Hochschule für Technik
und Wirtschaft Berlin
University of Applied Sciences

Exploration von Handschriften vor und nach 1452

Seminararbeit

Hochschule für Technik und Wirtschaft (HTW) Berlin
Fachbereich 4: Informatik, Kommunikation und Wirtschaft
Studiengang *Angewandte Informatik*

Grundlagen sozialer Netze
bei Frau Prof. Dr. Christin Schmidt

Eingereicht von Emma Calewaert [571460], Iuliia Chelysheva [559019],
Ursula Kundert [574661], Dennis Dominik Lehmann [568827],
Magnus Petersen [573258], Tanja Schlanstedt [573417], Samet Yildiz [569278]

19.01.2021

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Problemformulierung (Ursula Kundert)

Zur Zeit entstehen immer mehr Meta-Portale über Handschriften und alte Drucke. Damit werden alte Texte aus aller Welt zusammen suchbar. Was aber oft fehlt sind gute Metadaten zu den abgebildeten Objekten, z. B. eine einheitliche Datierung. Hier setzt unser Projekt an: Wir untersuchen, ob es möglich ist, einen selbstlernenden Algorithmus zu trainieren, der entscheidet, ob ein Objekt **vor oder nach 1452** entstanden ist. Dieses Stichdatum haben wir gewählt, weil es uns entscheidend scheint, ob ein Objekt vor oder nach der (Wieder-)Erfindung des Drucks mit beweglichen Lettern entstanden ist.(s. Abbildung 2.1) Wir widmen uns damit im Kleinen einem Trend, der 2021 auch in der Geschäfts-Analytik wichtig werden wird: dass nämlich Metadaten durch maschinelles Lernen angereichert und dadurch gleichsam zum Leben erweckt werden (engl. *activating passive metadata*).[Sum20]

2. Theorie

2.1. Ansätze für maschinelles Lernen (Ursula Kundert)

[Zha+20a], 22-38 beschreiben drei grundsätzlich verschiedene Ansätze für maschinelles Lernen: Überwachtes Lernen (engl. *supervised learning*), nicht-überwachtes Lernen (engl. *unsupervised learning*) und Lernen am Erfolg (engl. *reinforcement learning*, Deutsch oft falsch als "verstärktes Lernen" übersetzt). Wir haben uns für **überwachtes Lernen entschieden**. Dafür spricht zunächst ganz allgemein, dass es für die **meisten erfolgreichen Anwendungen von maschinellem Lernen** in der Industrie verantwortlich ist.([Zha+20a], 23)

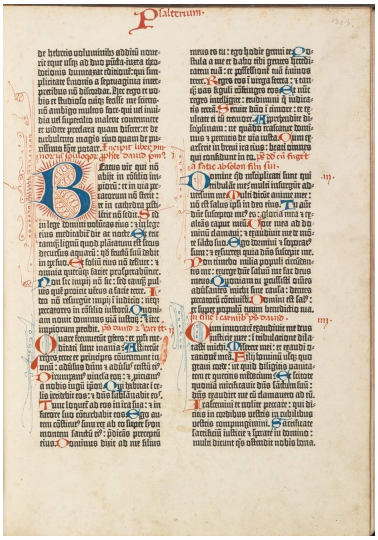
Beim überwachten Lernen geht es um die Vorhersage einer Ausgabe (engl. *output*) zu einer Eingabe (engl. *input*). Die Eingabe besteht meist aus einem Bündel von Merkmalen (engl. *input features*, bei uns allen Pixeln des Bildes), die Ausgabe meist aus einem einzigen (engl. *label*, bei uns der Datierung). Jedes Paar aus eingegebenen Merkmalen und vorhergesagtem Merkmal stellt einen Beispielpaar dar (engl. *example*). Ziel des Vorgehens ist es, ein Modell zu erstellen, das es erlaubt, zu jedem Datensatz von eingegebenen Merkmalen das richtige vorhergesagte Merkmal zu finden.

Damit nach dem Training eine Vorhersage möglich ist, wird in dieser Methode die **Wahrscheinlichkeit** abgeschätzt, mit der aus den eingegebenen Merkmalen auf das *label* geschlossen werden kann. Dieses einfache Prinzip macht einen Teil des Erfolgs von überwachtem Lernen aus.([Zha+20a], 23) Das wird anhand von Datensätzen trainiert, zu denen schon ein geprüft wahres *label* vorliegt.

Der jeweilige Lern-Algorithmus (zur Auswahl s. unten) erstellt aus dem *training set*, einer zufällig ausgewählten Stichprobe aus unseren geprüften Beispielpaaren, das erlernte Modell (engl. *learned model*). Eine weitere Stichprobe, die keine Schnittmenge mit der ersten bildet, dient der Überprüfung des Modells, mit der die Wahrscheinlichkeit des richtigen Schlusses durch das Modell ermittelt werden kann. Dies kann wiederum auf unterschiedliche Arten angegangen werden: Hätten wir unsere Frage so formuliert, dass wir uns dem tatsächlichen Schreibjahr einer Handschrift in der Vorhersage möglichst nähern wollen, wäre eine Datengrundlage nötig gewesen, welche mehrere Jahrhunderte mit ausreichenden Beispielen pro Zeitraum abdeckt. Da die Handschriften-Digitalisate nicht nur nach Bibliotheken, sondern auch nach Epochen (Antike, Mittelalter, 16., 17., 18. Jahrhundert und Moderne) über verschiedenste Repositorien verstreut sind, haben wir unsere Fragestellung aufgrund einer Ökonomie des Datensammelns eingeschränkt.

2.2. Medientheorie (Ursula Kundert)

Mit unserer Problemformulierung, ob eine Handschrift vor oder nach 1452 entstanden ist, haben wir deshalb ein **Klassifizierungs-Problem** formuliert, das medienhistorisch anschlussfähig ist, da es das Datierungsproblem an einem historischen Ereignispunkt angeht, der ausgehend von Marshall McLuhans mehr postulierten als bewiesenen Medien-Genealogie *"The Gutenberg galaxis"* von 1962 [McL62] gerne als Paradigmenwechsel beschrieben wird. In neuerer Zeit wird allerdings kritisch zu McLuhan weniger von einem klaren historischen Wechsel als vielmehr von einem Medienwandel gesprochen.[CK07] Unser Modell wird also überprüfen, ob sich ein solcher Paradigmenwechsel an den Digitalisaten von Psalmentexten ablesen lässt, also auch im Medium der geistlichen Handschrift, dem häufigsten Medium der damaligen Buchproduktion, direkt niederschlägt.



[Bil]

Abbildung 2.1.: *Anfang des Psalters in einem Exemplar der 42-zeiligen Gutenberg-Bibel auf Papier*

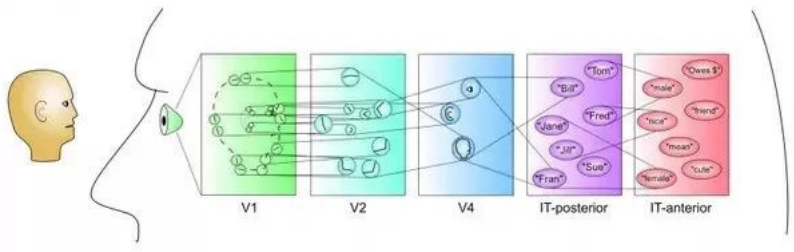
2.3. Mustererkennung (Tom Magnus Petersen)

Dies führen wir mit Mustererkennung (engl. *image recognition*) durch. Wir bauen nach Möglichkeit auf den Vorarbeiten von Sheng He, Petros Samara, Jan Burgers und Lambert Schomaker 2016 auf []. Mustererkennung besteht aus Vorverarbeitung, Merkmalsgewinnung, Merkmalsreduktion und Klassifizierung ([DHS00] 4). Vorverarbeitung steht in unserem Fall (Bilderkennung) lediglich für sämtliche Aktionen, die Bilddaten in den idealen Zustand für Verwertung zu bringen (s. Kap. 3.3). Für die Klassifizierung entschieden wir uns für die Verwendung eines künstlichen neuronalen Netzwerkes (ANN, eng.: *artificial neural network*). Ferner entschieden wir auf die Verwendung von CNN (*Convolutional Neural Networks*, dt.: „faltendes neurales Netzwerk“, auch Faltungsnetz), welche Merkmalsgewinnung (-> *Convolutions*), Merkmalsreduktion (->*Pooling*) und Klassifizierung (-> *Dense Layer*) in einer Methode vereinen und seit mehreren Jahren als am zuverlässigsten Lernalgorithmus im Feld der Bilderkennung gilt (S.

255 [Zha+20a] S. 255 und [Sch14a] S.19). Implementierungsschwierigkeiten und ein Mangel an Lernressourcen führten zudem zum Ausschließen von Hybrid-Systemen (zum Beispiel die Verwendung von *Gradient Boosting* in Form von XGBoost oder die Verwendung von *Support Vector Machines* (SVM) als Klassifizierer da unser Datensatz nicht von der Einschränkung auf binäre Klassifikation betroffen ist [Aga17]).

2.4. Convolutional Neural Network (Iuliia Chelysheva, Samet Yildiz)

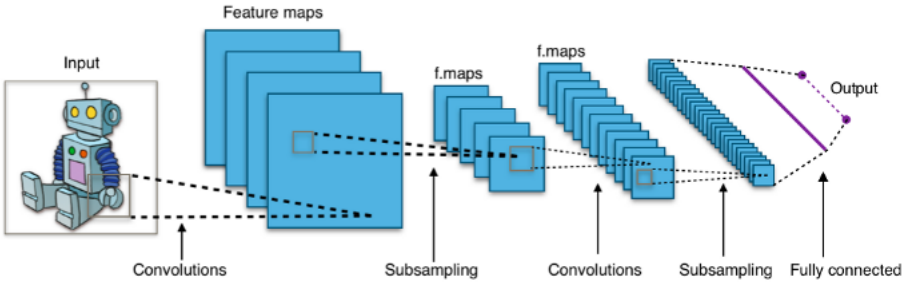
Ein **CNN** ist eine spezielle Art von mehrschichtigen neuronalen Netzen, welche visuelle Muster direkt aus Pixelbildern mit minimaler Vorverarbeitung erkennen.[ZF13] CNNs sind von der Natur inspirierte Modelle, welche in den Forschungen von D. H. Hubel und T. N. Wiesel [HW62] beschrieben wurden. Sie schlugen eine Erklärung für die Art und Weise vor, wie Menschen und Tiere die Welt um sich herum mithilfe einer Schichtenarchitektur von Neuronen im Gehirn visuell wahrnehmen, was wiederum die Ingenieure dazu inspirierte, zu versuchen, ähnliche Mustererkennungsmechanismen in der Bildverarbeitung zu entwickeln. In ihrer Hypothese werden innerhalb des visuellen Kortex komplexe funktionelle Antworten, die von "komplexen Zellenërzeugt werden, aus einfacheren Antworten von einfachen Zellen konstruiert. Auf der folgenden Abbildung (s. Abbildung 2.2) wird der visuelle Erkennungsprozess eines Menschen dargestellt.



[Cha]

Abbildung 2.2.: *Wie Menschen Objekte visuell erkennen*

Ein CNN lässt sich somit wie eine Folge von Schichten beschreiben, wobei jede Schicht mit der Umwandlung der Informationen aus den in den vorherigen Schichten verfügbaren Werten in komplexere Informationen verbunden ist und zur weiteren Verallgemeinerung an die nächsten Schichten weitergegeben wird. Auf der unteren Abbildung (s. Abbildung 2.3) ist eine allgemeine Darstellung eines CNNs zu sehen.

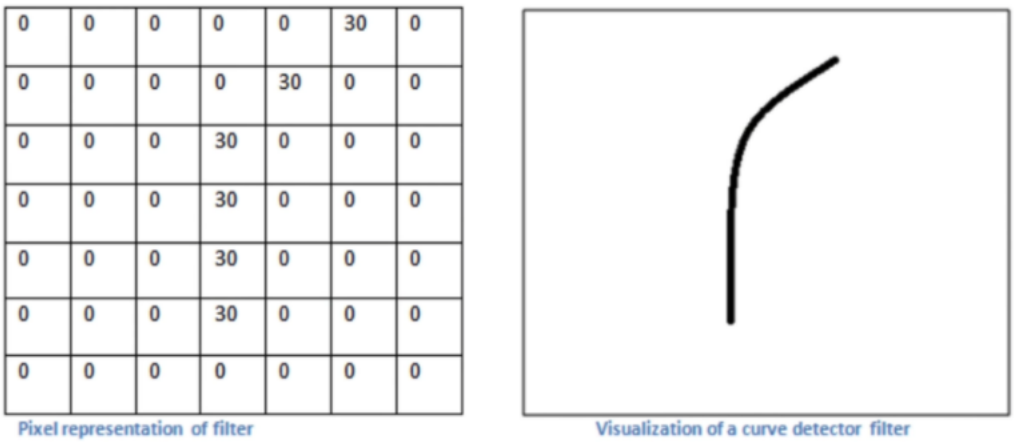


[201]

Abbildung 2.3.: *Allgemeine Darstellung von CNN*

CNNs lassen sich auf zwei grundlegende Blöcke unterteilen: *Convolutional Block* (Faltungsblock) und *Fully Connected Block*. Der Faltungsblock besteht aus dem *Convolutional Layer* und dem *Pooling Layer*. Dieser Block bildet zusammen den wesentlichen Bestandteil der Extraktion. Der *Fully Connected Block* ist zweite Bestandteil und besteht aus einer vollständig verbundenen einfachen neuronalen Netzwerkarchitektur. Diese Schicht die Aufgabe der Klassifizierung basierend auf der Eingabe aus dem Faltungsblock.

