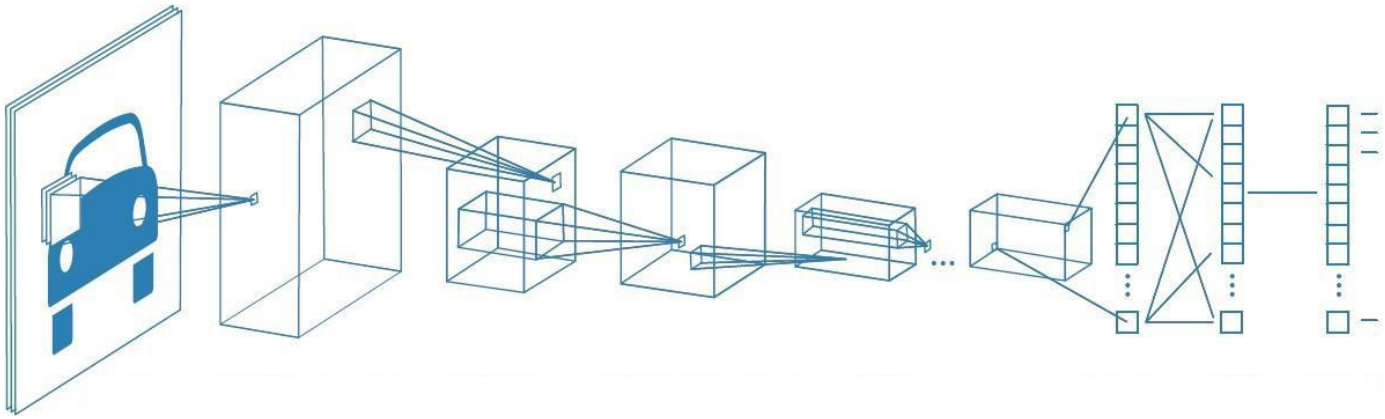
Besondere Lernleistung

Wie funktioniert ein Convolutional Neural Network und wie lässt es sich weiter optimieren?



Eine besondere Lernleistung von Nabil Salama

im Fach Informatik beim Betreuungslehrer Herrn Lange

Inhaltsverzeichnis

1. Einführung

Convolutional Neural Networks (CNN) sind spezielle Neuronale Netzwerke, die auf Bildverarbeitung spezialisiert sind, d.h. sie werden in erster Linie dazu genutzt, den Inhalt eines Bildes festzustellen, sodass jedes Bild mit einer dazugehörigen Klasse assoziiert werden kann, die den Inhalt besagten Bildes beschreibt. Es existieren noch weitere Varianten von CNNs und Einbettungen dieser in weitere Netzwerktypen, die dann z.B. zur Klassifizierung einer Aktivität auf einem Video (Bild 1.1) oder zur Segmentierung und der darauffolgenden Klassifizierung von Inhalten (Bild 1.2) eines Bildes genutzt werden. In der folgenden Arbeit soll sich jedoch ausschließlich auf die Klassifizierung eines Bildes fokussiert werden.

Es wird die Funktionsweise und der Lernprozess des CNN sowohl allgemein als auch mathematisch weiterführend erklärt. Das Verständnis von Letzterem setzt das Verständnis von Ersterem voraus, andersherum gilt dies jedoch nicht. Mathematisch tiefergehende Abschnitte sind als solche gekennzeichnet, im Anhang zu finden und nicht von essentieller Bedeutung für das Gesamtverständnis, es genügt die Idee nachvollziehen zu können. Die mathematischen Formulierungen werden im jeweiligen Abschnitt gegeben und im Anhang hergeleitet. Darüber hinaus wird mein selbst entwickelter Ansatz eines Neuronalen Netzes, die programmiertechnische Umsetzung und das Resultat dessen erläutert.
Der von mir entwickelte Programmcode für ein Convolutional Neural Network befindet sich neben den Bildern, auf die verwiesen werden, im Anhang.

Im CNN wird der Klassifizierungsprozess mithilfe zweier grundlegender Schritte durchgeführt. Die Idee ist als erstes das Bild separierbar zu machen und es danach zu klassifizieren.
Unter Ersterem versteht man die tiefergehende Datenaufbereitung, dessen Zweck es ist, das Klassifizieren einfacher möglich zu machen. Dieser Prozess kann anschaulich dargestellt werden, indem die Datenmenge, in diesem Fall die Menge aller Bilder, in einem kartesischen Koordinatensystem als zahlreiche

Datenpunkte dargestellt werden. Vor dem ersten Schritt sind diese Punkte insofern durcheinander, als dass sie nicht durch eine gerade Linie voneinander getrennt werden können (Bild 2.1). Nach dem Separierungsprozess ist dies jedoch möglich (Bild 2.2). Eben diese Trennlinie zu ziehen geschieht im zweiten Schritt, der Klassifizierung.
Dieses Unterfangen wird mit einem linearen Trennungsalgorithmus durchgeführt. Ein Punkt, der jenseits der Trennlinie liegt, wird der jeweiligen Klasse zugeordnet, die den Inhalt des Bildes beschreibt.

Zur Umsetzung dieser beiden Schritte wird die Bildklassifizierung zunächst als Funktion betrachtet. Diese hat als Input das Bild, als Parameter bestimmte Werte, die darauf abgestimmt sind, den Inhalt eines Bildes zu erkennen und als Output die Wahrscheinlichkeit zu einer Klassenzugehörigkeit, also mit welcher Wahrscheinlichkeit dieses Bild jener Klasse zuzuordnen ist. Im Prinzip gibt es also zwei Fragen, die es zu beantworten gilt: Erstens, wie eine solche Funktion aufgebaut ist und zweitens, was die Parameter dieser Funktion sind. Die Antwort auf ersteres ist die Netzwerkarchitektur, auf letzteres ein Lernalgorithmus zur Bestimmung der Werte.

2. Netzwerkarchitektur

Wie der Name Convolutional Neural Network bereits erahnen lässt, besteht ein CNN zum einen aus „Convolutional Layers“ im Kodierungsblock, in welchem die Separierung stattfindet und zum anderen aus „Neural Layers“ im Prädiktionsblock, welcher der Klassifizierung dient und auch als „Fully Connected“ bezeichnet wird. Zudem gibt es einige weitere Layer, wie etwa „Concatenation Layer“, „Pooling Layer“, „Normalization Layer“, „Dropout“ oder „Softmax Layer“, wobei die ersten drei ausschließlich im Kodierungsblock und der Letztere im Prädiktionsblock zum Einsatz kommt.
Die Layer können hintereinander oder auch parallel angeordnet werden und bestehen aus einer gewissen Anzahl an Neuronen. Dabei hat ein Layer immer als Input einen oder mehrere vorherige Layer und sein Output ist wieder der Input für einen oder mehrere weitere Layer. Das Ganze beginnt mit dem Input Layer,

der das Eingabe-Bild ist und endet mit einem Output Layer, der bei CNNs oftmals ein Softmax Layer ist und die Wahrscheinlichkeiten für eine Klassenzugehörigkeit angibt.

2.1 Layertypen

Die Details zu der folgenden groben Beschreibung der Layer werden später unter dem Punkt Funktionsweise erklärt.

Zunächst zu den Convolutional Layers: Diese sind der wesentliche Bestandteil eines CNNs und ermöglichen es, Konturen im Bild zu erkennen, z.B. Kanten, Linien, Rundungen oder auch komplexere und detailreichere Formen. Es sei vorweggenommen, dass dies mit einer Vielzahl an Filtern, auch Kernel genannt, geschieht, die auf den Input angewendet werden. Für jeden Filter existiert eine sog. „Feature bzw. Activation Map“, ein Convolutional Layer besteht also aus mehreren dieser untergeordneten Layer.

Der nächste Layer ist der Concatenation Layer, welcher mehrere Inputs annimmt und diese zu einem Layer zusammenführt.

Zudem existieren noch die Pooling Layer, die ab und an eingesetzt werden, um den Input auf das Wesentliche zu reduzieren und möglicherweise obsolete Daten herauszunehmen. Dies geschieht in der Regel mithilfe des Max Pooling oder des Average Pooling.

Weiter gibt es die Normalization Layer, bei denen man grundlegend zwischen zwei Techniken unterscheidet: Zum einen der Local Response Normalization (RLN), welche den biologischen Prozess der lateralen Hemmung zwecks Kontrasterhöhung in der Aktivierung der Neuronen simuliert, d.h. dass benachbarte Neuronen gehemmt werden. Dieses erhöhte Gefälle benachbarter Aktivierungswerte von Neuronen ermöglicht es, dass darauffolgende Layer besser eine Erregung an diesen Pixelwerten registrieren können.

Zum anderen existiert die Methode des Batch Normalization (BN), die das beim Lernprozess auftauchende Problem des Internal Covariate Shift löst, dazu später mehr.

Der Dropout Layer ist ebenfalls eine Technik aus dem Gebiet des Lernprozesses, bei der zufällig bestimmte Neuronen während des Trainings ausgelassen werden, um einen gewissen Grad an Unabhängigkeit von den Trainingsdaten zu erreichen.

Der Zweck der Batch Normalization und Dropout Layer wird weiter unten im Abschnitt Lernprozess näher erläutert.
Die Neural Layers, oftmals auch als Dense Layer bezeichnet, bestehen aus spaltenweise angeordneten Neuronen, wobei jedes Neuron mit jedem Neuron des vorherigen und darauffolgenden Layers verbunden ist. In ihrer Gesamtheit stellen die Neural Layers ein Neuronales Netz dar, welches exklusiv dem Input aus genauso vielen Ebenen besteht wie das CNN Dense Layers hat.

2.2 Netzwerkentwicklung

Nun ist noch zu klären wie diese Layer anzuordnen sind, bzw. wie das CNN global betrachtet aufzubauen ist. Im Jahre 1981 erhielten Torsten Wiesel und David Hubel den Nobelpreis für Physiologie und Medizin für ihre Entdeckungen zu der Verarbeitung visueller Informationen im Gehirn.
Sie fanden zum einen heraus, dass örtlich nahe beieinanderliegende Neuronen im visuellen Cortex für gleiche Regionen im Blickfeld zuständig sind und dass sich diese auch überlappen. Das bedeutet für das CNN, dass sowohl die Input Daten eines Bildes als auch die Daten der Convolutional Layer nicht beliebig angeordnet in ein Netzwerk gespeist werden dürfen, indem zum Beispiel sämtliche Zahlenwerte chronologisch aufgezählt und als Input verwendet werden, sondern dass Regionen im Kontext des Bildes zu berücksichtigen sind, indem etwa benachbarte Pixelwerte mit einbezogen werden. Außerdem bedeutet die Überlappung, dass die Neuronen der Convolutional Layer teilweise auf gleiche Neuronen im Input Layer reagieren.
Zum anderen entdeckten T. Wiesel und D. Hubel, dass die Neuronen an eine gewisse Hierarchie gebunden sind. Demzufolge sind die Neuronen der Ganglienzellen im rezeptiven Feld für vergleichsweise einfache Farb- und Lichtanordnungen zuständig, spätere Neuronen reagieren auf genauere Strukturen usw. Insgesamt nimmt also mit zunehmender Neuronentiefe auch die Detailreichhaltigkeit, die von den Neuronen berücksichtigt wird, zu. Je tiefer in dem neuronalen Netz bzw. je höher in der Neuronenhierarchie die Neuronen liegen, desto genauer sind also auch die Bildinformationen verarbeitet. Auf das CNN angewandt resultiert dies in einer Vielzahl an hintereinander geschalteten Convolutional Layers. Beim graphischen Betrachten bestimmter Feature Maps

aus verschiedenen Convolutional Layer eines trainierten CNN lässt sich dieses Verhalten daran beobachten, dass am Anfang des Netzwerks liegende Activation Maps etwa Kanten und Linien erkennen, etwas tiefer liegende bereits Ecken oder ähnliches zur Kenntnis nehmen und zum Ende hin liegende Bereiche bereits Muster und Texturen berücksichtigen (Bild 3.1).
Eine daraus folgende mögliche Netzwerkarchitektur wäre die des Neocognitron CNN, das bereits 1980 entwickelt wurde. Aufgrund der damaligen Rechenleistung wurden die Netze aber erst etwa 30 Jahre später brauchbar.

2.3 Mögliche Architekturen

Ein einfaches Beispiel für ein CNN wäre das der Oxford Vision Geometry Group, auch VGG genannt, welches im Jahre 2014 entwickelt wurde und aus fünf Blöcken im Kodierungsblock besteht, die je nach Konfiguration aus eins bis vier Convolutional Layers und einem Pooling Layer bestehen. Der Prädiktionsblock besteht dann aus drei Fully Connected und einem Softmax Layer (Bild 3.2). Mit etwa 138 Millionen zu erlernenden Parametern ist dieses Netzwerk nicht gerade effizient, dafür jedoch sowohl einfach als auch effektiv.
Es besteht auch die Möglichkeit deutlich komplexere und entsprechend akkuratere Strukturen mit den Layern aufzubauen, die gleichzeitig auch effizienter sind. Als Beispiel dafür seit etwa das GoogLeNet Inception-v3 CNN (Bild 3.3) angeführt. Bei diesem Netzwerk wird zunächst der Input durch Normalization, Convolutional und Pooling Layer aufbereitet. Daran angehängt wird dann der wesentliche Teil, eine Anreihung von sog. „Inception-Modulen“. Diese Blöcke haben stets einen Input, der durch einen Pooling Layer dargestellt wird und einen Output, der ein Concatenation Layer ist.
Innerhalb der Module werden einige mögliche Konstellationen von Convolutional Layers parallel zueinander ausgeführt und durch den Lernprozess wird die beste Kombination aus Anordnungen der Convolutional Layer erlernt. Das Besondere an dieser Netzwerkarchitektur ist, dass sie sich so verstehen lässt, dass die optimale Anordnungen der Layer nicht von den Entwicklern festgelegt und beschränkt wird, sondern dass sie erlernt werden kann. Jedoch ist diese Methode äußerst rechenintensiv. Dieses Problem wird dadurch gelöst, dass vor den eigentlichen Convolutional Layers sog. „Bottleneck Layer“ gestellt werden, die

die Auflösung in z-Richtung, also die Anzahl der Feature Maps reduziert, ohne dabei wesentliche Daten zu verlieren. Dieser Layer ist ein praktisch trivialer Convolutional Layer, der aber einen anderen Zweck erfüllt.
Das Ende besteht aus einem Pooling und einem Fully Connected Layer, gefolgt von einem Softmax Layer. Zur Optimierung und Analyse des Lernprozesses werden an zwei Stellen in der Mitte des CNNs bereits Endblöcke eingefügt, die einerseits eine Art von Zwischenergebnissen liefern und andererseits das Lernen deutlich vereinfachen. Zum Vergleich hat dieses Netzwerk lediglich 5 Millionen Parameter, also knapp 30-mal weniger als das VGG CNN, erzielt aber eine Fehlerrate von 6,7% gegenüber dem VGG Netzwerk mit 7,3%.
Die beiden zuvor erwähnten Netzwerke bestehen aus etwa 20 hintereinander geschalteten Convolutional Layers. Man hat versucht, die Netzwerke zu verbessern, indem man mehr dieser Layer verwendet. Dies führt jedoch dazu, dass wider Erwarten die Fehlerrate steigt. Es wurde im Zuge dessen die Hypothese aufgestellt, die zwar bis heute nicht bewiesen ist, aber einen praktischen Nutzen hat, wenn man ihr nachgeht, dass diese Abnahme der Trefferrate auf einen erschwerten Lernprozess zurückzuführen ist, der durch die Vielzahl an Layern entsteht. Dies bedeutet, dass sich die beim Lernen zu bestimmenden Parameter des CNN schlechter ermitteln lassen. Der Ansatz zum Lösen des Problems ist, dass zum Beispiel ein mehr als 20 Layer tiefes Neuronales Netz mindestens genauso gute oder bessere Resultate liefern muss, als ein 20 Layer großes Netz. Im Zuge dessen wurde das ResNet CNN entwickelt, welches aus vielen der sog. „Residuenblöcken“ bestehen, die durch einen Bottleneck Layer, einen Convolutional Layer und einen umgekehrten Bottleneck Layer dargestellt werden. Letzterer Layer erfüllt die Funktion, den durch den ersten Layer herunterskalierten Input wieder hochzuskalieren. Angenommen das Netzwerk hat dann nach 20 Layern ein angemessenes Resultat erzielt, dann wird dieses Resultat, welches etwa den Input für den Residuenblock am 21. Layer darstellt, dem Output diesen Residuenblocks hinzuaddiert. Dies wird Residuenverbindung oder Residuum genannt. Ein solcher Block ist in Bild 3.4 abgebildet.
Das Residuum führt dazu, dass falls sich das Netzwerk jenseits der 20 Layer verschlechtern sollte, der noch bessere Status des CNNs vor dem jeweiligen Residuenblock berücksichtigt wird und falls notwendig das Neuronale Netz

erlernen kann, die Parameter im Convolutional Layer des Blockes so zu setzen, dass der Output null ist. Daher ist bei einer Addition mit dem besseren Input der gesamte Output des Blocks mindestens genauso performant wie sein Input ist. Mit dieser Erkenntnis müsste sich feststellen lassen, dass ein mehr als 20 Layer tiefes CNN mindestens genauso gut, aber nicht zwangsläufig besser ist. Letzteres kann man erst unter der Berücksichtigung von Folgendem Schlussfolgern: Angenommen der gewünschte Output ist gegeben durch eine Funktion $H(x)$, dann müssten die Convolutional Layer diese Funktion vollständig erlernen. Im Residuenblock muss ein Convolutional Layer hingegen lediglich die Änderung $F(x)$ vom Input berechnen, weil der gesamte Output des Residuenblocks gegeben ist durch $H(x) = F(x) + x$. Dies erleichtert den Lernprozess stark und lässt das Netzwerk eine Fehlerrate von lediglich 3,57% erzielen. Der Aufbau des gesamten ResNet ist in Bild 3.5 abgebildet.

Eine Kombination des GoogleNet und ResNet CNN ist das Google Res Net, bei dessen Architektur ein Inception Modul und ein Residuenblock zusammengeführt wurden (Bild 3.6). Dies ist das heute Leistungsfähigste Convolutional Neural Network, dass mit einer Fehlerrate von 3,08% die des Menschen mit etwa 5% übertrifft.

Einen Überblick über die Performance verschiedener CNNs in Abhängigkeit der benötigten Rechenoperationen ist in Bild 3.7 zu betrachten.

3. Funktionsweise

Wenn man erneut ein CNN als eine Funktion betrachtet, dann besitzt diese gewisse Parameter, die Biases und die Gewichte, die im Lernprozess erlernt werden müssen. Dazu müssen jedoch zunächst Anfangswerte vorliegen. Diese werden bei der Initialisierung des CNN zufällig gewählt. Zwar handelt es sich dabei nicht um einen Lernprozess, doch um nachvollziehen zu können, weshalb gewisse Vorgaben beim Bestimmen der Zufallswerte notwendig sind, sei dies später unter dem Abschnitt Lernprozess erläutert.

Zunächst liegt ein $m_1^l \; x \; m_2^l \; x \; m_3^l$ (der „Exponent“ gibt den Index l des Layers an) Pixel großes Bild mit drei Farbkanälen, Rot, Grün und Blau (RGB), vor, welches zu klassifizieren ist. Jeder Farbkanal enthält den Rot-, Grün- bzw. Blauanteil. Die Tiefe ist daher m_3^l. Das Bild stellt den Input Layer in Form von drei $m_1^l \; x \; m_2^l$ großen Feature Maps (für jeden Farbkanal eine Map) dar und der Layer hat somit die Dimensionen $m_1^l \; x \; m_2^l \; x \; m_3^l$.

3.1 Convolutional Layer

Auf den Input Layer folgt dann z.B. ein Convolutional Layer mit den Dimensionen $m_1^l \; x \; m_2^l \; x \; m_3^l$ (Da im Convolutional Layer l einen anderen Wert hat als im Input Layer, sind die Dimensionen abweichend). Im Gegensatz zum Input-Layer gibt nun m_3^l die Anzahl der Feature Maps an, die von den Filtern erzeugt werden. Es sei noch einmal erwähnt, dass jeder Filter genau eine Activation Map erzeugt. Diese in Breite und Höhe relativ kleinen Filter werden zeilenweise über den Input-Layer geschoben. Der Ort im vorherigen Layer, an dem sich gerade der Filter befindet wird auch rezeptives Feld genannt. Für jede Verschiebung des Kernels, also für jeden Schritt wird dann ein Wert in das Neuron der Activation Map des aktuellen Layers geschrieben, der sich aus den Gewichten des Kernels und den Aktivierungswerten der betroffenen Neuronen des Input Layers ergibt. Dieser Wert wird dann mit dem Bias des Filters verrechnet und in eine Aktivierungsfunktion gegeben, dessen Rückgabewert die Aktivierung des Neurons darstellt. Dieser Vorgang ist in Bild 4.1 und Bild 4.2 vereinfacht mit nur einer Feature Map als Input dargestellt. Die Werte des Kernels werden auch Gewichte genannt, da sie die Gewichtung der Neuronenwerte angeben, mit denen sie Verrechnet werden.

Nun zur mathematisch korrekten Beschreibung dieses Vorgangs:
Der vom rezeptiven Feld durch den Filter und Input Layer erzeugte Wert $z_{i,j}^l$ ist die Summe über die Elemente der dreidimensionalen Matrix M, die aus der Elementweisen Multiplikation (Hadamard-Produkt) $M = K \odot A$ der Kernel-Werte mit den Aktivierungswerten der jeweiligen Neuronen entstehen:

$$z_{p,q}^{l} = b^{l,f} + \sum_{i=1}^{k}\sum_{j=1}^{k}\sum_{o=1}^{m_3^{l-1}} k_{i,j,o}^{l,f} \cdot a_{I,J,o}^{l-1} \qquad I = i + n_1 s, \qquad J = j + n_2 s, \qquad n_{1,2} \in \mathbb{N}_0$$

Index f der Feature Map, Stride s,

$n_{1,2}$-te Verschiebung des Filters in horizontaler bzw. vertikaler Richtung

Um die Elemente $k_{i,j,o}^{l,f}$ der Matrix $K^{l,f}$ zu erhalten, wird über die k Zeilen und k Spalten des Kernels mit der Tiefe, die der Anzahl der Feature Maps im vorherigen Layer entspricht, iteriert. Jedes Element wird mit dem Aktivierungswert $a_{I,J,o}^{l-1}$ des jeweiligen Neurons im vorherigen Layer multipliziert.
Ein Kernel ist frontal betrachtet immer quadratisch, also genauso breit wie hoch und kann beim Verschieben über den Input Layer verschiedene Schrittweiten, welche durch den sog. „Stride“ definiert sind, annehmen.
Für die Breite bzw. Höhe k des Filters gilt meistens: $\{k \in \mathbb{N}_0 \mid 3 \leq k \leq 7\}$
Schlussendlich wird zu der errechneten Summe noch ein Kernel-spezifischer Bias $b^{l,f}$ addiert. Dieser Vorgang ist in Bild 4.3 und Bild 4.4 veranschaulicht animiert.

Es folgt eine fachlich weiterführende Beschreibung dieses Vorgangs:
Um die Dimensionen der Output Feature Map zu berechnen lässt sich folgende Formel heranziehen:

$$x_o = \frac{x_i + 2p - k}{s} + 1$$

Breite oder Höhe der Output bzw. Input Feature Map x_o bzw. x_i, Padding p

Um zu überprüfen, ob sich ein Stride auf einen Input anwenden lässt, muss folgende Bedingung gelten: $x_o \in \mathbb{N}$, denn eine Dimension hat immer ganzzahlig zu sein. Falls sich ein Stride, nicht auf einen Input anwenden lässt, kann zu einem Padding gegriffen werden, welcher einen Rand zu den Input Feature Maps hinzufügt.
Man unterscheidet zwischen Constant/Zero Padding, Symmetric Padding und Reflect Padding. Beim Zero Padding werden die Werte des hinzugefügten Randes mit Nullen besetzt, bei den anderen beiden Methoden werden die Werte im Falle von Symmetric Padding an der äußersten, im Falle von Reflect Padding

an der zweitäußersten Zeile bzw. Spalte der eigentlichen Feature Map gespiegelt. Mögliche Padding-Einstellungen sind unter Bild 4.5 zu betrachten.
Paddings werden jedoch in erster Linie angewendet, um die Dimensionen der Convolutional Layer mit zunehmender Tiefe im Netzwerk nicht zu schnell schrumpfen zu lassen, denn durch zu kleine Layer können notwendige Informationen verloren gehen.
Die zuvor erwähnten trivialen Convolutional Layer, die Bottleneck Layer, haben eine Filtergröße und einen Stride von 1, sodass die Zahl der Feature Maps eines Inputs auf die Anzahl der genutzten Filter reduziert werden kann, ohne dabei die Dimensionen in x- und y- Richtung zu ändern.
Wie bereits angesprochen ist die Aktivierung eines Neurons nicht direkt der Wert $z_{i,j}^{l}$, der sich aus Kernel und vorherigem Layer ergibt, sondern der Funktionswert einer Aktivierungsfunktion f_{akt}, deren Input $z_{i,j}^{l}$ ist:

$$a_{i,j}^{l} = f_{akt}\left(z_{i,j}^{l}\right)$$

Diese Funktion darf nicht linear sein, weil sie in jedem Neuron jeden Layers angewendet wird und eine Kombination linearer Funktionen stets wieder eine lineare Funktion ergibt:

$$f_1\big(f_2(x)\big) = a_1 f(x) + b_1 = a_1(a_2 x + b_2) + b_1 = a_1 a_2 x + a_1 b_2 + b_1 = a_3 x + b_3$$

Wäre die Aktivierungsfunktion linear würde dies bedeuten, dass das gesamte Neuronale Netz eine lineare Funktion ist und der zu erlernende Sachverhalt eine lineare Abhängigkeit aufweisen müsste, was bei Bildern und auch anderen Anwendungsgebieten sicher nicht der Fall ist.
In der Biologie feuert ein Neuron oder es feuert nicht, d.h. es gibt entweder das Eingangssignal weiter oder nicht. Genau dieses Verhalten simuliert eine Aktivierungsfunktion.
Das Ergebnis dieser ersten Intuition wäre die sog. „Step Function“, die je nach Input entweder 0 oder 1 ausgibt:

$$f_{akt}(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases}$$

Nun ist diese Funktion jedoch lediglich dazu in der Lage, einen binären Output zu liefern, was in Anbetracht unseres Netzwerkes das erforderliche Komplexitätsniveau bei weitem nicht erreicht. Außerdem ist für den Lernprozess

die erste Ableitung der Funktion relevant und die Änderungsrate der Step Function ist zu jeder Zeit Null, wodurch das Lernen verhindert werden würde.
Daher wurde zu der logistischen sog. „Sigmoidfunktion“ (auch als „Schwanenhalsfunktion“ bekannt) gegriffen, welche vom Prinzip her ähnlich der Step Function ist, aber einen kontinuierlichen Übergang von inaktiv zu aktiv aufweist, das heißt der Übergang von nicht feuerndem zu feuerndem Neuron ist fließend. Auch die Sigmoidfunktion kann sämtliche Werte auf dem Intervall $]0;1[$ annehmen:

$$f_{akt}(x) = \sigma(x) = \frac{1}{1+e^{-x}}$$

Dabei konvergiert die Sigmoidfunktion für $\lim_{z\to\infty}\big(\sigma(x)\big)$ gegen 1^- und für $\lim_{z\to-\infty}\big(\sigma(x)\big)$ gegen 0^+, sodass mit zunehmender Anregung des Neurons auch die Aktivierung zunimmt, wodurch das ganze realitätsnäher wird. Allerdings ist diese Funktion bei sehr häufigem Ausführen zu rechenintensiv und für relativ große und kleine Inputs konvergiert auch der Wert der Ableitung gegen Null, was den Lernprozess erschwert. Eine weitere Problematik ist, dass die Funktion nicht zum Ursprung zentriert ist, d.h. dass ein Input kleiner Null in der einen Positiven Wert ergibt, der dazu führt, dass in der darauffolgenden Aktivierungsfunktion der Output potentiell größer wird, was langfristig zu großen Inputs führt. Dieses Problem wird durch eine andere Funktion gelöst, der Tangens hyperbolicus Funktion, die durch den Ursprung geht:

$$f_{akt}(x) = tanh(x) = 1 - \frac{2}{e^{2x}+1}$$

Jedoch löst diese Funktion nicht die anderen Probleme. Daher wird heutzutage üblicherweise zu der ReLu Funktion (Rectified Linear Unit) gegriffen. Diese ist gegeben durch folgende Gleichung:

$$f_{akt}(x) = max(0,x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

Diese Funktion ist kaum rechenintensiv und nicht linear. Allerdings besteht eine nicht allzu gravierend Problematik darin, dass das sog. „Dying ReLu Problem“ eintritt. Dies bedeutet, dass falls der Input kleiner Null ist und somit auch die Ableitung an dieser Stelle, dass das Neuron nicht lernt und somit „stirbt“. Deshalb wird teilweise auch die Leaky ReLu Aktivierungsfunktion verwendet, dessen Ableitung für $x < 0$ ungleich Null ist:

$$f_{akt}(x) = max(0{,}1x\ , x) = \begin{cases} x, & x \geq 0 \\ 0{,}1x, & x < 0 \end{cases}$$

Es existieren noch weitere Abwandlungen der ReLu Funktion, aber diese bringen keine signifikant performanteren Ergebnisse, bis auf die Swish Aktivierungsfunktion, die tatsächlich die Fehlerrate um bis zu knapp einen Prozent verringert. Die Funktion ist zwar sehr ähnlich zu ReLu, aber dennoch Rechenintensiv:

$$f_{akt}(x) = x \cdot \sigma(x) = \frac{x}{1 + e^{-x}}$$

Eine Übersicht über die Aktivierungsfunktionen mit den zugehörigen Graphen ist in Bild 4.6.

3.2 Concatenation Layer

Es existieren zwei Varianten von Concatenation Layern. Entweder werden die Aktivierungen mehrerer Inputs gleicher Dimension elementweise zusammenaddiert, sodass als Output ein Layer gleicher Dimension wie die Inputs herauskommt oder es werden mehrere Inputs, die in zwei der drei Dimensionen übereinstimmen entlang der dritten Dimension aneinandergefügt (Bild 4.7). Die erstere Variante kommt bei Residuen-Blocks zum Einsatz, wohingegen letztere dazu verwendet wird, eher voneinander abweichende Inputs zusammen zu führen, wie es am Ende eines Inception-Moduls des GoogLeNet der Fall ist.

3.3 Pooling Layer

Der Pooling-Layer verringert die Dimension eines Layers in x- und y-Richtung, wobei hierfür keine Gewichte oder Biases verwendet werden, es handelt sich daher um einen nicht lernbaren Prozess. Dennoch ist das Vorgehen ähnlich wie bei einem Convolutional Layer, nur das außerdem erstens Kernel-Größe dem Stride entspricht: $k = s$, sodass keine Überlappungen der Filterbereiche zustande kommen und zweitens das Vorgehen je Feature-Map ausgeführt wird, sodass man sich lediglich im zweidimensionalen Raum befindet und der Output in der Länge der dritten Dimension dem Input entspricht. Auch für diese Pooling-Layer existieren zwei Varianten. Beim Max-Pooling wird der höchste Wert des vom Filter bedeckten Input-Neuron übernommen, wohingegen beim Max-Pooling

der Output-Wert das arithmetische Mittel der Aktivierungswerte der vom Filter abgedeckten Neuronen der Activation-Map ist (Bild 4.8).

3.4 Normalization Layer

Als nächstes werden die Normalization Layer thematisiert, bei denen zwischen dem Local Response Normalization (LRN) und dem Batch Normalization (BN) zu unterscheiden ist.

Der LRN Layer passt die Werte der in der Umgebung liegenden Aktivierungswerte, die bis zu n Neuronen entfernt liegen, an. Es werden dabei keine Gewichte oder Biases benötigt, was bedeutet, dass dieser Layer nicht lernbar ist, also nicht mit in den Lernprozess involviert ist. Man unterscheidet dabei ebenfalls zwischen den Methoden der Inter-Channel LRN und der Intra-Channel LRN.

Im Inter-Channel LRN bezieht sich die Umgebung, in der angepasst wird ausschließlich auf die dritte Dimension eines Layers, also die Achse der Feature Maps. Entsprechend ist dieses Verfahren eindimensional.

Anders sieht das bei dem Intra-Channel LRN aus, bei dem die Umgebung nicht die Activation Maps sind, sondern ausschließlich die Neuronen innerhalb einer Map. Da diese sowohl in x- als auch y-Richtung liegen, ist diese Methode zweidimensional.

Dieses beiden Verfahren sind in Bild 4.9 abgebildet.

Mathematisch lässt sich das Verfahren des Inter-Channel LRN wie folgt beschrieben:

$$b_{x,y}^{f} = a_{x,y}^{f}\left(\delta + \alpha \sum_{i=f_{min}(1,f-n)}^{f_{max}(m_3,f+n)} \left(a_{x,y}^{i}\right)^2\right)^{-\beta}$$

$$f_{min}(x,y) = min(x,y) = \begin{cases} x, & x \leq y \\ y, & x > y \end{cases} \quad f_{max}(x,y) = max(x,y) = \begin{cases} x, & x \geq y \\ y, & x < y \end{cases}$$

Index d. Feature Map f, Position des Neurons $x, y \in \mathbb{N}_0$ und $x \in [0, m_1], y \in [0, m_2]$

Dabei steht $b_{x,y}^{f}$ für den Neuronenwert nach dem LRN und $a_{x,y}^{f}$ für den Aktivierungswert vor dem LRN. α und β sind Hyperparameter, die stets

konstanten sind, welche die Einstellungen eines Neuronalen Netzwerkes darstellen. Hierbei wird mit α kontrolliert, wie sehr die Werte abgeändert werden sollen und mit β wird eingestellt, wie hoch der Kontrast zwischen den Werten ausfallen soll. Die Konstante δ hat einen sehr niedrigen Wert, üblicherweise um 10^{-15}, welche dazu dient eine Division mit Null zu vermeiden, falls die umliegenden Aktivierungswerte Null sein sollten. Eine übliche Wahl der Parameter ist $\alpha = 1$, $\beta = 1$ und $n = m_3$.
Da es in der ersten Feature Map keine Werte für Aktivierungen von Neuronen vorangehender Activation Maps und in der letzten Feature Map keine Aktivierungen für darauffolgende Maps gibt, wird nicht über das Intervall $[f - n; f + n]$ iteriert, sondern es wird mithilfe von $min(x, y)$ und $max(x, y)$ als Untergrenze 1, also die erste Feature Map und als Obergrenze m_3, also die letzte Feature Map festgelegt, sodass $a^i_{x,y}$ stets definiert ist und $[f - n; f + n] \in [1; m_3]$ gilt. Ein mögliches Ergebnis des Inter-Channel LRN ist in Bild 4.10 abgebildet.

Da sich das Intra-Channel vom Inter-Channel Verfahren bezüglich der Dimensionen unterscheidet, wird hierbei sowohl in x- als auch y-Richtung iteriert, nicht aber entlang der z-Dimension. Ansonsten unterscheidet es sich in der Berechnung nicht vom Inter-Channel:

$$b^f_{x,y} = a^f_{x,y}\left(\delta + \alpha \sum_{i=f_{min}(1,x-n)}^{f_{max}(m_1,x+n)} \sum_{j=f_{min}(1,y-n)}^{f_{max}(m_2,y+n)} \left(a^f_{i,j}\right)^2\right)^{-\beta}$$

Ein CNN wird mit Beispielbildern trainiert. Aus Effizienzgründen werden diese in der Regel zu einer Sequenz von Bildern der Länge m zusammengefasst, sodass ein Trainingsdurchlauf mehrere Bilder berücksichtigt. Diese Sequenz wird auch Batch genannt. Der Batch Normalization Layer standardisiert also die Bilder einer Sequenz. Dies ist im Prinzip dasselbe wie das Transformieren einer Normalverteilung auf die Standardnormalverteilung und anschließendes Rücktransformieren in eine für jeden Batch und jedes Neuron einheitliche Verteilung. Näheres dazu wird im Abschnitt Lernprozess erläutert. Im ersten Schritt wird der individuelle Erwartungswert und die spezifische

Standardabweichung einer Sequenz berechnet. Danach wird im zweiten Schritt der Batch standardisiert bzw. normalisiert, also praktisch auf die Standardnormalverteilung transformiert. Als letztes wird dann die Rücktransformation in die nicht Batch-Spezifischen Verteilung vorgenommen. Wie diese Verteilung auszusehen hat, muss jedoch erlernt werden. Falls eine Transformierung jedoch nicht vorteilhaft sein sollte, so kann das Netzwerk die Rücktransformierung so erlernen, dass auf die ursprüngliche Verteilung transformiert wird, so als wäre der Normalization Layer ausgelassen worden.
Mathematisch können diese Schritte diesen Layers wie folgt beschrieben werden:

$$\mu = \sum_{i=1}^{m} x_i \cdot p_i = \sum_{i=1}^{m} x_i \cdot \frac{1}{m} = \frac{1}{m} \sum_{i=1}^{m} x_i$$

$$\sigma^2 = \sum_{i=1}^{m} (x_i - \mu)^2 \cdot p_i = \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu)^2$$

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \delta}}$$

$$y_i = BN_{\gamma,\beta}(x_i) \equiv \gamma \hat{x}_i + \beta$$

Input x_i, Output y_i, Erwartungswert μ, Standardabweichung σ

Auch hier wird wieder um eine Division mit Null zu vermeiden ein kleiner Wert δ im Nenner hinzuaddiert. Ob und wie die Wurzel im Nenner zu setzen ist, ist daher eigentlich irrelevant. Im Idealfall ist δ an dieser Stelle eigentlich nicht notwendig, da die Standardabweichung bei normativem Gebrauch des Batch Normalization Layers nie Null ist und ein Batch immer aus unterschiedlichen Bildern besteht. Ein Batch mit mehreren gleichen Bildern hätte beim Training den Effekt eines Batchs mit einem Bild beziehungsweise den Effekt des Verzichtens auf die Nutzung der Sequenz-Technik. Jedoch wird diese Transformation Pixel- und Farbkanalweise durchgeführt und sollte der äußerst unwahrscheinliche Fall eintreten, dass diese Werte eines Batchs tatsächlich gleich sind und die Standardabweichung daher Null ist, wird gewährleistet, dass kein Rechenfehler auftritt.

Die zu lernenden Parameter γ und β geben die Rücktransformation an, das heißt die standardisierte Verteilung wird um γ gestreckt/gestaucht und um β verschoben. Das fertig trainierte Netzwerk verwendet bei der Nutzung für ein Bild den durchschnittlichen Erwartungswert und die mittlere Standardabweichung aller Batches.
Dieser Normalisierungslayer wird nicht nur auf den Input des CNNs angewendet, sondern in der Regel auch auf Zwischensummen der meisten Convolutional und Dense Layer, also noch vor dem Anwenden der Aktivierungsfunktion. Dies ist insofern sinnvoll, als dass die ReLU-ähnlichen Aktivierungsfunktionen im Gegensatz zu den logistischen Funktionen einen beliebig hohen Output erlauben, der sich mit zunehmendem Layer immer weiter vergrößern würde. Ein weiter Grund für die Anwendung der Batch Normalization ist das später näher erläuterte Problem des Internal Covariate Shift sowie der dadurch stark beschleunigt Lernprozess.

3.5 Dense Layer

Während die bis jetzt erläuterten Layer dem Separierungsprozess dienten und im Kodierungsblock eingesetzt werden, ist das Einsatzgebiet der Fully Connected Layers der Pärdiktionsblock. Im Prinzip ließe sich an dieser Stelle zur Klassifizierung ein beliebiges passendes Machine Learning Verfahren anwenden, jedoch hat sich das Neuronale Netz zum Erfüllen dieser Aufgabe in CNNs etabliert, was nicht zuletzt ihrer Effektivität zuzuschreiben ist.
Ein einfaches Neuronales Netz hat immer einen Vektor als Input und Output. Um den dreidimensionalen Output des Prädiktionsblocks, also seinen letzten Layer, dem eindimensionalen Input des Neuronalen Netzes anzupassen, werden dessen Werte in beliebiger, einmal festgelegter Reihenfolge nacheinander als Vektor geschrieben.

Bei dem aus mehreren Dense oder auch Hidden Layers bestehendem Neuronalen Netz ist jedes Neuron eines jeden Layers mit jedem Neuron des vorherigen Layers über Gewichte verknüpft.

Der Wert eines Neurons ergibt sich daher aus den Aktivierungswerten der Neuronen im vorherigen Layer und den Gewichten zwischen dem vorherigen und aktuellen Layer.
Danach wird der errechnete Wert der Aktivierungsfunktion übergeben, dessen Ausgabe dann der Aktivierungswert des Neurons ist. Lediglich im letzten Layer, dem Output Layer ist diese Aktivierungsfunktion die Softmax Funktion.
Ein Fully Connected Layer kann theoretisch aus einer beliebigen Anzahl an Neuronen bestehen, üblicherweise ist diese Zahl jedoch eine Zweierpotenz um 2^{11}. Ein schematischer Aufbau ist in Bild 4.11 zu betrachten.

Die genaue Berechnung der Aktivierungswerte eines Neurons geschieht, indem über sämtliche Neuronen des vorherigen Layers summiert wird und deren Aktivierungswert a_i^{l-1} mit dem Gewicht $w_{j,i}^l$ der Verbindung vom i-ten zum j-ten Neuron multipliziert wird. Das Gewicht kann als den Einfluss, den das Neuron des vorherigen Layer auf das aktuelle Neuron hat, interpretiert werden. Nach dieser Aufsummierung wird dem Resultat noch ein Bias hinzuaddiert und der Wert wieder einer Aktivierungsfunktion übergeben:

$$z_j^l = b_j^l + \sum_i a_i^{l-1} w_{j,i}^l$$

$$a_j^l = f_{akt}\left(z_j^l\right)$$

Der Grund, warum die Indizes der Elemente der Gewichtsmatrix $w_{j,i}^l$ vertauscht werden ist, dass sich die zuvor erläuterte Operation auch als Matrix-Vektor Produkt und Vektoraddition schreiben lässt. Dazu müssen jedoch die Spalten die Gewichte der Neuronen enthalten, damit bei der Multiplikation von der Gewichtsmatrix mit dem Vektor der Aktivierungswerte der Neuronen die richtigen Elemente miteinander verrechnet werden. Dies bedeutet, dass der Zweite Index i eines Elements der Matrix der des Neurons des vorherigen Layers sein muss und der erste Index j der des Neurons des aktuellen Layers zu sein hat:

$$W^l = \begin{pmatrix} w_{1,1} & \cdots & w_{1,i} \\ \vdots & \ddots & \vdots \\ w_{j,1} & \cdots & w_{j,i} \end{pmatrix}$$

Allerdings scheint es einfacher und einprägsamer zu sein, die Gewichte eines Neurons Zeilenweise anzuordnen, weshalb es sich etabliert hat, die Gewichtsmatrix nicht wie oben zu schreiben, sondern sich folgende Schreibweise etabliert hat:

$$W^l = \begin{pmatrix} w_{1,1} & \cdots & w_{1,j} \\ \vdots & \ddots & \vdots \\ w_{i,1} & \cdots & w_{i,j} \end{pmatrix}$$

Dazu muss jedoch auf die Gewichtsmatrix M der Transponierungsoperator M^T angewendet werden, der dann die Indizes in die korrekte Reihenfolge bringt und eine korrekte Berechnung ermöglicht, die dann wie folgt aussieht:

$$\vec{Z}^l = \vec{B}^l + (W^l)^T \cdot \vec{A}^{l-1}$$
$$\vec{A}^l = f_{akt}(\vec{Z}^l)$$

Die Aktivierungsfunktion nimmt keinen Vektor entgegen, sondern wird auf jedes Element des Vektors $\vec{Z}^l$ angewendet. Wie den Indizes zu entnehmen ist, ist mit der Gewichtsmatrix eines Layers stets die Matrix gemeint, die die Gewichte enthält, die den vorherigen Layer mit dem aktuellen Layer verbinden. Eine schematische Abbildung eines Neurons ist in Bild 4.12 zu betrachten.

Der letzte Layer besteht aus genauso vielen Neuronen, wie das Neuronale Netz Outputs hat, denn diese Stellen die Ausgabe dar. Da der Output des CNNs die Wahrscheinlichkeit zu einer Klassenzugehörigkeit sein soll, müssen die Aktivierungen der Neuronen der letzten Schicht auf dem Intervall $[0; 1[$ liegen und die Summe der Aktivierungen muss Eins sein: $\sum_k a_k^l = 1$
Es bedarf hier also einer Wahrscheinlichkeitsverteilung, die durch die Aktivierungsfunktion der Softmax Funktion gegeben ist, welche lediglich im letzten Layer verwendet wird. Diese berechnet im Prinzip den Anteil eines Aktivierungswertes eines Neurons an den gesamten Aktivierungswerten als Wahrscheinlichkeit:

$$f_{akt}(z_i) = \frac{e^{z_i}}{\sum_k e^{z_k}}$$

Es wird nicht der direkte Wert z_i bzw. z_k genutzt, sondern eine Potenz mit diesen Exponenten, um zum einen nur Positive Summanden zu haben und zum anderen für größere Aktivierungswerte einen erhöhten Kontrast gegenüber andere Werte zu erhalten. Als Basis wird die Eulersche Zahl e verwendet, um die für den Lernprozess notwendige Ableitung einfacher zu gestalten.

4. Lernprozess

Im Lernprozess eines Neuronalen Netzes sollen die Werte, also Gewichte, Biases und Parameter, erlernt werden, die zur korrekten Klassifizierung eines Bildes notwendig sind. Dies geschieht mit Trainingsdaten, die aus vielen Input-Bildern und dem zugehörigen idealen Output des Netzwerkes bestehen. Wenn Beispielsweise auf einem Input ein Auto abgebildet ist, so ist der ideale Output die Wahrscheinlichkeit 100% für die Klasse Auto. Die restlichen Wahrscheinlichkeiten liegen dann bei 0%.

4.1 Cost Function

Als erstes gilt es festzustellen, wann ein Bild vom CNN richtig erkannt wird. Dazu bedarf es eines Vergleiches zwischen dem errechneten Output a_i und dem idealen Output y_i aus den Trainignsdaten. Dieser Vergleich ist durch eine Cost Function gegeben, auch Loss Function genannt. Eine mögliches Vorgehen für einen Vergleich wäre das Betragsquadrat der Differenz beider Werte:

$$C_i = \frac{1}{2}\|a_i - y_i\|^2$$

Der Koeffizient ½ wird lediglich vorangestellt, um die Ableitung zu vereinfachen. Dieser Vergleich muss für alle Outputs angewendet und dann der Durchschnitt gebildet werden, wodurch sich die Cost Function C ergibt, auch Error Function genannt.

$$C(a) = \frac{1}{n}\sum_i C_i = \frac{1}{2n}\sum_i \|a_i - y_i\|^2$$

Anzahl der Neuronen im letzten Layer n

Für einen Output Layer, dessen Werte auf dem Intervall $[0; 1]$ liegen, wird jedoch meistens folgende Cost Function genutzt, die auch Cross Entropy Loss genannt wird:

$$C(a) = -\sum_i y_i \cdot ln(a_i)$$

Der Grund dafür, dass diese Funktion bevorzugt wird ist, dass die Ableitung einer quadratischen Funktion für größere Fehler, das heißt für a_i nahe Null und $y_i = 1$, deutlich geringer ist als die Steigung der Logarithmusfunktion (Bild 5.1). Es sei angemerkt, dass bei dieser Cost Function im Gegensatz zu ersterer lediglich der errechnete Output für die tatsächliche Klasse des Inputs berücksichtigt wird, denn alle anderen idealen Wahrscheinlichkeiten y_i betragen Null. Dennoch haben die Neuronen mit $y_i = 0$ einen Einfluss auf die Cost Function, da ihre Aktivierung in der Softmax Funktion des Neurons mit $y_i = 1$ einfließen. Eine Loss Function gibt also an, wie gut das Netzwerk einen Input klassifiziert. Je bessere das Neuronale Netz war, desto geringer ist der Wert der Funktion.

