Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

**Иркутский национальный исследовательский   
технический университет**

|  |
| --- |
| Институт информационных технологий и анализа данных |
| наименование института |

|  |
| --- |
| Отчет по лабораторной работе №2  по дисциплине  «Программные средства для задач искусственного интеллекта» |
| «Разработка и тестирование моделей в TensorFlow 2» |
| наименование темы |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Выполнил студент группы |  | ИИТм-23-1 |  |  |  | Д.Д. Солопов |
|  |  | Шифр группы |  | Подпись |  | И.О. Фамилия |
| Проверил преподаватель |  |  |  |  |  | А.Б. Столбов |
|  |  |  |  | Подпись |  | И.О. Фамилия |

Иркутск 2024 г.

Содержание

[Постановка задач 3](#_Toc165506734)

[Решение задания №1 4](#_Toc165506735)

[Решение задания №2 10](#_Toc165506736)

[Решение задания №3 15](#_Toc165506737)

[Заключение 20](#_Toc165506738)

[Приложения 21](#_Toc165506739)

Постановка задач

Необходимо решить следующие задачи:

1. **Задание №1**:
2. Создать многослойную нейронную сеть с использованием функционального API TenzorFlow 2, имеющую структуру ориентированного ациклического графа (т.е. не последовательность слоёв как в keras.Sequential). Для объединения слоёв нужно использовать keras.layers.Concatenate. В результате должен получиться вариант графа с Y-DAG структурой.
3. Сеть должна иметь не менее 7-ми слоёв.
4. Сохранить модель в формате json.
5. **Задание №2**:
6. Создать собственную реализацию обратного вызова (keras.callbacks.Callback) и протестировать её для метода keras.Model.fit() сети из задания №1.
7. Создать собственную реализацию функции потерь (keras.losses.Loss) и метрики (tf.keras.metrics.Metric). Протестировать.
8. **Задание №3**:
9. Создать собственный слой (или слои) keras.layers и добавить этот слой (или заменить существующий) в сеть из задания 1.
10. Сохранить модель в формате json и pb.

Решение задания №1

Для создания многослойной нейронной сети, имеющей структуру ориентированного ациклического графа, будет использован фреймворк TensorFlow.

Сеть DAG - это нейронная сеть для глубокого обучения со слоями, расположенными в виде направленного ациклического графа. Сеть DAG может иметь более сложную архитектуру, в которой слои имеют входы с нескольких уровней и выходы на несколько уровней.

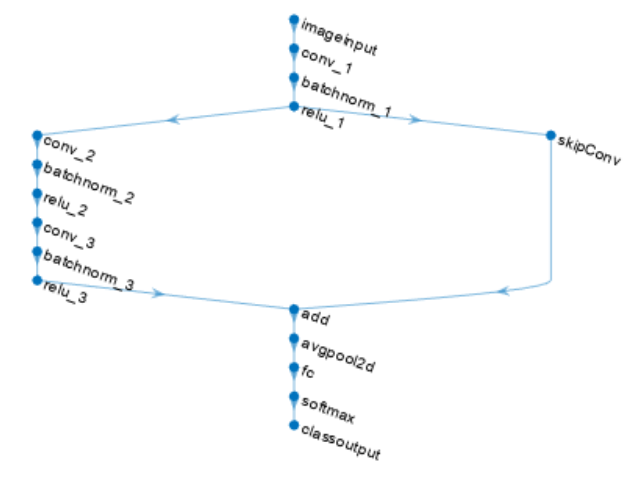


Рисунок 1 – Пример структуры DAG-сети

Для разработки Y-DAG сети, нам необходимо подключить следующие зависимости:

import numpy as np

import tensorflow as tf

from tensorflow.keras.layers import Dense, Dropout , Input , Flatten, concatenate , Conv2D, MaxPooling2D

from tensorflow.keras.models import Model

from tensorflow.keras.datasets import mnist

from tensorflow.keras.utils import to\_categorical

from tensorflow.keras.utils import plot\_model

Нейронная сеть будет выполнять задачу распознавания рукописных цифр из стандартного датасета MNIST, поэтому необходимо провести предварительную подготовку данных.

Для начала проведём загрузку датасета, определим число меток и преобразуем метки в формат one-hot-encoding:

# Загрузка тренировочного и тестового набора данных

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

# Определение числа меток

num\_labels = len(np.unique(y\_train))

# Конвертация меток в формат hot-encoding

y\_train = to\_categorical(y\_train)

y\_test = to\_categorical(y\_test)

Затем, проведём нормализацию датасета:

# Нормализация данных

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

После этого определяем конфигурацию нейронной сети:

# Определение конфигурации нейронной сети

# Размер входного изображения

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

# Размер пакета

batch\_size = 32

# Размер ядра свёртки

kernel\_size = 3

# Коэффициент отключения случайных нейронов

dropout = 0.4

# Число фильтров

n\_filters = 32

Y-DAG сеть имеет структуру, которая похожа на Y. В такой структуре есть две ветви с входными слоями – x и y. Сначала для этих ветвей собираются слои, а затем они объединяются с помощью метода concatenate и к результату объединения добавляются ещё слои. Таким образом данная нейронная сеть относится к классу Y-DAG.

