KARLSRUHER INSTITUT FÜR TECHNOLOGIE

ENTWURF

Neuronale Netze zur Bilderklassifizierung auf Heterogenen Plattformen

vorgelegt von:

Viet Doan Xuan Pham
Friedemann David Claus
Aleksandr Eismont
Jakub Marceli Trzciński
Dmitrii Seletkov

Betreuer:

M.Sc. Dennis Weller
M.Sc. Sarath Mohanachandran Nair

8. Juli 2019

Inhaltsverzeichnis

1 Einleitung

Folgendes Entwurfsdokument des Projektes „AlexLens“ entstand im Rahmen des Moduls „Praxis der Softwareentwicklung – PSE“ am Lehrstuhl Programmierparadigmen - IPD Snelting. Das Modul ist Teil des Bachelorstudiengangs Informatik am Karlsruher Institut für Technologie.

Die Anwendung ermöglicht es dem Nutzer im wesentlichen, Bilder mittels des vortrainierten neuronalen Netzes AlexNet zu klassifizieren. Dabei wird zwischen drei verschiedenen Betriebsmodi unterschieden (Hohe Performance, geringer Leistungsverbrauch und hohe Energieeffizienz). Außerdem soll es mittels Transfer-Learnings möglich sein, das neuronale Netz auf weitere Objektklassen hin weiter zu trainieren.

Wie bereits im Spezifikationsdokument beschrieben, wird die Anwendung auf einem Host PC des CDNC Instituts betrieben, weshalb die wesentliche Hardwareunterstützung auf diesen Host PC fokussiert ist. Der Host PC hat Zugriff auf die folgenden Plattformen: Eine Intel CPU, eine Intel GPU sowie vier Intel Movidius Neural Compute Sticks der ersten Generation. Zum jetzigen Zeitpunkt ist eine Unterstützung der FPGA-Plattform nicht vorgesehen, jedoch soll die Anwendung so aufgebaut sein, dass eine spätere Unterstützung durchaus möglich wäre.

Ziel dieses Entwurfsdokumentes soll eine klare, genaue und gewissenhafte Dokumentation der Ergebnisse der Entwurfsphase sein. Es soll die Systemzerlegung in Pakete bzw. Module, Klassen und Schnittstellen dargestellt werden, sowie der globale Kontrollfluss mittels geeigneter Diagramme sowie detaillierten textuellen Beschreibungen erkenntlich sein.

Wir haben uns für ein modulares und objektorientiertes Konzept beim Entwurf entschieden. Die Anwendung soll in C++ geschrieben werden. Bibliotheken die wir für die Plattformen nutzen sind OpenVino (Intel Movidius NCS) sowie OpenCL (CPU und GPU). Für die GUI nutzen wird die Qt Bibliothek. Für die Skalierung der Bilder soll ebenfalls eine Bibliothek genutzt werden, die jedoch noch nicht fest steht.

Im folgenden Kapitel möchten wir darlegen, welche Wunschkriterien der Spezifikationsphase übernommen wurden und welche weiterhin offen sind. In Kapitel 3 wird die Paketstruktur dargestellt und erläutert. Anschließend wird in Kapitel 4 auf die einzelnen Pakete eingegangen.

2 Aktualisierung der Wunschkriterien

Von den Wunschkriterien flossen die Unterstützung der GPU-Plattform, die Implementierung von Data Augmentation und das Anbieten eines optimalen Modus in den Entwurf mit ein (WK10, WK30 und WK40).
Die anderen Wunschkriterien wurden wegen Priorisierung aus zeitlichen Gründen nicht weiter ausgeführt. Dies heißt jedoch nicht unbedingt, dass sie nicht trotzdem noch implementiert werden.
WK50 (Weiteres Deep Neural Network unter Nutzung von Bibliotheken) wurde insofern teilweise realisiert, als dass der Anwender nun weitere Deep Neural Networks benutzen kann, indem er die Konfigurationsdatei des Neuronalen Netzes verändert. So kann er zum Beispiel zusätzliche Schichten hinzufügen oder die Learning-Rate verändern. Möglicherweise wird auch dieses Wunschkriterium noch in Gänze eingebaut werden. Standardmäßig sind beide Konfigurationsdateien gemäß AlexNet eingestellt. Die Möglichkeit über Konfigurationsdateien des Neuronalen Netzes und des Transfer-Learnings zusätzliche Funktionalität zu nutzen wurde im Pflichtenheft noch nicht erwähnt.
An den Musskriterien gibt es keine Änderungen, sie konnten vollständig entworfen werden.

3 Paketstruktur

Um solch ein komplexes System zustande zu bringen, haben wir den Architekturstil Model-View-Controller ausgewählt. Ziel dieses Stils ist ein flexibler Programmentwurf, der eine spätere Änderung oder Erweiterung, ohne negative Auswirkungen auf die jeweiligen anderen Teile der Software zu haben, vereinfacht und eine Wiederverwendbarkeit der einzelnen Komponenten ermöglicht. Darüber hinaus erleichtert dies die Aufteilung der Arbeit auf verschiedene Teams.
Das Paket „GUIModule“ repräsentiert die View und ist für die Darstellung der Daten des Modells sowie die Realisierung der Benutzerinteraktionen zuständig.
Die Pakete „EventModule“, „ManagerModule“und „PlatformModule“ repräsentieren den Controller. Das EventModule bearbeitet alle Aktionen des Nutzers, die im Programm erfasst wurden. Das ManagerModule verwaltet das GUIModule und die Model-Klassen. Das PlatformModule dient zur Verwaltung der Berechnungen auf den heterogenen Plattformen.
Die Pakete „NeuralNetModule“, „TrainingModule“, „ImageEditorModule“, „DataAugmentationModule“und „MathModule“ repräsentieren das Model. Ziel des NeuralNetModules ist die Bilderklassifizierung, TrainingModule - das Trainieren eines ausgewählten Neuronalen Netz, ImageEditorModule - Bildverarbeitungsoptionen, DataAugmentationModule - die Erweiterung der Trainingsdaten und MathModule - die Modellierung von grundlegenden mathematischen Operationen mit Tensorobjekten. Somit ist das Model von der View und dem Controller unabhängig.

4 Paketbeschreibungen

4.1 ManagerModule

4.1.1 Funktion

Das ManagerModule ist das zentrale Controller-Modul des Programms. Deswegen bietet es auch mit der main()-Funktion den Eintrittspunkt in die Anwendung.

4.1.2 Verwendung im Kontext

Über die Schnittstelle **IManager** wird auf das Modul zugegriffen. Das GUIModule erhält über das Entwurfsmuster Beobachter von Klassen des ManagerModules Benachrichtigungen über Zustandsänderungen (zum Beispiel, wenn ein Bild fertig klassifiziert wurde). Den neuen Zustand holen sich die Beobachter des GUIModules dann über die Schnittstelle **IManager**. EventModule greift über **IManager** auf das Modul zu, um Änderungen (zum Beispiel, wenn der Nutzer die Klassifizierung gestartet hat) am Programmzustand anzuregen...

