1. 신경망 복습

1.4 신경망으로 문제를 풀다

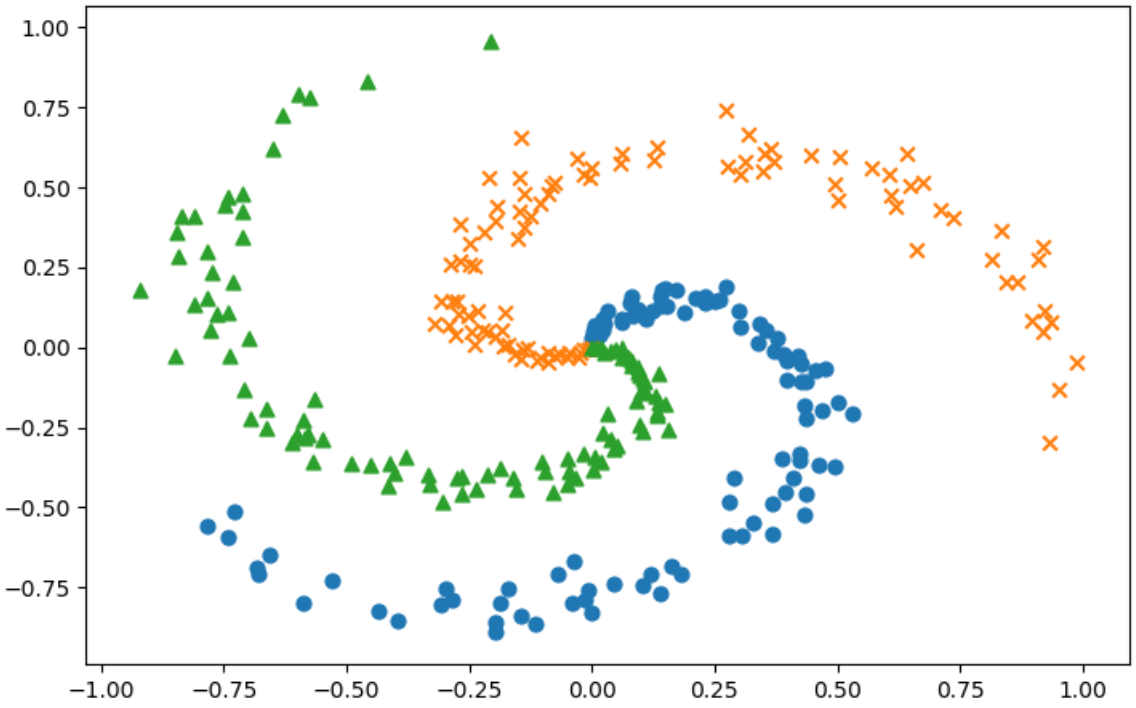
1.4.1 스파이럴 데이터셋

이 책에서는 데이터셋을 다루는 편의 클래스 몇 개를 dataset 디렉터리에 준비해뒀다. 이 번절에서는 그 중 dataset/spiral.py 파일을 이용한다. 이 파일에는 '스파이럴 데이터'를 읽어 들이는 클래스가 구현되어 있으며, 다음과 같이 사용한다.

```
# coding: utf-8
import sys
sys.path.append('..')  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
from dataset import spiral
import matplotlib.pyplot as plt


x, t = spiral.load_data()
print('x', x.shape)  # (300, 2)
print('t', t.shape)  # (300, 3)

# 데이터점 플롯
N = 100
CLS_NUM = 3
markers = ['o', 'x', '^']
for i in range(CLS_NUM):
    plt.scatter(x[i*N:(i+1)*N, 0], x[i*N:(i+1)*N, 1], s=40, marker=markers[i])
plt.show()
```

```
# coding: utf-8
import numpy as np

def load_data(seed=1984):
    np.random.seed(seed)
    N = 100  # 클래스당 샘플 수
    DIM = 2  # 데이터 요소 수
    CLS_NUM = 3  # 클래스 수

    x = np.zeros((N*CLS_NUM, DIM)) # (300, 2)
    t = np.zeros((N*CLS_NUM, CLS_NUM), dtype=np.int) # (300, 3)

    for j in range(CLS_NUM):
        for i in range(N): # N*j, N*(j+1)):
            rate = i / N
            radius = 1.0*rate
            theta = j*4.0 + 4.0*rate + np.random.randn()*0.2

            ix = N*j + i
            x[ix] = np.array([radius*np.sin(theta),
                              radius*np.cos(theta)]).flatten()
            t[ix, j] = 1

    return x, t
```

x가 입력 데이터이고, t가 정답 레이블이다. x와 t의 형상을 출력해보면 각각 300개의 샘플 데이터를 담고 있으며, x는 2차 데이터이고 t는 3차원 데이터임을 알 수 있다. 참고로 t는 원핫 벡터로, 정답에 해당하는 클래스에는 1이, 그 외에는 0이 레이블 되어 있다.



