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 №3  по дисциплине  «Программные средства для задач искусственного интеллекта» |
| «Разработка и тестирование моделей с использованием TensorBoard» |
| 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](#_Toc166820861)

[Решение задания №1 4](#_Toc166820862)

[Решение задания №2 14](#_Toc166820863)

[Заключение 31](#_Toc166820864)

[Приложения 32](#_Toc166820865)

Постановка задач

Необходимо решить следующие задачи:

1. **Задание №1 “Основы TensorBoard”**:
2. Выгрузить данные и проанализировать процесс и результат работы Tensorflow с использованием TensorBoard.
3. Опубликовать результаты TensorBoard.dev и в отчёте указать ссылку.
4. **Задание №2 “Анализ нейронной сети с использованием TensorBoard”**:
5. Использовать стандартную функцию обратного вызова для сохранения статистки, явно указав параметры.
6. Использовать TensorFlow Summary API для сохранения пользовательской статистики
7. Использовать Summary Trace API для изучения параметров вычислительного графа (например, функции-графа из лабораторной 1).
8. Изучить панели инструментов HParams для анализа гиперпараметров, например, параметров оптимизации
9. Зарегестироваться в wandb account here и получить API key here.
10. Опубликовать результаты в wandb и в отчёте указать ссылку.

Решение задания №1

Так как в машинном обучении часто необходимо что-то измерять, для улучшения работы модели, требуется инструмент, который помогает в визуализации различного рода измерений.

TensorBoard – это инструмент для обеспечения измерений и визуализаций, которые необходимы во время рабочего процесса машинного обучения. Он позволяет отслеживать такие показатели эксперимента, как потери и точность, визуализировать график модели, проецировать вложения в пространство с более низкими измерениями и многое другое.

Для того, чтобы в Google Colab добавить поддержку TensorBoard необходимо выполнить следующую строку:

%load\_ext tensorboard

Часто, при повторном запуске элементов блокнота требуется удалить журнал записанных измерений с помощью команды:

!rm -rf ./logs/

В качестве примера для работы с TensorBoard реализуем классическую нейронную сеть для распознавания цифр из стандартного датасета MNIST.

Реализация модели выглядит следующим образом:

# Загрузка датасета

mnist = tf.keras.datasets.mnist

# Распределение датасета по кортежам

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

# Нормализация данных

x\_train, x\_test = x\_train / 255.0, x\_test / 255.0

# Построение последовательной модели

model = tf.keras.models.Sequential()

model.add(tf.keras.layers.Flatten(input\_shape=(28, 28)))

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

model.add(tf.keras.layers.Dropout(0.2))

model.add(tf.keras.layers.Dense(10, activation='softmax'))

# Суммаризация модели

model.summary()

Затем скомпилируем модель:

model.compile(optimizer='adam',

loss='sparse\_categorical\_crossentropy',

metrics=['accuracy'])

В качестве оптимизатора используется Adam, в качестве функции потерь – разряженная категориальная кросс-энтропия, а в качестве метрики был взят стандартный accuracy.

После компиляции модели необходимо настроить использование TensorBoard при обучении нейронной сети.

Для этого используется следующий программный код:

# Формирование пути к директории для хранения логов

log\_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")

# Создание функции обратного вызова TensorBoard

tensorboard\_callback = tf.keras.callbacks.TensorBoard(log\_dir=log\_dir, histogram\_freq=1)

# Обучение модели

model.fit(x=x\_train,

y=y\_train,

epochs=5,

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

callbacks=[tensorboard\_callback])

Сначала указывается путь к директории, в которой будут хранится журналы с измерениями. Затем создаётся функция обратного вызова, в которой указывается директория для хранения журналов и параметр histogram\_freq, позволяющий формировать гистограмму производя вычисления на каждой эпохе.

После того, как сформируется журнал всех необходимых измерений, можно выгрузить данный журнал в TensorBoard с помощью следующей команды:

%tensorboard --logdir logs/fit

В результате получаем следующее:

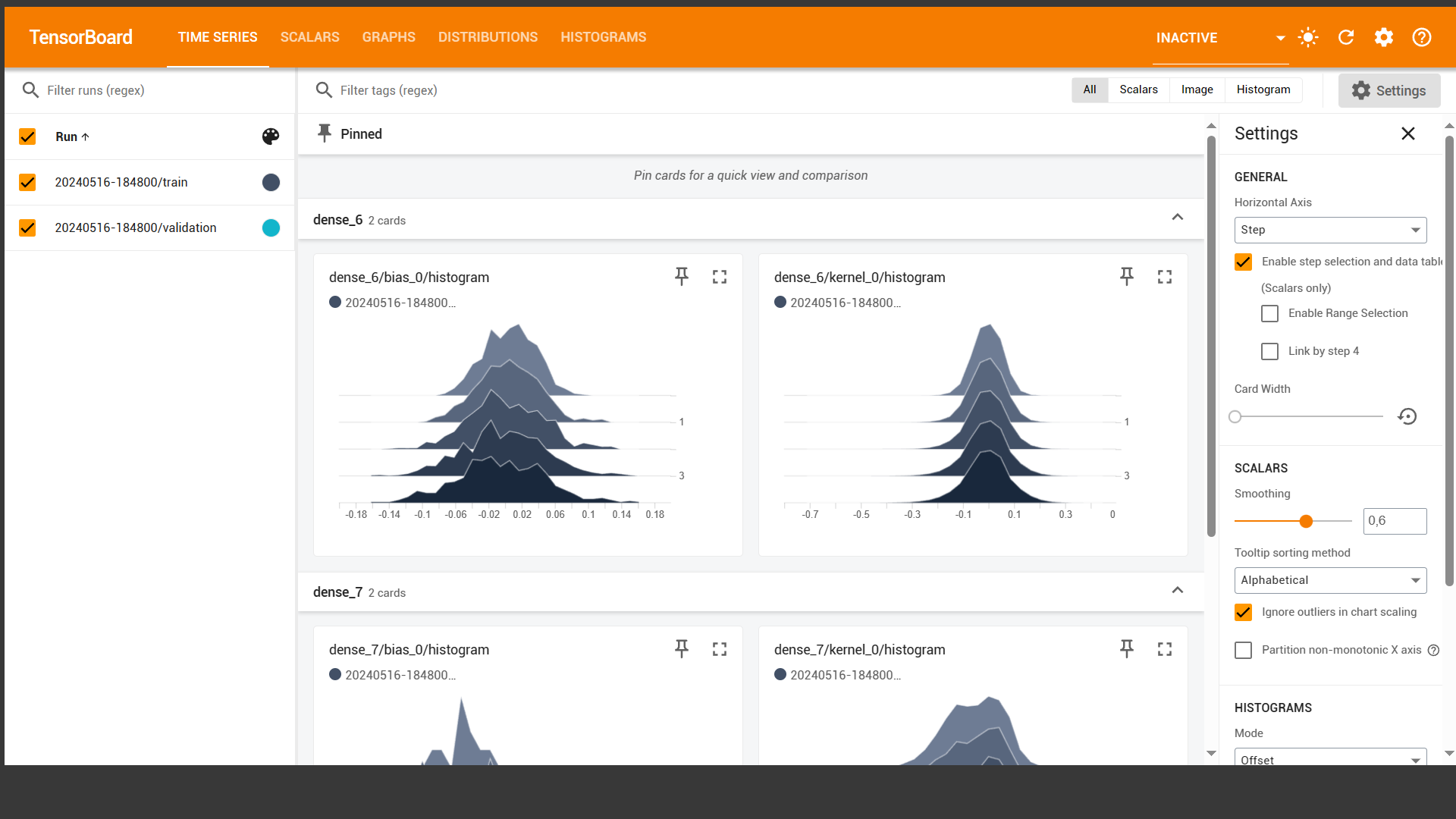


Рисунок 1 – Вывод полученного журнала измерений в TensorBoard

С помощью интерфейса TensorBoard можно, например, увидеть значение метрик по эпохам:

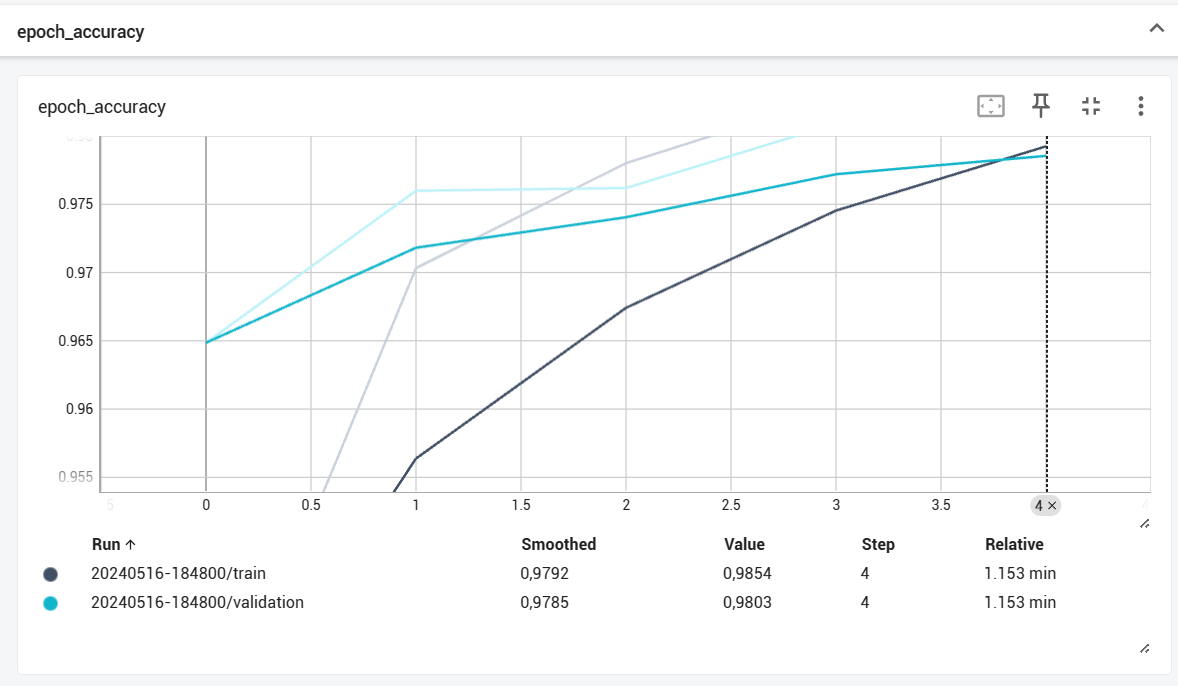


Рисунок 2 – Вывод метрики по эпохам

Или узнать, как менялись значения функции потерь:

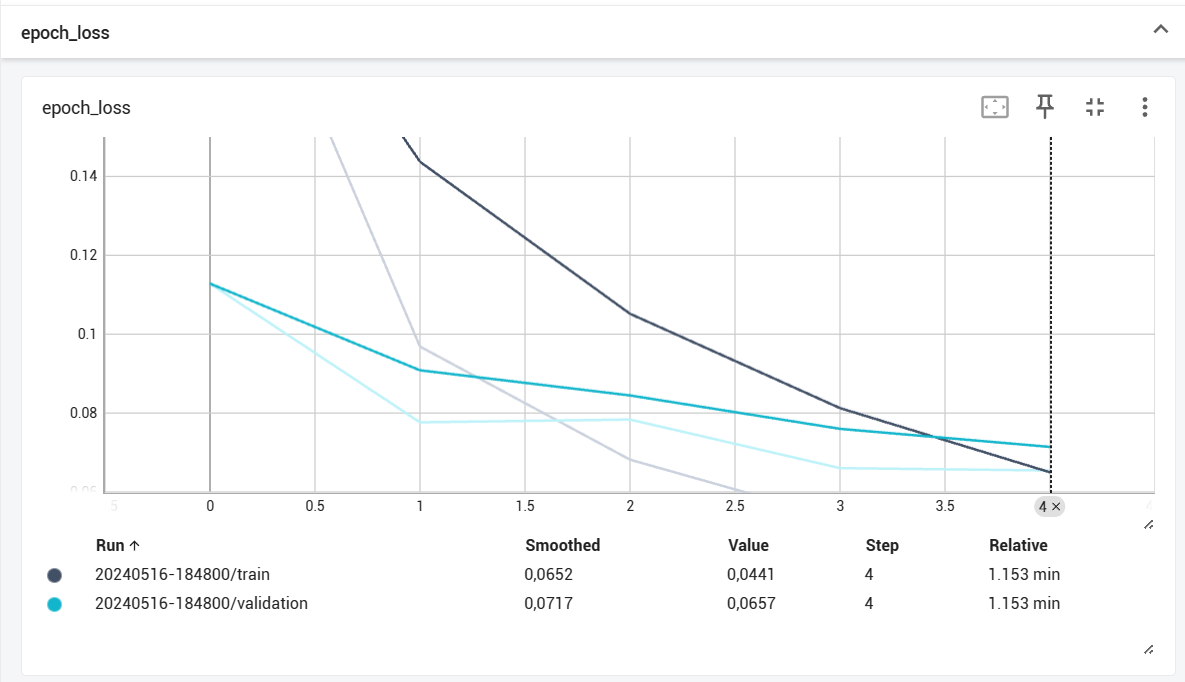


Рисунок 3 – График изменения значений функции потерь

Помимо этого, можно сформировать изображение полученного вычислительного графа нейронной сети:

