|  |
| --- |
| Изображение выглядит как зарисовка, эмблема, символ, герб  Автоматически созданное описание |
| МИНОБРНАУКИ РОССИИ |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА - Российский технологический университет»**  **РТУ МИРЭА** |

Институт Информационных Технологий

Кафедра Вычислительной Техники (ВТ)

**ОТЧЁТ ПО ПРАКТИЧЕСКИМ РАБОТАМ**

по дисциплине

«Проектирование и обучение нейронных сетей»

|  |  |
| --- | --- |
| Выполнил студент группы  ИКБО-04-21 | Даурбеков М.И. |
| Принял старший преподаватель кафедры ВТ | Семенов Р.Э. |
| Практическая работа выполнена | «\_\_»\_\_\_\_\_\_\_2023 г. |
| «Зачтено» | «\_\_»\_\_\_\_\_\_\_2023 г. |

Москва 2023 г.

ОГЛАВЛЕНИЕ

[ВВЕДЕНИЕ 4](#_Toc153843113)

[1 ОБУЧЕНИЕ ПО ПРАВИЛАМ ХЕББА 4](#_Toc153843114)

[1.1 Постановка задачи 5](#_Toc153843115)

[1.2 Теоретический раздел 5](#_Toc153843116)

[1.3 Программная реализация 6](#_Toc153843117)

[2 ДЕЛЬТА ПРАВИЛО 7](#_Toc153843118)

[2.1 Постановка задачи 7](#_Toc153843119)

[2.2 Теоретический раздел 7](#_Toc153843120)

[2.3 Программная реализация 8](#_Toc153843121)

[3 ОБРАТНОЕ РАСПРОСТРАНЕНИЕ ОШИБКИ 9](#_Toc153843122)

[3.1 Постановка задачи 9](#_Toc153843123)

[3.2 Теоретический раздел 9](#_Toc153843124)

[3.3 Программная реализация 9](#_Toc153843125)

[4 РАДИАЛЬНО-БАЗИСНЫЕ ФУНКЦИИ 11](#_Toc153843126)

[4.1 Постановка задачи 11](#_Toc153843127)

[4.2 Теоретический раздел 11](#_Toc153843128)

[4.3 Программная реализация 11](#_Toc153843129)

[5 КАРТЫ КОХОНЕНА 13](#_Toc153843130)

[5.1 Постановка задачи 13](#_Toc153843131)

[5.2 Теоретический раздел 13](#_Toc153843132)

[5.3 Программная реализация 13](#_Toc153843133)

[6 ВСТРЕЧНОЕ РАСПРОСТРАНЕНИЕ 15](#_Toc153843134)

[6.1 Постановка задачи 15](#_Toc153843135)

[6.2 Теоретический раздел 15](#_Toc153843136)

[6.3 Программная реализация 15](#_Toc153843137)

[7 РЕКУРЕНТНЫЕ СЕТИ 16](#_Toc153843138)

[7.1 Постановка задачи 16](#_Toc153843139)

[7.2 Теоретический раздел 16](#_Toc153843140)

[7.3 Программная реализация 16](#_Toc153843141)

[8 СВЕРТОЧНЫЕ СЕТИ 18](#_Toc153843142)

[8.1 Постановка задачи 18](#_Toc153843143)

[8.2 Теоретический раздел 18](#_Toc153843144)

[8.3 Программная реализация 19](#_Toc153843145)

[ЗАКЛЮЧЕНИЕ 23](#_Toc153843146)

[ПРИЛОЖЕНИЯ 24](#_Toc153843147)

# ВВЕДЕНИЕ

В настоящее время нейронные сети представляют собой перспективное направление в исследованиях машинного обучения. Их уникальная способность анализа и обработки сложных данных позволяет решать разнообразные задачи, начиная от распознавания образов и классификации данных до выявления паттернов и прогнозирования трендов.

В рамках данной работы проведены исследования в области проектирования и обучения нейронных сетей, включающие основные аспекты создания архитектуры нейронной сети, выбор оптимальных гиперпараметров, подготовку данных для обучения, а также использование методов обучения и оценки результатов.

Основной целью нашей работы является предоставление полного и понятного описания процесса проектирования и обучения нейронных сетей, а также подробных результатов и выводов. Путем анализа различных моделей и методов обучения мы стремились достичь максимального качества предсказания и эффективности работы нейронной сети.

В нашей работе представлено детальное описание каждого этапа процесса, начиная от выбора архитектуры сети и определения оптимальных гиперпараметров, заканчивая обработкой и предварительной обработкой данных. Кроме того, мы предоставляем результаты экспериментов, проведенных для оценки производительности различных моделей и методов обучения.

Начало формы

# 1 ОБУЧЕНИЕ ПО ПРАВИЛАМ ХЕББА

## 1.1 Постановка задачи

Необходимо построить алгоритм обучения нейронных сетей по правилам Хебба.

## 1.2 Теоретический раздел

Метод Хебба является одним из самых простых и понятных методов обучения нейронных сетей. Он основан на принципе, что если два нейрона активируются одновременно, то связь между ними укрепляется. Этот принцип называется правилом Хебба.

Правило Хебба гласит, что если нейрон А активируется одновременно с нейроном В, то вес связи между ними увеличивается. Если же нейрон А активируется в отсутствие нейрона В, то вес связи между ними остается неизменным. Если же нейрон В активируется в отсутствие нейрона А, то вес связи между ними уменьшается.

Таким образом, метод Хебба позволяет настраивать веса связей между нейронами на основе опыта и обучения. Например, если мы хотим обучить нейронную сеть распознавать образы, то мы можем использовать метод Хебба для того, чтобы укреплять связи между нейронами, которые активируются одновременно при предъявлении одного и того же образа.

Метод Хебба также может быть использован для обучения нейронных сетей на различных задачах машинного обучения, таких как классификация, кластеризация, ассоциативная память и т.д. Он может быть применен как для однослойных, так и для многослойных нейронных сетей.

Однако, метод Хебба имеет некоторые ограничения. Например, он не учитывает отрицательные взаимодействия между нейронами и не учитывает динамические свойства нейронных сетей. Поэтому, в более сложных задачах машинного обучения, метод Хебба может быть использован только как один из шагов в процессе обучения нейронной сети.

В целом, метод Хебба является важным инструментом в обучении нейронных сетей и может быть использован для решения многих задач машинного обучения.

## 1.3 Программная реализация

Был реализован код на языке Python обучения по правилам Хебба для расчета операции логического "ИЛИ" при значениях: -1 - Ложь, 1 - Истина.

