

Deep Learning

조희철

2018년 9월 11일

차례

1	Language Model	3
2	Preliminary	10
2.1	Chain Rule	10
2.2	기타 등등	12
3	Deep Neural Net	13
4	Back Propagation	14
4.1	A Neural Network in 11 Lines	14
4.2	Linear Classifier	15
4.3	Simple Model Training	16
5	Tensorflow & Network	18
5.1	Tensorflow	18
5.2	Network 설계	27
5.3	Advanced Tensorflow	31
6	Convolution Neural Network	35
6.1	im2col	35
6.2	VGG의 이해	39
6.3	Image File Loading & Save	42
6.4	SqueezeNet	44
7	Recurrent Neural Network	45
7.1	RNN Toy Model: i am trask blog	45
7.2	DENNY BRITZ Language Model	48
7.3	Minimal Character-Level Vanilla RNN Model by Andrej Karpathy	50
7.4	LSTM(Long Short Term Memory) Networks	52
7.5	RNN with TensorFlow	56
7.6	Character-Level Vanilla RNN Model by Andrej Karpathy	62
7.7	Image Captioning with RNNs(CS231n)	62
7.8	Neural Machine Translation	64
7.9	Text Data 다루기	70

8 Reinforcement Learning	71
8.1 Q-Table, Q-Network	71
8.2 DQN	73
8.3 Policy Gradient	74
9 Variational AutoEncoder & Generative Adversarial Networks	75
9.1 AutoEncoder	75
9.2 Cross Entropy & KL Divergence	78
9.3 Variational AutoEncoder	79
9.4 여러가지 VAE	83
9.5 Vanilla GAN	86
9.6 Wasserstein GAN	90
9.7 EBGAN & BEGAN	95
9.8 DCGAN	98
9.9 StackGAN(2016)	104
9.10 Texture Synthesis(2015)	114
9.11 Neural Style(2017)	115
9.12 Deep Photo Style Transfer	119
9.13 Pix2Pix GAN(2016)	121
9.14 CycleGAN(2017)	125
10 Speech Recognition	129
10.1 From Speech Waveform To MFCC	129
10.2 Model Training	133
11 Miscellaneous	135
11.1 Optimization Methods	135
11.2 Batch Normalization	135
11.3 Naive Bayes Spam Filter	138
11.4 Principal Components Analysis	138
11.5 Support Vector Machine	140
11.6 Redistricted Boltzmann Machine	143
11.7 기타	145
11.8 Deep Learning Note	146

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>)

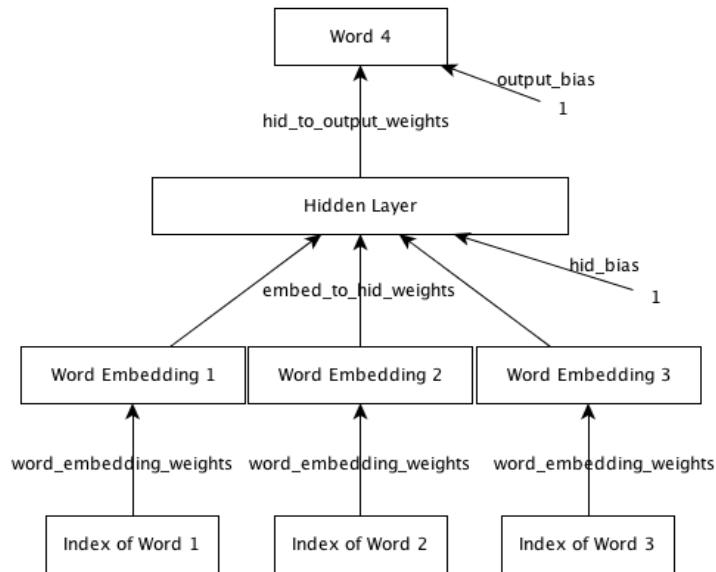


그림 1: 강의에 나오는 network 그림

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).

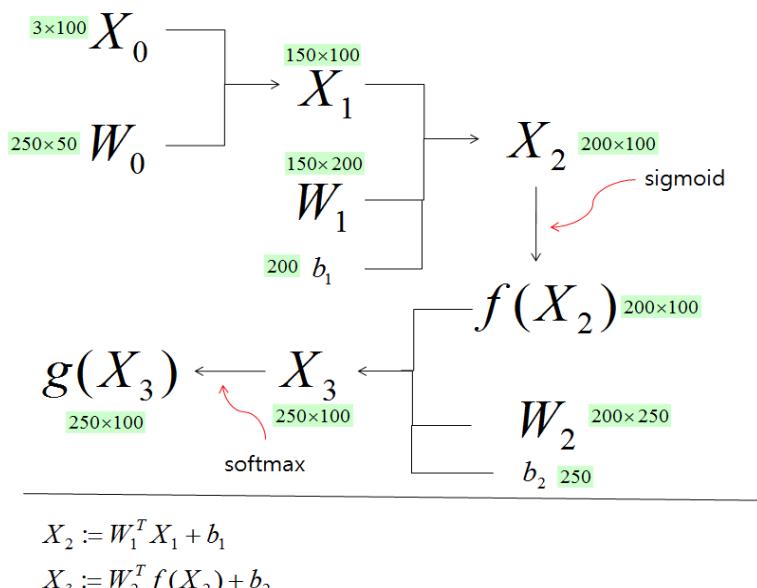


그림 2: 행렬로 표현한 network

♠ 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, 3×100 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 250×50 이 된다. 학습이 이루어지면, 이 값이 각 단어들의 feature가 되고, 이 feature가 유사하다는 것은 대체할 수

있다는 것으로 해석할 수도 있다. Assignment에서는 ‘display_nearest_words’라는 함수를 제공하고 있는데, 이 함수는 유사한 feature(Euclidean Distance)를 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.

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

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 250×100 행렬이 된다. 이게 W_0 와 K_1, K_2, K_3 를 결합한 행렬과의 곱을 계산을 보자.

$$W_0^T \ (K_1 \ | \ K_2 \ | \ K_3) = (W_0^T K_1 \ | \ W_0^T K_2 \ | \ W_0^T K_3)$$

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

forwad 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). λ 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 50×100 행렬로 나누어져 있으므로, $dX_1(150 \times 100)$ 도 동일하게 50×100 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)¹: 여러개의 단어들을 나열한 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, brown)으로 부터 (fox)를 예측하는 모델.

- Skip-Gram: 주어진 단어를 중심으로 전, 후의 단어를 예측하는 모델. 느리지만 빈도가 낮은 단어들에 적합. 대규모 데이터셋에서는 Skip-Gram이 더 정확한 것으로 알려져 있다.

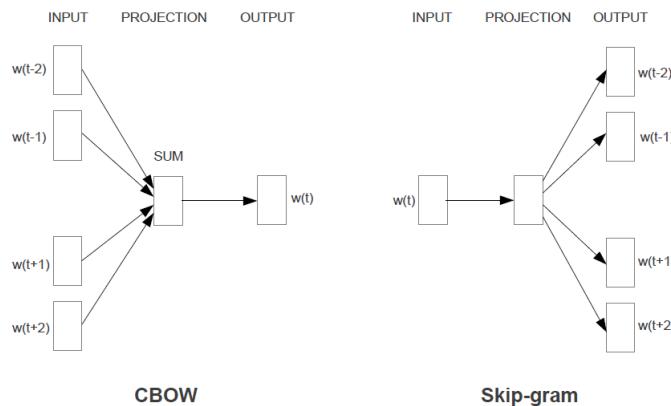


그림 3: CBOW & SKip-Gram

Source Text	Training Samples
The quick brown fox jumps over the lazy dog. →	(the, quick) (the, brown)
The quick brown fox jumps over the lazy dog. →	(quick, the) (quick, brown) (quick, fox)
The quick brown fox jumps over the lazy dog. →	(brown, the) (brown, quick) (brown, fox) (brown, jumps)
The quick brown fox jumps over the lazy dog. →	(fox, quick) (fox, brown) (fox, jumps) (fox, over)

그림 4: SKip-Gram 방식에서 training data 만들기. 단순히 (input,output) 쌍을 반복해서 만들어 준다.

- Word2Vec²은 CBOW와 Skip-Gram 방식의 embedding을 구현한 것으로, Google에 있던 Mikolov 등에 의해 C++ 라이브러리로 개발했다. Python에서는 gensim package에 'Word2Vec', Doc2Vec, TaggedDocument class가 구현되어 있다.
- Word2Vec 모델에서는 기본적으로 모델 사이즈가 크다. 예를 들어, 단어가 10,000개, embedding dimension이 300만 되어도 3백만개의 변수가 필요하다. Mikolov는 후속 논문에서 이를 해결하는 방법을 제시했다³.
 - Treating common word pairs or phrases as single “words” in their model.

¹BOW는 문장의 구조는 무시하고 단어의 출연 빈도만을 counting하는 가장 간단하고 효과적인 방법이다.

²<http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/>

³Distributed Representations of Words and Phrases and their Compositionalities, <https://arxiv.org/abs/1310.4546>

- Subsampling frequent words to decrease the number of training examples.
- Modifying the optimization objective with a technique they called “Negative Sampling”, which causes each training sample to update only a small percentage of the model’s weights.
- “Negative Sampling”에 대한 설명은 이곳⁴을 참조. tensorflow로 구현할 때는 `tf.nn.nce_loss`를 이용하면 된다.
- When training the network on the word pair (“fox”, “quick”), recall that the “label” or “correct output” of the network is a one-hot vector. That is, for the output neuron corresponding to “quick” to output a 1, and for all of the other thousands of output neurons to output a 0. With negative sampling, we are instead going to randomly select just a small number of “negative” words (let’s say 5) to update the weights for. (In this context, a “negative” word is one for which we want the network to output a 0 for). We will also still update the weights for our “positive” word (which is the word “quick” in our current example). The paper says that selecting 5-20 words works well for smaller datasets, and you can get away with only 2-5 words for large datasets.

♠ NLTK: Natural Language Toolkit

- NLTK(Natural Language Toolkit) 패키지는 교육용으로 개발된 자연어 처리 및 문서 분석용 파이썬 패키지다. 다양한 기능 및 예제를 가지고 있으며 실무 및 연구에서도 많이 사용된다.
- 말뭉치(corpus, 자연어 분석 작업을 위해 만든 샘플 문서 집합), 토큰 생성(파일로 부터 문장으로 단위로 분할, 또는 단어 단위로 분할), 형태소 분석, 품사 태깅.
- `sent_tokenize`, `word_tokenize`, `RegexpTokenizer`, `PorterStemmer`, `LancasterStemmer`, `untag`

♠ KoNLPy: Korean NLP in Python

- KoNLPy는 꼬꼬마, 코모란, 트위터, 한나눔, 은전한닢 다섯개 형태소 분석기를 묶어서 사용할 수 있도록 한 Package.
- 형태소 분석후 Word2Vec을 어떻게 적용하는지 구체적으로 알아보자. `konlpy.tag.Twitter`을 이용하여 `data`를 형태소 분석하여 파일로 저장한다.

```

fp = codecs.open("toji.txt","r",encoding="utf-8")
soup = BeautifulSoup(fp, "html.parser")
body = soup.select_one("body > text")
text = body.getText()

twt = Twitter()
lines = text.split("\n")
result = []
for line in lines:
    malist = twt.pos(line,norm=True,stem=True)
    temp = []
    for word in malist:
        if not word[1] in [ "Josa", "Eomi", "Punctuation" ]:

```

⁴<http://mccormickml.com/2017/01/11/word2vec-tutorial-part-2-negative-sampling/>

```

        temp.append(word[0])
result.append( " ".join(temp)).strip()

filename = 'toji_for_word2vec.txt'
with open(filename, 'w', encoding = 'utf-8') as fp:
    fp.write("\n".join(result))

```

Before	After
”저승에나 가서 잘사는가.”	저승 가다 잘살다
”등잔에 불을 켜야겠습니다.”	등잔 불 켄다

표 1: Twitter 형태소 분석 전후

- 형태소 분석 결과를 word2vec에 전달하여 모델을 만든다.

```

data = word2vec.LineSentence(filename)
model = word2vec.Word2Vec(data,size=200,window=10,hs=1,min_count=2,sg=1)
model.save('toji.model') # binary 파일

```

- size는 word embedding vector의 크기를 지정하고, window는 주변단어 몇개까지 분석할 것인지를 지정한다. sg=1은 skip-gram 방식을 그외는 CBOW방식을 지정한다. min_count 미만의 출현빈도를 가지는 단어는 무시한다.

♠ Skip Thought Vectors(2015)

- Ryan Kiros, Yukun Zhu, Ruslan Salakhutdinov, Richard S. Zemel, Antonio Torralba, Raquel Urtasun, Sanja Fidler, Skip-Thought Vectors⁵.
- Word2Vec이 단어에 대한 embedding vector를 구하는 것인데 반해, Skip Thought Vector는 문장에 대한 embedding vector를 구하는 모델이다.
- 모델은 문장을 encoding 하는 RNN, 이전 문장을 decoding하는 RNN, 다음 문장을 decoding하는 RNN으로 구성된다.

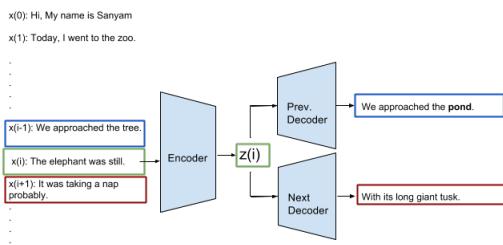


그림 5: Skip Thought Vectors Model Structure.

⁵<https://arxiv.org/abs/1506.06726>

2 Preliminary

2.1 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

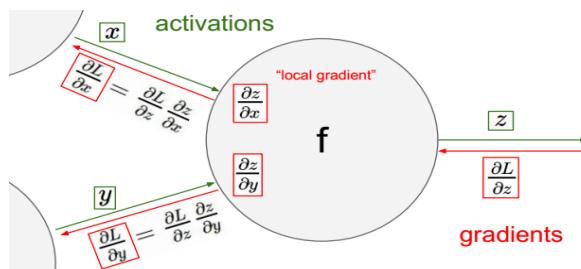


그림 6: Back Propagation & Chain Rule

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

♠ Example

[그림7]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 7: Example: back propagation

Forward Propagation:

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

Backward Propagation:

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

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

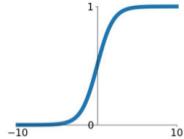
2.2 기타 등등

♠ Activation Functions

Activation Functions

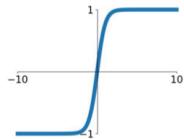
Sigmoid

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



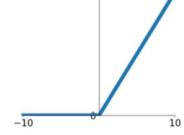
tanh

$$\tanh(x)$$



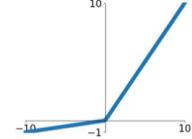
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$



Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

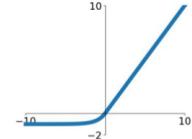


그림 8: Activation Functions

- Tensorflow의 `tf.nn.elu`에서 $\alpha = 1$.

3 Deep Neural Net

- Simple Net without Hidden layer

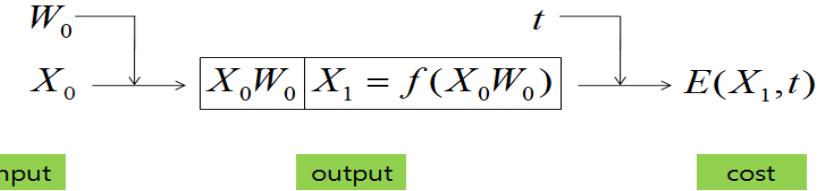
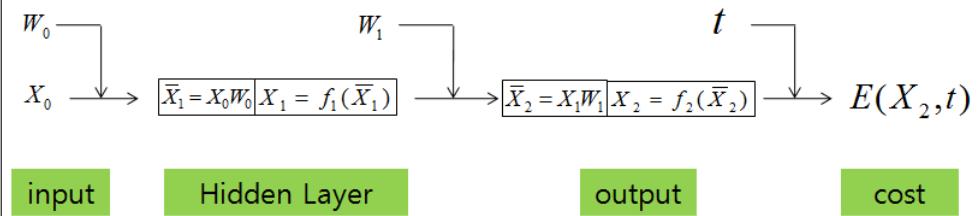


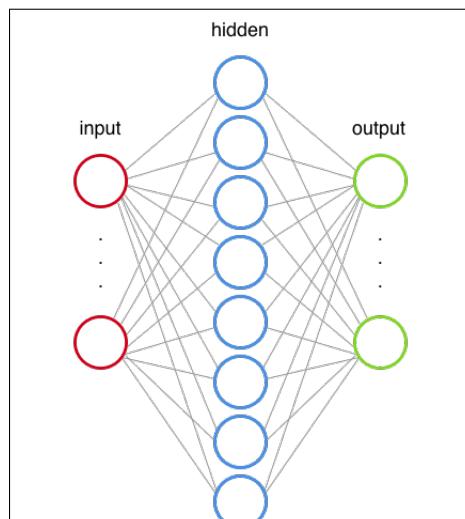
그림 9: Simple Net without Hidden Layer

- $f(x) = x$ 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 10: Neural Network with 1-hidden layer

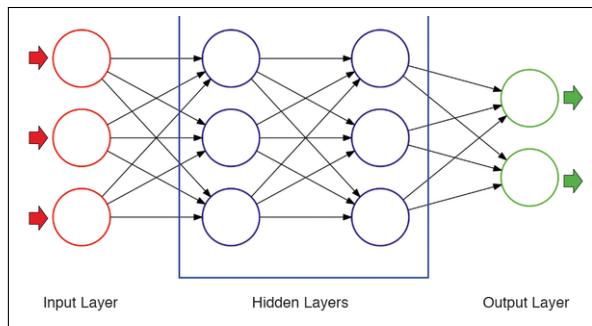
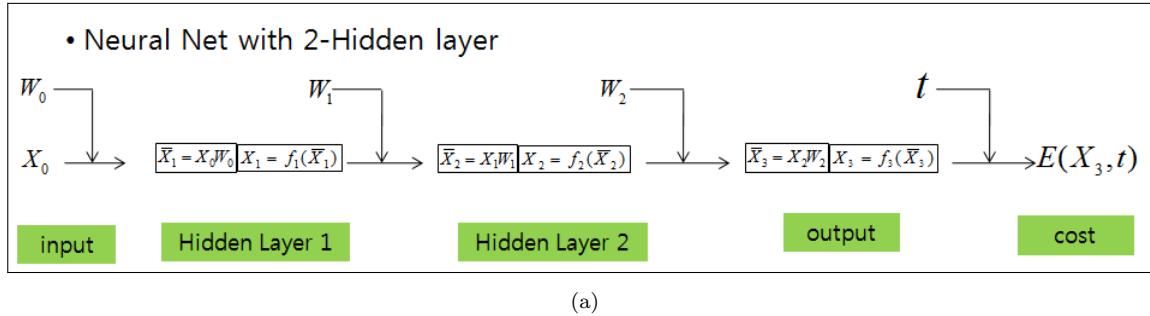


그림 11: Neural Network with 2-hidden layers

4 Back Propagation

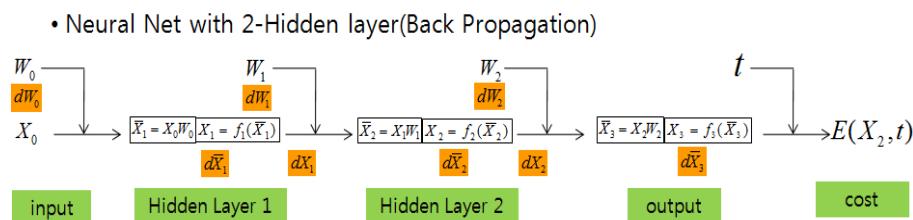


그림 12: Back Propagation

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 13: A Neural Network in 11 Lines

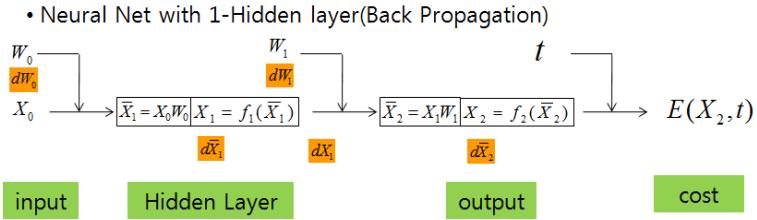


그림 14: Back Propagation with 1-Hidden Layer

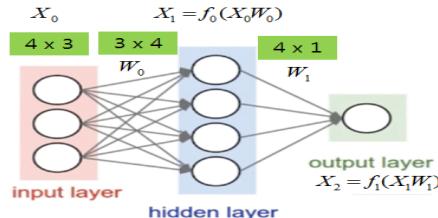


그림 15: Size of Neural Net

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

```

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 16: 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.

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

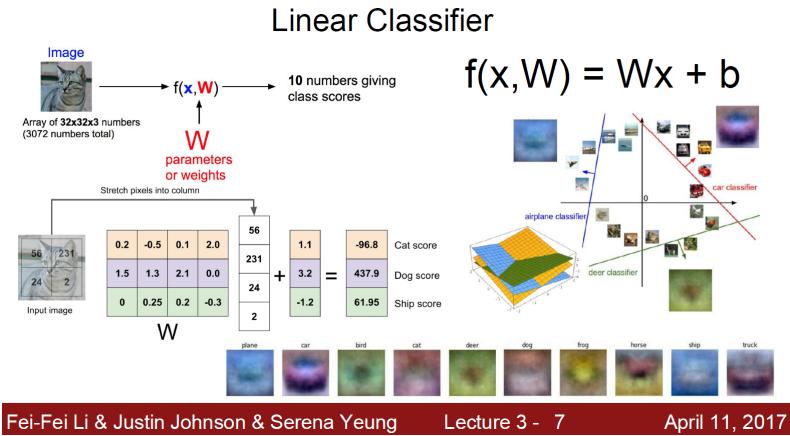


그림 17: Linear Classifier Model

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

```

4.3 Simple Model Training

- XOR 같은 간단한 문제일 경우, layer의 수가 많지 않기 때문에 weight 초기화가 중요하다. tensorflow의 default 초기값으로 잘 되지 않을 수 있다.
- learning rate은 input/output data의 scale과 관련이 있다. output data scale에 더 많은 영향을 받는다.
- tf.layers.dense의 kernel 초기값은 `tf.get_variable`와 동일. weight shape (N, M) 으면

$$\text{Uniform Random } \left[-\sqrt{\frac{6}{N+M}}, \sqrt{\frac{6}{N+M}} \right]$$

```

x_data = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=np.float32)
y_data = np.array([[0, 1], [1, 0], [1, 0]], dtype=np.float32)

```

```

#x_data = x_data/10.0
#y_data = y_data/10.0

```

weight 초기값	learning rate	loss
layer.desce default	0.1	loss = 0.25(bad)
layer.desce default	0.01	loss = 0.00001723(good)
layer.desce default	0.1	data scale을 1/10로 변경 loss = 0.000000458(good)
xavier	0.1	loss = 0.25(bad)
N(0,1)	0.1	XOR 문제는 feature가 2개 이므로 N(0,1)은 He initialization loss = 0.000001077(good)
N(0,0.1)	0.1	loss = 0.33(bad)

표 2: XOR test 결과: iteration 10,000번.

```

X = tf.placeholder(tf.float32, [None, 2]); Y = tf.placeholder(tf.float32, [None, 1])
learning_rate = 0.1

#init = tf.contrib.layers.xavier_initializer()
init = tf.random_normal_initializer(mean=0.0, stddev=1, dtype=tf.float32)

L1 = tf.layers.dense(X, units=2, activation=tf.nn.sigmoid, kernel_initializer=init)
L2 = tf.layers.dense(L1, units=1, activation=tf.nn.sigmoid, kernel_initializer=init)

loss = tf.reduce_mean(tf.nn.l2_loss(L2-Y))
opt = tf.train.AdamOptimizer(learning_rate).minimize(loss)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    for i in range(10001):
        sess.run(opt, feed_dict={X: x_data, Y: y_data})

        if i % 1000 == 0:
            err = sess.run(loss, feed_dict={X: x_data, Y: y_data})
            print('{} , loss = {}'.format(i, err))

    out = sess.run(L2, feed_dict={X: x_data, Y: y_data})
    print('result: {}'.format(out))

```

5 Tensorflow & Network

5.1 Tensorflow

♠ Tensorflow 기본 구조

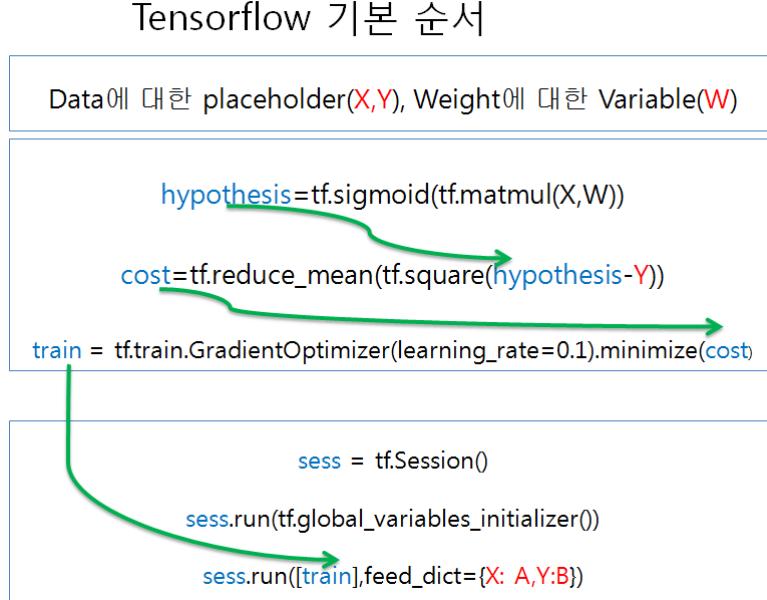


그림 18: Tensorflow 기본 흐름

♠ 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 rate⁶: learning rate을 training을 진행하면서 줄여주는 경우.
`lr = tf.Variable(0.0, trainable=False)`로 선언하고, `assign`으로 조절.
`sess.run(tf.assign(lr, learning_rate * (args.decay_rate ** e)))`

```
X = np.array([[0,0,1],[0,1,1],[1,0,1],[1,1,1]])
Y = np.array([[0,1,1,1]]).T
x = tf.placeholder(tf.float32, [None,3])
y = tf.placeholder(tf.float32, [None,1])

L1 = tf.layers.dense(x,units=3, activation = tf.sigmoid,name='L1')
L2 = tf.layers.dense(L1,units=1, activation = tf.sigmoid,name='L2')
train = tf.train.AdamOptimizer(learning_rate=1).minimize( tf.reduce_mean( 0.5*tf.square(L2-y)))
with tf.Session() as sess:
```

⁶<http://cs231n.github.io/neural-networks-3/>

```
sess.run(tf.global_variables_initializer())
for j in range(60000):
    sess.run(train, feed_dict={x: X, y: Y})
```

♠ Tensorflow Weight 선언

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

- `tf.Varialbe`: 같은 name의 변수를 생성해도 변형된 이름으로 생성된다(`W2:0`, `W2_1:0`).

```
W1 = tf.Variable([[ 0.8330605 ],[-0.35352278],[-0.44010514]], name='W1') # explicit
                initialization
W2 = tf.Variable(tf.truncated_normal([4, 4], stddev=0.02), name="W2")
W3 = tf.Variable(tf.random_normal([2, 2], stddev=0.02), name="W3")
```

- `tf.get_variable`: 같은 name의 변수를 생성하려하면 error 발생.

```
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 ‘checkpoint’에는 최근 5개의 checkpoint 가 저장되어 있다.
- ‘checkpoint’ 없이도 restore 가능하다.
- model_4000.ckpt.data-00000-of-00001, model_4000.ckpt.index, model_4000.ckpt.meta 3개 파일로 저장된다.
- 저장된 값을 복원할 때, 소스코드에서 graph 구조를 가지고 있으면, graph 구조를 저장하는 meta 파일은 없어도 된다.
- 실제 저장되어 있는 파일 이름은 model_4000.ckpt.data-00000-of-00001로 되어 있어도
saver.restore(sess, '...\\model_4000.ckpt') 형식으로 사용한다.
- saver = tf.train.Saver(tf.global_variables()) 또는
saver = tf.train.Saver(tf.trainable_variables())를 통해 saver를 만든다. batch normalization에서 moving-mean, variance는 trainable variable이 아니기 때문에
saver = tf.train.Saver(tf.trainable_variables())를 통해서는 restore가 되지 않는다.

```
self.saver = tf.train.Saver() # or saver = tf.train.Saver(tf.global_variables())
-----
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))
    # self.saver.restore(self.sess, tf.train.latest_checkpoint(checkpoint_dir))

    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']

```

♠ meta파일로 graph 복원

checkpoint 파일의 일부로 저장되는 meta파일을 이용하여 graph를 복원할 수 있다. tensor name을 알고 있어야 복원이 가능하다. tensor name은 GraphDef를 통해 알아 낼 수도 있다.

```

with tf.Session() as sess:
    last_chekpoint = tf.train.latest_checkpoint(checkpoint_dir)
    if ckpt and last_chekpoint:
        saver = tf.train.import_meta_graph(last_chekpoint+'.meta')
        saver.restore(sess, last_chekpoint)

    graph = tf.get_default_graph()

    loss = graph.get_tensor_by_name('sequence_loss/truediv:0')

    # feed dict 를위해placeholder 만별아보자 .
    ci = graph.get_tensor_by_name('caption_in:0')
    c0 = graph.get_tensor_by_name('caption_out:0')
    f = graph.get_tensor_by_name('features:0')
    batch_size = ci.get_shape().as_list()[0]

    minibatch = sample_coco_minibatch(small_data, split='val', batch_size=batch_size)
    captions, features, urls = minibatch

    l=sess.run(loss,feed_dict={f:features,ci:captions[:, :-1],c0:captions[:, 1:]})
    print('loss: {:.6f}'.format(l))
    graph_def = graph.as_graph_def()

    for node in graph_def.node:
        print(node.name)

```

♠ Tensorboard

- 아래의 예처럼, 같은 accuracy라도 train과 test에서 각각 계산해야 하므로, summary를 각각 만들어야 한다.

- log 저장 디렉토리는 없으면, 자동 생성된다.
- 간략하게 정리하면, 먼저 필요한 summary들을 만들고(필요하면 merge), FileWriter도 만든다. summary나 merge를 sess.run 한다. 그 결과를 writer에 add_summary 한다.
- 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)

```

♠ TFRecord

- TFRecord를 살펴보기 전에 Deep Learning에서 이미지를 다루는 방식에 대하여 생각보자. 이미지 파일을 읽어 pickle 형태의 파일로 저장할 때 주로 정수 값으로 저장한다. training 할 때, 정수값을 $-1 \sim 1$ 값으로 변환하여 사용한다. 변환 과정을 거치는 것이 비효율적이지만, 저장 단계에서 float 값으로 저장하지 않는 이유는 파일 사이즈 때문이다.

type	file size	
uint8	1	
float16	2배	
float32	4배	
float96	8배	img.astype('uint8') 와 같이 명시적으로 하지 않고, 단순히 img/255.0 으로만 변환하여 저장하면 float의 default type

표 3: data type별 file size

- TFRecord는 serial하게 data를 저장하기 때문에 random access에는 적합하지 않고 대용량 데이터 스트리밍에 적합함. 대용량 data는 memory에 다 올릴 수 없기 때문에 파일 형태로 training 중간에 반복적으로 읽어와야 하기 때문이다.

- data(image)를 binary로 저장하여 효율적인 읽기가 가능하다.
- 이제 TFRecord 파일을 만드는 방법에 대하여 알아보자.
- TFRecordWriter를 만든 후, example을 만들어서 writer에 쌓아주면 된다. example을 만드는 과정이 조금 복잡하다.
- dict 형의 feature를 통해서 example을 만드는데, 몇개의 보조 함수가 필요하다.

```
writer = tf.python_io.TFRecordWriter('apple.tfrecords')
for ...
    example = tf.train.Example(features=tf.train.Features(feature={...}))
    writer.write(example.SerializeToString())
writer.close()
```

- 보조 함수 _bytes_feature, _int64_feature를 이용하여 feature의 value를 만든다.
- feature를 만들 때, 정수는 _int64_feature, 바이너리 data는 _bytes_feature를 이용하여 만든다.
- string은 _bytes_feature(tf.compat.as_bytes('xxx'))를 사용하면 된다.
- float은 FloatList를 이용하면 된다.

```
def _int64_feature(value):
    """Wrapper for inserting int64 features into Example proto."""
    if not isinstance(value, list): value = [value]
    return tf.train.Feature(int64_list=tf.train.Int64List(value=value))

def _bytes_feature(value):
    """Wrapper for inserting bytes features into Example proto."""
    return tf.train.Feature(bytes_list=tf.train.BytesList(value=[value]))

...
with tf.gfile.FastGFile('apple001.jpg', 'rb') as f:
    img = f.read() # b'\xff\xd8\xff\xe0\x00\x10JFIF\x00\x01\x01\x00\x00'

example = tf.train.Example(features=tf.train.Features(feature={
    'file_name': _bytes_feature(tf.compat.as_bytes('apple001.jpg')),
    'image': _bytes_feature((img)),
    'size': _int64_feature(16)
}))
```

- 저장된 tfrecords 파일로 부터 data를 읽는 방법은 2가지가 있다.
- tf.parse_single_example
- tf.python_io.tf_record_iterator

♠ Tensorflow 기타

- `tf.Variable(<initial-value>, name=<optional-name>)`

```
import tensorflow as tf
tf.reset_default_graph()
W2 = tf.Variable(tf.random_normal([2, 2], stddev=0.02), name="W2")
W3 = tf.random_normal([2, 2], mean=0, stddev=1, name='W3')

sess = tf.Session()

# W3 는 W3.eval(sess)로는 안됨. 또한 sess.run(tf.global_variables_initializer())도 필요없음
print(sess.run(W3)) # [[-0.722506 -1.3118668] [-0.32936007 -1.6726391]]
print(sess.run(W3)) # [[ 1.1486942 0.5345976] [-0.7241453 -0.09306274]] 다른값
```

```
# W2 는 sess.run(tf.global_variables_initializer()) 이 반드시 필요함.
sess.run(tf.global_variables_initializer())
```

```
print(sess.run(W2)) # [[-0.01013633 -0.01422151] [ 0.0119183 -0.04426296]]
print(W2.eval(sess)) # [[-0.01013633 -0.01422151] [ 0.0119183 -0.04426296]] 같은값
```

- `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`

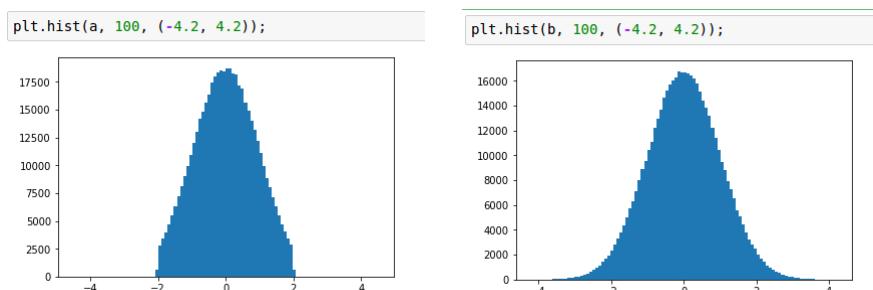


그림 19: Normal 비교

- `X.get_shape().as_list()`
- `tf.size(X)`는 tensor X의 전체 원소 개수.
- class 내에서 tensorflow variable이나 layer를 name scope 없이 만들어, tensorflow는 global하게 변수를 관리한다. 따라서 class object를 2개 만들면(즉, 함수를 2번 호출하면) error가 발생한다. 다음 3가지 방법을 생각해 볼 수 있다.

- object마다 name scope를 다르게 주는 방법: 이렇게 하면, object마다 각각의 variable을 만든다.

```
with tf.variable_scope(self.variable_scope):
    self.params={}
    self.params['embedding'] = tf.get_variable("embedding",
                                                initializer=0.01*np.random.randn(self.vocab_dim,
                                                self.wordvec_dim).astype(np.float32),dtype = tf.float32)
    self.params['Wp'] = tf.get_variable("Wp", shape=[input_dim,hidden_dim],
                                       initializer=tf.constant_initializer(np.random.randn(input_dim,hidden_dim)/np.sqrt(input_dim)))
```

- scope에 reuse=tf.AUTO_REUSE를 주는 방법: 이렇게 하면, name scope가 같은 object가 variable을 공유한다.

```
with tf.variable_scope(self.variable_scope,reuse=tf.AUTO_REUSE):
    self.params={}
    self.params['embedding'] = tf.get_variable("embedding",
                                                initializer=0.01*np.random.randn(self.vocab_dim,
                                                self.wordvec_dim).astype(np.float32),dtype = tf.float32)
    self.params['Wp'] = tf.get_variable("Wp", shape=[input_dim,hidden_dim],
                                       initializer=tf.constant_initializer(np.random.randn(input_dim,hidden_dim)/np.sqrt(input_dim)))
```

- scope.reuse_variables() 를 호출하는 쪽에서 사용.

```
with tf.variable_scope("") as scope:
    z = discriminator_WGAN(x)
    scope.reuse_variables()
    z2 = discriminator_WGAN(x2)
```

- tf.trainable_variables(), tf.global_variables(), tf.all_variables()
$$\begin{aligned} & \text{tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES)} \\ & \text{tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope='some_scope')} \\ & \text{for v in tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope='discriminator'): print(v.name)} \end{aligned}$$

```
t_vars = tf.trainable_variables()
d_vars = [var for var in t_vars if 'd_' in var.name]
g_vars = [var for var in t_vars if 'g_' in var.name]
```

```
t_vars = tf.trainable_variables()
d_vars = [var for var in t_vars if var.name.startswith("discriminator")]
g_vars = [var for var in t_vars if var.name.startswith("generator")]
```

```
x = tf.placeholder(tf.float32, [None,284])
with tf.variable_scope("hccho"):
    fc1 = tf.layers.dense(x,units=1024, activation = tf.nn.relu,name='fc1')
    fc2 = tf.layers.dense(fc1,units=10, activation = None,name='fc2')
    out = tf.nn.softmax(fc2)

image = np.random.randn(5,284)
```

```

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    out_ = sess.run(out, feed_dict={x: image})

var_list = tf.trainable_variables()
var_list2 = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES)
print(var_list)
print(var_list2)

```

- graph.get_tensor_by_name(name='L1/kernel:0')

```

L1 = tf.layers.dense(X, units=1, activation = None, name='L1')
...
graph = tf.get_default_graph()
print(sess.run(graph.get_tensor_by_name(name='L1/kernel:0')))

```

- checkpoint에서 variable 읽기. 디렉토리를 넘겨주면 가장 최근의 checkpoint에서 읽어오고, 파일까지 명시하면 그 파일의 variable들을 읽어온다.

```

var_list = checkpoint_utils.list_variables(checkpoint_dir)
sess = tf.Session()
for v in var_list:
    print(v) # tuple(variable name, [shape])
    vv = checkpoint_utils.load_variable(checkpoint_dir, v[0])
    print(vv) # values

```

- 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')

```

- initialization: tf.contrib.layers.xavier_initializer(), tf.initializers.random_normal(), tf.initializers.zeros() 등이 가능하다.