Convolutional Layer erstellt eine *Feature-Map*, um die Klassenwahrscheinlichkeiten für jedes *Feature* vorherzusagen, indem ein Filter angewendet wird, der das gesamte Bild mit jeweils wenigen Pixeln untersucht. Filter bzw. *Kernel* sind auch eine Abbildung, welche eine bestimmte Funktion darstellt. In folgender Abbildung (s. Abbildung 2.4) wird beispielsweise eine Kurve dargestellt. Diese dient als Beispielmerkmal, welches erkannt werden soll - es soll mithilfe von diesen Filtern bestimmt werden, ob so eine Kurve in einem Bild vorhanden ist. Die *Pooling* layer verkleinert die Informationsmenge, welche die *Convolutional Layer* für jedes Merkmal generiert, und behält die wichtigsten Informationen bei (der Prozess *Convolutional-* und *Pooling Layer* wird normalerweise mehrmals wiederholt). ([Mis])



[Cha]

Abbildung 2.4.: *Beispiel von einem Filter, welcher eine Kurve in Pixel-Werten wiedergibt*

Wie im Kapitel zu DenseNet unten ausführlicher beschrieben wird, haben herkömmliche Faltungsnetzwerke mit n Schichten $n-1$ Verbindungen; eine zwischen jeder Schicht und ihrer nachfolgenden Schicht. Für die jeweilige Ebene wird die *feature-map* der vorhergehenden Schicht als Eingabe verwendet, und ihre eigene *feature-map* wird anschließend als Eingabe für die nachfolgende Schicht benutzt.

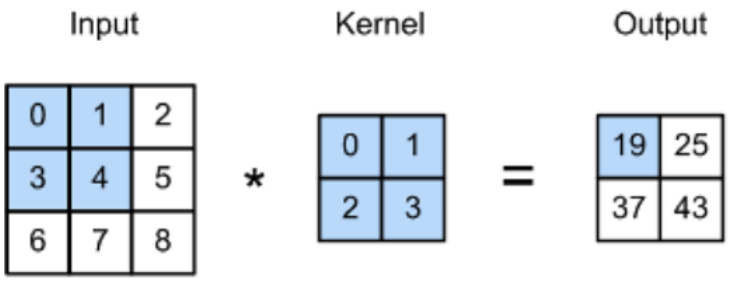
Die Reihenfolge der Merkmale spielt bei den Faltungsnetzen keine Rolle, somit können wir ähnliche Ergebnisse erzielen, unabhängig davon, ob wir eine Reihenfolge beibehalten, die der räumlichen Struktur der Pixel entspricht, oder ob wir die Spalten unserer Entwurfsmatrix permutieren, bevor wir die Parameter des mehrlagigen Perzeptrons (MLP) anpassen. Vorzugsweise würden wir unser Vorwissen nutzen, dass nahegelegene Pixel normalerweise miteinander in Beziehung stehen, um effiziente Modelle zum Lernen aus Bilddaten zu erstellen. Die Faltungsnetze (CNNs) wurden genau für diesen Zweck entwickelt.

CNN-basierte Architekturen sind im Bereich der Bildverarbeitung mittlerweile allgegenwärtig und dominant geworden. CNNs sind dank mehrerer Schichten bei der Erzielung genauer Modelle **stichproben- und auch rechnerisch effizient**, da sie weniger Parameter als vollständig verbundene Architekturen erfordern und weil Faltungen leicht über Grafikprozessor-Kerne (GPU) parallelisiert werden können. Es erscheint sinnvoll, dass bei jeder Methode, mit der wir Objekte erkennen, die genaue Position des Objekts im Bild nicht übermäßig berücksichtigt werden sollte. Wie ein konkretes Objekt aussieht, hängt nicht davon ab, wo in der Umgebung sich dieses Objekt befindet. CNNs systematisieren diese Idee der räumlichen Invarianz und nutzen sie, um nützliche Darstellungen mit weniger Parametern zu lernen.

Folgende Aspekte werden bei CNNs berücksichtigt: Die **Übersetzungsinvarianz** in Bildern impliziert, dass alle Patches eines Bildes auf dieselbe Weise behandelt werden. **Lokalität** bedeutet, dass nur eine kleine Nachbarschaft von Pixeln verwendet wird, um die entsprechenden versteckten Darstellungen zu berechnen. Bei der Bildverarbeitung erfordern Faltungsschichten typischerweise **viel weniger Parameter** als vollständig verbundene Schichten. CNNs sind eine spezielle Familie neuronaler Netze, die **Faltungs-Schichten** enthalten. Mit den Ein- und Ausgabekanälen kann ein Modell an **jedem räumlichen Ort** mehrere Aspekte eines Bildes erfassen. ([Zha+20a] Ch. 6)

2.5. Die Kreuzkorrelations-Operation (Iuliia Chelysheva)

Da wir für Bilderkennung Faltungs-Schichten brauchen, benötigen wir eine Matrix-Multiplikation, wie sie in folgender Abbildung (s. Abbildung 2.5) zu sehen ist: Die Eingabe ist hier ein zweidimensionaler Tensor mit einer Maße nxn (in unserem Projekt werden die konkreten Parameter im Kap. 4.4 beschrieben).



[Zha+20a]

Abbildung 2.5.: *Matrizenmultiplikation*

Die Operationen welche die Faltungsschichten darstellen, werden genauer als Kreuzkorrelationen beschrieben. Basierend auf den oben beschriebenen Faltungsschichten, werden in einer solchen Schicht ein Eingangstensor und ein Kerneltensor kombiniert, um durch eine **Kreuzkorrelationsoperation** einen Ausgangstensor zu erzeugen. Diese Operation beinhaltet das Multiplizieren der Werte einer Zelle, welche einer bestimmten Zeile und Spalte der Bildmatrix entspricht, mit dem Wert der entsprechenden Zelle in der Filter-Matrix (engl. *kernel matrix*). Diese Operation wird für die Werte aller Zellen innerhalb der Spanne der Filter-Matrix durchgeführt, die Werte werden anschließend zu einer Ausgabe addiert.

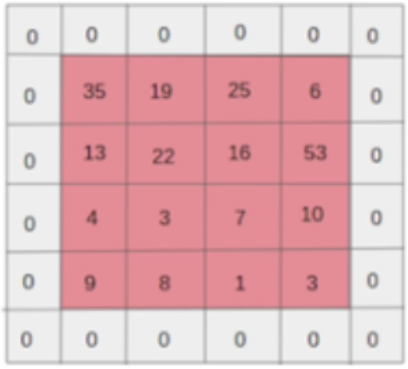
Bei der zweidimensionalen Kreuzkorrelations-Operation beginnen wir mit dem Faltungsfenster in der oberen linken Ecke des Eingangstensors und schieben es über den Eingangstensor, sowohl von links nach rechts als auch von oben nach unten. Wenn das Faltungsfenster an eine bestimmte Position verschoben wird, werden der in diesem Fenster enthaltene Eingabe-Subtensor und der *Kernel*-Tensor elementweise multipliziert und der resultierende Tensor wird summiert, was einen einzelnen Skalarwert ergibt. Dieses Ergebnis gibt den Wert des Ausgangstensors an der entsprechenden Stelle an. Auf der oberen Abbildung hat der Ausgangstensor eine Höhe von 2 und eine Breite von 2, und die vier Elemente werden aus der zweidimensionalen Kreuzkorrelationsoperation abgeleitet. Es ist auch zu beachten, dass wenn das Merkmal in einem bestimmten Teil des Bildes vorhanden ist, gibt es einen sehr großen Wert bei der Faltung zurück, während es für die anderen Stellen einen kleinen Wert zurückgegeben würde, welcher bedeutet, dass dieses Merkmal dort nicht vorhanden ist. ([Zha+20a] Kap. 6.2.1)

2.6. Padding und Stride (Iuliia Chelysheva)

In unserem Projekt verwenden wir Padding und Stride, welche sich auf die Größe der Ausgabedaten auswirken. Es ist zu beachten, dass die Filter-Matrix immer eine Breite und Höhe größer als 1 haben. Nachdem viele aufeinanderfolgende *convolutions* geschaltet werden, werden die Ausgaben produziert, die erheblich kleiner sind als unsere ursprüngliche Eingabe. Wenn beispielsweise mit einem Bild von 240x240 Pixeln gearbeitet wird, so werden schon zehn Schichten von 5x5 *convolutions* das Bild auf 200x200 Pixel reduzieren. Somit werden schon ca. 30% des Bildes abgeschnitten und

damit alle interessanten Informationen über die Grenzen des Originalbildes.([Zha+20a], Kap. 6.3)

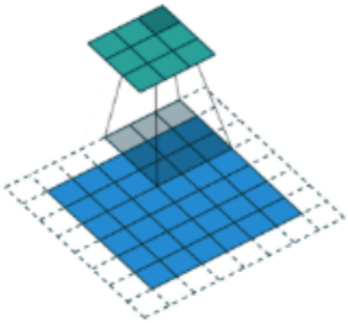
Padding ist das beliebteste Tool zur Behandlung dieses Problems. Durch Padding können Höhe und Breite der Ausgabe erhöht werden. Dies wird häufig verwendet, um der Ausgabe die gleiche Höhe und Breite wie der Eingabe zu geben. In folgender Abbildung (s. Abbildung 2.6) ist ein Beispiel für Padding dargestellt, wobei die aktuelle Matrix vergrößert und mit Nullen befüllt wird.



[Zha+20a]

Abbildung 2.6.: *Beispiel für Padding*

Mithilfe von Stride kann die Auflösung der Ausgabe verringert werden, z. B. die Höhe und Breite der Ausgabe auf nur $\frac{1}{n}$ der Höhe und Breite der Ausgabe (n ist eine Ganzzahl größer als 1). Somit kann mithilfe von Padding und Stride die Dimensionalität der Daten effektiv angepasst werden. In folgender Abbildung (s. Abbildung 2.7) ist ein Teil vom Striding-Prozess dargestellt: dabei wird der Filter nicht um jeweils eine Zeile oder eine Spalte, sondern möglicherweise jedes Mal um zwei oder drei Zeilen oder Spalten verschoben. Dies wird im Allgemeinen durchgeführt, um die Anzahl der Berechnungen und auch die Größe der Ausgabematrix zu verringern. Bei großen Bildern führt dies nicht zu Datenverlust, sondern reduziert die Rechenkosten in großem Maßstab.



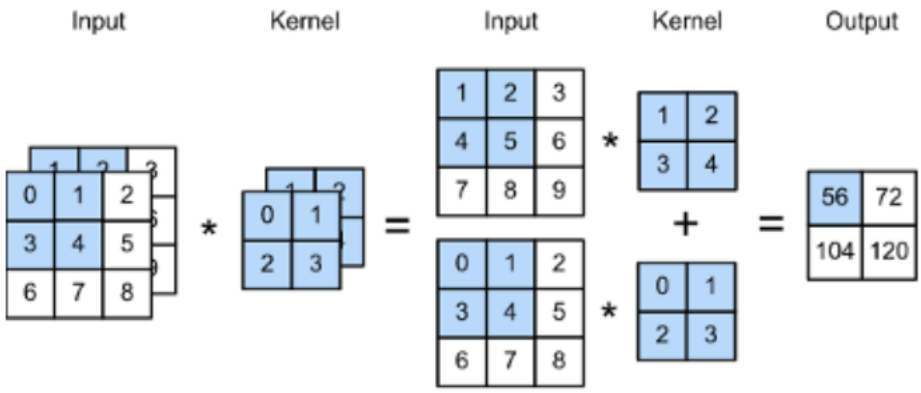
[Zha+20a]

Abbildung 2.7.: *Beispiel für Striding*

2.7. Mehrere Eingangskanäle (Iuliia Chelysheva)

Mehrere Kanäle können verwendet werden, um die Modellparameter der Faltungsschicht zu erweitern. Die $1x1$-Faltungsschicht entspricht der vollständig verbundenen Schicht, wenn sie pro Pixel angewendet wird. Sie wird typischerweise verwendet, um die Anzahl der Kanäle zwischen den Netzwerkschichten anzupassen und die Komplexität des Modells zu steuern.([Zha+20a] Kap. 6)

In nächster Abbildung (s. Abbildung 2.8) wird Beispiel für eine zweidimensionale Kreuzkorrelation mit zwei Eingangskanälen dargestellt. Die schattierten Bereiche sind das erste Ausgabeelement sowie die Eingabe- und Filter-Tensorelemente, die für die Ausgabeberechnung verwendet werden: $(1x1 + 2x2 + 4x3 + 5x4) + (0x0 + 1x1 + 3x2 + 4x3) = 56$. Nachdem die Filter auf das Eingabe-Bild angewendet werden, können *feature maps* generiert werden. Die Filter helfen dabei, verschiedene in einem Bild vorhandene Merkmale wie Kanten, vertikale oder horizontale Linien oder sonstige Deformierungen zu identifizieren. Das *Pooling* wird anschließend auf die *feature maps* angewendet, um eine *Translational Invariance* zu erreichen. Im folgenden Abschnitt werden *Pooling* und weitere Komponente näher beschrieben.

