Gottfried Wilhelm Leibniz Universität Hannover
Institut für Informationsverarbeitung

Bachelorarbeit

Ansätze für die bitrateneffiziente Übertragung von Gewichtsupdates einer DNN-basierten Bildcodierung

Florian Fingscheidt

Matrikelnr. 10037322

Betreuer:	Martin Benjak, M. Sc.
Erstprüfer:	Prof. Dr.-Ing. J. Ostermann
Zweitprüfer:	Prof. Dr.-Ing. B. Rosenhahn

Hannover, Oktober 2024

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Ich stimme der Verwertung meiner Arbeit durch das Institut in den folgenden Punkten zu:

- Aufnahme der gedruckten und der elektronischen Fassung der Arbeit in die Institutsbibliothek,
- Vervielfältigung der gesamten Arbeit oder von Auszügen für Lehrzwecke und
- Wiedergabe der Arbeit durch Bild- und Tonträger

Hannover, den 01.10.2024

Florian Fürgscheidt

Institut für Informationsverarbeitung
Prof. Dr.-Ing. J. Ostermann

Leibniz Universität Hannover

Institut für Informationsverarbeitung
Appelstr. 9A - 30167 Hannover
Sekretariat: +49 511 762 - 5316
Fax: +49 511 762 - 5333
E-Mail: office@tnt.uni-hannover.de
URL: http://www.tnt.uni-hannover.de

31. Mai 2024

Bachelorarbeit

für

Florian Fingscheidt

10037322

Ansätze für die bitrateneffiziente Übertragung von Gewichtsupdates einer DNN-basierten Bildcodierung

Neuronale Netze werden heute in vielen Produkten des täglichen Lebens eingesetzt. Im Jahr 2022 wurde der MPEG-Standard für Neural Network Coding (NNC) als ISO/IEC 15938-17:2022 finalisiert, der eine effiziente Kompression der Gewichte neuronaler Netze ermöglicht. Bei einem inkrementellen Update, z.B. um weitere Features hinzuzufügen, ändert sich ein neuronales Netz in der Regel nur geringfügig. In diesem Fall kann eine effizientere Codierung erreichen werden, indem nur die Differenz zwischen dem neuen und dem alten, auf Empfängerseite bereits bekannten, Netz codiert wird.

Die Aufgabe von Herrn Fingscheidt ist es, ein Verfahren zum effizienten differentiellen Update von neuronalen Netzen zu entwickeln. Dabei sollen Techniken untersucht werden, die bereits beim Transferlernenen einer neuen Netzversion aus der alten sicherstellen, dass nur wenige wichtige Parameter upgedatet werden. Ein einfaches Beispiel für eine solche Technik ist das Einfrieren einiger Netzlagen während des Trainings. Der Fokus soll dabei insbesondere auf sogenannte Adaptoren gelegt werden, die in ein ansonsten statisches Netz eingefügt werden, um eine nachträgliche Adaption zu ermöglichen. Die Differenz des neu trainierten Netzes zum ursprünglichen Netz wird anschließend gemäß des NNC Standards komprimiert.

Das entwickelte Verfahren soll am Beispiel von lernbasierten Bildcodecs evaluiert werden. Dabei ist der Einfluss der Quantisierung des NNC und der Art und Position der Adaptoren auf die Leistungsfähigkeit des Bildcodecs getrennt zu evaluieren.

Die eingereichte Arbeit und die Ergebnisse bleiben Eigentum des Instituts.

Prof. Dr.-Ing. J. Ostermann

Danksagung

Mein Dank geht an Martin Benjak, der für diese Bachelorarbeit mein Betreuer war und mir dieses spannende Thema zugetraut hat. Er hat mich über die gesamte Zeit gut betreut und mich immer unterstützt. Dabei hatte er stets ein offenes Ohr, sowohl für Ergebnisse meiner Experimente, welche ich besprechen wollte, als auch für Probleme, die im Laufe der Zeit auftauchten. Ich danke auch Professor Ostermann, in dessen Institut ich dieses aktuelle Thema bearbeiten durfte. Ein besonderes Dank geht an meinen Vater, der mich bereits in jungen Jahren für die Elektrotechnik und Informatik begeistert hat und der weite Teile meiner Arbeit probegelesen hat. Gespräche am Esstisch mit ihm haben mein Interesse für künstliche Intelligenz und besonders neuronale Netze geweckt. Ein Paper, von dem er mir erzählt hat [1], hat mich zu dem Gedanken bewegt, Adaptoren in dieser Arbeit zu benutzen, um Updates an neuronalen Netzen bitrateneffizient zu gestalten. Ich danke auch meiner Mutter, die mich immer unterstützt hat und meiner Freundin Annika, die mich meist erst Abends nach dem Arbeiten an dieser Bachelorarbeit gesehen hat und mir einen gesunden Ausgleich zu dieser Arbeit geboten hat.

Kurzfassung

Die Einsatzgebiete neuronaler Netze sind vielfältig. In dieser Arbeit wird am Beispiel gelernter Bildcodecs mit neuronalen Faltungsschichten untersucht, wie neuronale Netze ein Gewichtsupdate erfahren können, häufig auch Over-the-air-Update genannt. Dabei werden sowohl Ansätze ohne Modifikation des Netzes, als auch zusätzliche Adaptoren an verschiedenen Stellen im Netz nach einem Fine-Tuning untersucht, und bezüglich des PSNR-Gewinns und der zu übertragenen Gewichteanzahl evaluiert. Schließlich wird auch die Bitrateneffizienz bei der Übertragung der geupdateten Gewichte mitbetrachtet. Es zeigt sich, dass die Verwendung von Adaptoren im Fine-Tuning und deren anschließende Benutzung als Update ein besonders bitrateneffizienter Ansatz ist. Ohne Verwendung von Adaptoren würde man bei Komplettübertragung des Bildcodec-Netzes eine Nachrichtengröße von 50.610 KB und eine komprimierte Nachrichtengröße von 15.609 KB benötigen und in der gegebenen Aufgabe (engl. Task) eine BD-Rate von -50,21% erzielen. Wir zeigen, wie ein Bildcodec mit knapp 12 Mio. Parametern durch die Verwendung von Adaptoren mit einer Update-Nachrichtengröße von 1.498 KB bzw. einer komprimierten Update-Nachrichtengröße von nur 568 KB zu einer Performanzsteigerung mit einer BD-Rate von immerhin ca. -17,16% führen kann. Dies ist zwar nur 1/3 des BD-Ratengewinns im Vergleich zum komplett übertragenen Bildcodec-Netz, jedoch bei lediglich 3,6% (!) der erforderlichen Update-Nachrichtengröße in komprimierter Form.

Inhaltsverzeichnis

1. Einleitung

Heutzutage sind bildverarbeitende neuronale Netze kaum noch wegzudenken, sie werden in immer mehr Bereichen wie zum Beispiel in Smartphones und in der Automobilbranche eingesetzt. Solche Netze können mehrere Millionen oder sogar Milliarden Parameter besitzen. Es kommt vor, dass an neuronalen Netzen Updates durchgeführt werden um die Performanz des Netzes zu steigern. Klassischerweise wird für solche Updates das gesamte besserperformante Netz versandt und das schwächerperformante Netz am Empfangsort ausgetauscht. Dabei werden alle Gewichte des besserperformanten Netzes übertragen, was bei großen Netzen eine große Nachrichtenmenge bedeutet. Die große Nachrichtenmenge bei solchen Netz-Updates trägt beim Versenden zu einer hohen Netzwerkauslastung bei.

Um die Netzwerkauslastung durch ein Netz-Update gering zu halten, ist es nötig, die Nachrichtengröße des Updates zu verringern. Dies bedeutet, dass ein Update *bitrateneffizient* sein sollte.

Am Beispiel eines gelernten Bildcodecs zielt die vorliegende Arbeit darauf, zu untersuchen, wie mit Hilfe von Adaptoren Netz-Updates an neuronalen Netzen bitrateneffizient gestaltet werden können und ob sich dieser Ansatz im Vergleich zu anderen Methoden lohnt.

Naheliegend wäre als Erstes, das komplette neuronale Bildcodec-Netz zu übertragen. Diese Methode (unsere Baseline) ist jedoch nicht bitrateneffizient. In Experimenten werden wir untersuchen, wie wir die Anzahl an versendeten Parametern verringern können. Dies wollen wir dadurch erreichen, dass wir das schwächerperformante Netz, welches nur mit einem kleinen Datensatz trainiert wurde, bis auf wenige ausgewählte Schichten einfrieren und in einem Fine-Tuning mit einem größeren Datensatz derselben Domäne nur diese ausgewählten Schichten lernen lassen. Somit müssten bei einem Update des Ausgangsnetzes nur die im Fine-Tuning nachtrainierten Schichten übertragen werden, und nicht mehr das gesamte Netz. Dabei werden wir bestimmte Blöcke im Encoder und im Decoder, sowie spezifische Blöcke im sogenannten Hyperprior-Teil des gelernten Bildcodecs auf ihre Brauchbarkeit für Netzupdates überprüfen.

Andere Autoren passen ihre Netze an geänderte Aufgaben (engl. Tasks) an, indem sie sogenannte Adaptoren - also kleine zusätzliche Netzstrukturen - an verschiedenen Stellen in ihr Netz anhängen. Diese bestehen in der Regel aus vollverbundenen Schichten [2, 3] und werden in einem Fine-Tuning, nach ihrem Anhängen ans Netz, trainiert, wobei der Rest des Netzes eingefroren wird. Dies bedeutet, dass nur diese Adaptoren im Fine-

Tuning lernen.

Gelernte Bild-Codecs, wie wir sie in dieser Arbeit untersuchen wollen, sind jedoch typischerweise Faltungsnetze (engl. Convolutional Neural Networks, CNNs), um unterschiedliche Bildgrößen flexibel verarbeiten zu können. Für CNNs gibt es nur wenige Publikationen zu Adaptoren. Chen et al. [4] z.B. verwenden einen Adaptor, welcher auf Faltungsschichten bassiert und testen ihn für eine Domain-Shift Task (Änderung der Daten-Domäne) in mehreren unterschiedlichen Netzen. Shen et al. [5] verwenden diesen Adaptor ebenfalls, um den Decoder eines Bild-Codecs einen Domain-Shift lernen zu lassen. Der Vorteil ist dabei, dass nur wenige Gewichte im Fine-Tuning mit dem neuen Datensatz lernen müssen, und somit das Lernen schneller abläuft. Für uns ist dabei jedoch wichtiger, dass es nur wenige Gewichte sind, die lernen. Somit müssten bei einem Update, welches Adaptoren verwendet, nur deren, verglichen zum gesamten Netz, wenige Gewichte übertragen werden.

Wir werden einen neuen Adaptor für neuronale Faltungsnetze entwickeln. Unser Adaptor hebt sich in folgender Weise vom Stand der Technik ab: Wir verwenden unseren Adaptor zum einen *parallel* zu bereits vorhandenen Blöcken und *sowohl im Bild-Encoder als auch im Bild-Decoder*. Wir haben einen *neuen Aufbau* mit einer Faltungsschicht und einstellbarem Kernel im Bottleneck, was wir bislang bei CNN-Adaptoren noch nicht gesehen haben.

Mit diesem neuen Adaptor wollen wir ein möglichst leistungssteigerndes und gleichzeitig bitrateneffizientes Update in *gleichbleibender Bild-Domäne* realisieren. Dies wollen wir dadurch erreichen, dass wir dem schwächerperformanten Netz, welches nur mit einem kleinen Datensatz trainiert wurde, Adaptoren anhängen und diese sendeseitig in einem Fine-Tuning mit einem größeren Datensatz derselben Domäne lernen lassen. Da sich an diesem Netz nur die Adaptoren im Fine-Tuning geändert haben, müssen auch nur diese für ein Netz-Update anderer Netze zur Empfangsseite übermittelt werden. In einem zweiten Schritt werden wir auch noch die übertragenen Adaptoren mit Hilfe eines sogenannten Neural Network Coding [6] komprimieren, so dass wir nur noch komprimierte Adaptoren versenden müssen, was unsere Methode nochmals wesentlich bitrateneffizienter machen wird. Diese sollen auf der Empfangsseite wieder dekomprimiert und an das dortige Netz angehängt werden.

Die Arbeit gliedert sich in fünf weitere Kapitel. In Kapitel 2 werden wichtige Grundlagen, die dem Verständnis unserer Experimente und deren Auswertung dienen, vorgestellt. Dazu gehören Grundlagen der Quellencodierung, welche für unseren verwendeten Bildcodec wichtig sind, aber auch Grundlagen zu neuronalen Faltungsnetzen, Architekturen von Bildcodecs, die Funktionsweise des Neural Network Codings und die in dieser Arbeit für die Evaluation verwendeten Metriken. Im Anschluss wird in Kapitel 3 der Stand der Technik vorgestellt und kurz erklärt, in welchen Punkten sich unser Ansatz von den bereits erforschten Ansätzen unterscheidet. In Kapitel 4 stellen wir unsere Methode vor. Dabei werden die "Storyline" und der von uns verwendete Bildcodec erklärt. Auch die von uns entwickelten Adaptoren werden hier vorgstellt, ihre Einsatzorte im

Bildcodec besprochen und verdeutlicht, wo wir uns vom Stand der Technik abheben. Anschließend werden in Kapitel 5 die Ergebnisse der Experimente zu unserem Ansatz der Netz-Updates mit Adaptoren besprochen und mit anderen Methoden zur bitrateneffizienten Übertragung von Netz-Updates verglichen. Zum Schluss fassen wir in Kapitel 6 unsere Ergebnisse noch einmal zusammen und geben einen Ausblick, welche Untersuchungen in Zukunft noch von Interesse sein werden.

2. Grundlagen

Im Folgenden werden Konzepte und Grundbausteine erklärt, die für den Verlauf der Arbeit von Bedeutung sind. Sie bilden das Fundament, auf dem wir uns in dieser Arbeit bewegen. Zunächst stellen wir in Abschnitt 2.1 ein wichtiges Konzept der Informationstheorie vor. Darauf aufbauend widmen wir uns in Abschnitt 2.2 einem Verfahren zur verlustlosen Kompression. Anschließend behandeln wir in Abschnitt 2.3 Konzepte der verlustbehafteten Kompression. In Abschnitt 2.4 werden verschiedene Schichten neuronaler Netze und ihre mathematische Notation als Funktion vorgestellt. Abschnitt 2.5 stellt den Grundaufbau von Codecs vor. Die Netzstruktur zweier wichtiger Bild-Codec-Ansätze ist in Abschnitt 2.6 beschrieben. Anschließend werden in Abschnitt 2.7 Aufbau und Funktion des Neural Network Coding eingeführt. Abschnitt 2.8 behandelt Adaptoren für vollverbundene Schichten, eine wichtige Grundlage für die eigene vorgeschlagene Methode. Zum Schluss werden in Abschnitt 2.9 wichtige Metriken für diese Arbeit erklärt.

2.1. Informationsgehalt und Entropie

Für die Kompression, die in dieser Arbeit eine zentrale Rolle einnimmt, ist ein informationstheoretisches Konzept sehr wichtig, das sich Entropie nennt [7]. Die Entropie stellt den durchschnittlichen Informationsgehalt eines Zeichens dar. Wichtig dafür ist unter anderem die Auftrittswahrscheinlichkeit eines Zeichens $P(\hat{x}) \in [0, 1]$ für das Zeichen $\hat{x} \in \hat{\mathcal{X}}$ wobei $\hat{\mathcal{X}} = \{\hat{X}_1, \hat{X}_2, ...\hat{X}_k\}$ ist und k die Anzahl der unterschiedlichen Zeichen im Alphabet $\hat{\mathcal{X}}$ ist[1].

Für die Entropie ist der Informationsgehalt

$$I(\hat{x}) = \log_2 \left(\frac{1}{P(\hat{x})} \right) \tag{2.1}$$

des Zeichens $\hat{x}$ besonders wichtig [7]. Dieser nimmt zu, je niedriger die Auftrittswahrscheinlichkeit $P(\hat{x})$ ist. Der durchschnittliche Informationsgehalt pro Zeichen, die soge-

[1] Aus Konsistenzgründen mit anderen Abschnitten dieser Arbeit, z.B. Abschnitt 2.3, verwenden wir für das diskrete Symbol $\hat{x}$ das Dach.

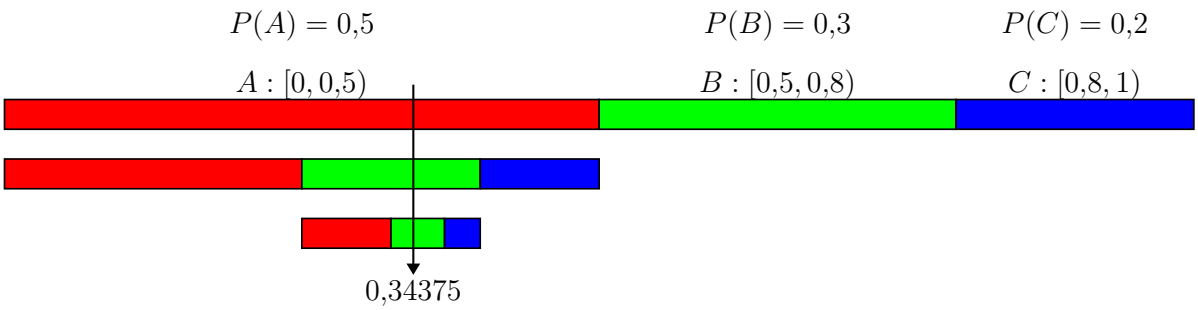


Abbildung 2.1.: Codierung des Wortes ABB mit $P(A) = 0{,}5$, $P(B) = 0{,}3$ und $P(C) = 0{,}2$ mit Hilfe eines arithmetischen Encoders.

nannte Entropie ergibt sich also aus folgendem mathematischem Zusammenhang [7]:

$$\begin{aligned} H &= \sum_{\hat{x}\in\hat{\mathcal{X}}} P(\hat{x}) I(\hat{x}) \\ &= \sum_{\hat{x}\in\hat{\mathcal{X}}} P(\hat{x}) \log_2\left(\frac{1}{P(\hat{x})}\right) \\ &= \sum_{\hat{x}\in\hat{\mathcal{X}}} P(\hat{x}) \big(\log_2(1) - \log_2(P(\hat{x}))\big) \\ &= \sum_{\hat{x}\in\hat{\mathcal{X}}} P(\hat{x}) \big(0 - \log_2(P(\hat{x}))\big) \\ &= -\sum_{\hat{x}\in\hat{\mathcal{X}}} P(\hat{x}) \log_2(P(\hat{x})). \end{aligned} \tag{2.2}$$

Die Entropie beschreibt dabei auch die minimale Anzahl an Bits pro Zeichen, die im Durchschnitt erforderlich ist, um eine Nachricht verlustlos zu codieren. *Das bedeutet, dass die Entropie eine untere Schranke für die Bitrate aller verlustlosen Codierverfahren bildet.* Daraus resultiert auch die Bedeutsamkeit der Entropie für verlustlose Kompressionsverfahren, denn je näher ein Codierer der Entropie kommt, desto besser ist er. Im Folgenden wird eine solche verlustlose Kompressionsmethode vorgestellt.