위 그림 처럼 입력은 2차원 데이터이고, 분류할 클래스 수는 3개이다. 이 그래프를 보면 직선만으로는 클래스들을 분리할 수 없음을 알 수 있다. 따라서 비선형 분리를 학습해야 한다. (비선형인 시그모이드 함수를 활성화 함수로 사용하는 은닉층) 우리의 신경망은 이 비선형 패턴 올바르게 학습할 수 있는지 확인해야 한다. 해보자.

1.4.2 신경망 구현

이번 절에서는 은닉층이 하나인 신경망을 구현한다. 임포트 문과 초기화를 담당하는 __init__ 메서드부터 보겠다.

```
# coding: utf-8
import sys
sys.path.append('..')  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
import numpy as np
from common.layers import Affine, Sigmoid, SoftmaxWithLoss


class TwoLayerNet:
    def __init__(self, input_size, hidden_size, output_size):
        I, H, O = input_size, hidden_size, output_size

        # 가중치와 편향 초기화
        W1 = 0.01 * np.random.randn(I, H)
        b1 = np.zeros(H)
        W2 = 0.01 * np.random.randn(H, O)
        b2 = np.zeros(O)

        # 계층 생성
        self.layers = [
            Affine(W1, b1),
            Sigmoid(),
            Affine(W2, b2)
```

```
        ]
        self.loss_layer = SoftmaxWithLoss()

        # 모든 가중치와 기울기를 리스트에 모은다.
        self.params, self.grads = [], []
        for layer in self.layers:
            self.params += layer.params
            self.grads += layer.grads
```

초기화 메서드는 3개의 인수를 받는데 차례대로 input_size는 입력층의 뉴런 수, hidden_size는 은닉층의 뉴런 수 ,output_size는 출력층의 뉴런 수이다. 메서드 안에서는 우선 편향을 영벡터로 초기화하고 (np.zeros), 가중치는 작은 무작위 값으로 초기화한다. 가중치는 작은 무작위 값으로 초기화 한다. (0.01 * np.random.randn()). 참고로 가중치를 작은 무작위 값으로 설정하면 학습이 잘 진행될 가능성이 커진다. 계속해서 필요한 계층을 생성해 인스턴스 변수인 layers 리스트에 모아두고, 마지막으로 이 모델에서 사용하는 매개변수들과 기울기들을 각각 하나로 모은다.

WARNING_ Softmax with Loss 계층은 다른 계층과 다르게 취급하여, layers 리스트가 아닌 loss_layer 인스턴스 변수에 별도로 저장한다.

이어서 TwoLayerNet에 3개의 메서드를 구현해 넣는다. 추론을 수행하는 predict() 메서드, 순전파를 담당하는 forward() 메서드, 역전파를 담당하는 backward() 메서드이다.

```
    def predict(self, x):
        for layer in self.layers:
            x = layer.forward(x)
        return x

    def forward(self, x, t):
        score = self.predict(x)
        loss = self.loss_layer.forward(score, t)
        return loss

    def backward(self, dout=1):
        dout = self.loss_layer.backward(dout)
        for layer in reversed(self.layers):
            dout = layer.backward(dout)
        return dout
```

보다시피 이번 구현은 이전보다 깔끔하다. 신경망에서 사용하는 처리 블록들을 '계층' 단위로 미리 구현해놨으므로, 여기에서는 그 계층들의 forward()와 backward()를 적절한 순서로 호출만 하면 된다.

1.4.3 학습용 코드

여기서는 학습 데이터를 읽어들여 신경망 (모델)과 옵티마이저 (최적화기) 를 생성한다. 그리고 앞 절에서 본 학습의 네 단계의 절차대로 학습을 수행한다. 참고로 머신러닝 분야에서는 문제를 풀기 위해서 설계한 기법을 가리켜 보통 '모델' 이라고 부른다. 학습용 코드는 다음과 같다.