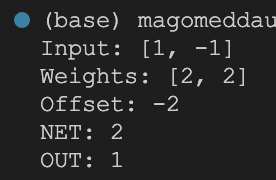


Рис. 1.1 – Вывод работы программы

Код реализации представлен в приложении A.

# 2 ДЕЛЬТА ПРАВИЛО

## 2.1 Постановка задачи

Необходимо построить алгоритм обучения нейронных сетей по дельта-правилу.

## 2.2 Теоретический раздел

Дельта-правило является методом обучения нейронных сетей, который основан на правиле Хебба. Правило Хебба утверждает, что если два нейрона активируются одновременно, то связь между ними усиливается. Данный принцип был использован для создания дельта-правила, который позволяет настраивать веса связей между нейронами на основе разницы между желаемым и фактическим выходом нейронной сети.

Дельта-правило работает следующим образом: сначала нейронная сеть получает входные данные и вычисляет выходные значения для каждого нейрона. Затем сравниваются полученные значения с желаемыми значениями и вычисляется ошибка, которая представляет собой разницу между желаемым и фактическим выходом. Далее веса связей между нейронами корректируются на основе этой ошибки.

Корректировка весов связей происходит по формуле (2.1):

(2.1),

где w(i,j) – вес связи между i-м и j-м нейронами, α – коэффициент обучения, δ(j) – ошибка выходного значения j-го нейрона, x(i) – входное значение i-го нейрона.

Таким образом, если ошибка выходного значения большая, то вес связи усиливается, а если ошибка мала, то вес связи ослабевает. Это позволяет нейронной сети быстро настраиваться на новые данные и достигать высокой точности работы.

Дельта-правило широко используется в обучении нейронных сетей, особенно в задачах классификации и распознавания образов. Однако, он имеет недостатки, такие как склонность к переобучению и проблемы с выбором оптимального значения коэффициента обучения. Для решения этих проблем используются другие методы обучения, такие как обратное распространение ошибки и генетические алгоритмы.

## 2.3 Программная реализация

Было реализовано обучение по дельта-правилу на языке Python.

Код реализации представлен в приложении C.

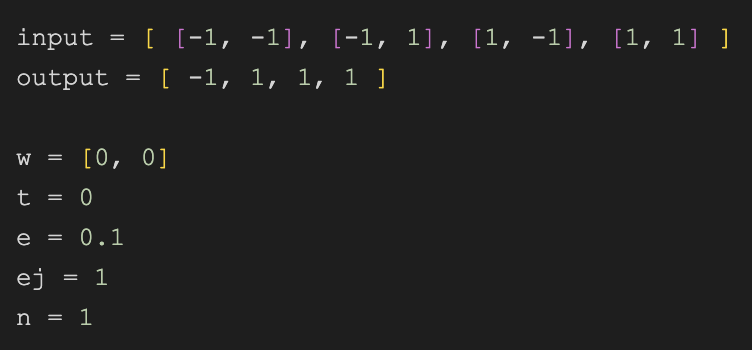


Рис. 2.1 – Входные данные

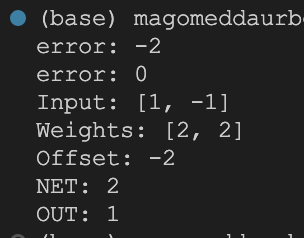


Рис. 2.2 – Вывод работы программы

# 3 ОБРАТНОЕ РАСПРОСТРАНЕНИЕ ОШИБКИ

## 3.1 Постановка задачи

Необходимо обучить нейронную сеть по правилу обратного распространения ошибки.

## 3.2 Теоретический раздел

Метод обратного распространения ошибки является итеративным процессом, который заключается в последовательном вычислении ошибки на выходе нейронной сети и корректировке весов связей между нейронами. Ошибка на выходе сравнивается с ожидаемым результатом, и на основе этого вычисляется градиент функции ошибки по весам связей. Градиент показывает, как изменение весов связей повлияет на значение ошибки. Затем веса связей корректируются в направлении, противоположном градиенту, чтобы минимизировать ошибку.

Обучение методом обратного распространения ошибки требует большого количества данных для обучения нейронной сети. Чем больше данных, тем точнее будет работать нейронная сеть. Кроме того, для успешного обучения нейронной сети необходимо правильно выбрать параметры обучения, такие как скорость обучения и количество эпох обучения.

Метод обратного распространения ошибки является одним из наиболее распространенных методов обучения нейронных сетей, и его использование позволяет достигать высокой точности в широком спектре задач, таких как распознавание образов, прогнозирование временных рядов, классификация текстов и многие другие. Однако, этот метод имеет свои ограничения и не всегда может быть эффективным для всех типов задач и данных.

## 3.3 Программная реализация

Реализован код обратного распространения ошибки на языке Python.

Код реализации представлен в приложении B.

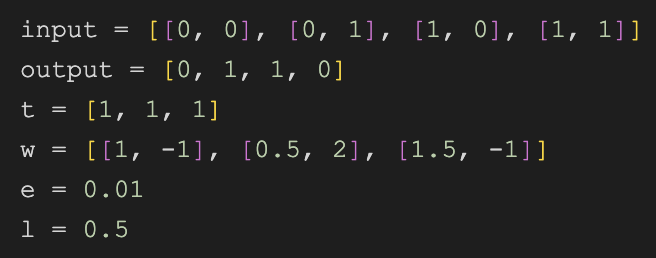


Рис. 3.2 – Входные данные

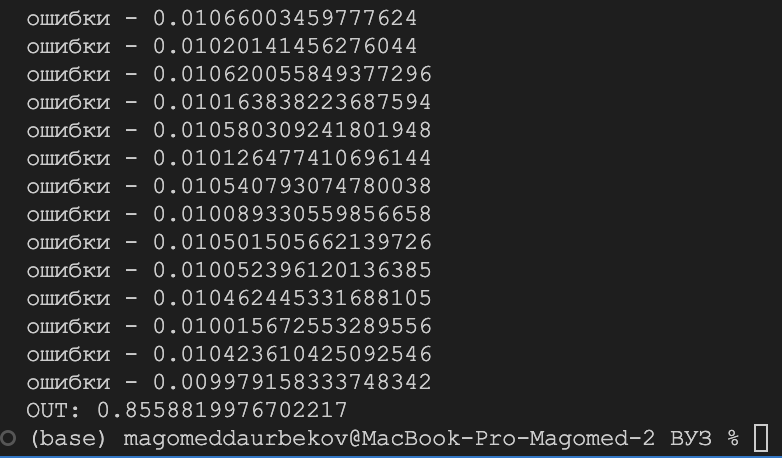


Рис. 3.3 – Вывод работы программы

# 4 РАДИАЛЬНО-БАЗИСНЫЕ ФУНКЦИИ

## 4.1 Постановка задачи

Необходимо обучить нейронную сеть по правилу радиально-базисной функции.

