Inhaltsverzeichnis

Abkürzungverzeichnis

API Application Programming Interface

CNN Convolutional Neural Network

ERP Enterprise-Resource-Planning

FKR Falschklassifikationsrate

GUI Graphical User Interface

KKR Korrektklassifikationsrate

MLP Multilayer Perzeptron

OCR Optical Character Recognition

ReLU Rectified Linear Unit

RVL-CDIP Ryerson Vision Lab Complex Document Information Processing

Abbildungsverzeichnis

Verzeichnis der Listings

1 Künstliche Neuronale Netze

1.1 Künstliche neuronale Netze

Dieser Abschnitt behandelt zunächst die Grundlagen von künstlichen neuronalen Netzen. Da für diese Arbeit ein Convolutional Neural Network (CNN) als Klassifikator verwendet wird, werden diese im Anschluss betrachtet.

Stark vereinfacht bestehen biologische neuronale Netze aus einer Vielzahl hochgradig verknüpfter Neuronen [Kru+15]. Diese akkumulieren empfangene Eingangsreize und geben entsprechende Ausgangsreize weiter. So können Sinneswahrnehmungen ausgewertet und Informationen weitergeleitet werden. Diese Verarbeitung von Reizen läuft parallel im gesamten Nervensystem ab.

Auch ein künstliches neuronales Netz soll aufgrund von Reizen in Form von numerischen Werten eine Ausgabe erzeugen. Wird das Netz, wie in dieser Arbeit, zur Klassifikation von digitalen Bildern verwendet, soll es die zugehörigen Klassenwahrscheinlichkeiten für ein betrachtetes Muster ausgeben.

1.1.1 Künstliche Neuronen

In einem künstlichen neuronalen Netz wird das Konzept der biologischen Neuronen nachempfunden, indem der Ausgabewert in Abhängigkeit der Höhe eines Eingangswertes gesetzt wird. Ein künstliches Neuron ist eine aus n Eingängen $x_1, ...x_n$ und einem Ausgang z bestehende Einheit. Jeder Eingang x_j hat eine bestimmte Gewichtung w_{ij}. Aus allen Eingangswerten und den korrespondierenden Gewichten wird eine Summe gebildet [Nie17].

$$z = \sum_{j=1}^{n} x_j w_{ij} \tag{1.1}$$

Der Ausgangswert, *Aktivitätsniveau* genannt, des Neurons wird gebildet, indem man die Summe der Eingangsgrößen über eine *Aktivierungsfunktion* θ leitet. Zusätzlich kann das Aktivitätsniveau durch ein *Bias*-Neuron beeinflusst werden. Diese spezielle Art von Neuron besitzt keinen Input und hat ein konstantes Aktivitätsniveau gleich Eins. Durch gezieltes Verändern der Verbindungs-Gewichte eines Bias-Neurons können andere Neuronen entweder gehemmt oder verstärkt werden [Nie17].

So ergibt sich zur Berechnung des Aktivitätsniveaus eines künstlichen Neurons folgende Formel (1.2) [GBC16].

$$output = \theta(z) = \theta(\sum_{j=1}^{n} x_j w_{ij} - b) \tag{1.2}$$

1.1.2 Struktur künstlicher neuronaler Netze

Das einfachste künstliche neuronale Netz ist das Perzeptron [Ros58, S. 386–408]. Es besteht in seiner einfachsten Form lediglich aus einem künstlichen Neuron. Da ein solches einschichtiges Perzeptron nur in der Lage ist, linear separable Probleme zu lösen, müssen ein oder mehrere Zwischenschichten von Neuronen hinzugefügt werden, um beliebige Funktionen approximieren zu können. Dadurch entsteht ein verknüpftes Netzwerk aus Neuronen-Schichten, auch Multilayer Perzeptron (MLP) genannt.

Ein solches Netzwerk besteht aus mindestens zwei Schichten, der Eingangsschicht (*Input-Layer*) und der Ausgangsschicht (*Output-Layer*). Dazwischen können beliebig viele Zwischenschichten (*Hidden-Layer*) liegen. Netze mit mehr als einer Zwischenschicht werden als *tiefe künstliche neuronale Netze* bezeichnet, man spricht in diesem Zusammenhang von *Deep Learning* [RM18].

Die Anzahl der Input-Neuronen entspricht der Anzahl der Komponenten des zugeführten Merkmalsvektors. Im Output-Layer entspricht die Anzahl der Neuronen der Anzahl der Klassen. Die Zahl der Neuronen in den Hidden-Layern kann beliebig gewählt werden. Ihre Anzahl gibt die Breite eines Netzes an. Meist wird die Struktur des Netzes in den Hidden-Layern zunächst verbreitert, indem mehr Neuronen als im Input-Layer verwendet werden. Dadurch wird zunächst die Anzahl der Parameter des Netzes erhöht. In späteren Hidden-Layern wird die Anzahl der Neuronen wieder verringert und dadurch auch die Anzahl der trainierbaren Parameter. Das Netz muss somit auf weniger, aber abstraktere Merkmale zurückgreifen, was eine bessere Generalisierung zur Folge hat.

Bei dem in Abbildung 1.1 dargestellten MLP handelt es sich um ein *Feed-Forward-Netzwerk*. Vom Input-Layer aus werden die Eingangsdaten durch das Netz propagiert. In jeder Schicht werden die Aktivitätsniveaus der Neuronen berechnet, welche wiederum Eingangswerte der folgenden Schicht sind [Nie17]. Der Ausgangswert des Output-Layers bildet die Aussage des Netzes zu einem ihm präsentierten Muster.

