
ICLR 2016

**Deep Compression:
Compressing Deep Neural Networks with Pruning,
Trained Quantization and Huffman Coding**

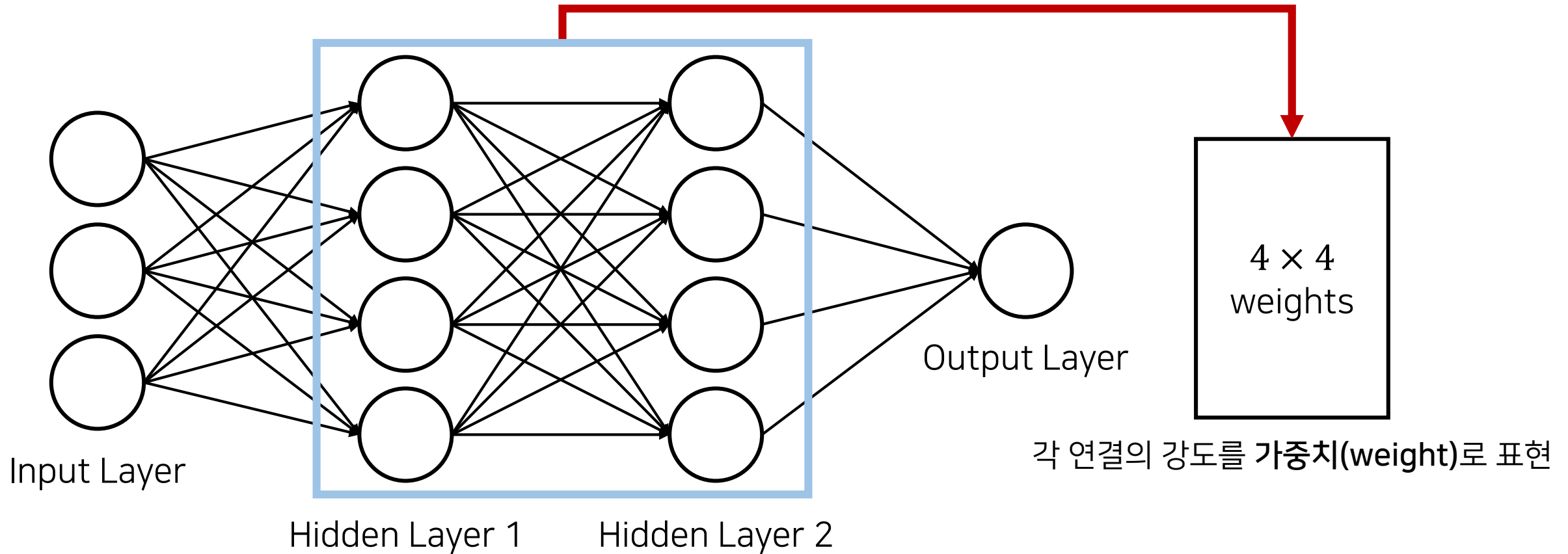
Song Han, Huizi Mao, William J. Dally

Stanford University, Tsinghua University

[배경지식] 딥러닝(Deep Learning) 모델

- 딥러닝 모델은 여러 레이어로 구성되며 다수의 뉴런이 서로 **연결(connection)**됩니다.

4개의 입력 뉴런과 4개의 출력 뉴런으로 구성되므로 16개의 연결(connection) 존재



[배경지식] 모바일(Mobile) 환경에서의 딥러닝(Deep Learning) 기술

- 모바일 환경에서의 딥러닝 기술은 개인정보(privacy) 보호, 실시간 처리(real-time processing), 네트워크 대역폭(network bandwidth) 측면에서의 이점이 있습니다.
- 하지만 딥러닝 모델의 **큰 용량** 때문에 모바일 앱에 딥러닝 모델을 직접적으로 탑재하기 어렵습니다.



Phones



Drones



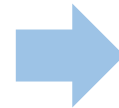
Self Driving Cars

[배경지식] 딥러닝 모델의 에너지 소비(Energy Consumption) 문제

- 큰 네트워크를 사용할 때 가중치를 인출(fetch)하는 과정에서 많은 메모리 대역폭을 요구하고, 포워딩을 위해 많은 수의 내적(dot product) 연산을 수행합니다.
- 특히 메모리 접근(memory access) 과정에서 많은 에너지 소비가 발생할 수 있습니다.

< 뉴럴 네트워크 예시 >

- 초당 프레임(FPS): 20
- 메모리 접근 에너지 소비량: 640pJ
- 뉴럴 네트워크 연결(connection) 수: 1 billion



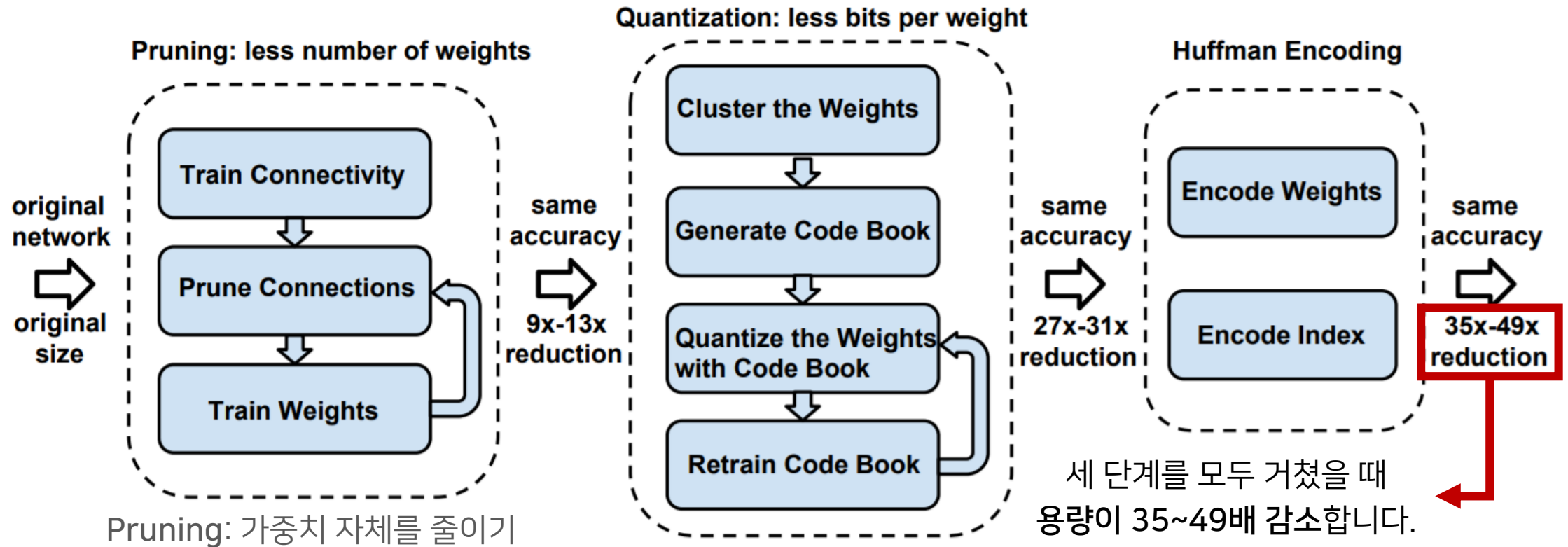
$$\begin{aligned}\text{초당 전력량(Watt)} \\ &= (20\text{Hz})(1\text{G})(640\text{pJ}) \\ &= 12.8\text{W}\end{aligned}$$



본 논문에서는 뉴럴 네트워크의 용량과 에너지 소비량을 줄일 수 있는 **Deep Compression**을 제안합니다.

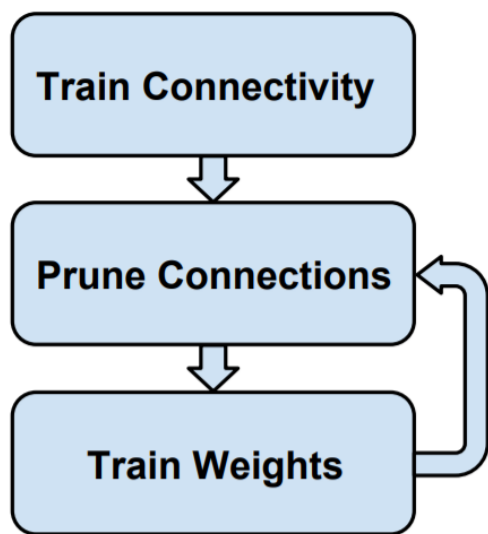
Deep Compression: 3단계 압축 파이프라인(Three Stage Compression Pipeline)

1. 가지치기(Pruning): 불필요한 연결을 가지치기하여 중요한(important) 연결만을 남깁니다.
2. 양자화(Quantization): 각 가중치를 나타내기 위한 비트(bit)의 수를 감소시킵니다.
3. 허프만 코딩(Huffman coding) : 자주 등장하는 가중치(weight)에 작은 코드워드(codeword) 할당합니다.

