9,10장 정리노트

산업데이터사이언스학부
201904213
심성빈

9장

Optimizer 클래스

-매개변수 갱신을 위한 기반 클래스

```
class Optimizer:
    def __init__(self):
        self.target = None
        self.hooks = []

    def setup(self, target):
        self.target = target
        return self

    def update(self):
        params = [p for p in self.target.params() if p.grad is not None]
        for f in self.hooks:
            f(params)

        for param in params:
            self.update_one(param)

    def update_one(self, param):
        raise NotImplementedError()

    def add_hook(self, f):
        self.hooks.append(f)
```

- ➔ 구체적인 최적화 기법은 Optimizer 클래스를 상속한 자식 클래스에서 구현
- ➔ 초기화 메서드 target과 hooks라는 두 개의 인스턴스 변수를 초기화
- ➔ Setup 메서드는 매개변수를 갖는 클래스를 인스턴스 변수인 target으로 설정
- ➔ Update 메서드는 모든 매개변수를 갱신
- ➔ 구체적인 매개변수 갱신은 update_one 메서드에서 수행, 자식 클래스에서 재정의
- ➔ 전처리는 add_hook메서드를 사용하여 전처리 수행

SGD 클래스 구현

-경사하강법으로 매개변수를 갱신하는 클래스를 구현

-SGD클래스는 Optimizer클래스를 상속

```
class SGD(Optimizer):
    def __init__(self, lr=0.01):
        super().__init__()
        self.lr = lr

    def update_one(self, param):
        param.data -= self.lr * param.grad.data
```

➔ __init__메서드는 학습률을 받아 초기화

➔ Update_one 메서그에서 매개변수 갱신 코드를 구현

SGD클래스를 사용하여 회귀 문제 풀기

-MLP 클래스를 사용하여 모델을 생성

-SGD 클래스로 매개변수를 갱신 : optimizer.update()

기울기를 이용한 최적화 기법

-Momentum, AdaGard, AdaDelta, Adam

-Optimizer클래스를 이용해 다양한 최적화 기법을 필요에 따라 손쉽게 전환

-Optimizer 클래스를 상속하여 다양한 최적화 기법을 구현

Momentum 기법

-W는 갱신할 가중치 매개변수

-v는 물리에서 말하는 속도

-알파v는 물체가 아무런 힘을 받지 않을 때 서서히 감속시키는 역할

MomentumSGD 구현 코드

-속도에 해당하는 데이터, 딕셔너리 타입의 변수 self.vs에 유지

-초기화 시에는 vs에 아무것도 담겨있지 않음

```
class MomentumSGD(Optimizer):
    def __init__(self, lr=0.01, momentum=0.9):
        super().__init__()
        self.lr = lr
        self.momentum = momentum
        self.vs = {}

    def update_one(self, param):
        v_key = id(param)
        if v_key not in self.vs:
            xp = cuda.get_array_module(param.data)
            self.vs[v_key] = xp.zeros_like(param.data)

        v = self.vs[v_key]
        v *= self.momentum
        v -= self.lr * param.grad.data
        param.data += v
```

➔ Update_one()이 처음 호출될 때 매개변수와 같은 타입의 데이터를 생성

➔ 구현한 학습 코드에서 손쉽게 Momentum으로 전환

➔ Optimizer = MomentumSGD(lr) 변경

다중 클래스 분류

-여러 클래스로 분류하는 문제

-분류 대상이 여러가지 클래스 중 어디에 속하는지 추정

슬라이스 조작함수

Get_item 함수

-Variable의 다차원 배열 중에서 일부를 슬라이스하여 뽑아줌

-(2,3)형상의 x에서 1번째 행의 원소를 추출

-DeZero함수로 구현했기 때문에 역전파도 제대로 수행

역전파

-y.backward()를 호출하여 역전파 수행

-슬라이스로 인한 계산은 다차원 배열의 데이터 일부를 수정하지 않고 전달

-> 원래의 다차원 배열에서 데이터가 추출된 위치에 행당 기울기를 설정

-> 그 외에는 0으로 설정

슬라이스

-다차원 배열의 일부를 추출하는 작업

-get_item 함수 사용시 같은 인덱스를 반복 지정하여 동일한 원소를 여러 번 빼낼 수 있음

```
x = Variable(np.array([[1, 2, 3], [4, 5, 6]]))
indices = np.array([0, 0, 1])
y = F.get_item(x, indices)
print(y)
```

