Искусственные нейронные сети: основы практического применения

Лекция 4

Крощенко А.А.

Брестский государственный технический университет

31.05.2017

Задача кластеризации. Постановка

Задача кластеризации – разбиение множества образов обучающей выборки на непересекающиеся подмножества таким образом, чтобы каждый кластер состоял из схожих объектов.
Цели, преследуемые выполнением кластеризации:

- Выявление скрытых закономерностей в данных
- Сжатие данных
- Выявление аномалий

Формально: элементы обучающей выборки X разбить на непересекающиеся подмножества $X_i, i = 1, ..M$ элементов близких по некоторой метрике ρ.
Кластеризация выполняется как правило без учителя.
Основная проблема кластеризации – выбор количества кластеров.

Алгоритмы, применяемые для решения задачи кластеризации

- Метод k-средних
- Нейронная сеть Кохонена

Выборка, используемая при проведении экспериментов

Для сравнения алгоритмов кластеризации использовалась выборка Seeds Dataset, содержащая информацию о 3-х классах семян пшеницы. Размер выборки – 210 образов, количество компонент – 7.

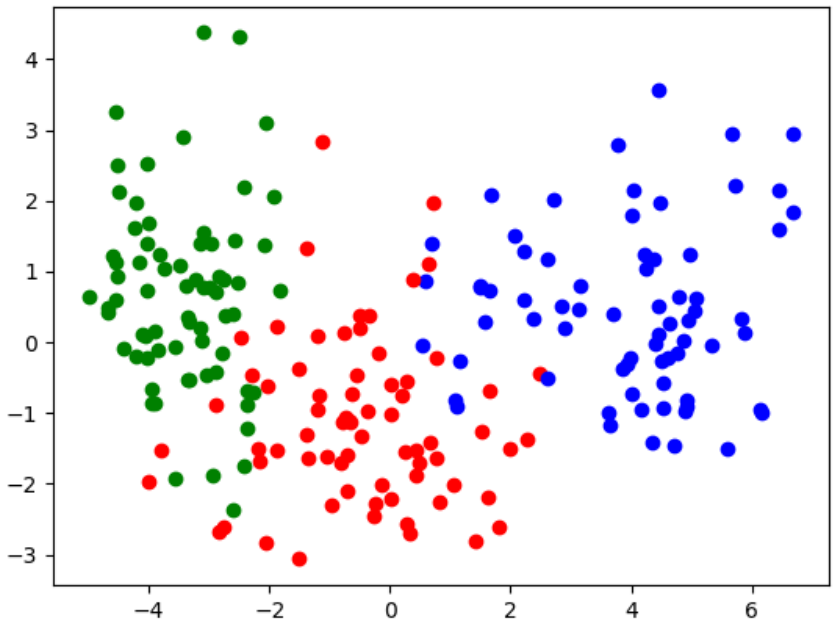


Figure 1: Визуализация, полученная PCA

Потери информации составили **0.6982 %**.

Метод k-средних

Вход: X – неразмеченные данные
Результат: Набор центроидов $c_i \in C$ – точек, формирующих центры соответствующих кластеров
Выбрать начальные k точек, которые должны (желательно) принадлежать различным кластерам
Сделать эти точки центроидами кластеров $c_i \in C$
foreach $x_i \in X \setminus C$ **do**
    Найти ближайший к x_i центроид
    Включить x_i в кластер с этим центроидом
    Пересчитать центроид кластера, в который включена точка x_i
end

Выбор начальных точек-центроидов

- Выбрать точки, отстоящие друг от друга как можно дальше

  Вход: X – неразмеченные данные

  Результат: Начальный набор центроидов для алгоритма k-means ($c_i \in C$)

  Выбрать первую точку-центроид случайно

  while *точек меньше k* **do**

  | Добавить точку, для которой минимум расстояний до каждой из выбранных точек достигает максимума

  end

Решение

```
import numpy as np
import random as rnd

class KMeans:
  def __init__(self, data, count_clusters):
    count_samples = len(data)
    self.count_clusters = count_clusters
    self.data = data
    self.clustering = np.zeros(count_samples).astype('int')
    self._init_clustering()

  def _init_clustering(self):
    count_samples = len(self.data)
    for i in range(0, count_samples):
      if i < self.count_clusters:
        self.clustering[i] = i
      else:
        self.clustering[i] = int(self.count_clusters* rnd.random())
```

Решение

```
def run(self):
  iterations = 0
  while True:
    means = self._update_means()
    centroids = self._compute_centroids(means)
    iterations += 1
    if not self._assign_samples_to_clusters(centroids):
      break
  return self.clustering
```

Решение

```
def _update_means(self):
    count_components = len(self.data[0])
    count_samples_in_clusters = np.zeros(self.count_clusters)
    means = np.zeros((self.count_clusters, count_components))
    i = 0
    for elem in self.clustering:
        count_samples_in_clusters[elem] += 1
        means[elem] += self.data[i]
        i += 1
    for i in range(0, self.count_clusters):
        means[i] /= count_samples_in_clusters[i]
    return means
```

Решение

```
def _compute_centroids(self, means):
    count_samples = len(self.data)
    count_components = len(self.data[0])
    centroids = np.zeros((self.count_clusters, count_components))
    distance = np.zeros(self.count_clusters)
    centroids_index = np.zeros(self.count_clusters).astype('int')
    for i in range(0, self.count_clusters):
        distance[i] = float('inf')
        centroids_index[i] = -1
    for i in range(0, count_samples):
        num = self.clustering[i]
        current_distance = np.linalg.norm(self.data[i] - means[num])
        if current_distance < distance[num]:
            distance[num] = current_distance
            centroids_index[num] = i
    for i in range(0, self.count_clusters):
        centroids[i] = self.data[centroids_index[i]]
    return centroids
```

