Hochschule Düsseldorf
University of Applied Sciences
HSD

Fachbereich Elektro- und Informationstechnik
Faculty of Electrical Engineering and Information Technology

**Projektarbeit:**

# „Vergleich von datengesteuerten Klassifizierungsmethoden im CIFAR10- und MNIST-Datensatz“

Von: Roman Juhls

Matrikelnummer: 804729

Projektarbeit für das Modul „Künstliche Intelligenz“ des Studiengangs: „Elektro- und Informationstechnik“

bei: Prof. Dr. -Ing. Dorothea Schwung

## Inhaltsverzeichnis

# 1. Einleitung

Im Rahmen des Moduls „Künstliche Intelligenz“ wurde das vorliegende Projekt bearbeitet, um die Grundlagen und Anwendungen verschiedener KI-Techniken zu erkunden. Insbesondere stehen die Themen Klassifizierung, Multilayer Perceptron (MLP), Auto-Encoder, Generative Adversarial Network (GAN), Merkmalsextraktion sowie vollständig verbundene und Faltungsschichten im Fokus.

## 1.1 Hintergrund und Motivation

Die Klassifizierung von Daten spielt eine zentrale Rolle in vielen Anwendungen der KI, von Bilderkennung bis hin zu Sprachverarbeitung. Durch den Einsatz von neuronalen Netzwerken, wie dem MLP, Auto-Encoder und GAN, werden komplexe Muster erkannt und abstrakte Repräsentationen gelernt. Dieses Projekt hat das Ziel, diese Algorithmen auf den CIFAR-10- und MNIST-Benchmark-Datensätzen anzuwenden, um ihre Leistung in der Klassifizierung zu evaluieren und zu vergleichen.

## 1.2 Zielsetzung

Das übergeordnete Ziel dieses Projekts ist es, einen umfassenden Einblick in verschiedene KI-Techniken zu gewinnen und ihre Anwendbarkeit auf realen Datensätzen zu evaluieren. Durch die Entwicklung eines maschinellen Lernrahmens für die Klassifizierung und die Implementierung der ausgewählten Algorithmen in Python sollen praxisrelevante Erkenntnisse gewonnen werden. Die Definition und Implementierung von Metriken, sowie deren Visualisierung dienen dabei der objektiven Bewertung der Algorithmen. Anschließend sollen mithilfe eines trainierten GAN’s realistische Bilder des trainierten Datensatzes entwickelt werden, auf denen dann die Klassifizierung angewendet werden soll.

# 2. Datensätze

Im Folgenden werden die für die Klassifikationsaufgaben verwendeten Datensätze vorgestellt.

## 3.1 MNIST-Datensatz

Der MNIST-Datensatz ist ein berühmtes Benchmark in der Welt der künstlichen Intelligenz, insbesondere im Bereich des maschinellen Lernens und der Bilderkennung. Er besteht aus einer umfangreichen Sammlung von insgesamt 70.000 handgeschriebenen Ziffern (0 bis 9), die von verschiedenen Autoren stammen. Jede Ziffer ist in einem festen Raster von 28x28 Pixeln abgebildet, wobei die Bilder als Graustufen-Bilder abgespeichert sind. Hierbei ist jedem Bild einer eindeutigen Klasse (0-9) zugeordnet, welche die Ziffer darstellt, die das Bild repräsentieren soll. Der Datensatz setzt sich aus 60.000 Bildern im Trainingsdatensatz und 10.000 Bildern im Testdatensatz zusammen. Dieser Datensatz, der oft als "Hello World" der Bilderkennung bezeichnet wird, dient als Benchmark und Testumgebung für eine Vielzahl von Klassifizierungs- und Mustererkennungsalgorithmen. Aufgrund seiner klaren Struktur und der geringen Komplexität bietet MNIST eine ideale Grundlage für die Entwicklung und Evaluierung von Modellen im Bereich der KI.

## 3.2 CIFAR10-Datensatz

Ähnlich wie der MNIST-Datensatz ist auch der CIFAR-10-Datensatz ist ein bedeutender Benchmark in der Forschung der künstlichen Intelligenz. Dieser Datensatz besteht aus 60.000 farbigen Bildern in einer Auflösung von 32x32 Pixeln, aufgeteilt in zehn Klassen. Jede Klasse repräsentiert eine spezifische Kategorie: Flugzeuge, Autos, Vögel, Katzen, Hirsche, Hunde, Frösche, Pferde, Schiffe und Lastwagen. Die Bilder in CIFAR-10 sind bunt und vielfältig, was eine realitätsnahe Abbildung der Herausforderungen in der Bilderkennung darstellt. Die 60.000 Bilder sind in Trainings- und Testsets unterteilt, wobei das Trainingsset 50.000 und das Testset 10.000 Bilder umfasst. Diese Aufteilung ermöglicht eine gründliche Evaluation der Modelle, indem sie sowohl während des Trainings als auch in einem unabhängigen Testprozess auf ihre Leistungsfähigkeit überprüft werden. Die Eigenschaften des CIFAR-10-Datensatzes machen ihn zu einer ausgezeichneten Ressource für die Entwicklung und den Vergleich von Bilderkennungsalgorithmen. Seine Herausforderungen liegen in der Unterscheidung von Objekten in Farbbildern in einer geringen Auflösung, was eine robuste und effiziente Merkmalsextraktion erfordert. Die Klassenrepräsentationen spiegeln eine breite Palette von Objekten wider, was sicherstellt, dass Modelle auf unterschiedliche visuelle Merkmale und Kontexte trainiert werden.

# 3. Entwicklung des maschinellen Lernrahmens

## 4.1 Wahl des Frameworks und verwendeter Tools

Im Zuge der Entwicklung meines maschinellen Lernrahmens habe ich mich bewusst für die Kombination von bewährten Technologien entschieden, um Effizienz und Leistungsfähigkeit zu gewährleisten. Die Programmiersprache meiner Wahl ist Python, deren klare Syntax und umfangreiche Bibliotheken das Fundament meiner Implementierung bilden. Als integrierte Entwicklungsumgebung (IDE) nutze ich Visual Studio Code (Vs-Code), eine leistungsstarke Plattform, die eine effiziente Entwicklung und einfache Handhabung meiner Projekte ermöglicht. Für die Modellentwicklung setze ich auf PyTorch, ein fortschrittliches und flexibles Framework für neuronale Netzwerke. Die dynamische Berechnungsgaph von PyTorch erleichtert die Modellierung und Anpassung an verschiedene Aufgaben im Bereich des maschinellen Lernens. Die Evaluation meiner Modelle erfolgt mithilfe von scikit-learn, einer umfassenden Bibliothek für maschinelles Lernen in Python. Scikit-learn bietet eine breite Palette von Metriken, die mir ermöglichen, die Leistung meiner Modelle präzise zu bewerten, von Genauigkeit bis hin zu präzisionsorientierten Metriken wie Recall und F1-Score. Um die Modelle zu testen und experimentelle Iterationen durchzuführen, nutze ich Jupyter Notebooks. Diese interaktive Umgebung ermöglicht es mir, Code, visuelle Darstellungen und textuelle Erläuterungen nahtlos zu kombinieren, was die Entwicklung und Analyse meiner Modelle transparent und nachvollziehbar macht. Matplotlib dient als verlässlicher Plotter für die Visualisierung von Daten und Modellergebnissen. Die Vielseitigkeit von Matplotlib ermöglicht es mir, aussagekräftige Diagramme und Grafiken zu erstellen, um komplexe Zusammenhänge zu verstehen und die Ergebnisse effektiv zu kommunizieren. In der Summe bieten Python, Vs-Code, PyTorch, scikit-learn, Jupyter Notebooks und Matplotlib eine ausgereifte technologische Basis für die Entwicklung, Evaluierung und Visualisierung meiner maschinellen Lernmodelle in einem effizienten und benutzerfreundlichen Rahmen.

