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 №1  по дисциплине  «Программные средства для задач искусственного интеллекта» |
| «Программные библиотеки для решения вычислительных задач ИИ» |
| 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](#_Toc165476928)

[Решение задания №1 4](#_Toc165476929)

[Решение задания №2 7](#_Toc165476930)

[Решение задания №3 15](#_Toc165476931)

[Решение задания №4 17](#_Toc165476932)

[Заключение 21](#_Toc165476933)

[Приложения 22](#_Toc165476934)

Постановка задач

Необходимо решить следующие задачи:

1. **Задание №1**:
2. Ручное создание вычислительного графа функции от нескольких переменных и нахождение производной сложной функции.
3. Проверка вычислений с помощью кода на TensorFlow 2.
4. **Задание №2**:
5. Создать свою модель типа tf.Module для функции из первого задания.
6. Сгенерировать данные для обучения. Провести идентификацию параметров функции.
7. Вставить в отчёт программный код на TensorFlow 2
8. Провести тестирование программного кода.
9. **Задание №3**:
10. Создать функцию для вычисления значений (для функции из первого задания) как функцию python и создать на её основе вычислительный граф с использованием tf.functiion в форме декоратора.
11. Провести тестирование программного кода.
12. **Задание №4**:
13. Написать код вычислительного графа для функции

Решение задания №1

Для начала необходимо определить функцию, вычислительный граф для которой будет строится.

Функция выглядит следующим образом:

Производная функции будет найдена для точки x = 2, y = -2.

Теперь можно составить вычислительный граф по отдельным элементам определённой функции. На рисунке 1 представлен вычислительный граф (без обратного прохода) определённой функции.

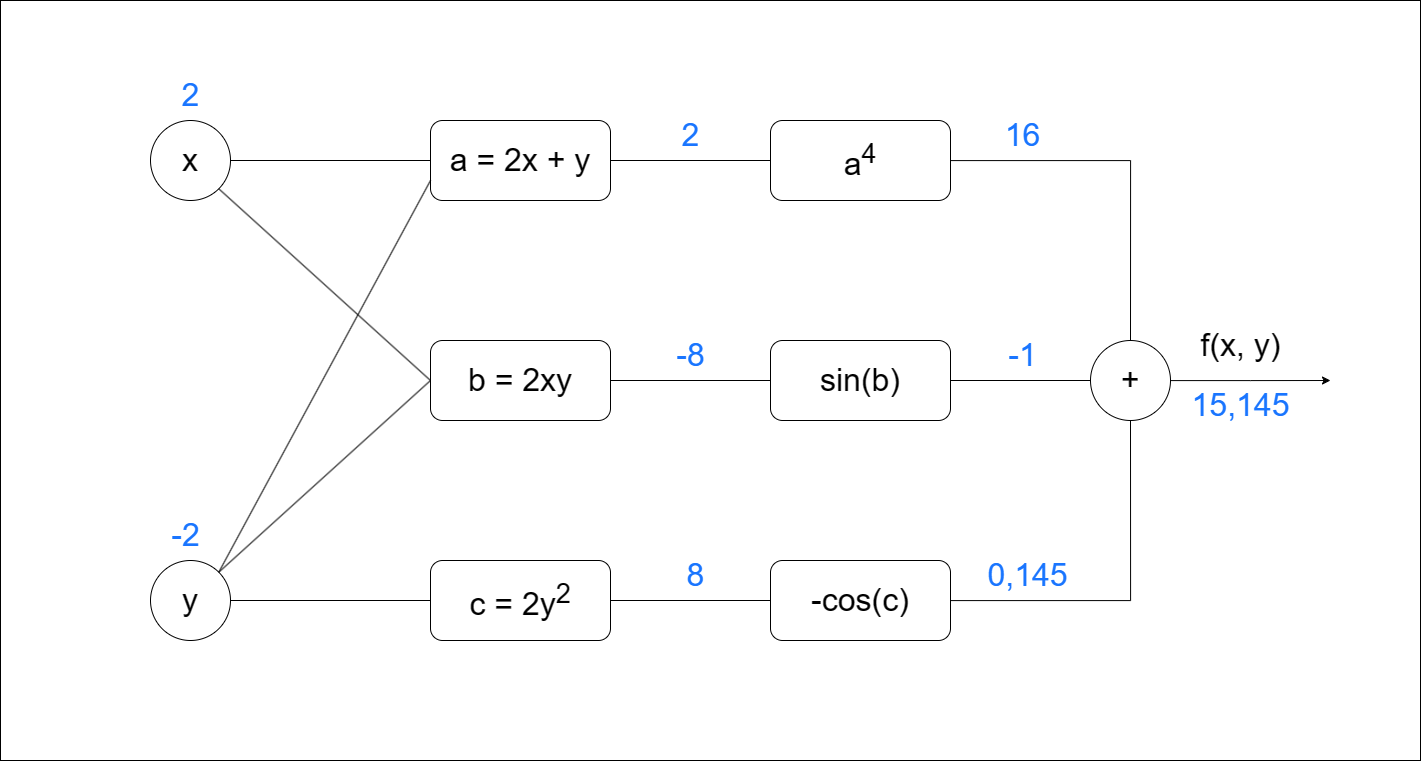


Рисунок 1 – Вычислительный граф для функции f(x, y)

Из вычислительного графа функции на рисунке 1 находим, что значение функции f(x, y) в точке (2, -2) равно 15.145.

