

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Московский институт электроники и математики им. А. Н. Тихонова

Чанке Иван Сергеевич, группа БИВ192

Разработка нейронной сети для распознавания
изображений.

Курсовая работа по направлению
09.03.01 «Информатика и вычислительная техника»

Студент:

Чанке И.С

Руководитель:

Романова И.И.

Москва 2020

Задание

на курсовую работу по дисциплине «Алгоритмизация и программирование»

студенту группы БИВ192 Чанке Ивану Сергеевичу

1. Тема работы

Разработка нейронной сети для распознавания изображений

2. Требования к работе

2.1 Общие требования

Нейронная сеть должна работать на одноплатном компьютере Raspberry Pi

2.2 Требования к процессу разработки и итоговому прототипу

К процессу разработки, а также к реализации итогового прототипа нейронной сети предъявляются следующие требования:

- Нейронная сеть должна быть написана на языке Python 3 с применением парадигмы объектно-ориентированного программирования без привлечения специализированных библиотек для машинного обучения.
- Нейронная сеть должна распознавать рукописные цифры на изображении разрешением 28x28 пикселей.
- Должен быть разработан графический пользовательский интерфейс (GUI) для демонстрации работы нейронной сети.
- Программа должна работать на ОС Windows.

3. Содержание работы

3.1 Анализ существующих способов машинной обработки изображений, в т.ч. аналоговых.

3.2 Разработка программы на языке Python 3, реализующей нейронную сеть.

3.3 Выделение оригинальных классов и методов, использующихся для реализации сети, в отдельную библиотеку.

3.4 Составление документации, включающей подробное описание структуры нейронной сети.

3.5 Демонстрация работы нейронной сети на примере данных базы MNIST.

4. Сроки выполнения этапов работы

Сроки выполнения этапов работы подробно прописаны в документе “Успеваемость за курсовую работу”

Дата защиты оформленного отчёта – 31.05.2020.

| | | |
|-----------------|--------------------|----------------------|
| Задание выдано | «14» ноября 2019г. | _____ И. И. Романова |
| Задание принято | «14» ноября 2019г. | _____ И. С. Чанке |

Оглавление

| | |
|---|----|
| Введение. Цели работы..... | 5 |
| 1. Обзор существующих аналогов. | 7 |
| 1.1 Обучение с учителем в общем случае. Задача классификации | 7 |
| 1.2 Деревья решений | 10 |
| 1.3 Модель случайного леса | 12 |
| 1.4 Метод k ближайших соседей..... | 13 |
| 2. Описание решения поставленной задачи. Линейные модели. | 14 |
| 2.1 Логистическая регрессия.. | 14 |
| 2.2 Нейронная сеть как совокупность простейших линейных моделей | 16 |
| 2.3 Обучение нейронной сети. Градиентный спуск. | 20 |
| 2.4 Подсчёт градиента функции ошибки методом обратного распространения. ... | 23 |
| 3. Решение поставленной задачи..... | 26 |
| 3.1 Структура нейронной сети | 26 |
| 3.2 Библиотека ptron | 27 |
| Выводы | 30 |
| Список источников | 30 |
| Приложение 1. Код библиотеки ptron | 31 |
| Приложение 2. Скриншоты работы приложения. Тестирование. | 37 |
| Приложение 3. Руководство пользователя | 40 |
| Приложение 4. Запуск программы на Raspbian..... | 44 |

Введение

Цель данной курсовой работы: Изучить принципы работы и применения линейных моделей для задач машинного обучения, научиться реализовывать и использовать такие модели, углубить знания соответствующих разделов математики – линейной алгебры и математического анализа функции многих переменных. Также в данной работе присутствует описание машинно обучаемых аналогов нейронных сетей, однако акцент делается в первую очередь именно на линейных моделях.

Продуктами работы являются:

- 1) Реализация математической модели полносвязной нейронной сети прямого распространения в виде библиотеки классов Python 3.
- 2) Написанное на Python 3 с использованием вышеуказанной библиотеки снабженное графическим интерфейсом приложение, позволяющее проверить работу модели на задачах обучения основным логическим функциям.
- 3) Аналогичное названному в пункте 2 приложение, представляющее простой и эффективный способ опробовать работу модели в задаче распознавания изображений рукописных цифр размером 28x28 пикселей.

Актуальность данной курсовой работы обоснована активным ростом сфер применения технологий машинного обучения в последнее время: сегодня машинное обучение в целом и линейные модели в частности широко используются в самых разных сферах науки, производства, бизнеса, медицины. Данные технологии

позволяют принимать решения на основе накопленных массивов данных, строить прогнозирующие модели. Одной из наиболее востребованных и активно развивающихся сфер машинного обучения является компьютерное зрение, открывающее большие возможности для автоматизации управления техникой, контроля процессов производства, диагностики неполадок и отслеживания нарушений. Именно линейные модели лежат в основе технологий компьютерного зрения.

Вышеперечисленные факторы делают машинное обучение практически самым перспективным направлением исследований в современной прикладной математике. Остро стоящая проблема развития технологий обучения с целью увеличения скорости и точности работы моделей требуют от современных инженеров-математиков не только умения строить и применять такие модели, но и глубокого понимания принципов их работы.

1. Обзор существующих аналогов.

Линейные модели не являются единственным семейством применяемых в машинном обучении моделей. Более того, некоторые задачи компьютерного зрения, такие, как распознавание небольших изображений, теоретически могут быть решены с помощью альтернативных обучаемых моделей. Рассмотрим существующие семейства моделей, применяемых для задач классификации, сравним их с нейронными сетями.

1.1 Обучение с учителем в общем случае. Задача классификации.

В основе большинства задач машинного обучения лежит принцип так называемого обучения с учителем. Формализовать его задачу можно так:

Пусть существует конечное множество объектов X , определенное на пространстве признаков размерности N (Каждый объект множества однозначно определяется значениями N признаков объекта). Найти функцию, однозначно отображающую каждый объект множества X на множество ответов Y .

Иными словами, имеется некий набор записей об объектах, состоящих из характеристик – предикторных переменных, описывающих каждый объект. Также имеется целевой признак объекта – тот, который необходимо научиться предсказывать по известному описанию объекта. Предполагается, что этот признак находится в зависимости от предикторов, то есть его возможно восстановить по заданному набору характеристик объекта. Задача обучения – понять характер этой зависимости, научиться предсказывать ответ по предикторам.

Объекты удобно представлять как векторы, составляющие матрицу признаков – как таблицы, строки которых являются записью об объектах, а столбцы – признаками этих объектов, включая целевой.

| Рост | Пол | Возраст | Вес |
|------|-----|---------|-----|
| 150 | М | 25 | 51 |
| 160 | М | 35 | 70 |
| 180 | Ж | 25 | 80 |
| 170 | М | 45 | 81 |
| 170 | Ж | 25 | 51 |
| 176 | Ж | 20 | 75 |
| 176 | М | 20 | 81 |
| 160 | Ж | 30 | 65 |

Рисунок 1. Таблица, содержащая записи об объектах – людях.

На рисунке выше в таблице представлены данные о росте, весе, поле и возрасте некоторых людей. Логично предположить, что, например, вес зависит от остальных признаков. Возможно обучить модель для предсказания целевого признака «вес» по предикторам «рост», «пол» и «возраст».

Суть обучения с учителем заключается в использовании некоторого количества размеченных данных – записей с известным целевым признаком, как в представленной выше таблице. На этих данных модель учится предсказывать целевой признак на неразмеченных данных, на которых модель не обучалась.

| Рост | Пол | Возраст | Вес |
|------|-----|---------|-----|
| 150 | Ж | 23 | ??? |
| 175 | Ж | 18 | ??? |
| 185 | М | 25 | ??? |

Рисунок 2. Неразмеченные данные.

Задача классификации – частный случай задачи обучения с учителем, в котором множество значений целевого признака является конечным. Так, в задаче

классификации рукописных цифр на черно-белых изображениях размером 28x28 пикселей предикторами являются 28x28 = 784 вещественных значения насыщенности цвета каждого пикселя, а целевым признаком – множество цифр от 0 до 9. Формально задача обучения выглядит так:

$$f: X \rightarrow Y$$

$$|X| = 784; \forall (x \in X) \in [0; 1]$$

$$Y = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$f - ?$$

Как было сказано выше, для решения задач классификации используются разные подходы – линейные модели, нейронные сети, а также следующие модели-аналоги – деревья решений (Decision Trees), случайный лес (Random Forest), метод k ближайших соседей (kNN – k-Nearest-Neighbors). Далее рассмотрим их преимущества и недостатки.

1.2 Деревья решений

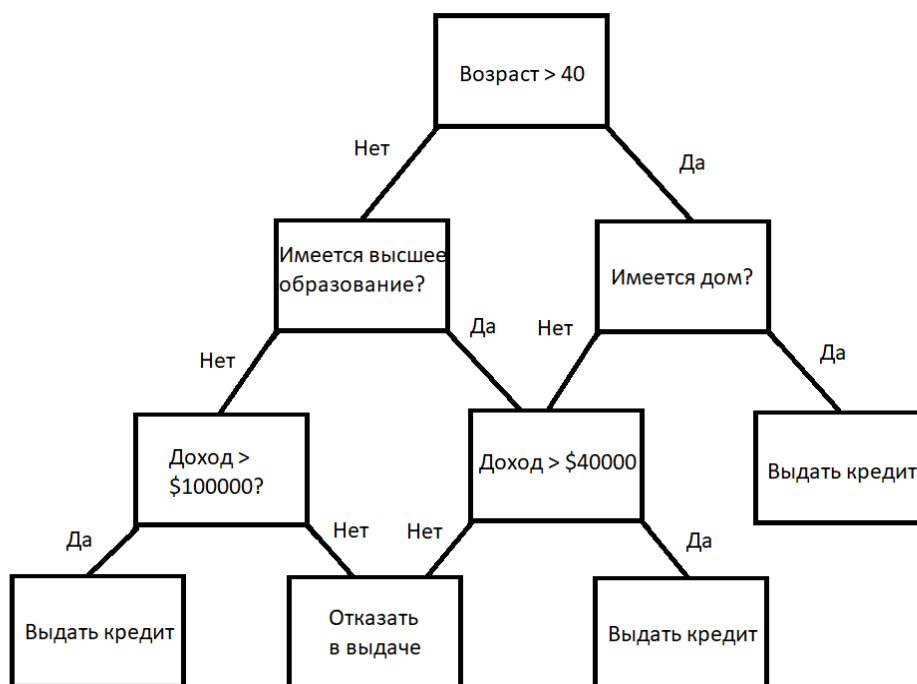


Рисунок 3. Пример дерева решений.

Дерево решения как алгоритм машинного обучения (задача классификации) представляет собой объединение логических правил вида

«(Значение признака $\alpha < x$) И (значение признака $\beta < y$) И (...) \rightarrow Класс n »

в структуре данных «Дерево».

Обучаясь на размеченных данных, алгоритм строит такое дерево по принципу «жадной максимизации прироста информации» либо руководствуясь критерием Джини, описание которого выходит за рамки данной курсовой работы.

Преимущества деревьев решений:

- Интерпретируемость. Построенное в результате работы алгоритма дерево можно просмотреть и увидеть, по каким критериям данные были разделены.
- Малая чувствительность к выбросам. Выброс – результат измерения, сильно выделяющийся из общей выборки. Так, в задаче предсказания веса по росту, весу и полу выбросом могло бы быть значение «250» параметра «Рост». Выбросы представляют большую проблему для линейных моделей, которая будет разобрана позднее в работе. Особенности алгоритма построения деревьев практически сводят на нет влияние выбросов на качество модели.
- Дерево решений как модель составляет основу других, намного более эффективных алгоритмов обучения.

