

HCC745 Signal und Bildverarbeitung – WS 2022

Übungsabgabe 2

Lisa-Marie Moser und Caroline Wagner

5. Dezember 2022

Zusammenfassung

2.1 Resampling und Interpolation

In dieser Aufgabe war es das Ziel, die Eingangsbilder mittels Nearest-Neighbour und Bilinear Interpolation ein Resampling durchzuführen und die Bilder somit zu verkleinern oder zu vergrößern.

2.1.1 a)

Um den Faktor der Verkleinerung / Vergrößerung auf 10.0 zu begrenzen wurde zu dem Dialogfeld ein Slider hinzugefügt. Hier kann der Benutzer den Slider zwischen dem Wert 0.0 und 10.0 und in 0.01 Schritten den gewünschten Faktor angeben.

```
1 double scaleFactor = 2.11;
2 double min = 0.0;
3 double max = 10.0;
4
5 GenericDialog gd = new GenericDialog("resample");
6 gd.addSlider("scaleFactor", min, max, scaleFactor, 0.01);
7 gd.showDialog();
8 if(!gd.wasCanceled()){
9     scaleFactor = gd.getNextNumber();
10 }
```

Um das Resampling des Eingangsbildes umsetzen zu können, muss zuerst die neue Größe des Bildes berechnet werden. Hierfür wird die Breite und Höhe des Eingangsbildes mit dem zuvor eingegebenen scaleFactor multipliziert.

```
1 int tgtWidth = (int) Math.round(width * scaleFactor);
2 int tgtHeight = (int) Math.round(height * scaleFactor);
```

Im nächsten Schritt wird die Methode `double[][] getResampledImage(double[][] inImg, int width, int height, int tgtWidth, int tgtHeight, boolean useBilinear)` aufgerufen.

Über den `useBilinear` Parameter wird angegeben, welche Methode zum Resampling benutzt werden soll. Für die Nearest-Neighbour Methode muss also beim Aufruf `false` angegeben werden.

Für die Berechnung der neuen Werte wurde Strategie A angewendet. Auch wenn diese Methode nicht optimal ist, da es hier zur Verschiebungen der Wertebereiche kommt. Der untere Bereich wird hier zu wenig repräsentiert und der obere zu viel. Eine Verbesserung hierzu wäre die Strategie B. Hierbei gehen allerdings im Randbereich die Bilddaten verloren. Die beste Variante wäre die Strategie C, da hier die relative Position im Bildbereich berechnet werden würde.

In folgendem Code-Abschnitt ist die `getResampledImage()` Methode zu sehen. Zuerst wird ein neues double Array angelegt, das die neu berechnete Größe hat. Des Weiteren wird der scaleFactor für die Breite und die Höhe berechnet. Danach wird über jedes Pixel in dem neuen Ausgangsbild iteriert und für jede Position wird der neue Wert berechnet. Durch das Multiplizieren und Dividieren bei der Berechnung der `posX` und `posY` wird sicher gestellt, dass hier die ersten Kommastellen, die zur Berechnung benötigt werden, nicht verloren gehen.

```

1 double[][] getResampledImage(double[][] inImg, int width, int height, int tgtWidth,
2                               int tgtHeight, boolean useBilinear){
3     double[][] returnImg = new double[tgtWidth][tgtHeight];
4     // this is strategy A
5     double scaleFactorW = (double) tgtWidth / width;
6     double scaleFactorH = (double) tgtHeight / height;
7
8     // get all resulting pixels in B from any position in A, from B ==> A, Backward
9     // mapping
10    for (int x = 0; x < tgtWidth; x++){
11        for (int y = 0; y < tgtHeight; y++){
12            // example pos I(2, 3) with scaleFactor 4.0 ==> I'(8,12)
13            double posX = Math.round((x / scaleFactorW) * 10.0) / 10.0;
14            double posY = Math.round((y / scaleFactorH) * 10.0) / 10.0;
15
16            if(useBilinear){
17                returnImg[x][y] = getBilinearInterpolateValue(inImg, width, height,
18                                                               posX, posY);
19            } else {
20                returnImg[x][y] = getNNInterpolatedValue(inImg, width, height, posX,
21                                                               posY);
22            }
23        }
24    }
25
26    return returnImg;
27 }
```

Berechnet wird der neue Wert mit der folgenden Methode. Hier werden die neuen Positionen gerundet und der Wert an der gerundeten Position im Eingangsbild wird zurück gegeben. In den Grenzbereichen muss darauf geachtet werden, dass keine Position außerhalb des Bildes ausgewählt wird.

```
1 public double getNNInterpolatedValue(double[][] inImg, int width, int height, double
2     posX, double posY){
3     // e.g. get value of inImg at position (posX, posY)(2.2, 3.7) ==> inImg[2][4]
4     int posXint = (int) Math.round(posX);
5     int posYint = (int) Math.round(posY);
6
7     //Let's check the range
8     if(posXint < 0){
9         posXint = 0;
10    }
11    if(posYint < 0){
12        posYint = 0;
13    }
14    if (posXint >= width){
15        posXint = width-1;
16    }
17    if (posYint >= height){
18        posYint = height -1;
19    }
20    return inImg[posXint][posYint];
21 }
```



