Redes neuronales para el reconocimiento de números y símbolos aritméticos escritos a mano

Entregado a:

Gilberto Vargas Cano

Profesor Materia Computación blanda

Héctor David Mesa Santa

Gustavo García Londoño

Septiembre de 2018

Universidad Tecnológica de Pereira

1. Introducción

Consideremos la siguiente secuencia de dígitos escritos a mano:

La mayoría de la gente la reconocería sin esfuerzo. Sin embargo, no todo es tan fácil como aparenta. El cerebro humano contiene cientos de millones de neuronas, con decenas de miles de millones de conexiones entre ellas encargadas del procesamiento de la información visual. Reconocer dígitos escritos a mano no es nada fácil. Sin embargo, los seres humanos lo hacen casi sin esfuerzo, pero eso si todo sucede de manera inconsciente. Y tal vez por eso, subestimamos lo difícil que es este problema para nuestros sistemas visuales.

Resolver el problema de reconocer dígitos como los anteriores mediante un programa de computadora no es tarea fácil, tratar de expresar las formas de cada digito de forma algorítmica de por sí ya parece imposible. Por suerte, existe un modelo computacional muy interesante que nos muestra este problema de una manera distinta, las redes neuronales.

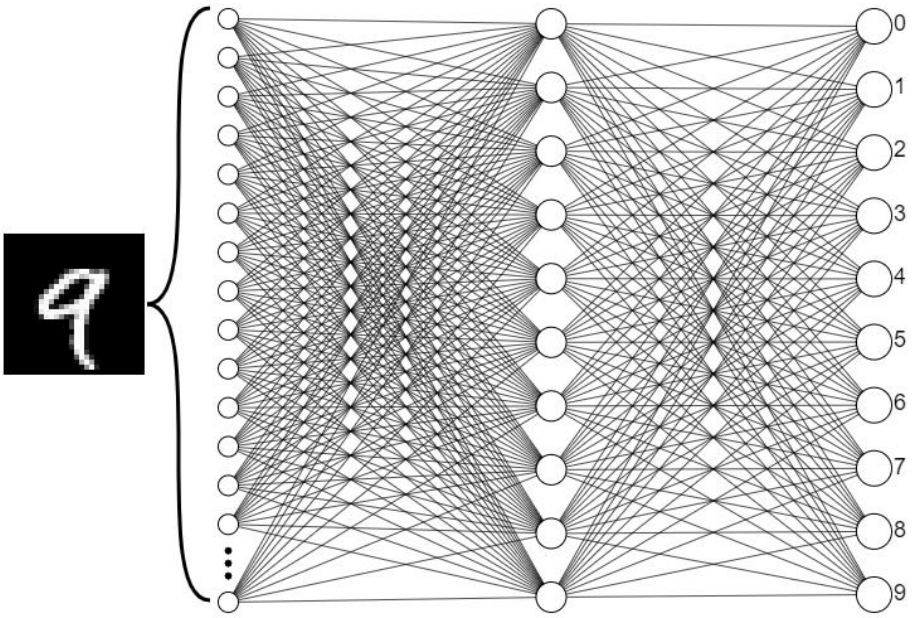
Las redes neuronales artificiales son el modelo matemático más cercano que intenta reproducir el funcionamiento del cerebro humano. Su principal objetivo es la construcción de sistemas capaces de presentar un cierto comportamiento inteligente. Esto implica la capacidad de aprender a realizar una determinada tarea. En nuestro caso, la idea es tomar un gran número de dígitos y símbolos aritméticos escritos a mano, conocido como ejemplos de entrenamiento, y luego desarrollar un sistema que pueda aprender de esos ejemplos.

2. Objetivo

En esta ocasión vamos a desarrollar un programa que implementa una red neuronal que aprende a reconocer patrones. Utilizaremos esta misma red para reconocer tanto los dígitos escritos a mano como los operadores aritméticos (+,-, x, /) y finalmente evaluaremos la red con el fin de realizar operaciones aritméticas entre dos dígitos como 5+6 , 9-7, 3*5 o 6/3.

