МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ

Кафедра дифференциальных уравнений и системного анализа

ЛЕГУШЕВ

Дмитрий Александрович

БАЙЕСОВСКИЕ МЕТОДЫ В МАШИННОМ ОБУЧЕНИИ

Магистерская диссертация

Специальность 1-31 80 03 «Математика и компьютерные науки»

Научный руководитель:
ассистент А. П. Тишуров

Допущена к защите

«___» _____ 2022 г.

Зав. кафедрой дифференциальных уравнений и системного анализа

канд. физ.-мат. наук, доцент Л. Л. Голубева

Минск, 2022

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

С развитием современных технологий и научного прогресса человечество начинает пытаться применять Байесовские методы во многих сферах жизнедеятельности, начиная с разработки компьютерных игр и заканчивая исследованием медицинских препаратов. Данный подход даёт ещё большие возможности многим алгоритмам машинного обучения: заполнение недостающих данных, извлечение гороздо большего количества информации из небольших датасетов, оптимизации параметров нейронных сетей и других моделей. Также Байесовские методы позволяют нам оценивать неопределённости в анализе данных, что является необходимой особенностью для таких областей, как медицина. Применительно к алгоритмам глубокого обучения, Байесовские методы позволят сжимать глубокие модели в сотни раз, автоматически подбирать гиперпараметры процесса обучения, сохраняя время и деньги.

Байесовская оптимизация - это подход к оптимизации целевых функций, для оценки которого требуется много времени (минут или часов). Данный процесс лучше всего подходит для оптимизации в непрерывных пространствах размерности менее 20 и допускает стохастический шум при вычислении функции. Он строит статистическую модель для целевой функции и количественно определяет неопределенность в этой моделе, используя регрессию гауссовского процесса и метод байесовского машинного обучения, а затем использует функцию оценки, определенную из этой модели, чтобы решить, какая наиболее оптимальная позицию, чтобы вычислить целевую функцию.

В дипломной работе будет показано как работает байесовская оптимизация и регрессия гауссовского процесса на примере итеративной генерации лиц несуществующих людей по заданным характеристикам. При этом генерация будет осуществляться при помощи модели с набором нейросетей определённого типа, которые обучались на основе новейших исследований.

ОБЩАЯ ХАРАКТЕРИСТИКА РАБОТЫ

В магистерской диссертации 57 страниц, 16 иллюстраций, 6 источников, 3 приложения.

Ключевые слова: МОДЕЛЬ СКРЫТЫХ ПЕРЕМЕННЫХ, ЕМ-АЛГОРИТМ, ВАРИАЦИОННЫЕ АВТОКОДИРОВЩИКИ, ГАУССОВСКИЙ ПРОЦЕСС, БАЙЕСОВСКАЯ ОПТИМИЗАЦИЯ

Объектом исследования диссертации является компьютерное представление изображения лица человека.

Целью работы является реализация итеративного генерирования изображения лица человека по заданным параметрам.

Для достижения поставленной цели были использованы: язык программирования Python, библиотека для тензорных вычислений TensorFlow (метод генерации изображений на основе вариационного автокодировщика) и библиотеки GPy и GPyOpt (реализация нюансов алгоритма Байесовской оптимизации для подбора параметров генерации).

В диссертации получены следующие результаты:

1. Разработана модель вариационного автокодировщика для генерации лиц. Реализован алгоритм работы Байесовской оптимизации.

2. Произведены эксперименты по подбору параметров для генерации лица человека с заданными характеристиками.

Все результаты магистерской диссертации доказаны в соответствии с ... Новизна результатов состоит в ... Обоснованность и достоверность полученных результатов обусловлена ...

Магистерская диссертация выполнена автором самостоятельно.

АГУЛЬНАЯ ХАРАКТАРЫСТЫКА ПРАЦЫ

У магістарскай дысертацыі 57 старонак, 16 малюнкаў, 6 крыніц, 3 дадатка.

Ключавыя словы: МАДЭЛЬ ПРЫХАВАНЫХ ПЕРАМЕННЫХ, ЕМ-АЛГАРЫТМ, ВЫРАЯЦЫЙНЫЯ АЎТАКАДАВАЛЬНІКІ, ГАУСАЎСКІ ПРАЦЭС, БАЕСАЎСКАЯ АПТЫМІЗАЦЫЯ

Аб'ектам даследавання дысертацыі з'яўляецца кампутарная выява асобы чалавека.

Мэтай працы з'яўляецца ітэратыўнае генераванне выявы асобы чалавека па зададзеных параметрах.

Для дасягнення пастаўленай мэты выкарыстоўваліся: мова праграмавання Python, бібліятэка для тэнзарных вылічэнняў TensorFlow (метад генерацыі малюнкаў на аснове варыяцыйнага аўтакадавальніка) і бібліятэкі GPy і GPyOpt (рэалізацыі нюансаў алгарытму Баесаўскай аптымізацыі для падбору параметраў генерацыі).

У дысертацыі атрыманы наступныя вынікі:

1. Распрацавана мадэль варыяцыйнага аўтакадавальніка для генерацыі выяў асоб. Рэалізаваны алгарытм Баесаўскай аптымізацыі.

2. Праведзены эксперыменты па падбору параметраў для генерацыі асобы чалавека з зададзенымі якасцямі.

Усе вынікі магістарскай дысертацыі даказаны ў адпаведнасці з ... Навізна вынікаў складаецца ў ... Абгрунтаванасць і дакладнасць атрыманых вынікаў абумоўлена ...

Магістарская дысертацыя выканана аўтарам самастойна.

GENERAL DESCRIPTION OF WORK

The master thesis is presented in the form of an explanatory note of 57 pages, 16 figures, 6 references, 3 applications.

Keywords: LATENT VARIABLE MODEL, EM-ALGORITHM, VARIATIONAL AUTOENCODER, GAUSSIAN PROCESSING, BAYESIAN OPTIMIZATION

The research object of the thesis is to study a computer generated picture of a person's face.

The purpose of this work is an iterative generation of a person's face by specified parameters.

To achieve the goal we used Python programming language, the framework for tensor computing TensorFlow (picture generation method based on variational autoencoders) and GPy and GPyOpt libraries (implementation of the nuances of the Bayesian optimization algorithm for selecting the generation parameters).

The main results of the thesis are as follows:

1. The model of variational autoencoder for face generation was obtained. Bayesian optimization algorithm was implemented.

2. Experiments to select parameters for generating a person's face with given characteristics were carried out.

All the results of the thesis are proven in accordance with ... The novelty of the results lies in ... The validity and reliability of the obtained results are due to ...

The master thesis was done solely by the author.

ГЛАВА 1

Введение в байесовские методы

1.1 Основные определения.

Вероятность на Ω с σ-алгеброй $\mathscr{A}$ — это отображение $P : \mathscr{A} \to \mathbb{R}$, удовлетворяющая следующим аксиомам:

1. неотрицательность: $P(A) \geq 0, \quad \forall A \in \mathscr{A}$
2. нормированность: $P(\Omega) = 1$
3. аддитивность: $P(A \cup B) = P(A) + P(B), \quad A \bigcap B = \emptyset$
4. непрерывность в нуле: $\forall B_1 \supset B_2 \supset B_3 \supset \ldots \quad$ и $\quad \bigcap\limits_{i=1}^{\infty} B_i = \emptyset \Rightarrow$ $P(B_n) \underset{n \to 0}{\longrightarrow} 0$

тогда $(\Omega, \mathscr{A}, P)$ — вероятностное пространство.

Две случайные величины называются независимыми, если:

$$P(A \cap B) = P(A)P(B)$$

Вероятность события B при условии свершения события A называется условной, если:

$$P(B|A) = \frac{P(B \cap A)}{P(A)}, \quad P(A) > 0,$$

$(\Omega, \mathscr{A}, P)$— вероятностное пространство.

Исходя из определения условной вероятности следует цепное правило:

- $P(X \cap Y) = P(B|A)P(A)$
- $P(X \cap Y \cap Z) = P(X|Y \cap Z)P(Y|Z)P(Z)$
- $P(X_1 \cap \cdots \cap X_n) = \prod\limits_{i=1}^{N} P(X_i|X_1, \ldots, X_{i-1})$

А также правило суммы:

$$P(X) = \int_{-\infty}^{\infty} P(X, Y) dY$$

И, наконец, самая важная теорема данной работы — теорема Байеса:

Θ – параметры (например параметры модели),

X – наблюдения (например некоторые данные), тогда

$$P(\Theta|X) = \frac{P(X \cap \Theta)}{P(X)} = \frac{P(X|\Theta)P(\Theta)}{P(X)} \text{ , где:}$$

$P(\Theta)$ – априорная вероятность параметров Θ (начальные сведения (вероятность) о параметрах (исходя из распределения Θ));

$P(\Theta|X)$ – апостериорная вероятность (начальные сведения (вероятность) о параметрах, после получения некоторых данных);

$P(X|\Theta)$ – правдоподобие (показывает насколько хорошо параметры интерпретируют наши данные);

$P(X)$ – полная вероятность наступления события B.

Исходя из данной теоремы можно ввести понятие online обучения. Предполагая, что у нас есть данные, приходящие маленькими порциями, можно обновлять параметры модели и затем, используя новую апостериорную вероятность как априорную можно перейти к следующему эксперименту с новой порцией данных:

$$P_k(\Theta) = P(\Theta|X_k) = \frac{P(X_k|\Theta)P_{k-1}(\Theta)}{P(X_k)}$$

Рассмотрим модель наивного байесовского классификатора. У нас есть значения класса C, которые прямо воздействует на значения из других классов A_1, A_3, A_3, A_4. При этом A_1, A_2, A_3, A_4 не влияют друг на друга и их распределение может быть различное, тогда совместное распределение при помощи теоремы Байеса можно представить в виде:

$$P(C \cap A_1 \cap A_2 \cap A_3 \cap A_4) = P(C) \prod_{i=1}^{4} P(A_i|C)$$

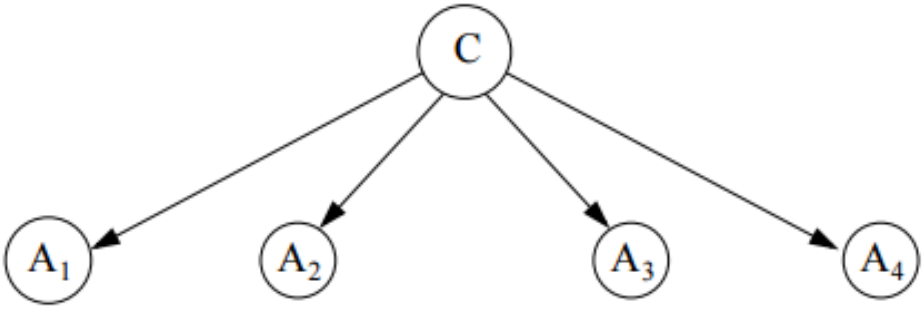


Рисунок 1.1 Наивный байесовский классификатор

Также введём понятие многомерного нормального распределения как:

$$\mathcal{N}(x|\mu,\Sigma) = \frac{1}{\sqrt{|2\pi\Sigma|}}\exp^{-\frac{1}{2}(x-\mu)^\top\Sigma^{-1}(x-\mu)}$$, где:

$\mathbb{E}X = \mu$ – вектор средних значений (вектор мат. ожидания)
$Cov[X] = \Sigma$ – матрица ковариации

Представим таким же способом модель линейной регрессии. Есть три класса веса, данные и метки под данные. Тогда, с учётом того, что все распределения нормальные, имеем:

$$\begin{aligned}
&P(w,y|X) = P(y|X,w)P(w)\\
&P(y|w,X) = \mathcal{N}(y|w^\top X,\sigma^2 I)\\
&P(w) = \mathcal{N}(y|0,\gamma^2 I)
\end{aligned}$$

Тогда, исходя из данного представления, поставив задачу максимизации вероятности весов при условии предоставленных данных, получим следующую задачу оптимизации $P(w|y,X) \to \max\limits_{w}$. Из которой можно получить:

$$\|y - w^\top X\|^2 + \underbrace{\lambda\|w\|^2}_{L_2 \text{ regualizer}} \to \max_w$$

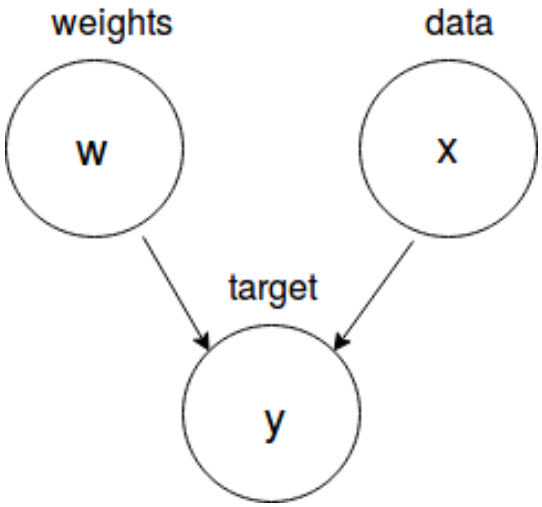


Рисунок 1.2 Модель линейной регрессии

1.2 Сопряжённое априорное распределение.

Априорная вероятность $P(\Theta)$ называется сопряжённой его правдоподобию $P(X|\Theta)$, если априорная вероятность $P(\Theta)$ и апостериорная $P(\Theta|X)$ из одного семейства распределений.

