Neural Network PRNG

Android STUDIO.

Previously, this was called "sloppily written program code." Now it's called obfuscation / Science doesn't yet know who said it /

Obfuscation (from the Latin obfuscare — to shade, to darken; and the English obfuscate — to make non-obvious, confusing, to confuse) or code obfuscation is bringing the source code or executable code of a program to a form that preserves its functionality, but complicates analysis, understanding of the algorithms of work

SOURCE CODE
A software pseudo-random number generator (PRNG) based on a simple neural network without training - NPRNG

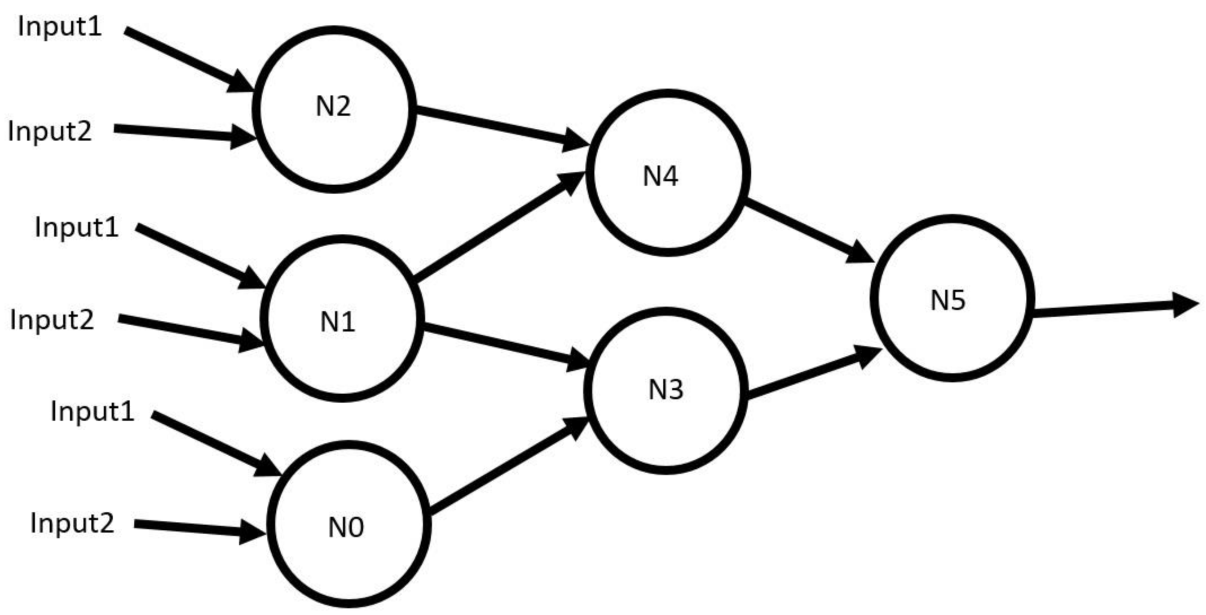
A complete description of the process of creating a prototype of a random number generator based on a neural network of 6 neurons and 3 layers.

We use a very simple neural network.

1. Let's create a method that will correct the WEIGHT of a neuron so that the output value is less than one.
2. Use one of the bits of the neuron's output value as one of the values of the generated pseudo-random number
3. Use another bit to correct the input values of the neural network (master key) before each subsequent calculation.
4. Compose two pseudo-random numbers from the selected bits
5. Divide one number by the other and, if there is a remainder from the division, take the parity bit.
6. Compose the desired pseudo-random number of the REQUIRED SIZE from the selected bits
7. Application of Neural PRNG in radio communications and radar systems resistant to electronic warfare.

The application listing provided here is a prototype of a neural PRNG. But it can (see point 6) generate pseudo-random numbers that are quite suitable for use in commercial encryption. Then we will consider ways to improve the quality of generation.

