

# 18 Автокодировщики

## 18 Автокодировщики

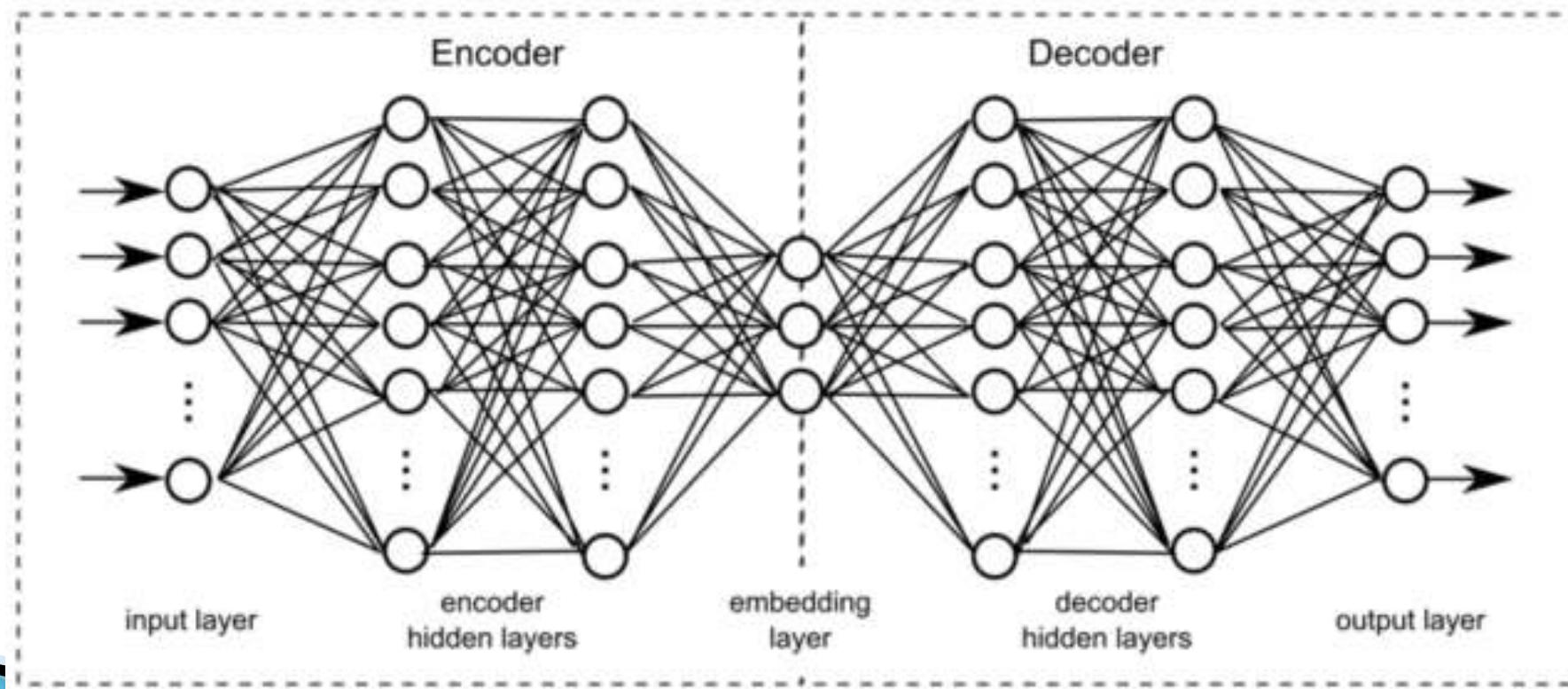
- Модели нейронных сетей глубокого обучения могут также применяться в случаях обучения без учителя.
- К таким нейронным сетям относится такая модель глубокого обучения как **автокодировщик – autoencoder**. Впервые модель автокодировщика была представлена еще в 1986 году в классической работе Дэвида Румельхарта, Джеки Хинтона и Рональда Уильямса, «*Learning internal representations by error propagation*».
- Цель модели автокодировщика – получить представление исходного множества данных  $X$  меньшей размерности или же вложение исходного пространства данных в пространство информативных признаков меньшей размерности на этапе кодирования с возможностью восстановления исходных данных на этапе декодирования.
- Предположим, что у нас есть некий набор данных, который мы хотели бы описать с помощью пространства новых «умных» или более информативных признаков меньшей размерности, которые были бы лучше и «интереснее». Например, мы бы хотели описать рукописные цифры не пиксел за пикселом, а с помощью каких-то признаков, из которых было бы достаточно просто выяснить, какая цифра написана, или чьим почерком, или каким цветом. Это выглядит как задача обучения без учителя, которую трудно решать нейронными сетями: непонятно, как определить функцию ошибки для задачи «найти интересные признаки».

## 18 Автокодировщики

- Автокодировщики - это искусственные нейронные сети, которые способны создавать эффективные представления входных данных, называемые кодировками (coding), без учителя (т.е. обучающий набор является непомеченным).
- Основная идея автокодировщиков : давайте превратим задачу обучения без учителя в задачу обучения с учителем, сами себе придумаем тестовые примеры с известными правильными ответами.
- Цель – обучиться выдавать на выходе ровно тот же образец, который подавали ей на вход.
- Кодировки обычно имеют гораздо меньшую размерность, чем входные данные, делая автокодировщики удобным средством понижения размерности.
- Автокодировщики действуют как мощные обнаружители признаков и могут использоваться для предварительного обучения без учителя глубоких нейронных сетей.
- Теоретической основой данного класса сетей является *анализ главных компонент* (principal component analysis). Автоэнкодерные нейронные сети называются также рециркуляционными, автоассоциативными, NPCA (nonlinear principal component analysis) нейронными сетями.
- Автокодировщики способны случайным образом генерировать новые данные, которые выглядят очень похожими на обучающие данные; это называется порождающей моделью (generative model). Например, вы могли бы обучить автокодировщик на фотографиях лиц, и он был бы в состоянии генерировать новые лица.

## 18 Автокодировщики

- Автокодировщик состоит из двух частей функции кодировщика (encoder)  $f(\cdot)$  и функции декодировщика (decoder)  $g(\cdot)$ , где размерность функции  $f$  кодировщика, как правило, меньше размерности входных данных (см. рисунок ниже). Восстановление входного вектора  $X$  задается путем последовательного объединения или суперпозиции функции декодировщика на функцию кодировщика:  $r(X)=g(f(X))$ . При этом автокодировщик обучается минимизировать ошибки восстановления входных векторов  $J(X,r(X))$ , т.е. аппроксимировать тождественную функцию  $r(X)=g(f(X))=X$ .



## 18 Автокодировщики

- Возникает противоположный вопрос: почему это вообще может работать? Отображение пространства большой размерности  $R^D$  в пространство меньшей размерности  $R^d$ ,  $d < D$ , не может быть взаимно однозначным, то есть мы всегда будем терять какую-то информацию; как же мы тогда будем восстанавливать исходные векторы  $x \in R^D$ ?
- Если бы нам нужно было научить сеть отображать все возможные вектора, например, для картинок размера  $28 \times 28$  пикселов выполнить отображение  $R^{784} \rightarrow R^{100}$ , это была бы невозможная задача: случайный вектор из пространства  $R^{784}$  содержит гораздо больше информации, чем вектор из  $R^{100}$ .
- Но хотя базовое пространство, в котором представлены данные, и имеет большую размерность, сами данные в нем лежат неподалеку от *многообразия* гораздо меньшей размерности. Нам вовсе не нужно кодировать любой белый шум из случайно зажженных 784 пикселов; нам нужно кодировать некие «осмысленные» изображения, которые обладают множеством очевидных для человека свойств: они выражают одну из десяти цифр, имеют определенную толщину линии, наклон, представляют собой обычно неразрывные, связные объекты или состоят из немногих штрихов. Все это очень сильно сужает множество возможных изображений цифр и позволяет надеяться на то, что существует представление рукописных цифр в пространстве меньшей размерности.

## 18.1 Автокодировщики. Обучение

- В основе обучения автокодировщиков – настройки весов, лежит метод обратного распространения ошибки или градиентного спуска.
- **Правило обучения Ойя**
- Правило обучения было предложено в 1982 г. финским ученым Е. Ойя (Erkki Oja) [Oja E. A simplified neuron model as a principal component analyzer // J. of mathematical biology. – 1982. – № 15. – Р. 267–273]. Данное правило обучения можно получить на основе классического метода градиентного спуска.
- Автоэнкодерные нейронные сети должны обеспечивать такое преобразование информации, чтобы достигалась минимальная суммарная квадратичная ошибка между входными и реконструированными образами:

$$E_S = \frac{1}{2} \sum_{k=1}^L \sum_{i=1}^n (\bar{x}_i^k - x_i^k)^2, \quad (18.1)$$

где  $\bar{x}_i^k$  –  $i$ -я компонента  $k$ -го выходного образа сети;  $x_i^k$  –  $i$ -я компонента  $k$ -го входного образа сети;  $L$  – общее количество образов.

- Для минимизации суммарной квадратичной ошибки сети будем использовать метод градиентного спуска для последовательного обучения. Тогда

$$w'_{ji}(t+1) = w'_{ji}(t) - \alpha \frac{\partial E}{\partial w'_{ji}(t)}.$$

## 18.1 Автокодировщики. Обучение

- Здесь  $E$  – квадратичная ошибка сети для одного образа, которая вычисляется как

$$E = \frac{1}{2} \sum_{i=1}^n (\bar{x}_i - x_i)^2.$$

- В случае использования линейной функции активации нейронных элементов выходное значение  $j$ -го нейрона скрытого слоя определяется следующим образом:

$$y_j = S_j = \sum_{i=1}^n w_{ij} x_i.$$

- Аналогично для  $i$ -го нейрона выходного слоя:

$$\bar{x}_i = S_i = \sum_{j=1}^p w'_{ji} y_j.$$

- Тогда

$$\frac{\partial E}{\partial w'_{ji}} = \frac{\partial E}{\partial \bar{x}_i} \frac{\partial \bar{x}_i}{\partial w'_{ji}} = (\bar{x}_i - x_i) y_j = -(x_i - \bar{x}_i) y_j = -y_j \left( x_i - \sum_{j=1}^p w'_{ji} y_j \right).$$

- В результате правило модификации весовых коэффициентов восстанавливающего слоя можно представить как

$$w'_{ji}(t+1) = w'_{ji} + \alpha y_j \left( x_i - \sum_{j=1}^p w'_{ji} y_j \right). \quad (18.2)$$

## 18.1 Автокодировщики. Обучение

- Поскольку в соответствии с методом главных компонент матрица весовых коэффициентов сжимающего и восстанавливающего слоя автоэнкодерной сети связана операцией транспонирования

$$W = (W')^T,$$

- то весовые коэффициенты нейронных элементов сжимающего слоя будут изменяться в соответствии со следующим выражением:

$$w_{ij}(t+1) = w_{ij}(t) + \alpha y_j \left( x_i - \sum_{j=1}^p w_{ij} y_j \right). \quad (18.3)$$

- Приведенные выше выражения (18.2) и (18.3) называются правилом обучения Ойя. Последнее слагаемое в выражении (18.3) препятствует неограниченному возрастанию весовых коэффициентов.
- Теоретически доказано, что Ойя-правило эквивалентно методу главных компонент. Нейроны сжимающего слоя соответствуют главным компонентам. Нейронный элемент сжимающего слоя, который имеет максимальное значение дисперсии, определяет первую главную компоненту, нейронный элемент со следующим значением дисперсии определяет вторую компоненту и т.д.

## 18.1 Автокодировщики. Обучение

- Рассмотрим правило обучения Ойя для нелинейной сети. В этом случае выходные значения сжимающего и восстановливающего слоя определяются следующим образом:

$$y_j = F(S_j) = F\left(\sum_{i=1}^n w_{ij}x_i\right),$$

$$\bar{x}_i = F(S_i) = F\left(\sum_{j=1}^p w'_{ji}y_j\right).$$

- Тогда правило обучения Ойя для нелинейной сети имеет вид

$$w_{ij}(t+1) = w_{ij}(t) + \alpha y_j \left( x_i - F\left(\sum_{j=1}^p w_{ij}y_j\right) \right). \quad (18.4)$$

- Как следует из последнего выражения, правило обучения Ойя для нелинейной сети записывается в том же виде, что и для линейной сети, где в качестве выходных значений сжимающего и восстановливающего слоя используется нелинейная функция активации.
- Нейроны сжимающего слоя в этом случае также соответствуют главным компонентам.

Другие правила и методы обучения можно посмотреть в  
В.А.Головко, В.В.Краснопрошин. Нейросетевые технологии обработки данных : учеб.  
Пособие. – Минск : БГУ, 2017. – 263 с., раздел 6.3.

## 18.2 Автокодировщики. Модификации

- Автокодировщики за более, чем двадцать лет своего существования получили сразу несколько серьезных «апгрейдов».

- **Распространенные варианты автокодировщиков**

Две важные разновидности автокодировщиков: *сжимающие* и *шумоподавляющие*.

**Сжимающие автокодировщики.** Эта архитектура изображена на рис. выше.

Входные данные должны пройти через «бутылочное горлышко» слоя вложения и только потом раздвигаются в выходное представление.

**Шумоподавляющие автокодировщики.** В этом случае автокодировщик получает искаженные входные данные (например, некоторые признаки случайным образом удалены), а сеть обучается выдавать на выходе неискаженный сигнал. Например, давайте выберем 10 % пикселов изображения и заменим их нулями (черными пикселями) или случайными значениями интенсивностей, но восстановить при этом попросим не искаженный вариант картинки, а исходный, в котором все пиксели стоят на своих местах. Таким образом, автокодировщик должен будет не просто сжать полученный пример, но еще и частично восстановить утраченные в процессе зашумления данные.



## 18.3 Автокодировщики. Выполнение анализа главных компонентов с помощью сжимающего линейного автокодировщика

- Если автокодировщик использует только линейные функции активации и функцию издержек в виде среднеквадратической ошибки (MSE), то можно показать, что он в итоге выполняет анализ главных компонентов (PCA).
- Для выполнения простого анализа PCA мы не применяем какую-то функцию активации (т.е. все нейроны линейны), а функция потерь (ошибка) основана на MSE.
- Следующий код строит простой линейный автокодировщик для выполнения анализа PCA на трехмерном наборе данных, проецируя его в два измерения:

Build 3D dataset:

```
In [4]: import numpy.random as rnd

rnd.seed(4)
m = 200
w1, w2 = 0.1, 0.3
noise = 0.1

angles = rnd.rand(m) * 3 * np.pi / 2 - 0.5
data = np.empty((m, 3))
data[:, 0] = np.cos(angles) + np.sin(angles)/2 + noise * rnd.randn(m) / 2
data[:, 1] = np.sin(angles) * 0.7 + noise * rnd.randn(m) / 2
data[:, 2] = data[:, 0] * w1 + data[:, 1] * w2 + noise * rnd.randn(m)
```

Normalize the data:

```
In [5]: from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train = scaler.fit_transform(data[:100])
X_test = scaler.transform(data[100:])
```

## 18.3 Автокодировщики. Выполнение анализа главных компонентов с помощью сжимающего линейного автокодировщика

```
In [6]: import tensorflow as tf

reset_graph()

n_inputs = 3
n_hidden = 2 # codings
n_outputs = n_inputs

learning_rate = 0.01

X = tf.placeholder(tf.float32, shape=[None, n_inputs])
hidden = tf.layers.dense(X, n_hidden)
outputs = tf.layers.dense(hidden, n_outputs)

reconstruction_loss = tf.reduce_mean(tf.square(outputs - X))

optimizer = tf.train.AdamOptimizer(learning_rate)
training_op = optimizer.minimize(reconstruction_loss)

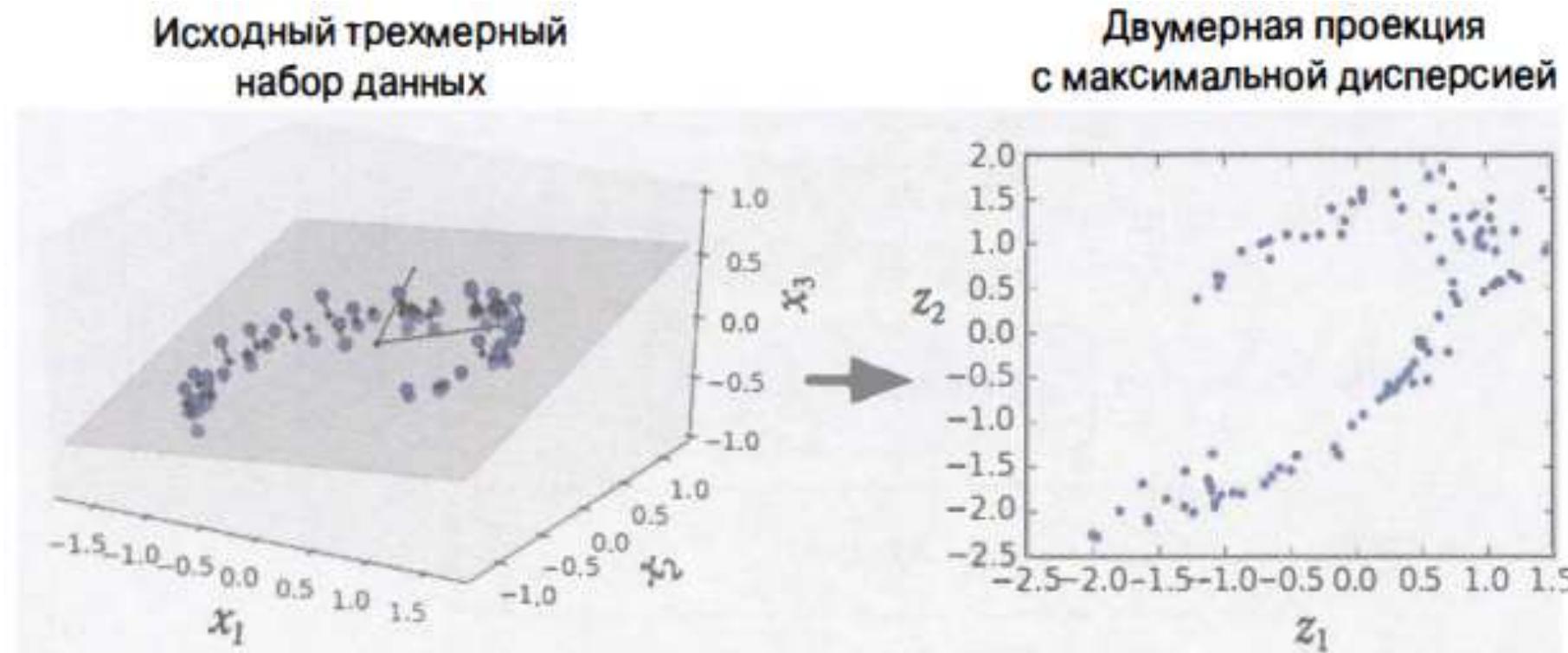
init = tf.global_variables_initializer()
```

```
In [7]: n_iterations = 1000
codings = hidden

with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
        training_op.run(feed_dict={X: X_train})
    codings_val = codings.eval(feed_dict={X: X_test})
```

## 18.3 Автокодировщики. Выполнение анализа главных компонентов с помощью сжимающего линейного автокодировщика

- Слева на рисунке показан исходный трехмерный набор данных, а справа - выход скрытого слоя автокодировщика (т.е. кодирующего слоя). Как видите, автокодировщик нашел наилучшую двумерную плоскость для проецирования на нее данных, сохраняя как можно больше дисперсии (подобно РСА).



