Внедрение моделей в продакшн

# Содержание

[Содержание](#_fxt8pjoibt68)

[Интро](#_c4s8m6unxl97)

[Урок 1: Деплоймент нейросетевых моделей](#_9mnfdp2vbhxo)

[Урок 2: [ПРАКТИКА] Сохранение и загрузка моделей TensorFlow](#_r4oxsfa68d61)

[Урок 3: [ПРАКТИКА] TensorFlow Serving](#_4iltrpgen326)

[Домашнее задание](#_yc4krgrft1es)

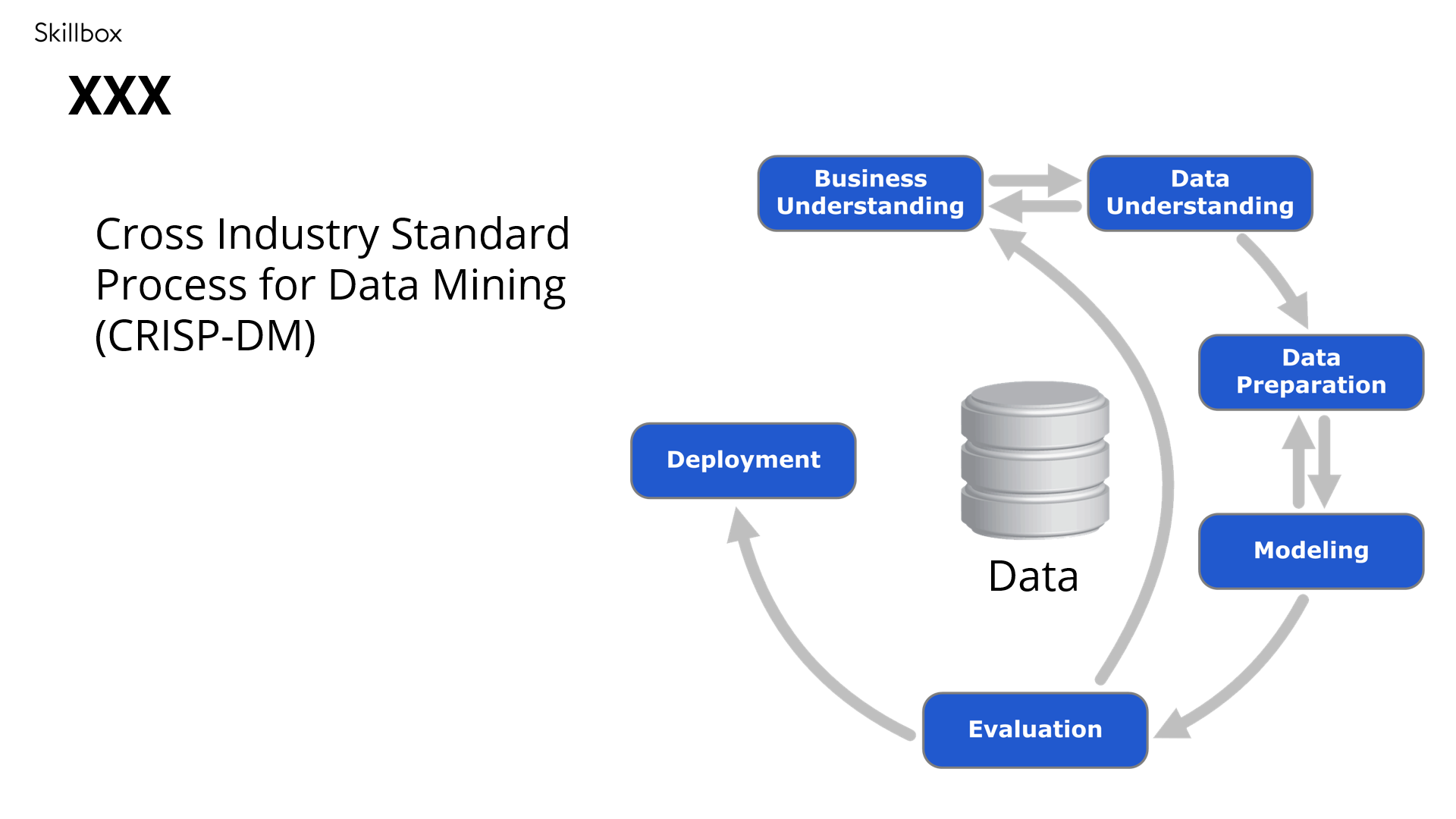
[Заключение](#_26sdwrg8pjx9)

## Интро

В предыдущих модулях мы рассматривали разнообразные типы нейронных сетей и технологии их обучения. Однако, на практике мы не просто так обучаем модели, чтобы поиграть с ними и забыть о них. Как правило, после этапа **обучения** нейросетевой модели (экспериментирование, подбор гиперпараметров и.т.д) наступает этап “боевого” использования модели в какой-то прикладной задаче. Этот этап называется **внедрение модели в продакшн** или **Deployment модели**. В этом модуле мы познакомимся с некоторыми опциями того, как можно осуществлять эту процедуру.

