Оглавление

ЧАСТЬ II
ПОНИМАНИЕ ПРИЛОЖЕНИЙ НА БАЗЕ FASTAI

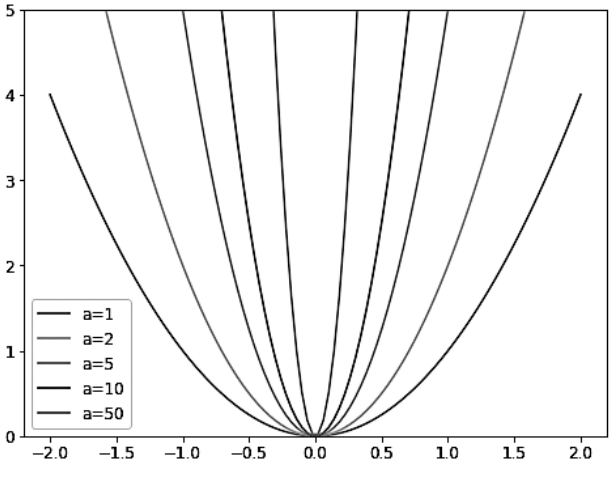
ЧАСТЬ IV
ГЛУБОКОЕ ОБУЧЕНИЕ С ЧИСТОГО ЛИСТА

ПРИЛОЖЕНИЯ

Сокращение весов

Сокращение весов, иначе называемое *регуляризацией L2*, представляет собой добавление к функции потерь, возведенной в квадрат, суммы всех весов. Зачем нам это делать? Потому что при вычислении градиентов это будет добавлять им вклад, способствующий максимальному сокращению весов.

Почему это должно предотвратить переобучение? Суть в том, что чем больше коэффициенты, тем круче спуски, получаемые в функции потерь. Если взять простой пример параболы `y = a * (x**2)`, то чем больше `a`, тем более *узкой* получается парабола:



Поэтому если позволить модели обучать высокие параметры, то она подстроит все точки данных обучающей выборки с помощью сверхсложной функции, имеющей резкие изменения, что приведет к переобучению.

Препятствуя чрезмерному увеличению весов, мы замедлим обучение модели, но она окажется в состоянии, способствующем лучшей обобщаемости. Если коротко вернуться к теории, то сокращение весов (или просто `wd`) — это параметр, контролирующий сумму квадратов, добавляемых нами к потерям (предполагая, что `parameters` является тензором всех параметров):

```
loss_with_wd = loss + wd * (parameters**2).sum()
```

Тем не менее на практике будет очень неэффективно (а может, и численно нестабильно) вычислять настолько большую сумму и добавлять ее к потерям. Вы можете припомнить из курса высшей математики, что производная `p**2` по отношению к `p` равна `2*p`. Поэтому добавление такой большой суммы к потерям эквивалентно следующему:

```
parameters.grad += wd * 2 * parameters
```

В реальности, так как `wd` является выбираемым нами параметром, мы можем его удвоить, исключив из данного уравнения `*2`. Для использования сокращения весов в fastai нужно передать `wd` в вызов `fit` или `fit_one_cycle` (можно передать в оба):

```
model = DotProductBias(n_users, n_movies, 50)
learn = Learner(dls, model, loss_func=MSELossFlat())
learn.fit_one_cycle(5, 5e-3, wd=0.1)
```

epoch	train_loss	valid_loss	time
0	0.972090	0.962366	00:13
1	0.875591	0.885106	00:13
2	0.723798	0.839880	00:13
3	0.586002	0.823225	00:13
4	0.490980	0.823060	00:13

Намного лучше!

