricted Boltzmann Machine . . . . .	71
10.6 기타 . . . . .	73

# 1 Language Model

이 내용은 Geoffrey Hinton 교수님의 Machine Learning 강의 중 Language Model(A Neural Probabilistic Language Model, Bengio et al)을 정리하는 것임을 밝힌다. ML의 초보자로서 처음 Language Model을 접했을 때, 이해하지 못한 것들을 나름의 방식으로 이해하여 정리하고자 한다. 좀 더 자세히 말하면, Assignment2에 있는 코드를 분석하여 정리한 것이다. 다만, Back Propagation 계산에 필요한 Chain Rule은 어느정도 알고 있다고 가정하고 설명한다. (그림이나 내용은 추가적인 인용없이 사용하겠습니다. <https://www.coursera.org/learn/neural-networks/exam/915Ts/programming-assignment-2-learning-word-representations>)

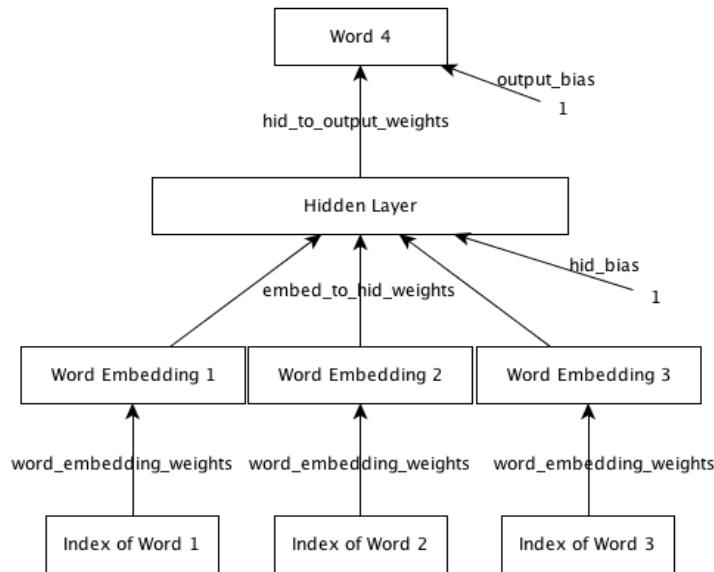


그림 1: 강의에 나오는 network 그림

입력 data는 data.mat 파일로 제공되고 있는데, 분석해 보면 다음과 같다. 예를 들어, 다음과 같은

no way to do business around here.

문장이 있다면 아래와 같이 4-gram sequence를 만든다. 마침표 다음 문장과 연결되게 data를 만들지는 않는다.

- no way to do
- way to do business
- to do business around
- do business around here
- business around here.

## ♠ 들어가며

1. 이 모델은 문장 속 4개의 연속된 단어를 학습한 후, 앞 3개의 단어가 주어졌을 때, 다음 네번째 단어를 예측하는 모델이다.
2. 앞으로 설명할 때, data의 개수, hidden layer의 dimension등은  $N, M$ 과 같이 일반화하여 나타내지 않고, 구체적인 숫자로 설명하고자 한다. 구체적인 숫자로 설명하는 것이 행렬의 사이즈를 따라가기 편하기 때문.
3. Assignment에 예로 나오는 숫자를 동일하게 기본으로 사용하겠습니다. 숫자를 달리하여 일반화 하는 것은 별 문제 없습니다.
  - 3: 학습하는 모델이 단어 세개를 받아들여서 학습하는 모델
  - 100: batch size
  - 50: 각 단어의 feature를 만들기 위해 feature의 크기를 50으로 설정함(Dimensionality of embedding space).
  - 250: 학습에 등장하는 모든 단어가 250개.
  - 200: hidden layer의 dimension(또는, Number of units in hidden layer).

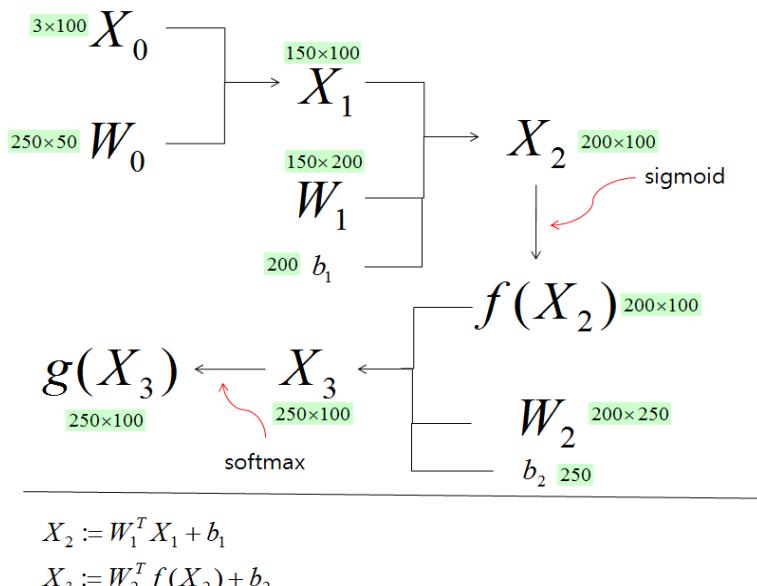


그림 2: 행렬로 표현한 network

## ♠ Forward Propagation

0. learning에 사용할 단어는 모두 250개. 각 단어를 1 ~ 250의 숫자로 표시.
1.  $X_0$ : 입력 데이터로 3개의 단어와 batch size를 100으로 하기 때문에,  $3 \times 100$  행렬.
2.  $W_0$ : 강의에서는 이 변수를 ‘word embedding weights’라고 하는데, 250개의 각 단어들의 feature를 만들기 위한 것. 각 단어의 feature를 50(Dimensionality of embedding space) 개로 설정하면,  $W_0$ 의 크기는  $250 \times 50$  이 된다. 학습이 이루어지면, 이 값이 각 단어들의 feature가 되고, 이 feature가 유사하다는 것은 대체할 수 있

다는 것으로 해석할 수도 있다. Assignment에서는 ‘display\_nearest\_words’라는 함수를 제공하고 있는데, 이 함수는 유사한 feature를 가지 단어를 제시해 준다. 예를 들면 learning 후, ‘could’를 ‘display\_nearest\_words’에 넣어보면, ‘can’, ‘should’, ‘would’, ‘might’, ‘may’, ‘will’ 순으로 유사성이 높은 단어로 제시해 준다.

3.  $X_0, W_0$ 로 부터  $X_1$  생성: 먼저  $W_0$ 를 행벡터 형식으로 표시하면,

$$W_0 = \begin{pmatrix} w_{01} \\ w_{02} \\ \vdots \\ w_{0250} \end{pmatrix}, w_{0i} : 1 \times 50.$$

예를 들어,  $X_0$ 으로부터  $X_1$ 을 다음과 같이 생성한다.

$$X_0 = \begin{pmatrix} & 3 & \\ \cdots & 21 & \cdots \\ & 7 & \end{pmatrix} \implies X_1 = \begin{pmatrix} w_{03}^T \\ \vdots \\ w_{021}^T \\ \cdots \\ w_{07}^T \end{pmatrix}$$

이렇게  $X_0, W_0$ 로 부터,  $X_1$ 을 생성하는 것은 간단할 수 있지만, 나중에 back propagation을 위한 미분 과정을 이해하기는 어려워진다.

4. 이번에는  $X_1$ 을 다른 방식으로 생성하기 위해, 먼저  $X_0$ 를 one hot encoding 형식으로 변환해 보자.

$X_0$ 의 첫행을 one hot encoding으로 변환한 행렬을  $K_1(250 \times 100)$ ,

$X_0$ 의 두번행을 one hot encoding으로 변환한 행렬을  $K_2$ ,

$X_0$ 의 세번행을 one hot encoding으로 변환한 행렬을  $K_3$ .

이렇게 구해진  $K_1, K_2, K_3$ 는 각각  $250 \times 100$  행렬이 된다. 이게  $W_0$ 와  $K_1, K_2, K_3$ 를 결합한 행렬과의 곱을 계산을 보자.

$$W_0^T \begin{pmatrix} K_1 & | & K_2 & | & K_3 \end{pmatrix} = \begin{pmatrix} W_0^T K_1 & | & W_0^T K_2 & | & W_0^T K_3 \end{pmatrix}$$

이렇게 구한  $W_0^T K_1, W_0^T K_2, W_0^T K_3$ 를 세로로 재배치하면  $X_1$ 과 동일한 행렬이 된다. 즉,

$$X_1 = \begin{pmatrix} W_0^T K_1 \\ \hline \hline W_0^T K_2 \\ \hline \hline W_0^T K_3 \end{pmatrix} \quad (1)$$

여기서는 (행렬 연산이 아닌) 재배치 과정을 통해  $X_1$ 을 구했는데, 행렬 연산만으로  $X_1$ 을 구할 수 있는지는 모르겠음.

5. 이제 나머지 forward propagation은 [그림 2]를 보면, 쉽게 알 수 있다.

$$\begin{aligned}
 X_2 &= W_1^T X_1 + b_1 \leftarrow 200 \times 100 \\
 f(X_2) &: \text{sigmoid function} \\
 X_3 &= W_2^T f(X_2) + b_2 \leftarrow 250 \times 100 \\
 g(X_3) &: \text{softmax function}
 \end{aligned}$$

### ♠ Backward Propagation

forward propagation을 통해 구해진 output을  $y(250 \times 100)$ , one hot encoding으로 변환된 target값을  $t(250 \times 100)$ 로 하자. cross-entropy-cost(regularization) 함수와 softmax 함수를 미분하여 구해진, error 값  $y - t$ 로부터 구해지는 gradient는 다음과 같다( $\circ$ : Hadamard product).  $\lambda$ 를 weight decay coefficient라 하자.

$$\begin{aligned}
 dX_3 &= (y - t)/100 \\
 dW_2 &= f(X_2) (y - t)^T + \lambda W_2 \leftarrow 200 \times 250 \\
 db_2 &= \text{sum over rows}(y - t) \leftarrow 250 \\
 dX_2 &= \left( W_2(y - t) \right) \circ \left( f(X_2) \circ (1 - f(X_2)) \right) \leftarrow 200 \times 100 \\
 dW_1 &= X_1 dX_2^T + \lambda W_1 \leftarrow 150 \times 200 \\
 db_1 &= \text{sum over rows}(dX_2) \leftarrow 200 \\
 dX_1 &= W_1 dX_2 \leftarrow 150 \times 100
 \end{aligned}$$

이제 남은 것은  $dW_0$ 를 구하는 것만 남았다. 식(1)을 다시 보면 알 수 있듯이,  $W_0$ 로부터  $X_1$ 을 구하는 과정을 보면  $X_1$ 은 3개의 block으로 분할되어 있다.  $W_0$ 의 gradient는 각각의 block에 대한  $W_0$ 의 gradient를 구하여 합하면 된다. 따라서,  $X_1(150 \times 100)$ 이 3개의  $50 \times 100$  행렬로 나누어져 있으므로,  $dX_1(150 \times 100)$ 도 동일하게  $50 \times 100$  크기의 3개의 block( $dX_{11}, dX_{12}, dX_{13}$ )으로 분해하자.

$$X_1 = \begin{pmatrix} W_0^T K_1 \\ \cdots \\ W_0^T K_2 \\ \cdots \\ W_0^T K_3 \end{pmatrix}, \quad dX_1 = \begin{pmatrix} dX_{11} \\ \cdots \\ dX_{12} \\ \cdots \\ dX_{13} \end{pmatrix}, \quad dX_{1i} : 50 \times 100$$

이제  $dW_0$ 은 다음과 같이 구해진다.

$$dW_0 = \sum_{i=1}^3 K_i dX_{1i}^T$$

### ♠ From Bengio's Language Model to Word2Vec

- Bengio Language Model에서와 같이 단어를 feature vector로 만드는 것을 embedding이라고 하는데, embedding 방법에는 Word2Vec, Glove, Fasttext가 있다. embedding 방식의 논문은 다음과 같다.
  - ”A Neural Probabilistic Language Model”, Bengio, et al. 2003
  - ”Efficient Estimation of Word Representations in Vector Space”, Mikolov, et al. 2013
  - ”word2vec Parameter Learning Explained”, Xin Rong

- 학습이 느린 Bengio Model을 변형한 Model을 Mikolov가 제안함.
- CBOW(Continuous Bag of Words): 여러개의 단어들을 나영한 뒤, 다음 단어를 예측하는 방식. 학습 속도가 아주 빠르다.

the quick brown fox jumped over the lazy dog

예를 들어, 위와 같은 문장에서 (the, quick, brow)으로 부터 (fox)를 예측하는 모델.

- Skip-Gram: 주어진 단어를 중심으로 전, 후의 단어를 예측하는 모델. 느리지만 빈도가 낮은 단어들에 적합.
- Word2Vec은 CBOW와 Skip-Gram 방식의 embedding을 구현한 것으로, Google에 있던 Mikolov등에 의해 C++ 라이브러리로 개발했다. Python에서는 gensim package에 'Word2Vec' class로 구현되어 있다.

## 2 Chain Rule

### ♠ 행렬의 미분

$m \times n$  행렬  $A$ ,  $n \times k$  행렬  $X$  와 함수  $f : \mathbb{R}^{mk} \rightarrow \mathbb{R}$ 에 대하여,  $A, X$  의 각 원소에 대한 미분을 구해보자.

$$\begin{array}{c} A \\ \longmapsto AX \xrightarrow{f} f(AX) \in \mathbb{R} \\ X \end{array}$$

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}, X = \begin{pmatrix} x_{11} & \cdots & x_{1k} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nk} \end{pmatrix}, AX = \begin{pmatrix} y_{11} & \cdots & y_{1k} \\ \vdots & \ddots & \vdots \\ y_{m1} & \cdots & y_{mk} \end{pmatrix}.$$

$\frac{\partial f}{\partial Y}$  를 다음과 같이 정의하면,

$$\frac{\partial f}{\partial Y} := \begin{pmatrix} \frac{\partial f}{\partial y_{11}} & \cdots & \frac{\partial f}{\partial y_{1k}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial y_{m1}} & \cdots & \frac{\partial f}{\partial y_{mk}} \end{pmatrix}$$

$A, X$  의 각 원소별 미분은 다음과 같다.

$$\begin{pmatrix} \frac{\partial f}{\partial a_{11}} & \cdots & \frac{\partial f}{\partial a_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial a_{m1}} & \cdots & \frac{\partial f}{\partial a_{mn}} \end{pmatrix} = \frac{\partial f}{\partial Y} X^T, \quad \begin{pmatrix} \frac{\partial f}{\partial x_{11}} & \cdots & \frac{\partial f}{\partial x_{1k}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial x_{n1}} & \cdots & \frac{\partial f}{\partial x_{nk}} \end{pmatrix} = A^T \frac{\partial f}{\partial Y}$$

### ♠ Chain Rule

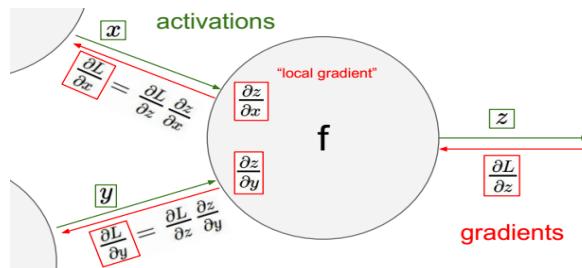


그림 3: Back Propagation & Chain Rule

$$\begin{array}{ccccccc} x & \xleftarrow[\times(-1)]{\times(-1)} & f & \xleftarrow[g]{\exp} & g & \xleftarrow[\times 1]{+1} & h & \xleftarrow[-y^2]{\text{inverse}} & y \\ x & \longrightarrow & -x & \longrightarrow & e^{-x} & \longrightarrow & 1 + e^{-x} & \longrightarrow & \frac{1}{1 + e^{-x}} \\ y^2 e^{-x} & \longleftarrow & -y^2 e^{-x} & \longleftarrow & -y^2 & \longleftarrow & -y^2 & \longleftarrow & 1 \end{array}$$

### ♠ Example

[그림4]와 같이 모델이 주어져 있다고 하자.

$$\begin{aligned}
z_1 &= w_1x_1 + w_2x_2 \\
z_2 &= w_3x_2 + w_4x_3 \\
h_1 &= \sigma(z_1) \\
h_2 &= \sigma(z_2) \\
y &= u_1h_1 + u_2h_2 \\
E &= \frac{1}{2}(t - y)^2
\end{aligned}$$

그림 4: Example: back propagation

Forward Propagation:

$$\begin{aligned}
\text{Data: } & \begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix}, \quad \text{weight: } \begin{pmatrix} w_1 & 0 \\ w_2 & w_3 \\ 0 & w_4 \end{pmatrix} \\
\longrightarrow & \begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix} \begin{pmatrix} w_1 & 0 \\ w_2 & w_3 \\ 0 & w_4 \end{pmatrix} =: \begin{pmatrix} z_1 & z_2 \end{pmatrix} \\
\longrightarrow & \begin{pmatrix} \sigma(z_1) & \sigma(z_2) \end{pmatrix} =: \begin{pmatrix} h_1 & h_2 \end{pmatrix}, \quad \sigma : \text{sigmoid} \\
\longrightarrow & \begin{pmatrix} h_1 & h_2 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} =: y, \quad u_i : \text{weight} \\
\longrightarrow & \frac{1}{2}(t - y)^2 := E(\text{Error})
\end{aligned}$$

Backward Propagation:

$$\begin{aligned}
\frac{\partial E}{\partial y} &= -(t - y) \leftarrow 1 \times 1, \\
\begin{pmatrix} \frac{\partial E}{\partial h_1} & \frac{\partial E}{\partial h_2} \end{pmatrix} &= \left( \frac{\partial E}{\partial y} \right) \begin{pmatrix} u_1 \\ u_2 \end{pmatrix}^T, \\
&= \begin{pmatrix} \frac{\partial E}{\partial y} u_1 & \frac{\partial E}{\partial y} u_2 \end{pmatrix} \leftarrow 1 \times 2, \\
\begin{pmatrix} \frac{\partial E}{\partial z_1} & \frac{\partial E}{\partial z_2} \end{pmatrix} &= \begin{pmatrix} h_1(1 - h_1) & h_2(1 - h_2) \end{pmatrix} \circ \begin{pmatrix} \frac{\partial E}{\partial y} u_1 & \frac{\partial E}{\partial y} u_2 \end{pmatrix} \\
&= \begin{pmatrix} h_1(1 - h_1) \frac{\partial E}{\partial y} u_1 & h_2(1 - h_2) \frac{\partial E}{\partial y} u_2 \end{pmatrix}, \\
\begin{pmatrix} \frac{\partial E}{\partial w_1} & 0 \\ \frac{\partial E}{\partial w_2} & \frac{\partial E}{\partial w_3} \\ 0 & \frac{\partial E}{\partial w_4} \end{pmatrix} &= \begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix}^T \begin{pmatrix} h_1(1 - h_1) \frac{\partial E}{\partial y} u_1 & h_2(1 - h_2) \frac{\partial E}{\partial y} u_2 \end{pmatrix} \\
&= \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \begin{pmatrix} h_1(1 - h_1) \frac{\partial E}{\partial y} u_1 & h_2(1 - h_2) \frac{\partial E}{\partial y} u_2 \end{pmatrix} \leftarrow 3 \times 2
\end{aligned}$$

$$\begin{array}{c}
\hline
W = \begin{pmatrix} w_1 & 0 \\ w_2 & w_3 \\ 0 & w_4 \end{pmatrix} \quad \searrow \quad \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} \quad \searrow \quad t \quad \searrow \\
X = \begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix} \rightarrow \begin{pmatrix} z_1 & z_2 \end{pmatrix} = XW \xrightarrow{\text{sigmoid}} (h_1 h_2) \rightarrow y \rightarrow E \\
\hline
X^T \left( \begin{array}{cc} \frac{\partial E}{\partial z_1} & \frac{\partial E}{\partial z_2} \end{array} \right) \nwarrow \quad (h_1 h_2)^T \frac{\partial E}{\partial y} \nwarrow \quad (M \times 1) \\
\left( \begin{array}{cc} \frac{\partial E}{\partial z_1} & \frac{\partial E}{\partial z_2} \end{array} \right) \quad \longleftarrow \quad \frac{\partial E}{\partial y} \begin{pmatrix} u_1 \\ u_2 \end{pmatrix}^T \quad \longleftarrow \quad \frac{\partial E}{\partial y}
\end{array}$$

### 3 Deep Neural Net

- Simple Net without Hidden layer

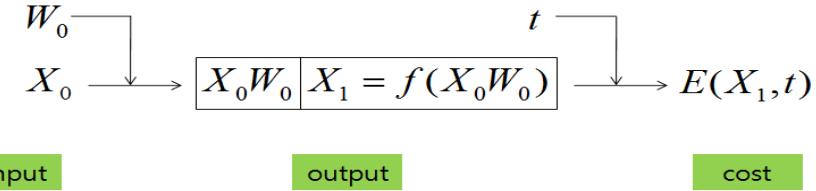
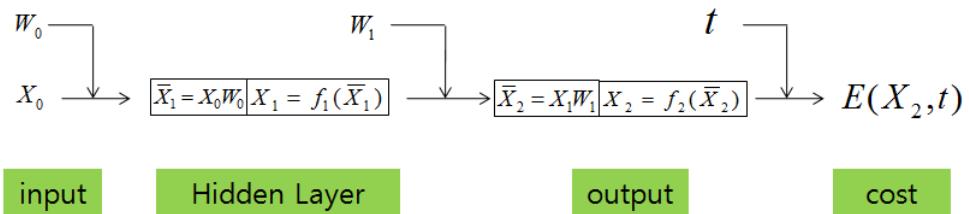


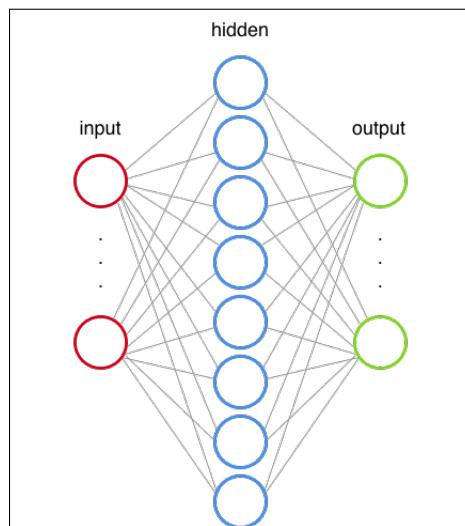
그림 5: Simple Net without Hidden Layer

- $f(x) = x$  인 경우: Linear Regression.
- $f(x) = \text{sigmoid}$ : Logistic Regression.
- $f(x) = \text{softmax}$ : Multinomial Classification.
- $E$ : cost function,  $L_2$ -norm, cross entropy error.

- Neural Net with 1-Hidden layer



(a)



(b)

그림 6: Neural Network with 1-hidden layer

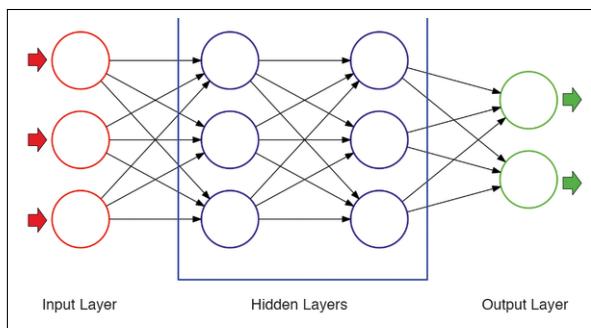
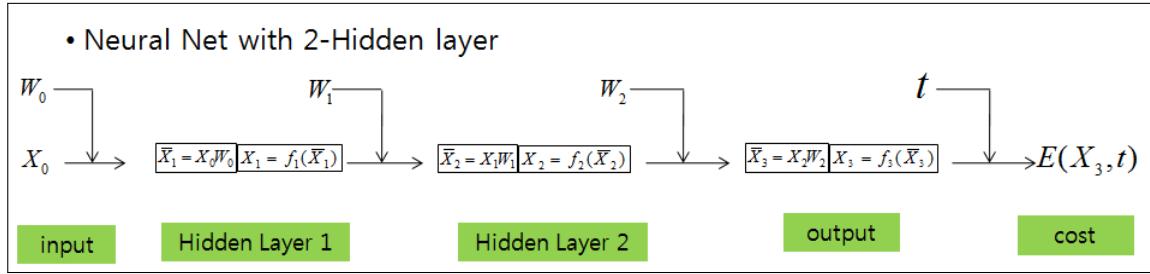


그림 7: Neural Network with 2-hidden layers

## 4 Back Propagation

- Neural Net with 2-Hidden layer(Back Propagation)

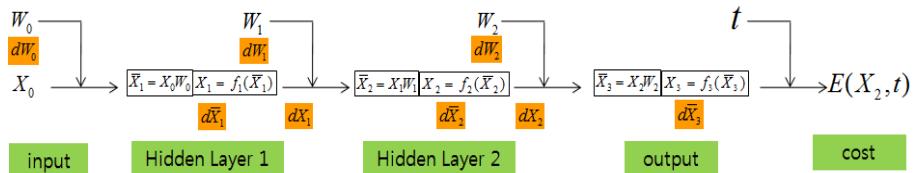


그림 8: Back Propagation

$X_i, \bar{X}_i$ 에 대한 미분값은 뒤로 계속 흘러가고,  $W_i$ 는 각 layer마다 계산된다. 미분값이 뒤로 계속 흘러가는 것을 전파(back propagation) 된다고 하는 것이다.