```

w_init = tf.contrib.layers.variance_scaling_initializer()
b_init = tf.constant_initializer(0.)

w0 = tf.get_variable('w0', [100, 200], initializer=w_init)
b0 = tf.get_variable('b0', [200], initializer=b_init)

```

- tf.gradients

```

x = tf.Variable([1.0], name="x")
y = tf.Variable([1.0], name="y")
g = tf.gradients([3*x+10*y, 2*x+100*y], [x,y])

==> g: 5, 110

```

initializer	dence/conv	
<code>glorot_uniform_initializer = xavier_initializer()</code>	dense	uniform $[-a, a]$, $a = \sqrt{\frac{6}{FanIn+FanOut}}$
<code>glorot_uniform_initializer = xavier_initializer()</code>	conv kernel weight (k, k, C, C')	uniform $[-a, a]$, $a = \sqrt{\frac{6}{k^2(C+C')}}$
<code>glorot_normal_initializer = xavier_initializer(False)</code>	dense	$N(0, \sigma^2)$, $\sigma = \sqrt{\frac{2}{FanIn+FanOut}}$
<code>glorot_normal_initializer = xavier_initializer(False)</code>	conv kernel weight (k, k, C, C')	$N(0, \sigma^2)$, $\sigma = \sqrt{\frac{2}{k^2(C+C')}}$

표 4: tensorflow initializer

5.2 Network 설계

♠ Network

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

- `__init__`: weights 생성. 생성된 weight는 `self.params` dict에 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): foward, 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. kernel weight의 초기값은 `kernel_initializer`로 주면되고, 초기값을 주지 않았을 때 default는 $[-a, a]$ uniform distribution(=glorot uniform initializer=Xavier uniform initializer)인데, a 는 다음과 같이 계산된다. kernel size = k , input channel = C , output channel = C' 라면, kernel weight의 shape = (k, k, C, C') ,

$$a = \sqrt{\frac{6}{k^2(C + C')}}.$$

```

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=(lambda x:
                           tf.nn.leaky_relu(x,0.01)),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

```

5.3 Advanced Tensorflow

♠ Estimator

Estimator는 만들어진 모델을 training, evaluate(test), predict 할 수 있게 해주는 API를 제공해 준다. 모델은

tensorflow에서 제공하는 모델(LinearClassifier, LinearRegressor, DNNClassifier, DNNRegressor)도 있고, 사용자가 직접 만들 수도 있다.

```
X = tf.estimator.Estimator(model_fn=my_model, params=my_params)
```

tf.estimator.Estimator에는 기본적으로 model_fn과 params가 전달되어야 한다. params는 dict형으로 model_fn에 넘겨지는 추가적인 parameter(예: learning rate)들이라고 보면된다. model_fn은 다음과 같은 형태로 정의되어야 한다.

```
def my_model(features, labels, mode, params):
    if (mode == tf.estimator.ModeKeys.TRAIN or
        mode == tf.estimator.ModeKeys.EVAL):
        loss = ...
    else:
        loss = None
    if mode == tf.estimator.ModeKeys.TRAIN:
        train_op = ...
    else:
        train_op = None
    if mode == tf.estimator.ModeKeys.PREDICT:
        predictions = ...
    else:
        predictions = None

    return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions, loss=loss, train_op=train_op)
```

- model_fn은 tf.estimator.EstimatorSpec를 return해야 한다. mode('train','eval','infer')에 따라 EstimatorSpec에 넘어가는 argument가 서로 다르다.

```
def hccho_model(features, labels, mode, params):
    inputs = tf.feature_column.input_layer(features, params['feature_columns'])

    # model
    L1 = tf.layers.dense(inputs, units=5, activation = tf.nn.relu, name='L1')
    logits = tf.layers.dense(L1, units=1, activation = None, name='L2')

    # prediction
    if mode == tf.estimator.ModeKeys.PREDICT:
        predictions = {'logits': logits}
        return tf.estimator.EstimatorSpec(mode, predictions=predictions)

    # Compute loss. loss 을 tf.estimator.ModeKeys.PREDICT 보다 앞쪽에 정의하면 predict 모드에서 error 발생
    loss = tf.reduce_mean(tf.square(logits - labels))

    # Compute evaluation metrics.
    if mode == tf.estimator.ModeKeys.EVAL:
        accuracy = tf.metrics.mean_absolute_error(labels=labels, predictions=logits)
        metrics = {'xxxx': accuracy}
        return tf.estimator.EstimatorSpec(mode, loss=loss, eval_metric_ops=metrics)
```

```

optimizer = tf.train.AdagradOptimizer(learning_rate=0.1)
train_op = optimizer.minimize(loss, global_step=tf.train.get_global_step())
return tf.estimator.EstimatorSpec(mode, loss=loss, train_op=train_op)

tf.logging.set_verbosity(tf.logging.INFO)
A = np.array([[73., 80., 75.], [93., 88., 93.], [89., 91., 90.], [96., 98., 100.], [73., 66., 70.]])
B = np.array([[152.], [185.], [180.], [196.], [142.]])  

input_fn_train = tf.estimator.inputs.numpy_input_fn(
    x = {"x":np.array(A,dtype=np.float32)},
    y = np.array(B,dtype=np.float32),
    num_epochs=10000,batch_size=2,shuffle=True
)  

input_fn_eval = tf.estimator.inputs.numpy_input_fn(
    x = {"x":np.array(A,dtype=np.float32)},
    y = np.array(B,dtype=np.float32),
    num_epochs=1, batch_size=2, shuffle=False
)
column_x = tf.feature_column.numeric_column("x",shape=(3,),dtype=tf.float32)
my_feature_columns = [column_x]
params={'feature_columns': my_feature_columns,'hidden_units': [10, 10], 'n_classes': 3 }

classifier = tf.estimator.Estimator(model_fn=hccho_model,model_dir =
'D:\\hccho\\RNN\\seq2seq\\Estimator-ckpt',params=params)
classifier.train( input_fn=input_fn_train,steps=1000)  

# evaluation
eval_result = classifier.evaluate(input_fn = input_fn_eval,steps=10)
print('\nTest set loss: {loss:.3f}, xxxx: {xxxx:.3f}\n'.format(**eval_result))  

# predict
A2 = np.array([[73., 80., 75.], [73., 66., 70.]])
input_fn_predict = tf.estimator.inputs.numpy_input_fn(x = {"x":A2},batch_size=len(A2),shuffle=False)
predictions = classifier.predict(input_fn=input_fn_predict) # user defined 0|function 0| |
    면, lambda function 으로 넘기면 안됨
print(list(predictions))

```

- `tf.train.LoggingTensorHook`를 통해 실행과정에서 추가적인 information을 출력할 수 있다.

```

def my_format(values): # values: dict
    res = []
    for v in values:
        res.append('{}={}'.format(v,values[v]))
    return '\n'.join(res)
def hccho_model(features, labels, mode, params):
    ...

```

```

logging_hook = tf.train.LoggingTensorHook({"loss" : loss, "accuracy" :
    tf.reduce_mean(tf.abs(logits-labels))}, every_n_iter=200)
logging_hook2 = tf.train.LoggingTensorHook({"my logits" : -logits, "my labels": labels},
    every_n_iter=200)

# dict 형으로첫번째 argument 를만들어넘기면 , fotmatter 함수로넘겨진 my_format 함수
# argument 로dict 형이넘어간다 .
logging_hook3 = tf.train.LoggingTensorHook({"xxxx":-logits, "yyy":labels},
    every_n_iter=200,formatter=my_format)

return tf.estimator.EstimatorSpec(mode, loss=loss, train_op=train_op,training_hooks =
[logging_hook,logging_hook2,logging_hook3])

```

- `tf.train.FeedFnHook`를 통해, input data feeding을 제어할 수 있다. 2개의 함수가 필요. `input_fn_train`은 기준과 동일하고 `feed_fn`을 정의하여 `classifier.train`의 hooks에 넘겨주어야 한다.
 - `input_fn_train`은 placeholder를 선언하고, `feed_fn`은 tensor 이름을 이용하여 값을 전달한다.
-

```

def input_fn_train():
    inp = tf.placeholder(tf.float32, shape=[None, 28*28], name='x')
    output = tf.placeholder(tf.int64, shape=[None, 10], name='y')
    #return {'x': inp,'y': output}, None # 두번째return targets 은여기서는사용하지않으므로None
    return {'my_x': inp}, output # targets(model-fn 에서labels)를 사용하려면...

def feed_fn():
    batch_x, batch_y = mnist.train.next_batch(batch_size)
    return {'x':batch_x, 'y':batch_y }

...
classifier.train( input_fn=input_fn_train,hooks=[tf.train.FeedFnHook(feed_fn)],steps=10000)

```

6 Convolution Neural Network

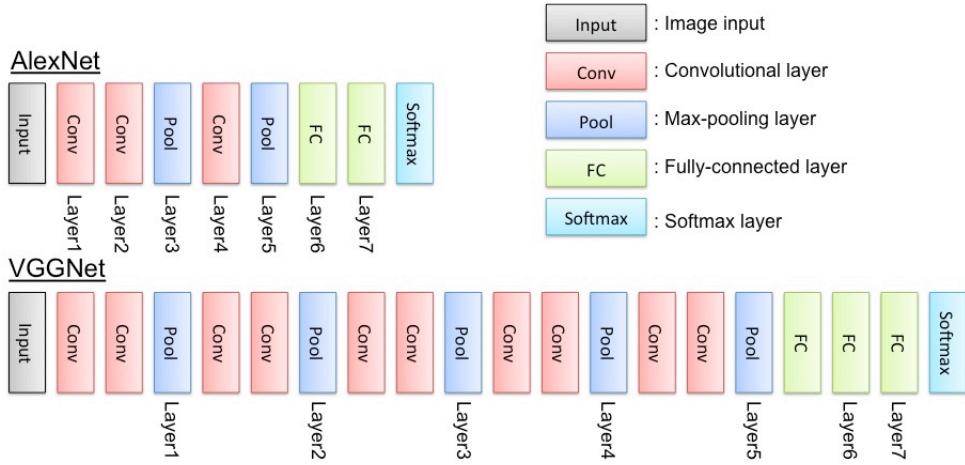


그림 20: CNN Architecture. Convolution Layer는 ReLU를 포함하고 있음.

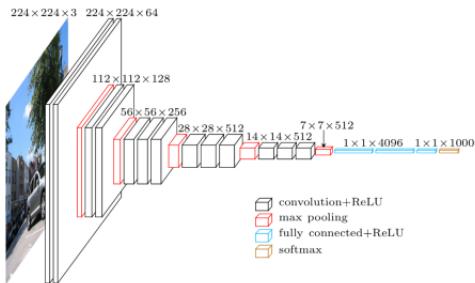


그림 21: VGG16 Architecture. GoogLeNet보다 성능은 떨어지지만, 구성이 간단하여 응용하기 좋은 장점이 있음.
단점으로는 메모리 사용과 연산이 많이 필요함

6.1 im2col

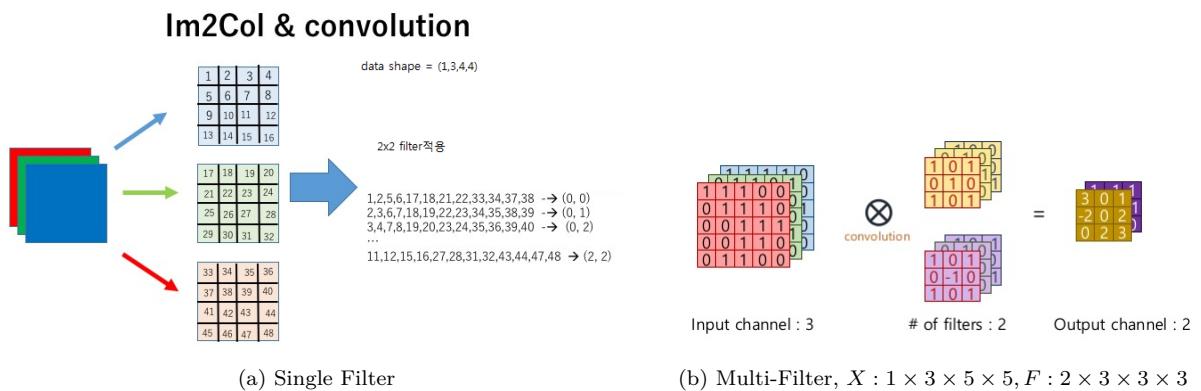


그림 22: Im2Col

- 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.

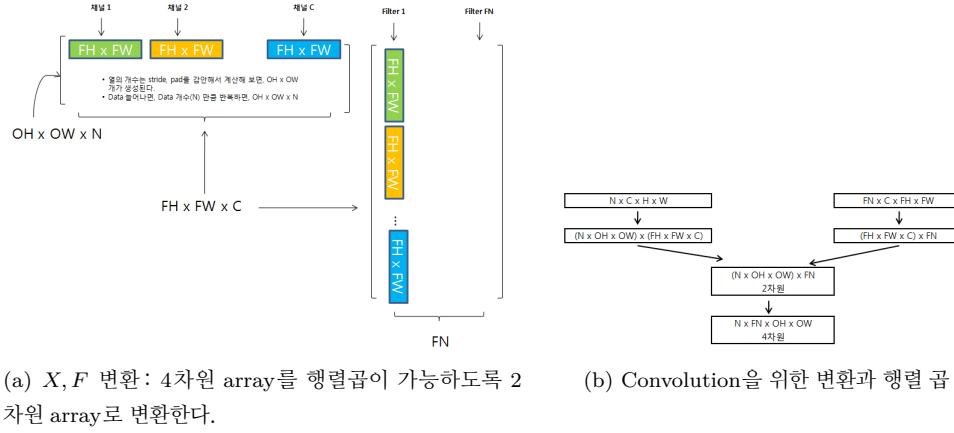


그림 23: Convolution

♠ 그림 (24)에 대한 설명

- $N = 1, C = 3, H \times W = 7 \times 7$ image, data shape = $(1, 3, 7, 7)$. 이 Data를 주어진 필터와 convolution하기 위해서는 계산 목적으로 Data를 im2col 변환하는 것이 필요하다.

- $FN = 1, FH = 5, FW = 5, \text{stride} = 1, \text{pad} = 0$ 인 필터를 적용하기 위한 im2col 변환을 생각해 보자.

$$OH = \frac{H + 2 \times \text{pad} - FH}{\text{stride}} + 1 = 3, OW = \frac{W + 2 \times \text{pad} - FW}{\text{stride}} + 1 = 3$$

- im2col 변환은 필터 5×5 만큼의 Data를 행으로 펼쳐 첫번째 행을 생성한다. 2번째 행은 window를 이동시키 5×5 Data를 행으로 펼친다. 이렇게 새로운 Data를 생성하면, $9 (= 3 \times 3 = OH \times OW)$ 만큼의 행이 만들어 진다.

- 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.

- 생성된 im2col 결과를 다른 관점에서 볼 수도 있다. im2col 열(숫자 9개)을 순서대로 보면, 입력 Data를 3×3 window를 이동시키면서 생성한 것으로 볼 수도 있다.

- 이제, Back Propagation에서 dcol로 부터 입력 Data에 대한 gradient를 계산해 보자.

- [그림 24]를 보면 im2col에서 22가 나타나는 위치의 dcol의 값을 합한 것이 22에 대한 gradient가 된다.

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

- 참고로 Pooling에서의 Output image의 크기는 다음과 같다.

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

그림 24: 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.

♠ tensorflow conv2d, conv2d_transpose

- tensorflow의 `tf.layers.conv2d`에서 `filters`는 output 채널 개수이고, `kernel_size`는 필터의 개수이다.

```
conv1_1 = tf.layers.conv2d(x, filters=32, padding='VALID', kernel_size=5, strides=1,  
                         activation=lambda x: tf.nn.leaky_relu(x,0.01), name = 'conv1_1')
```

- Tensorflow Padding: ‘valid’, ‘same’

```
padding = 'VALID'

out_height = ceil(float(in_height - filter_height + 1) / float(strides[1]))
out_width = ceil(float(in_width - filter_width + 1) / float(strides[2]))


padding = 'SAME'

out_height = ceil(float(in_height) / float(strides[1]))
out_width = ceil(float(in_width) / float(strides[2]))
```

- tensorflow `tf.layers.conv2d_transpose`은 `tf.layers.conv2d`에서 `in, out`이 서로 바뀌는데, `tf.layers.conv2d`에서의 조건을 만족하는 최소의 정수라고 보면 된다.

♠ tensorflow conv1d

- conv1d은 3 dim tensor가 input으로 들어간다.

```
x = tf.constant(np.array([[0,1,2,3,4,5,6,7,8,9],[ 0,2,4,6,8,10,12,14,16,18]]), tf.float32)
x = tf.expand_dims(x,2) # shape: (2,10,1)
y = tf.layers.conv1d(x,filters=3,kernel_size=2,strides=1,use_bias=False) # shape: (2,9,3)
```

♠ MNIST Classification Accuracy

Model	설명	Result
FC-1 layer(logistic regression)	100 epoch	GD: 89.54% Adam: 92.8 % weight를 $N(0, 0.02)$ 정도로 잘 초기화 하는 것도 중요하다 ⁷ .
FC-3 layer	100 epoch(256,256,10)	98% 5 layer로 늘이면 89.57%
CNN 2-layer, FC 2 layer	15 epoch, BN	98.85
CNN 3-layer, FC 2 layer	Max-Pooling, Dropout	99.3 BN 보다 Pooling-Dropout 조합이 결과적으로 더 좋다.

표 5: MNIST classification accuracy 비교

⁷간단한 2 layer XOR 모델도 random normal로 잘 초기화해야만 풀린다.

6.2 VGG의 이해

VGG는 ILSVRC 2014년에 2등(오류율 7.3%, 1등은 GoogLeNet)한 모델이다⁸. Tensorflow VGG16 and VGG19⁹를 분석하여 VGG모델을 이해해 보자. VGG19는 VGG16보다 3,4,5번째 Convolution Layer 그룹에 Convolution Layer 가 1개씩 더 있다는 것만 다르다.

- training된 data(vgg16.npy, vgg19.npy)를 다운 받을 수 있다¹⁰.
- ○] npy파일은 dict형 data를 가지고 있는 numpy array○이다.

```
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 list○이고, `data_dict['conv1_1'][0]`에는 filter값, `data_dict['conv1_1'][1]`에는 bias가 들어 있다. `data_dict['fc6']`도 len 2인 list○이고, `data_dict['fc6'][0]`에는 weight값, `data_dict['fc6'][1]`에는 bias가 들어 있다.

- ○] `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.

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

⁸<https://arxiv.org/abs/1409.1556>

⁹<https://github.com/machrisaa/tensorflow-vgg>

¹⁰각각 약 550M의 크기. 이 training data는 1,000개의 image category에 대하여 훈련된 결과를 가지고 있다. ImageNet의 image data category는 8만2천개가 넘는다.

```

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 ~ 1이 된다. 이런 방식으로 각각 변형된 이미지는 결합되어 $(N \times 224 \times 224 \times 3)$ 형태의 numpy array로 만들어진다.

`load_image(filename)`

- load_image를 통해 만들어진 data는 tf.placeholder를 통해서 Tensor로 변환된다. 그리고, VGG16, VGG19의 Forward Propagation을 수행하는 'build' 함수로 넘어간다.
- 'build' 함수는 넘어온 image data에 대하여, Normalization을 수행한다. 다시 255를 곱하고, VGG_MEAN = $(103.939, 116.779, 123.68)$ ¹¹ 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, 2차원 array로 변환되어 계산이 진행된다.

¹¹Blue,Green,Red순. 순서를 RGB로 하면, 코드가 간단해질 수 있는데, 왜 BGR순서로 해놓고 복잡하게 만드는지... 어떤 코드들은 RGB로 순서를 재배열 한 것도 있네.

VGG16	Input/Output	Filter/Weight FH, FW, 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 size가 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	$N \times 7 \times 7 \times 512$	25088×4096	$7 \times 7 \times 512 = 25088$, KeepProb = 0.5
FC 7 & Relu 7 & Dropout	$N \times 4096$	4096×4096	KeepProb = 0.5
FC 8 & Softmax	$N \times 4096$	4096×1000	
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)
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

```

- DropOut 구현: training에서는 포함되고, prediction에서는 포함되지 않는다.

```

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

```

- FC layer 구현

```
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.
- 나누어진 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 data 전체는 많은 메모리를 차지하기 때문에, 메모리가 충분하지 못할 경우를 대비하여 image의 순서를 shuffle하여 10개 정도로 나누어 저장.
- 이렇게 작업한 후, VGG모델은 244×224 크기의 이미지를 사용하는 모델임을 다시 알게됨. 다운 받은 64×64 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 Dataset¹²에는 다양한 music, image 등의 data가 link되어 있다.

- Caltech256 image: category가 256개(256_ObjectCategories.tar-1.15G¹³). image들의 size는 동일하지 않고 다양함(e.g. 499×278 , 200×150 , 1024×470 , ...). 전체 image 개수는 30,608개.
- Caltech101 image: category가 101개¹⁴. category별 image 개수 차이가 많이 난다(e.g. car-side는 모두 흑백 이미지. motorbikes는 798개). 전체 image 개수는 9,145개. int 형의 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.

¹²<http://deeplearning.net/datasets/>

¹³압축 풀면, 257개 category

¹⁴128M. 이것도 실제 102개 category

- category별로 보면, 40개 category는 50장 이하, 49개 category는 50 ~ 100장, 그 외 category는 100장, 200장, 400장, 800장 등.
- category별 이미지 개수의 편차가 심해, VGG training data로 사용하기에는 부족함.

♠ VGG Dataset

VGG Dataset¹⁵에서도 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 github¹⁶에 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.¹⁷

- 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.

¹⁵<http://www.robots.ox.ac.uk/~vgg/data/>

¹⁶<https://github.com/hccho2/CNN-VGG16-DogCat-Classification>

¹⁷<https://www.cs.toronto.edu/~kriz/cifar.html>

6.4 SqueezeNet

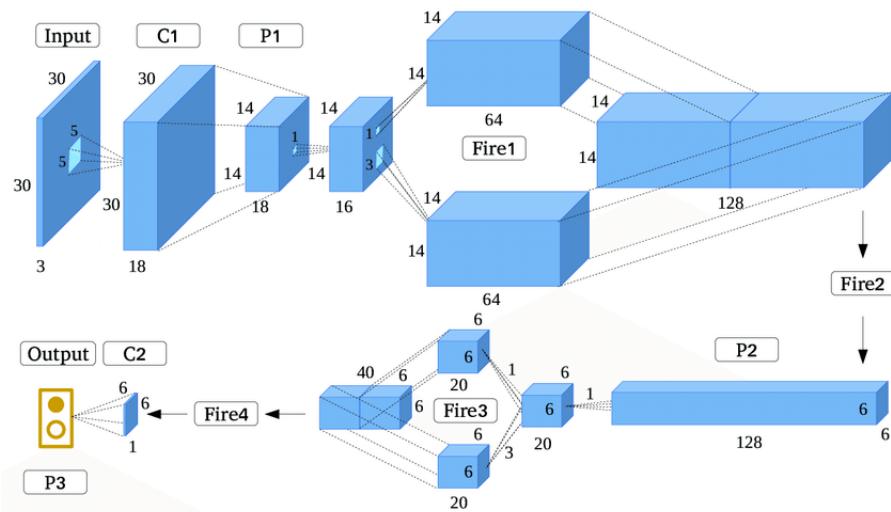


그림 25: SqueezeNet Structure

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.

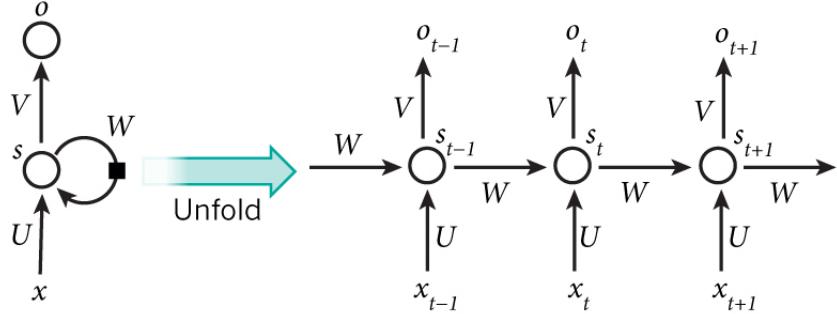


그림 26: RNN

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

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.¹⁸

$n = 7$ 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 = 1로 하면 구현이 쉽다.)

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

¹⁸출처: <http://iamtrask.github.io/2015/11/15-anyone-can-code-lstm/>

Multi-Layer RNN

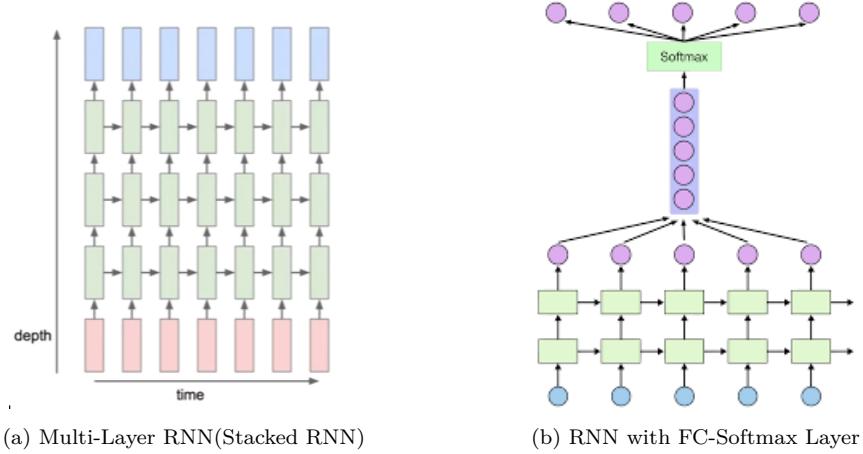


그림 27: Multi-RNN

- 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 σ 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, $Ax = \lambda x, A^2x = \lambda^2 x, \dots$ 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 Model¹⁹

```

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 = 1로 하면 구현이 쉽다. 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)

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

¹⁹<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^{o]} 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 Karpathy²⁰

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

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

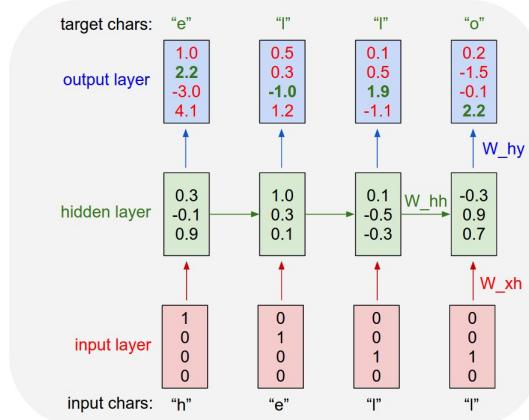


그림 28: An example RNN with 4-dimensional input and output layers, and a hidden layer of 3 units(source: Karpathy's Blog).

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.

²⁰<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) Networks²¹

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

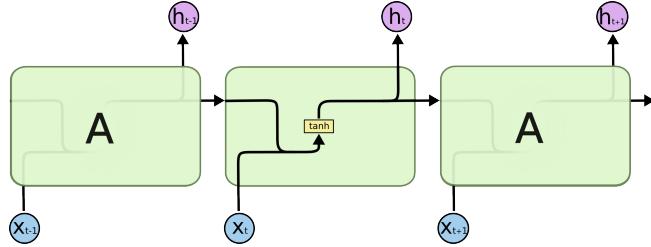


그림 29: The repeating module in a standard RNN contains a single layer(source: Colah's Blog).

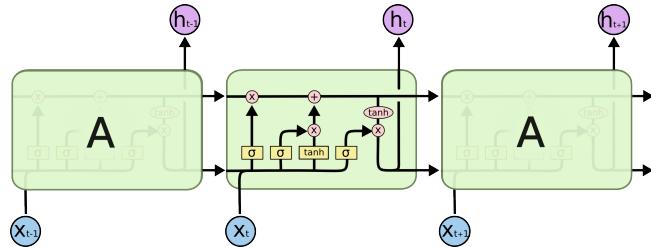


그림 30: The repeating module in an LSTM contains four interacting layers(source: Colah's Blog).

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

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

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

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

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

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

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

♠ 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	$N_y \times 1$

²¹<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 Blog²²의 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.

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

²²<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

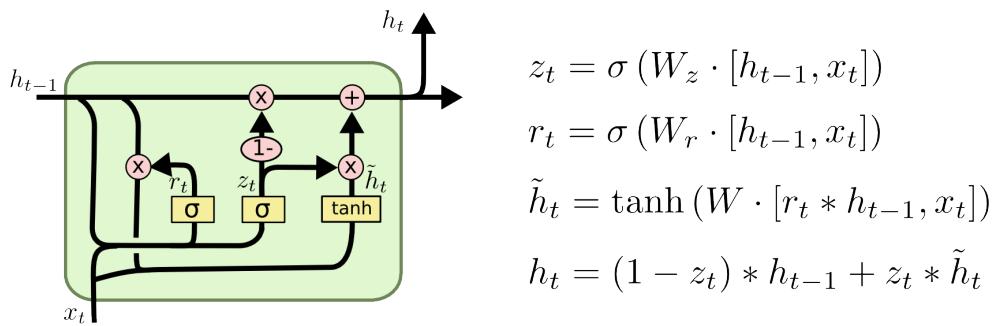


그림 31: GRU(source: Colah's Blog).

- tensorflow에서 GRU cell의 weight W_z, W_r 을 gates/kernel라는 이름으로 하나로 묶여있고 $(N_x + N_h) \times 2 * N_h$ 크기로 구현되어 있다.
- W 는 candidate/kernel이라는 이름으로 크기가 $(N_x + N_h) \times N_h$ 이다.

Bidirectional RNN

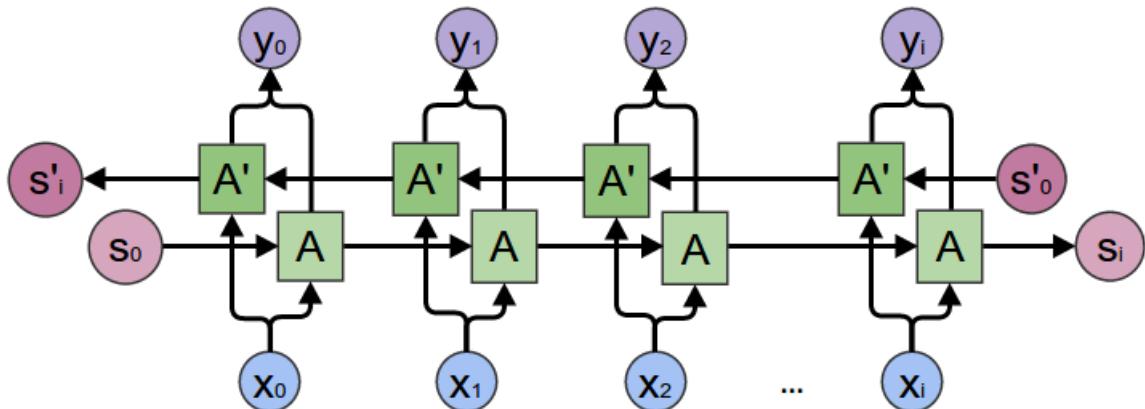


그림 32: GRU(Bidirectional RNN).

- tensorflow의 bidirectional rnn의 forward, backward output 모두 왼쪽에서 오른쪽으로 index가 부여된다. forward의 final state는 forward output의 마지막 index에 해당하는(제일 오른쪽) 값인 반면, backward final state는 backward output의 첫번째 index에 해당하는(제일 왼쪽) 값이다.
- `tf.nn.static_bidirectional_rnn`: input data가 list 형태로 들어간다.
- `tf.nn.bidirectional_dynamic_rnn`

```

encoder_inputs_embedded = tf.unstack(tf.nn.embedding_lookup(embeddings,
    self.encoder_inputs), axis=1)
encoder_outputs, encoder_output_state_fw, encoder_output_state_bw
    = tf.contrib.rnn.static_bidirectional_rnn(lstm_fw_cell, lstm_bw_cell, encoder_inputs_embedded
        , dtype=tf.float32)

```

```
-----  
encoder_inputs_embedded = tf.nn.embedding_lookup(embeddings, self.encoder_inputs)  
encoder_outputs, encoder_output_state  
= tf.nn.bidirectional_dynamic_rnn(lstm_fw_cell, lstm_bw_cell, encoder_inputs_embedded  
, dtype=tf.float32)
```

7.5 RNN with TensorFlow

- dynamic_rnn의 return 값 outputs의 마지막 값은 states와 같은 값이다. ↪ 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
```

- BasicRNNCell 의 kernel size는 $(\text{hidden_size} + \text{input_size}) \times \text{hidden_size}$.

```
# 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]  
  
mode = 1  
with tf.variable_scope('3_batches') as scope:  
    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  
  
    if mode == 0:  
        cell = rnn.BasicRNNCell(num_units=hidden_size)  
        #cell = rnn.BasicLSTMCell(num_units=hidden_size, state_is_tuple=True)  
    else:  
        cells = []  
        for _ in range(3):  
            #cell = tf.contrib.rnn.BasicRNNCell(num_units=hidden_dim)  
            cell = tf.contrib.rnn.BasicLSTMCell(num_units=hidden_size, state_is_tuple=True)  
            cells.append(cell)  
    cell = tf.contrib.rnn.MultiRNNCell(cells)  
  
    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_loss의 logits 입력은 activation function을 거치지 않은 것을 넣어주어야 한다. dynamic_rnn의 outputs은 activation function을 거친 값(좀 더 정확히 말하면, BasicLSTMCell에서 return되는 hidden state값은 activation function을 거쳐서 나온 값, 식(7)). 그래서 dynamic_rnn 이후 FC layer를 추가로 연결하는 것이 일반적이다. sequence_loss는 outputs에 softmax 함수를 취한 후, cross-entropy loss mean을 계산해 준다. targets은 one-hot-encoding으로 변환하지 않아야 한다.

```

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]]]

```

- `tf.contrib.rnn.OutputProjectionWrapper`: output에 FC layer를 추가한 cell을 만들어준다. Multi-RNN의 경우는 마지막 output에만 FC layer가 붙는다.

```

cell = tf.contrib.rnn.OutputProjectionWrapper(cell, 4) # output 에FC layer 를추가하여원하는size 로변
                                                환해준다.
outputs, _states = tf.nn.dynamic_rnn(cell,x_data,initial_state=initial_state,dtype=tf.float32)

```

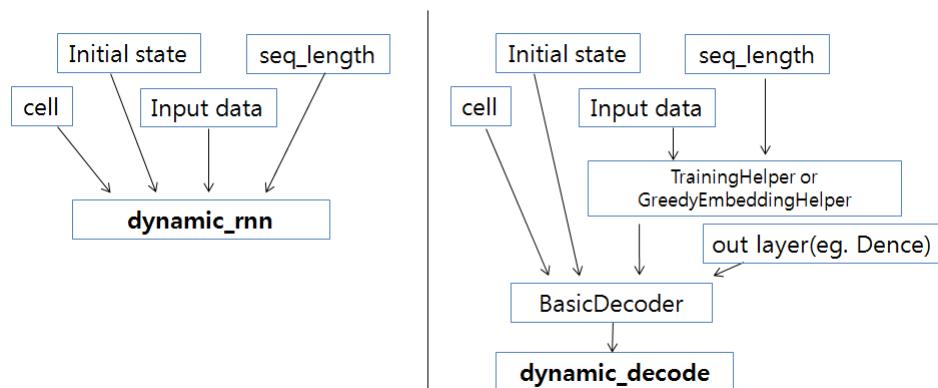


그림 33: `tf.nn.dynamic-rnn` vs `tf.contrib.seq2seq.dynamic-decoder`. inference mode에서는 `TrainingHelper` 대신 `GreedyEmbeddingHelper`를 사용하면 이전 단계의 output을 다음 단계의 input으로 넣어준다.

