Бинарная классификация

Классификация по двум классам, или бинарная классификация, является едва ли не самой распространенной задачей машинного обучения. Рассмотрим пример бинарной классификации на задаче классификации отзывов к фильмам на положительные и отрицательные, опираясь на текст отзывов.

Для решения задачи будет использовать набор данных IMDB: множеством из 50 000 самых разных отзывов к кинолентам в интернет-базе фильмов (Internet Movie Database). Набор разбит на 25 000 обучающих и 25 000 контрольных отзывов, каждый набор на 50 % состоит из отрицательных и на 50 % из положительных отзывов. Данный набор поставляется вместе с библиотекой Keras.

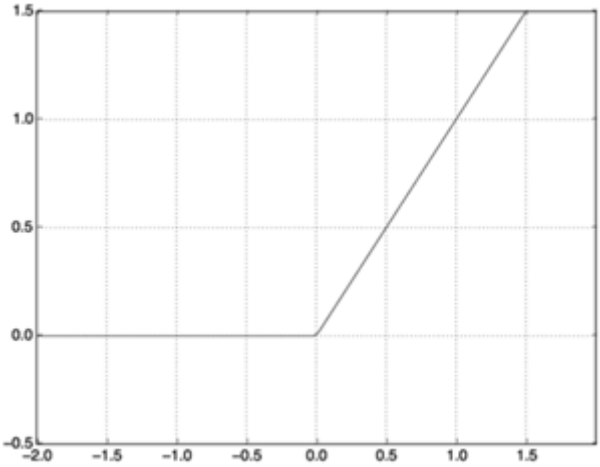
Для начала, подключим этот набор данных, а затем создадим и инициализируем переменные *train_data* и *test_data* (списки отзывов; каждый отзыв — это список индексов слов) и переменные *train_labels* и *test_labels* (списки нулей и единиц, где нули соответствуют отрицательным отзывам, а единицы — положительным):

```
from keras.datasets import imdb
(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words=10000)
#num_words = 10000 – означает, что рассматриваться будут только 10000 самых популярных
слов в отзывах
```

Так как нельзя передать списки целых чисел непосредственно в нейронную сеть. Поэтому необходимо преобразовать их в тензоры. Для данной задачи это сделаем следующим образом: выполним прямое кодирование списков в векторы нулей и единиц . Например, преобразование последовательности [3, 5] в 10 000-мерный вектор, все элементы которого содержат нули, кроме элементов с индексами 3 и 5, которые содержат единицы. Таким образом, будет учитываться только наличие слов в тексте, но не то, сколько раз они встречаются, также не будет учитываться порядок слов.

```
import numpy as np #импорт библиотеки NumPy с тензорами
#определение ф-ции векторизации
def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results

x_train = vectorize_sequences(train_data) #векторизация обучающих данных
x_test = vectorize_sequences(test_data) #векторизация тестовых данных

#векторизация меток данных с вещественным типом
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
```

Входные данные представлены векторами, а метки — скалярами, это самый простой набор данных, какой можно встретить. С задачами этого вида прекрасно справляются сети, организованные как простой стек полносвязных (Dense) слоев с операцией активации relu:

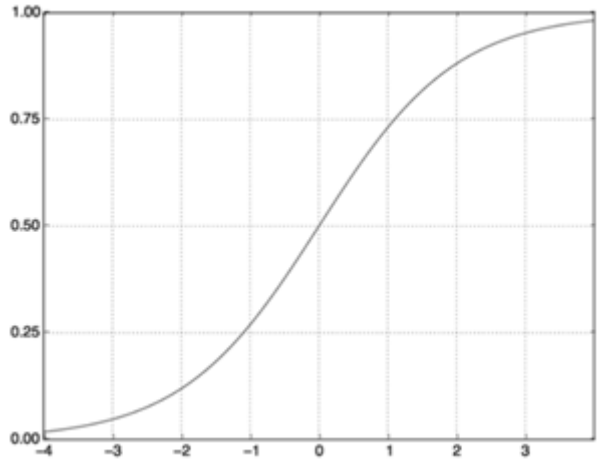


При построении такой сети (еще называется многослойный перцептрон, многослойная сеть прямого распространения) необходимо принимать следующие решения:

- Сколько слоев использовать
- Сколько скрытых нейронов выбрать для каждого слоя

Для данной задачи выберем следующую архитектуру:

- 2 полносвязных скрытых слоя с 16 нейронами
- 1 выходной слой с 1 нейроном, так как в результате необходимо получить скаляр
- На полносвязных слоях будет использовать ф-ция активации relu (рис. 1)
- На выходном слое будет использоваться сигмоидная функция (рис. 2), которая имеет область значений [0 1]. Данные значения можно интерпретировать как вероятность того, что отзыв положительный.



Построение модели:

```
from keras import models #импорт моделей
from keras import layers #импорт слоев

#создание последовательной модели
model = models.Sequential()
#добавление слоев
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

При создании слоя Dense, также можно указывать наличие нейронной сдвига, способ инициализации весов, ограничения на веса в слое, регуляризацию весов.

После того, как модель создана, необходимо выбрать оптимизатор (то есть каким методом будет проводится оптимизация), функция потерь (как происходит расчет ошибки сети) и метрики (значения, которые будут рассчитываться в ходе обучения). Для настройки обучения модели, используется функция *compile* , для которой в качестве аргументов передаются вышеописанные характеристики обучения:

```
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])
```

В нашей задачи в качестве оптимизатора будет использоваться RMSProp, функцией потерь бинарная кросс-энтропия (функция, которая в основном используется при бинарной классификации), а в качестве метрики используется точность.

Чтобы проконтролировать точность модели во время обучения на данных, которые она прежде не видела, создадим проверочный набор, выбрав 10 000 образцов из оригинального набора обучающих данных:

```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]

y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

Теперь проведем обучение модели в течение 20 эпох (выполнив 20 итераций по всем образцам в тензорах x_train и y_train) пакетами по 512 образцов. В то же время будем следить за потерями и точностью на 10 000 отложенных образцов. Для этого достаточно передать проверочные данные в аргументе validation_data:

```
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

Обратите внимание на то, что вызов model.fit() возвращает объект History . Этот объект имеет поле history — словарь с данными обо всем происходившем в процессе обучения. Заглянем в него:

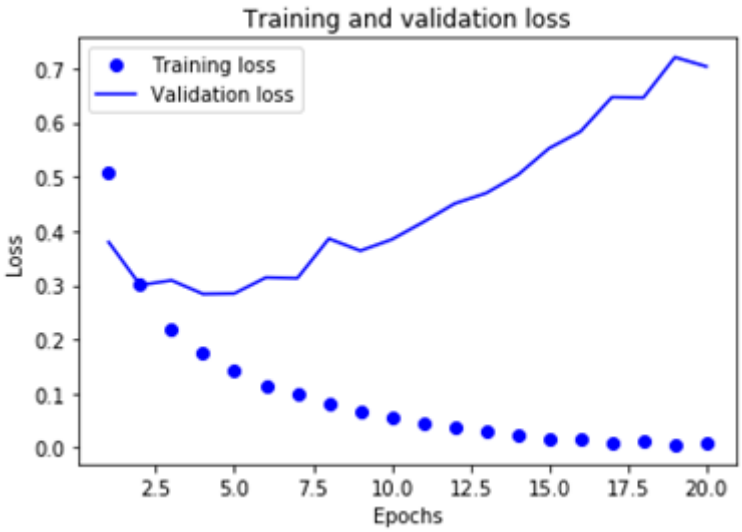
```
history_dict = history.history
print(history_dict.keys())
#dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
```

Словарь содержит четыре элемента — по одному на метрику, — за которыми осуществлялся мониторинг в процессе обучения и проверки.

Далее, выведем графики ошибки и точности в ходе обучения используя библиотеку MatPlotLib:

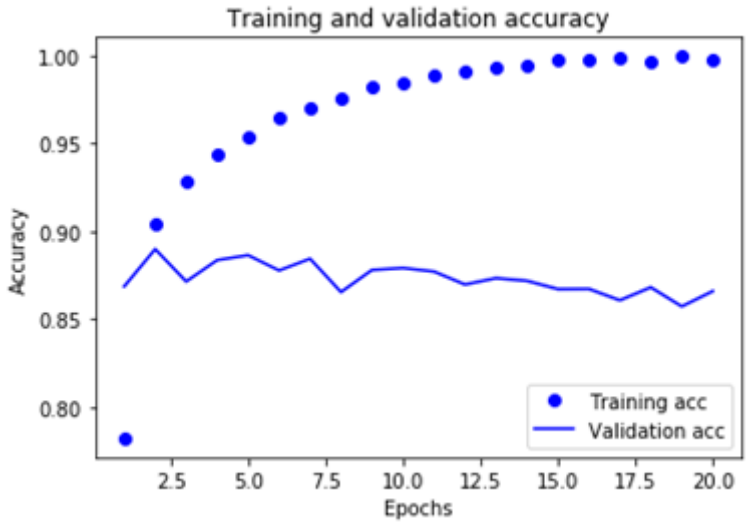
```
import matplotlib.pyplot as plt #импорт модуля для графиков

loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss']
epochs = range(1, len(loss_values) + 1)
plt.plot(epochs, loss_values, 'bo', label='Training loss')
plt.plot(epochs, val_loss_values, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

График ошибки:




```
plt.clf()
acc_values = history_dict['acc']
val_acc_values = history_dict['val_acc']
plt.plot(epochs, acc_values, 'bo', label='Training acc')
plt.plot(epochs, val_acc_values, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

График точности:



Как видите, на этапе обучения потери снижаются с каждой эпохой, а точность растет. Именно такое поведение ожидается от оптимизации градиентным спуском: величина, которую вы пытаетесь минимизировать, должна становиться все меньше с каждой итерацией. Но это не относится к потерям и точности на этапе проверки: похоже, что они достигли пика в четвертую эпоху. Это пример переобучения.

Для оценки модели после обучения на тестовых данных используется функция *evaluate* :

```
model.evaluate(x_test, y_test)
#25000/25000 [==============================] - 2s 79us/step
#[0.7763081940078735, 0.84952]
```

В итоге получили, что для тестовых данных точность составляет 84%.

После обучения сети ее можно использовать для решения практических задач.

Например, попробуем предсказать вероятность того, что отзывы будут положительными, с помощью метода predict:

```
model.predict(x_test)
#array([[0.00725254],
#       [0.9999999 ],
#       [0.692512  ],
#       ...,
#      [0.00239742],
#       [0.01149198],
#       [0.6964724 ]], dtype=float32)
```

Как видно, есть результаты близкие к 0 или 1, но есть и такие как 0.69. При новом обучении сети, может оказаться, что результаты будут отличаться. Поэтому целесообразно обучить на 4 эпохах.

Классификация нескольких классов

Данную задачу рассмотрим на наборе данных Reuters — выборкой новостных лент и их тем, публиковавшихся агентством Reuters в 1986 году. Это простой набор данных, широко используемых для классификации текста. Существует 46 разных тем; некоторые темы более широко представлены, некоторые — менее, но для каждой из них в обучающем наборе имеется не менее 10 примеров.

Конструирование сети для данной задачи по большей части похож на конструирование сети для бинарной классификации.

Подобно IMDB, набор данных Reuters поставляется в составе Keras. Поэтмоу загрузка, а также подготовка данных аналогична как и в прошлой задаче:

```
from keras.datasets import reuters

(train_data, train_labels), (test_data, test_labels) =
reuters.load_data(num_words=10000)

import numpy as np
def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results

x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
```

Так как в результате необходимо получать информацию о принадлежности к одному из классов, то необходимо подготовить соответствующе метки. Для этого. Векторизовать метки можно одним из двух способов: сохранить их в тензоре целых чисел или использовать прямое кодирование. Прямое кодирование (one-hot encoding) широко используется для форматирования категорий и также называется кодированием категорий (categorical encoding). В данном случае прямое кодирование меток заключается в конструировании вектора с нулевыми элементами со значением 1 в элементе, индекс которого соответствует индексу метки:

```
def to_one_hot(labels, dimension=46):
    results = np.zeros((len(labels), dimension))
    for i, label in enumerate(labels):
        results[i, label] = 1.
    return results

one_hot_train_labels = to_one_hot(train_labels)
one_hot_test_labels = to_one_hot(test_labels)
```

Данный метод уже реализован в библиотеке Keras:

```
from keras.utils.np_utils import to_categorical
one_hot_train_labels = to_categorical(train_labels)
one_hot_test_labels = to_categorical(test_labels)
```

Задача классификации по темам напоминает предыдущую задачу классификации отзывов: в обоих случаях мы пытаемся классифицировать короткие фрагменты текста. Но в данном случае количество выходных классов увеличилось с 2 до 46. Размерность выходного пространства теперь намного больше. В стеке слоев Dense , как в предыдущем примере, каждый слой имеет доступ только к информации, предоставленной предыдущим слоем. Если один слой отбросит какую-то информацию, важную для решения задачи классификации, последующие слои не смогут восстановить ее: каждый слой может стать узким местом для информации. В предыдущем примере мы использовали 16-мерные промежуточные слои, но 16-мерное пространство может оказаться слишком ограниченным для классификации на 46 разных классов: такие малоразмерные слои могут сыграть роль «бутылочного горлышка» для информации, не пропуская важные данные. По этой причине в данном примере мы будем использовать слои с большим количеством измерений. Давайте выберем 64 нейрона:

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))
```

Отметим еще две особенности этой архитектуры:

- Сеть завершается слоем Dense с размером 46. Это означает, что для каждого входного образца сеть будет выводить 46-мерный вектор.
- Последний слой использует функцию активации softmax. Он означает, что сеть будет выводить распределение вероятностей по 46 разным классам — для каждого образца на входе сеть будет возвращать 46-мерный вектор, где output[i] — вероятность принадлежности образца классу i . Сумма 46 элементов всегда будет равна 1.

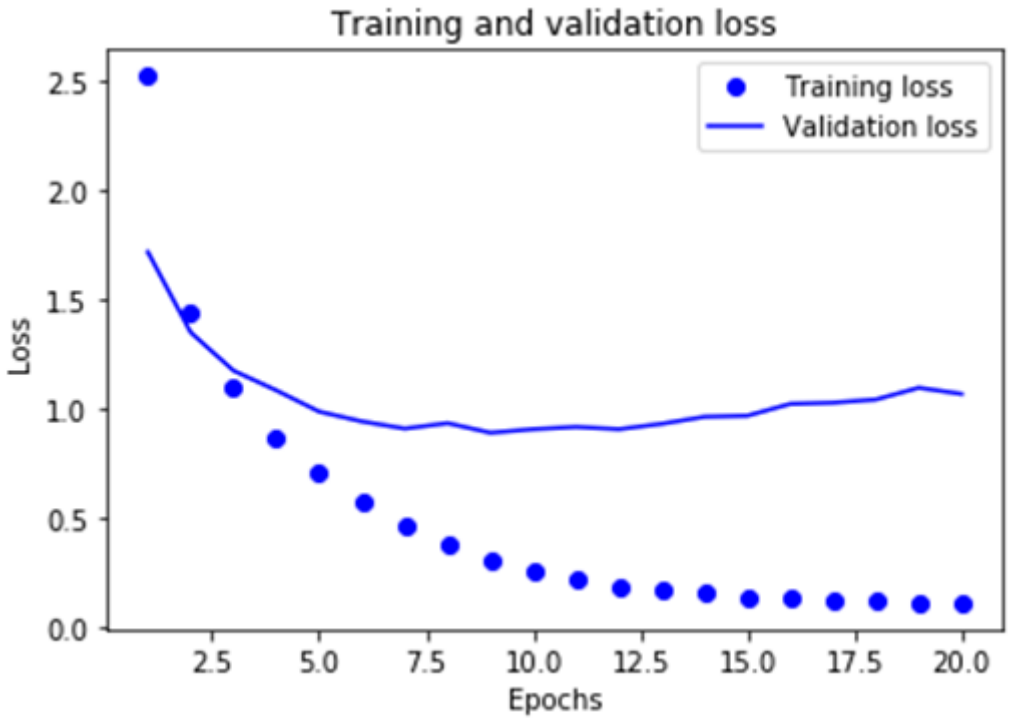
Лучшим вариантом в данном случае является использование функции потерь categorical_crossentropy . Она определяет расстояние между распределениями вероятностей: в данном случае между распределением вероятности на выходе сети и истинным распределением меток. Минимизируя расстояние между этими двумя распределениями, мы учим сеть выводить результат, максимально близкий к истинным меткам:

