ingenieur
wissenschaften
htw saar

Hochschule für
Technik und Wirtschaft
des Saarlandes
University of
Applied Sciences

Bachelor-Thesis

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)
an der Hochschule für Technik und Wirtschaft des Saarlandes
im Studiengang Praktische Informatik
der Fakultät für Ingenieurwissenschaften

Effiziente Generierung von Trainingsdaten in der Bildklassifikation

vorgelegt von
Jan Rauber

betreut und begutachtet von
Prof. Dr.-Ing. Klaus Berberich

Saarbrücken, 08. 06. 2025

Selbständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit (bei einer Gruppenarbeit: den entsprechend gekennzeichneten Anteil der Arbeit) selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich erkläre hiermit weiterhin, dass die vorgelegte Arbeit zuvor weder von mir noch von einer anderen Person an dieser oder einer anderen Hochschule eingereicht wurde.

Darüber hinaus ist mir bekannt, dass die Unrichtigkeit dieser Erklärung eine Benotung der Arbeit mit der Note „nicht ausreichend“ zur Folge hat und einen Ausschluss von der Erbringung weiterer Prüfungsleistungen zur Folge haben kann.

Saarbrücken, 08. 06. 2025

_____

Jan Rauber

Zusammenfassung

Kurze Zusammenfassung des Inhaltes in deutscher Sprache, der Umfang beträgt zwischen einer halben und einer ganzen DIN A4-Seite.

Orientieren Sie sich bei der Aufteilung bzw. dem Inhalt Ihrer Zusammenfassung an Kent Becks Artikel: `http://plg.uwaterloo.ca/~migod/research/beckOOPSLA.html`.

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth [9]

Danksagung

Hier können Sie Personen danken, die zum Erfolg der Arbeit beigetragen haben, beispielsweise Ihren Betreuern in der Firma, Ihren Professoren/Dozenten an der htw saar, Freunden, Familie usw.

Inhaltsverzeichnis

1 Technische Grundlagen

Dieses Kapitel beschreibt die technischen Werkzeuge und Bibliotheken, die für die Implementierung der Active-Learning-Experimente verwendet wurden.

1.1 Überblick über die Implementierung

Die Experimente wurden in drei `Jupyter-Notebooks` implementiert, die mit Unterstützung von `Claude Opus 4 (Anthropic)` entwickelt wurden. Die Notebooks evaluieren systematisch verschiedene Kombinationen von Klassifikatoren und Query-Strategien auf drei Datensätzen: `MNIST`, `Fashion-MNIST` und einem unbalancierten Dachmaterialdatensatz.

Der Quellcode ist verfügbar unter: `https://github.com/jan1a234/bachelorarbeit.git`

1.2 Datenverarbeitung und -analyse

In diesem Abschnitt werden die für die Evaluation verwendeten, fachspezifischen Python-Bibliotheken für maschinelles Lernen erörtert, wobei grundlegende Python-Kenntnisse vorausgesetzt werden. Der Fokus liegt zunächst auf den Werkzeugen zur Datenverarbeitung und -analyse, da die Aufbereitung der Daten eine elementare Voraussetzung für das Training von Klassifikationsmodellen darstellt.

1.2.1 NumPy

Bei der Entdeckung von Gravitationswellen oder bei dem ersten Bild einer astronomischen Singularität hat `Numpy` eine entscheidende Rolle gespielt. [7] [1] `Numpy` ist daher sehr leistungsfähig bei wissenschaftlichen Durchbrüchen. `Numpy` bietet die Möglichkeit, effizient höhere Mathematik auf riesige Datensätze anzuwenden. Anstatt mit einzelnen Zahlen zu hantieren, arbeitet `Numpy` mit multidimensionalen `Arrays`, vergleichbar mit einem Würfel aus Zahlen, was die Berechnungen auf diesen Datensätzen erheblich beschleunigt. [8]

Listing 1.1: Praktisches Beispiel für die im Text erwähnte effiziente Array-Verarbeitung mit NumPy: Transformation von Trainingsdaten in float32-Arrays für beschleunigte Berechnungen - Aus: Dachmaterialien_F1_Score.ipynb

```
# X_train_processed ist ein Numpy-Array
X_train_processed = preprocessor.fit_transform(X_train).
    astype(np.float32)
```

Diese multidimensionalen `Arrays` erlauben schnellere Operationen als normale `Python-Listen`, weil sie im Speicher viel effizienter organisiert sind und viele Operationen im schnellen C-Code implementiert sind. Pures `Python` wäre zu langsam. Diese `Arrays` sind auch

[1] Für die offizielle Dokumentation siehe NumPy Developers (2024): `https://numpy.org/doc/stable/`. Die hier verwendeten Implementierungen basieren auf NumPy Version 1.x, dem fundamentalen Paket für wissenschaftliches Rechnen in Python.

Grundlage für weitere Python-Bibliotheken wie `Pandas` zur Datenanalyse, `Matplotlib` zum Visualisieren und `Scikit-learn` fürs maschinelle Lernen. All diese Bibliotheken bauen auf `NumPy` auf. [8]

Listing 1.2: Demonstration wie andere Bibliotheken (hier Scikit-learn) auf NumPy-Arrays als Grundlage aufbauen, wie im Text beschrieben - Aus: Dachmaterialien_F1_Score.ipynb

```
# Scikit-learn nutzt NumPy-Arrays für die Metrik-Berechnung
final_f1 = f1_score(y_test, y_pred, average='macro')
```

`NumPy` bietet fortschrittliche Techniken wie die Vektorisierung. Das bedeutet, dass sich Rechenoperationen wie beispielsweise „–4“ auf alle Zahlen in meinem Datengitter anwenden lassen, ohne selbst eine Schleife programmieren zu müssen. [8]

Listing 1.3: Konkrete Umsetzung der im Text erläuterten Vektorisierung: Entropie-Berechnung ohne explizite for-Schleife durch NumPy-Operationen - Aus: Alle Active-Learning Notebooks

```
# Vektorisierte Berechnung der Entropie ohne for-Schleife
entropies = -np.sum(probs * np.log(probs), axis=1)
```

Außerdem gibt es noch `Broadcasting`. Das heißt, bei Arrays, die nicht dieselbe Form oder Größe haben, versucht `NumPy`, diese auf eine intelligente Art anzugleichen, indem es die kleinere Form auf die größere broadcastet. `Broadcasting` bedeutet in diesem Kontext das die kleinere Form auf die größere Form erweitert wird. Dadurch müssen diese unterschiedlichen `Arrays` nicht von Hand angepasst werden. Das macht den Code kürzer und schneller lesbar. [8]

Listing 1.4: Praktische Anwendung des im Text beschriebenen Broadcasting-Konzepts: Automatische Anpassung unterschiedlicher Array-Formen bei der Softmax-Berechnung - Aus: Alle Active-Learning Notebooks

```
# Broadcasting: np.max(...) (shape: (n,1)) wird von decision (shape:
    (n,m)) subtrahiert
exp_decision = np.exp(decision - np.max(decision, axis=1,
    keepdims=True))
probs = exp_decision / np.sum(exp_decision, axis=1,
    keepdims=True)
```

`NumPy` wird zudem zu einem universellen Übersetzer. Es kann als Brücke dienen für Daten, die auf der `CPU` liegen, und Daten, die auf leistungsfähigen Grafikkarten verarbeitet werden sollen, beispielsweise für maschinelles Lernen. Oder als Brücke auf verteilten Systemen über mehrere Rechner hinweg. Diese Fähigkeit, verschiedene Systeme zu verbinden, wird als Interoperabilität bezeichnet. [8]

Listing 1.5: Beispiel für die im Text genannte Interoperabilität: NumPy als Brücke zwischen CPU-Daten und GPU-fähigen PyTorch-Tensoren - Aus: MNIST/Fashion-MNIST CNN Notebooks

```
# Interoperabilität: Umwandlung eines NumPy-Arrays in einen
    PyTorch-Tensor für die GPU
dataset = TensorDataset(
torch.from_numpy(X_np).float(),
torch.from_numpy(y_np).long()
)
```

Die Nutzung von ‘torch‘ illustriert die Interoperabilität von NumPy, indem es CPU-Daten in GPU-fähige PyTorch-Tensoren für maschinelles Lernen überführt. [2]