2.2. Arithmetische Codierung

Im Folgenden schauen wir uns Beispiele an, wobei die drei Symbole $\hat{x}_1, \hat{x}_2$ und $\hat{x}_3$ mit den Buchstaben A, B und C dargestellt werden. In Abbildung 2.1 ist die Codierung des Wortes ABB durch einen arithmetischen Coder [8] gezeigt. Wir nehmen an, dass die drei Symbole A, B, C jeweils mit den Wahrscheinlichkeiten $P(A) = 0{,}5, P(B) = 0{,}3$ und $P(C) = 0{,}2$ auftreten. Dabei wird ein Intervall $[0, 1)$ ensprechend der Auftrittswahrscheinlichkeiten der Symbole auf diese aufgeteilt, so dass A das Intervall $[0, 0{,}5)$, B das Intervall $[0{,}5, 0{,}8)$ und C das Intervall $[0{,}8, 1)$ zugewiesen bekommt. Würde man es dabei

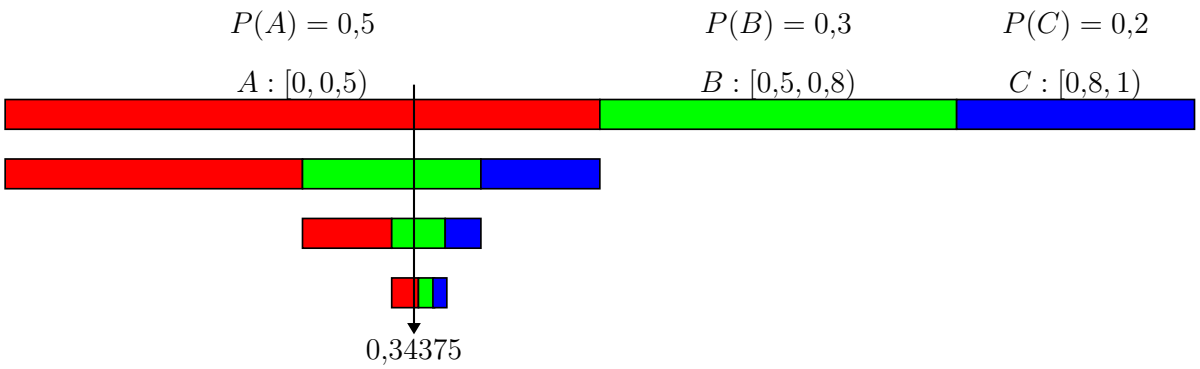


Abbildung 2.2.: Codierung des Wortes $ABBA$ mit $P(A) = 0{,}5$, $P(B) = 0{,}3$ und $P(C) = 0{,}2$ mit Hilfe eines arithmetischen Encoders.

nun belassen, würde man wie in anderen Codierungsstandards (z.B. Huffman-Code [9]) jedes Symbol einzeln codieren. Bei der arithmetischen Codierung werden jedoch mehrere Symbole zusammen codiert, das bedeutet, dass anstelle von A gleich das ganze Wort ABB als *ein* Wert codiert wird. Dabei werden die einzelnen Intervalle der Symbole weiter aufgeteilt, so dass das gesamte Intervall von unserem ersten Symbol A nochmals entsprechend der Auftrittswahrscheinlichkeit der Symbole in A, B und C aufgeteilt wird. Dadurch entstehen dann die Kombinationen AA, AB und AC. Das Verfahren des Unterteilens wird für jedes weitere Symbol im Wort wiederholt und das Intervall des aktuellen Symbols wird dabei als Ausgangspunkt für das nächste Symbol gewählt. Das ganze wird so lange wiederholt, bis wir am Ende des Wortes angekommen sind. Unser Wort in Abbildung 2.1 sei AAB, und ist nun durch Auswahl eines einzelnen Intervalls, in diesem Fall des untersten grünen Intervalls ausdrückbar. Für jeden Wert, für den gilt, dass er im letzendlichen Intervall von ABB liegt, gilt, das dieser eine Codierung eines Wortes ist, welches mit ABB anfängt. *Für die letztendliche Codierung wird nun derjenige Wert des Intervalls gewählt, welcher in binärer Darstellung die wenigsten Zeichen benötigt.* Dabei gilt, dass wenn das Intervall, in welches unser Wort fällt, mindestens eine Breite von $\frac{1}{2^b}$ mit $b \in \mathbb{N}$ besitzt, dass dieses in höchstens b Nachkommastellen als binäre Darstellung codiert werden kann. In unserem Beispiel von ABB bietet sich der Wert 0,34375 gut an, da dieser in Binärdarstellung die wenigsten Nachkommastellen von allen Werten im Intervall ABB benötigt. Die Binärdarstellung des Wertes ergibt sich mit $0 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} + 1 \cdot 2^{-5} = 2^{-2} + 2^{-4} + 2^{-5} = 0{,}34375$ zu 0,01011. Da es keinen Wert ≥ 1 in dem anfangs beschriebenen Intervall $[0, 1)$ gibt, können die Vorkommastellen nach der Codierung ignoriert werden. Somit haben wir unser Wort ABB in die fünf Bits 01011 codiert.

Es gibt jedoch eine Schwierigkeit, da wie in Abbildung 2.2 gezeigt, der Wert 0,34375 auch $ABBA$ codieren kann und viele weitere Wörter auch. Dadurch ist es notwendig, entweder dem codierten Wort anzuhängen, wie lang das Ursprungswort ist, oder es wird ein weiteres Symbol ins Alphabet eingefügt, welches das Ende des Wortes markieren soll.

Auf Empfängerseite kann dieses Vorgehen leicht rückgängig gemacht werden. Dazu wird

wieder ein Intervall $[0, 1)$, wie in der Codierung, eingeteilt. Danach wird geschaut, in welchem Sub-Intervall der empfangene Wert liegt, dementsprechend ist das Symbol des Sub-Intervalls das erste Symbol des Wortes. Dieses Sub-Intervall wird dann wieder unterteilt und das Verfahren wiederholt sich, bis man entweder auf das Symbol, welches das Ende eines Wortes markieren soll, stößt, oder die Länge des Wortes beim Decodieren erreicht wurde.

Um nun errechnen zu können, wie viele Bits b man für eine "typische" n Zeichen lange Nachricht benötigt, um sie mit dem Verfahren der arithmetischen Codierung codieren zu können, stellen wir eine Formel auf. Bei einer "typischen" Nachricht handelt es sich um eine Nachricht, für die gilt, dass sie $n \cdot P(\hat{x}_k)$ mal das Zeichen $\hat{x}_k$ enthält für alle $\hat{x}_k$ im Alphabet $\hat{\mathcal{X}}$. Also gilt in einem Beispiel mit $n = 100$, dass $n \cdot P(A) = 100 \cdot 0{,}5 = 50$ mal das A enthalten wäre, 30 mal das B und 20 mal das C. Der mathematische Zusammenhang, um nun zu berechnen, wie viele Bits zur Codierung einer n Symbole langen Nachricht notwendig sind, lautet in unserem Beispiel:

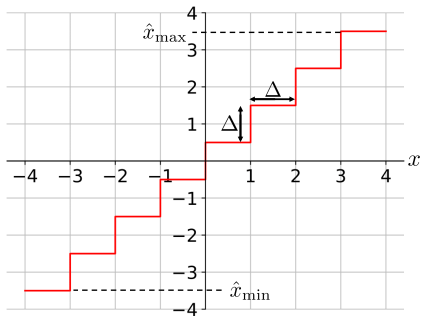
$$\begin{aligned}
\frac{1}{2^b} &\leq P(A)^{n \cdot P(A)} \cdot P(B)^{n \cdot P(B)} \cdot P(C)^{n \cdot P(C)} \\
&= \left(P(A)^{P(A)} \cdot P(B)^{P(B)} \cdot P(C)^{P(C)}\right)^n \\
\Rightarrow -b &\leq n \cdot (P(A) \log_2(P(A)) + P(B) \log_2(P(B)) + P(C) \log_2(P(C))) \\
\Rightarrow b &\geq n \cdot H
\end{aligned} \tag{2.3}$$

mit der Entropie H (2.2). Der unten stehende Zusammenhang ist dabei allgemeingültig und beschränkt sich nicht nur auf unser Beispiel. Das Verfahren der arithmetischen Codierung kommt, je größer die Nachricht ist, mit seiner durchschnittlichen Anzahl an Bits zur Codierung eines Symbols immer näher an die Entropie heran.

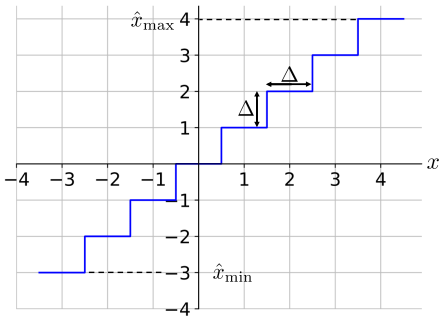
2.3. Verlustbehaftete Kompression / Quantisierung

Zur verlustbehafteten Kompression werden Quantisierer eingesetzt. Dabei wird ein kontinuierlicher Zahlenbereich oder ein umfangreicher diskreter Zahlenbereich auf einen kleineren diskreten abgebildet. Die Abbildung auf den reduzierten diskreten Zahlenbereich kann unterschiedlich aussehen. Zur Veranschaulichung werden Quantisierungskennlinien genutzt. In dieser Arbeit wird dabei die x-Achse die Eingabe in den Quantisierer bezeichnen und $\hat{x}$ die quantisierte Ausgabe des Quantisierers. Wichtig für die Quantisierung ist die Anzahl der Werte im diskreten Zahlenbereich, auf den die Quantisierung abbildet, sie wird $k \in \mathbb{N}$ gennant. Es gilt, $k = 2^b$, wobei $b \in \mathbb{N}$ die Anzahl der Bits ist, mit der jeder Wert beschrieben werden soll. Ebenfalls wichtig ist für die Quantisierung die Quantisierungsschrittweite $\Delta \in \mathbb{R}^+$, die beschreibt, wie breit der Bereich ist, innerhalb dessen alle Eingabewerte auf denselben Ausgabewert abgebildet werden.

Die simpelste Art von Quantisierern stellt den sogenannten gleichförmigen Quantisierer (uniform quantizer) dar, der häufig bei skalarem Input eingesetzt wird. Man unterschei-



(a) Kennlinie eines *Midrise*-Quantisierers mit $b = 3$ bit.



(b) Kennlinie eines *Midtread*-Quantisierers mit $b = 3$ bit.

Abbildung 2.3.: Kennlinien zweier gleichförmiger Quantisierer mit 8 Stufen ($b = 3$). Die Eingabe ist x und der quantisierte Wert/die Ausgabe ist $\hat{x}$.

det bei einem gleichförmigen Quantisierer den *Midrise*-Quantisierer von dem *Midtread*-Quantisierer [10]. Bei beiden gilt für die Quantisierungsschrittweite

$$\Delta = \frac{\hat{x}_{\max} - \hat{x}_{\min}}{k - 1}, \tag{2.4}$$

wobei $\hat{x}_{\max}$ und $\hat{x}_{\min}$ den jeweils maximalen und minimalen Rekonstruktionswert darstellt. Die Quantisierungskennlinie eines beispielhaften *Midrise*-Quantisierers ist in Abbildung 2.3a zu sehen. Die Quantisierungsschrittweite Δ findet man hierbei in der Breite der Stufen, da jeder Wert x, der auf dieselbe Stufe fällt, auf denselben Wert $\hat{x}$ abgebildet wird. Die Schrittweite Δ entspricht aber auch der Höhe jeder Stufe. Besonders ist hierbei, dass der Wert $x = 0$ nur mit einem Fehler $e = \hat{x} - x = \pm\frac{\Delta}{2}$ dargestellt werden kann. Der mathematische Zusammenhang ist wie folgt:

$$\hat{x} = \operatorname{sign}(x) \cdot \Delta \left(\left\lfloor \frac{|x|}{\Delta} \right\rfloor + \frac{1}{2} \right). \tag{2.5}$$

Dabei gilt:

$$\operatorname{sign}(x) = \begin{cases} -1, & x < 0 \\ +1, & x \geq 0 \end{cases}$$

Die Quantisierungskennlinie eines *Midtread*-Quantisierers ist in Abbildung 2.3b zu sehen. Die Quantisierungsschrittweite Δ entspricht auch hier der Breite und der Höhe einer Stufe. Die Besonderheit des *Midtread*-Quantisierers ist, dass er anders als der *Midrise*-Quantisierer die Null ($x = 0$) perfekt abbildet ($\hat{x} = 0$). *Midrise*- und *Midtread*-Quantisierer sind sich sehr ähnlich, der *Midtread*-Quantierer ist dabei ein *Midrise*-Quantisierer, bei dem die Kennlinie um $\Delta/2$ in positive x-Richtung und in positive $\hat{x}$-Richtung verschoben ist. Der mathematische Zusammenhang des *Midtread*-Quanisierers

ist folgender:

$$\begin{aligned}\hat{x} &= \text{sign}(x) \cdot \Delta \left(\left\lfloor \frac{|x + \frac{\Delta}{2}|}{\Delta} + \frac{1}{2} \right\rfloor \right) + \frac{\Delta}{2} \\ &= \text{sign}(x) \cdot \Delta \left\lfloor \frac{|x|}{\Delta} + \frac{1}{2} \right\rfloor \end{aligned} \tag{2.6}$$

Im Weiteren gibt es noch viele andere Quantisiererarten (ungleichförmige Quantisierer [11], Vektorquantisierer [12], etc.), deren Behandlung für diese Arbeit jedoch nicht maßgeblich ist.

2.4. Grundbausteine neuronaler Netze

Neuronale Netze werden immer häufiger verwendet, um Probleme verschiedenster Art zu lösen [13, 14, 15, 16]. Sie gewinnen immer mehr Relevanz und Einfluss in verschiedenen Feldern. Da wir in dieser Arbeit viel mit neuronalen Netzen arbeiten, stellen wir hier einige wichtige neuronale Schichten vor und erklären diese. In Abschnitt 2.4.1 werden die simpelsten Schichten, die vollvernetzten Schichten erklärt. Die für diese Arbeit maßgeblichen sogenannten Faltungsschichten werden in Abschnitt 2.4.2 behandelt. Danach wird in Abschnitt 2.4.3 die sogenannte tiefenweise Faltung behandelt und im Anschluss in Abschnitt 2.4.4 Faltungsschichten mit Stride. In Abschnitt 2.4.5 werden die maskierten Faltungsschichten erklärt. Anschließend werden in Abschnitt 2.4.6 zwei Arten von Pooling-Schichten vorgestellt. In Abschnitt 2.4.7 wird eine Pixel Shuffle-Schicht erläutert. Am Ende wird in Abschnitt 2.4.8 anhand eines Beispielnetzes der Aufbau eines neuronalen Netzes erklärt.

2.4.1. Vollverbundene Schichten

In Abbildung 2.4 ist der Aufbau einer vollverbundenen neuronalen Schicht abgebildet. Die sogenannten Knoten (engl. nodes) werden hier als Kreisflächen dargestellt. Auf der linken Seite sehen wir vier Knoten, diese sind entweder Eingangsknoten oder die Knoten der vorherigen Schicht in unserem Netzwerk. In unserer Abbildung dienen sie als Eingangsknoten für die rechts abgebildete Schicht, welche wir beschreiben. Unsere Schicht besitzt zwei Ausgangs-Knoten, welche auch Neuronen genannt werden. Sie bekommen die Ausgabedaten aller vorherigen Knoten übergeben und wir nennen sie $X_i \in \mathbb{R}$ mit $i \in \mathcal{I}$ und $\mathcal{I} = \{0, ..., M-1\}$ der beispielhaften Dimension $M = 4$. Die Übertragung der Signale ist hier in Form von Verbindungen zwischen diesen Neuronen gekennzeichnet. Die Ausgaben $Y_j \in \mathcal{J}$ mit $j \in \mathcal{J}$ mit $\mathcal{J} = \{0, ..., N-1\}$ der beispielhaften Dimension $N = 2$ hängt von den Neuronen unserer Schicht ab. Wir sehen, dass in der Abbildung die Verbindungen ein sogenanntes Gewicht $w_{i,j} \in \mathbb{R}$ besitzen, mit dem der Ausgabewert

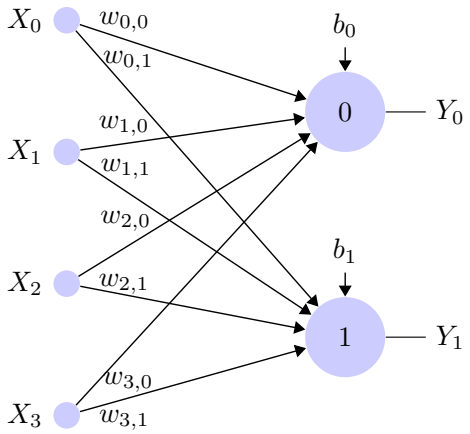


Abbildung 2.4.: Visuelle Repräsentation einer FC(2)-Schicht, welche 4 Eingangssignale bekommt und zwei Ausgangs-Neuronen enthält.

des Neurons auf der linken Seite multipliziert und an das empfangende (rechte) Neuron übertragen wird. Jede dieser Verbindungen besitzt ein Gewicht, diese werden beim Lernen angepasst, um die gewünschte Aufgabe zu erfüllen. Die Gewichte ergeben zusammen eine Gewichtsmatrix $\mathbf{W} = (w_{i,j})$ der Dimension $M \times N$. Jedes Neuron verfügt noch über einen ebenfalls lernbaren Bias $b_j \in \mathbb{R}$, welcher nach der Addition aller gewichteten Eingangssignale in diesem Neuron hinzuaddiert wird. Der mathematische Zusammenhang ist wie folgt:

$$Y_j = \sum_{i \in \mathcal{I}} X_i \cdot w_{i,j} + b_j. \tag{2.7}$$

Vollverbundene (engl. fully connected) Schichten werden im Folgenden mit $FC(N)$ bezeichnet.

2.4.2. Faltungsschicht und Zero Padding

Faltungsschichten werden besonders bei der Verarbeitung von Bildern häufig genutzt, da sie deutlich sparsamer in der Anzahl der Parameter sind, als es vollverbundene Schichten im Aufgabenbereich der Bildverarbeitung sind. Faltungsschichten benötigen nicht pro Pixel für jedes Neuron ein Gewicht, sie lassen stattdessen einen sogenannten Kernel mit wenigen Gewichten über das Bild laufen.

In Abbildung 2.5 ist eine 2-dimensionale Faltung mit einem 2-dimensionalen Kernel der beispielhaften Dimension $K \times K = 3 \times 3$ dargestellt. In grüner Farbe sind die Eingangsdaten $X_{i,j} \in \mathbb{R}$ mit $i, j \in \mathcal{I} = \{0, 1, ..., 9\}$ der beispielhaften Dimension 10×10 gezeigt. Der Kernel mit seinen neun lernbaren Gewichten $w_{k,\ell} \in \mathbb{R}$ mit $k, \ell \in \mathcal{K}$ mit $\mathcal{K} = \{0, 1, 2\}$ hat die Farbe orange. Die Ausgangsdaten $Y_{i,j} \in \mathbb{R}$ mit $i, j \in \{0, 1, ..., 9\}$ der Dimension 10×10 sind in blauer Farbe dargestellt.

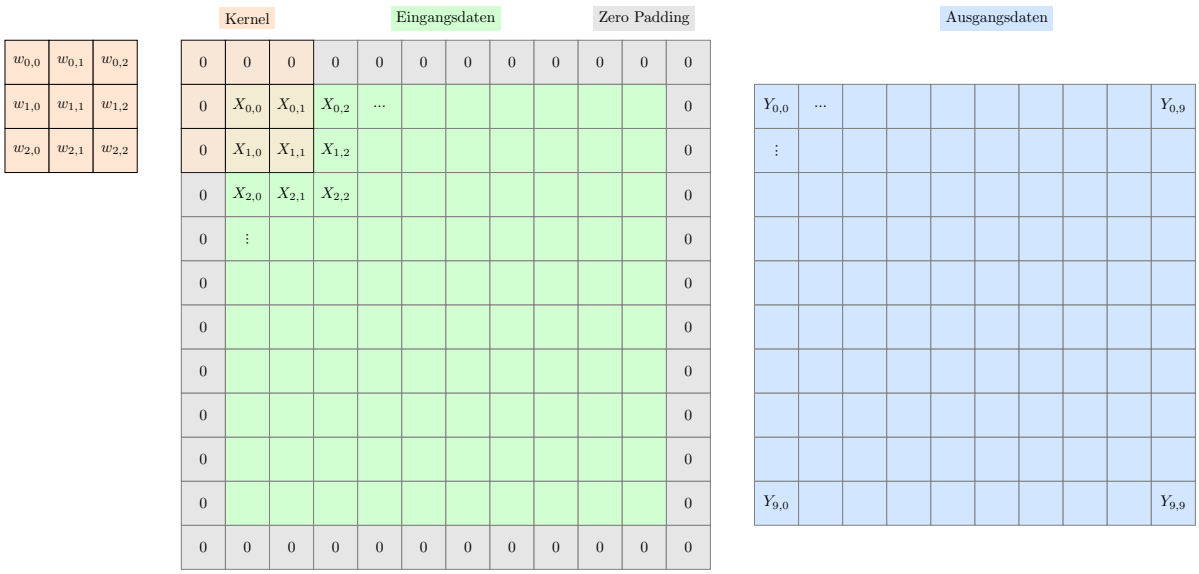


Abbildung 2.5.: Visuelle Repräsentation einer Conv(3×3)-Schicht mit Zero Padding. Der auf den Eingangsdaten abgebildete Kernel liefert den Ausgangswert $Y_{0,0}$.