Недостатки деревьев решений:

- Низкое качество модели в целом.
- Легко «переобучить» модель. Переобучение в машинном обучении – ситуация, когда алгоритм слишком сильно подстраивается под обучающие данные, что значительно (вплоть до нуля) снижает качество работы на новых данных, на которых модель не обучалась. Переобучение является проблемой для любого класса моделей, однако способы борьбы с ним для разных алгоритмов могут быть разными.

Теоретически, деревья решений могут использоваться в задачах компьютерного зрения, сведенных к классификации изображений, однако качество их работы будет неудовлетворительным. Более того, обучение на больших выборках, которые неизбежно требуются в любой задаче классификации изображений, займет долгое время.

1.3 Случайный лес

Модель случайного леса является композицией деревьев решений. При обучении модели на размеченных данных на случайных отобранных с повтором подвыборках этих данных независимо друг от друга обучаются множество деревьев решений, а в качестве ответа модели выступает усредненный ответ каждого из деревьев. Случайный лес наследует низкую чувствительность к выбросам деревьев решений, а также имеет уникальные преимущества – внутреннюю оценку качества модели непосредственно во время обучения (Out-of-bag оценка, подробное описание которой также выходит за рамки данной работы). Кроме того, являясь ансамблем простейших моделей, случайный лес работает весьма качественно.

Теоретически, случайный лес также можно использовать для некоторых задач компьютерного зрения, однако модель обладает существенными недостатками, которые сводят на нет преимущества алгоритма. Как уже было сказано выше, задачи компьютерного зрения требуют больших выборок с большим количеством признаков объектов. Так, в решаемой в рамках данной работы задаче распознавания изображений у каждого объекта имеется 784 признака. Однако качество основанных на деревьях моделей падает с увеличением числа признаков, а время обучения значительно растёт. Более того, использование большого числа признаков приводит к высоким затратам памяти – алгоритму необходимо хранить построенные деревья, сложность и размер которых зависит от количества признаков. Таким образом, использование основанных на деревьях моделей в задачах компьютерного зрения нецелесообразно.

1.4 Метод k ближайших соседей (kNN).

Метод работы алгоритма k ближайших соседей основательно отличается от древоподобных моделей, рассмотренных выше. Основой метода является гипотеза компактности, заключающаяся в предположении о том, что схожие объекты находятся близко друг к другу в N-мерном пространстве признаков, где каждый признак задает координату объекта по соответствующей оси. Критерием близости (метрикой расстояния между объектами) как правило является евклидово расстояние, расстояние городских кварталов или более сложные метрики (расстояние Чебышева, расстояние Минковского). В случае использования евклидова расстояния пространство признаков можно упрощенно представить в виде привычного нам геометрического пространства, а расстояние понимать как аналогию расстояния в бытовом смысле. Классификация методом k ближайших соседей сводится к вычислению расстояния от классифицируемого объекта до каждого из объектов обучающей выборки, а затем назначению целевому признаку объекта усредненного значения целевых признаков k ближайших соседей в обучающей выборке. Метод k ближайших соседей не используется в задачах классификации изображений по причине большого объема данных, на которых решающие задачу компьютерного зрения модели обучаются. Дело в том, что обучение kNN-алгоритма сводится к запоминанию всей обучающей выборки, что в случае с изображениями приведет к колоссальному расходу памяти.

Вывод

Алгоритмы, основанные на деревьях решений, а также на полном запоминании обучающей выборки не являются эффективными применительно к задачам компьютерного зрения. Таким образом, необходимо перейти к рассмотрению линейных моделей.

2. Описание решения поставленной задачи.

Линейные модели.

Задача: построить эффективную машинно обучаемую модель, решающую задачу классификации рукописных цифр на изображениях размером 28x28 пикселей.

До сих пор в работе рассматривались в основном древовидные обучаемые модели, однако наибольшее распространение в задачах компьютерного зрения получили линейные модели. Ниже будут рассмотрены модель простейшего линейного классификатора, на которой строятся более сложные, принцип работы этой модели. Затем будет описано устройство полносвязных нейронных сетей прямого распространения как совокупности простейших линейных моделей, обоснована необходимость использования такой структуры, приведен алгоритм её обучения.

2.1. Логистическая регрессия

Примером простейшего линейного классификатора является модель *логистической регрессии*, задача которой заключается в бинарной классификации – предсказании двоичной метки объекта на основании его заданных характеристик. Решение о присвоении объекту дискретной метки принимается в результате сравнения возвращаемой моделью непрерывной величины с некоторым порогом классификации – как правило, значение порога полагается равным 0,5. Объекты, для которых моделью возвращаются величины больше или же меньше установленного порога классифицируются как принадлежащие к противоположным классам – положительные и отрицательные соответственно. Возвращаемой непрерывной величиной является *степень уверенности* модели в том, что рассматриваемый объект

следует классифицировать как положительный, эта величина принимает значения от 0 до 1 включительно.

Работу алгоритма логистической регрессии можно описать геометрически. Ранее мною уже была упомянута возможность представить множество объектов как линейное пространство размерности N , где каждый объект является точкой, координатами которой выступают признаки этого объекта. Задача обучения логистической регрессии сводится к построению в данном пространстве гиперплоскости, однозначно отделяющей объекты противоположных классов друг от друга. В ходе обучения находится уравнение этой гиперплоскости, принимающее следующий вид:

$$\beta_0 + \beta_1 x_1 + \dots + \beta_q x_q = 0, \quad q = N$$

Где β – коэффициенты, с которыми в уравнение входят x – признаки объекта.

Координаты каждого нового классифицируемого объекта подставляются в полученное уравнение, которое принимает значение, пропорциональное евклидову расстоянию от объекта до плоскости. Знак полученного значения играет ключевую роль в классификации, так как объекты, для которых уравнение принимает разные по знаку значения, находятся по разные стороны разделяющей гиперплоскости. Очевидно, что полученные величины могут принимать любые вещественные значения, однако же уверенность модели принадлежит отрезку $[0; 1]$. Для перехода от расстояния к уверенности используется *логистическое преобразование*: к полученному значению применяется сигмоидальная функция:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\lim_{x \rightarrow +\infty} \sigma(x) = 1, \quad \lim_{x \rightarrow -\infty} \sigma(x) = 0$$

Следовательно, уверенность модели выражается как $\frac{1}{1 + e^{-\beta_0 + \beta_1 x_1 + \dots + \beta_q x_q}}$.

2.2 Нейронная сеть как совокупность простейших линейных моделей.

Отделить объекты противоположных классов единственной гиперплоскостью не всегда представляется возможным. Ярким примером задачи, решение которой невозможно с помощью единственной логистической модели, является задача «исключающего ИЛИ-2». Исключающее ИЛИ-2 – логическая функция двух аргументов, принимающая значение «Ложь» в случае равенства аргументов, «Истина» – в остальных случаях. Пусть имеется 4 объекта с бинарными признаками 0 и 1. Изобразим их геометрически в двумерном пространстве признаков:

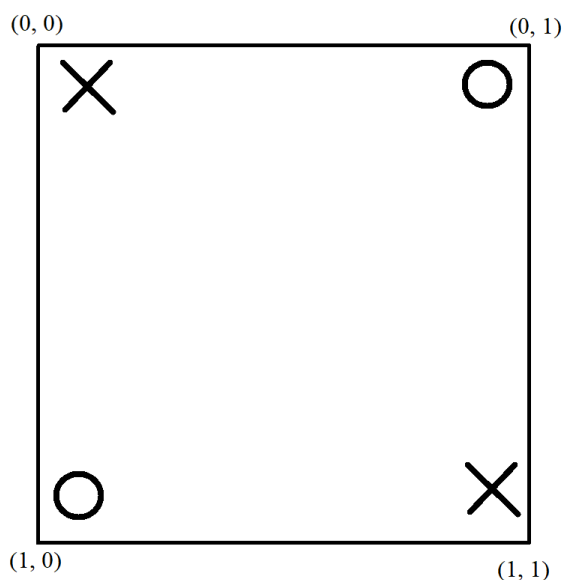


Рисунок 4. Изображение объектов (x, y) на плоскости.

Очевидно, что невозможно построить одну линию, отделяющую объекты противоположных классов. Однако возможно построить несколько линий – для решения задачи «исключающего ИЛИ-2» достаточно двух. Воплощением такой модели являются два обученных на одинаковых исходных данных линейных логистических классификатора – первый отделяет отрицательный объект $(0, 0)$, задача второго – отделить отрицательный объект $(1, 1)$.

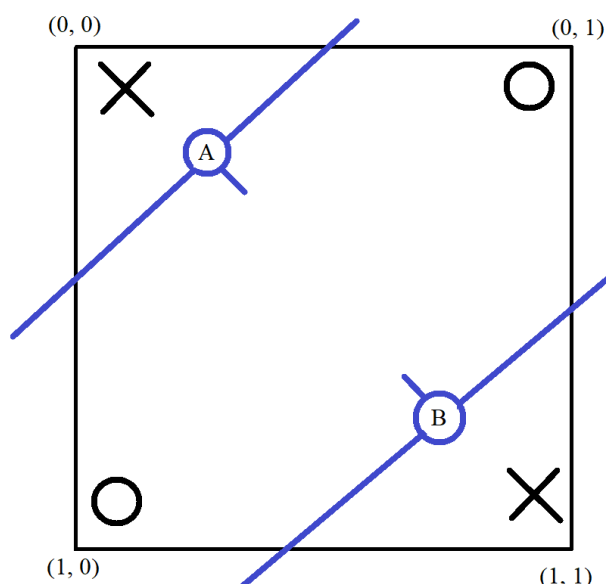


Рисунок 5. Две линии, построенные классификаторами А и В, а также нормали к ним.

Для классификатора А объекты $(0, 1)$, $(1, 0)$ и $(1, 1)$ будут принадлежать к положительному классу, а объект $(0, 0)$ – к отрицательному. Классификатор В выделяет объект $(1, 1)$ как отрицательный, остальные для него – положительные. Любой линейный классификатор, получая на вход вектор признаков, выдает скалярный ответ – степень своей уверенности $\gamma \rightarrow 0$ для отрицательных объектов и $\gamma \rightarrow 1$ для положительных. Так как в данной ситуации классификаторов два, система не даст однозначного ответа – на выходе мы увидим лишь «точку зрения» каждой логистической регрессии. Обобщить данные можно с помощью еще одного линейного классификатора, подав в качестве признаков ответы предыдущих. Рассмотрим множество этих ответов:

| Объект | Ответ А | Ответ В |
|----------|-------------|-------------|
| $(0, 0)$ | ≈ 0 | ≈ 1 |
| $(0, 1)$ | ≈ 1 | ≈ 1 |
| $(1, 0)$ | ≈ 1 | ≈ 1 |
| $(1, 1)$ | ≈ 1 | ≈ 0 |

Таблица 1. Ответы классификаторов

Подадим ответы классификаторов на вход третьему в качестве признаков объектов. Стоит заметить, что, так как признаки есть координаты объекта в пространстве признаков, сами объекты изменили своё местоположение:

| Начальные координаты | Конечные координаты |
|-------------------------|------------------------|
| (0, 0) | (0, 1) |
| (0, 1) | (1, 1) |
| (1, 0) | (1, 1) |
| (1, 1) | (1, 0) |

Таблица 2. Измененные координаты.