## 18.4 Автокодировщики. Пример применения линейного сжимающего автокодировщика для данных генной экспрессии

```
8 import tensorflow as tf
9 import pandas as pd
10
11 data = pd.read_csv("F:\\Python_projects\\DiffusionNets\\Data\\expr_set_nameoff.csv")
12 data_= data.dropna()
13 data_arr = data_.values
14 data_arrrt = data_arr.T
15 n_inputs = 14376 # входы
16 n_hidden = 100 # кодировка - число нейронов скрытым слоем
17 n_outputs = n_inputs
18 learning_rate = 0.01
19 n_iterations = 300
20 display_step = 1
21 X = tf.placeholder(tf.float32, shape=[None,n_inputs])
22 # Полносвязная сеть кодировщика
23 hidden = tf.layers.dense(X, n_hidden)
24 # Полносвязная сеть декодировщика
25 outputs = tf.layers.dense(hidden, n_outputs )
26 # Определение ошибки
27 reconstruction_loss = tf.reduce_mean(tf.square(outputs - X)) # MSE
28 # Определение оптимизатора
29 optimizer = tf.train.AdamOptimizer(learning_rate)
30 training_op = optimizer.minimize(reconstruction_loss)
31 init = tf.global_variables_initializer()
32 # Обучающий набор данных
33 x_train = data_arrrt[0:201,:]
34 # Тестовый набор данных
35 x_test = data_arrrt[201:295,:]
36 codings = hidden # выход скрытого слоя предоставляет кодировки
```

## 18.4 Автокодировщики. Пример применения линейного сжимающего автокодировщика для данных генной экспрессии

```
38 # Блок обучения и тестирования
39 with tf.Session() as sess:
40     init.run()
41 # Обучение автокодировщика
42     for iteration in range (n_iterations):
43 # метки отсутствуют (без учителя)
44         training_op.run(feed_dict={X:x_train})
45         if iteration % display_step == 0:
46 # определение ошибки
47             loss_train = reconstruction_loss.eval(feed_dict={X: x_train})
48             print("Iteration:", '%04d' % (iteration+1), "Train MSE:", loss_train)
49
50     print("Autoencoder training completed !")
51 # Тестирование автокодировщика
52     codings_val = codings.eval(feed_dict= {X:x_test})
53 # определение ошибки
54     loss_test = reconstruction_loss.eval(feed_dict={X: x_test})
55     print("Test MSE:", loss_test)
56
```

## 18.4 Автокодировщики. Пример применения линейного сжимающего автокодировщика для данных генной экспрессии

```
Iteration: 0001 Train MSE: 0.9656317
Iteration: 0002 Train MSE: 0.21108313
Iteration: 0003 Train MSE: 1.1029868
Iteration: 0004 Train MSE: 4.7528386
Iteration: 0005 Train MSE: 0.6432017
Iteration: 0006 Train MSE: 1.8234264
Iteration: 0007 Train MSE: 0.42494324
Iteration: 0008 Train MSE: 1.344124
Iteration: 0009 Train MSE: 0.7248412
Iteration: 0010 Train MSE: 0.36573994
Iteration: 0011 Train MSE: 0.40927544
Iteration: 0012 Train MSE: 0.33240685
Iteration: 0013 Train MSE: 0.30559298
Iteration: 0014 Train MSE: 0.31131405
Iteration: 0015 Train MSE: 0.3131206
Iteration: 0016 Train MSE: 0.3075008
Iteration: 0017 Train MSE: 0.24985352
Iteration: 0018 Train MSE: 0.16004896
Iteration: 0019 Train MSE: 0.11472199
Iteration: 0020 Train MSE: 0.13481523
Iteration: 0021 Train MSE: 0.14602934
Iteration: 0022 Train MSE: 0.13636829
Iteration: 0023 Train MSE: 0.11391294
Iteration: 0024 Train MSE: 0.07752005
Iteration: 0025 Train MSE: 0.076608144
Iteration: 0026 Train MSE: 0.07690825
Iteration: 0027 Train MSE: 0.077692345
Iteration: 0028 Train MSE: 0.07002507
Iteration: 0029 Train MSE: 0.053117383
Iteration: 0030 Train MSE: 0.051354095
Iteration: 0031 Train MSE: 0.046688933
Iteration: 0032 Train MSE: 0.051081646
Iteration: 0033 Train MSE: 0.046927348
Iteration: 0034 Train MSE: 0.04917706
Iteration: 0035 Train MSE: 0.04048209
Iteration: 0268 Train MSE: 0.008050769
Iteration: 0269 Train MSE: 0.008030306
Iteration: 0270 Train MSE: 0.008009267
Iteration: 0271 Train MSE: 0.007988421
Iteration: 0272 Train MSE: 0.007967969
Iteration: 0273 Train MSE: 0.007947549
Iteration: 0274 Train MSE: 0.007927315
Iteration: 0275 Train MSE: 0.007907091
Iteration: 0276 Train MSE: 0.007887168
Iteration: 0277 Train MSE: 0.007867044
Iteration: 0278 Train MSE: 0.00784722
Iteration: 0279 Train MSE: 0.007827664
Iteration: 0280 Train MSE: 0.0078080543
Iteration: 0281 Train MSE: 0.0077885888
Iteration: 0282 Train MSE: 0.0077691814
Iteration: 0283 Train MSE: 0.00774997
Iteration: 0284 Train MSE: 0.0077307853
Iteration: 0285 Train MSE: 0.0077118175
Iteration: 0286 Train MSE: 0.0076929512
Iteration: 0287 Train MSE: 0.0076740524
Iteration: 0288 Train MSE: 0.007655339
Iteration: 0289 Train MSE: 0.007636739
Iteration: 0290 Train MSE: 0.00761826
Iteration: 0291 Train MSE: 0.007599851
Iteration: 0292 Train MSE: 0.00758155
Iteration: 0293 Train MSE: 0.007563336
Iteration: 0294 Train MSE: 0.007545179
Iteration: 0295 Train MSE: 0.0075271972
Iteration: 0296 Train MSE: 0.0075092935
Iteration: 0297 Train MSE: 0.007491453
Iteration: 0298 Train MSE: 0.007473711
Iteration: 0299 Train MSE: 0.0074560577
Iteration: 0300 Train MSE: 0.0074385074
Autoencoder training completed !
Test MSE: 0.019969454
```

## 18.5 Автокодировщики. Многослойные автокодировщики. Реализация с помощью TensorFlow

- Автокодировщики могут иметь множество скрытых слоев. В таком случае они называются многослойными автокодировщиками (stacked autoencoder) или глубокими автокодировщиками (deep autoencoder). Добавление дополнительных слоев помогает автокодировщику строить более сложные кодировки. Однако нужно соблюдать осторожность, чтобы не сделать автокодировщик чересчур мощным. Представьте себе кодировщик, который является настолько мощным, что он просто научится сопоставлять каждый вход с единственным произвольным числом (а декодировщик научится обратному сопоставлению). Очевидно, такой автокодировщик будет идеально реконструировать обучающие данные, но в процессе не узнает ни одного полезного представления данных (и вряд ли хорошо обобщится на новые образцы в результате переобучения).
- Архитектура многослойного автокодировщика обычно симметрична относительно центрального скрытого слоя (кодирующего слоя). Попросту говоря, она похожа на бутерброд. Например, автокодировщик для датасета MNIST может иметь 784 входа, за которыми следует скрытый слой из 300 нейронов, центральный скрытый слой из 150 нейронов, еще один скрытый слой из 300 нейронов и выходной слой из 784 нейронов. Такой многослойный автокодировщик показан на рисунке ниже.

## 18.5 Автокодировщики. Многослойные автокодировщики. Реализация с помощью TensorFlow



## 18.5 Автокодировщики. Многослойные автокодировщики. Реализация с помощью TensorFlow

- Многослойный автокодировщик можно реализовать способом, очень похожим на реализацию обычновенного глубокого многослойного персептрана. Приведенный ниже код строит многослойный автокодировщик для MNIST с применением инициализации Хе, функции активации ELU и регуляризации  $\ell_2$ .

```
from functools import partial

n_inputs = 28 * 28
n_hidden1 = 300
n_hidden2 = 150 # codings
n_hidden3 = n_hidden1
n_outputs = n_inputs

learning_rate = 0.01
l2_reg = 0.0001

X = tf.placeholder(tf.float32, shape=[None, n_inputs])

he_init = tf.contrib.layers.variance_scaling_initializer() # He initialization
#Equivalent to:
#he_init = Lambda shape, dtype=tf.float32: tf.truncated_normal(shape, 0., stddev=np.sqrt(2/shape[0]))
l2_regularizer = tf.contrib.layers.l2_regularizer(l2_reg)
my_dense_layer = partial(tf.layers.dense,
                        activation=tf.nn.elu,
                        kernel_initializer=he_init,
                        kernel_regularizer=l2_regularizer)
```

## 18.5 Автокодировщики. Многослойные автокодировщики. Реализация с помощью TensorFlow

```
hidden1 = my_dense_layer(x, n_hidden1)
hidden2 = my_dense_layer(hidden1, n_hidden2)
hidden3 = my_dense_layer(hidden2, n_hidden3)
outputs = my_dense_layer(hidden3, n_outputs, activation=None)

reconstruction_loss = tf.reduce_mean(tf.square(outputs - x))

reg_losses = tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
loss = tf.add_n([reconstruction_loss] + reg_losses)

optimizer = tf.train.AdamOptimizer(learning_rate)
training_op = optimizer.minimize(loss)

init = tf.global_variables_initializer()
```

## 18.5 Автокодировщики. Многослойные автокодировщики. Реализация с помощью TensorFlow

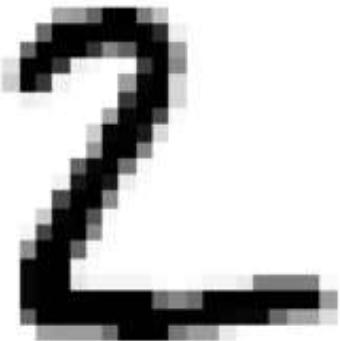
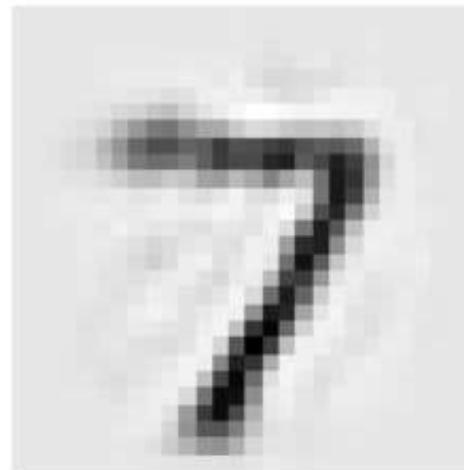
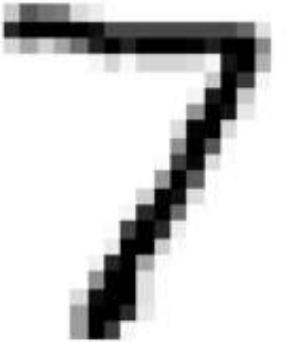
- Затем модель можно обучить обычным образом. Обратите внимание, что метки цифр (`y batch`) не используются:

```
n_epochs = 5
batch_size = 150

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        n_batches = mnist.train.num_examples // batch_size
        for iteration in range(n_batches):
            print("\r{}%".format(100 * iteration // n_batches), end="")
            sys.stdout.flush()
            X_batch, _ = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch})
            loss_train = reconstruction_loss.eval(feed_dict={X: X_batch})
            print("\r{}".format(epoch), "Train MSE:", loss_train)
            saver.save(sess, "./my_model_all_layers.ckpt")
```

```
0 Train MSE: 0.0204011
19% Train MSE: 0.0114192
2 Train MSE: 0.0102221
3 Train MSE: 0.00989991
4 Train MSE: 0.0103724
```

## 18.5 Автокодировщики. Многослойные автокодировщики. Реализация с помощью TensorFlow



## 18.5 Автокодировщики. Многослойные автокодировщики. Реализация с помощью TensorFlow

### □ Визуализация признаков

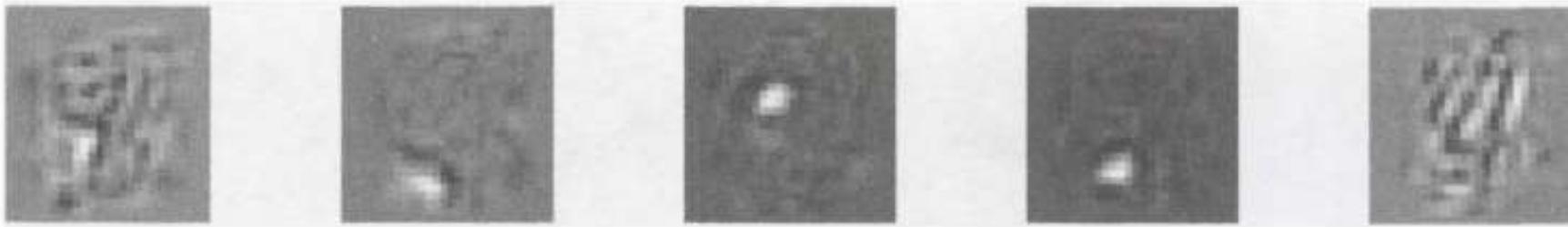
После того как автокодировщик выучил определенные признаки, может возникнуть желание взглянуть на них. Рассмотрим следующий прием. Для каждого нейрона в первом скрытом слое вы можете создать изображение, в котором яркость пикселя соответствует весу связи с заданным нейроном. Например, следующий код вычерчивает признаки, найденные пятью нейронами из первого скрытого слоя:

```
with tf.Session() as sess:  
    saver.restore(sess, "./my_model_one_at_a_time.ckpt") # Training  
    weights1_val = weights1.eval()  
  
    for i in range(5):  
        plt.subplot(1, 5, i + 1)  
        plot_image(weights1_val.T[i])  
  
    save_fig("extracted_features_plot")  
    plt.show()
```

## 18.5 Автокодировщики. Многослойные автокодировщики. Реализация с помощью TensorFlow

### Визуализация признаков

Первые четыре признака, похоже, соответствуют небольшим пятнам, в то время как пятый признак, кажется, ищет вертикальные штрихи.



## 18.5 Автокодировщики. Шумоподавляющие автокодировщики. Реализация с помощью TensorFlow

- Идея использования автокодировщиков для подавления шума существует с 1980-х годов (скажем, в 1987 году Ян Лекун упомянул о ней в своей кандидатской диссертации). В работе Паскаля Винсента и др. ["Extracting and Composing Robust Features with Denoising Autoencoders" ("Выделение и компоновка устойчивых признаков с помощью шумоподавляющих автокодировщиков"), П. Винсент и др. (2008 год) ([https : / /goo . gl/K9rqcx](https://goo.gl/K9rqcx))], опубликованной в 2008 году, было показано, что автокодировщики могли бы применяться также для выделения признаков. В работе Винсента и др.[ "Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion" ("Многослойные шумоподавляющие автокодировщики: обучение глубокой сети выявлению полезных представлений с помощью локального критерия устранения шумов"), П. Винсент и др. (2010 год) ([https : / /goo . gl/HgCDIA](https://goo . gl/HgCDIA))], вышедшей в 2010 году, были представлены многослойные шумоподавляющие автокодировщики (stacked denoising autoencoder).

## 18.5 Автокодировщики. Шумоподавляющие автокодировщики. Реализация с помощью TensorFlow

- Шум может быть чистым гауссовым шумом, добавленным к входам, или случайным выключением входов.



## 18.5 Автокодировщики. Шумоподавляющие автокодировщики. Реализация с помощью TensorFlow

- Реализовать шумоподавляющие автокодировщики в TensorFlow не слишком трудно. Давайте начнем с гауссова шума. На самом деле это похоже на обучение обычновенного автокодировщика, но только к входам добавляется шум, а потеря из-за реконструкции подсчитывается на основе исходных входов:

```
noise_level = 1.0
X = tf.placeholder(tf.float32, shape=[None, n_inputs])
X_noisy = X + noise_level * tf.random_normal(tf.shape(X))

hidden1 = tf.layers.dense(X_noisy, n_hidden1, activation=tf.nn.relu,
                         name="hidden1")
[...]
reconstruction_loss = tf.reduce_mean(tf.square(outputs - X)) # MSE
[...]
```

- Поскольку на стадии построения форма X определяется только частично, мы не знаем заранее форму шума, который должен быть добавлен к X. Мы не можем вызывать X.get\_shape(), потому что в результате возвратилась бы только частично определенная форма X([None , n\_inputs]), а функция random\_normal() ожидает полностью определенной формы и потому сгенерирует исключение. Взамен мы вызываем функцию tf.shape(X), создающую операцию, которая будет возвращать форму X во время выполнения, полностью определенную на тот момент.