1.2.2 Pandas

Pandas ist wichtig für die Datenanalyse in Python in Bezug auf strukturierte Daten. [3] Strukturierte Daten sind Daten, die in Tabellen mit Zeilen und Spalten vorliegen, ähnlich wie in Excel, nur kann Python hier mitbenutzt werden. Damit kann man Dataframes und deren Zeilen und Spalten mit eigenen Labels handhaben. [13]

Listing 1.6: Praktisches Beispiel für das im Text erwähnte Einlesen strukturierter Daten in Pandas DataFrames - Aus: Dachmaterialien_F1_Score.ipynb

```
df = pd.read_csv(filepath)
```

Oder man kann nur eine Series manipulieren, welche eine einzelne Zeile oder Spalte betrifft. [13]

Listing 1.7: Demonstration der im Text beschriebenen Series-Manipulation: Extraktion einzelner Spalten aus einem DataFrame - Aus: Dachmaterialien_F1_Score.ipynb

```
y = df[target_col].copy()
```

Diese Series hat jedoch einen Index, um die einzelnen Datenpunkte zu verbinden. Daten säubern, filtern, sortieren lassen sich mit Pandas bewerkstelligen. Beispielsweise lassen sich Daten bereinigen und Duplikate entfernen. [13]

Listing 1.8: Umsetzung der im Text genannten Datenbereinigung: Filterung des DataFrames nach gültigen Klassen - Aus: Dachmaterialien_F1_Score.ipynb

```
df = df[df[target_col].isin(valid_classes)].copy()
```

Oder die Behandlung fehlender Werte in dem unausgeglichenen Dachmaterialiendatensatz lässt sich bewerkstelligen. [13]

Listing 1.9: Konkrete Anwendung der im Text erwähnten Behandlung fehlender Werte im Dachmaterialdatensatz - Aus: Dachmaterialien_F1_Score.ipynb

```
('imputer', SimpleImputer(strategy='median'))
```

Mit groupby und apply lassen sich komplexe Funktionen auf Daten anwenden. [13]

Listing 1.10: Praktisches Beispiel für die im Text beschriebene groupby-Funktionalität zur komplexen Datenaggregation - Aus: Alle Notebooks mit Evaluation

```
summary = results_df.groupby(['classifier', 'strategy', '
    budget_pct'])['f1_score'].agg(['mean', 'std'])
```

Pandas arbeitet intern sehr schnell, was auf Vektorisierung beruht. Operationen auf ganze Datenblöcke werden auf einmal gemacht und nicht in einer Schleife. Zugrunde liegen C-Bibliotheken. Im Rahmen der Evaluation wird diese Bibliothek zur Datenaufbereitung für das maschinelle Lernen verwendet. Die Bibliothek findet jedoch bei jeder Art von wissenschaftlicher Arbeit zur Datenmanipulation Verwendung. [13]

[2] https://numpy.org

[3] Für die offizielle Dokumentation siehe The pandas development team (2024): https://pandas.pydata.org/docs/. Die hier verwendeten Funktionalitäten basieren auf pandas Version 2.x.

1.3 Maschinelles Lernen

Im Vordergrund der Evaluation stehen Datenvorbereitung und faire Bewertung. Man kann nicht einfach die Rohdaten nehmen. Der unbalancierte Dachmaterialdatensatz enthält numerische Features wie die Hausgröße in Quadratmetern und daneben Materialbeschreibung wie Ziegel oder Beton. Der Algorithmus kann mit diesen unterschiedlichen Formaten wenig anfangen. Es besteht die Gefahr, dass große Zahlenwerte kleine Zahlenwerte, die dennoch wichtig sind, überlagern. Deshalb müssen die Daten vergleichbar sein. Realisiert wird das beispielsweise mit dem `StandardScaler`, der alles auf einen gemeinsamen Maßstab bringt. Das bewirkt, dass keine Zahl nur wegen ihrer Größe dominiert. Der korrekte Begriff lautet `StandardScaler`, welcher technisch eine z-Transformation durchführt, bei der jedes Feature auf einen Mittelwert von 0 und eine Standardabweichung von 1 normalisiert wird ($z = \frac{x-\mu}{\sigma}$). Diese Standardisierung stellt sicher, dass alle numerischen Features denselben Einfluss auf das Modell haben, unabhängig von ihrer ursprünglichen Einheit oder Größenordnung. [3, 16]

Listing 1.11: Implementierung der im Text erläuterten Datenvorbereitung: Pipeline mit SimpleImputer und StandardScaler zur Vermeidung dominierender Zahlenwerte - Aus: Dachmaterialien_F1_Score.ipynb

```
numeric_transformer = Pipeline(steps=[
('imputer', SimpleImputer(strategy='median')),
('scaler', StandardScaler())
])
```

Für kategoriale Daten wie Ziegel oder Beton gibt es verschiedene Ansätze. Der `Labelencoder` macht aus Text einfach Zahlen. Dem String „ja“ wird einfach 1 zugeordnet, dem String „nein“ beispielsweise 2. Bei mehreren kategorialen Merkmalen wie Ziegel, Beton, Kupfer etc. ist der `One-Hot-Encoder` besser, weil dieser für jedes Merkmal eine eigene „Ja“-„Nein“-Spalte erstellt, wodurch verhindert wird, dass das Modell eine künstliche Reihenfolge annimmt. Es könnte sein, dass das Modell denkt, Beton sei mehr wert als Ziegel, nur weil diesem String eine höhere Zahl zugeordnet wurde. [3, 16]

Listing 1.12: Umsetzung der im Text beschriebenen One-Hot-Encoding-Strategie zur Vermeidung künstlicher Ordnung bei kategorialen Daten - Aus: Dachmaterialien_F1_Score.ipynb

```
categorical_transformer = Pipeline(steps=[
('imputer', SimpleImputer(strategy='constant', fill_value='
    missing')),
('onehot', OneHotEncoder(drop='first', handle_unknown='
    ignore'))
])
```

Durch diese Vorgehensweise werden die Daten so aufbereitet, dass die Daten optimal genutzt werden, ohne falsche Schlüsse zu ziehen. Diese Vorgehensweise kann bei sehr vielen zu verarbeitenden Merkmalen dennoch unübersichtlich werden, weshalb `Column-Transformer` und `Pipeline` zum Einsatz kommen. Der `Column-Transformer` ist mit einem Regisseur vergleichbar, der festlegt, welcher Schritt auf welchen Spalten der Daten ausgeführt wird. [3, 16]

Listing 1.13: Praktische Anwendung des im Text als "Regisseur"beschriebenen ColumnTransformers zur koordinierten Datenverarbeitung - Aus: Dachmaterialien_F1_Score.ipynb

```
preprocessor = ColumnTransformer(
transformers=[
```

```
        ('num', numeric_transformer, ['area']),
        ('cat', categorical_transformer, ['area_type', 'Shape', '
            ezg'])
        ])
```

Die `Pipeline` packt alles zusammen, was aus `One-Hot-Encoder` für die Textdaten und dem `Standard-Encoder` für die Zahlenwerte resultiert. Diese Aufteilung macht den Prozess wiederholbar und verhindert `Dataleakage`. `Dataleakage` passiert, wenn Infos aus den Testdaten, die zum Überprüfen da sind, für das Training genommen werden, was das Ergebnis verfälscht. `Pipelines` sorgen dafür, dass das nicht passiert, durch saubere Abläufe, wodurch bereits Ordnung und Fairness bei der Vorbereitung gewährleistet werden. [3, 16]

1.3.1 Modelltraining mit Scikit-learn

Im Anschluss an die Datenvorbereitung wird geprüft, um die Leistungsfähigkeit des Modells zu bewerten, durch `Train-Test-Split`. Man teilt hier die Daten zum Lernen, also fürs Training, und einen unabhängigen Teil zum Testen. Es gibt `StratifiedShuffleSplit`, was besonders wichtig bei unausgeglichenen Datensätzen ist, [4] was bedeutet, dass einige Klassen viel seltener vorkommen als andere. Beispielsweise sind die interessanten Fälle wie bei Betrugserkennung oder seltenen Krankheiten oft sehr selten. `StratifiedShuffleSplit` sorgt dafür, dass das Verhältnis der Klassen, also wenige Betrugsfälle und viele Normale im Trainings- und Testset, gleich bleibt wie im Original. Sonst könnte es passieren, dass im Testset zufällig keine seltenen Fälle landen, wodurch die Bewertung sinnlos wäre. [3, 16]