♠ tf.nn.raw_rnn

- tf.nn.raw_rnn은 RNN모델을 더욱 정밀하게 제어할 수 있게 해준다. 함수 loop_fn를 정의하여 RNN모델의 cell과 cell 사이의 값 전달 과정에서 정밀한 제어가 가능하게 해준다.

```
def loop_fn(time, cell_output, cell_state, loop_state):
    ...
    return (elements_finished, next_input, next_cell_state, emit_output, next_loop_state)

outputs_ta, final_state, final_loop_state = tf.nn.raw_rnn(cell, loop_fn)
```

- loop_fn는 앞 단계 cell의 결과인 cell_output과 cell_state를 받아 가공후 다음 cell에 넘길 수 있게 만들어 준다.
- loop_fn의 첫번째 argument time은 RNN의 time step을 나타내는 tensor이다. time=0²³ 일때는 첫번째 time=1인 RNN cell에 들어가기 직전이고, 그 외에는 time = 1 ~ seq_length까지를 나타낸다. 즉 time = i에서의 loop_fn은 time = i RNN cell의 결과를 입력받아, i 번째 output과 i + 1 번째 RNN cell에 들어갈 state 값을 만든다.
- loop_state, next_loop_state는 추가적인 필요 data(any additional information)를 넘기는 공간으로 활용하거나, None.

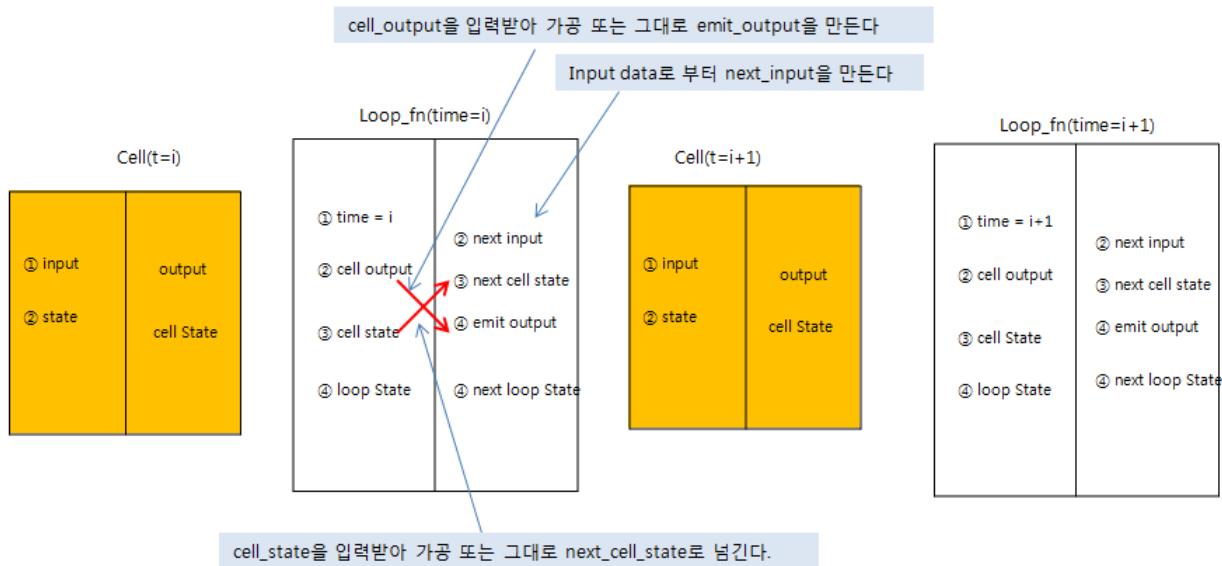


그림 34: raw-rnn

```
x_data = np.array([[0, 3, 1, 2, 4, 3], [1, 3, 1, 2, 3, 2], [2, 4, 0, 2, 4, 1]], dtype=np.int32)
init = np.arange(vocab_size*embedding_dim).reshape(vocab_size,-1)
embedding = tf.get_variable("embedding", initializer=init.astype(np.float32), dtype = tf.float32)

# time_major 이어야함. 그래서 transpose
inputs = tf.transpose(tf.nn.embedding_lookup(embedding, x_data),(1,0,2))

output_ta = tf.TensorArray(size=seq_length, dtype=tf.float32) # loop_state 를위하여...
```

²³time이 tensor이기 때문에, numeric값은 아니다.

```

def loop_fn(time, cell_output, cell_state, loop_state):
    # time = 0 -> time = 1 첫번째( cell) 에넘어갈 data 를만든다 .
    # time = 1 -> time = 2 에넘어갈 data 를만든다

    # 넘어온 cell_output 을가공하여새로운 output 인 emit_output 을만든다.
    emit_output = ...

    # 넘어온 cell_state 를가공하여 next_cell_state 를만든다.
    next_cell_state = ...

    # batch size 크기의sequence_length 에따라계속진행여부를판단한다 .
    elements_finished = (time >= sequence_length)

    # global 변수로정의되어있는 input_ta 로부터다음 data 를만든다 .
    next_input = tf.cond(tf.reduce_all(elements_finished),
                         lambda: tf.zeros([batch_size, embedding_dim], dtype=tf.float32),
                         lambda: inputs_ta.read(time))

    # next_loop_state
    next_loop_state = loop_state.write(time - 1, [some value])

    return (elements_finished, next_input, next_cell_state, emit_output, next_loop_state)

outputs_ta, final_state, final_loop_state = tf.nn.raw_rnn(cell, loop_fn2)
outputs = outputs_ta.stack()

```

- loop_fn을 구현하는데 있어, 그 내부를 2개의 함수로 나눌 수 있다. loop_fn_initial, loop_fn_transition.

```

def loop_fn(time, previous_output, previous_state, previous_loop_state):
    if previous_state is None: # time == 0
        return loop_fn_initial()
    else:
        return loop_fn_transition(time, previous_output, previous_state, previous_loop_state)
def loop_fn_initial():
    initial_elements_finished = (0 >= decoder_lengths) # all False at the initial step
    initial_input = eos_step_embedded
    initial_cell_state = encoder_final_state
    initial_cell_output = None
    initial_loop_state = None # we dont need to pass any additional information
    return (initial_elements_finished, initial_input, initial_cell_state,
            initial_cell_output, initial_loop_state)

def loop_fn_transition(time, previous_output, previous_state, previous_loop_state):
    def get_next_input():
        output_logits = tf.add(tf.matmul(previous_output, W), b)
        prediction = tf.argmax(output_logits, axis=1)
        next_input = tf.nn.embedding_lookup(embeddings, prediction)

```

```

    return next_input

# this operation produces boolean tensor of [batch_size]
# defining if corresponding sequence has ended
elements_finished = (time >= decoder_lengths)

finished = tf.reduce_all(elements_finished) # -> boolean scalar
input = tf.cond(finished, lambda: pad_step_embedded, get_next_input)
state = previous_state
output = previous_output
loop_state = None
return (elements_finished, input, state, output, loop_state)

```

7.6 Character-Level Vanilla RNN Model by Andrej Karpathy

Karpathy의 Minimal Character Model을 Multi-Layer RNN으로 확장해보자.

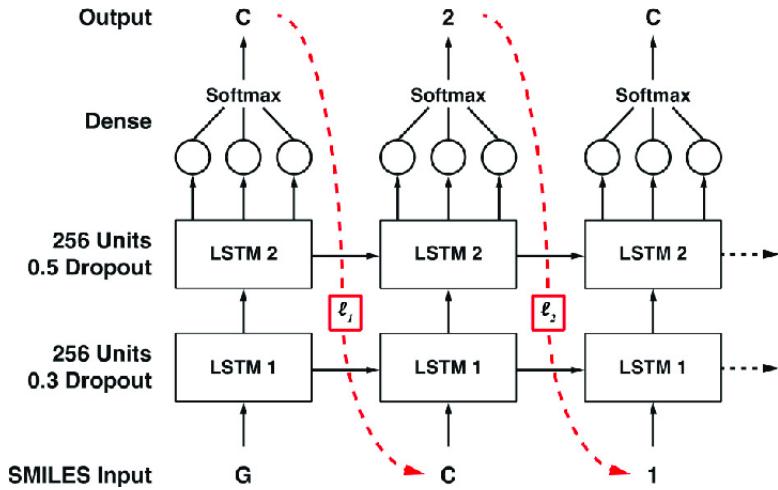


그림 35: Multi-Layer RNN

- Karpathy의 Lua 코드²⁴를 Tensorflow 형태로 구현한 코드²⁵를 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)

- MS COCO²⁶: 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 5

²⁴<https://github.com/karpathy/char-rnn>

²⁵<https://github.com/sherjilozair/char-rnn-tensorflow>

²⁶<http://mscoco.org>

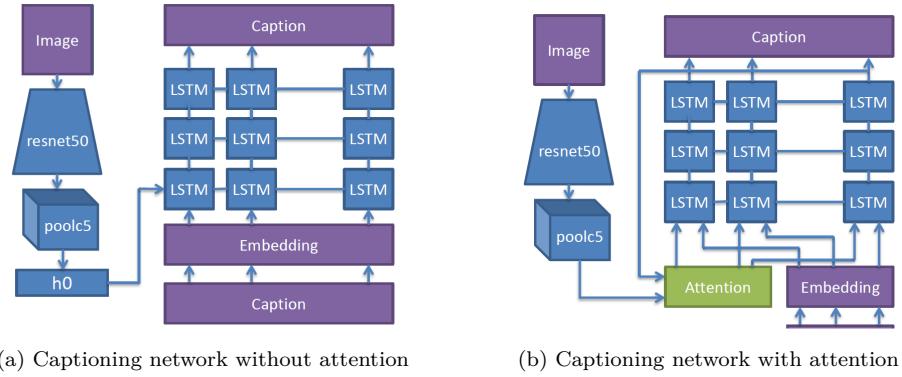


그림 36: Captioning RNN Models

개 정도의 caption이 주어져 있기 때문이다²⁷. caption에서 사용된 단어의 갯수는 1,004개이다.

- mini batch로 data를 뽑는 단계에서, 먼저 caption을 random하게 선택한 후, 그 caption의 image index를 이용하여 image를 뽑아내는 방식으로 처리한다.

```
def sample_coco_minibatch(data, batch_size=100, split='train'):
    split_size = data['%s_captions' % split].shape[0]
    mask = np.random.choice(split_size, batch_size)
    captions = data['%s_captions' % split][mask]
    image_idxs = data['%s_image_idxs' % split][mask]
    image_features = data['%s_features' % split][image_idxs]
    urls = data['%s_urls' % split][image_idxs]
    return captions, image_features, urls
```

- 그림(36a)의 h_0 는 이미지의 feature(F)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.

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

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

- Caption 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. Caption data는 시작과 끝이 <START>, <END>로 되어 있고, <NULL>을 붙여 길이가 T 로 동일하게 만들었다. <NULL>을 붙이지 않고, batch size를 1로 하여, 길이가 다른 data를 사용할 수도 있다(쉽게 하는 방법). 길이가 다른 batch data도 구현이 가능하다.

```
# 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
```

²⁷caption number 111236, 111487, 111555, 111591, 111849는 하나의 image인 13585를 설명하고 있다.



그림 37: 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 46에서 언급한 것 처럼, 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

- Guillaume Genthial blog²⁸
- Tensorflow Neural Machine Translation (seq2seq) Tutorial²⁹
- 그림 (38a), (39)의 encoder-decoder 모델(즉, 가장 기본적인 seq2seq 모델)에서는 encoder의 마지막 hidden state를 decoder의 initial state로 넘겨준다.

²⁸<https://guillaumegenthial.github.io/sequence-to-sequence.html>

²⁹<https://www.tensorflow.org/tutorials/seq2seq>

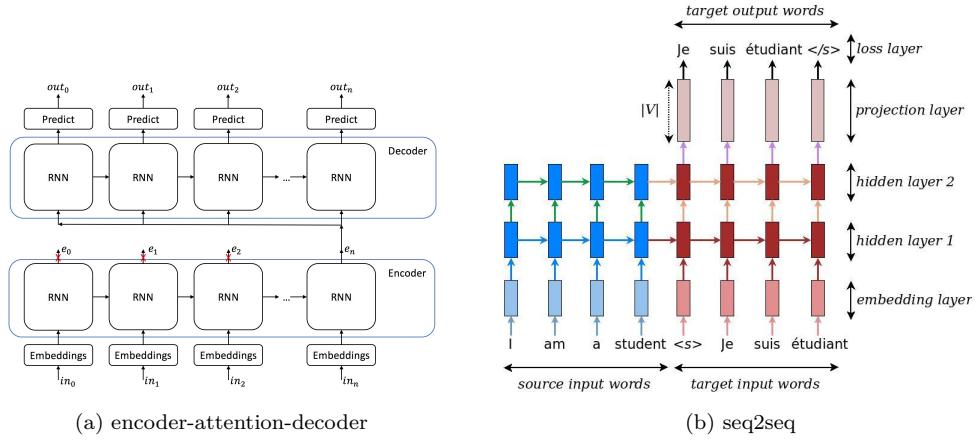


그림 38: Neural machine translation

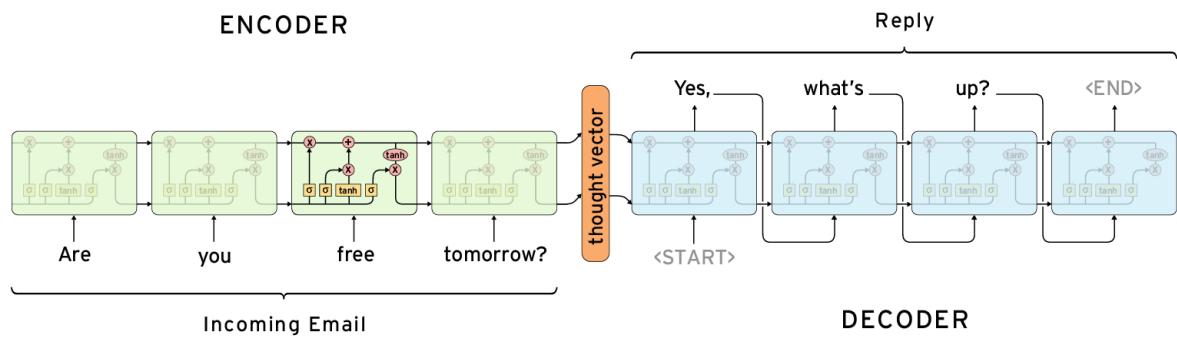


그림 39: Encoder-Decoder with Attention

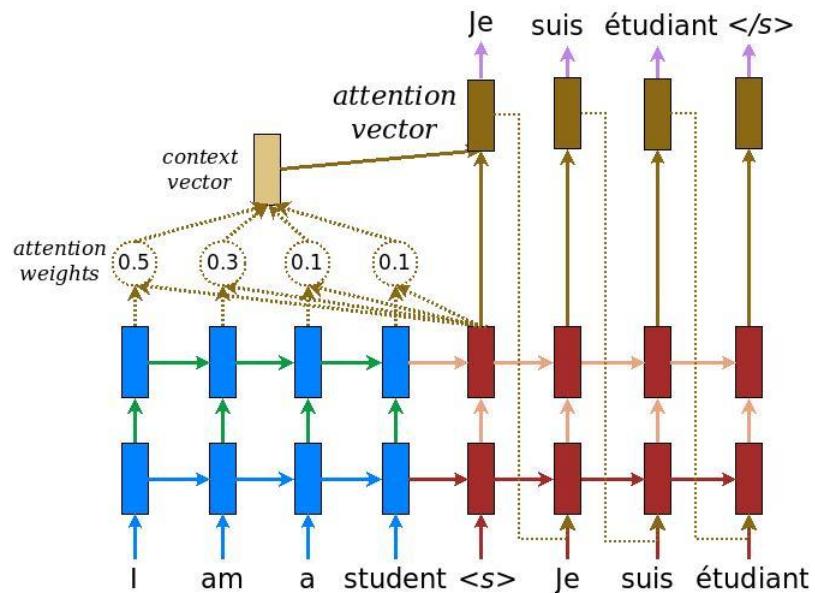


그림 40: Encoder-Decoder with Attention

- 기본적인 encoder-decoder 모델의 구현 예: ‘practical seq2seq’³⁰
- 그런데 encoder의 last hidden state만으로는 충분하지 못하다는 것이 문제인데, Attention이 그 해결책을 될

³⁰https://github.com/suriyadeepan/practical_seq2seq

수 있다.

- Attention은 encoder와 decoder 사이에서 decoder로 넘길 중요한 정보를 encoder에서 추출하여 넘기는 역할을 한다.
- seq2seq 모델을 training 할 때, decoder의 input으로 주어진 data를 넣어줄 수도 있고 (“Teacher Forcing”) decoder가 예측한 값을 넣어줄 수도 있다.
- “Teacher Forcing”, or maximum likelihood sampling, means using the real target outputs as each next input when training. The alternative is using the decoder’s own guess as the next input. Using teacher forcing may cause the network to converge faster, but when the trained network is exploited, it may exhibit instability.

You can observe outputs of teacher-forced networks that read with coherent grammar but wander far from the correct translation - you could think of it as having learned how to listen to the teacher’s instructions, without learning how to venture out on its own.

The solution to the teacher-forcing “problem” is known as Scheduled Sampling, which simply alternates between using the target values and predicted values when training. We will randomly choose to use teacher forcing with an if statement while training - sometimes we’ll feed use real target as the input (ignoring the decoder’s output), sometimes we’ll use the decoder’s output.

♠ Bahdanau Attention

- Dzmitry Bahdanau, Kyunghyun Cho, Yoshua Bengio, Neural Machine Translation by Jointly Learning to Align and Translate, 2014.³¹
- The Bahdanau paper uses a bidirectional RNN for the encoder. encoder의 정방향(\vec{h}_j), 역방향(\overleftarrow{h}_j) hidden state를 concatenation해서 ‘annotations’ h_j 를 생성한다.

$$h_j = \text{concat}(\vec{h}_j, \overleftarrow{h}_j), \quad j \in \{\text{encoder time step}\}$$

- Decoder의 time step i 에서 hidden state s_i 를 구해보자³². 논문에서는 GRU를 사용하고 있다.

$$e_{ij} \rightarrow \alpha_{ij}(\text{attention weight}) \rightarrow c_i(\text{context vector})$$

³¹<https://arxiv.org/abs/1409.0473>

³²encoder에서 hidden state를 h 로 했기 때문에 decoder의 hidden state를 s 로 표시

N	# of batch	y_i	decoder input
T_e	encoder sequence length	N_e	embed size
T_d	decoder sequence length	W_z^d	$N_e \times N_h$
N_h	# of hidden units	U_z^d	$N_h \times N_h$
N_a	# of align hidden units	C_z^d	$2N_h \times N_h$
W_q	$N_h \times N_a$	b_z^d	$1 \times N_h$
U_a	$2N_h \times N_a$	W_r^d	$N_e \times N_h$
b_a	$1 \times N_a$	U_r^d	$N_h \times N_h$
v_a	$1 \times N_a$	C_r^d	$2N_h \times N_h$
b_r^d	$1 \times N_h$	W^d	$N_e \times N_h$
U^d	$N_h \times N_h$	C^d	$2N_h \times N_h$
b^d	$1 \times N_h$	N_m	# of maxout hidden units
U_o	$N_h \times 2N_m$	V_o	$N_e \times 2N_m$
C_o	$2N_h \times 2N_m$	b_o	$2N_h \times 1$
N_v	decoder vocab size	W_o	$N_m \times N_v$
b_w	$1 \times N_v$		
W_s	$N_h \times N_h$	b_s	$1 \times N_h$

- 먼저 e_{ij} 를 구해보자. for each j in encoder time step,

$$\begin{aligned} s_0 &= \tanh(\overleftarrow{h}_1 W_s + b_s) \\ e_{ij} &= \tanh(s_{i-1} W_q + h_j U_a + b_a) v_a^T \leftarrow N \times 1 \end{aligned} \quad (10)$$

$$\begin{aligned} e_i &= [e_{i1}, \dots, e_{iT_e}] \\ \alpha_i &= \text{softmax}(e_i) = [\alpha_{i1}, \dots, \alpha_{iT_e}] \leftarrow \text{alignment} \\ c_i &= \sum_{j=1}^{T_e} \alpha_{ij} h_j \leftarrow N \times 2N_h \end{aligned} \quad (11)$$

- context vector c_i 로 부터 attention a_i 를 구해보자. 다음의 2가지 case는 tensorflow `tf.contrib.seq2seq.AttentionWrapper`에서 `attention_layer_size`의 값은 `None`으로 하거나 값(N_l)을 주거나 이다.

$$a_i = \begin{cases} c_i & \text{or} \\ [s_{i-1} | c_i] W_a & \text{where } W_a : 3N_h \times N_l \end{cases}$$

- 이렇게 계산한 attention a_i 로 부터 GRU모델³³을 이용하여 hidden state s_i 를 구해보자.

$$\begin{aligned} z_i^d &= \text{sigmoid}(y_i W_z^d + s_{i-1} U_z^d + a_i C_z^d + b_z^d) \\ r_i^d &= \text{sigmoid}(y_i W_r^d + s_{i-1} U_r^d + a_i C_r^d + b_r^d) \\ \tilde{s}_i &= \tanh(y_i W^d + (r_i^d * s_{i-1}) U^d + a_i C^d + b^d) \\ s_i &= (1 - z_i^d) * s_{i-1} + z_i^d * \tilde{s}_i \end{aligned}$$

- 위 식을 보면, GRU모델에서 attention a_i 에 Weight C_z^d, C_r^d, C^d 각각 곱해진 후, 더해져 있는 것을 알 수 있다.
- 구현의 입장에서 보면, hidden state s_{i-1}, a_i 를 뮤어주면 기존 GRU모델을 계속 사용할 수 있다. 즉,

$$s_{i-1} \leftarrow \text{concat}(s_{i-1}, a_i)$$

³³ a_i 가 추가된 GRU

- output 을 계산해보자. output \circ hidden state s_i 에 FC layer를 바로 적용하지 않고, 좀 더 복잡하게 계산하고 있다. \tilde{t}_i 를 구한 후, maxout을 통해, size를 줄인 후, FC layer를 적용한다.

$$\begin{aligned}\tilde{t}_i &= s_{i-1}U_o + y_iV_o + a_iC_o + b_o \\ t_i &= \text{maxout}(\tilde{t}_i, N_m) \leftarrow N \times N_m \\ L_i &= t_iW_o + b_w \leftarrow \text{logit} \\ \hat{y}_i &= \text{argmax}(L_i)\end{aligned}$$

- Attention Visualization: weight α_{ij} 를 visualization하면 그림 (41)과 같다.

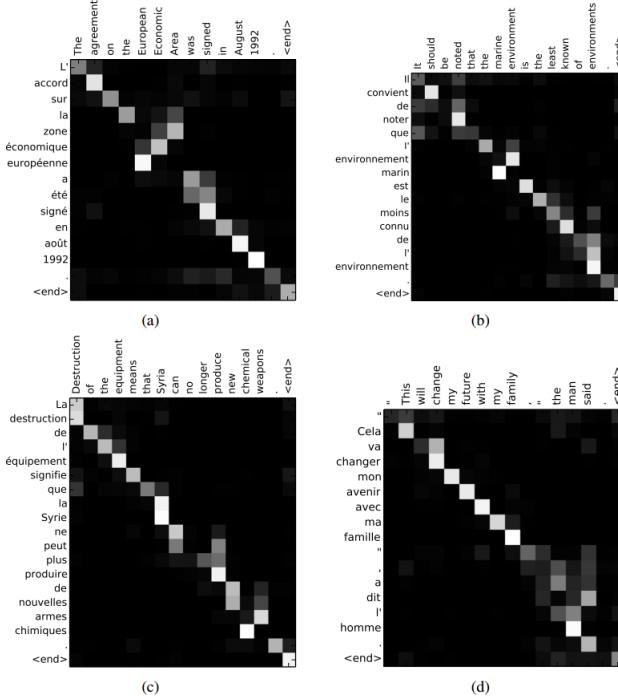


그림 41: Attention Visualization

♠ Luong Attention

- Minh-Thang Luong, Hieu Pham, Christopher D. Manning, Effective Approaches to Attention-based Neural Machine Translation, 2015³⁴.
- 식(10)에서 e_{ij} 를 $\text{score}(s_{i-1}, h_j)$ 로 칭하고, Luong Attention은 \circ score 를 계산하는 다른 방법을 제안하고 있다.

$$\text{score}(s_{i-1}, h_j) = \begin{cases} s_{i-1}^T h_j & \text{dot} \\ s_{i-1}^T W_a h_j & \text{general} \\ v_a^T \tanh(W_a[s_{i-1}; h_j]) & \text{concat(Bahdanau)} \end{cases}$$

♠ Attention with Tensorflow

³⁴<https://arxiv.org/abs/1508.04025>

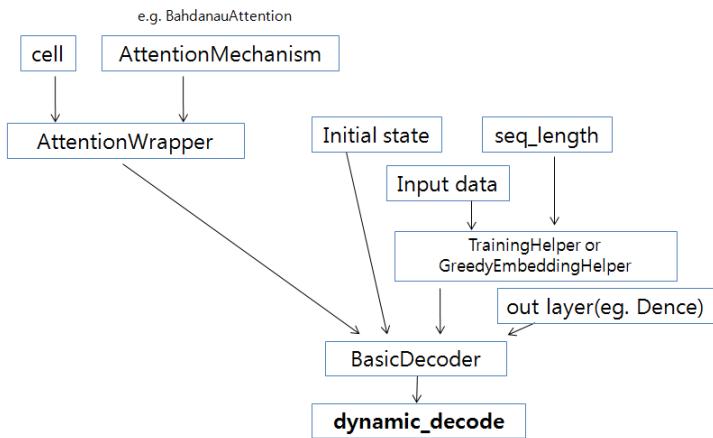


그림 42: 그림 33과 비교하여 보면, cell 위치에 AttentionWrapper가 새로운 cell이 되어 BasicDecoder에 들어간다. AttentionWrapper는 보통의 cell(eg. LSTMCell, GRUCell)과 AttentionMechanism을 결합하여 만든다. AttentionWrapper도 RNNCell을 상속받아 구현된 class이다.

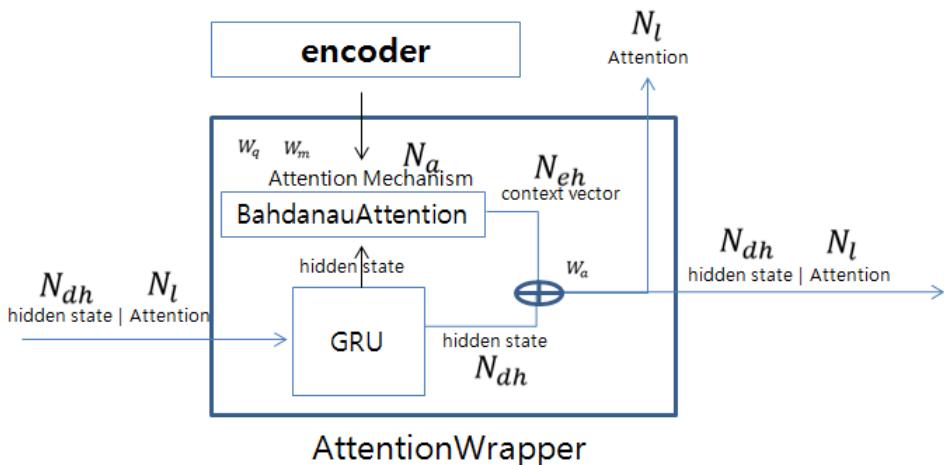


그림 43: Tensorflow AttentionWrapper: GRU cell과 BahdanauAttention은 각각 만들어 지지만 AttentionWrapper로 들어가 context vector를 만드는데 GRU cell의 hidden state가 사용된다. 다시 hiddent state와 context vector가 결합하여 attention을 만든다. 그 다음 단계로 hidden state와 attention o concatenation되어 AttentionWrapper의 hidden state가 된다.

N_{eh}	encoder hidden size	N_{dh}	decoder hidden size
N_a	attention depth (BahdanauAttention에 전달)	N_l	attention layer size (AttentionWrapper에 전달)
$W_m,$	memory layer kernel weight($N_{eh} \times N_a$)	W_q	query layer kernel weight($N_{dh} \times N_a$)
$v_a,$	attention v($1 \times N_a$)	W_a	$(N_{eh} + N_{dh}) \times N_l$

- 식(10)가 변형되어 적용된다. 논문에서는 s_{i-1} 의 weight 계산에 사용되지만, Tensorflow에서는 s_i 를 사용한다.

$$e_{ij} = \begin{cases} \tanh(s_i W_q + h_j W_m) v_a^T, & \text{if normalize=False (default)} \\ \frac{g}{\|v_a\|} \tanh(s_i W_q + h_j W_m + b_a) v_a^T, g \in \mathbb{R} & \text{if normalize=True} \end{cases}$$

- tensorflow `tf.contrib.seq2seq.AttentionWrapper`에서 `attention_layer_size`의 값은 None으로 하거나

나 값(N_l)을 주거나 이다.

$$a_i = \begin{cases} c_i & (\text{if None}) \text{ or} \\ [s_i | c_i]W_a & (\text{if } N_l \text{ given}) \text{ where } W_a : (N_{eh} + N_{dh}) \times N_l \end{cases}$$

- attention a_i 와 원래 GRU cell의 hidden state가 concatenation되어 AttentionWrapper의 새로운 hidden state($N_l + N_{dh}$)가 된다.
- 예를 들어, cell이 BasicRNNCell이라면, kernel weight의 크기는 (input dim + $N_l + N_{dh}$, N_{dh})
- output은 a_i 가 된다(s_i 가 output이 아님에 주의하자). 여기에 FC layer를 붙혀 원하는 크기 (eg. vocab size)로 만들면 된다.
- BahdanauMonotonicAttention은 식(11)에서 softmax 대신 다른 방식의 함수를 적용하는 것으로 보면 된다.

♠ Transformer

- Vaswani et al, 2017. Attention Is All You Need³⁵
- sequence model에서는 encoder-decoder model의 주류를 이룬다. 그 중 성능면에서는 attention mechanism을 활용하는 모델이 뛰어나다. transformer모델은 rnn을 활용하지 않고 attention에만 기반을 둔 모델이다. rnn을 사용하지 않기 때문에 병행처리가 가능하다는 장점이 있다.

7.9 Text Data 다루기

♠ seq2seq 모델을 위한 Dataset

- 번역: <http://www.manythings.org/anki/>
- 발음-spelling: CMU Pronouncing Dictionary, <http://www.speech.cs.cmu.edu/cgi-bin/cmudict>

♠ Text Data

- raw data의 format에 따라 process_data()를 만들어, 3개의 파일을 만든다.
`metadata.pkl(dict), ind_input.npy(numpy array), ind_output.npy(numpy array)`
 - 단어의 출현 빈도에 따른 dict 형을 만들고, 상위 몇개의 Data만으로 vocabulary를 만든다.
 - 문장을 index로 변환하면서 vocabulary에 들어 있지 않은 단어는 unknown으로 대체한다.
 - metadata.pkl은 dict 형으로 key는 다음과 같다.`'w2idx'(dict), 'idx2w'(list), 'limit'(dict), 'freq_dist'`

```
metadata['freq_dist']
FreqDist({'you': 72476, 'i': 54630, 'the': 46988, ...})
limit = {'maxq' : 25, 'minq' : 2, 'maxa' : 25, 'mina' : 2}
```

- training할 때마다 위에서 만든 3개의 파일을 읽어 사용하면 된다.
- Language가 2개 일 경우는 metadata.pkl에 2가지 언어에 대한 정보를 더 저장하면 된다.

³⁵<https://arxiv.org/abs/1706.03762>

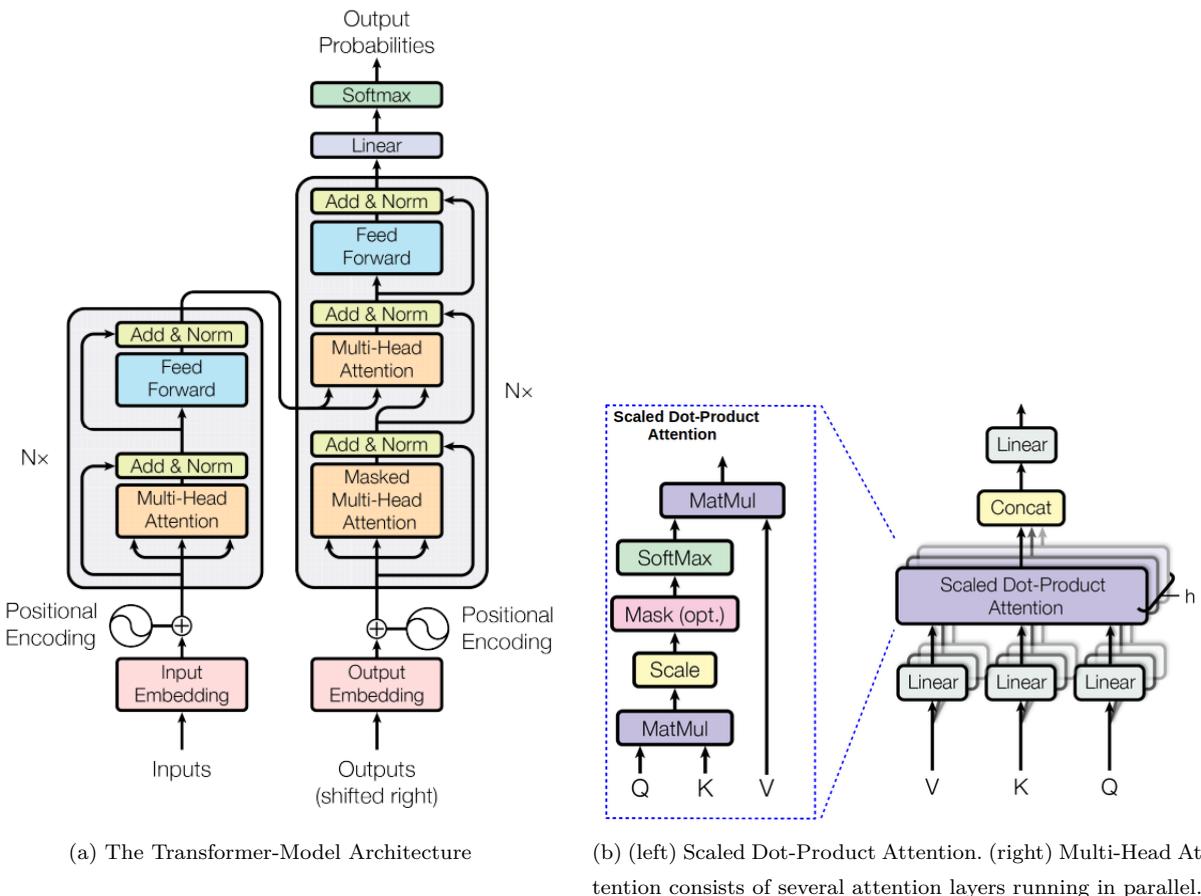


그림 44: The Transformer

8 Reinforcement Learning



그림 45: RL 대표 Research Group

'모두를 위한 RL강좌'³⁶를 기반으로 RL의 기초를 다져보자.

8.1 Q-Table, Q-Network

- 준비작업 : OpenAI Gym에서 제공하는 game환경을 설정한다. msys2, xming, atari설치, gym[atari] 등을 설치해야 한다³⁷.
- FrozenLake를 keyboard 입력을 받아 한번 해본 후, random으로 움직였을 때의 결과를 확인해 본다. 성공 확률이 1.25% 정도 나온다.

³⁶<https://www.youtube.com/playlist?list=PL1MkM4tgefjnKsCWav-Z2F-MMFRx-2gMGG>

³⁷<http://freeablelab.tistory.com/126> <http://ishuca.tistory.com/390>

3. (Dummy)Q-Learning Algorithm을 FrozenLake에 적용해 보면, 성공 확률이 85% ~ 95% 정도 나온다.

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

4. 다음 단계로 (Dummy)Q-Learning을 개선하여 Q-Learning Method를 완성해 보자. Exploit & Exploration, Discount Rate γ 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))

```

action a 가 결정되고 나면, Discount Rate γ 를 적용하면 된다.

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

5. 이제 Deterministic환경을 Stochastic환경으로 변경하면, 성공 확률이 1 ~ 2%로 낮아진다. Stochastic환경이기 때문에, s 어서 action a 를 취했을 때, 얻게 되는 state s' 는 변하기 때문에, Q-Learning Algorithm을 수정해야 한다.

6. Learning Rate α , Discount Rate γ 를 도입하여, 기존 Q값과 새로운 Q값을 혼합한다.

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

이제 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

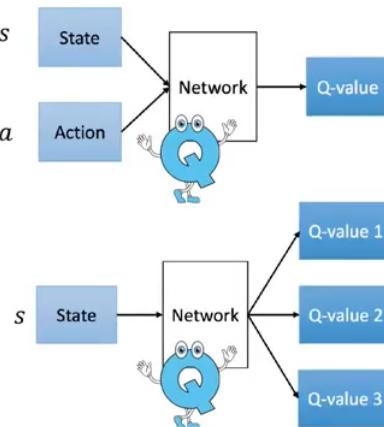


그림 46: Q Netrowk

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'

```

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 47: Q Netrowk

9. ⑤ Q-Network를 실행해 보면, 잘 되지는 않는데 (성공 확률 50 ~ 60%), 그 이유는 다음과 같다.

- Correlations between samples
- Non-Stationary targets

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)

♠ 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.

