

Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
Факультет компьютерных систем и сетей
Кафедра информатики

Машинное обучение

Лабораторные работы №№1-8

Выполнила:
магистрант специальности ИиТРПО
Евтушенко Елизавета Юрьевна
2 курс, группа 858341

Проверил:
Стержанов Максим Валерьевич

Лабораторная работа №1

Логистическая регрессия в качестве нейронной сети

В работе предлагается использовать набор данных *notMNIST*, который состоит из изображений размерностью 28×28 первых 10 букв латинского алфавита (*A* ... *J*, соответственно). Обучающая выборка содержит порядка 500 тыс. изображений, а тестовая – около 19 тыс.

Данные можно скачать по ссылке:

- https://commondatastorage.googleapis.com/books1000/notMNIST_large.tar.gz (https://commondatastorage.googleapis.com/books1000/notMNIST_large.tar.gz) (большой набор данных);
- https://commondatastorage.googleapis.com/books1000/notMNIST_small.tar.gz (https://commondatastorage.googleapis.com/books1000/notMNIST_small.tar.gz) (маленький набор данных);

Описание данных на английском языке доступно по ссылке: <http://yaroslavvb.blogspot.sg/2011/09/notmnist-dataset.html> (<http://yaroslavvb.blogspot.sg/2011/09/notmnist-dataset.html>).

Задание 1

Загрузите данные и отобразите на экране несколько из изображений с помощью языка Python.

In [0]:

```
SMALL_DS_URL = 'https://commondatastorage.googleapis.com/books1000/notMNIST_small.tar.gz'  
LARGE_DS_URL = 'https://commondatastorage.googleapis.com/books1000/notMNIST_large.tar.gz'
```

In [0]:

```
%matplotlib inline  
  
import matplotlib.pyplot as plt
```

In [0]:

```
from urllib.request import urlretrieve
import tarfile
import os

def tar_to_dir(_tar_url, _key):
    dir_name_ = 'dataset_' + _key
    local_file_name_ = dir_name_ + '.f'

    urlretrieve(_tar_url, local_file_name_)

    with tarfile.open(local_file_name_, 'r:gz') as tar_:
        tar_.extractall(dir_name_)

    os.remove(local_file_name_)

    return dir_name_
```

In [0]:

```
def get_examples(_dataframe, _label_column_name, _data_column_name):
    n_ = _dataframe[_label_column_name].nunique()

    examples_ = _dataframe.sample(n_)[_data_column_name]

    return examples_
```

In [0]:

```
from math import ceil
import numpy as np

def print_examples(_examples):
    fig = plt.figure(figsize = (16, 6))

    height_ = 2
    width_ = ceil(_examples.count() / height_)

    for i, item_ in enumerate(_examples):

        ax = fig.add_subplot(height_, width_, i + 1)
        ax.axis('off')
        ax.imshow(item_, cmap = 'gray', interpolation = 'none')

    plt.show()
```

In [0]:

```
from imageio import imread
import pandas as pd

def image_to_array(_image):
    try:
        array_ = imread(_image)

        return True, array_
    except:
        return False, None

def get_inner_dir(_dir_path):
    return [x[0] for x in os.walk(_dir_path)][1]

def remove_duplicates(_dataframe, _data_column_name):
    return _dataframe.loc[_dataframe[_data_column_name].astype(str).drop_duplicates().index]

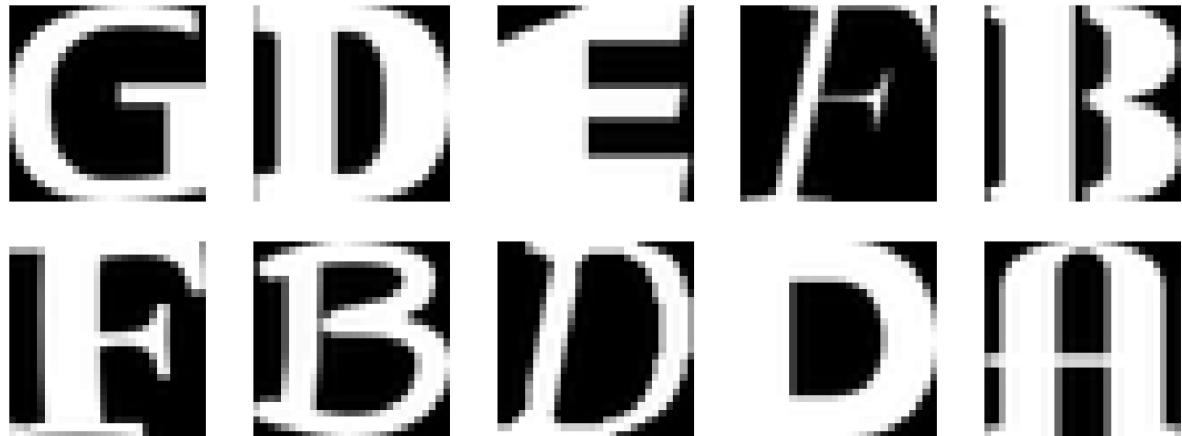
def dir_to_dataframe(_dir_path):
    dataframes_ = []
    inner_dir_path_ = get_inner_dir(_dir_path)
    for subdir_ in sorted(os.listdir(inner_dir_path_)):
        letter_ = subdir_
        data_ = []
        files_ = os.listdir(os.path.join(inner_dir_path_, subdir_))
        for f in files_:
            file_path_ = os.path.join(inner_dir_path_, subdir_, f)
            can_read_, im = image_to_array(file_path_)
            if can_read_:
                data_.append(im)
        g = [letter_] * len(data_)
        e = np.array(data_)
        h = pd.DataFrame()
        h['data'] = data_
        h['label'] = letter_
        dataframes_.append(h)
    result_ = pd.concat(dataframes_, ignore_index = True)
    unique_ = remove_duplicates(result_, 'data')
    return unique_
```

In [0]:

```
def tar_to_dataframe(_tar_url, _key):  
    dir_name_ = tar_to_dir(_tar_url, _key)  
    inner_dir_ = get_inner_dir(dir_name_)  
    dataframe_ = dir_to_dataframe(dir_name_)  
    examples_ = get_examples(dataframe_, 'label', 'data')  
    print_examples(examples_)  
  
    return dataframe_
```

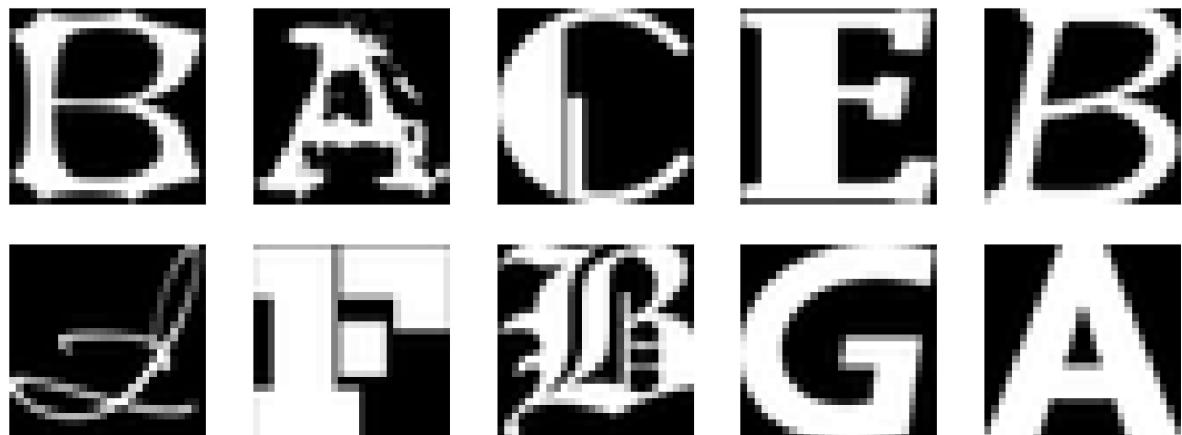
In [8]:

```
small_dataframe = tar_to_dataframe(SMALL_DS_URL, 'small')
```



In [9]:

```
large_dataframe = tar_to_dataframe(LARGE_DS_URL, 'large')
```



Задание 2

Проверьте, что классы являются сбалансированными, т.е. количество изображений, принадлежащих каждому из классов, примерно одинаково (в данной задаче 10 классов).

In [0]:

```
def print_balance(_dataframe, _label_column_name):  
    values_ = _dataframe[_label_column_name].value_counts().sort_values(ascending = False)  
    print((':>10') * len(values_)).format(*values_))
```

In [11]:

```
print_balance(small_dataframe, 'label')
```

1853	1850	1850	1848	1848	1848	1847	1847
1847	1845	1596					

In [12]:

```
print_balance(large_dataframe, 'label')
```

47226	47102	47012	46890	46771	46663	46577	4
6521	46098	41086					

Как видим, классы сбалансированы.

Задание 3

Разделите данные на три подвыборки: обучающую (200 тыс. изображений), валидационную (10 тыс. изображений) и контрольную (тестовую) (19 тыс. изображений).

In [0]:

```
def split(_dataframe, _n_train, _n_test, _n_val):  
  
    assert _dataframe.shape[0] >= _n_train + _n_test + _n_val  
  
    to_be_split_ = _dataframe.copy(deep = True)  
  
    seed_ = 666  
  
    train_ = to_be_split_.sample(n = _n_train, random_state = seed_)  
  
    to_be_split_ = to_be_split_.drop(train_.index)  
    test_ = to_be_split_.sample(n = _n_test, random_state = seed_)  
  
    val_ = to_be_split_.drop(test_.index).sample(n = _n_val, random_state = seed_)  
  
    return train_, test_, val_
```

In [14]:

```
large_dataframe.shape[0]
```

Out[14]:

461946

In [15]:

```
train, test, validation = split(large_dataframe, 200000, 10000, 19000)

print_balance(train, 'label')
print_balance(test, 'label')
print_balance(validation, 'label')
```

	20415	20350	20317	20290	20278	20191	20170	2
0124	20100	17765						
	1049	1043	1033	1029	1021	1016	995	
981	961	872						
	2014	1945	1934	1931	1912	1903	1898	
1887	1864	1712						

Видно, что удалось сохранить баланс между классами.

Задание 4

Проверьте, что данные из обучающей выборки не пересекаются с данными из валидационной и контрольной выборок. Другими словами, избавьтесь от дубликатов в обучающей выборке.

In [0]:

```
def no_duplicates(_dataframe, _data_column_name):

    original_length_ = _dataframe.shape[0]

    unique_length_ = _dataframe[_data_column_name].astype(str).unique().shape[0]

    print(str(original_length_) + ' -- ' + str(unique_length_))

    return original_length_ == unique_length_
```

In [17]:

```
print(no_duplicates(small_dataframe, 'data'))
```

```
18232 -- 18232
True
```

In [18]:

```
print(no_duplicates(large_dataframe, 'data'))
```

```
461946 -- 461946
True
```

In [0]:

```
small_dataframe.to_pickle("./small.pkl")
large_dataframe.to_pickle("./large.pkl")
```

Дубликатов не обнаружено, так как они были удалены на шаге построения датасета из файлов.

Задание 5

Постройте простейший классификатор (например, с помощью логистической регрессии). Постройте график зависимости точности классификатора от размера обучающей выборки (50, 100, 1000, 50000). Для построения классификатора можете использовать библиотеку *SkLearn* (<http://scikit-learn.org>). (<http://scikit-learn.org>).

In [0]:

```
def dataframe_to_x_y(_dataframe):  
    x_ = np.stack(_dataframe['data']).reshape((_dataframe.shape[0], -1))  
    y_ = _dataframe['label'].to_numpy()  
  
    return x_, y_
```

In [0]:

```
X_train, y_train = dataframe_to_x_y(train)  
X_test, y_test = dataframe_to_x_y(test)
```

In [0]:

```
sizes = [50, 100, 1000, 50000]  
  
clfs = {}  
  
scores = {}
```

In [23]:

```
from sklearn.linear_model import LogisticRegression

for size_ in sizes:

    clf_ = LogisticRegression(max_iter = 100).fit(X_train[:size_], y_train[:size_])

    clfs[size_] = clf_
```

/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:94
0: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. OF ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
<https://scikit-learn.org/stable/modules/preprocessing.html> (<https://scikit-learn.org/stable/modules/preprocessing.html>)

Please also refer to the documentation for alternative solver options:

https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression (https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression)

extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)

/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:94
0: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. OF ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
<https://scikit-learn.org/stable/modules/preprocessing.html> (<https://scikit-learn.org/stable/modules/preprocessing.html>)

Please also refer to the documentation for alternative solver options:

https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression (https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression)

extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)

/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:94
0: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. OF ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
<https://scikit-learn.org/stable/modules/preprocessing.html> (<https://scikit-learn.org/stable/modules/preprocessing.html>)

Please also refer to the documentation for alternative solver options:

https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression (https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression)

extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)

In [24]:

```
print(*clfs[50000].predict(X_test[:10]), sep = '\t')
```

E C B H C G E C G H

In [25]:

```
print(*y_test[:10], sep = '\t')
```

C C B H C G B E G H

In [0]:

```
for size_ in sizes:  
    scores[size_] = clfs[size_].score(X_test, y_test)
```

In [27]:

```
print(scores)
```

```
{50: 0.559, 100: 0.6609, 1000: 0.7231, 50000: 0.8155}
```

In [28]:

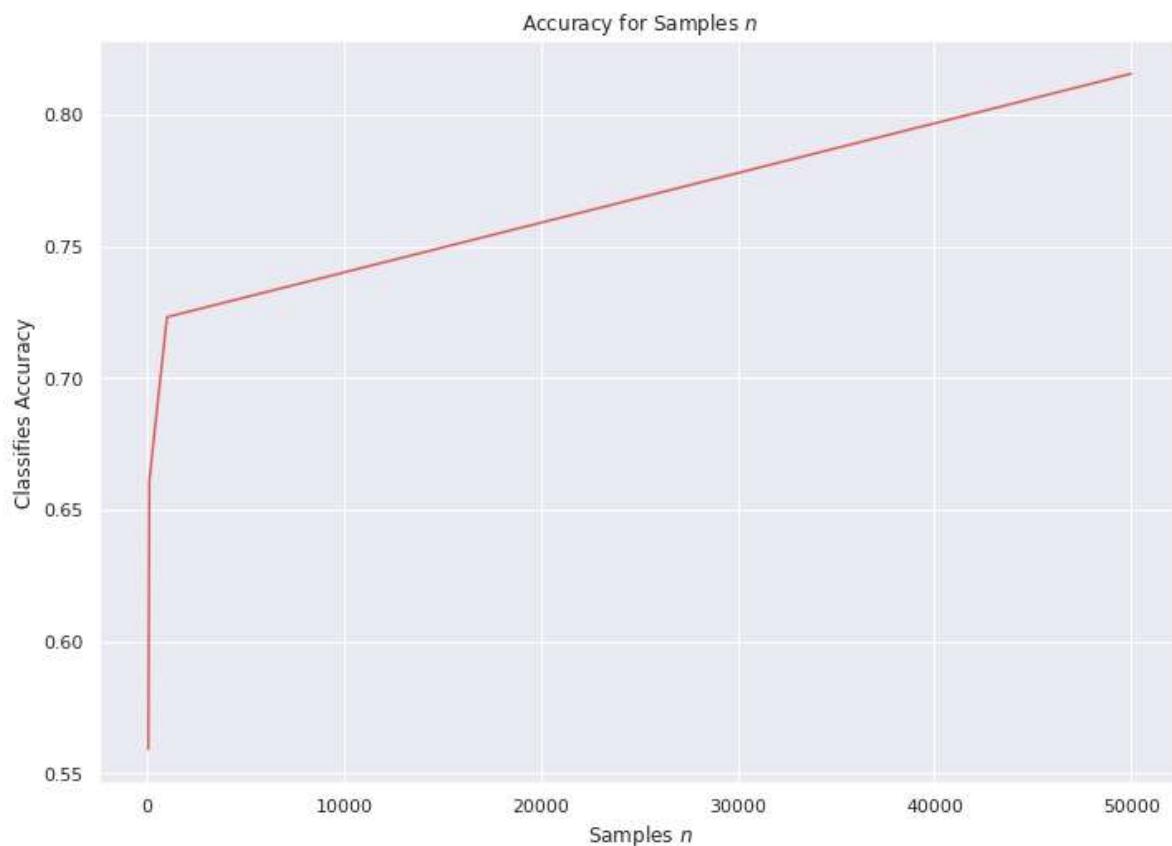
```
import seaborn as sns  
  
from matplotlib import rcParams  
  
rcParams['figure.figsize'] = 11.7, 8.27  
  
sns.set()  
  
sns.set_palette(sns.color_palette('hls'))
```

```
/usr/local/lib/python3.6/dist-packages/statsmodels/tools/_testing.py:19: FutureWarning: pandas.util.testing is deprecated. Use the functions in the public API at pandas.testing instead.
```

```
    import pandas.util.testing as tm
```

In [29]:

```
sns.lineplot(sizes, [scores[s] for s in sizes])  
  
plt.xlabel('Samples $n$')  
plt.ylabel('Classifies Accuracy')  
  
plt.title('Accuracy for Samples $n$')  
  
plt.show()
```



На графике видим, что с увеличением выборки качество классификации растёт как логарифмическая функция от размера выборки.

Лабораторная работа №2

Реализация глубокой нейронной сети

В работе предлагается использовать набор данных *notMNIST*, который состоит из изображений размерностью 28×28 первых 10 букв латинского алфавита (*A* ... *J*, соответственно). Обучающая выборка содержит порядка 500 тыс. изображений, а тестовая – около 19 тыс.

Данные можно скачать по ссылке:

- https://commondatastorage.googleapis.com/books1000/notMNIST_large.tar.gz (https://commondatastorage.googleapis.com/books1000/notMNIST_large.tar.gz) (большой набор данных);
- https://commondatastorage.googleapis.com/books1000/notMNIST_small.tar.gz (https://commondatastorage.googleapis.com/books1000/notMNIST_small.tar.gz) (маленький набор данных);

Описание данных на английском языке доступно по ссылке: <http://yaroslavvb.blogspot.sg/2011/09/notmnist-dataset.html> (<http://yaroslavvb.blogspot.sg/2011/09/notmnist-dataset.html>).

Задание 1

Реализуйте полносвязную нейронную сеть с помощью библиотеки *TensorFlow*. В качестве алгоритма оптимизации можно использовать, например, стохастический градиент (*Stochastic Gradient Descent*, *SGD*). Определите количество скрытых слоев от 1 до 5, количество нейронов в каждом из слоев до нескольких сотен, а также их функции активации (кусочно-линейная, сигмоидная, гиперболический тангенс и т.д.).

In [1]:

```
from google.colab import drive  
  
drive.mount('/content/drive', force_remount = True)
```

Go to this URL in a browser: [\(https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redirect_uri=urn%3aietf%3awg%3aoauth%3a2.0%3aoob&response_type=code&scope=email%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdocs.test%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive.photos.readonly\)](https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redirect_uri=urn%3aietf%3awg%3aoauth%3a2.0%3aoob&response_type=code&scope=email%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdocs.test%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive.photos.readonly%20https%3a%2f%2fwww.googleapis.com%2fauth%2fpeopleapi.readonly (https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redirect_uri=urn%3aietf%3awg%3aoauth%3a2.0%3aoob&response_type=code&scope=email%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdocs.test%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive.photos.readonly))

Enter your authorization code:

.....