Abbildung 2.1: original



Abbildung 2.2: Resampled 0.5



Abbildung 2.3: original



Abbildung 2.4: Resampled 2.11



Abbildung 2.5: original



Abbildung 2.6: Resampled 2.0



Abbildung 2.7: original



Abbildung 2.8: Resampled 0.7

Bei den Bildern handelt es sich nicht um die originalen Größen, damit diese auf die Seiten passen wurden alle Bilder um den selben Faktor 0.2 skaliert um die Größenverhältnisse beizubehalten.

2.1.2 b)

Für die Bilineare Interpolation müssen zuerst die vier Nachbarn der gewünschten Position ermittelt werden. Diese werden für die Berechnung des neuen Wertes benötigt. Hierbei wird drei Mal Interpoliert zuerst zwischen den beiden X-Achsen und zuletzt in der Y-Achse. Hierfür wurde die folgende Methode angewendet.

```
1 public double getBilinearInterpolateValue(double[][] inImg, int width, int height,
2     double posX, double posY){
3     double result;
4     int posXA = (int) posX;
5     int posYA = (int) posY;
6
7     int posXB = (int) posX + 1;
8     int posYB = (int) posY + 1;
9
10    if(posXA < 0){
11        posXA = 0;
12    }
13    if(posYA < 0){
14        posYA = 0;
15    }
16    if (posXA >= width){
17        posXA = width-1;
18    }
19    if (posYA >= height){
20        posYA = height -1;
21    }
22
23    if(posXB < 0){
24        posXB = 0;
25    }
26    if(posYB < 0){
27        posYB = 0;
28    }
29    if (posXB >= width){
30        posXB = width-1;
31    }
32    if (posYB >= height){
33        posYB = height -1;
34    }
35
36    double resultA = (((inImg[posXB][posYA] - inImg[posXA][posYA]) / 100) * Math.
37        round((posX % 1) * 100)) + inImg[posXA][posYA];
38
39    double resultB = (((inImg[posXB][posYB] - inImg[posXA][posYB]) / 100) * Math.
40        round((posX % 1) * 100)) + inImg[posXA][posYB];
41
42    result = (((resultB - resultA) / 100) * Math.round((posY % 1) * 100)) + resultA;
43
44    return Math.round(result);
45 }
```



Abbildung 2.9: original



Abbildung 2.10: Resampled 0.5



Abbildung 2.11: original



Abbildung 2.12: Resampled 2.11



Abbildung 2.13: original



Abbildung 2.14: Resampled 2.0



Abbildung 2.15: original



Abbildung 2.16: Resampled 0.7

Wird nun die Nearest-Neighbour Strategie mit der Biliniearen Interpolation verglichen, ergibt sich folgendes Differenzbild. Bei dem verkleinerten Bild „Boats“ gab es kaum bis keine Differenzen, bei den anderen hingegen sind die Differenzen schön zu sehen.

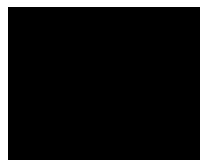


Abbildung 2.17: Differenzbild



Abbildung 2.18: Differenzbild



Abbildung 2.19: Differenzbild



Abbildung 2.20: Differenzbild

Die Unterschiede in den skalaren Pixelwerten entstehen dadurch, dass bei der Bilinearen Interpolation auch die Nachbarn zur Berechnung des neuen Wertes herangezogen werden.

2.1.3 c)

Für die Checkerboard Darstellung wurde die Methode aus Übung 1 wiederverwendet. Mit der Anpassung, dass die einzelnen Kacheln durch eine weiße Linie getrennt werden. Um die Unterscheidung der Bilder zu erleichtern.

Die Nearest-Neighbour Strategie ist von der Laufzeit her schneller, da weniger Berechnungen durchgeführt werden müssen. Allerdings ist die Qualität der Bilinearen Interpolation besser, was am besten in den Kantenbereichen zusehen ist. Bei den Bildern „Boats“ und „Bridge“ ist der Unterschied kaum zu erkennen. Bei dem Bild „Lena“ ist der Unterschied sehr schön an der Schulter zu sehen. Was im genaueren Vergleich von 2.16 und 2.8 zu sehen ist.

```

1 public static double[][] checkerboard(double [][] oldImg, double[][] newImg, int
2   width, int height){
3   double[][] returnImg = new double[width][height];
4
5   for (int x = 0; x < width; x++){
6     for (int y = 0; y < height; y++){
7       if(x == Math.round(width / 2) || y == Math.round(height / 2)){
8         returnImg[x][y] = 255;
9       } else if (x < Math.round(width / 2) && y < Math.round(height / 2) || x > Math.
10         round(width / 2) && y > Math.round(height / 2) ){
11         returnImg[x][y] = oldImg[x][y];
12       } else {
13         returnImg[x][y] = newImg[x][y];
14       }
15     }
16   return returnImg;
17 }
```



