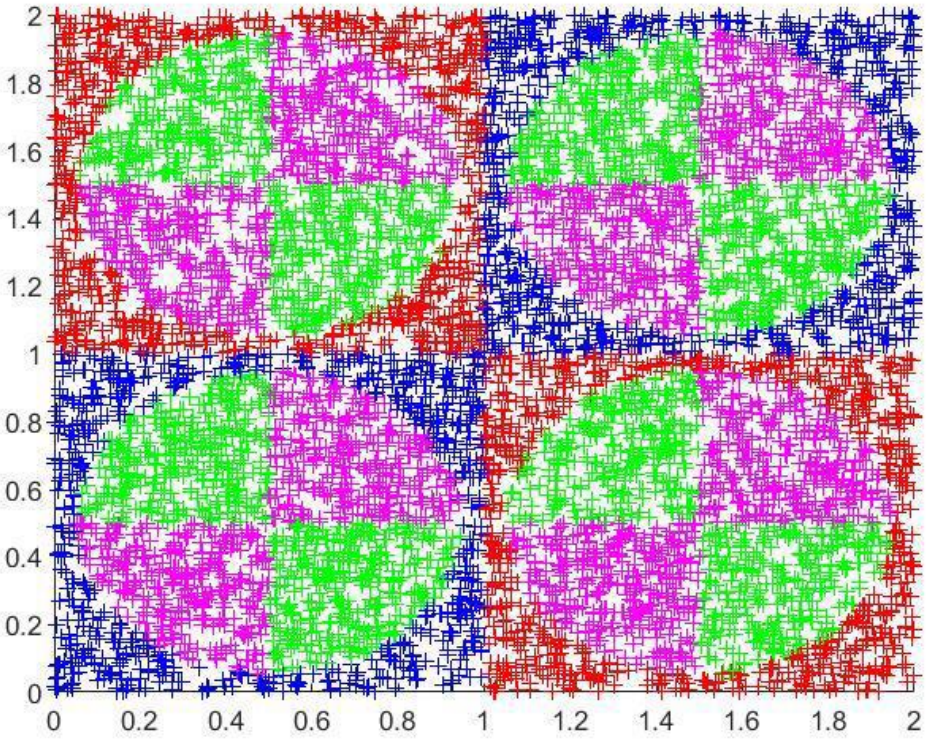
ΜΥΕ035
Υπολογιστική Νοημοσύνη

Άσκηση 1



Μέλη Ομάδας:

Γεώργιος Κεφάλας, Α.Μ: 5252

Δημήτριος Λούκας, Α.Μ: 5270

Εισαγωγή

Στην παρούσα εργασία υλοποιείται ένα πολυεπίπεδο τεχνητό νευρωνικό δίκτυο (Multi-Layer Perceptron – MLP) για πρόβλημα ταξινόμησης. Το δίκτυο εκπαιδεύεται με τον αλγόριθμο backpropagation και ενημερώνει τα βάρη μέσω gradient descent, χρησιμοποιώντας εκπαίδευση με mini-batches.

Το σύνολο δεδομένων χωρίζεται σε σύνολο εκπαίδευσης (train.txt) και σύνολο ελέγχου (test.txt). Η εκπαίδευση πραγματοποιείται στο σύνολο εκπαίδευσης, ενώ η απόδοση του δικτύου αξιολογείται στο σύνολο ελέγχου μέσω του ποσοστού σωστών ταξινομήσεων (γενικευτική ικανότητα).

Στο πλαίσιο της εργασίας εξετάζεται η επίδραση της αρχιτεκτονικής του δικτύου (αριθμός νευρώνων στα κρυμμένα επίπεδα), της συνάρτησης ενεργοποίησης και του μεγέθους των mini-batches στη γενικευτική ικανότητα του μοντέλου.

Η υλοποίηση πραγματοποιήθηκε σε γλώσσα Java και περιλαμβάνει τα αρχεία MLP_SDT.java, Activation.java, Main.java καθώς και τα αρχεία δεδομένων train.txt και test.txt.

Σχολιασμός Κώδικα

Activation.java

Το αρχείο αυτό υλοποιεί τις συναρτήσεις ενεργοποίησης και τις αντίστοιχες παραγώγους τους, οι οποίες χρησιμοποιούνται τόσο στη φάση της προώθησης όσο και στη φάση της οπισθοδιάδοσης του σφάλματος.
Υποστηρίζονται οι εξής συναρτήσεις με βάση την εκφώνηση:

Logistic, tanh, relu

Η επιλογή της συνάρτησης γίνεται δυναμικά μέσω της παραμέτρου type.

```java
public class Activation {

    public static double activate(String type, double u) {
        switch (type.toLowerCase()) {
            case "logistic":
                return 1.0 / (1.0 + Math.exp(-u));
            case "tanh":
                return Math.tanh(u);
            case "relu":
            default:
                return Math.max(a: 0.0, u);
        }
    }

    public static double derivative(String type, double u, double activatedValue) {
        switch (type.toLowerCase()) {
            case "logistic":
                return activatedValue * (1.0 - activatedValue);
            case "tanh":
                return 1.0 - activatedValue * activatedValue;
            case "relu":
            default:
                return u > 0.0 ? 1.0 : 0.0;
        }
    }
}
```

MLP_SDT.java

Το αρχείο αυτό υλοποιεί το MLP της άσκησης. Το δίκτυο υποστηρίζει τρεις κρυμμένες στρώσεις, εκπαίδευση με mini-batch gradient descent καθώς και αξιολόγηση της γενικευτικής ικανότητας.

