**Лабораторная работа № 2**

**«Свёрточные сети и автоэнкодеры»**

**Задание № 1:** Классификация изображений

Свёрточная нейронная сеть должна проверять изображения на принадлежность одному из 10 классов: горы, коты, цветы, деревья, звёзды, камни, ковры, машины, самолёты, хомяки.

Реализация на языке Python:

from tensorflow import keras

from tensorflow.keras.applications import MobileNet

from tensorflow.keras.applications.mobilenet import preprocess\_input

from tensorflow.keras.preprocessing.image import ImageDataGenerator

from tensorflow.keras.models import load\_model

train\_datagen=ImageDataGenerator(preprocessing\_function=preprocess\_input)

train\_generator=train\_datagen.flow\_from\_directory('./Train/', target\_size=(224,224), color\_mode='rgb', batch\_size=30, class\_mode='categorical', shuffle=True)

valid\_datagen=ImageDataGenerator(preprocessing\_function=preprocess\_input)

valid\_generator=valid\_datagen.flow\_from\_directory('./Valid/', target\_size=(224,224), color\_mode='rgb', batch\_size=1, class\_mode='categorical', shuffle=True)

from tensorflow.keras.layers import Dense,GlobalAveragePooling2D

from tensorflow.keras.layers import Activation

from tensorflow.keras.models import Model

base\_model=MobileNet(weights='imagenet',include\_top=False)

output\_model=base\_model.output

output\_model=GlobalAveragePooling2D()(output\_model)

output\_model=Dense(1024,activation='relu')(output\_model)

output\_model=Dense(512,activation='relu')(output\_model)

numClasses = 10

output\_model=Dense(numClasses,activation='softmax')(output\_model)

for layer in base\_model.layers:

layer.trainable=False

model=Model(inputs=base\_model.input,outputs=output\_model)

model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])

step\_size\_train=train\_generator.n//train\_generator.batch\_size

step\_size\_valid=valid\_generator.n//valid\_generator.batch\_size

model.fit\_generator(generator=train\_generator, steps\_per\_epoch=step\_size\_train, validation\_data=valid\_generator, validation\_steps =step\_size\_valid, epochs=10)

model.save('our\_first\_convmodel2.h5')

model = load\_model('our\_first\_convmodel2.h5')

model.compile(loss='categorical\_crossentropy', metrics=['accuracy'])

import numpy as np

def answer(y):

a = max(y[0])

for i in range(len(y[0])):

if (a == y[0][i]):

j = i

break

if (j == 0):

print('Это гора')

elif (j == 1):

print('Это котик')

elif (j == 2):

print('Это цветы')

elif (j == 3):

print('Это дерево')

elif (j == 4):

print('Это звёзды')

elif (j == 5):

print('Это камни')

elif (j == 6):

print('Это ковёр')

elif (j == 7):

print('Это машина')

elif (j == 8):

print('Это самолёт')

elif (j == 9):

print('Это хомяк')

print(j)

valid\_datagen=ImageDataGenerator(preprocessing\_function=preprocess\_input)

valid\_generator=valid\_datagen.flow\_from\_directory('./Proverka/', target\_size=(224,224), color\_mode='rgb', batch\_size=1, class\_mode='categorical', shuffle=True)

y = model.predict(valid\_generator)

answer(y)

Результаты обучения с замороженными весовыми коэффициентами базового слоя:

Found 100 images belonging to 10 classes.

Found 30 images belonging to 10 classes.

Epoch 1/10

3/3 [==============================] - 46s 15s/step - loss: 2.1783 - accuracy: 0.3444 - val\_loss: 0.8764 - val\_accuracy: 0.7333

Epoch 2/10

3/3 [==============================] - 27s 9s/step - loss: 1.1211 - accuracy: 0.7200 - val\_loss: 0.5084 - val\_accuracy: 0.7667

Epoch 3/10

3/3 [==============================] - 45s 15s/step - loss: 0.2772 - accuracy: 0.8778 - val\_loss: 0.1555 - val\_accuracy: 0.9667

Epoch 4/10

3/3 [==============================] - 35s 12s/step - loss: 0.0865 - accuracy: 0.9571 - val\_loss: 0.2263 - val\_accuracy: 0.9000