Die Faltungsoperation in Abbildung 2.5 verwendet ein sogenanntes Zero Padding. Hierbei werden die Eingangsdaten durch eine gewisse Anzahl an Nullwerten umgeben (graue Farbe). Die Anzahl und Positionierung der hinzugefügten Nullpunkte wählt man so, dass die 2-dimensionale Faltung der Eingangsdaten inklusive der Nullwerte Ausgangsdaten erzeugt, die die gleiche Dimension wie die Eingangsdaten haben, in diesem Falle 10×10. Der mathematische Zusammenhang ist wie folgt:

$$Y_{i,j} = \sum_{k \in \mathcal{K}} \sum_{\ell \in \mathcal{K}} X_{i+k-d, j+\ell-d} \cdot w_{k,\ell} + b, \quad i,j \in \mathcal{I}, \quad d = \left\lfloor \frac{K}{2} \right\rfloor \tag{2.8}$$

Hierbei steht $b \in \mathbb{R}$ für den sogenannten Bias, der dem gezeigten orangenen Kernel zugeordnet und ebenfalls lernbar ist. Die Positionierung des Kernels in Abbildung 2.5 auf vier grünen Werten und fünf Nullwerten erzeugt mit (2.8) den ersten Ausgangswert $Y_{0,0}$. Der zweite Ausgangswert $Y_{0,1}$ würde sich durch Rechtsverschiebung des Kernels um eine Position ergeben, und so weiter. Um das Zero Padding einfach zu verstehen, kann man sich vorstellen, dass das blaue Faltungsergebnis an der Mittenposition des Kernels abgelegt wird.

Nun soll noch auf den Fall eingegangen werden, was passiert, wenn man kein Padding einsetzt. Da in der Faltungssumme (2.8) $K \times K$ Produkte zu bestimmen sind, müssen alle Kernelgewichte w auf Eingangsdaten X positioniert werden. Hat man die grau hinterlegten Nullwerte wie in der Abbildung 2.5 nicht, würde als erster Ausgangswert $Y_{1,1}$ bestimmt werden. Insgesamt würden nur Ausgangswerte $Y_{i,j}$ mit $i,j \in \{1,2,...,8\}$ bestimmt werden, was dazu führt, dass ohne Zero Padding die Ausgangsdaten abhängig von der Kernel-Dimension stets eine kleinere Dimension als die Eingangsdaten haben.

Im Folgenden wird eine Faltungsschicht mit Conv$(K \times K, N)$ bezeichnet. Wie schon be-

kannt, steht $K \times K$ für die Dimension des Kernels. Der Wert $N \in \mathbb{N}$ bezeichnet die Anzahl der Kernels in der betreffenden Schicht. Damit wird nicht nur eine, sondern es werden insgesamt N 2-dimensionale Ausgangsstrukturen erzeugt. Eine einzelne 2-dimensionale Ausgangsstruktur nennt man auch Feature Map. Nun kann es sein, dass unsere Faltungsschicht nicht nur eine 2-dimensionale Eingangsdatenstruktur verarbeitet, sondern M Feature Maps am Eingang verarbeiten muss. Dies würde einen 3-dimensionalen Kernel erfordern. Die dritte Kernel-Dimension wird typischerweise nicht in der Conv()-Funktion aufgeführt, weil sie sich zwingend aus der Menge der Feature Maps am Eingang ergibt. Der Kernel hätte also eigentlich die Dimension $K \times K \times M$, die Faltung bleibt jedoch eine 2-dimensionale Faltungsoperation.

2.4.3. Tiefenweise Faltung

Die tiefenweise Faltung (engl. depth-wise convolution) funktioniert ähnlich wie die bereits bekannte Faltung, bei ihr aber gibt es zwei Besonderheiten. Zum einen werden die Eingangs-Feature Maps aufgeteilt in $D \in \mathbb{N}$ Blöcke. Zum anderen erhält jeder Block seine eigene Faltungsschicht. Dadurch haben wir kleinere Kernels, da nicht jeder Kernel jede Eingangs-Feature-Map zu sehen bekommt. Am Ende werden von allen diesen parallel operierenden Faltungsschichten die Ausgaben konkateniert und als Ausgabe-Feature Maps der tiefenweisen Faltungsschicht gesehen.

Im Folgenden wird eine Schicht mit tiefenweiser Faltung als $\mathrm{DWConv}(K \times K, D, N)$ beschrieben. Hierbei steht $K \times K \in \mathbb{N}^2$ für die Größe des Kernels und $N \in \mathbb{N}$ für die Anzahl der Ausgabe-Feature Maps. Die Größe $D \in \mathbb{N}$ bezeichnet die Anzahl der Blöcke.

Die Besonderheit hierbei ist die verringerte Parameteranzahl der tiefenweisen Faltungsschicht. Hierbei ist $M \in \mathbb{N}$ die Anzahl der Eingangs-Feature Maps. Statt der

$$\#Parameter = (K \times K \times M) \cdot N + N$$

Parameter einer klassischen Faltungsschicht benötigen wir für eine tiefenweise Faltung mit D Blöcken, von denen jeder nur $\frac{N}{D}$ Ausgangs-Feature-Maps erzeugt, lediglich

$$\begin{aligned} \#Parameter &= \left[(K \times K \times \frac{M}{D}) \cdot \frac{N}{D} + \frac{N}{D} \right] \cdot D \\ &= (K \times K \times \frac{M}{D}) \cdot N + N. \end{aligned}$$

2.4.4. Faltungsschicht mit Stride

In Abbildung 2.6 ist eine 2-dimensionale Faltung mit einem sogenannten Stride und einem 2-dimensionalen Kernel der beispielhaften Dimension $K \times K = 3 \times 3$ dargestellt. Die Eingangsdaten, der Kernel und das Zero Padding entsprechen denen aus Abschnitt 2.4.2. Was sich hier jedoch ändert, ist die Dimension der Ausgangsdaten, die nun statt

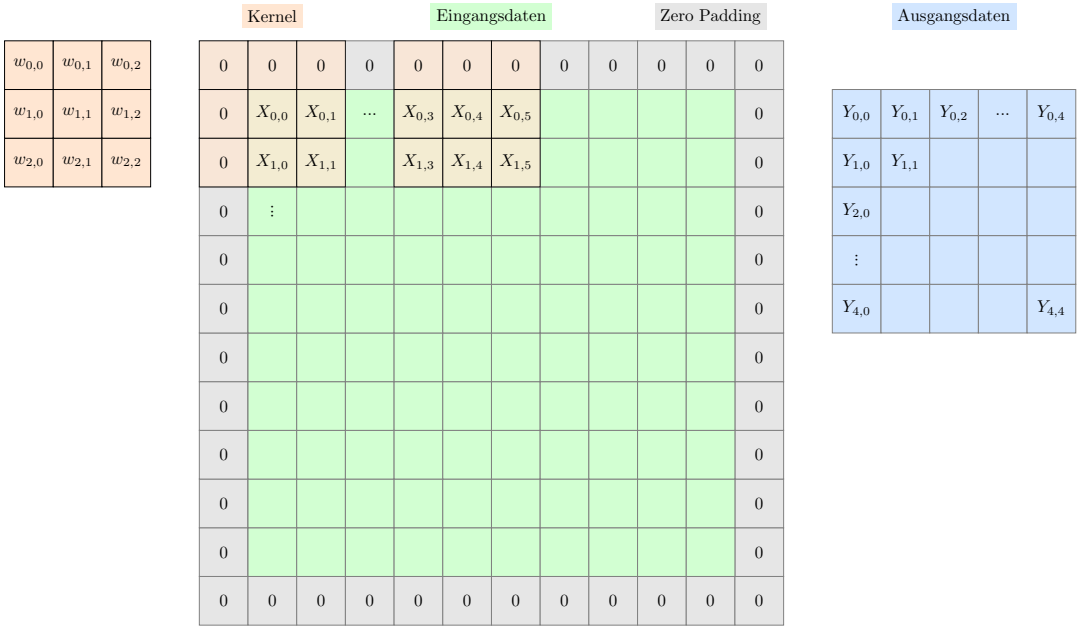


Abbildung 2.6.: Visuelle Repräsentation einer $\text{Conv}(3 \times 3)_{/2}$-Schicht mit Stride $S = 2$. Der erste auf den Eingangsdaten positionierte Kernel oben links liefert den Ausgangswert $Y_{0,0}$. Nun gleitet der Kernel immer um 2 Positionen weiter nach rechts, bevor er am Ende der Zeile wieder nach links springt und 2 Zeilen tiefer die Ausgangswerte $Y_{1,0}, Y_{1,1}, ...$ erzeugt. Der weiter rechts auf den Eingangsdaten abgebildete Kernel liefert den Ausgangswert $Y_{0,2}$.

der ursprünglichen 10×10 eine Dimension von 5×5 besitzt, was aus der Verwendung des Strides hervorgeht. Der Stride $S \in \mathbb{N}$ beschreibt, um wie viele Positionen der Kernel auf den Eingangsdaten weiter geschoben wird, bevor der nächste Ausgangswert berechnet werden soll. In Abschnitt 2.4.2 wäre $S = 1$, da sich der Kernel für jede nächste Berechnung um eine Position verschiebt. In Abbildung 2.6 wurde $S = 2$ gewählt, was dazu führt, dass die Dimension der Eingangsdaten (10×10) in Zeilen und Spalten halbiert der Dimension der Ausgangsdaten (5×5) entspricht. Wichtig ist zu erwähnen, dass in Abbildung 2.6 die erste und die dritte Position des Kernels für eine Berechnung gezeigt ist. Bei 2-dimensionalen Feature Maps *mit Zero Padding* gilt, dass sich für die Dimension der Eingangsdaten $M \times M$ mit $M \in \mathbb{N}$ und Stride $S \in \mathbb{N}$, die Dimension der Ausgangsdaten zu $M/S \times M/S$ ergibt. Der mathematische Zusammenhang ist wie folgt:

$$Y_{i,j} = \sum_{k \in \mathcal{K}} \sum_{\ell \in \mathcal{K}} X_{i \cdot S + k - d, j \cdot S + \ell - d} \cdot w_{k,\ell} + b, \quad d = \left\lfloor \frac{K}{2} \right\rfloor \tag{2.9}$$

Im Folgenden wird eine Faltungsschicht mit Stride mit $\text{Conv}(K \times K, N)_{/S}$ bezeichnet. Wie schon bekannt, steht $K \times K$ für die Dimension des Kernels. Der Wert $N \in \mathbb{N}$

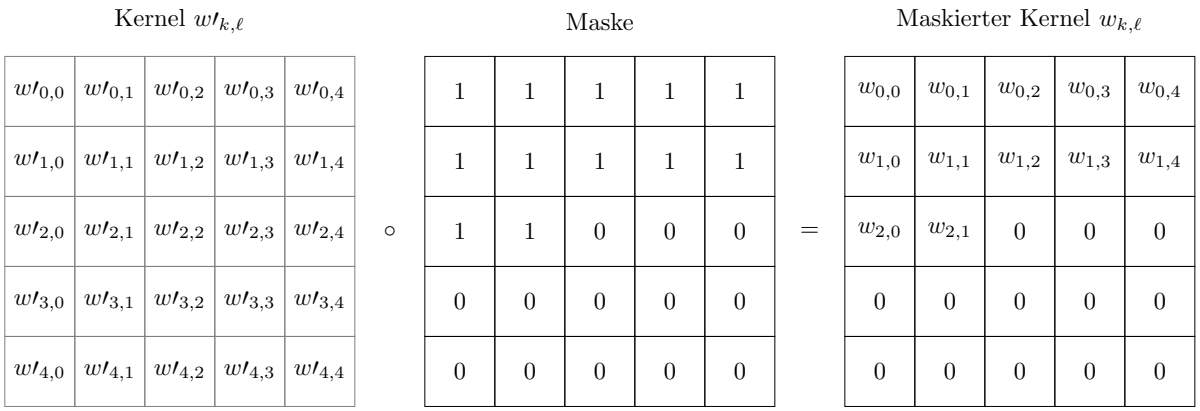


Abbildung 2.7.: **Konzept eines maskierten Kernels**. Dabei wird der ursprüngliche Kernel $w\prime_{k,\ell}$ elementweise mit der Maske multipliziert, um den maskierten Kernel zu erzeugen.

bezeichnet wieder die Anzahl der Kernels in der betreffenden Schicht. Der Wert $S \in \mathbb{N}$ bezeichnet den Stride.

2.4.5. Maskierte Faltungsschicht

In Abbildung 2.7 ist ein Kernel der Dimension $K \times K = 5 \times 5$ mit lernbaren Gewichten $w\prime_{k,\ell} \in \mathbb{R}$ und $k, \ell \in \mathcal{K}$ mit $\mathcal{K} = \{0, 1, 2, 3, 4\}$, sowie eine Maske derselben Dimension dargestellt. Die Werte der Maske $M_{k,\ell}$ sind hierbei entweder auf 1 oder auf 0 gesetzt. Multipliziert man nun Kernel $w\prime_{k,\ell}$ und Maske $M_{k,\ell}$ elementweise, erhält man den maskierten Kernel $w_{k,\ell}$. Der Effekt hinter jeder 1 in der Maske ist, dass der maskierte Kernel das dazugehörige Gewicht des Kernels übernimmt und so für die Berechnung der Ausgangswerte berücksichtigen soll. Bei jeder 0 resultiert, dass das dazugehörige Gewicht aus dem Kernel nicht in den maskierten Kernel übernommen wird, und so nicht in der Berechnung der Ausgangswerte berücksichtigt wird.

In Abbildung 2.8 wird der maskierte Kernel der Dimension $K \times K = 5 \times 5$ auf die Eingangsdaten angewendet. In grüner Farbe sind die Eingangsdaten $X_{i,j} \in \mathbb{R}$ mit $i, j \in \{0, 1, ..., 9\}$ der bespielhaften Dimension 10×10 gezeigt. Der maskierte Kernel mit seinen Gewichten $w_{k,\ell} \in \mathbb{R}$ und $k, \ell \in \mathcal{K}$ mit $\mathcal{K} = \{0, 1, 2, 3, 4\}$ hat die Farbe orange. Die Ausgangsdaten $Y_{i,j} \in \mathbb{R}$ mit $i, j \in \{0, 1, ..., 9\}$ der Dimension 10×10 sind in blauer Farbe dargestellt.

Der mathematische Zusammenhang unter Betrachtung des maskierten Kernels $w_{k,\ell}$ entspricht damit wieder einer gewöhnlichen 2-dimensionalen Faltung:

$$Y_{i,j} = \sum_{k \in \mathcal{K}} \sum_{\ell \in \mathcal{K}} X_{i+k-d, j+\ell-d} \cdot w_{k,\ell} + b, \quad d = \left\lfloor \frac{K}{2} \right\rfloor \tag{2.10}$$

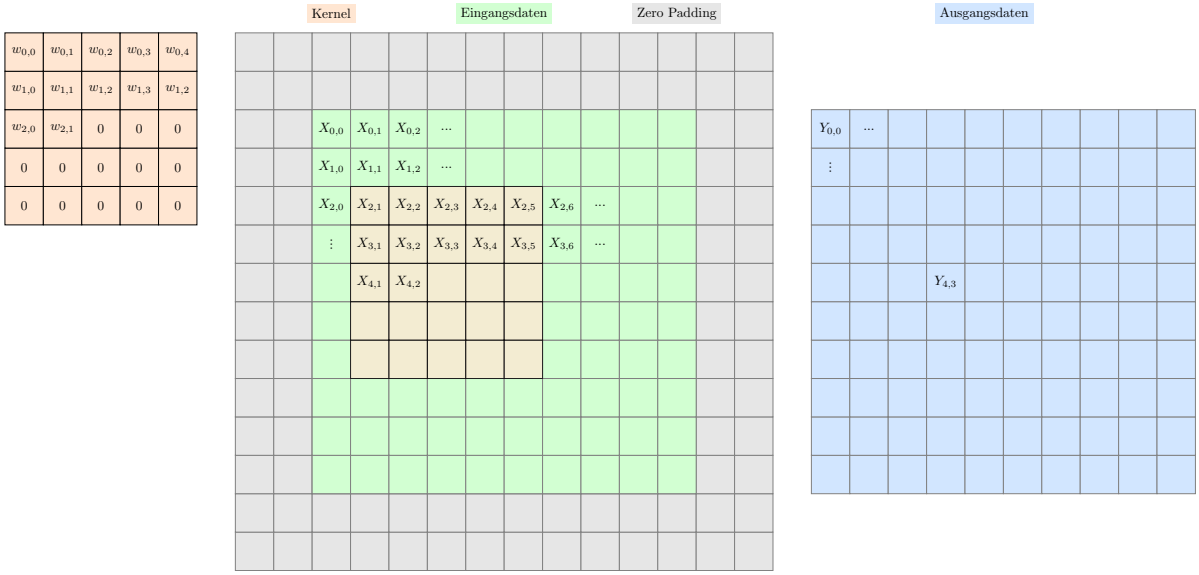


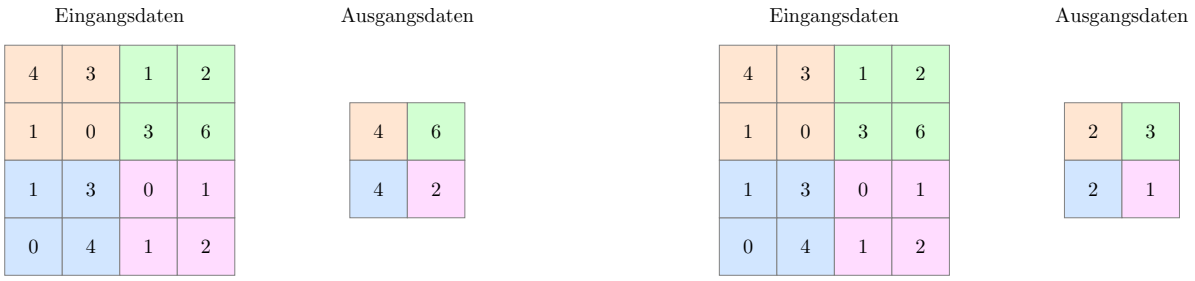
Abbildung 2.8.: Das Prinzip, wie der maskierte Kernel arbeitet, ist dasselbe wie bei einer gewöhnlichen Conv(5×5)-Schicht, siehe Abbildung 2.5. Der auf den Eingabedaten abgebildete Kernel liefert den Ausgangswert $Y_{4,3}$, der sich nur aus den von Null verschiedenen Kernelgewichten und den explizit notierten Eingabedaten $X_{2,1}...X_{4,2}$ darunter ergibt.

Ein solcher maskierter Kernel wie in Abbildung 2.8 betrachtet für jeden Ausgangsdatenpunkt $Y_{i,j}$ nur Eingangsdatenpunkte, die in Laufrichtung des Kernels (links $\rightarrow$ rechts, oben $\rightarrow$ unten) hinter ihm liegen. Solch eine Faltungsschicht mit maskiertem Kernel wird im Folgenden mit MaskedConv($K \times K, N$) bezeichnet.

2.4.6. Pooling-Schichten

In Abbildung 2.9a und Abbildung 2.9b sind Eingangsdaten der beispielhaften Dimension 4×4 gezeigt, die einzelnen Eingangsdatenpunkte werden im folgenden ähnlich wie in Abschnitt 2.4.2 mit $X_{i,j} \in \mathbb{R}$ mit $i, j \in \{0, 1, 2, 3\}$ bezeichnet. Die Ausgangsdaten haben eine Dimension von 2×2, die einzelnen Ausgangsdatenpunkte werden ebenfalls ähnlich wie in Abschnitt 2.4.2 $Y_{i,j} \in \mathbb{R}$ mit $i, j \in \{0, 1\}$ genannt.

In diesem Beispiel haben wir statt eines Kernels jeweils einfarbig markierte quadratische Bereiche der Dimension $K \times K = 2 \times 2$. Darüber hinaus haben wir einen Stride $S = 2$. Die einzelnen Positionen, die der Bereich zur Berechnung einnimmt, sind auf den Eingangsdaten farblich gekenzeichnet, die gleichfarbig gekennzeichneten Felder auf den Ausgangsdaten sind die resultierenden Ausgangsdaten. Auf alle Eingangsdatenpunkte, die im Bereich liegen, wird eine mathematische Funktion angewandt und der Wert, der daraus resultiert, wird als Wert des Ausgangsdatenpunkts für diese Bereichsposition definiert.