Algorithm 1 Deep Q-learning with Experience Replay

```
Initialize replay memory  $\mathcal{D}$  to capacity  $N$ 
Initialize action-value function  $\hat{Q}$  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  $\phi_{t+1} = \phi(s_{t+1})$ 
        Store transition  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $\mathcal{D}$ 
        Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $\mathcal{D}$ 
        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
```

(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  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$ 
        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  $\hat{Q} = Q$ 
    End For
End For
```

(b) DQN, Nature 2015

그림 48: DQN Algorithms

```
batch_size = N. initial state = s
while()

    Get Q = FP(s)  $\leftarrow$  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  $\leftarrow$  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  $\leftarrow$  s'
```

8.3 Policy Gradient

9 Variational AutoEncoder & Generative Adversarial Networks

Generative Adversarial Networks(GAN)은 2014년 I. Goodfellow³⁸등이 제안하면서 발전하기 시작한 Unsupervised Learning 모델이다. A. Ng, Y. Lecun등은 미래의 딥러닝 기술의 키는 Unsupervised Learning이라고 말하고 있다.

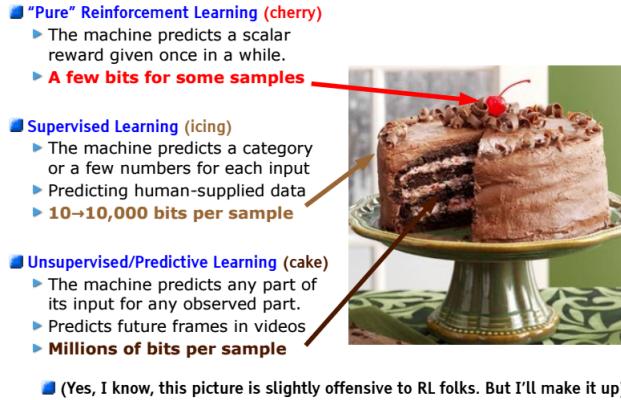


그림 49: How Much Information Does the Machine Need to Predict? (Y. Lecun, NIPS 2016)

9.1 AutoEncoder

♠ Vanilla AutoEncoder

AutoEncoder 모델은 latent vector에 대한 제어가 되지 않는 단점이 있는 반면, VAE모델은 latent vector를 제어하여 새로운 image를 생성할 수 있다.

```
def Encoder(x,dims=None):
    if dims is None: hidden_dims = [256,128]
    else: hidden_dims = dims
    init = tf.contrib.layers.xavier_initializer()
    with tf.variable_scope("encoder"):
        fc1 = tf.layers.dense(x, units=hidden_dims[0] ,
                             activation=tf.nn.sigmoid,kernel_initializer=init,name='fc1')
        fc2 = tf.layers.dense(fc1, units=hidden_dims[1] ,
                             activation=tf.nn.sigmoid,kernel_initializer=init,name='fc2')
    return fc2

def Decoder(x,dims=None):
    if dims is None: hidden_dims = [256,784]
    else: hidden_dims = dims
    init = tf.contrib.layers.xavier_initializer()
    with tf.variable_scope("decoder"):
        fc1 = tf.layers.dense(x, units=hidden_dims[0] ,
                             activation=tf.nn.sigmoid,kernel_initializer=init,name='fc1')
        fc2 = tf.layers.dense(fc1, units=hidden_dims[1] ,
                             activation=tf.nn.sigmoid,kernel_initializer=init,name='fc2')
    return fc2

def Run_AutoEncoder():
    mnist = input_data.read_data_sets("D:\hccho\ML\PythonCode\CommonDataset\mnist", one_hot=True)
```

³⁸<https://arxiv.org/abs/1406.2661>

```

learning_rate = 0.01
num_input = 784 # MNIST data input (img shape: 28*28)
num_epoch = 200; batch_size = 256
max_iter = int(mnist.train.num_examples*num_epoch/batch_size)

x = tf.placeholder(tf.float32, [None, num_input])
z = Encoder(x)
x_hat = Decoder(z)

loss = tf.reduce_mean(tf.pow(x_hat - x, 2))
optimizer = tf.train.RMSPropOptimizer(learning_rate).minimize(loss)

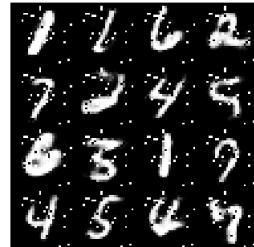
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for i in range(max_iter):
        batch_x, _ = mnist.train.next_batch(batch_size)
        _, loss_ = sess.run([optimizer, loss], feed_dict={x: batch_x})

if __name__ == '__main__':
    s = time.time()
    Run_AutoEncoder()
    e = time.time()
    print(e-s, "sec")

```

1 9 3 8 1 5 8 2 7 0 1 9 3 8 1 5 8 2 7 0
 6 0 1 8 4 7 6 3 0 1 6 0 1 8 4 7 6 3 0 1
 5 2 9 2 4 8 5 3 8 6 5 2 9 2 4 8 5 3 8 6
 0 4 6 0 7 2 6 7 9 2 0 4 6 0 7 2 6 7 9 2
 8 6 8 6 9 5 4 1 4 3 8 6 8 6 9 5 4 1 4 3
 6 3 6 5 6 0 2 9 1 6 6 3 6 5 6 0 2 9 1 6
 0 8 9 7 4 9 2 3 2 2 0 8 9 7 4 9 2 3 2 2
 4 8 1 1 8 2 9 1 5 0 4 8 1 1 8 2 9 1 5 0
 7 1 2 9 1 1 6 0 5 5 7 1 2 9 1 1 6 0 5 5
 6 4 4 1 7 3 2 0 1 6 6 4 4 1 7 3 2 0 1 6

(a) MNIST, 10 epoch. latent dim = 128. 원쪽:
Original, 오른쪽: Reconstructed



(b) weight의 초기값의 stdev
를 1로 설정하여 noise가 많이
발생

그림 50: Vanilla AutoEncoder 결과.

0 7 7 9 3 9 2 2 8 7 0 1 8 9 1 1 1 9 9 9
 1 1 0 0 1 2 9 5 4 6 1 1 0 0 1 9 9 9 9 9 9
 1 5 9 1 4 0 3 1 8 2 1 8 9 1 9 8 9 1 9 9
 8 5 / 5 1 4 0 7 6 2 9 9 1 9 1 9 8 9 9 9 9
 3 3 0 6 2 0 0 5 0 7 1 8 9 0 8 0 0 9 0 9
 0 2 7 0 3 4 1 2 2 7 0 9 9 0 9 9 1 8 1 8
 0 9 3 2 1 7 8 7 1 2 0 8 1 1 1 9 9 9 1 9
 1 5 9 8 1 0 3 4 4 0 1 1 9 9 1 0 9 9 9 0
 2 2 7 7 7 2 0 2 2 0 8 0 9 1 9 9 8 1 9 0
 3 1 0 2 7 1 1 4 7 2 0 8 1 0 9 9 9 9 9 1 0

(a) MNIST, 10 epoch. latent dim = 1

5 2 3 2 3 2 4 2 2 2 7 1 3 0 3 8 9 8 1 1
 3 3 7 2 6 9 0 3 1 1 2 3 8 8 9 0 3 1 1
 3 6 0 0 6 2 1 5 8 2 3 0 0 8 2 1 8 8 1
 2 5 3 3 1 8 1 0 6 0 3 1 3 8 1 9 1 0 0 5
 6 0 8 6 5 1 1 5 1 6 4 0 8 0 8 1 7 7 1 0
 0 3 8 2 9 7 9 5 4 9 0 3 7 8 9 9 9 5 9 9
 3 3 5 4 6 6 8 6 5 4 3 3 3 9 0 0 8 0 1 9
 6 9 4 8 6 2 4 5 7 3 8 9 9 8 0 0 9 8 7 3
 8 7 9 5 4 8 3 1 1 8 9 0 4 1 9 8 3 1 1
 5 4 1 1 2 4 0 0 4 8 8 9 1 1 1 8 0 0 8 3

(b) MNIST, 10 epoch. latent dim = 2

그림 51: Vanilla AutoEncoder 결과

♠ Convolution AutoEncoder

```
def Encoder_Conv(x,):
    init = tf.contrib.layers.xavier_initializer()
    with tf.variable_scope("encoder-conv"):
        x = tf.reshape(x, [-1, 28, 28, 1])
        conv1_1 = tf.layers.conv2d(x, filters=32, padding='VALID', kernel_size=5, strides=1,
                                activation=lambda x: tf.nn.leaky_relu(x, 0.01), name = 'conv1_1')
        pool1_2 = tf.layers.max_pooling2d(conv1_1, pool_size=2, strides=2, padding='SAME',
                                         name='pool1_2')
        conv2_1 = tf.layers.conv2d(pool1_2, filters=64, padding='VALID', kernel_size=5, strides=1,
                                activation=lambda x: tf.nn.leaky_relu(x, 0.01), name = 'conv2_1')
        pool2_2 = tf.layers.max_pooling2d(conv2_1, pool_size=2, strides=2, padding='SAME',
                                         name='pool2_2')

        flatten = tf.contrib.layers.flatten(pool2_2)
        fc3 = tf.layers.dense(flatten, units=4*4*64 , activation= lambda x: tf.nn.leaky_relu(x,0.01)
                             ,name='fc3')
        fc4 = tf.layers.dense(fc3, units=128 , activation= None ,name='fc4')
        latent = fc4
    return latent

def Decoder_Conv(x,training=True):
    init = tf.contrib.layers.xavier_initializer()
    with tf.variable_scope("decoder-conv"):

        fc1 = tf.layers.dense(x, units=1024 , activation=tf.nn.relu,kernel_initializer=init,name='fc1')
        bn2 = tf.layers.batch_normalization(fc1, training=training, name='bn2')
        fc3 = tf.layers.dense(bn2, units=7*7*128 , activation=tf.nn.relu,kernel_initializer=init
                             ,name='fc3')
        bn4 = tf.layers.batch_normalization(fc3, training=training, name='bn4')
        to_img = tf.reshape(bn4, [-1, 7, 7, 128])
        deconv5 = tf.layers.conv2d_transpose(to_img,padding='SAME',filters=64,kernel_size=4, strides=2,
                                            activation=tf.nn.relu, name='deconv5')
        bn6 = tf.layers.batch_normalization(deconv5, training=training, name='bn6')
        deconv7 = tf.layers.conv2d_transpose(bn6,padding='SAME',filters=1,kernel_size=4, strides=2,
                                            activation=tf.nn.tanh, name='deconv7')
        img = tf.contrib.layers.flatten(deconv7)
    return img
```

♠ AutoEncoder의 활용

- Data 압축, Dimension Reduction 또는 Dimension Reduction 차원을 사전에 가능하기 위한 방법.
- Pre-training을 통한 weight 초기화.

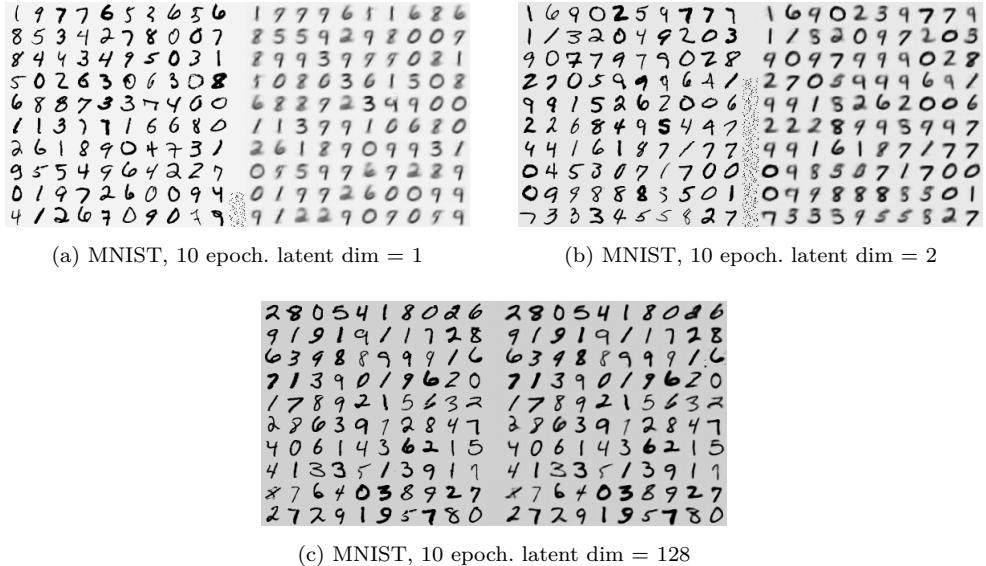


그림 52: Convolution AutoEncoder. 왼쪽: Original, 오른쪽: Reconstructed

9.2 Cross Entropy & KL Divergence

- Entropy: 여러가지 해석이 가능하지만, 여기서는 정보량에 대한 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 에 의한 정보량을 확률 P 에 대한 기대값을 Cross Entropy라 한다.

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

- Kullback–Leibler divergence

$$\begin{aligned} D_{KL}(P||Q) &= (\text{Cross Entropy}) - (\text{Entropy}) \\ &= H(P, Q) - H(P) \\ &= - \sum_i p_i \log q_i + \sum_i p_i \log p_i \geq 0 \end{aligned}$$

The Kullback–Leibler divergence is always non-negative, a result known as Gibbs' inequality.

예를 들어, $p(x) = N(\mu_1, \sigma_1)$, $q(x) = N(\mu_2, \sigma_2)$ 일 때,

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

- 정답이 되는 확률분포 P 가 있을 때, 확률분포 Q 가 P 에 가깝게 되기를 바란다면, $D_{KL}(P||Q)$ 가 최소가 되도록 하면 된다. 이는 cross entropy $H(P, Q)$ 가 최소화되는 것과도 같다.
- Recall that for continuous distributions P and Q , the KL divergence is

$$KL(P||Q) = \int_x P(x) \log \frac{P(x)}{Q(x)} dx$$

In the limit (as $m \rightarrow \infty$), samples will appear based on the data distribution P_r , so

$$\begin{aligned}
\lim_{m \rightarrow \infty} \max_{\theta \in \mathbb{R}^d} \frac{1}{m} \sum_{i=1}^m \log P_\theta(x^{(i)}) &= \max_{\theta \in \mathbb{R}^d} \int_x P_r(x) \log P_\theta(x) dx \\
&\Leftrightarrow \min_{\theta \in \mathbb{R}^d} - \int_x P_r(x) \log P_\theta(x) dx \\
&\Leftrightarrow \min_{\theta \in \mathbb{R}^d} \int_x P_r(x) \log P_r(x) dx - \int_x P_r(x) \log P_\theta(x) dx \\
&= \min_{\theta \in \mathbb{R}^d} KL(P_r \| P_\theta)
\end{aligned}$$

9.3 Variational AutoEncoder

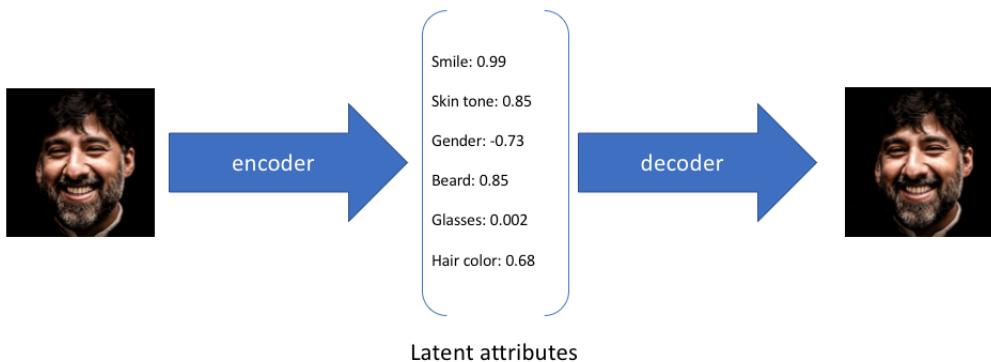


그림 53: AutoEncoder³⁹

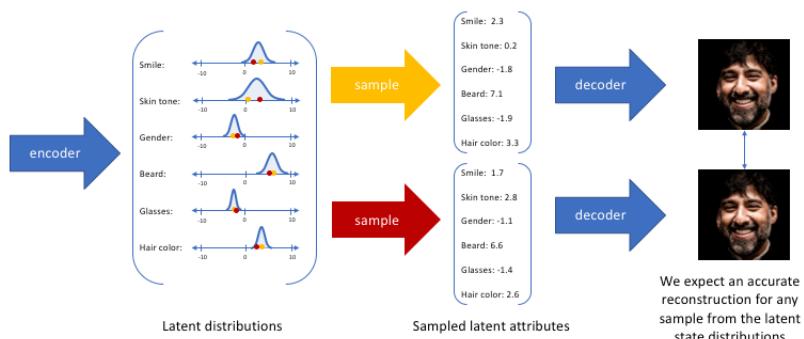


그림 54: VAE: Latent Distribution and Sampling

설명을 위해 batch size가 1이라고 생각해보자. VAE는 Latent vector $z_i (i = 1, \dots, N_z)$ 의 각 분포를 만들어 내야 하는데, 이를 위해서 정규분포라 가정하고, 평균(μ_i)과 표준편차(σ_i)를 구하는 방식으로 접근한다. 즉, Latent vector의 분포를 나타내기 위한 평균, 표준편차를 뜻하는 parameter가 된다. 여기서의 평균과 분산은 input image 여러개의 data로 부터 통계적으로 계산되는 평균, 분산이 아니다. VAE는 N 개의 input data(images)에 대하여 latent vector $z \in \mathbb{R}^N \times \mathbb{R}^{N_z}$ 를 생성해야 한다(사실은 random variable z 가 실수의 형태로 생성되지는 않는다. 정규분포로 가정하고 평균, 분산을 생성한다). 생성되는 z 의 distribution을 잘 control하는 것이 중요하다. 다시

³⁹<https://www.jeremyjordan.me/variational-autoencoders/>

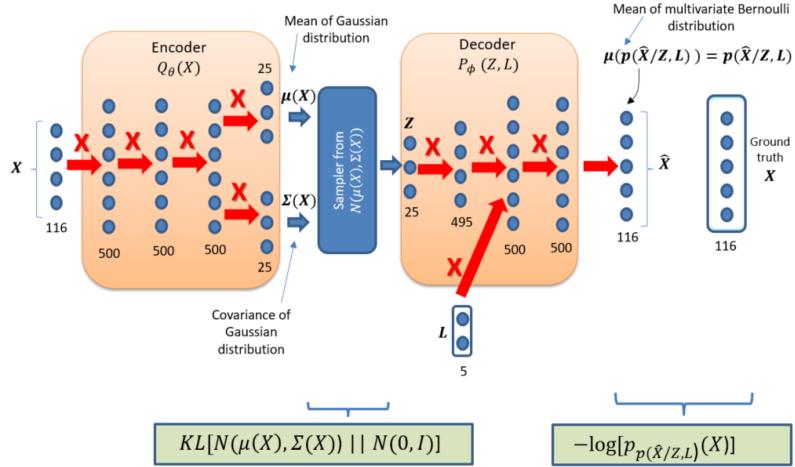


그림 55: Variational AutoEncoder

말해, $z = (z_{ij})$ 의 원소 z_{ij} 마다의 distribution을 training 시키는 것이다.

training 단계의 중간에 계산되는 평균(μ_i)과 표준편차(σ_i)는 training이 종료된 이후에는 사용되지 않는다. latent vector는 $N(0, 1)$ 로 부터 sampling해야 한다. 그리고 encoder와 decoder의 구현은 분리하여 구현해야 한다. 다시 말해 encoder와 decoder의 batch size가 다를 수도 있다는 것을 염두해 두고 구현해야 한다.

Encoder:

```
X: input image(N × Nx)
X̂(xi): input image + noise
X̂ → X̂w0 + b0: (N × Nh) → elu → dropout
    → h0 → h0w1 + b1: (N × Nh) → tanh → dropout → h1
    → h1w2 + b2: (N × 2Nz) → μ: (N × Nz), softplus(Σ): (N × Nz)
    → sampling z ~ N(μ, Σ)
```

Decoder:

```
z → z̄w0 + b̄0: (N × Nh) → tanh → dropout
    → b̄0 → b̄0w1 + b̄1: (N × Nh) → elu → dropout
    → b̄1 → b̄1w2 + b̄2: (N × Nx) → sigmoid
    → Y(yi)
```

Reconstruction Loss:

$$-\sum_{i=1}^{N_x} \left(x_i \log y_i + (1 - x_i) \log(1 - y_i) \right) \rightarrow \text{reduce mean}$$

Regularization Loss (KL divergence):

$$\sum_{i=1}^{N_z} \left(-\log \sigma_i + \frac{\sigma_i^2 + \mu_i^2}{2} - \frac{1}{2} \right) \rightarrow \text{reduce mean}$$

- VAE의 단점: VAEs tend to be far more stable and easier to train but currently don't produce samples that are as pretty as GANs.
- MLE 방법을 통해 likelihood를 최대화하는 파라미터를 구하는 접근법을 사용한다. log-likelihood $\log p_\theta(x)$ 를 구해보자.

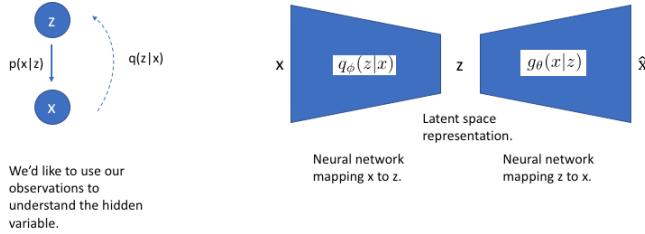


그림 56: VAE: probability distribution structure. p : real distribution, q : tractable distribution. How can we know how well our variational posterior $q_\phi(z|x)$ approximates the true posterior $p(z|x)$?

$$\begin{aligned}
 \log p_\theta(x) &= \int_z q_\phi(z|x) \log p_\theta(x) \\
 &= \int_z q_\phi(z|x) \log \frac{p_\theta(z, x)}{p_\theta(z|x)} \\
 &= \int_z q_\phi(z|x) \log \left(\frac{p_\theta(z, x)}{q_\phi(z|x)} \frac{q_\phi(z|x)}{p_\theta(z|x)} \right) \\
 &= \int_z q_\phi(z|x) \log \frac{p_\theta(z, x)}{q_\phi(z|x)} + \int_z q_\phi(z|x) \log \frac{q_\phi(z|x)}{p_\theta(z|x)} \\
 &= L(\theta, \phi; x) + D_{KL}(q_\phi(z|x) || p_\theta(z|x)) \\
 &\geq L(\theta, \phi; x)
 \end{aligned}$$

- 위 식에서 ELBO(Evidence Lower Bound) $L(\theta, \phi; x)$ 를 다음과 같이 정의한 것이다.

$$\begin{aligned}
 L(\theta, \phi; x) &:= \int_z q_\phi(z|x) \log \frac{p_\theta(z, x)}{q_\phi(z|x)} \\
 &= \int_z q_\phi(z|x) (\log p_\theta(z, x) - \log q_\phi(z|x)) \\
 &= \mathbb{E}_{q_\phi(z|x)} [-\log q_\phi(z|x) + \log p_\theta(z, x)]
 \end{aligned}$$

log-likelihood $\log p_\theta(x)$ 를 최대화하기 위해, 간접적으로 ELBO $L(\theta, \phi; x)$ 를 최대화하자. 그러기 위해서 $L(\theta, \phi; x)$ 를 다음과 같이 전개해 보자.

$$\begin{aligned}
 L(\theta, \phi; x) &= \int_z q_\phi(z|x) \log \frac{p_\theta(z)p_\theta(x|z)}{q_\phi(z|x)} \\
 &= - \int_z q_\phi(z|x) \log \frac{q_\phi(z|x)}{p_\theta(z)} + \int_z q_\phi(z|x) \log p_\theta(x|z) \\
 &= -D_{KL}(q_\phi(z|x) || p_\theta(z)) + \mathbb{E}_{q_\phi(z|x)} [\log p_\theta(x|z)]
 \end{aligned}$$

- regularization loss는 prior distribution $p_\theta(z)$ 을 multivariate standard normal distribution으로 가정한다. encoder를 얻어지는 μ, Σ 를 posterior distribution $q_\phi(z|x)$ 의 평균과 표준편차로 간주하고, posterior-prior distribution의 KL divergence $D_{KL}(q_\phi(z|x) || p_\theta(z))$ 를 최소화한다. KL divergence를 최소화하는 것이 $\mu = (0, \dots, 0), \Sigma = (1, \dots, 1)$ 이 정답이라서가 아니고, variable들의 자유도를 제한하는 regularization 효과를 주는 것이다.

- prior distribution을 정규분포로 가정하는 것은 variable을 모아주는 효과를 주는 것이 가능하기 때문이다.
- reconstructin loss: $\mathbb{E}_{q_\phi(z|x)} [\log p_\theta(x|z)]$ 를 최대화하는 부분을 살펴보자. output y_i 를 input x_i 에 가깝게 하는 것을 목적으로 binary cross entropy(Bernoulli 가정)나 L_2 -norm(MSE, Gaussian가정)을 최소화하는 방식을 취할 수도 있다.

- 그런데 식 자체가 log-likelihood이므로 직접적인 해석을 해보자⁴⁰. output y_i 를 Bernoulli distribution의 확률로 간주해 보자. MNIST image의 각 픽셀은 0 또는 1(흑백)의 값만을 가진다고 보면, y_i 를 픽셀이 1의 값을 가질 확률로 해석할 수 있다. input data의 픽셀 값도 1이 나오는 출현 빈도로 간주한다. 예를 들어, 픽셀 값이 0.4이면 10번 중 4번은 1의 값이 나왔고, 6번은 0의 값이 나왔다고 보는 것이다. 이런식의 해석을 하면 binary cross entropy식은 Bernoulli distribution의 Maximum Likelihood function이 된다.

$$x_i \log y_i + (1 - x_i) \log(1 - y_i) \leftarrow x_i: \text{관측 회수}, y_i: \text{추정 확률}$$

- output y_i 를 gaussian의 평균으로 해석하면 Maximum Likelihood function 식은 Mean Square Error가 된다.

Encoder:

```
X: input image( $N \times 784$ ) -> reshape( $N, 28, 28, 1$ )
-> conv[w=(5,5,1,16), stride (1,2,2,1)] -> ( $N, 14, 14, 16$ ) -> leaky relu
-> conv[w=(5,5,16,32), stride (1,2,2,1)] -> ( $N, 7, 7, 32$ ) -> leaky relu
-> reshape( $N, 7 * 7 * 32$ ) -> FC( $7 * 7 * 32, 2 * N_z$ ) ->  $\mu : (N \times N_z)$ , softplus( $\Sigma$ ): ( $N \times N_z$ )
-> sampling  $z \sim N(\mu, \Sigma)$ 
```

Decoder:

```
 $z \rightarrow$  FC( $N_z, 7 * 7 * 32$ ) -> reshape( $N, 7, 7, 32$ ) -> relu
-> deconv[w=(5,5,16,32), outshape=( $N, 14, 14, 16$ )] -> relu
-> deconv[w=(5,5,1,16), outshape=( $N, 28, 28, 1$ )] -> sigmoid
-> reshpaes( $N, 28 * 28$ )
->  $Y(y_i)$ 
```

Reconstruction Loss:

$$-\sum_{i=1}^{N_x} \left(x_i \log y_i + (1 - x_i) \log(1 - y_i) \right) \rightarrow \text{reduce mean}$$

Regularization Loss(KL divergence):

$$\sum_{i=1}^{N_z} \left(-\log \sigma_i + \frac{\sigma_i^2 + \mu_i^2}{2} - \frac{1}{2} \right) \rightarrow \text{reduce mean}$$

⁴⁰Neural Net의 출력값을 MLE에서의 확률분포 parameter로 보는 관점. Bernoulli 분포라면 발생 확률 p 로 보는 것이고, Gaussian 이라면 평균 μ 로 보는 것이다. 당연히 입력값은 given data로 보면 된다.

9.4 여러가지 VAE

VAE에는 여러가지 변형된 모형이 있는데, CVAE, Denosing VAE, Adversarial AE, Adversarial Variational Bayes 등이 있다.

♠ CVAE(Conditional VAE)

CVAE(Conditional VAE): encoder의 input data와 decoder의 latent vector에 label과 같은 추가 정보를 넣어주는 모델.

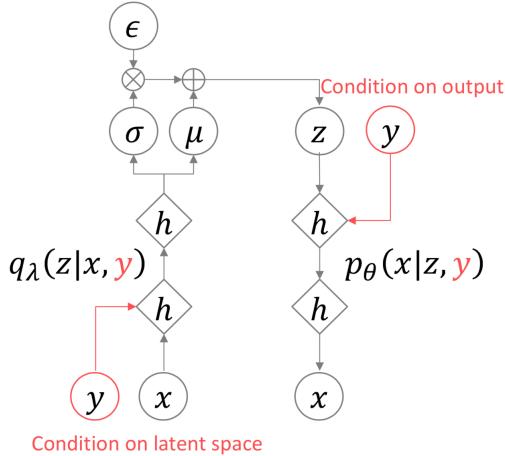
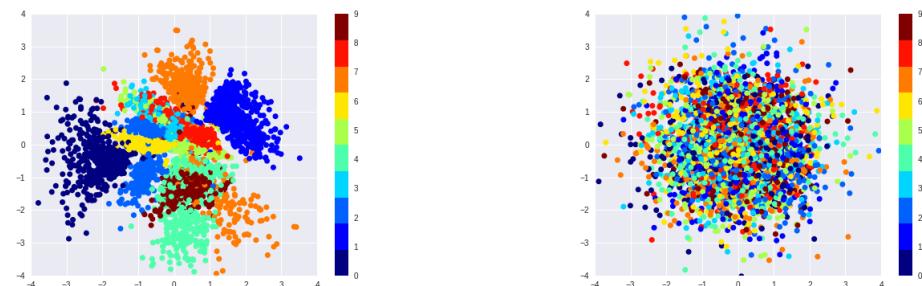


그림 57: Conditional VAE Structure: label 정보가 input image(x), latent vector(z)와 각각 concat되어 들어간다. input image가 4-dimension인 경우는 label 정보를 4차원으로 변형하여 concat해야 한다.⁴¹



(a) VAE: 2차원 latent vector에 mnist 숫자 정보를 포함해야하기 때문에, 가장 중요한 digit에 대한 정보로 확연히 구분되게 latent vector가 학습되었다.

(b) CVAE: VAE와 달리 digit에 대한 정보는 추가로 제공되기 때문에, latent vector는 digit에 대한 정보를 제외하고 다른 중요한 정보를 학습했다. 예를 들어, 굵기와 회전.

그림 58: 2-dim latent vector에 대한 결과(100 epoch)

♠ Denosing VAE

encoder의 input data에 noise를 추가한 후, 원 data를 복원할 수 있도록 training한다. 예를 들어, input image의 각 pixel x 에 random number 0 또는 1 r_1, r_2 를 각각 만들어,

$$x \leftarrow x * r_1 + r_2$$

⁴¹y=tf.reshape(y, [-1, 1, 1, 10]); x=tf.concat((x,y*tf.ones_like(x)),axis=-1)

♠ AAE(Adversarial AE, 2015)

VAE(Alioreza Makhzani, Jonathon Shlens, Navdeep Jaitly, Ian Goodfellow, Brendan Frey, Adversarial Autoencoders⁴²)는 latent distribution에 대한 KL-Divergence를 계산해야 하기 때문에 KL Divergence 계산이 가능한 Gaussian distribution을 이용해서 모델을 전개했다. AAE는 KL Divergence 대신 GAN loss를 사용하는 모델이다. KL Divergence를 계산하지 않기 때문에 latent distribution으로 다양한 분포를 가져와서 모델을 구성할 수 있다.

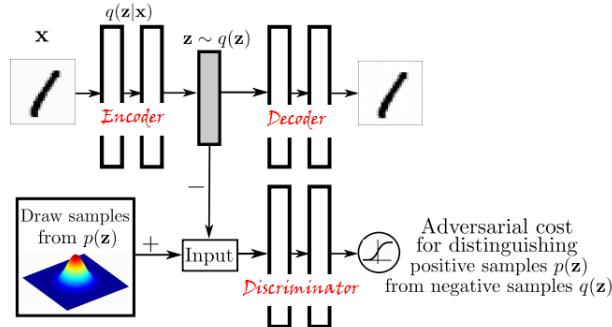


그림 59: Adversarial AutoEncoder Model Structure

- AAE모델은 Encoder(Generator), Decoder, Discriminator 3개로 구성되어 있다.
- Encoder는 latent distribution을 만들어 내는 역할을 한다. Encoder가 만드는 latent vector는 Decoder로 넘어가면 image를 잘 생성하게 해야하고 Discriminator를 속이기도 해야 한다.
- Decoder는 Encoder가 만든 latent vector를 받아 image를 만드는 역할(reconstruction)을 한다. Decoder의 reconstruction loss를 optimization 할 때는 Encoder, Decoder의 variable들을 같이 update한다(반드시 그래야 되는 것 같지는 않고, 구현된 코드들이 이렇게 되어 있음).
- Discriminator는 Real Distribution과 Encoder가 만든 가짜 Distribution을 잘 구분해야 한다.
- KL Divergence를 계산할 필요가 없는 모델이기 때문에 다양한 분포를 latent distribution으로 사용할 수 있다. latent distribution은 label에 무관하게 하나의 distribution을 사용할 수도 있고, label마다 다른 분포를 사용할 수도 있다. 예를 들어, mixed gaussian, swiss roll 등을 사용할 수도 있다.

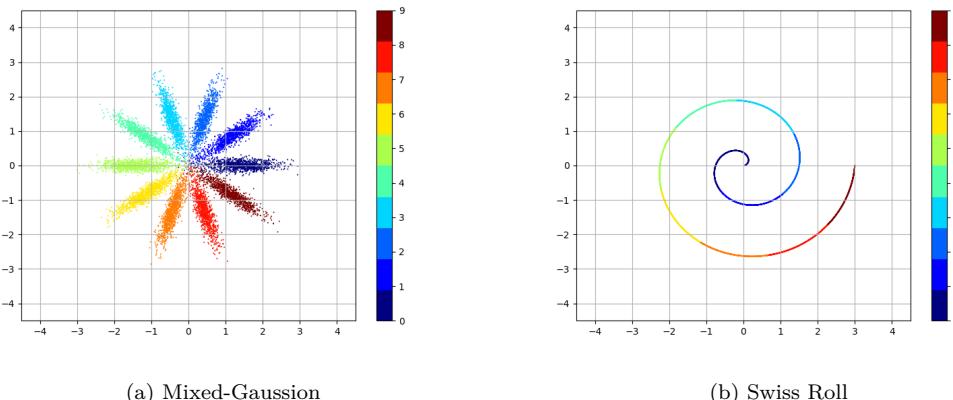


그림 60: Latent Distribution(dim 2) 예

⁴²<https://arxiv.org/abs/1511.05644>

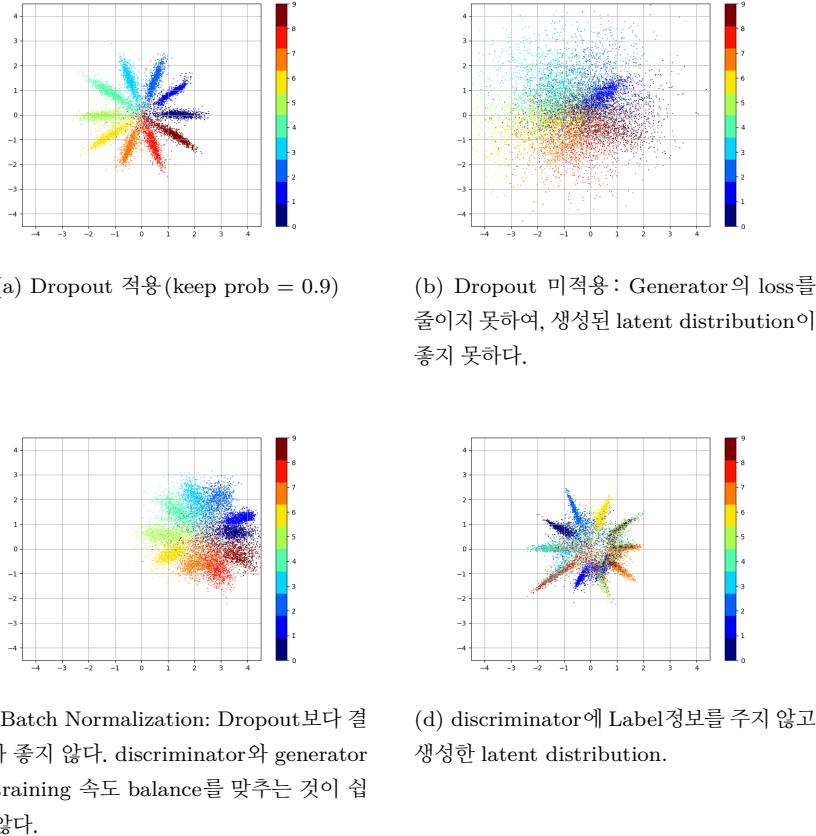


그림 61: Dropout 효과 (prior = mixed gaussian) & Label 효과

- 모델에서는 생성된 latent vector z 에 label을 concat해서 discriminator에 전달하고 있다. real distribution과 fake distribution이 생성된 distribution이 무엇을 만들기 위한 distribution인지 알려주고 있는 것이다. 단순히 '이게 진짜인지 가짜인지 맞춰봐?' 하는 것과 '이건 3을 만들려고 한 것인데, 진짜인지 맞춰봐?'라고 하는 것의 차이이다. 후자가 월씬 쉽게 진짜, 가짜를 구분할 수 있다. 그래서 discriminator의 성능을 향상시킬 수 있는 것이다.

♠ AVB(Adversarial Variational Bayes, 2017)

- Lars Mescheder, Sebastian Nowozin, Andreas Geiger, Adversarial Variational Bayes: Unifying Variational Autoencoders and Generative Adversarial Networks.⁴³
- AVB는 AAE와 거의 유사한 구조를 가지고 있지만, 모형 구조에 대한 해석을 달리하고 있다⁴⁴. AVB는 Maximum Likelihood 관점에서 모형을 전개한다. 논문에서는 AAE가 AVB의 approximation 방법이라고 주장하고 있다.
 - encoder 입력 $\text{image}(x)$ 에 random noise를 concat해서 넣어준다. (concat해서 넣어주는 noise가 의미 있는 역할을 하는지 잘 이해되지 않는다.)
 - encoder가 생성해 내는 distribution이 posterior distribution이다.
 - discriminator에 prior 또는 posterior distribution(z) 외에 추가적으로 $\text{image}(x)$ 도 같이 입력되는 구조.

⁴³Adversarial Variational Bayes, <https://arxiv.org/abs/1701.04722>

⁴⁴<https://chrisorm.github.io/AVB-pyt.html>

- discriminator는 posterior에 대해서는 1을 prior에 대해서는 0이 되도록 discriminator를 training한다.
- generator loss는 reconstruction loss(L_2 대신 cross-entropy loss) 와 posterior에 대한 discriminator의 (sigmoid 취하기 전) logit 값이 작아지도록 training한다. The output of T (without the sigmoid) are the logits, which at optimality of the discriminator is the ratio of the two distributions(likelihood ratio).
- 전반적으로 AAE보다 결과가 좋지 않다. dim- z = 2 일 때도 AAE만큼의 결과가 나오지 않는다.

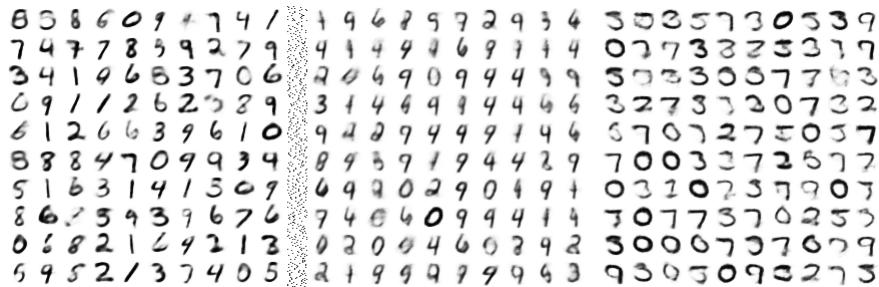


그림 62: AVB 결과: dim- z = 10, dim-eps = 10, G/D training 비율=1. 왼쪽: vanilla 구조, epoch = 20. 가운데: DC 구조, epoch = 20. 오른쪽: DC 구조, epoch = 100

9.5 Vanilla GAN

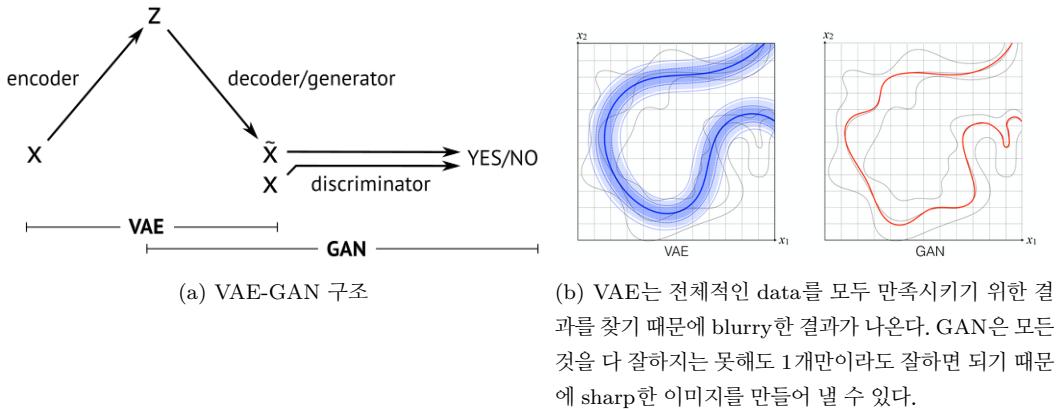


그림 63: VAE-GAN. Mode Collapse: GAN의 Generator가 Discriminator를 속일 수는 있지만 복잡한 실제 데이터 분포를 배우지 못하고 극히 낮은 종류의 작은 공간에 갇히게 되는 경우.

GAN(2014)⁴⁵에 들어가기 이전에 앞서, VAE와 GAN의 구조를 비교해 보자.

VAE: input image(예: dim 784)로부터 latent z (예: dim $2N_z$)를 뽑아내는데, latent의 각 원소는 Normal Distribution의 평균과 표준편차를 fitting하는 parameter로 모델링한다. 즉, z 는 embedding distribution의 평균 (N_z 개)과 표준편차 (N_z 개)를 추정하는 vector이다. 이때, embedding distribution을 서로 독립인 standard normal distribution에 가깝도록 KL Divergence를 최소화 한다. 서로 독립으로 가정하는 것은 encoder가 (PCA와 유사하게) 서로 독립인 분포를 뽑아내는 것으로 보는 것이다.

GAN: input latent vector z (예: 서로 독립인 128차원 standard normal random vector or $[-1, 1]$ uniform distribution)로 부터 image를 생성하는 것. image를 생성하는 것을 image의 distribution을 찾는 것이라 표현하기도 한다.

⁴⁵Generative Adversarial Networks, <https://arxiv.org/abs/1406.2661>

Neural Network 모델에서 정규분포나 서로 독립임을 가정하는 이유는 deep한 Network은 앞단에서 더 복잡한 분포 간단한 분포로 변환할 수도 있고 상관관계를 가지기 않도록 변환할 수도 있기 때문이다.

♠ Vanilla GAN

GAN의 Value Function

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

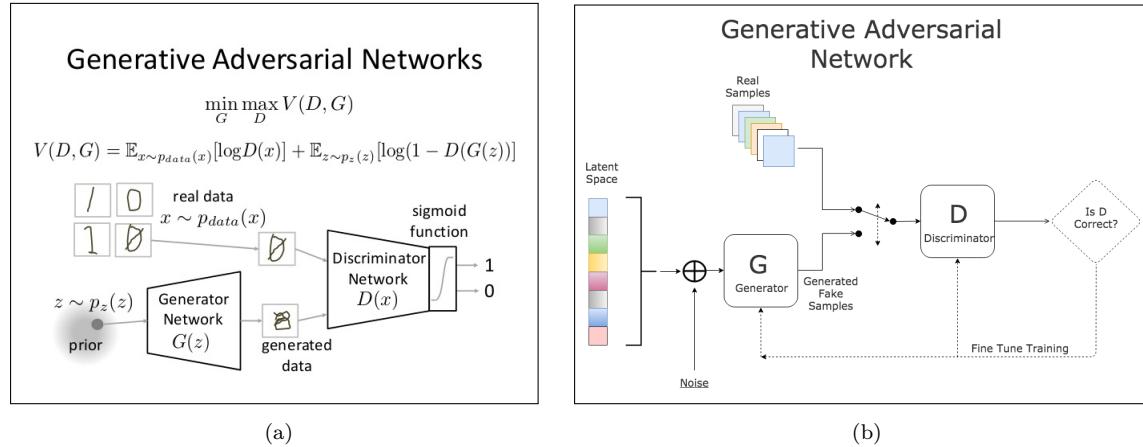


그림 64: GAN Model

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

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

Loss_D 를 줄이기 위해 Discriminator의 weight update와 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)