## 4.2 Теоретический раздел

Радиальная базисная функция (РБФ) — функция из набора однотипных радиальных функций, используемых как функция активации в одном слое искусственной нейронной сети или как-либо ещё, в зависимости от контекста. Радиальная функция— это любая вещественная функция, значение которой зависит только от расстояния до начала координатϕ(x)=ϕ(‖x‖) или от расстояния между некоторой другой точкойc, называемой *центром*ϕ(x,c)=ϕ(‖x−c‖). В качестве нормы обычно выступает евклидово расстояние, хотя можно использовать и другие метрики.

Линейные комбинации радиальных базисных функций также можно использовать для аппроксимации заданной функции. Аппроксимация может быть интерпретирована как простейшая разновидность нейронной сети

## 4.3 Программная реализация

Реализован код радиально-базисной функции на языке Python.

Код реализации представлен в приложении D.

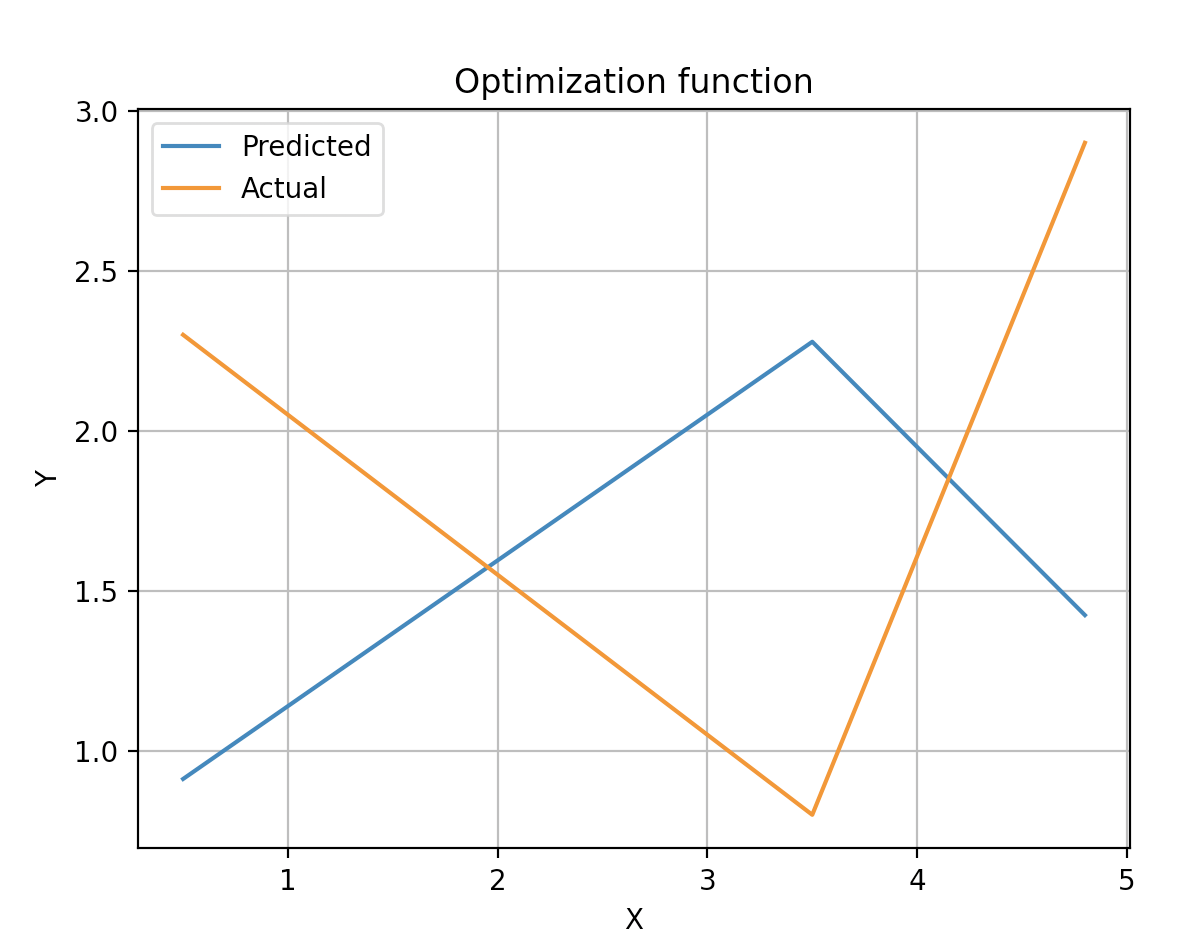


Рис. 4.1– График истинных и предсказанных значений



Рис. 4.2 – Вывод работы программы

# 5 КАРТЫ КОХОНЕНА

## 5.1 Постановка задачи

Необходимо изучить правило карт Кохонена.

## 5.2 Теоретический раздел

Карты Кохонена, или самоорганизующиеся карты (SOM), или самоорганизующиеся карты характеристик (SOFM) — этот метод машинного обучения без учителя, используемый для создания пространства с низкой размерностью (обычно двумерного) для представления многомерного набора данных с сохранением топологической структуры данных. Например, имеем набор данных из ***p*** переменных, измеряемых в *n* наблюдениях — их можно представить в виде кластеров наблюдений с одинаковыми значениями переменных. Затем эти кластеры можно представить в виде двухмерной карты, при этом наблюдения в близких кластерах будут иметь более схожие значения, чем в отдаленных. Это может упростить визуализацию и анализ многомерных данных.

## 5.3 Программная реализация

Реализован код правила карт Кохонена на языке Python.

Код реализации представлен в приложении E.