4.2 Backpropagation

Bei dem Lernprozess wird für jeden Parameter g untersucht, welchen Einfluss dieser auf die Cost Function hat, beziehungsweise welche Änderung m dieser in der Cost Function hervorruft. Diese Änderungsrate ist die partielle Ableitung der Cost Function nach dem zu aktualisierenden Parameter: $m = \partial C / \partial g$

Bei der Berechnung dieser Größe ist die Ableitung sämtlicher verwendeter Funktionen, auch der Aktivierungsfunktion zu berücksichtigen, was der Grund dafür ist, dass dieser Aspekt bereits zuvor bei der Erläuterung der Funktionsweise zu berücksichtigen war. Um die Änderungsrate zu bestimmen, wird zunächst der Output für ein Trainingsbeispiel berechnet. Dabei werden sämtliche Inputs und Outputs der Aktivierungsfunktionen jedes Neurons gespeichert. Dieser Schritt wird Forward Pass genannt, da man praktisch vom Anfang bis zum Ende hin alle Werte berechnet, also einfach das CNN ausführt. Da das Netzwerk jedoch nicht trainiert ist, enthält der Output einen mit der Cost Function gemessenen Fehler. Dieser wird schrittweise vom Output auf die Parameter des letzten Layers, des vorletzten Layers usw. zurückgeführt. Daher der Name Backpropagation. Im letzten Schritt wird der zugehörige Fehler eines Parameters dann (teilweise) von diesem subtrahiert, wodurch dieser in Richtung

Optimum geändert wird. Der Durchgang vom Ende zum Anfang wird Backward Pass genannt. Durch dieses numerische Verfahren lernt das Netzwerk Inputs der Klasse des Trainingsbeispiels besser zu klassifizieren. Diese Schritte werden für jedes Trainingsbeispiel beziehungsweise für jede Sequenz an Trainingsdaten durchgeführt, zu Letzterem später mehr.

Eine andere Möglichkeit, den Lernprozess zu verstehen ist, sich die Cost Function als eine Funktion vorzustellen, die es zu optimieren gilt. Angenommen, man vereinfacht die Funktion auf drei Dimensionen, dann lässt sich ihr Graph als Gelände betrachten, dessen tiefster Punkt, also das globale Minima, es zu finden gilt. Mit den aktuellen Parametern befindet man sich an einem beliebigem Punkt im Gelände, der die Steigung m hat. Um dem Minima näher zu kommen, wird also einen Schritt von der jeweiligen Position aus in die entgegengesetzte Richtung der Steigung gemacht, wobei ein Schritt einem Trainingsbeispiel entspricht (Bild 5.2). Dies wird als „Gradient Descent“ bzw. „Gradientenverfahren“ bezeichnet. Das Vorgehen lässt sich beschleunigen, indem man einen Schritt nicht basierend auf einem Bild tätigt, sondern auf mehreren Bildern, also einem Batch. Dies führt dazu, dass die Schritte zwar weniger präzise, also nicht exakt in die Richtung des Minimums gerichtet sind, aber dafür größer sind, wodurch das Trainieren insgesamt schneller wird. Durch die mathematischen Betrachtung diesen Sachverhalts, ergibt sich folgende Formel für die Änderung der Parameter in Richtung des Minimums.

Jedem Parameter g wird ein Teil η der entgegengesetzten Richtung der Änderung hinzuaddiert:

$$g = g_0 + (-\eta m) = g_0 - \eta \frac{\partial C}{\partial g}$$

Für die Fully Connected Layer lässt sich diese Aktualisierung wie Folgt beschreiben. Zunächst ist in jedem Neuron von einem Fehler δ_i^l auszugehen. Dieser wird berechnet durch:

$$\delta_i^l = \frac{\partial C}{\partial z_i^l} = f'_{akt}\left(z_i^l\right) \cdot \sum_j w_{j,i}^{l+1} \delta_j^{l+1}$$

$$\delta^l = \left((w^{l+1})^T \delta^{l+1}\right) \odot f'_{akt}(z^l)$$

Da es im Letzten Layer L jedoch keinen darauffolgenden Layer mit dem Index $l+1$ gibt, wird der Fehler in dort wie Folgt berechnet:

$$\delta_i^L = \frac{\partial C}{\partial z_i^L} = f'_{akt}(z_i^L) \cdot C'(a_i^L)$$

$$\delta^L = f'_{akt}(z^l) \odot \vec{\nabla}_a C \quad \text{mit} \quad \vec{\nabla}_a C = \left(\frac{\partial C}{\partial a_1^L}, \frac{\partial C}{\partial a_2^L}, \dots, \frac{\partial C}{\partial a_j^L}\right)$$

Die Änderung der Biases ist dann gegeben durch:

$$b_i^l = b_{0,i}^l - \eta \cdot \frac{\partial C}{\partial b_i^l} = b_{0,i}^l - \eta \cdot \delta_j^l$$

$$b^l = b_0^l - \eta \cdot \delta^l$$

Hierbei wird der neue Wert des Bias b_i^l aus dem alten Wert $b_{0,i}^l$, der Lernrate η, die ein Hyperparameter ist, und dem Fehler eines Neurons berechnet.
Die Gewichte werden nach folgender Formel aktualisiert:

$$w_{j,i}^l = w_{0,j,i}^l - \eta \cdot \frac{\partial C}{\partial w_{j,i}^l} = w_{0,j,i}^l - \eta \cdot a_i^{l-1} \delta_j^l$$

Das neue Gewicht $w_{j,i}^l$, welches das i-te Neuron im Layer $l-1$ mit dem j-ten Neuron im Layer l verbindet wird aus dem alten Wert, der Aktivierung des vorherigen Neurons und dem Fehler des darauffolgenden Neurons berechnet.

Diese Gleichungen gelten jedoch nicht für die Convolutional Layer. Um die Änderungsrate der Parameter dieser Layer auf die Cost Function zu bestimmen, wird dem zu optimierenden Layer l die Änderung seines Outputs auf die Cost Function übergeben, welcher im Layer $l+1$ als die Änderung des Inputs auf die Cost Function berechnet wird. Mit dieser Vorgehensweise wird die Änderung, auch Gradient genannt, einfacher berechnet.
Für die Ableitung der Gewichte eines Filters einer Activation Map mit dem Index n im Layer l nach der Cost Function ergibt sich eine einfache zweidimensionale Convolution Operation. Bei dieser ist die Feature Map gegeben durch die Aktivierungswerte aus dem Forward Pass einer Activation Map des vorherigen Layers $l-1$, dessen Index für die z-Achse gleich dem z-Index des Gewichts ist, nach dem Abgleitet werden soll. Der Kernel in dieser Convolution Operation ist die Änderung der Zwischensumme $\partial C / \partial z_{i,j,m}^l$, die der Fehler $\delta_{i,j}^{l,m}$ ist. Auch der Kernel ist zweidimensional, denn es wird für jeden eigentlichen Filter aus dem Forward Pass eine Feature Map m berechnet. Es wird Stride und Padding aus dem Forward Pass verwendet.

$$\frac{\partial C}{\partial w_{i,j,k}^{l}} = \sum_{n_2=1}^{x_w} \sum_{n_1=1}^{x_h} \delta_{n_1,n_2}^{l} \cdot a_{I,J,k}^{l-1} \qquad I = i + n_1 s, \qquad J = j + n_2 s$$

Schritt n_1 in x-Richtung, n_2 in y-Richtung, Stride s, Breite bzw. Höhe des Kernels $x_{w,h} = m_{1,2}^{l}$, Padding p des Layers $l-1$

Der Fehler und die Änderung des Biases ist gegeben durch:

$$\delta_{i,j,k}^{l} = \frac{\partial C}{\partial a_{i,j,k}^{l}} \cdot f'_{akt}\left(z_{i,j,k}^{l}\right) \qquad \frac{\partial C}{\partial b^{l}} = \delta_{i,j,k}^{l}$$

Die Ableitung des Inputs nach der Cost Function ist ebenfalls eine Convolutional Operation, jedoch mit der Änderung des Outputs des aktuellen Layers als Feature Map. Diese ist dabei mit einem Zero-Padding der Breite von $k-1$ versehen. Der um 180° um die z-Achse rotierten eigentlichen Kernel mit den Gewichten dient nun als Filter, der wie Folgendermaßen rotiert wird:

$$\begin{pmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \end{pmatrix} \Rightarrow W = \begin{pmatrix} w_{2,2} & w_{2,1} \\ w_{1,2} & w_{1,1} \end{pmatrix}$$

Daraus ergibt sich folgende Convolution Operation:

$$\frac{\partial C}{\partial a_{i,j,p}^{l-1}} = \sum_{f=1}^{m_3^l} \sum_{n_2=m_j}^{m_j+v_j-1} \sum_{n_1=m_i}^{m_i+v_i-1} \delta_{n_2,n_1}^{l,f} \cdot w_{I,J,p}^{l,f} \qquad I = i - n_1 s + 1 \qquad J = j - n_2 s + 1$$

Für den Backward Pass in der

Die Herleitungen sämtlicher in diesem Abschnitt gegebenen Formeln sind im Anhang zu finden.

4.3 Vanishing & Exploding Gradients

Die Parameter eines Neuronalen Netzes werden je nach ihrem Gradient aktualisiert. Wenn diese Änderung jedoch äußerst klein ist und der zugehörige Parameter daher beim Aktualisieren nur geringfügig verändert wird, spricht man vom Vanishing Gradient Problem. Analog führen sehr hohe Änderungen zum Exploding Gradient Problem, bei dem die Parameter zu sehr abgeändert werden und nicht in Richtung des Optimums konvergieren. Der Gradient eines Parameters $\partial C / \partial g$ wird mithilfe der Kettenregel berechnet:

$$\frac{\partial C}{\partial g^{L-n}} = \frac{\partial C}{\partial a^{L}} \cdot \frac{\partial a^{L}}{\partial a^{L-1}} \cdot \frac{\partial a^{L-1}}{\partial a^{L-2}} \cdot \ldots \cdot \frac{\partial a^{L-n+1}}{\partial a^{L-n}} \cdot \frac{\partial a^{L-n}}{\partial g^{L-n}}$$

Wenn viele der Faktoren des Produkts kleiner als Eins sind, so ergeben sich bei einer Multiplikation noch kleinere Werte. Je weiter ein Neuron vom Output entfernt ist, umso mehr Faktoren, die potentiell kleiner als Eins sind, werden zur Berechnung des Gradients benötigt. Daher tritt das zugrunde liegende Problem eher in Layern auf, die weiter vom Output entfernt sind.
Ein gewichtiger Grund, warum die Faktoren äußerst klein sind ist, dass diese durch die Ableitung der Aktivierungsfunktion berechnet werden. Wenn man etwa die Sigmoid oder Tangens Hyperbolicus Funktion verwendet, ist die Steigung mit zunehmender Entfernung vom Wendepunkt sehr gering, und meist kleiner als Eins. Um das Problems zu vermeiden ist daher die Verwendung von anderen Funktionen ohne dieses Phänomen, etwa der ReLU Funktion notwendig, dessen Ableitung entweder Null oder Eins ist. Zudem ist es unumgänglich das Problem bei der Initialisierung der Parameter zu berücksichtigen.

4.4 Initialisierung der Parameter

Die Gewichte in einem Neuronalen Netz werden zu Beginn zufällig gewählt. Die Werte sind dabei normalverteilt mit einem Erwartungswert von $\mu = 0$. Die Wahl der Standardabweichung σ ist essentiell zur Vermeidung des Vanishing und Exploding Gradient Problems. Angenommen man initialisiert Die Gewichte in einem Dense Layer, die zu einem Neuron führen mit $\sigma = 1$. Die Aktivierungswerte der Neuronen des vorherigen Layers seien alle etwa bei Eins, sodass die Zwischensumme des zu untersuchenden Neurons die Summe der Gewichte ist. Unter der Berücksichtigung, dass die Varianz der Summe stochastisch unabhängiger Zahlen gleich der Summe der Varianz jeden Summanden ist, ergäbe sich eine sehr hohe Varianz, die dazu führt, dass der Input der Aktivierungsfunktion weit von der Ordinate entfernt ist, was für logistische Aktivierungsfunktionen einen geringen Wert der Ableitung an dieser Stelle hervorruft. Aufgrund dessen werden für verschiedene Funktionen verschiedene Standardabweichungen beim Initialisieren verwendet. Für die ReLU Funktion etwa wird die von Kaiming He entwickelte He Initialisierung verwendet. Diese wird im Anhang hergeleitet und sieht Folgendes vor:

$$\sigma = \sqrt{\frac{2}{n^{l-1}}}$$

Hierbei gibt n^{l-1} die Anzahl an Neuronen im vorherigen Dense Layer an, wenn ein Gewicht w^l initialisiert werden soll. Da ein Layer viele Neuronen hat, ist die Standardabweichung relativ klein, was bedeutet, dass sämtliche Gewichte nahe um Null verteilt sind, ebenso wie ihre Summe. Für die Convolutional Layer ist zu berücksichtigen, dass n der Anzahl an Gewichten in einem Kernel entspricht, denn dies ist die Anzahl an Neuronen, die in einem Schritt einer Convolution Operation berücksichtigt werden: $n^{l-1} = k^2 m_3^{l-1}$

Die Biases werden üblicherweise mit Null initialisiert, es kann aber auch eine Gleichverteilung zwischen Null und Eins verwendet werden. Für die anderen zu erlernenden Parameter werden bestimmte, dem Kontext angepasste, meist konstante Initialisierungswerte vergeben.

4.5 Durchführung

Damit das CNN lernen kann, bedarf es vieler Trainingsbeispiele. Eine Möglichkeit diese zu nutzen, ist mehrmals für jedes Beispiel den Forward sowie Backward Pass durchzuführen und die Parameter zu aktualisieren. Dies wird als Stochastisches Gradientenabstiegsverfahren bezeichnet, ist aber für größere Datensätze äußerst rechenintensiv, weil für jedes der mehreren Millionen Bilder die Parameter zu aktualisieren sind. Um das Netzwerk effizienter zu trainieren, werden die Trainingsdaten in Mini-Batches unterteilt, die aus 2^6 bis 2^8 zufällig gewählten Bildern bestehen. Für jedes Element dieser Sequenz wird ein Forward Pass durchgeführt, aber für den gesamten Batch der Länge m nur ein Backward Pass, bei dem die Parameter g mithilfe des durchschnittlichen Gradients, der sich aus dem Batch ergibt, aktualisiert werden:

$$g = g' - \frac{\eta}{m} \sum_{i=0}^{m} \frac{\partial C_m}{\partial g}$$

Des Weiteren ist es üblich, vor Beginn des Trainings die Daten jedes Bildes um Null zu verteilen, indem von jedem Pixel der Erwartungswert, der sich aus sämtlichen Trainingsdaten ergibt, subtrahiert wird. Diese Zentrierung um Null ist insofern sinnvoll, als dass die Ableitung der Gewichte durch das Produkt aus dem vom folgenden Layer übergebenen Gradient $\partial C/\partial z$ und dem lokalen Gradienten $\partial z/\partial w$ gegeben ist. Letzterer ist der Input:

$$\frac{\partial}{\partial w}[b + Wa] = a$$

Wenn der Input nicht um Null zentriert wäre, also eher positiv oder eher negativ ist, auch als Covariate Shift bezeichnet, so werden die Gewichte alle in Abhängigkeit des Vorzeichens des Gradients $\partial C/\partial z$ in eine einheitliche Richtung aktualisiert, alle Werte werden entweder größer oder kleiner. Möglicherweise ist es aber notwendig, dass einige Gewichte in die eine und andere in die entgegengesetzte Richtung aktualisiert werden. Damit dadurch nicht nur die Gewichte des letzten Layers profitieren, sondern auch die in sämtlichen anderen Layern, wird die Batch Normalisierung nach jedem Output eines Layers angewendet, sodass sämtliche Inputs sämtlicher Layer um Null verteilt sind.
Wenn das CNN genutzt wird, werden jedoch keine Batches verwendet, sondern es liegt nur ein Bild als Output vor, weshalb weder ein Erwartungswert noch eine Varianz berechnet werden kann. Dennoch werden diese beiden Größen im Batch Normalization Layer benötigt. Daher verwendet man als Erwartungswert den durchschnittlichen Erwartungswert aller Batches nachdem das Netzwerk bereits trainiert wurde. Gleiches gilt für die Varianz, hinzu kommt lediglich, dass diese noch mit einem Koeffizienten multipliziert wird, der sich aus der absoluten Größe eines Batches m ergibt, um eine Unverzerrtheit zu gewährleisten:

$$\mu = \frac{1}{n}\sum_{i=1}^{n}\mu_i$$

$$\sigma^2 = \frac{m}{m-1}\cdot\frac{1}{n}\sum_{i=1}^{n}\sigma_i^2$$

Insgesamt liegt für das Trainieren eines CNNs eine gewisse Anzahl an Beispielen, also Bildern mit ihrem zugehörigen Label, welches die Zugehörigkeit zu einer bestimmten Klasse angibt, vor. Diese Bilder werden in drei Datensätze unterteilt: Die Trainings-, Validierungs- und Testdaten. Von ersterem wird ein Teil genutzt, mit dem das Netzwerk mehrmals mit verschiedenen Hyperparametern trainiert wird. Die verschiedenen resultierenden CNN Modelle werden dann mithilfe der Validierungsdaten hinsichtlich ihrer Leistung begutachtet. Das Modell, welches am besten mit dem Validierungsset gearbeitet hat, wird dann

erneut mit den vollständigen Trainingsdaten trainiert. Zum Schluss wird dieses Modell dann mit den Testdaten getestet.

4.6 Regularisierung

Es gibt mehrere Konstellationen der Parameter, die ein Minima der Cost Function darstellen, das heißt es existieren verschiedene Anordnungen der Werte für Gewichte, die für die Trainingsdaten gleiche Resultate erbringen. Die Resultate für die Testdaten können jedoch deutlich schlechter sein, wenn die Gewichte zu sehr darauf ausgelegt werden, die Trainingsbilder korrekt zu klassifizieren. Dies wird als Overfitting bezeichnet. Mithilfe von Regularisierung wird versucht dies zu verhindern. Dies kann etwa geschehen, indem man der Loss Function L einen Wert hinzuaddiert, der sich aus sämtlichen Gewichten ergibt.

$$L = C + R(W)$$

Wenn die Gewichte kleiner sind, kann dies so verstanden werden, dass diese die einfachere Lösung sind und der Fehler durch die Regularisierung geringer ist, als mit einer anderen Wahl. Man unterscheidet zwischen L1 und L2 Regularisierung.

$$L2\colon\ R(W) = \lambda \sum_{l} \|W^l\|_F^2$$

$$L1\colon\ R(W) = \lambda \sum_{l} \sum_{i} \left|w_i^l\right|$$

Hierbei gibt λ an, wie sehr die Gewichte bei der Regularisierung zu berücksichtigen sind.

Eine weitere Möglichkeit dem Overfitting entgegenzuwirken ist das Early Stopping. Bei dieser Methode wird untersucht, an welchem Punkt im Trainingsprozess der Trainingsfehler zwar sinkt, der Testfehler aber sein Minimum hat und beginnt zu steigen (Bild 5.6). Die zu diesem Zeitpunkt verwendeten Parameter werden dann für das fertige CNN verwendet.

Der Gebrauch von Dropout Layern ist ebenfalls eine Regularisierung. Dabei werden meist in den Dense Layers die Aktivierungswerte zufälliger Neuronen auf null gesetzt, sodass diese nicht in der weiteren Berechnung berücksichtigt werden. In der Regel werden 20% der Neuronen deaktiviert. Dieses Vorgehen kann so verstanden werden, dass das Netzwerk sich nicht von bestimmten Neuronen abhängig machen kann, weil diese unter Umständen wegfallen, sodass die Klassifizierung nicht zu sehr von einer bestimmten Eigenschaft,

welche von einem Neuron repräsentiert wird, die etwa für eine Klasse nur in den Trainingsbildern, nicht aber in den Testbildern vorhanden ist, abhängt.
Da die Methode des Dropouts durch die Zufälligkeit es theoretisch ermöglichen würde, dass ein Input bei unterschiedlichen Durchläufen unterschiedlich klassifiziert wird, muss beim Testen und zur Anwendungszeit darauf verzichtet werden. Wenn im Zuge dessen dann sämtliche Neuronen berücksichtigt werden, so ist der Input des folgenden Layers in einer anderen Größenordnung, als während der Trainingszeit, in der einige Aktivierungswerte bei null liegen. Um in beiden Phasen die gleiche Größenordnung zu erzielen, wird während der Verwendung des Dropouts die Zwischensumme im nächsten Layer erhöht, indem diese mit der Wahrscheinlichkeit, dass ein Neuron aktiv ist, dividiert wird.

4.7 Optimierung

Beim Ausführen des Lernprozess mit bis hierhin erläuterten Mitteln lässt bestehen noch einige Problematiken. Die gewichtigste ist, dass die Parameter je nach Steigung angepasst werden. Dabei kann es passieren, dass man sich zur Zeit eines Lernschrittes in einem lokalen Minima oder an einem Sattelpunkt befindet, an denen die Steigung Null ist, was das Lernen verhindert. Der Lernprozess lässt sich in dieser Hinsicht jedoch weiter optimieren. Eine Möglichkeit dazu wäre die Art, mit der die Parameter aktualisiert werden, zu ändern. Dies ließe sich mit der Momentum Optimierung vollbringen. Wenn man die aktuellen Parameter als ein Objekt im Gelände betrachtet, welches zum niedrigsten Punt, dem Tal, gebracht werden soll, an dem der Fehler in der Loss Function den geringsten Wert hat, dann aktualisiert die Momentum Optimierung die Objekte nicht nach der Steigung des Ortes, an dem sie sich zurzeit befinden, sondern anhand der Geschwindigkeit v des Objekts, die sich aus der Steigung und der Geschwindigkeit im vorherigen Lernschritt ergibt.
Die kann wie folgt ausgedrückt werden:

$$v_t = \gamma v_{t-1} + \frac{\partial C}{\partial g}$$

$$g = g' - \eta \cdot v_t$$

Die Geschwindigkeit im ersten Lernschritt ist $v_1 = 0$. Der Hyperparameter γ kann als Reibung verstanden werden und gibt an, wie sehr die vorherige

Geschwindigkeit gewichtete wird. In der Regel wird für γ ein Wert zwischen $0{,}9$ und $0{,}99$ gewählt.

Eine weiterer weit verbreiteter Ansatz ist die Adaptive Momentum Optimierung, kurz Adam Optimierung. Diese sieht vor, dass noch eine weitere Geschwindigkeit s berechnet wird, die sich aber aus dem Quadrat der partiellen Ableitung der Cost Function nach dem Parameter ergibt. Dieser wird dann unter Berücksichtigung des Verhältnisses von erster zu zweiter Geschwindigkeit abgeändert. Weil der Impuls des Objekts jedoch mit Null initialisiert wird, hätte dieser zu Beginn einen zu geringen Einfluss. Um dem entgegenzuwirken wird die Geschwindigkeit zu Beginn hochkorrigiert. Diese Korrektur wird zwar durchgehend durchgeführt, ist aber nur in den ersten Schritten relevant, da sie exponentiell abnimmt.

$$v_t = \beta_1 \cdot v_{t-1} + (1 - \beta_1) \cdot \frac{\partial C}{\partial g}$$

$$s_t = \beta_2 \cdot s_{t-1} + (1 - \beta_2) \cdot \left(\frac{\partial C}{\partial g}\right)^2$$

$$\hat{v}_t = \frac{v_t}{1 - (\beta_1)^t}$$

$$\hat{s}_t = \frac{s_t}{1 - (\beta_2)^t}$$

$$g = g' - \eta \cdot \frac{\hat{v}_t}{\sqrt{\hat{s}_t + \delta}}$$

Für die Hyperparameter wird in der Regel $\beta_1 = 0{,}9$, $\beta_2 = 0{,}999$ und $\eta = 10^{-3}$ oder $\eta = 5 \cdot 10^{-4}$ gewählt.

5. Eigener Ansatz

Ein großes Problem von Neuronalen Netzen ist die immense Anzahl an Parametern, insbesondere in den Fully Connected Layern. Diese machen das Netzwerk weniger speichereffizient. Eine drastische Reduzierung der Gewichte, die nahezu sämtliche Parameter eines Neuronalen Netzes darstellen, ließe sich mit einer räumlichen Verteilung der Neuronen im Dense Layer erzielen.

Die Idee meines Ansatzes ist eine Reduzierung der Parameter, nicht aber indem die Anzahl der Gewichte verringert wird, sondern indem die Gewichte zur Laufzeit individuell berechnet werden, je nachdem welche Neuronen diese miteinander verbinden. Ein Gewicht ergibt sich aus den Positionen beider Neuronen im k-

dimensionalen Raum sowie einer Gewichtsfunktion mit zwei zu erlernenden Parametern, welche aus der Entfernung der Neuronen ein Gewicht berechnet. In herkömmlichen Dense Layern ist jedes Neuron des Layers l mit jedem Neuron des Layers $l-1$ verbunden, weshalb die Anzahl der Gewichte zwischen diesen Layern $n^{l-1} \cdot n^l$ beträgt, wobei n^l die Zahl der Neuronen des Layers l angibt.
In einem k-dimensionalen Fully Connected Layer ist jedem Neuron beider Layer eine Position zugeordnet, die durch k zu erlernende Ortswerte festgelegt wird. Zusätzlich ist jedem Neuron noch zwei zu erlernende Parameter für die Gewichtsfunktion zuzuordnen. Eine Position eines Neurons im Layer l wird außer im ersten und letzten Layer zur Bestimmung von $n^{l-1} + n^{l+1}$ Gewichten verwendet, zum einen für den Input vom vorherigen Layer und zum anderen für den Output des folgenden Layers. Die massiven Ersparnisse von zu erlernenden Parametern sei Beispielhaft an den drei Fully Connected Layern des VGG-Net demonstriert, dessen Gewichte etwa 90% aller Parameter ausmachen.
Die tatsächliche Anzahl an gewichten in den Dense Layern diesen Netzwerks ist:

$$n^1 \cdot n^2 + n^2 \cdot n^3 + n^3 \cdot n^4 = 7 \cdot 7 \cdot 512 \cdot 2^{12} + 2^{12} \cdot 2^{12} + 2^{12} \cdot 10^3 = 123.633.664$$

Das gleiche Neuronale Netz mit $k = 3$ räumlichen Dimensionen ergäbe folgende Anzahl an zu lernenden Parametern:

$$(3n^1 + 5n^2) + 5n^3 + 5n^4 = (3 \cdot 7 \cdot 7 \cdot 512 + 5 \cdot 2^{12}) + 5 \cdot 2^{12} + 5 \cdot 10^3 = 121.224$$

Auf diese Art lässt sich die Anzahl der Parameter für diesen Fall auf weniger als $0{,}1\%$ senken.

5.1 Funktionsweise

Um die Entfernung zwischen zwei Neuronen mit den Koordinaten x im k-dimensionalen Raum zu berechnen wird folgende Formel herangezogen:

$$d(x^{l-1}, x^l) = \sqrt{\sum_k \left(x_k^{l-1} - x_k^l\right)^2}$$

Das Gewicht in Abhängigkeit von der Entfernung d nimmt mit einer Zunahme dieser ab und ist gegeben durch die Gewichtsfunktion:

$$w(d) = \beta - \alpha\sqrt{d + \delta}$$

Der geringe Wert $\delta = 0{,}1$ wird hinzugefügt, um die Funktion für $d = 0$ differenzierbar zu machen. Im Vergleich zu anderen zuvor erwähnten Werten die dieselbe Funktion wie dieser Wert haben, ist δ hierbei relativ groß gewählt, damit

die Ableitung für Entfernungen um Null nicht allzu groß ist und die Parameter möglicherweise zu stark aktualisiert werden. Die Parameter α und β sind zu erlernen, wenn α jedoch kleiner Null sein sollte, so nimmt das Gewicht mit der Entfernung zu.

Ein Problem mit diesem Layertyp ist die gegenüber den herkömmlichen Dense Layern signifikante Erhöhung der erforderlichen Rechenleistung, die in erster Linie durch die Quadratwurzel-Operationen in der Gewichtsfunktion und Bestimmung der Entfernung verursacht wird. Daher ist es sinnvoll die Funktion zu Bestimmung der Gewichte wie Folgt zu vereinfachen.

Bei dem Entwerfen dieser Funktion wurde darauf geachtet, dass die Änderung der Gewichtsabnahme mit zunehmender Entfernung abnimmt, ohne dabei zu konvergieren. Diese Eigenschaft erweist sich jedoch als obsolet, weil eine Betrachtung von den Gewichten eines bereits trainierten Netzwerkes ergibt, dass ihr Betrag meist auf dem Intervall $[0; 1]$ liegt, sodass auch die Entfernungen nicht allzu stark vom Initialisierungswert abweichen, weshalb besagte Eigenschaft der Gewichtsfunktion nicht zum Tragen kommt. Dies bietet die Möglichkeit der Vereinfachung zu einer linearen Funktion:

$$w = \alpha d + \beta$$

Es ist zwar intuitiver, wenn die Gewichte mit zunehmender Entfernung abnehmen, aber nicht notwendig, weshalb α positiv sein kann.

Dadurch werden die Berechnungen sowohl für den Forward- als auch Backward Pass einfacher und somit schnelle.

5.2 Backpropagation

Um den Einfluss der Parameter auf die Loss Function zu berechnen, muss zunächst, wie es bereits im Abschnitt Lernprozess erläutert wurde, der Einfluss des Gewichts $\partial C / \partial w$ berechnet werden. Dies wird als gegeben betrachtet.

Die Parameter werden wie alle anderen aktualisiert. Für ihre Änderung auf die Cost Function gilt:

$$\frac{\partial C}{\partial \beta_j^l} = \sum_{i=1}^{n^{l-1}} \frac{dC}{dw_{j,i}^l}$$

$$\frac{\partial C}{\partial \alpha_j^l} = -\sum_{i=1}^{n^{l-1}} \frac{dC}{dw_{j,i}^l} \cdot \sqrt{d(x_i^{l-1}, x_j^l) + \delta}$$

$$l = L: \quad \frac{\partial C}{\partial x_{k'}^{L}} = \sum_{i=1}^{n^{L-1}} \frac{\partial C}{\partial w_{j,i}^{L}} \cdot \frac{-\alpha_j^L}{2\sqrt{d(x_i^{L-1}, x_j^L) + \delta}} \cdot \frac{x_{k'}^L - x_{k'}^{L-1}}{d(x_i^{L-1}, x_j^L)}$$

$$l = 1: \quad \frac{\partial C}{\partial x_{k'}^{1}} = \sum_{j=1}^{n^2} \frac{\partial C}{\partial w_{j,i}^{2}} \cdot \frac{-\alpha_j^2}{2\sqrt{d(x_i^1, x_j^2) + \delta}} \cdot \frac{x_{k'}^1 - x_{k'}^2}{d(x_i^1, x_j^2)}$$

$$sonst: \quad \frac{\partial C}{\partial x_{k'}^{l}} = \left[\sum_{i=1}^{n^{l-1}} \frac{\partial C}{\partial w_{j,i}^{l}} \cdot \frac{-\alpha_j^l}{2\sqrt{d(x_i^{l-1}, x_j^l) + \delta}} \cdot \frac{x_{k'}^l - x_{k'}^{l-1}}{d(x_i^{l-1}, x_j^l)} \right] + \sum_{i=1}^{n^{l+1}} \frac{\partial C}{\partial w_{i,j}^{l+1}} \cdot \frac{-\alpha_i^{l+1}}{2\sqrt{d(x_j^l, x_i^{l+1}) + \delta}} \cdot \frac{x_{k'}^l - x_{k'}^{l+1}}{d(x_j^l, x_i^{l+1})}$$

Es sei angemerkt, dass x_i^l die Koordinate des Neurons i ist, die aus k Komponenten besteht, welche jeweils als $x_{k'}^l$ bezeichnet werden.

Im Falle der linearen Gewichtsfunktion können die mittleren Terme in den Summen ähnlich $-\alpha_j^l / 2\sqrt{d(x_i^{l-1}, x_j^l) + \delta}$ durch α_j^l substituiert werden.

5.3 Initialisierung der Parameter

Ein guter Startwert für die Parameter der ersten, nicht-linearen Gewichtsfunktion ist $\alpha = 2$ und $\beta = 5$. In der linearen Funktion gibt α die Steigung an, also wie sehr sich das Gewicht mit einer neuen Distanz ändert. Der Parameter β ist so zu wählen, dass die Nullstelle der Erwartungswert der Entfernung ist. Passende Werte sind daher $\alpha = 1$ und $0 = \alpha d + \beta \Leftrightarrow \beta = -\alpha d$.

Die Koordinaten der Neuronen sind zu Beginn so zu setzen, dass die resultierenden Gewichte stochastisch genauso verteilt sind, wie bei der Nutzung normaler Dense Layer, das heißt der Erwartungswert sollte $\mu = 0$ und die Standardabweichung im Falle der ReLU Aktivierungsfunktion $\sigma = \sqrt{2/n}$ betragen. Hierbei gibt n die Anzahl der Input-Gewichte, also die Anzahl an Neuronen im vorherigen Layer an. Weil sich die Gewichte durch die Entfernung ergeben, ist lediglich die relative Positionierung der Neuronen im Raum relevant. Daher werden die Koordinaten der Neuronen des ersten Layers willkürlich auf den Ursprung des Raumes gesetzt. Für die Neuronen aller anderen Layer

werden sämtliche Koordinatenwerte bis auf eine einheitliche räumliche Achse mit Null initialisiert, sodass sämtliche Neuronen auf einer Linie liegen. Ein Koordinatenwert der gewählten räumlichen Achse wird in den anderen Layern mit der Summe aus dem Erwartungswert des vorherigen Layers und einem mit der Verteilungsfunktion $F_D(x)$ für die Distanz ermittelten Wert berechnet:

$$x_{k'} = \mu^{l-1} + d$$

Dadurch werden die Koordinaten so gewählt, dass sich für die resultierenden Gewichte die normalerweise verwendete Wahrscheinlichkeitsverteilung ergibt. Das konkrete mathematische Vorgehen dafür wird im Anhang näher erläutert.

Für den Backpropagation Algorithmus hat sich jedoch herausgestellt, dass die Ableitung nach der x_k-Koordinate eines Neurons abhängig von der Differenz der Koordinaten der k-ten Dimension zweier Neuronen ist. Für die zuvor beschrieben Initialisierungsmethode bedeutet dies, dass die Differenz in jeder Dimension bis auf die der gewählten Koordinatenachse null ist und somit auch die Ableitung null ist. Dies führt dazu, dass das Verwenden von beliebig vielen räumlichen Dimensionen äquivalent zu dem Verwenden einer einzigen räumlichen Dimension ist.

Im Falle eines dreidimensionalen Raumes bietet es sich an, die Position der Neuronen wie zuvor zu initialisieren, dabei aber die Neuronen jeden zweiten Layers gleichmäßig auf einem Ring zu verteilen, der senkrecht zur gewählten Koordinatenachse steht. Der Radius r entspricht dabei der Differenz zwischen dem Erwartungswert der vorherigen Layers und der x_k-Komponente der Neuronen, wobei k die gewählte Koordinatenachse ist. r wird dabei so gewählt, dass die Entfernung der Neuronen zweier Layer noch immer der aus der Funktion $F_D(x)$ berechneten Distanz entspricht (Bild 5.0).

Diese Initialisierung führt jedoch zum einen dazu, dass die durchschnittliche Distanz je nach Koordinatenachse abweicht, sodass der Lerneffekt nicht vollständig ausgenutzt werden kann. Zum anderen ist der Gradient der Koordinaten der Neuronen gegenüber den Parametern α und β der Gewichtsfunktion deutlich geringer, wodurch das Lernen weniger durch eine Änderung in der Position, sondern eher in den Parametern hervorgerufen wird. Um diesem Effekten entgegenzuwirken, bietet es sich an, die Neuronen jeden zweiten Layers auf der Oberfläche einer Kugel mit dem Koordinatenursprung als

Mittelpunkt und der Distanz aus $F_D(x)$ als Radius zu verteilen. Die restlichen Koordinaten sind um den Koordinatenursprung zu initialisieren. Auf diese Weise ist der Erwartungswert der Gradients für jede Dimensionskomponente gleich und die Position der Neuronen trägt mehr zum Lernen bei.
Eine genauere mathematisch Begründung dessen ist im Anhang gegeben.

6. Programmiertechnische Umsetzung

Der Folgende Abschnitt ist im Rahmen der Leitfrage entsprechend kurz gehalten. Das Programmieren eines Convolutional Neural Network erfolgt in der Programmiersprache C++ und orientiert sich am objektorientierten Entwicklungsmuster und macht Gebracht von den Ideen des generischen Programmierens und der Polymorphie. Der Anspruch des Programms besteht darin, es wie eine Library nutzen zu können und nach Belieben eigene Netzwerkarchitekturen zusammenbauen zu können. Auf einen einfachen Befehl hin lässt sich das Netzwerk trainieren und exportieren oder auch ein bereits bestehendes Netzwerk importieren und nutzen.
Das Programm ist dabei so aufgebaut (Bild 6.1), dass die Layertypen, d.h. Convolutional-, Dense-, Pooling-, Flatten- und mehrdimensionale Dense Layer, je durch eine gleichnamige Klasse repräsentiert werden, die sich von der abstrakten Klasse `Layer` ableitet und die Funktionalitäten eines Layers besitzen. Als Schnittstelle zwischen dem programmierenden, das Netzwerk erstellenden Nutzer und den Layern existiert die Klasse Network, mit dessen Hilfe das Neuronale Netz zusammengebaut werden kann, in welchem Instanzen der Layer-Klassen erstellt und miteinander verknüpft werden. Dies bedeutet für den Forward Pass, dass als Input eines Layers der Output eines vorherigen Layers festgelegt wird, beziehungsweise für den Backward Pass, dass der Input der Output des nächsten Layers ist. Die Verknüpfung erfolgt, indem den Klassen die Speicheradresse des Inputs übergeben wird, sodass auf diesen Klassenübergreifend zugegriffen werden kann und das Verlinken nur einmal notwendig ist.
Um das Netzwerk zu trainieren, verfügt der Nutzer über die Klasse Trainer, die die Trainingsdaten verarbeitet und der Klasse Network übergibt, die den Backward Pass in den Memberfunktionen der Layer-Klassen veranlasst.

Des Weiteren existieren noch die statischen Klassen `FileManager` zum einlesen und bearbeiten von Bildern, `jExporter` zum Ex- und Importieren des Netzwerks als Hybrid-JSON Datei und `MathCNN` für den allgemeinen Teil der erforderlichen internen Berechnungen.
Aus Speicherkapazitätsgründen werden die temporären Daten für den Forward- und Backward Pass auf einem statischen Laufwerk und nicht dem Arbeitsspeicher gespeichert.
Insbesondere der Backward Pass durch die Convolutional Layer ist besonders rechenintensiv, weshalb dieser Teil in zwei Aufgaben unterteilt wird, die wiederum jeweils auf vier Threads aufgeteilt werden.
Eine bei weitem größere Herausforderung als das Programmieren der eigentlichen Funktionalität, also der tatsächlichen Berechnungen, die hier erläutert werden, ist der Umgang mit den großen Datenmengen sowohl vor der Ausführung des Programms als auch während der Laufzeit, zu welcher das Programm mit diesen Daten rechnen muss. Für dieses Problem gibt es zwei Lösungsansätze: Zum einen ließe sich die die Batch-Größe, das heißt die Anzahl an Bildern, die parallel in das Netzwerk gegeben werden, reduzieren, sodass die Kapazität des Arbeitsspeichers zum Rechnen genügt. Dies hat den Vorteil, dass die Berechnungen schnell sind, weil der Zugriff auf den RAM entsprechend schnell ist. Allerdings besteht der Nachteil darin, dass eine kleiner Batch-Größe das Training verlangsamt, wie im Abschnitt Lernprozess erläutert. Der Ansatz der zweiten Option ist dem ersten Lösungsansatz genau entgegengesetzt, das heißt, es wird die erforderliche Trainingszeit verringert, indem die Batch-Größe erhöht wird, aber die Berechnungen verlangsamen sich, weil der verfügbare Arbeitsspeicher nicht mehr ausreichend groß ist, sodass dieser auf die Festplatte ausgeweitet werden muss, von der sich langsamer lesen und schreiben lässt als vom RAM. Dieser Effekt lässt sich durch das Nutzen einer SSD jedoch etwas kompensieren. Ein Testlauf bestätigt, dass insgesamt die Verbesserung in der Trainingsgeschwindigkeit der in der Rechengeschwindigkeit überwiegt, sodass die zweite Option, für die ich mich auch entschieden habe, vorteilhafter ist.
Eine Übersicht über die wichtigsten Funktionen für den Nutzer ist in Bild 6.2.

7. Fazit

Im Folgenden sollen die Ergebnisse der praktischen Umsetzung der zuvor erläuterten Inhalte diskutiert werden.

Zunächst gilt es zu beurteilen, inwiefern mein eigener Ansatz eines Dense Layers einen solchen ersetzen kann. Dazu habe ich drei Versionen eines Neuronalen Netzes programmiert. Die künstlichen Intelligenzen bestehen jeweils ausschließlich aus Dense Layern (Bild 7.1), mehrdimensionalen Dense Layern mit einer nicht-linearen Gewichtsfunktion und der Riniginitialisierung (Bild 7.2) oder mehrdimensionalen Dense Layern mit einer linearen Gewichtsfunktion und einer Kugelinitialisierung (Bild 7.3). Sämtliche Netzwerke haben die gleiche Architektur und wurden mit gleichen Einstellungen mithilfe des sog. MNIST-Datensatzes darauf trainiert, handschriftlich geschriebene Ziffern zu erkennen. Die Resultate des Training sind als Graph des Fehlers in Abhängigkeit vom Durchgang in den zuvor verwiesenen Bildern dargestellt.

Es ist festzustellen, dass keine der Varianten des eigenen Ansatzes mit den Ergebnissen des herkömmlichen Dense Layers mithalten kann, dessen Fehler gegen 0,9 konvergiert, während die Fehler der anderen Netzwerke um 2.0 herum konvergieren. Unter Abwägung der starken Ersparnisse in den Rechenressourcen mit dem Fehler ist mein eigener Ansatz jedoch deutlich effizienter als die normalen Dense Layer und für weniger komplexe Problemstellungen sicherlich deutlich besser geeignet.

Aus den Graphen für die mehrdimensionalen Dense Layer geht auch hervor, dass das Verwenden einer linearen Gewichtsfunktion und einer Kugelinitialisierung einen deutlichen Vorteil gegenüber der anderen Variante hat, weil der Fehler mit 1,9 deutlich geringer ist als der der anderen Variante von ca. 2,3.

Des Weiteren habe ich ein Convolutional Neural Network programmiert, welches mithilfe des ImageNet Datensatzes darauf trainiert wurde, die Objekte Füller, Tisch, Fernbedienung und Fernseher zu klassifizieren. Das Ergebnis von über 110 Stunden Trainingsdauer ist in Bild 7.4 abgebildet. Festzustellen ist dabei, dass der Fehler erst nachdem alle 5.200 Trainingsbilder das erste Mal nach ca. 160 Durchgängen berücksichtigt wurden, stärker zu fallen beginnt. Dies ist darauf zurückzuführen, dass das Netzwerk alle Folgenden Bilder bereits einmal

durchgegangen ist und diese bereits etwas kannte, weshalb diese im weiteren Verlauf besser klassifiziert werden können, sodass der Fehler insgesamt geringer ist.

Meine Motivation für das Thema Künstlichen Intelligenz stammt aus dem zweiten Semesterthema des Fachs Informatik, bei dem ich zum ersten Mal in Kontakt mit dem Bereich maschinelles Lernen gekommen bin, was mein Interesse geweckt, weil es sehr gut die beiden Themengebiete Informatik und Mathematik abdeckt, die mich generell interessieren. Außerdem fasziniert es mich, wie es möglich ist, etwas zu programmieren, ohne dabei Algorithmen mit logischen Operatoren zu verwenden, die direkt darauf ausgelegt sind, ein bestimmtes Problem zu lösen. Neuronale Netze hingegen können mit dem gleichen Programmcode verschiedene Dinge erlenen, vom Lesen der Handschrift bis zu abstrakten mathematischen Problemen wie das Erlernen eines Funktionsterms.
Aus dem gesamten Projekt habe ich weit mehr gelernt als nur das inhaltliche dieser Arbeit, sei es das Programmieren in C++, die Vertiefung in mathematische Themengebiete, das Schreiben eines längeren wissenschaftlichen Textes wie diesen hier oder die Methodik und Herangehensweise für ein solches Unterfangen.

Hiermit erkläre ich, Nabil Salama, dass die vorliegende Arbeit einschließlich des Programmcodes (Libraries zwecks anderem als die zu zeigende Funktionalität ausgeschlossen) ausschließlich von mir verfasst wurde. Literatur- und Bildquellen sind als solche angegeben.

Hamburg, den 08.05.2021

gez. Nabil Salama

8. Literaturverzeichnis

- https://www.youtube.com/playlist?list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv

- https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi
- „Neural Networks and Deep Learning“ von Michael Nielson (https://static.latexstudio.net/article/2018/0912/neuralnetworksanddeeplearning.pdf)
- https://arxiv.org/pdf/1502.01852.pdf
- https://arxiv.org/pdf/1603.07285.pdf
- https://arxiv.org/pdf/1409.1556.pdf
- https://arxiv.org/pdf/1412.6980.pdf
- https://arxiv.org/pdf/1502.03167.pdf
- https://de.qaz.wiki/wiki/Error_function#Inverse_functions
- http://www.cs.cmu.edu/~bhiksha/courses/deeplearning/Spring.2018/www/slides/lec8.stochastic_gradient.pdf

Alle Links zuletzt am 28.04.2021 abgerufen.

9. Bildverzeichnis

506643544031/Comparison-of-popular-CNN-architectures-The-vertical-axis-shows-top-1-accuracy-on.png

4. Funktionsweise
   - 4.1. https://www.researchgate.net/profile/Hiromu_Yakura/publication/323792694/figure/fig1/AS:615019968475136@1523643595196/Outline-of-the-convolutional-layer.png
   - 4.2. https://miro.medium.com/max/2340/1*Fw-ehcNBR9byHtho-Rxbtw.gif
   - 4.3. https://miro.medium.com/max/2560/1*ciDgQEjViWLnCbmX-EeSrA.gif
   - 4.4. https://m-alcu.github.io/assets/cntk103d_conv2d_final.gif
   - 4.5. https://www.rheinwerk-verlag.de/neuronale-netze-programmieren-mit-python/
   - 4.6. http://nabils.bplaced.net/Bild%204.6.png (selbst erstellt)
   - 4.7. https://arxiv.org/pdf/1409.4842.pdf
   - 4.8. https://www.researchgate.net/profile/Alla_Eddine_Guissous/publication/337336341/figure/fig15/AS:855841334898691@1581059883782/Example-for-the-max-pooling-and-the-average-pooling-with-a-filter-size-of-22-and-a.jpg
   - 4.9. https://miro.medium.com/max/1050/1*MFl0tPjwvc49HirAJZPhEA.png
   - 4.10. https://miro.medium.com/max/1050/1*DmnOhSTIzn04sC0w1d3FPg.png
   - 4.11. https://chercher.tech/images/deep-learning/deep-learning.png
   - 4.12. https://hackernoon.com/hn-images/1*zjzWdMucBfRbkqMzgm2xKg.png (ausschnitt)
5. Lernprozess
   - 5.1. http://nabils.bplaced.net/Bild%205.1.png (selbst erstellt)
   - 5.2. https://blog.paperspace.com/content/images/2019/09/F1-02.large.jpg
   - 5.3. http://nabils.bplaced.net/transposed-convolution/ (selbst erstellt)
   - 5.4. https://miro.medium.com/max/1972/1*uk4KJEtyDuPOipfG4yd-WA.gif
   - 5.5. https://upload.wikimedia.org/wikipedia/commons/9/92/Convolution_arithmetic_-_Padding_strides_transposed.gif
   - 5.6. https://miro.medium.com/max/1840/1*iAK5uMoOlX1gZu-cSh1nZw.png

5.0 und 6.1 bis 10.3 http://nabils.bplaced.net/bilder/ (selbst erstellt)

Alle Links zuletzt am 28.04.2021 abgerufen.