```

- (cs231n) Make an AdamOptimizer with a 0.001 learning rate, beta1=0.5 to minimize G-loss and D-loss separately. The trick of decreasing beta was shown to be effective in helping GANs converge in the 'Improved Techniques for Training GANs'⁴⁶ paper. In fact, with our current hyperparameters, if you set beta1 to the Tensorflow default of 0.9, there's a good chance your discriminator loss will go to zero and the generator will fail to learn entirely. In fact, this is a common failure mode in GANs; if your $D(x)$ learns to be too fast (e.g. loss goes near zero), your $G(z)$ is never able to learn. Often $D(x)$ is trained with SGD with Momentum or RMSProp instead of Adam, but here we'll use Adam for both $D(x)$ and $G(z)$.

⁴⁶<https://arxiv.org/abs/1606.03498>

- A vanilla GAN architecture allows no real spatial reasoning. It is unable to reason about things like “sharp edges” in general because it lacks any convolutional layers.
- Tips and tricks to make GANs work⁴⁷.
 - Normalize the images between -1 and 1.
 - tanh as the last layer of the generator output.
 - Dont sample from a Uniform distribution. Sample from a gaussian distribution.
- 그림(65)는 DCGAN 논문에 있는 내용인데, GAN의 일반적인 구조에서 적용될 것 같다. Generator의 마지막 activation function을 tanh로 사용하는 것은 이미지의 색을 대칭적인 구조가 될수 있도록 하여 흑/백을 명확하게 하는 효과가 있다. 그러기 위해서는 MNIST image 값을 $[0, 1]$ 에서 $[-1, 1]$ 로 변환해 주어야 한다.

Architecture guidelines for stable Deep Convolutional GANs
<ul style="list-style-type: none"> • Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator). • Use batchnorm in both the generator and the discriminator. • Remove fully connected hidden layers for deeper architectures. • Use ReLU activation in generator for all layers except for the output, which uses Tanh. • Use LeakyReLU activation in the discriminator for all layers.

그림 65: DCGAN Guideline

- 어째든 GAN에서는 Discriminator와 Generator가 같이 잘 training 되어야 하는데, 특히 Vanilla GAN에서는 Generator의 생성 능력이 떨어지기 때문에, Generator와 Discriminator가 균형있게 training 되는 것이 쉽지 않다.

♠ **LSGAN: Least Squares GAN** We'll now look at Least Squares GAN⁴⁸, a newer, more stable alternative to the original GAN loss function. For this part, all we have to do is change the loss function and retrain the model. 여기서 Discriminator D 가 return하는 값은 sigmoid를 취하지 않은 값이다 ($D(x) \in (-\infty, \infty)$). We'll implement the generator loss:

$$\ell_G = \frac{1}{2} \mathbb{E}_{z \sim p(z)} [(D(G(z)) - 1)^2]$$

and the discriminator loss:

$$\ell_D = \frac{1}{2} \mathbb{E}_{x \sim p_{\text{data}}} [(D(x) - 1)^2] + \frac{1}{2} \mathbb{E}_{z \sim p(z)} [(D(G(z)))^2]$$

♠ Conditional GAN(2014) & InfoGAN(2016)

- CGAN: Mehdi Mirza, Simon Osindero, Conditional Generative Adversarial Nets⁴⁹
- CGAN은 Latent vector z 에 label 정보를 같이 주어, Generator가 어떤 image가 생성되어야 하는지 가이드해 준다. 그리고, Discriminator에게는 image가 어떤 이미지를 인지 label 정보를 추가로 알려준다. 즉, Discriminator에게 ‘이건 3이라는 이미지인데 진짜인지 가짜인지 구별해봐?’라고 하는 것이다.

⁴⁷<https://github.com/soumith/ganhacks>

⁴⁸<https://arxiv.org/abs/1611.04076>

⁴⁹<https://arxiv.org/abs/1411.1784>



그림 66: MNIST 20 epoch에 대한 결과: Vanilla GAN(왼쪽)-LSGAN(가운데)-DCGAN(오른쪽). Vanilla GAN 보다 LSGAN의 결과가 조금 나아 보인다. DCGAN의 결과는 상대적으로 훨씬 우수하다.

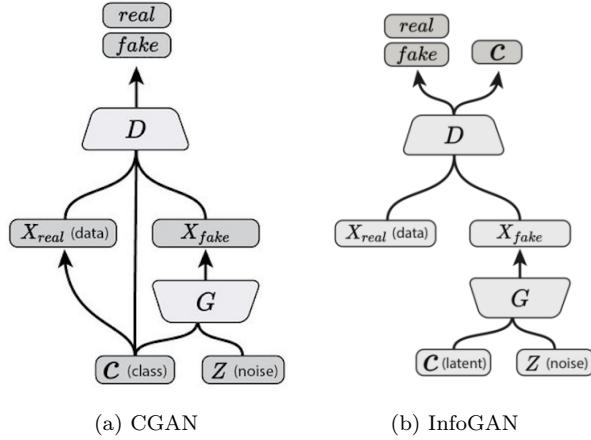


그림 67: CGAN & InfoGAN

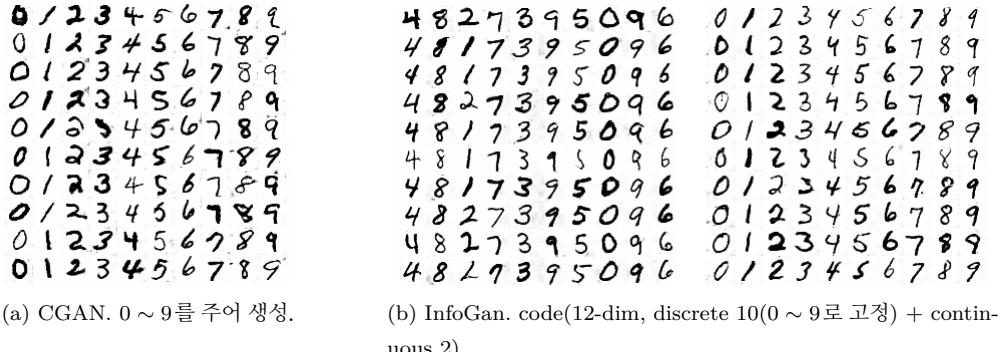


그림 68: CGAN-InfoGan 결과: DCGAN 구조, epoch 20. InfoGAN에서 왼쪽 결과는 discrete code 10개가 training 을 통해 digit 정보를 정확히 잡아 내기를 바랬지만, 9는 2개로 나누어 졌고, 1과 2는 서로 섞여있는 결과가 나왔다. real image에 label 정보를 주기 않았기 때문이다. 오른쪽은 Supervised-InfoGAN의 결과이다. real image의 discrete code가 복원될 수 있도록 real image에 대한 reconstruction loss를 추가했기 때문에 digit이 제대로 복원되고 있다.

- InfoGAN: Xi Chen, Yan Duan, Rein Houthooft, John Schulman, Ilya Sutskever, Pieter Abbeel, InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets⁵⁰
- InfoGAN은 GAN에서의 latent vector z 에 추가적인 information(code c)을 Generator에 넘겨주고 이미지를 생성하게 한다. 생성된 이미지에 code c 가 내포되게 하기 위해서 code를 복원하는 network을 추가로 둔다. 그래서 InfoGAN은 Generator, Discriminator, Mutual Information Reconstructor 3개를 가지는 구조이다.

⁵⁰<https://arxiv.org/abs/1606.03657>

- code에는 discrete code와 continuous code의 결합으로 주는데, discrete code는 MNIST에서 label 정보에 해당한다. continuous code는 image의 중요한 특성을 잘 뽑아서 나타내주기를 바라는 부분이다. 예를 들어, rotation이나 thickness를 나타내는 정보등이 해당된다.

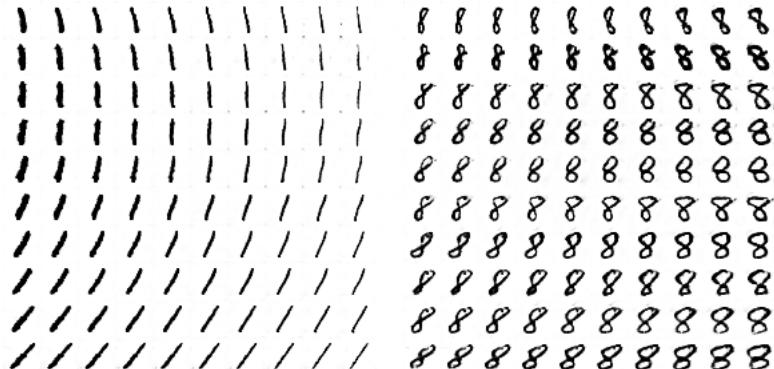


그림 69: continuous code 변화에 대한 InfoGAN 결과. label과 latent vector는 고정하고, continuous code에 대해, $[-2, 2] \times [-2, 2]$ 에서 grid(10×10)를 생성한 결과.

♠ ACGAN(2014)

ACGAN⁵¹은 InfoGAN과 거의 유사하다.

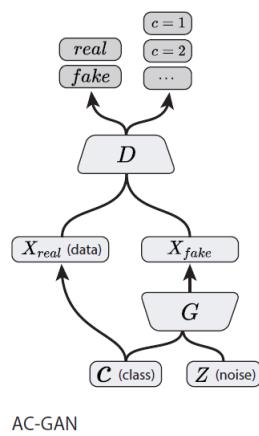


그림 70: ACGAN Structure

9.6 Wasserstein GAN

- A background on generative models⁵² ⁵³: When learning generative models, we assume the data we have comes from some unknown distribution P_r . (The r stands for real.) We want to learn a distribution P_θ that approximates P_r , where θ are the parameters of the distribution.

You can imagine two approaches for doing this.

- Directly learn the probability density function P_θ . Meaning, P_θ is some differentiable function such that $P_\theta(x) \geq 0$ and $\int_x P_\theta(x) dx = 1$. We optimize P_θ through maximum likelihood estimation.

⁵¹Conditional Image Synthesis With Auxiliary Classifier GANs(2016), <https://arxiv.org/abs/1610.09585>

⁵²<https://www.alexirpan.com/2017/02/22/wasserstein-gan.html>

⁵³<https://lilianweng.github.io/lil-log/2017/08/20/from-GAN-to-WGAN.html>

- 이와 같은 방법을 위해서는 P_θ 를 구체적으로 알거나 가정해야 하지만, 쉽지 않다. 그래서 다음의 방법을 생각해 보자.
- Learn a function that transforms an existing distribution Z into P_θ . Here, g_θ is some differentiable function, Z is a common distribution (usually uniform or Gaussian), and $P_\theta = g_\theta(Z)$
- GAN모델에서의 generator(network)가 여기서의 g_θ 역할을 하게된다.
- Note that if $Q(x) = 0$ at an x where $P(x) > 0$, the KL divergence goes to $+\infty$. This is bad for the MLE if P_θ has low dimensional support, because it'll be very unlikely that all of P_r lies within that support. If even a single data point lies outside P_θ 's support, the KL divergence will explode.
- To deal with this, we can add random noise to P_θ when training the MLE. This ensures the distribution is defined everywhere. But now we introduce some error, and empirically people have needed to add a lot of random noise to make models train. That kind of sucks. Additionally, even if we learn a good density P_θ , it may be computationally expensive to sample from P_θ .
- This motivates the latter approach, of learning a g_θ (a generator) to transform a known distribution Z . The other motivation is that it's very easy to generate samples. Given a trained g_θ , simply sample random noise $z \sim Z$, and evaluate $g_\theta(z)$. (The downside of this approach is that we don't explicitly know what P_θ , but in practice this isn't that important.)
- To train g_θ (and by extension P_θ), we need a measure of distance between distributions.(o] 내용을 정리한 분이 엄밀하게는 다른 용어지만, metric, distance, divergence를 혼용하여 사용한다고 양해를 구하고 있다.)
- Looping back to generative models, given a distance d , we can treat $d(P_r, P_\theta)$ as a loss function. Minimizing $d(P_r, P_\theta)$ with respect to θ will bring P_θ to P_r . This is principled as long as the mapping $\theta \mapsto P_\theta$ is continuous (which will be true if g_θ is a neural net).
- 분포의 수렴을 이야기하기 위해서는 convergence를 정의하기 위한 metric이 있어야 하는데, 몇 가지 metric에 대해서 알아보자.

- The Total Variation (TV) distance is

$$\delta(P_r, P_g) = \sup_A |P_r(A) - P_g(A)|$$

- The Kullback-Leibler (KL) divergence is

$$KL(P_r \| P_g) = \int_x \log \left(\frac{P_r(x)}{P_g(x)} \right) P_r(x) dx$$

This isn't symmetric. The reverse KL divergence is defined as $KL(P_g \| P_r)$

- The Jenson-Shannon (JS) divergence: Let P_m be the mixture distribution $P_m = P_r/2 + P_g/2$

$$JS(P_r, P_g) = \frac{1}{2}KL(P_r \| P_m) + \frac{1}{2}KL(P_g \| P_m)$$

- Finally, the Earth Mover (EM) or Wasserstein distance: Let $\Pi(P_r, P_g)$ be the set of all joint distributions γ whose marginal distributions are P_r and P_g . Then

$$W(P_r, P_g) = \inf_{\gamma \in \Pi(P_r, P_g)} \mathbb{E}_{(x,y) \sim \gamma} [\|x - y\|]$$

♠ Wasserstein distance(Earth Mover)

- distribution P_r, P_q 간의 잘 정의된 metric이 있어, $P_g \rightarrow P_r$ 로 수렴할 수 있게 하는 것이 바람직함.
- 그런데, Total Variance, KL divergence, JS divergence는 이런 조건을 만족시키지 못함.

Example. Total Variance, KL divergence, JS divergence에 의한 metric이 연속적이지 못한 예를 들어보자. Consider probability distributions defined over \mathbb{R}^2 . Let the true data distribution be $(0, y)$, with y sampled uniformly from $U[0, 1]$. Consider the family of distributions P_θ , where $P_\theta = (\theta, y)$, with y also sampled from $U[0, 1]$.

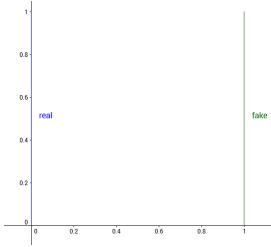


그림 71: Real and fake distribution when $\theta = 1$

We'd like our optimization algorithm to learn to move θ to 0, as $\theta \rightarrow 0$, the distance $d(P_0, P_\theta)$ should decrease. But for many common distance functions, this doesn't happen.

- Total variation: For any $\theta \neq 0$, let $A = \{(0, y) : y \in [0, 1]\}$. This gives

$$\delta(P_0, P_\theta) = \begin{cases} 1 & \text{if } \theta \neq 0, \\ 0 & \text{if } \theta = 0. \end{cases}$$

- KL divergence and reverse KL divergence: Recall that the KL divergence $KL(P\|Q)$ is $+\infty$ if there is any point (x, y) where $P(x, y) > 0$ and $Q(x, y) = 0$. For $KL(P_0\|P_\theta)$, this is true at $(\theta, 0.5)$. For $KL(P_\theta\|P_0)$, this is true at $(0, 0.5)$

$$KL(P_0\|P_\theta) = KL(P_\theta\|P_0) = \begin{cases} +\infty & \text{if } \theta \neq 0, \\ 0 & \text{if } \theta = 0, \end{cases}$$

- Jenson-Shannon divergence: Consider the mixture $M = P_0/2 + P_\theta/2$, and now look at just one of the KL terms.

$$KL(P_0\|M) = \int_{(x,y)} P_0(x, y) \log \frac{P_0(x, y)}{M(x, y)} dy dx$$

For any x, y where $P_0(x, y) \neq 0$, $M(x, y) = \frac{1}{2}P_0(x, y)$, so this integral works out to $\log 2$. The same is true of $KL(P_\theta\|M)$, so the JS divergence is

$$JS(P_0, P_\theta) = \begin{cases} \log 2 & \text{if } \theta \neq 0, \\ 0 & \text{if } \theta = 0, \end{cases}$$

- Earth Mover distance: Because the two distributions are just translations of one another, the best way transport plan moves mass in a straight line from $(0, y)$ to (θ, y) . This gives $W(P_0, P_\theta) = |\theta|$

- 이 예는 TV, JS, KL divergence하에서는 수렴하지 않지만, EM distance로는 수렴하는 경우를 보여주고 있다. 또한 gradient도 항상 0인 것을 알 수 있다. 즉 gradient를 계산하여 optimization을 수행하는 경우, optimal solution을 찾을 수 없다.
- Admittedly, this is a contrived example because the supports are disjoint, but the paper points out that when the supports are low dimensional manifolds in high dimensional space, it's very easy for the intersection to be measure zero, which is enough to give similarly bad results.

■

이제 Wasserstein distance를 계산하는 방법에 대하여 알아보자. 정의에 의한 계산은 쉽지 않다.

- A result from Kantorovich-Rubinstein duality⁵⁴ shows W is equivalent to

$$W(P_r, P_\theta) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{x \sim P_r}[f(x)] - \mathbb{E}_{x \sim P_\theta}[f(x)]$$

where the supremum is taken over all 1-Lipschitz functions. Wasserstein distance를 계산하는 방법에 대한 것은 이 포스트⁵⁵를 참고하면 된다.

- 이제 현실적인 WGAN⁵⁶모델에서 Wasserstein distance를 계산하는 것을 알아보자. Suppose this function f comes from a family of K-Lipschitz continuous functions, $\{f_w\}_{w \in W}$, parameterized by w . In the modified Wasserstein-GAN, the “discriminator” model is used to learn w to find a good f_w and the loss function is configured as measuring the Wasserstein distance between p_r and p_g .

$$L(p_r, p_g) = W(p_r, p_g) = \max_{w \in W} \mathbb{E}_{x \sim p_r}[f_w(x)] - \mathbb{E}_{z \sim p_r(z)}[f_w(g_\theta(z))]$$

좀 더 자세히 설명하자면, Wasserstein distance를 계산하기 위해서는 모든 1-Lipschitz function을 대상으로 최대값(superimum)을 구하는 것은 쉽지 않기 때문에, 함수 집합 $\{f_w\}_{w \in W}$ 를 대상으로 최대값을 계산하고자 한다. 그래서 함수 집합 $\{f_w\}_{w \in W}$ 를 discriminator의 parameter(weight)를 변화시켜서 구성하는 것이다. 최대값을 구하는 것은 training 과정에서 optimization을 수행하는 것과 동일하다. 그래서 일반적인 GAN 모델에서 dicriminator를 WGAN에서는 critic이라 부른다.

$$\max_{w \in W} [\mathbb{E}(D(\text{real image})) - \mathbb{E}(D(\text{fake image}))]$$

여기서 discriminator D 가 return하는 값은 sigmoid 취하지 않은 logit 값. discriminator D 는 이제 더 이상 real image와 fake image를 구분하는 “discriminator”가 아니고, Wasserstein distance를 계산하는 함수이다. 그래서 critic이라 부른다.

- Thus the “discriminator” is not a direct critic of telling the fake samples apart from the real ones anymore. Instead, it is trained to learn a K-Lipschitz continuous function to help compute Wasserstein distance. As the loss function decreases in the training, the Wasserstein distance gets smaller and the generator model’s output grows closer to the real data distribution.
- One big problem is to maintain the K-Lipschitz continuity of f_w during the training in order to make everything work out. The paper presents a simple but very practical trick: After every gradient update, clamp the weights w to a small window, such as $[-0.01, 0.01]$, resulting in a compact parameter space W and thus f_w obtains its lower and upper bounds to preserve the Lipschitz continuity.

⁵⁴https://en.wikipedia.org/wiki/Wasserstein_metric

⁵⁵<https://vincentherrmann.github.io/blog/wasserstein/>

⁵⁶Wasserstein GAN, <https://arxiv.org/abs/1701.07875>

Algorithm 1 WGAN, our proposed algorithm. All experiments in the paper used the default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$.

Require: : α , the learning rate. c , the clipping parameter. m , the batch size.
 n_{critic} , the number of iterations of the critic per generator iteration.

Require: : w_0 , initial critic parameters. θ_0 , initial generator's parameters.

```

1: while  $\theta$  has not converged do
2:   for  $t = 0, \dots, n_{\text{critic}}$  do
3:     Sample  $\{x^{(i)}\}_{i=1}^m \sim \mathbb{P}_r$  a batch from the real data.
4:     Sample  $\{z^{(i)}\}_{i=1}^m \sim p(z)$  a batch of prior samples.
5:      $g_w \leftarrow \nabla_w [\frac{1}{m} \sum_{i=1}^m f_w(x^{(i)}) - \frac{1}{m} \sum_{i=1}^m f_w(g_\theta(z^{(i)}))]$ 
6:      $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$ 
7:      $w \leftarrow \text{clip}(w, -c, c)$ 
8:   end for
9:   Sample  $\{z^{(i)}\}_{i=1}^m \sim p(z)$  a batch of prior samples.
10:   $g_\theta \leftarrow -\nabla_\theta \frac{1}{m} \sum_{i=1}^m f_w(g_\theta(z^{(i)}))$ 
11:   $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$ 
12: end while
```

그림 72: WGAN Algorithm

- Compared to the original GAN algorithm, the WGAN undertakes the following changes:
 - After every gradient update on the critic function, clamp the weights to a small fixed range, $[-c, c]$.
 - Use a new loss function derived from the Wasserstein distance, no logarithm anymore. The “discriminator” model does not play as a direct critic but a helper for estimating the Wasserstein metric between real and generated data distribution.
 - Empirically the authors recommended RMSProp optimizer on the critic, rather than a momentum based optimizer such as Adam which could cause instability in the model training([이 포스트를 정리한 사람은 이부분에 대한 자세한 설명을 본적이 없다고 말함](#)).

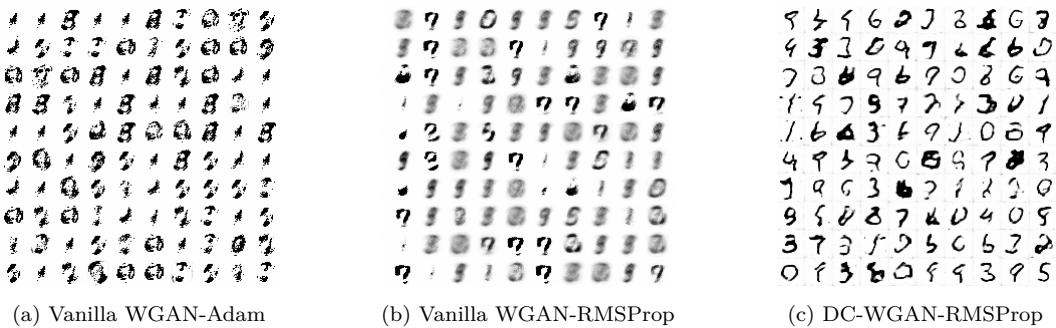


그림 73: WGAN 결과(100 epoch). Adam보다 RMSProp이 조금 나아보이지만 Vanilla GAN이라 결과가 좋지 않다. 제일 오른쪽은 DCGAN 구조의 WGAN에 대한 20 epoch 결과인데, epoch을 늘려도 좋아지지는 않는다.

- Sadly, Wasserstein GAN is not perfect. Even the authors of the original WGAN paper mentioned that “Weight clipping is a clearly terrible way to enforce a Lipschitz constraint”. WGAN still suffers from unstable training, slow convergence after weight clipping (when clipping window is too large), and vanishing gradients (when clipping window is too small).
- Some improvement, precisely replacing weight clipping with gradient penalty, has been discussed in Gulrajani et al(2017)⁵⁷. Gradient Penalty를 적용하면, clipping을 적용하지 않는다.

⁵⁷<https://arxiv.org/abs/1704.00028>

$$L = \underbrace{\mathbb{E}_{\hat{x} \sim \mathbb{P}_g} [D(\hat{x})] - \mathbb{E}_{x \sim \mathbb{P}_r} [D(x)]}_{\text{Original critic loss}} + \lambda \underbrace{\mathbb{E}_{\hat{x} \sim \mathbb{P}_{\hat{x}}} [(\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1)^2]}_{\text{Our gradient penalty}}$$

그림 74: WGAN Gradient Penalty

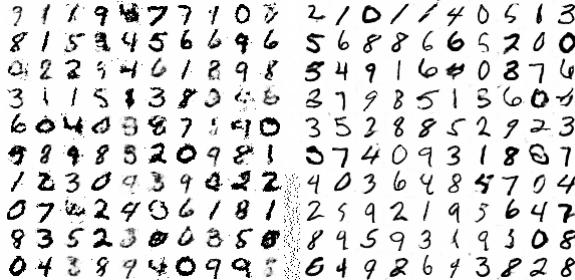


그림 75: WGAN Gradient Penalty 결과. 왼쪽은 Vanilla GAN(epoch 100), 오른쪽은 DCGAN(epoch 20)

- Gradient Penalty를 계산하는 과정에서 real image와 fake image를 interpolation하는데, fake image 대신 real image의 perturbation을 사용하는 DRAGAN(Deep Regret Analytic GAN)⁵⁸도 있다.

9.7 EBGAN & BEGAN

♠ EBGAN(2016)

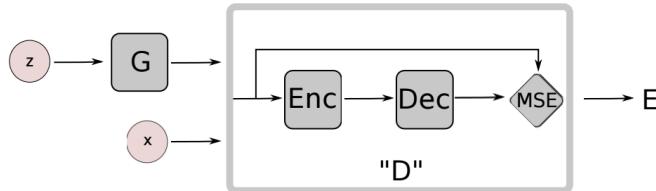


그림 76: The diagram of an EBGAN with an auto-encoder discriminator. The generator(G) takes random vector z as input, and transform it into a sample $G(z)$, for example an image. The discriminator(D), whose output is a scalar energy, takes either real or generated images, and estimates the energy value E accordingly where $E \in \mathbb{R}$.

- Energy-based Generative Adversarial Network⁵⁹.
- 보통의 GAN 모델은 Discriminator가 real image/fake image를 구분하는 $[0, 1]$ 값을 return하도록 설계되는데, EBGAN 모델은 Discriminator에 AutoEncoder 구조를 도입하고 return 값으로 pixel-wise Reconstruction Loss(L_2 -loss) 값을 돌려준다⁶⁰.
- 그래서 Discriminator를 Energy Function이라 부르고, real image에 대해서는 낮은 energy 값을, fake image

⁵⁸On Convergence and Stability of GANs, <https://arxiv.org/abs/1705.07215>

⁵⁹Energy-based Generative Adversarial Network, <https://arxiv.org/abs/1609.03126?context=cs>

⁶⁰논문에는 L_2 -loss(RMSE)로 써어 있고, 그림(76)에는 MSE라도 되어 있다. 구현한 code들에도 제곱이 되어 있는 경우와 그렇지 않은 경우가 모두 있다.

에서는 높은 energy 값을 return한다고 본다.

$$\begin{aligned}\text{Loss}_D(x, z) &= D(x) + [m - D(G(z))]^+, \\ \text{Loss}_G(z) &= D(G(z))\end{aligned}$$

- 논문에서는 이렇게 Loss Function을 잡으면 System의 Nash Equilibrium이 존재하며, 그 때의 Value Function의 값과 그것을 만족하는 Energy 값이 존재함을 수식적으로 증명하고 있다.
- PT(pulling-away term) or repelling regularizer: 논문에서는 GAN모델이 다양한 image를 학습하지 못하는 mode collapse에 빠지는 것을 방지하기 위해서 repelling regularizer를 도입하고 있다. To avoid mode collapse, EBGAN-PT, a variant of EBGAN, adds an additional cost function for the generator. S is the feature output from the encoder for the generated images. We compute the pulling-away term (PT) which measures the cosine similarity among all generated images features S in a mini-batch.

$$\text{Loss}_G(z) = D(G(z)) + \lambda f_{PT}, \text{ where } f_{PT} = \frac{1}{N(N-1)} \sum_{i=1}^N \sum_{j \neq i}^N \left(\frac{S_i^T \cdot S_j}{|S_i| \cdot |S_j|} \right)^2$$

- S 는 Discriminator의 Encoder의 output이다.



그림 77: EBGAN결과: vanilla 구조로는 잘 안됨. DCGAN 구조로도 training 시키는 것이 쉽지 않다. 여러번 시행착오를 반복하여 얻은 결과(20epoch). 왼쪽(Adam): (D loss learning rate = 0.00005), (G loss learning rate = 0.001), D 1번당 G를 5번 training. 가운데(RMSProp): 같은 조건으로 RMSProp 적용. 오른쪽(RMSProp): D, G에 같은 learning rate 0.001 적용. D 1번당 G 5번 training.

♠ BEGAN(2017)

- David Berthelot, Thomas Schumm, Luke Metz, Boundary Equilibrium Generative Adversarial Networks⁶¹.
- BEGAN은 quality가 좋은 image를 만들어 낸다.
- Discriminator로 EBGAN과 동일하게 auto-encoder 구조를 사용하면서, Discriminator를 Wasserstein Distance를 계산하는 critic으로 본다.

$$\begin{aligned}\text{Loss}_D(x, z) &= D(x) - D(G(z)), \\ \text{Loss}_G(z) &= D(G(z))\end{aligned}$$

- 이상적으로는 $D(x) = D(G(z))$ 가 되는 것이지만 실제 상황에서는 잘 이루어지지 않는 것이 GNA모델의

⁶¹Boundary Equilibrium Generative Adversarial Networks, <https://arxiv.org/abs/1703.10717?context=cs>

문제점으로 지적된다. 그래서 BEGAN에서는 diversity ratio γ 를 도입한다(Proportional Control Theory).

$$\gamma = \frac{D(G(z))}{D(x)}.$$

γ 값이 높으면 다양한 이미지를 생성하고, γ 이 낮으면 AutoEncoder의 이미지 복원에 집중하게 된다.

- BEGAN 모델에서의 최종 loss function은 다음과 같다.

$$\begin{aligned} \text{Loss}_D(x, z) &= D(x) - k_t D(G(z)), \\ \text{Loss}_G(z) &= D(G(z)), \\ k_{t+1} &= k_t + \lambda_k (\gamma D(x) - D(G(z))), \quad k_0 = 0, k_t \in [0, 1]. \end{aligned}$$

- 구현 상에서 γ, λ_k (learning rate for k)는 상수값이 주어진다.
- BEGAN 논문에서는 수렴 정도를 측정할 수 있는 하나의 지표, Global Convergence Measurement를 제안하고 있다.

$$M := D(x) + |\gamma D(x) - D(G(z))|$$

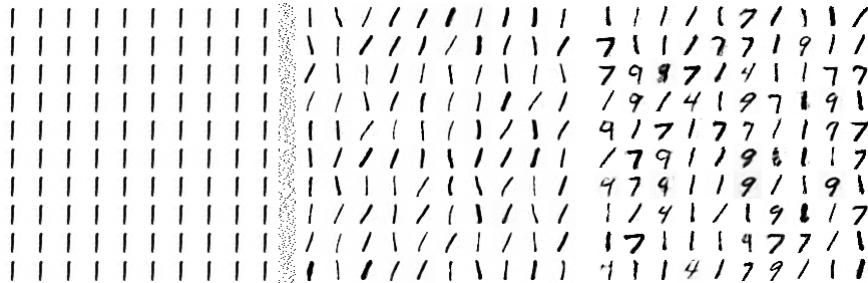


그림 78: Vanilla-BEGAN 결과. 원쪽(epoch 20): $\gamma = 0.25$, 가운데(epoch 20): $\gamma = 0.9$, 오른쪽(epoch 100): $\gamma = 0.9$, D:G=1:5 비율로 training



그림 79: DC-BEGAN 결과(epoch 20). 원쪽: $\gamma = 0.75$, 오른쪽: $\gamma = 0.9$

9.8 DCGAN

DCGAN은 2016년 1월에 발표된 논문 Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks - Alec Radford et al, 2016⁶²이다.

