[논문 리뷰] AlexNet (2012)

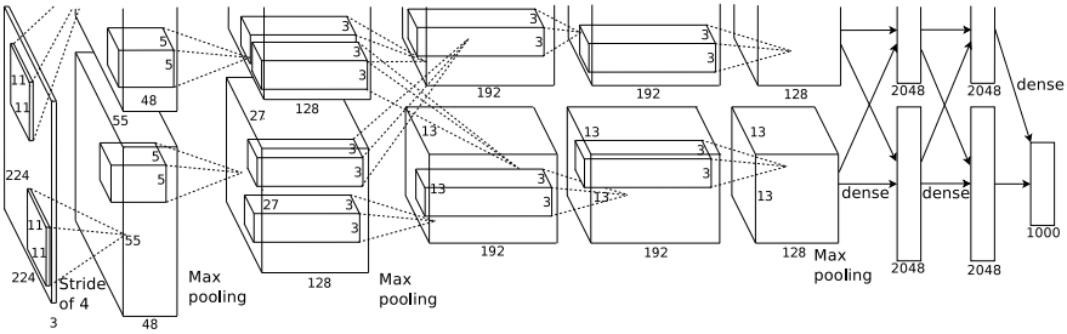
1. Introduction

AlexNet은 training time을 줄일수 있고, 성능을 올리는 방식을 찾았습니다. (Section 3)

AlexNet은 Neural Network가 매우커서, Overfitting 문제가 있었지만, 과적합을 막기 위해effective techniques을 사용하였습니다. (Section 4)

AlexNet은 5개의 Convolutional 과 3개의 FC를 사용함으로서, depth의 중요성을 알게 되었다.