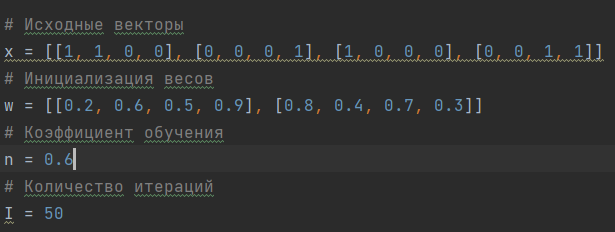


Рис. 5.1 – Входные данные

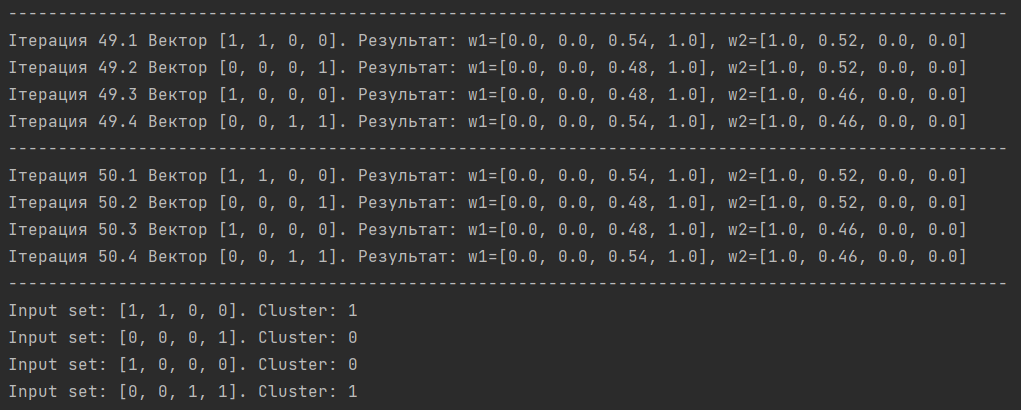


Рис. 5.2 – Вывод работы программы

# 6 ВСТРЕЧНОЕ РАСПРОСТРАНЕНИЕ

## 6.1 Постановка задачи

Необходимо обучить нейронную сеть по правилу встречного распространения.

## 6.2 Теоретический раздел

Применение нейронной сети к данным (вычисление выхода по заданному входу) часто называют прямым проходом, или же forward propagation (forward pass). На этом этапе происходит преобразование исходного представления данных в целевое и последовательно строятся промежуточные (внутренние) представления данных — результаты применения слоёв к предыдущим представлениям. Именно поэтому проход называют прямым.

## 6.3 Программная реализация

Реализован код правила встречного распространения на языке Python.

Код реализации представлен в приложении F.

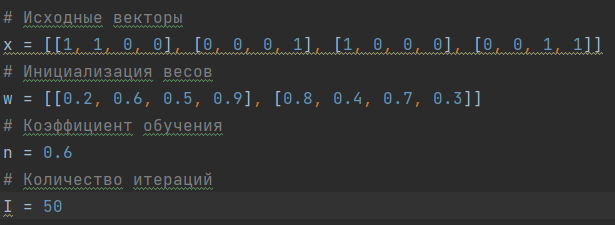


Рис. 6.1 – Входные данные

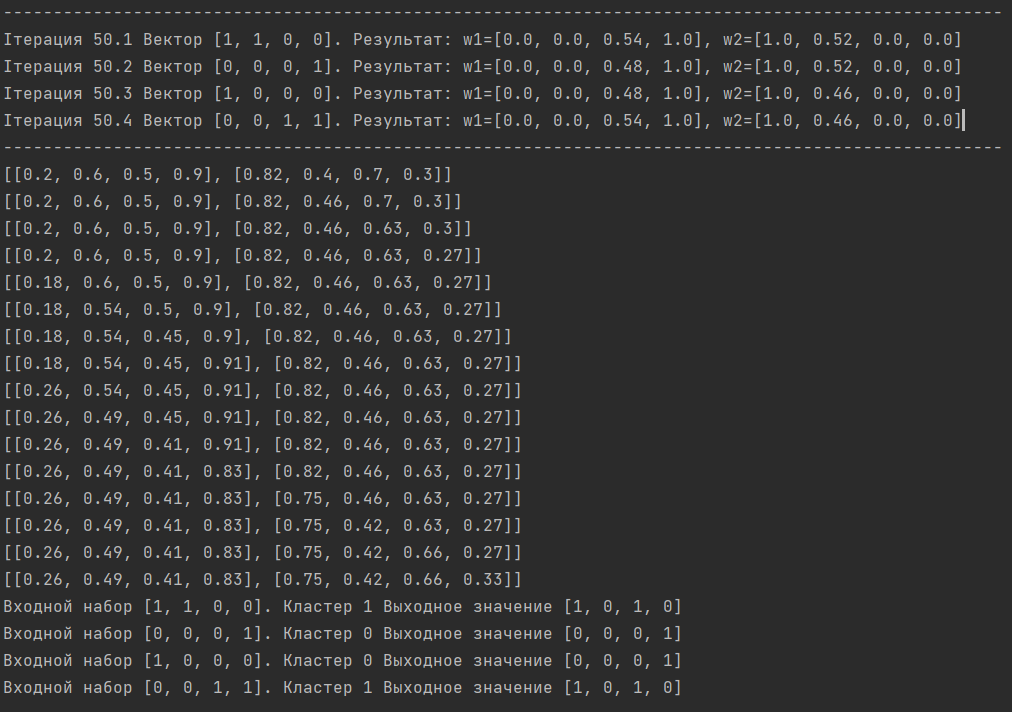


Рис. 6.2 – Вывод работы программы

# 7 РЕКУРЕНТНЫЕ СЕТИ

## 7.1 Постановка задачи

Необходимо обучить рекуррентную нейронную сеть.

## 7.2 Теоретический раздел

Рекуррентная нейронная сеть (RNN) — это тип искусственных нейронных сетей, способных учитывать последовательность данных и использовать информацию из предыдущих шагов для принятия решений на текущем шаге. Она отличается от других типов нейронных сетей, таких как сверточные нейронные сети (CNN), которые не имеют такой способности к запоминанию предыдущих шагов.

Обучение рекуррентной нейронной сети является процессом настройки параметров модели с использованием набора обучающих данных. Основная задача обучения RNN состоит в том, чтобы научить сеть предсказывать следующий элемент в последовательности данных на основе предшествующих элементов.

Для обучения RNN используется метод обратного распространения ошибки, который позволяет определить, насколько модель ошибается в своих предсказаниях, и корректировать параметры сети для уменьшения этой ошибки. Этот процесс продолжается до тех пор, пока сеть не достигнет определенного уровня точности или не будет удовлетворять поставленным требованиям.

Одно из самых распространенных применений обучения RNN - это задачи предсказания временных рядов, таких как прогнозирование погоды, финансовых данных или акций. RNN также широко применяются в областях обработки естественного языка, машинного перевода и анализе текста.

В итоге, благодаря своей способности к запоминанию предыдущих шагов, обучение рекуррентной нейронной сети является мощным инструментом для моделирования последовательных данных и решения широкого спектра задач.