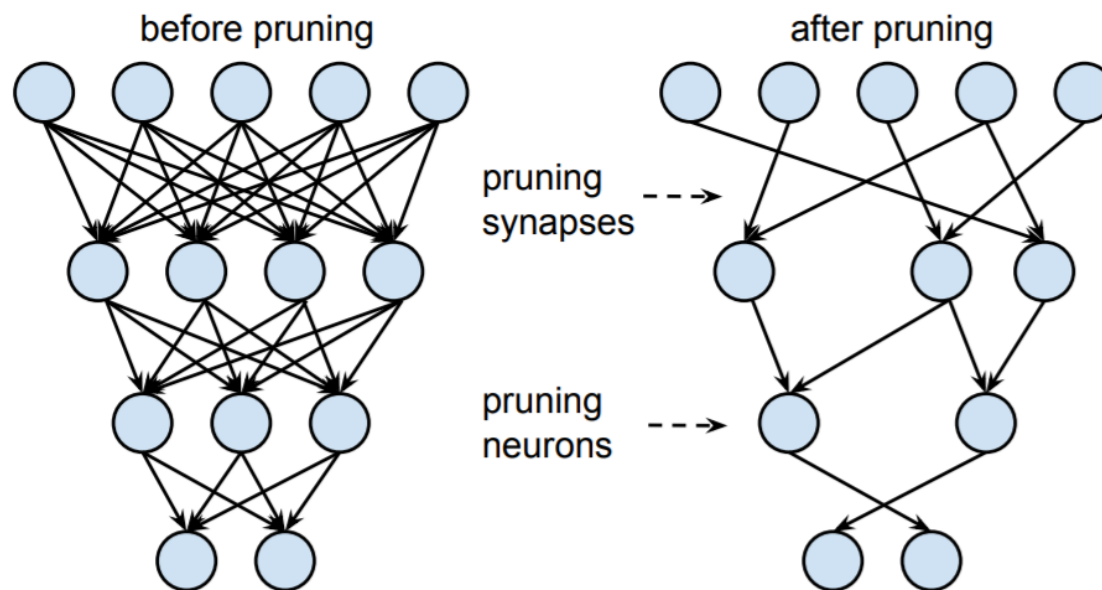


1단계: 네트워크 가지치기(Network Pruning)

- 네트워크 가지치기(pruning)는 복잡도(complexity)와 과적합(over-fitting) 감소에 효과가 있습니다.
 - 일반적인 네트워크 학습을 진행합니다.
 - 가중치(weight) 값이 작은 연결(connection)을 제거합니다.
 - 남아있는 연결(connection)을 유지한 상태로 가중치를 **재학습(retraining)**합니다.



[Figure] Three-Step Training Pipeline.



[Figure] Synapses and neurons before and after pruning.

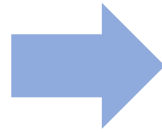
[배경지식] 밀집행렬(Dense Matrix)과 희소행렬(Sparse Matrix)

- 가지치기를 수행하여 희소행렬을 생성할 수 있습니다.
 - 6 X 6 가중치 행렬이 있을 때 가중치 값이 3 이상인 연결(connection)만 남기면 어떻게 될까요?

원본 행렬(Matrix)

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

가지치기
(Pruning)



가지치기된 행렬(Matrix)

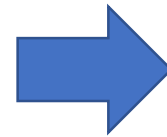
0	0	0	0	6	0
0	3	0	4	0	0
0	0	0	0	0	0
5	0	0	0	0	0
0	0	0	0	0	0
0	0	3	7	0	0

[배경지식] 희소 구조(Sparse Structure): Coordinate Format (COO)

- 행렬에 포함된 0이 아닌 값을 가진 데이터에 대하여 행(row)과 열(column) 위치 정보를 기록합니다.

0	0	0	0	6	0
0	3	0	4	0	0
0	0	0	0	0	0
5	0	0	0	0	0
0	0	0	0	0	0
0	0	3	7	0	0

COO 형식



Data	Row	Column
6	0	4
3	1	1
4	1	3
5	3	0
3	5	2
7	5	3

- 0이 아닌 원소의 개수가 a 일 때 $3a$ 만큼의 원소가 요구

[배경지식] 희소 구조(Sparse Structure): Compressed Sparse Row (CSR)

- 행 압축 정보(Row Pointer)를 이용해 희소행렬을 표현하는 자료구조입니다.



[배경지식] 희소 구조(Sparse Structure): COO와 CSR 형식 파이썬 구현

- 임의의 희소 행렬을 COO와 CSR 형식으로 저장하고 사용할 수 있습니다.

```
import numpy as np
from scipy import sparse

# 원본 행렬(Matrix)
arr = [
    [0, 0, 0, 0, 0, 1],
    [0, 3, 0, 4, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [5, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 3, 7, 0, 0]
]
dense_mat = np.array(arr)
print(dense_mat)
```

```
data = np.array([1, 3, 4, 5, 3, 7])
col_idx = np.array([5, 1, 3, 0, 2, 3])
row_idx = np.array([0, 1, 1, 3, 5, 5])

# COO (Coordinate) 형식
coo_mat = sparse.coo_matrix((data, (row_idx, col_idx)))
print(coo_mat.toarray())

# CSR (Compressed Sparse Row) 형식
row_ptr = np.array([0, 1, 3, 3, 4, 4, 6])
csr_mat = sparse.csr_matrix((data, col_idx, row_ptr))
print(csr_mat.toarray())
```

1단계: 네트워크 가지치기(Network Pruning)

- Deep Compression에서는 희소 구조(sparse structure)를 저장하기 위한 방법을 선택할 수 있습니다.
 - Compressed Sparse Row (CSR)
 - Compressed Sparse Column (CSC)
- 값이 0이 아닌 원소의 개수가 a 이고 n 개의 행과 열이 있을 때, $2a + n + 1$ 개의 수를 저장해야 합니다.
- 추가적인 압축을 위해 절대적인 위치(position)를 저장하지 않고 위치 사이의 차이(difference) 저장합니다.
 - 예시) 차이 값을 저장하기 위해 3bits만을 사용할 때, 그 차이가 3bits보다 크다면 패딩을 삽입합니다.

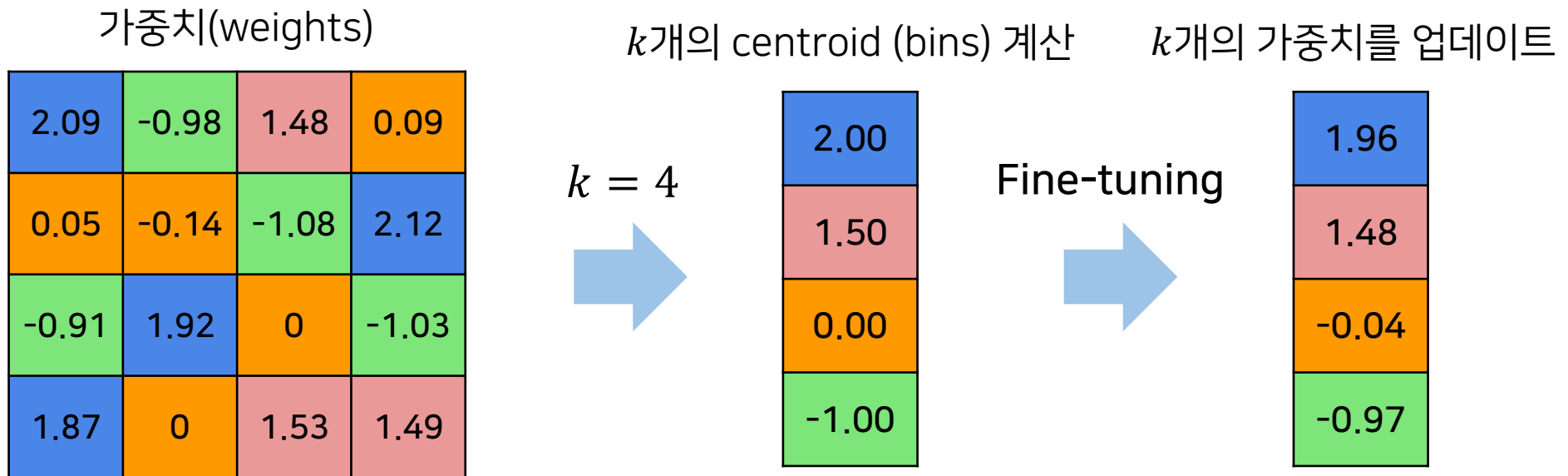
Span Exceeds $8=2^3$

idx	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
diff		1			3								8			3
value		3.4			0.9								0			1.7

↑
Filler Zero

2단계: 학습된 양자화(Trained Quantization)와 가중치 공유(Weight Sharing)

- 양자화: 각 가중치(weight)를 표현하기 위한 비트(bit)의 수를 감소시키는 효과가 있습니다.
 - 실제로 사용할 가중치의 개수 k 를 설정합니다.
 - 해당 k 개의 가중치를 저장한 뒤에 이를 공유(sharing)합니다.
 - 해당 k 개의 가중치를 미세조정(fine-tuning)합니다.
- 예를 들어 각 FC 레이어에서 5-bits (32개의 공유 가중치)만을 사용할 수 있습니다.