Neural network scheme



Android JAVA Project

Since we are creating a prototype of the application, in the JAVA project we will not waste time on the user interface and all the results will be shown in the LOG

There is nothing in **MainActivity.java** except for a call to the **App.java** class, in which all processing is performed.

```
App cls2 = new App(); // Declare a class with a neural network
cls2.trainAndPredict(); // Calling a method from App.java
```

If you intend to create a user interface and get a fully functional application, then you can work with it in **MainActivity.java**

Let's look at **App.java**

The input data of a neural network (dataset or master key) consists of two values that are used only in the first computation of the network and are then adjusted through feedback.

```
data[0][0] = 115; // input1
data[0][1] = 66; // input2
```

We use only one pair of values and only in the first step of direct computation in the neural network.
Immediately, the output values in each neuron will be checked. If they are 0 or 1 (saturation), then the method of adjusting the weights for this neuron will be used until the neuron's output is different from 0 or 1.

Unlike the classic neural network initialization, in this project we will introduce weight coefficients statically. This is also part of the master key, which ensures the repetition of results in other instances of the application

```
        // The numbers were selected during neural network testing
        neurobiasweight [0][1] = 0.2921867401328437; // Neuron 0 weight 1
        neurobiasweight [0][2] = 0.6753277674739143; // Neuron 0 weight 2
        neurobiasweight [1][1] = 0.638204802420569; // Neuron 1 weight 1
        neurobiasweight [1][2] = 0.8256511687684168; // Neuron 1 weight 2
        neurobiasweight [2][1] = 0.9570075176019762; // Neuron 2 weight 1
        neurobiasweight [2][2] = 0.39564267118987084; // Neuron 2 weight 2
        neurobiasweight [3][1] = 0.2463716717762578; // Neuron 3 weight 1
        neurobiasweight [3][2] = 0.8565685160656823; // Neuron 3 weight 2
        neurobiasweight [4][1] = 0.8529762783770741; // Neuron 4 weight 1
        neurobiasweight [4][2] = 0.7818602822337215; // Нейрон 4 weight 2
        neurobiasweight [5][1] = 0.36204206369478864; // Neuron 5 weight 1
        neurobiasweight [5][2] = 0.6998272379065108; // Neuron 5 weight 2
        // We will write the offset into the same array
        neurobiasweight [0][0] = 0.6372796901523922; // Neuron 0 bias
        neurobiasweight [1][0] = 0.24087856177175287; // Neuron 1 bias
        neurobiasweight [2][0] = 0.20294311087331438; // Neuron 2 bias
        neurobiasweight [3][0] = 0.6005802079109952; // Neuron 3 bias
        neurobiasweight [4][0] = 0.24210376151411686; // Neuron 4 bias
        neurobiasweight [5][0] = 0.3291532687509765; // Neuron 5 bias
        // Fill the array of output values of neurons with any initial numbers
        outputnnum [0] = 0.0;
        outputnnum [1] = 0.0;
        outputnnum [2] = 0.0;
        outputnnum [3] = 0.0;
        outputnnum [4] = 0.0;
        outputnnum [5] = 0.0;
```

The neurobiasweight values will be adjusted immediately after the first neural network evaluation if they are equal to 0 or 1.

```
    // Executes before public double compute. If neuron output = 0 or 1, then adjust each
weight and recalculate
    public double computeOOS(double input1, double input2, int nnum){ // Correction of
weight coefficients for neuron nnum
    // If the output of neuron nnum is 1 or 0, then adjust weight1 and weight2 of this neuron
        App app = new App(); // Class instance

        while (compute(input1, input2,nnum) == 1.0) {// Calling the standard COMPUTE
            app.neurobiasweight [nnum][1] -= 0.021E1; // Weight 1 of neuron nnum
            app.neurobiasweight [nnum][2] -= 0.027E1; // Weight 2 of neuron nnum
            //Log.i("== TRAIN =", "
==>>>==============================================================
==== == ");
        }

        while (compute(input1, input2,nnum) == 0.0) {// Calling the standard COMPUTE
            app.neurobiasweight [nnum][1] += 0.027E1; // Weight 1 of neuron nnum
            app.neurobiasweight [nnum][2] += 0.021E1; // Weight 2 of neuron nnum
            //Log.i("== TRAIN =", " ==>>>
>>>================================================================
== == ");
        }

        double outputoos = compute(input1, input2,nnum);

        app.outputnnum[nnum]=outputoos; // Remember the output values of a neuron nnum

        return outputoos;
    } // computeOOS
```

The values for weight adjustment are not large and were chosen experimentally.