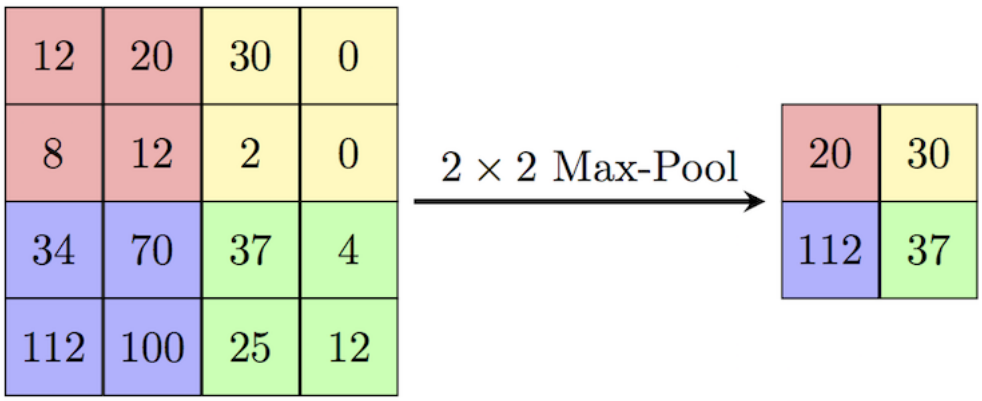


[Zha+20a]

Abbildung 2.8.: *Beispiel für eine zweidimensionale Kreuzkorrelation mit zwei Eingangskanälen*

2.8. Pooling (Samet Yildiz)

Pooling layers werden benutzt um die Dimension von Eingangsdaten zu reduzieren und diejenigen Merkmale hervorhebt (aggregiert), welche für die Klassifizierung wichtiger sind. Dadurch entsteht eine Generalisierung der Daten. Es ist zum Beispiel in der realen Welt nie garantiert, dass ein Objekt sich in einem Bild immer an der gleichen Position befindet. Durch den Generalisierungseffekt wird die Empfindlichkeit auf Lokalität in den *convolutional layers* verringert (ïnvariance to translation", [Zha+20a], Kap. 6.5). Die Reduktion der Dimension und der zu erlernenden Parameter (Petersen) reduziert gleichzeitig den Rechenaufwand. Beim *pooling* sind die 2 gängigsten Methoden *max-pooling* und *average-pooling*. Bei beiden Methoden definiert man zuerst eine quadratische **Fenstergröße** (engl. *kernel-size*) genannt. Dieses **Fenster** (engl. *kernel*) wird Schritt für Schritt über die Eingabe (engl. *input*) geschoben. Die **Schrittgröße** nennt man auf Englisch *stride*. Angenommen, wir haben eine Eingabe-Matrix der Größe 4x4 und möchten *max-pooling* darauf anwenden mit einer Fenstergröße von 2x2 und der Schrittweite 2 (Abbildung 2.9).



[Zha+20a]

Abbildung 2.9.: *Max-Pool Darstellung*

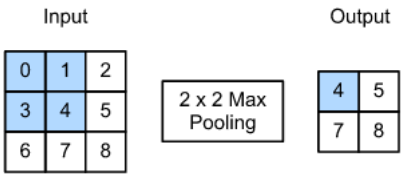
Die Fenstergröße passt genau viermal auf die Eingab-Matrix wenn man das Fenster zweimal verschiebt, angefangen von oben links bis nach unten rechts. Man nimmt immer den größten Wert der Untermatrix, die das Fenster zurzeit überdeckt. Die vier Berechnungen in diesem Falle sind:

max (12, 20, 8, 12)	= 20
max (30, 0, 2, 0)	= 30
max (34, 70, 112, 100)	= 112
max (37, 4, 25, 12)	= 37

Die Output Matrix hat die Größe 2x2. Bei **average-pooling** wird der Durchschnitt der Werte berechnet. In dem oberen Beispiel wären das folgende Berechnungen:

avg (12, 20, 8, 12)	= 13
avg (30, 0, 2, 0)	= 8
avg (34, 70, 112, 100)	= 79
avg (37, 4, 25, 12)	= 19,5

Die Fenstergröße des *pooling* kann sich auch überschneiden (Abbildung 2.10).



[Zha+20a]

Abbildung 2.10.: *Max-Pooling 2x2*

Wenn die Fenstergröße nicht zur Schrittweite passt oder man eine andere Ausgabe-Dimension generieren muss, weil die nächsten Schichten im CNN eine bestimmte Eingabe-Dimension erwarten, wird **padding** eingesetzt in Kombination mit der Schrittweite. Man erweitert die Dimensionen und füllt die neue entstandenen Elemente meist mit Nullen (*Zero-Padding*).

Als Beispiel haben wir eine 5x5 Eingabe-Matrix a auf der man mit einer Schrittweite von 2 und einer Fenstergröße von 2x2 *pooling* anwenden will. Wie man in Abbildung 2.11 erkennt, passt die Fenstergröße mit der gegebenen Schrittgröße nicht auf meine Eingabe-Matrix. Durch das *padding* der Dimension auf 6x6 (gestrichelte Kacheln), passt die Fenstergröße wieder und unsere Ausgabe ist eine 3x3 Matrix.

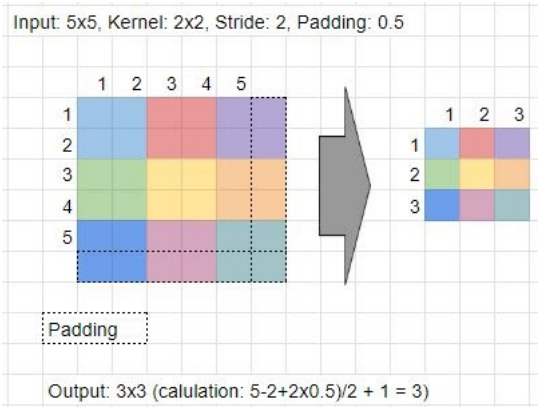


Abbildung 2.11.: *https://i.stack.imgur.com/NFQKa.jpg*

Bei einer Eingabe mit mehreren Kanälen (engl. *channels*) wird *pooling* auf jeden einzelnen Kanal angewendet. Das heißt, bei einer 64-Kanal-Eingabe kommt eine 64-Kanal-Ausgabe heraus (Abbildung 2.12).

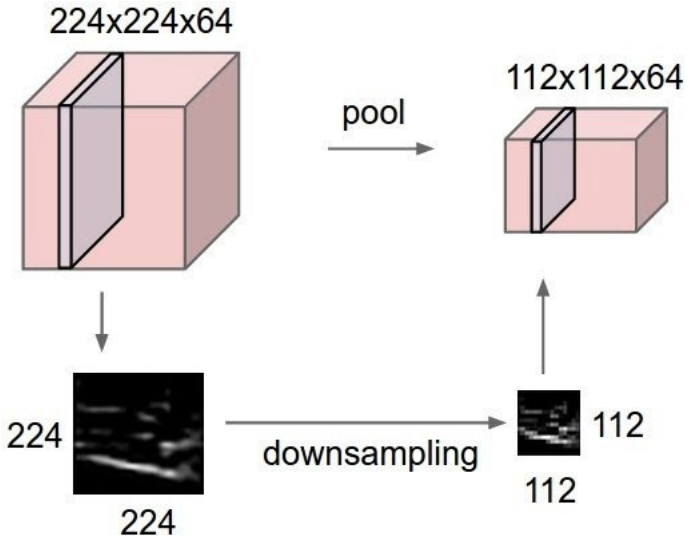


Abbildung 2.12.: *https://miro.medium.com/max/900/1*mAb72pBCgfSG707fgDoxgA.jpeg*

Average-pooling und *max-pooling* werden eigentlich auf der Ausgabe der davor geschalteten *convolutional layer* angewendet, aber sieht man, dass *average-Pooling* die Schärfe des Bildes verringert und eine allgemeinere Representation hinterlässt, wobei *max-pool* die Kanten verstärkt (Abbildung 2.13).[Bas19]

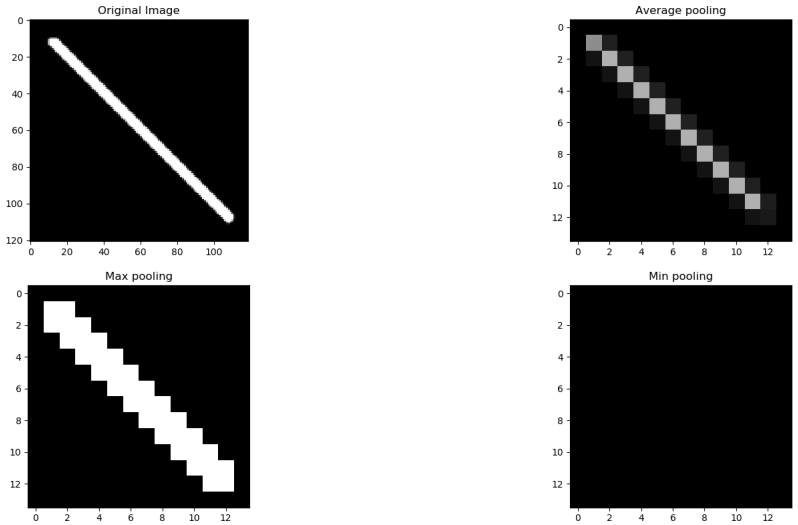
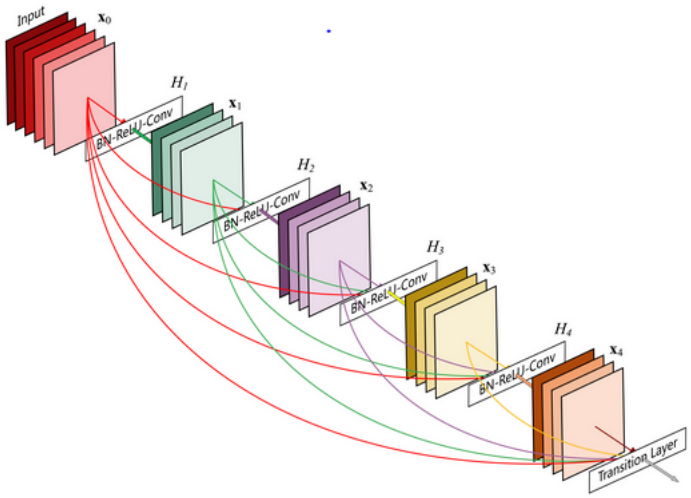


Abbildung 2.13.: *https://miro.medium.com/max/5280/1*fpmtYoP9e8hycFIILPfW5A.png*

2.9. DenseNet (Tanja Schlanstedt)

Weil bei der Netzwerk-Architektur DenseNet jede Schicht in Vorwärtsrichtung mit jeder folgenden Schicht verbunden ist, hat sie $\frac{N(N+1)}{2}$ direkte Verbindungen. Dieses **dichte Konnektivitätsmuster** erklärt auch den Namen *densely connected convolutional network*. Jede Schicht benutzt alle *feature maps* der vorhergehenden Schichten als Eingabe. Die

erzeugten *feature maps* werden als Eingabe für alle nachfolgenden Schichten verwendet. Folglich hat die *n-te* Schicht n Eingaben, die aus den *feature maps* der vorherigen Convolutional Blocks besteht. Die eigenen *feature maps* werden an alle folgenden $N - n$ Schichten übergeben. In der Abbildung 2.14 ist das Layout von dicht vernetzten Blöcken (engl. *dense blocks*) dargestellt. ([Hua+16])



[Hua+16]

Abbildung 2.14.: *Aufbau eines Dense Blocks*

Ein Unterschied zwischen DenseNet und anderen Netzwerkarchitekturen, den Huang u.a. 2016 erklären, ist die Möglichkeit, **langsam zu wachsen**. Der Hyperparameter k drückt die Wachstumsrate des Netzwerkes aus. Grundsätzlich ist anzunehmen, dass die Wachstumsrate bestimmt, wie viele neue Informationen jede Schicht zum globalen Zustand des Netzwerkes hinzufügt. Den globalen Zustand kann man hier mit der Anzahl aller erlernten *feature maps* gleichsetzen. Bereits eine geringe Wachstumsrate (z.B. ($k = 12$) reicht also, um hohe Genauigkeiten zu erhalten. Das ist darauf zurückzuführen, dass jede Schicht Zugriff auf die vorhergehenden *feature maps*, das kollektive Wissen eines Netzwerkes, in seinem jeweiligen *dense block* hat. Bei der abschließenden Klassifizierung wird die Entscheidung durch die Auswertung aller *feature maps* des Netzwerkes getroffen. Oberflächlich betrachtet ähnelt die Struktur unseres Modells dem **ResNet**, einer anderen populären Architektur für Bild-Klassifizierungen, mit dem Unterschied, dass die Eingaben konkateniert anstatt summiert werden.([Hua+16])