2단계 1) 가중치 k개를 생성하여 이를 공유(Sharing)하기

- 4×4 ($n = 16$) 개의 가중치
- $k = 4$ 개의 클러스터
- 각 연결당 비트 수(b) = 32 bits

압축률
(Compression Rate)



$$r = \frac{nb}{n \log_2(k) + kb}$$

$$3.2 = \frac{16 * 32}{16 * 2 + 4 * 32}$$

가중치(Weights)

2.09	-0.98	1.48	0.09
0.05	-0.14	-1.08	2.12
-0.91	1.92	0	-1.03
1.87	0	1.53	1.49

Cluster



Cluster index (2 bit unit)

3	0	2	1
1	1	0	3
0	3	1	0
3	1	2	2

Centroids

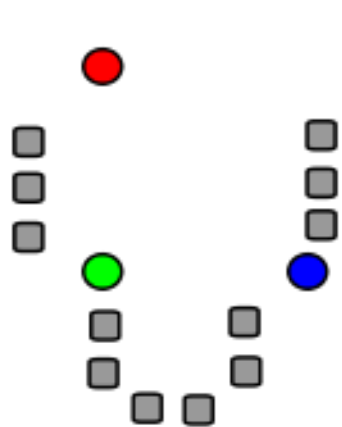
3:	2.00
2:	1.50
1:	0.00
0:	-1.00

- 가중치 공유(Weight Sharing)를 위한 중심점(Centroid) 계산
 - 센트로이드(Centroid) = 학습된 가중치 = 코드북(Codebook)

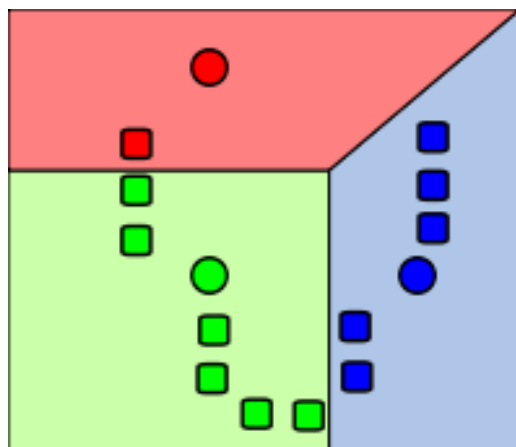
$$\arg \min_C \sum_{i=1}^k \sum_{w \in c_i} |w - c_i|^2$$

[배경지식] K-means Clustering 알고리즘

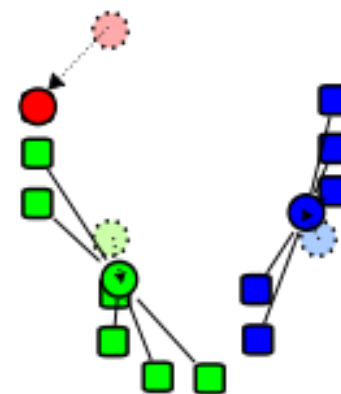
- n 개의 데이터에 대하여 각 집합 내 데이터간 응집도를 최대로 하는 k 개의 집합을 찾는 알고리즘입니다.
 - 1) 먼저 k 개의 중심점(centroid)을 초기화합니다.
 - 2) 각 데이터는 가장 가까이 있는 중심점을 기준으로 클러스터를 구성합니다.
 - 3) 이후에 k 개 클러스터의 중심점의 값을 조정합니다.
 - 4) 수렴할 때까지 2번과 3번의 과정을 반복합니다.



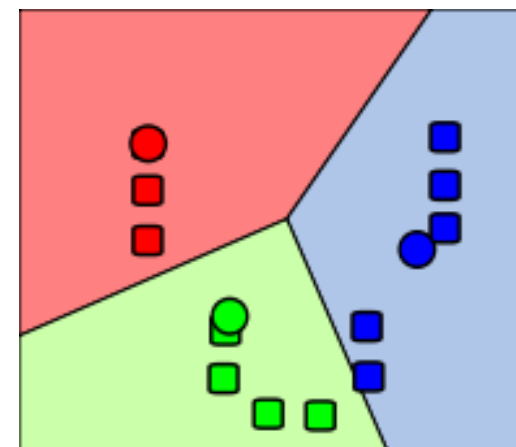
(1) 중심점 초기화



(2) 클러스터 구성



(3) 중심점 조정

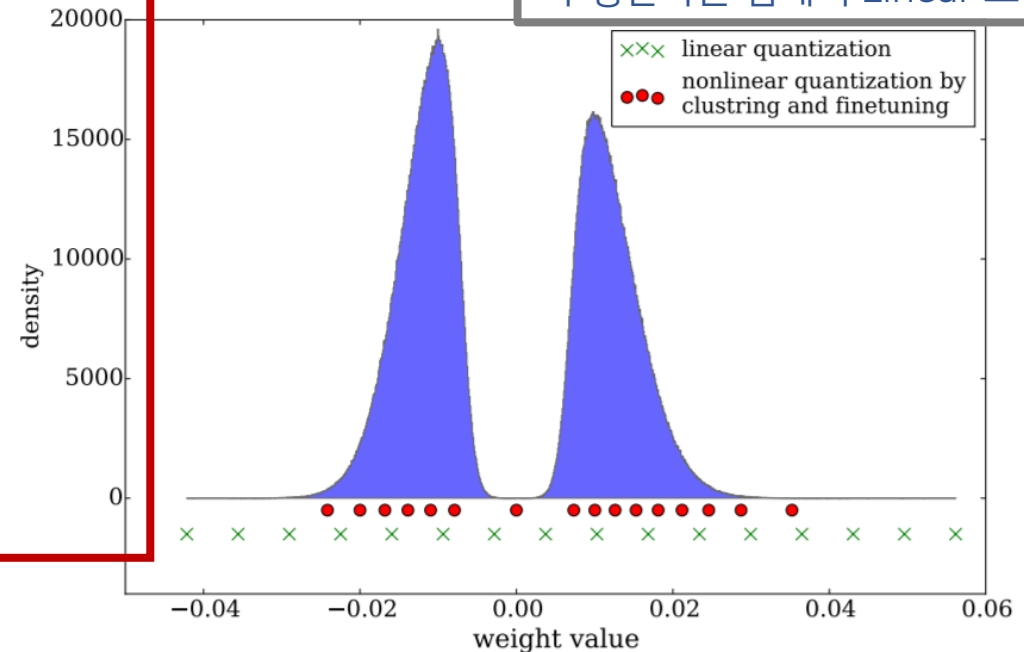
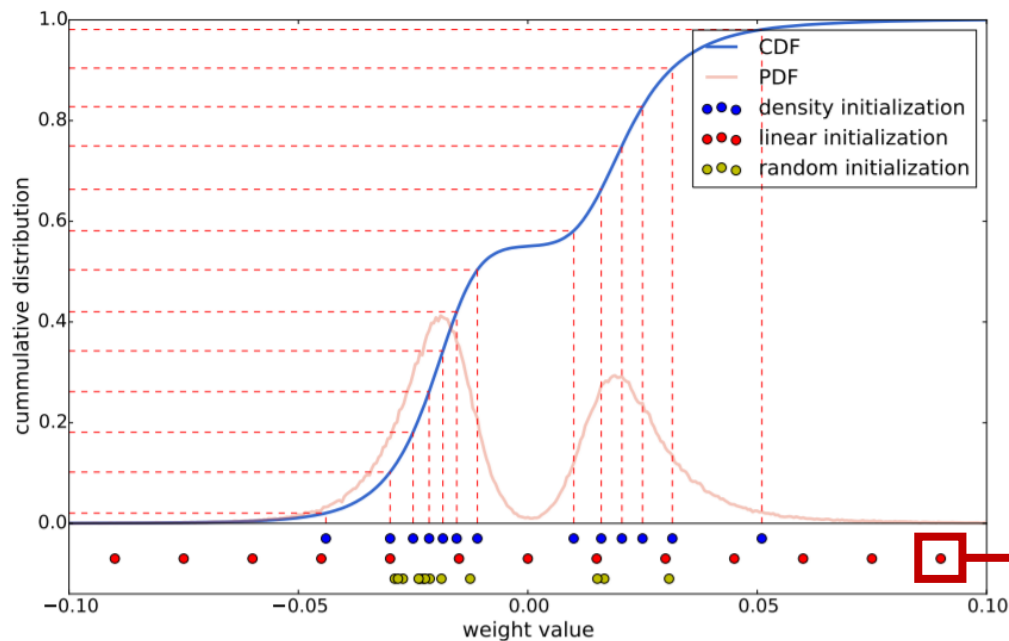