### 4.1 A Neural Network in 11 Lines

```

01. x = np.array([ [0,0,1],[0,1,1],[1,0,1],[1,1,1] ])
02. y = np.array([[0,1,1,0]]).T
03. syn0 = 2*np.random.random((3,4)) - 1
04. syn1 = 2*np.random.random((4,1)) - 1
05. for j in xrange(60000):
06.     l1 = 1/(1+np.exp(-(np.dot(x,syn0))))
07.     l2 = 1/(1+np.exp(-(np.dot(l1,syn1))))
08.     l2_delta = (y - l2)*(l2*(1-l2))
09.     l1_delta = l2_delta.dot(syn1.T) * (l1 * (1-l1))
10.     syn1 += l1.T.dot(l2_delta)
11.     syn0 += x.T.dot(l1_delta)
    
```

그림 9: A Neural Network in 11 Lines

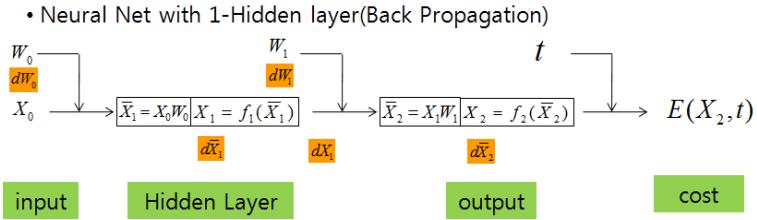


그림 10: Back Propagation with 1-Hidden Layer

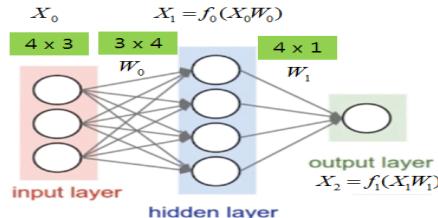


그림 11: Size of Neural Net

$$\begin{aligned}
 dX_2 &= -(t - X_2) \\
 d\bar{X}_2 &= dX_2 \circ (1 - X_2) \circ X_2 = -(t - X_2) \circ (1 - X_2) \circ X_2 \\
 dW_1 &= X_1^T d\bar{X}_2 \\
 dX_1 &= d\bar{X}_2 W_1^T \\
 d\bar{X}_1 &= dX_1 \circ (1 - X_1) \circ X_1 = d\bar{X}_2 W_1^T \circ (1 - X_1) \circ X_1 \\
 dW_0 &= X_0^T d\bar{X}_1
 \end{aligned}$$

```

1 x0 = np.array([[0,0,1],[0,1,1],[1,0,1],[1,1,1] ])
2 t = np.array([[0,1,1,0]]).T
3 W0 = 2*np.random.random((3,4)) - 1
4 W1 = 2*np.random.random((4,1)) - 1
5 for j in range(60000):
6     X1 = 1/(1+np.exp(-(np.dot(x0,W0))))
7     X2 = 1/(1+np.exp(-(np.dot(X1,W1))))
8     dX2 = (t - X2)*(X2*(1-X2))
9     dX1 = dX2.dot(W1.T) * (X1 * (1-X1))
10    W1 += X1.T.dot(dX2)
11    W0 += x0.T.dot(dX1)

```

그림 12: A Neural Network in 11 Lines

## 4.2 Linear Classifier

cs231n에 나오는 Linear Clasifier의 SVM Loss에 대하여 Back Propagation을 적용해 보자. 주어진 Dataset는  $\{(x_i, y_i)\}$ ,  $x_i$ 는 image,  $y_i$ 는 정수 label이다.

$$\begin{aligned}
 f(x_i, W) &= x_i W + b (= s^i) \\
 L &= \frac{1}{N} \sum_i^N L_i(f(x_i, W), y_i), \\
 \text{where } L_i &= \sum_{y \neq y_i} \max(0, s_j^i - s_{y_i}^i + 1)
 \end{aligned}$$

## Linear Classifier

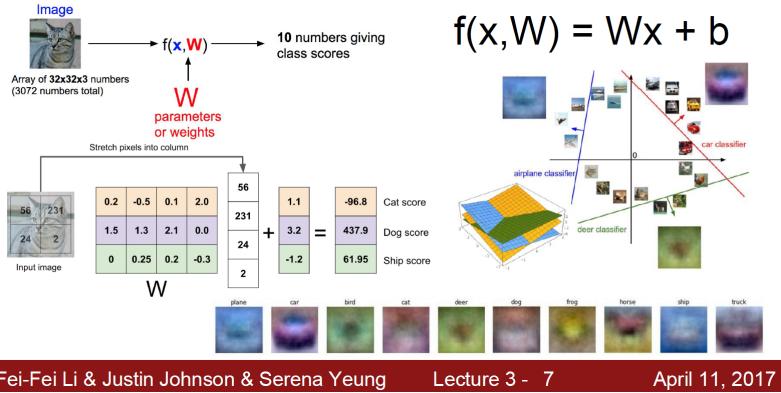


그림 13: Linear Classifier Model

위와 같이 정의된 Loss를 SVM Loss 또는 Hinge Loss라 하다.

---

```

# No bias
# y: label, shape(N,)
# C: number of class
scores = X.dot(W)

scores = scores - scores[np.arange(N), y].reshape(-1,1) + 1
scores[scores < 0] = 0
scores[np.arange(N), y] = 0
L = np.sum(scores)/N
L += labmda * np.sum(W * W)

dL = np.zeros((N,C))
dL[scores > 0] = 1
dL[np.arange(num_train), y] = -np.sum(dL, axis=1)
dW = np.dot(X.T, dL)/N + 2*labmda*W

```

---

## 5 Tensorflow & Network

### 5.1 Tensorflow

#### ♠ Tensorflow 기본 구조

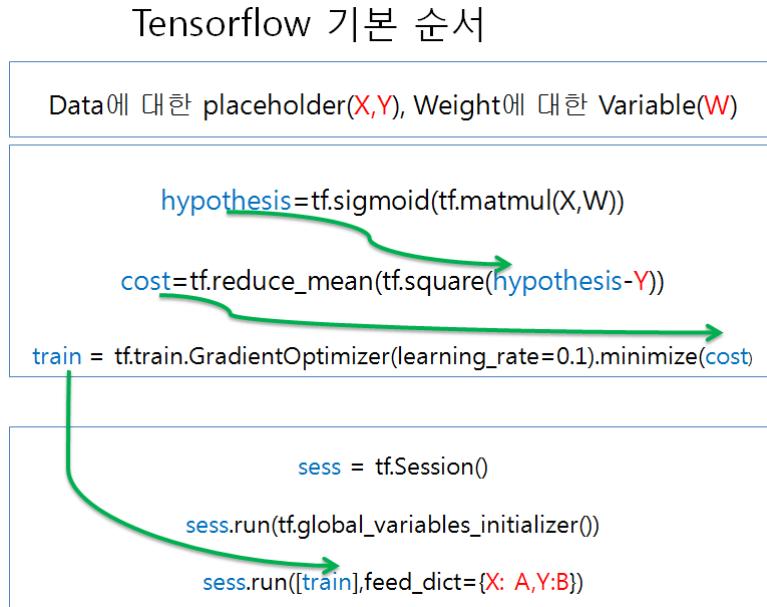


그림 14: Tensorflow 기본 흐름

#### ♠ placeholder

placeholder는 주로 Data를 담는다.

1. `train_mode = tf.placeholder(tf.bool)`
2. `learning_rate = tf.placeholder(tf.float32)`
3. `images = tf.placeholder(tf.float32, [None, 224, 224, 3])`
4. Annealing the learning rate<sup>1</sup>: learning rate을 training을 진행하면서 줄여주는 경우.  
`lr = tf.Variable(0.0, trainable=False)`로 선언하고, assign으로 조절.  
`sess.run(tf.assign(lr, learning_rate * (args.decay_rate ** e)))`

#### ♠ Tensorflow Weight 선언

Tensorflow에서 weight(bias) 변수 선언은 주로 다음의 방식으로 이루어진다.

- `tf.Varialbe`

```
W1 = tf.Variable([[ 0.8330605 ],[-0.35352278],[-0.44010514]], name='weight') # explicit  
initialization  
W2 = tf.Variable(tf.truncated_normal([4, 4], stddev=0.02), name="G_W1")
```

- `tf.get_variable`

<sup>1</sup><http://cs231n.github.io/neural-networks-3/>

---

```
W1 = tf.get_variable("W1", shape=[4,3], initializer=tf.contrib.layers.xavier_initializer())
W2 = tf.get_variable("W2", shape=[7, 7, 3, 32],
                     initializer=tf.constant_initializer(0.01*np.random.randn(7, 7, 3, 32)))
```

---

- 변수를 명시적으로 선언하지 않고, Tensorflow의 layer을 이용할 수도 있다.

---

```
self._X = tf.placeholder(tf.float32, [None, self.input_size], name="input_x")
net = self._X
net = tf.layers.dense(net, h_size, activation=tf.nn.relu)
net = tf.layers.dense(net, self.output_size)
```

---

## ♠ Optimization 3가지 방법

- gradient를 명시적으로 계산하지 않는 방법 :

---

```
optimizer = tf.train.AdamOptimizer(learning_rate=0.001)
optimizer.minimize(cost)
```

---

- gradient 계산을 분리 :

---

```
optimizer = tf.train.AdamOptimizer(learning_rate=0.001)
gradient = optimizer.compute_gradients(cost, [W])
update = optimizer.apply_gradients(gradient)
```

---

- optimizer 없이 gradient 직접 계산 후, update:

---

```
gradient = tf.matmul(tf.transpose(X), (hypothesis-Y)/N)
descent = W - learning_rate*gradient
update = W.assign(descent)
```

---

## ♠ checkpoint 파일로 save & restore

아래 예에서 'my-model' 이 저장할 때, 파일명의 일부로 들어간다. 파일 checkpoint에는 최근 5개의 checkpoint가 저장되어 있다.

---

```
self.saver = tf.train.Saver()
-----
self.saver.save(sess, 'my-model', global_step=0) ==> filename: 'my-model-0'
...
self.saver.save(sess, 'my-model', global_step=1000) ==> filename: 'my-model-1000'
...
self.saver.save(sess, os.path.join('checkpoint_dir', 'my-model'), global_step=1000) ==> checkpoint_dir
+ filename: 'my-model-1000'
-----
ckpt = tf.train.get_checkpoint_state(checkpoint_dir) # get ckpt filename
if ckpt and ckpt.model_checkpoint_path:
```

```

ckpt_name = os.path.basename(ckpt.model_checkpoint_path) # get last saved checkpoint from all
    checkpoint(ckpt.model_checkpoint_path)
self.saver.restore(self.sess, os.path.join(checkpoint_dir, ckpt_name))
counter = int(next(re.finditer("(\\d+)(?!.*\\d)", ckpt_name)).group(0)) # global_step parsing
print(" [*] Success to read {}".format(ckpt_name))
return True, counter

```

```

checkpoint file: 'save123' directory

model_checkpoint_path: "model.ckpt-22299"
all_model_checkpoint_paths: "model.ckpt-0"
all_model_checkpoint_paths: "model.ckpt-1000"
all_model_checkpoint_paths: "model.ckpt-2000"
all_model_checkpoint_paths: "model.ckpt-22299"

-----
ckpt.model_checkpoint_path ----> 'save123\\model.ckpt-22299'
ckpt.all_model_checkpoint_paths --> ['save123\\model.ckpt-0', 'save123\\model.ckpt-1000',
                                         'save123\\model.ckpt-2000', 'save123\\model.ckpt-22299']

```

## ♠ Tensorboard

- 아래의 예처럼, 같은 accuracy라도 train과 test에서 각각 계산해야 하므로, summary를 각각 만들어야 한다.
- log 저장 디렉토리는 없으면, 만든다.
- 간략하게 정리하면, 먼저 필요한 summary들을 만들고(필요하면 merge), FileWriter도 만든다. summary나 merge를 sess.run한다. 그 결과를 writer에 add\_summary한다.
- command창에서 tensorboard를 실행한다. >tensorboard --logdir=~/tensorbaord-logs
- tensorboard가 실행된 상태에서, crom 브라우저에서 http://localhost:6006
- 브라우저는 단순히 보여주는 역할만 한다. 먼저 생성된 log파일은 지워주어야 한다.
- summary는 scalar, image, histogram 등 있다.
- tensorboard에서 그래프는 tf.name\_scope, tf.variable\_scope를 잘 정해주어야 보기 편하다.

```

cost_summary = tf.summary.scalar("cost", cost)
acc_train_summary = tf.summary.scalar("acc_train", accuracy)
acc_test_summary = tf.summary.scalar("acc_test", accuracy)
#merged = tf.summary.merge_all()
merged = tf.summary.merge([cost_summary, acc_train_summary])
-----
writer = tf.summary.FileWriter("./tensorbaord-logs", sess.graph) # tensorboade Writer
-----
summary_str = sess.run(merged, feed_dict={...})
writer.add_summary(summary_str, step) # step: x-axis

```

```
-----  
summary_str = sess.run(acc_test_summary, feed_dict={...})  
writer.add_summary(summary_str, epoch)
```

## ♠ Tensorflow 기타

- `sess.run(tf.global_variables_initializer())` 대신에 `tf.global_variables_initializer().run()`으로 초기화 할 수도 있다.
- `sess.run(xxx, feed_dict={...})` 대신에 `xxx.eval(feed_dict = {...})`도 가능함.
- `tf.layers.dense`, `tf.contrib.layer.fully_connected`는 거의 같다. `dense`는 default activation function이 없고, `fully_connected`는 `relu`이다.
- `tf.random_normal`, `tf.truncated_normal`

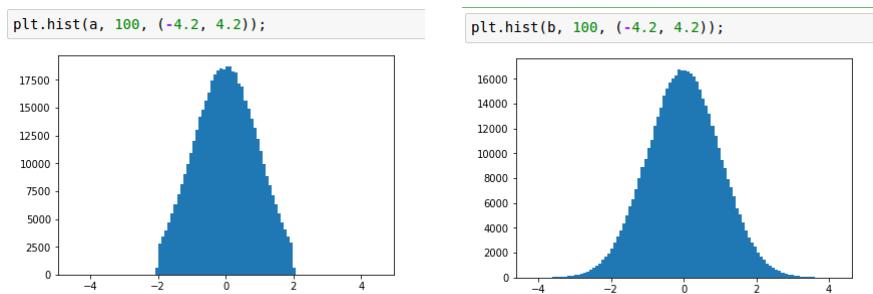


그림 15: Normal 비교

- `X.get_shape().as_list()`
- `tf.size(X)`는 tensor X의 전체 원소 개수.
- `tf.trainable_variables()`, `tf.global_variables()`
- regularization:

```
regularizer = tf.contrib.layers.l2_regularizer(scale=0.1)  
conv1_1 = tf.layers.conv2d(inputs=X, filters=32, padding='SAME', kernel_size=3, strides=1,  
                           kernel_regularizer=regularizer, activation=tf.nn.relu, name = 'conv1_1')
```

## 5.2 Network 설계

### ♠ Network

Network은 기본적으로 4개의 함수로 구성된다(`init, loss, train, predict`). `loss` 함수와 `weight(param dict)`은 Trainer(Solver)에서 불려진다.

- `__init__`: weights 생성. 생성된 weight는 `self.params dict`에 잘 저장되어야, Trainer에서 update 된다.
- `loss`: loss값과 gradient 계산.
- `train`: loss함수를 반복 호출하여 gradient update.

- predict: train 결과로 부터 새로운 data에 대한 예측.
- 

```
class TwoLayerNet(object):

    def __init__(self, input_size, hidden_size, output_size, std=1e-4):
        self.params = {}
        self.params['W1'] = std * np.random.randn(input_size, hidden_size)
        self.params['b1'] = np.zeros(hidden_size)
        self.params['W2'] = std * np.random.randn(hidden_size, output_size)
        self.params['b2'] = np.zeros(output_size)

    def loss(self, X, y=None, reg=0.0):
        # Unpack variables from the params dictionary
        W1, b1 = self.params['W1'], self.params['b1']
        W2, b2 = self.params['W2'], self.params['b2']
        N, D = X.shape

        # Compute the forward pass
        scores = None

        h1 = X.dot(W1) + b1
        mask = (h1 <= 0) # for Relu
        h1[mask] = 0

        scores = h1.dot(W2)+b2
        z = softmax(scores)

        # Compute the loss
        loss = np.sum(-np.log(z[np.arange(N),y]+0.0000001))/N + reg*(np.sum(W1*W1)+np.sum(W2*W2))

        t = np.zeros_like(z)
        t[np.arange(N),y] = 1 # one-hot
        dL = (z-t)/N
        dW2 = np.dot(h1.T,dL) + 2*reg*W2
        db2 = np.sum(dL, axis=0)
        dh1 = np.dot(dL,W2.T)
        dh1[mask] = 0

        dW1 = np.dot(X.T,dh1) + 2*reg*W1
        db1 = np.sum(dh1, axis=0)

        # Backward pass: compute gradients
        grads = {}
        grads['W2'] = dW2; grads['W1'] = dW1
        grads['b2'] = db2; grads['b1'] = db1

    return loss, grads
```

---

---

```

def train(self, X, y, X_val, y_val, learning_rate=1e-3, learning_rate_decay=0.95, reg=5e-6,
          num_iters=100, batch_size=200, verbose=False):
    num_train = X.shape[0]
    iterations_per_epoch = max(num_train / batch_size, 1)

    # Use SGD to optimize the parameters in self.model
    loss_history = []
    train_acc_history = []
    val_acc_history = []

    for it in range(num_iters):
        rand_idx = np.random.choice(num_train, batch_size)
        X_batch = X[rand_idx]
        y_batch = y[rand_idx]

        # Compute loss and gradients using the current minibatch
        loss, grads = self.loss(X_batch, y=y_batch, reg=reg)
        loss_history.append(loss)

        for key in grads:
            self.params[key] -= learning_rate * grads[key]

        if verbose and it % 100 == 0:
            print('iteration %d / %d: loss %f' % (it, num_iters, loss))

        # Every epoch, check train and val accuracy and decay learning rate.
        if it % iterations_per_epoch == 0:
            # Check accuracy
            train_acc = (self.predict(X_batch) == y_batch).mean()
            val_acc = (self.predict(X_val) == y_val).mean()
            train_acc_history.append(train_acc)
            val_acc_history.append(val_acc)

            # Decay learning rate
            learning_rate *= learning_rate_decay

    return {'loss_history': loss_history, 'train_acc_history': train_acc_history, 'val_acc_history':
            val_acc_history}

```

---

```

def predict(self, X):
    W1, b1 = self.params['W1'], self.params['b1']
    W2, b2 = self.params['W2'], self.params['b2']
    h1 = X.dot(W1) + b1
    mask = (h1 <= 0) # for Relu
    h1[mask] = 0
    y_pred = np.argmax(h1.dot(W2)+b2, axis=1) #argmax: not need softmax
    return y_pred

```

- 
- layer를 함수로 만드는 경우(cs231n): forward, backward를 각각 만들고, backward에 필요한 cache를 forward 계산할 때 받아서 저장해 놓아야 한다.
  - layer를 class로 만들면(밑바닥딥러닝), cache를 따로 저장할 필요 없다.

## ♠ Optimizer & Trainer

- Optimizer(SGD, Adam, ...)는 update함수에서 gradient를 update한다.

---

```
class SGD:
    def __init__(self, lr=0.01):
        self.lr = lr

    def update(self, params, grads):
        for key in params.keys():
            params[key] -= self.lr * grads[key]
```

---

- Trainer는 (Network, Dataset, Optimizer, OptimizerParam)를 받아 들여서 training을 수행한다.
- Network Class에서
  - Gradient, Loss 2개의 함수를 만들 수도 있고(밑바닥딥러닝), Loss 1개만 만들어 gradient, loss 모두를 계산(cs231n) 할 수도 있다.
  - Gradient, Loss 2개의 함수를 분리해서 만들면, Gradient 내부에서, forward 계산 후, backward를 순서대로 계산해야 함.

## ♠ Network with Tensorflow 1

Weight를 명시적으로 선언하지 않고, Tensorflow 함수만을 이용하여 Network 구성.

- tf.nn.conv2d: 명시적으로 weight(filter)를 정의해서 넣어 주어야 함.
- tf.layers.conv2d: 명시적으로 weight를 넣지 않는다.

---

```
def MyCNN(X,y,is_training):
    conv1_1 = tf.layers.conv2d(inputs=X, filters=32, padding='SAME', kernel_size=3, strides=1,
                             activation=tf.nn.relu, name = 'conv1_1')
    conv1_2 = tf.layers.conv2d(inputs=conv1_1, filters=32, padding='SAME', kernel_size=3, strides=1,
                             activation=tf.nn.relu, name = 'conv1_2')
    pool1_3 = tf.layers.max_pooling2d(inputs=conv1_2, pool_size=2, strides=2, padding='SAME', name=
                                         'pool1_3') # ==> (?, 16, 16, 32)

    conv2_1 = tf.layers.conv2d(inputs=pool1_3, filters=64, padding='SAME', kernel_size=3, strides=1,
                             activation=tf.nn.relu, name = 'conv2_1')
    conv2_2 = tf.layers.conv2d(inputs=conv2_1, filters=64, padding='SAME', kernel_size=3, strides=1,
                             activation=tf.nn.relu, name = 'conv2_2')
    pool2_3 = tf.layers.max_pooling2d(inputs=conv2_2, pool_size=2, strides=2, padding='SAME', name=
                                         'pool2_3') # ==> (?, 8, 8, 64)
```

```

conv3_1 = tf.layers.conv2d(inputs=pool2_3, filters=128, padding='SAME', kernel_size=3, strides=1,
    activation=tf.nn.relu, name = 'conv3_1')
conv3_2 = tf.layers.conv2d(inputs=conv3_1, filters=128, padding='SAME', kernel_size=3, strides=1,
    activation=tf.nn.relu, name = 'conv3_2')
pool3_3 = tf.layers.max_pooling2d(inputs=conv3_2, pool_size=2, strides=2, padding='SAME', name=
    'pool3_3') # ==> (?, 4, 4, 128)

flatten1 = tf.reshape(pool3_3, [-1, 2048])
mode = 1
if mode == 1:
    fc3_1 = tf.layers.dense(inputs=flatten1, units=1024, activation=tf.nn.relu, name='fc3_1')
    drop3_2 = tf.layers.dropout(inputs=fc3_1, rate=0.5, training=is_training, name='drop3_2')
    fc4_1 = tf.layers.dense(inputs=drop3_2, units=1024, activation=tf.nn.relu, name='fc4_1')
else:
    fc3_1 = tf.layers.dense(inputs=flatten1, units=1024, activation=None, name='fc3_1')
    bn3_2 = tf.layers.batch_normalization(inputs=fc3_1, training=is_training, name='bn3_2')
    relu3_3 = tf.nn.relu(bn3_2, name='relu3_3')
    fc4_1 = tf.layers.dense(inputs=relu3_3, units=1024, activation=tf.nn.relu, name='fc4_1')

y_out = tf.layers.dense(inputs=fc4_1, units=10, activation=None, name='out')

return y_out

```

---

## ♠ Network with Tensorflow 2

Weight를 명시적으로 선언 후, Network 설계

```

def MyCNN2(X,y,is_training):
    weight_scale = 0.01
    filter_size = 3
    Wconv1 = tf.get_variable("Wconv1", shape=[filter_size, filter_size, 3, 32],
        initializer=tf.constant_initializer(weight_scale*np.random.randn(filter_size, filter_size, 3,
            32)))
    bconv1 = tf.get_variable("bconv1", shape=[32], initializer=tf.constant_initializer(np.zeros(32)))
    W4 = tf.get_variable("W4", shape=[8192,
        1024], initializer=tf.constant_initializer(weight_scale*np.random.randn(8192, 1024))) # 16 x
    16 x 32 = 8192
    b4 = tf.get_variable("b4", shape=[1024], initializer=tf.constant_initializer(np.zeros(1024)))

    W6 = tf.get_variable("W6", shape=[1024,
        10], initializer=tf.constant_initializer(weight_scale*np.random.randn(1024, 10))) # 13 x 13 x
    32 = 5408
    b6 = tf.get_variable("b6", shape=[10], initializer=tf.constant_initializer(np.zeros(10)))

    h1 = tf.nn.conv2d(X, Wconv1, strides=[1,1,1,1], padding='SAME') + bconv1
    h2 = tf.nn.relu(h1)
    h2_flat = tf.reshape(h2, [-1, 32])
    h3_flat = tf.contrib.layers.batch_norm(h2_flat, decay=0.9, updates_collections=None, epsilon=1e-5,

```

```
        center=True,scale=True,is_training=is_training,scope='BN')

h3 = tf.reshape(h3_flat,[-1,32,32,32])

h4 = tf.nn.max_pool(h3, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME', name='Pooling')
    # 16 x 16 x 32 = 8192

h4_flat = tf.reshape(h4,[-1,8192])

h5 = tf.matmul(h4_flat,W4) + b4

h6 = tf.nn.relu(h5)

y_out = tf.matmul(h6,W6) + b6

loss_reg = tf.nn.l2_loss(Wconv1) + tf.nn.l2_loss(W4) + tf.nn.l2_loss(W6)

return y_out, loss_reg
```

---

## 6 Convolution Neural Network

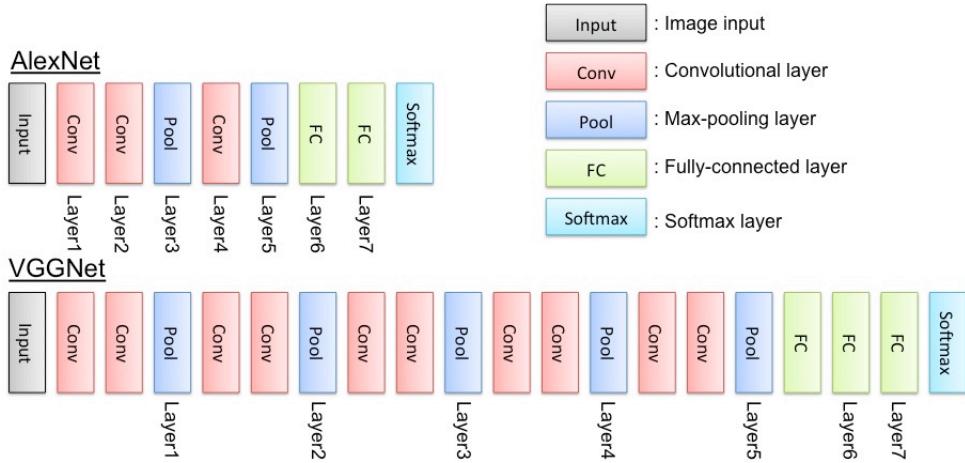


그림 16: CNN Architecture. Convolution Layer는 ReLU를 포함하고 있음.