Создание собственного модуля вложений

До сих пор мы использовали `Embedding`, не обращая внимания на то, как он фактически работает. Давайте воссоздадим `DotProductBias` *без* применения этого класса. Нам понадобится матрица случайным образом инициализированных весов для каждого из вложений. При этом нужно быть осторожными. В главе 4 мы писали, что оптимизаторы требуют возможности получения всех параметров модуля из его метода `parameters`. Тем не менее это происходит не полностью автоматически. Если мы просто добавим тензор к `Module` в качестве атрибута, то в `parameters` он включен не будет:

```
class T(Module):
    def __init__(self): self.a = torch.ones(3)

L(T().parameters())
(#0) []
```

Чтобы сообщить `Module` о своем желании рассматривать тензор как параметр, нужно обернуть его в класс `nn.Parameter`. Этот класс не привносит никакую функциональность (за исключением автоматического вызова `requires_grad`). Он просто используется как «маркер», показывающий, что нужно включить в `parameters`:

```
class T(Module):
    def __init__(self): self.a = nn.Parameter(torch.ones(3))
```

```
L(T().parameters())
(#1) [Parameter containing:
tensor([1., 1., 1.], requires_grad=True)]
```

Все модули PyTorch задействуют `nn.Parameter` для всех обучаемых параметров, почему нам и не требовалось специально использовать эту обертку до текущего момента:

```
class T(Module):
    def __init__(self): self.a = nn.Linear(1, 3, bias=False)

t = T()
L(t.parameters())

(#1) [Parameter containing:
tensor([[-0.9595],
        [-0.8490],
        [ 0.8159]], requires_grad=True)]

type(t.a.weight)

torch.nn.parameter.Parameter
```

Мы можем создать тензор в качестве параметра, используя случайную инициализацию:

```
def create_params(size):
    return nn.Parameter(torch.zeros(*size).normal_(0, 0.01))
```

Используем это для повторного создания `DotProductBias`, но без `Embedding`:

```
class DotProductBias(Module):
    def __init__(self, n_users, n_movies, n_factors, y_range=(0,5.5)):
        self.user_factors = create_params([n_users, n_factors])
        self.user_bias = create_params([n_users])
        self.movie_factors = create_params([n_movies, n_factors])
        self.movie_bias = create_params([n_movies])
        self.y_range = y_range

    def forward(self, x):
        users = self.user_factors[x[:,0]]
        movies = self.movie_factors[x[:,1]]
        res = (users*movies).sum(dim=1)
        res += self.user_bias[x[:,0]] + self.movie_bias[x[:,1]]
        return sigmoid_range(res, *self.y_range)
```

Еще раз проведем обучение, чтобы проверить, получим ли мы такой же результат, что и в предыдущем разделе:

```
model = DotProductBias(n_users, n_movies, 50)
learn = Learner(dls, model, loss_func=MSELossFlat())
learn.fit_one_cycle(5, 5e-3, wd=0.1)
```

epoch	train_loss	valid_loss	time
0	0.962146	0.936952	00:14
1	0.858084	0.884951	00:14
2	0.740883	0.838549	00:14
3	0.592497	0.823599	00:14
4	0.473570	0.824263	00:14

А теперь посмотрим, чему наша модель научилась.

Интерпретация вложений и смещений

Наша модель уже достаточно эффективна в своей способности предоставлять рекомендации для наших пользователей, но при этом очень интересно узнать, какие же параметры она обнаружила. Проще всего интерпретировать смещения. Вот фильмы с наименьшими значениями в векторе смещений:

```
movie_bias = learn.model.movie_bias.squeeze()
idxs = movie_bias.argsort()[:5]
[dls.classes['title'][i] for i in idxs]
```

```
['Children of the Corn: The Gathering (1996)',
 'Lawnmower Man 2: Beyond Cyberspace (1996)',
 'Beautician and the Beast, The (1997)',
 'Crow: City of Angels, The (1996)',
 'Home Alone 3 (1997)']
```

Подумайте о том, что это значит. Здесь говорится, что, несмотря на высокое соответствие пользователя латентным факторам приведенных фильмов (которые, как мы вскоре увидим, представляют уровень экшена, возрастную категорию фильма и т. д.), сам фильм, как правило, пользователю не нравится. Мы могли бы просто отсортировать фильмы непосредственно по их средней оценке, но, глядя на обученное смещение, становится понятно кое-что более интересное. Это говорит нам не только о том, что фильм относится к категории тех, которые пользователи не любят смотреть, но и о том, что пользователи не хотели бы смотреть такой фильм, даже если он относится к жанру, который их интересует. А вот аналогичный результат, но уже для фильмов с самым высоким смещением:

```
idxs = movie_bias.argsort(descending=True)[:5]
[dls.classes['title'][i] for i in idxs]
```

```
['L.A. Confidential (1997)',
 'Titanic (1997)',
 'Silence of the Lambs, The (1991)',
 'Shawshank Redemption, The (1994)',
 'Star Wars (1977)']
```

То есть, например, даже если обычно вы не цените детективные фильмы, *LA Confidential* может вам понравиться.