Allgemein spaltet sich die Architektur in *dense blocks* und Übergangs-Schichten (engl. *transition layers*). Der Aufbau der *dense blocks* ist wie oben beschrieben. Vor dem ersten *dense block* findet eine 7x7-Faltung auf dem Eingabefoto und ein 3x3-*max-pooling* statt. Innerhalb der *dense blocks* darf aufgrund der Konkatenations-Operation die Dimensionen der *feature maps* nicht verändert werden, weshalb hier keine *pooling*-Operationen stattfinden. Es wird lediglich die Tiefe bzw. Anzahl der Kanäle vergrößert. Die Auflösung der *feature maps* wird erst in den *transition layers*, die in der DenseNet-Architektur

immer auf einen *dense block* folgen, mittels *pooling* reduziert. In den *transition layers* findet jeweils nacheinander eine *batch*-Normalisierung, eine 1x1-*convolution* und ein 2x2-*average-pooling* mit Schrittweite 2 statt. Nach dem letzten *dense block* wird ein globales *average-pooling* und anschließend eine Softmax-Klassifizierung ausgeführt. Die gesamte Architektur wird in Abbildung 2.15 ersichtlich. ([Hua+16])

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112 × 112	7 × 7 conv, stride 2			
Pooling	56 × 56	3 × 3 max pool, stride 2			
Dense Block (1)	56 × 56	[1 × 1 conv; 3 × 3 conv] × 6	[1 × 1 conv; 3 × 3 conv] × 6	[1 × 1 conv; 3 × 3 conv] × 6	[1 × 1 conv; 3 × 3 conv] × 6
Transition Layer (1)	56 × 56	1 × 1 conv			
	28 × 28	2 × 2 average pool, stride 2			
Dense Block (2)	28 × 28	[1 × 1 conv; 3 × 3 conv] × 12	[1 × 1 conv; 3 × 3 conv] × 12	[1 × 1 conv; 3 × 3 conv] × 12	[1 × 1 conv; 3 × 3 conv] × 12
Transition Layer (2)	28 × 28	1 × 1 conv			
	14 × 14	2 × 2 average pool, stride 2			
Dense Block (3)	14 × 14	[1 × 1 conv; 3 × 3 conv] × 24	[1 × 1 conv; 3 × 3 conv] × 32	[1 × 1 conv; 3 × 3 conv] × 48	[1 × 1 conv; 3 × 3 conv] × 64
Transition Layer (3)	14 × 14	1 × 1 conv			
	7 × 7	2 × 2 average pool, stride 2			
Dense Block (4)	7 × 7	[1 × 1 conv; 3 × 3 conv] × 16	[1 × 1 conv; 3 × 3 conv] × 32	[1 × 1 conv; 3 × 3 conv] × 32	[1 × 1 conv; 3 × 3 conv] × 48
Classification Layer	1 × 1	7 × 7 global average pool			
		1000D fully-connected, softmax			

[Hua+16]

Abbildung 2.15.: *Die Architektur des DenseNet*

3. Methodologie

3.1. Korpus-Wahl (Ursula Kundert)

Wir wollen mit unserem Modell die Datierung von Psalmen-Handschriften im Bereich der Westkirche (heutiges West- und Mitteleuropa) abbilden. Dieser Kulturraum war durch die gemeinsame Kultus- und Bildungssprache Latein gut vernetzt und tauschte seit der Antike rege Handschriften zwischen den Regionen aus, meist um sie woanders abzuschreiben. Die **Grundgesamtheit** (engl. *population*) zu unserer Frage bilden deshalb alle Psalmen-Handschriften des westkirchlichen Abendlandes. Damit wird das Modell für Handschriften aus anderen Weltregionen nicht geeignet sein. Die Exploration hat das Ziel herauszufinden, ob das Modell an einer relativ kleinen Stichprobe daraus trainiert werden kann. Denn je größer die Stichprobe sein muss, je weniger lohnt sich natürlich der Trainingsaufwand, weil dann keine Digitalisate übrig bleiben, die das trainierte Modell, wenn es fertig ist, datieren kann. Um unserem Modell auch bei kleiner Stichprobe gute Erfolgschancen zu geben, wählten wir das Korpus in der Form einer **geschichteten Zufallsstichprobe** aus der Grundgesamtheit (engl. *stratified sampling*). Psalmen sind bis zur Reformation in abnehmender Häufigkeit hauptsächlich in Gebetbüchern, Psalterien, Psalmenkommentaren und Vollbibeln überliefert. Diese Häufigkeitsverteilung haben wir berücksichtig bei der Ziehung der Stichprobe.

Bereits auf der Ebene der Stichprobe trafen wir aber auch mehrere Maßnahmen, um **Verzerrung** (engl. *bias*) zu verhindern: Die Beschränkung auf eine einzige Sprache hätte eine starke Verzerrung (engl. *bias*) in unsere Stichprobe hineingetragen, weil es Nationalstaaten in Mittelalter und früher Neuzeit noch nicht gab und deshalb von einer Situation der **Mehrsprachigkeit** ausgegangen werden muss, wie sie historisch und geografisch sowieso die sprachliche Normalsituation darstellt. In unser Korpus haben wir deshalb die Digitalisate ein- und mehrsprachiger Handschriften unterschiedlicher Sprachen aufgenommen. Da wir zudem keine Datierung erreichen wollten, die sich an einer immer schwer zu bestimmenden literaturgeschichtlichen Abfolge von Werken orientiert, haben wir uns auf **ein einziges Werk** beschränkt, zu dem viele Datierungen anhand der Kolophone und der Schrift vorliegen, und damit die inhaltliche Variation (in Form eines Zeicheninventars), die der Algorithmus zu sehen bekommt, reduziert. Damit trainieren wir anhand des Werks 'Buch der Psalmen', welches das **meistüberlieferte Werk der Menschheit** darstellt. Wir erhoffen uns dadurch eine größere Relevanz und Nachnutzbarkeit unserer Exploration. Denn es ist davon auszugehen, dass alle geistlichen und die meisten weltlichen Schreiber*innen dieses Werk sogar mehrfach abgeschrieben haben, sodass wohl unser Training mit keinem oder wenig zusätzlichem Trainingsaufwand auch für andere Werke nutzbar wird. Zur Verminderung des Verzerrung hatten wir zunächst sogar nur die jeweilige Seite berücksichtigt, auf der Psalm 2 "*quare fremuerunt*" (bzw. seine volkssprachige Übersetzung) beginnt. Weil die erhebliche

Erweiterung des Datensatzes so zeitlich aber nicht zu erreichen war, ergänzten wir jeweils diese Seite um die zwanzig darauffolgenden. Durch beide Stellen-Wahlen ist praktisch ausgeschlossen, dass Bilder mit einer **ganzseitigen Initiale** in die Stichprobe geraten, was sonst möglicherweise dazu geführt hätte, dass unser Algorithmus sich zu sehr an den B- oder S-Initialen des Psalter-Beginns orientiert hätte.

Die Verwendung von Digitalisaten verschiedener Bibliotheken dient der Vermeidung von **Rauschen**, indem wir dadurch verhindern, dass die Lichtqualität des bibliothekstypischen Fototisches bei der trainierten Vorhersage eine Rolle spielt. Der von uns programmierte Algorithmus wählt aus dem so bestimmten Korpus eine einfache **Zufalls-Stichprobe** an Beispielpaaren für die Bearbeitung in Training und Test (engl. *batch*). Bei der relativ kleinen Anzahl an Beispielpaaren, vor allem in Phase I unseres Trainings, könnte dieses Verfahren allerdings heikel sein, da es empfindlich auf zu kleine Mengen reagiert.

3.2. Datenbeschaffung (Ursula Kundert)

Als Trainingsdaten für dieses Projekt auf der Seite der Eingabe-Daten werden die als Bild eingescannten oder fotografierten Seiten (im Folgenden: **Digitalisate**) der Handschriften verschiedener Jahre aus staatlichen Bibliotheken Europas verwendet. Die Digitalisate sind gut zugänglich, allerdings **über unterschiedlichste Repositorien verstreut**, sodass wir die Datenbeschaffung von Hand vornehmen mussten. Systematisch aus unserem Korpus heraus fällt jedoch Psalmen-Überlieferung in Gebetbüchern, weil diese oft eng gebunden sind und deswegen **nicht digitalisiert werden können**. Sobald Digitalisate auch dieses Buchtyps, etwa durch Trigonometrie mit kleinen Kameras, zur Verfügung stehen und automatisch datiert werden sollen, wird deshalb ein zusätzliches Training nötig sein. Die *Koninklijke Bibliotheek* im Haag und die *Koninklijke Bibliotheek* in Brüssel stellen ihre Digitalisate **nur gegen Bezahlung** zur Verfügung. Deswegen fallen sie aus unserem Korpus heraus. Das ist deswegen besonders ungünstig, weil die Niederlande im Spätmittelalter und in der frühen Neuzeit ein Zentrum der Produktion geistlicher Bücher sind.

3.3. Datenpräparation (Ursula Kundert, Tom Magnus Petersen)

Wir bilden geordnete Beispielpaare aus Digitalisat und *label*. Wir erstellen dafür jeweils eine CSV-Datei mit den Pfaden zu den Digitalisaten einerseits und den *labels* andererseits (s. unten). Die Digitalisate teilen wir dabei in zwei Gruppen ein: "**vor 1452**" und "**nach 1452**". Bei beiden achten wir peinlich darauf, nur zuverlässige Datierungen als Trainingsmaterial auf der Seite der Ausgabe-Daten zu verwenden. Das heißt in unserem Fall, dass die Digitalisate von wissenschaftlichen Bibliotheken bzw. von der philologischen Forschung datiert wurden.

Die **Ausgewogenheit** zwischen den Datenmengen für die beiden Zeiträume war nicht ganz einfach zu erreichen, da die Handschriften-Repositorien oft ausschließlich

auf mittelalterliche Handschriften ausgerichtet sind und Handschriften nach dem disziplinären **Epochenbruch**, der meist um 1492, 1500 oder 1517 angesetzt wird, nicht mehr berücksichtigen. Solche Digitalisate waren deshalb schwerer zu finden. Einige unwichtige Merkmale haben wir auch in diesem Schritt entfernt, um Rauschen zu vermeiden: Da viele Digitalisate schwarze **Ränder** oder Lineale oder Fußzeilen mit Angabe der Bibliothek enthalten, haben wir die Bilder zurechtgeschnitten auf den Schriftspiegel einschließlich der Marginalien. Dadurch geht allerdings ein wichtiges Merkmal für die Datierung verloren: die Breite des Randes im Buch. Diese ist abhängig vom Preis des Beschreibstoffes (Pergament oder Papier) und variiert deshalb in der Zeit. Für die Frage, ob der Text vor oder nach 1452 geschrieben wurde, ist der Papierpreis vermutlich zu vernachlässigen. Da sowohl die Handschriften- als auch die Druckproduktion nach 1452 erheblich zunehmen, kann es allerdings sein, dass es zu einer Verknappung des Papiervorkommens kam und unsere Annahme nicht stimmt, zu beachten hierbei ist, dass Informationen über die Herstellung zwischen dem 13. und 17. Jahrhundert aufgrund von Handelsgeheimnissen [Hun11] spärlich sind, wobei die Industrie über den hier relevanten Zeitraum stieg [Bar18] und der Preis eines Buches in Europa hingegen weitestgehend gleich blieb[Dye89] . Auch könnte die **Größe der Bilddatei** einen Einfluss auf die Leistung des Lern-Programms haben, da größere Bildauflösungen die Genauigkeit auf Kosten geringerer *Batch-Sizes* erhöht [SS20] und Variationen in Bildmaßen (engl. *scale-variance*) "nicht-trivial" behandelbar sind [NP17].

3.4. Modell (Tom Magnus Petersen und Tanja Schlanstedt)

Zur Durchführung unseres Experiments planen wir zunächst die Datensätze zu standardisieren, d.h. die einzelnen Digitalisate nach händischer Präparation für die Eingabe in den Algorithmus angleichend vorzubereiten, hierfür bietet sich zum Beispiel die Python-Bibliothek „Albumentations" [Bus+20] an. Ferner auszuwählen sind Framework, Architektur, sowie Eingabeparameter wie *Batch*-Größe, i.e. die Menge an Bildern aus unserem Datensatz, die pro Iteration des Algorithmus verwendet wird. Als Ausgabewerte erwarten wir ein Label (Null oder Eins) sowie Informationen über die Effektivität der gewählten Architektur (*loss-function*). Abhängig von diesen Parametern müssen wir ggf. unseren Datensatz um Bilder erweitern, die nicht den in Datenpräperation genannten Anforderungen genügen, da CNNs von großen Datensätzen profitieren [KC20].