Αρχικά ορίζουμε τη δομή του δικτύου. Ορίζουμε όλες τις παραμέτρους που επηρεάζουν την δομή του δικτύου όπως αριθμό νευρώνων κάθε επιπέδου, αριθμό κλάσεων, batch size, κτλπ.

```java
// CONFIG (like #define)
public static final int d = 2;            // input dimension
public static final int K = 4;            // number of classes (SDT -> 4)
public static final int H1 = 32;          // neurons in hidden layer 1
public static final int H2 = 32;          // hidden layer 2
public static final int H3 = 16;          // hidden layer 3

// "logistic", "tanh", "relu"
public static final String HIDDEN_ACTIVATION = "tanh";

// Training hyperparameters (can be changed at runtime)
public double learningRate = 0.02;
public int batchSize = 20;        // L: mini-batch size
public int maxEpochs = 1000;      // safety cap
public int minEpochs = 800;       // must run at least this many epochs
public double stopDiffThreshold = 0.001; // threshold on difference of TOTAL training error between epochs

// Network parameters (global variables)
private double[][] W1; private double[] b1; // H1 x d, H1
private double[][] W2; private double[] b2; // H2 x H1, H2
private double[][] W3; private double[] b3; // H3 x H2, H3
private double[][] W4; private double[] b4; // K x H3, K
```

Στην συνέχεια υλοποιήθηκε η forward συνάρτηση, η οποία αντιστοιχεί στην forward-pass της εκφώνησης. Υλοποιεί πλήρως τον υπολογισμό των εξόδων του δικτύου για ένα δείγμα εισόδου. Το επίπεδο εξόδου χρησιμοποιεί τη λογιστική συνάρτηση ενεργοποίησης, καθώς σύμφωνα με τις διαφάνειες του μαθήματος είναι ιδανική για προβλήματα ταξινόμησης.

```java
public double[] forward(double[] x) {
    // layer 1
    for (int i = 0; i < H1; i++) {
        double s = b1[i];
        for (int j = 0; j < d; j++) s += W1[i][j] * x[j];
        z1[i] = s;
        a1[i] = Activation.activate(HIDDEN_ACTIVATION, s);
    }
    // layer 2
    for (int i = 0; i < H2; i++) {
        double s = b2[i];
        for (int j = 0; j < H1; j++) s += W2[i][j] * a1[j];
        z2[i] = s;
        a2[i] = Activation.activate(HIDDEN_ACTIVATION, s);
    }
    // layer 3
    for (int i = 0; i < H3; i++) {
        double s = b3[i];
        for (int j = 0; j < H2; j++) s += W3[i][j] * a2[j];
        z3[i] = s;
        a3[i] = Activation.activate(type: "tanh", s);          // CHANGE 3RD LAYER
    }
    // output layer (logistic activation)
    for (int i = 0; i < K; i++) {
        double s = b4[i];
        for (int j = 0; j < H3; j++) s += W4[i][j] * a3[j];
        z4[i] = s;
        a4[i] = Activation.activate(type: "logistic", s); // logistic
    }

    // return copy
    double[] out = new double[K];
    System.arraycopy(a4, srcPos: 0, out, destPos: 0, K);
    return out;
}
```

Η συνάρτηση της εκφώνησης backprop υλοποιείται στο πρόγραμμά μας από την computeGradientsSingle. Η συνάρτηση αυτή υπολογίζει τις μερικές παραγώγους του σφάλματος ως προς όλα τα βάρη και όλες τις πολώσεις. Ξεκινά από το επίπεδο εξόδου και προχωρά αντίστροφα. Δεν ενημερώνει άμεσα τα βάρη αλλά τα συσσωρεύει.

```java
private void computeGradientsSingle(double[] x, double[] t,
                                    double[][] dW1, double[] db1,
                                    double[][] dW2, double[] db2,
                                    double[][] dW3, double[] db3,
                                    double[][] dW4, double[] db4) {

    // forward(x) must have been called before to populate a1,a2,a3,a4 and z1,z2,z3,z4
    // output delta for LINEAR output + MSE
    double[] delta4 = new double[K];
    for (int i = 0; i < K; i++) {
        delta4[i] = (a4[i] - t[i]) * Activation.derivative(type: "logistic", z4[i], a4[i]); // MSE + logistic derivative
    }

    // gradients for W4 and b4
    for (int i = 0; i < K; i++) {
        db4[i] += delta4[i];
        for (int j = 0; j < H3; j++) dW4[i][j] += delta4[i] * a3[j];
    }

    // backprop to layer3
    double[] delta3 = new double[H3];
    for (int j = 0; j < H3; j++) {
        double s = 0.0;
        for (int i = 0; i < K; i++) s += W4[i][j] * delta4[i];
        delta3[j] = s * Activation.derivative(type: "tanh", z3[j], a3[j]);        // CHANGE 3RD LAYER
    }
    for (int j = 0; j < H3; j++) {
        db3[j] += delta3[j];
        for (int k = 0; k < H2; k++) dW3[j][k] += delta3[j] * a2[k];
    }

    // backprop to layer2
    double[] delta2 = new double[H2];
    for (int j = 0; j < H2; j++) {
        double s = 0.0;
        for (int i = 0; i < H3; i++) s += W3[i][j] * delta3[i];
        delta2[j] = s * Activation.derivative(HIDDEN_ACTIVATION, z2[j], a2[j]);;
    }
    for (int j = 0; j < H2; j++) {
        db2[j] += delta2[j];
        for (int k = 0; k < H1; k++) dW2[j][k] += delta2[j] * a1[k];
    }

    // backprop to layer1
    double[] delta1 = new double[H1];
    for (int j = 0; j < H1; j++) {
        double s = 0.0;
        for (int i = 0; i < H2; i++) s += W2[i][j] * delta2[i];
        delta1[j] = s * Activation.derivative(HIDDEN_ACTIVATION, z1[j], a1[j]);
    }
    for (int j = 0; j < H1; j++) {
        db1[j] += delta1[j];
        for (int k = 0; k < d; k++) dW1[j][k] += delta1[j] * x[k];
    }
}
```