## 18.5 Автокодировщики. Шумоподавляющие автокодировщики. Реализация с помощью TensorFlow

- Реализовать версию с отключением, которая более распространена, не намного труднее:

```
dropout_rate = 0.3

training=tf.placeholder_with_default(False, shape=(), name='training')

X = tf.placeholder(tf.float32, shape=[None, n_inputs])
X_drop = tf.layers.dropout(X, dropout_rate, training=training)

hidden1 = tf.layers.dense(X_drop, n_hidden1, activation=tf.nn.relu,
                         name="hidden1")
[...]
reconstruction_loss = tf.reduce_mean(tf.square(outputs - X)) # MSE
[...]
```

## 18.6 Автокодировщики. Вариационные автокодировщики

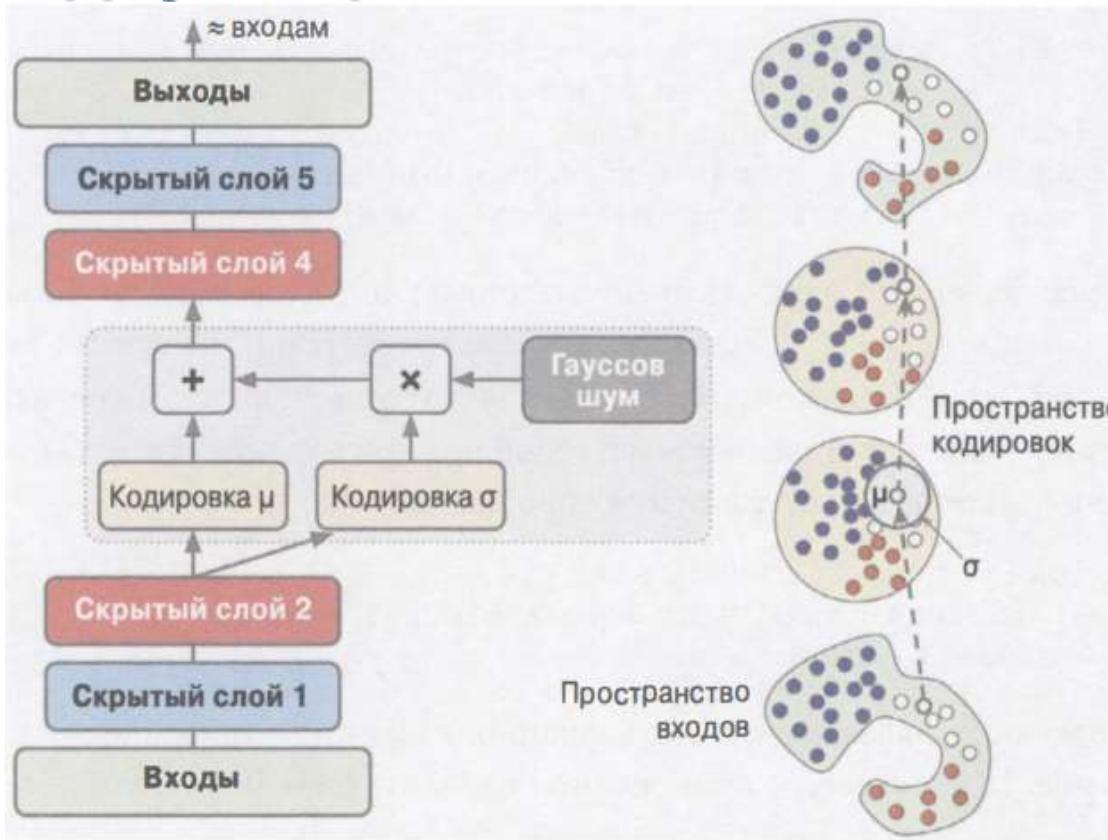
- **Вариационные автокодировщики** (variational autoencoder – VAE) были предложены сравнительно недавно в работе Кингма и Уэллинга (Kingma and Welling, 2014 ["Auto-Encoding Variational Bayes" ("Автокодирование на основе вариационного байесовского подхода"), Д.Кингма и М.Уэллинг (2014 год) (<https://goo.gl/NZq7r2>)]. *Auto-Encoding Variational Bayes* (VAE) похожи на сжимающие и шумоподавляющие автокодировщики в том смысле, что обучаются реконструировать свой вход без учителя. Однако механизм обучения VAE совершенно иной. В сжимающих и шумоподавляющих автокодировщиках значения активации одного слоя отображаются на значения активации следующего слоя, как в стандартной нейронной сети. А в VAE на прямом проходе применяется вероятностный подход.
- Сеть обучается (методом обратного распространения) максимизировать нижнюю границу маргинального правдоподобия обучающих данных,  $\log p(x_1, \dots, x_m)$ .
- Модель VAE также была обобщена на обучение без учителя на временных рядах – это называется вариационным рекуррентным автокодировщиком.

## 18.6 Автокодировщики. Вариационные автокодировщики

- Основные два отличия вариационных автокодировщиков от автокодировщиков, которые обсуждались до сих пор:
  - Они являются вероятностными автокодировщиками, т.е. их выходы отчасти определяются случайно, даже после обучения (в противоположность шумоподавляющим автокодировщикам, которые действуют случайность только во время обучения).
  - Более того, они являются порождающими автокодировщиками, т.е. могут генерировать новые образцы, которые выглядят так, будто они были выбраны из обучающего набора.
- Обе характеристики делают вариационные автокодировщики похожими на ограниченные машины Больцмана, но их проще обучать, а процесс выборки намного быстрее (в случае ограниченных машин Больцмана необходимо ждать, пока сеть придет в «энергетическое равновесие», прежде чем можно будет выбирать новый образец).

## 18.6 Автокодировщики. Вариационные автокодировщики

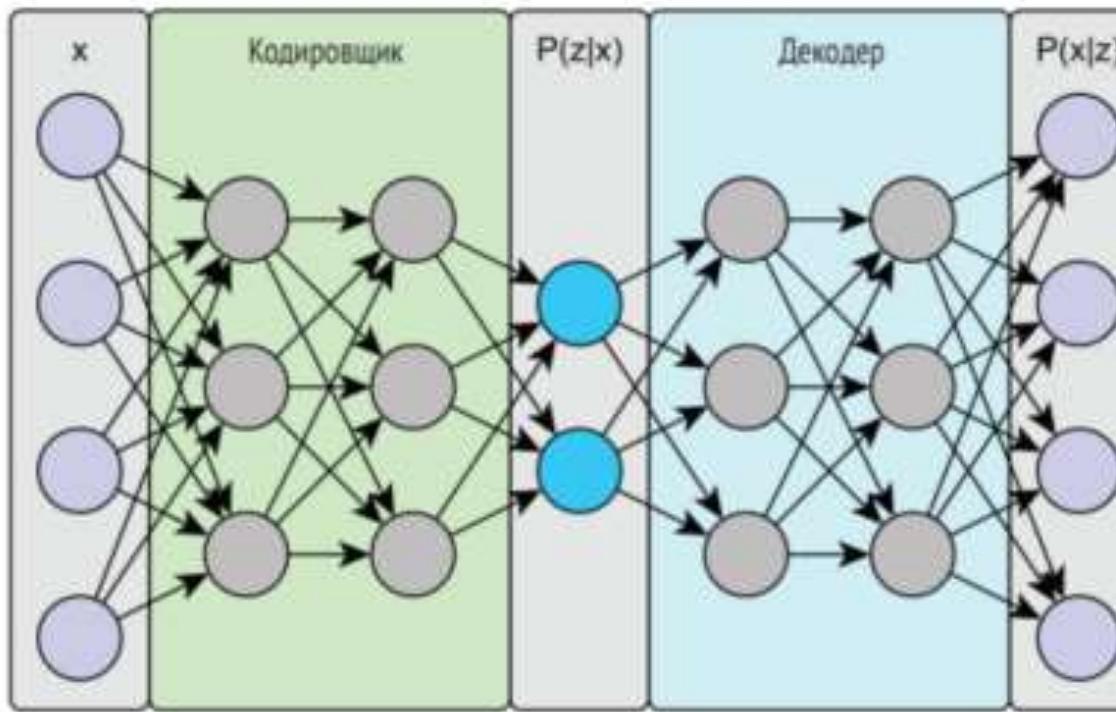
- VAE имеет такую же базовую структуру, но вместо выпуска кодировки для заданного входа напрямую кодировщик выдает среднюю кодировку  $\mu$  и стандартное отклонение  $\sigma$ .
- Действительная кодировка затем выбирается случайным образом из гауссова распределения со средним  $\mu$  и стандартным отклонением  $\sigma$ .
- Далее декодировщик просто декодирует выбранную кодировку как обычно.
- В правой части рис. представлен образец, проходящий через данный автокодировщик. Сначала кодировщик вырабатывает  $\mu$  и  $\sigma$ , затем случайным образом выбирается кодировка (обратите внимание, что она расположена не точно в позиции  $\mu$ ), а в заключение эта кодировка декодируется, и финальный выход напоминает обучающий образец. На диаграмме видно, что хотя входы могут иметь весьма спиралевидное распределение, вариационный автокодировщик стремится выдавать кодировки, которые выглядят так, будто они были выбраны из простого гауссова распределения. В общем случае кодировки не ограничиваются гауссовым распределением.



проходящий через данный автокодировщик. Сначала кодировщик вырабатывает  $\mu$  и  $\sigma$ , затем случайным образом выбирается кодировка (обратите внимание, что она расположена не точно в позиции  $\mu$ ), а в заключение эта кодировка декодируется, и финальный выход напоминает обучающий образец. На диаграмме видно, что хотя входы могут иметь весьма спиралевидное распределение, вариационный автокодировщик стремится выдавать кодировки, которые выглядят так, будто они были выбраны из простого гауссова распределения. В общем случае кодировки не ограничиваются гауссовым распределением.

## 18.6 Автокодировщики. Вариационные автокодировщики

- Если говорить немного более формально, то в модели VAE предполагается, что данные  $x$  порождаются в два приема: (а) из априорного распределения порождается значение  $z^i \sim p(z)$  и (б) порождается пример в соответствии с некоторым условным распределением  $x^i \sim p(x|z)$ .
- Конечно, мы не знаем фактических значений  $z$ , и точный вывод  $p(z|x)$  – вычислительно неразрешимая задача. Чтобы справиться с этой проблемой, мы аппроксимируем оба распределения,  $p(z|x)$  и  $p(x|z)$ , нейронными сетями – соответственно кодировщиком и декодером. Например, если  $P(z|x)$  – нормальное распределение, то прямой проход кодировщика даст его параметры  $\mu$  и  $\sigma$ . Аналогично параметры распределения  $P(x|z)$  дает проход декодера.



## 18.6 Автокодировщики. Вариационные автокодировщики

- Вариационный кодировщик превращает входное изображение в параметры статистического распределения: среднее и дисперсию. По сути, это означает предположение, что входное изображение было сгенерировано статистическим процессом и что случайную составляющую этого процесса необходимо учитывать в ходе кодирования и декодирования. Вариационный автокодировщик затем использует среднее и дисперсию как параметры для случайного отбора одного элемента из распределения и декодирует его обратно в оригинальный вход. Стохастичность этого процесса повышает надежность и заставляет скрытое пространство кодировать значимые представления: каждая точка, выбранная в скрытом пространстве, декодируется в допустимый вывод.



## 18.6 Автокодировщики. Вариационные автокодировщики

- Вот как работает вариационный автокодировщик с технической точки зрения:
  1. Модуль кодирования превращает выборки из входного изображения `input_img` в два параметра в скрытом пространстве, `z_mean` и `z_log_variance`.
  2. Вы выбираете из скрытого нормального распределения произвольную точку `z` для генерации входного изображения как  $z = z_{\text{mean}} + \exp(z_{\log_{\text{variance}}}) * \epsilon$ , где `epsilon` — это случайный тензор небольших значений.
  3. Модуль декодера отображает эту точку из скрытого пространства обратно в оригинальное изображение.



## 18.6 Автокодировщики. Вариационные автокодировщики

- Поскольку  $\epsilon$  является случайным тензором, процесс гарантирует, что каждая точка, близкая к скрытому местоположению, где закодировано  $\text{input\_img}$  ( $z$ -mean), может быть декодирована в нечто, похожее на  $\text{input\_img}$ , что обеспечивает непрерывную значимость скрытого пространства. Любые две близкие точки в скрытом пространстве будут декодированы в очень похожие изображения. Непрерывность в сочетании с малой размерностью скрытого пространства заставляет каждое направление в скрытом пространстве кодировать значимую ось изменений данных, что делает скрытое пространство высокоструктурированным и прекрасно подходящим для манипуляций посредством концептуальных векторов.
- Параметры вариационного автокодировщика обучаются на двух видах функций потерь:
  - **потерях восстановления** (reconstruction loss), которая заставляет декодированные образцы совпадать с исходными входами,
  - и **потерях регуляризации** (regularization loss) или **латентных потерях** (latent loss), которая помогает извлекать хорошо сформированные скрытые пространства и ослабляет проблему переобучения на обучающих данных. Вторая часть вынуждает автокодировщик иметь кодировки, которые выглядят так, будто они были выбраны из простого гауссова распределения, для чего мы используем расстояние Кульбака-Лейблера ([KL divergence](#)) между целевым (гауссовым) распределением и фактическим распределением кодировок.

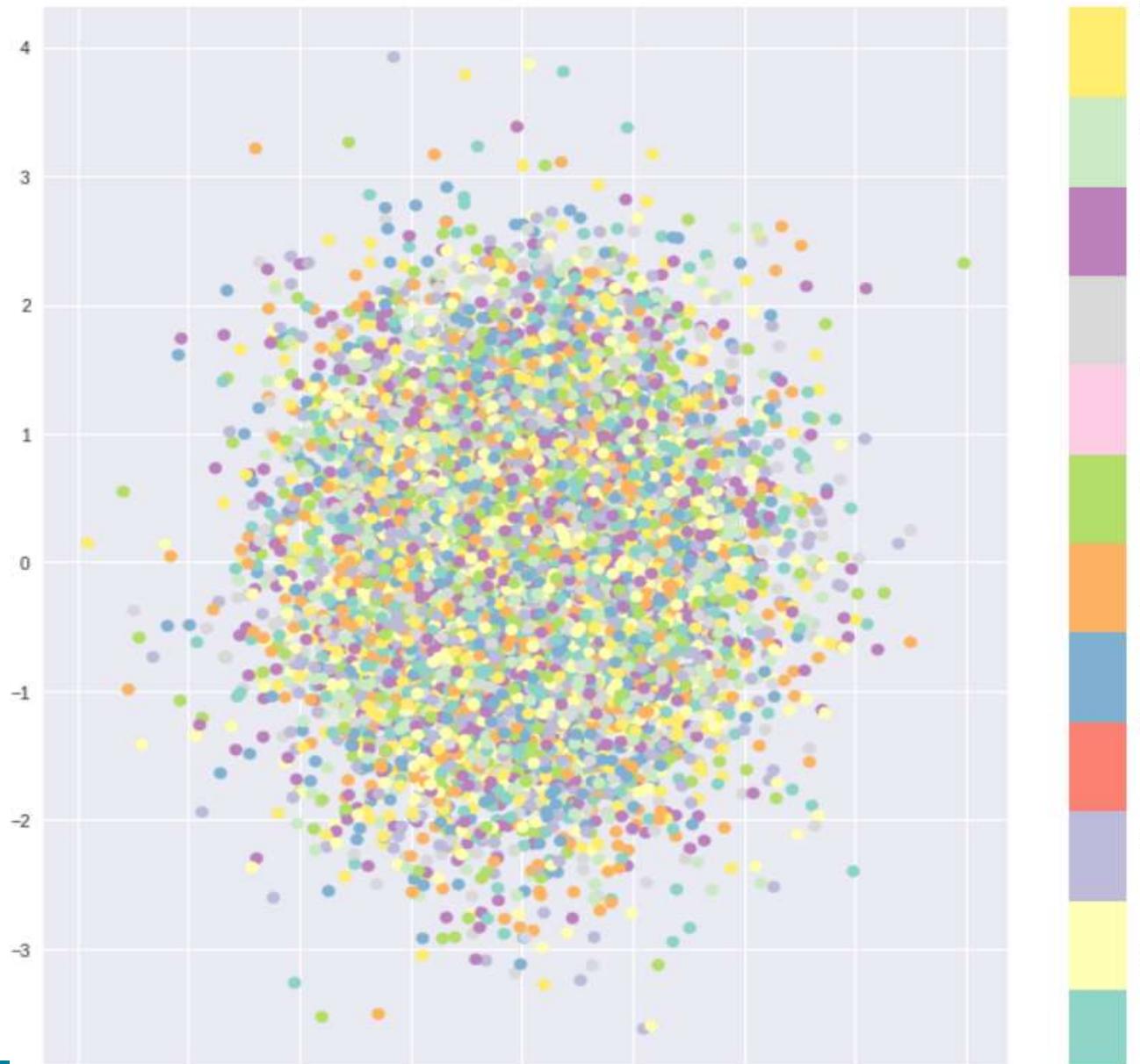
## 18.6 Автокодировщики. Вариационные автокодировщики. 18.6.1 Пример реализации в Tensorflow

$$\sum_{i=1}^n \sigma_i^2 + \mu_i^2 - \log(\sigma_i) - 1$$

- Мы хотим, чтобы все области в скрытом пространстве были как можно ближе друг к другу, но при этом оставались различимыми как отдельные составляющие. В этом случае мы можем производить гладкую интерполяцию и создавать новые данные на выходе.
- KL расходимость между двумя функциями распределения показывает насколько сильно они отличаются друг от друга. Минимизация KL расходимости означает оптимизацию параметров распределения  $\mu$  и  $\sigma$  таким образом, что они становятся близки к параметрам целевого распределения.
- Для VAE KL потери эквивалентны сумме всех KL расходимостей между распределением компонент  $X_i \sim N(\mu_i, \sigma_i^2)$  в векторе  $X$  и нормальным (гауссовым) распределением. Минимум достигается, когда  $\mu_i = 0$  и  $\sigma_i = 1$ .

