Vergleich von Autoencoder und PCA zur Dimensionsreduktion am CIFAR-100 Datensatz

Ahmet Efe[1]
Betreuer: Dr. Daniel Kondermann [2]

Zusammenfassung
In diesem Bericht werden die beiden Dimensionsreduktionsmethoden *Autoencoder* und *PCA* verglichen. Dieser Vergleich basiert auf einer theoretischen Beschreibung beider Ansätze und einer Analyse des Rekonstruktionsfehlers sowie einer Auswertung der Suche nach den *Nächste-Nachbarn.* Sowohl der *Autoencoder* als auch die *PCA* konnten keine guten Ergebnisse bei der Suche nach den *Nächste-Nachbarn* liefern, da sie sich auf visuelle Merkmale konzentrieren, jedoch semantische Merkmale für gute Ergebnisse nötig sind. Zudem werden die Bilder mithilfe des *Autoencoders* geclustert, auch diese Anwendung war nicht erfolgreich. Als Eingabedaten wurde der *CIFAR-100* Datensatz ausgewählt.

Code zu finden unter:
`https://github.com/ahmetefe98/APAutoencoder`

[1] *im224@stud.uni-heidelberg.de*
[2] *daniel@kondermann.de*

Inhaltsverzeichnis

1. Einleitung

Die ursprüngliche Aufgabe dieses Fortgeschrittenenpraktikums war es, Eingabebilder mit einem *Autoencoder* zu clustern und die Ergebnisse zu analysieren. Als Eingabebilder wurde der *CIFAR-100* Datensatz [1] gewählt. Detaillierte Informationen zu den Eingabedaten sind zu finden im *Kapitel 2.1 Eingabedaten*.

Der Hintergrund dieser Aufgabenstellung ist, dass traditionelle *"k-means"* Algorithmen, die sehr effizient Clusterungen durchführen, bei hohen Eingabedimensionen wie beim *CIFAR-100* Datensatz mit 1024 Pixeln an ihre Grenzen geraten. Daher wird ein *Autoencoder* trainiert und im Anschluss auf den *Encoder* ein *"Clustering-Schicht"* gebaut, der die Ausgabe des *Encoders* einem Cluster zuweisen soll. Die Ausgabe des *Encoders* wird benutzt, da so die Daten eine geringere Dimension haben [2].

Nach den ersten Versuchen wurde klar, dass diese Idee mit dem Clustern mithilfe des *Autoencoders* nicht erfolgreich ist, daher war die neue Aufgabe dieses Fortgeschrittenenpraktikums die *Nächste-Nachbarn* mithilfe des *Autoencoders* zu bestimmen.

Das Trainieren des *Autoencoders* und das Bestimmen der *Nächste-Nachbarn* mithilfe des *Autoencoders* ist sehr zeitaufwendig, daher wird die Genauigkeit des

Autoencoders mit der einer *PCA* [1] verglichen. Der Hintergrund dieses Vergleiches ist die Tatsache, dass die *PCA* viel schneller als der *Autoencoder* arbeitet. Die Frage ist, ob sich der Mehraufwand des *Autoencoders* lohnt, jedoch wurde durch die Analyse festgestellt, das dies nicht der Fall ist.

In *Kapitel 2. Theoretischen Aspekte* werden die theoretischen Grundlagen der wichtigen Komponenten, die während des Praktikums benutzt wurden, erläutert. In *Kapitel 3. Praktische Umsetzung und Ergebnisse* wird die Umsetzung und die Durchführung diese Komponenten beschrieben und zudem die Ergebnisse analysiert. Im letzten Kapitel wird ein Fazit gezogen und einen Ausblick auf das Thema *"Image Retrieval"* mit dem *CIFAR-100* Datensatz gegeben.

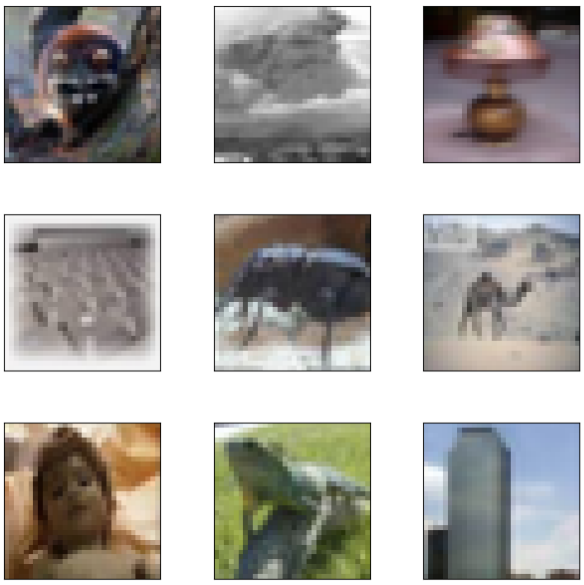
Der dem Bericht zugrunde liegende Code ist zu finden unter: `https://github.com/ahmetefe98/APAutoencoder`

2. Theoretische Aspekte

In diesem zweiten Kapitel werden die theoretischen Hintergründe dieses Projektes erläutert. Zuerst wird der bereits erwähnten *CIFAR-100* Datensatz im Detail besprochen. Der Großteil dieses Kapitels wird sich mit dem *Autoencoder* befassen, dabei werden auch die mathematischen Hintergründe kurz beschrieben. Weiterhin werden die Themen *PCA* und Clustern vorgestellt. Dieses Kapitel endet mit einer detaillierten Erläuterung zu der Suche nach den *Nächste-Nachbarn* abgeschlossen.

2.1 Eingabedaten[1]

Wie schon erwähnt, werden als Eingabedaten der *CIFAR-100* Datensatz benutzt, welches ein Teil des *"tiny images"*[2] Datensatzes ist, der aus 80 Millionen Bildern besteht. Der Datensatz kann entweder von der Homepage heruntergeladen werden[1] oder, falls *Tensorflow* benutzt wird, direkt importiert werden[3]. *CIFAR-100* beinhaltet 60.000 Bilder, 50.000 Trainings- und 10.000 Testbilder, die je eine Auflösung von 32x32 Pixel haben. Jedes einzelne der 1024 Pixel besteht aus drei Zahlen aus dem Intervall [0, 255] für die Rot-, Grün- und Blauwerte, dadurch lässt sich ein Bild mit 3072 Zahlen darstellen. Jedes Bild wird einer der 100 vorhandenen Klassen zugeordnet, sodass jede Klasse aus 600 Bildern, 500 Trainings- und 100 Testbilder, besteht. Die 100 Klassen können in 20 Oberklassen mit je 5 Klassen gruppiert werden, daraus folgt, dass jedes Bild zwei Bezeichnungen hat, den Namen der Klasse und den Namen der Oberklasse, zu der das Bild zuzuordnen ist. *(Siehe Tabelle 1, S. 3).*



Nr.	Klassename	Oberklassename
1	raccoon	medium mammals
2	cloud	large natural outdoor scenes
3	lamp	household electrical devices
4	keyboard	household electrical devices
5	beetle	insects
6	camel	large omnivores and herbivores
7	boy	people
8	lizard	reptiles
9	skyscraper	large man-made outdoor things

Abbildung 1. Beispielbilder aus *CIFAR-100* und die dazugehörigen Klassennamen und Oberklassennamen[4]

Durch die geringe Auflösung von 32x32 Pixel fällt es dem Menschen teilweise sehr schwer, das Bild einer Klasse zuzuordnen, jedoch bereiten hohe Auflösungen und damit eine hohe Anzahl an Dimensionen dem Model Schwierigkeiten und verursachen zudem auch einen hohen Ressourcenverbrauch und Zeitaufwand. In der *Abbildung 1* sind neun Beispielbilder aus dem Datensatz und die dazugehörigen Klassen- und Oberklassennamen zu sehen. Die Bilder sind nummeriert von links oben nach rechts unten. Einige dieser Bilder können ohne Probleme einer Klasse zugeordnet werden, bei anderen ist das nicht ganz deutlich. An dieser Stelle sollte erwähnt werden, dass diese Beispielbilder im Vergleich zu anderen Bildern des Datensatzes leicht zu erkennen und zu

[1] Hauptkomponentenanalyse
[2] https://groups.csail.mit.edu/vision/TinyImages/

Oberklassen	Klassen
aquatic mammals	beaver, dolphin, otter, seal, whale
fish	aquarium fish, flatfish, ray, shark, trout
flowers	orchids, poppies, roses, sunflowers, tulips
food containers	bottles, bowls, cans, cups, plates
fruit and vegetables	apples, mushrooms, oranges, pears, sweet peppers
household electrical devices	clock, computer keyboard, lamp, telephone, television
household furniture	bed, chair, couch, table, wardrobe
insects	bee, beetle, butterfly, caterpillar, cockroach
large carnivores	bear, leopard, lion, tiger, wolf
large man-made outdoor things	bridge, castle, house, road, skyscraper
large natural outdoor scenes	cloud, forest, mountain, plain, sea
large omnivores and herbivores	camel, cattle, chimpanzee, elephant, kangaroo
medium-sized mammals	fox, porcupine, possum, raccoon, skunk
non-insect invertebrates	crab, lobster, snail, spider, worm
people	baby, boy, girl, man, woman
reptiles	crocodile, dinosaur, lizard, snake, turtle
small mammals	hamster, mouse, rabbit, shrew, squirrel
trees	maple, oak, palm, pine, willow
vehicles 1	bicycle, bus, motorcycle, pickup truck, train
vehicles 2	lawn-mower, rocket, streetcar, tank, tractor

Tabelle 1. Klassennamen und Oberklassennamen

deuten sind. Bei vielen Bildern ist eine Zuordnung nicht möglich.