## 4.2 Erstellung der Datenpipelines

Die Datenpipelines bilden das Rückgrat eines erfolgreichen maschinellen Lernprojekts, und ihre Implementierung ist entscheidend für die Qualität der Modelle. Bei der Arbeit mit dem MNIST- und CIFAR-10-Datensatz habe ich mich für die Verwendung der torchvision-Bibliothek entschieden, um diese effizient zu laden und zu verarbeiten. Für den MNIST-Datensatz habe ich sowohl einen Trainings- als auch einen Testdatensatz erstellt. Der Trainingssatz besteht dabei aus 60.000 der insgesamt 70.000 Bildern und der Testdatensatz aus den übrigen 10.000 Bildern. Der gesamte Trainingsdatensatz ist weiter in 80% Trainingsdaten (48.000 Bilder) und 20% Validierungsdaten (12.000 Bilder) unterteilt. Diese Aufteilung ermöglicht es, das Modell auf einer ausreichenden Menge von Trainingsdaten zu trainieren, während die Validierungsdaten die Leistung des Modells überwachen. Darüber hinaus habe ich sowohl für den Trainingssatz als auch für den Testsatz eine normalisierte Version erstellt. Hierbei werden die Datensätze mit der sogenannten Z-Score-Normalisierung normalisiert. Dabei wird der Mittelwert des Datensatzes von jedem Bildpunkt subtrahiert und durch die Standardabweichung dividiert. Dies erfolgt für alle Bildkanäle. Dadurch werden die Eingangsdaten so normalisiert, dass sie eine Standardabweichung von 1 haben. Dies ermöglicht es mir, den Einfluss der Normalisierung auf die Modellleistung zu evaluieren. Für den CIFAR-10-Datensatz habe ich eine ähnliche Vorgehensweise gewählt. Hier gibt es einen Trainingsdatensatz bestehend aus 50.000 der 60.000 Bildern und einen Testdatensatz, bestehend aus 10.000 Bildern, wobei der Trainingssatz erneut in 80% Trainingsdaten (40.000 Bilder) und 20% Validierungsdaten (10.000 Bilder) aufgeteilt ist. Zusätzlich zu den normalisierten Versionen der Datensätze habe ich außerdem einen speziellen Datensatz erstellt, der sowohl alle originalen CIFAR-10-Bilder als auch die Bilder in augmentierter Form enthält. Diese augmentierten Daten wurden durch zufällige Rotation, Spiegelung und andere Transformationen erzeugt und dienen dazu, die Robustheit des Modells gegenüber verschiedenen Ansichten der Daten zu stärken. Insgesamt ermöglicht die Implementierung dieser Datenpipelines eine effektive Verwaltung und Nutzung der MNIST- und CIFAR-10-Datensätze in verschiedenen Formen und Zuständen. Dies bildet die Basis für ein umfassendes Training und eine gründliche Bewertung der maschinellen Lernmodelle in Bezug auf Normalisierung und Datenaugmentierung.

## 4.3 Verwendete Metriken

In meinem KI-Projekt verwende ich eine sorgfältige Auswahl von Metriken, um die Leistung meiner neuronalen Netzwerke umfassend zu evaluieren. Für Klassifikatoren jeglicher Art setze ich auf klassische Maße wie Accuracy, F1-Score, Präzision und Recall, um eine ganzheitliche Bewertung der Vorhersagen zu erhalten. Diese Metriken ermöglichen es mir, nicht nur die Gesamtgenauigkeit, sondern auch die spezifischen Aspekte wie den Ausgleich zwischen Präzision und Rückruf zu berücksichtigen. Für Autoencoder liegt der Fokus auf dem Mean Squared Error (MSE), der den durchschnittlichen quadratischen Fehler zwischen den originalen und rekonstruierten Daten misst. Zusätzlich implementiere ich eine Validierung während des Trainings, um die Modellleistung auf einem separaten Datensatz zu überwachen. Nach dem Training erfolgt eine eingehende Evaluation der Trainingsgenauigkeit und des Trainingsverlusts jedes Netzwerks. Falls verfügbar, werden auch die Validierungsgenauigkeit und das Validierungsloss berücksichtigt. Diese umfassende Metrikenauswahl gewährleistet eine gründliche Beurteilung der Modellleistung und trägt zur Optimierung meiner KI-Modelle bei.

## 4.4 Methodik

In meinem Projekt verfolge ich einen methodischen Ansatz zur Klassifizierung und Generierung von Bildern unter Verwendung verschiedener neuronaler Netzwerkarchitekturen. Zunächst implementiere ich ein Multilayer Perceptron (MLP) für die Klassifizierung der beiden Datensätze. Durch die Anwendung von Hyperparameter-Optimierung strebe ich die Maximierung der Modelleffizienz an und suche die optimalen Konfigurationen für das MLP. Die Evaluation der Ergebnisse umfasst eine Analyse von Trainings- und Validierungsverlusten sowie Genauigkeiten, um die Leistung des MLPs zu bewerten. Im nächsten Schritt integriere ich einen Autoencoder, um die Rekonstruktion der Daten zu untersuchen. Dieser Unsupervised-Learning-Ansatz ermöglicht es mir, die Fähigkeit des Autoencoders zu beurteilen, bedeutungsvolle Merkmale zu extrahieren und die ursprünglichen Bilder erfolgreich wiederherzustellen. Die Analyse der Rekonstruktionsverluste bietet Einblicke in die Qualität der erlernten Repräsentationen. Der Vergleich der rekonstruierten Bilder mit den Originalbildern ermöglicht eine Einsicht in die Effizienz des Auto-Encoders Darauf aufbauend nutze ich den Encoder-Teil des Autoencoders in Kombination mit einem separaten Klassifikator, um erneut eine Klassifizierung durchzuführen. Die Idee ist, durch den Encoder die Eingangsdaten in eine niederdimensionale, kompakte, aber dennoch repräsentative Form zu überführen, und in dieser die Klassifizierung vorzunehmen. Die erneute Evaluierung der Ergebnisse ermöglicht es mir, die Auswirkungen der Verwendung des Encoder-Teils auf die Klassifizierungsgenauigkeit zu verstehen und zu bewerten. Für eine weitergehende Analyse implementiere ich Convolutional Neural Networks (CNNs) für die Klassifizierung der Daten. Durch die Anwendung von Hyperparameter-Optimierung optimiere ich die Netzwerkarchitektur, um sicherzustellen, dass das CNN optimal auf die strukturierten Merkmale von Bildern reagiert. Auch hier wird mit Trainings, und Validierungsmetriken die Effizienz beurteilt und gegebenenfalls das Modell angepasst. Ein Schwerpunkt meines Projekts liegt auch auf der Generierung von Bildern unter Verwendung von Generative Adversarial Networks (GANs). Durch das Training des GANs auf beide Datensätze beabsichtige ich, realistische und bisher ungesehene Beispiele zu generieren. Abschließend teste ich alle entwickelten Klassifikationsmodelle auf den durch das GAN generierten Bildern. Dieser Schritt ermöglicht es mir, die Robustheit und Generalisierungsfähigkeit meiner Modelle zu überprüfen und sicherzustellen, dass sie effektiv mit generierten Daten umgehen können. Abschließend werde ich die Effizienz aller Klassifikatoren evaluieren und vergleichen, um eine Übersicht über deren Vor- und Nachteile, sowie deren Verhalten zu erhalten.

# 4. Implementierung

In diesem Teil werde ich grob auf die Implementierung des Projektes in Python eingehen. Dabei gehe ich auf die Implementierung der zuvor beschriebenen Daten Pipeline ein, und auf die verwendeten Machine-Learning Modelle. Die Implementierung und Entwicklung des neuronalen Netzwerks erfolgte größtenteils in der integrierten Entwicklungsumgebung (IDE) Visual Studio Code (VSCode). VSCode bietet eine umfassende Umgebung für die effiziente Entwicklung von Python-Anwendungen und ermöglicht eine nahtlose Integration mit verschiedenen Bibliotheken und Frameworks.

## 5.1 DataManager.py

Diese Python Datei übernimmt die Aufgabe des Ladens und Verarbeitung der verwendeten Datensätze. Sie besteht grundsätzlich aus der DataManagerSingleton-Klasse und der DataSetConfig. Letztere bietet ein Gerüst zum Speichern eines Datensatzes. Die Klasse DataSetConfig wurde entwickelt, um die Konfigurationen und Attribute eines Datensatzes zu verwalten. Hierbei werden Eigenschaften wie der Index, Klassen, Anzahl der Klassen, Dimensionen und Anzahl der Kanäle definiert. Die Klasse enthält

auch Attribute für verschiedene Versionen desselben Datensatzes, einschließlich Trainings-, Test- und Validierungssets sowie deren normalisierte Varianten. Die übergeordnete Klasse DataManagerSingleton übernimmt die Verantwortung für das Laden und Verwalten von Datensätzen, insbesondere für die CIFAR-10 und MNIST Datensätze. Diese ist eine Singleton-Klasse, um sicherzustellen, dass nur eine Instanz dieses Typs vorhanden sein kann. Die Methode load_CIFAR10_dataset lädt den CIFAR-10 Datensatz und erstellt DataLoader für Trainings-, Test- und Validierungssets sowie deren normalisierte Versionen. Eine ähnliche Methode, load_CIFAR10_dataset_augmented, führt zusätzliche Datenverstärkungstransformationen durch, um den Trainingsdatensatz zu erweitern. Die Methode load_MNIST_dataset hingegen lädt den MNIST Datensatz und erstellt entsprechende DataLoader in normalisierter und nicht normalisierter Form. Die geladenen Datensätze werden in Klassenvariablen vom Typ DataSetConfig abgespeichert, um über die Klasse auf die Datensätze zugreifen zu können. Die Struktur der Klasse DataManagerSingleton bietet Modularität, indem verschiedene Datensätze einfach geladen und verwaltet werden können. Sie unterstützt Datenverstärkung, Normalisierung und ermöglicht die Aufteilung in Trainings-, Test- und Validierungssets.