Если априорная вероятность – нормальная и параметризированна некоторыми параметрами μ и σ, то, чтобы она была сопряженной правдоподобию,z мы ожидаем, чтобы апостериорная вероятность была также нормальной, но с уже другими μ и σ. Например, правдоподобие нормальное со средним Θ и дисперсией σ^2 . Если взять априорное распределнеие за нормальное, но с другими параметрами, то после перемножения и нормализацию на определённую константу, мы также получим нормальное распределение.

$$\underbrace{P(\Theta|X)}_{\mathcal{N}(y|a,b^2)} = \frac{\overbrace{P(X|\Theta)}^{\mathcal{N}(y|\Theta,\sigma^2)}\overbrace{P(\Theta)}^{\mathcal{N}(\Theta|m,s^2)}}{\underbrace{P(X)}_{\text{константа}}}$$

Рассмотрим реальный пример. Представим, что правдоподобие нормальное со средним Θ и дисперсией 1 и априорное распределение является стандартным нормальным (среднее - 0, дисперсия - 1), тогда:

$$p(\Theta|x) = \frac{p(x|\Theta)p(\Theta)}{p(x)} = \frac{\mathcal{N}(x|\Theta,1)\mathcal{N}(\Theta|0,1)}{p(x)}$$
$$p(\Theta|x) \propto e^{-\frac{1}{2}(x-\Theta)^2}e^{-\frac{1}{2}\Theta^2}$$
$$p(\Theta|x) \propto e^{-(\Theta-\frac{x}{2})^2}$$
$$p(\Theta|X) = \mathcal{N}(\Theta|\frac{x}{2},\frac{1}{2})$$

ГЛАВА 2
EM-алгоритм

2.1 Модель скрытых переменных.

EM-алгоритм - алгоритм, используемый в математической статистике для нахождения оценок максимального правдоподобия параметров вероятностных моделей, в случае, когда модель зависит от некоторых скрытых переменных. Как правило, EM-алгоритм применяется для решения задач двух типов:

- К первому типу можно отнести задачи, связанные с анализом действительно неполных данных, когда некоторые статистические данные отсутствуют в силу каких-либо причин.

- Ко второму типу задач можно отнести те задачи, в которых функция правдоподобия имеет вид, не допускающий удобных аналитических методов исследования, но допускающий серьезные упрощения, если в задачу ввести дополнительные «ненаблюдаемые» (скрытые, латентные) переменные. Примерами прикладных задач второго типа являются задачи распознавания образов, реконструкции изображений. Математическую суть данных задач составляют задачи кластерного анализа, классификации и разделения смесей вероятностных распределений.

На передовых рубежах глубокого обучения часто требуется построить вероятностную модель входа, $p_{\text{model}}(x)$. В принципе, в такой модели может использоваться вероятностный вывод для предсказания любой переменной в ее окружении при известных других переменных. Во многих моделях имеются также латентные переменные h, так что $p_{\text{model}}(x) = \mathbb{E}_h p_{\text{model}}(x|h)$. Латентные переменные дают еще один способ представления данных. Распределенные представления, основанные на латентных переменных, могут пользоваться всеми преимуществами обучения представлений. 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, задача порождения картинок с изображением цифр. В данной задаче необходимо породить всю картинку целиком так, чтобы она была «посвящена» одной и той же цифре, ведь половина двойки и половина 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 $p(x|z)$ будет уже зависеть от значения z.

2.2 EM-алгоритм

Чтобы применить методы к нейронным сетям, сначала нужно выяснить, как работают байесовские методы. Исходя из того, что рассмотрим байесовское обоснование для одного из основных методов классического обучения без учителя, алгоритм Expectation – Maximization, который обычно называют просто EM-алгоритмом.

Идея EM-алгоритма довольно проста. Например, задача кластеризации на обычной плоскости $\mathbb{R}^2$: существует множество точек, необходимо разбить их на подмножества так, чтобы внутри каждой из них точки были «похожи» друг на друга, то есть они близки друг к другу, и точки из разных подмножеств могли бы быть значительно более различны, то есть находились бы далеко на плоскости.

Предположим, что существует вероятностная модель, определяемая совместным распределением $p(X, T|\Theta)$. Здесь X – набор наблюдаемых переменных, T – набор ненаблюдаемых переменных и Θ – набор параметров модели. Рассмотрим проблему изучения модели (поиск параметров Θ по образцу X) с использованием метода максимального правдоподобия:

$$\log p(X|\Theta) = \log \int p(X, T|\Theta) dT \to \max_{\Theta} \tag{2.1}$$

Рассмотрим произвольное вероятностное распределение $q(T)$. Тогда справедлива следующая цепочка равенств:

$$\log p(X|\Theta) = \int q(T) \log p(X|\Theta) dT = \int q(T) \log \frac{p(X,T|\Theta)}{p(T|X,\Theta)} dT =$$
$$= \int q(T) \log \left[\frac{p(X,T|\Theta)}{q(T)} \frac{q(T)}{p(T|X,\Theta)} \right] dT =$$
$$= \underbrace{\int q(T) \log p(X,T|\Theta) dT - \int q(T) \log q(T) dT}_{\mathcal{L}(q)} - \underbrace{\int q(T) \log \frac{p(T|X,\Theta)}{q(T)} dT}_{KL(q\|p(T|X,\Theta))}$$

Дивергенция $KL(q\|p(T|X,\Theta)) \geq 0$, следовательно:

$$\log p(X|\Theta) \geq \mathcal{L}(q) \tag{2.2}$$

EM-алгоритм для решения задачи (2.1) является аналогом метода Ньютона для оптимизации произвольной функции, где вместо квадратичного приближения в текущей точке используется нижняя оценка (2.2). Пусть некоторое значение параметров Θ_{old} будет фиксированным. Сначала (на шаге E) определяется распределение $q(T)$ как распределение значений скрытых переменных с заданными параметрами:

$$q(T) = p(T|X,\Theta_{\text{old}}) = \frac{p(X,T|\Theta_{\text{old}})}{\int p(X,T|\Theta_{\text{old}}) dT} \tag{2.3}$$

При таком выборе $q(T)$ нижняя оценка (2.2) является точной для $\Theta = \Theta_{\text{old}}$. Затем (на M-шаге) новые значения параметров находят путем максимизации нижней границы $\mathcal{L}(q)$, что эквивалентно решению задачи:

$$\mathbb{E}_{T|X,\Theta_{\text{old}}} \log p(X,T|\Theta) \to \max_{\Theta} \tag{2.4}$$

так как энтропия – $\mathbb{E}_q \log q(T)$ распределения $q(T)$ не зависит от Θ. Шаги E и M повторяются в цикле до сходимости. Очевидно, что в процессе EM-итераций нижняя оценка (2.2), а также значение правдоподобия $p(X|\Theta)$ не убывают. Монотонное неубывание правдоподобия $p(X|\Theta)$ и его ограниченность сверху гарантируют общую сходимость EM-итераций.

Итерационный процесс в EM-алгоритме проиллюстрирован на рисунке 2.1. Нижняя оценка (2.2) обозначена через $\mathcal{L}(q,\Theta)$. По аналогии со многими ите-

рационными методами оптимизации, EM-алгоритм позволяет находить только локальный максимум правдоподобия.

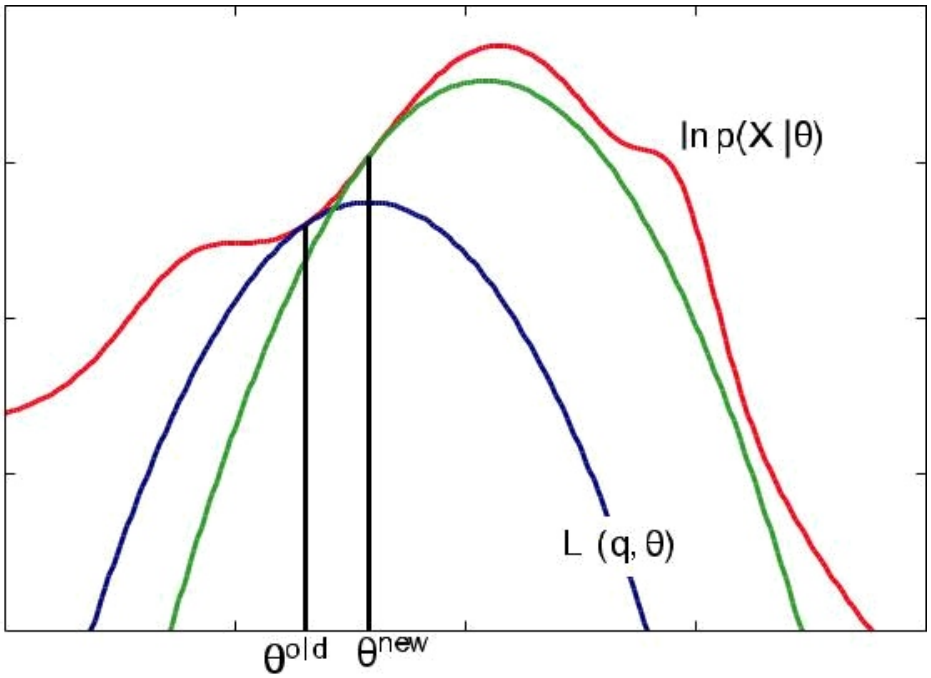


Рисунок 2.1 Итерационный процесс в EM-алгоритме

Заметим, что во многих практических случаях решение задачи 2.4 намного проще, чем решение задачи 2.1. В частности, в рассматриваемой ниже задаче разделения гауссовской смеси задача оптимизации на M шаге может быть решена аналитически.

Вычисление значения функции правдоподобия $p(X|\Theta)$ в фиксированной точке требует интегрирования по пространству T и в ряде случаев может представлять собой вычислительно трудоемкую задачу. Заметим, что эта величина правдоподобия необходима также для вычисления апостериорного распределения $p(T|X, \Theta_{\text{old}})$ на E-шаге. Однако, распределение $p(T|X, \Theta_{\text{old}})$ используется затем только для вычисления математического ожидания логарифма полного правдоподобия на M-шаге. Как правило, здесь не требуется знать все апостериорное распределение целиком, а достаточно знать лишь несколько статистик этого распределения (например, только мат.ожидания отдельных компонент $\mathbb{E}_{T|X,\Theta_{\text{old}}} t_n$ и парные ковариации $\mathbb{E}_{T|X,\Theta_{\text{old}}} t_n t_k$). Поэтому EM-алгоритм может быть вычислительно эффективен даже в тех случаях, когда вычисление значения правдоподобия $p(X|\Theta)$ в одной точке затруднено.

ЕМ-алгоритм можно рассматривать как подъем по координатам для максимизации нижней границы 2.2. Сначала при фиксированных параметрах Θ_{old} нижняя граница максимизируется по распределению $q(T)$ (эта задача имеет аналитическое решение $q(T) = p(T|X, \Theta_{\text{old}})$). Затем для фиксированного $q(T)$ нижняя граница максимизируется по параметрам. Если апостериорное распределение $p(T|X, \Theta_{\text{old}})$ не может быть вычислено, то мы можем ограничить семейство распределений $q(T)$ параметрическим или факторизованным семейством и решить задачу максимизации нижней границы в пределах выбранного семейства распределений. В этом случае будет обеспечен монотонный рост нижнего предела правдоподобия, но, вообще говоря, не самого значения правдоподобия $p(X|\Theta)$. В случае невозможности точного решения задачи на шаге М можно ограничиться только движением в направлении увеличения нижней границы.[1]

ЕМ-алгоритм можно применять также для решения задачи обучения вероятностной модели со скрытыми переменными $p(X, T|\Theta)$ с помощью максимизации апостериорного распределения:

$$p(\Theta|X) \to \max_{\Theta} \Leftrightarrow \log p(X|\Theta) + \log p(\Theta) \to \max_{\Theta}$$

Здесь $p(\Theta)$ — априорное распределение на параметры модели. В этом случае нижней оценкой для оптимизируемого функционала является следующая величина:

$$\log p(X|\Theta) + \log p(\Theta) \geq \mathcal{L}(q) + \log p(\Theta)$$

Итерационная оптимизация данной нижней границы по распределению $q(T)$ и по параметрам приводит к Е-шагу 2.3 и М шагу:

$$\mathbb{E}_{T|X,\Theta_{\text{old}}} \log p(X, T|\Theta) + \log p(\Theta) \to \max_{\Theta}$$

2.3 Применение и примеры ЕМ-алгоритма

Рассмотрим вероятностную модель смеси нормальных распределений:

$$p(x) = \sum_{k=1}^{K} \omega_k \mathcal{N}(x|\mu_k, \Sigma_k), \quad \sum_{k=1}^{K} \omega_k = 1, \quad \omega_k \geq 0 \tag{2.5}$$

Модель смеси распределений (не обязательно нормальных) можно рассматривать как модель со скрытой переменной t, которая обозначает номер компоненты смеси:

$$p(t = k) = \omega_k \tag{2.6}$$
$$p(x|t = k) = \mathcal{N}(x|\mu_k, \Sigma_k) \tag{2.7}$$

Легко показать, что маргинальное распределение $p(x) = \sum_k p(x|t = k)p(t = k)$ в этой модели совпадает с распределением 2.5. В этом смысле модели 2.6 – 2.7 и 2.5 эквивалентны. Интерпретация вероятностной модели смеси распределений как модели со скрытой переменной позволяет генерировать выборку из модели смеси следующим образом. Сначала с вероятностями, равными ω, генерируется номер компоненты смеси, из которой затем генерируется точка x.

