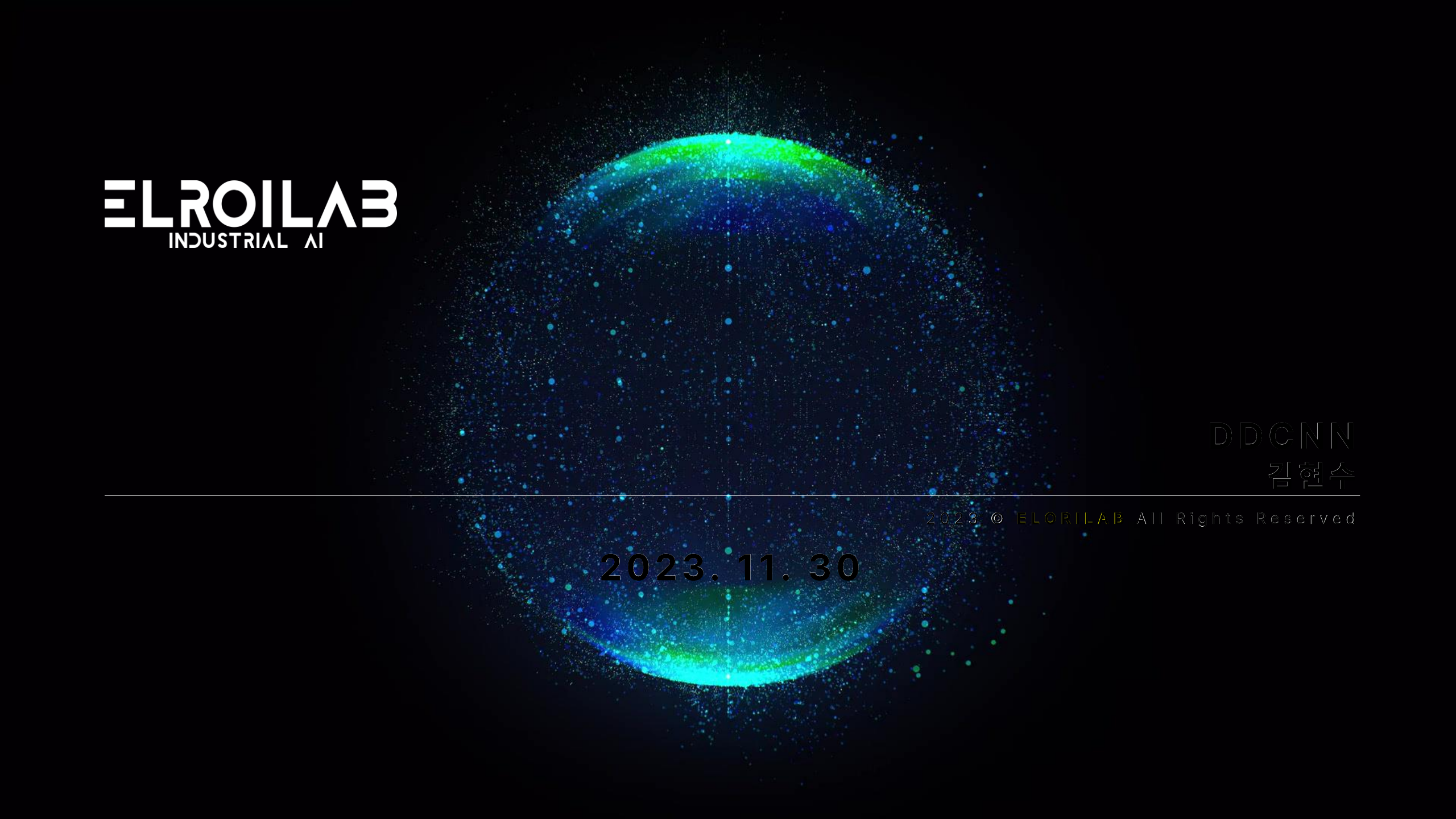


DDCNN

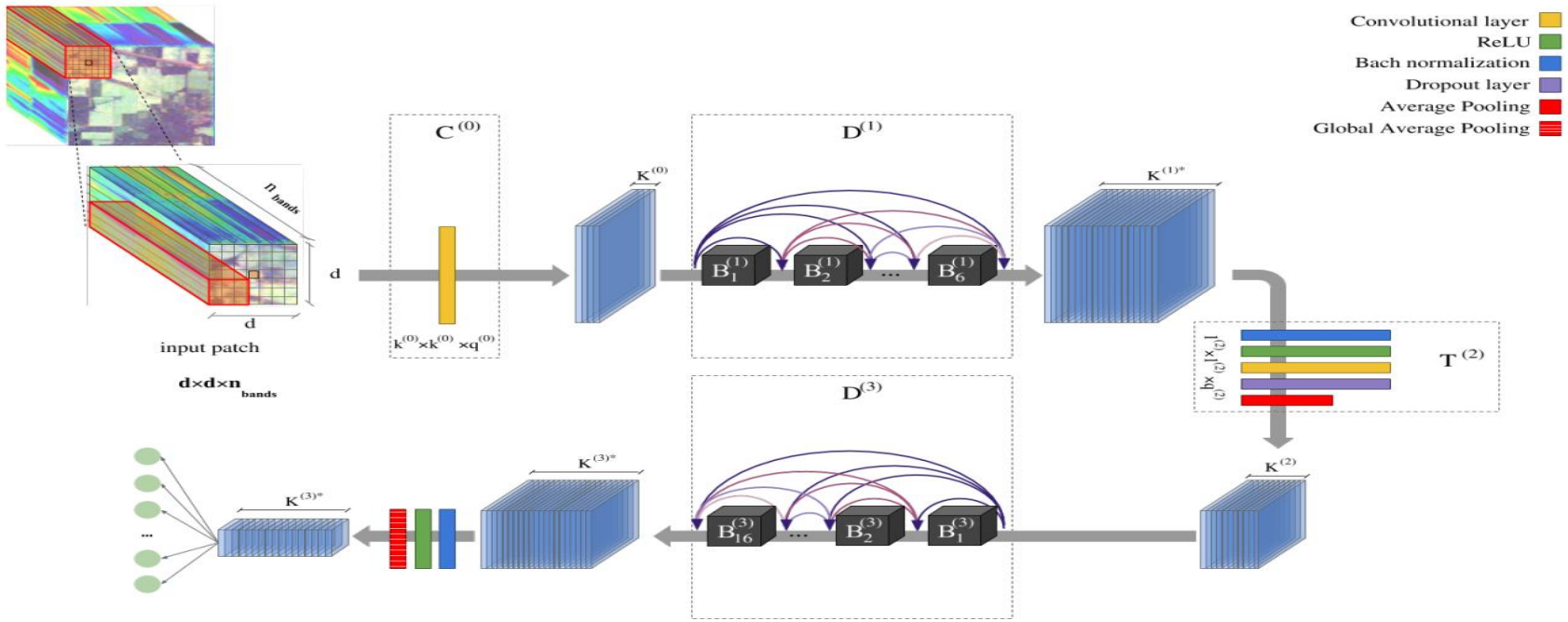
김현수

2023. 11. 30

DDCNN

Contribution

- It exploits the rich and diverse amount of information contained in HSI data
- It improves the network generalization while avoiding the vanishing of the model gradient
- It combines both low-level and high-level features in the classification process
- It can perform properly in the presence of limited training samples



DDCNN

모델 구조 및 파라미터

Input Convolutional Layer	
$C^{(0)}$	kernels = $16 \times 3 \times 3 \times n_{bands}$, $s = 1$
First dense block	
$D^{(1)}$	kernels = $\begin{bmatrix} (g \cdot \alpha) \times 1 \times 1 \times (16 + (n-1) \cdot g) \\ g \times 3 \times 3 \times (g \cdot \alpha) \end{bmatrix} \cdot 6$ with $n = 1, 2, \cdots, 6$, $s = 1$, ReLU, dropout = 10%
Transition layer	
$T^{(2)}$	kernels = $\frac{K^{(1)*}}{2} \times 1 \times 1 \times K^{(1)*}$, with $K^{(1)*} = 16 + 6 \cdot g$, $s = 1$, ReLU, dropout = 10% Average Pooling 2×2, $s = 2$
Second dense block	
$D^{(3)}$	kernels = $\begin{bmatrix} (g \cdot \alpha) \times 1 \times 1 \times (\frac{K^{(1)*}}{2} + (n-1) \cdot g) \\ g \times 3 \times 3 \times (g \cdot \alpha) \end{bmatrix} \cdot 16$ with $n = 1, 2, \cdots 16$, $s = 1$, ReLU, dropout = 10%
Classification layers	
ReLU, Global Average Pooling with output $1 \times 1 \times K^{(3)*}$, with $K^{(3)*} = \frac{K^{(1)*}}{2} + 16 \cdot g$ Fully connected of $n_{classes}$ layers, with softmax	

