Министерство образования Республики Беларусь

Учреждение образования

БелорусскиЙ государственный университет

информатики и радиоэлектроники

Факультет компьютерных систем и сетей

Кафедра информатики

|  |
| --- |
|  |

**ЛабоРАТОРНАЯ РАБОТА №6**

«Применение свёрточных нейронных сетей (многоклассовая классификация)»

|  |  |  |
| --- | --- | --- |
| Студент |  | К. Д. Зюсько |
| Преподаватель |  | М. В. Стержанов |

Минск 2020

ХОД РАБОТЫ

**Задание.**

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, как показано на рисунке ниже (рисунок цветной, а 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

1. Загрузите данные. Разделите исходный набор данных на обучающую и валидационную выборки.
2. Реализуйте глубокую нейронную сеть со сверточными слоями. Какое качество классификации получено? Какая архитектура сети была использована?
3. Примените дополнение данных (data augmentation). Как это повлияло на качество классификатора?
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?

**Результат выполнения:**

Поскольку в данной лабораторной предлагается сравнить результаты трёх различных конфигураций нейронных сетей, я установил количество эпох для обучение равным 30, чтобы условия, в которых происходит сравнение, были одинаковыми.

1. Сначала я посмотрел на распределение данных:

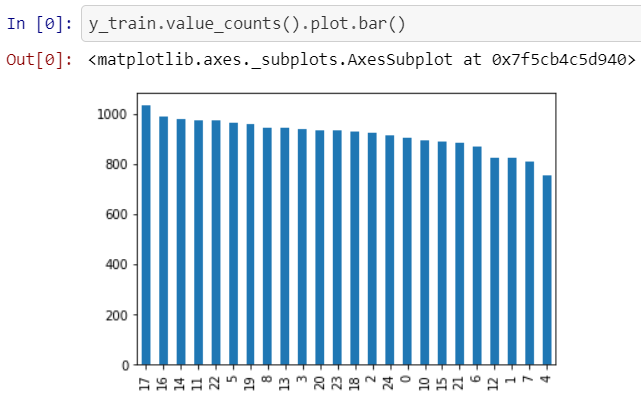
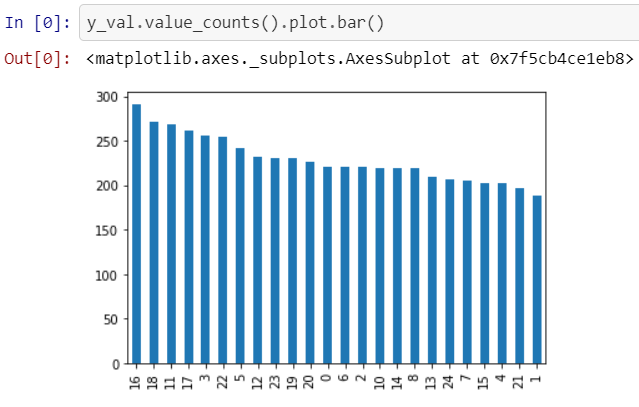
:

Рисунок 1 – распределение данных на трёх выборках

Как видно на рисунках выше все данные распределены неравномерно, что особенно заметно на тестовой выборке. Код реализации:

!mkdir ~/.kaggle

!cp /content/.kaggle/kaggle.json ~/.kaggle/kaggle.json

!kaggle datasets download datamunge/sign-language-mnist -p /content/kaggle/signlanguage

!unzip kaggle/signlanguage/sign-language-mnist.zip -d data

import pandas as pd

import os

from sklearn.model\_selection import train\_test\_split

TRAIN\_PATH = './data/sign\_mnist\_train.csv'

TEST\_PATH = './data/sign\_mnist\_test.csv'

IMAGE\_WIDTH=28

IMAGE\_HEIGHT=28

IMAGE\_SIZE=(IMAGE\_WIDTH, IMAGE\_HEIGHT)

IMAGE\_CHANNELS=1

train = pd.read\_csv(TRAIN\_PATH)

test = pd.read\_csv(TEST\_PATH)

# train, val = train\_test\_split(train, test\_size=0.2, random\_state=42)

def split\_dataset\_to\_x\_y(df, is\_numpy=True):

y = pd.concat([df['label'], pd.get\_dummies(df['label'], prefix='label')], axis=1)

df.drop(['label'], axis=1, inplace=True)

y.drop(['label'], axis=1, inplace=True)

X = (df / 255).values.reshape((df.shape[0], IMAGE\_HEIGHT, IMAGE\_WIDTH, IMAGE\_CHANNELS)) # data scaling & reshaping

return X, y.to\_numpy() if is\_numpy else y

X, Y = split\_dataset\_to\_x\_y(train)

X\_test, Y\_test = split\_dataset\_to\_x\_y(test, is\_numpy=False)

LABELS = 24

Чтобы результаты сравнения были в относительно одинаковых условия, во время обучения я использовал KFold кросс-валидацию.