업데이트 결과

https://en.wikipedia.org/wiki/K-means_clustering

2단계 2) 공유된 가중치 초기화(Initialization of Shared Weights)

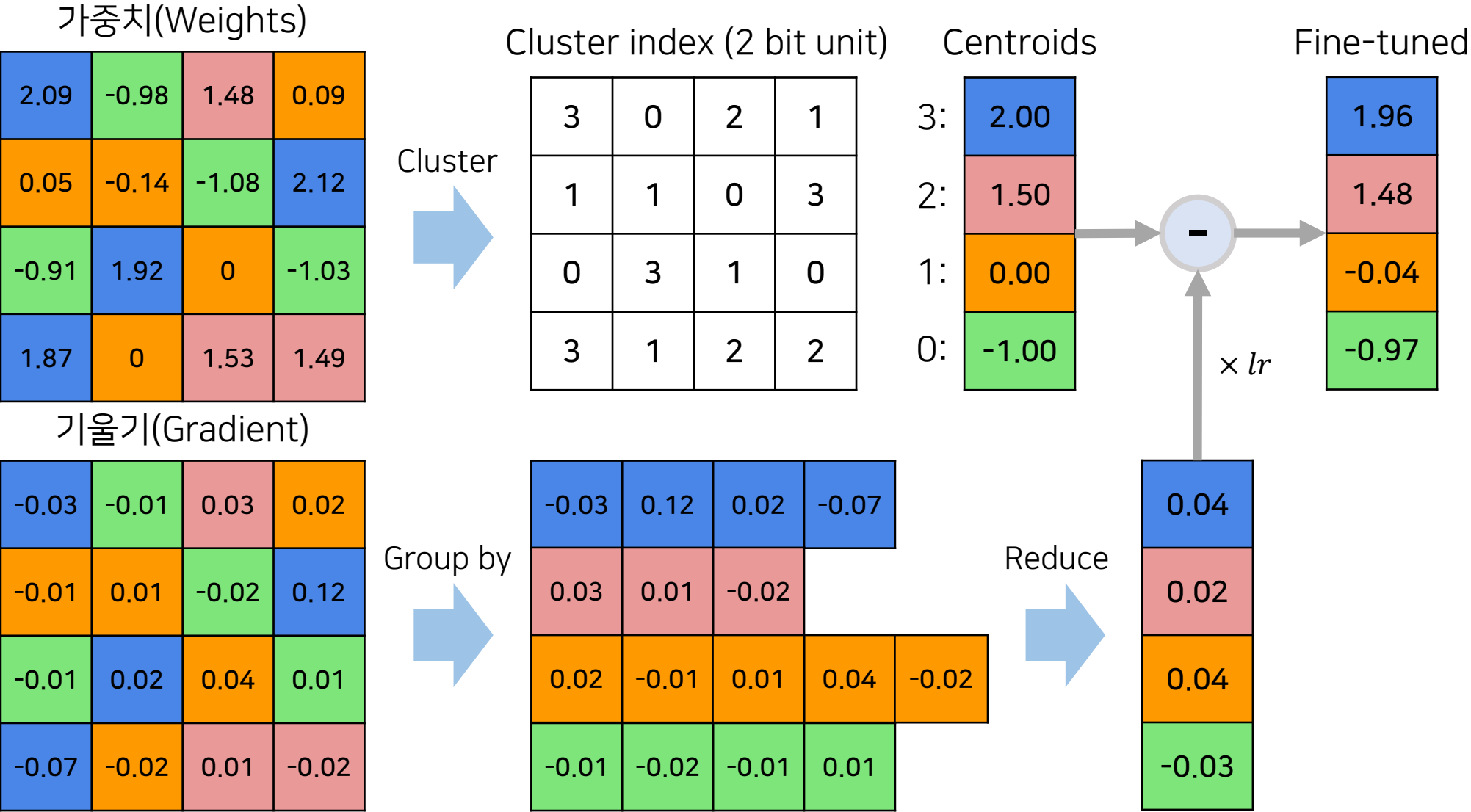
1. Forgy (random) 초기화: 가중치 중에서 랜덤으로 k 개를 선택하여 이것을 초기 중심점으로 사용합니다.
2. Density-based 초기화: 가중치의 CDF에서 y-axis에 대하여 동일한 간격으로 선택합니다.
3. Linear 초기화: 가중치의 [min, max] 사이에서 동일한 간격으로 선택합니다.



절댓값이 큰 가중치가 더 중요한 역할을 수행한다는 점에서 Linear 초기화가 유리

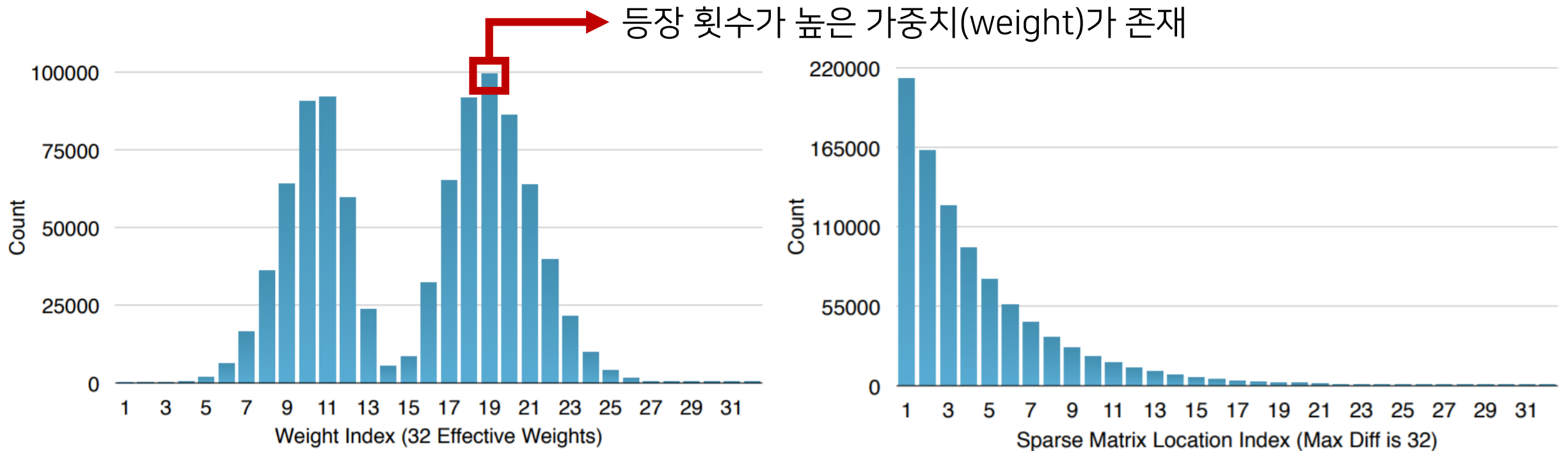
[Figure] Left: Three different methods for centroids initialization. Right: Distribution of weights (blue) and distribution of codebook before (green cross) and after fine-tuning (red dot).

2단계 3) Feed-Forward and Back-Propagation



3단계: 허프만 부호화(Huffman Coding)

- 특정한 중심점(centroid)이 많이 등장합니다. (biased distribution)
- Variable-length codewords: 이처럼 많이 등장하는 심볼(symbol)에 적은 비트(bit)를 할당할 수 있습니다.



[Figure] Distribution for weight (Left) and index (Right). The distribution is biased.