Wichtig für eine Clusterung ist einerseits eine hohe klasseninterne Ähnlichkeit, andererseits eine geringe Ähnlichkeit zwischen den Klassen. Ersteres gilt für den *CIFAR-100* Datensatz, Letzteres nicht. Wie schon erwähnt, fällt es dem Menschen bei vielen Bildern schwer, aufgrund der sehr geringen Auflösung von 32x32 Pixel, das Bild einer Klasse zuzuordnen, wie gut das der *Autoencoder* oder die *PCA* schaffen wird, wird im *Kapitel 3. Praktische Umsetzung und Ergebnisse* ausgewertet.

2.2 Autoencoder

Die Hauptthematik und Architektur dieses Fortgeschrittenpraktikums ist der *Autoencoder.* Ein *Autoencoder* ist ein neuronales Netzwerk, das aus zwei Teilen dem *Encoder* und *Decoder* besteht und dessen Aufgabe es ist, eine Eingabe wieder auszugeben, also die Identitätsfunktion zu approximieren, jedoch mit einer Einschränkung. Die mittlere Schicht des neuronalen Netzes soll eine geringere Dimension als die Eingabe- und Ausgabeschicht haben, deren Dimension von den Eingabedaten abhängt. In der Anwendung mit dem *CIFAR-100* Datensatz hat die Eingabe- und Ausgabeschicht 3072 Dimensionen.[3] Wenn die Anzahl der Dimensionen der mittleren Schicht nicht klein genug ist, kann dies dazu führen, dass der *Autoencoder* die Eingabe ohne Bearbeitung direkt an die Ausgabe weiterleitet oder die Eingaben auswendig lernt. Diese mittlere Schicht ist eine komprimierte Wissenspräsentation der Eingabedaten. Die Aufgabe dieser Komprimierung hat der *Encoder,* er soll aus den Daten lernen und herausfinden wie er am Besten die Daten komprimiert, sodass diese im Anschluss mit dem *Decoder* wieder entkomprimiert werden können und dabei der Fehler minimal bleibt. Der Wunsch ist das Erlernen von semantischen Merkmalen der Bilder, jedoch wird deutlich, das sich der *Autoencoder* auf die Rekonstruierung der Pixel konzentrieren wird und somit die semantischen Merkmale nicht lernen wird.

Der Unterschied zur *PCA,* welches noch im Detail erläutert wird, ist die Möglichkeit einer nicht-linearen Transformation der Eingabedaten. Auch zu sehen in der *Abbildung 2 (S. 4)*. Der zweite Teil der Architektur ist der *Decoder,* der mit dieser komprimierten Wissenspräsentation die Ursprungsdaten wiederherstellen und dabei so wenig wie möglich vom Original abweichen soll. Der

[3] 32*32Pixel*3RGB-Werte

Erfolg des *Decoders* hängt maßgeblich von der Qualität des *Encoders* ab, je besser der *Encoder,* desto besser der Decoder.

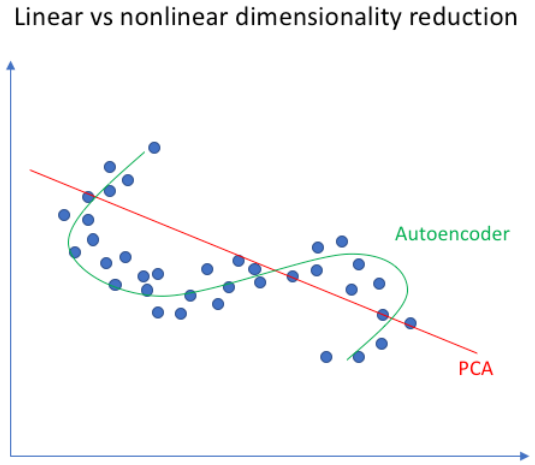


Abbildung 2. Vergleich *Autoencoder* und *PCA* [5]

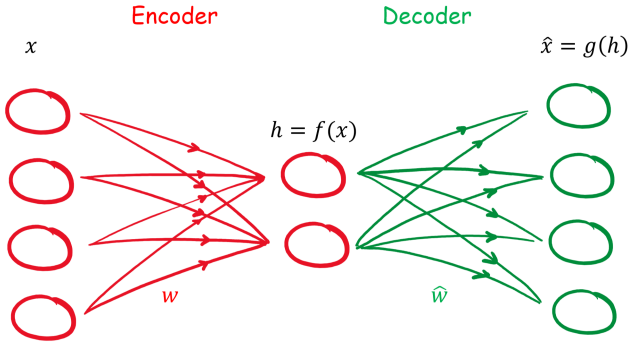


Abbildung 3. Aufbau eines *Autoencoders* [6]

Im Folgenden wird der mathematische Hintergrund dieses Prozesses kurz erläutert. Dazu wird als Grundlage die *Abbildung 3* benutzt. Die Eingabe wird als x, die Ausgabe als $\hat{x}$ und die komprimierte Wissenspräsentation der Eingabe als h bezeichnet. Zudem stehen W und $\hat{W}$ für die Gewichte, B und $\hat{B}$ für die Bias und σ und $\hat{\sigma}$ für die Aktivierungsfunktionen des *Encoders* und *Decoders.* Sei f die Funktion des *Encoders*

$$h = f(x) = \sigma(Wx + B) \tag{1}$$

und g die Funktion des *Decoders.*

$$\hat{x} = g(h) = \hat{\sigma}(\hat{W}h + \hat{B}) \tag{2}$$

Das Ziel ist die Minimierung des Rekonstruktionsfehlers L

$$L(x, \hat{x}) \tag{3}$$

bei Benutzung der mittleren quadratischen Abweichung (MSE) folgt für den Fehler L

$$L(x,\hat{x}) = MSE(x,\hat{x}) = \|x - \hat{x}\|^2 = \left\|x - \hat{\sigma}(\hat{W}(\sigma(Wx + B)) + \hat{B})\right\|^2 \tag{4}$$

Da x die Daten repräsentiert und die Aktivierungsfunktionen σ und $\hat{\sigma}$ im Vorfeld festlegt wird, müssen W, $\hat{W}$, B und $\hat{B}$ durch das Trainieren des *Autoencoders* geeignet gewählt werden, sodass der Rekonstruktionsfehler L minimal wird.

Im Folgenden werden einige Eigenschaften dieser Architektur im Detail erläutert. *Autoencoder* sind datenspezifisch, d.h. sie können nur mit den Daten arbeiten, mit denen sie trainiert wurden, da sie sich an die Daten anpassen und spezialisieren müssen. Dabei muss erwähnt werden, dass die Daten eine Klassenstruktur benötigen, bei zufällig gewählten und nicht miteinander zusammen passenden Daten ist der *Autoencoder* nutzlos. Nachdem ein Modell mit dem *CIFAR-100* Datensatz trainiert wurde, kann es im Anschluss nur mit diesen Daten arbeiten. Der Versuch, nach dem Training andere Daten, wie z.B. *MNIST* [4], auszuwerten, wird keine brauchbaren Ergebnisse liefern. Zudem bedeutet dies, das für jeden Datensatz das Model neu trainiert werden muss und dafür neue Trainingsdaten gebraucht werden. Darüber hinaus ist es verlustbehaftet, d.h. das Informationen verloren gehen und es so immer einen, wenn auch minimalen Unterschied zwischen dem rekonstruierten und dem originalen Bild geben wird. Durch die beiden erläuterten Eigenschaften, daten-spezifisch und verlustbehaftet, lässt sich schließen, das *Autoencoder* nicht zur Datenkompression geeignet sind. Der einzige Anwendungsfall zur Datenkompression wären sehr spezielle Daten, für die keine verlustfreie Möglichkeit zur Kompression existiert. Geeignet sind sie z.B. zum entrauschen von Daten, zur Dimensionsreduzierung der Repräsentation oder zum unüberwachten Lernen, bei dem keine Bezeichnungen nötig sind. Die letzten beiden Einsatzfelder werden für die Suche nach den *Nächste-Nachbarn* und die Dimensionsreduzierung für das Clustern benötigt [2, 7, 8, 9, 5, 10].

2.3 Hauptkomponentenanalyse (PCA)

Eine weitere wichtige Komponente ist die *PCA,* die durch lineare Dimensionsreduzierung (*Abbildung 2*) Informationen aus Daten mit einer hohen Anzahl an Dimensionen extrahiert und mit einer geringen Anzahl

[4] https://en.wikipedia.org/wiki/MNIST_database

an Dimensionen repräsentiert. Dabei wird versucht, die wichtigen Informationen herauszufinden, um diese zu speichern und die unwichtigen Informationen zu verwerfen. Auch wenn die Informationen, die gelöscht werden, unwichtig sind, geht Genauigkeit verloren. Dieser Verlust wird aber gerne toleriert, da dadurch Dimensionen gespart und somit die Daten vereinfacht werden können. Auch durch die Rekonstruktion können die verloren gegangenen Informationen nicht vollständig ausgeglichen werden.