특수 메서드로 설정

-get_item 함수를 Variable의 메서드로 사용

-슬라이스 작업의 역전파도 이루어짐

신경망으로 다중 클래스 분류

-선형 회귀 때 이용한 신경망을 그대로 사용할 수 있음

-MLP클래스로 구현해둔 신경망을 그대로 이용할 수 있음

-입력 데이터의 치원 수가 2이고 3개의 클래스를 분류하는 문제

-2층으로 이루어진 완전연결 신경망을 만들어 줌

-첫 번째 완전연결 계층의 출력 크기는 10. 두 번째 완전연결 계층의 출력 크기는 3

-model은 입력 데이터를 3차원 벡터로 변환

```
x = np.array([[0.2, -0.4]])
y = model(x)
print(y)
```

➔ x의 형상은 (1,2), 샘플 데이터가 하나 있고 그 데이터는 원소가 2개인 2차원 벡터

➔ 신경망의 출력 형태는 (1,3), 하나의 샘플 데이터가 3개의 원소로 변환

소프트맥스 함수

-신경망 출력이 단순한 수치인데, 이 수치를 확률로 변환

-소프트맥스 함수의 입력 y의 k가 총 n개라고 가정

```python
from dezero import Variable, as_variable
import dezero.functions as F

def softmax1d(x):
    x = as_variable(x)
    y = F.exp(x)
    sum_y = F.sum(y)
    return y / sum_y
```

배치 데이터에도 소프트맥스 함수 적용

배치 데이터를 처리하는 소프트맥스 함수

-인수 x는 2차원 데이터로 가정

-axis =1(행축)

-keepdims = True이므로 각 행에서 나눗셈

```python
def softmax_simple(x, axis=1):
    x = as_variable(x)
    y = exp(x)
    sum_y = sum(y, axis=axis, keepdims=True)
    return y / sum_y
```

➔ 더 나은 구현 방식은 Function 클래스를 상속하여 Softmax 클래스를 구현

➔ 파이썬 함수로 softmax를 구현

교차 엔트로피 오차

-다중 클래스 분류에 적합한 손실 함수

-정답 데이터의 원소는 정답에 해당하는 클래스면 1로, 그렇지 않으면 0으로 기록

-> 이러한 표현 방식을 원핫 벡터라함

-정답 클래스에 해당하는 번호의 확률 p를 추출함으로써 교차 엔트로피의 오차를 계산

교차 엔트로피 오차 구현

-인수 x는 신경망에서 소프트맥스 적용하기 전의 출력, t는 정답 데이터

-clip 함수는 x_min 이하면 x_min으로 변환, x_max이상이면 x_max로 변환

```python
def softmax_cross_entropy_simple(x, t):
    x, t = as_variable(x), as_variable(t)
    N = x.shape[0]
    p = softmax(x)
    p = clip(p, 1e-15, 1.0)  # To avoid log(0)
    log_p = log(p)
    tlog_p = log_p[np.arange(N), t.data]
    y = -1 * sum(tlog_p) / N
    return y
```

➔ Np_arrange(N)은 [0, 1,..., N-1] 형태의 ndarray인스턴스를 생성

➔ X와 정답 데이터 t를 가지고 교차 엔트로피 오차를 계산

다중 클래스 분류

-소프트맥스 함수와 교차 엔트로피 오차를 구현

-스파이럴 데이터셋이라는 작은 데이터셋을 사용하여 다중 클래스 분류 실제 수행

(스파이러은 나선형 혹은 소용돌이 모양)

스파이럴 데이터셋

-get_spiral 함수를 사용하여 스파이럴 데이터셋을 읽어옴

```
import dezero.datasets as ds

x, t = ds.get_spiral(train=True)
print(x.shape)
print(t.shape)

print(x[10], t[10])
print(x[110], t[110])
```

➔ Train= True면 학습용 데이터를 반환, False면 테스트용 데이터를 반환

➔ 반환되는 값은 입력 데이터인 x와 정답 데이터인 t임

➔ X와 t는 모두 ndarray인스턴스이면, 형상은 각각 (300,2)와 (300,)임

➔ 문제는 3클래스 분류이므로 t원소는 0,1,2 중 하나가 됨

다중 클래스 분류 코드

```
import math
import numpy as np
import matplotlib.pyplot as plt
import dezero
from dezero import optimizers
import dezero.functions as F
from dezero.models import MLP

# (1) Hyperparameters
max_epoch = 300
batch_size = 30
hidden_size = 10
lr = 1.0

# (2) 데이터 읽기 / 모델, 옵티마이저 생성
x, t = dezero.datasets.get_spiral(train=True)
model = MLP((hidden_size, 3))
optimizer = optimizers.SGD(lr).setup(model)
```