Abbildung 2.21: Differenzbild



Abbildung 2.22: Differenzbild



Abbildung 2.23: Differenzbild



Abbildung 2.24: Differenzbild

2.2 Klassifizierung mittels Kompression

2.2.1 a)

Für diese Aufgabe wurden 8 verschiedene Textfiles erstellt, die jeweils das Märchen Rapunzel als Text in 8 verschiedenen Sprachen enthalten. Dies sind die repräsentativen Vergleichsdatensätze. Als unbekannter Text wurde der gestiefelte Kater in Deutsch in ein weiteres Textfile gespeichert. Um nun Klassifizieren zu können, welche Sprache der unbekannte Text hat wurde zuerst jedes einzelne der Vergleichsdatensätze in einen Zip-Ordner gespeichert. Danach wurden weitere Textfiles erstellt, wo nach dem jeweiligen Vergleichsdatensatz der unbekannte Text hinzugefügt wurde. Aus diesen einzelnen Files wurde wieder jeweils ein Zip-Ordner erstellt. Die Größe jeder einzelnen Zip-Ordners wurde in eine Excel-Tabelle zum Vergleich eingetragen. Danach wurde der Unterschied zwischen den verschiedenen Zip-Ordnern berechnet, nämlich zwischen den jeweiligen einzelnen Zip-Ordnern der Vergleichsdatensätze und den dazugehörigen zusammengefügten Zip-Ordner. Der Unterschied der beiden war bei dem Deutschen Textfile am geringsten. Der Beispieltext ist also Deutsch.

	Klassifikation Texte						Rang
	einzelne Zip		gemeinsam Zip		Unterschied		
	KB	Bytes	KB	Bytes	KB	Bytes	
Chinesisch	2,01	2.065	5,98	6.124	3,97	4.059	8.Rang
Deutsch	2,04	2.097	5,53	5.666	3,49	3.569	1.Rang
English	1,86	1.911	5,54	5.675	3,68	3.764	2.Rang
Finish	1,96	2.008	5,66	5.799	3,7	3.791	4.Rang
Griechisch	2,44	2.505	6,28	6.436	3,84	3.931	6.Rang
Italienisch	1,94	1.996	5,64	5.784	3,7	3.788	3.Rang
Japanisch	2,01	2.063	5,93	6.077	3,92	4.014	7.Rang
Polnisch	2,07	2.121	5,78	5.924	3,71	3.803	5.Rang

Abbildung 2.25: Berechnung des Unterschiedes der Unterschiedlichen Zip-Ordner.

Hierbei ist wichtig, dass die Textfiles ungefähr die selbe Länge haben bzw.. sodass der gewählte Text die Sprache und ihr Alphabet sehr gut repräsentiert. Wie an der Tabelle abzulesen ist, haben die romanischen Sprachen noch am wenigsten Unterschied. Chinesisch und Japanisch haben hingegen am schlechtesten abgeschnitten, da diese auch ein anderes Alphabet benutzen. Auch Griechisch hat andere Schriftzeichen und ist daher auch unter den letzten drei.

Im nächsten Schritt wurde das Vorgehen mit dem selbigen Gestiefelten Kater Text, allerdings in den anderen Sprachen übersetzt, wiederholt. Die Ergebnisse wurden ebenfalls in eine weitere Tabelle eingetragen.

	Chinesisch	Deutsch	English	Finish	Griechisch	Italienisch	Japanisch	Polnisch
Chinesisch	1	6	7	5	3	7	2	4
Deutsch	8	1	2	4	6	3	7	5
English	8	4	1	2	7	5	6	2
Finish	8	3	4	1	7	4	6	2
Griechisch	3	6	8	5	1	7	2	4
Italienisch	8	4	5	2	7	1	6	2
Japanisch	2	6	8	4	3	7	1	5
Polnisch	8	3	4	2	7	4	6	1

Abbildung 2.26: Alle Ergebnisse in Rängen.