Die einzelnen Bestandteile der reduzierten Repräsentation werden als *Hauptkomponenten* bezeichnet, die eine Richtung und sowie eine Größe haben. Die Richtung steht für die Hauptachse, auf der die Daten die meiste Varianz haben. Die Größe ist ein Betrag, der die Menge der Varianz der Daten angibt, die von den *Hauptkomponenten* auf dieser Achse erfasst werden können. Die erste *Hauptkomponente*, die genauso wie alle anderen eine gerade Linie ist, hat die höchste Varianz. Die darauf folgenden *Hauptkomponenten* liegen orthogonal auf ihrer vorherigen und haben eine geringere Varianz als die Vorherige.
Die einzelnen Schritte der *PCA*

1. Standardisierung
2. Kovarianzmatrix Berechnung
3. Berechnung der Eigenvektoren und Eigenwerten der Kovarianzmatrix zur Bestimmung der *Hauptkomponenten*
4. Eigenschaften Vektor
5. Darstellung der Daten entlang der Hauptkomponentenachsen

werden gut beschrieben in [11].

PCA sind u.a. hilfreich zur Visualisierung von Daten und zur Dimensionsreduzierung. Die Reduzierung der Dimensionen wird benutzt, um die *Nächste-Nachbarn* zu bestimmen, da dadurch wie beim *Autoencoder* eine geringe Anzahl an Dimensionen erreicht und so die Suche nach den *Nächste-Nachbarn* beschleunigt und verbessert werden kann. Inwieweit dies zum Erfolg führen kann wird im *Kapitel 3.7 Vergleich Nächste-Nachbarn* gezeigt [11, 12, 13].

2.4 Clustern mithilfe des Autoencoders

In der Einleitung wurde erwähnt, das *"k-means"* Algorithmen bei hohen Eingabedimensionen an ihre Grenzen geraten. An dieser Stelle sollen die Vorteile eines *Autoencoders* genutzt werden. Wie im Unterkapitel *Autoencoder* ausführlich erklärt, gleicht die mittlere Schicht einer komprimierten Wissenspräsentation der Eingabe mit einer bestimmten Anzahl an Dimensionen. Die Hoffnung ist, dass sich die komprimierten Wissenspräsentationen innerhalb einer Klasse ähneln und es keine Überschneidungen zu den anderen Klassen gibt. Dies wäre der Optimalfall für diese Anwendung. Um diese niedrigen Dimensionen fürs Clustern nutzen zu können, muss der *Autoencoder* im Vorfeld trainiert werden. Nachdem dies erfolgt, wird auf die Ausgabe des *Encoders,* also der mittleren Schicht, eine *"Clustering-Schicht"* gebaut, um die Eingaben mit dem "k-means"-Algorithmus zu clustern. Der *Decoder* ist hier beim Trainieren des *Autoencoders* und Minimieren des Rekonstruktionsfehlers von Bedeutung, auf das Clustern hat er keinen direkten Einfluss. Auch zu sehen in *Abbildung 4 S. 6*. Da der *Autoencoder* beim Trainieren sich auf die Minimierung des Rekonstruktionsfehlers konzentriert hat, muss das Modell erneut mit der *"Clustering-Schicht"* und dem *"k-means"* trainiert werden, sodass jetzt neben dem Rekonstruktionsfehler auch der *"Clusterfehler"* minimiert wird. Durch die niedrige Anzahl an Dimensionen fällt es *"k-means"* jetzt viel einfacher die Daten zu clustern [2, 10, 14].

2.5 Nächste-Nachbarn

Anhand eines gegebenen Bildes sollen seine *Nächste-Nachbarn* bestimmt werden, d.h. es soll eine bestimmte Anzahl k an Bildern ausgegeben werden, die dem Ausgangsbild am *"Nächsten"* sind. Am *"Nächsten"* wird definiert als den geringsten Abstand zu dem Ausgangsbild, wohingegen Abstand ein beliebiges metrisches Maß sein kann. Am häufigsten wird in diesem Fall der *euklidische Abstand*[5] benutzt.

Sei q das Ausgangsbild, s ein beliebiges aus der Menge M aller Bilder und n die Anzahl der Dimension. Es muss gelten, dass beide Bilder dieselbe Anzahl an Dimensionen haben. Dann kann der euklidische Abstand $d(q,s)$ mit folgender Formel berechnet werden.

$$d(q,s) = \sqrt{(q_1 - s_1)^2 + (q_2 - s_2)^2 + ... + (q_n - s_n)^2} \tag{5}$$

Der Algorithmus muss den euklidischen Abstand des Ausgangsbildes zu allen anderen Bildern ausrechnen und zwischenspeichern, das benötigt $|M| - 1$, in diesem Fall 59.999, Rechnungen. Die k Bilder mit dem geringsten Abstand werden ausgegeben, wobei k vorher festgelegt wird. Die Erwartung ist, dass die ausgegebenen

[5] https://en.wikipedia.org/wiki/Euclidean_distance

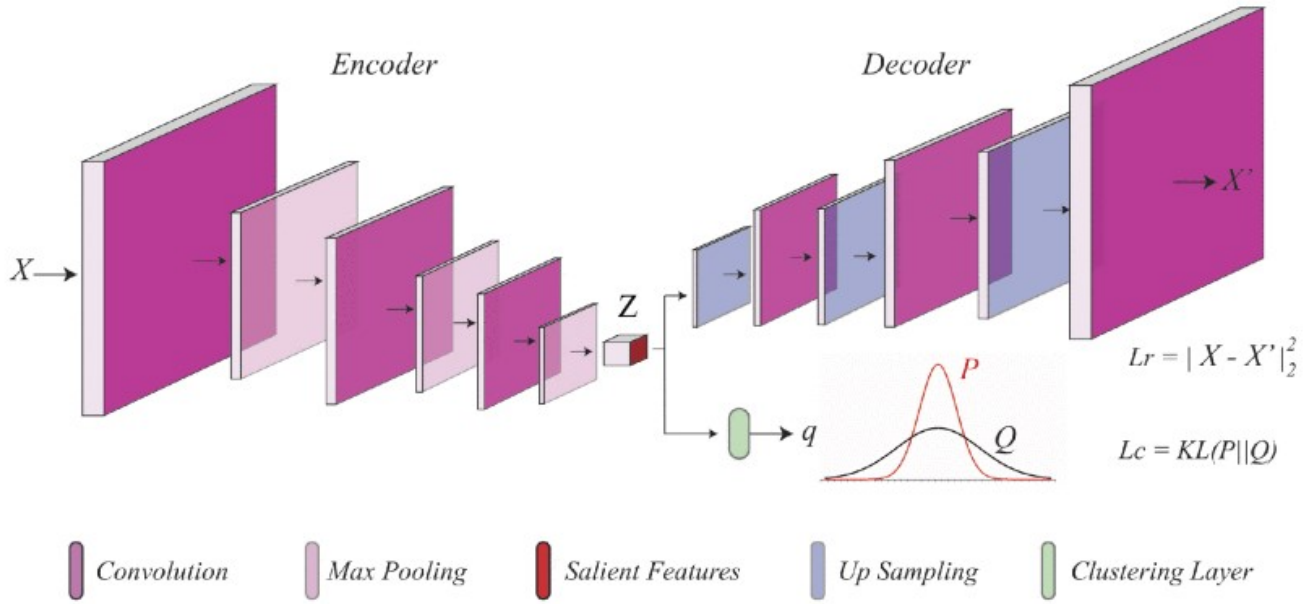


Abbildung 4. Aufbau eines *Autoencoder* mit Clustering Schicht [10]

Bilder aus derselben Klasse stammen wie das Ausgangsbild. Im *Kapitel 3.7 Vergleich Nächste-Nachbarn* wird deutlich, das es Klassen gibt, die diese Erwartung erfüllen, jedoch auch Klassen bei denen das nicht der Fall ist. Theoretisch könnten die *Nächste-Nachbarn* anhand der unverarbeiteten Bilder ermittelt werden. Jedoch würde der *Nächste-Nachbarn* -Algorithmus die einzelnen Pixel der Bilder vergleichen und die Bilder ausgeben, die in der Pixelebene am *"Nächsten"* sind. Dies ist jedoch nicht das Ziel und dauert zudem bei einer hohen Anzahl an Dimensionen sehr lange. Auch hier soll der *Autoencoder* genutzt werden und die Suchen nach den *Nächste-Nachbarn* anhand der komprimierten Wissenspräsentation der mittleren Schicht erfolgen. Dadurch haben die Daten wieder eine gering Dimension, sodass es dem Algorithmus einfacher fallen sollte. Der Wunsch hier ist, dass die komprimierte Wissenspräsentation des *Autoencoders* der Bilder einer Klasse ähnlich zueinander und verschieden zu den komprimierten Wissenspräsentationen der Bilder anderer Klassen sind. Wie erwähnt soll der *Autoencoder* mit der *PCA* vergleichen werden, dafür werden auch die *Nächste-Nachbarn* , nach anwenden der *PCA* auf die Daten, bestimmt und die Ergebnisse analysiert. In beiden Fällen ist der Wunsch, das die semantische Merkmale durch *Autoencoder* und *PCA* erkannt und gespeichert werden [15].

