Глава 10
Нейробайесовские методы,
или Прошлое и будущее машинного обучения

TL;DR

В этой достаточно математически насыщенной главе будет дано краткое и поверхностное введение в то, как сочетаются в современном машинном обучении нейронные сети и байесовский вывод. Мы:

- подробно разберем общий вид алгоритма EM для обучения моделей со скрытыми переменными;
- увидим на простых примерах основные идеи вариационных приближений;
- поговорим о конструкции вариационного автокодировщика, еще одной важной порождающей модели с глубокими нейронными сетями;
- отметим, как байесовский вывод помогает построить новые формы дропаута;
- закончим книгу нашими размышлениями о том, куда все это катится и что ждет нас в будущем.

10.1. Теорема Байеса и нейронные сети

> О святая математика, в общении с тобой хотел бы я провести остаток дней своих, забыв людскую злобу и несправедливость Вседержителя.
>
> *Лотреамон*

В главе 2 мы говорили о теореме Байеса и ее роли в машинном обучении, долго обсуждали пример с априорными и апостериорными распределениями броска монетки. А на протяжении всей книги не раз упоминали, что байесовский вывод в нейронных сетях тоже присутствует, и ссылались на какие-то загадочные вариационные приближения, которые якобы делают возможным вывод в очень сложных моделях. В этой главе мы на простых примерах поймем, что это такое. Но сначала давайте ненадолго вернемся к самой теореме Байеса:

$$p(\boldsymbol{\theta}|D) = \frac{p(\boldsymbol{\theta})p(D|\boldsymbol{\theta})}{p(D)} = \frac{p(\boldsymbol{\theta})p(D|\boldsymbol{\theta})}{\int_{\boldsymbol{\theta}\in\Theta} p(D \mid \boldsymbol{\theta})p(\boldsymbol{\theta})\mathrm{d}\boldsymbol{\theta}}.$$

В машинном обучении теорема Байеса позволяет из правдоподобия, которое обычно задается структурой модели, и априорного распределения, которое выражает наши представления об окружающем модель мире, получить апостериорное распределение, а затем, при желании, делать байесовские предсказания, интегрируя по всем возможным значениям параметров $\boldsymbol{\theta}$. С точки зрения нейронных сетей важно, что с помощью теоремы Байеса мы можем легко строить сложные вероятностные модели из простых, уточняя распределение скрытых параметров. Кроме того, выбирая априорное распределение, легко проводить регуляризацию. Более того, мы можем легко вводить модели с латентными (скрытыми) переменными и обучать их EM-алгоритмом или вариационными методами, как мы увидим в этой главе. Задача байесовской оценки непрерывных скрытых переменных — это что-то вроде задачи понижения размерности: надо хорошо восстановить латентные переменные, и они станут важными признаками, кратко описывающими каждый объект обучающей выборки.

Может показаться, что вся эта байесовская машинерия для нас не так уж важна, и можно обойтись максимальным правдоподобием. В главе 2 мы рассматривали пример подбрасывания монетки: конечно, если бросков монетки один-два, без байесовских методов получится ерунда, но после тысячи бросков уже не так важно, каким было априорное распределение. А когда мы говорим о глубоких нейронных сетях, наборы данных исчисляются даже не тысячами, а миллионами и миллиардами — сколько пикселей в ImageNet? Однако проблема в том, что роль априорного распределения определяется не общим числом точек N_{data}, а числом точек *на каждый свободный параметр* сети $N_{\text{data}}/N_{\text{model}}$. И в реальности оказывается, что как только N_{data} увеличивается, мы тут же хотим подтянуть к ней и N_{model}, чтобы

сделать модель выразительнее. Сверточные сети, которые обучаются на ImageNet, могут иметь десятки миллионов параметров. Так что на самом деле машинное обучение все время находится в «зоне актуальности» байесовских методов.

На этом месте достаточно подготовленный читатель может подумать, что дальше мы будем говорить о том, как добавить в нейронную сеть априорные распределения и как сделать байесовский вывод в нейронной сети. Это действительно заслуживающая внимания тема, и мы к ней вернемся в разделе 10.5. Байесовский подход на самом деле может дать возможность не только получить веса в локальном минимуме функции ошибки, но и оценить их дисперсию, а также и все апостериорное распределение $p(\boldsymbol{w} \mid D)$. Более того, в разделе 10.5 мы увидим, что такой подход дает совсем другой взгляд на дропаут, объяснив его с байесовской стороны.

Но по большей части далее мы будем говорить о совсем других вещах. Задача этой главы — постараться улучшить методы, использующиеся в байесовском выводе, особенно вариационные приближения к сложным апостериорным распределениям, с помощью «магии глубокого обучения». В качестве основных источников мы здесь рекомендуем исходную статью о байесовских автокодировщиках [280] и недавно появившееся подробное введение в тему [124].

Здесь появляется существенная разница в постановке задачи по сравнению со всем тем, что мы делали раньше. В машинном обучении модели часто имеют вероятностный смысл: мы обучаем не просто какую-то разделяющую поверхность, а распределение $p(\boldsymbol{x})$, которое показывает, насколько, по мнению обученной модели, вероятно появление той или иной точки в качестве точки данных. Уметь считать $p(\boldsymbol{x})$ вполне достаточно для того, чтобы решать, например, задачу классификации: чтобы распознавать рукописные цифры, мы обучаем десять моделей $p_0, p_1, \ldots, p_9$, а потом для классификации просто сравниваем $p_0(\boldsymbol{x}), p_1(\boldsymbol{x}), \ldots, p_9(\boldsymbol{x})$: какая вероятность больше, ту цифру мы и выдадим в качестве ответа.

А в разделе 8.2 мы говорили о том, что часто хочется не просто распознавать уже взятые откуда-то объекты, но и создавать их самостоятельно. Например, мы бы хотели научиться не только распознавать рукописные цифры на основе датасета MNIST, но и самостоятельно их генерировать, «писать от руки». Или генерировать картины в узнаваемом стиле известного художника, как в сервисах DeepArt и Prisma. Или... но здесь оставим читателю место, чтобы он мог дать волю собственной фантазии. Для этого нужно научиться не просто *вычислять* распределение $p(\boldsymbol{x})$, а *сэмплировать* из него, порождать точки по распределению $p(\boldsymbol{x})$.

Эта задача решалась, конечно, и в классическом байесовском обучении: есть, например, большая область методов Монте-Карло на основе марковских цепей (Markov chain Monte-Carlo), которые предназначены как раз для того, чтобы научиться сэмплировать из распределений, которые мы умеем только вычислять. Мы сейчас не будем углубляться в эту тему и отошлем читателя к соответствующим учебникам [44, 343, 381], а сами только скажем, что эти методы для очень сложных распределений (таких, как, например, распределение человеческих лиц в пространстве фотографий) работают крайне медленно или не работают вовсе.

В этой главе мы будем обучать модель сэмплировать (порождать точки) из сложного распределения с помощью нейронных сетей, которые будут здесь играть роль черного ящика для приближения какой угодно функции — в данном случае как раз плотности нужного распределения. Точнее говоря, нейронные сети будут обучаться преобразовывать какое-нибудь простое распределение, обычно нормальное со стандартными параметрами, в то, которое нам нужно. Если кратко описать предстоящую главу с математической точки зрения, мы рассмотрим способ строить приближения к сложным распределениям, в котором приближение берется из некоторого класса функций, а потом начнем в качестве этого класса функций подставлять нейронные сети. В результате получатся очень хорошо идейно мотивированные модели, которые многим исследователям представляются будущим глубокого обучения.

Сразу предупредим, что легко не будет — это, наверное, самая математически плотная глава во всей книге. С одной стороны, при первом чтении остаток главы, наверное, можно пропустить, да и на практике никто вас пока что не заставит применять именно эти методы. Но с другой стороны, эту главу можно считать своеобразной лакмусовой бумажкой: если вы можете ее прочесть и понять, у вас не должно возникнуть проблем с тем, чтобы читать самые современные статьи о глубоком обучении, и основную цель, которую мы ставим в этой книге, можно будет считать достигнутой. Итак, в путь!

10.2. Алгоритм EM

> Бедный Карлсон! Ему не позавидуешь. Алгоритм заставит его съесть пятьсот плюшек. Все до единой! И он наверняка умрет от обжорства.
>
> *В. Паронджанов.*
> *Дружелюбные алгоритмы, понятные каждому*

Чтобы применить байесовские методы к нейронным сетям (точнее, как мы увидим ниже, наоборот — нейронные сети к байесовским методам), сначала придется разобраться в том, как эти самые байесовские методы вообще работают. Мы начнем с того, что дадим байесовское обоснование одного из главных методов классического обучения без учителя, алгоритма Expectation-Maximization, который обычно называют просто EM-алгоритмом.

Идея EM-алгоритма довольно проста. Представьте себе, к примеру, задачу *кластеризации* на обычной плоскости $\mathbb{R}^2$: у вас есть набор точек, и вы хотите разбить их на подмножества так, чтобы внутри каждого из них точки были «похожи» друг на друга, то есть находились бы близко друг от друга, а точки из разных подмножеств как можно более существенно различались бы, то есть были бы далеки на плоскости.

Задачу кластеризации, конечно, можно решать очень многими разными способами [5, 573], и EM не всегда будет лучшим (как минимум он довольно медленный по сравнению с другими), но этот метод применим и к массе других задач, а кластеризация — хороший способ его проиллюстрировать.

Чтобы применить EM к кластеризации, нужно представить себе *все* данные о решении задачи кластеризации, которые могли бы у нас быть. В полном наборе данных у каждой точки будет написано, какие у нее координаты (это мы знали и раньше) и какому кластеру она должна принадлежать — а вот это как раз та информация, которую мы должны получить. В этом представлении один тестовый пример — это пара $(\boldsymbol{x}_i, z_i)$, где z_i показывает, какому кластеру принадлежит эта точка, и мы не знаем значений z_i.

Заметьте разницу: в «обычной» модели тоже всегда есть веса или параметры, которые мы пытаемся обучать, но их, как правило, относительно мало, меньше, чем данных, а тут мы видим, что на каждую точку в данных приходится целая своя переменная. Именно в таких ситуациях, когда в имеющихся данных некоторые переменные нам известны, а некоторые отсутствуют, и пригождается обычно EM-алгоритм.

Далее, чтобы применить EM-алгоритм, нужно сделать какие-то вероятностные предположения о том, как были порождены эти самые данные со скрытыми переменными. В случае кластеризации, проще говоря, нужно сделать предположения о форме кластеров. Давайте рассмотрим самый простой случай, когда точки лежат даже не на плоскости, а на прямой, и мы предполагаем, что они порождены двумя кластерами в виде нормальных распределений; более того, давайте предполагать, что дисперсия у этих распределений одинакова и равна σ: $x_1 \sim \mathcal{N}(x_1; \mu_1, \sigma^2)$ в первом кластере, $x_2 \sim \mathcal{N}(x_2; \mu_1, \sigma^2)$ во втором кластере.

А это значит, что распределение точек из обоих кластеров вместе представляет собой *смесь* двух нормальных распределений:

$$x \sim \alpha\mathcal{N}(x; \mu_1, \sigma^2) + (1 - \alpha)\mathcal{N}(x; \mu_2, \sigma^2),$$

где α показывает сравнительный вес кластеров друг относительно друга, то, насколько больше точек попадает в один кластер, чем в другой. То есть, математически говоря, наша задача состоит в том, чтобы найти параметры максимального правдоподобия (α, μ_1, μ_2) у этой смеси распределений.

Заметьте, что никаких скрытых переменных z пока что в модели нет. И мы сейчас можем записать эту задачу поиска параметров максимального правдоподобия в виде обычной задачи оптимизации:

$$\alpha, \mu_1, \mu_2 = \arg\max_{\alpha,\mu_1,\mu_2} \prod_{i=1}^{N} \Big(\alpha\mathcal{N}(x_i; \mu_1, \sigma^2) + (1 - \alpha)\mathcal{N}(x_i; \mu_2, \sigma^2)\Big),$$

где $x_1, \ldots, x_N$ — это точки из имеющегося набора данных.

Проблема здесь состоит в том, что эта функция слишком сложна; посмотрите — это огромное произведение сумм, и если раскрыть в нем скобки, слагаемых будет экспоненциально много. А если перейти к логарифмам, то мы получим большую сумму логарифмов сумм, что тоже не очень помогает. Как все это оптимизировать — непонятно.

Но если бы мы знали, какая точка из какого кластера была взята, C_1 или C_2, то легко подсчитали бы параметры максимального правдоподобия. Мы тогда просто разделили бы точки по соответствующим кластерам, и оценили среднее нормального распределения в каждом и долю каждого кластера:

$$\hat{\alpha} = \frac{|C_1|}{N}, \quad \hat{\mu}_1 = \frac{1}{|C_1|}\sum_{x_i \in C_1} x_i, \quad \hat{\mu}_2 = \frac{1}{|C_2|}\sum_{x_i \in C_2} x_i.$$

Или формально — давайте введем переменную z_i, которая равна нулю, если точка была взята из C_1, и единице, если из C_2. Тогда наш вероятностный процесс, порождающий точки, становится двухступенчатым, разбивается на две независимые части[1]: сначала мы случайно выбираем переменные z_i с вероятностью α, то есть общим правдоподобием:

$$p(Z \mid \alpha) = \prod_{i=1}^{N} \alpha^{z_i}(1-\alpha)^{1-z_i},$$

а потом с уже известными z_i набрасываем точки из соответствующих кластеров:

$$p(X \mid Z, \mu_1, \mu_2) = \prod_{i=1}^{N} \mathcal{N}(x; \mu_1, \sigma^2)^{z_i} \mathcal{N}(x; \mu_2, \sigma^2)^{1-z_i}.$$

Иначе говоря, мы теперь решаем такую задачу оптимизации (давайте для удобства сразу перейдем к логарифмам):

$$\alpha = \arg\max_{\alpha} \sum_{i=1}^{N} \left(z_i \ln \alpha + (1 - z_i) \ln (1 - \alpha) \right),$$

$$\mu_1, \mu_2 = \arg\max_{\mu_1, \mu_2} \sum_{i=1}^{N} \Big(z_i p(x; \mu_1, \sigma^2) + (1 - z_i)\, p(x; \mu_2, \sigma^2) \Big),$$

где $p(x; \mu, \sigma)$ — плотность нормального распределения, $p(x; \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}(x-\mu)^2}$.

[1] Здесь можно было бы нарисовать графическую вероятностную модель, поговорить о d-разделимости, подробно обсудить, как условия d-разделимости связаны с условной независимостью в вероятностных моделях... но это все-таки увело бы нас очень далеко от предмета книги, поэтому мы просто оставим здесь несколько ссылок [44, 381, 594] и продолжим «на пальцах».

Обратите внимание, что теперь эта задача оптимизации распадается на три совершенно независимые и по отдельности довольно простые задачи:

- найти α как $\arg\max_\alpha \sum_{i=1}^{N} (z_i \ln \alpha + (1 - z_i) \ln (1 - \alpha))$; это всего лишь поиск параметра максимального правдоподобия для подбрасывания монетки, и найти оптимальное α легко: $\hat{\alpha} = \frac{|i:z_i=1|}{N} = \frac{|C_1|}{N}$;
- найти μ_1 как $\arg\max_{\mu_1} \sum_{i:z_i=1} p\left(x; \mu_1, \sigma^2\right)$ (обратите внимание, что в сумме только те слагаемые, для которых $z_i = 1$, зависят от μ_1, и — чудесное совпадение — они-то как раз и не зависят от μ_2); это всего лишь поиск параметра максимального правдоподобия для нормального распределения, и тут тоже μ_1 легко найти просто как среднее точек в первом кластере: $\hat{\mu}_1 = \frac{1}{|C_1|} \sum_{x_i \in C_1} x_i$;
- и, аналогично, зависящие от μ_2 слагаемые теперь не содержат μ_1, мы решаем задачу $\arg\max_{\mu_1} \sum_{i:z_i=0} p\left(x; \mu_2, \sigma^2\right)$ и находим $\hat{\mu}_2 = \frac{1}{|C_2|} \sum_{x_i \in C_2} x_i$.