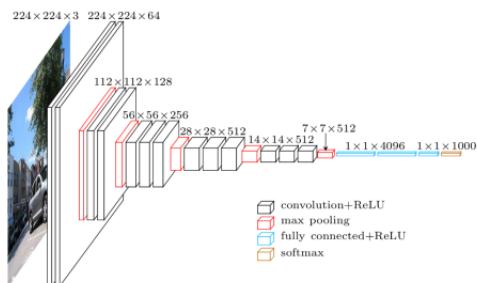


그림 17: VGG16 Architecture. GoogLeNet보다 성능은 떨어지지만, 구성이 간단하여 응용하기 좋은 장점이 있음.  
단점으로는 메모리 사용과 연산이 많이 필요함

### 6.1 im2col

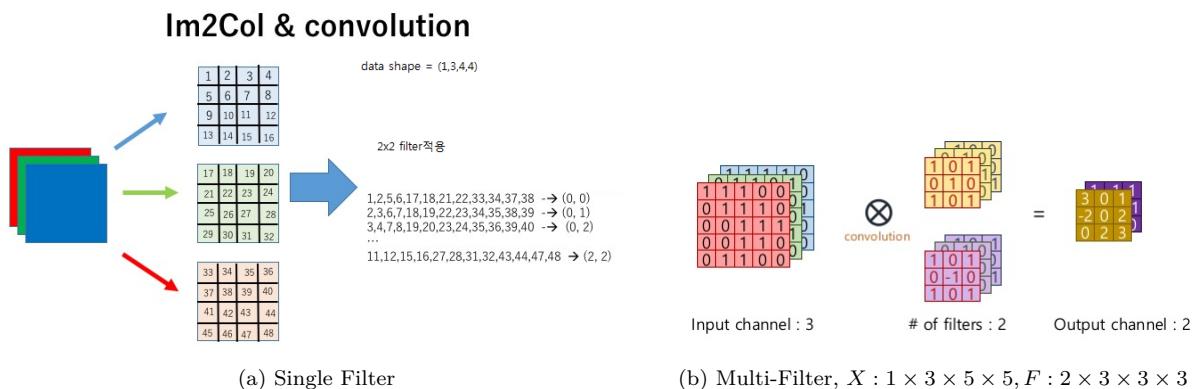


그림 18: Im2Col

- im2col은 각각 4차원 array로 된 입력 data와 필터를 2차원 array로 변화하여 효율적인 계산을 목적으로 한다.
- 입력 Data  $X : N \times C \times H \times W$  와 필터  $F : FN \times C \times FH \times FW$  가 주어져 있다고 하자. Convolution에서  $X, F$ 의 채널 갯수  $C$ 는 같은 값이어야 한다.

- im2col함수는  $(X, FH, FW)$ 를 입력 받아,  $(N \times OH \times OW) \times (FH \times FW \times C)$  크기의 2차원 array를 만든다. 여기서  $OH, OW$ 는 공식에 의해 계산되는 값이다.

---

`im2col(X, FH, FW)`

---

- $F$ 도 재배열하여  $(FH \times FW \times C) \times FN$  크기의 2차원 array로 변환한다.

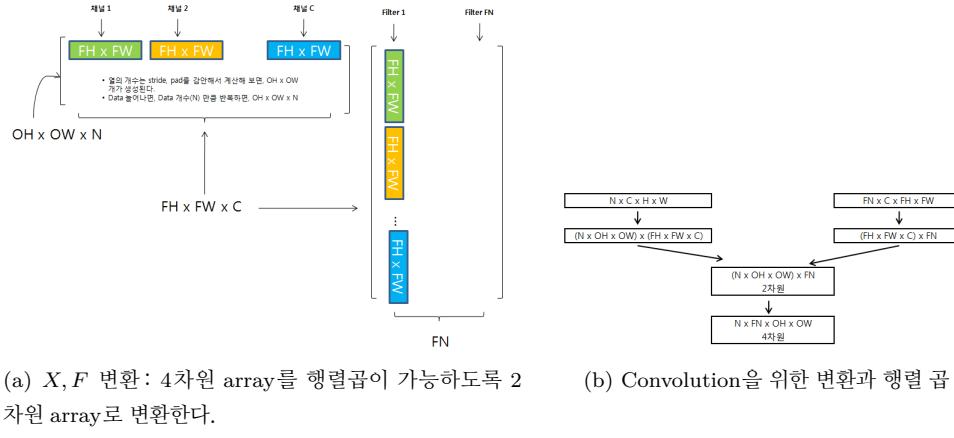


그림 19: Convolution

#### ♠ 그림 (20)에 대한 설명

1.  $N = 1, C = 3, H \times W = 7 \times 7$  image, data shape =  $(1, 3, 7, 7)$ . 이 Data를 주어진 필터와 convolution하기 위해서는 계산 목적으로 Data를 im2col 변환하는 것이 필요하다.
2.  $FN = 1, FH = 5, FW = 5, \text{stride} = 1, \text{pad} = 0$  인 필터를 적용하기 위한 im2col 변환을 생각해 보자.
3. im2col 변환은 필터  $5 \times 5$  만큼의 Data를 행으로 펼쳐 첫번째 행를 생성한다. 2번째 행은 window를 이동시켜  $5 \times 5$  Data를 행으로 펼친다. 이렇게 새로운 Data를 생성하면,  $9 (= 3 \times 3 = OH \times OW)$  만큼의 행이 만들어 진다.
4. 채널 갯수  $C$ 만큼 열 size가  $C$ 배 늘어난다. im2col의 결과는  $(N \times OH \times OW) \times (FH \times FW \times C)$  크기의 행렬이 만들어 진다.
5. 생성된 im2col 결과를 다른 관점에서 볼 수도 있다. im2col 열(숫자 9개)을 순서대로 보면, 입력 Data를  $3 \times 3$  window를 이동시키면서 생성한 것으로 볼 수도 있다.
6. 이제, Back Propagation에서 dcol로 부터 입력 Data에 대한 gradient를 계산해 보자.
7. [그림 20]를 보면 im2col에서 22가 나타나는 위치의 dcol의 값을 합한 것이 22에 대한 gradient가 된다.

$$(-0.53) + 0.77 + 0.05 + 0.03 + 0.29 + (-0.29) = 0.32$$

8. 참고로 Pooling에서의 Output image의 크기는 다음과 같다.

$$POH = \frac{H - PH}{\text{stride}} + 1$$

그림 20: im2col 예:  $N = 1, C = 3, H \times W = 7 \times 7$  image. data shape = (1,3,7,7),  $FH = 5, FW = 5$ , stride = 1, pad = 0. 입력 Data를 필터 size에 따라 im2col변환 후, back propagation을 위해 넘어온 dcol로 부터 입력 Data에 대한 gradient 계산.

## 6.2 VGG의 이해

VGG는 ILSVRC 2014년에 2등(오류율 7.3%, 1등은 GoogLeNet) 한 모델이다<sup>2</sup>. Tensorflow VGG16 and VGG19<sup>3</sup>를 분석하여 VGG모델을 이해해 보자. VGG19는 VGG16보다 3,4,5번째 Convolution Layer 그룹에 Convolution Layer가 1개씩 더 있다는 것만 다르다.

- training된 data(vgg16.npy, vgg19.npy)를 다운 받을 수 있다<sup>4</sup>.
  - 이 npy 파일은 dict 형 data를 가지고 있는 numpy array이다.

```
data_dict = np.load('vgg16.npy', encoding='latin1').item()
data_dict.keys(): ['conv5_4', 'conv5_1', 'fc6', 'conv5_3', 'conv5_2', 'conv4_4', 'fc7',
    'conv4_1', 'conv4_2', 'conv4_3', 'fc8', 'conv3_4', 'conv3_3', 'conv3_2', 'conv3_1',
    'conv1_1', 'conv1_2', 'conv2_2', 'conv2_1']
```

예를 들어, `data_dict['conv1_1']`은 len 2인 list이고, `data_dict['conv1_1'][0]`에는 filter 값, `data_dict['conv1_1'][1]`에는 bias가 들어 있다. `data_dict['fc6']`도 len 2인 list이고, `data_dict['fc6'][0]`에는 weight 값, `data_dict['fc6'][1]`에는 bias가 들어 있다.

<sup>2</sup><https://arxiv.org/abs/1409.1556>

<sup>3</sup><https://github.com/machrisaa/tensorflow-vgg>

<sup>4</sup>각각 약 550M의 크기. 이 training data는 1,000개의 image category에 대하여 훈련된 결과를 가지고 있다. ImageNet의 image data category는 8만2천개가 넘는다.

- o] `data_dict`에는 길이 2인 list가 있는데, weight(4차원 numpy array) 와 bias(numpy array) 가 들어 있다.  
weight 는

$$FH \times FW \times C \times FN$$

순으로 되어 있는데, 이 순서는 TensorFlow(`tf.nn.conv2d`)가 convolution의 weight를 받아들이는 형태이다.

- 처음 입력되는 image data는 4차원 numpy array인데, 다음과 같은 shape을 가진다.

$$N \times H \times W \times C$$

- 정리하면, convolution의 input, weight, output shape은 다음과 같다.

$$(N \times H \times W \times C) \otimes (FH \times FW \times C \times FN) \Rightarrow (N \times OH \times OW \times FN)$$

```
VGG16 Weight & Bias:
input_data
-> conv1_1 (3, 3, 3, 64) (64,) -> relu -> conv1_2 (3, 3, 64, 64) (64,) -> relu -> pooling
-> conv2_1 (3, 3, 64, 128) (128,) -> relu -> conv2_2 (3, 3, 128, 128) (128,) -> relu -> pooling
-> conv3_1 (3, 3, 128, 256) (256,) -> relu -> conv3_2 (3, 3, 256, 256) (256,) -> relu
    -> conv3_3 (3, 3, 256, 256) (256,) -> relu -> pooling
-> conv4_1 (3, 3, 256, 512) (512,) -> relu -> conv4_2 (3, 3, 512, 512) (512,) -> relu
    -> conv4_3 (3, 3, 512, 512) (512,) -> relu -> pooling
-> conv5_1 (3, 3, 512, 512) (512,) -> relu -> conv5_2 (3, 3, 512, 512) (512,) -> relu
    -> conv5_3 (3, 3, 512, 512) (512,) -> relu -> pooling
-> fc6 (25088, 4096) (4096,) -> relu -> dropout
-> fc7 (4096, 4096) (4096,) -> relu -> dropout
-> fc8 (4096, 1000) (1000,) -> softmax

VGG19 Weight & Bias:
input_data
-> conv1_1 (3, 3, 3, 64) (64,) -> relu -> conv1_2 (3, 3, 64, 64) (64,) -> relu -> pooling
-> conv2_1 (3, 3, 64, 128) (128,) -> relu -> conv2_2 (3, 3, 128, 128) (128,) -> relu -> pooling
-> conv3_1 (3, 3, 128, 256) (256,) -> relu -> conv3_2 (3, 3, 256, 256) (256,) -> relu
    -> conv3_3 (3, 3, 256, 256) (256,) -> relu -> conv3_4 (3, 3, 256, 256) (256,) -> relu -> pooling
-> conv4_1 (3, 3, 256, 512) (512,) -> relu -> conv4_2 (3, 3, 512, 512) (512,) -> relu
    -> conv4_3 (3, 3, 512, 512) (512,) -> relu -> conv4_4 (3, 3, 512, 512) (512,) -> relu -> pooling
-> conv5_1 (3, 3, 512, 512) (512,) -> relu -> conv5_2 (3, 3, 512, 512) (512,) -> relu
    -> conv5_3 (3, 3, 512, 512) (512,) -> relu -> conv5_4 (3, 3, 512, 512) (512,) -> relu -> pooling
-> fc6 (25088, 4096) (4096,) -> relu -> dropout
-> fc7 (4096, 4096) (4096,) -> relu -> dropout
-> fc8 (4096, 1000) (1000,) -> softmax
```

- Input Image Data: `load_image` 함수에서는 입력되는 이미지의 중앙을 중심으로 잘라내어,  $(224 \times 224 \times 3)$  크기의 파일로 변환한다. 변환된 image의 값은  $0 \sim 1$ 이 된다. 이런 방식으로 각각 변형된 이미지는 결합되어  $(N \times 224 \times 224 \times 3)$  형태의 numpy array로 만들어진다.

---

```
load_image(filename)
```

---

- `load_image`를 통해 만들어진 data는 `tf.placeholder`를 통해서 Tensor로 변환된다. 그리고, VGG16, VGG19 의 Forward Propagation을 수행하는 'build' 함수로 넘어간다.

- ‘build’ 함수는 넘어온 image data에 대하여, Normalization을 수행한다. 다시 255를 곱하고, VGG\_MEAN = (103.939, 116.779, 123.68)<sup>5</sup> 값을 빼서 정규화 시킨다. 다시 말해, VGG 모델은 image data를 0~1 값으로 변환하지 않고, 평균값을 차감하는 normalization을 사용한다.
- 구현된 code의 설명에서는  $FH, FW$ 를 kernel\_size,  $FN$ 을 num\_output으로 말하고 있다.
- Convolution과 Relu는 하나로 묶여서 구현되어 있다.
- 마지막 Convolution인 ‘Convolution-5’ 통과 후 나오는 4차원 array는 Full Connected Layer에 들어가면, 2 차원 array로 변환되어 계산이 진행된다.

VGG16	Input/Output	Filter/Weight $FH, FW, \text{in-channels, out-channels}$	
Convolution 1-1& Relu	$N \times H \times W \times C$ $N \times 224 \times 224 \times 3$	$FH \times FW \times C \times FN$ $3 \times 3 \times 3 \times 64$	Input의 4번째 size와 Weight의 3번째 size는 일치. Weight의 4번째 size는 Output의 4번째 size와 일치한다. stride = 1. TensorFlow(tf.nn.conv2d)에서 padding='SAME'을 적용하면 Output size 가 Input size/stride와 동일하게 되는데, 이는 padding size = 1에 해당한다.
Convolution 1-2& Relu	$N \times 224 \times 224 \times 64$	$3 \times 3 \times 64 \times 64$	kernel size = 3, $FN = 64$
Max Pooling 1	$N \times 224 \times 224 \times 64$	stride=2,kernel_size =2	padding='SAME'을 적용하면, output size = Int(H/stride)가 된다. 이 경우, 실질적인 padding =0. image size는 반으로 줄어든다.
Convolution 2-1& Relu	$N \times 112 \times 112 \times 64$	$3 \times 3 \times 64 \times 128$	
Convolution 2-2& Relu	$N \times 112 \times 112 \times 128$	$3 \times 3 \times 128 \times 128$	
Max Pooling 2	$N \times 112 \times 112 \times 128$	stride=2,kernel_size =2	
Convolution 3-1& Relu	$N \times 56 \times 56 \times 128$	$3 \times 3 \times 128 \times 256$	
Convolution 3-2& Relu	$N \times 56 \times 56 \times 256$	$3 \times 3 \times 256 \times 256$	
Convolution 3-3& Relu	$N \times 56 \times 56 \times 256$	$3 \times 3 \times 256 \times 256$	
Max Pooling 3	$N \times 56 \times 56 \times 256$	stride=2,kernel_size =2	
Convolution 4-1& Relu	$N \times 28 \times 28 \times 256$	$3 \times 3 \times 256 \times 512$	
Convolution 4-2& Relu	$N \times 28 \times 28 \times 512$	$3 \times 3 \times 512 \times 512$	
Convolution 4-3& Relu	$N \times 28 \times 28 \times 512$	$3 \times 3 \times 512 \times 512$	
Max Pooling 4	$N \times 28 \times 28 \times 512$	stride=2,kernel_size =2	
Convolution 5-1& Relu	$N \times 14 \times 14 \times 512$	$3 \times 3 \times 512 \times 512$	
Convolution 5-2& Relu	$N \times 14 \times 14 \times 512$	$3 \times 3 \times 512 \times 512$	
Convolution 5-3& Relu	$N \times 14 \times 14 \times 512$	$3 \times 3 \times 512 \times 512$	
Max Pooling 5	$N \times 14 \times 14 \times 512$	stride=2,kernel_size =2	
FC 6 & Relu 6 & Dropout	$N \times 7 \times 7 \times 512$	$25088 \times 4096$	$7 \times 7 \times 512 = 25088$ , KeepProb = 0.5
FC 7 & Relu 7 & Dropout	$N \times 4096$	$4096 \times 4096$	KeepProb = 0.5
FC 8 & Softmax	$N \times 4096$	$4096 \times 1000$	
Prob	$N \times 1000$		

- VGG16 TensorFlow 구현을 보면,  $C = FH = FW = 3$ 으로 고정되어 있다.
- convolution layer 구현은 다음과 같다.

```

filt = tf.truncated_normal([filter_size, filter_size, in_channels, out_channels], 0.0, 0.001) #
    filter_size =3, in_channels = C, out_channels = FN
conv_biases = tf.truncated_normal([out_channels], .0, .001)

```

<sup>5</sup>Blue,Green,Red순. 순서를 RGB로 하면, 코드가 간단해질 수 있는데, 왜 BGR순서로 해놓고 복잡하게 만드는지... 어떤 코드들은 RGB로 순서를 재배열 한 것도 있네.

```
conv = tf.nn.conv2d(bottom, filt, [1, 1, 1, 1], padding='SAME') # input, filter, stride, padding
bias = tf.nn.bias_add(conv, conv_biases)
relu = tf.nn.relu(bias)
```

---

- MaxPooling 구현.

```
tf.nn.max_pool(bottom, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME', name=name) #
    input, kernel size, stride, padding
```

---

- DropOut구현: training에서는 포함되고, prediction에서는 포함되지 않는다.

```
tf.nn.dropout(input, keep_prob) # input, keep_prob
```

---

- FC layer구현

```
weights, biases = self.get_fc_var(in_size, out_size, name)
x = tf.reshape(bottom, [-1, in_size]) # if first FC, reshape
fc = tf.nn.bias_add(tf.matmul(x, weights), biases)
```

---

### 6.3 Image File Loading & Save

#### ♠ 64 × 64 Images

100개의 category로 된 그림파일들로 작업해 보자. 각 category는 500개의  $64 \times 64 \times 3$  크기의 image file(jpeg)로 되어 있다. 모두 5만개의 파일들인데, 파일 사이즈는 약 100M정도 이다.

- 파일 전체를 로드하는데는 시간이 지남에 따라, 같은 수의 파일을 로드하는데도 점점 느려짐.
- 그래서 10또는 20개의 npz파일로 나누어 저장함. 10개의 npz로 나눌 경우, 각각의 npz파일 크기는 150M정도.
- 나누어지 npz파일을 np.concatenate로 다시 합치면, 약 1.5G. numpy array data를 float으로 하지 않고, np.int 형으로 저장하면 파일 사이즈는 절반으로 줄어든다. pickle파일로 저장할 경우, 파일 사이즈는 훨씬 커짐.
- 1.5G의 npz파일을 메모리로 올리면, 4.8G정도가 된다.
- 합쳐진 image data전체는 많은 메모리를 차지하기 때문에, 메모리가 충분하지 못할 경우를 대비하여 image의 순서를 shuffle하여 10개 정도로 나누어 저장.
- 이렇게 작업한 후, VGG모델은  $244 \times 224$  크기의 이미지를 사용하는 모델임을 다시 알게됨. 다운 받은  $64 \times 64$  를 resize해야 됨.

```
resized_img = skimage.transform(img,(224,224),preserve_range = True)
```

---

- preserve\_range = False 이면, 0 ~ 1 값으로 바뀜. True로 옵션을 주면, 정수 형태지만 type은 np.float. 실행 속도도 느림.
- 어째든 해상도가 낮아, VGG모델을 수정하여 저해상도에 맞게 적용하면 되겠지만, VGG를 바로 적용해 보고 싶어서 다른 Dataset을 찾아보기로 함.

## ♠ Deep Learning Dataset

Deep Learning Dataset<sup>6</sup>에는 다양한 music, image 등의 data가 link되어 있다.

- Caltech256 image: category가 256개(256\_ObjectCategories.tar-1.15G<sup>7</sup>). image들의 size는 동일하지 않고 다양함(eg.  $499 \times 278$ ,  $200 \times 150$ ,  $1024 \times 470$ , ...). 전체 image 개수는 30,608개.
- Caltech101 image: category가 101개<sup>8</sup>. category별 image 개수 차이가 많이 난다(eg. car-side는 모두 흑백 이미지, motorbikes는 798개). 전체 image 개수는 9,145개. int형의 npz 파일로 저장하면 1.12G(흑백 이미지를 제외하면 8,733개)
- Deep Learning을 수행하기 위해서는 category당 1,000여개의 이미지가 있어야 하는데, category당 80 ~ 90 개는 너무 적다.
- category별로 보면, 40개 category는 50장 이하, 49개 category는 50 ~ 100장, 그 외 category는 100장, 200 장, 400장, 800장 등.
- category별 이미지 개수의 편차가 심해, VGG training data로 사용하기에는 부족함.

## ♠ VGG Dataset

VGG Dataset<sup>9</sup>에서도 여러가지 data를 구할 수 있다.

- Pet Dataset: 25 category의 개 사진과 12 category의 고양이 사진이 7,390장 있다. category별로 200개 정도로 균일하다.
- 'Egyptian\_Mau\_129.jpg', 'staffordshire\_bull\_terrier\_2.jpg', 'staffordshire\_bull\_terrier\_22.jpg' 3개의 이미지는 흑백이라 제외. 남은 data는 총 7,387개.
- 위에서 마찬가지로 많은 이미지 파일을 단일 loop로 읽어 들이면, 점점 느려지기 때문에, category 1개씩 batch로 처리하여 37개의 npz파일을 먼저 생성.
- 37개 npz파일을 Memory에 load하고 이미지 순서를 shuffle한 후, 10개의 npz파일로 나누어 저장함.
- 이렇게 구한 data로 이용하여, Classification 구현은 github<sup>10</sup>에 있다.

## ♠ CIFAR-10 and CIFAR-100

The CIFAR-10 and CIFAR-100 are labeled subsets of the 80 million tiny images dataset. They were collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton.<sup>11</sup>

- The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.
- The CIFAR-100 dataset is just like the CIFAR-10, except it has 100 classes containing 600 images each. There are 500 training images and 100 testing images per class.

---

<sup>6</sup><http://deeplearning.net/datasets/>

<sup>7</sup>압축 풀면, 257개 category

<sup>8</sup>128M. 이것도 실제 102개 category

<sup>9</sup><http://www.robots.ox.ac.uk/~vgg/data/>

<sup>10</sup><https://github.com/hccho2/CNN-VGG16-DogCat-Classification>

<sup>11</sup><https://www.cs.toronto.edu/~kriz/cifar.html>

## 7 Recurrent Neural Network

일반적인 Neural Network는 RNN과 대비하기 위해 Feed Forward Neural Network라고 하기도 한다. 기본적인 RNN 모델을 LSTM과 같이 복잡한 모델과 대비하기 위해 Vanilla RNN이라 부르기도 한다. RNN은 speech recognition, language modeling, translation, image captioning 등에 활용된다.

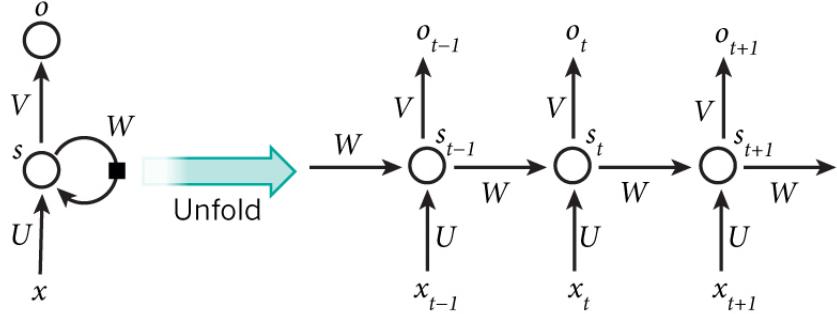


그림 21: RNN

$$\begin{aligned}s_{-1} &= 0, \\ s_t &= f(Ux_t + Ws_{t-1} + b_h), \text{ where } f = \tanh \text{ or ReLU,} \\ o_t &= \text{SoftMax}(Vs_t + b_y)\end{aligned}$$

모든  $t$ 에 동일한  $U, W, V, b_h, b_y$ 를 적용하는데, shared weights라고 한다. RNN에서 activation function로 sigmoid 대신에 tanh를 사용하는 이유는 gradient vanishing 현상을 줄이기 위하여 2차 미분도 0이 아닌 함수가 필요하기 때문이다.

