Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Рыбинский государственный авиационный технический университет имени П. А. Соловьева»

Факультет радиоэлектроники и информатики

Кафедра математического и программного обеспечения электронных вычислительных средств

**Курсовая работа**

По дисциплине

Нейрокомпьютерные системы

по теме

“Реализация нейронной сети Хэмминга”

Студент группы ИПБ-13 Ивашин А. В.

Преподаватель Паламарь И. Н.

Рыбинск 2017

Оглавление

[Задание 3](#_Toc474119737)

[Анализ задания 4](#_Toc474119738)

[Модель нейронной сети 4](#_Toc474119739)

[Разработка модели программы 5](#_Toc474119740)

[Разработка объектной модели 6](#_Toc474119741)

[Разработка алгоритмов 8](#_Toc474119742)

[Алгоритм обучения нейронной сети Хэмминга 8](#_Toc474119743)

[Кодирование программы 10](#_Toc474119744)

[Тестирование 14](#_Toc474119745)

[Вывод 16](#_Toc474119746)

[Список использованных источников 17](#_Toc474119747)

# Задание

Требуется написать программу, реализующую нейронную сеть Хэмминга для решения задач классификации бинарных входных векторов, и фильтрации зашумленного изображения от шумов.

# Анализ задания

## Модель нейронной сети

Нейронная сеть Хэмминга состоит из двух слоев, количество нейронов в которых равно количеству классов (эталонных образцов), число входов соответствует числу бинарных признаков, по которым различаются образы. Значения входных переменных принадлежат множеству {–1; 1}. Выходные значения подаются по обратным связям на входы нейронов второго слоя, в том числе свой собственный. Структурная схема сети изображена на рисунке 1.

Искусственная нейронная сеть Хэмминга используется для решения задач классификации бинарных входных векторов. В основе ее работы лежат процедуры, направленные на выбор в качестве решения задачи классификации одного из эталонных образов, наиболее близкого к поданному на вход сети зашумленному входному образу, и отнесение данного образа к соответствующему классу. Для оценки меры близости к каждому классу используется критерий, учитывающий расстояние Хэмминга – количество различающихся переменных у зашумленного и эталонного входных образов.

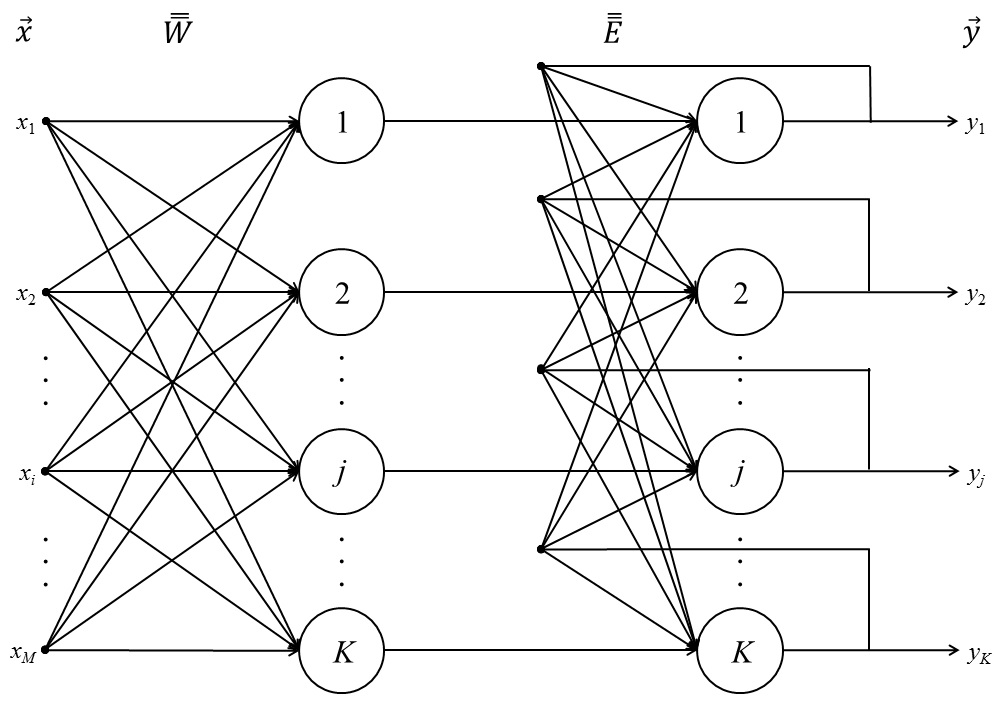


Рисунок 1 – Структурная схема сети Хэмминга

## Разработка модели программы

При работе с программой пользователь должен иметь возможность выполнить следующие операции:

1. Инициализировать сеть, задав количество нейронов и бинарных признаков;
2. Обучить нейронную сеть “распознаванию” (фильтрации) эталонного набора образов;
3. Сохранить результаты обучения (матрицу весов нейронной сети) в файл;
4. Загрузить матрицу весов нейронной сети из файла;
5. Произвести фильтрацию выбранного изображения и посмотреть результат работы сети в окне приложения.

Набор эталонных образов пользователь задает сам с помощью интерфейса. Ввод нового образа представляет собой сетку, в которой черными квадратами заполняются единицы, а белыми -1 (изначально сетка проинициализирована -1).

Матрицы весовых коэффициентов нейронной сети Хэмминга и весов обратных связей нейронной сети Хэмминга сохраняется и загружается из бинарного файла. Это позволяет сэкономить дисковое пространство и ускорить процесс сохранения и загрузки матриц.

# Разработка объектной модели

Нейронная сеть Хэмминга была представлена классом HemmingNet. Данный класс имеет следующие поля и методы:

1. Поля:
   1. referenceSamplesMatrix - данное поле хранит матрицу эталонных образцов.
   2. weightingCoefficientMatrix - данное поле хранит матрицу весовых коэффициентов нейронной сети Хэмминга.
   3. scalesFeedbackMatrix - поле для хранения матрицы весов обратных связей сети Хэмминга.
   4. numberNeurons - поле для хранения количества нейронов.
   5. numberBinarySigns - поле для хранения количества бинарных признаков.
   6. threshold - поле для хранения значения порога функции активации.
   7. weightInhibitorySynapse - поле для хранения значения ингибиторного синапса.
   8. errors - список для хранения значений ошибки на каждом шаге.
2. Методы:
   1. **public void** teachTheNetwork()- Метод выполняющий обучение нейронной сети (определяет порог, значение ингибиторного синопсиса и рассчитывающий матрицы весовых коэффициентов и весов обратных связей.
   2. **public** Integer runNet()- метод для запуска сети и нахождения номера образца, соответствующего заданному.
   3. И ряд различных геттеров и сеттеров, для обращения к полям класса.

Также в классе описан ряд приватных методов, необходимых для обеспечения функционирования нейронной сети:

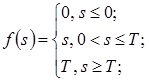
1. **private void** calculateWeightsMatrixNeuronsFirstLayer()- Метод для расчета матрицы весовых коэффициентов.
2. **private** Double calculateWeightsCoefficient (Integer coefficient) -метод для расчета весового коэффициента по заданному значение, он принимает значение и возвращает весовой коэффициент.
3. **private void** determineThreshold()- метод для определения порога.
4. **private void** determineWeightInhibitorySynapse()- Метод для определения значения ингибиторного синопсиса.
5. **private void** calculationScalesFeedbackMatrix()- Метод, для расчета матрицы обратных связей.
6. **private double** definesValueSynopsis(**int** j, **int** p)- Метод, определяющий значение для обратной связи.
7. **private** List<Double> calculateFirstState(List<Integer> inputVector)- метод для расчета значений первого состояния. Принимает изображение (бинарный вектор).
8. **private** List<Double> calculateOutputVector - метод для расчета выходного вектора для каждого вектора состояния.
9. **private double** activationFunction(**double** parameter) - метод реализующий функцию активации, принимает значение для которого и выполняет активацию.
10. **private double** calculationConditionStabilizationForOutputVector(List<Double> previousVector, List<Double> currentVector)- метод для расчета ошибки на каждом шаге. Принимает предыдущие значение вектора выходов и текущее значение вектора выходов.
11. **private** Integer getFindVector(List<Double> followingOutputVector) - метод для получение искомого вектора. Принимает вектор выходов сети, полученный на последнем шаге ее работы.
12. **private** List<Double> calculateFollowingState(List<Double> previousOutputVector)- рассчитать следующий вектор состояний по предыдущему вектору выходов. Принимает вектор выходов полученный на предыдущем шаге.

