SoSe 2020

Programmierpraktikum - Block 4

AG Verteilte Systeme

Nikolaus Korfhage, Daniel Schneider

Ziel der folgenden Aufgaben ist es einerseits Grundlagen der Netzwerkprogrammierung (Aufgabe 1) kennen zu lernen und sich andererseits mit Bilderverarbeitung (Aufgabe 2) und Machine Learning (Aufgabe 3) vertraut zu machen. Zu den Aufgaben 2 und 3 finden Sie Vorgaben im Ilias.

Aufgabe 1 - Sockets (10 Punkte)

Das Konzept eines Datenstroms erlaubt es Programmierern, sich bei der Programmierung auf die eigentliche Datenverarbeitung zu konzentrieren und von den Implementierungsdetails der Input-/Output-Operationen zu trennen. Dies erlaubt Programmierern, ihren Quellcode für unterschiedliche Arten von Datenströmen wiederzuverwenden, zum Beispiel für Lese- und Schreibzugriffe auf Dateien, aber auch für Netzwerkkommunikation.

Eine Verbindung über ein Netzwerk wird über so genannte Sockets hergestellt. Ein Socket stellt dabei ein Ende einer Verbindung zwischen zwei Endpunkten dar. Über den `InputStream` und den `OutputStream` des `Socket` werden Bytes von der Gegenstelle empfangen bzw. an diese gesendet.

- **(5 Punkte)** Implementieren Sie zunächst mit Hilfe von Sockets ein einfaches, bidirektionales, interaktives und text-orientiertes Netzwerk-Protokoll[1]. Erstellen Sie eine Klasse `TextClient`, in der Sie in der `main`-Methode eine Verbindung zu einem Server aufbauen. Ihr Programm soll Benutzereingaben von der Konsole lesen, sie zum Server schicken und anschließend die Antwort des Servers auf der Konsole ausgeben. Sie können zum Testen Ihres Clients unseren Echo-Server auf `dsgw.mathematik.uni-marburg.de` und Port 32823 verwenden. Der Echo-Server ist ein einfacher Server basierend auf Sockets, der Verbindungen akzeptiert und der die empfangenen Textnachrichten an den Sender zurückschickt.

- **(5 Punkte)** Implementieren Sie anschließend einen eigenen Echo-Server. Erstellen Sie eine Klasse `EchoServer`, in der Sie einen eigenen Echo-Server, der Verbindungen akzeptiert und die gesendeten Nachrichten an den Sender zurück schickt, implementieren. Zum Akzeptieren von Verbindungen können Sie die Klasse `java.net.ServerSocket` in der `main`-Methode verwenden.

  Nutzen Sie für parallele Verbindungen eine fixe Anzahl von Threads aus einem Thread-Pool. In Java erstellen Sie einen solchen mit der Funktion `Executors.newFixedThreadPool (numberOfThreads)`.

  Sie können ihren Server testen, indem Sie ihn auf localhost mit beliebiger freier Portnummer ≥ 1024 starten und sich anschließend mit ihrem Client auf diese Adresse verbinden.

[1] Kommunikation über Sockets sollte in der Praxis keinesfalls wie hier ungesichert verwendet werden und dient hier nur zur Veranschaulichung.

Aufgabe 2 - Bildfilterung (10 Punkte)

Digitale Bilder können auf vielfältige Weise verarbeitet werden. Eine verbreitete Methode ist die Filterung. Sie erzeugt ein neues Bild basierend auf einer pixelbasierten Manipulation des Originalbildes. Jedes Pixel im neuen Bild wird als Funktion von Pixeln des Originalbildes, normalerweise aus der Umgebung des Zielpixels, berechnet. In Abbildung 1 ist die Anwendung eines Gauß-Filter auf das Originalbild zu sehen. Dabei wird für jedes Pixel im Ergebnisbild eine Filtermatrix auf einen Bereich um das entsprechende Pixel im Originalbild angewendet. Diese Operation wird als Faltung (engl. *convolution*) bezeichnet. Weitere Beispiele für Bildfilter sind Kantendetektion, Schärfung oder Weichzeichner.

In dieser Aufgabe soll zunächst ein Bild eingelesen werden und dieses anschließend in Graustufen konvertiert werden. Auf dem resultierenden 1-Kanal 2D-Array sollen dann Filter angewendet werden. Die entsprechenden Funktionen (Faltung und Konvertierung in Graustufen) sollen dabei selbst implementiert werden d.h. es sollen keine Libraries verwendet werden.