Epoch 5/10

3/3 [==============================] - 35s 12s/step - loss: 0.1838 - accuracy: 0.9143 - val\_loss: 0.0968 - val\_accuracy: 0.9667

Epoch 6/10

3/3 [==============================] - 40s 13s/step - loss: 0.1034 - accuracy: 0.9286 - val\_loss: 0.0715 - val\_accuracy: 0.9667

Epoch 7/10

3/3 [==============================] - 36s 12s/step - loss: 0.0568 - accuracy: 0.9714 - val\_loss: 0.1219 - val\_accuracy: 0.9333

Epoch 8/10

3/3 [==============================] - 44s 15s/step - loss: 0.0165 - accuracy: 1.0000 - val\_loss: 0.0775 - val\_accuracy: 0.9667

Epoch 9/10

3/3 [==============================] - 38s 13s/step - loss: 0.0083 - accuracy: 1.0000 - val\_loss: 0.0398 - val\_accuracy: 0.9667

Epoch 10/10

3/3 [==============================] - 45s 15s/step - loss: 0.0050 - accuracy: 1.0000 - val\_loss: 0.0221 - val\_accuracy: 1.0000

Результаты обучения с размороженными весовыми коэффициентами базового слоя:

Found 100 images belonging to 10 classes.

Found 30 images belonging to 10 classes.

Epoch 1/10

3/3 [==============================] - 44s 15s/step - loss: 2.1243 - accuracy: 0.3000 - val\_loss: 1.0191 - val\_accuracy: 0.7667

Epoch 2/10

3/3 [==============================] - 44s 15s/step - loss: 0.4603 - accuracy: 0.8556 - val\_loss: 5.1239 - val\_accuracy: 0.2333

Epoch 3/10

3/3 [==============================] - 35s 12s/step - loss: 0.1526 - accuracy: 0.9429 - val\_loss: 2.0047 - val\_accuracy: 0.4667

Epoch 4/10

3/3 [==============================] - 35s 12s/step - loss: 0.1549 - accuracy: 0.9286 - val\_loss: 0.5345 - val\_accuracy: 0.7667

Epoch 5/10

3/3 [==============================] - 47s 16s/step - loss: 0.0642 - accuracy: 0.9889 - val\_loss: 1.4699 - val\_accuracy: 0.5333

Epoch 6/10

3/3 [==============================] - 37s 12s/step - loss: 0.0044 - accuracy: 1.0000 - val\_loss: 2.9173 - val\_accuracy: 0.4667

Epoch 7/10

3/3 [==============================] - 36s 12s/step - loss: 0.0381 - accuracy: 1.0000 - val\_loss: 2.3134 - val\_accuracy: 0.6333

Epoch 8/10

3/3 [==============================] - 35s 12s/step - loss: 0.0033 - accuracy: 1.0000 - val\_loss: 4.7614 - val\_accuracy: 0.5333

Epoch 9/10

3/3 [==============================] - 35s 12s/step - loss: 0.2301 - accuracy: 0.9714 - val\_loss: 1.9267 - val\_accuracy: 0.8333

Epoch 10/10

3/3 [==============================] - 36s 12s/step - loss: 0.3053 - accuracy: 0.9571 - val\_loss: 2.4058 - val\_accuracy: 0.7667

Можно сначала обучить сеть с замороженными коэффициентами базового слоя, а потом разморозить их и провести ещё несколько эпох обучения. Таким образом, можно увеличить точность нейронной сети и уменьшить значение функции ошибки.

Реализация на языке Python:

for layer in base\_model.layers:

layer.trainable=False

model=Model(inputs=base\_model.input,outputs=output\_model)

model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])

step\_size\_train=train\_generator.n//train\_generator.batch\_size

step\_size\_valid=valid\_generator.n//valid\_generator.batch\_size

model.fit\_generator(generator=train\_generator, steps\_per\_epoch=step\_size\_train, validation\_data=valid\_generator, validation\_steps =step\_size\_valid, epochs=10)

for layer in base\_model.layers:

layer.trainable=True

model.fit\_generator(generator=train\_generator, steps\_per\_epoch=step\_size\_train, validation\_data=valid\_generator, validation\_steps =step\_size\_valid, epochs=5)

Результаты обучения:

Found 100 images belonging to 10 classes.