Mounted at /content/drive

In [0]:

```
BASE_DIR = '/content/drive/My Drive/Colab Files/mo-2'

import sys

sys.path.append(BASE_DIR)

import os

os.chdir(BASE_DIR)
```

In [0]:

```
import pandas as pd

dataframe = pd.read_pickle("./large.pkl")
```

In [4]:

```
dataframe['data'].shape
```

Out[4]:

```
(461946,)
```

In [5]:

```
! pip install tensorflow-gpu --pre --quiet

! pip show tensorflow-gpu
```

```
|██████████| 516.2MB 26kB/s
Name: tensorflow-gpu
Version: 2.2.0rc3
Summary: TensorFlow is an open source machine learning framework for everyone.
Home-page: https://www.tensorflow.org/ (https://www.tensorflow.org/)
Author: Google Inc.
Author-email: packages@tensorflow.org
License: Apache 2.0
Location: /usr/local/lib/python3.6/dist-packages
Requires: termcolor, tensorflow-estimator, opt-einsum, wrapt, absl-py, protobuf, tensorboard, astunparse, keras-preprocessing, h5py, numpy, wheel, scipy, gast, six, google-pasta, grpcio
Required-by:
```

In [0]:

```
import tensorflow as tf
```

In [0]:

```
import numpy as np
```

In [0]:

```
dataframe_test = dataframe.sample(frac = 0.1)

dataframe = dataframe.drop(dataframe_test.index)
```

In [9]:

```
x = np.asarray(list(dataframe['data']))[..., np.newaxis]

x = tf.keras.utils.normalize(x, axis = 1)

x.shape
```

Out[9]:

```
(415751, 28, 28, 1)
```

In [31]:

```
x_test = np.asarray(list(dataframe_test['data']))[..., np.newaxis]

x_test = tf.keras.utils.normalize(x_test, axis = 1)

x_test.shape
```

Out[31]:

```
(46195, 28, 28, 1)
```

In [0]:

```
IMAGE_DIM_0, IMAGE_DIM_1 = x.shape[1], x.shape[2]
```

In [11]:

```
from tensorflow.keras.utils import to_categorical

y = to_categorical(dataframe['label'].astype('category').cat.codes.astype('int32'))

y.shape
```

Out[11]:

```
(415751, 10)
```

In [32]:

```
y_test = to_categorical(dataframe_test['label'].astype('category').cat.codes.astype('int32')

y_test.shape
```

Out[32]:

```
(46195, 10)
```

In [0]:

```
LAYER_WIDTH = 5000
```

In [0]:

```
CLASSES_N = y.shape[1]
```

In [0]:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Reshape

model = tf.keras.Sequential()

model.add(Reshape((IMAGE_DIM_0 * IMAGE_DIM_1,), input_shape = (IMAGE_DIM_0, IMAGE_DIM_1, 1)))
model.add(Dense(LAYER_WIDTH, activation = 'relu'))
model.add(Dense(LAYER_WIDTH, activation = 'sigmoid'))
model.add(Dense(LAYER_WIDTH, activation = 'tanh'))
model.add(Dense(LAYER_WIDTH, activation = 'elu'))
model.add(Dense(LAYER_WIDTH, activation = 'softmax'))
model.add(Dense(CLASSES_N))
```

In [0]:

```
def cat_cross_from_logits(y_true, y_pred):
    return tf.keras.losses.categorical_crossentropy(y_true, y_pred, from_logits = True)

model.compile(optimizer = 'sgd',
              loss = cat_cross_from_logits,
              metrics = ['categorical_accuracy'])
```

In [16]:

```
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
<hr/>		
reshape (Reshape)	(None, 784)	0
dense (Dense)	(None, 5000)	3925000
dense_1 (Dense)	(None, 5000)	25005000
dense_2 (Dense)	(None, 5000)	25005000
dense_3 (Dense)	(None, 5000)	25005000
dense_4 (Dense)	(None, 5000)	25005000
dense_5 (Dense)	(None, 10)	50010
<hr/>		

Total params: 103,995,010

Trainable params: 103,995,010

Non-trainable params: 0

In [0]:

```
VAL_SPLIT_RATE = 0.1
```

In [0]:

```
EPOCHS_N = 10
```

In [19]:

```
model.fit(x = x, y = y, epochs = EPOCHS_N, validation_split = VAL_SPLIT_RATE)
```

```
Epoch 1/10
11693/11693 [=====] - 103s 9ms/step - loss: 2.2195
- categorical_accuracy: 0.1122 - val_loss: 4.8722 - val_categorical_accuracy: 0.0000e+00
Epoch 2/10
11693/11693 [=====] - 102s 9ms/step - loss: 2.2042
- categorical_accuracy: 0.1130 - val_loss: 5.4886 - val_categorical_accuracy: 0.0000e+00
Epoch 3/10
11693/11693 [=====] - 102s 9ms/step - loss: 2.2025
- categorical_accuracy: 0.1126 - val_loss: 5.8407 - val_categorical_accuracy: 0.0000e+00
Epoch 4/10
11693/11693 [=====] - 102s 9ms/step - loss: 2.2018
- categorical_accuracy: 0.1127 - val_loss: 6.0791 - val_categorical_accuracy: 0.0000e+00
Epoch 5/10
11693/11693 [=====] - 102s 9ms/step - loss: 2.2015
- categorical_accuracy: 0.1123 - val_loss: 6.2565 - val_categorical_accuracy: 0.0000e+00
Epoch 6/10
11693/11693 [=====] - 102s 9ms/step - loss: 2.2013
- categorical_accuracy: 0.1127 - val_loss: 6.3940 - val_categorical_accuracy: 0.0000e+00
Epoch 7/10
11693/11693 [=====] - 102s 9ms/step - loss: 2.2012
- categorical_accuracy: 0.1134 - val_loss: 6.5053 - val_categorical_accuracy: 0.0000e+00
Epoch 8/10
11693/11693 [=====] - 102s 9ms/step - loss: 2.2011
- categorical_accuracy: 0.1130 - val_loss: 6.5967 - val_categorical_accuracy: 0.0000e+00
Epoch 9/10
11693/11693 [=====] - 102s 9ms/step - loss: 2.2010
- categorical_accuracy: 0.1134 - val_loss: 6.6732 - val_categorical_accuracy: 0.0000e+00
Epoch 10/10
11693/11693 [=====] - 102s 9ms/step - loss: 2.2010
- categorical_accuracy: 0.1134 - val_loss: 6.7377 - val_categorical_accuracy: 0.0000e+00
```

Out[19]:

```
<tensorflow.python.keras.callbacks.History at 0x7f399433fbe0>
```

In [34]:

```
results = model.evaluate(x_test, y_test)

print('Test loss, test accuracy:', results)
```

```
1444/1444 [=====] - 5s 4ms/step - loss: 2.6571 - categorical_accuracy: 0.1008
Test loss, test accuracy: [2.657094955444336, 0.10081177949905396]
```

Задание 2

Как улучшилась точность классификатора по сравнению с логистической регрессией?

Стало хуже — на тестовой выборке точность составила 10%. Похоже, что данная модель совершенно не подходит для решения этой задачи.

Задание 3

Используйте регуляризацию и метод сброса нейронов (*dropout*) для борьбы с переобучением. Как улучшилось качество классификации?

In [0]:

```
REG_RATE = 0.001
```

In [0]:

```
from tensorflow.keras.regularizers import l2
l2_reg = l2(REG_RATE)
```

In [0]:

```
DROPOUT_RATE = 0.2
```

In [0]:

```
from tensorflow.keras.layers import Dropout
dropout_layer = Dropout(DROPOUT_RATE)
```

In [0]:

```
model_2 = tf.keras.Sequential()

model_2.add(Reshape((IMAGE_DIM_0 * IMAGE_DIM_1,), input_shape = (IMAGE_DIM_0, IMAGE_DIM_1,
model_2.add(Dense(LAYER_WIDTH, activation = 'relu', kernel_regularizer = l2_reg))
model_2.add(dropout_layer)
model_2.add(Dense(LAYER_WIDTH, activation = 'sigmoid', kernel_regularizer = l2_reg))
model_2.add(dropout_layer)
model_2.add(Dense(LAYER_WIDTH, activation = 'tanh', kernel_regularizer = l2_reg))
model_2.add(dropout_layer)
model_2.add(Dense(LAYER_WIDTH, activation = 'sigmoid', kernel_regularizer = l2_reg))
model_2.add(dropout_layer)
model_2.add(Dense(LAYER_WIDTH, activation = 'relu', kernel_regularizer = l2_reg))
model_2.add(dropout_layer)
model_2.add(Dense(CLASSES_N))
```

In [0]:

```
model_2.compile(optimizer = 'sgd',
                 loss = cat_crossentropy,
                 metrics = ['categorical_accuracy'])
```

In [26]:

```
model_2.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
=====		
reshape_1 (Reshape)	(None, 784)	0
dense_6 (Dense)	(None, 5000)	3925000
dropout (Dropout)	(None, 5000)	0
dense_7 (Dense)	(None, 5000)	25005000
dense_8 (Dense)	(None, 5000)	25005000
dense_9 (Dense)	(None, 5000)	25005000
dense_10 (Dense)	(None, 5000)	25005000
dense_11 (Dense)	(None, 10)	50010
=====		
Total params: 103,995,010		
Trainable params: 103,995,010		
Non-trainable params: 0		

In [27]:

```
model_2.fit(x = x, y = y, epochs = EPOCHS_N, validation_split = VAL_SPLIT_RATE)

Epoch 1/10
11693/11693 [=====] - 210s 18ms/step - loss: 19.270
- categorical_accuracy: 0.1154 - val_loss: 19.6030 - val_categorical_accuracy: 0.0000e+00
Epoch 2/10
11693/11693 [=====] - 210s 18ms/step - loss: 12.889
- categorical_accuracy: 0.1193 - val_loss: 15.0547 - val_categorical_accuracy: 0.0000e+00
Epoch 3/10
11693/11693 [=====] - 210s 18ms/step - loss: 8.4027
- categorical_accuracy: 0.3462 - val_loss: 12.9659 - val_categorical_accuracy: 0.0000e+00
Epoch 4/10
11693/11693 [=====] - 210s 18ms/step - loss: 5.6802
- categorical_accuracy: 0.4805 - val_loss: 11.1155 - val_categorical_accuracy: 0.0000e+00
Epoch 5/10
11693/11693 [=====] - 210s 18ms/step - loss: 4.0491
- categorical_accuracy: 0.5286 - val_loss: 9.5503 - val_categorical_accuracy: 0.0000e+00
Epoch 6/10
11693/11693 [=====] - 210s 18ms/step - loss: 3.0297
- categorical_accuracy: 0.5674 - val_loss: 8.9475 - val_categorical_accuracy: 0.0000e+00
Epoch 7/10
11693/11693 [=====] - 210s 18ms/step - loss: 2.3880
- categorical_accuracy: 0.5944 - val_loss: 8.4980 - val_categorical_accuracy: 0.0000e+00
Epoch 8/10
11693/11693 [=====] - 210s 18ms/step - loss: 1.9772
- categorical_accuracy: 0.6292 - val_loss: 7.8002 - val_categorical_accuracy: 0.0000e+00
Epoch 9/10
11693/11693 [=====] - 210s 18ms/step - loss: 1.7196
- categorical_accuracy: 0.6475 - val_loss: 7.6441 - val_categorical_accuracy: 0.0000e+00
Epoch 10/10
11693/11693 [=====] - 211s 18ms/step - loss: 1.5578
- categorical_accuracy: 0.6599 - val_loss: 7.5225 - val_categorical_accuracy: 0.0000e+00
```

Out[27]:

```
<tensorflow.python.keras.callbacks.History at 0x7f399345a9b0>
```

In [36]:

```
results_2 = model_2.evaluate(x_test, y_test)

print('Test loss, test accuracy:', results_2)
```

```
1444/1444 [=====] - 8s 6ms/step - loss: 1.4656 - categorical_accuracy: 0.7228
Test loss, test accuracy: [1.4655554294586182, 0.7227838635444641]
```

Регуляризация и сброс нейронов значительно помогли — модель показывает 72% точности на тестовой

выборке!

Задание 4

Воспользуйтесь динамически изменяемой скоростью обучения (*learning rate*). Наилучшая точность, достигнутая с помощью данной модели составляет 97.1%. Какую точность демонстрирует Ваша реализованная модель?

In [28]:

```
from tensorflow.keras.optimizers import SGD

dyn_lr_sgd = SGD(lr = 0.01, momentum = 0.9)

model_2.compile(optimizer = dyn_lr_sgd,
                 loss = cat_crossentropy,
                 metrics = ['categorical_accuracy'])

model_2.fit(x = x, y = y, epochs = EPOCHS_N, validation_split = VAL_SPLIT_RATE)
```

```
Epoch 1/10
11693/11693 [=====] - 240s 20ms/step - loss: 1.3780
- categorical_accuracy: 0.6521 - val_loss: 6.3145 - val_categorical_accuracy: 0.0000e+00
Epoch 2/10
11693/11693 [=====] - 239s 20ms/step - loss: 1.0968
- categorical_accuracy: 0.7462 - val_loss: 6.1084 - val_categorical_accuracy: 0.0000e+00
Epoch 3/10
11693/11693 [=====] - 239s 20ms/step - loss: 1.0805
- categorical_accuracy: 0.7522 - val_loss: 6.2995 - val_categorical_accuracy: 0.0000e+00
Epoch 4/10
11693/11693 [=====] - 238s 20ms/step - loss: 1.0759
- categorical_accuracy: 0.7534 - val_loss: 6.1664 - val_categorical_accuracy: 0.0000e+00
Epoch 5/10
11693/11693 [=====] - 239s 20ms/step - loss: 1.0707
- categorical_accuracy: 0.7564 - val_loss: 6.4368 - val_categorical_accuracy: 0.0000e+00
Epoch 6/10
11693/11693 [=====] - 238s 20ms/step - loss: 1.0186
- categorical_accuracy: 0.7755 - val_loss: 6.6284 - val_categorical_accuracy: 0.0000e+00
Epoch 7/10
11693/11693 [=====] - 239s 20ms/step - loss: 0.9949
- categorical_accuracy: 0.7805 - val_loss: 6.6951 - val_categorical_accuracy: 0.0000e+00
Epoch 8/10
11693/11693 [=====] - 239s 20ms/step - loss: 0.9902
- categorical_accuracy: 0.7824 - val_loss: 6.1755 - val_categorical_accuracy: 0.0000e+00
Epoch 9/10
11693/11693 [=====] - 239s 20ms/step - loss: 0.9873
- categorical_accuracy: 0.7840 - val_loss: 6.3812 - val_categorical_accuracy: 0.0000e+00
Epoch 10/10
11693/11693 [=====] - 239s 20ms/step - loss: 0.9844
- categorical_accuracy: 0.7852 - val_loss: 6.3063 - val_categorical_accuracy: 0.0000e+00
```

Out[28]:

```
<tensorflow.python.keras.callbacks.History at 0x7f39922ee0f0>
```

In [38]:

```
results_3 = model_2.evaluate(x_test, y_test)
print('Test loss, test accuracy:', results_3)
```

```
1444/1444 [=====] - 8s 6ms/step - loss: 1.4656 - categorical_accuracy: 0.7228
Test loss, test accuracy: [1.465554294586182, 0.7227838635444641]
```

Динамически изменяемая скорость обучения не улучшила результат — 72% на тестовой выборке.

Можно сделать вывод, что модель с полносвязными слоями может использоваться для решения задачи распознавания изображений, однако она очевидно не является наилучшей.

Лабораторная работа №3

Реализация сверточной нейронной сети

В работе предлагается использовать набор данных *notMNIST*, который состоит из изображений размерностью 28×28 первых 10 букв латинского алфавита (*A* ... *J*, соответственно). Обучающая выборка содержит порядка 500 тыс. изображений, а тестовая – около 19 тыс.

Данные можно скачать по ссылке:

- https://commondatastorage.googleapis.com/books1000/notMNIST_large.tar.gz (https://commondatastorage.googleapis.com/books1000/notMNIST_large.tar.gz) (большой набор данных);
- https://commondatastorage.googleapis.com/books1000/notMNIST_small.tar.gz (https://commondatastorage.googleapis.com/books1000/notMNIST_small.tar.gz) (маленький набор данных);

Описание данных на английском языке доступно по ссылке: <http://yaroslavvb.blogspot.sg/2011/09/notmnist-dataset.html> (<http://yaroslavvb.blogspot.sg/2011/09/notmnist-dataset.html>).

Задание 1

Реализуйте нейронную сеть с двумя сверточными слоями, и одним полно связанным с нейронами с кусочно-линейной функцией активации. Какова точность построенной модели?

In [1]:

```
from google.colab import drive  
drive.mount('/content/drive', force_remount = True)
```

Mounted at /content/drive

In [0]:

```
BASE_DIR = '/content/drive/My Drive/Colab Files/mo-2'  
  
import sys  
  
sys.path.append(BASE_DIR)  
  
import os  
  
os.chdir(BASE_DIR)
```

In [0]:

```
import pandas as pd  
  
dataframe = pd.read_pickle("./large.pkl")
```

In [4]:

```
! pip install tensorflow-gpu --pre --quiet
```

```
! pip show tensorflow-gpu
```

```
Name: tensorflow-gpu
Version: 2.2.0rc3
Summary: TensorFlow is an open source machine learning framework for everyone.
Home-page: https://www.tensorflow.org/ (https://www.tensorflow.org/)
Author: Google Inc.
Author-email: packages@tensorflow.org
License: Apache 2.0
Location: /usr/local/lib/python3.6/dist-packages
Requires: astunparse, six, google-pasta, tensorflow-estimator, gast, tensorboard, grpcio, scipy, termcolor, wrapt, keras-preprocessing, protobuf, absl-py, numpy, h5py, opt-einsum, wheel
Required-by:
```

In [0]:

```
import tensorflow as tf
```

In [0]:

```
# To fix memory leak: https://github.com/tensorflow/tensorflow/issues/33009
```

```
tf.compat.v1.disable_eager_execution()
```

In [0]:

```
import numpy as np
```

In [0]:

```
dataframe_test = dataframe.sample(frac = 0.1)
dataframe = dataframe.drop(dataframe_test.index)
```

In [9]:

```
x = np.asarray(list(dataframe['data']))[..., np.newaxis]
x = tf.keras.utils.normalize(x, axis = 1)
x.shape
```

Out[9]:

```
(415751, 28, 28, 1)
```

In [10]:

```
x_test = np.asarray(list(dataframe_test['data']))[..., np.newaxis]  
x_test = tf.keras.utils.normalize(x_test, axis = 1)  
x_test.shape
```

Out[10]:

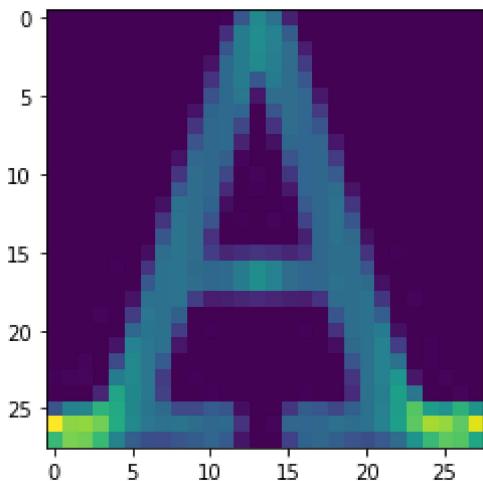
(46195, 28, 28, 1)

In [11]:

```
import matplotlib.pyplot as plt  
  
plt.imshow(x[100].squeeze())
```

Out[11]:

<matplotlib.image.AxesImage at 0x7f6ba8e81a90>



In [0]:

```
IMAGE_DIM_0, IMAGE_DIM_1 = x.shape[1], x.shape[2]
```

In [13]:

```
from tensorflow.keras.utils import to_categorical  
  
y = to_categorical(dataframe['label'].astype('category').cat.codes.astype('int32'))  
y.shape
```

Out[13]:

(415751, 10)

In [14]:

```
y_test = to_categorical(dataframe_test['label'].astype('category').cat.codes.astype('int32')
y_test.shape
```

Out[14]:

```
(46195, 10)
```

In [0]:

```
CLASSES_N = y.shape[1]
```

In [0]:

```
DENSE_LAYER_WIDTH = 5000
```

In [17]:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, Dense, Flatten

model = tf.keras.Sequential()

model.add(Conv2D(16, 3, padding = 'same', activation = 'relu', input_shape = (IMAGE_DIM_0,
model.add(Conv2D(32, 3, padding = 'same', activation = 'relu'))
model.add(Flatten())
model.add(Dense(DENSE_LAYER_WIDTH, activation = 'relu'))
model.add(Dense(CLASSES_N))
```

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow/python/ops/resource_variable_ops.py:1666: calling BaseResourceVariable.__init__(from tensorflow.python.ops.resource_variable_ops) with constraint is deprecated and will be removed in a future version.

Instructions for updating:

If using Keras pass *_constraint arguments to layers.

In [0]:

```
def cat_cross_from_logits(y_true, y_pred):
    return tf.keras.losses.categorical_crossentropy(y_true, y_pred, from_logits = True)

model.compile(optimizer = 'sgd',
              loss = cat_cross_from_logits,
              metrics = ['categorical_accuracy'])
```

In [19]:

```
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
=====		
conv2d (Conv2D)	(None, 28, 28, 16)	160
conv2d_1 (Conv2D)	(None, 28, 28, 32)	4640
flatten (Flatten)	(None, 25088)	0
dense (Dense)	(None, 5000)	125445000
dense_1 (Dense)	(None, 10)	50010
=====		
Total params:	125,499,810	
Trainable params:	125,499,810	
Non-trainable params:	0	

In [0]:

```
VAL_SPLIT_RATE = 0.1
```

In [0]:

```
EPOCHS_N = 10
```

In [22]:

```
model.fit(x = x, y = y, epochs = EPOCHS_N, validation_split = VAL_SPLIT_RATE)

Train on 374175 samples, validate on 41576 samples
Epoch 1/10
374175/374175 [=====] - 114s 304us/sample - loss:
0.4815 - categorical_accuracy: 0.8573 - val_loss: 2.9091 - val_categorical_a
ccuracy: 0.3111
Epoch 2/10
374175/374175 [=====] - 114s 304us/sample - loss:
0.3455 - categorical_accuracy: 0.8956 - val_loss: 2.7738 - val_categorical_a
ccuracy: 0.2886
Epoch 3/10
374175/374175 [=====] - 113s 303us/sample - loss:
0.2926 - categorical_accuracy: 0.9110 - val_loss: 1.9089 - val_categorical_a
ccuracy: 0.5299
Epoch 4/10
374175/374175 [=====] - 113s 302us/sample - loss:
0.2517 - categorical_accuracy: 0.9231 - val_loss: 2.2537 - val_categorical_a
ccuracy: 0.4842
Epoch 5/10
374175/374175 [=====] - 114s 303us/sample - loss:
0.2144 - categorical_accuracy: 0.9342 - val_loss: 2.7603 - val_categorical_a
ccuracy: 0.3479
Epoch 6/10
374175/374175 [=====] - 113s 301us/sample - loss:
0.1803 - categorical_accuracy: 0.9447 - val_loss: 2.1086 - val_categorical_a
ccuracy: 0.5454
Epoch 7/10
374175/374175 [=====] - 114s 305us/sample - loss:
0.1466 - categorical_accuracy: 0.9551 - val_loss: 2.4950 - val_categorical_a
ccuracy: 0.4780
Epoch 8/10
374175/374175 [=====] - 113s 302us/sample - loss:
0.1170 - categorical_accuracy: 0.9647 - val_loss: 2.0592 - val_categorical_a
ccuracy: 0.6014
Epoch 9/10
374175/374175 [=====] - 113s 301us/sample - loss:
0.0925 - categorical_accuracy: 0.9728 - val_loss: 2.6105 - val_categorical_a
ccuracy: 0.5198
Epoch 10/10
374175/374175 [=====] - 113s 302us/sample - loss:
0.0738 - categorical_accuracy: 0.9786 - val_loss: 2.7623 - val_categorical_a
ccuracy: 0.5034
```

Out[22]:

```
<tensorflow.python.keras.callbacks.History at 0x7f6ba8e81cf8>
```

In [23]:

```
results = model.evaluate(x_test, y_test)

print('Test loss, test accuracy:', results)
```

```
Test loss, test accuracy: [0.5621323866975915, 0.88420826]
```

Лучшая точность построенной модели на тестовой выборке составила 88%.

Задание 2

Замените один из сверточных слоев на слой, реализующий операцию пулинга (*Pooling*) с функцией максимума или среднего. Как это повлияло на точность классификатора?

In [0]:

```
from tensorflow.keras.layers import MaxPooling2D

model_2 = tf.keras.Sequential()

model_2.add(Conv2D(16, 3, padding = 'same', activation = 'relu', input_shape = (IMAGE_DIM_0,
model_2.add(MaxPooling2D())
model_2.add(Flatten())
model_2.add(Dense(DENSE_LAYER_WIDTH, activation = 'relu'))
model_2.add(Dense(CLASSES_N))
```

In [0]:

```
model_2.compile(optimizer = 'sgd',
                 loss = cat_crossentropy,
                 metrics = ['categorical_accuracy'])
```

In [26]:

```
model_2.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
=====		
conv2d_2 (Conv2D)	(None, 28, 28, 16)	160
=====		
max_pooling2d (MaxPooling2D)	(None, 14, 14, 16)	0
=====		
flatten_1 (Flatten)	(None, 3136)	0
=====		
dense_2 (Dense)	(None, 5000)	15685000
=====		
dense_3 (Dense)	(None, 10)	50010
=====		
Total params: 15,735,170		
Trainable params: 15,735,170		
Non-trainable params: 0		

In [27]:

```
model_2.fit(x = x, y = y, epochs = EPOCHS_N, validation_split = VAL_SPLIT_RATE)

Train on 374175 samples, validate on 41576 samples
Epoch 1/10
374175/374175 [=====] - 36s 97us/sample - loss: 0.5
802 - categorical_accuracy: 0.8338 - val_loss: 3.2642 - val_categorical_accuracy: 0.0382
Epoch 2/10
374175/374175 [=====] - 36s 96us/sample - loss: 0.4
135 - categorical_accuracy: 0.8762 - val_loss: 3.0404 - val_categorical_accuracy: 0.1671
Epoch 3/10
374175/374175 [=====] - 36s 97us/sample - loss: 0.3
677 - categorical_accuracy: 0.8891 - val_loss: 2.6523 - val_categorical_accuracy: 0.3309
Epoch 4/10
374175/374175 [=====] - 36s 95us/sample - loss: 0.3
384 - categorical_accuracy: 0.8975 - val_loss: 2.3369 - val_categorical_accuracy: 0.4434
Epoch 5/10
374175/374175 [=====] - 36s 96us/sample - loss: 0.3
159 - categorical_accuracy: 0.9037 - val_loss: 2.4419 - val_categorical_accuracy: 0.4272
Epoch 6/10
374175/374175 [=====] - 36s 95us/sample - loss: 0.2
965 - categorical_accuracy: 0.9099 - val_loss: 2.2025 - val_categorical_accuracy: 0.4449
Epoch 7/10
374175/374175 [=====] - 36s 96us/sample - loss: 0.2
797 - categorical_accuracy: 0.9147 - val_loss: 2.2130 - val_categorical_accuracy: 0.4772
Epoch 8/10
374175/374175 [=====] - 37s 98us/sample - loss: 0.2
644 - categorical_accuracy: 0.9193 - val_loss: 2.1537 - val_categorical_accuracy: 0.5218
Epoch 9/10
374175/374175 [=====] - 36s 95us/sample - loss: 0.2
509 - categorical_accuracy: 0.9233 - val_loss: 1.8914 - val_categorical_accuracy: 0.5983
Epoch 10/10
374175/374175 [=====] - 36s 96us/sample - loss: 0.2
375 - categorical_accuracy: 0.9274 - val_loss: 2.2922 - val_categorical_accuracy: 0.5274
```

Out[27]:

```
<tensorflow.python.keras.callbacks.History at 0x7f6ba828da58>
```

In [28]:

```
results_2 = model_2.evaluate(x_test, y_test)
print('Test loss, test accuracy:', results_2)
```

```
Test loss, test accuracy: [0.5017564680594646, 0.87370926]
```

Замена свёрточного слоя на операцию пулинга снизила точность на тестовой выборке до 87%.