Отметим, что результаты вычислений будут округлены до 3-го знака после запятой при этом промежуточные результаты могут не округляться (для получения более точных результатов). Значение sin (8) округлено до 1, т.к. значение 0,989 крайне близко к 1 и этим значением можно принебречь, т.к. на конечные результаты оно не влияет.

Для обратного прохода по вычислительному графу воспользуемся следующим цепным правилом:

Благодаря данному цепному правилу мы можем вычислить значения производной функции в точках x = 2 и y=-2.

На рисунке 2 представлен обратный проход по вычислительному графу с вычислением производных (см. рис. 2).

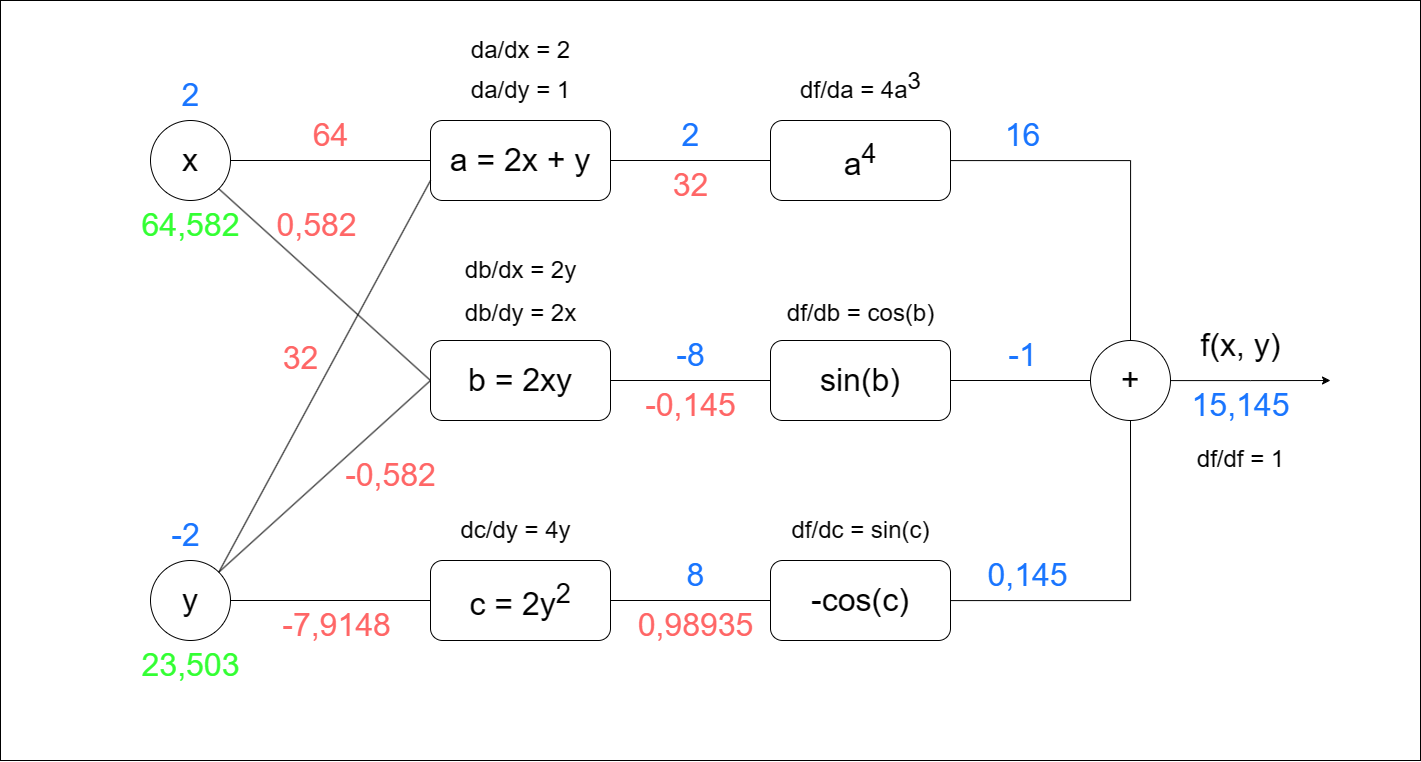


Рисунок 2 – Обратный проход по вычислительному графу

Производная функции f(x, y) в точке x = 2 равна 64,582, а в точке y = -2 равна 23,503. Данные значения совпадают с теми, что были вычислены при определении цепного правила.