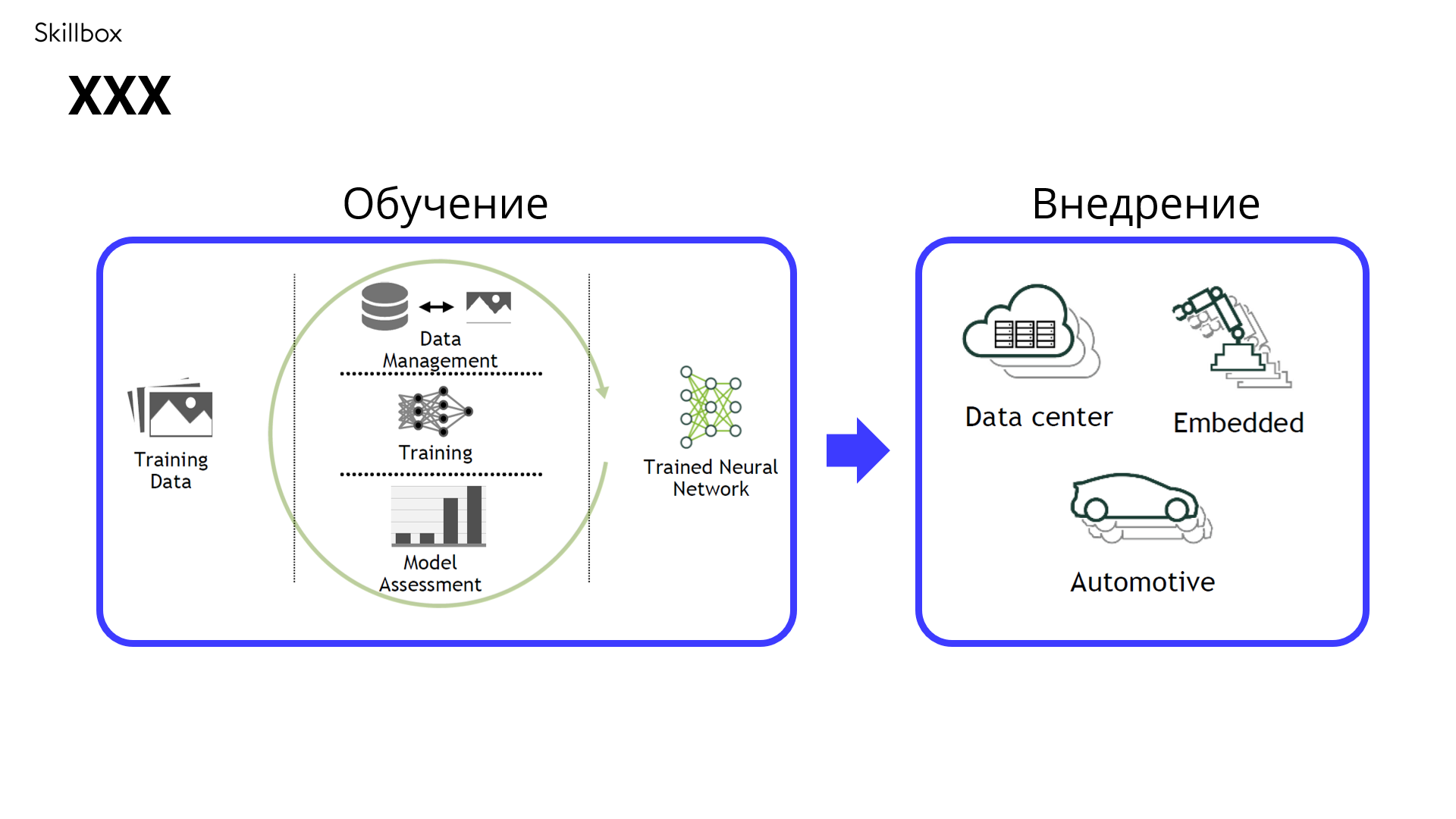
## Урок 1: Деплоймент нейросетевых моделей

В машинном обучении (и, в частности, в глубоком обучении) выделяют несколько этапов жизненного цикла проекта. Один из способов формализовать такие этапы — **Cross-Industry Standard Process for Data Mining (Межотраслевой стандартный процесс для исследования данных)**. Несмотря на сложное название, у такого процесса вполне понятные шаги и их взаимосвязь: понимание вашей бизнес задачи и данных, с которыми вы работаете, подготовка данных для обучения, обучение модели (Modeling), оценка качества обученной модели (Evaluation) и, наконец, внедрение обученной модели (Deployment).



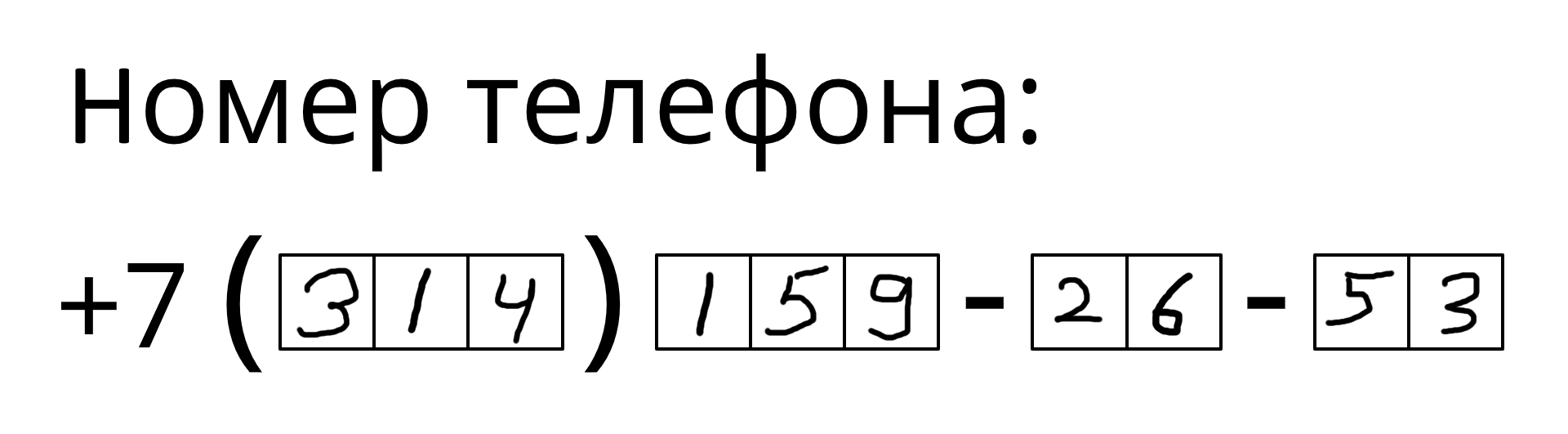
В принципе, в реальности можно возвращаться практически на любой этап такого процесса (уточнить бизнес задачу, переобучить модель, внедрить новую версию модели, и так далее).