♠ Data 준비

- celebA Dataset⁶³: 156,253개의 얼굴이미지 파일을 얻을 수 있다. 사진 크기는 128×128 .
- lsun Dataset⁶⁴: 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.

♠ Transposed-Convolution(Deconvolution)

Deconvolution에 대한 설명은 이곳⁶⁵을 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.

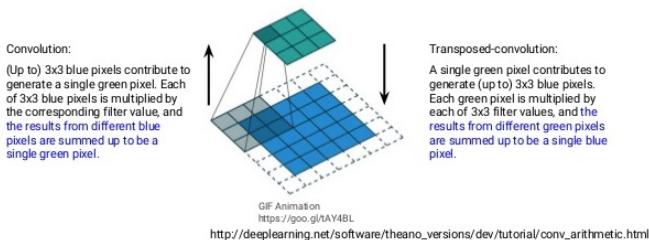


그림 80: Convolution & Transposed-Convolution

```
# 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]],
                  [[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 `padding` 조절된다.

⁶²<https://arxiv.org/abs/1511.06434>

⁶³https://www.dropbox.com/s/e0ig4nf1v94hyj8/CelebA_128crop_FD.zip?dl=0

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

⁶⁴<https://github.com/fyu/lsun>

⁶⁵<https://datascience.stackexchange.com/questions/6107/what-are-deconvolutional-layers>

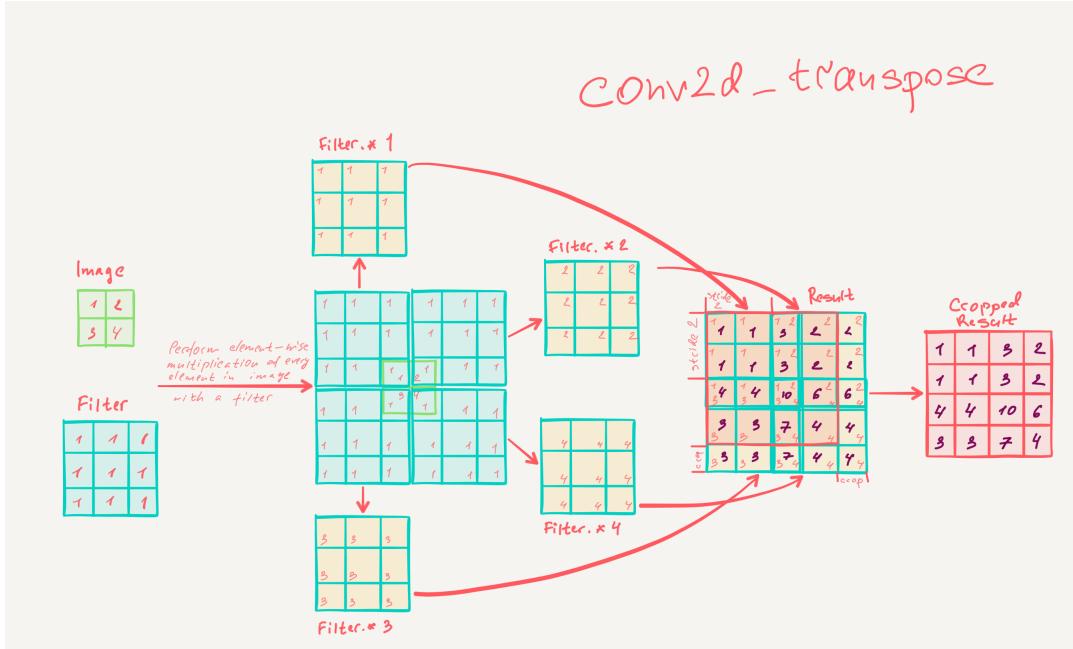


그림 81: Deconvolution 계산 과정

♠ DCGAN-CelebA 구조

DCGAN에 대한 설명과 여러가지 구현에 대한 것은 Brandon Amos의 Blog⁶⁶를 참조하면 된다.

먼저 살펴볼 구현⁶⁷은 논문 저자가 구현한 DCGAN⁶⁸ 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.

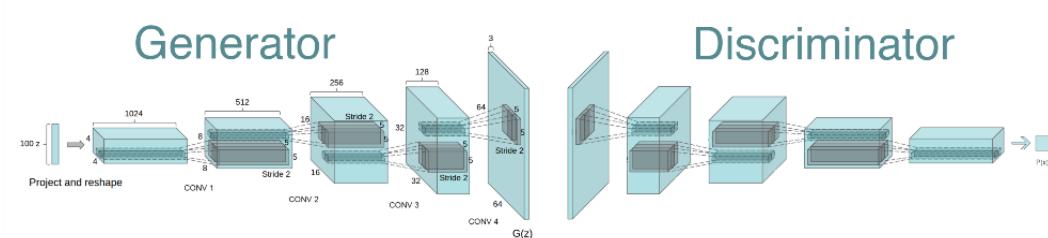


그림 82: DCGAN architecture

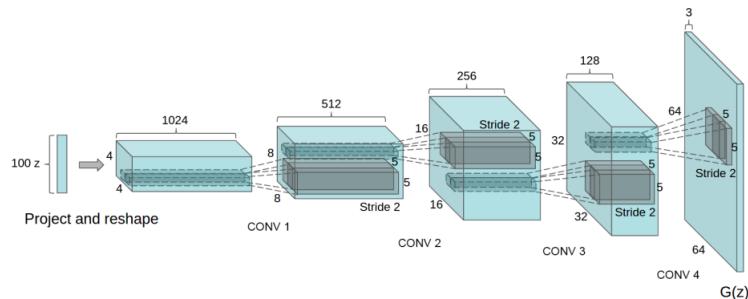


그림 83: DCGAN Generator used for LSUN scene modeling

⁶⁶<http://bamos.github.io/2016/08/09/deep-completion/>

⁶⁷https://github.com/HyeongminLEE/Tensorflow_DCGAN

⁶⁸https://github.com/Newmu/dcgan_code

All Tensors are 4D (include Batch_size Dimension)

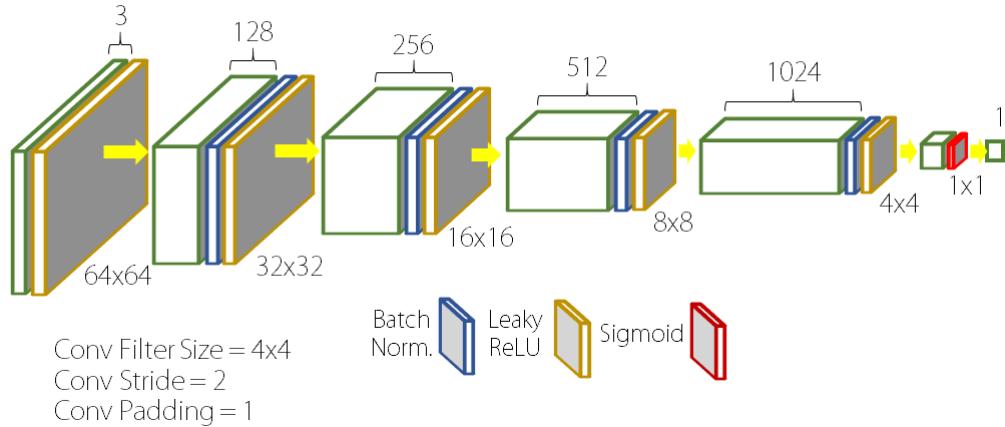


그림 84: DCGAN Discriminator

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(128×128)는 64×64 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 size가 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(128×128)를 (108×108) 크기로 자른 후, 다시

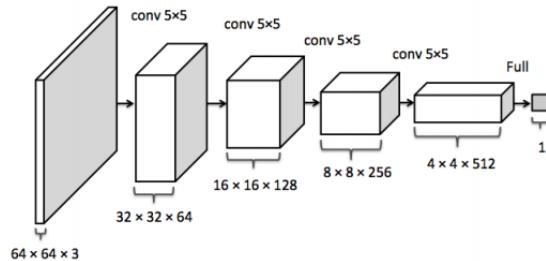


그림 85: DCGAN discriminator architecture

(64×64)로 resize했다. 그림(84)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.

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(128×128)는 64×64 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 size가 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	$N \times 8192$	8192×1	
Prob	$N \times 1$		

DCGAN Generator	Input/Output	Filter/Weight (FH, FW, out-channels, in-channels) out-shape	
FC	$N \times 100$	100×8192	
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 ture/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

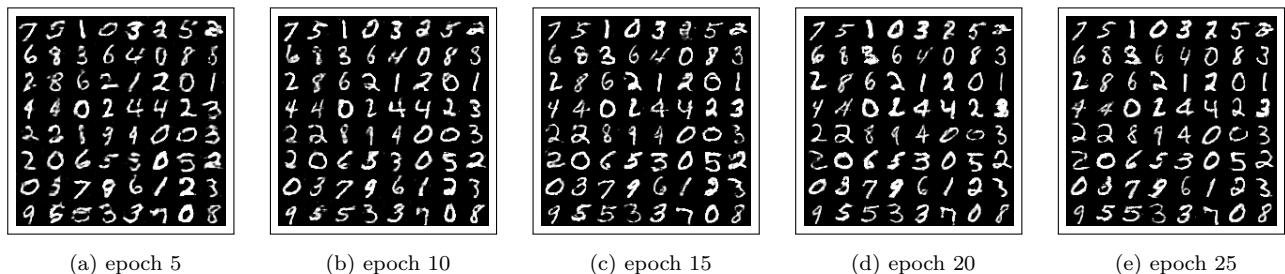


그림 86: Generated MNIST Images

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	$N \times 3636$	3636×1024	
concat label	$N \times 1024$		
FC, Sigmoid	$N \times 1034$	1034×1	
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	$N \times 110$	110×1024	
concat label	$N \times 1024$		
FC,BN,Relu	$N \times 1034$	1034×6272	$6272 = 7 \times 7 \times 128$
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	$N \times 28 \times 28 \times 1$		

9.9 StackGAN(2016)

Han Zhang, Tao Xu, Hongsheng Li, Shaoting Zhang, Xiaogang Wang, Xiaolei Huang, Dimitris Metaxas, StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks⁶⁹

♠ Data 준비 & 실행

- 저자의 구현 코드⁷⁰에 Data(bird, flower)를 다운받을 수 있게 링크되어 있다.
- download the pretrained char-CNN-RNN text encoders and extract text embeddings.
- Download the birds and flowers image data.
- preprocess_birds.py를 실행하면 (train,test) 각각의 filenames.pickle를 읽어, 76images.pickle, 304images.pickle을 각각 만든다.
 - 새 image는 모두 11,788개로 200개의 카테고리로 나누어져 있는데, 대부분 60개씩 들어 있지만 60개보다 적은 경우도 있다.
 - images.txt, bounding_boxes.txt에 있는 파일명과 새의 위치 정보를 이용하여 (76×76) , (304×304) 를 만들어 ‘76images.pickle’, ‘304images.pickle’, 파일을 만든다. 각각 stageI, stageII에서 사용한다.
 - pickle 파일에 저장되는 값은 정수값이다. training할 때, batch data를 가져오는 단계에서 $[-1, 1]$ 값으로 변환한다. 그렇기 때문에 generator의 마지막 activation function은 tanh.
 - 실제로 training할 때는 이미지 size가 (256×256) , (64×64) 이되는데, 이는 (304×304) , (76×76) 으로부터 random하게 이미지를 crop해서 사용하기 때문이다. 그리고, 확률 $\frac{1}{2}$ 로 이미지를 좌우 반전시킨다⁷¹. stageII에서 필요한 (64×64) 이미지는 76images.pickle을 load하지 않고, (256×256) 이미지를 resize(tf.image.resize_bilinear)해서 사용한다.

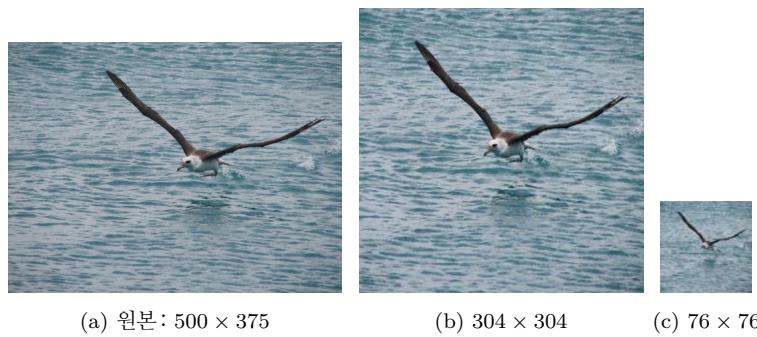


그림 87: Bird image Sample. 원본 이미지들의 크기는 일정하지 않다.

- embedding-type은 2가지를 가지고 했다. char-CNN-RNN-embeddings.pickle(downlaod 가능) 또는 skip-thought-embeddings.pickle(제공되지 않음. 그렇지만, theano로 구현된 code로 만들어 보면, 1.66G(train), 549M(test) 크기의 pickle파일이 만들어진다.)

- char-CNN-RNN-embeddings.pickle을 load해 보면, $(8855, 10, 1024)$ numpy array가 나오는데, 이미지 1

⁶⁹<https://arxiv.org/abs/1612.03242>

⁷⁰<https://github.com/hanzhanggit/StackGAN>

⁷¹numpy의 변환 함수: numpy.fliplr(좌우), numpy.flipud(상하), numpy.rot90(회전)

개당 10개의 caption이 들어있기 때문이다. CNN-RNN embedding dim은 1,024임을 알 수 있다. 참고로 skip-thought embedding dim은 4,800을 사용한다.

- training 할 때, 10개 중에서 random하게 NUM_EMBEDDING = 4를 평균내어 사용한다.
- stageII에서는 plain text를 char-CNN-RNN(lua code)과 skip-thought(theano로 구현되어 있어 python 환경에서 실행 가능)로 encoding하여 돌려볼 수 있다.
- StackGAN 변수 개수는 약 93M.

dir	filename	설명
Data/birds/train	char-CNN-RNN-embeddings.pickle	shape: (8855, 10, 1024)
Data/birds/train	class_info.pickle	filename에 대한 class. integer list
Data/birds/train	filenames.pickle	image파일 11,788 개 중 8855개의 caption filename. e.g. '002.Laysan_Albatross/Laysan_Albatross_0002_1027' 확장자(.txt) 생략
Data/birds/train	class_info.pickle	각 파일이 200개 class중에 어디에 있는지...
Data/birds/test	char-CNN-RNN-embeddings.pickle	
Data/birds/test	class_info.pickle	filename에 대한 class. integer list.
Data/birds/test	filenames.pickle	image파일 11,788 개 중 2933개의 caption filename
Data/birds/test	class_info.pickle	
models	birds_model_164000.ckpt	stageII에 대한 Pretrained Model stageI에 대한 Pretrained Model은 없음
models	birds_skip_thought_model_164000.ckpt	plain text를 encoding하는 theano code와 연계하여 돌려 볼 수 있다.
Data/birds/text_c10	200개의 directory가 있고, 각각 50 ~ 60개의 text파일	각 파일은 10개의 caption을 가지고 있다.

표 6: StackGAN 모델 실행에 필요한 파일 설명

♠ StackGAN구조: stageI, II

- stageI과 stageII로 나누어져 있다. stageI 부분의 weight를 먼저 training하고, 그 결과를 받아서 stageII를 training하는 구조. stageII는 stageI의 모델 구조를 포함하고 있기 때문에, stageII를 독립적으로 training할 수 있다. 즉, stageI의 ouput인 저해상도 image를 stageII에 입력으로 넣는 것은 아니다. stageII가 stageI의 모델 구조를 포함하고 있기 때문에, stageII 자체적으로 저해상도 image를 생성해서 고해상도 image를 만들고 있다.
- 다운받은 모델 파일 'birds_model_164000.ckpt'은 stageII를 training해서 만든 파일. batch size 64로 164,000 iteration 된 결과인데, epoch으로 보면 1,188 epoch에 해당된다. 모델 설정 파일을 보면, stageI의 82,000 iteration(약 594 epoch) 결과를 받아 stageII에서 164,000 iteration 한 것으로 추정된다(stageI만 training한 모델 파일은 없음).
- stageII의 모델 파일은 stageI 모델을 포함하고 있다. 다운받은 stageII모델에서 stageI의 전체 variable을 load

하면 error가 발생하지만⁷², trainable variable만 load는 된다.

- trainable variable만 load하면 batch-normalization의 moving-mean, variance를 가져오지 못하는 문제가 있다. 그래서 직접 10,500(76epoch)를 training 시킨 후, trainable variable을 제외한 나머지 variable은 여기서 load하고, trainable variable은 다운받은 'birds_model_164000.ckpt'에서 가져 온 후, 몇 epoch을 더 training 시켜 자체적인 stageI 모델 파일 ('model_164500.ckpt')을 생성했다.
- 다운받은 ckpt파일 load과정에서 오류를 발생시키는 variable들은 제외시키고 load하여 몇 epoch을 더 수행해서, 자체적인 stageI 모델 파일 ('model_164565.ckpt')을 생성했다.

```
restore_vars = tf.global_variables()
saver = tf.train.Saver(restore_vars)
saver.restore(sess, './stageI/model_10500.ckpt')

restore_vars = tf.trainable_variables()
saver = tf.train.Saver(restore_vars)
saver.restore(sess, './stageII/birds_model_164000.ckpt')
```

- stageII의 코드를 보면, stageI 모델 파일을 load하는 구조로 되어 있다. stageII의 모델 파일을 로드 후 이어서 training하는 구조는 아니다. 이어서 training하려면 코드를 일부 수정해야 한다.
- stageII에서 다운받은 파일 ('birds_model_164000.ckpt')로부터 전체 variable을 load하면 stageI에서와 동일하게 error발생(tensorflow 버전 차이로 추정).
- 파일 ('birds_model_164000.ckpt')로 부터 stageII를 batch size 64로 실행하면 메모리 부족(12G memory pc)으로 돌지 않는다. batch size 32로 줄이면 도는 경우도 있음.

♠ StackGAN 모델 구조-StageI

Conditioning Augmentation C_1 :

```
embedding: batch_size x embedding_shape( $N \times 1024$ ) -> FC( $2 \times 128$ ) -> lrelu(0.2)
->  $m(N \times 128)$ ,  $\log \sigma(N \times 128)$ 
-> sampling  $c \sim N(m, \sigma)$ , compute KL-loss with  $N(0, 1)$ ,  $KL(N(m, \sigma) | N(0, 1))$ 
```

- StackGAN 구현 코드 전체적으로 conv2d, deconv2d에서 bias term을 사용하지 않고 있다. Batch Normalization을 사용하기 때문에 bias term이 불필요하다.
- stageI은 3개의 block을 가지고 있다. text-encoder, generator, discriminator.
 - text-encoder, cnn-rnn-embedding된 data를 받아 VAE같이 평균, 분산을 찾고 이로부터 random sampling한다. 그리고, KL divergence도 구한다.
 - generator는 simple version과 default version이 있다. simple version은 deconv2d를 사용하고, default version은 image resize와 conv2d를 사용한다.

⁷²non trainable variable인 mean/biase, mean/local-step, variance/biased, variance/local-step 이런 변수가 다운받은 ckpt파일에는 없다. 직접 training하여 ckpt파일로 저장하면 이런 variable이 저장되어 있다. 따라서 tensorflow 버전 차이로 발생하는 문제. 이 2종류의 variable들만 제외한 나머지 모든 변수를 load하여 training을 이어갈 수도 있다. skip-thought embedding으로 demo image를 생성(birds-

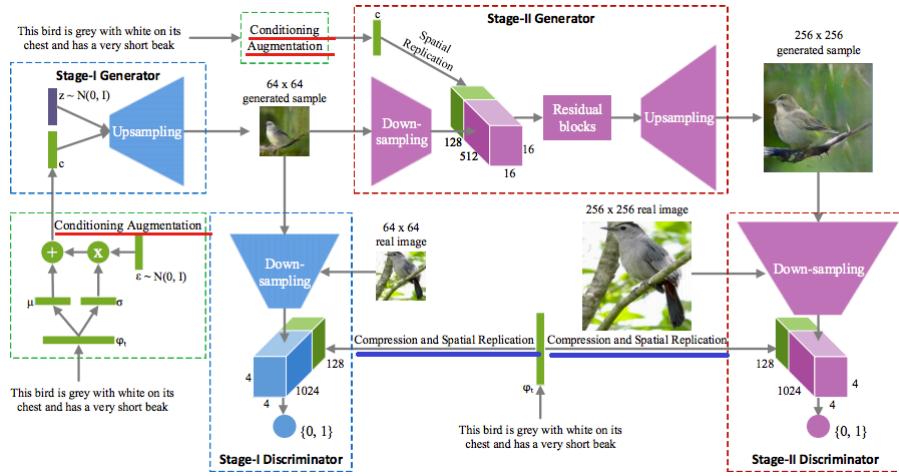


그림 88: StackGAN 모델 구조: 1. 빨간색으로 밑줄친 Conditioning Augmentation은 같은 구조를 가지지만 Variable들을 공유하지는 않는다.

2. 마찬가지로 파란색으로 밑줄친 Compression & Spatial Replication도 같은 구조이지만 Variable들을 공유하지는 않는다.
3. StageI, II의 Discriminator 마지막 부분(파란색, 분홍색 원으로 표시)도 동일한 구조이고 Variable들을 공유하는 아니다.

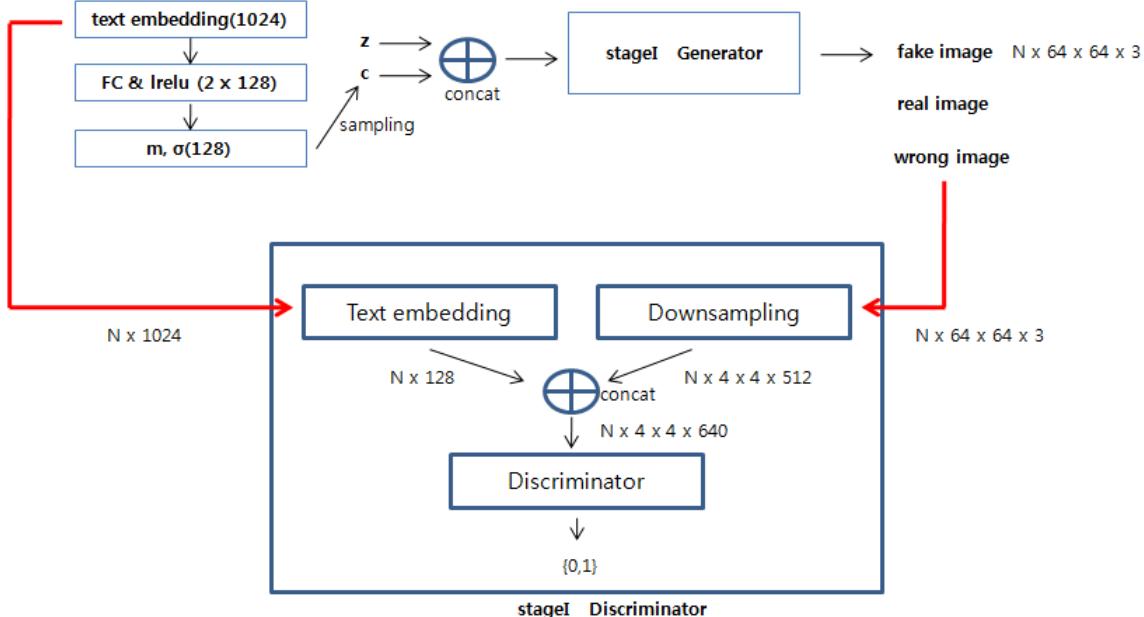


그림 89: StackGAN StageI 모델 구조

- cnn-rnn encoding을 통해 embedding data를 받아, embedding latent space $N(m, \sigma)$ 를 구하고, 이로 부터 random sampling을 한다. 그리고 $N(m, \sigma)$ 에 대해서는 $N(0, 1)$ 과 KL-Divergence를 계산하여 KL-loss를 구한다.
- sampling한 c 는 generator의 latent vector z 와 concat하여 Generator의 입력으로 들어간다.
- Generator는 simple, default 2가지 버전이 있다. simple version은 보통의 generator처럼 deconv2d를 이용하는 반면, default 버전은 image resize와 conv2d를 이용하여 이미지를 키워 나간다.

skip-thought-demo.py) 하는 코드에서는 이 variable들이 사용되지 않기 때문에 다운받은 ckpt 파일로 실행이 된다.

```

Generator(Upsampling) - simple version  $G_1$ 
 $c(N \times 128), z(N \times 100) \rightarrow \text{concat}(c, z)$ 
 $\rightarrow \text{FC}(4 \times 4 \times (128 \times 8)) \rightarrow \text{reshape}(N, 4, 4, 128 \times 8) \rightarrow \text{batch-normalization} \rightarrow \text{relu}$ 
 $\rightarrow \text{deconv2d}(f=128 \times 4, k=4, s=2, p='s') \rightarrow \text{batch-normalization} \rightarrow \text{relu}(N, 8, 8, 128 \times 4)$ 
 $\rightarrow \text{deconv2d}(f=128 \times 2, k=4, s=2, p='s') \rightarrow \text{batch-normalization} \rightarrow \text{relu}(N, 16, 16, 128 \times 2)$ 
 $\rightarrow \text{deconv2d}(f=128, k=4, s=2, p='s') \rightarrow \text{batch-normalization} \rightarrow \text{relu}(N, 32, 32, 128)$ 
 $\rightarrow \text{deconv2d}(f=3, k=4, s=2, p='s') \rightarrow \text{tanh} = \text{fake\_image}(N, 64, 64, 3)$ 

```

```

Generator(Upsampling) - default version  $G_1$ 
 $c(N \times 128), z(N \times 100) \rightarrow \text{concat}(c, z) \rightarrow \text{FC}(4 \times 4 \times (128 \times 8))$ 
 $\rightarrow \text{BN} \rightarrow \text{reshape}(N, 4, 4, 128 \times 8) =: \text{node1\_0}$ 

 $\text{node1\_0} \rightarrow \text{conv2d}(f=128 \times 2, k=1, s=1, p='s') \rightarrow \text{BN} \rightarrow \text{relu}$ 
 $\rightarrow \text{conv2d}(f=128 \times 2, k=3, s=1, p='s') \rightarrow \text{BN} \rightarrow \text{relu}$ 
 $\rightarrow \text{conv2d}(f=128 \times 8, k=3, s=1, p='s') \rightarrow \text{BN} =: \text{node1\_1}(N, 4, 4, 128 \times 8)$ 

 $\text{node1\_0} + \text{node1\_1} \rightarrow \text{relu} =: \text{node1}(N, 4, 4, 128 \times 8)$ 

 $\text{node1} \rightarrow \text{tf.image.resize_nearest_neighbor}(N, 8, 8, 128 \times 8)$ 
 $\rightarrow \text{conv2d}(f=128 \times 4, k=3, s=1, p='s') \rightarrow \text{BN} =: \text{node2\_0}(N, 8, 8, 128 \times 4)$ 

 $\text{node2\_0} \rightarrow \text{conv2d}(f=128, k=1, s=1, p='s') \rightarrow \text{BN} \rightarrow \text{relu}$ 
 $\rightarrow \text{conv2d}(f=128, k=3, s=1, p='s') \rightarrow \text{BN} \rightarrow \text{relu}$ 
 $\rightarrow \text{conv2d}(f=128 \times 4, k=3, s=1, p='s') \rightarrow \text{BN} =: \text{node2\_1}(N, 8, 8, 128 \times 4)$ 

 $\text{node2\_0} + \text{node2\_1} \rightarrow \text{relu} =: \text{node2}(N, 4, 4, 128 \times 4)$ 

 $\text{node2} \rightarrow \text{tf.image.resize_nearest_neighbor}(N, 16, 16, 128 \times 4)$ 
 $\rightarrow \text{conv2d}(f=128 \times 2, k=3, s=1, p='s') \rightarrow \text{BN} \rightarrow \text{relu}$ 
 $\rightarrow \text{tf.image.resize_nearest_neighbor}(N, 32, 32, 128 \times 2)$ 
 $\rightarrow \text{conv2d}(f=128, k=3, s=1, p='s') \rightarrow \text{BN} \rightarrow \text{relu}$ 
 $\rightarrow \text{tf.image.resize_nearest_neighbor}(N, 64, 64, 128)$ 
 $\rightarrow \text{conv2d}(f=3, k=3, s=1, p='s') \rightarrow \text{tanh} = \text{fake\_image}(N, 64, 64, 3)$ 

```

- Discriminator는 image(real-image 또는 fake-image 또는 wrong-image)와 text embedding($N \times 1024$)을 입력으로 받는다. Discriminator는 3개의 block을 구성되는데, image를 Downsampling하는 부분(simple, default 2가지 version), text embedding을 처리하는 부분, 그리고 이 2가지 결과를 concat해서 최종적으로 처리하는 부분이다.

```

Discriminator(Text Embedding),  $D_1$  일부(Compression & Spatial Replication)
embedding( $N, 1024$ )
 $\rightarrow \text{FC}(128) \rightarrow \text{lrelu}(0.2) \rightarrow \text{reshape}(N, 4, 4, 128) =: c\_code$ 

```

Discriminator(Downsampling) - simple version, D_1 일부

```
image(N, 64, 64, 3)
-> conv2d(f=64, k=4, s=2, p='s') -> lrelu(0.2) (N, 32, 32, 64)
-> conv2d(f=64 × 2, k=4, s=2, p='s') -> BN -> lrelu(0.2) (N, 16, 16, 64 × 2)
-> conv2d(f=64 × 4, k=4, s=2, p='s') -> BN -> lrelu(0.2) (N, 8, 8, 64 × 4)
-> conv2d(f=64 × 8, k=4, s=2, p='s') -> BN -> lrelu(0.2) (N, 4, 4, 64 × 8) =: x_code
```

Discriminator(Downsampling) - default version, D_1 일부

```
image(N, 64, 64, 3)
-> conv2d(f=64, k=4, s=2, p='s') -> lrelu(0.2) (N, 32, 32, 64)
-> conv2d(f=64 × 2, k=4, s=2, p='s') -> BN -> lrelu(0.2) (N, 16, 16, 64 × 2)
-> conv2d(f=64 × 4, k=4, s=2, p='s') -> BN(N, 8, 8, 64 × 4)
-> conv2d(f=64 × 8, k=4, s=2, p='s') -> BN(N, 4, 4, 64 × 8) =: node1_0

node1_0
-> conv2d(f=64 × 2, k=1, s=1, p='s') -> BN -> lrelu(0.2) (N, 4, 4, 64 × 2)
-> conv2d(f=64 × 2, k=3, s=1, p='s') -> BN -> lrelu(0.2) (N, 4, 4, 64 × 2)
-> conv2d(f=64 × 8, k=3, s=1, p='s') -> BN (N, 4, 4, 64 × 8) = node1_1

node1_0 + node1_1 -> lrelu(0.2) =: x_code(N, 4, 4, 64 × 8)
```

- 이제, Discriminator의 앞단에서 처리된 `x_code`, `c_code`(spatial Replication 필요)를 concat해서 Discriminator의 마지막 부분을 처리하면 된다.

Discriminator(discriminator), D_1 last block

```
x_code(N, 4, 4, 512), c_code(N, 4, 4, 128)
-> concat(x_code, c_code) (N, 4, 4, 640)
-> conv2d(f=64 × 8, k=1, s=1, p='s') -> BN -> lrelu(0.2) (N, 4, 4, 64 × 8)
-> conv2d(f=1, k=4, s=4, p='s') -> sigmoid (N, 1, 1, 1)
```

- stageI의 loss function이 대하여 살펴보자. stageI의 loss function의 일반적인 GAN모델의 loss function과 동일하다.
- Conditional Augmentation C_1 , stageI Generator G_1 , stageI Discriminator D_1 .
- text embedding e , latent vector $z \sim N(0, 1)$, real image X_l , wrong image \bar{X}_l . 여기서 wrong image는 text embedding과 무관한 real image를 말한다.
- λ 는 KL loss에 대한 weight(eg. 2.0)

$$\begin{aligned} c_1, \text{Loss}_{KL_1} &= C_1(e) \\ \text{fake-image} &= G_1(z, c_1) \\ \text{Loss}_{D_1} &= -\left[\log D_1(X_l, e) + \frac{1}{2} \log \left(1 - D_1(G_1(z, c_1), e) \right) + \frac{1}{2} \log \left(1 - D_1(\bar{X}_l, e) \right) \right] \\ \text{Loss}_{G_1} &= -\log D_1(G_1(z, c_1), e) + \lambda \text{Loss}_{KL_1} \end{aligned}$$

- optimizer는 Adam optimizer를 사용하였고, learning rate는 0.0002, $\beta_1 = 0.5$ 로 설정했고, 일정 epoch(eg. 100)마다 learning rate을 반으로 줄이고 있다.

♠ StackGAN 모델 구조-StageII

- stageII는 stageI의 weight 값을 받아, training을 이어간다.
- 먼저 Generator G_2 를 살펴보자. G_2 의 앞부분은 D_1 에서처럼 Downsampling block이 있다.

G_2 의 Downsampling block:

```
fake_image: (N, 64, 64, 3)
-> conv2d(f=128, k=3, s=1, p='s') -> relu (N, 64, 64, 128)
-> conv2d(f=128 × 2, k=4, s=2, p='s') -> BN -> relu (N, 32, 32, 128 × 2)
-> conv2d(f=128 × 4, k=4, s=2, p='s') -> BN -> relu (N, 16, 16, 128 × 4) =: x_code
```

```
c_code =  $C_2(e)$ : (N, 128) -> spatial replication (N, 16, 16, 128) := c_code0
```

G_2 의 Residual block:

```
concat(x_code, c_code0): (N, 16, 16, 640)
-> conv2d(f=128 × 4, k=3, s=1, p='s') -> BN -> relu (N, 16, 16, 128 × 4) = n0
```

n0

```
-> conv2d(f=128 × 4, k=3, s=1, p='s') -> BN -> relu (N, 16, 16, 128 × 4)
-> conv2d(f=128 × 4, k=3, s=1, p='s') -> BN := n0_1 (N, 16, 16, 128 × 4)
```

```
n0 + n0_1 -> relu := n1 (N, 16, 16, 128 × 4)
```

같은 방법 반복, Variable 공유는 아님.

```
n0 -> n1 -> n2 -> n3 -> n4 (N, 16, 16, 128 × 4)
```

- The encoded image features coupled with text features are fed into several residual blocks, which are designed to learn multi-modal representations across image and text features.
- residual block을 거친 후, Generator의 마지막 단계인 Upsampling block을 거치면 최종 high resolution image 가 생성된다.

G_2 의 Upsampling block

```

n4: (N, 16, 16, 128 × 4)
-> tf.image.resize_nearest_neighbor(N, 32, 32, 128 × 4)
-> conv2d(f=128 × 2, k=3, s=1, p='s') -> BN -> relu (N, 32, 32, 128 × 2)

-> tf.image.resize_nearest_neighbor(N, 64, 64, 128 × 2)
-> conv2d(f=128, k=3, s=1, p='s') -> BN -> relu (N, 64, 64, 128)

-> tf.image.resize_nearest_neighbor(N, 128, 128, 128)
-> conv2d(f=64, k=3, s=1, p='s') -> BN -> relu (N, 128, 128, 64)

-> tf.image.resize_nearest_neighbor(N, 256, 256, 64)
-> conv2d(f=32, k=3, s=1, p='s') -> BN -> relu (N, 256, 256, 32)

-> conv2d(f=3, k=3, s=1, p='s') -> tanh

```

- 이제 stageII의 마지막 단계인 Discriminator D_2 만 남았다. D_2 는 D_1 과 유사하다. text embedding e 는 D_1 에서와 동일하게 Compression & Spatial Replication 과정을 거친다.
- D_1 에서의 Downsampling block은 2가지 (simple, default) 버전이 있었는데, D_2 의 Downsampling은 1가지만 있다.
- D_2 의 마지막 block인 discriminator block은 D_1 의 것과 동일하다 (Variables 공유는 아님).