일반적으로 Deep하게 쌓으면, Neural Network은 성능이 좋아지므로, 독립적인 RNN(또는 LSTM)을 여러개 쌓아서 Deep Networks을 구성할 수 있다.

$$\begin{aligned}y_1 &= RNN_1(x) \\ y_2 &= RNN_2(y_1) \\ &\vdots \\ y &= RNN_n(y_{n-1})\end{aligned}$$

### 7.1 RNN Toy Model: i am trask blog

'I am trask' blog에 있는 이진수 2개를 더하는 RNN모델을 수식으로 다시 정리한 내용입니다.<sup>12</sup>

$n = 7$ 자리 2진수( $0 \sim 2^7 - 1$ ) 2개를 더하는 training을 수행한다. 예를 들어, 9(00001001), 60(00111100)를 입력하면 결과로 69(01000101) 가 나오기를 기대하는 Model을 만들고자 한다. 7자리 2진수를 더하기 때문에 그 합은 8자리 까지 된다. (RNN에서는 batch size = 1이 기본이다.)

output( $y_i$ )	1	0	1	0	0	0	1	0
input	1	0	0	1	0	0	0	0
( $x_i$ )	0	0	1	1	1	1	0	0
time $i$	0	1	2	3	4	5	6	7

<sup>12</sup>출처: <http://iamtrask.github.io/2015/11/15-anyone-can-code-lstm/>

## Multi-Layer RNN

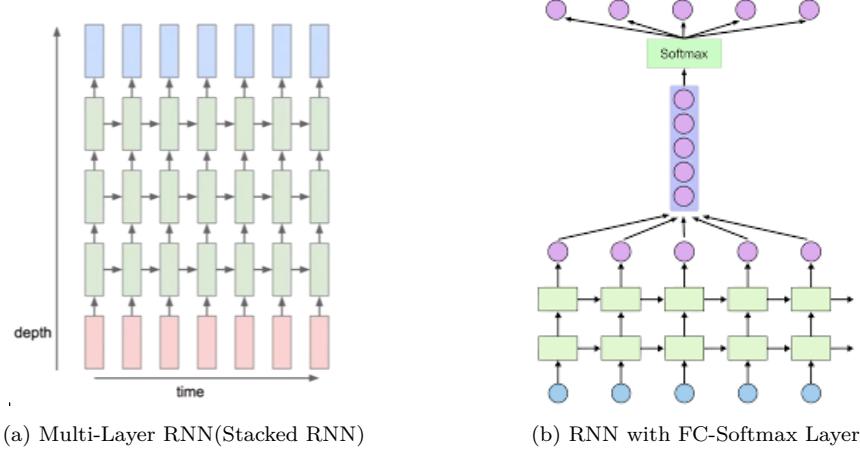


그림 22: Multi-RNN

- Cost Function으로  $L^2$ -norm을 사용하자.  $L^2$ -norm 대신에 Logistic Cost를 사용해도 됨.
- 2개의 숫자를 더하는 Model이므로, input( $x_i : 2 \times 1$ ), hidden layer( $h_i : N_h \times 1$ ), output( $y_i : 1 \times 1$ ).
- $W_{xh} : N_h \times 2, W_{hh} : N_h \times N_h, W_{hy} : 1 \times N_h$
- Forward Propagation: activation function  $\sigma$ 는 sigmoid함수를 사용한다.

$$\begin{aligned} z_i &= W_{xh}x_i + W_{hh}h_{i-1} \\ h_i &= \sigma(z_i) = \sigma(W_{xh}x_i + W_{hh}h_{i-1}) \leftarrow (N_h \times 1) \\ y_i &= \sigma(W_{hy}h_i) \leftarrow (1 \times 1) \end{aligned}$$

- Backward Propagation: 주어진 target data  $t_i$ 에 대하여,

$$\begin{aligned} dW_{hy} &= \sum_{i=0}^n ((t_i - y_i) \circ (y_i(1 - y_i))) h_i^T \leftarrow (1 \times 1)(1 \times N_h) \\ dW_{hh} &= \sum_{i=0}^n dz_i h_{i-1}^T, \text{ where } dz_i = \left( W_{hy}^T ((t_i - y_i) \circ (y_i(1 - y_i))) + W_{hh}^T dz_{i+1} \right) \circ (h_i(1 - h_i)) \\ dW_{xh} &= \sum_{i=0}^n dz_i x_i^T \end{aligned}$$

$dz_i$ 를 보면, recursive하게 구해지는데,  $W_{hh}^T$ 가 반복적으로 곱해지는 결과를 가져온다.  $W_{hh}^T$ 의 eigenvalue가 있다면,  $Ax = \lambda x, Ax = \lambda^2 x, \dots$  형태가 된다.  $\lambda > 1$ 이면 exploding gradient가 되고,  $\lambda < 1$ 이면 vanishing gradient 현상이 나타난다. 이런 문제를 해결하기 위해서는 gradient 값을 clip하여 잘라 주거나, vanilla RNN 대신 LSTM을 사용해야 한다.

- $L^2$ -norm 대신에 Logistic Cost Function을 사용한다면,

$$\begin{aligned} dW_{hy} &= \sum_{i=0}^n (t_i - y_i) h_i^T \leftarrow (1 \times 1)(1 \times N_{hy}) \\ dW_{hh} &= \sum_{i=0}^n dz_i h_{i-1}^T, \text{ where } dz_i = \left( W_{hy}^T (t_i - y_i) + W_{hh}^T dz_{i+1} \right) \circ (h_i(1 - h_i)) \end{aligned}$$

- $z_{n+1} = 0$ 이고,  $h_{-1} = 0$ 이므로  $i = 0$ 에서의  $dW_{hh}$  계산은 생략해도 됨.  $dz_i$ 의 값은  $dz_{i+1}$ 에서 구해지므로,  $dW_{hh}$ 는  $n, n-1, \dots, 1$  순으로 계산.
- Ⓡ RNN은 0 또는 1로 된 두 수와 넘어온 carry  $h_{i-1}$  모두 3개의 숫자를 결합(연산)하여  $y_i$ 와 carry로 넘겨줄 새로운  $h_i$ 를 구하는 과정을 구현한 것임. 따라서 7자리 2진수로 training한 RNN에 10자리 이진수를 적용해도 좋은 결과를 얻을 수 있다.

## 7.2 DENNY BRITZ Language Model<sup>13</sup>

```

I joined a new league this year and they have different scoring rules than I'm used to.
It's a slight PPR league- .2 PPR.
Standard besides 1 points for 15 yards receiving, .2 points per completion, 6 points per TD thrown,
My question is, is it wildly clear that QB has the highest potential for points?
...
x_irain[0]: SENTENCE_START i joined a new league this year and they have different
scoring rules than i 'm used to .
[0, 6, 3492, 7, 155, 792, 25, 223, 8, 33, 20, 202, 4952, 349, 91, 6, 66, 207, 5, 2]
y_irain[0]: i joined a new league this year and they have different scoring rules
than i 'm used to . SENTENCE_END
[6, 3492, 7, 155, 792, 25, 223, 8, 33, 20, 202, 4952, 349, 91, 6, 66, 207, 5, 2, 1]

x_irain[1]: SENTENCE_START it 's a slight ppr UNKNOWN_iOKEN UNKNOWN_iOKEN ppr .
[0, 11, 17, 7, 3095, 5974, 7999, 7999, 5974, 2]
y_irain[1]: it 's a slight ppr UNKNOWN_iOKEN UNKNOWN_iOKEN ppr . SENTENCE_END
[11, 17, 7, 3095, 5974, 7999, 7999, 5974, 2, 1]

```

- data 생성 : 입력 파일을 parsing하여 단어를 구하고, 빈도수 기준으로 상위  $N_x$  개만을 대상으로 한다. 그 외의 단어는 'UNKNOWN\_iOKEN'으로 대체한다. 문장 단위로 training data를 만든다. 각 문장의 앞에는 'SENTENCE\_START'를 넣는다. target data의 끝에는 'SENTENCE\_END'를 넣는다. 각각의 단어는 숫자에 mapping한다.
- 주어진 Example('reddit-comments-2015-08.txt')에서는 모든 단어 개수가 65517개 주어져 있고, 빈도순으로 상위  $N_x = 8,000$ 개를 선택했다. 그리고, 전체 training data 개수는 79,343개 이다.
- RNN에서는 batch size = 1을 사용한다. 하나의 data(문장 1개)가 length  $T$  일때,  $t = 0, \dots, T - 1$ . 이 모델에서는 data 각각의 문장 길이가 다르므로, training data마다  $T$ 가 달라 진다. 이렇게 batch data마다  $T$  가 달라지는 RNN을 Dynamic RNN이라 한다.
- 각 training data는 ('SENTENCE\_START', ..., 'SENTENCE\_END')와 같이 되어 있는데, input data는 마지막 단어인 'SENTENCE\_END'만 제외하고 만들고, output data는 시작 단어인 'SENTENCE\_START'만을 제외하여 만든다.
- $U : N_h \times N_x$ .  $N_h$ 는 hidden layer의 dimension.  $N_x$ 는 input data의 차원. 예를 들어, Language Model에서 사용하는 모든 단의의 갯수가 8,000개 이면,  $N_x = 8,000$ .  $U$ 의 각 column은 각 단어의 embedding weight라고 볼 수 있는데, Bengio Language Model에서의 embedding weight와 유사하다고 볼 수 있다.
- input  $x_i : N_x \times 1$ ,  $W : N_h \times N_h$ ,  $V : N_x \times N_h$ , output  $y_i : N_x \times 1$ .
- Model(Forward Propagation)

$$\begin{aligned}
h_i &= \tanh(Ux_i + Wh_{i-1}), \\
y_i &= \text{softmax}(Vh_i)
\end{aligned}$$

<sup>13</sup><http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-2-implementing-a-language-model-rnn-with-python-numpy/>

- training을 시키기 위해서는 training data를 하나(batch size=1) 씩 입력하여, SGD를 적용한다. loop를 통해 모든 data를 훈련시킨 후, 원하는 epoch만큼 반복한다.
- Backward Propagation: Cost Function<sup>o]</sup> cross-entropy error 일 때, 주어진 target data  $t_i$ 에 대하여

$$dV = \sum_{i=T-1}^0 (y_i - t_i) h_i^T$$

$dV$ 의 계산은 RNN Toy Model과 동일하나,  $dW$ 의 계산은 조금 변형되었다. 일반적으로 RNN에서 weight 행렬의 gradient를 back propagation을 통해서 구하는 과정을 BPTT(Back Propagation Through Time)이라고 하는데, 여기서는 Truncated-BPTT를 사용한다. Truncated-BPTT를 사용하는 이유는 계산량을 줄이려는 목적과 시간 간격이 멀어질 수록 영향력이 줄어드는 효과를 반영하려는 것이다.

- BPTT truncate 파라미터를  $K$ 라 하자(예를 들어,  $K = 4$ ).

$$\begin{aligned} dW &= \sum_{i=T-1}^0 \left[ z_i h_{i-1}^T + \sum_{j=i-1}^{i-K} b_{ij} h_{j-1}^T \right], \\ \text{where } z_i &= (V^T(y_i - t_i)) \circ (1 - h_i^2), \\ \text{and } b_{ij} &= \begin{cases} (W^T z_{j+1})(1 - h_j^2) & \text{if } j = i - 1, \\ (W^T b_{j+1})(1 - h_j^2) & \text{else: } j < i - 1 \end{cases} \\ dU &= \sum_{i=T-1}^0 z_i x_i^T + \sum_{j=i-1}^{i-K} b_{ij} x_j^T \end{aligned}$$

- $K$ 가 충분히 큰 경우는 RNN Toy Model과 동일한 결과를 가져온다.
- 문장의 생성: training 결과를 이용하여 문장을 생성하기 위해서는 먼저, 'SENTENCE\\_START' 1개로 된 문장을 입력하여 output으로 다음 단어의 확률을 구한다. 확률을 이용하여 random하게 다음 단어를 정한다. 이제 2개의 단어로 된 문장을 이용하여 다음 단어를 구한다. 이 작업을 마지막 단어가 'SENTENCE\\_END'가 나올 때까지 반복한다.
- RNN에서 gradient vanishing 현상은 Feed Forward Neural Network보다 잘 나타난다.
- gradient vanishing을 해결하는 방법은  $W$ 의 초기값을 잘 주는 방법, regularization을 사용하는 방법, sigmoid나 tanh 대신 RELU를 사용하는 방법, gradient clipping 등이 있다. 하지만, 이런 방법만으로는 완벽한 해결이 될 수 없어, Vanilla RNN은 많이 사용하지 않고, LSTM과 같은 모델을 사용하게 된다.
- 이보다 나은 방법으로 Long Short Term Memory(LSTM, 1997년), Gated Recurrent Unit(GRU, 2004년)이 있다. GRU는 LSTM의 간소화 버전이다.

### 7.3 Minimal Character-Level Vanilla RNN Model by Andrej Karpathy<sup>14</sup>

$$h_i = \tanh(W_{xh}x_i + W_{hh}h_{i-1} + b_h)$$

$$y_i = \text{softmax}(W_{hy}h_i + b_y)$$

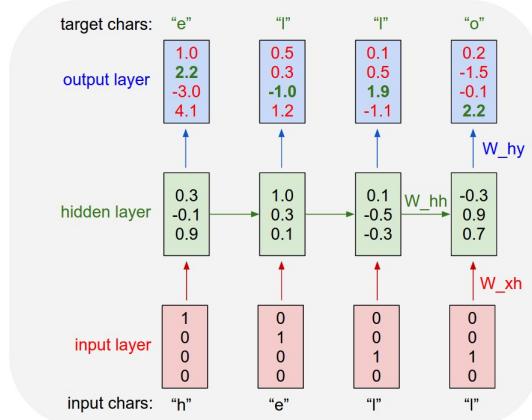


그림 23: An example RNN with 4-dimensional input and output layers, and a hidden layer of 3 units(source: Karpathy's Blog).

First Citizen:

Before we proceed any further, hear me speak.

All:

Speak, speak.

...

```
x_train[0]: 'First Citizen: Before we '
y_train[0]: 'irst Citizen: Before we p'
```

```
x_train[1]: 'proceed any further, hear'
y_train[1]: 'roceed any further, hear '
```

```
x_train[2]: ' me speak. All: Speak, s'
y_train[2]: 'me speak. All: Speak, sp'
```

- 예를 들어, Shakespeare의 작품(The Tragedy of Coriolanus)을 입력 data(size = 1,115,393)로 활용한다. 입력 data는 단어가 아닌 character 단위로 쪼개어 입력한다. 즉, 1,115,393 길이의 텍스트를 고정된 길이  $T = 24$  개씩 분할하여 training data를 만든다(이 data의 경우, character 종류는 모두  $N_x = 65$ ).
- 다시 말해, 이 Model은 training 단계에서는 RNN의 sequence 길이  $T (= 24)$ 를 고정 한다. 즉, 입력 data를  $T$  개씩 잘라 training data를 만들고, training을 수행한다.

<sup>14</sup><http://karpathy.github.io/2015/05/21/rnn-effectiveness/>

- training 후, 1개의 character를 입력하여 다음 character를 구한다 ( $T=1$ 인 RNN 모델에 적용). 이 작업을 원하는 만큼 (예를 들어, 200번) 반복한다. 각각의 작업이 완전히 독립적인 것은 아니다. 왜냐하면, Hidden State 값이 입력 값으로 전달되고, 다시 update되기 때문이다 (Recurrent Connection).
- hidden state의 dimension은  $N_h = 100$ 으로 잡으면,  $W_{hh} : N_h \times N_h$ ,  $W_{xh} : N_h \times N_x$ ,  $W_{hy} : N_x \times N_h$ 가 된다.
- bias  $b_h : N_h \times 1$ ,  $b_y : N_x \times 1$ .
- Forward Propagation: 이 모델에서 주목할 점이 하나 있는데, 각각의 길이  $T$ 인 training data를 완전히 독립적으로 보지 않고, 연결 고리를 가지고 있다. training data의 마지막 hidden state 값을 다음 training data의 hidden state의 시작 값으로 전달한다.
- Back Propagation: 주어진 target data  $t_i (N_x \times 1)$ 은 one-hot encoding 또는 그에 준하는 형태를 가지고 있다.

$$\begin{aligned}
dW_{hy} &= \sum_{i=T-1}^0 (y_i - t_i) h_i^T, \\
db_y &= \sum_{i=T-1}^0 (y_i - t_i), \\
dW_{hh} &= \sum_{i=T-1}^0 dz_i h_{i-1}^T, \text{ where } dz_i = \left( W_{hy}^T (t_i - y_i) + W_{hh}^T dz_{i+1} \right) \circ \left( 1 - h_i^2 \right), \\
dW_{xh} &= \sum_{i=T-1}^0 dz_i x_i^T, \\
db_h &= \sum_{i=T-1}^0 z_i
\end{aligned}$$

- Andrej Karpathy는 Paul Graham의 예세이, Shakespeare의 작품, LATEX 코드, Linux 소스 C 코드, 아기 이름을 Multi-Layer RNN이나 LSTM 모델을 적용하여 그 결과를 보여주고 있다.

## 7.4 LSTM(Long Short Term Memory) Networks<sup>15</sup>

- Long Short Term Memory by Hochreiter, Schmidhuber(1997)

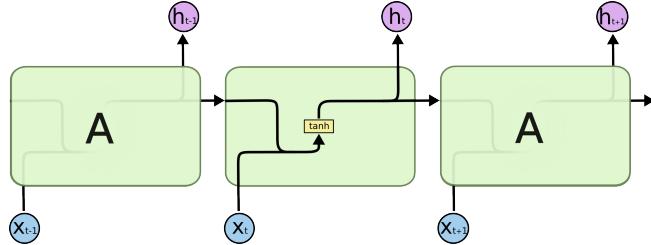


그림 24: The repeating module in a standard RNN contains a single layer(source: Colah's Blog).

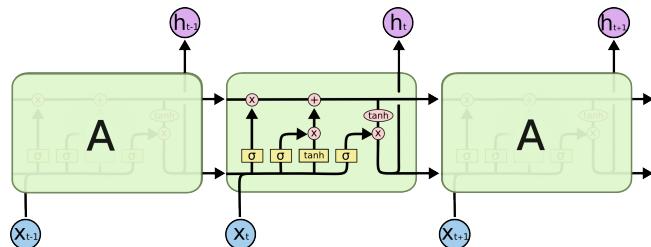


그림 25: The repeating module in an LSTM contains four interacting layers(source: Colah's Blog).

$$f_t = \sigma(W_{xh}^f x_t + W_{hh}^f h_{t-1} + b_h^f) \leftarrow \text{forget gate} \quad (2)$$

$$i_t = \sigma(W_{xh}^i x_t + W_{hh}^i h_{t-1} + b_h^i) \leftarrow \text{input gate} \quad (3)$$

$$o_t = \sigma(W_{xh}^o x_t + W_{hh}^o h_{t-1} + b_h^o) \leftarrow \text{output gate} \quad (4)$$

$$g_t = \tanh(W_{xh}^g x_t + W_{hh}^g h_{t-1} + b_h^g) \leftarrow \text{block input} \quad (5)$$

$$c_t = f_t \circ c_{t-1} + i_t \circ g_t \leftarrow \text{cell state} \quad (6)$$

$$h_t = o_t \circ \tanh(c_t) \leftarrow \text{hidden state} \quad (7)$$

$$y_t = \text{Softmax}(W_{hy} h_t + b_y) \leftarrow \text{output} \quad (8)$$

### ♠ LSTM Implementation

$x_t$	$N_x \times 1$
$W_{xh}^f, W_{xh}^i, W_{xh}^o, W_{xh}^g$	$N_h \times N_x$
$W_{hh}^f, W_{hh}^i, W_{hh}^o, W_{hh}^g$	$N_h \times N_h$
$b_h^f, b_h^i, b_h^o, b_h^g$	$N_h \times 1$
$f_t, i_t, o_t, g_t, c_t, h_t$	$N_h \times 1$
$W_{hy}$	$N_y \times N_h$
$b_y, y_t$	$N_y \times 1$

<sup>15</sup><http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

$W : 4N_h \times (N_x + N_h)$  와  $X_t : (N_x + N_h) \times 1$  를 다음과 같이 정의하자.

$$W := \begin{pmatrix} W_{xh}^f & | & W_{hh}^f \\ \hline \cdots & & \cdots \\ W_{xh}^i & | & W_{hh}^i \\ \hline \cdots & & \cdots \\ W_{xh}^o & | & W_{hh}^o \\ \hline \cdots & & \cdots \\ W_{xh}^g & | & W_{hh}^g \end{pmatrix}$$

$$X_t := \begin{pmatrix} x_t \\ \vdots \\ h_{t-1} \end{pmatrix}$$

이렇게  $W, X_t$  를 정의하면  $WX_t$  를 계산하여,  $f_t, i_t, o_t, g_t$  를 동시에 계산할 수 있다.

### ♠ Nico's Blog

Nico's Blog<sup>16</sup>의 모델은 output  $y_t$  를 따로 두지 않고,  $h_t = o_t \circ \tanh(c_t)$  로 하는 대신,

$$h_t = o_t \circ c_t \quad (9)$$

로 정의하고,  $h_t[0]$  을 output 으로 본다. target 값을  $TA(t)$  로 두고, Loss Function  $L$  을 다음과 같으  $L^2$ -norm 으로 두자

$$L = \sum_{t=0}^{T-1} l(t), \text{ where } l(t) = (h_t[0] - TA(t))^2$$

그러면,  $dl(t) = 2(h_t[0] - TA(t))$  가 된다. 아래의 Back Propagation 계산에서  $d\tilde{h}_t, d\tilde{c}_t$  는  $t+1$  에서의 gradient 계산과정에서 미리 계산된다.

$$\begin{aligned} dh_t &= dl(t) + d\tilde{h}_t \\ dc_t &= o_t \circ dh_t + d\tilde{c}_t \quad \text{by(9), (6)} \\ do_t &= c_t \circ dh_t \quad \text{by(9)} \\ di_t &= g_t \circ dc_t \quad \text{by(6)} \\ dg_t &= i_t \circ dc_t \quad \text{by(6)} \\ df_t &= c_{t-1} \circ dc_t \quad \text{by(6)} \end{aligned}$$

---

<sup>16</sup><http://nicodjimenez.github.io/2014/08/08/lstm.html>

o] 제 BPTT 을 통해  $W_{xh}^f, W_{xh}^i, W_{xh}^o, W_{xh}^g, W_{hh}^f, W_{hh}^i, W_{hh}^o, W_{hh}^g$  의 gradient 를 구해보자.

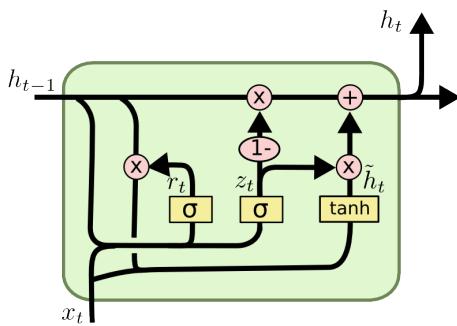
$$\begin{aligned} \left[ dW_{xh}^i \mid dW_{hh}^i \right] &= \sum_{t=T-1}^0 \left( i_t(1-i_t) \circ di_t \right) \left[ x_t^T \mid h_{t-1}^T \right] \leftarrow (N_h \times 1)(1 \times (N_x + N_h)) \quad \text{by(3)} \\ \left[ dW_{xh}^f \mid dW_{hh}^f \right] &= \sum_{t=T-1}^0 \left( f_t(1-f_t) \circ df_t \right) \left[ x_t^T \mid h_{t-1}^T \right] \quad \text{by(2)} \\ \left[ dW_{xh}^o \mid dW_{hh}^o \right] &= \sum_{t=T-1}^0 \left( o_t(1-o_t) \circ do_t \right) \left[ x_t^T \mid h_{t-1}^T \right] \quad \text{by(4)} \\ \left[ dW_{xh}^g \mid dW_{hh}^g \right] &= \sum_{t=T-1}^0 \left( (1-g_t^2) \circ dg_t \right) \left[ x_t^T \mid h_{t-1}^T \right] \quad \text{by(5)} \\ db_h^i &= \sum_{t=T-1}^0 \left( i_t(1-i_t) \circ di_t \right) \\ db_h^f &= \sum_{t=T-1}^0 \left( f_t(1-f_t) \circ df_t \right) \\ db_h^o &= \sum_{t=T-1}^0 \left( o_t(1-o_t) \circ do_t \right) \\ db_h^g &= \sum_{t=T-1}^0 \left( (1-g_t^2) \circ dg_t \right) \end{aligned}$$

BPTT 계산 과정에서 전 단계로 넘겨 주어야 할 gradient 의 계산은 다음과 같다.

$$\begin{aligned} d\tilde{c}_{t-1} &= dc_t \circ f_t \\ d\tilde{h}_{t-1} &= (W_{hh}^i)^T \left( i_t(1-i_t) \circ di_t \right) + (W_{hh}^f)^T \left( f_t(1-f_t) \circ df_t \right) \\ &\quad + (W_{hh}^o)^T \left( o_t(1-o_t) \circ do_t \right) + (W_{hh}^g)^T \left( (1-g_t^2) \circ dg_t \right) \end{aligned}$$

단,  $\tilde{c}_{T-1} = 0, \tilde{h}_{T-1} = 0$ .

