DeepLearning

Kurzfassung:

Eine kleine Funktion, die die Zuordnung der Klassen für die Validierungs-pipeline übernimmt und so für eine einheitliche Zuordnung sorgt.

Schreiben von Funktionen, die es ermöglichen Metriken in Graphen zu verwandeln, die auch über mehrere Läufe der Validierungs-pipeline aufzeichnen.

Des Weiteren werden 2 weitere Metriken implementiert. Die eine Metrik gibt die Genauigkeit einer Klasse über einen Validierungslauf aus, während die andere die Precision angibt.

Grund:

Mithilfe eines Graphen kann man eine KI einfacher und schneller bewerten, da man sofort erkennen kann ob sich die Metriken verbessern oder verschlechtern. Des Weiteren hilft es enorm, wenn man weiß welche Klassen in bestimmten Metriken schlechter sind, da man dann dort entgegen steuern kann, zum Beispiel durch mehr Trainingsdaten oder überprüfen der Validierung und Trainingsdaten auf Fehler. Darstellung über Punkte und zusätzlich ein interpolierter Graph dritten Grades um einen ungefähren Verlauf mit Konfidenzintervall. Die Punkte sind in einer komplementären Farbe und zeigen die absoluten Werte an, während der interpolierte Graph den ungefähren Verlauf der Validierung zeigen soll, um sehen zu können, ob die KI noch besser wird.

Idee:

Zuordnungsfunktion ordnet die Klassen die in test_labels stehen, den echten Klassen zu, da test_labels, ja nur die Klassen, die in der Validierung vorkommen, somit also eine andere Nummerierung hat.

Um über mehrere Läufe die Graphen zeichnen zu können, werden die Daten erst von einer Funktion in eine Textdatei abgespeichert.

Eine weitere Funktion liest die Textdateien aus und erzeugt dann die zugehörigen Graphen.

Mit den neuen Metriken lassen sich Rückschlüsse auf die KI und das Validierungsdatenset ziehen, da man die Genauigkeit der einzelnen Klassen sehen kann, sowie die Wahrscheinlichkeit, dass eine bestimmte Klasse richtige erkannt wurde. Dadurch kann man dann gezielt die Klassen analysieren, die nicht gut erkannt werden und somit das Trainingsdatenset oder das Validierungsdatenset überarbeiten.

Probleme:

Es sollte für jede Metrik und Klasse nur eine Text und eine PDF-Datei geben, um die Zugehörigkeit zu erkennen, dadurch kann es jedoch schnell unübersichtlich werden oder der Ordner „zugemüllt“ werden. Die Klassen haben in der Validierungs-pipeline nur Zahlen als Namen.

Lösung:

Als Abhilfe wird von der Funktion, die die Textdateien erstellt auch ein Ordnerverzeichnis für die Metrik angelegt, durch das Verzeichnis lassen sich die Dateien dann sortiert abspeichern und es bleibt übersichtlich. Der Nutzer erstellt eine Mapping-Datei, mit dieser lassen sich zu den Zahlen der

Klassen sinnvolle Namen zuordnen, damit können dann die Dateien der Klassen abgespeichert werden.

Dateien:

Accuracy.py

Mapping.py:

Plot_file.py

Plot_graph.py:

Assign.py (Zuordnung von Klassen zu test_labels)

```
def assign(class_names, test_labels):
    for i in range(len(test_labels)):
        test_labels[i] = class_names[test_labels[i]]
    return test_labels
```

Die Assign.py ordnet test_labels die richtigen Klassen zu, indem sie an die i-te Stelle in test_labels die Klasse reinschreibt, die in class_names an der Stelle test_labels[i] liegt. Dies funktioniert, da die Validierungs-pipeline die test_labels einfach nach 1te, 2te, 3te, … Klasse vergibt, also schauen wir in class_names an der Stelle nach, die dem Wert von test_labels[i] entspricht.

Bsp.:

test_labels[0, 1, 2, 3] und class_names[1, 12, 14, 18]

Die Klassen-„Namen“ stehen in class_names, test_labels nummeriert von 0 beginnend. Um jetzt die richtigen Klassen der Nummerierung zuzuordnen, gucken wir jetzt in class_names was an der Stelle 0 steht und tragen es in test_labels an der ersten stelle ein. Dies machen wir bis wir diesen Schritt für alle Stellen in test_labels getan haben und fertig ist die Zuordnung.

