AlexNet

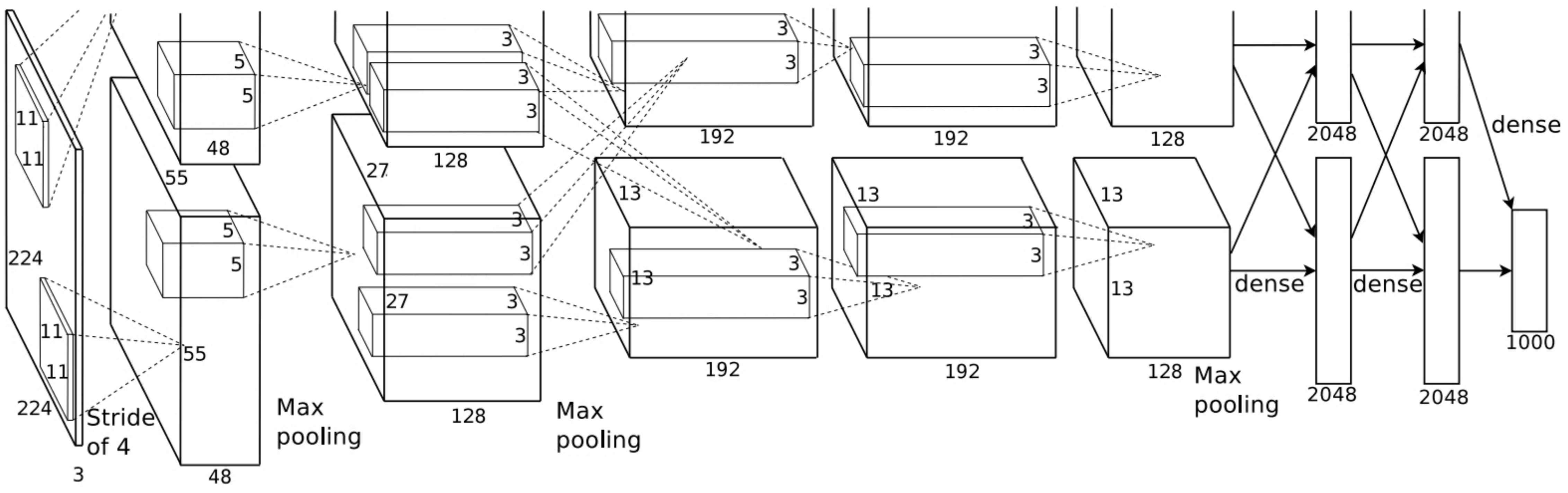
ImageNet Classification with Deep Convolutional Neural Networks

전종섭

1. Introduction

- To improve performance
  - Collect larger datasets
  - learn more powerful models
  - preventing overfitting

3. The Architecture



- ReLU Nonlinearity
- Training on Multiple GPUs
- Local Response Normalization
- Overlapping Pooling
- Overall Architecture

