복합 출력(오피스31 다차원 분류 신경망)

adam_model.ipynb

%run ../chap05/mlp_model.ipynb

: 이전 실습에서 사용했던 패키지 등을 불러온다.

```
class AdamModel(MlpModel):
    def __init__(self, name, dataset, hconfigs):
        self.use_adam = False
        super(AdamModel, self).__init__(name, dataset, hconfigs)
```

: 객체 초기화를 위한 메소드만 재정의하고 아담 알고리즘에 관련된 메소드들은 클래스 선언 밖에서 함수로 정의해 등록한다. 아담 알고리즘은 모델 객체 생성 과정과 무관하며 이후 모델이 수행되기 전에 use_adam을 True로 설정해 아담 알고리즘을 이용하면 된다.

```
def adam_backprop_layer(self, G_y, hconfig, pm, aux):
    x, y = aux

    if hconfig is not None: G_y = relu_derv(y) * G_y

    g_y_weight = x.transpose()
    g_y_input = pm['w'].transpose()

    G_weight = np.matmul(g_y_weight, G_y)
    G_bias = np.sum(G_y, axis=0)
    G_input = np.matmul(G_y, g_y_input)

    self.update_param(pm, 'w',   G_weight)
    self.update_param(pm, 'b',   G_bias)

    return G_input
```

AdamModel.backprop_layer = adam_backprop_layer

: self.update_param 부분만 self.use_adam 플래그 값에 따라 기존 방식 혹은 Adam 방식에 따라 동작

def adam_update_param(self, pm, key, **delta**): **# delta : 파라미터의 손실 기울기**

if self.use_adam:

delta = self.eval_adam_delta(pm, key, delta) # adam 적용 후 delta 갱신

pm[key] -= self.learning_rate * delta # adam으로 갱신된 delta값과 self.learning_rate에 저장된 learning rate를 이용해서 파라미터 값 수정

AdamModel.update_param = adam_update_param

: 파라미터의 손실 기울기 값에 adam을 적용하여 수정. -> 바뀐 delta값과 learning_rate에 저장된 학습률 정보를 이용해 파라미터 값 수정

def adam_eval_adam_delta(self, pm, key, delta):

ro_1 = 0.9

ro_2 = 0.999

epsilon = 1.0e-8

skey, tkey, step = 's' + key, 't' + key, 'n' + key

if skey not in pm:

pm[**skey**] = np.zeros(pm[key].shape)

pm[**tkey**] = np.zeros(pm[key].shape)

pm[step] = 0

s = pm[skey] = ro_1 * pm[skey] + (1 - ro_1) * delta

t = pm[tkey] = ro_2 * pm[tkey] + (1 - ro_2) * (delta * delta)

pm[step] += 1

```
    s = s / (1 - np.power(ro_1, pm[step]))
    t = t / (1 - np.power(ro_2, pm[step]))

    return s / (np.sqrt(t)+epsilon)
```

AdamModel.eval_adam_delta = adam_eval_adam_delta

: 이동 평균 계산에서 기존 값의 비율을 ro_1, ro_2라는 hyperparameter로 이용, 분모의 임계값 epsilon도 hyperparameter로 이용해 분모가 0이 되어 생기는 나눗셈 오류 방지

모멘텀, 2차 모멘텀 값, 처리 횟수를 각각 skey, tkey, step라는 이름으로 pm구조에 저장.(buffer size 부족 대비해 버퍼 공간 생성)

기존값과 새 값의 가중평균(이동평균)을 계산하여 모멘텀 값과 2차 모멘텀 값을 구한다.

np.zeros()를 통해 기존 값들이 0으로 초기화되었기 때문에 초반 처리 과정에서는 기존 값의 영향을 빨리 벗어나게 해줘야 하는데, 이를 위해 비율 값의 거듭제곱과 나눗셈 연산을 이용하는 처리를 두어, step가 작으면 모멘텀 값과 2차 모멘텀 값을 증폭시켰다.

return에 epsilon을 더해준 것은 분모가 0이나 0에 가까운 값이 되는 것을 방지하기 위해서이다.

dataset_office31

%run ../chap05/dataset.ipynb

: dataset을 가져온다.

```
class Office31Dataset(Dataset):
    @property
    def base(self):
        return super(Office31Dataset, self)
```