Found 30 images belonging to 10 classes.

Epoch 1/10

3/3 [==============================] - 46s 15s/step - loss: 1.6677 - accuracy: 0.4222 - val\_loss: 0.6358 - val\_accuracy: 0.8333

Epoch 2/10

3/3 [==============================] - 35s 12s/step - loss: 0.7438 - accuracy: 0.8714 - val\_loss: 0.1599 - val\_accuracy: 1.0000

Epoch 3/10

3/3 [==============================] - 35s 12s/step - loss: 0.1198 - accuracy: 0.9714 - val\_loss: 0.0679 - val\_accuracy: 1.0000

Epoch 4/10

3/3 [==============================] - 35s 12s/step - loss: 0.0950 - accuracy: 0.9857 - val\_loss: 0.2427 - val\_accuracy: 0.9333

Epoch 5/10

3/3 [==============================] - 34s 11s/step - loss: 0.0829 - accuracy: 0.9571 - val\_loss: 0.0478 - val\_accuracy: 0.9667

Epoch 6/10

3/3 [==============================] - 36s 12s/step - loss: 0.0102 - accuracy: 1.0000 - val\_loss: 0.0106 - val\_accuracy: 1.0000

Epoch 7/10

3/3 [==============================] - 44s 15s/step - loss: 0.0028 - accuracy: 1.0000 - val\_loss: 0.0139 - val\_accuracy: 1.0000

Epoch 8/10

3/3 [==============================] - 35s 12s/step - loss: 0.0492 - accuracy: 0.9857 - val\_loss: 0.0140 - val\_accuracy: 1.0000

Epoch 9/10

3/3 [==============================] - 43s 14s/step - loss: 0.0076 - accuracy: 1.0000 - val\_loss: 0.0396 - val\_accuracy: 1.0000

Epoch 10/10

3/3 [==============================] - 26s 9s/step - loss: 0.1285 - accuracy: 0.9800 - val\_loss: 0.0118 - val\_accuracy: 1.0000

Epoch 1/5

3/3 [==============================] - 35s 12s/step - loss: 2.9064e-04 - accuracy: 1.0000 - val\_loss: 0.0162 - val\_accuracy: 1.0000

Epoch 2/5

3/3 [==============================] - 40s 13s/step - loss: 0.0440 - accuracy: 0.9714 - val\_loss: 0.0478 - val\_accuracy: 0.9667

Epoch 3/5

3/3 [==============================] - 44s 15s/step - loss: 0.0026 - accuracy: 1.0000 - val\_loss: 0.0135 - val\_accuracy: 1.0000

Epoch 4/5

3/3 [==============================] - 35s 12s/step - loss: 1.1494e-04 - accuracy: 1.0000 - val\_loss: 0.0233 - val\_accuracy: 1.0000

Epoch 5/5

3/3 [==============================] - 43s 14s/step - loss: 1.1844e-04 - accuracy: 1.0000 - val\_loss: 0.0643 - val\_accuracy: 0.9667

**Задание № 1:** Автоэнкодеры

Реализация автоэнкодера для фильтрации нормального шума с математическим ожиданием 0.5 и дисперсией 0.5.

from \_\_future\_\_ import absolute\_import

from \_\_future\_\_ import division

import keras

from keras.datasets import mnist

import numpy as np

from tensorflow.keras.models import load\_model

(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

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.)