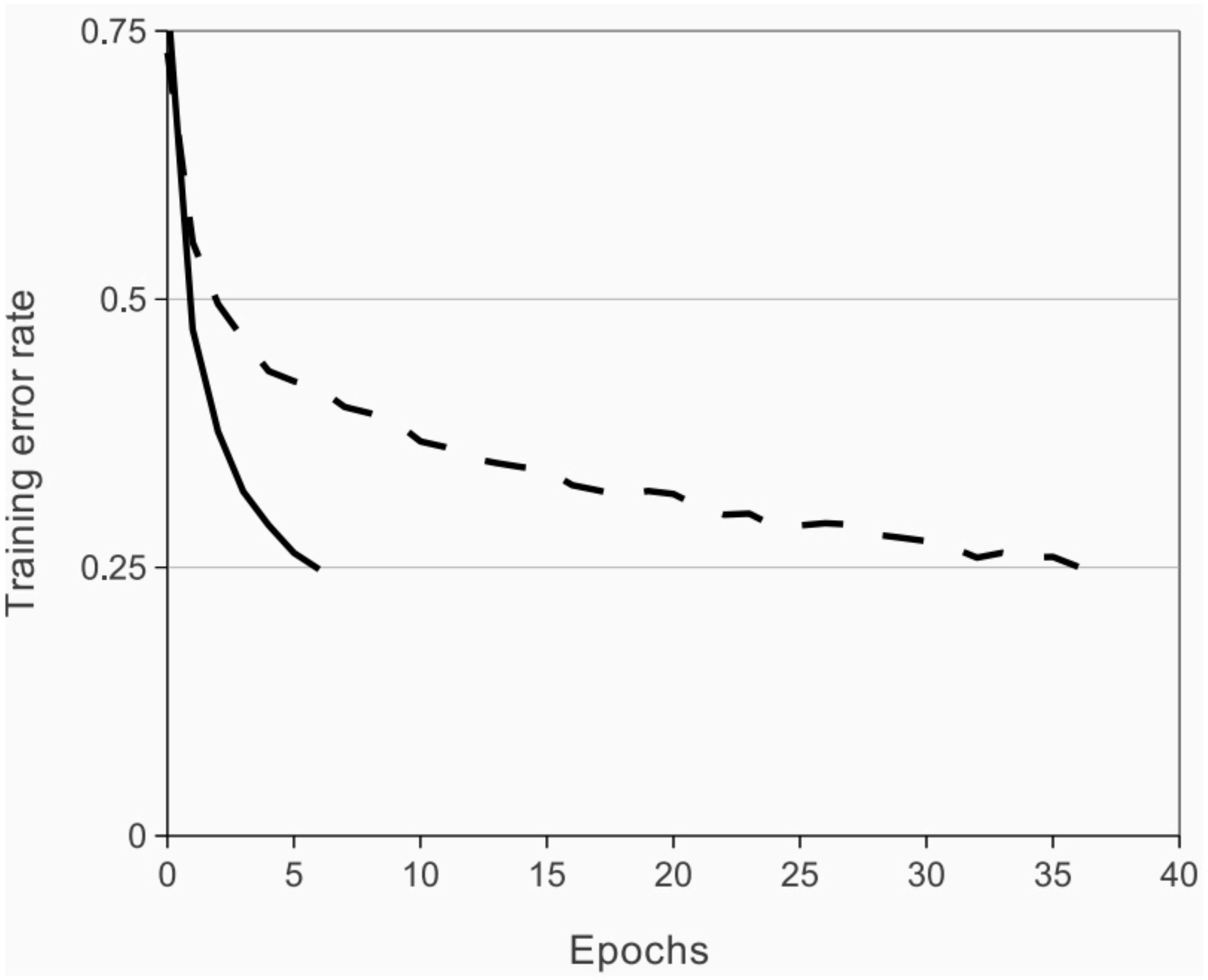
3.1 ReLU Nonlinearity

- Sigmoid
  - Standard way to model a neuron’s output
  - Biological Neuron 모델링을 위한 활성화 함수
  - Saturating nonlinearity are much slower than non-saturating nonlinearity ReLU
- Tanh (Hyperbolic tangent)
  - Sigmoid 보다 개선된 학습 속도

➡ 그러나 Sigmoid 와 Tanh 둘 다 망의 크기가 작을 경우 문제가 없지만, 망의 크기가 클 경우 치명적인 속도 문제가 있다.

3.1 ReLU Nonlinearity

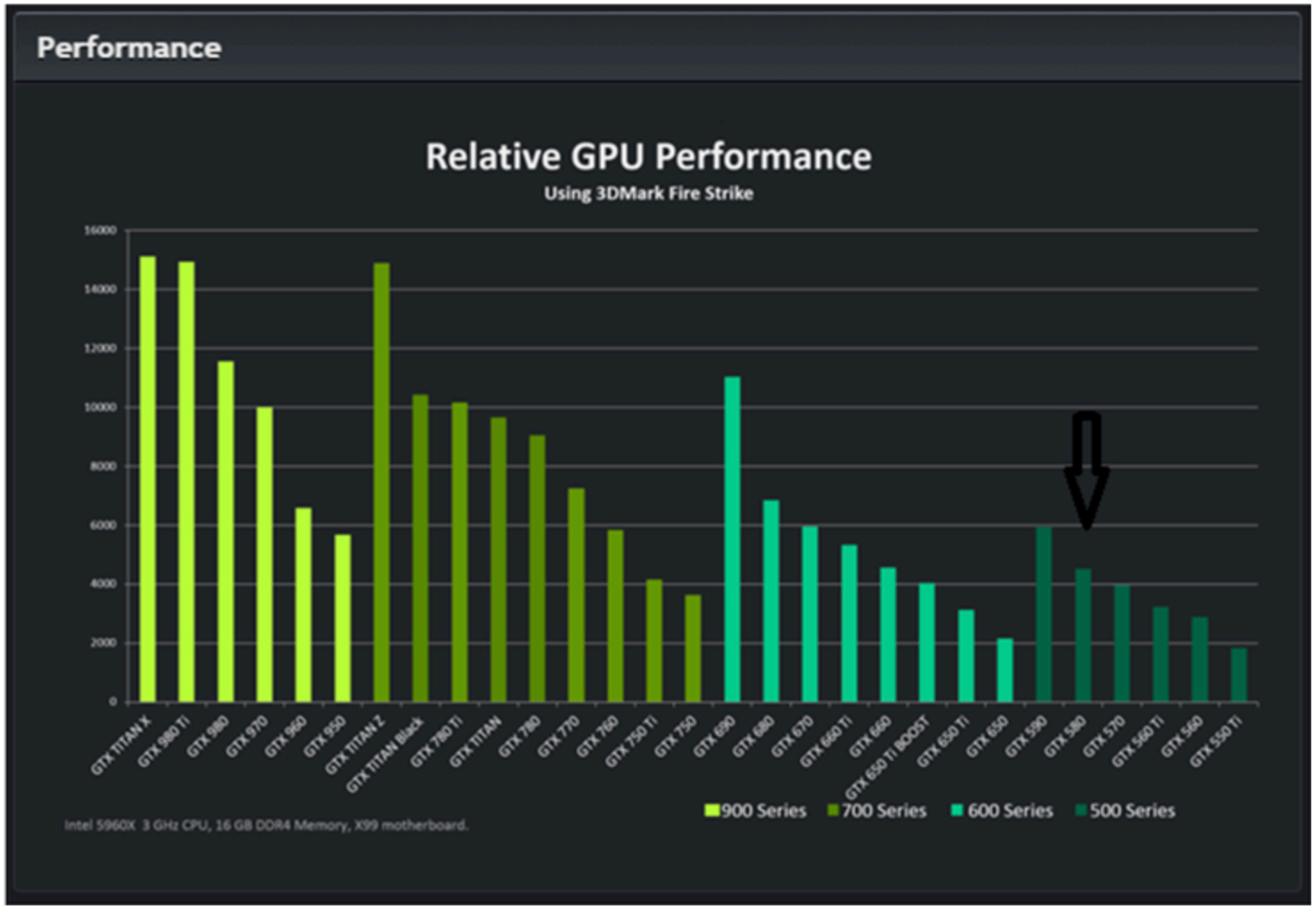
- Rectified Linear Unit
  - Saturating nonlinearity are much slower than non-saturating nonlinearity ReLU