Поэтому, иногда можно упростить и всё, что не касается внедрения (постановка задачи, работа с данными, обучение, оценка) обозначить одним большим этапом (внутри которого процесс может быть устроен довольно сложно. Назовём это всё просто “**Обучение**”. Это включает в себя и эксперименты и подбор гиперпараметров и подготовка всего, что нужно для обучения, включая данные. И как только мы обучили модель, мы хотим использовать её в в рамках какого-то другого (более крупного проекта), то есть осуществить **“Внедрение”**. Использование модели в “боевом” режиме после внедрения может не иметь ничего общего с тем, как модель была обучена. Например, мы могли обучать модель в облаке с использованием Google Collab или чего-то еще. А запускать модель для “боевого” инференса можем на каком-то другом устройстве с совершенно другим кодом (это может быть и компьютер, и облако, и мобильное устройство, и даже автономный автомобиль).

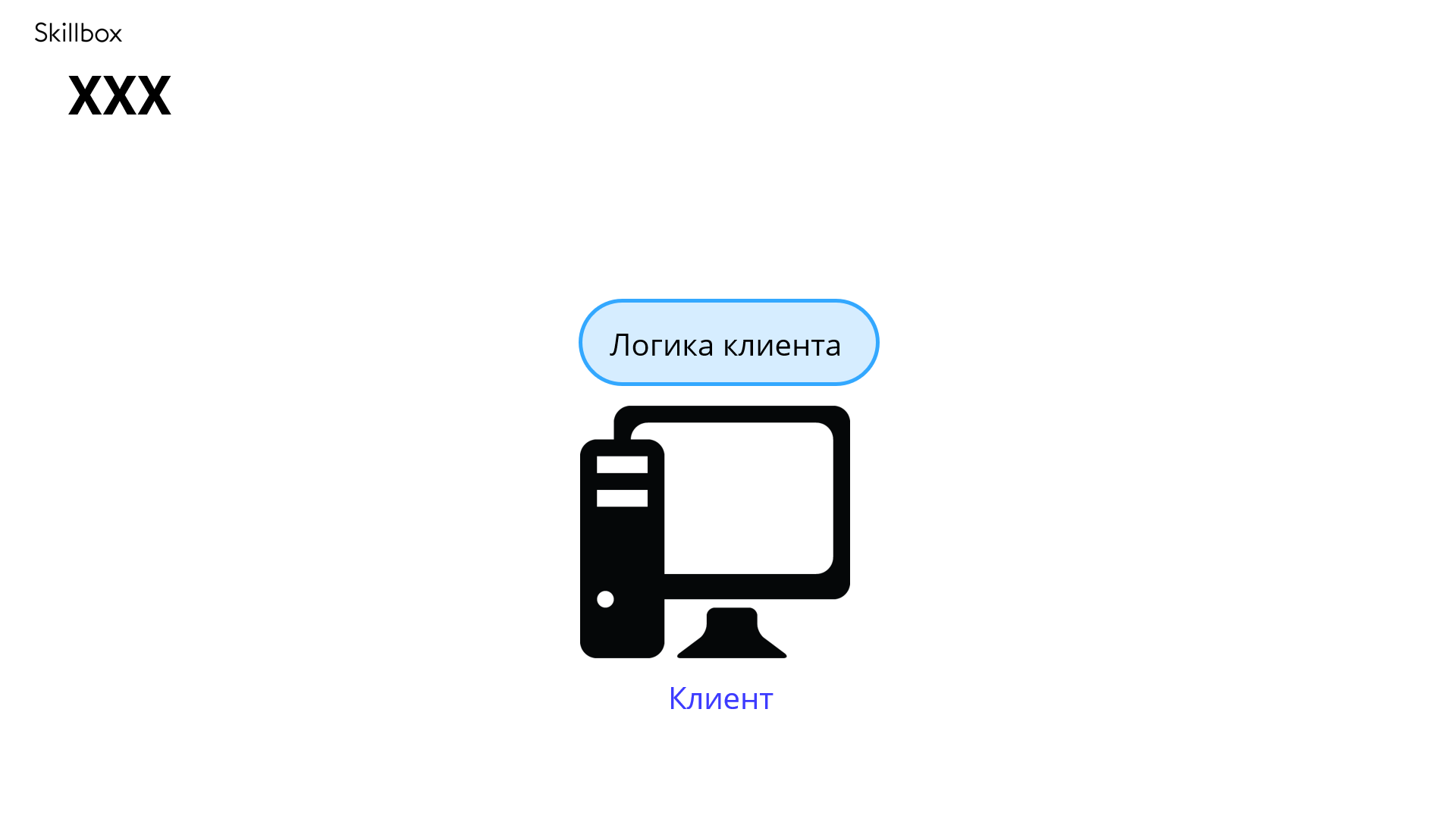


Как правило, наша финальная цель состоит в том, чтобы создать некое приложение (в широком смысле), которое решает какую-то бизнес задачу. Рассмотрим простой пример.

Пусть мы хотим создать приложение (или сервис), которое по скану или фотографии заполненной от руки формы распознает, что туда вписал человек. Для простоты остановимся на задаче распознавания номера телефона по написанным от руки цифрам:



Будем считать, что такое приложение или сервис работает на некотором устройстве (это может быть компьютер, телефон, веб-сайт, облако и.т.д). И так как там крутится наше целевое приложение, назовём это устройство **“клиент”**, а программное обеспечение, реализующее наше приложение — “**логика клиента**”.