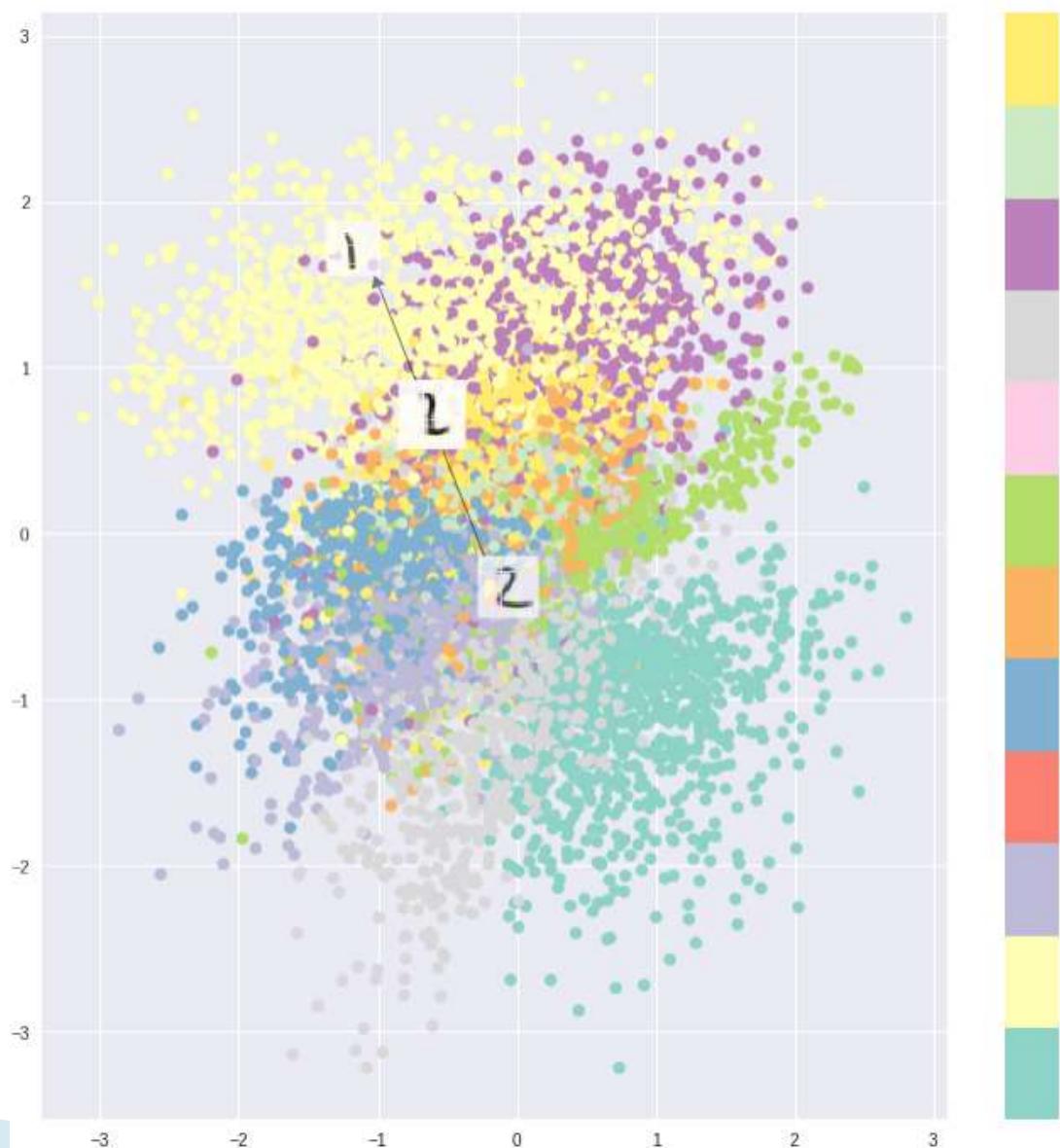
## 18.6 Автокодировщики. Вариационные автокодировщики. 18.6.1 Пример реализации в Tensorflow

- При использовании KL потерь области кодирования расположены случайным образом в окрестности выделенной точки в скрытом пространстве со слабым учётом сходства между образцами входных данных. Поэтому декодер не способен извлечь что-либо значащее из этого пространства:



## 18.6 Автокодировщики. Вариационные автокодировщики. 18.6.1 Пример реализации в Tensorflow

- Однако, оптимизируя и энкодер и декодер, мы получаем скрытое пространство, которое отражает схожесть соседних векторов на глобальном уровне, и имеет вид плотно расположенных областей возле начала координат скрытого пространства:
- Достигнутый результат – это компромисс между кластерной природой потерь восстановления, необходимой декодеру, и нашим желанием иметь плотно расположенные векторы при использовании KL потерь, т.е. оптимизации по обоим ф-ям потерь.



## 18.6 Автокодировщики. Вариационные автокодировщики. 18.6.1 Пример реализации в Tensorflow

- Создание модели VAE сети:

```
from functools import partial

n_inputs = 28 * 28
n_hidden1 = 500
n_hidden2 = 500
n_hidden3 = 20 # codings
n_hidden4 = n_hidden2
n_hidden5 = n_hidden1
n_outputs = n_inputs
learning_rate = 0.001

initializer = tf.contrib.layers.variance_scaling_initializer()

my_dense_layer = partial(
    tf.layers.dense,
    activation=tf.nn.elu,
    kernel_initializer=initializer)

X = tf.placeholder(tf.float32, [None, n_inputs])
hidden1 = my_dense_layer(X, n_hidden1)
hidden2 = my_dense_layer(hidden1, n_hidden2)
hidden3_mean = my_dense_layer(hidden2, n_hidden3, activation=None)
hidden3_sigma = my_dense_layer(hidden2, n_hidden3, activation=None)
noise = tf.random_normal(tf.shape(hidden3_sigma), dtype=tf.float32)
hidden3 = hidden3_mean + hidden3_sigma * noise
hidden4 = my_dense_layer(hidden3, n_hidden4)
hidden5 = my_dense_layer(hidden4, n_hidden5)
logits = my_dense_layer(hidden5, n_outputs, activation=None)
outputs = tf.sigmoid(logits)
```

## 18.6 Автокодировщики. Вариационные автокодировщики. 18.6.1 Пример реализации в Tensorflow

- Вычисление функции потерь и определение оптимизатора:

```
xentropy = tf.nn.sigmoid_cross_entropy_with_logits(labels=x, logits=logits)
reconstruction_loss = tf.reduce_sum(xentropy)
```

```
eps = 1e-10 # smoothing term to avoid computing log(0) which is NaN
latent_loss = 0.5 * tf.reduce_sum(
    tf.square(hidden3_sigma) + tf.square(hidden3_mean)
    - 1 - tf.log(eps + tf.square(hidden3_sigma)))
```

```
loss = reconstruction_loss + latent_loss

optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(loss)

init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

## 18.6 Автокодировщики. Вариационные автокодировщики. 18.6.1 Пример реализации в Tensorflow

### □ Обучение:

```
n_epochs = 50
batch_size = 150

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        n_batches = mnist.train.num_examples // batch_size
        for iteration in range(n_batches):
            print("\r{}%".format(100 * iteration // n_batches), end="")
            sys.stdout.flush()
            x_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: x_batch})
            loss_val, reconstruction_loss_val, latent_loss_val =
                sess.run([loss, reconstruction_loss, latent_loss], feed_dict={X: x_batch})
            print("\r{}".format(epoch), "Train total loss:", loss_val,
                  "\tReconstruction loss:", reconstruction_loss_val, "\tLatent loss:", latent_loss_val)
    saver.save(sess, "./my_model_variational.ckpt")
```

## 18.6 Автокодировщики. Вариационные автокодировщики. 18.6.1 Пример реализации в Tensorflow

- Если использовать построенную VAE сеть для генерации цифр, то мы получим следующий результат:



## 18.6 Автокодировщики. Вариационные автокодировщики. 18.6.2 Пример реализации в Keras

- Давайте пройдемся по реализации вариационного автокодировщика в Keras.  
Схематически она выглядит так:

```
z_mean, z_log_variance = encoder(input_img)           ← Кодирование входа  
z = z_mean + exp(z_log_variance) * epsilon          в среднее и дисперсию  
reconstructed_img = decoder(z) ← Декодирование z обратно в изображение  
model = Model(input_img, reconstructed_img) ← Создание модели автокоди-  
Извлечение скрытой точки с использованием        ровщика, которая отображает  
случайной величины epsilon                         входное изображение в его  
                                                       реконструкцию
```

- Затем можно обучить модель, используя потери восстановления и потери регуляризации.

## 18.6 Автокодировщики. Вариационные автокодировщики. 18.6.2 Пример реализации в Keras

- В следующем листинге демонстрируется используемая нами сеть кодировщика, отображающая изображения в параметры распределения вероятности в скрытом пространстве. Эта простая сверточная сеть отображает входное изображение  $x$  в два вектора,  $z\_mean$  и  $z\_log\_var$ .

```
import keras
from keras import layers
from keras import backend as K
from keras.models import Model
import numpy as np

img_shape = (28, 28, 1)
batch_size = 16

latent_dim = 2 ← Размерность скрытого пространства: двумерная плоскость

input_img = keras.Input(shape=img_shape)
x = layers.Conv2D(32, 3,
                  padding='same', activation='relu')(input_img)
x = layers.Conv2D(64, 3,
                  padding='same', activation='relu',
                  strides=(2, 2))(x)
x = layers.Conv2D(64, 3,
                  padding='same', activation='relu')(x)
x = layers.Conv2D(64, 3,
                  padding='same', activation='relu')(x)
shape_before_flattening = K.int_shape(x)

x = layers.Flatten()(x)
x = layers.Dense(32, activation='relu')(x)

z_mean = layers.Dense(latent_dim)(x) | Входное изображение кодируется
z_log_var = layers.Dense(latent_dim)(x) | в эти два параметра
```

## 18.6 Автокодировщики. Вариационные автокодировщики. 18.6.2 Пример реализации в Keras

- Далее приводится код, использующий z\_mean и z\_log\_var, параметры статистического распределения, которое, как предполагается, произвело input\_img, для создания точки z скрытого пространства. Здесь мы обернули некоторый произвольный код (основанный на примитивах Keras) в слой Lambda.
- **Функция выбора точки из скрытого пространства**

```
def sampling(args):
    z_mean, z_log_var = args
    epsilon = K.random_normal(shape=(K.shape(z_mean)[0], latent_dim),
                               mean=0., stddev=1.)
    return z_mean + K.exp(z_log_var) * epsilon

z = layers.Lambda(sampling)([z_mean, z_log_var])
```

## 18.6 Автокодировщики. Вариационные автокодировщики. 18.6.2 Пример реализации в Keras

- Следующий листинг демонстрирует реализацию декодера. Здесь мы приводим размерность вектора z в соответствие с размерами изображения и затем используем несколько сверточных слоев, чтобы получить выходное изображение с теми же размерами, что и оригинальное input\_img.

```
decoder_input = layers.Input(K.int_shape(z)[1:]) ← Передача z на вход  
  
x = layers.Dense(np.prod(shape_before_flattening[1:]), activation='relu')(decoder_input) | Увеличение  
                                                               | разрешения входа  
  
→ x = layers.Reshape(shape_before_flattening[1:])(x)
```

```
x = layers.Conv2DTranspose(32, 3,  
                           padding='same',  
                           activation='relu',  
                           strides=(2, 2))(x)
```

```
x = layers.Conv2D(1, 3,  
                  padding='same',  
                  activation='sigmoid')(x)
```

Использование слоев  
Conv2DTranspose и Conv2D  
для декодирования z в карту  
признаков с тем же размером,  
что и входное изображение

```
decoder = Model(decoder_input, x)
```

```
z_decoded = decoder(z) ←
```

Преобразование z в карту  
признаков с той же формой,  
которую имела карта признаков  
перед последним слоем Flatten  
в модели кодировщика

Применяет  
декодер к z, чтобы  
восстановить  
декодированное  
значение z

Создается модель  
декодера, которая  
преобразует  
"decoder\_input"  
в декодированное  
изображение

## 18.6 Автокодировщики. Вариационные автокодировщики. 18.6.2 Пример реализации в Keras

- Двойственная природа потерь VAE не соответствует традиционной форме `loss(input, target)`. Поэтому мы реализуем вычисление потерь, написав свой слой, который внутренне использует встроенный метод `add_loss` слоя для создания произвольных потерь.

```
class CustomVariationalLayer(keras.layers.Layer):  
  
    def vae_loss(self, x, z_decoded):  
        x = K.flatten(x)  
        z_decoded = K.flatten(z_decoded)  
        xent_loss = keras.metrics.binary_crossentropy(x, z_decoded)  
        kl_loss = -5e-4 * K.mean(  
            1 + z_log_var - K.square(z_mean) - K.exp(z_log_var), axis=-1)  
        return K.mean(xent_loss + kl_loss)
```

```
def call(self, inputs): ← Собственные слои реализуются  
    x = inputs[0]          определением метода call  
    z_decoded = inputs[1]  
    loss = self.vae_loss(x, z_decoded)  
    self.add_loss(loss, inputs=inputs)  
    return x
```

```
y = CustomVariationalLayer()([input_img, z_decoded]) ←
```

Мы не используем этот  
результат, однако слой должен  
что-то возвращать

Вызов собственного слоя  
с исходными и декодированными  
данными для получения  
окончательного вывода модели

## 18.6 Автокодировщики. Вариационные автокодировщики. 18.6.2 Пример реализации в Keras

- **Создание и обучение модели.** Поскольку вычислением потерь у нас занимается собственный слой, мы не указываем функцию потерь на этапе компиляции (loss=None). Это, в свою очередь, означает, что нам не нужно передавать целевые данные в процесс обучения (как можно заметить, в метод fit обучаемой модели передается только x\_train).

```
from keras.datasets import mnist

vae = Model(input_img, y)
vae.compile(optimizer='rmsprop', loss=None)
vae.summary()

(x_train, _), (x_test, y_test) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_train = x_train.reshape(x_train.shape + (1,))
x_test = x_test.astype('float32') / 255.
x_test = x_test.reshape(x_test.shape + (1,))

vae.fit(x=x_train, y=None,
        shuffle=True,
        epochs=10,
        batch_size=batch_size,
        validation_data=(x_test, None))
```

## 18.6 Автокодировщики. Вариационные автокодировщики. 18.6.2 Пример реализации в Keras

- После обучения такой модели — в данном случае на наборе MNIST — мы можем использовать сеть decoder для превращения произвольных векторов из скрытого пространства в изображения.
- **Выбор сетки с точками из двумерного скрытого пространства и их декодирование в изображения**

```
import matplotlib.pyplot as plt
from scipy.stats import norm

n = 15                                Будет отображаться
digit_size = 28                           сетка 15 × 15 цифр
figure = np.zeros((digit_size * n, digit_size * n))    (всего 225 цифр)

grid_x = norm.ppf(np.linspace(0.05, 0.95, n))           Преобразует координаты линей-
grid_y = norm.ppf(np.linspace(0.05, 0.95, n))           ного пространства с использова-
                                                               нием функции ppf из пакета SciPy
                                                               для получения значений скрытой
                                                               переменной z (поскольку пред-
                                                               шествующее скрытое простран-
                                                              ство является гауссовым)

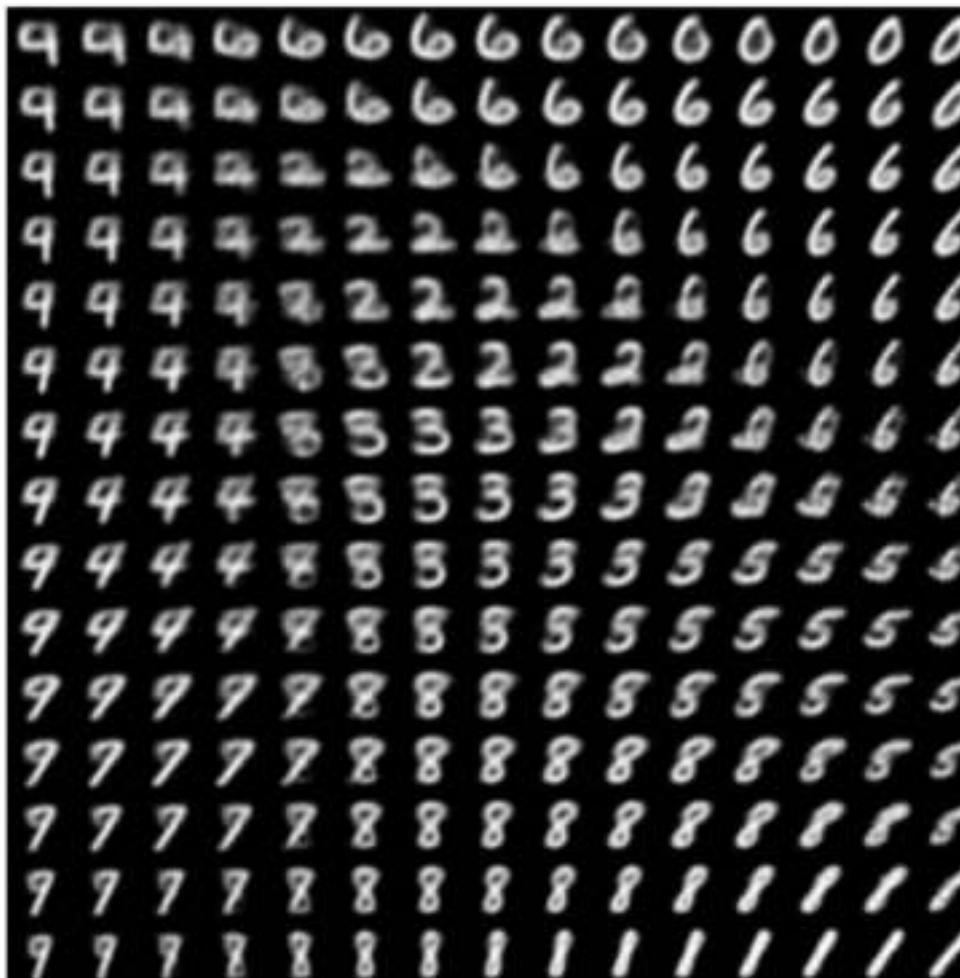
for i, yi in enumerate(grid_x):
    for j, xi in enumerate(grid_y):
        z_sample = np.array([[xi, yi]])                  Многократное повторение выбора z
        z_sample = np.tile(z_sample, batch_size).reshape(batch_size, 2)   для формирования полного пакета
        x_decoded = decoder.predict(z_sample, batch_size=batch_size)
        digit = x_decoded[0].reshape(digit_size, digit_size)
        figure[i * digit_size: (i + 1) * digit_size,
               j * digit_size: (j + 1) * digit_size] = digit

plt.figure(figsize=(10, 10))
plt.imshow(figure, cmap='Greys_r')
plt.show()                                Преобразование первой цифры
                                            в пакете из размерности 28 × 28 × 1
                                            в 28 × 28

