УДК 519.68:007.5

Лабораторная работа 2. Сверточные сети и автоэнкодеры

Цели работы: работа с предобученными сверточными сетями. Адаптация готовых сетей под новые задачи распознания изображений.

Ключевые слова: сверточные сети, распознание изображений, перенос обучения, автоэнкодеры.

§ 1. Перенос обучения и сверточные сети

1.1. Подготовка к работе. Введем следующие команды в рабочей тетради jupyter и проверим, что они исполнились без ошибок:

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
import matplotlib.pyplot as plt
import os
```

Как вариант, можно вместо локально установленного Python'а использовать облачный сервис Google Colab с его реализацией тетради jupyter.

```
_URL = # загружаем готовый набор изображений кошек и собак с данного адреса
 'https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.zip'
path_to_zip = tf.keras.utils.get_file( 'cats_and_dogs.zip',
                                       origin=_URL,
                                       extract=True)
# при загрузке функция вернет полный путь до корня распакованного архива
PATH = os.path.join(os.path.dirname(path_to_zip),
                    'cats_and_dogs_filtered')
# директория train будет источником данных для обучения модели
train_dir = os.path.join(PATH, 'train')
# директория validation будет источником данных для вылидации
validation_dir = os.path.join(PATH, 'validation')
BATCH_SIZE = 32 # можете установить по желанию любой другой размер пакета
IMG_SIZE = (150, 150) # приведем к этому размеру все картинки
train_dataset = # сохраняем датасет для обучения модели
    tf.keras.utils.image_dataset_from_directory(train_dir,
                                                shuffle=True,
                                                batch_size=BATCH_SIZE,
                                                image_size=IMG_SIZE)
validation_dataset = # аналогично получаем датасет для валидации
    tf.keras.utils.image_dataset_from_directory(validation_dir,
                                                shuffle=True,
                                                batch_size=BATCH_SIZE,
                                                image_size=IMG_SIZE)
```

Проверим что изображения были загружены корректно, выведя на печать первые 9 изображений из обучающего набора вместе с их метками.

```
class_names = train_dataset.class_names
plt.figure(figsize=(10, 10))
for images, labels in train_dataset.take(1):
  for i in range(9):
    ax = plt.subplot(3, 3, i + 1)
    plt.imshow(images[i].numpy().astype("uint8"))
    plt.title(class_names[labels[i]])
    plt.axis("off")
```

Выделим двадцать процентов данных из валидатационной выборки в тестовую, чтобы проконтролировать работу модели после завершения обучения.

```
val_batches = tf.data.experimental.cardinality(validation_dataset)
test_dataset = validation_dataset.take(val_batches // 5)
validation_dataset = validation_dataset.skip(val_batches // 5)
```

Предусмотрим кеширование данных для всех наших датасетов, чтобы предотвратить постоянное обращение к диску.

```
AUTOTUNE = tf.data.AUTOTUNE
train_dataset = train_dataset.prefetch(buffer_size=AUTOTUNE)
validation_dataset = validation_dataset.prefetch(buffer_size=AUTOTUNE)
test_dataset = test_dataset.prefetch(buffer_size=AUTOTUNE)
```

1.2. Искуственное расширение обучающей выборки. При ограниченном наборе обучающих примеров имеет смысл искуственно расширить их число за счет разнообразных преобразований.

```
data_augmentation = tf.keras.Sequential([
  tf.keras.layers.RandomFlip('horizontal'), # симметрическое отражение
  tf.keras.layers.RandomRotation(0.2),      # поворот изображения
])
```

Применим наш преобразователь к обучающей выборке и проверим что преобразования работают как задуманно.

```
for image, _ in train_dataset.take(1):
  plt.figure(figsize=(10, 10))
  first_image = image[0]
  for i in range(9):
    ax = plt.subplot(3, 3, i + 1)
    augmented_image = data_augmentation(tf.expand_dims(first_image, 0))
    plt.imshow(augmented_image[0] / 255)
    plt.axis('off')
```

1.3. Перенос обучения с готовой модели. Возмьмем за основу модель MobileNet с весами для классификации изображений из набора Imagenet, отключив у неё выходные слои.

```
IMG_SHAPE = IMG_SIZE + (3,)
base_model = tf.keras.applications.MobileNetV2(input_shape=IMG_SHAPE,
                                               include_top=False,
                                               weights='imagenet')
```

Загрузим для этой модели подходящую функцию для предобработки изображений, чтобы преобразовать наши картинки к привычному для неё формату

```
preprocess_input = tf.keras.applications.mobilenet_v2.preprocess_input
```

Заморозим коэффициенты этой предобученной модели.

```
base_model.trainable = False
base_model.summary() # проверим что они заморозились
```

Построим итоговую модель, добавив к базовой модели с одной стороны слои для предобработки данных, а с другой — слои для классификации.