3. Praktische Umsetzung und Ergebnisse

In diesem Abschnitt werden die Komponenten, die im *Kapitel 2. Theoretische Aspekte* behandelt wurden, praktisch umgesetzt und die Ergebnisse analysiert. Die beiden Komponenten *Autoencoder* und *PCA* werden implementiert und durch den *Rekonstruktionsfehler* sowie der Suche nach den *Nächste-Nachbarn* vergleichen. Die Suche nach den *Nächste-Nachbarn* wird beim *Autoencoder* anhand der komprimierten Wissenspräsentation und bei der *PCA* nach Verringerung der Dimensionen durchgeführt. Wie bereits angekündigt, wird auch versucht, mithilfe der komprimierten Wissenspräsentation des *Autoencoders* die Bilder zu clustern.

3.1 Vorbereitung der Daten

Unabhängig von der Komponente, die genutzt wird, müssen die Daten im vorbereitet werden. Die Vorbereitung ist schnell abgearbeitet. Die Daten müssen zuerst geladen werden, das lässt sich am kompaktesten das genutzte Framework erledigen. Da *Tensorflow* benutzt wird, kann dies anhand [3] geschehen. Falls die Daten über die Homepage [1] heruntergeladen werden, ist dieser Prozess etwas umständlicher und erfordert die Funktion *unpickle(file)* die der Homepage zu finden ist. Nachdem die Daten geladen wurden, müssen sie in den Datentyp *"float32"* umgewandelt werden. Der letzte Schritt ist die Normalisierung, dies geschieht durch die Division mit

255, damit die Werte aus dem Intervall $[0,1]$ statt $[0,255]$ stammen.

3.2 Autoencoder

Wie bereits ausführlich erklärt, ist die Hauptarchitektur dieses Fortgeschrittenenpraktikums der *Autoencoder*. Als Basis wird der Code von Mahtab Noor Shaan [16] unter den Abschnitten *"Autoencoder Model"* und *"Implement Convolutional AE"* genutzt. Der Aufbau ist identisch zu dem in der *Abbildung 4 S. 6*. Ein Unterschied ist, dass das genutzte Modell nach der Input-Schicht immer zwei "Convolution"-Schichten je "Convolution"-Schicht der Abbildung besitzt. Zudem besteht der *Autoencoder* nur aus zwei statt drei *"Max Pooling"*- und *"Up Sampling"*-Schichten.
Im Detail:

$$\begin{aligned} &Input \rightarrow Convolution \rightarrow Convolution \\ &\rightarrow MaxPooling \rightarrow Convolution \rightarrow Convolution \\ &\rightarrow MaxPooling \rightarrow Convolution \rightarrow Convolution \\ &\rightarrow UpSampling \rightarrow Convolution \rightarrow Convolution \\ &\rightarrow UpSampling \rightarrow Convolution \rightarrow Convolution \\ &\rightarrow Convolution(Output) \end{aligned} \tag{6}$$

Als Aktivierungsfunktion der *"Convolution"*- Schichten wird "relu", als Optimierer "SGD" und als Fehlerfunktion "mean_squared_error" benutzt. Das Modell wird mit 100 Epochen trainiert, genauere Informationen können dem Code entnommen werden.

3.3 Clustern

Wie im *Kapitel 1. Einleitung* erwähnt, war das ursprüngliche Ziel dieses Fortgeschrittenenpraktikums die Bilder des Datensatzes zu clustern. Zur Umsetzung wurde als Grundlage Codeteile von Mahtab Noor Shaan [16] genommen und angepasst. In diesem GitHub-Repository wurden mehrere Möglichkeiten miteinander verglichen, übernommen wurde der Abschnitt "Convolutional AE with simple NN as classifier" und die dazu gehörigen Funktionen. Mahtab Noor Shaan hat mit dem *CIFAR-10* Datensatz[6] gearbeitet.

Um herauszufinden, wie gut das funktioniert, wurden die 10.000 Testbilder geclustert und eine Matrix ausgeben. Die Zeile zeigt den Klassennamen des Bildes und die Spalte den Klassennamen zudem das Bild geclustert wurde. Zu sehen in *Abbildung 5 S. 8*. Die ausgegeben Matrix hat $100x100$ Einträge, was zur Folge hat, dass die Werte der einzelnen Zellen nicht erkennbar sind[7]. Jedoch können die Resultate anhand der Farbskala abgeschätzt werden, im Optimalfall müsste eine helle Diagonale erkennbar sein, was leider nicht der Fall ist. An dieser Stelle lohnt sich ein Vergleich mit der Matrix des *CIFAR-10* Datensatzes, dessen einzelne Zellen besser erkennbar sind, da diese Matrix nur $10x10$ Einträge hat. Zu sehen in *Abbildung 6 S. 8*.

Trotz der Abweichungen wird bei diesem Datensatz eine klare helle Diagonale erkannt. Das lässt sich darauf zurück führen, dass der *CIFAR-10* Datensatz 10 Klassen je 5000 Trainingbilder und bei *CIFAR-100* Datensatz 100 Klassen je 500 Bilder hat. Der *Autoencoder* und die *Clustering-Schicht* haben mit dieser Kombination viele Klassen und wenig Bilder pro Klasse ihre Probleme. Bei Datensätzen mit vielen Bildern pro Klasse scheint die Kombination aus *Autoencoder* und *Clustering-Schicht* gute Arbeit zu leisten. Da aber das Ziel ist, mit dem *CIFAR-100* Datensatz zu arbeiten, wurden nach diesem Ergebnis beschlossen, die *Nächste-Nachbarn* zu bestimmen.

3.4 Hauptkomponentenanalyse (PCA)

Die Implementierung der *PCA* ist sehr kompakt und benötigt nur 3 Zeilen Code. Nachdem die Anzahl der Dimensionen, auf die reduziert werden soll, bestimmt wurde, muss die *PCA* noch trainiert werden. Nach dem Training können die Dimensionen der Eingabebilder reduziert werden.

3.5 Vergleich Rekonstruktionsfehler

An dieser Stelle wird der Rekonstruktionsfehler des *Autoencoders* und der *PCA* verglichen. Zum Berechnen des Fehlers wird die Summe der absoluten Differenzen zwischen dem originalen Bild und dem wieder rekonstruierten Bild bestimmt. Im Fall des *Autoencoders* wird die Ausgabe des *Decoders* benutzt und bei der *PCA* werden aus den komprimierten Informationen die Bilder wieder rekonstruiert. Für jedes einzelne Pixel wird die Differenz zwischen dem Wert des originalen Bildes mit dem Wert des wieder entkomprimierten Bildes gebildet. Der Betrag dieser Differenz wird aufsummiert. Da dieser Wert von der Anzahl der Pixel abhängt und dadurch nicht aussagekräftig ist, wird er noch durch 1024 geteilt. Das Ergebnis ist die Summe der absoluten Differenzen

[6] Der Unterschied zum *CIFAR-100* Datensatzes ist, das der *CIFAR-10* Datensatz nur 10 Klassen hat, da die Gesamtanzahl der Bilder bei 60.000 bleibt besteht hier jede Klasse aus 6000 statt 600 Bildern [1].