Задание 3

Реализуйте классическую архитектуру сверточных сетей LeNet-5 (<http://yann.lecun.com/exdb/lenet/>) (<http://yann.lecun.com/exdb/lenet/>).

In [0]:

```
from tensorflow.keras.layers import AveragePooling2D

model_3 = tf.keras.Sequential()

model_3.add(Conv2D(6, kernel_size = (5, 5), strides = (1, 1), activation = 'tanh', padding
                  input_shape = (IMAGE_DIM_0, IMAGE_DIM_1, 1)))
model_3.add(AveragePooling2D(pool_size = (2, 2), strides = (2, 2), padding = 'valid'))
model_3.add(Conv2D(16, kernel_size = (5, 5), strides = (1, 1), activation = 'tanh', padding
                  input_shape = (IMAGE_DIM_0, IMAGE_DIM_1, 1)))
model_3.add(AveragePooling2D(pool_size = (2, 2), strides = (2, 2), padding = 'valid'))
model_3.add(Flatten())
model_3.add(Dense(120, activation = 'tanh'))
model_3.add(Dense(84, activation = 'tanh'))
model_3.add(Dense(CLASSES_N, activation = 'softmax'))
```

In [0]:

```
model_3.compile(optimizer = 'adam',
                 loss = 'categorical_crossentropy',
                 metrics = ['categorical_accuracy'])
```

In [31]:

```
model_3.fit(x = x, y = y, epochs = EPOCHS_N, validation_split = VAL_SPLIT_RATE)

Train on 374175 samples, validate on 41576 samples
Epoch 1/10
374175/374175 [=====] - 30s 79us/sample - loss: 0.4
504 - categorical_accuracy: 0.8640 - val_loss: 2.7422 - val_categorical_accuracy: 0.2516
Epoch 2/10
374175/374175 [=====] - 30s 80us/sample - loss: 0.3
514 - categorical_accuracy: 0.8912 - val_loss: 2.9686 - val_categorical_accuracy: 0.2343
Epoch 3/10
374175/374175 [=====] - 29s 79us/sample - loss: 0.3
239 - categorical_accuracy: 0.8995 - val_loss: 2.3722 - val_categorical_accuracy: 0.4675
Epoch 4/10
374175/374175 [=====] - 30s 79us/sample - loss: 0.3
086 - categorical_accuracy: 0.9037 - val_loss: 2.9093 - val_categorical_accuracy: 0.3062
Epoch 5/10
374175/374175 [=====] - 30s 79us/sample - loss: 0.2
972 - categorical_accuracy: 0.9069 - val_loss: 2.3950 - val_categorical_accuracy: 0.4489
Epoch 6/10
374175/374175 [=====] - 30s 79us/sample - loss: 0.2
891 - categorical_accuracy: 0.9092 - val_loss: 2.7761 - val_categorical_accuracy: 0.2896
Epoch 7/10
374175/374175 [=====] - 30s 80us/sample - loss: 0.2
811 - categorical_accuracy: 0.9117 - val_loss: 2.7471 - val_categorical_accuracy: 0.3655
Epoch 8/10
374175/374175 [=====] - 30s 80us/sample - loss: 0.2
749 - categorical_accuracy: 0.9136 - val_loss: 1.9750 - val_categorical_accuracy: 0.5368
Epoch 9/10
374175/374175 [=====] - 29s 79us/sample - loss: 0.2
707 - categorical_accuracy: 0.9148 - val_loss: 2.1891 - val_categorical_accuracy: 0.5343
Epoch 10/10
374175/374175 [=====] - 30s 80us/sample - loss: 0.2
659 - categorical_accuracy: 0.9163 - val_loss: 2.2132 - val_categorical_accuracy: 0.5457
```

Out[31]:

```
<tensorflow.python.keras.callbacks.History at 0x7f6ba7593048>
```

In [32]:

```
results_3 = model_3.evaluate(x_test, y_test)

print('Test loss, test accuracy:', results_3)
```

```
Test loss, test accuracy: [0.49347359862197515, 0.8702024]
```

Удивительно, но LeNet-5 показала результат хуже, чем первая модель — 87% на тестовой выборке.

Объяснить это можно тем, что первая модель содержит свёрточный слой с большей выходной размерностью.

Задание 4

Сравните максимальные точности моделей, построенных в лабораторных работах 1-3. Как можно объяснить полученные различия?

Результаты на валидационной выборке:

- логистическая регрессия — 81%;
- модель с только полно связанными слоями — 10%;
 - с регуляризацией и сбросом нейронов — 72%;
 - с адаптивным шагом — 72%;
- модель с двумя свёрточными слоями и одним полно связанным — 88%;
- модель с одним свёрточным слоем, операцией пулинга и одним полно связанным — 87%;
- *LeNet-5* — два свёрточных слоя две операции пулинга два полно связанных слоя — 87%.

Объяснение превосходства свёрточных сетей над полно связанными — такая архитектура просто предназначена для работы с изображениями.

Лабораторная работа №4

Реализация приложения по распознаванию номеров домов

Набор изображений из *Google Street View* с изображениями номеров домов, содержащий 10 классов, соответствующих цифрам от 0 до 9.

- 73257 изображений цифр в обучающей выборке;
- 26032 изображения цифр в тестовой выборке;
- 531131 изображения, которые можно использовать как дополнение к обучающей выборке;
- В двух форматах:
 - Оригинальные изображения с выделенными цифрами;
 - Изображения размером 32×32, содержащие одну цифру;
- Данные первого формата можно скачать по ссылкам:
 - <http://ufldl.stanford.edu/housenumbers/train.tar.gz> (<http://ufldl.stanford.edu/housenumbers/train.tar.gz>) (обучающая выборка);
 - <http://ufldl.stanford.edu/housenumbers/test.tar.gz> (<http://ufldl.stanford.edu/housenumbers/test.tar.gz>) (тестовая выборка);
 - <http://ufldl.stanford.edu/housenumbers/extrtar.gz> (<http://ufldl.stanford.edu/housenumbers/extrtar.gz>) (дополнительные данные);
- Данные второго формата можно скачать по ссылкам:
 - http://ufldl.stanford.edu/housenumbers/train_32x32.mat (http://ufldl.stanford.edu/housenumbers/train_32x32.mat) (обучающая выборка);
 - http://ufldl.stanford.edu/housenumbers/test_32x32.mat (http://ufldl.stanford.edu/housenumbers/test_32x32.mat) (тестовая выборка);
 - http://ufldl.stanford.edu/housenumbers/extr_32x32.mat (http://ufldl.stanford.edu/housenumbers/extr_32x32.mat) (дополнительные данные);
- Описание данных на английском языке доступно по ссылке:
 - <http://ufldl.stanford.edu/housenumbers/> (<http://ufldl.stanford.edu/housenumbers/>)

Задание 1

Реализуйте глубокую нейронную сеть (полносвязную или сверточную) и обучите ее на синтетических данных (например, наборы *MNIST* (<http://yann.lecun.com/exdb/mnist/>) (<http://yann.lecun.com/exdb/mnist/>)) или *notMNIST*).

Ознакомьтесь с имеющимися работами по данной тематике: англоязычная статья (<http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/42241.pdf> (<http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/42241.pdf>)), видео на *YouTube* (https://www.youtube.com/watch?v=vGPI_JvLoN0) (https://www.youtube.com/watch?v=vGPI_JvLoN0).

Используем архитектуру *LeNet-5* и обучим сеть сначала на данных из набора *MNIST*.

In [1]:

```
! pip install tensorflow-gpu --pre --quiet
```

```
! pip show tensorflow-gpu
```

```
Name: tensorflow-gpu
Version: 2.2.0rc3
Summary: TensorFlow is an open source machine learning framework for everyone.
Home-page: https://www.tensorflow.org/ (https://www.tensorflow.org/)
Author: Google Inc.
Author-email: packages@tensorflow.org
License: Apache 2.0
Location: /usr/local/lib/python3.6/dist-packages
Requires: grpcio, tensorboard, termcolor, six, gast, absl-py, tensorflow-estimator, opt-einsum, wrapt, wheel, numpy, google-pasta, protobuf, scipy, astunparse, keras-preprocessing, h5py
Required-by:
```

In [0]:

```
import tensorflow as tf
from tensorflow import keras
```

In [0]:

```
import numpy as np
```

In [0]:

```
from tensorflow.keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

In [0]:

```
x_train, x_test = tf.keras.utils.normalize(x_train, axis = 1), tf.keras.utils.normalize(x_t
```

In [0]:

```
x_train, x_test = x_train[..., np.newaxis], x_test[..., np.newaxis]
```

In [7]:

```
from tensorflow.keras.utils import to_categorical
y_train, y_test = to_categorical(y_train), to_categorical(y_test)
y_train.shape
```

Out[7]:

```
(60000, 10)
```

In [0]:

```
IMAGE_DIM_0, IMAGE_DIM_1 = x_train.shape[1], x_train.shape[2]
```

In [0]:

```
CLASSES_N = y_train.shape[1]
```

In [10]:

```
x_train.shape, x_test.shape
```

Out[10]:

```
((60000, 28, 28, 1), (10000, 28, 28, 1))
```

In [0]:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import AveragePooling2D, Conv2D, Dense, Flatten

model = tf.keras.Sequential()

model.add(Conv2D(6, kernel_size = (5, 5), strides = (1, 1), activation = 'tanh', padding =
               input_shape = (IMAGE_DIM_0, IMAGE_DIM_1, 1)))
model.add(AveragePooling2D(pool_size = (2, 2), strides = (2, 2), padding = 'valid'))
model.add(Conv2D(16, kernel_size = (5, 5), strides = (1, 1), activation = 'tanh', padding =
               input_shape = (16, 14, 14, 1)))
model.add(AveragePooling2D(pool_size = (2, 2), strides = (2, 2), padding = 'valid'))
model.add(Flatten())
model.add(Dense(120, activation = 'tanh'))
model.add(Dense(84, activation = 'tanh'))
model.add(Dense(CLASSES_N, activation = 'softmax'))
```

In [0]:

```
# 'sparse_categorical_crossentropy' gave NAN loss

model.compile(optimizer = 'adam',
              loss = 'categorical_crossentropy',
              metrics = ['categorical_accuracy'])
```

In [13]:

```
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
=====		
conv2d (Conv2D)	(None, 28, 28, 6)	156
=====		
average_pooling2d (AveragePo	(None, 14, 14, 6)	0
=====		
conv2d_1 (Conv2D)	(None, 10, 10, 16)	2416
=====		
average_pooling2d_1 (Average	(None, 5, 5, 16)	0
=====		
flatten (Flatten)	(None, 400)	0
=====		
dense (Dense)	(None, 120)	48120
=====		
dense_1 (Dense)	(None, 84)	10164
=====		
dense_2 (Dense)	(None, 10)	850
=====		
Total params:	61,706	
Trainable params:	61,706	
Non-trainable params:	0	

In [0]:

```
EPOCHS_N = 20
```

In [15]:

```
model.fit(x = x_train, y = y_train, validation_split = 0.15, epochs = EPOCHS_N)
```

Epoch 1/20
1594/1594 [=====] - 5s 3ms/step - loss: 0.2895 - categorical_accuracy: 0.9129 - val_loss: 0.1330 - val_categorical_accuracy: 0.9591
Epoch 2/20
1594/1594 [=====] - 5s 3ms/step - loss: 0.1186 - categorical_accuracy: 0.9639 - val_loss: 0.0995 - val_categorical_accuracy: 0.9694
Epoch 3/20
1594/1594 [=====] - 5s 3ms/step - loss: 0.0801 - categorical_accuracy: 0.9748 - val_loss: 0.0960 - val_categorical_accuracy: 0.9697
Epoch 4/20
1594/1594 [=====] - 5s 3ms/step - loss: 0.0631 - categorical_accuracy: 0.9799 - val_loss: 0.0737 - val_categorical_accuracy: 0.9786
Epoch 5/20
1594/1594 [=====] - 5s 3ms/step - loss: 0.0484 - categorical_accuracy: 0.9848 - val_loss: 0.0679 - val_categorical_accuracy: 0.9802
Epoch 6/20
1594/1594 [=====] - 5s 3ms/step - loss: 0.0394 - categorical_accuracy: 0.9874 - val_loss: 0.0647 - val_categorical_accuracy: 0.9806
Epoch 7/20
1594/1594 [=====] - 5s 3ms/step - loss: 0.0328 - categorical_accuracy: 0.9894 - val_loss: 0.0688 - val_categorical_accuracy: 0.9811
Epoch 8/20
1594/1594 [=====] - 5s 3ms/step - loss: 0.0268 - categorical_accuracy: 0.9919 - val_loss: 0.0698 - val_categorical_accuracy: 0.9806
Epoch 9/20
1594/1594 [=====] - 4s 3ms/step - loss: 0.0242 - categorical_accuracy: 0.9921 - val_loss: 0.0617 - val_categorical_accuracy: 0.9822
Epoch 10/20
1594/1594 [=====] - 5s 3ms/step - loss: 0.0197 - categorical_accuracy: 0.9935 - val_loss: 0.0687 - val_categorical_accuracy: 0.9822
Epoch 11/20
1594/1594 [=====] - 5s 3ms/step - loss: 0.0168 - categorical_accuracy: 0.9944 - val_loss: 0.0719 - val_categorical_accuracy: 0.9811
Epoch 12/20
1594/1594 [=====] - 4s 3ms/step - loss: 0.0155 - categorical_accuracy: 0.9948 - val_loss: 0.0680 - val_categorical_accuracy: 0.9808
Epoch 13/20
1594/1594 [=====] - 5s 3ms/step - loss: 0.0137 - categorical_accuracy: 0.9955 - val_loss: 0.0679 - val_categorical_accuracy: 0.9822
Epoch 14/20
1594/1594 [=====] - 5s 3ms/step - loss: 0.0121 - categorical_accuracy: 0.9965 - val_loss: 0.0768 - val_categorical_accuracy: 0.9806
Epoch 15/20

```
1594/1594 [=====] - 5s 3ms/step - loss: 0.0115 - categorical_accuracy: 0.9961 - val_loss: 0.0747 - val_categorical_accuracy: 0.9824
Epoch 16/20
1594/1594 [=====] - 5s 3ms/step - loss: 0.0122 - categorical_accuracy: 0.9958 - val_loss: 0.0687 - val_categorical_accuracy: 0.9840
Epoch 17/20
1594/1594 [=====] - 5s 3ms/step - loss: 0.0095 - categorical_accuracy: 0.9966 - val_loss: 0.0633 - val_categorical_accuracy: 0.9848
Epoch 18/20
1594/1594 [=====] - 5s 3ms/step - loss: 0.0100 - categorical_accuracy: 0.9965 - val_loss: 0.0701 - val_categorical_accuracy: 0.9836
Epoch 19/20
1594/1594 [=====] - 5s 3ms/step - loss: 0.0055 - categorical_accuracy: 0.9984 - val_loss: 0.0706 - val_categorical_accuracy: 0.9842
Epoch 20/20
1594/1594 [=====] - 5s 3ms/step - loss: 0.0093 - categorical_accuracy: 0.9967 - val_loss: 0.0714 - val_categorical_accuracy: 0.9838
```

Out[15]:

```
<tensorflow.python.keras.callbacks.History at 0x7f60025300f0>
```

In [16]:

```
results = model.evaluate(x_test, y_test)

print('Test loss, test accuracy:', results)
```

```
313/313 [=====] - 1s 2ms/step - loss: 0.0828 - categorical_accuracy: 0.9808
Test loss, test accuracy: [0.08280119299888611, 0.9807999730110168]
```

Удалось достичь отличного результата — точность распознавания на тестовой выборке составила 98,0%.

Задание 2

После уточнения модели на синтетических данных попробуйте обучить ее на реальных данных (набор *Google Street View*). Что изменилось в модели?

In [0]:

```
DS_URL_FOLDER = 'http://ufldl.stanford.edu/housenumbers/'

FIRST_DS_EXT = '.tar.gz'
SECOND_DS_EXT = '_32x32.mat'

TRAIN_DS_NAME = 'train'
TEST_DS_NAME = 'test'
EXTRA_DS_NAME = 'extra'
```

In [0]:

```
from urllib.request import urlretrieve
import tarfile
import os

def load_file(_url_folder, _name, _ext, _key, _local_ext = ''):

    file_url_ = _url_folder + _name + _ext

    local_file_name_ = _name + '_' + _key + _local_ext

    urlretrieve(file_url_, local_file_name_)

    return local_file_name_

def tar_gz_to_dir(_url_folder, _name, _ext, _key):

    local_file_name_ = load_file(_url_folder, _name, _ext, _key, _ext)

    dir_name_ = _name + '_' + _key

    with tarfile.open(local_file_name_, 'r:gz') as tar_:
        tar_.extractall(dir_name_)

    os.remove(local_file_name_)

    return dir_name_
```

In [0]:

```
first_ds_train_dir = tar_gz_to_dir(DS_URL_FOLDER, TRAIN_DS_NAME, FIRST_DS_EXT, 'first')
first_ds_test_dir = tar_gz_to_dir(DS_URL_FOLDER, TEST_DS_NAME, FIRST_DS_EXT, 'first')
first_ds_extra_dir = tar_gz_to_dir(DS_URL_FOLDER, EXTRA_DS_NAME, FIRST_DS_EXT, 'first')
```

In [0]:

```
second_ds_train_file = load_file(DS_URL_FOLDER, TRAIN_DS_NAME, SECOND_DS_EXT, 'second')
second_ds_test_file = load_file(DS_URL_FOLDER, TEST_DS_NAME, SECOND_DS_EXT, 'second')
second_ds_extra_file = load_file(DS_URL_FOLDER, EXTRA_DS_NAME, SECOND_DS_EXT, 'second')
```

In [0]:

```
from scipy import io

second_ds_train = io.loadmat(second_ds_train_file)
second_ds_test = io.loadmat(second_ds_test_file)
second_ds_extra = io.loadmat(second_ds_extra_file)
```

In [22]:

```
X_second_ds_train = np.moveaxis(second_ds_train['X'], -1, 0)
X_second_ds_test = np.moveaxis(second_ds_test['X'], -1, 0)
X_second_ds_extra = np.moveaxis(second_ds_extra['X'], -1, 0)

y_second_ds_train = second_ds_train['y']
y_second_ds_test = second_ds_test['y']
y_second_ds_extra = second_ds_extra['y']

print(X_second_ds_train.shape, y_second_ds_train.shape)
print(X_second_ds_test.shape, y_second_ds_test.shape)
print(X_second_ds_extra.shape, y_second_ds_extra.shape)
```

```
(73257, 32, 32, 3) (73257, 1)
(26032, 32, 32, 3) (26032, 1)
(531131, 32, 32, 3) (531131, 1)
```

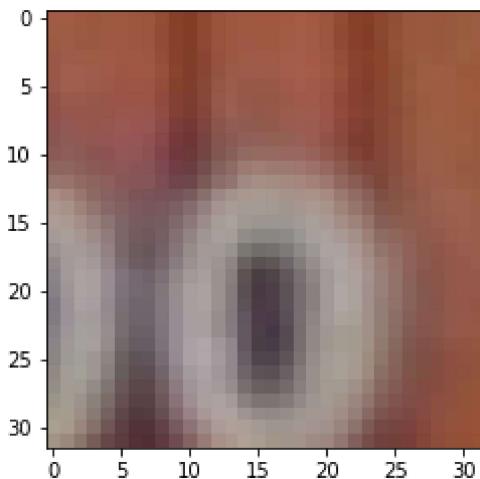
In [23]:

```
import matplotlib.pyplot as plt

plt.imshow(X_second_ds_train[100])
plt.imshow(X_second_ds_test[100])
plt.imshow(X_second_ds_extra[100])
```

Out[23]:

```
<matplotlib.image.AxesImage at 0x7f6002399320>
```



In [0]:

```
IMAGE_DIM_0_2, IMAGE_DIM_1_2, IMAGE_DIM_2_2 = X_second_ds_train.shape[-3], X_second_ds_train
```

In [0]:

```
y_second_ds_train_cat = to_categorical(y_second_ds_train)
y_second_ds_test_cat = to_categorical(y_second_ds_test)
```

In [0]:

```
CLASSES_N_2 = y_second_ds_train_cat.shape[1]
```

In [0]:

```
model_2 = tf.keras.Sequential()

model_2.add(Conv2D(6, kernel_size = (5, 5), strides = (1, 1), activation = 'tanh', padding
                  input_shape = (IMAGE_DIM_0_2, IMAGE_DIM_1_2, IMAGE_DIM_2_2)))
model_2.add(AveragePooling2D(pool_size = (2, 2), strides = (2, 2), padding = 'valid'))
model_2.add(Conv2D(16, kernel_size = (5, 5), strides = (1, 1), activation = 'tanh', padding
                  input_shape = (IMAGE_DIM_0_2, IMAGE_DIM_1_2, IMAGE_DIM_2_2)))
model_2.add(AveragePooling2D(pool_size = (2, 2), strides = (2, 2), padding = 'valid'))
model_2.add(Flatten())
model_2.add(Dense(120, activation = 'tanh'))
model_2.add(Dense(84, activation = 'tanh'))
model_2.add(Dense(CLASSES_N_2, activation = 'softmax'))
```

In [0]:

```
model_2.compile(optimizer = 'adam',
                 loss = 'categorical_crossentropy',
                 metrics = ['categorical_accuracy'])
```

In [29]:

```
model_2.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
<hr/>		
conv2d_2 (Conv2D)	(None, 32, 32, 6)	456
<hr/>		
average_pooling2d_2 (Average)	(None, 16, 16, 6)	0
<hr/>		
conv2d_3 (Conv2D)	(None, 12, 12, 16)	2416
<hr/>		
average_pooling2d_3 (Average)	(None, 6, 6, 16)	0
<hr/>		
flatten_1 (Flatten)	(None, 576)	0
<hr/>		
dense_3 (Dense)	(None, 120)	69240
<hr/>		
dense_4 (Dense)	(None, 84)	10164
<hr/>		
dense_5 (Dense)	(None, 11)	935
<hr/>		
Total params: 83,211		
Trainable params: 83,211		
Non-trainable params: 0		

In [30]:

```
model_2.fit(x = X_second_ds_train, y = y_second_ds_train_cat, validation_split = 0.15, epoch
```

Epoch 1/20
1946/1946 [=====] - 6s 3ms/step - loss: 1.1878 - categorical_accuracy: 0.6141 - val_loss: 0.8286 - val_categorical_accuracy: 0.7351
Epoch 2/20
1946/1946 [=====] - 5s 3ms/step - loss: 0.6917 - categorical_accuracy: 0.7843 - val_loss: 0.6128 - val_categorical_accuracy: 0.8132
Epoch 3/20
1946/1946 [=====] - 6s 3ms/step - loss: 0.5979 - categorical_accuracy: 0.8130 - val_loss: 0.5611 - val_categorical_accuracy: 0.8280
Epoch 4/20
1946/1946 [=====] - 6s 3ms/step - loss: 0.5475 - categorical_accuracy: 0.8287 - val_loss: 0.5569 - val_categorical_accuracy: 0.8276
Epoch 5/20
1946/1946 [=====] - 6s 3ms/step - loss: 0.5193 - categorical_accuracy: 0.8383 - val_loss: 0.5476 - val_categorical_accuracy: 0.8333
Epoch 6/20
1946/1946 [=====] - 5s 3ms/step - loss: 0.4886 - categorical_accuracy: 0.8474 - val_loss: 0.5300 - val_categorical_accuracy: 0.8375
Epoch 7/20
1946/1946 [=====] - 6s 3ms/step - loss: 0.4755 - categorical_accuracy: 0.8511 - val_loss: 0.5751 - val_categorical_accuracy: 0.8284
Epoch 8/20
1946/1946 [=====] - 6s 3ms/step - loss: 0.4482 - categorical_accuracy: 0.8613 - val_loss: 0.5043 - val_categorical_accuracy: 0.8469
Epoch 9/20
1946/1946 [=====] - 5s 3ms/step - loss: 0.4372 - categorical_accuracy: 0.8642 - val_loss: 0.5296 - val_categorical_accuracy: 0.8393
Epoch 10/20
1946/1946 [=====] - 5s 3ms/step - loss: 0.4162 - categorical_accuracy: 0.8697 - val_loss: 0.5304 - val_categorical_accuracy: 0.8381
Epoch 11/20
1946/1946 [=====] - 5s 3ms/step - loss: 0.4058 - categorical_accuracy: 0.8733 - val_loss: 0.5445 - val_categorical_accuracy: 0.8385
Epoch 12/20
1946/1946 [=====] - 5s 3ms/step - loss: 0.4027 - categorical_accuracy: 0.8753 - val_loss: 0.5371 - val_categorical_accuracy: 0.8347
Epoch 13/20
1946/1946 [=====] - 5s 3ms/step - loss: 0.3966 - categorical_accuracy: 0.8747 - val_loss: 0.5472 - val_categorical_accuracy: 0.8381
Epoch 14/20
1946/1946 [=====] - 5s 3ms/step - loss: 0.3922 - categorical_accuracy: 0.8759 - val_loss: 0.5105 - val_categorical_accuracy: 0.8482
Epoch 15/20

```
1946/1946 [=====] - 5s 3ms/step - loss: 0.3835 - categorical_accuracy: 0.8784 - val_loss: 0.5198 - val_categorical_accuracy: 0.8477
Epoch 16/20
1946/1946 [=====] - 5s 3ms/step - loss: 0.3669 - categorical_accuracy: 0.8850 - val_loss: 0.4903 - val_categorical_accuracy: 0.8523
Epoch 17/20
1946/1946 [=====] - 5s 3ms/step - loss: 0.3616 - categorical_accuracy: 0.8858 - val_loss: 0.5543 - val_categorical_accuracy: 0.8359
Epoch 18/20
1946/1946 [=====] - 6s 3ms/step - loss: 0.3559 - categorical_accuracy: 0.8873 - val_loss: 0.5141 - val_categorical_accuracy: 0.8467
Epoch 19/20
1946/1946 [=====] - 6s 3ms/step - loss: 0.3472 - categorical_accuracy: 0.8914 - val_loss: 0.5160 - val_categorical_accuracy: 0.8511
Epoch 20/20
1946/1946 [=====] - 5s 3ms/step - loss: 0.3439 - categorical_accuracy: 0.8917 - val_loss: 0.6758 - val_categorical_accuracy: 0.7925
```

Out[30]:

```
<tensorflow.python.keras.callbacks.History at 0x7f60022ca4e0>
```

In [31]:

```
results = model_2.evaluate(X_second_ds_test, y_second_ds_test_cat)
print('Test loss, test accuracy:', results)
```

```
814/814 [=====] - 1s 2ms/step - loss: 0.7207 - categorical_accuracy: 0.7746
Test loss, test accuracy: [0.7207155227661133, 0.7745851278305054]
```

Прежде всего, в модели изменилось то, что добавился ещё один класс — *не распознано*.

Эти данные более сложны для распознавания, что повлияло на результат — точность распознавания на тестовой выборке составила 77,4%.

Задание 3

Сделайте множество снимков изображений номеров домов с помощью смартфона на ОС *Android*. Также можно использовать библиотеки *OpenCV*, *Simple CV* или *Pygame* для обработки изображений с общедоступных камер видеонаблюдения (например, <https://www.earthcam.com/>) (<https://www.earthcam.com/>)).

В качестве примера использования библиотеки *TensorFlow* на смартфоне можете воспользоваться демонстрационным приложением от *Google* (<https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/android>) (<https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/android>)).

Задание 4

Реализуйте приложение для ОС *Android*, которое может распознавать цифры в номерах домов, используя разработанный ранее классификатор. Какова доля правильных классификаций?

Лабораторная работа №5

Применение сверточных нейронных сетей (бинарная классификация)

Набор данных *DogsVsCats*, который состоит из изображений различной размерности, содержащих фотографии собак и кошек.

Обучающая выборка включает в себя 25 тыс. изображений (12,5 тыс. кошек: *cat.0.jpg*, ..., *cat.12499.jpg* и 12,5 тыс. собак: *dog.0.jpg*, ..., *dog.12499.jpg*), а контрольная выборка содержит 12,5 тыс. неразмеченных изображений.

Скачать данные, а также проверить качество классификатора на тестовой выборке можно на сайте Kaggle: <https://www.kaggle.com/c/dogs-vs-cats/data> (<https://www.kaggle.com/c/dogs-vs-cats/data>)

Задание 1

Загрузите данные. Разделите исходный набор данных на обучающую, валидационную и контрольную выборки.

In [1]:

```
from google.colab import drive  
drive.mount('/content/drive', force_remount = True)
```

Mounted at /content/drive

In [0]:

```
BASE_DIR = '/content/drive/My Drive/Colab Files/mo-2/dogs-vs-cats'  
  
import sys  
  
sys.path.append(BASE_DIR)  
  
import os
```

In [0]:

```
TRAIN_ARCHIVE_NAME = 'train.zip'  
TEST_ARCHIVE_NAME = 'test1.zip'  
  
LOCAL_DIR_NAME = 'dogs-vs-cats'
```

In [0]:

```
from zipfile import ZipFile

with ZipFile(os.path.join(BASE_DIR, TRAIN_ARCHIVE_NAME), 'r') as zip_:
    zip_.extractall(path = os.path.join(LOCAL_DIR_NAME, 'train'))

with ZipFile(os.path.join(BASE_DIR, TEST_ARCHIVE_NAME), 'r') as zip_:
    zip_.extractall(path = os.path.join(LOCAL_DIR_NAME, 'test-1'))
```

In [5]:

```
from matplotlib import pyplot
from matplotlib.image import imread

pyplot.rcParams["figure.figsize"] = (10, 10)

dir_ = 'dogs-vs-cats/train/train'

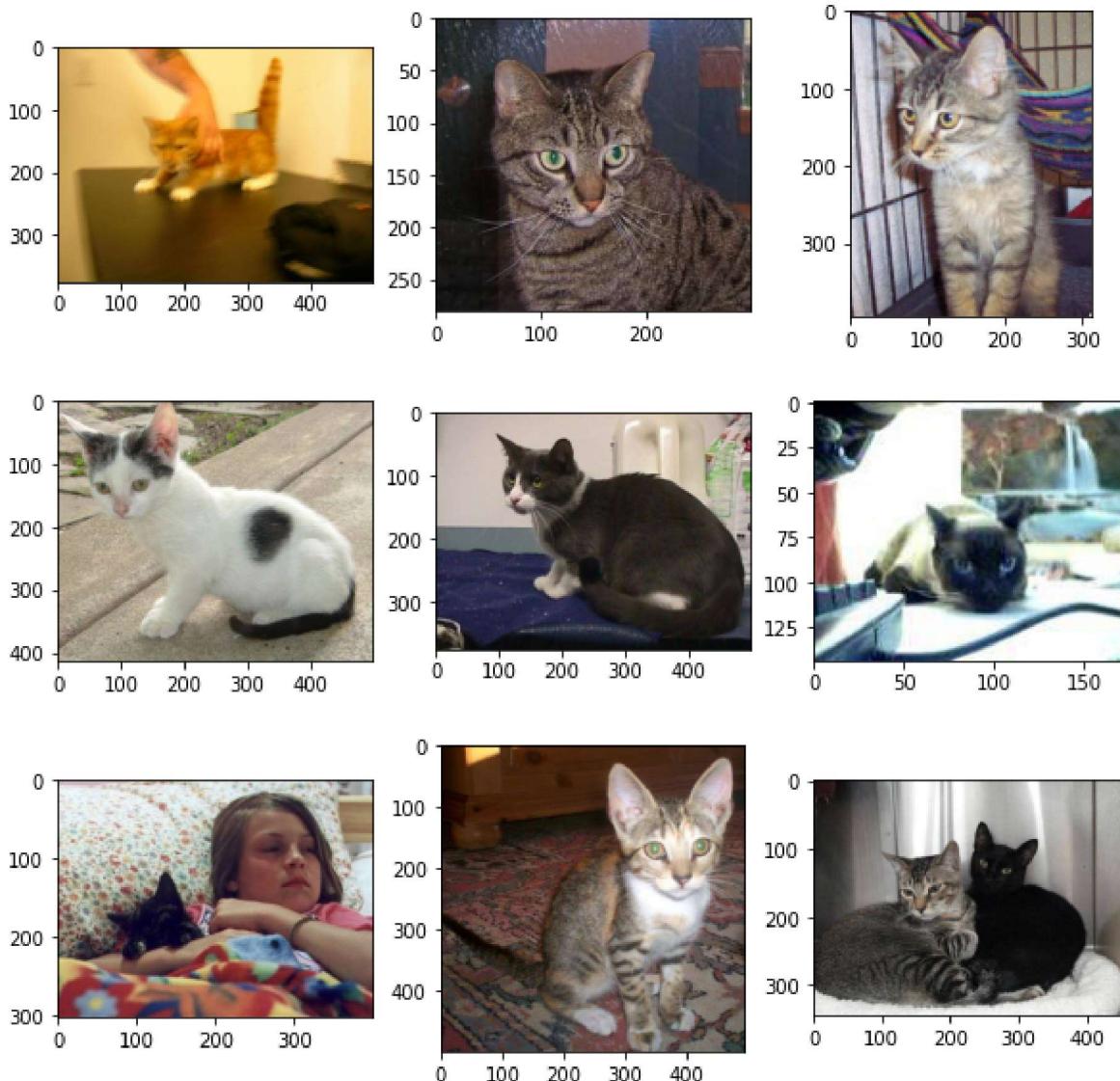
for i in range(9):

    pyplot.subplot(330 + 1 + i)

    image_ = imread('{}/cat.{}.jpg'.format(dir_, i))

    pyplot.imshow(image_)

pyplot.show()
```



Изображения необходимо привести к одному размеру.

In [0]:

```
NEW_IMAGE_WIDTH = 100
```

In [7]:

```
from os import listdir
from os.path import join
from numpy import asarray
from numpy import save
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array

def dir_to_dataset(_dir_path):

    photos_, labels_ = [], []
    for file_ in listdir(_dir_path):
        if file_.startswith('cat'):
            label_ = 1.0
        else:
            label_ = 0.0
        photo_ = load_img(join(_dir_path, file_), target_size = (NEW_IMAGE_WIDTH, NEW_IMAGE_HEIGHT))
        photo_ = img_to_array(photo_)
        photos_.append(photo_)
        labels_.append(label_)

    photos_norm_ = tf.keras.utils.normalize(photos_, axis = 1)
    return asarray(photos_norm_), asarray(labels_)
```

Using TensorFlow backend.

In [8]:

```
! pip install tensorflow-gpu --pre --quiet
```

```
! pip show tensorflow-gpu
```

```
Name: tensorflow-gpu
Version: 2.2.0rc3
Summary: TensorFlow is an open source machine learning framework for everyone.
Home-page: https://www.tensorflow.org/ (https://www.tensorflow.org/)
Author: Google Inc.
Author-email: packages@tensorflow.org
License: Apache 2.0
Location: /usr/local/lib/python3.6/dist-packages
Requires: six, opt-einsum, numpy, scipy, wheel, protobuf, google-pasta, astunparse, wrapt, termcolor, h5py, tensorboard, gast, tensorflow-estimator, absl-py, keras-preprocessing, grpcio
Required-by:
```

In [0]:

```
import tensorflow as tf
```

In [0]:

```
import numpy as np
```

In [0]:

```
X_all, y_all = dir_to_dataset('dogs-vs-cats/train/train')
```

In [0]:

```
TEST_LEN_HALF = 1000
```

In [13]:

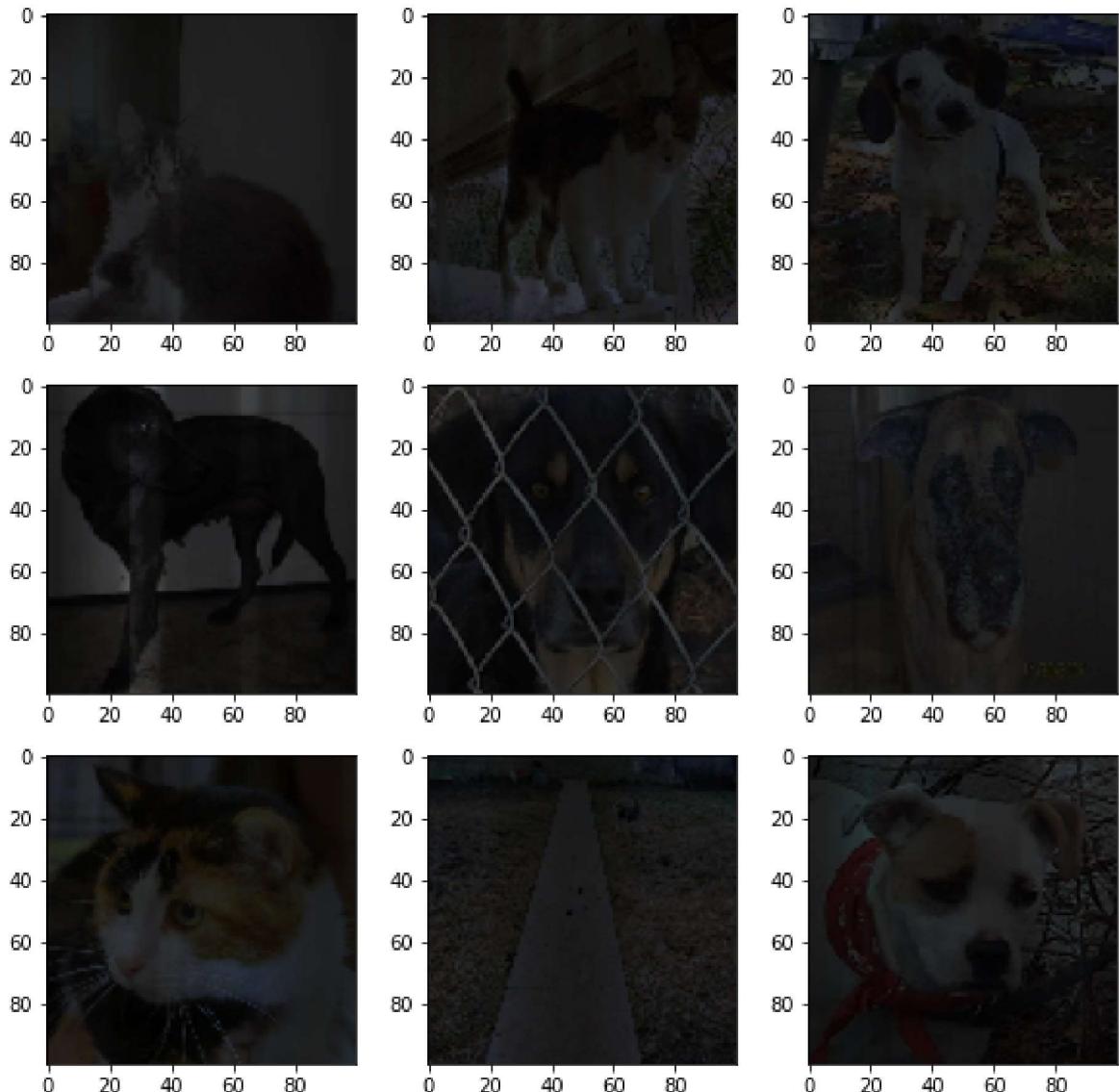
```
test_interval = np.r_[0:TEST_LEN_HALF, -TEST_LEN_HALF:-0]
X, y = X_all[TEST_LEN_HALF:-TEST_LEN_HALF], y_all[TEST_LEN_HALF:-TEST_LEN_HALF]
X_test, y_test = X_all[test_interval], y_all[test_interval]

print(X.shape, y.shape)
print(X_test.shape, y_test.shape)
```

```
(23000, 100, 100, 3) (23000,)
(2000, 100, 100, 3) (2000,)
```

In [14]:

```
for i in range(9):
    pyplot.subplot(330 + 1 + i)
    pyplot.imshow(X[i])
pyplot.show()
```



Выделение валидационной выборки произойдёт автоматически по параметру `validation_split` метода `model.fit()`.

Задание 2

Реализуйте глубокую нейронную сеть с как минимум тремя сверточными слоями. Какое качество классификации получено?