(a) MaxPooling$(2 \times 2)_{/2}$-Schicht. (b) AvgPooling$(2 \times 2)_{/2}$-Schicht.

Abbildung 2.9.: Visuelle Repräsentation von zwei Pooling-Schichten.

In Abbildung 2.9a wird Max-Pooling gezeigt, diese Schicht hat als mathematische Funktion das Maximum, also wird in dieser Schicht der Ausgangsdatenpunkt auf den maximalen Wert der vom Bereich betrachteten Eingangsdatenpunkte gesetzt. Der mathematische Zusammenhang ist also Folgender:

$$Y_{i,j} = \max_{k \in \mathcal{K}, \ell \in \mathcal{K}} X_{i \cdot S + k - d, j \cdot S + \ell - d}, \quad d = \left\lfloor \frac{K}{2} \right\rfloor \tag{2.11}$$

Im Folgenden wird eine Max-Pooling-Schicht mit MaxPooling$(K \times K, N)_S$ bezeichnet.

In Abbildung 2.9b wird Avgerage-Pooling gezeigt, diese Schicht führt als mathematische Funktion die Mittelwertbildung aus. In dieser Schicht wird also der Ausgangsdatenpunkt auf den Durchschnitt der vom Bereich betrachteten Eingangsdatenpunkte gesetzt. Der mathematische Zusammenhang ist also folgender:

$$Y_{i,j} = \frac{1}{K^2} \cdot \sum_{k \in \mathcal{K}} \sum_{\ell \in \mathcal{K}} X_{i \cdot S + k - d, j \cdot S + \ell - d}, \quad d = \left\lfloor \frac{K}{2} \right\rfloor \tag{2.12}$$

Im Folgenden wird eine Average-Pooling-Schicht mit AvgPooling$(K \times K, N)_S$ bezeichnet.

2.4.7. Pixel Shuffle

In Abbildung 2.10 ist eine Pixel-Shuffle-Schicht [17] mit Eingangsdaten in Form von $R \times R = 2 \times 2$ Feature Maps mit $X_{i,j}, X'_{i,j}, X''_{i,j}, X'''_{i,j} \in \mathbb{R}$ und $i, j \in \{0, 1\}$ jeweils der beispielhaften Dimension $h \times w = 2 \times 2$ gezeigt. Die einzelnen Feature Maps haben sind jeweils farblich markiert. Die Ausgangsdaten haben eine Dimension von $Rh \times Rw = 4 \times 4$ und die einzelnen Ausgangsdaten sind ebenfalls farblich markiert, dabei zeigt die Farbe und das Symbol denjenigen Datenpunkt an, von welcher Feature Map welcher Wert im jeweiligen Ausgangsdatenwert abgebildet wurde. Diese Schicht kombiniert die Eingangs-Feature Maps so, dass in diesem Fall in jeder ersten von zwei Ausgangsdatenzeilen die Reihenfolge orange, grün wiederholt wird mit den jeweiligen Daten der orangenen und grünen Feature Map. Also wird zuerst $X_{0,0}$ der orangenen, dann $X'_{0,0}$ der grünen, dann

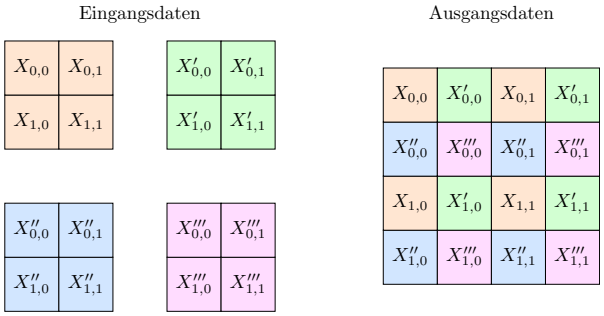


Abbildung 2.10.: Visuelle Repräsentation einer PixelShuffle(2)-Schicht.

$X_{0,1}$ der orangenen und zum Abschluss der Zeile $X'_{0,1}$ der grünen Feature Map genommen. Dasselbe Schema läuft mit jeder zweiten von zwei Ausgangsdatenzeilen mit der blauen und lila-farbigen Feature Map ab.

Die Funktion Pixel Shuffle bekommt R^2N Feature Maps übergeben mit $R, N \in \mathbb{N}$. Die Dimension jeder Feature Map muss gleich sein und resultiert in N Ausgangs-Feature-Maps entsprechender Dimension der Eingangsdatendimensionen faktorisiert durch R. Es gilt:

$$(h, w, R^2N) \mapsto (Rh, Rw, N) \tag{2.13}$$

mit $h, w, R, N \in \mathbb{N}$ wobei $h \times w$ die Dimension der Eingangs-Feature-Maps und R^2N die Anzahl der Eingangs-Feature-Maps bezeichnet.

Im Folgenden wird eine Pixel-Shuffle-Schicht mit PixleShuffle(R) bezeichnet.

2.4.8. Beispielnetz: LeNet-5

In Abbildung 2.11 ist als ein Beispielnetz das LeNet-5 [18] gezeigt. Das Netz bekommt als Eingabe ein Bild $\mathbf{x}$ in Graustufen der Dimension $28 \times 28 \times 1$ und hat am Ausgang $\mathbf{y}$ die Dimension 10. Die 10 Ausgangsknoten ergeben sich aus der Aufgabenstellung (engl. Task) für das LeNet-5, nämlich Klassifikation mit 10 Klassen. Zwischen Schichten, die miteinander verbunden sind, gibt es einen Pfeil, an dem die Ausgangsdimension der vorangehenden und somit auch die Eingangsdimension der nachfolgenden der beiden Schichten steht.

Die Operation "Flatten" bedeutet, dass die beispielhafte Eingangs-Dimension $M \times N \times R$ in eine eindimensionale Repräsentation geschrieben wird, die Dimension wäre dann MNR.

Das LeNet-5 Netz aus Abbildung 2.11 beinhaltet sowohl Faltungsschichten mit Operator Conv($K \times K, N$), Pooling-Schichten (AvgPool($K \times K, N)_S$), eine Flatten-Schicht, als auch vollvernetzte Schichten (FC(N)).

Im Folgenden wird diese Art der Repräsentation neuronaler Netze verwendet.

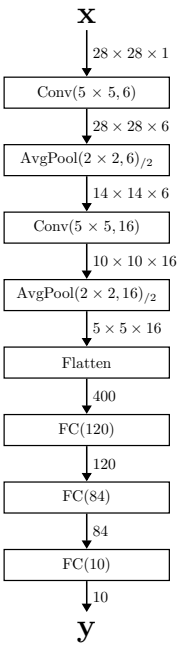


Abbildung 2.11.: Architektur des **LeNet-5** [18].

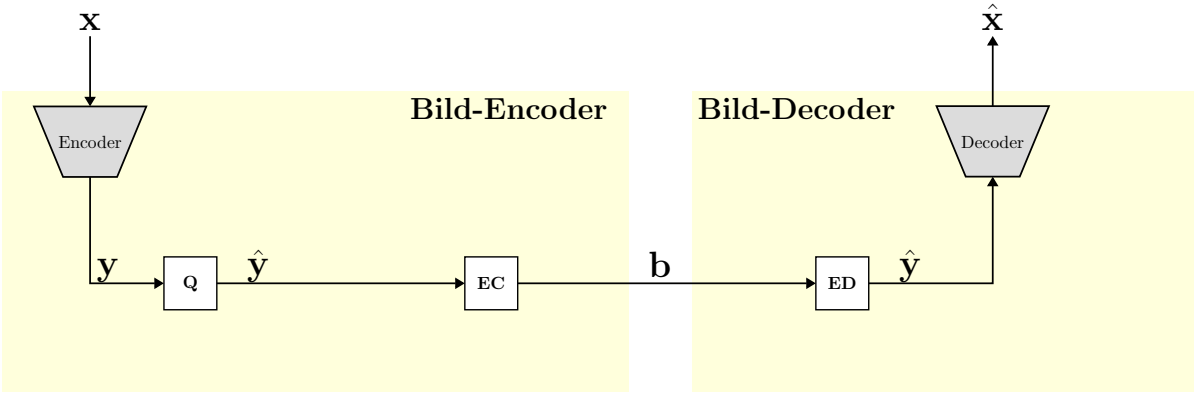


Abbildung 2.12.: Grundarchitektur eines Bildcodecs mit Entropiecodierung.

2.5. Grundarchitektur von Bildcodecs

In Abbildung 2.12 wird die Grundarchitektur eines Bildcodecs mit Entropiecodierung gezeigt. Wir sehen, dass die Eingabe $\mathbf{x}$ in einen sogenannten Encoder-Block gelangt, in dieser wird die Eingabe verarbeitet, bis sie als Ausgabe $\mathbf{y}$ ausgegeben wird. Dann wird $\mathbf{y}$ im Block **Q** zu $\hat{\mathbf{y}}$ quantisiert (vergleiche Abschnitt 2.3) und anschließend im Block **EC** entropiecodiert. Die nun entstandene Nachricht $\mathbf{b}$ wird nun an den Decoder übertragen, welcher diese in einem Entropiedecoder **ED** decodiert und somit das Signal $\hat{\mathbf{y}}$ erhält. Dieses wird dann im Decoder-Block zur rekonstruierten Eingabe $\hat{\mathbf{x}}$. Die Prozesse, welche im Encoder- und Decoder-Block ablaufen, nennt man Transformation. Das Zusammenspiel aus Entropie-Encoder und Entropie-Decoder wird Entropiecodierung genannt.

Die gezeigte Architektur gilt sowohl für klassische Bildcodecs [19, 20, 21, 22], als auch für neuronal gelernte Bildcodecs [23, 24, 25, 26, 27, 28]. All diesen verlustbehafteten Bild-Codecs ist eine Kombination aus Quantisierung und anschließender verlustloser Kompression durch Entropiecodierung gemeinsam. Für die weitere vorliegende Arbeit sind nur gelernte Bildcodecs relevant, deren Prinzipien nachfolgend vorgestellt werden.

2.6. Gelernte Bildcodecs

Im Folgenden werden zwei Architekturen von gelernten Bildcodecs gezeigt, welche auf die in Abschnitt 2.5 beschriebene Grundstruktur aufbauen und diese in ihrer Transformation (Encoder- und Decoder-Block) und ihrer Entropiecodierung verändern. In Abschnitt 2.6.1 wird die Struktur sogenannter Scale-Hyperprior-Netze erklärt und in Abschnitt 2.6.2 die der sogenannten Joint-Autoregressive Hierarchical-Prior-Netze.

2.6.1. Scale-Hyperprior-Netze

Neben klassischen Methoden zur Bildkomprimierung wie dem JPEG-Codec [20, 19], JPEG2000-Codec [21] oder dem JBIG-Codec [22] wird vermehrt an dem Ansatz der gelernten Bildkompression geforscht. Mittlerweile gibt es einige dieser gelernten Codecs, so zum Beispiel den von Mentzer at al. [23], Ballé at al. [24], Cheng et al. [25], Johnston at al. [26], Theis at al. [27] und Toderici at al. [28]. Gelernte Codecs zur Bildkompression ergänzen dabei klassische Verfahren meist um die Einbindung von CNNs, welche in Abschnitt 2.4 beschrieben wurden, da ihre Feature Maps als flexible Repräsentation eines Bildes geeignet sind.

In Abbildung 2.13 sehen wir die Struktur eines gelernten Bildcodecs, welche eine Erweiterung der Grundstruktur aller Codecs ist, wie sie in Abschnitt 2.5 beschrieben wurde. In diesem Fall ist es ein Scale-Hyperprior-Netz nach [24]. Wir sehen, dass wir auf der linken Seite einen Bild-Encoder und auf der rechten Seite einen Bild-Decoder haben. Der Bild-Encoder bekommt als Eingabe ein Bild $\mathbf{x}$ und komprimiert es in eine Binärnachricht $\mathbf{b}^R$ und $\mathbf{b}^H$, welche an den Bild-Decoder übermittelt werden. Der Bild-Decoder dekomprimiert diese Binärnachrichten und gibt das rekonstruierte Bild $\hat{\mathbf{x}}$ aus. Wir sehen in Abbildung 2.13, dass der Bild-Encoder das Eingangsbild $\mathbf{x}$ in einem *Encoder* verarbeitet. Dieser *Encoder* besteht meist aus mehreren CNN-Schichten, von denen jede in der Regel Feature Maps mit reduzierter Dimension generiert, dafür aber eine größere Anzahl dieser Feature Maps als Ausgabe besitzt. Die Ausgabe des Encoders wird mit $\mathbf{y}$ bezeichnet. Wir sehen, dass $\mathbf{y}$ nun in einem Block $\mathbf{Q}$ zu $\hat{\mathbf{y}}$ quantisiert wird, aber auch in einem *Hyper-Encoder* weiterverarbeitet wird. Dieser *Hyper-Encoder* ist der Anfang der Hyperprior-Struktur, die später dem arithmetischen Encoder wichtige statistische Eigenschaften über das Bild liefern soll, um dieses bitrateneffizienter zu komprimieren. Der *Hyper-Encoder* besteht, wie der Encoder, aus meist mehreren CNN-Schichten, die

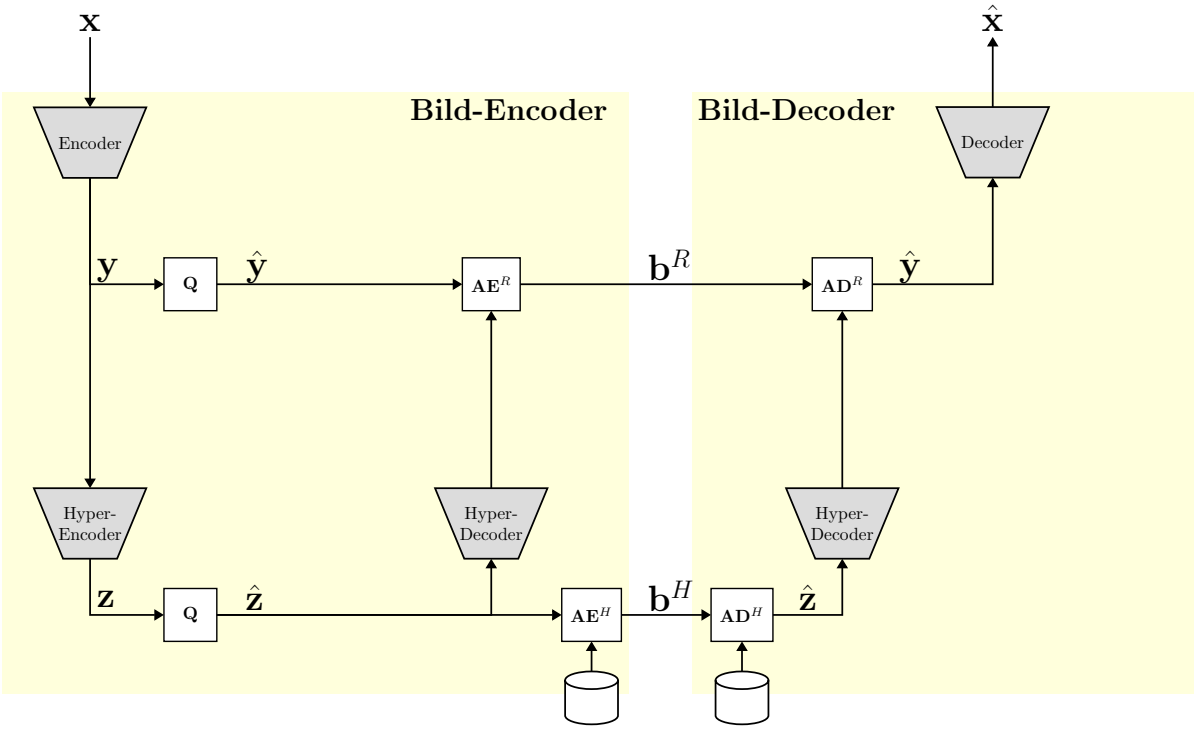


Abbildung 2.13.: Architektur eines **Scale-Hyperprior-Netzes** nach [24].

meist die Dimension der einzelnen Feature Maps weiter verringern. Die Ausgabe des Hyper-Encoders wird mit $\mathbf{z}$ bezeichnet, welche von einem Quantisierer $\mathbf{Q}$ zu $\hat{\mathbf{z}}$ quantisiert wird. In einem *Hyper-Decoder*, welcher wieder aus CNN-Schichten besteht, wird nun $\hat{\mathbf{z}}$ verarbeitet und liefert die für den arithmetischen Encoder nötige Wahrscheinlichkeitsverteilung, welche sich in unserem Fall des Scale Hyperpriors bei $\mathbf{AE}^R$ und $\mathbf{AD}^R$ auf die räumliche Verteilung der Standardabweichungen beschränkt. Der arithmetische Encoder $\mathbf{AE}^R$ codiert nun auf Grundlage der Wahrscheinlichkeitsverteilung des Hyperpriors die quantisierte Darstellung des Bildes $\hat{\mathbf{y}}$ zu dem Bitstrom $\mathbf{b}^R$. Im Hyperprior wird $\hat{\mathbf{z}}$ nun ebenfalls von einem arithmetischen Encoder $\mathbf{AE}^H$, welcher eine gelernte und nach dem Lernen unveränderte Wahrscheinlichkeitsverteilung besitzt (siehe "Tonnen"-Symbol in Abbildung 2.13), zu dem Bitstrom $\mathbf{b}^H$ codiert.

Vom Bild-Encoder werden die beiden Bitströme $\mathbf{b}^R$ und $\mathbf{b}^H$ nun an den Bild-Decoder übertragen, welcher im Hyperprior nun $\mathbf{b}^H$ mit einem arithmetischen Decoder $\mathbf{AD}^H$, mit derselben gelernten Wahrscheinlichkeitsverteilung wie der arithmetischen Encoder $\mathbf{AE}^R$, decodiert wird und somit wieder in $\hat{\mathbf{z}}$ resultiert. Anschließend folgt wie auf seiten des Bild-Encoders der *Hyper-Decoder* (dieser entspricht auch dem aus dem Bild-Encoder, wird also zwischen Bild-Encoder und Bild-Decoder geteilt) welcher die Wahrscheinlichkeitsverteilung hier für den arithmetischenn Decoder $\mathbf{AD}^R$ liefert[2]. Der arithmetische

[2]Dabei sind die Wahrscheinlichkeitsverteilungen im Bild-Encoder und im Bild-Decoder, die aus dem Hyperprior kommen, genau dieselben, da $\hat{\mathbf{z}}$ durch denselben Block geht und auf Bild-Encoder-Seite verlustlos zu demselben $\hat{\mathbf{z}}$ wie auf der Seite des Bild-Encoders rekonstruiert wird.

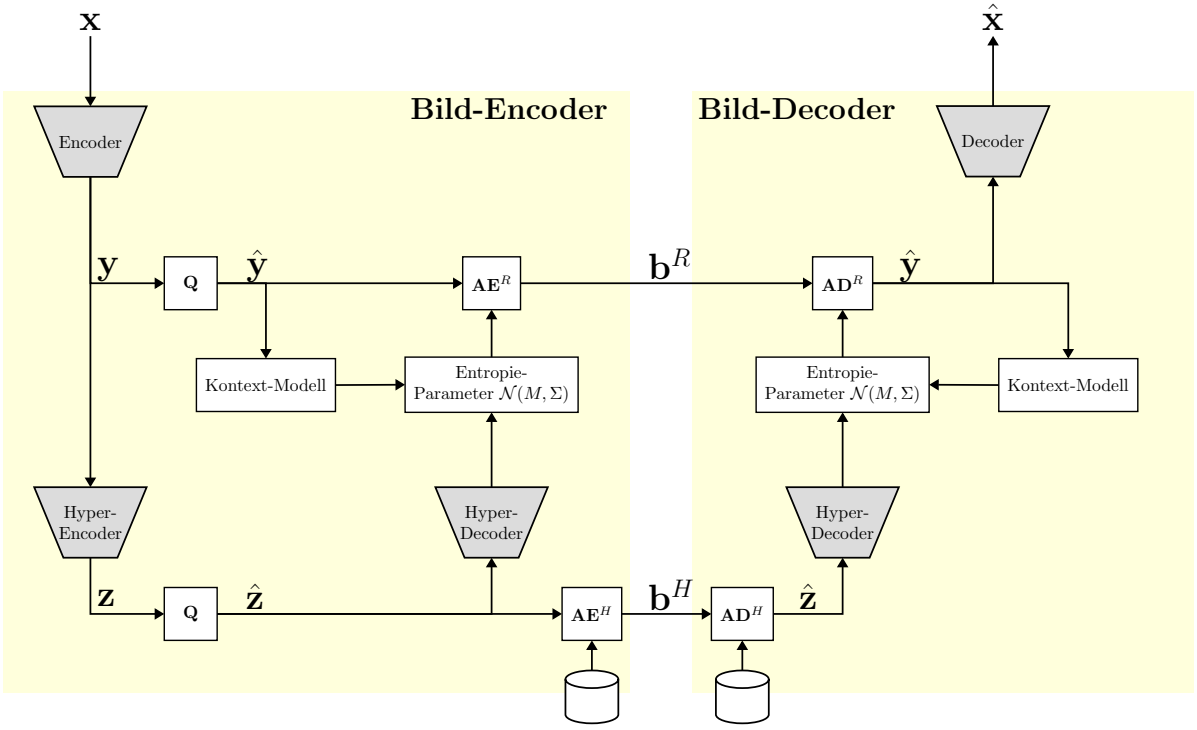


Abbildung 2.14.: Architektur eines **Joint-Autoregressive Hierarchical-Prior-Netzes** nach [29].