```
inputs = tf.keras.Input(shape=(160, 160, 3))
x = data_augmentation(inputs)
x = preprocess_input(x)
x = base_model(x, training=False)
x = tf.keras.layers.GlobalAveragePooling2D()(x)
x = tf.keras.layers.Dropout(0.2)(x)
outputs = tf.keras.layers.Dense(1)(x)
model = tf.keras.Model(inputs, outputs)
```

Скомпилируем итоговую модель, выбрав в качестве функции потерь бинарную кроссэнтропию, а также установив флаг для автоматического преобразования выходных данных к бинарным классам.

```
base_learning_rate = 0.0001
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=base_learning_rate),
    loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
    metrics=['accuracy'])
```

Проведем обучение нашей модели на набольшом количестве эпох.

```
initial_epochs = 10
history = model.fit(train_dataset,
                    epochs=initial_epochs,
                    validation_data=validation_dataset)
```

Визуализируем процесс обучения, построив графики для обучающей и валидатационных выборок для функций потерь и метрик точности.

```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

plt.figure(figsize=(8, 8))
plt.subplot(2, 1, 1)
plt.plot(acc, label='Training Accuracy')
plt.plot(val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.ylabel('Accuracy')
plt.ylim([min(plt.ylim()),1])
plt.title('Training and Validation Accuracy')
```

```
plt.subplot(2, 1, 2)
plt.plot(loss, label='Training Loss')
plt.plot(val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.ylabel('Cross Entropy')
plt.ylim([0,1.0])
plt.title('Training and Validation Loss')
plt.xlabel('epoch')
plt.show()
```

1.4. Тонкая настройка модели. Начинаем работу с разморозки слоев базовой модели (как вариант, можно разморозить не все, а только их часть).

```
base_model.trainable = True
```

Перекомпилируем модель, выбрав другой метод оптимизации (RMSprop) и на порядок уменьшив скорость обучения. Если оставить скорость старой, то модель может переобучиться всего за пару эпох.

```
new_rate = base_learning_rate/10
model.compile(
    loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
    optimizer = tf.keras.optimizers.RMSprop(learning_rate=new_rate),
    metrics=['accuracy'])
model.summary() # проверяем что слои разморозились
```

Запустим модель на обучение, продолжив нумерацию эпох с последней эпохи нашего прошлого сеанса обучения.

```
fine_tune_epochs = 10
total_epochs =  initial_epochs + fine_tune_epochs

history_fine = model.fit(train_dataset,
                         epochs=total_epochs,
                         initial_epoch=history.epoch[-1],
                         validation_data=validation_dataset)
```

Добавим новые данные по метрикам точности и функциям потерь к данным от прошлого сенаса обучения и построим итоговые графики.

```
acc += history_fine.history['accuracy']
val_acc += history_fine.history['val_accuracy']

loss += history_fine.history['loss']
val_loss += history_fine.history['val_loss']

plt.figure(figsize=(8, 8))
plt.subplot(2, 1, 1)
plt.plot(acc, label='Training Accuracy')
plt.plot(val_acc, label='Validation Accuracy')
plt.ylim([0.8, 1])
plt.plot([initial_epochs-1,initial_epochs-1],
          plt.ylim(), label='Start Fine Tuning')
```

```
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(2, 1, 2)
plt.plot(loss, label='Training Loss')
plt.plot(val_loss, label='Validation Loss')
plt.ylim([0, 1.0])
plt.plot([initial_epochs-1,initial_epochs-1],
         plt.ylim(), label='Start Fine Tuning')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.xlabel('epoch')
plt.show()
```

Для выделенной тестовой части датасета проведем финальную апробацию нашей модели.

```
loss, accuracy = model.evaluate(test_dataset)
print('Test accuracy :', accuracy)
```

Итоговую модель мы можем экспортировать в файл с помощью функции model.save('our_first_convmodel.h5'). Проверить её работу далее можно с помощью функций predict и evaluate, предварительно прочитав из файла функцией model.load_model и скомпилировав model.compile со старыми настройками (loss='binary_crossentropy', metrics=['accuracy']).

§ 2. Построение автоэнкодера на основе сверточных сетей

Для нашего примера возьмём базу изображений MNIST (рукописные цифры в монохромном формате). Искомый автоэнкодер должен будет проводить фильтрацию изображений и удалять из них шумы. Строить итоговую модель автоэнкодера мы будем на основе двух базовых моделей: энкодера и декодера.

```
from __future__ import absolute_import
from __future__ import division
import keras
from keras.datasets import mnist
import numpy as np
# Считываем данные из MNIST
(x_train, _), (x_test, _) = mnist.load_data()
# Нормируем данные и приводим массивы к виду (N,size1,size1,1)
image_size = x_train.shape[1]
x_train = np.reshape(x_train, [-1, image_size, image_size, 1])
x_test = np.reshape(x_test, [-1, image_size, image_size, 1])
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
```

Добавим в наши обучающие примеры случайный шум с нормальным законом распределения и $m = \sigma = 0.5$.