## ♠ GRU(Gated Recurrent Unit) Networks



$$\begin{aligned}
 z_t &= \sigma(W_z \cdot [h_{t-1}, x_t]) \\
 r_t &= \sigma(W_r \cdot [h_{t-1}, x_t]) \\
 \tilde{h}_t &= \tanh(W \cdot [r_t * h_{t-1}, x_t]) \\
 h_t &= (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t
 \end{aligned}$$

그림 26: GRU(source: Colah's Blog).

## 7.5 RNN with TensorFlow

- dynamic\_rnn의 return 값 outputs의 마지막 값은 states와 같은 값이다. 이 outputs를 받아, FC layer, Softmax layer 등을 더 붙힐 수 있다.

```
cell = tf.contrib.rnn.BasicRNNCell(num_units=hidden_size) # num_units = output dim = hidden dim
outputs, states = tf.nn.dynamic_rnn(cell, x_data, dtype=tf.float32) # x_data: batch_size x
seq_length x input dim, outputs: batch_size x seq_length x output dim
```

```
# One hot encoding for each char in 'hello'
h = [1, 0, 0, 0]; e = [0, 1, 0, 0]
l = [0, 0, 1, 0]; o = [0, 0, 0, 1]

with tf.variable_scope('3_batches') as scope:
    scope.reuse_variables()

    x_data = np.array([[h, e, l, l, o],
                      [e, o, l, l, l],
                      [l, l, e, e, l]], dtype=np.float32)
    batch_size = len(x_data)
    hidden_size = 2
    cell = rnn.BasicRNNCell(num_units=hidden_size)
    #cell = rnn.BasicLSTMCell(num_units=hidden_size, state_is_tuple=True)

    initial_state = cell.zero_state(batch_size, tf.float32)
    outputs, _states = tf.nn.dynamic_rnn(cell, x_data, initial_state=initial_state,
                                         dtype=tf.float32)

    sess = tf.Session()
    sess.run(tf.global_variables_initializer())
    pp pprint(sess.run(outputs))
    pp pprint(sess.run(_states))

    sess.close()
```

---

output의 마지막 값들을 모아 놓은 것이 state값이 된다는 것을 확인할 수 있다.

```

output:
array([[[ -0.4359245 ,  0.13481933], [-0.07057824, -0.11087653],[-0.62125361, -0.39745134],
       [-0.6342904 , -0.69737196],[ 0.65984339, -0.82264704]],

      [[ 0.19345769, -0.02967541],[ 0.57927972, -0.31469607],[-0.31637764, -0.2922264 ],
       [-0.60422462, -0.57655394],[-0.54986298, -0.75639403]],

      [[-0.65008968, -0.29962495],[-0.67916012, -0.66499412],[ 0.36651206, -0.66683197],
       [ 0.66517806, -0.35485405],[-0.25961858, -0.28670925]]], dtype=float32)

state:
array([[ 0.65984339, -0.82264704],[-0.54986298, -0.75639403],
       [-0.25961858, -0.28670925]], dtype=float32)

```

- BasicRNNCell, BasicLSTMCell, GRUCell, LSTMCell은 서로 대등.
- BasicLSTMCell의 state\_is\_tuple: If True, accepted and returned states are 2-tuples of the c\_state and m\_state. If False, they are concatenated along the column axis.

---

```

cell = rnn.BasicLSTMCell(num_units=hidden_size, state_is_tuple=True) # states =(c_state ,
m_state)

initial_state = cell.zero_state(batch_size, tf.float32)
outputs, states = tf.nn.dynamic_rnn(cell,x_data,initial_state=initial_state, dtype=tf.float32) #
states[0] = c_state = cell state, states[1] = m_state = hidden state = end value of outputs

```

---

output의 마지막 값들을 모아 놓은 것이 state값이 된다는 것을 확인할 수 있다.

```

array([[[ 0.02563098,  0.02743041],[ 0.04659975,  0.08679023],[ 0.10533179,  0.17980342],
       [ 0.14791322,  0.25055808],[ 0.13946146,  0.10292473]],

      [[ 0.02489312,  0.06242029],[ 0.06387331, -0.0016607 ],[ 0.1307383 ,  0.12094586],
       [ 0.16763097,  0.21373361],[ 0.18991804,  0.28005067]],

      [[ 0.08519164,  0.1144949 ],[ 0.13702379,  0.2043138 ],
       [ 0.14732455,  0.18357307],[ 0.12587772,  0.17263964],
       [ 0.14331181,  0.24550749]]], dtype=float32)

LSTMStateTuple
(c=array([[ 0.35456109,  0.2470004 ],[ 0.46305579,  0.50542879],[ 0.33805162,  0.44125667]], dtype=float32),
 h=array([[ 0.13946146,  0.10292473],[ 0.18991804,  0.28005067],[ 0.14331181,  0.24550749]], dtype=float32))

```

- MultiRNNCell

---

```

pp = pprint.PrettyPrinter(indent=4)
sess = tf.InteractiveSession()

batch_size=3
seq_length=5
input_dim=3

```

```

hidden_dim=4
num_layers = 7

x_data = np.arange(45, dtype=np.float32).reshape(batch_size, seq_length, input_dim)

with tf.variable_scope('MultiRNNCell') as scope:
    # Make rnn
    cells = []
    for _ in range(num_layers):
        cell = rnn.BasicLSTMCell(num_units=hidden_dim, state_is_tuple=True)
        cells.append(cell)
    cell = tf.contrib.rnn.MultiRNNCell(cells, state_is_tuple=True)
    # rnn in/out
    outputs, _states = tf.nn.dynamic_rnn(cell, x_data, dtype=tf.float32)
    print("dynamic rnn: ", outputs)
    sess.run(tf.global_variables_initializer())
    pp pprint(outputs.eval()) # batch size, unrolling (time), hidden_size

```

---

위 코드에서 `len(_states) = num_layers`. 각 `_states[i]`는 c,h를 가지고 있는데, c,h의 shape은 `batch x hidden_dim` 된다. `_states[-1]`의 h값은 output의 마지막 값을 모아 놓은 것과 같다.

- Loss: `sequence_loss`의 logits 입력은 activation function을 거치지 않은 것을 넣어주어야 한다. `dynamic_rnn`의 outputs은 activation function을 거친 값. `sequence_loss`는 outputs에 softmax 함수를 취한 후, cross-entropy loss mean을 계산해 준다.

```

sequence_loss = tf.contrib.seq2seq.sequence_loss(logits=outputs, targets=Y, weights=weights) #
    targets: not one-hot, but label, [batch_size,seq_length], weights = [1,...,1]
loss = tf.reduce_mean(sequence_loss) # mean all sequence loss
train = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(loss)
#####

```

---

- `sequence_length`: batch data의 길이가 다른 경우.

```
outputs, states = tf.nn.dynamic_rnn(cell, x_data, sequence_length=[5,3,4], dtype=tf.float32)
```

---

- word embedding:

```

tf.reset_default_graph()
x_data = np.array([[0, 3, 1, 2, 4],[1, 3, 1, 2, 3],[2, 4, 0, 2, 4]], dtype=np.int32) #
    (batch_size,seq_length)

input_dim = 5; embedding_dim = 6;
init = np.arange(input_dim*embedding_dim).reshape(input_dim,-1) # sample initialization

sess = tf.InteractiveSession()
with tf.variable_scope('test', reuse=tf.AUTO_REUSE) as scope:
    embedding = tf.get_variable("embedding", initializer=init) # shape=(input_dim, embedding_dim)
    inputs = tf.nn.embedding_lookup(embedding, x_data) # shape=(batch_size, seq_length,
        embedding_dim)

```

```

sess.run(tf.global_variables_initializer())
print(sess.run(embedding))
print(sess.run(inputs))

```

```

[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]
 [24 25 26 27 28 29]]

[[[ 0  1  2  3  4  5]
 [18 19 20 21 22 23]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [24 25 26 27 28 29]]

[[ 6  7  8  9 10 11]
 [18 19 20 21 22 23]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]

[[12 13 14 15 16 17]
 [24 25 26 27 28 29]
 [ 0  1  2  3  4  5]
 [12 13 14 15 16 17]
 [24 25 26 27 28 29]]]

```

## 7.6 Character-Level Vanilla RNN Model by Andrej Karpathy

Karpathy의 Minimal Character Model을 Multi-Layer RNN으로 확장해보자.

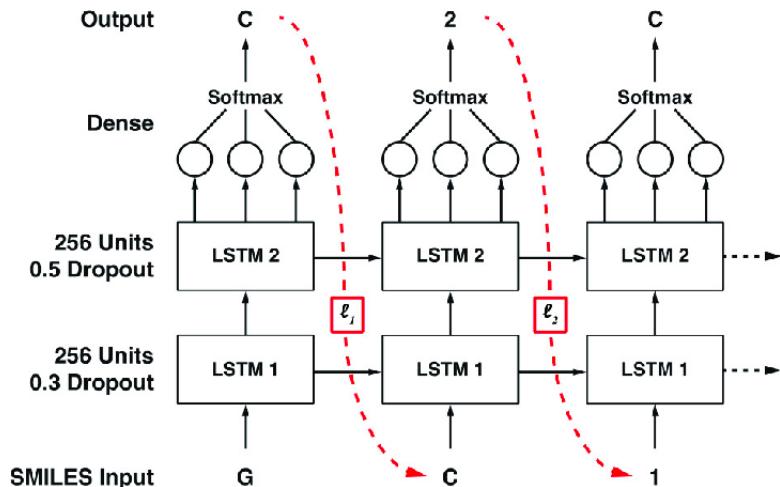


그림 27: Multi-Layer RNN

- Karpathy의 Lua 코드<sup>17</sup>를 Tensorflow 형태로 구현한 코드<sup>18</sup>를 중심으로 살펴보자.
- 각 character를 embedding vector로 변환하여, Multi-Layer RNN에 넣는 구조로 되어 있다.
- Multi-Hidden-Layer를 거치고 나온 결과에 FC layer를 적용 후, softmax를 통해 최종 output을 얻는다.
- training할 때는 sequence-length가 1보다 길게 설정한다. training 후, 결과를 저장하고, sampling 할 때는 sequence-length = 1인 모델을 만들어 반복한다.

## 7.7 Image Captioning with RNNs(CS231n)

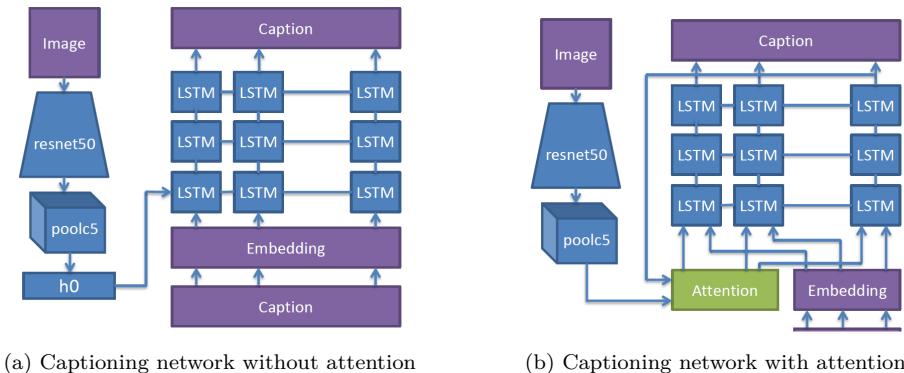


그림 28: Captioning RNN Models

- MS COCO<sup>19</sup>: cs231n assignment3에서는 image feature(train-82,783개, val-40,504개)와 image caption(train-400,135개, val-195,954개)가 주어진다. image feature보다 caption이 더 많은데, 이는 하나의 image에 평균 5개 정도의 caption이 주어져 있기 때문이다<sup>20</sup>.
- 그림 (28a)의  $h_0$ 는 이미지의 feature( $F_0$ )에 weight를 곱한 것이다. 자세히 설명하면, 먼저 VGG16의 FC7 layer 결과인 dimension 4,096 vector를 PCA를 통해 dimension 512으로 줄인 후( $F$ ), weight  $W_p$ 를 곱하고 bias  $b_p$ 를 더해 구해졌다.

$$h_0 = FW_p + b_p$$

- initial hidden state  $h_0$ 를 이렇게 구현하는 것은 강의 slide의 모델과는 표현상의 차이가 있다. 강의 slide에는 다음과 같이 되어 있다.

$$h_1 = \tanh(W_{xh}X + W_{hh}h_0 + W_{ih}V)$$

하지만, 여기서  $h_0$ 를 0으로 넣기 때문에, 본질적으로는 동일한 모델이다.

- Capiton vector  $C(N \times T)$ 은 word embedding을 통해  $X(N \times T \times V)$ 로 변환되어 RNN의 input이 된다. 여기서  $N$ 은 batch size,  $T$ 는 sequence length,  $V$ 는 word dimension.

---

```
# word embedding forward propagation
W = 0.01*np.random.randn(V,D) # V: word dim, D: embedding dim
X = W[C, :] # C: N x T
```

<sup>17</sup><https://github.com/karpathy/char-rnn>

<sup>18</sup><https://github.com/sherjilozair/char-rnn-tensorflow>

<sup>19</sup>[mscoco.org](http://mscoco.org)

<sup>20</sup>caption number 111236,111487,111555,111591,111849는 하나의 image인 13585를 설명하고 있다.



그림 29: Caption samples for image 13945

232365: <START> a man and woman standing next to each other on snow covered ground <END>  
232744: <START> young couple on the ski <UNK> preparing to ski <END>  
232902: <START> two people takes a photo while holding snowboard and wearing skis <END>  
232912: <START> two people with ski and snowboard gear on a slope <END>  
232946: <START> a couple is standing in the snow in an <UNK> <END>

---

```
# word embedding back propagation
dW = np.zeros_like(W) # gradient of W
np.add.at(dW,C,dX) # for given dX
```

---

- output은 hidden layer  $h_{out}(N \times T \times H)$ 에 Affine Transformation을 적용하여  $N \times T \times V$ 로 변환된다. 그 후, 마지막으로 softmax를 적용하면 된다.
- 그런데, 이 모델을 training 시키면 그 결과가 좋지는 않다. page 32에서 언급한 것처럼, gradient vanishing, exploding 현상 때문이다(Vanilla RNNs can be tough to train on long sequences due to vanishing and exploding gradients caused by repeated matrix multiplication. LSTMs solve this problem by replacing the simple update rule of the vanilla RNN with a gating mechanism as follows.)
- 그런데, LSTM으로 모델을 구성했을 때, BLEU score는 0.3을 넘기 어렵다.

## 7.8 Neural Machine Translation

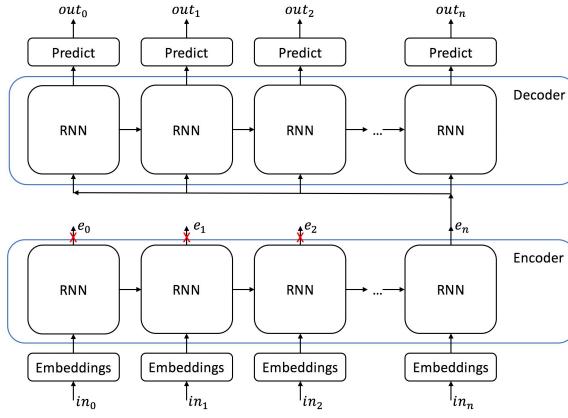


그림 30: Encoder-Attention-Decoder

## 8 Reinforcement Learning



그림 31: RL 대표 Research Group

'모두를 위한 RL강좌'<sup>21</sup>를 기반으로 RL의 기초를 다져보자.

### 8.1 Q-Table, Q-Network

- 준비작업 : OpenAI Gym에서 제공하는 game환경을 설정한다. msys2, xming, atari설치, gym[atari] 등을 설치해야 한다<sup>22</sup>.
- FrozenLake를 keyboard 입력을 받아 한번 해본 후, random으로 움직였을 때의 결과를 확인해 본다. 성공 확률이 1.25% 정도 나온다.
- (Dummy)Q-Learning Algorithm을 FrozenLake에 적용해 보면, 성공 확률이 85% ~ 95% 정도 나온다.

$$Q(s, a) = r + \max_{a'} Q(s', a')$$

- 다음 단계로 (Dummy)Q-Learning을 개선하여 Q-Learning Method를 완성해 보자. Exploit & Exploration, Discount Rate  $\gamma$ 를 적용하기만 하면 된다. 성공 확률은 약간 더 좋아진다.

---

```

if method == 1: # decaying E-greedy
    if np.random.rand(1) < e:
        action = env.action_space.sample()
    else:
        action = np.argmax(Q[state, :])
elif method == 2: # add random noise
    action = np.argmax(Q[state, :] + np.random.randn(1,env.action_space.n) / (i + 1))

```

---

<sup>21</sup><https://www.youtube.com/playlist?list=PL1MkM4tgefjnKsCWav-Z2F-MMFRx-2gMGG>

<sup>22</sup><http://freeablelab.tistory.com/126> <http://ishuca.tistory.com/390>

action  $a$ 가 결정되고 나면, Discount Rate  $\gamma$ 를 적용하면 된다.

$$Q(s, a) = r + \gamma \max_{a'} Q(s', a')$$

5. ○] 제 Deterministic 환경을 Stochastic 환경으로 변경하면, 성공 확률이 1 ~ 2%로 낮아진다. Stochastic 환경이기 때문에,  $s$ 에서 action  $a$ 를 취했을 때, 얻게 되는 state  $s'$ 는 변하기 때문에, Q-Learning Algorithm을 수정해야 한다.

6. Learning Rate  $\alpha$ , Discount Rate  $\gamma$ 를 도입하여, 기존 Q값과 새로운 Q값을 혼합한다.

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a')]$$

○] 제 Learning Rate을 적용한 성공 확률은 50 ~ 60%로 개선 된다.

7. 다음 단계로 좀 더 복잡한 문제를 풀기 위하여 Q-Network를 도입하자. 현실적인 문제는 어마어마한 크기의 Q-Table을 만들어야 하기 때문에 Q-Table 방식을 적용할 수는 없다.

## Q-function Approximation

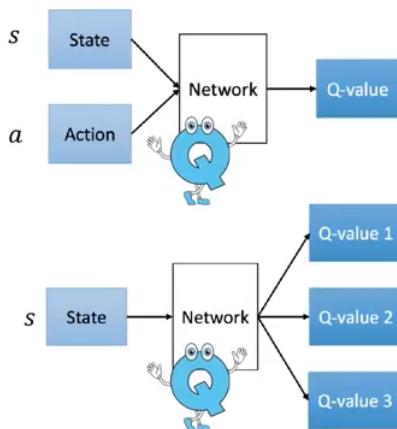


그림 32: Q Network

8. Q-Network의 Target은 Q-Value로 설정한다.

---

```

batch_size = 1. initial state = s
while()
    Get Q = FP(s) <-- FP: Forward Propagation, dim of Q = out_dim
    Get best next action a from Q.
    s', reward, done, _ = env.step(a)
    Get Q_ = FP(s')
    Update Q(s) from Q_, Q at only s
    Update Network(Weight) via Back Propagation from target Q
    s <- s'

```

---

9. ○] Q-Network를 실행해 보면, 잘 되지는 않는데 (성공 확률 50 ~ 60%), 그 이유는 다음과 같다.

- Correlations between samples
- Non-Stationary targets

# Algorithm

**Algorithm 1** Deep Q-learning

---

```

Initialize action-value function  $Q$  with random weights
for episode = 1,  $M$  do
    Initialise sequence  $s_1 = \{x_1\}$  and preprocessed sequenced  $\phi_1 = \phi(s_1)$ 
    for  $t = 1, T$  do
        With probability  $\epsilon$  select a random action  $a_t$ 
        otherwise select  $a_t = \max_a Q^*(\phi(s_t), a; \theta)$ 
        Execute action  $a_t$  in emulator and observe reward  $r_t$  and image  $x_{t+1}$ 
        Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $\phi_{t+1} = \phi(s_{t+1})$ 

        Set  $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$ 
        Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  according to equation 3
    end for
end for

```

---

Playing Atari with Deep Reinforcement Learning - University of Toronto by V Mnih et al.

그림 33: Q Netrowk

## 8.2 DQN

Deep Q-Network 알고리즘은 2013년, 2015년 DeepMind에서 발표했는데, 핵심 아이디어는 다음과 같다.

- Go Deep(2013)
- Capture and replay: Correlations between samples 해결 (NIPS 2013)
- Separate Networks: Create a target network. Non-stationary targets 문제 해결 (Nature 2015)

---

**Algorithm 1** Deep Q-learning with Experience Replay

```

Initialize replay memory  $D$  to capacity  $N$ 
Initialize action-value function  $Q$  with random weights
for episode = 1,  $M$  do
    Initialise sequence  $s_1 = \{x_1\}$  and preprocessed sequenced  $\phi_1 = \phi(s_1)$ 
    for  $t = 1, T$  do
        With probability  $\epsilon$  select a random action  $a_t$ 
        otherwise select  $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$ 
        Execute action  $a_t$  in emulator and observe reward  $r_t$  and image  $x_{t+1}$ 
        Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $\phi_{t+1} = \phi(s_{t+1})$ 
        Store transition  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $D$ 
        Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $D$ 
        Set  $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$ 
        Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  according to equation 3
    end for
end for

```

---

(a) DQN, NIPS 2013

**Algorithm 1: deep Q-learning with experience replay.**

```

Initialize replay memory  $D$  to capacity  $N$ 
Initialize action-value function  $Q$  with random weights  $\theta$ 
Initialize target action-value function  $\hat{Q}$  with weights  $\theta^- = \theta$ 
For episode = 1,  $M$  do
    Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequence  $\phi_1 = \phi(s_1)$ 
    For  $t = 1, T$  do
        With probability  $\epsilon$  select a random action  $a_t$ 
        otherwise select  $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$ 
        Execute action  $a_t$  in emulator and observe reward  $r_t$  and image  $x_{t+1}$ 
        Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $\phi_{t+1} = \phi(s_{t+1})$ 
        Store transition  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $D$ 
        Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $D$ 
        Set  $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$ 
        Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  with respect to the
        network parameters  $\theta$ 
        Every  $C$  steps reset  $Q = Q$ 
    End For
End For

```

(b) DQN, Nature 2015

그림 34: DQN Algorithms

## ♠ DQN, NIPS 2015

매 state마다 (Main)Network에 back propagation을 적용하여 weight를 update하지 않고, buffer에 계속 쌓는다. 어느 정도 쌓이면, 쌓여있는 buffer에서 random하게 mini batch size만큼 sampling을 하여 back propagation을 수행한다. 이 때, target value는 Main Network에서 얻어 오지 않고, Target Network에서 가져온다. 그리고, 주기적으로 Target Network의 weight를 Main Network의 값으로 복사해 온다. 지금까지의 과정은 하나의 episode 내에서 수행되며, episode가 끝나면 처음부터 반복한다.

---

```
batch_size = N. initial state = s
while()
    Get Q = FP(s) <-- FP: Forward Propagation from main_net, dim of Q = out_dim
    Get best next action a from Q.
    s', reward, done, _ = env.step(a)
    add_buffer(s,a,reward,s',done)
    if len(buffer) > batch_size
        sample minibatch from buffer
        Get Q_ = FP(s') from target_net <-- key point of DQN 2015
        Update Q(s) from Q_, Q at only s
        Update main_net(Weight) via Back Propagation from target Q
        Update target_net from main_net every m steps
    s <-- s'
```

---

### 8.3 Policy Gradient

## 9 Generative Adversarial Networks

Generative Adversarial Networks(GAN)은 2014년 I. Goodfellow<sup>23</sup>등이 제안하면서 발전하기 시작한 Unsupervised Learning 모델이다. A. Ng, Y. Lecun등은 미래의 딥러닝 기술의 키는 Unsupervised Learning이라고 말하고 있다.