➔ 하이퍼 파라미터 설정 – 은닉층 수와 학습률

➔ 데이터 셋을 읽고 모델과 옵티마이저를 생성

➔ Max_epoch = 300, 미니배치로 배치 크기는 30으로 설정

```
data_size = len(x)
max_iter = math.ceil(data_size / batch_size)

for epoch in range(max_epoch):
    # (3) Shuffle index for data
    index = np.random.permutation(data_size)
    sum_loss = 0

    for i in range(max_iter):
        # (4) 미니배치 생성
        batch_index = index[i * batch_size:(i + 1) * batch_size]
        batch_x = x[batch_index]
        batch_t = t[batch_index]

        # (5) 기울기 산출 / 매개변수 갱신
        y = model(batch_x)
        loss = F.softmax_cross_entropy(y, batch_t)
        model.cleargrads()
        loss.backward()
        optimizer.update()
        sum_loss += float(loss.data) * len(batch_t)

    # (6) Print loss every epoch
    avg_loss = sum_loss / data_size
    print('epoch %d, loss %.2f' % (epoch + 1, avg_loss))
```

➔ Np.random.permutation 함수를 사용하여 데이터 셋의 인덱스를 무작위로 섞음 (무작위로 정렬된 색인 리스트를 새로 생성)

➔ 미니 배치 생성(미니배치의 인덱스는 방금 생성한 Index에서 앞에서부터 차례로 꺼내 사용)

➔ 기울기를 구하고 매개변수 갱신

➔ 에포크마다 손실 함수 결과 출력

코드 실행 후 손실 그래프

-학습 진행할 수록 손실이 줄어듬

-학습이 완료된 신경망의 클래스 영역에 대한 결정 경계 시각화

-딥러닝의 특징은 층을 더 깊게 쌓는 식으로 표현력을 키울 수 있음

대규모 데이터셋 처리

-스파이럴 데이터셋은 작은 데이터셋이라서 ndarray인스턴스 하나로 처리

-대규모 데이터를 처리할 수 있도록 데이터셋 전용 클래스인 Dataset클래스를 만듬

-Dataset 클래스에는 데이터를 전처리할 수 있는 구조도 추가

Dataset 클래스 구현

-Dataset 클래스는 기반 클래스로서의 역할을 함

-실제로 사용하는 데이터 셋은 이를 상속하여 구현

```
class Dataset:
    def __init__(self, train=True):
        self.train = train
        self.transform = transform
        self.target_transform = target_transform
        if self.transform is None:
            self.transform = lambda x: x
        if self.target_transform is None:
            self.target_transform = lambda x: x
        self.data = None
        self.label = None
        self.prepare()

    def __getitem__(self, index):
        assert np.isscalar(index)
        if self.label is None:
            return self.transform(self.data[index]), None
        else:
            return self.transform(self.data[index]),₩
                   self.target_transform(self.label[index])

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

    def prepare(self):
        pass
```

➔ Prepare메서드가 데이터 준지 작업을 하도록 구현

➔ __getitem__ 메서드는 단순히 지정된 인덱스에 위치하는 데이터를 꺼냄

➔ __len__메서드는 데이터셋의 길이를 알려줌

큰 데이터셋 처리

-작은 데이터셋은 Dataset 클래스의 인스턴스 변수인 data와 label에 직접 ndarray인스턴스를 유지해도 무리가 없음

-큰 데이터셋의 구현 방식은 지금의 방식을 사용할 수 없음

-빅데이터 처리 방법

```
class BigData(Dataset):
    def __getitem__(index):
        x = np.load('data/{}.npy'.format(index))
        t = np.load('label/{}.npy'.format(index))
        return x, t

    def __len__():
        return 1000000
```