10. Anhang

Im Folgenden sind die mathematischen Herleitungen für die in der Arbeit gegebenen Formeln zu finden.

10.1 Dense Layer

Um mathematisch die Änderungsrate eines Parameters auf die Cost Function zu bestimmen, muss die partielle Ableitung der Cost Function nach dem Parameter berechnet werden. Da es sich dabei um viele ineinander verschachtelte Funktionen handelt, nutzt man die Kettenregel. Zunächst wird jede Funktion einzeln abgeleitet:

$$\frac{\partial C}{\partial f_{akt}} = \frac{\partial C}{\partial a_j} = \frac{\partial}{\partial a_j}\left[-\sum_i y_i \cdot \ln(a_i)\right] = \frac{\partial}{\partial a_j}\left[-y_j \cdot ln(a_j)\right] = -y_j \frac{1}{a_j} = \frac{-y_j}{a_j}$$

$$\frac{\partial f_{akt}}{\partial z_i} = \begin{cases} f_{akt}(z_i) \cdot (1 - f_{akt}(z_i)), & i = j \\ -f_{akt}(z_i) \cdot f_{akt}(z_j), & i \neq j \end{cases}$$

$$\frac{\partial z_i^L}{\partial b_i^L} = \frac{\partial}{\partial b_i^L}\left[b_i^l + \sum_k a_k^{l-1} w_{i,k}^l\right] = 1$$

$$\frac{\partial z_i^L}{\partial w_{j,i}^L} = \frac{\partial}{\partial w_{j,i}^L}\left[b_j^l + \sum_k a_k^{l-1} w_{j,k}^l\right] = a_i^{l-1}$$

Da die Softmax Aktivierungsfunktion eigentlich nicht nur die Zwischensumme z_i^l des j-ten Neurons entgegennimmt, welche im Zähler steht, sondern auch sämtliche anderen Werte des Layers für z_i^l, die im Nenner als Potenz aufsummiert werden, muss beim Ableiten unterschieden werden, nach welchem Neuron abgeleitet werden soll. Daher ergibt sich für die Ableitung der Softmax Funktion Folgendes:

$$\frac{\partial f_{akt}}{\partial z_i} = \frac{\partial}{\partial z_i}\left[\frac{e^{z_j}}{\sum_k e^{z_k}}\right] = \frac{\frac{d}{dz_i}[e^{z_j}] \cdot \sum_k e^{z_k} - \frac{d}{dz_i}[\sum_k e^{z_k}] \cdot e^{z_j}}{(\sum_k e^{z_k})^2}$$

$$i = j: \quad \frac{\partial f_{akt}}{\partial z_i} = \frac{e^{z_i} \cdot \sum_k e^{z_k} - e^{z_i} \cdot e^{z_i}}{(\sum_k e^{z_k})^2} = \frac{e^{z_i}}{\sum_k e^{z_k}} \cdot \left(\frac{\sum_k e^{z_k} - e^{z_i}}{\sum_k e^{z_k}}\right)$$

$$= \frac{e^{z_i}}{\sum_k e^{z_k}} \cdot \left(\frac{\sum_k e^{z_k}}{\sum_k e^{z_k}} - \frac{e^{z_i}}{\sum_k e^{z_k}}\right) = f_{akt} \cdot (1 - f_{akt})$$

$$i \neq j: \quad \frac{\partial f_{akt}}{\partial z_i} = \frac{0 \cdot \sum_k e^{z_k} - \frac{d}{dz_i}[e^{z_i} + \sum_{k \neq i} e^{z_k}] \cdot e^{z_j}}{(\sum_k e^{z_k})^2} = \frac{-e^{z_i} \cdot e^{z_j}}{(\sum_k e^{z_k})^2}$$

$$= -\frac{e^{z_i}}{\sum_k e^{z_k}} \cdot \frac{e^{z_j}}{\sum_k e^{z_k}} = -f_{akt}(z_i) \cdot f_{akt}(z_j)$$

Mithilfe der Ableitungen ergeben sich für die Biases, Gewichte und Fehler des letzten Layers durch die Kettenregel folgende Gleichungen:

$$\delta_i^L = \frac{\partial C}{\partial z_i^L} = \frac{\partial C}{\partial f_{akt}} \cdot \frac{\partial f_{akt}}{\partial z_i^L} = f'_{akt}(z_i^L) \cdot C'(a_i^L)$$

$$\frac{\partial C}{\partial b_i^L} = \frac{\partial C}{\partial f_{akt}} \cdot \frac{\partial f_{akt}}{\partial z_i^L} \cdot \frac{\partial z_i^L}{\partial b_i^L} = \delta_i^L \cdot 1 = \delta_i^L$$

$$\frac{\partial C}{\partial w_{j,i}^L} = \frac{\partial C}{\partial f_{akt}} \cdot \frac{\partial f_{akt}}{\partial z_i^L} \cdot \frac{\partial z_i^L}{\partial w_{j,i}^L} = \delta_j^L \cdot a_i^{l-1}$$

Die Gleichungen für Biases und Gewichte gelten im Gegensatz zu dem Fehler auch in den anderen Layern. Letzterer ergibt sich durch folgende Betrachtung:

$$\delta_i^l = \frac{\partial C}{\partial z_i^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}} \cdot \frac{\partial z_k^{l+1}}{\partial z_i^l} = \sum_k \delta_k^{l+1} \cdot \frac{\partial}{\partial z_i^l}\left[b_k^{l+1} + \sum_j a_j^l w_{k,j}^{l+1}\right]$$

$$= \sum_k \delta_k^{l+1} \cdot \frac{\partial}{\partial z_i^l}\left[b_k^{l+1} + \sum_j f_{akt}(z_j^l) w_{k,j}^{l+1}\right] = \sum_k \delta_k^{l+1} \cdot f'_{akt}(z_i^l) w_{k,i}^{l+1}$$

Da in den Fully Connected Layer ein Neuron mit jedem Neuron des darauffolgenden Layers verbunden ist, beeinflusst die Zwischensumme sämtliche Neuronen des Layers $l+1$, was bedeutet, dass beim Berechnen des Fehlers über sämtliche Neuronen diesen Layers aufsummiert werden muss.

10.2 Convolutional Layer

Unter mathematischer Betrachtung ergibt sich die Ableitung der Loss Function nach den Gewichten, Biases und Aktivierungswerten des Inputs in den Convolutional Layern durch die Kettenregel:

$$\frac{\partial C}{\partial a^{l-1}} = \frac{\partial C}{\partial a^l} \cdot \frac{\partial a^l}{\partial z^l} \cdot \frac{\partial z^l}{\partial a^{l-1}} \qquad \frac{\partial C}{\partial W} = \frac{\partial C}{\partial a^l} \cdot \frac{\partial a^l}{\partial z^l} \cdot \frac{\partial z^l}{\partial W} \qquad \frac{\partial C}{\partial b^l} = \frac{\partial C}{\partial a^l} \cdot \frac{\partial a^l}{\partial z^l} \cdot \frac{\partial z^l}{\partial b^l}$$

Da $\partial C/\partial a^l$ vom Layer $l+1$ geliefert wird, lässt sich der Fehler einfach berechnen:

$$\delta^l = \frac{\partial C}{\partial a^l} \cdot \frac{\partial a^l}{\partial z^l} = \frac{\partial C}{\partial a^l} \cdot \frac{\partial}{\partial z^l}[f_{akt}(z^l)] = \frac{\partial C}{\partial a^l} \cdot f'_{akt}(z^l)$$

Der Bias ist dann gegeben durch:

$$\frac{\partial C}{\partial b_i^l} = \delta_i^l \cdot \frac{\partial z_i^l}{\partial b_i^l} = \delta_i^l \cdot \frac{\partial}{\partial b_i^l}\left[b^l + \sum_{n=1}^{k^2 \cdot m_3^{l-1}} w_n^l \cdot a_N^{l-1}\right] = \delta_i^l \cdot 1$$

Der Einfachheit halber wurde unter eine Summe geschrieben. Für die Indizes gilt:

$$w_n^l = w_{i,j,k}^l \qquad i = \left\lceil \frac{n}{km_3} \right\rceil \qquad j = \left\lceil \frac{n}{k} \right\rceil \qquad k = n - \left\lfloor \frac{n}{k} \right\rfloor \cdot k$$

$$a_N^{l-1} = a_{I,J,k}^{l-1} \qquad I = i + n_1 s \qquad J = j + n_2 s$$

Um den Einfluss der Gewichte $\partial z^l / \partial W$ in der Convolution Operation bestimmen zu können, ist zunächst die Berechnung der Änderung eines Gewichts für ein einziges Neuron t im Layer l zu untersuchen.

$$\frac{\partial C}{\partial w_{p,q,r}^l} = \delta_t^l \cdot \frac{\partial z_t^l}{\partial w_{p,q,r}^l} = \delta_t^l \cdot \frac{\partial}{\partial w_{p,q,r}^l}\left[b^l + \sum_{n=1}^{k^2 \cdot m_3^{l-1}} w_n^l \cdot a_N^{l-1}\right] = \delta_t^l \cdot a_K^{l-1}$$

Die Ableitung nach dem Gewicht in einem Schritt der Convolution Operation ergibt also den Aktivierungswert eines Neurons im Vorherigen Layer, dessen Index K von der Position des Gewichts im Kernel sowie von der jeweiligen Schrittzahl abhängt. Für diesen gilt:

$$a_K^{l-1} = a_{P,Q,r}^{l-1} \qquad P = p + n_1 s \qquad Q = q + n_2 s$$

Bei der Convolution Operation muss berücksichtigt werden, dass jedes Gewichte in der Berechnung jeden Neurons einer Activation Map mit einfließt, sodass bei der Ableitung nach dem Gewicht auch über jedes Neuron summiert werden muss.

$$\frac{\partial C}{\partial w_j^l} = \sum_{i=1}^{m_1 \cdot m_2} \frac{\partial C}{\partial z_i^l} \cdot \frac{\partial z_i^l}{\partial w_j^l}$$

Wenn man nun die Summe sowie die Indizes ausschreibt und $\partial z_i^l / \partial w_j^l$ durch den zuvor berechneten Ausdruck und $\partial C / \partial z_i^l$ durch den Fehler substituiert, ergibt sich:

$$\frac{\partial C}{\partial w_{j,k,l}^l} = \sum_{n_2=1}^{m_2^l} \sum_{n_1=1}^{m_1^l} \delta_{n_1,n_2}^l \cdot a_{J,K,l}^{l-1} \qquad J = j + n_1 s, \qquad K = k + n_2 s$$

Dies ist eine Convolution Operation mit den Fehlern als Kernel und Aktivierungswerten des vorherigen Layers als Feature Map.

Nun ist noch der Einfluss des Inputs $\partial C^l/\partial a^{l-1}$, also der Aktivierungswerte des vorherigen Layers, auf die Cost Function zu berechnen, damit diese Werte dem vorherigen Layer übergeben werden können.
Als erstes ist wieder zu untersuchen, wie sich ein Pixel im Input auf ein einziges Neuron t im Layer l auswirkt:

$$\frac{\partial C}{\partial a^l_{p,q,r}} = \delta^l_t \cdot \frac{\partial z^l_t}{\partial a^l_{p,q,r}} = \delta^l_t \cdot \frac{\partial}{\partial a^l_{p,q,r}} \left[b^l + \sum_{n=1}^{k^2 \cdot m_3^{l-1}} w^l_n \cdot a^{l-1}_N \right] = \delta^l_t \cdot w^l_{n'}$$

Ein Input-Wert wirkt sich auf jede Feature Map f aus, weil bei der Berechnung jeder einzelnen Map der zugehörige Filter jeweils über den Input geschoben wird. Außerdem kann sich ein Input auf mehrere Elemente der Output-Feature Map auswirken. Für die Ableitung eines Input-Wertes muss daher sowohl über die Feature Maps als auch über die Neuronen, die mithilfe des zu untersuchenden Inputs berechnet werden, summiert werden:

$$\frac{\partial C}{\partial a^{l-1}_{i,j,p}} = \sum_f \sum_{n_2,n_1} \delta^{l,f}_{n_2,n_1} \cdot w^{l,f}_{I,J,p}$$

Als nächstes müssen die Indizes sowie die Summengrenzen untersucht werden. Für die erste Summe muss wie zuvor festgestellt über jede Feature Map iteriert werden, es gilt daher $f \epsilon [1; m^l_3]$.
Die folgenden Betrachtung wird exemplarisch für einen Stride von $s = 1$ durchgeführt. Die variablen n_2 und n_1 geben den Schrittindex in vertikaler bzw. horizontaler Richtung an, bei welchem a^{l-1} vom Filter verdeckt wird, also an welchem Schritt der Wert in die Berechnung einer Zwischensumme einfließt. Entsprechend sind die Intervalle von den Indizes des Input-Neurons abhängig. Die horizontale Untergrenze m_i wäre daher für n_1 das erste Mal, bei dem a^{l-1} vom Filter verdeckt wird und die Obergrenze $m_i + v_i$ das letzte Mal, bei dem dies geschieht, wobei v_i angibt, in wie vielen Schritten der Filter das Neuron verdeckt. Gleiches gilt für n_2 in vertikaler Richtung mit m_j und v_j. Für das Intervall von n_2 und n_1 ergibt sich:

$$n_1 \in [m_i;\ m_i + v_i - 1] \qquad n_2 \in [m_j;\ m_j + v_j - 1]$$

Das Subtrahieren von 1 an der Obergrenze dient lediglich der Indexkorrektur.
Im ersten Schritt einer Convolution Operation sind die ersten k Neuronen vom Filter verdeckt, wobei k die Kernelgröße ist. Alle anderen Neuronen an der Stelle

i, für die $i > k$ gilt, ergibt sich bei näherer Betrachtung, dass sie zum ersten Mal im Schritt $i - k + 1$ in die Berechnung mit einfließen. Diese Beobachtungen lassen sich ebenfalls auf die vertikale Dimension anwenden und wie folgt formulieren:

$$m_i = \begin{cases} 1, & i \leq k \\ i - k + 1, & i > k \end{cases}$$

Neuronen, dessen Entfernung zu den Rändern kleiner als die Filtergröße ist werden seltener verdeckt, als die anderen Neuronen. Zum Beispiel wird das erste sowie letzte Neuron einmal verdeckt, das zweite sowie vorletzte zweimal usw. Alle anderen Neuronen werden so oft verdeckt, wie der Filter groß ist. Es ergibt sich:

$$v_i = \begin{cases} k, & k \leq i \leq m_1^l \\ i, & i < k \\ m_1^l - i, & i > m_1^l - k \end{cases}$$

Gleiches gilt in Vertikaler Richtung mit v_j und m_2^l. Mithilfe dieser Erkenntnisse kann die Summe, die die Ableitung von a^{l-1} angibt präziser formuliert werden:

$$\frac{\partial C}{\partial a_{i,j,p}^{l-1}} = \sum_{f=1}^{m_3^l} \sum_{n_2=m_j}^{m_j+v_j-1} \sum_{n_1=m_i}^{m_i+v_i-1} \delta_{n_2,n_1}^{l,f} \cdot w_{I,J,p}^{l,f}$$

Der Index, an dem sich das erste Element $(1{,}1{,}k)$ des Kernels zu einem Schritt befindet ist gegeben durch $n_{1,2} \cdot s$. Es müssen die Indizes I, J, p des Gewichts bestimmt werden, das mit a^{l-1} im Forward Pass multipliziert wird. Diese sind für den sowohl horizontalen als auch vertikalen Fall durch die Entfernung zwischen dem ersten Filterelement und dem Index des Neurons a^{l-1} gegeben:

$$I = i - n_1 s + 1 \qquad J = j - n_2 s + 1$$

Der Index des Gewichts im Kernel ist für die dritte Dimension gleich dem Index des Neurons im vorherigen Layer, denn dies kann nur mit Gewichten mit gleichem Tiefenindex multipliziert werden, weil in diese Richtung keine Verschiebung des Kernels stattfindet.

Die Form der Summierung, die $\partial C / \partial a_{i,j,p}^{l-1}$ darstellt erinnert stark an eine Convolution Operation. Allerdings sind die Grenzen der Iteratoren n_1 und n_2 abhängig von der Position des Neurons und die Gewichtindizes I sowie J berechnen sich leicht abweichend gegenüber einer echten Convolution Operation. Die Summierung ist in Bild 5.3 für eine Feature Map f graphisch

dargestellt. Die grünen Pixel zeigen die Fehler der Neuronen dieser Map, während die blauen Pixel die Neuronen sind, die mit dem Gewicht mit dem im Pixel angegebenem Index multipliziert werden. Die Grafik ähnelt einem ausschnitt einer Convolution Operation, jedoch ist der Filter um 180° gedreht, denn das letzte Elemente $(3,3)$ des Kernels befindet sich an erster Stelle und das erste Element $(1,1)$ an letzter Stelle. Wenn man die neu entstandene grüne Feature Map um ein Zero-Padding der Breit $k-1$ erweitert und als Filter den eigentlichen, zweimal rotierten Kernel verwendet, entsteht eine vollwertige Convolution Operation (Bild 5.4). Allerding ist zu beachten, dass dies lediglich für einen Stride von $s=1$ der Fall ist. Für einen größeren Stride s wirkt sich jedes Neuron a^{l-1} auf den Output im Forward Pass seltener durch eine Verdeckung aus, weil die Schritte größer sind, sodass zwischen den Elementen der Feature Map im Backward Pass ein Abstand von s hinzugefügt werden muss. Dies ist beispielhaft für $s=2$ in Bild 5.5 dargestellt.

10.3 Batch Normalization

Um den Backward Pass für den Batch Normalization Layer nachvollziehen zu können, ist es wichtig, zu verstehen welche Größen sich auf welche auswirken (Bild 10.1), da dies stets beim Ableiten der Cost Function nach einer Variable berücksichtigt werden muss. Ziel ist es, den Einfluss der zu erlernenden Parameter β und γ sowie dem Inputs der BN z auf die Loss Function in Abhängigkeit vom Gradient des Outputs $\partial C/\partial a$ eines Neurons zu berechnen.
Für den Output $\hat{z}$ der BN gilt:

$$a = f_a(\hat{z})\ , \quad \frac{\partial C}{\partial \hat{z}} = \frac{\partial C}{\partial a} \cdot \frac{\partial a}{\partial \hat{z}} = \frac{\partial C}{\partial a} \cdot f_a'(\hat{z})$$

Es ergibt sich dann für den Gradient der zu erlernenden Werte sowie für den Wert des Zwischenschritts u in der BN:

$$\frac{\partial C}{\partial \beta} = \frac{\partial C}{\partial \hat{z}} \cdot \frac{\partial \hat{z}}{\partial \beta} = \frac{\partial C}{\partial \hat{z}} \cdot \frac{\partial}{\partial \beta}[\gamma u + \beta] = \frac{\partial C}{\partial a} f_a'(\hat{z}) \cdot 1$$

$$\frac{\partial C}{\partial \gamma} = \frac{\partial C}{\partial \hat{z}} \cdot \frac{\partial \hat{z}}{\partial \gamma} = \frac{\partial C}{\partial \hat{z}} \cdot \frac{\partial}{\partial \gamma}[\gamma u + \beta] = \frac{\partial C}{\partial a} f_a'(\hat{z}) \cdot u$$

$$\frac{\partial C}{\partial u} = \frac{\partial C}{\partial \hat{z}} \cdot \frac{\partial \hat{z}}{\partial u} = \frac{\partial C}{\partial \hat{z}} \cdot \frac{\partial}{\partial u}[\gamma u + \beta] = \frac{\partial C}{\partial a} f_a'(\hat{z}) \cdot \gamma$$

Mit letzterem kann der Gradient der Varianz und des Erwartungswerts berechnet werden. Beim Trainieren mit Mini-Batches hält jedes Neuron so viel Werte wie es

Batches gibt. Dies bedeutet, dass das Quadrat der Standardabweichung sich auf jeden dieser Werte auswirkt, weshalb bei der Ableitung über diese summiert werden muss:

$$\frac{\partial C}{\partial \sigma^2} = \sum_{i=0}^{B} \frac{\partial C}{\partial u_i} \cdot \frac{\partial u_i}{\partial \sigma^2} = \sum_{i=0}^{B} \frac{\partial C}{\partial u_i} \cdot \frac{\partial}{\partial \sigma^2} \left[\frac{z_i - \mu}{\sqrt{\sigma^2 + \delta}} \right]$$

$$= \sum_{i=0}^{B} \frac{\partial C}{\partial u_i} \cdot \frac{(-1)(z_i - \mu)}{2\sqrt{(\sigma^2 + \delta)^3}} \cdot \frac{\partial}{\partial \sigma^2} [\sigma^2 + \delta] = \sum_{i=0}^{B} \frac{\partial C}{\partial u_i} \cdot \frac{(\mu - z_i)}{2\sqrt{(\sigma^2 + \delta)^3}} \cdot 1$$

Der Erwartungswert wirkt sich ebenfalls auf die gleichen Werte wie zuvor und zudem noch auf die Varianz aus.

$$\frac{\partial C}{\partial \mu} = \frac{\partial C}{\partial \sigma^2} \cdot \frac{\partial \sigma^2}{\partial \mu} + \sum_{i=0}^{B} \frac{\partial C}{\partial u_i} \cdot \frac{\partial u_i}{\partial \mu} = \frac{\partial C}{\partial \sigma^2} \cdot \frac{\partial}{\partial \mu} \left[\frac{1}{B} \sum_{i=1}^{B} (z_i - \mu)^2 \right] + \sum_{i=0}^{B} \frac{\partial C}{\partial u_i} \cdot \frac{\partial}{\partial \mu} \left[\frac{z_i - \mu}{\sqrt{\sigma^2 + \delta}} \right]$$

$$= \frac{\partial C}{\partial \sigma^2} \cdot \frac{1}{B} \sum_{i=1}^{B} 2(z_i - \mu)(-1) + \sum_{i=0}^{B} \frac{\partial C}{\partial u_i} \cdot \frac{-1}{\sqrt{\sigma^2 + \delta}}$$

$$= \frac{\partial C}{\partial \sigma^2} \cdot \frac{2}{B} \sum_{i=1}^{B} (\mu - z_i) - \sum_{i=0}^{B} \frac{\partial C}{\partial u_i} \cdot (\sigma^2 + \delta)^{-0,5}$$

Der Input z wirkt sich auf drei Größen aus: Erwartungswert, Varianz und Zwischenergebnis u. Folglich gilt:

$$\frac{\partial C}{\partial z_{i'}} = \frac{\partial C}{\partial \mu} \cdot \frac{\partial \mu}{\partial z_{i'}} + \frac{\partial C}{\partial \sigma^2} \cdot \frac{\partial \sigma^2}{\partial z_{i'}} + \frac{\partial C}{\partial u} \cdot \frac{\partial u}{\partial z_{i'}}$$

$$= \frac{\partial C}{\partial \mu} \frac{\partial}{\partial z_{i'}} \left[\frac{1}{B} \sum_{i=0}^{B} z_i \right] + \frac{\partial C}{\partial \sigma^2} \frac{\partial}{\partial z_{i'}} \left[\frac{1}{B} \sum_{i=1}^{B} (z_i - \mu)^2 \right] + \frac{\partial C}{\partial u} \frac{\partial}{\partial z_{i'}} \left[\frac{z_i - \mu}{\sqrt{\sigma^2 + \delta}} \right]$$

$$= \frac{\partial C}{\partial \mu} \frac{1}{B} \cdot 1 + \frac{\partial C}{\partial \sigma^2} \frac{1}{B} \cdot 2(z_{i'} - \mu) + \frac{\partial C}{\partial u} \frac{1}{\sqrt{\sigma^2 + \delta}}$$

$$= \frac{1}{B} \left(\frac{\partial C}{\partial \mu} + 2 \frac{\partial C}{\partial \sigma^2} (z_{i'} - \mu) \right) + \frac{\partial C}{\partial u} \sqrt[-2]{\sigma^2 + \delta}$$

Mit $\partial C / \partial z$ kann dann wie zuvor beschrieben weitergerechnet werden.

10.4 He Initialisierung

Unter mathematischer Betrachtung ergibt sich die He Initialisierung durch die Untersuchung der Varianz, um so die Standardabweichung der Gewichte zu berechnen.

Im Dense Layer oder auch im Convolutional Layer, wenn die Elemente der drei Dimensionen eines Kernels eindimensional hintereinander aufgeschrieben werden, gilt:

$$z_j^l = b_j^l + \sum_{i=1}^{n^{l-1}} a_i^{l-1} \cdot w_{j,i}^l$$

Folglich gilt für die Varianz eines Neurons

$$Var\left(z_j^l\right) = Var\left(b_j^l + \sum_{i=1}^{n^{l-1}} a_i^{l-1} \cdot w_{j,i}^l\right)$$

Da die Gewichte $w_{j,i}^l$ für alle i in einem Layer l mithilfe gleicher Parameter einer Verteilung initialisiert werden sollen, haben diese die Gleiche Verteilungsfunktion. Zudem sind die Werte unabhängig voneinander, was bedeutet, dass die Varianz der Summe gleich der Summe der Varianz ist. Gleiches gilt für den Aktivierungswert a_i^{l-1}. Der Bias b_j^l ist zum einen eine Konstante und wird zum anderen mit Null initialisiert, sodass er wegfällt.

$$Var\left(b_j^l + \sum_{i=1}^{n^{l-1}} a_i^{l-1} \cdot w_{j,i}^l\right) = \sum_{i=1}^{n^{l-1}} Var\left(a_i^{l-1} \cdot w_{j,i}^l\right)$$

Weil die Gleiche Verteilungsfunktion verwendet wird, ist auch $Var\left(a_i^{l-1} \cdot w_{j,i}^l\right)$ für jedes i gleich:

$$\sum_{i=1}^{n^{l-1}} Var\left(a_i^{l-1} \cdot w_{j,i}^l\right) = n^{l-1} \cdot Var\left(a_i^{l-1} \cdot w_{j,i}^l\right)$$

Für das Produkt einer Varianz gilt:

$$Var(X \cdot Y) = Var(X)Var(Y) + \big(E(X)\big)^2 Var(Y) + \big(E(Y)\big)^2 Var(X)$$

Dies angewendet und unter der Berücksichtigung, dass die Gewichte mit einem Erwartungswert von Null zu initialisieren sind, ergibt sich:

$$\begin{aligned} n^{l-1} \cdot Var\left(a_i^{l-1} \cdot w_{j,i}^l\right) &= n^{l-1} \cdot \left[Var\left(a_i^{l-1}\right)Var\left(w_{j,i}^l\right) + \left(E\left(a_i^{l-1}\right)\right)^2 Var\left(w_{j,i}^l\right) + \left(E\left(w_{j,i}^l\right)\right)^2 Var\left(a_i^{l-1}\right)\right] \\ &= n^{l-1} \cdot \left[Var\left(a_i^{l-1}\right)Var\left(w_{j,i}^l\right) + \left(E\left(a_i^{l-1}\right)\right)^2 Var\left(w_{j,i}^l\right) + (0)^2 Var\left(a_i^{l-1}\right)\right] \\ &= n^{l-1} \cdot \left[Var\left(a_i^{l-1}\right)Var\left(w_{j,i}^l\right) + \left(E\left(a_i^{l-1}\right)\right)^2 Var\left(w_{j,i}^l\right)\right] \\ &= n^{l-1} \cdot Var\left(w_{j,i}^l\right) \cdot \left[Var\left(a_i^{l-1}\right) + \left(E\left(a_i^{l-1}\right)\right)^2\right] \end{aligned}$$

Unter Anwendung des Verschiebungssatzes des Erwartungswerts mit $a = 0$ folgt:

$$E((X-a)^2) = Var(X) + (E(X)-a)^2$$

$$Var\left(z_j^l\right) = n^{l-1}Var\left(w_{j,i}^l\right)\left[Var\left(a_i^{l-1}\right) + \left(E\left(a_i^{l-1}\right)\right)^2\right] = n^{l-1}Var\left(w_{j,i}^l\right)E\left((a_i^{l-1})^2\right)$$

Um $E((a^{l-1})^2)$ zu berechnen, wird berücksichtigt, dass a^{l-1} stetig verteilt ist und der Aktivierungswert der Wert der Aktivierungsfunktion für die Zwischensumme z_i^{l-1} ist. Die He Initialisierung ist für die ReLU Funktion ausgelegt:

$$f_{akt}(z) = max(0,z)$$

$$E((a^{l-1})^2) = \int_{-\infty}^{\infty} (a^{l-1})^2 \cdot P(a^{l-1})\, da^{l-1} = \int_{-\infty}^{\infty} (max(0,z^{l-1}))^2 \cdot P(z^{l-1}) dz^{l-1}$$

Weil die ReLU Funktion für $z < 0$ gleich Null ist, gilt:

$$E((a^{l-1})^2) = \int_{-\infty}^{\infty} (max(0,z^{l-1}))^2 \cdot P(z^{l-1})\, dz^{l-1} = \int_{0}^{\infty} (z^{l-1})^2 \cdot P(z^{l-1})\, dz^{l-1}$$

Unter Beachtung der Definition des Erwartungswerts für stetig verteilte Zufallszahlen wie z^{l-1} folgt:

$$E\left((a^{l-1})^2\right) = \int_{0}^{\infty} \left(z^{l-1}\right)^2 \cdot P\left(z^{l-1}\right) dz^{l-1} = \frac{1}{2}\int_{-\infty}^{\infty} \left(z^{l-1}\right)^2 \cdot P\left(z^{l-1}\right) dz^{l-1} = \frac{1}{2}E\left(\left(z^{l-1}\right)^2\right)$$

Die Zwischensumme z^{l-1} ergibt sich aus den Gewichten, die mit einem Erwartungswert von Null initialisiert werden und einem Bias, der Null ist, sodass auch der Erwartungswert von z^{l-1} Null ist. Für um Null verteilte Zufallsvariablen gilt:

$$E(X^2) = \int_{-\infty}^{\infty} x^2 P(x)\, dx = \int_{-\infty}^{\infty} (x-\mu)^2 P(x)\, dx = Var(X)$$

Daraus folgt:

$$E((a^{l-1})^2) = \frac{1}{2}E((z^{l-1})^2) = \frac{1}{2}Var(z^{l-1})$$

Es ergibt sich unter Einsetzung dessen in das frühere Ergebnis:

$$Var\left(z_j^l\right) = n^{l-1}Var\left(w_{j,i}^l\right)E\left((a_i^{l-1})^2\right) = n^{l-1}Var\left(w_{j,i}^l\right)\cdot\frac{1}{2}\cdot Var\left(z_i^{l-1}\right)$$

Für die Varianz der Zwischensumme des letzten Layers L folgt daraus:

$$Var\left(z_j^L\right) = \frac{n^{L-1}}{2}Var\left(w_{j,i}^L\right)Var(z_i^{L-1}) = \frac{n^{L-1}}{2}Var\left(w_{j,i}^L\right)\left[\frac{n^{L-2}}{2}Var\left(w_{j,i}^{L-1}\right)Var(z_i^{L-2})\right]$$

Der Term $Var(z_i^{L-a})$ kann bis zum ersten Layer immer weiter durch $Var(z_i^{L-a-1})$ substituiert werden, sodass sich folgendes Produkt ergibt:

$$Var(z_i^L) = Var(z_i^1) \cdot \prod_{l=2}^{L} \frac{n^{l-1}}{2} Var\left(w_{j,k}^l\right)$$

Um dem Exploding und Vanishing Gradient Problem entgegenzuwirken, sollte die Varianz der Zwischensummen immer gleich bleiben. Damit $Var(z_i^L) = Var(z_i^1)$ ist, muss das Produkt und somit auch die Faktoren des Produkts Eins sein:

$$\frac{n^{l-1}}{2} Var\left(w_{j,k}^l\right) = 1 \Leftrightarrow Var\left(w_{j,k}^l\right) = \frac{2}{n^{l-1}} \quad \Rightarrow \quad \sigma = \sqrt{\frac{2}{n^{l-1}}}$$

Die Standardabweichung mit der die Gewichte zu initialisieren sind ergibt sich aus der Quadratwurzel der Varianz.

10.5 Mehrdimensionale Dense Layer

Die Formeln für den Backpropagation Algorithmus solcher Layer ergeben sich unter folgender Betrachtung.

Zunächst ist zu berücksichtigen, dass die Parameter der Gewichtsfunktion sämtliche Gewichte, die als Input dienen, beeinflussen, weshalb beim Ableiten über diese summiert werden muss. Für α und β ergibt sich für den Einfluss auf ein einziges Gewicht:

$$\frac{\partial C}{\partial \beta} = \frac{\partial C}{\partial w} \cdot \frac{\partial w}{\partial \beta} = \frac{\partial C}{\partial w} \cdot \frac{\partial}{\partial \beta}\left[\beta - \alpha\sqrt{d+\delta}\right] = \frac{\partial C}{\partial w} \cdot 1$$

$$\frac{\partial C}{\partial \alpha} = \frac{\partial C}{\partial w} \cdot \frac{\partial w}{\partial \alpha} = \frac{\partial C}{\partial w} \cdot \frac{\partial}{\partial \alpha}\left[\beta - \alpha\sqrt{d+\delta}\right] = \frac{\partial C}{\partial w} \cdot \left(-\sqrt{d+\delta}\right)$$

Unter Berücksichtigung aller Gewichte ergibt sich dann:

$$\frac{\partial C}{\partial \beta_j^l} = \sum_{i=1}^{n^{l-1}} \frac{dC}{dw_{j,i}^l} \cdot \frac{dw_{j,i}^l}{d\beta_j^l} = \sum_{i=1}^{n^{l-1}} \frac{dC}{dw_{j,i}^l} \cdot 1$$

$$\frac{\partial C}{\partial \alpha_j^l} = \sum_{i=1}^{n^{l-1}} \frac{dC}{dw_{j,i}^l} \cdot \frac{dw_{j,i}^l}{d\alpha_j^l} = -\sum_{i=1}^{n^{l-1}} \frac{dC}{dw_{j,i}^l} \cdot \sqrt{d(x_i^{l-1}, x_j^l) + \delta}$$

Für die Ableitung der Gewichtsfunktion nach der Entfernung ergibt sich:

$$\frac{\partial w}{\partial d} = \frac{\partial}{\partial d}\left[\beta - \alpha\sqrt{d+\delta}\right] = \frac{-\alpha}{2\sqrt{d+\delta}}$$

Die Ableitung der Entfernung nach einem Ortswert der Input Neuron ist für die Beeinflussung eines einzigen Gewichts gegeben durch:

$$\frac{\partial d}{\partial x_{k'}^{l}} = \frac{\partial}{\partial x_{k'}^{l}}\left[\sqrt{\sum_{k}\left(x_k^{l-1} - x_k^{l}\right)^2}\right] = \frac{1}{2\sqrt{\sum_k\left(x_k^{l-1} - x_k^{l}\right)^2}} \cdot \frac{d}{dx_{k'}^{l}}\left[\sum_{k}\left(x_k^{l-1} - x_k^{l}\right)^2\right]$$

$$= \frac{1}{2d} \cdot \frac{d}{dx_{k'}^{l}}\left[\left(x_{k'}^{l-1} - x_{k'}^{l}\right)^2\right] = \frac{1}{2d} \cdot 2\left(x_{k'}^{l-1} - x_{k'}^{l}\right) \cdot (-1)$$

$$= \frac{1}{d} \cdot \left(x_{k'}^{l} - x_{k'}^{l-1}\right)$$

Die Koordinate x^l eines Neurons j im letzten Layer L beeinflusst jedoch sämtliche Input-Gewichte, die alle zu berücksichtigen sind:

$$\frac{\partial C}{\partial x_{k'}^{L}} = \sum_{i=1}^{n^{L-1}} \frac{\partial C}{\partial w_{j,i}^{L}} \cdot \frac{\partial w_{j,i}^{L}}{\partial d(x_i^{L-1}, x_j^{L})} \cdot \frac{\partial d(x_i^{L-1}, x_j^{L})}{\partial x_{k'}^{L}}$$

$$= \sum_{i=1}^{n^{L-1}} \frac{\partial C}{\partial w_{j,i}^{L}} \cdot \frac{-\alpha_j^{L}}{2\sqrt{d(x_i^{L-1}, x_j^{L}) + \delta}} \cdot \frac{x_{k'}^{L} - x_{k'}^{L-1}}{d(x_i^{L-1}, x_j^{L})}$$

Im ersten Layer hingegen beeinflusst die Position eines Neurons i sämtliche Output-Gewichte:

$$\frac{\partial C}{\partial x_{k'}^{1}} = \sum_{j=1}^{n^{2}} \frac{\partial C}{\partial w_{j,i}^{2}} \cdot \frac{\partial w_{j,i}^{2}}{\partial d(x_i^{1}, x_j^{2})} \cdot \frac{\partial d(x_i^{1}, x_j^{2})}{\partial x_{k'}^{1}} = \sum_{j=1}^{n^{2}} \frac{\partial C}{\partial w_{j,i}^{2}} \cdot \frac{-\alpha_j^{2}}{2\sqrt{d(x_i^{1}, x_j^{2}) + \delta}} \cdot \frac{x_{k'}^{1} - x_{k'}^{2}}{d(x_i^{1}, x_j^{2})}$$

Die Position der Neuronen j aller anderen Layer l, für die $1 < l < L$ gilt, beeinflussen sowohl die Input-Gewichte, als auch die Output-Gewichte, weshalb zur Berechnung von $\partial C/\partial x_{k'}^{l}$ die Berechnungen für $\partial C/\partial x_{k'}^{1}$ sowie $\partial C/\partial x_{k'}^{L}$ kombiniert werden müssen:

$$\frac{\partial C}{\partial x_{k'}^{l}} = \left[\sum_{i=1}^{n^{l-1}} \frac{\partial C}{\partial w_{j,i}^{l}} \cdot \frac{-\alpha_j^{l}}{2\sqrt{d(x_i^{l-1}, x_j^{l}) + \delta}} \cdot \frac{x_{k'}^{l} - x_{k'}^{l-1}}{d(x_i^{l-1}, x_j^{l})}\right]$$

$$+ \sum_{i=1}^{n^{l+1}} \frac{\partial C}{\partial w_{i,j}^{l+1}} \cdot \frac{-\alpha_i^{l+1}}{2\sqrt{d(x_j^{l}, x_i^{l+1}) + \delta}} \cdot \frac{x_{k'}^{l} - x_{k'}^{l+1}}{d(x_j^{l}, x_i^{l+1})}$$

Unter Verwendung der linearen Gewichtsfunkton vereinfachen sich die Formeln für den Backpropagation Algorithmus wie Folgt.

Zunächst gilt für die partiellen Ableitungen der Funktion:

$$\frac{\partial w}{\partial d} = \frac{\partial}{\partial d}[\alpha d + \beta] = \alpha\,, \quad \frac{\partial w}{\partial \beta} = \frac{\partial}{\partial \beta}[\alpha d + \beta] = 1\,, \quad \frac{\partial w}{\partial \alpha} = \frac{\partial}{\partial \alpha}[\alpha d + \beta] = d$$

Bei der gleichen Vorgehensweise wie zuvor ergibt sich für die Ableitungen der zu aktualisierenden Parameter:

$$\frac{\partial C}{\partial \beta_j^l} = \sum_{i=1}^{n^{l-1}} \frac{dC}{dw_{j,i}^l} \cdot \frac{dw_{j,i}^l}{d\beta_j^l} = \sum_{i=1}^{n^{l-1}} \frac{dC}{dw_{j,i}^l} \cdot 1$$

$$\frac{\partial C}{\partial \alpha_j^l} = \sum_{i=1}^{n^{l-1}} \frac{dC}{dw_{j,i}^l} \cdot \frac{dw_{j,i}^l}{d\alpha_j^l} = \sum_{i=1}^{n^{l-1}} \frac{dC}{dw_{j,i}^l} \cdot d(x_i^{l-1}, x_j^l)$$

$$\frac{\partial C}{\partial x_{k'}^l} = \left[\sum_{i=1}^{n^{l-1}} \frac{\partial C}{\partial w_{j,i}^l} \cdot \alpha_j^l \cdot \frac{x_{k'}^l - x_{k'}^{l-1}}{d(x_i^{l-1}, x_j^l)}\right] + \sum_{i=1}^{n^{l+1}} \frac{\partial C}{\partial w_{i,j}^{l+1}} \cdot \alpha_j^{l+1} \cdot \frac{x_{k'}^l - x_{k'}^{l+1}}{d(x_j^l, x_i^{l+1})}$$

$$\frac{\partial C}{\partial x_{k'}^L} = \sum_{i=1}^{n^{L-1}} \frac{\partial C}{\partial w_{j,i}^L} \cdot \alpha_j^L \cdot \frac{x_{k'}^L - x_{k'}^{L-1}}{d(x_i^{L-1}, x_j^L)}$$

$$\frac{\partial C}{\partial x_{k'}^1} = \sum_{j=1}^{n^2} \frac{\partial C}{\partial w_{j,i}^2} \cdot \alpha_j^2 \cdot \frac{x_{k'}^1 - x_{k'}^2}{d(x_i^1, x_j^2)}$$

Die Formeln für die Initialisierung bei der nicht-linearen Gewichtsfunktion ergeben sich wie Folgt:

Zunächst gilt es die Verteilungsfunktion $F_D(x)$ zu bestimmen, mit der die Entfernungen zu wählen sind, die sich aus einer Transformation der Verteilungsfunktion $F_W(x)$ der Gewichte mit der Umkehrfunktion $w^{-1}(d)$ zur Berechnung eines Gewichts ergibt:

$$w(d) = \beta - \alpha\sqrt{d + \delta} \Leftrightarrow \frac{w - \beta}{-\alpha} = \sqrt{d + \delta} \Rightarrow d = d(w) \coloneqq \left(\frac{w - \beta}{\alpha}\right)^2 - \delta = w^{-1}(d)$$

Die Verteilungsfunktion der Entfernung lässt sich wie folgt ausdrücken. Die Zufallsvariable für die Distanz ist gleich der der Entfernungsfunktion: $D = d(W)$

$$F_D(d) = P(D \leq d) = P(d(W) \leq d) = P\left(d^{-1}\big(d(W)\big) \leq d^{-1}(d)\right) = P\big(W \leq w(d)\big)$$
$$= F_W\big(w(d)\big)$$

Die Gewichte sind normalverteilt, weshalb die Verteilungsfunktion $F_W(x)$ die Phi-Funktion $\phi(x)$ ist.

$$F_D(d) = F_W \circ w = \phi\left(\frac{w(d) - \mu}{\sigma}\right) = \frac{1}{2}\left(1 + erf\left(\frac{\beta - \alpha\sqrt{d + \delta} - \mu}{\sigma\sqrt{2}}\right)\right)$$

Hierbei ist $erf(x)$ die Gaußsche Fehlerfunktion:

$$erf(x) = \frac{2}{\sqrt{\pi}} \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1) \cdot k!} x^{2n+1}$$

Es ist zu berücksichtigen, dass der Erwartungswert der Gewichte $\mu = 0$ ist.
Um die Position eines Neurons zu berechnen benötigt man nicht nur die Distanz zum vorherigen Layer, sondern auch seinen Erwartungswert μ_D. Dieser lässt sich über die Definition und mit der zugehörigen Dichtefunktion (Bild 10.2), die es zu berechnen gilt, bestimmen:

$$f_D(d) = F_D'(d) = \varphi\big(w(d)\big) \cdot w'(d) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{\beta - \alpha\sqrt{x+\delta}}{\sigma}\right)^2} \cdot \frac{-\alpha}{2\sqrt{x+\delta}}$$

$$\mu = \int_{-\infty}^{\infty} x f_D(x) dx = \frac{-\alpha}{2\sigma\sqrt{2\pi}} \int_{-\infty}^{\infty} \frac{x}{\sqrt{x+\delta}} e^{-\frac{1}{2}\left(\frac{\beta - \alpha\sqrt{x+\delta}}{\sigma}\right)^2} dx$$

Die Lösung diesen Integrals wird aus Platzgründen hier nicht aufgeführt und ist als Funktion des Erwartungswerts der Distanz in Abhängigkeit der Standardabweichung der Gewichte in Bild 10.3 Abgebildet. Der Tatsächliche Erwartungswert eines Layers ergibt sich jedoch aus der Summe des Erwartungswerts des vorherigen sowie des letzten Layers, damit die Abstände der Neuronen zwischen den Layern immer in etwa gleich groß sind.

$$\mu_D^l = \mu_D^{l-1} + \mu$$

Um bei der Initialisierung zufällige Entfernungen zu erstellen, wird mit der Inversionsmethode gearbeitet. Dabei werden gleichverteilte Zufallszahlen U auf dem Intervall $[0; 1]$ in die Umkehrfunktion der Verteilungsfunktion eingesetzt, die wie folgt berechnet wird.

$$F_D(d) = \frac{1}{2}\left(1 + erf\left(\frac{w(d) - \mu}{\sigma\sqrt{2}}\right)\right) \Leftrightarrow 2F_D(d) - 1 = erf\left(\frac{w(d) - \mu}{\sigma\sqrt{2}}\right)$$

$$\Leftrightarrow \frac{w(d) - \mu}{\sigma\sqrt{2}} = erf^{-1}(2F_D(d) - 1) \Leftrightarrow \beta - \alpha\sqrt{d + \delta} = \sigma\sqrt{2} erf^{-1}(2F_D(d) - 1) + \mu$$

$$\Leftrightarrow d = \left(\frac{\sigma\sqrt{2} erf^{-1}(2F_D(d) - 1) + \mu - \beta}{\alpha}\right)^2 - \delta \Rightarrow F_D^{-1}(d) = \left(\frac{\sigma\sqrt{2} erf^{-1}(2d - 1) + \mu - \beta}{\alpha}\right)^2$$

Für $erf^{-1}(x)$ gilt:

$$erf^{-1}(x) = \sum_{k=0}^{\infty} \frac{c_k}{2k+1} \left(\frac{\sqrt{\pi}}{2} x\right)^{2k+1}, \qquad c_k = \sum_{i=0}^{k-1} \frac{c_i \cdot c_{i-k-1}}{(i+1)(2i+1)}$$

Eine Entfernung lässt sich also Folgendermaßen berechnen: $d = F_D^{-1}(U)$, sodass sich dann für die resultierende Koordinate $x_{k'} = \mu_D^{l-1} + d$ ergibt.

Die besondere Initialisierung für den dreidimensionalen Raum, bei der die Neuronen jeden zweiten Layers gleichmäßig auf einen Ring aufgeteilt werden erfordert, zum einen, dass die Entfernung der vorherigen Neuronen zum Ringmittelpunkt dem Radius des Rings entspricht, sodass die Gradients für die weiteren Koordinatenachsen gleichmäßig aufgeteilt sind. Zum anderen soll der Erwartungswert der Entfernungen aller Neuronen aufeinanderfolgender Layer noch immer dem zuvor berechneten Erwartungswert für einen Layer entsprechen, das heißt, der Abstand zweier Neuronen soll immer noch mit f_D verteilt sein. Für den Radius r ergibt sich:

$$d = \sqrt{r^2 + r^2} \Leftrightarrow r = \frac{d}{\sqrt{2}}$$

Die Neuronen sollen auf einem Kreis senkrecht zu einer gewählten Koordinatenachse verteilt sein, weshalb die anderen Koordinatenkomponenten mit dem Sinus beziehungsweise Cosinus eines zufälligen Winkels φ multipliziert werden, der sich aus einer Gleichverteilung U auf dem Intervall $[0; 1]$ ergibt:

$$x_1 = \mu_D^{l-1} + \frac{d}{\sqrt{2}}\ , \quad x_2 = \frac{d}{\sqrt{2}} cos(\varphi)\ , \quad x_3 = \frac{d}{\sqrt{2}} sin(\varphi)$$

$$\varphi = 2\pi u\ ,\ u \in U$$

Mit diesen Werten entspricht der Abstand der Neuronen zum Erwartungswert des vorherigen Layers der erforderlichen Entfernung d:

$$\sqrt{x_1^2 + x_2^2 + x_3^2} \Rightarrow \sqrt{\left(\frac{d}{\sqrt{2}}\right)^2 + \left(\frac{d}{\sqrt{2}} cos(\varphi)\right)^2 + \left(\frac{d}{\sqrt{2}} sin(\varphi)\right)^2}$$

$$= \sqrt{d^2\left(\frac{1}{2} + \frac{1}{2}\left(sin^2(\varphi) + cos^2(\varphi)\right)\right)} = d\sqrt{\frac{1}{2} + \frac{1}{2}} = d$$

Für die Kugelinitialisierung wird ein Vektor der Länge d auf einer der Koordinatenachsen positioniert und um die anderen beiden Achsen mit zwei gleichverteilten zufälligen Winkeln φ und θ gedreht, sodass die Form der Einhüllenden bezüglich der Positionierung der Neuronen einer Kugeloberfläche entspricht.