: @property를 통해 super 함수 호출을 간단하게 하며 인자가 필요없는 base()를 사용. 이에 따라 self.base로 줄여 표현 가능하다.

```
def office31_init(self, resolution=[100,100], input_shape=[-1]):
    self.base.__init__('office31', 'dual_select')

    path = '../../data/chap06/office31'
```

```
domain_names = list_dir(path)

images = []
didxs, oidxs = [], []

for dx, dname in enumerate(domain_names): # 순회하면서 .jpg 파일을 읽는다.
    domainpath = os.path.join(path, dname, 'images')
    object_names = list_dir(domainpath)
    for ox, oname in enumerate(object_names):
        objectpath = os.path.join(domainpath, oname)
        filenames = list_dir(objectpath)
        for fname in filenames:
            if fname[-4:] != '.jpg':
                continue
            imagepath = os.path.join(objectpath, fname)
            pixels = load_image_pixels(imagepath, resolution, input_shape)
            images.append(pixels)
            didxs.append(dx)
            oidxs.append(ox)

self.image_shape = resolution + [3] # RGB

xs = np.asarray(images, np.float32) # 이미지 내용을 2차원 숫자 배열 형식으로 변경

ys0 = onehot(didxs, len(domain_names)) # ys0, ys1을 one-hot vector 형태로 변경
ys1 = onehot(oidxs, len(object_names))
ys = np.hstack([ys0, ys1]) # ys0, ys1를 넣어 길이 34의 벡터들로 구성된 정답 정보를 만듬

self.shuffle_data(xs, ys, 0.8)
```

```
        self.target_names = [domain_names, object_names]
        self.cnts = [len(domain_names)]
```

: 주석에서 설명

```
def office31_forward_postproc(self, output, y):
        outputs, ys = np.hsplit(output, self.cnts), np.hsplit(y, self.cnts)

        loss0, aux0 = self.base.forward_postproc(outputs[0], ys[0], 'select')
        loss1, aux1 = self.base.forward_postproc(outputs[1], ys[1], 'select')

        return loss0 + loss1, [aux0, aux1]
```

: 로짓값과 정답에 대해 softmax crossentropy를 구해 더하면 전체 Loss값이 구해진다. forward_postproc는 기존에 구해진 것을 이용한다.

```
def office31_backprop_postproc(self, G_loss, aux):
        aux0, aux1 = aux

        G_ouput0 = self.base.backprop_postproc(G_loss, aux0, 'select')
        G_ouput1 = self.base.backprop_postproc(G_loss, aux1, 'select')

        return np.hstack([G_ouput0, G_ouput1])
```

: 도메인과 품목 정보에 대한 손실 기울기를 따로 구한 후 둘을 연결하는 방식으로 수행

```
def office31_eval_accuracy(self, x, y, output):
        outputs, ys = np.hsplit(output, self.cnts), np.hsplit(y, self.cnts)

        acc0 = self.base.eval_accuracy(x, ys[0], outputs[0], 'select')
        acc1 = self.base.eval_accuracy(x, ys[1], outputs[1], 'select')
```

```
        return [acc0, acc1]

def office31_train_prt_result(self, epoch, costs, accs, acc, time1, time2):
        acc_pair = np.mean(accs, axis=0)
        print('       Epoch {}: cost={:5.3f}, ' ₩
                'accuracy={:5.3f}+{:5.3f}/{:5.3f}+{:5.3f} ({}/{} secs)'. ₩
                format(epoch, np.mean(costs), acc_pair[0], acc_pair[1], ₩
                        acc[0], acc[1], time1, time2))

def office31_test_prt_result(self, name, acc, time):
        print('Model {} test report: accuracy = {:5.3f}+{:5.3f}, ({} secs)₩n'. ₩
                format(name, acc[0], acc[1], time))
```

: 도메인과 품목에 대한 각각의 정확도(acc1, acc2)를 얻어낼 수 있다.

```
def office31_get_estimate(self, output):
        outputs = np.hsplit(output, self.cnts)

        estimate0 = self.base.get_estimate(outputs[0], 'select')
        estimate1 = self.base.get_estimate(outputs[1], 'select')

        return np.hstack([estimate0, estimate1])

def office31_visualize(self, xs, estimates, answers):
        draw_images_horz(xs, self.image_shape)

        ests, anss = np.hsplit(estimates, self.cnts), np.hsplit(answers, self.cnts)

        captions = ['도메인', '상품']
```

```
for m in range(2):
    print('[ {} 추정결과 ]'.format(captions[m]))
    show_select_results(ests[m], anss[m], self.target_names[m], 8)
```

: 도메인 선택과 품목 선택에 대한 추정치 산출 메소드를 호출한다.