Изобразим новое пространство признаков:

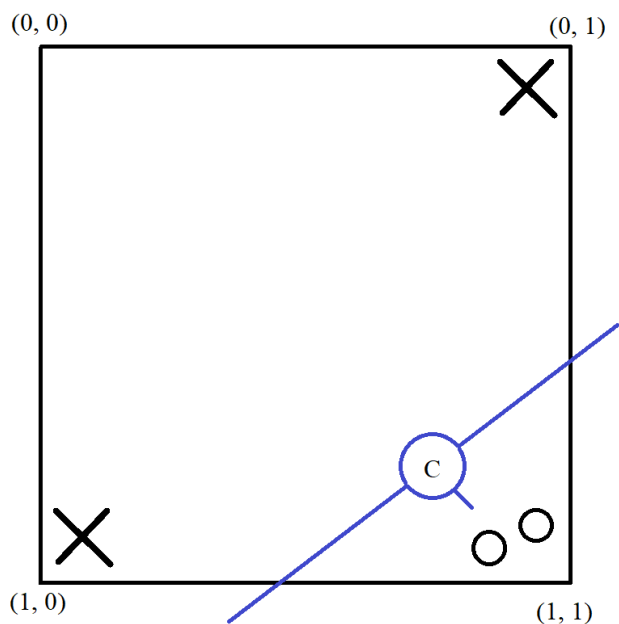


Рисунок 6. Новое пространство признаков.

Видно, что теперь объекты возможно разделить одним, третьим, линейным классификатором.

Иногда обобщения очередным классификатором в конце не требуется – поставленная задача требует ответа в векторной форме. Такими задачами являются, например, задачи множественной классификации, когда классов объектов больше двух. В таком случае можно смотреть на вектор ответов «как есть», а метку класса кодировать двоично:

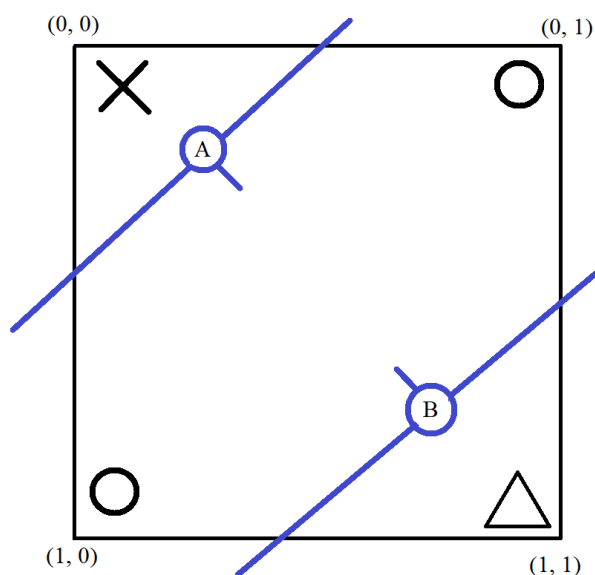


Рисунок 7. Множественная классификация.

| Объект | Ответ А | Ответ В | Метка | Класс |
|--------|-------------|-------------|-------|-------------|
| (0, 0) | ≈ 0 | ≈ 1 | 0-1 | Крест |
| (0, 1) | ≈ 1 | ≈ 1 | 1-1 | Круг |
| (1, 0) | ≈ 1 | ≈ 1 | 1-1 | Круг |
| (1, 1) | ≈ 1 | ≈ 0 | 1-0 | Треугольник |

Таблица 3. Таблица ответов и меток для классификации с тремя классами.

Выше было сказано, что объекты переместились в ходе работы классификаторов А и В, однако это не совсем корректно – в ходе работы модели трансформировалось само пространство признаков. Объекты по-прежнему характеризуются своими изначальными предикторами, поданными на вход системе, состоящей из *двух слоев классификаторов*. Такая система называется *двухслойной нейронной сетью прямого распространения*, где нейрон – простейшая логистическая регрессия. Нейронные сети применяются для решения задач классификации *линейно неразделимых множеств* – каждый слой сети трансформирует пространство признаков до тех пор, пока объекты не станут возможно разделить гиперплоскостью.

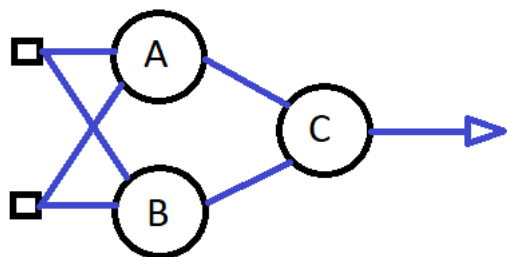


Рисунок 8. Схема нейронной сети из трех классификаторов.

На рисунке выше представлена схема полученной структуры. Слой классификаторов, состоящий из моделей A и B называется *скрытым*. Число логистических регрессий, входящих в этот слой, соответствует числу выстраиваемых сетью гиперплоскостей. Количество скрытых слоев равно количеству трансформаций, через которые проходит пространство исходных признаков – в данном случае такая трансформация единственна; трансформированный вектор затем подается на вход классификатору C, который осуществляет финальное линейное разделение классифицируемых объектов.

2.3 Обучение нейронной сети. Градиентный спуск.

По сути своей нейронная сеть является функцией, отображающей исходный вектор признаков в вектор ответов. Множеством параметров этой функции является совокупность параметров индивидуальных логистических регрессий – коэффициентов выстраиваемых ими плоскостей β . Размерность гиперплоскости, выстраиваемой конкретным классификатором, определяется размерностью входного вектора признаков. Каждый признак затем подставляется как предиктор на соответствующее место в уравнение гиперплоскости вида $\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_q x_q$.

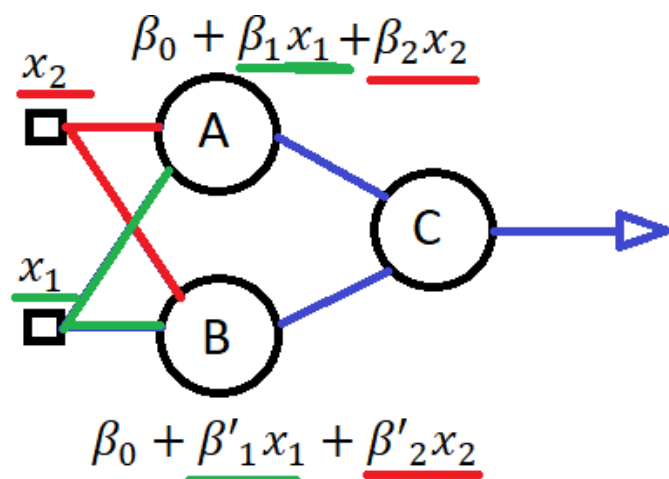


Рисунок 8. Подстановка компонент вектора признаков в уравнения гиперплоскостей внутри классификаторов.

На рисунке выше проиллюстрирована подстановка компонент вектора признаков в уравнения внутри классификаторов. Каждая подстановка обозначена линией, соединяющей элементы входного вектора и нейроны классификаторов. С каждой подстановкой связан уникальный коэффициент β , с которым признак входит в уравнение гиперплоскости. В терминологии нейронных сетей подстановка называется *синаптической связью*, а коэффициент – *весом* этой связи.

Обучение нейронной сети сводится к нахождению параметров β , при которых модель максимально точно классифицирует объекты обучающей выборки. Характеристикой точности является квадратичная функция ошибок, которая определяется как половина суммы квадратов разностей между фактическим и ожидаемым выходом каждого нейрона выходного слоя:

$$E = \frac{1}{2} \sum (y - \hat{y})^2$$

Где \hat{y} – фактический выход, y – ожидаемый.

Наилучшая точность модели достигается при значениях параметров β таких, что E принимает наименьшее значение. Вычисление параметров функции происходит с помощью итеративного алгоритма градиентного спуска.

Градиент функции определяется как вектор в пространстве аргументов функции, состоящий из производных функции по этим аргументам, и показывает направление наибольшего возрастания функции:

$$\nabla f = \left(\frac{\partial f}{\partial w_1}, \frac{\partial f}{\partial w_2}, \dots, \frac{\partial f}{\partial w_n} \right)$$

Используя градиент, можно итеративно приблизить минимум функции f и найти вектор аргументов, при котором этот минимум достигается. Для этого необходимо взять случайный вектор аргументов, с которого алгоритм начнёт минимизацию, а затем последовательно пересчитывать этот вектор, используя градиент функции в каждой новой точке:

$$\vec{w}^{(t)} = \vec{w}^{(t-1)} - \mu \nabla f(\vec{w}^{(t-1)})$$

Поясним данную формулу: градиент есть вектор, состоящий из производных функции по каждому аргументу; каждая производная в свою очередь показывает, как изменится функция при малом приросте данного аргумента; самое важное – знак производной показывает *направление* изменения функции – в сторону увеличения или уменьшения. Необходимо понять, как изменить каждый аргумент, чтобы приблизить функцию к минимуму – решением может послужить изменение каждого аргумента на соответствующую компоненту градиента, взятого с противоположным знаком. Если производная положительная, функция увеличивается при приросте аргумента, задача же требует уменьшения функции – *уменьшаем* аргумент; если производная отрицательная, функция уменьшается при приросте – *увеличиваем* аргумент. Модуль же производной показывает, насколько резко изменится значение функции при малом приросте аргумента, поэтому разумно сделать шаг аргумента пропорциональным производной (коэффициент пропорциональности μ), ведь важно не «перескочить» минимум слишком большими шагами аргумента, производная же уменьшается по мере приближения к минимуму. Следовательно, уменьшается и шаг градиентного спуска.

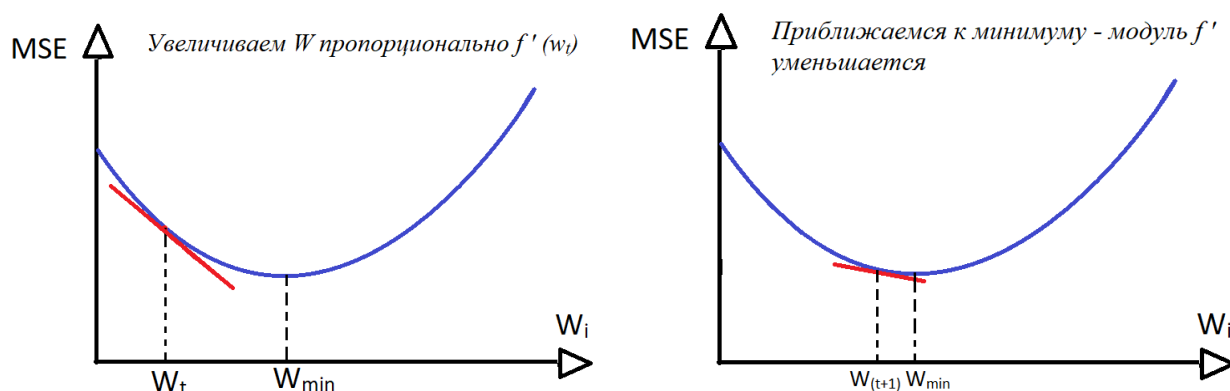


Рисунок 9. Градиентный спуск.

На рисунке 9 изображен пример процесса градиентного спуска по аргументу w_i . Синим цветом изображен график функции ошибки, красным – касательная, по углу наклона которой можно судить о значении производной в точке касания. При приближении к минимуму этот угол, а также производная уменьшаются – уменьшается и шаг градиентного спуска. В самом минимуме производная равна нулю – спуск прекращается. На практике градиентным спуском очень сложно найти точный минимум функции, поэтому обычно алгоритм останавливается после заданного числа итераций или же после достижения точки, достаточно близкой к минимуму.

2.3 Подсчёт градиента функции ошибки методом обратного распространения.