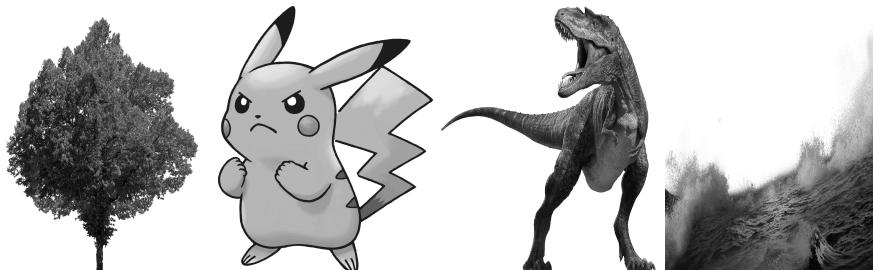
Hierbei ist gut zu erkennen, dass Chinesisch, ausgenommen von den anderen Sprachen mit Schriftzeichen immer den 8.Rang belegte. Sowie Griechisch den 7.Rang und Japanisch fast ausschließlich auf dem 6 Rang gelandet ist. Erstaunlich ist, dass jede Klassifizierung die richtige Antwort liefert hat. Das Ergebnis ist also statistisch signifikant. Umso mehr diskriminierende Klassen es gibt, desto besser ist die Klassifikationsgenauigkeit, da es mehr Beispiele gibt mit denen der unbekannte Text verglichen werden kann. Die genauen Berechnungen sind in dem beigelegtem Excel-File nach zu schlagen.

2.2.2 b)

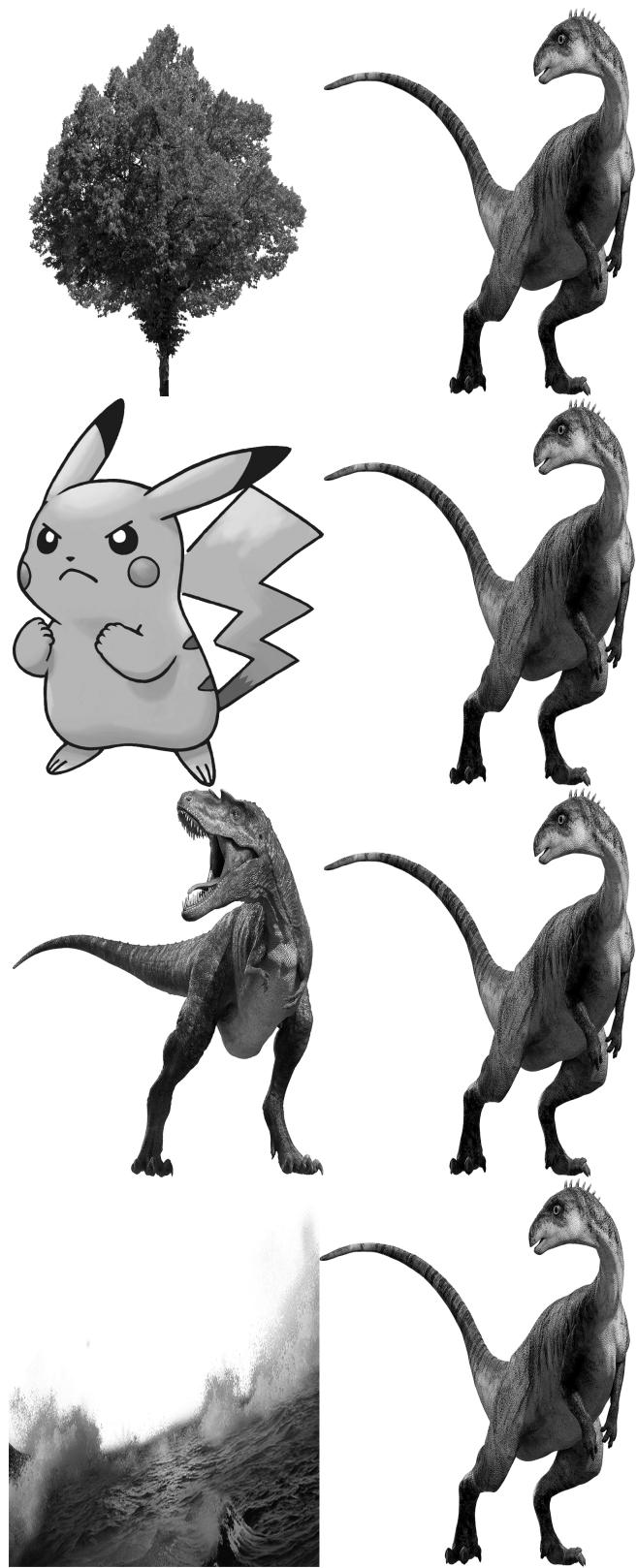
Als Vorbereitung zur Klassifikation wurden zuerst alle Bilder auf die selbe Größe zugeschnitten. Als nächsten wurden alle Bilder in 8-Bit Graustufen Bilder umgewandelt, damit alle den selben Farbraum aufweisen. Bei der Klassifikation mit Bildern ist ebenfalls darauf zu achten, das alle Bilder verlustfrei sind und nicht bereits z.B. durch JPEG komprimiert wurden. An sonst kann es zu großen Unterschieden in der Größe kommen. Im nächsten Schritt wurden die einzelnen Bilder gezipt.