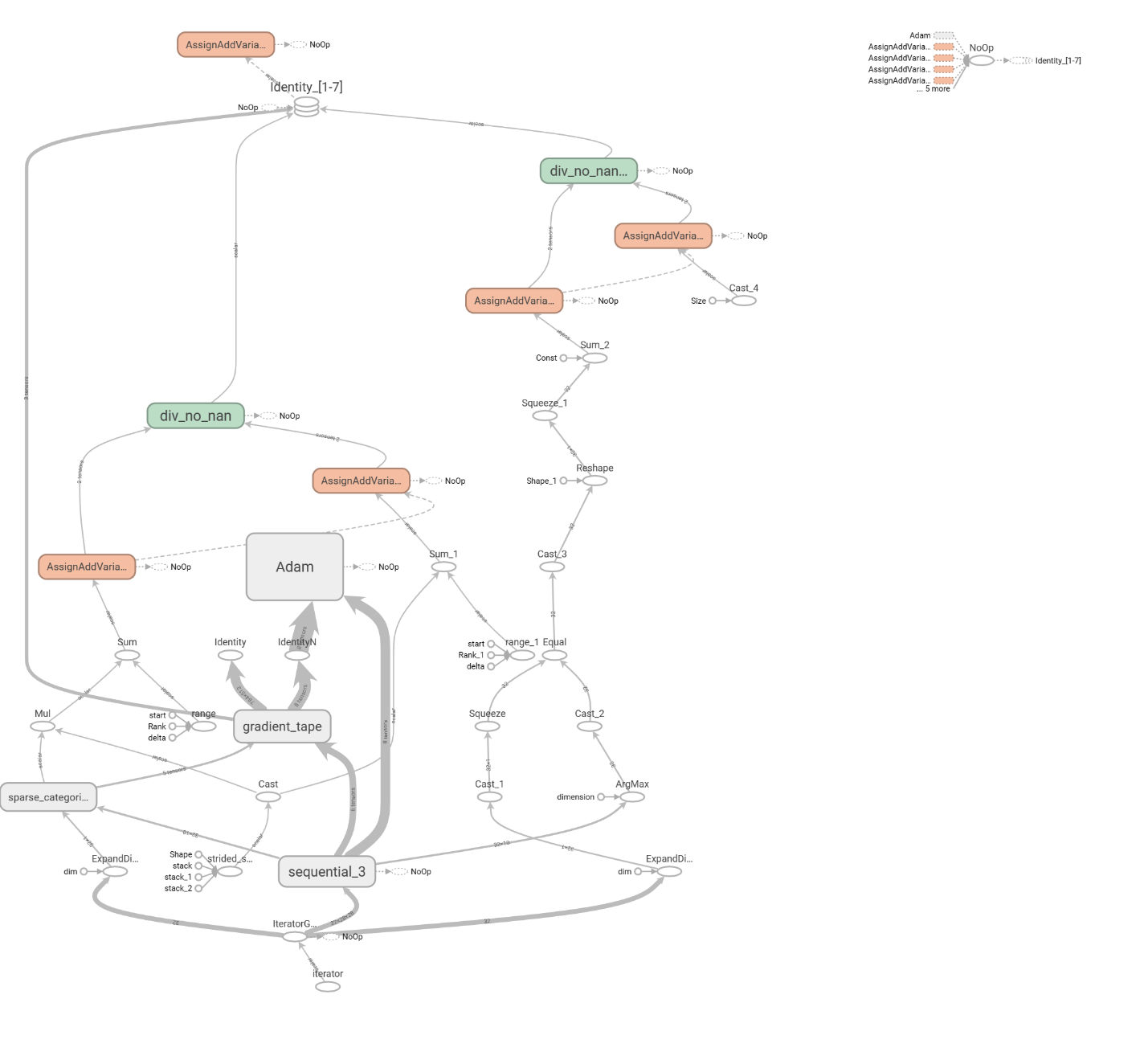


Рисунок 4 – Вычислительный граф нейронной сети

На схеме вычислительного графа можно выделить отдельные важные элементы. Например, выделим операции, которые совместимы с TPU:

TPU – это тензорный процессор, относящийся к классу нейронных процессоров, являющийся специализированной интегральной схемой, разработанной корпорацией Google и предназначенной для использования с библиотекой машинного обучения TensorFlow.

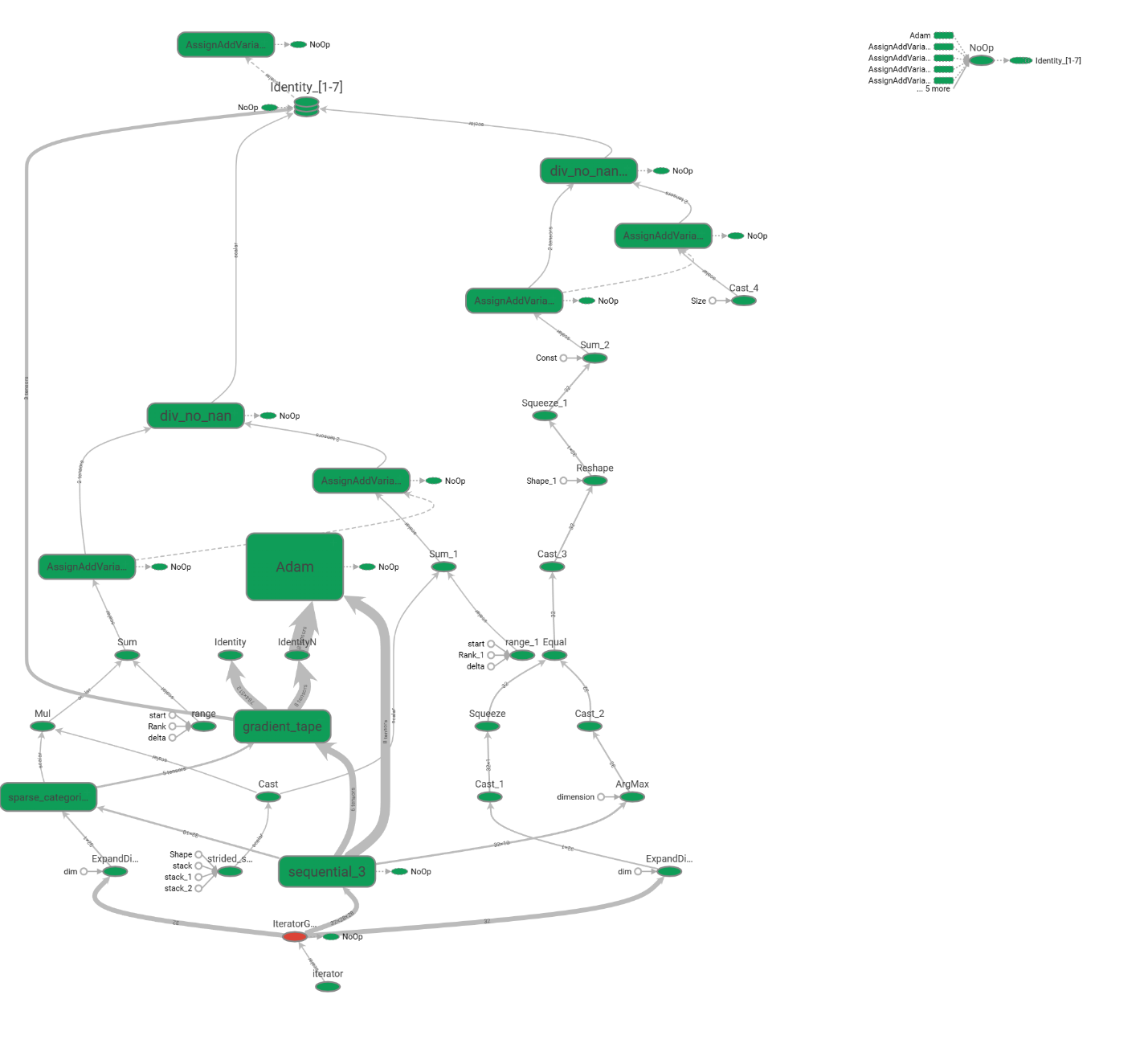


Рисунок 5 – Выделение на вычислительном графе операций совместимых с TPU

Для примера использования tf. summary реализуем программный код для обучения с помощью использования объекта tf.GradientTape.

Для начала загрузим датасет:

# Определение датасета на основе уже имеющегося путём срезов

train\_dataset = tf.data.Dataset.from\_tensor\_slices((x\_train, y\_train))

test\_dataset = tf.data.Dataset.from\_tensor\_slices((x\_test, y\_test))

train\_dataset = train\_dataset.shuffle(60000).batch(64)

test\_dataset = test\_dataset.batch(64)

Затем добавим определение объекта функции разряженной категориальной кросс-энтропии и зададим оптимизатор:

# Создание объекта функции разряжённой категориальной кросс-энтропии

loss\_object = tf.keras.losses.SparseCategoricalCrossentropy()

# Создание оптимизатора

optimizer = tf.keras.optimizers.Adam()

Затем определим функции метрики для тренировочного и тестового шага:

# Создание показателей для отслеживания и регистравии во время обучения

# Для шага обучения

train\_loss = tf.keras.metrics.Mean('train\_loss', dtype=tf.float32)

train\_accuracy = tf.keras.metrics.SparseCategoricalAccuracy('train\_accuracy')

# Для шага обучения

test\_loss = tf.keras.metrics.Mean('test\_loss', dtype=tf.float32)

test\_accuracy = tf.keras.metrics.SparseCategoricalAccuracy('test\_accuracy')

Теперь определим функции-шаги для обучения модели и её тестирования:

# Определение шага для обучения модели

def train\_step(model, optimizer, x\_train, y\_train):

# Создание контекста tf.GradientTape

with tf.GradientTape() as tape:

# Вычисление предсказаний модели

predictions = model(x\_train, training=True)

# Вычисление значения потери

loss = loss\_object(y\_train, predictions)

# Вычисление градиентов

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

# Выполнение оптимизации весов модели

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

# Вычисление метрик

train\_loss(loss)

train\_accuracy(y\_train, predictions)

# Определение шага для тестирования модели

def test\_step(model, x\_test, y\_test):

# Вычисление предсказаний модели

predictions = model(x\_test)

# Вычисление значений функции потерь модели

loss = loss\_object(y\_test, predictions)

# Вычисление метрик

test\_loss(loss)

test\_accuracy(y\_test, predictions)

В этих функциях просто происходит одна итерация обучения (train\_step) и одна итерация тестирования модели (test\_step).