▼ 2. AlexNet Architecture

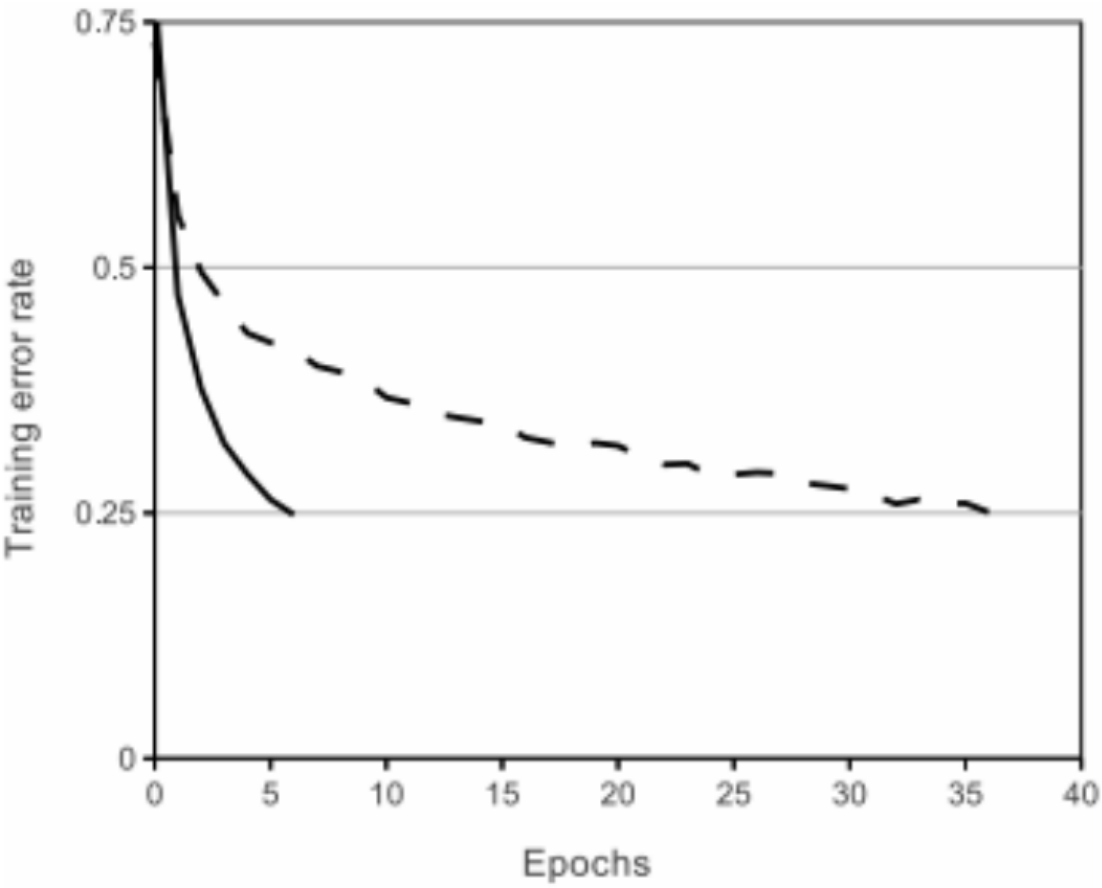


2-1. ReLU Nonlinearity

Model의 일반적인 방식은 $f(x) = tanh$ or $f(x) = (1 + e^{-x})^{-1}$ 이다.
이러한 방식은 ReLU보다 gradient descent면에서 느리다.

DCNN(Deep CNN)은 **ReLU는 tanh보다 몇배는 빠른 training이 가능**합니다.

🌟 6번의 epochs로 ReLU는 25%의 Error에 도달하였습니다.



2-2. Local Response Normalization

ReLU는 Input에 대한 saturating을 막기 위한 Normalization할 필요가 없습니다.

하지만, local normalization scheme가 generalization을 하는 것을 발견하였습니다.

$$b^i_{x,y} = a^i_{x,y} / \left(k + \alpha \sum_{j=\max(0,i-n/2)}^{\min(N-1,i+n/2)} (a^j_{x,y})^2 \right)^\beta$$

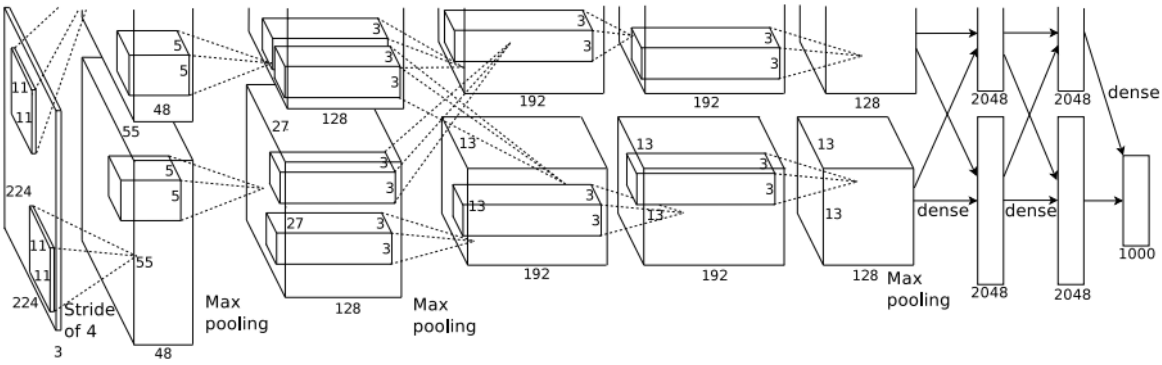
위의 방식은 Jarret의 정규화 방법과 비슷하지만, AlexNet에서는 “brightness Normalzation”이 정확합니다.
Pooling 방식에서 Overlapping을 하용하여, Error를 낮추었습니다.

2-3. Overlapping Pooling

CNN내의 Pooling layer는 같은 Kernel(filter)의 뉴런들을 summarize하여, 출력합니다.

이러한 방식으로 Model을 훈련하는 것이 Overfit하는 것을 방지한다는 것을 알 수 있었습니다.

2-4. Overall Architecture



AlexNet은 총 8개의 layer로 이루어져 있습니다. 5개의 CNN과 3개의 FC로 이루어져 있습니다.

마지막, Layer에서는 1000개의 class를 분류하기 위해, 1000개의 softmax계층으로 이루어져있습니다.

AlexNet의 궁극적인 목적은 다항 로지스틱 회귀(Multinomial logistic regression objective)를 극대화하는 것에 있습니다.