(ch01/train_custom_loop.py)

```
# coding: utf-8
import sys
sys.path.append('..')  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
import numpy as np
from common.optimizer import SGD
from dataset import spiral
import matplotlib.pyplot as plt
```

```
from two_layer_net import TwoLayerNet

#1. 하이퍼파라미터 설정
max_epoch = 300
batch_size = 30
hidden_size = 10
learning_rate = 1.0

#2. 데이터 읽기, 모델과 옵티마이저 생성
x, t = spiral.load_data()
model = TwoLayerNet(input_size=2, hidden_size=hidden_size, output_size=3)
optimizer = SGD(lr=learning_rate)

# 학습에 사용하는 변수
data_size = len(x)
max_iters = data_size // batch_size
total_loss = 0
loss_count = 0
loss_list = []

for epoch in range(max_epoch):
    #3. 데이터 뒤섞기
    idx = np.random.permutation(data_size)
    x = x[idx]
    t = t[idx]

    for iters in range(max_iters):
        batch_x = x[iters*batch_size:(iters+1)*batch_size]
        batch_t = t[iters*batch_size:(iters+1)*batch_size]

        #4. 기울기를 구해 매개변수 갱신
        loss = model.forward(batch_x, batch_t)
        model.backward()
        optimizer.update(model.params, model.grads)

        total_loss += loss
        loss_count += 1

        #5. 정기적으로 학습 경과 출력
        if (iters+1) % 10 == 0:
            avg_loss = total_loss / loss_count
            print('| 에폭 %d |  반복 %d / %d | 손실 %.2f'
                  % (epoch + 1, iters + 1, max_iters, avg_loss))
            loss_list.append(avg_loss)
            total_loss, loss_count = 0, 0


# 학습 결과 플롯
plt.plot(np.arange(len(loss_list)), loss_list, label='train')
plt.xlabel('반복 (x10)')
plt.ylabel('손실')
plt.show()

# 경계 영역 플롯
h = 0.001
x_min, x_max = x[:, 0].min() - .1, x[:, 0].max() + .1
y_min, y_max = x[:, 1].min() - .1, x[:, 1].max() + .1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
```

```
X = np.c_[xx.ravel(), yy.ravel()]
score = model.predict(X)
predict_cls = np.argmax(score, axis=1)
Z = predict_cls.reshape(xx.shape)
plt.contourf(xx, yy, Z)
plt.axis('off')

# 데이터점 플롯
x, t = spiral.load_data()
N = 100
CLS_NUM = 3
markers = ['o', 'x', '^']
for i in range(CLS_NUM):
    plt.scatter(x[i*N:(i+1)*N, 0], x[i*N:(i+1)*N, 1], s=40, marker=markers[i])
plt.show()
```

#1. 하이퍼파라미터 설정

우선 하이퍼파라미터를 설정한다. 구체적으로는 학습하는 에폭 수, 미니배치 크기, 은닉층의 뉴런 수, 학습률을 설정한다.

#2. 데이터 읽기, 모델과 옵티마이저 생성

계속해서 데이터를 읽어 들이고, 신경망(모델)과 옵티마이저를 생성한다. 우리는 이미 2층 신경망을 TwoLaterNet 크래스로, 또 옵티마이저를 SGD 클래스로 구현해놨으니, 여기에서는 이 클래스들을 이용하겠다.

Note_ 에폭은 학습 단위이다. 1에폭은 학습 데이터를 모두 '살펴본' 시점(데이터셋을 1바퀴 돌아본 시점)을 뜻한다. 이 코드에서는 300에폭을 학습한다.

학습은 미니배치 방식으로 진행되며 데이터를 무작위로 선택한다.

#3. 데이터 뒤섞기

여기에서는 에폭 단위로 데이터를 뒤섞고, 뒤섞은 데이터 중 앞에서부터 순서대로 뽑아내는 방식을 사용했다. 데이터 뒤섞기(정확하게는 데이터의 '인덱스'뒤섞기)에는 np.random.permutation() 메서드를 사용한다. 이 메서드에 인수로 N을 주면, 0에서 N-1까지의 무작위 순서를 생성해 반환한다. 실제 사용 예는 다음과 같다.

```
In [7]: import numpy as np

In [8]: np.random.permutation(10)
Out[8]: array([3, 5, 0, 9, 7, 6, 1, 2, 4, 8])
```

이처럼 np.random.permutation()을 호출하면 데이터 인덱스를 무작위로 뒤섞을 수 있다.

#4. 기울기를 구해 매개변수 갱신

계속해서 기울기를 구해 매개변수를 갱신한다.

```
    #4. 기울기를 구해 매개변수 갱신
        loss = model.forward(batch_x, batch_t)
        model.backward()
        optimizer.update(model.params, model.grads)

        total_loss += loss
        loss_count += 1
```