Опишем слои левой ветви Y-DAG сети:

# Левая ветвь сети Y-DAG

filters = n\_filters

# Определение входного слоя для левой ветви

left\_inputs = Input(shape=input\_shape)

# Инициализация переменной x

x = left\_inputs

# Создание свёрточного слоя с числом фильтров 32

x = Conv2D(filters=filters, kernel\_size=kernel\_size, padding='same', activation='relu')(x)

# Отключение определённого числа случайных нейронов на случайных слоях

x = Dropout(dropout)(x)

# Слой для уменьшшения выборки входного слоя

x = MaxPooling2D()(x)

# Создание свёрточного слоя с числом фильтров 64

x = Conv2D(filters=(2 \* filters), kernel\_size=kernel\_size, padding='same', activation='relu')(x)

x = Dropout(dropout)(x)

x = MaxPooling2D()(x)

# Создание свёрточного слоя с числом фильтров 128

x = Conv2D(filters=(4 \* filters), kernel\_size=kernel\_size, padding='same', activation='relu')(x)

x = Dropout(dropout)(x)

x = MaxPooling2D()(x)

Опишем слои правой ветви Y-DAG сети:

# Правая ветвь сети Y-DAG

filters = n\_filters

# Определение входного слоя для правой ветви

right\_inputs = Input(shape=input\_shape)

y = right\_inputs

# Создание свёрточного слоя с числом фильтров 32

y = Conv2D(filters = filters,kernel\_size = kernel\_size,padding = 'same',

activation = 'relu',dilation\_rate =2)(y)

# Отключение определённого числа случайных нейронов на случайных слоях

y = Dropout(dropout)(y)

# Слой для уменьшшения выборки входного слоя

y = MaxPooling2D()(y)

# Создание свёрточного слоя с числом фильтров 64

y = Conv2D(filters = (2 \* filters),kernel\_size = kernel\_size,padding = 'same',

activation = 'relu',dilation\_rate =2)(y)

y = Dropout(dropout)(y)

y = MaxPooling2D()(y)

# Создание свёрточного слоя с числом фильтров 128

y = Conv2D(filters = (4 \* filters),kernel\_size = kernel\_size,padding = 'same',

activation = 'relu',dilation\_rate =2)(y)

y = Dropout(dropout)(y)

y = MaxPooling2D()(y)

Теперь, когда собрана левая и правая ветвь Y-DAG сети их можно объединить с помощью функции concatentate:

# Объединение ветвей нейронной сети

merge\_xy = concatenate([x, y])

Добавим к результату объединения левой и правой ветви ещё несколько слоёв:

# Добавление новых слоёв

# Слой сжатия многомерных тензоров в одно измерение

merge\_xy = Flatten()(merge\_xy)

merge\_xy = Dropout(dropout)(merge\_xy)

# Выходной слой

outputs = Dense(num\_labels , activation = 'softmax')(merge\_xy)

После чего можно собрать модель:

# Сборка модели нейронной сети

model = Model(inputs=[left\_inputs, right\_inputs], outputs=outputs)

На рисунке 2 представлена полная архитектура, получившийся Y-DAG нейронной сети. Все слои полностью соответствуют описанным блокам в программном коде.