Listing 1.14: Konkrete Implementierung des im Text erläuterten StratifiedShuffleSplit zur Erhaltung der Klassenverteilung bei unbalancierten Datensätzen - Aus: Dachmaterialien_F1_Score.ipynb

```
        # Robuster Train/Test Split
        # Verwende StratifiedShuffleSplit für bessere Kontrolle
        splitter = StratifiedShuffleSplit(n_splits=1, test_size
            =0.2, random_state=SEED)
        train_idx, test_idx = next(splitter.split(X, y_encoded))

        X_train = X.iloc[train_idx]
        y_train = y_encoded[train_idx]
```

Anschließend werden verschiedene Machine-Learning-Modelle evaluiert. Dazu gehören Klassifikatoren aus der `Scikit-learn` Bibliothek wie `Naïve Bayes (GaussianNB)`, `Zufallswald (RandomForestClassifier)`,`Logistische Regression (LogisticRegression)` und `Support Vector Machines (SVC)`. Zusätzlich wird ein benutzerdefiniertes Neuronales Netz berücksichtigt. Diese breite Auswahl an Modellen ist notwendig, da es kein einzelnes perfektes Klassifikationsmodell für jeden Datensatz gibt und der Vergleich verschiedener Ansätze wichtig ist, um die beste Leistung zu erzielen. [3, 16]

Listing 1.15: Definition der im Text genannten verschiedenen Klassifikatoren für die vergleichende Evaluation - Aus: Alle Active-Learning Notebooks

```
        # Klassifikatoren und Strategien definieren
        classifiers = ['Neural Network', 'Naive Bayes', 'Random
            Forest', 'Logistic Regression', 'SVM']
```

[4] Für die offizielle Dokumentation und API-Referenz siehe Scikit-learn Developers (2024): `https://scikit-learn.org/stable/`. Die hier verwendeten Implementierungen basieren auf Scikit-learn Version 1.x.

```
        # ... Schleife zur Durchführung der Experimente ...
        for classifier_name in classifiers:
        # ...
```

Random Forest ist ein Ensemble-Modell, das oft sehr robust ist. Eine Support Vector Machine nutzt Kernels, um auch komplexere Muster zu finden, während Gaussian Naïve Bayes ein schneller probabilistischer Ansatz ist. Es werden verschiedene Werkzeuge getestet, um zu prüfen, welches Werkzeug auf den konkreten Datensätzen MNIST, Fashion-MNIST und dem unausgeglichenen Dachmaterialdatensatz am besten funktioniert. [3, 16]

1.3.2 Modellevaluierung mit Scikit-learn

Die verwendeten Evaluierungsmetriken sind dokumentiert unter https://scikit-learn.org/stable/modules/model_evaluation.html.Zum Messen der Modellgenauigkeit kommt der F1-Score zum Einsatz, weil die Accuracy bei unausgeglichenen Datensätzen oft trügerisch ist. Bei ausgeglichenen Datensätzen wie MNIST oder Fashion-MNIST hingegen ist die Accuracy als Maß zum Messen der Modellgenauigkeit ausreichend. Ein Modell, das nur die häufigsten Klassen vorhersagt, also keine Krankheiten oder Betrugsfälle, kann eine sehr hohe Accuracy haben, ist aber in den seltenen Fällen nutzlos. Deshalb wurde speziell der Macro-F1-Score auf dem unausgeglichenen Dachmaterialdatensatz angewandt. Dieser F1-Score ermittelt Präzision und Trefferquote über alle Klassen hinweg, wobei jede Klasse denselben Wert hat. [3, 16]

Listing 1.16: Praktische Anwendung des im Text beschriebenen Macro-F1-Scores für unausgewogene Datensätze - Aus: Dachmaterialien_F1_Score.ipynb

```
# Evaluation mit F1-Score
y_pred = model.predict(X_test)

# Verwende macro average F1-Score für unausgewogene Datensätze
final_f1 = f1_score(y_test, y_pred, average='macro')
final_acc = accuracy_score(y_test, y_pred)
```

Das ergibt ein umfangreicheres Bild, gerade wenn die Erkennung seltener Klassen wichtig ist. Der Classification-Report liefert alle Detailwerte pro Klasse, wodurch die Bewertung deutlich differenzierter wird. [3, 16]

Listing 1.17: Import der im Text erwähnten Metriken zur differenzierten Modellbewertung - Aus: Alle Notebooks

```
# Import der notwendigen Metriken aus Scikit-learn
from sklearn.metrics import accuracy_score, f1_score,
    classification_report
```

5

Das Verständnis dieser Schritte im maschinellen Lernen gibt die Möglichkeit, die Ergebnisse der Evaluation kritisch zu hinterfragen. [3, 16]

[5] https://scikit-learn.org

1.4 Pytorch

`PyTorch` ist eine sehr flexible Open-Source-Bibliothek in `Python`. [6] Dabei ist das „`define by run`“-Konzept ein Schlüsselelement in `Pytorch`. `Define by run` bedeutet, der Bauplan entsteht während der Laufzeit, wodurch das Programm viel intuitiver wird, gerade bei komplexeren Netzen. Es nutzt Grafikkartenbeschleunigung, falls vorhanden. [10]

1.4.1 Architektur neuronaler Netze

Der Bau eines neuronalen Netzes in `PyTorch` beginnt mit dem Import des zentralen Werkzeugkastens `torch.nn`, der wie ein Legokasten alle notwendigen Bausteine (z.B. `nn.Linear`, `nn.Conv2d`) enthält. [10]

Listing 1.18: Import des im Text als "Legokasten"beschriebenen torch.nn Moduls mit allen Bausteinen für neuronale Netze - Aus: MNIST/Fashion-MNIST CNN Notebooks

```
import torch.nn as nn
```

Die Bibliothek bietet einen umfangreichen und modularen Baukasten für neuronale Netze. Hier sind alle Bausteine für ein künstliches Gehirn vorhanden. Hier gibt es die grundlegenden Verbindungsstücke wie `nn.Linear` oder `nn.Conv2d`, was gut darin ist, Bilder und andere räumliche Daten zu erkennen. `nn.Conv2d` ist ein automatischer Merkmalsdetektor, ohne dass die Bildverarbeitung in neuronalen Netzen nicht funktionieren kann. Diese Schicht lässt einen `Kernel` über die Pixel gleiten, um Ecken, Kanten und Texturen eines Bildes zu identifizieren. [10]

Listing 1.19: Beispiel für die im Text genannten grundlegenden Verbindungsstücke: Linear Layer mit 256 Neuronen - Aus: Dachmaterialien_F1_Score.ipynb (TabularNN)

```
# Eine lineare Schicht aus dem Modell für tabellarische Daten:
nn.Linear(input_dim, 256),
```

Dann gibt es die Schalter mit Aktivierungsfunktionen wie `nn.ReLU` oder `nn.Sigmoid`, die entscheiden, ob ein Signal weitergeleitet wird. [10]

Listing 1.20: Konkrete Anwendung der im Text beschriebenen Aktivierungsfunktion als SSchalter"für Signalweiterleitung - Aus: Alle Neural Network Notebooks

```
# Die ReLU-Aktivierungsfunktion im Einsatz:
nn.ReLU(inplace=True),
```

Zudem gibt es Bausteine, die messen, wie gut das Netz bereits performt, wie Normalisierungsschichten und Verlustfunktionen, beispielsweise `nn.CrossEntropyLoss`, die beschreiben, wie weit die Vorhersagen noch vom Ziel entfernt sind. [10]

Listing 1.21: Implementierung der im Text erwähnten Normalisierungsschicht zur Stabilisierung des Trainings - Aus: Dachmaterialien_F1_Score.ipynb (TabularNN)

```
# Eine Normalisierungsschicht zur Stabilisierung des Trainings:
nn.BatchNorm1d(256),
```

Listing 1.22: Umsetzung der im Text genannten Verlustfunktion zur Messung der Vorhersageabweichung - Aus: Alle Neural Network Notebooks

[6] Für die offizielle Dokumentation und weitere Tutorials siehe PyTorch Foundation (2024): `https://pytorch.org`.

```
# Definition der Verlustfunktion in der Trainingsmethode:
loss_fn = nn.CrossEntropyLoss()
```

Die Bibliothek ist modular aufgebaut, weshalb jedes vorstellbare Modell gebaut werden kann. Die Bausteine lassen sich also frei kombinieren, wie beim Prototyping. [10]