# Разработка алгоритмов

## Алгоритм обучения нейронной сети Хэмминга

Алгоритм жизненного цикла нейронной сети Хэмминга состоит из двух основных стадий: обучения и практического использования.

На стадии **обучения** выполняется следующая последовательность действий:

* + 1. Формируется матрица эталонных образов http://neuronus.com/images/theory/ins/05072015/01.jpg размера *K*x *M*.
    2. Рассчитывается матрица весовых коэффициентов нейронов первого слоя по формуле: http://neuronus.com/images/theory/ins/05072015/02.jpg
    3. Определяются настройки активационной функции: линейная пороговая функция:  и параметр http://neuronus.com/images/theory/ins/05072015/05.jpg.
    4. Задаются значения синапсов обратных связей нейронов второго слоя в виде элементов квадратной матрицы размера *K* x *K*:http://neuronus.com/images/theory/ins/05072015/06.jpg, гдеhttp://neuronus.com/images/theory/ins/05072015/07.jpg. Синапсы обратных связей нейронной сети Хэмминга, имеющие отрицательный вес, называются **ингибиторными**, или **тормозящими**.

На стадии **практического использования** выполняются следующие действия:

1. На входы сети подается неизвестный, в общем случае, зашумленный вектор сигналов http://neuronus.com/images/theory/ins/05072015/09.jpg.
2. Рассчитываются состояния и выходные значения нейронов первого слоя. Для расчета состояний нейронов используется соотношение:http://neuronus.com/images/theory/ins/05072015/10.jpg.
3. Выходам нейронов второго слоя в качестве начальных величин присваиваются значения выходов нейронов первого слоя, полученные на предыдущем шаге:http://neuronus.com/images/theory/ins/05072015/13.jpg. Далее первый слой нейронов на стадии практического использования больше не задействуется.
4. Для каждой итерации *q* рассчитываются новые значения состояний и выходов нейронов второго слоя. Состояния нейронов определяются по соотношению:http://neuronus.com/images/theory/ins/05072015/14.jpg. Новые выходные значения http://neuronus.com/images/theory/ins/05072015/16.jpg определяются в результате применения линейной пороговой активационной функции к соответствующим состояниям нейронов http://neuronus.com/images/theory/ins/05072015/17.jpg.
5. Цикл в п. 4 повторяется до стабилизации выходного вектора в соответствии с условием:http://neuronus.com/images/theory/ins/05072015/18.jpg.

В идеальном случае после стабилизации должен получиться выходной вектор с одним положительным и всеми остальными нулевыми элементами. Индекс единственного положительного элемента непосредственно указывает на класс неизвестного входного образа.

Если данные входного образа сильно зашумлены или в обучающей выборке отсутствовал подходящий эталон, в результате остановки цикла в п. 4 могут быть получены несколько положительных выходов, причем значение любого из них окажется меньше, чем *E*max. В этом случае делается заключение о невозможности отнесения входного образа к определенному классу, однако индексы положительных выходов указывают на наиболее схожие с ним эталоны.

# Кодирование программы

Ниже представлен код класса HemmingNet.

**public class** HemmingNet {  
 **private static double** *EPS* = 0.1;  
 **private** List<List<Integer>> **referenceSamplesMatrix**;  
 **private** List<List<Double>> **weightingCoefficientMatrix**;  
 **private** List<List<Double>> **scalesFeedbackMatrix**;  
 **private int numberNeurons**;  
 **private int numberBinarySigns**;  
 **private double threshold**;  
 **private double weightInhibitorySynapse**;  
 **private** List<Integer> **inputVector**;  
 **private** List<Double> **errors**;  
  
 **public** HemmingNet() {  
 **referenceSamplesMatrix** = **new** ArrayList<>();  
 **weightingCoefficientMatrix** = **new** ArrayList<>();  
 **scalesFeedbackMatrix** = **new** ArrayList<>();  
 **errors** = **new** ArrayList<>();  
 }  
  