3.1 ReLU Nonlinearity

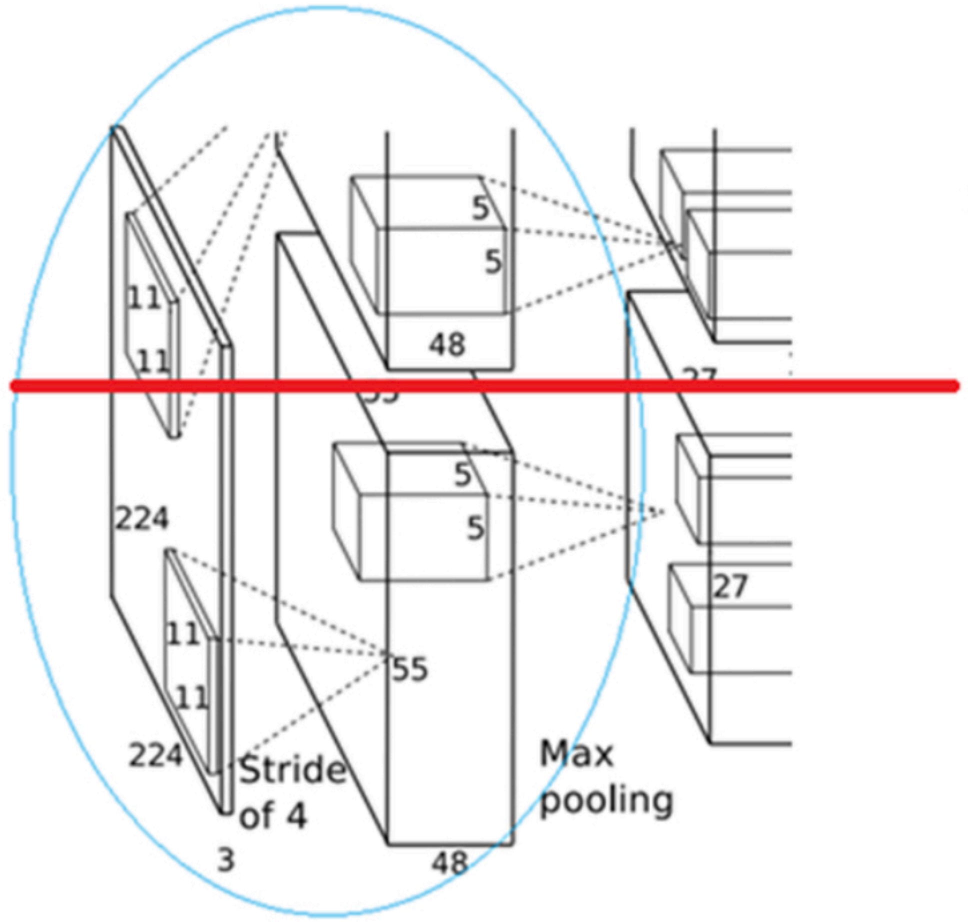
- Others
  - f(x) = |tanh(x)|
  - alternatives to traditional neuron models in CNNs
  - However, on this dataset the primary concern is preventing overfitting
  - So the effect they are observing is different from the accelerated ability to fit the training set