두 번 반복 실행하여 처음에는 도메인 분류에 대해, 두 번째는 상품 분류에 대해 show_select_results() 함수가 호출되어 선택 내용을 출력

Office31Dataset.__init__ = office31_init

Office31Dataset.forward_postproc = office31_forward_postproc

Office31Dataset.backprop_postproc = office31_backprop_postproc

Office31Dataset.eval_accuracy = office31_eval_accuracy

Office31Dataset.get_estimate = office31_get_estimate

Office31Dataset.train_prt_result = office31_train_prt_result

Office31Dataset.test_prt_result = office31_test_prt_result

Office31Dataset.visualize = office31_visualize

: Office31Dataset 클래스의 메소드로 일괄 등록

결론

[Test1] epoch_count = 20, hidden layer 1개(퍼셉트론 10)인 모델

```
om1 = AdamModel('office31_model_1', od, [10])
om1.exec_all(epoch_count=20, report=10)
```

```
Model office31_model_1 train started:
    Epoch 10: cost=4.296, accuracy=0.685+0.035/0.790+0.040 (42/42 secs)
    Epoch 20: cost=4.268, accuracy=0.685+0.037/0.780+0.020 (42/84 secs)
Model office31_model_1 train ended in 84 secs:
Model office31_model_1 test report: accuracy = 0.661+0.048, (0 secs)

Model office31_model_1 Visualization
```

```
[ 도메인 추정결과 ]
추정확률분포 [65,15,20] => 추정 amazon : 정답 amazon => O
추정확률분포 [65,15,20] => 추정 amazon : 정답 amazon => O
추정확률분포 [65,15,20] => 추정 amazon : 정답 dslr => X
[ 상품 추정결과 ]
추정확률분포 [ 3, 3, 3, 3, 3, 3, 3, 3,...] => 추정 monitor : 정답 ring_binder
=> X
추정확률분포 [ 3, 3, 3, 3, 3, 3, 3, 3,...] => 추정 monitor : 정답 pen => X
추정확률분포 [ 3, 3, 3, 3, 3, 3, 3, 3,...] => 추정 monitor : 정답 mouse => X
```

각각 정확도 0.661, 0.048

[Test2] epoch_count = 50, hidden layer 3개(퍼셉트론 각각 64, 32, 10)인 모델, learning rate = 0.0001

```
om2 = AdamModel('office31_model_2', od, [64,32,10])
om2.exec_all(epoch_count=50, report=10, learning_rate=0.0001)
```

```
Model office31_model_2 train started:
    Epoch 10: cost=3.745, accuracy=0.805+0.098/0.860+0.080 (99/99 secs)
    Epoch 20: cost=3.471, accuracy=0.843+0.146/0.830+0.090 (95/194 secs)
    Epoch 30: cost=3.263, accuracy=0.866+0.198/0.840+0.100 (103/297 secs)
    Epoch 40: cost=3.106, accuracy=0.880+0.223/0.880+0.140 (102/399 secs)
    Epoch 50: cost=2.964, accuracy=0.889+0.252/0.880+0.160 (106/505 secs)
Model office31_model_2 train ended in 505 secs:
Model office31_model_2 test report: accuracy = 0.867+0.197, (0 secs)

Model office31_model_2 Visualization
```

```
[ 도메인 추정결과 ]
추정확률분포 [100, 0, 0] => 추정 amazon : 정답 amazon => 0
추정확률분포 [100, 0, 0] => 추정 amazon : 정답 amazon => 0
추정확률분포 [96, 1, 3] => 추정 amazon : 정답 amazon => 0
[ 상품 추정결과 ]
추정확률분포 [ 1, 0, 0,23, 3, 9, 6,16,...] => 추정 bookcase : 정답 tape_dispe
nser => X
추정확률분포 [ 0, 1, 4, 0, 0, 0, 0, 0,...] => 추정 keyboard : 정답 letter_tra
y => X
추정확률분포 [ 0, 1, 0, 1, 1, 3, 1, 8,...] => 추정 pen : 정답 ruler => X
```

각각 정확도 0.867, 0.197

[Test3] epoch_count = 50, hidden layer 3개(퍼셉트론 각각 64, 32, 10)인 모델, learning rate = 0.0001, Adam Algorithm 사용

```
Model office31_model_3 train started:
    Epoch 10: cost=3.713, accuracy=0.818+0.084/0.870+0.040 (62/62 secs)
    Epoch 20: cost=3.322, accuracy=0.865+0.160/0.880+0.130 (61/123 secs)
    Epoch 30: cost=3.118, accuracy=0.880+0.203/0.900+0.160 (63/186 secs)
    Epoch 40: cost=2.948, accuracy=0.891+0.234/0.860+0.250 (61/247 secs)
    Epoch 50: cost=2.812, accuracy=0.905+0.270/0.880+0.210 (63/310 secs)
Model office31_model_3 train ended in 310 secs:
Model office31_model_3 test report: accuracy = 0.878+0.246, (0 secs)

Model office31_model_3 Visualization
```

```
[ 도메인 추정결과 ]
추정확률분포 [99, 1, 0] => 추정 amazon : 정답 amazon => 0
추정확률분포 [100, 0, 0] => 추정 amazon : 정답 amazon => 0
추정확률분포 [100, 0, 0] => 추정 amazon : 정답 amazon => 0
[ 상품 추정결과 ]
추정확률분포 [ 0, 1, 1, 2, 4, 2, 2,10,...] => 추정 scissors : 정답 scissors => 0
추정확률분포 [ 2, 1, 1,11, 6, 3, 0,10,...] => 추정 paper_notebook : 정답 printer => X
추정확률분포 [ 0, 0, 1, 2, 0, 1, 2, 1,...] => 추정 stapler : 정답 calculator => X
```

각각 정확도 0.878, 0.246

use_adam = True로 설정하여 adam 알고리즘을 적용하도록 설정하였다.

adam을 이용하여 무조건 학습 결과가 좋아지는 것은 아니지만 이전과 동일한 조건으로 adam만 적용하였을 때, 학습 정확도가 향상되는 것을 확인할 수 있었다.