                                            Декодирование пакета
                                            в изображения цифр
```

## 18.6 Автокодировщики. Вариационные автокодировщики. 18.6.2 Пример реализации в Keras

- Сетка выбранных цифр демонстрирует полностью непрерывное распределение разных классов цифр, где одна цифра превращается в другую по пути через непрерывное скрытое пространство.
- Сетка с цифрами, декодированными из скрытого пространства



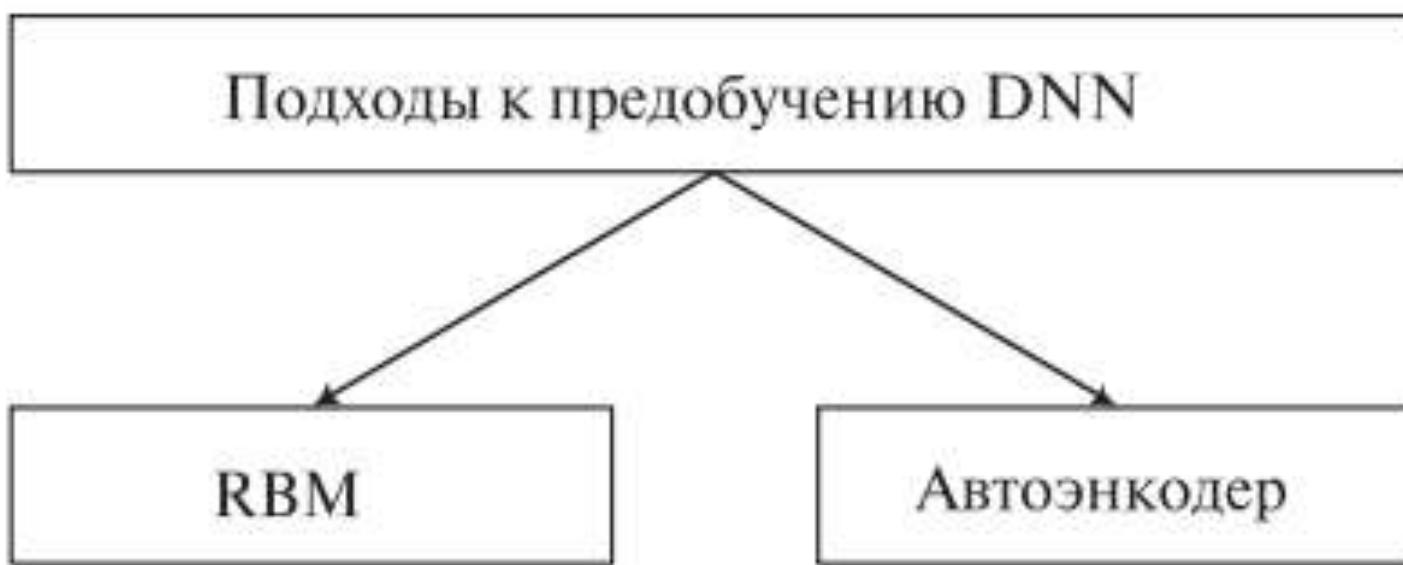
# 19 Обучение глубоких нейронных сетей

# 19 Обучение глубоких нейронных сетей

- Можно выделить два основных подхода к обучению глубоких нейронных сетей:
  - ✓ *Метод с предварительным обучением*, который состоит из двух этапов:
    - предобучение нейронной сети методом постепенного обучения, начиная с первого слоя (pre-training). Данное обучение осуществляется без учителя и базируется на модели ограниченной машины Больцмана (RBM) или автоэнкодерном подходе;
    - настройка синаптических связей всей сети (fine-tuning) при помощи алгоритма обратного распространения ошибки.
  - ✓ *Метод стохастического градиента* (SGD) или ему подобные.
- Принята следующая парадигма для обучения глубоких нейронных сетей:
  - ✓ Если обучающая выборка большая, т. е. размерность обучающей выборки намного больше, чем количество настраиваемых параметров сети, то используется метод стохастического градиента.
  - ✓ Если размерность обучающей выборки сравнима с количеством настраиваемых параметров сети, то применяется предварительное обучение нейронной сети и алгоритм обратного распространения ошибки для точной настройки синаптических связей сети (fine-tuning).

## 19 Обучение глубоких нейронных сетей

- Важным этапом обучения глубоких нейронных является предобучение слоев нейронной сети. Существует два основных подхода к предварительному обучению слоев глубоких нейронных сетей:
  - ✓ Первый подход основан на представлении каждого слоя нейронной сети в виде *ограниченной машины Больцмана*.
  - ✓ *Автоэнкодерный* подход базируется на представлении каждого слоя в виде *автоассоциативной* нейронной сети;

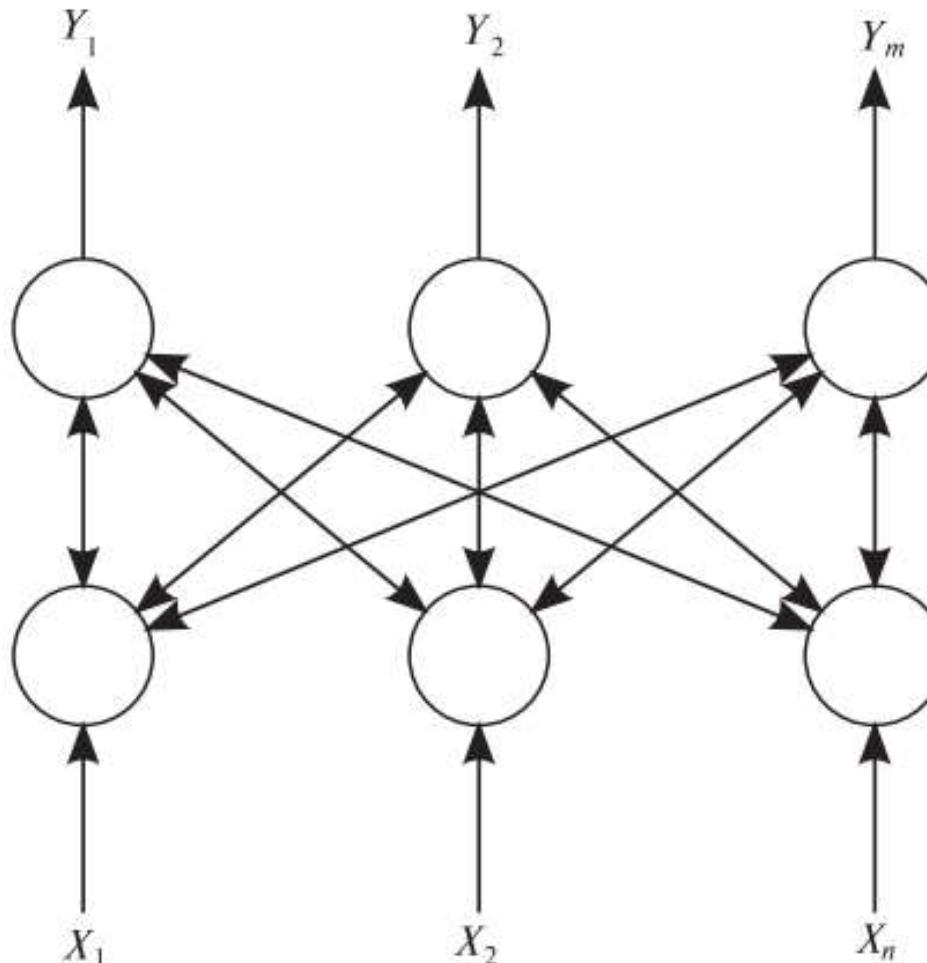


## 19.1 Ограниченные машины Больцмана

- **Ограниченные машины Больцмана** ( ОМБ, RMB – Restricted Boltzmann Machine) — вид генеративной стохастической нейронной сети, которая определяет распределение вероятности на входных образцах данных.
- Первая ограниченная машина Больцмана была построена в 1986 году Полом Смоленски под названием *Harmonium* (Smolensky, Paul. Chapter 6: Information Processing in Dynamical Systems: Foundations of Harmony Theory // Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations (англ.) / Rumelhart, David E.; McLelland, James L.. — MIT Press, 1986. — Р. 194—281. — ISBN 0-262-68053-X.), но приобрела популярность только после изобретения Д.Хинтоном быстрых алгоритмов обучения в середине 2000-х годов.
- RBM применяются для следующих целей:
  - ✓ предварительного обучения глубоких сетей;
  - ✓ выделения признаков;
  - ✓ понижение размерности.
- Слово «ограниченные» в названии означает, что связи между блоками одного слоя запрещены (т. е. отсутствуют прямые взаимодействия между любыми видимыми или любыми двумя скрытыми блоками). Джеффри Хинтон, пионер глубокого обучения, популяризировавший RBM 29 десять лет назад, описывает более общую машину Больцмана следующим образом:  
Сеть, состоящая из симметрично связанных нейроноподобных блоков, которые принимают стохастические решения о своем состоянии: возбужден или нет.

## 19.1 Ограниченные машины Больцмана

- Базовая RBM состоит из двух слоев стохастических бинарных нейронных элементов, которые соединены между собой двунаправленными симметричными связями. Входной слой нейронных элементов называется видимым (слой  $X$ ), а второй слой – скрытым (слой  $Y$ ).
- Глубокую нейронную сеть можно представить как совокупность ограниченных машин Больцмана. Ограниченная машина Больцмана может аппроксимировать (генерировать) любое дискретное распределение, если используется достаточно большое количество нейронов скрытого слоя (*Bengio Y. Learning deep architectures for AI // Foundations and trends in machine learning. – 2009. – № 2(1). – Р. 1–127.*)



## 19.1 Ограниченные машины Больцмана

- RBM можно определить без ограничения общности как *стохастическую нейронную сеть*, в которой состояния видимых и скрытых нейронов меняются в соответствии с вероятностной версией сигмоидной функции активации:

$$p(y_j | x) = \frac{1}{1 + e^{-S_j}}, \quad S_j = \sum_{i=1}^n w_{ij} x_i + T_j, \quad (19.1)$$

$$p(x_i | x) = \frac{1}{1 + e^{-S_i}}, \quad S_i = \sum_{j=1}^m w_{ij} y_j + T_i. \quad (19.2)$$

- Состояния видимых и скрытых нейронных элементов принимаются независимыми:

$$P(x | y) = \prod_{i=1}^n P(x_i | y),$$

$$P(y | x) = \prod_{j=1}^m P(y_j | x).$$

- Таким образом, состояния всех нейронных элементов ограниченной машины Больцмана определяются через распределение вероятностей. В RBM нейроны скрытого слоя – это детекторы признаков, которые обуславливают закономерности входных данных.

# 19.1 Ограниченные машины Больцмана

## Обучение

- Основная задача обучения RBM состоит в воспроизведении распределения входных данных на основе состояний нейронов скрытого слоя как можно точнее. Это эквивалентно максимизации функции правдоподобия путем модификации синаптических связей нейронной сети, как и для классической ВМ.
- Как и для классической ВМ вероятность нахождения видимого и скрытого нейрона в состоянии  $(x, y)$  определяется на основе распределения Гиббса:

$$P(x, y) = \frac{e^{-E(x, y)}}{Z}, \quad (19.3)$$

где  $E(x, y)$  – энергия системы в состоянии  $(x, y)$ ;  $Z$  – параметр условия нормализации вероятностей, сумма вероятностей должна равняться единице. Данный параметр вычисляется следующим образом:

$$Z = \sum_{x, y} e^{-E(x, y)}. \quad (19.4)$$

# 19.1 Ограниченные машины Больцмана

## Обучение

- Для нахождения правила модификации синаптических связей необходимо максимизировать вероятность воспроизведения состояний видимых нейронов  $P(x)$  RBM:

$$P(x) = \sum_y P(x, y) = \sum_y \frac{e^{-E(x, y)}}{Z} = \frac{\sum_y e^{-E(x, y)}}{\sum_{x, y} e^{-E(x, y)}}. \quad (19.5)$$

- По аналогии с классической ВМ будем определять максимум функции логарифмического правдоподобия методом градиентного спуска в пространстве весовых коэффициентов и пороговых значений сети, где в качестве градиента возьмем:

$$\ln P(x) = \ln \sum_y e^{-E(x, y)} - \ln \sum_{x, y} e^{-E(x, y)}. \quad (19.6)$$

- Проводя цепочку рассуждений и преобразований, аналогичных классической ВМ выше, получаем в итоге градиент:

$$\frac{\partial \ln P(x)}{\partial \omega_{ij}} = \sum_y P(y|x)x_iy_j - \sum_{x, y} P(x, y)x_iy_j. \quad (19.7)$$

## 19.1 Ограниченные машины Больцмана

### Обучение

- Поскольку математическое ожидание вычисляется как :

$$E(x) = \sum_i x_i P_i, \quad (19.8)$$

то

$$\frac{\partial \ln P(x)}{\partial \omega_{ij}} = E[x_i y_j]_{\text{data}} - E[x_i y_j]_{\text{model}}. \quad (19.9)$$

- Как следует из выражения (19.9), первое слагаемое характеризует работу сети на основе данных из обучающей выборки, а второе слагаемое – работу сети на основе данных модели (данные, генерируемые сетью), т. е. в свободном режиме независимо от окружающей среды.

Более подробно вывод можно посмотреть в

В.А.Головко, В.В.Краснопрошин. Нейросетевые технологии обработки данных : учеб. Пособие. – Минск : БГУ, 2017. – 263 с.

# 19.1 Ограниченные машины Больцмана

## Обучение

- Поскольку вычисление математического ожидания на основе RBM сети является достаточно сложным, Дж. Хинтон предложил использовать аппроксимацию данных слагаемых, которую он назвал контрастным расхождением (contrastive divergence (CD)) (*Hinton G. E., Osindero S., Teh Y. A fast learning algorithm for deep belief nets // Neural computation. – 2006. – № 18. – Р. 1527–1554.*).)
- Такая аппроксимация основывается на сэмплировании Гиббса (Gibbs sampling), которое также применялось выше для классической ВМ. В этом случае первые слагаемые в выражениях для градиента характеризуют распределение данных в момент времени  $t = 0$ , а вторые – реконструированные или генерируемые моделью состояния в момент времени  $t = k$ . Исходя из этого CD- $k$ -процедура может быть представлена следующим образом:

$$x(0) \rightarrow y(0) \rightarrow x(1) \rightarrow y(1) \rightarrow \dots \rightarrow x(k) \rightarrow y(k).$$

- Сэмплирование по Гиббсу не требует явно выраженное совместное распределение, а нужны лишь условные вероятности для каждой переменной, входящей в распределение. Алгоритм на каждом шаге берет одну случайную величину и выбирает ее значение при условии фиксированных остальных. Можно показать, что последовательность получаемых значений образуют возвратную цепь Маркова, устойчивое распределение которой является как раз искомым совместным распределением.

# 19.1 Ограниченные машины Больцмана

## Обучение

- В результате можно получить следующие правила для обучения RBM-сети. В случае применения CD-1  $k=1$  и учетом того, что в соответствии с методом градиентного спуска

$$\omega_{ij}(t+1) = \omega_{ij}(t) + \alpha \frac{\partial \ln P(x)}{\partial \omega_{ij}(t)},$$

для последовательного обучения имеем

$$\omega_{ij}(t+1) = \omega_{ij}(t) + \alpha (x_i(0)y_j(0) - x_i(1)y_j(1)), \quad (19.10)$$

- Аналогичным образом для алгоритма CD- $k$

$$\omega_{ij}(t+1) = \omega_{ij}(t) + \alpha (x_i(0)y_j(0) - x_i(k)y_j(k)). \quad (19.11)$$

- Для группового обучения и алгоритма CD- $k$

$$\omega_{ij}(t+1) = \omega_{ij}(t) + \alpha \sum_{l=1}^L (x_i^l(0)y_j^l(0) - x_i^l(k)y_j^l(k)). \quad (19.12)$$

Из последних выражений видно, что правила обучения ограниченной машины Больцмана минимизируют разницу между оригиналными данными и результатами, генерируемыми моделью. Генерируемые моделью значения получаются при помощи сэмплирования Гиббса.

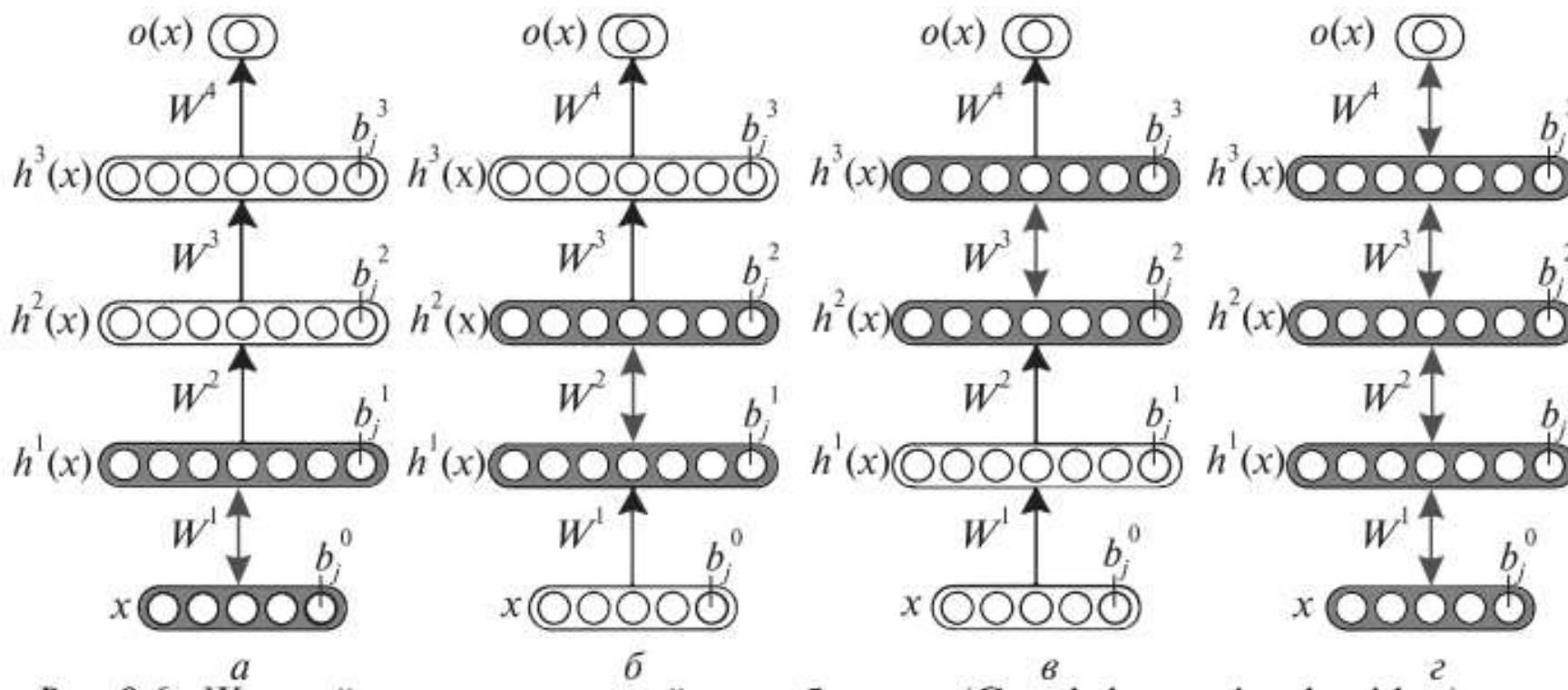
## 19.1 Ограниченные машины Больцмана

### Обучение

- Обучение нейронной сети глубокого доверия происходит на основе «жадного» алгоритма послойного обучения (greedy layer-wise algorithm). В соответствии с ним вначале обучается первый слой сети как RBM-машина. Для этого входные данные поступают на видимый слой нейронных элементов и с использованием CD- $k$  процедуры вычисляются состояния скрытых  $p(y|x)$  и видимых нейронов  $p(x|y)$ . В процессе выполнения данной процедуры (не более 100 эпох) изменяются весовые коэффициенты и пороговые значения RBM-сети, которые затем фиксируются. Затем берется второй слой нейронной сети и конструируется RBM-машина. Входными данными для нее являются данные с предыдущего слоя. Происходит обучение, и процесс продолжается для всех слоев нейронной сети.
- В результате такого обучения без учителя можно получить подходящую начальную инициализацию настраиваемых параметров глубокой нейронной сети. На заключительном этапе осуществляется точная настройка параметров всей сети при помощи алгоритма обратного распространения ошибки.

# 19.1 Ограниченные машины Больцмана

## Обучение



- «Жадный» алгоритм послойного обучения (Greedy layer-wise algorithm):
  - а* – преобразование 1-го скрытого слоя; *б* – преобразование 2-го скрытого слоя;
  - в* – преобразование 3-го скрытого слоя; *г* – точная настройка всей сети

## 19.2 Ограниченные машины Больцмана. Пример реализации в Tensorflow и работы на данных MNIST

- Загружаем необходимые библиотеки