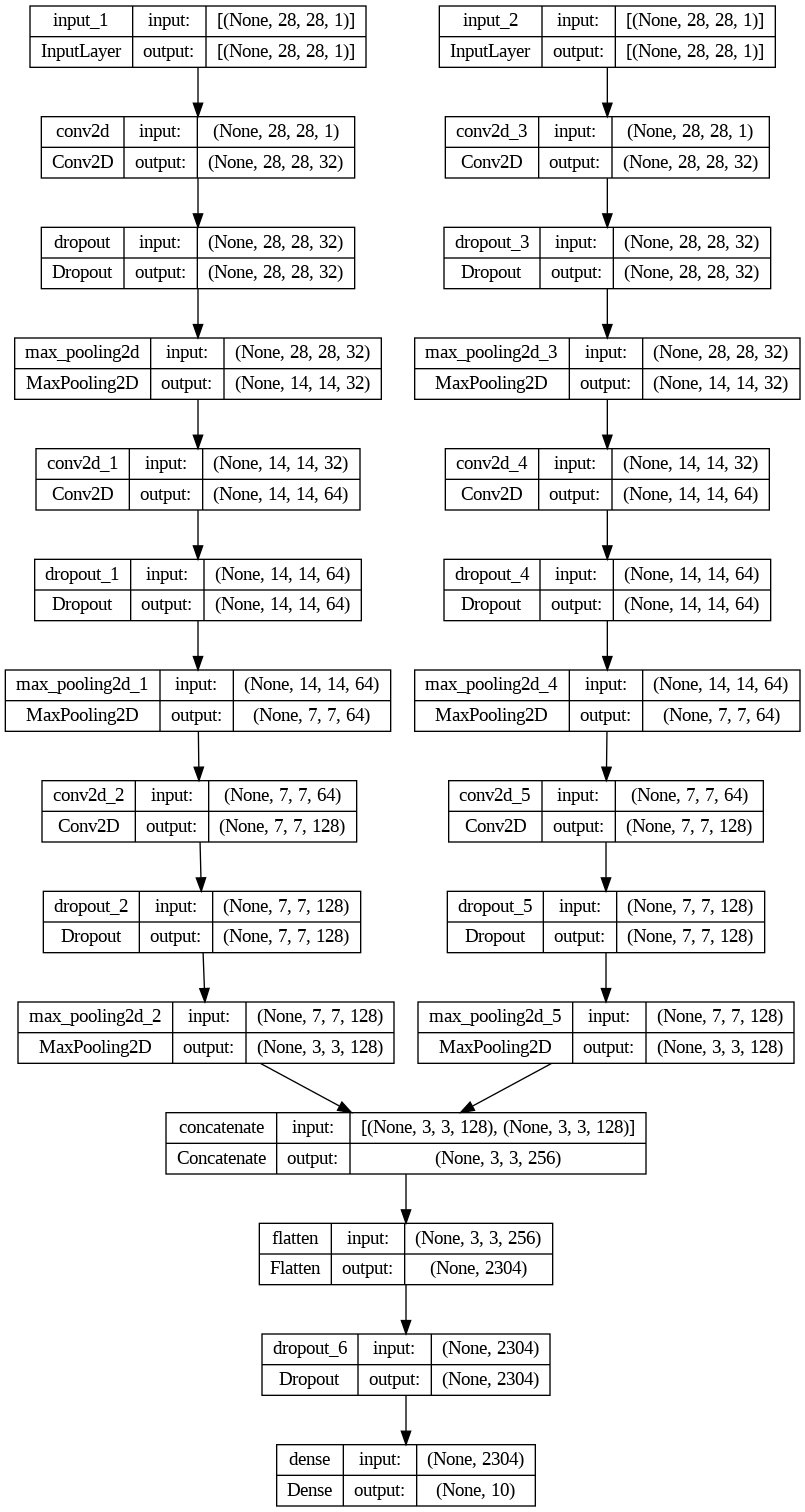


Рисунок 2 – Архитектура Y-DAG нейронной сети

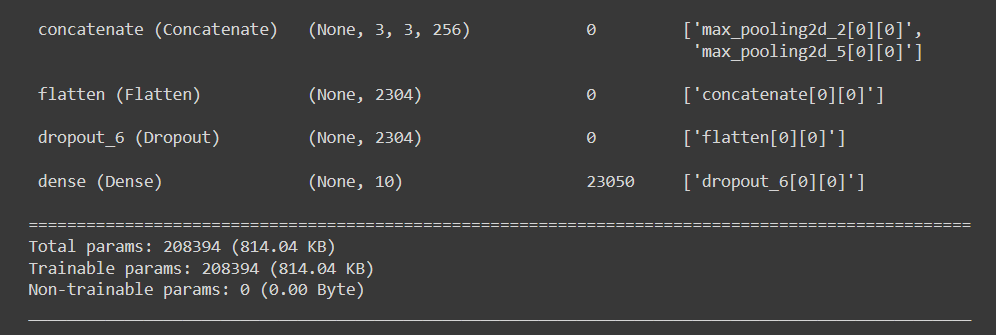


Рисунок 3 – Суммаризация модели

Модель нейронной сети получилась с 208394 параметрами и занимает 814.04 KB (см. рис. 3).

Для запуска обучения нейронной сети используется метод fit:

# Обучение модели

model.fit([x\_train, x\_train],

y\_train,

validation\_data=([x\_test, x\_test], y\_test),

epochs=5,

batch\_size=batch\_size)

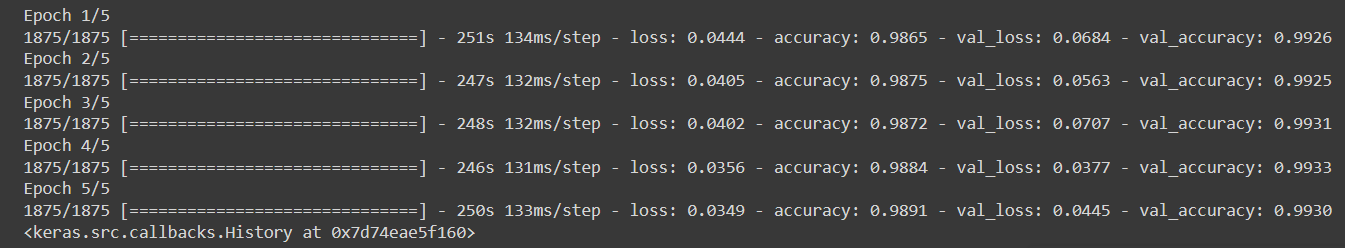


Рисунок 4 – Итерации обучения нейронной сети

При 5-ти эпохах точность модели составляет 99,3%.