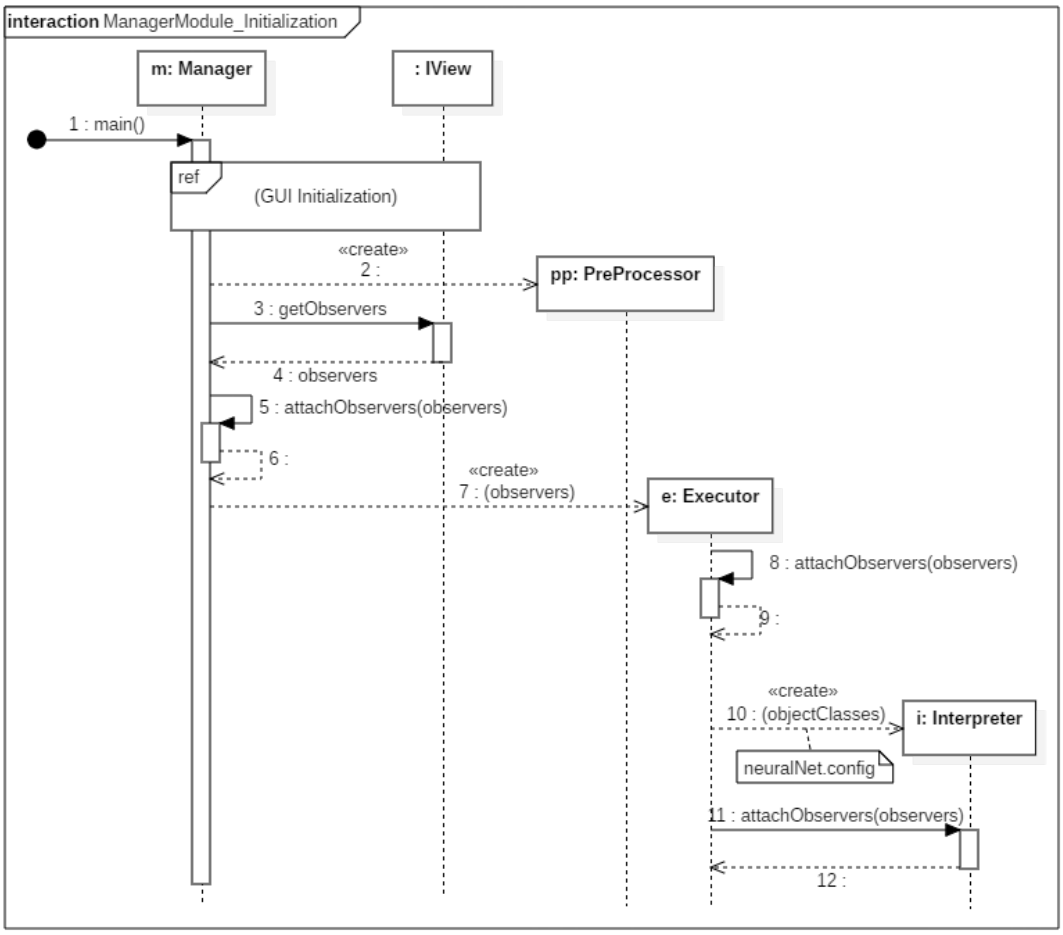

Abbildung 1: Sequenzdiagramm zur Initialisierung des ManagerModules

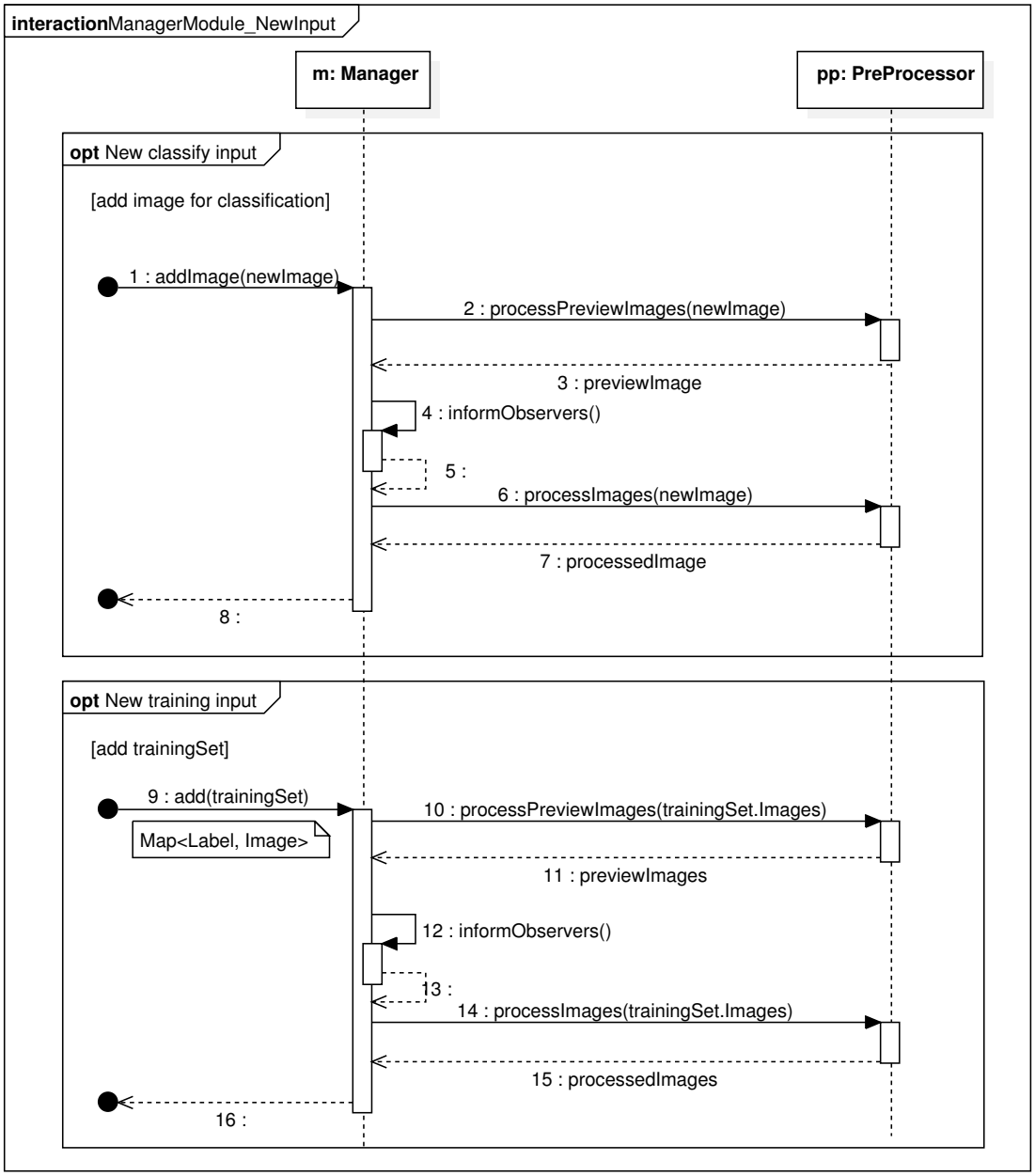

Abbildung 2: Sequenzdiagramm zur neuen Eingabe

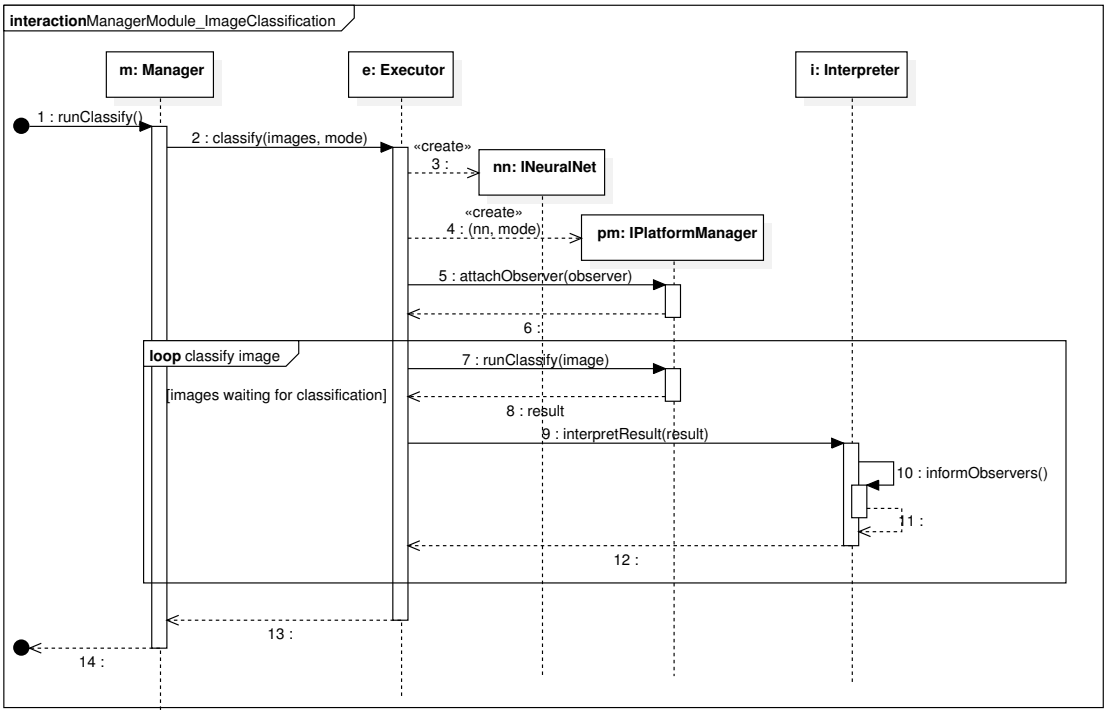

Abbildung 3: Sequenzdiagramm zur Bilderklassifizierung

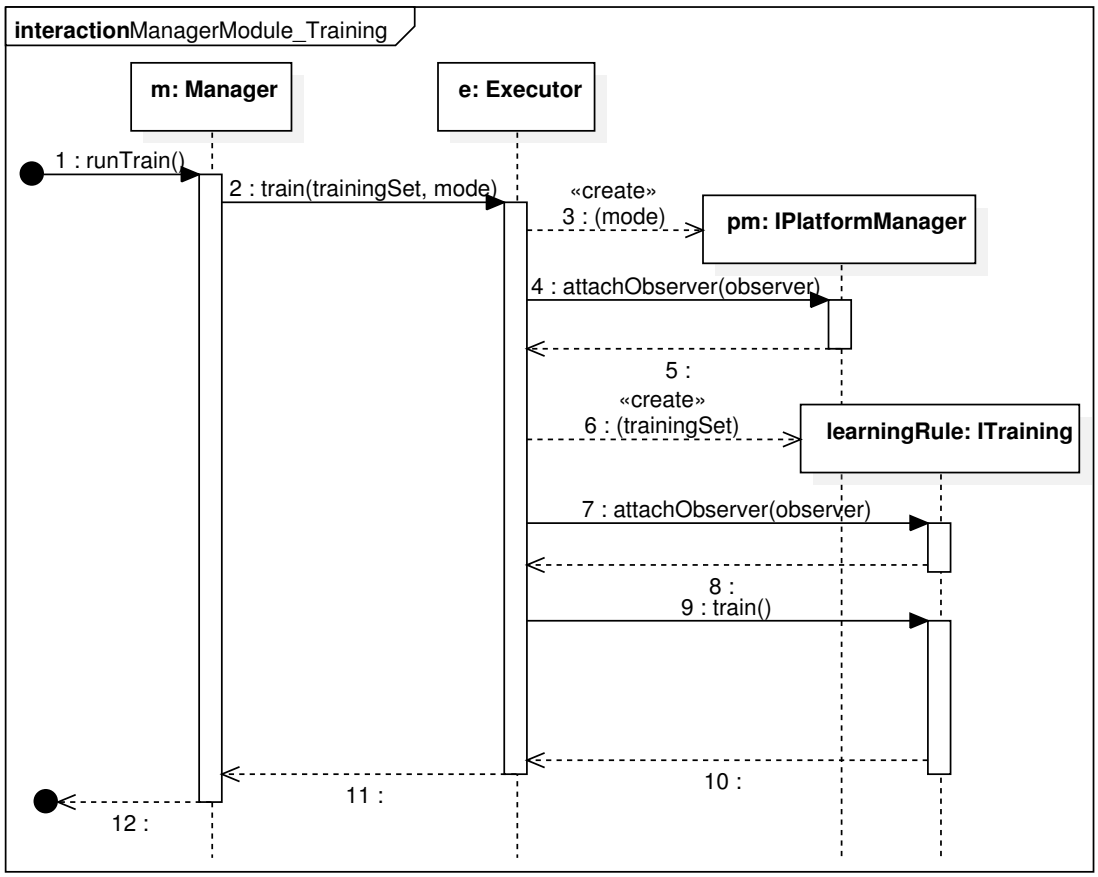

Abbildung 4: Sequenzdiagramm zum Training

4.1.3 Aufbau

Die zentrale Klasse **Manager** implementiert die Schnittstelle **IManager**. **Manager** bietet die main()-Funktion und stößt bei deren Ausführung die Initialisierung des Programms an. Außerdem ist **Manager** über **IManager** der Ansprechpartner des GUIModules (über das Beobachter-Entwurfsmuster) und des EventModules, wie im Abschnitt „Verwendung im Kontext“ beschrieben. Zum vorverarbeiten der Bilder benutzt **Manager** die Klasse **PreProcessor** und zum klassifizieren oder trainieren von Bildern die Klasse **Executor**. Ist ein Bild fertig klassifiziert, benutzt **Executor** die Klasse **Interpreter** zur Verarbeitung des Ergebnisses.

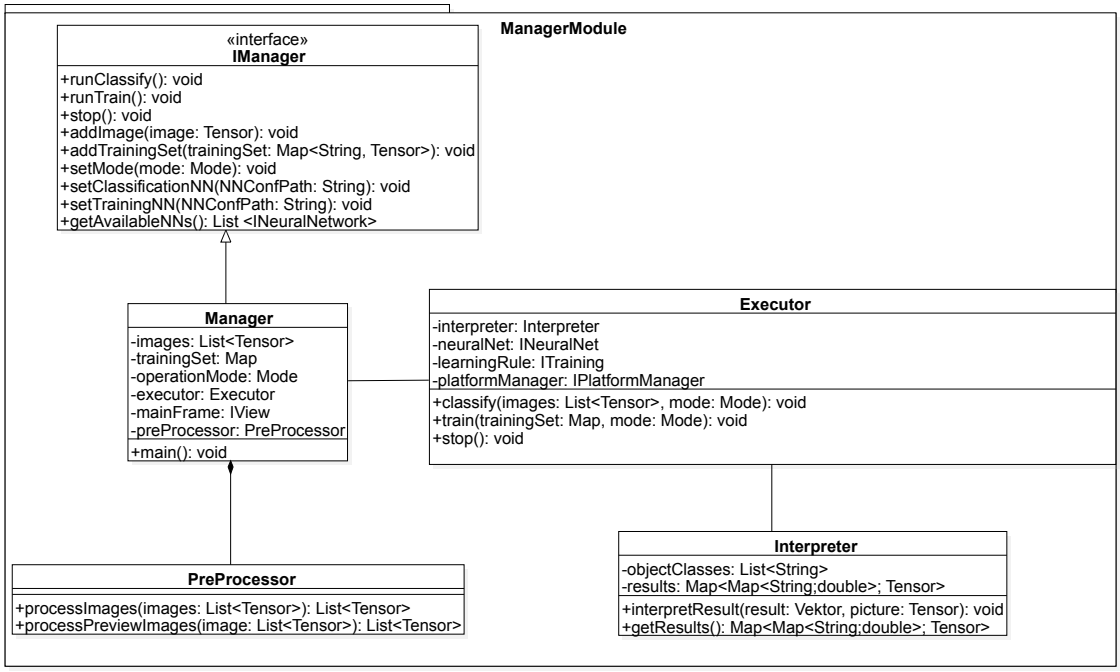

Abbildung 5: Klassendiagramm: Paket ManagerModule

4.1.4 Detaillierte Klassenbeschreibungen

○ **«interface» IManager**
Über diese Schnittstelle kann auf das ManagerModule zugegriffen werden

- ▷ **runClassify(): void**
  Mit dieser Methode wird die Klassifizierung der bereits hinzugefügten Bilder angestoßen.
  @return void

- ▷ **runTrain(): void**
  Mit dieser Methode wird das Trainieren des (möglicherweise standardmäßig) ausgewählten Neuronalen Netzes mit den bereits hinzugefügten Bildern angestoßen.
  @return void

- ▷ **stop(): void**
  Mit dieser Methode wird die Klassifizierung oder das Trainieren eines Neuronalen Netzes gestoppt.
  @return void

- ▷ **addImage(image: Tensor): void**
  Mit dieser Methode wird ein Bild zum Klassifizieren hinzugefügt.
  image: das Bild, das hinzugefügt werden soll.
  @return void

- ▷ **addTrainingSet(trainingSet: Map<String, Tensor>): void**
  Mit dieser Methode werden Bilder mit ihren Labels zum Klassifizieren hinzugefügt.
  trainingSet: eine Map von Bildern und ihren zugehörigen Labels, die hinzugefügt werden sollen.
  @return void

- ▷ **setMode(mode: Mode): void**
  Mit dieser Methode kann der Berechungsmodus festgelegt werden.
  mode: der festzulegende Berechnungsmodus.
  @return void

- ▷ **setNeuralNet(NNConfPath: String): void**
  Mit dieser Methode kann das für die Klassifizierung zu verwendende Neuronale Netz ausgewählt werden. Über eine nicht weiter beschriebene Methode der Klasse Executor wird das gewählte Neuronale Netz an Executor weitergeleitet und dort initialisiert.
  NNConfPath: der Pfad der Konfigurationsdatei des Neuronalen Netzes, das zur Klassifizierung verwendet werden soll.
  @return void

- ▷ **getAvailablePlatforms(): List <Platform>**
  Mit dieser Methode können die im Moment verfügbaren Plattformen abgefragt werden. Dies ist zum Beispiel bei entfernbaren USB-Compute-Sticks von Relevanz.
  @return eine Liste der verfügbaren Plattformen.

- ▷ **getAvailableNNs(): List <INeuralNetworks>**
  Mit dieser Methode können die verfügbaren Neuronalen Netze abgefragt werden. Dies sind sowohl für die Klassifizierung als auch für das Trainieren die selben Netze.
  @return eine Liste der verfügbaren Neuronalen Netze.

○ **Manager**
Die zentrale Controller-Klasse. Sie implementiert die Schnittstelle IManager. Mit der main()-Funktion ist sie der Ausgangspunkt des Programms.

- **«private» images: List<Tensor>**
  Die Bilder für die Klassifikation.

- **«private» trainingSet: Map<String, Tensor>**
  Die Bilder für das Training mit ihren Labels.

- **«private» operationMode: Mode**
  Der Berechnungsmodus (für Klassifkation und Training gleich).

- **«private» executor:Executor**
  Eine Instanz der Klasse Executor.

- **«private» mainFrame: IView**
  Eine Instanz der Klasse MainFrame, welche die Schnittstelle IView implementiert. Diese wird unter anderem dafür verwedet, die Schnittstelle sichtbar zu schalten und getOberservs() aufzurufen, um zentral die Subjekte an die Beobachter anheften (mit der Methode attach()) zu können.

- **«private» preProcessor: PreProcessor**
  Eine Instanz der Klasse PreProcessor.

▷ **main()**
Die main()-Methode des Programms. Sie initialisiert unter anderem das GUIModule und schält es mit der Methode start() der Schnittstelle IView auf sichtbar.

○ **Executor**
Executor ist dafür zuständig, die Ausführung der Klassifikation und des Trainings zu kontrollieren. Sie erbt von der Klasse Subject aus dem Paket GUIModule.

- **«private» interpreter: Interpreter**
  Eine Instanz der Klasse Interpreter.

- **«private» neuralNet: INeuralNet**
  Die Instanz des aktuell ausgewählten neuronalen Netzes.

- **«private» learningRule: ITraining**
  Eine Instanz einer Klasse, welche ITraining implementiert. Sie wird für den Training-Vorgang verwendet.

- **«private» platformManager: IPlatformManager**
  Eine Instanz einer Klasse, welche IPlatformManager implementiert.

▷ **classify(images: List<Tensor>, mode: Mode): void**
Diese Methode kontrolliert den Klassifikationsvorgang.
images: die Bilder, die klassifizeirt werden sollen.
mode: der Berechnungsmodus, in dem die Klassifikation stattfinden soll.
@return void

▷ **train(trainingSet: Map<String, Tensor>, mode: Mode): void**
Diese Methode kontrolliert den Trainingsvorgang.
trainingSet: die Bilder mit ihren Labels, mit denen trainiert werden soll.
mode: der Berechnungsmodus, in dem das Training stattfinden soll.
@return void

▷ **stop(): void**
Mit dieser Methode bricht der Manager das Training ab, nachdem er vom Event-Module dazu aufgerufen wurde.
@return void

○ **Interpreter**
Interpreter ist dafür zuständig, beim Klassifizieren den Ergebnisvektor zu interpretieren. Dafür ermittelt er die fünf wahrscheinlichsten Objektklassen und informiert danach in interpretResult() das GUIModule entsprechend des Observer-Patterns. Dafür erbt Interpreter von der Klasse Subject.

– **«private» objectClasses: List<String>**
Eine Liste der möglichen Objektklassen. Standardmäßig sind dies die 1000 Objektklassen von AlexNet. Das kann sich jedoch ändern, wenn ein anderes Neuronales Netz zur Klassifikation verwendet wird.

– **«private» results: Map<Map<String;double>; Tensor>**
Die Ergebnisse der Klassifizierungen. Pro Bild fünf Strings der Namen der fünf wahrscheinlichsten Objektklassen mit ihrer Wahrscheinlichkeit (als double).

▷ **interpretResult(result: Vektor picture: Tensor): void**
Diese Methode ermittelt anhand des Ergebnisvektors die fünf wahrscheinlichesten Objektklassen, speichert das Ergebnis und informiert die Beobachter der Klasse mit informObservers().
result: der Ergebnisvektor.
picture: das Bild, das klassifiziert wurde.
@return void

▷ **getResults(): Map<Map<String;double>; Tensor>**
Diese Methode wird von den Beobachtern aus dem GUIModule aufgerufen, um

die Klassifizierungsergebnisse anzuzeigen.
@return: die Klassifizierungsergebnisse. Pro Bild fünf Strings der Namen der fünf wahrscheinlichsten Objektklassen mit ihrer Wahrscheinlichkeit (als double).

4.2 EventModule

4.2.1 Funktion

Das Event-Modul (EventModule) führt die über das GUI vom Nutzer getätigten Aktionen und Änderungen aus, beziehungsweise stößt sie an.

4.2.2 Verwendung im Kontext

Das Event-Modul wird zunächst über die Schnittstelle **IEventsInvoker** von dem GUI-Modul aufgerufen und anschließend von der Klasse **EventInvoker**, die diese Schnittstelle implementiert, initialisiert. Bei einem Ereignis (z.B. Knopfdruck auf „Durchsuchen“) greift ein Panel (z.B. **InputPanel**), das im GUI-Modul beschrieben ist, auf die Schnittstelle **IEventsInvoker** zu, um den entsprechenden Event-Handler(z.B. **NewInputHandler**) aufzurufen.

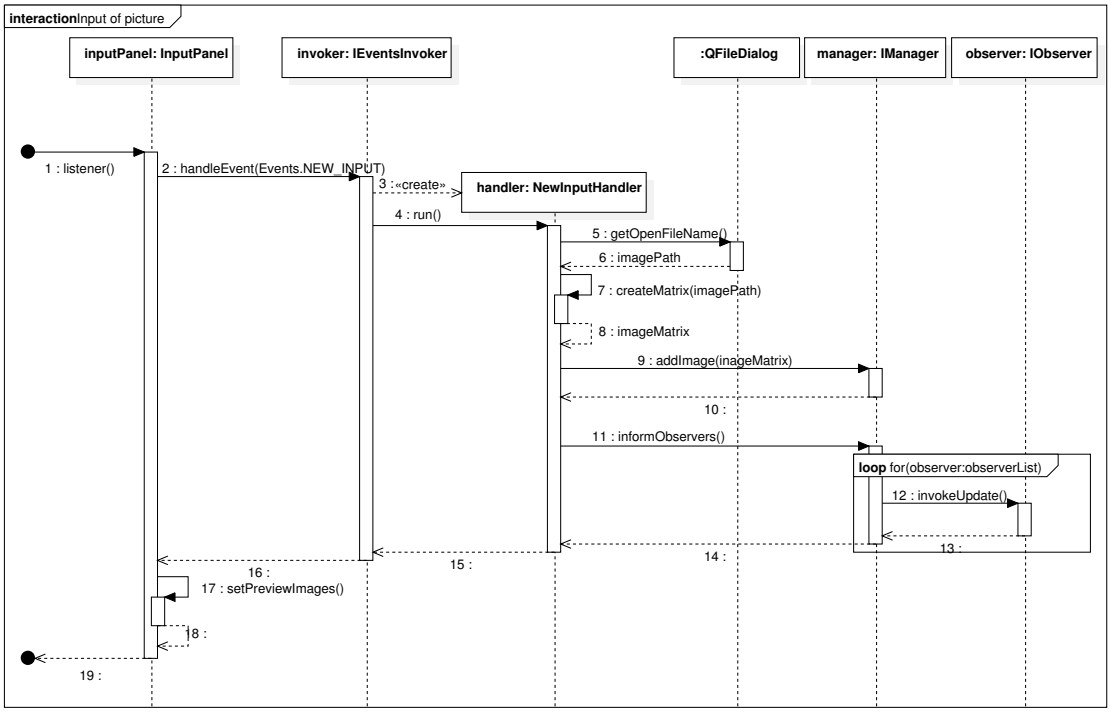

Abbildung 6: Sequenzdiagramm: Eingabe eines neuen Bildes

4.2.3 Aufbau

Alle möglichen Ereignisse sind in der Enum-Klasse **Events** aufgezählt. Sie werden in den konkreten Handlers (**NewInputHandler**, **RunHandler** etc.), die seinerseits von der abstrakten Klasse **Handler** erben, bearbeitet. Der **Handler** entspricht dem abstrakten Befehl und **NewInpuHandler**, **RunHandler** etc. den konkreten Befehlen gemäß dem Entwurfsmuster „Befehl“, das hier eingesetzt wird. Um mit dem Modul zu kommunizieren, muss auf die Schnittstelle **IEventsInvoker** zugegriffen werden. Für die Erstellung der GUI wird außerdem die Bibliothek Qt verwendet.