```
import numpy as np
import pandas as pd
import tensorflow as tf
import matplotlib.pyplot as plt
%matplotlib inline
```

- Читаем данные из датасета MNIST

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data", one_hot=True)
```

- Определяем параметры обучения сети

```
n_visible      = 784
n_hidden       = 500
display_step   = 1
num_epochs     = 200
batch_size     = 256
lr             = tf.constant(0.001, tf.float32)
```

## 19.2 Ограниченные машины Больцмана. Пример реализации в Tensorflow и работы на данных MNIST

- Определение переменных Tensorflow для весов и смещений, а также для входа x

```
x = tf.placeholder(tf.float32, [None, n_visible], name="x")
w = tf.Variable(tf.random_normal([n_visible, n_hidden], 0.01), name="w")
b_h = tf.Variable(tf.zeros([1, n_hidden]), tf.float32, name="b_h"))
b_v = tf.Variable(tf.zeros([1, n_visible]), tf.float32, name="b_v"))
```

- Конвертация вероятности в двоичные состояния 0 или 1

```
def sample(probs):
    return tf.floor(probs + tf.random_uniform(tf.shape(probs), 0, 1))
```

- Реализация одного шага семплирования Гиббса с сигмоидальной активационной ф-ей

```
def gibbs_step(x_k):
    h_k = sample(tf.sigmoid(tf.matmul(x_k, w) + b_h))
    x_k = sample(tf.sigmoid(tf.matmul(h_k, tf.transpose(w)) + b_v))
    return x_k
```

- Реализация k шагов семплирования Гиббса, начиная с заданной начальной точки

```
def gibbs_sample(k,x_k):
    for i in range(k):
        x_out = gibbs_step(x_k)
    # Returns the gibbs sample after k iterations
    return x_out
```

## 19.2 Ограниченные машины Больцмана. Пример реализации в Tensorflow и работы на данных MNIST

- Алгоритм CD-k контрастного расхождения
  1. На основе семплирования Гиббса получаем новое видимое состояние  $x$  из текущего видимого состояния
  2. На основе нового видимого состояния  $x$  сэмплируем новое скрытое состояние  $h$

```
x_s = gibbs_sample(2,x)
h_s = sample(tf.sigmoid(tf.matmul(x_s, w) + b_h))
```

- Сэмплируем скрытые состояния на основе видимых состояний

```
h = sample(tf.sigmoid(tf.matmul(x, w) + b_h))
# Sample visible states based given hidden states
x_ = sample(tf.sigmoid(tf.matmul(h, tf.transpose(w)) + b_v))
```

- Реализация корректировки весов на основе градиентного спуска

```
size_batch = tf.cast(tf.shape(x)[0], tf.float32)
w_add = tf.multiply(lr/size_batch, tf.subtract(tf.matmul(tf.transpose(x), h),
                                              tf.matmul(tf.transpose(x_s), h_s)))
bv_add = tf.multiply(lr/size_batch, tf.reduce_sum(tf.subtract(x, x_s), 0, True))
bh_add = tf.multiply(lr/size_batch, tf.reduce_sum(tf.subtract(h, h_s), 0, True))
updt = [w.assign_add(w_add), b_v.assign_add(bv_add), b_h.assign_add(bh_add)]
```

## 19.2 Ограниченные машины Больцмана. Пример реализации в Tensorflow и работы на данных MNIST

- Запуск выполнения графа вычислений Tensorflow

```
with tf.Session() as sess:  
    # Initialize the variables of the Model  
    init = tf.global_variables_initializer()  
    sess.run(init)
```

- Реализация обучения на основе градиентного спуска

```
total_batch = int(mnist.train.num_examples/batch_size)  
# Start the training  
for epoch in range(num_epochs):  
    # Loop over all batches  
    for i in range(total_batch):  
        batch_xs, batch_ys = mnist.train.next_batch(batch_size)  
        # Run the weight update  
        batch_xs = (batch_xs > 0)*1  
        _ = sess.run([updt], feed_dict={x:batch_xs})  
  
        # Display the running step  
        if epoch % display_step == 0:  
            print("Epoch:", '%04d' % (epoch+1))  
  
print("RBM training Completed !")
```

- Этап тестирования

```
out = sess.run(h,feed_dict={x:(mnist.test.images[:20]> 0)*1})  
label = mnist.test.labels[:20]
```

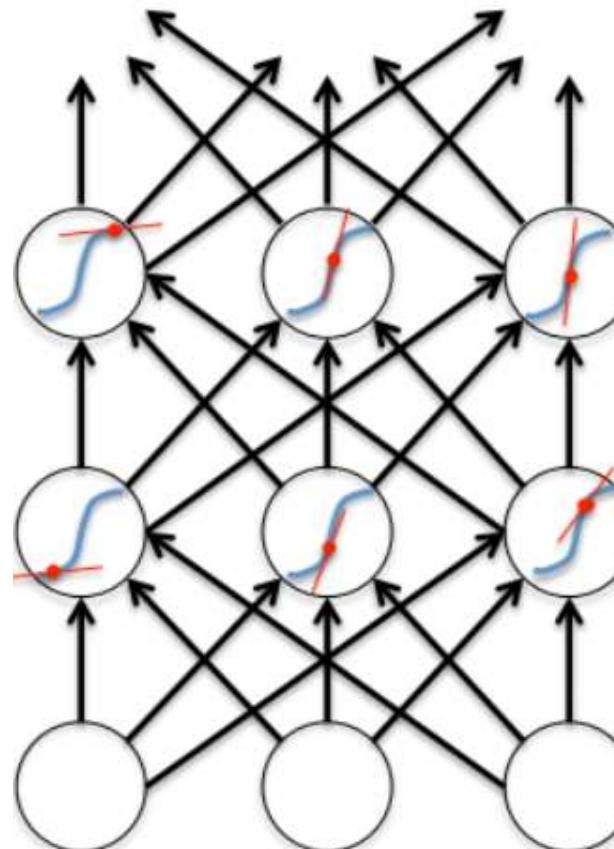
## 19.2 Ограниченные машины Больцмана. Пример реализации в Tensorflow и работы на данных MNIST

- Если добавить к построенной таким образом RBM полносвязный слой классификации с обучением на основе оптимизатора типа стохастического градиентного спуска, то в результате мы получим базовую нейронную сеть глубокого доверия, дающую для датасета MNIST следующую точность на этапах обучения и тестирования

```
Iter 0, Minibatch Loss= 7.632383, Training Accuracy= 0.21484
Iter 10, Minibatch Loss= 1.504026, Training Accuracy= 0.68750
Iter 20, Minibatch Loss= 1.055537, Training Accuracy= 0.78516
Iter 30, Minibatch Loss= 0.963917, Training Accuracy= 0.85156
Iter 40, Minibatch Loss= 0.554650, Training Accuracy= 0.86328
Iter 50, Minibatch Loss= 0.615745, Training Accuracy= 0.86719
Iter 60, Minibatch Loss= 0.466320, Training Accuracy= 0.91016
Iter 70, Minibatch Loss= 0.456979, Training Accuracy= 0.92969
Iter 80, Minibatch Loss= 0.381748, Training Accuracy= 0.89453
Iter 90, Minibatch Loss= 0.392191, Training Accuracy= 0.87891
Iter 100, Minibatch Loss= 0.318070, Training Accuracy= 0.93359
Iter 110, Minibatch Loss= 0.320216, Training Accuracy= 0.92188
Iter 120, Minibatch Loss= 0.351340, Training Accuracy= 0.91406
Iter 130, Minibatch Loss= 0.320061, Training Accuracy= 0.93359
Iter 140, Minibatch Loss= 0.380079, Training Accuracy= 0.92578
Iter 150, Minibatch Loss= 0.190657, Training Accuracy= 0.94531
Iter 160, Minibatch Loss= 0.206040, Training Accuracy= 0.93750
Iter 170, Minibatch Loss= 0.219785, Training Accuracy= 0.94141
Iter 180, Minibatch Loss= 0.213141, Training Accuracy= 0.94531
Iter 190, Minibatch Loss= 0.227372, Training Accuracy= 0.92578
Optimization Finished! Testing Accuracy: 0.933594
```

## 19.3 Предобучение с использованием ограниченной машины Больцмана. Проблема затухания и взрыва градиента. Эксперименты

- На каждом слое вычисляется линейная комбинация входного сигнала слоя и весов каждого нейрона, в результате получается скаляр в каждом нейроне, а затем вычисляется значение функции активации.
- Функция активации, помимо необходимой нам нелинейности, помогает ограничить выходные значения нейронов. Таким образом, на выходе получается вектор, каждая компонента которого находится в заранее известных пределах.
- На этапе обратного прохода градиенты на выходном слое вычисляются и получаются *существенные изменения* весов нейронов выходного слоя. Для любого скрытого слоя имеем



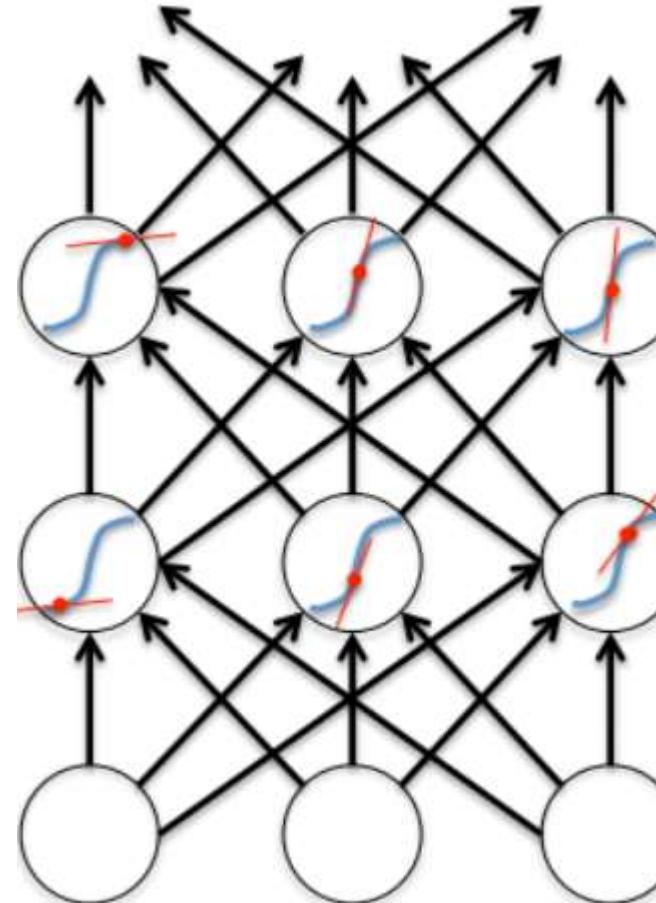
$$\frac{\partial E}{\partial w_{ij}^{(n)}} = x_i^{(n-1)} \sum_k \frac{\partial E}{\partial z_k^{n+1}} \frac{\partial z_k^{n+1}}{\partial y_j^n} \frac{\partial y_j^n}{\partial z_j^n} = x_i^{(n-1)} \sum_k w_{jk}^{n+1} \frac{\partial E}{\partial z_k^{n+1}} \frac{\partial y_j^n}{\partial z_j^n}$$

## 19.3 Предобучение с использованием ограниченной машины Больцмана. Проблема затухания и взрыва градиента. Эксперименты

- или если заменить выражение  $dE/dz$ , где участвует  $k$ , дельтой

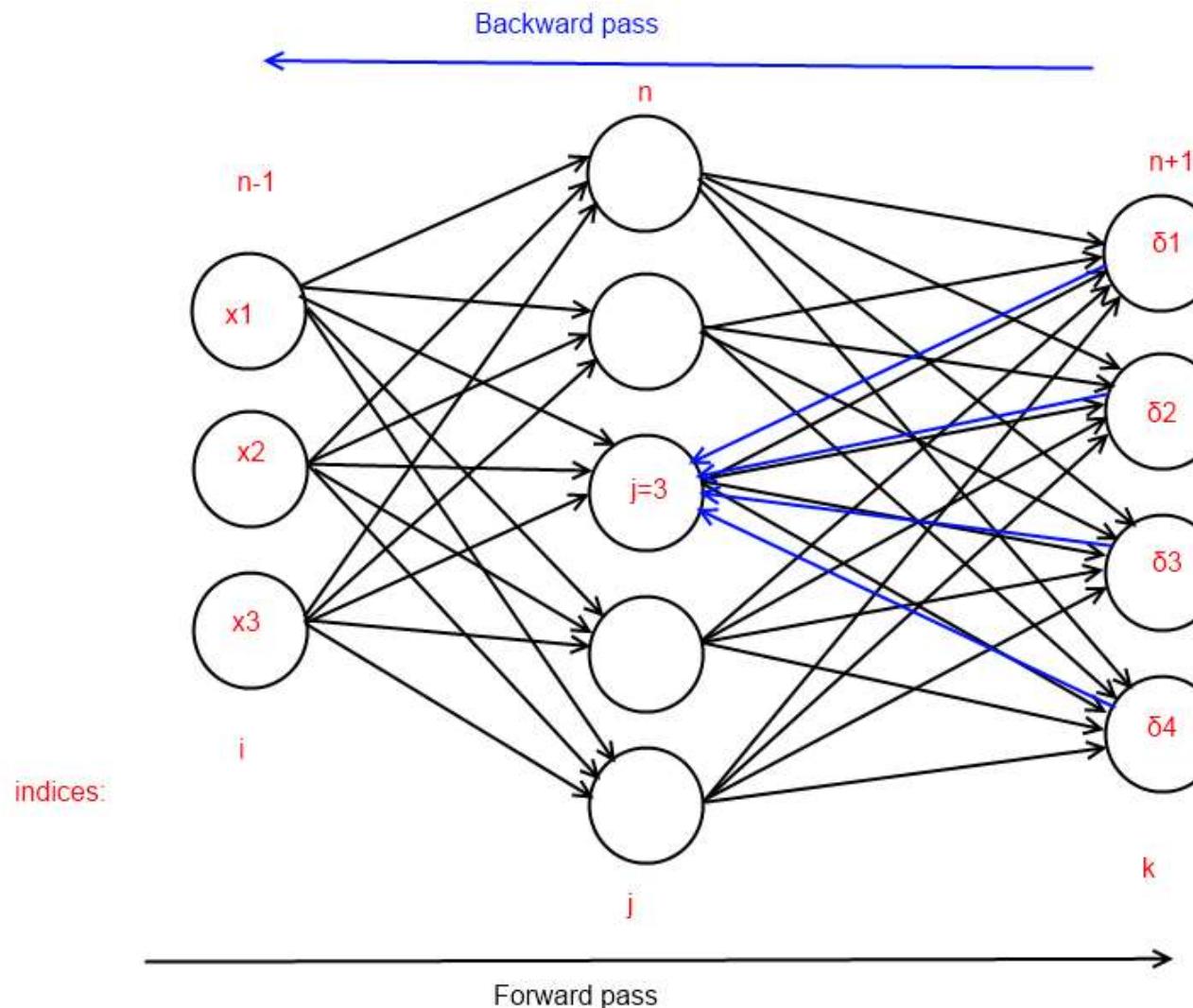
$$\frac{\partial E}{\partial w_{ij}^{(n)}} = x_i^{(n-1)} \sum_k w_{jk}^{(n+1)} \delta_k^{(n)} \frac{\partial y_j^{(n)}}{\partial z_j^{(n)}} = x_i^{(n-1)} \frac{\partial y_j^{(n)}}{\partial z_j^{(n)}} \sum_k w_{jk}^{(n+1)} \delta_k^{(n)}$$

мы увидим, что обратный проход очень похож на прямой, но он полностью линеен, со всеми вытекающими последствиями. Дельта называется локальным градиентом/ошибкой нейрона. При обратном проходе дельты можно интерпретировать как выходы (в другую сторону) соответствующих нейронов. Значение градиента веса нейрона более глубокого слоя пропорционально значению производной функции активации в точке, полученной на прямом проходе.