[배경지식] 허프만 부호화(Huffman Coding)

- 저장하고자 하는 데이터(문자열)가 있을 때, 고정 길이(fixed-length) 부호화를 이용하면 어떻게 될까요?
- 저장할 문자열: ABCDABAABBDAAACBADACAACABA (25개의 문자)

[고정 길이(Fixed-length) 부호 테이블]

심볼(Symbol)	부호(Code)
A	00
B	01
C	10
D	11

(25개의 심볼)

ABCDABAABBDAAACBADACAACABA



부호화(Coding)

0001101100010000010111000
0100100110010000010000100

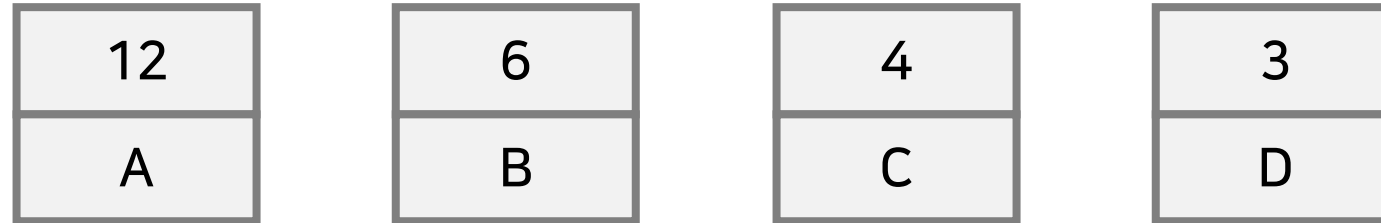
(50 bits)

[배경지식] 허프만 부호화(Huffman Coding)

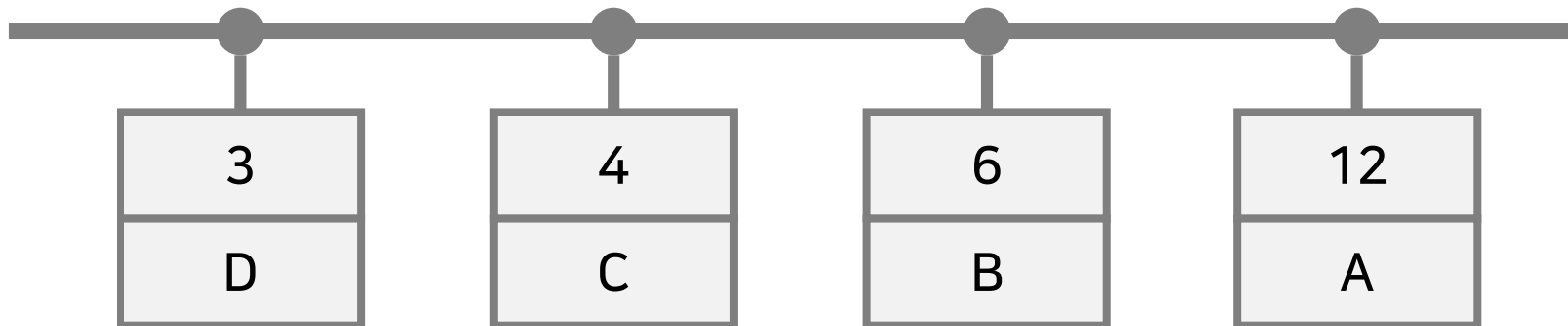
- 허프만 트리(Huffman Tree)를 구축하는 알고리즘은 다음과 같습니다.
 1. 각 심볼(symbol)을 그 출현 빈도와 함께 하나의 노드로 만듭니다.
 2. 모든 노드를 우선순위 큐에 삽입합니다.
 3. 우선순위 큐에 노드가 하나 남을 때까지 아래 과정을 반복합니다.
 - 1) 우선순위 큐에서 두 개의 노드를 추출합니다.
 - 2) 두 개의 노드를 자식 노드로 하는 새로운 노드를 생성하여 우선순위 큐에 삽입합니다.

[배경지식] 허프만 부호화(Huffman Coding)

- Step 1) 각 심볼을 출현 빈도와 함께 묶어 노드로 만듭니다.

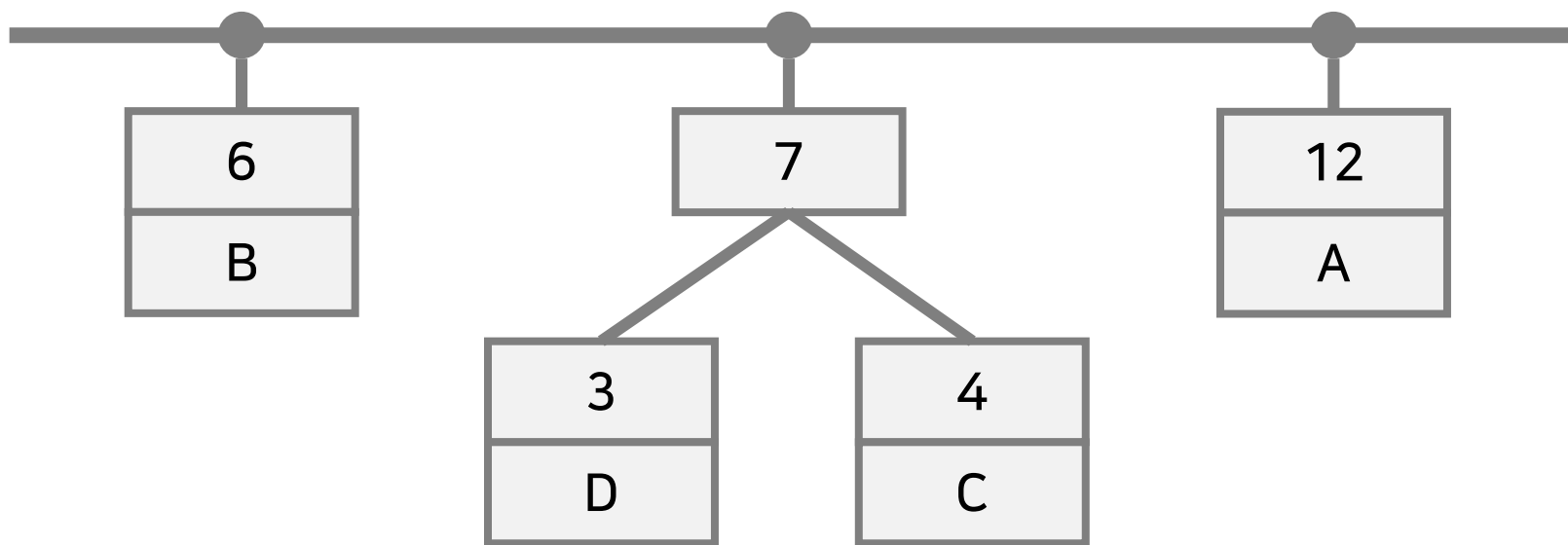


- Step 2) 모든 노드를 우선순위 큐에 삽입합니다.



