Praktikum zu Informatik 2

Freudenreich, Paulus, Schröder

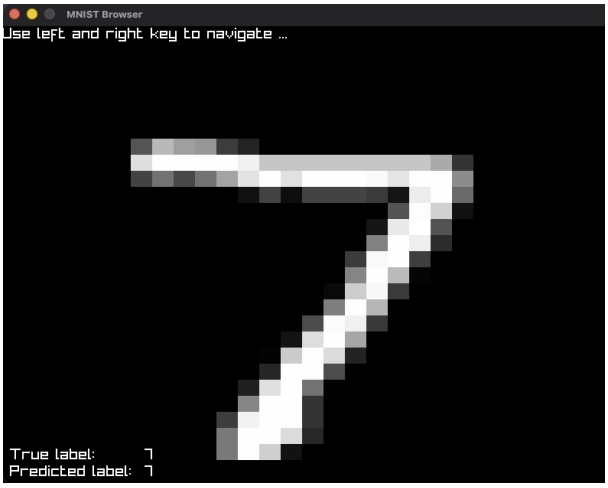
Dateien, Strukturen und dynamische Speicherverwaltung

Im zweiten Praktikumsblock werden wir uns mit erweiterten Konzepten der Sprache C beschäftigen, hauptsächlich mit Strukturen, der dynamischen Speicherverwaltung und der erweiterten Verwendung von Zeigern.

Auch hier werden Sie wieder eine vorgegebene Anwendung vervollständigen. Dieses Mal machen wir eine kleine Exkursion in Richtung **KI** und **Neuronale Netze**.

Die Anwendung erkennt handgeschriebene Ziffern mit Hilfe eines neuronalen Netzes. Das bedeutet, dass sie ein Graustufenbild einer handgeschriebenen Ziffer lädt und die Pixel durch ein neuronales Netz schiebt, an dessen Ende die wahrscheinlichste Ziffer ausgegeben wird. Es handelt sich um eine s.g. **Klassifikation**, d.h. das Netz weist dem Eingabebild eine *Klasse* bzw. ein *Label* zu (in diesem Falle gibt es die Klassen 0 bis 9).

Im folgenden Screenshot sehen wir ein Beispiel der Anwendung. In der Mitte das Eingangsbild und unten links die vom Netzwerk erkannte Ziffer ("Predicted Label") sowie die *wahre* Ziffer ("True Label"). Hier hat hat das neuronale Netz korrekt die Ziffer *7* erkannt.



Die Herausforderung besteht normalerweise darin, ein passendes neuronales Netz zu entwerfen und mittels s.g. Trainingsdaten so zu trainieren, dass es geeignete Merkmale für die Klassifikation aus den Daten lernt. Keine Sorge, das müssen Sie hier *nicht* machen (wie das geht, können Sie in einem späteren Semester lernen). Wir werden uns auf das Ausführen bzw. Nutzen des fertig traininerten Netzes (die s.g. **Inferenz**) beschränken.

Hinweis: Es handelt sich hierbei um ein etwas komplexeres Projekt, bei dem Sie einige Schlüsselfunktionalitäten selbst implementieren müssen. Versuchen Sie, den vorgegebenen

Code als Blackbox zu sehen… Sie können, müssen Ihn jedoch inhaltich nicht vollständig nachvollizehen. Wichtig ist, dass Sie die Schnittstellen verstehen. Dies spiegelt die Arbeitsweise in vielen Softwareprojekten wieder, die häufig so groß sind, dass sie gar nicht von einzelnen Personen komplett überblickt werden können. Damit ist diese Aufgabe auch eine gute Vorbereitung auf Ihre spätere Arbeitswelt.

Vorbereitungen

- Machen Sie sich mit der Projektstruktur vertraut:
  - Welche Module gibt es und was machen sie?
  - Welche Funktionen sind bereits vorhanden, welche fehlen?
- Kompilieren Sie das Projekt mit dem mitgelieferten Makefile und testen Sie das (noch unvollständige) Programm, indem Sie das Target `mnist_initial` bauen.
  - Hinweis: Hierbei wird das (mitgelieferte) Modul `libmnist_complete.a` verwendet. Dies enthält eine lauffähige Version des Programms zum Testen.

```
make mnist_initial // Bauen des Programms
./mnist mnist_test.info2 mnist_model.info2 // Starten des Programms
```

Lineare Algebra für neuronale Netze

Inhalte: Dynamische Speicherverwaltung, Strukturen

Stark vereinfacht gesagt besteht ein neuronales Netz hauptsächlich aus Matrizen mit Parametern (die s.g. *Gewichte*), welche auf bestimmte Art und Weise mit den Eingabedaten multiplizert werden. Daher benötigen wir in unserem Programm die Möglichkeit, Matrizen abzubilden und mit Ihnen zu rechnen.