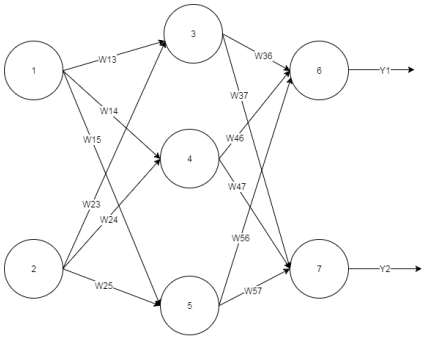

Abbildung 1.1: MLP mit einem Hidden-Layer

Eine alternative Topologie bieten *rekurrente Netze*. Diese Netze können einen gewissen Kontextbezug herstellen, da sie Rückkopplungen in den Verbindungen der Neuronen erlauben [Nie17].

1.1.3 Trainingsphase

Das Lernen in künstlichen neuronalen Netzen findet durch iterative Veränderung der Gewichte und der Bias-Werte statt [Nie17]. Diese werden zu Beginn des Trainings zunächst mit zufälligen, sehr kleinen Werten initialisiert [RM18]. Anschließend präsentiert man dem Netz die Trainingsmuster und ändert die Gewichte sowie die Bias-Werte in Abhängigkeit der Ausgabe. Die Gewichte werden während des Trainings iterativ so oft angepasst bis entweder die Genauigkeit oder der Fehler, den das Netz bei der Vorhersage macht, konvergiert.

Beim *überwachten Lernen* wird eine *Straffunktion* J (engl. *loss*) definiert, welche während des Trainings optimiert wird. Sie ist ein Maß dafür, wie sehr die Ausgabe des Netzes von dem bekannten Sollwert eines Trainingsmusters abweicht [Nie17]. Je besser das neuronale Netz den Sollwert approximiert, umso kleiner ist der Wert der Kosten-Funktion. Ziel ist es deshalb, diesen Fehler durch ein Optimierungsverfahren zu minimieren. Ein solches bietet das *Gradientenabstiegsverfahren*.

Der Gradient der Kosten-Funktion ∇J beschreibt deren Steigungsverhalten für die aktuellen Gewichte und Biaswerte [RM18]. Nun kann die Richtung bestimmt werden, in welche die Gewichtskoeffizienten geändert werden müssen, um den Fehler zu verringern. Die Größe der Änderung wird durch die *Lernrate* η bestimmt. Sie kann während des Trainings variiert werden und bestimmt dadurch die Lerngeschwindigkeit.

$$\mathbf{w} := \mathbf{w} + \Delta\mathbf{w}, \ \textit{wobei} \ \Delta\mathbf{w} = -\eta\nabla J(\mathbf{w}) \tag{1.3}$$

Beim Gradientenabstiegsverfahren werden alle Trainingsmuster durchlaufen und der Fehler summiert. Am Ende einer solchen Epoche werden die Gewichte geändert. Bei großen Datenmengen kann dies sehr ineffizient sein. Deshalb wird zur Optimierung häufig das *stochastische Gradientenabstiegsverfahren* verwendet [RM18]. Hierbei wird der Gradient nicht für alle Muster der Stichprobe auf einmal berechnet. Die Stichprobe wird hier in kleinere Teilmengen (*Batches*) aufgeteilt und der Gradient pro Teilmenge bestimmt. Anstatt die Gewichte nur einmal je Epoche zu verändern, werden sie beim stochastischen Gradientenabstiegsverfahren mehrfach pro Epoche angepasst. Durch die häufigere Aktualisierung konvergiert dieses Verfahren schneller als der normale Gradientenabstieg.

Beim *unüberwachten Lernen* dagegen sind keine Soll-Ausgaben bekannt. Das neuronale Netz verändert die Gewichte hierbei aufgrund von Ähnlichkeiten innerhalb der Trainingsmuster [Kru+15].

Das *bestärkende Lernen* ist ein Lernverfahren, bei dem die Parameter verändert werden, je nachdem, ob ein Verarbeitungsschritt hin zu einem gewünschten Ergebnis oder weg davon geführt hat [Kru+15].

1.1.4 Überanpassung und Unteranpassung

Überanpassung und Unteranpassung resultieren aus mangelnder Generalisierung des Netzes während des Trainings. Als Folge kann ein Netz für neue Muster keine zuverlässigen Aussagen treffen.

Die Überanpassung hat ihre Ursache häufig in zu vielen trainierbaren Parametern eines Modells im Verhältnis zu wenigen Trainingsdaten [RM18]. Überanpassung kann also vermieden werden, indem die Menge an Trainingsmustern vergrößert wird. Alternativ, falls dies nicht oder nur begrenzt möglich ist, kann die Netzkomplexität verringert werden. Eine Methode dazu bietet *Dropout* [Sri+14, S. 1929-1958]. Dabei wird die Anzahl der Parameter eines Modells durch zufälliges Kappen von Verbindungen verringert, wie dies in Abbildung 1.2 dargestellt wird

Von einer Unteranpassung spricht man dann, wenn ein Modell das gegebene Problem aufgrund einer zu geringen Anzahl an Parametern nicht ausreichend generalisieren kann und daher zu bestimmten Ergebnissen tendiert [RM18]. In diesem Fall muss die Komplexität des Modells erhöht werden.

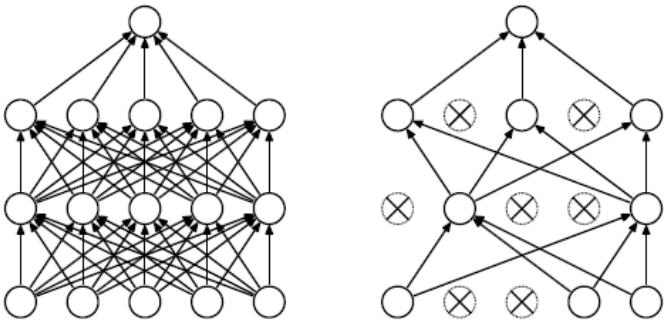

Abbildung 1.2: Links, Modell ohne Dropout. Rechts, mit Dropout [GBC16]

1.2 Convolutional Neural Network

Ein CNN gehört zu der Familie der künstlichen neuronalen Netze und stellt bei der Klassifikation von Bild-Daten den momentan leistungsfähigsten Klassifikator dar. Im Rahmen dieser Arbeit wird ein CNN verwendet, um die Dokumente zu klassifizieren.