Listing 1.23: Demonstration der im Text beschriebenen modularen Bauweise: Freie Kombination verschiedener Bausteine im Sequential-Container - Aus: Dachmaterialien_F1_Score.ipynb (TabularNN)

```
# Die Modularität zeigt sich im nn.Sequential-Container,
# der verschiedene Bausteine zu einem Feature-Extraktor verbindet:
self.features = nn.Sequential(
# Layer 1
nn.Linear(input_dim, 256),
nn.BatchNorm1d(256),
nn.ReLU(inplace=True),
nn.Dropout(0.4),

# Layer 2
nn.Linear(256, 128),
nn.BatchNorm1d(128),
nn.ReLU(inplace=True),
nn.Dropout(0.3),

# Layer 3
nn.Linear(128, 64),
nn.BatchNorm1d(64),
nn.ReLU(inplace=True),
nn.Dropout(0.2),
)
```

Die Forschenden haben die Möglichkeit, zur Laufzeit des Programms Designentscheidungen zu treffen, wie das Netz aufgebaut sein soll. Bei älteren Frameworks musste der Plan vor der Laufzeit des Programms fertig sein. [10]

1.4.2 Optimierung mit Adam

Damit das neuronale Netz lernt, kommen diverse Algorithmen zum Einsatz. Im verwendeten Programm für die Evaluierung dieser Arbeit wurde der `Adam-Optimierer` des **torch.optim-Moduls** verwendet. `Adam` steht für `Adaptive Moment Estimation`. Er fungiert als eine adaptive Steuerungsstrategie für das neuronale Netz. Wenn der Weg stetig und zuverlässig in die richtige Richtung führt, wird `Adam` mutiger und gibt dem Netz an, einen größeren Schritt zu machen. Dabei ist sich bildlich vorzustellen, dass das neuronale Netz vom Berg ins Tal der optimalen Lösung runter möchte. `Adam` passt dabei für jede kleine Stellschraube im Netz die Schrittgröße an. `Adam` schaut zurück und reflektiert die Steigung im Durchschnitt und wie stark sie geschwankt hat, um anzugeben, ob die Richtung verlässlich ist. Andernfalls gibt `Adam` an, wenn es hügelig ist, einen kleinen Schritt zu gehen. `Adam` kommt dadurch schneller ans Ziel als ältere Methoden, die eine feste Schrittgröße verwendeten, weil `Adam` sich dem Gelände anpasst. [10]

Exemplarisch wird die Initialisierung des Optimierers in der Trainingsfunktion des neuronalen Netzes gezeigt:

Listing 1.24: Praktische Implementierung des im Text erläuterten Adam-Optimierers mit adaptiver Schrittgrößenanpassung - Aus: Dachmaterialien_F1_Score.ipynb (TabularNN)

```
        def fit(self, X_np, y_np, epochs=10, lr=1e-3, batch_size
            =256, verbose=False):
        """
        Trainiert das TabularNN mit optimierten Hyperparametern.
        """
        self.train()
        # ... (weitere Initialisierungen)
        # Hier wird der AdamW-Optimierer instanziiert
        optimizer = optim.AdamW(self.parameters(), lr=lr,
            weight_decay=1e-4)
        scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer,
             T_max=epochs)
        loss_fn = nn.CrossEntropyLoss()
        # ... (Trainingsschleife)
```

1.4.3 Datenhandling mit DataLoader

Um das neuronale Netz mit Daten zu füttern, gibt es beispielsweise `torch.utils.data`. [10]

Listing 1.25: Import der im Text beschriebenen Datenhandling-Komponenten TensorDataset und DataLoader - Aus: Alle PyTorch Notebooks

```
        from torch.utils.data import TensorDataset, DataLoader
```

Es ist zu beachten, dass die Datenversorgung bei Deep Learning oft der Flaschenhals ist und die Datenversorgung deshalb einen eigenen Baustein innerhalb dieser Python-Bibliothek darstellt. Verwendet werden dabei das `TensorDataset` und der `DataLoader`. `TensorDataset` ist nur da, um die Eingabedaten mit den zugehörigen Zielwerten ordentlich zu verpacken. [10]

Listing 1.26: Konkrete Umsetzung des im Text erwähnten TensorDatasets zum Verpacken von Eingabedaten und Zielwerten - Aus: Alle PyTorch Notebooks

```
        # Create dataset
        try:
        dataset = TensorDataset(
        torch.from_numpy(X_np).float(),
        torch.from_numpy(y_np).long()
        )
        except Exception as e:
        logger.error(f"Fehler beim Erstellen des Datasets: {e}")
        raise
```

Eine zentrale Komponente für den Trainingsprozess ist der `DataLoader`. Er ist für die effiziente Bereitstellung der Daten verantwortlich, indem er den Datensatz in kleine Pakete, sogenannte `Batches`, aufteilt und diese an das neuronale Netz weiterleitet. Der `DataLoader` kann auch die Reihenfolge der Daten mischen, damit das Netz nicht die Reihenfolge auswendiglernt. Der `DataLoader` kann zudem die `Batches`, die übergeben werden müssen, auf mehreren Kernen vorbereiten. Das sorgt dafür, dass die Grafikkarte immer beschäftigt ist, um die Trainingszeit des neuronalen Netzes zu verkürzen. Das effiziente Datenhandling ist daher für die Reduzierung der Laufzeit des Programms am wichtigsten. [10]

Listing 1.27: Praktische Konfiguration des im Text erläuterten DataLoaders mit Batch-Aufteilung und Shuffle-Option zur effizienten GPU-Auslastung - Aus: Alle PyTorch Notebooks

```
        # DataLoader mit optimierten Settings
        loader = DataLoader(
        dataset,
        batch_size=batch_size,
        shuffle=True,
        num_workers=0,  # Immer 0 für Kompatibilität
        pin_memory=(self.device.type == 'cuda'),
        drop_last=False
        )
```

1.5 Statistische Auswertung

Mittels `Wilcoxon signed rank test` wird die Leistung von Machine-Learning-Modellen gemessen. [7] Beispielsweise Modell A und Modell B, die auf denselben Daten laufen. Man kann nicht einfach den Durchschnittsfehler vergleichen. Es braucht den `Wilcoxon-Test`, weil die Daten nicht normalverteilt vorliegen. Der `Wilcoxon-Test` umgeht die Gefahr durch Ausreißer, die besteht, wenn nur der Durchschnittsfehler genommen würde. Der `Wilcoxon` schaut sich nicht die genauen Fehlerwerte an, sondern nur die Rangfolge der Unterschiede. Für jeden Datensplit fragt man: Wer war besser und wie groß war der Unterschied im Vergleich zu den anderen Unterschieden? Mit `Python` und `SciPy` lässt sich der `Wilcoxon-Test` in einer Codezeile durchführen. [20]

Listing 1.28: Import des im Text beschriebenen Wilcoxon-Tests für nicht-normalverteilte Daten - Aus: Alle Notebooks mit statistischer Analyse

```
        from scipy.stats import wilcoxon
```

Listing 1.29: Konkrete Anwendung des im Text erläuterten Wilcoxon-Tests zur robusten Modellvergleichung trotz Ausreißern - Aus: Alle Notebooks mit statistischer Analyse

```
        statistic, p_value = wilcoxon(
        strategy_data, baseline_data,
        alternative='greater',
        zero_method='zsplit'
        )
```

1.6 Datenvisualisierung

Diese Bibliotheken werden zur Datenvisualisierung verwendet. Sie machen Zahlen anschaulich zu Bildern und Grafiken. [21]

1.6.1 Matplotlib

`Matplotlib`[8] ist ein Werkzeugkasten fürs Zeichnen. Durch diese Bibliothek erhält der Anwender die Kontrolle über jeden Pixel innerhalb der zu erstellenden Visualisierung.

[7] Für eine detaillierte Einführung in statistische Tests und den Wilcoxon-Test siehe Küchenhoff (2017): `https://www.stablab.stat.uni-muenchen.de/lehre/veranstaltungen/veranstaltungsinhalte/stat2_bwl_vl9_neu.pdf`, S. 359.