Напрямую интерпретировать матрицы вложений не так-то просто. Здесь присутствует слишком много факторов для просмотра. Но при этом есть техника, которая способна выделить в такой матрице наиболее важные определяющие *направления*. Называется эта техника *методом главных компонентов* (principal componentlysis — PCA).

Подробно рассматривать эту технику мы не станем, потому что ее понимание не особо важно для успешной практики глубокого обучения. Но если вам интересно, то предлагаем ознакомиться с курсом fast.ai под названием «Вычислительная линейная алгебра для программистов» (https://oreil.ly/NLj2R). На рис. 8.3 показано, как выглядят фильмы на основе двух самых сильных компонентов PCA.

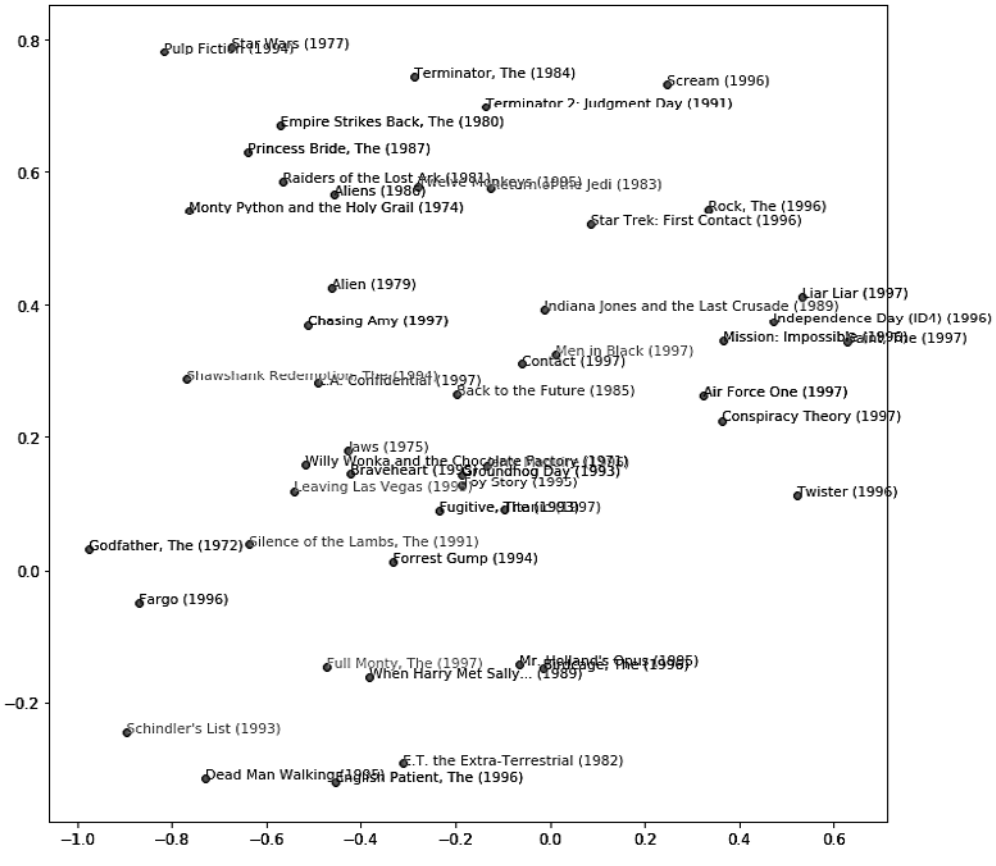


Рис. 8.3. Представление фильмов на основе двух самых сильных компонентов PCA