1.2.1 Struktur eines Convolutional Neural Network

Die vorderen Schichten eines CNNs dienen zur Extraktion der Merkmale eines Musters. Sie bestehen typischerweise aus einem *Convolutional-Layer*, einer Aktvierungs-Funktion, gefolgt von einem *Pooling-Layer*. Ein Convolutional-Layer besteht in der Regel aus mehreren Filtern. Die letzten Schichten bestehen aus mindestens einem oder mehreren Fully-Connected-Layern, gefolgt von einem Output-Layer, wie in Abschnitt 1.1.2 beschrieben.

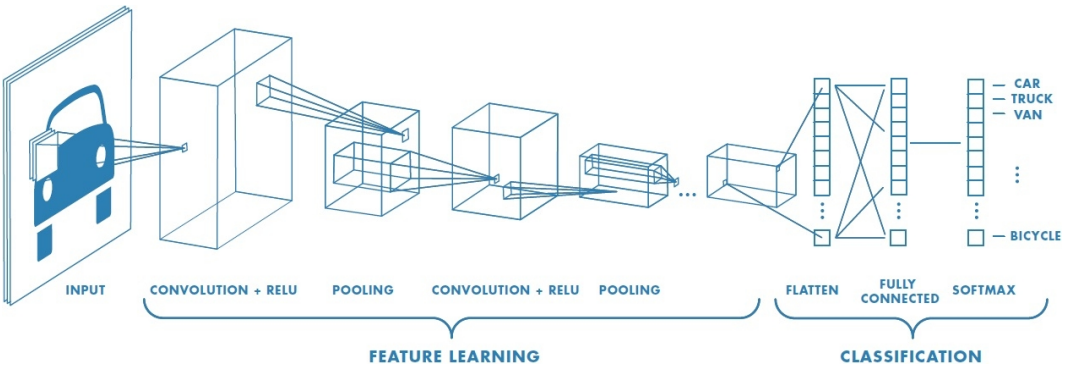

Abbildung 1.3: Topologie eines CNN [Net17]

1.2.2 Faltung

Ein digitales Bild besteht aus einer zweidimensionalen Matrix, deren Elemente je einem Pixel eines Bildes entsprechen und die Farbinformation für jeden Bildpunkt enthalten. Analog zu Techniken der herkömmlichen filterbasierten Bildverarbeitung, werden in einem Convolutional-Layer Filter mit dieser Matrix gefaltet. Dazu wird ein Filter zeilen- und spaltenweise über das Bild verschoben [GBC16]. Bei der Faltung (engl. *convolution*) werden die Koeffizienten eines Filters und die Pixel-Werte eines Bildes paarweise multipliziert und addiert. Daher können die Filter eines Convolutional-Layers als Neuronen betrachtet werden, deren Eingänge die Pixel und deren Gewichte die Koeffizienten der Filtermaske sind. In der Trainings-Phase werden die Koeffizienten der Filter bestimmt.

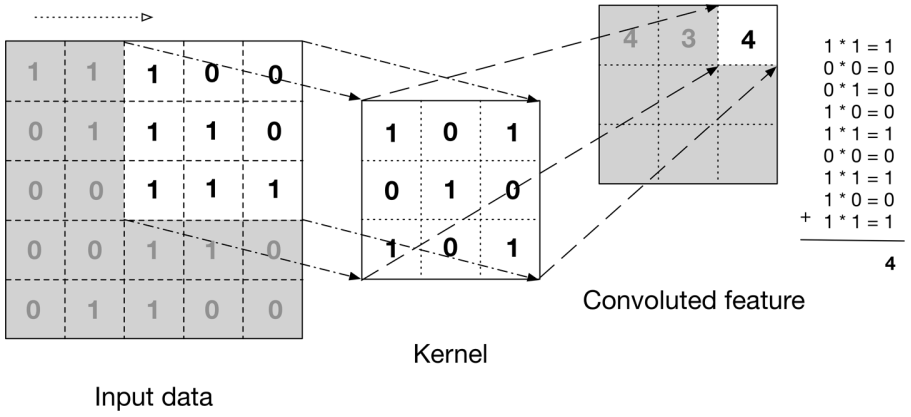

Abbildung 1.4: Prinzip der Faltung einer Matrix mit einem Filterkern [Pat17]

Abbildung 1.4 zeigt das Prinzip der Faltung. Formel 1.4 beschreibt die Faltung einer zweidimensional Matrix I mit einem Filter K der selben Dimension [GBC16].

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{1.4}$$

In einem herkömmlichen neuronalen Netz mit m Neuronen in einer Schicht und n Neuronen in der folgenden Schicht ist zur Berechnung des Output-Werts eine Matrixmultiplikation mit $m \times n$ Parametern notwendig [GBC16]. Jedes Neuron einer Schicht nimmt also Einfluss auf alle folgenden. In einem CNN dagegen ist jedes Neuron der folgenden Schicht nur mit einem Teil der Neuronen der vorangegangen Schicht verknüpft, dem sogenannten *rezeptiven Feld*. Jeder Filter bildet das rezeptive Feld eines Neurons der folgenden Schicht, wie in Abbildung 1.5 dargestellt.