Решение

```
def _assign_samples_to_clusters(self, centroids):
    change = False
    i = 0
    for sample in self.data:
        distances = np.linalg.norm(sample - centroids, axis=1)
        num = distances.argmin()
        if num != self.clustering[i]:
            self.clustering[i] = num
            change = True
        i += 1
    return change
```

Результаты, полученные непосредственным применением метода к PCA-данным

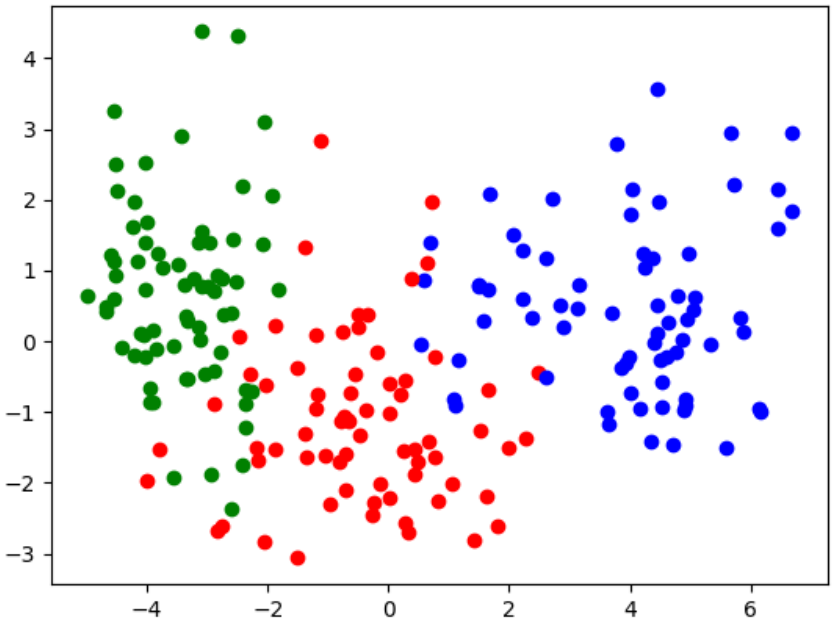


Figure 2: PCA-визуализация

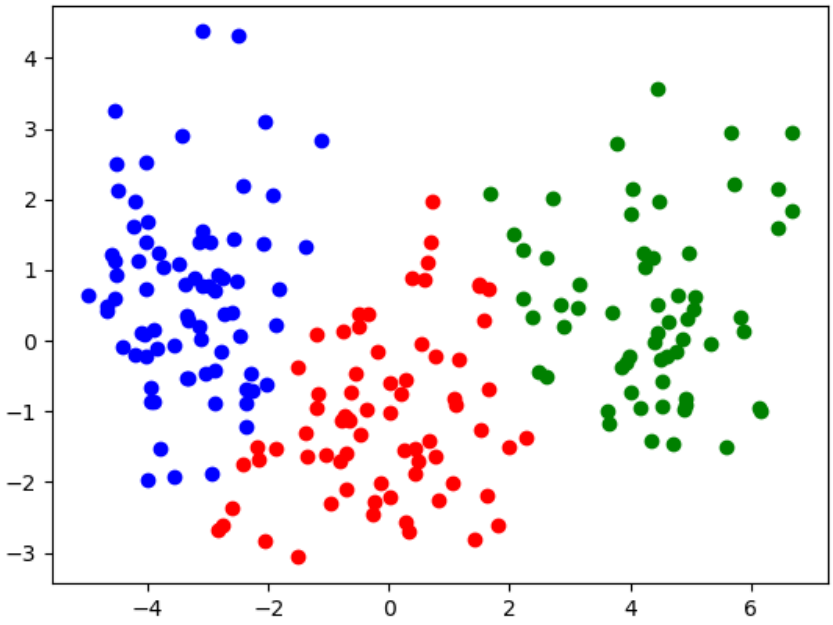


Figure 3: Результат работы метода k-средних

Результаты, полученные применением кластеризации до PCA

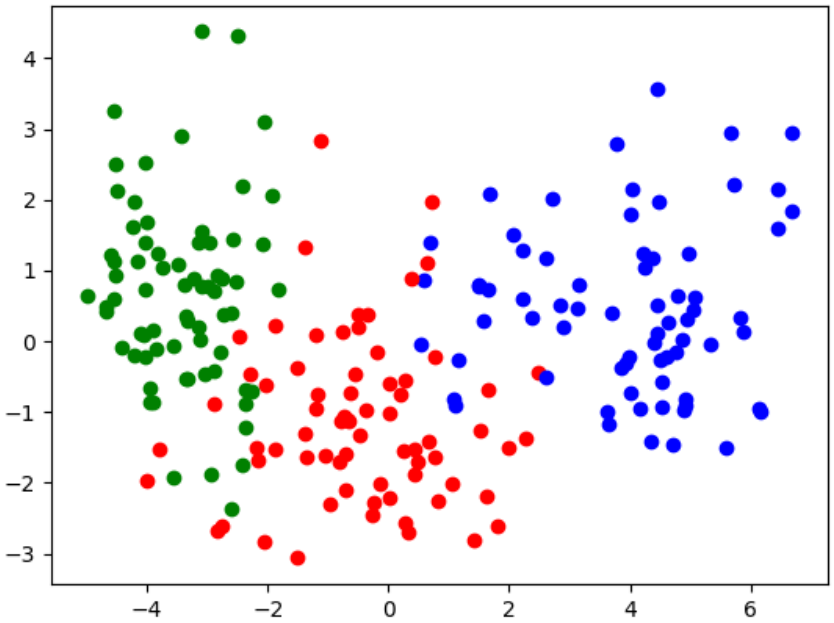


Figure 4: PCA-визуализация

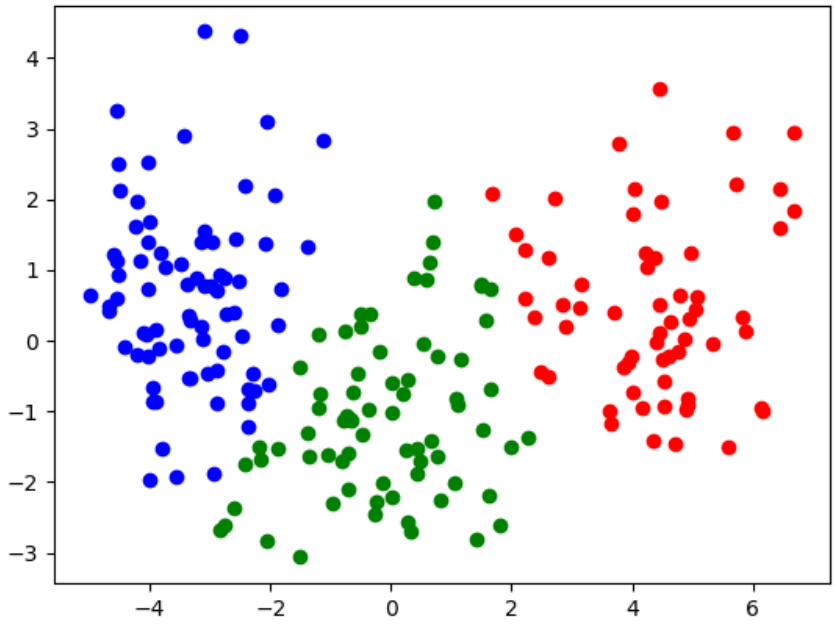


Figure 5: Результат работы метода k-средних

Достоинства и недостатки

Достоинства:

- Метод k-средних прост в программировании и применении
- Быстрый на небольших выборках

Недостатки:

- Начальная инициализация может существенно влиять на итоговый результат