На данный момент у нас получилось, что если z_i известны, то задача сразу разбивается на простые подзадачи и решается легко и непринужденно. Но что делать в реальной ситуации, когда z_i неизвестны?

Идея алгоритма EM состоит в том, чтобы *притвориться*, будто бы мы знаем z_i, а затем попеременно уточнять то скрытые переменные, то параметры модели. EM означает Expectation-Maximization, и в полном соответствии с названием алгоритм EM повторяет в цикле два шага:

- *E-шаг*: для известных параметров модели α, μ_1, μ_2 подсчитать ожидания скрытых переменных z_i; для кластеризации мы этого еще не делали, но это тоже несложно — если все параметры известны, то чтобы подсчитать, с какой вероятностью точка принадлежит тому или иному кластеру, нужно просто найти плотности распределений одного и другого кластера в этой точке:

$$\mathbb{E}\left[z_{ij}\right] = \frac{p(x = x_i | \mu = \mu_j)}{p\left(x = x_i \mid |\mu = \mu_1\right) + p\left(x = x_i \mid |\mu = \mu_2\right)} = \\ = \frac{e^{-\frac{1}{2\sigma^2}(x_i - \mu_j)^2}}{e^{-\frac{1}{2\sigma^2}(x_i - \mu_1)^2} + e^{-\frac{1}{2\sigma^2}(x_i - \mu_2)^2}};$$

- *M-шаг*: зафиксировать $z_i = \mathbb{E}\left[z_i\right]$ и пересчитать параметры модели по уже известным формулам:

$$\hat{\alpha} = \frac{|C_1|}{N}, \quad \hat{\mu}_1 = \frac{1}{|C_1|} \sum_{x_i \in C_1} x_i, \quad \hat{\mu}_2 = \frac{1}{|C_2|} \sum_{x_i \in C_2} x_i.$$

Получился итеративный алгоритм, который сначала инициализирует параметры модели случайными значениями (в случае кластеризации разумно выбрать две случайные точки из данных и объявить их центрами кластеров), а затем постепенно уточняет и параметры, и оценки скрытых переменных, пока не сойдется.

Итак, мы поняли, как работает EM-алгоритм, на примере задачи кластеризации. Мы получили некий метод, который выглядит разумно и дает хорошие результаты (по крайней мере на этой задаче), но звучит он пока что довольно подозрительно: с какой стати такое «вытягивание самого себя за уши» из случайных значений латентных переменных или параметров модели должно приводить к успеху? Давайте теперь разберемся не только в том, *как* работает EM в общем виде, но и *почему* он работает.

Для этого придется перейти к той самой обещанной «сложной математике». Сейчас мы дадим формальное обоснование алгоритма EM в общем виде; формул будет много, и для читателя, который раньше их не видел, у нас есть такой совет: постарайтесь «протащить» через эти формулы интуицию из примера, который мы не случайно так подробно рассмотрели. Пример с кластеризацией (то есть со смесью нормальных распределений) достаточно полно отражает все особенности задачи, и на нем можно понять и то, что происходит ниже.

Как и в примере с кластеризацией, мы будем решать задачу поиска параметров максимального правдоподобия $\boldsymbol{\theta}$ по данным $\mathcal{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$; правдоподобие можно записать так:

$$L(\boldsymbol{\theta} \mid \mathcal{X}) = p(\mathcal{X} \mid \boldsymbol{\theta}) = \prod p(\boldsymbol{x}_i \mid \boldsymbol{\theta}),$$

а максимизировать, понятное дело, можно не само правдоподобие, а его логарифм $\ell(\boldsymbol{\theta} \mid \mathcal{X}) = \log L(\boldsymbol{\theta} \mid \mathcal{X})$. EM может помочь, если этот максимум трудно найти аналитически.

Давайте предположим, что в данных есть *латентные переменные* (скрытые переменные, скрытые компоненты), причем модель устроена так, что если бы мы их знали, задача стала бы проще, а то и допускала бы аналитическое решение. На самом деле, совершенно не обязательно, чтобы эти переменные имели какой-то физический смысл, может быть, так просто удобнее, но физический смысл (такой, как номер кластера в предыдущем примере), конечно, обычно помогает их придумать. Например, представьте себе задачу порождения картинок с изображением цифр (мы рассмотрим этот пример в разделе 10.4). В этой задаче нам нужно породить всю картинку целиком так, чтобы она была «посвящена» одной и той же цифре, ведь половинка двойки плюс половинка восьмерки дадут скорее что-то похожее на рукописный твердый знак, чем на конкретную цифру. И это удобнее всего выразить именно так: сначала мы порождаем скрытую переменную z, которая соответствует тому, какую цифру хочется породить, а потом все распределение видимых данных $p(\boldsymbol{x} \mid z)$ будет уже зависеть от значения z [124].

В любом случае совместная плотность набора данных $\mathcal{Y} = (\mathcal{X}, \mathcal{Z})$ такова:

$$p(\boldsymbol{y} \mid \boldsymbol{\theta}) = p(\boldsymbol{x}, \boldsymbol{z} \mid \boldsymbol{\theta}) = p(\boldsymbol{z} \mid x, \boldsymbol{\theta}) p(x \mid \boldsymbol{\theta}),$$

и полное правдоподобие данных теперь составляет $L(\boldsymbol{\theta} \mid \mathcal{Z}) = p(\mathcal{X}, \mathcal{Y} \mid \boldsymbol{\theta})$.

Это случайная величина (потому что $\mathcal{Y}$ неизвестно), а настоящее правдоподобие можно вычислить как ожидание полного правдоподобия по скрытым переменным $\mathcal{Y}$:

$$L(\boldsymbol{\theta}) = \mathbb{E}_{\mathcal{Y}}\left[p(\mathcal{X}, \mathcal{Y} \mid \boldsymbol{\theta}) \mid \mathcal{X}, \boldsymbol{\theta}\right].$$

Теперь E-шаг алгоритма EM вычисляет условное ожидание (логарифма) полного правдоподобия при условии $\mathcal{X}$ и текущих оценок параметров $\boldsymbol{\theta}_n$:

$$Q(\boldsymbol{\theta}, \boldsymbol{\theta}_n) = \mathbb{E}\left[\log p(\mathcal{X}, \mathcal{Y} \mid \boldsymbol{\theta}) \mid \mathcal{X}, \boldsymbol{\theta}_n\right].$$

Здесь $\boldsymbol{\theta}_n$ — текущие оценки параметров модели, а $\boldsymbol{\theta}$ — неизвестные значения, которые мы хотим получить в конечном счете; это значит, что $Q(\boldsymbol{\theta}, \boldsymbol{\theta}_n)$ — это функция от $\boldsymbol{\theta}$. Условное ожидание можно переписать так:

$$E\left[\log p(\mathcal{X}, \mathcal{Y} \mid \boldsymbol{\theta}) \mid \mathcal{X}, \boldsymbol{\theta}_n\right] = \int_{\boldsymbol{z}} \log p(\mathcal{X}, \boldsymbol{z} \mid \boldsymbol{\theta}) p(\boldsymbol{z} \mid \mathcal{X}, \boldsymbol{\theta}_n) \mathrm{d}\boldsymbol{z},$$

где $p(\boldsymbol{z} \mid \mathcal{X}, \boldsymbol{\theta}_n)$ — маргинальное распределение скрытых компонентов данных.

EM лучше всего применять, когда это выражение легко подсчитать, может быть, даже можно подсчитать аналитически. Вместо $p(\boldsymbol{z} \mid \mathcal{X}, \boldsymbol{\theta}_n)$ можно подставить $p(\boldsymbol{z}, \mathcal{X} \mid \boldsymbol{\theta}_n) = p(\boldsymbol{z} \mid \mathcal{X}, \boldsymbol{\theta}_n) p(\mathcal{X} \mid \boldsymbol{\theta}_n)$, от этого ничего не изменится.

В итоге после E-шага алгоритма EM мы получаем функцию $Q(\boldsymbol{\theta}, \boldsymbol{\theta}_n)$. А на M-шаге максимизируем функцию Q по параметрам модели, получая новую оценку $\boldsymbol{\theta}_{n+1}$:

$$\boldsymbol{\theta}_{n+1} = \arg\max_{\boldsymbol{\theta}} Q(\boldsymbol{\theta}, \boldsymbol{\theta}_n).$$

Затем эта процедура повторяется до сходимости.

Осталось понять, что же, собственно, означает эта таинственная функция $Q(\boldsymbol{\theta}, \boldsymbol{\theta}_n)$ и почему все это работает.

Мы хотели перейти от $\boldsymbol{\theta}_n$ к $\boldsymbol{\theta}$, для которого $\ell(\boldsymbol{\theta}) > \ell(\boldsymbol{\theta}_n)$. Давайте оценим разность этих двух величин. Дальше мы будем вести достаточно длинную цепочку равенств, которые будем по ходу комментировать:

$$\ell(\boldsymbol{\theta}) - \ell(\boldsymbol{\theta}_n) = \ldots$$

(по определению и одной из предыдущих формул)

$$\ldots = \log\left(\int_{\boldsymbol{z}} p(\mathcal{X} \mid \boldsymbol{z}, \boldsymbol{\theta}) p(\boldsymbol{z} \mid \boldsymbol{\theta}) \mathrm{d}\boldsymbol{z}\right) - \log p(\mathcal{X} \mid \boldsymbol{\theta}_n) = \ldots$$

(домножим и разделим на $p(\boldsymbol{z} \mid \mathcal{X}, \boldsymbol{\theta}_n)$)

$$\ldots = \log\left(\int_{\boldsymbol{z}} p(\boldsymbol{z} \mid \mathcal{X}, \boldsymbol{\theta}_n) \frac{p(\mathcal{X} \mid \boldsymbol{z}, \boldsymbol{\theta}) p(\boldsymbol{z} \mid \boldsymbol{\theta})}{p(\boldsymbol{z} \mid \mathcal{X}, \boldsymbol{\theta}_n)} \mathrm{d}\boldsymbol{z}\right) - \log p(\mathcal{X} \mid \boldsymbol{\theta}_n) \geq \ldots$$

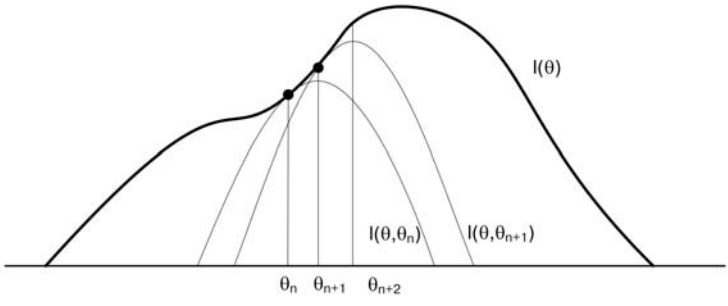

Рис. 10.1. Алгоритмы миноризации-максимизации

(по неравенству Йенсена: логарифм — выпуклая функция, а математическое ожидание — линейное преобразование, так что $\log \mathbb{E}_{p(x)} f(x) \geq \mathbb{E}_{p(x)} \log f(x)$, а у нас берется ожидание по распределению $p(\boldsymbol{z} \mid \mathcal{X},\boldsymbol{\theta}_n)$)

$$\ldots \geq \int_{\boldsymbol{z}} p(\boldsymbol{z} \mid \mathcal{X},\boldsymbol{\theta}_n) \log \left(\frac{p(\mathcal{X} \mid \boldsymbol{z},\boldsymbol{\theta}) p(\boldsymbol{z} \mid \boldsymbol{\theta})}{p(\boldsymbol{z} \mid \mathcal{X},\boldsymbol{\theta}_n)} \right) \mathrm{d}\boldsymbol{z} - \log p(\mathcal{X} \mid \boldsymbol{\theta}_n) = \ldots$$

(занесем теперь $p(\mathcal{X} \mid \boldsymbol{\theta}_n)$ под интеграл и под логарифм — в нем $\boldsymbol{z}$ не участвует, так что для математического ожидания по $\boldsymbol{z}$ это просто константа)

$$= \int_{\boldsymbol{z}} p(\boldsymbol{z} \mid \mathcal{X},\boldsymbol{\theta}_n) \log \left(\frac{p(\mathcal{X} \mid \boldsymbol{z},\boldsymbol{\theta}) p(\boldsymbol{z} \mid \boldsymbol{\theta})}{p(\mathcal{X} \mid \boldsymbol{\theta}_n) p(\boldsymbol{z} \mid \mathcal{X},\boldsymbol{\theta}_n)} \right) \mathrm{d}\boldsymbol{z}.$$

Итак, мы получили, что

$$\ell(\boldsymbol{\theta}) \geq l(\boldsymbol{\theta},\boldsymbol{\theta}_n) = \ell(\boldsymbol{\theta}_n) + \int_{\boldsymbol{z}} p(\boldsymbol{z} \mid \mathcal{X},\boldsymbol{\theta}_n) \log \left(\frac{p(\mathcal{X} \mid \boldsymbol{z},\boldsymbol{\theta}) p(\boldsymbol{z} \mid \boldsymbol{\theta})}{p(\mathcal{X} \mid \boldsymbol{\theta}_n) p(\boldsymbol{z} \mid \mathcal{X},\boldsymbol{\theta}_n)} \right) \mathrm{d}\boldsymbol{z}.$$

Кроме того, легко доказать, что $l(\boldsymbol{\theta}_n,\boldsymbol{\theta}_n) = \ell(\boldsymbol{\theta}_n)$. Иначе говоря, мы нашли нижнюю оценку для функции $\ell(\boldsymbol{\theta})$, которая касается, оказывается равной $\ell(\boldsymbol{\theta})$ в точке $\boldsymbol{\theta}_n$. Это значит, что в рамках EM-алгоритма мы сначала нашли нижнюю оценку для функции правдоподобия, которая касается самого правдоподобия в текущей точке, а потом сместились туда, где она максимальна; естественно, при этом сама функция правдоподобия тоже увеличится, ей просто деваться некуда (рис. 10.1). Такая общая схема оптимизации, при которой максимизируют нижнюю оценку на оптимизируемую функцию, иногда еще называется *MM-алгоритмом*, от слов Minorization-Maximization.

Осталось только понять, что максимизировать можно Q. И снова придется вытерпеть цепочку равенств:

$$\boldsymbol{\theta}_{n+1} = \arg\max_{\boldsymbol{\theta}} l(\boldsymbol{\theta},\boldsymbol{\theta}_n) = \ldots$$

(перепишем по формуле выше)

$$= \ldots \arg\max_{\boldsymbol{\theta}} \left\{ \ell(\boldsymbol{\theta}_n) + \int_{\boldsymbol{z}} p(\boldsymbol{z} \mid \mathcal{X}, \boldsymbol{\theta}_n) \log \left(\frac{p(\mathcal{X} \mid \boldsymbol{z}, \boldsymbol{\theta}) p(\boldsymbol{z} \mid \boldsymbol{\theta})}{p(\mathcal{X} \mid \boldsymbol{\theta}_n) p(\boldsymbol{z} \mid \mathcal{X}, \boldsymbol{\theta}_n)} \right) \mathrm{d}\boldsymbol{z} \right\} = \ldots$$

(выбросим из-под arg max все, что не зависит от $\boldsymbol{\theta}$ — от этого точка, где достигается максимум, не изменится)

$$= \ldots \arg\max_{\boldsymbol{\theta}} \left\{ \int_{\boldsymbol{z}} p(\boldsymbol{z} \mid \mathcal{X}, \boldsymbol{\theta}_n) \log \left(p(\mathcal{X} \mid \boldsymbol{z}, \boldsymbol{\theta}) p(\boldsymbol{z} \mid \boldsymbol{\theta}) \right) \mathrm{d}\boldsymbol{z} \right\} = \ldots$$

(свернем опять произведение $p(x \mid \boldsymbol{z})p(\boldsymbol{z})$ в совместную вероятность $p(x, \boldsymbol{z})$)

$$= \ldots \arg\max_{\boldsymbol{\theta}} \left\{ \int_{\boldsymbol{z}} p(\boldsymbol{z} \mid \mathcal{X}, \boldsymbol{\theta}_n) \log p(\mathcal{X}, \boldsymbol{z} \mid \boldsymbol{\theta}) \mathrm{d}\boldsymbol{z} \right\} = \arg\max_{\boldsymbol{\theta}} \left\{ Q(\boldsymbol{\theta}, \boldsymbol{\theta}_n) \right\}.$$

Вот и получился EM-алгоритм.