2, 4, 5번째 Convolutional Layer는 이전의 layer에 있는 Kernel map에서만 input을 받아들인다. **(Input부분은 제외하고 시작합니다.)**

3번째 Convolutional Layer에서는 모든 Kernel map들로부터, input을 받아들이고, FC Layer의 각 뉴런들은 이전 레이어의 모든 뉴런들과 연결되어 있습니다.

💫 Structure에 대한 **상세한 설명은 하단 REVIEW**를 참고해주시기 바랍니다.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 96, 55, 55]          34,944
              ReLU-2           [-1, 96, 55, 55]               0
 LocalResponseNorm-3           [-1, 96, 55, 55]               0
         MaxPool2d-4           [-1, 96, 27, 27]               0
            Conv2d-5          [-1, 256, 27, 27]         614,656
              ReLU-6          [-1, 256, 27, 27]               0
 LocalResponseNorm-7          [-1, 256, 27, 27]               0
         MaxPool2d-8          [-1, 256, 13, 13]               0
            Conv2d-9          [-1, 384, 13, 13]         885,120
             ReLU-10          [-1, 384, 13, 13]               0
           Conv2d-11          [-1, 384, 13, 13]       1,327,488
             ReLU-12          [-1, 384, 13, 13]               0
           Conv2d-13          [-1, 256, 13, 13]         884,992
             ReLU-14          [-1, 256, 13, 13]               0
        MaxPool2d-15            [-1, 256, 6, 6]               0
          Dropout-16                 [-1, 9216]               0
           Linear-17                 [-1, 4096]      37,752,832
             ReLU-18                 [-1, 4096]               0
          Dropout-19                 [-1, 4096]               0
           Linear-20                 [-1, 4096]      16,781,312
             ReLU-21                 [-1, 4096]               0
           Linear-22                   [-1, 10]          40,970
================================================================
Total params: 58,322,314
Trainable params: 58,322,314
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.59
Forward/backward pass size (MB): 14.72
Params size (MB): 222.48
Estimated Total Size (MB): 237.79
----------------------------------------------------------------
```

▼ 3. Reducing Overfitting

3-1. Data Augmentation

Image데이터에 관하여 Overfitting을 피하는 가장 흔한 방법입니다.

첫 번째 방법 : 256*256 Image로부터 224*224 patches를 무작위로 추출하는 방식입니다.

TEST시에는 224*224 patch들을 총5개를가져와 예측할 뿐만 아니라, patch들을

horizontal reflection하여 (10개 patches)들을 softmax결과를 평균화하여 predict를 진행합니다.

두번째 방법 : RGB Pixel에 PCA를 적용합니다.

$$[\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3][\alpha_1\lambda_1, \alpha_2\lambda_2, \alpha_3\lambda_3]^T$$

where $\mathbf{p}_i$ and λ_i are ith eigenvector and eigenvalue of the 3×3 covariance matrix of RGB pixel values, respectively, and α_i is the aforementioned random variable. Each α_i is drawn only once for all the pixels of a particular training image until that image is used for training again, at which point it is re-drawn. This scheme approximately captures an important property of natural images, namely, that object identity is invariant to changes in the intensity and color of the illumination. This scheme reduces the top-1 error rate by over 1%.

3-2. Dropout

많은 다양한 모델의 예측 결과를 결합하는 방식으로 test Error를 감소시켜왔지만, 이 방식은 상당히 오랜 시간이 걸립니다.

하지만, training cost가 2배이상이지만, 효율적으로 모델을 결합하는 방식이 존재합니다.

각 은닉층의 뉴런들은 0.5의 확률로 0을 출력하게 합니다.

그러므로, 많은 뉴런들의 결합 집합들로부터 robust feature들을 학습하는데 초점을 둘 수 있습니다.

3-3. Detail of learning

우리는 Batch_size = 128, momentum = 0.9, weight_decay = 5e-4에 SGD를 적용하였습니다.

작은 Weight Decay를 사용하는 것에 모델을 학습하는데 중요하다는 것을 발견하였습니다.

(훈련 에러를 줄여줍니다.)

▼ 4. Results

Model	Top-1	Top-5
Sparse coding [2]	*47.1%*	*28.2%*
SIFT + FVs [24]	*45.7%*	*25.7%*
CNN	**37.5%**	**17.0%**

Model	Top-1 (val)	Top-5 (val)	Top-5 (test)
SIFT + FVs [7]	—	—	*26.2%*
1 CNN	40.7%	18.2%	—
5 CNNs	38.1%	16.4%	**16.4%**
1 CNN*	39.0%	16.6%	—
7 CNNs*	36.7%	15.4%	**15.3%**

Table 2: ILSVRC-2012 대회의 검증, 테스트셋에 대한 결과, 이탤릭체(기울어진 글꼴)은 다른팀의 결과이다. * 표시가 있는 모델은 ImageNet2011의 전체 데이터에 대해 사전학습된 모델이다.

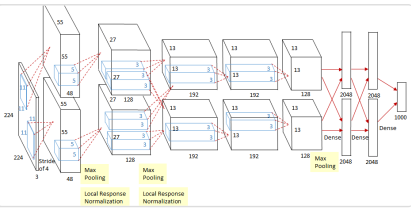
▼ 5. Discussion

AlexNet에서는 순전히 지도학습만으로, 엄청난 성능을 나타냈습니다. 뿐만 아니라, 하나의 Layer만을 제거해도, 성능이 나빠지는 것을 알 수 있었습니다.

Review: AlexNet, CaffeNet - Winner of ILSVRC 2012 (Image Classification)

In this story, AlexNet and CaffeNet are reviewed. AlexNet is the winner of the ILSVRC (ImageNet Large Scale Visual Recognition Competition) 2012, which is an image classification competition. This is a 2012 NIPS paper from Prof. Hinton's Group with

https://medium.com/coinmonks/paper-review-of-alexnet-caffenet-winner-in-ilsvrc-2012-image-classification-b93598314160



ImageNet Classification with Deep Convolutional Neural Networks

ImageNet Classification with Deep Convolutional Neural Networks Part of Advances in Neural Information Processing Systems 25 (NIPS 2012) Authors Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton Abstract We trained a large, deep convolutional neural network to classify the 1.3 million high-resolution images in the LSVRC-2010 ImageNet training set into the 1000 different classes.

https://papers.nips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html

▼ MODEL CODE [Pytorch]