데이터 셋 구성

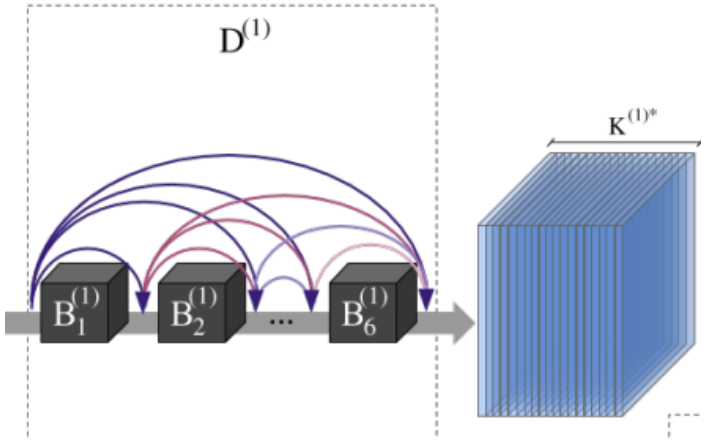
- 이번 논문 구현에서 사용한 데이터 셋은 HSI classification에서 많이 쓰기로 유명한 Indian Pines 데이터 셋을 사용
- Dataset을 구성할 때 image, ground truth, index, patch size를 파라미터로 사용
- Center pixel을 index 파라미터로 받아오고 그에 맞는 patch image와 ground truth를 사용한다.

```
class hsi_dataset(Dataset):
    def __init__(self, img_dir, gt_dir, index_num, patch_size):
        super().__init__()
        img_mat = io.loadmat(img_dir)
        gt_mat = io.loadmat(gt_dir)
        self.img = img_mat['indian_pines_corrected']
        self.gt = gt_mat['indian_pines_gt']
        self.d = patch_size
        self.boundary = int(self.d / 2)
        self.img_shape = self.img.shape[0]
        self.index_num = index_num

    def __len__(self):
        return len(self.index_num)

    def __getitem__(self, index):
        center_pixel = self.index_num[index]
        img_pixel = self.img[center_pixel[0] - self.boundary:center_pixel[0] + self.boundary + 1, center_pixel[1] - self.boundary:center_pixel[1] + self.boundary + 1]
        gt_pixel = self.gt[center_pixel[0], center_pixel[1]]
        img_pixel = np.float32(img_pixel)
        img_pixel = torch.from_numpy(img_pixel)
        gt_pixel = torch.as_tensor(gt_pixel)
        return  img_pixel.permute(2, 0, 1), gt_pixel-1
```

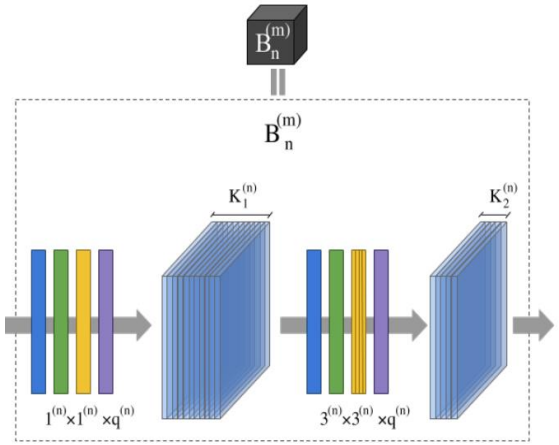
DDCNN

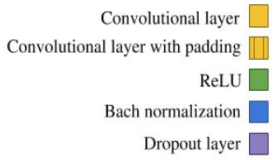


Dense Block의 구조는 Resnet에서 영감을 얻었다.
일반적인 CNN 같은 구조는 layer의 수 만큼 n개의 connection이 존재하지만,
이와 같은 구조는 n(n + 1) / 2개의 connection이 존재하게 된다.

이때 생기는 의문은 이러한 connection의 증가가 model parameters 의 증가를 뜻하는 것이 아니냐는 점이다.

다만 Resnet과 같은 구조는 이전 정보를 재사용 함으로서 오히려 feature를 강화 시키고, overfitting, vanishing gradient 문제를 줄일 수 있다.