2. В качестве архитектуры нейронной сети я использовал ту же самую, как и в третьей лабораторной, потому что данная архитектура показывала очень хорошие результаты в задачах распознавания букв:

from keras.models import Sequential

from keras.layers import Conv2D, MaxPooling2D, Dropout, Flatten, Dense, Activation, BatchNormalization

model = Sequential()

model.add(Conv2D(32, (3, 3), activation='relu', input\_shape=(IMAGE\_WIDTH, IMAGE\_HEIGHT, IMAGE\_CHANNELS)))

model.add(BatchNormalization())

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), activation='relu'))

model.add(BatchNormalization())

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Dropout(0.25))

model.add(Conv2D(128, (3, 3), activation='relu'))

model.add(BatchNormalization())

model.add(MaxPooling2D(pool\_size=(2, 2)))

model.add(Dropout(0.25))

model.add(Flatten())

model.add(Dense(512, activation='relu'))

model.add(BatchNormalization())

model.add(Dropout(0.5))

model.add(Dense(LABELS, activation='softmax')) # 24 classes

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

model.summary()

model.save\_weights('initial.h5')

from keras.preprocessing.image import ImageDataGenerator

from sklearn.model\_selection import KFold

n\_folds = 5

skf = KFold(n\_folds, shuffle=True).split(X)

datagen = ImageDataGenerator()

datagen.fit(X)

# compute quantities required for featurewise normalization

# (std, mean, and principal components if ZCA whitening is applied)

models = []

results = []

results\_val = []

results\_train = []

# fits the model on batches with real-time data augmentation:

for train\_index, test\_index in skf:

model.load\_weights('initial.h5')

x\_train,x\_test=X[train\_index],X[test\_index]

y\_train,y\_test=Y[train\_index],Y[test\_index]

h = model.fit\_generator(

datagen.flow(x\_train, y\_train, batch\_size=32),

steps\_per\_epoch=len(x\_train) / 32,

validation\_data=datagen.flow(x\_test, y\_test),

validation\_steps=len(x\_test) / 32,

epochs=30,

verbose=0

)

score, acc = model.evaluate(x\_test, y\_test, verbose=1)

# models.append(model)

results\_val.append([score, acc])

score, acc = model.evaluate(x\_train, y\_train, verbose=1)

# models.append(model)

results\_train.append([score, acc])

score, acc = model.evaluate(X\_test, Y\_test, verbose=1)

# models.append(model)

results.append([score, acc])

print(results)

print(results\_val)

print(results\_train)

Результаты обучения приведены в таблице ниже.

Таблица 1 – результаты обучения

|  |  |  |
| --- | --- | --- |
|  | Loss | Accuracy |
| Тренировочный датасет | 3.95e-07 | 100% |
| Валидационный датасет | 0.1763 | 96.45% |

Как можно видеть выше, ошибка на валидационном датасете в несколько раз больше, чем на тренировочном. Стоит также учитывать тот факт, что в таблице отображены лучшие результаты из KFold. В худшем случае разбежка (между тренировочным и валидационными датасетами) была около 8% по accuracy (и составляла 92.26%) и в несколько сотен раз больше по loss (составляла 0.46), хотя в то же время, ошибка на тренировочном датасете отличалась незначительно от того, что приведено в таблице выше. Потому здесь очевидна проблема high variance.

3. Все дополнительные препроцессинги можно увидеть ниже. Аугментирование данных применялось только к тренировочному датасету:

datagen = ImageDataGenerator(

shear\_range=0.2,

zoom\_range=0.2,

rotation\_range=20,

)

# compute quantities required for featurewise normalization

# (std, mean, and principal components if ZCA whitening is applied)

datagen.fit(X)

skf = KFold(n\_folds, shuffle=True).split(X)

models = []

results = []

results\_val = []

results\_train = []

# fits the model on batches with real-time data augmentation:

for train\_index, test\_index in skf:

model.load\_weights('initial.h5')

x\_train,x\_test=X[train\_index],X[test\_index]

y\_train,y\_test=Y[train\_index],Y[test\_index]

h = model.fit\_generator(

datagen.flow(x\_train, y\_train, batch\_size=32),

steps\_per\_epoch=len(x\_train) / 32,

validation\_data=datagen.flow(x\_test, y\_test),

validation\_steps=len(x\_test) / 32,

epochs=30,

verbose=0

)

score, acc = model.evaluate(x\_test, y\_test, verbose=1)

# models.append(model)

results\_val.append([score, acc])

score, acc = model.evaluate(x\_train, y\_train, verbose=1)