## 7.3 Программная реализация

Реализован код обучения рекуррентной нейронной сети на языке Python.

Код реализации представлен в приложении G.

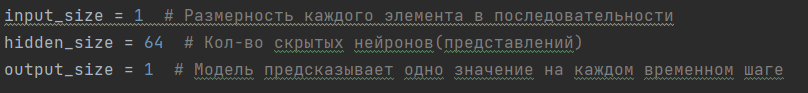


Рис. 7.1 – Входные данные

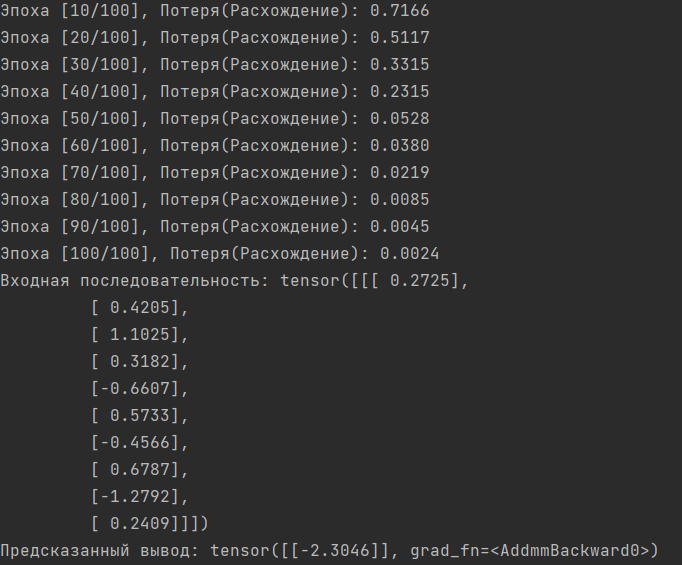


Рис. 7.2 – Вывод работы программы

# 8 СВЕРТОЧНЫЕ СЕТИ

## 8.1 Постановка задачи

Необходимо обучить сверточную нейронную сеть.

## 8.2 Теоретический раздел

Сверточная нейронная сеть (СНС) является одним из наиболее популярных и успешных подходов в области машинного обучения, особенно в обработке и анализе изображений. Она была разработана специально для работы с входными данными, имеющими пространственную структуру, например, изображениями.

Одним из ключевых преимуществ сверточных нейронных сетей является их способность автоматически извлекать иерархические признаки из сложных данных, благодаря чему они способны достичь высокой точности в распознавании и классификации изображений. Это достигается путем применения операций свертки и пулинга, которые позволяют нейронной сети «сканировать» изображение, и, в процессе обработки, извлекать информацию о небольших участках изображения.

Для успешного обучения сверточной нейронной сети необходимо создать достаточно большую и разнообразную обучающую выборку и выбрать подходящую архитектуру модели. Кроме того, важным этапом является оптимизация гиперпараметров и настройка алгоритма оптимизации для более эффективного обучения.

Обучение сверточной нейронной сети может занимать большое количество времени и вычислительных ресурсов, особенно при работе с большими объемами данных или сложными моделями. Однако, развитие вычислительной техники и появление специализированных аппаратных решений, таких как графические процессоры (GPU) или тензорные процессоры (TPU), значительно ускоряют процесс обучения.

В целом, сверточные нейронные сети представляют собой мощный инструмент для работы с изображениями, их распознавания и анализа. Их применение охватывает множество областей, включая компьютерное зрение, медицину, автоматическое распознавание лиц, автоматическую классификацию и многие другие.

## 8.3 Программная реализация

Код реализации представлен в приложении H.

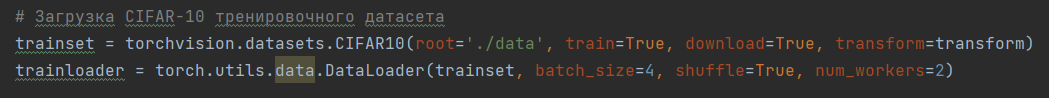


Рис. 8. 1 – Входные данные

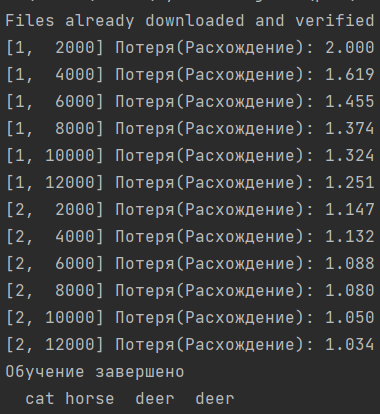


Рис. 8.2 – Результат работы программы

Изображение выглядит как текст, снимок экрана, прямоугольный, искусство

Автоматически созданное описание

Рис. 8.4 – Картинки

# ЗАКЛЮЧЕНИЕ

Заканчивая курс "Проектирование и обучение нейронных сетей", необходимо подчеркнуть его высокую значимость в контексте современных научных и технологических достижений. В ходе обучения были тщательно освоены основные принципы функционирования и проектирования нейронных сетей, а также изучены разнообразные методы и алгоритмы обучения.

Полученные знания стали ключом к пониманию принципов применения нейронных сетей в различных областях, таких как компьютерное зрение, обработка естественного языка, робототехника и многие другие. В процессе обучения мы ознакомились с актуальными тенденциями и инструментами, такими как глубокое обучение и фреймворки TensorFlow и PyTorch, что расширило наши возможности в проектировании и обучении нейронных сетей.

Курс также предоставил практические навыки в обработке данных, создании моделей нейронных сетей и их эффективном обучении. Приобретенный опыт работы с реальными данными и использование современных методов и подходов в области машинного обучения стали неоценимым ресурсом.

В целом, завершение курса "Проектирование и обучение нейронных сетей" не только снабдило нас фундаментальными знаниями и навыками, но и предоставило ценный инструментарий для успешной работы в быстро меняющемся мире технологий и науки. Эти глубокие познания и умения окажутся весьма полезными в нашей дальнейшей профессиональной деятельности и исследовательской работе в области машинного обучения.

# ПРИЛОЖЕНИЯ

Приложение A - Код обучения нейронной сети по правилам Хебба;

Приложение B – Код обучения нейронной сети по дельта-правилу;

Приложение C – Код обучения нейронной сети по правилам обратного распространения ошибки;

Приложение D – Код обучения нейронной сети по правилам радиально-базисных функций;

Приложение E – Код обучение нейронной сети по правилам карт Кохонена;

Приложение F – Код обучение нейронной сети по правилам встречного распространения;

Приложение G – Код обучения нейронной сети RNN (рекуррентные сети);

Приложение H – Код обучения нейронной сети CNN (сверточные сети).