Η ενημέρωση των βαρών γίνεται με τη συνάρτηση applyGradients. Εφαρμόζει mini-batch gradient descent. Τα gradients αθροίζονται σε κάθε mini-batch και κανονικοποιούνται με το μέγεθος του batch. Τα βάρη και οι πολώσεις ενημερώνονται με ρυθμό learningRate.

```java
private void applyGradients(double[][] dW1, double[] db1,
                            double[][] dW2, double[] db2,
                            double[][] dW3, double[] db3,
                            double[][] dW4, double[] db4,
                            int batchCount) {
    double scale = learningRate / batchCount;
    for (int i = 0; i < H1; i++) {
        b1[i] -= scale * db1[i];
        for (int j = 0; j < d; j++) W1[i][j] -= scale * dW1[i][j];
    }
    for (int i = 0; i < H2; i++) {
        b2[i] -= scale * db2[i];
        for (int j = 0; j < H1; j++) W2[i][j] -= scale * dW2[i][j];
    }
    for (int i = 0; i < H3; i++) {
        b3[i] -= scale * db3[i];
        for (int j = 0; j < H2; j++) W3[i][j] -= scale * dW3[i][j];
    }
    for (int i = 0; i < K; i++) {
        b4[i] -= scale * db4[i];
        for (int j = 0; j < H3; j++) W4[i][j] -= scale * dW4[i][j];
    }
}
```

Η βασική εκπαίδευση του προγράμματος γίνεται στην συνάρτηση train. Η εκπαίδευση πραγματοποιείται σε επαναλήψεις (epochs), με ανώτατο όριο των maxEpochs ενώ τρέχει για τουλάχιστον 800. Σε κάθε εποχή δημιουργείται πίνακας δεικτών και γίνεται shuffle των δειγμάτων ώστε να αποφεύγεται η δοκιμή των ίδιων παραδειγμάτων κάθε φορά και να βελτιώνεται η σύγκλιση. Το σύνολο εκπαίδευσης χωρίζεται σε mini-batches.

Για κάθε mini-batch:

Μηδενίζονται οι συσσωρευτές των κλίσεων

> Για κάθε δείγμα:
>
> Εκτελείται προώθηση, υπολογίζεται το σφάλμα με MSE και υπολογίζονται οι παράγωγοι με οπισθοδίαδοση και προστίθενται στους συσσωρευτές.

Μετά την επεξεργασία του batch, τα βάρη και οι πολώσεις ενημερώνονται.

Στο τέλος κάθε εποχής εκτυπώνουμε το συνολικό σφάλμα εκπαίδευσης και ελέγχεται το κριτήριο τερματισμού βάσει της μεταβολής του σφάλματος, αφού πρώτα συμπληρωθεί ο ελάχιστος αριθμός εποχών (800).

Η διαδικασία τερματίζει είτε όταν επιτευχθεί σύγκλιση είτε όταν συμπληρωθεί ο μέγιστος αριθμός εποχών, maxEpochs.

```java
public void train(List<double[]> X, int[] labels) {
    int N = X.size();
    if (N == 0) return;

    int epoch = 0;
    double prevTotalError = Double.POSITIVE_INFINITY;

    // prepare index array for shuffling
    Integer[] idx = new Integer[N];

    while (epoch < maxEpochs) {
        epoch++;
        // shuffle
        for (int i = 0; i < N; i++) idx[i] = i;
        List<Integer> idxList = Arrays.asList(idx);
        Collections.shuffle(idxList, rnd);
        idxList.toArray(idx);

        double totalError = 0.0;

        int processed = 0;
        while (processed < N) {
            int end = Math.min(processed + batchSize, N);
            int currentBatchSize = end - processed;

            // zero accumulators
            double[][] dW1 = new double[H1][d]; double[] db1 = new double[H1];
            double[][] dW2 = new double[H2][H1]; double[] db2 = new double[H2];
            double[][] dW3 = new double[H3][H2]; double[] db3 = new double[H3];
            double[][] dW4 = new double[K][H3]; double[] db4 = new double[K];
```

```java
            // process batch
            for (int bi = processed; bi < end; bi++) {
                int ii = idx[bi];
                double[] x = X.get(ii);
                int lab = labels[ii];

                // forward
                double[] y = forward(x);

                double sampleLoss = 0.0;
                for (int k = 0; k < K; k++) {
                    double t_k = (k == lab ? 1.0 : 0.0);
                    double diff = t_k - y[k];
                    sampleLoss += 0.5 * diff * diff;     // MSE
                }
                totalError += sampleLoss;

                // build target one-hot vector t
                double[] t = new double[K];
                t[lab] = 1.0;

                // compute gradients for this example into accumulators
                computeGradientsSingle(x, t, dW1, db1, dW2, db2, dW3, db3, dW4, db4);
            }

            // apply gradients averaged over batch
            applyGradients(dW1, db1, dW2, db2, dW3, db3, dW4, db4, currentBatchSize);

            processed = end;
        } // end processing all batches

        // Print total training error for this epoch (sum over samples)
        System.out.printf(format: "Epoch %d: Total training error = %.8f%n", epoch, totalError);

        // stopping condition: must run at least minEpochs
        if (epoch >= minEpochs) {
            double diff = Math.abs(totalError - prevTotalError);
            if (diff < stopDiffThreshold) {
                System.out.printf(format: "Stopping after %d epochs: |E - E_prev| = %.10f < %.10f%n",
                        epoch, diff, stopDiffThreshold);
                break;
            }
        }

        prevTotalError = totalError;
    } // end epochs

    System.out.printf(format: "Training finished after %d epochs.%n", Math.min(epoch, maxEpochs));
}
```