Для вывода оценки модели использовался следующий код:

# Оценка работы модели

score = model.evaluate([x\_test, x\_test],

y\_test,

batch\_size=batch\_size,

verbose=2)

print("\nTest accuracy: %.1f%%" % (100.0 \* score[1]))

Для сохранения модели в формате JSON можно использовать следующий код:

# Сохранение архитектуры модели в формате JSON

model\_json = model.to\_json()

# Открытие файла JSON для записи

with open('model\_architecture.json', 'w') as json\_file:

# Запись данных модели в файл JSON

json\_file.write(model\_json)

Решение задания №2

В качестве кастомного класса Callback был реализован класс LossPrintingCallback, который выводит значение функции потерь в процессе обучения. Вывод происходит после обработки какого-то пакета (batch) из тренировочной или тестовой выборки.

Код класса LossPrintingCallback:

# Реализация кастомного класса Callback

class LossPrintingCallback(Callback):

# Вызов функции "завершение обучение на тренировочном пакете ..."

def on\_train\_batch\_end(self, batch, logs=None):

print(

" - Up to batch {}, the average loss is {:7.2f}.".format(batch, logs["loss"])

)

# Вызов функции "завершение обучение на тестовом пакете ..."

def on\_test\_batch\_end(self, batch, logs=None):

print(

" - Up to batch {}, the average loss is {:7.2f}.".format(batch, logs["loss"])

)

Проверка данного Callback’a возможна при добавлении в атрибут callbacks метода fit экземпляра класса LossPrintingCallback:

# Обучение модели

model.fit([x\_train, x\_train],

y\_train,

validation\_data=([x\_test, x\_test], y\_test),

epochs=5,

batch\_size=batch\_size,

callbacks=[LossPrintingCallback()])

На рисунке 5 представлен результат работы данного Callback’a.

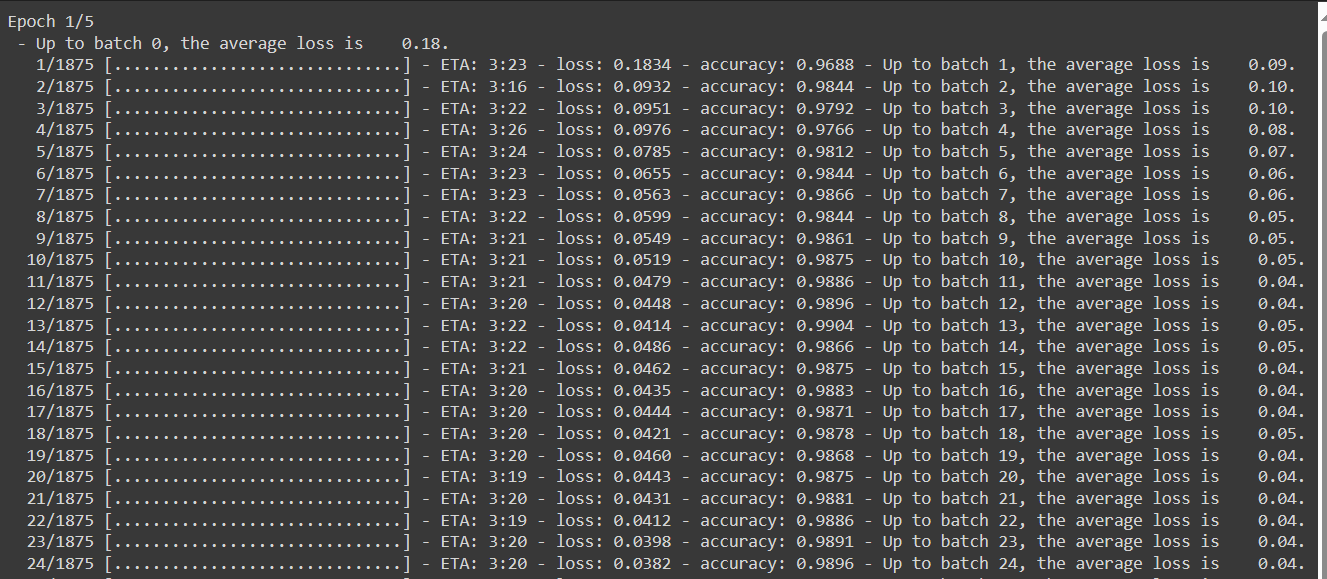


Рисунок 5 – Результат работы Callback’a LossPrintingCallback

Для использования кастомного Callback’a при оценке работы модели, используется следующий код:

# Оценка работы модели

score = model.evaluate([x\_test, x\_test],

y\_test,

batch\_size=batch\_size,

verbose=2,

callbacks=[LossPrintingCallback()])

На рисунке 6 представлен результат работы кастомного Callback’a при оценке работы модели.