Для аппроксимации выборки $X = \{x_1, \ldots, x_N\}$ с помощью модели смеси из K гауссиан можно воспользоваться методом максимального правдоподобия:

$$\begin{aligned} p(X|\omega, \mu_k, \Sigma_k) &= \prod_{n=1}^{N} p(X|\omega, \mu_k, \Sigma_k) \\ &= \prod_{n=1}^{N} \Big(\sum_K \omega_k \mathcal{N}(x_n|\mu_k, \Sigma_k) \Big) \to \max_{\omega, \mu_k, \Sigma_k}, \\ &\sum_{k=1}^{K} \omega_k = 1, \quad \omega_k \geq 0 \\ &\sum\nolimits_K = \sum_K^T, \quad \sum\nolimits_K \succ 0 \end{aligned} \tag{2.8}$$

Данная проблема условной оптимизации может быть эффективно решена с использованием EM-алгоритма, описанного в предыдущем подразделе. Стоит обратить внимание, что число компонентов смеси K не может быть найдено аналогичным образом путем максимизации вероятности, поскольку значение вероятности данных тем выше, чем больше используется компонент K. Чтобы найти оптимальное значение K, можно использовать кросс-валидацию, где критерием качества аппроксимации тестовых данных является значение правдоподобия. [1] При использовании EM-алгоритма для решения задачи 2.8 вероятностная модель смеси распределений интерпретируется как вероятностная модель со скрытыми переменными. Вычислим значение математическое ожидание логарифма полного правдоподобия, необходимого для решения задачи

оптимизации на M шаге:

$$\mathbb{E}_q \log p(X, T|\omega, \{\mu_k\}, \{\Sigma_k\}) = \sum_{n=1}^{N} \sum_{k=1}^{K} q(t_n = k)(\log \omega_k + \log \mathcal{N}(x_n|\mu_k, \Sigma_k)) \quad (2.9)$$

Выражение зависит только от вероятностей отдельных скрытых переменных $q(t_n = k)$. Эти величины можно найти следующим образом:

$$\begin{aligned} \gamma_{nk} \coloneqq q(t_n = k) &= p(t_n = k|x_n, \omega^{\text{old}}, \{\mu_k^{\text{old}}\}, \{\Sigma_k^{\text{old}}\}) = \\ &= \frac{\omega_k^{\text{old}} \mathcal{N}(x_n|\mu_k^{\text{old}}, \Sigma_k^{\text{old}})}{\sum_{j=1}^{K} \omega_j^{\text{old}} \mathcal{N}(x_n|\mu_j^{\text{old}}, \Sigma_j^{\text{old}})} \end{aligned} \quad (2.10)$$

Задача максимизации критерия 2.10 при ограничениях $\sum_{k=1}^{K} \omega_k = 1$, $w_k \geq 0$ может быть решена аналитически:

$$\omega_k = \frac{1}{N} \sum_{n=1}^{N} \gamma_{nk} \quad (2.11)$$

$$\mu_k = \frac{\sum_{n=1}^{N} \gamma_{nk} x_n}{\sum_{n=1}^{N} \gamma_{nk}} \quad (2.12)$$

$$\Sigma_k = \frac{\sum_{n=1}^{N} \gamma_{nk} (x_n - \mu_k)(x_n - \mu_k)^\top}{\sum_{n=1}^{N} \gamma_{nk}} \quad (2.13)$$

Решение для $\sum_K$ 2.13 удовлетворяет условию симметричности и положительной определенности. Кроме того, формулы 2.12, 2.13 соответствуют оценкам максимального правдоподобия для многомерного нормального распределения, в которых каждый объект x_n берется с весом γ_{nk}.

Таким образом, EM-алгоритм для смеси нормальных распределений заключается в итерационном применении формул 2.10 и 2.11-2.13. Этот процесс имеет простую интерпретацию. Величина γ_{nk} показывает степень соответствия между объектом x_n и компонентой k (определяет вес объекта x_n для компоненты k). Эти веса затем используются на M шаге для вычисления новых значений параметров компонент. Пример применения EM-алгоритма для разделения нормальной смеси с двумя компонентами показана на рисунке 2.2. На рисунке 2.2(a) показана исходная выборка и начальное приближение для двух компонент. На

рисунке 2.2(b) показан результат E шага, при этом цвета объектов соответствуют значениям γ_{nk}. На рисунках 2.2(c) – 2.2(f) показаны результаты вычислений после 1, 2, 5 и 20 итераций. [1]

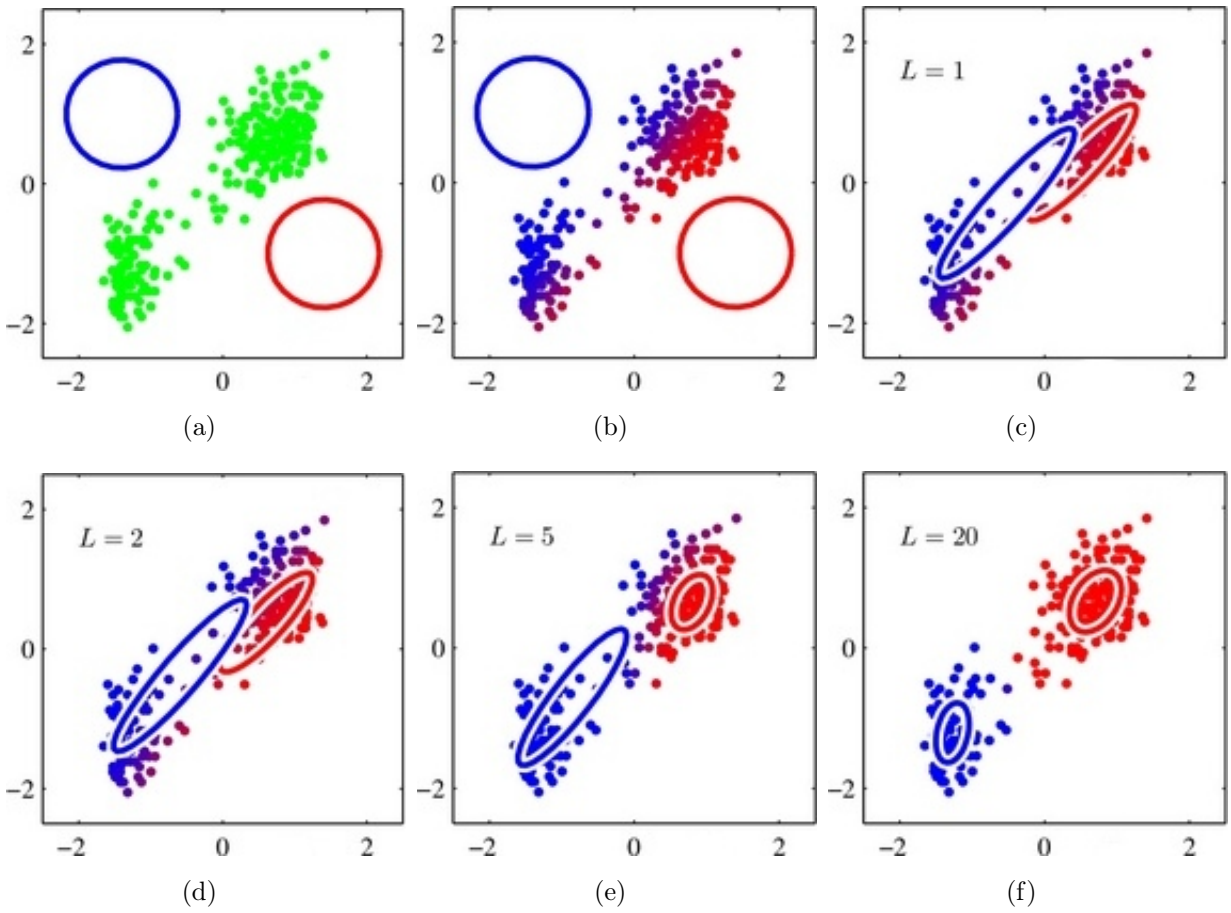


Рисунок 2.2 Применения EM-алгоритма для разделения нормальной смеси

Одним из применений смеси нормальных распределений является решение задачи кластеризации на K кластеров. В этом случае номер кластера для объекта x_n определяется величиной:

$$k_n = \arg\max_k \gamma_{nk}$$

Такая схема кластеризации является вероятностным обобщением известного метода кластеризации K-средних. В заключение этого раздела заметим, что восстановление плотности по данным (в частности, смеси нормальных распределений) является простейшим способом решения задачи идентификации. Для этого сначала для всех объектов обучающей выборки, обладающих заданным

свойством, восстанавливается плотность распределения. Затем, для нового объекта x решение о наличии у него заданного свойства принимается, если значение восстановленной плотности $p(x)$ выше определенного порога.

ГЛАВА 3

Вариоционные автокодировщики

3.1 Введение в вариоционные автокодировщики

Глубокие сверточные нейронные сети (CNN) были использованы для достижения самых современных характеристик во многих контролируемых задачах компьютерного зрения, таких как классификация изображений, поиск, детекция и восстановление. Глубокие генеративные модели на основе CNN, ветвь неконтролируемых методов обучения в машинном обучении, стали горячей темой исследований в области компьютерного зрения в последние годы. Генеративная модель, обученная с данным набором данных, может использоваться для генерации данных, имеющих свойства, аналогичные образцам в наборе данных, для изучения внутренней сущности набора данных и «хранения» всей информации в ограниченных параметрах, которые значительно меньше, чем обучающий набор данных.

Вариационный автокадировщик Variational Autoencoder (VAE) стал популярной порождающей моделью, что позволяет формализовать проблему в рамках вероятностных графических моделей со скрытыми переменными. По умолчанию для измерения разницы между восстановленными и исходными изображениями используется измерение по пикселям, например функция потерь L_2 или функция потерь логистической регрессии. Такие измерения легко осуществимы и эффективны для глубокого обучения нейронной сети. Однако сгенерированные изображения имеют тенденцию быть очень размытыми по сравнению с естественными изображениями. Это связано с тем, что потери по пикселям не отражают разницу в восприятии и пространственную корреляцию между двумя изображениями. Например, одно и то же изображение, смещенное на несколько пикселей, будет иметь незначительное визуальное восприятие для людей, но оно может иметь очень высокую попиксельную функцию потерь. Это хорошо известная проблема в сообществе по измерению качества изображения.

Из литературы по измерению качества изображения известно, что потеря пространственной корреляции является основным фактором, влияющим на

визуальное качество изображения. Недавние работы по синтезу текстур и передаче стиля показали, что скрытые представления глубокой CNN могут захватывать различные свойства пространственной корреляции входного изображения. Воспользуемся этим свойством CNN и пытаемся улучшить VAE, заменив попиксельную потерю на потерю восприятия объекта, которая определяется как разница между скрытыми представлениями двух изображений, извлеченными из предварительно обученной глубокой CNN, такой как AlexNet и VGGNet прошли обучение по ImageNet. Основная идея состоит в том, чтобы попытаться улучшить качество генерируемых изображений VAE путем обеспечения согласованности скрытых представлений входного и выходного изображений, что, в свою очередь, налагает пространственную корреляционную согласованность двух изображений. Также будет показано, что скрытые векторы AVE, обученные с помощью данного метода, обладают мощной концептуальной способностью представления и могут использоваться для достижения самых современных характеристик в прогнозировании характеристик лица.

VAE помогает делать две вещи. Во-первых, это позволяет кодировать изображение x в скрытый вектор $z = \text{Encoder}(x) \sim q(z|x)$ с помощью сети кодера, а затем сеть декодера используется для декодирования скрытого вектора z обратно в изображение, которое будет таким же, как исходное изображение $\overline{x} = \text{Decoder}(z) \sim p(x|z)$. То есть нам нужно максимизировать предельную логарифмическую вероятность каждого наблюдения (пикселя) в x, а потери реконструкции VAE $\mathcal{L}_{\text{rec}}$ - это отрицательная ожидаемая логарифмическая вероятность наблюдений в x. Другим важным свойством VAE является способность управлять распределением скрытого вектора z, который имеет характерную независимость от единичных гауссовских случайных величин, то есть $z \sim \mathcal{N}(0, I)$. Кроме того, различие между распределением $q(z|x)$ и распределением Гаусса (называемое расходимостью дивергенции кульбака-лейблера (KL)) может быть количественно оценено и минимизировано с помощью алгоритма градиентного спуска. Следовательно, модели VAE можно обучать путем оптимизации суммы потерь на восстановление $(\mathcal{L}_{\text{rec}})$ и потерь на расхождение дивергенции кульбака-лейблера $(\mathcal{L}_{\text{kl}})$ путем градиентного спуска.

$$\mathcal{L}_{\text{rec}} = -\mathbb{E}_{q(z|x)}\big[\log p(x|z)\big] \tag{3.1}$$

$$\mathcal{L}_{\text{kl}} = D_{\text{kl}}(q(z|x)\|p(z)) \tag{3.2}$$

$$\mathcal{L}_{\text{vae}} = \mathcal{L}_{\text{rec}} + \mathcal{L}_{\text{kl}} \tag{3.3}$$

Были предложены другие методы для улучшения производительности VAE. Например можно расширить вариационные автокодеры до полууправляемого обучения с помощью меток классов. Другой метод предлагает разнообразные условно-зависимые глубокие вариационные автокодеры и демонстрирует, что они способны генерировать реалистичные лица с разнообразным внешним видом, Deep Recurrent Attentive Writer (DRAW) сочетает в себе механизм пространственного внимания с последовательной вариационной структурой автокодирования, которая позволяет повторять генерацию изображений. Принимая во внимание недостаток попиксельной потери, заменить ее на многомасштабную оценку структурного сходства (MS-SSIM) и демонстрирует, что он может лучше измерять восприятие качества изображения. Также можно усилить целевую функцию дискриминационной регуляризацией. Еще один подход пытается объединить VAE и порождающую состязательную сеть (GAN) и использовать представление изученных признаков в дискриминаторе GAN в качестве основы для цели реконструкции VAE.

3.2 Высокоуровневая пространственная функция потерь (perceptrual loss)

Несколько недавних работ успешно генерируют изображения путем оптимизации потери восприятия, которая основана на высокоуровневых функциях, извлеченных из предварительно обученных глубоких CNN. Передача нейронного стиля и синтез текстуры пытаются совместно минимизировать потери при восстановлении элементов высокого уровня и потери при реконструкции стиля путем оптимизации. Кроме того, изображения могут быть также получены пу-

тем максимизации оценки классификации или отдельных функций. В других работах делается попытка обучить сеть прямой распространения для передачи в режиме реального времени и сверхразрешения на основе потери восприятия функции. В данной главе будет обучаться глубокий сверточный вариационный автокодировщик для генерации изображения, заменяя потерю реконструкции пиксель за пикселем потерей восприятия функции, основанной на предварительно обученной глубокой CNN.