## 5.2 NeuronalNetworkBase.py

Die Basisklasse für neuronale Netze, NeuralNetworkBase, wurde geschaffen, um eine abstrakte Vorlage für die Implementierung von neuronalen Netzwerken in PyTorch zu bieten. Die Klasse erbt von PyTorch's nn.Module und stellt sicher, dass sie als abstrakte Klasse definiert ist, indem sie das ABC (Abstract Base Class) Modul importiert. Das ermöglicht es, abstrakte Methoden zu definieren, die in den abgeleiteten Klassen implementiert werden müssen. Die abstrakte Klasse enthält zwei Schlüsselmethoden, train_model und test_model, die als abstrakte Methoden deklariert sind. Diese Methoden müssen in den abgeleiteten Klassen implementiert werden, um das Training und Testen von neuronalen Netzwerken zu ermöglichen. Dies bietet eine klare Struktur für die Erweiterung der Funktionalität je nach den Anforderungen eines bestimmten Modells.

Darüber hinaus bietet die Basisklasse nützliche Methoden für das Speichern und Laden von Modellgewichten. Die save_model Methode ermöglicht es, den Zustand des Modells in einer Datei zu speichern, während die load_model Methode den gespeicherten Zustand aus einer Datei wiederherstellt. Dies ist besonders hilfreich, um trainierte Modelle für zukünftige Verwendung oder Weiterentwicklung zu verwenden. Insgesamt stellt die NeuralNetworkBase eine flexible und erweiterbare Grundlage für die Implementierung und Verwaltung von neuronalen Netzwerken in PyTorch dar, auf dessen Basis die Implementierung spezifischer Netzwerkmodelle erfolgen kann.

## 5.3 Multilayer Perceptron (MLP)

Das implementierte mehrschichtige Perzeptron (MLP) ist eine abgeleitete Klasse von der vorher definierten Basisklasse NeuralNetworkBase. Diese MLP-Klasse ermöglicht die einfache Konstruktion und Verwaltung von MLP-Modellen in PyTorch. Bei der Initialisierung wird die Eingabegröße, die Anzahl der versteckten Schichten, die Anzahl der Klassen und die Option zur Verwendung der Batch-Normalisierung berücksichtigt. Die Methode create_layers wird verwendet, um die Schichten des MLP dynamisch zu erstellen, wobei die Anzahl der Neuronen in den versteckten Schichten durch den Parameter hidden_layers festgelegt wird. Als Aktivierungsfunktion wird hier die ReLU-Aktivierung gewählt. Für das Training des Modells steht die Methode train_model zur Verfügung, die auf einem Trainingsdatensatz iteriert, den Verlust minimiert und die Gewichtungen anpasst. Optional kann die Validierung auf einem separaten Validierungsdatensatz durchgeführt werden. Die Methode test_model evaluiert die Leistung des trainierten Modells auf einem Testdatensatz und gibt Genauigkeit, Präzision, Recall und F1-Score aus. Zusätzlich gibt die Methode classify_images die vorhergesagten Labels für

eine Menge von Bildern zurück.Für eine visuelle Inspektion von Fehlklassifikationen bietet die Methode show_misses die Möglichkeit, missklassifizierte Bilder aus dem Testdatensatz anzuzeigen. Dieses MLP bietet somit eine solide Grundlage für die Implementierung, das Training und die Bewertung von neuronalen Netzwerken für Klassifikationsaufgaben.

## 5.4 Auto-Encoder

Der implementierte Autoencoder ist eine spezialisierte Klasse, die von der vorher definierten Basisklasse NeuralNetworkBase abgeleitet ist. Dieser Autoencoder besteht aus einem Encoder, einem latenten Raum und einem Decoder. Die Methode create_layers wird verwendet, um die Schichten für den Encoder und Decoder entsprechend den übergebenen Parametern zu erstellen. Der Encoder besteht aus linearen Schichten mit ReLU-Aktivierungsfunktionen, während der Decoder die Spiegelung des Encoders ist, mit Ausnahme der Aktivierungsfunktion im Ausgabelayer, die eine Sigmoid-Funktion ist. Der forward-Methode des Autoencoders akzeptiert Eingabedaten und führt sowohl den Encoder als auch den Decoder aus. Zusätzlich wird eine Klassifizierungsschicht hinzugefügt, die den latenten Raum zu Klassenzuweisungen abbildet. Die Methode train_model ermöglicht das Training des Autoencoders je nach ausgewähltem Modus, wobei Trainings- und Validierungsverluste sowie Genauigkeiten aufgezeichnet werden. Die Methode test_model evaluiert das Modell entsprechend dem gewählten Modus auf einem Testdatensatz und gibt Genauigkeit, Präzision, Recall und F1-Score (für den "classify"-Modus) oder den durchschnittlichen MSE (für den "decode"-Modus) aus. Die Methode show_misses visualisiert missklassifizierte Bilder aus dem Testdatensatz im "classify"-Modus. Schließlich ermöglichen die Methoden encode und decode die separate Verwendung des Encoder- bzw. Decoder-Teils des Autoencoders für spezifische Anwendungen.

## 5.5 Convolutional Auto-Encoder

Der implementierte Convolutional Autoencoder (ConvAE) erweitert den klassischen Autoencoder für die Verarbeitung von Bilddaten, insbesondere für verschiedene Kanäle. Im Gegensatz zum regulären Autoencoder verwendet der ConvAE Convolutional- und Transpose-Convolutional-Schichten für den Encoder und Decoder, um räumliche Strukturen in den Bildern besser zu erfassen. Die Architektur enthält auch eine Klassifikationsschicht, die den latenten Raum zu Klassenzuweisungen abbildet. Der Aufbau der Klasse ist hierbei grundsätzlich identisch zum normalen Auto-Encoder. Er unterscheidet sich zum normalen Auto-Encoder nur in der Netzwerkarchitektur.

## 5.6 Convolutional Neural Network (CNN)

Eine weitere verwendete Netzwerkarchitektur ist das Convolutional Neural Network (CNN). Die CNN-Architektur besteht aus zwei Faltungsschichten mit ReLU-Aktivierung und Max-Pooling, gefolgt von vollständig verbundenen Schichten. Die Anzahl der Kanäle, die Größe der vollständig verbundenen Schicht und die Option zur Verwendung von Batch-Normalisierung sind konfigurierbar. Das Netzwerk enthält Methoden zum Trainieren, Testen und Klassifizieren von Bildern. Im Konstruktor wird das CNN mit Faltungsschichten, Batch-Normalisierung (optional), Max-Pooling und vollständig verbundenen Schichten definiert. Die forward-Methode skizziert den Vorwärtsdurchlauf durch das Netzwerk. Die Methode create_layers generiert eine Liste von Faltungs-schichten mit ReLU-Aktivierung, Max-Pooling und Dropout, was nützlich ist, um eine ähnliche Architektur in einem anderen Kontext zu erstellen. Die Methode train_model ermöglicht das Training des CNN mit bereitgestellten Datenladern, einer festgelegten Anzahl von Epochen, einem Kriterium (Verlustfunktion) und einem Optimierer. Sie zeichnet Trainings- und Validierungsverluste sowie Genauigkeiten während jeder Epoche auf, und gibt

diese als Rückgabewert zurück. Die Methode test_model bewertet das CNN anhand eines Testdatensatzes und berechnet Genauigkeit, Präzision, Recall und F1-Score. Insgesamt bietet die CNN-Implementierung Flexibilität in Bezug auf die Konfiguration der Architektur und stellt Methoden für das Training, Testen und die Klassifizierung bereit, wodurch sie für verschiedene Bildklassifizierungsaufgaben geeignet ist.

## 5.7 Deep Convolutional Neural Network (DeepCNN)

Die implementierte Klasse DeepCNN repräsentiert ein tiefes neuronales Netzwerk (CNN) für maschinelles Lernen, insbesondere für Bildklassifikationsaufgaben. Die Struktur der Klasse ist hierbei identisch der des CNN. Im Vergleich zu einem herkömmlichen CNN zeichnet sich dieses Modell durch eine komplexere Architektur aus und mehreren Schichten aus.