Accuracy.py (Alte und Neue Metriken)

In Accuracy.py wurde die ursprüngliche Metrik der Validierungs-pipeline eingefügt, sowie 2 neue Metriken geschrieben.

```
def class_accuracy(predictions, test_labels):

    class_predictions = {}
    class_accuracies = {}

    for prediction, test_label in zip(predictions, test_labels):
        if test_label in class_predictions:
            class_predictions[test_label].append(prediction == test_label)
        else:
            class_predictions[test_label] = [prediction == test_label]

    for k, v in class_predictions.items():
        count = 0
        for match in v:
            if match:
                count += 1
```

```
        class_accuracies[k] = count/len(v)

    return class_accuracies
```

Die erste Metrik ist die class_accuracy, dafür erstellen wir erst einmal 2 arrays, in dem einen Speichern wir die Vorhersagen zu einer bestimmten Klasse, also wie oft diese Klasse richtig vorausgesagt wurde und wie oft falsch. In dem anderen array speichern wir dann an den Stellen, die den Klassen entsprechen die Klassengenauigkeit, also wie genau die Klassen erkannt wurden.

Über die zip-Funktion können wir über den predictions (also ground truth) und den test_labels array iterieren in dem wir 2 Laufvariablen verwenden. Danach prüfen wir mit einer if-Abfrage ob wir die Klasse die dem Wert test_label entspricht bereits in unserem class_predictions array ist, falls ja fügen wir an dieser Stelle einen booleschen Begriff hinzu, je nachdem ob beide übereinstimmen (dann true) oder nicht (false). Am Ende zählen wir mit einer Schleife die wahren Fälle durch und teilen sie durch die Gesamtlänge, um auf die Genauigkeit der Klasse zu kommen.

Bsp.:

In der Validierung sind 3 Bilder, ein Stoppschild und 2 Vorfahrtschilder, die KI erkennt alle als Stoppschilder. Da die KI das 1te Bild was tatsächlich ein Stoppschild war als Stoppschild erkennt ist die Class-Accuracy 100%, bei den Vorfahrtsschildern jedoch 0 da es diese nicht erkannt hat, hätte es eines der Vorfahrtschilder erkannt, dann wäre die Class-Accuracy 50% für Vorfahrtsschilder.

```
def precision(predictions, test_labels):
…
if prediction in class_predictions:
            class_predictions[prediction].append(prediction == test_label)
…
```

Die 2te Metrik, basiert auf einem ähnlichen Prinzip, jedoch fragen wir diesmal nicht die test_labels ab, sondern die predictions, dies hat zur Folge, dass wir in class_predictions statt Klasse erkannt und nicht erkannt, die „true“ und „false“ „positives“ bekommen. Also schauen wir diesmal nicht von der ground truth Seite und checken, ob die KI das Richtig hat, sondern wir schauen von der KI-Seite und gucken, wie oft er mit der Vorhersage eine Klasse richtig oder falsch in der Validierung erkannt hat.

Bsp.:

KI denkt, das die ersten 3 Bilder von einem Stoppschild sind, dabei ist nur das Erste von einem Stoppschild. True positive = 1; false positive = 2;

Somit 1/3 als precision.

Mapping.py (Zuordnung der Namen in einem Dictionary)

```
mapping = {
    0 : "20kmh",
    1 : "30kmh",
    2 : "50kmh",
      …
   42 : "Überholverbot_LKW_aufgehoben",
    }
```

Mapping.py ist ein einfaches dictionary, was von dem Nutzer für die Anwendung geschrieben werden muss, darüber können die Skripte dann den erstellten Dateien und den Graphen Namen zuordnen. Wichtig ist hierbei, dass die Namen einzigartig sind und nicht mehrmals vorkommen, da man sonst Dateien überschreiben würde.

Plot_file.py (Speichern der Daten in Textdateien und Erstellung Ordnerverzeichnis)

```
from pdb import line_prefix
from mapping import *
import os

def log_in_file(class_label, metric_name, metric_number):

    class_name = mapping[class_label]
    file_name = class_name + "_" + metric_name + ".txt"
    path_to_file: str = "Plots/" + metric_name + "_Plots/Textfiles/" +
file_name

    try:
        os.makedirs("Plots/" + metric_name + "_Plots/Textfiles")
        os.makedirs("Plots/" + metric_name + "_Plots/Graphs")
    except FileExistsError as e:
        pass

    try:
        open(path_to_file)
        existance = True
    except FileNotFoundError:
        existance = False

    if existance == False :
        with open(path_to_file, "w") as p:
            line = "1 : " + str(metric_number)
            p.write(line)

    elif existance == True :
        with open(path_to_file, "r+") as p:
            last_line = p.readlines()[-1]
            i = 1 + int(last_line.split(" : ")[0])
            line = "\n" + str(i) + " : " + str(metric_number)
            p.write(line)
        p.close()
```

Als erstes holen wir uns in der Plot_file.py ein paar Strings zusammen, um die Graphen mit Namen zu versehen und eine übersichtliche Ordnerstruktur zu generieren, wichtig ist hierbei, dass dieses Programm für Linux geschrieben wurde, sollte man dieses Programm unter einem anderen OS wie zum Beispiel Windows verwenden muss man evtl. die „/“ bis zum ersten except mit „\“ (im Falle von Windows) ersetzen.