Пространственная функция потерь признаков двух изображений определяется как разница между скрытыми признаками в предварительно обученной глубокой сверточной нейронной сети. Будем использовать VGGNet в качестве сети потерь в эксперименте, который обучен решению задачи классификации в наборе данных ImageNet. Основная идея пространственная функция потерь заключается в поиске согласованности между скрытыми представлениями двух изображений. Поскольку скрытые представления могут охватывать важные характеристики воспринимаемого качества, такие как пространственная корреляция, меньшая разница скрытых представлений указывает на согласованность пространственных корреляций между входом и выходом, в результате мы можем получить лучшее визуальное качество выходного изображения. В частности, пусть $\Phi(x)^l$ представляет вид l-го скрытого слоя, когда входное изображение x подается в сеть Φ. Математически $\Phi(x)$ l - массив трехмерных блоков формы $[C^l \times W^l \times H^l]$, где C^l - количество фильтров, W^l и H^l - ширина и высота каждой карты объектов для l-го слоя соответственно. Потеря восприятия для одного слоя (l) между двумя изображениями x и $\overline{x}$ может быть просто определена квадратом евклидова расстояния. На самом деле это очень похоже на попиксельную функция потерь, за исключением того, что цветовой канал не равен больше 3.

$$\mathcal{L}^l_{\text{rec}} = \frac{1}{2C^l W^l H^l} \sum_{c=1}^{C^l} \sum_{c=1}^{W^l} \sum_{c=1}^{H^l} (\Phi(x)^l_{\text{c,w,h}} - \Phi(\overline{x})^l_{\text{c,w,h}})^2$$

Конечные потери при восстановлении определяются как общие потери путем объединения различных уровней сети VGG, $\mathcal{L}_{\text{rec}} = \sum_l \mathcal{L}^l_{\text{rec}}$. Кроме того, мы принимаем потери дивергенции кульбака-лейблера $\mathcal{L}_{\text{kl}}$, чтобы регулеризировать сеть кодировщиков для управления распределением скрытой переменной

z. Чтобы обучить VAE, мы совместно минимизируем потерю дивергенции KL $\mathcal{L}_{\text{kl}}$ и потерю восприятия $\mathcal{L}^l_{\text{rec}}$ для разных слоев.

$$\mathcal{L}_{\text{total}} = \alpha\mathcal{L}_{\text{kl}} + \beta \sum_{i}^{l} \mathcal{L}^l_{\text{rec}}$$

где α и β - весовые параметры для дивергенции KL и восстановления изображения. Это очень похоже на передачу стиля, если мы рассматриваем дивергенцию кульбака-лейблера как реконструкцию стиля.

3.3 Детали модели автокодировщика

Разрабатываемая система состоит из двух основных компонентов, как показано на рисунке 3.1: сеть автокодера, включающая в себя сеть $E(x)$ кодера и сеть $D(z)$ декодера, и сеть потерь Φ, которая является предварительно обученной глубокой CNN для определения характеристики восприятия потери. Входное изображение x кодируется как скрытый вектор $z = E(x)$, который будет декодирован обратно в пространство изображения $\overline{x} = D(z)$. После обучения сетью декодеров может быть сгенерировано новое изображение с заданным вектором z.

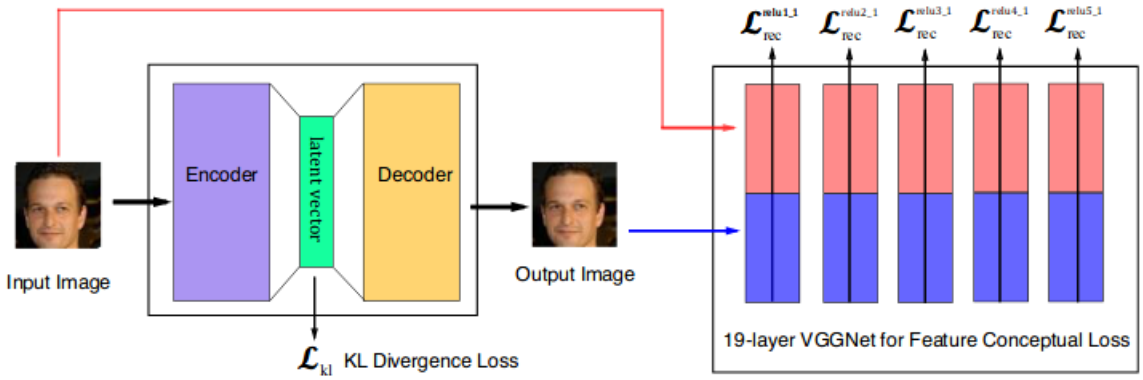


Рисунок 3.1 Модель автоколировщика

Чтобы обучить VAE, нам нужны две потери, одна из которых - потеря дивергенции кульбака-лейблера $\mathcal{L}_{\text{kl}} = D_{\text{kl}}(q(zx)\|p(z))$, которая используется для того, чтобы убедиться, что скрытый вектор z является независимой единицей случайной переменной Гаусса. Другая особенность - пространственная потеря.

Вместо того, чтобы непосредственно сравнивать входное изображение и сгенерированное изображение в пиксельном пространстве, мы подаем их обоих на предварительно обученную глубокую CNN Φ соответственно и затем измеряем разницу между представлениями скрытого слоя, то есть $\mathcal{L}_{\text{rec}} = \mathcal{L}^1 + \mathcal{L}^2 + \cdots + \mathcal{L}^l$, где $\mathcal{L}_l$ представляет потерю объекта на l-м скрытом слое. Таким образом, мы используем потерю признаков высокого уровня, чтобы лучше измерить воспринимаемые и семантические различия между двумя изображениями, потому что предварительно обученная сеть по классификации изображений уже включила воспринимаемую и семантическую информацию, которую мы желали. Во время обучения сеть с предварительно подготовленными потерями является фиксированной и предназначена только для высокоуровневого извлечения признаков, а потеря дивергенции кульбака-лейблера $\mathcal{L}_{\text{kl}}$ используется для обновления сети кодера, в то время как функция потери восприятия функции отвечает за обновление параметров как кодера, так и декодер.

Сеть кодера и декодера основана на глубокой CNN, такой как VGGNet. Создадим 4 сверточных слоя в сети кодера с 4×4 ядрами, и шаг фиксируется равным 2 для достижения пространственной понижающей дискретизации вместо использования детерминированных пространственных функций, таких как maxpooling. Каждый сверточный слой сопровождается слоем batch normalization и слоем активации LeakyReLU. Затем два полностью связанных выходных слоя (для среднего значения и дисперсии) добавляются в кодировщик и будут использоваться для вычисления потерь на дивергенцию кульбака-лейблера и выборки скрытой переменной z. Для декодера мы используем 4 сверточных слоя с ядрами 3×3 и устанавливаем шаг равным 1, а стандартное заполнение нулями заменяем заполнением репликации, то есть карта объектов ввода дополняется репликацией границы входа. Также используется batch normalizationи, чтобы помочь стабилизировать обучение и используем LeakyReLU в качестве функции активации. Детали архитектуры автоэнкодера показаны на рисунке 3.2.

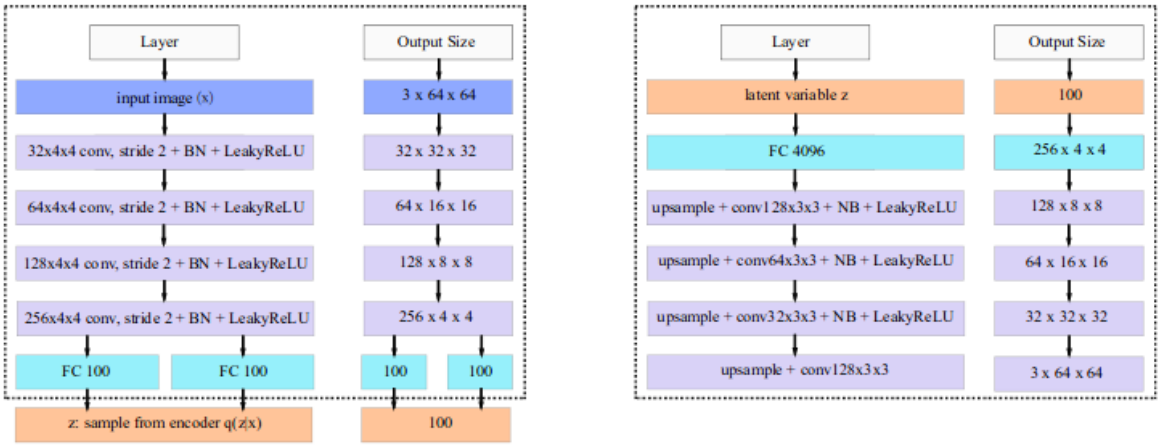


Рисунок 3.2 Архитектура сетей автокодировщика

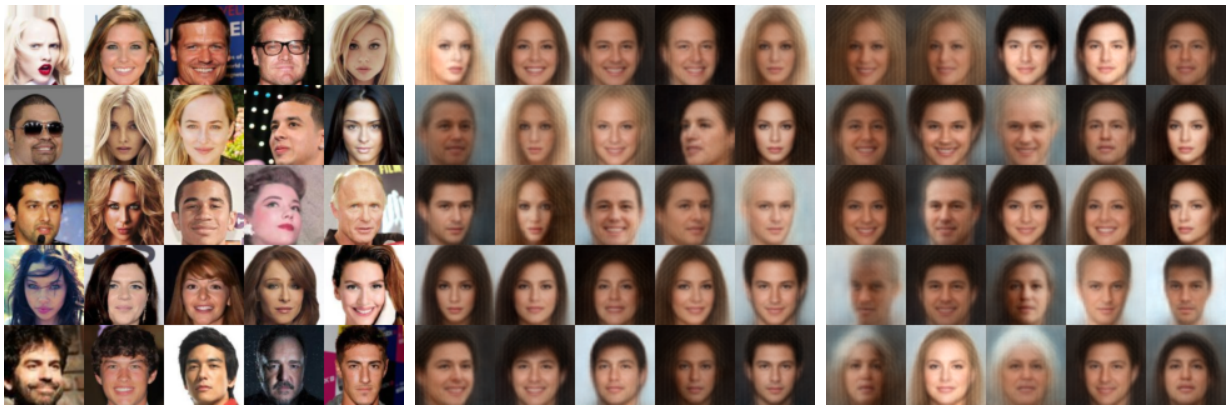
3.4 Результаты и детали обучения

Был проводен эксперимент на изображениях лиц, чтобы проверить метод. В частности, сначала оцениваем производительность генерации изображения и сравниваем с другими генеративными моделями. Кроме того, мы также исследуем скрытое пространство и изучаем семантические отношения между различными скрытыми представлениями и применяем их для прогнозирования характеристик лица.

Модель была обучена на наборе данных CelebFaces Attributes (CelebA). CelebA - это крупномасштабный набор данных атрибутов лица с 202599 изображениями лиц. Набор обучающих данных путем обрезки и масштабирования выровненных изображений до размера 64×64 пикселя. Мы обучаем нашу модель с batch size равным 64 для 5 эпох по набору для обучения и используем метод Adam для оптимизации с начальной скоростью обучения 0,0005, которая уменьшается в 0,5 раза для последующих эпох. 19-слойная сеть VGGNet выбрана в качестве сети потерь Φ для построения потери восприятия функции для восстановления изображения. Были проведены эксперименты с различными комбинациями слоев, чтобы построить пространственную функцию потерь признаков. Соответственно были обучены две модели: VAE-123 и VAE-345, используя слои relu1_1, relu2_1, relu3_1 и relu3_1, relu4_1, relu5_1 соответственно. Кроме того, размер скрытого вектора z установлен равным 100, как DCGAN, а весовые параметры потерь α и β равны 1 и 0,5 соответственно. Реализация

была выполнена во фреймворке TensorFlow.

Результаты обучения разделены на две части: реконструкция лица энкодером и декодером, генерация лица из $\mathcal{N}(0, 1)$ декодером и представлены на рисунках 3.3 и 3.4.



(a) Оригинальные изображения лиц (b) Реконструкция изображений лиц (c) Случайная генерация лиц декодером

Рисунок 3.3 Результаты обучения вариационного автокодировщика (размер скрытого вектора 8)

(a) Оригинальные изображения лиц (b) Реконструкция изображений лиц (c) Случайная генерация лиц декодером

Рисунок 3.4 Результаты обучения вариационного автокодировщика (размер скрытого вектора 100)

ГЛАВА 4

Гауссовский процесс и байесовская оптимизация

4.1 Гауссовский процесс

В настоящее время одним из наиболее популярных методов решения проблемы восстановления (аппроксимации) неизвестной функции путем выборки ее значений является регрессия, основанная на гауссовских процессах.

В рамках этого подхода предполагается, что аппроксимируемая функция является реализацией гауссовского процесса, распределение которого полностью определяется заданной функцией мат. ожидания и функцией дисперсии. Считается, что функция дисперсии между значениями гауссовского процесса зависит только от точек, в которых эти значения получены. Тогда в качестве прогноза, значения аппроксимируемой функции в новой точке используют апостериорное значение мат. ожидания, а для оценки неопределенности этого прогноза - соответствующую апостериорную дисперсию. При этом апостериорное мат. ожидание и апостериорную дисперсию. можно рассчитать аналитически и полностью определить функцией гауссовского процесса.

Обычно предполагается, что функция гауссовского процесса принадлежит некоторому параметрическому семейству, и поэтому задача построения регрессии на основе гауссовских процессов сводится к проблеме оценки параметров самой функции гауссовских процессов.[5]

Стандартные методы, основанные на гауссовских процессах, имеют кубическую сложность размера выборки, что делает невозможным их использование в задачах с большими объемами данных. В связи с этим, начиная с 2000-х годов, многие исследователи разрабатывают приблизительные схемы для обучения моделям гауссовских процессов. Развитие методов, основанных на вспомогательных точках (индуцирующих входных данных), позволило применить гауссовские процессы к задачам с большими выборками (более 100 000 объектов).

Гауссовские процессы определяют априорное распределение по набору функций и позволяют находить сложные взаимосвязи в данных. На основе гауссов-

ских процессов было построено много моделей, которые успешно используются для решения различных задач машинного обучения - регрессии и классификации. Эти методы позволяют автоматически корректировать сложность модели, а также позволяют оценить неопределенность в прогнозе.

Таким образом, гауссовский процесс называется случайным процессом, чьи конечномерные распределения гауссовские.

$$p(\xi(\omega, x_1), \ldots, \xi(\omega, x_n)) = \mathcal{N}(\xi|\mu, \Sigma)$$

На рисунке 4.1 приводится пример гауссовского процесса. Голубая область показывает 3σ - интервал неопределённости процесса (доверительный интервал), а темно-синяя линия — мат. ожидание процесса. Цветные линии показывают случайные реализации процесса.