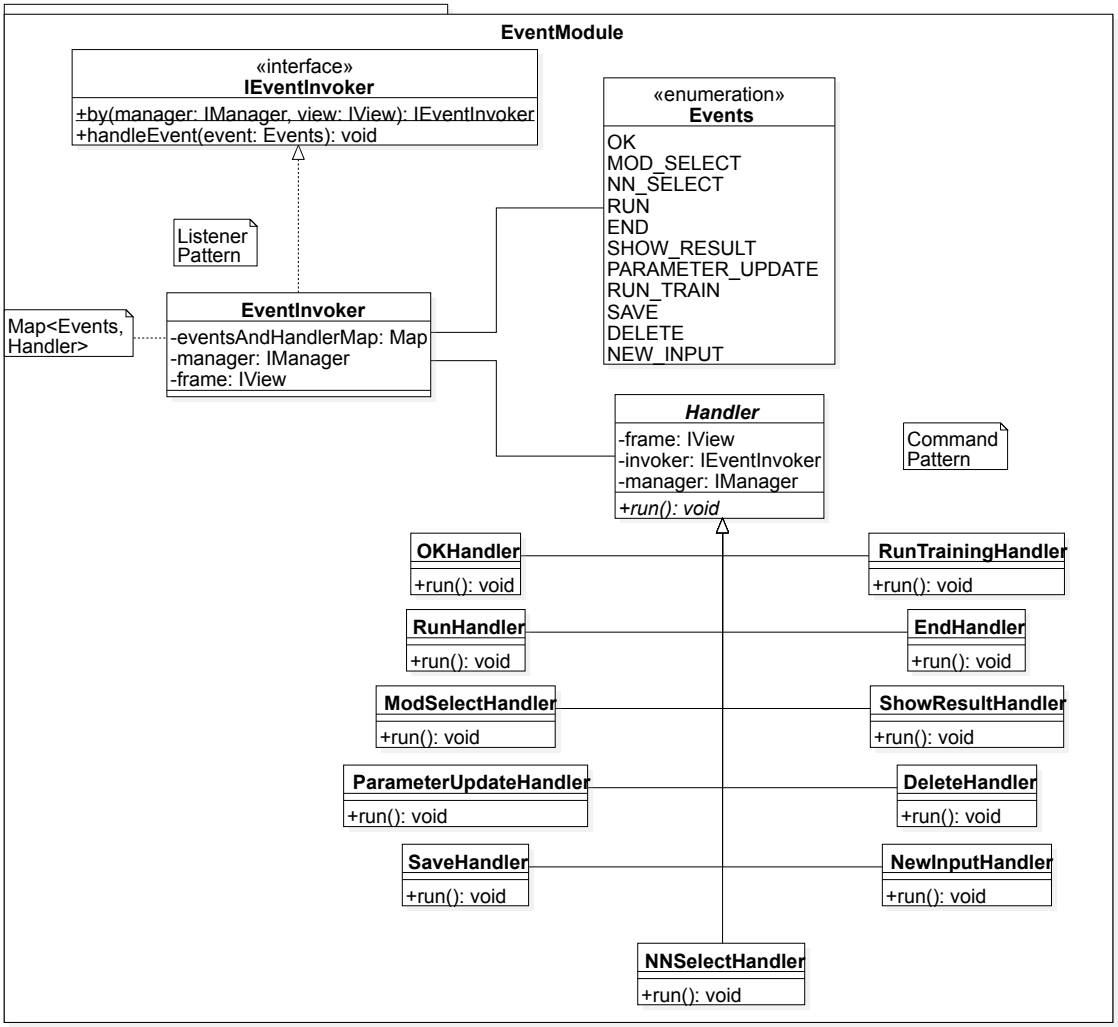

Abbildung 7: Klassendiagramm: Paket EventModule

4.2.4 Detaillierte Klassenbeschreibungen

○ **«interface» IEventInvoker**
Die Schnittstelle des Event-Moduls, über die die bestimmten Ereignisse(Events) aufgerufen werden können.

- ▷ **«static» by(manager: IManager, view: IView): IEventInvoker**
  Die statische Methode der Schnittstelle, die eine Instanz der implementierenden Klasse EventsInvoker zurückgibt.
  manager: eine Instanz einer Klasse, die IManager implementiert (in diesem Fall Manager), um Änderungen am Programm anzustoßen.
  view: eine Instanz einer Klasse, die IView implementiert (in diesem Fall MainFrame) um Änderungen an der Anzeige anzustoßen.
  @return eine Instanz der Klasse, die diese Schnittstelle implementiert.

- ▷ **handleEvent(event: Events): void**
  Die Methode, die ein bestimmtes Ereignis mithilfe von dynamischer Typisierung aufruft.
  event: ein bestimmtes Ereignis, das aufgerufen werden soll.
  @return void

○ **EventInvoker**
Die Klasse, die die Schnittstelle des Event-Moduls implementiert und den Aufrufer im Entwurfsmuster „Befehl“ repräsentiert.

- – **«private» eventsAndHandlerMap: Map<Events, Handler>**
  Das Attribut, das die Assoziation zwischen Events und Handlers bildet.

- ▷ **EventInvoker(manager: IManager, view: IView)**
  Der Konstruktor, der eine Instanz der Klasse EventInvoker erstellt.
  manager: eine Instanz einer Klasse, die IManager implementiert (in diesem Fall Manager), um Änderungen am Programm anzustoßen.
  view: eine Instanz einer Klasse, die IView implementiert (in diesem Fall MainFrame) um Änderungen an der Anzeige anzustoßen.

○ **«abstract» Handler**
Handler definiert eine abstrakte Klasse für die Bearbeitung der möglichen Anfragen, die jeweils einem bestimmten Ereignis entsprechen. Darüber hinaus repräsentiert Handler den abstrakten Befehl im Entwurfsmuster „Befehl“ und somit die Basis-Klasse für alle weiteren Handlers.

- ▷ **Handler(frame: IView, invoker: IEventInvoker, manager: IManager)**
  Der Konstruktor, der eine Instanz der Klasse Handler erstellt.
  frame: eine Instanz einer Klasse, die IView implementiert (in diesem Fall MainFrame) um Änderungen an der Anzeige anzustoßen.
  invoker: eine Instanz einer Klasse, die IEventInvoker implementiert (in diesem Fall EventInvoker) für die Bestimmung des konkreten Handlers.
  manager: eine Instanz einer Klasse, die IManager implementiert (in diesem Fall Manager), um Änderungen am Programm anzustoßen.

- ▷ **«abstract» run()**: void
  Die abstrakte Methode für die Ausführung des Handlers (gemäß dem Entwurfsmuster „Befehl“).
  @return void

○ **OKHandler**
Die Klasse OKHandler erbt von der abstrakten Klasse Handler und bearbeitet die Anfrage, die dem Ereignis OK entspricht und somit den konkreten Befehl im Entwurfsmuster „Befehl“ repräsentiert.

- ▷ **OKHandler(frame: IView, invoker: IEventInvoker, manager: IManager)**
  Der Konstruktor, der eine Instanz der Klasse OKHandler erstellt.
  frame: eine Instanz einer Klasse, die IView implementiert (in diesem Fall MainFrame) um Änderungen an der Anzeige anzustoßen.
  invoker: eine Instanz einer Klasse, die IEventInvoker implementiert (in diesem Fall EventInvoker) für die Bestimmung des konkreten Handlers.
  manager: eine Instanz einer Klasse, die IManager implementiert (in diesem Fall Manager), um Änderungen am Programm anzustoßen.

- ▷ **run():** void
  Die Methode implementiert die abstrakte Methode run() der Superklasse Handler und führt das Ereignis OK aus.
  @return void

○ **RunHandler**
Die Klasse RunHandler erbt von der abstrakten Klasse Handler und bearbeitet die Anfrage, die dem Ereignis RUN entspricht und somit den konkreten Befehl im Entwurfsmuster „Befehl“ repräsentiert.

- ▷ **RunHandler(frame: IView, invoker: IEventInvoker, manager: IManager)**
  Der Konstruktor, der eine Instanz der Klasse RunHandler erstellt.
  frame: eine Instanz einer Klasse, die IView implementiert (in diesem Fall MainFrame) um Änderungen an der Anzeige anzustoßen.

invoker: eine Instanz einer Klasse, die IEventInvoker implementiert (in diesem Fall EventInvoker) für die Bestimmung des konkreten Handlers.
manager: eine Instanz einer Klasse, die IManager implementiert (in diesem Fall Manager), um Änderungen am Programm anzustoßen.

- **run():** void die Methode implementiert die abstrakte Methode run() der Superklasse Handler und führt das Ereignis RUN aus.
  @return void

○ **ModSelectHandler**
Die Klasse ModSelectHandler erbt von der abstrakten Klasse Handler und bearbeitet die Anfrage, die dem Ereignis MOD_SELECT entspricht und somit den konkreten Befehl im Entwurfsmuster „Befehl“ repräsentiert.

- **ModSelectHandler(frame: IView, invoker: IEventInvoker, manager: IManager)**
  Der Konstruktor, der eine Instanz der Klasse ModSelectHandler erstellt.
  frame: eine Instanz einer Klasse, die IView implementiert (in diesem Fall MainFrame) um Änderungen an der Anzeige anzustoßen.
  invoker: eine Instanz einer Klasse, die IEventInvoker implementiert (in diesem Fall EventInvoker) für die Bestimmung des konkreten Handlers.
  manager: eine Instanz einer Klasse, die IManager implementiert (in diesem Fall Manager), um Änderungen am Programm anzustoßen.

- **run():** void
  die Methode implementiert die abstrakte Methode run() der Superklasse Handler und führt das Ereignis MOD_SELECT aus.
  @return void

○ **NNSelectHandler**
Die Klasse NNSelectHandler erbt von der abstrakten Klasse Handler und bearbeitet die Anfrage, die dem Ereignis NN_SELECT entspricht und somit den konkreten Befehl im Entwurfsmuster „Befehl“ repräsentiert.

- **NNSelectHandler(frame: IView, invoker: IEventInvoker, manager: IManager)**
  Der Konstruktor, der eine Instanz der Klasse NNSelectHandler erstellt.
  frame: eine Instanz einer Klasse, die IView implementiert (in diesem Fall MainFrame) um Änderungen an der Anzeige anzustoßen.
  invoker: eine Instanz einer Klasse, die IEventInvoker implementiert (in diesem Fall EventInvoker) für die Bestimmung des konkreten Handlers.
  manager: eine Instanz einer Klasse, die IManager implementiert (in diesem Fall Manager), um Änderungen am Programm anzustoßen.

▷ **run():** void
die Methode implementiert die abstrakte Methode run() der Superklasse Handler und führt das Ereignis NN_SELECT aus.
@return void

○ **ParameterUpdateHandler**
Die Klasse ParameterUpdateHandler erbt von der abstrakten Klasse Handler und bearbeitet die Anfrage, die dem Ereignis PARAMETER_UPDATE entspricht und somit den konkreten Befehl im Entwurfsmuster „Befehl“ repräsentiert.

▷ **NNSelectHandler(frame: IView, invoker: IEventInvoker, manager: IManager)**
Der Konstruktor, der eine Instanz der Klasse ParameterUpdateHandler erstellt.
frame: eine Instanz einer Klasse, die IView implementiert (in diesem Fall MainFrame) um Änderungen an der Anzeige anzustoßen.
invoker: eine Instanz einer Klasse, die IEventInvoker implementiert (in diesem Fall EventInvoker) für die Bestimmung des konkreten Handlers.
manager: eine Instanz einer Klasse, die IManager implementiert (in diesem Fall Manager), um Änderungen am Programm anzustoßen.

▷ **run():** void
die Methode implementiert die abstrakte Methode run() der Superklasse Handler und führt das Ereignis PARAMETER_UPDATE aus.
@return void

○ **SaveHandler**
Die Klasse SaveHandler erbt von der abstrakten Klasse Handler und bearbeitet die Anfrage, die dem Ereignis SAVE entspricht und somit den konkreten Befehl im Entwurfsmuster „Befehl“ repräsentiert.

▷ SaveHandler(frame: IView, invoker: IEventInvoker, manager: IManager)
Der Konstruktor, der eine Instanz der Klasse SaveHandler erstellt.
frame: eine Instanz einer Klasse, die IView implementiert (in diesem Fall MainFrame) um Änderungen an der Anzeige anzustoßen.
invoker: eine Instanz einer Klasse, die IEventInvoker implementiert (in diesem Fall EventInvoker) für die Bestimmung des konkreten Handlers.
manager: eine Instanz einer Klasse, die IManager implementiert (in diesem Fall Manager), um Änderungen am Programm anzustoßen.

▷ **run():** void
die Methode implementiert die abstrakte Methode run() der Superklasse Handler und führt das Ereignis SAVE aus.
@return void

○ **NewInputHandler**
Die Klasse NewInputHandler erbt von der abstrakten Klasse Handler und bearbeitet die Anfrage, die dem Ereignis NEW_INPUT entspricht und somit den konkreten Befehl im Entwurfsmuster „Befehl“ repräsentiert.

- ▷ **NewInputHandler(frame: IView, invoker: IEventInvoker, manager: IManager)**
  Der Konstruktor, der eine Instanz der Klasse NewInputHandler erstellt.
  frame: eine Instanz einer Klasse, die IView implementiert (in diesem Fall MainFrame) um Änderungen an der Anzeige anzustoßen.
  invoker: eine Instanz einer Klasse, die IEventInvoker implementiert (in diesem Fall EventInvoker) für die Bestimmung des konkreten Handlers.
  manager: eine Instanz einer Klasse, die IManager implementiert (in diesem Fall Manager), um Änderungen am Programm anzustoßen.

- ▷ **run():** void
  die Methode implementiert die abstrakte Methode run() der Superklasse Handler und führt das Ereignis NEW_INPUT aus.
  @return void

○ **DeleteHandler**
Die Klasse DeleteHandler erbt von der abstrakten Klasse Handler und bearbeitet die Anfrage, die dem Ereignis DELETE entspricht und somit den konkreten Befehl im Entwurfsmuster „Befehl“ repräsentiert.

- ▷ **DeleteHandler(frame: IView, invoker: IEventInvoker, manager: IManager)**
  Der Konstruktor, der eine Instanz der Klasse DeleteHandler erstellt.
  frame: eine Instanz einer Klasse, die IView implementiert (in diesem Fall MainFrame) um Änderungen an der Anzeige anzustoßen.
  invoker: eine Instanz einer Klasse, die IEventInvoker implementiert (in diesem Fall EventInvoker) für die Bestimmung des konkreten Handlers.
  manager: eine Instanz einer Klasse, die IManager implementiert (in diesem Fall Manager), um Änderungen am Programm anzustoßen.

- ▷ **run():** void
  die Methode implementiert die abstrakte Methode run() der Superklasse Handler und führt das Ereignis DELETE aus.
  @return void

○ **ShowResultHandler**
Die Klasse ShowResultHandler erbt von der abstrakten Klasse Handler und bearbeitet die Anfrage, die dem Ereignis SHOW_RESULT entspricht und somit den konkreten Befehl im Entwurfsmuster „Befehl“ repräsentiert.

- ▷ **ShowResultHandler(frame: IView, invoker: IEventInvoker, manager: IManager)**
  Der Konstruktor, der eine Instanz der Klasse ShowResultHandler erstellt.
  frame: eine Instanz einer Klasse, die IView implementiert (in diesem Fall MainFrame) um Änderungen an der Anzeige anzustoßen.
  invoker: eine Instanz einer Klasse, die IEventInvoker implementiert (in diesem Fall EventInvoker) für die Bestimmung des konkreten Handlers.
  manager: eine Instanz einer Klasse, die IManager implementiert (in diesem Fall Manager), um Änderungen am Programm anzustoßen.

- ▷ **run():** void die Methode implementiert die abstrakte Methode run() der Superklasse Handler und führt das Ereignis SHOW_RESULT aus.
  @return void

○ **EndHandler**
Die Klasse EndHandler erbt von der abstrakten Klasse Handler und bearbeitet die Anfrage, die dem Ereignis END entspricht und somit den konkreten Befehl im Entwurfsmuster „Befehl“ repräsentiert.

- ▷ **EndHandler(frame: IView, invoker: IEventInvoker, manager: IManager)**
  Der Konstruktor, der eine Instanz der Klasse EndHandler erstellt.
  frame: eine Instanz einer Klasse, die IView implementiert (in diesem Fall MainFrame) um Änderungen an der Anzeige anzustoßen.
  invoker: eine Instanz einer Klasse, die IEventInvoker implementiert (in diesem Fall EventInvoker) für die Bestimmung des konkreten Handlers.
  manager: eine Instanz einer Klasse, die IManager implementiert (in diesem Fall Manager), um Änderungen am Programm anzustoßen.

- ▷ **run():** void
  die Methode implementiert die abstrakte Methode run() der Superklasse Handler und führt das Ereignis END aus.
  @return void

○ **RunTrainingHandler**
Die Klasse RunTrainingHandler erbt von der abstrakten Klasse Handler und bearbeitet die Anfrage, die dem Ereignis RUN_TRAIN entspricht und somit den konkreten Befehl im Entwurfsmuster „Befehl“ repräsentiert.

- ▷ **RunTrainingHandler(frame: IView, invoker: IEventInvoker, manager: IManager)**
  Der Konstruktor, der eine Instanz der Klasse RunTrainingHandler erstellt.
  frame: eine Instanz einer Klasse, die IView implementiert (in diesem Fall MainFrame) um Änderungen an der Anzeige anzustoßen.
  invoker: eine Instanz einer Klasse, die IEventInvoker implementiert (in diesem

Fall EventInvoker) für die Bestimmung des konkreten Handlers.
manager: eine Instanz einer Klasse, die IManager implementiert (in diesem Fall Manager), um Änderungen am Programm anzustoßen.

- ▷ **run():** void
  die Methode implementiert die abstrakte Methode run() der Superklasse Handler und führt das Ereignis RUN_TRAIN aus.
  @return void

○ **«enumeration» Events**
Die Klasse Events zählt alle möglichen Ereignisse auf, die im Programm geschehen können.

- ▷ **OK**
  Das Button OK wird gedrückt.

- ▷ **MOD_SELECT**
  Ein neuer Modus wird ausgewählt.

- ▷ **NN_SELECT**
  Ein neues neuronales Netz wird ausgewählt.

- ▷ **RUN**
  Die Bilderklassifizierung wird gestartet.