Auf der Grundlage der Schichten-Erklärung von Rohrer 2017 haben wir im weiteren Verlauf verschiedene Architekturen verglichen. Zuerst haben wir das LeNet untersucht, eines der ersten CNNs, das vom Turing Award Preisträger Yann LeCun vorgestellt wurde. Es dient hauptsächlich der Erkennung handgeschriebener Zahlen in Bildern und wird auch heute noch für die Verarbeitung von handschriftlichen Einzahlungen verwendet ([Zha+20a] Kap. 6.6). In unserem Fall war es vorteilhaft, LeNet zur Verdeutlichung der Funktionsweise einer Faltung zu benutzen. Um hochauflösende Bilder wie die Psalmen-Digitalisate zu testen, brauchen wir jedoch mehr als die fünf *convolutional layers* der LeNet-Architektur, da sich aus der Entwicklung immer größerer Netzwerke herausgestellt hat, dass **mehr Schichten** aufgrund der steigenden Komplexität mehr Merkmale integrieren und somit eine **höhere Genauigkeit** erzeugen können. Neben

LeNet haben wir deshalb das **ResNet** in Betracht gezogen, das wegen seines einfachen Designs und der Fähigkeit, viele Schichten zu integrieren ohne das verschwindende Gradientenproblem auszulösen, beliebt ist. Dieses wird durch Verbindungen, mit denen Schichten übersprungen werden, ergänzt ([He+15]). Diese sogenannten *identity shortcuts* tragen zwar dazu bei, bestimmte Merkmale zu erhalten, jedoch **verringern** sie gleichzeitig die **Repräsentationskraft** des Netzwerkes. Des Weiteren ist problematisch, dass die Schichten durch Summation kombiniert werden, da dadurch der **Informationsfluss** im Netzwerk **behindert** wird. Allgemein sind Architekturen, die linear verlaufen, wie Algorithmen mit Zuständen, die von einer Schicht zur nächsten verändert werden ([Zha+20b]). Die Veränderung des Zustands erlaubt nicht, bereits **gelernte Merkmale ohne Informationsverlust zu erhalten**, was vor allem bei einem **kleinen Datensatz** wie dem unsrigen gefragt ist.

Das vorher beschriebene **DenseNet** (siehe Kap. 2.9) kann genau dieses Problem lösen. Die Architektur des DenseNets mit seinem **dichten Konnektivitätsmuster** erlaubt es, explizit zwischen *feature maps*, die dem Netzwerk hinzugefügt werden, und denen, die schon gewonnen wurden, zu unterscheiden. Da alle Schichten auf die bereits gelernten *feature maps* der vorhergehenden Schichten zugreifen können, ist die Wiederverwendung bereits erlernter Merkmale möglich, wodurch das Modell **weniger Parameter** benötigt und folglich kompakter und **effizienter** wird. Auch die **geringe Wachstumsrate** begünstigt die **Minimierung der Parameteranzahl**. Ein weiterer Punkt, der für unsere Entscheidung ausschlaggebend war, ist die bemerkenswert geringere Fehlerrate beim DenseNet bei wachsender Schichtenanzahl N und Erhöhung der Wachstumsrate k im Vergleich zum ResNet. Die Repräsentationsfähigkeit des ResNet nimmt bei steigender Komplexität auch zu, erzeugt allerdings schneller **Überanpassungsprobleme**, da es nicht derart effizient mit den Parametern umgeht. Erst bei sehr tiefen (>250 Schichten) Modellen gibt es keine Verbesserungen zum Gegenstück. ([Hua+16]) Problematisch bei unserer Architektur ist laut [Zha+20a] 2020, dass ein **größerer GPU-Speicher** durch den Konkatenationsprozess nötig ist. Zwar gibt es Speicher-effizientere Implementierungen, jedoch sind diese komplexer und **erhöhen** hinzukommend die **Trainingszeit** um etwa 20%. Der Hauptgrund für die Erhöhung der Trainingszeit sind die kleinen Faltungen im Netzwerk, die langsamer auf der GPU arbeiten als kompakte, große Faltungen. Die hohe Trainingszeit hat sich auch bei uns bestätigt.

4. Durchführung

4.1. Welches Framework? (Emma Calewaert, Ursula Kundert)

Es gibt zwei weit verbreitete Frameworks: PyTorch und TensorFlow. PyTorch wurde 2016 durch das *AI Research Lab* von Facebook veröffentlicht und verlässt sich, wie man am Namen schon vermutet, auf die Programmiersprache Python, hat aber auch eine Schnittstelle für C++. TensorFlow hat Schnittstellen für viele Programmiersprachen. Diese zwei *frameworks* ähneln einander immer mehr, weil sie Funktionalitäten ihrer Wettbewerber integrieren. Es gibt jedoch immer noch eine Spaltung in zwei Nutzer-Gemeinschaften: PyTorch ist oft das bevorzugte Framework für maschinelles Lernen in der Forschung, TensorFlow ist hingegen in der Produktion weiter verbreitet.([Wel20]) Wir haben uns entschieden, PyTorch zu verwenden. Der wichtigste Grund sind PyTorchs **leichte Benutzung** und die Möglichkeiten zur **feineren Einstellung**, die sich gut eignen für die Nutzung bei schnelleren, kleinen Modellen. Dazu kommt auch noch, dass wir uns als Entwickler schon besser mit Python auskennen.

4.2. Erstellen einer CSV-Datei (Dennis Lehmann)

Ziel war es die Daten dahingehend zu klassifizieren, ob sie vor oder nach 1452 verfasst wurden. Dafür mussten sie zunächst in ein Format umgewandelt werden, das durch ein neuronales Netz verarbeitet werden kann. Dafür wurde das CSV-Format gewählt. Die Implementierung des Algorithmus', welcher die CSV-Dateien (*comma separated values*) erstellt, ist im Jupyter Notebook „CSVCreator" zu finden.

Bei den Daten handelt es sich um Bilder im png/jpg-Format verschiedener Größen und Farbspektren unterschiedlicher Bit-Tiefe. Bei der Suche nach einer geeigneten Form, um die Daten *labeln* und zu speichern, haben wir zwei Möglichkeiten in Betracht gezogen: Die Bilder könnte man durch Aufteilung des Bildes in die einzelnen Farbkanalmatrizen speichern. Jede Matrix würde somit einen der drei RGB-Farbkanäle repräsentieren und jeder Eintrag in einer solchen Matrize würde dann für ein Drittel eines Pixels im Bild stehen. Nach reiflicher Überlegung haben wir uns gegen diese Vorgehensweise entschieden, um die Anwendung **performant in Hinsicht auf den Speicherbedarf** zu halten. Stattdessen meistert unser neuronales Netz diese Herausforderung dadurch, dass die **Umwandlung erst beim Laden der Resource** stattfindet und so eine redundante Speicherung prozessierter Daten vermeidet:

```
for i, image_path in tqdm(enumerate(image_paths), total=len(image_paths)):
    label = image_path.split(os.path.sep)[-2]
    data.loc[i, 'image_path'] = image_path
    labels.append(label)
```

Listing 4.1: *Pfad Speicherung der Bilder*

Da es sich bei unserem Projekt um ein binäres Klassifizierungsproblem handelt, können die Ergebnisse durch eins oder null dargestellt werden. Das Labeln der Trainings- und Test-Daten erfolgt nach händischer Klassifizierung. Dafür wurden die positiv (1, vor 1452) und negativ (0, nach 1452) klassifizierten Daten in separaten, dafür vorgesehenen Ordnern abgelegt. Wir haben die Digitalisate danach **anhand ihres Speicherortes** automatisiert mit *labels* versehen.

Dies ist durch eine IF-Abfrage innerhalb einer FOR-Schleife geschehen.

```
for i in range(len(labels)):
    if(labels[i]== "vor"):
        index=1
    else:
        index=0
    data.loc[i, 'target'] = int(index)
```

Listing 4.2: *Labeln der Digitalisate*

Exemplarischer Ausschnitt nach Ausführen des CSV-Creators von der CSV Datei:

$image_path$	target
0../imageTraining/vor/Muenchen$_B$ayerischeStaats...	1.0
1../imageTraining/vor/Oxford$_B$odleianLibrary$_4$2...	1.0
2../imageTraining/nach/Karlsruhe$_B$adischeLandes...	0.0
3../imageTraining/nach/HannoverschMuenden$_B$ayer...	0.0
4../imageTraining/vor/Heidelberg$_U$B$_S$ignatur$_S$e...	1.0

Beim Programmieren dieser Jupyter Notebook „CSVCreator“ wurden wir inspiriert durch [Rat20].

4.3. Laden der CSV-Datei (Data-Loader) (Dennis Lehmann)

Damit die Daten der CSV Datei auch vom DenseNet gelesen werden können, musste ein *data loader* geschrieben werden. PyTorch stellt dafür bereits eine dafür geeignete Klasse (`Dataset`) zur Verfügung, zu finden unter `torch.utils.data`. Damit man die Funktionen für seinen Anwendungsfall nutzen kann, muss man eine Klasse schreiben, die von `Dataset` erbt. Um sie in diesem Kontext nutzen zu können, muss man unter anderem die `__init__`-Methode überschreiben. Einmalig übergibt man dann die Pfade des Bildes (`X`) in Form einer Liste mit den dazugehörigen Labeln (`y`). Weiterhin muss die Methode `__getitem__` überschrieben werden. Der Methode muss ein Index übergeben werden.Mit Hilfe des Index erhält man aus `X` den Pfad der Ressource und öffnet das dort hinterlegte Bild. Mithilfe der `PIL`-Bibliothek wandelt man dieses nun in oben genannte Matrix-Repräsentation um.

```
image = Image.open(self.X[index]).convert('RGB')
```

Listing 4.3: *Umwandel des Bild zu einer Matrix*

Um die Bilder hinsichtlich ihres Formats zu standardisieren, benutzt man die `transform`-Methode (zu finden bei `torchvision`). Mit deren Hilfe werden in unserem Beispiel die Bilder auf ein Format von 224x224 Pixel formatiert und zentriert. Nach dem Formatieren lässt man sich noch mithilfe des Index das dazugehörige Label geben und speichert es in einem Tensor (einer mathematischen Speicherstruktur), damit Bild und Label vom neuronalen Netz prozessiert werden können. Zudem wird auch die Methode `_len_` überschrieben, diese sollte die Anzahl der Bilder zurückgeben, die geladen wurden.

4.4. Konstruktion des neuronalen Netzes (Dennis Lehmann)

Wie oben erläutert, haben wir uns für unsere Untersuchung für ein DenseNet entschieden. Das DenseNet wurde auf der Grundlage von [Zha+20a], 292-299 wie folgt im Jupyter Notebook `DataLoderAndDenseNet` implementiert (Abbildung 4.5): Für die unterste Ebene des DensNets wurde die Funktion `conv_block` implementiert. Diese Ebene bekommt als Parameter die `input_channels` übergeben, welche angeben, wie viele Eingabe-Kanäle der `conv_block` bearbeiten muss. Dabei gibt `num_channels` an, wie weit verzweigt die Ausgabe für nachgeschaltete Ebenen sein soll. Der `conv_block` (Abbildung 4.1) verbindet eine `BatchNorm2d` mit einer Schicht `ReLU` (Rectified Linear Unit) und einem `Conv2d`-Schicht.

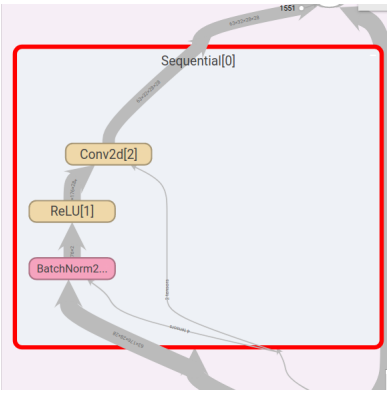


Abbildung 4.1.: *Ansicht aus Tensor Board zeigt conv_ block*

Alle drei Elemente sind Funktionen von PyTorch aus dem `torch.nn`-Bereich. Diese Funktion ist einer der Grundbestandteile für den `DenseBlock`, der die einzelnen dense layers verbindet (Abbildung 4.2). Der `DenseBlock` fasst die `conv_block` zusammen zu einer Schicht. Der `DenseBlock` wird als Klasse geschrieben und erbt von `nn.Module`, was beim Training ermöglicht, dass durch die Backpropagation die Gewichtungen (engl. *weigths*) und Verzerrungen (engl. *biases*) auf den neuen Stand gebracht werden können. Bei der Initialisierung der Klasse werden die Parameter

`num_conves`, welches die Anzahl der `conv_block` im `DenseBlock` repräsentiert, neben `input_channels` und `num_channels` übergeben. Diese Parameter haben die gleiche Aufgabe wie die Parameter im `conv_block` . In der `__init__` werden mehrere `conv_block` mittels `nn.Sequential` nacheinander geschaltet. Bei jedem `conv_block` ändert sich der `input_channels`-Parameter um die Addition der Werte der `input_channels` und des Produkts aus `num_channels` und dem Platz des `DenseBlock`. Außerdem musste noch eine `forward`-Funktion geschrieben werden, in der das Ergebnis des `DenseBlock` mit den Eingabe-Werten konkateniert wurde, bevor sie zurückgegeben wurden.