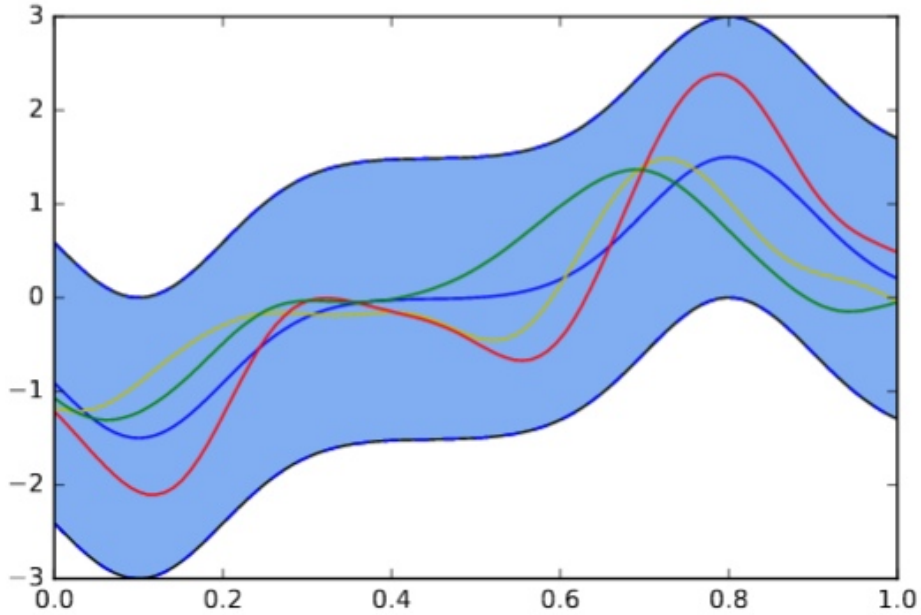


Рисунок 4.1 Одномерный гауссовский процесс

В данной главе рассмотрим гауссовские процессы, заданные на вещественных пространствах $\mathbb{R}^D$. Тогда случайный процесс называется гауссовским процессом, если для любого n, и для любого набора $t_1, t_2, \ldots, t_n \in \mathbb{R}^D$ совместное распределение имеет вид.

$$f(t_1), f(t_2), \ldots, f(t_n) \sim \mathcal{N}(m_t, K_t), \text{ где } m_t \in \mathbb{R}^n,\ K_t \in \mathbb{R}^{n \times n}$$

Математическое ожидание определяется функцией среднего $m : \mathbb{R}^D \to \mathbb{R}$ гауссовского процесса:

$$m_t = (m(t_1), m(t_2), \ldots, m(t_n))^\top$$

Данная функция может быть произвольной.

Аналогично, матрица ковариации K_t определяется ковариационной (ядровой) функцией процесса $k : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$.

$$K_t = \begin{pmatrix} k(t_1, t_1) & k(t_1, t_2) & \ldots & k(t_1, t_n) \\ k(t_2, t_1) & k(t_2, t_2) & \ldots & k(t_2, t_n) \\ \ldots & \ldots & \ldots & \ldots \\ k(t_n, t_1) & k(t_n, t_2) & \ldots & k(t_n, t_n) \end{pmatrix}$$

Матрица ковариации для любого набора значений $t_1, t_2, \ldots, t_n$ должна быть симметричной и неотрицательно определенной.

Гауссовский процесс полностью определяется своими функцией математического ожидания и ковариационной функцией. Для гауссовского процесса будем использовать запись:

$$f \sim \mathcal{GP}(m(\cdot), k(\cdot, \cdot))$$

с функиец среднего m и ядровой функцией k.

Гауссовские процессы являются довольно гибким средством описания данных, а степень “гладкости” процесса определяется видом ковариационной функции. На рисунке 4.2 представлены примеры реализаций стационарных гауссовских случайных процессов с различными ковариационными функциями.

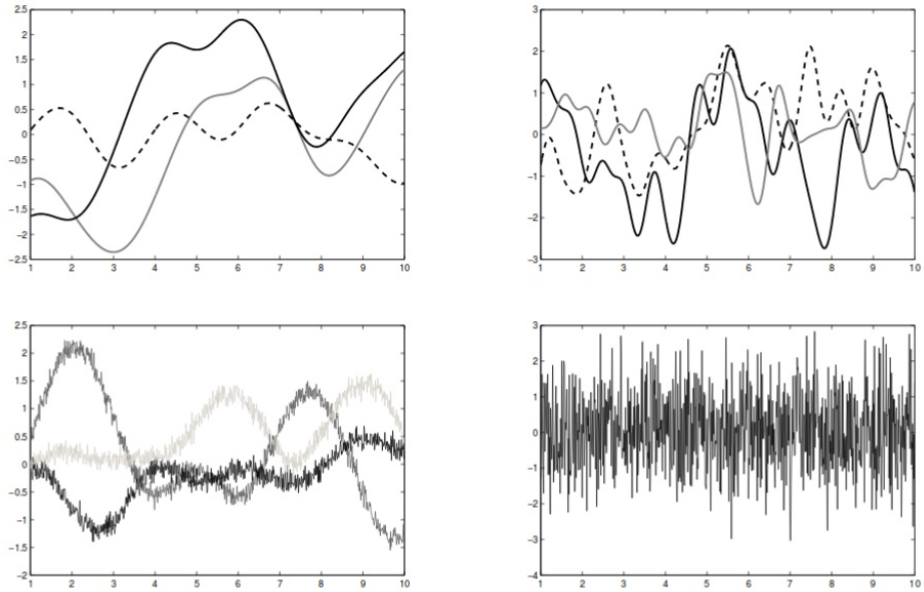


Рисунок 4.2 Реализаций стационарных гауссовских случайных процессов с различными ковариационными функциями

Рассмотрим регрисионную модель на основе гауссовских процессов. Пусть $X = (x_1, x_2, \ldots, x_n)^\top \in \mathbb{R}^{n \times D}$ - описание выборки из n объектов. Пусть $Y = (y_1, y_2, \ldots, y_n)^\top \in \mathbb{R}^n$ - соответствующие им значения. Будем считать, что наблюдаемые переменные y - зашумленные значения некоторого скрытого гауссовского процесса $f : \mathbb{R}^D \to \mathbb{R}$, с нулевым математическим ожиданием и ковариационной функцией $k(\cdot, \cdot)$.

$$f \sim \mathcal{GP}(0, k(\cdot, \cdot))$$

Введем скрытые переменные $f = (f_1, f_2, \ldots, f_n) \in \mathbb{R}^n$ значения процесса f в точках обучающей выборки. Тогда

$$p(y_i|f_i) = \mathcal{N}(y_i|f_i, \nu^2)$$

где ν – разброс шума. Требуется оценить неизвестное значение процесса $f_* \in \mathbb{R}^l$ в наборе новых точек $X_* \in \mathbb{R}^{l \times D}$. Рассмотрим модель:

$$p(y, f|X) = p(y|f)p(f|X) = p(f|X) \prod_{i=1}^{n} p(y_i|f_i) \tag{4.1}$$

Графическая модель для данной задачи изображена на рисунок 4.3.

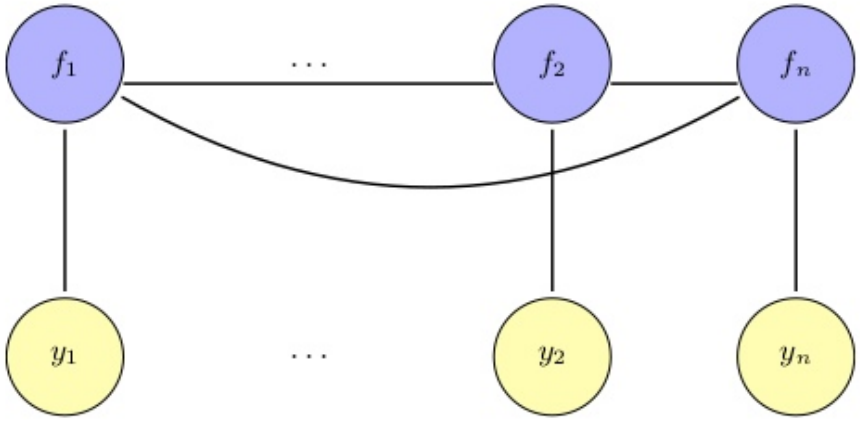


Рисунок 4.3 Графическая модель для модели регрессии на основе гауссовских процессов

Обозначим матрицу попарных значений ковариационной функции, вычисленную на двух наборах точек $A = (a_1, a_2, \dots, a_n)^\top \in \mathbb{R}^{n \times D}$ и $B = (b_1, b_2, \dots, b_m)^\top \in \mathbb{R}^{m \times D}$ как:

$$K(A, B) = \begin{pmatrix} k(a_1, b_1) & k(a_1, b_2) & \dots & k(a_1, b_m) \\ k(a_2, b_1) & k(a_2, b_2) & \dots & k(a_2, b_m) \\ \dots & \dots & \dots & \dots \\ k(a_n, b_1) & k(a_n, b_2) & \dots & k(a_n, b_m) \end{pmatrix} \in \mathbb{R}^{n \times m}$$

Тогда по определению гауссовского процесса распределение

$$\begin{bmatrix} f \\ f_* \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} K(X, X) & K(X, X_*) \\ K(X_*, X) & K(X_*, X_*) \end{bmatrix}\right)$$

Так как y представляет из себя зашумленную версию f , то

$$\begin{bmatrix} y \\ f_* \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} K(X, X) + \nu^2 I & K(X, X_*) \\ K(X_*, X) & K(X_*, X_*) \end{bmatrix}\right)$$

Можем выразить условное распределение на f_* при условии данных

$$f_* | y \sim \mathcal{N}(\hat{m}, \hat{K})$$

где

$$\begin{aligned} &\mathbb{E}[f_*|y] = \hat{m} = K(X_*, X)(K(X, X) + \nu^2 I)^{-1} y \\ &cov(f_*|y) = \hat{K} = K(X_*, X_*) - K(X_*, X)(K(X, X) + \nu^2 I)^{-1} K(X, X_*) \end{aligned} \tag{4.2}$$

Таким образом, сложность получения распределения на f_* определяется сложностью обращения матрицы $K(X, X) + \nu^2 I \in \mathcal{R}^{n \times n}$ и имеет асимптотику $\mathcal{O}(n^3)$.

На рисунке 4.4 показан пример предсказательного распределения для гауссовского процесса с $\nu = 0$, настроенного по трем точкам, обозначенным синими кругами. Голубая область показывает 3σ-интервал для значений процесса, а темно-синяя линия — его математическое ожидание. Цветные линии показывают случайные реализации процесса.

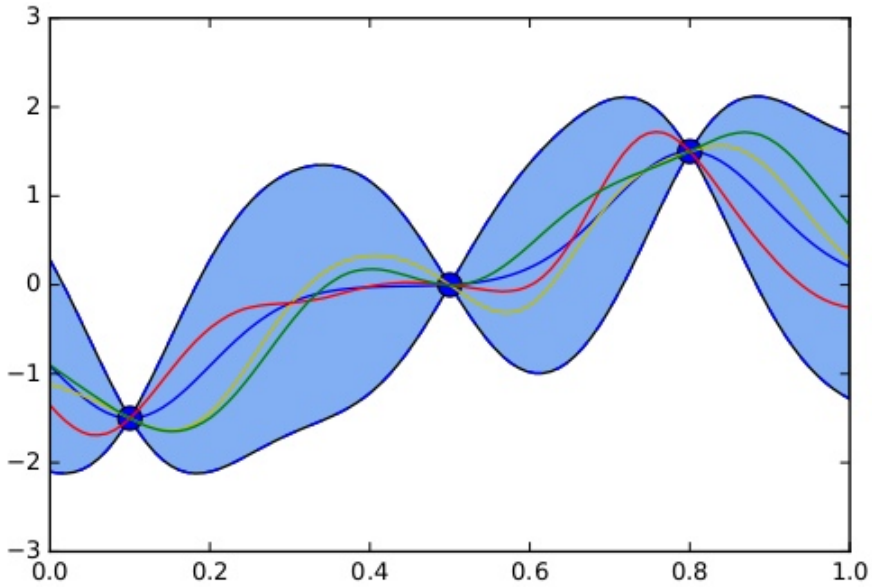


Рисунок 4.4 Предсказательное распределение в задаче регрессии

4.2 Байесовская оптимизация

Байесовская оптимизация (Bayes Optimization) представляет собой класс методов оптимизации на основе машинного обучения, ориентированных на решение задачи.

$$\max_{x \in A} f(x) \tag{4.3}$$

где допустимое множество и целевая функция обычно имеют следующие свойства:

- x находится в $\mathbb{R}^d$, где значени d не слишком велико. Обычно при $d \leq 20$ наиболее успешные приложения байесовской оптимизации,
- целевая функция f непрерывна. Обычно это требуется для моделирования f с использованием регрессии на основе гауссовского процесса,
- f «дорого вычислять» в том смысле, что число вычислений, которые могут быть выполнены, ограничено, как правило, несколькими сотнями. Это ограничение обычно возникает из-за того, что каждое вычисление значения функции занимает значительное количество времени (обычно часов), но также может происходить из-за того, что данное вычисление может нести денежные затраты (например, от покупки мощности облачных вычислений или покупки лабораторных материалов) или альтернативные затраты (например, если вычисление f требует задавать человеку вопросы, которые будут терпеть только ограниченное число раз),
- f, по своей природе, не является вогнутой или линейной, хотя данные характеристики позволили бы упростить оптимизацию с использованием известных и класиических методов, которые используют данные свойства для повышения эффективности,
- когда происходит вычисление f, нет возможности получить никаких производных первого или второго порядка. Это отрицате применение методов оптимизации первого и второго порядка, таких как градиентный спуск, метод Ньютона или квазиньютоновские методов. Проблемы с данным свойством называются «без производных»,
- $f(x)$ наблюдается без шума. Однако почти во всех работах по байесовской оптимизации шум считается независимым от вычислений и гауссовым с постоянной дисперсией (обычно стандартное гауссовое распределение),
- внимание акцентировано на поиске глобального, а не локального оптимума.