```

Discriminator(Text Embedding),  $D_2$  일부(Compression & Spatial Replication)
embedding( $N$ , 1024)
-> FC(128) -> lrelu(0.2) -> reshape( $N$ , 4, 4, 128) =: c_code

```

```

Discriminator(Downsampling) -  $D_2$  Downampling block
image( $N$ , 256, 256, 3)
-> conv2d(f=64, k=4, s=2, p='s') -> lrelu(0.2) ( $N$ , 128, 128, 64)
-> conv2d(f=64 × 2, k=4, s=2, p='s') -> BN -> lrelu(0.2) ( $N$ , 64, 64, 64 × 2)
-> conv2d(f=64 × 4, k=4, s=2, p='s') -> BN-> lrelu(0.2) ( $N$ , 32, 32, 64 × 4)
-> conv2d(f=64 × 8, k=4, s=2, p='s') -> BN-> lrelu(0.2) ( $N$ , 16, 16, 64 × 8)
-> conv2d(f=64 × 16, k=4, s=2, p='s') -> BN-> lrelu(0.2) ( $N$ , 8, 8, 64 × 16)
-> conv2d(f=64 × 32, k=4, s=2, p='s') -> BN-> lrelu(0.2) ( $N$ , 4, 4, 64 × 32)
-> conv2d(f=64 × 16, k=1, s=1, p='s') -> BN-> lrelu(0.2) ( $N$ , 4, 4, 64 × 16)
-> conv2d(f=64 × 8, k=1, s=1, p='s') -> BN( $N$ , 4, 4, 64 × 8) =: node1_0

```

```

node1_0
-> conv2d(f=64 × 2, k=1, s=1, p='s') -> BN -> lrelu(0.2) ( $N$ , 4, 4, 64 × 2)
-> conv2d(f=64 × 2, k=3, s=1, p='s') -> BN -> lrelu(0.2) ( $N$ , 4, 4, 64 × 2)
-> conv2d(f=64 × 8, k=3, s=1, p='s') -> BN ( $N$ , 4, 4, 64 × 8) = node1_1

```

```
node1_0 + node1_1 -> lrelu(0.2) =: x_code( $N$ , 4, 4, 64 × 8)
```

```

Discriminator(discriminator),  $D_2$  last block
x_code( $N$ , 4, 4, 512), c_code( $N$ , 4, 4, 128)
-> concat(x_code, c_code) ( $N$ , 4, 4, 640)
-> conv2d(f=64 × 8, k=1, s=1, p='s') -> BN -> lrelu(0.2) ( $N$ , 4, 4, 64 × 8)
-> conv2d(f=1, k=4, s=4, p='s') -> sigmoid ( $N$ , 1, 1, 1)

```

♠ StackGAN 모델 정리 & Loss Function

- StackGAN은 skip connection(Residual Block)을 많이 활용하고 있다.
- stageI에서 image를 생성하는 Upsampling 과정을 보면, simple, default 2가지 버전 모두, image size의 변화 과정은 다음과 같은 경향을 가진다.

$$(N, 4, 4, 1024) \rightarrow (N, 8, 8, 512) \rightarrow (N, 16, 16, 256) \rightarrow (N, 32, 32, 128) \rightarrow (N, 64, 64, 3)$$

- stageII에서 training은 G_2, D_2 에 대한 것이지만, Finetuning mode에서는 stageI의 G_1, D_1 도 같이 training 한다 (learning rate $\frac{1}{10}$).
- Conditional Augmentation C_1, C_2 , Generator G_1, G_2 , Discriminator D_1, D_2 .
- text embedding e , latent vector $z \sim N(0, 1)$, low resolution real image X_l , high resolution real image X_h , low resolution wrong image \bar{X}_l , high resolution wrong image \bar{X}_h . 여기서 wrong image는 text embedding

과 무관한 real image를 말한다.

$$\begin{aligned}
 c_1, \text{Loss}_{KL_1} &= C_1(e) \\
 Y_l &= G_1(z, c_1), \text{ lr-fake-image} \\
 \text{Loss}_{D_1} &= -\left[\log D_1(X_l, e) + \frac{1}{2} \log \left(1 - D_1(Y_l, e)\right) + \frac{1}{2} \log \left(1 - D_1(\bar{X}_l, e)\right) \right] \\
 \text{Loss}_{G_1} &= -\log D_1(Y_l, e) + \lambda \text{Loss}_{KL_1} \\
 c_2, \text{Loss}_{KL_2} &= C_2(e) \\
 Y_h &= G_2(Y_l, c_2), \text{ hr-fake-image} \\
 \text{Loss}_{D_2} &= -\left[\log D_2(X_h, e) + \frac{1}{2} \log \left(1 - D_2(Y_h, e)\right) + \frac{1}{2} \log \left(1 - D_2(\bar{X}_h, e)\right) \right] \\
 \text{Loss}_{G_2} &= -\log D_2(Y_h, e) + \lambda \text{Loss}_{KL_2}
 \end{aligned}$$

- optimizer는 Adam optimizer를 사용하였고, learning rate는 0.0002, $\beta_1 = 0.5$ 로 설정했고, 일정 epoch(eg. 100)마다 learning rate을 반으로 줄이고 있다.

9.10 Texture Synthesis(2015)

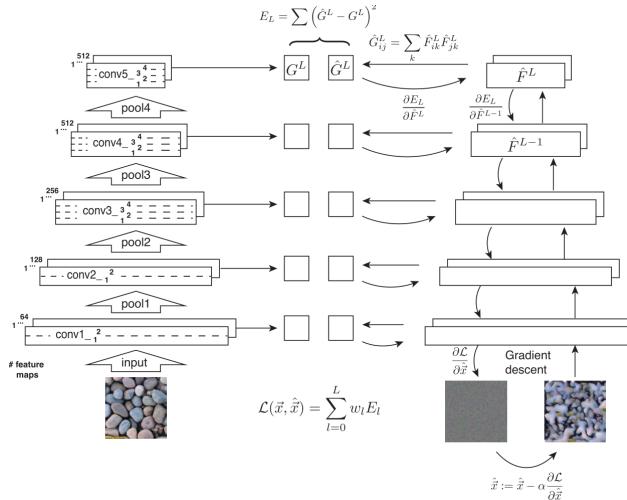


그림 90: Texture synthesis method

- Gatys, Leon A., Alexander S. Ecker, and Matthias Bethge. Texture Synthesis Using Convolutional Neural Networks, 2015⁷³
- VGG19 network에서 MaxPooling을 AvgPooling으로 바꾸어 사용한다.
- Loss L 은 texture loss(L_t), norm loss(L_n), total variance loss(L_{tv}) 3개의 합으로 이루어진다. 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 (13)과 동일하게 Gram matrix로 변환하여 계산한다.
- norm loss는 X 의 모든 원소들의 제곱 합.
- total variance loss는 total variance denosing이라고도 하는데, 식 (14)에서 정의되어 있다.

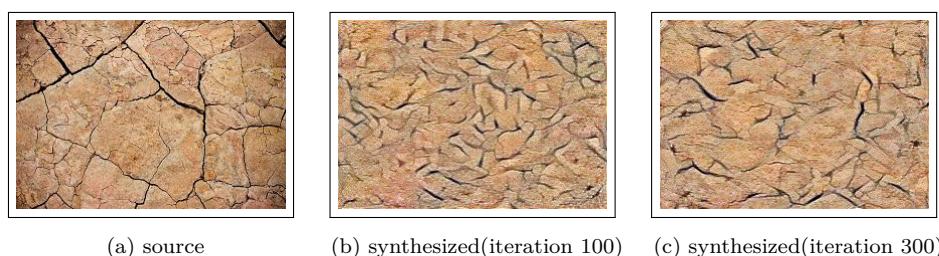


그림 91: Examples of synthesized texture

⁷³<https://arxiv.org/abs/1505.07376>

9.11 Neural Style(2017)

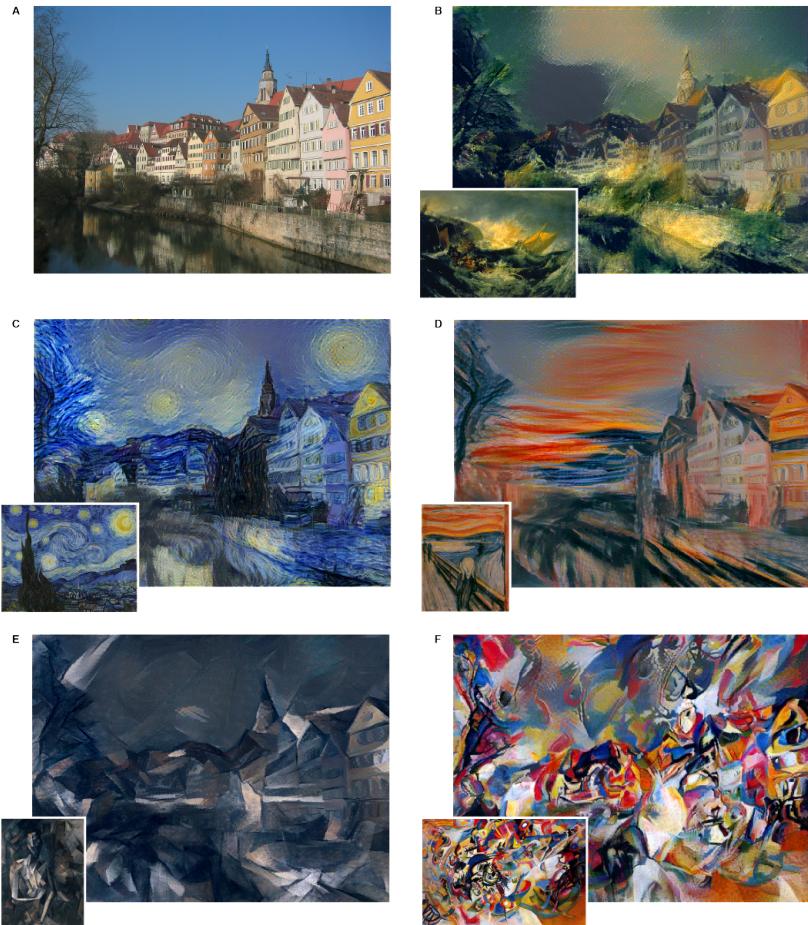


그림 92: Images that combine the content of a photograph with the style of several well-known artworks. The images were created by finding an image that simultaneously matches the content representation of the photograph and the style representation of the artwork (see Methods). The original photograph 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 Bethge⁷⁴이 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.
 - Mahendran, Aravindh, and Andrea Vedaldi. Understanding deep image representations by inverting

⁷⁴<https://arxiv.org/abs/1508.06576>

them, 2014⁷⁵. -> 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 ℓ compares the image representation $\Phi(\mathbf{x})$ to the target one Φ_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, 2015⁷⁶.

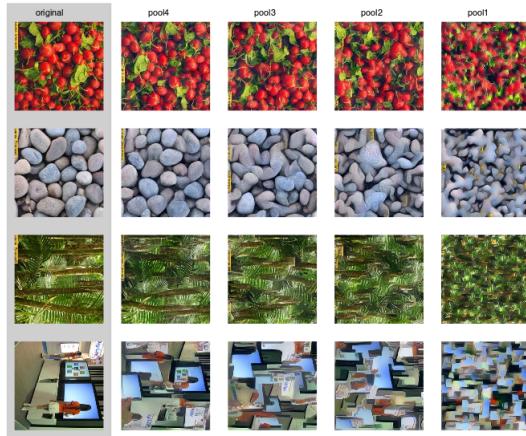


그림 93: Texture synthesis

- Nguyen, Anh, Jason Yosinski, and Jeff Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images, 2014⁷⁷.

- 이제 이 논문에서 제시하는 방법을 구체적으로 살펴보자. 그림(95)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)에 feature map 들 중 일부를 뽑아내게 된다. style과 content는 각각 서로 다른 feature map을 뽑아 낸다.
- 먼저, content loss를 구해보자. C, X 를 VGG19 network(feature extractor⁷⁸)으로 입력하여 layer($\ell \in \text{CONTENT-LAYERS} = \text{CL}$)의 featrue map을 뽑아 낸다. C 에서 뽑아낸 feature map을 P^ℓ , X 에서 뽑아낸 feature map을 F^ℓ 로 표시하자. 뽑아낸 C, X 의 feature map의 L_2 -norm을 계산하여 합한다. 합쳐진 값(w_c^ℓ -weighted sum)이 content loss L_c 이다. 단, L_2 -norm은 2차원 행렬이 아니라, 4차원 array를 대상으로 한 것이다.

$$L_c = \sum_{\ell \in \text{CL}} w_c^\ell \|P^\ell - F^\ell\|^2 \times \lambda_c^\ell \quad (12)$$

⁷⁵<https://arxiv.org/abs/1412.0035>

⁷⁶<https://arxiv.org/abs/1505.07376>

⁷⁷<https://arxiv.org/abs/1412.1897>

⁷⁸cs231n에서는 연습용의 작은 모델로 SqueezeNet을 feature extractor로 사용하고 있다.

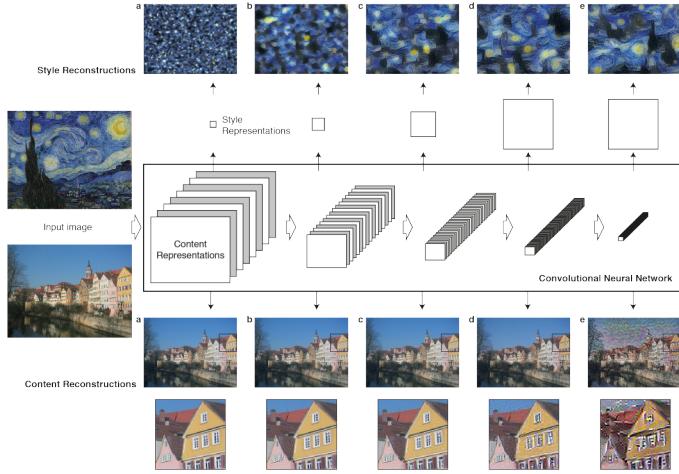


그림 94: **Convolutional Neural Network (CNN)**. A given input image is represented as a set of filtered images at each processing stage in the CNN. While the number of different filters increases along the processing hierarchy, the size of the filtered images is reduced by some downsampling mechanism (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.

λ_c^ℓ 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^ℓ , X 에서 뽑아낸 feature map을 G^ℓ 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^ℓ -weighted sum한 것이 L_s 가 된다.

$$L_s = \sum_{\ell \in SL} w_s^\ell \| \text{Gram}(A^\ell) - \text{Gram}(G^\ell) \|^2 \times \lambda_s^\ell. \quad (13)$$

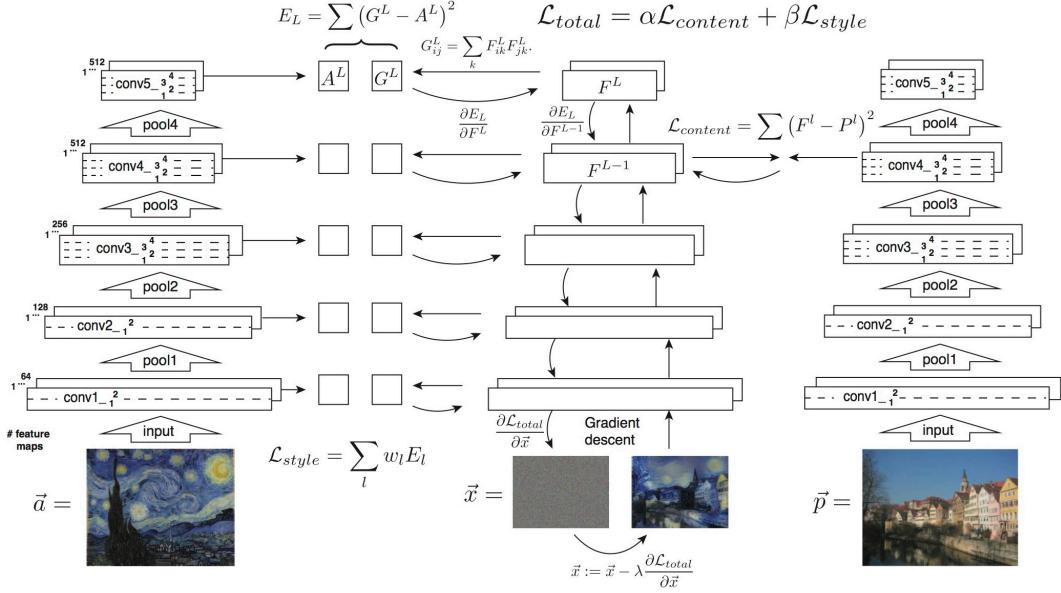


그림 95: Style Transfer Structure

λ_s^ℓ 는 layer(feature map) ℓ 의 가로, 세로 곱을 M_ℓ 필터 수를 N_ℓ 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,

$$L_t = \sum_{i,j} (x_{i+1,j} - x_{i,j})^2 + (x_{i,j+1} - x_{i,j})^2 \quad (14)$$

- L_c, L_s, L_t 에 weight α, β, γ 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.

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

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

9.12 Deep Photo Style Transfer

Fujun Luan, Sylvain Paris, Eli Shechtman, Kavita Bala, Deep Photo Style Transfer, 2017⁷⁹

- 논문 저자들은 Lua로 작성된 코드를 공개했다⁸⁰.
- Tensorflow로 구현된 코드⁸¹를 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(12)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(13)와는 다른 방식을 제시하고 있다. style 변환을 위해서 content image, style image 외에 content segmentation, style segmentation을 같이 입력받는다. 식(13)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.

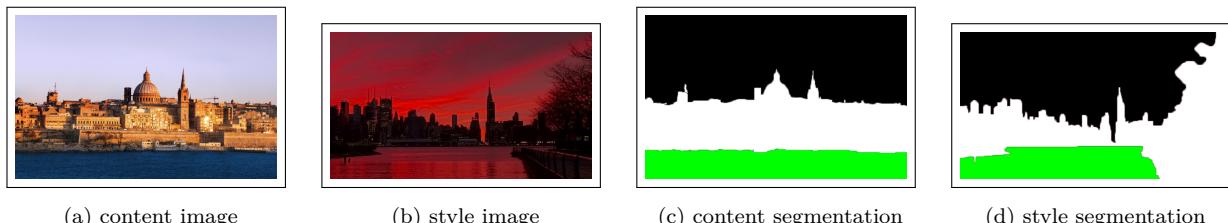


그림 96: Input Images

['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.

⁷⁹<https://arxiv.org/abs/1703.07511>

⁸⁰<https://github.com/luanfujun/deep-photo-styletransfer>

⁸¹<https://github.com/LouieYang/deep-photo-styletransfer-tf>



그림 97: 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, 3×3 , 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(14)와 동일하다.
- Loss minimization을 통해 생성된 image는 local affine smooth 방식으로 다시 처리되었다 (local affine smooth의 효과가 얼마나 되는지 확인 필요).

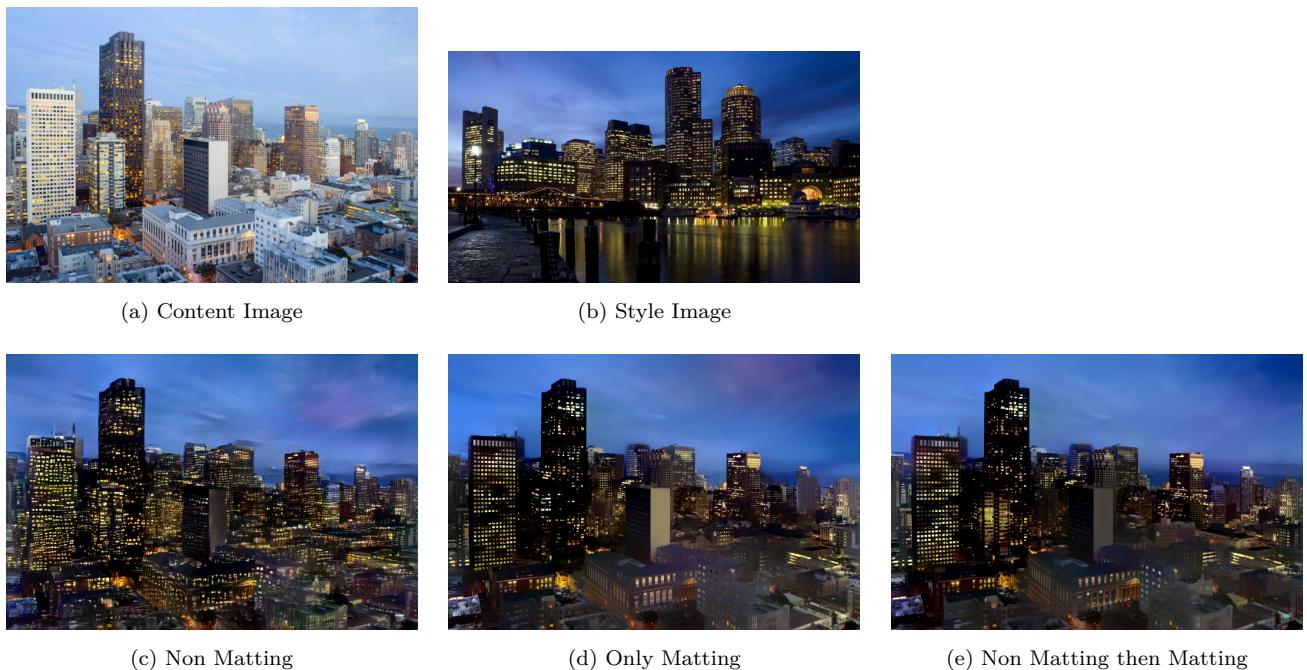


그림 98: Content, Style and Stylized Images

9.13 Pix2Pix GAN(2016)

♠ Conditional GAN

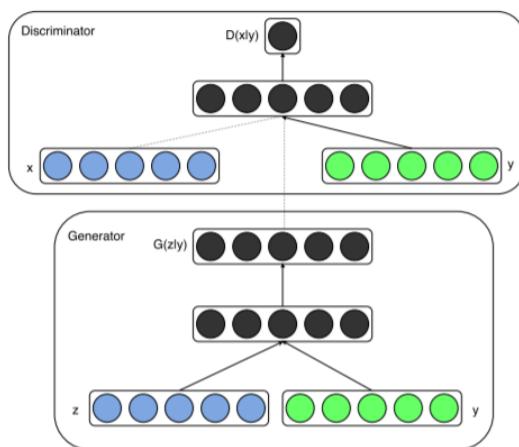


그림 99: Conditional GAN

- GAN의 Generator와 Discriminator에 어떤 추가적인 정보 y 를 넣어주는 것이 Conditional GAN. y 는 어떠한 정보여도 상관없음. Generator, Discriminator는 원래 입력과 추가되는 y 를 concatenation하면 된다.
 - 추가 정보 y 를 통해 output을 제어하는 것이 목적이다.

♠ Image to Image Translation

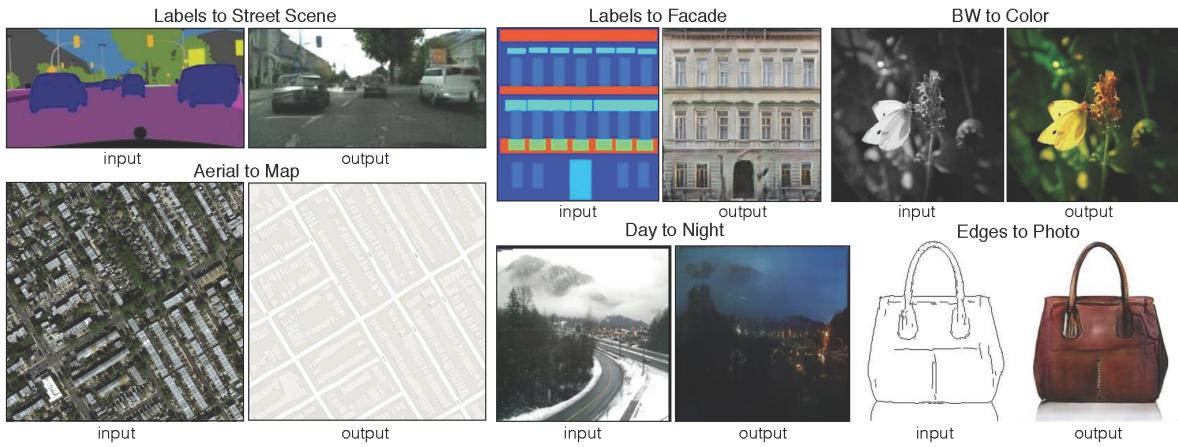


그림 100: Many problems in image processing, graphics, and vision involve translating an input image into a corresponding output image. These problems are often treated with application-specific algorithms, even though the setting is always the same: map pixels to pixels. Conditional adversarial nets are a general-purpose solution that appears to work well on a wide variety of these problems. Here we show results of the method on several. In each case we use the same architecture and objective, and simply train on different data.

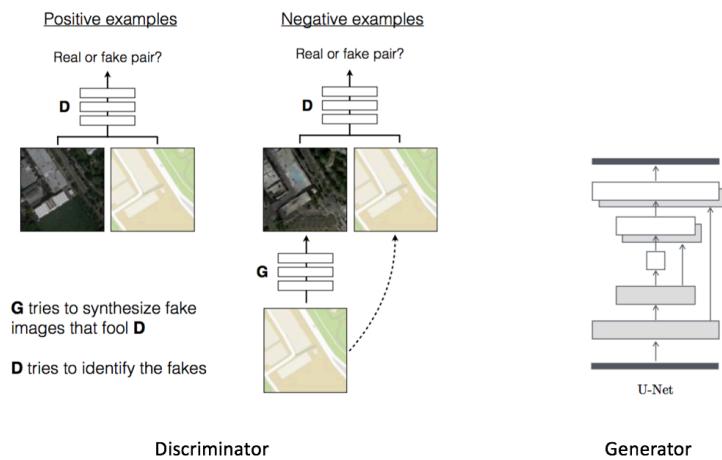


그림 101: Pix2Pix GAN structure

- Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, Alexei A. Efros. Image-to-Image Translation with Conditional Adversarial Networks, 2016⁸²
- 이 논문에서는 스케치로 부터 이미지를 만들어 내는 모델을 제안하고 있다. (최종적으로는) latent vector를 제외하고 deterministic하게 이미지를 만들어 내기 위해서 target 이미지에 대한 reconstruction loss를 추가한다.
- latent로 random vector만을 입력하는 보통의 GAN과 달리 입력 이미지 (x)를 추가적으로 Generator에 넣는다 (latent vector 없이 x 만 넣기도 한다). 이런 측면에선 보면 generative보다는 transfer라고 봐야 한다. 논문 제목에서도 translation이라고 되어 있다.
- Discriminator는 target(y , real image) 또는 Generator가 생성한 결과를 입력으로 받는다. Discriminator에 추가적인 정보 x 를 같이 넣어주는데, 이를 Conditional GAN(cGAN)이라고 한다.

⁸²<https://arxiv.org/abs/1611.07004>

- U-Net, PatchGAN: Generator는 U-Net 구조를 이용한다. 그리고 Discriminator는 PatchGAN으로 명명된 구조로 설계되었다. 일반적인 GAN의 Discriminator는 real인지 fake인지 판별하기 위해 0과 1사이의 실수값을 만들어낸다. 반면 PatchGAN은 $N \times N$ 개의 0과 1사이 실수를 만들어내어, 각각의 숫자가 real image라면 1에 가깝게, fake image라면 0에 가깝도록 loss function을 만든다. 논문에서는 70×70 PatchGAN을 제시하고 있다(여기서의 70은 receptive field 크기를 뜻하며, output 크기 N 이 아니다).
- Loss function: z 는 일반적인 GAN 모델에서의 latent vector.

$$\begin{aligned}\mathcal{L}_{cGAN}(G, D) &= \mathbb{E}_{x,y \sim p_{data}(x,y)}[\log D(x,y)] + \mathbb{E}_{x \sim p_{data}(x), z \sim p_z(z)}[\log(1 - D(x, G(x,z)))] , \\ \mathcal{L}_{L1}(G) &= \mathbb{E}_{x,y \sim p_{data}(x,y), z \sim p_z(z)}[\|y - G(x,z)\|_1].\end{aligned}$$

The final objective is

$$G^* = \arg \min_G \max_D \mathcal{L}_{cGAN}(G, D) + \lambda \mathcal{L}_{L1}(G)$$

reconstruction loss로 L_2 대신 L_1 -norm을 사용하는 것은 blurring을 방지하기 위해서다.

- 논문에서는 z 없는 loss function을 사용한다. Without z , the net could still learn a mapping from x to y , but would produce deterministic outputs, and therefore fail to match any distribution other than a delta function. Past conditional GANs have acknowledged this and provided Gaussian noise z as an input to the generator, in addition to x .
- Generator를 좀 더 자세히 살펴보자. Generator는 Encoder-Decoder 모델을 사용하는데, Encoder의 각 layer를 Decoder의 각 layer를 concatenation하는 U-Net 구조를 취하고 있다. encoder에서는 leaky relu, decoder에서는 relu를 사용하고 있다.

```
pix2pix generator:
input_data (N,256,256,3)
-> conv2d stride 2 (N,128,128,64) (L0)
-> lrelu -> conv2d stride 2, kernel 4 (N,64,64,128) -> batch_norm(L1)
-> lrelu-> conv2d stride 2, kernel 4 (N,32,32,256) -> batch_norm(L2)
-> lrelu-> conv2d stride 2, kernel 4 (N,16,16,512) -> batch_norm(L3)
-> lrelu-> conv2d stride 2, kernel 4 (N,8,8,512) -> batch_norm(L4)
-> lrelu-> conv2d stride 2, kernel 4 (N,4,4,512) -> batch_norm(L5)
-> lrelu-> conv2d stride 2, kernel 4 (N,2,2,512) -> batch_norm(L6)
-> lrelu-> conv2d stride 2, kernel 4 (N,1,1,512) -> batch_norm

-> relu -> deconv2d stride 2 (N,2,2,512) -> batch_norm -> dropout(0.5)
-> concat with L6(N,2,2,1024) -> relu -> deconv2d stride 2, kernel 4 (N,4,4,512) -> batch_norm -> dropout(0.5)
-> concat with L5(N,4,4,1024) -> relu -> deconv2d stride 2, kernel 4 (N,8,8,512) -> batch_norm -> dropout(0.5)
-> concat with L4(N,8,8,1024) -> relu -> deconv2d stride 2, kernel 4 (N,16,16,512) -> batch_norm
-> concat with L3(N,16,16,1024) -> relu -> deconv2d stride 2, kernel 4 (N,32,32,256) -> batch_norm
-> concat with L2(N,32,32,512) -> relu -> deconv2d stride 2, kernel 4 (N,64,64,128) -> batch_norm
-> concat with L1(N,64,64,256) -> relu -> deconv2d stride 2, kernel 4 (N,128,128,64) -> batch_norm
-> concat with L0(N,128,128,128) -> relu -> deconv2d stride 2, kernel 4 (N,256,256,3)
-> tanh (N,256,256,3)
```

```
pix2pix discriminator:  
concat: input_data|target_data (N,256,256,6)  
-> conv2d stride 2, kernel 4 (N,128,128,64) -> lrelu  
-> conv2d stride 2, kernel 4 (N,64,64,128) -> batch_norm -> lrelu  
-> conv2d stride 2, kernel 4 (N,32,32,256) -> batch_norm -> lrelu  
-> conv2d stride 1, kernel 4 (N,31,31,512) -> batch_norm -> lrelu  
-> conv2d stride 1, kernel 4 (N,30,30,1)  
-> sigmoid
```

9.14 CycleGAN(2017)

Jun-Yan Zhu, Taesung Park, Phillip Isola, Alexei A. Efros, Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks⁸³

♠ Data 준비 & 실행

- 이곳⁸⁴이 cylcegan project site. 이곳을 보면, 저자는 Torch, PyTorch로 구현된 코드를 올려놓았다. Tensorflow로 구현된 코드는 다른 사람(5명)이 구현한 코드를 링크해 놓았다.
- 여기서는 그 중 하나인 Van Huy⁸⁵가 구현한 tensorflow 코드를 중심으로 알아보자.
- 필요 data는 이곳⁸⁶에서 다운받으면 된다. 기본 image size는 256×256 .
- CycleGAN의 변수 개수는 약 29M.

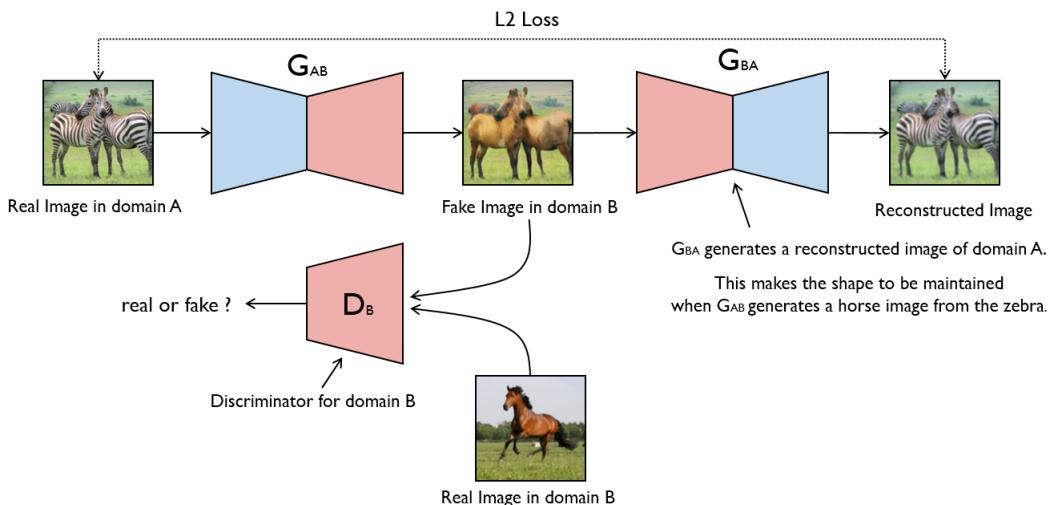


그림 102: CycleGAN. 논문에서는 cycle-consistency loss(reconstruction loss)로 L_1 norm을 제시하고 있다. 논문에서 G, F 로 표기되는 generator를 이 그림에서는 각각 G_{AB}, G_{BA} 로 나타내고 있다. G, F 는 같은 구조의 generator이지만, 각각 생성된 network이다.

♠ Generator G, F 구조

- CycleGAN에서는 batch size를 1.
- 보통의 BN 대신 instance normalization을 사용한다.
- 구현 코드는 Resnet을 사용한 것도 있고, Unet을 사용한 것도 있다. 여기서는 Resnet을 사용한다.
- generator 구조는 Justin Johnson model이다⁸⁷.

⁸³<https://arxiv.org/abs/1703.10593>

⁸⁴<https://junyanz.github.io/CycleGAN/>

⁸⁵<https://github.com/vanhuyz/CycleGAN-TensorFlow>

⁸⁶https://people.eecs.berkeley.edu/~taesung_park/CycleGAN/datasets/

⁸⁷<https://github.com/jcjohnson/fast-neural-style/>

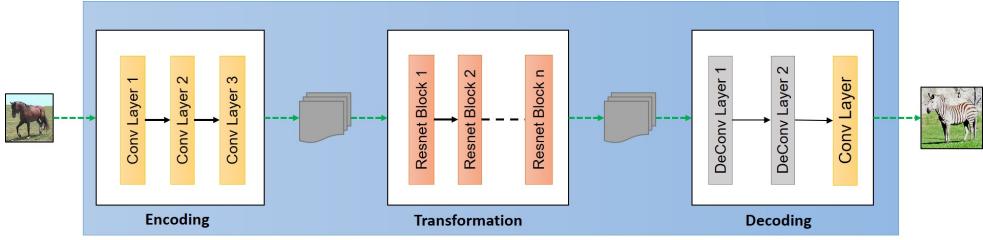


그림 103: Generator 구조는 StackGAN stageII의 Generator와 유사하다. encoding block(downsampling) - residual block - decoding block(upsampling)으로 구성되어 있다. input image size가 128이하이면 residual block을 6번 거치고, 128을 넘으면 9번 거치도록 설계되어 있다.

G, F

Downsampling block:

Real Image In domain X: $(N, 256, 256, 3)$

```
-> padding(3, 'REFLECT')  $(N, 262, 262, 3)$ 
-> conv2d(f=64, k=7, s=1, p='v') -> IN(Instance Normalization) -> relu  $(N, 256, 256, 64)$ 
-> conv2d(f=64 × 2, k=3, s=2, p='s') -> IN -> relu  $(N, 128, 128, 64 \times 2)$ 
-> conv2d(f=64 × 4, k=3, s=2, p='s') -> IN -> relu  $(N, 64, 64, 64 \times 4) := A_0$ 
```

Residual block:

A_0

```
-> padding(1, 'REFLECT')  $(N, 66, 66, 64 \times 4)$  -> conv2d(f=64 × 4, k=3, s=1, p='v') -> IN -> relu
 $(N, 64, 64, 64 \times 4)$ 
-> padding(1, 'REFLECT')  $(N, 66, 66, 64 \times 4)$  -> conv2d(f=64 × 4, k=3, s=1, p='v') -> IN  $(N, 64, 64, 64 \times 4)$ 
->  $A_{01}$ 
```

$A_0 + A_{01} \rightarrow A_1 (N, 64, 64, 64 \times 4)$

같은 방법 반복, Variable 공유는 아님.

$A_0 \rightarrow A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow A_4 \rightarrow A_5 \rightarrow A_6 \rightarrow A_7 \rightarrow A_8 \rightarrow A_9 (N, 64, 64, 64 \times 4) := A_n$

Upsampling block:

$A_n (N, 64, 64, 64 \times 4)$

```
-> deconv2d(f=64 × 2, k=3, s=2, p='s') -> IN -> relu(N, 128, 128, 64 × 2)
-> deconv2d(f=64, k=3, s=2, p='s') -> IN -> relu(N, 256, 256, 64)
-> padding(3, 'REFLECT')  $(N, 262, 262, 64)$  -> conv2d(f=3, k=7, s=1, p='v') -> tanh  $(N, 256, 256, 3)$ 
```

- Generator의 residual block에서 skip connection을 할 때, 2개의 node를 더한 후에 relu를 취하지 않았다. relu를 취하지 않는 것이 더 낫다는 주장이 있다.
- Generator의 마지막 convolution 후, normalization을 하지 않기 때문에 convolution 할 때 bias term이 있어야 되지 않나? 이 구현 코드에서는 없음. StackGAN 구현에서도 bias term이 없기는 했음.

♠ Discriminator D_X, D_Y 구조

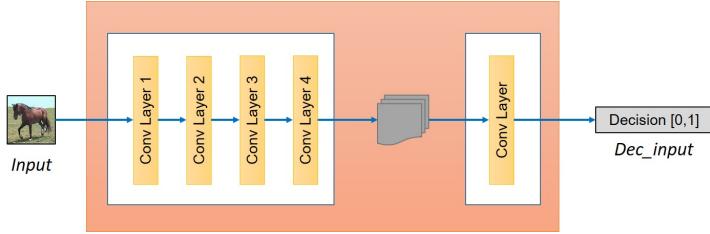


그림 104: Discriminator 구조.

D_X, D_Y

Discriminator block:

$X (N, 256, 256, 3)$
 $\rightarrow \text{conv2d}(f=64, k=4, s=2, p='s') \rightarrow \text{lrelu}(0.2) (N, 128, 128, 64)$
 $\rightarrow \text{conv2d}(f=64 \times 2, k=4, s=2, p='s') \rightarrow \text{IN} \rightarrow \text{lrelu}(0.2) (N, 64, 64, 64 \times 2)$
 $\rightarrow \text{conv2d}(f=64 \times 4, k=4, s=2, p='s') \rightarrow \text{IN} \rightarrow \text{lrelu}(0.2) (N, 32, 32, 64 \times 4)$
 $\rightarrow \text{conv2d}(f=64 \times 8, k=4, s=2, p='s') \rightarrow \text{IN} \rightarrow \text{lrelu}(0.2) (N, 16, 16, 64 \times 4)$

last convolution:

$\rightarrow \text{conv2d}(f=1, k=4, s=1, p='s', \text{bias=True}) \rightarrow (\text{sigmoid}) (N, 16, 16, 1)$

- last convolution의 sigmoid는 LSGAN에서의 least-square loss를 사용하는 경우에는 생략한다.
- discriminator는 70×70 PatchGAN을 사용하여, output의 크기가 16×16 이 된다. 70×70 PatchGAN에서 70의 의미는 receptive field의 크기로 discriminator의 output 크기(여기서는 16)를 말하는 것이 아니다. receptive field는 output 16×16 에서 각각의 숫자가 input image의 몇개 pixel에 영향을 받고 있는지를 나타내는 것이다. 여기서는 output중의 숫자 하나가 input image의 70개 pixel의 영향을 받는다는 의미이다.
- 논문에서는 안정적인 training을 위해서 2가지 technique을 제시한다.
- 첫번째는 least-square loss를 사용하면 더 stable하고 higher quality의 결과를 준다고 말하고 있다.
- 두번째는 discriminator에 넣어주는 fake image로 generator가 직전에 생성한 것을 사용하는 것이 아니라, 50개의 image pool을 만들고, 이 pool에서 뽑아 사용하는 방법이다. 자세히 살펴보면, 확률 $\frac{1}{2}$ 로 직전에 생성한 fake image를 사용하거나 또는 random하게 하나(batch size 만큼)를 뽑아 직전 생성한 fake image로 뽑은 이미지를 대체하고, pool에서 뽑은 image를 training에 사용한다.
- 이제 loss function에 대해서 살펴보자. 먼저 cycle consistency loss:

$$\text{forward cycle consistency: } x \rightarrow G(x) \rightarrow F(G(x)) \approx x,$$

$$\text{backward cycle consistency: } y \rightarrow F(y) \rightarrow G(F(y)) \approx y.$$

$$\text{Loss}_{cyc} := \lambda_1 \|F(G(x)) - x\|_1 + \lambda_2 \|G(F(y)) - y\|_1$$

$$\text{Loss}_{GG} := (D_Y(G(x)) - 1)^2 \text{ or } -\frac{1}{2} \log(D_Y(G(x)))$$

$$\text{Loss}_G := \text{Loss}_{GG} + \text{Loss}_{cyc}$$

$$\text{Loss}_{D_Y} := \frac{1}{2} \left[(D_Y(y) - 1)^2 + D_Y(G(x))^2 \right] \text{ or } \frac{1}{2} \left[\log D_Y(y) + \log (1 - D_Y(G(x))) \right]$$

- 대칭적으로 Loss_{FG} , Loss_F , Loss_{D_X} 도 정의된다.

$$\begin{aligned}\text{Loss}_{FG} &:= (D_X(F(y)) - 1)^2 \quad \text{or} \quad -\frac{1}{2} \log(D_X(F(y))) \\ \text{Loss}_F &:= \text{Loss}_{FG} + \text{Loss}_{cyc} \\ \text{Loss}_{D_X} &:= \frac{1}{2} \left[(D_X(x) - 1)^2 + D_X(F(y))^2 \right] \quad \text{or} \quad \frac{1}{2} \left[\log D_X(x) + \log(1 - D_X(F(y))) \right]\end{aligned}$$

10 Speech Recognition

10.1 From Speech Waveform To MFCC

- 음파를 다루는 python library로 librosa가 있다. wav파일을 가공하기 위해서는 librosa를 이용하거나 tensorflow 내에 있는 api를 이용하면 된다. data는 $[-1, 1]$ 의 값을 가진다.