➔ BigData클래스를 초기화할 때는 데이터를 읽지 않음

➔ 데이터에 접근하는 __getitem__(index)가 불리는 시점에 데이터를 읽음

신경망 입력 형태로의 데이터 준비

-신경망을 학습시킬 때는 데이터 셋 중 일부를 미니배치로 꺼냄

```
train_set = ds.Spiral()

batch_index = [0, 1, 2]
batch = [train_set[i] for i in batch_index]

x = np.array([example[0] for example in batch])
t = np.array([example[1] for example in batch])

print(x.shape)
print(t.shape)
```

➔ 인덱스를 지정하여 batch에 여러 데이터가 리스트로 저장하고, ndarray인스턴스로 변화 (데이터를 DeZero)의 신경망에 입력하기 위해)

➔ Batch의 각 원소에서 데이터만을 꺼내 하나의 ndarray인스턴스로 변형

Spiral 클래스를 사용하여 학습시 달라진 점

-Spiral클래스 사용시 미니배치를 만드는 부분의 코드를 수정
-Dataset 클래스를 사용하여 신경망을 학습
-> 이 점은 다른 데이터셋으로 교체하는 것만으로 훨씬 큰 데이터셋을 대응 할 수 있음
-> Spiral을 BigData로 교체하는 것만으로 훨씬 큰 데이터셋을 대응할 수 있음
-데이터셋 인터페이스를 통일하여 다양한 데이터셋을 똑같은 코드로 처리 할 수 있음

데이터 셋 전처리

-모델에 데이터를 입력하기 전에 데이터를 특정한 형태로 가공할 경우가 많음
-데이터 형상 변형, 데이터 확장 등등
-초기화 시에 transform과 target_transform을 새롭게 받음
-transform은 입력 데이터 하나에 대한 변한을 처리하고, target_transform레이블 하나에 대한 변환을 처리
-데이터셋에 대해 사용자가 원하는 전처리를 추가 할 수 있음

10장

데이터셋의 인터페이스 제공

-Dataset 클래스를 만들어서 통일된 인터페이스로 데이터셋을 다루도록 제공
-Dataset클래스에서 미니배치를 뽑아주는 DataLoader클래스를 구현
-DataLoader는 미니 배치생성과 데이터셋 뒤섞기 기능 제공

반복자란?

-원소를 반복하여 꺼내주는 역할
-파이썬의 반복자는 리스트나 튜플 등 여러 원소를 담고 있는 데이터 타입으로부터 데이터를 순차적으로 추구하는 기능 제공
-리스트를 반복자로 변환하려면 iter 함수를 사용

-리스트 t에서 x라는 반복자를 만듬
-다음 반복자에서 데이터를 순서대로 추출하기 위해 next 함수를 사용함
-next 함수가 실행될 때마다 리스트의 원소가 차례대로 꺼내짐
-원소가 더 이상 존재하지 않으면 StopIteration 예외 발생

반복자 클래스 구현
-MyIterator 클래스 구현
-클래스를 파이썬 반복자로 사용하려면 __iter__특수 메서드를 구현하고 자기 자신 반환

```
class MyIterator:
    def __init__(self, max_cnt):
        self.max_cnt = max_cnt
        self.cnt = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.cnt == self.max_cnt:
            raise StopIteration()

        self.cnt += 1
        return self.cnt
```

➔ __next__ 메서드는 다음(next) 원소 반환, 반환할 원소가 없으면 raise StopIteration() 수행

DaraLoader 클래스를 구현
-반복자 구조를 이용하여 미니배치를 뽑아줌
-주어진 데이터셋의 첫 데이터부터 차례로 꺼내주지만, 필요에 따라 뒤섞을 수 있음
-초기화 메서드
-> dataset : Dataset 인터페이스를 만족하는 인스턴스
-> batch_size : 배치 크기
-> shuffle : 에포크별로 데이터셋을 뒤섞을지 여부
-reset 메서드는 인스턴스 변수의 반복 횟수를 0으로 설정하고, 필요에 따라 데이터의 인덱스를 뒤섞음
-__next__메서드는 미니배치를 꺼내 ndarray인스턴스로 변환

```
class DataLoader:
    def __init__(self, dataset, batch_size, shuffle=True):
        self.dataset = dataset
        self.batch_size = batch_size
        self.shuffle = shuffle
        self.data_size = len(dataset)
        self.max_iter = math.ceil(self.data_size / batch_size)
        self.gpu = gpu

        self.reset()

    def reset(self):
        self.iteration = 0
        if self.shuffle:
            self.index = np.random.permutation(len(self.dataset))
        else:
            self.index = np.arange(len(self.dataset))

    def __iter__(self):
        return self
def __next__(self):
    if self.iteration >= self.max_iter:
        self.reset()
        raise StopIteration

    i, batch_size = self.iteration, self.batch_size
    batch_index = self.index[i * batch_size:(i + 1) * batch_size
    batch = [self.dataset[i] for i in batch_index]

    xp = cuda.cupy if self.gpu else np
    x = xp.array([example[0] for example in batch])
    t = xp.array([example[1] for example in batch])

    self.iteration += 1
    return x, t

def next(self):
    return self.__next__()
```