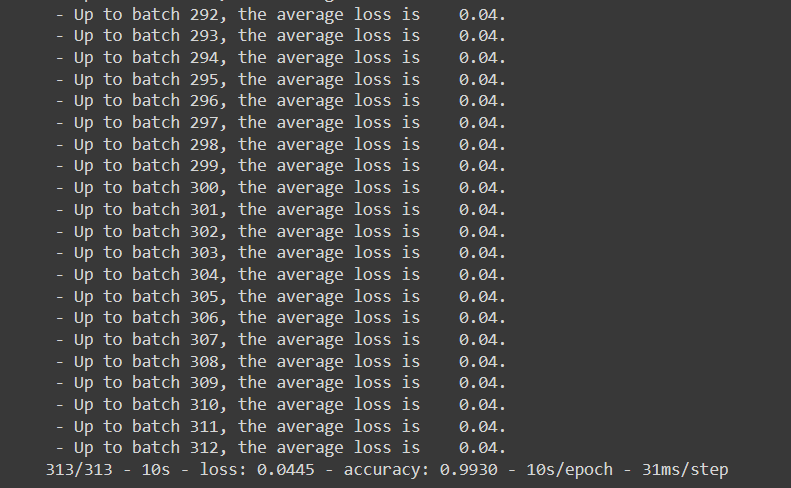


Рисунок 6 – Работа Callback’a при оценке работы модели

Таким образом, кастомный Callback полностью рабочий.

Теперь создадим собственную реализацию функции потерь (keras.losses.Loss) и метрики (tf.keras.metrics.Metric).

Собственная реализация функция потерь через класс выглядит следующим образом:

# Определение кастомной функции потерь

class CustomLoss(tf.keras.losses.Loss):

def \_\_init\_\_(self, alpha=1.0, beta=1.0):

super().\_\_init\_\_()

self.alpha = alpha

self.beta = beta

def call(self, y\_true, y\_pred):

# Средний квадрат ошибки

return tf.reduce\_mean(tf.square(self.alpha \* y\_true - self.beta \* y\_pred))

Класс CustomLoss наследуется от keras.losses.Loss, что позволяет использовать экземпляры объекта данного класса как функцию потерь в TensorFlow.

Функция потерь определяется следующей формулой:

Параметры a и b – это дополнительные параметры функции потерь, которые можно задавать в конструкторе класса CustomLoss.

Чтобы кастомная функция потерь работала её необходимо подключить на этапе компиляции модели в атрибут loss метода compile:

# Компиляция модели

model.compile(loss=CustomLoss(0.5, 2.0),

optimizer='adam',

metrics=['accuracy'])

На рисунке 7 продемонстрировано обучение модели с кастомной функцией потерь.

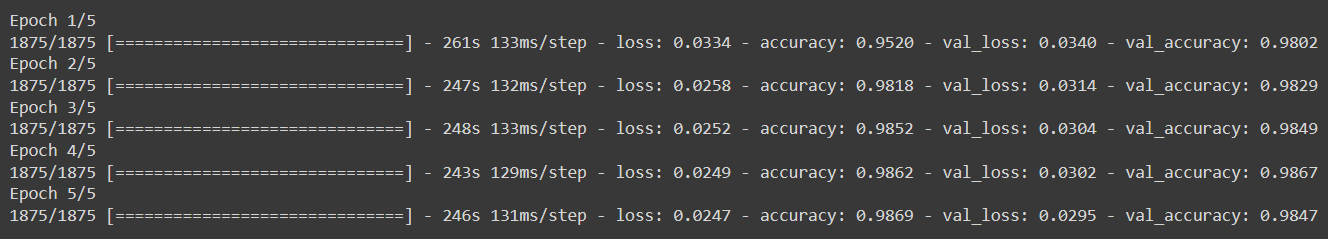


Рисунок 7 – Обучение модели с кастомной функцией потерь

Точность на тестовом наборе данных составляет 98.5%, что на 0.8% меньше, чем с функцией потери categorical\_crossentropy.

Таким образом можно считать, что кастомная функция потерь реализована.