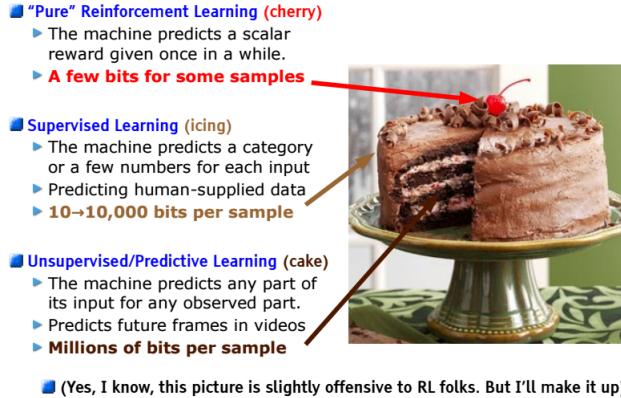


그림 35: How Much Information Does the Machine Need to Predict? (Y. Lecun, NIPS 2016)

### 9.1 Variational Auto Encoder

- Entropy: 여러가지 해석이 가능하지만, 여기서는 정보량을 기대값으로 해석해보자. 확률이 낮은 사건이 발생하면 더 많은 정보를 얻을 수 있다고 볼 수 있다. 그래서 확률  $p_i$ 의 사건이 발생했을 때, 정보의 양을  $-\log p_i$ 로 정의하면 확률  $P$ 에 대한 정보량의 기대값 Entropy는

$$H(P) = - \sum_i p_i \log p_i$$

- Cross Entropy: 확률  $Q$ 의 의한 정보량을 확률  $P$ 에 대한 기대값을 Cross Entropy라 한다.

$$H(P, Q) = - \sum_i p_i \log q_i$$

- Kullback–Leibler divergence

$$\begin{aligned} D_{KL}(P||Q) &= (\text{Cross Entropy}) - (\text{Entropy}) \\ &= H(P, Q) - H(P) \\ &= - \sum_i p_i \log q_i + \sum_i p_i \log p_i \end{aligned}$$

예를 들어,  $p(x) = N(\mu_1, \sigma_1)$ ,  $p(x) = N(\mu_2, \sigma_2)$  ]면,

$$D_{KL}(P||Q) = \log \frac{\sigma_2}{\sigma_1} + \frac{\sigma_1^2 + (\mu_1 - \mu_2)^2}{2\sigma_2^2} - \frac{1}{2}$$

### 9.2 Vanilla GAN

Value Function

$$V(D, G) = \min_G \max_D \left[ \mathbb{E}_{x \sim p_{data}(x)} \log D(x) + \mathbb{E}_{z \sim p_z(z)} \log (1 - D(G(z))) \right]$$

<sup>23</sup><https://arxiv.org/abs/1406.2661>

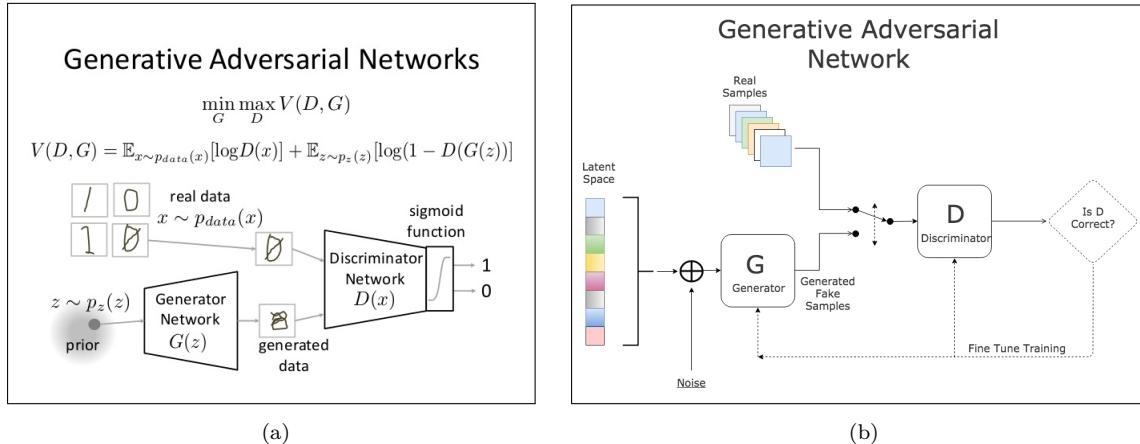


그림 36: GAN Model

$$\text{Loss}_D = - \left[ \log D(x) + \log(1 - D(G(z))) \right] \quad \text{or} \quad - \left[ \log D(x) - \log(D(G(z))) \right]$$

$$\text{Loss}_G = -\log D(G(z))$$

$\text{Loss}_D$ 를 줄이기 위해 Discriminator의 weight update와  $\text{Loss}_G$ 를 줄이기 위한 Generator의 weight update가 동시에 각각 이루어 진다. 다시말해, Discriminator는 정상적인 이미지는 잘 판단하고 Generator가 만든 이미지는 가짜로 판별하기 위해 training을 하는 것이고 Generator는 Discriminator를 속이기 위한 이미지를 잘 만들기 위해 training 한다.

## Vanilla Gan Discriminator

```
input_data(image) X(N,784)
-> fc1 (784, 256) (256,) -> relu
-> fc2 (256, 1) (1,) -> sigmoid
```

## Vanilla Gan Generator:

```
latent(random) Z(N,128)
-> fc1 (128, 256) (256,) -> relu
-> fc2 (256, 784) (784,) -> sigmoid(tanh)
```

### 9.3 DCGAN

DCGAN은 2016년 1월에 발표된 논문 Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks - Alec Radford et al., 2016<sup>24</sup>]다.

## ◆ Data 준비

- celebA Dataset<sup>25</sup>: 156,253개의 얼굴이미지 파일을 얻을 수 있다. 사진 크기는  $128 \times 128$ .
  - lsun Dataset<sup>26</sup>: 10 scene categories, 20 object categories. All the images in one category are stored in one lmdb database file. The value of each entry is the jpg binary data. We resize all the images so that the smaller dimension is 256 and compress the images in jpeg with quality 75.

<sup>24</sup><https://arxiv.org/abs/1511.06434>

<sup>25</sup>[https://www.dropbox.com/s/e0ig4nf1v94hyj8/CelebA\\_128crop\\_FD.zip?dl=0](https://www.dropbox.com/s/e0ig4nf1v94hyj8/CelebA_128crop_FD.zip?dl=0)

<http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html>

<sup>26</sup><https://github.com/fyu/lsun>

## ♠ Transposed-Convolution(Deconvolutiton)

Deconvolution에 대한 설명은 이곳<sup>27</sup>을 참조.

### Illustration of Convolution Operations

- Convolutional filters in CNN and transposed-convolutional filters in DCGAN works in the opposite directions. Here's a good illustration how they work.

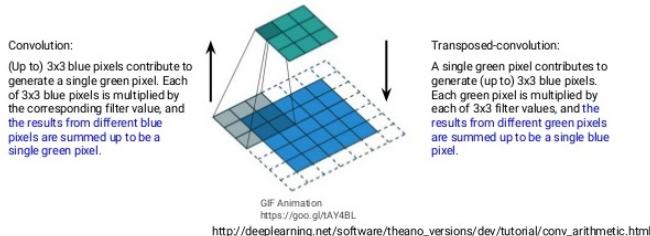


그림 37: Convolution & Transposed-Convolution

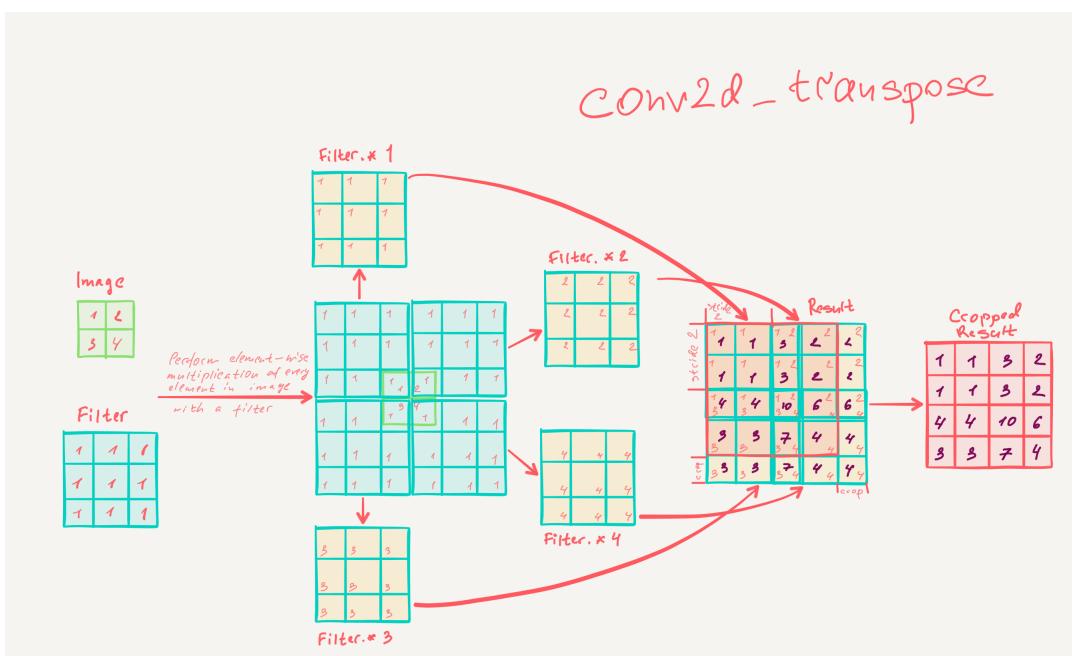


그림 38: Deconvolution 계산 과정

```
# input batch shape = (1, 2, 2, 1) -> (batch_size,height,width,channels) - 2x2x1 image in batch of 1
x = tf.constant(np.array([[[[1], [2]], [[3], [4]]]]), tf.float32)

# shape = (3, 3, 1, 1) -> (height,width,input_channels,output_channels) - 3x3x1 filter
f = tf.constant(np.ones([3,3,1,1]), tf.float32)

conv = tf.nn.conv2d_transpose(x,f,output_shape=(1, 4, 4, 1),strides=[1, 2, 2, 1],padding='SAME')

with tf.Session() as session:
    result = session.run(conv)

assert (np.array([[[[1.0], [1.0], [3.0], [2.0]], [[1.0], [1.0], [3.0], [2.0]]]),
```

<sup>27</sup><https://datascience.stackexchange.com/questions/6107/what-are-deconvolutional-layers>

---

```
[[4.0], [4.0], [10.0], [6.0]], [[3.0], [3.0], [7.0], [4.0]]]) == result).all()
```

---

`tf.nn.conv2d_transpose`는 `output_shape`에 따라 padding이 조절된다.

### ♠ DCGAN-CelebA 구조

DCGAN에 대한 설명과 여러가지 구현에 대한 것은 Brandon Amos의 Blog<sup>28</sup>를 참조하면 된다.

먼저 살펴볼 구현<sup>29</sup>은 논문 저자가 구현한 DCGAN<sup>30</sup> 구조와 약간 차이가 있다. discriminator convolution filter 가 ( $4 \times 4 \times 3 \times 128$ )로 되어 있는데, 저자의 구현에서는 ( $5 \times 5 \times 3 \times 64$ )로 되어 있다. 그리고 이 구조의 DCGAN model은 발산하기도 한다.

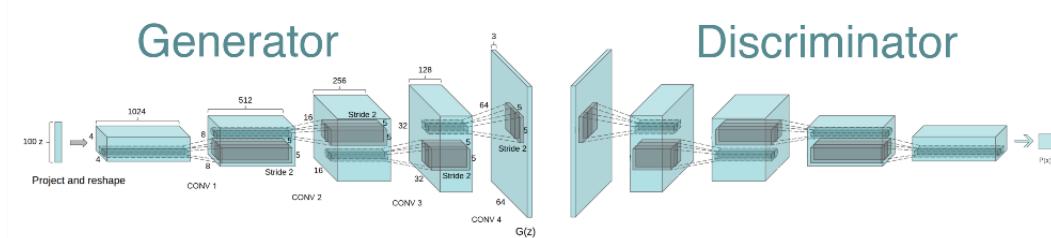


그림 39: DCGAN architecture

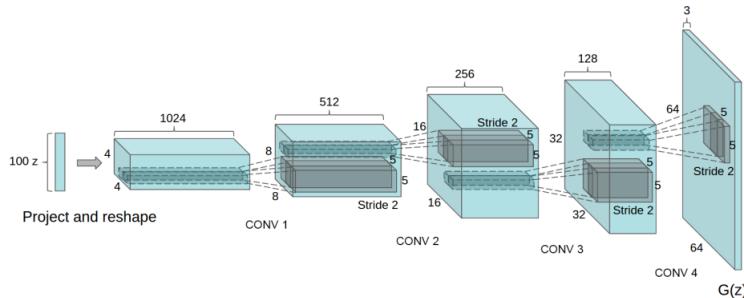


그림 40: DCGAN Generator used for LSUN scene modeling

All Tensors are 4D (include Batch\_size Dimension)

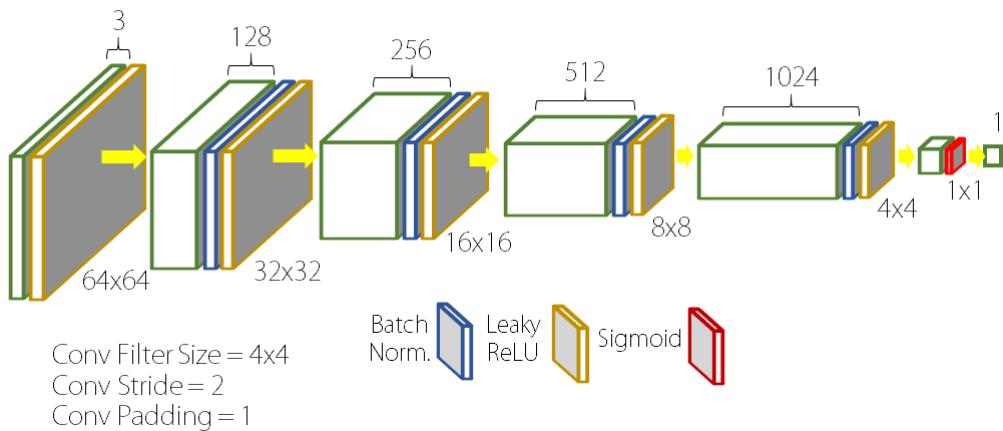


그림 41: DCGAN Discriminator

<sup>28</sup><http://bamos.github.io/2016/08/09/deep-completion/>

<sup>29</sup>[https://github.com/HyeongminLEE/Tensorflow\\_DCGAN](https://github.com/HyeongminLEE/Tensorflow_DCGAN)

<sup>30</sup>[https://github.com/Newmu/dcgan\\_code](https://github.com/Newmu/dcgan_code)

Discriminator를 training할 때는 image data와 latent(random noise)가 필요하고, Generator를 training할 때는 latent만 필요하다. Discriminator와 Generator에 넘기는 latent vector는 각각 생성한다(논문 저자의 구현에서는 같은 것을 사용했다). generator 시작 시점에서 latent vector를 변환할 때, 저자의 구현에서는 FC를 사용한 반면, 이 구현은 deconvolution을 사용한다.

DCGAN Discriminator	Input/Output	Filter/Weight FH, FW, in-channels, out-channels	
Convolution 1 & leaky-Relu	$N \times H \times W \times C$ $N \times 64 \times 64 \times 3$	$FH \times FW \times C \times FN$ $4 \times 4 \times 3 \times 128$	celebA image( $128 \times 128$ )는 $64 \times 64$ 로 resize. Input의 4번째 size와 Weight의 3번째 size는 일치. Weight의 4번째 size는 Output의 4번째 size와 일치한다. stride = 2. TensorFlow(tf.nn.conv2d)에서 padding='SAME'을 적용하면 Output size 가 Input size/stride가 되는데, 이는 padding size = 1에 해당한다.
Convolution 2, BN, leaky-Relu	$N \times 32 \times 32 \times 128$	$4 \times 4 \times 128 \times 256$	stride = 2
Convolution 3, BN, leaky-Relu	$N \times 16 \times 16 \times 256$	$4 \times 4 \times 256 \times 512$	stride = 2
Convolution 4, BN, leaky-Relu	$N \times 8 \times 8 \times 512$	$4 \times 4 \times 512 \times 1024$	stride = 2
Convolution 5 & leaky-Relu	$N \times 4 \times 4 \times 1024$	$4 \times 4 \times 1024 \times 1$	stride = 4
Sigmoid	$N \times 1 \times 1 \times 1$		reshape( $N \times 1$ )
Prob	$N \times 1$		

DCGAN Generator	Input/Output	Filter/Weight (FH, FW, out-channels, in-channels) out-shape	
reshape	$N \times 100$		reshape( $N \times 1 \times 1 \times 100$ )
deconvolution 1, BN, Relu	$N \times 1 \times 1 \times 100$	$4 \times 4 \times 1024 \times 100$ $N \times 4 \times 4 \times 1024$	stride = 4
deconvolution 2, BN, Relu	$N \times 4 \times 4 \times 1024$	$4 \times 4 \times 512 \times 1024$ $N \times 8 \times 8 \times 512$	stride = 2
deconvolution 3, BN, Relu	$N \times 8 \times 8 \times 512$	$4 \times 4 \times 256 \times 512$ $N \times 16 \times 16 \times 256$	stride = 2
deconvolution 4, BN, Relu	$N \times 16 \times 16 \times 256$	$4 \times 4 \times 128 \times 256$ $N \times 32 \times 32 \times 128$	stride = 2
deconvolution 5	$N \times 32 \times 32 \times 128$	$4 \times 4 \times 3 \times 128$ $N \times 64 \times 64 \times 3$	stride = 2
Tanh	$N \times 64 \times 64 \times 3$		
Output	$N \times 64 \times 64 \times 3$		

### ♠ 계산 결과

Hardware	GPU-GALAX GeForce GTX 970(Memory 4G)
Data	CelebA
mini batch size = 200, 3 epoch	3,725초
mini batch size = 300, 1 epoch	1,194초

### ♠ DCGAN-CelebA 구조2

논문 저자의 DCGAN 구현 구조는 다음과 같다. celebA image( $128 \times 128$ )를 ( $108 \times 108$ ) 크기로 자른 후, 다시 ( $64 \times 64$ )로 resize했다. 그림(41)에서 generator는 latent vector를 ( $N \times 4 \times 4 \times 1024$ )로 변환한 걸로 나오는데, CelebA를 대상으로 구현할 때는 ( $N \times 4 \times 4 \times 512$ )로 했다.

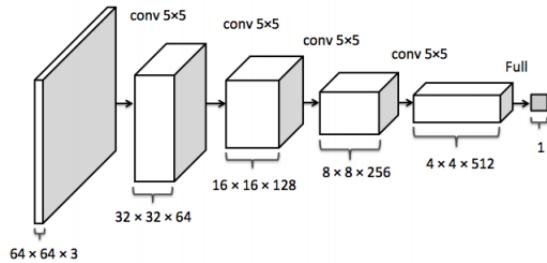


그림 42: DCGAN discriminator architecture

DCGAN Discriminator	Input/Output	Filter/Weight FH, FW, in-channels, out-channels	
Convolution 1 & leaky-Relu	$N \times H \times W \times C$ $N \times 64 \times 64 \times 3$	$FH \times FW \times C \times FN$ $5 \times 5 \times 3 \times 64$	celebA image( $128 \times 128$ )는 $64 \times 64$ 로 resize. Input의 4번째 size와 Weight의 3번째 size는 일치. Weight의 4번째 size는 Output의 4번째 size와 일치한다. stride = 2. TensorFlow(tf.nn.conv2d)에서 padding='SAME'을 적용하면 Output size 가 Input size/stride가 되는데, 이는 padding size = 1에 해당한다.
Convolution 2, BN, leaky-Relu	$N \times 32 \times 32 \times 64$	$5 \times 5 \times 64 \times 128$	stride = 2
Convolution 3, BN, leaky-Relu	$N \times 16 \times 16 \times 128$	$5 \times 5 \times 128 \times 256$	stride = 2
Convolution 4, BN, leaky-Relu	$N \times 8 \times 8 \times 256$	$5 \times 5 \times 256 \times 512$	stride = 2
Reshape	$N \times 4 \times 4 \times 512$		$4 \times 4 \times 512 = 8192$
FC, Sigmoid	$N \times 8192$	$8192 \times 1$	
Prob	$N \times 1$		

DCGAN Generator	Input/Output	Filter/Weight (FH, FW, out-channels, in-channels) out-shape	
FC	$N \times 100$	$100 \times 8192$	
reshape	$N \times 8192$		reshape( $N \times 4 \times 4 \times 512$ )
BN, Relu	$N \times 4 \times 4 \times 512$		
deconvolution, BN, Relu	$N \times 4 \times 4 \times 512$	$5 \times 5 \times 256 \times 512$ $N \times 8 \times 8 \times 256$	stride = 2
deconvolution, BN, Relu	$N \times 8 \times 8 \times 256$	$5 \times 5 \times 128 \times 256$ $N \times 16 \times 16 \times 128$	stride = 2
deconvolution, BN, Relu	$N \times 16 \times 16 \times 128$	$5 \times 5 \times 64 \times 128$ $N \times 32 \times 32 \times 64$	stride = 2
deconvolution	$N \times 32 \times 32 \times 64$	$5 \times 5 \times 3 \times 64$ $N \times 64 \times 64 \times 3$	stride = 2
Tanh	$N \times 64 \times 64 \times 3$		
Output	$N \times 64 \times 64 \times 3$		

### ♠ DCGAN-MNIST 구조

latent vector  $z$ 의 dimension을 100이라고 하자. MNIST data의 class 수는 10이므로,  $y$ -dim=10이 된다. DCGAN 논문 저자의 Theano 구현 코드를 보면,  $(x, y, z)$ 가 쌍으로 입력되어, discriminator, generator에 사용된다. discriminator에서는  $(x, y)$ 가 concatenate되어 입력되고, generator에서는  $(z, y)$ 가 concatenate되어 처리된다. discriminator 입장에서는 true/false만 판별하면 되지만, 입력되는 숫자 이미지가 어떤 숫자인지 구체적으로 알고 있다는 의미이다. 다시 말해, discriminator에게 어떤 숫자 이미지인지 알려주는 training을 한다는 것이다. generator에게도 label  $y$ 가 중간 중간 concat되는데, discriminator가 label을 알고 있기 때문에, label에 맞는 숫자를 생성해야 한다. 예를 들어, label이 3인데, 아무리 정교하게 2을 만들어도 discriminator를 속일 수 없다. 3의 이미지를 만들어야 한다.

예 : random latent vector에 다음과 같은 label을 concat시키면, label에 맞는 숫자가 생성되는 걸 볼 수 있다.

75103252 68364083 28621201 44024423 22894003 20653052 03796123 95533708

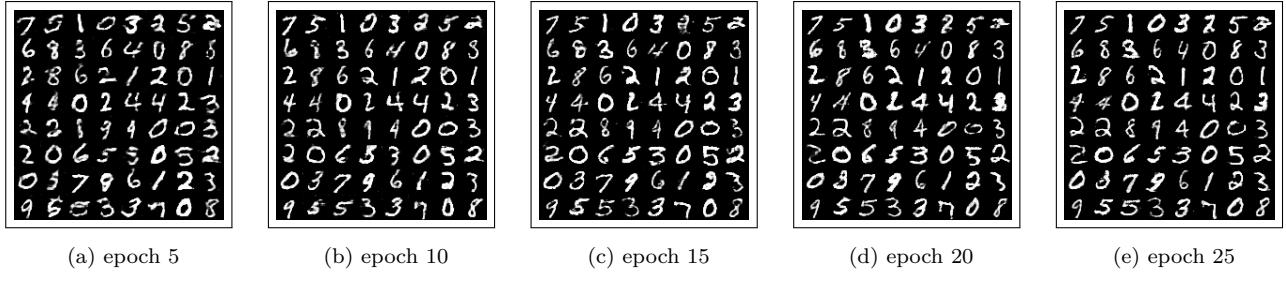


그림 43: Generated MNIST Images

DCGAN Discriminator	Input/Output	Filter/Weight FH, FW, in-channels, out-channels	
concat(image,label)	$(N \times 28 \times 28 \times 1), (N \times 1 \times 1 \times 10)$		concat([image,y],axis=3)
Convolution & leaky-Relu	$N \times 28 \times 28 \times 11$	$5 \times 5 \times 11 \times 11$	stride = 2
concat label	$N \times 14 \times 14 \times 11$		
Convolution, BN, leaky-Relu	$N \times 14 \times 14 \times 21$	$5 \times 5 \times 21 \times 74$	stride = 2
reshape	$N \times 7 \times 7 \times 74$		$N \times 3626, 3626 = 7 \times 7 \times 74$
concat label	$N \times 3626$		
FC,BN,leaky-Relu	$N \times 3636$	$3636 \times 1024$	
concat label	$N \times 1024$		
FC, Sigmoid	$N \times 1034$	$1034 \times 1$	
Prob	$N \times 1$		

DCGAN Generator	Input/Output	Filter/Weight (FH, FW, out-channels, in-channels) out-shape	
concat(z,y)	$N \times 100, N \times 10$		concat([z,y],axis=1) z-dim=100,y-dim=10
FC,BN,Relu	$N \times 110$	$110 \times 1024$	
concat label	$N \times 1024$		
FC,BN,Relu	$N \times 1034$	$1034 \times 6272$	$6272 = 7 \times 7 \times 128$
reshape	$N \times 6272$		$N \times 7 \times 7 \times 128$
concat label	$N \times 7 \times 7 \times 128$		
deconvolution,BN,Relu	$N \times 7 \times 7 \times 138$	$5 \times 5 \times 128 \times 138$ $N \times 14 \times 14 \times 128$	stride = 2
concat label	$N \times 14 \times 14 \times 128$		
deconvolution,Sigmoid	$N \times 14 \times 14 \times 138$	$5 \times 5 \times 3 \times 138$ $N \times 28 \times 28 \times 1$	stride = 2
Output	$N \times 28 \times 28 \times 1$		

## 9.4 Texture Synthesis(2015)

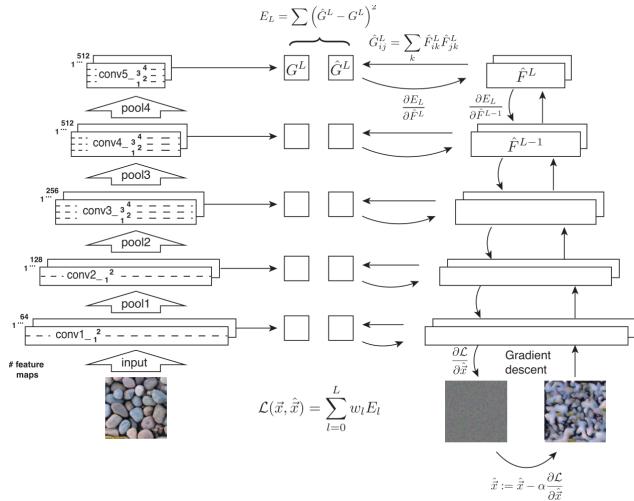


그림 44: Texture synthesis method

- Gatys, Leon A., Alexander S. Ecker, and Matthias Bethge. Texture Synthesis Using Convolutional Neural Networks, 2015<sup>31</sup>
  - VGG19 network에서 MaxPooling을 AvgPooling으로 바꾸어 사용한다.
  - Loss  $L$ 은 texture loss( $L_t$ ), norm loss( $L_n$ ), total variance loss( $L_{tv}$ ) 3개의 합으로 이루워진다. 논문에서는 texture loss만 언급되어 있다.
  - synthesis image  $X$ , texture  $T$ ,에 대하여 Loss  $L$ 은 각 Loss들의 weighted sum이다.

$$L = w_t L_t(X, T) + w_n L_n(X) + w_{tv} L_{tv}(X)$$

- texture loss  $L_t$ 는 식(11)과 동일하게 Gram matrix로 변환하여 계산한다.
  - norm loss는  $X$ 의 모든 원소들의 제곱 합.
  - total variance loss는 total variance denosing이라고도 하는데, 식(12)에서 정의되어 있다.