Wenn als Koordinatenachse des Einheitsvektors die x_1-Achse gewählt wird, ergibt sich für diesen: $\vec{v} = (d, 0,0)$

Mit einer Rotation um die x_2- und dann x_3-Achse folgt:

$$R_{x_2} \cdot \vec{v} = \begin{pmatrix} cos(\varphi) & 0 & sin(\varphi) \\ 0 & 1 & 0 \\ -sin(\varphi) & 0 & cos(\varphi) \end{pmatrix} \cdot \begin{pmatrix} d \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} d \cdot cos(\varphi) \\ 0 \\ -d \cdot sin(\varphi) \end{pmatrix}$$

$$R_{x_3} \cdot \vec{v}_1 = \begin{pmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} d \cdot cos(\varphi) \\ 0 \\ -d \cdot sin(\varphi) \end{pmatrix} = \begin{pmatrix} d \cdot cos(\varphi) \cdot cos(\theta) \\ d \cdot cos(\varphi) \cdot sin(\theta) \\ -d \cdot sin(\varphi) \end{pmatrix}$$

Daraus ergibt sich für die Koordinaten:

$$x_1 = d \cdot cos(\varphi) \cdot cos(\theta)\,, \quad x_2 = d \cdot cos(\varphi) \cdot sin(\theta)\,, \quad x_3 = -d \cdot sin(\varphi)$$

$$\varphi = 2\pi u_1\,, \quad \theta = 2\pi u_2\,, \quad u_1, u_2 \in U$$

Auch hierbei ist der Abstand zum Koordinatenursprung stets d:

$$\begin{aligned} \sqrt{x_1^2 + x_2^2 + x_3^2} &= \sqrt{\big(dcos(\varphi)cos(\theta)\big)^2 + \big(dcos(\varphi)sin(\theta)\big)^2 + \big(-dsin(\varphi)\big)^2} \\ &= \sqrt{d^2\big(cos^2(\varphi)cos^2(\theta) + cos^2(\varphi)sin^2(\theta) + sin^2(\varphi)\big)} \\ &= d\sqrt{cos^2(\varphi)\big(cos^2(\theta) + sin^2(\theta)\big) + sin^2(\varphi)} \\ &= d\sqrt{cos^2(\varphi) + sin^2(\varphi)} = d \end{aligned}$$

Der Erwartungswert der Gradients für die Entfernung ergibt sich aus $\partial d / \partial x_k^l$ unter der Verwendung von den Durchschnittswerten der Variablen.

$$\frac{\partial d}{\partial x_k^l} = \frac{x_{k'}^l - x_{k'}^{l-1}}{d}$$

Da die Neuronen eines von zwei aufeinanderfolgenden Layern stets um den Ursprung initialisiert werden, ist eine der Variablen im Zähler im Mittel Null. Die andere Variable hat für jede Dimension durchschnittlich den Wert der mittleren Höhe eines Halbkreises, da die Neuronen kugelförmig angeordnet sind und nur eine Dimension betrachtet wird. Somit ergibt sich für den Mittelwert m aus der Gleichung für einen Kreis:

$$x^2 + y^2 = r^2 \Leftrightarrow y = \sqrt{r^2 - x^2}$$

$$m = \frac{1}{r - (-r)} \int_{-r}^{r} \sqrt{r^2 - x^2}\,dx = \frac{1}{2r} \cdot \frac{1}{2}\pi r^2 = \frac{\pi r}{4}$$

Da der Radius der Kugel der Distanz der Neuronen entspricht ergibt sich:

$$\frac{x_{k'}^{l} - x_{k'}^{l-1}}{d} \Rightarrow \frac{\frac{\pi d}{4}}{d} = \frac{\pi}{4}$$

Dieser Gradient ist im Gegensatz zu der Ringinitialisierung mit $r = \frac{d}{\sqrt{2}}$ größer:

$$\frac{x_{k'}^{l} - x_{k'}^{l-1}}{d} \Rightarrow \frac{\frac{\pi}{4}\frac{d}{\sqrt{2}}}{d} = \frac{1}{\sqrt{2}}\frac{\pi}{4} < \frac{\pi}{4}$$

Aufgrund dessen wird eine Änderung in der Position der Neuronen gegenüber der anderen Initialisierung verstärkt zum Lernen herangezogen.

Wenn die Neuronen um den Ursprung herum Initialisiert werden, geschieht dies mit dem Erwartungswert $\mu = 0$ und einer Varianz $Var(D_1)$. Die Gleiche Varianz $Var(D_2)$ gilt für die Verteilung der Entfernungen vom Ursprung der in der Kugel anzuordnenden Neuronen, wobei der Erwartungswert d ist.
Um die Standardabweichung bei der linearen Gewichtsfunktion zu bestimmen, mit der die Position der Neuronen gewählt wird, hilft Folgende Betrachtung:

$$w = \alpha d + \beta \Rightarrow Var(W) = Var(\alpha D + \beta)$$

Die Entfernung bezüglich einer Dimension ergibt sich aus den zugehörigen Koordinatenkomponenten, die je mit $Var(D_1)$ beziehungsweise $Var(D_2)$ initialisiert werden. Daraus sowie aus den Gesetzmäßigkeiten der Linearität und Summe unabhängiger Zufallszahlen für die Varianz ergibt sich:

$$Var(\alpha D + \beta) = \alpha^2 Var(D) = \alpha^2 Var(D_1 + D_2) = \alpha^2\big(Var(D_1) + Var(D_2)\big)$$

$$Var(D_1) = Var(D_2) \;\Rightarrow\; Var(W) = \alpha^2 \cdot 2 \cdot Var(D_1)$$

Mit $Var(W) = 2/n$ im Falle der ReLu-Initialisierung ergibt sich für die Standardabweichung der Normalverteilung, mit der die Entfernung d und die Neuronen im Ursprung je Koordinatenachse zu initialisieren sind:

$$\Rightarrow \frac{2}{n} = 2\alpha^2 Var(D_1) \Leftrightarrow Var(D_1) = \frac{1}{\alpha^2 n} \Rightarrow \sigma = \frac{1}{\alpha\sqrt{n}}$$