Decoder $\mathbf{AD}^R$ liefert nun auf Grundlage der Wahrscheinlichkeitsverteilung und der Eingabe des Bitstroms $\mathbf{b}^R$ das verlustlos rekonstruierte $\hat{\mathbf{y}}$. Am Ende des Bild-Decoders wird $\hat{\mathbf{y}}$ durch einen *Decoder*, welcher das Gegenstück zum Encoder aus dem Bild-Encoder ist, nun zum Bild $\hat{\mathbf{x}}$ derselben Dimension wie das Eingangsbild $\mathbf{x}$ rekonstruiert. Das Ausgabebild $\hat{\mathbf{x}}$ ist nun das rekonstruierte und mit Verlust behaftete Bild.

2.6.2. Joint-Autoregressive Hierarchical-Prior-Netze

In Abbildung 2.14 ist die Struktur eines Joint-Autoregressive Hierarchical-Prior-Netzes nach [29] gezeigt. Dieses Netz baut sehr stark auf das des Scale-Hyperprior-Netzes aus Abschnitt 2.6.1 auf. Wir sehen jedoch auch ein paar Unterschiede zwischen dem Scale-Hyperprior-Netz aus Abbildung 2.13 und dem Joint-Autoregressive Hierarchical-Prior-Netz aus Abbildung 2.14. Letzteres wurde nämlich um jeweils zwei Blöcke im Bild-Encoder und im Bild-Decoder erweitert. Die Netzgewichte der Blöcke *Kontext-Modell*, *Entropie-Parameter* $\mathcal{N}(\mathbf{M}, \mathbf{\Sigma})$ und *Hyper-Decoder* werden dabei vom Bild-Encoder und vom Bild-Decoder geteilt. Dies bedeutet, dass im Bild-Decoder diese Blöcke die identischen gelernten Parameter wie im Bild-Encoder besitzen. Das *Kontext-Modell* besteht aus einer wie in Abschnitt 2.4.5 beschriebenen maskierten Faltungsschicht, welche auf Grundlage schon gesehener Pixel das nächste Pixel vorhersagen will. Die Eingabe für das *Kontext-Modell* ist $\hat{\mathbf{y}}$, die Ausgabe wird direkt an den zweiten neuen Block (*Entropie-*

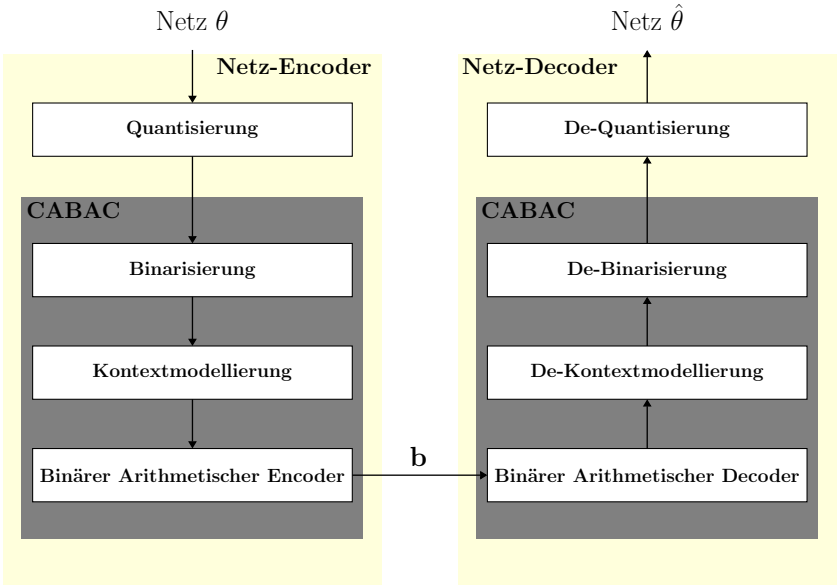


Abbildung 2.15.: Aufbau des Neural Network Codings.

Parameter) weitergeleitet. Der *Entropie-Parameter*-Block bekommt als Eingabe auch die Ausgabe des Hyper-Decoders und besteht ebenfalls aus CNN-Schichten. Am Ende gibt dieser Block die Mittelwerte $\mathbf{M}$ und die räumliche Verteilung der Standardabweichungen $\mathbf{\Sigma}$ an den arithmetischen Encoder $\mathbf{AE}^R$ und den arithmethischen Decoder $\mathbf{AD}^R$ weiter, welche diesem als dessen Wahrscheinlichkeitsverteilung $P(\hat{y}_k)$ dienen, siehe Abschnitt 2.2.

2.7. Netzkompression mittels Neural Network Coding

Der MPEG-Standard MPEG-7 Part 17 oder auch ISO/IEC 15938-17, welcher Nerual Network Coding genannt wird, ist eine Codierung für neuronale Netze, die diese sowohl komprimieren (codieren) und in einen Bitstrom umwandeln, als auch diesen Bitstrom wieder in ein funktionstüchtiges Netz dekomprimieren (decodieren) kann. Dabei soll die Genauigkeit des wieder decodierten Netzes weitgehend der des ursprünglichen Netzes entsprechen.

In Abbildung 2.15 ist der Aufbau des Neural Network Codings dargestellt. Auf der linken Seite ist der Encoder gezeigt, welcher ein Netz θ als Eingabe erhält und dieses im Quantisierungsblock quantisiert. Dabei werden Skalare durch einen sogenannten gleichförmigen Rekonstruktions-Quantisierer (URQ, uniform reconstruction quantizer), wie er in Abschnitt 2.3 erklärt wurde, quantisiert. Vektoren werden durch eine sogenannte abhängige Quantisierung (DQ, dependent quantization) quantisiert, genauer durch die Trellis-codierte Quantisierung (TCQ) [30]. Die TCQ arbeitet mit Hilfe zweier Skalarquantiserer (URQ) und einer Zustandsmaschine, die die Ergebnisse der beiden Ska-

qp	-40	-60	-80	-100	-120
Δ	$9{,}7656 \cdot 10^{-4}$	$3{,}0518 \cdot 10^{-5}$	$9{,}5367 \cdot 10^{-7}$	$2{,}9802 \cdot 10^{-8}$	$9{,}3132 \cdot 10^{-10}$

Tabelle 2.1.: Beispielwerte für qp nach (2.14) in Δ umgerechnet.

larquantisierer zur Quantisierung kombiniert. Dieser Aufbau ist komplexer als der des URQs, ist dafür aber für das Quantisieren von Vektoren und Matrizen besser geeignet. In beiden Quantisierungsverfahren wird die Quantisierungsschrittweite Δ folgendermaßen berechnet[3] [31]:

$$\Delta = (4 + (qp \mod 4)) \cdot 2^{\lfloor \frac{qp}{4} \rfloor - 2}. \tag{2.14}$$

Der Parameter $qp \in \mathbb{N}$ ist dabei der sogenannte Quantisierungsparameter. Beispielwerte für qp und die daraus folgenden Werte für Δ sind in Tabelle 2.1 zu finden.

Wurde das Netz quantisiert, werden die quantisierten Ausgangssymbole, wie in Abbildung 2.15 gezeigt, in einen sogenannten CABAC-Block [32] eingespeist, wo zuerst jedes Symbol mit Hilfe eines gebildeten Binärbaums durch eine Binärdarstellung ersetzt wird. Die Abkürzung CABAC steht für "Context-based Adaptive Binary Arithmatic Coding" (kontextbasiertes adaptives binäres arithmetisches Codieren). Die Binärdarstellungen der Symbole werden auch Bins genannt. Die Bins werden an die Kontextmodellierung weitergeleitet, wo jedem Bin ein Kontextmodell zugeschrieben wird. Die Kontextmodelle geben dem jeweiligen Bin eine geschätzte Auftrittswahrscheinlichkeit, welche nach jeder Codierung der binären Darstellung durch den binären arithmetischen Encoder angepasst wird. Wurde eine Binärdarstellung des Bins "$\hat{w}$" durch den binären arithmetischenn Encoder codiert, steigt die Auftrittswahrscheinlichkeit im Kontextmodell für $\hat{w}$, die Auftrittswahrscheinlichkeiten aller anderen Kontextmodelle wird dabei gesenkt. Die Auftrittswahrscheinlichkeit der jeweiligen Bins ist wichtig für den binären arithmetischen Encoder, da er den Bitstrom auf Grundlage der Binärdarstellung und der dazugehörigen Auftrittswahrscheinlichkeiten erzeugt, siehe auch Abschnitt 2.2. Der Decoder, der in Abbildung 2.15 auf der rechten Seite dargestellt ist, erhält den übertragenen Bitstrom des Encoders und führt für alle Berechnungen des Encoders die jeweils komplementären Berechnungen durch, um am Ende das Netz θ mit etwas Quantisierungsfehler der Gewichte als $\hat{\theta}$ zu rekonstruieren.

2.8. Adaptoren für vollverbundene Schichten

Adaptoren sind zusätzliche Schichten, die in ein bereits bestehendes, trainiertes Netz eingebaut werden, um es zum Beispiel an einen Domain-Shift (Änderung der Anwendungs-

[3] Im Paper von Becking et al. [31] ist eine Formel in Appendix A für Δ gegeben, die aber nicht mit der letztendlichen Implementierung übereinstimmt. Wir haben die Formel aus dem GitHub-Projekt verwendet (`https://github.com/fraunhoferhhi/nncodec/wiki/Usage`).

domäne) zu adaptieren. Dabei wird das ursprüngliche Netz (häufig Backbone genannt) im sogenannten Fine-Tuning (ein Nachtraining) eingefroren, und nur die Adaptoren werden trainiert. Durch ihr Lernen soll sich das Netz an die neue Domäne anpassen. Der Vorteil dabei liegt darin, nicht den gesamten Backbone trainieren zu müssen, sondern nur einige Adaptoren mit wenigen Gewichten. Ein weiterer Vorteil von Adaptoren liegt im Kontext dieser Arbeit darin, dass nur ihre trainierten Gewichte für ein Netzupdate übertragen werden müssen und nicht das ganze Netz.

Die am weitesten verbreitete Art von Adaptoren sind Adaptoren für vollverbundene Schichten. Dabei kann ein Adaptor zum Beispiel zwischen zwei vollverbundene Schichten im Backbone eingesetzt werden [33, 2, 3]. Er kann auch als eine Residual-Verbindung im Netz eingebaut werden [34]. Viele Adaptoren bestehen aus mehreren Schichten, die meist einen Bottleneck (eine zunächst dimensionsreduzierende und dann wieder dimensionsvergrößernde Struktur) aufweisen. Eine besondere Art dieser Bottleneck-Adaptoren sogenannten sind die LoRA-Adaptoren. Sie reduzieren die benötigte Parameteranzahl für Adaptoren drastisch [35, 36].

2.9. Metriken

2.9.1. Bits Per Pixel (bpp)

Die Metrik bits per pixel (bpp), die in dieser Arbeit häufig verwendet wird, beschreibt mit wie vielen Bits im Durchschnitt ein Farb-Pixel eines Bildes codiert ist. Hätten wir nun ein komprimiertes Bild, welches unkomprimiert aus 8×8 Pixeln besteht und in seiner komprimierten Form 100 Bits belegt, folgt daraus, dass es mit $\frac{100\,\text{bit}}{8\times 8\,\text{pixel}} = 1{,}5625$ bpp codiert wurde.

2.9.2. Peak Signal-to-Noise Ratio (PSNR)

Mit dem sogenannten peak signal-to-noise ratio (PSNR) kann die Qualität der Rekonstruktion eines Bildes gemessen werden. Dabei wird dieser Wert geringer, je mehr Störung im Vergleich zum Ursprungsbild enthalten ist. Um das PSNR zweier Bilder (ursprüngliches Bild und rekonstruiertes Bild) zu berechnen, benötigen wir die Höhe $H \in \mathbb{N}$ der Bilder (in Pixeln), die Breite $W \in \mathbb{N}$ der Bilder und den maximalen Grauwert $x_{\max}$, den ein Farbkanal eines Pixels annehmen kann, was im Fall von 8-Bit Bildern (welche in dieser Arbeit betrachtet werden) 255 ist. Mit unserem Originalbild $\mathbf{x}$ und unserem rekonstruierten Bild $\hat{\mathbf{x}}$ ergibt sich das PSNR nun folgendermaßen [37]:

$$PSNR \quad (\text{dB}) \triangleq 10 \log_{10} \frac{x_{\max}^2}{MSE} \tag{2.15}$$

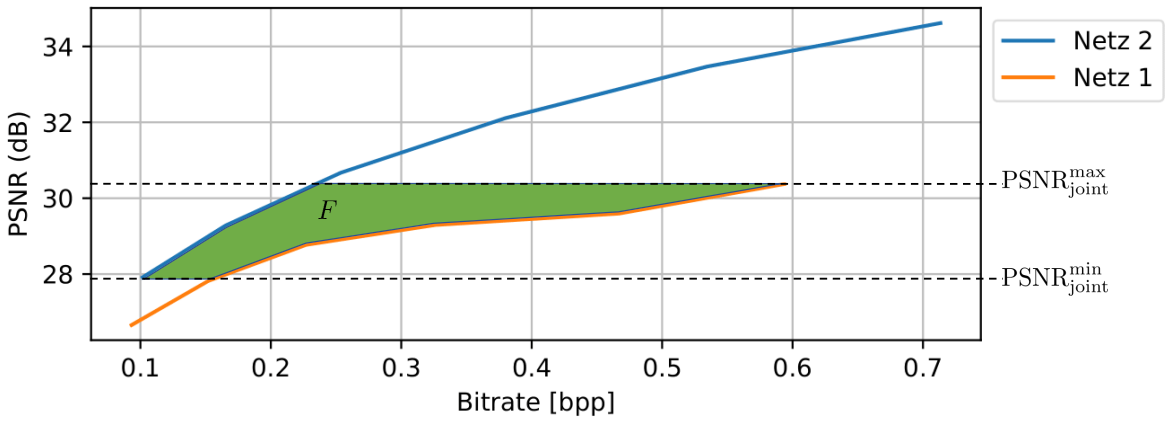


Abbildung 2.16.: Bildliche Verdeutlichung der bei der Berechnung der BD-Rate aufgespannten Fläche F. Hier ist die Interpolation durch lineare Verbindungen zwischen den Punkten realisiert.

mit:

$$MSE \triangleq \frac{1}{H \cdot W} \sum_{i \in \mathcal{I}} \sum_{j \in \mathcal{J}} ||\hat{\mathbf{x}}(i,j) - \mathbf{x}(i,j)||^2, \tag{2.16}$$

wobei die Mengen $\mathcal{I} = \{0, 1, ..., H{-}1\}$ und $\mathcal{J} = \{0, 1, ..., W{-}1\}$ die Zeilen- und Spaltenindizes enthalten und $\hat{\mathbf{x}}(i,j), \mathbf{x}(i,j) \in \mathbb{R}^3$ eines Bildpixel mit den 3 Farbkanälen beschreibt.

2.9.3. Bjøntegaard-Delta-Rate (BD-Rate)

Die Bjøntegaard-Delta-Rate (BD-Rate) von Bjøntegaard et al. [38] ist eine Metrik, um zwei Bild- oder Video-Codecs miteinander zu vergleichen. Dabei werden von beiden Codecs mehrere sogenannte Qualitätspunkte ermittelt; diese zeigen für unterschiedliche Kompressionsraten (bpp) deren Rekonstruktionsqualität (PSNR). Mittels Interpolation wird eine Kurve durch diese gelegt. Dabei gibt es unterschiedliche Arten, diese Kurve zu interpolieren. In dieser Arbeit wird die Interpolation durch das sogennante Akima-Verfahren von Akima et al. [39] berechnet. Wichtig zu beachten ist, dass sich die Interpolationen der beiden Codecs für eine saubere Berechnung nicht schneiden dürfen. Entscheidend ist nun die Fläche, die diese zwei Interpolationskurven aufspannen. Dabei fängt die Fläche F aber erst bei dem kleinsten gemeinsamen PSNR-Wert PSNR$^{\text{min}}_{\text{joint}}$ an und endet bei dem größten gemeinsamen PSNR-Wert PSNR$^{\text{max}}_{\text{joint}}$. Dieses Vorgehen ist beispielhaft in Abbildung 2.16 gezeigt. Ausgehend von dieser Fläche wird berechnet, wie groß im Durchschnitt die Distanz der Bitrate des betrachteten Codecs zum Vergleichscodec bei demselbem PSNR ist. Dabei steht ein negativer Wert für eine Verbesserung des betrachteten Codecs zum Vergleichscodec, da somit unsere Bitrate (bpp) im Schnitt bei demselben PSNR geringer ausfällt.

3. Stand der Technik

Im Folgenden wird beschrieben, wie Adaptoren in CNNs bereits eingesetzt wurden, um ein bereits bestehendes Netz anzupassen. Wir werden auch die Unterschiede in Aufgabenstellung und Umsetzung zu diesen Ansätzen und unserer Arbeit erläutern.

Adaptoren wurden in erster Linie zum Fine-Tuning von Transformern z.B. in Anwendungen der natürlichsprachigen Sprachverarbeitung (engl. natural language processing) verwendet, um diese bei einem Domain-Shift auf diesen anzupassen [33, 40, 41, 42]. Dabei wird das ursprüngliche Netz im Fine-Tuning-Training eingefroren und nur die Adaptoren lernen. Erst in jüngerer Zeit wird auch der Einsatz von Adaptoren in Faltungsnetzen untersucht. So untersuchen Chen et al. [4] den Einsatz eines Adaptors, welcher aus Faltungsschichten besteht. Der Aufbau des Adaptors aus [4] ist in Abbildung 3.1 gezeigt. Zu sehen ist eine Bottleneck-Struktur mit einer tiefenweisen Faltungsschicht (engl. depth-wise convolution, siehe Abschnitt 2.4.3) DWConv($v \times v, D, M/R$) als erste Schicht, welche die Dimension von $h \times w \times M$ auf $h \times w \times M/R$ verringert. Bei der Wahl von $R \in \mathbb{N}$ ist zu beachten, dass sich M/R wieder als natürliche Zahl ergeben muss. Auf dieses Reduzieren der Anzahl der Feature Maps folgt eine nichtlineare Aktivierungsfunktion und im Anschluss eine Faltungsschicht Conv($1 \times 1, N$), um wieder auf die gewünschte Ausgangsdimension zu kommen. In der Regel führt die letzte Schicht dabei eine Vergrößerung der Anzahl der Feature Maps durch. Dieser Adaptor wird in [4] sowohl parallel als auch sequentiell (in diesem Fall *nach* der zu adaptierenden Schicht) an einzelne Faltungsschichten angehängt. Der Adaptor wird aber auch parallel und sequentiell zu Residual-Blöcken untersucht, wobei er im sequentiellen Fall nach der letzten Faltung im Residual-Block mit einem zusätzlichen Bypass eingefügt wird. Chen et al. [4] haben untersucht, inwieweit dieser Adaptor mit seinen unterschiedlichen Möglichkeiten der Einbindung in Aufgaben der Klassifikation, der semantischen Segmentierung und Objektdetektion bei einem Domain-Shift helfen kann.