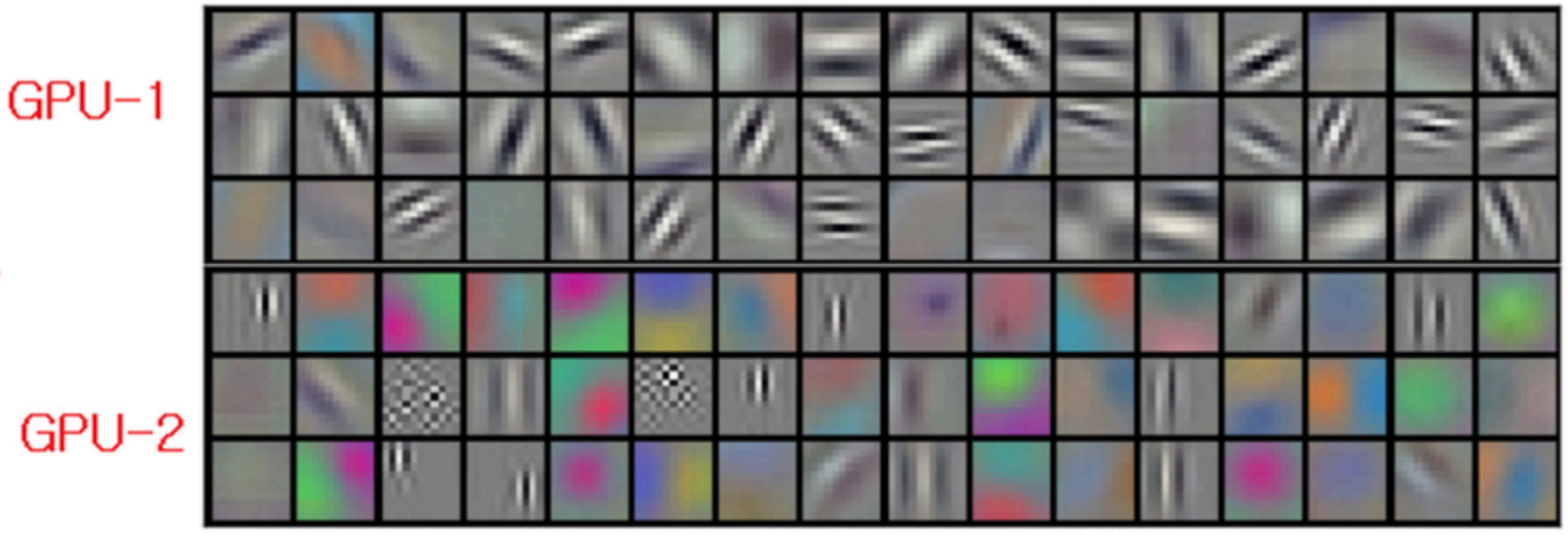
3.2 Training on Multiple GPUs



- The one-GPU net actually has the same number of kernels as the two-GPU net in the final convolutional layer. This is because most of the net's parameters are in the first fully-connected layer, which takes the last convolutional layer as input.

3.2 Training on Multiple GPUs





- GPU1 에서는 주로 색과 관련 없는 정보를 추출하는 kernel 학습
- GPU2 에서는 주로 색과 관련된 정보를 추출하는 kernel 학습

3.3 Local Response Normalization (LRN)

- Sigmoid and Tanh need normalization to avoid saturation
- ReLU doesn't need to do at input.
- But, if normalizing several results from feature maps
- -> 생물학적 뉴런에서의 literal inhibition(강한 자극이 주변의 약한 자극이 전달되는 것을 막는 효과)와 같은 효과를 얻을 수 있기에 generalization 관점에서는 훨씬 좋아짐
- We applied this normalization after applying the ReLU nonlinearity in certain layers

➡ 최근엔 안쓰는 추세

3.4 Overlapping Pooling

- Traditionally, the neighborhoods summarized by adjacent pooling units do not overlap.
- Pooling layers in CNNs summarize the outputs of neighboring groups of neurons in the same kernel map.
- averaging pooling or max pooling
- 최대 크기를 갖는 자극만 전달한다는 것은 생물학적인 특성과 좀 더 유사하다고 볼 수 있다.
- We generally observe during training that models with overlapping pooling find it slightly more difficult to overfit.