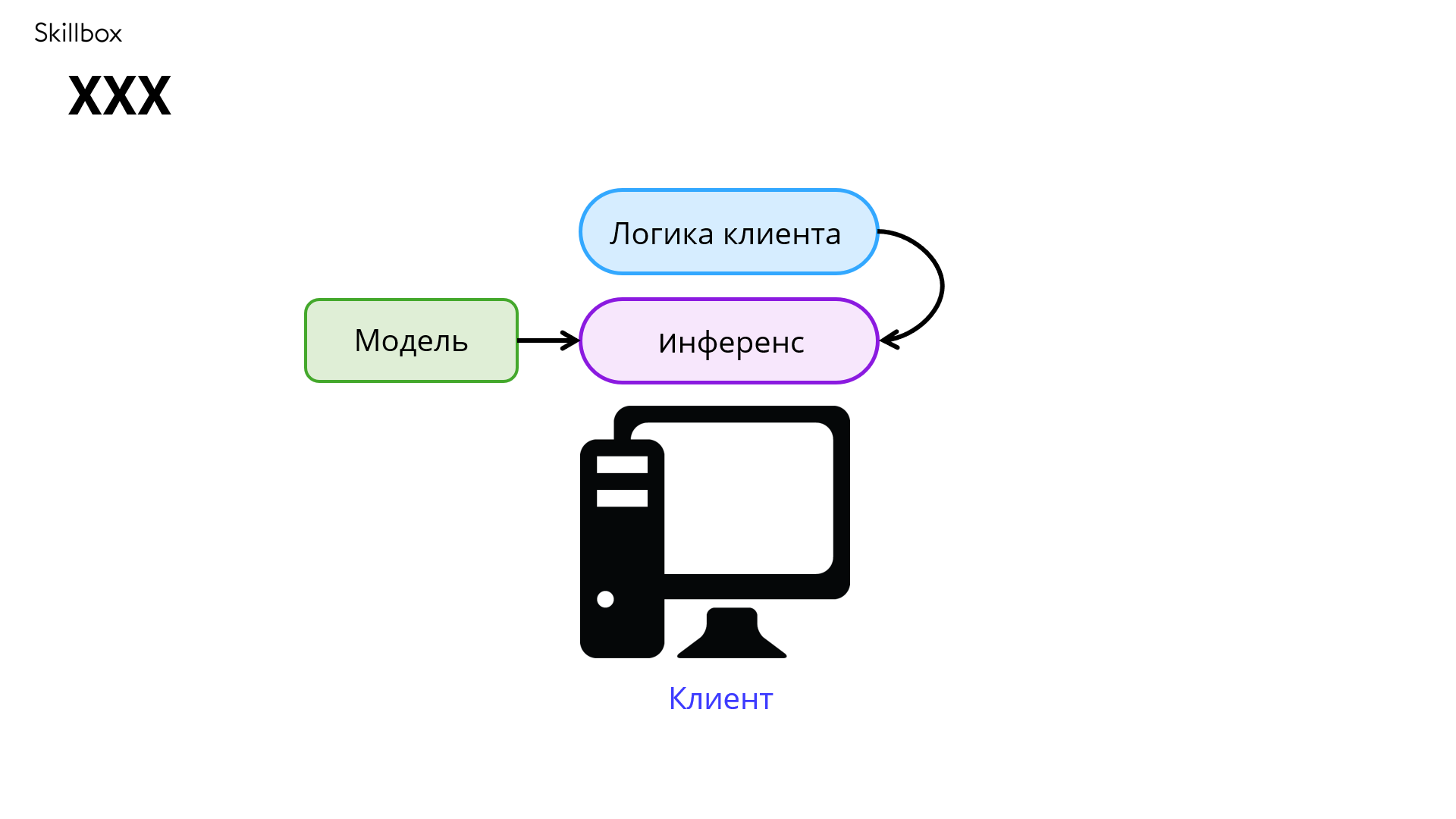
Часть нашей клиентской логики основана на использовании нейронной сети в боевом режиме, то есть в режиме инференса. В нашем случае мы будем использовать нейронную сеть только для **классификации** цифр. А вся остальная часть клиентской логики может содержать ввод/вывод данных, взаимодействие с пользователем, подготовку данных для инференса и многое другое.

Как правило, создание клиентского приложения — отдельная большая задача, которая выходит за рамки обязанностей Machine Learning инженера. Однако, ML инженер зачастую должен участвовать в налаживании взаимодействия между клиентской логикой и нейросетевым инференсом.

В случае с распознаванием телефонного номера пусть клиентская логика будет ответственна за первичную обработку входного изображения, вырезания соответствующих клеточек с цифрами и подачу их на вход в нейросеть для классификации.

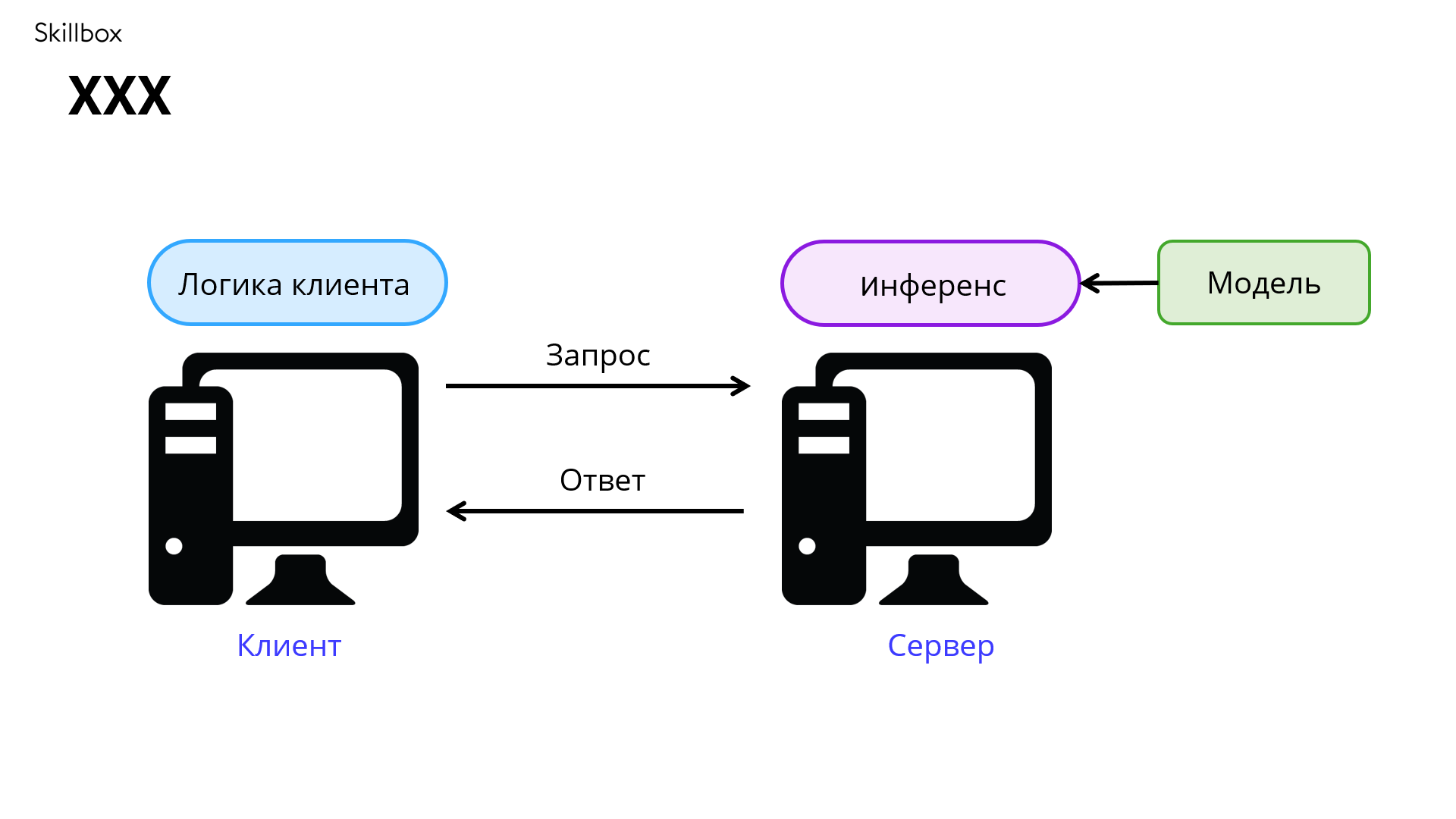
Как можно организовать взаимодействие клиентской логики и нейросетевого инференса?