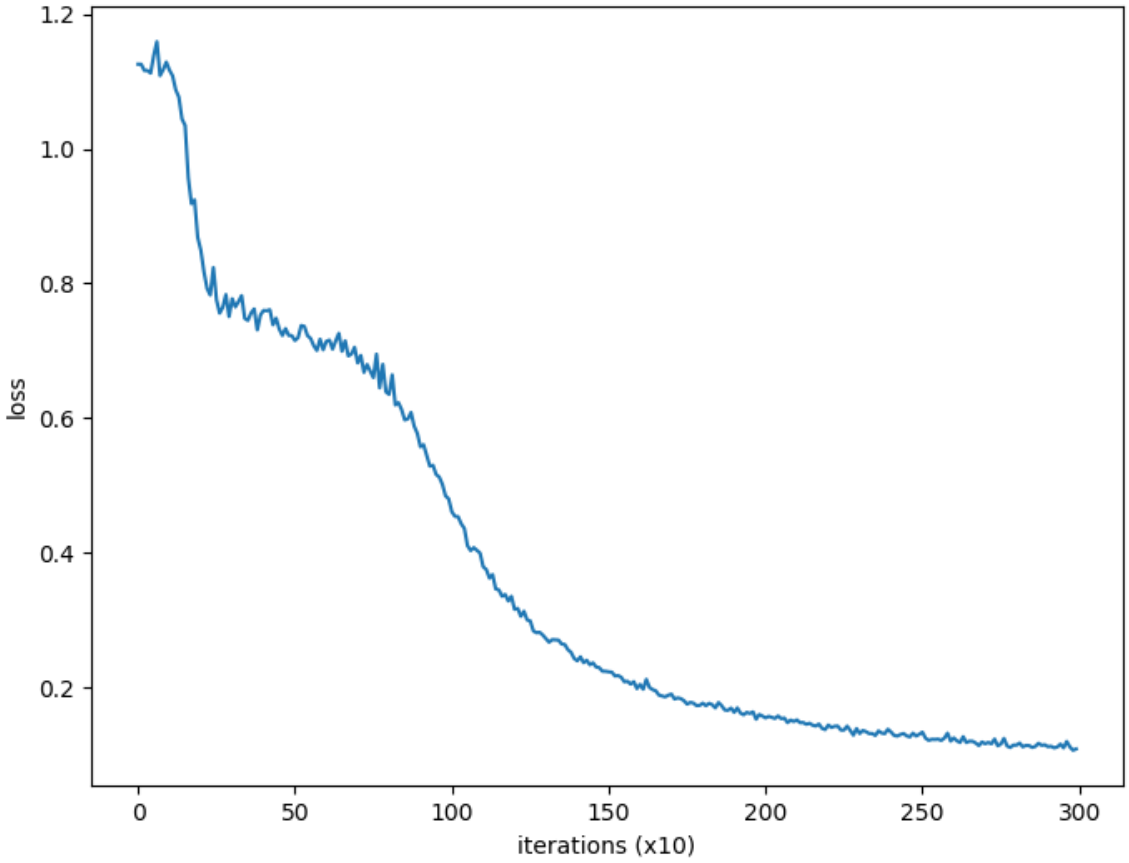
#5. 정기적으로 학습 경과 출력

```
        #5. 정기적으로 학습 경과 출력
        if (iters+1) % 10 == 0:
            avg_loss = total_loss / loss_count
            print('| 에폭 %d |  반복 %d / %d | 손실 %.2f'
                  % (epoch + 1, iters + 1, max_iters, avg_loss))
            loss_list.append(avg_loss)
            total_loss, loss_count = 0, 0
```

마지막으로 정기적으로 학습 결과를 출력한다. 이 코드에서는 10번째 반복마다 손실의 평균을 구해 list_loss 변수에 추가했다. 이상으로 학습을 수행하는 코드를 살펴봤다.

WARNING_ 여기서 구현한 신경망의 학습 코드는 이 책의 다른 장소에서도 사용가능하다. 이에 이 코드를 Trainer 클래스로 만들어뒀다. 신경망 학습의 상세 내용을 이 클래스 안으로 밀어넣은 것이다.

이제 이 코드(ch01/train_custom_loop.py)를 실행해보자. 결과는 다음과 같다.