Байесовская оптимизация состоит из двух основных компонентов: байесовской статистической модели для моделирования целевой функции (обычно гауссовский процесс) и функции оценки (acquisition function) для принятия решения о том, где искать оптимум дальше. После вычисления целевой функции в соответствии с первоначальными данными, просиходит вычисление Гауссовского процесса и его обучения для подбора параметров функции ковариционной (ядровой) функции. После этого составляется функция оценки на основе полученного Гауссовского процесса, находится оптимум данной функции оценки, вычисляется значение целевой функции в данном оптимуме, после чего данное значение добавляется в набор данных. Процесс повторяется снова до определённой сходимости или после вычисления определённого количества итераций. Всё вышеописанное обобщается в алгоритм.

Одна итерация байесовской оптимизации с использованием регрессии гауссовского процесса и функцией оценки ожидаемого улучшения (Expected improvement) проиллюстрирована на рисунке 4.5. На верхней панели показаны бесшумные значения целевой функции с синими кружками в трех точках. Он также показывает результат регрессии гауссовского процесса. Регрессия гауссова процесса дает апостериорное распределение вероятностей для каждого $f(x)$, которое обычно распределяется со средним значением $\mu_n(x)$ и дисперсией $\sigma_n^2(x)$. Это изображено на рисунке 4.5 с $\mu_n(x)$ в виде сплошной красной линии и 95 % байесовским доверительным интервалом для $f(x)$, $\mu_n(x) \pm 1,96 \times \sigma_n(x)$ в виде пунктирных красных линий. Среднее можно интерпретировать как точечную оценку $f(x)$. Достоверный интервал действует как доверительный интервал в статистике частых случаев и содержит $f(x)$ с вероятностью 95 % в соответствии с последним распределением. Среднее значение интерполируется ранее данным точкам (начальным данным). В этих точках доверительный интервал имеет ширину 0 и расширяется по мере удаления от них.[3]

Нижний рисунок показывает функцию оценки ожидаемого улучшение (Expected improvement) функции сбора, которая соответствует данному апостериору Гауссовского процесса. Стоит обратить внимание, что она принимает значение 0 в точках, которые были ранее вычислены. Это приемлимо, когда значения целевой функции не имеют шума, потому что они не дают полезной информации для решения. Также стоит обратить внимание, что она имеет тенденцию быть

больше для точек с большими доверительными интервалами, потому что значения точки, где Гауссовский процесс более не уверен в цели, имеет тенденцию быть более полезным в поиске хороших приближенных глобальных оптимумов. Стоит обратить внимание, что она имеет тенденцию быть больше для точек с большим апостериорным средним, потому что такие точки, как правило, близки к хорошим приблизительным глобальным оптимумам.

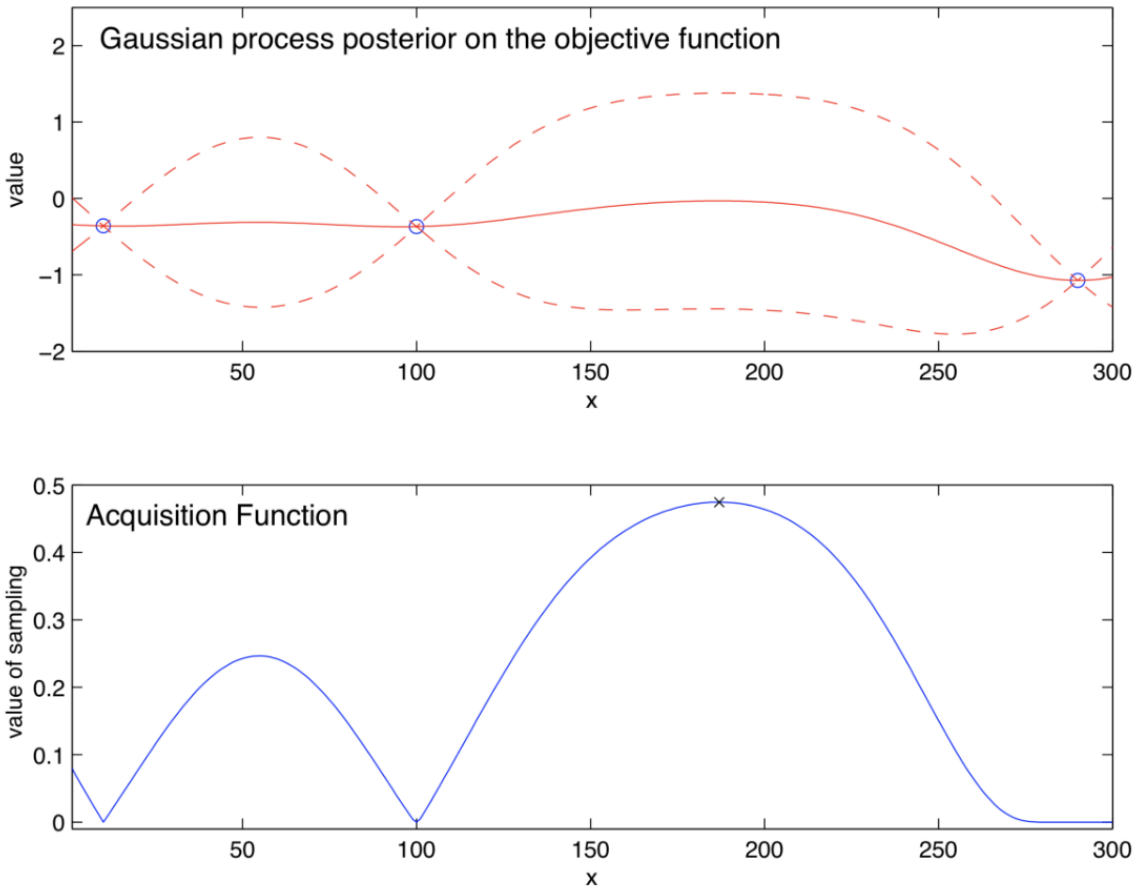


Рисунок 4.5 Предсказательное распределение в задаче регрессии

Функция оценки ожидаемого улучшение (Expected improvement) выводится из эксперимента. Допустим использование алгоритма байесовской оптимизации для решения (4.3), в котором x_n обозначает точку, выбранную на итерации n, а y_n обозначает соответствующее ей значение. Допустим решение как наше окончательное решение (4.3). Также допустим, что нет возможности получить значение функции, которое необходимо, и решение должно быть основано на тех, которые уже выполнены. Поскольку мы вычисляем f без шума, оптимальным решением является предварительно вычисленная точка с наибольшим получен-

ным значением. Пусть $f_n^* = \max_{m \le n} f(x_m)$ будет значением этой точки, где n - количество итераций.

Теперь предположим, что на самом деле есть возможность выполнить одно дополнительное вычисление, и можно его выполнить где угодно и когда угодно. Если вычислить в x, получится значение $f(x)$. После получения нового значения лучшей точки должно быть либо $f(x)$ (если $f(x) \ge f_n^*$), либо f_n^* (если $f(x) \le f_n^*$). Улучшение $f(x) - f_n^*$ положительное или 0 в противном случае. Тогда данное улучшение можно записать более компактно как $[f(x) - f_n^*]^+$, где $a^+ = \max(a, 0)$.

Необходимо выбрать x, чтобы данное улучшение было большим, так как $f(x)$ неизвестно до окончания вычисления. Однако можно взять мат. ожидание этого улучшения и выбирать x, относительно максимизации мат. ожидания. Определяем мат. ожидание как.

$$EI_n(x) \coloneqq E_n[[f(x) - f_n^*]^+]$$

Здесь $E_n[\cdot] = E_n[\cdot | x_{1:n}, y_{1:n}]$ указывает мат. ожидание апостериорного распределения, данное при вычислении f в $x_1, \ldots, x_n$. Это апостериорное распределение $f(x)$ задаётся формулой (4.2).

Ожидаемое улучшение может быть вычислено с помощью формулы интегрирования по частям. Опустив вычисления, получим.

$$EI_n(x) = var[\hat{f}_n(x)](z_n \Phi(z_n) + \Phi(z_n))$$

где

$$z_n = \frac{\mathbb{E}\hat{f}_n(x) - m(x)}{var[\hat{f}_n(x)]},$$

Φ – функция плотности нормального распределения (функция распределения), $\hat{f}_n$ – статистическая модель на n-ой итерации (Гауссовский процесс).

Затем решается задача оптимизации:

$$x_{n+1} = \operatorname{argmax} EI_n(x) \tag{4.4}$$

В отличие от целевой функции f в нашей первоначальной задаче оптимизации (4.3), $EI_n(x)$ является недорогой для вычисления и позволяет легко

оценить производные первого и второго порядка. Реализации алгоритма ожидаемого улучшения могут также использовать непрерывный методы оптимизации первого или второго порядка для решения (4.4). Например, один метод, который хорошо зарекомендовал себя для автора, - это вычислить первые производные и использовать квазиньютоновский метод L-BFGS-B (Liu and Nocedal, 1989).

На рисунке 4.6 показан график линий уровня $EI_n(x)$ в терминах $\triangle_n(x) = \mathbb{E}\hat{f}(x) - m(x)$ и апостериорного стандартного отклонения $\sigma_n(x) = var[\hat{f}_n(x)]$. $EI_n(x)$ увеличивается при увелечении и $\triangle_n(x)$, и $\sigma_n(x)$. Кривые $\triangle_n(x)$ и $\sigma_n(x)$ с равным EI показывают, как EI балансирует между оценкой в точках с высоким ожидаемым улучшением (высоким $\triangle_n(x)$)) и высокой неопределенностью (высоким $\sigma_n(x)$). В контексте оптимизации вычисление в точках с высоким ожидаемым улучшением относительно предыдущей наилучшей точки является наиболее важные, потому что хорошие приблизительные глобальные оптимумы, вероятно, будут находиться около таких точек. С другой стороны, вычисление в точках с высокой неопределенностью перспективна, потому что поиск оптимума просихожит в тех местах, о которых мало известно Гауссовскому процессу (высокий разброс значений) и которые, как правило, находятся далеко от того, что ранне было вычислено. Оптимум, который может быть значительно лучше, чем тот, которую уже есть, вполне возможно может быть в тех местах. [3]

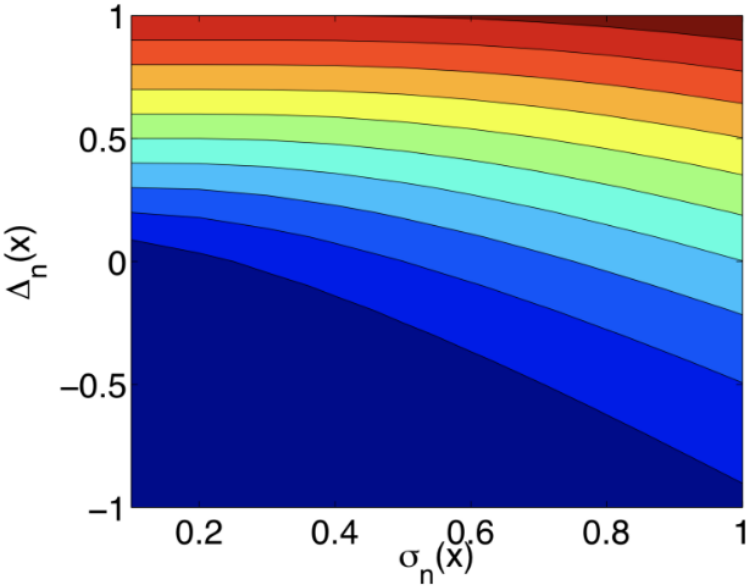
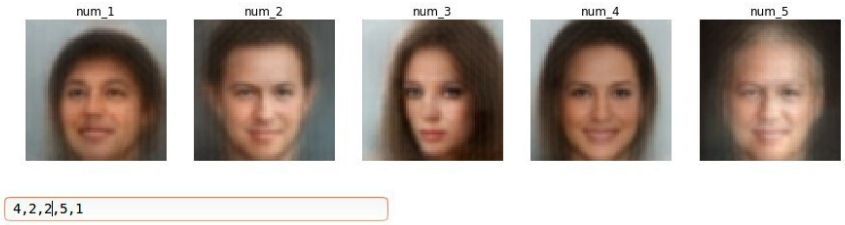


Рисунок 4.6 График линий уровня $EI(x)$

Рисунок 4.5 показывается $EI()$ на нижнем графике. Можно отследить поведение данной функции, где наибольшее ожидаемое улучшение, ищется там, где высокое стандартное отклонение (далеко от ранее вычисленных точек) и высокое мат. ожидание. Наименьшее ожидаемое улучшение - 0 в точках, где известны значения целевой функции. Стандартное отклонение в этих точках равны 0, а мат. ожидание обязательно не больше, чем лучшая ранее вычисленный оптимум.

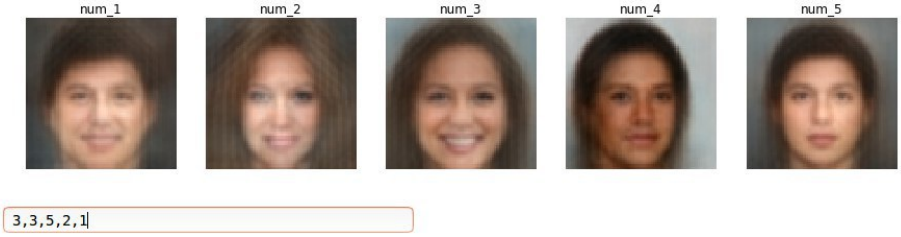
В данной работе было предложено подстраивать значения скрытого вектора z декодера автокодировщика под генерацию лица наперёд заданным визуальным характеристикам при помощи Байесовской оптимизации. Задача пользователя заключается в оценивании списка сгенерированных (по определённым скрытым векторам) лиц по 5-бальной шкале по заданной характеристике (больше-лучше). Затем байесовская оптимизация пытается предложить наиболее лучший скрытый вектор z (необязательно из существующих в списке сгенерированных, но он может быть очень похож на один из существующих), которые отправляется в декодер автокодировщике и из этого вектора декодируется лицо с наиболее лучшей визуальной характеристикой. Целевая функция для данного типа оптимизации задавалась как $\frac{\text{mark}}{\max(\text{mark})}$. Веса модели автокодировщика взяты с размером скрытого вектор равного 8. Результаты представлены на рисунках 4.7 и 4.8.