Теперь добавим журналы для просмотра информации о обучении и тестировании модели:

# Текущее время

current\_time = datetime.datetime.now().strftime("%Y%m%d-%H%M%S")

# Запись в журнал информации о обучении модели

train\_log\_dir = 'logs/gradient\_tape/' + current\_time + '/train'

# Запись в журнал информации о тестировании модели

test\_log\_dir = 'logs/gradient\_tape/' + current\_time + '/test'

# Создание "писателей" для файлов журнала

train\_summary\_writer = tf.summary.create\_file\_writer(train\_log\_dir)

test\_summary\_writer = tf.summary.create\_file\_writer(test\_log\_dir)

Осталось выполнить обучение модели нейронной сети:

EPOCHS = 5

# Прохождение 5-ти эпох обучения

for epoch in range(EPOCHS):

# Выполнение шага обучения для набора данных

for (x\_train, y\_train) in train\_dataset:

train\_step(model, optimizer, x\_train, y\_train)

# Контекст "писателя" для тренировки по-умолчанию

with train\_summary\_writer.as\_default():

# Регистрация скалярных значений функции потерь и метрики

tf.summary.scalar('loss', train\_loss.result(), step=epoch)

tf.summary.scalar('accuracy', train\_accuracy.result(), step=epoch)

# Выполнение шага тестирования для набора данных

for (x\_test, y\_test) in test\_dataset:

test\_step(model, x\_test, y\_test)

# Контекст "писателя" для теста по-умолчанию

with test\_summary\_writer.as\_default():

# Регистрация скалярных значений функции потерь и метрики

tf.summary.scalar('loss', test\_loss.result(), step=epoch)

tf.summary.scalar('accuracy', test\_accuracy.result(), step=epoch)

# Шаблон вывода информации в консоль

template = 'Epoch {}, Loss: {}, Accuracy: {}, Test Loss: {}, Test Accuracy: {}'

# Вывод информации в консоль о прохождении конкретной эпохи

print (template.format(epoch+1,

train\_loss.result(),

train\_accuracy.result()\*100,

test\_loss.result(),

test\_accuracy.result()\*100))

# Сброс метрик для каждой эпохи

train\_loss.reset\_states()

test\_loss.reset\_states()

train\_accuracy.reset\_states()

test\_accuracy.reset\_states()

В рамках обучения нейронной сети мы выполняем тренировочный шаг по всему набору данных для обучения, затем регистрируем скалярные значения функции потерь и метрик и тоже самое выполняем для тестирования. В конце обучения сбрасываем метрики для каждой эпохи.

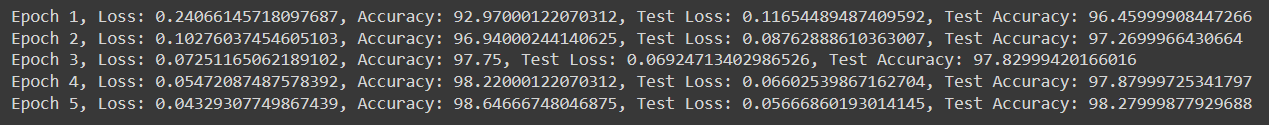


Рисунок 6 – Результат обучения модели нейронной сети

Теперь для выгрузки данных в TensorBoard следует выполнить команду:

%tensorboard --logdir logs/gradient\_tape

В результате получаем следующее:

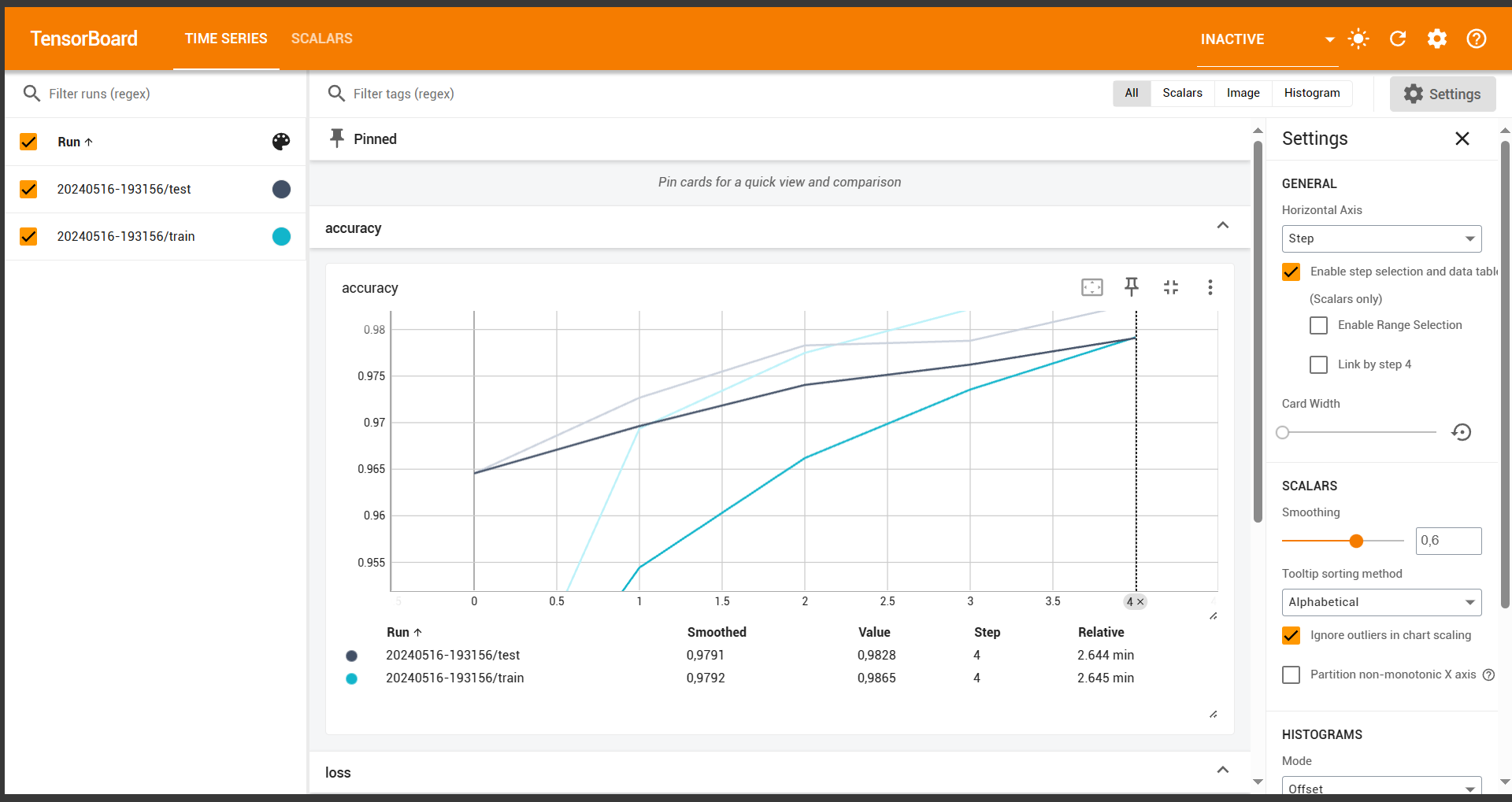


Рисунок 7 – Результат выгрузки данных журналов в TensorBoard

В TensorBoard выводятся те скалярные данные, которые мы регистрировали на протяжении всего этапа обучения и тестирования модели:

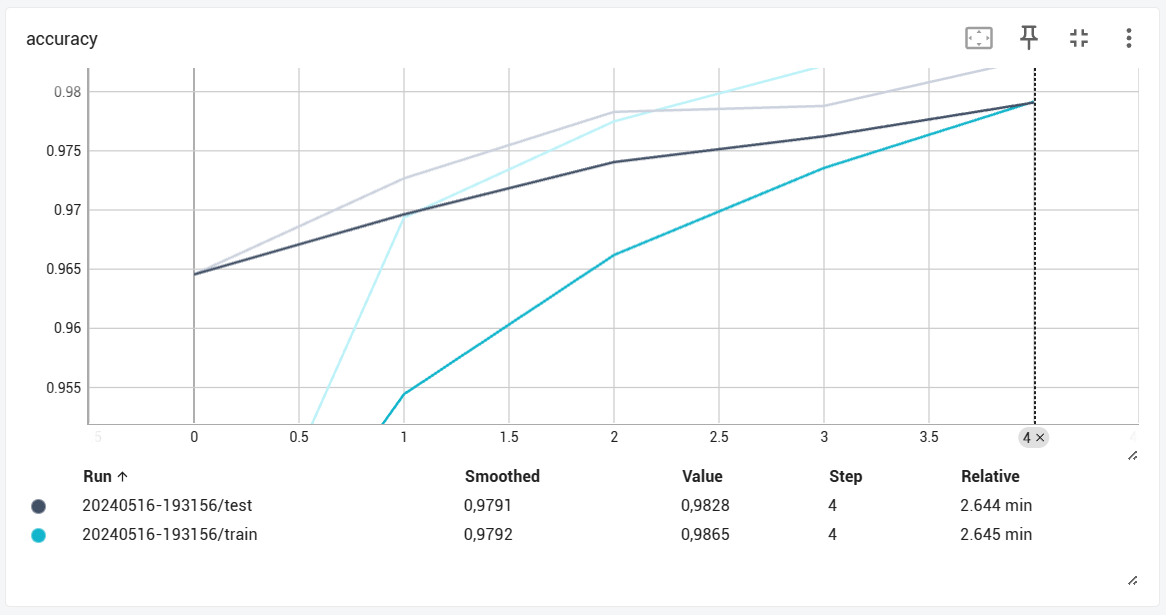


Рисунок 8 – Вывод метрики accuracy

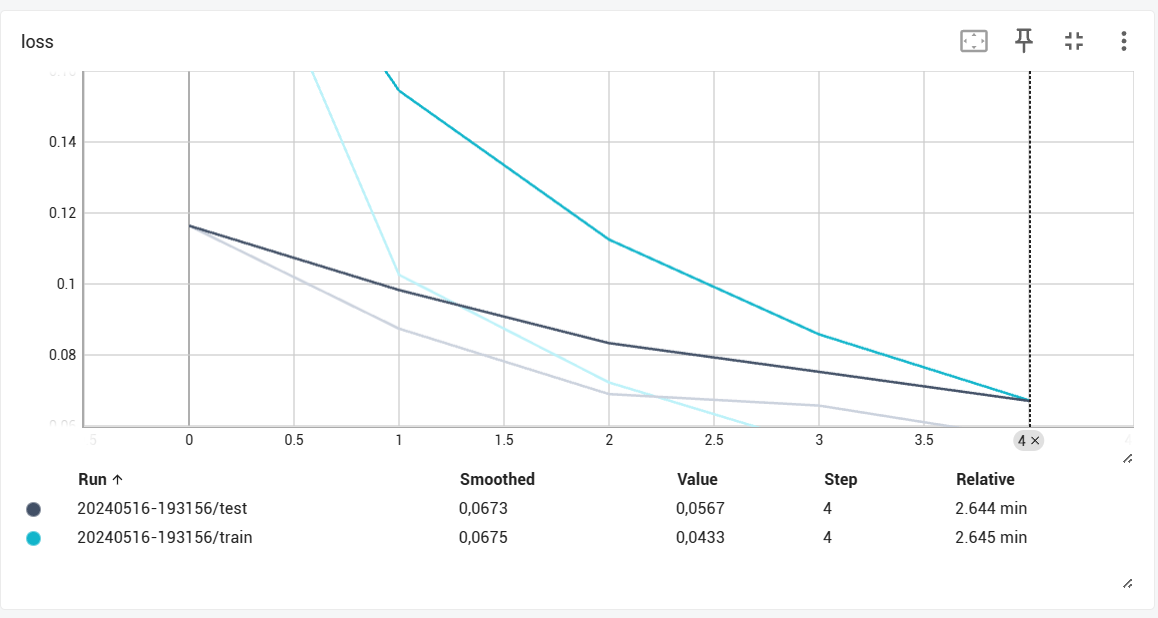


Рисунок 9 – Вывод значений функции потерь

Теперь опубликуем результаты работы на TensorBoard.dev.

TensorBoard.dev – это бесплатная общественная служба, которая позволяет загружать журналы TensorBoard и получать постоянную ссылку, которая может использоваться совместно со всеми в научных статьях, блогах, социальных медиа и т.д. Это может дать лучшую воспроизводимость и сотрудничество.

