Ch08 딥러닝

딥러닝은 층을 깊게 한 심층 신경망입니다. 이번 장에서는 딥러닝의 특징과 과제, 가능성에 대해 살펴봅니다.

8.1 더 깊게

앞 장에서 많은 것을 배웠습니다. 신경망을 구성하는 다양한 계층(Affine, Conv, Pooling, softmax-loss 등)과 학습에 효과적인 기술(하이퍼파라미터 최적화, 오버피팅을 방지하는 방법 등)을 배웠습니다. 이번 절에서는 그동안 배운 기술을 집약해 MNIST 데이터셋의 손글씨 숫자 인식을 위한 심층 신경망을 만들어봅니다.

8.1.1 더 깊은 신경망으로

이번 절에서 구성할 CNN은 다음과 같습니다.

Conv -> ReLU -> Conv -> ReLU -> Pool -> Conv -> ReLU -> Conv -> ReLU -> Pool -> Conv -> ReLU -> Conv -> ReLU -> Pool -> Affine -> ReLU -> Dropout -> Affine -> Dropout -> Softmax

딱 보아도 이제까지 구현한 신경망보다 층이 깊습니다. 신경망의 특징은 다음과 같습니다.

* 3x3의 작은 필터를 이용한 합성곱 계층
* 활성화 함수는 ReLU
* 완전연결 계층 뒤에 드롭아웃 계층 사용
* Adam을 사용해 최적화
* 가중치 초깃값은 ‘He 초깃값’

이때까지 배운 신경망 기술을 잔뜩 녹여냈습니다! 책에서 제공하는 예제 코드를 참고해 구현해 보겠습니다.

import os, sys  
sys.path.append(os.pardir)  
import pickle  
import numpy as np  
from collections import OrderedDict  
from common.layers import \*  
  
  
class DeepConvNet:  
 *"""  
 네트워크 구성은 다음과 같습니다.  
 conv - relu - conv - relu - pool -  
 conv - relu - conv - relu - pool -  
 conv - relu - conv - relu - pool -  
 affine - relu - dropout - affine - dropout - softmax  
 """* def \_\_init\_\_(self, input\_dim=(1, 28, 28),  
 conv\_param\_1 = {'filter\_num': 16, 'filter\_size':3, 'pad':1, 'stride':1},  
 conv\_param\_2 = {'filter\_num': 16, 'filter\_size':3, 'pad':1, 'stride':1},  
 conv\_param\_3 = {'filter\_num': 32, 'filter\_size':3, 'pad':1, 'stride':1},  
 conv\_param\_4 = {'filter\_num': 32, 'filter\_size':3, 'pad':1, 'stride':1},  
 conv\_param\_5 = {'filter\_num': 64, 'filter\_size':3, 'pad':1, 'stride':1},  
 conv\_param\_6 = {'filter\_num': 64, 'filter\_size':3, 'pad':1, 'stride':1},  
 hidden\_size=50, output\_size=10  
 ):  
 # 가중치 초기화  
 # 각 층의 뉴런 하나당 앞 층의 몇 개 뉴런과 연결되는가  
 pre\_node\_nums = np.array([1\*3\*3, 16\*3\*3, 16\*3\*3, 32\*3\*3, 32\*3\*3, 64\*3\*3, 64\*4\*4, hidden\_size])  
 weight\_init\_scales = np.sqrt(2.0 / pre\_node\_nums) # ReLu를 사용할 대 권장되는 He 초깃값  
  
 self.params={}  
 pre\_channel\_num = input\_dim[0]  
 for idx, conv\_param in enumerate([conv\_param\_1, conv\_param\_2, conv\_param\_3, conv\_param\_4, conv\_param\_5, conv\_param\_6]):  
 self.params['W' + str(idx+1)] = weight\_init\_scales[idx] \* np.random.randn(conv\_param['filter\_num'], pre\_channel\_num, conv\_param['filter\_size'], conv\_param['filter\_size'])  
 self.params['b' + str(idx+1)] = np.zeros(conv\_param['filter\_num'])  
 pre\_channel\_num = conv\_param['filter\_num']  
 self.params['W7'] = weight\_init\_scales[6] \* np.random.randn(64\*4\*4, hidden\_size)  
 self.params['b7'] = np.zeros(hidden\_size)  
 self.params['W8'] = weight\_init\_scales[7] \* np.random.randn(hidden\_size, output\_size)  
 self.params['b8'] = np.zeros(output\_size)  
  