(a) Список сгенерированных лиц и их оценки

(b) Результат выбора лица с наиболее тёмными волосами

Рисунок 4.7 Результат применения Байесовской оптимизации к подбору значений скрытого вектора декодера автокодировщика для характеристики наиболее тёмных волос.



(a) Список сгенерированных лиц и их оценки

(b) Результат выбора лица с наиболее широкой улыбкой

Рисунок 4.8 Результат применения Байесовской оптимизации к подбору значений скрытого вектора декодера автокодировщика для характеристики наиболее широкой улыбки.

ЗАКЛЮЧЕНИЕ

В ходе работы были исследованы новейшие принципы генерации изображений на основе автокодировщиков, а также особенный тип обучения моделей данного типа. Также были изучены основные принципы байесовской оптимизации и регрессии Гауссовского процесса.

Результатом работы стало обученный на датасете лиц знаменитостей celebA автокодировщик, способный генерировать лица несуществующих людей. Тип обучения сопровождался специальной пространственной функцией потери (perceptrual loss). Также результатом стало некоторое приложение основанное на байесовской оптимизации, которое способно итеративно подбирать лицо по заданным характеристикам.

В перспективе ставятся задачи улучшения способа поиска оптимальных областей, значений у целевой функции на основе Байесовской оптимизации при помощи измения типа целевой фунции и способа оценивания изображений, а также улучшение качества работы модели автокодировщика путём усовершенствования архитектуры моделей и усложнения пространственной функции потери (perceptrual loss function).

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

[1] Ветров Д. П. Курс лекций байесовские методы машинного обучения / Ветров Д. П., Кропотов Д. А.

[2] Xianxu Hou. Deep Feature Consistent Variational Autoencoder / Xianxu Hou, Linlin Shen, Ke Sun, Guoping Qiu

[3] Peter I. Frazier. A Tutorial on Bayesian Optimization

[4] Николенко С. Глубокое обучение / Николенко С., Кадурин А., Архангельская Е. — СПб: Питер, 2018. — 480 с.

[5] Измаилов П. А. Выпускная квалификационная работа. Алгоритмы обучения гауссовских процессов для больших объемов данных / Измаилов П. А., Ветров Д. П., Кропотов Д. А.

[6] Bayesian Methods in Machine Learning [Электронный ресурс] — Режим доступа: https://www.coursera.org/learn/bayesian-methods-in-machine-learning/home/welcome

ПРИЛОЖЕНИЕ А

Код программы. Модель автокодировщика

```
import numpy as np
import tensorflow as tf
from tf_vae.vgg16 import vgg16

class dfc_vae_model(object):

    def __init__(self, shape, inputs, alpha = 1, beta = 0.5, vgg_layers = [], learning_r
        self.shape = shape
        self.img_input = inputs
        self.alpha = alpha
        self.beta = beta
        self.gstep = tf.Variable(0, dtype=tf.int32, trainable=False, name='global_step')
        self.vgg_layers = vgg_layers
        self.learning_rate = learning_rate

    def _get_weights(self, name, shape):
        with tf.variable_scope("weights", reuse=tf.AUTO_REUSE) as scope:
            w = tf.get_variable(name=name + '_W',
                                shape=shape,
                                initializer=tf.truncated_normal_initializer(stddev=0.1))
        return w

    def _get_biases(self, name, shape):
        with tf.variable_scope("biases", reuse=tf.AUTO_REUSE) as scope:
            b = tf.get_variable(name=name + '_b',
                                shape=shape,
                                initializer=tf.truncated_normal_initializer(stddev=0.1))
        return b

    def _conv2d_bn_relu(self, inputs, name, kernel_size, in_channel, out_channel, stride
        with tf.variable_scope(name) as scope:

            ### setup weights and biases
            filters = self._get_weights(name, shape=[kernel_size, kernel_size, in_channe
            biases = self._get_biases(name, shape=[out_channel])

            ### convolutional neural network
            conv2d = tf.nn.conv2d(input=inputs,
                                  filter=filters,
                                  strides=[1,stride,stride,1],
                                  padding='SAME',
                                  name=name + '_conv')
            conv2d = tf.nn.bias_add(conv2d, biases, name=name+'_add')

            ### in case of batch normalization
            if bn == True:
                conv2d = tf.contrib.layers.batch_norm(conv2d,
                                                      center=True, scale=True,
```

```
                                            is_training=True,
                                            scope='bn')

        ### in case of leaky relu activation
        if activation == True:
            conv2d = tf.nn.leaky_relu(conv2d, alpha=0.1, name=name)

    return conv2d

def encoder(self, reuse=False):

    with tf.variable_scope("encoder", reuse = reuse):
        ### Conv2d_bn_relu Layer 1
        conv1 = self._conv2d_bn_relu(self.img_input,
                                     name="conv1",
                                     kernel_size=4,
                                     in_channel=3,
                                     out_channel=32,
                                     stride=2)

        ### Conv2d_bn_relu Layer 2
        conv2 = self._conv2d_bn_relu(conv1,
                                     name="conv2",
                                     kernel_size=4,
                                     in_channel=32,
                                     out_channel=64,
                                     stride=2)

        ### Conv2d_bn_relu Layer 3
        conv3 = self._conv2d_bn_relu(conv2,
                                     name="conv3",
                                     kernel_size=4,
                                     in_channel=64,
                                     out_channel=128,
                                     stride=2)

        ### Conv2d_bn_relu Layer 4
        conv4 = self._conv2d_bn_relu(conv3,
                                     name="conv4",
                                     kernel_size=4,
                                     in_channel=128,
                                     out_channel=256,
                                     stride=2)

        ### flatten the output
        conv4_flat = tf.reshape(conv4, [-1, 256*4*4])

        ### FC Layer for mean
        fcmean = tf.layers.dense(inputs=conv4_flat,
                                 units=8,
                                 activation=None,
                                 name="fcmean")

        ### FC Layer for standard deviation
        fcstd = tf.layers.dense(inputs=conv4_flat,
                                units=8,
```

```
                                    activation=None,
                                    name="fcstd")

        ### fcmean and fcstd will be used for sample z value (latent variables)
        return fcmean, fcstd + 1e-6

    def decoder(self,inputs, reuse=False):

        with tf.variable_scope("decoder", reuse = reuse):
            ### FC Layer for z
            fc = tf.layers.dense(inputs=inputs,
                                 units = 4096,
                                 activation = None)
            fc = tf.reshape(fc, [-1, 4, 4, 256])

            ### Layer 1
            deconv1 = tf.image.resize_nearest_neighbor(fc, size=(8,8))
            deconv1 = self._conv2d_bn_relu(deconv1,
                                    name="deconv1",
                                    kernel_size=3,
                                    in_channel=256,
                                    out_channel=128,
                                    stride=1)

            ### Layer 2
            deconv2 = tf.image.resize_nearest_neighbor(deconv1, size=(16,16))
            deconv2 = self._conv2d_bn_relu(deconv2,
                                    name="deconv2",
                                    kernel_size=3,
                                    in_channel=128,
                                    out_channel=64,
                                    stride=1)

            ### Layer 3
            deconv3 = tf.image.resize_nearest_neighbor(deconv2, size=(32,32))
            deconv3 = self._conv2d_bn_relu(deconv3,
                                    name="deconv3",
                                    kernel_size=3,
                                    in_channel=64,
                                    out_channel=32,
                                    stride=1)

            ### Layer 4
            deconv4 = tf.image.resize_nearest_neighbor(deconv3, size=(64,64))
            deconv4 = self._conv2d_bn_relu(deconv4,
                                           name="deconv4",
                                           kernel_size=3,
                                           in_channel=32,
                                           out_channel=3,
                                           stride=1,
                                           activation=False,
                                           bn=False)

        return deconv4

    def load_vgg(self):
```

```
        ### pass the input image to VGG model
        self.resize_input_img = tf.image.resize_images(self.img_input, [224,224])
        self.vgg_real = vgg16(self.resize_input_img, 'vgg16_weights.npz')
        self.l1_r, self.l2_r, self.l3_r = self.vgg_real.get_layers()

        self.resize_gen_img = tf.image.resize_images(self.gen_img, [224,224])
        self.vgg_gen = vgg16(self.resize_gen_img, 'vgg16_weights.npz')
        self.l1_g, self.l2_g, self.l3_g = self.vgg_gen.get_layers()

    def calculate_loss(self):

        ### calculate perception loss
        l1_loss = tf.reduce_sum(tf.square(self.l1_r-self.l1_g), [1,2,3])
        l2_loss = tf.reduce_sum(tf.square(self.l2_r-self.l2_g), [1,2,3])
        l3_loss = tf.reduce_sum(tf.square(self.l3_r-self.l3_g), [1,2,3])
        self.pct_loss = tf.reduce_mean(l1_loss + l2_loss + l3_loss)

        ### calculate KL loss
        self.kl_loss = tf.reduce_mean(-0.5*tf.reduce_sum(
            1 + self.std - tf.square(self.mean) - tf.exp(self.std), 1))

        ### calculate total loss
        self.loss = tf.add(self.beta*self.pct_loss,self.alpha*self.kl_loss)

    def optimize(self):

        ### create optimizer
        var_list = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope='encoder') + t
        self.optimizer = tf.train.AdamOptimizer(self.learning_rate).minimize(self.loss,g

    def build_model(self,reuse=tf.AUTO_REUSE):
        ### get mean and std from encoder
        self.mean, self.std = self.encoder(reuse)
        ### sampling z and use reparameterization trick
        epsilon = tf.random_normal((tf.shape(self.mean)[0],8), mean = 0.0, stddev=1.0)
        self.z = self.mean + epsilon * tf.exp(.5*self.std)
        ### decode to get a generated image
        self.gen_img = self.decoder(self.z,reuse)
        ### load vgg
        self.load_vgg()
        ### calculate loss
        self.calculate_loss()
        ### setup optimizer
        self.optimize()
        ### generate random latent variable for random images
        self.random_latent = tf.random_normal((tf.shape(self.mean)[0], 8))
        self.ran_img = self.decoder(self.random_latent,reuse)

    ### load VGG weight
    def load_vgg_weight(self, weight_file, sess):
        self.vgg_real.load_weights(weight_file,sess)
        self.vgg_gen.load_weights(weight_file,sess)
```