Для вычисления градиента функции ошибки необходимо найти производную функции E по каждому из её параметров β :

$$\frac{\partial E}{\partial \beta_{ji}} = ?$$

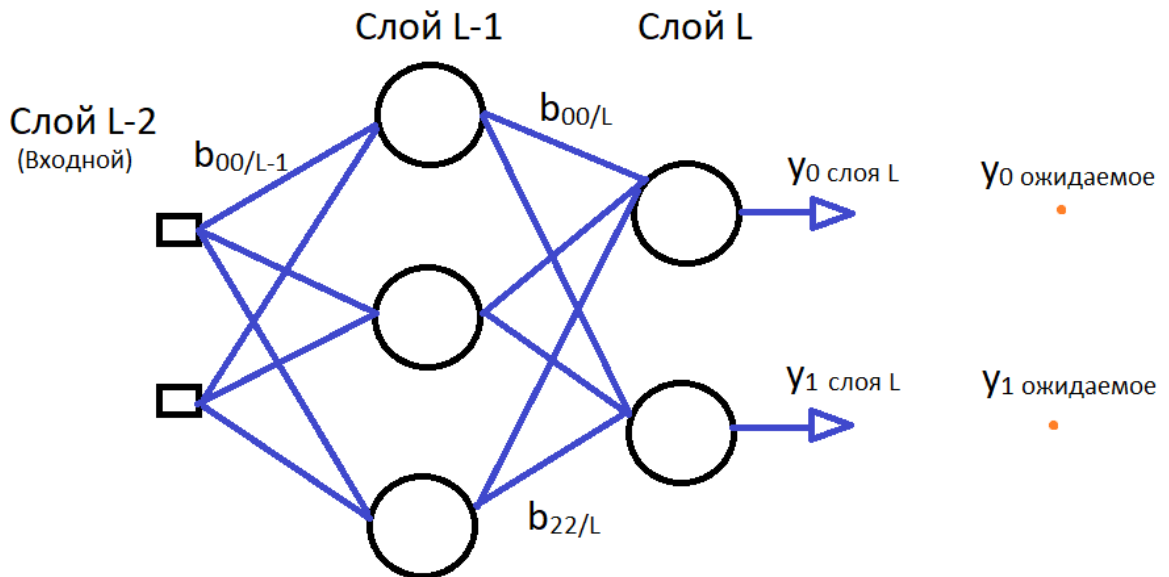


Рисунок 10. Пример схемы сети для демонстрации наименования слоев и весов. Слои подписаны, также подписаны некоторые веса.

Для того, чтобы проследить зависимость функции ошибки от параметров β_{ji}^L (на рисунке $b_{ji/L}$) воспользуемся правилом нахождения производной сложной функции:

$$\frac{\partial E}{\partial \beta_{ji}^L} = \frac{\partial E}{\partial y_j^L} \frac{\partial y_j^L}{\partial v_j^L} \frac{\partial v_j^L}{\partial \beta_{ji}^L}$$

Поясним: необходимо найти производную функции ошибки по весу β_{ji}^L , соединяющему нейрон j слоя L с нейроном i слоя $L - 1$. Вес β_{ji}^L непосредственно влияет на степень уверенности нейрона j слоя L ∂v_j^L , которая в свою очередь влияет на выход этого нейрона y_j^L ($y_j^L = \sigma(v_j^L)$), выход же нейрона непосредственно влияет на функцию ошибки:

$$E = \frac{1}{2} ((y_j^{\text{ожид}} - y_j^L)^2 + (y_j^{\text{ожид}} - y_{\text{от другого нейрона}}^L) + \dots)$$

Найдём производную $\frac{\partial v_j^L}{\partial \beta_{ji}^L}$: $v_j^L = \beta_{ji}^L y_i^{L-1} + \dots \rightarrow \frac{\partial v_j^L}{\partial \beta_{ji}^L} = y_i^{L-1}$. Таким образом, степень влияния веса на уверенность нейрона зависит от сигнала предыдущего нейрона y_i^{L-1} , который этот вес соединяет с рассматриваемым нейроном y_j^L .

Найдём производную $\frac{\partial y_j^L}{\partial v_j^L}$: $\frac{\partial y_j^L}{\partial v_j^L} = \sigma'(v_j^L)$

Способ нахождения множителя $\frac{\partial E}{\partial y_j^L}$ будет различаться в зависимости от того, является ли рассматриваемый нейрон y_j^L нейроном выходного или же скрытого слоя. Для нейрона выходного слоя:

$$\frac{\partial E}{\partial y_j^L} = ((y_j^{\text{ожид}} - y_j^L))$$

Для нейрона скрытого слоя:

$$\frac{\partial E}{\partial y_j^L} = \sum_{k=0}^n \left(\frac{\partial E}{\partial y_k^{L+1}} \frac{\partial y_k^{L+1}}{\partial v_k^{L+1}} \frac{\partial v_k^{L+1}}{\partial y_j^L} \right) = \sum_{k=0}^n \left(\frac{\partial E}{\partial y_k^{L+1}} \frac{\partial y_k^{L+1}}{\partial v_k^{L+1}} \beta_{kj}^{L+1} \right)$$

Величину $\frac{\partial E}{\partial y_j^L} \frac{\partial y_j^L}{\partial v_j^L}$ назовем локальным градиентом нейрона j и обозначим δ_j .

Тогда $\delta_j = \begin{cases} \sigma'(v_j^L) ((y_j^{\text{ожид}} - y_j^L)), & L - \text{выходной слой} \\ \sigma'(v_j^L) \sum_{k=0}^n (\delta_k \beta_{kj}^{L+1}), & L - \text{скрытый слой} \end{cases}$

Обратное распространение ошибки заключается в рекурсивном подсчете локальных градиентов для нейронов скрытых слоев.

3. Решение поставленной задачи.

Ниже будут приведены схема используемой нейронной сети и код написанной на python 3 библиотеки `ptron`, реализующей данную сеть.

а. Структура нейронной сети

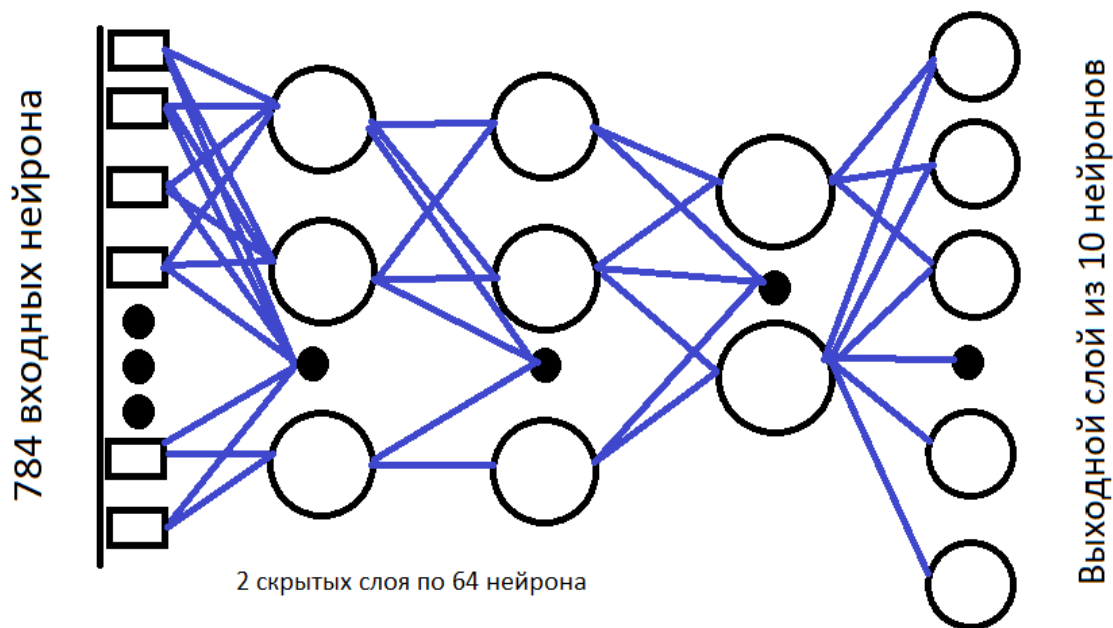


Рисунок 11. Схема нейронной сети.

Для решения поставленной задачи используется многослойная нейронная сеть прямого распространения, принимающая на вход вектор признаков, состоящий из 784 значений. Каждое значение принадлежит отрезку от 0 до 1 и соответствует одному пикселю изображения 28x28. Величина значения говорит о степени насыщенности черного цвета в данном пикселе, где 0 – белый, 1 – абсолютно черный. В нейронной сети имеется 3 скрытых слоя, 2 из которых содержат 64 нейрона, третий – 32. Количество слоев было подобрано экспериментально для обеспечения наилучших качества классификации и скорости работы сети. Выходной слой содержит 10 нейронов, каждый из которых соответствует одной десятичной цифре.

3.3. Библиотека `ptron`.

Библиотека включает в себя следующие классы и методы:

- Класс `Node` – реализация нейрона сети. Методы класса:
 1. **`__init__`** : метод, вызывающийся при создании экземпляра класса.
Принимает на вход позицию нейрона в сети – индекс слоя и индекс в слое. Инициализирует нейрон, объявляет атрибуты нейрона: *weights* – веса исходящих из нейрона связей, *bias_weight* – вес сдвига, *local_grad* – локальный градиент, *induced_field* – степень уверенности нейрона, *weights_delta* – поправка к весам, *bias_weight_delta* – поправка к сдвигу. Значения атрибутов вычисляются в ходе работы программы.
 2. **`connect_node`**: метод, соединяющий нейрон со всеми нейронами следующего за ним слоя. Веса инициализируются случайно. Принимает на вход объект класса `Layer`.
 3. **`apply_deltas`**: применяет поправки к весам. Вызывается в ходе обучения.
 4. **`modify_bias`**: применяет поправки к сдвигу. Вызывается в ходе обучения.
- Класс `Layer` – реализация слоя нейронной сети. Использует объекты класса `Node`. Методы класса:
 1. **`__init__`**: вызывается при создании слоя. Принимает на вход количество нейронов и свой индекс слоя в сети. Инициализирует список объектов класса `Node` в атрибуте *nodes*.
 2. **`connect_layer`**: соединяет слой с предыдущим слоем в сети, который передается в метод в качестве аргумента. Вызывает метод `connect_node` для каждого нейрона в списке переданного слоя.
 3. **`start_forward_propagation`**: начинает распространение сигнала через нейронную сеть. Предназначен исключительно для входного слоя.
Работа метода полностью аналогична нижеописанному

`propagate_forward` с единственным различием – входной вектор признаков не проходит через функцию активации.

4. **`propagate_forward`**: преобразует поданный в качестве аргумента методу сигнал, используя нейроны слоя. Строит матрицу весов собственных нейронов, добавляет в матрицу веса для сдвига, хранящиеся в нейронах следующего слоя. Осуществляет матричное перемножение вектора и матрицы весов, результат затем проходит через сигмоидальную функцию активации.
5. **`start_backpropagation`**: начинает обратное распространение ошибки. Предназначен исключительно для выходного слоя. Подсчитывает и сохраняет локальный градиент каждого собственного нейрона, вычисляет поправки к весам. Принимает на вход коэффициент скорости обучения μ и вычисленный вектор ошибки $(\vec{y} - \hat{\vec{y}})$.
6. **`propagate_backward`**: реализует рекурсивный подсчет локальных градиентов для собственных нейронов. На вход принимает следующий в сети слой, локальные градиенты нейронов которого используются для подсчета собственных. Вычисляет поправки к весам для собственных нейронов.