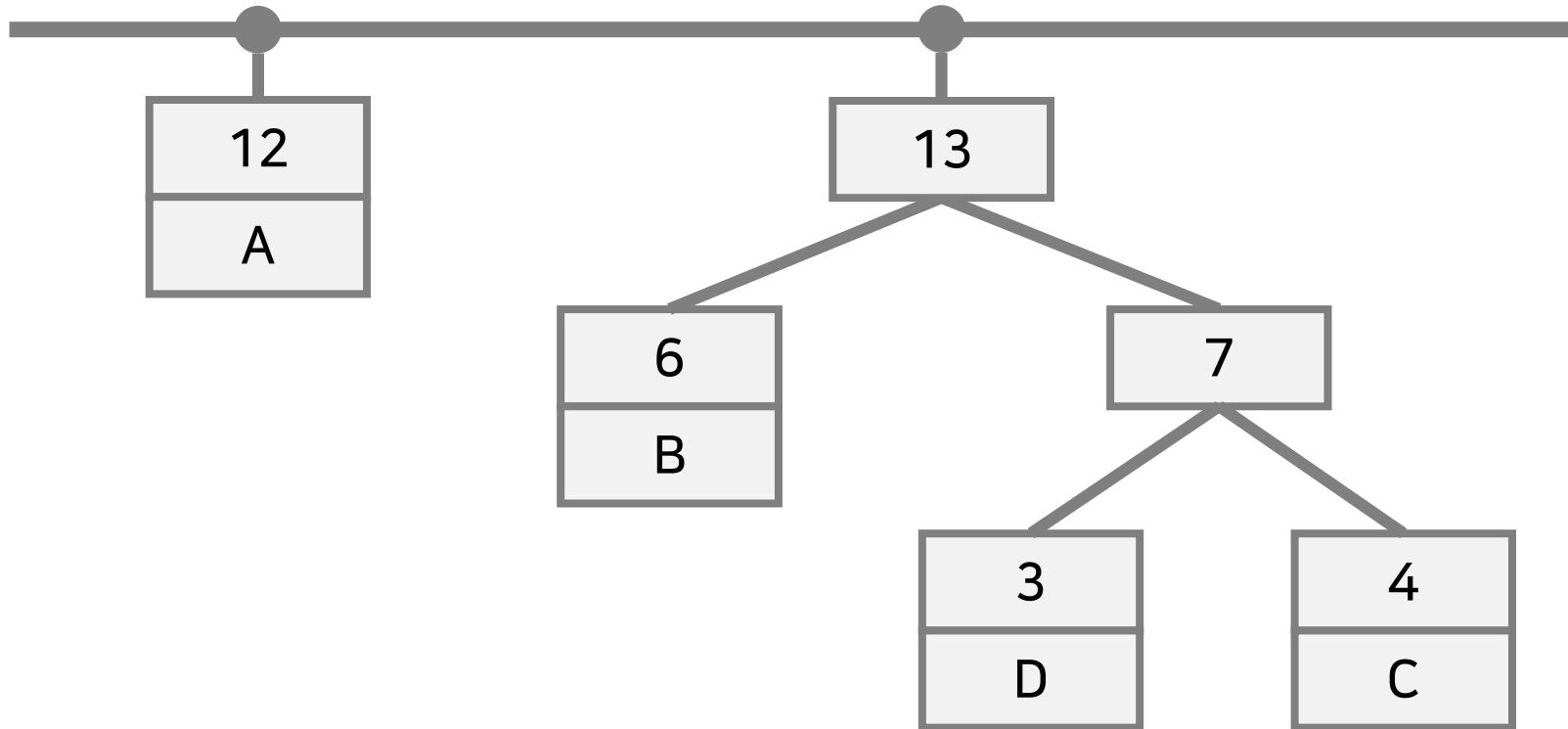
[배경지식] 허프만 부호화(Huffman Coding)

- Step 3) 가장 작은 두 노드(키의 값이 3인 노드와 키의 값이 4인 노드)를 꺼내 합친 뒤에 다시 삽입합니다.



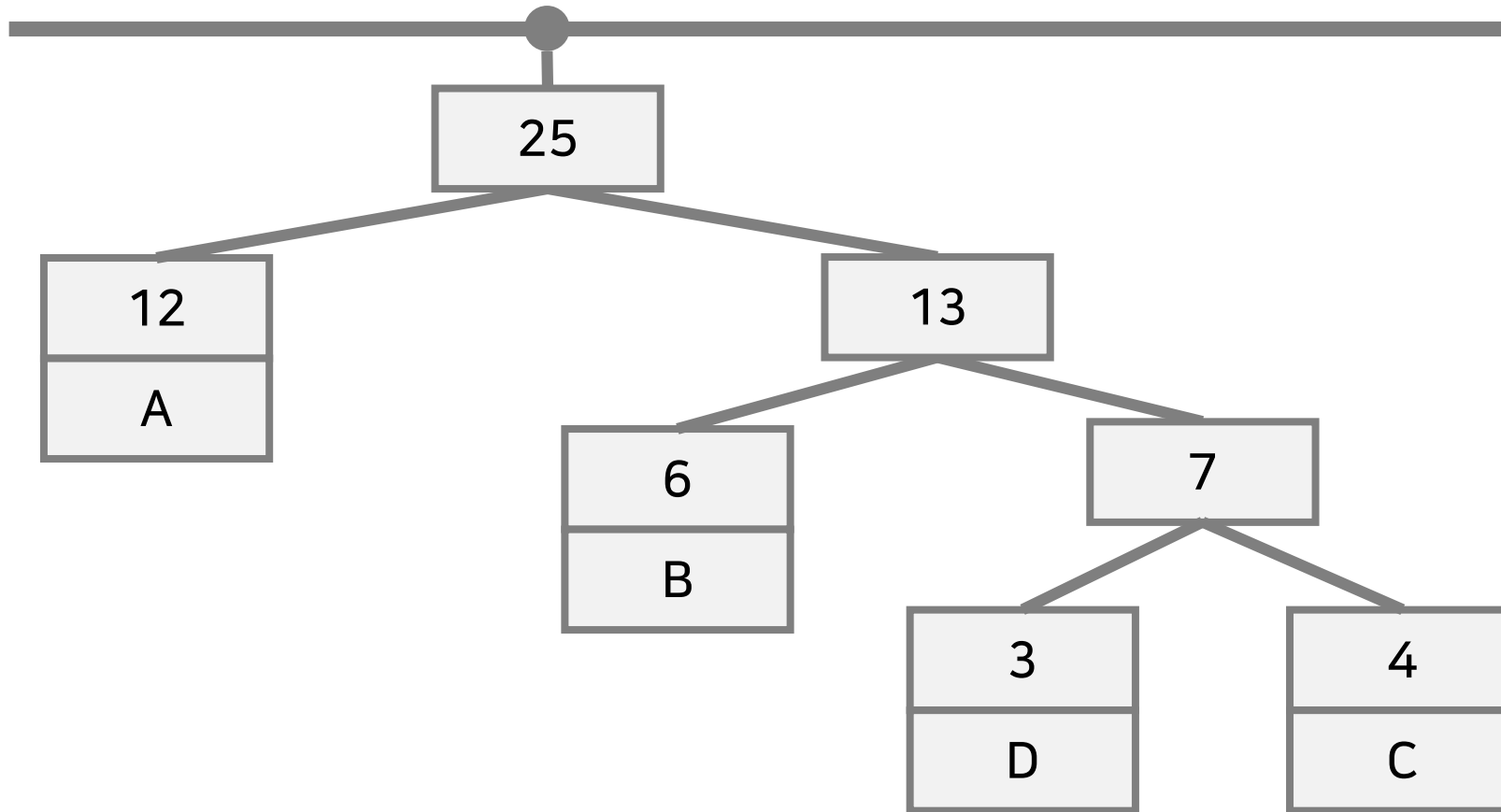
[배경지식] 허프만 부호화(Huffman Coding)

- Step 4) 가장 작은 두 노드(키의 값이 6인 노드와 키의 값이 7인 노드)를 꺼내 합친 뒤에 다시 삽입합니다.



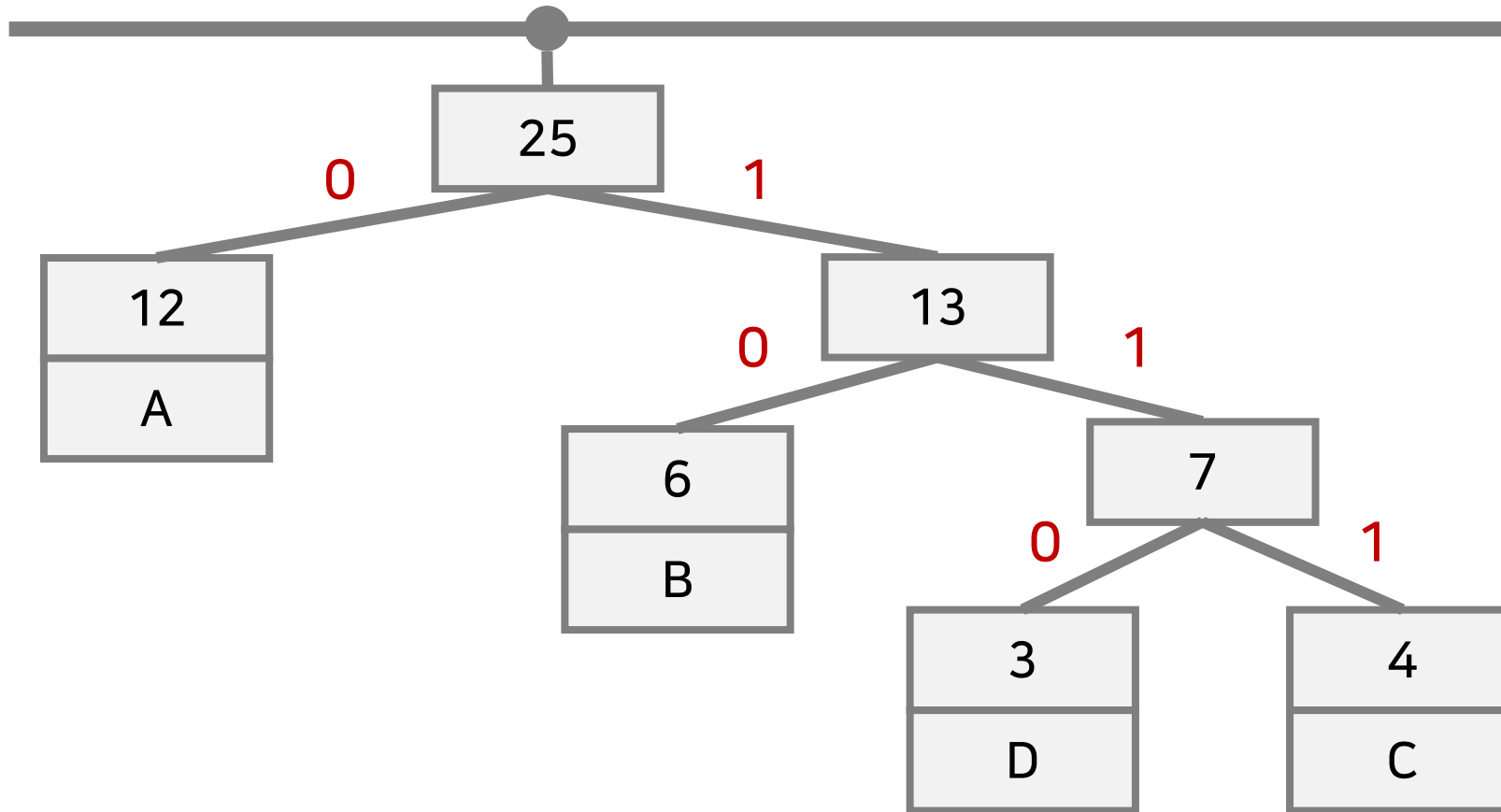
[배경지식] 허프만 부호화(Huffman Coding)