[7] Das Bild ist mit einer besseren Auflösung auf GitHub zu finden

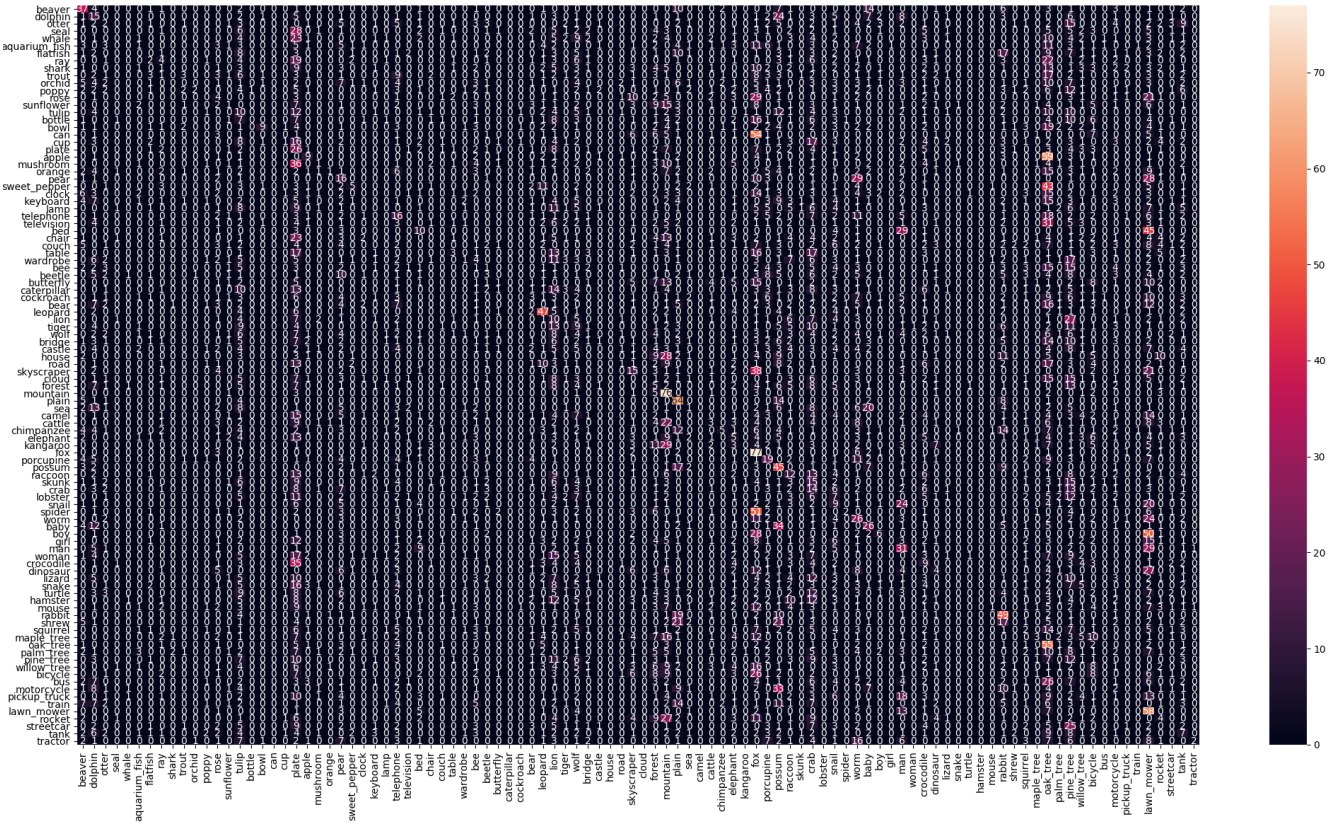


Abbildung 5. Vergleich echte Klasse und vermutetet Klasse *CIFAR-100*

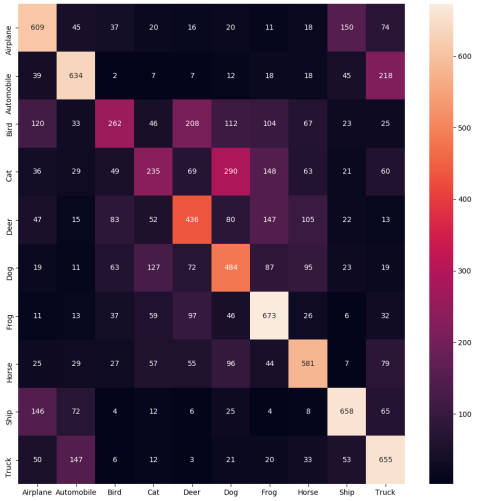


Abbildung 6. Vergleich echte Klasse und vermutetet Klasse *CIFAR-100*

für ein Pixel. Um die Differenz eines RGB-Wertes zu erhalten, muss das Ergebnis noch durch 3 geteilt werden.

Der Vergleich wurde mit allen 60 000 Bildern gemacht. Alle 60 000 Bilder wurden beim *Autoencoder* mit dem *Encoder* auf 128 Dimensionen komprimiert und mit dem *Decoder* wieder die ursprüngliche Größe entkomprimiert, bei der *PCA* wurden auch alle Bilder auf 128 Dimensionen komprimiert und im Anschluss wieder auf die ursprüngliche Anzahl entkomprimiert. Um zu sehen, welche Methode besser die Bilder rekonstruieren kann, wurde das Minimum, das Maximum, das arithmetischer Mittel und der Median bestimmt. Beim Minimum bzw. Maximum ist das Bild gemeint, das mit dem geringsten bzw. größten Unterschied auf Pixelebene rekonstruiert wurde. Zudem werden die 5 Bilder, die am besten und die 5 Bilder, die am schlechtesten rekonstruiert worden sind, ermittelt. Außerdem wurden auch die 5 besten und 5 schlechtesten Klassen ermittelt.

Die Tabelle *"Comparison reconstruction loss"* ist zu finden im Anhang und beinhaltet eine Spalte für den *Autoencoder* und eine für die *PCA*. Im ersten Block sind die besten Werte der Zeilen grün markiert. Beim Minimum ist die *PCA* besser, der *Autoencoder* hat hier einen viermal so hohen Fehler. Beim arithmetischen Mittel, Median und Maximum ist *Autoencoder* besser. Bei den ersten beiden ist die Abweichung nur gering, wohingegen es beim Maximum deutlich ist. Da diese vier Werte und die Unterschiede nicht aussagekräftig genug sind, ist in *Abbildung 7 S. 9* ein Histogramm mit der

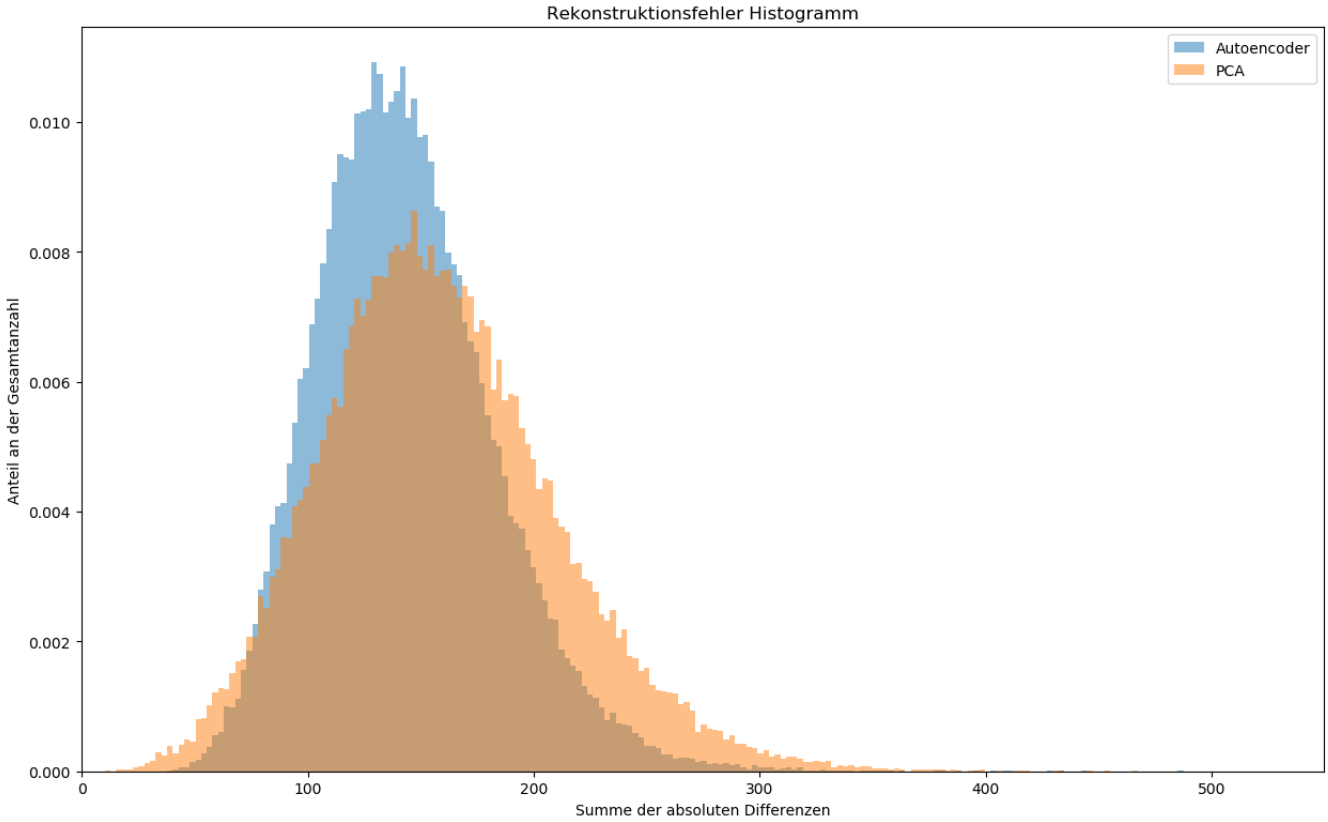


Abbildung 7. Vergleich Verteilung Autoencoder und PCA

Häufigkeitsverteilung beider Methoden zu sehen. Das Diagramm lässt sich durch die beiden Schnittpunkte, in 3 Abschnitte unterteilen. Im ersten und im letzten Bereich [0, 75] und [170, 500] sind fast doppelt so viele Bilder von der *PCA* wie vom *Autoencoder,* 2 400 zu 1 300 und 20 700 zu 11 600. Im mittleren Abschnitt [75, 170] sind viel mehr Bilder vom *Autoencoder,* 47 100 zu 36 800. Nach Auswerten des Minimums, Maximums, Medians, arithmetischen Mittels und des Histogramms ist der *Autoencoder* in diesem Vergleich des Rekonstruktionsfehlers besser als die *PCA,* jedoch ist die Frage, ob sich der Mehraufwand lohnt.