Aufgaben:

1) Eine Matrix ist definiert durch die Anzahl Zeilen und Spalten sowie einem Speicherbereich mit den eigentlichen Werten. Implementieren Sie an geeigneter Stelle (welche ist das?) eine C-Struktur mit dem Namen `Matrix`, welche eine Matrix auf diese Art beschreibt.
2) Implementieren Sie die fehlenden Funktionen in `matrix.c`. Aus den Funktionsnamen sollten Sie herleiten können, was die einzelnen Funktionen machen. Zusätzliche Hinweise können Sie aus den Unittests ableiten.
   - Eine Besonderheit ist die Funktion `add`. Diese soll zwei Modi unterstützen:
     - **Elementweise Addition**: Hierbei werden die Werte der beiden Matrizen an den jeweiligen Positionen addiert.
     - **Broadcasting**: Dies ermöglicht die Addition einer Matrix mit einem Vektor. Dafür muss die Anzahl der Spalten von einer der beiden Matrizen 1 sein (es muss sich also um einen Spaltenvektor handeln) und die Anzahl der Zeilen beider Matrizen muss übereinstimmen. In diesem Fall werden die Werte aller Elemente einer Zeile der Matrix mit dem jeweiligen Wert des Vektors in der gleichen Zeile addiert. Beispiel:

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix}, \qquad \mathbf{v} = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$$

$$\mathbf{A} + \mathbf{v} = \begin{pmatrix} a_{11} + v_1 & a_{12} + v_1 & a_{13} + v_1 \\ a_{21} + v_2 & a_{22} + v_2 & a_{23} + v_2 \end{pmatrix}.$$

Implementieren Sie beide Modi in der Funktion `add`, sodass sie je nach Beschaffenheit der Eingabeparameter den passenden Modus verwendet.

3) Führen Sie immer wieder die Unittests für das Matrixmodul aus, um Ihren Fortschritt zu überprüfen.

```
make matrixTests && ./runMatrixTests
```

4) Ihre Implementierung ist dann komplett und fehlerfrei, wenn die Unittests alle erfolgreich ausgeführt werden.

Bilder einlesen

Inhalte: Dynamische Speicherverwaltung, Strukturen, Dateien lesen

Da wir in unserem Programm Bildklassifikation durchführen wollen, müssen wir natürlich auch das Laden von Bildern implementieren. Die Bilder sind in der Datei `mnist_test.info2` gespeichert. Sie enthält einen ganzen Serie an Bildern in einem speziellen Format, das Sie einlesen müssen. *Hinweis*: Es handelt sich dabei um eine **Binärdatei**, Sie müssen Sie also im Binarmodus öffnen und byteweise auslesen.

Die Datei beginnt mit dem String `__info2_image_file_format__`, gefolgt von drei Ganzzahlen:

1. **Anzahl der Bilder** *unsigned short*
2. **Breite eines Bildes (Pixel)** *unsigned short*
3. **Höhe eines Bildes (Pixel)** *unsigned short*

Danach folgen für jedes Bild:
- Die **Pixelwerte** (jeweils als Ganzzahlen, 0–255 für Graustufen, *unsigned char*),
- gefolgt von der zugehörigen **Klasse** (Label 0-9, *unsigned char*).

Beispielhafter Aufbau: *Hinweis - Die Leerzeichen dienen nur der besseren Lesbarkeit, in der Datei selbst sind die Werte sequentiell als Bytes geschrieben.*

```
2000 56 56
0 0 0 0 0 ... 0 5
0 0 0 0 0 ... 0 0
...
```

Aufgaben:

In der Datei `imageInput.h` sind Strukturen definiert, die ein einzelnes Bild bzw. eine Bilderserie abbilden. Außerdem ist die Funktion `readImages` deklariert, die eine Bilderserie im oben beschriebenen Format auslesen soll.

1) Anstatt die komplette Funktionalität in `readImages` zu implementieren, überlegen Sie sich, welche Schritte Sie für die gewünschte Funktionalität implementieren müssen und wie Sie diese sinnvoll auf kleinere Hilfsfunktionen aufteilen können. Implementieren diese als Modul-lokale Funktionen (**static**) in `imageInput.c` und fügen geeignete Unittests in `imageInputTests.c` hinzu.
2) Implementieren Sie nun (unter Benutzung Ihrer Hilfsfunktionen) die Funktion `readImages` und stellen Sie sicher, dass die Unittests erfolgreich durchlaufen.

Tipp: Testen Sie Ihre Implementationen schrittweise und führen Sie nach jedem Teilabschnitt einen Commit durch.

```
make imageInputTests && ./runImageInputTests
```

Das Neuronale Netz

Inhalte: Strukturen, Dateien schreiben

Ein neuronales besteht aus verschiedenen Schichten und ihren Parametern. Die Struktur `NeuralNetwork` und die Implementierung in der entsprechenden Quelltextdatei bildet dies ab. Für die passenden Unittests fehlt jedoch noch eine Methode, die eine gültige *Testdatei* erzeugt, mit der die Funktionalität getestet werden kann.

Die Datei beginnt mit dem Identifikationstag `__info2_neural_network_file_format__`, gefolgt von den einzelnen Schichten.

Aufgaben

1) Implementieren Sie die Funktion `prepareNeuralNetworkFile()` in `neuralNetworkTests.c`.
2) Stellen Sie sicher, dass alle Unittests erfolgreich durchlaufen.

```
make neuralNetworkTests && ./runNeuralNetworkTests
```

Feinschliff

- Stellen Sie sicher, dass das gesamte Projekt:
  - ohne Warnungen kompiliert,
  - alle Unit-Tests besteht
- Kompilieren und testen Sie das fertige Programm

```
make mnist
./mnist mnist_test.info2 mnist_model.info2
```

(Optionale Aufgabe) Eigene Ziffern malen

Inhalte: Strukturen, Graphische Programmierung, Inputbehandlung, Raylib

Erweitern Sie den Visualisierungscode um einen Modus, bei dem mit der Maus eigene Ziffern "gemalt" werden können, welche dann vom neuronalen Netz erkannt werden.

Modifizieren Sie dazu den Code in `mnistVisualization.c`. *Hinweis*: Das Programm nutzt Raylib für die graphische Benutzeroberfläche. Schauen Sie sich die Dokumentation bzw. die Beispiele auf dessen Website an.