```
import torch.nn as nn
import torch.nn.functional as F

class AlexNet(nn.Module):
  def __init__(self, n_classes):
    super(AlexNet, self).__init__()

    self.feature_extractor = nn.Sequential(
      # First Layer
      nn.Conv2d(in_channels = 3, out_channels = 96, kernel_size = 11, stride = 4),
      nn.ReLU(inplace = True),
      nn.LocalResponseNorm(size = 5, alpha = 1e-3, beta = 0.75, k = 2),
      nn.MaxPool2d(kernel_size = 3, stride = 2),

      # Second Layer
      nn.Conv2d(in_channels = 96, out_channels = 256, kernel_size = 5,stride = 1, padding = 2),
      nn.ReLU(),
      nn.LocalResponseNorm(size = 5, alpha = 1e-4, beta = 0.75, k = 2),
      nn.MaxPool2d(kernel_size = 3, stride = 2),

      # Third Layer
      nn.Conv2d(in_channels = 256, out_channels = 384, kernel_size = 3, stride = 1, padding = 1),
      nn.ReLU(),

      # Fourth Layer
      nn.Conv2d(in_channels = 384, out_channels = 384, kernel_size = 3, stride = 1, padding = 1),
      nn.ReLU(),

      nn.Conv2d(in_channels = 384, out_channels = 256, kernel_size = 3, stride = 1, padding = 1),
      nn.ReLU(),
      nn.MaxPool2d(3, 2),
    )

    self.classifier = nn.Sequential(
        nn.Dropout(0.5),
        nn.Linear(256 * 6 * 6, out_features = 4096),
        nn.ReLU(),
        nn.Dropout(0.5),
        nn.Linear(in_features = 4096, out_features = 4096),
        nn.ReLU(),
        nn.Linear(in_features = 4096, out_features = n_classes),
    )

  def init_weight(self):
    for layer in self.feature_extractor:
      if isinstance(layer, nn.Conv2d):
        nn.init.normal_(layer.weight, mean = 0, std = 0.01)
        nn.init.constant_(layer.bias, 0)
    nn.init.constant_(self.net[4].bias, 1)
    nn.init.constant_(self.net[10].bias, 1)
    nn.init.constant_(self.net[12].bias, 1)
```