In [0]:

```
from tensorflow import keras
```

In [16]:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

model = tf.keras.Sequential()

model.add(Conv2D(16, 3, padding = 'same', activation = 'relu', input_shape = (NEW_IMAGE_WID
model.add(MaxPooling2D())
model.add(Conv2D(32, 3, padding = 'same', activation = 'relu'))
model.add(MaxPooling2D())
model.add(Conv2D(64, 3, padding = 'same', activation = 'relu'))
model.add(MaxPooling2D())
model.add(Flatten())
model.add(Dense(512, activation = 'relu'))
model.add(Dense(1, activation = 'sigmoid'))

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

model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
<hr/>		
conv2d (Conv2D)	(None, 100, 100, 16)	448
<hr/>		
max_pooling2d (MaxPooling2D)	(None, 50, 50, 16)	0
<hr/>		
conv2d_1 (Conv2D)	(None, 50, 50, 32)	4640
<hr/>		
max_pooling2d_1 (MaxPooling2	(None, 25, 25, 32)	0
<hr/>		
conv2d_2 (Conv2D)	(None, 25, 25, 64)	18496
<hr/>		
max_pooling2d_2 (MaxPooling2	(None, 12, 12, 64)	0
<hr/>		
flatten (Flatten)	(None, 9216)	0
<hr/>		
dense (Dense)	(None, 512)	4719104
<hr/>		
dense_1 (Dense)	(None, 1)	513
<hr/>		
Total params: 4,743,201		
Trainable params: 4,743,201		
Non-trainable params: 0		

In [17]:

```
model.fit(x = X, y = y, epochs = 20, validation_split = 0.15)
```

```
Epoch 1/20
611/611 [=====] - 5s 7ms/step - loss: 0.6910 - accuracy: 0.5330 - val_loss: 0.6884 - val_accuracy: 0.5693
Epoch 2/20
611/611 [=====] - 4s 7ms/step - loss: 0.6865 - accuracy: 0.5647 - val_loss: 0.6819 - val_accuracy: 0.6322
Epoch 3/20
611/611 [=====] - 4s 7ms/step - loss: 0.6805 - accuracy: 0.5809 - val_loss: 0.6717 - val_accuracy: 0.6336
Epoch 4/20
611/611 [=====] - 4s 7ms/step - loss: 0.6738 - accuracy: 0.5808 - val_loss: 0.6702 - val_accuracy: 0.5719
Epoch 5/20
611/611 [=====] - 4s 7ms/step - loss: 0.6655 - accuracy: 0.5966 - val_loss: 0.6482 - val_accuracy: 0.6475
Epoch 6/20
611/611 [=====] - 4s 7ms/step - loss: 0.6527 - accuracy: 0.6171 - val_loss: 0.6380 - val_accuracy: 0.6557
Epoch 7/20
611/611 [=====] - 4s 7ms/step - loss: 0.6378 - accuracy: 0.6395 - val_loss: 0.6373 - val_accuracy: 0.6383
Epoch 8/20
611/611 [=====] - 4s 7ms/step - loss: 0.6241 - accuracy: 0.6536 - val_loss: 0.6068 - val_accuracy: 0.6841
Epoch 9/20
611/611 [=====] - 4s 7ms/step - loss: 0.6128 - accuracy: 0.6647 - val_loss: 0.5983 - val_accuracy: 0.6896
Epoch 10/20
611/611 [=====] - 4s 7ms/step - loss: 0.6053 - accuracy: 0.6741 - val_loss: 0.6148 - val_accuracy: 0.6684
Epoch 11/20
611/611 [=====] - 4s 7ms/step - loss: 0.5972 - accuracy: 0.6816 - val_loss: 0.6089 - val_accuracy: 0.6713
Epoch 12/20
611/611 [=====] - 4s 7ms/step - loss: 0.5857 - accuracy: 0.6942 - val_loss: 0.5799 - val_accuracy: 0.7052
Epoch 13/20
611/611 [=====] - 4s 7ms/step - loss: 0.5761 - accuracy: 0.6972 - val_loss: 0.6049 - val_accuracy: 0.6748
Epoch 14/20
611/611 [=====] - 4s 7ms/step - loss: 0.5627 - accuracy: 0.7119 - val_loss: 0.5664 - val_accuracy: 0.7087
Epoch 15/20
611/611 [=====] - 4s 7ms/step - loss: 0.5504 - accuracy: 0.7217 - val_loss: 0.5417 - val_accuracy: 0.7391
Epoch 16/20
611/611 [=====] - 4s 7ms/step - loss: 0.5363 - accuracy: 0.7303 - val_loss: 0.5456 - val_accuracy: 0.7264
Epoch 17/20
611/611 [=====] - 4s 7ms/step - loss: 0.5231 - accuracy: 0.7423 - val_loss: 0.5623 - val_accuracy: 0.7006
Epoch 18/20
611/611 [=====] - 4s 7ms/step - loss: 0.5147 - accuracy: 0.7442 - val_loss: 0.5286 - val_accuracy: 0.7368
Epoch 19/20
611/611 [=====] - 4s 7ms/step - loss: 0.5017 - accuracy: 0.7559 - val_loss: 0.5169 - val_accuracy: 0.7475
```

```
Epoch 20/20
611/611 [=====] - 4s 7ms/step - loss: 0.4890 - accuracy: 0.7633 - val_loss: 0.5155 - val_accuracy: 0.7487
```

Out[17]:

```
<tensorflow.python.keras.callbacks.History at 0x7fa01a7c4358>
```

In [18]:

```
results = model.evaluate(X_test, y_test)

print('Test loss, test accuracy:', results)
```

```
63/63 [=====] - 0s 4ms/step - loss: 0.4963 - accuracy: 0.7505
Test loss, test accuracy: [0.4963347613811493, 0.7505000233650208]
```

Результат — 75% на тестовой выборке.

Задание 3

Примените дополнение данных (*data augmentation*). Как это повлияло на качество классификатора?

In [0]:

```
def augment_image(image):

    image = tf.image.convert_image_dtype(image, tf.float32)
    image = tf.image.resize_with_crop_or_pad(image, NEW_IMAGE_WIDTH + 40, NEW_IMAGE_WIDTH + 40)
    image = tf.image.random_crop(image, size = [NEW_IMAGE_WIDTH, NEW_IMAGE_WIDTH, 3])

    return image.numpy()
```

In [20]:

```
X_augmented = np.zeros_like(X)

for i, img in enumerate(X):
    X_augmented[i] = augment_image(img)

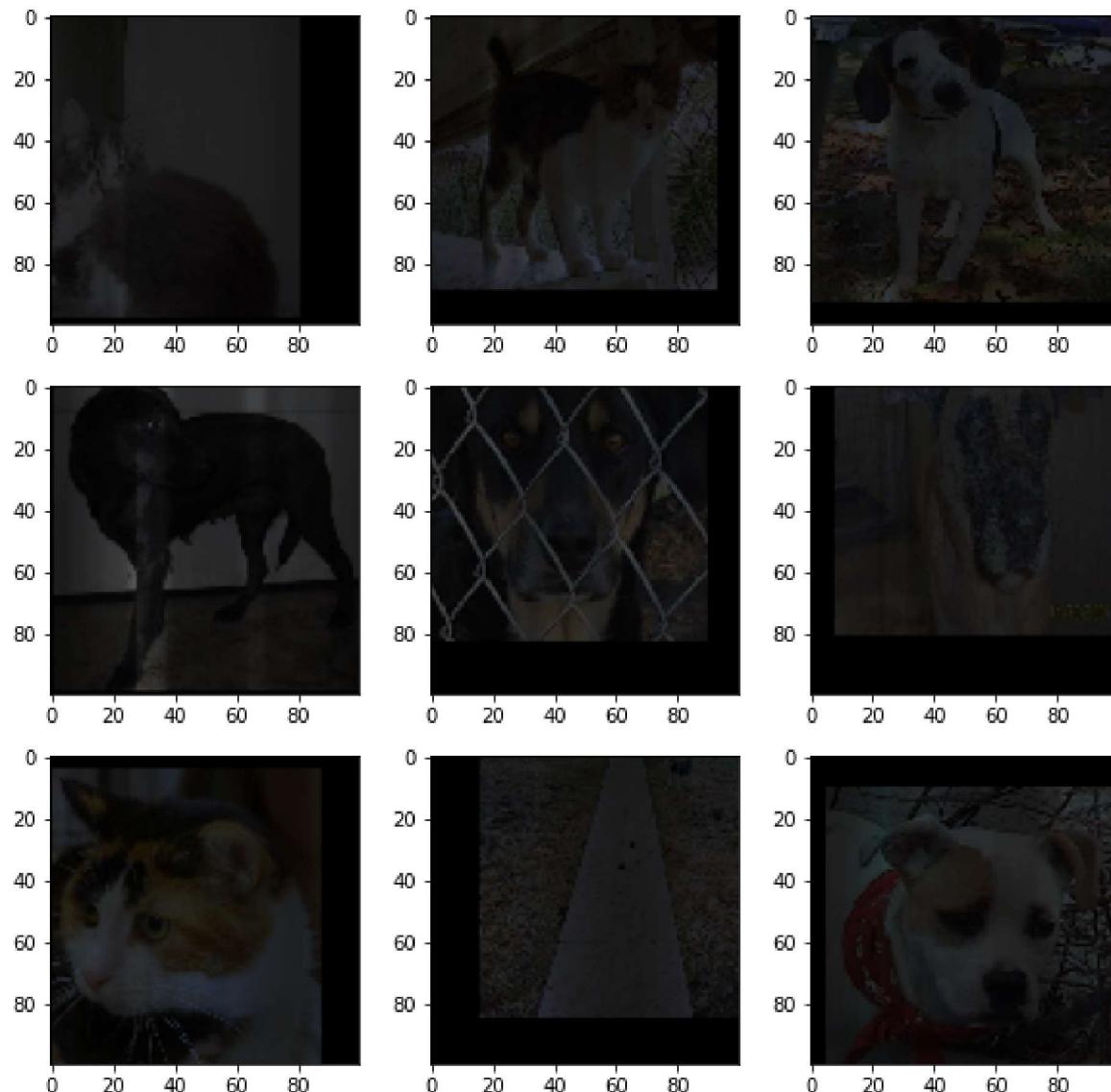
X_augmented.shape
```

Out[20]:

```
(23000, 100, 100, 3)
```

In [21]:

```
for i in range(9):
    pyplot.subplot(330 + 1 + i)
    pyplot.imshow(X_augmented[i])
pyplot.show()
```



In [0]:

```
y_augmented = y
```

In [23]:

```
model.fit(x = X_augmented, y = y_augmented, epochs = 20, validation_split = 0.15)
```

Epoch 1/20
611/611 [=====] - 4s 7ms/step - loss: 0.6068 - accuracy: 0.6690 - val_loss: 0.5740 - val_accuracy: 0.7000
Epoch 2/20
611/611 [=====] - 4s 7ms/step - loss: 0.5807 - accuracy: 0.6941 - val_loss: 0.5641 - val_accuracy: 0.7110
Epoch 3/20
611/611 [=====] - 4s 7ms/step - loss: 0.5697 - accuracy: 0.7006 - val_loss: 0.5608 - val_accuracy: 0.7157
Epoch 4/20
611/611 [=====] - 4s 7ms/step - loss: 0.5559 - accuracy: 0.7105 - val_loss: 0.5525 - val_accuracy: 0.7200
Epoch 5/20
611/611 [=====] - 4s 7ms/step - loss: 0.5443 - accuracy: 0.7204 - val_loss: 0.5469 - val_accuracy: 0.7212
Epoch 6/20
611/611 [=====] - 4s 7ms/step - loss: 0.5325 - accuracy: 0.7330 - val_loss: 0.5758 - val_accuracy: 0.6991
Epoch 7/20
611/611 [=====] - 4s 7ms/step - loss: 0.5196 - accuracy: 0.7426 - val_loss: 0.5517 - val_accuracy: 0.7067
Epoch 8/20
611/611 [=====] - 4s 7ms/step - loss: 0.5080 - accuracy: 0.7480 - val_loss: 0.5318 - val_accuracy: 0.7354
Epoch 9/20
611/611 [=====] - 4s 7ms/step - loss: 0.4939 - accuracy: 0.7575 - val_loss: 0.5328 - val_accuracy: 0.7278
Epoch 10/20
611/611 [=====] - 4s 7ms/step - loss: 0.4813 - accuracy: 0.7673 - val_loss: 0.5384 - val_accuracy: 0.7296
Epoch 11/20
611/611 [=====] - 4s 7ms/step - loss: 0.4676 - accuracy: 0.7761 - val_loss: 0.5519 - val_accuracy: 0.7133
Epoch 12/20
611/611 [=====] - 4s 7ms/step - loss: 0.4524 - accuracy: 0.7871 - val_loss: 0.5521 - val_accuracy: 0.7162
Epoch 13/20
611/611 [=====] - 4s 7ms/step - loss: 0.4373 - accuracy: 0.7955 - val_loss: 0.5264 - val_accuracy: 0.7406
Epoch 14/20
611/611 [=====] - 4s 7ms/step - loss: 0.4180 - accuracy: 0.8090 - val_loss: 0.5300 - val_accuracy: 0.7386
Epoch 15/20
611/611 [=====] - 4s 7ms/step - loss: 0.3987 - accuracy: 0.8180 - val_loss: 0.6854 - val_accuracy: 0.6788
Epoch 16/20
611/611 [=====] - 4s 7ms/step - loss: 0.3834 - accuracy: 0.8274 - val_loss: 0.5405 - val_accuracy: 0.7377
Epoch 17/20
611/611 [=====] - 4s 7ms/step - loss: 0.3588 - accuracy: 0.8413 - val_loss: 0.5376 - val_accuracy: 0.7400
Epoch 18/20
611/611 [=====] - 4s 7ms/step - loss: 0.3368 - accuracy: 0.8551 - val_loss: 0.5803 - val_accuracy: 0.7209
Epoch 19/20
611/611 [=====] - 4s 7ms/step - loss: 0.3151 - accuracy: 0.8673 - val_loss: 0.5872 - val_accuracy: 0.7287

```
Epoch 20/20
611/611 [=====] - 4s 7ms/step - loss: 0.2849 - accuracy: 0.8827 - val_loss: 0.5898 - val_accuracy: 0.7235
```

Out[23]:

```
<tensorflow.python.keras.callbacks.History at 0x7fa0e319d470>
```

In [24]:

```
results_2 = model.evaluate(X_test, y_test)
```

```
print('Test loss, test accuracy:', results_2)
```

```
63/63 [=====] - 0s 4ms/step - loss: 0.5276 - accuracy: 0.7670
```

```
Test loss, test accuracy: [0.5276018381118774, 0.7670000195503235]
```

После того, как сеть обучилась на тех же данных, к которым был применён data augmentation, точность предсказания на тестовой выборке увеличилась ненамного — до 76%.

Задание 4

Поэкспериментируйте с готовыми нейронными сетями (например, *AlexNet*, *VGG16*, *Inception* и т.п.), применив передаточное обучение. Как это повлияло на качество классификатора?

Какой максимальный результат удалось получить на сайте *Kaggle*? Почему?

Лабораторная работа №6

Применение сверточных нейронных сетей (многоклассовая классификация)

Набор данных для распознавания языка жестов, который состоит из изображений размерности 28x28 в оттенках серого (значение пикселя от 0 до 255).

Каждое из изображений обозначает букву латинского алфавита, обозначенную с помощью жеста (изображения в наборе данных в оттенках серого).

Обучающая выборка включает в себя 27,455 изображений, а контрольная выборка содержит 7172 изображения.

Данные в виде csv-файлов можно скачать на сайте Kaggle: <https://www.kaggle.com/datamunge/sign-language-mnist> (<https://www.kaggle.com/datamunge/sign-language-mnist>)

Задание 1

Загрузите данные. Разделите исходный набор данных на обучающую и валидационную выборки.

In [1]:

```
from google.colab import drive  
  
drive.mount('/content/drive', force_remount = True)
```

Mounted at /content/drive

In [0]:

```
BASE_DIR = '/content/drive/My Drive/Colab Files/mo-2'  
  
import sys  
  
sys.path.append(BASE_DIR)  
  
import os
```

In [0]:

```
DATA_ARCHIVE_NAME = 'sign-language-mnist.zip'  
  
LOCAL_DIR_NAME = 'sign-language'
```

In [0]:

```
from zipfile import ZipFile  
  
with ZipFile(os.path.join(BASE_DIR, DATA_ARCHIVE_NAME), 'r') as zip_:  
    zip_.extractall(path = os.path.join(LOCAL_DIR_NAME, 'train'))
```

In [0]:

```
TRAIN_FILE_PATH = 'sign-language/train/sign_mnist_train.csv'  
TEST_FILE_PATH = 'sign-language/train/sign_mnist_test.csv'
```

In [0]:

```
import pandas as pd  
  
train_df = pd.read_csv(TRAIN_FILE_PATH)  
test_df = pd.read_csv(TEST_FILE_PATH)
```

In [7]:

```
train_df.shape, test_df.shape
```

Out[7]:

```
((27455, 785), (7172, 785))
```

In [0]:

```
IMAGE_DIM = 28
```

In [0]:

```
def row_to_label(_row):  
    return _row[0]  
  
def row_to_one_image(_row):  
    return _row[1:].values.reshape((IMAGE_DIM, IMAGE_DIM, 1))
```

In [0]:

```
def to_images_and_labels(_dataframe):  
  
    llll = _dataframe.apply(lambda row: row_to_label(row), axis = 1)  
    mmmm = _dataframe.apply(lambda row: row_to_one_image(row), axis = 1)  
  
    data_dict_ = { 'label': llll, 'image': mmmm }  
  
    reshaped_ = pd.DataFrame(data_dict_, columns = ['label', 'image'])  
  
    return reshaped_
```

In [0]:

```
train_df_reshaped = to_images_and_labels(train_df)  
test_df_reshaped = to_images_and_labels(test_df)
```

Задание 2

Реализуйте глубокую нейронную сеть со сверточными слоями. Какое качество классификации получено? Какая архитектура сети была использована?

Возьмём LeNet-5.

In [12]:

```
! pip install tensorflow-gpu --pre --quiet
```

```
! pip show tensorflow-gpu
```

```
Name: tensorflow-gpu
Version: 2.2.0rc3
Summary: TensorFlow is an open source machine learning framework for everyone.
Home-page: https://www.tensorflow.org/ (https://www.tensorflow.org/)
Author: Google Inc.
Author-email: packages@tensorflow.org
License: Apache 2.0
Location: /usr/local/lib/python3.6/dist-packages
Requires: google-pasta, h5py, grpcio, opt-einsum, scipy, protobuf, wrapt, six, absl-py, tensorboard, wheel, gast, termcolor, numpy, tensorflow-estimator, keras-preprocessing, astunparse
Required-by:
```

In [0]:

```
import tensorflow as tf
```

In [0]:

```
from tensorflow.keras.utils import to_categorical
import numpy as np

X_train = tf.keras.utils.normalize(np.asarray(list(train_df_reshaped['image'])), axis = 1)
X_test = tf.keras.utils.normalize(np.asarray(list(test_df_reshaped['image'])), axis = 1)

y_train = to_categorical(train_df_reshaped['label'].astype('category').cat.codes.astype('int32'))
y_test = to_categorical(test_df_reshaped['label'].astype('category').cat.codes.astype('int32'))
```

In [15]:

```
X_train.shape, y_train.shape, X_test.shape, y_test.shape
```

Out[15]:

```
((27455, 28, 28, 1), (27455, 24), (7172, 28, 28, 1), (7172, 24))
```

In [0]:

```
CLASSES_N = y_train.shape[1]
```

In [0]:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import AveragePooling2D, Conv2D, Dense, Flatten

model = tf.keras.Sequential()

model.add(Conv2D(6, kernel_size = (5, 5), strides = (1, 1), activation = 'tanh', padding =
               input_shape = (IMAGE_DIM, IMAGE_DIM, 1)))
model.add(AveragePooling2D(pool_size = (2, 2), strides = (2, 2), padding = 'valid'))
model.add(Conv2D(16, kernel_size = (5, 5), strides = (1, 1), activation = 'tanh', padding =
               input_shape = (IMAGE_DIM, IMAGE_DIM, 1)))
model.add(AveragePooling2D(pool_size = (2, 2), strides = (2, 2), padding = 'valid'))
model.add(Flatten())
model.add(Dense(120, activation = 'tanh'))
model.add(Dense(84, activation = 'tanh'))
model.add(Dense(CLASSES_N, activation = 'softmax'))
```

In [0]:

```
model.compile(optimizer = 'adam',
              loss = 'categorical_crossentropy',
              metrics = ['categorical_accuracy'])
```

In [19]:

```
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
<hr/>		
conv2d (Conv2D)	(None, 28, 28, 6)	156
<hr/>		
average_pooling2d (AveragePo	(None, 14, 14, 6)	0
<hr/>		
conv2d_1 (Conv2D)	(None, 10, 10, 16)	2416
<hr/>		
average_pooling2d_1 (Average	(None, 5, 5, 16)	0
<hr/>		
flatten (Flatten)	(None, 400)	0
<hr/>		
dense (Dense)	(None, 120)	48120
<hr/>		
dense_1 (Dense)	(None, 84)	10164
<hr/>		
dense_2 (Dense)	(None, 24)	2040
<hr/>		
Total params: 62,896		
Trainable params: 62,896		
Non-trainable params: 0		

In [20]:

```
model.fit(x = X_train, y = y_train, epochs = 20, validation_split = 0.15)
```

Epoch 1/20

```
730/730 [=====] - 2s 3ms/step - loss: 1.2958 - categorical_accuracy: 0.6188 - val_loss: 0.5418 - val_categorical_accuracy: 0.8657
```

Epoch 2/20

```
730/730 [=====] - 2s 3ms/step - loss: 0.3189 - categorical_accuracy: 0.9325 - val_loss: 0.1453 - val_categorical_accuracy: 0.9830
```

Epoch 3/20

```
730/730 [=====] - 2s 3ms/step - loss: 0.0807 - categorical_accuracy: 0.9949 - val_loss: 0.0389 - val_categorical_accuracy: 0.9998
```

Epoch 4/20

```
730/730 [=====] - 2s 3ms/step - loss: 0.0262 - categorical_accuracy: 0.9997 - val_loss: 0.0160 - val_categorical_accuracy: 0.9998
```

Epoch 5/20

```
730/730 [=====] - 2s 3ms/step - loss: 0.0111 - categorical_accuracy: 1.0000 - val_loss: 0.0099 - val_categorical_accuracy: 0.9998
```

Epoch 6/20

```
730/730 [=====] - 2s 3ms/step - loss: 0.0058 - categorical_accuracy: 1.0000 - val_loss: 0.0044 - val_categorical_accuracy: 1.0000
```

Epoch 7/20

```
730/730 [=====] - 2s 3ms/step - loss: 0.0033 - categorical_accuracy: 1.0000 - val_loss: 0.0028 - val_categorical_accuracy: 1.0000
```

Epoch 8/20

```
730/730 [=====] - 2s 3ms/step - loss: 0.0145 - categorical_accuracy: 0.9964 - val_loss: 0.0043 - val_categorical_accuracy: 1.0000
```

Epoch 9/20

```
730/730 [=====] - 2s 3ms/step - loss: 0.0020 - categorical_accuracy: 1.0000 - val_loss: 0.0014 - val_categorical_accuracy: 1.0000
```

Epoch 10/20

```
730/730 [=====] - 2s 3ms/step - loss: 0.0011 - categorical_accuracy: 1.0000 - val_loss: 9.1311e-04 - val_categorical_accuracy: 1.0000
```

Epoch 11/20

```
730/730 [=====] - 2s 3ms/step - loss: 7.3560e-04 - categorical_accuracy: 1.0000 - val_loss: 6.3469e-04 - val_categorical_accuracy: 1.0000
```

Epoch 12/20

```
730/730 [=====] - 2s 3ms/step - loss: 5.1847e-04 - categorical_accuracy: 1.0000 - val_loss: 4.7602e-04 - val_categorical_accuracy: 1.0000
```

Epoch 13/20

```
730/730 [=====] - 2s 3ms/step - loss: 3.6735e-04 - categorical_accuracy: 1.0000 - val_loss: 3.2560e-04 - val_categorical_accuracy: 1.0000
```

Epoch 14/20

```
730/730 [=====] - 2s 3ms/step - loss: 2.6058e-04 - categorical_accuracy: 1.0000 - val_loss: 2.3029e-04 - val_categorical_accuracy: 1.0000
```

Epoch 15/20

```
730/730 [=====] - 2s 3ms/step - loss: 1.8485e-04 -  
categorical_accuracy: 1.0000 - val_loss: 1.6227e-04 - val_categorical_accuracy: 1.0000  
Epoch 16/20  
730/730 [=====] - 2s 3ms/step - loss: 1.2857e-04 -  
categorical_accuracy: 1.0000 - val_loss: 1.1164e-04 - val_categorical_accuracy: 1.0000  
Epoch 17/20  
730/730 [=====] - 2s 3ms/step - loss: 8.9278e-05 -  
categorical_accuracy: 1.0000 - val_loss: 7.7735e-05 - val_categorical_accuracy: 1.0000  
Epoch 18/20  
730/730 [=====] - 2s 3ms/step - loss: 6.2117e-05 -  
categorical_accuracy: 1.0000 - val_loss: 5.2588e-05 - val_categorical_accuracy: 1.0000  
Epoch 19/20  
730/730 [=====] - 2s 3ms/step - loss: 4.2049e-05 -  
categorical_accuracy: 1.0000 - val_loss: 3.7133e-05 - val_categorical_accuracy: 1.0000  
Epoch 20/20  
730/730 [=====] - 2s 3ms/step - loss: 2.8740e-05 -  
categorical_accuracy: 1.0000 - val_loss: 2.5821e-05 - val_categorical_accuracy: 1.0000
```

Out[20]:

```
<tensorflow.python.keras.callbacks.History at 0x7fd5b1433ba8>
```

In [21]:

```
results = model.evaluate(X_test, y_test)  
  
print('Test loss, test accuracy:', results)  
  
225/225 [=====] - 0s 2ms/step - loss: 0.8982 - categorical_accuracy: 0.8256  
Test loss, test accuracy: [0.8982465267181396, 0.8255716562271118]
```

За 20 эпох удалось достичь точности 82% на тестовой выборке.