Gezeigt wurde, dass bei Einsatz von solchen Adaptoren ein Adaptieren an einen Domain-Shift in Klassifikations- und dichten Vorhersagenetzten eine vergleichbare Performanz erreicht wie Fine-Tuning des gesamten Netzes. Dabei ist dieser Ansatz anpassbar an verschiedene Domänen, modellunabhängig und benötigt nur wenige zusätzliche Parameter. Allerdings ist nicht sicher, wie dies für größere Domain-Shifts als den untersuchten aussieht.

Diese Umsetzung von Faltungsadaptoren wurde von Shen et al. [5] aufgegriffen und in das sogenannte Cheng-2020 [25] Bild-Codec-Netz eingefügt. Dabei wurden Adaptoren wie vorher beschrieben *seriell* in Residual-Blöcke des Decoders (Vergleich Abbildung

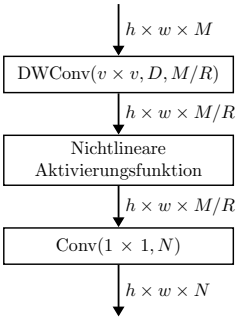


Abbildung 3.1.: Adaptor aus [4].

2.14) eingefügt. *Dabei wurden sie nur im Decoder und nicht zusätzlich auch im Encoder eingesetzt.* Mit Hilfe der Adaptoren ist ein Domain-Shift von natürlichen Bildern zu Bildern mit Bildschirminhalten mit einer Verbesserung in der BD-Rate von über 2dB (bei Bildschirminhalten) gegenüber dem Baseline-Netz möglich.

Untersucht wurde nach unserem Wissen und jetztigem Stand auch nicht, wie effektiv Faltungsadaptoren sind, um Updates an Faltungsnetzen bitrateneffizient zu gestalten.

Der in Kapitel 4 und von dieser Arbeit erstmals und noch ohne Kenntnis von [4] und [5] entworfene Adaptor für Faltungsschichten weicht an entscheidenen Stellen von den genannten Vorarbeiten ab. Anders als in [4] verwenden wir *keine tiefenweise Faltungsschicht*, sondern Kernels, die über alle M Eingangs-Feature-Maps ausgedehnt sind. Damit hat in unserem Vorschlag *jede* der M/R Ausgangs-Feature-Maps eine Abhängigkeit von *allen* M Eingangs-Feature-Maps. Anstelle einer nicht lernbaren nicht-linearen Aktivierungsfunktion [4] *setzen wir im Bottleneck effizient eine Faltung mit kleinem Kernel ein*, verbleiben damit linear, aber nutzen lernbare Intelligenz. Die Faltungsschicht am Ausgang ist bei uns und in [4] identisch.

Der wesentliche Unterschied zu Shen et al. [5] besteht in dieser Arbeit darin, dass wir (a) einen *anderen Adaptor* untersuchen, (b) ihn immer *parallel* zum Residual-Block einsetzen und (c) ihn in jeden Residual-Block *sowohl im Decoder als auch im Encoder* einfügen.

4. Methoden

Im Folgenden werden die Methoden, die wir in dieser Arbeit verwenden, sowie unsere "Storyline" behandelt und erklärt. In Abschnitt 4.1 behandeln wir die in dieser Arbeit verwendeten Datensätze und ihre gestückelten Aufteilungen (engl. Splits). Anschließend wird in Abschnitt 4.2 das in dieser Arbeit verwendete Baseline-Netz vorgestellt. Zum Schluss folgt in Abschnitt 4.3 der Adaptor, welcher in dieser Arbeit eingeführt wird, mit Illustration seiner Einsatzorte.

4.1. "Storyline" und Daten der Arbeit

Um die "Storyline", d.h. die Aufgabe (engl. Task), der sich diese Arbeit stellt, gut zu verstehen, ist es ratsam, sich mit den in dieser Arbeit verwendeten Daten auszukennen, weswegen wir diese vorab vorstellen.

Die in dieser Arbeit verwendeten Datensätze sind in Tabelle 4.1 gezeigt. Verwendet wird hauptsächlich der Vimeo-90k Datensatz [43]. Er besteht aus 64.612 Containern, von denen jeder eine Bildsequenz mit je 7 aufeinanderfolgenden Bildern enthält, womit wir auf eine Gesamtanzahl von 452.284 Bildern im Trainingsdatensatz kommen. Die Aufteilung des Datensatzes in Trainings- und Validierungsdaten erfolgt folgendermaßen: 90% der Daten bilden den sogenannten Vimeo-90k-train*-Split, 10% den Vimeo-90k-val*-Split. Wichtig ist, dass bei jeder Aufteilung nie ein Container geteilt wird, so bekommt ein Split also immer nur volle Container. Der Vimeo-90k-test-Datensatz beinhaltet 7.824 Container, was 54.768 Bilder bedeutet. Es gibt noch weitere Stückelungen der Datensätze, welche durch ein Anhängen der Reduktion z.B. "/7" markiert wird. Dabei gilt für alle Splits, welche ein "/7" beinhalten, dass für diesen aus jedem Container nur das erste Bild betrachtet wird. Im Fall von Vimeo-90k-train*/70 wird nur das erste Bild jedes zehnten Containers betrachtet, so dass insgesamt der Datensatz nur noch ein siebzigstel des ursprünglichen Vimeo-90k-train*-Splits beinhaltet. Zum Testen wird neben Vimeo-90k-test, Vimeo-90k-test/7 auch der Kodak-Bildsatz [44] verwendet, da dieser in der Bildcodierung sehr häufig verwendet wird und damit eine bessere Vergleichbarkeit ermöglicht wird.

Die "Story", die wir in dieser Arbeit verfolgen, ist anders als die in [4] und [5], da wir keinen Domain-Shift von zum Beispiel natürlichen Bildern zu Bildern mit Bildschirminhalten behandeln. Was in dieser Arbeit untersucht wird, ist, dass ein Netz mit schwächerer Performanz durch ein Update (welches bitrateneffizeint sein soll) zu einem

Daten-Split / Datensatz	Symbol	# Bilder	Prozentsatz
Vimeo-90k-train	$\mathcal{D}^{\text{train}}$	452.284	100,00%
→Vimeo-90k-train*	$\mathcal{D}^{\text{train}*}$	407.057	90,00%
→Vimeo-90k-train*/7	$\mathcal{D}^{\text{train}*/7}$	58.151	12,86%
→Vimeo-90k-train*/70	$\mathcal{D}^{\text{train}*/70}$	5.815	1,29%
→Vimeo-90k-val*	$\mathcal{D}^{\text{val}*}$	45.227	10,00%
→Vimeo-90k-val*/7	$\mathcal{D}^{\text{val}*/7}$	6.461	1,43%
Vimeo-90k-test	$\mathcal{D}^{\text{test}}$	54.768	100,00%
→Vimeo-90k-test/7	$\mathcal{D}^{\text{test}/7}$	7.824	14,29%
Kodak	$\mathcal{D}^{\text{Kodak}}$	24	100,00%

Tabelle 4.1.: Zahl der Bilder in den verwendeten **Vimeo-90k-Daten-Splits**, sowie im verwendeten Test-Set **Kodak** in dieser Arbeit.

stärker performanten Netz werden soll. Dabei nutzen wir Adaptoren und den NNCodec, um dieses Update bitrateneffizient zu gestalten, da nur die Adaptoren gelernt werden und komprimiert übertragen werden. Die komprimierten Adaptoren werden dann beim Empfänger dekomprimiert und an das bestehende Netz an richtiger Stelle eingefügt. Im Folgenden wird das Netz, welches auf der Baseline beruht und das schwächer performende ist, auf $\mathcal{D}^{\text{train}*/70}$ trainiert und mit $\mathcal{D}^{\text{val}*/7}$ im Training evaluiert. Das Netz, welches eine stärkere Performanz besitzt und ebenfalls auf der Baseline beruht, wird auf $\mathcal{D}^{\text{train}*/7}$ trainiert und ebenfalls im Training durch $\mathcal{D}^{\text{val}*/7}$ evaluiert.

4.2. Baseline-Bildcodec

Das Netz, welches in dieser Arbeit betrachtet wird und im Folgenden häufig Baseline genannt wird, ist das von Bégaint et al. [46] implementierte und leicht umgeänderte sogenannte Cheng2020-Netz von Cheng et al. [25]. Das Netz basiert dabei auf der Architektur der Joint-Autoregressive Hierarchical-Prior-Netze aus Abschnitt 2.6.2 und ist in seinem Aufbau in Abbildung 4.1 gezeigt. Einzelne Blöcke des Netzes werden in Abbildung 4.2 im Detail gezeigt. Das Baseline-Netz basiert auf der grundlegenden Joint-Autoregressive Hierarchical-Prior-Struktur, welche im Abschnitt 2.6.2 beschrieben wurde. Einige thematisch zusammenhängende Blöcke sind in diesem Netz zusätzlich grau hinterlegt und beschriftet. Die Beschriftung der Signale im Netz sind dieselben wie in der grundlegenden Struktur.

In Abbildung 4.1 sind vier Arten von Blöcken farblich blau und grün markiert. Diese finden sich mit selbiger farblicher Kennzeichnung mit Details in Abbildung 4.2 wieder. Die drei mit Blautönen markierten Blöcke sind Residual-Blöcke, also solche mit einem Bypass, so dass das Eingangssignal am Ende zum Ausgangssignal des Blocks addiert wird. Im Fall des ResBlock(N) (Abbildung 4.2a) ist das deutlich zu erkennen, im Fall von

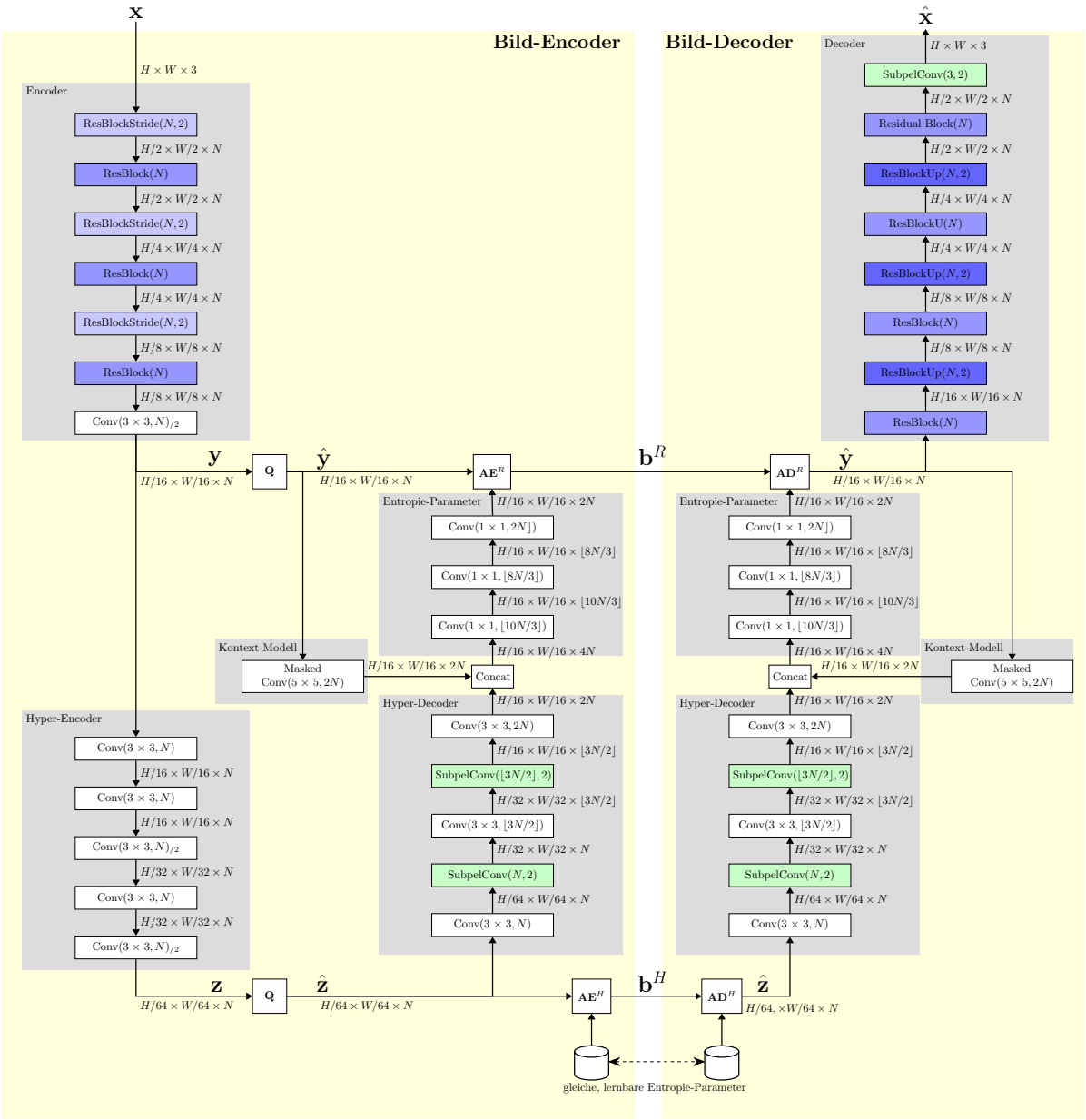
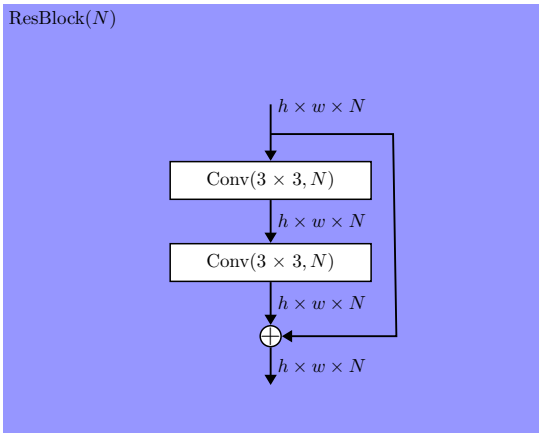
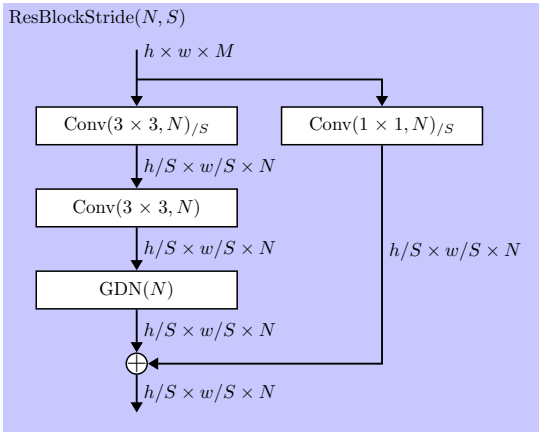


Abbildung 4.1.: **Netzwerkstruktur des Baseline-Netzes**. Blöcke mit der Beschriftung "**Q**" sind Quantisierungsblöcke, "**AE**" steht für "Arithmetischer Encoder" und "**AD**" für "Arithmetischer Decoder". Die Architekturen der einzeln farblich hinterlegten Blöcke sind in Abbildung 4.2 gezeigt.

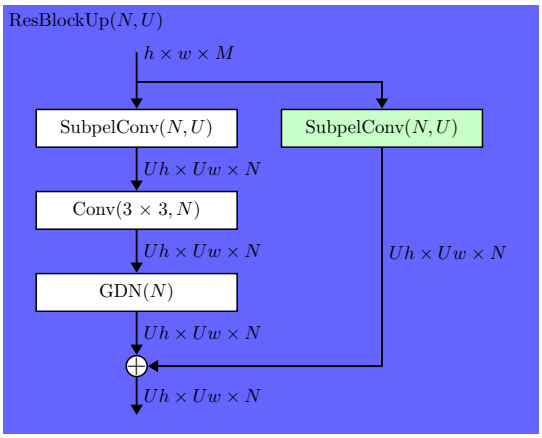
ResBlockStride(N, S) (Abbildung 4.2b) ist der Bypass jedoch durch eine Schicht ergänzt, da im ResBlockStride(N, S) eine notwendige Reduktion der Dimension von $h \times w \times M$ auf $h/S \times w/S \times N$ stattfindet. Würden wir die Dimensionen nicht wie hier im Bypass anpassen, könnten wir diese Daten nicht auf den Ausgang des Blocks addieren. Diese Notwendigkeit der Dimensionsanpassung im Bypass ist ebenfalls im ResBlockUp(N, U) vorhanden (Abbildung 4.2c), da dieser die Eingangsdimension $h \times w \times M$ auf $Uh \times Uw \times N$



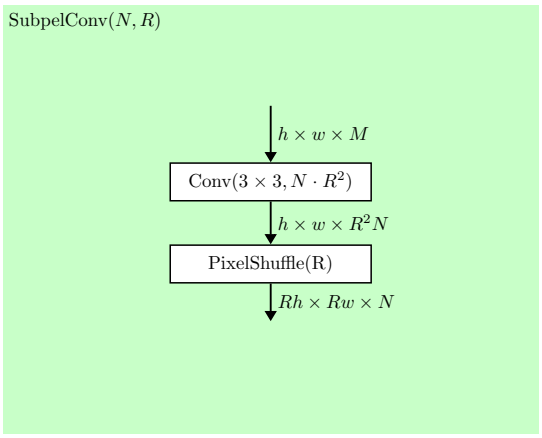
(a) Architektur von ResBlock(N).



(b) Architektur von ResBlockStride(N, S) mit Stride S.



(c) Architektur von ResBlockUp(N, U) mit Auflösungsskalierungsfaktor U.



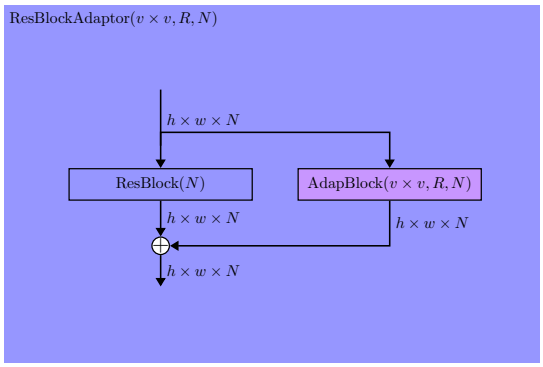
(d) Architektur von SubpelConv(N, R) mit Auflösungsskalierungsfaktor R.

Abbildung 4.2.: Architekturen der wichtigsten **in dieser Arbeit neu entwickelten Blöcke** in Abbildung 4.1. Der Block GDN(N) beschreibt eine generalisierte Normalisierungstransformation nach [45].

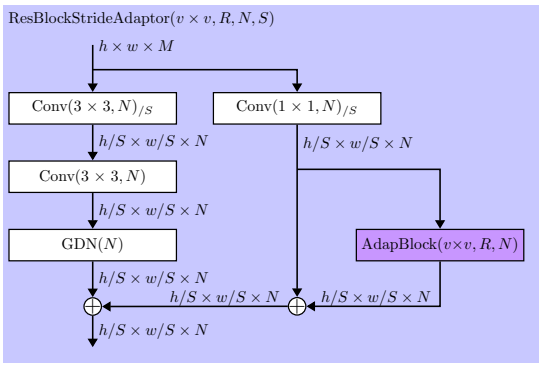
erhöht. Der verwendete Block SubpelConv(N, U) (Abbildung 4.2d) ist dabei ein Block, der die Dimension von $h \times w \times M$ auf $Uh \times Uw \times N$ anpassen kann. Die Funktion der Schicht PixelShuffle(R) ist in Abschnitt 2.4.7 erklärt.

4.3. Neuer Adaptor für CNNs

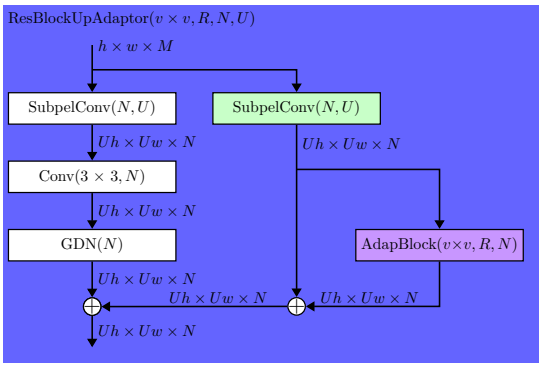
In diesem Abschnitt wird der in dieser Arbeit vorgeschlagene Adaptor mit seinen verschiedenen Möglichkeiten der Einstellungen und seiner Einbindung im Baseline-Netz besprochen. In Abschnitt 4.3.1 wird die Einbindung und Verortung der Adaptoren im



(a) Einbindung des AdapBlock($v \times v, R, N$).