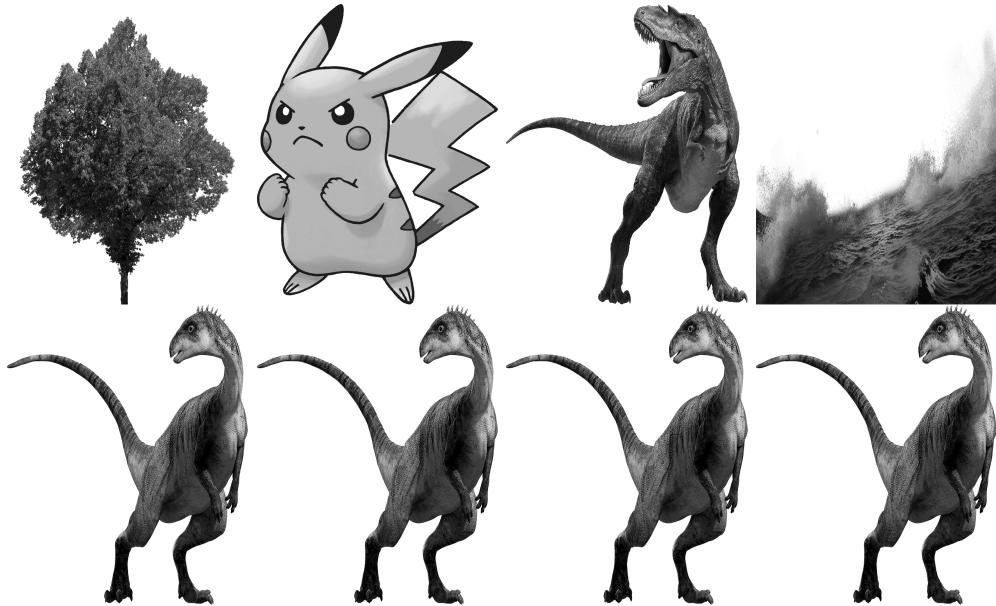
Abbildung 2.27: Unbekannte Klasse



Als nächsten Schritt wurden die Bilder nebeneinander gestellt und als neues Bild gespeichert. Nun wurden für alle Zip-Ordner die jeweiligen Größen in eine Tabelle geschrieben.



Das Ergebnis war hier mit dem 1.Rang der Baum und mit dem 2.Rang erst der T-Rex. Als zweites wurde dann Versucht, die Bilder untereinander anzugeordnen. Um zu sehen, ob die Position eventuell ausschlaggebend für das Ergebnis ist.



Bei diesem Versuch war der T-Rex nur auf dem 3.Rang. Laut der Klassifikation ist der Dinosaurier nun eine Welle. Ich habe die Größen der Zip-Ordner mehrmals überprüft und es sind keine Tippfehler in der Tabelle vorhanden. Meine Vermutung, warum der Dinosaurier hier nicht als solches erkannt wird ist der Unterschied zwischen den beiden Dinosaurier Rassen. Wobei ich bewusst zwei sehr ähnliche Dinosaurier ausgewählt habe. Eventuell sind die anderen Vergleichsklassen auch zu nah am Unbekannten Dinosaurier.

	Klassifikation Bilder nebeneinander						Rang
	einzelne Zip		gemeinsam Zip		Unterschied		
	KB	Bytes	KB	Bytes	KB	Bytes	
Pikachu	199	204.144	439	449.789	240	245.645	3.Rang
Tree	335	343.330	554	568.316	219	224.986	1.Rang
T-Rex	239	245.291	459	470.948	220	225.657	2.Rang
Wave	362	371.132	612	627.405	250	256.273	4.Rang

	Klassifikation Bilder untereinander						Rang
	einzelne Zip		gemeinsam Zip		Unterschied		
	KB	Bytes	KB	Bytes	KB	Bytes	
Pikachu	199	204.144	721	739.315	522	535.171	2.Rang
Tree	335	343.330	917	939.767	582	596.437	4.Rang
T-Rex	239	245.291	772	790.849	533	545.558	3.Rang
Wave	362	371.132	812	832.149	450	461.017	1.Rang

Abbildung 2.28: Berechnung des Unterschiedes der Unterschiedlichen Zip-Ordner.

Ebenfalls ausprobiert wurde, dass die zu vergleichenden Bilder nicht in gemeinsam in ein Bild sondern in ein Textfile zusammengefügt wurden. Hierbei wurden wieder die verschiedene Referenzbilder herausgesucht. diesmal von jeder Klasse 4 ähnlicher Bilder, die mit 4 verschiedenen Bildern jeder Klasse verglichen wurden.