 **private void** calculateWeightsMatrixNeuronsFirstLayer() {  
 List<Double> weightingCoefficient;  
 **for** (List<Integer> referenceSample : **referenceSamplesMatrix**) {  
 weightingCoefficient = **new** ArrayList<>();  
 weightingCoefficient.addAll(  
 referenceSample.stream()  
 .map(**this**::calculateWeightsCoefficient)  
 .collect(Collectors.*toList*())  
 );  
 **weightingCoefficientMatrix**.add(weightingCoefficient);  
 }  
 }  
  
 **private** Double calculateWeightsCoefficient(Integer coefficient) {  
 **return** 0.5D \* coefficient;  
 }  
  
 **private void** determineThreshold() {  
 **threshold** = **numberBinarySigns** / 2D;  
 }  
  
 **private void** determineWeightInhibitorySynapse() {  
 **weightInhibitorySynapse** = 1D / **numberNeurons**;  
 }  
  
 **private void** calculationScalesFeedbackMatrix() {  
 List<Double> scalesFeedback;  
 **for** (**int** i = 0; i < **numberNeurons**; i++) {  
 scalesFeedback = **new** ArrayList<>();  
 **for** (**int** j = 0; j < **numberNeurons**; j++) {  
 scalesFeedback.add(definesValueSynopsis(i, j));  
 }  
 **scalesFeedbackMatrix**.add(scalesFeedback);  
 }  
 }  
  
 **private double** definesValueSynopsis(**int** j, **int** p) {  
 **return** j == p ? 1.0 : -**weightInhibitorySynapse**;  
 }  
  
 **private** List<Double> calculateFirstState(List<Integer> inputVector) {  
 List<Double> result = **new** ArrayList<>();  
 **double** neuronState;  
 **for** (List<Double> weightingCoefficient : **weightingCoefficientMatrix**) {  
 neuronState = 0;  
 **for** (**int** i = 0; i < **numberBinarySigns**; i++) {  
 neuronState += weightingCoefficient.get(i) \* inputVector.get(i);  
 }  
 result.add(neuronState + **threshold**);  
 }  
 **return** result;  
 }  
  
 **private** List<Double> calculateOutputVector(List<Double> stateVector) {  
 **return** stateVector.stream().map(**this**::activationFunction).collect(Collectors.*toList*());  
 }  
  
 **private double** activationFunction(**double** parameter) {  
 **return** parameter > **threshold** ? **threshold** : parameter < 0D ? 0D : parameter;  
 }  
  
 **private double** calculationConditionStabilizationForOutputVector(List<Double> previousVector, List<Double> currentVector) {  
 **double** result = 0D;  
 **for** (**int** i = 0; i < **numberNeurons**; i++) {  
 result += Math.*pow*(currentVector.get(i) - previousVector.get(i), 2);  
 }  
 **return** result;  
 }  
  
 **public void** teachTheNetwork() {  
 *// обучаем сеть* determineThreshold(); *// считаем парог* determineWeightInhibitorySynapse(); *// опеределяем абсолютное значение веса каждого ингибиторного синапса* calculateWeightsMatrixNeuronsFirstLayer();  
 calculationScalesFeedbackMatrix();  
 }  
  
 **public void** setNumberNeurons(**int** numberNeurons) {  
 **this**.**numberNeurons** = numberNeurons;  
 }  
  
 **public void** setNumberBinarySigns(**int** numberBinarySigns) {  
 **this**.**numberBinarySigns** = numberBinarySigns;  
 }  
  
 **public void** addReferenceSample(List<Integer> sample) {  
 **referenceSamplesMatrix**.add(sample);  
 }  
  
 **public** List<List<Integer>> getReferenceSamplesMatrix() {  
 **return referenceSamplesMatrix**;  
 }  
  
 **public void** submitInputVector(List<Integer> inputVector) {  
 **this**.**inputVector** = inputVector;  
 }  
  
 **public** List<List<Double>> getWeightingCoefficientMatrix() {  
 **return weightingCoefficientMatrix**;  
 }  
  
 **public** List<List<Double>> getScalesFeedbackMatrix() {  
 **return scalesFeedbackMatrix**;  
 }  
  
 **public** Integer runNet() {  
 *// состояния первог слоя* List<Double> followingVectorState = calculateFirstState(**inputVector**);  
 *// выходы для первого слоя* List<Double> followingOutputVector = calculateOutputVector(followingVectorState);  
  