10.6 Bilder

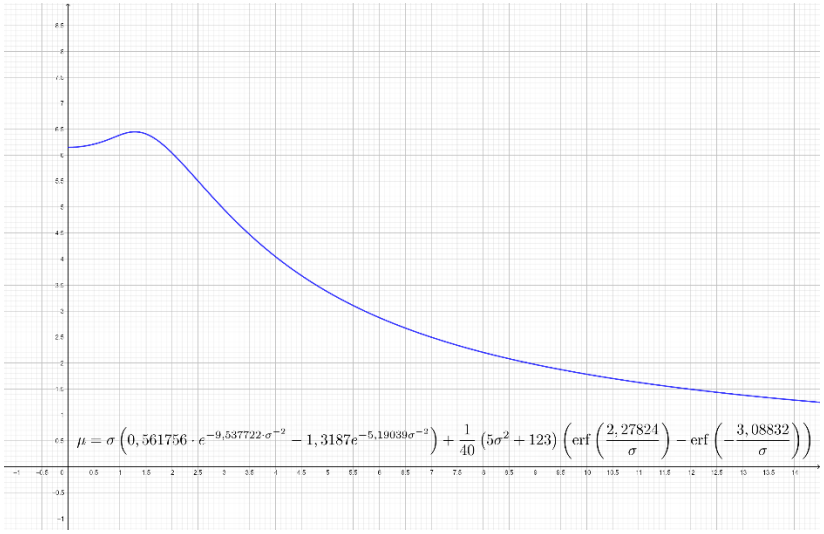


Bild 10.3

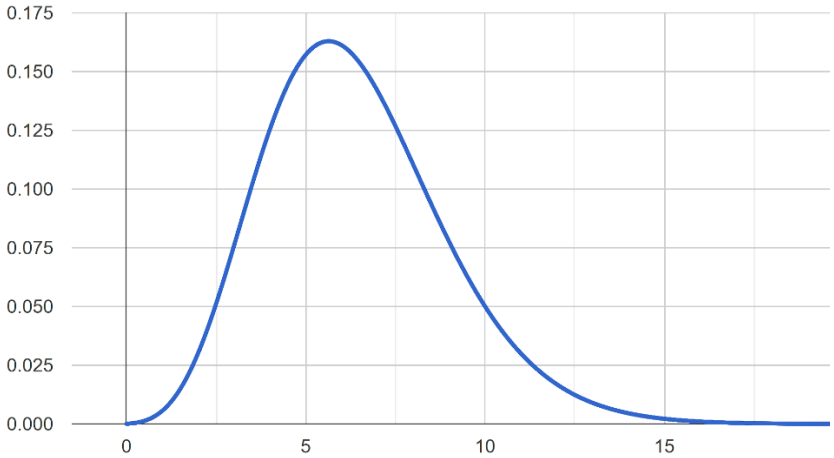


Bild 10.2

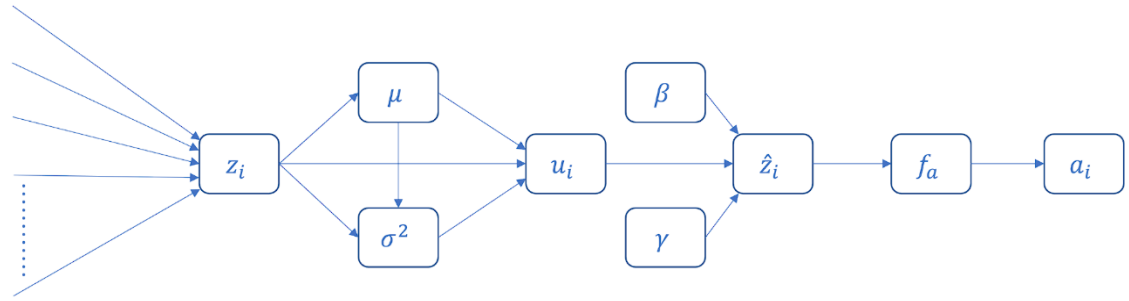


Bild 10.1

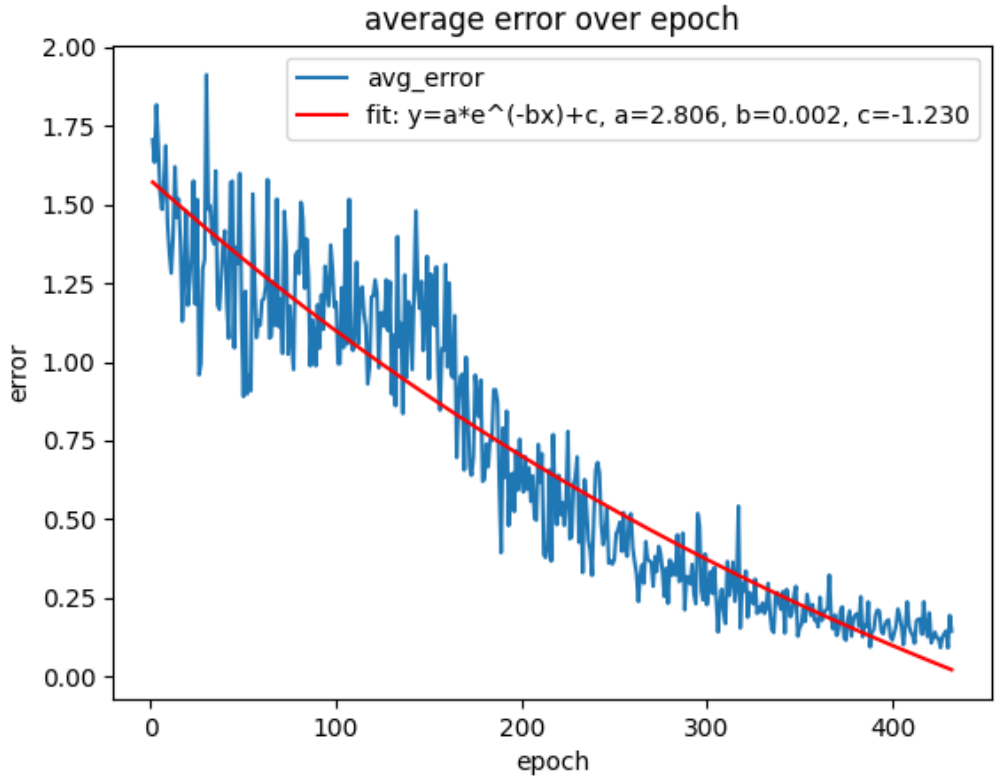


Bild 7.4

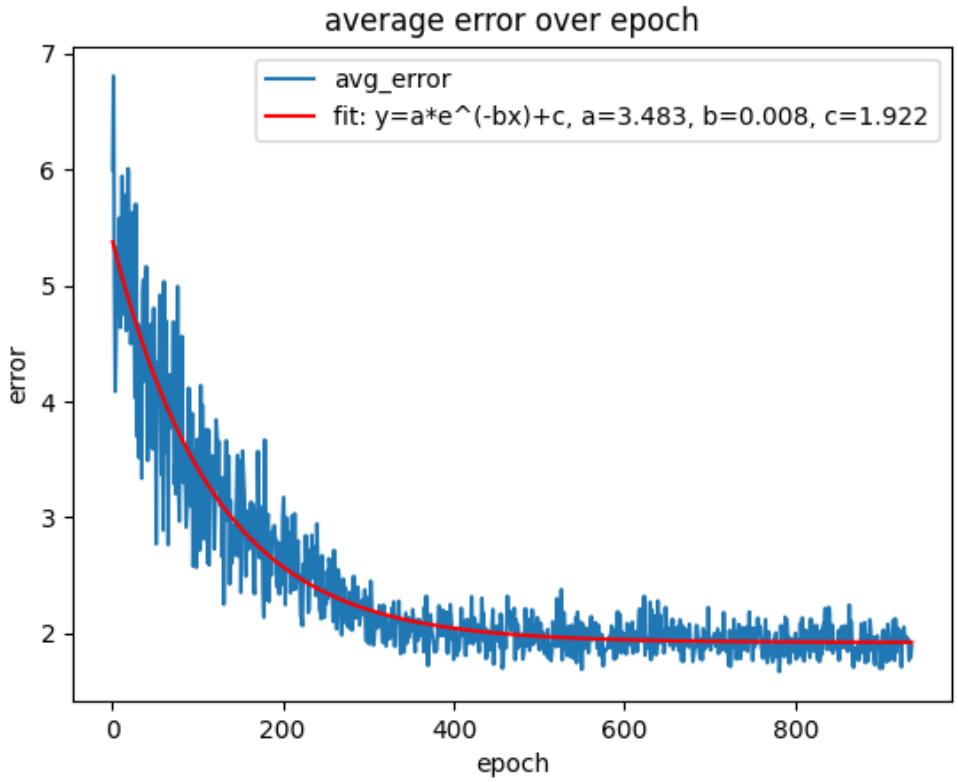


Bild 7.3

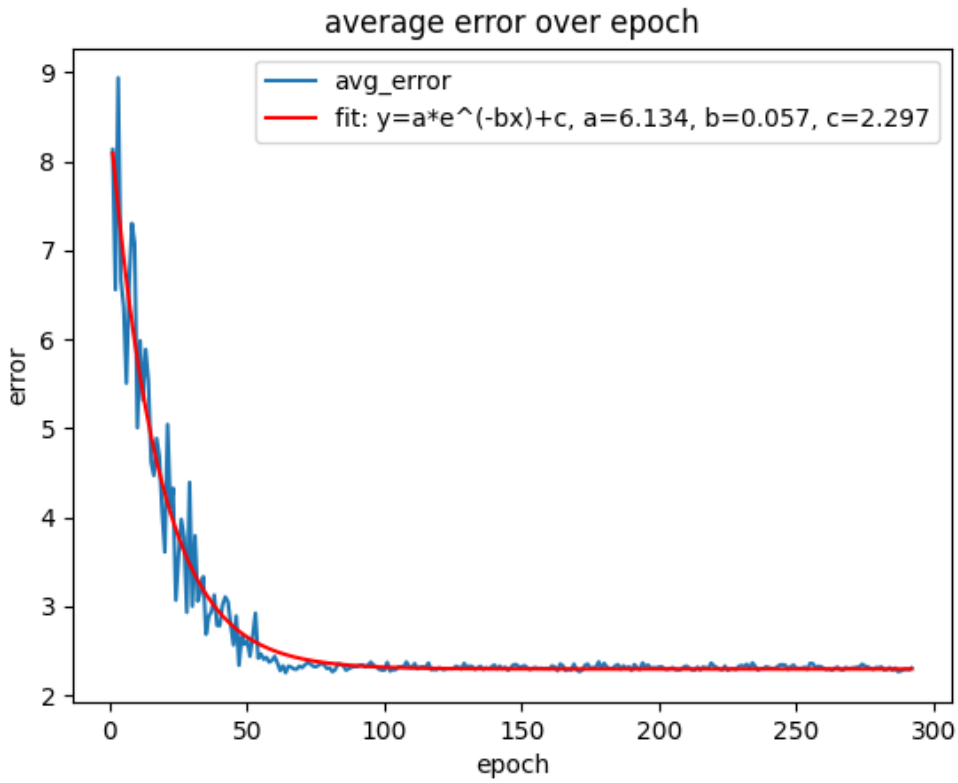


Bild 7.2

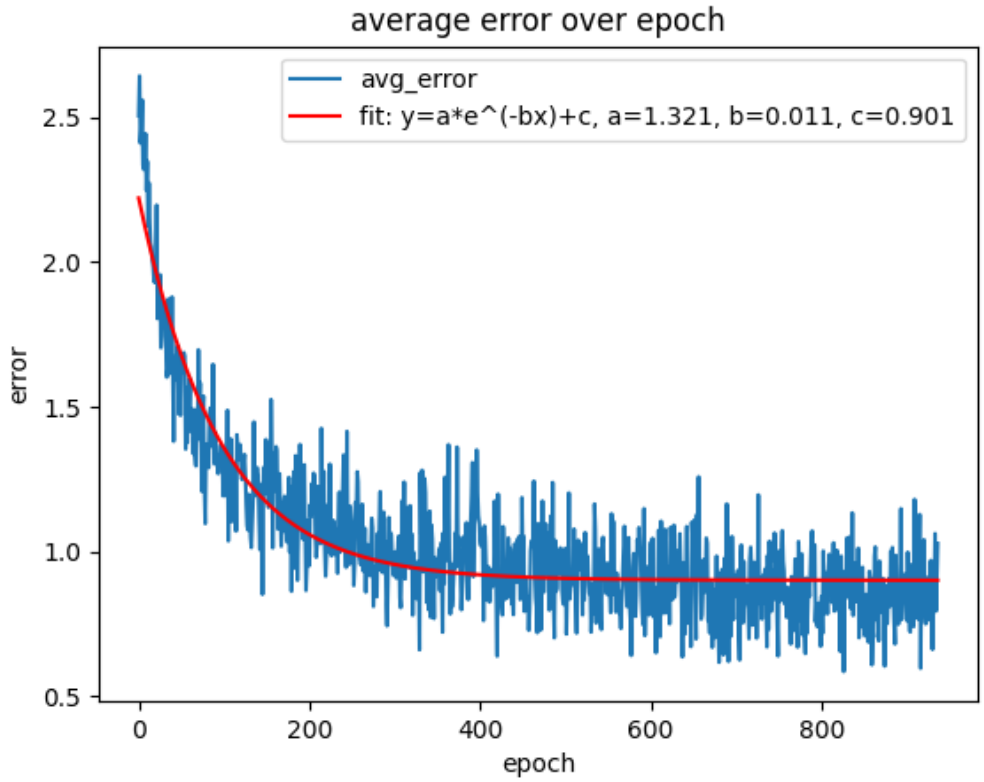


Bild 7.1

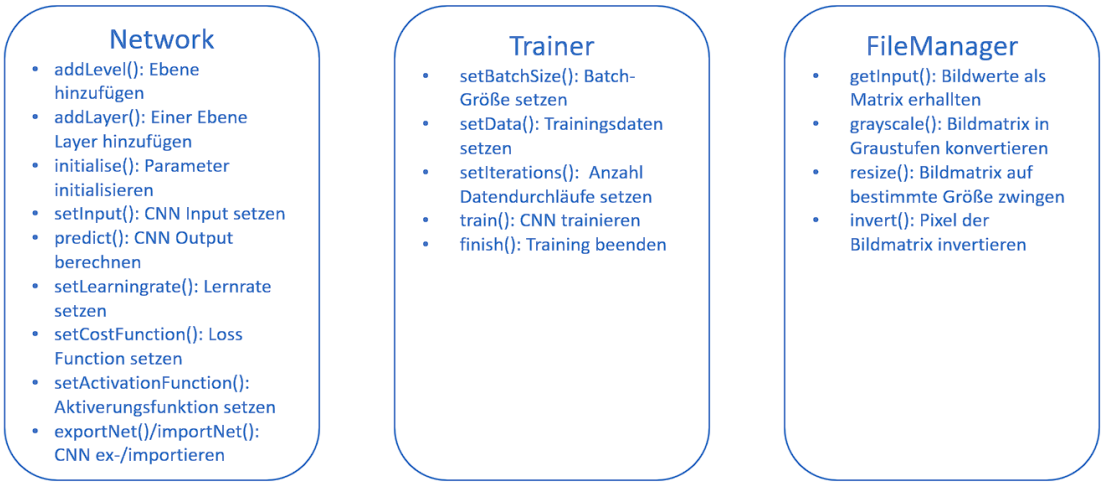


Figure 6.2

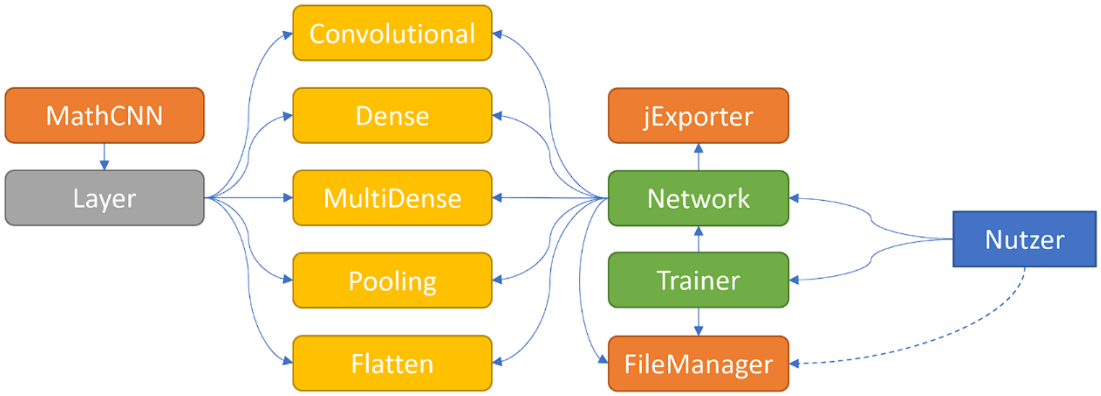


Bild 6.1

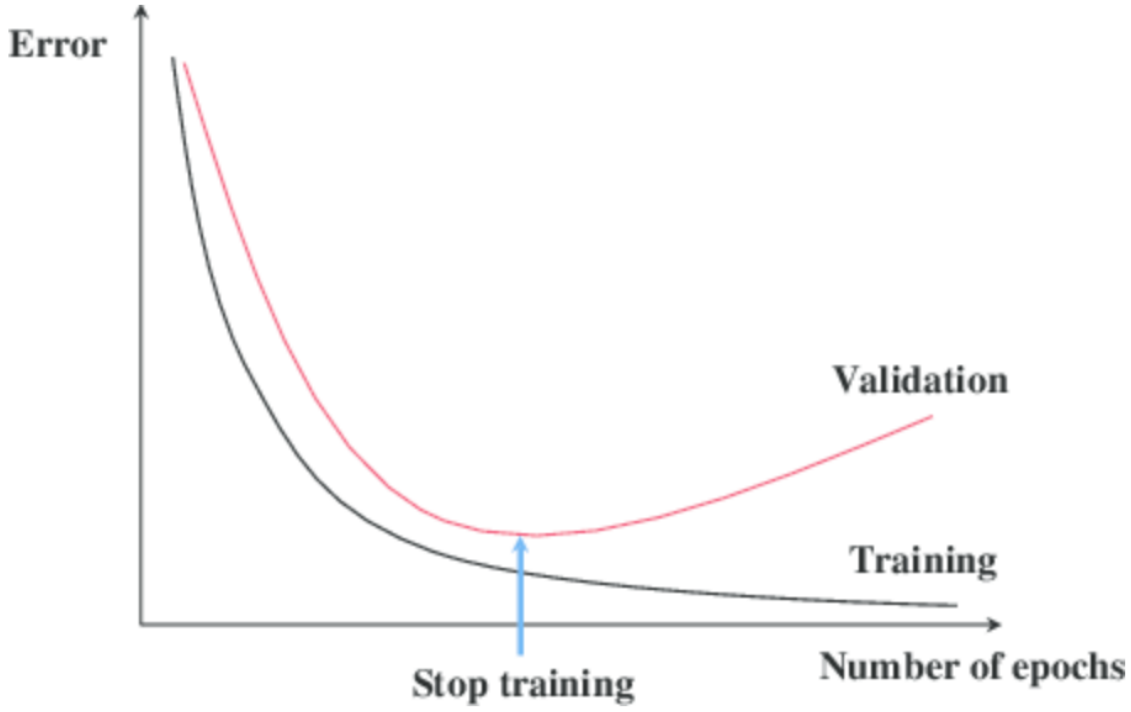


Bild 5.6

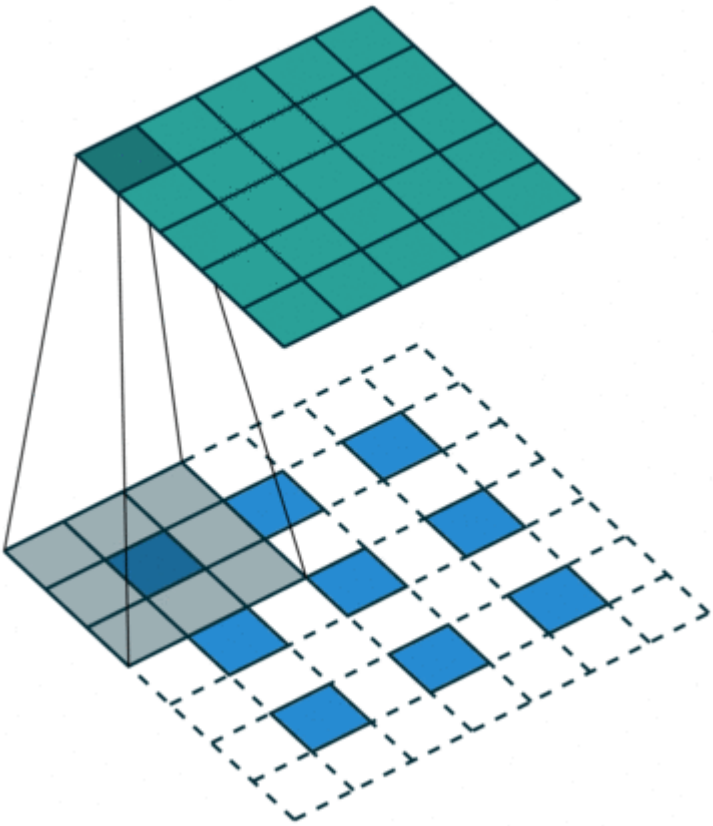


Bild 5.5

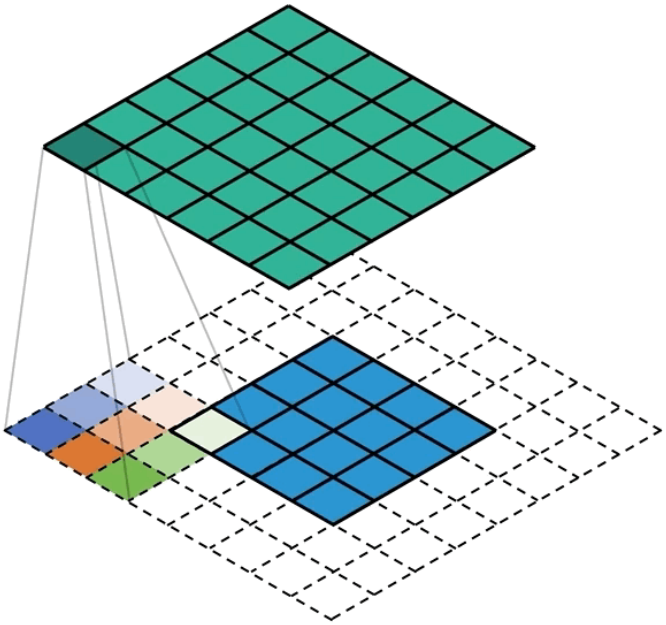


Bild 5.4

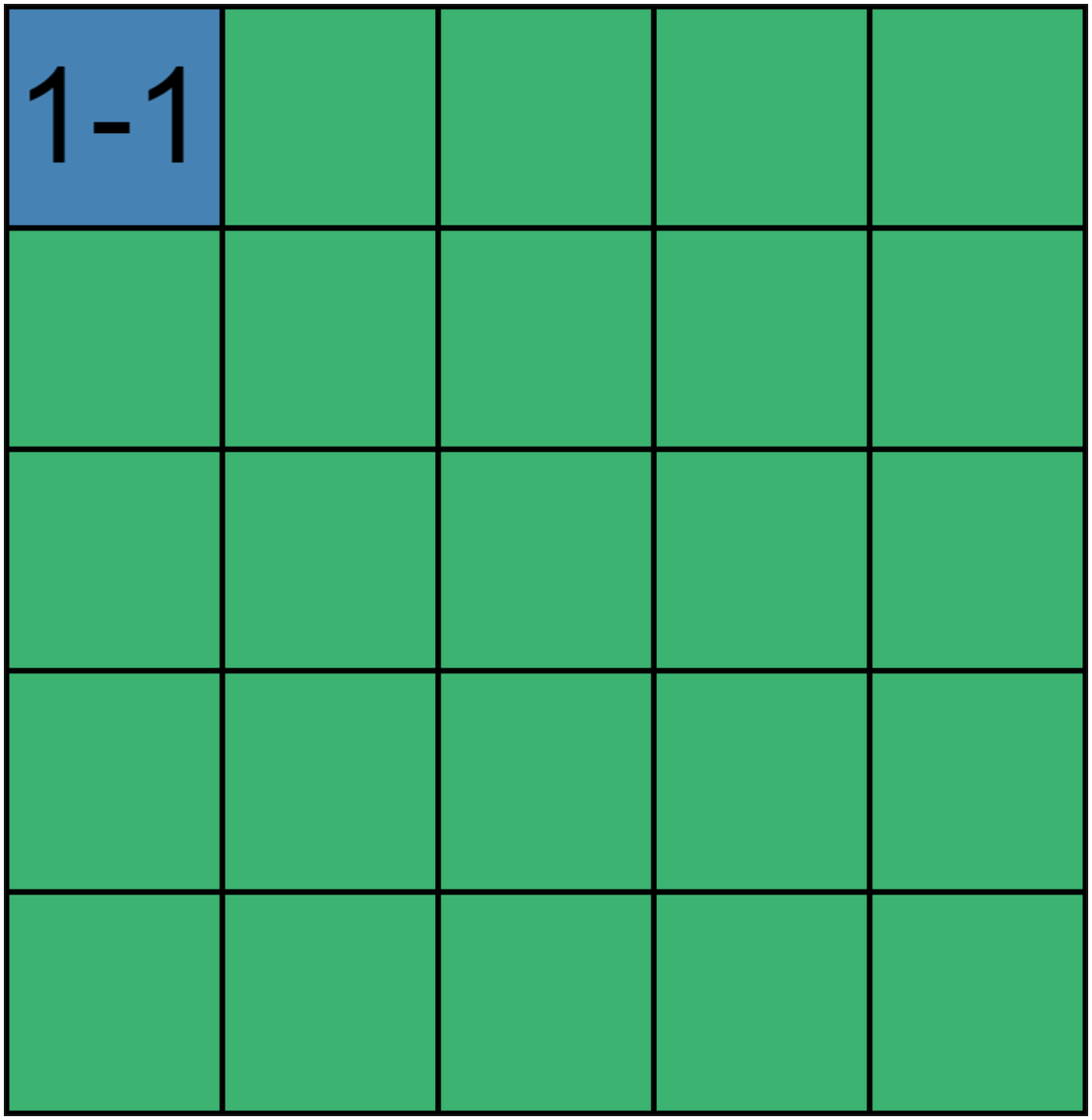


Bild 5.3

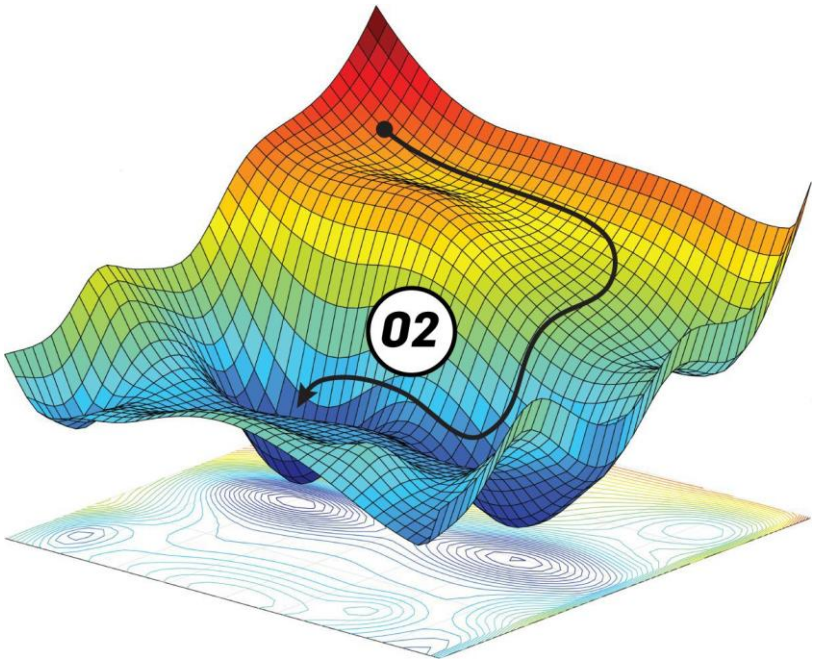


Bild 5.2

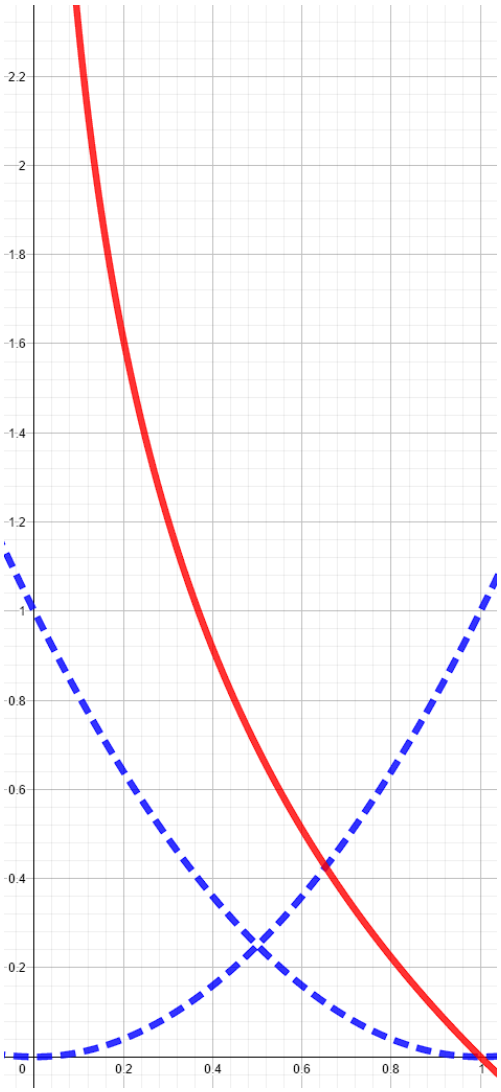


Bild 5.1

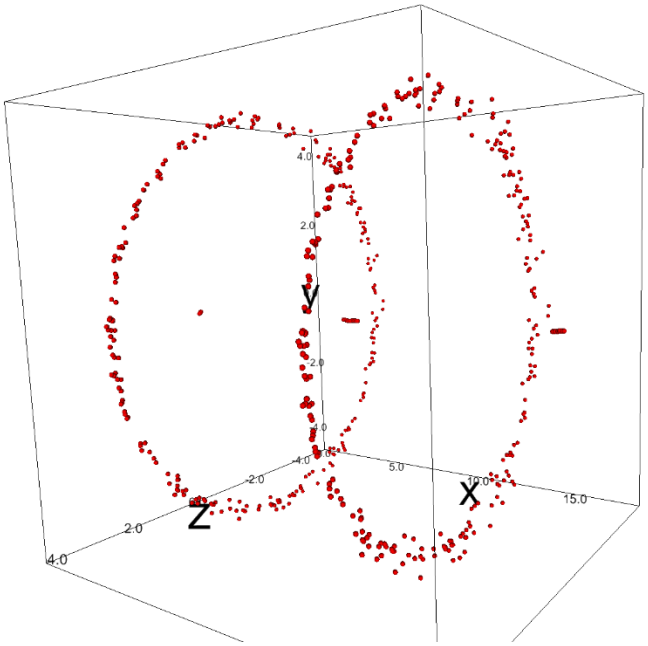


Bild 5.0

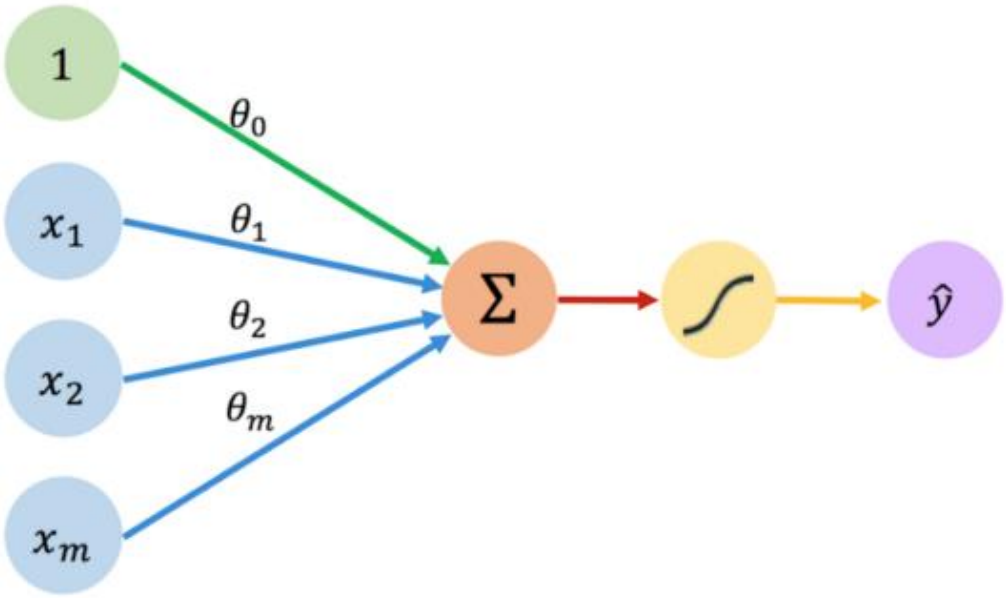


Bild 4.12

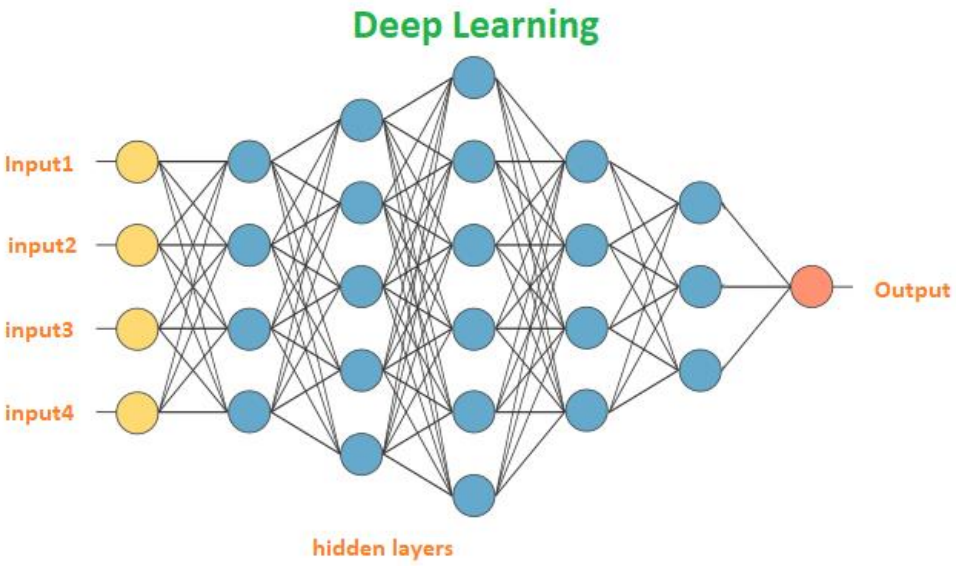


Bild 4.11

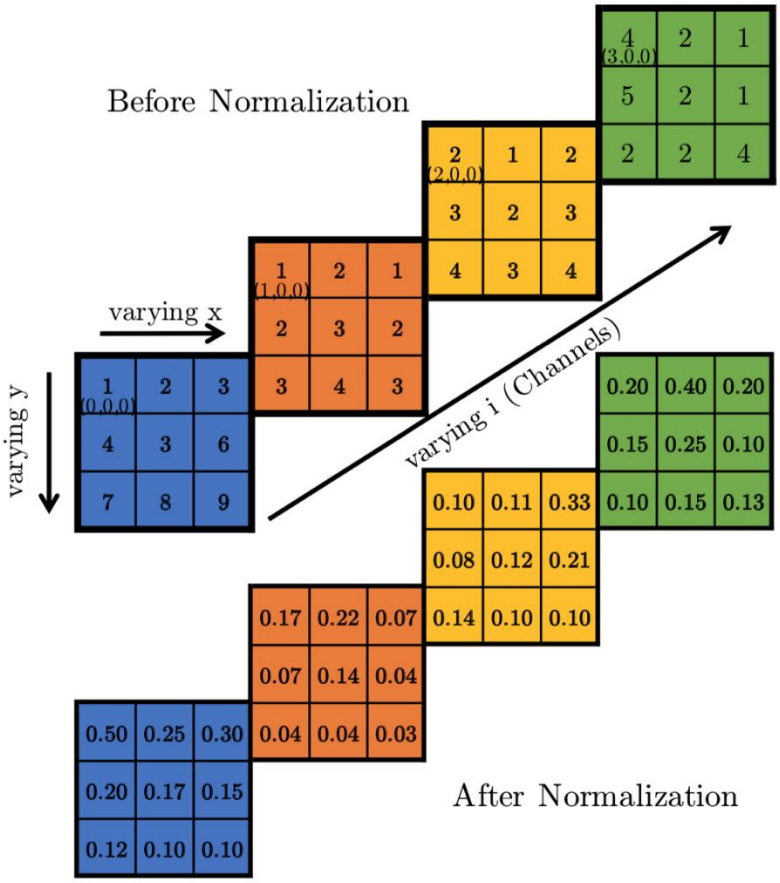


Figure 4.10

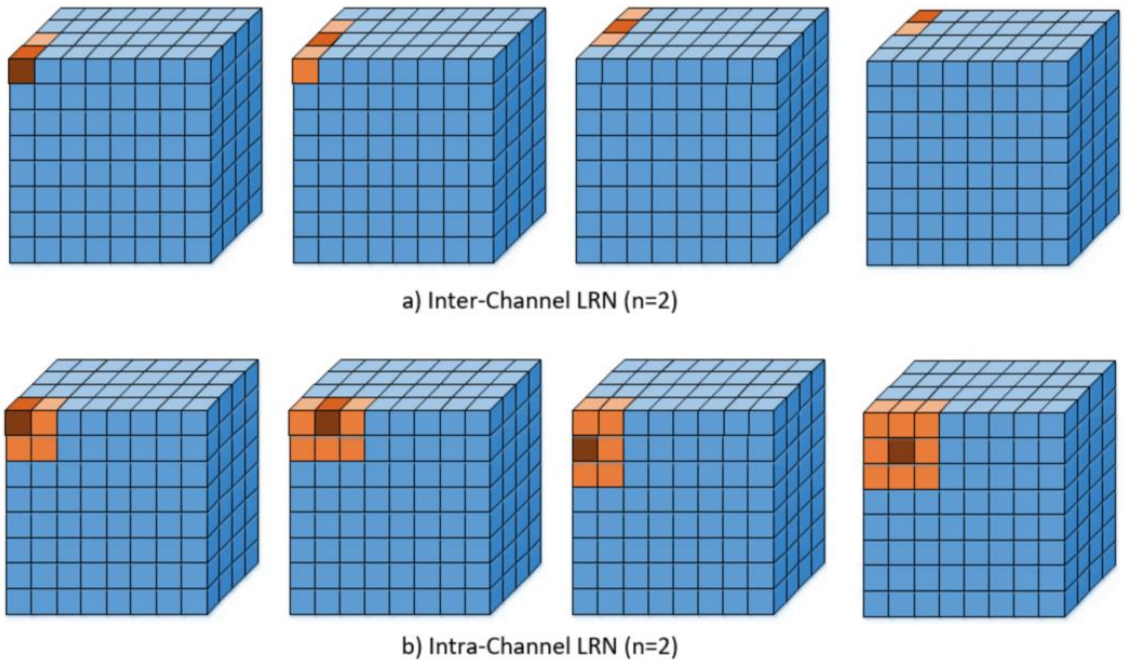


Bild 4.9

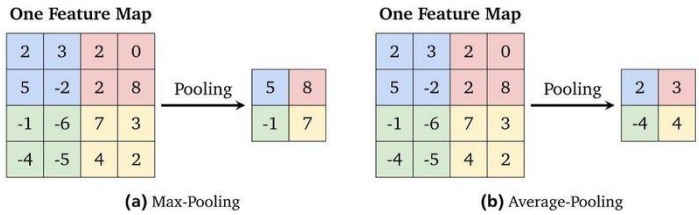


Bild 4.8

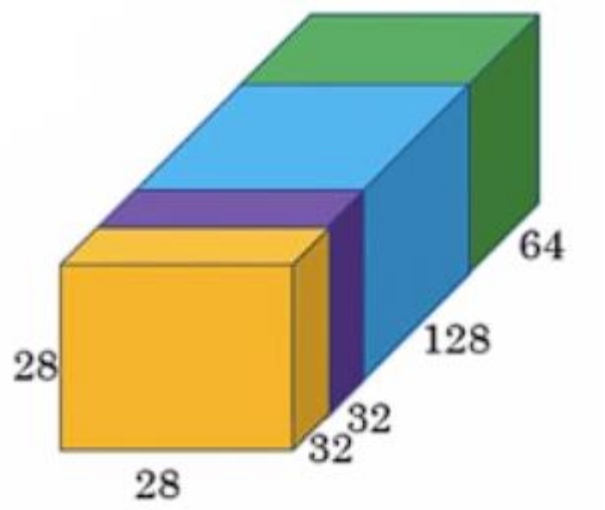


Bild 4.7

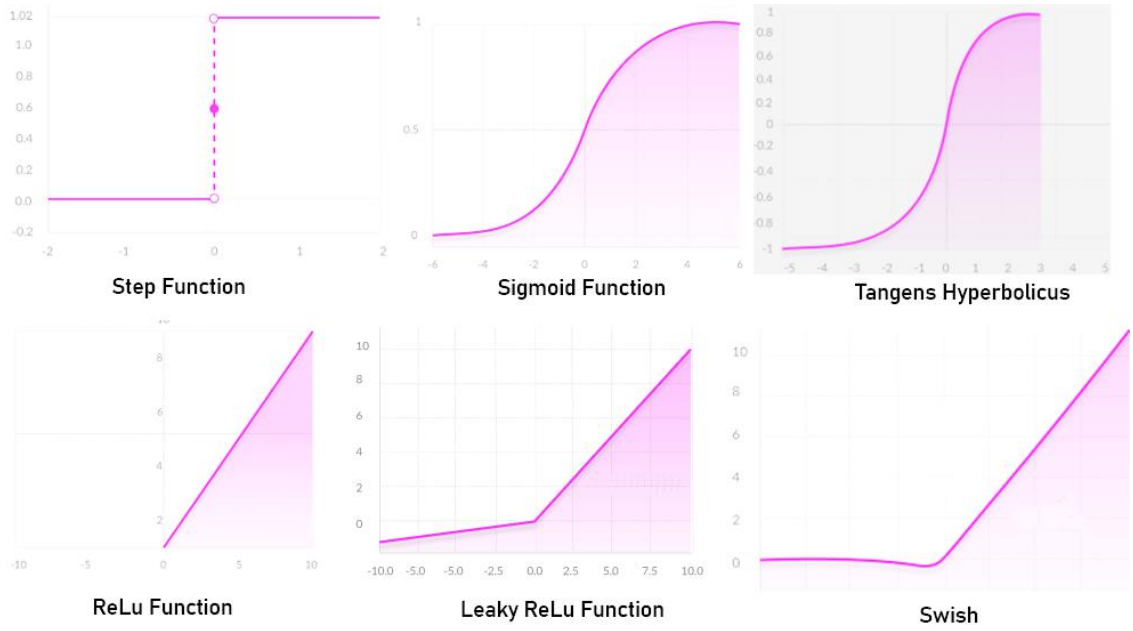


Bild 4.6

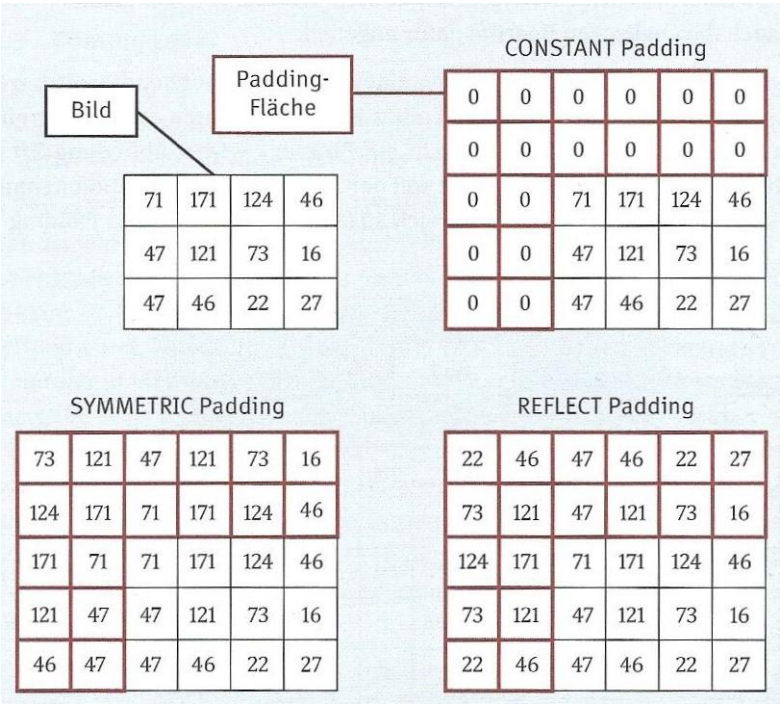


Bild 4.5

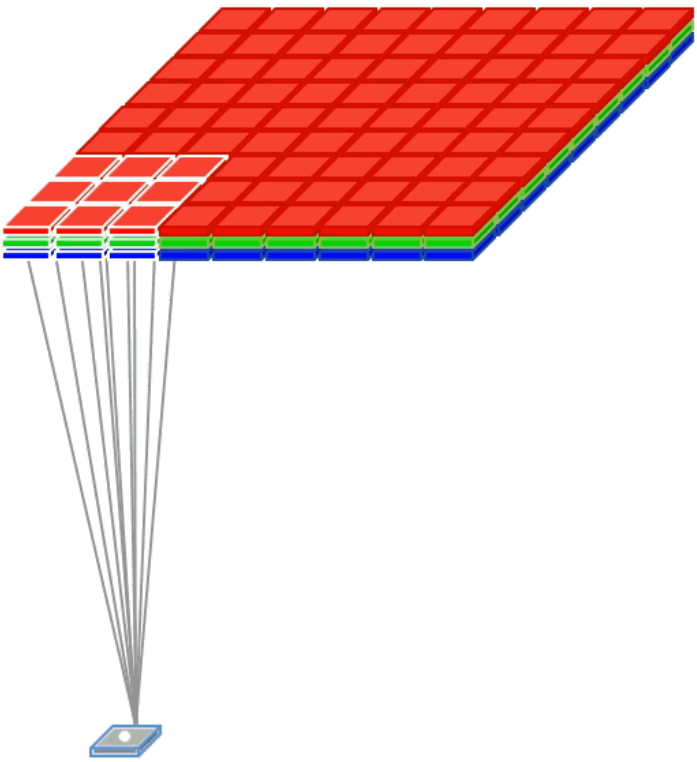


Bild 4.4

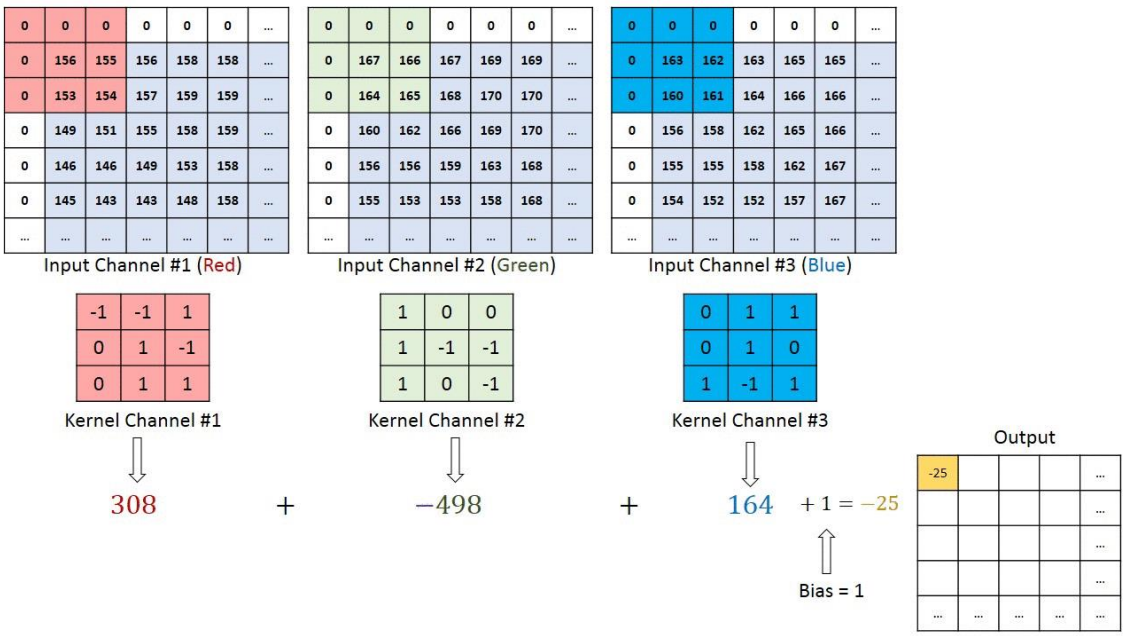


Bild 4.3

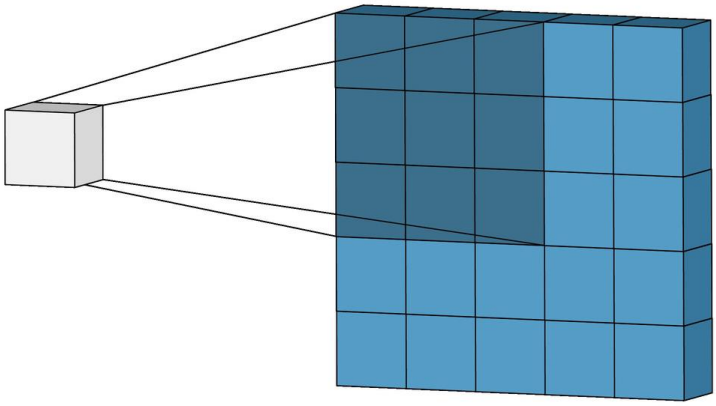


Bild 4.2

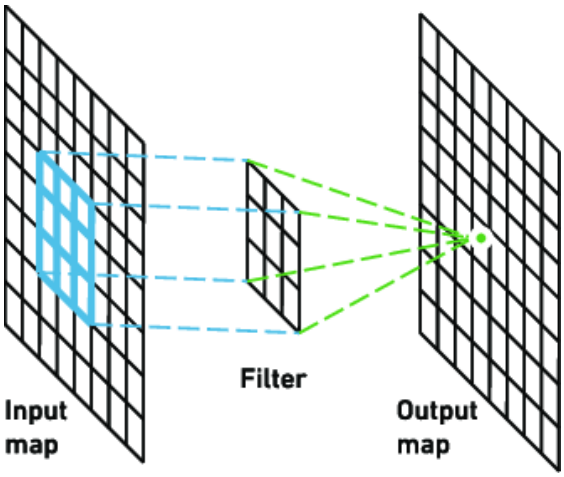


Bild 4.1

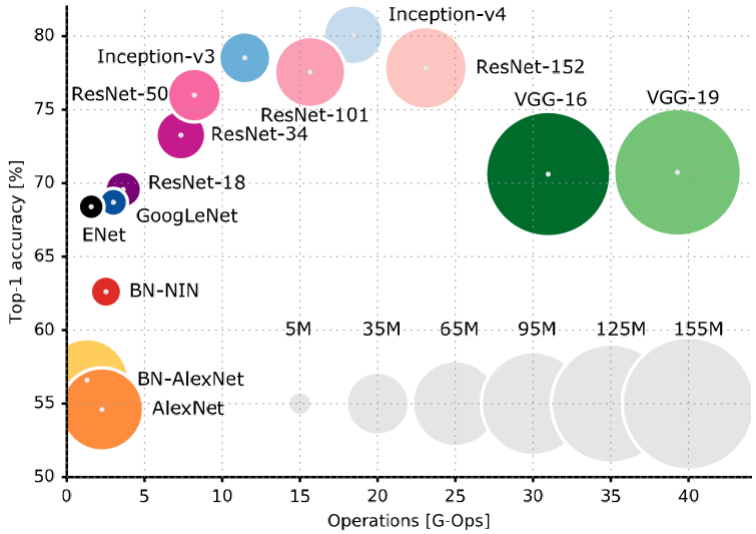


Bild 3.7

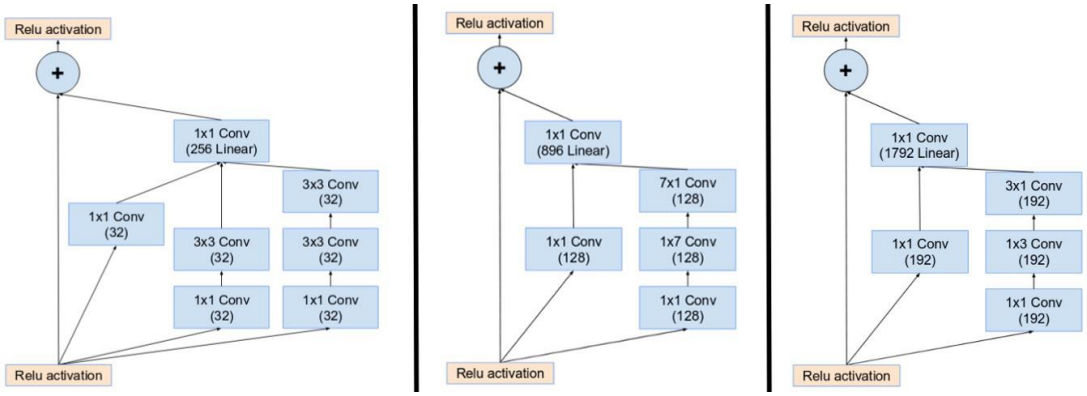


Bild 3.6

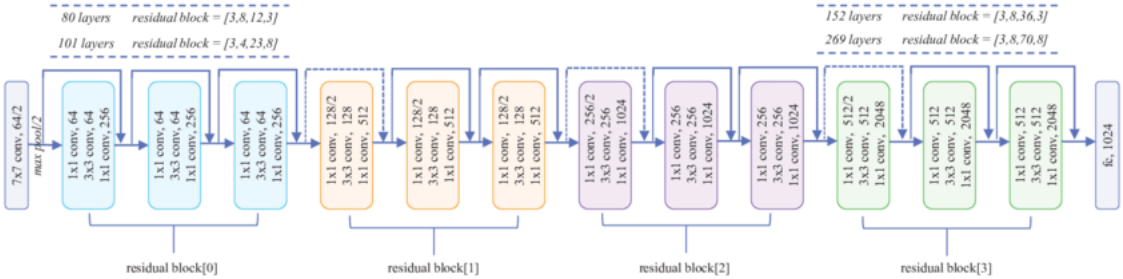


Bild 3.5

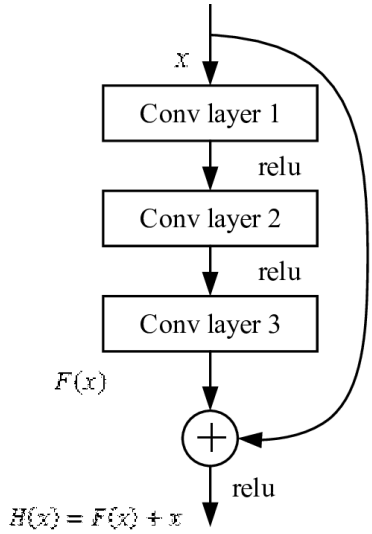


Bild 3.4

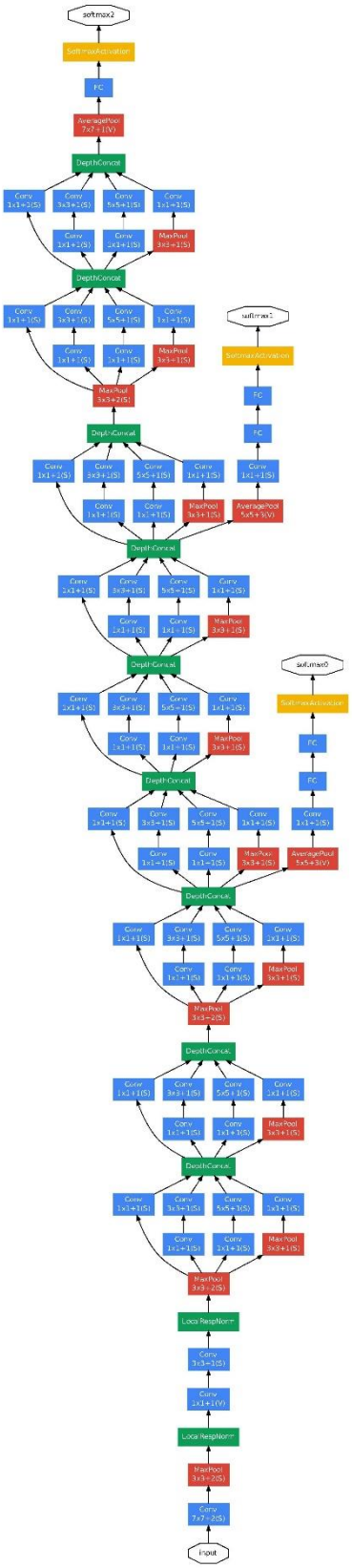

softmax2
SoftmaxActivation
FC
AveragePool 7x7+1(V)
DepthConcat
Conv 1x1+2(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 1x1+1(S)
Conv 2x1+2(S)
Conv 2x1+1(S)
MaxPool 3x3+1(S)
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
MaxPool 3x3+2(S)
softmax1
SoftmaxActivation
FC
FC
DepthConcat
Conv 1x1+2(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
AveragePool 5x5+3(V)
DepthConcat
Conv 2x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
DepthConcat
softmax0
SoftmaxActivation
FC
FC
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 1x1+2(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+2(S)
AveragePool 5x5+3(V)
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 2x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
MaxPool 3x3+2(S)
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 2x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
MaxPool 3x3+2(S)
Local RespNorm
Conv 3x3+1(S)
Conv 2x1+1(V)
Local RespNorm
MaxPool 3x3+2(S)
Conv 7x7+2(S)
Input

Bild 3.3

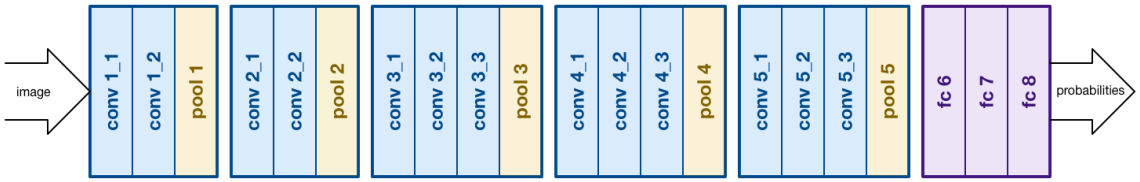


Bild 3.2

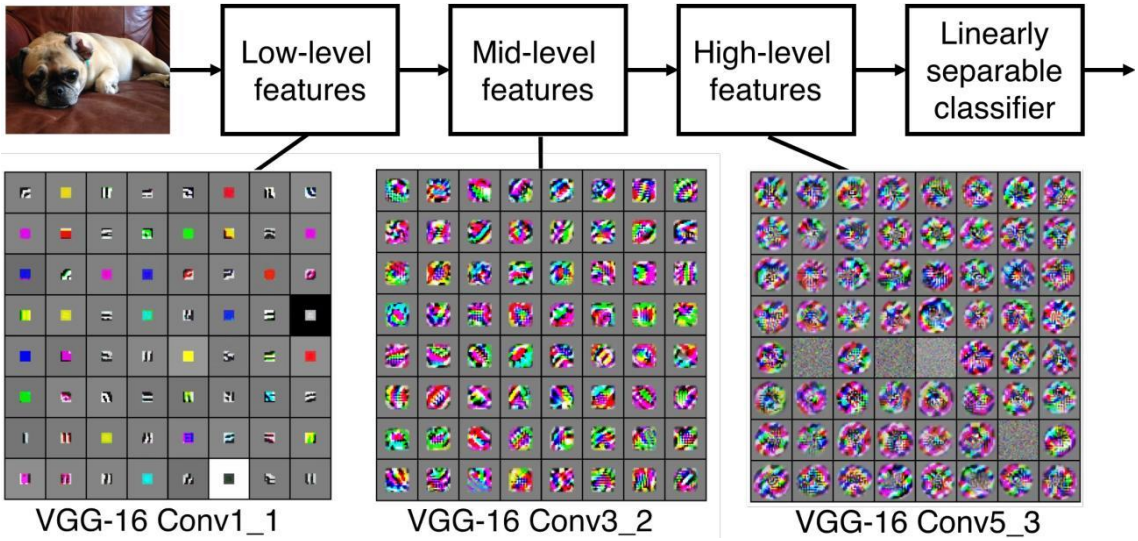


Bild 3.1

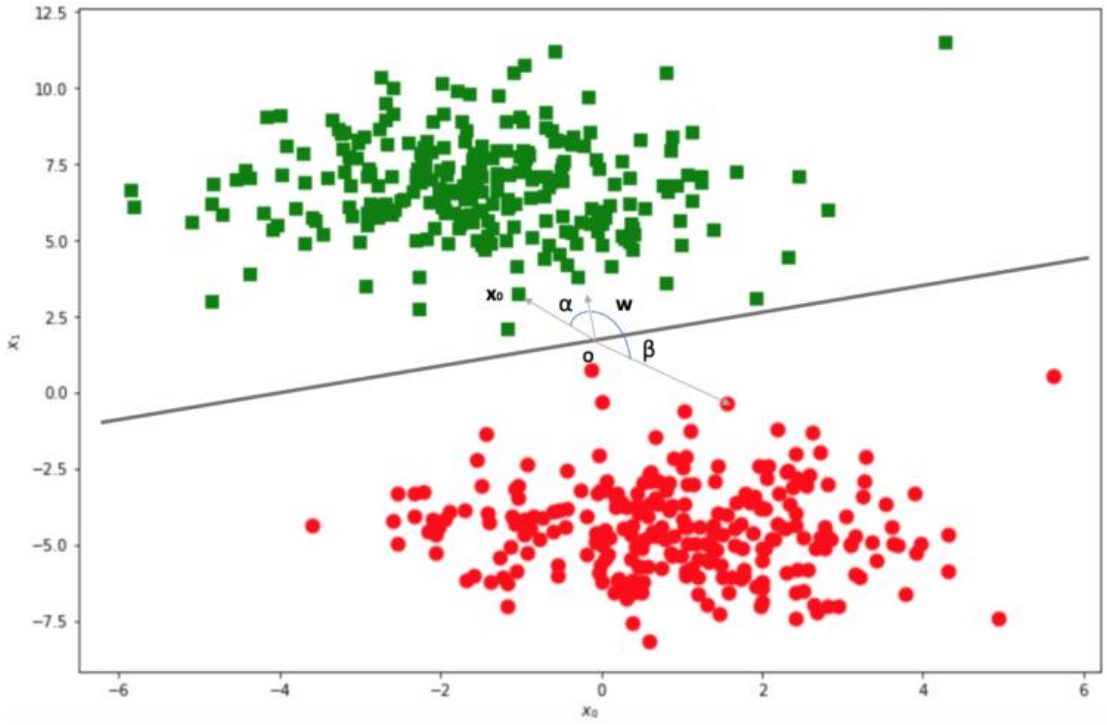


Bild 2.2

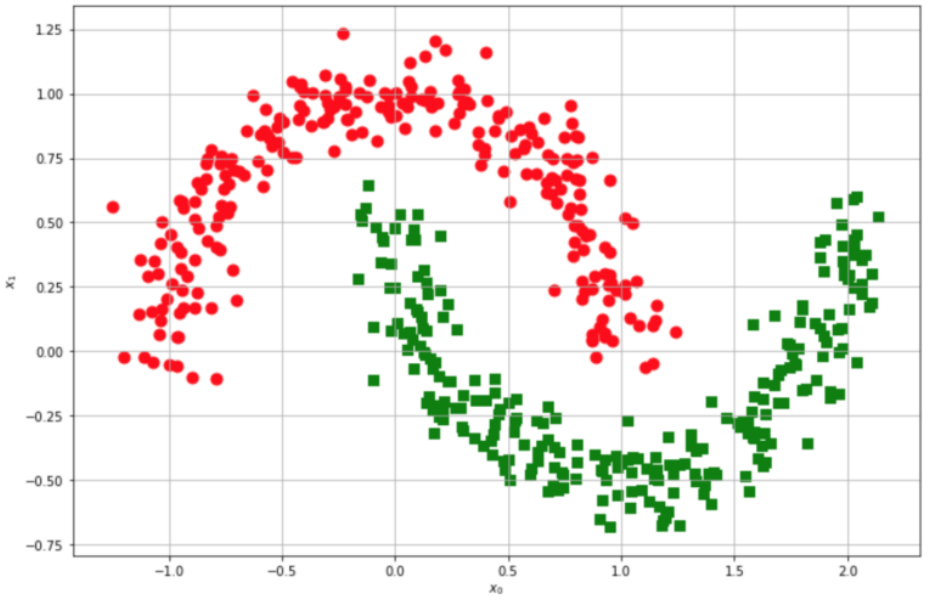


Bild 2.1

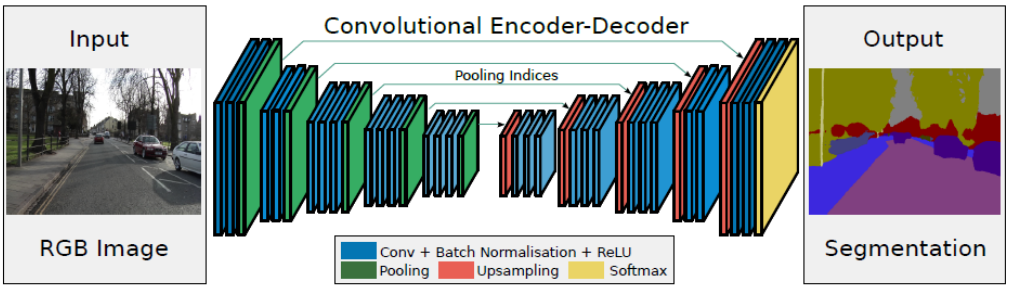


Bild 1.2

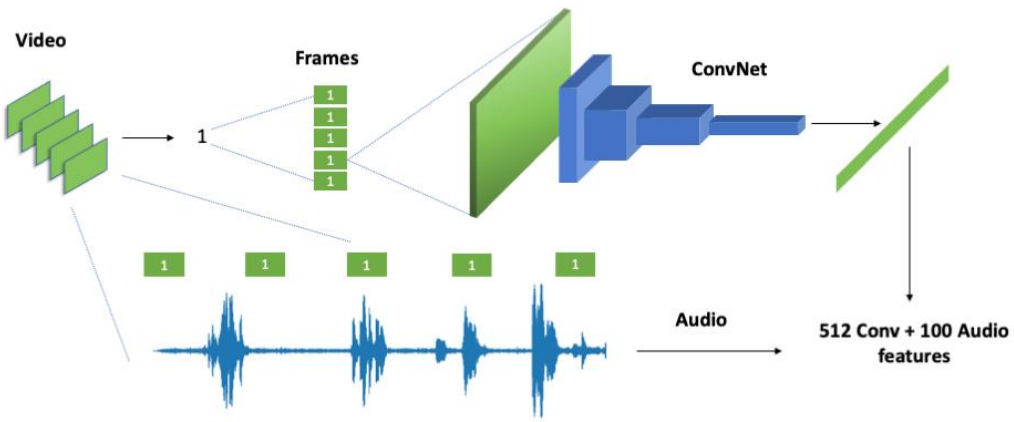


Bild 1.1

10.7 Programmcode

activations.h

```
#ifndef _activationsH_
#define _activationsH_
#include <vector>
#include <stxxl/vector>
typedef std::vector<float> denseA;
typedef std::vector<std::vector<std::vector<float>>> convA;
typedef std::vector<std::vector<float>> denseW;
typedef std::vector<std::vector<float>> featureMap;
typedef std::vector<float> vectorf;
typedef std::vector<std::vector<float>> vectorff;
typedef std::vector<std::vector<std::vector<float>>> convAL;
typedef std::vector<convA> vconvAL;
#endif
```

conv.cpp

```
#include "Conv.h"
#include <math.h>
#include <iostream>
#include <thread>

Conv::Conv(int actMaps, vconvAL* input_, int kernelS_, int stride_, int padding_) : input(input_
), stride(stride_), kernelS(kernelS_), padding(padding_) {
    sl_1[0] = (*input)[0].size();
    sl_1[1] = (*input)[0][0].size();
    sl_1[2] = (*input)[0][0][0].size();
    size[0] = MathCNN::convSize(sl_1[0], stride, padding, kernelS);
    size[1] = MathCNN::convSize(sl_1[1], stride, padding, kernelS);
    size[2] = actMaps;
    kernel.reserve(size[2]);
    for (unsigned int i = 0; i < size[2]; i++) {
        kernel.emplace_back(convA(kernelS, std::vector<std::vector<float>>(kernelS, std::vector<
float>(sl_1[2]))));
    }
    layerDim.reserve(size[0]);
    for (unsigned int i = 0; i < size[0]; i++) {
        layerDim.emplace_back(std::vector<std::vector<float>>(size[1], std::vector<float>(size[2
])));
    }
};
void Conv::forward() {
    vconvAL z;
    if (padding == 0) {
        std::vector<std::vector<int>> tDistribution;
        if (input->size() == 1) {
            tDistribution = { {0, 1} };
        }
        else {
            int cut = (int)input->size() / 4;
            tDistribution.push_back({ 0, cut });
```

```
            tDistribution.push_back({ cut, 2 * cut });
            tDistribution.push_back({ 2 * cut, 3 * cut });
            tDistribution.push_back({ 3 * cut, (int)input->size() });
        }
        std::vector<std::thread> kernelThreads = std::vector<std::thread>(tDistribution.size());
        int(&size_)[3] = size;
        std::vector<convA>& kernel_ = kernel;
        vconvAL* input_ = input;
        vconvAL& z_ = z;
        convA& layerDim_ = layerDim;
        int& stride_ = stride;
        for(unsigned int i=0; i<input->size(); i++) {
            z_.push_back(layerDim);
        }
        for (unsigned int t = 0; t < tDistribution.size(); t++) {
            kernelThreads[t] = std::thread([&size_, &kernel_, input_, &stride_, tDistribution, t
, &z_, &layerDim_]() {
                for (unsigned int i = tDistribution[t][0]; i < tDistribution[t][1]; i++) {
                    convA tmp;
                    for (unsigned int j = 0; j < size_[2]; j++) { //actMaps
                        featureMap actMap = MathCNN::conv(&(*input_)[i], &kernel_[j], stride_);
                        tmp.emplace_back(actMap);
                    }
                    convA tmp2 = layerDim_;
                    for (unsigned int j = 0; j < size_[0]; j++) { //width
                        for (unsigned int k = 0; k < size_[1]; k++) { //height
                            for (unsigned int l = 0; l < size_[2]; l++) { //actMaps
                                tmp2[j][k][l] = tmp[l][j][k];
                            }
                        }
                    }
                    z_[i] = tmp2;
                }
            });
        }
        for (unsigned int b = 0; b < tDistribution.size(); b++) {
            kernelThreads[b].join();
        }
    } else {
        for (unsigned int i = 0; i < input->size(); i++) { //batches
            convA tmp;
            tmp.reserve(size[2]);
            for (unsigned int j = 0; j < size[2]; j++) { //actMaps
                featureMap actMap = MathCNN::conv(&(*input)[i], &kernel[j], stride, padding);
                tmp.emplace_back(actMap);
            }
            convA tmp2 = layerDim;
            for (unsigned int j = 0; j < size[0]; j++) { //width
```

```
                for (unsigned int k = 0; k < size[1]; k++) { //height
                    for (unsigned int l = 0; l < size[2]; l++) { //actMaps
                        tmp2[j][k][l] = tmp[l][j][k];
                    }
                }
            }
            z.push_back(tmp2);
        }
    }
    zws = z;
    if (BN) {
        if (!converged) {
            m = std::vector<float>(size[2]);
            var = std::vector<float>(size[2]);
            BatchNorm(&z, &m, &var, delta);
            avgM = MathCNN::addVec(&m, &avgM);
            avgVar = MathCNN::addVec(&var, &avgVar);
        } else {
            BatchNorm(&z, avgM, avgVar, delta);
        }
        u = z;
        MathCNN::retransformBN(&z, &gamma, &beta);
        zws_hat = z;
    } else {
        MathCNN::addActMapwise(&z, &biases);
    }
    if (activations.size() != input->size()) {
        activations = vconvAL();
        activations.reserve(input->size());
        for (unsigned int i = 0; i < input->size(); i++) {
            activations.push_back(layerDim);
        }
    }
    MathCNN::ReLU(&z, &activations);
};
void Conv::backward() {
    if (stride != 1) return;
    vconvAL relu_(zws_hat);
    if (!BN) relu_ = zws;
    MathCNN::ReLU_(&relu_, &relu_);
    vconvAL d_zh, dz;
    d_zh.reserve(input->size());
    for (unsigned int b = 0; b < input->size(); b++) { //batches
        d_zh.push_back(MathCNN::dotProduct(&relu_[b], &((*upGradient)[b])));
    }
    if (BN) {
        std::vector<float> d_beta(size[2]),
                           d_gamma(size[2]);
```

```
vconvAL d_u;
for (unsigned int i = 0; i < input->size(); i++) {
    d_u.push_back(layerDim);
}
for (unsigned int d = 0; d < size[2]; d++) {
    float avg_dB = 0,
          avg_dG = 0;
    for (unsigned int w = 0; w < size[0]; w++) {
        for (unsigned int h = 0; h < size[1]; h++) {
            for (unsigned int b = 0; b < input->size(); b++) {
                avg_dB += d_zh[b][w][h][d];
                avg_dG += d_zh[b][w][h][d] * u[b][w][h][d];
            }
        }
    }
    avg_dB /= input->size();// * size[0] * size[1];
    avg_dG /= input->size();// * size[0] * size[1];
    d_beta[d] = avg_dB;
    d_gamma[d] = avg_dG;
}
for (unsigned int w = 0; w < size[0]; w++) {
    for (unsigned int h = 0; h < size[1]; h++) {
        for (unsigned int d = 0; d < size[2]; d++) {
            for (unsigned int b = 0; b < input->size(); b++) {
                d_u[b][w][h][d] = d_zh[b][w][h][d] * gamma[d];
            }
        }
    }
}
betaGradient = d_beta;
gammaGradient = d_gamma;
std::vector<float> d_var(size[2]);
for (unsigned int d = 0; d < size[2]; d++) {
    for (unsigned int b = 0; b < input->size(); b++) {
        for (unsigned int w = 0; w < size[0]; w++) {
            for (unsigned int h = 0; h < size[1]; h++) {
                d_var[d] += d_u[b][w][h][d] * (m[d] - zws[b][w][h][d]);
            }
        }
    }
    d_var[d] /= 2 * sqrt(pow(var[d] + delta, 3));
}
std::vector<float> d_mean(size[2]);
std::vector<float> tmp1(size[2]), tmp2(size[2]);
int batchS = input->size() * size[0] * size[1];
for (unsigned int d = 0; d < size[2]; d++) {
    for (unsigned int b = 0; b < input->size(); b++) {
        for (unsigned int w = 0; w < size[0]; w++) {
```

```
                    for (unsigned int h = 0; h < size[1]; h++) {
                        tmp1[d] += m[d] - zws[b][w][h][d];
                        tmp2[d] += d_u[b][w][h][d];
                    }
                }
            }
            d_mean[d] = d_var[d] * 2 * tmp1[d] / (float)batchS;
            d_mean[d] -= tmp2[d] / sqrt(var[d] + delta);
        }
        vconvAL d_z;
        d_z.reserve(input->size());
        for (unsigned int d = 0; d < size[2]; d++) {
            for (unsigned int b = 0; b < input->size(); b++) {
                if (d == 0) d_z.push_back(layerDim);
                for (unsigned int w = 0; w < size[0]; w++) {
                    for (unsigned int h = 0; h < size[1]; h++) {
                        d_z[b][w][h][d] = d_mean[d] / (float)batchS;
                        d_z[b][w][h][d] += d_var[d] * 2 * (zws[b][w][h][d] - m[d]) / (float)batc
hS;
                        d_z[b][w][h][d] += d_u[b][w][h][d] / sqrt(var[d] + delta);
                    }
                }
            }
        }
        dz = d_z;
    }
    else {
        dz = d_zh;
        biasesGradient = std::vector<float>();
        biasesGradient.reserve(size[2]);
        for (unsigned int d = 0; d < size[2]; d++) {
            float avgB = 0;
            for (unsigned int b = 0; b < input->size(); b++) {
                for (unsigned int w = 0; w < size[0]; w++) {
                    for (unsigned int h = 0; h < size[1]; h++) {
                        avgB += dz[b][w][h][d];
                    }
                }
            }
            avgB /= (float)input->size();
            biasesGradient.emplace_back(avgB);
        }
    }
    try {
        std::vector<std::vector<int>> tDistribution;
        if (input->size() == 1) {
            tDistribution = { {0, 1} };
        }
```

```
        else {
            int cut = (int)input->size() / 4;
            tDistribution.push_back({ 0, cut });
            tDistribution.push_back({ cut, 2 * cut });
            tDistribution.push_back({ 2 * cut, 3 * cut });
            tDistribution.push_back({ 3 * cut, (int)input->size() });
        }
        kernelGradient = std::vector<convA>();
        kernelGradient.reserve(kernel.size());
        for (unsigned int i = 0; i < kernel.size(); i++) {
            kernelGradient.push_back(convA(kernel[0].size(), std::vector<std::vector<float>>(ker
nel[0][0].size(), std::vector<float>(kernel[0][0][0].size()))));
        }
        std::vector<std::thread> kernelThreads = std::vector<std::thread>(tDistribution.size());
        if (kernelThreads.size() == 0) {
            kernelThreads = std::vector<std::thread>(tDistribution.size());
        }
        int (&size_)[3] = size;
        int (&sl_1_)[3] = sl_1;
        std::vector<convA>& kernel_ = kernel;
        std::vector<convA>& kernelGradient_ = kernelGradient;
        vconvAL& gradient_ = gradient;
        vconvAL& dz_ = dz;
        vconvAL* input_ = input;
        int& padding_ = padding;
        int& stride_ = stride;
        for (unsigned int t = 0; t < tDistribution.size(); t++) {
            kernelThreads[t] = std::thread([&size_, &sl_1_, &kernel_, &kernelGradient_, input_,
&padding_, &stride_, tDistribution, t, &dz_]() {
                for (unsigned int b = tDistribution[t][0]; b < tDistribution[t][1]; b++) {
                    for (unsigned int d = 0; d < size_[2]; d++) { //kernels = depth l
                        for (unsigned int dk = 0; dk < kernel_[0][0][0].size(); dk++) { //kernel
 depth = depth l-1
                            convA kernel2d(size_[0], std::vector<std::vector<float>>(size_[1], s
td::vector<float>(1)));
                            for (unsigned int k1 = 0; k1 < kernel2d.size(); k1++) {
                                for (unsigned int k2 = 0; k2 < kernel2d[k1].size(); k2++) {
                                    kernel2d[k1][k2][0] = dz_[b][k1][k2][d];
                                }
                            }
                            convA actMap(sl_1_[0], std::vector<std::vector<float>>(sl_1_[1], std
::vector<float>(1)));
                            for (unsigned int w = 0; w < actMap.size(); w++) {
                                for (unsigned int h = 0; h < actMap[w].size(); h++) {
                                    actMap[w][h][0] = (*input_)[b][w][h][dk];
                                }
                            }
                            featureMap dw2d;
```

```
                            if (padding_ == 0) {
                                dw2d = MathCNN::conv(&actMap, &kernel2d, stride_);
                            }
                            else {
                                dw2d = MathCNN::conv(&actMap, &kernel2d, stride_, padding_);
                            }
                            for (unsigned int wk = 0; wk < dw2d.size(); wk++) {
                                for (unsigned int hk = 0; hk < dw2d.size(); hk++) {
                                    kernelGradient_[d][wk][hk][dk] += dw2d[wk][hk];
                                }
                            }
                        }
                    }
                }
            });
        }
        convA templatelp1 = convA(sl_1[0], std::vector<std::vector<float>>(sl_1[1], std::vector<
float>(sl_1[2])));
        for (unsigned int i = 0; i < input->size(); i++) {
            gradient.push_back(templatelp1);
        }
        std::vector<std::thread> gradientThreads = std::vector<std::thread>(tDistribution.size()
);
        if (gradientThreads.size() == 0) {
            gradientThreads = std::vector<std::thread>(tDistribution.size());
        }
        for (unsigned int t = 0; t < tDistribution.size(); t++) {
            gradientThreads[t] = std::thread([&size_, &sl_1_, &kernel_, &gradient_, input_, &pad
ding_, &stride_, tDistribution, t, &dz_]() {
                for (unsigned int b = tDistribution[t][0]; b < tDistribution[t][1]; b++) {
                    convA gradientT;
                    for (unsigned int di = 0; di < sl_1_[2]; di++) { //depth input
                        convA kernel_d(kernel_[0].size(), std::vector<std::vector<float>>(kernel
_[0][0].size(), std::vector<float>(size_[2])));
                        for (unsigned int d = 0; d < size_[2]; d++) { //depth
                            for (unsigned int wk = 0; wk < kernel_[0].size(); wk++) {
                                for (unsigned int hk = 0; hk < kernel_[0][wk].size(); hk++) {
                                    kernel_d[wk][hk][d] = kernel_[d][wk][hk][di];
                                }
                            }
                        }
                        convA kernel_d180 = MathCNN::rotate180(&kernel_d);
                        convA* dz2 = const_cast<convA*>(&dz_[b]);
                        featureMap d_aMap = MathCNN::conv(dz2, &kernel_d180, stride_, (int)kerne
l_[0].size() - 1);
                        gradientT.emplace_back(d_aMap);
                    }
                    for (unsigned int w = 0; w < sl_1_[0]; w++) {
```

```
                        for (unsigned int h = 0; h < sl_1_[1]; h++) {
                            for (unsigned int di = 0; di < sl_1_[2]; di++) {
                                gradient_[b][w][h][di] = gradientT[di][w][h];
                            }
                        }
                    }
                }
            });
        }
        for (unsigned int b = 0; b < tDistribution.size(); b++) {
            gradientThreads[b].join();
            kernelThreads[b].join();
        }
        updateParameters();
    } catch (const std::system_error& e) {
        std::cout << "error " << e.code() << ": " << e.what() << std::endl;
        system("pause");
        std::cout << std::endl;
    }
};
void Conv::initialiseParameters(std::string actF) {
    float mean = (float)0;
    float dev = 1;
    if (actF == "ReLU") {
        dev = sqrt((float)2 / (float)(kernelS * kernelS * sl_1[2])); //He initialization
    }
    std::default_random_engine generator;
    for (unsigned int l = 0; l < size[2]; l++) {
        for (unsigned int i = 0; i < kernelS; i++) {
            for (unsigned int j = 0; j < kernelS; j++) {
                for (unsigned int k = 0; k < sl_1[2]; k++) {
                    kernel[l][i][j][k] = MathCNN::getRandomNormal(&mean, &dev, &generator);
                }
            }
        }
    }
    kernelVel1.reserve(kernel.size());
    kernelVel2.reserve(kernel.size());
    for (unsigned int i = 0; i < kernel.size(); i++) {
        kernelVel1.emplace_back(convA(kernel[0].size(), std::vector<std::vector<float>>(kernel[0
][0].size(), std::vector<float>(kernel[0][0][0].size()))));
        kernelVel2.emplace_back(convA(kernel[0].size(), std::vector<std::vector<float>>(kernel[0
][0].size(), std::vector<float>(kernel[0][0][0].size()))));
    }
    avgM = std::vector<float>(size[2]);
    avgVar = std::vector<float>(size[2]);
    if (BN) {
        beta = std::vector<float>(size[2]);
```

```
        betaVel1 = std::vector<float>(size[2]);
        betaVel2 = std::vector<float>(size[2]);
        gamma.reserve(size[0]);
        gamma = std::vector<float>(size[2], 1);
        gammaVel1 = std::vector<float>(size[2]);
        gammaVel2 = std::vector<float>(size[2]);
    }
    else {
        biases = std::vector<float>(size[2], 0);
        biasesVel1 = std::vector<float>(size[2], 0);
        biasesVel2 = std::vector<float>(size[2], 0);
    }
};
void Conv::setUpGradient(vconvAL* upGradient_) {
    upGradient = upGradient_;
};
void Conv::disableBN() {
    BN = false;
}
void Conv::endTraining() {
    converged = true;
    kernelGradient.clear();
    kernelGradient.shrink_to_fit();
    kernelVel1.clear();
    kernelVel1.shrink_to_fit();
    kernelVel2.clear();
    kernelVel2.shrink_to_fit();
    gammaGradient.clear();
    gammaGradient.shrink_to_fit();
    gammaVel1.clear();
    gammaVel1.shrink_to_fit();
    gammaVel2.clear();
    gammaVel2.shrink_to_fit();
    betaGradient.clear();
    betaGradient.shrink_to_fit();
    betaVel1.clear();
    betaVel1.shrink_to_fit();
    betaVel2.clear();
    betaVel2.shrink_to_fit();
    biasesGradient.clear();
    biasesGradient.shrink_to_fit();
    biasesVel1.clear();
    biasesVel1.shrink_to_fit();
    biasesVel2.clear();
    biasesVel2.shrink_to_fit();
    gradient.clear();
    activations.clear();
};
```

```
void Conv::updateParameters() {
    if (BN) {
        for (unsigned int i = 0; i < size[2]; i++) {
            betaVel1[i] = beta_o1 * betaVel1[i] + ((1 - beta_o1) * betaGradient[i]);
            betaVel2[i] = beta_o2 * betaVel2[i] + ((1 - beta_o2) * pow(betaGradient[i], 2));
            gammaVel1[i] = beta_o1 * gammaVel1[i] + ((1 - beta_o1) * gammaGradient[i]);
            gammaVel2[i] = beta_o2 * gammaVel2[i] + ((1 - beta_o2) * pow(gammaGradient[i], 2));
            float betaV1_h = betaVel1[i] / (1 - pow(beta_o1, epoch));
            float betaV2_h = betaVel2[i] / (1 - pow(beta_o2, epoch));
            float gammaV1_h = gammaVel1[i] / (1 - pow(beta_o1, epoch));
            float gammaV2_h = gammaVel2[i] / (1 - pow(beta_o2, epoch));
            beta[i] -= eta * (betaV1_h / sqrt(betaV2_h + 0.00001));
            gamma[i] -= eta * (gammaV1_h / sqrt(gammaV2_h + 0.00001));
        }
    } else {
        for (unsigned int i = 0; i < biases.size(); i++) {
            biasesVel1[i] = beta_o1 * biasesVel1[i] + ((1 - beta_o1) * biasesGradient[i]);
            biasesVel2[i] = beta_o2 * biasesVel2[i] + ((1 - beta_o2) * pow(biasesGradient[i], 2)
);
            float biasesV1_h = biasesVel1[i] / (1 - pow(beta_o1, epoch));
            float biasesV2_h = biasesVel2[i] / (1 - pow(beta_o2, epoch));
            biases[i] -= eta * (biasesV1_h / sqrt(biasesV2_h + 0.00001));
        }
    }
    //kernel
    for (unsigned int i = 0; i < kernel.size(); i++) {
        for (unsigned int j = 0; j < kernel[i].size(); j++) {
            for (unsigned int k = 0; k < kernel[i][j].size(); k++) {
                for (unsigned int l = 0; l < kernel[i][j][k].size(); l++) {
                    kernelVel1[i][j][k][l] = beta_o1 * kernelVel1[i][j][k][l] + ((1 - beta_o1) *
 kernelGradient[i][j][k][l]);
                    kernelVel2[i][j][k][l] = beta_o2 * kernelVel2[i][j][k][l] + ((1 - beta_o2) *
 pow(kernelGradient[i][j][k][l], 2));
                    float kernelV1_h = kernelVel1[i][j][k][l] / (1 - pow(beta_o1, epoch));
                    float kernelV2_h = kernelVel2[i][j][k][l] / (1 - pow(beta_o2, epoch));
                    kernel[i][j][k][l] -= eta * (kernelV1_h / sqrt(kernelV2_h + 0.00001));
                }
            }
        }
    }
    epoch++;
};

std::vector<float> Conv::getAvgMean() {
    std::vector<float> avg = std::vector<float>(avgM);
    for (unsigned int i = 0; i < avgM.size(); i++) {
        avg[i] /= (epoch - 1);
    }
```

```
    return avg;
}

std::vector<float> Conv::getAvgVar() {
    std::vector<float> avg = std::vector<float>(avgVar);
    float m = zws[0].size() * zws[0][0].size() * zws[0][0][0].size();
    for (unsigned int i = 0; i < avgM.size(); i++) {
        avg[i] *= m / ((m - 1) * (epoch - 1));
    }
    return avg;
}
```