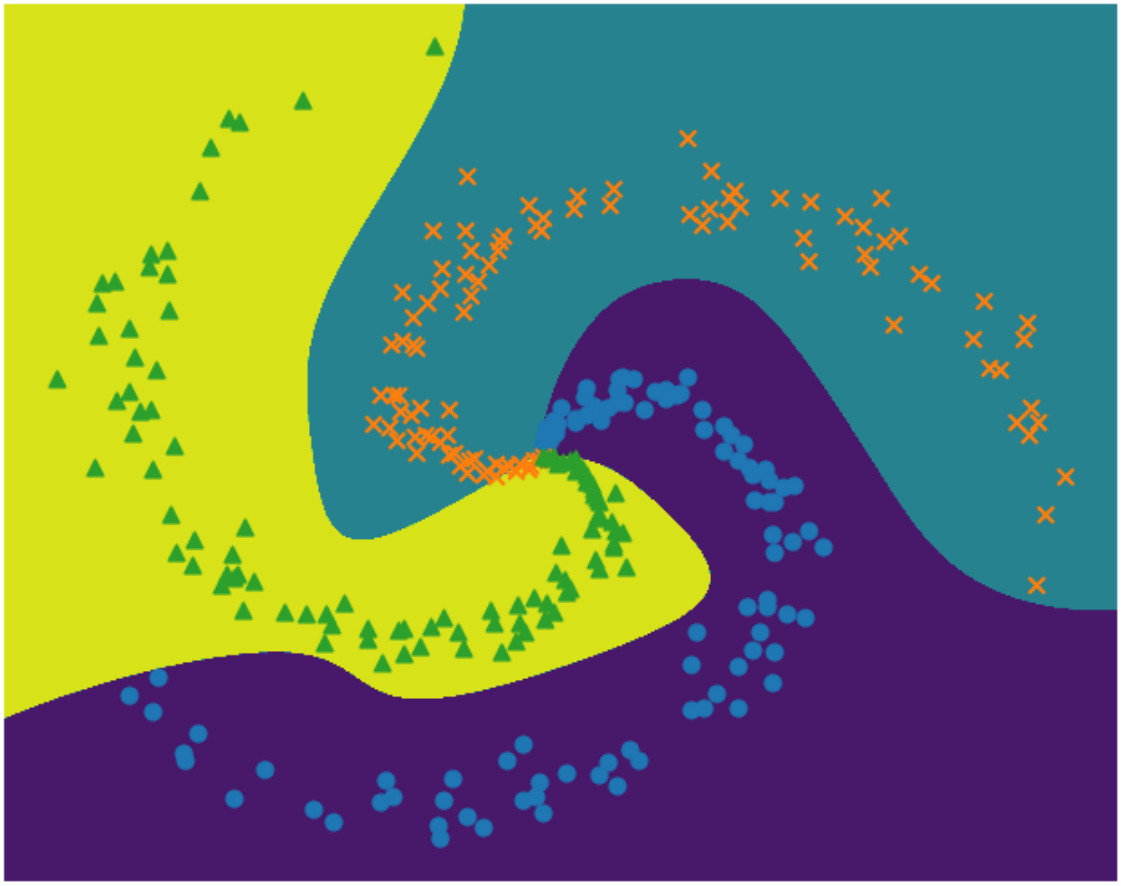
손실 그래프 : 가로축은 학습의 반복 수(눈금 값의 10배), 세로축은 학습 10번 반복당 손실 평균

위 그림에서 보듯, 학습을 진행함에 따라 손실이 줄어 들고 있다. 우리 신경망이 올바른 방향으로 학습되고 있는 것이다. 학습 후 신경망이 영역을 어떻게 분리했는지 시각화해보자. (이를 **결정 경계**라고 한다.) 결과는 다음과 같다.

```
# 경계 영역 플롯
h = 0.001
x_min, x_max = x[:, 0].min() - .1, x[:, 0].max() + .1
y_min, y_max = x[:, 1].min() - .1, x[:, 1].max() + .1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
X = np.c_[xx.ravel(), yy.ravel()]
score = model.predict(X)
```

```
predict_cls = np.argmax(score, axis=1)
Z = predict_cls.reshape(xx.shape)
plt.contourf(xx, yy, Z)
plt.axis('off')

# 데이터점 플롯
x, t = spiral.load_data()
N = 100
CLS_NUM = 3
markers = ['o', 'x', '^']
for i in range(CLS_NUM):
    plt.scatter(x[i*N:(i+1)*N, 0], x[i*N:(i+1)*N, 1], s=40, marker=markers[i])
plt.show()
```

위 그림에서 보듯 학습된 신경망은 '나선형' 패턴을 올바르게 파악했음을 알 수 있다. 즉, 비선형 분리 영역을 학습할 수 있었다! 이처럼 신경망에 은닉층을 추가하면 더 복잡한 표현이 가능해진다. 층을 더 깊게 하면 표현력 또한 더 풍부해지는 것이 딥러닝의 특징이다.



1.4.4 Trainer 클래스

앞에서 언급했듯, 본 학습서를 공부하면서 신경망을 수행할 기회를 많이 준다. 다시 말해, 앞 절에서 본 학습 코드가 자주 필요한데, 매번 똑같은 코드를 다시 쓰고 있으면 아주 지루하겠다. 이에 이 책에서는 학습을 수행하는 역활을 Trainer 라는 클래스로 제공한다. 내용은 앞 절의 소스 코드와 거의 같다.