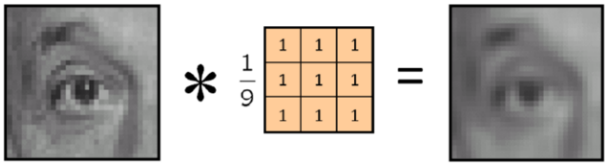


Abbildung 1: Glättung durch Gauß-Filter

- **(1 Punkt)** Das Eingabebild soll zunächst als `BufferedImage` eingelesen werden. Dies können Sie in der `main`-Methode der Klasse `ImageProcessing` erledigen. Jedes Bildpixel ist in `BufferedImage` als `int` gespeichert, hat also 32 Bit zur Verfügung. In Abbildung 2 ist die Aufteilung der Bit-Repräsentation eines RGB Pixel dargestellt. Jeweils ein Byte ist dem Alpha-, Rot-, Grün und Blaukanal zugeordnet und kann Werte zwischen 0 und 255 annehmen[2]. Mit `getRGB(x,y)` kann der RGB Wert eines Pixels im Bild ausgelesen werden. Z.B kann mit `getRGB(0,0)` auf den RGB Wert des Pixel links oben im Bild zugegriffen werden.

ALPHA								RED								GREEN								BLUE							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0

Abbildung 2: Zuordnung der Bit Positionen in `BufferedImage`

- **(1,5 Punkte)** Implementieren Sie die Methode `convertToGrayScaleArray`, die ein `BufferedImage` in ein 2-dimensionales `int`-Array konvertiert, wobei jeder Wert im Array einen Grauwert repräsentiert. Es gibt verschiedene Möglichkeiten, ein RGB-Bild in Graustufen zu konvertieren. Nutzen Sie hier folgende Formel: $G = 0,299 \times R + 0,587 \times G + 0,114 \times B$. Diese Verteilung der Farben spiegelt die entsprechende Farbempfindlichkeit der menschlichen Wahrnehmung wieder.

- **(1,5 Punkte)** Implementieren Sie dann die Methode `convertToBufferedImage`, die aus einem Integer-Array mit Werten für Graustufen ein `BufferedImage` vom Typ `BufferedImage.TYPE_INT_RGB` erzeugt. Schreiben Sie das konvertiere Bild in eine Datei.

[2]Der Alphakanal, der die Transparenz des Pixel festlegt, soll hier nicht berücksichtig werden. D.h. $A = 11111111_2 = 255_{10}$.

Als nächstes sollen auf das Graustufen-Bild Filtermasken (engl. *kernel*) angewendet werden. Filtermasken sind meist quadratische Matrizen in unterschiedlichen Größen. Für digitale Bilder wird die Filtermaske K mit einer diskrete 2-dimensionale Faltung auf das Bild I angewendet[3]:

$$I^*(x,y) = \sum_{i=1}^{n}\sum_{j=1}^{n} I(x-i+a,\ y-j+a)K(i,j), \tag{1}$$

Dabei ist $I^*(x,y)$ das Ergebnispixel, a ist die Koordinate des Mittelpunktes in der quadratischen Filtermaske und $K(i,j)$ ist ein Element der Filtermaske. Um den Mittelpunkt eindeutig definieren zu können, sind ungerade Abmessungen der Filtermasken notwendig. Abbildung 3 veranschaulicht die Anwendung der Filtermaske auf eine bestimmte Position im Eingabebild.

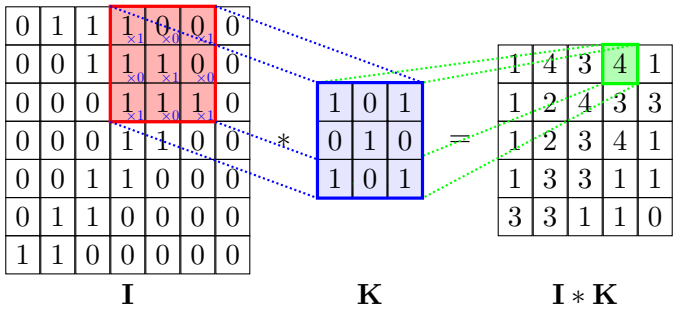


Abbildung 3: Faltung von Filtermaske K mit Eingabebild I

- **(2 Punkte)** Implementieren Sie in der Klasse `Kernel` die Funktion `convolve` die ein Eingabebild als 2D-Array (`int[][]`) entgegennimmt, die Faltung durchführt und das Ausgabebild ebenfalls als 2D-Array vom Typ `int[][]` zurückliefert. Beachten Sie, dass die Größe des Ausgabebildes nach der Faltung entsprechend der Filtergröße kleiner ist als die des Eingabebildes. Die Breite des Ausgabebildes ist dann $w_{I_{out}} = w_{I_{in}} - w_K + 1$, analoges gilt für die Höhe. Sowohl beim Array des Eingabebildes als auch beim Array des Kernel steht die erste Dimension für die `x`-Koordinate und die zweite für die `y`-Koordinate. Der Konstruktor der Klasse `Kernel` ist bereits implementiert: Um die Eingabe der Filtermatrix zu erleichtern wird die übergebene Filtermatrix dort transponiert. So ist es möglich einen Kernel, z.B. $k = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$ mit
```
new Kernel ( new double[][] {{0,  0,  1},
                             {0,  1,  0},
                             {1,  0,  0}} );
```
zu initialisieren.