Программный код для проверки работы вычислительного графа из рисунка 2 выглядит следующим образом:

import os

os.environ['TF\_CPP\_MIN\_LOG\_LEVEL'] = '2'

import tensorflow as tf

x = tf.Variable([[2.0]])

y = tf.Variable([[-2.0]])

with tf.GradientTape() as tape:

f = (2 \* x + y) \*\* 4 + tf.math.sin(2 \* x \* y) - tf.math.cos(2 \* y \*\* 2)

df = tape.gradient(f, [x, y])

print(df[0], df[1], sep="\n")

При тестировании были созданы переменные x и y со значением 2.0 и -2.0 соответственно. В таблице 1 предсталвен результат тестирования данного программного кода.

Таблица 1 – Таблица тестов

|  |  |  |  |
| --- | --- | --- | --- |
| Номер теста | Назначение  теста | Входные данные | Выходные данные |
| 1 | Проверка правильности вычислений вычислительного графа с использованием TensorFlow 2 | x = 2.0  y = -2.0 | 64.582  23.503 |

Результаты тестирования в таблице 1 подтверждают корректность вычислений в вычислительном графе, представленном на рисунке 2, что подтверждает успешное решение задания №1.

Решение задания №2

Исходная функция выглядит следующим образом:

Для детекции параметров функции необходимо их сначала выделить. В данной функции в качестве параметров, которые будет предсказывать нейронная сеть, будут выбраны коэффициенты в каждом выражении функции.

Нейронная сеть должна будет предсказать значения коэффициентов a, b и c таким образом, чтобы значения функции с этими параметрами совпадали с исходной функцией.

Создание модели типа tf.Module для функции из первого задания выглядит следующим образом:

# Определение класса DenseNN

class DenseNN(tf.Module):

def \_\_init\_\_(self, outputs):

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

# Добавление выходных нейронов

self.outputs = outputs

# Флаг инициализации (первый вызов объекта через \_\_call\_\_)

self.fl\_init = False

# Добавление объекту класса возможности "вызываться" как функции

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

# Если fl\_init равен False, то

if not self.fl\_init:

# Предварительно инициализируем весов и смещения

self.w = tf.random.truncated\_normal((3, self.outputs), stddev=0.1, name="w")

self.b = tf.zeros([self.outputs], dtype=tf.float32, name="b")

# Конвертация значений весов и смещения в переменные

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

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

self.fl\_init = True

# Получение значений точки x и y, в контексте функции

a, b = x[0, 0], x[0, 1]

# Определение значения функции

y = (self.w[0] \* a + b) \*\* 4 + tf.math.sin(self.w[1] \* a \* b) - tf.math.cos(self.w[2] \* b \*\* 2) + self.b

return y

Модель состоит из определения двух функций – конструктора и определения \_\_call\_\_, которая служит для того, чтобы можно было вызывать экземпляр данного класса как функцию (через скобки).

В конструкторе предварительно определены два атрибута класса – outputs и fl\_init. Первый атрибут определяет число выходных нейронов модели, а второй показывает состояние инициализации весов и нейрона смещения модели. Если он равен False, то инициализации ещё не было, а если True, то инициализация уже была.

Инициализация весов и нейрона смещения происходит только один раз – при первом вызове экземпляра класса DenseNN как функции.

Далее, определим функцию из первой задачи в виде Python-кода:

# Определение функции из первой задачи

func = lambda x, y: (2 \* x + y) \*\* 4 + tf.math.sin(2 \* x \* y) - tf.math.cos(2 \* y \*\* 2)

В функции используется библиотека математики TensorFlow (tf.math). Это сделано для того, чтобы все будущие вычисления этих функций не выходили за рамки существующего TensorFlow API.

Далее, определим датасет для нейронной сети и основные параметры конфигурации модели:

# Размер выборки

SIZE = 1000

# Обучающий набор данных (образцы)

x\_train = tf.random.uniform(minval=0, maxval=1.5, shape=(SIZE, 2))

# Правильные "ответы" для нейроной сети (метки)

y\_train = [func(a, b) for a, b in x\_train]

# Создание экземпляра класса DenseNN с одним выходным нейроном

model = DenseNN(1)

# Определение функции потерь - средний квадрат ошибки

loss = lambda x, y: tf.reduce\_mean(tf.square(x - y))

# Определение оптимизатора - оптимизатор Адам со скоростью обучения 0.001

opt = tf.optimizers.Adam(learning\_rate=0.001)

# История функции потерь

loss\_history = []

# Минимальная ошибка

min\_loss = -1

Размер выборки для обучающего набора данных составляет 1000 экземпляров.

