ФГАОУ ВО «СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»
Институт космических и информационных технологий
Кафедра «Информатика»

Компьютерный статистический анализ данных

Практическая работа №4
Программная реализация сверточных нейронных сетей

Красноярск, 2021

Цель: изучение архитектур сверточных нейронных сетей, технологии переноса обучения; создание и исследование моделей сверточных нейронных сетей для задачи медицинской диагностики.

Исходные данные:

Медицинские изображения, представленные в открытом доступе (МРТ-снимки, КТ-снимки, рентгенограммы, флюорограммы и др.).

Искусственно уменьшить размер исходных данных до объема (2500-3000 изображений) для увеличения скорости обучения. Например, при решении задачи многоклассовой классификации ограничиться двумя классами. Исходная выборка должна быть выбрана по возможности со сбалансированным числом классов.

Общая последовательность действий

1. Изучение принципа работы сверточных нейронных сетей.
2. Проектирование и программная реализация моделей сверточных нейронных сетей. Блоки сверточных нейронных сетей

2.1 Разработать различные архитектуры сверточных нейронных сетей. Выполнить подбор гиперпараметров данных моделей:

- количество и комбинация Conv2D и MaxPooling2D слоев;

- количество каналов;

- kernel_size - размер ядра Conv2D слоя: [3-6];

- pool_size - размер окна MaxPooling2D слоя: [2-3];

- padding = same;

- stride = 1;

- kernel_initializer и bias_initializer [he_uniform, he_normal, glorot_uniform, glorot_normal, orthogonal];

- функция активации;

- метод оптимизации;

- количество слоев и нейронов для полносвязной сети.

Выбрать наилучшую сверточную нейронную сеть по величине точности на валидационном/тестирующем множестве.

2.2 Разработать различные архитектуры сверточных нейронных сетей с использованием различных блоков (vgg block, inception block, residual block, dense block).

Выполнить подбор гиперпараметров данных моделей. Выбрать наилучшую сверточную нейронную сеть по величине точности на валидационном/тестирующем множестве.

Пример кода создания каждого блока приведен в приложении.

Настройка гиперпараметров

Для подбора гиперпараметров сверточной нейронной сети допускается использование автоматических методов подбора параметров.

Для повышения точности прогноза и предотвращения возможности переобучения предусмотреть использование следующих методов:

- L-регуляризация ($L1$, $L2$, $L1 - L2$)
- Dropout, BatchNormalization
- Early stopping
- Техника изменения коэффициента скорости обучения нейронной сети при достижении условного «плато» точности $ReduceLROnPlateau$

3. Исследовать возможность переноса обучения для решения целевой задачи с использованием техники fine tuning.

Выполнить поиск сверточной нейронной сети (подходящей по смыслу к целевой задаче). Заморозить все слои предварительно обученной модели (кроме нескольких верхних слоев). Обучить добавленные слои. Обучить эти слои и добавленную часть вместе.

Пример кода, реализующий технику fine tuning, приведен в приложении.

Список доступным моделей приведен на данном сайте https://keras.io/api/applications/.

Требования к выполнению практической работы:

1. Написание программного кода и формирование результатов согласно заданию.

2. Составление отчета, содержащего описание решаемых задач методов решения и полученных результатов.

Программный код и отчет должны быть выполнены в среде Jupyter notebook. Отдельные блоки персептрона могут быть реализованы в виде программных модулей на языке Python.

Приложение

VGG block

```
# function for creating a vgg block
def vgg_block(layer_in, n_filters, n_conv):
	# add convolutional layers
	for _ in range(n_conv):
		layer_in = Conv2D(n_filters, (3,3), padding='same',
activation='relu')(layer_in)
	# add max pooling layer
	layer_in = MaxPooling2D((2,2), strides=(2,2))(layer_in)
	return layer_in

# define model input
visible = Input(shape=(256, 256, 3))
# add vgg module
layer = vgg_block(visible, 64, 2)
# create model
model = Model(inputs=visible, outputs=layer)
```

Inception block

```
# function for creating a inception block
def inception_module(layer_in, f1, f2_in, f2_out, f3_in, f3_out, f4_out):
	# 1x1 conv
	conv1 = Conv2D(f1, (1,1), padding='same', activation='relu')(layer_in)
	# 3x3 conv
	conv3 = Conv2D(f2_in, (1,1), padding='same', activation='relu')(layer_in)
	conv3 = Conv2D(f2_out, (3,3), padding='same', activation='relu')(conv3)
	# 5x5 conv
	conv5 = Conv2D(f3_in, (1,1), padding='same', activation='relu')(layer_in)
	conv5 = Conv2D(f3_out, (5,5), padding='same', activation='relu')(conv5)
	# 3x3 max pooling
	pool = MaxPooling2D((3,3), strides=(1,1), padding='same')(layer_in)
	pool = Conv2D(f4_out, (1,1), padding='same', activation='relu')(pool)
	# concatenate filters, assumes filters/channels last
	layer_out = concatenate([conv1, conv3, conv5, pool], axis=-1)
```

```
	return layer_out

# define model input
visible = Input(shape=(256, 256, 3))
# add inception module
layer = inception_module(visible, 64, 128, 32)
# create model
model = Model(inputs=visible, outputs=layer)
```

Residual block

```
# function for creating an identity or projection residual module
def residual_module(layer_in, n_filters):
	merge_input = layer_in
	# check if the number of filters needs to be increase, assumes channels last
format
	if layer_in.shape[-1] != n_filters:
		merge_input = Conv2D(n_filters, (1,1), padding='same', activation='relu',
kernel_initializer='he_normal')(layer_in)
	# conv1
	conv1 = Conv2D(n_filters, (3,3), padding='same', activation='relu',
kernel_initializer='he_normal')(layer_in)
	# conv2
	conv2 = Conv2D(n_filters, (3,3), padding='same', activation='linear',
kernel_initializer='he_normal')(conv1)
	# add filters, assumes filters/channels last
	layer_out = add([conv2, merge_input])
	# activation function
	layer_out = Activation('relu')(layer_out)
	return layer_out

# define model input
visible = Input(shape=(256, 256, 3))
# add vgg module
layer = residual_module(visible, 64)
# create model
model = Model(inputs=visible, outputs=layer)
```

Dense block

```
def dense_factor(layer_in):
    batcnorm = BatchNormalization()(layer_in)
    conv2 = Conv2D(n_filters, (3,3), padding='same')(batcnorm)
    layer_out = Activation('relu')(conv2)
    return layer_out

def dense_block(layer_in):
    concatenated_inputs = layer_in
    for _ in range(3):
        x = dense_factor(concatenated_inputs)
        concatenated_inputs = concatenate([concatenated_inputs, x], axis=3)
    return concatenated_inputs

# define model input
visible = Input(shape=(256, 256, 3))
# add dense module
layer = dense_block(visible)
# create model
```

```
model = Model(inputs=visible, outputs=layer)
```

Fine tuning

```
from tensorflow.keras.applications import vgg16

# Load VGG model
vgg_conv = vgg16.VGG16(weights='imagenet', include_top=False, input_shape=(image_size,
image_size, 3))

# Freeze CNN layers (except the last)
for layer in vgg_conv.layers:
    if layer.name in ['block5_conv1', 'block5_conv2', 'block5_conv3', 'block5_pool']:
        set_trainable = True
    else:
        layer.trainable = False

model = models.Sequential([
       vgg_conv,
       layers.Flatten(),
       layers.Dense(256, activation='relu'),
       layers.Dense(8, activation='softmax')
])
model.compile()
model.fit()
```