## 5.7 Generative Adversarial Network (GAN):

Ein weiteres Implementiertes Netzwerk ist das Generative Adversarial Network (GAN). Die Architektur besteht aus einem Generator und einem Diskriminator. Der Generator erzeugt Bilder aus zufälligen Rauschvektoren, während der Diskriminator versucht, zwischen echten und generierten Bildern zu unterscheiden. Der Generator verwendet eine sequenzielle Schicht mit linearen Transformationen, Leaky ReLU-Aktivierung und einer abschließenden Tanh-Aktivierungsfunktion. Der Diskriminator besteht ebenfalls aus einer sequenziellen Schicht, verwendet jedoch Leaky ReLU-Aktivierung, Dropout und eine Sigmoid-Aktivierungsfunktion am Ende, um eine binäre Klassifizierung durchzuführen, da er nur zwischen echt und falsch unterscheiden muss. Das GAN wird von der Basisklasse NeuralNetworkBase abgeleitet, die gemeinsame Funktionen für neuronale Netze bereitstellt. Die Trainingsmethode des GANs umfasst sowohl das Training des Generators als auch des Diskriminators. Die Verlustfunktionen werden dabei auf die jeweiligen Modelle angewendet. Zusätzlich werden die Verluste während des Trainingsprotokolliert. Die Implementierung bietet Methoden zum Speichern und Laden des trainierten GAN-Modells, um es später für die Bildgenerierung oder weitere Trainingsläufe wiederzuverwenden. Insgesamt stellt dieses GAN einen einfachen Rahmen für die Generierung von Bildern dar, wobei der Generator versucht, den Diskriminator zu täuschen, und der Diskriminator versucht, die Authentizität der Bilder zu bewerten.

## 5.8 Deep Convolutional Generative Adversarial Network (DCGAN)

Diese Klasse implementiert ein Generatives Adversarial Network (GAN) unter Verwendung der Deep Convolutional GAN (DCGAN)-Architektur. Im Wesentlichen handelt es sich um dasselbe Grundkonzept wie bei einem herkömmlichen GAN, jedoch wurde die Netzarchitektur speziell angepasst, um bilddatenbasierte Generierungsaufgaben besser zu bewältigen. Hierzu wurden Faltungsschichten in die Netzwerkarchitektur hinzugefügt. Hierbei gibt es Generator sowie Diskriminator in zwei verschiedenen Varianten, die auf die beiden Datensätze abgestimmt sind. Die beiden Versionen unterscheiden sich nur in der Komplexität des Netzwerks. Da CIFAR10 Bilder wesentlich komplexer sind als MNIST-Bilder, ist es notwendig eine komplexe Struktur zu verwenden.

# 5. Ausführung und Tests

Während der Großteil der Codes in VSCode programmiert ist, wird die Ausführung und Testphase der entwickelten Modelle jedoch bewusst in der Jupyter Notebook Umgebung durchgeführt. Diese

Entscheidung wurde getroffen, um die Möglichkeit zu nutzen, den Code schrittweise auszuführen, visuelle Darstellungen von Daten zu erstellen und die Ergebnisse in einem interaktiven Format zu präsentieren.

Die Tests umfassten dabei nicht nur die Überprüfung der Modellgenauigkeit, sondern auch die Auswertung von Metriken wie Präzision, Recall und F1-Score. Darüber hinaus wurden Visualisierungen von Trainingsverläufen und Ergebnissen erstellt, um ein umfassendes Verständnis für das Verhalten des Modells zu gewinnen. Auf Grund mangelnden Platzes im Bericht werde ich dabei nur auf die Ergebnisse der Tests eingehen.

## 6.1 Vorbereitung

Vor der Nutzung der Klassen müssen zunächst einige Vorbereitungen getroffen werden. Hierzu werden zunächst die benötigten Bibliotheken in das Projekt integriert. Des Weiteren wird hier die Instanz der DataManagerSingleton Singleton-Klasse erstellt, um auf die verschiedenen Datensätze zugreifen zu können.

Außerdem wird angestrebt ein grundsätzliches Verständnis für die verwendeten Datensätze zu schaffen. Dafür werden einige beispielhafte Bilder der verschiedenen Datensätze angezeigt, sowie die dazugehörenden Klassen/Labels.

## 6.2 MLP

Im ersten Schritt meiner Testphase habe ich ein zufälliges Multilayer-Perceptron (MLP) erstellt und es mit den üblichen Parametern auf den MNIST-Datensatz trainiert und getestet. Dieses initiale Training diente dazu, eine Grundlinie für die Leistung des Modells zu schaffen. Im Anschluss daran habe ich eine systematische Hyperparameteroptimierung durchgeführt, um die optimalen Parameter für das MLP zu finden. Hierbei kam die Grid-Suche zum Einsatz, bei der verschiedene Kombinationen von Hyperparametern getestet wurden, darunter Lernrate, Anzahl der versteckten Schichten, Anzahl der Neuronen pro Schicht und die Möglichkeit der Batch-Normalisierung. Da eine solche Grid-Suche sehr ressourcenaufwendig ist, lässt sich hier nur eine begrenze Anzahl Hyperparametern testen. Nach der Extraktion der bestmöglichen Hyperparameterkombination habe ich auf Basis dieser das MLP-Modell aufgebaut und trainiert. Als Aktivierungsfunktion der versteckten Schichten wurde hier die ReLU-Aktivierung verwendet, und eine Softmax-Aktivierung in der Ausgabeschicht. Dabei habe ich als Verlustfunktion Den Cross-Entropy Loss, und als Optimierungsfunktion den Adam-Optimierer verwendet. Die Anzahl der Neuronen in der Eingabeschicht ergibt sich aus der Anzahl der Bildelemente eines Bildes, also $Höhe * Breite * Kanäle$ Für den MNIST-Datensatz also $28x28x1 = 784$. Die Anzahl der Neuronen in der Ausgabeschicht ergibt sich aus der Anzahl zu klassifizierenden Klassen. Diese ist für den MNIST-Datensatz 10 (Ziffern 0-9). Während des Trainings habe ich die Trainings- und Validierungsverluste sowie die Genauigkeiten über die Epochen hinweg verfolgt und visualisiert. Dabei haben sich folgende Verlust- und Genauigkeitskurven ergeben:

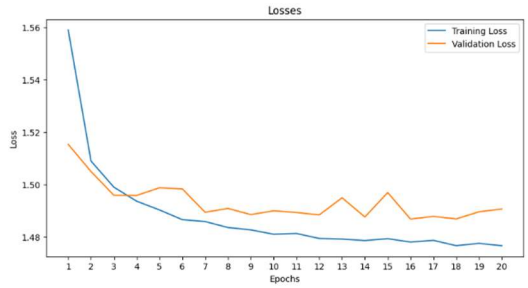


*Abb: 1: Trainings-und Validierungsverlust des MLP (MNIST)*

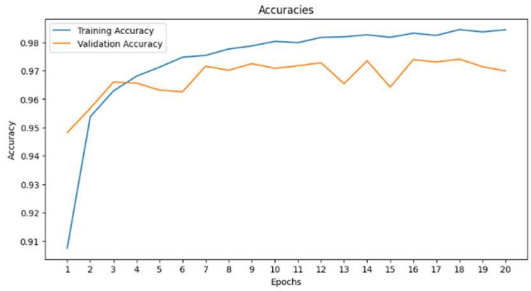


*Abb: 2: Trainings-und Validierungsgenauigkeit des MLP (MNIST)*

Man sieht, dass sowohl Trainings als auch Validierungsverlust kleiner werden, und die Genauigkeiten zunehmen. Dabei sind die Validierungsgraphen allerdings sehr viel „statischer". Das liegt wohlmöglich daran, dass der Datensatz sehr simpel ist, und bereit in den ersten 5 Epochen der Großteil der Merkmale extrahiert wurde. Nach dem Training habe ich das Modell anhand der definierten Metriken getestet, um eine umfassendere Bewertung seiner Leistung zu erhalten. Hierbei ergaben sich folgende Werte:

| Genauigkeit | Präzision | Recall | F1 |
|---|---|---|---|
| 0.9787 ≈ 98% | 0.9787 ≈ 98% | 0.9787 ≈ 98% | 0.9786 ≈ 98% |

Es ergab sich eine gute Klassifizierung, wie anhand der Metriken zu erkennen war. Um ein tieferes Verständnis für die Stärken und Schwächen des MLPs zu gewinnen, habe ich auch einige falsch klassifizierte Beispiele analysiert und visualisiert. Dies ermöglichte mir, Muster und Trends in den Fehlklassifikationen zu erkennen. Dabei ist aufgefallen, dass gerade die Zahlen „4" und „9" oft „verwechselt" werden.