Заметим, что для того, чтобы EM-алгоритм сработал, в принципе достаточно просто находить $\boldsymbol{\theta}_{n+1}$, для которого $Q(\boldsymbol{\theta}_{n+1}, \boldsymbol{\theta}_n) > Q(\boldsymbol{\theta}_n, \boldsymbol{\theta}_n)$: чтобы $\ell(\boldsymbol{\theta})$ увеличивалось, не обязательно находить именно максимум ее нижней оценки, достаточно сместиться туда, где нижняя оценка просто хоть на сколько-нибудь увеличится. Такая схема называется *обобщенным EM-алгоритмом* (Generalized EM).

10.3. Вариационные приближения

> Бах решил, что тут, пожалуй, лучше всего подойдут вариации, хотя до сих пор считал, что это дело неблагодарное... Тем не менее, эти вариации, как и все, что он создавал в то время, получились у него великолепными... Граф называл этот цикл своими вариациями. Он никак не мог ими насладиться, и долго еще, как только у него начиналась бессонница, он, бывало, говорил: «Любезный Гольдберг, сыграй-ка мне какую-нибудь из моих вариаций».
>
> *И. Форкель. О жизни, искусстве и о произведениях Иоганна Себастьяна Баха*

Следующая важная идея байесовского вывода, которую нам нужно осознать и для которой, собственно, и предназначены описанные в этой главе методы, — это *вариационные приближения*. Их рассмотрение будет очень похоже на то, что мы говорили о EM-алгоритме, и мы сможем еще раз обсудить всю эту ситуацию со скрытыми параметрами с немножко другого угла зрения.

Мы по-прежнему находимся в той же общей ситуации, что и в предыдущем разделе, где нам помогал EM-алгоритм: предположим, что набор данных $X = \{\boldsymbol{x}_i\}_{i=1}^N$ был порожден двухступенчатым процессом с параметрами модели $\boldsymbol{\theta}$: сначала из

какого-то априорного распределения $p(\boldsymbol{z} \mid \boldsymbol{\theta})$ породили вектор скрытых переменных $\boldsymbol{z}$, а затем из условного распределения $p(\boldsymbol{x} \mid \boldsymbol{z},\boldsymbol{\theta})$ породили собственно точку $\boldsymbol{x}$. Мы знаем распределения $p(\boldsymbol{z} \mid \boldsymbol{\theta})$ и $p(\boldsymbol{x} \mid \boldsymbol{z},\boldsymbol{\theta})$, но распределение $p(\boldsymbol{x} \mid \boldsymbol{\theta})$:

$$p(\boldsymbol{x} \mid \boldsymbol{\theta}) = \int_Z p(\boldsymbol{x} \mid \boldsymbol{z},\boldsymbol{\theta})p(\boldsymbol{z} \mid \boldsymbol{\theta})\mathrm{d}\boldsymbol{z}$$

для нас слишком сложное, мы не можем его подсчитать напрямую.

EM-алгоритм применялся в ситуациях, когда мы можем либо аналитически, либо численно подсчитать распределение скрытых переменных при фиксированных параметрах $p(\boldsymbol{z} \mid \boldsymbol{x},\boldsymbol{\theta})$; обычно его можно подсчитать по теореме Байеса:

$$p(\boldsymbol{z} \mid \boldsymbol{x},\boldsymbol{\theta}) = \frac{p(\boldsymbol{x} \mid \boldsymbol{z},\boldsymbol{\theta})p(\boldsymbol{z} \mid \boldsymbol{\theta})}{p(\boldsymbol{x} \mid \boldsymbol{\theta})}.$$

Тогда мы считаем это распределение на E-шаге EM-алгоритма, получаем оценки ожиданий $\boldsymbol{z}$, максимизируем целевую функцию на M-шаге по $\boldsymbol{\theta}$ с фиксированными оценками и так далее, как в предыдущем разделе.

Но что если даже распределение $p(\boldsymbol{z} \mid \boldsymbol{x},\boldsymbol{\theta})$ подсчитать не получается? Это не такой уж редкий случай: представим себе, например, что распределение $p(\boldsymbol{x} \mid \boldsymbol{z},\boldsymbol{\theta})$ задано привычной нам нейронной сетью. Даже обычная двухслойная нейронная сеть с нелинейным скрытым уровнем — это настолько сложное распределение, что просто взять и умножить его на априорное распределение $p(\boldsymbol{z} \mid \boldsymbol{\theta})$ не получится.

В математическом моделировании есть универсальный ответ на все подобные затруднения: если структура объекта слишком сложна, значит, нужно просто приблизить его с помощью какого-нибудь более простого объекта. Суть вариационного метода при этом состоит в том, что более простое приближение мы ищем среди некоторого класса функций (распределений), и пытаемся найти в этом более простом классе элемент, который наиболее похож на тот «сложный» элемент, который мы приближаем. Если этот класс будет задан параметрически, значит, у нас появятся новые *вариационные параметры*, по которым мы, будем надеяться, сможем оптимизировать приближение. Впрочем, одна из ключевых особенностей метода, который мы сейчас рассматриваем, как раз в том, что приближение мы ищем вовсе не параметрически.

Однако в целом смысл происходящего точно такой же: мы приближаем сложное распределение более простой формой, выбирая то распределение из более простого семейства, которое лучше всего приближает сложное распределение. «Похожесть» можно определить по расстоянию Кульбака — Лейблера:

$$\mathrm{KL}(p\|q) = \int p(x)\ln\frac{p(x)}{q(x)}\mathrm{d}x = -\int p(x)\ln\frac{q(x)}{p(x)}\mathrm{d}x.$$

Чтобы объяснить, как все это работает в реальности, вернемся сначала к одному из взглядов на EM-алгоритм: пусть X — наблюдаемые переменные, а Z —

латентные. EM-алгоритм предполагает, что мы умеем считать плотность совместного распределения $p(X, Z \mid \boldsymbol{\theta})$, а хотим максимизировать $p(X \mid \boldsymbol{\theta})$. Они связаны, например, так:

$$p(X, Z \mid \boldsymbol{\theta}) = p(X \mid \boldsymbol{\theta})p(Z \mid X,\boldsymbol{\theta}),$$

а значит,

$$\ln p(X \mid \boldsymbol{\theta}) = \ln p(X, Z \mid \boldsymbol{\theta}) - \ln p(Z \mid X,\boldsymbol{\theta}).$$

Рассмотрим новое распределение $q(Z)$, которое и будет служить тем самым приближением. Давайте просто механически добавим его в наши формулы. Рассмотрим формулу выше и возьмем в ее левой и правой частях ожидание по распределению $q(Z)$:

$$\int_Z q(Z) \ln p(X \mid \boldsymbol{\theta})\mathrm{d}Z = \int_Z q(Z) \ln p(X, Z \mid \boldsymbol{\theta})\mathrm{d}Z - \int_Z q(Z) \ln p(Z \mid X,\boldsymbol{\theta})\mathrm{d}Z.$$

Но $\ln p(X \mid \boldsymbol{\theta})$ от Z не зависит, так что ожидание слева равно самому $\ln p(X \mid \boldsymbol{\theta})$:

$$\ln p(X \mid \boldsymbol{\theta}) = \int_Z q(Z) \ln p(X, Z \mid \boldsymbol{\theta})dZ - \int_Z q(Z) \ln p(Z \mid X, \boldsymbol{\theta})dZ.$$

А теперь справа давайте под интегралом добавим и вычтем $q(Z) \ln q(Z)$ (да, это выглядит как непонятная магия — спокойствие, сейчас все прояснится); в результате у нас получится:

$$\begin{aligned}\ln p(X \mid \boldsymbol{\theta}) &= \int_Z q(Z) \left(\ln p(X, Z \mid \boldsymbol{\theta}) - \ln q(Z) + \ln q(Z) - \ln p(Z \mid X, \boldsymbol{\theta})\right) dZ \\ &= \int_Z q(Z) \ln \frac{p(X, Z \mid \boldsymbol{\theta})}{q(Z)} dZ - \int_Z q(Z) \ln \frac{p(Z \mid X, \boldsymbol{\theta})}{q(Z)} dZ = \\ &= \mathcal{L}(q, \boldsymbol{\theta}) + \mathrm{KL}(q \| p(Z \mid X, \boldsymbol{\theta})),\end{aligned}$$

где $\mathcal{L}(q, \boldsymbol{\theta}) = \int_Z q(Z) \ln \frac{p(X,Z|\boldsymbol{\theta})}{q(Z)} dZ$ — некий функционал уже от $p(X,Z \mid \boldsymbol{\theta})$, от совместного распределения, которое, по нашему предположению, достаточно простое.

Посмотрите, что у нас получилось: $\mathrm{KL}(q \| p(Z \mid X, \boldsymbol{\theta}))$ — это расстояние Кульбака — Лейблера; оно всегда неотрицательно, и равно нулю только если $q = p$. Поэтому $\mathcal{L}(q, \boldsymbol{\theta})$ — это нижняя оценка на $\ln p(X \mid \boldsymbol{\theta})$. И в EM-алгоритме, получается, мы для текущего $\boldsymbol{\theta}_n$:

- на E-шаге максимизируем $\mathcal{L}(q, \boldsymbol{\theta}_n)$ по q для фиксированного $\boldsymbol{\theta}_n$; максимум достигается, когда $\mathrm{KL}(q \| p(Z \mid X, \boldsymbol{\theta})) = 0$, то есть когда $q(Z) = p(Z \mid X, \boldsymbol{\theta}_n)$;
- на M-шаге фиксируем $q(Z)$ и максимизируем нижнюю оценку правдоподобия $\mathcal{L}(q, \boldsymbol{\theta})$ по $\boldsymbol{\theta}$, получая $\boldsymbol{\theta}_{n+1}$.

Нижняя оценка $\mathcal{L}(q, \boldsymbol{\theta})$ — это один из важнейших объектов в теории вариационных приближений. Она называется *вариационной нижней оценкой* (variational

lower bound, evidence lower bound, ELBO). А сами вариационные методы работают так: пусть мы хотим максимизировать $p(X)$ со скрытыми переменными Z, и $p(Z \mid X)$ — сложное распределение. Давайте искать приближение к нему в виде распределения более простой структуры $q(Z)$. Тогда по тем же соображениям

$$\ln p(X) = \mathcal{L}(q) + \mathrm{KL}(q \| p(Z \mid X)), \text{ где}$$

$$\mathcal{L}(q) = \int_Z q(Z) \ln \frac{p(X,Z)}{q(Z)} dZ, \quad \mathrm{KL}(q\|p) = -\int_Z q(Z) \ln \frac{p(Z \mid X)}{q(Z)} dZ.$$

И мы снова можем максимизировать вариационную нижнюю оценку $\mathcal{L}(q)$, приближая $q(Z)$ к $p(Z \mid X)$. Здесь получился редкий частный случай функциональной оптимизации: казалось бы, нам нужно было решить нереально сложную задачу, максимизировать $\mathcal{L}(q, \boldsymbol{\theta}_n)$ по функции, по распределению q. Но в результате нехитрых преобразований оказалось, что для этого достаточно всего лишь подсчитать $q(Z) = p(Z \mid X, \boldsymbol{\theta}_n)$, или хотя бы приблизить $q(Z) \approx p(Z \mid X, \boldsymbol{\theta}_n)$. Если бы мы брали $q(Z)$ из параметрического семейства, то поиск $q(Z)$ свелся бы к (возможно, приближенному) решению некоторой задачи оптимизации, но прелесть вариационных приближений в том, что часто параметрическое семейство и вовсе не нужно!

Рассмотрим сначала главный частный случай вариационных приближений — случай, когда q полностью факторизуется, раскладывается на множители, зависящие от одной переменной или непересекающихся подмножеств переменных:

$$q(Z) = \prod_{i=1}^{M} q_i(Z_i), \quad Z_i \cap Z_j = \emptyset \text{ для всех } i,j.$$

Ключевой момент здесь в том, что мы никак не ограничиваем q_i! Никаких предположений о форме распределений, никакого параметрического семейства. Однако теперь мы просто должны максимизировать $\mathcal{L}(q)$, сделав предположение о форме $q(Z)$: $q(Z) = \prod_{i=1}^{M} q_i(Z_i)$. Оставим то, что зависит от одного фактора q_j:

$$\begin{aligned}
\mathcal{L}(q) &= \int q \ln \frac{p(X,Z)}{q(Z)} dZ = \int \prod_{i=1}^{M} q_i \left(\ln p(X,Z) - \sum_i \ln q_i \right) dZ = \\
&= \int q_j \left[\int \ln p(X,Z) \prod_{i \neq j} q_i dZ_i \right] dZ_j - \int q_j \ln q_j dZ_j + \text{const} = \\
&= \int q_j \ln \tilde{p}(X, Z_j) dZ_j - \int q_j \ln q_j dZ_j + \text{const},
\end{aligned}$$

где $\ln \tilde{p}(X, Z_j) = \mathbb{E}_{i \neq j} [\ln p(X,Z)] + \text{const}$.

Как теперь максимизировать такую величину:

$$\mathcal{L}(q) = \int q_j \ln \tilde{p}(X, Z_j) dZ_j - \int q_j \ln q_j dZ_j + \text{const}$$

по q_j при фиксированных q_i, $i \neq j$? Эта формула – просто KL-расстояние между $q_j(Z_j)$ и $\tilde{p}(X, Z_j)$! Значит, можно брать q^* так, чтобы:

$$\ln q^*(Z_j) = \mathbb{E}_{i \neq j} [\ln p(X, Z)] + \text{const},$$

а эта задача обычно решается достаточно просто.

Давайте разберем конкретный пример: приблизим двумерное нормальное распределение произведением одномерных. Рассмотрим двумерный гауссиан:

$$p(\boldsymbol{z}) = \mathcal{N}(\boldsymbol{z} \mid \boldsymbol{\mu}, \Lambda^{-1}), \quad \boldsymbol{\mu} = \begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix}, \quad \Lambda = \begin{pmatrix} \lambda_{11} & \lambda_{12} \\ \lambda_{12} & \lambda_{22} \end{pmatrix},$$

$$p(\boldsymbol{z}) = \frac{1}{2\pi|\Lambda|} e^{-\frac{1}{2}(\boldsymbol{z}-\boldsymbol{\mu})^\top \Lambda (\boldsymbol{z}-\boldsymbol{\mu})}.$$

Это распределение не раскладывается на независимые множители, но давайте приблизим его распределением, которое раскладывается: $q(\boldsymbol{z}) = q_1(z_1)q_2(z_2)$.