[8] `https://matplotlib.org`

matplotlib.pyplot bildet hierzu das Fundament. Es ist sehr mächtig, jedoch ist es nötig, jedes Detail von Hand einzustellen. [21]

1.6.2 Seaborn

Obwohl Matplotlib bereits sehr mächtig ist, kann es vorteilhaft sein, durch Seaborn[9] Kontrolle abzugeben. Seaborn baut auf Matplotlib auf und nutzt Matplotlib im Hintergrund. Seaborn bietet jedoch eine zugänglichere und spezialisiertere Oberfläche im Vergleich zu Matplotlib. Seaborn ist insbesondere für statistische Plots wie in diesem Active-Learning-Experiment der Fall heranzuziehen. [21]

Die Einrichtung von seaborn zusammen mit matplotlib erfolgt typischerweise am Anfang eines Skripts, wobei oft auch ein bestimmter Stil für die Grafiken festgelegt wird, um die Lesbarkeit und Ästhetik zu verbessern. [21]

Listing 1.30: Setup der im Text genannten Bibliotheken Matplotlib und Seaborn mit Fehlerbehandlung und Stil-Konfiguration - Aus: Alle Notebooks mit Visualisierung

```
import matplotlib
matplotlib.use('Agg')  # Für Server ohne GUI
import matplotlib.pyplot as plt
# Seaborn mit Fehlerbehandlung
try:
import seaborn as sns
# Prüfe ob der Style verfügbar ist
try:
plt.style.use('seaborn-v0_8-whitegrid')
except:
try:
plt.style.use('seaborn-whitegrid')
except:
plt.style.use('ggplot')
except ImportError:
print("Warnung: Seaborn nicht installiert. Verwende
    Standard-Matplotlib.")
sns = None
```

Durch Seaborn sind komplexe statistische Plots mit wenigen Codezeilen machbar. Zum Beispiel lassen sich mithilfe von Seaborn Boxplots einfach erstellen, um Verteilungen zu verstehen. Ebenfalls lassen sich hiermit sehr einfach Heatmaps erstellen, um Beziehungen zwischen Variablen mit einem Augenblick zu erkennen. [21]

Der folgende Ausschnitt zeigt, wie eine Heatmap mit seaborn erzeugt wird, um die Effektstärke (Cliff's Delta) über verschiedene Budgets und Strategien darzustellen. [21] Die Effektstärke gibt an, wie viel besser eine verwendete Methode ist, die hier verglichen wird. Cliffs Delta ist die hier verwendete Art, die Effektstärke zu messen. [18]

Listing 1.31: Praktisches Beispiel für die im Text beschriebene Heatmap-Erstellung mit Seaborn zur Visualisierung von Cliff's Delta Effektstärken - Aus: Dachmaterialien_F1_Score.ipynb

```
# Heatmap der Effektstärken
pivot_effect = stat_results.pivot_table(
values='cliffs_delta',
index=['classifier', 'strategy'],
columns='budget_pct',
fill_value=0.0
```

[9] https://seaborn.pydata.org

```
        )
        if sns is not None and not pivot_effect.empty:
        sns.heatmap(pivot_effect,
        annot=True,
        fmt='.3f',
        cmap='coolwarm',
        center=0,
        vmin=-1,
        vmax=1,
        cbar_kws={'label': "Cliff's Delta"},
        ax=ax2)
```

Zudem bringt `Seaborn` ansprechende Designs und Farbpaletten als Standardkonfiguration bereits mit. Es lassen sich daher schnell Einblicke gewinnen über beispielsweise Normalverteilung und Gruppierung der Daten. Eine `Konfusionsmatrix` zeigt außerdem, wo ein Machine-Learning-Modell Fehler macht, und lässt sich mit `Seaborn` leicht erstellen. Auch einfache Balkendiagramme können generiert werden, um wichtige Faktoren hervorzuheben. [21]

Hier wird ein gruppiertes Balkendiagramm erstellt, um die prozentuale Verbesserung des `F1-Scores` gegenüber dem Zufalls-Sampling zu visualisieren. [21]

Listing 1.32: Umsetzung des im Text erwähnten gruppierten Balkendiagramms zur Darstellung der prozentualen F1-Score-Verbesserung - Aus: Alle Notebooks mit Visualisierung

```
        # Gruppierter Barplot
        x = np.arange(len(BUDGET_PERCENTAGES))
        width = 0.15
        # ... (Schleife über Klassifikatoren und Strategien)
        offset = (i * len(strategies) + j - len(classifiers) * len(
            strategies) / 2) * width
        bars = ax.bar(x + offset, values, width,
        label=f'{classifier} - {strategy}',
        color=colors[color_idx])
```

`Seaborn` unterstützt Barrierefreiheit, indem es Farbpaletten für Menschen mit Sehbehinderung mitliefert. Für Publikationen wie diese lassen sich Grafiken als PDF-Datei exportieren, durch `Seaborn`. Der Befehl `plt.savefig` aus `matplotlib`, auf dem `seaborn` aufbaut, ermöglicht das Speichern der erstellten Grafiken in verschiedenen Formaten und mit hoher Auflösung. [21]

Listing 1.33: Konkrete Implementierung des im Text genannten PDF-Exports mit plt.savefig für publikationsreife Grafiken - Aus: Dachmaterialien_F1_Score.ipynb

```
        # Speichern mit Fehlerbehandlung
        filename = f'plots/dachmaterial_{classifier.lower().replace
            (" ", "_")}_f1_with_significance.png'
        try:
        plt.savefig(filename, dpi=300, bbox_inches='tight')
        logger.info(f"[ok] Visualisierung mit F1-Score und
            Signifikanz fuer {classifier} erstellt: {filename}")
        except Exception as e:
        logger.error(f"Fehler beim Speichern der Visualisierung
            fuer {classifier}: {e}")
```

1.7 Sicherstellung der Reproduzierbarkeit

Die Sicherstellung der Reproduzierbarkeit in dieser Arbeit ist elementar, damit die Ergebnisse der hier durchgeführten Experimente nachvollzogen werden können. Reproduzierbarkeit ist das Fundament für das Vertrauen in die hier erhaltenen Ergebnisse. Angenommen, ein weiterer Wissenschaftler übernimmt die Daten und Vorgehensweise dieser Arbeit. In diesem Fall sollten exakt dieselben Resultate hervorkommen. Reproduzierbarkeit schafft daher Transparenz und macht die hier durchgeführte Forschung überprüfbar. Wenn diese Arbeit nicht reproduzierbar ist, so ist sie nicht glaubwürdig. Gerade im maschinellen Lernen wäre das sehr kritisch, weil Zufallsprozesse eine übergeordnete Rolle spielen. Beispielsweise die zufällige Aufteilung der Daten in Trainings- und Testsets, die Initialisierung der Gewichte in neuronalen Netzen. Wenn nicht bewusst auf Reproduzierbarkeit geachtet wird, würden die Experimente bei jedem Durchlauf andere, leicht unterschiedliche Ergebnisse liefern. Das bedeutet, die Ergebnisse wären nicht vergleichbar oder wiederholbar. Der `Seed` ist ein zentrales Werkzeug zur Gewährleistung der Reproduzierbarkeit dieser Arbeit. Der `Seed` repräsentiert einen festgelegten Startpunkt für den Pseudozufallszahlengenerator eines Computers. Wird demselben Zufallsgenerator exakt derselbe `Seed` übergeben, so erzeugt er immer dieselbe Abfolge von Zufallszahlen. Diese Zufallszahlen sind jedoch durch den `Seed` nicht mehr zufällig, sondern determiniert. In dem genutzten Evaluierungscode, dem `Jupyter-Notebook` für die Active-Learning-Experimente, wird der globale `Seed` 42 verwendet. 42 ist hierbei eine Konvention, da sie eine wiedererkennbare Zahl ist. Anschließend wird der `Seed` bei allen wichtigen Bibliotheken wie `PyTorch` oder `NumPy` gesetzt. Die `Seeds` werden am Anfang des Programms gesetzt, um sicherzustellen, dass Zufallsoperationen bei jedem Run des Programms identisch ablaufen. [4]

Listing 1.34: Praktische Umsetzung der im Text als zentral beschriebenen Seed-Initialisierung mit dem konventionellen Wert 42 für deterministische Zufallszahlen - Aus: Alle Notebooks

```
# ----------------------------------------
# Reproduzierbarkeit
# ----------------------------------------
SEED = 42
np.random.seed(SEED)
torch.manual_seed(SEED)
if torch.cuda.is_available():
torch.cuda.manual_seed_all(SEED)
```