- Может быть медленным на больших выборках данных
- Вообще говоря, может терять стабильность на данных большой размерности

Нейронная сеть Кохонена

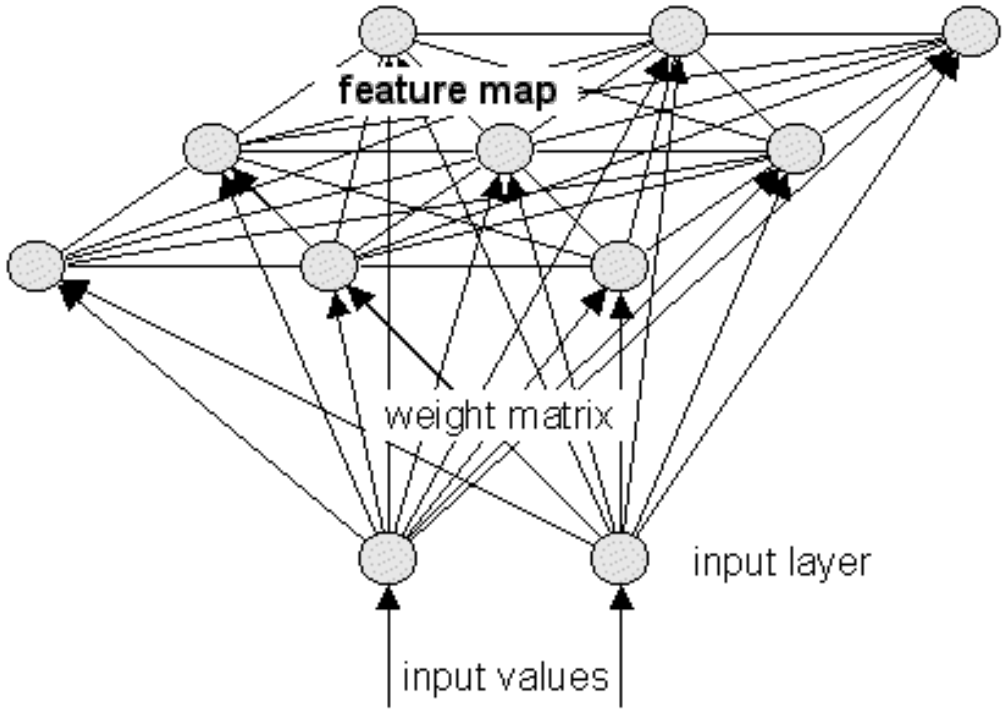


Figure 6: Нейронная сеть Кохонена

http://www.nnwj.de/kohonen-feature-map.html

Алгоритм обучения НС Кохонена

- **Инициализация**. Для исходных векторов синаптических весов $w_j(0)$ выбираются случайные значения. единственным требованием является различие векторов для разных значений $j = 1, 2, ..., l$, где l – общее количество нейронов в решетке.
- **Подвыборка**. Выбираем вектор x из входного пространства с определенной вероятностью.
- **Поиск максимального подобия**. Находим наиболее подходящий (победивший) нейрон $i(x)$ на шаге n, используя критерий минимума Евклидова расстояния:

$$i(x) = \arg\min_j ||x - w_j||, j = 1, 2, ..., l$$

- **Коррекция**.

$$w_j(n+1) = w_j(n) + \eta(n) h_{j,i(x)}(n)(x - w_j(n))$$

где $\eta(n)$ – параметр скорости обучения; $h_{j,i(x)}$ – функция окрестности с центром в победившем нейроне $i(x)$.