4. Reducing Overfitting

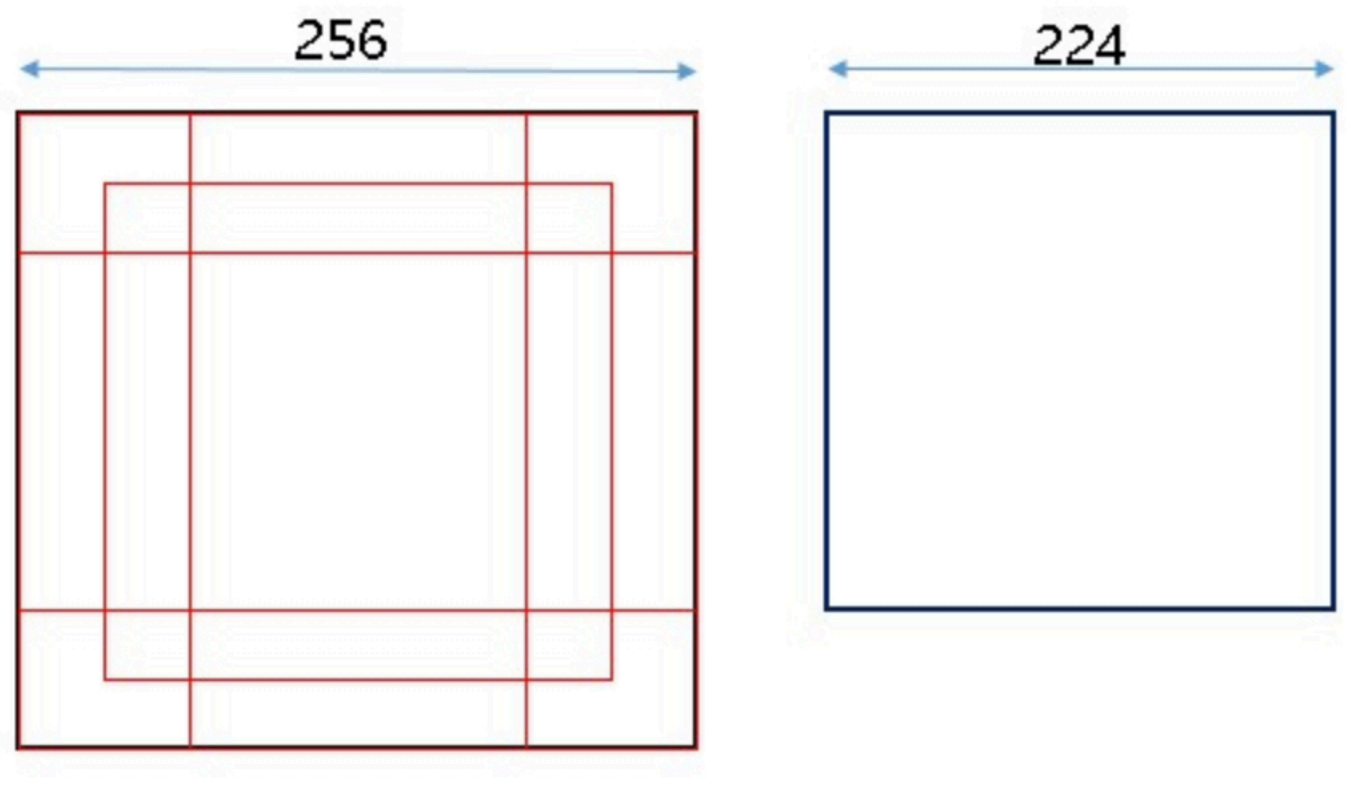
- Data Augmentation
- Dropout

4.1 Data Augmentation

- Label preserving transformations
  - The easiest and most common method to reduce overfitting on image data is to artificially enlarge the dataset.
  - 1) Generating image translations and horizontal reflections
  - 2) Altering the intensities of the RGB channels in training images
  - Transformed images to be produced from the original images with very little computation, so the transformed images do not need to be stored on disk.
  - the transformed images are generated in Python code on the CPU while the GPU is training on the previous batch of images

4.1 Data Augmentation

- Generating image translations and horizontal reflections



- We do this by extracting random 224 × 224 patches (and their horizontal reflections) from the 256×256 images and training our network on these extracted patches.

4.1 Data Augmentation

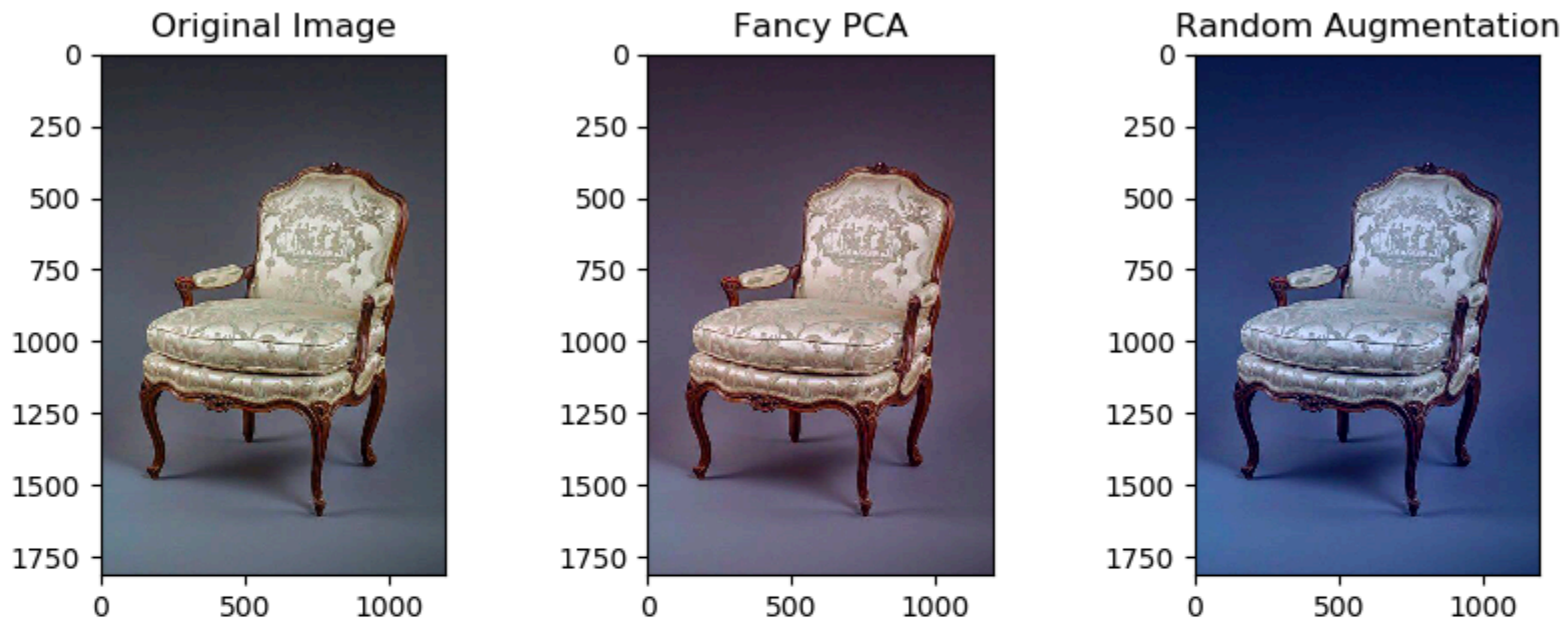
- Generating image translations and horizontal reflections