Τέλος η predict επιστρέφει την κατηγορία με τη μέγιστη έξοδο και η evaluateAccuracy υπολογίζει το ποσοστό των σωστών ταξινομήσεων.

```java
// Predict label index for one sample
public int predict(double[] x) {
    double[] y = forward(x);
    int best = 0;
    double mv = y[0];
    for (int i = 1; i < K; i++) {
        if (y[i] > mv) { mv = y[i]; best = i; }
    }
    return best;
}

// Evaluate accuracy on dataset
public double evaluateAccuracy(List<double[]> X, int[] labels) {
    int N = X.size();
    if (N == 0) return 0.0;
    int ok = 0;
    for (int i = 0; i < N; i++) {
        int p = predict(X.get(i));
        if (p == labels[i]) ok++;
    }
    return (double) ok / N;
}
```

Main.java

Το αρχείο αυτό είναι το σημείο εκκίνησης του προγράμματος και είναι υπεύθυνο για την φόρτωση των δεδομένων, την εκπαίδευση του νευρωνικού δικτύου και την αξιολόγηση της γενικευτικής του ικανότητας.

Η μέθοδος loadFile διαβάζει τα αρχεία train.txt και test.txt, εξάγοντας τα διανύσματα εισόδου και τις αντίστοιχες ετικέτες κλάσης.

Η μέθοδος exportTestClassificationResults χρησιμοποιείται για την ανάλυση του δικτύου με την καλύτερη γενικευτική ικανότητα. Για κάθε δείγμα του συνόλου ελέγχου αποθηκεύει τις συντεταγμένες του και ένα σύμβολο, + αν το δειγμα ταξινομείται σωστά, - αν ταξινομείται λανθασμένα. Το παραγόμενο αρχείο μπορεί να χρησιμοποιηθεί για οπτικοποίηση και σχολιασμό της θέσης των σφαλμάτων, όπως ζητείται στην εκφώνηση.

Στη μέθοδο main φορτώνονται τα σύνολα εκπαίδευσης και ελέγχου, δημιουργείται το πολυεπίπεδο νευρωνικό δίκτυο με συγκεκριμένο seed για καλύτερη σύγκριση μελλοντικών πειραμάτων, ορίζονται οι βασικές παράμετροι εκπαίδευσης και εκτελείται η εκπαίδευση του δικτύου και στη συνέχεια υπολογίζεται η ακρίβεια στο σύνολο ελέγχου, η οποία εκφράζει τη γενικευτική του ικανότητα. Η εξαγωγή των αποτελεσμάτων ταξινόμησης με την χρήση της συνάρτησης exportTestClassificationResults βρίσκεται μέσα σε σχόλια, για να επιλεχθεί από το χρήστη αν θέλει να την χρησιμοποιήσει.

```java
public class Main {

    public static void loadFile(String filename, List<double[]> X_out, List<Integer> labels_out) throws IOException {
        BufferedReader br = new BufferedReader(new FileReader(filename));
        String line;
        while ((line = br.readLine()) != null) {
            line = line.trim();
            if (line.isEmpty()) continue;
            String[] parts = line.split(regex: ",");
            double x1 = Double.parseDouble(parts[0]);
            double x2 = Double.parseDouble(parts[1]);
            int lab = Integer.parseInt(parts[2]);
            X_out.add(new double[]{x1,x2});
            labels_out.add(lab - 1);
        }
        br.close();
    }

    public static void exportTestClassificationResults(
        String outFilename,
        MLP_SDT mlp,
        List<double[]> Xtest,
        int[] labelsTest) throws IOException {

    BufferedWriter bw = new BufferedWriter(new FileWriter(outFilename));

    for (int i = 0; i < Xtest.size(); i++) {
        double[] x = Xtest.get(i);
        int trueLabel = labelsTest[i];
        int predictedLabel = mlp.predict(x);

        char mark = (predictedLabel == trueLabel) ? '+' : '-';

        bw.write(x[0] + "," + x[1] + "," + mark);
        bw.newLine();
    }

    bw.close();
    System.out.println("Test classification results written to: " + outFilename);
}
```

```java
public static void main(String[] args) throws IOException {
    String trainFile = "train.txt";
    String testFile  = "test.txt";

    List<double[]> Xtrain = new ArrayList<>();
    List<Integer> Ytrain = new ArrayList<>();
    List<double[]> Xtest  = new ArrayList<>();
    List<Integer> Ytest  = new ArrayList<>();

    System.out.println(x: "Loading datasets...");
    loadFile(trainFile, Xtrain, Ytrain);
    loadFile(testFile, Xtest, Ytest);
    System.out.printf(format: "Loaded: train=%d, test=%d%n", Xtrain.size(), Xtest.size());

    int[] labelsTrain = Ytrain.stream().mapToInt(Integer::intValue).toArray();
    int[] labelsTest  = Ytest.stream().mapToInt(Integer::intValue).toArray();

    MLP_SDT mlp = new MLP_SDT(seed: 1926);
    mlp.learningRate = 0.02;
    mlp.maxEpochs = 1000;
    mlp.minEpochs = 800;
    mlp.stopDiffThreshold = 0.0001;

    System.out.println(x: "Starting training...");
    mlp.train(Xtrain, labelsTrain);

    System.out.println(x: "Evaluating on test set...");
    double acc = mlp.evaluateAccuracy(Xtest, labelsTest);
    System.out.printf(format: "Test accuracy: %.4f (%.2f%%)%n", acc, acc*100.0);

    //exportTestClassificationResults("test_results_best_model.txt", mlp, Xtest, labelsTest);

}
```