- **Продолжение**. Возвращаемся к шагу 2 и продолжаем вычисления до тех пор, пока в карте признаков не перестанут происходить заметные изменения.

Решение

```
class KohonenMap:

  def __init__(self, neurons_count,dimension, rate0, sigma0, tau2):
    self.weights = np.random.random((neurons_count, dimension))
    self.tau2 = tau2
    self.rate0 = rate0
    self.rate = rate0
    self.sigma0 = sigma0
    self.sigma = sigma0
    self.neurons_count = neurons_count
```

Решение

```
def subtrain(self, data):
  samples_count = len(data)
  while True:
    index = rnd.randint(0, samples_count-1)
    sample = data[index]
    win_neuron_index = self._define_win_neuron(sample)
    top_loc = self._top_loc(win_neuron_index, range(0, self.
        neurons_count))
    for i in range(0, self.neurons_count):
      self.weights[i] += self.rate * top_loc[i] * (sample - self.
          weights[i])
    self._change_rate(iterations)
    self._change_sigma(iterations)
    iterations += 1
```

Решение

```
def train(self, data):
  iterations = 0
  while True:
    subtrain(data)
    if not iterations < 1000:
      break
  self.rate = 0.01
  while True:
    subtrain(data)
    if not iterations < 1500:
      break
```

Решение

```
def print_clusters(self, data):
  for sample in data:
    index = self._define_win_neuron(sample)
    print str(index) + ': (' + str(sample[0]) + ', ' + str(sample
        [1]) + ')'

def _define_win_neuron(self, sample):
  win_neuron_index = np.linalg.norm(self.weights - sample).argmin()
  return win_neuron_index

def _top_loc(self, i, j):
  distance = np.abs(i - j)
  return np.exp(-distance/(2*self.sigma**2))

def _change_sigma(self, n):
  tau1 = 1000.0 / math.log(self.sigma0)
  self.sigma = self.sigma0 * math.exp(-n/tau1)

def _change_rate(self, n):
  self.rate = self.rate0 * math.exp(-n/self.tau2)
```

Автоассоциативная нейронная сеть

Автоассоциативная сеть – это многослойная нейронная сеть прямого распространения сигнала, обученная для того, чтобы выдавать входные данные на выходе сети. При обучении автоассоциативной сети «учителем» для нее является сама входная информация. На первой половине сети происходит так называемое «прямое распространение», т.е. осуществляется сжатие входных данных. Далее на второй половине сети, которая является зеркальным отражение первой, происходит «обратное распространение», т.е. восстановление входного образа. Обычно сеть имеет скрытый слой меньшей размерности, который выделяет наиболее значимые признаки во входной информации. Пример такой сети представлен на рисунке 7.
Автоассоциативные сети с таким слоем оказываются полезны при решении задач визуализации и обработки данных высокой размерности, так как позволяют существенно сократить объем данных.