Running a computation in a neural network

In public class App we execute

Double Te = network1000.predict(*data*[0][0],*data*[0][1]);

In Double Te - the result of direct computation of the neural network - the output of neuron #5

Neural Network Calculations - **predict** Method

```
    //----- DIRECT (MAIN) CALCULATION -------------------------
    public Double predict(Integer input1, Integer input2){ // Input1 and input2 are used as
input.
        Log.i("== NCOMPUTE =", " ==
===============================================================
== ");
        return neurons.get(5).computeOOS( // Where neurons is List<Neuron> neurons =
Arrays.asList(new Neuron(), new Neuron(), new Neuron(),new Neuron(), new Neuron(),new
Neuron());
                neurons.get(4).computeOOS( // Compute - initial activation. class Neuron
                        neurons.get(2).computeOOS(input1, input2, 2), // compute - here is SIGMOID
- Util.sigmoid(preActivation)
                        neurons.get(1).computeOOS(input1, input2, 1),4  // compute - here is
SIGMOID - Util.sigmoid(preActivation)
                ),
            neurons.get(3).computeOOS(
                  neurons.get(1).computeOOS(input1, input2,1), // Get the FIRST element
(neuron) from the list of neurons with input values input1, input2
                  neurons.get(0).computeOOS(input1, input2,0),3  // compute - here is
SIGMOID - Util.sigmoid(preActivation)
            ),5
      );
   }
```

Here the input values and neuron number are used - **input1, input2, 0**

Calculating the output values of neurons in the first forward computation

```
   //----- Calculate the output of neuron nnum -------------------------
   public double compute(double input1, double input2, int nnum){ // Calculate the output of
neuron nnum
      App app = new App(); // Class instance
      double preActivation = (app.neurobiasweight [nnum][1] * input1) +
(app.neurobiasweight [nnum][2] * input2) + app.neurobiasweight [nnum][0]; // Вес 1*Вход 1
+ Вес 2*Вход 2 * Смещение
      // Pass through Sigmoid
      double output = Util.sigmoid(preActivation); // Range of output values - [0,1].

      normalize(output); // To organize feedback
      correctinput(); // If not even, then subtract from Input

      return output;
   }
```

nnum - neuron number

After the first calculation of the output values of neurons, we determine whether any of them are equal to 0 or 1. If “yes”, then we begin to adjust the weight coefficients of this neuron until its output value becomes greater than 0 and less than 1.
This is done immediately during the first calculation in the neural network.

After that, you can select ONE DIGIT from each output value of each neuron and get 6 digits of a PSEUDO-RANDOM NUMBER.

Here we need to check that the generated output values of neurons have enough digits (after the decimal point). **WRITE CODE FOR THE PROJECT** (we will do this later)

Testing the prototype with a sample of one digit from the output value of each neuron showed the weakness of this approach. The neurons of the input layer give close values and the pseudo-random number contains many repetitions of digits in a row.

```
==>>> >>> >>>== PRNG OUTPUT pseudornd=88981291921
```

You can try to select numbers from different positions, but we will try another solution - we will use one digit of the neural network output value after each calculation. This works slower, but the generated (accumulated) number looks better.

```
==>>> >>> >>>== PRNG OUTPUT pseudornd=9870945791
```

Full listing of the program at this stage:

(http://templates.oflameron.ru/page00014.htm)

When trying to generate a very large number (more than 100 characters), the application freezes. We need to find the reason. Either calculation errors, or an invalid range of values, or we need to perform actions in a separate thread. But this is in the next part of the project