DataLoader 클래스 사용

-DataLoader 클래스를 사용하면 미니배치를 꺼내오는 일이 간단해짐

-훈련용과 테스트용 두개의 DataLoader 생성

```
from dezero.datasets import Spiral
from dezero.dataloaders import DataLoader

batch_size = 10
max_epoch = 1

train_set = Spiral(train=True)
test_set = Spiral(train=False)
train_loader = DataLoader(train_set, batch_size)
test_loader = DataLoader(test_set, batch_size, shuffle=False)

for epoch in range(max_epoch):
    for x, t in train_loader:
        print(x.shape, t.shape)
        break

    for x, t in test_loader:
        print(x.shape, t.shape)
        break
```

➔ 훈련용 DataLoader는 에포크별로 데이터를 뒤섞어야 하기 때문에 shuffle = True설정

➔ 미니배치 추출과 데이터 뒤섞기는 DataLoader가 알아서 해줌

Accuracy 함수 구현

-인식 정확도를 평가해주는 함수

-인수 y(신경망의 예측 결과)와 t(정답 데이터)를 받아서 정답률을 계산해 줌

-신경망의 예측결과를 pred에 저장,예측 겨로가의 최대 인덱스를 찾아서 형상 변경

```
def accuracy(y, t):
    y, t = as_variable(y), as_variable(t)

    pred = y.data.argmax(axis=1).reshape(t.shape)
    result = (pred == t.data)
    acc = result.mean()
    return Variable(as_array(acc))
```

```
import numpy as np
import dezero.functions as F

y = np.array([[0.2, 0.8, 0], [0.1, 0.9, 0], [0.8, 0.1, 0.1]])
t = np.array([1, 2, 0])
acc = F.accuracy(y, t)
print(acc)
```

➔ As_array() 함수는 np.float64, np.float32를 ndarray인스턴스로 변환

스파이럴 데이터셋 학습

```
max_epoch = 300
batch_size = 30
hidden_size = 10
lr = 1.0

train_set = dezero.datasets.Spiral(train=True)
test_set = dezero.datasets.Spiral(train=False)
train_loader = DataLoader(train_set, batch_size)
test_loader = DataLoader(test_set, batch_size, shuffle=False)

model = MLP((hidden_size, 3))
optimizer = optimizers.SGD(lr).setup(model)

for epoch in range(max_epoch):
    sum_loss, sum_acc = 0, 0

    for x, t in train_loader: # (1)
        y = model(x)
        loss = F.softmax_cross_entropy(y, t)
        acc = F.accuracy(y, t) # (2)
        model.cleargrads()
        loss.backward()
        optimizer.update()

        sum_loss += float(loss.data) * len(t)
        sum_acc += float(acc.data) * len(t)

    print('epoch: {}'.format(epoch+1))
    print('train loss: {:.4f}, accuracy: {:.4f}'.format(
        sum_loss / len(train_set), sum_acc / len(train_set)))
```

```
sum_loss, sum_acc = 0, 0
with dezero.no_grad(): # (3)
    for x, t in test_loader: # (4)
        y = model(x)
        loss = F.softmax_cross_entropy(y, t)
        acc = F.accuracy(y, t) # (5)
        sum_loss += float(loss.data) * len(t)
        sum_acc += float(acc.data) * len(t)

print('test loss: {:.4f}, accuracy: {:.4f}'.format(
    sum_loss / len(test_set), sum_acc / len(test_set)))
```

➔ DataLoader를 사용해 미니배치를 꺼냄

➔ Accuaracy함수를 사용하여 인식 정확도 계산

➔ 에포그별로 테스트 데이터셋을 사용하여 훈련 결과를 평가 (테스트 시에는 역전파가 필요 없음)