**Приложение A**

Листинг A.1 – Код обучения нейронной сети по правилам Хебба

|  |
| --- |
| def OUT(NET):  return 1 if NET >= 0 else 0  input = [ [-1, -1], [-1, 1], [1, -1], [1, 1] ]  output = [ -1, 1, 1, 1 ]  w = [0, 0]  t = 0  for i in range(len(input)):  w[0] += input[i][0] \* output[i]  w[1] += input[i][1] \* output[i]  t -= output[i]  x = [1, -1]  NET = x[0] \* w[0] + x[1] \* w[1] - t  print(f'Input: {x}')  print(f'Weights: {w}')  print(f'Offset: {t}')  print(f'NET: {NET}')  print(f'OUT: {OUT(NET)}') |

**Приложение B**

Листинг B.1 – *Код обучения нейронной сети по правилам обратного распространения ошибки*

|  |
| --- |
| from math import \*  #Входные значения  input = [[0, 0], [0, 1], [1, 0], [1, 1]]  output = [0, 1, 1, 0]  t = [1, 1, 1]  w = [[1, -1], [0.5, 2], [1.5, -1]]  e = 0.01  l = 0.5  errors = [0, 0, 0]  #Среднеквадратичная ошибка  def MSE(d, y):  return 0.5 \* (d - y) \*\* 2  #Активация  def activate(NET):  return 1 / (1 + e \*\* (-1 \* NET))  #Деактивация  def Dactivate(OUT):  return OUT \* (1 - OUT)  #Изменение веса  def change\_weight(w, l, e, input):  return w +(l \* e \* activate(input))  #Реальный алгортим обучения(жесть)  weight1 = 0  weight2 = 0  weight = 0  E = 1  i = 0  while E > e:  if i > len(input)-1:  i = 0  weight1 = input[i][0] \* w[0][0] + 1 \* t[0] + input[i][1] \* w[1][0]  weight1 = activate(weight1)  weight2 = input[i][0] \* w[0][1] + 1 \* t[1] + input[i][1] \* w[1][1]  weight2 = activate(weight2)  weight = weight1 \* w[2][0] + 1 \* t[2] + weight2 \* w[2][1]  weight = activate(weight)  E = MSE(output[i], weight)  print(f"ошибка - {E}")  errors[0] = Dactivate(weight) \* (output[i] - weight)  errors[1] = Dactivate(weight1) \* (w[2][0] \* errors[0])  errors[2] = Dactivate(weight2) \* (w[2][1] \* errors[0])  w[2][0] = w[2][0] + (l \* errors[0] \* weight1)  w[2][1] = w[2][1] + (l \* errors[0] \* weight2)  t[2] = t[2] + (l \* errors[0] \* activate(1))  for j in range(1, -1, -1):  for u in range(1, -1, -1):  w[j][u] = change\_weight(w[j][u], l, errors[2] if j==1 else errors[1], input[i][u])  t[1] = t[1] + (l \* errors[2] \* activate(1))  t[0] = t[0] + (l \* errors[1] \* activate(1))  i=i+1  x = [0, 0]  y = 0  weight1 = input[i][0] \* w[0][0] + t[0] + input[i][1] \* w[1][0]  weight1 = activate(weight1)  weight2 = input[i][0] \* w[0][1] + t[1] + input[i][1] \* w[1][1]  weight2 = activate(weight2)  weight = weight1 \* w[2][0] + t[2] + weight2 \* w[2][1]  weight = activate(weight)  print(f'OUT: {weight}') |

**Приложение C**

*Листинг C.1* – *Код обучения нейронной сети по дельта правилу*

|  |
| --- |
| def OUT(NET):  return 1 if NET >= 0 else -1  input = [ [-1, -1], [-1, 1], [1, -1], [1, 1] ]  output = [ -1, 1, 1, 1 ]  w = [0, 0]  t = 0  e = 0.1  ej = 1  n = 1  i = 0  while abs(ej) >= e:  d = OUT(input[i][0] \* w[0] + input[i][1] \* w[1] - t)  ej = output[i] - d  w[0] += n \* ej \* input[i][0]  w[1] += n \* ej \* input[i][1]  t += n \* ej  print(f'error: {ej}')  i = (i + 1) % len(input)  x = [1, -1]  NET = x[0] \* w[0] + x[1] \* w[1] - t  print(f'Input: {x}')  print(f'Weights: {w}')  print(f'Offset: {t}')  print(f'NET: {NET}')  print(f'OUT: {OUT(NET)}') |

**Приложение D**

Листинг D.1 – Код обучения нейронной сети по правилам радиально-базисных функций

|  |
| --- |
| import numpy as np  import sys  import matplotlib.pyplot as plt  # Точки (x, y) для обучения и начальные веса w  inp = [[0.9, 1], [2.1, 1.9], [3.1, 3]]  w = [0, 0, 0]  # Функция РБФ  def h(x, n):  return np.exp(((x - n) \*\* 2) / (-2)) #h1(x)=exp(-((x-a1)^2)/2)  # Функция для предсказания значений на основе весов и РБФ  def f(w, n, x):  return np.dot(w, [h(x, n\_i) for n\_i in n])  # Функция для вычисления центров интервалов по x  def center(k = inp):  return [(k[j][0] + k[j + 1][0]) / 2 for j in range(len(k) - 1)]  # Вычисление матрицы u и её транспонирование  a = center()  u = np.asarray([[h(inp[j][0], a[i]) for j in range(len(inp))] for i in range(len(a))])  j = u.dot(u.transpose())  # Вычисление обратной матрицы  rev = np.linalg.inv(j)  # Умножение обратной матрицы на матрицу u  summ = rev.dot(u)  # Формирование вектора Y из исходных значений y  Y = np.asarray([inp[i][1] for i in range(len(inp))])  # Умножение на вектор Y  summ = summ.dot(Y)  # Формирование векторов x, y и yb  x = [inp[k][0] for k in range(len(inp))]  y = [f(summ, a, inp[k][0]) for k in range(len(inp))]  yb = [inp[k][1] for k in range(len(inp))] |

Завершение листинга D

print(x, y, yb)

plt.plot(x, y, label='Прогнозируемые значения')

plt.plot(x, yb, label='Истинные значения')

plt.title('Функция оптимизации')

plt.xlabel('X')

plt.ylabel('Y')

plt.legend()

plt.grid(True)

plt.show()