Conv.h

```
#ifndef _ConvH_
#define _ConvH_

#include "activations.h"
#include "MathCNN.h"
#include "Layer.h"
#include <thread>

class Conv : public Layer {
private:
    convA layerDim;
    int size[3];
    int sl_1[3];
    int stride, kernelS, padding;
    bool BN = true;
    vconvAL* input;
    vconvAL* upGradient;
    vconvAL zws;
    std::vector<float> m, var;
    int epoch = 1;
    float delta = 0.0001;
    float eta = 0.0001,
          beta_o1 = 0.9,
          beta_o2 = 0.999;
    vconvAL u, zws_hat;
    std::vector<convA> kernelGradient, kernelVel1, kernelVel2;
    std::vector<float> gammaGradient, gammaVel1, gammaVel2;
    std::vector<float> betaGradient, betaVel1, betaVel2;
    std::vector<float> biasesGradient, biasesVel1, biasesVel2;
    void updateParameters();
public:
    bool converged;
    std::vector<convA> kernel;
    std::vector<float> gamma, beta;
    std::vector<float> biases;
    std::vector<float> avgM, avgVar;
    vconvAL activations;
```

```
    vconvAL gradient;
    Conv(int actMaps, vconvAL* input, int kernel, int stride, int padding);
    void forward();
    void backward();
    void initialiseParameters(std::string actF);
    void setUpGradient(vconvAL* upGradient);
    void disableBN();
    void endTraining();
    std::vector<float> getAvgMean();
    std::vector<float> getAvgVar();
};

#endif
```

deepNet.h

```
#ifndef _deepNetH_
#define _deepNetH_
#include "Network.h"
#include "FileManager.h"
#include "MathCNN.h"
#include "Trainer.h"
#endif
```

Dense.cpp

```
#include "Dense.h"
#include "MathCNN.h"
#include <random>
#include <thread>
#include <iostream>

Dense::Dense(int size_, std::vector<denseA>* input_) : size(size_), input(input_) {
    sl_1 = (*input_)[0].size();
    activations = std::vector<denseA>(1, denseA(size)); //denseA(size);
    gradient = std::vector<denseA>(1, denseA(sl_1));        //denseA(sl_1);
    upGradient = nullptr;
    avgM = denseA(size);
    avgVar = denseA(size);
};
void Dense::forward() {
    std::vector<denseA> weightsT = MathCNN::transposeM(&weights);
    std::vector<std::vector<float>> z = MathCNN::MMProduct(input, &weightsT);
    zws = z;

    if (!lastLayer) {
        if (!converged) {
            denseA mean = denseA(zws[0].size()),
                   stddev2 = denseA(zws[0].size());
            BatchNorm(&z, &mean, &stddev2, delta);
            m = mean;
            var = stddev2;
            avgM = MathCNN::addVec(&avgM, &m);
```

```
            avgVar = MathCNN::addVec(&avgVar, &var);
        } else {
            BatchNorm(&z, avgM, avgVar, delta);
        }
        u = z;
        z = MathCNN::transposeM(&z);
        MathCNN::MVmultiplyColumn(&z, &gamma);
        MathCNN::MVaddColumn(&z, &beta);
        zws_hat = MathCNN::transposeM(&z);
    } else {
        z = MathCNN::transposeM(&z);
        MathCNN::MVaddColumn(&z, &beta);
    }
    activations = MathCNN::transposeM(&z);
    if(!lastLayer) MathCNN::ReLU(&activations);
};
void Dense::backward() {
    std::vector<denseA> d_zh = *upGradient;
    std::vector<denseA> dz;
    dz.reserve(d_zh.size());
    if (!lastLayer) {
        for (unsigned int i = 0; i < d_zh.size(); i++) {
            denseA z_h = zws_hat[i];
            MathCNN::ReLU_(&z_h);
            d_zh[i] = MathCNN::dotProduct(&z_h, &((*upGradient)[i]));
        }
        std::vector<float> d_beta;
        std::vector<float> d_gamma;
        std::vector<denseA> d_u;
        d_u.reserve(u.size());
        d_u = std::vector<denseA>(u.size(), denseA(u[0].size()));
        d_beta.reserve(d_zh[0].size());
        d_gamma.reserve(d_zh[0].size());
        denseA d_var = denseA(d_zh[0].size());
        for (unsigned int j = 0; j < d_zh[0].size(); j++) { //this Layer
            float avg_dB = 0,
                  avg_dG = 0;
            for (unsigned int i = 0; i < d_zh.size(); i++) { //batches
                avg_dB += d_zh[i][j];
                avg_dG += d_zh[i][j] * u[i][j];
                d_u[i][j] = d_zh[i][j] * gamma[j];
                d_var[j] += d_u[i][j] * (m[j] - zws[i][j]);
            }
            d_var[j] /= 2 * sqrt(pow(var[j] + delta, 3));
            avg_dB /= (float) d_zh.size();
            avg_dG /= (float) d_zh.size();
            d_beta.emplace_back(avg_dB);
            d_gamma.emplace_back(avg_dG);
```

```
        }
        int batchS = zws.size();
        betaGradients = d_beta;
        gammaGradients = d_gamma;
        denseA d_mean = denseA(zws[0].size());
        for (unsigned int j = 0; j < zws[0].size(); j++) { //this Layer
            float tmp1 = 0, tmp2 = 0;
            for (unsigned int i = 0; i < zws.size(); i++) { //batches
                tmp1 += m[j] - zws[i][j];
                tmp2 += d_u[i][j];
            }
            d_mean[j] = d_var[j] * 2 * tmp1 / ((float) batchS);
            d_mean[j] -= tmp2 / sqrt(var[j] + delta);
        }
        std::vector<denseA> d_z(zws.size(), denseA(zws[0].size()));
        for (unsigned int i = 0; i < d_z.size(); i++) { //batches
            for (unsigned int j = 0; j < d_z[i].size(); j++) { //this Layer
                d_z[i][j] = d_mean[j] / ((float) batchS);
                d_z[i][j] += d_var[j] * 2 * (zws[i][j] - m[j]) / ((float) batchS);
                d_z[i][j] += d_u[i][j] / sqrt(var[j] + delta);
            }
        }
        dz = d_z;
    } else {
        dz = d_zh;
        betaGradients = std::vector<float>();
        betaGradients.reserve(dz[0].size());
        for (unsigned int i = 0; i < dz[0].size(); i++) {
            float avgB = 0;
            for (unsigned int j = 0; j < dz.size(); j++) {
                avgB += dz[j][i];
            }
            avgB /= (float) dz.size();
            betaGradients.emplace_back(avgB);
        }
    }
    weightGradients = denseW();
    weightGradients.reserve(weights.size());
    weightGradients = denseW(weights.size(), std::vector<float>(weights[0].size()));
    denseW& weights_ = weights;
    denseW& weightGradients_ = weightGradients;
    std::vector<denseA>* input_ = input;
    std::vector<denseA>& zws_ = zws;
    std::vector<denseA>& dz_ = dz;
    std::thread t = std::thread([&weights_, &weightGradients_, &zws_, &dz_, input_]() {
        for (unsigned int i = 0; i < weights_.size(); i++) {
            for (unsigned int j = 0; j < weights_[i].size(); j++) {
                for (unsigned int k = 0; k < zws_.size(); k++) {
```

```
                    weightGradients_[i][j] += dz_[k][i] * (*input_)[k][j];
                }
                weightGradients_[i][j] /= zws_.size();
            }
        }
    });
    if (gradient.size() != input->size()) {
        gradient.reserve(input->size());
        gradient = std::vector<denseA>(input->size(), std::vector<float>((*input)[0].size()));
    }
    for (unsigned int i = 0; i < gradient.size(); i++) { //i=batches
        for (unsigned int j = 0; j < gradient[i].size(); j++) { //j=prevLayer
            gradient[i][j] = 0;
            for (unsigned int k = 0; k < zws[0].size(); k++) { //k=thisLayer
                gradient[i][j] += weights[k][j] * dz[i][k];
            }
        }
    }
    t.join();
    updateParameters();
};
void Dense::initialiseParameters(std::string act) {
    float mean = (float) 0;
    float dev = 1;
    if (act == "ReLU") {
        dev = sqrt((float)2 / (float)sl_1); //He initialization
    }
    std::default_random_engine generator;
    weights.reserve(size);
    weightVelocity1.reserve(size);
    weightVelocity2.reserve(size);
    for (int i = 0; i < size; i++) {
        weights.emplace_back(std::vector<float>());
        weightVelocity1.emplace_back(std::vector<float>());
        weightVelocity2.emplace_back(std::vector<float>());
        weights[i].reserve(sl_1);
        weightVelocity1[i].reserve(sl_1);
        weightVelocity2[i].reserve(sl_1);
        for (int j = 0; j < sl_1; j++) {
            weights[i].emplace_back(MathCNN::getRandomNormal(&mean, &dev, &generator));
            weightVelocity1[i].emplace_back(0);
            weightVelocity2[i].emplace_back(0);
        }
    }
    beta.reserve(size);
    gamma.reserve(size);
    for (int i = 0; i < size; i++) {
        beta.emplace_back((float)0);
```

```
        gamma.emplace_back((float)1);
    }
    gammaVelocity1.reserve(size);
    gammaVelocity2.reserve(size);
    betaVelocity1.reserve(size);
    betaVelocity2.reserve(size);
    gammaVelocity1 = std::vector<float>(gamma.size());
    gammaVelocity2 = std::vector<float>(gamma.size());
    betaVelocity1 = std::vector<float>(beta.size());
    betaVelocity2 = std::vector<float>(beta.size());
};
void Dense::setUpGradient(std::vector<denseA>* upGradient_) {
    upGradient = upGradient_;
};
void Dense::updateParameters() {
    if (!lastLayer) {
        for (unsigned int i = 0; i < beta.size(); i++) {
            betaVelocity1[i] = beta_o1 * betaVelocity1[i] + ((1 - beta_o1) * betaGradients[i]);
            betaVelocity2[i] = beta_o2 * betaVelocity2[i] + ((1 - beta_o2) * pow(betaGradients[i
], 2));
            gammaVelocity1[i] = beta_o1 * gammaVelocity1[i] + ((1 - beta_o1) * gammaGradients[i]
);
            gammaVelocity2[i] = beta_o2 * gammaVelocity2[i] + ((1 - beta_o2) * pow(gammaGradient
s[i], 2));
        }
        for (unsigned int i = 0; i < beta.size(); i++) {
            float betaV1_h = betaVelocity1[i] / (1 - pow(beta_o1, epoch));
            float betaV2_h = betaVelocity2[i] / (1 - pow(beta_o2, epoch));
            float gammaV1_h = gammaVelocity1[i] / (1 - pow(beta_o1, epoch));
            float gammaV2_h = gammaVelocity2[i] / (1 - pow(beta_o2, epoch));
            beta[i] -= eta * (betaV1_h / sqrt(betaV2_h + 0.00001));
            gamma[i] -= eta * (gammaV1_h / sqrt(gammaV2_h + 0.00001));
        }
    } else { //lastLayer
        for (unsigned int i = 0; i < beta.size(); i++) {
            betaVelocity1[i] = beta_o1 * betaVelocity1[i] + ((1 - beta_o1) * betaGradients[i]);
            betaVelocity2[i] = beta_o2 * betaVelocity2[i] + ((1 - beta_o2) * pow(betaGradients[i
], 2));
        }
        for (unsigned int i = 0; i < beta.size(); i++) {
            float betaV1_h = betaVelocity1[i] / (1 - pow(beta_o1, epoch));
            float betaV2_h = betaVelocity2[i] / (1 - pow(beta_o2, epoch));
            beta[i] -= eta * (betaV1_h / sqrt(betaV2_h + 0.00001));
        }
    }
    for (unsigned int i = 0; i < weights.size(); i++) {
        for (unsigned int j = 0; j < weights[i].size(); j++) {
```

```
            weightVelocity1[i][j] = beta_o1 * weightVelocity1[i][j] + ((1 - beta_o1) * weightGra
dients[i][j]);
            weightVelocity2[i][j] = beta_o2 * weightVelocity2[i][j] + ((1 - beta_o2) * pow(weigh
tGradients[i][j], 2));
        }
    }
    for (unsigned int i = 0; i < weights.size(); i++) {
        for (unsigned int j = 0; j < weights[i].size(); j++) {
            float weightV1_h = weightVelocity1[i][j] / (1 - pow(beta_o1, epoch));
            float weightV2_h = weightVelocity2[i][j] / (1 - pow(beta_o2, epoch));
            weights[i][j] -= eta * (weightV1_h / sqrt(weightV2_h + 0.00001));
        }
    }
    epoch++;
}
void Dense::endTraining() {
    converged = true;
    betaGradients.clear();
    betaGradients.shrink_to_fit();
    betaVelocity1.clear();
    betaVelocity1.shrink_to_fit();
    betaVelocity2.clear();
    betaVelocity2.shrink_to_fit();
    gammaGradients.clear();
    gammaGradients.shrink_to_fit();
    gammaVelocity1.clear();
    gammaVelocity1.shrink_to_fit();
    gammaVelocity2.clear();
    gammaVelocity2.shrink_to_fit();
    weightGradients.clear();
    weightGradients.shrink_to_fit();
    weightVelocity1.clear();
    weightVelocity1.shrink_to_fit();
    weightVelocity2.clear();
    weightVelocity2.shrink_to_fit();
    gradient.clear();
    gradient.shrink_to_fit();
    activations.clear();
    activations.shrink_to_fit();
}

denseA Dense::getAvgMean() {
    denseA avg = denseA(avgM);
    for (unsigned int i = 0; i < zws[0].size(); i++) {
        avg[i] /= (epoch - 1);
    }
    return avg;
}
```

```
denseA Dense::getAvgVar() {
    float m = zws.size();
    denseA avg = denseA(avgVar);
    for (unsigned int i = 0; i < zws[0].size(); i++) {
        avg[i] *= m / ((m - 1) * (epoch - 1));
    }
    return avg;
}
```

Dense.h

```
#ifndef _DenseH_
#define _DenseH_

#include "Layer.h"
#include <string>
#include <thread>

class Dense : public Layer {
private:
    int size, sl_1;
    std::vector<denseA>* input;
    std::vector<denseA>* upGradient;

    std::vector<std::vector<float>> zws;
    std::vector<denseA> u;
    std::vector<denseA> zws_hat;
    denseA m, var;
    float delta = 0.0001;

    denseA betaGradients;
    denseA betaVelocity1;
    denseA betaVelocity2;
    denseA gammaGradients;
    denseA gammaVelocity1;
    denseA gammaVelocity2;
    denseW weightGradients;
    denseW weightVelocity1;
    denseW weightVelocity2;

    void updateParameters();
    float eta = 0.0001,
          beta_o1 = 0.9,
          beta_o2 = 0.999;
    int epoch = 1;
public:
    bool converged = false;
    denseA beta;
    denseA gamma;
    denseW weights;
```

```
    denseA avgM,
           avgVar;
    bool lastLayer = false;
    std::vector<denseA> activations;
    std::vector<denseA> gradient;
    Dense(int size, std::vector<denseA>* input);
    void forward() ;
    void backward();
    void initialiseParameters(std::string actF);
    void setUpGradient(std::vector<denseA>* upGradient);
    void endTraining();
    denseA getAvgMean();
    denseA getAvgVar();
};

#endif;
```

FileManager.cpp

```
#include "FileManager.h"
#include <iostream>

typedef std::vector<float> denseA;
typedef std::vector<std::vector<std::vector<float>>> convA;
typedef std::vector<std::vector<float>> denseW;
typedef std::vector<std::vector<float>> featureMap;

extern "C" {
    #define STB_IMAGE_IMPLEMENTATION
    #include "stb_image.h"
}

convA FileManager::getInput(std::string filename) {
    std::vector<std::vector<std::vector<float>>> val;
    int height, width, channels;
    unsigned char* data = stbi_load(filename.c_str(), &width, &height, &channels, 3);
    for (int ch = 0; ch < channels; ch++) {
        int dataIx = ch;
        std::vector<float> chVals;
        for (int i = 0; i < height * width; i++) {
            chVals.push_back((float)static_cast<int>(data[dataIx]) / (float)255);
            dataIx += channels;
        }
        std::vector<std::vector<float>> tmp;
        for (int y = 0; y < height; y++) {
            tmp.push_back(std::vector<float>());
            for (int x = 0; x < width; x++) {
                int ix = y * width + x;
                tmp[y].push_back(chVals[ix]);
            }
```

```
        }
        val.push_back(tmp);
    }
    std::vector<std::vector<std::vector<float>>> val2(width, std::vector<std::vector<float>>(hei
ght, std::vector<float>(channels)));
    stbi_image_free(data);
    for (unsigned int i = 0; i < val.size(); i++) {
        for (unsigned int j = 0; j < val[i].size(); j++) {
            for (unsigned int k = 0; k < val[i][j].size(); k++) {
                val2[k][j][i] = val[i][j][k];
            }
        }
    }
    return val2;
}
convA FileManager::grayscale(convA* image) {
    if ((*image)[0][0].size() == 1) {
        return *image;
    }
    std::vector<std::vector<std::vector<float>>> result(image-
>size(), std::vector<std::vector<float>>((*image)[0].size(), std::vector<float>(1)));
    for (unsigned int i = 0; i < image->size(); i++) {
        for (unsigned int j = 0; j < (*image)[i].size(); j++) {
            result[i][j][0] = (*image)[i][j][0];
        }
    }
    return result;
};
convA FileManager::invert(convA* image) {
    std::vector<std::vector<std::vector<float>>> result(image-
>size(), std::vector<std::vector<float>>((*image)[0].size(), std::vector<float>((*image)[0][0].s
ize())));
    for (unsigned int i = 0; i < image->size(); i++) {
        for (unsigned int j = 0; j < (*image)[i].size(); j++) {
            for (unsigned int k = 0; k < (*image)[i][j].size(); k++) {
                result[i][j][k] = 1 - (*image)[i][j][k];
            }
        }
    }
    return result;
};
```

FileManager.h

```
#ifndef _FileManagerH_
#define _FileManagerH_

#include "activations.h"
#include <string>
```

```
class FileManager {
public:
    static convA getInput(std::string imgPath);
    static convA grayscale(convA* image);
    static convA invert(convA* image);
};

#endif
```

Flatten.cpp

```
#include "Flatten.h"
Flatten::Flatten(vconvAL* input_) : input(input_) {
    sl_1[0] = (*input)[0].size();
    sl_1[1] = (*input)[0][0].size();
    sl_1[2] = (*input)[0][0][0].size();
    size = sl_1[0] * sl_1[1] * sl_1[2];
    activations = std::vector<denseA>(1, denseA(size));
    gradient = vconvAL();
    upGradient = nullptr;
};
void Flatten::forward() {
    activations = std::vector<denseA>(0);
    activations.reserve(input->size());
    for (unsigned int i = 0; i < input->size(); i++) {
        activations.push_back(denseA(0));
        for (unsigned int j = 0; j < (*input)[i].size(); j++) {
            for (unsigned int k = 0; k < (*input)[i][j].size(); k++) {
                for (unsigned int l = 0; l < (*input)[i][j][k].size(); l++) {
                    activations[i].push_back((*input)[i][j][k][l]);
                }
            }
        }
    }
};
void Flatten::backward() {
    gradient = *input;
    for (unsigned int i = 0; i < upGradient->size(); i++) {
        for (unsigned int j = 0; j < (*upGradient)[i].size(); j++) {
            int I = ixs[j][0],
                J = ixs[j][1],
                K = ixs[j][2];
            gradient[i][I][J][K] = (*upGradient)[i][j];
        }
    }
};
void Flatten::initialiseParameters(std::string actF) {
    for (unsigned int j = 0; j < sl_1[0]; j++) {
        for (unsigned int k = 0; k < sl_1[1]; k++) {
            for (unsigned int l = 0; l < sl_1[2]; l++) {
```

```
                ixs.push_back(std::vector<unsigned int>({ j, k, l }));
            }
        }
    }
};
void Flatten::setUpGradient(std::vector<denseA>* upGradient_) {
    upGradient = upGradient_;
};
void Flatten::endTraining() {
    activations.clear();
    activations.shrink_to_fit();
};
```

Flatten.h

```
#ifndef _FlattenH_
#define _FlattenH_

#include "Layer.h"
#include "Conv.h"

class Flatten : public Layer {
private:
    int size;
    int sl_1[3];
    vconvAL* input;
    std::vector<denseA>* upGradient;
    std::vector<std::vector<unsigned int>> ixs;
public:
    std::vector<denseA> activations;
    vconvAL gradient;
    Flatten(vconvAL* input);
    void forward();
    void backward();
    void initialiseParameters(std::string actF);
    void setUpGradient(std::vector<denseA>* upGradient);
    void endTraining();
};

#endif
```

Input.cpp

```
#include "Input.h"

Input::Input(int* size_) : size(size_) {
    activations.push_back(convAL());
    for (unsigned int i = 0; i < size[0]; i++) {
        activations[0].push_back(vectorff());
        for (unsigned int j = 0; j < size[1]; j++) {
            activations[0][i].push_back(vectorf());
            for (unsigned int k = 0; k < size[2]; k++) {
```

```
                activations[0][i][j].push_back(0);
            }
        }
    }
};
void Input::forward() {
};
void Input::backward() {};
void Input::initialiseParameters(std::string actF) {};
```

Input.h

```
#ifndef _InputH_
#define _InputH_

#include "Layer.h"
#include "activations.h"

class Input : public Layer {
private:
    int *size;
public:
    vconvAL activations;
    Input(int *size);
    void forward();
    void backward();
    void initialiseParameters(std::string actF);
};

#endif
```

jExporter.cpp

```
#include "jExporter.h"

#include "nlohmann.hpp"
#include <filesystem>
#include <fstream>
#include <iostream>

using json = nlohmann::json;

typedef std::vector<float> denseA;
typedef std::vector<std::vector<std::vector<float>>> convA;
typedef std::vector<std::vector<float>> denseW;
typedef std::vector<std::vector<float>> featureMap;

void jExporter::exportData(std::string filename, denseExp layer) {
    std::ofstream out;
    out.open(filename, std::ios_base::app);
    json layerJ = json({});
```

```
    layerJ["weights"] = json(*layer.weights);
    layerJ["beta"] = json(*layer.beta);
    layerJ["gamma"] = json(*layer.gamma);
    layerJ["mean"] = json(*layer.avgM);
    layerJ["variance"] = json(*layer.avgVar);
    out << layerJ;
    out.close();
    out.open(filename, std::ios_base::app);
    out << "\n";
    out.close();
};
void jExporter::exportData(std::string filename, convExp layer) {
    std::ofstream out;
    out.open(filename, std::ios_base::app);
    json layerJ = json({});
    layerJ["kernel"] = json(*(layer.kernel));
    layerJ["beta"] = json(*layer.beta);
    layerJ["gamma"] = json(*layer.gamma);
    layerJ["bias"] = json(*layer.biases);
    layerJ["mean"] = json(*layer.avgM);
    layerJ["variance"] = json(*layer.avgVar);
    out << layerJ;
    out.close();
    out.open(filename, std::ios_base::app);
    out << "\n";
    out.close();
};
void jExporter::exportData(std::string filename, inputExp layer) {
    std::ofstream out;
    out.open(filename, std::ios_base::app);
    json layerJ = json({});
    layerJ["batchMean"] = json(*layer.batchMean);
    out << layerJ;
    out.close();
    out.open(filename, std::ios_base::app);
    out << "\n";
    out.close();
};
void jExporter::exportData(std::string filename, multiDenseExp layer) {
    std::ofstream out;
    out.open(filename, std::ios_base::app);
    json layerJ = json({});
    layerJ["avgM"] = json(*layer.avgM);
    layerJ["avgVar"] = json(*layer.avgVar);
    layerJ["betaM"] = json(*layer.betaM);
    layerJ["alphaM"] = json(*(layer.alphaM));
    layerJ["beta"] = json(*layer.beta);
    layerJ["gamma"] = json(*layer.gamma);
```

```
    layerJ["position"] = json(*layer.position);
    out << layerJ;
    out.close();
    out.open(filename, std::ios_base::app);
    out << "\n";
    out.close();
};
void jExporter::exportData(std::string filename, multiDenseInputExp layer) {
    std::ofstream out;
    out.open(filename, std::ios_base::app);
    json layerJ = json({});
    layerJ["position"] = json(*layer.position);
    out << layerJ;
    out.close();
    out.open(filename, std::ios_base::app);
    out << "\n";
    out.close();
};


void jExporter::importData(int i, std::string filename, denseW* weights, denseA* beta, denseA* g
amma, denseA* avgM, denseA* avgVar) {
    std::ifstream inFile;
    inFile.open(filename);
    std::string data;
    int line = 1;
    while (!inFile.eof()) {
        std::string tmp;
        getline(inFile, tmp);
        if (line == i + 1) {
            data = tmp;
            break;
        }
        line++;
    }
    inFile.close();
    json layerJ = json::parse(data);
    *weights = layerJ["weights"].get<denseW>();
    *beta = layerJ["beta"].get<denseA>();
    *gamma = layerJ["gamma"].get<denseA>();
    *avgM = layerJ["mean"].get<denseA>();
    *avgVar = layerJ["variance"].get<denseA>();
}; //dense
void jExporter::importData(int i, std::string filename, std::vector<j::convA>* kernel, j::denseA
* beta, denseA* gamma, j::denseA* biases, j::denseA* avgM, j::denseA* avgVar) {
    std::ifstream inFile;
    inFile.open(filename);
    std::string data;
    int line = 1;
```

```
    while (!inFile.eof()) {
        std::string tmp;
        getline(inFile, tmp);
        if (line == i + 1) {
            data = tmp;
            break;
        }
        line++;
    }
    inFile.close();
    json layerJ = json::parse(data);
    *kernel = layerJ["kernel"].get<std::vector<convA>>();
    *beta = layerJ["beta"].get<std::vector<float>>();
    *gamma = layerJ["gamma"].get<std::vector<float>>();
    *biases = layerJ["bias"].get<std::vector<float>>();
    *avgM = layerJ["mean"].get<std::vector<float>>();
    *avgVar = layerJ["variance"].get<std::vector<float>>();

}; //conv
void jExporter::importData(int i, std::string filename, float* batchMean) {
    std::ifstream inFile;
    inFile.open(filename);
    std::string data;
    int line = 1;
    while (!inFile.eof()) {
        std::string tmp;
        getline(inFile, tmp);
        if (line == i + 1) {
            data = tmp;
            break;
        }
        line++;
    }
    inFile.close();
    json layerJ = json::parse(data);
    *batchMean = layerJ["batchMean"].get<float>();
}; //input
void jExporter::importData(int i, std::string filename, denseA* avgM, denseA* avgVar, denseA* be
taM, denseA* alphaM, denseA* beta, denseA* gamma, denseW* position) {
    std::ifstream inFile;
    inFile.open(filename);
    std::string data;
    int line = 1;
    while (!inFile.eof()) {
        std::string tmp;
        getline(inFile, tmp);
        if (line == i + 1) {
            data = tmp;
```

```
            break;
        }
        line++;
    }
    inFile.close();
    json layerJ = json::parse(data);
    *avgM = layerJ["avgM"].get<denseA>();
    *avgVar = layerJ["avgVar"].get<denseA>();
    *betaM = layerJ["betaM"].get<denseA>();
    *alphaM = layerJ["alphaM"].get<denseA>();
    *beta = layerJ["beta"].get<denseA>();
    *gamma = layerJ["gamma"].get<denseA>();
    *position = layerJ["position"].get<std::vector<std::vector<float>>>();

}; //multiDense
void jExporter::importData(int i, std::string filename, denseW* position) {
    std::ifstream inFile;
    inFile.open(filename);
    std::string data;
    int line = 1;
    while (!inFile.eof()) {
        std::string tmp;
        getline(inFile, tmp);
        if (line == i + 1) {
            data = tmp;
            break;
        }
        line++;
    }
    inFile.close();
    json layerJ = json::parse(data);
}; //multiDenseInput
```

jExporter.h

```
#ifndef _jExporterH_
#define _jExporterH_

#include <vector>
#include <string>

namespace j {
    typedef std::vector<float> denseA;
    typedef std::vector<std::vector<std::vector<float>>> convA;
    typedef std::vector<std::vector<float>> denseW;
    typedef std::vector<std::vector<float>> featureMap;
}

struct denseExp {
    j::denseW* weights;
```

```
    j::denseA* beta, *gamma;
    j::denseA* avgM, *avgVar;
};

struct convExp {
    std::vector<j::convA>* kernel;
    std::vector<float>* beta, *gamma;
    std::vector<float>* biases;
    j::denseA* avgM, *avgVar;
};

struct inputExp {
    float* batchMean;
};

struct multiDenseExp {
    j::denseA* avgM, *avgVar;
    j::denseA* betaM, *alphaM;
    j::denseA* beta, *gamma;
    std::vector<std::vector<float>>* position;
};

struct multiDenseInputExp {
    std::vector<std::vector<float>>* position;
};

class jExporter {
public:
    static void exportData(std::string filename, denseExp layer);
    static void exportData(std::string filename, convExp layer);
    static void exportData(std::string filename, inputExp layer);
    static void exportData(std::string filename, multiDenseExp layer);
    static void exportData(std::string filename, multiDenseInputExp layer);

    static void importData(int i, std::string filename, j::denseW* weights, j::denseA* beta, j::
denseA* gamma, j::denseA* avgM, j::denseA* avgVar); //dense
    static void importData(int i, std::string filename, std::vector<j::convA>* kernel, j::denseA
* beta, j::denseA* gamma, j::denseA* biases, j::denseA* avgM, j::denseA* avgVar); //conv
    static void importData(int i, std::string filename, float* batchMean); //input
    static void importData(int i, std::string filename, j::denseA* avgM, j::denseA* avgVar, j::d
enseA* betaM, j::denseA* alphaM, j::denseA* beta, j::denseA* gamma, j::denseW* position); //mult
iDense
    static void importData(int i, std::string filename, j::denseW* position); //multiDenseInput
filename, float* batchMean, std::vector< std::vector<Layer*>>* level);
};

#endif
```

Layer.cpp

```
#include "Layer.h"
#include <iostream>

void Layer::BatchNorm(std::vector<std::vector<float>>* z, denseA* mean, denseA* stddev, float de
lta) {
    for (unsigned int i = 0; i < z->size(); i++) {
        for (unsigned int j = 0; j < (*z)[i].size(); j++) {
            (*mean)[j] += (*z)[i][j];
        }
    }
    for (unsigned int j = 0; j < (*z)[0].size(); j++) {
        (*mean)[j] /= z->size();
    }
    for (unsigned int i = 0; i < z->size(); i++) {
        for (unsigned int j = 0; j < (*z)[i].size(); j++) {
            (*stddev)[j] += (float)pow((*z)[i][j] - (*mean)[j], 2);
        }
    }
    for (unsigned int j = 0; j < (*z)[0].size(); j++) {
        (*stddev)[j] /= z->size();
    }
    for (unsigned int i = 0; i < z->size(); i++) {
        for (unsigned int j = 0; j < (*z)[i].size(); j++) {
            (*z)[i][j] -= (*mean)[j];
            (*z)[i][j] /= sqrt((*stddev)[j] + delta);
        }
    }
};

void Layer::BatchNorm(std::vector<std::vector<float>>* z, denseA mean, denseA var, float delta)
{
    for (unsigned int i = 0; i < z->size(); i++) {
        for (unsigned int j = 0; j < (*z)[i].size(); j++) {
            (*z)[i][j] -= mean[j];
            (*z)[i][j] /= sqrt(var[j] + delta);
        }
    }
};

void Layer::BatchNorm(vconvAL* z, std::vector<float>* mean, std::vector<float>* stddev2, float d
elta) {
    for (unsigned int i = 0; i < z->size(); i++) {
        for (unsigned int j = 0; j < (*z)[i].size(); j++) {
            for (unsigned int k = 0; k < (*z)[i][j].size(); k++) {
                for (unsigned int l = 0; l < (*z)[i][j][k].size(); l++) {
                    (*mean)[l] += (*z)[i][j][k][l];
                }
            }
```

```
        }
    }
    int bxwxh = z->size() * (*z)[0].size() * (*z)[0][0].size();
    for (unsigned int l = 0; l < (*z)[0][0][0].size(); l++) {
        (*mean)[l] /= (float)bxwxh;
    }
    for (unsigned int i = 0; i < z->size(); i++) {
        for (unsigned int j = 0; j < (*z)[i].size(); j++) {
            for (unsigned int k = 0; k < (*z)[i][j].size(); k++) {
                for (unsigned int l = 0; l < (*z)[i][j][k].size(); l++) {
                    (*stddev2)[l] += (float) pow((*z)[i][j][k][l] - (*mean)[l], 2);
                }
            }
        }
    }
    for (unsigned int l = 0; l < (*z)[0][0][0].size(); l++) {
        (*stddev2)[l] /= (float)bxwxh;
    }
    for (unsigned int i = 0; i < z->size(); i++) {
        for (unsigned int j = 0; j < (*z)[i].size(); j++) {
            for (unsigned int k = 0; k < (*z)[i][j].size(); k++) {
                for (unsigned int l = 0; l < (*z)[i][j][k].size(); l++) {
                    (*z)[i][j][k][l] -= (*mean)[l];
                    (*z)[i][j][k][l] /= sqrt((*stddev2)[l] + delta);
                }
            }
        }
    }
};

void Layer::BatchNorm(vconvAL* z, std::vector<float> mean, std::vector<float> var, float delta)
{
   for (unsigned int i = 0; i < z->size(); i++) {
        for (unsigned int j = 0; j < (*z)[i].size(); j++) {
            for (unsigned int k = 0; k < (*z)[i][j].size(); k++) {
                for (unsigned int l = 0; l < (*z)[i][j][k].size(); l++) {
                    (*z)[i][j][k][l] -= mean[l];
                    (*z)[i][j][k][l] /= sqrt(var[l] + delta);
                }
            }
        }
    }
};
```

Layer.h

```
#ifndef _LayerH_
#define _LayerH_

#include "activations.h"
```

```
#include <string>
#include <vector>

class Layer {
public:
    void BatchNorm(std::vector<std::vector<float>>* z, denseA* mean, denseA* stddev, float delta
);
    void BatchNorm(std::vector<std::vector<float>>* z, denseA mean, denseA var, float delta);
    void BatchNorm(vconvAL* z, std::vector<float>* mean, std::vector<float>* stddev, float delta
);
    void BatchNorm(vconvAL* z, std::vector<float> mean, std::vector<float> var, float delta);
    virtual void forward() = 0;
    virtual void backward() = 0;
    virtual void initialiseParameters(std::string actF) = 0;
};

#endif
```

MathCNN.cpp

```
#include "MathCNN.h"
#include <random>
#include <vector>
#include <algorithm>
#include <cctype>
#include <string>
#include <regex>
#include <time.h>;

MathCNN::MathCNN() {};
std::vector<float> MathCNN::MVProduct(const std::vector<std::vector<float>>* matrix, const std::
vector<float>* vec) { //matrix: row<column>
    std::vector<float> result;
    for (unsigned int i = 0; i < (*matrix).size(); i++) {
        float sum = 0;
        int j = 0;
        for (auto it2 = vec->begin(); it2 != vec->end(); it2++) {
            sum += ((*matrix)[i][j]) * (*it2);
            j++;
        }
        result.push_back(sum);
    }
    return result;
};
std::vector<float> MathCNN::addVec(std::vector<float>* vec1, std::vector<float>* vec2) {
    std::vector<float> result;
    if (vec1->size() != vec2->size()) return result;
    for (unsigned int i = 0; i < vec1->size(); i++) {
        result.push_back((*vec1)[i] + (*vec2)[i]);
    }
```

```
    return result;
};
float MathCNN::getRandomNormal(const float* mean, const float* dev, std::default_random_engine*
generator) {
    std::normal_distribution<float> distribution(*mean, *dev);
    return distribution(*generator);
}
float MathCNN::ReLU(float z) {
    if (z > 0) return z;
    return 0;
}
void MathCNN::ReLU(float* z) {
    if (*z < 0) *z = 0;
}
void MathCNN::ReLU(std::vector<float>* z) {
    for (unsigned int i = 0; i < z->size(); i++) {
        if ((*z)[i] < 0) (*z)[i] = 0;
    }
}
void MathCNN::ReLU_(std::vector<float>* z) {
    for (unsigned int i = 0; i < z->size(); i++) {
        if ((*z)[i] < 0) {
            (*z)[i] = 0;
        }
        else {
            (*z)[i] = 1;
        }
    }
}
void MathCNN::softmax(const std::vector<float>* in, std::vector<float>* out) {
    float inputSum = 0;
    for (unsigned int i = 0; i < in->size(); i++) {
        inputSum += exp((*in)[i]);
    }
    for (unsigned int i = 0; i < out->size(); i++) {
        (*out)[i] = exp((*in)[i]) / inputSum;
    }
};

void MathCNN::dotProduct(const std::vector<float>* in1, const std::vector<float>* in2, std::vect
or<float>* result) {
    for (unsigned int i = 0; i < in1->size(); i++) {
        (*result)[i] = (*in1)[i] * (*in2)[i];
    }
};
std::vector<std::vector<float>> MathCNN::transposeM(const std::vector<std::vector<float>>* matri
x) {
```

```
    std::vector<std::vector<float>> transposed((*(matrix))[0].size(), std::vector<float>(matrix-
>size()));
    for (unsigned int i = 0; i < matrix->size(); i++) {
        for (unsigned int j = 0; j < (*(matrix))[i].size(); j++) {
            transposed[j][i] = (*(matrix))[i][j];
        }
    }
    return transposed;
};
int MathCNN::position(std::string letter) {
    std::vector<int> charIx(alphabet.size());
    for (unsigned int i = 0; i < charIx.size(); i++) {
        charIx[i] = i + 1;
    }
    auto it = find(alphabet.begin(), alphabet.end(), letter);
    return it - alphabet.begin();
}
std::string MathCNN::encode(std::string input) {
    reverse(input.begin(), input.end());
    std::vector<std::string> split;
    std::string exception = "";
    if (input.size() % 2 != 0) {
        std::string exception = "x";
        input += "0";
    }
    for (unsigned i = 0; i < input.size(); i+=2) {
        std::string tmp = std::string(1, input[i]) + std::string(1, input[i+1]);
        split.push_back(tmp);
    }
    std::string result = "";
    for (unsigned int i = 0; i < split.size(); i++) {
        int number = stoi(split[i]);
        int letterIx = number % 26;
        std::string digit = std::to_string((int) stoi(split[i])/26);
        result += alphabet[letterIx] + digit;
    }
    result += exception;
    return result;
}

std::string MathCNN::decode(std::string input) {
    std::vector<std::string> split;
    std::string result = "";
    int size = input.size();
    bool exception = false;
    if (std::string(1, input[size - 1]) == "x") {
        input.erase(size - 1, 1);
        exception = true;
```

```
    }
    for (unsigned i = 0; i < input.size(); i += 2) {
        std::string tmp = std::string(1, input[i]) + std::string(1, input[i + 1]);
        split.push_back(tmp);
    }
    for (unsigned int i = 0; i < split.size(); i++) {
        std::string letter = std::string(1, split[i][0]),
               digit = std::string(1, split[i][1]);
        int number = position(letter) + stoi(digit)*26;
        std::string tmp = std::to_string(number);
        if (tmp.size() == 1) {
            tmp = "0" + tmp;
        }
        result += tmp;
    }
    if(exception) result.erase(result.size() - 1, 1);
    reverse(result.begin(), result.end());
    return result;
}


float MathCNN::scalarProduct(const std::vector<float>* vec1, const std::vector<float>* vec2) {
    float sum = 0;
    for (unsigned int i = 0; i < vec1->size(); i++) {
        sum += (*vec1)[i] * (*vec2)[i];
    }
    return sum;
};

std::vector<std::vector<float>> MathCNN::MMProduct(const std::vector<std::vector<float>>* matrix
1, const std::vector<std::vector<float>>* matrix2) {
    std::vector<std::vector<float>> result;
    for (unsigned int i = 0; i < matrix1->size(); i++) {
        result.push_back(std::vector<float>());
        //vec1 = matrix1[i]
        for (unsigned int j = 0; j < (*matrix2)[0].size(); j++) {
            std::vector<float> vec2;
            for (unsigned k = 0; k < matrix2->size(); k++) {
                vec2.push_back((*matrix2)[k][j]);
            }
            result[i].push_back(scalarProduct(&((*matrix1)[i]), &vec2));
        }
    }
    return result;
};


void MathCNN::MVaddColumn(std::vector<std::vector<float>>* matrix, const std::vector<float>* vec
) {
```

```
    for (unsigned int i = 0; i < matrix->size(); i++) {
        for (unsigned int j = 0; j < (*matrix)[i].size(); j++) {
            (*matrix)[i][j] += (*vec)[i];
        }
    }
};

void MathCNN::MVmultiplyColumn(std::vector<std::vector<float>>* matrix, const std::vector<float>
* vec) {
    for (unsigned int i = 0; i < matrix->size(); i++) {
        for (unsigned int j = 0; j < (*matrix)[i].size(); j++) {
            (*matrix)[i][j] *= (*vec)[i];
        }
    }
};
float MathCNN::distance(std::vector<float>* p1, std::vector<float>* p2) {
    float sum = 0;
    for (int i = 0; i < p1->size(); i++) {
        sum += pow((*p1)[i] - (*p2)[i], 2);
    }
    return sqrt(sum);
};
void MathCNN::ReLU(std::vector<std::vector<float>>* z) {
    for (unsigned int i = 0; i < z->size(); i++) {
        for (unsigned int j = 0; j < (*z)[i].size(); j++) {
            if ((*z)[i][j] < 0) (*z)[i][j] = 0;
        }
    }
};
static std::vector<float> c = { 1 };
float MathCNN::ck(float k) {
    if (k == 0) return c[k];
    c.push_back(0);
    float sum = 0;
    for (int m = 0; m < k; m++) {
        float tmp = c[m] * c[k - 1 - m];
        tmp /= (m + 1) * (2 * m + 1);
        sum += tmp;
    }
    c[k] = sum;
    return c[k];
};
float MathCNN::erf_1(float x) {
    const float PI = 3.141592741;
    float sum = 0;
    for(int k=0; k<12; k++) {
        float tmp = ck(k) * pow(sqrt(PI) * 0.5 * x, 2 * k + 1);
        tmp /= 2 * k + 1;
```

```
        sum += tmp;
    }
    return sum;
};
float MathCNN::getDistance(float sigmaW, float u) {
    return pow(((sigmaW * sqrt(2) * erf_1(2 * u - 1) - 5) / 2), 2) - 0.1;
};
int MathCNN::factorial(int n) {
    if (n == 0) return 1;
    int tmp = n * factorial(n - 1);
    return n * factorial(n - 1);
};
float MathCNN::getMeanDistance(float s) {
    const float E = 2.718281828;
    float m = 0.561756 * pow(E, -9.537722 * pow(s, -2));
    m -= 1.3187 * pow(E, -5.19039 * pow(s, -2));
    m *= s;
    m += (5 * pow(s, 2) + 123) * (erf(2.27824 / s) - erf(-3.08832 / s)) / 40;
    return m;
};
std::vector<float> MathCNN::dotProduct(const std::vector<float>* in1, const std::vector<float>*
in2) {
    std::vector<float> result;
    for (unsigned int i = 0; i < in1->size(); i++) {
        result.push_back((*in1)[i] * (*in2)[i]);
    }
    return result;
};
std::vector<float> MathCNN::subtractVec(std::vector<float>* vec1, std::vector<float>* vec2) {
    std::vector<float> result;
    if (vec1->size() != vec2->size()) return result;
    for (unsigned int i = 0; i < vec1->size(); i++) {
        result.push_back((*vec1)[i] - (*vec2)[i]);
    }
    return result;
};
int MathCNN::convSize(int x, int s, int p, int k) {
    return ((x + 2 * p - k) / s) + 1;
};
featureMap MathCNN::conv(convAL* layer, convA* kernel, int stride) {
    //conv: BxHxT
    int kernelS = kernel->size();
    int resWidth = MathCNN::convSize(layer->size(), stride, 0, kernelS);
    int resHeight = MathCNN::convSize((*layer)[0].size(), stride, 0, kernelS);
    int featureMaps = (*layer)[0][0].size();
    featureMap result = featureMap(resWidth, std::vector<float>(resHeight));
    for (int i = 0; i < resWidth; i++) {
        for (int j = 0; j < resHeight; j++) {
```

```
            float sum = 0;
            for (int k1 = 0; k1 < kernelS; k1++) {
                for (int k2 = 0; k2 < kernelS; k2++) {
                    for (int f = 0; f < featureMaps; f++) {
                        int I = k1 + i * stride,
                            J = k2 + j * stride;
                        sum += (*kernel)[k1][k2][f] * (*layer)[I][J][f];
                    }
                }
            }
            result[i][j] = sum;
        }
    }
    return result;
};
featureMap MathCNN::conv(convAL* layer, convA* kernel, int stride, int padding) {
    convA newL = *layer;
    int featureMaps = (*layer)[0][0].size();
    std::vector<float> border2(featureMaps);
    for (int i = 0; i < newL.size(); i++) { //width
        for (int j = 0; j < padding; j++) {
            newL[i].insert(newL[i].begin(), border2);
            newL[i].push_back(border2);
        }
    }
    std::vector<std::vector<float>> border(newL[0].size(), std::vector<float>(featureMaps));
    for (int j = 0; j < padding; j++) {
        newL.insert(newL.begin(), border);
        newL.push_back(border);
    }
    return conv(&newL, kernel, stride);
};
std::vector<convA> MathCNN::ReLU(std::vector<convA>* z) {
    std::vector<convA> result = *z;
    for (unsigned int i = 0; i < z->size(); i++) {
        for (unsigned int j = 0; j < (*z)[i].size(); j++) {
            for (unsigned int k = 0; k < (*z)[i][j].size(); k++) {
                for (unsigned int l = 0; l < (*z)[i][j][k].size(); l++) {
                    if ((*z)[i][j][k][l] > 0) {
                        result[i][j][k][l] = (*z)[i][j][k][l];
                    } else {
                        result[i][j][k][l] = 0;
                    }
                }
            }
        }
    }
    return result;
```

```
};
std::vector<convA> MathCNN::ReLU_(std::vector<convA>* z) {
    std::vector<convA> result = *z;
    for (unsigned int i = 0; i < z->size(); i++) {
        for (unsigned int j = 0; j < (*z)[i].size(); j++) {
            for (unsigned int k = 0; k < (*z)[i][j].size(); k++) {
                for (unsigned int l = 0; l < (*z)[i][j][k].size(); l++) {
                    if ((*z)[i][j][k][l] > 0) {
                        result[i][j][k][l] = 1;
                    }
                    else {
                        result[i][j][k][l] = 0;
                    }
                }
            }
        }
    }
    return result;
};
void MathCNN::ReLU(vconvAL* z, vconvAL* writeInto) {
    for (unsigned int i = 0; i < z->size(); i++) {
        for (unsigned int j = 0; j < (*z)[i].size(); j++) {
            for (unsigned int k = 0; k < (*z)[i][j].size(); k++) {
                for (unsigned int l = 0; l < (*z)[i][j][k].size(); l++) {
                    if ((*z)[i][j][k][l] > 0) {
                        (*writeInto)[i][j][k][l] = (*z)[i][j][k][l];
                    }
                    else {
                        (*writeInto)[i][j][k][l] = 0;
                    }
                }
            }
        }
    }
};
void MathCNN::ReLU_(vconvAL* z, vconvAL* writeInto) {
    for (unsigned int i = 0; i < z->size(); i++) {
        for (unsigned int j = 0; j < (*z)[i].size(); j++) {
            for (unsigned int k = 0; k < (*z)[i][j].size(); k++) {
                for (unsigned int l = 0; l < (*z)[i][j][k].size(); l++) {
                    if ((*z)[i][j][k][l] > 0) {
                        (*writeInto)[i][j][k][l] = 1;
                    }
                    else {
                        (*writeInto)[i][j][k][l] = 0;
                    }
                }
            }
```

```
        }
    }
};
void MathCNN::multiplyElementwise(std::vector<convA>* z, convA* gamma) {
    for (unsigned int i = 0; i < z->size(); i++) {
        for (unsigned int j = 0; j < (*z)[i].size(); j++) {
            for (unsigned int k = 0; k < (*z)[i][j].size(); k++) {
                for (unsigned int l = 0; l < (*z)[i][j][k].size(); l++) {
                    (*z)[i][j][k][l] *= (*gamma)[j][k][l];
                }
            }
        }
    }
};
void MathCNN::addElementwise(std::vector<convA>* z, convA* beta) {
    for (unsigned int i = 0; i < z->size(); i++) {
        for (unsigned int j = 0; j < (*z)[i].size(); j++) {
            for (unsigned int k = 0; k < (*z)[i][j].size(); k++) {
                for (unsigned int l = 0; l < (*z)[i][j][k].size(); l++) {
                    (*z)[i][j][k][l] += (*beta)[j][k][l];
                }
            }
        }
    }
};
void MathCNN::retransformBN(vconvAL* z, std::vector<float>* gamma, std::vector<float>* beta) {
    for (unsigned int i = 0; i < z->size(); i++) {
        for (unsigned int j = 0; j < (*z)[i].size(); j++) {
            for (unsigned int k = 0; k < (*z)[i][j].size(); k++) {
                for (unsigned int l = 0; l < (*z)[i][j][k].size(); l++) {
                    (*z)[i][j][k][l] *= (*gamma)[l];
                    (*z)[i][j][k][l] += (*beta)[l];
                }
            }
        }
    }
};
void MathCNN::addActMapwise(vconvAL* z, std::vector<float>* beta) {
    for (unsigned int i = 0; i < z->size(); i++) {
        for (unsigned int j = 0; j < (*z)[i].size(); j++) {
            for (unsigned int k = 0; k < (*z)[i][j].size(); k++) {
                for (unsigned int l = 0; l < (*z)[i][j][k].size(); l++) {
                    (*z)[i][j][k][l] += (*beta)[l];
                }
            }
        }
    }
};
```

```
convAL MathCNN::dotProduct(convAL* mtrx1, convAL* mtrx2) {
    convAL result;
    for (unsigned int i = 0; i < mtrx1->size(); i++) {
        result.push_back(vectorff());
        for (unsigned int j = 0; j < (*mtrx1)[i].size(); j++) {
            result[i].push_back(vectorf());
            for (unsigned int k = 0; k < (*mtrx1)[i][j].size(); k++) {
                result[i][j].push_back((*mtrx1)[i][j][k] * (*mtrx2)[i][j][k]);
            }
        }
    }
    return result;
};
convA MathCNN::rotate180(convA* kernel) {
    convA result = *kernel;
    for (unsigned int w = 0; w < kernel->size(); w++) {
        for (unsigned int h = 0; h < (*kernel)[w].size(); h++) {
            int W = (*kernel).size() - w - 1,
                H = (*kernel)[w].size() - h - 1;
            for (unsigned int d = 0; d < (*kernel)[w][h].size(); d++) {
                result[W][H][d] = (*kernel)[w][h][d];
            }
        }
    }
    return result;
};
convAL MathCNN::asStxxl(convA* in) {
    convAL ret;
    for (unsigned int i = 0; i < in->size(); i++) {
        ret.push_back(vectorff());
        for (unsigned int j = 0; j < (*in)[i].size(); j++) {
            ret[i].push_back(vectorf());
            for (unsigned int k = 0; k < (*in)[i][j].size(); k++) {
                ret[i][j].push_back((*in)[i][j][k]);
            }
        }
    }
    return ret;
};
vconvAL MathCNN::asStxxl(std::vector<convA>* in) {
    vconvAL ret;
    for (unsigned int i = 0; i < in->size(); i++) {
        ret.push_back((*in)[i]);
    }
    return ret;
};
convA MathCNN::aStd(convAL* in) {
    convA ret;
```

```
    for (unsigned int i = 0; i < in->size(); i++) {
        ret.push_back(denseW());
        for (unsigned int j = 0; j < (*in)[i].size(); j++) {
            ret[j].push_back(denseA());
            for (unsigned int k = 0; k < (*in)[i][j].size(); k++) {
                ret[i][j].push_back((*in)[i][j][k]);
            }
        }
    }
    return ret;
};
```