Одна из возможных ситуаций — инференс осуществляется на том же устройстве, где работает клиентская логика (и, возможно, прямо в клиентской программе). Например, это может быть смартфон, в котором есть приложение, которое внутри себя использует инференс нейронной сети. Как правило, за инференс должен отвечать некоторый инференс движок (**inference engine**). В качестве такого движка можно использовать, например, тот же фреймворк, что мы использовали для обучения этой модели (например, TensorFlow). Но это не обязательно — инференс движок может быть не связан с фреймворком, использовавшимся для обучения. Кроме движка, нужна сама обученная модель (вычислительный граф + обученные веса). Если инференс движок отличается от фреймворка, который использовался для обучения модели, то может потребоваться дополнительная конвертация модели.



Однако, в более крупных проектах архитектура вашего приложения может быть более сложной. Один из классических примеров — инференс нейросетевых моделей вынесен на отдельный сервер (отдельная машина или сервер в облаке). Такая вещь иногда называется инференс сервер **(inference server)**. В таком случае ваше приложение или программа должны посылать **запрос (request)** на инференс сервер с задачей произвести инференс на данном входном изображении (или других данных) и получать **ответ (response)** от сервера с результатом работы нейросети.

То, что мы описали называется **клиент-серверная архитектура** (клиент — это наша программа или приложение, которое посылает данные и получает ответ, а сервер — это инференс сервер).



В такой архитектуре клиент “пользуется услугами” сервера и может ничего не знать про нейронные сети, как делать инференс, и т.д. Ему даже не нужно соответствующее железо: инференс может производиться, например, в облаке на GPU кластере, а клиентом может являться браузер на низкопроизводительном компьютере или мобильный телефон. Инференс движок и модель в таком случае хранятся и выполняются на инференс сервере. Инференс сервер может обслуживать несколько клиентов, обрабатывая их запросы.

В следующих практических уроках мы рассмотрим, как экспортировать обученную модель из TensorFlow, использовать её в некотором клиентском приложении и как реализовать клиент-серверную архитектуру с использованием **TensorFlow Serving**.

## Урок 2: [ПРАКТИКА] Сохранение и загрузка моделей TensorFlow

**Notebook:** 1\_Saved\_model\_1.ipynb

**Notebook:** 1\_Saved\_model\_2.ipynb

## Урок 3: [ПРАКТИКА] TensorFlow Serving

В предыдущем уроке мы рассмотрели сохранение и загрузку обученной модели для последующего инференса. В нашем простом примере для инференса мы использовали ту же самую машину, в которой нам необходимо было использовать полученный результат.

Однако, зачастую, в более крупных проектах архитектура вашего приложения может быть более сложной. Один из классических примеров — инференс нейросетевых моделей вынесен на отдельный сервер (отдельная машина или сервер в облаке). Такая вещь иногда называется **Инференс Сервер**. В таком случае ваше приложение или программа должны посылать **запрос (request)** на инференс сервер с задачей произвести инференс на данном входном изображении (или других данных) и получать **ответ (response)** от сервера с результатом работы нейросети.

То, что мы описали называется клиент-серверная архитектура. (клиент — это наша программа или приложение, которое посылает данные и получает ответ, а сервер — это инференс сервер). Такую архитектуру можно реализовать и самостоятельно. Но можно также воспользоваться готовым решением, которое поставляется вместе с TensorFlow. Такая серверная система называется **TensorFlow Serving**. Рассмотрим пример создания и использования инференс сервера с помощью TensorFlow Serving.

В реальности нам нужно как минимум две машины (одна сервер, одна клиент), но для демонстрации поместим и серверную и клиентскую логику на одну и ту же машину (все инструменты будут работать так же).

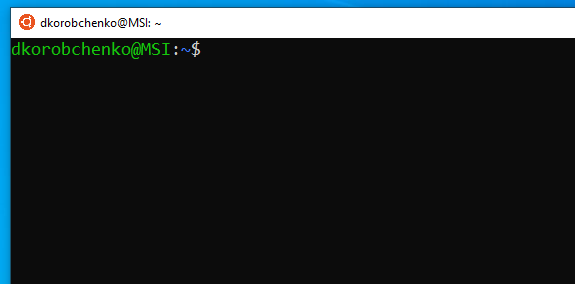
Такой эксперимент мы уже не сможем сделать в Google Colab, для этого нам понадобится реальная машина. Проще всего это будет сделать, если у вас уже есть операционная система Linux (лучше всего Ubuntu). Другой случай, если у вас Windows или Mac OS, можно использовать виртуальную машину с установленной Ubuntu. Кроме того, для Mac OS можно использовать docker (но это для более продвинутых пользователей). В данном примере мы рассмотрим еще один вариант: с использованием Ubuntu внутри Windows 10.