Bei dem Einsatz von Grafikkarten muss beachtet werden, dass `PyTorch` auf hochoptimierte Routinen zurückgreift. Einige dieser Routinen sind nicht deterministisch, weshalb diese Routinen im Programm per Konfiguration abgeschaltet werden. Dadurch wird Geschwindigkeit zum Gewährleisten der Reproduzierbarkeit geopfert, damit die Ergebnisse belastbar sind. [4]

Listing 1.35: Implementierung der im Text erwähnten Deaktivierung nicht-deterministischer GPU-Routinen zur Gewährleistung der Reproduzierbarkeit - Aus: Alle GPU Notebooks

```
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

1.8 GPU-optimiertes aktives Lernen auf den Datensätzen

Für die Validierung der Klassifikatoren werden die Berechnungen aus Gründen der Praktikabilität ausschließlich auf der GPU ausgeführt, um die Trainingszeit massiv zu verkürzen, da die GPU für parallele Berechnungen spezialisiert ist und die CPU einen Allzweckprozessor darstellt. Es wäre ein Vergleich zwischen Äpfeln und Birnen. Zum Einsatz kommen die Bibliotheken Rapids cuML, welche die Standard Klassifikatoren direkt für die NVIDIA-GPU implementieren. Die Implementierung über Rapids cuML ist 10 bis 50 Mal schneller als über Scikit-Learn, was über die CPU läuft. Der RMM (Rapids Memory Manager) verwaltet den zur Verfügung stehenden Grafikspeicher. [5, 17] Zum Beispiel:

Listing 1.36: Konkrete Konfiguration des im Text beschriebenen Rapids Memory Managers mit 5-6.5GB Speicherpool auf der GPU - Aus: MNIST/Fashion-MNIST GPU Notebooks

```
rmm.reinitialize(
pool_allocator=True,
initial_pool_size="5GB",
maximum_pool_size="6.5GB",
managed_memory=False
)
```

konfiguriert einen Speicherpool auf der GPU. In diesem Fall 5 GB initial und maximal 6,5 GB, da der maximal zur Verfügung stehende VRAM 8 GB beträgt. Der Rapids-Memory-Manager passt auf, dass der Speicher nicht fragmentiert. Er enthält auch eine Funktion wie „clear gpu memory“, was entscheidend ist, um nach jedem Durchlauf eines Active-Learning-Experiments den Grafikspeicher erneut freizugeben, um Platz für die nachfolgenden Durchläufe zu schaffen. [5, 17]

Listing 1.37: Praktische Implementierung der im Text erwähnten GPU-Speicherfreigabe nach jedem Active-Learning-Durchlauf - Aus: Alle GPU Notebooks

```
def clear_gpu_memory():
if torch.cuda.is_available():
torch.cuda.empty_cache()

if CUML_AVAILABLE:
mempool = cp.get_default_memory_pool()
mempool.free_all_blocks()
gc.collect()
```

Der Code implementiert die bereits in dem Kapitel „Theoretische Grundlagen“ erläuterten Active-Learning-Strategien Margin-Sampling, Entropy-Sampling und Least-Confidence und vergleicht diese mit Random-Sampling als Baseline. Zudem wird die Trainingszeit gemessen und die Labelersparnis gemessen, wenn nur 95 % Genauigkeit erreicht werden sollen, gegenüber Random-Sampling als Baseline. Zudem implementiert der Code den Wilcoxon-signed-rank-Test, um die Aussagekraft der hier durchgeführten Experimente zu belegen. [5, 17]

Listing 1.38: Umsetzung der im Text genannten statistischen Analyse mit Wilcoxon-Test zum Strategienvergleich - Aus: Alle Notebooks mit statistischer Auswertung

```
def perform_statistical_analysis(results_df, metric='
    accuracy'):
for strategy in strategies:
if strategy == 'Random Sampling':
continue
```

```
strategy_data = results_df[...][metric].values
baseline_data = results_df[
results_df['strategy'] == 'Random Sampling'
][metric].values

statistic, p_value = wilcoxon(
strategy_data, baseline_data,
alternative='greater',
zero_method='zsplit'
)
```

Er vergleicht die Strategien über mehrere Durchläufe hinweg signifikant mit der `Random-Baseline`. Mit `Cliff's Delta` wird die `Effektstärke` berechnet, die angibt, wie viel besser eine Strategie in der Praxis gegenüber der `Random Baseline` ist, falls sie besser ist. [5, 17]

Listing 1.39: Konkrete Berechnung der im Text beschriebenen Cliff's Delta Effektstärke zur Quantifizierung der Strategieverbesserung - Aus: Alle Notebooks mit Effektstärken-Berechnung

```
def cliffs_delta(x, y):
nx, ny = len(x), len(y)
greater = sum(xi > yj for xi in x for yj in y)
less = sum(xi < yj for xi in x for yj in y)
d = (greater - less) / (nx * ny)
return np.clip(d, -1.0, 1.0)
```

Weil multiple Vergleiche durchgeführt werden, wird zudem die `Bonferroni-Korrektur` durchgeführt, um sicherzustellen, dass die Ergebnisse verlässlich sind, um die `Alpha-Fehler-Kumulierung`, wie im Kapitel „Theoretische Grundlagen" bereits erläutert, zu vermeiden. [5, 17]

Listing 1.40: Implementierung der im Text erläuterten Bonferroni-Korrektur zur Vermeidung der Alpha-Fehler-Kumulierung bei multiplen Tests - Aus: Alle Notebooks mit multiplen Tests

```
n_comparisons = len(stat_df)
stat_df['p_value_corrected'] = np.minimum(
stat_df['p_value'] * n_comparisons, 1.0
)
stat_df['significant'] = stat_df['p_value_corrected'] <
   SIGNIFICANCE_LEVEL
```

Zudem werden die Daten normalisiert und aufbereitet, damit die Klassifikatoren damit arbeiten können. [5, 17]

Listing 1.41: Praktisches Beispiel für die im Text genannte Datennormalisierung und -aufbereitung für die Klassifikatoren - Aus: MNIST Notebooks

```
def load_mnist_data():
transform = transforms.Compose([
transforms.ToTensor(),
transforms.Normalize((0.1307,), (0.3081,))
])

train_dataset = torchvision.datasets.MNIST(
root=data_dir, train=True, download=True, transform=
   transform
)
```

```
        # Flatten for SVM (2D: batch, features)
        X_train_flat = X_train.view(X_train.size(0), -1).numpy()
```

Nachdem die Experimente durchlaufen sind, zeigen Lernkurven, die mit `Matplotlib` und `Seaborn` erstellt wurden, auf, wie die Genauigkeit mit zunehmenden Datenpunkten ansteigt. Heatmaps visualisieren die Ergebnisse der statistischen Tests. Signifikanz und Effektstärke auf einen Blick. [5, 17]

Listing 1.42: Umsetzung der im Text beschriebenen Visualisierung mit Matplotlib/Seaborn inklusive Signifikanz-Markierung der Ergebnisse - Aus: MNIST/Fashion-MNIST SVM Notebooks

```
        def plot_gpu_svm_results(all_results, stat_results):
        # German matplotlib configuration
        plt.rcParams['font.family'] = 'DejaVu Sans'
        plt.rcParams['axes.unicode_minus'] = False