- We do this by extracting random 224 × 224 patches.
- This increases the size of our training set by a factor of 2048
- So they are highly interdependent.
- At test time, the network makes a prediction by extracting five 224 × 224 patches (the four corner patches and the center patch) as well as their horizontal reflections (hence ten patches in all), and averaging the predictions made by the network’s softmax layer on the ten patches.

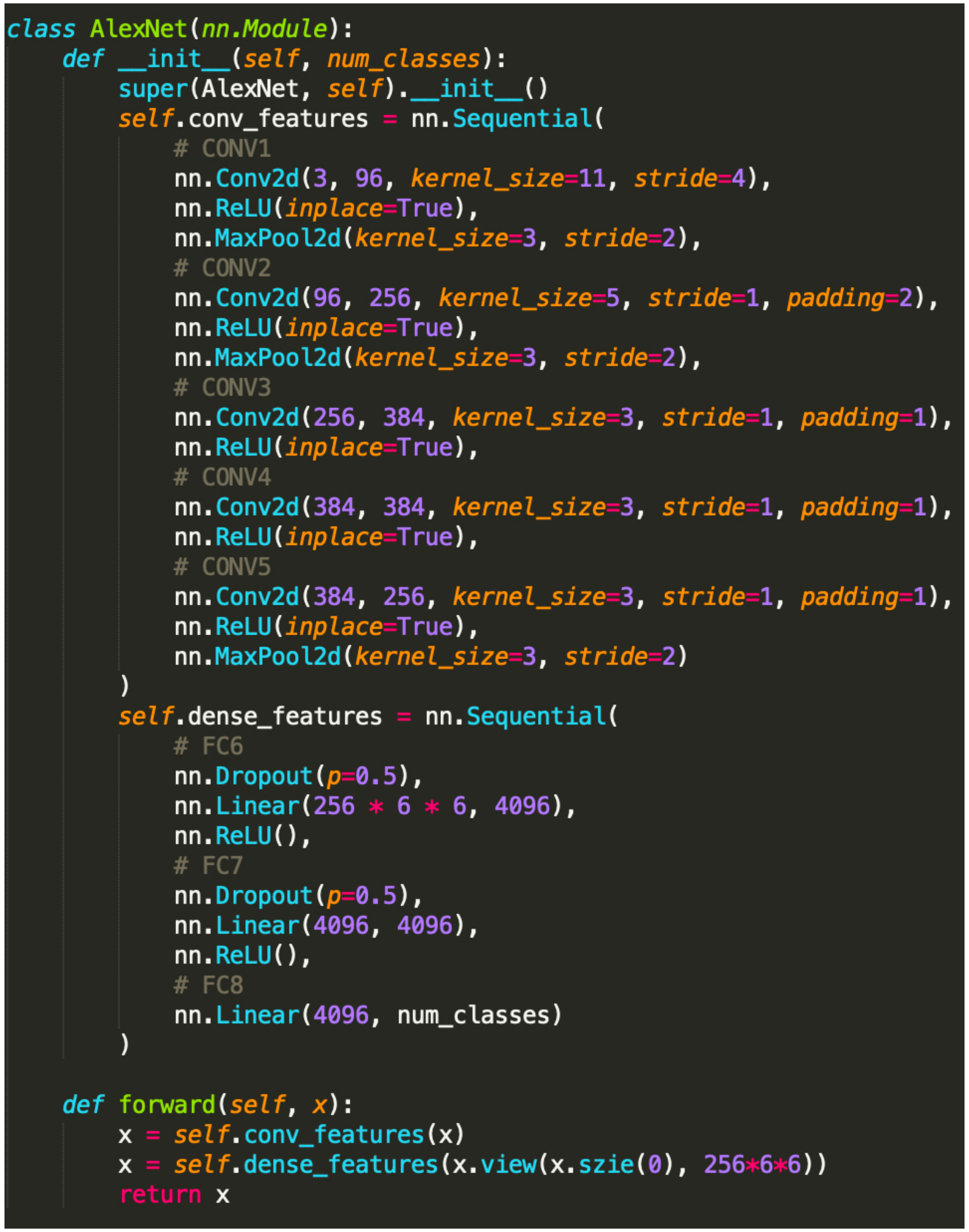
4.1 Data Augmentation

- Altering the intensities of the RGB channels in training images

- Perform PCA on the set of RGB pixel values throughout the ImageNet training set
- Fancy PCA



5. 구현



```python
class AlexNet(nn.Module):
    def __init__(self, num_classes):
        super(AlexNet, self).__init__()
        self.conv_features = nn.Sequential(
            # CONV1
            nn.Conv2d(3, 96, kernel_size=11, stride=4),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            # CONV2