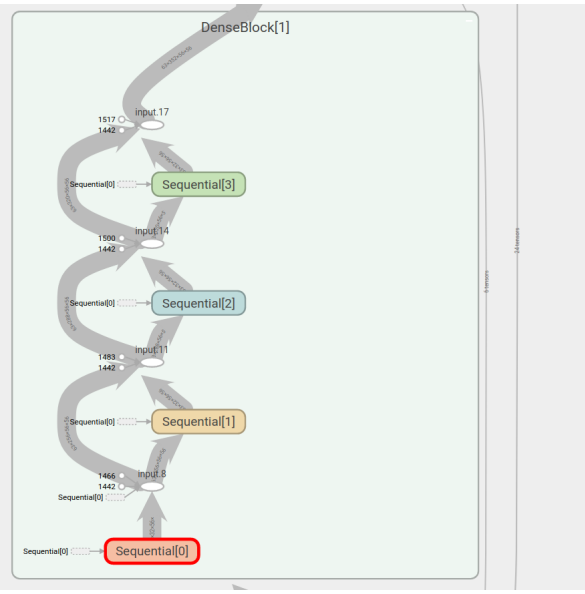


Abbildung 4.2.: *Ansicht aus Tensor Board zeigt den DenseBlock.*

Als Übergang zwischen zwei *dense blocks* benötigt man für die Regulierung der Komplexität des Modells ([Zha+20a], S. 294) einen `transition_block` (Abbildung 4.3). Dieser bekommt als Übergabeparameter wieder `input_channels` und `num_channels`. Er vereint die Elemente `BatchNorm2d` mit einer `ReLU`-, einer `Conv2d`- und einer *AvgPool2d*-Schicht. Die *AvgPool2d*-Schicht ist zur Regulierung gedacht.

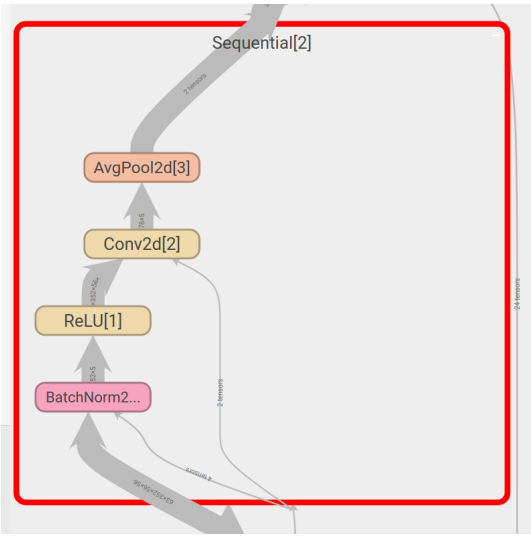


Abbildung 4.3.: *Ansicht aus Tensor Board zeigt transition Block.*

Die *dense blocks* und die *transition blocks* werden mithilfe einer FOR-Schleife zusammengefügt (Abbildung 4.4). Mit der IF-Abfrage wird überprüft, ob die angestrebte Größe des Blockes schon erreicht ist; wenn das nicht der Fall sein sollte, wird nach einem `DenseBlock` erneut ein `transition_block` eingefügt.

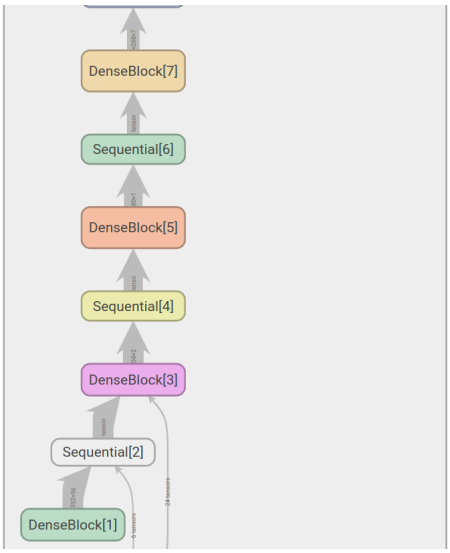


Abbildung 4.4.: *DenseBlock und transition Block, Ansicht aus Tensor Board des eigenen Modells*

Als letzter Schritt werden **alle Elemente zum DenseNet zusammengefügt**. Das geschieht durch Nutzung eines Aufrufs der `nn.Sequential`-Methode: Den Anfang bildet eine Schicht, deren Details für uns nicht von Belang sind, danach folgt der Block mit den *dense* und *transition blocks* und anschließend eine `AdaptiveMaxPool2d`-Schicht. Diese letzte Schicht wird gefaltet und abschließend folgt eine lineare Schicht, welche die Vorhersage als Ergebnis ausgibt. Dazu müssen die Werte, welche das neuronale Netz produziert, noch eine `Softmax`-Funktion durchlaufen, damit die Ergebnisse in Werte mit statistischer Aussagekraft transformiert werden.

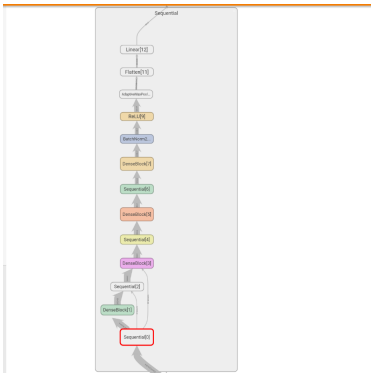


Abbildung 4.5.: *Ansicht aus Tensor Board komplettes Model*

4.5. Trainieren des DenseNet (Emma Calewaert, Iuliia Chelysheva)

Die Idee des Trainieren basiert darauf, das Netz mit mehreren Durchläufen oder Epochen zu verbessern, indem wir zwischen den Epochen die Variablen angemessen anpassen. In jeder Epoche testen wir mit einer kleinen Stichprobe (engl. *minibatch*) aus dem Datensatz und bekommen eine Vorhersage zurück. Mit diesen Vorhersagen, können wir mit der *loss*-Funktion die Kosten berechnen. Die Kosten eines Netzes zeigen, wie groß der Abstand zwischen die realen und die vorhergesagten Werten ist. In unserem Netz benutzen wir hierzu die Kreuz-Entropie-Funktion.([Zha+20a], Kap. 3.2) Der nächste Schritt besteht darin, die Kosten in der nächsten Epoche zu minimieren. Hierzu benutzen wir die explorativen Methode des Gradienten-Abstiegsverfahren mit kleinen Stichproben (*minibatch stochastic gradient descent*).([Zha+20a], Kap. 3.2, vgl. Abbildung 4.6)

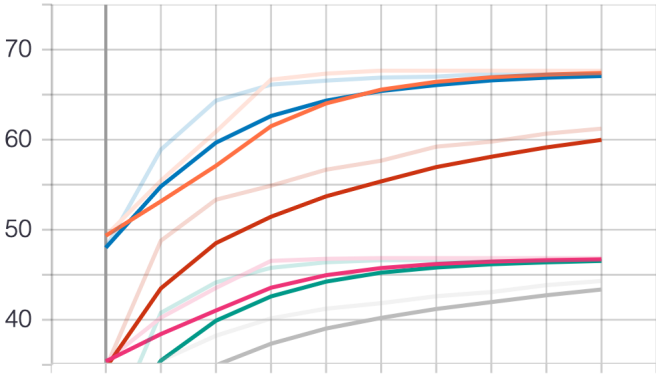


Abbildung 4.6.: *training accuracy Phase II*

Dieses Training wird mithilfe von Tensorboard visualisiert. Folgende Grafik zeigt die durch Training erreichte Genauigkeit (engl. *training accuracy*) in Funktion der Epoche für verschiedene Durchläufe mit unterschiedlichen Startwerten. Es ist ersichtlich, dass die Genauigkeit mit der oben genannten Methode besser wird. Die Grafik von Abbildung 4.7 wurde auf der Grundlage des ursprünglichen Datensatzes erstellt.

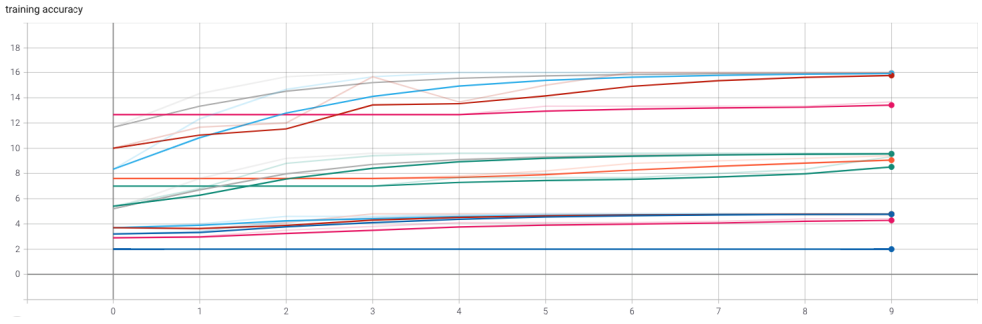


Abbildung 4.7.: *training accuracy Phase I*

Danach haben wir nochmal mit einem größeren Datensatz trainiert (der Grund dafür folgt bei der Evaluation). Auf Abbildung 4.8 sehen wir, dass die Genauigkeit mit einem größeren Datensatz schneller zunimmt.

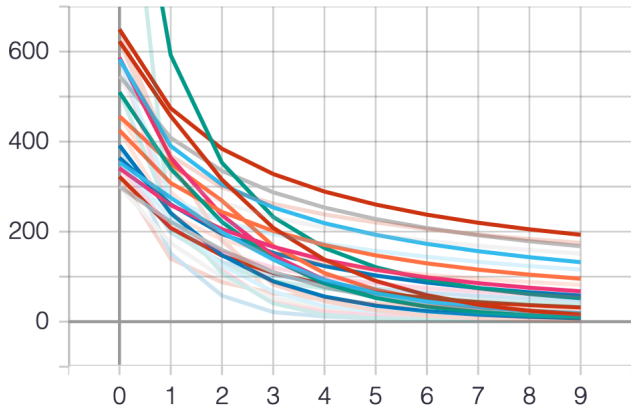


Abbildung 4.8.: *Loss Grafik Phase III*

4.6. Evaluation des jeweiligen Netzes (Emma Calewaert)

Wir benutzen für die Evaluation unseres Netzes eine Wahrheits-Matrix (engl. *confusion matrix*). Diese Matrix wird bei solchen Klassifizierung-Problemen wie unserem benutzt, um die Verlässlichkeit der Vorhersage zu beurteilen. Die Spalten repräsentieren die möglichen Werte der Vorhersagen, und die Zeilen stellen die möglichen Werte der tatsächlichen Werte dar. In den Zellen steht die Anzahl der Daten, für welche die Kombination aus der jeweiligen Spalte und Zeile zutrifft.([Ped+12], Kap. 3.3.)

Die erste Diagonale sind die richtig vorhergesagten Werte: richtig negative und richtig positive. Diese repräsentieren die Werte, bei denen der tatsächliche Wert mit dem vorhergesagten Wert übereinstimmt. Die zweite Diagonale zeigt dann die falschen Vorhersagen: die falsch positiven und die falsch negterativen. Die Wahrheits-Matrix liefert auch eine Aussage über die Genauigkeit und über das Maß an Fehlklassifikation. Die Genauigkeit zeigt, wie oft der Klassifikator richtig funktioniert hat: die Anzahl der richtig positiven Vorhersagen plus die Anzahl der richtig negativen geteilt durch die Anzahl der Datensätze. Das Maß für die Fehl-Klassifizierung ist das Umgekehrte und zeigt somit, wie oft der Klassifikator falsche Ergebnisse liefert.([Mar14]) Die Klassfikationsmaße der Verlässlichkeit (engl. *accuracy*), Trefferquote (engl. *recall*) und Genauigkeit (engl. *precision*) beeinflussen sich gegenseitig, deshalb ist ein kombinierter Index das bessere Maß für die Qualität unseres trainierten Netzwerks. Ein solcher kombinierter Index stellt der F1-Wert dar, der wie folgt berechnet wird:

$$F1 - Wert = \frac{2xGenauigkeitxTrefferquote}{(Genauigkeit+Trefferquote}$$

In unserem Training haben wir eine Genauigkeit von 0.8125. Aus der Matrix kann man besser ablesen, wie die Verteilung aussieht.(vgl. Abbildung 4.9)

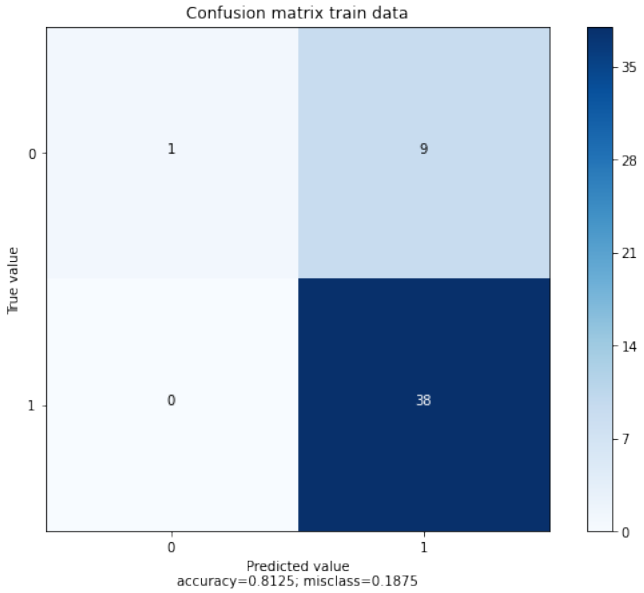


Abbildung 4.9.: *confusion matrix*

Unser Model hat nicht für alle Daten den Wert eins vorhergesagt, sondern auch mal den Wert null geliefert. Hieraus kann man ableiten, dass das Modell schon einigermaßen trainiert ist. Der Datensatz ist relativ klein, und die Klasse von Daten, die vor 1452 entstanden ist, ist größer, als die Klasse von Daten, die nach 1452 entstanden sind. Die Vermutung ist, dass diese zwei Faktoren zu einem schlechteren Modell geführt haben. Dies können wir auch in der Wahrheits-Matrix der Test-Daten nachverfolgen.(vgl. Abbildung 4.10)