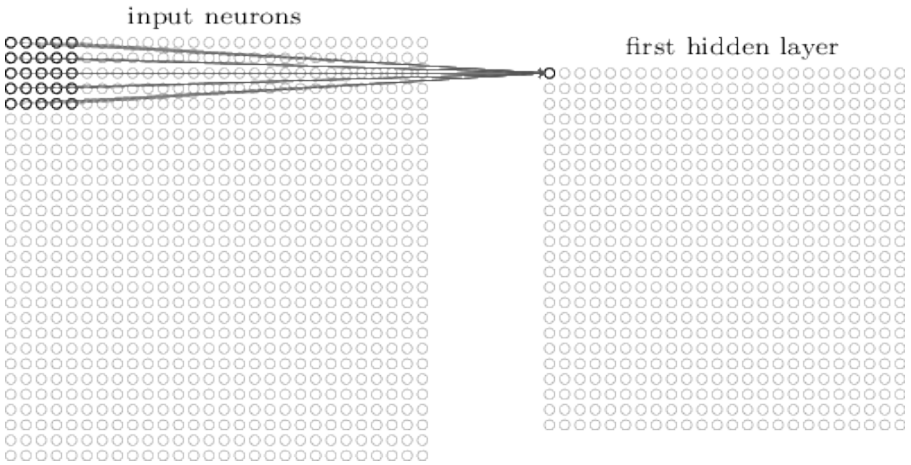

Abbildung 1.5: Rezeptives Feld [Nie17]

Dadurch wird die Anzahl der notwendigen Parameter bei einer Filter-Größe k auf $k \times n$ verringert [GBC16]. Somit muss nicht jeder Eingangswert mit jedem Ausgang verrechnet werden, wodurch sich der Berechnungsaufwand deutlich verringert. Dies ist möglich, da die gesuchten Merkmale in Bezug auf die gesamte Größe eines Bildes vergleichsweise klein sind und deshalb auch die Filter nur wenige Elemente beinhalten müssen. In höheren Schichten vergrößert sich das rezeptive Feld, da Neuronen indirekt durch mehr Neuronen vorhergehender Schichten beeinflusst werden, wie in Abbildung 1.6 veranschaulicht.

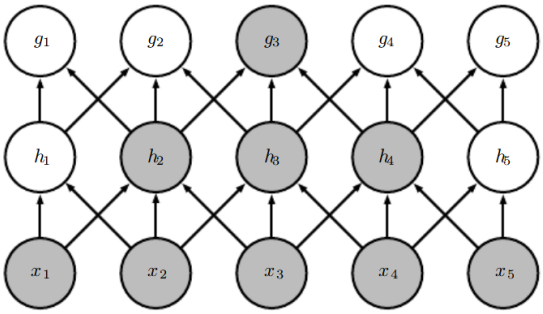

Abbildung 1.6: Vergrößertes rezeptives Feld in tieferen Schichten [GBC16]

Ein weiterer Vorteil, der sich aus der Verwendung von Convolutional-Layern ergibt, ist, dass einmal gelernte Gewichte wiederverwendet werden. Dies ist in dem Begriff *Parameter sharing* oder *Shared Weights* zusammengefasst [GBC16]. Durch das Verschieben des Filters werden die Koeffizienten des Filters mit jedem Bildpunkt verrechnet. Dadurch können Merkmale überall in den Eingangsdaten detektiert werden.

Die Gewichte zum Detektieren eines bestimmten Merkmals müssen daher nur einmal für das gesamte Bild bestimmt werden, während in einem normalen neuronalen Netz für mehrfach auftretende Merkmale die Gewichte jeweils bestimmt werden müssen. Zusätzlich werden CNNs dadurch invariant gegen Translationen, denn Merkmale müssen nicht an bestimmten Positionen auftreten, um detektiert werden zu können.

Als Ergebnis entsteht nach der Faltung eine Matrix, welche als *Feature-Map* bezeichnet wird [GBC16]. Darin ist gespeichert, an welcher Position in den Eingangsdaten ein entsprechender Filter ein Merkmal detektieren konnte. Die Größe der Feature-Map wird durch die Behandlung der Randbereiche eines Bildes (*Padding*) und dem Verschieben des Filters (*Stride*) auf dem Bild beeinflusst.

Üblich ist ein Stride=1, also ein pixelweises Verschieben des Filters. Es können aber auch beliebige andere Werte verwendet werden. Ein größerer Stride führt zu einer stärkeren Verkleinerung der Feature-Map und dadurch zur Verschlechterung der Auflösung zur Detektion eines Merkmals [GBC16].

Bei der Faltung muss der komplette Filter-Kern innerhalb des Bildbereiches liegen. Somit wird bei einer Bildbreite m und einer Filtergröße k die Feature-Map pro Convolutional-Layer um $m - k + 1$ Pixel verringert [GBC16]. Dieser Effekt ist in Abbildung 1.5 zu sehen. Folgen mehrere Convolutional-Layer aufeinander, führt dies zu kleinen und wenig aussagekräftigen Feature-Maps. Man kann diesen Effekt durch Verkleinerung der Filter zwar verlangsamen, allerdings werden die Feature-Maps dennoch bei vielen Schichten deutlich verkleinert. Außerdem werden Merkmale, die direkt in den Randbereichen liegen, nicht berücksichtigt.

Das sogenannte *Zero-Padding* verhindert die Verkleinerung der Feature-Maps. Zudem werden damit auch auf dem Rand liegende Merkmale detektiert [GBC16]. Dabei wird der Rand eines Bildes mit so vielen Nullen (schwarze Punkte in Abbildung 1.7) aufgefüllt, dass bei der Faltung der Mittenwert des Filters auf der jeweils äußersten Pixelreihe des Bildes liegt (Siehe Abbildung 1.7).

Abbildung 1.7: Zero-Padding [GBC16]

Ein Convolutional-Layer besteht üblicherweise aus einer Reihe von Filtern, wobei jeder unterschiedlich aufgebaut ist und somit auch unterschiedliche Merkmale bestimmen kann.

Während in den vorderen Convolutional-Layern grundlegende Merkmale wie Ecken, Kanten, Farb- oder Helligkeitsübergänge detektiert werden, werden diese in tieferen Schichten zu abstrakteren Merkmalen verknüpft.