```
  def forward(self , x):
    x = self.feature_extractor(x)
    x = x.view(-1, 256*6*6)
    x = self.classifier(x)

    return x
```

▼ Baseline [Pytorch]

```
import numpy as np

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import matplotlib.pyplot as plt

from datetime import datetime
```

```
random_seed = 42
learning_rate = 0.001
batch_size = 64
n_epochs = 15
n_classes = 10
```

```
def get_accuracy(model, data_loader, device):
  correct_pred = 0
  n = 0

  with torch.no_grad():
    model.eval()
    for X, y_true in data_loader:
      X, y_true = X.to(device), y_true.to(device)

      y_pred = model(X)
      _, predicted_labels = torch.max(y_pred, 1)

      n += y_true.size(0)
      correct_pred += (predicted_labels == y_true).sum()

  return correct_pred.float() / n
```

```
def train(train_loader, model, criterion, optimizer, device):
  model.train()
  running_loss = 0
  for X, y_true in train_loader:
    optimizer.zero_grad()

    X, y_true = X.to(device), y_true.to(device)
    y_hat = model(X)
    loss = criterion(y_hat, y_true)
    running_loss += loss.item() * X.size(0)
    loss.backward()
    optimizer.step()

  epoch_loss = running_loss / len(train_loader.dataset)
  return model, optimizer, epoch_loss
```

```
def validation(valid_loader, model, criterion, device):
  model.eval()
  running_loss = 0
  for X, y_true in valid_loader:
    X, y_true = X.to(device), y_true.to(device)

    y_hat= model(X)
    loss = criterion(y_hat, y_true)
    running_loss += loss.item() * X.size(0)

  epoch_loss = running_loss / len(valid_loader.dataset)

  return model, epoch_loss
```

```
def training_loop(model, criterion, optimizer, train_loader, valid_loader, epochs, device, print_every = 1):
  best_loss = 1e10
  train_losses = []
  valid_losses = []

  for epoch in range(epochs):
    model, optimizer, train_loss = train(train_loader, model, criterion, optimizer, device)
    train_losses.append(train_loss)

    with torch.no_grad():
      model, valid_loss = validation(valid_loader, model, criterion, device)
      valid_losses.append(valid_loss)

    if epoch % print_every == (print_every - 1):
      train_acc = get_accuracy(model, train_loader, device)
      valid_acc = get_accuracy(model, valid_loader, device)

      print(f'{datetime.now().time().replace(microsecond=0)} --- '
            f'Epoch: {epoch}\t'
            f'Train loss: {train_loss:.4f}\t'
            f'Valid loss: {valid_loss:.4f}\t'
            f'Train accuracy: {100 * train_acc:.2f}\t'
            f'Valid accuracy: {100 * valid_acc:.2f}')

  return model, optimizer, (train_losses, valid_losses)
```

```
common_transforms = transforms.Compose([
                transforms.ToTensor(),
                transforms.Resize(227)
])

train_dataset = datasets.CIFAR10(root = 'cifar10',
                                 train = True,
                                 transform = common_transforms,
                                 download = True)

test_dataset = datasets.CIFAR10(root = 'cifar10',
                                train = False,
                                transform = common_transforms,
                                download = True)
```

```
import numpy as np

meanRGB = [np.mean(x.numpy(), axis = (1, 2)) for x, _ in train_dataset]
stdRGB = [np.std(x.numpy(), axis = (1, 2)) for x, _ in train_dataset]

meanR = np.mean([m[0] for m in meanRGB])
meanG = np.mean([m[1] for m in meanRGB])
meanB = np.mean([m[2] for m in meanRGB])
```