Im zweiten Block stehen die fünf am besten und am schlechtesten rekonstruierten Bilder vom *Autoencoder* und von der *PCA*. Links die Nummer des Bildes und rechts der Name der Unterklasse, zudem das Bild gehört. In diesen Abschnitt sind Klassen, die sowohl beim *Autoencoder* als auch bei der *PCA* auftauchen, gelb markiert. Der *Autoencoder* konnte am besten ein Bild aus der Klasse *"otter"* und die *PCA* ein Bild aus der Klasse *"sea"* rekonstruieren. Die dazugehörigen Werte stehen im ersten Block unter Minimum. Das Bild vom *"sea"* hat pro RGB-Wert einen Fehler von 0,0011, multipliziert mit 255, da die Daten bei der Normalisierung mit 255 dividiert wurden, ergibt das eine durchschnittliche Abweichung von 0,2805 je RGB-Wert. Das Bild vom *"otter"* beim *Autoencoder* hat mit 1,122 eine vier mal so hohe Abweichung. Die Werte sind nicht aussagekräftig, daher sind in den *Abbildungen 8 und 9 (S. 10)* die original und die rekonstruierten Bilder zu sehen. In *Abbildung 9* ist kaum ein Unterschied zwischen beiden Bildern zu sehen, jedoch ist das nicht verwunderlich, da fast alle Pixel dieselbe Farbe haben. Das gleiche gilt auch für den *"otter"* in *Abbildung 8*, jedoch verliert der *"otter"* im rekonstruierten Bild an Details. Keins der vier Bilder ist bei einer Annotation einer Klasse klar zuordenbar.

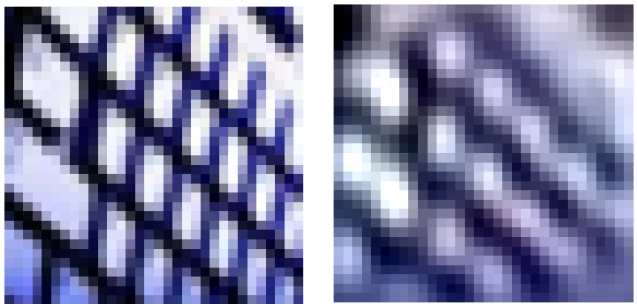
Der *Autoencoder* konnte am schlechtesten ein Bild aus der Klasse *"cattle"* und die *PCA* ein Bild aus der Klasse *"keyboard"* rekonstruieren. Die dazugehörigen Werte stehen im ersten Block unter Maximum. Das Bild vom *"keyboard"* hat pro RGB-Wert einen Fehler von 0,06, multipliziert mit 255, da die Daten bei der Normalisierung mit 255 dividiert wurden, ergibt das eine durchschnittliche Abweichung von 15,3 je RGB-Wert. Das Bild vom *"cattle"* beim *Autoencoder* hat mit 13,515, eine etwas niedrigere Abweichung. Die Werte sind nicht

(a) original **(b)** rekonstruiert

Abbildung 8. Das am besten rekonstruierte Bild von der *PCA* "sea"

(a) original **(b)** rekonstruiert

Abbildung 9. Das am besten rekonstruierte Bild vom *Autoencoder* "otter"



(a) original **(b)** rekonstruiert

Abbildung 10. Das am schlechtesten rekonstruierte Bild von der *PCA* "keyboard"

(a) original **(b)** rekonstruiert

Abbildung 11. Das am schlechtesten rekonstruierte Bild vom *Autoencoder* "cattle"

aussagekräftig, daher sind in den *Abbildungen 10 und 11 (S. 10)* die originalen und die rekonstruierten Bilder zu sehen. In beiden Abbildungen sind große Abweichungen zwischen den Bildern zu erkennen.

Im letzten Block der Tabelle *"Comparison reconstruction loss"* stehen die fünf besten und die fünf schlechtesten rekonstruierten Klassen, dafür wurde der Durchschnitt über allen Bildern einer Klasse ermittelt. Hierdurch wird klar, mit welcher Art von Bildern der *Autoencoder* und die *PCA* gut umgehen können. Bei beiden Methoden stimmen die fünf am besten rekonstruierten Klassen überein. Diese Klassen beinhalten Bilder, die kaum Details und viele gleichfarbige Pixel haben. Wohingegen die Bilder bei den fünf am schlechtesten rekonstruierten Klassen das Gegenteil der Fall ist. Generell lässt sich sagen, dass sowohl der *Autoencoder* als auch die *PCA* gut mit Bildern umgehen können, die große gleichfarbige Fläche haben. Durch die Beispielbilder in *Abbildung 9 und 11* wird deutlich, dass der *Autoencoder* aus vier Pixeln eins macht, indem er den Mittelwert ermittelt, beim Rekonstruieren wird aus dem einen Pixel wieder vier, jedoch bekommen diese dieselbe Farbe und sehen dann im rekonstruierten Bild aus wie ein Pixel. Dadurch ist das nicht verwunderlich, das Bilder mit wenig Details und vielen gleichfarbigen Nachbarpixeln gut rekonstruiert werden. Sollten alle vier Pixel unterschiedliche Farben haben, bekommen sie nach dem rekonstruieren eine Farbe, die durch diese vier Farben gemischt wurden und haben keinen Zusammenhang mehr zum originalen Bild.

3.6 Nächste-Nachbarn

Die Implementierung der Suche nach den *Nächste-Nachbarn* ist durch Klassen und Funktionen aus der Python Bibliothek *sklearn* sehr kompakt. Nachdem der *Autoencoder* trainiert und die Daten mithilfe des *Encoders* komprimiert wurden, kann die Klasse *NearestNeigbors*[8] instanziiert werden. Durch die Methode *kneighbors* werden dann die *k Nächste-Nachbarn* ausgegeben.

[8] `https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.NearestNeighbors.html`

3.7 Vergleich Nächste-Nachbarn

An dieser Stelle wird die Qualität der Suche nach den *Nächste-Nachbarn* mithilfe des *Autoencoders* und der *PCA* verglichen. Dafür werden für ein Bild die fünf *Nächste-Nachbarn* bestimmt und die Klassen der *Nächste-Nachbarn* verglichen mit der Klassen dieses Bildes. Bei einer 100 % Genauigkeit stimmen die Klassen aller fünf ausgegebenen Bilder mit der Klasse des gegebenen Bildes überein, bei einer 20 % Genauigkeit nur die Klasse eines Bildes. Da die Auswertung der 60 000 Bilder zu lange dauert, wurde dieser Vergleich mit nur 1 000 Bilder gemacht, dabei stammen je 10 Bilder aus allen 100 Klassen, um keine Verzerrung durch eine Ungleichverteilung der Klassen zu erhalten. Durch einen *sample size calculater*[9] werden Konvergenzintervalle ermittelt. Es wurde neben dem arithmetischen Mittel und dem Median auch die fünf Klassen, deren Suche am Erfolgreichsten bzw. am Erfolglosesten waren, bestimmt.

Die Tabelle *"Comparison nearest neighbors"* ist zu finden im Anhang und beinhaltet die Ergebnisse dieses Vergleichs. Im Gegensatz zum Vergleich des Rekonstruktionsfehlers wurden hier die Bilder mithilfe des *Autoencoders* und der *PCA,* nicht nur auf 128 Dimensionen reduziert, sondern auch auf 32 und 64 Dimensionen. Es werden also sechs Modelle miteinander verglichen. *Autoencoder* mit 32, 64 und 128 Dimensionen sowie *PCA* mit 32, 64 und 128 Dimensionen. Wie auch beim Vergleich des Rekonstruktionsfehlers ist in der Tabelle die besten Werte grün markiert und die gleichen Klassen gelb markiert.

Im ersten Block stehen die Werte für das arithmetische Mittel und den Median, zudem wurde für den arithmetischen Mittel das Konfidenzintervall für ein 95 % und 99 % Konfidenzniveau ermittelt. Beim arithmetischen Mittel ist beim *Autoencoder* das Modell mit 32 Dimensionen am Besten, jedoch ist der Unterschied zu den anderen beiden Modellen nur minimal. Beim Median sind alle drei gleich. Bei einem Konfidenzniveau von 99 % beträgt das Konfidenzintervall beim Modell mit 128 Dimensionen $[9,1;14,3]$. Die *PCA* ist nur minimal besser, beim Modell mit 128 Dimensionen und einem Konfidenzniveau von 99 % beträgt das Konfidenzintervall $[9,84;15,2]$. Den besten Wert bei der *PCA* hat das Modell mit 64 Dimensionen. Wie beim *Autoencoder* sind auch bei der *PCA* der Median bei allen drei Modellen gleich. Auch wenn die Werte der *PCA* minimal höher sind, lässt sich hierdurch nicht ermitteln ob *Autoencoder* oder *PCA* besser ist, generell sind die Werte beider sehr schlecht. Nur 11 bzw. 12 % der *Nächste-Nachbarn* stammen aus der Klasse des Ausgangsbildes, all diese Modelle sind in der Anwendung unbrauchbar.

Im zweiten Block wurden jeweils die fünf Klassen mit der durchschnittlich höchsten Genauigkeit bestimmt. Eine Genauigkeit von z.B. 60 % bedeutet das im Durchschnitt 60 % der *Nächste-Nachbarn* dieselbe Klasse wie das gegeben Bild haben. Auch bei diesem Vergleich tauchen einige Klassen bei allen sechs Modellen auf, diese wären {*sea, plain, cockroach*}. Dies sind alles Klassen, deren Bilder kaum Details und viel Fläche mit derselben Farbe haben. Zudem haben die Bilder dieser Klassen immer dieselbe Farbe, z.B. ein *sea* ist immer blau, daher fällt es den Modellen leicht, nach anderen Bildern mit viel blauen Flächen zu suchen. Bei diesen besten Klassen stimmen im Durchschnitt die Klassen von zwei bis drei der fünf *Nächste-Nachbarn* mit der Klasse des Ausgangsbildes überein. Dies wäre ein guter Wert, wenn es durchschnittlich für alle Klassen der Fall wäre, dafür, dass so eine hohe Genauigkeit nur bei den "einfachen" Klassen erreicht werden, ist dies ein schlechter Wert.