- Step 5) 가장 작은 두 노드(키의 값이 12인 노드와 키의 값이 13인 노드)를 꺼내 합친 뒤에 다시 삽입합니다.
- 남은 노드의 개수가 1개이므로, 여기에서 허프만 트리 구축이 완료됩니다.



[배경지식] 허프만 부호화(Huffman Coding)

- 허프만 트리를 구축한 뒤에는 각각의 노드의 위치까지 재귀적으로 조회하며, 왼쪽 경로에 0을, 오른쪽 경로에 1을 부여합니다. 결과적으로 다음과 같이 각 심볼에 대하여 다른 길이의 부호(code)가 부여됩니다.



[배경지식] 허프만 부호화(Huffman Coding)

- 저장할 문자열: ABCDABAABBDAAACBADACAACABA (25개의 문자)
- 허프만 부호화를 이용해 가변 길이(variable-length) 부호화를 진행한 결과는 다음과 같습니다.

[가변 길이(Variable-length) 부호 테이블]

심볼(Symbol)	부호(Code)
A	0
B	10
C	110
D	111

(25개의 심볼)

ABCDABAABBDAAACBADACAACABA



Huffman Coding

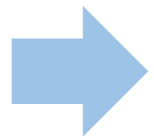
0101101110100010101110011
01001110110001100100

(45 bits)

실험 결과 분석: 네 개의 네트워크 아키텍처를 대상으로 실험 진행

- 본 논문에서 제안한 Deep Compression은 네트워크의 용량을 35배에서 49배까지 압축할 수 있었습니다.
 - 10MB 정도의 크기는 on-chip SRAM에 적재할 수 있을 정도로 작은 크기입니다.
 - 정확도(accuracy)는 거의 그대로 유지되는 것을 확인할 수 있습니다.

Network	Top-1 Error	Top-5 Error	Parameters	Compress Rate
LeNet-300-100 Ref	1.64%	-	1070 KB	
LeNet-300-100 Compressed	1.58%	-	27 KB	40×
LeNet-5 Ref	0.80%	-	1720 KB	
LeNet-5 Compressed	0.74%	-	44 KB	39×
AlexNet Ref	42.78%	19.73%	240 MB	
AlexNet Compressed	42.78%	19.70%	6.9 MB	35×
VGG-16 Ref	31.50%	11.32%	552 MB	
VGG-16 Compressed	31.17%	10.91%	11.3 MB	49×



ILSVRC-2012 데이터 세트에 대하여 좋은 성능을 보입니다.
CONV 레이어의 가중치는 8 bits, FC 레이어는 5 bits를 사용합니다.

실험 결과 분석: VGG-16 아키텍처

- VGG-16 아키텍처의 경우 $138\text{M} \times 4\text{B} = 552\text{MB}$ 에서 11.3MB ($552\text{MB} \times 2.05\%$)로 압축 가능합니다.
 - 압축 과정: Pruning(13X) → Quantization(31X) → Huffman Coding(49X)
 - 원래 각 가중치(weight)는 32 bits을 사용하지만, 각각 8 bits (CONV)와 5 bits (FC)로 Quantization을 수행합니다.

Layer	#Weights	Weights% (P)	Weigh bits (P+Q)	Weight bits (P+Q+H)	Index bits (P+Q)	Index bits (P+Q+H)	Compress rate (P+Q)	Compress rate (P+Q+H)
conv1_1	2K	58%	8	6.8	5	1.7	40.0%	29.97%
conv1_2	37K	22%	8	6.5	5	2.6	9.8%	6.99%
conv2_1	74K	34%	8	5.6	5	2.4	14.3%	8.91%
conv2_2	148K	36%	8	5.9	5	2.3	14.7%	9.31%
conv3_1	295K	53%	8	4.8	5	1.8	21.7%	11.15%
conv3_2	590K	24%	8	4.6	5	2.9	9.7%	5.67%
conv3_3	590K	42%	8	4.6	5	2.2	17.0%	8.96%
conv4_1	1M	32%	8	4.6	5	2.6	13.1%	7.29%
conv4_2	2M	27%	8	4.2	5	2.9	10.9%	5.93%
conv4_3	2M	34%	8	4.4	5	2.5	14.0%	7.47%
conv5_1	2M	35%	8	4.7	5	2.5	14.3%	8.00%
conv5_2	2M	29%	8	4.6	5	2.7	11.7%	6.52%
conv5_3	2M	36%	8	4.6	5	2.3	14.8%	7.79%
fc6	103M	4%	5	3.6	5	3.5	1.6%	1.10%
fc7	17M	4%	5	4	5	4.3	1.5%	1.25%
fc8	4M	23%	5	4	5	3.4	7.1%	5.24%
Total	138M	7.5%(13×)	6.4	4.1	5	3.1	3.2% (31×)	2.05% (49×)

[Table] Compression statistics for VGG-16. P: pruning, Q: quantization, H: Huffman coding.

실험 결과 분석: AlexNet 성능 분석

- 본 논문의 메서드를 사용해 AlexNet을 압축했을 때 압축률이 가장 높았습니다. 240MB → 6.9MB (35X)
 - 압축 결과: 정확도(accuracy)의 감소가 거의 없다는 점에서 성능이 우수합니다.

Network	Top-1 Error	Top-5 Error	Parameters	Compress Rate
Baseline Caffemodel (BVLC)	42.78%	19.73%	240MB	1×
Fastfood-32-AD (Yang et al., 2014)	41.93%	-	131MB	2×
Fastfood-16-AD (Yang et al., 2014)	42.90%	-	64MB	3.7×
Collins & Kohli (Collins & Kohli, 2014)	44.40%	-	61MB	4×
SVD (Denton et al., 2014)	44.02%	20.56%	47.6MB	5×
Pruning (Han et al., 2015)	42.77%	19.67%	27MB	9×
Pruning+Quantization	42.78%	19.70%	8.9MB	27×
Pruning+Quantization+Huffman	42.78%	19.70%	6.9MB	35×

[Table] Comparison with other compression methods on AlexNet.