Trainer 클래스는 common/trainer.py에 있다. 이 클래스의 초기화 메서드는 신경망(모델)과 옵티마이저를 인수로 받는다. 구체적으로는 다음과 같이 사용한다.

```
class Trainer:
    def __init__(self, model, optimizer):
        self.model = model
        self.optimizer = optimizer
        self.loss_list = []
        self.eval_interval = None
        self.current_epoch = 0
```

```
model = TwoLayerNet(...)
optimizer = SGD(lr=1.0)
trainer = Trainer(model, optimizer)
```

그리고 fit() 메서드를 호출해 학습을 시작한다. 이 fit() 메서드가 받는 인수는 다음 표로 정리했다.

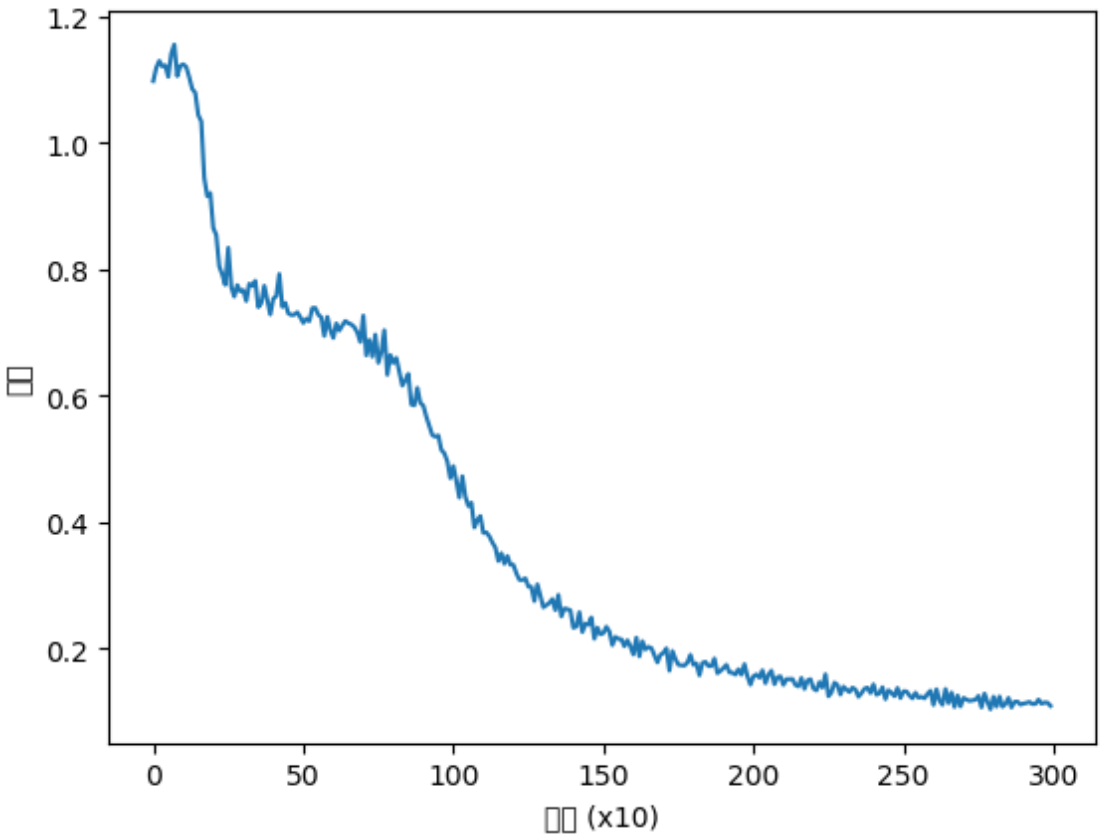
```
    def fit(self, xs, ts, max_epoch=10, batch_size=20, time_size=35,
            max_grad=None, eval_interval=20):
        data_size = len(xs)
        max_iters = data_size // (batch_size * time_size)
        self.time_idx = 0
        self.ppl_list = []
        self.eval_interval = eval_interval
        model, optimizer = self.model, self.optimizer
        total_loss = 0
        loss_count = 0

        start_time = time.time()
        for epoch in range(max_epoch):
            for iters in range(max_iters):
                batch_x, batch_t = self.get_batch(xs, ts, batch_size, time_size)

                # 기울기를 구해 매개변수 갱신
                loss = model.forward(batch_x, batch_t)
                model.backward()
                params, grads = remove_duplicate(model.params, model.grads)  # 공
유된 가중치를 하나로 모음
                if max_grad is not None:
                    clip_grads(grads, max_grad)
                optimizer.update(params, grads)
                total_loss += loss
                loss_count += 1

                # 퍼플렉서티 평가
                if (eval_interval is not None) and (iters % eval_interval) == 0:
                    ppl = np.exp(total_loss / loss_count)
                    elapsed_time = time.time() - start_time
                    print('| 에폭 %d |  반복 %d / %d | 시간 %d[s] | 퍼플렉서티 %.2f'
                          % (self.current_epoch + 1, iters + 1, max_iters,
elapsed_time, ppl))
                    self.ppl_list.append(float(ppl))
                    total_loss, loss_count = 0, 0

            self.current_epoch += 1
```