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]

from tensorflow.keras.layers import Activation, Dense, Input

from tensorflow.keras.layers import Conv2D, Flatten

from tensorflow.keras.layers import Reshape, Conv2DTranspose

from tensorflow.keras.models import Model

from tensorflow.keras import backend as K

import matplotlib.pyplot as plt

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')

encoder.summary()

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 "разверточные", 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 значения.

autoencoder.fit(x\_train\_noisy, x\_train, validation\_data=(x\_test\_noisy, x\_test), epochs=10, batch\_size=batch\_size)

autoencoder.save('autoencoder.h5')

model = load\_model('autoencoder.h5')

y = autoencoder.predict(x\_test\_noisy)

data\_x = np.reshape(x\_test\_noisy, [-1, image\_size, image\_size])

data\_y = np.reshape(y, [-1, image\_size, image\_size])

for i in range(1,10):

fig, ax = plt.subplots()

ax.imshow(data\_x[i])

fig.set\_figwidth(5) # ширина и

fig.set\_figheight(5) # высота "Figure"

plt.show()

fig, ax = plt.subplots()

ax.imshow(data\_y[i])

fig.set\_figwidth(5) # ширина и

fig.set\_figheight(5) # высота "Figure"

plt.show()

Результаты обучения:

Train on 60000 samples, validate on 10000 samples

Epoch 1/10

60000/60000 [==============================] - 118s 2ms/sample - loss: 0.0560 - val\_loss: 0.0252

Epoch 2/10

60000/60000 [==============================] - 109s 2ms/sample - loss: 0.0219 - val\_loss: 0.0194

Epoch 3/10

60000/60000 [==============================] - 110s 2ms/sample - loss: 0.0188 - val\_loss: 0.0179

Epoch 4/10

60000/60000 [==============================] - 110s 2ms/sample - loss: 0.0175 - val\_loss: 0.0170

Epoch 5/10

60000/60000 [==============================] - 110s 2ms/sample - loss: 0.0167 - val\_loss: 0.0166

Epoch 6/10

60000/60000 [==============================] - 110s 2ms/sample - loss: 0.0161 - val\_loss: 0.0162

Epoch 7/10

60000/60000 [==============================] - 110s 2ms/sample - loss: 0.0157 - val\_loss: 0.0159

Epoch 8/10

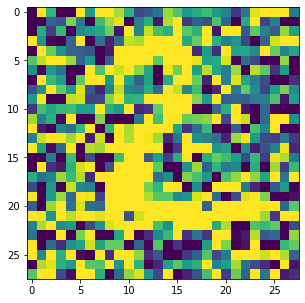
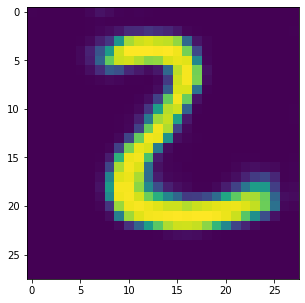
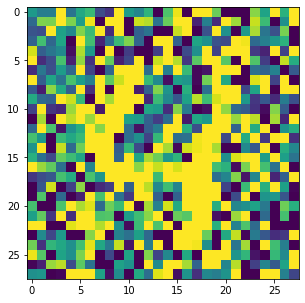
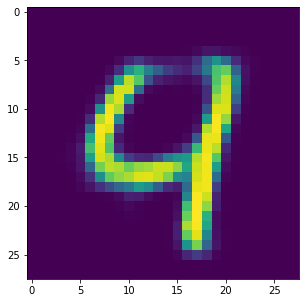
60000/60000 [==============================] - 112s 2ms/sample - loss: 0.0153 - val\_loss: 0.0158

Epoch 9/10

60000/60000 [==============================] - 114s 2ms/sample - loss: 0.0150 - val\_loss: 0.0159

Epoch 10/10

60000/60000 [==============================] - 110s 2ms/sample - loss: 0.0148 - val\_loss: 0.0156

Результаты работы нейросети – изображения до и после фильтрации соответственно:

Добавив в сеть ещё по одному свёрточному и развёрточному слоям со 128 нейронами в каждом, можно уменьшить значения функции ошибки и улучшить восприятие цифр на изображениях.