Для публикации полученного журнала на TensorBoard.dev воспользуемся следующей командой:

!tensorboard dev upload \

--logdir logs/fit \

--name "My first experiment" \

--description "Simple comparison of several hyperparameters" \

--one\_shot

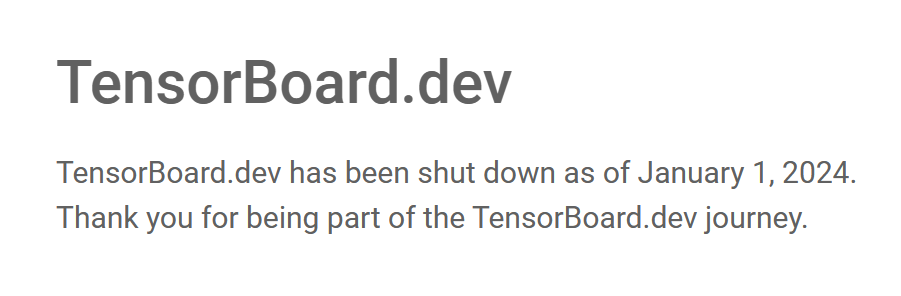


Рисунок 10 – Прекращение работы TensorBoard.dev с 1 января 2024 года

К сожалению, в настоящий момент опубликовать результаты на TensorBoard.dev не представляется возможным, т.к. данный проект прекратил своё существование с 1 января 2024 года. При попытке выполнить код для публикации результатов в TensorBoard.dev выходит следующий лог:

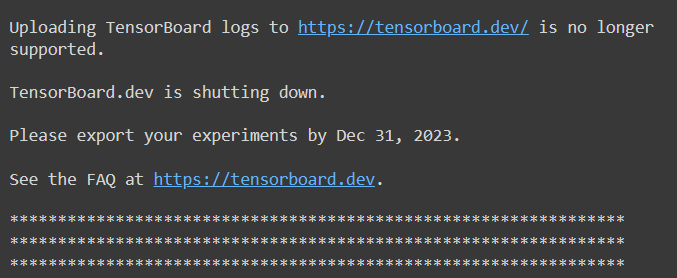


Рисунок 11 – Прекращение поддержки TensorBoard.dev

Таким образом можно считать, что первая задача полностью решена.

Решение задания №2

Для реализации примера использования стандартной функции обратного вызова для сохранения статистики будем использовать нейронную сеть из второй лабораторной работы (создание Y-DAG нейронной сети).

К уже существующей реализации Y-DAG нейронной сети (см. приложение 2) на этапе обучения модели добавим следующий программный код:

# Формирование пути к директории для хранения логов

log\_dir = "logs/ydag/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")

# Создание функции обратного вызова TensorBoard

ydag\_tensorboard\_callback = tf.keras.callbacks.TensorBoard(log\_dir=log\_dir, histogram\_freq=1)

# 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=[ydag\_tensorboard\_callback])

После того, как модель была обучена, для запуска TensorBoard необходимо использовать следующую команду:

%tensorboard --logdir logs/ydag

После открытия TensorBoard получаем следующую картину метрик по эпохам:

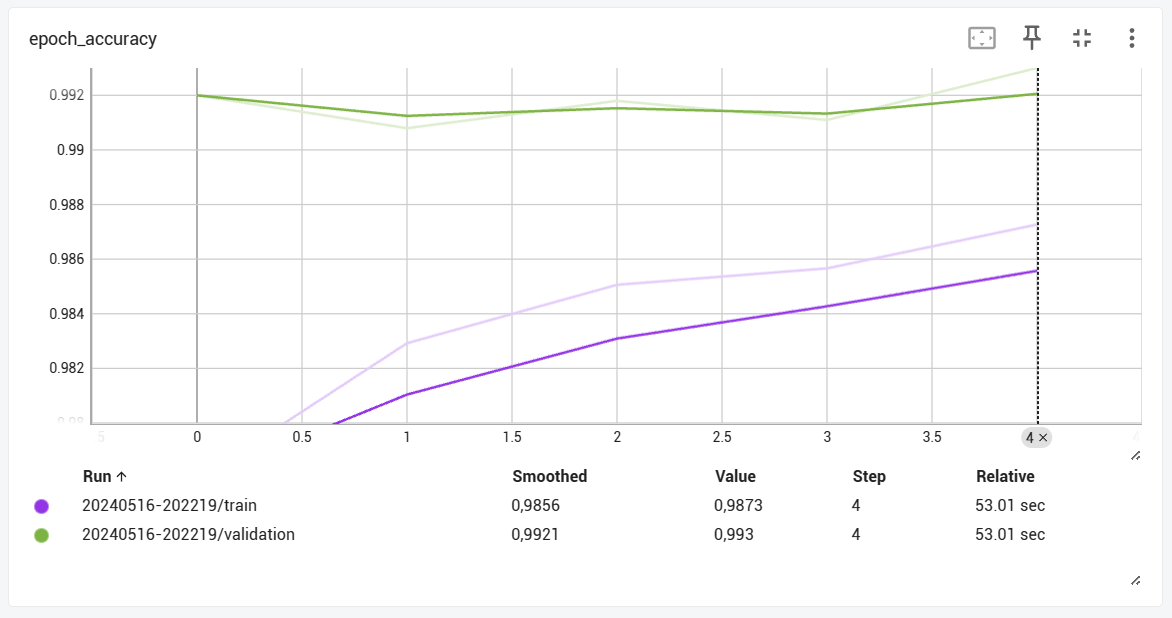


Рисунок 12 – График accuracy по каждой эпохе

Также, можно наблюдать каким образом меняются значения функции потерь:

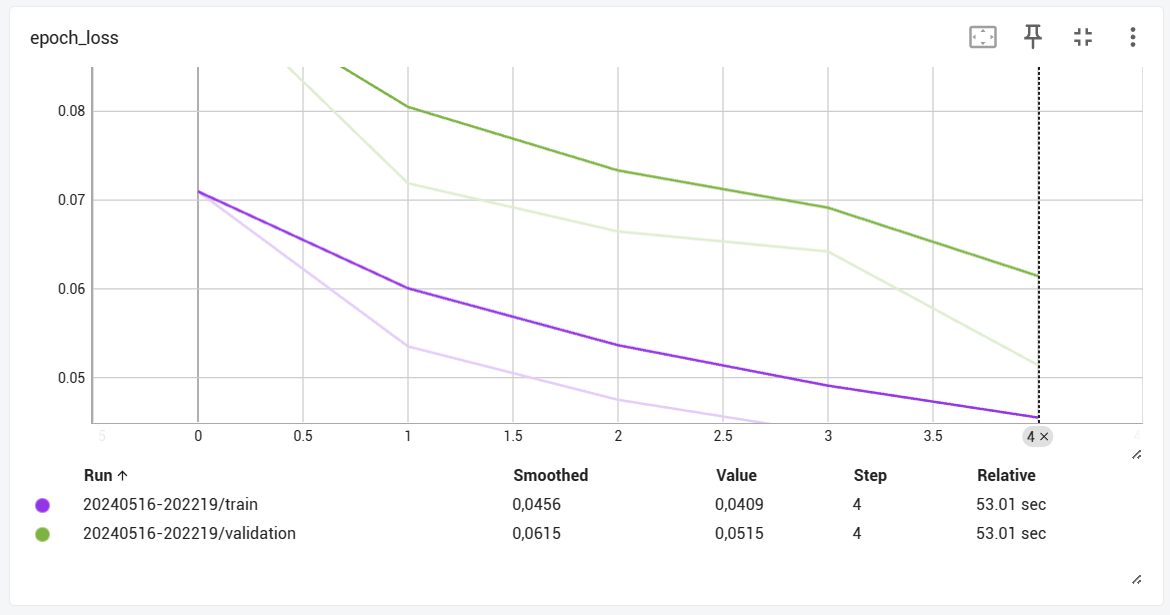


Рисунок 13 – График изменения значения функции потерь

Можно также определить структуру вычислительного графа для данной нейронной сети:

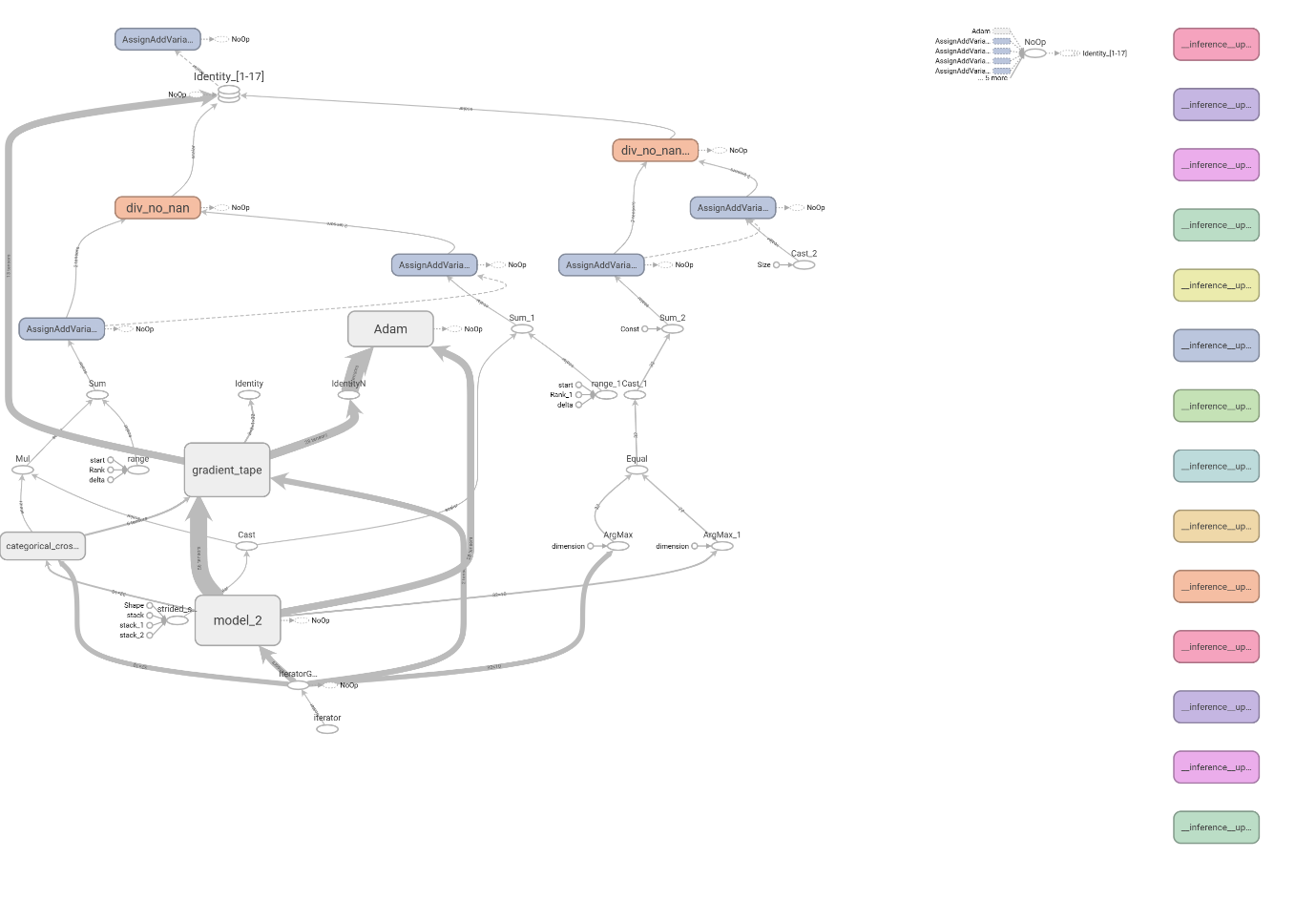


Рисунок 14 – Вычислительный граф для нейронной сети

Также можно сгенерировать и более привычную структуру нейронной сети:

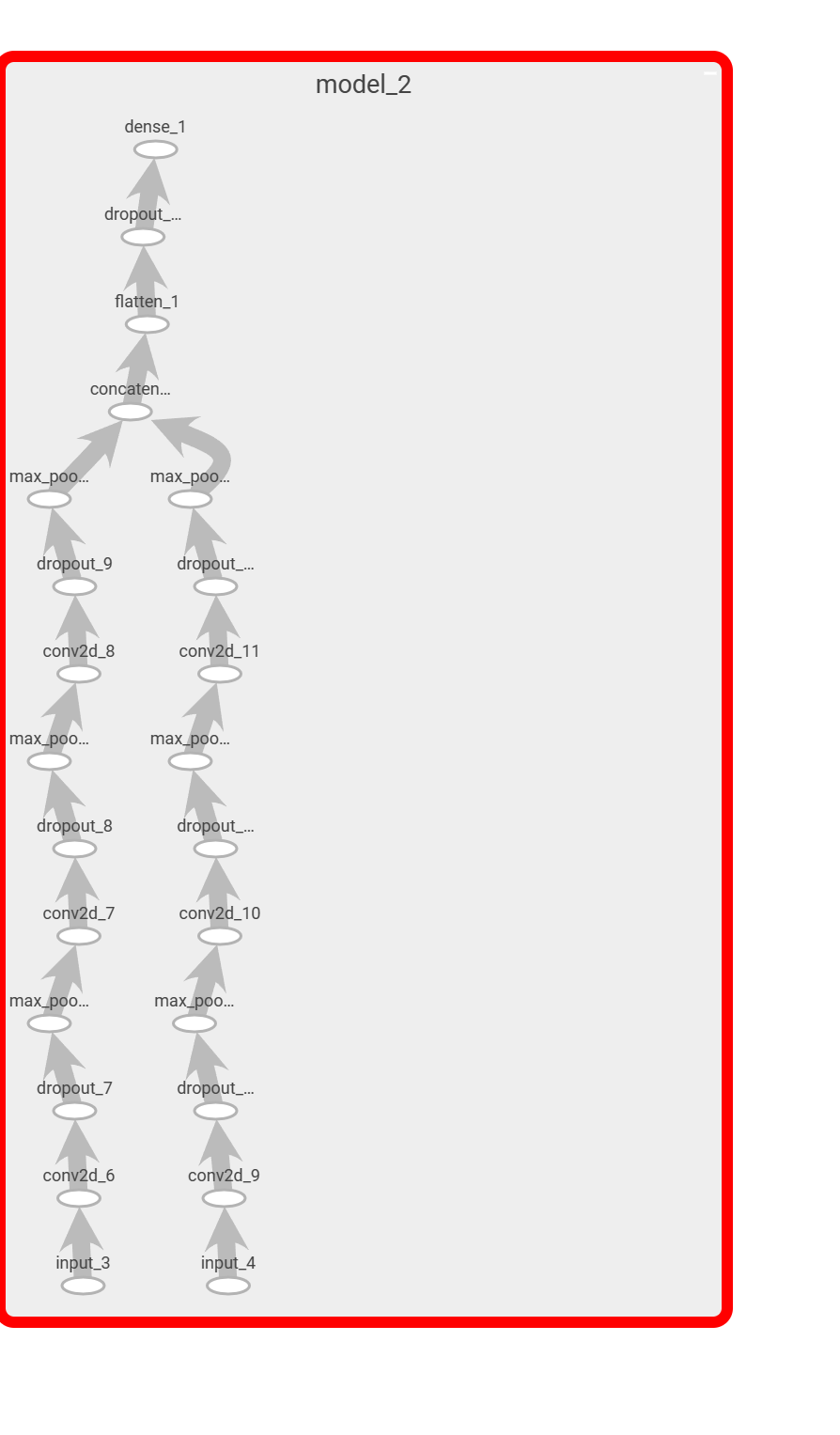


Рисунок 15 – Граф слоёв Y-DAG нейронной сети

Как видно из рисунка 15 граф слоёв полностью соответствует Y-DAG версии нейронной сети.

Для использования TensorFlow Summary API с сохранением пользовательской статистики изменим код на этапе подготовки данных и обучения нейронной сети.

Сначала сделаем срез данных для обучения и тестирования модели для удобства:

# Определение датасета на основе уже имеющегося путём срезов

train\_dataset = tf.data.Dataset.from\_tensor\_slices((x\_train, y\_train))

test\_dataset = tf.data.Dataset.from\_tensor\_slices((x\_test, y\_test))

train\_dataset = train\_dataset.shuffle(60000).batch(64)

test\_dataset = test\_dataset.batch(64)

Затем создадим объект для вычисления значения потерь и оптимизатор Adam:

# Создание объекта функции разряжённой категориальной кросс-энтропии

loss\_object = tf.keras.losses.CategoricalCrossentropy()

# Создание оптимизатора

optimizer = tf.keras.optimizers.Adam()

Далее определим метрики:

# Создание показателей для отслеживания и регистравии во время обучения

# Для шага обучения

train\_loss = tf.keras.metrics.Mean('train\_loss', dtype=tf.float32)

train\_accuracy = tf.keras.metrics.CategoricalAccuracy('train\_accuracy')

# Для шага обучения

test\_loss = tf.keras.metrics.Mean('test\_loss', dtype=tf.float32)

test\_accuracy = tf.keras.metrics.CategoricalAccuracy('test\_accuracy')

А затем “писателей” и пути к директориям, где будут хранится регистрации скалярных значений:

# Текущее время

current\_time = datetime.datetime.now().strftime("%Y%m%d-%H%M%S")

# Запись в журнал информации о обучении модели

train\_log\_dir = 'logs/ydag\_gradient\_tape/' + current\_time + '/train'

# Запись в журнал информации о тестировании модели

test\_log\_dir = 'logs/ydag\_gradient\_tape/' + current\_time + '/test'

# Создание "писателей" для файлов журнала

train\_summary\_writer = tf.summary.create\_file\_writer(train\_log\_dir)

test\_summary\_writer = tf.summary.create\_file\_writer(test\_log\_dir)

Также определим отдельные шаги обучения нейронной сети:

# Определение шага для обучения модели

def train\_step(model, optimizer, x\_train, y\_train):

# Создание контекста tf.GradientTape

with tf.GradientTape() as tape:

# Вычисление предсказаний модели

predictions = model([x\_train, x\_train], training=True)

# Вычисление значения потери

loss = loss\_object(y\_train, predictions)

# Вычисление градиентов

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

# Выполнение оптимизации весов модели

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

# Вычисление метрик

train\_loss(loss)

train\_accuracy(y\_train, predictions)

# Определение шага для тестирования модели

def test\_step(model, x\_test, y\_test):

# Вычисление предсказаний модели

predictions = model([x\_test, x\_test])

# Вычисление значений функции потерь модели

loss = loss\_object(y\_test, predictions)

# Вычисление метрик

test\_loss(loss)

test\_accuracy(y\_test, predictions)

И, наконец, запустим процесс обучения нейронной сети:

EPOCHS = 5

# Прохождение 5-ти эпох обучения

for epoch in range(EPOCHS):

# Выполнение шага обучения для набора данных

for (x\_train, y\_train) in train\_dataset:

train\_step(model, optimizer, x\_train, y\_train)

# Контекст "писателя" для тренировки по-умолчанию

with train\_summary\_writer.as\_default():

# Регистрация скалярных значений функции потерь и метрики

tf.summary.scalar('loss', train\_loss.result(), step=epoch)

tf.summary.scalar('accuracy', train\_accuracy.result(), step=epoch)

# Выполнение шага тестирования для набора данных

for (x\_test, y\_test) in test\_dataset:

test\_step(model, x\_test, y\_test)

# Контекст "писателя" для теста по-умолчанию

with test\_summary\_writer.as\_default():

# Регистрация скалярных значений функции потерь и метрики

tf.summary.scalar('loss', test\_loss.result(), step=epoch)

tf.summary.scalar('accuracy', test\_accuracy.result(), step=epoch)

# Шаблон вывода информации в консоль

template = 'Epoch {}, Loss: {}, Accuracy: {}, Test Loss: {}, Test Accuracy: {}'

# Вывод информации в консоль о прохождении конкретной эпохи

print (template.format(epoch+1,

train\_loss.result(),

train\_accuracy.result()\*100,

test\_loss.result(),

test\_accuracy.result()\*100))

# Сброс метрик для каждой эпохи

train\_loss.reset\_states()

test\_loss.reset\_states()

train\_accuracy.reset\_states()

test\_accuracy.reset\_states()

В результате будет готов журнал, загрузить в TensorBoard который можно с помощью следующей команды:

%tensorboard --logdir logs/ydag\_gradient\_tape

На рисунках 16-17 представлены полученные значения пользовательских метрик.