ПРИЛОЖЕНИЕ Б

Код программы. Обучение автокодировщика

```
import dfc_vae_model as dfc
import tensorflow as tf
import numpy as np
import cv2
import scipy.misc
import urllib
from zipfile import ZipFile
from PIL import Image
import matplotlib.pyplot as plt
import os
import sys
import imageio

############## Hyper-Parameters #################
### Adjust parameters in this part
BATCH_SIZE = 32
NUM_EPOCH = 10
VGG_LAYERS = ['conv1_1','conv2_1','conv3_1']
ALPHA = 1
BETA = 8e-6
LEARNING_RATE = 0.0001
IMG_HEIGHT = 64
IMG_WIDTH = 64
TRAINING_DATA = 'celeb_data_tfrecord'
IMG_MEAN = np.array([134.10714722, 102.52040863, 87.15436554])
IMG_STDDEV = np.sqrt(np.array([3941.30175781, 2856.94287109, 2519.35791016]))
#################################################


### get the image
def crop_center_image(img):
    width_start = int(img.shape[1]/2 - 150/2)
    height_start = int(img.shape[0]/2 - 150/2)
    cropped_img = img[height_start: height_start+150, width_start: width_start+150, :]
    #print(cropped_img.shape)
    return cropped_img

### download according to address provided and perform cropping
def load_and_crop_image(img, img_width, img_height):
    img = scipy.misc.imread(img_addr)
    img = crop_center_image(img)
    img = scipy.misc.imresize(img, [img_width,img_height])
    return img

def register_extension(id, extension):
    Image.EXTENSION[extension.lower()] = id.upper()
```

```
def register_extensions(id, extensions):
    for extension in extensions: register_extension(id, extension)

### create grid_img
### the image inputs will be 4 dimensions, which 0 dimention is the number of example
def build_grid_img(inputs, img_height, img_width, n_row, n_col):
    grid_img = np.zeros((img_height*n_row, img_width*n_col, 3))
    print(inputs.shape)
    count = 0
    for i in range(n_col):
        for j in range(n_row):
            grid_img[i*img_height:(i+1)*img_height, j*img_width:(j+1)*img_width,:] = inp
            count += 1
    return grid_img

### save images as a grid
def save_grid_img(inputs, path, img_height, img_width, n_row, n_col):

    Image.register_extension = register_extension
    Image.register_extensions = register_extensions
    grid_img = build_grid_img(inputs, img_height, img_width, n_row, n_col)
    scipy.misc.imsave(path, grid_img)

def _int64_feature(value):
    return tf.train.Feature(int64_list=tf.train.Int64List(value=[value]))

def _bytes_feature(value):
    return tf.train.Feature(bytes_list=tf.train.BytesList(value=[value]))

### convert image into binary format
def get_image_binary(img):
    shape = np.array(img.shape, np.int32)
    img = np.asarray(img,np.uint8)
    return img.tobytes(), shape.tobytes()

### write data into tf record file format (images are stored in zip file)
def write_tfrecord(tfrecord_filename, zipFileName, img_height, img_width):

    ### images counter
    count = 0

    ### create a writer
    writer = tf.python_io.TFRecordWriter(tfrecord_filename)

    with ZipFile(zipFileName) as archive:

        for entry in archive.infolist():

            # skip the folder content
            if entry.filename == 'content/':
                continue

            with archive.open(entry) as file:

                sys.stdout.write('\r'+str(count))
```

```
                ### pre-process data
                img = np.asarray(Image.open(file))
                img = crop_center_image(img)
                img = scipy.misc.imresize(img, [img_height,img_width])
                img, shape = get_image_binary(img)

                ### create features
                feature = {'image': _bytes_feature(img),
                           'shape': _bytes_feature(shape)}
                features = tf.train.Features(feature=feature)

                ### create example
                example = tf.train.Example(features=features)

                ### write example
                writer.write(example.SerializeToString())
                sys.stdout.flush()

                count += 1

        writer.close()

### parse serialized data back into the usable form
def _parse(serialized_data):
    features = {'image': tf.FixedLenFeature([], tf.string),
                'shape': tf.FixedLenFeature([], tf.string)}
    features = tf.parse_single_example(serialized_data,
                                       features)
    img = tf.cast(tf.decode_raw(features['image'],tf.uint8), tf.float32)
    shape = tf.decode_raw(features['shape'],tf.int32)
    img = tf.reshape(img, shape)

    return img

### read tf record
def read_tfrecord(tfrecord_filename):

    ### create dataset
    dataset = tf.data.TFRecordDataset(tfrecord_filename)
    dataset = dataset.map(_parse)
    return dataset

def download(url, file_path):
    if os.path.exists(file_path):
        print("the file is already existed")
        return
    else:
        print("downloading file...")
    urllib.request.urlretrieve(url, file_path)
    print("downloading done")

### restore checkpoint from "Checkpoint" folder
def _restore_checkpoint(saver, sess):

    ckpt_path = os.path.dirname(os.path.join(os.getcwd(),'checkpoint/'))
    ckpt = tf.train.get_checkpoint_state(ckpt_path)
```

```
    if ckpt and ckpt.model_checkpoint_path:
        saver.restore(sess, ckpt.model_checkpoint_path)
        print("get checkpoint")
    return ckpt_path

### create dataset and return iterator and dataset
def _get_data(training_data_tfrecord, batch_size):

    dataset = util.read_tfrecord(training_data_tfrecord)
    dataset = dataset.batch(batch_size)
    iterator = dataset.make_initializable_iterator()
    return iterator, dataset

def train_dfc_vae():

    ### setup hyper-parameter
    batch_size = BATCH_SIZE
    epoch = NUM_EPOCH
    vgg_layers = VGG_LAYERS
    alpha = ALPHA
    beta = BETA
    learning_rate = LEARNING_RATE
    img_height = IMG_HEIGHT
    img_width = IMG_WIDTH
    training_data_tfrecord = TRAINING_DATA

    ### get training data
    iterator, _ = _get_data(training_data_tfrecord, batch_size)

    ### create iterator's initializer for training data
    iterator_init = iterator.initializer
    data = iterator.get_next()

    ### define input data
    img_input = (tf.reshape(data, shape=[-1, img_height, img_width, 3])-IMG_MEAN) / IMG_S

    ### build model graph
    model = dfc.dfc_vae_model([img_height,img_width], img_input, alpha, beta, vgg_layers
    model.build_model(tf.AUTO_REUSE)

    ### create saver for restoring and saving variables
    saver = tf.train.Saver()

    with tf.Session() as sess:

        ### initialize global variable
        sess.run(tf.global_variables_initializer())

        ### restore checkpoint
        ckpt_path = _restore_checkpoint(saver,sess)

        ### load pre-trained vgg weights
        model.load_vgg_weight('vgg16_weights.npz', sess)

        ### lists of losses, used for tracking
        kl_loss = []
```

```
pct_loss = []
total_loss = []
iteration = []

### count how many training iteration (different from epoch)
iteration_count = 0

for i in range(epoch):

    ### initialize iterator
    sess.run(iterator_init)

    try:
        while True:
            sys.stdout.write('\r' + 'Iteration: ' + str(iteration_count))
            sys.stdout.flush()

            ### every 100 iteration, print losses
            if iteration_count % 100 == 0:
                pct, kl, loss, tempmean, tempstd = sess.run([model.pct_loss, mod
                pct = pct * beta
                print("\nperceptual loss: {}, kl loss: {}, total loss: {}".forma
                print(tempmean[0,0:5])
                print(tempstd[0,0:5])

                iteration.append(iteration_count)
                kl_loss.append(kl)
                pct_loss.append(pct)
                total_loss.append(loss)

            ### every 500 iteration, save images
            if iteration_count % 500 == 0:

                ### get images from the dfc_vae_model
                original_img, gen_img, ran_img = sess.run([model.img_input,model

                ### denomalize
                original_img = original_img*IMG_STDDEV + IMG_MEAN
                gen_img = gen_img*IMG_STDDEV + IMG_MEAN
                ran_img = ran_img*IMG_STDDEV + IMG_MEAN

                ### clip values to be in RGB range and transform to 0..1 float
                original_img = np.clip(original_img,0.,255.).astype('float32')/2
                gen_img = np.clip(gen_img,0.,255.).astype('float32')/255.
                ran_img = np.clip(ran_img,0.,255.).astype('float32')/255.

                ### save images in 5x5 grid
                util.save_grid_img(original_img, os.path.join(os.getcwd(), 'outp
                util.save_grid_img(gen_img, os.path.join(os.getcwd(), 'outputs',
                util.save_grid_img(ran_img, os.path.join(os.getcwd(), 'outputs',

                ### plot losses
                plt.figure()
                plt.plot(iteration, kl_loss)
                plt.plot(iteration, pct_loss)
                plt.plot(iteration, total_loss)
```

```
                    plt.legend(['kl loss', 'perceptual loss', 'total loss'], bbox_to
                    plt.title('Loss per iteration')
                    plt.show()

                ### run optimizer
                sess.run(model.optimizer)
                iteration_count += 1

            except tf.errors.OutOfRangeError:
                pass

            ### save session for each epoch
            ### recommend to change to save encoder, decoder, VGG's variables separately
            print("\nepoch: {}, loss: {}".format(i, loss))
            saver.save(sess, os.path.join(ckpt_path,"Face_Vae"), global_step = iteration_
            print("checkpoint saved")


def test_gen_img(model, sess, i):

    real_img, gen_img, ran_img = sess.run([model.img_input, model.gen_img, model.ran_img

    ran_img = ran_img*IMG_STDDEV + IMG_MEAN
    real_img = real_img*IMG_STDDEV + IMG_MEAN
    gen_img = gen_img*IMG_STDDEV + IMG_MEAN

    ran_img = ran_img/255.
    real_img = real_img/255.
    gen_img = gen_img/255.

    util.save_grid_img(ran_img, os.path.join(os.getcwd(), 'outputs', 'test' , 'random-' +
    util.save_grid_img(real_img, os.path.join(os.getcwd(), 'outputs', 'test' , 'real-' +
    util.save_grid_img(gen_img, os.path.join(os.getcwd(), 'outputs', 'test' , 'gen-' + s

def test_interpolation(model, sess, i):

    z1 = sess.run(model.z)
    z2 = sess.run(model.z)
    print(z1.shape)
    print(z2.shape)
    print(z1[0,:5])
    print(z2[0,:5])

    z = tf.Variable(np.zeros(z1.shape).astype(np.float32))
    gen_img = model.decoder(z, tf.AUTO_REUSE)

    interpolated_img_list = []

    for j in range(31):
        interpolated_z = z1 * (30-j)/30. + z2 * j/30.
        sess.run(z.assign(interpolated_z))
        interpolated_img = sess.run(gen_img)
        interpolated_img = interpolated_img*IMG_STDDEV + IMG_MEAN
        interpolated_img = interpolated_img/255.
        interpolated_img = util.build_grid_img(interpolated_img, interpolated_img.shape[
        interpolated_img_list.append(interpolated_img)
```

```
    for j in range(31):
        imageio.mimsave(os.path.join(os.getcwd(), 'outputs', 'test_interpolated' , 'inter

    return interpolated_img_list


def test():

    ### setup hyper-parameter
    num_test_set = 2
    batch_size = 64
    vgg_layers = VGG_LAYERS
    img_height = IMG_HEIGHT
    img_width = IMG_WIDTH
    training_data_tfrecord = TRAINING_DATA
    ### get training data
    iterator, _ = _get_data(training_data_tfrecord, batch_size)
        ### create iterator's initializer for training data
    iterator_init = iterator.initializer
    data = iterator.get_next()
    ### define input data
    img_input = (tf.reshape(data, shape=[-1, img_height, img_width, 3]) - IMG_MEAN)/IMG_S
    ### build model graph
    model = dfc.dfc_vae_model([img_height,img_width], img_input)
    model.build_model(tf.AUTO_REUSE)
    ### create saver for restoring and saving variables
    saver = tf.train.Saver()

    with tf.Session() as sess:

        ### initialize global variable
        sess.run(tf.global_variables_initializer())

        ### restore checkpoint
        ckpt_path = _restore_checkpoint(saver,sess)

        sess.run(iterator_init)

        for i in range(num_test_set):

            test_gen_img(model, sess, i)
            x = test_interpolation(model, sess, i)

if __name__ == '__main__':
    tf.reset_default_graph()
    train_dfc_vae()
    test()
```

ПРИЛОЖЕНИЕ В

Код программы. Гауссовский процесс и байесовская оптимизаия

```
import tensorflow as tf
import numpy as np
import os
import GPy
import GPyOpt

import tf_vae.dfc_vae_model as dfc

import matplotlib.pyplot as plt

def _restore_checkpoint(saver, sess):
    ckpt_path = os.path.dirname(os.path.join(os.getcwd(),'/home/dl/dl/thesis/checkpoint_8
    ckpt = tf.train.get_checkpoint_state(ckpt_path)
    if ckpt and ckpt.model_checkpoint_path:
        saver.restore(sess, ckpt.model_checkpoint_path)
        print("get checkpoint")
    return ckpt_path

### define input data
IMG_MEAN = np.array([134.10714722, 102.52040863, 87.15436554])
IMG_STDDEV = np.sqrt(np.array([3941.30175781, 2856.94287109, 2519.35791016]))
img_input = tf.placeholder(dtype=tf.float32, shape=(1,64,64,3))

### build model graph
model = dfc.dfc_vae_model([64, 64], img_input)
model.build_model(tf.AUTO_REUSE)

### create saver for restoring and saving variables
saver = tf.train.Saver()
sess = tf.Session()
sess.run(tf.global_variables_initializer())

### restore checkpoint
ckpt_path = _restore_checkpoint(saver,sess)

class FacialComposit:
    def __init__(self, model, latent_size):
        self.latent_size = latent_size
        self.latent_placeholder = tf.placeholder(tf.float32, (1, latent_size))
        self.decode = model.decoder(self.latent_placeholder, tf.AUTO_REUSE)
        self.samples = None
        self.images = None
        self.rating = None

    def _get_image(self, latent):
        img = sess.run(self.decode,
```

```
                    feed_dict={self.latent_placeholder: latent[None, :]})
        img = img*IMG_STDDEV + IMG_MEAN
        img = img/255.
        return img

    @staticmethod
    def _show_images(images, titles):
        assert len(images) == len(titles)
        clear_output()
        plt.figure(figsize=(3*len(images), 3))
        n = len(titles)
        for i in range(n):
            plt.subplot(1, n, i+1)
            plt.imshow(images[i])
            plt.title(str(titles[i]))
            plt.axis('off')
        plt.show()

    @staticmethod
    def _draw_border(image, w=2):
        bordred_image = image.copy()
        bordred_image[:, :w] = [1, 0, 0]
        bordred_image[:, -w:] = [1, 0, 0]
        bordred_image[:w, :] = [1, 0, 0]
        bordred_image[-w:, :] = [1, 0, 0]
        return bordred_image

    def query_initial(self, n_start=5, select_top=None):
        '''
        Creates initial points for Bayesian optimization
        Generate *n_start* random images and asks user to rank them.
        Gives maximum score to the best image and minimum to the worst.
        :param n_start: number of images to rank initialy.
        :param select_top: number of images to keep
        '''
        self.samples = np.zeros((n_start, self.latent_size))

        self.images = np.zeros((n_start, 64, 64, 3))
        self.rating = np.zeros((n_start,))

        ### Show user some samples (hint: use self._get_image and input())
        prior_distr = tf.random_normal((1, 8), mean = 0.0, stddev=1.0)
        for i in range(n_start):
          self.samples[i] = sess.run(prior_distr)
          self.images[i] = self._get_image(self.samples[i])
        self._show_images(self.images, ['num_'+str(i) for i in range(1, n_start+1)])
        rate = input()
        self.rating = np.array([ 100*(int(i) / n_start) for i in rate.split(',')])
        # Check that tensor sizes are correct
        np.testing.assert_equal(self.rating.shape, [n_start])
        np.testing.assert_equal(self.images.shape, [n_start, 64, 64, 3])
        np.testing.assert_equal(self.samples.shape, [n_start, self.latent_size])

    def evaluate(self, candidate):
        '''
        Queries candidate vs known image set.
```

```
        Adds candidate into images pool.
        :param candidate: latent vector of size 1xlatent_size
        '''
        initial_size = len(self.images)
        ## Show user an image and ask to assign score to it.
        ## You may want to show some images to user along with their scores
        ## You should also save candidate, corresponding image and rating
        self._show_images(self.images, self.rating)
        image = self._get_image(candidate[0])
        self.images = np.append(self.images, image, axis=0)
        self._show_images(image, ['mark?'])
        rate = 100*(int(input())/ initial_size)
        self.rating = np.append(self.rating, [rate], axis=0)
        self.samples = np.append(self.samples, candidate, axis=0)

        candidate_rating = rate
        assert len(self.images) == initial_size + 1
        assert len(self.rating) == initial_size + 1
        assert len(self.samples) == initial_size + 1
        return candidate_rating

    def optimize(self, n_iter=10, w=4, acquisition_type='MPI', acquisition_par=0.3):
        if self.samples is None:
            self.query_initial(n_start=5)

        bounds = [{'name': 'z_{0:03d}'.format(i),
                   'type': 'continuous',
                   'domain': (-w, w)}
                  for i in range(self.latent_size)]
        optimizer = GPyOpt.methods.BayesianOptimization(f=self.evaluate, domain=bounds,
                                                        acquisition_type = acquisition_ty
                                                        acquisition_par = acquisition_par
                                                        exact_eval=False, # Since we are
                                                        model_type='GP',
                                                        X=self.samples,
                                                        Y=self.rating[:, None],
                                                        maximize=True)
        optimizer.run_optimization(max_iter=n_iter, eps=-1)

    def get_best(self):
        index_best = np.argmax(self.rating)
        return self.images[index_best]

    def draw_best(self, title=''):
        index_best = np.argmax(self.rating)
        image = self.images[index_best]
        plt.imshow(image)
        plt.title(title)
        plt.axis('off')
        plt.show()
composit = FacialComposit(model, 8)
composit.optimize(n_iter=5)
composit.draw_best('Darkest hair ')
composit = FacialComposit(model, 8)
composit.optimize(n_iter=10)
composit.draw_best('Widest smile')
```