Το πρόγραμμα μεταγλωττίζεται με την εντολή

javac Main.java

και εκτελείται με την εντολή

java Main

Μελέτη προβλήματος ταξινόμησης ΣΔΤ

Στο παρόν μέρος της εργασίας μελετάται το πρόβλημα ταξινόμησης ΣΔΤ με το πρόγραμμα που υλοποιήσαμε. Η ανάλυση επικεντρώνεται στη μελέτη της επίδρασης της αρχιτεκτονικής του δικτύου (αριθμός νευρώνων στα κρυμμένα επίπεδα H1,H2,H3), της συνάρτησης ενεργοποίσης στο τρίτο κρθμμένο επίπεδο καθώς και το μέγεθος του mini-batch κατά την εκπαίδευση.

Αρχικά μας ζητείται να ελέγξουμε διάφορους συνδυασμούς τιμών για τα H1,H2,H3. Έτσι αποφασίσαμε να ελέγξουμε τους εξής συνδυασμούς.

H1	H2	H3
8	8	8
16	8	4
32	16	8

Στη συνέχεια η εκφώνηση μας προτρέπει να χρησιμοποιήσουμε ως συνάρτηση ενεργοποίησης την υπερβολική εφαπτομένη στο πρώτο και στο δεύτερο κρυμμένο επίπεδο και υπερβολική εφαπτομένη ή λογιστική ή relu στο τρίτο. Τρεις επιλογές για το H3 δηλαδή.

Τέλος μας ζητάει να τσεκάρουμε διαφορετικά batch sizes, τα οποία θα έχουν το εξής μέγεθος:

N/10, N/20, N/100, N/200

Δηλαδή το batch size στο πρόγραμμά μας θα πρέπει να δοκιμαστεί με τις τιμές 400, 200, 40 και 20.

3 αρχιτεκτονικές x 3 ενεργοποιήσεις x 4 batch sizes = 36 πειράματα

H1 – H2 – H3	L	Act(H3)	Accuracy
8 – 8 - 8	N/10	Tanh	0.3560
8 – 8 - 8	N/10	Logistic	0.3158
8 – 8 - 8	N/10	Relu	0.3290
8 – 8 - 8	N/20	Tanh	0.3905
8 – 8 - 8	N/20	Logistic	0.3705
8 – 8 - 8	N/20	Relu	0.3755
8 – 8 - 8	N/100	Tanh	0.5393
8 – 8 - 8	N/100	Logistic	0.3613
8 – 8 - 8	N/100	Relu	0.4353
8 – 8 - 8	N/200	Tanh	0.6120
8 – 8 - 8	N/200	Logistic	0.4478
8 – 8 - 8	N/200	Relu	0.5850
16 – 8 - 4	N/10	Tanh	0.3208
16 – 8 - 4	N/10	Logistic	0.3070
16 – 8 - 4	N/10	Relu	0.3338
16 – 8 - 4	N/20	Tanh	0.2920
16 – 8 - 4	N/20	Logistic	0.3533
16 – 8 - 4	N/20	Relu	0.3395
16 – 8 - 4	N/100	Tanh	0.5713
16 – 8 - 4	N/100	Logistic	0.2970
16 – 8 - 4	N/100	Relu	0.4905
16 – 8 - 4	N/200	Tanh	0.6493
16 – 8 - 4	N/200	Logistic	0.4073
16 – 8 - 4	N/200	Relu	0.6978
32 – 16 - 8	N/10	Tanh	0.3818
32 – 16 - 8	N/10	Logistic	0.3310
32 – 16 - 8	N/10	Relu	0.3628
32 – 16 - 8	N/20	Tanh	0.5090
32 – 16 - 8	N/20	Logistic	0.3320
32 – 16 - 8	N/20	Relu	0.4008
32 – 16 - 8	N/100	Tanh	0.8215
32 – 16 - 8	N/100	Logistic	0.5218
32 – 16 - 8	N/100	Relu	0.8148
32 – 16 - 8	N/200	Tanh	0.8920
32 – 16 - 8	N/200	Logistic	0.7370
32 – 16 - 8	N/200	Relu	0.8350

Αρχικά, για όλες τις αρχιτεκτονικές παρατηρείται σαφής βελτίωση της ακρίβειας όταν το μέγεθος του mini-batch μειώνεται. Τα μικρότερα batches οδηγούν σε πιο αποτελεσματική εκπαίδευση βελτιώνοντας αισθητά τη γενίκευση στο σύνολο ελέγχου.