Im ersten try/except wird getestet, ob die Ordnerstruktur für die Metrik bereits existiert, andernfalls wird sie erstellt. Im nächsten try/except wird geprüft ob zu der Metrik und Klasse bereits Textdateien existieren, danach wird diese entweder erzeugt und die erste Reihe mit x = 1 und y = die Zahl der Metrik erstellt oder eine neue Zeile mit x = x Wert der letzten Zeile + 1 und y = s.o. (siehe oben).

Ein weiterer wichtiger Punkt ist es die Datei zu schließen und wieder frei zu geben, siehe p.close().

Plot_graph.py (Plotten der Graphen)

```
import matplotlib.pyplot as plt
from mapping import *
import numpy as np
import os

def plot_graph(class_label, metric_name):

    class_name: str = mapping[class_label]
    file_name: str = class_name + "_" + metric_name + ".txt"
    path_to_file: str = "Plots/" + metric_name + "_Plots/Textfiles/" +
file_name
    path_to_plot: str = "Plots/" + metric_name + "_Plots/Graphs/" +
file_name.rstrip(".txt")

    x_axis = []
    y_axis = []

    try:
        os.remove(path_to_plot)
    except FileNotFoundError:
        pass

    try:
            open(path_to_file)
            existance = True
    except FileNotFoundError:
            existance = False


    if existance == False :
        print("There is no file for this metric \n")


    elif existance == True :
        with open(path_to_file, "r") as p:
          for row in p:
            row = row.split(' : ')
            x_axis.append(int(row[0]))
            y_axis.append(float(row[1]))
        p.close()

    x = np.linspace(1, len(x_axis), len(x_axis))
    a, b, c, d = np.polyfit(x, y_axis, deg=3)
    y_est = a * x**3 + b * x**2 + c * x + d
    y_err = (np.array(y_axis)-y_est).std() * np.sqrt(1/len(x) + (x -
x.mean())**2 / np.sum((x - x.mean())**2))
    plt.plot(x, y_est, "-")
    plt.fill_between(x, y_est - y_err, y_est + y_err, alpha=0.2)
```

```
    plt.plot(x_axis, y_axis, "o", color="tab:red")

    plt.xlabel('Number of iterations ', fontsize = 12)
    plt.ylabel('Value of ' + metric_name, fontsize = 12)

    plt.axis([1, None, -0.05, 1.05)
    plt.yticks(np.arange(0, 1.05, 0.05)
    plt.xticks(np.arange(1, len(x_axis) + 1, int(1 + len(x_axis)/10)))
    plt.xticks(rotation = 90)

    plt.title( class_name + " " + metric_name + ' over time', fontsize = 20)
    plt.savefig(path_to_plot)
    plt.cla()
```

Das plot_graph Skript findet über die mitgegebenen Informationen die Textdateien in der Ordnerstrukur, des Weiteren erzeugen wir einen Array für unsere x-Achse und einen für die y-Achse.

Danach wird der alte Plot gelöscht, falls er existiert und geprüft, ob die nötige Textdatei existiert. Wenn die Textdatei existiert, wird durch jede Zeile gegangen und die x und y Koordinaten unseren Arrays zugeordnet. Danach erzeugen wir einen weiteren Array für die x-Achse den wir zum Rechnen verwenden (ansonsten würden wir in den darauffolgenden Funktionen von numpy Typenprobleme bekommen). Nun erzeugen wir eine interpolierte Linie zwischen unseren Punkten, eine Funktion 3ten Grades hat sich nach einigen Tests als am besten geeignet ergeben, da diese Funktion nicht zu stark versucht alle Punkte zu erreichen aber auch nicht einfach eine sinkende Linie ist so bald ein paar Punkte weiter unten sind. Danach berechnen wir y_err was eine Standardabweichung für die Polynomfunktion erzeugt. Danach wird die Funktion geplottet und eine Fläche mit fill_between hinterlegt, die die Abweichung darstellt.

Danach kommen einige Einstellungen wie Beschriftung, Schrittweite und Titel. Was hier auch hervorzuheben ist, ist plt.cla() da ansonsten alle nachfolgenden Graphen mit den vorherigen überlappen.