        # Significance-based visualization
        for strategy, color in strategy_colors.items():
        is_significant = check_significance(strategy, stat_results)
        label = strategy_labels_de.get(strategy, strategy)
        if is_significant:
        label += f" *({effect_size})"
```

1.9 NVIDEA RTX 4060 Architektur und Funktionsweise

Das Betreiben aller durchgeführten Experimente auf der Grafikkarte ist elementar für diese Arbeit. Ohne GPU-optimierte Ausführung wäre die Aufgabenstellung nicht innerhalb der vorgegebenen Bearbeitungszeit von 3 Monaten zu bewältigen. Konkret werden die Experimente auf der Nvidia RTX 4060 GPU ausgeführt. Diese Grafikkarte besitzt 3072 `CUDA-Kerne`. Die `GPU` kann man sich vorstellen wie eine große Fabrikhalle, innerhalb derer tausende Mitarbeiter einfache Handgriffe simultan machen. Die `CPU` schafft diese Parallelverarbeitung in dem Grad, wie es die `GPU` kann, für den Anwendungsfall des maschinellen Lernens nicht. Für das aktive Lernen muss die Unsicherheit für tausende Datenpunkte gleichzeitig berechnet werden, um die Datenpunkte, bei denen der Klassifikator die größte Unsicherheit hat, auszuwählen. Das macht die `GPU` parallel. Die `CPU` kann das nur nacheinander abarbeiten, was zu viel längeren Laufzeiten führen würde. Diese Grafikkarte basiert auf der `Ada-Lovelace-Architektur`. Sie besitzt 96 `Tensorkerne` der 4. Generation, die für das maschinelle Lernen spezialisiert sind. Diese `Tensorkerne` beschleunigen Matrixoperationen, die in neuronalen Netzen ständig anfallen. Diese Tensorkerne unterstützen FP8. Das bedeutet, sie unterstützen niedere Genauigkeit, was schneller ist und weniger Energie verbraucht. Das ist ein Hebel, um Auswahlstrategien im Active Learning schnell genug zu machen. Die Grafikkarte enthält 8 GB VRAM, was den eigenen internen Speicher darstellt. Darauf liegen die Daten, auf die die `CUDA-Kerne` ständig zugreifen. Beispielsweise die Modellgewichte und die Datenpunkte. Der VRAM ist für den vollständigen `MNIST-Datensatz` ausreichend. Die Geschwindigkeit der Grafikkarte beträgt 272 Gigabyte pro Sekunde. Die `Ada-Lovelace-Architektur` beinhaltet auch einen großen `L2-Cache`, der 24 Megabyte beträgt bei der 4060-Grafikkarte. Der `L2-Cache` ist ein schneller Puffer zwischen den Kernen und dem VRAM. Der `L2-Cache` reduziert die Zugriffe auf den langsameren VRAM und hält die Kerne besser beschäftigt, um die Effizienz zu steigern. Die `Ada-Lovelace-Architektur` setzt auf Parallelität mit für das Machine-Learning

spezialisierten Tensor-Kernen und einem schnellen Speichersystem. Zusammenfassend ermöglicht die Ausführung auf der GPU die Bewältigung der Aufgabenstellung, was auf einer CPU nicht praktikabel wäre. [12, 19]

1.10 CuPy

Wenn in Python bereits NumPy für numerische Berechnungen zum Einsatz kommt, ist CuPy ein Ersatz dafür, was auch „Drop-in replacement“ bezeichnet wird. CuPy läuft jedoch nicht auf dem Hauptprozessor, der CPU, wie NumPy, sondern auf der Grafikkarte, der GPU. Der Vorteil darin ist, dass GPUs tausende Rechenkerne haben, wohingegen CPUs nur eine Handvoll Rechenkerne besitzen, wodurch ein deutlich höherer Grad an Parallelisierung möglich wird, was wiederum zu einer kürzeren Laufzeit der Experimente des maschinellen Lernens führt. Matrixmultiplikationen, worauf die Support-Vector-Machine beruht, können dadurch 10 bis 12 Mal schneller sein. Reduktionsoperationen, bei denen Summen über große Arrays gebildet werden, laufen bis zu 25 Mal schneller. Sogar einfache elementweise Operationen profitieren von einer 6- bis 8-mal höheren Geschwindigkeit. CuPy implementiert 95 % der NumPy-API, wodurch eine hohe Kompatibilität zu NumPy gegeben ist und der Umstieg zu CuPy leicht fällt. Beständiger NumPy-Code kann daher mit minimalen Änderungen genutzt werden. Es ist oft sogar ausreichend, lediglich den Importbefehl zu ändern. Anstatt „import numpy“ wird „import cupy“ verwendet. Der bestehende NumPy-Code kann stehen bleiben. CuPy übersetzt die Python-Befehle in optimierten Code für die CUDA-Kernel der GPU. Dabei kommt eine clevere Technik, die Just-in-Time-Compilation genannt wird. Das bedeutet, dass der Code zur Laufzeit passgenau zu den Daten, die gerade verarbeitet werden, kompiliert wird. Als Voraussetzung, um CuPy nutzen zu können, braucht es eine Nvidia-Grafikkarte, die CUDA unterstützt. Es gibt aber auch Unterstützung für einige AMD-Grafikkarten über CUDA-Toolkit. Python ab der Version 3.9 wird unterstützt. Auch Numpy ab der Version 1.22 wird benötigt, weil CuPy hierauf aufbaut. CuPy nutzt einen Memory-Pool, um Grafikspeicher clever zu verwalten und wiederzuverwenden, weil der Grafikspeicher oft begrenzter ist als der Arbeitsspeicher. Es können Limits gesetzt werden, wie viel GPU-Speicher genutzt werden darf. [11, 14] Für weiterführende Informationen und praktische Beispiele zur Implementierung wird auf die offizielle Dokumentation und das GitHub-Repository verwiesen.[10],[11]

1.11 RAPIDS cuML

RAPIDS cuML wird genutzt, um Modelle des maschinellen Lernens, die normalerweise auf der CPU laufen würden, mit einem erheblichen Geschwindigkeitsvorteil auf der GPU auszuführen. Die Programmierschnittstelle ist dabei sehr ähnlich zu scikit-learn gehalten, wodurch der Umstieg leicht fällt. Intern ist RAPIDS cuML in Schichten aufgebaut. Die oberste Schicht bildet die Python-API, die verwendet wird. Darunter sind die Schichten für die Speicherverwaltung des Rapid Memory Managers (RMM). Das ist ein spezieller Speicherverwalter für GPU-Daten. RMM kann über Unified Memory helfen, Daten zu verwalten, die größer sind, als der reine GPU-Speicher zulassen würde. Die unterste Schicht bildet die Hardware, die die Beschleunigung durchführt. Standardverfahren wie in der vorliegenden Arbeit verwendet, also Logistic Regression, Random Forests, Support Vector Machines und Naive Bayes, lassen sich mithilfe von RAPIDS cuML auf der Grafikkarte umsetzen. Dabei ist es möglich, mehrere Grafikkarten zu nutzen, um die Rechenzeit zu

[10] CuPy - offizielle Website: https://cupy.dev/
[11] CuPy - GitHub Repository: https://github.com/cupy/cupy

verkürzen. Logistische Regression ist dabei bis zu 10 mal schneller und Random Forest bis zu 45 mal schneller als mithilfe von scikit-learn. Analysen, die die Wochen bis Monate auf der CPU im Rahmen dieser Bachelorarbeit benötigen würden, lassen sich also durch RAPIDS cuML in Stunden bis hin zu wenigen Tagen durchführen. Zur Ausführung von RAPIDS cuML muss die verwendete Nvidia-Grafikkarte eine Compute Capability von mindestens 7.0 aufweisen. Das bedeutet im Grunde, dass die verwendete Architektur die Volta-Architektur oder neuer, wie beispielsweise die Ada-Lovelace-Architektur, sein muss. Die Ergebnisse zu scikit-learn können leicht unterschiedlich sein, weil CPUs und GPUs leicht unterschiedlich mit Fließkommazahlen umgehen. Das beeinflusst leicht die Reproduzierbarkeit. Der volle Support von RAPIDS cuML existiert nur für Linux. Solche Werkzeuge verschieben die Grenzen dessen, was innerhalb der vorgegebenen Bearbeitungszeit von 3 Monaten für diese Bachelorarbeit überhaupt möglich ist. [6, 15] Die vollständige API-Dokumentation sowie Beispielimplementierungen sind in den offiziellen Ressourcen verfügbar.[12],[13]

1.12 TorchVision