Dasselbe Vorgehen habe ich auch für den CIFAR10-Datensatz angewendet. Auch hier habe ich eine Grid-Suche angewendet, um die optimalen Hyperparameter zu finden. Als Aktivierungsfunktion wird hier die ReLU-Aktivierung verwendet und eine Sigmoid-Aktivierung in der Ausgabeschicht. Anschließend habe ich das Modell aufgebaut und trainiert. Dabei ergibt sich die Anzahl der Neuronen in der Eingabeschicht wieder aus der Anzahl der Bildelemente pro Bild. Für CIFAR10-Bilder ergibt sich also: $32 * 32 * 3 = 3072$. Die Anzahl der Neuronen in der Ausgabeschicht ergibt sich wieder aus der Anzahl zu klassifizierenden Klassen. Im CIFAR10-Datensatz sind das ebenfalls 10. Es wurde wieder der Adam-Optimierer sowie der Cross-Entropy-Loss verwendet. Dabei ergaben sich folgende Genauigkeits- und Verlustkurven:

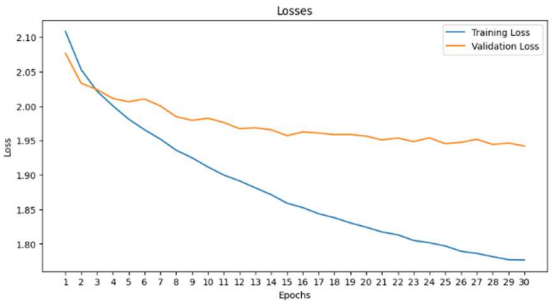


*Abb: 3: Trainings- und Validierungsloss*

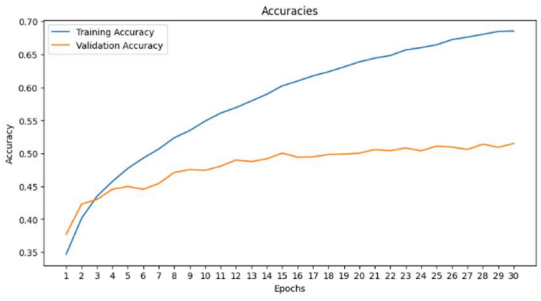


*Abb: 4: Trainings- und Validierungsgenauigkeit*

Die Validierungsgraphen des CIFAR10-Datensatzes scheinen dabei sehr viel weniger zu fluktuieren. Auch hier ändern sich die Validierungsgraphen im Vergleich zu den Trainingsgraphen nur minimal. Nach dem Testen ergaben sich folgende Metriken:

| Genauigkeit | Präzision | Recall | F1 |
|---|---|---|---|

| | | | |
|---|---|---|---|
| 0.515 ≈ 52% | 0.5177 ≈ 52% | 0.515 ≈ 52% | 0.5152 ≈ 52% |

Diese Werte sind deutlich schlechter als die des MNIST-Datensatzes. Dies liegt an der erhöhten Komplexität des CIFAR10-Datensatzes. Daraufhin habe ich wieder einige falsch Klassifizierte Bilder dargestellt. Dabei ergab sich allerdings kein übergeordneter Zusammenhang warum diese falsch klassifiziert wurden. Das Modell ist wohlmöglich nicht komplex genug wichtige Merkmale vollständig zu extrahieren.

## 6.3 Auto Encoder

Nun wurde ein AutoEncoder verwendet um Bilder der Datensätze möglichst genau zu Rekonstruieren. Mithilfe dieses soll später auch eine Klassifizierung durchgeführt werden. Zunächst wurde wieder ein einfacher Auto-Encoder mit Beispielhaften Parametern aufgebaut und trainiert. Als Aktivierungsfunktion habe ich den MSE-Loss und als Optimierer den Adam-Optimierer verwendet. Der Auto Encoder wurde nun auf den normalisierten MNIST-Datensatz trainiert. Nach dem Training gab sich folgender Verlauf für den Trainings- und Validierungsverlust:

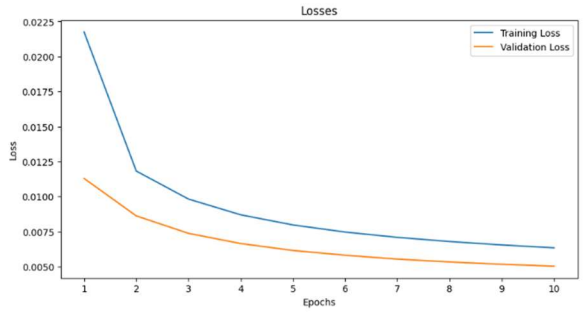


*Abb: 5: Trainings- und Validierungsloss des AutoEncoders (MNIST)*

Wobei erkenntlich ist, dass beide Verläufe stark gegen 0 gehen. Beim Testen des Modells ergibt sich ein MSE von 0.0048. Dieser Wert lässt vermuten, dass die Eingangsbilder wohl sehr gut rekonstruiert werden können. Daraufhin habe ich mir einige Originalbilder des Datensatzes, sowie die vom AutoEncoder rekonstruierten Bilder anzeigen lassen, um ein Gefühl über seine Funktion zu erhalten.

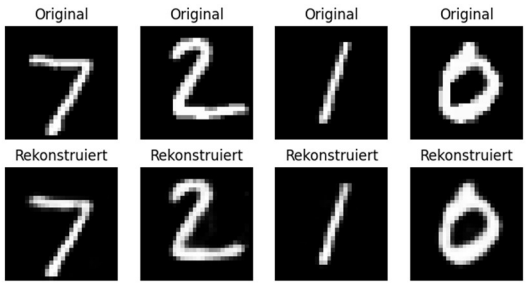


*Abb: 6: Originalbilder und deren Rekonstruktion vom Auto Encoder*

Man erkennt kaum einen unterschied zu den Originalbildern, was zeigt, dass der Auto Encoder eine sehr gute Rekonstruktion der Bilder erschaffen kann. Aus diesem Grund habe ich hier auf eine Hyperparameteroptimierung verzichtet.

Für den CIFAR10-Datensatz bin ich auf gleiche Weise vorgegangen wie zuvor. Zuerst habe ich den Auto Encoder mit initialen Hyperparametern aufgebaut. Hier habe ich dieselben Parameter und dieselbe Netzwerkarchitektur wie beim MNIST-Datensatz gewählt, wobei die Anzahl der Neuronen in der Ein-

und Ausgabeschicht entsprechend angepasst wurden. Nach dem Training ergaben sich folgende Verlustkurven.

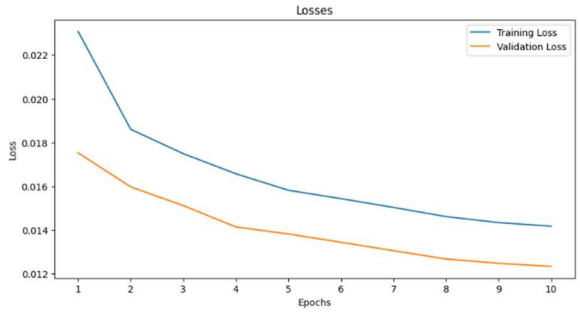


*Abb: 7: Trainings-und Validierungsloss des Auto Encoders (CIFAR10)*

Der Test des Modells auf den Testdatensatz ergab einen MSE von 0.012. Dieser klingt zunächst niedrig, ist allerdings eine Zehnerpotenz höher im Vergleich zum MSE des MNIST-Datensatzes. Daraufhin habe ich wieder einige Bilder und deren rekonstruierte Version des Auto Encoders geplottet.

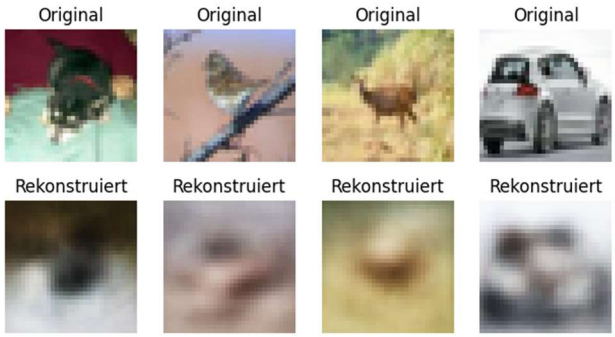


*Abb: 8: Originalbilder und Rekonstruktion des Auto Encoders (CIFAR10)*

Dabei fällt auf, dass die Rekonstruktionen der Bilder sehr ungenau sind. Man könnte an dieser Stelle versuchen eine Hyperparameter-optimierung zu verwenden, um eine Verbesserung des Auto Encoders anzustreben, ich habe mich allerdings dazu entschlossen einen anderen Ansatz zu gehen. Da die Rekonstruktionen sehr ungenau sind, und wohlmöglich auch mit perfekter Anpassung nur minimal verbessert werden kann, werde ich für den CIFAR10-Datensatz einen Convolutional Auto Encoder verwenden. Dieser erweitert den regulären Auto Encoder durch Faltungsschichten.