Abbildung 2.29: Klasse Forest

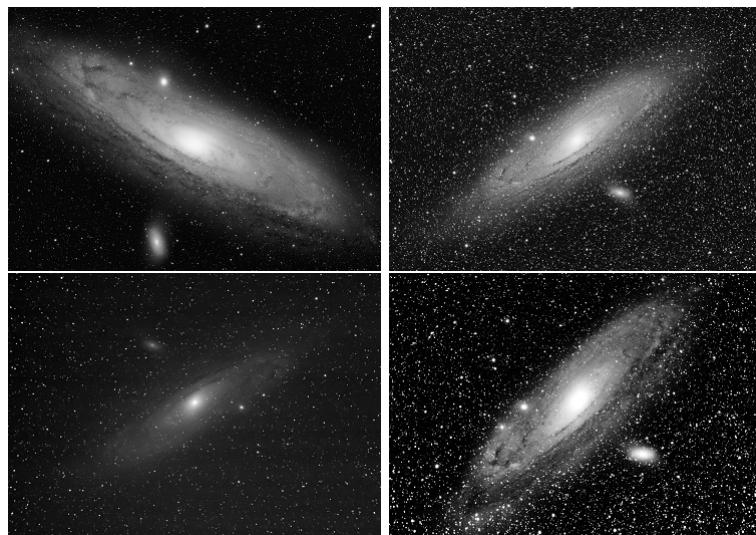


Abbildung 2.30: Klasse Galaxy



Abbildung 2.31: Klasse Japan



Abbildung 2.32: Klasse Wave



Abbildung 2.33: Die zu klassifizierenden Bilder.

Zip einzeln					Unterschied				
	Forest	Galaxy	Japan	Wave		Forest	Galaxy	Japan	Wave
1	136	106	110	98,9					
2	130	129	108	100					
3	130	94,4	109	98					
4	134	124	99,8	89,4					
					4.Rang	3.Rang	1.Rang	2.Rang	
Forest 1	263	243	238	239	127	107	102	103	
Forest 2	257	236	232	233	127	106	102	103	
Forest 3	257	237	232	233	127	107	102	103	
Forest 4	261	241	236	237	127	107	102	103	
					4.Rang	3.Rang	1.Rang	2.Rang	
Galaxy 1	233	213	208	209	127	107	102	103	
Galaxy 2	256	236	231	232	127	107	102	103	
Galaxy 3	221	201	196	197	126,6	106,6	101,6	102,6	
Galaxy 4	251	230	226	226	127	106	102	102	
					4.Rang	3.Rang	1.Rang	2.Rang	
Japan 1	238	217	213	213	128	107	103	114,1	
Japan 2	236	215	312	211	128	107	204	111	
Japan 3	237	216	212	212	128	107	103	114	
Japan 4	227	206	201	202	127,2	106,2	101,2	112,6	
					4.Rang	2.Rang	1.Rang	3.Rang	
Wave 1	226	205	201	201	127,1	106,1	102,1	102,1	
Wave 2	227	207	202	203	127	107	102	103	
Wave 3	225	204	200	200	127	106	102	102	
Wave 4	216	196	191	192	126,6	106,6	101,6	102,6	
					4.Rang	3.Rang	1.Rang	2.Rang	

Abbildung 2.34: Berechnung des Unterschiedes der Unterschiedlichen Zip-Ordner.

Die Ergebnisse hierbei waren nicht sehr gut. Die einzige Klasse die Erkannt wurde war die Klasse Japan. Die Ergebnisse werden wieder in folgender Tabelle ausgegeben. Was mir ein wenig komisch vorkommt, ist das bei allen Berechnungen Japan mit dem 1.Rang herauskommt. Und die Klasse Forest immer den 4.Rang belegt. Ich habe die Daten mehrmals überprüft und auch die Daten erneut zusammengefügt mit dem selben Ergebnis. In diesem Fall sind die Testergebnisse nicht statistisch signifikant. Hier gilt selbiges, wie bei den Texten, umso mehr diskriminierenden Klassen es gibt desto besser ist die Klassifikationsgenauigkeit. Da es mehr Beispiele gibt mit denen das unbekannte Bild verglichen werden kann.

Auch hier sind die genauen Berechnungen in dem beigelegtem Excel-File nach zu schlagen.