Όσον αφορά την αρχιτεκτονική του δικτύου, τα αποτελέσματα δείχνουν ότι η αύξηση του αριθμού των νευρώνων στα κρυμμένα επίπεδα βελτιώνει τη γενικευτική ικανότητα. Η αρχιτεκτονική 32–16–8 υπερέχει ξεκάθαρα σε σχέση με τις 8-8-8 και 18-8-4, επιτυγχάνοντας τα υψηλότερα ποσοστά επιτυχίας στο σύνολο ελέγχου.

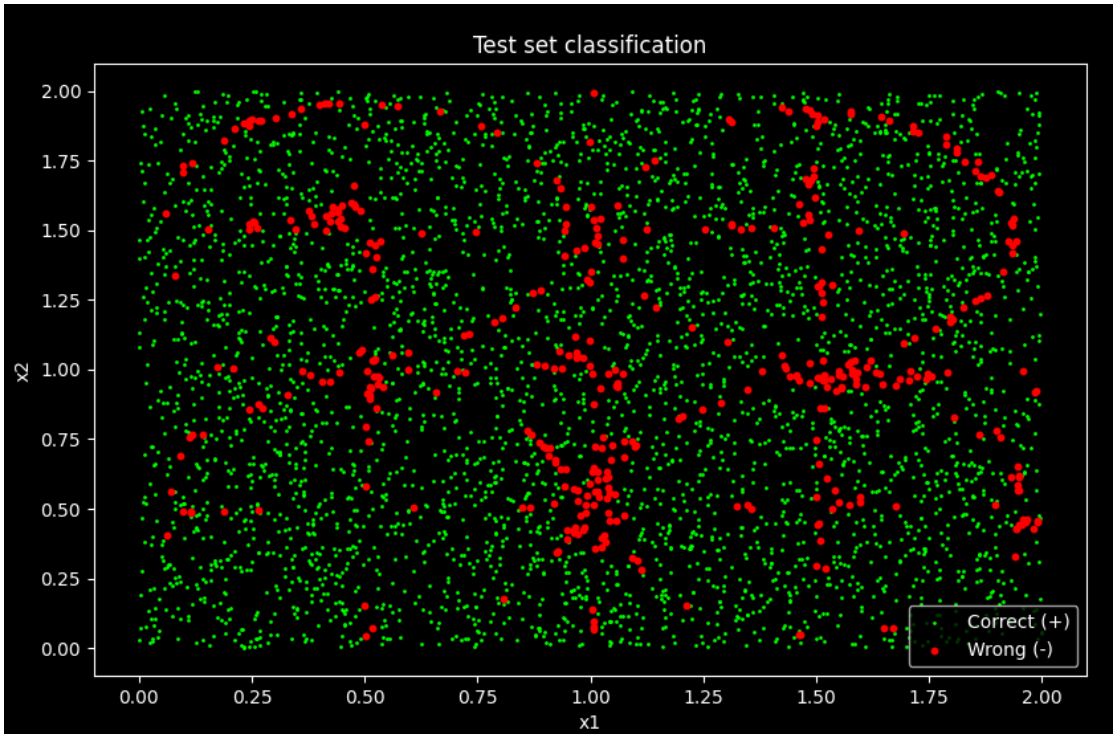
Για τη συνάρτηση ενεργοποίησης στο τρίτο κρυμμένο επίπεδο, η υπερβολική εφαπτομένη παρουσιάζει σχεδόν πάντα τις καλύτερες επιδόσεις, ιδιαίτερα για μικρά batch sizes. Η relu εμφανίζει επίσης πολύ καλές επιδόσεις ενώ η λογιστική υστερεί σχεδόν σε όλες τις ρυθμίσεις.