MathCNN.h

```
#ifndef _MathCNNH_
#define _MathCNNH_

#include <string>
#include <vector>
#include <random>
#include "activations.h"

class MathCNN {
private:
    int static factorial(int n);
    float static ck(float k);
    float static erf_1(float x);
    const std::vector<std::string> alphabet = { "a", "b", "c", "d", "e", "f", "g", "h", "i", "j"
, "k", "l", "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z"};
    int position(std::string letter);
    float static scalarProduct(const std::vector<float>* vec1, const std::vector<float>* vec2);
public:
    MathCNN();
    float static getMeanDistance(float sigmaW);
    float static getDistance(float sigmaW, float uniformNo);
    std::vector<float> static MVProduct(const std::vector<std::vector<float>>* matrix, const std
::vector<float>* vector);
    void static MVaddColumn(std::vector<std::vector<float>>* matrix, const std::vector<float>* v
ector);
    void static MVmultiplyColumn(std::vector<std::vector<float>>* matrix, const std::vector<floa
t>* vector);
    std::vector<std::vector<float>> static MMProduct(const std::vector<std::vector<float>>* matr
ix1, const std::vector<std::vector<float>>* matrix2);
    float static getRandomNormal(const float* mean, const float* dev, std::default_random_engine
* generator);
    std::vector<float> static addVec(std::vector<float>* vec1, std::vector<float>* vec2);
    std::vector<float> static subtractVec(std::vector<float>* vec1, std::vector<float>* vec2);
    float static distance(std::vector<float>* p1, std::vector<float>* p2);
    float ReLU(float z);
    static void ReLU(float* z);
```

```
    static void ReLU(std::vector<float>* z);
    static void ReLU(std::vector<std::vector<float>>* z);
    static void ReLU_(std::vector<float>* z);
    static std::vector<convA> ReLU(std::vector<convA>* z);
    static std::vector<convA> ReLU_(std::vector<convA>* z);
    static void ReLU(vconvAL* z, vconvAL* writeInto);
    static void ReLU_(vconvAL* z, vconvAL* writeInto);
    static void softmax(const std::vector<float>* in, std::vector<float>* out);
    static void dotProduct(const std::vector<float>* in1, const std::vector<float>* in2, std::ve
ctor<float>* result);
    static std::vector<float> dotProduct(const std::vector<float>* in1, const std::vector<float>
* in2);
    std::vector<std::vector<float>> static transposeM(const std::vector<std::vector<float>>* mat
rix);
    std::string encode(std::string input);
    std::string decode(std::string input);
    featureMap static conv(convAL *layer, convA *kernel, int stride);
    featureMap static conv(convAL* layer, convA* kernel, int stride, int padding);
    int static convSize(int x, int s, int p, int k);
    static void multiplyElementwise(std::vector<convA> *u, convA *gamma);
    static void addElementwise(std::vector<convA>* u, convA* beta);
    static void retransformBN(vconvAL* u, std::vector<float>* gamma, std::vector<float>* beta);
    static void addActMapwise(vconvAL* u, std::vector<float>* beta);
    static convAL dotProduct(convAL* mtrx1, convAL* mtrx2);
    static convA rotate180(convA* kernel);
    static convAL asStxxl(convA* in);
    static convA aStd(convAL* in);
    static vconvAL asStxxl(std::vector<convA>* in);
};

#endif
```

MultiDense.cpp

```
#include "MultiDense.h"
#include "MathCNN.h"
#include <math.h>
#include <iostream>
#include <random>
#include <time.h>

MultiDense::MultiDense(int size_, std::vector<denseA>* inputAct_, std::vector<std::vector<float>
>* inputPos_, float *m, int ixL) : size(size_), inputAct(inputAct_), inputPos(inputPos_), meanl_
1(m), ixLayer(ixL) {
    sl_1 = (*inputAct)[0].size();
    activations = std::vector<denseA>(1, denseA(size));
    gradient = std::vector<denseA>(1, denseA(size));
    position = std::vector<std::vector<float>>(size, std::vector<float>(dimensions));
    upGradient = nullptr;
    avgM = denseA(size);
```

```
    avgVar = denseA(size);
};
void MultiDense::forward() {
    std::vector<denseA> z = std::vector<denseA>(inputAct->size(), denseA(size));
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < inputAct->size(); j++) {
            for (unsigned int k = 0; k < (*inputAct)[j].size(); k++) {
                std::vector<float>* inPos = &((*inputPos)[k]),
                            * outPos = &(position[i]);
                float distance = MathCNN::distance(inPos, outPos);
                float weight = betaM[i] - alphaM[i] * sqrt(distance + deltaM);
                z[j][i] += weight * (*inputAct)[j][k];
            }
        }
    }
    zws = z;
    if (!lastLayer) {
        if (!converged) {
            denseA mean = denseA(zws[0].size()),
                   stddev2 = denseA(zws[0].size());
            BatchNorm(&z, &mean, &stddev2, delta);
            m = mean;
            var = stddev2;
            avgM = MathCNN::addVec(&avgM, &m);
            avgVar = MathCNN::addVec(&avgVar, &var);
        } else {
            BatchNorm(&z, avgM, avgVar, delta);
        }
        u = z;
        z = MathCNN::transposeM(&z);
        MathCNN::MVmultiplyColumn(&z, &gamma);
        MathCNN::MVaddColumn(&z, &beta);
        zws_hat = MathCNN::transposeM(&z);
    } else {
        z = MathCNN::transposeM(&z);
        MathCNN::MVaddColumn(&z, &beta);
    }
    activations = MathCNN::transposeM(&z);
    if (!lastLayer) MathCNN::ReLU(&activations);
};
void MultiDense::backward() {
    std::vector<denseA> d_zh = *upGradient;
    std::vector<denseA> dz;
    if (!lastLayer) {
        for (unsigned int i = 0; i < d_zh.size(); i++) {
            denseA z_h = zws_hat[i];
            MathCNN::ReLU_(&z_h);
            d_zh[i] = MathCNN::dotProduct(&z_h, &((*upGradient)[i]));
```

```
        }
        std::vector<float> d_beta;
        std::vector<float> d_gamma;
        std::vector<denseA> d_u = u;
        denseA d_var = denseA(d_zh[0].size());
        for (unsigned int j = 0; j < d_zh[0].size(); j++) {
            float avg_dB = 0,
                  avg_dG = 0;
            for (unsigned int i = 0; i < d_zh.size(); i++) {
                avg_dB += d_zh[i][j];
                avg_dG += d_zh[i][j] * u[i][j];
                d_u[i][j] = d_zh[i][j] * gamma[j];
                d_var[j] += d_u[i][j] * (m[j] - zws[i][j]);
            }
            d_var[j] /= 2 * sqrt(pow(var[j] + delta, 3));
            avg_dB /= (float) d_zh.size();
            avg_dG /= (float) d_zh.size();
            d_beta.push_back(avg_dB);
            d_gamma.push_back(avg_dG);
        }
        int batchS = zws.size();
        betaGradients = d_beta;
        gammaGradients = d_gamma;
        denseA d_mean = denseA(zws[0].size());
        for (unsigned int j = 0; j < zws[0].size(); j++) { //this Layer
            float tmp1 = 0, tmp2 = 0;
            for (unsigned int i = 0; i < zws.size(); i++) { //batches
                tmp1 += m[j] - zws[i][j];
                tmp2 += d_u[i][j];
            }
            d_mean[j] = d_var[j] * 2 * tmp1 / ((float)batchS);
            d_mean[j] -= tmp2 / sqrt(var[j] + delta);
        }
        std::vector<denseA> d_z(zws.size(), denseA(zws[0].size()));
        for (unsigned int i = 0; i < d_z.size(); i++) {
            for (unsigned int j = 0; j < d_z[i].size(); j++) {
                d_z[i][j] = d_mean[j] / ((float)batchS);
                d_z[i][j] += d_var[j] * 2 * (zws[i][j] - m[j]) / (float)batchS;
                d_z[i][j] += d_u[i][j] / sqrt(var[j] + delta);
            }
        }
        dz = d_z;
    } else {
        dz = d_zh;
        betaGradients = std::vector<float>(0);
        for (unsigned int i = 0; i < dz[0].size(); i++) {
            float avgB = 0;
            for (unsigned int j = 0; j < dz.size(); j++) {
```

```
                avgB += dz[j][i];
            }
            avgB /= dz.size();
            betaGradients.push_back(avgB);
        }
    }
    std::vector<denseW> weightGradients = std::vector<denseW>(zws.size(), denseW(size, std::vect
or<float>(sl_1, 0)));
    for (unsigned int i = 0; i < weightGradients.size(); i++) { //i=batches
        for (unsigned int j = 0; j < weightGradients[i].size(); j++) { //j=thisLayer
            for (unsigned int k = 0; k < weightGradients[i][j].size(); k++) { //k=prevLayer
                weightGradients[i][j][k] = dz[i][j] * (*inputAct)[i][k];
            }
        }
    }
    wG = weightGradients;
    std::vector<denseA> gra(inputAct->size(), std::vector<float>((*inputAct)[0].size(), 0));
    for (unsigned int i = 0; i < gra.size(); i++) { //i=batches
        for (unsigned int j = 0; j < gra[i].size(); j++) { //j=prevLayer
            for (unsigned int k = 0; k < zws[0].size(); k++) { //k=thisLayer
                std::vector<float>* inPos = &((*inputPos)[j]),
                            * outPos = &(position[k]);
                float distance = MathCNN::distance(inPos, outPos);
                float weight = betaM[k] - alphaM[k] * sqrt(distance + deltaM);
                gra[i][j] += weight * dz[i][k];
            }
        }
    }
    gradient = gra;
    //own backprop
    betaMGradients = std::vector<float>(size);
    alphaMGradients = std::vector<float>(size);
    for (unsigned int i = 0; i < betaMGradients.size(); i++) { //thisLayer
        float avgB = 0;
        float avgA = 0;
        for (unsigned int k = 0; k < zws.size(); k++) { //batches
            float sumB = 0;
            float sumA = 0;
            for (unsigned int j = 0; j < sl_1; j++) { //prevLayer
                sumB += weightGradients[k][i][j];
                std::vector<float>* inPos = &((*inputPos)[j]),
                            * outPos = &(position[i]);
                float distance = MathCNN::distance(inPos, outPos);
                sumA += weightGradients[k][i][j] * sqrt(distance + deltaM);
            }
            avgB += sumB;
            sumA *= (-1);
            avgA += sumA;
```

```
        }
        avgB /= zws.size();
        avgA /= zws.size();
        betaMGradients[i] = avgB;
        alphaMGradients[i] = avgA;
    }
    for (unsigned int i = 0; i < size; i++) { //this Layer
        for (unsigned int j = 0; j < dimensions; j++) { //dimensions
            float avgC = 0;
            for (unsigned int k = 0; k < zws.size(); k++) { //batches
                float sum = 0;
                for (unsigned int l = 0; l < sl_1; l++) { //prev Layer
                    float tmp = weightGradients[k][i][l];
                    std::vector<float>* inPos = &((*inputPos)[l]),
                                * outPos = &(position[i]);
                    float distance = MathCNN::distance(inPos, outPos);
                    tmp *= (-1) * alphaM[i] / (2 * sqrt(distance + deltaM));
                    tmp *= (position[i][j] - (*inputPos)[l][j]) / distance;
                    sum += tmp;
                }
                avgC += sum;
            }
            avgC /= zws.size();
            positionGradients[i][j] = avgC;
        }
    }
    if (!lastLayer && nextW) {
        for (unsigned int i = 0; i < size; i++) { //this Layer
            for (unsigned int j = 0; j < dimensions; j++) { //dimensions
                float avgC = 0;
                for (unsigned int k = 0; k < zws.size(); k++) { //batches
                    float sum = 0;
                    for (unsigned int l = 0; l < posp1->size(); l++) { //next Layer
                        float tmp = (*wGp1)[k][l][i];
                        std::vector<float>* inPos = &(position[i]),
                                    * outPos = &((*posp1)[l]);
                        float distance = MathCNN::distance(inPos, outPos);
                        tmp *= (-1) * (*ap1)[l] / (2 * sqrt(distance + deltaM));
                        tmp *= (position[i][j] - (*posp1)[l][j]) / distance;
                        sum += tmp;
                    }
                    avgC += sum;
                }
                avgC /= zws.size();
                positionGradients[i][j] += avgC;
            }
        }
    }
```

```
    updateParameters();
};
void MultiDense::initialiseParameters(std::string actF) {
    for (int i = 0; i < size; i++) {
        alphaM.push_back((float)2);
        betaM.push_back((float)5);
        gamma.push_back((float)1);
        beta.push_back((float)0);
    }
    float stddev = 0;
    if (actF == "ReLU") {
        stddev = sqrt((float)2 / (float)sl_1); //He initialization
    }
    mean = *meanl_1 + (MathCNN::getMeanDistance(stddev) / sqrt(2));
    srand((unsigned)time(NULL));
    const float PI = 3.141592741;
    std::default_random_engine generator;
    for (unsigned int i = 0; i < position.size(); i++) {
        float u = (float)rand() / RAND_MAX;
        float distance = (MathCNN::getDistance(stddev, u) / sqrt(2));
        position[i][0] = *meanl_1 + distance;
        if (ixLayer % 2 != 0) {
            u = (float)rand() / RAND_MAX;
            float phi = 2 * PI * u;
            position[i][1] = distance * cos(phi);
            position[i][2] = distance * sin(phi);
        } else {
            float m2 = 0;
        }
    }

    betaVelocity1 = std::vector<float>(beta.size());
    betaVelocity2 = std::vector<float>(beta.size());
    gammaVelocity1 = std::vector<float>(gamma.size());
    gammaVelocity2 = std::vector<float>(gamma.size());
    betaMVelocity1 = std::vector<float>(betaM.size());
    betaMVelocity2 = std::vector<float>(betaM.size());
    alphaMVelocity1 = std::vector<float>(alphaM.size());
    alphaMVelocity2 = std::vector<float>(alphaM.size());
    positionVelocity1 = std::vector<std::vector<float>>(size, std::vector<float>(dimensions));
    positionVelocity2 = std::vector<std::vector<float>>(size, std::vector<float>(dimensions));
    positionGradients = std::vector<std::vector<float>>(size, std::vector<float>(dimensions));
};
void MultiDense::setUpGradient(std::vector<denseA>* upGradient_) {
    upGradient = upGradient_;
};
void MultiDense::setInputPosition(std::vector<std::vector<float>>* inputPos_) {
    inputPos = inputPos_;
```

```
};
void MultiDense::updateParameters() {
    if (!lastLayer) {
        for (unsigned int i = 0; i < beta.size(); i++) {
            betaVelocity1[i] = beta_o1 * betaVelocity1[i] + ((1 - beta_o1) * betaGradients[i]);
            betaVelocity2[i] = beta_o2 * betaVelocity2[i] + ((1 - beta_o2) * pow(betaGradients[i
], 2));
            gammaVelocity1[i] = beta_o1 * gammaVelocity1[i] + ((1 - beta_o1) * gammaGradients[i]
);
            gammaVelocity2[i] = beta_o2 * gammaVelocity2[i] + ((1 - beta_o2) * pow(gammaGradient
s[i], 2));
        }
        for (unsigned int i = 0; i < beta.size(); i++) {
            float betaV1_h = betaVelocity1[i] / (1 - pow(beta_o1, epoch));
            float betaV2_h = betaVelocity2[i] / (1 - pow(beta_o2, epoch));
            float gammaV1_h = gammaVelocity1[i] / (1 - pow(beta_o1, epoch));
            float gammaV2_h = gammaVelocity2[i] / (1 - pow(beta_o2, epoch));
            beta[i] -= eta * (betaV1_h / sqrt(betaV2_h + 0.00001));
            gamma[i] -= eta * (gammaV1_h / sqrt(gammaV2_h + 0.00001));
        }
    }
    else { //lastLayer
        for (unsigned int i = 0; i < beta.size(); i++) {
            betaVelocity1[i] = beta_o1 * betaVelocity1[i] + ((1 - beta_o1) * betaGradients[i]);
            betaVelocity2[i] = beta_o2 * betaVelocity2[i] + ((1 - beta_o2) * pow(betaGradients[i
], 2));
        }
        for (unsigned int i = 0; i < beta.size(); i++) {
            float betaV1_h = betaVelocity1[i] / (1 - pow(beta_o1, epoch));
            float betaV2_h = betaVelocity2[i] / (1 - pow(beta_o2, epoch));
            beta[i] -= eta * (betaV1_h / sqrt(betaV2_h + 0.00001));
        }
    }
    for (unsigned int i = 0; i < alphaMGradients.size(); i++) {
        betaMVelocity1[i] = beta_o1 * betaMVelocity1[i] + ((1 - beta_o1) * betaMGradients[i]);
        betaMVelocity2[i] = beta_o2 * betaMVelocity2[i] + ((1 - beta_o2) * pow(betaMGradients[i]
, 2));
        alphaMVelocity1[i] = beta_o1 * alphaMVelocity1[i] + ((1 - beta_o1) * alphaMGradients[i])
;
        alphaMVelocity2[i] = beta_o2 * alphaMVelocity2[i] + ((1 - beta_o2) * pow(alphaMGradients
[i], 2));
    }
    for (unsigned int i = 0; i < alphaMGradients.size(); i++) {
        float betaMV1_h = betaMVelocity1[i] / (1 - pow(beta_o1, epoch));
        float betaMV2_h = betaMVelocity2[i] / (1 - pow(beta_o2, epoch));
        float alphaMV1_h = alphaMVelocity1[i] / (1 - pow(beta_o1, epoch));
        float alphaMV2_h = alphaMVelocity2[i] / (1 - pow(beta_o2, epoch));
        betaM[i] -= eta * (betaMV1_h / sqrt(betaMV2_h + 0.00001));
```

```
            alphaM[i] -= eta * (alphaMV1_h / sqrt(alphaMV2_h + 0.00001));
        }
        for (unsigned int i = 0; i < positionGradients.size(); i++) {
            for (unsigned int j = 0; j < positionGradients[i].size(); j++) {
                positionVelocity1[i][j] = beta_o1 * positionVelocity1[i][j] + ((1 - beta_o1) * posit
ionGradients[i][j]);
                positionVelocity2[i][j] = beta_o2 * positionVelocity2[i][j] + ((1 - beta_o2) * pow(p
ositionGradients[i][j], 2));
            }
        }
        for (unsigned int i = 0; i < position.size(); i++) {
            for (unsigned int j = 0; j < position[i].size(); j++) {
                float positionV1_h = positionVelocity1[i][j] / (1 - pow(beta_o1, epoch));
                float positionV2_h = positionVelocity2[i][j] / (1 - pow(beta_o2, epoch));
                position[i][j] -= eta * (positionV1_h / sqrt(positionV2_h + 0.00001));
            }
        }
        epoch++;
};
void MultiDense::endTraining() {
        converged = true;
        betaGradients.clear();
        betaGradients.shrink_to_fit();
        betaVelocity1.clear();
        betaVelocity1.shrink_to_fit();
        betaVelocity2.clear();
        betaVelocity2.shrink_to_fit();
        gammaGradients.clear();
        gammaGradients.shrink_to_fit();
        gammaVelocity1.clear();
        gammaVelocity1.shrink_to_fit();
        gammaVelocity2.clear();
        gammaVelocity2.shrink_to_fit();
        betaMGradients.clear();
        betaMGradients.shrink_to_fit();
        betaMVelocity1.clear();
        betaMVelocity1.shrink_to_fit();
        betaMVelocity2.clear();
        betaMVelocity2.shrink_to_fit();
        alphaMGradients.clear();
        alphaMGradients.shrink_to_fit();
        alphaMVelocity1.clear();
        alphaMVelocity1.shrink_to_fit();
        alphaMVelocity2.clear();
        alphaMVelocity2.shrink_to_fit();
        positionGradients.clear();
        positionGradients.shrink_to_fit();
        positionVelocity1.clear();
```

```
    positionVelocity1.shrink_to_fit();
    positionVelocity2.clear();
    positionVelocity2.shrink_to_fit();
    gradient.clear();
    gradient.shrink_to_fit();
    activations.clear();
    activations.shrink_to_fit();
}

denseA MultiDense::getAvgMean() {
    denseA avg = denseA(avgM);
    for (unsigned int i = 0; i < zws[0].size(); i++) {
        avg[i] /= (epoch - 1);
    }
    return avg;
}

denseA MultiDense::getAvgVar() {
    float m = zws.size();
    denseA avg = denseA(avgVar);
    for (unsigned int i = 0; i < zws[0].size(); i++) {
        avg[i] *= m / ((m - 1) * (epoch - 1));
    }
    return avg;
}
```

MultiDense.h

```
#ifndef _MultiDenseH_
#define _MultiDenseH_

#include "Layer.h"
#include <string>

class MultiDense : public Layer {
private:
    int dimensions = 3;
    int size, sl_1;
    std::vector<denseA>* inputAct;
    std::vector<std::vector<float>>* inputPos;
    std::vector<denseA>* upGradient;

    float delta = 0.0001;
    std::vector<denseA> zws;
    std::vector<denseA> u;
    std::vector<denseA> zws_hat;
    denseA m, var;
    float deltaM = 0.1;

    denseA betaGradients;
```

```
    denseA betaVelocity1;
    denseA betaVelocity2;
    denseA gammaGradients;
    denseA gammaVelocity1;
    denseA gammaVelocity2;
    denseA betaMGradients;
    denseA betaMVelocity1;
    denseA betaMVelocity2;
    denseA alphaMGradients;
    denseA alphaMVelocity1;
    denseA alphaMVelocity2;
    std::vector<std::vector<float>> positionGradients;
    std::vector<std::vector<float>> positionVelocity1;
    std::vector<std::vector<float>> positionVelocity2;

    void updateParameters();
    float eta = 0.001,
          beta_o1 = 0.9,
          beta_o2 = 0.999;
    int epoch = 1;

public:
    bool converged = false;
    denseA avgM, avgVar;
    denseA betaM;
    denseA alphaM;
    denseA gamma;
    denseA beta;
    std::vector<std::vector<float>> position;
    int ixLayer;
    bool lastLayer = false,
         nextW = true;
    float mean, *meanl_1;
    std::vector<denseA> activations;
    std::vector<denseA> gradient;
    std::vector<denseW> wG;
    std::vector<denseW> *wGp1;
    std::vector<float>* ap1;
    std::vector<std::vector<float>> *posp1;
    MultiDense(int size, std::vector<denseA>* inputAct, std::vector<std::vector<float>>* inputPo
s, float *meanl_1, int ixLayer);
    void forward();
    void backward();
    void initialiseParameters(std::string actF);
    void setUpGradient(std::vector<denseA>* upGradient);
    void setInputPosition(std::vector<std::vector<float>>* inputPos);
    void endTraining();
    denseA getAvgMean();
```

```
    denseA getAvgVar();
};

#endif;
```

MultiDense2.cpp

```
#include "MultiDense2.h"
#include "MathCNN.h"
#include <math.h>
#include <iostream>
#include <random>
#include <time.h>

MultiDense2::MultiDense2(int size_, std::vector<denseA>* inputAct_, std::vector<std::vector<floa
t>>* inputPos_, float* m, int ixL) : size(size_), inputAct(inputAct_), inputPos(inputPos_), mean
l_1(m), ixLayer(ixL) {
    sl_1 = (*inputAct)[0].size();
    activations = std::vector<denseA>(1, denseA(size));
    gradient = std::vector<denseA>(1, denseA(size));
    position = std::vector<std::vector<float>>(size, std::vector<float>(dimensions));
    upGradient = nullptr;
    avgM = denseA(size);
    avgVar = denseA(size);
};
void MultiDense2::forward() {
    std::vector<denseA> z = std::vector<denseA>(inputAct->size(), denseA(size));
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < inputAct->size(); j++) {
            for (unsigned int k = 0; k < (*inputAct)[j].size(); k++) {
                std::vector<float>* inPos = &((*inputPos)[k]),
                                  * outPos = &(position[i]);
                float distance = MathCNN::distance(inPos, outPos);
                float weight = alphaM[i] * distance + betaM[i];
                z[j][i] += weight * (*inputAct)[j][k];
            }
        }
    }
    zws = z;
    if (!lastLayer) {
        if (!converged) {
            denseA mean = denseA(zws[0].size()),
                   stddev2 = denseA(zws[0].size());
            BatchNorm(&z, &mean, &stddev2, delta);
            m = mean;
            var = stddev2;
            avgM = MathCNN::addVec(&avgM, &m);
            avgVar = MathCNN::addVec(&avgVar, &var);
        }
        else {
```

```
            BatchNorm(&z, avgM, avgVar, delta);
        }
        u = z;
        z = MathCNN::transposeM(&z);
        MathCNN::MVmultiplyColumn(&z, &gamma);
        MathCNN::MVaddColumn(&z, &beta);
        zws_hat = MathCNN::transposeM(&z);
    }
    else {
        z = MathCNN::transposeM(&z);
        MathCNN::MVaddColumn(&z, &beta);
    }
    activations = MathCNN::transposeM(&z);
    if (!lastLayer) MathCNN::ReLU(&activations);
};
void MultiDense2::backward() {
    std::vector<denseA> d_zh = *upGradient;
    std::vector<denseA> dz;
    if (!lastLayer) {
        for (unsigned int i = 0; i < d_zh.size(); i++) {
            denseA z_h = zws_hat[i];
            MathCNN::ReLU_(&z_h);
            d_zh[i] = MathCNN::dotProduct(&z_h, &((*upGradient)[i]));
        }
        std::vector<float> d_beta;
        std::vector<float> d_gamma;
        std::vector<denseA> d_u = u;
        denseA d_var = denseA(d_zh[0].size());
        for (unsigned int j = 0; j < d_zh[0].size(); j++) {
            float avg_dB = 0,
                avg_dG = 0;
            for (unsigned int i = 0; i < d_zh.size(); i++) {
                avg_dB += d_zh[i][j];
                avg_dG += d_zh[i][j] * u[i][j];
                d_u[i][j] = d_zh[i][j] * gamma[j];
                d_var[j] += d_u[i][j] * (m[j] - zws[i][j]);
            }
            d_var[j] /= 2 * sqrt(pow(var[j] + delta, 3));
            avg_dB /= (float)d_zh.size();
            avg_dG /= (float)d_zh.size();
            d_beta.push_back(avg_dB);
            d_gamma.push_back(avg_dG);
        }
        int batchS = zws.size();
        betaGradients = d_beta;
        gammaGradients = d_gamma;
        denseA d_mean = denseA(zws[0].size());
        for (unsigned int j = 0; j < zws[0].size(); j++) { //this Layer
```

```
            float tmp1 = 0, tmp2 = 0;
            for (unsigned int i = 0; i < zws.size(); i++) { //batches
                tmp1 += m[j] - zws[i][j];
                tmp2 += d_u[i][j];
            }
            d_mean[j] = d_var[j] * 2 * tmp1 / ((float)batchS);
            d_mean[j] -= tmp2 / sqrt(var[j] + delta);
        }
        std::vector<denseA> d_z(zws.size(), denseA(zws[0].size()));
        for (unsigned int i = 0; i < d_z.size(); i++) {
            for (unsigned int j = 0; j < d_z[i].size(); j++) {
                d_z[i][j] = d_mean[j] / ((float)batchS);
                d_z[i][j] += d_var[j] * 2 * (zws[i][j] - m[j]) / (float)batchS;
                d_z[i][j] += d_u[i][j] / sqrt(var[j] + delta);
            }
        }
        dz = d_z;
    }
    else {
        dz = d_zh;
        betaGradients = std::vector<float>(0);
        for (unsigned int i = 0; i < dz[0].size(); i++) {
            float avgB = 0;
            for (unsigned int j = 0; j < dz.size(); j++) {
                avgB += dz[j][i];
            }
            avgB /= dz.size();
            betaGradients.push_back(avgB);
        }
    }
    std::vector<denseW> weightGradients = std::vector<denseW>(zws.size(), denseW(size, std::vect
or<float>(sl_1, 0)));
    for (unsigned int i = 0; i < weightGradients.size(); i++) { //i=batches
        for (unsigned int j = 0; j < weightGradients[i].size(); j++) { //j=thisLayer
            for (unsigned int k = 0; k < weightGradients[i][j].size(); k++) { //k=prevLayer
                weightGradients[i][j][k] = dz[i][j] * (*inputAct)[i][k];
            }
        }
    }
    wG = weightGradients;
    std::vector<denseA> gra(inputAct->size(), std::vector<float>((*inputAct)[0].size(), 0));
    for (unsigned int i = 0; i < gra.size(); i++) { //i=batches
        for (unsigned int j = 0; j < gra[i].size(); j++) { //j=prevLayer
            for (unsigned int k = 0; k < zws[0].size(); k++) { //k=thisLayer
                std::vector<float>* inPos = &((*inputPos)[j]),
                    * outPos = &(position[k]);
                float distance = MathCNN::distance(inPos, outPos);
                float weight = alphaM[k] * distance + betaM[k];
```

```
                gra[i][j] += weight * dz[i][k];
            }
        }
    }
    gradient = gra;
    //own backprop
    betaMGradients = std::vector<float>(size);
    alphaMGradients = std::vector<float>(size);
    for (unsigned int i = 0; i < betaMGradients.size(); i++) { //thisLayer
        float avgB = 0;
        float avgA = 0;
        for (unsigned int k = 0; k < zws.size(); k++) { //batches
            float sumB = 0;
            float sumA = 0;
            for (unsigned int j = 0; j < sl_1; j++) { //prevLayer
                sumB += weightGradients[k][i][j];
                std::vector<float>* inPos = &((*inputPos)[j]),
                                   * outPos = &(position[i]);
                float distance = MathCNN::distance(inPos, outPos);
                sumA += weightGradients[k][i][j] * distance;
            }
            avgB += sumB;
            avgA += sumA;
        }
        avgB /= zws.size();
        avgA /= zws.size();
        betaMGradients[i] = avgB;
        alphaMGradients[i] = avgA;
    }
    for (unsigned int i = 0; i < size; i++) { //this Layer
        for (unsigned int j = 0; j < dimensions; j++) { //dimensions
            float avgC = 0;
            for (unsigned int k = 0; k < zws.size(); k++) { //batches
                float sum = 0;
                for (unsigned int l = 0; l < sl_1; l++) { //prev Layer
                    float tmp = weightGradients[k][i][l];
                    std::vector<float>* inPos = &((*inputPos)[l]),
                                       * outPos = &(position[i]);
                    float distance = MathCNN::distance(inPos, outPos);
                    tmp *= alphaM[i];
                    tmp *= (position[i][j] - (*inputPos)[l][j]) / distance;
                    sum += tmp;
                }
                avgC += sum;
            }
            avgC /= zws.size();
            positionGradients[i][j] = avgC;
        }
```

```
    }
    if (!lastLayer && nextW) {
        for (unsigned int i = 0; i < size; i++) { //this Layer
            for (unsigned int j = 0; j < dimensions; j++) { //dimensions
                float avgC = 0;
                for (unsigned int k = 0; k < zws.size(); k++) { //batches
                    float sum = 0;
                    for (unsigned int l = 0; l < posp1->size(); l++) { //next Layer
                        float tmp = (*wGp1)[k][l][i];
                        std::vector<float>* inPos = &(position[i]),
                            * outPos = &((*posp1)[l]);
                        float distance = MathCNN::distance(inPos, outPos);
                        tmp *= (*ap1)[l];
                        tmp *= (position[i][j] - (*posp1)[l][j]) / distance;
                        sum += tmp;
                    }
                    avgC += sum;
                }
                avgC /= zws.size();
                positionGradients[i][j] += avgC;
            }
        }
    }
    updateParameters();
};
void MultiDense2::initialiseParameters(std::string actF) {
    for (int i = 0; i < size; i++) {
        alphaM.push_back((float)1);
        betaM.push_back((float)-1 * meanDistance); //-alpha*distance
        gamma.push_back((float)1);
        beta.push_back((float)0);
    }
    float stddev = 0;
    if (actF == "ReLU") {
        stddev = 1 / (1 * sqrt((float)sl_1)); // 1/alpha*sqrt(n)
    }
    srand((unsigned)time(NULL));
    const float PI = 3.141592741;
    std::default_random_engine generator;
    for (unsigned int i = 0; i < position.size(); i++) {
        if (ixLayer % 2 != 0) {
            float distance = MathCNN::getRandomNormal(&meanDistance, &stddev, &generator);
            float u = (float)rand() / RAND_MAX;
            float phi = 2 * PI * u;
            u = (float)rand() / RAND_MAX;
            float theta = 2 * PI * u;
            position[i][0] = distance * cos(phi) * cos(theta);
            position[i][1] = distance * cos(phi) * sin(theta);
```

```
            position[i][2] = -1 * distance * sin(phi);
        }
        else {
            float mean = 0;
            position[i][0] = MathCNN::getRandomNormal(&mean, &stddev, &generator);
            position[i][1] = MathCNN::getRandomNormal(&mean, &stddev, &generator);
            position[i][2] = MathCNN::getRandomNormal(&mean, &stddev, &generator);
        }
    }

    betaVelocity1 = std::vector<float>(beta.size());
    betaVelocity2 = std::vector<float>(beta.size());
    gammaVelocity1 = std::vector<float>(gamma.size());
    gammaVelocity2 = std::vector<float>(gamma.size());
    betaMVelocity1 = std::vector<float>(betaM.size());
    betaMVelocity2 = std::vector<float>(betaM.size());
    alphaMVelocity1 = std::vector<float>(alphaM.size());
    alphaMVelocity2 = std::vector<float>(alphaM.size());
    positionVelocity1 = std::vector<std::vector<float>>(size, std::vector<float>(dimensions));
    positionVelocity2 = std::vector<std::vector<float>>(size, std::vector<float>(dimensions));
    positionGradients = std::vector<std::vector<float>>(size, std::vector<float>(dimensions));
};
void MultiDense2::setUpGradient(std::vector<denseA>* upGradient_) {
    upGradient = upGradient_;
};
void MultiDense2::setInputPosition(std::vector<std::vector<float>>* inputPos_) {
    inputPos = inputPos_;
};
void MultiDense2::updateParameters() {
    if (!lastLayer) {
        for (unsigned int i = 0; i < beta.size(); i++) {
            betaVelocity1[i] = beta_o1 * betaVelocity1[i] + ((1 - beta_o1) * betaGradients[i]);
            betaVelocity2[i] = beta_o2 * betaVelocity2[i] + ((1 - beta_o2) * pow(betaGradients[i
], 2));
            gammaVelocity1[i] = beta_o1 * gammaVelocity1[i] + ((1 - beta_o1) * gammaGradients[i]
);
            gammaVelocity2[i] = beta_o2 * gammaVelocity2[i] + ((1 - beta_o2) * pow(gammaGradient
s[i], 2));
        }
        for (unsigned int i = 0; i < beta.size(); i++) {
            float betaV1_h = betaVelocity1[i] / (1 - pow(beta_o1, epoch));
            float betaV2_h = betaVelocity2[i] / (1 - pow(beta_o2, epoch));
            float gammaV1_h = gammaVelocity1[i] / (1 - pow(beta_o1, epoch));
            float gammaV2_h = gammaVelocity2[i] / (1 - pow(beta_o2, epoch));
            beta[i] -= eta * (betaV1_h / sqrt(betaV2_h + 0.00001));
            gamma[i] -= eta * (gammaV1_h / sqrt(gammaV2_h + 0.00001));
        }
    }
```

```
    else { //lastLayer
        for (unsigned int i = 0; i < beta.size(); i++) {
            betaVelocity1[i] = beta_o1 * betaVelocity1[i] + ((1 - beta_o1) * betaGradients[i]);
            betaVelocity2[i] = beta_o2 * betaVelocity2[i] + ((1 - beta_o2) * pow(betaGradients[i
], 2));
        }
        for (unsigned int i = 0; i < beta.size(); i++) {
            float betaV1_h = betaVelocity1[i] / (1 - pow(beta_o1, epoch));
            float betaV2_h = betaVelocity2[i] / (1 - pow(beta_o2, epoch));
            beta[i] -= eta * (betaV1_h / sqrt(betaV2_h + 0.00001));
        }
    }
    for (unsigned int i = 0; i < alphaMGradients.size(); i++) {
        betaMVelocity1[i] = beta_o1 * betaMVelocity1[i] + ((1 - beta_o1) * betaMGradients[i]);
        betaMVelocity2[i] = beta_o2 * betaMVelocity2[i] + ((1 - beta_o2) * pow(betaMGradients[i]
, 2));
        alphaMVelocity1[i] = beta_o1 * alphaMVelocity1[i] + ((1 - beta_o1) * alphaMGradients[i])
;
        alphaMVelocity2[i] = beta_o2 * alphaMVelocity2[i] + ((1 - beta_o2) * pow(alphaMGradients
[i], 2));
    }
    for (unsigned int i = 0; i < alphaMGradients.size(); i++) {
        float betaMV1_h = betaMVelocity1[i] / (1 - pow(beta_o1, epoch));
        float betaMV2_h = betaMVelocity2[i] / (1 - pow(beta_o2, epoch));
        float alphaMV1_h = alphaMVelocity1[i] / (1 - pow(beta_o1, epoch));
        float alphaMV2_h = alphaMVelocity2[i] / (1 - pow(beta_o2, epoch));
        betaM[i] -= eta * (betaMV1_h / sqrt(betaMV2_h + 0.00001));
        alphaM[i] -= eta * (alphaMV1_h / sqrt(alphaMV2_h + 0.00001));
    }
    for (unsigned int i = 0; i < positionGradients.size(); i++) {
        for (unsigned int j = 0; j < positionGradients[i].size(); j++) {
            positionVelocity1[i][j] = beta_o1 * positionVelocity1[i][j] + ((1 - beta_o1) * posit
ionGradients[i][j]);
            positionVelocity2[i][j] = beta_o2 * positionVelocity2[i][j] + ((1 - beta_o2) * pow(p
ositionGradients[i][j], 2));
        }
    }
    for (unsigned int i = 0; i < position.size(); i++) {
        for (unsigned int j = 0; j < position[i].size(); j++) {
            float positionV1_h = positionVelocity1[i][j] / (1 - pow(beta_o1, epoch));
            float positionV2_h = positionVelocity2[i][j] / (1 - pow(beta_o2, epoch));
            position[i][j] -= eta * (positionV1_h / sqrt(positionV2_h + 0.00001));
        }
    }
    epoch++;
};
void MultiDense2::endTraining() {
    converged = true;
```

```
    betaGradients.clear();
    betaGradients.shrink_to_fit();
    betaVelocity1.clear();
    betaVelocity1.shrink_to_fit();
    betaVelocity2.clear();
    betaVelocity2.shrink_to_fit();
    gammaGradients.clear();
    gammaGradients.shrink_to_fit();
    gammaVelocity1.clear();
    gammaVelocity1.shrink_to_fit();
    gammaVelocity2.clear();
    gammaVelocity2.shrink_to_fit();
    betaMGradients.clear();
    betaMGradients.shrink_to_fit();
    betaMVelocity1.clear();
    betaMVelocity1.shrink_to_fit();
    betaMVelocity2.clear();
    betaMVelocity2.shrink_to_fit();
    alphaMGradients.clear();
    alphaMGradients.shrink_to_fit();
    alphaMVelocity1.clear();
    alphaMVelocity1.shrink_to_fit();
    alphaMVelocity2.clear();
    alphaMVelocity2.shrink_to_fit();
    positionGradients.clear();
    positionGradients.shrink_to_fit();
    positionVelocity1.clear();
    positionVelocity1.shrink_to_fit();
    positionVelocity2.clear();
    positionVelocity2.shrink_to_fit();
    gradient.clear();
    gradient.shrink_to_fit();
    activations.clear();
    activations.shrink_to_fit();
}

denseA MultiDense2::getAvgMean() {
    denseA avg = denseA(avgM);
    for (unsigned int i = 0; i < zws[0].size(); i++) {
        avg[i] /= (epoch - 1);
    }
    return avg;
}

denseA MultiDense2::getAvgVar() {
    float m = zws.size();
    denseA avg = denseA(avgVar);
    for (unsigned int i = 0; i < zws[0].size(); i++) {
```

```
        avg[i] *= m / ((m - 1) * (epoch - 1));
    }
    return avg;
}
```

MultiDense2.h

```
#ifndef _MultiDense2H_
#define _MultiDense2H_

#include "Layer.h"
#include <string>

class MultiDense2 : public Layer {
private:
    int dimensions = 3;
    int size, sl_1;
    std::vector<denseA>* inputAct;
    std::vector<std::vector<float>>* inputPos;
    std::vector<denseA>* upGradient;

    float delta = 0.0001;
    std::vector<denseA> zws;
    std::vector<denseA> u;
    std::vector<denseA> zws_hat;
    denseA m, var;
    float deltaM = 0.1;
    float meanDistance = 1;

    denseA betaGradients;
    denseA betaVelocity1;
    denseA betaVelocity2;
    denseA gammaGradients;
    denseA gammaVelocity1;
    denseA gammaVelocity2;
    denseA betaMGradients;
    denseA betaMVelocity1;
    denseA betaMVelocity2;
    denseA alphaMGradients;
    denseA alphaMVelocity1;
    denseA alphaMVelocity2;
    std::vector<std::vector<float>> positionGradients;
    std::vector<std::vector<float>> positionVelocity1;
    std::vector<std::vector<float>> positionVelocity2;

    void updateParameters();
    float eta = 0.0001,
        beta_o1 = 0.9,
        beta_o2 = 0.999;
    int epoch = 1;
```

```
public:
    bool converged = false;
    denseA avgM, avgVar;
    denseA betaM;
    denseA alphaM;
    denseA gamma;
    denseA beta;
    std::vector<std::vector<float>> position;
    int ixLayer;
    bool lastLayer = false,
        nextW = true;
    float mean, * meanl_1;
    std::vector<denseA> activations;
    std::vector<denseA> gradient;
    std::vector<denseW> wG;
    std::vector<denseW>* wGp1;
    std::vector<float>* ap1;
    std::vector<std::vector<float>>* posp1;
    MultiDense2(int size, std::vector<denseA>* inputAct, std::vector<std::vector<float>>* inputP
os, float* meanl_1, int ixLayer);
    void forward();
    void backward();
    void initialiseParameters(std::string actF);
    void setUpGradient(std::vector<denseA>* upGradient);
    void setInputPosition(std::vector<std::vector<float>>* inputPos);
    void endTraining();
    denseA getAvgMean();
    denseA getAvgVar();
};

#endif;
```