Trainer 클래스의 fit() 메서드가 받는 인수 : '(=XX)' 는 기본값을 뜻함

인수	설명
x	입력 데이터
t	정답 레이블
max_epoch(=10)	학습을 수행하는 에폭 수
batch_size(=32)	미니배치 크기
eval_interval(=20)	결과(평균 손실 등)를 출력하는 간격 예컨대 eval_interval = 20으로 설정하면, 20번째 마다 손실의 평균을 구해 화면에 출력한다.
max_grad(=None)	기울기 최대 노름 기울기 노름이 이 값을 넘어서면 기울기를 줄인다(이를 기울기 클리핑이라 하며, 자세한 설명은 5장에 나온다.)

Trainer 클래스는 plot() 메서드도 제공한다. 이 메서드는 fit()에서 기록한 손실(정확하게는 eval_interval 시점에 평가된 평균 손실)을 그래프로 그려준다.

```
# coding: utf-8
import sys
sys.path.append('..')  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
from common.optimizer import SGD
from common.trainer import Trainer
from dataset import spiral
from two_layer_net import TwoLayerNet


# 하이퍼파라미터 설정
max_epoch = 300
batch_size = 30
hidden_size = 10
learning_rate = 1.0

x, t = spiral.load_data()
model = TwoLayerNet(input_size=2, hidden_size=hidden_size, output_size=3)
optimizer = SGD(lr=learning_rate)

trainer = Trainer(model, optimizer)
trainer.fit(x, t, max_epoch, batch_size, eval_interval=10)
trainer.plot()
```



위와 같이 코드를 실행하면 이전 절과 같은 신경망 학습이 이뤄진다. 앞에서 본 학습용 코드를 Trainer 클래스에 맡겼기에 코드가 깔끔해졌다. 이 책에서는 앞으로 학습할 때면 항상 Trainer 클래스를 사용할 것이다.

1.5 계산 고속화

신경망의 학습과 추론에 드는 연산량은 상당하다. 이에 신경망에서는 얼마나 빠르게 계산하는지가 매우 중요한 주제다. 이에 이번 절에서는 신경망 고속화에 도움되는 '비트 정밀도'와 'GPU'에 관해 가볍게 설명해보겠다.

Note_ 이 책은 속도보다는 알기 쉽게 구현하기에 우선순위를 두었다. 다만, 고속화의 관점에서 앞으로는 데이터의 비트 정밀도를 의식해 구현한다. 또한 계산이 오래 걸리는 부분에서는 (선택적으로) GPU로 실행할 수 있도록 준비해뒀다.

1.5.1 비트 정밀도

넘파이의 부동소수점 수는 기본적으로 64비트 데이터 타입을 사용한다. 실젤 64비트 보동소수점 수가 사용되는지는 다음 코드로 확인할 수 있다.

```
In [13]: a = np.random.randn(3)

In [14]: a
Out[14]: array([-0.62770321,  1.0043732 , -0.98858318])

In [15]: a.dtype
Out[15]: dtype('float64')
```

이처럼 넘파이 배열의 인스턴스 변수 dtype을 출력해 데이터 타입을 알아볼 수 있다.

넘파이는 64비트 부동소수점 수를 표준으로 사용한다. 그러나 신경망의 추론과 학습은 32비트 부동소수점 수로도 문제없이(인식률을 거의 떨어뜨리는 일 없이) 수행할 수 있다고 한다. 32비트는 64비트의 절반이므로, 메모리 관점에서는 항상 32비트가 더 좋다고 말할 수 있다. 또, 신경망 계산 시 데이터를 전송하는 '버스 대역폭(Bus bandwidth)'이 병목이 되는 경우가 왕왕 있다. 이런 경우에도 데이터 타입이 작은 것이 유리하다. 마지막으로 계산 속도 측면에서도 32비트 부동소수점 수가 일반적으로 더 빠르다. (CPU나 GPU 아키텍쳐에 따라 다르다)

이런 이유로 이 책에서는 32비트 부동소수점 수를 우선으로 사용한다. 넘파이에서 32비트 부동소수점 수를 사용하려면 다음과 같이 데이터 타입을 np.float32나 'f'로 지정한다.

```
In [16]: b = np.random.randn(3).astype(np.float32)

In [17]: b.dtype
Out[17]: dtype('float32')

In [18]: c = np.random.randn(3).astype('f')

In [19]: c.dtype
Out[19]: dtype('float32')
```

또한 신경망 추론으로 한정하면 16비트 부동소수점 수를 사용해도 인식률이 거의 떨어지지 않는다. 그리고 넘파이에도 16비트 부동소수점 수가 준비되어 있다. 다만, 일반적으로 CPU와 GPU는 연산 자체를 32비트로 수행한다. 따라서 16비트 부동소수점 수로 변환하더라도 계산 자체는 32비트로 이뤄져서 처리 속도 측면에서는 혜택이 없을 수도 있다.