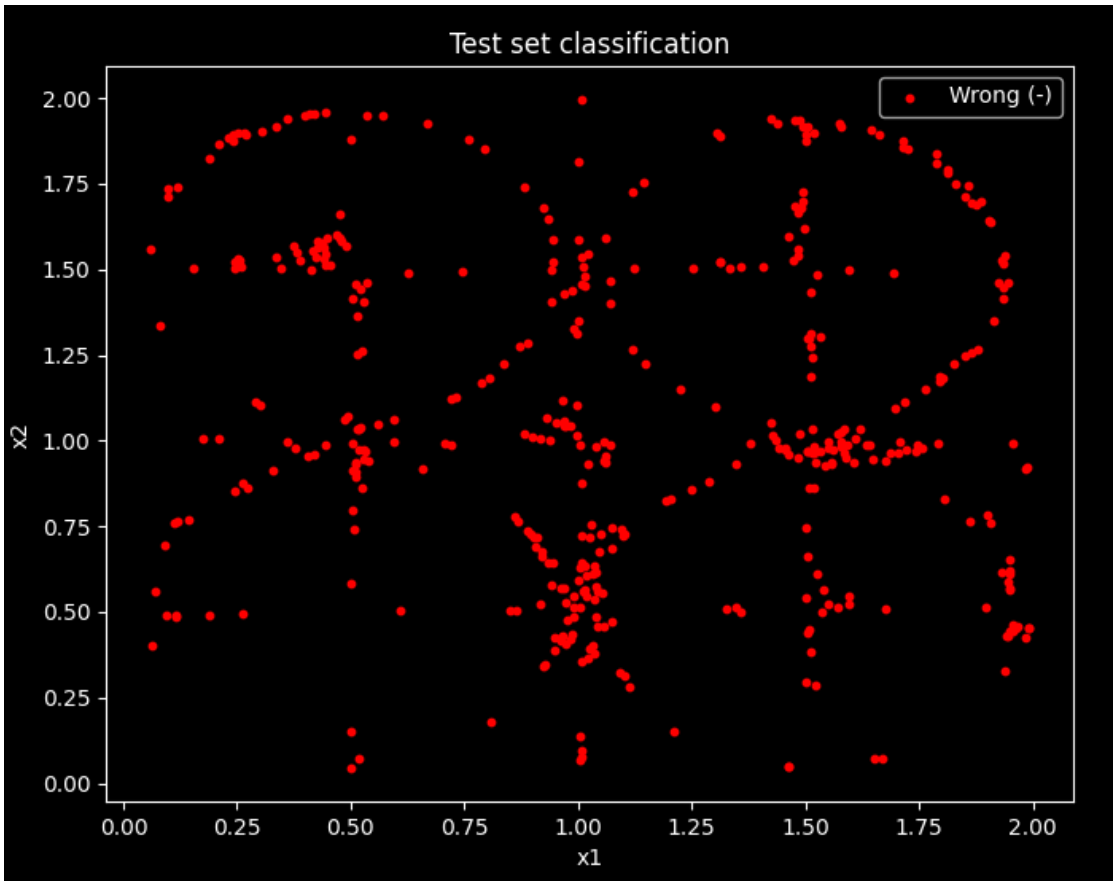
Το καλύτερο συνολικό αποτέλεσμα επιτυγχάνεται με την αρχιτεκτονική 32-16-8, συνάρτηση ενεργοποίησης tanh στο τρίτο κρυμμένο επίπεδο, (στο συγκεκριμένο δίκτυο μάλιστα είναι και στα τρία επίπεδα λόγω της εκφώνησης) και μέγεθος mini-batch L = N/200 = 20, όπου η ακρίβεια στο σύνολο ελέγχου φτανει το 89.20%. Ακολουθεί screenshot της εκτέλεσης του προγράμματος.

```
Epoch 998: Total training error = 314,98105332
Epoch 999: Total training error = 310,44505488
Epoch 1000: Total training error = 307,43316478
Training finished after 1000 epochs.
Evaluating on test set...
Test accuracy: 0,8920 (89,20%)
```

Παρακάτω εμφανίζουμε τα σημεία που ταξινομήθηκαν σωστά και λάθος από το παραπάνω δίκτυο με ακρίβεια 89.20%.

Δημιουργήσαμε ένα νέο python πρόγραμμα απλά για να εμφανίσουμε σε ένα plot τα παραδείγματα. Στο πρώτο διάγραμμα φαίνονται όλα τα παραδείγματα του συνόλου ελέγχου, με πράσινο όσα ταξινομήθηκαν σωστά και με κόκκινο όσα ταξινομήθηκαν λάθος. Στο δεύτερο διάγραμμα απομονώσαμε τα λάθος σημεία.


Test set classification
x2
x1
Correct (+)
Wrong (-)


Test set classification
x2
x1
Wrong (-)

Από τα παραπάνω διαγράμματα καταλαβαίνουμε ότι τα σφάλματα συγκεντρώνονται κυρίως κοντά στα όρια μεταξύ των κατηγοριών, γεγονός που υποδεικνύει ότι το δίκτυο δυσκολεύεται να ξεχωρίσει σημεία που βρίσκονται πολύ κοντά στο όριο απόφασης των κλάσεων. Τα σωστά παραδείγματα τείνουν να βρίσκονται είτε πιο «βαθιά» μέσα στην περιοχή της σωστής τους κατηγορίας είτε μακριά από τα όρια. Αυτό σημαίνει ότι το δίκτυο γενικεύει καλά σε «καθαρά» σημεία αλλά η ακρίβεια μειώνεται όσο πιο κοντά βρισκόμαστε στη «διαχωριστική» γραμμή μεταξύ των κατηγοριών.

Συνεπώς τα δίκτυα με μεγαλύτερο αριθμό νευρώνων μπορούν να διαχωρίσουν πιο καθαρά τις περιοχές, μειώνοντας τα σφάλματα κοντά στα όρια.

Τέλος προσπαθώντας να βρούμε το δίκτυο με την μεγαλύτερη γενικευτική ικανότητα που μπορούμε εργαστήκαμε ως εξής:

Διατηρήσαμε σταθερή τη συνάρτηση ενεργοποίησης του τρίτου κρυφού επιπέδου (tanh) και το μέγεθος των mini-batches (L=N/200), τα οποία είχαν δώσει τα καλύτερα αποτελέσματα στις προηγούμενες δοκιμές. Εξετάστηκαν διάφοροι συνδυασμοί νευρώνων στα τρία κρυφά επίπεδα και καταγράφηκε η αντίστοιχη ακρίβεια ταξινόμησης.