Inner block에서 normalization을 처음 해주는 이유는 scaling을 해주고, internal covariance shift현상을 막기 위해서이다.
첫 bottleneck $K_1^{(n)}$에서는 spatial dimension에 영향을 주는 일 없이 output의 depth를 변화시키는 것이 목표
두번째 bottleneck $K_2^{(n)}$에서는 spatial-spectral features를 추출하는 것이 목표인데, 실제 데이터 concatenation을 수행하기 위해 convolution 작업을 거친 후에 spatial dimension을 유지하는 것이 중요하기 때문에 zero-padding을 추가한다.

DDCNN

$D^{(1)}$에 쓰이는 inner block과 $D^{(3)}$에 쓰이는 inner block은 channel을 다르게 다루기 때문에 나뉘어서 정의

```
self.first_inner_block = nn.Sequential(
    nn.BatchNorm2d(16 + (self.num - 1) * self.gamma),
    nn.ReLU(),
    nn.Conv2d(in_channels = 16 + (self.num - 1) * self.gamma, out_channels = self.gamma * self.alpha, kernel_size = 1, stride = 1),
    nn.Dropout(p = 0.1),

    nn.BatchNorm2d(self.gamma * self.alpha),
    nn.ReLU(),
    nn.Conv2d(in_channels = self.gamma * self.alpha, out_channels = self.gamma, kernel_size = 3, stride = 1, padding = 'same'),
    nn.Dropout(p = 0.1)
)
self.second_inner_block = nn.Sequential(
    nn.BatchNorm2d(self.k1 // 2 + (self.num - 1) * self.gamma),
    nn.ReLU(),
    nn.Conv2d(in_channels = self.k1 // 2 + (self.num - 1) * self.gamma, out_channels = self.gamma * self.alpha, kernel_size = 1, stride = 1),
    nn.Dropout(p = 0.1),

    nn.BatchNorm2d(self.gamma * self.alpha),
    nn.ReLU(),
    nn.Conv2d(in_channels = self.gamma * self.alpha, out_channels = self.gamma, kernel_size = 3, stride = 1, padding = 'same'),
    nn.Dropout(p = 0.1)
)
```

만들어둔 inner block을 쌓아서 Dense layer 생성

```
self.first_dense_layer = []
for i in range(6):
    self.first_dense_layer.append(Dense_block(True, i + 1).cuda())

self.second_dense_layer = []
for i in range(16):
    self.second_dense_layer.append(Dense_block(False, i + 1).cuda())
```

```
for i in range(6):
    output = self.first_dense_layer[i](output)
    feature_maps.append(output)
    for j in range(len(feature_maps) - 1):
        output = torch.cat([output, feature_maps[j]],1)
```

각 Dense layer 실행할때 feature들을 concat 해주어 사용

DDCNN

- 이전 성과요약에서 부족했던 부분에 대한 보완점

  - Training 방식 변화
    - 기존에는 model에 대해 training만 진행 -> OA : 95.43 ± 0.06%, Kappa : 95.49 ± 0.8%
    - training 을 진행하면서 일정 epoch 마다 validation 진행 -> 98.55 ± 0.4%, 98.37±0.42%
      - 10 epoch 마다 진행
      - early stop 적용(early stop 기준은 5로 설정)

  - test 방식 변화
    - 기존에는 각 class별로 dataset, dataloader를 만들어 test 적용
    - Indian Pines dataset을 전부 test dataset으로 적용하여 ground truth label 별로 나누어서 계산

DDCNN

Classification result

- Patch size : 11 x 11, epoch : 100, batch size : 100, 15% training data

Class	Average	Class	Average
1	94.98±4.8%	9	98.23±1.8%
2	98.16±1.6%	10	98.44±1.3%
3	98.14±1.8%	11	99.03±0.8%
4	98.17±1.3%	12	98.04±1.4%
5	98.31±1.1%	13	99.13±0.3%
6	99.02±0.8%	14	99.04±1%
7	98.33±1.7%	15	98.25±1.3%
8	99.77±0.3%	16	98.98±1.1%
OA	98.55±0.4%	Kappa	98.37±0.42%

- Patch size : 9 x 9, epoch : 100, batch size : 100, 15% training data

Class	Average	Class	Average
1	96.6%	9	96.7%
2	97.45%	10	97.03%
3	97.08%	11	98.02%
4	95.44%	12	97.46%
5	97.4%	13	99.75%
6	99.3%	14	99.65%
7	95.02%	15	97.65%
8	99.82%	16	97.91%
OA	97.79%	Kappa	97.3%


감사합니다.