**Приложение E**

Листинг E.1 – Код обучения нейронной сети по правилам карт Кохонена

|  |
| --- |
| import numpy as np  # Функция вычисления расстояния между векторами X и W  def distance(X, W):  if len(X) != len(W):  return -1  else:  d = np.sqrt(np.sum((np.array(X) - np.array(W)) \*\* 2))  return d  # Функция обновления весов нейрона  def changeWeightNeuron(n, X, W):  temp = [W[i] + n \* (X[i] - W[i]) for i in range(len(X))]  return temp  # Исходные векторы  x = [[1, 1, 0, 0], [0, 0, 0, 1], [1, 0, 0, 0], [0, 0, 1, 1]]  # Инициализация весов  w = [[0.2, 0.6, 0.5, 0.9], [0.8, 0.4, 0.7, 0.3]]  # Коэффициент обучения  n = 0.6  # Количество итераций  I = 50  d = []  number = 0  # Цикл по итерациям  for l in range(I):  # Цикл по входным векторам  for i in x:  # Вычисление расстояний от входного вектора до каждого нейрона  for j in w:  d.append(distance(i, j) if distance(i, j) != -1 else print("error"))  # Находим индекс минимального расстояния  mins = d.index(min(d))  # Обновляем веса нейрона  w[mins] = changeWeightNeuron(n, i, w[mins])  d = [] # Очищаем список расстояний  number = number + 1  # Округляем значения весов до сотых  w\_rounded = [[round(val, 2) for val in row] for row in w]  # Вывод результатов для каждой итерации и входного вектора  print(f"Iтерация {l+1}.{number} Вектор {i}. Результат: w1={w\_rounded[0]}, w2={w\_rounded[1]}")  number = 0  n = n - 0.01 # Уменьшаем коэффициент обучения |

Завершение листинга Е

print("-" \* 100) # Разделитель между итерациями

# Функция вычисления весов

def getWeight(x):

c = 0

for i in range(len(x)):

for j in w:

c += x[i] \* j[i]

return c

# Функция классификации

def clast(c):

return int(c)

for i in x:

print(f"Input set: {i}. Cluster: {clast(getWeight(i))} ")

**Приложение F**

Листинг F.1 – Код обучения нейронной сети по правилам встречного распространения

|  |
| --- |
| import numpy as np  from typing import ValuesView  # Функция вычисления расстояния между векторами X и W  def distance(X, W):  if len(X) != len(W):  return -1  else:  d = np.sqrt(np.sum((np.array(X) - np.array(W)) \*\* 2))  return d  # Функция обновления весов нейрона  def changeWeightNeuron(n, X, W):  temp = [W[i] + n \* (X[i] - W[i]) for i in range(len(X))]  return temp  # Исходные векторы  x = [[1, 1, 0, 0], [0, 0, 0, 1], [1, 0, 0, 0], [0, 0, 1, 1]]  # Инициализация весов  w = [[0.2, 0.6, 0.5, 0.9], [0.8, 0.4, 0.7, 0.3]]  wg = [[0.2, 0.6, 0.5, 0.9], [0.8, 0.4, 0.7, 0.3]]  # Коэффициент обучения  n = 0.6  # Количество итераций  I = 50  d = []  number = 0  # Цикл по итерациям  for l in range(I):  # Цикл по входным векторам  for i in x:  # Вычисление расстояний от входного вектора до каждого нейрона  for j in w:  d.append(distance(i, j) if distance(i, j) != -1 else print("error"))  # Находим индекс минимального расстояния  mins = d.index(min(d))  # Обновляем веса нейрона  w[mins] = changeWeightNeuron(n, i, w[mins])  d = [] # Очищаем список расстояний  number = number + 1  # Округляем значения весов до сотых  w\_rounded = [[round(val, 2) for val in row] for row in w]  # Вывод результатов для каждой итерации и входного вектора |

Завершение листинга F

print(f"Iтерация {l+1}.{number} Вектор {i}. Результат: w1={w\_rounded[0]}, w2={w\_rounded[1]}")

number = 0

n = n - 0.01 # Уменьшаем коэффициент обучения

print("-" \* 100) # Разделитель между итерациями

# Функция вычисления весов

def getWeight(x):

c = 0

for i in range(len(x)):

for j in w:

c += x[i] \* j[i]

return c

# Функция классификации

def clast(c):

return int(c)

x = [[1, 1, 0, 0], [0, 0, 0, 1], [1, 0, 0, 0], [0, 0, 1, 1]]

b = 0.1

for i in range(len(x)):

j = clast(getWeight(x[i]))

# Обновление весов для текущего кластера

for r in range(len(wg[j])):

wg[j][r] = wg[j][r] + b \* (x[i][r] - wg[j][r])

b -= 0.001

wg\_rounded = [[round(val, 2) for val in row] for row in wg]

print(wg\_rounded)

# Проход по исходным входным векторам

for i in range(len(x)):

v = [None] \* len(x[i])

f = clast(getWeight(x[i]))

# Округление значений весов до целых

for g in range(len(v)):

v[g] = round(wg[f][g])

# Вывод результатов

print(f"Входной набор {x[i]}. Кластер {clast(getWeight(x[i]))} Выходное значение {v}")

**Приложение G**

Листинг G.1 – Код обучения рекуррентной нейронной сети

|  |
| --- |
| import torch  import torch.nn as nn  import torch.optim as optim  '''PyTorch — это фреймворк для машинного обучения и глубокого обучения,  который предоставляет гибкие инструменты для разработки и обучения моделей  и исследования в области искусственного интеллекта. '''  # Определение простой RNN  class SimpleRNN(nn.Module):  def \_\_init\_\_(self, input\_size, hidden\_size, output\_size):  super(SimpleRNN, self).\_\_init\_\_()  self.hidden\_size = hidden\_size  self.rnn = nn.RNN(input\_size, hidden\_size, batch\_first=True) # Реккурентный слой - nn.rnn  self.fc = nn.Linear(hidden\_size, output\_size) # Линейный слой - nn.linear  def forward(self, x):  h0 = torch.zeros(1, x.size(0), self.hidden\_size) # Начальное состояние скрытого слоя  out, \_ = self.rnn(x, h0)  out = self.fc(out[:, -1, :]) # Получаем вывод только из последнего временного шага  return out  input\_size = 1 # Размерность каждого элемента в последовательности  hidden\_size = 64 # Кол-во скрытых нейронов(представлений)  output\_size = 1 # Модель предсказывает одно значение на каждом временном шаге  model = SimpleRNN(input\_size, hidden\_size, output\_size)  criterion = nn.MSELoss() # Функция потерь: среднеквадратичная ошибка  '''Loss - метрика, которая измеряет расхождение между предсказанными значениями модели  и фактическими (истинными) значениями. Это количественная мера того,  насколько хорошо модель выполняет задачу обучения на данном этапе.'''  optimizer = optim.Adam(model.parameters(), lr=0.001) # Оптимизатор: Adam  '''Adam (Adaptive Moment Estimation) - это метод оптимизации, предназначенный для эффективного обновления весов  в процессе обучения нейронных сетей. Он является комбинацией методов стохастического градиентного спуска (SGD) |

Завершение листинга G

и адаптивного градиентного метода (RMSprop). Adam автоматически регулирует шаг обучения для каждого параметра

на основе наблюдаемых статистик градиентов.'''