В переменной x\_train будут записан результат генерации точек (x, y) по равномерному закону распределения. Все значения x и y будут находиться в диапазоне от 0 до 1.5, в силу специфики исходной функции.

В переменной y\_train будут находится метки для набора данных из x\_train. Если точнее – значения функции func по значениям из набора данных \_x\_train. Это необходимо для того, чтобы можно было обучить нейронную сеть и затем протестировать.

В качестве функции потерь выступает функция среднеквадратической ошибки:

Процесс обучения представлен следующим программным кодом:

# Обучение модели нейронной сети

for i in range(30):

# Проход по кортежам

for x, y in zip(x\_train, y\_train):

# Расширение формы входного массива

x = tf.expand\_dims(x, axis=0)

# Создание константы по существующему значению y

y = tf.constant(y, shape=(1, 1))

# Процесс вычисления градиента

with tf.GradientTape() as tape:

f\_loss = loss(y, model(x))

# Получение значения градиентов функции

grads = tape.gradient(f\_loss, model.trainable\_variables)

# Определение правил обновления градиента функции

opt.apply\_gradients(zip(grads, model.trainable\_variables))

# Вывод промежуточных результатов обучения

print(str(i+1) + ': ' + str(f\_loss.numpy()))

# Если значение функции потерь меньше минимального значения, то

if f\_loss.numpy() < min\_loss or min\_loss == -1:

# Вывести на экран промежуточную информацию

print('loss reduced from', min\_loss, 'to', f\_loss.numpy())

# Задать новое минимальное значение функции потерь

min\_loss = f\_loss.numpy()

# Записать в файл промежуточные результаты обучения нейронной сети

f = open('saved\_model', 'wb')

pickle.dump(model, f)

f.close()

# Добавление в историю результатов функции потерь

loss\_history.append(f\_loss.numpy())

# Открытие файла с результатами обучения нейронной сети

f = open('saved\_model', 'rb')

# Загрузка из файла данных в модель

model = pickle.load(f)

# Закрытие файла

f.close()

# Вывод количества тренировочных параметров

print(model.trainable\_variables)

# Вывод графика функции потерь

plt.plot(loss\_history)

Обучение происходит следующим образом. Запускается цикл на 30 итераций. В этом цикле есть другой цикл, который проходит по всему тренировочному набору данных (образцам и меткам).

В цикле обучения происходит вычисление градиента функции и задание правил обновления градиента функции (apply\_gradients).

Затем вне этого цикла происходит вывод промежуточной информации, в которой можно узнать текущее значение функции потерь.

После этого происходит запись в файл saved\_model текущей конфигурации модели нейронной сети и запись в переменную min\_loss нового значения, которое характеризует текущее значение функции потерь. Также добавляется значение функции потерь в список с историей (loss\_history).

После обучения модели она загружается из файла (в файл заносятся только лучшие результаты), происходит вывод количества тренировочных параметров и вывод изменения значений функции потерь.

На рисунке 3 представлены первые 10 итераций обучения нейронной сети.

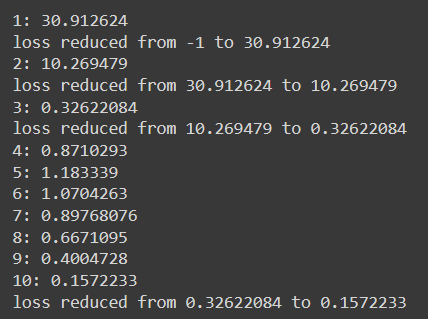


Рисунок 3 – Первые 10 итераций обучения нейронной сети

Как видно из рисунка 3 значение функции потерь постепенно уменьшается, что также видно и на рисунке 4.

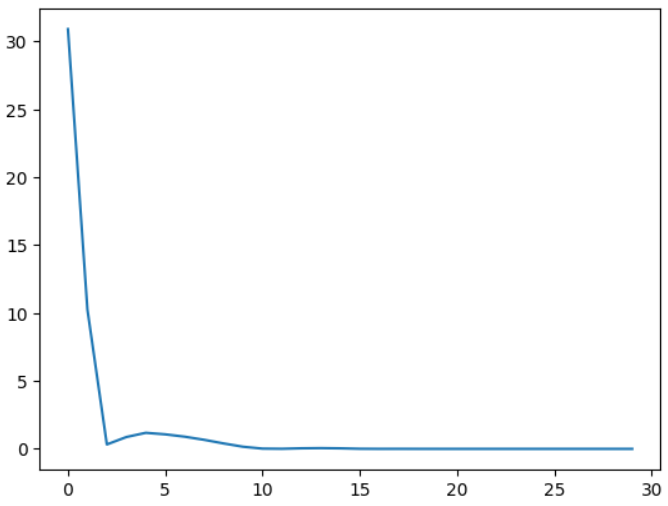


Рисунок 4 – График уменьшения значения функции потерь с течением времени

За 30 итераций удалось достичь значения функции ошибки равное 4.8466973e-09.