Реализация на языке Python:

input\_shape = (image\_size, image\_size, 1)

batch\_size = 128

kernel\_size = [3, 3, 1]

latent\_dim = 16

# Зададим два сверточных слоя и число нейронов в каждом слое:

layer\_filters = [32, 64, 128]

kern\_stride = [2, 2, 1]

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 i in range(len(layer\_filters)):

x = Conv2D(filters=layer\_filters[i], kernel\_size=kernel\_size[i], strides=kern\_stride[i], 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')

encoder.summary()

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 "разверточные", цикл в обратном порядке:

for i in range(len(layer\_filters) - 1,-1,-1):

print(i)

x = Conv2DTranspose(filters=layer\_filters[i], kernel\_size=kernel\_size[i], strides=kern\_stride[i], activation='relu', padding='same')(x)

x = Conv2DTranspose(filters=1, kernel\_size=kernel\_size[0], 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 значения.

autoencoder.fit(x\_train\_noisy, x\_train, validation\_data=(x\_test\_noisy, x\_test), epochs=10, batch\_size=batch\_size)

autoencoder.save('autoencoder2.h5')

model = load\_model('autoencoder2.h5')

y = autoencoder.predict(x\_test\_noisy)

data\_x = np.reshape(x\_test\_noisy, [-1, image\_size, image\_size])

data\_y = np.reshape(y, [-1, image\_size, image\_size])

for i in range(1,10):

fig, ax = plt.subplots()

ax.imshow(data\_x[i])

fig.set\_figwidth(5) # ширина и

fig.set\_figheight(5) # высота "Figure"

plt.show()

fig, ax = plt.subplots()

ax.imshow(data\_y[i])

fig.set\_figwidth(5) # ширина и

fig.set\_figheight(5) # высота "Figure"

plt.show()

Результаты обучения:

Train on 60000 samples, validate on 10000 samples

Epoch 1/10

60000/60000 [==============================] - 118s 2ms/sample - loss: 0.0560 - val\_loss: 0.0252

Epoch 2/10

60000/60000 [==============================] - 109s 2ms/sample - loss: 0.0219 - val\_loss: 0.0194

Epoch 3/10

60000/60000 [==============================] - 110s 2ms/sample - loss: 0.0188 - val\_loss: 0.0179

Epoch 4/10

60000/60000 [==============================] - 110s 2ms/sample - loss: 0.0175 - val\_loss: 0.0170

Epoch 5/10

60000/60000 [==============================] - 110s 2ms/sample - loss: 0.0167 - val\_loss: 0.0166

Epoch 6/10

60000/60000 [==============================] - 110s 2ms/sample - loss: 0.0161 - val\_loss: 0.0162

Epoch 7/10

60000/60000 [==============================] - 110s 2ms/sample - loss: 0.0157 - val\_loss: 0.0159

Epoch 8/10

60000/60000 [==============================] - 112s 2ms/sample - loss: 0.0153 - val\_loss: 0.0158

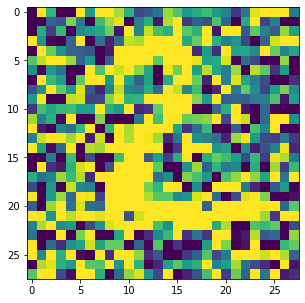
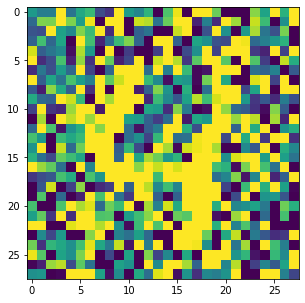
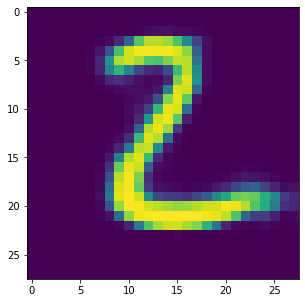
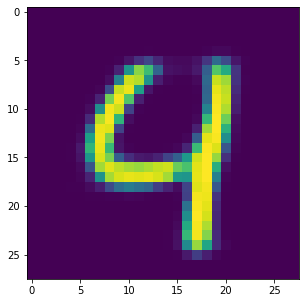
Epoch 9/10

60000/60000 [==============================] - 114s 2ms/sample - loss: 0.0150 - val\_loss: 0.0159

Epoch 10/10

60000/60000 [==============================] - 110s 2ms/sample - loss: 0.0148 - val\_loss: 0.0156

Результаты работы нейросети – изображения до и после фильтрации соответственно:



Сравнив с результатами работы сети с двумя свёрточными и двумя развёрточными слоями, можно видеть, что визуально изображения после фильтрации стали менее зашумлёнными, а контуры цифр – более чёткими.