## 19.3 Предобучение с использованием ограниченной машины Больцмана. Проблема затухания и взрыва градиента. Эксперименты

- Обратный проход для одного нейрона скрытого слоя выглядит следующим образом:

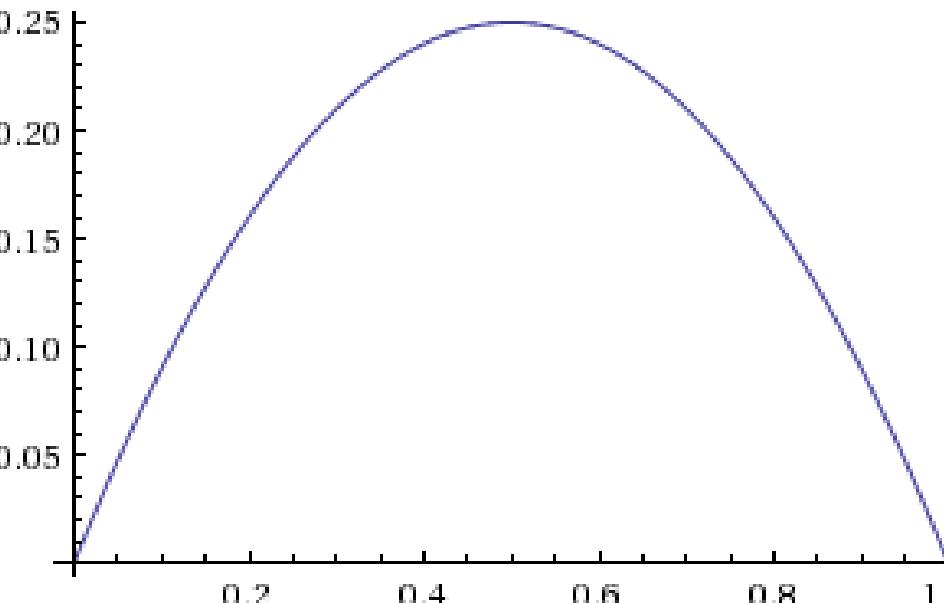


## 19.3 Предобучение с использованием ограниченной машины Больцмана. Проблема затухания и взрыва градиента. Эксперименты

- Имеем в итоге:
  - взрывной рост может случиться если веса слишком большие, или значение производной в точке слишком велико;
  - затухание происходит если значение весов или производной в точке очень мало.
- Как правило, *неглубокие сети* не страдают от этого из-за малого количества слоев, так как эффект не успевает аккумулироваться при обратном проходе (пример аккумуляции рассмотрим в следующем разделе).
- Если же в качестве функции активации используется сигмоид с производной следующего вида:

$$y' = f'(x) = \frac{\alpha e^{-\alpha x}}{(1 + e^{-\alpha x})^2} = \alpha f(x) (1 - f(x))$$

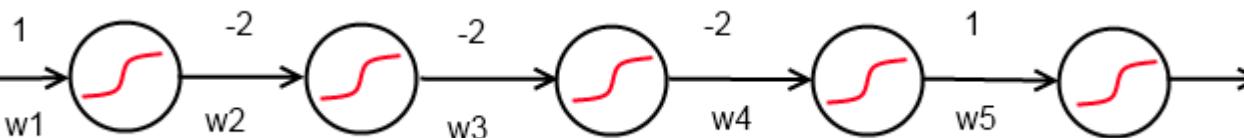
то при  $\alpha = 1$  мы легко можем убедиться, что максимальное значение, которого достигает производная, равно 0.25 в точке  $x = 0.5$ , а минимальное — 0 в точках  $x = 0$  и  $x = 1$  (на интервале от 0 до 1, области значений логистической функции):



## 19.3 Предобучение с использованием ограниченной машины Больцмана. Проблема затухания и взрыва градиента. Эксперименты

- Следовательно, для сетей с логистической функцией активации взрывной рост и затухание в большей степени зависит от значения весов, хотя существует перманентное затухание из-за того, что в лучшем случае значение тангенса угла наклона касательной будет всего 0.25, а эти значения, как мы увидим в следующем разделе, от слоя к слою перемножаются:
  - при очень малых значениях весов происходит затухание
  - при достаточно крупных весах — рост.

Пример:



Дана простая сеть, минимизирующая квадрат Евклидова расстояния, с логистической функцией активации, найдем градиенты весов при входном образе  $x = 1$  и целевом  $t = 0.5$ .

После прямого прохода получаем выходные значения каждого нейрона:

0.731058578630005, 0.188143725087062, 0.407022630083087, 0.307029145835665, 0.576159944190332. Обозначим их за  $y_1, \dots, y_5$ .

## 19.3 Предобучение с использованием ограниченной машины Больцмана. Проблема затухания и взрыва градиента. Эксперименты

- $\frac{\partial E}{\partial w_5} = y_4(y_5 - t)y_5(1 - y_5) = 0.00571019949926838$
- $\frac{\partial E}{\partial w_4} = y_3w_5y_4(1 - y_4)(y_5 - t)y_5(1 - y_5) = 0.0016105892898339$
- $\frac{\partial E}{\partial w_3} = y_2w_4y_3(1 - y_3)w_5y_4(1 - y_4)(y_5 - t)y_5(1 - y_5) = -0.00035937069569135$
- $\frac{\partial E}{\partial w_2} = y_1w_3y_2(1 - y_2)w_4y_3(1 - y_3)w_5y_4(1 - y_4)(y_5 - t)y_5(1 - y_5) = 0.000426583433503455$
- $\frac{\partial E}{\partial w_1} = xw_2y_1(1 - y_1)w_3y_2(1 - y_2)w_4y_3(1 - y_3)w_5y_4(1 - y_4)(y_5 - t)y_5(1 - y_5) = -0.000229451909878624$

□ Как видим, происходит затухание градиента. По ходу вычисления заметно, что аккумуляция происходит за счет перемножения весов и значений производных текущего слоя и всех последующих. Хотя начальные значения далеко не маленькие, по сравнению с тем как веса инициализируются на практике, но и этих значений недостаточно, чтобы подавить перманентное затухание от тангенсов угла наклона касательной.

## 19.3 Предобучение с использованием ограниченной машины Больцмана. Проблема затухания и взрыва градиента. Эксперименты

- Исходные данные для экспериментов
- Обучающее множество состоит из изображений больших печатных букв английского алфавита трех шрифтов (Arial, Courier и Times) как представители шрифтов разной породы) размером 29 на 29 пикселей, и их же изображений с некоторыми случайными шумами
- Кросс-валидационное множество состоит из того же набора шрифтов, плюс Tahoma и чуть больше шумов
- Тестовое множество состоит из того же набора данных, что и кросс-валидационное, отличается лишь фактором случайности при генерации шумов
- Во всех опытах обучение будет останавливаться если увеличение ошибки кросс-валидации больше, чем некоторый процент от минимальной ошибки, то происходит остановка (как обучение сетей прямого распространения, так и машин Больцмана).
- В сетях прямого распространения выходным слоем будет softmax слой, а вся сеть обучается на минимизации перекрестной энтропии, соответственно, значение ошибок вычисляется тем же способом. А ограниченные машины Больцмана в качестве меры ошибки будут использовать расстояние Хэмминга между бинарным входным образом и бинарным восстановленным образом.

## 19.3 Предобучение с использованием ограниченной машины Больцмана. Проблема затухания и взрыва градиента. Эксперименты

### □ Эксперименты

- Посмотрим, какой результат даст нам сеть с одним скрытым слоем из 100 нейронов, т.е. структура ее будет  $100 \rightarrow 26$ . Каждый процесс обучения проиллюстрируем графиком с ошибкой обучения и кросс-валидации.

Параметры обучения:

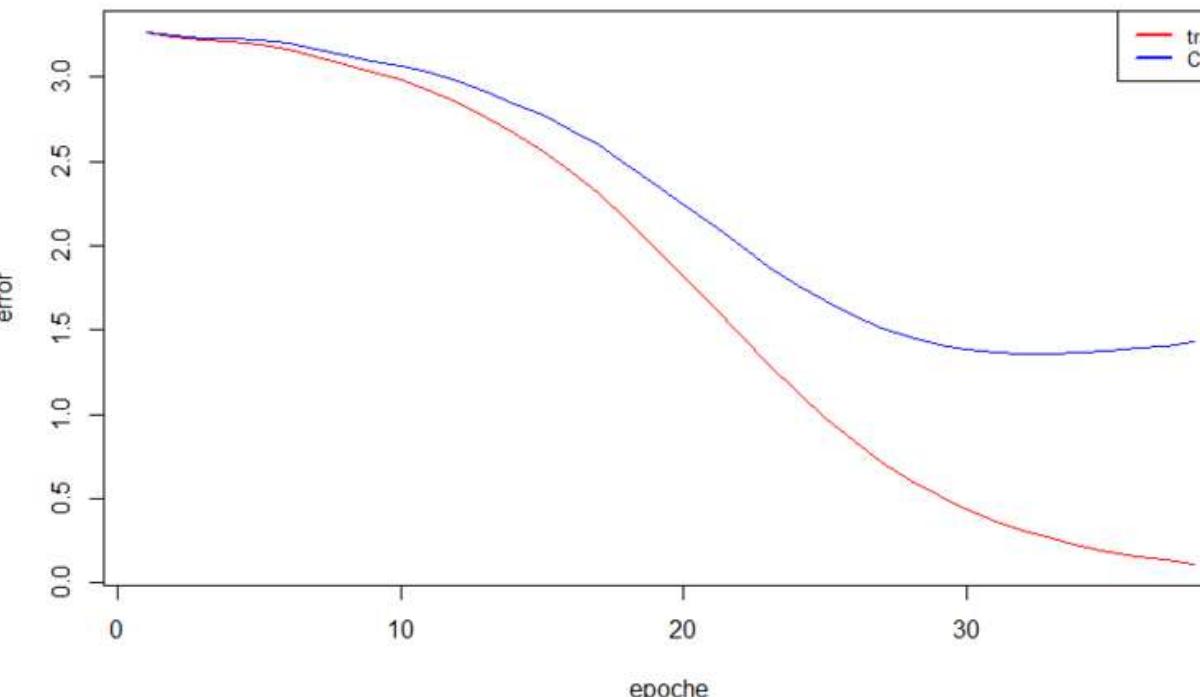
LearningRate = 0.001 BatchSize = full batch

RegularizationFactor = 0 Momentum = 0.9

NeuronLocalGainLimit: Limit = [0.1, 10]; Bonus = 0.05; Penalty = 0.95

CrossValidationStopFactor is 0.05

Результат на  
тестовом  
множестве,  
обучение заняло  
около 35 секунд:  
Error:  
1.39354424678178  
Wins: 231 of 312,  
74.03%



## 19.3 Предобучение с использованием ограниченной машины Больцмана. Проблема затухания и взрыва градиента. Эксперименты

- Рассмотрим предобученную сеть
- Предобучать мы будем сеть из 5 слоев следующей структуры: 100 -> 300 -> 100 -> 50 -> 26. Но для начала давайте посмотрим, как выглядит процесс обучения и результат, если обучить такую сеть без предобучения.
- Параметры обучения:

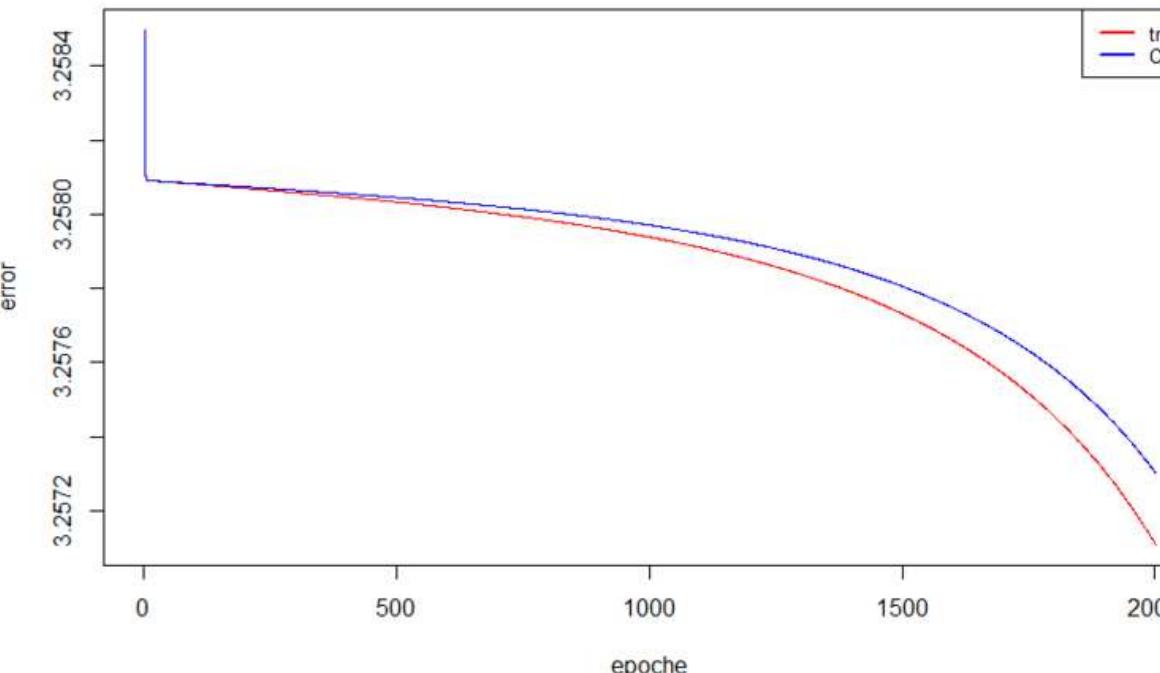
LearningRate = 0.001 BatchSize = full batch

RegularizationFactor = 0.1 MaxEpoches = 2000

MinError = 0.1 Momentum = 0

NeuronLocalGainLimit: not setted CrossValidationStopFactor is 0.001

Результаты на тестовом множестве ничтожны, около 3 попаданий из 312. Обучение заняло 4300 секунд, это почти час двадцать (обучение было прервано на 2000 эпохе, хотя могло бы и продолжаться дальше). За 2000 эпох было достигнуто значение ошибки 3.25710954400557, в то время как в предыдущем teste уже на второй эпохе было 3.2363407570674, а на последней 0.114535596947578.



## 19.3 Предобучение с использованием ограниченной машины Больцмана. Проблема затухания и взрыва градиента. Эксперименты

- Теперь давайте обучим RBM'ы для каждой последовательной пары. Для начала это будет 841  $\leftrightarrow$  100 со следующими параметрами:

LearningRate = 0.001 BatchSize = 10 Momentum = 0.9

NeuronLocalGainLimit: Limit = [0.001, 1000]; Bonus = 0.0005; Penalty = 0.9995

GibbsSamplingChainLength = 35 UseBiases = True

CrossValidationStopFactor is 0.5 MinErrorChange = 0.0000001

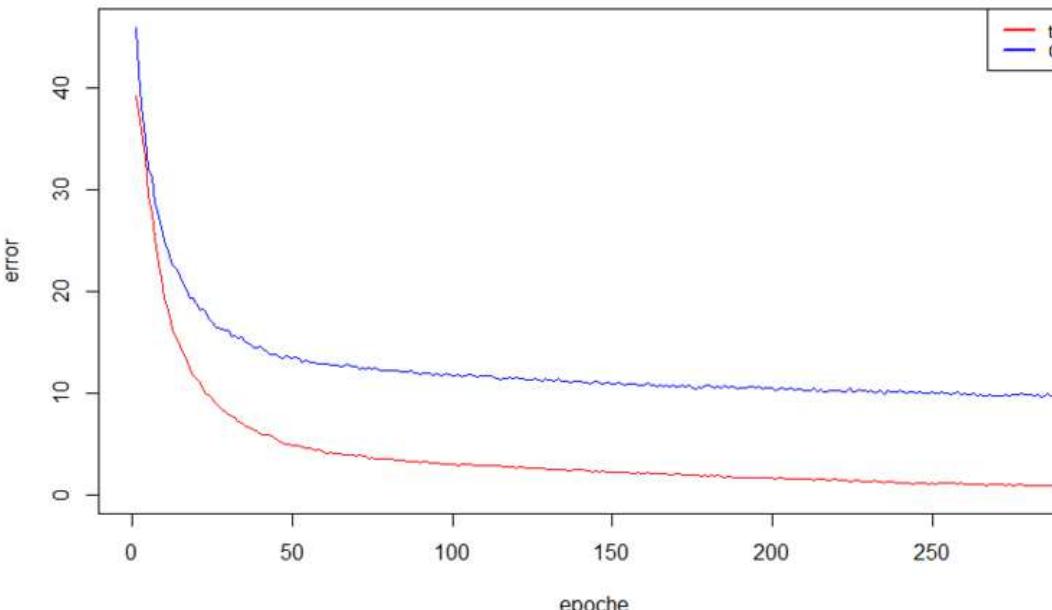
Обучение заняло 1274 секунды.