Пример автоассоциативной НС

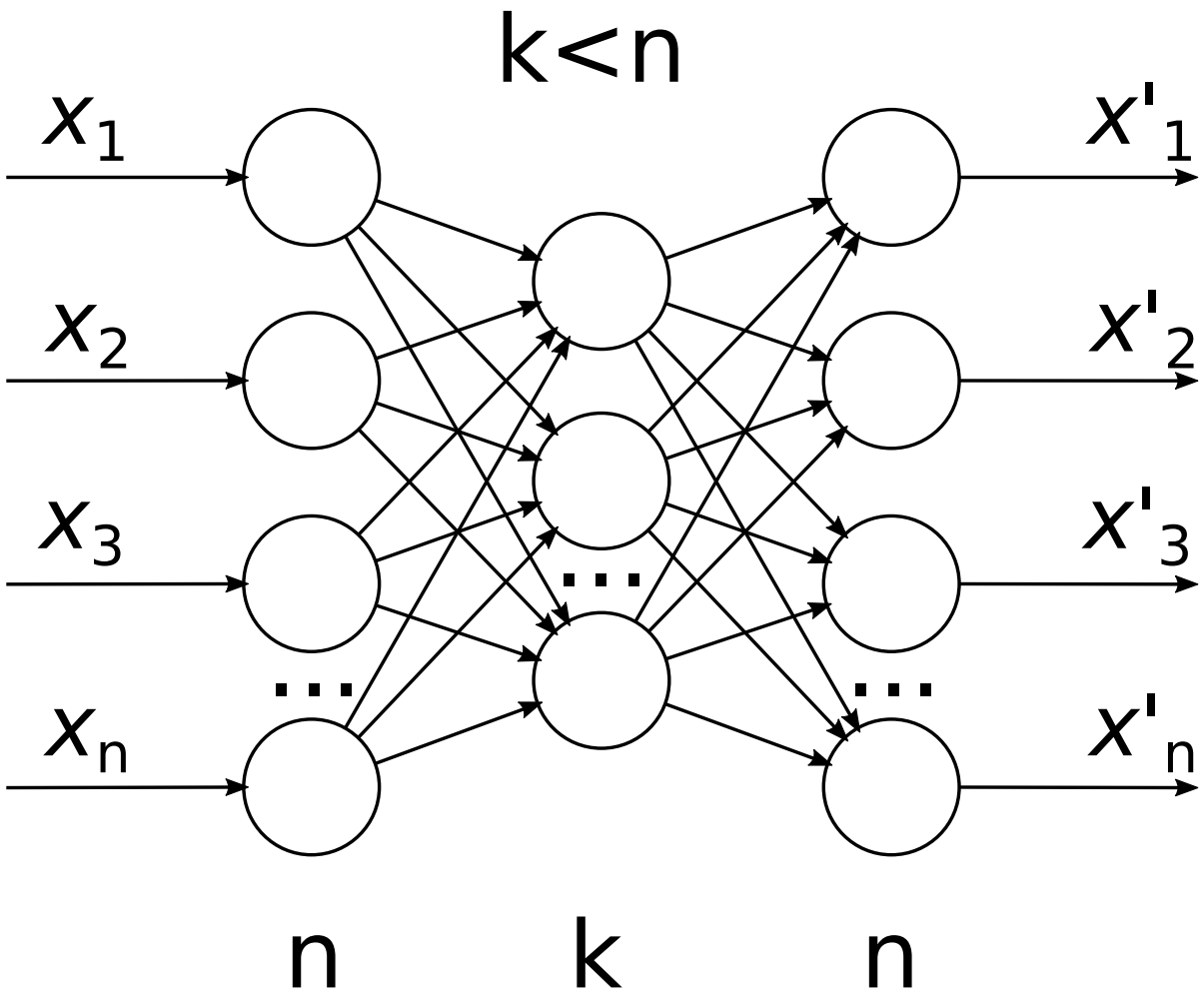


Figure 7: Автоассоциативная сеть

Области применения

- Сжатие информации
- Получение хешированного представления входной информации для удобства сравнения нескольких образов (поисковая система)
- Уменьшение размерности входных данных (NPCA) с возможностью обучения классификатора на новом представлении

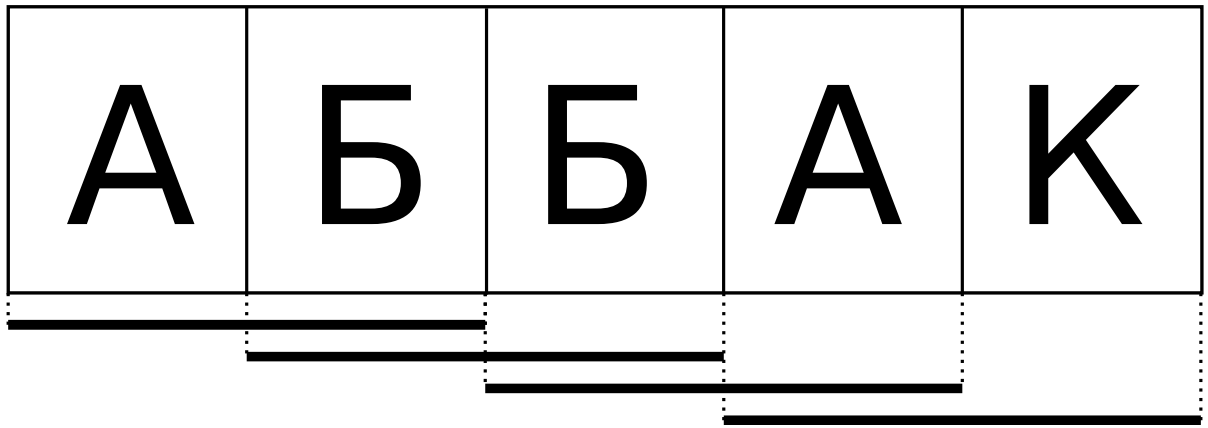
Задача

Имеется **орфографический словарь** (исп. прямой словарь А. А. Зализняка с 93392 словами). Необходимо обучить **автоэнкодер** сжатию входной информации, которая представляет собой некое **промежуточное представление** слов из словаря, чтобы затем использовать получившийся автоэнкодер для поиска **схожих слов**.
Иначе говоря, с помощью **автоэнкодера** сформировать представление слов в виде **векторов фиксированной (и меньшей, чем входная) размерности** и использовать получившуюся модель для формирования пула схожих слов.

В каком виде слова будут подаваться на нейронную сеть?

Слова на нейронную сеть будут подаваться в виде векторов, содержащих частоту встречаемости т.н. **n-грамм**.
N-грамма – это последовательность из n элементов. N-граммы из двух элементов называются биграммами.
Для решения этой задачи использовались биграммы. Рассмотрим пример формирования биграмм.

Пример формирования биграмм



АБ - 1
ББ - 1
БА - 1
АК - 1

Решение: загрузка словаря

```
def load_dictionary(path):
  input_file = open(path, 'rb')
  _dict = []
  for _str in input_file:
    _str = _str.rstrip('\n').decode('utf-8')
    if not ('-' in _str) and len(_str) > 1:
      _dict.append(_str)
  return _dict
```

Решение: формирование обучающей выборки

```
def get_ngrams():
  alphabet = "abcdefghijklmnopqrstuvwxyz"
  ngrams = []
  for i in range(0, len(alphabet)):
    for j in range(0, len(alphabet)):
      ngrams.append(alphabet[i] + alphabet[j])
  return ngrams

def get_ngrams_data_presentation(data):
  ngrams = get_ngrams()
  ngram_data = np.zeros((len(data), len(ngrams)))
  for i in xrange(0, len(data)):
    for j in xrange(0, len(ngrams)):
      if ngrams[j] in data[i]:
        ngram_data[i, j] += 1
  return ngram_data
```