그러나 학습된 가중치를 (파일에) 저장할 때는 16비트 부동소수점 수가 여전히 유효하다. 가중치 데이터를 16비트로 저장하면 32비트를 쓸 때보다 절반의 용량만 사용하니 말이다. 이에 이 책에서는 학습된 가중치를 저장하는 경우에 한해 16비트 부동소수점 수로 변환하겠다.

Note_ 딥러닝이 주목받으면서, 최근 GPU들은 '저장'과 '연산' 모두 16비트 반정밀도 부동소수점 수를 지원하도록 진화했다. 구글에서 개발한 TPU칩은 8비트 계산도 지원한다.

1.5.2 GPU(쿠파이)

딥러닝의 계산은 대량의 곱하기 연산으로 구성된다. 이 대량의 곱하기 연산 대부분은 병렬로 계산할 수 있는데 바로 이 점에서는 CPU보다 GPU가 유리하다. 대부분의 딥러닝 프레임워크가 CPU뿐만 아니라 GPU도 지원하는 이유가 바로 이것이다.

이 책의 예제 중에는 쿠파이라는 파이썬 라이브러리를 사용할 수 있는 게 있다. 쿠파이는 GPU를 이용해 병렬 계산을 수행해주는 라이브러리인데, 아쉽게도 엔비디아의 GPU에서만 동작한다. 또한 CUDA라는 GPU 전용 범용 병렬 컴퓨팅 플랫폼을 설치해야 한다.

쿠파이를 사용하면 엔비디아 GPU를 사용해 간단하게 병렬 계산을 수행할 수 있다. 더욱 중요한 점은 쿠파이는 넘파이와 호환되는 API를 제공한다는 사실이다. 다음은 간단한 사용 예시이다.

실행이 안된다. 엔비디아 GPU가 아니기 때문이다.

어찌됐건 쿠파이의 사용법은 기본적으로 넘파이와 같다. 사용법은 같지만 뒤에서 열심히 GPU를 사용해 계산하는 것이다. 다시 말해, 넘파이로 작성한 코드를 'GPU용' 으로 변경하기가 아주 쉽다는 뜻이다. 그저 (보통은) numpy를 cupy로 대체해주기만 하면 끝이다.

다시말하지만, 이 책에서는 이해하기 쉽게 구현하는 것을 우선시하므로, 기본적으로는 CPU에서 수행되는 코드로 작성한다. 그러나 계산이 오래 걸리는 코드는 선택적으로 쿠파이를 사용한 구현도 제공한다.

이 책에서는 GPU를 지원하는 코드가 처음 등장하는 것은 4장이다.

CPU에서 실행하면 몇 시간이 걸리는 코드가 GPU에서는 수십 분이면 된다.

1.6 정리

이번 장에서는 신경망의 기본을 학습했다. 벡터와 행렬 같은 수학 개념부터 시작해서 파이썬 (특히 넘파이)의 기본적인 사용법을 확인했다. 그런 다음 신경망의 구조를 살펴봤다. 특히 계산 그래프의 기본 부품(덧셈 노드와 곱셈 노드 등)을 몇 개 사용해서 순전파와 역전파를 설명했다.

params와 grads라는 인스턴스 변수를 갖는다는 '구현 규칙'을 따랐다.

이번 장에서 배운 내용

- **신경망은 입력층, 은닉층(중간층), 출력층을 지닌다.**
- **완전연결계층에 의해 선형 변환은 이뤄지고, 활성화 함수에 의해 비선형 변환이 이뤄진다.**
- **완전연결계층이나 미니배치 처리는 행렬로 모아 한꺼번에 계산할 수 있다.**
- **오차역전파법을 사용하여 신경망의 손실에 관한 기울기를 효율적으로 구할 수 있다.**
- **신경망이 수행하는 처리는 계산 그래프로 시각화할 수 있으며, 순전파와 역전파를 이해하는데 도움이 된다.**
- **신경망의 구성요소들을 '계층'으로 모듈화해두면, 이를 조립하여 신경망을 쉽게 구성할 수 있다.**
- **신경망 고속화에는 GPU를 이용한 병렬 계산과 데이터의 비트 정밀도가 중요하다.**