- ▷ **END**
  Der Klassifizierungsprozess oder Trainingsprozess wird beendet.

- ▷ **SHOW_RESULT**
  Die Ergebnisse werden angezeigt.

- ▷ **PARAMETER_UPDATE**
  Die eingegebenen Parameter werden aktualisiert.

- ▷ **RUN_TRAIN**
  Der Trainingsprozess wird gestartet.

- ▷ **SAVE**
  Das trainierte neuronale Netz wird abgespeichert.

- ▷ **DELETE**
  Das trainierte neuronale Netz wird verworfen.

- **NEW_INPUT**
  Ein neues Bild/neue Bilder wird/werden hochgeladen.

4.3 GUIModule

4.3.1 Funktion

Das GUIModule stellt eine graphische Benutzeroberfläche zur Verfügung, über die ein Nutzer durch die visuellen Darstellungen mit dem Programm interagiert.

4.3.2 Verwendung im Kontext

Das GUI-Modul wird von der Klasse **Manager** des ManagerModuls initialisiert. Alle visuellen Interaktionen mit Grafiken (z.B. Knopfdruck) des GUI-Moduls werden mithilfe der entsprechenden Handlers des Event-Moduls registriert und bearbeitet. Darüber hinaus werden die erhaltenen Informationen weitergeleitet und die GUI wird entsprechend aktualisiert. Der Zugriff auf die Darstellung der GUI-Komponente erfolgt über die Schnittstelle **IView**.

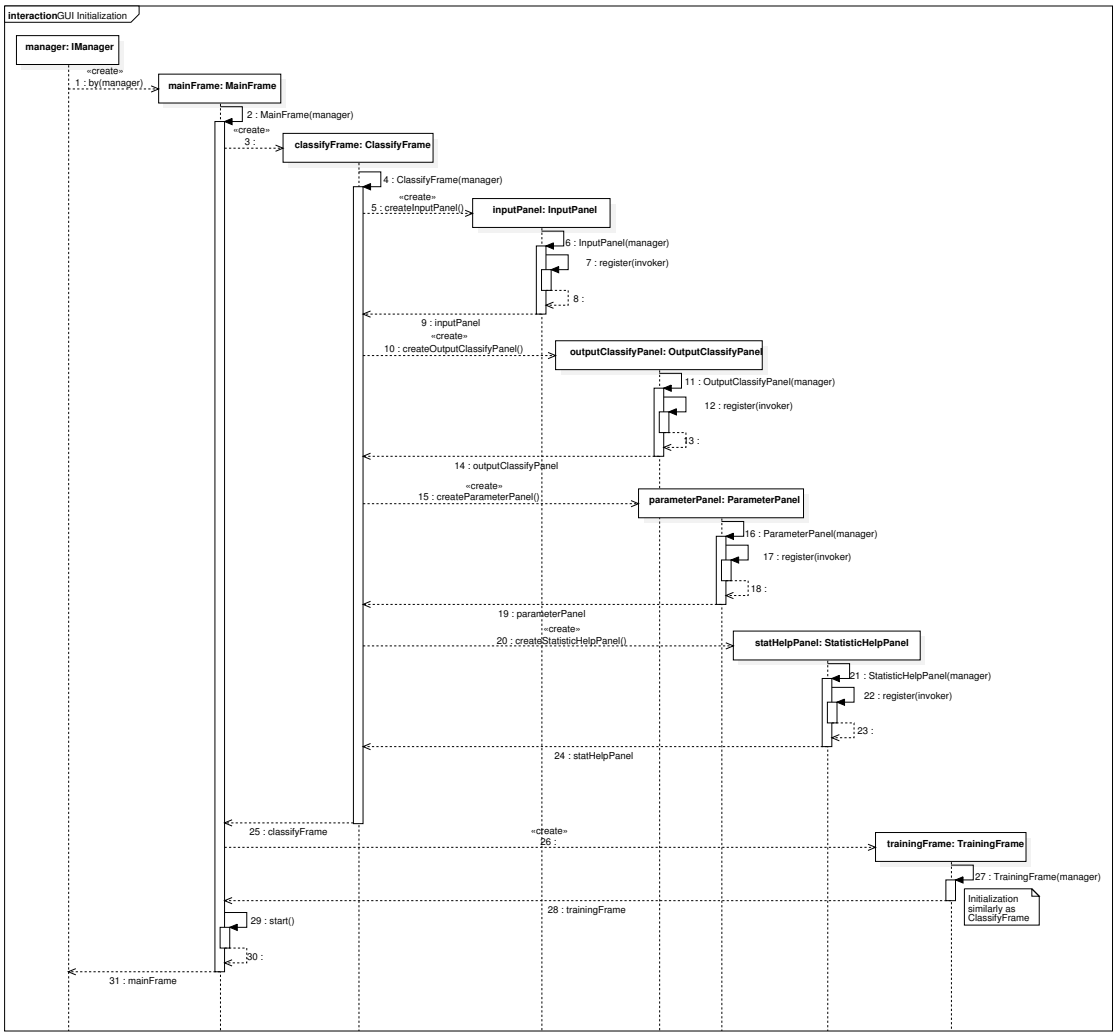

Abbildung 8: Sequenzdiagramm: Initialisierung der GUI

4.3.3 Aufbau

Das Hauptfenster, das sich beim Starten des Programms öffnet, wird in der Klasse **MainFrame**, die seinerseits die Schnittstelle **IView** implementiert, beschrieben. Das Hauptfenster besteht aus zwei Reitern (Tabs): **ClaffifyFrame** und **TrainingFrame**. Diese haben einen ähnlichen Aufbau. Zur Erfüllung des Kapselungsprinzips werden daher die sich wiederholenden Teile der Reiter herausgezogen und in den Klassen **InputPanel**, **ParameterPanel**, **StatisticHelpPanel**, **OutputClassifyPaneel** und **OutputTrainingPanel** beschrieben. Die Klasse **InputPanel** beschreibt die Visualisierung der Bildeingabe, **ParameterPanel** - der Parametereinstellungen, **StatisticHelpPanel** - der Knöpfe „Help“ und „Statistic“, **OutputClassifyPanel** - der Klassifizierungsergebnisse, **OutputTrainPanel** - der Statistik des Trainingsprozesses.
Außerdem implementieren die eben genannten Panels die Schnittstelle **IObserver**. Somit werden sie konkrete Beobachter gemäß dem Entwurfsmuster „Beobachter“. Die Subjekte sind die Klassen **Manager**, **Executor**, **Interpreter**, **Learning Rule** und **PlatformManager**.
Andere Bestandteile des GUI-Moduls, nämlich die Klassen **ErrorFrame**, **HelpFrame**, **StatisticFrame** und **SaveDeleteTrainingFrame** sind für die Darstellung der separat geöffneten Fenster Fehler-Fenster, Hilfe-Fenster, Statistik-Fenster, und das Fenster für die Speicherung oder Verwerfung des neu trainierten neuronalen Netzes zuständig.

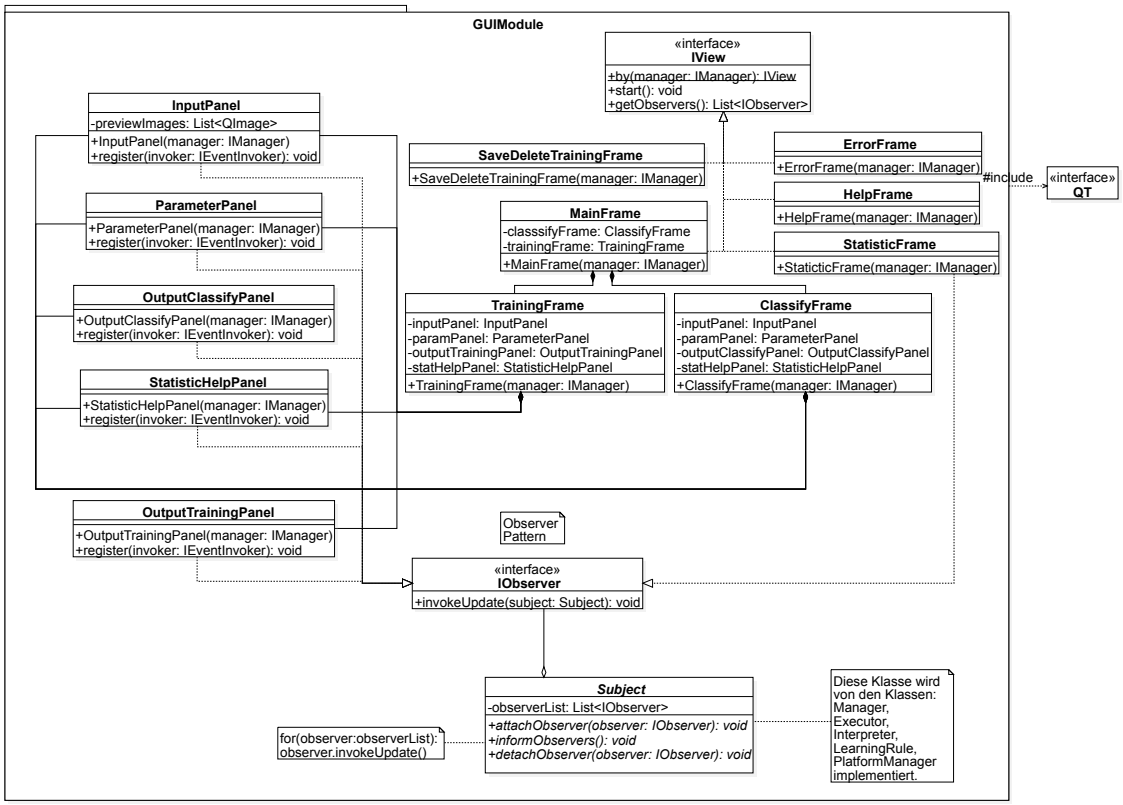

Abbildung 9: Klassendiagramm: Paket GUIModule

4.3.4 Detaillierte Klassenbeschreibungen

○ **«interface» IView**
Die Schnittstelle des GUI-Moduls, über welche die graphische Benutzeroberfläche der Software erstellt werden kann.

- ▷ **«static» by(manager: IManager): IView**
  Statische Methode der Schnittstelle, die eine Instanz der implementierten Klassen MainFrame, ClassifyFrame, TrainingFrame, StatisticFrame, SaveDeleteTrainingFrame, HelpFrame, ErrorFrame erstellt.
  manager: die Instanz der Klasse IManager für die Darstellung der erhaltenen Informationen über das ganze System.
  @return eine Instanz der Klasse, die diese Schnittstelle implementiert.

- ▷ **start(): void**
  Macht die GUI für den Nutzer sichtbar.
  @return void

- ▷ **getObservers(): List<IObserver>**
  Gibt die Liste von beobachteten Subjekten zurück.
  @return die Liste von beobachteten Subjekten.

○ **MainFrame**
MainFrame definiert eine Klasse für das Hauptfenster der Software.

- – **«private» classifyFrame: ClassifyFrame**
  Das Attribut der Klasse ClassifyFrame, das den Reiter Klassifizierung enthält.

- – **«private» trainingFrame: TrainingFrame**
  Das Attribut der Klasse TrainingFame, das den Reiter Transfer Learning enthält.

- ▷ **MainFrame(manager: IManager)**
  Der Konstruktor, der eine Instanz der Klasse MainFrame und damit zwei Bestandteile des Hauptfensters, nämlich den Klassifizierungs- und Transfer Learning Reiter erstellt.
  manager: die Instanz der Klasse IManager, die dargestellt wird.

○ **ClassifyFrame**
ClassifyFrame definiert eine Klasse für den Reiter (Tab) Klassifizierung des Hauptfensters.

- – **«private» inputPanel: InputPanel**

Das Attribut der Klasse InputPanel, das das Panel für die Bildeingabe darstellt.

– **«private» paramPanel: ParameterPanel**
Das Attribut der Klasse ParameterPanel, das das Panel für die Parametereinstellung des Klassifizierungsprozesses darstellt.

– **«private» outputClassifyPanel: outputClassifyPanel**
Das Attribut der Klasse OutputClassifyPanel, das die Klassifizierungsergebnisse darstellt.

– **«private» statHelpPanel: StatisticHelpPanel**
Das Attribut der Klasse StatisticHelpPanel, das die zwei Knöpfe Statistik und Hilfe im Klassifizierung-Tab darstellt.

▷ **ClassifyFrame(manager: IManager)**
Der Konstruktor, der eine Instanz der Klasse ClassifyFrame und damit vier Bestandteile des Klassifizierung-Tabs, nämlich Bildeingabe, Parametereinstellung, Klassifizierungsergebnisse und die Knöpfe Statistik und Hilfe erstellt.
manager: die Instanz der Klasse IManager, die dargestellt wird.

○ **TrainingFrame**
TrainingFrame definiert eine Klasse für den Reiter (Tab) Transfer Learning des Hauptfensters.

– **«private» inputPanel: InputPanel**
Das Attribut der Klasse InputPanel, das das Panel für die Bildeingabe darstellt.

– **«private» paramPanel: ParameterPanel**
Das Attribut der Klasse ParameterPanel, das das Panel für die Parametereinstellung des Trainingsprozesses darstellt.

– **«private» outputTrainingPanel: outputTrainingPanelPanel**
Das Attribut der Klasse OutputTrainingPanel, das die Statistik des Trainingsprozesses darstellt.

– **«private» statHelpPanel: StatisticHelpPanel**
Das Attribut der Klasse StatisticHelpPanel, das die zwei Knöpfe Statistik und Hilfe im Transfer Learning Tab darstellt.

▷ **TrainingFrame(manager: IManager)**
Der Konstruktor, der eine Instanz der Klasse TrainingFrame und damit vier Bestandteile des Transfer Learning Tabs, nämlich Bildeingabe, Parametereinstellung,

Statistik des Trainingsprozesses und die Knöpfe Statistik und Hilfe erstellt.
manager: die Instanz der Klasse IManager, die dargestellt wird.

○ **StatisticFrame**
StatisticFrame definiert eine Klasse für das Statistiken-Fenster.

▷ **StatisticFrame(manager: IManager)**
Der Konstruktor, der eine Instanz der Klasse StatisticFrame erstellt.
manager: die Instanz der Klasse IManager, die dargestellt wird.

○ **SaveDeleteTrainingFrame**
SaveDeleteTrainingFrame definiert eine Klasse für das Erfolg-Fenster.

▷ **SaveDeleteTrainingFrame(manager: IManager)**
Der Konstruktor, der eine Instanz der Klasse SaveDeleteTrainingFrame erstellt.
manager: die Instanz der Klasse IManager, die dargestellt wird.

○ **HelpFrame**
HelpFrame definiert eine Klasse für das Hilfe-Fenster.

▷ **HelpFrame(manager: IManager)**
Der Konstruktor, der eine Instanz der Klasse HelpFrame erstellt.
manager: die Instanz der Klasse IManager, die dargestellt wird.

○ **ErrorFrame**
ErrorFrame definiert eine Klasse für das Fehler-Fenster.

▷ **ErrorFrame(manager: IManager)**
Der Konstruktor, der eine Instanz der Klasse ErrorFrame erstellt.
manager: die Instanz der Klasse IManager, die dargestellt wird.

○ **«interface» IObserver**
Der IObserver definiert eine Aktualisierungsschnittstelle des Beobachter Entwurfsmusters.

▷ **invokeUpdate(subject: Subject): void**
Die Methode benachrichtigt über Änderungen des Subjektes.
subject: die Instanz, die geändert wird.
@return void

○ **InputPanel**
InputPanel definiert eine Klasse für den Bildeingabeteil des Klassifizierung- und Transfer Learning-Tabs. Dieses Panel wird mit einem hochgeladenen Bild/hochgeladenen

Bildern ausgefüllt.

- **«private» inputPreviewImages: List<QImage>**
  Das Attribut, das die Liste der Vorschaubilder enthält.

▷ **InputPanel(manager: IManager)**
Der Konstruktor, der eine Instanz der Klasse InputPanel erstellt.
manager: die Instanz der Klasse IManager, die dargestellt wird.

▷ **register(invoker: IEventInvoker): void**
Die Methode, die das InputPanel beim invoker registriert.
invoker: eine Instanz einer Klasse, die IEventInvoker implementiert (in diesem Fall EventInvoker) für die Bestimmung des konkreten Handlers.
@return void

○ **ParameterPanel**
ParameterPanel definiert eine Klasse für den Teil des Klassifizierung- und Transfer-Learning-Tabs, in dem der Nutzer den Betriebsmodus und das Neuronale Netz auswählen kann.

▷ **ParameterPanel(manager: IManager)**
Der Konstruktor, der eine Instanz der Klasse ParameterPanel erstellt.
manager: die Instanz der Klasse IManager, die dargestellt wird.

▷ **register(invoker: IEventInvoker): void**
Die Methode, die das ParameterPanel beim invoker registriert.
invoker: eine Instanz einer Klasse, die IEventInvoker implementiert (in diesem Fall EventInvoker) für die Bestimmung des konkreten Handlers.
@return void

○ **OutputClassifyPanel**
OutputPanel definiert eine Klasse für den Ausgabeteil des Klassifizierung-Tabs. Dieses Panel wird mit einem eingelesenen Bild/eingelesenen Bildern und deren jeweiligem Klassifizierungsergebnis ausgefüllt.