- Класс `Network` – реализация нейронной сети. Использует объекты класса `Layer`. Методы класса:

1. **`__init__`**: вызывается при инициализации объекта класса. Принимает на вход кортеж, содержащий информацию о структуре сети в формате $(n_1, n_2, n_3, \dots, n_m)$ – где n – число нейронов в каждом слое. Объявляет атрибуты класса *structure* – кортеж структуры сети, *signal* – вектор признаков, проходящий через сеть, *layers* – список слоев, *mse* – текущая функция ошибки, *epochs* – число прошедших эпох обучения, *printing_option* – формат вывода для процесса обучения.

Инициализирует список слоев объектами класса `Layer`, соединяет слои

методом `connect_layer` данного класса. Случайным образом инициализирует вес сдвига для каждого нейрона сети.

2. **feed_forward**: отображает поданный на вход вектор признаков в вектор ответов, прогоняя его через всю структуру сети. Использует методы `start_forward_propagation` и `propagate_forward` объектов класса `Layer`.
3. **feed_backward**: вычисляет поправки ко всем весам, обратно распространяя поданный на вход вектор ошибки. Использует методы `start_backpropagation` и `propagate_backward` объектов класса `Layer`.
4. **learning_iteration**: выполняет одну итерацию обучения на поданных на вход в виде кортежа обучающих данных. Также принимает на вход коэффициент скорости обучения μ .
5. **set_printing_option**: устанавливает поданный на вход формат вывода для процесса обучения.
6. **save**: сохраняет модель в двоичном формате, создавая новый файл с поданным на вход в качестве строки именем.
7. **ask**: позволяет опрашивать сеть. Отображает поданный на вход вектор признаков в вектор ответа и печатает его.

Библиотека включает в себя следующие функции:

- **load_model**: загружает модель из файла, название файла передается строкой в качестве аргумента. Возвращает модель.
- **sigmoid**: логистическая функция активации. Принимает на вход вещественное число x , возвращает $\sigma(x)$.
- **s_df**: производная логистической функции активации. Принимает на вход вещественное число x , возвращает $\sigma'(x)$.

Библиотека написана на Python 3.7 и использует исключительно пакеты `numpy` и `pickle`.

Выводы

В ходе работы были выполнены поставленные задачи: разработана библиотека для Python 3, позволяющая строить модели полносвязной нейронной сети прямого распространения; с помощью данной библиотеки была успешно решена задача классификации рукописных цифр из набора MNIST.

Список источников.

1. synset.com/ai/ru/nn/NeuralNet_01_Intro.html
2. youtube.com/channel/UC4UJ26WkceqONNF5S26OiVw
3. youtube.com/channel/UCYO_jab_esuFRV4b17AjtAw
4. habr.com/ru/company/io/blog/265007/
5. Хайкин, С. Нейронные сети: полный курс, 2-е издание. : Пер. с англ. – М. : Издательский дом “Вильямс”, 2006. – 1104 с. : ил. – Парал. тит. англ. ISBN 5-8459-0890-6 (рус.)
6. Рашид, Тарик. Создаем нейронную сеть. :Пер. с англ. – СПб.: ООО “Диалектика”, 2019. – 272 с. : ил. – Парал. тит. англ. ISBN 978-5-9909445-7-2 (рус.)

Приложение 1. Код библиотеки ptron.

```
"""
(C) Ivan Chanke 2020

Contains classes:
    Node -> Layer -> Network
The three constantly refer to one another and aren't supposed to work
separately
For details on math model visit GitHub directory corresponding to the project
"""
import numpy as np
import pickle

bias_signal = 1

def load_model(file):
    """
    Loads network model from file
    """
    f = open(file, 'rb')
    model = pickle.load(f)
    f.close()

    return model

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def s_df(x):
    return sigmoid(x) * (1 - sigmoid(x))

class Node:
    """
    Stores:
        Its position in the network (position, layernum)
        A vector of weights going out of it.
        Local gradient
        Induced field
        Weights deltas vector
        Bias weight delta
        Its own bias weight; bias signal is always 1
    """
    def __init__(self, position):
        """
        position is a (node index, layer index) tuple
        """
        self.position = position[0]
        self.layernum = position[1]
        self.weights = None
        self.bias_weight = None
        self.local_grad = None
        self.induced_field = None
```

```

        self.weights_delta = None
        self.bias_weight_delta = 0

    def connect_node(self, layer):
        """
        Initializes a vector of weights for synapses this node has with the
        next layer
        Weights are initialized randomly
        """
        self.weights = (np.random.rand(layer.nnodes)) * 2 - 1
        self.weights_delta = np.zeros(layer.nnodes)

    def apply_deltas(self):

        self.weights += self.weights_delta
        self.weights_delta = np.zeros(self.weights_delta.shape)

    def modify_bias(self):

        self.bias_weight += self.bias_weight_delta
        self.bias_weight_delta = 0


class Layer:
    """
    Handles backpropagation and feeding forward.
    Stores:
        Layer index
        A list of nodes
        Number of nodes in the list (nnodes)
    """
    def __init__(self, nnodes, number):

        self.number = number
        self.nnodes = nnodes
        self.nodes = [Node((i, number)) for i in range(nnodes)]

    def connect_layer(self, other):
        """
        Connects layer self with the PREVIOUS layer other
        """
        for node in other.nodes:
            node.connect_node(self)

    def start_forward_propagation(self, other, vector):
        """
        Begins feeding forward
        This method works similarly to "propagate_forward" defined below
        The only difference is that the input vector doesn't go through the
        activation function
        """
        bias_weights_vector = np.array([node.bias_weight for node in
other.nodes])

        weights_stack = [node.weights for node in self.nodes]
        weights_stack.append(bias_weights_vector)

        memory_matrix = np.vstack(weights_stack)
        memory_matrix = np.transpose(memory_matrix)

        for i in range(len(self.nodes)):
            self.nodes[i].induced_field = vector[i]

```



```

        vector = vector
        vector = np.append(vector, bias_signal)

        return memory_matrix.dot(vector)

    def propagate_forward(self, other, vector):
        """
        Maps input vector of layer i to input vector of layer i+1
        """
        bias_weights_vector = np.array([node.bias_weight for node in
other.nodes]) # Composes a vector of bias weights for layer other

        weights_stack = [node.weights for node in self.nodes]
        weights_stack.append(bias_weights_vector)

        memory_matrix = np.vstack(weights_stack) # Stacks up weight vectors
of each node in self and bias weight vector of other
        memory_matrix = np.transpose(memory_matrix) # Each column is a weight
vector; ncols = nnodes + 1, nrows + 1 = nsynapses;

        for i in range(len(self.nodes)):
            self.nodes[i].induced_field = vector[i] # Each node stores its
induced field which is later used to compute local_grad

        vector = sigmoid(vector) # Vector goes through the activation
function
        vector = np.append(vector, bias_signal) # Bias signal is added to a
vector

        return memory_matrix.dot(vector) # Returns input vector for the next
layer

    def start_backpropagation(self, e, learning_rate): # For output layer
only; computes local_grad for each node in self; e - error vector
        """
        Computes a local gradient for each neuron in the output vector
        """
        for i in range(len(self.nodes)):
            self.nodes[i].local_grad = s_df(self.nodes[i].induced_field) *
e[i]

            self.nodes[i].bias_weight_delta += learning_rate *
self.nodes[i].local_grad * bias_signal

    def propagate_backward(self, other, learning_rate): # For layers except
output only; connection scheme: self-other
        """
        Computes local gradients for nodes in self

        """
        for i in range(len(self.nodes)): # Local_grad for each node in layer
self is computed
            lgv = np.array([node.local_grad for node in other.nodes]) # Local
gradient vector for other
            d = s_df(self.nodes[i].induced_field) * np.dot(lgv,
np.transpose(self.nodes[i].weights))
            self.nodes[i].local_grad = d
        """
        Computes deltas using local gradients
        """
        for node in self.nodes:

```

```

        deltas = []
        for i in range(len(node.weights)):
            delta = learning_rate * sigmoid(node.induced_field) *
other.nodes[i].local_grad
            deltas.append(delta)

        bias_delta = learning_rate * node.local_grad * bias_signal

        node.weights_delta += np.array(deltas)
        node.bias_weight_delta += bias_delta

class Network:
    """
    Stores:
        Current signal
        A list of layers
        last mse
        Number of epoch trained
        Task it performs
    Initializing a network also initializes its layers and nodes in them
    """
    def __init__(self, structure): # Structure is a tuple of nnodes in each
layer
        self.structure = structure
        self.signal = None
        self.layers = []
        self.mse = None
        self.epochs = 0
        self.printing_option = None

        for i in range(len(structure)): # Constructing layers
            self.layers.append(Layer(structure[i], i))

        for i in range(1, len(self.layers)):
            self.layers[i].connect_layer(self.layers[i - 1])

        for i in range(1, len(self.layers)): # Initializing bias weights
            for j in range(len(self.layers[i].nodes)):
                self.layers[i].nodes[j].bias_weight = (np.random.rand()) * 2
- 1

    def feed_forward(self, vector):
        """
        Maps input-output
        """
        self.signal =
self.layers[0].start_forward_propagation(self.layers[1], vector)

        for i in range(1, len(self.layers) - 1): # Last layer does nothing,
hence range(len - 1)
            self.signal = self.layers[i].propagate_forward(self.layers[i +
1], self.signal)

        for i in range(len(self.layers[-1].nodes)): # Stores last layer's
nodes' induced fields
            self.layers[-1].nodes[i].induced_field = self.signal[i]

        return self.signal

    def feed_backward(self, error_vector, learning_rate):

```

```

        """
        Backpropagation
        Recursively computes deltas for weights; applies them
        """
        self.layers[-1].start_backpropagation(error_vector, learning_rate)

        for i in range(len(self.layers) - 1, 0, -1):
            self.layers[i - 1].propagate_backward(self.layers[i],
learning_rate)

    def learning_iteration(self, batch_tuple, learning_rate):
        """
        One complete learning epoch
        batch_tuple is a tuple of tuples: ((in, desired_out), (...), ...,
(...))
        """
        self.epochs += 1
        self.mse = 0
        for instance in batch_tuple:
            output = sigmoid(self.feed_forward(np.array(instance[0])))
            error_vector = instance[1] - output
            self.mse += sum(error_vector**2)

            self.feed_backward(error_vector, learning_rate)

            if self.printing_option == 1:
                print(instance[0], ': ', output)
            else:
                print(output)

        for node in self.layers[0].nodes:
            node.apply_deltas()

        for i in range(1, len(self.layers) - 1):
            for node in self.layers[i].nodes:
                node.apply_deltas()
                node.modify_bias()

        for node in self.layers[-1].nodes:
            node.modify_bias()

        print('MSE:', self.mse)
        print('-----')

    def set_printing_option(self, arg):
        self.printing_option = arg

    def save(self, file):
        """
        Stores current model as a file
        """
        f = open(file, 'wb')
        pickle.dump(self, f)
        f.close()

    def ask(self, question):

```

```
output = sigmoid(self.feed_forward(np.array(question)))
print('-----')
if self.printing_option == 1:
    print('Input:', np.array(question))
    print('Response:', output)

print('Successfully imported ptron')
```