3. Arquitectura de la red neuronal

Para reconocer dígitos individuales usaremos una red neuronal de tres capas:



La capa de entrada contendrá un total de 784 neuronas, debido a que nuestros datos de entrenamiento para la red consistirán en muchas imágenes en escala de grises de 28x28 pixeles con dígitos escritos a mano escaneados, por lo que la capa de entrada contiene 784= 28 x 28 neuronas. Los píxeles de entrada son en escala de grises, con un valor de 0.0 para el blanco, un valor de 1.0 para negro, y tonos de gris de más claro a más oscuro para los valores entre ellos.

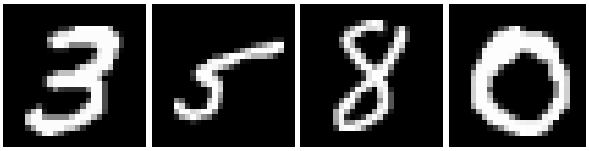
La segunda capa de la red es una capa oculta. En este caso utilizaremos 100 neuronas.

La capa de salida de la red contiene 10 neuronas. Si la primera neurona se activa, es decir, tiene una salida ≈1, indicará que la red cree que el dígito es un 0. Si la segunda neurona se activa indicará que la red cree que el dígito es un 1. Y así sucesivamente. Más concretamente, numeraremos las neuronas de salida de 0 a 9, y miraremos qué neurona tiene el valor más alto de activación. Si esa neurona es, por ejemplo, la número 6, significa que nuestra red cree que el dígito de entrada fue un 6. Y así sucesivamente para cada una de las neuronas de salida.

Para reconocer símbolos aritméticos usaremos una red neuronal de tres capas. Esta red contendrá la misma cantidad de neuronas de entrada y en la capa oculta que en la red anterior, pero a diferencia de esta solo tendrá 4 neuronas de salida. Si la primera neurona de salida se activa, es decir, tiene una salida ≈1, indicará que la red cree que el símbolo es un +. Si la segunda neurona se activa indicará que la red cree que el dígito es un -. Y así sucesivamente. Más concretamente, numeraremos las neuronas de salida de 0 a 3, y miraremos qué neurona tiene el valor más alto de activación. Si esa neurona es, por ejemplo, la número 1, significa que nuestra red cree que el símbolo de entrada fue un -. Y así sucesivamente para cada una de las neuronas de salida.

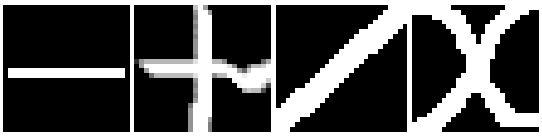
4. Entrenamiento de la red neuronal

Una vez definido el diseño para nuestra red neuronal necesitamos que esta aprenda a reconocer tanto los dígitos como los símbolos. Lo primero que necesitaremos es un conjunto de datos para aprender, un denominado conjunto de datos de entrenamiento. Utilizaremos el conjunto de datos MNIST, que contiene decenas de miles de imágenes escaneadas de dígitos escritos a mano. A continuación, algunas imágenes de MNIST:



Los datos de MNIST vienen en dos partes. La primera parte contiene 59,084 imágenes para usar como datos de entrenamiento. Las imágenes son en escala de grises y de 28 por 28 píxeles de tamaño. La segunda parte del conjunto de datos MNIST son 10,000 imágenes para ser usadas como datos de prueba. De nuevo, estas son 28 por 28 imágenes en escala de grises.

Por otra parte, utilizaremos un conjunto de datos hecho por nosotros, que contiene varias imágenes de símbolos aritméticos escritos a mano. A continuación, algunas imágenes:



Los datos de vienen en dos partes. La primera parte contiene 36 imágenes para usar como datos de entrenamiento. Las imágenes son en escala de grises y de 28 por 28 píxeles de tamaño. La segunda parte del conjunto de datos son 36 imágenes para ser usadas como datos de prueba. De nuevo, estas son 28 por 28 imágenes en escala de grises.

Utilizaremos los datos de prueba para evaluar qué tan bien nuestra red neuronal ha aprendido a reconocer los patrones.