Воспользуемся выведенной выше формулой $\ln q^*(Z_j) = \mathbb{E}_{i \neq j} [\ln p(X, Z)] + \text{const}$; когда мы подсчитываем $q_1(z_1)$, в const попадает все, что не зависит от z_1:

$$\begin{aligned} \ln q_1^*(z_1) = \mathbb{E}[\ln p(\boldsymbol{z})] + \text{const} = \mathbb{E}\left[-\frac{1}{2}(z-\mu)^\top \Lambda (z-\mu)\right] + \text{const} = \\ = \mathbb{E}\left[-\frac{1}{2}(z_1-\mu_1)^\top \lambda_{11}(z_1-\mu_1) - (z_1-\mu_1)^\top \lambda_{12}(z_2-\mu_2)\right] + \text{const} = \\ = -\frac{1}{2}\lambda_{11} z_1^2 + (\mu_1 \lambda_{11} - (\mathbb{E}[z_2] - \mu_2))\, z_1 + \text{const}. \end{aligned}$$

Смотрите — в качестве приближения у нас получилось нормальное распределение, причем получилось само по себе, без нашего участия! Выделяя полный квадрат:

$$q_1^*(z_1) = \mathcal{N}(z_1 \mid m_1, \lambda_{11}^{-1}), \quad \text{где} \quad m_1 = \mu_1 - \lambda_{11}^{-1}\lambda_{12}(\mathbb{E}[z_2] - \mu_2).$$

Аналогично,

$$q_2^*(z_2) = \mathcal{N}(z_2 \mid m_2, \lambda_{22}^{-1}), \quad \text{где} \quad m_2 = \mu_2 - \lambda_{22}^{-1}\lambda_{12}(\mathbb{E}[z_1] - \mu_1).$$

Здесь $\mathbb{E}[z_1] = m_1$, $\mathbb{E}[z_2] = m_2$, и нам надо просто решить систему; получится, естественно, $m_1 = \mu_1$, $m_2 = \mu_2$.

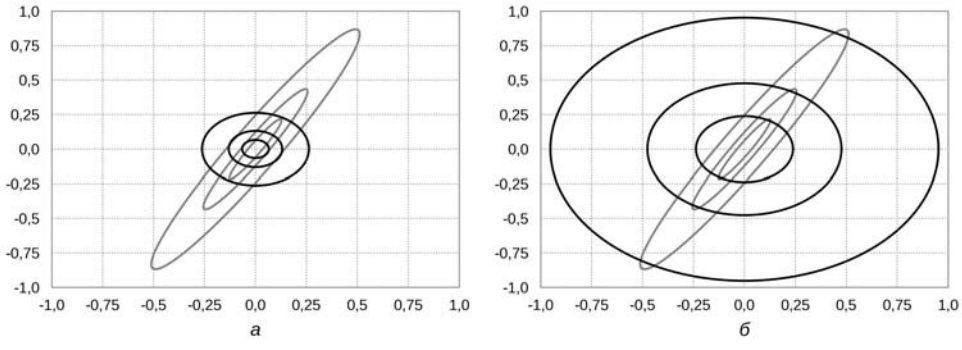

Рис. 10.2. Вариационные приближения к двумерному гауссиану

Сравним теперь это с обычным маргиналом (проекцией): на рис. 10.2, *а* мы изобразили полученное вариационное приближение, а на рис. 10.2, *б* — произведение проекций на оси X и Y. Мы видим здесь тот самый эффект, который мы обсуждали в разделе 8.5 (вспомните рис. 8.11) для разных вариантов расстояния Кульбака — Лейблера: в то время как произведение маргиналов дает широкое «накрывающее» распределение, вариационные приближения выбирают один конкретный пик, что для задач машинного обучения обычно предпочтительнее.

А важный для нашего дальнейшего рассмотрения конкретный пример — это *метод главных компонент* (principal components analysis, PCA). Это простейшая модель с непрерывными латентными переменными; возможно, вы слышали о ней в «обычных» курсах статистики или машинного обучения. Смысл PCA состоит в том, чтобы сократить размерность, потеряв как можно меньше информации о датасете; формально это значит, что мы пытаемся найти такое подпространство меньшей размерности, проекция на которую сохраняет как можно большую часть дисперсии исходного набора точек. Алгоритмически обычный PCA сводится к диагонализации эмпирической ковариационной матрицы датасета, то есть к подсчету ее собственных векторов.

Но сейчас нас интересует байесовский взгляд на то, что происходит в методе главных компонент. Итак, пусть наши исходные объекты находятся в пространстве размерности D, а латентные переменные — в пространстве размерности d: $X \in \mathbb{R}^D$, $Z \in \mathbb{R}^d$. Запишем распределения:

$$p(X,Z \mid \boldsymbol{\theta}) = \prod_{i=1}^{m} p(x_i,z_i \mid \boldsymbol{\theta}) = \prod_{i=1}^{m} p(x_i \mid z_i,\boldsymbol{\theta})p(z_i \mid \boldsymbol{\theta}).$$

Будем предполагать, что латентные переменные z_i берутся из нормального распределения вокруг нуля, а собственно наблюдаемые переменные получаются из них линейным преобразованием с нормально распределенным шумом:

$$p(X,Z \mid \boldsymbol{\theta}) = \prod_{i=1}^{m} \mathcal{N}(x_i \mid \mu + Wz_i, \sigma I)\mathcal{N}(z_i \mid 0, I).$$

Здесь матрица W размерности $D \times d$ и вектор $\mu \in \mathbb{R}^d$ — это и есть искомые веса. Задача та же: максимизировать $p(X \mid \boldsymbol{\theta})$ по $\boldsymbol{\theta}$. Ее можно, конечно, решить явно, как в исходном методе главных компонент, а можно и EM-алгоритмом: на E-шаге искать $p(z_i \mid x_i, \boldsymbol{\theta})$, а на M-шаге максимизировать $\mathbb{E}_z \log p(X,Z \mid \boldsymbol{\theta})$; в этом простом частном случае и E-, и M-шаг можно выразить явно, аналитически.

Казалось бы, странно: зачем запускать итеративный процесс, если можно просто явно все посчитать через собственные векторы? Оказывается, что иногда так действительно лучше: сложность метода главных компонент в явном виде составляет $O(nD^2 + D^3)$ (искать собственные векторы у больших матриц недешево!), а сложность одной итерации EM-алгоритма — всего лишь $O(nDd)$. При маленьких d это гораздо лучше, и итераций можно себе позволить довольно много.

Есть и другие преимущества. Например, что делать, если во входных данных, в X, попадаются пропущенные данные? PCA нужна правильная плотная матрица, строки или столбцы с неизвестными данными придется выкидывать, а в таком случае мы рискуем остаться без данных совершенно.

Или предположим, что часть скрытых переменных нам уже известна (обучение с частичным привлечением учителя): опять же, в явном алгоритме непонятно, что делать, а тут совершенно понятно, надо просто зафиксировать известные z_i и не пересчитывать их потом.

Еще более интересное байесовское обобщение идеи PCA — это смесь нескольких подпространств главных компонент (mixture of PCA). Идея такая: что если данные не укладываются вдоль одного линейного подпространства малой размерности, но укладываются вдоль нескольких? В базовом PCA совершенно непонятно, как ввести такое расширение. А при вероятностном подходе все получается само собой: мы добавляем скрытые переменные t_i, которые показывают, из какого именно линейного подпространства (из нескольких) выбирается точка, и общее совместное распределение теперь выглядит так:

$$p(X,Z,T \mid \boldsymbol{\theta}) = \prod_{i=1}^{m} p(x_i \mid t_i, z_i, \boldsymbol{\theta}) p(z_i \mid \boldsymbol{\theta}) p(t_i \mid \boldsymbol{\theta}).$$

Распределения $p(x_i \mid t_i, z_i, \boldsymbol{\theta})$ выглядят точно так же, как $p(x_i \mid z_i, \boldsymbol{\theta})$ выше, но параметры выбираются соответственно значению t_i. Теперь можно точно так же вести вывод EM-алгоритмом, но метод уже получается не совсем линейный!

Однако даже такие вариации метода главных компонент в любом случае предполагают линейность базового подпространства. Но в жизни поверхности часто нам достаются совершенно нелинейные: согласитесь, вряд ли «настоящие фотографии» образуют линейное подпространство в пространстве всех изображений. Для того, чтобы их выразить, нам и помогут нейронные сети...

10.4. Вариационный автокодировщик

> Здесь следует обратить внимание на то, что прежде всего я могу мыслить самого себя, это аподиктическое Ego, совсем иначе, в свободной вариации, и таким образом получить систему возможных вариаций себя самого, причем любая такая вариация уничтожается любой другой и тем Ego, которым я действительно являюсь. Это есть система априорной несовместимости.
>
> *Э. Гуссерль. Картезианские размышления*

Вариационный автокодировщик [124, 280] — это недавно появившийся вариант порождающих моделей (вспомните раздел 8.2), который основан на вариационных приближениях. И на самом деле основная цель процесса здесь будет ровно та же, что и в соперничающих автокодировщиках (AAE; см. раздел 8.5): мы хотим сделать так, чтобы распределение скрытых факторов в автокодировщике было похоже на какое-нибудь стандартное распределение (например, нормальное), что затем поможет нам сэмплировать подходящие скрытые факторы и разворачивать их в правдоподобные объекты декодирующей частью.

Однако теперь мы будем идти к этой цели совсем другим путем. Содержательно идея вариационного автокодировщика поразительно похожа на то, что мы делали выше для метода главных компонент. Давайте просто вместо линейной функции подставим нелинейную:

$$p(X,Z \mid \boldsymbol{\theta}) = \prod_{i=1}^{m} p(x_i,z_i \mid \boldsymbol{\theta}) = \prod_{i=1}^{m} p(x_i \mid z_i,\boldsymbol{\theta})p(z_i \mid \boldsymbol{\theta}) = \prod_{i=1}^{m}\prod_{j=1}^{D} \mathcal{N}(x_{ij} \mid \mu_j(z_i),\sigma_j(z_i))\mathcal{N}(z_i \mid 0, I).$$

Обратите внимание: это ровно то же самое, что в PCA. Точно так же мы предполагаем, что латентные переменные берутся из стандартного нормального распределения, но теперь функции $\mu_j(z_i)$ и $\sigma_j(z_i)$ могут быть какими угодно. Как можно задать «какие угодно» функции? Конечно же, нейронной сетью! У нас появляется нейронная сеть, которая берет на вход z и производит те самые μ и σ, а параметры $\boldsymbol{\theta}$ теперь становятся просто параметрами этой нейронной сети.

Конечно, явным образом задачу максимизации теперь решить не получится. Но и с EM-алгоритмом не все так просто: на E-шаге нужно рассчитать $p(z_i \mid x_i,\boldsymbol{\theta})$, и это аналитически сделать не получится:

$$p(z_i \mid x_i,\boldsymbol{\theta}) = \frac{p(x_i \mid z_i,\boldsymbol{\theta})p(z_i)}{\int p(x_i \mid z_i,\boldsymbol{\theta})p(z_i)},$$

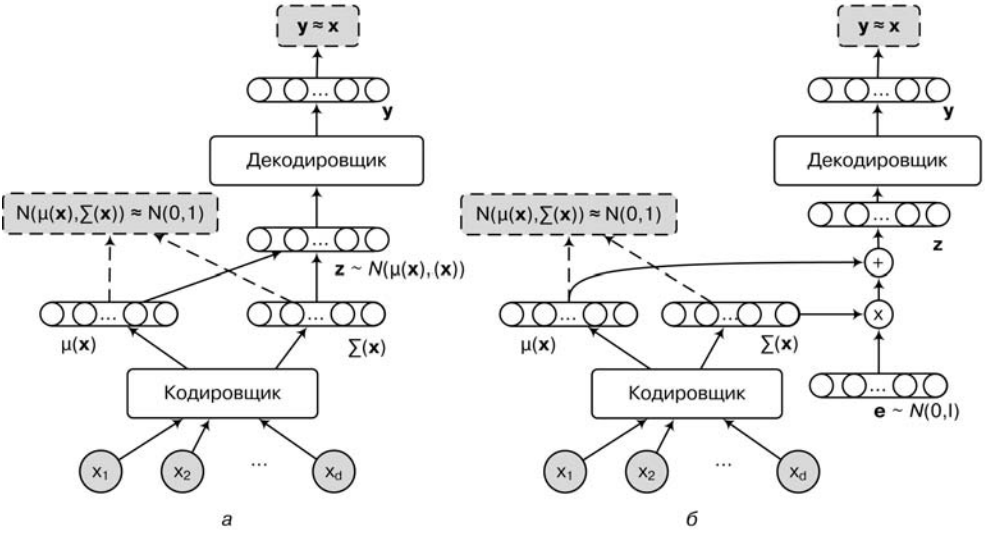

Рис. 10.3. Структура вариационного автокодировщика: *а* — основная идея; *б* — реализация с помощью репараметризации

и интеграл в знаменателе не берется. Что делать? Нужно вспомнить, что EM-алгоритм максимизирует нижнюю оценку на $q(Z)$. Теперь не получается оптимизировать функционально, по всем-всем-всем $q(Z)$, ну так давайте теперь просто параметризуем $q(Z)$ и будем решать задачу параметрической оптимизации.

С этим получается так: хочется взять $q(Z)$ побогаче (чтобы лучше приближать), но при этом надо все равно уметь оптимизировать. Чтобы этого добиться, давайте в качестве $q(Z)$... тоже возьмем нейронную сеть! С параметрами ϕ:

$$q(Z \mid X,\phi) = \prod_{i=1}^{N} p(z_i \mid x_i,\phi) = \prod_{i=1}^{N}\prod_{j=1}^{D} \mathcal{N}(z_i \mid \mu_j(x_i),\sigma_j(x_i)).$$

Итого в нашей конструкции есть сразу две нейронных сети: первая получает на вход d-мерный вектор скрытых переменных z и выдает вектор размерности D, объект из распределения над x, а вторая получает на вход D-мерный вектор тренировочного примера x и выдает вектор размерности $2d$ с параметрами распределений на скрытые переменные z. Глубокая сеть может аппроксимировать крайне сложное, нелинейное распределение, то есть для вариационного приближения мы берем очень богатый класс распределений. Структура вариационного автокодировщика проиллюстрирована на рис. 10.3, *а*.

Остается только один вопрос — как же это все вместе обучить? Начнем с того, что запишем вариационную нижнюю оценку:

$$\mathcal{L}(q(Z \mid X,\phi),\boldsymbol{\theta}) = \mathcal{L}(\phi,\boldsymbol{\theta}) = \sum_{i=1}^{n} \int q(z_i \mid x_i,\phi) \log \frac{p(x_i,z_i \mid \boldsymbol{\theta})}{q(z_i \mid x_i,\phi)} \mathrm{d}z_i.$$

Эту функцию надо максимизировать по ϕ и $\boldsymbol{\theta}$. Как это сделать, если мы даже не можем подсчитать интеграл, то есть не можем даже подсчитать саму функцию, которую мы оптимизируем?

Чтобы оптимизировать, надо уметь считать или градиент, или хотя бы стохастический градиент. Обычный градиент считать сложно, уж функцию точно надо уметь считать. А вот стохастический — пожалуйста. Например, для функции вида $f(x) = \frac{1}{n}\sum_{i=1}^{n} f(x)$ стохастический градиент можно подсчитать, выбирая слагаемое случайным образом; при этом все становится в n раз быстрее, а стохастический градиент все равно будет несмещенной оценкой настоящего, то есть в ожидании получится то, что надо. Точно так же, если верно следующее:

$$f(x) = \mathbb{E}_{p(y)}[h(x,y))] = \int h(x,y)p(y)\mathrm{d}y,$$

то можно просто породить точку из распределения $p(y)$ и подсчитать производную $\frac{\partial h(x,y_0)}{\partial x}$, где точка y_0 порождена из распределения $p(y)$. И получится опять несмещенная оценка:

$$\int p(y \frac{\partial h(x,y)}{\partial x} \mathrm{d}y = \frac{\partial}{\partial x} \int p(y)h(x,y)\mathrm{d}y = \frac{\partial f}{\partial x}.$$

Можно, конечно, и получше оценить, с меньшей дисперсией, если взять выборку из нескольких точек и в качестве стохастического градиента усреднить получающиеся в этих точках производные. Если мы видим интеграл, представляющий математическое ожидание какой-то функции, можно заменить это ожидание на случайную выборку с соответствующим распределением.