 List<Double> previousOutputVector;  
 **double** error;  
 **do** {  
 previousOutputVector = **new** ArrayList<>(followingOutputVector);  
 followingVectorState = calculateFollowingState(previousOutputVector);  
 followingOutputVector = calculateOutputVector(followingVectorState);  
 error = calculationConditionStabilizationForOutputVector(previousOutputVector, followingOutputVector);  
 **errors**.add(error);  
 } **while** (error > *EPS*);  
 **return** getFindVector(followingOutputVector);  
 }  
  
 **public** List<Double> getErrors() {  
 **return errors**;  
 }  
  
 **private** Integer getFindVector(List<Double> followingOutputVector) {  
 **int** findIndex = -1;  
 **int** count = 0;  
 **for** (**int** i = 0; i < **numberNeurons**; i++) {  
 **if** (followingOutputVector.get(i) > 0) {  
 ++count;  
 findIndex = i;  
 }  
 }  
 **return** count > 1 ? -1 : findIndex;  
 }  
  
 **private** List<Double> calculateFollowingState(List<Double> previousOutputVector) {  
 List<Double> followingVectorState = **new** ArrayList<>();  
 **double** result;  
 **double** sum;  
 **for** (**int** i = 0; i < **numberNeurons**; i++) {  
 sum = 0;  
 **for** (**int** j = 0; j < **numberNeurons**; j++) {  
 **if** (i != j) {  
 sum += previousOutputVector.get(j);  
 }  
 }  
 sum \*= **weightInhibitorySynapse**;  
 result = previousOutputVector.get(i) - sum;  
 followingVectorState.add(result);  
 }  
 **return** followingVectorState;  
 }  
}

# Тестирование

Для тестирования на вход нейронной сети были поданы 3 эталонных изображения. Количество бинарных признаков в которых было 9 и входные вектора разной зашумленности. Тестовая выборка представлена на рисунке 2. Результаты тестирования представлены в таблице 1.

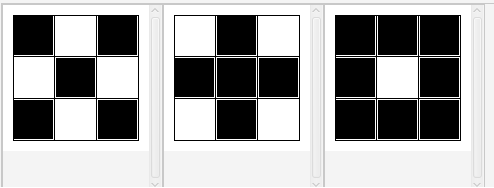


Рисунок 2. Тестовая выборка.

Таблица 1 – Результаты тестирования нейронной сети, обученной распознаванию трех эталонных

|  |  |  |
| --- | --- | --- |
| **Входной вектор** | **Результат работы сети** | **Количество итераций** |
| тест1вход.png | тест1выход.png | 3 |
| тест2вход.png | тест2выход.png | 4 |
| тест3вход.png | тест3выход.png | 3 |
| тест4вход.png | тест4выход.png | 3 |
| тест5вход.png | тест5выход.png | 3 |
| тест6вход.png | тест6выход.png | 7 |

# Вывод

В ходе проделанной работы был изучен принцип работы ассоциативной нейронной сети Хэмминга.

В качестве практического задания реализована программа, выполняющая обучение нейронной сети распознаванию эталонных образов заданных пользователем, и фильтрацию эталонных изображений от шумов.

Нейронная сеть была обучена фильтровать изображения символов “Х”, “+” и “▫” и успешно прошла тестирование на изображениях различной степени зашумленности.

# Список использованных источников

1. Комарцова Л. Г., Максимов А. В. “Нейрокомпьютеры”
2. Конспект лекций, прочитанный преподавателем Паламарь И. Н. в РГАТУ по дисциплине “Нейрокомпьютерные системы”.

Интернет источники:

1. https://ru.wikipedia.org/wiki/Нейронная\_сеть\_Хемминга
2. http://neuronus.com/nn/38-theory/971-nejronnye-seti-khemminga.html