- **(1 Punkt)** Erstellen Sie mindestens einen JUnit Test für die Klasse `Kernel`. Testen Sie ob die Funktion `convolve` korrekte Ergebnisse liefert, indem Sie prüfen, ob die in Abbildung 3 dargestellte Faltung auf dem rot gekennzeichneten Bildausschnitt tatsächlich den Wert 4 liefert und ob die Breite und Höhe des Ausgabebildes korrekt sind. Erstellen Sie bei Bedarf weitere Tests.

- **(1 Punkt)** Wenden Sie verschiedene Filtermatrizen aus der Klasse `Kernels` auf das in Graustufen konvertierte Beispielbild an, konvertieren Sie es zurück in ein `BufferedImage` und schreiben das Ergebnisbild jeweils in eine Datei. Überlegen Sie sich weitere Filtermatrizen und wenden Sie sie auf das Bild an.

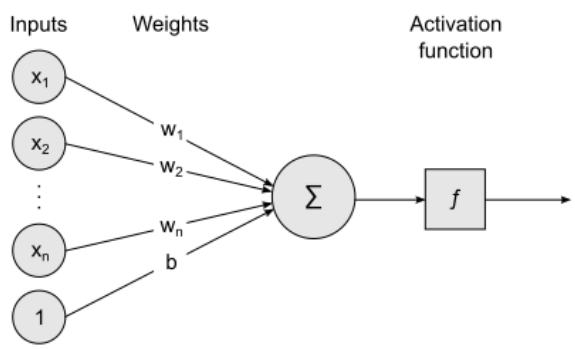
[3] In der Praxis würde die Faltung durch eine Fourier-Transformation erfolgen, sodass sie in ein Produkt überführt wird. Dadurch reduziert sich der Aufwand zur Berechnung einer diskreten Faltung von $\mathcal{O}(n^2)$ auf $\mathcal{O}(n \cdot \log n)$.

- **(2 Punkte)** Die Bilder sind nach der Faltung entsprechend der Kernelgröße kleiner als das Eingabebild. Überlegen Sie sich, wie dieses Problem behoben werden könnte. Implementieren Sie eine Lösung, bei der die Breite und Höhe des Eingabebildes erhalten bleiben.

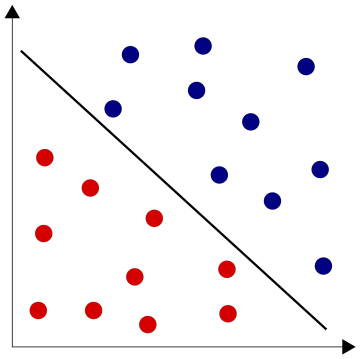
Aufgabe 3 - Perzeptron (10 Punkte)

Das Perzeptron ist ein früher Vorläufer von heutigen künstlichen neuronalen Netzen und stellt ein einfaches neuronales Netz bzw. einzelnes Neuron dar. Als einfacher linearer Klassifikator kann es auf einen Eingabevektor $\mathbf{x}$ angewendet werden und für diesen die Klassen 0 oder 1 voraussagen. Lineare Klassifikatoren trennen Klassen durch eine lineare Hyperebene wie in Abbildung 4b dargestellt. Die schwarze Linie ist dabei die Entscheidungsgrenze des Klassifikators, Beispiele unterhalb werden der einen Klasse zugeordnet, Beispiele oberhalb der anderen.

Ein Perzeptron besitzt lernbare Parameter: den Gewichtsvektor $\mathbf{w}$ und den Bias b. Mathematisch betrachtet ist der Gewichtsvektor der Normalenvektor der Hyperebene. Das in Abbildung 4a dargestellte Perzeptron multipliziert jedes Element des Eingabevektors $\mathbf{x}$ mit dem jeweiligen Element des Gewichtsvektors $\mathbf{w}$ (Skalarprodukt). Die Aktivierungsfunktion f ist bei einem Perzeptron lediglich eine Schwellwertfunktion. Der Einfachheit halber betrachten wir in dieser Aufgabe ein Perzeptron das 2-dimensionale Eingabevektoren binär klassifiziert.