## 6.4 Convolutional Auto Encoder

Da der einfache Auto Encoder bereits zufriedenstellende Ergebnisse auf den MNIST-Datensatz erzielt, nutze ich den Convolutional-Auto Encoder nur für den CIFAR10-Datensatz. Diesen habe ich mit üblichen Hyperparameter erstellt und anschließend trainiert Als Aktivierungsfunktion wurde wieder die ReLU-Aktivierung verwendet, und die Sigmoid-Aktivierung in der Ausgabeschicht. Die Verlustfunktion ist der MSE-Loss und der Optimierer der Adam-Optimierer. Während des Trainings ergaben sich dabei folgende Verluste für Training- und Validierung:

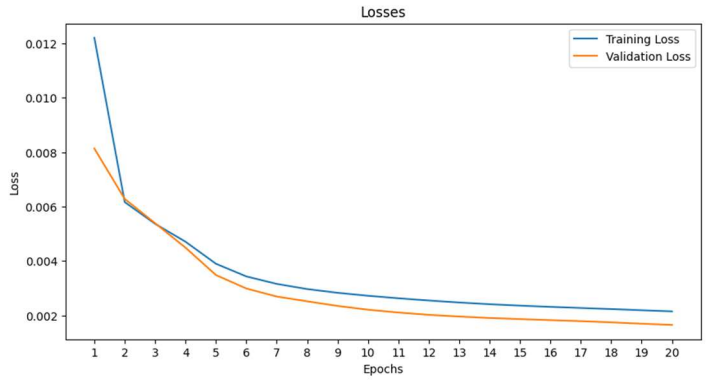


*Abb: 9: Trainings- und Validierungsloss des Convolutional Auto Encoder (CIFAR10)*

Beim Testen des Modells ergab sich ein MSE von 0.0013, was deutlich besser gegenüber dem gewöhnlichen Auto Encoder ist. Einige Rekonstruierte Bilder sind unten dargestellt.

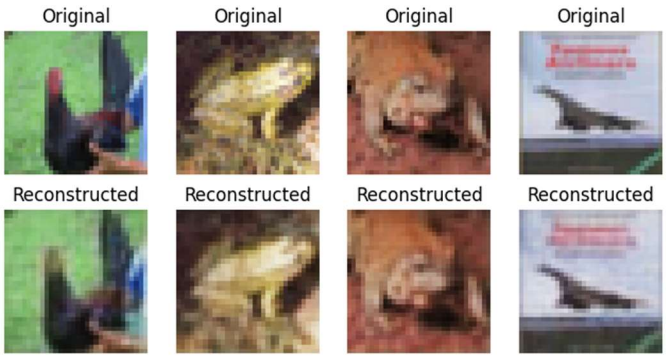


*Abb: 10: Originalbilder und Rekonstruktionen vom Convolutional Auto Encoder (CI4FAR10)*

Diese Bilder stellen eine sehr genaue Rekonstruktion der Eingangsbilder dar.

## 6.5 Auto Encoder zur Klassifizierung

Im Folgenden soll der Auto Encoder als Feature Extractor verwendet werden, und damit eine Klassifikation erreicht werden. Dafür wird der Decoder Teil des Auto Encoders entfernt, und stattdessen eine vollständig verbundene Schicht am Ende des Encoders verwendet, mit der eine Klassifikation erreicht werden soll. Dafür wurde wieder eine Hyperparameter Optimierung vorgenommen und anhand der gefundenen Werte das Modell aufgebaut. Dabei wurde als Verlustfunktion der Cross-Entropy Loss und als Optimierer der Adam Optimierer verwendet. Die Visualisierung der Genauigkeit und Verluste ist unten dargestellt.

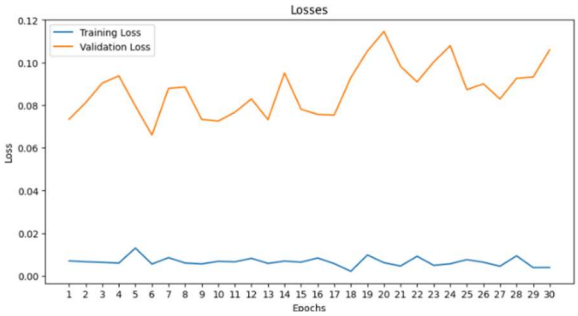


*Abb 11: Trainings- und Validierungsloss*

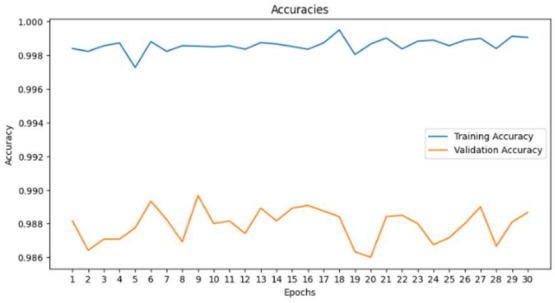


*Abb: 11: Trainings- und Validierungsgenaugikeit*

Das Testen des Modells ergab folgende Werte für die Metriken.

| Genauigkeit | Präzision | Recall | F1 |
|---|---|---|---|
| 0.9820 ≈ 98% | 0.9820 ≈ 98% | 0.9820 ≈ 98% | 0.9819 ≈ 98% |

Es Fällt auf, dass sich die Kurven nicht großartig einem Wert annähern, und stark fluktuieren. Dies liegt wohlmöglich daran, dass die Klassifikation bereits in der ersten Epoche maximal ist, und im Verlauf keine großen Änderungen mehr erfährt. Grund dafür ist, dass durch den Encoder die sowieso simplen Eingangsbilder noch niederdimensionaler werden, und somit noch einfacher zu klassifizieren sind. Deshalb habe ich denselben Prozess erneut mit einer einzigen Lernepoche getestet. Und es ergaben sich ähnliche Ergebnisse für die Metriken.

Es ergibt sich eine minimal bessere Klassifizierung gegenüber dem MLP. Daraufhin habe ich erneut einige falsch Klassifizierte Bilder anzeigen lassen.

Beim CIFAR10-Datensatz bin ich ähnlich vorgegangen, nur dass ich hier den vorher erstellten Convolutional Auto Encoder als Feature Extractor verwendet habe. Das Training des Modells hat dabei folgende Kurven ergeben.

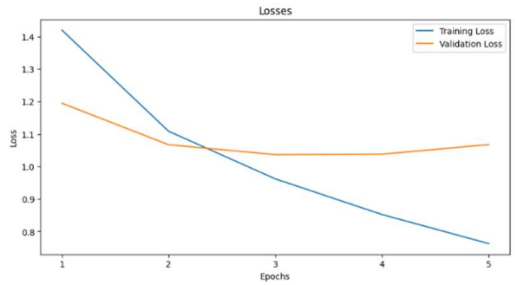


*Abb: 12: Trainings- und Validierungsloss (CIFAR)*

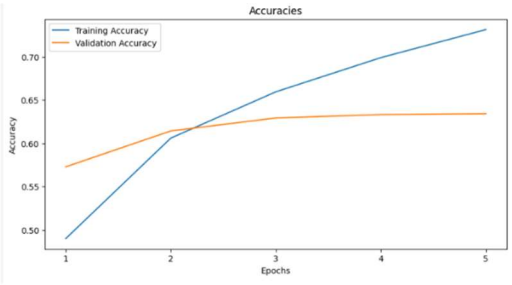


*Abb: 13: Training- und Validierungsgenaugikeit (CIFAR10)*

Obwohl nur 5 Epochen trainiert wurden, scheint es so ob bei Epoche 5 schon langsam ein Overfitting eintritt. Auch bei diesem Datensatz wurden scheinbar sehr schnell die Merkmale erlernt. Dabei ergaben sich folgende Werte für die Metriken.

| Genauigkeit | Präzision | Recall | F1 |
|---|---|---|---|
| 0.6346≈ 63% | 0.6500≈ 65% | 0.6346≈ 63% | 0.6376 ≈ 64% |

Die Werte haben sich im Vergleich zum MLP besonders stark verbessert. Es ergibt sich eine Verbesserung um ca. 25%.