Full JAVA code for this stage 23-08-2024

App.java

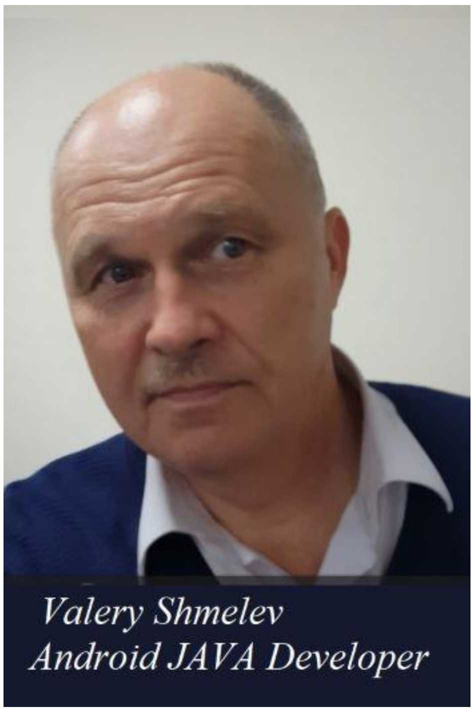
```
package com.example.neuroprng1;


import android.util.Log;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Random;

// Neural PRNG. Minimal Java Code - Minimal working Java code
// 23-08-2024 Valery Shmelev
// Java code is written for student work as a demonstration prototype
// http://templates.oflameron.ru/page00014.htm
// https://docs.google.com/document/d/1wQMjcCZXPgPtuc-s0QSABNItSGmaTdF6Py1-
mRGYcBI/edit?usp=sharing

//---------------------------------------------------------------
public class App {
   public static Double neurobiasweight [][] = new Double[6][3]; // Array of weights for neurons [weight1][weight2]
   public static Integer deltainput1 = 0; // Correcting input1 via feedback
   public static Integer deltainput2 = 0; // Correcting input2 via feedback
   public static Integer data [][] = new Integer[4][4]; // Two-dimensional array of INPUT DATA (mpster-key) -
input1 and input2
   public static Integer neuronnum = 0; // Neuron Number
   public static Double outputnnum [] = new Double[6]; // Output values of neurons
   public static String pseudornd = ""; // Here we will accumulate a pseudo-random number - String

   public static void main( String[] args ) {

   }


   public void trainAndPredict() {
      // Fill in input1 and input2 with values
```

```
        data[0][0] = 115; // input1
        data[0][1] = 66; // input2

        // Let's assign a static weight value to each neuron. To ensure repeatability of results across different
instances of the application NPRNG
        // The numbers were selected during neural network testing
        neurobiasweight [0][1] = 0.2921867401328437; // Neuron 0 weight 1
        neurobiasweight [0][2] = 0.6753277674739143; // Neuron 0 weight 2
        neurobiasweight [1][1] = 0.638204802420569; // Neuron 1 weight 1
        neurobiasweight [1][2] = 0.8256511687684168; // Neuron 1 weight 2
        neurobiasweight [2][1] = 0.9570075176019762; // Neuron 2 weight 1
        neurobiasweight [2][2] = 0.39564267118987084; // Neuron 2 weight 2
        neurobiasweight [3][1] = 0.2463716717762578; // Neuron 3 weight 1
        neurobiasweight [3][2] = 0.8565685160656823; // Neuron 3 weight 2
        neurobiasweight [4][1] = 0.8529762783770741; // Neuron 4 weight 1
        neurobiasweight [4][2] = 0.7818602822337215; // Нейрон 4 weight 2
        neurobiasweight [5][1] = 0.36204206369478864; // Neuron 5 weight 1
        neurobiasweight [5][2] = 0.6998272379065108; // Neuron 5 weight 2
        // We will write the offset into the same array
        neurobiasweight [0][0] = 0.6372796901523922; // Neuron 0 bias
        neurobiasweight [1][0] = 0.24087856177175287; // Neuron 1 bias
        neurobiasweight [2][0] = 0.20294311087331438; // Neuron 2 bias
        neurobiasweight [3][0] = 0.6005802079109952; // Neuron 3 bias
        neurobiasweight [4][0] = 0.24210376151411686; // Neuron 4 bias
        neurobiasweight [5][0] = 0.3291532687509765; // Neuron 5 bias
        // Fill the array of output values of neurons with any initial numbers
        outputnnum [0] = 0.0;
        outputnnum [1] = 0.0;
        outputnnum [2] = 0.0;
        outputnnum [3] = 0.0;
        outputnnum [4] = 0.0;
        outputnnum [5] = 0.0;

        // We create a network of 6 neurons, set the input values and write the result to the screen - the variable Te
        Network networkprng = new Network(); // We create a neural network networkprng

        Double Te = networkprng.predict(data[0][0],data[0][1]); // predict(inpur1,input2) - Return Double
        Log.i("== NNETWORK 1 Output =", " == == data[0][0] и data[0][1]   Input1=" + String.valueOf(data[0][0]) + "
и Input2=" + String.valueOf(data[0][1]));

        for (int n=0; n<10; n++) { // Some direct calculations for accumulating a pseudo-random number
          // The following direct calculation
          Te = networkprng.predict(data[0][0], data[0][1]); // predict(inpur1,input2) - Return Double