 # 계층 생성  
 self.layers = {}  
 self.layers.append(Convolution(self.params['W1'], self.params['b1'],  
 conv\_param\_1['stride'], conv\_param\_1['pad']))  
 self.layers.append(Relu())  
 self.layers.append(Convolution(self.params['W2'], self.params['b2'],  
 conv\_param\_2['stride'], conv\_param\_2['pad']))  
 self.layers.append(Relu())  
 self.layers.append(Pooling(pool\_h=2, pool\_w=2, stride=2))  
 self.layers.append(Convolution(self.params['W3'], self.params['b3'],  
 conv\_param\_3['stride'], conv\_param\_3['pad']))  
 self.layers.append(Relu())  
 self.layers.append(Convolution(self.params['W4'], self.params['b4'],  
 conv\_param\_4['stride'], conv\_param\_4['pad']))  
 self.layers.append(Relu())  
 self.layers.append(Pooling(pool\_h=2, pool\_w=2, stride=2))  
 self.layers.append(Convolution(self.params['W5'], self.params['b5'],  
 conv\_param\_5['stride'], conv\_param\_5['pad']))  
 self.layers.append(Relu())  
 self.layers.append(Convolution(self.params['W6'], self.params['b6'],  
 conv\_param\_6['stride'], conv\_param\_6['pad']))  
 self.layers.append(Relu())  
 self.layers.append(Pooling(pool\_h=2, pool\_w=2, stride=2))  
 self.layers.append(Affine(self.params['W7'], self.params['b7']))  
 self.layers.append(Relu())  
 self.layers.append(Dropout(0.5))  
 self.layers.append(Affine(self.params['W8'], self.params['b8']))  
 self.layers.append(Dropout(0.5))  
  
 self.last\_layer = SoftmaxWithLoss()  
  
 def predict(self, x, train\_flg=False):  
 for layer in self.layers:  
 if isinstance(layer, Dropout):  
 x = layer.forward(x, train\_flg)  
 else:  
 x = layer.forward(x)  
 return x  
  
 def loss(self, x, t):  
 y = self.predict(x, train\_flg=True)  
 return self.last\_layer.forward(y, t)  
  
 def accuracy(self, x, t, batch\_size=100):  
 if t.ndim != 1 : t = np.argmax(t, axis=1)  
  
 acc = 0.0  
  
 for i in range(int(x.shape[0] / batch\_size)):  
 tx = x[i\*batch\_size:(i+1)\*batch\_size]  
 tt = t[i\*batch\_size:(i+1)\*batch\_size]  
 y = self.predict(tx, train\_flg=False)  
 y = np.argmax(y, axis=1)  
 acc += np.sum(y == tt)  
  
 return acc / x.shape[0]  
  
 def gradient(self, x, t):  
 self.loss(x, t)  
  
 dout = 1  
 dout = self.last\_layer.backward(dout)  
  
 tmp\_layers = self.layers.copy()  
 tmp\_layers.reverse()  
 for layer in tmp\_layers:  
 dout = layer.backward(dout)  
  
 grads ={}  
 for i, layer\_idx in enumerate((0, 2, 5, 7, 10, 12, 15, 18)):  
 grads['W' + str(i+1)] = self.layers[layer\_idx].dW  
 grads['b' + str(i+1)] = self.layers[layer\_idx].db  
  
 return grads  
  
 def save\_params(self, file\_name="params.pkl"):  
 params = {}  
 for key, val in self.params.items():  
 params[key] = val  
 with open(file\_name, 'wb') as f:  
 pickle.dump(params, f)  
  
 def load\_params(self, file\_name="params.pkl"):  
 with open(file\_name, 'rb') as f:  
 params = pickle.load(f)  
 for key, val in params.items():  
 self.params[key] = val  
  
 for i, layer\_idx in enumerate((0, 2, 5, 7, 10, 12, 15, 18)):  
 self.layers[layer\_idx].W = self.params['W' + str(i+1)]  
 self.layers[layer\_idx].b = self.params['b' + str(i+1)]

학습시키는 코드는 다음과 같습니다.