Нейронная сеть предсказала следующие параметры функции:

a = 1.9999267, b = 1.9931198, c = 2.0007467.

Можно отметить, что при округлении этих значений до целых эти коффициенты будут равны 2, как и в исходной функции (a = b = c = 2, в исходной функции).

Подставив предсказанные коэффициенты в функцию, получаем:

Сравним графики полученной и исходной функции.

На рисунке 5 представлен график исходной функции (см. рис. 5).

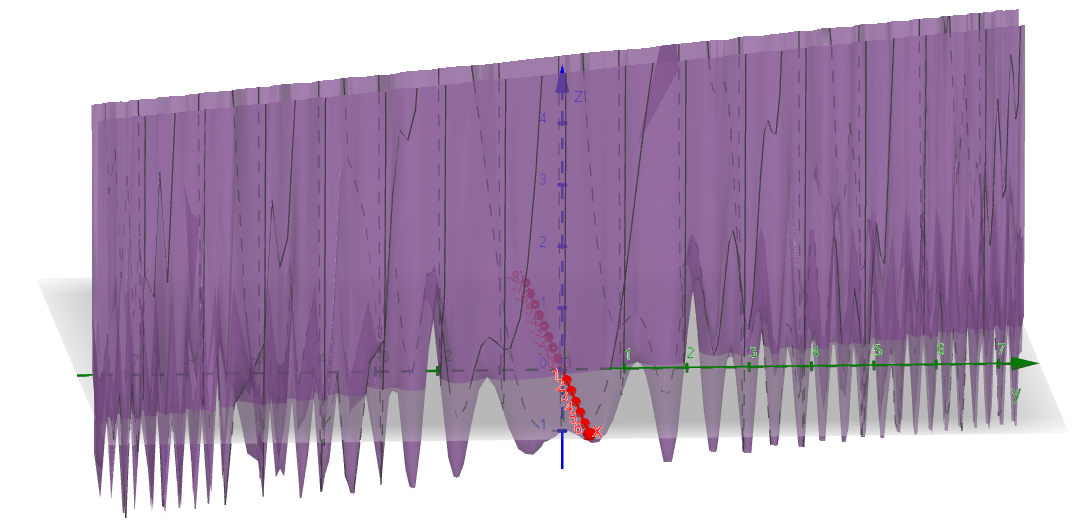


Рисунок 5 – График исходной функции

Теперь на данный график будет наложен график функции с новыми коэффициентами (см. рис. 6).

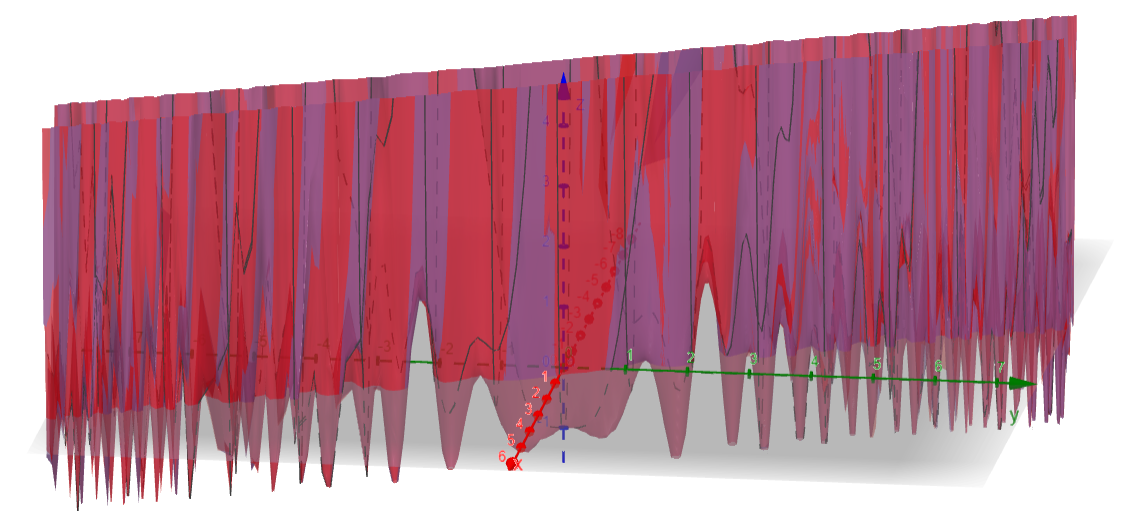


Рисунок 6 – Результат накладывания графика новой функции на исходную

Как видно из рисунка 6 новая функция практически полностью ложиться на исходную, а поэтому параметры были определены нейронной сетью довольно точно.