(b) Einbindung des AdapBlock($v \times v, R, N$).



(c) Einbindung des AdapBlock($v \times v, R, N$).

Abbildung 4.3.: **Einsatz des neu vorgeschlagenen Adaptor-Blocks** mit der Bezeichnung AdapBlock($v \times v, R, N$) (aus Abbildung 4.4) in die Blöcke ResBlock(N) (Abbildung 4.2a), ResBlockStride(N, S) (Abbildung 4.2b) und ResBlockUp(N, U) (Abbildung 4.2c). Die Kernel-Größe $v \times v$ wird auf einer um den Faktor R reduzierten Anzahl N/R von Feature Maps angewandt.

Baseline-Netz erläuert. Darauf folgt in Abschnitt 4.3.2 die Erklärung des Aufbaus und der Einstellungsmöglichkeiten des in dieser Arbeit verwendeten Adaptors.

4.3.1. Verortung unseres neuen Adaptors im Bildcodec-CNN

Der in dieser Arbeit verwendete Adaptor wird in drei der Blöcke aus Abbildung 4.2 eingesetzt. Die Blöcke sind der ResBlock(N) (Abbildung 4.2a), der ResBlockStride(N, S) (Abbildung 4.2b) und der ResBlockUp(N, U) (Abbildung 4.2b), da diese, wie in Abschnitt 4.2 bereits beschrieben, Residual-Blöcke sind und wir in dieser Arbeit die Adaptoren parallel an den Bypass der Residual-Blöcke anhängen. Dieses Vorgehen wurde teils bereits in Abschnitt 2.8 und Abschnitt 3 behandelt.

In Abbildung 4.3 ist der Einsatz des in dieser Arbeit verwendeten Adaptors, welcher hier

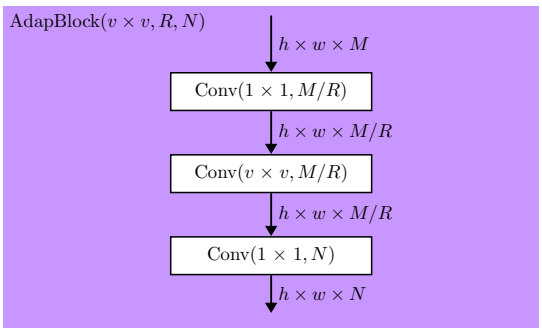


Abbildung 4.4.: Der **in dieser Arbeit vorgeschlagene Adaptor** mit der Bezeichnung AdapBlock($v \times v, R, N$).

und im Folgenden als AdapBlock($v \times v, R, N$) bezeichnet wird, gezeigt. In der Abbildung 4.3a ist der Einsatz des Adaptors im neu bezeichneten ResBlockAdaptor($v \times v, R, N$) gezeigt. Da sich die Datendimension bei einem ResBlock(N) nicht ändert, wird der Adaptor parallel zu diesem eingesetzt und die Ausgänge des ResBlock(N) und des AdapBlock($v \times v, R, N$) werden vor der Ausgabe des ResBlockAdaptor($v \times v, R, N$) addiert. In Abbildung 4.3b ist der Einsatz des Adaptors in den Block ResBlockStride(N, S) gezeigt, dieser wird parallel zum Bypass des Blocks eingefügt, jedoch erst nach der Dimensionsanpassung. Der Block mit dem nun integrierten Adaptor wird im Folgenden ResBlockStride($v \times v, R, N, S$) genannt. In Abbildung 4.3c sehen wir den Einsatz des Adaptors in den ResBlockUp(N, U), dabei wird dieser wie in Abbildung 4.3b gezeigt im ResBlockStrideAdaptor($v \times v, R, N, S$) parallel zum Bypass jedoch nach der Dimensionsanpassung durch SubpelConv(N, U) eingesetzt.

Wir untersuchen zwei Methoden, bei der ersten werden die Adaptoren nur wie in Abbildung 4.3a parallel zum ResBlock(N) in das Baseline-Netz eingesetzt, diese Netze werden im Folgenden Adaptornetz 1 genannt. In der zweiten Methode, werden wie in Abbildung 4.3 in alle Residual-Blöcke Adaptoren eingesetzt, im Folgenden werden solche Netze Adaptornetz 2 genannt.

Neben der *geänderten Architektur* ist das Neue beim Einsatz dieser Adaptoren, dass diese nicht nur wie von Chen et al. [5] in den Decoder eingesetzt werden, sondern hier *auch im Encoder*. Des Weiteren fügen wir unsere Adaptoren *parallel* zum Bypass der Residual-Blöcke ein. Auch die *Aufgabe unserer Adaptoren ist eine andere*, sie sollen nicht einen Domain-Shift lernen [5, 34], sondern in ihrer Domäne lernen und durch Lernen auf mehr Trainingsdaten das Netz in seiner Performanz steigern.

4.3.2. Aufbau des neuen Adaptors

Der in dieser Arbeit verwendete und unabhängig von [4, 5] entwickelte Adaptor AdapBlock($v \times v, R, N$) ist in Abbildung 4.4 gezeigt. Er besteht aus drei Faltungsschichten, von denen die erste einen 1×1-Kernel besitzt. Damit kann man eine Bottleneck-

Struktur erzielen und durch die damit einhergehende Dimensionsreduktion Parameter einsparen. Die Aufgabe der ersten Faltungsschicht ist die Datenreduktion auf lediglich M/R Feature Maps. Die Größe $R \in \mathbb{N}$ kann im AdaptorBlock eingestellt werden, um die Reduktion anzupassen. Dabei ist es notwendig, dass $M/R \in \mathbb{N}$ gilt. In der zweiten Schicht hat der Adaptor einen Kernel der Größe $v \times v$. Die Größe $v \in \mathbb{N}$ kann dabei eingestellt werden, um die Größe des Kernels in dieser Schicht anzupassen und somit dem Adaptor mehr Parameter zur Verfügung zu stellen. Die Anzahl der Feature Maps wird durch die zweite Faltungsschicht nicht verändert, erst durch die dritte wird die Anzahl der Feature Maps wieder erhöht, so dass sie der gewünschten Anzahl N entspricht, die für den Einsatz des Adaptors im Parallelbetrieb notwendig ist. Im Folgenden wird $M = N$ gelten, somit stimmt die Ausgabedimension mit der Eingabedimension überein.

Der Aufbau unseres Adaptors unterscheidet sich von denen aus [5, 34] darin, dass wir *keine tiefenweise Faltung* verwenden, sondern nur Faltungsschichten mit einem 1×1-Kernel am Ein- und Ausgang. Des Weiteren verwenden wir *keine nichtlineare Funktion* im Bottleneck, sondern eine weitere Faltungsschicht, von der wir die Kernelgröße einstellen können. Damit kommen wir auf *insgesamt drei Schichten* und nicht auf zwei wie in [5, 34]. Wir können unsere Adaptoren mit zwei verstellbaren Parametern anpassen, während die Adaptoren in [5, 34] nur einen einstellbaren Parameter besitzen. Somit erhalten wir *mehr Freiheitsgrade in unserem Adaptor* und können z.B. mehr Gewichte lernen, als es mit anderen Adaptoren möglich wäre.

Der in Abbildung 4.4 gezeigte Adaptor AdapBlock$(v \times v, R, N)$ hat die folgende Anzahl an Parametern:

$$\begin{aligned} L^{\text{AdapBlock}} = \quad & (1 \times 1) \times M \cdot \frac{M}{R} + \frac{M}{R} \\ & + (v \times v) \times \frac{M}{R} \cdot \frac{M}{R} + \frac{M}{R} \\ & + (1 \times 1) \times \frac{M}{R} \cdot N + N \\ = & \left(M + v^2 \cdot \frac{M}{R} + N + 2 \right) \frac{M}{R} + N. \end{aligned} \tag{4.1}$$

5. Evaluation und Diskussion

In diesem Abschnitt werden die zuvor besprochenen Methoden im Einsatz geprüft und mit anderen Methoden verglichen. In Abschnitt 5.1 wird das Trainings- und Evaluations-Setup unserer Experimente erläutert. Darauf folgen in Abschnitt 5.2 erste Experimente zu bestehenden und von uns trainierten Netzen, die wichtige Erkenntnisse und Vorentscheidungen für den Rest der Arbeit liefern. Im Anschluss werden in Abschnitt 5.3 Methoden des Fine-Tunings bereits bestehender Schichten im Netz untersucht. In Abschnitt 5.4 werden Methoden, welche im Fine-Tuning eingefügte Adaptoren trainieren, untersucht. Danach werden in Abschnitt 5.5 alle vorgestellten Methoden miteinander verglichen. Zum Schluss werden in Abschnitt 5.6 die für am besten befundenen Methoden für ein Netzupdate verwendet.

5.1. Trainings- und Evaluations-Setup

Im Folgenden werden Details zum Training und zur Evaluation unserer Netze dargelegt.

Auf der Maschine, auf der wir arbeiten, läuft das Betriebssystem `OpenSUSE` mit der Versionsbezeichnung `Leap 15.5`, welches eine Linux-Distribution ist. Wir verwenden als Programmiersprache Python, und nutzen dabei die Version 3.11.7. Zum Erstellen und Trainieren von neuronalen Netzen nutzen wir PyTorch in der Version 2.2.2 und PyTorch Cuda in der Version 0.17.2. In manchen Anwendungen nutzen wir auch Torch Vision in der Version 0.17.2. Die Bibliothek CompressAi [46], in welcher sich unser Baseline-Netz Cheng2020(-anchor) [25] befindet, verwenden wir in der Version 1.2.6.dev0. Als Implementierung des Neural Network Coding-Standards [6] verwenden wir die Implementierung von Becking et al. [31] in der Version 0.3.1.

Falls nicht anders beschrieben, gilt im Folgenden, dass für das Baseline-Netz (siehe Abbildung 4.1) die Kanalgröße $N = 128$ ist. Vor dem ersten Training wird das Baseline-Netz mit der CompressAi-internen Qualität "3" als untrainiertes Netz und mit der internen Metrik MSE geladen. Im Training wird dieses, falls nicht anders beschrieben, immer mit 70 Epochen einer Batch-Größe von 16 und einer Patch Size (Größe des Bildausschnittes eines Trainingsbildes, welcher dem Netz zum Lernen gegeben wird) von 256×256 trainiert. Als Lernrate für das Netz wird $1 \cdot 10^{-4}$ und als sogenannte aux-Lernrate (welche den zusätzlichen Loss des arithmetischen Encoders $\mathbf{AE}^H$ und Decoders $\mathbf{AD}^H$ bestimmt) wird $1 \cdot 10^{-3}$ verwendet. Als Optimierer haben wir in beiden Fällen Adam [47] verwendet. Der Tradeoff-Hyperparameter $\lambda \in \mathbb{R}$ steuert die Gewichtung zwischen der Optimierung

Qualitätspunkt	0	1	2	3	4	5	6
λ-Werte	0,0100	0,0018	0,0035	0,0067	0,0130	0,0250	0,0483

Tabelle 5.1.: λ-Werte zur Erzeugung unterschiedlicher Qualitätspunkte durch Einsatz in der Verlustfunktion (5.1).

einer angestrebt niedrigen geschätzen Bitrate ($\mathcal{R}$) und der Optimierung einer angestrebt niedrigen Verzerrung ($\mathcal{D}_{\text{MSE}}$). Damit hat λ einen Einfluss auf die Loss-Funktion (deutsch Verlustfunktion), mit Hilfe derer der Verlust $\mathcal{L}$ folgendermaßen berechnet wird:

$$\mathcal{L} = \lambda \cdot x_{\text{max}}^2 \cdot \mathcal{D}_{\text{MSE}} + \mathcal{R}. \tag{5.1}$$

Wie in Tabelle 5.1 gezeigt, haben wir für λ eine Auswahl von sieben verschiedenen Werten, welche in Netze mit unterschiedlichen Qualitätspunkten resultieren. Deswegen bezeichnen wir, wie in der Tabelle gezeigt, die unterschiedlichen Werte für λ mit den Qualitätspunkten 0 bis 6.

5.2. Vorbereitende Experimente

In diesem Abschnitt werden einige vorbereitende Experimente durchgeführt. Diese dienen dazu, eigene Implementierungen zu testen, wichtige Parameter für die restliche Arbeit zu bestimmen und den in dieser Arbeit verwendeten Datensatz Vimeo-90k zu untersuchen. In Abschnitt 5.2.1 werden Experimente mit dem Baseline-Netz gemacht und mit diesem auch der Vimeo-90k-Datensatz untersucht. Darauf folgt in Abschnitt 5.2.2 die Untersuchung des in dieser Arbeit verwendeten Neuronalen Netz-Codec NNCodec [31] und die Ermittlung eines geeigneten Quantisierungsparameters.

5.2.1. Bildcodec

In Abbildung 5.1 ist gezeigt, wie das von [46] bereitgestellte, bereits gelernte Cheng-2020-Netz zur Bildcodierung bei einer Evaluation von unserer Seite mit dem Kodak-Datensatz, im Vergleich zu derselben Evaluation von Bégaint et al. [46], abschneidet. Die von uns gefundenen Datenpunkte sind mit Kreisen über der Original-Grafik von Bégaint et al. [46] dargestellt. Zu sehen ist, dass diese mit denen von Bégaint et al. recht gut übereinstimmen. So konnten wir unsere Implementierung der Evaluation überprüfen und bestätigen, das diese nach allem menschlichen Ermessen korrekt ist.

In Abbildung 5.2 ist der Loss auf $\mathcal{D}^{\text{val}*}$ beim Training des Baseline-Modells [46] dargestellt. Trainiert wurde über 10 Epochen mit dem Trainingsdatensatz $\mathcal{D}^{\text{train}*}$. Zu sehen ist ein absteigender Loss, was ein progressives Lernen des Netzes anzeigt.

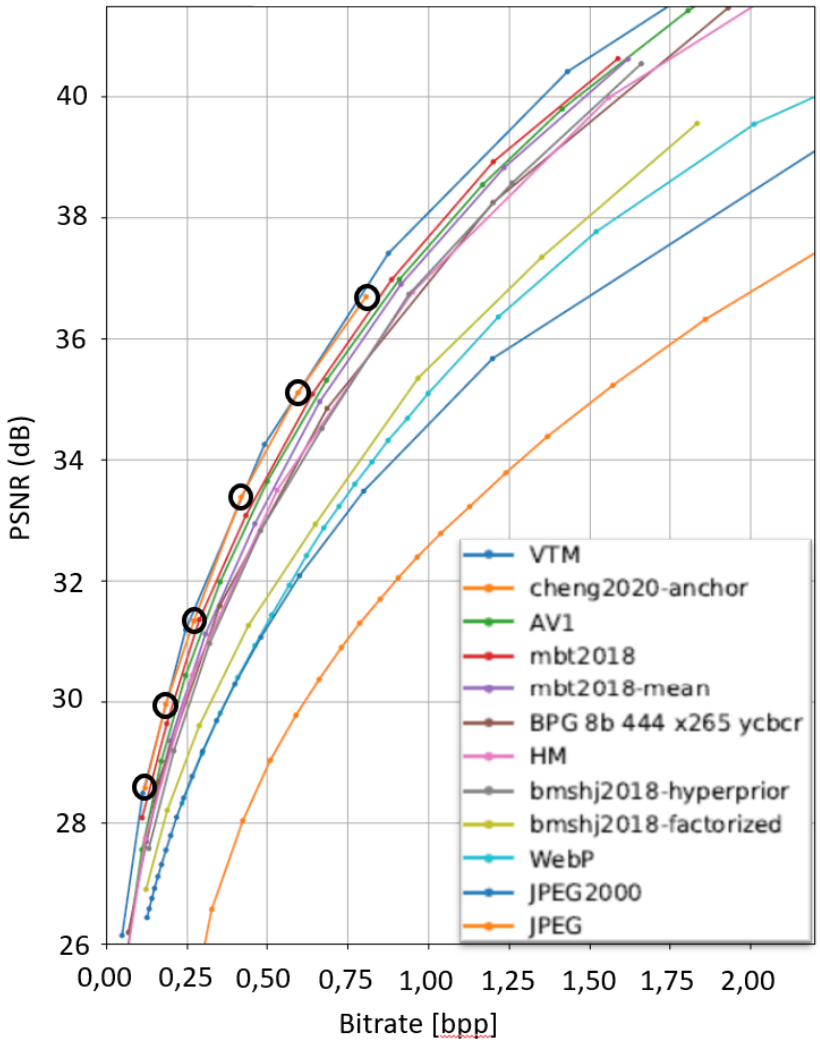


Abbildung 5.1.: **PSNR (dB) auf Kodak**; Grafik aus [46] und angepasst. Die schwarzen Marker dokumentieren die Performanz des Cheng2020-anchor-Modells [46] (im Weiteren "Baseline"genannt), welches eine Abwandlung des Modells aus [25] ist, **mit den aus [46] zur Verfügung gestellten Gewichten.**

Da das Training mit 10 Epochen bei $\mathcal{D}^{\text{train}*}$ schon eine große Zeit in Anspruch nimmt und die Loss-Kurve nur wenige Punkte enthält, untersuchen wir, wie wichtig es ist, alle sieben Bilder aus einer Sequenz des Datensatzes $\mathcal{D}^{\text{train}*}$ zu trainieren, im Vergleich zu nur dem ersten Bild einer Sequenz ($\mathcal{D}^{\text{train}*/7}$) bei gleich vielen Lernschritten. Die gleich vielen Lernschritte haben in dem Fall, in dem nur die ersten Bilder einer Sequenz gelernt werden, eine sieben mal höhere Epochenanzahl und somit sieben mal mehr Punkte auf der Loss-Kurve zur Folge. Die Vermutung dabei ist, dass aufgrund der niedrigen Varianz der sieben aufeinander folgenden Bilder einer Sequenz das Netz im Training keinen

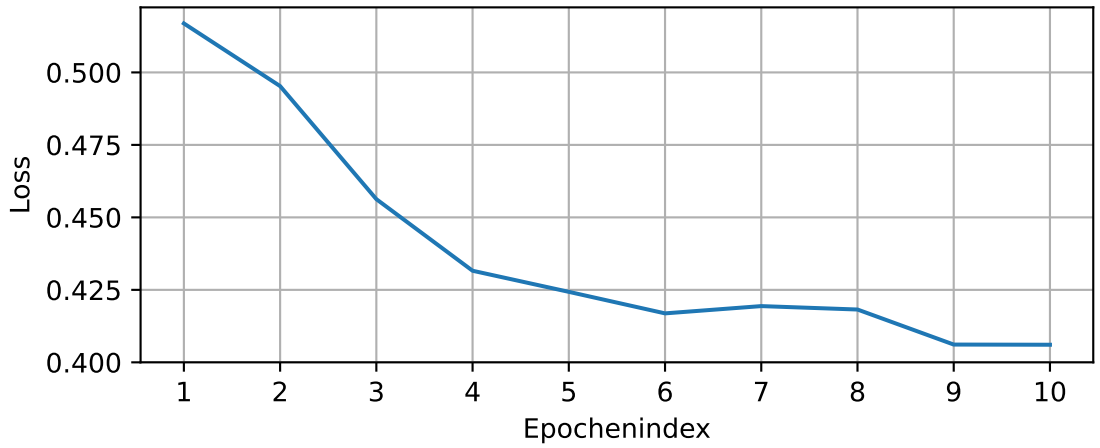
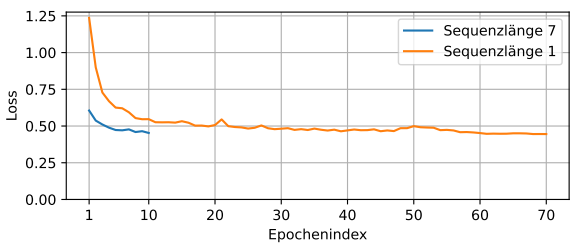
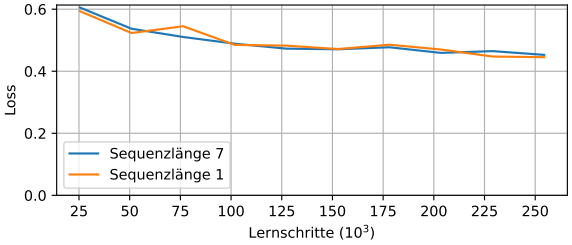


Abbildung 5.2.: **Loss**-Kurve auf $\mathcal{D}^{\text{val}*}$ beim eigenen Training des Baseline-Modells [25] auf $\mathcal{D}^{\text{train}*}$ mit Qualitätspunkt 0.