Код кастомного класса метрик выглядит следующим образом:

# Создание кастомного класса метрики

class CategoricalTruePositives(tf.keras.metrics.Metric):

def \_\_init\_\_(self, name="my\_metric"):

super().\_\_init\_\_(name=name)

self.true\_positives = self.add\_weight(name="acc", initializer="zeros")

self.count = tf.Variable(0.0)

# Обработка обновления переменных состояния

def update\_state(self, y\_true, y\_pred, sample\_weight=None):

y\_pred = tf.reshape(tf.argmax(y\_pred, axis=1), shape=(-1, 1))

y\_true = tf.reshape(tf.argmax(y\_true, axis=1), shape=(-1, 1))

values = tf.cast(y\_true, "int32") == tf.cast(y\_pred, "int32")

if sample\_weight is not None:

sample\_weight = tf.cast(sample\_weight, "float32")

values = tf.multiply(values, sample\_weight)

values = tf.cast(values, "float32")

self.true\_positives.assign\_add(tf.reduce\_mean(values))

self.count.assign\_add(1.0)

# Метод для вычисления метрик на основе переменных состояния

def result(self):

return self.true\_positives / self.count

# Сброс переменных состояния

def reset\_states(self):

self.true\_positives.assign(0.0)

self.count.assign(0.0)

В самом начале, в конструкторе, формируются две переменные состояния: true\_positives – сумма долей верной классификации и count – общее число долей.

Затем, в методе update\_state() для каждого мини-батча вычисляется вектор верной классификации (values) и определяется доля правильной классификации с увеличением счётчика count на единицу.

В методе result() происходят окончательные вычисления для метрики (вычисление средней доли верной классификации), а в методе reset\_states() происходит сброс переменных true\_positives и count в ноль.

Добавление кастомного класса с метрикой происходит при компиляции модели:

# Компиляция модели

model.compile(loss=CustomLoss(0.5, 2.0),

optimizer='adam',

metrics=['accuracy', CategoricalTruePositives()])

На рисунке 8 продемонстрирован результат добавления кастомной метрики.

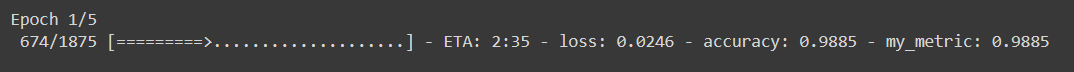


Рисунок 8 – Результат добавления кастомной метрики my\_metric

Таким образом, кастомный класс метрики работает и задание №2 полностью решено.

Решение задания №3

Для решения этой задачи будет разработан класс TfConv2D, который будет выполнять туже работу, которую выполняет стандартный слой Conv2D:

# Кастомный свёрточный слой

class TfConv2D(tf.keras.layers.Layer):

# Конструктор класса

def \_\_init\_\_(self, kernal=(3,3), chanels=1, strides=(2,2), padding='SAME', acivate='relu'):

super().\_\_init\_\_()

self.kernal = kernal

self.chanels = chanels

self.strides = strides

self.padding = padding

self.acivate = acivate

self.fl\_init = False

# Делаем экземпляры класса вызываемыми

def \_\_call\_\_(self,x):

if not self.fl\_init:

self.w = tf.random.truncated\_normal((\*self.kernal, x.shape[-1], self.chanels), stddev=0.1, dtype=tf.float32)

self.b = tf.zeros([self.chanels], dtype=tf.float32)

# Преобразуем тензор в переменную, чтобы использовать в градиентном спуске

self.w = tf.Variable(self.w)

self.b = tf.Variable(self.b)

self.fl\_init = True

y = tf.nn.conv2d(x, self.w, strides=(1, \*self.strides, 1), padding=self.padding) + self.b

# Определяем функцию активации и делегируем вычисления ей

if self.acivate == "relu":

return tf.nn.relu(y)

elif self.acivate == "softmax":

return tf.nn.softmax(y)

return y

В конструкторе определены основные параметры слоя Conv2D.

При вызове экземпляра объекта класса TfConv2D как функции будет происходить проверка на инициализацию данного слоя. Если инициализация уже была, то результат выполнения этой функции никак не отобразиться на работе программы. Если же её не было, то происходит работа слоя Conv2D.

Сначала инициализируются веса (w) и нейрон смещения (b), которые затем конвертируются в переменные TensorFlow для работы градиентного спуска.

После этого происходит вызов функции conv2d из стандартной библиотеки TensorFlow, которая вычисляет двумерную свёртку, заданную входными и четырёхмерными фильтрующими (filters) тензорами.

Далее, в зависимости от функции активации, вычисленная свёртка делегируется функции активации, тем самым завершая работу свёрточного слоя.