➔ 테스트용 DataLoader에서 미니배치 데이터 꺼내 평가

➔ Accuracy 함수를 사용하여 인식 정확도 계산

손실과 인식 정확도 추이 그래프

-에포크가 진행됨에 따라 손실(loss)이 낮아지고 인식 정확도는 상승

-학습이 제대로 이루어지고 있음을 보임

-훈련과 테스트 차이가 작아 모델이 과대적합을 일으키기 않았다는 뜻

MNIST 학습

DeZero 데이터셋 구조

-Dataset 클래스로 데이터셋 처리를 위한 공통 인터페이스 마련

-전처리 설정 할수 있도록 함

-DataLoader클래스로 Dataset에서 미니배치 단위로 데이터를 꺼내 올 수 있게함

-전처리를 수행하는 객체인 Callable은 Dataset이 보유

-Dataset은 DataLoader가 보유

-사용자는 DataLoader로부터 미니배치를 가져옴

MNIST 데이터셋

-DeZero는 dezero/datasets.py에 MNIST 클래스 준비

-MNIST클래스는 Dataset클래스를 상속

-transform = None으로 설정하여 아무런 전처리도 수행하지 않도록 함

```
from dezero.datasets import MNIST

train_set = MNIST(train=True, transform=None)
test_set = MNIST(train=False, transform=None)

print(len(train_set))
print(len(test_set))
x, t = train_set[0]
print(type(x), x.shape)
print(t)
```

➔ Train_set의 0번째 샘플 데이터를 확인하는 코드

➔ MNIST의 입력 데이터 형상은 (1,28,28) -1채널의 28 x 28 픽셀 이미지 데이터임

➔ 레이블에는 정답 숫자의 인덱스 (0 ~ 9)가 들어있음

입력 데이터 시각화

```
import matplotlib.pyplot as plt

x, t = train_set[0]
plt.imshow(x.reshape(28, 28), cmap='gray')
plt.axis('off')
plt.show()
print('label:', t)
```

입력 데이터 전처리

-(1,28,28)형상인 입력 데이터를 평탄화하여 (784,) 형상으로 변환

-데이터 타입을 np.float32로 변환

```
def f(x):
    x = x.flatten()
    x = x.astype(np.float32)
    x /= 255.0
    return x

train_set = dezero.dataset.MNIST(train=True, transform=f)
test_set = dezero.dataset.MNIST(train=False, transform=f)
```

➔ MNIST(train = True)로 호출하면 이상의 전처리가 자동으로 수행

➔ Dezero/trainforms.py에 정의된 클래스를 이용하여 전처리하도록 작성되어 있음

MNIST 학습하기

-이전 단계와 달라진 점은 MNIST 데이터셋 사용과 하이퍼피라미터 값 변경

-인식 정확도는 테스트 데이터셋에서 86%를 얻음

모델 개선하기

활성화 함수 변경

-활성화 함수를 시그모이드에서 ReLU로 변경

-ReLU는 입력이 0보다 크면 입력 그대로 출력하고, 0이하면 0을 출력하는 함수

ReLU함수 구현

```
class ReLU(Function):
    def forward(self, x):
        y = xp.maximum(x, 0.0) # (1)
        return y

    def backward(self, gy):
        x, = self.inputs
        mask = x.data > 0 # (2)
        gx = gy * mask  # (3)
        return gx

def relu(x):
    return ReLU()(x)
```

➔ 순전파에서는 np.maximum(x, 0.0)에 의해 x의 원소와 0.0중 큰 쪽을 반환함

➔ 역전파에서는 입력 x에서 0보다 큰 원소에 해당하는 위치의 기울기는 그대로 흘려보내고, 0이하라면 기울기를 0으로 설정해야함

➔ 출력 쪽에서 전해지는 기울기를 통과시킬지 표시한 마스크를 준비한 후 가울기를 곱합

ReLU함수를 사용한 새로운 신경망 만들기

-3층 신경망을 만듬
-활성화 함수를 ReLU로 바꾸어서 학습함
-이 신경망에서 최적화 기법을 SGD에서 Adam으로 바꾼 후 학습을 수행
-훈련용 데이터는 약 99%, 테스트용 데이터는 약 98%라는 인식 정확도를 얻을 수 있음

느낀점

Attention코드 분석을 하면서 나온 부분들이 있어 이해 안된 부분을 이해할 수 있게 되면서 정확하게 알게 되었다.