С недавних времен в Windows 10 реализована Linux подсистема (Windows Subsystem for Linux), которая во многом является полноценным линуксом и пригодна для нашего примера. Для её использования необходимо включить соответствующий флаг в Windows и установить приложение Ubuntu (подробнее по ссылкам).

<https://docs.microsoft.com/en-us/windows/wsl/install-win10>

<https://tutorials.ubuntu.com/tutorial/tutorial-ubuntu-on-windows#0>

После установки и запуска приложения Ubuntu мы увидим классический bash терминал, в котором можно выполнять unix команды.



Сразу после установки, у нас скорее всего не будет нужных нам пакетов и библиотек, их нужно будет установить вручную. Необходимый список того, что нужно установить, может варьироваться в зависимости от времени. Для установки TensorFlow необходимо сделать следующее:

sudo apt-get update

sudo apt install python3-pip

pip3 install tensorflow-gpu==2.0

После установки TensorFlow необходимо установить **TensorFlow Serving**

sudo echo "deb http://storage.googleapis.com/tensorflow-serving-apt stable tensorflow-model-server tensorflow-model-server-universal" | sudo tee /etc/apt/sources.list.d/tensorflow-serving.list

sudo curl https://storage.googleapis.com/tensorflow-serving-apt/tensorflow-serving.release.pub.gpg | sudo apt-key add -

sudo apt update

sudo apt-get install tensorflow-model-server

Теперь у нас есть всё, что нужно для проведения демонстрации.

Давайте реализуем клиент-серверное приложение для задачи распознавания номера телефона заполненной от руки форме. Клиентское приложение будет являться Python скриптом, получающим на вход выровненный скан документа и печатать распознанный номер телефона. Инференс (классификация изображений с цифрами) будет осуществляться на инференс сервере.

Для демо мы будем запускать и сервер и клиент на одном и том же компьютере, но так как для взаимодействия между ними будет использоваться протокол HTTP, всё будет выглядеть так же, если клиент и сервер расположены на разных машинах (например, сервер в облаке, а клиент на мобильном телефоне).

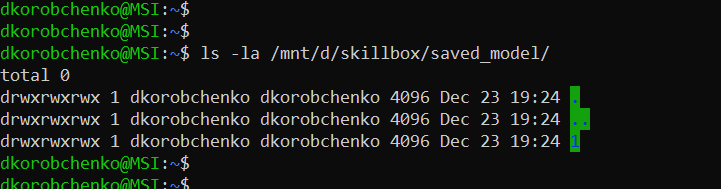
Будем использовать обученную ранее модель для классификации цифр. Разместим нашу сохраненную модель в следующей директории:

D:/skillbox/saved\_model/

Чтобы получить доступ к этой директории из нашей Ubuntu подсистемы, надо трансформировать путь следующим образом:

/mnt/d/skillbox/saved\_model/

Посмотрим, что находится в этой директории. Видим подпапку “1”, там лежит наша обученная модель, версия 1.

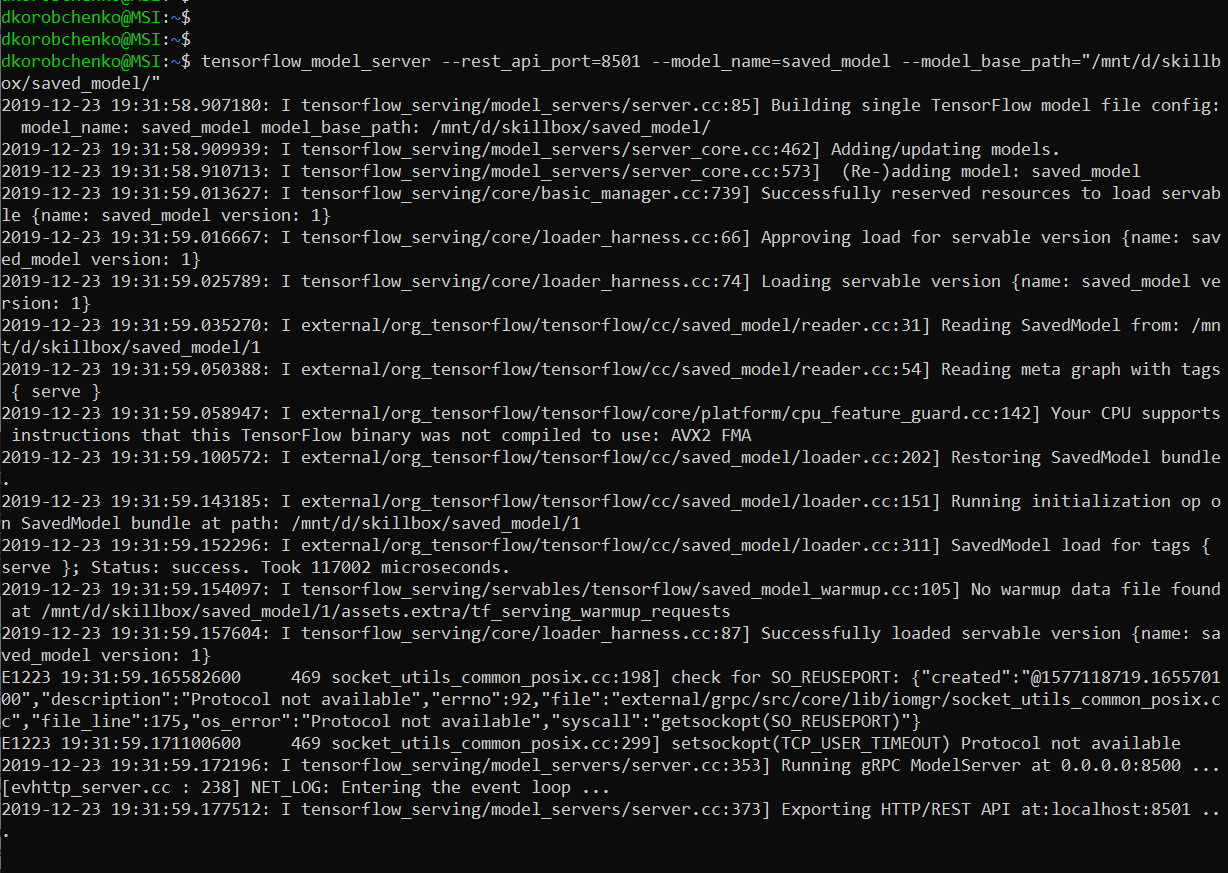


Теперь, зная этот путь, мы можем запустить **TensorFlow Model Server**:

tensorflow\_model\_server --rest\_api\_port=8501 --model\_name=saved\_model --model\_base\_path="/mnt/d/skillbox/saved\_model/"