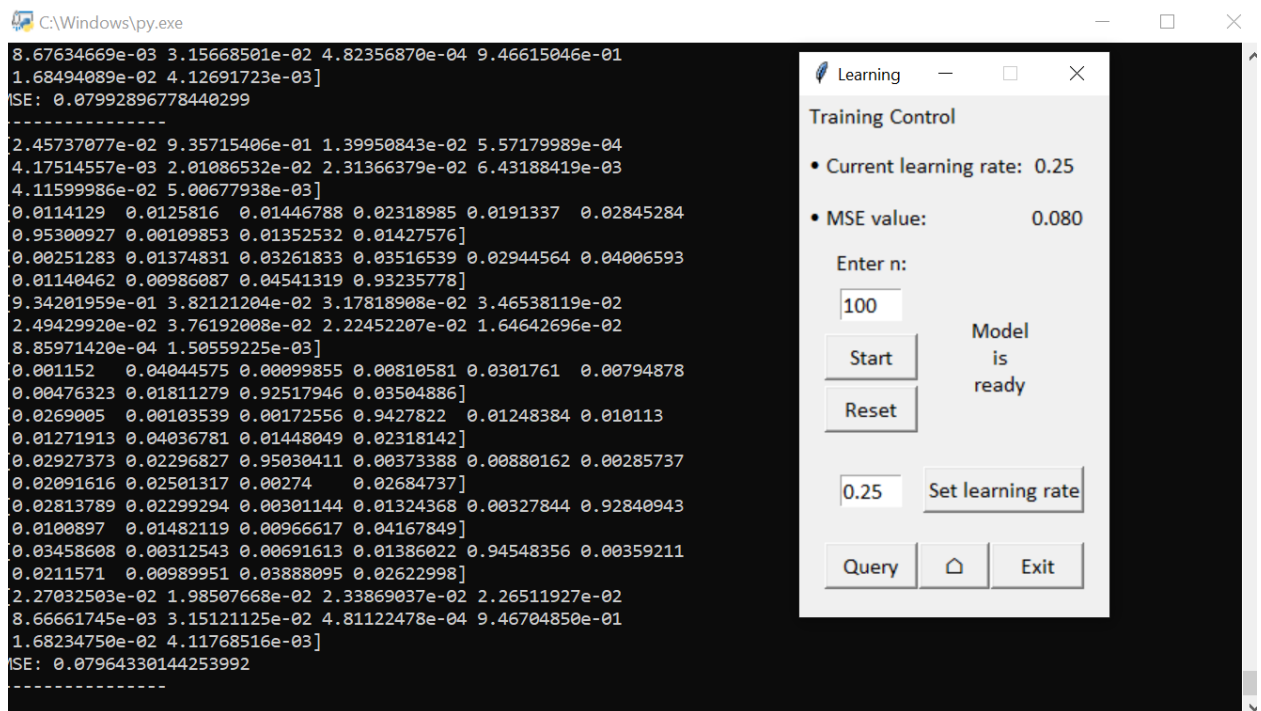



Рисунок 16. Обученная на 500 итерациях модель.

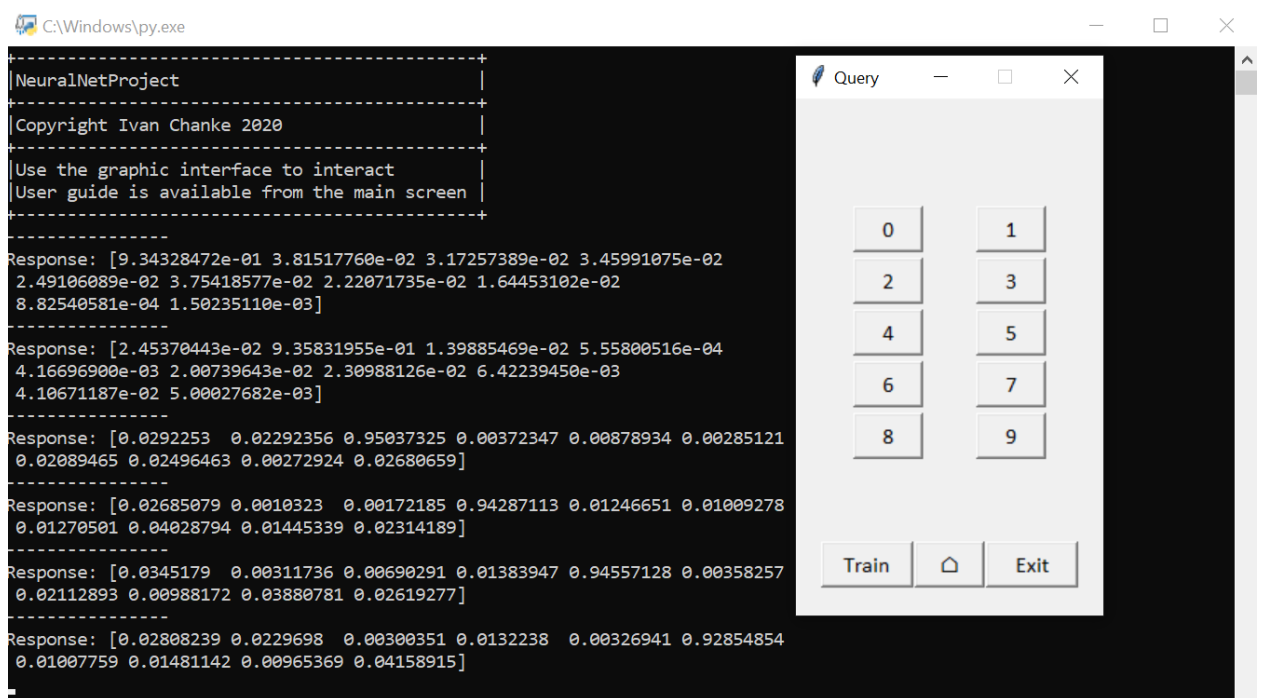


Рисунок 17. На вход обученной модели были поочередно поданы векторы, соответствующие изображениям с цифрами 0, 1, 2, 3, 4 и 5 соответственно. На выходе можно наблюдать активацию нейронов, с индексами 0, 1, 2, 3, 4 и 5 – их значения в выходном векторе близки к единице, остальные – к нулю.

Приложение 3. Руководство пользователя

Обзор главного экрана

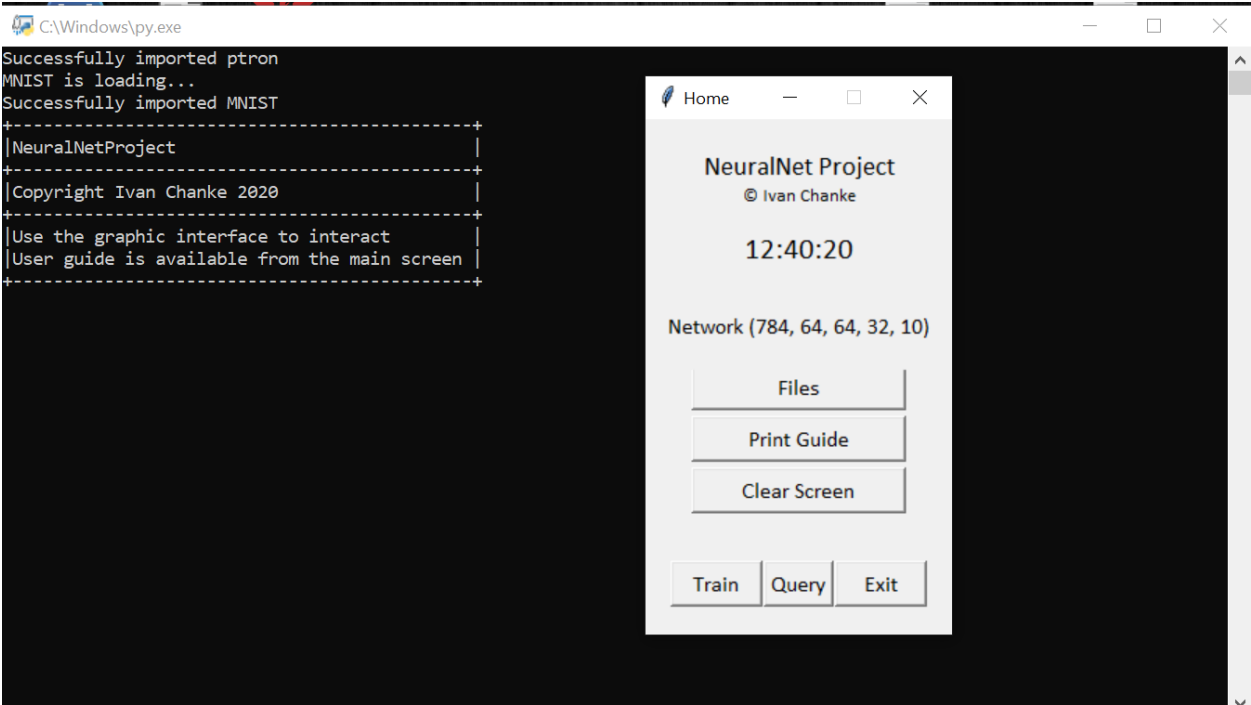


Рисунок 18. Стартовый экран приложения.

На рисунке выше продемонстрирован стартовый экран приложения, а также консоль. Графический интерфейс предоставляет пользователю возможность управлять процессом обучения модели, в то время как консоль служит средством вывода информации о работе программы.

Стартовый экран содержит следующие кнопки:

| Кнопка | Назначение |
|--------------|--|
| Files | Открывает меню, позволяющее сохранить/загрузить модель |
| Print Guide | Печатает краткое руководство по использованию приложения |
| Clear Screen | Очищает консоль |

| | |
|-------|--------------------------------|
| Train | Открывает меню обучения модели |
| Query | Открывает меню опроса модели |
| Exit | Закрывает приложение |

Над кнопками в окне графического интерфейса расположена информация о конфигурации нейронной сети – кортеж из 5 чисел, каждое из которых показывает число нейронов в соответствующем слое.

Меню загрузки/сохранения

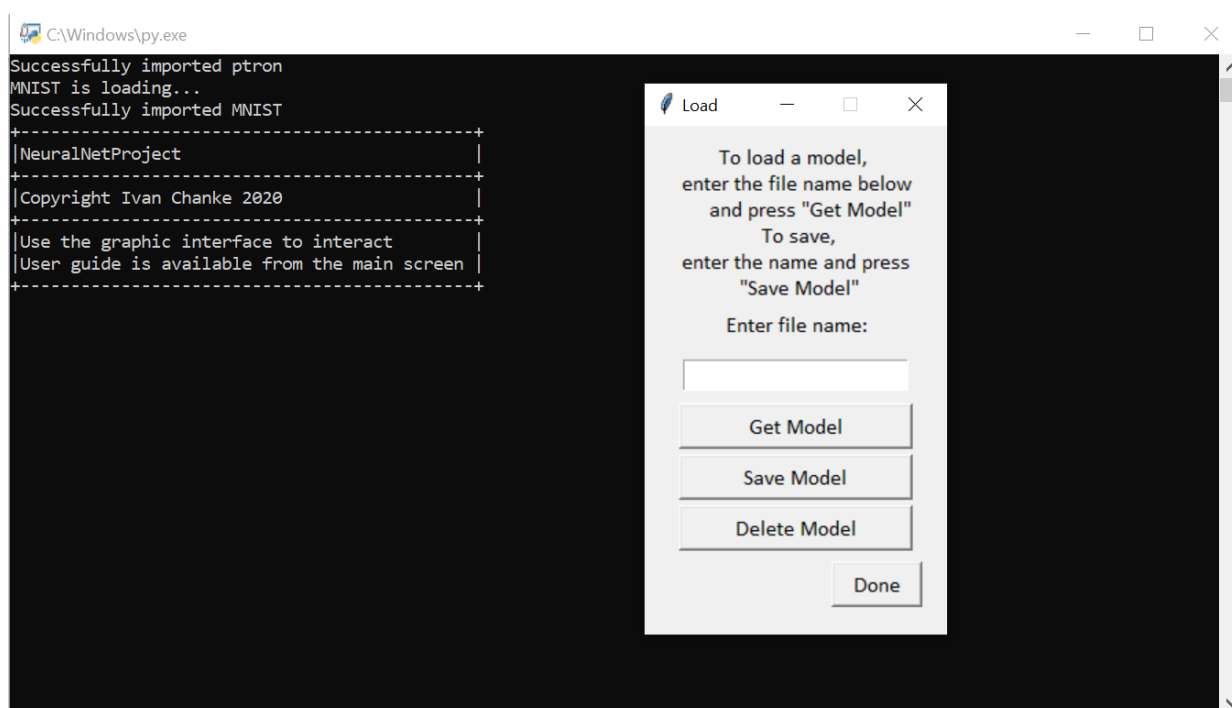


Рисунок 19. Меню загрузки/сохранения.