А теперь чуть сложнее: пусть распределение в ожидании зависит от x:

$$f(x) = \int h(x,y)p(y \mid x)\mathrm{d}y.$$

Дифференцируем как произведение, что ж поделать:

$$\frac{\partial f}{\partial x} = \frac{\partial}{\partial x} \int h(x,y)p(y \mid x)\mathrm{d}y = \int \left[\frac{\partial h(x,y)}{\partial x} p(y \mid x) + h(x,y)\frac{\partial p(y \mid x)}{\partial x}\right] \mathrm{d}y.$$

Первый интеграл можно теперь оценить точно так же, как выше — сгенерировать выборку, подставить частную производную, все хорошо. А вот со вторым теперь проблема: нигде нет математического ожидания.

Но здесь приходит на помощь так называемый log-derivative trick: если хочется представить $\frac{\partial p(y|x)}{\partial x}$ как $p(y \mid x)$, умноженное на что-то, то можно просто представить это в таком виде:

$$\frac{\partial p(y \mid x)}{\partial x} = p(y \mid x)\frac{\partial \log p(y \mid x)}{\partial x}.$$

Итого получается, что большой интеграл можно переписать так:

$$\int p(y \mid x)\left[\frac{\partial h(x,y)}{\partial x} + h(x,y)\frac{\partial \log p(y \mid x)}{\partial x}\right]\mathrm{d}y.$$

Подсчитать это честно, конечно, по-прежнему не получится, но нам же достаточно получить несмещенную оценку. Поэтому мы просто породим y из распределения $p(y \mid x)$ и подставим его туда:

$$\frac{\partial h(x,y_0)}{\partial x} + h(x,y_0)\frac{\partial \log p(y_0 \mid x)}{\partial x}, \quad \text{где} \quad y_0 \sim p(y \mid x).$$

Опять же, можно породить несколько точек и усреднить результаты. Заметьте, что мы ни разу нигде не брали интеграл, все, что нужно уметь делать — это порождать выборку из $p(y \mid x)$.

Теперь возвращаемся: нам нужно уметь максимизировать $\mathcal{L}(\phi,\boldsymbol{\theta})$ по ϕ и $\boldsymbol{\theta}$ по отдельности. Максимизация по $\boldsymbol{\theta}$ — это простой случай стохастического градиента, при фиксированных ϕ получится просто конкретное распределение $p(z_i \mid x_i,\phi)$, и из него можно породить выборку:

$$\frac{\partial}{\partial \boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta},\phi) = \frac{\partial}{\partial \boldsymbol{\theta}}\sum_{i=1}^{n}\int q(z_i \mid x_i,\phi)\log\frac{p(x_i,z_i \mid \boldsymbol{\theta})}{q(z_i \mid x_i,\phi)}\mathrm{d}z_i \approx n\frac{\partial}{\partial \boldsymbol{\theta}}\log p(x_i,z \mid \boldsymbol{\theta}).$$

Приближенное равенство в этой формуле объясняется так: мы равномерно взяли случайную точку x_i из выборки и породили для нее $z \sim p(z \mid x_i,\phi)$.

А стохастический градиент по ϕ получается так, как описано выше, с помощью log-derivative trick. Важная проблема здесь в том, что у таких стохастических градиентов получается очень большая дисперсия. Есть большая наука о том, как понизить дисперсию в таких вычислениях, но в общем случае это пока открытая проблема, получаются только эвристические методы.

Но авторы вариационного автокодировщика придумали еще один трюк: репараметризацию (reparametrization trick)! Идея очень простая: если у нас есть выражение вида $\int f(x)p(x \mid \boldsymbol{\theta})\mathrm{d}x$, и мы хотим его продифференцировать по $\boldsymbol{\theta}$, давайте устраним параметры из распределения заменой переменных:

$$\frac{\partial}{\partial \boldsymbol{\theta}}\int f(x)p(x \mid \boldsymbol{\theta})\mathrm{d}x = \quad [p(x \mid \boldsymbol{\theta})\mathrm{d}x = p(\epsilon)\mathrm{d}\epsilon,\ x = g(\epsilon,\boldsymbol{\theta})] \quad =$$
$$= \frac{\partial}{\partial \boldsymbol{\theta}}\int f(g(\epsilon,\boldsymbol{\theta}))p(\epsilon)\mathrm{d}\epsilon \approx \frac{\partial}{\partial \boldsymbol{\theta}}f(g(\epsilon_0,\boldsymbol{\theta})),$$

где $\epsilon_0 \sim p(\epsilon)$.

Такой трюк возможен не всегда, но в нашем случае, когда распределения параметризованы нейронными сетями, он всегда проходит. Поэтому можно получить выражение для градиента по ϕ:

$$\frac{\partial}{\partial\phi}\mathcal{L}(\boldsymbol{\theta},\phi) = \frac{\partial}{\partial\phi}\sum_{i=1}^{n}\int q(z_i \mid x_i,\phi)\log\frac{p(x_i,z_i \mid \boldsymbol{\theta})}{q(z_i \mid x_i,\phi)}\mathrm{d}z_i =$$

$$[q(z_i \mid x_i,\phi)\mathrm{d}z_i = g(x_i,\epsilon) = \mathcal{N}(\epsilon \mid 0,I)\mathrm{d}\epsilon, \quad z_i = \mu(x_i) + \sigma(x_i)\cdot\epsilon]$$

$$= \frac{\partial}{\partial\phi}\sum_{i=1}^{n} n\int g(x_i,\epsilon)\log\frac{p(x_i,g(x_i,\epsilon) \mid \boldsymbol{\theta})}{q(g(x_i,\epsilon) \mid x_i,\phi)}\mathrm{d}\epsilon \approx n\frac{\partial}{\partial\phi}\log\frac{p(x_i,g(x_i,\epsilon) \mid \boldsymbol{\theta})}{q(g(x_i,\epsilon) \mid x_i,\phi)},$$

и теперь последнее выражение уже вполне можно и подсчитать, и дифференцировать, потому что это просто выходы наших двух нейронных сетей. Получающаяся структура графа вычислений изображена на рис. 10.3, *б*. Любопытно, что репараметризацию придумали почти одновременно сразу три группы исследователей — очевидно, эта идея действительно назревала.

На самом деле, хотя технических трудностей было немало, концептуально все довольно просто: мы ввели латентные переменные, попытались приблизить распределение q с помощью нейронных сетей (двух: одна по z генерирует x с параметрами $\boldsymbol{\theta}$, другая по x генерирует z с параметрами ϕ), и тем самым свели задачу функциональной оптимизации к параметрической. Дальше дело техники: как подсчитать стохастический градиент; получилась крайне эффективно обучаемая конструкция, и практические результаты ее тоже впечатляют. В исходной работе приводятся результаты на MNIST и датасете из разных выражений лица одного и того же человека. Имея это внутреннее представление, полученное вариационным автокодировщиком, мы можем решить, например, такую задачу: попробовать сгенерировать другие объекты, похожие на данный. Похожие в том самом многообразии низкой размерности, многообразии скрытых факторов.

Следующий шаг: если мы уже знаем структуру датасета, можно усложнить априорное распределение. Например, для распознавания цифр можно просто заменить нормальное распределение латентных переменных на смесь десяти гауссианов, в итоге обучится по одному гауссиану на циферку. Можно получить глубокое обобщение фильтра Калмана, скрытой марковской модели. Все это делается ровно так же, стохастические градиенты берутся по одному и тому же принципу.

И, как уже повелось в нашей книге, давайте приведем практический пример — это упрощенный и адаптированный пример из кода Яна Хендрика Метцена, опубликованного в 2015 году [358]. Мы будем, как и во многих других примерах в этой книге, использовать стандартный набор данных MNIST с рукописными цифрами. Начнем с импорта TensorFlow и загрузки датасета:

```
import numpy as np
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
```

```
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
n_samples = mnist.train.num_examples
```

Затем проведем случайную инициализацию весов. Для этого будем использовать привычную нам инициализацию Ксавье (вспомните раздел 4.2). Для кодировщика и декодировщика возьмем по два уровня с 500 нейронами на каждом, размерность входа у MNIST равна $28 \times 28 = 784$, а размерность скрытого латентного пространства — 20:

```
def xavier_init(fan_in, fan_out, constant=1):
    low = -constant*np.sqrt(6.0/(fan_in + fan_out))
    high = constant*np.sqrt(6.0/(fan_in + fan_out))
    return tf.random_uniform((fan_in, fan_out),
                             minval=low, maxval=high,
                             dtype=tf.float32)

w, n_input, n_z = {}, 784, 20
n_hidden_recog_1, n_hidden_recog_2 = 500, 500
n_hidden_gener_1, n_hidden_gener_2 = 500, 500
w['w_recog'] = {
    'h1': tf.Variable(xavier_init(n_input, n_hidden_recog_1)),
    'h2': tf.Variable(xavier_init(n_hidden_recog_1, n_hidden_recog_2)),
    'out_mean': tf.Variable(xavier_init(n_hidden_recog_2, n_z)),
    'out_log_sigma': tf.Variable(xavier_init(n_hidden_recog_2, n_z))}
w['b_recog'] = {
    'b1': tf.Variable(tf.zeros([n_hidden_recog_1], dtype=tf.float32)),
    'b2': tf.Variable(tf.zeros([n_hidden_recog_2], dtype=tf.float32)),
    'out_mean': tf.Variable(tf.zeros([n_z], dtype=tf.float32)),
    'out_log_sigma': tf.Variable(tf.zeros([n_z], dtype=tf.float32))}
w['w_gener'] = {
    'h1': tf.Variable(xavier_init(n_z, n_hidden_gener_1)),
    'h2': tf.Variable(xavier_init(n_hidden_gener_1, n_hidden_gener_2)),
    'out_mean': tf.Variable(xavier_init(n_hidden_gener_2, n_input)),
    'out_log_sigma': tf.Variable(xavier_init(n_hidden_gener_2, n_input))}
w['b_gener'] = {
    'b1': tf.Variable(tf.zeros([n_hidden_gener_1], dtype=tf.float32)),
    'b2': tf.Variable(tf.zeros([n_hidden_gener_2], dtype=tf.float32)),
    'out_mean': tf.Variable(tf.zeros([n_input], dtype=tf.float32)),
    'out_log_sigma': tf.Variable(tf.zeros([n_input], dtype=tf.float32))}
```

Зададим скорость обучения, размер мини-батча и входную переменную:

```
l_rate=0.001
batch_size=100
x = tf.placeholder(tf.float32, [None, n_input])
```

Теперь все готово для того, чтобы реализовывать кодировщики и декодировщики. Кодировщик (распознаватель) берет входы и отображает их в нормальное распределение в пространстве скрытых признаков:

```
enc_layer_1 = tf.nn.softplus(tf.add(
    tf.matmul(x, w["w_recog"]['h1']), w["b_recog"]['b1']))
enc_layer_2 = tf.nn.softplus(tf.add(
    tf.matmul(enc_layer_1, w["w_recog"]['h2']), w["b_recog"]['b2']))
z_mean = tf.add( tf.matmul(
    enc_layer_2, w["w_recog"]['out_mean']),
    w["b_recog"]['out_mean'])
z_log_sigma_sq = tf.add( tf.matmul(
    enc_layer_2, w["w_recog"]['out_log_sigma']), w["b_recog"]['out_log_sigma'])
```

Затем мы набрасываем одну точку из нормального распределения в пространстве латентных признаков; чтобы породить точку из нормального распределения с заданными параметрами, достаточно набросить ее из стандартного нормального распределения (функцией `tf.random_normal`), а затем сдвинуть куда надо:

```
eps = tf.random_normal((batch_size, n_z), 0, 1, dtype=tf.float32)
z = tf.add(z_mean, tf.mul(tf.sqrt(tf.exp(z_log_sigma_sq)), eps))
```

Декодировщик (генератор) отображает точки в латентном пространстве в распределение Бернулли в пространстве данных:

```
dec_layer_1 = tf.nn.softplus(tf.add(
    tf.matmul(z, w["w_gener"]['h1']), w["b_gener"]['b1']))
dec_layer_2 = tf.nn.softplus(tf.add(
    tf.matmul(dec_layer_1, w["w_gener"]['h2']), w["b_gener"]['b2']))

x_reconstr_mean = tf.nn.sigmoid(tf.add(
    tf.matmul(dec_layer_2, w["w_gener"]['out_mean']),
    w["b_gener"]['out_mean']))
```

И теперь осталось только определить функцию потерь. Она состоит из двух частей. Первая — собственно ошибка восстановления, то есть минус логарифм правдоподобия входа в восстановленном распределении Бернулли (как обычно, мы добавляем маленькую константу, чтобы не пришлось считать логарифм нуля):

```
reconstr_loss = -tf.reduce_sum(x * tf.log(1e-10 + x_reconstr_mean) +
    (1-x) * tf.log(1e-10 + 1 - x_reconstr_mean), 1)
```

Вторая часть — это ошибка латентного распределения, которая определяется как расстояние Кульбака — Лейблера между распределением в пространстве латентных признаков, индуцированном кодировщиком на данных, и некоторым фиксированным априорным распределением, например обычным гауссианом:

```
latent_loss = -0.5 * tf.reduce_sum(1 + z_log_sigma_sq
    - tf.square(z_mean) - tf.exp(z_log_sigma_sq), 1)
cost = tf.reduce_mean(reconstr_loss + latent_loss)
```

Ошибка латентного распределения делает его максимально похожим на то самое априорное распределение (в нашем случае — стандартное нормальное распределение с нулевым средним и единичной дисперсией). Осталось только определить собственно оптимизатор; в этом качестве мы используем адаптивный алгоритм Adam (см. раздел 4.5):

```
optimizer = tf.train.AdamOptimizer(learning_rate=l_rate).minimize(cost)
```

Теперь можно обучать:

```
def train(sess, batch_size=100, training_epochs=10, display_step=5):
    for epoch in range(training_epochs):
        avg_cost = 0.
        total_batch = int(n_samples / batch_size)
        # Цикл по мини-батчам
        for i in range(total_batch):
            xs, _ = mnist.train.next_batch(batch_size)

            # Обучаем на текущем мини-батче
            _, c = sess.run((optimizer, cost), feed_dict={x: xs})
            # Compute average loss
            avg_cost += c / n_samples * batch_size

        # Каждые display_step шагов выводим текущую функцию потерь
        if epoch % display_step == 0:
            print("Epoch: %04d\tcost: %.9f" % (epoch+1, avg_cost))

init = tf.initialize_global_variables()
sess = tf.InteractiveSession()
sess.run(init)

train(sess, training_epochs=200, batch_size=batch_size)
```

После того как этот код отработает, мы получим автокодировщик, который понимает структуру возможных входов (рукописных цифр) и может как реконструировать цифры, так и порождать новые, выбирая скрытые факторы из нормального распределения. Чтобы посмотреть, как он реконструирует, достаточно подставить какие-нибудь новые рукописные цифры в качестве входов `x` и посмотреть, что из него получится в среднем реконструированного распределения `x_reconstr_mean`:

```
x_sample = mnist.test.next_batch(100)[0]
x_logits = sess.run(x_reconstr_mean_logits,
                    feed_dict={x: x_sample, eps:
                        np.random.normal(loc=0., scale=1., size=(batch_size, n_z))})
gen_logits = sess.run(x_reconstr_mean_logits,
                    feed_dict={z_mean:
                        np.random.normal(loc=0., scale=1., size=(batch_size, n_z))})
```

Результаты реконструкции показаны на рис. 10.4.