1.2.3 Aktivierungsfunktion

Ebenso wie in normalen künstlichen neuronalen Netzen wird in einem CNN zur Bestimmung des Ausgangswertes eines Neurons, in diesem Fall eines Filters, eine Aktivierungsfunktion verwendet.

Meist wird eine *logistische Funktion*, wie die *Sigmoid*-Funktion (1.5) oder die *Softmax*-Funktion (1.6) im Output-Layer verwendet [Nie17]. Diese Funktionen sind effizient differenzierbar was die Verwendung des *Backpropagation-Algorithmus* erlaubt.

$$\theta(z) = sigmoid(z) = \frac{1}{1 + e^{-z}} \tag{1.5}$$

$$\theta(z) = softmax(z)_i = \frac{\exp z_i}{\sum_j \exp z_i} \tag{1.6}$$

Bei $z \to \infty$ geht $\theta(z) \to 1$ und bei $z \to -\infty$ gegen $\theta(z) \to 0$ [RM18]. Es können also damit große Werte auf den Bereich von $0 \ldots 1$ abgebildet werden. Die Sigmoid-Funktion wird bei der binären Klassifikation verwendet. Ihre Ausgabe kann als bedingte Wahrscheinlichkeit $\theta(z) = P(y = 1|\mathbf{x}, \mathbf{w})$ interpretiert werden. Für diese gilt, dass bei den beobachteten Merkmalen $\mathbf{x}$ und den Gewichtungen $\mathbf{w}$ das aktuell betrachtete Muster zur Klasse 1 gehört. Die Softmax-Funktion kann bei Klassifikations-Aufgaben mit mehreren Klassen verwendet werden, da sie für jede Klasse eine Wahrscheinlichkeit liefert.

In den anderen Schichten wird die *Rectified Linear Unit (ReLU)* als Aktivierungsfunktion eingesetzt.

$$\theta(z) = max(z, 0) \tag{1.7}$$

Die ReLU-Funktion verhält sich wie eine Schwellwert-Funktion und kann performant berechnet werden. Sie begrenzt das Aktivitätslevel eines Neurons für negative Eingangswerte auf Null und hat im restlichen Verlauf einen konstanten Gradienten. Dies ermöglicht ebenso die Verwendung des Backprobagation-Algorithmus, allerdings mit dem Vorteil, dass die konstante Steigung der Funktion nicht zum *Vanishing-Gradient-Problem* [GBC16] führt. Dieses resultiert aus der Begrenzung des Aktivitätslevels der Sigmoid- und der Softmax-Funktion.

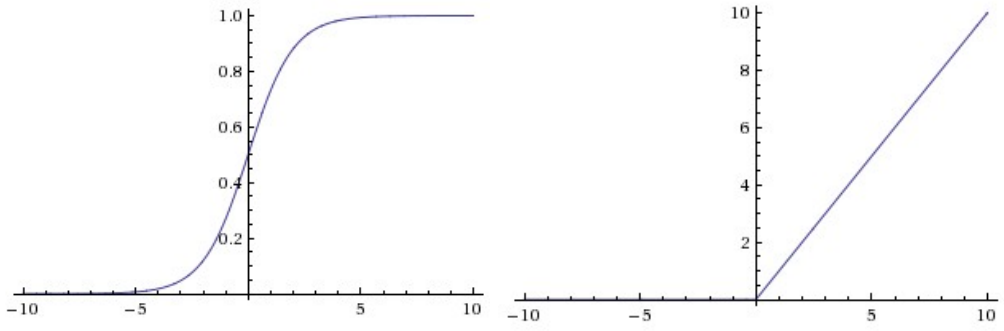

Abbildung 1.8: Aktivierungsfunktion Sigmoid, ReLU [Vis17]

1.2.4 Pooling

Auf die meisten Convolutional-Layer folgt ein *Pooling-Layer*. Ein Pooling-Layer hat eine feste Größe und wird ebenso wie die Filter in den Convolutional-Layer mit einem vorgegebenen Stride über die Eingangsdaten verschoben [GBC16]. Die Pooling-Operation fasst die Eingangswerte ihres rezeptiven Feldes zusammen. Dies kann auf unterschiedliche Weise geschehen und soll hier anhand des häufig verwendeten *Max-Poolings* erläutert werden. Abbildung 1.9 zeigt schematisch das Vorgehen beim Max-Pooling.

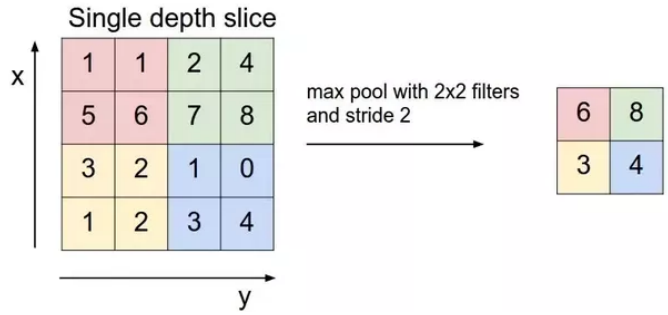

Abbildung 1.9: Beispiel Max-Pooling mit stride=2 [Vis17]

Das Max-Pooling fungiert als Maximalwert-Filter. Aus dem Receptive-Field des Pooling-Layers wird der höchste Wert zur Weiterverarbeitung gespeichert. Die restlichen Werte werden verworfen. In der Feature-Map eines Pooling-Layers werden also nur die stärksten Merkmalswerte notiert, wobei deren exakte vorherige Lokalisierung in den Daten verloren geht.

Die Anwendung von Pooling-Operationen hat mehrere Vorteile. Zunächst reduziert sich die Anzahl der benötigten Parameter deutlich. In der Abbildung 1.9 wird durch Pooling die Anzahl der Parameter um 75% verringert. Dies führt zu einer Steigerung der Verarbeitungsgeschwindigkeit bei gleichzeitig verringertem Speicherbedarf [GBC16].