Im letzten Block wurden jeweils die Klassen mit der durchschnittlich schlechtesten Genauigkeit bestimmt. Leider haben einige Klassen eine Genauigkeit von 0 %, sodass bei den 10 Bilder, die von jeder Klasse getestet wurden, nie eine Klasse der *Nächste-Nachbarn* mit der Klasse des Bildes übereingestimmte. Beim *Autoencoder* sind das 11, 12 und 19 Klassen, bei der *PCA* 6, 8 und 8 Klassen. In dieser Hinsicht ist die *PCA* minimal besser. Auch hier tauchen wieder Klassen wie {*flatfish, fox, girl*} in fast allen der sechs Modelle auf. Dies sind alles Klassen deren Bilder viele Details und nur wenig Fläche mit der selben Farbe haben, zudem sind diese Bilder schwer zu unterscheiden mit Bildern anderer Klassen. Ein *girl* ist sehr ähnlich zu einem {*boy, man, women*}, dasselbe gilt auch für einen *flatfish* oder für ein *fox* die eine hohe Ähnlichkeit zu anderen Klassen haben.

3.8 Zusammenhang zwischen Rekonstruktionsfehler und der Suche nach Nächste-Nachbarn

Nach den beiden Vergleichen stellt sich die Frage, ob es einen Zusammenhang zwischen der Qualität der Rekonstruktion und der Suche nach den *Nächste-Nachbarn* gibt. Dafür sollten die Spalten für die besten und schlechtesten Klassen beider Tabellen miteinander verglichen werden. Die Klassen {*plain, sea*} gehören sowohl zu den Klassen mit dem geringsten Rekonstruktionsfehler als auch zu den Klassen, die bei der Suche nach den

[9] https://www.surveysystem.com/sscalc.htm

Nächste-Nachbarn die meisten Übereinstimmungen hatten. Das sind beides Klassen, die, wie schon erwähnt, wenig Details und große Flächen mit derselben Farben haben. Durch diese großen Flächen mit derselben Farbe können die Bilder gut rekonstruiert werden, da die vier Pixel, die durch den *Autoencoder* zu einem Pixel werden, in den meisten Fällen bereits dieselbe Farbe haben. Daher ist in diesen Bereichen der Rekonstruktionsfehler sehr gering. Da außerdem deutlich wurde, das der *Autoencoder* und die *PCA* nur die visuellen Merkmale bei der Komprimierung lernen, ist es auch nicht verwunderlich, das die Klassen, deren Bilder dieselben Bereiche mit denselben Farben haben, hohe Werte bei der Suche nach den *Nächste-Nachbarn* erzielen. Der euklidische Abstand, der bei der Suche verwendet wird, kann nur den Abstand der Daten, die ihm zur Verfügung stehen, berechnen. Erfolgreicher wäre die Suche nach den *Nächste-Nachbarn,* wenn der Algorithmus eine semantische Repräsentation der Daten erhalten würde, dies hat jedoch weder der *Autoencoder* noch die *PCA* geschafft. In beiden Vergleichen haben die am schlechtesten abgeschnittenen Klassen viele Details und viele verschiedene Farben. Außerdem haben die Klassen keine hohe klasseninterne Ähnlichkeit, jedoch eine hohe Ähnlichkeit zu den anderen Klassen. Allerdings ist das Gegenteil für eine gute Clusterung nötig.

4. Fazit und Ausblick

In diesem Fortgeschrittenpraktikum wurden zwei Methoden, *Autoencoder* und *PCA,* zur Dimensionsreduzierung miteinander verglichen. Dieser Vergleich wurde anhand des Rekonstruktionsfehlers und der Suche nach den *Nächste-Nachbarn* vollzogen. Zudem wurde versucht, mit dem *Autoencoder* die Bilder zu clustern. Beim Vergleich waren beide Ansätze sehr ähnlich und haben schlecht abgeschnitten. Weder das Clustern noch die Suche nach den *Nächste-Nachbarn* waren gut genug, um dies in der Praxis umsetzen zu können. Diese beiden Anwendungsfälle erfordern das Lernen semantischer Merkmale, jedoch können der *Autoencoder* und die *PCA* nur visuelle Merkmale lernen. Der *CIFAR-100* Datensatz ist für diese Aufgabenstellung nicht optimal. Beide Ansätze konnten bei anderen Datensätzen gute Ergebnisse erzielen. Allerdings soll ein Ansatz gefunden werden, der auch bei einem ”komplizierteren” Datensatz gute Ergebnisse liefert.

Am Ende des Fortgeschrittenenpraktikums wurde noch ein Artikel [17] gefunden, der mit einem anderen Ansatz sehr gute Ergebnisse mit dem *CIFAR-100* Datensatz erzielte. Im Artikel *”Hierarchy-based Image Embeddings for Semantic Image Retrieval”* aus dem Jahr 2019 erläutern *”Barz & Denzler”* einen Ansatz zur semantischen Bildsuche. Sie beschreiben das Problem, dass neuronale Netze sich auf Bilderrepräsentation konzentrieren und dadurch die visuellen, aber leider keine semantischen Ähnlichkeiten lernen. Ihr Vorschlag ist die Abbildung der Bilder auf Klasseneinbettungen. Durch diesen Ansatz werden sehr gute Ergebnisse bei der Suche nach semantisch ähnlichen Bildern gemacht. Eine weitere Problematik, die bei traditionellen Ansätzen auftaucht, ist, dass teilweise Bilder ausgegeben werden, deren Klassen keinen semantischen Zusammenhang zu der gesuchten Klasse haben. Zum Beispiel kann es dazu kommen, das bei der Suche nach einem Bild einer Katze ein Bild eines Waldes ausgegeben wird, jedoch wäre das Bild eines Hundes semantisch näher zu dem Bild einer Katze. Diese Problematik taucht bei diesem neuen Ansatz nicht mehr auf. Laut dem Artikel wird je nach Architektur eine Genauigkeit bei der Bildersuche von 82-86 % erreicht und ist besser als jeder traditionelle Ansatz. Auf dem GitHup Repository [18] der Arbeitsgruppe *Computer Vision der Universität Jena,* zu der die beiden Autoren gehören, ist der Code zum Artikel zu finden und beinhaltet zu mehreren Datensätzen Anleitungen zum Trainieren und Auswerten des Ansatzes. Für den *CIFAR-100* Datensatz stehen für drei Architekturen je ein bereits trainierten Model zum Herunterladen zur Verfügung. In der *Abbildung 12 (S. 13)* ist die Anwendung dieses Ansatzes an drei Beispielbildern zusehen. Dabei wurden aber nicht die ersten zehn Bilder der Suche ausgegeben, sondern das $\{1., 21., 41., 61., 81., 101., 121., 141., 161., 181.\}$ Bild der Suche. Bei jeden Bild ist in der ersten Zeile die Ergebnisse bei der Anwendung eines traditionellen Ansatzes und in der zweiten Zeile die Ergebnisse bei der Anwendung des neuen Ansatzes. Je grüner der Rand des Bildes, desto semantisch näher zum Ausgangsbild und je roter der Rand des Bildes, desto semantisch weiter entfernt.