## 6.6 Klassifizierung mit CNN

Nun soll ein Convolutional Neural Network die Klassifizierungsaufgabe übernehmen. Diese sind speziell für diese Aufgabe entwickelt worden und sollten sehr guten darin abschneiden. Für den MNIST-Datensatz wurde hier wieder eine Hyperparameter-Optimierung mit Grid-Suche verwendet, auf dessen Basis dann das CNN-Modell instanziiert wurde. Hierbei wurde die ReLU-Aktivierung für die Schichten des Netzwerkes verwendet, sowie der Adam-optimierer und die Cross-Entropy Loss. Durch das Training ergaben sich folgende Kurven.

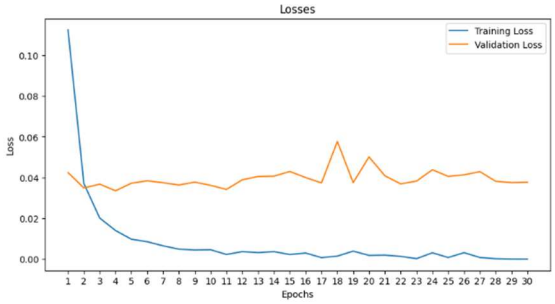


*Abb: 14: Trainings- und Validierungsloss des CNN (MNIST)*

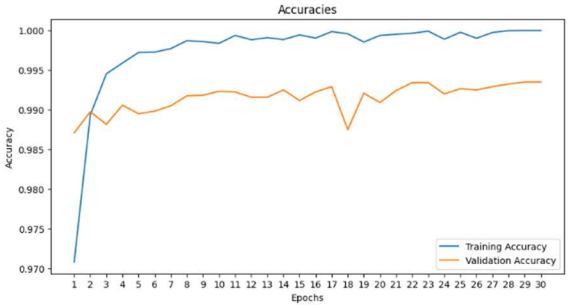


*Abb: 15: Trainings- und Validierungsgenauigkeit des CNN (MNIST)*

Auch in diesem Fall sehen die Validierungskurven sehr statisch aus, was darauf hindeutet, dass bereits in den ersten Epochen die maximale Effizienz erreicht ist. Der Test des Modells ergab dabei folgende Werte für die Metriken.

| Genauigkeit | Präzision | Recall | F1 |
|---|---|---|---|
| 0.994 ≈ 99,4% | 0.9940≈ 99,4% | 0.994≈ 99,4% | 0.9939 ≈ 99,4% |

Die Klassifizierung ergibt eine nahezu perfekte Vorhersage. Nun wird das CNN auf den CIFAR10-Datensatz verwendet. Hierzu wurde erneut eine Hyperparameteroptimierung mit Grid-Suche angewendet und das Netzwerk aufgebaut. Dabei ergaben sich folgende Verlust- und Genauigkeitskurven.

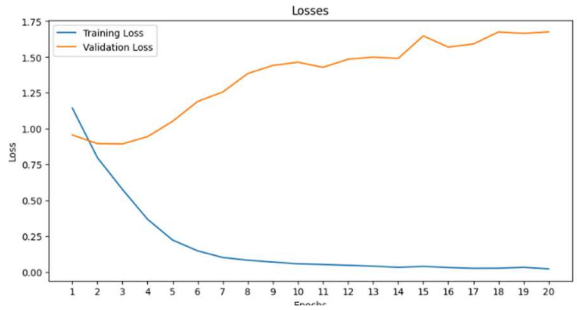


*Abb: 16: Trainings- und Validierungsloss des CNN (CIFAR10)*

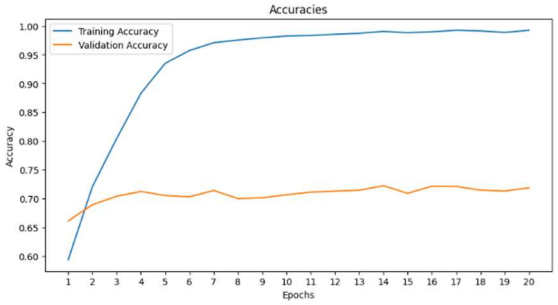


*Abb: 17: Trainings- und Validierungsgenauigkeit des CNN (CIFAR10)*

Wobei sich durch das Testen des Modells folgende Metriken ergaben.

| Genauigkeit | Präzision | Recall | F1 |
|---|---|---|---|
| 0.7175 ≈ 71,7% | 0.7180≈ 71,8% | 0.7175≈ 71,7% | 0.7157 ≈ 71,5% |

Hierbei fällt auf, dass bereits ab Epoche 3 Overfitting einzutreten scheint. Das Training mit demselben Modell auf 3 Epochen liefert wie erwartet ähnliche Metriken.

Es ergibt sich wieder eine etwas bessere Klassifizierung gegenüber den Convolutional Auto Encoder mit Klassifizierer. Schließlich soll ein tieferes Neuronales Netz die Klassifizierungsaufgabe übernehmen, um den CIFAR10-Datensatz genauer zu klassifizieren.

## 6.7 Klassifizierung mit DeepCNN

Nun wird der CIFAR10-Datensatz mithilfe des DeepCNN aufgebaut. Auch hier wurden ReLU-Aktivierungen für die versteckten Schichten gewählt. Als Verlustfunktion wurde die Cross-Entropy Loss gewählt und der Adam-Optimierer als Optimierer. Dabei ergaben sich folgende Kurven.

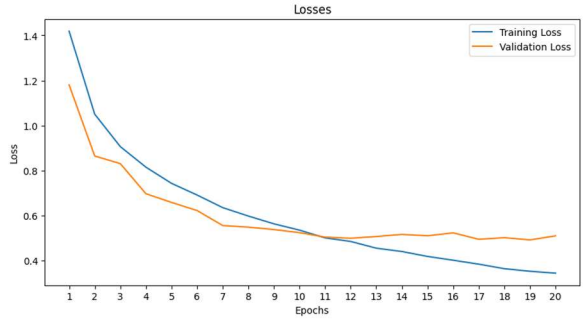


*Abb: 18: Trainings- und Validierungsloss des DeepCNN*

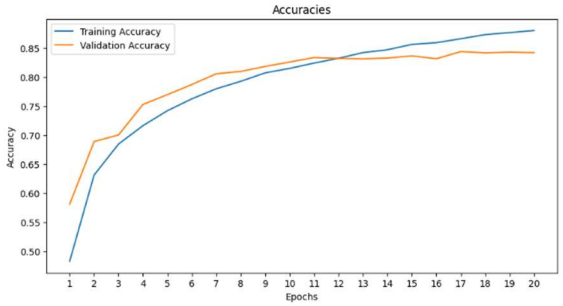


*Abb: 19: Trainings- und Validierungsgenaugikeit des DeepCNN*

Diese Kurven sehen sehr gut aus und nähern sich einer besseren Genauigkeit. 20 Epochen scheinen dabei eine gute Wahl zu sein, da der Validierungsverlust anscheinend langsam wieder ansteigt, was für ein Overfitting spricht. Der Test des Modells auf die Metriken ergab folgende Werte.

| Genauigkeit | Präzision | Recall | F1 |
|---|---|---|---|
| 0.8399 ≈ 84% | 0.8410 ≈ 84% | 0.8399 ≈ 84% | 0.8395 ≈ 84% |

Dies sind die bisher besten Werte für die Klassifizierung des CIFAR10-Datensatzes.

## 6.8 Bildgenerierung mit GAN

Jetzt sollen mithilfe eines GAN's Bilder generiert werden, die den trainierten Datensätzen gleichen. Dabei wurde das GAN mit vielversprechenden Parametern erstellt und zunächst auf den MNIST-Datensatz getestet. Dabei besteht der Generator aus Sequenziellen Schichten vollständig gebundener Schichten und Leaky-ReLU Akivierungsfunktionen. Die Aktivierungsfunktion der Ausgabeschicht ist dabei die Tanh-Funktion. Der Diskriminator besteht aus vollständig gebundenen Schichten gefolgt von Leaky-ReLU-Aktivierungsfunktionen und Dropout-Schichten, um Overfitting zu vermeiden. Als Atkivierungsfunktion der Ausgabeschicht wurde die Sigmoid-Funktion verwendet. Mit dem BCE-Loss als Verlustfunktion und dem Adam-Optimierer für Generator und Diskriminator konnte nun das Modell trainiert werden. Nach dem Testen ist aufgefallen, dass mit hinreichender Optimierung zwar gute Bilder des MNIST-Datensatzes generiert, werden könnten, der CIFAR10-Datensatz allerdings viel zu kompliziert für ein einfaches GAN ist. Deshalb wurde das GAN erweitert, und die Bildgenerierung mit einem Deep Convolutional GAN angegangen.