          Log.i("== NNETWORK 2 Output =", " == == data[0][0] и data[0][1]   Input1=" + String.valueOf(data[0][0]) +
"  и Input2=" + String.valueOf(data[0][1]));
          income5bigint(); // Accumulate a pseudo-random number from only the output values of the neural
network (neuron #5)
        }

        Log.i("== PSEUDORANDOM =", " ==>>> >>> >>>== PRNG OUTPUT pseudornd=" + pseudornd);

    }

    public static Integer czech(){ // Write output values of neurons to Log (for debugging)
        Log.i("== NNUM OUTPUT =", " ==>>> >>> >>>=========== OUTPUT nnum0=" +
String.valueOf(outputnnum [0]));
        Log.i("== NNUM OUTPUT =", " ==>>> >>> >>>=========== OUTPUT nnum1=" +
String.valueOf(outputnnum [1]));
```

```
        Log.i("== NNUM OUTPUT =", " ==>>> >>> >>>============ OUTPUT nnum2=" +
String.valueOf(outputnnum [2]));
        Log.i("== NNUM OUTPUT =", " ==>>> >>> >>>============ OUTPUT nnum3=" +
String.valueOf(outputnnum [3]));
        Log.i("== NNUM OUTPUT =", " ==>>> >>> >>>============ OUTPUT nnum4=" +
String.valueOf(outputnnum [4]));
        Log.i("== NNUM OUTPUT =", " ==>>> >>> >>>============ OUTPUT nnum5=" +
String.valueOf(outputnnum [5]));
        return (null);
    }

    public static Integer incomebigint(){ // Accumulate pseudorandom number in pseudornd

    for (int g=0; g<6; g++){ // View the entire outputnnum array
        String str = String.valueOf(outputnnum [g]); // Convert Double to String
        String substr = str.substring(5, 6); // One digit
        pseudornd += substr;
    }
        Log.i("== PSEUDORANDOM =", " ==>>> >>> >>>== PRNG OUTPUT pseudornd=" + pseudornd);

        return (null);
    }

    public static Integer income5bigint(){ // Accumulate pseudorandom number in pseudornd only from OUTPUT
neuron #5

            String str = String.valueOf(outputnnum [5]); // Convert Double to String
            String substr = str.substring(5, 6); //  One digit
            pseudornd += substr;

        return (null);
    }


} // App

//---------------------------------------------------------------------

class Network { // Called from the App class
    List<Neuron> neurons = Arrays.asList( // An extensible list of neurons from the Neuron class
            new Neuron(), new Neuron(), new Neuron(),   // Entry level
            new Neuron(), new Neuron(),                 // Hidden level
            new Neuron());                              // Output level

    //----- DIRECT (MAIN) CALCULATION --------------------------
    public Double predict(Integer input1, Integer input2){ // Input1 and input2 are used as input.
        Log.i("== NCOMPUTE =", " ==
==================================================================== == ");
        return neurons.get(5).computeOOS( // Where neurons is List<Neuron> neurons = Arrays.asList(new
Neuron(), new Neuron(), new Neuron(),new Neuron(), new Neuron(),new Neuron());
                neurons.get(4).computeOOS( // Compute - initial activation. class Neuron
                        neurons.get(2).computeOOS(input1, input2, 2), // compute - here is SIGMOID -
Util.sigmoid(preActivation)
                        neurons.get(1).computeOOS(input1, input2, 1),4  // compute - here is SIGMOID -
Util.sigmoid(preActivation)
                ),
                neurons.get(3).computeOOS(
                    neurons.get(1).computeOOS(input1, input2,1), // Get the FIRST element (neuron) from the list of
neurons with input values input1, input2
                    neurons.get(0).computeOOS(input1, input2,0),3  // compute - here is SIGMOID -
Util.sigmoid(preActivation)
             ),5
      );
   }

} // Network

//-------------------------------------------------------------------

class Neuron {

   //----- Calculate the output of neuron nnum --------------------------
   public double compute(double input1, double input2, int nnum){ // Calculate the output of neuron nnum
      App app = new App(); // Class instance
      double preActivation = (app.neurobiasweight [nnum][1] * input1) + (app.neurobiasweight [nnum][2] * input2)