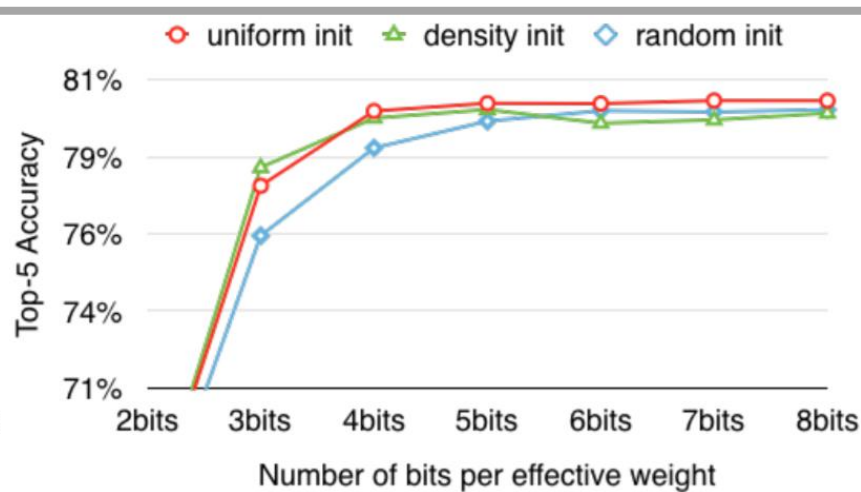
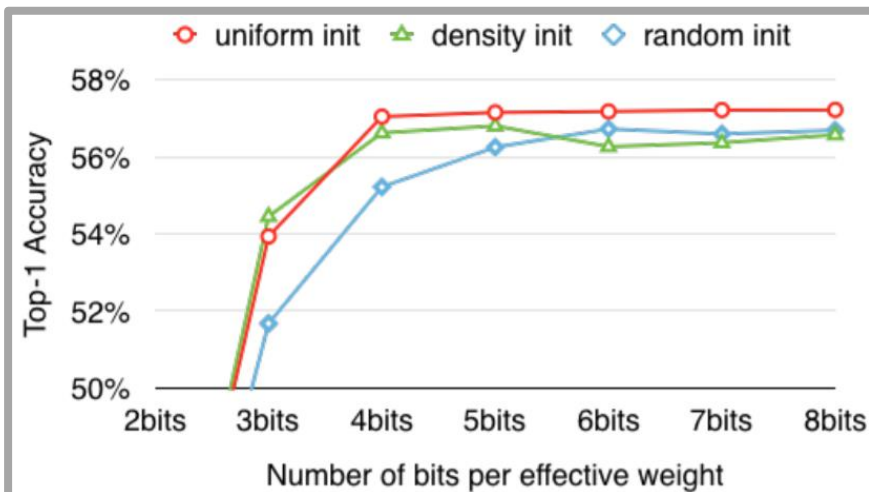
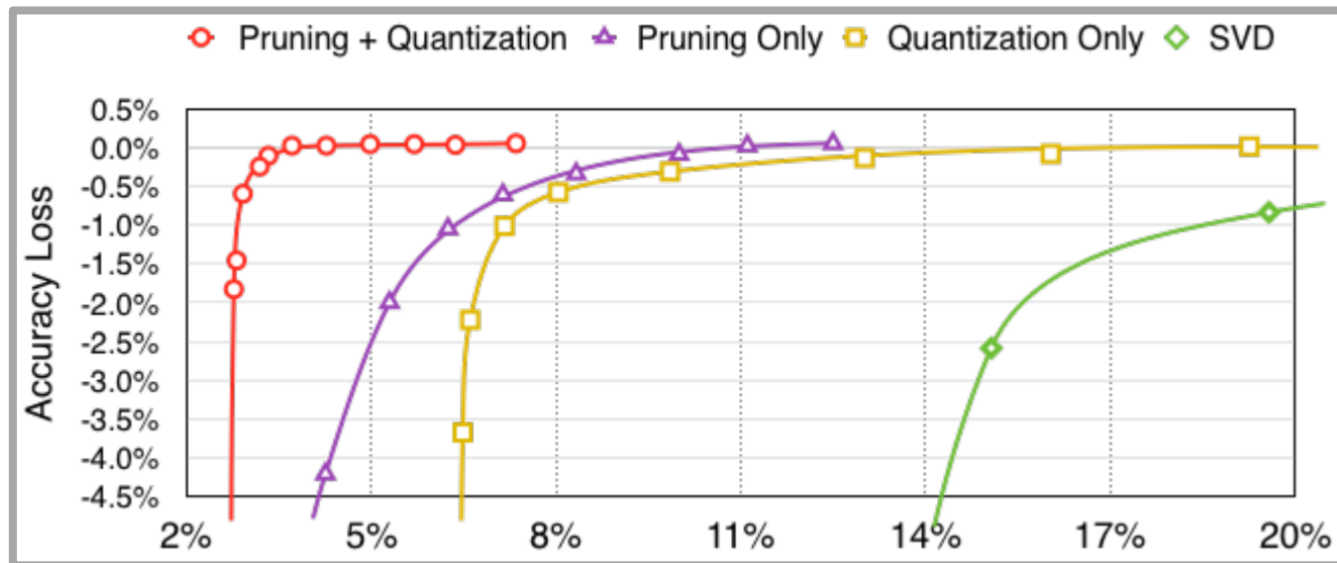
- 8 bits / 5 bits 조합을 사용할 때 정확도의 감소가 없음
- 8 bits / 4 bits는 하드웨어 친화적

#CONV bits / #FC bits	Top-1 Error	Top-5 Error	Top-1 Error Increase	Top-5 Error Increase
32bits / 32bits	42.78%	19.73%	-	-
8 bits / 5 bits	42.78%	19.70%	0.00%	-0.03%
8 bits / 4 bits	42.79%	19.73%	0.01%	0.00%
4 bits / 2 bits	44.77%	22.33%	1.99%	2.60%

[Table] Accuracy of AlexNet with different aggressiveness of weight sharing and quantization.

Discussion: 가지치기(Pruning)과 양자화(Quantization) 같이 사용하기

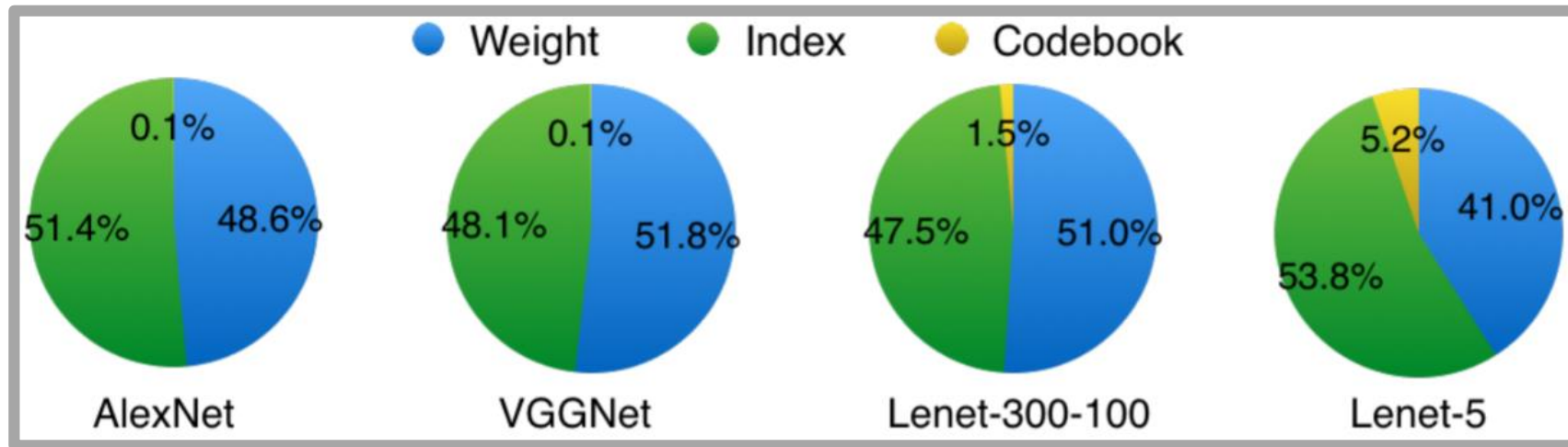
- Pruning과 Quantization을 함께 사용할 때 가장 성능이 우수합니다.
- 개별적으로 사용할 때보다 압축 성능이 훨씬 좋은 것을 확인할 수 있습니다.



- 큰 가중치 값을 유지하기 위하여 Linear 초기화 방식을 사용합니다.
- 뉴럴 네트워크의 특성상 Linear 초기화 방식이 가장 우수한 성능을 보입니다.

Discussion: 압축 결과 각각의 컴포넌트가 차지하는 비율

- 각각의 컴포넌트가 차지하는 비율을 확인할 수 있습니다. (필요한 추가적인 공간 모두 반영)
 - Pruning은 가중치 행렬을 희소 행렬로 바꾸므로 인덱스를 저장하기 위한 추가적인 공간이 필요합니다.
 - Quantization은 코드북(codebook)을 위한 추가적인 공간이 필요합니다.
- 비율을 확인해 보면, Codebook이 추가됨에 따른 오버헤드는 무시할 만한 수준인 것을 알 수 있습니다.



[Figure] Storage ratio of weight, index and codebook.

Conclusion

- The authors propose “**Deep Compression**” that compressed neural networks without affecting accuracy.
 1. **Pruning** the unimportant connections.
 2. **Quantizing** the network using weight sharing.
 3. Applying **Huffman coding**.
- Various networks can be compressed by **35x** to **49x** without loss of accuracy.
- Deep compression facilitates the use of complex neural networks in mobile applications.