# Генерация примера входных данных

sequence\_length = 10

batch\_size = 8

# Создаем случайную последовательность чисел

input\_data = torch.randn(batch\_size, sequence\_length, input\_size)

# Создаем соответствующие метки (предположим, что мы пытаемся предсказать следующее число)

target\_data = torch.randn(batch\_size, output\_size)

# Обучение модели на несколько эпох. Эпоха - 1 проход через через весь тренировочный набор данных

epochs = 100

for epoch in range(epochs):

optimizer.zero\_grad() # Обнуляем градиенты

output = model(input\_data) # Получаем вывод модели

loss = criterion(output, target\_data) # Вычисляем loss

loss.backward() # Обратное распространение ошибки и оптимизация

optimizer.step()

# Выводим loss каждые 10 эпох

if (epoch + 1) % 10 == 0:

print(f'Эпоха [{epoch+1}/{epochs}], Потеря(Расхождение): {loss.item():.4f}')

# После обучения можно использовать модель для предсказания

# Например, для первого элемента в последовательности

input\_example = input\_data[0:1, :, :]

output\_example = model(input\_example)

print("Входная последовательность:", input\_example)

print("Предсказанный вывод:", output\_example)

'''Тензор (tensor) - это многомерный массив, который может содержать числовые значения

и представляет собой основную структуру данных в машинном обучении и глубоком обучении.

Тензоры могут иметь различное количество измерений (осей),

и они используются для представления данных в виде массивов.'''

**Приложение H**

Листинг H.1 – Код обучения сверточной нейронной сети

|  |
| --- |
| '''Этот код использует датасет CIFAR-10, который содержит изображения 32x32 пикселя с 10 классами.'''  import torch  import torch.nn as nn  import torch.optim as optim  import torchvision  import torchvision.transforms as transforms  import torch.nn.functional as F  import matplotlib.pyplot as plt  import numpy as np  if \_\_name\_\_ == '\_\_main\_\_':  import multiprocessing  multiprocessing.set\_start\_method('spawn', True)  # Загрузка данных  transform = transforms.Compose([  transforms.ToTensor(),  transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  ])  # Загрузка CIFAR-10 тренировочного датасета  trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)  trainloader = torch.utils.data.DataLoader(trainset, batch\_size=4, shuffle=True, num\_workers=2)  # Определение CNN  class SimpleCNN(nn.Module):  def \_\_init\_\_(self):  super(SimpleCNN, self).\_\_init\_\_()  # Первый сверточный слой: вход - 3 канала (цветное изображение), выход - 16 каналов, размер ядра - 3x3, с дополнением 1  self.conv1 = nn.Conv2d(3, 16, kernel\_size=3, padding=1)  # Пуллинг: максимальное объединение с окном 2x2  self.pool = nn.MaxPool2d(2, 2)  # Второй сверточный слой: вход - 16 каналов, выход - 32 канала, размер ядра - 3x3, с дополнением 1  self.conv2 = nn.Conv2d(16, 32, kernel\_size=3, padding=1)  # Полносвязный слой 1: вход - 32 \* 8 \* 8 (размер после сверточных слоев и пуллинга), выход - 64  self.fc1 = nn.Linear(32 \* 8 \* 8, 64)  # Полносвязный слой 2: вход - 64, выход - 10 (число классов в CIFAR-10)  self.fc2 = nn.Linear(64, 10) |

Продолжение листинга H.1

def forward(self, x):

# Применение первого сверточного слоя, активации ReLU и пуллинга

x = self.pool(F.relu(self.conv1(x)))

# Применение второго сверточного слоя, активации ReLU и пуллинга

x = self.pool(F.relu(self.conv2(x)))

# Решейпинг тензора перед подачей в полносвязные слои

x = x.view(-1, 32 \* 8 \* 8)

# Применение первого полносвязного слоя и активации ReLU

x = F.relu(self.fc1(x))

# Применение второго полносвязного слоя (без активации, так как это логиты для CrossEntropyLoss)

x = self.fc2(x)

return x

# Обучение сети

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

net = SimpleCNN().to(device)

criterion = nn.CrossEntropyLoss()

optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

for epoch in range(2): # Проход по набору данных дважды

running\_loss = 0.0

# Цикл по мини-пакетам внутри DataLoader

for i, data in enumerate(trainloader, 0):

inputs, labels = data

inputs, labels = inputs.to(device), labels.to(device)

# Обнуление градиентов перед каждой итерацией

optimizer.zero\_grad()

# Прямой проход: подача входных данных через сеть

outputs = net(inputs)

# Вычисление функции потерь между предсказаниями и метками

loss = criterion(outputs, labels)

# Обратный проход: вычисление градиентов

loss.backward()

# Обновление весов модели с использованием оптимизатора

optimizer.step()

# Отслеживание общей потери на протяжении эпохи

running\_loss += loss.item()

# Вывод промежуточной информации каждые 2000 мини-пакетов

if i % 2000 == 1999:

print('[%d, %5d] Потеря(Расхождение): %.3f' %

(epoch + 1, i + 1, running\_loss / 2000))

running\_loss = 0.0

Завершение листинга H

print('Обучение завершено')

'''После завершения обучения сеть используется для предсказания классов изображений'''

# Функция для отображения изображения

def imshow(img):

img = img / 2 + 0.5 # Денормализация

npimg = img.numpy()

plt.imshow(np.transpose(npimg, (1, 2, 0)))

plt.show()

# Получение случайного изображения из DataLoader

images, labels = next(iter(trainloader))

# Отображение изображения

imshow(torchvision.utils.make\_grid(images))

# Вывод меток

print(' '.join('%5s' % trainset.classes[labels[j]] for j in range(4)))