2.3 Kompression und Code-Transformation

2.3.1 a)

a) dabbbabababbbb aaaaababababcc dd

Aktuell	Next	WB?	WB	OUT
d	a	x	d \rightarrow 256	<100> d
a	b	x	ab \rightarrow 257	<97> a
b	b	x	bb \rightarrow 258	<98> b
b	a	x	ba \rightarrow 259	<98> b
a	b	j	aba \rightarrow 260	<257> ab
a	b	j	abab \rightarrow 261	<260> aba
b	b	j	bbb \rightarrow 262	<258> bb
b	a	j	baa \rightarrow 263	<259> ba
a	a	x	aa \rightarrow 264	<97> a
a	a	j	aab \rightarrow 265	<264> aa
b	a	j	bab \rightarrow 266	<259> ba
b	a	j	babc \rightarrow 267	<266> bab
c	c	x	cc \rightarrow 268	<99> c
c	d	x	cd \rightarrow 269	<99> c
d	d	x	dd \rightarrow 270	<100> d
d	-	-	-	<100> d

Ausgabe: <100> <97> <98> <98> <257> <260> <258> <259> <97>
<264> <259> <266> <99> <99> <100> <100>

a)	$25 \times 16\text{bit} = 400$	$C_F = \text{alte Datenmenge / neue Datenmenge}$
	$16 \times 16\text{bit} = 256$	$400 / 256 \approx \underline{\underline{1,56}}$

Abbildung 2.35: Beispiel A

2.3.2 b)