Решение: основная программа

```
if __name__ == "__main__":
    _dict = load_dictionary('Datasets/dictionary.txt')
    data = get_ngrams_data_presentation(_dict)
    net = Network()
    layer_1 = FullyConnectedLayer(Logistic(), 1089, 100)
    layer_2 = FullyConnectedLayer(Linear(), 100, 1089)
    net.append_layer(layer_1)
    net.append_layer(layer_2)
    params = Backprop_params(200, 1e-5, 1000, 0.9, False, [0.01,
        0.01], 0)
    method = Backpropagation(params, net)

    rnd_index = np.random.permutation(len(data))
    data = data[rnd_index]

    method.train(data, data)

    Network.save_network(net, 'nets/network.net')
```

Решение: тестирование автоэнкодера

```
if __name__ == "__main__":
    _dict = load_dictionary('Datasets/dictionary.txt')

    words, words_ngram = get_random_words(_dict, 10)
    data = get_ngrams_data_presentation(_dict)
    net = Network.load_network('nets/network.net')
    output_all_data = net.activate_before_layer(data, 0)
    output_words = net.activate_before_layer(words_ngram, 0)

    similar_words = search_for_similar_words(output_words,
        output_all_data, 10)
    similar_words = np.array(similar_words).astype('int')

    for word, similar_indexes in it.izip(words, similar_words):
      print '>' + word
    for s_word_index in similar_indexes:
      print _dict[s_word_index]
```

Вспомогательные функции

```
def get_random_words(data, count):
  word_list = []
  for i in range(0, count):
    _str = data[random.randint(0, len(data))]
    word_list.append(_str)
  return word_list, get_ngrams_data_presentation(word_list)

def search_for_similar_words(words, data, count):
  similar = []
  for word in words:
    similar.append(np.argsort(np.linalg.norm(data - word, axis=1))
        [0:count])
  return similar
```

Вспомогательные функции

```
@staticmethod
def save_network(net, path):
  with open(path, 'wb') as f:
    pickle.dump(net, f)

@staticmethod
def load_network(path):
  with open(path, 'rd') as f:
    net = pickle.load(f)
  return net
```

Результат

>неблагонамеренность
неблагонамеренность
благонамеренность
неблагонамеренный
благонамеренный
намеренность
злонамеренность
неумеренность
неблагонадёжность
непреднамеренность
благонадёжность

>вытатуировать
вытатуировать
татуировать
вытатуироваться
жуировать
стажировать
флуктуировать
выбуксировать
татуироваться
таксировать
пировать

Результат

>отлитие
отлитие
влитие
слитие
отбитие
отит
отбытие
биотит
отплытие
житие
литий

>фонарик
фонарик
фонарщик
фонарь
шарик
сухарик
нар
монарх
икона
парик
нард

Глубокие автоассоциативные НС и их обучение

В обучении глубоких автоассоциативных сетей также, как и при обучении обычных глубоких сетей, можно выделить два этапа:

1. **Предобучение**. На этом этапе выполняются действия, аналогичные производимым при предобучении обычных глубоких сетей с той лишь разницей, что предобучаются лишь те слои, которые производят кодирование информации.
2. **Обучение (fine-tuning)**. На этом этапе предобученный кодировщик «разворачивается», формируя полную автоассоциативную сеть (рисунок 8). При этом в качестве матриц весовых коэффициентов для декодирующих слоев сети берутся транспонированные матрицы для соответствующих кодирующих слоев, а в качестве пороговых элементов выступают векторы порогов соответствующих видимых слоев RBM. После этого производится обучение получившейся автоассоциативной сети методом обратного распространения ошибки.

Обучение глубокой автоассоциативной НС

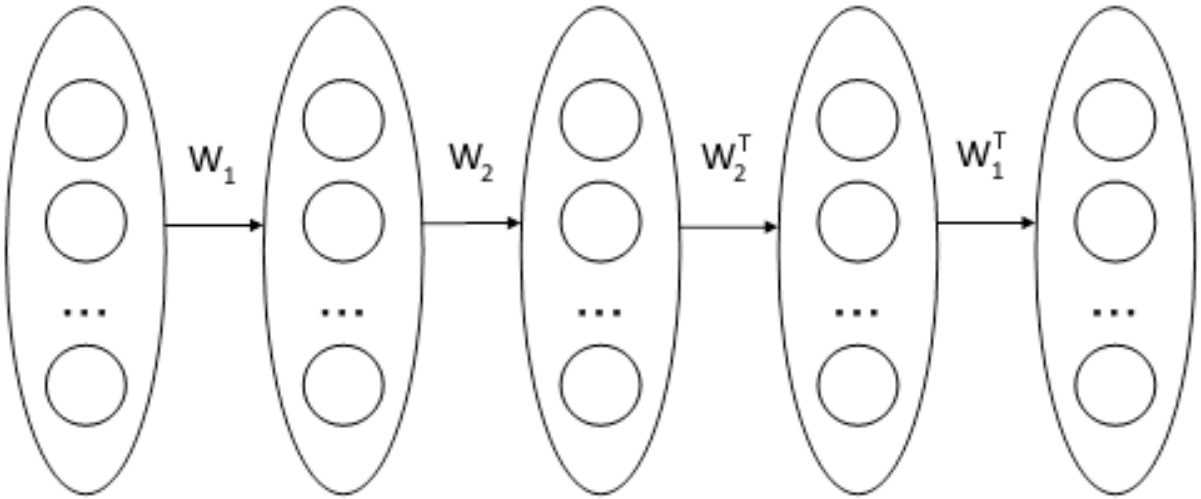


Figure 8: Структура глубокой автоассоциативной сети

Семантические хэширование

Семантическое кодирование (хеширование) относится к более сложному и не менее важному типу задач, решение которого позволяет сформировать бинарный код ограниченной длины, емко и однозначно описывающий изображение в редуцированном признаковом пространстве. С помощью такого преобразования можно сформировать базу изображений, представленных лишь бинарным кодом. На основе такой базы можно построить систему релевантного поиска изображений.