С помощью данного меню пользователь может сохранять обученную модель, а также загружать сохраненные ранее модели. Для загрузки необходимо ввести в окошко путь к файлу и нажать «Get Model». Для сохранения необходимо ввести имя нового файла и нажать «Save Model». Кнопка «Delete Model» удаляет модель.

Меню обучения модели

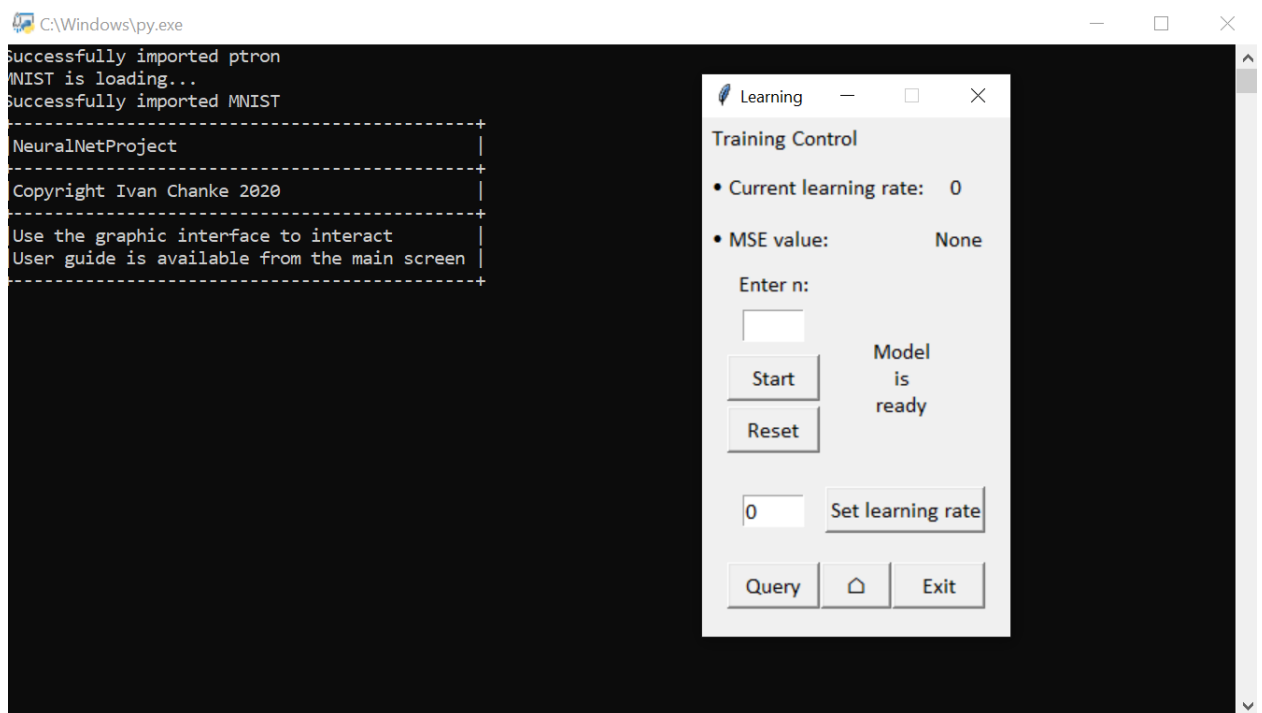


Рисунок 20. Меню управления обучением.

Данное меню позволяет пользователю управлять обучением модели.

Обучающие данные встроены в модель. Перед обучением необходимо выбрать коэффициент скорости обучения и ввести его в окошко, располагающееся слева от кнопки «Set learning rate», а затем нажать на неё. Затем необходимо ввести количество итераций обучения в окошко над кнопкой «Start». Нажатие этой кнопки запустит процесс. После выполнения заданного числа итераций пользователь увидит текущее значение функции ошибки в пункте MSE value.

Нажатие кнопки «Reset» повлечет сброс прогресса обучения, реинициализирует модель.

Меню опроса модели

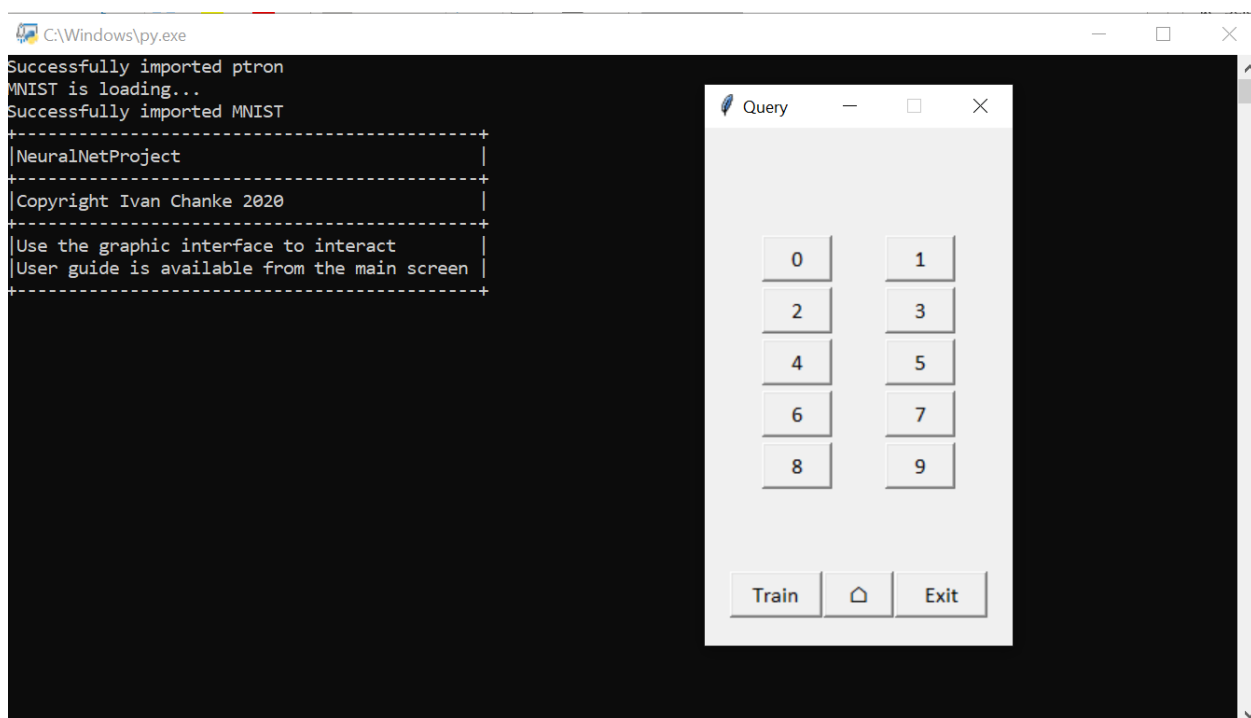


Рисунок 21. Меню опроса модели

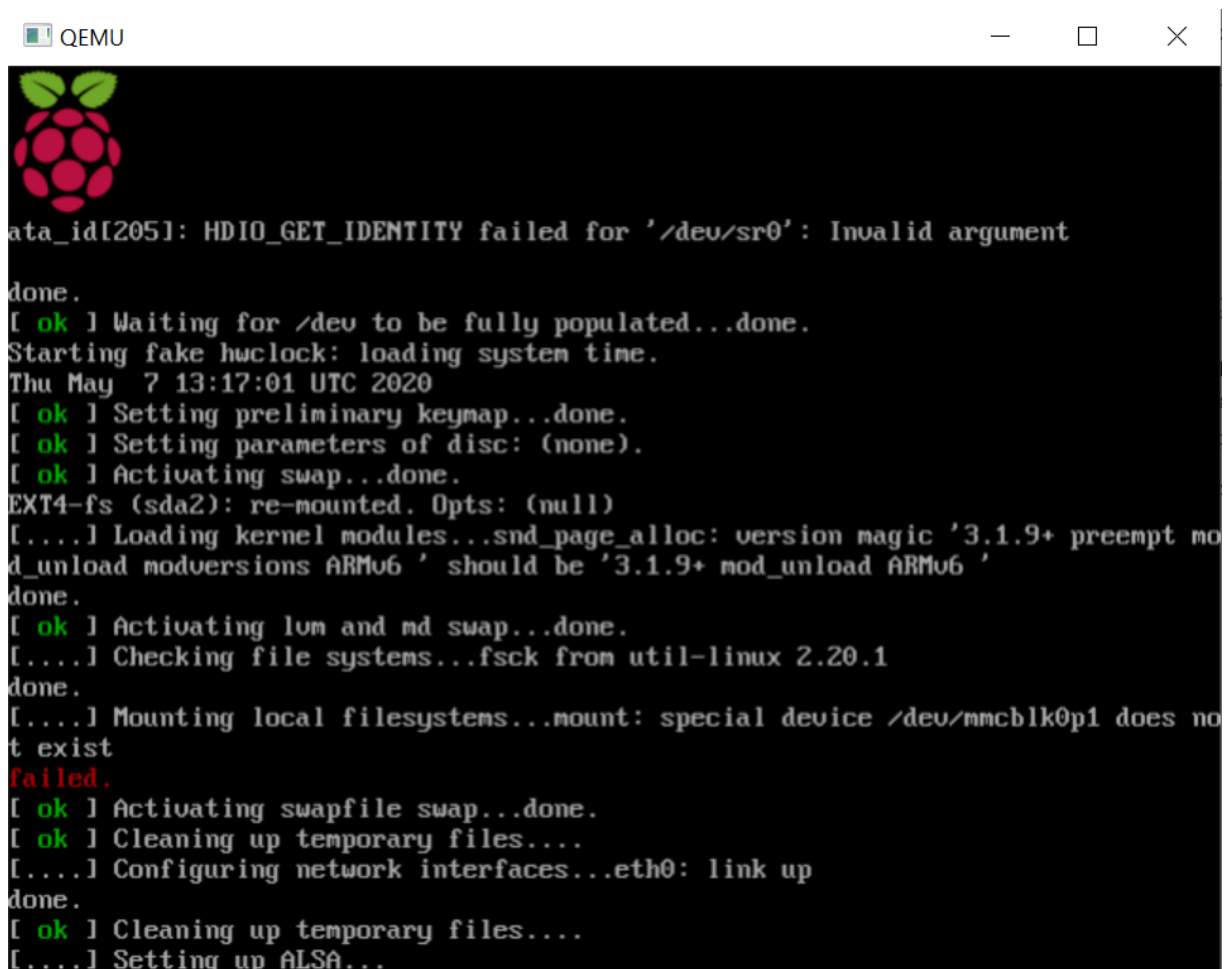
Данное меню позволяет пользователю проверять работу модели. На экране находятся кнопки, подписанные десятичными цифрами. Каждая кнопка подает на вход сети вектор, соответствующий изображению с соответствующей цифрой, в консоли печатается ответ сети.