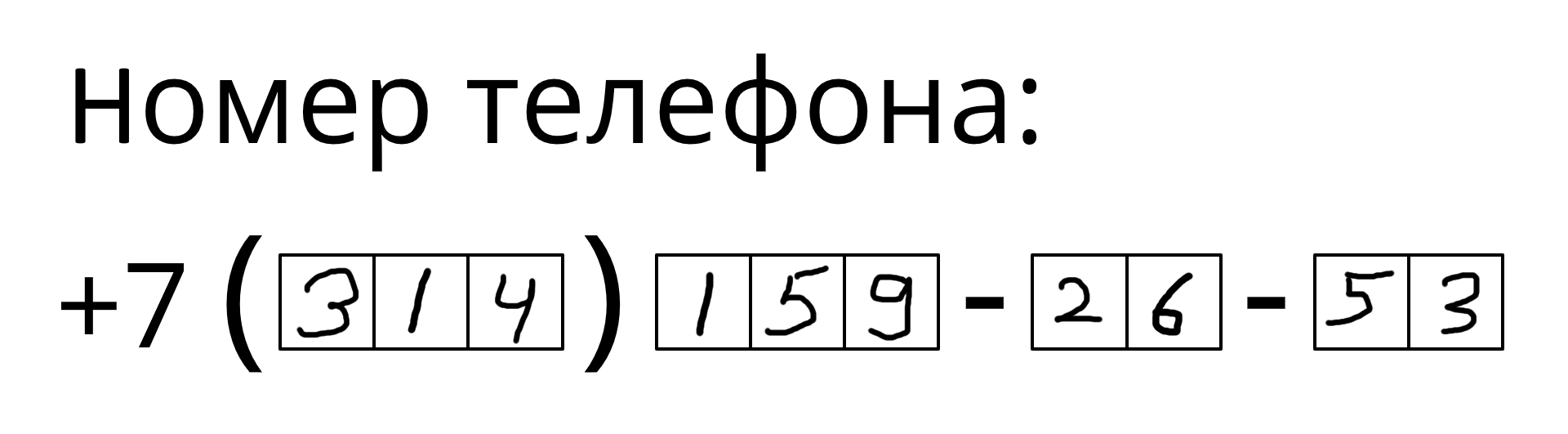
Какие аргументы у этой команды? “**rest\_api\_port**” — порт, по которому необходимо будет делать HTTP запрос к серверу. “**model\_name**” — название нашей модели, нужно если на сервере у нас крутятся несколько моделей, “**model\_base\_path**” — путь к модели, а точнее к директории со всеми версиями модели (TensorFlow Server возьмёт последнюю версию).

Выполняем команду и видим, что наш сервер запустился. После такого запуска мы должны держать это окно открытым, чтобы сервер не закрылся. Если нужно запустить сервер в фоновом режиме, можно использовать утилиты “nohup” или “screen”.



Теперь давайте создадим **клиентское приложение**. Пусть это будет просто Python скрипт, который открывает изображения с диска, подготавливает данные для нейросети, отсылает их на сервер для инференса, получает ответ и печатает его.

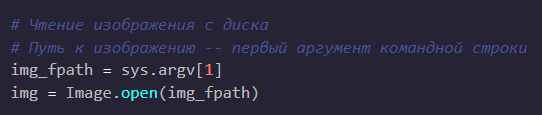
Создадим изображение phone.png (в Paint).



**Файл со скриптом**: recognize.py

Рассмотрим этот файл.

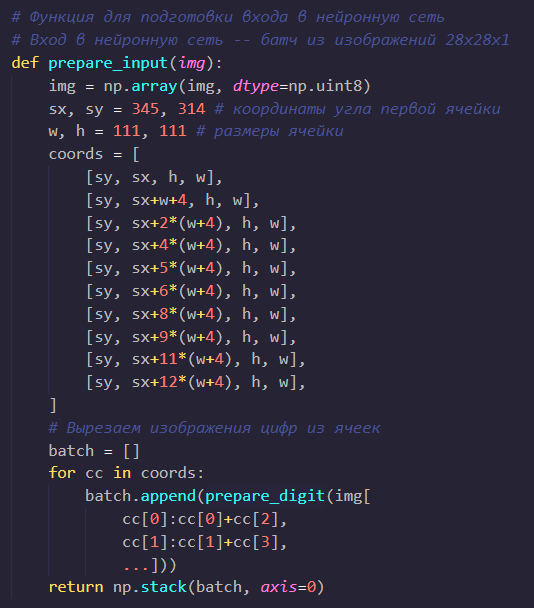
Получим путь к изображению с выровненным сканом из первого аргумента командной строки и откроем изображение.



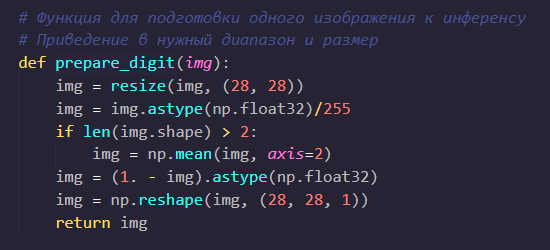
Затем необходимо подготовить данные для инференса, сделаем это с помощью функции **prepare\_input**



В этой функции мы будем вырезать из входного изображения скана ячейки, содержащие цифры и собирать их в батч для последующего инференса. Для простоты предположим, что скан выровнен, так что мы знаем координаты всех ячеек (они захардкожены в конкретных числах). Для каждой вырезанной ячейки применяем функцию **prepare\_digit** и складываем подготовленную картинку в батч. Всего в батче будет 10 элементов (10 цифр в телефоне).