4.7. Phase I

In unserem Training haben wir eine Genauigkeit von 0.8125. Aus der Matrix kann man besser ablesen, wie die Verteilung aussieht.(vgl. Abbildung 4.9) Unser Model hat nicht für alle Daten den Wert eins vorhergesagt, sondern auch mal den Wert null geliefert. Hieraus kann man ableiten, dass das Modell schon einigermaßen trainiert ist. Der Datensatz ist relativ klein, und die Klasse von Daten, die vor 1452 entstanden ist, ist größer, als die Klasse von Daten, die nach 1452 entstanden sind. Die Vermutung ist, dass diese zwei Faktoren zu einem schlechteren Modell geführt haben. Dies können wir auch in der Wahrheits-Matrix der Test-Daten nachverfolgen.(vgl. Abbildung 4.10)

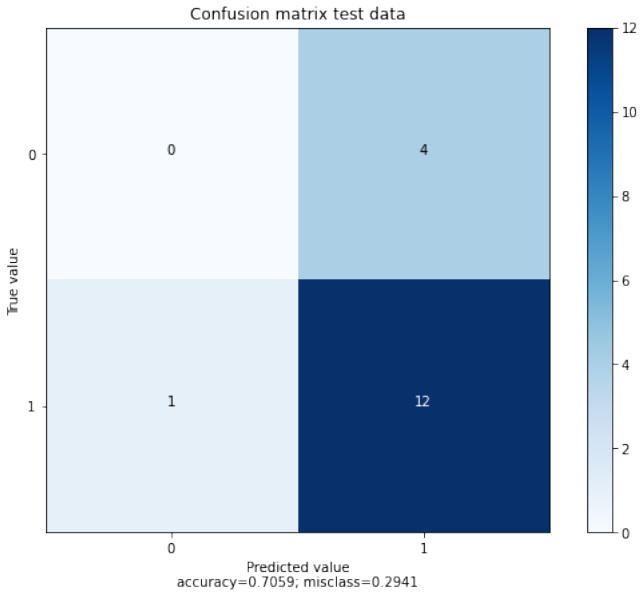


Abbildung 4.10.: *confusion-Matrix Phase I*

Verlässlichkeit (engl. *accuracy*): 70.59%
Trefferquote (engl. *recall*): $\frac{12}{13} = 92.30\%$
Genauigkeit (engl. *precision*): $\frac{12}{16} = 75\%$
F1: 0.8275

4.8. Phase II

Um diese Probleme zu beheben, brauchten wir einen ausgewogeneren und größeren Datensatz. Dazu sammelte unseres Team mehr Daten, mit welchen ein erneutes Training gestartet wurde. Mit unserem neuen Datensatz sieht die *confusion*-Matrix wie auf Abbildung 4.11 gezeigt aus.

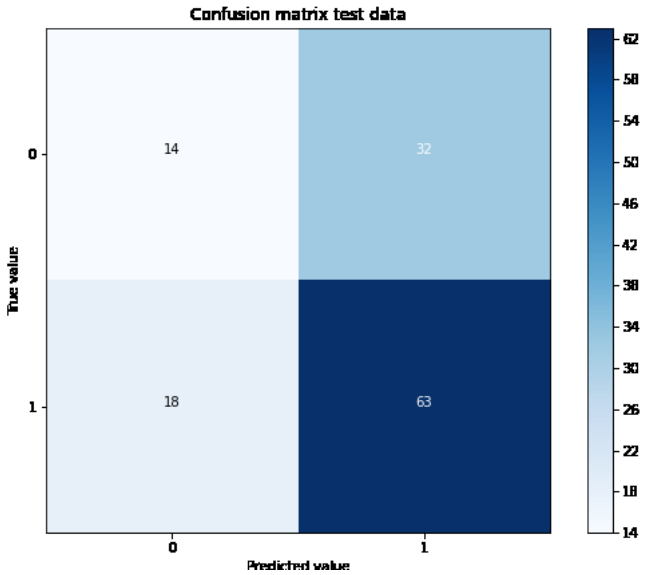


Abbildung 4.11.: *confusion-Matrix Phase II*

Damit erreichten wir folgende Werte:

Verlässlichkeit von 60.62%
Trefferquote (engl. *recall*): $\frac{63}{63} + 18 = 77.78\%$
Genauigkeit (engl. *precision*): $\frac{63}{63} + 32 = 66.32\%$
F1: 0.7159

Trotz der Erhöhung der Datenmenge auf 508 Beispielpaare hat sich die Qualität gesenkt. Wir vermuten, dass das durch die größere Verzerrung zustande kam dadurch, dass wir im Ordner “nach 1462” mehr Beispiele aus denselben Codices hatten als im anderen Ordner.

4.9. Phase III

Danach sammelte das Team noch mehr Daten. Hierauf haben wir auch nochmal trainiert. Hierbei war die Datensatz aber leider wieder weniger ausgewogen: Wir haben wiederum mehr Daten der Klasse 1. Die Genauigkeit und das Maß an Fehl-Klassifizierung sind ähnlich geblieben wie in Phase I.(vgl Abbildung 4.12)

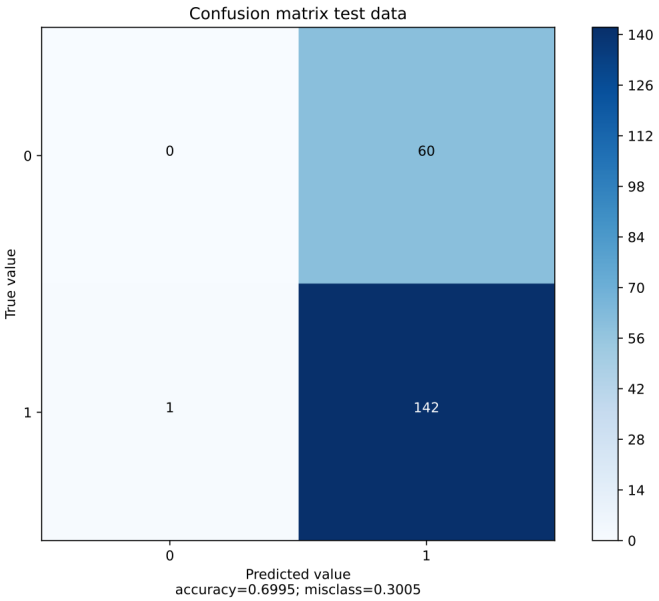


Abbildung 4.12.: *confusion-Matrix Phase II*

Verlässlichkeit von 69.95%
Trefferquote (engl. recall): 99.30%
Genauigkeit (engl. precision): 70.29%
F1: 0.8231

5. Diskussion (Gemeinsam)

Wir haben eine Trainierbarkeit feststellen können. Die Hinzunahme von zusätzlichen Daten hat die Ergebnisse nicht wesentlich verbessert: den besten F1-Wert von 0.8275 hat das Netzwerk in Phase I erreicht. Dieser höchste Wert ist mit dem kleinsten Datensatz zustande gekommen, die hohe Verlässlichkeit ist deshalb wohl eher zufällig. Das Training von Phase III war aber nicht signifikant schlechter. In einem wissenschaftlichen Umfeld, z. B. bei Metadaten für die philologische Forschung, ist allerdings eine sehr hohe Qualität der Datierung nötig. Unser Modell wäre geeignet für die philologische und medienhistorische Heuristik oder die Arbeitsteilung in einer Bibliothek zwischen Mitarbeitenden, welche nur Handschriften betrachten, und solchen, welche auch den Vergleich zu den Drucken ziehen. Auch bereits online verfügbare KI-basierte Dienstleistungen wie etwa die Bildähnlichkeitssuche der Bayerischen Staatsbibliothek gehen vorerst nicht über eine heuristische Funktion hinaus. Für die wissenschaftliche Klassifikation muss diese Grobsortierung durch die paläografische, buchgeschichtliche und historische Recherche im Rahmen einer menschlichen Handschriftenbeschreibung ergänzt werden. Die Vorhersagen für die Handschriften vor 1462 sind besser als für diejenigen danach.

Eine Limitierung des Ergebnisse könnte an der Hardware liegen, da wir unser Modell auf einem Grafikprozessor (GPU) oder Prozessor (CPU) trainiert haben. Dies schränkte den Umfang der Daten ein, die wir bearbeiten konnten, weil unsere Trainingszeit bis zu zwei Stunden dauerte und der Grafikspeicher die Stichproben-Größe auf hundert Bilder beschränkte. Mit einem Server oder Cluster hätten wir mehr Trainings-Versuche durchführen und verschiedene Hyperparameter trainieren können. Beispielsweise hätten wir auch eine höhere Wachstumsrate wählen können, da das Netzwerk so mehr Merkmale integrieren kann und folglich eventuell verlässlichere Ergebnisse geliefert hätte. Das hätte allerdings auch zu Überanpassungsproblemen führen können.

Andererseits kann es aber auch gut sein, dass unser Ergebnis genau zur medienhistorischen Diskussion passt, die seit Jahrzehnten diskutiert, ob es sich beim Ereignis der (Wieder-)Erfindung des Buchdrucks mit beweglichen Lettern um einen alles verändernden Paradigmenwechsel oder doch nur um einen graduellen Medienwandel handelt. Die Zunahme der Handschriftenproduktion, die Neddermeyer 1998 für das späte 15. Jahrhundert und frühe 16. Jahrhundert feststellt, und auf den Buchdruck wegen der größeren Verfügbarkeit von Vorlagen zurückführt, hätte dann nicht automatisch auch Formen aufgenommen, die auf die neue Technik zurückzuführen sind. Unser Ergebnis würde dann auch die buchhistorische These bestätigen, dass der Buchdruck zunächst eine (kostengünstigere) Imitation der Handschrift anstrebte, also die Anpassung in diese Richtung verlief.

6. Ausblick (Gemeinsam)

Um die Datierungfrage zu vertiefen ließen sich weitere Aspekte der Handschriftenüberlieferung des 15. Jahrhunderts betrachten. Die paläografische Forschung kannte bisher folgende Faktoren, die für die Datierung eine Rolle spielen könnten: Liniert (dann tendenziell vor 1452) Durchgezogene Schleifen bei den Schäften von b, d, h, l und k (sog. Schleifen-Bastarda, in den Niederlanden: Hybrida, oder Devoten-Bastarda) (nach 1420) Dazu gibt es eine Reihe von Merkmalen, die bisher nicht systematisch untersucht worden sind, wie Schneider 2014, 74 meint.[Sch14b] Dies betrifft die Buchstaben v, w, r, s-Formen und f. Durch die Kombination unserer allgemeinen Exploration mit einem spezifischen Training auf Buchstabenformen, könnte die Datierung verbessert werden.

Weiter kann das Projekt um eine automatische Datensammlung in Form einer Kombination eines Parsers und einem weiteren Algorithmus für die Erkennung von Handschriften erweitert werden. Hierfür sind sog. Meta-Plattformen wie *World Digital Libary* (World digital library) dienlich, da sie die Datenbanken mehrer Bibliotheken in einem HTML-Layout vereinigen. Dann ist aber eine automatische Bildauswahl nötig, die erkennt, ob es sich überhaupt um eine Textseite handelt, und die automatisch schwarze Ränder, Maßstäbe etc. wegschneidet. Das sind zusätzliche KI-Aufgaben, die den Rahmen unseres Projekts gesprengt hätten, da es schwer abzuschätzen ist, was alles vorkommen kann, weil jede Bibliothek wieder andere Digitalisierungs-Gepflogenheiten hat. Alternativ wäre es wünschenswert, dass solche Online-Datenbanken eine API zum Herunterladen der Daten nach den benötigten Parametern (wie z.B. Erstellungsjahr und Typ) zur Verfügung stellen.

Quellenverzeichnis

[201] *typical CNN architecture*. wikipedia.org. Dez. 2015. URL: https://de.wikipedia.org/wiki/Convolutional_Neural_Network.

[Aga17] Abien Fred Agarap. „An Architecture Combining Convolutional Neural Network (CNN) and Support Vector Machine (SVM) for Image Classification“. In: (10. Dez. 2017). arXiv: 1712.03541 [cs.CV].

[Bar18] Timothy Barrett. *European Papermaking Techniques 1300–1800*. Paper through Time: Nondestructive Analysis of 14th- through 19th-Century Papers. Juli 2018. URL: http://paper.lib.uiowa.edu/european.php.

[Bas19] Madhushree Basavarajaiah. *Maxpooling vs minpooling vs average pooling*. https://medium.com. Feb. 2019. URL: https://medium.com/@bdhuma/which-pooling-method-is-better-maxpooling-vs-minpooling-vs-average-pooling-95fb03f45a9.

[Bil] *Anfang des Psalters in einem Exemplar der 42-zeiligen Gutenberg-Bibel auf Papier (GW 4201, ISTC ib00526000)*. Ink B-408, Bl. 293r. URL: http://mdz-nbn-resolving.de/urn:nbn:de:bvb:12-bsb00004647-4.