            nn.Conv2d(96, 256, kernel_size=5, stride=1, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            # CONV3
            nn.Conv2d(256, 384, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            # CONV4
            nn.Conv2d(384, 384, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            # CONV5
            nn.Conv2d(384, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2)
        )
        self.dense_features = nn.Sequential(
            # FC6
            nn.Dropout(p=0.5),
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(),
            # FC7
            nn.Dropout(p=0.5),
            nn.Linear(4096, 4096),
            nn.ReLU(),
            # FC8
            nn.Linear(4096, num_classes)
        )

    def forward(self, x):
        x = self.conv_features(x)
        x = self.dense_features(x.view(x.szie(0), 256*6*6))
        return x
```

im2col

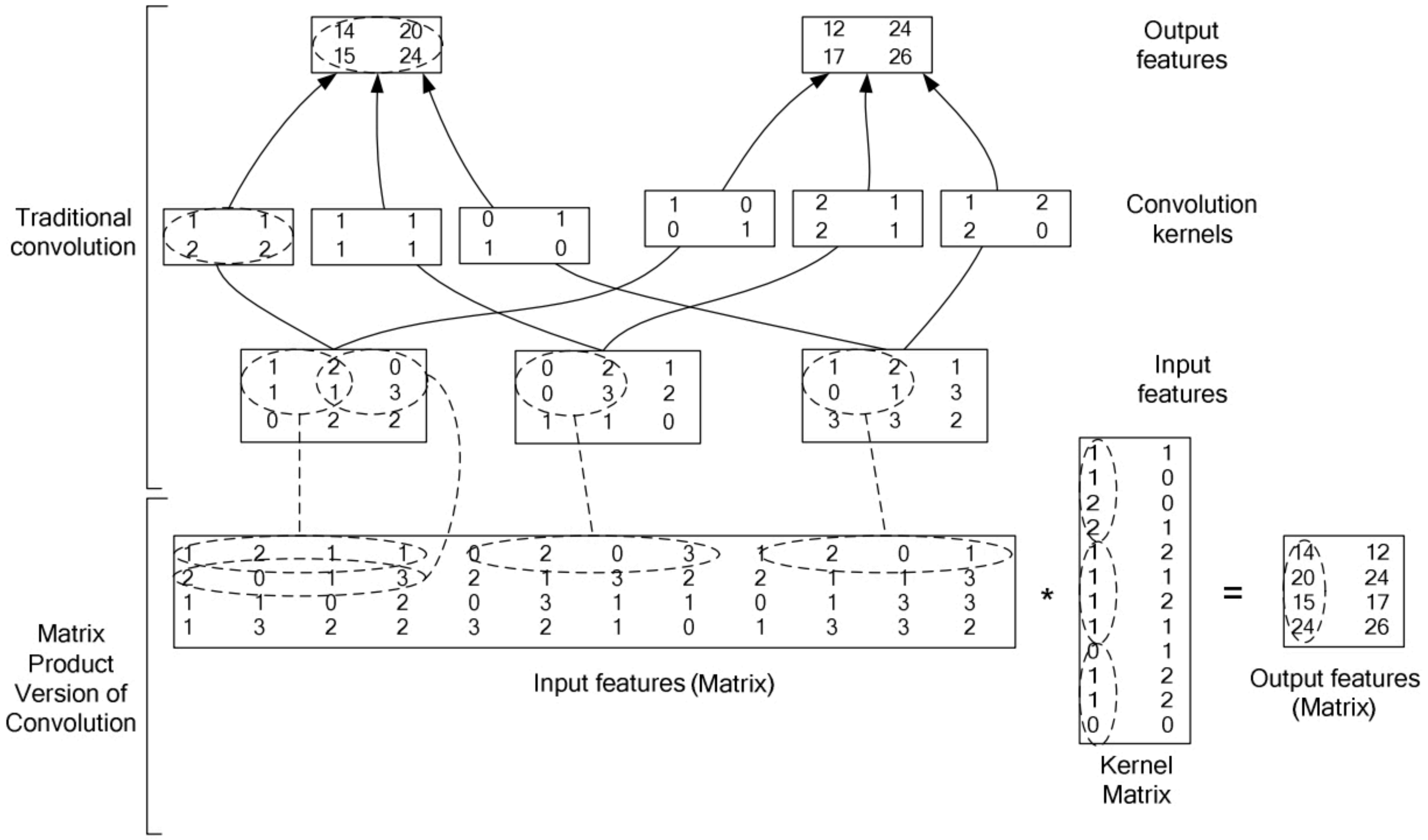
High Performance Convolutional Neural Networks for Document Processing