а *б* *в*

Рис. 10.4. Пример работы вариационного автокодировщика: *а* — исходные изображения; *б* — реконструированные; *в* — сэмплированные

А еще мы можем попробовать проверить, действительно ли в пространстве латентных признаков получается обещанное нормальное распределение. Давайте переобучим ту же модель, но с двумерным пространством латентных признаков, чтобы его можно было нарисовать и что-то понять в нем; для этого нужно просто в коде выше поменять `n_z=20` на `n_z=2`. Когда мы это сделаем, можно будет посмотреть, как цифры из тестового набора распределились в пространстве латентных признаков:

```
xs, ys = mnist.test.next_batch(5000)
z_mu = sess.run(z_mean, feed_dict={x: xs})
plt.figure(figsize=(8, 6))
plt.scatter(z_mu[:, 0], z_mu[:, 1], c=np.argmax(y_sample, 1))
plt.colorbar()
```

Результаты для трех разных этапов обучения показаны на рис. 10.5; на графиках точки, соответствующие разным цифрам, показаны разными маркерами. Действительно, общее распределение всех цифр во всех трех случаях выглядит весьма похожим на стандартное нормальное распределение. Но если до начала обучения все цифры в этом распределении перемешаны абсолютно случайным образом, то со временем в хаосе начинает появляться структура. К концу обучения уже довольно просто различить кластеры цифр в пространстве признаков, и при этом общая картинка все равно остается похожа на стандартный двумерный гауссиан! Учитывая, что мы взяли всего лишь двумерное пространство скрытых факторов, то есть существенно упростили модель, это отличный результат.

Можно пойти дальше: нарисовать конкретные примеры тех цифр, которые получаются из разных областей пространства латентных признаков. Для этого выберем точки в узлах решетки и построим соответствующую решетку из цифр:

```
nx = ny = 20
x_values = np.linspace(-3, 3, nx)
y_values = np.linspace(-3, 3, ny)

canvas = np.empty((28*ny, 28*nx))
for i, yi in enumerate(x_values):
    for j, xi in enumerate(y_values):
        z_mu = np.array([[xi, yi]])
```

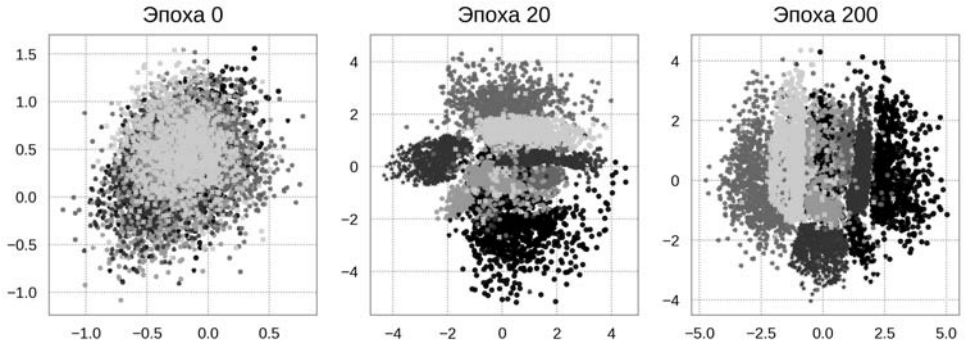

Рис. 10.5. 2D-представление распределения рукописных цифр в датасете MNIST

```
        x_mean = sess.run(x_reconstr_mean, feed_dict={z: z_mu})
        canvas[(nx-i-1)*28:(nx-i)*28, j*28:(j+1)*28] = x_mean[0].reshape(28, 28)

plt.figure(figsize=(8, 10))
Xi, Yi = np.meshgrid(x_values, y_values)
plt.imshow(canvas, origin="upper")
plt.tight_layout()
```

Результат показан на рис. 10.6: видно, что цифры меняются достаточно плавно, но при этом все время остаются вполне распознаваемыми, практически без непонятных промежуточных значений. Это и есть тот эффект, которого мы ожидаем от хорошего автокодировщика: путешествие через пространство латентных признаков не приведет нас к примерам, которых не бывает в реальной жизни, все время будут порождаться нормальные рукописные цифры (естественно, разные в разных областях пространства). Сравните, кстати, рис. 10.6 и результат 200 эпох обучения на рис. 10.5: эти картинки соответствуют друг другу, и вы видите соответствие между областями, приводящими к одним и тем же цифрам.

Давайте теперь сделаем следующий шаг и доведем наш пример до условного вариационного автокодировщика (conditional VAE). Эта конструкция аналогична условному состязательному автокодировщику (см. раздел 8.5): условный вариационный автокодировщик явным образом получает на вход метки цифр и в результате должен суметь порождать заданную цифру. Для этого нужно подать на вход кодировщику и декодировщику одну и ту же метку текущей цифры. В коде выше это можно сделать с совсем небольшими изменениями. Зададим заглушку для меток `y`, объединим векторы `x` и `y` и подадим оба на вход первого слоя кодировщика:

```
y = tf.placeholder(tf.float32, [None, 10])
xy = tf.concat([x, y], 1)
enc_layer_1 = tf.nn.tanh(tf.add(
        tf.matmul(xy, w["w_recog"]['h1']), w["b_recog"]['b1']))
```

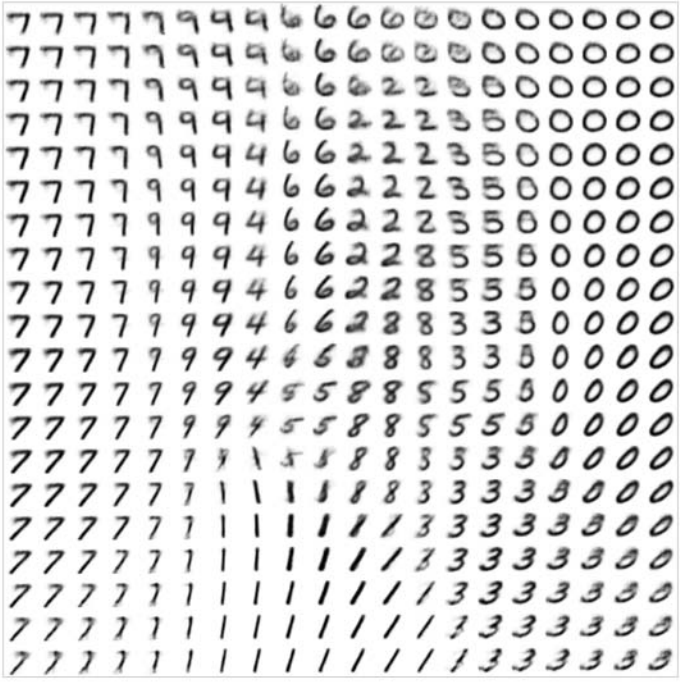

Рис. 10.6. Примеры рукописных цифр, сэмплированные из разных частей распределения скрытых факторов

Выход из первого слоя теперь тоже дополняется меткой текущего значения, и они вместе подаются в декодировщик:

```
zy_mean = tf.concat([z_mean, y], 1)
zy = tf.concat([z, y], 1)
dec_layer_1 = tf.nn.tanh(tf.add(
       tf.matmul(zy, w["w_gener"]['h1']), w["b_gener"]['b1']))
mean_dec_layer_1 = tf.nn.tanh(tf.add(
       tf.matmul(zy_mean, w["w_gener"]['h1']), w["b_gener"]['b1']))
```

Все остальное — абсолютно без изменений: ошибка по-прежнему складывается из ошибки автокодировщика и ошибки на скрытом слое, только теперь получается, что распределение скрытых факторов сравнивается с нормальным «по отдельности» для каждого класса, и на выходе получается стандартное нормальное распределение в каждом классе (см. рис. 10.7 — на нем никакой разницы между разными классами не просматривается). Зато теперь можно сделать условное порождение: на рис. 10.8 показаны примеры результатов сэмплирования с разными метками-условиями. Здесь нет общей двумерной картинки, как на рис. 10.6, но по-прежнему «путешествия» через пространство скрытых факторов, которые показаны в каждой строке рис. 10.8, все время приводят к разумным порожденным цифрам.

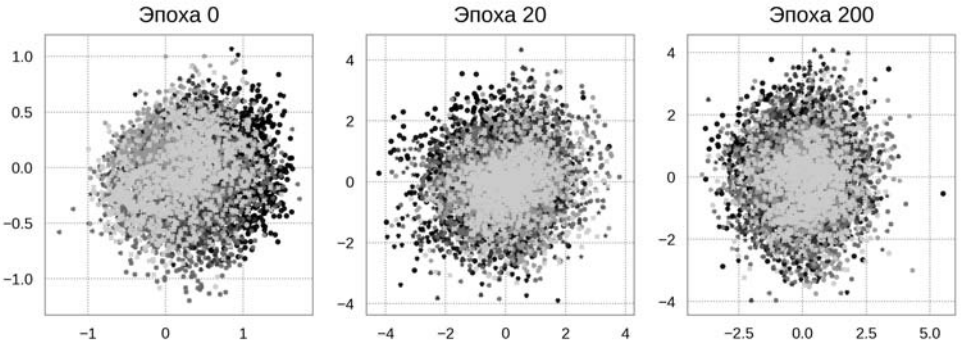

Рис. 10.7. 2D-представление распределения рукописных цифр в датасете MNIST в условном вариационном автокодировщике

Сейчас популярность вариационных автокодировщиков и разных вариантов этой архитектуры продолжает расти, они успешно соперничают с AAE за звание лучшей порождающей модели, основанной на нейронных сетях. Конечно, появились и расширения, и новые варианты вариационных автокодировщиков.

Например, в работе [550] строится конструкция так называемого вариационного автокодировщика с потерями (lossy VAE), в котором контроль над структурой скрытого представления используется для того, чтобы различить важные и неважные части поступающего на вход объекта (например, отделить объекты, изображенные на фото, от текстуры заднего плана), а затем «забыть» неважные части при дальнейшем порождении.

А в [394] строятся *непараметрические* вариационные автокодировщики, то есть VAE с непараметрическими априорными распределениями. Это значит, что они могут увеличивать сложность по мере усложнения и увеличения обучающего датасета, и эта идея далее развивается в *иерархические* непараметрические VAE, которые способны обучать иерархические структуры (фактически деревья) концепций и генерировать новые примеры из каждого узла получающихся деревьев.

Отдельно хочется отметить работу [400], которая предлагает общий взгляд на вариационные и состязательные автокодировщики. Оказывается, что и те, и другие в некотором смысле минимизируют дивергенцию Кульбака — Лейблера между истинным и модельным распределениями, просто в разных направлениях, и поэтому их можно рассматривать как две разные фазы классического алгоритма wake-sleep, который был разработан еще в 1990-е годы для обучения нейронных сетей без учителя [555], причем без участия Джеффри Хинтона и там не обошлось. Хотя мы уже не станем углубляться в детали этих и других современных расширений идей байесовских нейронных сетей, нет сомнений, что будущее глубокого обучения по крайней мере отчасти именно за такими методами.

Рис. 10.8. Примеры рукописных цифр, сэмплированные из разных частей распределения скрытых факторов условным вариационным автокодировщиком

10.5. Байесовские нейронные сети и дропаут

Лина. Ах, зачем вы меня схватили на руки и тащите к окну? Неужели вы хотите меня выбросить?

Ходотов. Нет, помилуйте... Это я вас так люблю!

А. Аверченко. На перепутье

А теперь вернемся к байесовскому выводу на собственно нейронных сетях, который мы анонсировали в первом разделе этой главы; такая последовательность изложения была нужна потому, что для вывода в *байесовских нейронных сетях* (Bayesian neural networks) мы опять будем активно использовать ту же самую идею вариационных приближений, которую объясняли в разделе 10.3 и затем расширяли аппроксиматорами в виде нейронных сетей. В качестве основных источников о современном байесовском выводе в нейронных сетях рекомендуем [171, 172, 191, 279] и недавно появившуюся диссертацию Ярина Гала [170].

Для нейронных сетей байесовский вывод в принципе выглядит точно так же, как и для любой другой вероятностной модели. Мы хотим найти апостериорное распределение:

$$p(\boldsymbol{w} \mid D) = \frac{p(D \mid \boldsymbol{w})p(\boldsymbol{w})}{p(D)} \propto p(D \mid \boldsymbol{w})p(\boldsymbol{w}),$$

где через $\boldsymbol{w}$ мы обозначили вектор всех весов сети, а через X — имеющиеся данные, а потом найти предсказательное распределение:

$$p(x \mid D) = \int_{\boldsymbol{w}} p(x \mid \boldsymbol{w}) p(\boldsymbol{w} \mid D) \mathrm{d}\boldsymbol{w} \propto \int_{\boldsymbol{w}} p(x \mid \boldsymbol{w}) p(D \mid \boldsymbol{w}) p(\boldsymbol{w}) \mathrm{d}\boldsymbol{w}.$$

В частности, в дальнейшем мы будем для определенности говорить о задаче классификации, когда $D = (X,Y)$ состоит из пар вида $(\boldsymbol{x}, y)$, и мы ищем апостериорное распределение:

$$p(\boldsymbol{w} \mid X, Y) = \frac{p(Y \mid X, \boldsymbol{w}) p(\boldsymbol{w})}{p(Y \mid X)} \propto p(Y \mid X, \boldsymbol{w}) p(\boldsymbol{w})$$

и предсказательное распределение:

$$p(y \mid \boldsymbol{x}, X, Y) = \int_{\boldsymbol{w}} p(y \mid \boldsymbol{x}, \boldsymbol{w}) p(\boldsymbol{w} \mid X, Y) \mathrm{d}\boldsymbol{w} \propto \int_{\boldsymbol{w}} p(y \mid \boldsymbol{x}, \boldsymbol{w}) p(Y \mid X, \boldsymbol{w}) p(\boldsymbol{w}) \mathrm{d}\boldsymbol{w}.$$

В нейронных сетях, разумеется, $p(\boldsymbol{w} \mid X, Y)$ имеет очень сложный вид, и аналитически найти это распределение не получается — нужны приближения.

Начнем с краткого исторического обзора. На протяжении всей книги мы раз за разом убеждались в том, что подавляющее большинство идей, использующихся в современных нейронных сетях, появились очень давно (по меркам машинного обучения, конечно): глубокие сети обучались с 1960-х годов, сверточные архитектуры известны и применяются на практике с конца 1980-х, LSTM разрабатывался в течение 1990-х и т. д. С байесовским выводом в нейронных сетях получилась ровно та же история: байесовские нейронные сети восходят к концу 1980-х годов [294, 534], а в явном виде идея добавить в нейронные сети априорные распределения и попробовать получать не одну оценку, а апостериорное распределение появилась в первой половине 1990-х, в работах известных специалистов по статистике и машинному обучению Рэдфорда Нила [383] и Дэвида Маккея [344]. Априорное распределение на веса обычно выбиралось нормальным, $p(\boldsymbol{w}) = \mathcal{N}(\mathbb{O}, \mathbb{I})$, а основная проблема состояла, как всегда в этом деле, в том, как провести вывод. В диссертации Нила [383] вывод был на основе методов Монте-Карло по схеме марковской цепи (Markov chain Monte Carlo, MCMC). А, например, в книге Кристофера Бишопа [44], которую мы всецело рекомендуем как одну из лучших книг о байесовском выводе, кульминацией главы о нейронных сетях становится описание основанного на лапласовском приближении алгоритма байесовского вывода, который предназначем для обучения параметров гауссовского приближения к апостериорному распределению. Таким образом можно получить приближенное апостериорное распределение, оптимизировать гиперпараметры и даже добраться до предсказательного распределения, заметно улучшив результаты, которые показывают «классические», неглубокие нейронные сети; но к современным глубоким сетям все эти рассуждения уже не вполне применимы.