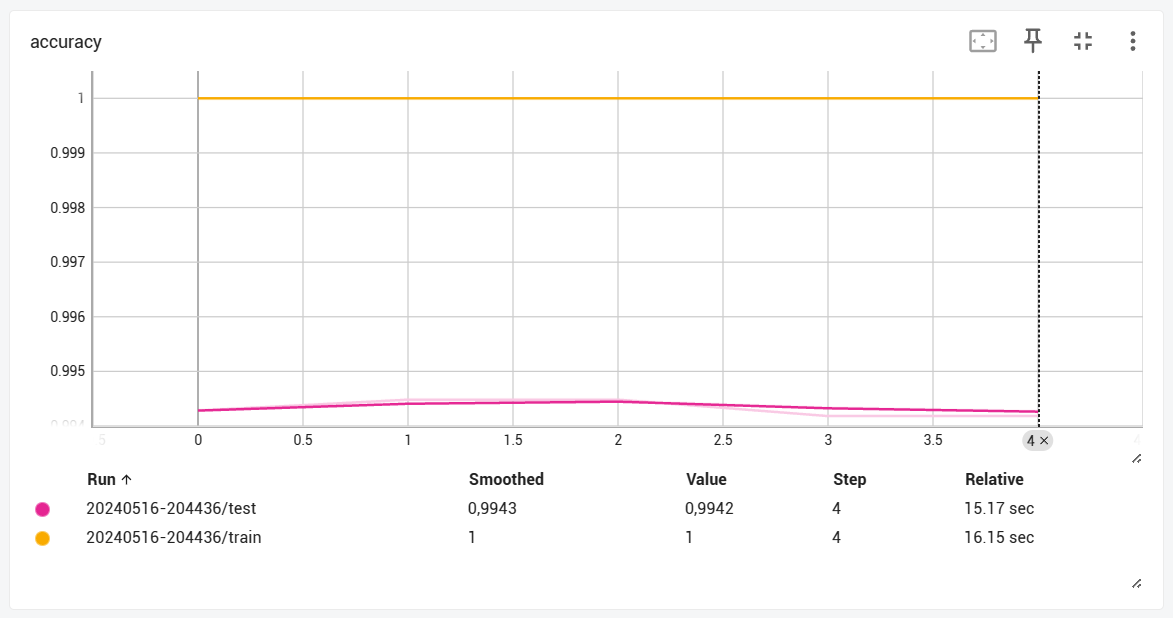


Рисунок 16 – График вычисления значений метрики

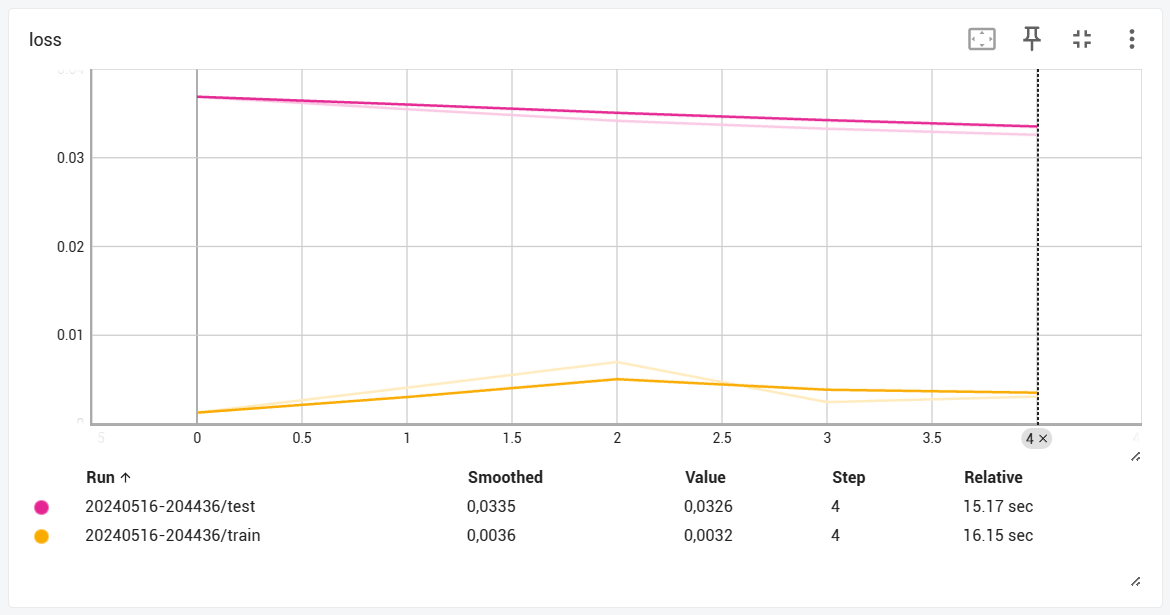


Рисунок 17 – График значений функции потерь

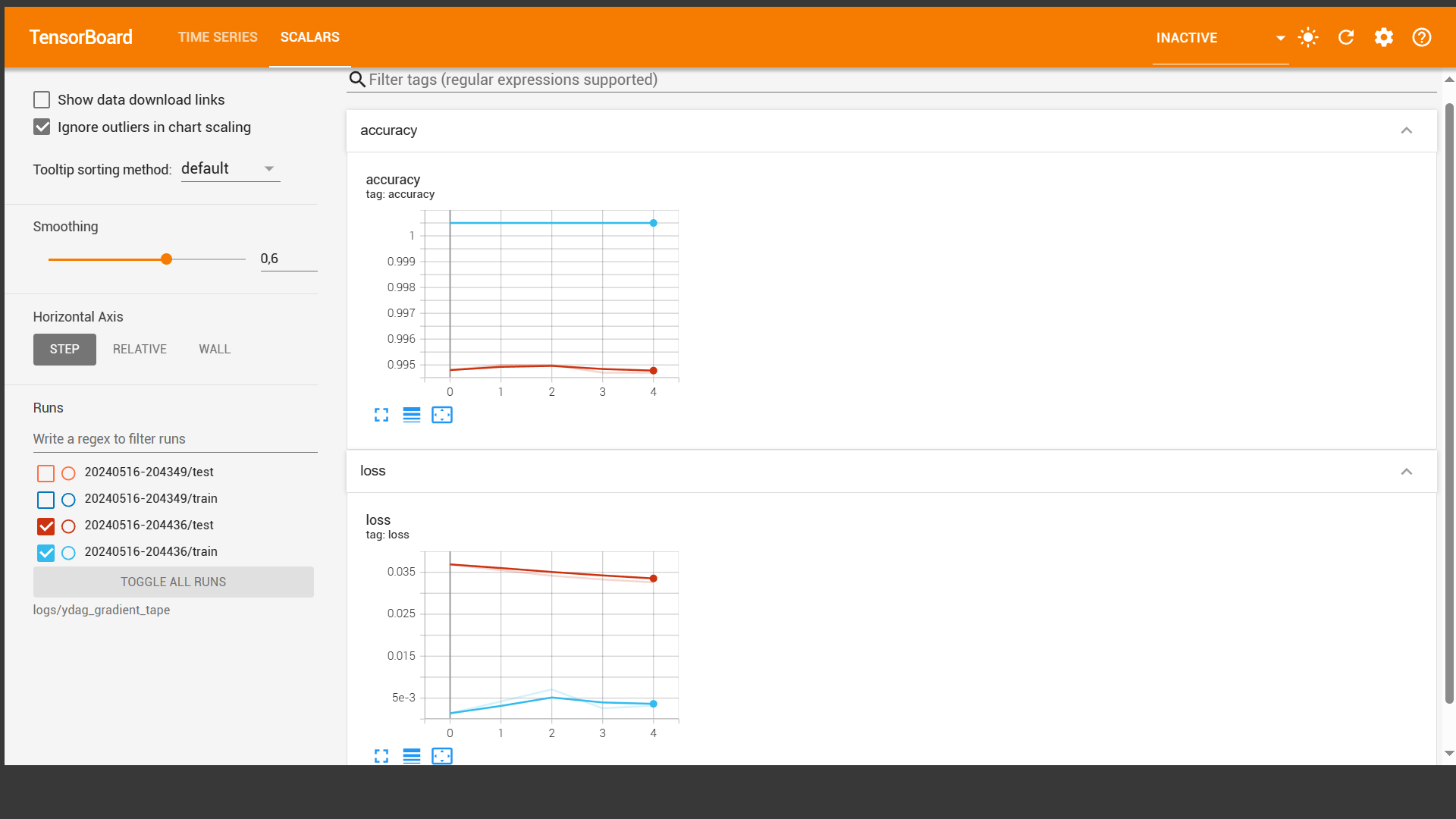


Рисунок 18 – Оба графика в одной вкладке “SCALARS”

Для использования Summary Trace API с целью изучения параметров вычислительного графа будем использовать сложную функцию из первой лабораторной работы (см. приложение 1).

Определим функцию, над которой будет происходить трассировка с декоратором tf.function (с целью оптимизации вычислений):

# Определение функции для трассировки

@tf.function

def hard\_func(x, y):

# Функция из первой лабораторной работы

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

Затем определим директорию для записи журнала и сгенерируем набор данных для функции:

# Фиксация текущего времени

stamp = datetime.now().strftime("%Y%m%d-%H%M%S")

# Определение пути к директории для регистрации значений журнала

logdir = 'logs/hard\_func/%s' % stamp

# Определение "писателя" в конкретный файл журнала

writer = tf.summary.create\_file\_writer(logdir)

# Генерация рандомных значений нормального распределения

x = tf.random.uniform((3, 3))

y = tf.random.uniform((3, 3))

Теперь настроим трассировку на запись графиков вычислений и информации о профилировании:

tf.summary.trace\_on(graph=True, profiler=True)

И финальный этап:

# Формирование значений сложной функции

z = hard\_func(x, y)

# Вход в контекст "писатель" по-умолчанию

with writer.as\_default():

# Запись трассировки в журнал

tf.summary.trace\_export(

name="hard\_func\_trace",

step=0,

profiler\_outdir=logdir)

Теперь можно увидеть данные о журнале с помощью команды:

%tensorboard --logdir logs/hard\_func

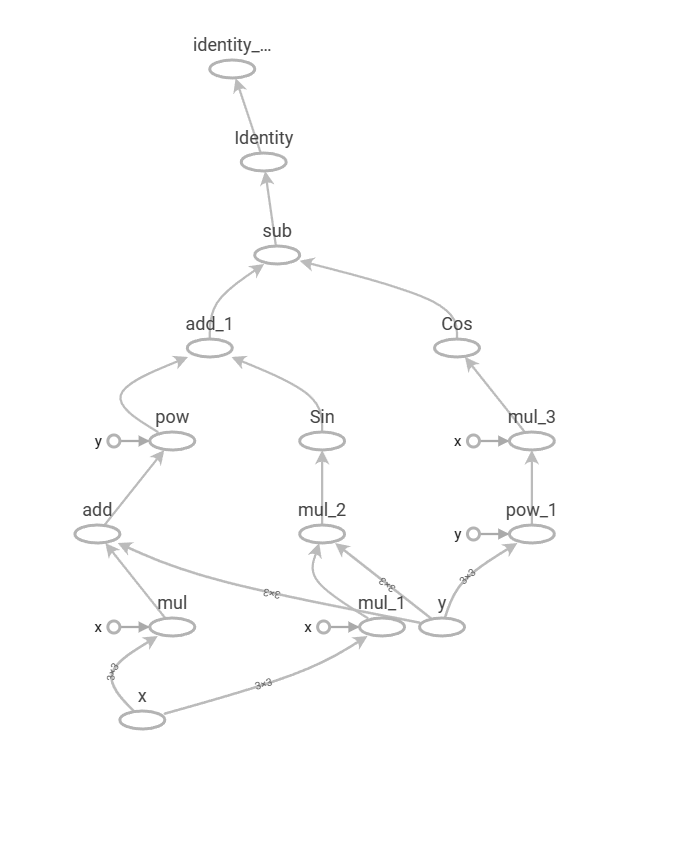


Рисунок 19 – Вычислительный граф сложной функции

Если мы сравним полученный вычислительный граф функции с результатом из первой лабораторной работы (см. рис. 20), то получим, что вычислительные графы концептуально очень похожи.

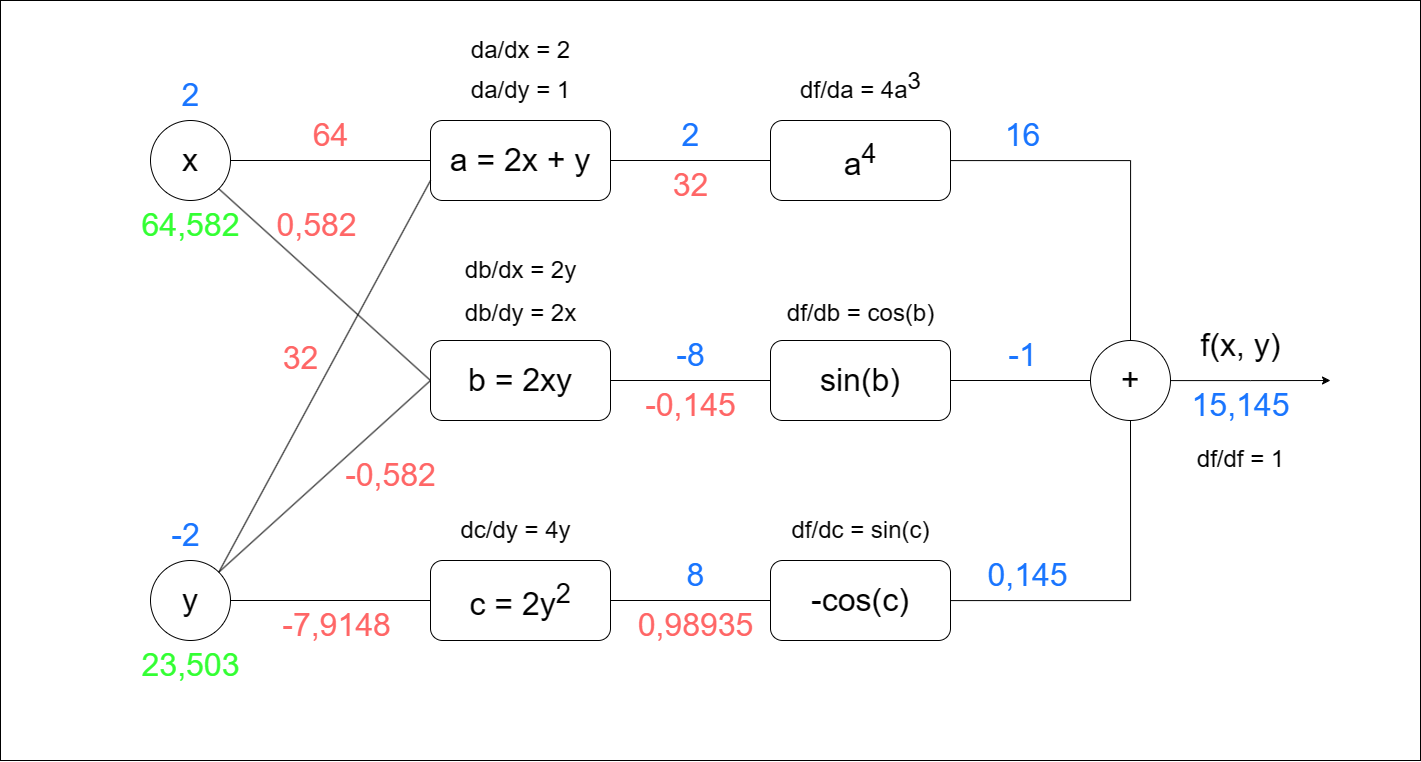


Рисунок 20 – Вычислительный граф сложной функции

Таким образом дополнительно подтверждаем, что вычислительный граф в первой лабораторной работе определён верно.

В рамках изучения панели инструментов HParams для анализа гиперпараметров будем использовать код из второй лабораторной работы (см. приложение 2).