Постановка задачи

В качестве обучающей выборки для решения задачи построения бинарных семантических кодов изображений нами была использована база CIFAR-10. Данная выборка предоставляет богатый материал для выявления семантических особенностей. Она включает в себя 50.000 изображений технических средств и живых существ, принадлежащий 10 различным классам (рис. 9). Каждое изображение имеет размер 32X32 пикселя. Помимо этого, CIFAR-10 включает тестирующую выборку, состоящую из 10.000 изображений.

Выборка CIFAR-10

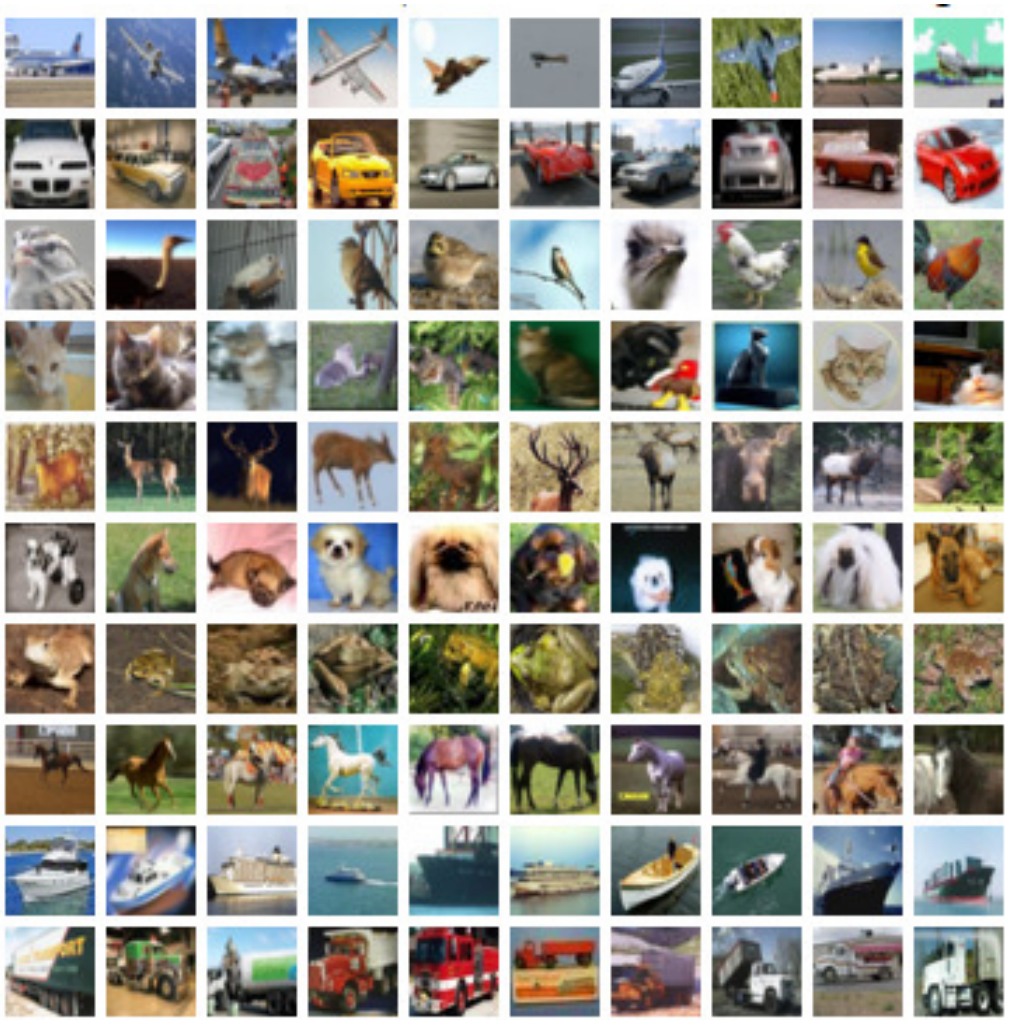


Figure 9: Фрагмент базы изображений CIFAR-10

Для решения задачи нами использовалась **11-слойная ассоциативная нейронная сеть** с архитектурой **3072-4096-2048-1024-512-256-512-1024-2048-4096-3072**. На среднем слое такой сети формируется вектор вещественных значений из отрезка [0, 1], элементы которого затем округляются. Таким образом формируется **бинарный код** изображения. Легко видеть, что таким образом может быть закодировано 2^{256} изображений, что более чем достаточно для базы CIFAR-10. Нами использовались все изображения из обучающей выборки указанной базы.

Обучение проводилось в два этапа. На первом этапе **предобучались** соответствующие RBM, формирующие кодирующие слои сети. Исходные данные перед обучением были стандартизованы: из каждого компонента вектора изображения вычиталось его среднее значение по всем изображениям выборки и затем полученное значение делилось на стандартное отклонение по всем компонентам всех изображений. Данное преобразование определило тип первой обучаемой машины Больцмана – линейно-бинарная RBM. Остальные машины обучались как бинарные RBM.