Современный байесовский вывод в нейронных сетях основан, как мы уже упоминали, на вариационных приближениях. Как и в разделе 10.3, мы вводим новое распределение $q(\boldsymbol{w})$ и пытаемся приблизить им истинное апостериорное распределение $p(\boldsymbol{w} \mid X, Y)$, минимизируя расстояние Кульбака — Лейблера между ними:

$$\begin{aligned}
&\mathrm{KL}\left(q(\boldsymbol{w}) \| p(\boldsymbol{w} \mid X, Y)\right) = \\
&= \int q(\boldsymbol{w}) \log \frac{q(\boldsymbol{w})}{p(\boldsymbol{w} \mid X, Y)} \mathrm{d}\boldsymbol{w} = \int q(\boldsymbol{w}) \log \frac{q(\boldsymbol{w})}{p(\boldsymbol{w}) p(Y \mid X, \boldsymbol{w})} \mathrm{d}\boldsymbol{w} + \mathrm{const} = \\
&= -\int q(\boldsymbol{w}) \log p(Y \mid X, \boldsymbol{w}) \mathrm{d}\boldsymbol{w} + \int q(\boldsymbol{w}) \log \frac{q(\boldsymbol{w})}{p(\boldsymbol{w})} \mathrm{d}\boldsymbol{w} + \mathrm{const} = \\
&= -\int q(\boldsymbol{w}) \log p(Y \mid X, \boldsymbol{w}) \mathrm{d}\boldsymbol{w} + \mathrm{KL}\left(q(\boldsymbol{w}) \| p(\boldsymbol{w})\right) + \mathrm{const} = \\
&= -\sum_{i=1}^{N} \int q(\boldsymbol{w}) \log p(y_i \mid f_{\boldsymbol{w}}(\boldsymbol{x}_i)) \mathrm{d}\boldsymbol{w} + \mathrm{KL}\left(q(\boldsymbol{w}) \| p(\boldsymbol{w})\right) + \mathrm{const},
\end{aligned}$$

где в const спрятаны слагаемые, не зависящие от $q(\boldsymbol{w})$ (они не участвуют в минимизации), а в последней строке правдоподобие $p(Y \mid X, \boldsymbol{w})$ разложено в произведение по точкам данных.

Впервые такую минимизацию попытались провести, опять же, в первой половине 1990-х годов, в работе [226], где использовалось очень сильное предположение о форме приближения. Там предполагалось, что $q(\boldsymbol{w})$ полностью раскладывается в произведение нормальных распределений по отдельным весам:

$$q(\boldsymbol{w}) = \prod_{w \in \boldsymbol{w}} q_{\mu_w, \sigma_w}(w) = \prod_{w \in \boldsymbol{w}} \mathcal{N}(w \mid \mu_w, \sigma_w).$$

Даже это оказалось нелегко: только для сетей с одним скрытым уровнем результат удается получить аналитически, что и было проделано в [226]. Но в любом случае метод, не учитывающий корреляции между весами, дает не самые лучшие результаты, а попытки добавить приближению выразительности [31] приводят к тому, что алгоритмы вывода становятся квадратичными от числа весов в сети — для современных огромных нейронных сетей это смерти подобно.

Основных сложностей с исходной задачей минимизации две: во-первых, $\int q(\boldsymbol{w}) p(y_i \mid f_{\boldsymbol{w}}(\boldsymbol{x}_i)) \mathrm{d}\boldsymbol{w}$ никак не вычислить, если у сети больше одного скрытого уровня, во-вторых, получается, что даже чтобы подсчитать функцию, которую мы минимизируем, нужно взять сумму по всему датасету. Вторую проблему решить не так уж сложно: давайте вместо суммирования по всем N тренировочным примерам выберем из них случайное подмножество S размера $M < N$ и перевзвесим. Теперь мы оптимизируем такую функцию:

$$\mathcal{L}(\boldsymbol{w}) = -\frac{N}{M} \sum_{i \in S} \int q(\boldsymbol{w}) p(y_i \mid f_{\boldsymbol{w}}(\boldsymbol{x}_i)) \mathrm{d}\boldsymbol{w} + \mathrm{KL}\left(q(\boldsymbol{w}) \| p(\boldsymbol{w})\right).$$

Если S выбирать случайно, получится несмещенная оценка исходной суммы. Примерно так же мы когда-то заменяли градиентный спуск (который тоже, формально говоря, требует суммирования по всему датасету) стохастическим градиентным спуском, где сумма берется по мини-батчу вместо всего датасета, а теперь мы выбираем мини-батч S для вычисления функции оптимизации.

С интегралом придется повозиться. Есть разные способы построить хорошую стохастическую оценку для этого интеграла [45, 405], но нам сейчас опять поможет предложенный в [279, 280] трюк, с которым мы уже познакомились в разделе 10.3.

Прежде всего заметим, что нам нужно оценить не сам интеграл, а производную от него, которую мы могли бы использовать в градиентном спуске:

$$I(\boldsymbol{\theta}) = \frac{\partial}{\partial \boldsymbol{\theta}} \int f(x) p_{\boldsymbol{\theta}}(x) \mathrm{d}x.$$

Предположим теперь (репараметризация), что $p_{\boldsymbol{\theta}}(x)$ можно параметризовать как $p(\epsilon)$ уже без параметров, где $x = g(\boldsymbol{\theta}, \epsilon)$. Это, например, верно для нормального распределения: если $p_{\boldsymbol{\theta}}(x) = \mathcal{N}(x \mid \mu,\sigma^2)$, то $g(\boldsymbol{\theta},\epsilon) = \mu + \sigma\epsilon$, и $p(\epsilon) = \mathcal{N}(\epsilon \mid \mathbb{O},\mathbb{I})$. Теперь, точно как выше, мы получаем несмещенную оценку:

$$\frac{\partial}{\partial \boldsymbol{\theta}} \int f(x) p_{\boldsymbol{\theta}}(x) \mathrm{d}x \approx \frac{\partial}{\partial \boldsymbol{\theta}} f(g(\boldsymbol{\theta},\epsilon)) = f'(g(\boldsymbol{\theta},\epsilon)) \frac{\partial}{\partial \boldsymbol{\theta}} g(\boldsymbol{\theta},\epsilon).$$

Например, для нормального распределения $x = \mu + \sigma\epsilon$, и мы получим:

$$\begin{aligned}
\frac{\partial}{\partial \mu} \int f(x) p_{\boldsymbol{\theta}}(x) \mathrm{d}x &= \int f'(x) p_{\boldsymbol{\theta}}(x) \mathrm{d}x, \\
\frac{\partial}{\partial \sigma} \int f(x) p_{\boldsymbol{\theta}}(x) \mathrm{d}x &= \int f'(x) \frac{x-\mu}{\sigma} p_{\boldsymbol{\theta}}(x) \mathrm{d}x.
\end{aligned}$$

И теперь можно подставить эту оценку обратно в целевую функцию $\mathcal{L}(\boldsymbol{w})$. Сделаем репараметризацию:

$$\begin{aligned}
\mathcal{L}(\boldsymbol{\theta}) = -\frac{N}{M} \sum_{i \in S} \int q_{\boldsymbol{\theta}}(\boldsymbol{w}) \log p(y_i \mid f_{\boldsymbol{w}}(\boldsymbol{x}_i)) \mathrm{d}\boldsymbol{w} + \mathrm{KL}\left(q(\boldsymbol{w}) \| p(\boldsymbol{w})\right) = \\
= -\frac{N}{M} \sum_{i \in S} \int p(\boldsymbol{\epsilon}) \log p(y_i \mid f_{g(\boldsymbol{\theta},\boldsymbol{\epsilon})}(\boldsymbol{x}_i)) \mathrm{d}\boldsymbol{\epsilon} + \mathrm{KL}\left(q(\boldsymbol{w}) \| p(\boldsymbol{w})\right),
\end{aligned}$$

а затем подставим несмещенную стохастическую оценку вместо интеграла:

$$\hat{\mathcal{L}}(\boldsymbol{\theta}) = -\frac{N}{M} \sum_{i \in S} \log p(y_i \mid f_{g(\boldsymbol{\theta},\boldsymbol{\epsilon})}(\boldsymbol{x}_i)) + \mathrm{KL}\left(q(\boldsymbol{w}) \| p(\boldsymbol{w})\right).$$

Таким образом, один шаг алгоритма минимизации расхождения между $q_{\boldsymbol{\theta}}(\boldsymbol{w})$ и $p(\boldsymbol{w} \mid X,Y)$ будет выглядеть так:

- сначала сэмплировать M случайных примеров из тренировочного набора размера N и M случайных величин $\hat{\boldsymbol{\epsilon}}_i \sim p(\boldsymbol{\epsilon})$;
- затем вычислить обновление весов:

$$\Delta\boldsymbol{\theta} = -\frac{N}{M}\sum_{i\in S}\frac{\partial}{\partial\boldsymbol{\theta}}\log p(y_i \mid f_{g(\boldsymbol{\theta},\hat{\boldsymbol{\epsilon}}_i)}(\boldsymbol{x}_i)) + \frac{\partial}{\partial\boldsymbol{\theta}}\mathrm{KL}\left(q(\boldsymbol{w})\|p(\boldsymbol{w})\right).$$

Этот алгоритм, постепенно обновляя параметры $\boldsymbol{\theta}$ распределения $q_{\boldsymbol{\theta}}(\boldsymbol{w})$, приведет это распределение как можно ближе к истинному апостериорному распределению $p(\boldsymbol{w} \mid X,Y)$ — ровно то, что мы и хотели сделать. Потом можно будет и байесовские предсказания делать обычным стохастическим методом: посэмплировать несколько разных наборов весов $\hat{\boldsymbol{w}}_r \sim q_{\boldsymbol{\theta}}(\boldsymbol{w})$ и усреднить результаты:

$$q_{\boldsymbol{\theta}}(y \mid \boldsymbol{x}) := \frac{1}{R}\sum_{r=1}^{R} p(y \mid \boldsymbol{x},\hat{\boldsymbol{w}}_r).$$

Но это еще не все — самое интересное только начинается! Давайте для примера рассмотрим обычную полносвязную нейронную сеть с одним скрытым слоем. У нее три вида параметров: матрица весов входного слоя W_1, матрица весов скрытого слоя W_2 и вектор свободных членов $\boldsymbol{b}$, то есть $\boldsymbol{\theta} = (W_1,W_2,\boldsymbol{b})$. Давайте выберем в качестве $q_{\boldsymbol{\theta}}(\boldsymbol{w})$ одно конкретное семейство распределений, которое выглядит так:

- сначала возьмем бинарные случайные величины $\boldsymbol{\epsilon}_1$ и $\boldsymbol{\epsilon}_2$, которые являются результатами независимых бросков нескольких монеток с вероятностями p_1 и p_2, и сэмплируем их, получив векторы $\hat{\boldsymbol{\epsilon}}_1$ и $\hat{\boldsymbol{\epsilon}}_2$;
- затем построим из них диагональные матрицы $\mathrm{diag}(\hat{\boldsymbol{\epsilon}}_1)$ и $\mathrm{diag}(\hat{\boldsymbol{\epsilon}}_2)$;
- и определим функцию $g(\boldsymbol{\theta},\hat{\boldsymbol{\epsilon}}) = (\mathrm{diag}(\hat{\boldsymbol{\epsilon}}_1)W_1, \mathrm{diag}(\hat{\boldsymbol{\epsilon}}_2)W_2, \boldsymbol{b})$.

При таком выборе функции g получится, что при вычислении целевой функции $\hat{\mathcal{L}}(\boldsymbol{\theta})$ для нашей минимизации мы считаем выход нейронной сети как будто с дропаутом с вероятностью p_1 после первого слоя и p_2 после второго! Если же мы теперь выберем в качестве априорного распределения $p(\boldsymbol{w})$ независимые нормальные распределения на каждом из весов, то целевая функция будет пропорциональна следующей величине:

$$-\frac{N}{M}\sum_{i\in S}\log p(y_i \mid f_{g(\boldsymbol{\theta},\boldsymbol{\epsilon})}(\boldsymbol{x}_i)) + \lambda_1\|W_1\|^2 + \lambda_2\|W_2\|^2 + \lambda_3\|\boldsymbol{b}\|^2,$$

то есть мы получим, что обычный вывод в нейронных сетях с дропаутом в точности соответствует байесовскому выводу в вариационном приближении с таким семейством распределений q[1].

[1] Доказательства этого утверждения и того, что $\log p(y_i \mid f_{g(\boldsymbol{\theta},\boldsymbol{\epsilon})}(\boldsymbol{x}_i))$ соответствует целевым функциям в обычных задачах регрессии и классификации, достаточно просты. Мы оставляем их читателю

Получилась вот какая последовательность. Дропаут изначально был инженерным решением, некоторое время это был просто трюк, который позволял нейронным сетям работать лучше. Конечно, люди и раньше пытались давать теоретические объяснения феномену дропаута. Например, Джеффри Хинтон в своих лекциях рассказывал о том, как дропаут представляет собой в некотором роде «байесовское усреднение» огромного числа разных моделей, отличающихся друг от друга архитектурой: все они делят между собой веса, и каждая модель обучается только на одном-единственном мини-батче, когда выпадает эта конкретная конфигурация выброшенных нейронов.

Некий образ связи между байесовским выводом и дропаутом в этом объяснении есть, но только после процитированных выше работ [171, 279], относящихся к 2015 году, дропауту наконец-то было дано действительно полное и удовлетворительное теоретическое обоснование[1].

Зачем нужно это обоснование? Казалось бы, дропаут и так отлично работает. Одно важное следствие состоит в том, что изначально доля выбрасываемых нейронов в дропауте была фиксированным числом. Но теперь, когда мы переформулировали обучение с дропаутом в виде максимизации вариационной нижней оценки, мы без проблем можем оптимизировать долю дропаута тоже: для этого достаточно просто зафиксировать остальные веса и максимизировать ту же самую оценку по доле дропаута! Более того, мы теперь можем подбирать ее индивидуально: для каждого слоя, для каждого нейрона, даже для каждой связи между нейронами. Этот метод получил название *вариационный дропаут* (variational dropout) [279]. Изначальная процедура дропаута таких вольностей совершенно не предусматривала, из нее не было понятно, как можно автоматически настраивать вероятность дропаута. При этом никакого страха переобучиться нет — мы не меняем априорное распределение, просто все точнее и точнее приближаем апостериорное. Более того, выяснилось, что вариационный дропаут добавляет разреженности в глубокие нейронные сети, то есть обнуляет подавляющее большинство весов [374]; этот результат, кстати, был получен в России, в лаборатории Дмитрия Ветрова.

А другое важное следствие состоит в том, что теперь мы можем лучше понять, как нужно делать дропаут в более сложных моделях. Рассмотрим, например, такой вопрос: как делать дропаут в рекуррентных сетях? В известной работе [583] проводилось подробное исследование методов дропаута для рекуррентных сетей. Авторы пришли к выводу, что делать дропаут нужно только между слоями, а между узлами одного рекуррентного слоя не нужно (рис. 10.9). Этому нашлось очень

в качестве упражнения, а также ссылаемся на [171, 279], где можно найти и много других интересных замечаний об этом выводе.

[1] Кстати, примерно такая же история случилась в конце 1990-х годов с другим известным аппаратом машинного обучения, бустингом: некоторое время он «просто работал», а потом уже удалось понять, что в бустинге происходит на самом деле и какой функционал оптимизируется. И сразу же оказалось, что когда исследователи поняли функционал для исходной конструкции бустинга, они сразу же смогли его обобщить, распространить на другие случаи и получить много новых моделей и алгоритмов вывода.