# coding: utf-8  
import sys, os  
sys.path.append(os.pardir) # 부모 디렉터리의 파일을 가져올 수 있도록 설정  
import numpy as np  
import matplotlib.pyplot as plt  
from dataset.mnist import load\_mnist  
from ch08\_trial.deep\_convnet\_trial import DeepConvNet  
from common.trainer import Trainer  
  
(x\_train, t\_train), (x\_test, t\_test) = load\_mnist(flatten=False)  
  
network = DeepConvNet()   
trainer = Trainer(network, x\_train, t\_train, x\_test, t\_test,  
 epochs=20, mini\_batch\_size=100,  
 optimizer='Adam', optimizer\_param={'lr':0.001},  
 evaluate\_sample\_num\_per\_epoch=1000)  
trainer.train()  
  
# 매개변수 보관  
network.save\_params("deep\_convnet\_params.pkl")  
print("Saved Network Parameters!")

신경망의 깊이가 깊은 만큼 학습을 위한 시간이 오래 걸립니다.

8.1.2 정확도를 더 높이려면

MNIST 데이터셋에 대한 기법의 순위를 알 수 있는 <What is the class of the image?>라는 웹 사이트가 있습니다. 이곳에 소개된 신경망들은 그다지 깊지 않지만 99.79%의 정확도를 가지고 있습니다. 이처럼 상위 기법들을 참고하면 정확도를 더 높일 수 있는 기술이나 힌트를 발견할 수 있습니다.

그 중에 데이터 확장(data augmentation)은 손쉬우면서 효과적인 방법입니다. 데이터 확장은 입력 이미지를 알고리즘을 동원해 인위적으로(입력 이미지를 회전하거나 세로로 이동하는 등) 확장하여 데이터 수를 늘립니다. 이미지의 일부를 잘라내는 crop, 좌우를 뒤집는 flip 등도 이용할 수 있습니다.

8.1.3 깊게 하는 이유

‘층을 깊게 하는 것’의 중요성에 대한 이론적인 근거는 아직 많이 부족한 것이 사실입니다. 직관적이기는 하나 이를 뒷받침하는 데이터와 연구에 대해 알아보도록 합시다. 우선 ILSVRC라른 대규모 이미지 인식 대회의 결과에서 층을 깊게 하는 방법을 통해 정확도를 개선하고 있다고 합니다.

또 다른 이점 하나는 신경망의 매개변수 수가 줄어든다는 것입니다. 합성곱의 필터 크기에 주목해 생각해보도록 합시다. 5x5 합성곱 연산은 3x3 합성곱 연산을 2회 하는 것으로 대체할 수 있습니다. 이것에 5x5 필터의 매개변수의 개수는 25개이지만, 3x3 필터의 매개변수의 개수는 18개로 5x5 보다 적습니다.

5x5

3x3

3x3

작은 필터를 겹쳐 신경망을 깊게 하면 넓은 수용 영역(receptive field)을 소화할 수 있습니다. 수용 영역은 뉴런에 변화를 일으키는 국소적인 공간 영역입니다. 그리고 층을 쌓으면서 ReLU 등의 활성화 함수를 끼움으로써 신경망의 표현력이 개선됩니다.

또한 보다 적은 데이터로 학습을 고속으로 수행할 수 있습니다. 합성곱 계층이 깊어지면서 학습하는 정보가 점점 ‘고급’이 됩니다. 층이 얕은 신경망은 이런 특징을 보다 적은 기회를 통해 이해해야 합니다. 이를 위해서는 많은 학습 데이터가 필요하고 결과적으로 학습 시간이 오래 걸립니다.

위의 얘기의 확장으로 계층적으로 정보를 전달한다는 점도 중요합니다. 이를 통해 각 층이 학습해야 할 문제를 ‘풀기 쉬운 단순한 문제’로 분해할 수 있어 효율적으로 학습하리라 기대할 수 있습니다.

층의 심화는 GPU와 빅데이터의 발전이 뒷받침을 해주어 나타날 수 있었던 현상입니다.

8.2 딥러닝의 초기 역사

딥러닝이 주목받기 시작한 건 2012년 ILSVRC(ImageNet Large Scale Visual Recognition Challenge)에서 AlexNet이 압도적인 성적으로 우승하면서 시작됐습니다. ILSVRC 대회를 축으로 딥러닝 트렌드를 살펴보겠습니다.