+ app.neurobiasweight [nnum][0]; // Вес 1*Вход 1 + Вес 2*Вход 2 * Смещение
      // Pass through Sigmoid
      double output = Util.sigmoid(preActivation); // Range of output values - [0,1].

      normalize(output); // To organize feedback
      correctinput(); // If not even, then subtract from Input

      return output;
   }

   // Executes before public double compute. If neuron output = 0 or 1, then adjust each weight and recalculate
   public double computeOOS(double input1, double input2, int nnum){ // Correction of weight coefficients for
neuron nnum
   // If the output of neuron nnum is 1 or 0, then adjust weight1 and weight2 of this neuron
      App app = new App(); // Class instance

      while (compute(input1, input2,nnum) == 1.0) {// Calling the standard COMPUTE
         app.neurobiasweight [nnum][1] -= 0.021E1; // Weight 1 of neuron nnum
         app.neurobiasweight [nnum][2] -= 0.027E1; // Weight 2 of neuron nnum
         //Log.i("== TRAIN =", "
==>>>================================================================== == ");
      }

      while (compute(input1, input2,nnum) == 0.0) {// Calling the standard COMPUTE
         app.neurobiasweight [nnum][1] += 0.027E1; // Weight 1 of neuron nnum
         app.neurobiasweight [nnum][2] += 0.021E1; // Weight 2 of neuron nnum
         //Log.i("== TRAIN =", " ==>>>
>>>==================================================================== == ");
      }

      double outputoos = compute(input1, input2,nnum);

      app.outputnnum[nnum]=outputoos; // Remember the output values of a neuron nnum

      return outputoos;
   } // computeOOS

   // From the output values of neurons, we select individual bits to adjust the master key (input1 and input2)
   public Integer normalize(double neuroutput) { // Check the output value of the neuron that it is not 0 and not 1
      App app = new App(); // Class instance
```

```
        Integer result= 0;
        if (neuroutput != 0){
            if (neuroutput !=1) {
                String str = String.valueOf(neuroutput); // Convert Double to String
                // Here you need to check the length of the resulting string, otherwise these digits may not be there -
method Czech_Line
                String substr = str.substring(7, 8); // One digit
                result = Integer.valueOf(substr); // Single-Digit Integer
                app.deltainput1 = Integer.valueOf(str.substring(7, 8)); // Single-Digit Integer for correction weight1
                app.deltainput2 = Integer.valueOf(str.substring(8, 9)); // Single-Digit Integer for correction weight2
                ////Log.i("== NORMALIZE =", " == == Integer.valueOf(substr) == == substr=" + str.substring(7, 8));
                ////Log.i("== NORMALIZE =", " == == Integer.valueOf(substr) == == substr=" + str.substring(8, 9));
            }
        }
        return (null);
    }

    // Correct master key (input1 and input2)
    public Integer correctinput() { // If deltaweight is even, then with a plus, if not even, then with a minus
        App app = new App(); // Class instance
        if (app.deltainput1 % 2 != 0){ // The number is not even
            app.deltainput1 *= -1; // Take with a minus sign
         }
        if (app.deltainput2 % 2 != 0){ // The number is not even
            app.deltainput2 *= -1; // Take with a minus sign
        }
        app.data[0][0] += app.deltainput1; // Correct input1
        app.data[0][1] += app.deltainput2; // Correct input2

        return (null);
    }

} // Neuron


//----- Activation function - SIGMOID ----------------------------
class Util {
    public static double sigmoid(double in){ // Activation function - SIGMOID. Range of values [0,1]
        // Here in is preActivation = input1 * weight1 + input2 * weight2 + bias

        return 1 / (1 + Math.exp(-in));  // Sigmoid. Compresses values into the range from 0 to 1
    }

    //    (1 / (1 + Math.exp(-in)))* (1 - in)
} // Util class
```

The JAVA code provided is not optimal and is provided “as is”.
Let’s continue



Neural Network PRNG