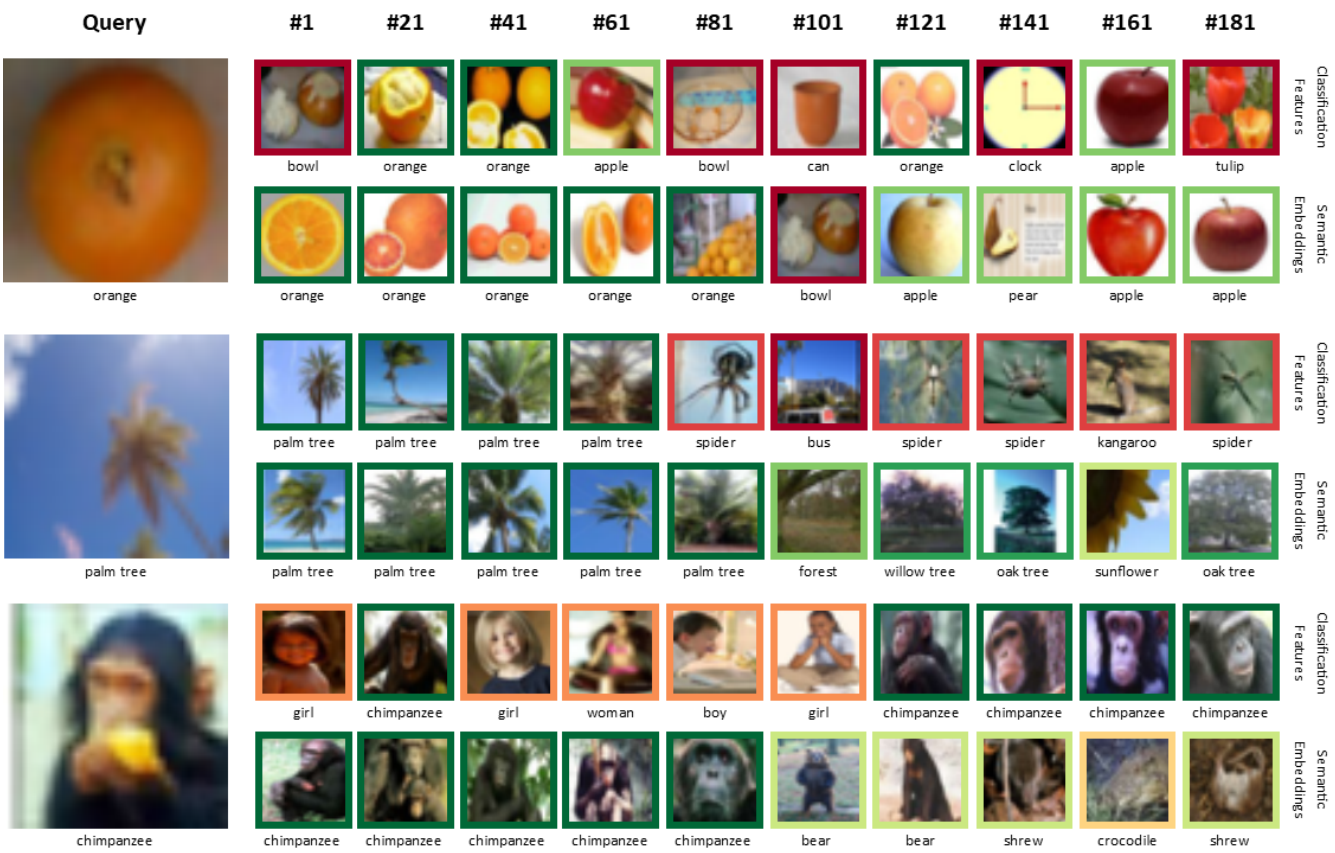


Abbildung 12. Vergleich aus GitHub [18]

Literatur

[1] Alex Krizhevsk. The cifar-100 dataset. besucht am 15.09.2020. URL: https://www.cs.toronto.edu/~kriz/cifar.html.

[2] Chengwei Zhang. How to do unsupervised clustering with keras. URL: https://www.dlology.com/blog/how-to-do-unsupervised-clustering-with-keras/.

[3] TensorFlow. tf.keras.datasets.cifar100.load_data. besucht am 04.01.2021. URL: https://www.tensorflow.org/api_docs/python/tf/keras/datasets/cifar100/load_data.

[4] Tensorflow. cifar100. besucht am 04.01.2021. URL: https://www.tensorflow.org/datasets/catalog/cifar100?hl=en.

[5] Jeremy Jordan. Introduction to autoencoders. besucht am 06.01.2021. URL: https://www.jeremyjordan.me/autoencoders/.

[6] Hoang Tu Nguyen. Einführung in die welt der autoencoder. besucht am 04.01.2021. URL: https://data-science-blog.com/blog/2020/04/01/einfuhrung-in-die-welt-der-autoencoder/.

[7] Francois Chollet. Building autoencoders in keras. besucht am 21.09.2020. URL: https://blog.keras.io/building-autoencoders-in-keras.html.

[8] UFLDL Stanford. Autoencoders. besucht am 06.01.2021. URL: http://ufldl.stanford.edu/tutorial/unsupervised/Autoencoders/.

[9] Aditya Sharma. Implementing autoencoders in keras: Tutorial. besucht am 29.09.2020. URL: https://www.datacamp.com/community/tutorials/autoencoder-keras-tutorial.

[10] W. Ellsworth S. M. Mousavi, W. Zhu and G. Beroza. Unsupervised clustering of seismic signals using deep convolutional autoencoders. In *IEEE Geoscience and Remote Sensing Letters*, volume 16, pages 1693–1697. IEEE, 2019. doi:10.1109/LGRS.2019.2909218.

[11] Zakaria Jaadi. A step-by-step explanation of principal component analysis. besucht am 15.01.2021. URL: https://builtin.com/data-science/step-step-explanation-principal-component-analysis.

[12] Aditya Sharma. Principal component analysis (pca) in python. besucht am 30.10.2020. URL: https://www.datacamp.com/community/tutorials/principal-component-analysis-in-python.

[13] Michael Galarnyk. Pca using python (scikit-learn). besucht am 15.01.2021. URL: https://towardsdatascience.com/pca-using-python-scikit-learn-e653f8989e60.

[14] Seungjae Patrick Lee. Autoencoder and k-means — clustering epl players by their career statistics. besucht am 10.01.2021. URL: https://medium.com/@iampatricolee18/autoencoder-k-means-clustering-epl-players-by-their-career-statistics-f38e2ea6e375.

[15] Nathan Hubens. Build a simple image retrieval system with an autoencoder. besucht am 10.01.2021. URL: https://towardsdatascience.com/build-a-simple-image-retrieval-system-with-an-autoencoder-673a262b7921.

[16] Mahtab Noor Shaan. autoencoder-as-feature-extractor-cifar-10. besucht am 26.09.2020. URL: https://github.com/MahtabShaan/autoencoder-as-feature-extractor-CIFAR-10/blob/master/autoencoder-as-feature-extractor-cifar10.ipynb.

[17] Bjorn Barz and Joachim Denzler. Hierarchy-based image embeddings for semantic image retrieval. In *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 638–647. IEEE, 2019. doi:10.1109/WACV.2019.00073.

[18] Computer Vision Group jena. semantic-embeddings. besucht am 26.11.2020. URL: https://github.com/cvjena/semantic-embeddings.

Comparison reconstruction loss

sum of absolute differences
60.000 samples

		Autoencoder 128	PCA 128 dimensions
minimum	total	39.3397	10.4469
	per pixel	0.0128	0.0034
	per RGB	0.0043	0.0011
arithmetic mean	total	143.118	158.241
	per pixel	0.047	0.052
	per RGB	0.016	0.017
median	total	139.923	154.312
	per pixel	0.046	0.050
	per RGB	0.015	0.017
maximum	total	487.392	549.457
	per pixel	0.159	0.179
	per RGB	0.053	0.060

top 5 picture	36 612		otter	48 628		sea
	49 112		worm	59 160		plain
	57 246		rabbit	24 642		cloud
	7 768		ray	32 559		sea
	26 736		sea	52 247		sea
worst 5 picture	15 804		clock	33 694		keyboard
	18 206		keyboard	50 621		keyboard
	4 155		keyboard	7 300		tiger
	32 758		keyboard	4 155		keyboard
	28 927		cattle	18 206		keyboard

top 5 classes	plain	sea
	sea	plain
	cloud	cloud
	road	ray
	ray	road
worst 5 classes	pickup	sweet_pepper
	keyboard	sunflower
	streetcar	poppy
	bus	tulip
	motorcyle	motorcycle

best value same classes

Comparison nearest neighbors

sample size: 1000 (= 100 classes x 10 images)

	Autoencoder						PCA					
dimensions	32		64		128		32		64		128	
average accurarcy	11.9%		11.3%		11.7%		12.52%		12.56%		12.52%	
median	8%		8%		8%		10%		10%		10%	
confidence interval by confidence level 95 % for average	[10.01, 13.89]		[9.35, 13.25]		[9.72, 13.68]		[10.49, 14,55]		[10.52, 14.6]		[10.49, 14.55]	
confidence interval by confidence level 99 % for average	[9.28, 14.52]		[8.72, 13.86]		[9.1, 14.3]		[9.84, 15.2]		[9.88, 15.24]		[9.84, 15.2]	
the top 5 classes	sea	56%	sea	56%	sea	56%	plain	52%	plain	56%	plain	60%
	plain	50%	plain	54%	cockroach	50%	sea	42%	streetcar	44%	sea	50%
	cockroach	48%	cockroach	46%	cloud	48%	cockroach	40%	cockroach	42%	cockroach	42%
	chair	40%	plate	36%	plain	44%	chair	38%	chair	38%	plate	38%
	wardrobe	36%	cloud	36%	wardrobe	42%	pear	34%	plate	36%	cloud	38%
the worst classes	12 classes with 0%		19 classes with 0 %		11 classes with 0%		6 classes with 0%		8 classes with 0%		8 classes with 0 %	
	bus	0%	bed	0%	elephant	0%	bottle	0%	bottle	0%	butterfly	0%
	butterfly	0%	bus	0%	flatfish	0%	flatfish	0%	butterfly	0%	flatfish	0%
	flatfish	0%	butterfly	0%	girl	0%	fox	0%	flatfish	0%	fox	0%
	fox	0%	cattle	0%	house	0%	otter	0%	fox	0%	girl	0%
	girl	0%	elephant	0%	palm_tree	0%	raccoon	0%	girl	0%	house	0%
	house	0%	flatfish	0%	possum	0%	table	0%	raccoon	0%	palm_tree	0%
	mouse	0%	fox	0%	raccoon	0%			table	0%	raccoon	0%
	palm_tree	0%	girl	0%	table	0%			train	0%	train	0%
	table	0%	house	0%	tractor	0%						

best value same classes