▷ **OutputClassifyPanel(manager: IManager)**
Der Konstruktor, der eine Instanz der Klasse OutputClassifyPanel erstellt.
manager: die Instanz der Klasse IManager, die dargestellt wird.

▷ **register(invoker: IEventInvoker): void**
Die Methode, die das OutputClassifyPanel beim invoker registriert.
invoker: eine Instanz einer Klasse, die IEventInvoker implementiert (in diesem Fall EventInvoker) für die Bestimmung des konkreten Handlers.

@return void

○ **StatisticHelpPanel**
StatisticHelpPanel definiert eine Klasse für den Teil des Klassifizierung- und Transfer-Learning-Tabs, in dem jeweils ein Button für Statistiken und einer für Hilfe dargestellt wird.

▷ **StatisticHelpPanel(manager: IManager)**
Der Konstruktor, der eine Instanz der Klasse StatisticHelpPanel erstellt.
manager: die Instanz der Klasse IManager, die dargestellt wird.

▷ **register(invoker: IEventInvoker): void**
Die Methode, die das StatisticHelpPanel beim invoker registriert.
invoker: eine Instanz einer Klasse, die IEventInvoker implementiert (in diesem Fall EventInvoker) für die Bestimmung des konkreten Handlers.
@return void

○ **OutputTrainingPanel**
OutputPanel definiert eine Klasse für den Ausgabeteil des Transfer-Learning-Tabs. Dieses Panel wird mit dem Berechnungsfortschritt (Informationen nach jeder abgeschlossenen Epoche über deren Verlust, Genauigkeit und Geschwindigkeit (in Bildern pro Sekunde) und die Verlustfunktion als Graph) ausgefüllt.

▷ **OutputTrainingPanel(manager: IManager)**
Der Konstruktor, der eine Instanz der Klasse OutputTrainingPanel erstellt.
manager: die Instanz der Klasse IManager, die dargestellt wird.

▷ **register(invoker: IEventInvoker): void**
Die Methode, die das OutputTrainingPanel beim invoker registriert.
invoker: eine Instanz einer Klasse, die IEventInvoker implementiert (in diesem Fall EventInvoker) für die Bestimmung des konkreten Handlers.
@return void

○ **«abstract» Subject**
Die abstrakte Klasse für das Befehls-Entwurfsmuster, die zur An- und Abmeldung von Beobachtern und zur Benachrichtigung von Beobachtern über Änderungen dient.

– **«private» observerList: List<IObserver>**
Liste von Beobachtern.

▷ **«abstract» attachObserver(observer: IObserver): void**
Fügt einen neuen Beobachter in die Liste der Beobachter ein.

observer: der hinzuzufügende Beobachter
@return void

▷ **«abstract» detachObserver(observer: IObserver): void**
Entfernt den Beobachter aus der Liste der Beobachter.
observer: der zu löschende Beobachter
@return void

▷ **«abstract» informObservers(): void**
Ruft alle angemeldeten Beobachter aus der Liste der Beobachter auf.
@return void

4.4 PlatformModule

4.4.1 Funktion

Das Plattform-Modul stellt die Funktionalität für die Verwaltung der Berechnungsaufgaben des Programms auf den heterogenen Plattformen bereit. Außerdem erfolgt über dieses Modul das Managing und die Sammlung der benötigten Statistiken jeder Plattform.

4.4.2 Verwendung im Kontext

Das Plattform-Modul wird von dem Executor-Modul über die Schnittstelle **IPlatformManager** initialisiert und weiterbenutzt, um verschiedene Berechnungen des Klassifizierungs- und Trainingsprozesses auf die heterogenen Plattformen ASIC, CPU und GPU zu verteilen.

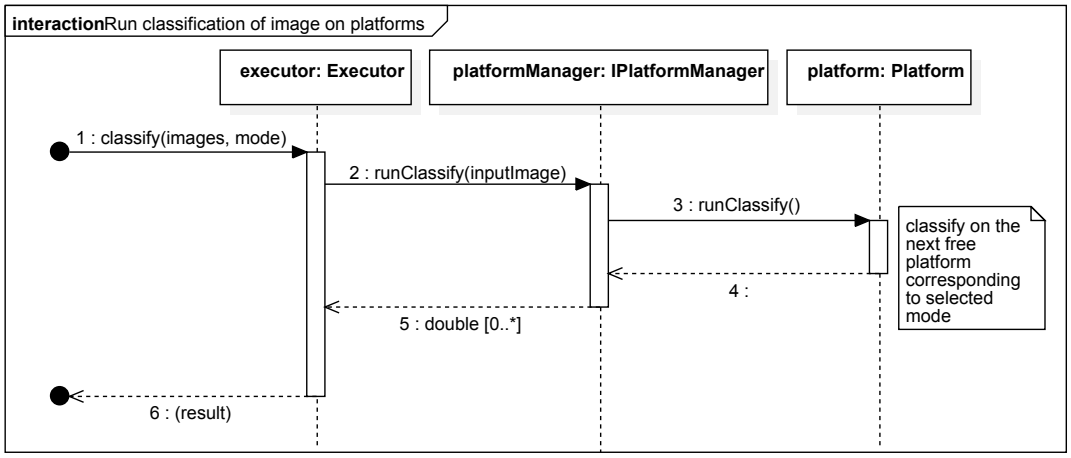

Abbildung 10: Sequenzdiagramm: Aufruf von Plattformen für die Klassifizierung eines Bildes

4.4.3 Aufbau

Es wird das Entwurfsmuster „Strategie“ angewendet. Die Klasse **PlatformManager** (Kontext), welche die Schnittstelle **IPlatformManager** implementiert, initialisiert alle Plattformen und bestimmt, welche davon verfügbar sind, um sie freizugeben. Die abstrakte Klasse **Platform** (Strategie) definiert den Klassifizierungsalgorithmus, der von den konkreten Plattformen **CPUPlatform**, **GPUPlatform** und **ASICPlatform** (konkrete Strategien) implementiert wird. Ferner wird im Modul das Entwurfsmuster „Dekorierer“ angewendet. Die abstrakte Klasse **PlatformTraining** definiert die zusätzlichen abstrakten Operationen, die von den erbenden Klassen **CPUPlatform** und **GPUPlatform** implementiert werden. Die Statistik der Berechnungen jeder Plattform lässt sich von der Klasse **PlatformStatistic** abrufen. Für die Implementierung des Klassifizirungsalgorithmus und Trainingsalgorithmus wird auf der CPU und der GPU die Bibliothek OpenCL verwendet. Auf dem ASIC wird, nur für den Klassifizierungsprozess, das OpenVINO-Toolkit benutzt.

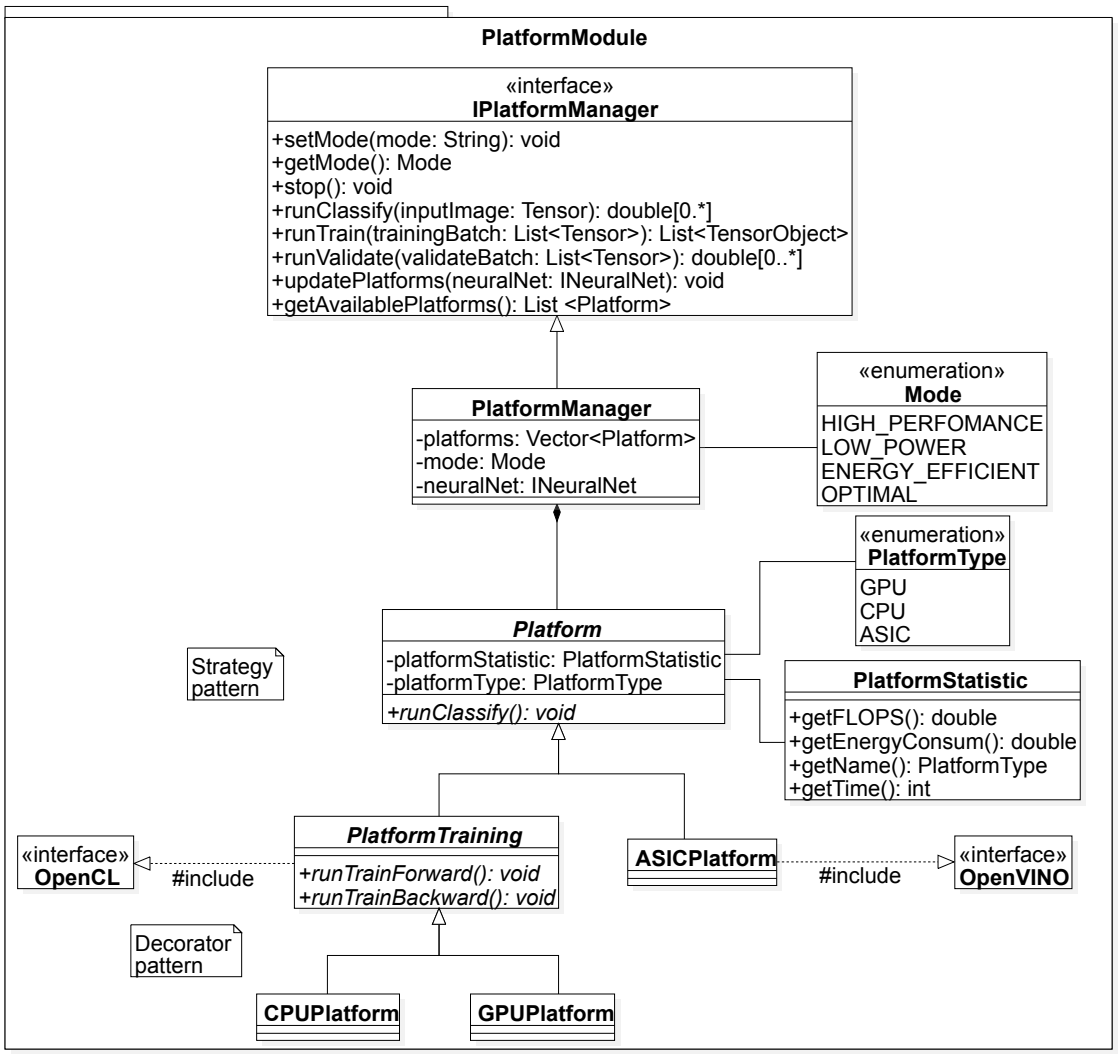

Abbildung 11: Klassendiagramm: Paket PlatformModule

4.4.4 Detaillierte Klassenbeschreibungen

○ **«interface» IPlatformManager**
Die Schnittstelle des Plattform-Moduls, über die verschiedene Berechnungen des Klassifizierung- und Trainingsprozesses auf heterogenen Plattformen verteilt werden können.

- ▷ **setMode(mode: Mode): void**
  Setzt den von dem Nutzer ausgewählten Betriebsmodus.
  mode: der zu setzende Berechnungsmodus (für Klassifizierung und Training gleich).
  @return void

- ▷ **getMode(): Mode**
  Gibt den von dem Nutzer ausgewählten Betriebsmodus zurück.
  @return den ausgewählten Modus.

- ▷ **stop(): void**
  Die Methode beendet den Klassifizierungs- oder Trainingsprozess.
  @return void

- ▷ **runClassify(inputImage:Tensor): double[0..*]**
  Startet die Klassifizierung des von dem Nutzer hochgeladenen Bildes. **inputImage:** das zu klassifizierende Bild.
  @return das Array, welches aus den errechneten Wahrscheinlichkeiten der Objektklassen besteht.

- ▷ **runTrain(trainingBatch: List<Tensor>): List<TensorObject>**
  Startet das Transfer Learning mit dem durch den Nutzer hochgeladenen Trainingsdatensatz.
  trainingBatch: der vom Nutzer hochgeladene Trainingsdatensatz.
  @return die errechneten Gewichte der trainierten Layer.

- ▷ **runValidate(validateBatch: List<Tensor>): void**
  Startet das Transfer Learning mit dem Validierung-Trainingssatz.
  validateBatch: das vom Nutzer hochgeladene Validierungsbatch.
  @return void

- ▷ **updatePlatforms(neuralNet: INeuralNet): void**
  Aktualisiert nach jedem Batch die Gewichte des neuronalen Netzes.
  neuralNet: das zu aktualisierende neuronale Netz.
  @return void

▷ **getAvailablePlatforms(): List <Platform>**
Die Methode gibt die Liste der verfügbaren Plattformen zurück.
@return die Liste der verfügbaren Plattformen.

○ **PlatformManager**
Die Klasse, welche die Schnittstelle des IPlatformManagers implementiert. PlatformManager enthält eine oder mehrere Plattformen. Diese können für den Klassifizierungs- und Trainingsprozess verwendet werden.

– **«private» platforms: Vector<Platform>**
Alle verfügbaren Plattformen.

– **«private» mode: Mode**
Der vom Nutzer ausgewählte Betriebsmodus.

▷ **PlatformManager(neuralNet: NeuralNetwork)**
Konstruktor, der eine Instanz der Klasse PlatformManager erstellt.
neuralNet: die Instanz der Klasse NeuralNetwork, die von dem Nutzer ausgewählt wird.

○ **«abstract» Platform**
Platform definiert eine abstrakte Klasse für den Klassifizierungsalgorithmus, welcher in den konkreten Plattformen CPU-Plattform, GPU-Plattform und ASIC-Plattform implementiert wird.

– **«private» platformStatistic: PlatformStatistic**
Statistiken über die Berechnungen jeder Plattform.

– **«private» platformType: PlatformType**
Der Name der Plattform.

▷ **«abstract» runClassify(): void**
Klassifiziert das von dem Nutzer hochgeladene Bild.
@return void

○ **«abstract» PlatformTraining**
PlatformTraining definiert eine abstrakte Klasse für den Transfer Learning Algorithmus, der in den konkreten Plattformen CPU-Plattform, GPU-Plattform implementiert wird.

▷ **«abstract» runTrainForward(): void**
Führt den Forward-Teil des Transfer Learning Prozesses mit dem von dem Nutzer

hochgeladenen Trainingsdatensatz aus.
@return void

- ▷ **«abstract» runTrainBackward(): void**
  Führt den Backward-Teil des Transfer Learning Prozesses mit dem von dem Nutzer hochgeladenen Trainingssatz aus.
  @return void

○ **ASICPlatform**
Definiert eine Klasse für die Plattform Intel Movidius Neural Compute Stick.

- ▷ **runClassify(): void**
  Die Methode implementiert die abstrakte Methode der Superklasse und klassifiziert das von dem Nutzer hochgeladene Bild in Bezug auf ASIC Architektur.
  @return void

○ **CPUPlatform**
CPUPlatform definiert eine Klasse für die Plattform Hauptprozessor.

- ▷ **runClassify(): void**
  Implementiert die abstrakte Methode der Superklasse und klassifiziert das von dem Nutzer hochgeladene Bild in Bezug auf die CPU Architektur.
  @return void

- ▷ **runTrainForward(): void**
  Implementiert die abstrakte Methode der Superklasse und führt den Forward-Teil des Transfer Learning Prozesses mit dem von dem Nutzer hochgeladenen Trainingssatz in Bezug auf die CPU Architektur aus.
  @return void

- ▷ **runTrainBackward(): void**
  Implementiert die abstrakte Methode der Superklasse und führt den Backward-Teil des Transfer Learning Prozesses mit dem von dem Nutzer hochgeladenen Trainingssatz in Bezug auf die CPU Architektur aus.
  @return void

○ **GPUPlatform**
GPUPlatform definiert eine Klasse für die Plattform Grafikprozessor.

- ▷ **runClassify(): void**
  Implementiert die abstrakte Methode der Superklasse und klassifiziert das von

dem Nutzer hochgeladene Bild in Bezug auf die GPU Architektur.
@return void

▷ **runTrainForward(): void**
Implementiert die abstrakte Methode der Superklasse und führt den Forward-Teil des Transfer Learning Prozesses mit dem von dem Nutzer hochgeladenen Trainingssatz in Bezug auf die GPU Architektur aus.
@return void

▷ **runTrainBackward(): void**
Implementiert die abstrakte Methode der Superklasse und führt den Backward-Teil des Transfer Learning Prozesses mit dem von dem Nutzer hochgeladenen Trainingssatz in Bezug auf die GPU Architektur aus.
@return void

○ **PlatformStatistic**
Sammelt die Informationen über die Berechnungen jeder Plattform, um sie bei Anfrage zurückzugeben

▷ **getFLOPS(): double**
Gibt die Anzahl von Floating Point Operations Per Second (FLOPS) einer bestimmten Plattform zurück.
@return Anzahl von FLOPS der Plattform.

▷ **getEnergyConsum(): double**
Gibt den Energieverbrauch einer bestimmten Plattform zurück.
@return Energieverbrauchseinheiten der Plattform.

▷ **getName(): PlatformType**
Gibt den Namen der Plattform zurück, bei der die Statistiken gesammelt werden.
@return Name der Plattform.

▷ **getTime(): int**
Gibt den Zeitverbrauch für die Berechnungen der Plattform zurück.
@return Zeitverbrauch in Millisekunden.

○ **«enumeration» PlatformType**
In der Enumeration PlattformType werden alle heterogenen Plattformen aufgezählt.

▷ **GPU**
Graphics processing unit oder Grafikprozessor.

- ▷ **CPU**
  Central processing unit oder Hauptprozessor.

- ▷ **ASIC**
  Intel Movidius Neural Compute Stick.

○ **«enumeration» Mode**
In der Klasse Mode werden alle Betriebsmodi aufgezählt.