아담 알고리즘(Adaptive Moments)을 사용할 때 기존의 SGD(Stochastic Gradient)를 사용하는 것에 비해 어떤 장점이 있는가?

Gradient Descent는 모든 자료를 검토해서 최적의 step을 찾으며 나아가는 방식이라서 확실하지만 모든 자료를 검토하는 시간이 매우 오래 걸린다.

SGD는 **Mini-batch**만 계산하여 검토의 소요시간을 단축시키고, 빠르게 전진한다. **최적의 Step을 밟으며 Global Optima로 나아가지는 않지만, 빠르게 갈 수 있다.**

Adam은 Adagrad와 RMSprop의 장점을 섞어 놓은 것이다. 장점은 **stepsize가 gradient의 rescaling에 영향을 받지 않는다는 것. 즉, Gradient가 커져도 stepsize는 bound 되어 있어서 어떠한 objective function을 사용한다고 하더라도 안정적으로 최적화를 위한 하강이 가능하다.** 게다가 **stepsize를 과거의 gradient 크기를 사용하여 adapted 시킬 수 있다.**

Adagard : 과거의 gradient 변화량을 참고하여 이미 많이 변화한 변수들은 optima에 거의 도달했다고 보고 stepsize를 작게 하고, 많이 변화되지 않은 변수들은 아직 가야할 길이 멀다고 보고 stepsize를 크게 한다. 따라서 Gt 변수를 도입해서 여태까지의 graident의 L2 norm을 저장한다.

$$\theta_t = \theta_{t-1} - \alpha \frac{g_t}{\sqrt{\sum_{i=1}^{t} g_i^2}}$$

문제점 : iteration이 계속될수록 g가 계속 증가해서 stepsize가 너무 작아질 수 있다. 이러한 문제를 보완하기 위해서 RMSprop처럼 **exponential moving average를 사용하는 방법**이 고안되었다. 수식에서 알 수 있듯이 이 방법은 **과거의 정보에 가중치를 적게 부여한다.**

RMSprop에서는 이런 점을 반영하여 수식을 수정하였다.

$$G_t = \gamma G_{t-1} + (1-\gamma) g_t^2$$

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{G_t + \epsilon}} g_t$$

문제점 : G 변수의 bias를 조정하지 않으면 B1이 1에 매우 가까워질수록 매우 큰 stepsize를 갖거나 발산하는 문제가 발생할 수 있다(iteration 초기에 주로 발생하는 문제)

아담 알고리즘을 이용하는 경우 개별 파라미터 수준에서 따로 계산되고 관리되는 모멘텀 정보와 2차 모멘텀 정보를 가진다. 이로 인해 파라미터 관리에 필요한 **메모리 소비량이 약 3배**로 늘어난다. 미니 배치 데이터를 처리해 학습이 이루어질 때마다 파라미터 값은 물론 모멘텀 정보도 다시 계산되어야 하기 때문에 **계산 부담도 커지게 된다.**

아담 알고리즘은 파라미터 별로 실질적인 학습률을 보정해 적용함으로써 일괄적인 학습률 적용에서 오는 품질 저하를 막아준다. (**파라미터를 갱신하는 마지막 순간에 학습률을 이용함**)

아담 알고리즘을 사용한다고 해도 적합한 학습률을 찾는 노력이 완전히 필요하지 않은 것은 아니다.

전이 학습(Transfer Learning) : 한 도메인에서 학습 시킨 결과를 다른 도메인에 활용하여 학습 효과를 높이는 학습 기법

- 기존 도메인의 데이터들이 **레이블 정보가 없거나 아주 적은 양만 있을 때**, 이미 확보된 다른 도메인 모델의 학습 정보를 잘 활용하면 다른 도메인에서 가져온 것일지라도 기존 도메인에서 더 나은 성능을 낼 수 있다.
- 레이블 정보를 표시하는 레이블링 작업에는 많은 경우 데이터 수집 이상의 더 많은 인력 및 시간이 든다. 하지만 레이블링이 전혀 없이 신경망을 학습시키는 방법은 마땅치 않기 때문에 이런 **레이블링 과정에서의 비용을 줄이기 위해** 전이 학습을 이용한다.