Cada entrada de entrenamiento se considerará como un vector de longitud 28x28 =784. Cada entrada en el vector representa el valor de gris para cada píxel de la imagen. La salida deseada será representada por un vector de longitud 10 en el caso de los dígitos y un vector de longitud 4 en el caso de los símbolos. Por ejemplo, para una imagen de entrenamiento particular que representa un 3, [0, 0, 1, 0, 0, 0, 0, 0, 0, 0] es la salida deseable de la red y para una imagen de entrenamiento que representa un +, [1.0,0.0,0.0,0.0] es la salida deseable.

5. Implementación de la red neuronal para reconocimiento de patrones

Vamos a escribir un programa que aprenda a reconocer los dígitos y símbolos escritos a mano, utilizando los datos de entrenamiento de MNIST y el creado por nosotros. Haremos

esto con el lenguaje de programación Java y utilizaremos Neuroph integrado a nuestro NetBeans IDE 8.2.

5.1 Neuroph

Neuroph es un framework liviano de redes neuronales de Java que se puede usar para desarrollar arquitecturas de redes neuronales comunes. Un pequeño número de clases básicas que corresponden a los conceptos básicos de NN, y el editor de GUI hace que sea fácil de aprender y usar.

Características

- Admite perceptrones de múltiples capas con retropropagación
- Soporte de reconocimiento de imagen
- GUI fácil de usar para crear y experimentar con NN
- Admite varias otras redes neuronales y reglas de aprendizaje
- API fácil de usar

Para más información: https://github.com/neuroph/neuroph, http://neuroph.sourceforge.net/

5.2 Conjunto de entrenamiento

Los datos de entrenamiento fueron descargados desde el repositorio https://github.com/myleott/mnist_png el cual incluye todas las imágenes de entrenamiento y evaluación de MNIST convertidas a formato png.

5.3 Código

//Red.java

//En esta clase se implementa la red neuronal para el reconocimiento de patrones.

Lo primero será importar las librerías de Neuroph necesarias para el diseño de la red neuronal:

```
import org.neuroph.core.NeuralNetwork;
import org.neuroph.core.data.DataSet;
import org.neuroph.core.data.DataSetRow;
import org.neuroph.core.events.LearningEvent;
import org.neuroph.core.events.LearningEventListener;
import org.neuroph.core.learning.LearningRule;
import org.neuroph.nnet.MultiLayerPerceptron;
import org.neuroph.nnet.learning.BackPropagation;
import org.neuroph.util.TransferFunctionType;
```

```
public class Red implements LearningEventListener {

    MultiLayerPerceptron myMlPerceptron;
    NeuralNetwork neuralNet=myMlPerceptron;
    int inputlayer;
    int hiddenlayer;
    int outputlayer;

    public Red() {
    }

    public Red(int inputlayer, int hiddenlayer, int outputlayer, double learning_rate, int iterations) {
        this.inputlayer=inputlayer;
        this.outputlayer =outputlayer;
        this.hiddenlayer=hiddenlayer;
        // Creación del perceptron multicapa
        myMlPerceptron = new MultiLayerPerceptron(TransferFunctionType.SIGMOID, this.inputlayer, this.hiddenlayer, this.outputlayer);
        //Inicializacion de los pesos
        myMlPerceptron.randomizeWeights(-0.1,0.1);
        //System.out.println(Arrays.toString(myMlPerceptron.getWeights()));

     //Establece el algoritmo de aprendizaje para esta red
     //setlearning rule  ->Devuelve el algoritmo de aprendizaje de esta red
     myMlPerceptron.setLearningRule(new BackPropagation());
     //System.out.println(myMlPerceptron.getLearningRule());  Devuelve backpropagation
     myMlPerceptron.getLearningRule().setLearningRate(learning_rate);
     //myMlPerceptron.getLearningRule().setMaxError(0.01);  //--->Asi es posible modificar el error
     myMlPerceptron.getLearningRule().setMaxIterations(iterations);
     //setMaxError-->Establece el error de red permitido, que indica cuándo detener la capacitación de aprendizaje

     LearningRule learningRule = myMlPerceptron.getLearningRule();
     learningRule.addListener(this);
     //System.out.println(new BackPropagation().getMaxError());---->0.01 valor predeterminado y 2147483647 iteraciones

 }
```

Inputlayer = Numero de neuronas de la capa de entrada.