Скриншоты работы программы в разных режимах приведены в приложении 2.

Приложение 4. Запуск программы на Raspbian.

Разработанное приложение также работает на операционной системе Raspbian, предназначенной для одноплатного компьютера Raspberry Pi. Ниже будет кратко описан процесс запуска программы, а также приведены скриншоты её работы.

Система Raspbian была эмулирована на Windows 10 с помощью QEMU.



```
QEMU
ata_id[205]: HDIO_GET_IDENTITY failed for '/dev/sr0': Invalid argument
done.
[ ok ] Waiting for /dev to be fully populated...done.
Starting fake hwclock: loading system time.
Thu May  7 13:17:01 UTC 2020
[ ok ] Setting preliminary keymap...done.
[ ok ] Setting parameters of disc: (none).
[ ok ] Activating swap...done.
EXT4-fs (sda2): re-mounted. Opts: (null)
[....] Loading kernel modules...snd_page_alloc: version magic '3.1.9+ preempt mo
d_unload modversions ARMv6 ' should be '3.1.9+ mod_unload ARMv6 '
done.
[ ok ] Activating lvm and md swap...done.
[....] Checking file systems...fsck from util-linux 2.20.1
done.
[....] Mounting local filesystems...mount: special device /dev/mmcblk0p1 does no
t exist
failed.
[ ok ] Activating swapfile swap...done.
[ ok ] Cleaning up temporary files....
[....] Configuring network interfaces...eth0: link up
done.
[ ok ] Cleaning up temporary files....
[....] Setting up ALSA...
```

Рисунок 22. Запуск эмулятора.

ОС Raspbian поддерживает работу Python 3. По умолчанию присутствуют стандартный интерпретатор и среда разработки IDLE 3, через которую и производится запуск приложения.

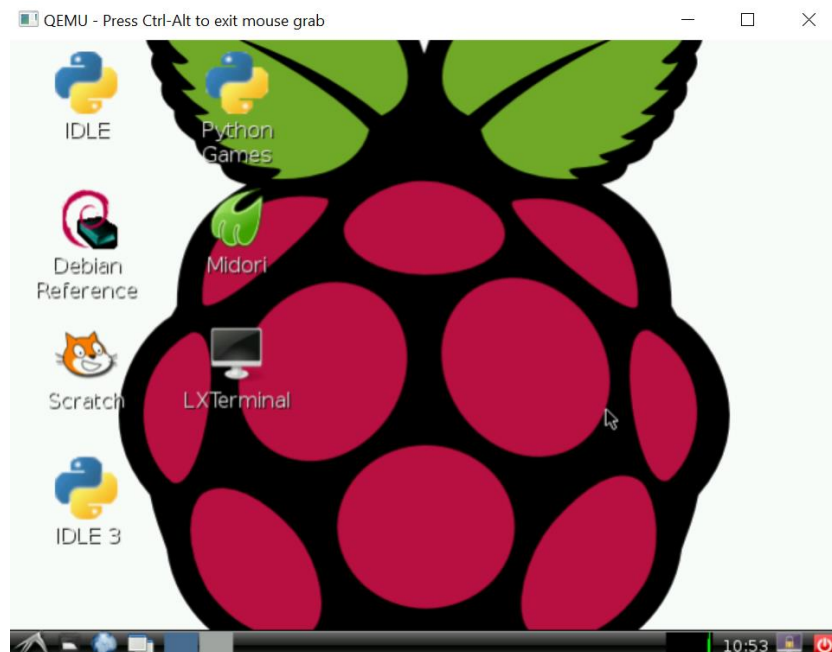


Рисунок 23. Стартовый экран системы Raspbian в эмуляторе.

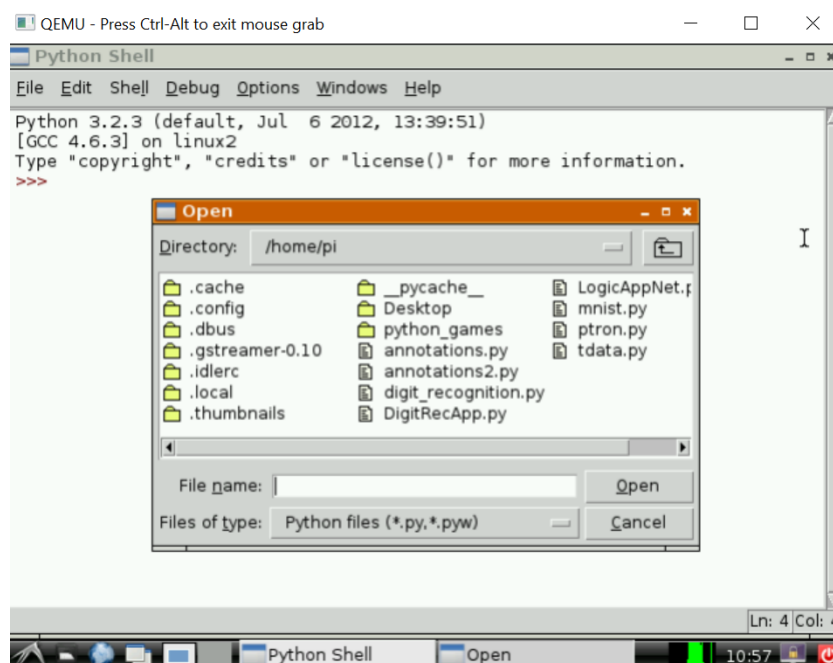


Рисунок 24. Запущенный IDLE 3

На рисунке выше представлено диалоговое окно выбора файла. В директории /home/pi присутствуют кроме прочего файл библиотеки ptron.py, а также скрипт приложения DigitRecApp.py.

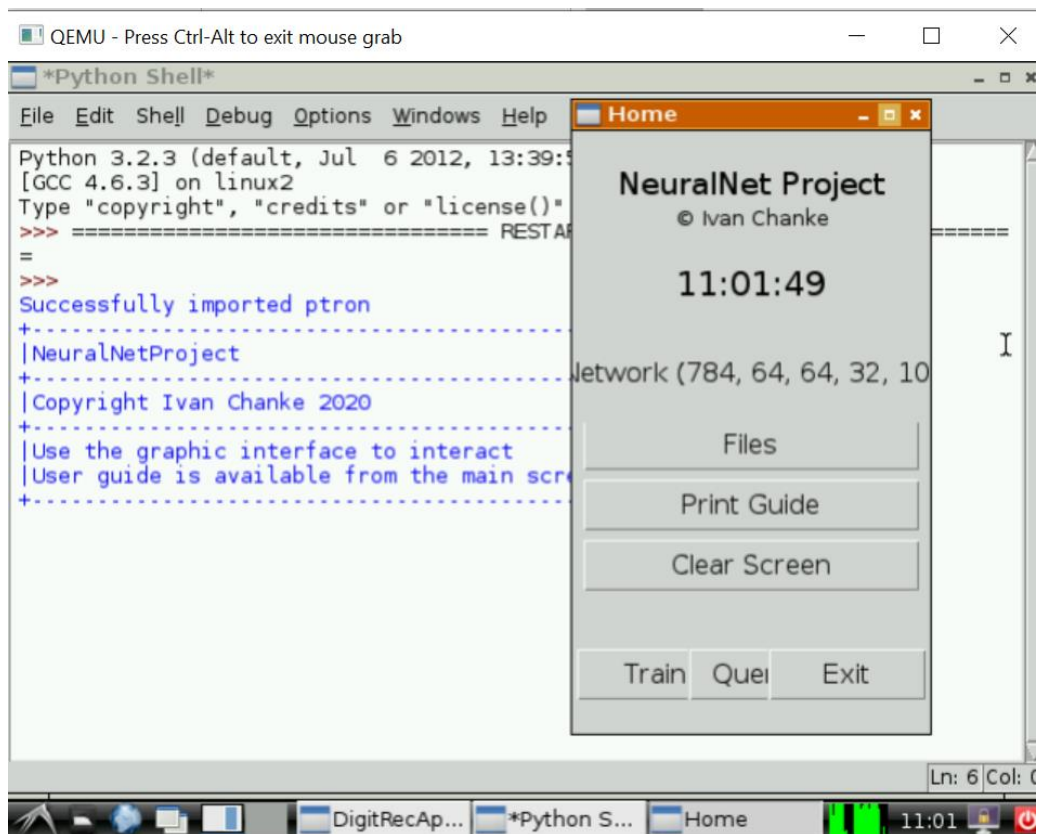


Рисунок 25. Стартовый экран запущенного на Raspbian приложения.

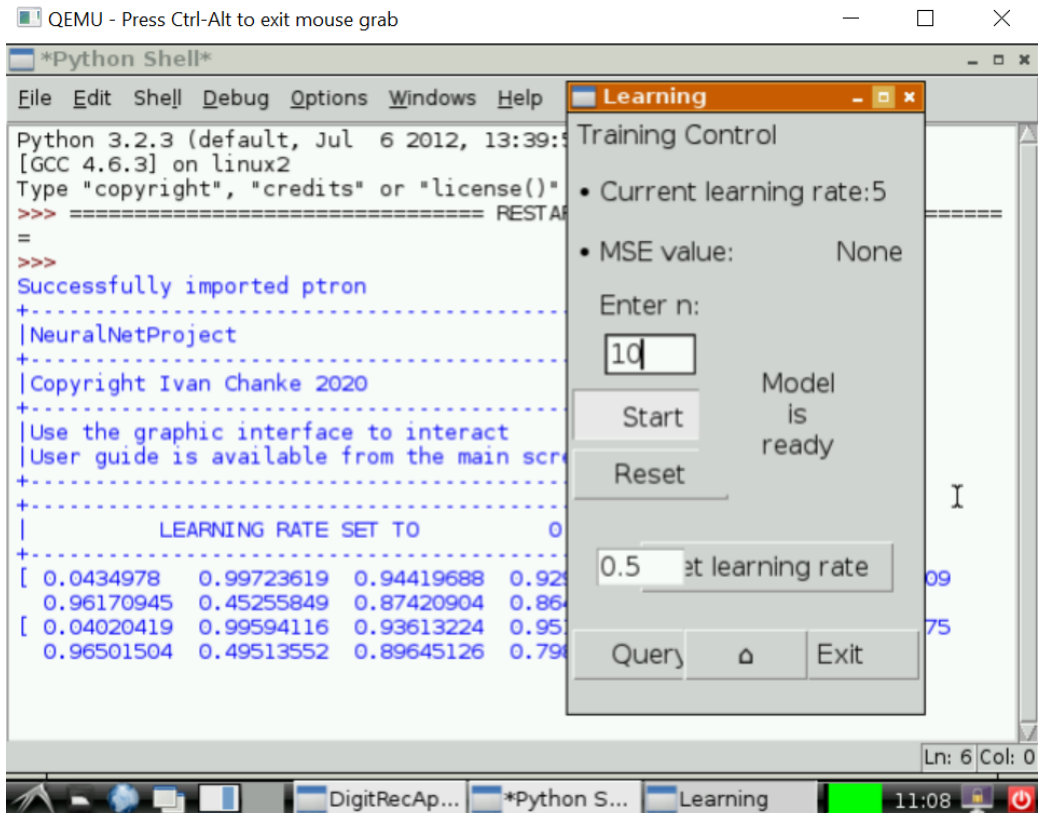


Рисунок 26. Обучающаяся сеть.