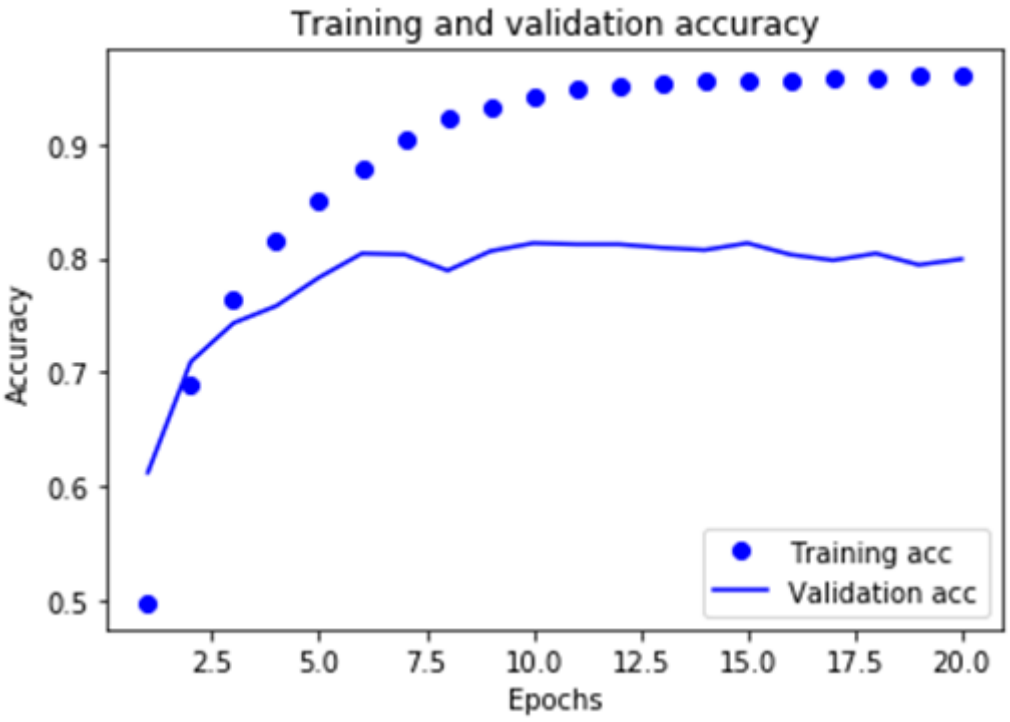
```
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

Проверка, обучение и вывод графиков проводятся аналогичным образом:

```
x_val = x_train[:1000]
partial_x_train = x_train[1000:]
y_val = one_hot_train_labels[:1000]
partial_y_train = one_hot_train_labels[1000:]

history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))

import matplotlib.pyplot as plt
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(loss) + 1)
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

plt.clf()
acc = history.history['acc']
val_acc = history.history['val_acc']
plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

results = model.evaluate(x_test, one_hot_test_labels)
print(results)
```

Итоговый результат, точность классификации тестовой выборки 77%:





Как видно, опять произошло переобучение. Из графиков видно, что обучение необходимо проводить в течении 9 эпох.

Просмотрим результаты для первого наблюдения из тестовой выборкиЖ

```
predictions = model.predict(x_test)
predictions[0].shape
#(46,) #размер выходного вектора
np.sum(predictions[0])
#1.0 #сумма вероятностей, из-за погрешности не ровно 1
np.argmax(predictions[0])
#3 #к какому классу отнесено наблюдение
np.max(predictions[0])
#0.8662524 #с какой вероятностью отнесено к классу
```