Протестируем модель нейронной сети для предсказания различных значений исходной функции (см. таблица 2).

Таблица 2 – Тестирование модели нейронной сети

|  |  |  |  |
| --- | --- | --- | --- |
| Номер теста | Назначение  теста | Входные данные | Выходные данные |
| 1 | Проверка предсказания нейронной сети | x = 3.0, y = 2.0, f(x, y) = 4095.609 | f(x, y) = 4095.1287  отклонение = 0.4802246 |
| 2 | Проверка предсказания нейронной сети | x = 2.0, y = -2.0, f(x, y) = 15.156142 | f(x, y) = 15.152173  отклонение = 0.00396919 |
| 3 | Проверка предсказания нейронной сети | x = 1.4, y = 2.7, f(x, y) = 916.4481 | f(x, y) = 916.3783  отклонение = 0.06982422 |
| 4 | Проверка предсказания нейронной сети | x = 0.9, y = -3, f(x, y) = 2.1860478 | f(x, y) = 2.1945248  отклонение = 0.00847697 |
| 5 | Проверка предсказания нейронной сети | x = 1.4, y = 4, f(x, y) = 2136.3242 | f(x, y) = 2136.1958  отклонение = 0.12841797 |

Средняя погрешность между значениями исходной функции и предсказанной нейронной сетью составляет 0.13818259239196778.

На рисунке 7 представлен результате тестирования нейронной сети.

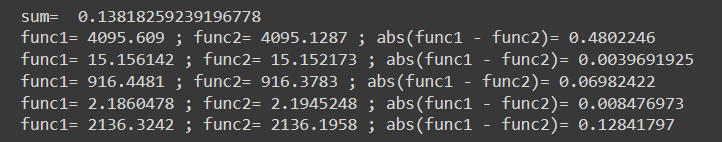


Рисунок 7 – Результат тестирования нейронной сети

Код для тестирования использовался следующий:

x = [3.0, 2.0, 1.4, 0.9, 1.4]

y = [2.0, -2.0, 2.7, -3.0, 4.0]

result = []

sum = 0

for i in range(len(x)):

a = func(x[i], y[i]).numpy()

b = model1(tf.constant([[x[i], y[i]]])).numpy()[0]

c = abs(a - b)

sum += c

result.append([a, b, c])

print("sum= ", (sum / len(x)))

for i in result:

print("func1=", i[0], "; func2=", i[1], "; abs(func1 - func2)=", i[2])

Нейронная сеть практически правильно предсказала значения коэффициентов функции a, b и c, которые, если округлить до целых, будут равны коэффициентам исходной функции. Таким образом задача решена.

Решение задания №3

Для начала определим исходную функцию в Python-коде:

# Исходная функция

def function\_tf(x, y):

f = (2 \* x + y) \*\* 4 + tf.math.sin(2 \* x \* y) - tf.math.cos(2 \* y \*\* 2)

return f

Далее, определим функцию с декоратором tf.function:

# Новая функция с декоратором tf.function

@tf.function

def function\_tf\_optimize(x, y):

f = (2 \* x + y) \*\* 4 + tf.math.sin(2 \* x \* y) - tf.math.cos(2 \* y \*\* 2)

return f

tf.function – это декоратор, предоставляемый TensorFlow, который преобразует функции в Python в операции на основе графа.

Это преобразование позволяет TensorFlow компилировать и оптимизировать вычисления обёрнутой в данный декоратор функции, что приводит к повышению производительности и эффективности.

Таким образом, с помощью данного декоратора можно улучшить скорость выполнения функции. Протестируем исходную и обёрнутую функции на вычислениях с 10000 элементами.

Для тестирования будет создана следующая функция:

# Функция для тестирования

def test\_function(fn):

# Локальная функция

def wrapper(\*args, \*\*kwargs):

# Запоминаем время начала вычислений

start = time.time()

# Вызов функции

fn(\*args, \*\*kwargs)

# Получаем продолжительность вычислений

dt = time.time() - start

print(f"Время обработки: {dt} сек")

# Возвращение локальной функции

return wrapper

Сам код тестирования выглядит следующим образом:

# Размер тестовых данных

SIZE = 10000

# Генерация тестовых данных, заполненных единицами

x = tf.ones((SIZE, SIZE), dtype=tf.float32)

y = tf.ones\_like(x, dtype=tf.float32)

print("Вычисления с исходной функцией")