## 6.9 Bildgenerierung mit DCGAN

Um bessere Ergebnisse der Bildgenerierung insbesondere für den CIFAR10-Datensatz zu erhalten, wird nun versucht die Bildgenerierung mit einem DCGAN zu verwirklichen. Dabei besteht der Generator aus Sequenziell angeordneten Schichten von transponierten Faltungsschichten, Batch-Normalisierung, und ReLU-Aktivierungsfunktionen. Die Aktivierungsfunktion der Ausgabeschicht ist dabei die Tanh-Funktion. Der Diskriminator besteht aus einer Anordnung von Faltungsschichten Leaky-ReLU-Aktivierungsfunktionen und einer Sigmoid-Aktivierung in der Ausgabeschicht. Während des Trainings wurden auch hier die Verluste des Generators und Diskriminator aufgezeichnet, welche folgenden Verlauf annahmen.

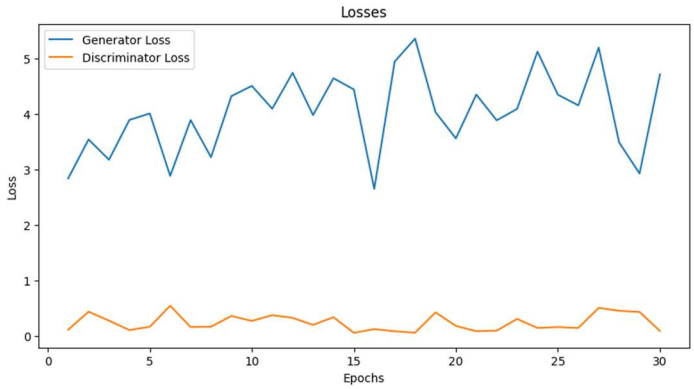


*Abb: 20: Generator und Diskriminatorloss des DCGAN's (MNIST)*

Es wurden wieder vom Generator generierte Bilder über die Epochen angezeigt, wobei sich nach der letzten Epoche folgende Beispielbilder ergaben.

*Abb: 21: Beispielbilder des DCGAN's (MNIST)*

Diese Bilder sind eine sehr genaue Repräsentation der Bilder des MNIST-Datensatzes und tatsächlich schwer von den Originalbilder zu unterscheiden.

Nun wurde das DCGAN auf den CIFAR10 Datensatz trainiert, wobei sich folgende Verluste für Generator und Diskriminator ergaben.

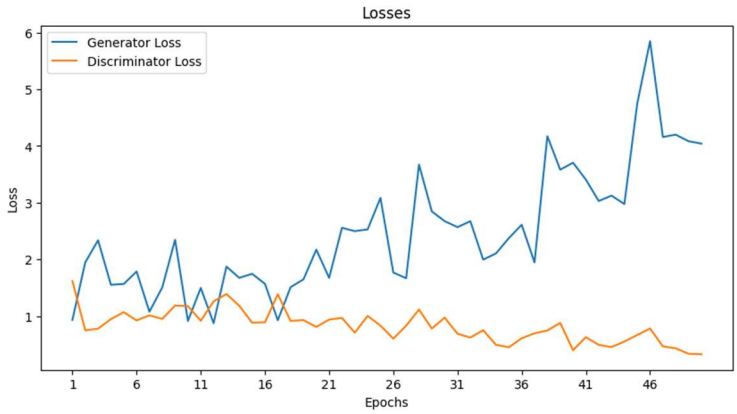


*Abb: 22: Generator- und Diskriminatorloss des DCGAN's (CIFAR10)*

Einige generierte Beispielbilder sind in der unteren Abbildung abgebildet.

*Abb: 23: Beispielbilder des DCGAN (CIFAR10)*

Obwohl nicht alle Bilder aus menschlicher Sicht eindeutig klassifizierbar sind, kann man doch in allen Bildern etwas erkennen bzw. hineininterpretieren. Das DCGAN erfüllt also eine gute Arbeit bei der Generierung von Bildern sowohl für den MNIST als auch für den CIFAR10-Datensatz.Obwohl auch das gewöhnliche GAN die Bilder des MNIST-Datensatzes gut generieren kann, wird im weiteren Verlauf das DCGAN verwendet, da dieses noch bessere Ergebnisse liefert. Auf Basis dieser generierten Bilder sollen nun die zuvor entwickelten Klassifikator-Modelle verwendet werden, um die generierten Bilder zu klassifizieren.

## 6.10 Klassifizierung der GAN-generierten Bilder

Jetzt sollen alle zuvor entwickelten Klassifikatoren verwendet werden, um generierte Beispielbilder des GAN's zu klassifizieren. Da diese generierten Bilder keine Labels haben und somit keinen vollständigen Datensatz darstellen, können sie nicht auf gewöhnliche Weise getestet werden. Daher werde ich die Klassifikation aus menschlicher, also Subjektiver Sicht analysieren, und bewerten, ob diese sinnvoll ist. Dabei hat sich ergeben, dass die Ergebnisse der Klassifizierung entsprechend der Metriken der Klassifizierung ausfallen. Bei Modellen, die bessere Leistung auf den Testdatensatz erbracht haben, konnte ich mehr bei ihren Entscheidungen zur Klassifizierung zustimmen.

# 6. Fazit

Die vorliegende Projektarbeit untersuchte mehrere datengesteuerte Klassifizierungsmethoden auf den Benchmark-Datensätzen MNIST und CIFAR-10. Die angewendeten Methoden umfassten Multilayer Perceptron (MLP), Autoencoder mit Klassifizierung, Convolutional Autoencoder mit Klassifizierung, sowie Convolutional Neural Network (CNN) und Deep CNN. Die Evaluation erfolgte durch umfassende Leistungstests auf den genannten Datensätzen.

Die Ergebnisse der Klassifizierung zeigten eine fortschreitende Verbesserung der Leistungsfähigkeit der Modelle. Das MLP bildete den Ausgangspunkt und diente als Referenzpunkt für die darauffolgenden Methoden. Der Autoencoder mit anschließender Klassifizierung konnte eine verbesserte Klassifikationsgenauigkeit bieten, während der Convolutional Autoencoder mit Klassifizierung zusätzlich die räumlichen Merkmale der Bilder besser erfasste.

Der Einsatz von Convolutional Neural Networks (CNN) steigerte die Leistung weiter, insbesondere in Bezug auf Bilddaten wie CIFAR-10. Schließlich erreichte das Deep CNN herausragende Ergebnisse, indem es tiefe Netzwerkarchitekturen nutzte, um komplexe Merkmale zu extrahieren und hochdimensionale Daten effektiv zu verarbeiten.

Insgesamt verdeutlichen die Ergebnisse den evolutionären Fortschritt von traditionellen MLPs zu fortgeschrittenen CNNs in Bezug auf die Klassifikationsgenauigkeit. Die Erkenntnisse dieser Arbeit bieten wertvolle Einblicke in die Anwendung unterschiedlicher Modelle auf reale Bildklassifikationsaufgaben und ermöglichen eine fundierte Bewertung der Leistungsfähigkeit datengesteuerter Methoden in verschiedenen Szenarien. Eine Zusammenfassung der Leistungen der Modelle auf die verschiedenen Metriken ist in der unteren Abbildung abgebildet.

MNIST-Datensatz

| | Genaugikeit | Präzision | Recall | F1-Score |
|---|---|---|---|---|
| MLP | 0.9787 | 0.9787 | 0.9787 | 0.9786 |
| AE+Klassifizierer | 0.982 | 0.982 | 0.9829 | 0.9819 |
| ConvAE+Klassifizierer | 0.9878 | 0.9878 | 0.9878 | 0.9878 |
| CNN | 0.994 | 0.994 | 0.994 | 0.9939 |
| DeepCNN | 0.9933 | 0.9933 | 0.9933 | 0.9932 |

CIFAR10-Datensatz

| | Genaugikeit | Präzision | Recall | F1-Score |
|---|---|---|---|---|
| MLP | 0.515 | 0.5177 | 0.515 | 0.5152 |
| AE+Klassifizierer | 0.4892 | 0.4952 | 0.4892 | 0.49 |
| ConvAE+Klassifizierer | 0.6336 | 0.65 | 0.6346 | 0.6376 |
| CNN | 0.7175 | 0.718 | 0.7175 | 0.7157 |
| DeepCNN | 0.8399 | 0.841 | 0.8399 | 0.8395 |

Für den MNIST-Datensatz hat das normale Convolutional Neural Network am besten performt, wohingegen für den CIFAR10-Datensatz das Tiefe Convolutional Neural Network am besten performt hat.