Weiter wird durch die Pooling-Operationen ein CNN annähernd invariant gegenüber Translationen [GBC16], da für die Klassifikation das Vorhandensein eines Merkmals wichtiger ist als dessen genaue Lokalisierung.

Zudem beugt die Konzentration auf eine geringere Anzahl an Parametern einer Überanpassung vor [GBC16]. Durch die Verringerung der Anzahl der Parameter des Netzes wird die Wahrscheinlichkeit für ein Auswendiglernen des Netzes reduziert.

1.2.5 Batch Normalization

Mit der internen Kovarianzverschiebung (engl. *Internal Covariance Shift*) wird die Veränderung der Parameter einer Schicht in Abhängigkeit der Aktivitätslevel der vorangegangenen Schicht bezeichnet [IS15]. Je größer die Änderung einer Schicht ist, umso stärker ist die Veränderung der folgenden. *Batch-Normalization* ist ein Mittel, den Betrag der internen Kovarianzverschiebung zu begrenzen.

Beim Training eines Netzes werden für jedes Batch der Trainingsdaten $\mathcal{B}$, Durchschnitt μ und Varianz σ für jeden Merkmalswert $x_{i...m}$ berechnet.

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i \tag{1.8}$$

$$\sigma_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \tag{1.9}$$

Jeder Eingangswert wird anschließend mit diesen beiden Werten normalisiert, wie in folgender Formel (1.10) veranschaulicht wird.

$$\hat{x_i} \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \tag{1.10}$$

Batch-Normalization verringert die Veränderung der Parameter einer Schicht, die durch eine Kovarianzverschiebung in früheren Schichten entstehen kann. So wird der überproportionale Einfluss, den sehr große Merkmalswerte auf die Gewichtsanpassung in der Trainingsphase haben, reduziert. Dadurch können höhere Lernraten eingesetzt und das Training beschleunigt werden [Fir17].

Als positiver Nebeneffekt wird die Überanpassung reduziert, da bei der Normalisierung in jeder Schicht ein kleiner Anteil an Rauschen entsteht [Fir17].

2 Vewendete Frameworks

2.1 Die Programmiersprache Python

Python ist eine Multiparadigmensprache, mit welcher Objektorientierung, funktionale sowie aspektorientierte Programmierung umgesetzt werden kann [Tut18]. Da Python Programme interpretiert werden, findet eine dynamische Typisierung zur Laufzeit statt. Die Programme sind durch Blöcke strukturiert, welche mit Einrückungen realisiert werden, anstatt wie bei vielen anderen Sprachen mit Klammerstrukturen. Methoden und Variablen sind in Python selbst Objekte. Dadurch ist es möglich, Funktionen als Parameter zu übergeben oder einer Variablen als Wert eine Funktion zuzuweisen.

Im Umfeld des maschinellen Lernens hat sich Python als eine der meist eingesetzten Programmiersprachen etabliert. Dies liegt zum einen an der einfachen Syntax, die ein schnelles Lernen dieser Sprache ermöglicht, als auch an der Vielzahl von Bibliotheken, die für diese Sprache entwickelt wurden.

NumPy und *SciPy* sind zwei Erweiterungen zur Python Standardbibliothek, die eine effiziente Verwendung von mehrdimensionalen Arrays in Python erlauben[vCV11, S. 22–30]. *Scikit-learn* ist eine mächtige Bibliothek mit Implementierungen verschiedener maschineller Lernalgorithmen [Ped+11, S. 2825–2830]. Die Bibliothek bietet zudem noch Methoden zur Vorbereitung der Trainingsdaten sowie zur Evaluation eines eigenen Modells.

Zur Entwicklung der Graphical User Interface (GUI) wird in dieser Arbeit *PyQt* verwendet. PyQt sind Bindings, die den Einsatz des *Qt*-Frameworks der Firma „The Qt Company“ in Python möglich machen [PyQ18]. Qt selbst ist eine Bibliothek zur plattformübergreifenden Entwicklung von Bedienoberflächen.

2.2 Machine Learning Framework

Für die Implementierung des CNN in Python wird *Tensorflow* als Framework zusammen mit *Keras* verwendet. Beide Werkzeuge werden in diesem Abschnitt vorgestellt.

2.2.1 TensorFlow

TensorFlow ist eine von Google 2015 veröffentlichte Open-Source-Bibliothek für maschinelles Lernen [Zac16]. Die Bibliothek unterstützt sowohl Python als auch C++. Modelle werden in TensorFlow als Graphen repräsentiert. Die einzelnen Operationen des Modells bilden die Knoten des Graphen. Die Kanten des Graphen sind die Eingangs- beziehungsweise die Ausgangsdaten der Knoten. Die Daten, auf welchen die Operationen ausgeführt werden, sind in TensorFlow durch Tensoren repräsentiert. Jeder Knoten erhält als Eingangsdatum einen Tensor und produziert als Ausgangsdatum ebenfalls einen Tensor. Der Rang von Eingangs- und Ausgangstensor kann je nach ausgeführter Operation variieren.