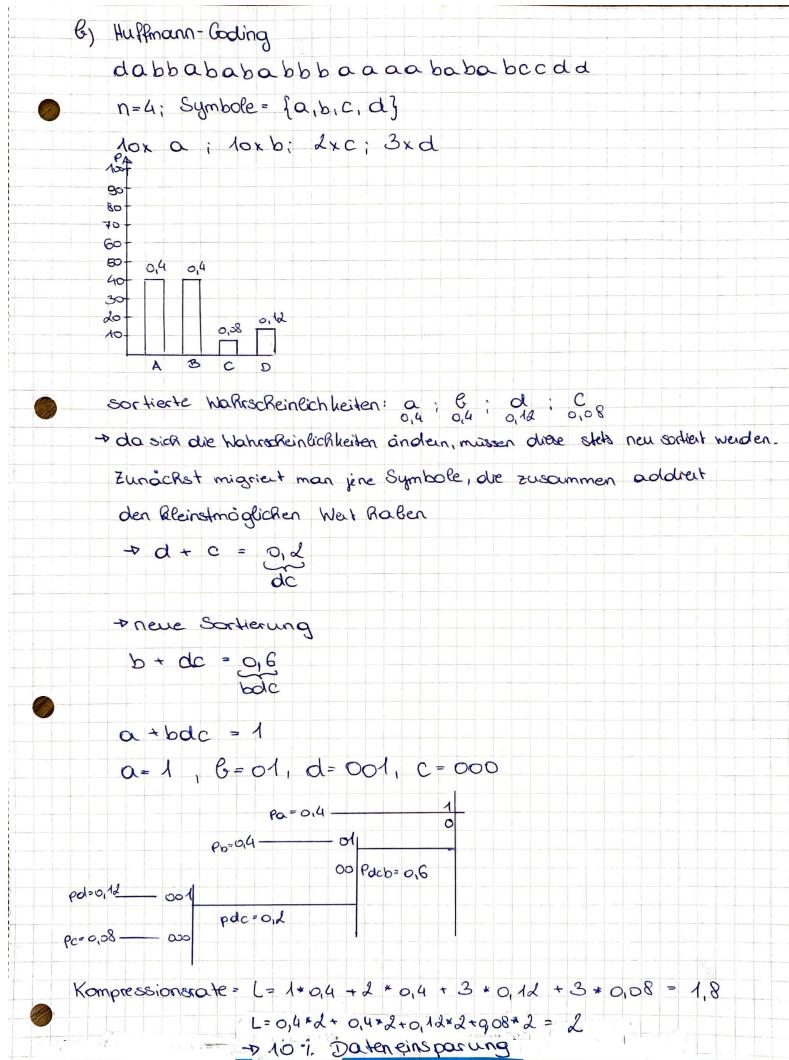


Abbildung 2.36: Beispiel B 1/2

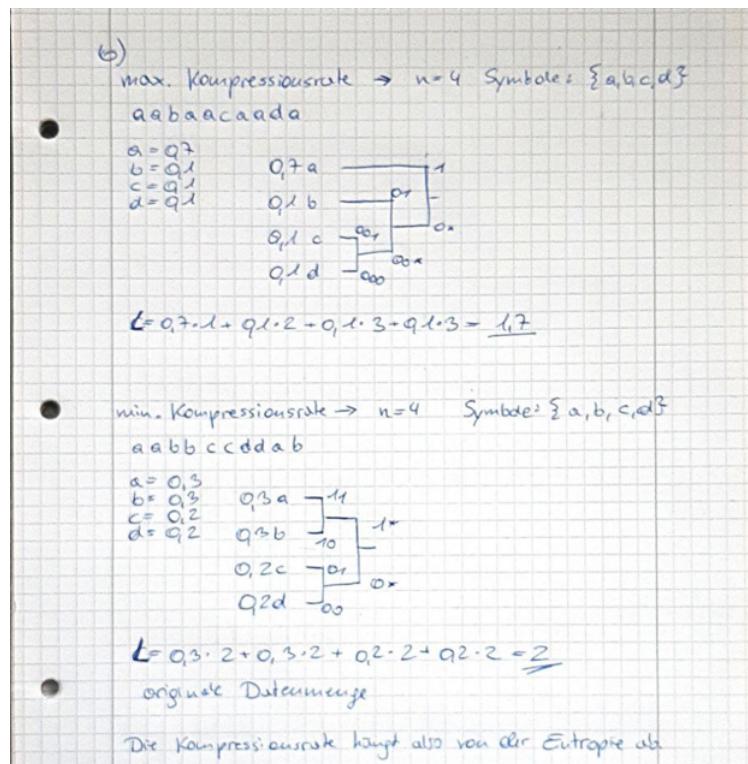


Abbildung 2.37: Beispiel B 2/2

2.3.3 c)

9) Runlength Coding

111101010111110000011110001010

● 30 Stellen; $n=2$; Symbole = {0;1}

Vorkommen der Symbole:

4×1 ; 1×0 ; 1×1 ; 1×0 ; 1×1 ; 1×0 ; 5×1 ; 5×0 ; 4×1 ; 3×0 ; 1×1

1×0 ; 1×1 ; 1×0

→ 4 1 1 1 1 1 5 5 4 3 1 1 1 1

→ 14 Stellen; 4 Symbole; Symbole = {1; 3; 4; 5}

(Man könnte an einer Stelle nach einer Null für 0x0 setzen, muss man aber nicht.)

● Beispiel mit 3 Symbolen:

Symbole = {0, 1, 2}

11110011010220112011022011100001

32 Stellen; 3 Symbole

4×1 ; 2×0 ; 2×1 ; 1×0 ; 1×1 ; 1×0 ; 2×2 ; 1×0 ; 2×1 ; 1×2 ; 1×0

2×1 ; 1×0 ; 2×2 ; 1×0 ; 3×1 ; 4×0 ; 1×1

→ 4 2 2 1 1 1 2 1 2 1 1 2 1 2 1 3 4 1

→ 18 Stellen; 4 Symbole

● → Da es nicht mehr abwechselnd 0 und 1 ist, ist es nicht klar, welche Wert für welche Zahl steht.

Wann kommt 1, wann kommt 0 oder 2;

Bei aufsteigenden, sortierten Werten stimmt die Reihenfolge nicht mehr überein.

Abbildung 2.38: Beispiel C

2.3.4 d)

d) Entropie

22 1111 2266 11122 3333 45645112111

Symbole = {1; 2; 3; 4; 5; 6} 30 Stellen

$$7 \times 2 \rightarrow p = 0,233$$

$$12 \times 1 \rightarrow p = 0,4$$

$$4 \times 3 \rightarrow p = 0,133$$

$$4 \times 4 \rightarrow p = 0,06667$$

$$2 \times 5 \rightarrow p = 0,06667$$

$$3 \times 6 \rightarrow p = 0,1$$

$$H = 0,233 \cdot \log(0,233) + 0,4 \cdot \log(0,4) + 0,133 \cdot \log(0,133) + 0,06667 \cdot \log(0,06667) + 0,06667 \cdot \log(0,06667) + 0,1 \cdot \log(0,1) = 0,69$$

→ Entropie ist relativ hoch, da sie viel Informationsgehalt enthält.

Bei welcher 10-stelligen Sequenz ist die Entropie maximal?

abcdefghijklm

Symbole = {a,b,c,d,e,f,g,h,i,j} 10 Stellen

$$1 \times a \rightarrow p = 0,1$$

$$1 \times b \rightarrow p = 0,1$$

$$1 \times c \rightarrow p = 0,1$$

$$1 \times d \rightarrow p = 0,1$$

$$1 \times e \rightarrow p = 0,1$$

$$1 \times f \rightarrow p = 0,1$$

$$1 \times g \rightarrow p = 0,1$$

$$1 \times h \rightarrow p = 0,1$$

$$1 \times i \rightarrow p = 0,1$$

$$1 \times j \rightarrow p = 0,1$$

$$H = 0,1 \cdot \log(0,1) + 0,1 \cdot \log(0,1) = -1$$

→ Entropie ist optimal hoch, da gleiche Wahrscheinlichkeit aller Symbole und viel Informationsgehalt

Abbildung 2.39: Beispiel D

Bei welcher 10-stelligen Sequenz ist die Entropie minimal?

aaaaaaa Symbole- $\{a\}$ 10stellen

$$10 \times a \rightarrow p = 1$$

$$H = 1 \cdot \log(1) = 0$$

→ Entropie ist minimal, da wenig Informationsgehalt

Abbildung 2.40: Beispiel D

Zusammenfassung und Anmerkungen

Für die Übung 2 haben wir insgesamt etwa 25 Stunden gebraucht.