- ▷ **HIGH_PERFOMANCE**
  Hohe Perfomance: der schnellste und performanteste Modus, in dem alle verfügbaren heterogenen Plattformen benutzt werden.

- ▷ **LOW_POWER**
  Geringer Leistungsverbrauch: der Modus, in dem genau ein Intel Movidius Neural Compute Stick genutzt wird.

- ▷ **ENERGY_EFFICIENT**
  Hohe Energieeffizienz : der Modus, in dem alle verfügbaren Intel Movidius Neural Compute Sticks benutzt werden.

- ▷ **OPTIMAL**
  Optimal: der optimale Modus, in dem die GPU und alle Intel Movidius Neural Compute Sticks genutzt werden.

4.5 ImageEditorModule

4.5.1 Funktion

Das ImageEditorModule stellt notwendige Bildbearbeitungsoperationen zur Verfügung. Neben einfachen Operationen wie dem Ausschneiden von Teilbildern oder dem Rotieren von Bildern beherrscht das Paket auch erweiterte Operationen wie die Kanaleranpassung von monochromen Bildern.

4.5.2 Verwendung im Kontext

Die Methode editImage() aus dem Interface **IImageEditor** wird von der Klasse **IImageEditor** implementiert und von der Klasse **PreProcessor** aus dem ManagerModule aufgerufen, um die geladenen Bildmatrizen für die weitere Verarbeitung im neuronalen Netz vorzubereiten. Des Weiteren wird diese Methode auch von den einzelnen DataAugmentationTechniques aus dem Modul DataAugmentationModule aufgerufen, um den Trainingsdatensatz zu erweitern. Die Klasse **ImageEditor** welche dieses Interface implementiert, initiiert anschließend die jeweilige Bildbearbeitungsoperation.

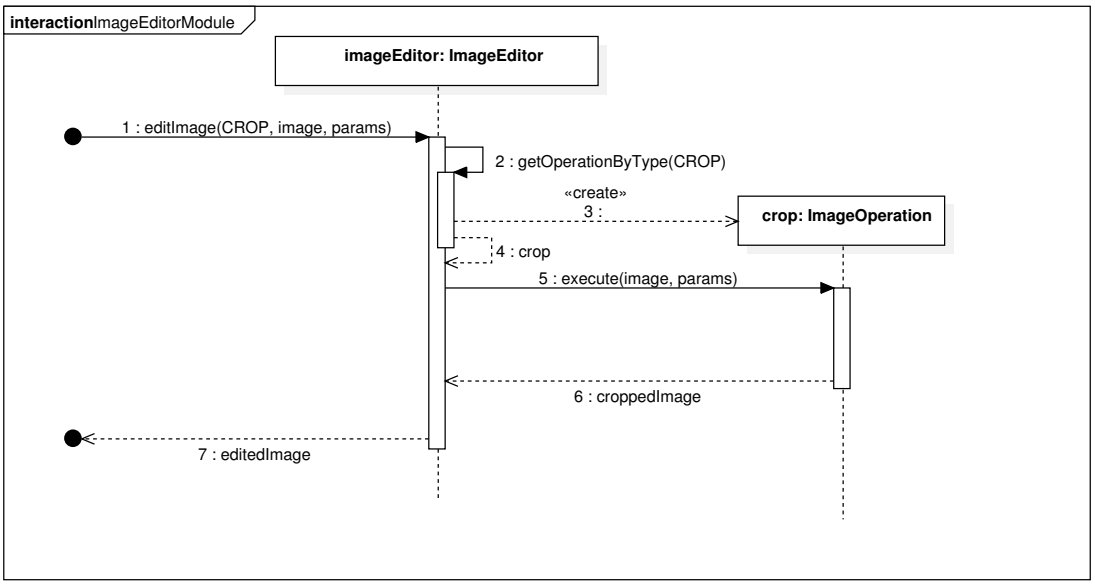

Abbildung 12: Sequenzdiagramm: Paket ImageEditorModule

4.5.3 Aufbau

Es wird das Strategie Entwurfsmuster angewendet. Die Klasse **ImageEditor**, welche die Methode editImage() des Interfaces **IImageEditor** implementiert dient hierbei als sogenannte Kontextklasse. Diese Methode wird von außerhalb, einem sogenannten Klienten aufgerufen. In der Kontextklasse wird das gewünschte **ImageOperation** Objekt als Attribut gesetzt und anschließend die Ausführung der Bildbearbeitung über die executeOperation() Methode initiiert. Infolgedessen wird die execute() Methode des soeben gesetzen **ImageOperation** Objekts aufgerufen. Die Klasse **ImageOperation** ist eine abstrakte Klasse, von der die jeweiligen Bildbearbeitungsoperationen in Form von Klassen erben.

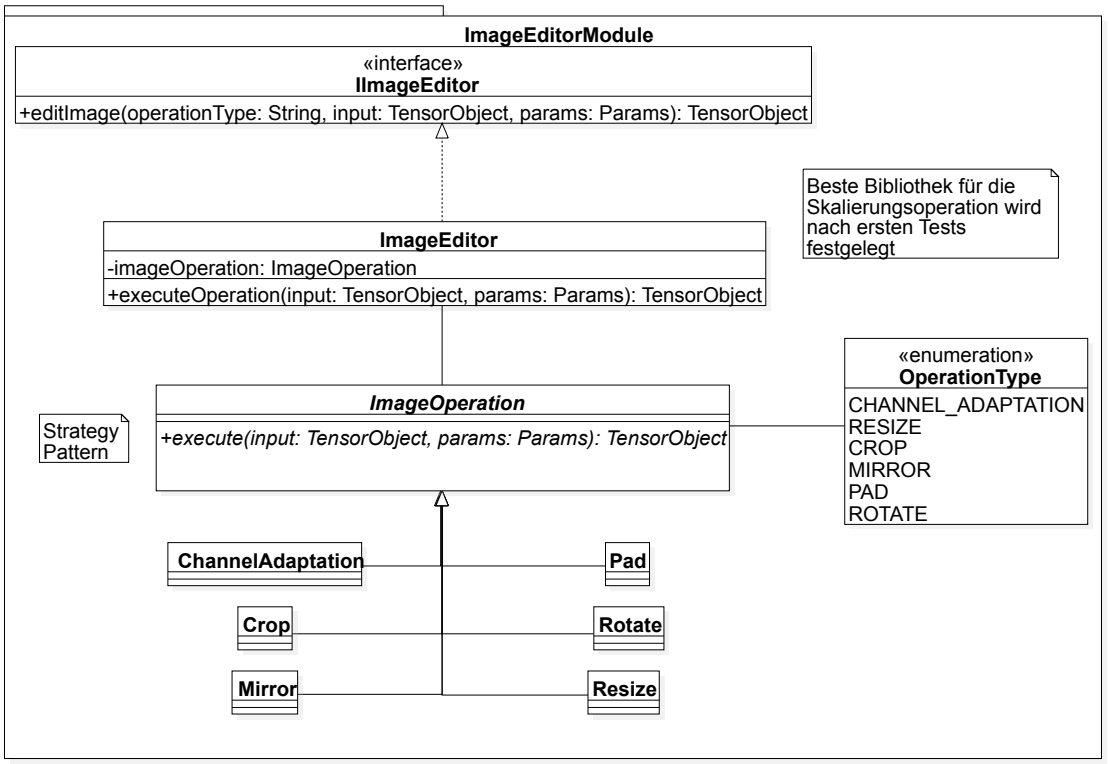

Abbildung 13: Klassendiagramm: Paket ImageEditorModule

4.5.4 Detaillierte Klassenbeschreibungen

○ **«interface» IImageEditor**
Schnittstelle des ImageEditorModuls welche die Methodendeklaration der Methode edi-

tImage() enthält.

- ▷ **editImage(operation: OperationType, input: TensorObject, params: Params): TensorObject**
  Bereitet die Bildbearbeitungsoperation vor und leitet die Bearbeitung in die Wege.
  operation: Rückgabeobjekt aus OperationType Enumeration.
  input: Bildmatrix in Form eines TensorObject Objekts.
  params: enthält sämtliche notwendigen Parameter für die jeweilige Operation.
  @return void

○ **ImageEditor**
ImageEditor initiiert den Bildbearbeitungsvorgang.

- – **«private» imageOperation: ImageOperation**
  Unsere gewünschte Operation als privates Attribut vom Typ ImageOperation.

- ▷ **executeOperation(input: TensorObject, params: Params): TensorObject**
  Diese Methode ruft die execute() Methode der durch den ImageEditor gesetzen ImageOperation auf.
  input: Bildmatrix in Form eines TensorObject Objekts.
  params: enthält sämtliche notwendigen Parameter für die Bildoperation.
  @return bearbeitete Bildmatrix in Form eines TensorObject Objekts.

- ▷ **getOperationByType(operation: OperationType): ImageOperation**
  Getter Methode welche uns die jeweilige ImageOperation anhand des OperationType zurück gibt.
  operation: OperationType
  @return ImageOperation

○ **«abstract» ImageOperation**
Abstrakte Klasse für unsere ImageOperation Objekte.

- – **«private» type: OperationType**
  Unsere gewünschter OperationType als privates Attribut.

- ▷ **execute(input: TensorObject, params: Params): TensorObject**
  Diese Methode enthält die gesamte Logik der jeweiligen Bildbearbeitungsoperation aus und wird von ihr auch implementiert. **input:** Bildmatrix in Form eines TensorObject Objekts.
  params: enthält sämtliche notwendigen Parameter für die Bildoperation.
  @return bearbeitete Bildmatrix in Form eines TensorObject Objekts.

○ **ChannelAdaptation**

Klasse für die Kanalanpassung von monochromen Bildern.

- ▷ **execute(input: TensorObject, params: Params): TensorObject**
  Führt die Kanalanpassung des übergebenen monochromen Bildes durch, indem der einzelne Bildkanal verdreifacht wird. Das resultierende Bild wird als TensorObject Objekt zurück gegeben.
  input: siehe Klasse «abstract» ImageOperation.
  params: ????
  @return kanalangepasste Bildmatrix in Form eines TensorObject Objekts.

○ **Pad**
Klasse für die Randerweiterung von Bildern.

- ▷ **execute(input: TensorObject, params: Params): TensorObject**
  Erweitert den Rand des übergebenen Bildes unter Berücksichtigung der ebenfalls übergebenen Parameter. Gibt das resultierende Bild als TensorObject Objekt zurück.
  input: siehe Klasse «abstract» ImageOperation.
  params: Maßangabe für den zu erweiternden Rand für alle vier Ränder.
  @return um Rand erweiterte Bildmatrix in Form eines TensorObject Objekts.

○ **Crop**
Klasse für das Ausschneiden eines Bildausschnitts.

- ▷ **execute(input: TensorObject, params: Params): TensorObject**
  Schneidet aus dem übergebenen Bild unter Berücksichtigung der übergebenen Parameter ein Bildausschnitt aus. Das ausgeschnittene Bild wird als TensorObject Objekt zurückgegeben.
  input: siehe Klasse «abstract» ImageOperation.
  params: Angabe des Bildausschnittes welcher ausgeschnitten werden soll.
  @return ausgeschnittene Bildmatrix in Form eines TensorObject Objekts.

○ **Rotate**
Klasse für die Rotation von Bildern.

- ▷ **execute(input: TensorObject, params: Params): TensorObject**
  Rotiert das übergebene Bild um die ebenfalls übergebene Gradangabe. Das resultierende Bild wird als TensorObject Objekt zurückgegeben.
  input: siehe Klasse «abstract» ImageOperation.
  params: Angabe um wie viel Grad das Bild gedreht werden soll.
  @return rotierte Bildmatrix in Form eines TensorObject Objekts.

○ **Mirror**

Klasse für die Spiegelung von Bildern.

- ▷ **execute(input: TensorObject, params: Params): TensorObject**
  Spiegelt das übergebene Bild an der gewünschten Achse und gibt das gespiegelte Bild als TensorObject Objekt zurück.
  input: siehe Klasse «abstract» ImageOperation.
  params: Achse um die das Bild gespiegelt werden soll.
  @return gespiegelte Bildmatrix in Form eines TensorObject Objekts.

○ **Resize**
Klasse für die Anpassung der Größe von Bildern.

- ▷ **execute(input: TensorObject, params: Params): TensorObject**
  Nutzt eine Bildbearbeitungsbibliothek um die Skalierung des übergebenen Bildes durchzuführen.
  input: siehe Klasse «abstract» ImageOperation.
  params: gewünschte Bildgröße auf die skaliert werden soll.
  @return skaliertes Bild als Bildmatrix in Form eines TensorObject Objekts.

○ **«enumeration» OperationType**
Enumeration aller vorhandenen Typen von Operationen.

- CHANNEL_ADAPTATION
- RESIZE
- CROP
- MIRROR
- PAD
- ROTATE

4.6 MathModule

4.6.1 Funktion

Das MathModule enthält die Utility-Klasse **Math**, welche eine Methode für die reLU Aktivierungsfunktion sowie eine gradient descent Methode enthält. Diese Operationen werden beim Klassifizierungs- bzw. Trainingsprozess benötigt. Da es sich hierbei um eine Utility-Klasse handelt sind sämtliche Methoden statisch. Des Weiteren werden hier die grundlegenden mathematischen Tensorobjekte modelliert und definiert.

4.6.2 Verwendung im Kontext

Die reLU Methode wird während des Klassifizierungsvorgangs nach jedem **Conv2DLayer** sowie nach jedem **FCLayer** aufgerufen und modifiziert den Output des Layers. Der modifizierte Output wird anschließend durch den Aufrufer an das nächste Layer übergeben. Die gradient descent Methode wird im Trainingsprozess im Rahmen von backpropagate benötigt.

4.6.3 Aufbau

Das Modul besteht aus der Utility-Klasse **Math** sowie einer abstrakten **TensorObject** Klasse welche von verschiedenen mathematischen Objekten in Form von Klassen geerbt werden.

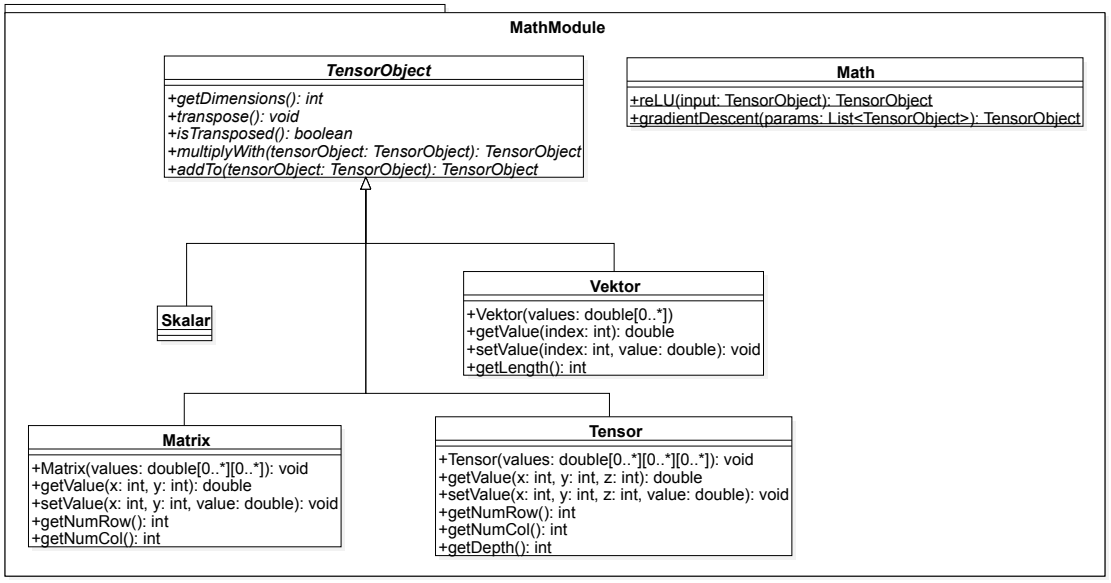

Abbildung 14: Klassendiagramm: Paket MathModule

4.6.4 Detaillierte Klassenbeschreibungen

○ **Math**
Utility-Klasse welche statische Mathematik Methoden enthält.

- ▷ **«static» reLU(input: TensorObject): TensorObject**
  Verarbeitet den Output eines Conv.- bzw. FC-Layers mittels der reLU-Funktion und gibt diesen als TensorObject Objekt zurück.
  input: Output eines Conv.- bzw. FC-Layers.
  @return Der durch die reLU-Funktion bearbeitete input in Form eines TensorObject Objekt.

- ▷ **«static» gradientDescent(params: Parameter[0..*]): TensorObject**
  Wendet das Gradientenverfahren an. **params:** Array welches sämtliche notwendigen Parameter enthält.
  @return Das Ergebnis in Form eines TensorObject Objekts.

○ **«abstract» TensorObject**
Abstrakte Klasse mit welcher ein allgemeines Tensorobjekt definiert wird. Die jeweiligen algebraischen Strukturen erben in Form von Klassen von dieser abstrakten Klasse.

- ▷ **«abstract» getDimensions(): int**
  Da auch allgemeine TensorObject Objekte als Rückgabewert zulässig sind, ist es von Vorteil die Dimension des vorliegenden TensorObject Objekts abfragen zu können.
  @return Anzahl von Dimensionen.