Для начала необходимо импортировать API для работы с гиперпараметрами:

from tensorboard.plugins.hparams import api as hp

В рамках данной лабораторной работы будет произведена работа с такими гиперпараметрами как число нейронов (num\_units), значение dropout и функция-оптимизатор Теперь определим объекты, которые будут рандомно подбирать гиперпараметры исходя из дискретной / интервальной природы генерации данных:

# Определение случайных значений для юнитов из дискретных значений

HP\_NUM\_UNITS = hp.HParam('num\_units', hp.Discrete([300, 200,512]))

# Определение случайных значений для dropout'а из определённого интервала

HP\_DROPOUT = hp.HParam('dropout', hp.RealInterval(0.1,0.5))

# Определение случайных значений для оптимизатора из дискретных строковых значений

HP\_OPTIMIZER = hp.HParam('optimizer', hp.Discrete(['adam', 'sgd', 'rmsprop']))

Теперь добавим код для записи результатов настройки гиперпараметров в журнал:

with tf.summary.create\_file\_writer('logs/hparam\_tuning').as\_default():

hp.hparams\_config(

hparams=[HP\_NUM\_UNITS, HP\_DROPOUT, HP\_OPTIMIZER],

metrics=[hp.Metric(METRIC\_ACCURACY, display\_name='Accuracy')],)

И также вынесем создание модели нейронной сети в отдельную функцию с заменых литеральных параметров на определённые нами гиперпараметры:

def create\_model(hparams):

# 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(hparams[HP\_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(hparams[HP\_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(hparams[HP\_DROPOUT])(x)

x = MaxPooling2D()(x)

# Правая ветвь сети Y-DAG

# 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(hparams[HP\_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(hparams[HP\_DROPOUT])(y)

#y = TfConv2D(kernel=(kernel\_size, kernel\_size), channels\_count=(2 \* filters), acivate='relu')(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(hparams[HP\_DROPOUT])(y)

y = MaxPooling2D()(y)

# Объединение ветвей нейронной сети

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

# Добавление новых слоёв

# Слой сжатия многомерных тензоров в одно измерение

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

merge\_xy = Dense(hparams[HP\_NUM\_UNITS], activation='relu')(merge\_xy)

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

# Выходной слой

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

# Сборка модели нейронной сети

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

model.compile(optimizer=hparams[HP\_OPTIMIZER],

loss='categorical\_crossentropy',

metrics=['accuracy'])

# Обучение модели

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

y\_train,

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

epochs=3,

batch\_size=batch\_size)

# Проверка работы модели

loss, accuracy = model.evaluate([x\_test, x\_test],

y\_test,

batch\_size=batch\_size,

verbose=2)

return accuracy

Теперь определим новую функцию для проведения одного эксперимента:

# Определение функции для проведения эксперимента

def experiment(experiment\_dir, hparams):

# Контекст "писателя" в определённую директорию

with tf.summary.create\_file\_writer(experiment\_dir).as\_default():

# Определение параметров

hp.hparams(hparams)

# Создание и обучение модели по параметрам

accuracy = create\_model(hparams)

# Регистрация скалярного значения в журнал

tf.summary.scalar(METRIC\_ACCURACY, accuracy, step=1)

И, наконец, определяем полный код проведения эксперимента над нейронной сетью:

# Номер эксперимента

experiment\_no = 0

# Выполняем цикл для разных значений параметра num\_units из домена HP\_NUM\_UNITS.domain.values

for num\_units in HP\_NUM\_UNITS.domain.values:

# Выполняем цикл для разных значений параметра dropout\_rate из домена HP\_DROPOUT.domain

for dropout\_rate in (HP\_DROPOUT.domain.min\_value, HP\_DROPOUT.domain.max\_value):

# Выполняем цикл для разных значений параметра optimizer из домена HP\_OPTIMIZER.domain

for optimizer in HP\_OPTIMIZER.domain.values:

# Формирование параметров

hparams = {

HP\_NUM\_UNITS: num\_units,

HP\_DROPOUT: dropout\_rate,

HP\_OPTIMIZER: optimizer,}

# Формирование имени эксперимента

experiment\_name = f'Experiment {experiment\_no}'

# Вывод имени эксперимента

print(f'Starting Experiment: {experiment\_name}')

# Вывод значений параметров

print({h.name: hparams[h] for h in hparams})

# Запись значений параметров в журнал

experiment('logs/hparam\_tuning/' + experiment\_name, hparams)

experiment\_no += 1

В результате получаем следующий список экспериментов с гиперпараметрами:

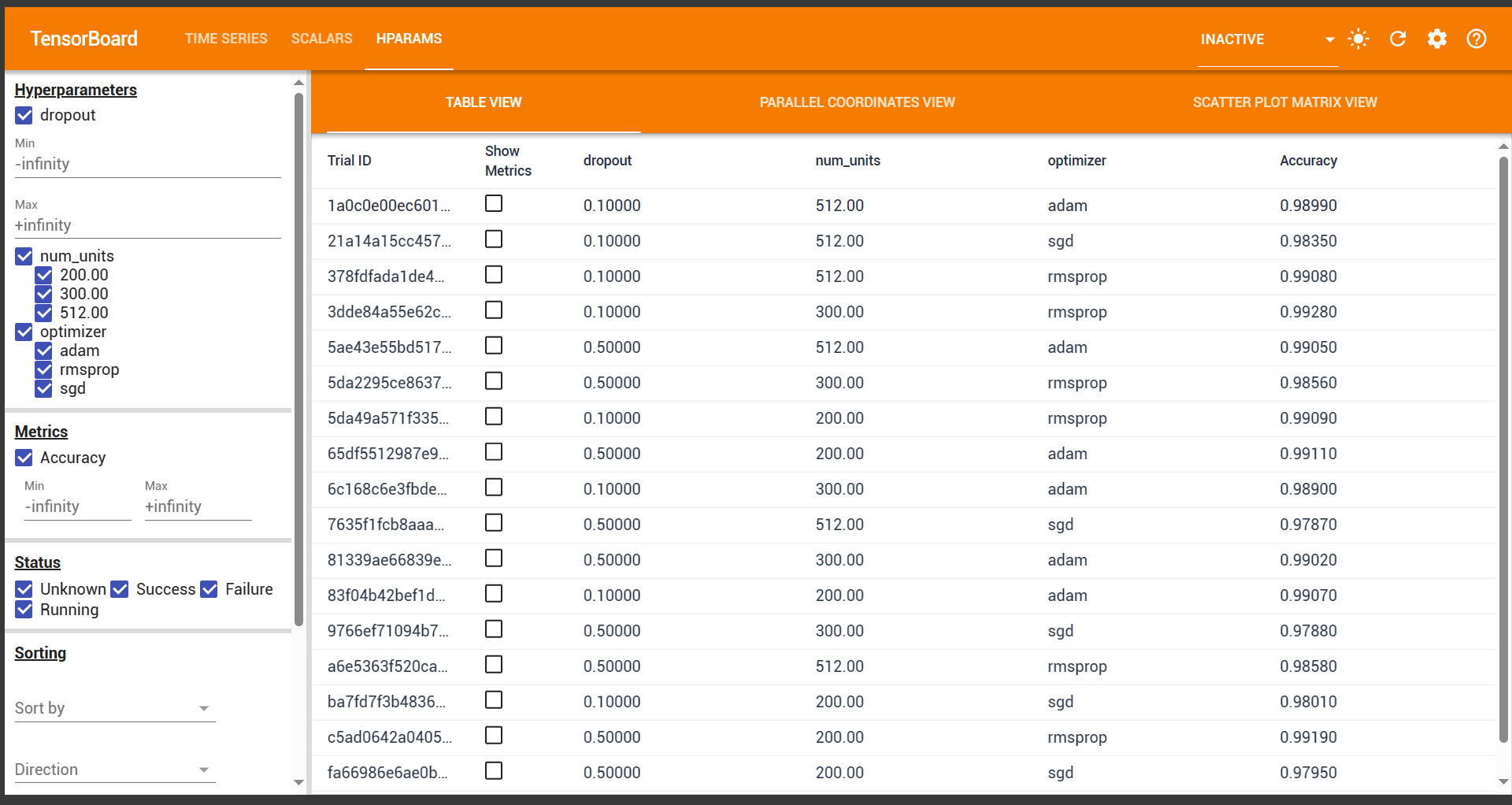


Рисунок 21 – Список экспериментов с гиперпараметрами

Исходя из данного списка можно получить очень полезную информацию, которая может подсказать какие параметры нужно добавить в модель для её лучшего обучения. Например, для данной нейронной сети оптимальными гиперпараметрами являются следующие:

Dropout: 0.1; Dense: 300; Optimizer: rmsprop;

При данных гиперпараметрах значение метрики (accuracy) достигает наибольшего максимума на тестовом наборе данных.

С помощью TensorBoard можно также увидеть координацию гиперпараметров в нейронной сети:

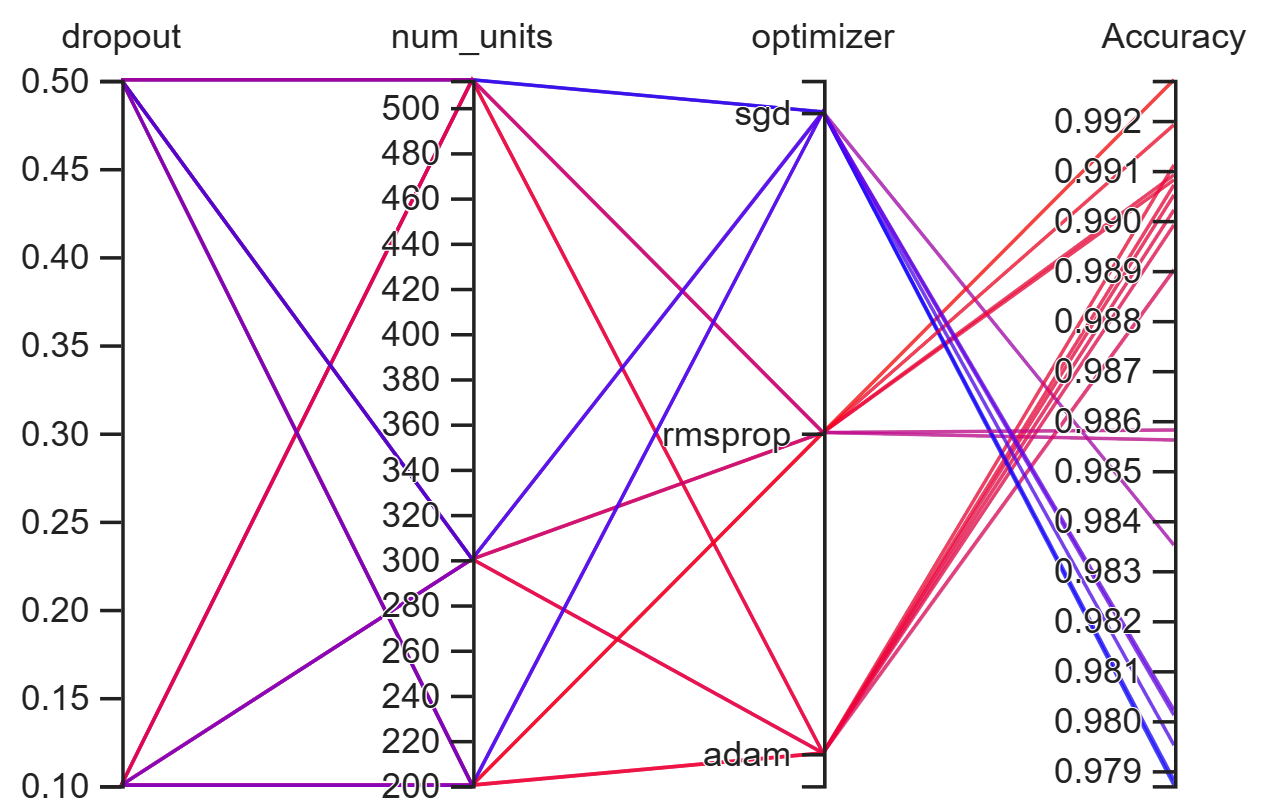


Рисунок 22 – Координация гиперпараметров в нейронной сети

Из рисунка 22 можно сразу заключить, что с оптимизатором rmsprop значение метрики достигает своего максимума, а с оптимизатором sgd – минимума. Оптимизатор sgd показывает результат даже лучше, чем оптимизатор adam.