Hiddenlayer = Numero de neuronas de la capa oculta.

Outputlayer = Número de neuronas en la capa de salida.

Learning_rate = tasa de aprendizaje.

Iterations = número máximo de iteraciones.

En el constructor de la clase **Red** creamos un perceptrón multicapa con **inputlayer** neuronas en la primera capa, **hiddenlayer** neuronas en la segunda capa y **outputlayer** neuronas en la capa final. Con **TransferFunctionType.SIGMOID** definimos la función sigmoidea como la función de transferencia.

Los sesgos y los pesos se almacenan en un double vector y se inicializan aleatoriamente entre [-0.1 y 0.1], utilizando **randomizeWeights**.

Con **setLearningRule(new BackPropagation())** establecemos el algoritmo de backpropagation como el algoritmo de aprendizaje para la red. En este, el error es propagado hacia atrás desde la capa de salida. Esto permite que los pesos sobre las conexiones de las neuronas ubicadas en las capas ocultas cambien durante el entrenamiento.

A través de **setLearningRate(learning_rate)** establecemos la tasa de aprendizaje de 0.5 y con **setMaxIterations(iterations)** definimos el número máximo de iteraciones.

```java
public void training(double[][] input) {
    // Creación del conjunto de entrenamiento

    //DataSet-->Esta clase representa una colección de filas de datos
    //(instancias de DataSetRow) utilizadas para entrenar y probar redes neuronales.
    DataSet trainingSet = new DataSet(inputlayer, outputlayer);

    double [][] output = new double [outputlayer][outputlayer];
    for (int i=0; i<outputlayer; i++){
        double [] pattern_out = new double [outputlayer];
        pattern_out[i]=1.0;
        output[i]= pattern_out;
    }

    int out_n=input.length/outputlayer;
    for (int i=0; i<input.length; i++){
        int index= i/out_n;
        trainingSet.addRow(new DataSetRow(input[i], output[index]));
    }

    // Aprender el conjunto de entrenamiento
    System.out.println("Training neural network...");

    myMlPerceptron.learn(trainingSet);   //otra forma -->learn(DataSet trainingSet,L
    neuralNet=myMlPerceptron;
    myMlPerceptron.save("myMlPerceptron30.nnet");

}
```

En el método **training s**e generan todos los patrones de salida que corresponden a todas las posibles salidas que puede tener la red neuronal. Cada salida se codifica de la siguiente manera:

En el caso de los dígitos:

[1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0] → Salida: 0

[0.0,1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0]→ Salida: 1

[0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0] → Salida: 2

[0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,0.0] → Salida: 3

[0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0] → Salida: 4

[0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0]→ Salida: 5

[0.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0] → Salida: 6

[0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0] → Salida: 7

[0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0] → Salida: 8

[0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0] → Salida: 9

En el caso de los símbolos aritméticos:

[1.0,0.0,0.0,0.0] → Salida: +

[0.0,1.0,0.0,0.0]→ Salida: -

[0.0,0.0,1.0,0.0] → Salida: *

[0.0,0.0,0.0,1.0] → Salida: /

A continuación, se almacena en el objeto **trainingSet** las entradas de entrenamiento (almacenadas en **input**) y las salidas deseadas correspondientes (almacenadas en **output**). En otras palabras, se guardan las entradas con sus etiquetas. Finalmente, con **learn(trainingSet)** entrenamos la red con el conjunto de entrenamiento.

```
public int testNeuralNetwork(double [] numbertest){
    neuralNet.setInput(numbertest);

    neuralNet.calculate();
    double[] networkOutput = neuralNet.getOutput();

    //System.out.println(NeuralNet.getLearningRule().getTotalNetworkError());
    double max=0.0;
    int index_max=0;

    for (int i=0; i<networkOutput.length;i++){
        if (networkOutput[i]>max){
            max=networkOutput[i];
            index_max=i;
        }
    }
    //System.out.print("Input: " + Arrays.toString( numbertest ) );
        //System.out.println(" Output: " + Arrays.toString( networkOutput) );
        //System.out.println(" Output: " + index_max );

    return index_max;
}
```