```
noise = np.random.normal(loc=0.5, scale=0.5, size=x_train.shape)
x_train_noisy = x_train + noise
```

```
noise = np.random.normal(loc=0.5, scale=0.5, size=x_test.shape)
x_test_noisy = x_test + noise
# Ограничим итоговые значения интервалом [0,1]
x_train_noisy = np.clip(x_train_noisy, 0., 1.)
x_test_noisy = np.clip(x_test_noisy, 0., 1.)
```

Зададим общие параметры для наших нейронных сетей.

```
input_shape = (image_size, image_size, 1)
batch_size = 128
kernel_size = 3
latent_dim = 16
# Зададим два сверточных слоя и число нейронов в каждом слое:
layer_filters = [32, 64]
```

Построим модель энкодера на основе стека сверточных слоев.

```
from keras.layers import Activation, Dense, Input
from keras.layers import Conv2D, Flatten
from keras.layers import Reshape, Conv2DTranspose
from keras.models import Model
from keras import backend as K

encoder_inputs = Input(shape=input_shape, name='encoder_input')
x = encoder_inputs
# Стек из сверточных слоев (strides - дискрет сдвига окна свертки
# в пикселях, padding - без заполнения нулями):
for filters in layer_filters:
    x = Conv2D(filters=filters,
               kernel_size=kernel_size,
               strides=2,
               activation='relu',
               padding='same')(x)

# Запоминаем размерность выхода для построения модели декодера
shape = K.int_shape(x)

# Преобразуем многомерный массив в вектор
x = Flatten()(x)
latent = Dense(latent_dim, name='latent_vector')(x)
# Итоговый код - одномерный вектор меньшей размерности (latent_dim)

# Итоговая модель энкодера:
encoder = Model(encoder_inputs , latent, name='encoder')
```

Распечатать краткую сводку по нашей модели можно командой encoder.summary(). Если всё было сделано без ошибок, то должно отобразиться:

```
Model: "encoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
```

```
encoder_input (InputLayer)   (None, 28, 28, 1)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 14, 14, 32)        320
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 7, 7, 64)          18496
_________________________________________________________________
flatten_1 (Flatten)          (None, 3136)              0
_________________________________________________________________
latent_vector (Dense)        (None, 16)                50192
=================================================================
Total params: 69,008
Trainable params: 69,008
Non-trainable params: 0
_________________________________________________________________
```

Аналогичным образом построим модель декодера, задействовав информацию о размере (shape) последнего слоя свертки.

```
latent_inputs = Input(shape=(latent_dim,), name='decoder_input')
# Обратное преобразование к размеру "shape":
x = Dense(shape[1] * shape[2] * shape[3])(latent_inputs)
# Выход должен быть трехмерным массивом:
x = Reshape((shape[1], shape[2], shape[3]))(x)

# Вместо сверточных слоев "развероточные", цикл в обратном порядке:
for filters in layer_filters[::-1]:
    x = Conv2DTranspose(filters=filters,
                        kernel_size=kernel_size,
                        strides=2,
                        activation='relu',
                        padding='same')(x)

x = Conv2DTranspose(filters=1,
                    kernel_size=kernel_size,
                    padding='same')(x)

outputs = Activation('sigmoid', name='decoder_output')(x)

# Итоговая модель декодера:
decoder = Model(latent_inputs, outputs, name='decoder')
decoder.summary()
```

Объединив две модели в одну мы получим наш автоэнкодер:

```
autoencoder = Model(encoder_inputs,
                    decoder(encoder(encoder_inputs)),
                    name='autoencoder')
autoencoder.compile(loss='mse', optimizer='adam')
```

```
# Запускаем модель на обучение, используя незашумленные данные как эталонные.
autoencoder.fit(x_train_noisy,
                x_train,
                validation_data=(x_test_noisy, x_test),
                epochs=30,
                batch_size=batch_size)
```

Полученную модель по окончанию ее обучения можно также сохранить с помощью функции model.save.

§ 3. Практические задания

Задание № 1: Классификация изображений.

1. Выбрать один из стандартных датасетов для классификации изображений (совпадение датасетов допускается у 2-3 студентов).
2. Загрузить данные, разбив изображения на обучающую и тестовую выборки. Предусмотреть аугментацию данных.
3. Импортировать сверточную сеть и адаптировать для обработки и распознания классов изображений из выбранного датасета.
4. Обучить сеть с заморозкой предобученных слоев, а затем разморозить их и повторить обучение с меньшим числом эпох.
5. Сравнить качество и скорость обучения при использовании заморозки коэффициентов предобученных слоев и без заморозки.

Задание № 2: Автоэнкодеры.

1. Доработать модель из разобранного примера, педусмотрев визуализацию итогового результата.
2. Вывести в виде изображений зашумленные и исправленные данные.
3. Реализовать альтернативную модель автоэнкодера с тремя слоями свертки/развертки. Сравнить качество работы с моделью из примера.