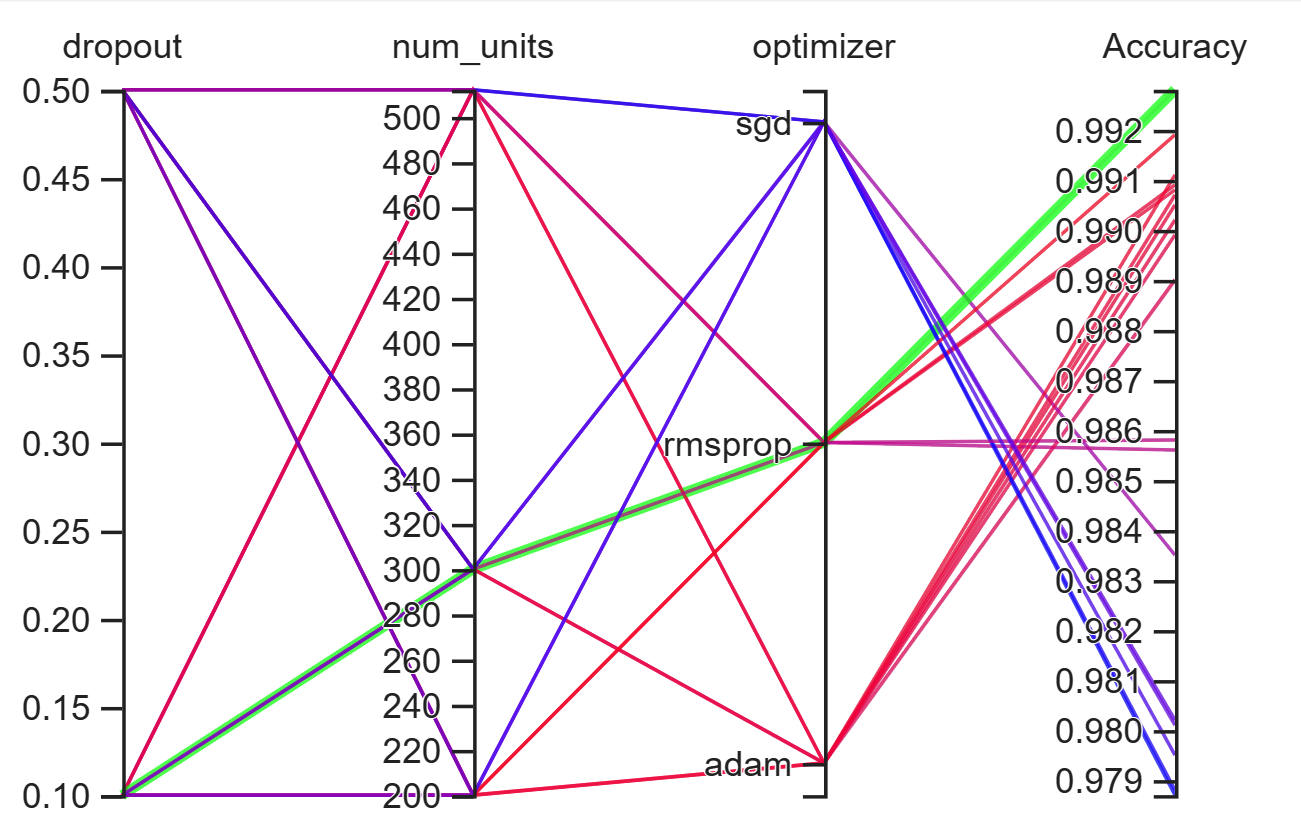


Рисунок 23 – Выделение оптимальных значений гиперпараметров

Таким образом, с помощью перебора гиперпараметров нейронной сети можно подобрать их таким образом, чтобы нейронная сеть лучше решала свою задачу.

Теперь сохраним результаты первой рассматриваемой нейронной сети задания №2 на сервисе wandb.

Для начала необходимо пройти процедуру логина на данной платформе:

!wandb login

Затем запустить задачу для отправки значений в сервис wandb:

wandb.init(

project="my-first-project",

config={

"learning\_rate": 0.02,

"architecture": "CNN",

"dataset": "CIFAR-100",

"epochs": 5,

}

)

И затем запустить цикл обучения модели нейронной сети следующим образом:

EPOCHS = 5

# Прохождение 5-ти эпох обучения

for epoch in range(EPOCHS):

# Выполнение шага обучения для набора данных

for (x\_train, y\_train) in train\_dataset:

train\_step(model, optimizer, x\_train, y\_train)

# Регистрация скалярных значений функции потерь и метрики в wandb (train)

# Выполнение шага тестирования для набора данных

for (x\_test, y\_test) in test\_dataset:

test\_step(model, x\_test, y\_test)

# Регистрация скалярных значений функции потерь и метрики в wandb (test)

wandb.log({"accuracy": test\_accuracy.result(), "loss": test\_loss.result()})

# Шаблон вывода информации в консоль

template = 'Epoch {}, Loss: {}, Accuracy: {}, Test Loss: {}, Test Accuracy: {}'

# Вывод информации в консоль о прохождении конкретной эпохи

print (template.format(epoch+1,

train\_loss.result(),

train\_accuracy.result()\*100,

test\_loss.result(),

test\_accuracy.result()\*100))

# Сброс метрик для каждой эпохи

train\_loss.reset\_states()

test\_loss.reset\_states()

train\_accuracy.reset\_states()

test\_accuracy.reset\_states()

# Завершение работы с wandb

wandb.finish()

В данном коде добавляется отправка значений loss и accuracy (на тестовых данных) модели на сервис wandb в проект my-first-project.

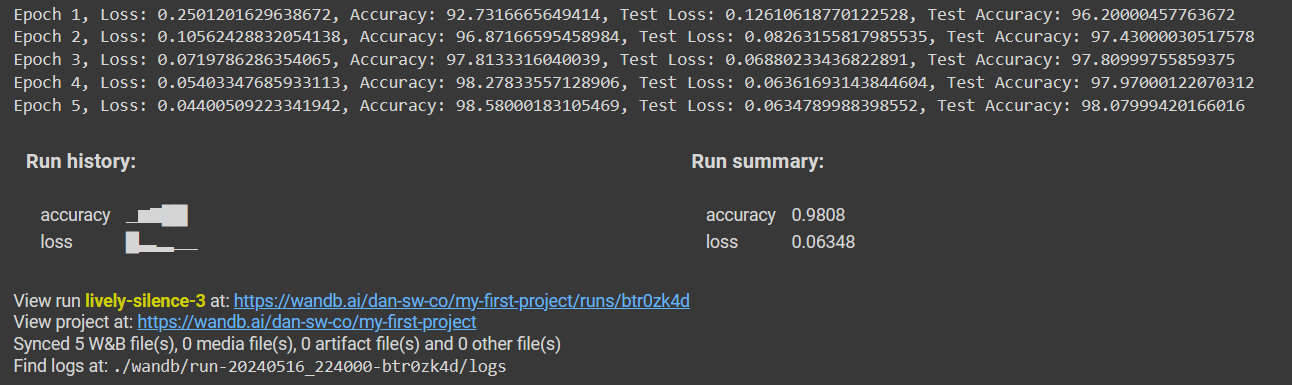


Рисунок 24 – Результат обучения модели нейронной сети

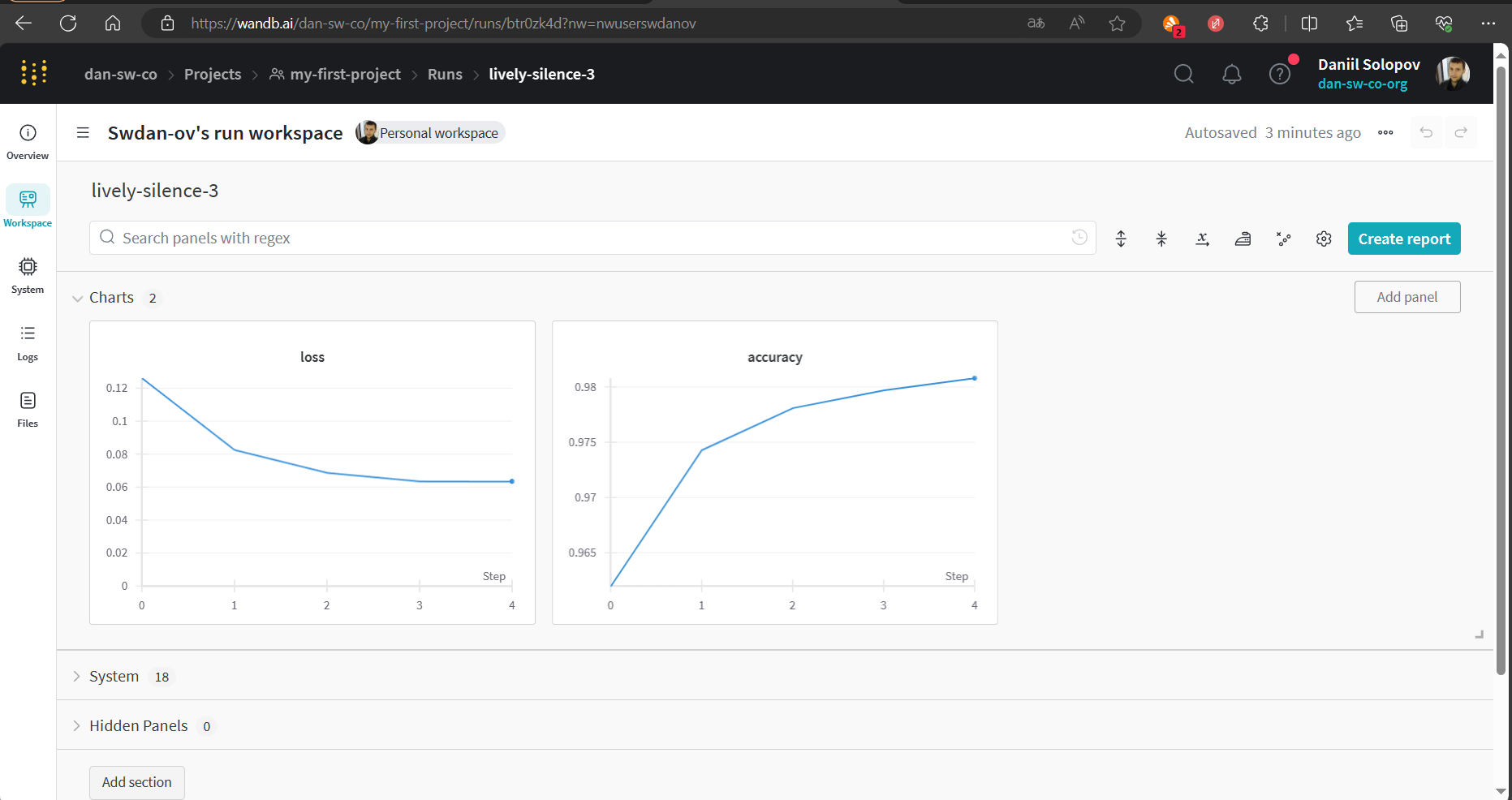


Рисунок 25 – Результат загрузки данных на сервис wandb

Заключение

В рамках лабораторной работы были выполнены все поставленные задачи.

При решении задания №1 был рассмотрен пример классической нейронной сети для анализа результатов обучения через TensorBoard и продемонстрирована невозможность отправки результатов анализа в TensorBoard.dev.

При решении задания №2 была реализована стандартная функция обратного вызова для сохранения статистики с явным указанием параметров, использовано TensorFlow Summary API для сохранения пользовательской статистики и изучена панель инструментов Hparams для анализа гиперпараметров на примере нейронной сети из второй лабораторной работы. Также был использован Summary Trace API для изучения параметров вычислительного графа на примере сложной функции из первой лабораторной работы. Был составлен её вычислительный граф и произведено соответствие с вычислительным графом, который был получен в первой лабораторной работе. Помимо этого, была осуществлена регистрация на сервисе wandb и при обучении одного из тестовых примеров нейронных сетей значения метрики и функции потерь записаны на данный сервис удалённо.

Таким образом все задания данной лабораторной работы успешно выполнены.

Приложения

1. Ссылка на исходный код первой лабораторной работы: <https://github.com/DanSoW/INRTU/blob/main/software-tools-for-artificial-intelligence-tasks/lab1/lab1.ipynb>.
2. Ссылка на исходный код второй лабораторной работы: <https://github.com/DanSoW/INRTU/blob/main/software-tools-for-artificial-intelligence-tasks/lab2/software_lab2.ipynb>.
3. Ссылка на исходный код третьей лабораторной работы: <https://github.com/DanSoW/INRTU/blob/main/software-tools-for-artificial-intelligence-tasks/lab3/lab3.ipynb>.
4. Ссылка на проект wandb: <https://wandb.ai/dan-sw-co/my-first-project/runs/btr0zk4d?nw=nwuserswdanov>.