# models.append(model)

results\_train.append([score, acc])

score, acc = model.evaluate(X\_test, Y\_test, verbose=1)

# models.append(model)

results.append([score, acc])

print(results)

print(results\_val)

print(results\_train)

Таблица 2 – результаты обучения

|  |  |  |
| --- | --- | --- |
|  | Loss | Accuracy |
| Тренировочный датасет | 0.00014 | 100% |
| Валидационный датасет | 0.0843 | 98.38% |

Из таблицы выше видно, что ошибки отличаются друг от друга, но не с таким большим разбегом как раньше, что свидетельствует о том, что проблема high variance решена, как и ожидалось, увеличением количества данных, необходимых для обучения (аугментированием данных в данном случае).

4. Здесь я также постарался создать те же условия, как и везде, поэтому полносвязный слой состоит из 512 нейронов, и также имеет дропаут слой с коэффицентом 0.5. Минимальный размер для VGG16 составляет 32х32, наши картинки имеют разрешение 28х28, также VGG16 тренировалась на цветных изображениях, наши же картинки чернобелые. Чтобы нивелировать эту разницу я увеличил их размер с помощью cv2.resize, и значения одного монохромного канала клонировал в каждый канал RGB. Код реализации:

import numpy as np

import cv2

def x\_to\_vgg\_input(X):

x = np.reshape(X, (X.shape[0], 28, 28))

x = np.asarray([cv2.resize(i, (32,32)) for i in x], dtype=np.float32)

x = np.repeat(x[..., np.newaxis], 3, -1)

return x

from keras.models import Model

from keras.applications.vgg16 import VGG16

from keras.layers import Conv2D, MaxPooling2D, Dropout, Flatten, Dense, Activation, BatchNormalization

# load model

vgg = VGG16(include\_top=False, input\_shape=(32, 32, 3))

# mark loaded layers as not trainable

for layer in vgg.layers:

layer.trainable = False

# define a new output layer to connect with the last fc layer in vgg

x = Flatten()(vgg.layers[-1].output)

class1 = Dense(512, activation='relu', kernel\_initializer='he\_uniform')(x)

bn = BatchNormalization()(class1)

do = Dropout(0.5)(bn)

output\_layer = Dense(LABELS, activation='softmax', name='predictions')(do)

# combine the original VGG model with the new output layer

vgg = Model(inputs=vgg.input, outputs=output\_layer)

vgg.compile(loss='categorical\_crossentropy', optimizer='rmsprop', metrics=['acc'])

vgg.save\_weights('vgg.h5')

# fits the model on batches with real-time data augmentation:

skf = KFold(n\_folds, shuffle=True).split(X)

models = []

results = []

results\_train = []

# fits the model on batches with real-time data augmentation:

for train\_index, test\_index in skf:

x\_train,x\_test=X[train\_index],X[test\_index]

y\_train,y\_test=Y[train\_index],Y[test\_index]

vgg.load\_weights('vgg.h5')

vgg.fit\_generator(

datagen.flow(x\_to\_vgg\_input(x\_train), y\_train, batch\_size=32),

steps\_per\_epoch=len(x\_train) / 32,

validation\_data=datagen.flow(x\_to\_vgg\_input(x\_test), y\_test),

validation\_steps=len(x\_test) / 32,

epochs=30,

verbose=0

)

score, acc = vgg.evaluate(x\_to\_vgg\_input(X\_test), Y\_test, verbose=1)

# models.append(model)

results.append([score, acc])

score, acc = model.evaluate(x\_to\_vgg\_input(x\_train), y\_train, verbose=1)

# models.append(model)

results\_train.append([score, acc])

score, acc = model.evaluate(x\_to\_vgg\_input(x\_test), y\_test, verbose=1)

# models.append(model)

results\_val.append([score, acc])

print(results)

print(results\_val)

print(results\_train)

Таблица 3 – результаты обучения

|  |  |  |
| --- | --- | --- |
|  | Loss | Accuracy |
| Тренировочный датасет | 0.2620 | 88.91% |
| Валидационный датасет | 0.2261 | 90.20% |

Как видно выше, результаты, практически, не изменились, по сравнению со вторым пунктом. Есть небольшие различия, однако, это можно списать на погрешность в обучении.

Но в сравнении с предыдущей конфигурации нейронной сети, если посмотреть на историю обучения, то можно заметить, что вариант с транферным обучением «обучается» быстрее, а именно ошибка после первой эпохи значительно меньше, чем в предыдущем случае. Объясняется это тем, что мы использовали предобученную конфигурацию нейронной сети, которая обучалась на других изображения, но конволюционные слои запомнили «фичи» изображений, что и сказалось на скорости уменьшения ошибки.

Загрузить предсказанные