```
audio_file = '00b01445_nohash_0.wav'  
y, sr = librosa.load(audio_file, sr=16000) 또는  
from tensorflow.python.ops import io_ops  
from tensorflow.contrib.framework.python.ops import audio_ops  
  
wav_loader = io_ops.read_file(audio_path)  
wav_decoder = audio_ops.decode_wav(wav_loader, desired_channels=1, desired_samples = sr)
```

- wav파일이나 mp3파일을 load하면 그림(105a)과 같은 1차원 data가 얻어진다. load할 때는 초당 몇개의 sampling을 할 것인지 sampling rate을 지정하여야 한다. 보통 16,000정도를 사용한다. 16,000은 사람의 가청 주파수를 고려하여 선택된 수이다.

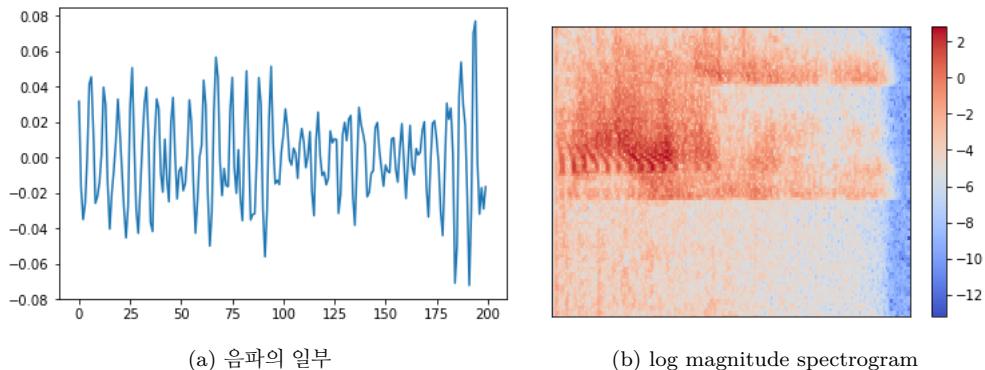


그림 105: 음파와 spectrogram

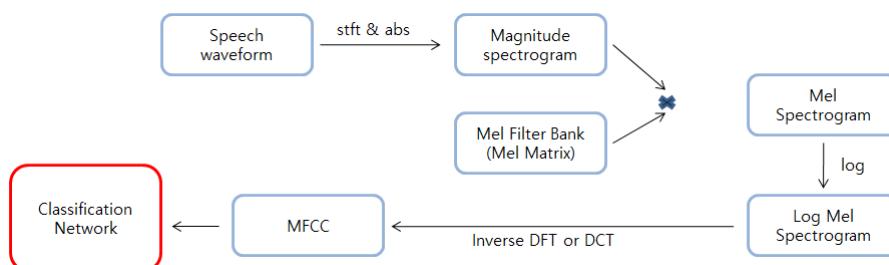


그림 106: 전체적인 모델 구조. wave data를 MFCC형태로 변환 후, Neural Network에 넣어서 분류작업을 수행한다. MFCC는 이미지와 동일한 2차원 array이므로 convolution layer를 가지는 분류 모델을 사용할 수 있다.

- 음파 data는 분석하기 위해서는 data를 window size(frame length) 크기로 stride(frame step) 만큼씩 이동하면서 자르고, 자른 data를 discrete Fourier Transformation 한다. 이렇게 하는 것을 stft(short time Fourier transform)이라고 한다. 이때 output의 크기를 결정할 fft-length를 정해주어야 한다(함수에 따라서는 fft-length를 지정하지 않고, window size보다 같거나 큰 2^n (fft-length)을 찾고 $2^{n-1} + 1$ 로 정하기도 한다. 같거나 큰 2^n 을 찾는 것은 bit_length()를 이용하면 된다.).

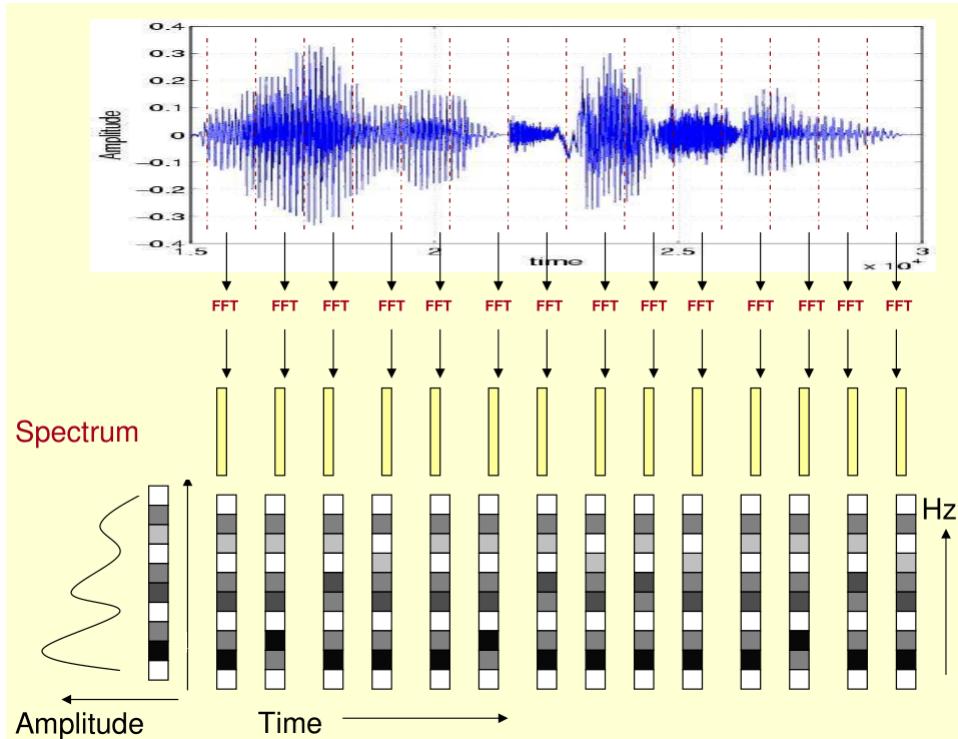


그림 107: 음파를 Discrete Fourier Transform하여 Spectrogram을 만드는 과정. 그림에서는 window size와 stride가 동일하여 겹치지 않게 Fourier Transform하는 것처럼 그려졌지만, window size보다 stride를 작게하여 Fourier Transform한다.

```
y: data, length=16,000
--> stft(window size = 480, stride = 160, fft length = 512)
--> (98, fft length / 2 + 1)
```

- stft를 하면, complex number로 된 2차원 array가 구해진다. 이렇게 구해진 frequency spectrum을 시간 축과 함께 시각적으로 표현한 것을 spectrogram이라 한다. complex number로 된 data를 spectrogram으로 나타내기 위해서 absolute를 취한다. absolute를 취한 것을 magnitude spectrogram, 제곱한 것은 power spectrogram이라 한다.
- stft를 계산하는 함수로 전달되는 input signal array는 시간 정보를 내부적으로 가지고 있다고 가정한다. sampling 간격이 시간이라고 보면된다. 예를 들어 다음 두 함수를 보자.

$$y_1 = \sin(1000 \times 2\pi x) + 0.5 \sin(3500 \times 2\pi x),$$

$$y_2 = \sin(2000 \times 2\pi x) + 0.5 \sin(7000 \times 2\pi x)$$

y_2 에서 y_1 보다 2배 조밀하게 sampling 한다면, sampling된 결과 array는 동일하게 된다. 이렇게 되면, sampling된 결과만으로 y_1, y_2 를 구별할 수가 없다. stft는 sampling된 결과 array를 입력으로 받기 때문에, 이 array 속에 시간 정보가 내재되어 있다고 보는 것이다⁸⁸.

- stft의 결과로 얻어지는 magnitude spectrum의 shape^o (N, M) 이라고 하면, N 은 input speech wave data

⁸⁸scipy.fftpack.fft는 2번째 argument로 넘어가는 n 이 시간축을 조절하는 역할을 한다.

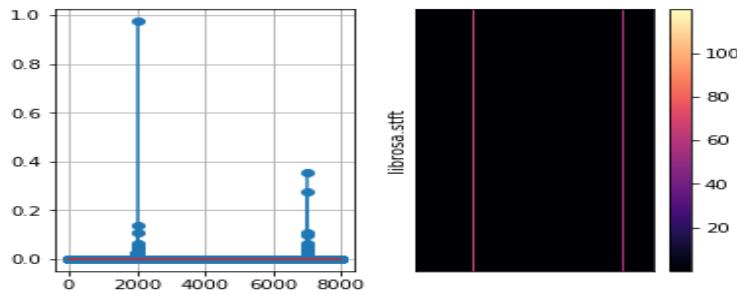


그림 108: $y = \sin(2000 \times 2\pi x) + 0.5 \sin(7000 \times 2\pi x)$ 에 대하여 DFT, librosa.stft를 적용한 결과. stft의 frequency 정보는 상대적인 frequency이지 절대적인 것이 아니다.

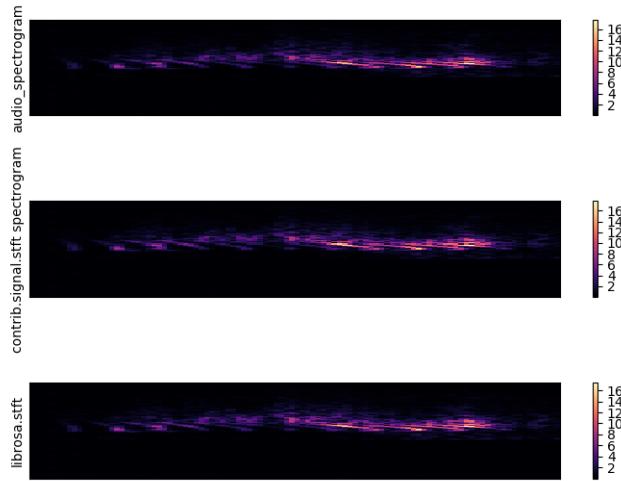


그림 109: stft를 구하는 3 가지 방식 (가로축이 frequency, 세로축이 time). 위에서부터 audio_ops.audio_spectrogram, tf.contrib.signal.stft, librosa.stft를 이용한 결과임. 위 2 가지는 수치적인 차이를 제외하면 완전히 일치하고, librosa를 이용한 것은 return 되는 array size도 약간 차이나고 결과값도 완전 일치하지는 않고 거의 유사하다.

의 길이, window size, stride에 의해 결정되고, M (fft unique bins)은 fft length에 의해

$$M = \frac{\text{fft length}}{2} + 1$$

로 계산된다. fft length를 입력받지 않고 stft를 계산하는 함수는 window size에 의해 내부적인 공식으로 결정한다.

- 주파수별 energy를 잘 나타내기 위해서는 log를 취해서 scale을 변경하기도 하고, mel scale이라는 방식으로 scale을 바꾸기도 한다.
- 인간의 귀는 낮은 주파수대에서 높은 resolution을 가지고, 높은 주파수대에서 낮은 resolution을 갖는다. 이러한 이유로 주파수를 log scale 변환이 필요하다. mel scale은 frequency f 를 다음과 같은 공식을 이용해서 m 으로 바꾼다. 그런데, mel scale formula는 여러 가지가 있다.

$$m = 2595 \log_{10} \left(1 + \frac{f}{700} \right)$$

- 이제 mel scale로 변화하는 구체적인 방법을 살펴보자. magnitude spectrogram($N \times M$)을 mel filter bank(mel matrix, $M \times K$)와 곱하여 mel spectrogram($N \times K$)을 만든다. 여기서 K 는 filter bank를 생성할 때 지정하는 mfcc의 갯수이다.

```

sr = 16000
n_mfcc = 40
librosa_mel_matrix =
    librosa.filters.mel(sr=sr,n_fft=512,n_mels=n_mfcc,fmin=80.0,fmax=7600.0,norm=1) #
    (n_mfcc,n_fft//2 +1)

num_spectrogram_bins = 257
tf_mel_matrix = tf.contrib.signal.linear_to_mel_weight_matrix(num_mel_bins=n_mfcc,
    num_spectrogram_bins=num_spectrogram_bins, sample_rate=sr, lower_edge_hertz=80.0,
    upper_edge_hertz=7600.0) # (257, 40)

```

- 이제, log 를 취하면 log mel spectrogram이 된다.
- Mel-frequency cepstral coefficients(MFCCs): log mel spectrogram에 Discrete Cosine Transform(DCT)이나 Inverse DFT를 취하여 MFCC를 구한다. DCT는 실수값으로 계산되기 때문에 더 편리하다. mfcc의 계산은 `tf.contrib.signal.mfccs_from_log_mel_spectrograms(log_mel_spectrogram)` 또는 `audio_ops.mfcc(squared_magnitude_spectrogram)`으로 하면된다. 최종적으로 MFCC array는 $N \times K$ 가 된다.

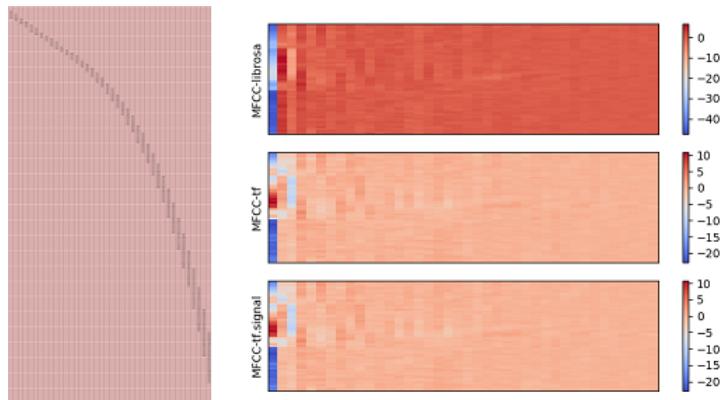


그림 110: Mel Filter Bank: sparse matrix인데, tensorflow의 `tf.contrib.signal.linear_to_mel_weight_matrix`와 `librosa.filters.mel`은 scale이 다른 결과를 준다. Filter Bank가 다르기 때문에, Spectrogram도 달라지고 계속해서 MFCC까지 다른 결과를 낸다.

- (참고) 지금까지 stft을 이용하여 spectrogram을 구하고 mfcc까지 계산하는 방식을 살펴보았다. mfcc로 feature를 구하지 않고, 단순 average로 feature를 구하는 방법도 있다. 먼저 구해진 squared spectrogram의 shape이 $(N, 98, 257)$ 이라고 하자. 앞에서 예로 mfcc를 40로 설정했는데, 비슷한 정도의 averaging data를 구하려면 $\text{int}(257/40) = 6$ 개씩 average하면 된다. 실제 6개씩 average pooling하면 output의 shape은 $(98, 43)$ 이 된다.

```

# to use tf.nn.pool, we have to make spectrogram 4 dimensional. (N, 98, 257, 1)
output = tf.nn.pool(tf.expand_dims(spectrogram, -1), window_shape=[1,6], strides=[1,6],
    pooling_type='AVG', padding='SAME')

```

10.2 Model Training

- tensorflow의 Simple Audio Recognition⁸⁹을 바탕으로 살펴보자. 먼저 data를 준비하는 단계부터 보자.
- 지금까지 audio 파일로 부터 mfcc 형태로 data를 변환하는 과정을 살펴보았다. 실제 training에서는 audio data에 volume 조절, time shift, background sound 추가를 통해, audio data를 변형한 후 mfcc로 변환한다. 이런 변형 과정에서 time shift 정도, background sound(noise) 추가 등이 random하게 이루어지기 때문에 data를 미리 만들어 놓을 수는 없다.
- 이 모델은 35개의 단어에 대한 1초짜리 wav 파일을 분류하는 모델이다. 각 단어는 평균 3000개 정도의 파일을 가지고 있다. 모델은 35개의 단어 중 10개 정도를 골라 분류하는 방식인데, 나머지 단어들은 ‘unknown’으로 분류한다. ‘unknown’으로 분류된 것들은 그 중 일부만 (e.g. 10%) training에 사용된다.
- ‘_silence_’: 모델이 관심을 두지 않을 수 있는 항목을 추가한다. silence label data는 완전히 소리를 없애는 것은 아니고, background noise만 더해서 생성한다.
- 지정한 단어 10개에 unknown, silence를 더해 모두 12개 항목을 분류대상으로 한다.

	training(80%)	validation(10%)	test(10%)
yes(2)			
no(3)			
up(4)	30,769개	3,703개	4,074개
:			
go(11)			
'_silence_'(0)	3,077개	371개	408개
'unknown'(1)	3,077개	371개	408개

표 7: training에 사용하는 audion data 구분

- audio data 가공
 - audio 파일로 부터 sampling rate/sec(=16000)를 지정하여 data를 읽어온다. 우리가 사용하는 audio file은 1초짜리이므로 data의 shape은 (16000,)이 된다.
 - `_silence_` flag로 지정된 data는 0을 곱해서 volume을 0으로 만든다. 다음 단계에서 background noise만 더해진다.
 - time shift: (-100, 100) ms = (-1600, 1600) sampling point 사이에서 random하게 time shift amount를 정해서 이동시킨다. 이동 후 빈공간은 0으로 padding한다.
 - background noise 추가: background_frequency(0.8) 확률로 background noise를 추가할지 결정한 후, 0~background_volume(0.1) 사이의 값으로 volume을 조절하여 background noise를 생성한다. background noise는 data 길이가 긴데, 16,000 sampling point(=1초) 만큼만 random하게 선택한다. 생성된 background noise를 time shift 처리된 audio data에 더한 후, [-1, 1] 사이의 값으로 clip 한다. 여기까지 처리하면, mfcc 형태로 변환하기 전 단계의 data가 된다.
- Model: audio 파일에 대한 분류 모델. audio data를 2 dimension data (98, 40)으로 변형했기 때문에 이미지와 동일하게 취급할 수 있다. 여기서는 5개의 모델을 제시하고 있다.

⁸⁹https://www.tensorflow.org/tutorials/sequences/audio_recognition

- Single FC: 1 layer FC 모델로 sanity check 목적.
- Standard Convolution: 가장 정확도가 높은 모델로 conv2d-relu-dropout-maxpooling-conv2d-relu-dropout-fc(12)-softmax로 되어 있다⁹⁰.

Standard Convolution:

```
mfcc data: (N,98,40) -> reshape (N,98,40,1)
-> conv2d(f=64,k=(20,8),s=1,p='s') (N,98,40,64)
-> relu -> dropout(0.5) -> max pooling(k=[1, 2, 2, 1], s=[1, 2, 2, 1]) (N,49,20,64)
-> conv2d(f=64,k=(10,4),s=1,p='s') (N,49,20,64)
-> relu -> dropout(0.5)
-> flatten (N,62720)
-> FC(12) -> softmax
```

- Tiny Convolution: micro controllers 같은 소형 device에 사용할 수 있는 모델.

Tiny Convolution:

```
mfcc data: (N,98,40) -> reshape (N,98,40,1)
-> conv2d(f=8,k=(10,8),s=2,p='s') (N,49,20,8)
-> relu -> dropout(0.5)
-> flatten (N,7840)
-> FC(12) -> softmax
```

- Low Latency Convolution⁹¹: multiplication 횟수와 parameter 갯수를 줄이는 CNN architecture를 찾는 것이 이 방법의 핵심이다. convolution layer를 1개만 사용한 모델이다. time에 대한 kernel size를 크게 하는 것이 특징이다.

Tiny Convolution:

```
mfcc data: (N,98,40) -> reshape (N,98,40,1)
-> conv2d(f=186,k=(98,8),s=1,p='v') (N,1,33,186)
-> relu -> dropout(0.5)
-> flatten (N,6138)
-> FC(128) -> dropout (-> relu) * 구현코드에 relu 가빠져있다 .
-> FC(128) -> dropout (-> relu)
-> FC(12) -> softmax
```

- Low Latency SVDF⁹²: We propose a scheme to compress an existing fully-trained DNN using a low-rank approximation of the weights associated with individual nodes in the first hidden layer by means of a rank-constrained DNN layer topology. This allows us to significantly reduce the number of independent parameters in the model, without any loss in performance. SVD는 singular value decomposition 인 것 같은데, F가 뭔지는 논문에 나오지도 않는다.

⁹⁰코드의 comment에는 두번쨰 max pooling이 적혀 있는데, 구현에서는 빠져 있음.

⁹¹http://www.isca-speech.org/archive/interspeech_2015/papers/i15_1478.pdf

⁹²<https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/43813.pdf>

11 Miscellaneous

11.1 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 α .
- 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 β and a learning rate α .
- Common values for β range from 0.8 to 0.999. If you don't feel inclined to tune this, $\beta = 0.9$ is often a reasonable default.

$$v \leftarrow \beta v + (1 - \beta)dW$$

$$W \leftarrow W - \alpha v$$

♠ Adam

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

11.2 Batch Normalization

- training 속도 향상.
- weight 초기값에 크게 영향 받지 않는다.
- overfitting 억제. dropout 등의 필요성 감소.

- Batch Normalization Layer는 주로 Activation Function 앞에 놓여진다⁹³. Batch Normalization의 backward propagation은 Computational Graph를 통해서 계산 할 수 있다.
- Spatial Batch Normalization: image($N \times H \times W \times C$)에 대하여 Batch Normalization을 적용할 때는, $N \times H \times W$ 에 대하여 평균, 분산을 취하기 때문에, C 개의 평균, 분산이 계산된다.

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

- training mode에서는 batch data 자체로 평균, 분산을 계산하여 normalization^{o]} 이루어진다(normalized with statistics of the current batch). 그리고 moving-mean, moving-variance는 update만 시킨다. momentum α (eg. 0.99)에 대하여

$$\begin{aligned} \text{moving-mean} &\leftarrow \alpha(\text{moving-mean}) + (1 - \alpha)(\text{batch-mean}) \\ \text{moving-variance} &\leftarrow \alpha(\text{moving-variance}) + (1 - \alpha)(\text{batch-variance}) \end{aligned}$$

- inference mode에서는 training mode에서 저장해 놓은 moving-mean, moving-variance로 normalization된다(normalized with moving statistics).

♠ Back Propagation Speed Up 구현⁹⁴

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.

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

⁹³Activation Function^{o]} Batch Normalization보다 앞에 와야 한다는 주장도 있다. <https://github.com/keras-team/keras/issues/5465> 아래쪽 comment.

⁹⁴<https://costapt.github.io/2016/07/09/batch-norm-alt/>

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

- 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 = x_n
    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

```

♠ Instance Normalization

```

x = tf.placeholder(tf.float32, [None,16,16,3])
param_initializers = {'beta':tf.random_normal_initializer(mean=0.0, stddev=0.02,
    dtype=tf.float32), 'gamma': tf.ones_initializer()}
#param_initializers = {'beta':tf.random_normal_initializer(mean=0.0, stddev=0.02, dtype=tf.float32)}
y = tf.contrib.layers.instance_norm(x,param_initializers=param_initializers)
sess = tf.Session()

sess.run(tf.global_variables_initializer())
graph = tf.get_default_graph()
beta = graph.get_tensor_by_name('InstanceNorm/beta:0')
gamma = graph.get_tensor_by_name('InstanceNorm/gamma:0')
print(sess.run([beta,gamma]))

```

11.3 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.

11.4 Principal Components Analysis

Let the random vector $\mathbf{X} = [X_1, X_2, \dots, X_p]^t$ ($X_i \in \mathbb{R}$, random variable) which has the covariance matrix $\Sigma = (\sigma_{ij})$ with eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_p \geq 0$. For $a_{ij} \in \mathbb{R}$, consider the linear combinations

$$\begin{aligned}
Y_1 &= \mathbf{a}_1^t \mathbf{X} = a_{11}X_1 + a_{12}X_2 + \cdots + a_{1p}X_p \\
Y_2 &= \mathbf{a}_2^t \mathbf{X} = a_{21}X_1 + a_{22}X_2 + \cdots + a_{2p}X_p \\
&\vdots \\
Y_p &= \mathbf{a}_p^t \mathbf{X} = a_{p1}X_1 + a_{p2}X_2 + \cdots + a_{pp}X_p.
\end{aligned}$$

Let

$$A = (a_{ij}) = \begin{bmatrix} \mathbf{a}_1^t \\ \mathbf{a}_2^t \\ \vdots \\ \mathbf{a}_p^t \end{bmatrix}.$$

Then $\mathbf{Y} = A\mathbf{X}$.

1. First principal component = linear combination $a_1^t \mathbf{X}$ that maximizes $\text{Var}(a_1^t \mathbf{X})$ subject to $\|\mathbf{a}_1\| = 1$
2. Second principal component = linear combination $a_2^t \mathbf{X}$ that maximizes $\text{Var}(a_2^t \mathbf{X})$ subject to $\|\mathbf{a}_2\| = 1$ and $\text{Cov}(a_1^t \mathbf{X}, a_2^t \mathbf{X}) = 0$.
3. i th principal component = linear combination $a_i^t \mathbf{X}$ that maximizes $\text{Var}(a_i^t \mathbf{X})$ subject to $\|\mathbf{a}_i\| = 1$ and $\text{Cov}(a_k^t \mathbf{X}, a_i^t \mathbf{X}) = 0$ for $k < i$.

Put

$$A = \begin{bmatrix} \text{the first eigenvector} \\ \text{the second eigenvector} \\ \vdots \\ \text{the } p\text{th eigenvector} \end{bmatrix} = \begin{bmatrix} \mathbf{a}_1^t \\ \mathbf{a}_2^t \\ \vdots \\ \mathbf{a}_p^t \end{bmatrix},$$

and $\mathbf{Y} = A\mathbf{X}$. Then

$$\begin{aligned} \Sigma &= A^t D A, \quad \text{where } D = \text{diag}(\lambda_1, \dots, \lambda_p), \\ &= [\mathbf{a}_1 \ \mathbf{a}_2 \ \dots \ \mathbf{a}_p] D A \\ &= [\lambda_1 \mathbf{a}_1 \ \lambda_2 \mathbf{a}_2 \ \dots \ \lambda_p \mathbf{a}_p] \begin{bmatrix} \mathbf{a}_1^t \\ \mathbf{a}_2^t \\ \vdots \\ \mathbf{a}_p^t \end{bmatrix}, \\ &= \lambda_1 \mathbf{a}_1 \mathbf{a}_1^t + \lambda_2 \mathbf{a}_2 \mathbf{a}_2^t + \dots + \lambda_p \mathbf{a}_p \mathbf{a}_p^t, \\ \text{Var}(\mathbf{Y}) &= A \Sigma A^t = D. \end{aligned}$$

Hence we find that

$$\begin{aligned} \sigma_{ij} &= \left(\sum_{k=1}^p \lambda_k \mathbf{a}_k \mathbf{a}_k^t \right)_{ij} \\ &= \sum_{k=1}^p \lambda_k a_{ki} a_{kj}. \end{aligned}$$

For example, the sum of squares of the first component of each eigenvector is σ_{11} up to eigenvalues. Note that⁹⁵

$$\begin{aligned}
 \sum_{i=1}^p \text{Var}X_i &= \sigma_{11} + \sigma_{22} + \cdots + \sigma_{pp} \\
 &= \text{tr}(\Sigma) \\
 &= \text{tr}(A\Sigma A^t) \\
 &= \text{tr}(D) \\
 &= \lambda_1 + \lambda_2 + \cdots + \lambda_p \\
 &= \sum_{i=1}^p \text{Var}Y_i \\
 &= \text{total population variance.}
 \end{aligned}$$

11.5 Support Vector Machine

SVM을 간략히 정리해 보자⁹⁶.

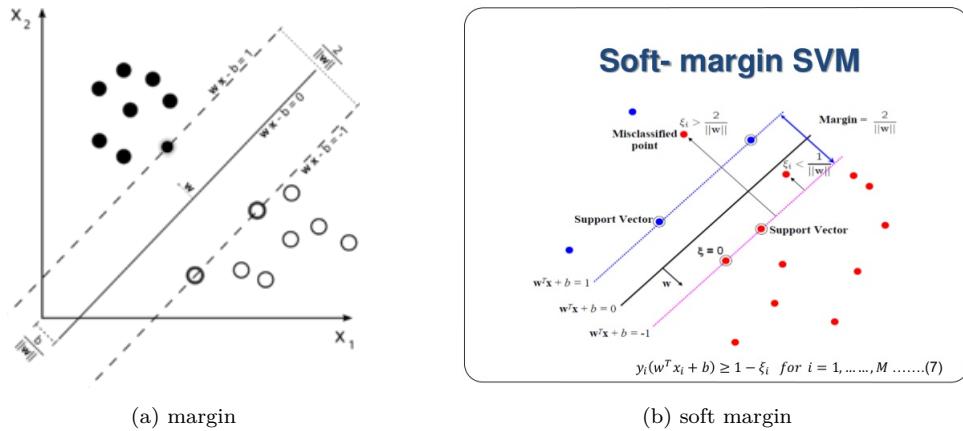


그림 111: SVM Margin

- 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

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

- 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

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

⁹⁵tr(AB) = tr(BA)

⁹⁶https://en.wikipedia.org/wiki/Support_vector_machine

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

This can be rewritten as:

$$y_i(\vec{w} \cdot \vec{x}_i - b) \geq 1, \quad \text{for all } 1 \leq i \leq n. \quad (15)$$

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 (15) 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 λ 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 λ , 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 (15) 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

$$\left[\frac{1}{n} \sum_{i=1}^n \max(0, 1 - y_i(\vec{w} \cdot \vec{x}_i - b)) \right] + \lambda \|\vec{w}\|^2, \quad (16)$$

where the parameter λ 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 λ , 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 ((16)) 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 ζ_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

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

○ 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, $\sum_{i=1}^n c_i y_i = 0$ ○] 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:

11.6 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,

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

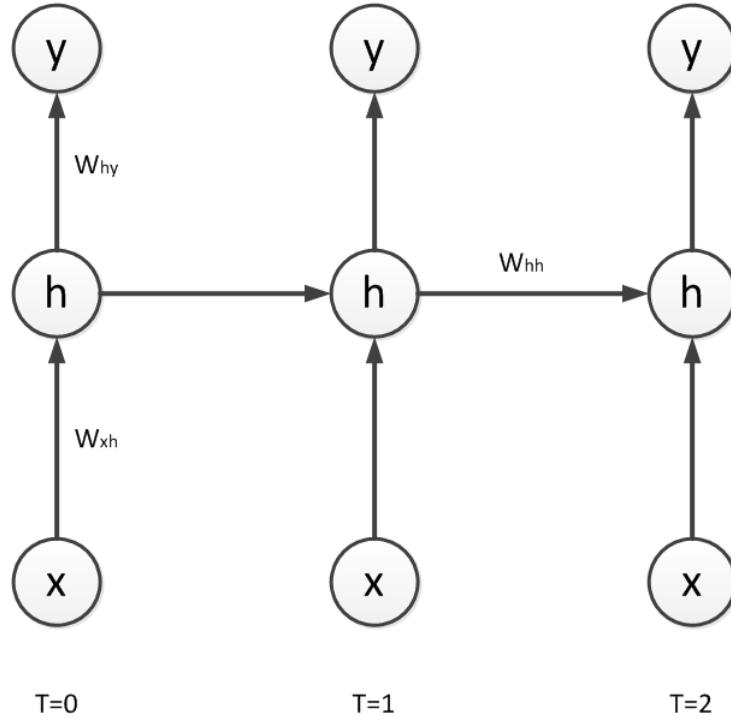
- 4가지 경우를 모두 합치면 다음과 같다.

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

row-wise product and then sum.

11.7 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 112: Example

Example 11.1. (그림 112)과 같이 주어진 RNN에서 $\frac{\partial E}{\partial z_1}$ 을 계산해 보자.

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

11.8 Deep Learning Note

2018-04-11 tensorflow bidirectional rnn example을 들려보던 중, output에 FC를 추가하는 부분이 weight W 를 곱하고, bias를 더하는 방식으로 되어 있었는데, 이부분을 변수를 직접 선언하지 않는 tf.layers.dense로 바꾸었는데, training이 되지 않았음. 기존 코드는 W 를 $N(0, 1)$ 초기화 하였고, 나는 xavier initialization을 했을 뿐인데... 원인은 learning rate. learning rate을 0.001에서 0.05 수정하면, 내가 수정한 코드는 물론 원래 코드로 accuracy 가 99% ~ 100%

2018-06-07 tensorflow에서 iteration이 수행되는 부분에 op를 숫자로 나누거나 하면, 계속 새로운 op가 생성되어 속도가 느려진다.

```
_ , g_loss = self.sess.run([g_loss_optimizer, self.g_loss/100] , feed_dict={self.z: z})
```

2018-07-03 stackgan stageII를 실행 중 python kernel이 이유없이 죽는 경우가 발생. spyder환경에서 실행했더니, scipy 관련 dll을 로드하지 못했다는 에러 메시지 출력. Eclipse나 cmd환경에서는 이런 메시지를 주지 않았음. scipy upgrade 후 이 문제는 해결됨.

2018-07-31 직접 구현한 cyclegan 결과가 reference의 결과보다 못함. 먼저 구현이 같은지를 비교하기 위해, trainable variable 개수가 일치하는지 정밀하게 확인. 확인 결과 instance norm이 적용되지 않고 있었음. self.args.norm이라고 해야되는 것을 self.norm으로 하고 있었음. 이렇게 수정 후에도 trainable variable 2개 차이가 남. 이 부분은 discriminator에 instance norm을 적용하지 않아야 되는데 적용하고 있었음. instance norm 적용 회수를 비교하여 찾아냄.

이 파일의 내용, 식, 그림들은 인터넷을 통해 얻은 자료들이다. 독창적인 자료가 아닙니다. 출처를 언급하지 못한 것들이 많이 있습니다.