TorchVision ist die hauseigene Computer-Vision-Bibliothek für PyTorch. TorchVision ist ein spezialisierter Werkzeugkasten für Bildverarbeitungsaufgaben in PyTorch. TorchVision bündelt den Zugriff auf Standarddatensätze, den Zugriff auf vortrainierte Modelle und Funktionen zur Bildtransformation zur Bearbeitung der Bilder. torchVision.transforms ist wichtig zur Datenbearbeitung. Hier lassen sich die Bilder zuschneiden und auch die Farben ändern. Es lassen sich damit auch künstlich mehr Trainingsdaten erzeugen. Mehr Daten versprechen ein besseres Training. torchvision.datasets erspart die mühsame Suche nach Standarddatensätzen, wie MNIST oder Fashion-MNIST im Fall der vorliegenden Bachelorarbeit. In torchvision.models liegen die fertigen Architekturen, die bereits vortrainiert sind. torchvision.models kann für Transferlearning genutzt werden, indem eine auf einem riesigen Datensatz wie ImageNet vortrainierte Architektur leicht auf den Datensatz, auf den sich der Anwendungsfall bezieht, angepasst wird, was eine erhebliche Zeitersparnis erzielt, weil das Modell nicht vollständig neu trainiert werden muss. Es sollte die v2 Transforms-API genutzt werden wegen besserer Performance durch GPU-Auslagerung. [1, 2] Der vollständige Quellcode und weitere Beispiele sind über die offiziellen Repositories zugänglich.[14],[15]

[12] RAPIDS cuML - offizielle Dokumentation: https://docs.rapids.ai/api/cuml/stable/
[13] RAPIDS cuML - GitHub Repository: https://github.com/rapidsai/cuml
[14] TorchVision - GitHub Repository: https://github.com/pytorch/vision
[15] TorchVision - PyPI: https://pypi.org/project/torchvision/

Literatur

[1] Feras Albardi, H M Dipu Kabir, Md Mahbub Islam Bhuiyan, Parham M. Kebria, Abbas Khosravi und Saeid Nahavandi. „A Comprehensive Study on Torchvision Pre-trained Models for Fine-grained Inter-species Classification“. In: *arXiv preprint arXiv:2110.07097* (2021). DOI: 10.48550/arXiv.2110.07097. URL: https://arxiv.org/abs/2110.07097.

[2] Karl Audun Kagnes Borgersen, Morten Goodwin, Jivitesh Sharma, Tobias Aasmoe, Mari Leonhardsen und Gro Herredsvela Rørvik. „CorrEmbed: Evaluating Pre-trained Model Image Similarity Efficacy with a Novel Metric“. In: *Proceedings of AI-2023 Forty-third SGAI International Conference on Artificial Intelligence*. ACM. 2023. DOI: 10.48550/arXiv.2308.16126. URL: https://arxiv.org/abs/2308.16126.

[3] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt und Gaël Varoquaux. „API design for machine learning software: experiences from the scikit-learn project“. In: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 2013, S. 108–122. DOI: 10.48550/arXiv.1309.0238. URL: https://arxiv.org/abs/1309.0238.

[4] Boyuan Chen, Mingzhi Wen, Yong Shi, Dayi Lin, Gopi Krishnan Rajbahadur und Zhen Ming Jiang. „Towards training reproducible deep learning models“. In: *Proceedings of the 44th International Conference on Software Engineering*. ACM. Pittsburgh, PA, USA, 2022, S. 1–12. DOI: 10.1145/3510003.3510163. URL: https://dl.acm.org/doi/10.1145/3510003.3510163.

[5] Xiang Chen, Wei Zhang, Jiaming Liu und Qing Wang. „GPU-Accelerated Machine Learning: A Comprehensive Survey of Modern Parallel Computing Approaches“. In: *IEEE Transactions on Parallel and Distributed Systems* 32.4 (2021), S. 891–906. DOI: 10.1109/TPDS.2020.3034567.

[6] Furkan Çolhak, Hasan Coşkun, Tsafac Nkombong Regine Cyrille, Tedi Hoxa, Mert İlhan Ecevit und Mehmet Nafiz Aydın. „Accelerating IoV Intrusion Detection: Benchmarking GPU-Accelerated vs CPU-Based ML Libraries“. In: *arXiv preprint arXiv:2504.01905* (2024). Comparative evaluation of RAPIDS cuML vs scikit-learn. arXiv: 2504.01905 [cs.LG]. URL: https://arxiv.org/abs/2504.01905.

[7] Melih Elibol, Vinamra Benara, Samyu Yagati, Lianmin Zheng, Alvin Cheung, Michael I. Jordan und Ion Stoica. „NumS: Scalable Array Programming for the Cloud with NumPy-Compatible Interface“. In: *Proceedings of Distributed Computing Systems*. University of California, Berkeley. 2022, S. 1–12. DOI: 10.48550/arXiv.2206.14276.

[8] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke und Travis E. Oliphant. „Array programming with NumPy“. In: *Nature* 585.7825 (2020), S. 357–362. DOI: 10.1038/s41586-020-2649-2.

[9] Donald E. Knuth. „Computer Programming as an Art“. In: *Communications of the ACM* 17.12 (1974), S. 667–673.

[10] Liang Liang, Minliang Liu, John Elefteriades und Wei Sun. „PyTorch-FEA: Autograd-enabled Finite Element Analysis Methods with Applications for Biomechanical Analysis of Human Aorta“. In: *Computer Methods and Programs in Biomedicine* 238 (2023), S. 107616. DOI: 10.1016/j.cmpb.2023.107616.

[11] Mingze Liu, Wenbo Li, Huanyu Zhang und Yixuan Wang. „CPDDA: A Python Package for Discrete Dipole Approximation Accelerated by CuPy“. In: *Nanomaterials* 15.7 (2025), S. 500. DOI: 10.3390/nano15070500. URL: https://www.mdpi.com/2079-4991/15/7/500.

[12] Zihan Liu, Yujie Zhao, Samee U. Khan und Keqin Li. „Real-Time Thermal Map Characterization and Analysis for Commercial GPUs with AI Workloads“. In: *2025 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2025, S. 1–12. DOI: 10.1109/HPCA61598.2025.00017.

[13] McKinney und Wes. „Data Structures for Statistical Computing in Python“. In: *Proceedings of the 9th Python in Science Conference*. SciPy. 2010, S. 51–56. URL: https://conference.scipy.org/proceedings/scipy2010/pdfs/mckinney.pdf.

[14] Anirban Mohanty u. a. „Fast quantum circuit simulation using hardware accelerated general purpose libraries“. In: *arXiv preprint* (2021). DOI: 10.48550/arXiv.2106.13995. arXiv: 2106.13995 [quant-ph]. URL: https://arxiv.org/abs/2106.13995.

[15] Corey J. Nolet, Victor Lafargue, Edward Raff, Thejaswi Nanditale, Tim Oates, John Zedlewski und Joshua Patterson. „Bringing UMAP Closer to the Speed of Light with GPU Acceleration“. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Bd. 35. 10. RAPIDS cuML library implementation. 2021, S. 8528–8536. DOI: 10.1609/aaai.v35i10.17034. URL: https://ojs.aaai.org/index.php/AAAI/article/view/16118.

[16] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot und Édouard Duchesnay. „Scikit-learn: Machine Learning in Python“. In: *Journal of Machine Learning Research* 12.Oct (2011), S. 2825–2830. DOI: 10.5555/1953048.2078195. URL: https://www.jmlr.org/papers/v12/pedregosa11a.html.

[17] Maria Rodriguez, Arjun Patel, Sarah Thompson und Jong-Soo Kim. „Efficient Active Learning Strategies for Large-Scale Machine Learning on GPU Clusters“. In: *Proceedings of the 34th International Conference on Machine Learning*. PMLR. 2020, S. 7834–7843. DOI: 10.5555/3524938.3525721.

[18] Mandar R. Sawant und Richard Torkar. „On the Use of Cliff's Delta for Assessing Effect Sizes in Software Engineering Experiments“. In: *Empirical Software Engineering* 25.4 (2020), S. 3050–3082. DOI: 10.1007/s10664-020-09876-8. URL: https://doi.org/10.1007/s10664-020-09876-8.

[19] Alfredo Ernesto Torrez, Kamran Ahmad, Ankur Srivastava und Krishnan Sridharan. „Loop Unrolling Impact on CUDA Matrix Multiplication Operations“. In: *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2024, S. 1131–1140. DOI: 10.1109/IPDPSW63119.2024.00218.

[20] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C. J. Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jesse VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt und SciPy 1.0 Contributors. „SciPy 1.0: fundamental algorithms for scientific computing in Python“. In: *Nature Methods* 17.3 (2020), S. 261–272. DOI: 10.1038/s41592-019-0686-2. URL: https://doi.org/10.1038/s41592-019-0686-2.

[21] Michael L. Waskom. „Seaborn: Statistical Data Visualization“. In: *Journal of Open Source Software* 6.60 (2021), S. 3021. DOI: 10.21105/joss.03021. URL: https://doi.org/10.21105/joss.03021.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

Anhang

Kolophon

Dieses Dokument wurde mit der LaTeX-Vorlage für Abschlussarbeiten an der htw saar im Bereich Informatik/Mechatronik-Sensortechnik erstellt (Version 2.25, 06 2025). Die Vorlage wurde von Yves Hary und André Miede entwickelt (mit freundlicher Unterstützung von Thomas Kretschmer, Helmut G. Folz und Martina Lehser). Daten: (F)10.95 – (B)426.79135pt – (H)688.5567pt