```
stdR = np.mean([s[0] for s in stdRGB])
stdG = np.mean([s[1] for s in stdRGB])
stdB = np.mean([s[2] for s in stdRGB])

print(meanR, meanG, meanB)
print(stdR, stdG, stdB)
```

```
train_transformer = transforms.Compose([
                transforms.ToTensor(),
                transforms.Resize((227, 227)),
                transforms.RandomHorizontalFlip(),
                transforms.Normalize([0.49139965, 0.48215845, 0.4465309],
                                     [0.20220213, 0.19931543, 0.20086348])
])

test_transformer = transforms.Compose([
                transforms.ToTensor(),
                transforms.Resize((227, 227)),
                transforms.Normalize([0.49139965, 0.48215845, 0.4465309],
                                     [0.20220213, 0.19931543, 0.20086348])
])
```

```
train_dataset.transforms = train_transformer
test_dataset.transforms = test_transformer
```

```
from sklearn.model_selection import StratifiedShuffleSplit

sss = StratifiedShuffleSplit(n_splits = 1, test_size = 0.2, random_state = 0)
indices = list(range(len(test_dataset)))
y_test0 = [y for _, y in test_dataset]

for test_index, val_index in sss.split(indices, y_test0):
    print('test :', len(test_index) , 'val :', len(val_index))

from torch.utils.data import Subset

valid_dataset = Subset(test_dataset, val_index)
test_dataset = Subset(test_dataset, test_index)

train_loader = DataLoader(train_dataset, batch_size = batch_size, shuffle = True)
valid_loader = DataLoader(valid_dataset, batch_size = batch_size, shuffle = False)
test_loader = DataLoader(test_dataset, batch_size = batch_size, shuffle = False)
```

```
import torch.nn as nn
import torch.nn.functional as F

class AlexNet(nn.Module):
  def __init__(self, n_classes):
    super(AlexNet, self).__init__()

    self.feature_extractor = nn.Sequential(
      # First Layer
      nn.Conv2d(in_channels = 3, out_channels = 96, kernel_size = 11, stride = 4),
      nn.ReLU(inplace = True),
      nn.LocalResponseNorm(size = 5, alpha = 1e-3, beta = 0.75, k = 2),
      nn.MaxPool2d(kernel_size = 3, stride = 2),

      # Second Layer
      nn.Conv2d(in_channels = 96, out_channels = 256, kernel_size = 5,stride = 1, padding = 2),
      nn.ReLU(),
      nn.LocalResponseNorm(size = 5, alpha = 1e-4, beta = 0.75, k = 2),
      nn.MaxPool2d(kernel_size = 3, stride = 2),
```

```
      # Third Layer
      nn.Conv2d(in_channels = 256, out_channels = 384, kernel_size = 3, stride = 1, padding = 1),
      nn.ReLU(),

      # Fourth Layer
      nn.Conv2d(in_channels = 384, out_channels = 384, kernel_size = 3, stride = 1, padding = 1),
      nn.ReLU(),

      nn.Conv2d(in_channels = 384, out_channels = 256, kernel_size = 3, stride = 1, padding = 1),
      nn.ReLU(),
      nn.MaxPool2d(3, 2),
    )

    self.classifier = nn.Sequential(
        nn.Dropout(0.5),
        nn.Linear(256 * 6 * 6, out_features = 4096),
        nn.ReLU(),
        nn.Dropout(0.5),
        nn.Linear(in_features = 4096, out_features = 4096),
        nn.ReLU(),
        nn.Linear(in_features = 4096, out_features = n_classes),
    )

  def init_weight(self):
    for layer in self.feature_extractor:
      if isinstance(layer, nn.Conv2d):
        nn.init.normal_(layer.weight, mean = 0, std = 0.01)
        nn.init.constant_(layer.bias, 0)
    nn.init.constant_(self.net[4].bias, 1)
    nn.init.constant_(self.net[10].bias, 1)
    nn.init.constant_(self.net[12].bias, 1)

  def forward(self , x):
    x = self.feature_extractor(x)
    x = x.view(-1, 256*6*6)
    x = self.classifier(x)

    return x
```

```
device = "cuda:0" if torch.cuda.is_available() else "cpu"
```

```
model = AlexNet(10).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr = 1e-3)
criterion = nn.CrossEntropyLoss()

model, optimizer, _ = training_loop(model, criterion, optimizer, train_loader, valid_loader, 15, device)
```