8.2.1 이미지넷

이미지넷(ImageNet)은 100만 장이 넘는 이미지와 이미지에 따른 레이블을 담고 있는 데이터셋입니다. ILSVRC 대회에는 시험 항목이 몇 가지 있는데, 그중 하나가 분류(classification)입니다. 분류 부분에서 우승팀의 톱-5 오류(top-5 error, 확률이 가장 높다고 예측한 후보 클래스 5개가 모두 틀린 경우)을 살펴보도록 합시다.

2010년 NEC America : 28.2%

2011년 Xerox : 25.8%

2012년 AlexNet : 16.4%

2013년 Clarifi : 11.7%

2014년 VGG : 7.3%

2014년 GoogLeNet : 6.7%

2015년 ResNet : 3.5%

2012년 AlexNet이 딥러닝을 통해 오류율을 크게 낮춘 후로 선두는 항상 딥러닝 방식입니다. 2015년 ResNet의 3.5% 오류율은 일반적인 인간의 인식 능력을 넘어섰다고 합니다. 이 중에서 VGG, GoogLeNet, ResNet은 다양한 딥러닝 분야에서 활용됩니다.

8.2.2 VGG

VGG는 합성곱 계층과 풀링 계층으로 구성되는 ‘기본적인’ CNN입니다. 합성곱 계층, 완전연결 계층을 모두 16(또는 19)층으로 심화한 것과 3x3의 작은 필터를 사용한 합성곱 계층을 연속으로 거친다는 것이 특징입니다(층의 개수에 따라 VGG16 또는 VGG19로 불립니다).

비록 GoogLeNet에 성능은 떨어지지만, 구성이 간단하여 응용하기 편해 많은 기술자가 VGG 기반의 신경망을 사용합니다.

8.2.3 GoogLeNet

GoogLeNet의 구성은 복잡합니다. 그래도 기본적으로는 지금까지 보아온 CNN과 다르지 않습니다. 다른 점은 세로 방향 깊이 뿐 아니라 가로 방향도 깊다는 점입니다. GoogLeNet은 가로 방향에 ‘폭’이 있는 인셉션 구조를 가지고 있으며 기반 구조는 다음과 같습니다.

필터 결합

앞 계층

5x5 합성곱

5x5 최대 풀링

3x3 합성곱

1x1 합성곱

인셉션 구조는 위 그림처럼 크기가 다른 필터, 풀링을 여러 개 적용하여 그 결과를 결합합니다. 또 GoogLeNet에서는 매개변수 제거, 고속 처리를 하기 위해 1x1 크기의 필터를 사용한 합성곱을 해 채널 쪽의 크기를 줄입니다.

8.2.4 ResNet

ResNet(Residual Network)은 마이크로소프트의 팀이 개발한 네트워크로 층을 지금보다 더 깊게 할 수 있는 특별한 ‘장치’가 특징입니다. 층을 깊게 하는 것이 네트워크의 성능 향상에 중요하다는 것을 알고 있습니다. 하지만 층이 지나치게 깊을 경우 학습이 잘 되지 않고, 성능이 떨어지는 경우도 있습니다. ResNet은 그런 문제를 해결하기 위해 스킵 연결(skip connection)을 도입합니다. 스킵 연결은 다음 그림처럼 입력 데이터를 합성곱 계층을 건너뛰어 출력에 바로 더합니다.

F(X)

X

F(X) + X

X

identity

relu

X

Weight layer

Weight layer

스킵 연결은 입력 데이터를 ‘그대로’ 흘리기 때문에 기울기가 작아지거나 지나치게 커질 걱정 없이 ‘의미 있는 기울기’가 전해지리라 기대할 수 있습니다. 층을 깊게 할수록 기울기가 작아지는 소실 문제를 이 스킵 연결이 줄여주는 것입니다.

ResNet은 VGG 신경망을 기반으로 스킵 연결을 도입해 층을 깊게 했습니다. 실험을 통해 150층 이상으로 해도 정확도가 오르는 모습을 확인할 수 있었고 대회에서 오류율이 겨우 3.5%라는 경이적인 결과를 냈습니다.

이미지넷이 제공하는 데이터셋으로 학습한 가중치 값들은 실제 제품에 활용해도 효과적입니다. 이를 전이 학습(transfer learning)이라 하며, 학습된 가중치를 다른 신경망에 복사한 다음, 그 상태로 재학습을 수행합니다. 데이터셋이 적을 때 유용한 방법입니다.