H1	H2	H3	Accuracy
32	16	16	0.8820
32	24	16	0.9273
32	24	24	0.9428
32	32	16	0.9495
32	32	24	0.9478
48	32	16	0.9520
64	32	16	0.9293
64	48	32	0.9508
64	64	32	0.9473
128	64	32	0.9560
128	96	32	0.9570
128	96	64	0.9555

Γενικά σχόλια:

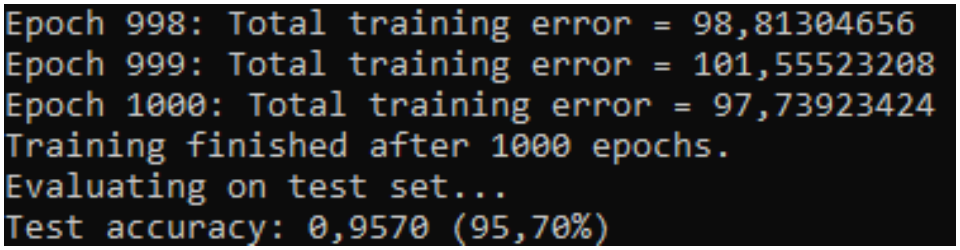
Μετά την επιλογή 64 νευρώνων στο πρώτο επίπεδο, το πρόγραμμα γίνεται αισθητά πιο αργό ενώ το συνολικό σφάλμα εκπαίδευσης μειώνεται πολύ πιο γρήγορα.

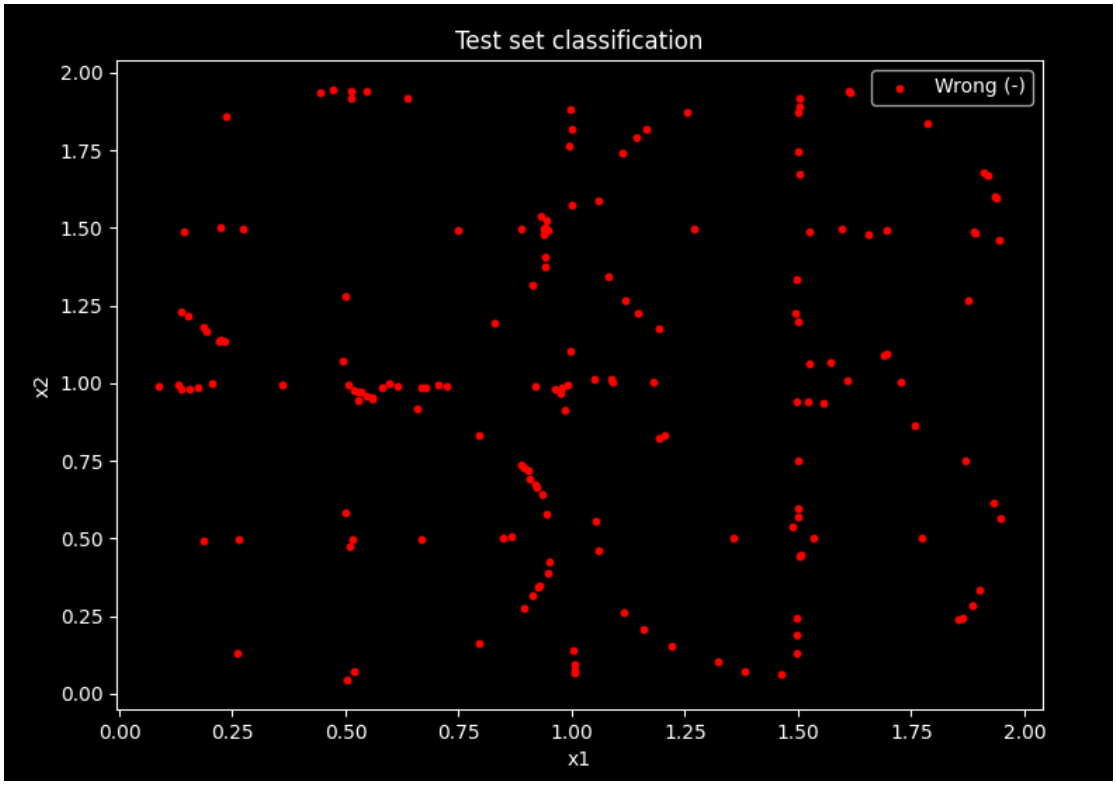
Η αύξηση του αριθμού των νευρώνων συνήθως οδηγεί σε βελτίωση της ακρίβειας ειδικά όταν η αύξηση γίνεται και στα δύο πρώτα κρυφά επίπεδα.

Ωστόσο, η αύξηση των νευρώνων δεν εγγυάται πάντα βελτίωση. Για παράδειγμα για H1=64,H2=32,H3=16 έχουμε χαμηλότερη ακρίβεια από μικρότερες δομές δείχνοντας ότι η κατανομή των νευρώνων ανά επίπεδο παίζει ρόλο.

Το καλύτερο ποσοστό (95.70%) το πετύχαμε για το δίκτυο με H1=128,H2=96,H3=32 αλλά το δίκτυο που είναι ίσως το πιο αποτελεσματικό έχει τον εξής συνδυασμό H1=32,H2=32,H3=16 με ποσοστό 94.95%.

Παρακάτω ακολουθούν screenshots από την εκτέλεση του προγράμματος του δικτύου με το καλύτερο ποσοστό που πετύχαμε (95.70%):





Εύκολα μπορούμε να αντιληφθούμε πως οι λάθος ταξινομήσεις έχουν μειωθεί. Τα σφάλματα περιορίζονται κυρίως στα όρια μεταξύ των κλάσεων γεγονός που υποδεικνύει ότι το δίκτυο έχει μάθει με ακόμα μεγαλύτερη ακρίβει τα μη γραμμικά σύνορα απόφασης. Η οπτική αυτή επιβεβαιώνει ότι η αύξηση της αρχιτεκτονικής πολυπλοκότητας σε συνδυασμό με μικρότερο batch size, οδηγεί σε οσυσιαστική βελτίωση της γενικευτικής ικανότητας του δικτύου.

Τέλος Report