[Bus+20] Alexander Buslaev u. a. „Albumentations: Fast and Flexible Image Augmentations“. In: *Information* 11.2 (2020). ISSN: 2078-2489. DOI: 10.3390/info11020125. URL: https://www.mdpi.com/2078-2489/11/2/125.

[Cha] Himadri Sankar Chatterjee. *A Basic Introduction to Convolutional Neural Network*. https://medium.com. URL: https://medium.com/@himadrisankarchatterjee/a-basic-introduction-to-convolutional-neural-network-8e39019b27c4.

[CK07] Martina Stercken in Verbindung mit Hg. Christian Kiening. *Medienwandel – Medienwechsel – Medienwissen*. Schriftenreihe des NCCR “Mediality. 2007-2015. URL: http://mediality.ch/publikationen_mw.php.

[DHS00] Richard O. Duda, Peter E. Hart und David G. Stork. „Pattern Classification“. In: Wiley John + Sons, 1. Okt. 2000. ISBN: 978-0471056690. URL: https://www.ebook.de/de/product/3244086/richard_o_duda_peter_e_hart_david_g_stork_pattern_classification.html.

[Dye89] Christopher Dyer. *Standards of Living in the Later Middle Ages*. Cambridge England New York: Cambridge University Press, 1989. ISBN: 9781139167697. DOI: https://doi.org/10.1017/CBO9781139167697.

[He+15] Kaiming He u. a. „Deep Residual Learning for Image Recognition“. In: (10. Dez. 2015). arXiv: 1512.03385 [cs.CV].

[Hua+16] Gao Huang u. a. „Densely Connected Convolutional Networks“. In: (25. Aug. 2016). arXiv: 1608.06993v5 [cs.CV].

[Hun11] Dard Hunter. *Papermaking*. DOVER PUBN INC, 1. März 2011, 233 ff. 688 S. ISBN: 9780486236193. URL: https://www.ebook.de/de/product/3302323/dard_hunter_papermaking.html.

[HW62] D. H. Hubel und T. N. Wiesel. „Receptive fields, binocular interaction and functional architecture in the visual cortex“. In: *The Journal of Physiology* 160.1 (1962), S. 106–154. DOI: 10.1113/jphysiol.1962.sp006837.

[KC20] Ibrahem Kandel und Mauro Castelli. *The effect of batch size on the generalizability of the convolutional neural networks on a histopathology dataset*. Bd. 6. 4. Elsevier BV, 2020, S. 312–315. DOI: https://doi.org/10.1016/j.icte.2020.04.010.

[Mar14] Kevin Markham. *Simple guide to confusion matrix terminology*. März 2014. URL: https://www.dataschool.io/simple-guide-to-confusion-matrix-terminology/.

[McL62] Marshall 1911-1980 McLuhan. *The Gutenberg galaxy : the making of typographic man*. eng. Toronto, Can. : University of Toronto Press, 1962, 293 S.

[Mis] MissingLink.ai. *Convolutional Neural Network Architecture: Forging Pathways to the Future*. MissingLink.ai. URL: https://missinglink.ai/guides/convolutional-neural-networks/convolutional-neural-network-architecture-forging-pathways-future/.

[NP17] Nanne van Noord und Eric Postma. *Learning scale-variant and scale-invariant features for deep image classification*. Bd. 61. Elsevier BV, 2017, S. 583–592. DOI: https://doi.org/10.1016/j.patcog.2016.06.005.

[Ped+12] Fabian Pedregosa u. a. „Scikit-learn: Machine Learning in Python“. In: *Journal of Machine Learning Research (2011)* abs/1201.0490 (2. Jan. 2012). arXiv: 1201.0490 [cs.LG]. URL: http://arxiv.org/abs/1201.0490.

[Rat20] Sovit Ranjan Rath. *Creating Efficient Image Data Loaders in PyTorch for Deep Learning*. Apr. 2020. URL: https://debuggercafe.com/creating-efficient-image-data-loaders-in-pytorch-for-deep-learning/.

[Sch14a] Jürgen Schmidhuber. „Deep Learning in Neural Networks: An Overview“. In: *CoRR* abs/1404.7828 (2014). arXiv: 1404.7828. URL: http://arxiv.org/abs/1404.7828.

[Sch14b] Karin Schneider. *Paläographie und Handschriftenkunde für Germanisten*. 3., durchgesehene Auflage. Sammlung kurzer Grammatiken germanischer Dialekte Nr. 8. De Gruyter, 2014. DOI: 10.1515/9783110338676.

[SS20] Carl F. Sabottke und Bradley M. Spieler. *The Effect of Image Resolution on Deep Learning in Radiography*. Bd. 2. 1. Radiological Society of North America (RSNA), 2020, e190015. DOI: https://doi.org/10.1148/ryai.2019190015.

[Sum20] Dan Summer. *BI- und Daten-Trends 2021. Der große digitale Umbruch. Webinar über Geschäfts-Analytik, Qlik Tech GmbH Düsseldorf*. Dez. 2020.

[Wel20] Stephen Welch. *Pytorch [sic] vs. TensorFlow. What You Need to Know*. Mai 2020. URL: https://blog.udacity.com/2020/05/pytorch-vs-tensorflow-what-you-need-to-know.htm.

[ZF13] Matthew D. Zeiler und Rob Fergus. „Visualizing and Understanding Convolutional Networks“. In: *CoRR* abs/1311.2901 (12. Nov. 2013). arXiv: 1311.2901 [cs.CV]. URL: http://arxiv.org/abs/1311.2901.

[Zha+20a] Aston Zhang u. a. *Dive into Deep Learning*. https://d2l.ai. 2020. URL: https://d2l.ai.

[Zha+20b] Chaoning Zhang u. a. „ResNet or DenseNet? Introducing Dense Shortcuts to ResNet“. In: (23. Okt. 2020). arXiv: 2010.12496 [cs.CV].

A. Appendix

A.1. Quellen für Trainings-Daten

Psalterien Deutscher Psalter; Basel, Universitätsbibliothek, AN IV 6, geschrieben in Augsburg (?), am 7. September 1485 (Kolophon); digitalisiert als DOI:10.5076/e-codices-ubb-AN-IV-0006. Psalmi, hymni ... cum pulchris picturis; Muenchen, Bayerische Staatsbibliothek, Clm 4301, geschrieben 1495; digitalisiert als urn:nbn:de:bvb:12-bsb00056557-8. Liber psalmorum, canticorum et hymnorum iuxta usum ... monasterii ... Ottoburani; Muenchen, Bayerische Staatsbibliothek, Clm 28999; digitalisiert als http://daten.digitale-sammlungen.de/ db/0005/bsb00056572/images. Claricia Psalter; geschrieben in Augsburg 1178-1250; Walters Art Museum, W-26; digitalisiert als https://www.thedigitalwalters.org/Data/WaltersManuscripts/W26/data/W.26/ Carrow Psalter; geschrieben in England 1250; Walters Art Museum, W.34; digitalisiert als https://thedigitalwalters.org/Data/WaltersManuscripts/html/W34/data. Bobbio Psalter; geschrieben 875-899 Miland, Lombardy; Muenchen, Bayerische Staatsbibliothek; Clm 343 urn:nbn:de:bvb:12-bsb00015213-6. Tegernsee Psalter; geschrieben 1515 Tegernsee; Bayerische Staatsbibliothek Clm 19201; digitalisiert als urn:nbn:de:bvb:12-bsb00056558-4. Tegernsee Psalter; geschrieben 1516-1518 Tegernsee; Bayerische Staatsbibliothek Clm 19203; digitalisiert als urn:nbn:de:bvb:12-bsb00056569-9. Psalter für Kloster Tegernsee; geschrieben 1516-1518 Tegernsee; Bayerische Staatsbibliothek Clm 19202; digitalisiert als urn:nbn:de:bvb:12-bsb00056559-4. Golden Munich Psalter; geschrieben 1200-1225 Oxford, England; Bayerische Staatsbibliothek, Clm 835; digitalisiert als urn:nbn:de:bvb:12-bsb00012920-3. Psalterium Benedictinum congregationis Bursfeldensis; geschrieben 8-29-1459 Hannoversch Muenden, Lower Saxony; Bayerische Staatsbibliothek, 2 L.impr.membr. 2; geschrieben als urn:nbn:de:bvb:12-bsb00036985-2. Psalterium von Queen Isabella; 1300-1400 Nottingham; Bayerische Staatsbibliothek, God.gall.16; geschrieben als urn:nbn:de:bvb:12-bsb00056556-3. Deutscher Psalter, in: Lateinisch-deutsche geistliche Sammelhandschrift; Sarnen, Benediktinerkolleg, Cod. chart. 536, ostschwäbisch geschrieben am 26. März 1480 (Bl. 2v); digitalisiert als DOI:10.5076/e-codices-bks-chart0536. Lateinisch-deutscher Psalter mit Kommentar; Leipzig, Universitätsbibliothek Leipzig, Ms 1502; geschrieben 1471; urn:nbn:de:bsz:15-0012-233230. Lateinischer Psalter mit deutschen Einsprengseln, Heidelberg, Universitätsbibliothek, geschrieben in Rheinland oder Straßburg(?), um 1250(?), Pal. lat. 26, digitalisiert als Lateinischer Psalter mit deutschen Einsprengseln, Heidelberg, Universitätsbibliothek, geschrieben in Frankreich, 13. Jahrhundert 1. Hälfte vor 1235, Pal. lat. 27, digitalisiert als Lateinischer Psalter, Heidelberg, Universitätsbibliothek, geschrieben in Mittelitalien(Toskana), im 14. oder 15. Jahrhundert, Pal. lat 35., digitalisiert als Lateinischer Psalter, Heidelberg, Universitätsbibliothek, geschrieben in Südwestdeutschland bzw. Ostfrankreich(?), im 13. Jahrhundert (Mitte?), Pal. lat. 32, digitalisiert als . Lateinischer Psalter mit deutschen Einsprengseln, Heidelberg, Universitätsbibliothek, geschrieben

in Frankreich(?), im 13. Jahrhundert 2.Hälfte, Pal. lat. 28, digitalisiert als . Lateinischer Psalter, Heidelberg, Universitätsbibliothek, geschrieben in (?), im 15. Jahrhundert verm. 1401, Pal. lat. 34, digitalisiert als . Lateinischer Psalter, Heidelberg, Universitätsbibliothek, geschrieben in Schottland(?), um 1170, Pal. lat. 65, digitalisiert als. Lateinischer Psalter mit deutschen Einsprengseln, Heidelberg, Universitätsbibliothek, geschrieben in Süddeutschland vielleicht Kloster Holzen, Ende 12. Jahrhundert oder Anfang 13. Jahrhundert, Pal. lat. 29, digitalisiert als https://doi.org/10.11588/diglit.12403. Lateinischer Psalter, Heidelberg, Universitätsbibliothek, geschrieben in Norditalien(?), um 800, Pal. lat. 187, digitalisiert als https://. Psalmenkommentare Österreichische Bibelübersetzung: Psalmenkommentar nach Nicolaus von Lyra, Cantica; Heidelberg, Universitätsbibliothek, cpg 32, geschrieben in Augsburg (?), 28. März 1470 (Kolophon); digitalisiert als DOI: https://doi.org/10.11588/diglit.332. Österreichische Bibelübersetzung: Psalmenkommentar nach Nicolaus von Lyra; Karlsruhe, Badische Landesbibliothek, Cod. Donaueschingen 187, datiert auf 1455; digitalisiert als https://nbn-resolving.de/urn:nbn:de:bsz:31-40695. Österreichische Bibelübersetzung: Psalmenkommentar nach Nicolaus von Lyra; Leipzig, Universitätsbibliothek Leipzig, Rep. II 62 (Leihgaber Leipziger Stadtbibliothek); geschrieben in Nordbayern/Mittelbayern im 3. Viertel des 15. Jahrhunderts; digitalisiert als https://nbn-resolving.de/urn:nbn:de:bsz:15-0012-230383. Österreichische Bibelübersetzung: Psalmenkommentar nach Nicolaus von Lyra, Cantica; Solothurn, Zentralbibliothek, Cod. S I 144, Oberrhein/Schweiz um 1460/1470; digitalisiert als DOI:10.5076/e-codices-zbs-SI-0144. Österreichische Bibelübersetzung: Psalmenkommentar nach Nicolaus von Lyra, mit lateinischem und deutschen Text der biblischen Cantica, des Te Deum, des Athanasianum und deutscher Litanei; St. Pölten, Diözesanarchiv, Hs. 30, Niederösterreich (Klosterneuburg ?) 1459-1461; digitalisiert als https://www.dasp.findbuch.net/php/view.php?link=492f30342d3032x34. Vollbibeln Deutsche Bibel, 1. Teil; Basel, Universitätsbibliothek, AN II 36; oberdeutsch, letztes Drittel des 15. Jahrhunderts, digitalisiert als DOI:10.5076/e-codices-ubb-AN-II-0036. Deutsche Bibel, Heidelberg, Universitätsbibliothek, herausgegeben in Nürnberg, 17. Februar 1483, Q 325-8 Folio INC, digitalisiert als .

A.2. Quell-Code

`https://github.com/deleAIst/GSN_20_21_text_image_classification` am 16. Januar 2020.