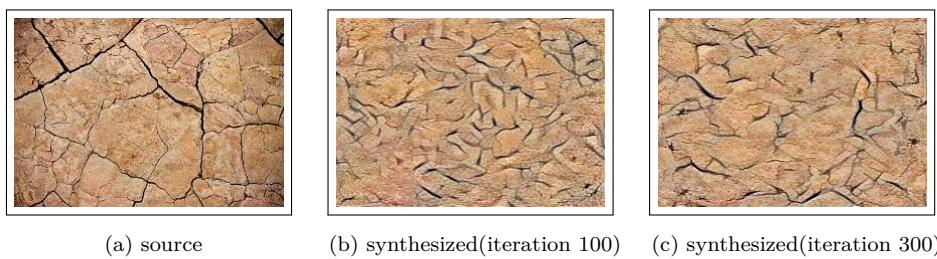


그림 45: Examples of synthesized texture

<sup>31</sup><https://arxiv.org/abs/1505.07376>

## 9.5 Neural Style(2017)

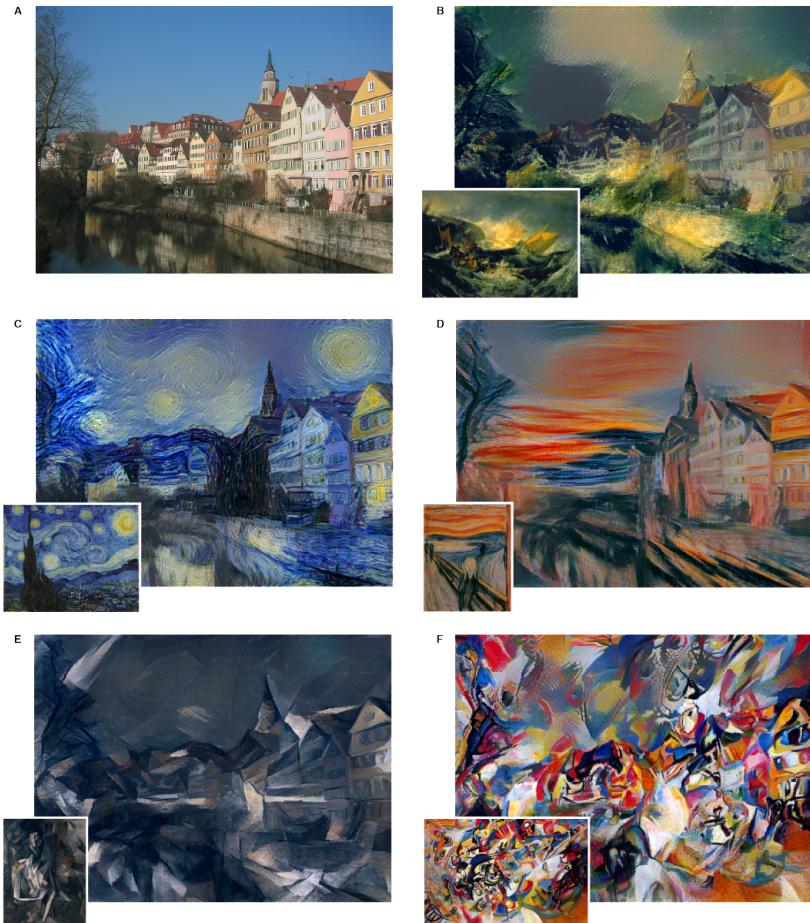


그림 46: Images that combine the content of a photograph with the style of several well-known artworks. The images were created by finding an image that simultaneously matches the content representation of the photograph and the style representation of the artwork (see Methods). The original photograph depicting the Neckarfront in Tübingen, Germany, is shown in **A** (Photo: Andreas Praefcke). The painting that provided the style for the respective generated image is shown in the bottom left corner of each panel. **B** *The Shipwreck of the Minotaur* by J.M.W. Turner, 1805. **C** *The Starry Night* by Vincent van Gogh, 1889. **D** *Der Schrei* by Edvard Munch, 1893. **E** *Femme nue assise* by Pablo Picasso, 1910. **F** *Composition VII* by Wassily Kandinsky, 1913.

- 2015년 9월 발표된 논문 A Neural Algorithm of Artistic Style-Leon A. Gatys, Alexander S. Ecker, Matthias Bethge<sup>32</sup>이 발표된다.
- 논문이 나오고 얼마 지나지 않아 jcjohnson(Justin Johnson)이 Github에 ‘neural style’이라는 porch 코드를 공개하여 논문의 방법대로 구현이 가능하다는 것을 보여준다.
- Tensorflow로 구현된 코드(Github, anishathalye-neural style, 2015년 12월) 또한 공개되었고, 이후 여러가지 변형된 버전이 계속 올라오고 있다(lengstrom, cysmith 등).
- 이 논문이 나오게된 배경을 살펴보자. CNN에서 생성되는 feature map으로부터 image를 복원하거나 변형하는 연구들이 계속되고 있었다.

<sup>32</sup><https://arxiv.org/abs/1508.06576>

- Mahendran, Aravindh, and Andrea Vedaldi. Understanding deep image representations by inverting them, 2014<sup>33</sup>. –> Feature Map으로 부터 이미지 복원. Inverting of an image representation is formulated as the problem of finding an image whose representation best matches the one given. Formally, given a representation function  $\Phi : \mathbb{R}^{H \times W \times C} \rightarrow \mathbb{R}^d$  and a representation  $\Phi_0 = \Phi(\mathbf{x}_0)$  to be inverted, reconstruction finds the image  $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$  that minimizes the objective:

$$\mathbf{x}^* = \underset{\mathbf{x} \in \mathbb{R}^{H \times W \times C}}{\operatorname{argmin}} \ell(\Phi(\mathbf{x}), \Phi_0) + \lambda \mathcal{R}(\mathbf{x})$$

where the loss  $\ell$  compares the image representation  $\Phi(\mathbf{x})$  to the target one  $\Phi_0$  and  $\mathcal{R} : \mathbb{R}^{H \times W \times C} \rightarrow \mathbb{R}$  is a regulariser capturing a *natural image prior*.

- Gatys, Leon A., Alexander S. Ecker, and Matthias Bethge. Texture Synthesis Using Convolutional Neural Networks, 2015<sup>34</sup>.

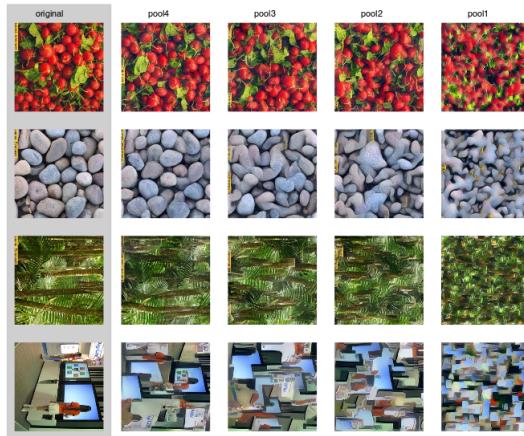


그림 47: Texture synthesis

- Nguyen, Anh, Jason Yosinski, and Jeff Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images, 2014<sup>35</sup>.

- 이제 이 논문에서 제시하는 방법을 구체적으로 살펴보자. 그림(49)에서 왼쪽 input이 style image( $S$ )이고 오른쪽이 content image( $C$ )이다. 그리고 가운데가 2개의 image를 결합해서 만들어야 할 합성 이미지( $X$ , synthesis image)가 된다. 이  $X$ 는 optimization을 통해서 구해야할 대상이 된다.
- $S, C, X$ 는 각각 VGG19 network으로 입력되어 중간(각각 STYLE-LAYERS, CONTENT-LAYERS)의 feature map 들 중 일부를 뽑아내게 된다. style과 content는 각각 서로 다른 feature map을 뽑아 낸다.
- 먼저, content loss를 구해보자.  $C, X$ 를 VGG19 network으로 입력하여 layer( $\ell \in \text{CONTENT-LAYERS=CL}$ )의 feature map을 뽑아 낸다.  $C$ 에서 뽑아낸 feature map을  $P^\ell$ ,  $X$ 에서 뽑아낸 feature map을  $F^\ell$ 로 표시하자. 뽑아낸  $C, X$ 의 feature map의  $L_2$ -norm을 계산하여 합한다. 합쳐진 값( $w_c^\ell$ -weighted sum)의 content loss  $L_c$ 이다. 단,  $L_2$ -norm은 2차원 행렬이 아니라, 4차원 array를 대상으로 한 것이다.

$$L_c = \sum_{\ell \in \text{CL}} w_c^\ell \|P^\ell - F^\ell\|^2 \times \lambda_c^\ell \quad (10)$$

<sup>33</sup><https://arxiv.org/abs/1412.0035>

<sup>34</sup><https://arxiv.org/abs/1505.07376>

<sup>35</sup><https://arxiv.org/abs/1412.1897>

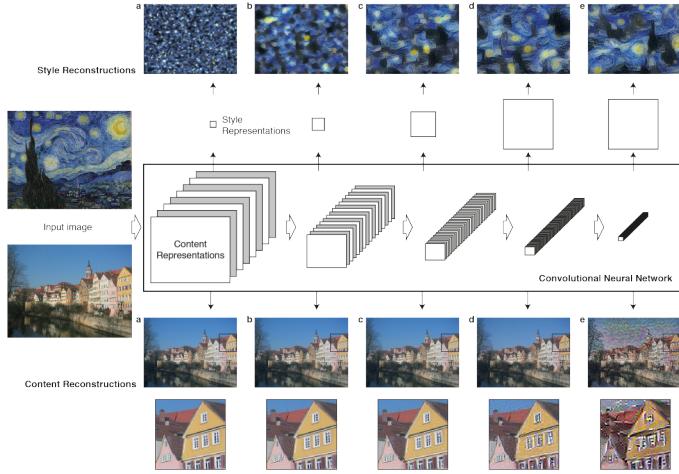


그림 48: **Convolutional Neural Network (CNN)**. A given input image is represented as a set of filtered images at each processing stage in the CNN. While the number of different filters increases along the processing hierarchy, the size of the filtered images is reduced by some downsampling mechanism (e.g. max-pooling) leading to a decrease in the total number of units per layer of the network. **Content Reconstructions.** We can visualise the information at different processing stages in the CNN by reconstructing the input image from only knowing the network's responses in a particular layer. We reconstruct the input image from layers 'conv1\_1' (a), 'conv2\_1' (b), 'conv3\_1' (c), 'conv4\_1' (d) and 'conv5\_1' (e) of the original VGG-Network. We find that reconstruction from lower layers is almost perfect (a,b,c). In higher layers of the network, detailed pixel information is lost while the high-level content of the image is preserved (d,e). **Style Reconstructions.** On top of the original CNN representations we built a new feature space that captures the style of an input image. The style representation computes correlations between the different features in different layers of the CNN. We reconstruct the style of the input image from style representations built on different subsets of CNN layers ('conv1\_1' (a), 'conv1\_1' and 'conv2\_1' (b), 'conv1\_1', 'conv2\_1' and 'conv3\_1' (c), 'conv1\_1', 'conv2\_1', 'conv3\_1' and 'conv4\_1' (d), 'conv1\_1', 'conv2\_1', 'conv3\_1', 'conv4\_1' and 'conv5\_1' (e)). This creates images that match the style of a given image on an increasing scale while discarding information of the global arrangement of the scene.

$\lambda_c^\ell$  같은 논문에서는  $\frac{1}{2}$  을 제시하고 있지만, 여러가지 구현에서는 다른 값을 사용하기도 한다.

$$\lambda_c^\ell = \begin{cases} \frac{1}{h_\ell \times w_\ell \times c_\ell} & \text{where } P^\ell\text{-shape} = (1, h_\ell, w_\ell, c_\ell) \text{ or} \\ \frac{1}{2\sqrt{h_\ell \times w_\ell \times c_\ell}} \end{cases}$$

그리고, 논문에서는 CONTENT-LAYERS로 'conv4-2' 하나 만을 제시하고 있다.

- ]제 style loss  $L_s$  를 구해보자  $S, X$  를 VGG19 network으로 입력하여 일부 layer( $\ell \in \text{STYLE-LAYERS}=SL$ ) 의 feature map을 구한다.  $S$  의 featrue map을  $A^\ell$ ,  $X$  에서 뽑아낸 feature map을  $G^\ell$ 로 표시하자. Gram matrix로 변환된 행렬은  $\text{Gram}(A^\ell), \text{Gram}(G^\ell)$ 로 표시하자. STYLE-LAYERS의 모든 layer에 대하여, Gram matrix 차의 Frobenius norm을  $w_s^\ell$ -weighted sum한 것이  $L_s$  가 된다.

$$L_s = \sum_{\ell \in SL} w_s^\ell \| \text{Gram}(A^\ell) - \text{Gram}(G^\ell) \|^2 \times \lambda_s^\ell. \quad (11)$$

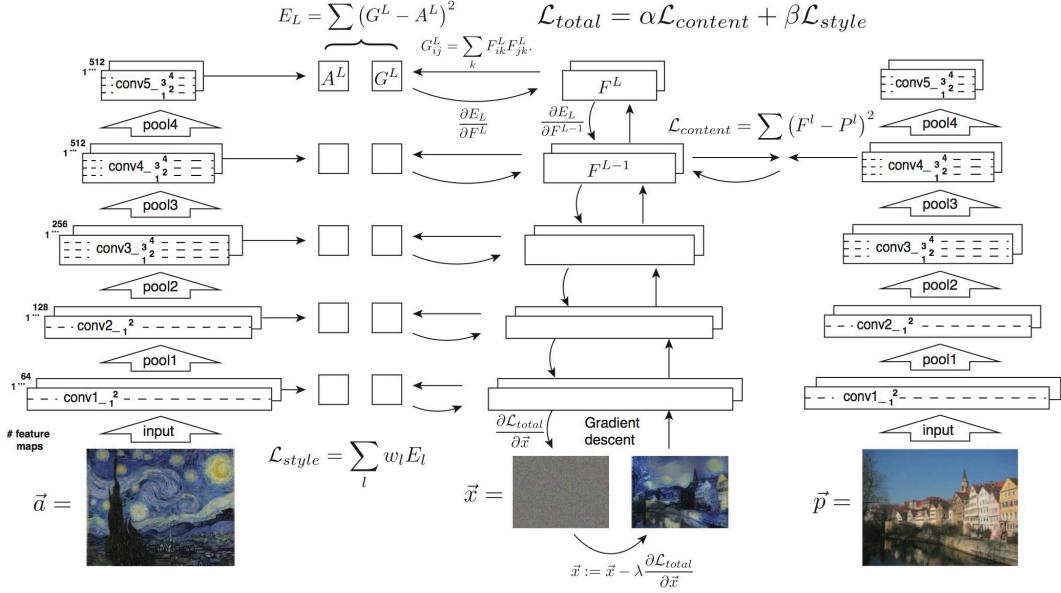


그림 49: Style Transfer Structure

$\lambda_s^\ell$ 는 layer(feature map)  $\ell$ 의 가로, 세로 곱을  $M_\ell$  필터 수를  $N_\ell$  일 때,

$$\lambda_s^\ell = \frac{1}{4(M_\ell \times N_\ell)^2}$$

그리고, 논문에서는 STYLE-LAYERS로 'conv1-1', 'conv2-1', 'conv3-1', 'conv4-1', 'conv5-1'를 대표적으로 제시하고 있다. 하지만 처음 구현을 공개한 jcjohnson(Justin Johnson)를 비롯한 여러 구현된 예를 보면, 'relu1-1', 'relu2-1', 'relu3-1', 'relu4-1', 'relu5-1'를 사용하고 있다.

- 논문에서는 제시되지 않았지만, total variance denoising을 Loss로 추가한 구현도 있다. synthesis image  $X = (x_{i,j})$ 에 대하여,

$$L_t = \sum_{i,j} (x_{i+1,j} - x_{i,j})^2 + (x_{i,j+1} - x_{i,j})^2 \quad (12)$$

- $L_c, L_s, L_t$ 에 weight  $\alpha, \beta, \gamma$ 를 곱하여 합하면 전체적인 Loss가 된다.

$$L = \alpha L_c + \beta L_s + \gamma L_t$$

- 앞으로 돌아가, style loss 계산 과정에 나오는 Gram Matrix에 대해 알아보자. feature map을  $G$ 의 shape  $\circ$   $(1, h, w, c)$ 라고 하면, reshape하여  $\bar{G} = (hw, c)$ 로 변환한다.

$$Gram(G) = \bar{G}^T \bar{G} \leftarrow c \times c$$

Gram matrix는 정확하게 covariance matrix와는 다르지만, feature map(filter)들 간의 correlation으로 해석 한다.

## 9.6 Deep Photo Style Transfer

Fujun Luan, Sylvain Paris, Eli Shechtman, Kavita Bala, Deep Photo Style Transfer, 2017<sup>36</sup>

- 논문 저자들은 Lua로 작성된 코드를 공개했다<sup>37</sup>.
- Tensorflow로 구현된 코드<sup>38</sup>를 중심으로 살펴보자. 이 구현에선 VGG모델로 image를 입력하기 전에 resize하지 않았다. FC layer를 통과시키지 않기 때문에 resize하지 않아도 된다.
- Loss는 모두 4가지로 이루어져 있다.

---

```
Loss = loss_content + loss_style + loss_tv + loss_affine
```

---

- content loss는 식(10)와 동일하다.

$$L_c = \sum_{\ell \in CL} w_c^\ell \|P^\ell - F^\ell\|^2$$

논문에서는 CONTENT-LAYERS를 따로 두지 않고, 모든 layer에 weight를 곱하는 방식으로 설명하고 있다. weight를 0으로 주어, 일부를 뽑아 낼 수 있기 때문에 아주 일반화하여 표현한 것이다. 구현 설명 부분에서는 conv4\_2에만 weight 1을 부여하고, 나머지에는 0을 주고 있음. 즉 CONTENT-LAYERS = {conv4\_2}. 여기서의 conv4\_2은 relu를 포함하고 있다.

- Augmented style loss with semantic segmentation: Style loss는 식(11)와는 다른 방식을 제시하고 있다. style 변환을 위해서 content image, style image 외에 content segmentation, style segmentation을 같이 입력받는다. 식(11)에서처럼 style image와 output image의 feature map을 gram matrix로 변환하여 loss를 구하는 것이 아니다. segmentation image를 mask해서 구한 masked image와 곱한 후, gram matrix를 구한다. 다시 말해, style image의 feature map( $S_j$ )은 masked-style segmentation image( $S_{ij}^s$ )와 곱한 후 gram matrix를 구한다. output image(X)의 feature map( $X_j$ )은 masked-content segmentation image( $S_{ij}^c$ )와 곱한 후 gram matrix를 구한다. segmentation은 다음의 9가지로 구성되어 있다.

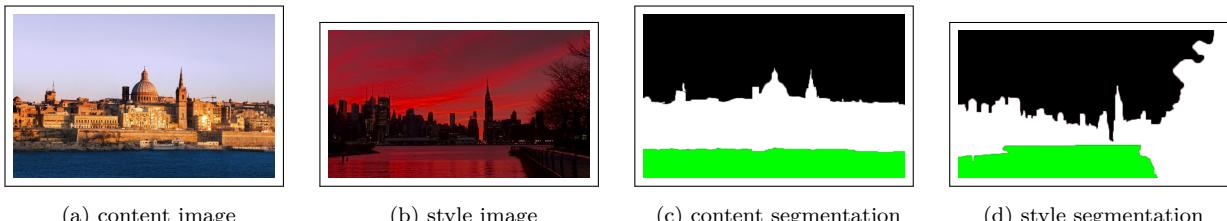


그림 50: Input Images

[ 'BLUE', 'GREEN', 'BLACK', 'WHITE', 'RED', 'YELLOW', 'GREY', 'LIGHT\_BLUE', 'PURPLE' ]

또한, STYLE-LAYERS = {conv1\_1, conv2\_1, conv3\_1, conv4\_1, conv5\_1}로 주어져 있다.

$$\text{style loss } L_s = \sum_{j=1}^5 \sum_{i=1}^9 \|Gram(X_j S_{ij}^c) - Gram(S_j S_{ij}^s)\|^2$$

여기서  $X_j, S_{ij}^c$ 의 곱,  $S_j, S_{ij}^s$ 의 곱은 broadcasting이 적용된다.

<sup>36</sup><https://arxiv.org/abs/1703.07511>

<sup>37</sup><https://github.com/luanfujun/deep-photo-styletransfer>

<sup>38</sup><https://github.com/LouieYang/deep-photo-styletransfer-tf>



그림 51: Deep Photo Style Transfer Result

Style Image	$1 \times 700 \times 530 \times 3$	Style Seg[i] or Content Seg[i]	$1 \times 700 \times 530 \times 1$	
conv1-1(relu 포함)	$1 \times 700 \times 530 \times 64$	avg-pool	$1 \times 700 \times 530 \times 1$	pooling 전에 size=1 padding 후, $3 \times 3$ , stride=1로 average pooling. Style Image의 output layer 를 $S_1$ , Style Segmentation $i$ 의 output layer를 $S_{i1}^s$ , Content Segmentation $i$ 의 output layer 를 $S_{i1}^c$
conv1-2	$1 \times 700 \times 530 \times 64$			
pool	$1 \times 350 \times 265 \times 64$	resize	$1 \times 350 \times 265 \times 1$	
conv2-1	$1 \times 350 \times 265 \times 128$	avg-pool	$1 \times 350 \times 265 \times 1$	$S_2, S_{i2}^s, S_{i2}^c$
conv2-2	$1 \times 350 \times 265 \times 128$			
pool	$1 \times 175 \times 133 \times 128$	resize	$1 \times 175 \times 133 \times 1$	
conv3-1	$1 \times 175 \times 133 \times 256$	avg-pool	$1 \times 175 \times 133 \times 1$	$S_3, S_{i3}^s, S_{i3}^c$
conv3-2, 3-3,3-4	$1 \times 175 \times 133 \times 256$			
pool	$1 \times 88 \times 67 \times 256$	resize	$1 \times 88 \times 67 \times 1$	
conv4-1	$1 \times 88 \times 67 \times 512$	avg-pool	$1 \times 88 \times 67 \times 1$	$S_4, S_{i4}^s, S_{i4}^c$
conv4-2, 4-3,4-4	$1 \times 88 \times 67 \times 512$			
pool	$1 \times 44 \times 34 \times 512$	resize	$1 \times 44 \times 34 \times 1$	
conv5-1	$1 \times 44 \times 34 \times 512$	avg-pool	$1 \times 44 \times 34 \times 1$	$S_5, S_{i5}^s, S_{i5}^c$

- Photorealism regularization: Affine loss( $L_m$ )는 output image( $X$ )와 content image( $C$ )의 Matting Laplacian Matrix( $M_C$ )로부터 구한다. Output image의 크기가  $h \times w \times c$ 라고 하면,  $M_C$ 의 크기는  $hw \times hw$ 가 된다. Output image  $X$ 의  $i$  번째 channel을  $hw \times 1$ 로 resize한 것을  $V_i[X]$ 로 두면, Affine loss는 다음과 같이 계산된다.

$$L_m = \sum_{i=1}^c V_i[X]^T M_C V_i[X]$$

참고로, Image Matting은 foreground image를 뽑아내는 방법이다.

- Total variance loss: 논문에서는 total variance loss를 언급하지 않고 있다. Total variance loss는 식(12)와 동일하다.
- Loss minimization을 통해 생성된 image는 local affine smooth 방식으로 다시 처리되었다 (local affine smooth의 효과가 얼마나 되는지 확인 필요).



(a) Content Image



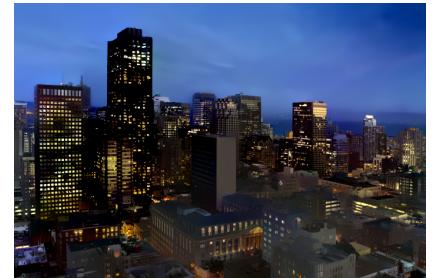
(b) Style Image



(c) Non Matting



(d) Only Matting



(e) Non Matting then Matting

그림 52: Content, Style and Stylized Images

## 10 Miscellaneous

### 10.1 Optimization Methods

#### ♠ Mini Batch and SGD

- In Stochastic Gradient Descent, you use only 1 training example before updating the gradients. When the training set is large, SGD can be faster. But the parameters will "oscillate" toward the minimum rather than converge smoothly.
- The difference between gradient descent, mini-batch gradient descent and stochastic gradient descent is the number of examples you use to perform one update step.
- You have to tune a learning rate hyperparameter  $\alpha$ .
- With a well-turned mini-batch size, usually it outperforms either gradient descent or stochastic gradient descent (particularly when the training set is large).