- ▷ **«abstract» transpose(): void**
  Alle Tensorobjekte sollen transponiert werden können.
  @return void

- ▷ **«abstract» isTransposed(): Boolean**
  Die algebraischen Strukturen sollen speichern ob sie im transponierten Zustand vorliegen.
  @return ob ein TensorObjekt transponiert ist.

- ▷ **«abstract» multiplyWith(tensorObject: TensorObject): TensorObject**
  Da algebraische Strukturen auf unterschiedlichste Weisen miteinander multipliziert werden können, sollen die jeweiligen Strukturen selber notwendige Multiplikationen implementieren.
  tensorObject Objekt vom Typ TensorObject mit dem multipliziert werden soll.
  @return Das resultierende Ergebnis vom Typ TensorObject.

- ▷ **«abstract» addTo(tensorObject: TensorObject): TensorObject**
  Algebraische Strukturen können auch addiert werden, weshalb die Strukturen die notwendigen Additionen implementieren müssen.
  tensorObject Objekt vom Typ TensorObject, welches addiert werden soll.
  @return Summe vom Typ TensorObject

○ **Skalar**
Diese Klasse wurde zur Veranschaulichung in den Entwurf übernommen. Sie erbt von der abstrakten Klasse TensorObject. In der Implementierung wird diese Klasse vermutlich nicht implementiert, da auf die primitiven Datentypen zurückgegriffen wird.

○ **Vektor**
Vektorklasse in welcher Vektoren als double Arrays dargestellt werden.

- ▷ **Vektor(values: double[0..*]): void**
  Konstruktor um einen Vektor zu instanziieren. Als Parameter wird ein double Array erwartet.
  input: der eingegebene Vektor.
  @return void

- ▷ **getValue(index: int): double**

Gibt für einen bestimmten Index den Wert des Vektorelements zurück.
index: Position des gewünschten Vektorelements vom Typ int.
@return der Wert des Vektorelements.

- ▷ **setValue(index: int, value: double): void**
  Setzt den Wert des Vektorelements an einem bestimmten Index.
  index: Position des zu setzenden Vektorelements vom Typ int.
  value: der Wert, der gesetzt werden soll.
  @return void

- ▷ **getLength(): int**
  Gibt die Länge des Vektors zurück. **@return** die Länge des Vektors.

○ **Matrix**
Matrixklasse in welcher Matrizen als zweidimensionale double Arrays dargestellt werden.

- ▷ **Matrix(values: double[0..*][0..*]): void**
  Konstruktor um eine Matrix zu instanziieren. Als Parameter wird ein zweidimensionales double Array erwartet.
  input: die eingegebene zweidimensionale Matrix.
  @return void

- ▷ **getValue(x: int, y:int): double**
  Gibt den Wert des Matrixelements an Position (x,y) zurück. **x:** Zeilennummer.
  y: Spaltennummer.
  @return der Wert des Matrixelements.

- ▷ **setValue(x: int, y: int, value: double): void**
  Setzt den Wert des Matrixelements an Position (x,y)
  x: Zeilennummer.
  y: Spaltennummer.
  value: Der Wert der gesetzt werden soll vom Typ double.
  @return void

- ▷ **getNumRow(): int**
  Gibt die Anzahl der Zeilen der Matrix zurück.
  @return Anzahl der Zeilen.

- ▷ **getNumCol(): int**
  Gibt die Anzahl der Spalten der Matrix zurück.
  @return Anzahl der Spalten.

○ **Tensor**
Die Klasse Tensor repräsentiert bei uns ein Tensorobjekt dritten Grades. Es wird als dreidimensionales double Array dargestellt.

▷ **Matrix(values: double[0..*][0..*][0..*]): void**
Konstruktor um ein Tensor zu instanziieren. Als Parameter wird ein dreidimensionales double Array erwartet.
input: die dreidimensionale Matrix in der Form eines Arrays.
@return void

▷ **getValue(x: int, y:int, z: int): double**
Gibt den Wert des Tensorelements an Position (x,y,z) zurück.
x: Zeilennummer.
y: Spaltennummer.
z: Tiefe.
@return der Wert der Matrix an der eingegeben Position.

▷ **setValue(x: int, y: int, z:int, value: double): void**
Setzt den Wert des Tensorelements an Position (x,y,z)
x: Zeilennummer.
y: Spaltennummer.
z: Tiefe.
value: Der Wert der gesetzt werden soll.
@return void

▷ **getNumRow(): int**
Gibt die Anzahl der Zeilen des Tensors zurück.
@return Anzahl der Zeilen.

▷ **getNumCol(): int**
Gibt die Anzahl der Spalten des Tensors zurück.
@return Anzahl der Spalten.

▷ **getDepth(): int**
Gibt die Tiefe des Tensors zurück.
@return die Tiefe des Tensors.

4.7 DataAugmentationModule

4.7.1 Funktion

Das DataAugmentationModule enthält die Utility-Klasse **DataAugmentation**, welche Methoden zur Erweiterung der Trainingsdaten zur Verfügung stellt. Diese können sowohl Overfitting von trainierbaren Neuronalen Netzen zur Bilderklassifikation reduzieren, als auch die Klassifikationsergebnisse verbessern.

4.7.2 Verwendung im Kontext

Sowohl beim Trainieren, als auch beim Klassifizieren werden Data-Augmentation-Techniken verwendet.

4.7.3 Aufbau

Das Modul besteht aus der Utility-Klasse **DataAugmentation**.

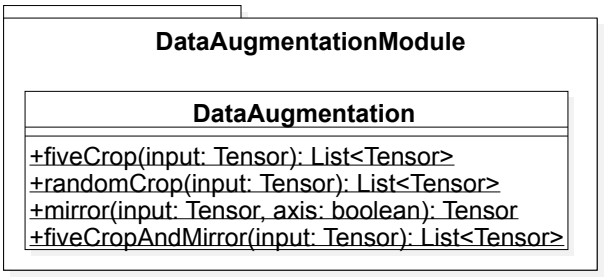

Abbildung 15: Klassendiagramm: Paket DataAugmentationModule

4.7.4 Detaillierte Klassenbeschreibungen

- **«static» fiveCrop(input: Tensor): List<Tensor>**
  Erhält als Parameter ein Bild der Größe 256x256x3 als Tensor Objekt und schneidet fünf 227x227x3 große Bildausschnitte aus (Oben links, oben rechts, unten links, unten rechts und Mitte) und gibt diese als Liste von Tensor Objekten zurück.

input: Bild als Tensor.
@return Liste von Tensor Objekten.

▷ **«static» randomCrop(input: Tensor): List<Tensor>**
Erhält als Parameter ein Bild der Größe 256x256x3 als Tensor Objekt und schneidet zufällig viele 227x227x3 große Bildausschnitte aus und gibt diese als Liste von Tensor Objekten zurück.
input: Bild als Tensor.
@return die Liste von Tensor Objekten.

▷ **«static» mirror(input: Tensor): Tensor**
Erhält als Parameter ein Bild als Tensor Objekt und ruft die Methode editImage(MIRROR, input: TensorObject, params: Params): TensorObject, des ImageEditorModule über das Interface IImageEditor auf. Gibt nach Rückgabe durch das ImageEditorModule das gespiegelte Bild als Tensor Objekt zurück.
input Bild als Tensor.
@return gespiegeltes Bild als Tensor.

▷ **«static» fiveCropAndMirror(input: Tensor): List<Tensor>**
Erhält als Parameter ein Bild als Tensor Objekt. Anschließend wird fünf mal die Methode editImage(CROP, input: TensorObject, params: Params): TensorObject, des ImageEditorModule über das Interface IImageEditor aufgerufen. Liegen die fünf gecroppten Bilder vor, wird für jedes Bild die Methode editImage(MIRROR, input: TensorObject, params: Params): TensorObject, des ImageEditorModule über das Interface IImageEditor aufgerufen. Es werden anschließend die nun insgesamt 10 Bilder als Liste von Tensor Objekten zurückgegeben.
input: Bild als Tensor
@return 10 Bilder als List<Tensor>

4.8 NeuralNetModule

4.8.1 Funktion

Das NeuralNetModule stellt die Architektur eines neuronalen Netzes dar. Dies beinhaltet unter anderem die Modellierung der Forward- und Backward-Prozesse.

4.8.2 Verwendung im Kontext

Der Zugriff auf das Modul erfolgt über die Schnittstelle **INeuralNet**. Die Klasse **Executor** aus dem ManagerModule initialisiert das neuronale Netz. Im Falle der Bilderklassifizierung übergibt sie es dem **PlatformManager** über die Schnittstelle **IPlatformManager**. Im Falle des Trainierens wird es dem TrainingModule über die Schnittstelle **ITraining** übergeben. Zu betonen ist, dass die Trainings- und Klassifizierungslogik hauptsächlich entsprechend in den Modulen TrainingModule und PlatformModule stattfindet.

4.8.3 Aufbau

Die Klasse **NeuralNet** implementiert das Interface **INeuralNet**. **NeuralNet** hat Zugriff auf mehrere Subklasse der Klasse **Layer**. Von der abstrakten Klasse **Layer** erben die abstrakten Klassen **PoolLayer** und **TrainableLayer** sowie die nicht abstrakte Klasse **NormLayer**. **PoolLayer** und **TrainableLayer** wiederum werden von **MaxPool2D** bzw. **FCLayer** und **Conv2DLayer** beerbt. **Conv2DLayer** hat eine Liste der Klasse **Kernel**. **Shape** wird von **Layer** dazu benutzt die Input- und Outputgröße zu abstrahieren. **NeuralNet** greift auf die **LayerType** Enumeration zu welche die verschiedenen Layer-Arten aufzählt, um die Konfigurationsdatei auszulesen.

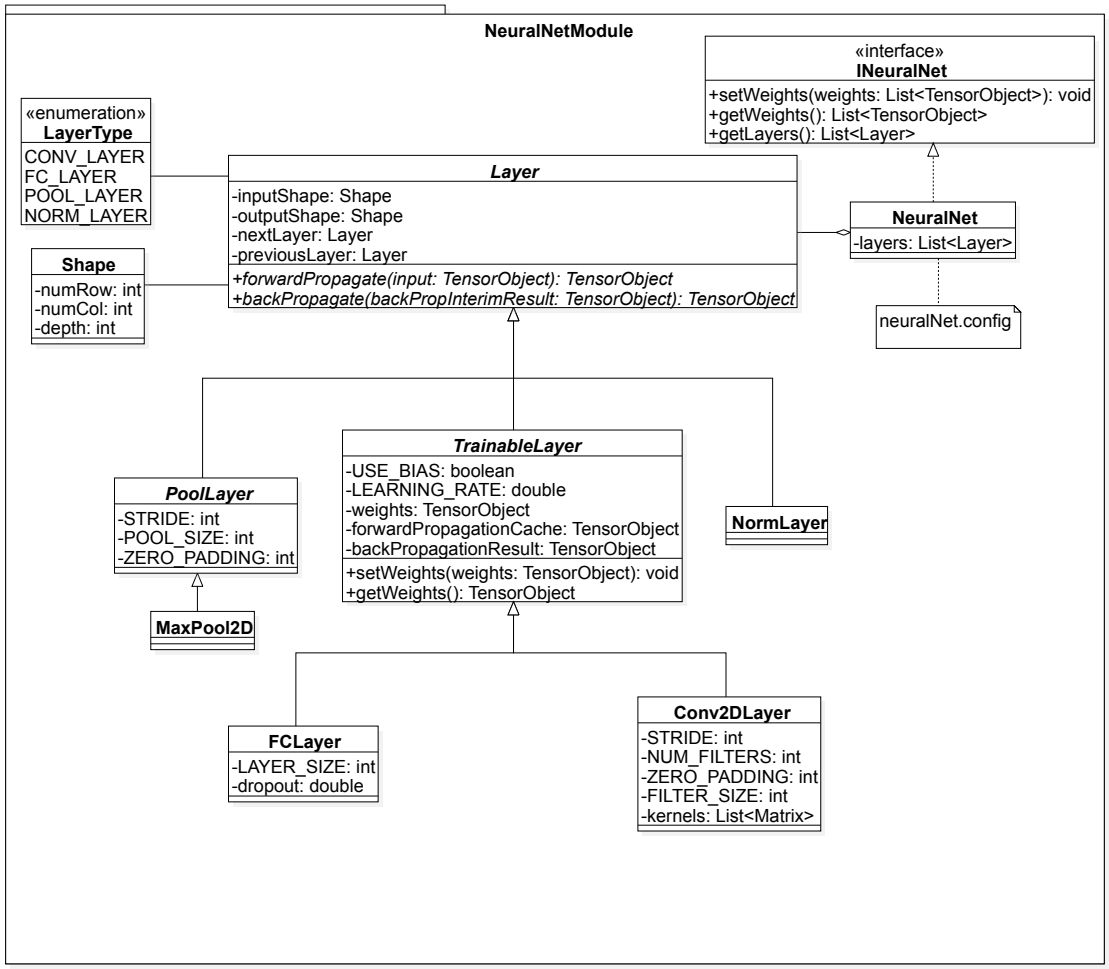

Abbildung 16: Klassendiagramm: Paket NeuralNetModule

4.8.4 Detaillierte Klassenbeschreibungen

○ **«abstract» Layer**
Eine Abstrakte Klasse, die eine Verallgemeinerung eines Layers darstellt. Es ist ein Hauptbaustein von der NN-Architektur

– **«private» inputShape: Shape**
Ein Shape Objekt, der die Größe der Eingabe bestimmt

– **«private» outputShape: Shape**
Ein Shape Objekt, der die Größe der Ausgabe bestimmt

– **«private» nextLayer: Layer**
Eine Verlinkung auf das nächste Layer-Objekt, um Ausgabe von Forward Propagation an das nächste Layer weiterzuleiten.

– **«private» previousLayer: Layer**
Eine Verlinkung auf das vorherige Layer-Objekt, um Ergebnis von Backward Propagation an das vorherige Layer weiterzuleiten.

▷ **forwardPropagate(input: TensorObject): TensorObject**
Diese Methode berechnet die Ausgabe des Layers beim Klassifizieren.
input: Die Eingabe des Layers für die Forward-Propagation. Beim ersten Layer ist sie das zu klassifizierende Bild, bei allen anderen Layers ist sie das Ergebnis von forwardProgate() des vorherigen Layers.
@return: Ergebnis der Forward-Propagation. Es ist die Activation Map

▷ **backPropagate(backPropagateInterimResult: TensorObject): TensorObject**
Diese Methode führt Back-Propagation auf dem Layer aus. Sie führt den Fehler beim Klassifizieren in umgekehrter topologischer Reihenfolge zurück.
backPropagateInterimResult: Ergebnis von backPropagate() des nächsten Layers. Im Falle des Letzten Layers ist es der Endfehler - die Differenz zwischen der Klassifizierung und der gewünschten Ausgabe.
@return: Back-Propagate Result.

○ **«abstract» TrainableLayer**
Eine Abstrakte Klasse, die eine Verallgemeinerung eines trainierbaren Layers darstellt. Es ist ein Hauptbaustein von der NN-Architektur

– **«private» USE_BIAS: boolean**
Ein Hyperparameter, der bestimmt, ob in dem Layer zu seiner Eingabe ein BIAS hinzugefügt werden soll.

– **«private» LEARNING_RATE: double**
Ein Hyperparameter, der bestimmt, wie groß die Änderungen an den Gewichtern beim Gradient Descent sein sollen.

– **«private» weights: TensorObject**
Die Gewichter des Layers

– **«private» forwardPropagationCache: TensorObject**
Zwischenspeicherung des Ergebnisses von der Methode forwardPropagate(). Dieser Ergebnis wird später beim Aufruf der Methode backPropagate() verwendet.

▷ **setWeights(weights: TensorObject): void**
Setzt die Gewichte des Layers.
weights: Gewichte als TensorObject Objekt
@return: void

▷ **getWeights(): TensorObject**
Gibt Gewichte des Layers zurück.
@return: Gewichte als TensorObject Objekt

○ **FCLayer**
Eine Klasse, die eine Modellierung eines trainierbaren Fully Connected Layers darstellt. Es ist ein Hauptbaustein von der NN-Architektur

– **«private» LAYER_SIZE: int**
Ein Hyperparameter, der bestimmt, wie viele Neuronen das Layer enthält.

– **«private» dropout: double**
Ein Hyperparameter, der bestimmt, wie wahrscheinlich die Aktivation eines Neurons ist.

○ **Conv2DLayer**
Eine Klasse, die eine Modellierung eines trainierbaren Convolutional 2D Layers darstellt. Es ist ein Hauptbaustein von der NN-Architektur

– **«private» STRIDE: int**
Ein Hyperparameter, der bestimmt, um wie viele Pixel das Kernelfenster verschoben wird.

– **«private» NUM_FILTERS: int**
Ein Hyperparameter, der bestimmt, wie viele Kernels an der Eingabe des Layers angewendet werden.

– **«private» ZERO_PADDING: int**
Ein Hyperparameter, der bestimmt, wie groß der Nullen-Rand der Eingabe sein soll.

– **«private» FILTER_SIZE: int**
Ein Hyperparameter, der bestimmt, wie groß der Kernel sein soll. Es bestimmt seine Seitengröße.

– **«private» kernels: List<Kernel>**
Eine Liste der Kernel Objekte, die an der Eingabe angewendet werden.

○ **Kernel**
Eine Klasse, die eine Modellierung eines Kernels darstellt. Es ist ein Baustein eines Convolutional 2D Layers.

○ **«interface» INeuralNet**
INeuralNet ist die Schnittstelle für das NeuralNet Module.