Функция **prepare\_digit** будет делать следующее: приводить картинку к размеру (28x28), так как именно такой размер ожидает наша нейронная сеть, преобразовывать картинку в numpy формат (np.array), приводить к типу float32 и диапазону [0, 1], избавляться от цветовой составляющей, если пришло RGB изображение (в нейронную сеть должен прийти один канал) путём усреднения каналов (np.mean), инвертированию цветов (у нас черная цифра на белом фоне, а в MNIST наоборот), и приведение к трёхмерному тензору размером (28, 28, 1) — (высота, ширина, каналы).

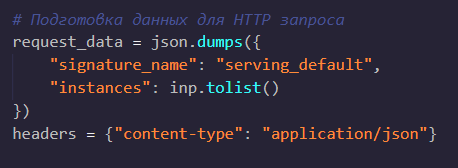


Для отправки батча мы будем использовать HTTP протокол (чтобы можно было отправлять по сети), поэтому нам необходимо подготовить данные в специальном виде. В качестве тела запроса будет некоторой JSON объект со следующими полями: “signature\_name” и “instances”.

Signature\_name — это название сигнатуры, которую мы хотим использовать. Ранее, когда мы экспортировали модель, мы никак не указывали “что означает сделать инференс”, то есть “где у модели вход, где у нее выход”. По сути это и есть сигнатура. Наша модель может быть произвольным графом, а сигнатура -- это указание входного узла и выходного узла. В нашей простой Sequential модели (которую мы обучали на MNIST) это и так понятно (где вход, где выход) и TensorFlow при экспорте сам это понял и создал для нас “дефолтную” сигнатуру “serving\_default”. В случае, если бы у нас был более сложны граф, с несколькими входами и выходами, то необходимо было бы создать сигнатуру в ручную. У одной модели может быть и несколько сигнатур.

А “instances” это наши входные данные. Только их нужно преобразовать в список, чтобы можно было предать наши цифровые данные в JSON формате.

Кроме того, нужно создать заголовок HTTP запроса “headers” (это стандартная вещь из протокола HTTP).



Теперь мы можем просто сделать HTTP запрос по соответствующему URL. Наш сервер доступен по адресу:

<http://localhost:8501>

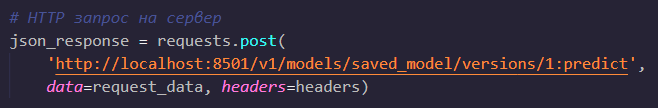
Если бы нужно было обратиться к этому серверу извне, вместо localhost нужно было бы указать IP этой машины.

URL, по которому необходимо посылать запросы по распознаванию выглядит следующим образом:

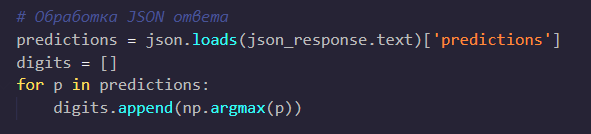
<http://localhost:8501/v1/models/saved_model/versions/1:predict>

“saved\_model” - это название нашей модели, “1:” -- номер версии 1, “predict” — название действия (сделать инференс).

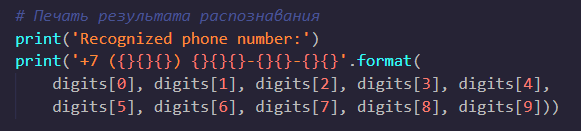
Заметьте, что здесь мы нигде не используем TensorFlow, то есть клиентское приложение вообще может ничего не знать о нейронных сетях и соответствующих фреймворках.



Результат мы получаем в виде JSON объекта, и если всё прошло хорошо (без ошибок), то в ответе будет содержаться поле “predictions”. Это список всех распределений по классам для каждой входной картинки. Пройдясь по этому списку и применив argmax, мы получим все предсказанные цифры.

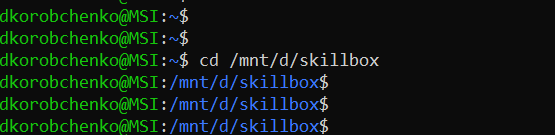


В итоге получим набор из 10 цифр — результат распознавания номера телефона. Печатаем их на экран.



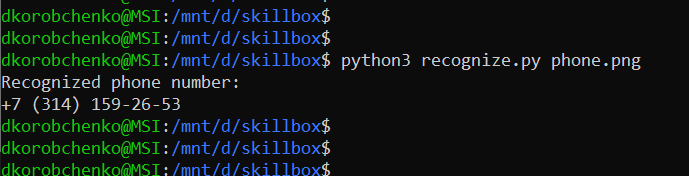
Теперь давайте запустим наше клиентское приложение. Перейдём в директорию с скриптом **recognize.py**. Там же лежит файл **phone.png**

cd /mnt/d/skillbox



Запустим скрипт и получим предсказание. В качестве аргумента передадим путь к файлу с изображением “phone.py”. Наш скрипт сделал запрос к серверу (который находится на нашей же машине, но в теории мог находиться где угодно) и получил от него ответ о предсказании.

python3 recognize.py phone.png



Мы рассмотрели довольно простое клиентское приложение. На самом деле это может быть программа любой сложности, начиная от простого скрипта, заканчивая мобильным приложением или web-страницей.

## Домашнее задание

Возьмите обученную в одном из прошлых модулей нейронную сеть (это может быть сеть для классификации изображений, детектирования объектов, сегментации, машинного перевода, чат бота, и.т.д) и оберните её в клиент-серверное приложение с помощью TensorFlow Serving. Экспортируйте модель, запустите сервер, реализуйте и протестируйте клиентское приложение.

## Заключение

В этом модуле мы познакомились с различными способами внедрения обученной нейросетевой модели в продакшн. Однако, мы, по сути, экспортировали нашу обученную модель “как есть”, без дополнительных оптимизаций. В следующем модуле мы рассмотрим то, как можно оптимизировать производительность модели при её деплойменте.