Теперь добавим данный свёрточный слой в левую и правую ветвь Y-DAG дерева там, где число фильтров 64:

# …

# Левая ветвь

# Создание свёрточного слоя с числом фильтров 64

x = Conv2D(filters=(2 \* filters), kernel\_size=kernel\_size, padding='same', activation='relu')(x)

x = Dropout(dropout)(x)

x = TfConv2D(kernel=(kernel\_size, kernel\_size), channels\_count=(2 \* filters), acivate='relu')(x)

x = MaxPooling2D()(x)

# …

# …

# Правая ветвь

# Создание свёрточного слоя с числом фильтров 64

y = Conv2D(filters = (2 \* filters),kernel\_size = kernel\_size,padding = 'same',

activation = 'relu',dilation\_rate =2)(y)

y = Dropout(dropout)(y)

y = TfConv2D(kernel=(kernel\_size, kernel\_size), channels\_count=(2 \* filters), acivate='relu')(y)

y = MaxPooling2D()(y)

#...

Архитектура получившейся Y-DAG сети будет отличаться от архитектуры, полученной в задании №1 (см. рис. 9).

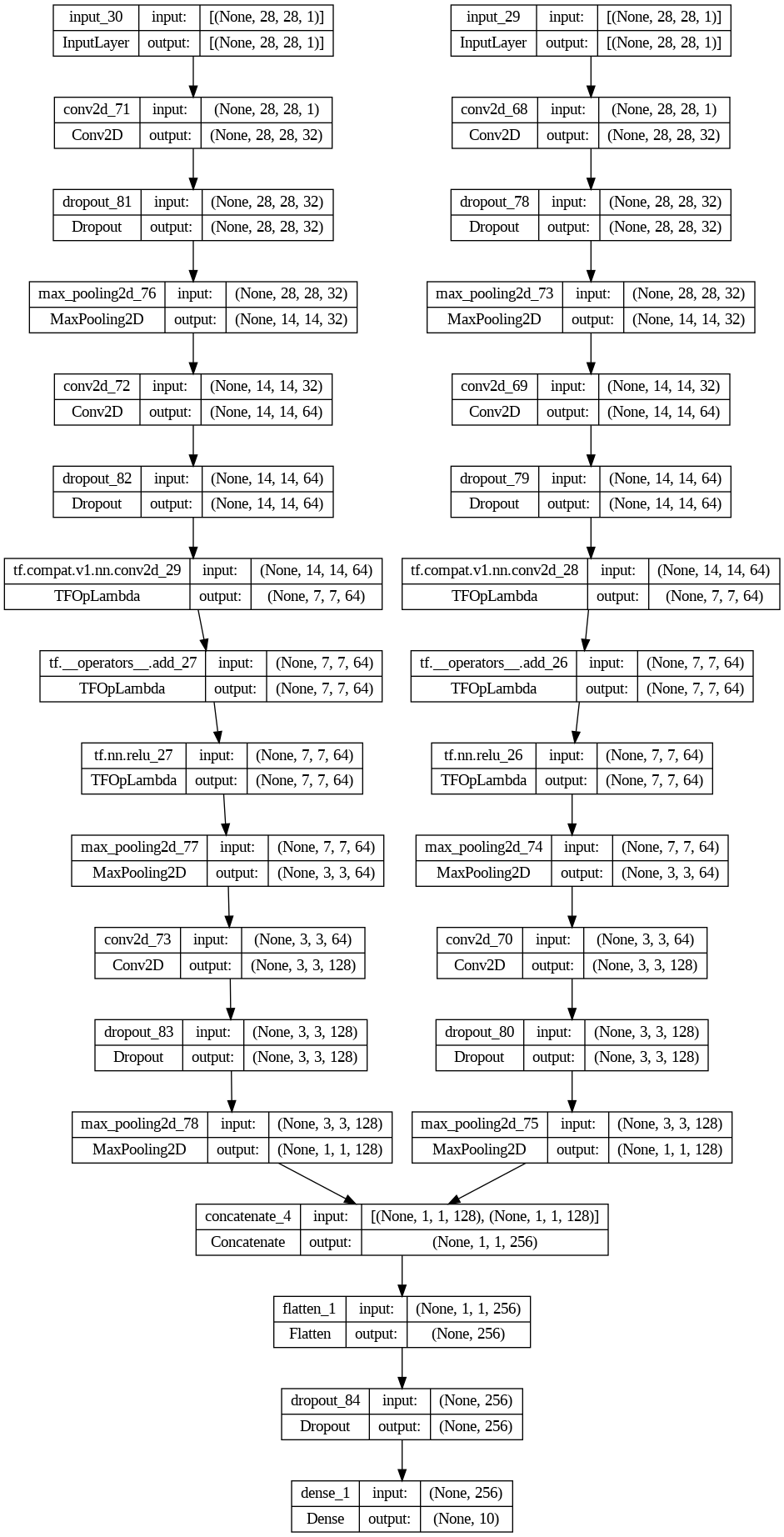


Рисунок 9 – Архитектура с кастомным слоев TfConv2D

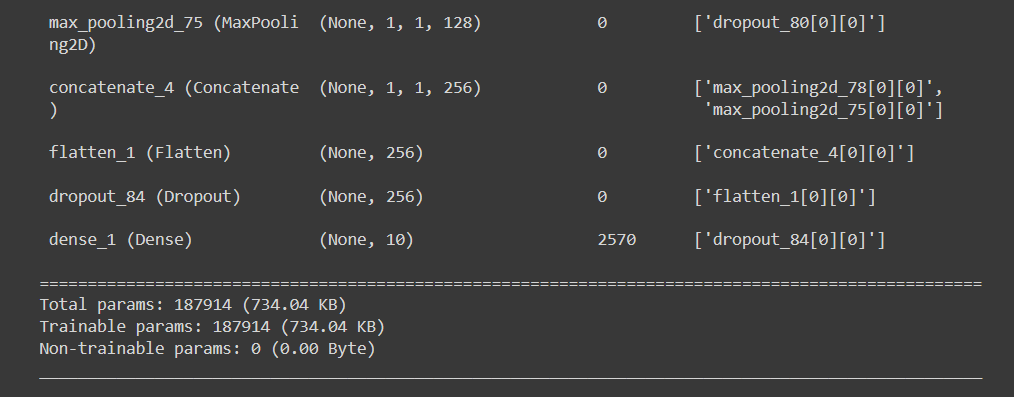


Рисунок 10 – Суммаризация новой модели

Также, при суммаризации модели, общее число параметров модели уменьшилось – теперь их 187914.

На рисунке 11 представлен результат обучения нейронной сети с кастомными слоями.

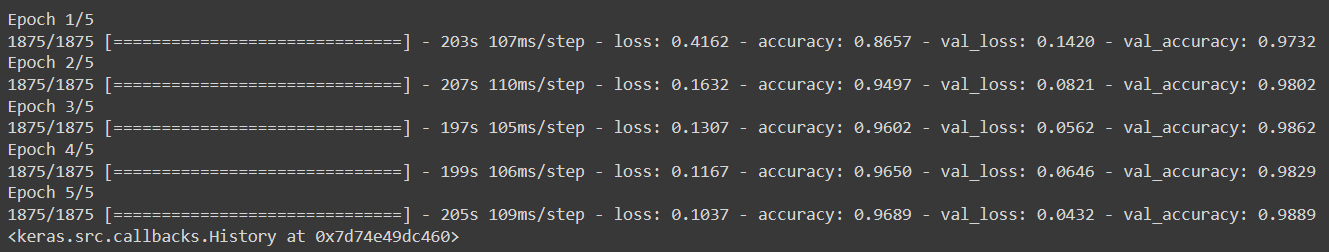


Рисунок 11 – Результат обучения нейронной сети с кастомными слоями

Точность на тестовом наборе данных составляет 98.9%.

Для сохранения данной модели в формате JSON используется следующий код:

# Сохранение архитектуры модели в формате JSON

model\_json = model.to\_json()

# Открытие файла JSON для записи

with open('model\_architecture.json', 'w') as json\_file:

# Запись данных модели в файл JSON

json\_file.write(model\_json)

А для сохранения модели в формате pb, используется уже другой код:

model.save("save\_pb")

При сохранении модели в формате pb мы получаем отдельную директорию, в которой хранится вся подробная информация о модели нейронной сети (см. рис. 12).

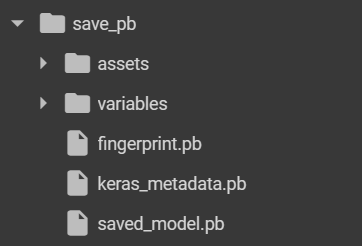


Рисунок 12 – Сохранённая модель в формате pb

В файле saved\_model.pb хранится архитектура модели и конфигурация обучения.

Веса хранятся отдельно в каталоге variables.

Заключение

В рамках лабораторной работы были выполнены все поставленные задачи.

При решении задания №1 была создана многослойная нейронная сеть с использованием функционального API TensorFlow 2, которая имеет структуру ориентированного ациклического графа и написан код для сохранения её в формате JSON.

При решении задания №2 была создана собственная реализация обратного вызова через объект keras.callbacks.Callback и протестирована с использованием keras.Model.fit() из сети задания №1. Также была создана собственная функция потерь и метрики, в которых использовались классы keras.losses.Loss и keras.metrics.Metric соответственно.

При решении задания №3 был создан кастомный слой через класс keras.layers и добавлен в ветви Y-DAG нейронной сети с её дальнейшим обучением и сохранением в формате JSON и pb.

Таким образом все задачи лабораторной работы были решены. Был приобретён опыт построения нейронных сетей с ориентированным ациклическим графом вида Y-DAG, изучены способы создания своих кастомных функций потерь, обратного вызова и метрик.

Приложения

1. Ссылка на исходный код: <https://github.com/DanSoW/INRTU/tree/main/software-tools-for-artificial-intelligence-tasks/lab2> .