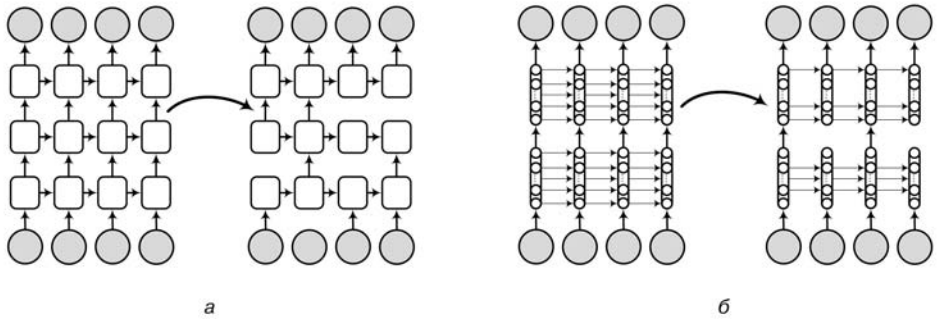

Рис. 10.9. Дропаут в рекуррентных сетях:
а — дропаут только между уровнями [583]; *б* — дропаут везде [169]

логичное объяснение: дропаут между узлами одного рекуррентного слоя разрушает возможность сделать долгосрочную память, ведь связь с предыдущим нейроном очень часто будет нарушена, а мы хотим как раз обучить как можно более долгосрочные зависимости. Однако после того как появилось теоретическое обоснование для дропаута, тут же родилась и идея о том, как его делать внутри рекуррентного слоя [169].

Действительно, давайте рассмотрим функцию $f_y = f_{g(\boldsymbol{\theta},\boldsymbol{\epsilon})}$ в случае рекуррентной сети (для простоты обычной, но на LSTM и GRU это тоже можно распространить). Она теперь зависит еще от скрытого состояния, оно, в свою очередь, зависит от весов на предыдущем уровне через функцию f_h, и так далее:

$$f_y(\boldsymbol{h}_t) = f_y(f_h(\boldsymbol{x}_t, \boldsymbol{h}_{t-1})) = f_y(f_h(\boldsymbol{x}_t, f_h(\boldsymbol{x}_{t-1} f_h(\ldots f_h(\boldsymbol{x}_1, \boldsymbol{h}_0)\ldots)).$$

Мы знаем, что дропаут соответствует подбрасыванию монетки для каждого веса матрицы. Матриц, как мы помним из главы 6, у рекуррентной сети три: входная U, выходная V и матрица рекуррентных весов W. Интуиция, описанная в [583], говорит, что дропаут на матрице W не работает, потому что рекуррентные связи должны сохраняться. Поэтому дропаут должен выглядеть как на рис. 10.9, *а*, сохраняя все рекуррентные связи и выбрасывая часть связей между уровнями.

Но если добавить дропаут по схеме выше, в виде подбрасывания монетки *один раз* для каждого столбца матрицы W, то эта проблема исчезает: да, матрица W участвует в формуле много раз, но все связи во времени между нейронами будут сохраняться, потому что каждый нейрон будет либо включен, либо выключен одновременно на всем протяжении входной последовательности. Таким образом, получается, что дропаут на рекуррентных связях использовать можно, нужно только зафиксировать случайно выбрасываемые нейроны один раз для каждого входа (или мини-батча), а на разных входах можно бросать монетки заново, ничего

не испортится. Такой дропаут мы изобразили рис. 10.9, *б*: теперь можно выбрасывать не только связи между уровнями, но и отдельные компоненты рекуррентных связей, только делать это надо одинаково на протяжении всего слоя. В [169] приводятся результаты экспериментов, достаточно убедительно показывающие, что такой дропаут действительно делает рекуррентные сети лучше и предотвращает оверфиттинг без ухудшения качества.

Байесовские нейронные сети — это очень быстро развивающаяся область. Все эти результаты были получены буквально в течение последнего года-двух, но уже есть и конкретные практические применения. Например, в работе [488] строится система автоматического перевода на основе корпуса переводных новостей в рамках конкурса, проводившемся на конференции WMT 2016 [157] (кстати, среди построенных моделей есть и перевод с английского на русский и обратно). Авторы использовали стандартную схему с энкодером, декодером и сетью внимания (вспомним раздел 8.1), но в некоторых случаях оказалось, что для того чтобы избежать оверфиттинга, нужно было использовать дропаут в рекуррентной архитектуре. Они использовали вариационный дропаут по приведенной выше схеме и получили существенное улучшение. В работе [574] рассматривается задача «сшивания» вместе частей одного изображения, в частности для таких медицинских применений, как МРТ разных частей одного и того же мозга. Авторы построили модель, основанную на сверточных сетях, но рассмотрели ее с байесовских позиций. Дело в том, что разные изображения здесь «сшиваются» не вполне точно: между разными снимками проходит время, они могут быть в разной степени зашумлены, пациент мог немного сдвинуться и так далее. И для медицины важно понимать, какие части полученного снимка мы знаем точно, а какие не очень; для этого и хотелось бы знать все апостериорное распределение целиком и уметь оценивать его дисперсию, а не только получить одну оптимальную точку. Аналогичные применения, в которых у изображения появляется «карта неопределенности», оценивающая неопределенность полученной модели в разных его частях, появились и, например, при создании беспилотных автомобилей [276, 277].

Есть и другие интересные применения, но закончить хочется все-таки на том, что байесовские нейронные сети — это, несомненно, будущее обучения нейронных сетей. Уже сам факт того, что эмпирические «трюки» обучения глубоких сетей постепенно начинают получать строгие математические обоснования, вселяет надежду. Байесовские объяснения происходящего могут привести к целому ряду новых важных продвижений. Например (см. также [172]):

- если мы понимаем, как связаны методы обучения и регуляризации с априорными распределениями и алгоритмами приближенного вывода, мы можем получить новые методы обучения, просто подставляя другие априорные распределения, те, которые лучше подходят для конкретной задачи;
- байесовское обучение обычно можно продолжить до обучения гиперпараметров, то есть параметров априорных распределений, из которых берутся параметры модели;

- байесовские методы, как в упомянутых выше приложениях, позволяют оценивать неопределенность, остающуюся в модели после обучения; это тоже имеет очевидную ценность для приложений;
- многие классические байесовские модели достигают хороших результатов, используя очень простые базовые предположения (например, тематическое моделирование работает, превращая тексты в мешки слов); выразительность этих моделей наверняка можно значительно улучшить, заменив упрощающие предположения на более гибкие модели, формализуемые с помощью глубоких нейронных сетей.

Итак, как мы увидели на протяжении этой главы, байесовские модели и глубокие нейронные сети могут помогать друг другу: нейронные сети делают байесовские модели более гибкими и богатыми, а байесовское обучение позволяет лучше понимать, как обучать нейронные сети. Однако сейчас эта область находится только в самом начале пути и ждет новых исследователей. Дерзайте, коллеги!

10.6. Заключение: что не вошло в книгу и что будет дальше

> ...Она совсем не из этой страны — она из тех светозарных краев, где нас давным-давно ждут. Там вечно искрится роса и колышется стройный тростник.
>
> *Г. Миллер. Тропик Козерога*

Вот и подходит к концу наша книга. В заключение мы расскажем о нескольких направлениях, которые в книгу не вошли — как знать, может быть, они будут ждать читателей в следующих изданиях — а также попробуем дать прогноз о том, куда будет дальше двигаться человеческая мысль.

В главе 5 мы подробно говорили о современных архитектурах для распознавания изображений, которые обучаются на ImageNet, то есть классифицируют фотографию согласно тому, какой объект на ней изображен. Однако в жизни мы, когда смотрим вокруг себя, решаем не эту задачу, а задачу *распознавания объектов* (object detection): на фотографии может быть много чего изображено, и мы умеем распознавать каждый из этих объектов и указывать, где именно он находится. А иногда приходится решать и задачу *сегментации* (segmentation): не просто распознать, какие и сколько объектов на фото, но и буквально уметь указывать, какие пиксели фотографии им принадлежат. Для этого применяются архитектуры, которые основаны на все тех же VGG или ResNet, но имеют дополнительные слои, которые обучаются отвечать на вопрос, где именно находятся на фотографии те или иные объекты. Лидерами этой области сейчас являются такие модели, как YoLo [444, 576] и Faster R-CNN [152] для распознавания объектов и SegNet [20], U-Net [454] и Mask R-CNN [353] для сегментации.

В главе 8 мы достаточно кратко поговорили о моделях с вниманием и обошли тем самым вниманием несколько интересных направлений развития этой идеи. Так, для ответов на вопросы применяются так называемые *сети с памятью* (memory networks) [10, 561, 565], которые могут явным образом хранить вход и управлять этой памятью с помощью своеобразного механизма внимания, который решает, к какой именно части памяти нужно сейчас обратиться.

А следующий шаг в развитии этой идеи сделали так называемые *нейронные машины Тьюринга* (neural Turing machines) [194], которые используют своеобразный вариант механизма внимания для того, чтобы буквально управлять лентой машины Тьюринга и обучаться тем или иным алгоритмам по набору входов и выходов. Современные «нейрокомпьютеры» обучаются с помощью обучения с подкреплением [582], и сейчас это направление активно развивается во все той же компании *DeepMind*. Последний вариант такой модели, получивший громкое (но заслуженное) название *дифференцируемый нейронный компьютер* (differentiable neural computer, DNC), способен уже управлять практически настоящей «оперативной памятью», делая чтение и запись по адресам, и решать задачи, придумывая алгоритмы на графах [240].

Порождающие состязательные сети (глава 8), глубокое обучение с подкреплением (глава 9) и нейробайесовское обучение, которому посвящена эта глава, сейчас развиваются так быстро, что для книги глупо и пытаться оставаться на переднем крае. Очередной набор важных новых идей был представлен на конференции ICML в августе 2017 года. Например, в работе [450] представлен вариант обучения с подкреплением с противником (adversarial reinforcement learning), в котором противник активно пытается сбить агента с толку теми или иными дестабилизирующими действиями. В [8] разработан вариант GAN, в котором вместо дивергенции Йенсена — Шеннона используется расстояние Вассерштейна (которое обычно называется Earth Mover's Distance, EMD), что приводит к более стабильному обучению и меньшему схлопыванию мод. А работа [357] развивает новый метод обучения вариационных автокодировщиков, который... представляет максимизацию правдоподобия как игру между двумя игроками и основан на соперничающих сетях.

В последнее время наблюдается очевидный тренд на сближение между состязательными сетями и байесовскими методами: в работе [461] предлагается конструкция байесовского варианта GAN, а в [549] к GAN успешно применяются методы вариационного вывода. Мы полагаем, что ближайшее будущее — именно за такой «смычкой» между байесовскими и состязательными методами.

Есть и новые повороты более классических идей: например, в [440] предлагаются так называемые recurrent highway networks (да, в этой работе тоже не обошлось без Юргена Шмидхубера), которые расширяют архитектуру LSTM новыми связями и тем самым радикально улучшают, например, языковое моделирование. И это лишь несколько примеров. В целом, сейчас можно смело открывать список работ любой ведущей конференции по машинному обучению (две самые лучшие — это

ICML, International Conference on Machine Learning, и NIPS, Annual Conference on Neural Information Processing Systems), обязательно будет очень много интересного, а если вы прочли эту книгу, то и с большинством современных статей по обучению глубоких сетей должны справиться.

И, наконец, главный вопрос: что будет дальше? Куда все это катится? Честно скажем, что пытаться детально предсказывать развитие науки — занятие абсолютно бесперспективное. Если бы мы знали, что будет дальше, мы бы уже давно это сделали. Но некий общий вектор увидеть можно.

Общее настроение в начале революции глубокого обучения было, если можно так выразиться, «луддистским». Внезапно оказалось, что вероятностные модели, которые составляли основное содержание машинного обучения в течение почти двух десятилетий, как бы и «не нужны»: можно просто придумать удачную архитектуру нейронной сети, начать обучать ее градиентным спуском, и при достаточно большом датасете и достаточно мощной видеокарте все обязательно получится. Возможно, вы и сами вынесли такое ощущение из первых глав этой книги: мы не раз подчеркивали, что многие прорывы в машинном обучении, о которых мы писали ранее, получены «всего лишь градиентным спуском», безо всякой особенно сложной математики, хоть и не без важных и неочевидных трюков. Но сейчас постепенно становится понятным, что без математики все-таки не обойтись. В этой главе мы попытались дать очень краткое введение в то, что сейчас происходит на переднем крае обучения глубоких сетей. Хотя сейчас революция глубокого обучения все еще в самом разгаре, и в этой науке расцветают все цветы, мы полагаем, что будущее — за байесовскими методами и слиянием вероятностных моделей с нейронными сетями. Именно на этом пути получены самые интересные из недавних результатов, и именно в этом направлении двигаются ведущие исследователи. В результате глубокие нейронные сети будут делать то, что они делают лучше всего: служить универсальными аппроксиматорами для очень сложных функций; а функции эти будут, например, плотностями интересующих нас распределений. Как это всегда и бывает, очередной виток спирали развития науки не отменяет предыдущий, а расширяет, усложняет и улучшает его.

Еще одно большое направление, которое сейчас только начинается, состоит в том, чтобы объединять разных «агентов», выраженных нейронными сетями (а возможно, и другими типами моделей машинного обучения) в некую единую архитектуру, которая могла бы использовать разных агентов для различных задач и не переобучаться для каждого нового типа задач заново. Теоретически, именно это должен так или иначе делать искусственный интеллект общего назначения, который мог бы иметь шансы достичь уровня человека.

Интересный первый шаг в этом направлении — модель PathNet, представленная все тем же DeepMind [414]. PathNet — это модулярная нейронная сеть, состоящая из нескольких уровней, на каждом из которых расположено несколько «модулей», каждый из которых тоже является нейронной сетью. Для разных задач PathNet строит новые пути через эту модулярную архитектуру, то есть для задачи

распознавания рукописных цифр может быть использовано другое подмножество модулей, чем для задачи игры в приставку Atari, но модули при этом общие, а разные пути выбираются по сути эволюционным методом, генетическим алгоритмом. Результаты в [414] достаточно убедительны, и хотя пока это только первый шаг, нет сомнений, что перенос обучения (transfer learning) и построение архитектур общего назначения — это одна из центральных задач в машинном обучении на ближайшие годы.

А закончить хочется результатами опросов, которые в 2014 году были подытожены в работе Винсента Мюллера и уже упоминавшегося нами Ника Бострома [380]. Они опрашивали ведущих экспертов в области искусственного интеллекта; в опросе было несколько групп участников, от участников конференции AGI (Artificial General Intelligence) до топ-100 ученых в этой области по индексу цитирования (кстати, ответили 29 из 100, что показывает серьезность затеи). На вопрос о том, когда мы сможем разработать полноценный искусственный интеллект человеческого уровня, медианный ответ был — к 2075 году (интересно, что топ-100 ученых здесь были даже чуть более оптимистичны, с медианой в 2070 году), а вероятность в 50 % этому событию половина участников опроса дают уже в 2040 году или раньше.

Это значит, что вовсе не исключено (хотя, конечно, никем и не гарантировано), что мы с вами, дорогие читатели, доживем до того момента, когда искусственный интеллект сравняется с нами... а там и превзойдет. После достижения паритета искусственному интеллекту уже вряд ли что-то сильно помешает нас превзойти. И тогда наша жизнь неизбежно изменится до полной неузнаваемости: даже если не будет никаких «взрывов» и «сингулярностей», скорость технологического и научного прогресса наверняка вырастет настолько, что все современные жалобы на «постоянно меняющийся мир», которые так любят некоторые психологи, покажутся детским лепетом. Как ужиться с настоящим искусственным интеллектом, а не просто набором специализированных моделей, каждая из которых умеет только играть в го или водить машину, — это вопрос, который нам с вами, возможно, придется решать в течение нашей жизни. И от того, как мы его решим, зависит все будущее нашей с вами цивилизации.

А может, все будет совсем не так. И это ужасно интересно.