# Вызов тестовой функции с передачей ей исходной функции

test\_function(function\_tf)(x, y)

print()

print("Вычисления с функцией, обёрнутой в tf.function")

# Вызов тестовой функции с передачей ей функции,

# которая преобразована в операции на основе графа

test\_function(function\_tf\_optimize)(x, y)

В результате запуска получаем следующий результат:

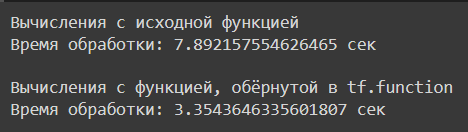


Рисунок 8 – Результат тестирования исходной и обёрнутой функции

Из рисунка 8 можно отметить, что вычисления с исходной функции продолжались почти 8 секунд, а с обёрнутой в декоратор tf.function время вычислений сократилось до 3-х секунд.

Таким образом использование декоратора tf.function позволяет ускорить вычисления, производимые в обёрнутой функции.

Решение задания №4

Для решения данной задачи необходимо составить узлы для каждого элемента операции в вычислительном графе из рисунка 9. Причём эти узлы должны поддерживать как прямой, так и обратный проходы.

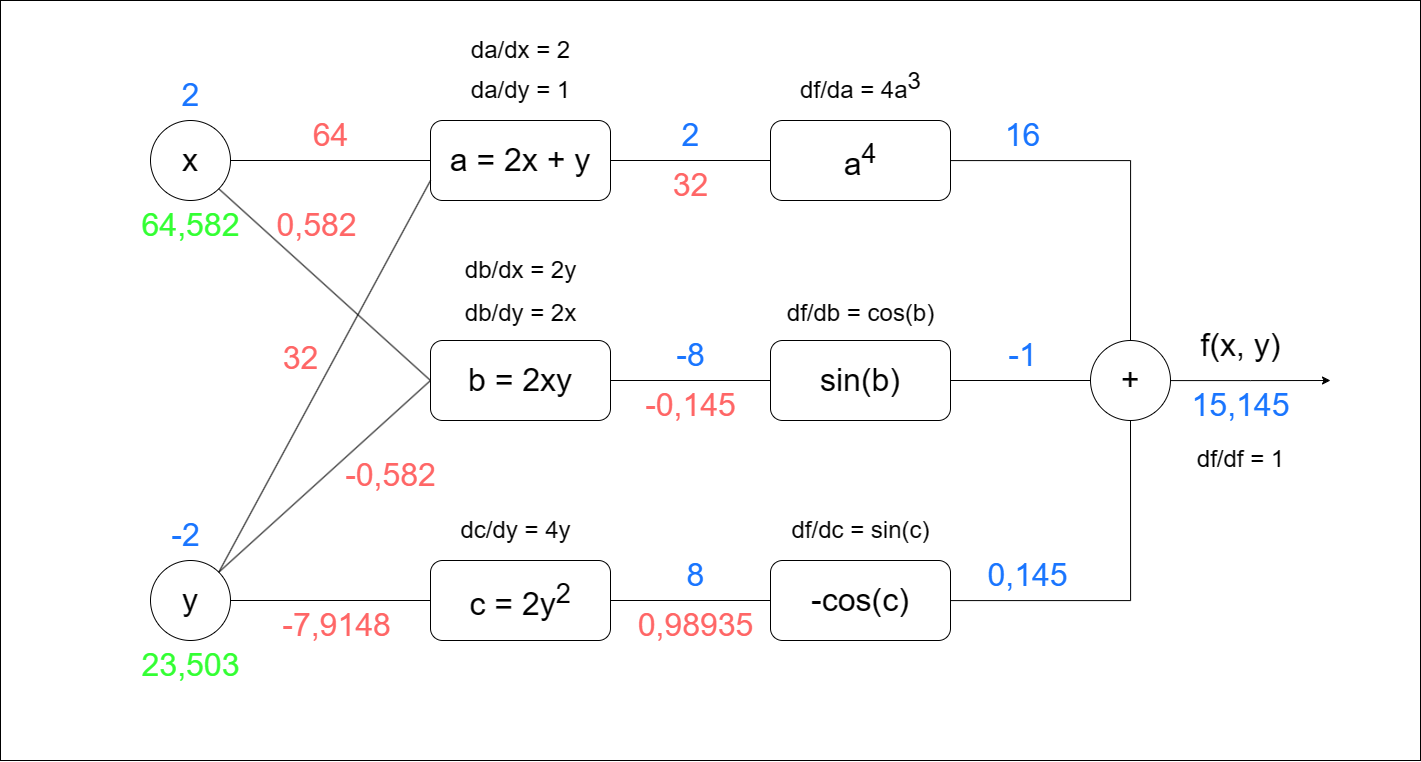


Рисунок 9 – Обратный проход по вычислительному графу

Узел для операции умножения определён следующим образом:

# Определение узла для умножения

class MultiplyNode(object):

def \_\_init\_\_(self, theParam=1):

self.param = theParam

# Прямой проход

def forward(self, x, y):

z = self.param \* x \* y

self.x = x

self.y = y

return z

# Обратный проход

def backward(self, dz = 1.0):

# Производная по y

dy = self.param \* self.x \* dz #dx/dt\*dL/dz

# Производная по x

dx = self.param \* self.y \* dz

return [dx, dy]

Узел для операции суммы определён следующим образом:

# Определение узла для суммы

class SumNode(object):

def \_\_init\_\_(self, theParam = 1):

self.param = theParam

# Прямой проход

def forward(self, x, y):

z = self.param \* x + y

self.x = x

self.y = y

return z

# Обратный проход

def backward(self, dz = 1.0):

dy = dz #dx/dt\*dL/dz

dx = self.param \* dz

return [dx, dy]

Узел для возведения в степень определён следующим образом:

# Определение узла для возведения в степень

class PowNode(object):

def \_\_init\_\_(self, theParam = 1.0, coef = 1.0):

self.param = theParam

self.coef = coef

# Прямой проход

def forward(self, x):

z = self.coef \* x \*\* self.param

self.x = x

return z

# Обратный проход

def backward(self, dz = 1.0):

dx = (self.param \* self.coef) \* (self.x \*\* (self.param - 1)) \* dz

return dx

Узлы для функций синуса и косинуса определены следующим образом:

# Определение узла для синуса

class SinNode(object):

def forward(self, x):

z = tf.math.sin(x)

self.x = x

return z

def backward(self, x):

dx = tf.math.cos(x)

return dx

# Определение узла для косинуса

class CosNode(object):

def forward(self, x):

z = -1.0 \* tf.math.cos(x)

self.x = x

return z

def backward(self, x):

dx = tf.math.sin(x)

return dx

После того, как были определены узлы для выполнения операций, необходимо провести прямой и обратный проход по графу так, как это сделано в первой задаче (см. рис. 9).

Для начала, мы должны инициализировать каждый узел:

nodeA = SumNode(2)

nodeB = PowNode(4)

nodeC = MultiplyNode(2)

nodeD = SinNode()

nodeE = PowNode(2, 2)

nodeF = CosNode()

Затем, сделать прямой проход по каждому из узлов:

resA = nodeA.forward(2.0, -2.0)

resB = nodeB.forward(resA)

resC = nodeC.forward(2.0, -2.0)

resD = nodeD.forward(resC)

resE = nodeE.forward(-2.0)

resF = nodeF.forward(resE);

И после этого нужно сделать обратный проход по графу:

gradAx = nodeA.backward()[0] \* nodeB.backward()

gradAy = nodeA.backward()[1] \* nodeB.backward()

gradBx = nodeC.backward()[0] \* nodeD.backward(resC)

gradBy = nodeC.backward()[1] \* nodeD.backward(resC)

gradCy = nodeE.backward() \* nodeF.backward(resE)

Далее, необходимо вычислить градиенты функции в точке x и y. Для этого определён следующий код:

gradX = gradAx + gradBx

gradY = gradAy + gradBy + gradCy

В результате были получены значения градиента [64.582, 23.503134] в точках x и y, для исходной функции.

Эти значения совпадают с теми, которые были получены с помощью решения через вычислительный граф TensorFlow. Таким образом данная задача решена.

Заключение

В рамках лабораторной работы были выполнены все поставленные задачи.

При решении задания №1 был вручную создан вычислительный граф функции от нескольких переменных и найдена производная этой сложной функции. Все вычисления были проверены с помощью кода на TensorFlow.

При решении задания №2 была создана своя модель типа tf.Module для функции из задания №1. Были сгенерированы данные для обучения и проведена идентификация параметров функции.

При решении задания №3 была создана функция для вычисления значений из задания №1 и её аналог, обёрнутый в декоратор tf.function, который позволил значительно оптимизировать вычислительную нагрузку исходной функции.

При решении задания №4 был написан программный код для вычислительного графа. В качестве функции использовалась функция из задания №1.

Таким образом все задачи лабораторной работы были решены. Был приобретён опыт построения вычислительного графа вручную, на языке программирования Python. Также получен опыт разработки нейронной сети без использования Keras, таким образом укрепив изученные теоретические знания на практике.

Приложения

1. Ссылка на исходный код: <https://github.com/DanSoW/INRTU/blob/main/software-tools-for-artificial-intelligence-tasks/lab1/lab1.ipynb> .
2. Ссылка на статью о распознавании чисел и практическом применении модели (к бонусному заданию №5): <https://habr.com/ru/articles/778546/> .