Каждая RBM обучалась на протяжении 100 эпох мини-батчами по 100 элементов. Для линейно-бинарной RBM использовалось скорость обучения 0,001, для бинарной – 0,01. Помимо этого для ускорения процесса обучения использовался моментный параметр, равный 0,9.
После проведения предобучения выполнялось обучение развернутой автоассоциативной сети методом обратного распространения ошибки. Параметры обучения: скорость – $1e^{-6}$, количество эпох обучения – 150, моментный параметр – 0.9.
После выполнения обучения, результат оценивался вычислением расстояния Хэмминга между бинарными кодами для тестового изображения и изображениями из базы. После этого полученный ряд значений сортировался по возрастанию для выделения наиболее релевантных результатов.

$$H(v_1, v_2) = \sum_{i=1}^{n} |v_1^i - v_2^i|$$

Мы протестировали нашу модель двумя способами. Первый вариант предусматривал визуальное сравнение оригинального и восстановленного изображения. Некоторые из выполненных тестов представлены на рисунке.

Второй вариант тестирования заключался в подаче на обученную автоассоциативную сеть изображения, получения его бинарного кода с промежуточного слоя сети и вычисления расстояния **Хэмминга** для всех остальных изображений. Отсортировав получившуюся последовательность по возрастанию, можно изучить наиболее релевантные результаты поиска.

Можно отметить, что с увеличением расстояния Хэмминга количество изображений того же класса, что и целевое изображение постепенно уменьшается.

Небольшое дополнение в классе Network

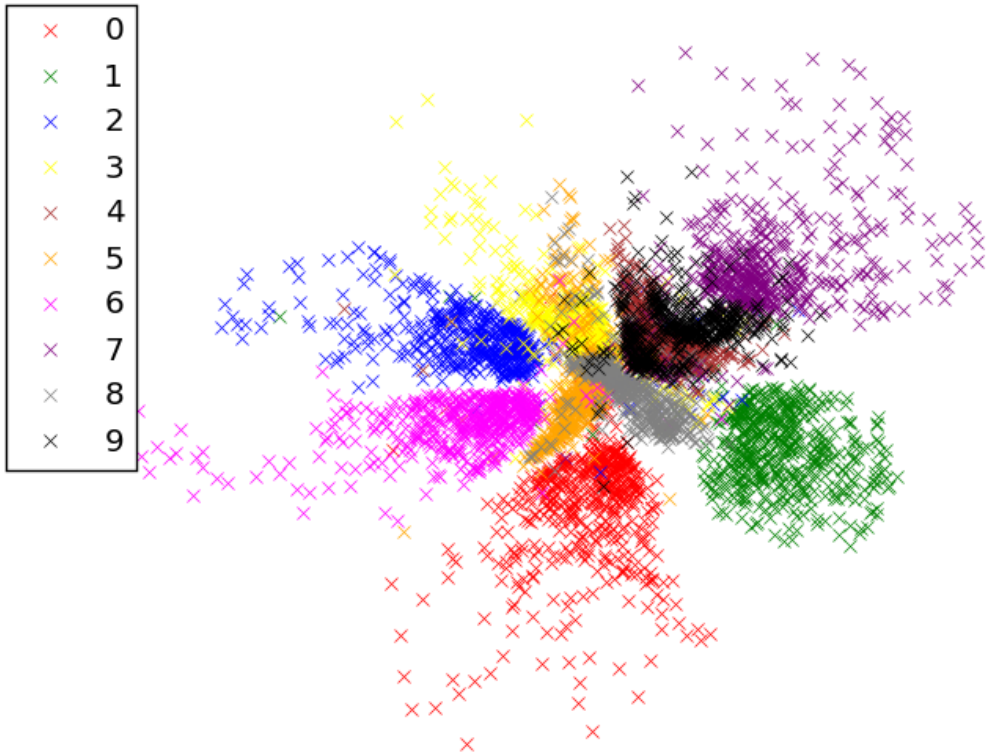
```
@staticmethod
def get_autoencoder_from_rbm_stack(rbm_stack):
  net = Network()
  for i in range(0, len(rbm_stack)):
    rbm = rbm_stack[i]
    new_layer = layer.FullyConnectedLayer(rbm.act_func[1], weights=
        rbm.weights, biases=rbm.hid_biases)
    net.append_layer(new_layer)
  for i in range(len(rbm_stack)-1, -1, -1):
    rbm = rbm_stack[i]
    new_layer = layer.FullyConnectedLayer(rbm.act_func[0], weights=
        rbm.weights.T, biases=rbm.vis_biases)
    net.append_layer(new_layer)

  return net
```

Использование автоэнкодера для сжатия данных

Для иллюстрации нелинейного сжатия использовались данные выборки **MNIST**. Для отображения 784-мерных данных, соответствующих количеству пикселей в исходном изображении, использовалась **глубокая автоассоциативная сеть** с архитектурой **784-500-500-250-10-2**. В качестве функции активации на всех слоях, кроме среднего использовалась **сигмоидная** функция активации. На среднем слое применялась **линейная** функция.
Вначале выполнялось **предобучение** в соответствии с «жадным» послойным алгоритмом. Затем выполнялось «разворачивание» сети в полную архитектуру и производилась **«тонкая» настройка параметров**.
Для предобучения использовались следующие параметры: скорость обучения – 0.2, скорость обучения для среднего слоя – 0.001.
Визуализация для первых 5000 образов из тестовой выборки представлена на рисунке

Визуализация MNIST



Домашнее задание

Изучить выборку Forest Fires (`https://archive.ics.uci.edu/ml/datasets/Forest+Fires`).

- Решить регрессионную задачу на прогнозирование площади леса, поврежденного лесными пожарами. Сравнить результаты и обсудить применяемые модели на следующем занятии.
- Решить задачу кластеризации выборки Seeds Data Set с помощью нейронной сети Кохонена. Сравнить полученные результаты с результатами для алгоритма k-means.