- Затем обучаем вторую пару 100  $\leftrightarrow$  300, заняло 2095 секунд:

LearningRate = 0.001 BatchSize = 10 Momentum = 0.9

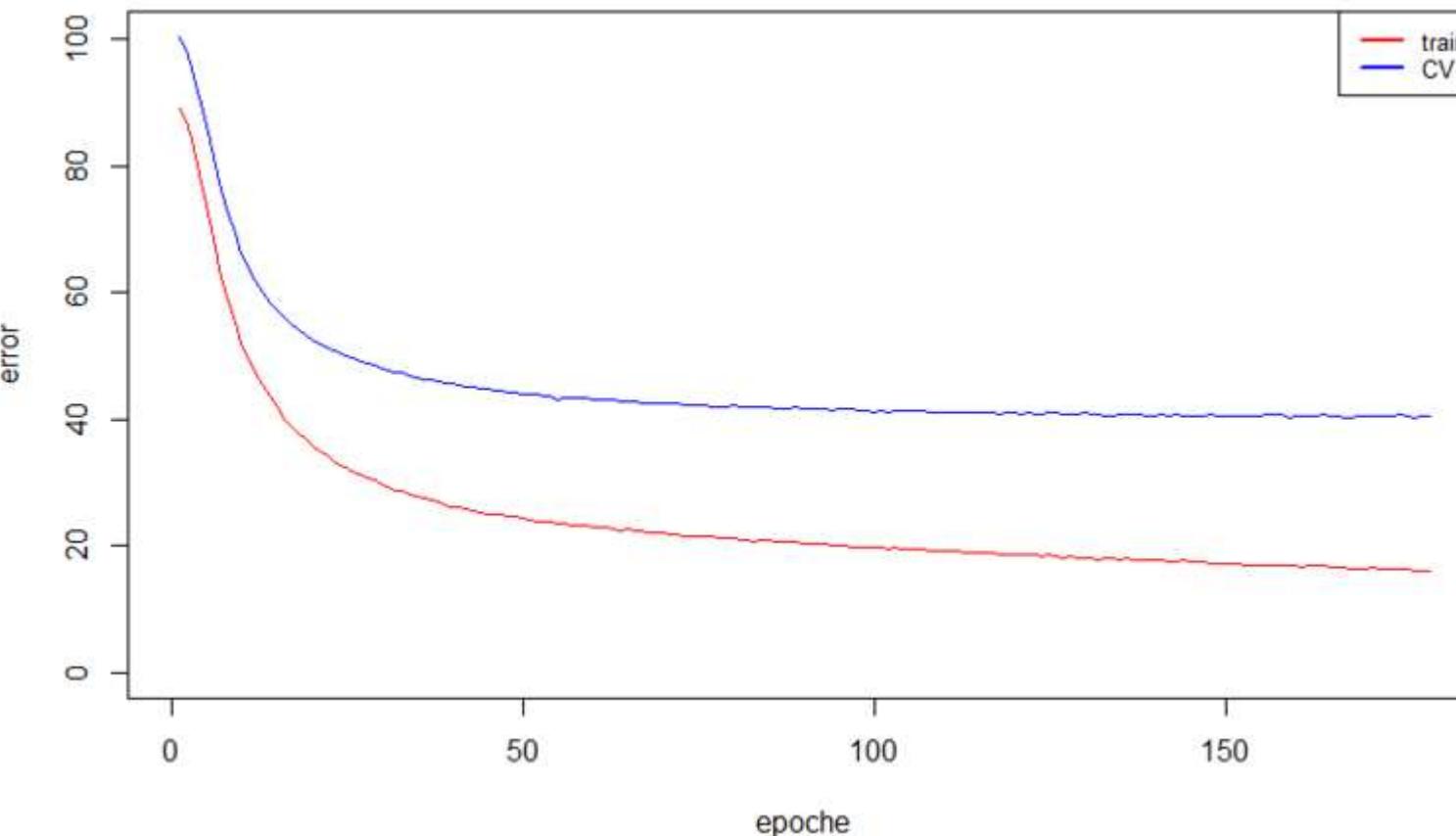
NeuronLocalGainLimit: Limit = [0.001, 1000]; Bonus = 0.0005; Penalty = 0.9995

GibbsSamplingChainLength = 35 UseBiases = True CrossValidationStopFactor is 0.1



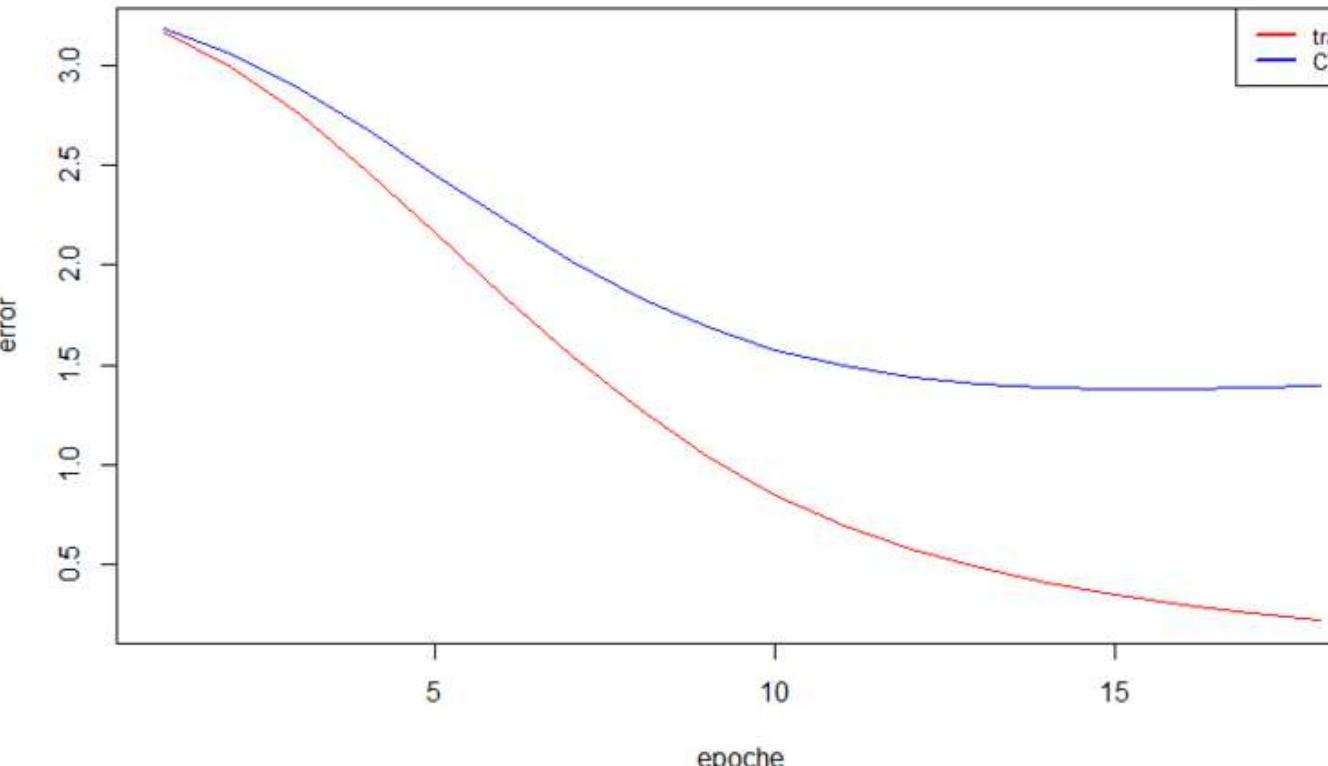
## 19.3 Предобучение с использованием ограниченной машины Больцмана. Проблема затухания и взрыва градиента. Эксперименты

- Обучаем третью пару 300 <-> 100, заняло 1300 секунд:  
LearningRate = 0.001 BatchSize = 10 Momentum = 0.9  
NeuronLocalGainLimit: Limit = [0.001, 1000]; Bonus = 0.0005; Penalty = 0.9995  
GibbsSamplingChainLength = 35 UseBiases = True CrossValidationStopFactor is 0.1



## 19.3 Предобучение с использованием ограниченной машины Больцмана. Проблема затухания и взрыва градиента. Эксперименты

- Теперь складываем все слои в одну сеть прямого распространения и обучаем со следующими параметрами:  
LearningRate = 0.001 BatchSize = full batch RegularizationFactor = 0  
Momentum = 0.9 NeuronLocalGainLimit: Limit = [0.01, 100]; Bonus = 0.005; Penalty = 0.995  
CrossValidationStopFactor is 0.01
- Результат на тестовом множестве, при том что обучение заняло всего 17 эпох или 36 секунд: Error: 1.40156176438071, Wins: 267 of 312, 85.57%  
Прирост качества составил почти 11% по сравнению с сетью с одним скрытым слоем.



## 19.3 Предобучение с использованием ограниченной машины Больцмана. Проблема затухания и взрыва градиента. Эксперименты

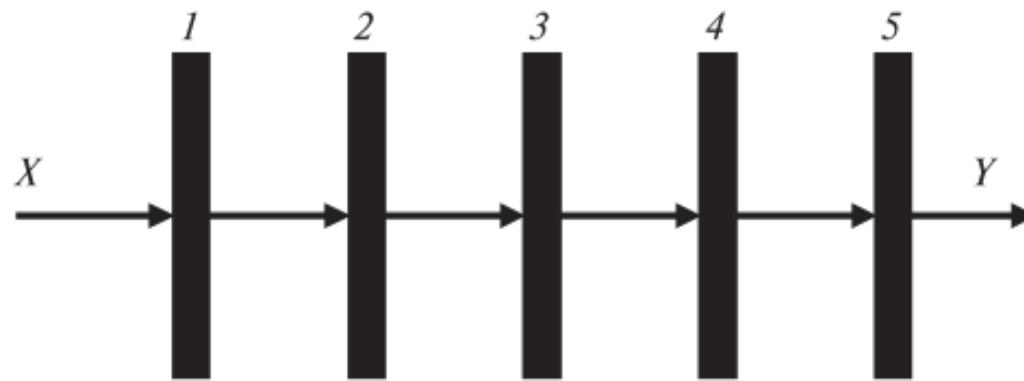
### Заключение

Предобучение - достаточно эффективный способ.

Предположим, есть задача распознать капчи. У нас скачано 100 000 картинок, из них распознано вручную всего 10 000. Мы бы хотели обучить нейросеть, но нам жалко не использовать оставшиеся 90 000 тысяч картинок. Тут нам на помощь приходят RBM. Используя массив из 100 000 картинок, мы обучаем deep belief network, а затем на множестве из 10 000 распознанных вручную картинок мы обучаем глубокую сеть.

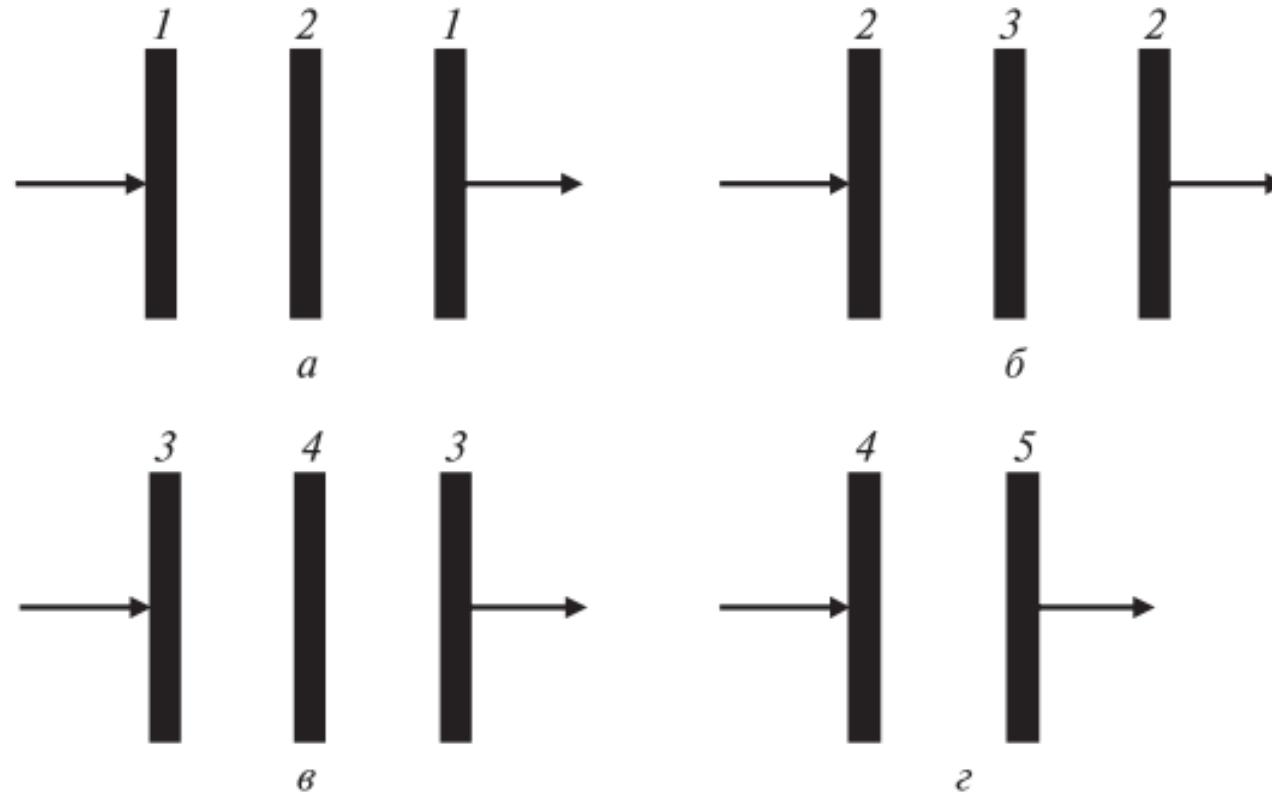
## 19.4 Предобучение с использованием автокодировщиков. 19.4.1 Послойное предобучение

- Данный подход базируется на представлении каждого слоя в виде *автоассоциативной нейронной сети*. В этом случае вначале обучается первый слой как автоассоциативная нейронная сеть в целях минимизации суммарной квадратичной ошибки реконструкции информации, затем второй и т. д. Для обучения каждого слоя можно применять алгоритм обратного распространения ошибки. После этого осуществляется точная настройка синаптических связей всей сети (*fine tuning*) с использованием алгоритма обратного распространения ошибки.
- Рассмотрим персепtron с тремя скрытыми слоями. Тогда в соответствии с автоэнкодерным методом прежде всего берутся первые два слоя нейронной сети (1 и 2) и на их основе конструируется автоассоциативная (PCA-сеть) нейронная сеть (1 – 2 – 1), т. е. добавляется восстановливающий слой *a*), а затем происходит обучение, например при помощи алгоритма обратного распространения ошибки такой сети в целях минимизации квадратичной ошибки реконструкции информации. Продолжительность обучения обычно составляет не больше 100 эпох.



## 19.4 Предобучение с использованием автокодировщиков. 19.4.1 Послойное предобучение

- После этого отбрасывается восстанавливающий слой (последний), фиксируются веса скрытого слоя и конструируется автоассоциативная сеть из следующих двух слоев нейронной сети 2 – 3 – 2 (рис. б), которая обучается на основе данных поступающих с предыдущего (2-го слоя). Процесс продолжается до последнего (рис. г) или предпоследнего (рис. в) слоя. В результате послойного обучения получается предварительно обученная нейронная сеть. Далее осуществляется точная настройка (fine tuning) посредством, например, алгоритма обратного распространения ошибки с учителем.



## 19.4 Предобучение с использованием автокодировщиков. 19.4.1 Послойное предобучение

- Данный процесс можно представить в виде следующего алгоритма:
  1. Конструируется автоассоциативная сеть с входным слоем  $X$ , скрытым  $Y$  и выходным слоем  $X$ .
  2. Обучается автоассоциативная сеть, например при помощи алгоритма обратного распространения ошибки (как правило, не более 100 эпох), и фиксируются синаптические связи первого слоя  $W_1$ .
  3. Берется следующий слой, и аналогичным образом формируется автоассоциативная сеть.
  4. С использованием настроенных синаптических связей предыдущего слоя  $W_1$  входные данные подаются на вторую автоассоциативную сеть, и она обучается аналогичным образом. В результате получаются весовые коэффициенты второго слоя  $W_2$ .
  5. Процесс продолжается до последнего слоя нейронной сети.
  6. Обучается вся сеть для точной настройки параметров при помощи алгоритма обратного распространения ошибки.

## 19.4 Предобучение с использованием автокодировщиков. 19.4.2 Предобучение на частично размеченных данных

- В случае если вы имеете крупный набор данных, но большинство из них не помечено, тогда можете сначала обучить многослойный автокодировщик с применением всех данных, затем повторно использовать самые нижние слои для создания нейронной сети, ориентированной на фактическую задачу, и обучить ее с применением помеченных данных.
- Многослойный автокодировщик обычно обучает по одному автокодировщику за раз. Замораживаем нижние заранее обученные слои.