TensorFlow trennt die Definition eines Modells von der Ausführung [Zac16]. Dazu muss, nach der Erstellung eines Graphen, ein Session-Objekt erzeugt werden. Dieses Objekt kapselt die Umgebung, in welcher der Graph ausgeführt wird. Das Session-Objekt stellt sicher, dass zur Ausführung eines Knotens alle benötigten Daten vorliegen. Die Berechnungen einzelner Teilgraphen können dadurch parallel in unterschiedlichen Prozessen auf mehreren Prozessoren ausgeführt werden.

```
1  import tensorflow as tf
2
3  x = tf.constant(3, dtype=tf.int8)
4  y = tf.constant(2, dtype=tf.int8)
5
6  op1 = tf.add(x, y)
7  op2 = tf.mul(x, y)
8  op3 = tf.pow(op2, op1)
9
10 with tf.Session() as sess:
11   op3 = sess.run(op3)
```

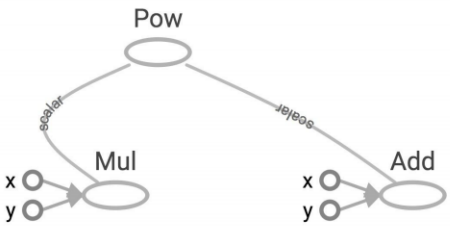

Abbildung 2.1: TensorFlow Programm mit Graph

Abbildung 2.1 zeigt die Definition eines Graphen in TensorFlow. In diesem werden zunächst drei Knoten definiert. Anschließend wird der Graph mit dem Session-Objekt ausgeführt.

Graphen großer Modelle können sehr komplex werden und sind aus dem Programmcode schlecht zu erschließen. Ein Werkzeug zur Visualisierung solcher Graphen bietet TensorFlow mit *TensorBoard*. Dieses Werkzeug bietet zudem auch die Möglichkeit, den Lernfortschritt, den Verlauf der Fehlerrate sowie eigene erstellte Parameter während des Trainings eines Modells darzustellen.

TensorFlow erlaubt die Auslagerung von Berechnungen auf die Prozessoren der Grafikkarte (GPU). Dies ist deutlich effektiver als die Verwendung normaler CPUs und reduziert den zeitlichen Aufwand beim Training eines Modells massiv. Mit der Verwendung einer von Tensorflow unterstützten Grafikkarte wird die Dauer einer Trainings-Epoche des InceptionV3 Modells mit den Ryerson Vision Lab Complex Document Information Processing (RVL-CDIP)-Daten von über 20 Minuten auf sechs Minuten reduziert.

2.2.2 Keras

Zur Erstellung der Modelle in dieser Arbeit wird *Keras* als High-Level-API für TensorFlow verwendet [Cho16]. Keras wurde in Python entwickelt und unterstützt neben Tensorflow noch *Theano* und das von Microsoft entwickelte *Cognitive Toolkit (CNTK)*.

Modelle in Keras werden durch Konkatenieren einzelner Schichten erstellt, wie in 2.1 dargestellt. Jeder Layer, der mit Keras einem Modell hinzugefügt werden kann, nimmt als Eingang einen Tensor entgegen und gibt ebenfalls einen Tensor als Ausgabe weiter. Sowohl die Form des Tensors des Input-Layers als auch die des Output-Layers müssen angegeben werden. Der Input-Tensor entspricht der Größe eines Bildes, in dieser Arbeit $150 \times 150 \times 3$. Die Drei gibt die Anzahl der Farbkanäle eines Bildes an, die 150 stehen für dessen Abmessung in Pixel. Der Output-Tensor entspricht der Anzahl der zu unterscheidenden Klassen. Für die Schichten dazwischen ist dies nicht notwendig, da deren Eingänge auf die vorhergehenden Ausgänge angepasst werden.

```
model = Sequential()
model.add(Conv2D(64, (3,3), activation='relu', input_shape=(150,150,3)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, (3,3), activation='relu'))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='sgd', loss='binary_crossentropy', metrics=['accuracy'])
model.fit(data, labels, epochs=10, batch_size=32)
```

Listing 2.1: Keras Beispiel

Listing 2.1 zeigt das Modell eines CNNs, bestehend aus zwei Convolutional-Layern, einem Max-Pooling-Layer und einem Fully-Connected sowie einem Output-Layer. Die beiden Convolutional-Layer bestehen aus jeweils 64 Filtern mit einer Filterkern-Größe von 3×3 Elementen. Als Aktivierungsfunktionen werden in den vorderen Schichten

ReLU und im Output-Layer eine Sigmoid-Funktion verwendet (siehe Abschnitt 1.2.3). Der Befehl *Flatten()* überführt die Ausgangsdaten des Convolution-Blocks in ein eindimensionales Array (1d-Tensor). Anschließend können diese Werte an die 128 Neuronen des Fully-Connected-Layer übergeben werden.

Nachdem ein Modell in Keras definiert wurde, muss dieses kompiliert werden (siehe Listing 2.1). Dazu muss die *loss*-Funktion sowie ein Optimierungs-Verfahren (*optimizer*) als Parameter vorgegeben werden. In dieser Arbeit wird als Optimierungs-Verfahren der in Abschnitt 1.1.3 beschriebene stochastische Gradientenabstieg verwendet. Anschließend kann das Modell durch den Befehl *model.fit()* trainiert werden.

Die *Datagenerator*-API von Keras liefert eine Schnittstelle, um Bilder einzulesen, diese auf eine gewünschte Auflösung zu skalieren sowie eine Vielzahl von Möglichkeiten, die Bilder zu bearbeiten.

Mit *Callbacks* bietet Keras die Möglichkeit, die Lernhistorie zu speichern, die Lernrate während des Trainings zu variieren oder ein Modell mit Gewichtung zu speichern. Callbacks ermöglichen es auch, Schwellwerte vorzugeben, um das Training vorzeitig zu beenden, falls der Lernfortschritt frühzeitig konvergiert.

Zudem bietet Keras auch eine Schnittstelle an das Scikit-learn-Modul. Das erleichtert die Evaluation von Modellen und erlaubt einen schnellen Vergleich verschiedener Modelle.

Literatur

[Afz+15] M. Z. Afzal u. a. „Deepdocclassifier: Document classification with deep Convolutional Neural Network“. In: *2015 13th International Conference on Document Analysis and Recognition (ICDAR)*. Aug. 2015, S. 1111–1115. DOI: 10.1109/ICDAR.2015.7333933.