8.3 더 빠르게(딥러닝 고속화)

딥러닝은 대량의 연산을 수행해야 하는데 CPU만으로는 처리하기 힘듭니다. 이를 GPU를 활용해 대량의 연산을 고속으로 처리할 수 있습니다. 이번 절에서는 딥러닝의 고속화에 관해 이야기해보도록 합시다.

8.3.1 풀어야 할 숙제

딥러닝은 오랜 시간을 합성곱 계층에서 소요합니다. 그래서 합성곱 계층(단순 곱셈-누산)에서 이뤄지는 연산을 어떻게 빠르게 효율적으로 하느냐가 딥러닝의 과제입니다.

8.3.2 GPU를 활용한 고속화

GPU는 원래 그래픽 전용이었습니다. 그러나 최근에는 그래픽 처리 뿐 아니라 범용 수치 연산에 이용합니다. GPU로 범용 수치 연산을 수행하는 것을 GPU 컴퓨팅이라 합니다. AlexNet의 학습시간을 CPU와 GPU에서 비교한 결과를 보면, CPU는 40여 일이 걸리지만 GPU로는 6일까지 단축된다고 합니다.

8.3.3 분산 학습

심층 신경망에서는 학습에 며칠 혹은 몇 주가 걸리기도 하는데, 시행착오를 동반하기 때문에 1회 학습에 걸리는 수간을 최대한 단축하면 좋습니다. 그래서 딥러닝 학습을 수평 확장(scale out)하자는 ‘분산 학습’이 중요해지는 것입니다.

실제로 GPU 수에 따른 학습 시간을 비교해보면, 100개까지 사용한 경우가 하나를 사용한 경우보다 56배 빨라졌다고 합니다. 이는 7일짜리 작업이 3시간 만에 끝난다는 것으로 놀라운 효과를 본 것입니다.

8.3.4 연산 정밀도와 비트 줄이기

계산 속도 외에도 메모리 용량과 버스 대역폭 등은 문제가 될 수 있습니다. 메모리 측면에서는 대량의 가중치 매개변수와 중간 데이터 저장이, 버스 대역폭 면에서는 데이터가 한계를 넘어설 때 일어나는 병목현상이 문제가 됩니다.

컴퓨터에서는 주로 64비트나 32비트 부동소수점 수를 사용해 실수를 표현합니다. 많은 비트를 사용할 수록 계산 오차는 줄어들지만, 메모리와 계산 비용이 늘어납니다. 다행히 딥러닝은 높은 수치 정밀도를 요구하지 않기 때문에 16비트 반정밀도(half-precision)정도만 사용해도 학습에 문제가 없다고 합니다.

딥러닝의 비트 수를 줄이기 위해 가중치와 매개변수를 1비트로 표현하는 <Binary Neural Network>라는 방법도 등장했습니다. 딥러닝을 고속화 하기 위해 비트를 줄이는 기술은 앞으로 주시해야 하고, 임베디드를 위해서 아주 중요한 주제입니다.

8.4 딥러닝의 활용

딥러닝이 할 수 있는 것들을 비전 분야를 중심으로 몇 가지 소개하겠습니다.

8.4.1 사물 검출

사물 검출은 이미지 속에 담긴 사물의 위치와 종류를 알아내는 기술입니다. 이런 사물 검출 문제에 CNN을 기반으로 한 기법이 몇 가지 제안되었는데, 이 중에서도 R-CNN(Regions with Convolutional Neural Network)이 유명합니다.

R-CNN은 ‘후보 영역 추출’과 ‘CNN 특징 계산’이라는 흐름을 지나게 됩니다. 사물의 위치한 영역을 찾아내고, 추출한 영역에 CNN을 적용하여 클래스를 분류하는 방법입니다. 후보 영역 추출을 위해 컴퓨터 비전 분야에서 발전해온 다양한 기법을 사용할 수 있지만, 최근에는 후보 영역 추출까지 CNN으로 처리하는 Faster R-CNN 기법이 등장했습니다.

8.4.2 분할

분할(segmentation)은 이미지를 픽셀 수준에서 분류하는 문제입니다. 신경망을 이용해 분할하는 가장 단순한 방법은 모든 픽셀을 각각 추론하는 것입니다. 하지만, 너무나 긴 시간이 걸리게 되는게 문제입니다. 이런 문제를 완화시키는 기법으로 FCN(Fully Convolutional Network)이 고안되었는데, 이는 단 한 번의 forward 처리로 모든 픽셀의 클래스를 분류해주는 기법입니다.