https://hal.inria.fr/inria-00112631/document

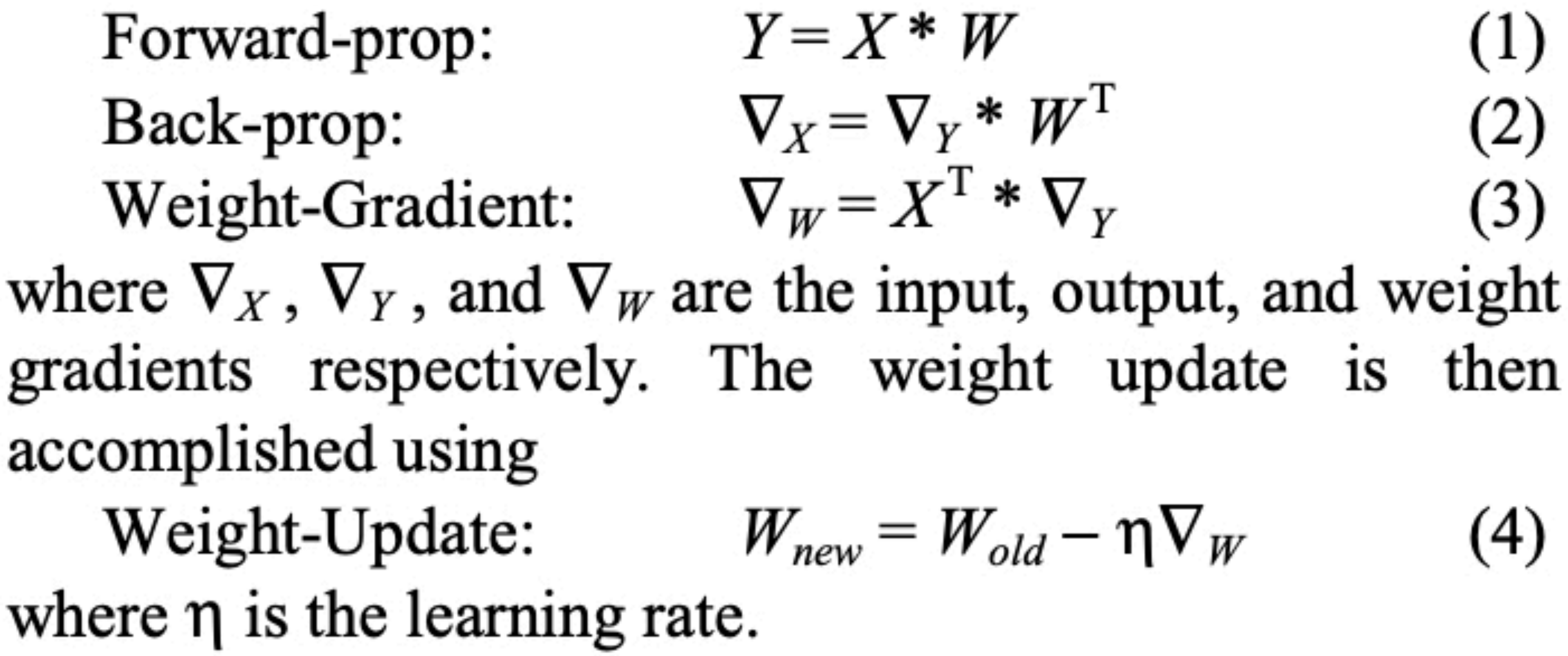
1. Introduction

- However, at runtime the convolution operations are computationally expensive and take up about 67% of the time. This makes typical CNNs about 3X slower than their fully connected equivalents (size-wise).
- The computational complexity of the convolution layers stems from three sources:
- a) the convolution operation
- b) small kernel sizes
- c) cache unfriendly memory access

2. Architecture



3. Computation



Forward-prop: $Y = X * W$ (1)

Back-prop: $\nabla_X = \nabla_Y * W^{\mathrm{T}}$ (2)

Weight-Gradient: $\nabla_W = X^{\mathrm{T}} * \nabla_Y$ (3)

where ∇_X , ∇_Y , and ∇_W are the input, output, and weight gradients respectively. The weight update is then accomplished using

Weight-Update: $W_{new} = W_{old} - \eta\nabla_W$ (4)

where η is the learning rate.