A través del método **testNeuralNetwork** evaluamos nuestra red neuronal un dato de prueba. Mas específicamente mediante **calculate()** se genera la salida para dato de entrada y finalmente se busca en cada vector de salida el índice del valor máximo que nos indica que numero o símbolo arroja nuestra red.

```
    public void handleLearningEvent(LearningEvent event) {
        BackPropagation bp = (BackPropagation)event.getSource();
                if (event.getEventType() != LearningEvent.Type.LEARNING_STOPPED)
                    System.out.println(bp.getCurrentIteration() + ". iteration : "+ bp.getTotalNetworkError());
    }

}
```

Finalmente, el método **handleLearningEvent** evaluará el error total de la red después de cada epoch de capacitación e imprimirá un progreso parcial. Esto es útil para rastrear el progreso, pero ralentiza considerablemente las cosas.

6. Reconocimiento de digitos o simbolos

El reconocimiento de digitos que utiliza la red neuronal para entrenarse y para “entender” que digito le esta ingresando y poder reconocerlo correctamente es el siguiente:

6.1 Reconocer la imagen

El primer paso que hace este algoritmo es reconocer la imagen, para esto leemos la imagen con **readImage()** como una **BufferedImage** que es un metodo de la clase **Number** que utilizamos para el reconocimiento de las imágenes, para leer la imagen utilizamos una librería de java llamada **Image.IO.read()**

6.2 Obtener pixeles

El segundo paso a seguir para el reconocimiento de los digitos o simbolos es obtener los pixeles, los cuales los obtenemos con el metodo **obtainPixelsList()** de la clase **Number**, este metodo lo primero que hace es en un arreglo de enteros almacenar el valor del **ancho*alto** de cada pixel, seguido a esto intera sobre el ancho y el alto del arreglo para obtener de cada pixel su respectivo **RGB**, entendamos esta parte como si el pixel seleccionado esta pintado ponemos 1 de lo contrario ponemos 0, y asi sucesivamente hasta recorrer la cantidad de pixeles de la imagen.

6.3 Cargar u entrenar la red con los patrones de entrenamiento

Antes de cargar todos los patrones o entrenar la red neuronal se crea un arreglo de enteros que va a almacenar el resultado de las verificaciones y comparaciones que hara el algoritmo mas adelante. Para cargar todos los patrones llamamos el metodo **LoadPatterns()** Que lo que hara es lo mismo que hicimos con la imagen anterior (digito a reconocer) pero con las imágenes de entrenamiento que tiene el algoritmo (10 de cada digito), entonces lee la imagen y obtiene los pixeles de la misma y esto es almacenado en un arreglo llamado **patternsForCurrentNumber** y despues almacenar cada uno de los patrones ya obtenidos en un **Hashmap** el cual su llave sera el digito y su valor de los diferentes patrones diferentes del digito.

6.4 Obtener numero de coincidencias

Como antes del paso anterior creamos un arreglo donde se iban a almacenar todos los resultados, en este paso esa variable toma una gran importancia por que en esta variable es donde se va a almacenar el resultado de la invocacion de un metodo llamado **numberOfCoincidences()**, el cual se encarga de sumar la cantidad de coincidencias encontradas de el patron del digito ingresado con todos los patrones ingresados en el paso anterior (**LoadPatterns()**) y retornar la cantidad de coincidencias encontradas.

6.5 Retornar winner

Como en el paso anterior recorrimos todo el arreglo de patrones del algoritmo buscando la cantidad de coincidencias encontradas con la imagen del digito o simbolo ingresado, ahora

tenemos que recorrer el arreglo para obtener el digito que mas coincidencias obtuvo para retornarlo y ese digito sera el correspondiente al digito que me ingresaron para reconocer.

//RedNeuronal.java

Para completar, aquí está el código que indica cómo se cargan y almacenan las imágenes los dígitos y símbolos escritos a mano.

Webgrafía

http://neuralnetworksanddeeplearning.com/chap1.html