#### ♠ GD and Momentum

- Momentum takes past gradients into account to smooth out the steps of gradient descent. It can be applied with batch gradient descent, mini-batch gradient descent or stochastic gradient descent.
- You have to tune a momentum hyperparameter  $\beta$  and a learning rate  $\alpha$ .
- Common values for  $\beta$  range from 0.8 to 0.999. If you don't feel inclined to tune this,  $\beta = 0.9$  is often a reasonable default.

$$\begin{aligned} v &\leftarrow \beta v + (1 - \beta)dW \\ W &\leftarrow W - \alpha v \end{aligned}$$

## ♠ Adam

$$\begin{aligned}
v &\leftarrow \beta_1 v + (1 - \beta_1)dW = v + (1 - \beta_1)(dW - v) \\
v_{bc} &= \frac{v}{1 - \beta_1^t} \\
s &\leftarrow \beta_2 s + (1 - \beta_2)dW^2 = s + (1 - \beta_2)(dW^2 - s) \\
s_{bc} &= \frac{s}{1 - \beta_2^t} \\
W &\leftarrow W - \alpha \frac{v_{bc}}{\sqrt{s_{bc}} + \epsilon}
\end{aligned}$$

## 10.2 Batch Normalization

Batch Normalization Layer는 Activation Function 앞에 놓여진다. Batch Normalization의 backward propagation은 Computational Graph를 통해서 계산 할 수 있다.

Forward Propagation	Backward Propagation	
$X$	$dX = dX_c + \frac{1}{N}d\mu$	$\nwarrow$
$\downarrow$	$\uparrow$	$\nearrow$
$\mu = \frac{1}{N} \sum X$	$d\mu = -\sum dX_c$	
$\searrow$	$\uparrow$	$\nearrow$
$X_c = X - \mu$	$dX_c = \frac{1}{\sigma}dX_n + \frac{2X_c}{N}dV$	$\nwarrow$
$\downarrow$	$\uparrow$	$\nearrow$
$V = \frac{X_c^2}{N}$	$dV = \frac{d\sigma}{2\sigma}$	
$\downarrow$	$\uparrow$	$\uparrow$
$\sigma = \sqrt{V}$	$d\sigma = -\sum \frac{X_c \circ dX_n}{\sigma^2}$	$\uparrow$
$\searrow$	$\uparrow$	$\nearrow$
$\gamma, \beta$	$dX_n = \gamma \circ dY$	$d\gamma = \sum X_n \circ dY$
$X_n = \frac{X_c}{\sigma}$	$d\beta = \sum dY$	
$\downarrow$	$\uparrow$	$\nearrow$
$Y = \gamma X_n + \beta$	$dY$	

## ♠ Back Propagation Speed Up 구현<sup>39</sup>

$N$  개의 data  $x_1, \dots, x_n (x_i \in \mathbb{R}^m)$ 에 대하여

$$\begin{aligned}
\mu &= \frac{1}{N} \sum_{i=1}^N x_i, \\
\sigma^2 &= \frac{1}{N} \sum_{i=1}^n (x_i - \mu)^2, \\
\bar{x}_i &= \frac{x_i - \mu}{\sigma}, \\
y_i &= \gamma \bar{x}_i + \beta.
\end{aligned}$$

이제 Loss Function  $L$ 에 대한 gradient  $\frac{\partial L}{\partial x_i}$ 를 계산해 보자.

$$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial \bar{x}_i} \frac{\partial \bar{x}_i}{\partial x_i} + \frac{\partial L}{\partial \sigma^2} \frac{\partial \sigma^2}{\partial x_i} + \frac{\partial L}{\partial \mu} \frac{\partial \mu}{\partial x_i}$$

$\frac{\partial L}{\partial x_i}$ 는 3개 식의 합으로 표현되는데, 각각을 나누어서 계산해 보자.

<sup>39</sup><https://costapt.github.io/2016/07/09/batch-norm-alt/>

- 첫번째 식  $\frac{\partial L}{\partial \bar{x}_i} \frac{\partial \bar{x}_i}{\partial x_i}$  의 두 식은 각각 다음과 같이 계산된다.

$$\begin{aligned}\frac{\partial L}{\partial \bar{x}_i} &= \frac{\partial L}{\partial y_i} \frac{\partial y_i}{\partial \bar{x}_i} \\ &= \frac{\partial L}{\partial y_i} \gamma, \\ \frac{\partial \bar{x}_i}{\partial x_i} &= \frac{1}{\sigma}, \\ \frac{\partial L}{\partial \bar{x}_i} \frac{\partial \bar{x}_i}{\partial x_i} &= \frac{\gamma}{\sigma} \frac{\partial L}{\partial y_i}.\end{aligned}$$

- 두번째 식  $\frac{\partial L}{\partial \sigma^2} \frac{\partial \sigma^2}{\partial x_i}$ :

$$\begin{aligned}\frac{\partial L}{\partial \sigma^2} &= \sum_i^N \frac{\partial L}{\partial \bar{x}_i} \frac{\partial \bar{x}_i}{\partial \sigma^2} \\ &= \sum_i^N \frac{\partial L}{\partial y_i} \gamma (x_i - \mu) \left( -\frac{1}{2\sigma^3} \right) \\ &= -\frac{\gamma}{2\sigma^3} \sum_i^N \frac{\partial L}{\partial y_i} (x_i - \mu), \\ \frac{\partial L}{\partial \sigma^2} \frac{\partial \sigma^2}{\partial x_i} &= -\frac{\gamma}{2\sigma^3} \left[ \sum_j^N \frac{\partial L}{\partial y_j} (x_j - \mu) \right] \left( \frac{2}{N}(x_i - \mu) \right) \\ &= -\frac{\gamma}{N\sigma} \left[ \sum_j^N \frac{\partial L}{\partial y_j} \bar{x}_j \right] \bar{x}_i \leftarrow \frac{\partial L}{\partial \gamma} = \sum_j^N \frac{\partial L}{\partial y_j} \bar{x}_j \\ &= -\frac{\gamma}{N\sigma} \frac{\partial L}{\partial \gamma} \bar{x}_i.\end{aligned}$$

- 마지막 세번째 식  $\frac{\partial L}{\partial \mu} \frac{\partial \mu}{\partial x_i}$ :

$$\begin{aligned}\frac{\partial L}{\partial \mu} &= \sum_{i=1}^N \frac{\partial L}{\partial \bar{x}_i} \frac{\partial \bar{x}_i}{\partial \mu} + \frac{\partial L}{\partial \sigma^2} \frac{\partial \sigma^2}{\partial \mu} \\ &= \sum_{i=1}^N \frac{\partial L}{\partial y_i} \gamma \left( -\frac{1}{\sigma} \right) + \frac{\partial L}{\partial \sigma^2} \sum_{i=1}^N -\frac{2}{N}(x_i - \mu) \\ &= -\frac{\gamma}{\sigma} \sum_{i=1}^N \frac{\partial L}{\partial y_i}, \\ \frac{\partial L}{\partial \mu} \frac{\partial \mu}{\partial x_i} &= -\frac{\gamma}{N\sigma} \sum_{i=1}^N \frac{\partial L}{\partial y_i} \\ &= -\frac{\gamma}{N\sigma} \frac{\partial L}{\partial \beta} \leftarrow \frac{L}{\beta} = \sum_{i=1}^N \frac{\partial L}{\partial y_i}.\end{aligned}$$

- 이제 세 식을 합치면

$$\frac{\partial L}{\partial x_i} = \frac{\gamma}{N\sigma} \left( N \frac{\partial L}{\partial y_i} - \frac{\partial L}{\partial \gamma} \bar{x}_i - \frac{\partial L}{\partial \beta} \right)$$

```
def batchnorm_backward_alt(dout, cache):
    gamma, xhat, istd = cache # istd = 1/std, xhat = Xn
    N, _ = dout.shape

    dbeta = np.sum(dout, axis=0)
```

```

dgamma = np.sum(xhat * dout, axis=0)
dx = (gamma*istd/N) * (N*dout - xhat*dgamma - dbeta)

return dx, dgamma, dbeta

```

### 10.3 Naive Bayes Spam Filter

각각의 단어가 나타나는 사건은 서로 독립임을 가정한다. 이 가정은 spam에 등장하는 단어들이 서로 연관되어 있을 수 있는 가능성을 무시하는 가정이다.

$$\begin{aligned}
P(\text{Spam}|w_1, w_2, \dots, w_n) &= \frac{P(w_1, w_2, \dots, w_n|\text{Spam}) P(\text{Spam})}{P(w_1, w_2, \dots, w_n)} \\
&= \frac{P(w_1|\text{Spam})P(w_2|\text{Spam}) \cdots P(w_n|\text{Spam}) P(\text{Spam})}{P(w_1)P(w_2) \cdots P(w_n)}
\end{aligned}$$

Naive Bayes Spam Filter를 구현한 일부 코드는 분모  $P(w_1)P(w_2) \cdots P(w_n)$ 는 카테고리와 무관하기 때문에 무시하는 경우도 있다.

### 10.4 Support Vector Machine

SVM을 간략히 정리해 보자<sup>40</sup>.

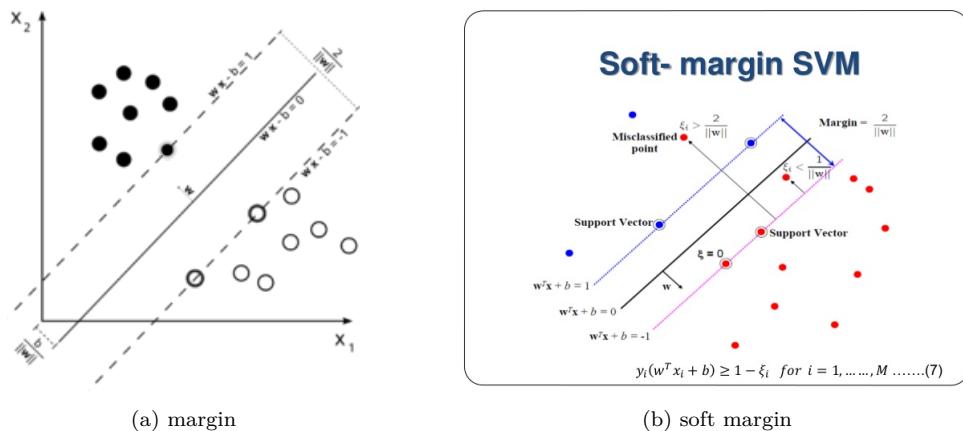


그림 53: SVM Margin

- 주어진 data

$$(\vec{x}_1, y_1), \dots, (\vec{x}_n, y_n), x_i \in \mathbb{R}^p, y_i \in \{-1, 1\}$$

- If the training data are linearly separable, we can select two parallel hyperplanes that separate the two classes of data, so that the distance between them is as large as possible. The region bounded by these two hyperplanes is called the "margin", and the maximum-margin hyperplane is the hyperplane that lies halfway between them. These hyperplanes can be described by the equations

$$\vec{w} \cdot \vec{x} - b = 1 \text{ and } \vec{w} \cdot \vec{x} - b = -1.$$

<sup>40</sup>[https://en.wikipedia.org/wiki/Support\\_vector\\_machine](https://en.wikipedia.org/wiki/Support_vector_machine)

- Geometrically, the distance between these two hyperplanes is  $\frac{2}{\|\vec{w}\|}$ , so to maximize the distance between the planes we want to minimize  $\|\vec{w}\|$ . As we also have to prevent data points from falling into the margin, we add the following constraint: for each  $i$  either

$$\vec{w} \cdot \vec{x}_i - b \geq 1, \text{ if } y_i = 1 \quad \text{or} \quad \vec{w} \cdot \vec{x}_i - b \leq -1, \text{ if } y_i = -1.$$

These constraints state that each data point must lie on the correct side of the margin.

This can be rewritten as:

$$y_i(\vec{w} \cdot \vec{x}_i - b) \geq 1, \quad \text{for all } 1 \leq i \leq n. \quad (13)$$

We can put this together to get the optimization problem:

$$\begin{aligned} & \text{Minimize} && \|\vec{w}\| \\ & \text{s.t.} && y_i(\vec{w} \cdot \vec{x}_i - b) \geq 1, \text{ for } i = 1, \dots, n \end{aligned}$$

The  $\vec{w}$  and  $b$  that solve this problem determine our classifier,  $\vec{x} \mapsto \text{sgn}(\vec{w} \cdot \vec{x} - b)$ . An easy-to-see but important consequence of this geometric description is that the max-margin hyperplane is completely determined by those  $\vec{x}_i$  which lie nearest to it. These  $\vec{x}_i$  are called "support vectors."

- Soft margin: To extend SVM to cases in which the data are not linearly separable, we introduce the "hinge loss" function,

$$\max(0, 1 - y_i(\vec{w} \cdot \vec{x}_i - b)).$$

This function is zero if the constraint in (13) is satisfied, in other words, if  $\vec{x}_i$  lies on the correct side of the margin. For data on the wrong side of the margin, the function's value is proportional to the distance from the margin.

We then wish to minimize

$$\left[ \frac{1}{n} \sum_{i=1}^n \max(0, 1 - y_i(\vec{w} \cdot \vec{x}_i - b)) \right] + \lambda \|\vec{w}\|^2,$$

where the parameter  $\lambda$  determines the tradeoff between increasing the margin-size and ensuring that the  $\vec{x}_i$  lie on the correct side of the margin. Thus, for sufficiently small values of  $\lambda$ , the soft-margin SVM will behave identically to the hard-margin SVM if the input data are linearly classifiable, but will still learn if a classification rule is viable or not. To extend SVM to cases in which the data are not linearly separable, we introduce the "hinge loss" function,

$$\max(0, 1 - y_i(\vec{w} \cdot \vec{x}_i - b)).$$

This function is zero if the constraint in (13) is satisfied, in other words, if  $\vec{x}_i$  lies on the correct side of the margin. For data on the wrong side of the margin, the function's value is proportional to the distance from the margin.

We then wish to minimize

$$\left[ \frac{1}{n} \sum_{i=1}^n \max(0, 1 - y_i(\vec{w} \cdot \vec{x}_i - b)) \right] + \lambda \|\vec{w}\|^2, \quad (14)$$

where the parameter  $\lambda$  determines the tradeoff between increasing the margin-size and ensuring that the  $\vec{x}_i$  lie on the correct side of the margin. Thus, for sufficiently small values of  $\lambda$ , the soft-margin SVM will behave identically to the hard-margin SVM if the input data are linearly classifiable, but will still learn if a classification rule is viable or not.

- Primal: Minimizing ((14)) can be rewritten as a constrained optimization problem with a differentiable objective function in the following way.

For each  $i \in \{1, \dots, n\}$  we introduce a variable  $\zeta_i = \max(0, 1 - y_i(w \cdot x_i - b))$ . Note that  $\zeta_i$  is the smallest nonnegative number satisfying  $y_i(w \cdot x_i - b) \geq 1 - \zeta_i$ .

Thus we can rewrite the optimization problem as follows

$$\begin{aligned} & \text{minimize} && \frac{1}{n} \sum_{i=1}^n \zeta_i + \lambda \|w\|^2 \\ & \text{s.t.} && y_i(w \cdot x_i - b) \geq 1 - \zeta_i \text{ and } \zeta_i \geq 0, \text{ for all } i. \end{aligned}$$

This is called the "primal" problem.

- Dual: By solving for the Lagrangian dual of the above problem, one obtains the simplified problem

$$\begin{aligned} & \text{maximize} && f(c_1 \dots c_n) = \sum_{i=1}^n c_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n y_i c_i (x_i \cdot x_j) y_j c_j, \\ & \text{s.t.} && \sum_{i=1}^n c_i y_i = 0, \text{ and } 0 \leq c_i \leq \frac{1}{2n\lambda} \text{ for all } i. \end{aligned}$$

This is called the "dual" problem. Since the dual maximization problem is a quadratic function of the  $c_i$  subject to linear constraints, it is efficiently solvable by quadratic programming algorithms.

Here, the variables  $c_i$  are defined such that

$$\vec{w} = \sum_{i=1}^n c_i y_i \vec{x}_i.$$

oir 식은 Lagrange function  $L = \frac{1}{2} \|\vec{w}\|^2 - \sum_{i=1}^n c_i (y_i(w \cdot x_i - b) - 1)$  을 편미분하여 얻을 수 있다 ( $\frac{\partial L}{\partial w} = 0$ ).  $\frac{\partial L}{\partial b} = 0$  를 계산하면,  $\sum_{i=1}^n c_i y_i = 0$  o] 나온다.

Moreover,  $c_i = 0$  exactly when  $\vec{x}_i$  lies on the correct side of the margin, and  $0 < c_i < (2n\lambda)^{-1}$  when  $\vec{x}_i$  lies on the margin's boundary. It follows that  $\vec{w}$  can be written as a linear combination of the support vectors. The offset,  $b$ , can be recovered by finding an  $\vec{x}_i$  on the margin's boundary and solving

$$y_i(\vec{w} \cdot \vec{x}_i - b) = 1 \iff b = \vec{w} \cdot \vec{x}_i - y_i.$$

(Note that  $y_i^{-1} = y_i$  since  $y_i = \pm 1$ .)

- Kernel Trick:

## 10.5 Redistricted Boltzmann Machine

### ♠ Partition Function

visible state  $V$  with 3 units, hidden state  $H$  with 2 units, weight matrix  $W$ .

$$V = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix},$$

$$W = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix}$$

- $H = (0 \ 0)^T$  인 경우,

$$H^T WV = \begin{pmatrix} 0 & 0 \end{pmatrix} \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix}$$

$$= 0.$$

$V$ 의 3원소  $v_1, v_2, v_3$ 에 대한, 0,1 조합은 모두  $2^3 = 8$ 가지. 따라서,  $\sum_{h=(0,0)} e^{-E(V,H)} = 8$ .

- $H = (0 \ 1)^T$  인 경우,

$$H^T WV = \begin{pmatrix} 0 & 1 \end{pmatrix} \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix}$$

$$= \begin{pmatrix} w_{21} & w_{22} & w_{23} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix}$$

$$= w_{21}v_1 + w_{22}v_2 + w_{23}v_3.$$

$$\sum_{H=(0,1), V} e^{-E(V,H)} = \sum e^{w_{21}v_1 + w_{22}v_2 + w_{23}v_3}$$

$$= (1 + e^{w_{21}})(1 + e^{w_{22}})(1 + e^{w_{23}}).$$

- $H = (1 \ 0)^T$  인 경우,

$$H^T WV = \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix}$$

$$= \begin{pmatrix} w_{11} & w_{12} & w_{13} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix}$$

$$= w_{11}v_1 + w_{12}v_2 + w_{13}v_3.$$

$$\sum_{H=(1,0), V} e^{-E(V,H)} = \sum e^{w_{11}v_1 + w_{12}v_2 + w_{13}v_3}$$

$$= (1 + e^{w_{11}})(1 + e^{w_{12}})(1 + e^{w_{13}}).$$

- $H = (1 \ 1)^T$  인 경우,

$$\begin{aligned}
H^T W V &= \begin{pmatrix} 1 & 1 \end{pmatrix} \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} \\
&= \begin{pmatrix} w_{11} + w_{21} & w_{12} + w_{22} & w_{13} + w_{23} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} \\
&= (w_{11} + w_{21})v_1 + (w_{12} + w_{22})v_2 + (w_{13} + w_{23})v_3. \\
\sum_{H=(1,1), V} e^{-E(V,H)} &= \sum e^{(w_{11}+w_{21})v_1+(w_{12}+w_{22})v_2+(w_{13}+w_{23})v_3} \\
&= (1 + e^{w_{11}+w_{21}})(1 + e^{w_{12}+w_{22}})(1 + e^{w_{13}+w_{23}}).
\end{aligned}$$

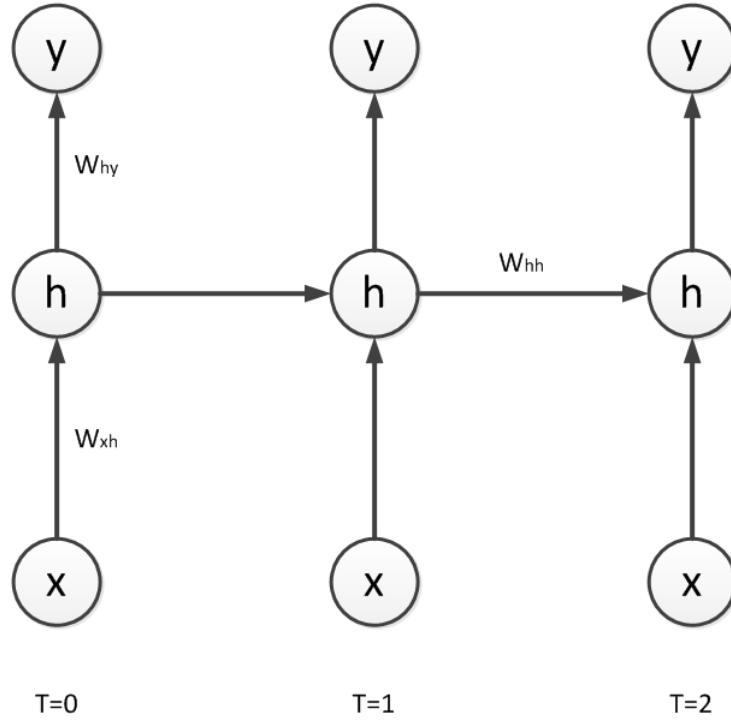
- 4가지 경우를 모두 합치면 다음과 같다.

$$\begin{aligned}
H^T &= \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix}, \\
H^T W V &= \begin{pmatrix} 0 & 0 & 0 \\ w_{21} & w_{22} & w_{23} \\ w_{11} & w_{12} & w_{13} \\ w_{11} + w_{21} & w_{12} + w_{22} & w_{13} + w_{23} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix}, \\
&\rightarrow \begin{pmatrix} 1 + e^0 & 1 + e^0 & 1 + e^0 \\ 1 + e^{w_{21}} & 1 + e^{w_{22}} & 1 + e^{w_{23}} \\ 1 + e^{w_{11}} & 1 + e^{w_{12}} & 1 + e^{w_{13}} \\ 1 + e^{w_{11}+w_{21}} & 1 + e^{w_{12}+w_{22}} & 1 + e^{w_{13}+w_{23}} \end{pmatrix}.
\end{aligned}$$

row-wise product and then sum.

## 10.6 기타

4. The figure below shows a Recurrent Neural Network (RNN) with one input unit  $x$ , one logistic hidden unit  $h$ , and one linear output unit  $y$ . The RNN is unrolled in time for  $T=0, 1$ , and  $2$ .



The network parameters are:  $W_{xh} = -0.1$ ,  $W_{hh} = 0.5$ ,  $W_{hy} = 0.25$ ,  $h_{\text{bias}} = 0.4$ , and  $y_{\text{bias}} = 0.0$ .

If the input  $x$  takes the values  $18, 9, -8$  at time steps  $0, 1, 2$  respectively, the hidden unit values will be  $0.2, 0.4, 0.8$  and the output unit values will be  $0.05, 0.1, 0.2$  (you can check these values as an exercise). A variable  $z$  is defined as the total input to the hidden unit before the logistic nonlinearity.

If we are using the squared loss, with targets  $t_0, t_1, t_2$ , then the sequence of calculations required to compute the total error  $E$  is as follows:

$$\begin{aligned} z_0 &= W_{xh}x_0 + h_{\text{bias}} & z_1 &= W_{xh}x_1 + W_{hh}h_0 + h_{\text{bias}} & z_2 &= W_{xh}x_2 + W_{hh}h_1 + h_{\text{bias}} \\ h_0 &= \sigma(z_0) & h_1 &= \sigma(z_1) & h_2 &= \sigma(z_2) \\ y_0 &= W_{hy}h_0 + y_{\text{bias}} & y_1 &= W_{hy}h_1 + y_{\text{bias}} & y_2 &= W_{hy}h_2 + y_{\text{bias}} \\ E_0 &= \frac{1}{2}(t_0 - y_0)^2 & E_1 &= \frac{1}{2}(t_1 - y_1)^2 & E_2 &= \frac{1}{2}(t_2 - y_2)^2 \\ E &= E_0 + E_1 + E_2 \end{aligned}$$

If the target output values are  $t_0 = 0.1, t_1 = -0.1, t_2 = -0.2$  and the squared error loss is used, what is the value of the error derivative just before the hidden unit nonlinearity at  $T = 1$  (i.e.  $\frac{\partial E}{\partial z_1}$ )? Write your answer up to at least the fourth decimal place.

그림 54: Example

**Example 10.1.** (그림 54)과 같이 주어진 RNN에서  $\frac{\partial E}{\partial z_1}$  을 계산해 보자.

$$\begin{aligned} \frac{\partial E}{\partial z_1} &= \frac{\partial E_1}{\partial z_1} + \frac{\partial E_2}{\partial z_1} \\ \frac{\partial E_1}{\partial z_1} &= \frac{\partial E_1}{\partial y_1} \frac{\partial y_1}{\partial h_1} \frac{\partial h_1}{\partial z_1} = (y_1 - t_1) \times W_{hy} \times h_1(1 - h_1) \\ \frac{\partial E_2}{\partial z_1} &= \frac{\partial E_2}{\partial y_2} \frac{\partial y_2}{\partial h_2} \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial h_1} \frac{\partial h_1}{\partial z_1} = (y_2 - t_2) \times W_{hy} \times h_2(1 - h_2) \times W_{hh} \times h_1(1 - h_1) \end{aligned}$$