[Cho16] François Chollet. *Xception: Deep Learning with Depthwise Separable Convolutions*. 2016. URL: http://arxiv.org/pdf/1610.02357.

[Den+09] Jia Deng u. a. „ImageNet: A large-scale hierarchical image database“. In: *IEEE Conference on Computer Vision and Pattern Recognition, 2009*. Piscataway, NJ: IEEE, 2009, S. 248–255. ISBN: 978-1-4244-3992-8. DOI: 10.1109/CVPR.2009.5206848.

[Fir17] Doukkali Firdaouss. *Batch normalization in Neural Networks*. 2017. URL: https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c (besucht am 13. 02. 2018).

[GBC16] Ian Goodfellow, Yoshua Bengio und Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org.

[Gro95] Grother, Patrick J.National Institute of Standards and Technology. „NIST Handprinted Forms and Characters, NIST Special Database 19“. In: National Institute of Standards and Technology, 1995. DOI: 10.18434/T4H01C.

[HUD15] Adam W. Harley, Alex Ufkes und Konstantinos G. Derpanis. „Evaluation of deep convolutional nets for document image classification and retrieval“. In: *2015 13th International Conference on Document Analysis and Recognition (ICDAR)*. IEEE, 2015, S. 991–995. ISBN: 978-1-4799-1805-8. DOI: 10.1109/ICDAR.2015.7333910.

[IS15] Sergey Ioffe und Christian Szegedy. „Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift“. In: *CoRR* abs/1502.03167 (2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

[Kan+14] Le Kang u. a. „Convolutional Neural Networks for Document Image Classification“. In: *2014 22nd International Conference on Pattern Recognition*. IEEE, 2014, S. 3168–3172. ISBN: 978-1-4799-5209-0. DOI: 10.1109/ICPR.2014.546.

[Kru+15] Rudolf Kruse u. a. *Computational Intelligence*. Wiesbaden: Springer Fachmedien Wiesbaden, 2015. ISBN: 978-3-658-10903-5. DOI: 10.1007/978-3-658-10904-2.

[Lew+06] D. Lewis u. a. „Building a test collection for complex document information processing“. In: *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*. Hrsg. von Efthimis N. Efthimiadis. New York, NY: ACM, 2006, S. 665. ISBN: 1595933697. DOI: 10.1145/1148170.1148307.

[Net17] Convolutional Neural Network. *Convolutional Neural Network*. 2017. URL: https://de.mathworks.com/discovery/convolutional-neural-network.html (besucht am 08. 01. 2018).

[Nie03] Heinrich Niemann. *Klassifikation von Mustern*. Erlangen: Springer, 2003.

[Nie17] Micheal A. Nielsen. *Neural networks and deep learning*. 2017. URL: http://neuralnetworksanddeeplearning.com/chap6.html (besucht am 08. 01. 2018).

[Pat17] Adam Gibson. Josh Patterson. *Deep Learning*. [S.l.]: O'Reilly Media, Inc, 2017. ISBN: 9781491924570.

[Ped+11] F. Pedregosa u. a. „Scikit-learn: Machine Learning in Python“. In: *Journal of Machine Learning Research* 12 (2011), S. 2825–2830.

[PyQ18] PyQT. *What is PyQt?* 2018. URL: https://riverbankcomputing.com/software/pyqt/intro.

[RM18] Sebastian Raschka und Vahid Mirjalili. *Machine Learning mit Python und Scikit-learn und TensorFlow: Das umfassende Praxis-Handbuch für Data Science, Deep Learning und Predictive Analytics*. 2., aktualisierte und erweiterte Auflage. Frechen: mitp, 2018. ISBN: 978-3-95845-733-1.

[Ros58] F. Rosenblatt. „The perceptron: A probabilistic model for information storage and organization in the brain“. In: *Psychological Review* 65.6 (1958), S. 386–408. ISSN: 1939-1471. DOI: 10.1037/h0042519.

[Rus+15] Olga Russakovsky u. a. „ImageNet Large Scale Visual Recognition Challenge“. In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), S. 211–252. DOI: 10.1007/s11263-015-0816-y.

[Sri+14] Nitish Srivastava u. a. „Dropout: A Simple Way to Prevent Neural Networks from Overfitting“. In: *Journal of Machine Learning Research* 15 (2014), S. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[Sze+15] Christian Szegedy u. a. *Rethinking the Inception Architecture for Computer Vision*. 2015. URL: http://arxiv.org/pdf/1512.00567.

[Ten18] Inception in TensorFlow. *Inception in TensorFlow*. 2018. URL: https://github.com/tensorflow/models/tree/master/research/inception.

[TM17] Chris Tensmeyer und Tony Martinez. *Analysis of Convolutional Neural Networks for Document Image Classification*. 2017. URL: http://arxiv.org/pdf/1708.03273v1.

[Tut18] The Python Tutorial. *The Python Tutorial — Python 3.6.4 documentation*. 2018. URL: https://docs.python.org/3/tutorial/index.html.

[vCV11] Stéfan van der Walt, S. Chris Colbert und Gaël Varoquaux. „The NumPy Array: A Structure for Efficient Numerical Computation“. In: *Computing in Science & Engineering* 13.2 (2011), S. 22–30. ISSN: 1521-9615. DOI: 10.1109/MCSE.2011.37.

[Vis17] CS231n Convolutional Neural Networks for Visual Recognition. *CS231n Convolutional Neural Networks for Visual Recognition*. 2017. URL: http://cs231n.github.io/ (besucht am 08. 01. 2018).

[Zac16] Giancarlo Zaccone. *Getting started with TensorFlow: Get up and running with the latest numerical computing library by Google and dive deeper into your data!* Community experience distilled. Birmingham, UK: Packt Publishing Ltd, 2016. ISBN: 978-1-78646-906-9. URL: http://lib.myilibrary.com/detail.asp?ID=944179.