Задание 3

Примените дополнение данных (*data augmentation*). Как это повлияло на качество классификатора?

In [0]:

```
def augment_image(image):  
  
    image = tf.image.convert_image_dtype(image, tf.float32)  
    image = tf.image.resize_with_crop_or_pad(image, IMAGE_DIM + 6, IMAGE_DIM + 6)  
    image = tf.image.random_crop(image, size = [IMAGE_DIM, IMAGE_DIM, 1])  
  
    return image.numpy()
```

In [23]:

```
X_train_augmented = np.zeros_like(X_train)

for i, img in enumerate(X_train):
    X_train_augmented[i] = augment_image(img)

X_train_augmented.shape
```

Out[23]:

```
(27455, 28, 28, 1)
```

In [0]:

```
y_train_augmented = y_train
```

In [25]:

```
model.fit(x = X_train_augmented, y = y_train_augmented, epochs = 20, validation_split = 0.1)
```

Epoch 1/20

```
730/730 [=====] - 2s 3ms/step - loss: 1.5178 - categorical_accuracy: 0.5917 - val_loss: 0.9175 - val_categorical_accuracy: 0.7242
```

Epoch 2/20

```
730/730 [=====] - 2s 3ms/step - loss: 0.7474 - categorical_accuracy: 0.7681 - val_loss: 0.6898 - val_categorical_accuracy: 0.7917
```

Epoch 3/20

```
730/730 [=====] - 2s 3ms/step - loss: 0.5514 - categorical_accuracy: 0.8275 - val_loss: 0.6034 - val_categorical_accuracy: 0.8126
```

Epoch 4/20

```
730/730 [=====] - 2s 3ms/step - loss: 0.4323 - categorical_accuracy: 0.8700 - val_loss: 0.5026 - val_categorical_accuracy: 0.8441
```

Epoch 5/20

```
730/730 [=====] - 2s 3ms/step - loss: 0.3474 - categorical_accuracy: 0.8959 - val_loss: 0.4365 - val_categorical_accuracy: 0.8667
```

Epoch 6/20

```
730/730 [=====] - 2s 3ms/step - loss: 0.2858 - categorical_accuracy: 0.9161 - val_loss: 0.3873 - val_categorical_accuracy: 0.8823
```

Epoch 7/20

```
730/730 [=====] - 2s 3ms/step - loss: 0.2393 - categorical_accuracy: 0.9299 - val_loss: 0.3671 - val_categorical_accuracy: 0.8866
```

Epoch 8/20

```
730/730 [=====] - 2s 3ms/step - loss: 0.1912 - categorical_accuracy: 0.9464 - val_loss: 0.3188 - val_categorical_accuracy: 0.9022
```

Epoch 9/20

```
730/730 [=====] - 2s 3ms/step - loss: 0.1655 - categorical_accuracy: 0.9549 - val_loss: 0.3211 - val_categorical_accuracy: 0.9000
```

Epoch 10/20

```
730/730 [=====] - 2s 3ms/step - loss: 0.1389 - categorical_accuracy: 0.9637 - val_loss: 0.2843 - val_categorical_accuracy: 0.9153
```

Epoch 11/20

```
730/730 [=====] - 2s 3ms/step - loss: 0.1186 - categorical_accuracy: 0.9684 - val_loss: 0.2725 - val_categorical_accuracy: 0.9145
```

Epoch 12/20

```
730/730 [=====] - 2s 3ms/step - loss: 0.0984 - categorical_accuracy: 0.9751 - val_loss: 0.2554 - val_categorical_accuracy: 0.9192
```

Epoch 13/20

```
730/730 [=====] - 2s 3ms/step - loss: 0.0848 - categorical_accuracy: 0.9790 - val_loss: 0.2307 - val_categorical_accuracy: 0.9272
```

Epoch 14/20

```
730/730 [=====] - 2s 3ms/step - loss: 0.0729 - categorical_accuracy: 0.9834 - val_loss: 0.2121 - val_categorical_accuracy: 0.9323
```

Epoch 15/20

```
730/730 [=====] - 2s 3ms/step - loss: 0.0724 - categorical_accuracy: 0.9811 - val_loss: 0.2220 - val_categorical_accuracy: 0.9272
Epoch 16/20
730/730 [=====] - 2s 3ms/step - loss: 0.0627 - categorical_accuracy: 0.9843 - val_loss: 0.2005 - val_categorical_accuracy: 0.9398
Epoch 17/20
730/730 [=====] - 2s 3ms/step - loss: 0.0499 - categorical_accuracy: 0.9891 - val_loss: 0.2056 - val_categorical_accuracy: 0.9378
Epoch 18/20
730/730 [=====] - 2s 3ms/step - loss: 0.0469 - categorical_accuracy: 0.9899 - val_loss: 0.2001 - val_categorical_accuracy: 0.9366
Epoch 19/20
730/730 [=====] - 2s 3ms/step - loss: 0.0539 - categorical_accuracy: 0.9859 - val_loss: 0.2164 - val_categorical_accuracy: 0.9366
Epoch 20/20
730/730 [=====] - 2s 3ms/step - loss: 0.0478 - categorical_accuracy: 0.9872 - val_loss: 0.1852 - val_categorical_accuracy: 0.9434
```

Out[25]:

```
<tensorflow.python.keras.callbacks.History at 0x7fd58874f5c0>
```

In [26]:

```
results_2 = model.evaluate(X_test, y_test)

print('Test loss, test accuracy:', results_2)
```

```
225/225 [=====] - 0s 2ms/step - loss: 0.2964 - categorical_accuracy: 0.9140
Test loss, test accuracy: [0.296428382396698, 0.9139710068702698]
```

После того, как сеть обучилась на тех же данных, к которым был применён *data augmentation*, точность предсказания на тестовой выборке увеличилась до 91%.

Задание 4

Поэкспериментируйте с готовыми нейронными сетями (например, *AlexNet*, *VGG16*, *Inception* и т.п.), применив передаточное обучение. Как это повлияло на качество классификатора? Можно ли было обойтись без него?

Какой максимальный результат удалось получить на контрольной выборке?

Лабораторная работа №7

Рекуррентные нейронные сети для анализа текста

Набор данных для предсказания оценок для отзывов, собранных с сайта *imdb.com*, который состоит из 50,000 отзывов в виде текстовых файлов.

Отзывы разделены на положительные (25,000) и отрицательные (25,000).

Данные предварительно токенизированы по принципу «мешка слов», индексы слов можно взять из словаря (*imdb.vocab*).

Обучающая выборка включает в себя 12,500 положительных и 12,500 отрицательных отзывов, контрольная выборка также содержит 12,500 положительных и 12,500 отрицательных отзывов.

Данные можно скачать на сайте Kaggle: <https://www.kaggle.com/iarunava/imdb-movie-reviews-dataset> (<https://www.kaggle.com/iarunava/imdb-movie-reviews-dataset>) <https://ai.stanford.edu/~amaas/data/sentiment/> (<https://ai.stanford.edu/~amaas/data/sentiment/>)

Задание 1

Загрузите данные. Преобразуйте текстовые файлы во внутренние структуры данных, которые используют индексы вместо слов.

Будем брать первые MAX_LENGTH слов, а если в отзыве слов меньше, чем это число, то применять паддинг.

In [1]:

```
from google.colab import drive  
  
drive.mount('/content/drive', force_remount = True)
```

Go to this URL in a browser: [https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redirect_uri=urn%3aietf%3awg%3aoauth%3a2.0%3aoob&response_type=code&scope=email%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdocs.test%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive.photos.readonly \(https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redirect_uri=urn%3aietf%3awg%3aoauth%3a2.0%3aoob&response_type=code&scope=email%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdocs.test%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive.photos.readonly%20https%3a%2f%2fwww.googleapis.com%2fauth%2fpeopleapi.readonly\)](https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redirect_uri=urn%3aietf%3awg%3aoauth%3a2.0%3aoob&response_type=code&scope=email%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdocs.test%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive.photos.readonly%20https%3a%2f%2fwww.googleapis.com%2fauth%2fpeopleapi.readonly (https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redirect_uri=urn%3aietf%3awg%3aoauth%3a2.0%3aoob&response_type=code&scope=email%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdocs.test%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive.photos.readonly%20https%3a%2f%2fwww.googleapis.com%2fauth%2fpeopleapi.readonly))

Enter your authorization code:

.....

Mounted at /content/drive

In [0]:

```
BASE_DIR = '/content/drive/My Drive/Colab Files/mo-2'

import sys

sys.path.append(BASE_DIR)

import os
```

In [0]:

```
DATA_ARCHIVE_NAME = 'imdb-dataset-of-50k-movie-reviews.zip'

LOCAL_DIR_NAME = 'imdb-sentiments'
```

In [0]:

```
from zipfile import ZipFile

with ZipFile(os.path.join(BASE_DIR, DATA_ARCHIVE_NAME), 'r') as zip_:
    zip_.extractall(LOCAL_DIR_NAME)
```

In [0]:

```
DATA_FILE_PATH = 'imdb-sentiments/IMDB Dataset.csv'
```

In [0]:

```
import pandas as pd

all_df = pd.read_csv(DATA_FILE_PATH)
```

In [0]:

```
df_test = all_df.sample(frac = 0.1)

df_train = all_df.drop(df_test.index)
```

In [8]:

```
df_train.shape, df_test.shape
```

Out[8]:

```
((45000, 2), (5000, 2))
```

In [9]:

```
import nltk  
  
nltk.download('punkt')
```

[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data] Unzipping tokenizers/punkt.zip.

Out[9]:

True

In [0]:

```
MAX_LENGTH = 40  
  
STRING_DTYPE = '<U12'  
  
PADDING_TOKEN = 'PAD'  
  
LIMIT_OF_TOKENS = 100000
```

In [0]:

```
from nltk import word_tokenize  
import numpy as np  
import string  
import re  
  
def tokenize_string(_string):  
    return [tok_.lower() for tok_ in word_tokenize(_string) if not re.fullmatch('[' + stri  
  
def pad(A, length):  
    arr = np.empty(length, dtype = STRING_DTYPE)  
    arr.fill(PADDING_TOKEN)  
    arr[:len(A)] = A  
    return arr  
  
def tokenize_row(_sentence):  
    return pad(tokenize_string(_sentence))[:MAX_LENGTH], MAX_LENGTH  
  
def encode_row(_label):  
    return 1 if _label == 'positive' else 0  
  
def encode_and_tokenize(_dataframe):  
  
    tttt = _dataframe.apply(lambda row: tokenize_row(row['review']), axis = 1)  
    llll = _dataframe.apply(lambda row: encode_row(row['sentiment']), axis = 1)  
  
    data_dict_ = { 'label': llll, 'tokens': tttt }  
  
    encoded_and_tokenized_ = pd.DataFrame(data_dict_, columns = ['label', 'tokens'])  
  
    return encoded_and_tokenized_
```

In [0]:

```
df_train_tokenized = encode_and_tokenize(df_train)
df_test_tokenized = encode_and_tokenize(df_test)
```

In [0]:

```
from collections import Counter

def get_tokens_list(_dataframe):
    all_tokens_ = []
    for sent_ in _dataframe['tokens'].values:
        all_tokens_.extend(sent_)
    tokens_counter_ = Counter(all_tokens_)
    return [t for t, _ in tokens_counter_.most_common(LIMIT_OF_TOKENS)]
```

In [0]:

```
tokens_list = get_tokens_list(pd.concat([df_train_tokenized, df_test_tokenized]))
```

In [0]:

```
word_to_int_dict = {}
word_to_int_dict.update(
    {t : i for i, t in enumerate(tokens_list)})
```

In [0]:

```
def intize_row(_tokens):
    return np.array([word_to_int_dict[t]
                    if t in word_to_int_dict
                    else 0
                    for t in _tokens])

def encode_and_tokenize(_dataframe):
    iii = _dataframe.apply(lambda row: intize_row(row['tokens']), axis = 1)
    data_dict_ = { 'label': _dataframe['label'], 'ints': iii }
    intized_ = pd.DataFrame(data_dict_, columns = ['label', 'ints'])
    return intized_
```

In [0]:

```
df_train_intized = encode_and_tokenize(df_train_tokenized)
df_test_intized = encode_and_tokenize(df_test_tokenized)
```

Задание 2

Реализуйте и обучите двунаправленную рекуррентную сеть (*LSTM* или *GRU*).

Какого качества классификации удалось достичь?

In [18]:

```
! pip install tensorflow-gpu --pre --quiet
```

```
! pip show tensorflow-gpu
```

```
|████████| 516.2MB 30kB/s
Name: tensorflow-gpu
Version: 2.2.0rc3
Summary: TensorFlow is an open source machine learning framework for everyone.
Home-page: https://www.tensorflow.org/ (https://www.tensorflow.org/)
Author: Google Inc.
Author-email: packages@tensorflow.org
License: Apache 2.0
Location: /usr/local/lib/python3.6/dist-packages
Requires: keras-preprocessing, wrapt, gast, grpcio, astunparse, h5py, termcolor, tensorflow-estimator, six, google-pasta, absl-py, opt-einsum, wheel, numpy, protobuf, scipy, tensorboard
Required-by:
```

In [0]:

```
import tensorflow as tf
from tensorflow import keras
```

In [0]:

```
# To fix memory leak: https://github.com/tensorflow/tensorflow/issues/33009
tf.compat.v1.disable_eager_execution()
```

Здесь будем использовать такую конфигурацию рекуррентного *LSTM*-слоя, которая позволит использовать очень быструю *cuDNN* имплементацию.

In [21]:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Bidirectional, LSTM, Dense

# The requirements to use the cuDNN implementation are:
# 1. `activation` == `tanh`
# 2. `recurrent_activation` == `sigmoid`
# 3. `recurrent_dropout` == 0
# 4. `unroll` is `False`
# 5. `use_bias` is `True`
# 6. `reset_after` is `True`
# 7. Inputs, if use masking, are strictly right-padded.

model = tf.keras.Sequential()

model.add(Bidirectional(LSTM(100, return_sequences = False), merge_mode = 'concat',
           input_shape = (MAX_LENGTH, 1)))
model.add(Dense(1, activation = 'sigmoid'))
```

WARNING:tensorflow:Layer lstm will not use cuDNN kernel since it doesn't meet the cudnn kernel criteria. It will use generic GPU kernel as fallback when running on GPU

WARNING:tensorflow:Layer lstm will not use cuDNN kernel since it doesn't meet the cudnn kernel criteria. It will use generic GPU kernel as fallback when running on GPU

WARNING:tensorflow:Layer lstm will not use cuDNN kernel since it doesn't meet the cudnn kernel criteria. It will use generic GPU kernel as fallback when running on GPU

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow/python/ops/resource_variable_ops.py:1666: calling BaseResourceVariable.__init__(from tensorflow.python.ops.resource_variable_ops) with constraint is deprecated and will be removed in a future version.

Instructions for updating:

If using Keras pass *_constraint arguments to layers.

In [22]:

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

model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
=====		
bidirectional (Bidirectional (None, 200))		81600
=====		
dense (Dense)	(None, 1)	201
=====		
Total params: 81,801		
Trainable params: 81,801		
Non-trainable params: 0		

In [0]:

```
X_train_intized = np.asarray(list(df_train_intized['ints'].values), dtype = float)[..., np.newaxis]
X_test_intized = np.asarray(list(df_test_intized['ints'].values), dtype = float)[..., np.newaxis]

y_train_intized = np.asarray(list(df_train_intized['label'].values))
y_test_intized = np.asarray(list(df_test_intized['label'].values))
```

In [24]:

```
model.fit(x = X_train_intized, y = y_train_intized, validation_split = 0.15, epochs = 20)

Train on 38250 samples, validate on 6750 samples
Epoch 1/20
38250/38250 [=====] - 51s 1ms/sample - loss: 0.6936
- accuracy: 0.5191 - val_loss: 0.6901 - val_accuracy: 0.5313
Epoch 2/20
38250/38250 [=====] - 51s 1ms/sample - loss: 0.6884
- accuracy: 0.5418 - val_loss: 0.6862 - val_accuracy: 0.5560
Epoch 3/20
38250/38250 [=====] - 50s 1ms/sample - loss: 0.6853
- accuracy: 0.5501 - val_loss: 0.6826 - val_accuracy: 0.5612
Epoch 4/20
38250/38250 [=====] - 51s 1ms/sample - loss: 0.6828
- accuracy: 0.5585 - val_loss: 0.6889 - val_accuracy: 0.5498
Epoch 5/20
38250/38250 [=====] - 50s 1ms/sample - loss: 0.6808
- accuracy: 0.5618 - val_loss: 0.6818 - val_accuracy: 0.5630
Epoch 6/20
38250/38250 [=====] - 50s 1ms/sample - loss: 0.6787
- accuracy: 0.5702 - val_loss: 0.6788 - val_accuracy: 0.5695
Epoch 7/20
38250/38250 [=====] - 49s 1ms/sample - loss: 0.6768
- accuracy: 0.5738 - val_loss: 0.6825 - val_accuracy: 0.5637
Epoch 8/20
38250/38250 [=====] - 50s 1ms/sample - loss: 0.6742
- accuracy: 0.5739 - val_loss: 0.6775 - val_accuracy: 0.5720
Epoch 9/20
38250/38250 [=====] - 49s 1ms/sample - loss: 0.6724
- accuracy: 0.5778 - val_loss: 0.6807 - val_accuracy: 0.5615
Epoch 10/20
38250/38250 [=====] - 50s 1ms/sample - loss: 0.6701
- accuracy: 0.5816 - val_loss: 0.6777 - val_accuracy: 0.5693
Epoch 11/20
38250/38250 [=====] - 49s 1ms/sample - loss: 0.6676
- accuracy: 0.5858 - val_loss: 0.6787 - val_accuracy: 0.5637
Epoch 12/20
38250/38250 [=====] - 49s 1ms/sample - loss: 0.6654
- accuracy: 0.5880 - val_loss: 0.6768 - val_accuracy: 0.5680
Epoch 13/20
38250/38250 [=====] - 49s 1ms/sample - loss: 0.6621
- accuracy: 0.5933 - val_loss: 0.6726 - val_accuracy: 0.5766
Epoch 14/20
38250/38250 [=====] - 49s 1ms/sample - loss: 0.6598
- accuracy: 0.5970 - val_loss: 0.6755 - val_accuracy: 0.5701
Epoch 15/20
38250/38250 [=====] - 49s 1ms/sample - loss: 0.6571
- accuracy: 0.5987 - val_loss: 0.6728 - val_accuracy: 0.5692
Epoch 16/20
38250/38250 [=====] - 49s 1ms/sample - loss: 0.6542
- accuracy: 0.6018 - val_loss: 0.6747 - val_accuracy: 0.5661
Epoch 17/20
38250/38250 [=====] - 50s 1ms/sample - loss: 0.6516
- accuracy: 0.6045 - val_loss: 0.6726 - val_accuracy: 0.5736
Epoch 18/20
38250/38250 [=====] - 49s 1ms/sample - loss: 0.6485
- accuracy: 0.6095 - val_loss: 0.6807 - val_accuracy: 0.5759
Epoch 19/20
38250/38250 [=====] - 49s 1ms/sample - loss: 0.6467
```

```
- accuracy: 0.6125 - val_loss: 0.6771 - val_accuracy: 0.5714
Epoch 20/20
38250/38250 [=====] - 49s 1ms/sample - loss: 0.6438
- accuracy: 0.6130 - val_loss: 0.6781 - val_accuracy: 0.5658
```

Out[24]:

```
<tensorflow.python.keras.callbacks.History at 0x7f8fd8a8b208>
```

In [25]:

```
results = model.evaluate(X_test_intized, y_test_intized)
print('Test loss, test accuracy:', results)
```

```
Test loss, test accuracy: [0.6761944528579712, 0.5762]
```

На валидационной выборке удалось достичь точности 57%.

Задание 3

Используйте индексы слов и их различное внутреннее представление (*word2vec, glove*). Как влияет данное преобразование на качество классификации?

Используем 300-мерные вектора *FastText* — лучшую на сегодняшний день имплементацию word2vec: <https://fasttext.cc/docs/en/english-vectors.html> (<https://fasttext.cc/docs/en/english-vectors.html>). Файл пришлось доработать — 9-я строка не читалась.

In [0]:

```
# VECTORS_ARCHIVE_NAME = 'wiki-news-300d-1M-fixed.zip'
# VECTORS_FILE_NAME = 'wiki-news-300d-1M-fixed.vec'
# VECTORS_LOCAL_DIR_NAME = 'vectors'
```

In [0]:

```
# with ZipFile(os.path.join(BASE_DIR, VECTORS_ARCHIVE_NAME), 'r') as zip_:
#     zip_.extractall(VECTORS_LOCAL_DIR_NAME)
```

Создадим уменьшенный словарь, содержащий только встреченные токены, чтобы уменьшить нагрузку на Google Drive:

In [0]:

```
# def build_vectors_dict(_actual_tokens, _vectors_file_path, _unknown_token = 'unknown'):
#     vec_data_ = pd.read_csv(_vectors_file_path, sep = ' ', header = None, skiprows = [9])
#     actual_vectors_ = [x for x in vec_data_.values if x[0] in _actual_tokens or x[0] == _unknown_token]
#     return actual_vectors_
```

In [0]:

```
# actual_vectors = build_vectors_dict(tokens_list, os.path.join(VECTORS_LOCAL_DIR_NAME, VEC
```

In [0]:

```
# vectors_np = np.array(actual_vectors)

# vectors_dict = dict(zip(vectors_np[:, 0], vectors_np[:, 1:]))

# vectors_dict_file_name = 'word-vec-dict-{}-items'.format(len(vectors_dict))

# vectors_dict_file_path = os.path.join(BASE_DIR, vectors_dict_file_name)

# np.savez_compressed(vectors_dict_file_path, vectors_dict, allow_pickle = True)
```

In [0]:

```
vectors_dict_file_path = './drive/My Drive/Colab Files/mo-2/word-vec-dict-56485-items.npz'
```

In [0]:

```
vectors_dict_data = np.load(vectors_dict_file_path, allow_pickle = True)

vectors_dict = vectors_dict_data['arr_0'][():]
```

In [0]:

```
VECTORS_LENGTH = 300
```

In [0]:

```
def tokens_to_vectors(_word_to_vec_dict, _tokens, _unknown_token):
    return [_word_to_vec_dict[t]
            if t in _word_to_vec_dict
            else _word_to_vec_dict[_unknown_token]
            for t in _tokens]

def row_to_vectors(_tokens):
    return np.array(tokens_to_vectors(vectors_dict, _tokens, 'unknown'))

def vectorize(_dataframe):

    vvvv = _dataframe.apply(lambda row: row_to_vectors(row['tokens']), axis = 1)

    data_dict_ = { 'label': _dataframe['label'], 'vectors': vvvv }

    vectorized_ = pd.DataFrame(data_dict_, columns = ['label', 'vectors'])

    return vectorized_
```

In [0]:

```
df_train_vectorized = vectorize(df_train_tokenized)
df_test_vectorized = vectorize(df_test_tokenized)
```

In [0]:

```
X_train_vectorized = np.asarray(list(df_train_vectorized['vectors'].values), dtype = float)
X_test_vectorized = np.asarray(list(df_test_vectorized['vectors'].values), dtype = float)

y_train_vectorized = np.asarray(list(df_train_vectorized['label'].values))
y_test_vectorized = np.asarray(list(df_test_vectorized['label'].values))
```

In [37]:

```
model_2 = tf.keras.Sequential()

model_2.add(Bidirectional(LSTM(100, return_sequences = False), merge_mode = 'concat',
                         input_shape = (MAX_LENGTH, VECTORS_LENGTH)))
model_2.add(Dense(1, activation = 'sigmoid'))
```

WARNING:tensorflow:Layer lstm_1 will not use cuDNN kernel since it doesn't meet the cuDNN kernel criteria. It will use generic GPU kernel as fallback when running on GPU

WARNING:tensorflow:Layer lstm_1 will not use cuDNN kernel since it doesn't meet the cuDNN kernel criteria. It will use generic GPU kernel as fallback when running on GPU

WARNING:tensorflow:Layer lstm_1 will not use cuDNN kernel since it doesn't meet the cuDNN kernel criteria. It will use generic GPU kernel as fallback when running on GPU

In [38]:

```
model_2.compile(optimizer = 'adam',
                 loss = 'binary_crossentropy',
                 metrics = ['accuracy'])
```

```
model_2.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
bidirectional_1 (Bidirection (None, 200)		320800
dense_1 (Dense)	(None, 1)	201
Total params:	321,001	
Trainable params:	321,001	
Non-trainable params:	0	

In [39]:

```
model_2.fit(x = X_train_vectorized, y = y_train_vectorized, validation_split = 0.15, epochs=20)

Train on 38250 samples, validate on 6750 samples
Epoch 1/20
38250/38250 [=====] - 52s 1ms/sample - loss: 0.5456
- accuracy: 0.7143 - val_loss: 0.4991 - val_accuracy: 0.7511
Epoch 2/20
38250/38250 [=====] - 53s 1ms/sample - loss: 0.4851
- accuracy: 0.7566 - val_loss: 0.4786 - val_accuracy: 0.7603
Epoch 3/20
38250/38250 [=====] - 52s 1ms/sample - loss: 0.4589
- accuracy: 0.7734 - val_loss: 0.4749 - val_accuracy: 0.7640
Epoch 4/20
38250/38250 [=====] - 52s 1ms/sample - loss: 0.4354
- accuracy: 0.7882 - val_loss: 0.4676 - val_accuracy: 0.7686
Epoch 5/20
38250/38250 [=====] - 53s 1ms/sample - loss: 0.4117
- accuracy: 0.8025 - val_loss: 0.4917 - val_accuracy: 0.7613
Epoch 6/20
38250/38250 [=====] - 52s 1ms/sample - loss: 0.3849
- accuracy: 0.8179 - val_loss: 0.4727 - val_accuracy: 0.7667
Epoch 7/20
38250/38250 [=====] - 52s 1ms/sample - loss: 0.3511
- accuracy: 0.8372 - val_loss: 0.5012 - val_accuracy: 0.7529
Epoch 8/20
38250/38250 [=====] - 53s 1ms/sample - loss: 0.3122
- accuracy: 0.8599 - val_loss: 0.5162 - val_accuracy: 0.7553
Epoch 9/20
38250/38250 [=====] - 52s 1ms/sample - loss: 0.2662
- accuracy: 0.8848 - val_loss: 0.5886 - val_accuracy: 0.7609
Epoch 10/20
38250/38250 [=====] - 52s 1ms/sample - loss: 0.2165
- accuracy: 0.9092 - val_loss: 0.6468 - val_accuracy: 0.7603
Epoch 11/20
38250/38250 [=====] - 53s 1ms/sample - loss: 0.1680
- accuracy: 0.9329 - val_loss: 0.7128 - val_accuracy: 0.7470
Epoch 12/20
38250/38250 [=====] - 53s 1ms/sample - loss: 0.1269
- accuracy: 0.9524 - val_loss: 0.8222 - val_accuracy: 0.7526
Epoch 13/20
38250/38250 [=====] - 52s 1ms/sample - loss: 0.0908
- accuracy: 0.9670 - val_loss: 0.8999 - val_accuracy: 0.7487
Epoch 14/20
38250/38250 [=====] - 53s 1ms/sample - loss: 0.0651
- accuracy: 0.9780 - val_loss: 1.0383 - val_accuracy: 0.7427
Epoch 15/20
38250/38250 [=====] - 52s 1ms/sample - loss: 0.0495
- accuracy: 0.9842 - val_loss: 1.1222 - val_accuracy: 0.7464
Epoch 16/20
38250/38250 [=====] - 52s 1ms/sample - loss: 0.0379
- accuracy: 0.9884 - val_loss: 1.2738 - val_accuracy: 0.7513
Epoch 17/20
38250/38250 [=====] - 52s 1ms/sample - loss: 0.0326
- accuracy: 0.9897 - val_loss: 1.3313 - val_accuracy: 0.7434
Epoch 18/20
38250/38250 [=====] - 52s 1ms/sample - loss: 0.0364
- accuracy: 0.9888 - val_loss: 1.2889 - val_accuracy: 0.7495
Epoch 19/20
38250/38250 [=====] - 52s 1ms/sample - loss: 0.0237
```

```
- accuracy: 0.9933 - val_loss: 1.4557 - val_accuracy: 0.7465
Epoch 20/20
38250/38250 [=====] - 52s 1ms/sample - loss: 0.0216
- accuracy: 0.9940 - val_loss: 1.3947 - val_accuracy: 0.7459
```

Out[39]:

```
<tensorflow.python.keras.callbacks.History at 0x7f8fd4befa20>
```

In [40]:

```
results_2 = model_2.evaluate(X_test_vectorized, y_test_vectorized)

print('Test loss, test accuracy:', results_2)
```

```
Test loss, test accuracy: [1.3811187601089479, 0.7412]
```

Как и ожидалось, использование эмбеддингов показало лучший результат, чем кодирование слов просто целыми числами — 74%.

Задание 4

Поэкспериментируйте со структурой сети (добавьте больше рекуррентных, полносвязных или сверточных слоев). Как это повлияло на качество классификации?

In [41]:

```
model_3 = tf.keras.Sequential()

model_3.add(Bidirectional(LSTM(5, return_sequences = True), merge_mode = 'concat',
                         input_shape = (MAX_LENGTH, VECTORS_LENGTH)))
model_3.add(LSTM(1, return_sequences = False))
model_3.add(Dense(10, activation = 'linear'))
model_3.add(Dense(1, activation = 'sigmoid'))
```

```
WARNING:tensorflow:Layer lstm_2 will not use cuDNN kernel since it doesn't meet the cuDNN kernel criteria. It will use generic GPU kernel as fallback when running on GPU
```

```
WARNING:tensorflow:Layer lstm_2 will not use cuDNN kernel since it doesn't meet the cuDNN kernel criteria. It will use generic GPU kernel as fallback when running on GPU
```

```
WARNING:tensorflow:Layer lstm_2 will not use cuDNN kernel since it doesn't meet the cuDNN kernel criteria. It will use generic GPU kernel as fallback when running on GPU
```

```
WARNING:tensorflow:Layer lstm_3 will not use cuDNN kernel since it doesn't meet the cuDNN kernel criteria. It will use generic GPU kernel as fallback when running on GPU
```

In [42]:

```
model_3.compile(optimizer = 'adam',
                 loss = 'binary_crossentropy',
                 metrics = ['accuracy'])
```

```
model_3.summary()
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
=====		
bidirectional_2 (Bidirection (None, 40, 10)		12240
lstm_3 (LSTM)	(None, 1)	48
dense_2 (Dense)	(None, 10)	20
dense_3 (Dense)	(None, 1)	11
=====		
Total params: 12,319		
Trainable params: 12,319		
Non-trainable params: 0		

In [44]:

```
model_3.fit(x = X_train_vectorized, y = y_train_vectorized, validation_split = 0.15, epochs=20)

Train on 38250 samples, validate on 6750 samples
Epoch 1/20
38250/38250 [=====] - 83s 2ms/sample - loss: 0.6033
- accuracy: 0.6808 - val_loss: 0.5700 - val_accuracy: 0.7108
Epoch 2/20
38250/38250 [=====] - 82s 2ms/sample - loss: 0.5302
- accuracy: 0.7388 - val_loss: 0.5133 - val_accuracy: 0.7453
Epoch 3/20
38250/38250 [=====] - 81s 2ms/sample - loss: 0.4987
- accuracy: 0.7546 - val_loss: 0.4906 - val_accuracy: 0.7566
Epoch 4/20
38250/38250 [=====] - 81s 2ms/sample - loss: 0.4819
- accuracy: 0.7621 - val_loss: 0.4779 - val_accuracy: 0.7625
Epoch 5/20
38250/38250 [=====] - 82s 2ms/sample - loss: 0.4681
- accuracy: 0.7725 - val_loss: 0.4778 - val_accuracy: 0.7579
Epoch 6/20
38250/38250 [=====] - 81s 2ms/sample - loss: 0.4602
- accuracy: 0.7770 - val_loss: 0.4755 - val_accuracy: 0.7636
Epoch 7/20
38250/38250 [=====] - 81s 2ms/sample - loss: 0.4515
- accuracy: 0.7817 - val_loss: 0.4838 - val_accuracy: 0.7563
Epoch 8/20
38250/38250 [=====] - 81s 2ms/sample - loss: 0.4444
- accuracy: 0.7843 - val_loss: 0.4796 - val_accuracy: 0.7613
Epoch 9/20
38250/38250 [=====] - 82s 2ms/sample - loss: 0.4363
- accuracy: 0.7929 - val_loss: 0.4616 - val_accuracy: 0.7683
Epoch 10/20
38250/38250 [=====] - 81s 2ms/sample - loss: 0.4329
- accuracy: 0.7923 - val_loss: 0.4661 - val_accuracy: 0.7726
Epoch 11/20
38250/38250 [=====] - 81s 2ms/sample - loss: 0.4258
- accuracy: 0.7961 - val_loss: 0.4658 - val_accuracy: 0.7674
Epoch 12/20
38250/38250 [=====] - 81s 2ms/sample - loss: 0.4192
- accuracy: 0.8000 - val_loss: 0.4680 - val_accuracy: 0.7711
Epoch 13/20
38250/38250 [=====] - 82s 2ms/sample - loss: 0.4138
- accuracy: 0.8031 - val_loss: 0.4609 - val_accuracy: 0.7754
Epoch 14/20
38250/38250 [=====] - 81s 2ms/sample - loss: 0.4084
- accuracy: 0.8092 - val_loss: 0.4622 - val_accuracy: 0.7763
Epoch 15/20
38250/38250 [=====] - 82s 2ms/sample - loss: 0.4028
- accuracy: 0.8098 - val_loss: 0.4717 - val_accuracy: 0.7705
Epoch 16/20
38250/38250 [=====] - 81s 2ms/sample - loss: 0.3975
- accuracy: 0.8142 - val_loss: 0.4648 - val_accuracy: 0.7742
Epoch 17/20
38250/38250 [=====] - 81s 2ms/sample - loss: 0.3942
- accuracy: 0.8166 - val_loss: 0.4751 - val_accuracy: 0.7673
Epoch 18/20
38250/38250 [=====] - 81s 2ms/sample - loss: 0.3886
- accuracy: 0.8202 - val_loss: 0.4730 - val_accuracy: 0.7730
Epoch 19/20
38250/38250 [=====] - 81s 2ms/sample - loss: 0.3844
```

```
- accuracy: 0.8226 - val_loss: 0.4740 - val_accuracy: 0.7753
Epoch 20/20
38250/38250 [=====] - 82s 2ms/sample - loss: 0.3797
- accuracy: 0.8246 - val_loss: 0.4765 - val_accuracy: 0.7727
```

Out[44]:

```
<tensorflow.python.keras.callbacks.History at 0x7f8ef6e7a6a0>
```

In [45]:

```
results_3 = model_3.evaluate(X_test_vectorized, y_test_vectorized)

print('Test loss, test accuracy:', results_3)
```

```
Test loss, test accuracy: [0.49010655212402343, 0.766]
```

Добавление ещё одного рекуррентного слоя ненамного улучшило результат — точность 76% на тестовой выборке.

Задание 5

Используйте предобученную рекуррентную нейронную сеть (например, *DeepMoji* или что-то подобное).

Какой максимальный результат удалось получить на контрольной выборке?

На своих моделях удалось достичнуть максимальной точности 76%.

Лабораторная работа №8

Рекуррентные нейронные сети для анализа временных рядов

Набор данных для прогнозирования временных рядов, который состоит из среднемесячного числа пятен на солнце, наблюдавшихся с января 1749 по август 2017.

Данные в виде csv-файла можно скачать на сайте Kaggle: <https://www.kaggle.com/robervalt/sunspots/> (<https://www.kaggle.com/robervalt/sunspots/>)

Задание 1

Загрузите данные. Изобразите ряд в виде графика. Вычислите основные характеристики временного ряда (сезонность, тренд, автокорреляцию).

In [1]:

```
from google.colab import drive  
drive.mount('/content/drive', force_remount = True)
```

Mounted at /content/drive

In [0]:

```
BASE_DIR = '/content/drive/My Drive/Colab Files/mo-2'  
  
import sys  
  
sys.path.append(BASE_DIR)  
  
import os
```

In [0]:

```
DATA_ARCHIVE_NAME = 'sunspots.zip'  
  
LOCAL_DIR_NAME = 'sunspots'
```

In [0]:

```
from zipfile import ZipFile  
  
with ZipFile(os.path.join(BASE_DIR, DATA_ARCHIVE_NAME), 'r') as zip_:  
    zip_.extractall(LOCAL_DIR_NAME)
```

In [0]:

```
DATA_FILE_PATH = 'sunspots/Sunspots.csv'
```

In [0]:

```
import pandas as pd

all_df = pd.read_csv(DATA_FILE_PATH, parse_dates = ['Date'], index_col = 'Date')
```

In [7]:

```
print(all_df.shape)

(3252, 2)
```

In [8]:

```
all_df.keys()
```

Out[8]:

```
Index(['Unnamed: 0', 'Monthly Mean Total Sunspot Number'], dtype='object')
```

In [9]:

```
from statsmodels.tsa.seasonal import seasonal_decompose

additive = seasonal_decompose(all_df['Monthly Mean Total Sunspot Number'], model = 'additiv

/usr/local/lib/python3.6/dist-packages/statsmodels/tools/_testing.py:19: FutureWarning: pandas.util.testing is deprecated. Use the functions in the public API at pandas.testing instead.
    import pandas.util.testing as tm
```

In [0]:

```
%matplotlib inline

import matplotlib.pyplot as plt
```

In [0]:

```
import seaborn as sns

from matplotlib import rcParams

rcParams['figure.figsize'] = 11.7, 8.27

sns.set()

sns.set_palette(sns.color_palette('hls', 8))
```

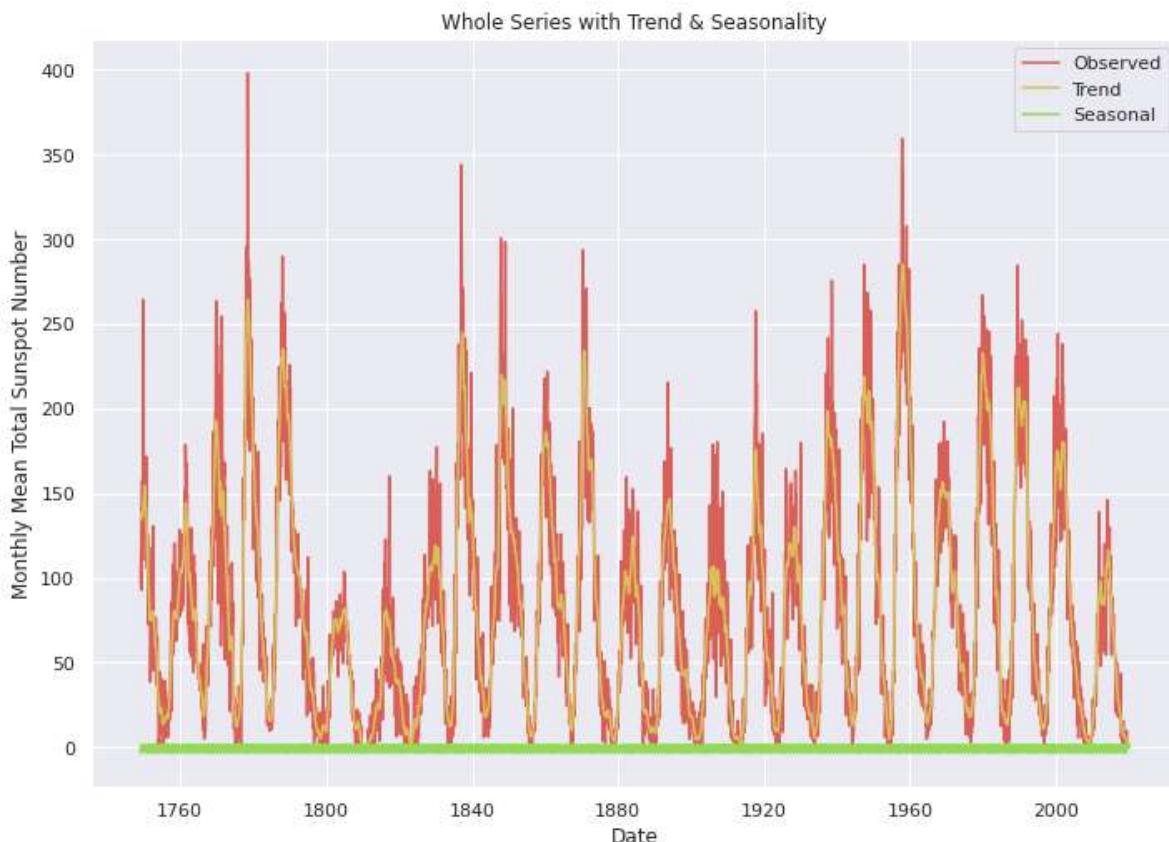
In [12]:

```
sns.lineplot(data = additive.observed, label = 'Observed')
sns.lineplot(data = additive.trend, label = 'Trend')
sns.lineplot(data = additive.seasonal, label = 'Seasonal')

plt.xlabel('Date')
plt.ylabel('Monthly Mean Total Sunspot Number')

plt.title('Whole Series with Trend & Seasonality')

plt.show()
```



Рассмотрим подробнее на небольшом промежутке:

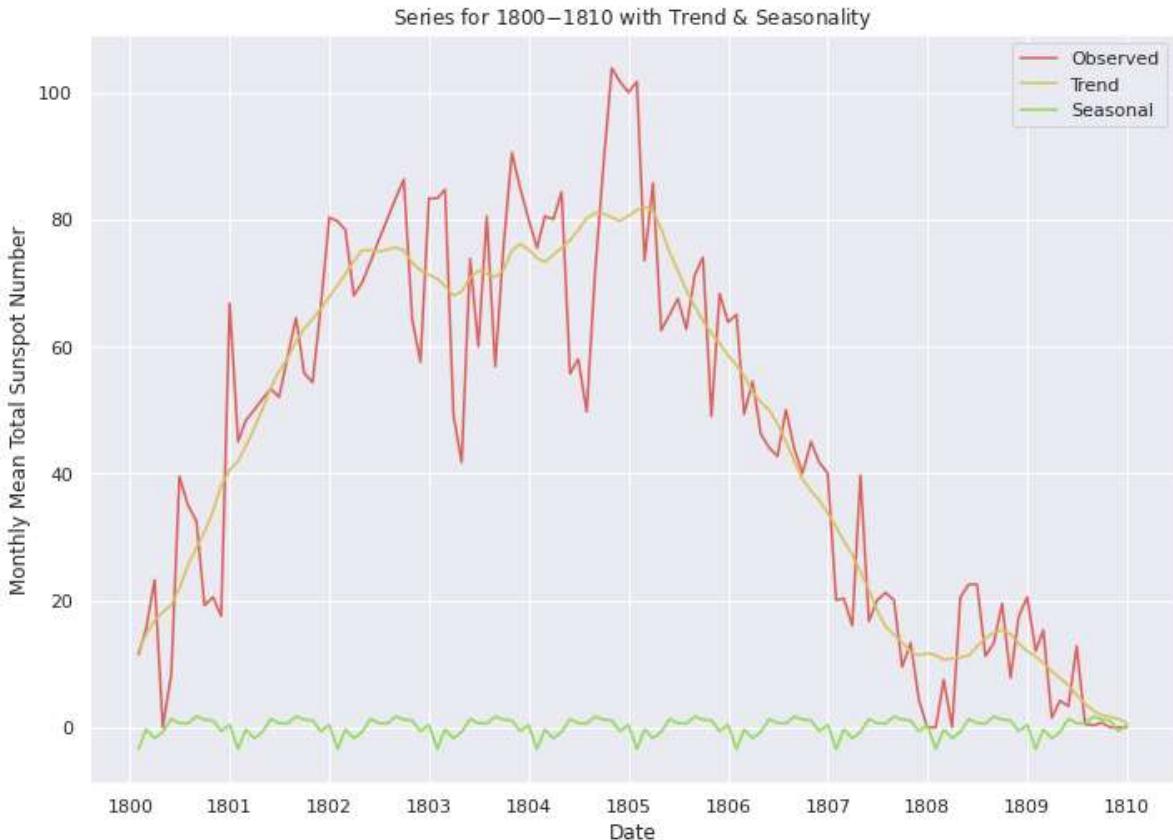
In [13]:

```
sns.lineplot(data = additive.observed['1800-01-01':'1810-01-01'], label = 'Observed')
sns.lineplot(data = additive.trend['1800-01-01':'1810-01-01'], label = 'Trend')
sns.lineplot(data = additive.seasonal['1800-01-01':'1810-01-01'], label = 'Seasonal')

plt.xlabel('Date')
plt.ylabel('Monthly Mean Total Sunspot Number')

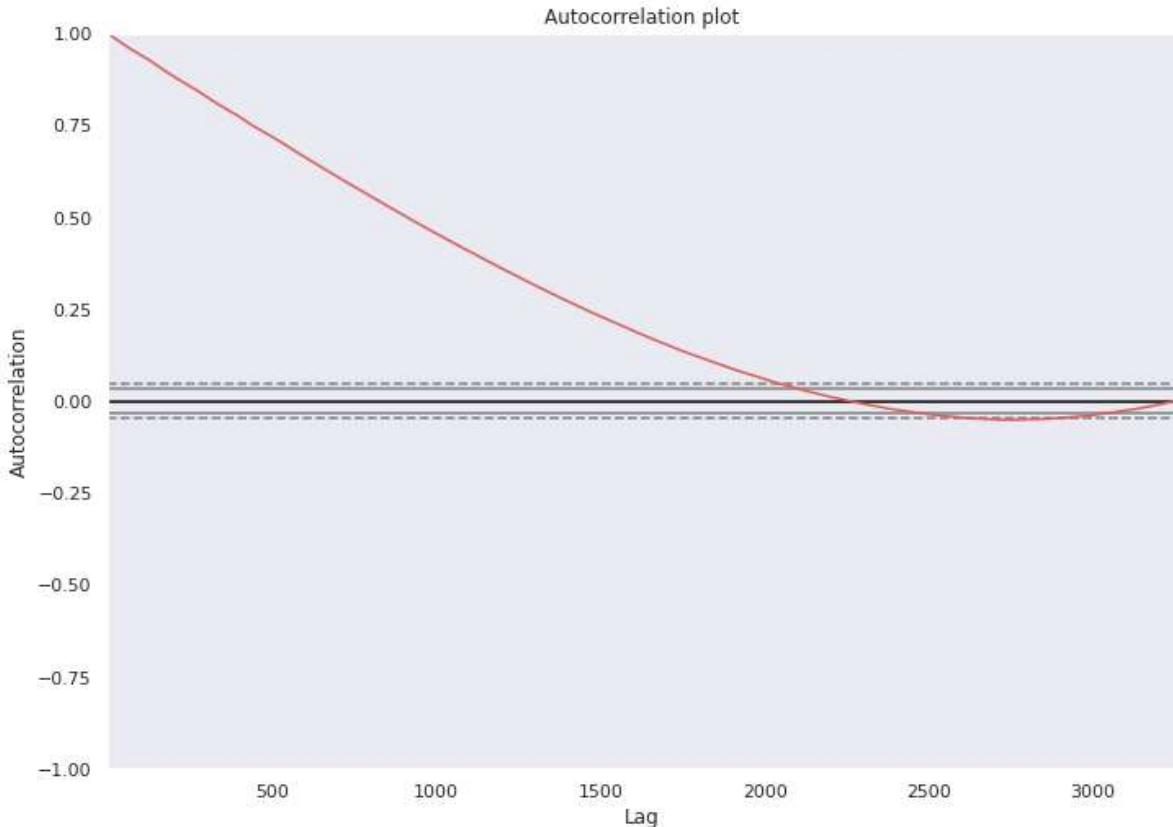
plt.title('Series for 1800$-$1810 with Trend & Seasonality')

plt.show()
```



In [14]:

```
from pandas.plotting import autocorrelation_plot  
  
autocorrelation_plot(all_df.values.tolist())  
  
plt.title('Autocorrelation plot')  
  
plt.show()
```



Задание 2

Для прогнозирования разделите временной ряд на обучающую, валидационную и контрольную выборки.

Этот шаг будет применён автоматически как параметр `validation_split` метода `model.fit()`.

Задание 3

Примените модель ARIMA для прогнозирования значений данного временного ряда.

In [15]:

```
! pip install pmdarima  
!  
! pip show pmdarima
```

```
Requirement already satisfied: pmdarima in /usr/local/lib/python3.6/dist-packages (1.5.3)  
Requirement already satisfied: Cython>=0.29 in /usr/local/lib/python3.6/dist-packages (from pmdarima) (0.29.16)  
Requirement already satisfied: pandas>=0.19 in /usr/local/lib/python3.6/dist-packages (from pmdarima) (1.0.3)  
Requirement already satisfied: scikit-learn>=0.22 in /usr/local/lib/python3.6/dist-packages (from pmdarima) (0.22.2.post1)  
Requirement already satisfied: numpy>=1.17.3 in /usr/local/lib/python3.6/dist-packages (from pmdarima) (1.18.2)  
Requirement already satisfied: scipy>=1.3.2 in /usr/local/lib/python3.6/dist-packages (from pmdarima) (1.4.1)  
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.6/dist-packages (from pmdarima) (0.14.1)  
Requirement already satisfied: statsmodels>=0.10.2 in /usr/local/lib/python3.6/dist-packages (from pmdarima) (0.10.2)  
Requirement already satisfied: urllib3 in /usr/local/lib/python3.6/dist-packages (from pmdarima) (1.24.3)  
Requirement already satisfied: pytz>=2017.2 in /usr/local/lib/python3.6/dist-packages (from pandas>=0.19->pmdarima) (2018.9)  
Requirement already satisfied: python-dateutil>=2.6.1 in /usr/local/lib/python3.6/dist-packages (from pandas>=0.19->pmdarima) (2.8.1)  
Requirement already satisfied: patsy>=0.4.0 in /usr/local/lib/python3.6/dist-packages (from statsmodels>=0.10.2->pmdarima) (0.5.1)  
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.6/dist-packages (from python-dateutil>=2.6.1->pandas>=0.19->pmdarima) (1.12.0)  
Name: pmdarima  
Version: 1.5.3  
Summary: Python's forecast::auto.arima equivalent  
Home-page: http://alkaline-ml.com/pmdarima (http://alkaline-ml.com/pmdarima)  
Author: None  
Author-email: None  
License: MIT  
Location: /usr/local/lib/python3.6/dist-packages  
Requires: joblib, Cython, pandas, urllib3, statsmodels, numpy, scikit-learn, scipy  
Required-by:
```

In [0]:

```
test_period = 6 * 12
```

In [17]:

```
from pmdarima.arima import auto_arima

arima = auto_arima(all_df[ 'Monthly Mean Total Sunspot Number'][:-test_period],
                  trace = True, error_action = 'ignore', suppress_warnings = True, seasonal=
```

Performing stepwise search to minimize aic
Fit ARIMA: (2, 0, 2)x(1, 0, 1, 12) (constant=True); AIC=29588.066, BIC=2963
6.583, Time=23.372 seconds
Fit ARIMA: (0, 0, 0)x(0, 0, 0, 12) (constant=True); AIC=35872.649, BIC=3588
4.778, Time=0.065 seconds
Fit ARIMA: (1, 0, 0)x(1, 0, 0, 12) (constant=True); AIC=30025.545, BIC=3004
9.804, Time=5.185 seconds
Fit ARIMA: (0, 0, 1)x(0, 0, 1, 12) (constant=True); AIC=32380.500, BIC=3240
4.758, Time=5.341 seconds
Fit ARIMA: (0, 0, 0)x(0, 0, 0, 12) (constant=False); AIC=38764.997, BIC=3877
1.062, Time=0.049 seconds
Fit ARIMA: (2, 0, 2)x(0, 0, 1, 12) (constant=True); AIC=29586.807, BIC=2962
9.260, Time=13.302 seconds
Fit ARIMA: (2, 0, 2)x(0, 0, 0, 12) (constant=True); AIC=29595.958, BIC=2963
2.345, Time=2.032 seconds
Fit ARIMA: (2, 0, 2)x(0, 0, 2, 12) (constant=True); AIC=29587.374, BIC=2963
5.891, Time=45.942 seconds
Fit ARIMA: (2, 0, 2)x(1, 0, 0, 12) (constant=True); AIC=29587.365, BIC=2962
9.817, Time=16.549 seconds
Fit ARIMA: (2, 0, 2)x(1, 0, 2, 12) (constant=True); AIC=29582.451, BIC=2963
7.033, Time=67.935 seconds
Fit ARIMA: (2, 0, 2)x(2, 0, 2, 12) (constant=True); AIC=29552.494, BIC=2961
3.140, Time=71.844 seconds
Near non-invertible roots for order (2, 0, 2)(2, 0, 2, 12); setting score to
inf (at least one inverse root too close to the border of the unit circle:
0.993)
Fit ARIMA: (2, 0, 2)x(2, 0, 1, 12) (constant=True); AIC=29581.341, BIC=2963
5.923, Time=64.230 seconds
Fit ARIMA: (2, 0, 2)x(2, 0, 0, 12) (constant=True); AIC=29587.014, BIC=2963
5.531, Time=52.283 seconds
Fit ARIMA: (1, 0, 2)x(2, 0, 1, 12) (constant=True); AIC=29579.106, BIC=2962
7.623, Time=43.009 seconds
Fit ARIMA: (1, 0, 2)x(1, 0, 1, 12) (constant=True); AIC=29586.021, BIC=2962
8.473, Time=18.368 seconds
Fit ARIMA: (1, 0, 2)x(2, 0, 0, 12) (constant=True); AIC=29585.061, BIC=2962
7.513, Time=39.560 seconds
Fit ARIMA: (1, 0, 2)x(2, 0, 2, 12) (constant=True); AIC=29551.544, BIC=2960
6.126, Time=72.061 seconds
Near non-invertible roots for order (1, 0, 2)(2, 0, 2, 12); setting score to
inf (at least one inverse root too close to the border of the unit circle:
0.995)
Fit ARIMA: (1, 0, 2)x(1, 0, 0, 12) (constant=True); AIC=29585.380, BIC=2962
1.768, Time=8.778 seconds
Fit ARIMA: (1, 0, 2)x(1, 0, 2, 12) (constant=True); AIC=29580.389, BIC=2962
8.906, Time=49.857 seconds
Fit ARIMA: (0, 0, 2)x(2, 0, 1, 12) (constant=True); AIC=31382.620, BIC=3142
5.073, Time=33.173 seconds
Near non-invertible roots for order (0, 0, 2)(2, 0, 1, 12); setting score to
inf (at least one inverse root too close to the border of the unit circle:
1.000)
Fit ARIMA: (1, 0, 1)x(2, 0, 1, 12) (constant=True); AIC=29629.733, BIC=2967
2.185, Time=54.417 seconds
Fit ARIMA: (1, 0, 3)x(2, 0, 1, 12) (constant=True); AIC=29580.861, BIC=2963
5.443, Time=48.949 seconds

Fit ARIMA: (0, 0, 1)x(2, 0, 1, 12) (constant=True); AIC=32043.176, BIC=32079.563, Time=28.353 seconds
Near non-invertible roots for order (0, 0, 1)(2, 0, 1, 12); setting score to inf (at least one inverse root too close to the border of the unit circle:
1.000)
Fit ARIMA: (0, 0, 3)x(2, 0, 1, 12) (constant=True); AIC=31074.278, BIC=31122.795, Time=42.232 seconds
Near non-invertible roots for order (0, 0, 3)(2, 0, 1, 12); setting score to inf (at least one inverse root too close to the border of the unit circle:
1.000)
Fit ARIMA: (2, 0, 1)x(2, 0, 1, 12) (constant=True); AIC=29592.648, BIC=29641.165, Time=51.681 seconds
Fit ARIMA: (2, 0, 3)x(2, 0, 1, 12) (constant=True); AIC=29579.013, BIC=29639.659, Time=62.712 seconds
Fit ARIMA: (2, 0, 3)x(1, 0, 1, 12) (constant=True); AIC=29586.998, BIC=29641.579, Time=22.034 seconds
Fit ARIMA: (2, 0, 3)x(2, 0, 0, 12) (constant=True); AIC=29585.838, BIC=29640.420, Time=60.607 seconds
Fit ARIMA: (2, 0, 3)x(2, 0, 2, 12) (constant=True); AIC=29543.930, BIC=29610.641, Time=80.082 seconds
Near non-invertible roots for order (2, 0, 3)(2, 0, 2, 12); setting score to inf (at least one inverse root too close to the border of the unit circle:
0.995)
Fit ARIMA: (2, 0, 3)x(1, 0, 0, 12) (constant=True); AIC=29586.591, BIC=29635.108, Time=17.100 seconds
Fit ARIMA: (2, 0, 3)x(1, 0, 2, 12) (constant=True); AIC=29580.547, BIC=29641.194, Time=72.056 seconds
Fit ARIMA: (3, 0, 3)x(2, 0, 1, 12) (constant=True); AIC=29578.087, BIC=29644.798, Time=79.448 seconds
Fit ARIMA: (3, 0, 3)x(1, 0, 1, 12) (constant=True); AIC=29586.766, BIC=29647.413, Time=22.877 seconds
Fit ARIMA: (3, 0, 3)x(2, 0, 0, 12) (constant=True); AIC=29584.745, BIC=29645.391, Time=79.798 seconds
Fit ARIMA: (3, 0, 3)x(2, 0, 2, 12) (constant=True); AIC=29546.099, BIC=29618.874, Time=82.080 seconds
Near non-invertible roots for order (3, 0, 3)(2, 0, 2, 12); setting score to inf (at least one inverse root too close to the border of the unit circle:
0.995)
Fit ARIMA: (3, 0, 3)x(1, 0, 0, 12) (constant=True); AIC=29585.293, BIC=29639.875, Time=25.847 seconds
Fit ARIMA: (3, 0, 3)x(1, 0, 2, 12) (constant=True); AIC=29579.288, BIC=29645.999, Time=71.712 seconds
Fit ARIMA: (3, 0, 2)x(2, 0, 1, 12) (constant=True); AIC=29587.135, BIC=29647.781, Time=73.021 seconds
Fit ARIMA: (4, 0, 3)x(2, 0, 1, 12) (constant=True); AIC=29579.706, BIC=29652.482, Time=82.381 seconds
Fit ARIMA: (3, 0, 4)x(2, 0, 1, 12) (constant=True); AIC=29579.646, BIC=29652.421, Time=91.857 seconds
Fit ARIMA: (2, 0, 4)x(2, 0, 1, 12) (constant=True); AIC=29576.620, BIC=29643.331, Time=70.104 seconds
Fit ARIMA: (2, 0, 4)x(1, 0, 1, 12) (constant=True); AIC=29583.960, BIC=29644.607, Time=29.800 seconds
Fit ARIMA: (2, 0, 4)x(2, 0, 0, 12) (constant=True); AIC=29582.698, BIC=29643.345, Time=71.533 seconds
Fit ARIMA: (2, 0, 4)x(2, 0, 2, 12) (constant=True); AIC=29547.505, BIC=29620.281, Time=100.620 seconds
Near non-invertible roots for order (2, 0, 4)(2, 0, 2, 12); setting score to inf (at least one inverse root too close to the border of the unit circle:
0.994)
Fit ARIMA: (2, 0, 4)x(1, 0, 0, 12) (constant=True); AIC=29568.222, BIC=29622.804, Time=23.695 seconds

```
Fit ARIMA: (2, 0, 4)x(0, 0, 0, 12) (constant=True); AIC=29479.053, BIC=2952  
7.570, Time=8.133 seconds  
Fit ARIMA: (2, 0, 4)x(0, 0, 1, 12) (constant=True); AIC=29472.268, BIC=2952  
6.849, Time=31.449 seconds  
Fit ARIMA: (2, 0, 4)x(0, 0, 2, 12) (constant=True); AIC=29573.231, BIC=2963  
3.878, Time=81.292 seconds  
Fit ARIMA: (2, 0, 4)x(1, 0, 2, 12) (constant=True); AIC=29578.088, BIC=2964  
4.799, Time=84.294 seconds  
Fit ARIMA: (1, 0, 4)x(0, 0, 1, 12) (constant=True); AIC=29579.628, BIC=2962  
8.145, Time=15.428 seconds  
Fit ARIMA: (2, 0, 3)x(0, 0, 1, 12) (constant=True); AIC=29585.945, BIC=2963  
4.462, Time=10.997 seconds  
Fit ARIMA: (3, 0, 4)x(0, 0, 1, 12) (constant=True); AIC=29523.347, BIC=2958  
3.993, Time=30.866 seconds  
Fit ARIMA: (2, 0, 5)x(0, 0, 1, 12) (constant=True); AIC=29583.600, BIC=2964  
4.247, Time=15.339 seconds  
Fit ARIMA: (1, 0, 3)x(0, 0, 1, 12) (constant=True); AIC=29586.789, BIC=2962  
9.241, Time=10.503 seconds  
Fit ARIMA: (1, 0, 5)x(0, 0, 1, 12) (constant=True); AIC=29579.796, BIC=2963  
4.378, Time=21.199 seconds  
Fit ARIMA: (3, 0, 3)x(0, 0, 1, 12) (constant=True); AIC=29584.591, BIC=2963  
9.173, Time=20.767 seconds  
Fit ARIMA: (3, 0, 5)x(0, 0, 1, 12) (constant=True); AIC=29449.139, BIC=2951  
5.850, Time=34.889 seconds  
Near non-invertible roots for order (3, 0, 5)(0, 0, 1, 12); setting score to  
inf (at least one inverse root too close to the border of the unit circle:  
0.997)  
Total fit time: 2443.158 seconds
```

In [0]:

```
arima_forecast = arima.predict(n_periods = test_period)
```

In [19]:

```
from sklearn.metrics import mean_squared_error  
  
mean_squared_error(all_df['Monthly Mean Total Sunspot Number'][~-test_period:], arima_forecast)
```

Out[19]:

```
3196.9943474969787
```

Задание 4

Повторите эксперимент по прогнозированию, реализовав рекуррентную нейронную сеть (с как минимум 2 рекуррентными слоями).

Сначала нужно создать датасет из данных.

In [20]:

```
! pip install tensorflow-gpu --pre --quiet
```

```
! pip show tensorflow-gpu
```

```
Name: tensorflow-gpu
Version: 2.2.0rc3
Summary: TensorFlow is an open source machine learning framework for everyone.
Home-page: https://www.tensorflow.org/ (https://www.tensorflow.org/)
Author: Google Inc.
Author-email: packages@tensorflow.org
License: Apache 2.0
Location: /usr/local/lib/python3.6/dist-packages
Requires: google-pasta, opt-einsum, tensorboard, six, h5py, tensorflow-estimator, termcolor, wrapt, wheel, keras-preprocessing, protobuf, gast, astunparse, scipy, numpy, grpcio, absl-py
Required-by:
```

In [0]:

```
TIME_STEPS = 100
```

```
TEST_PERIOD = 1000
```

In [0]:

```
import numpy as np
from datetime import timezone

def timeseries_to_dataset(_X_ts, _time_steps):
    samples_n_ = len(_X_ts) - _time_steps
    print(len(_X_ts))

    X_ = np.zeros((samples_n_, _time_steps))
    y_ = np.zeros((samples_n_,))

    for i in range(samples_n_):
        X_[i] = _X_ts[i:(i + _time_steps)]
        y_[i] = _X_ts[(i + _time_steps)]

    return X_[..., np.newaxis], y_
```

In [31]:

```
X, y = timeseries_to_dataset(all_df['Monthly Mean Total Sunspot Number'][:-TEST_PERIOD].values
X_test, y_test = timeseries_to_dataset(all_df['Monthly Mean Total Sunspot Number'][-TEST_PERIOD:]
```

2252
1000

In [0]:

```
import tensorflow as tf
from tensorflow import keras
```

In [37]:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense

model = tf.keras.Sequential()

model.add(LSTM(8, activation = 'relu', return_sequences = True, input_shape = X.shape[-2:]))
model.add(LSTM(8, activation = 'relu'))
model.add(Dense(1))
```

WARNING:tensorflow:Layer lstm_4 will not use cuDNN kernel since it doesn't meet the cuDNN kernel criteria. It will use generic GPU kernel as fallback when running on GPU

WARNING:tensorflow:Layer lstm_5 will not use cuDNN kernel since it doesn't meet the cuDNN kernel criteria. It will use generic GPU kernel as fallback when running on GPU

In [38]:

```
model.compile(optimizer = 'adam',
              loss = 'mse',
              metrics = ['accuracy'])
```

```
model.summary()
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
lstm_4 (LSTM)	(None, 100, 8)	320
lstm_5 (LSTM)	(None, 8)	544
dense_2 (Dense)	(None, 1)	9
Total params:	873	
Trainable params:	873	
Non-trainable params:	0	

In [39]:

```
model.fit(x = X, y = y, validation_split = 0.15, epochs = 20, verbose = 0)
```

Out[39]:

```
<tensorflow.python.keras.callbacks.History at 0x7f162e959fd0>
```

In [40]:

```
results = model.evaluate(X_test, y_test)

print('Test mse, test accuracy:', results)

29/29 [=====] - 1s 21ms/step - loss: 651.9083 - acc
uracy: 0.0011
Test mse, test accuracy: [651.9083251953125, 0.001111111380159855]
```

Задание 5

Сравните качество прогноза моделей.

Какой максимальный результат удалось получить на контрольной выборке?

Нейронная сеть дала среднеквадратичную ошибку в 4 раза больше, чем ARIMA, а точность предсказания вообще близка к нулю. Можно сделать вывод, что предсказание временных рядов требует более тонкой настройки архитектуры сетей.