(a) Perzeptron



(b) Linear separierbare Daten

Für eine Eingabe $\mathbf{x}$ erhält man die Ausgabe p des Perzeptron mit folgender Formel:

$$p = \begin{cases} 1 & \sum_i w_i x_i + b > 0 \\ 0 & \text{sonst} \end{cases} \tag{2}$$

Die Parameter des Perzeptron werden zunächst zufällig initialisiert. Um die Parameter den Daten anzupassen, sodass das Perzeptron lernt diese korrekt zu klassifizieren, werden diese iterativ angepasst. Die Parameter werden entsprechend einer Lernregel aktualisiert. Dabei liegen der Lernregel folgende Überlegungen zu Grunde:

1. Ist die Ausgabe 1 (bzw. 0) und soll auch der Wert 1 (bzw. 0) angenommen werden, dann werden die Gewichte nicht geändert.
2. Ist die Ausgabe 0, soll aber 1 sein, dann werden die Gewichte inkrementiert.
3. Ist die Ausgabe 1, soll aber 0 sein, dann werden die Gewichte dekrementiert.

Die Gewichte werden dann entsprechend der folgenden Lernregel aktualisiert:

$$w_i^{\text{neu}} = w_i^{\text{alt}} + \Delta w_i, \qquad \Delta w_i = \alpha \cdot (t - p) \cdot x_i \tag{3}$$

Dabei ist Δw_i die Änderung des Gewichtes w_i, t die gewünschte Ausgabe des Perzeptrons, p die tatsächliche Ausgabe des Perzeptron (wie in Formel 2 berechnet) und α die Lernrate.

Die Lernregel wird nun für jedes Trainingsbeispiel im Datensatz angewendet und die Gewichte entsprechend aktualisiert. Die Anpassung des Bias erfolgt analog, nur das dafür statt mit x_i mit 1 multipliziert wird. Wurde der Datensatz einmal durchlaufen, so wird von vorne begonnen.

Nach einigen Epochen findet das Perzeptron - sofern die Daten linear separierbar sind - eine Lösung.

Für diese Aufgabe sind bereits einige Klassen vorimplementiert:

- `Visualization` soll benutzt werden, um den Lernprozess des Perzeptrons zu visualisieren. Rufen Sie die Methode `update` mit dem Gewichtsvektor und Bias des Perzeptron und der aktuellen Epoche auf. Die Visualisierung wird dann neu gezeichnet. In dieser Klasse müssen keine Änderungen vorgenommen werden.
- `Vector` ist eine einfache Datenstruktur für die Eingabevektoren und den Gewichtsvektor des Perzeptrons.
- `DataPoint` erweitert `Vector` um ein Label, das entweder 0 oder 1 ist.
- `Dataset` liefert einen zufällig generierten Datensatz von Trainingsbeispielen (`DataPoint`) der zum Trainieren, Validieren und Visualisieren genutzt werden soll.
- `Training` soll zum Trainieren des Perzeptrons genutzt werden und enthält die `main`-Methode.
- `Evaluation` soll benutzt werden, um die Qualität des Modells während des Trainings zu beurteilen.

Gehen Sie nun wie folgt vor:

- **(1 Punkt)** Implementieren Sie zunächst die Methoden `dot`, `add` und `mult` in der Klasse `Vector`. Diese werden später für die Berechnung der Ausgabe des Perzeptron und die Anwendung der Lernregel benötigt.
- **(2 Punkte)** Erstellen Sie eine neue Klasse `Perzeptron`, die den Gewichtsvektor $\mathbf{w}$ und den Bias b enthält. Initialisieren Sie diese zufällig zwischen 0 und 1 im Konstruktor. Implementieren Sie eine Methode `predict` die einen Eingabevektor klassifiziert.
- **(3 Punkte)** Implementieren Sie die Methode `train` in der Klasse `Training`. Nutzen Sie die Klasse `Visualization` um die Datenpunkte und die vom Perzeptron gelernte Entscheidungsgrenze zu visualisieren. Aktualisieren Sie die Parameter des Perzeptrons entsprechend der Lernregel in Formel 3. Mischen Sie nach jeder Epoche das Dataset neu (`Collections.shuffle`).
- **(2 Punkte)** Implementieren Sie die Methode `Evaluation.accuracy` und geben Sie nach jeder Epoche im Training die Accuracy (Genauigkeit) des aktuellen Modells auf dem Datensatz aus. Die Accuracy ist die Anzahl der korrekt klassifizierten Beispiele geteilt durch die Anzahl aller Beispiele im Datensatz.
- **(2 Punkte)** Wenn die Daten nicht linear separierbar sind, kann das Perzeptron keine Lösung finden, die alle Punkte korrekt klassifiziert. Dennoch können während des Trainingsprozesses Gewichte gefunden werden die mehr Beispiele korrekt klassifizieren, als das finale Modell. Überlegen Sie sich, was geändert werden muss, damit am Ende des Trainings das beste gefundene Modell verfügbar ist und implementieren Sie Ihre Lösung.