▷ **setWeights(weights: List<TensorObject>): void**
Setzt die Gewichte des neuronalen Netzes.
weights: Gewichte als Liste von TensorObject Objekten
@return: void

▷ **getWeights(): List<TensorObject>**
Gibt alle Gewichte des neuronalen Netzes zurück. **@return:**
Gewichte als Liste von TensorObject Objekten

▷ **getLayers(): List<Layer>**
Gibt sämtliche Layers des neuronalen Netzes zurück.
@return: Liste von Layer Objekten

○ **PoolLayer**
Die abstrakte Klasse PoolLayer abstrahiert Bündelungsfunktionen.

– **«private» STRIDE: int**
Die Schrittweite der Bündelungsfunktion.

– **«private» POOL_SIZE: int**
Die Seitenlänge der Quadrate, die einzeln gebündelt werden.

○ **MaxPool2D**
MaxPool2D erbt von der abstrakten Klasse PoolLayer und implementiert die Pooling-Funktion Maximum-3D-Pooling.

○ **NeuralNet**
Implementiert die Schnittstelle INeuralNet und erstelltt die Layer anhand der Konfigurationsdatei neuralNet.config.

- **«private» layers: List<Layer>**
  Liste von Layer Objekten

○ **«enumeration» LayerType**
Aufzählung aller angebotenen Layer-Arten.

- **CONV_LAYER**
- **FC_LAYER**
- **POOL_LAYER**
- **NORM_LAYER**

○ **Shape**
Die Klasse Shape wird von der Klasse Layer dazu benutzt die Input- und Outputgröße zu abstrahieren.

- **«private» numRow: int**
  Anzahl der Zeilen
- **«private» numCol: int**
  Anzahl der Spalten
- **«private» depth: int**
  Tiefe des Objekts

4.9 TrainingModule

4.9.1 Funktion

Das TrainingModule leitet das Trainieren von dem ausgewählten Neuronalen Netz anhand von einer Konfigurationsdatei. Es stellt eine erweiterbare Schnittstelle zur Verwendung von verschiedenen Trainingsalgorithmen zur Verfügung. Anschließend ermöglicht es das Speichern von den angepassten Gewichtern.

4.9.2 Verwendung im Kontext

Auf das Modul wird über die Schnittstelle **ITraining** zugegriffen. Die Hyperparameter werden mittels einer Config Datei beim Initialisieren gesetzt. Die Steuersignale kommen von der Klasse **Executor** aus dem ManagerModule. Sie werden durch die Methodenaufrufe umgesetzt. Train() Methode startet den Trainingsprozess, die Methode stopTraining() bricht ihn ab und saveTrainingWeights() speichert die durch Training gewonnen Gewichter ab. Die Abstrakte Klasse **LearningRule** implementiert diese Schnittstelle. Zudem erbt diese Klasse von der **Subject** Klasse des GUI-Moduls.

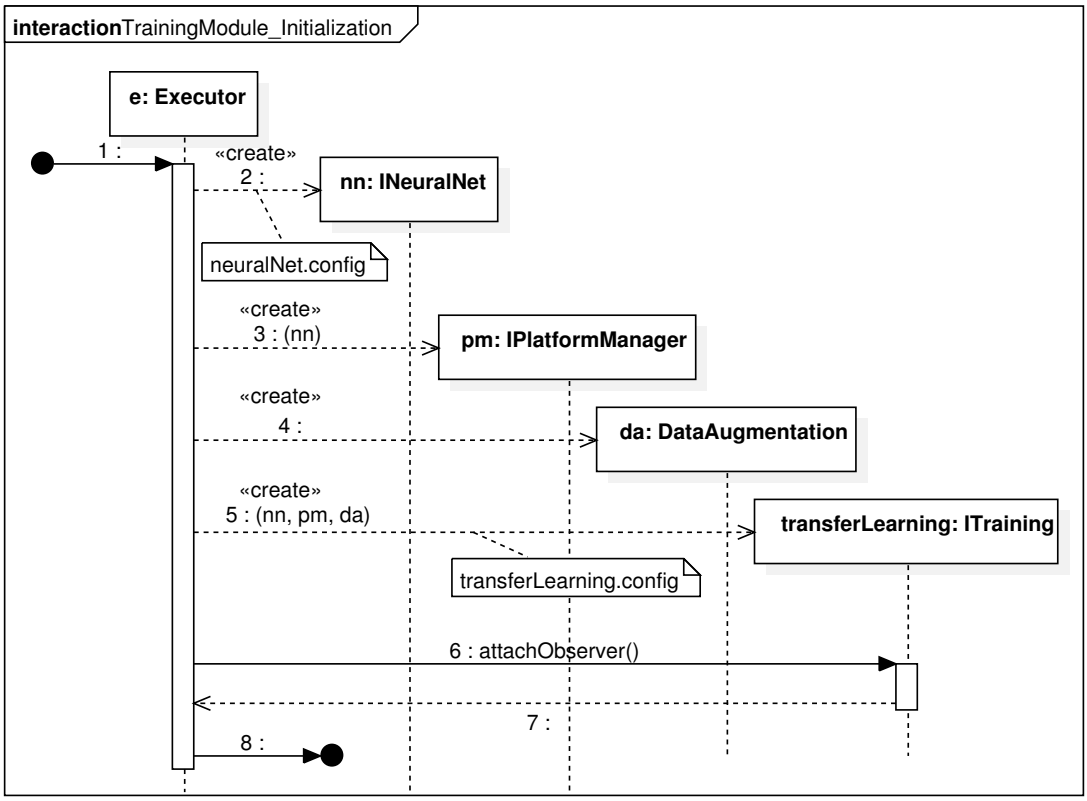

Abbildung 17: Sequenzdiagramm: Initialisierung des Training Moduls

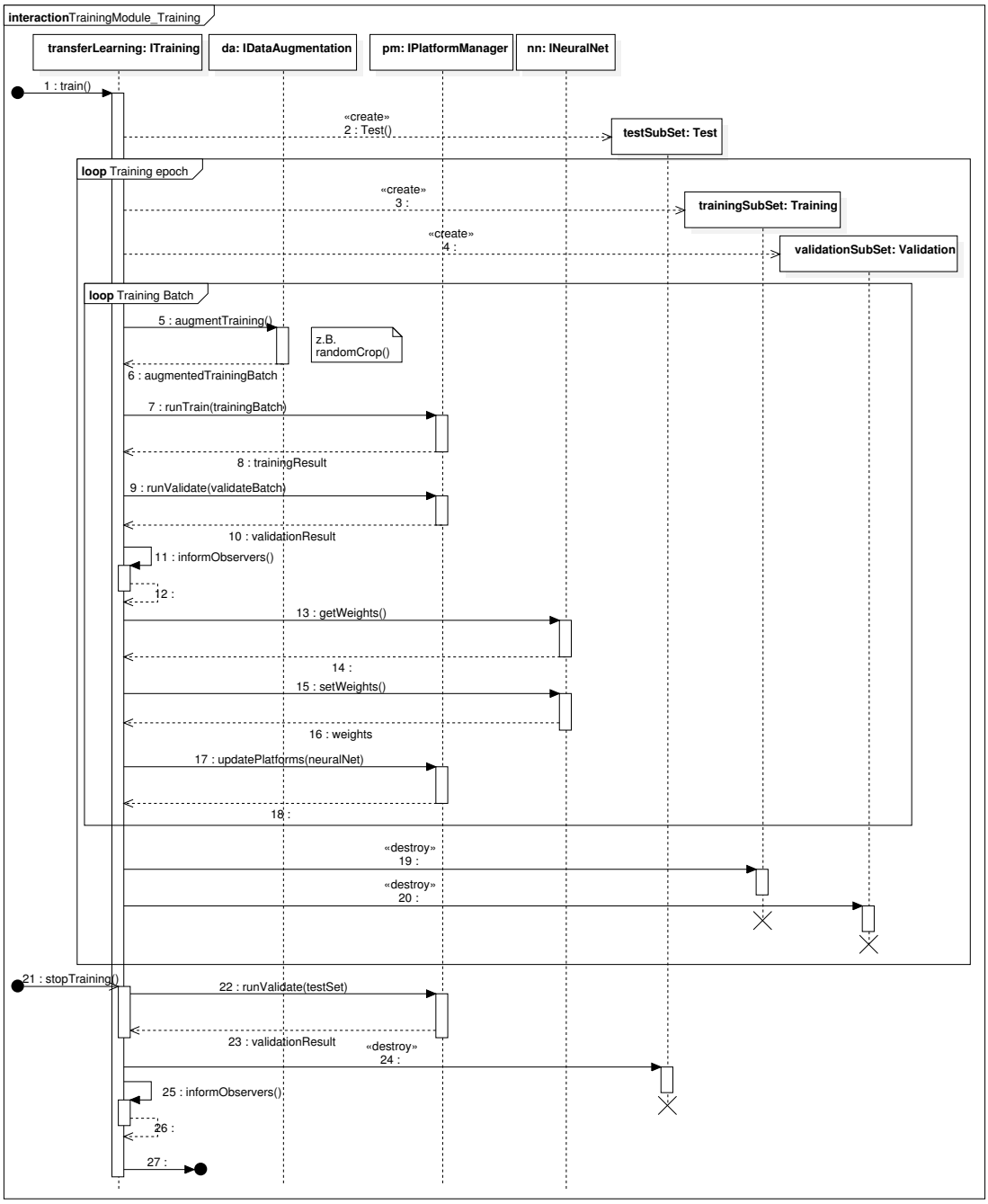

Abbildung 18: Sequenzdiagramm: Der Ablauf des Trainingprozesses

4.9.3 Aufbau

Das Modul enthält die Klasse **TransferLearning**, die von der abstrakten Klasse **LearningRule** erbt und ein konkretes Trainingverfahren darstellt.
Die Klasse **LearningRule** enthält eine Training-Datensatz Struktur in Form von Training, Validation und Test Klassen. Das eingegebene Trainingssatz wird mithilfe von **TrainingElement** Objekten eingespeichert. Die Instanz von **TransferLearning** bekommt nach dem Beobachter Entwurfsmuster dazugehörige GUI-Panels (**OutputTrainingPanel**, **StatisticFrame**) als Observerobjekte. Die Beobachter können bei einer Änderung informiert werden.
In dem TrainingModule wird ein Nullobjekt Entwurfsmuster angewendet. Ein NullObjekt implementiert neben der Klasse **LearningRule** ebenfalls die Schnittstelle **ITraining** und sorgt für fehlerfreie Operation selbst bei NN-Architekturen, die kein Training unterstützen.

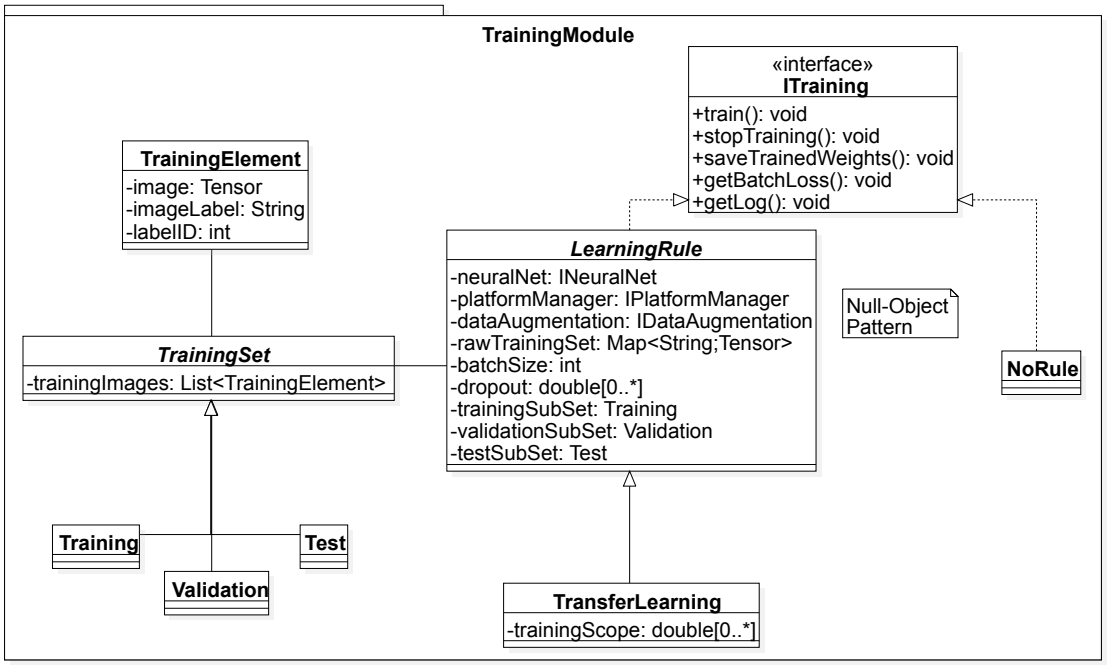

Abbildung 19: Klassendiagramm: Paket TrainingModule

4.9.4 Detaillierte Klassenbeschreibungen

○ **«interface» ITraining**
Schnittstelle des TrainingModuls welche die Methodendeklarationen der Methoden train(), stopTraining() und saveTrainedWeights() enthält.

▷ **train(): void**
Bereitet das Training-Datensatz vor, greift auf die Config Datei und setzt gemäß deren die Hyperparameter, anschließend führt den Trainingsprozess durch.
@return void

○ **«abstract» LearningRule**
LearningRule enthält fürs Trainieren relevante Hyperparameter

– **«private» neuralNet: INeuralNet**
Die aktuelle Instanz der Klasse NeuralNet, um den Zugriff auf das Modell zu gewähren.

– **«private» platformManager: IPlatformManager**
Die aktuelle Instanz der Klasse PlatformManager, um die für das Training benötigte Berechnungen auf den Platformen auszuführen

– **«private» dataAugmentation: DataAugmentation**
Eine Instanz der Klasse DataAugmentation, um das Training-Datensatz vor dem Trainieren zu erweitern.

– **«private» rawTrainingSet: Map<String, Tensor>**
Ein Map Objekt mit vorbearbeiteten Bilder und mit dazugehörigen Labels zur zwischenspeicherung vor der Aufteilung des Training-Datensatzes in Training/Validation/Test.

– **«private» batchSize: int**
Die Größe des Stapels, der beim Trainieren dem PlatformManager übergeben wird. Dieser Attribut bestimmt wie oft (nach wie vielen Bildern) die Gewichter des Modells aktualisiert werden.

– **«private» dropout: double[0..*]**
Dieser Attribut enthält Werte aus dem Bereich [0,1], die für jedes Layer des Modells bestimmen, wie wahrscheinlich eine Aktivation eines Neurons ist. Wert 0.5 bedeutet, dass alle Neuronen in diesem Layer mit 50% Wahrscheinlichkeit nicht an dem Trainieren teilnehmen.

– **«private» trainingSubSet: Training**

Ein Unterdatensatz, auf dem das Modell trainiert wird. Es wird für jede Epoche während des Trianings neu bestimmt.

– **«private» ValidationSubSet: Validation**
Ein Unterdatensatz, mithilfe dessen nach jedem Training-Batch die neu aktualisierte Gewichter des Modells validiert werden. Ebenso wie trainingSubSet wir es für jede Epoche des Trainings neu bestimmt.

– **«private» testSubSet: Test**
Ein Unterdatensatz, auf dem die Leistung des Modells nach dem abgeschlossenem Training einmalig überprüft wird. Anders als trainingSubSet und validationSubSet wird es nur einmalig erstellt und dem Modell bis zum Abschluss des Trainings nicht bekannt gegeben.

– **«private» dataSetPath: String**
Verzeichnispfad des Training-Datensatzes.

○ **NoRule**
Klasse für Handhabung eines Falles, bei dem hinzugefügte NN-Architektur Training nicht unterstützt.

▷ **NoRule()**
Ein Konstruktor der Klasse NoRule, die beim Training nichts tut.

○ **TransferLearning**
Klasse für das konkrete Trainigverfahren Transfer Learning. Dieses Verfahren nützt Verfeinerung des Modells aus, um ein vortrainiertes Modell auf ein Anwendungsfall des Nutzers anzupassen.

– **«private» trainingScope: double[0..*]**
Der Attribut enthält Werte aus dem Bereich [0,1], die für jedes Layer des Modells bestimmen, wie empfindlich es für Änderungen ist. Wert 0 bedeutet, dass das Layer “gefroren” ist - seine Gewichter werden nicht geändert. Wert 1 bedeutet, dass das Layer mit der durch das Hyperparamter LEARNING_RATE vorgegebener Geschwindigkeit trainiert wird.

○ **«abstract» TrainingSet**
Abstrakte Klasse für die erstellung der Training-Unterdatensätze.

– **«private» trainingImages: List<TrainingElement>**
Eine Liste der TrainingElement Objekte.

○ **Training**

Klasse für die Darstellung des Training-Unterdatensatzes.

○ **Validation**
Klasse für die Darstellung des Validation-Unterdatensatzes.

○ **Test**
Klasse für die Darstellung des Test-Unterdatensatzes.

○ **TrainingElement**
Klasse zur Kapselung eines Training-Elements.

- **«private» image: Tensor**
  Ein Trainingsbild.

- **«private» imageLabel: String**
  Ein zu dem image zugehörige Label mit dem Namen der Objekt-Klasse.

- **«private» labelID: int**
  Ein ID des Labels.