MultiDenseInput.cpp

```
#include "MultiDenseInput.h"
#include "MathCNN.h"
#include <random>

MultiDenseInput::MultiDenseInput(std::vector<denseA>* input_) : inputAct(input_) {
    size = (*inputAct)[0].size();
    sl_1 = size;
    activations = std::vector<denseA>(1, denseA(size));
    gradient = std::vector<denseA>(0);
    position.reserve(size);
    position = std::vector<std::vector<float>>(size, std::vector<float>(dimensions));
    upGradient = nullptr;
};
void MultiDenseInput::forward() {
    activations = *inputAct;
```

```
};
void MultiDenseInput::backward() {
    for (unsigned int i = 0; i < size; i++) { //this Layer
        for (unsigned int j = 0; j < dimensions; j++) { //dimensions
            float avgC = 0;
            for (unsigned int k = 0; k < activations.size(); k++) { //batches
                float sum = 0;
                for (unsigned int l = 0; l < posp1->size(); l++) { //next Layer
                    float tmp = (*wGp1)[k][l][i];
                    std::vector<float>* inPos = &(position[i]),
                                      * outPos = &((*posp1)[l]);
                    float distance = MathCNN::distance(inPos, outPos);
                    if (!v2) {
                        tmp *= (-1) * (*ap1)[l] / (2 * sqrt(distance + deltaM));
                    }
                    else {
                        tmp *= (*ap1)[l];
                    }
                    tmp *= (position[i][j] - (*posp1)[l][j]) / distance;
                    sum += tmp;
                }
                avgC += sum;
            }
            avgC /= activations.size();
            positionGradients[i][j] += avgC;
        }

    }
    gradient = *upGradient;
    updateParameters();
};
void MultiDenseInput::initialiseParameters(std::string actF) {
    if (v2) {
        float mean = (float)0;
        float dev = 1;
        if (actF == "ReLU") {
            dev = 1 / (1 * sqrt((float)sl_1)); // 1/alpha*sqrt(n)
        }
        std::default_random_engine generator;
        for (unsigned int i = 0; i < size; i++) {
            for (unsigned int j = 0; j < dimensions; j++) {
                position[i][j] = MathCNN::getRandomNormal(&mean, &dev, &generator);
            }
        }
    }
    positionGradients.reserve(size);
    positionVelocity1.reserve(size);
    positionVelocity2.reserve(size);
```

```
    positionGradients = std::vector<std::vector<float>>(size, std::vector<float>(dimensions));
    positionVelocity1 = std::vector<std::vector<float>>(size, std::vector<float>(dimensions));
    positionVelocity2 = std::vector<std::vector<float>>(size, std::vector<float>(dimensions));
};
void MultiDenseInput::setUpGradient(std::vector<denseA>* upGradient_) {
    upGradient = upGradient_;
};
void MultiDenseInput::updateParameters() {
    for (unsigned int i = 0; i < positionGradients.size(); i++) {
        for (unsigned int j = 0; j < positionGradients[i].size(); j++) {
            positionVelocity1[i][j] = beta_o1 * positionVelocity1[i][j] + ((1 - beta_o1) * posit
ionGradients[i][j]);
            positionVelocity2[i][j] = beta_o2 * positionVelocity2[i][j] + ((1 - beta_o2) * pow(p
ositionGradients[i][j], 2));
        }
    }
    for (unsigned int i = 0; i < position.size(); i++) {
        for (unsigned int j = 0; j < position[i].size(); j++) {
            float positionV1_h = positionVelocity1[i][j] / (1 - pow(beta_o1, epoch));
            float positionV2_h = positionVelocity2[i][j] / (1 - pow(beta_o2, epoch));
            position[i][j] -= eta * (positionV1_h / sqrt(positionV2_h + 0.00001));
        }
    }
};
void MultiDenseInput::endTraining() {
    positionGradients.clear();
    positionGradients.shrink_to_fit();
    positionVelocity1.clear();
    positionVelocity1.shrink_to_fit();
    positionVelocity2.clear();
    positionVelocity2.shrink_to_fit();
    gradient.clear();
    gradient.shrink_to_fit();
    activations.clear();
    activations.shrink_to_fit();
}
```

MultiDenseInput.h

```
#ifndef _MultiDenseInputH_
#define _MultiDenseInputH_

#include "Layer.h"

class MultiDenseInput : public Layer {
private:
    int dimensions = 3;
    int size, sl_1;
    std::vector<denseA>* inputAct;
    std::vector<denseA>* upGradient;
```

```
    std::vector<std::vector<float>> positionGradients;
    std::vector<std::vector<float>> positionVelocity1;
    std::vector<std::vector<float>> positionVelocity2;
    float deltaM = 0.1;
    void updateParameters();
    float eta = 0.0001,
        beta_o1 = 0.9,
        beta_o2 = 0.999;
    int epoch = 1;
public:
    bool v2 = false;
    std::vector<denseA> activations;
    std::vector<denseA> gradient;
    std::vector<std::vector<float>> position;
    std::vector<denseW> wG;
    std::vector<denseW>* wGp1;
    std::vector<float>* ap1;
    std::vector<std::vector<float>>* posp1;
    MultiDenseInput(std::vector<denseA>* input);
    void forward();
    void backward();
    void initialiseParameters(std::string actF);
    void setUpGradient(std::vector<denseA>* upGradient);
    void endTraining();
};

#endif
```

Network.cpp

```
#include "Network.h"
#include "FileManager.h"
#include <typeinfo>
#include <fstream>
#include <iostream>
#include <regex>
#include <math.h>
#include <algorithm>
#include <filesystem>
#include <stxxl/vector>
#include "jExporter.h"

Network::Network() {
    std::ifstream inFile;
    do {
        std::string filename;
        inFile.open("labels.txt");
        if (!inFile) std::cout << "Datei existiert nicht. Erneut eingeben. "; std::cin;
    } while (!inFile);
    std::string fileText = "";
```

```
    while (!inFile.eof()) {
        std::string tmp;
        getline(inFile, tmp);
        fileText += tmp;
    }
    inFile.close();
    std::regex regex1("\\;");
    std::vector<std::string> split1(
        std::sregex_token_iterator(fileText.begin(), fileText.end(), regex1, -1),
        std::sregex_token_iterator()
    );
    for (unsigned int i = 0; i < split1.size(); i++) {
        std::string tmp = split1[i];
        std::regex regex2("\\,");
        std::vector<std::string> split2(
            std::sregex_token_iterator(tmp.begin(), tmp.end(), regex2, -1),
            std::sregex_token_iterator()
        );
        synsetID.push_back(std::vector<std::string>({split2[0], split2[1]}));
    }
};
Network::~Network() {
    for (unsigned int i = 0; i < level.size(); i++) {
        for (unsigned int j = 0; j < level[i].size(); j++) {
            delete level[i][j];
        }
    }
};
int Network::addLevel() {
    level.push_back(std::vector<Layer*>());
    return level.size() - 1;
};
int Network::addLayer(std::string layerType, int size, int inputLevel, int inputLayer) {
    if (layerType == "Dense") {
        Layer* layer = level[inputLevel][inputLayer];
        if (typeid(*layer) == typeid(Dense)) {
            Dense* inLayer = (dynamic_cast<Dense*>(layer));
            Dense* dense = new Dense(size, &(inLayer->activations));
            inLayer->setUpGradient(&(dense->gradient));
            level[level.size() - 1].push_back(dense);
            return level[level.size() - 1].size() - 1;
        } else if (typeid(*layer) == typeid(Flatten)) {
            Flatten* inLayer = (dynamic_cast<Flatten*>(layer));
            Dense* dense = new Dense(size, &(inLayer->activations));
            inLayer->setUpGradient(&(dense->gradient));
            level[level.size() - 1].push_back(dense);
            return level[level.size() - 1].size() - 1;
        } else if (typeid(*layer) == typeid(MultiDense)) {
```

```
            MultiDense* inLayer = (dynamic_cast<MultiDense*>(layer));
            Dense* dense = new Dense(size, &(inLayer->activations));
            inLayer->setUpGradient(&(dense->gradient));
            inLayer->nextW = false;
            level[level.size() - 1].push_back(dense);
            return level[level.size() - 1].size() - 1;
        }
    } else if (layerType == "MultiDense") {
        Layer* layer = level[inputLevel][inputLayer];
        if (typeid(*layer) == typeid(MultiDense)) {
            MultiDense* inLayer = (dynamic_cast<MultiDense*>(layer));
            MultiDense* multiDense = new MultiDense(size, &(inLayer->activations), &(inLayer-
>position), &(inLayer->mean), inLayer->ixLayer + 1);
            inLayer->setUpGradient(&(multiDense->gradient));
            inLayer->wGp1 = &(multiDense->wG);
            inLayer->ap1 = &(multiDense->alphaM);
            inLayer->posp1 = &(multiDense->position);
            level[level.size() - 1].push_back(multiDense);
            return level[level.size() - 1].size() - 1;
        } else if (typeid(*layer) == typeid(MultiDenseInput)) {
            MultiDenseInput* inLayer = (dynamic_cast<MultiDenseInput*>(layer));
            float *mean = new float();
            *mean = 0;
            MultiDense* multiDense = new MultiDense(size, &(inLayer->activations), &(inLayer-
>position), mean, 1);
            inLayer->setUpGradient(&(multiDense->gradient));
            inLayer->wGp1 = &(multiDense->wG);
            inLayer->ap1 = &(multiDense->alphaM);
            inLayer->posp1 = &(multiDense->position);
            level[level.size() - 1].push_back(multiDense);
            return level[level.size() - 1].size() - 1;
        }
    } else if (layerType == "MultiDense2") {
        Layer* layer = level[inputLevel][inputLayer];
        if (typeid(*layer) == typeid(MultiDense2)) {
            MultiDense2* inLayer = (dynamic_cast<MultiDense2*>(layer));
            MultiDense2* multiDense = new MultiDense2(size, &(inLayer->activations), &(inLayer-
>position), &(inLayer->mean), inLayer->ixLayer + 1);
            inLayer->setUpGradient(&(multiDense->gradient));
            inLayer->wGp1 = &(multiDense->wG);
            inLayer->ap1 = &(multiDense->alphaM);
            inLayer->posp1 = &(multiDense->position);
            level[level.size() - 1].push_back(multiDense);
            return level[level.size() - 1].size() - 1;
        }
        else if (typeid(*layer) == typeid(MultiDenseInput)) {
            MultiDenseInput* inLayer = (dynamic_cast<MultiDenseInput*>(layer));
            float* mean = new float();
```

```
            *mean = 0;
            MultiDense2* multiDense = new MultiDense2(size, &(inLayer->activations), &(inLayer-
>position), mean, 1);
            inLayer->setUpGradient(&(multiDense->gradient));
            inLayer->wGp1 = &(multiDense->wG);
            inLayer->ap1 = &(multiDense->alphaM);
            inLayer->posp1 = &(multiDense->position);
            inLayer->v2 = true;
            level[level.size() - 1].push_back(multiDense);
            return level[level.size() - 1].size() - 1;
        }
    }
    return -1;
};
int Network::addLayer(std::string layerType, int actMaps, int inputLevel, int inputLayer, int ke
rnel, int stride, int padding) {
    if (layerType == "Conv") {
        Layer* layer = level[inputLevel][inputLayer];
        if (typeid(*layer) == typeid(Conv)) {
            Conv* inLayer = (dynamic_cast<Conv*>(layer));
            Conv* conv = new Conv(actMaps, &(inLayer->activations), kernel, stride, padding);
            inLayer->setUpGradient(&(conv->gradient));
            level[level.size() - 1].push_back(conv);
            return level[level.size() - 1].size() - 1;
        } else if (typeid(*layer) == typeid(Input)) {
            Input* inLayer = (dynamic_cast<Input*>(layer));
            Conv* conv = new Conv(actMaps, &(inLayer->activations), kernel, stride, padding);
            level[level.size() - 1].push_back(conv);
            return level[level.size() - 1].size() - 1;
        } else if (typeid(*layer) == typeid(Pooling)) {
            Pooling* inLayer = (dynamic_cast<Pooling*>(layer));
            Conv* conv = new Conv(actMaps, &(inLayer->activations), kernel, stride, padding);
            inLayer->setUpGradient(&(conv->gradient));
            level[level.size() - 1].push_back(conv);
            return level[level.size() - 1].size() - 1;
        }
    }
    return -1;
};
int Network::addLayer(int kernel, std::string layerType, int inputLevel, int inputLayer) {
    if (layerType == "Pooling") {
        Layer* layer = level[inputLevel][inputLayer];
        if (typeid(*layer) == typeid(Conv)) {
            Conv* inLayer = (dynamic_cast<Conv*>(layer));
            Pooling* pooling = new Pooling(&(inLayer->activations), kernel);
            inLayer->setUpGradient(&(pooling->gradient));
            inLayer->disableBN();
            level[level.size() - 1].push_back(pooling);
```

```
            return level[level.size() - 1].size() - 1;
        }
    }
    return -1;
};
int Network::addLayer(std::string layerType, int inputLevel, int inputLayer) {
    if (layerType == "Flatten") {
        Layer* layer = level[inputLevel][inputLayer];
        if (typeid(*layer) == typeid(Conv)) {
            Conv* inLayer = (dynamic_cast<Conv*>(layer));
            Flatten* flatten = new Flatten(&(inLayer->activations));
            inLayer->setUpGradient(&(flatten->gradient));
            level[level.size() - 1].push_back(flatten);
            return level[level.size() - 1].size() - 1;
        } else if (typeid(*layer) == typeid(Input)) {
            Input* inLayer = (dynamic_cast<Input*>(layer));
            Flatten* flatten = new Flatten(&(inLayer->activations));
            level[level.size() - 1].push_back(flatten);
            return level[level.size() - 1].size() - 1;
        } else if (typeid(*layer) == typeid(Pooling)) {
            Pooling* inLayer = (dynamic_cast<Pooling*>(layer));
            Flatten* flatten = new Flatten(&(inLayer->activations));
            inLayer->setUpGradient(&(flatten->gradient));
            level[level.size() - 1].push_back(flatten);
            return level[level.size() - 1].size() - 1;
        }
    } else if (layerType == "Softmax") {
        Layer* layer = level[inputLevel][inputLayer];
        if (typeid(*layer) == typeid(Dense)) {
            Dense* inLayer = (dynamic_cast<Dense*>(layer));
            Softmax* softmax = new Softmax(&(inLayer->activations));
            inLayer->setUpGradient(&(softmax->gradient));
            inLayer->lastLayer = true;
            level[level.size() - 1].push_back(softmax);
            return level[level.size() - 1].size() - 1;
        } else if (typeid(*layer) == typeid(Flatten)) {
            Flatten* inLayer = (dynamic_cast<Flatten*>(layer));
            Softmax* softmax = new Softmax(&(inLayer->activations));
            inLayer->setUpGradient(&(softmax->gradient));
            level[level.size() - 1].push_back(softmax);
            return level[level.size() - 1].size() - 1;
        } else if (typeid(*layer) == typeid(MultiDense)) {
            MultiDense* inLayer = (dynamic_cast<MultiDense*>(layer));
            Softmax* softmax = new Softmax(&(inLayer->activations));
            inLayer->setUpGradient(&(softmax->gradient));
            inLayer->lastLayer = true;
            level[level.size() - 1].push_back(softmax);
            return level[level.size() - 1].size() - 1;
```

```
        } else if (typeid(*layer) == typeid(MultiDense2)) {
            MultiDense2* inLayer = (dynamic_cast<MultiDense2*>(layer));
            Softmax* softmax = new Softmax(&(inLayer->activations));
            inLayer->setUpGradient(&(softmax->gradient));
            inLayer->lastLayer = true;
            level[level.size() - 1].push_back(softmax);
            return level[level.size() - 1].size() - 1;
        }
    } else if (layerType == "MultiDenseInput") {
        Layer* layer = level[inputLevel][inputLayer];
        if (typeid(*layer) == typeid(Dense)) {
            Dense* inLayer = (dynamic_cast<Dense*>(layer));
            MultiDenseInput* multiDenseInput = new MultiDenseInput(&(inLayer->activations));
            inLayer->setUpGradient(&(multiDenseInput->gradient));
            level[level.size() - 1].push_back(multiDenseInput);
            return level[level.size() - 1].size() - 1;
        }
        else if (typeid(*layer) == typeid(Flatten)) {
            Flatten* inLayer = (dynamic_cast<Flatten*>(layer));
            MultiDenseInput* multiDenseInput = new MultiDenseInput(&(inLayer->activations));
            inLayer->setUpGradient(&(multiDenseInput->gradient));
            level[level.size() - 1].push_back(multiDenseInput);
            return level[level.size() - 1].size() - 1;
        }
    }
    return -1;
};
int Network::addLayer(std::string layer, int* size) {
    if (layer == "Input" && level.size() == 1) {
        Input* input = new Input(size);
        level[level.size() - 1].push_back(input);
        return level[level.size() - 1].size() - 1;
    }
    return -1;
};
void Network::intialiseParameters() {
    for (unsigned int i = 0; i < level.size(); i++) {
        for (unsigned int j = 0; j < level[i].size(); j++) {
            level[i][j]->initialiseParameters(actF);
        }
    }
}
void Network::setActivationFunction(std::string actFunction) {
    actF = actFunction;
};
void Network::setInput(vconvAL* in) {
    for (unsigned int i = 0; i < (*in).size(); i++) {
        (*in)[i] = preprocess(&((*in)[i]));
```

```
    }
    for (unsigned int i = 0; i < (*in).size(); i++) {
        for (unsigned int j = 0; j < (*in)[i].size(); j++) {
            for (unsigned int k = 0; k < (*in)[i][j].size(); k++) {
                for (unsigned int l = 0; l < (*in)[i][j][k].size(); l++) {
                    (*in)[i][j][k][l] -= batchMean;
                }
            }
        }
    }
    Input* input = dynamic_cast<Input*>(level[0][0]);
    input->activations = *in;
};
std::vector<std::vector<std::string>> Network::predict() {
    for (unsigned int i = 0; i < level.size(); i++) {
        for (unsigned int j = 0; j < level[i].size(); j++) {
            level[i][j]->forward();
        }
    }
    Layer* lastLayer = level[level.size() - 1][0];
    if (typeid(*lastLayer) == typeid(Softmax)) {
        return encodeOutput(dynamic_cast<Softmax*>(lastLayer)->activations);
    } else if (typeid(*lastLayer) == typeid(Dense)) {
        return encodeOutput(dynamic_cast<Dense*>(lastLayer)->activations);
    } else if (typeid(*lastLayer) == typeid(MultiDense)) {
        return encodeOutput(dynamic_cast<MultiDense*>(lastLayer)->activations);
    }
    return std::vector<std::vector<std::string>>(0);
};
std::vector<denseA> Network::predictVal() {
    for (unsigned int i = 0; i < level.size(); i++) {
        for (unsigned int j = 0; j < level[i].size(); j++) {
            level[i][j]->forward();
        }
    }
    Layer* lastLayer = level[level.size() - 1][0];
    if (typeid(*lastLayer) == typeid(Softmax)) {
        return dynamic_cast<Softmax*>(lastLayer)->activations;
    }
    else if (typeid(*lastLayer) == typeid(Dense)) {
        return dynamic_cast<Dense*>(lastLayer)->activations;
    }
    else if (typeid(*lastLayer) == typeid(MultiDense)) {
        return dynamic_cast<MultiDense*>(lastLayer)->activations;
    }
    return std::vector<denseA>(0);
};
convAL Network::preprocess(convAL* img) {
```

```
    Input* inputL = dynamic_cast<Input*>(level[0][0]);
    int widthD = inputL->activations[0].size(),
        heightD = inputL->activations[0][0].size();
    int width = img[0].size(),
        height = img[0][0].size();
    convAL newImg;
    float scaleX = (float)width / (float)widthD;
    float scaleY = (float)height / (float)heightD;
    for (int x = 0; x < widthD; x++) {
        newImg.push_back(vectorff());
        for (int y = 0; y < heightD; y++) {
            newImg[x].push_back(vectorf());
            int pX = floor(scaleX * (float)x);
            int pY = floor(scaleY * (float)y);
            newImg[x][y] = (*img)[pX][pY];
        }
    }
    return newImg;
};
void Network::epoch(std::vector<std::vector<std::string>> *data) { //data = [[path, label], [...
], ...];
    vconvAL imgs;
    for (unsigned int i = 0; i < data->size(); i++) {
        convA img = FileManager::getInput((*data)[i][0]);
        imgs.push_back(img);
    }
    setInput(&imgs);
    std::vector<denseA> result = predictVal();
    std::vector<denseA> resultD;
    for (unsigned int i = 0; i < result.size(); i++) {
        resultD.push_back(denseA());
        for (unsigned int j = 0; j < result[i].size(); j++) {
            /*if ((*data)[i][1] == synsetID[j][0]) {
                resultD[i].push_back(1);
            } else {
                resultD[i].push_back(0);
            }*/
            if (j == stoi((*data)[i][1])) {
                resultD[i].push_back(1);
            } else {
                resultD[i].push_back(0);
            }
        }
    }
    std::vector<denseA> dCa;
    if (costF == "cross-entropy") {
        float avgError = 0;
        for (unsigned int i = 0; i < result.size(); i++) {
```

```
            float sum = 0;
            for (unsigned int j = 0; j < result[i].size(); j++) {
                sum += resultD[i][j] * log(result[i][j]);
            }
            sum *= -1;
            avgError += sum;
        }
        avgError /= result.size();
        for (unsigned int i = 0; i < result.size(); i++) {
            dCa.push_back(denseA());
            for (unsigned int j = 0; j < result[i].size(); j++) {
                dCa[i].push_back((-1)*resultD[i][j] / result[i][j]);
            }
        }
        std::cout << "error: " + std::to_string(avgError) << ", ";
        avgE += std::to_string(avgError) + ";";
    } else if(costF == "mean-squared") {}
    std::ofstream outfile("plotData.txt");
    outfile << avgE;
    outfile.close();
    Softmax* softmax = dynamic_cast<Softmax*>(level[level.size() - 1][0]);
    softmax->setUpGradient(&dCa);
    softmax->resultD = resultD;
    std::cout << "backward 00";
    for (unsigned int i = level.size() - 1; i >= 1; i--) {
        std::cout << std::string(std::to_string(i+1).length(), '\b') << std::string((std::to_str
ing(i + 1).length() > 1 && std::to_string(i).length() == 1) ? 1 : 0, ' ') << i;
        level[i][0]->backward();
    }
    std::cout << std::endl;
    std::string pos;
    bool plot2 = false;
    for (unsigned int i = 0; i < level.size(); i++) {
        for (unsigned int j = 0; j < level[i].size(); j++) {
            bool plot = false;
            std::vector<std::vector<float>>* positions = nullptr;
            if (typeid(*level[i][j]) == typeid(MultiDense)) {
                MultiDense* layer = dynamic_cast<MultiDense*>(level[i][j]);
                positions = &(layer->position);
                plot = true;
            } else if (typeid(*level[i][j]) == typeid(MultiDenseInput)) {
                MultiDenseInput* layer = dynamic_cast<MultiDenseInput*>(level[i][j]);
                positions = &(layer->position);
                plot = true;
            } else if (typeid(*level[i][j]) == typeid(MultiDense2)) {
                MultiDense2* layer = dynamic_cast<MultiDense2*>(level[i][j]);
                positions = &(layer->position);
                plot = true;
```

```
            }
            if (plot) {
                for (unsigned int k = 0; k < positions->size(); k++) {
                    std::string tmp = "";
                    for (unsigned int l = 0; l < (*positions)[k].size(); l++) {
                        tmp += std::to_string((*positions)[k][l]);
                        tmp += ',';
                    }
                    pos += tmp + ";";
                }
                plot2 = true;
            }
        }
    }
    if (plot2) {
        std::ofstream outfile2("posData.txt");
        outfile2 << pos;
        outfile2.close();
    }
};
void Network::setCostFunction(std::string costFct) {
    if (costF == "cross-entropy" || costF == "mean-squared") costF = costFct;
};
std::vector<std::vector<std::string>> Network::encodeOutput(std::vector<denseA> output) {
    std::vector<std::vector<std::string>> result;
    for (unsigned int i = 0; i < output.size(); i++) {
        std::vector<std::string> values;
        for (unsigned int j = 0; j < output[i].size(); j++) {
            float val = output[i][j];
            float percent = roundf(val * 10000) / 100;
            std::string st = "";
            if (percent < 10) st += "0";
            st += std::to_string(percent);
            std::string name = st.substr(0, st.size() - 4) + "% " + synsetID[j][1];
            values.push_back(name);
        }
        sort(values.begin(), values.end());
        reverse(values.begin(), values.end());
        result.push_back(values);
    }
    return result;
};
std::vector<denseA> Network::sortOutput(std::vector<denseA> output) {
    std::vector<denseA> ret = output;
    for (unsigned int i = 0; i < ret.size(); i++) {
        std::sort(ret[i].begin(), ret[i].end(), std::greater<float>());
    }
    return ret;
```

```
};
void Network::finishTraining() {
    Input* in = dynamic_cast<Input*>(level[0][0]);
    in->activations.clear();
    for (unsigned int i = 0; i < level.size(); i++) {
        for (unsigned int j = 0; j < level[i].size(); j++) {
            if (typeid(*level[i][j]) == typeid(Dense)) {
                Dense* layer = dynamic_cast<Dense*>(level[i][j]);
                layer->endTraining();
            } else if (typeid(*level[i][j]) == typeid(MultiDense)) {
                MultiDense* layer = dynamic_cast<MultiDense*>(level[i][j]);
                layer->endTraining();
            } else if (typeid(*level[i][j]) == typeid(MultiDenseInput)) {
                MultiDenseInput* layer = dynamic_cast<MultiDenseInput*>(level[i][j]);
                layer->endTraining();
            } else if (typeid(*level[i][j]) == typeid(Conv)) {
                Conv* layer = dynamic_cast<Conv*>(level[i][j]);
                layer->endTraining();
            } else if (typeid(*level[i][j]) == typeid(Flatten)) {
                Flatten* layer = dynamic_cast<Flatten*>(level[i][j]);
                layer->endTraining();
            }
        }
    }
};
void Network::exportNet(std::string filename) {
    std::filesystem::remove(filename);
    for (unsigned int i = 0; i < level.size(); i++) {
        for (unsigned int j = 0; j < level[i].size(); j++) {
            if (typeid(*level[i][j]) == typeid(Dense)) {
                Dense* layer = dynamic_cast<Dense*>(level[i][j]);
                denseExp data;
                data.weights = &layer->weights;
                data.beta = &layer->beta;
                data.gamma = &layer->gamma;
                denseA avgM = layer->getAvgMean();
                denseA avgVar = layer->getAvgVar();
                data.avgM = &avgM;
                data.avgVar = &avgVar;
                jExporter::exportData(filename, data);
            }
            else if (typeid(*level[i][j]) == typeid(Conv)) {
                Conv* layer = dynamic_cast<Conv*>(level[i][j]);
                convExp data;
                data.kernel = &layer->kernel;
                data.beta = &layer->beta;
                data.gamma = &layer->gamma;
                data.biases = &layer->biases;
```

```
        denseA avgM = layer->getAvgMean();
        denseA avgVar = layer->getAvgVar();
        data.avgM = &avgM;
        data.avgVar = &avgVar;
        jExporter::exportData(filename, data);
    }
    else if (typeid(*level[i][j]) == typeid(Input)) {
        Input* layer = dynamic_cast<Input*>(level[i][j]);
        inputExp data;
        data.batchMean = &batchMean;
        jExporter::exportData(filename, data);
    }
    else if (typeid(*level[i][j]) == typeid(MultiDense)) {
        MultiDense* layer = dynamic_cast<MultiDense*>(level[i][j]);
        multiDenseExp data;
        data.position = &layer->position;
        data.avgM = &layer->avgM;
        data.avgVar = &layer->avgVar;
        data.betaM = &layer->betaM;
        data.alphaM = &layer->alphaM;
        data.beta = &layer->beta;
        data.gamma = &layer->gamma;
        jExporter::exportData(filename, data);
    }
    else if (typeid(*level[i][j]) == typeid(MultiDense2)) {
        MultiDense2* layer = dynamic_cast<MultiDense2*>(level[i][j]);
        multiDenseExp data;
        data.position = &layer->position;
        data.avgM = &layer->avgM;
        data.avgVar = &layer->avgVar;
        data.betaM = &layer->betaM;
        data.alphaM = &layer->alphaM;
        data.beta = &layer->beta;
        data.gamma = &layer->gamma;
        jExporter::exportData(filename, data);
    }
    else if (typeid(*level[i][j]) == typeid(MultiDenseInput)) {
        MultiDenseInput* layer = dynamic_cast<MultiDenseInput*>(level[i][j]);
        multiDenseInputExp data;
        data.position = &layer->position;
        jExporter::exportData(filename, data);
    }
    else {
        std::ofstream out;
        out.open(filename, std::ios_base::app);
        out << "{}\n";
        out.close();
    }
```

```
        }
    }
};
void Network::importNet(std::string filename) {
    //std::cout << "reading..." << std::endl;
    std::vector<std::thread> readThreads;
    for (unsigned int i = 0; i < level.size(); i++) {
        for (unsigned int j = 0; j < level[i].size(); j++) {
            if (typeid(*level[i][j]) == typeid(Dense)) {
                Dense* layer = dynamic_cast<Dense*>(level[i][j]);
                readThreads.push_back(std::thread([layer, i, filename] {
                    jExporter::importData(i, filename, &(layer->weights), &(layer-
>beta), &(layer->gamma), &(layer->avgM), &(layer->avgVar));
                }));
                layer->converged = true;
            }
            else if (typeid(*level[i][j]) == typeid(Conv)) {
                Conv* layer = dynamic_cast<Conv*>(level[i][j]);
                readThreads.push_back(std::thread([layer, i, filename] {
                    jExporter::importData(i, filename, &(layer->kernel), &(layer-
>beta), &(layer->gamma), &(layer->biases), &(layer->avgM), &(layer->avgVar));
                }));
                layer->converged = true;
            }
            else if (typeid(*level[i][j]) == typeid(Input)) {
                Input* layer = dynamic_cast<Input*>(level[i][j]);
                jExporter::importData(i, filename, &batchMean);
            }
            else if (typeid(*level[i][j]) == typeid(MultiDense)) {
                MultiDense* layer = dynamic_cast<MultiDense*>(level[i][j]);
                jExporter::importData(i, filename,
                    &(layer->avgM),
                    &(layer->avgVar),
                    &(layer->betaM),
                    &(layer->alphaM),
                    &(layer->beta),
                    &(layer->gamma),
                    &(layer->position)
                );
                layer->converged = true;
            }
            else if (typeid(*level[i][j]) == typeid(MultiDense2)) {
                MultiDense2* layer = dynamic_cast<MultiDense2*>(level[i][j]);
                jExporter::importData(i, filename,
                    &(layer->avgM),
                    &(layer->avgVar),
                    &(layer->betaM),
                    &(layer->alphaM),
```

```
                    &(layer->beta),
                    &(layer->gamma),
                    &(layer->position)
                );
                layer->converged = true;
            }
            else if (typeid(*level[i][j]) == typeid(MultiDenseInput)) {
                MultiDenseInput* layer = dynamic_cast<MultiDenseInput*>(level[i][j]);
                jExporter::importData(i, filename, &(layer->position));
            }
        }
    }
    for (unsigned int i = 0; i < readThreads.size(); i++) {
        readThreads[i].join();
    }
};
void Network::calcBatchMean(int size, std::string dir) {
    int i = 0;
    for (const auto& entry : std::filesystem::recursive_directory_iterator(dir)) {
        if (!std::filesystem::is_directory(entry.path())) {
            std::string path{ entry.path().u8string() };
            convA img = FileManager::getInput(path);
            img = preprocess(&img);
            for (unsigned int j = 0; j < img.size(); j++) {
                for (unsigned int k = 0; k < img[j].size(); k++) {
                    for (unsigned int l = 0; l < img[j][k].size(); l++) {
                        batchMean += img[j][k][l];
                    }
                }
            }
        }
        i++;
        if (i == size) break;
    }
    Input* inputL = dynamic_cast<Input*>(level[0][0]);
    int width = inputL->activations[0].size(),
        height = inputL->activations[0][0].size(),
        channels = inputL->activations[0][0][0].size();
    batchMean /= size * width * height * channels;
};
```

Network.h

```
#ifndef _NetworkH_
#define _NetworkH_

#include "activations.h"
#include "Layer.h"
#include "Dense.h"
#include "Conv.h"
```

```
#include "Flatten.h"
#include "Softmax.h"
#include "Input.h"
#include "MultiDense.h"
#include "MultiDense2.h"
#include "MultiDenseInput.h"
#include "Pooling.h"
#include <string>
#include <vector>

class Network {
private:
    std::vector<std::vector<Layer*>> level;
    std::string actF = "ReLU";
    std::string costF = "cross-entropy";
    std::vector<std::vector<std::string>> encodeOutput(std::vector<denseA> output);
    convAL preprocess(convAL* img);
    float batchMean = 0;
    std::string avgE = "";
public:
    std::vector<std::vector<std::string>> synsetID;
    Network();
    ~Network();
    int addLevel();
    int addLayer(std::string layer, int size, int inputLevel, int inputLayer);
    int addLayer(std::string layerType, int actMaps, int inputLevel, int inputLayer, int kernel,
 int stride, int padding);
    int addLayer(int kernel, std::string layerType, int inputLevel, int inputLayer);
    int addLayer(std::string layer, int inputLevel, int inputLayer);
    int addLayer(std::string layer, int *size);
    void intialiseParameters();
    void setActivationFunction(std::string actFunction);
    void setInput(vconvAL* input);
    std::vector<std::vector<std::string>> predict();
    std::vector<denseA> predictVal();
    void epoch(std::vector<std::vector<std::string>> *data);
    void setCostFunction(std::string costFunction);
    std::vector<denseA> sortOutput(std::vector<denseA> output);
    void finishTraining();
    void exportNet(std::string filename);
    void importNet(std::string filename);
    void calcBatchMean(int batchSize, std::string imgPath);
};

#endif
```

Pooling.cpp

```
#include "Pooling.h"
```

```
Pooling::Pooling(vconvAL* input_, int kernel_) : input(input_), kernel(kernel_) {
    sl_1[0] = (*input)[0].size();
    sl_1[1] = (*input)[0][0].size();
    sl_1[2] = (*input)[0][0][0].size();
    size[0] = MathCNN::convSize(sl_1[0], kernel, 0, kernel);
    size[1] = MathCNN::convSize(sl_1[1], kernel, 0, kernel);
    size[2] = sl_1[2];
    activations.push_back(convA(size[0], std::vector<std::vector<float>>(size[1], std::vector<fl
oat>(size[2]))));

    upGradient = nullptr;
};
void Pooling::forward() {
    activations.clear();
    for (unsigned int i = 0; i < input->size(); i++) {
        activations.push_back(convA(size[0], std::vector<std::vector<float>>(size[1], std::vecto
r<float>(size[2]))));
    }
    for (unsigned int l = 0; l < input->size(); l++) { //batches
        for (unsigned int i = 0; i < size[2]; i++) { //depth
            for (unsigned int j = 0; j < size[0]; j++) { //width
                for (unsigned int k = 0; k < size[1]; k++) { //height
                    float avg = 0;
                    for (unsigned int k1 = 0; k1 < kernel; k1++) {
                        for (unsigned int k2 = 0; k2 < kernel; k2++) {
                            int J = k1 + j * kernel,
                                K = k2 + k * kernel; //stride = kernel
                            avg += (*input)[l][J][K][i];
                        }
                    }
                    avg /= kernel * kernel;
                    activations[l][j][k][i] = avg;
                }
            }
        }
    }
};
void Pooling::backward() {
    gradient.clear();
    for (unsigned int i = 0; i < input->size(); i++) {
        gradient.push_back(convA(sl_1[0], std::vector<std::vector<float>>(sl_1[1], std::vector<f
loat>(sl_1[2]))));
    }
    for (unsigned int l = 0; l < input->size(); l++) { //batches
        for (unsigned int i = 0; i < size[2]; i++) { //depth(l-1 = l)
            for (unsigned int j = 0; j < size[0]; j++) { //width
                for (unsigned int k = 0; k < size[1]; k++) { //height
                    for (unsigned int k1 = 0; k1 < kernel; k1++) {
```

```
                        for (unsigned int k2 = 0; k2 < kernel; k2++) {
                            int J = k1 + j * kernel,
                                K = k2 + k * kernel; //stride = kernel
                            gradient[l][J][K][i] = (*upGradient)[l][j][k][i] / (kernel * kernel)
;
                        }
                    }
                }
            }
        }
    }
};
void Pooling::initialiseParameters(std::string actF) {};
void Pooling::setUpGradient(vconvAL* upGradient_) {
    upGradient = upGradient_;
};
```

Pooling.h

```
#ifndef _PoolingH_
#define _PoolingH_

#include "Layer.h"
#include "activations.h"
#include "MathCNN.h"

class Pooling : public Layer {
private:
    int size[3];
    int sl_1[3];
    int kernel;
    vconvAL* input;
    vconvAL* upGradient;
public:
    vconvAL activations;
    vconvAL gradient;
    Pooling(vconvAL* input, int kernel);
    void forward();
    void backward();
    void initialiseParameters(std::string actF);
    void setUpGradient(vconvAL* upGradient);
};

#endif
```

Softmax.cpp

```
#include "Softmax.h"
#include "MathCNN.h"
#include <iostream>

Softmax::Softmax(std::vector<denseA> *input_) : input(input_) {
```

```
    size = (*input)[0].size();
    sl_1 = size;
    upGradient = nullptr;
    activations = std::vector<denseA>(1, denseA(size));
};
void Softmax::forward() {
    activations = std::vector<denseA>(input->size(), denseA(size));
    for (unsigned int i = 0; i < input->size(); i++) {
        MathCNN::softmax(&(*input)[i], &activations[i]);
    }
};
void Softmax::backward() {
    gradient = std::vector<denseA>(0);
    for (unsigned int i = 0; i < activations.size(); i++) {
        gradient.push_back(MathCNN::subtractVec(&activations[i], &resultD[i]));
    }
};
void Softmax::initialiseParameters(std::string actF) {};
void Softmax::setUpGradient(std::vector<denseA>* upG) {
    upGradient = upG;
};
```

Softmax.h

```
#ifndef _SoftmaxH_
#define _SoftmaxH_

#include "Layer.h"
#include "Dense.h"

class Softmax : public Layer {
private:
    int size, sl_1;
    std::vector<denseA>* input;
    std::vector<denseA>* upGradient;
public:
    std::vector<denseA> resultD;
    std::vector<denseA> activations;
    std::vector<denseA> gradient;
    Softmax(std::vector<denseA>* input);
    void forward();
    void backward();
    void initialiseParameters(std::string actF);
    void setUpGradient(std::vector<denseA>* upGradient);
};

#endif
```

Trainer.cpp

```
#include "Trainer.h"
```

```
#include "FileManager.h"
#include <filesystem>
#include <iostream>
#include <regex>

Trainer::Trainer(Network* net) : network(net) {};
void Trainer::setBatchSize(int size) {
    if (size > 0) batchSize = size;
};
void Trainer::train(std::string path, int iteration) {
    system("mkdir results");
    network->calcBatchMean(128, path);
    int i = 1,
        epoch = 0;
    std::vector<std::vector<std::string>> data(0);
    for (const auto& entry : std::filesystem::recursive_directory_iterator(path)) {
        if (!std::filesystem::is_directory(entry.path())) {
            std::string path{ entry.path().u8string() };
            std::string label, filename;
            std::regex regex1("\\\\");
            std::vector<std::string> out1(
                std::sregex_token_iterator(path.begin(), path.end(), regex1, -1),
                std::sregex_token_iterator()
            );
            filename = out1[out1.size() - 1];
            std::regex regex2("\\_");
            std::vector<std::string> out2(
                std::sregex_token_iterator(filename.begin(), filename.end(), regex2, -1),
                std::sregex_token_iterator()
            );
            label = out2[1];
            data.push_back(std::vector<std::string>({path, label}));
            if (i == batchSize) {
                std::cout << "epoch: " + std::to_string(epoch + 1) + ", " + "iteration " + std::
to_string(iteration + 1) + ", ";
                network->epoch(&data);
                data = std::vector<std::vector<std::string>>(0);
                epoch++;
                i = 0;
                if (epoch % 1 == 0) network-
>exportNet("results/converged" + std::to_string(epoch) + ";" + std::to_string(iteration) + ".txt
");
            }
            i++;
        }
    }
};
void Trainer::test(std::string networkPath, std::string testPath) {
```

```
    int sequenceSize = 4,
        maximumImgs = 4;
    vconvAL data;
    std::vector<std::string> labels;
    std::vector<int> quantity;
    std::vector<std::vector<std::vector<float>>> testRes; //testRes[iteration][epoch][acc/conf]
    for (const auto& entry : std::filesystem::recursive_directory_iterator(networkPath)) {
        if (!std::filesystem::is_directory(entry.path())) {
            std::string netPath{ entry.path().u8string() };
            std::string filename;
            std::regex regex1("\\\\");
            std::vector<std::string> out1(
                std::sregex_token_iterator(netPath.begin(), netPath.end(), regex1, -1),
                std::sregex_token_iterator()
            );
            filename = out1[out1.size() - 1];
            std::regex_iterator<std::string::iterator>::regex_type reg("[0-9]+");
            std::regex_iterator<std::string::iterator> next(filename.begin(), filename.end(), re
g), end;
            std::vector<int> res;
            for (; next != end; next++) {
                res.push_back(std::stoi(next->str()));
            }
            int epoch = res[0],
                iteration = res[1];
            if (iteration >= quantity.size()) {
                for (unsigned int i = quantity.size(); i <= iteration; i++) {
                    quantity.push_back(0);
                }
            }
            quantity[iteration] = quantity[iteration] < epoch ? epoch : quantity[iteration];
        }
    }
    for (unsigned int i = 0; i < quantity.size(); i++) {
        testRes.push_back(std::vector<std::vector<float>>());
        for (unsigned int j = 0; j < quantity[i]; j++) {
            testRes[i].push_back(std::vector<float>());
        }
    }
    for (const auto& entry : std::filesystem::recursive_directory_iterator(networkPath)) {
        if (!std::filesystem::is_directory(entry.path())) {
            std::string netPath{ entry.path().u8string() };
            network->importNet(netPath);
            int i = 0;
            std::vector<float> sucess;
            int imgN = 0;
            for (const auto& entry2 : std::filesystem::recursive_directory_iterator(testPath)) {
                if (!std::filesystem::is_directory(entry2.path())) {
```

```
                    imgN++;
                    i++;
                    std::string imgPath{ entry2.path().u8string() };
                    std::string label, filename;
                    std::regex regex1("\\\\");
                    std::vector<std::string> out1(
                        std::sregex_token_iterator(imgPath.begin(), imgPath.end(), regex1, -1),
                        std::sregex_token_iterator()
                    );
                    filename = out1[out1.size() - 1];
                    std::regex regex2("\\_");
                    std::vector<std::string> out2(
                        std::sregex_token_iterator(filename.begin(), filename.end(), regex2, -
1),
                        std::sregex_token_iterator()
                    );
                    label = out2[1];
                    data.push_back(FileManager::getInput(imgPath));
                    labels.push_back(label);
                    if (i == sequenceSize) {
                        network->setInput(&data);
                        std::vector<std::vector<float>> results = network->predictVal();
                        for (unsigned int j = 0; j < results.size(); j++) {
                            std::string outIx = std::to_string(std::max_element(results[j].begin
(), results[j].end()) - results[j].begin());
                            if (network->synsetID[stoi(outIx)][0] == labels[j]) {
                                float out = *std::max_element(results[j].begin(), results[j].end
());

                                sucess.push_back(100 * out);
                            }
                        }
                        i = 0;
                        data.clear();
                        labels.clear();
                        data.shrink_to_fit();
                        labels.shrink_to_fit();
                        if (imgN >= maximumImgs) break;
                    }
                }
            }
            std::string filename;
            std::regex regex1("\\\\");
            std::vector<std::string> out1(
                std::sregex_token_iterator(netPath.begin(), netPath.end(), regex1, -1),
                std::sregex_token_iterator()
            );
            filename = out1[out1.size() - 1];
            std::regex_iterator<std::string::iterator>::regex_type reg("[0-9]+");
```

```
            std::regex_iterator<std::string::iterator> next(filename.begin(), filename.end(), re
g), end;
            std::vector<int> res;
            for (; next != end; next++) {
                res.push_back(std::stoi(next->str()));
            }
            int epoch = res[0],
                iteration = res[1];
            float accuracy = 100 * ((float)sucess.size()) / ((float)imgN);
            float confidence = 0;
            for (unsigned int j = 0; j < sucess.size(); j++) {
                confidence += sucess[j];
            }
            if (sucess.size() > 0) confidence /= sucess.size();
            std::cout << "ep: " << epoch << ", it: " << iteration << ", acc: " << accuracy << ",
 conf: " << confidence << std::endl;
            testRes[iteration][epoch-1] = { accuracy, confidence };
        }
    }
    std::string acc = "",
                conf = "";
    for (unsigned int i = 0; i < testRes.size(); i++) {
        for (unsigned int j = 0; j < testRes[i].size(); j++) {
            acc += std::to_string(testRes[i][j][0]) + ";";
            conf += std::to_string(testRes[i][j][1]) + ";";
        }
    }
    std::ofstream out;
    out.open("testResults.txt");
    out << acc + "\n" + conf;
    out.close();
};
void Trainer::finish() {
    network->finishTraining();
};
```

Trainer.h

```
#ifndef _TrainerH_
#define _TrainerH_

#include "Network.h"

class Trainer {
private:
    int batchSize = 64;
    Network* network;
public:
    Trainer(Network *network);
    void setBatchSize(int size);
```

```
    void train(std::string dataFolder, int iteration);
    void test(std::string dataFolder, std::string networkFolder);
    void finish();
};

#endif
```

CNN.cpp

```
#include <iostream>
#include <limits>

#include <stxxl/vector>
#include <stxxl/random>
#include <stxxl/sort>
#include <filesystem>
#include "CNN/deepNet.h"

int main() {
    //CNN
    int inSize[3] = { 196, 196, 3 };
    Network network;
    int levelIx = network.addLevel();
    int layerIx = network.addLayer("Input", inSize);

    levelIx = network.addLevel();
    layerIx = network.addLayer("Conv", 64, levelIx - 1, layerIx, 3, 1, 0); //194x194 //64
    levelIx = network.addLevel();
    layerIx = network.addLayer("Conv", 64, levelIx - 1, layerIx, 3, 1, 0); //192x192 //64
    levelIx = network.addLevel();
    layerIx = network.addLayer(2, "Pooling", levelIx - 1, layerIx); //97x97
    levelIx = network.addLevel();
    layerIx = network.addLayer("Conv", 128, levelIx - 1, layerIx, 3, 1, 0); //95x95 //128
    levelIx = network.addLevel();
    layerIx = network.addLayer("Conv", 128, levelIx - 1, layerIx, 3, 1, 0); //93x93 //128
    levelIx = network.addLevel();
    layerIx = network.addLayer(3, "Pooling", levelIx - 1, layerIx); //31x31
    levelIx = network.addLevel();
    layerIx = network.addLayer("Conv", 256, levelIx - 1, layerIx, 3, 1, 0); //29x29 //256
    levelIx = network.addLevel();
    layerIx = network.addLayer("Conv", 256, levelIx - 1, layerIx, 3, 1, 0); //27x27 //256
    levelIx = network.addLevel();
    layerIx = network.addLayer("Conv", 128, levelIx - 1, layerIx, 1, 1, 0); //27x27 //64 (Bottle
neck)
    levelIx = network.addLevel();
    layerIx = network.addLayer(3, "Pooling", levelIx - 1, layerIx); //9x9
    levelIx = network.addLevel();
    layerIx = network.addLayer("Flatten", levelIx - 1, layerIx); //1x5184
    levelIx = network.addLevel();
    layerIx = network.addLayer("Dense", 2048, levelIx - 1, layerIx);
```

```
levelIx = network.addLevel();
layerIx = network.addLayer("Dense", 2048, levelIx - 1, layerIx);
levelIx = network.addLevel();
layerIx = network.addLayer("Dense", 2048, levelIx - 1, layerIx);
levelIx = network.addLevel();
layerIx = network.addLayer("Dense", 4, levelIx - 1, layerIx); //Output: 4 Classes
levelIx = network.addLevel();
layerIx = network.addLayer("Softmax", levelIx - 1, layerIx);

network.setActivationFunction("ReLU");
network.setCostFunction("cross-entropy");
network.intialiseParameters();

Trainer trainer(&network);
trainer.setBatchSize(32);
for (unsigned int i = 0; i < 10; i++) {
    trainer.train("structured", i);
}
trainer.finish();
network.exportNet("converged.txt");
trainer.test("results", "testData");

//Neural Network
int inSize[3] = { 28, 28, 1 };
Network network;
int levelIx = network.addLevel();
int layerIx = network.addLayer("Input", inSize);
levelIx = network.addLevel();
layerIx = network.addLayer("Flatten", levelIx - 1, layerIx);
levelIx = network.addLevel();
layerIx = network.addLayer("MultiDenseInput", levelIx - 1, layerIx);
for (unsigned int i = 0; i < 2; i++) {
    levelIx = network.addLevel();
    layerIx = network.addLayer("MultiDense", 128, levelIx - 1, layerIx);
}
levelIx = network.addLevel();
layerIx = network.addLayer("MultiDense", 10, levelIx - 1, layerIx); //Output: 10 Classes
levelIx = network.addLevel();
layerIx = network.addLayer("Softmax", levelIx - 1, layerIx);

network.setActivationFunction("ReLU");
network.setCostFunction("cross-entropy");
network.intialiseParameters();

Trainer trainer(&network);
trainer.setBatchSize(64);
for (unsigned int i = 0; i < 1; i++) {
    trainer.train("MNIST\\structured", i);
```

```
    }
    trainer.finish();
    network.exportNet("converged.txt");
    trainer.test("results", "MNIST\\testing");

    system("pause");
    return 0;
}
```