(a) Bei beiden Trainings werden insgesamt gleich viele Bilder dargeboten, daher 10 Epochen für $\mathcal{D}^{\text{train}*}$ und 70 Epochen für $\mathcal{D}^{\text{train}*/7}$.



(b) Bei beiden Trainings bedeutet ein Lernschritt eine Batchgröße von $B = 16$ gesehenen Bildern. Das Training auf $\mathcal{D}^{\text{train}*}$ läuft auf insgesamt 10 Epochen, auf $\mathcal{D}^{\text{train}*/7}$ insgesamt 70 Epochen.

Abbildung 5.3.: **Loss**-Kurve auf $\mathcal{D}^{\text{val}*}$ beim Training der Baseline auf $\mathcal{D}^{\text{train}*}$ mit Qualitätspunkt 0. Vergleich beim Training mit einer Sequenzlänge von 7 ($\mathcal{D}^{\text{train}*}$) bzw. 1 ($\mathcal{D}^{\text{train}*/7}$).

großen Mehrwert erhält, als wenn es bei gleichbleibend vielen Lernschritten mit nur dem ersten Bild jeder Sequenz trainiert wird.

Die Ergebnisse der Untersuchung sind in Abbildung 5.3 gezeigt. Zu sehen sind die Loss-Kurven des Trainings mit einer Sequenzlänge von 1 ($\mathcal{D}^{\text{train}*/7}$) und des Trainings mit einer Sequenzlänge von 7 ($\mathcal{D}^{\text{train}*}$), welche durch $\mathcal{D}^{\text{val}*}$ evaluiert wurden. In Abbildung 5.3a sehen wir das Lernen der beiden Netze über die Epochen, wobei beide progressiv lernen, also der Loss abnimmt. In Abbildung 5.3b sehen wir dieselben beiden Netze, aber hier der Verlauf ihres Losses über dieselbe Anzahl an Lernschritten. Es ist zu beobachten, dass sich das Training der beiden sehr ähnlich verhält und am Ende die Netze einen sehr ähnlichen Loss erzielen.

Im Weiteren wird erforscht, wie sich das Lernen mit $\mathcal{D}^{\text{train}*}$ und mit $\mathcal{D}^{\text{train}*/7}$ in Bezug auf

Trainingsdaten	Validierungsdaten	PSNR (dB)	Bitrate [bpp]
$\mathcal{D}^{\text{train}*}$	$\mathcal{D}^{\text{val}*}$	35,20	0,2023
$\mathcal{D}^{\text{train}*/7}$	$\mathcal{D}^{\text{val}*}$	35,38	0,2019
$\mathcal{D}^{\text{train}*/7}$	$\mathcal{D}^{\text{val}*/7}$	35,28	0,2029

Tabelle 5.2.: Performanz der Modelle aus den Abbildungen 5.3a und 5.3b auf $\mathcal{D}^{\text{test}/7}$. Hier wurde in allen Fällen das beste Modell auf den Validierungsdaten ausgewählt.

Trainingsdaten	Validierungsdaten	PSNR (dB)	Bitrate [bpp]
$\mathcal{D}^{\text{train}*}$	$\mathcal{D}^{\text{val}*}$	31,54	0,3265
$\mathcal{D}^{\text{train}*/7}$	$\mathcal{D}^{\text{val}*}$	31,58	0,3240
$\mathcal{D}^{\text{train}*/7}$	$\mathcal{D}^{\text{val}*/7}$	31,54	0,3280

Tabelle 5.3.: Performanz der Modelle aus Abbildung 5.3a und 5.3b auf $\mathcal{D}^{\text{Kodak}}$. Hier wurde in allen Fällen das beste Modell auf den Validierungsdaten ausgewählt.

die resultierende Bitrate [bpp] und die Verzerrung PSNR (dB) auswirkt. Auch soll untersucht werden, ob ebenfalls eine Reduktion der Sequenzlänge im Validierungsdatensatz $\mathcal{D}^{\text{val}*}$ auf $\mathcal{D}^{\text{val}*/7}$ ratsam ist, was ein schnelleres Training zur Folge hätte.

In Tabelle 5.2 ist nun das Training mit $\mathcal{D}^{\text{train}*}$ (und $\mathcal{D}^{\text{val}*}$) sowie das Training mit $\mathcal{D}^{\text{train}*/7}$ (und $\mathcal{D}^{\text{val}*}$) und auch das Training mit $\mathcal{D}^{\text{train}*/7}$ (und $\mathcal{D}^{\text{val}*/7}$) dargestellt. Getestet wird hier mit dem $\mathcal{D}^{\text{test}*}$-Split. Es ist gut zu sehen, dass es keinen Nachteil mit sich bringt, bei gleicher Anzahl an Lernschritten von $\mathcal{D}^{\text{train}*}$ auf $\mathcal{D}^{\text{train}*/7}$ zu wechseln. Im Gegenteil, durch die gleiche Anzahl an Lernschritten werden beim Lernen auf $\mathcal{D}^{\text{train}*/7}$ sieben mal so viele Epochen durchlaufen, verglichen mit $\mathcal{D}^{\text{train}*}$. Dies resultiert auch in sieben mal so vielen Modellen zur Auswahl auf $\mathcal{D}^{\text{val}*}$ (eins nach jeder Epoche). Durch die größere Anzahl an Modellen können wir uns am Ende das Beste aus diesen heraussuchen, welches unter Umständen bessere Resultate liefert, als das beste Modell beim Trainieren auf $\mathcal{D}^{\text{train}*}$. Die Reduktion des Validierungs-Datensatzes $\mathcal{D}^{\text{val}*}$ auf $\mathcal{D}^{\text{val}*/7}$ führt in unserem Experiment erwartbar zu leicht schlechteren Werten. Dies nehmen wir in unserer Arbeit in Kauf, da der Performazverlust mit Blick auf die Zeitersparnis beim Training nur minimal und vertretbar ausfällt. Auch bei einem Test auf $\mathcal{D}^{\text{Kodak}}$, wie er in Tabelle 5.3 dargestellt ist, sind dieselben Effekte zu bemerken.

Auf Grundlage unserer neuen Erkenntnisse entscheiden wir uns dafür, in dieser Arbeit im Training $\mathcal{D}^{\text{train}*/7}$ und in der Validierung im Training $\mathcal{D}^{\text{val}*/7}$ zu verwenden.

Baseline-Modell ...	PSNR (dB)	Bitrate [bpp]	Größe [MB]
... unkomprimiert	31,54	0,3265	50,61
... komprimiert	-	-	16,19
... nach Dekomprimierung	31,54	0,3265	50,61

Tabelle 5.4.: **Performanz und Größe** der auf $\mathcal{D}^{\text{train}*}$ selbst trainierten Baseline mit Qualitätspunkt 0, evaluiert auf $\mathcal{D}^{\text{Kodak}}$. Die Kompression erfolgte verlustbehaftet mittels NNCodec [31] mit Standardeinstellungen und folgender Zusatzeinstellung: qp=-100.

5.2.2. Neuronaler Netz-Codec

In Tabelle 5.4 sind die mittlere Bitrate und das mittlere PSNR des auf $\mathcal{D}^{\text{train}*}$ selbst trainieren Baseline-Modells bei Evaluation auf $\mathcal{D}^{\text{Kodak}}$ dargestellt. Die Tabelle ist das Resultat unserer ersten Kompression. Bei dieser Kompression wurde der Quantisierungsparameter qp auf -100 gesetzt. Zu erkennen ist eine Komprimierung des Netzes von ursprünglichen 50,61 MB auf 16,19 MB. Im Weiteren sieht man auch, dass sich die Werte für PSNR und Bitrate nach der Dekompression im Vergleich zum unkomprimierten Netz nicht ändern. Folglich haben wir bei dieser Parameterwahl eine Kompression, bei der wir keinen ersichtlichen Qualitätsverlust beobachten.

Um nachfolgende Netzkompressionen durchzuführen, muss ein wichtiger Parameter der Implementierung des NNCodecs nach Becking et al. [31] eingestellt werden. Bei diesem Parameter handelt es sich um den in der Kompression einstellbaren Quantisierungsparameter qp, welcher in Abschnitt 2.7 schon behandelt wurde. Da bei der Grundeinstellung von qp= -38 nach der Dekompression keine funktionierenden Netze resultieren, haben wir nach anderen Werten für qp gesucht und haben die Werte -40, -60, -80 und -100 für sinnvoll identifiziert. Zur Untersuchung werden für sechs unterschiedliche Qualitätspunkte (1-6) Baseline-Netze, welche auf $\mathcal{D}^{\text{train}*/7}$ trainiert wurden, mit jedem Wert für qp komprimiert und danach wieder dekomprimiert. Das auf $\mathcal{D}^{\text{train}*/7}$ trainierte und noch nicht komprimierte Netz wird im Folgenden "D/7" genannt. Ein auf $\mathcal{D}^{\text{train}*/70}$ trainiertes noch nicht komprimiertes Netz wird "D/70" genannt.

In Abbildung 5.4 sind die Ergebnisse dieser Untersuchung gezeigt. Zu sehen sind die Kurven, welche ihre sechs Qualitätspunkte miteinander verbinden und in der Legende die zu der Kompression zugehörige Einstellung des Parameters qp. Zu sehen ist, dass bei einer Kompression mit qp= -40 mit Verlusten zu rechnen ist (orange Kurve), bei allen anderen hier gezeigten Werten jedoch kaum Verluste zu erkennen sind.

Um dies weiter zu untersuchen, berechnen wir die BD-Raten der Kurven in Bezug zum Netz D/70, da wir bei diesem, anders als bei den Netzen in Abbildung 5.4, ein Kreuzen der Kurven ausschließen können. Diese Ergebnisse und die letztendliche Nachrichtengröße im Vergleich zur Nachrichtengröße des unkomprimierten Netzes sind in Tabelle 5.5 gezeigt. Zu sehen ist, dass auch hier ein qp-Wert von -40 zwar die stärkste Kompres-

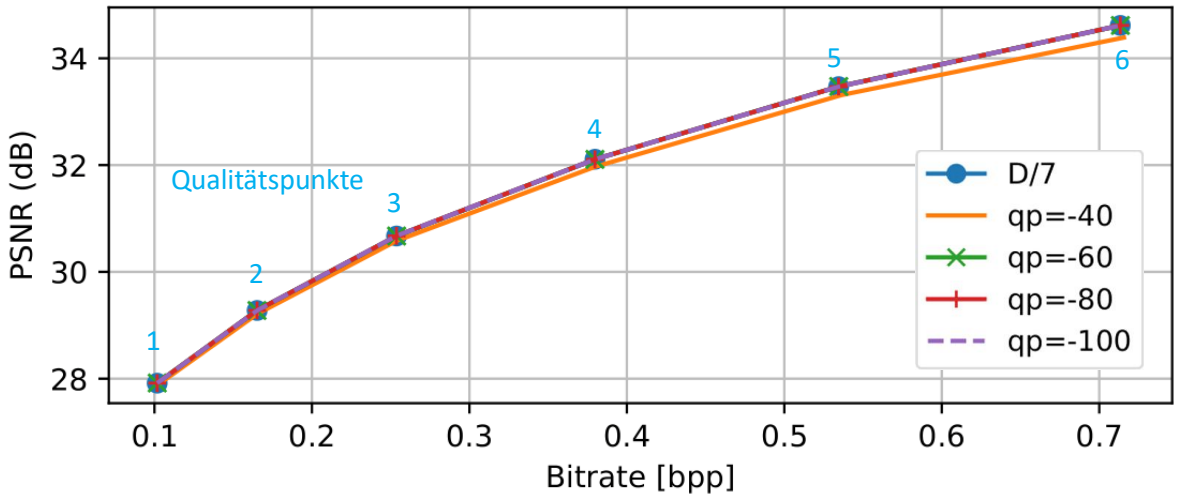


Abbildung 5.4.: Komprimierung und Dekomprimierung der Baseline-Netze mit den Qualitätspunkten 1 bis 6, bei unterschiedlichen Werten des Quantisierungsparameters qp. Zum Vergleich das Baseline-Netz D/7, welches das Ausgangsnetz für die Kompressionen ist. D/7 wurde dabei auf $\mathcal{D}^{\text{train}*/7}$ trainiert, mit $\mathcal{D}^{\text{val}*/7}$ im Training validiert und mit $\mathcal{D}^{\text{Kodak}}$ getestet.

qp	BD-Rate	Kompression
-40	-48,9587%	**17,49%**
-60	**-50,2465%**	31,09%
-80	-50,2335%	46,91%
-100	-50,2193%	61,64%

Tabelle 5.5.: **BD-Rate und Kompression** für die Netze aus Abbildung 5.4, welche mit $\mathcal{D}^{\text{train}*/7}$ trainiert, mit $\mathcal{D}^{\text{val}*/7}$ im Training validiert und mit $\mathcal{D}^{\text{Kodak}}$ getestet wurden. Die Kompression ist dabei der Anteil von der Nachricht des komprimierten Netzes verglichen mit der Nachricht des unkomprimiertem Netzes. Die BD-Rate wurde im Vergleich zum Netz D/70 berechnet, um ein Kreuzen der Kurven auszuschließen.

sion aufweist, bei der Qualität (BD-Rate) jedoch den anderen unterlegen ist. Für die anderen Werte von qp sind die BD-Raten sehr ähnlich und deuten auf ein Netz, welches ähnlich performant ist wie das ursprüngliche unkomprimierte Netz, welches eine BD-Rate von -50,2060% besitzt. Da die BD-Raten für qp$= -60$, qp$= -80$ und qp$= -100$ so nah beieinander liegen, entscheiden wir uns unter diesen Werten für denjenigen, bei dem sich die beste Kompression ergibt und gleichzeitig in diesem Experiment die beste BD-Rate erzielt wird. In diesem Fall ist dies qp$= -60$ bei einer Kompression auf durchschnittlich 31,09%. Von nun an wird, wenn ein neuronales Netz komprimiert wird und es nicht anders beschrieben ist, mit dem Quantisierungsparameter qp$= -60$ komprimiert.

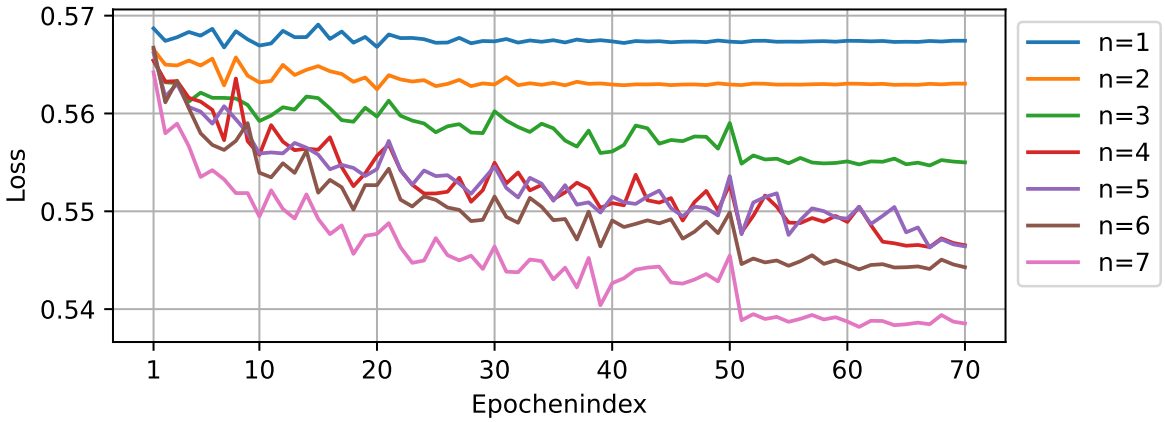


Abbildung 5.5.: **Loss**-Kurven auf $\mathcal{D}^{\text{val}*/7}$ beim Fine-Tuning des Baseline-Modells D/70 mit $\mathcal{D}^{\text{train}*/7}$, wobei nur die letzten n Schichten angepasst werden. Zu Details siehe Tabelle 5.6.

5.3. Fine-Tuning bereits bestehender Schichten

In diesem Abschnitt wird untersucht, welchen Effekt das Fine-Tuning weniger ausgewählter und bereits bestehender Schichten im Netz hat. Damit wird untersucht, wie gut sich diese Schichten oder auch ganzen Blöcke für ein Update eignen. Da in diesem Fall nur wenige Parameter gelernt werden, wird ein Update dieser Schichten bitrateneffizient sein. Dazu wird im Abschnitt 5.3.1 die Performanzsteigerung beim Nachtrainieren der letzten n Schichten im Bild-Decoder untersucht. Anschließend wird in Abschnitt 5.3.2 selbiges für das Nachtrainieren der Blöcke im Hyperprior untersucht. Zum Schluss wird in Abschnitt 5.3.3 untersucht, mit welcher Steigerung der Performanz zu rechnen ist, wenn die Residual-Blöcke nachtrainiert werden.

5.3.1. Fine-Tuning der letzten n Schichten

Untersucht wird die Performanzsteigerung des Baseline-Netzes D/70, wenn es mit vollem Trainingsdatensatz ($\mathcal{D}^{\text{train}*/7}$) in den letzten n Schichten des Bild-Decoders nachtrainiert wird. Dazu werden im Netz D/70 alle Schichten außer die gewünschten letzten n Schichten des Bild-Decoders während des Fine-Tunings eingefroren. Wir wählen dabei $n \in \{1, 2, 3, 4, 5, 6, 7\}$; um das Fine-Tuning der letzten bis zur siebtletzten Schicht zu realisieren.

In Abbildung 5.5 sehen wir den Loss im Fine-Tuning der letzten n Schichten nach dem beschriebenen Prinzip und beim Qualitätspunkt 0. Es ist zu erkennen, dass das Netz für $n = 1$ und $n = 2$ seinen Loss nur geringfügig verringert, also nur sehr schwach lernt. Bei allen anderen Werten für n ist ein besseres Minimieren des Losses über die Epochen zu sehen. Am stärksten minimiert das Netz seinen Loss, welches die letzten sieben Schichten

	Netz	Trainingsdaten	PSNR (dB)	Bitrate [bpp]	Anzahl Parameter
1	D/70	$\mathcal{D}^{\text{train}*/70}$	29,30	0,2844	-
2	D/70 || FT($n=1$)	$\mathcal{D}^{\text{train}*/70}$ || $\mathcal{D}^{\text{train}*/7}$	29,30	0,2844	**13.836**
3	D/70 || FT($n=2$)	$\mathcal{D}^{\text{train}*/70}$ || $\mathcal{D}^{\text{train}*/7}$	29,34	0,2844	161.420
4	D/70 || FT($n=3$)	$\mathcal{D}^{\text{train}*/70}$ || $\mathcal{D}^{\text{train}*/7}$	29,39	0,2844	309.004
5	D/70 || FT($n=4$)	$\mathcal{D}^{\text{train}*/70}$ || $\mathcal{D}^{\text{train}*/7}$	29,46	0,2844	899.340
6	D/70 || FT($n=5$)	$\mathcal{D}^{\text{train}*/70}$ || $\mathcal{D}^{\text{train}*/7}$	29,45	0,2844	915.852
7	D/70 || FT($n=6$)	$\mathcal{D}^{\text{train}*/70}$ || $\mathcal{D}^{\text{train}*/7}$	29,48	0,2844	1.063.436
8	D/70 || FT($n=7$)	$\mathcal{D}^{\text{train}*/70}$ || $\mathcal{D}^{\text{train}*/7}$	**29,53**	0,2844	1.653.772

Tabelle 5.6.: Performanz auf $\mathcal{D}^{\text{Kodak}}$ bei **Fine-Tuning der letzten n Schichten**: Metriken PSNR, Bitrate und Anzahl der Parameter, welche bei einem Netzupdate übertragen werden müssen. Der Validierungsdatensatz ist stets $\mathcal{D}^{\text{val}*/7}$ und trainiert wurde der Qualitätspunkt 0. Die Notation Netzteil1 || Netzteil2 bedeutet, dass erst Netzteil1 trainiert wird, dann fixiert und um Netzteil2 ergänzt, und schließlich in einem zweiten Trainingsschritt Netzteil2 ein Update erfährt. FT(n) bedeutet, dass ein Fine-Tuning von nur den letzten n Schichten im Netz durchgeführt wird.

nachtrainiert.

In Tabelle 5.6 sind die trainierten Netze aufgelistet und evaluiert. Wir sehen, dass ein Fine-Tuning mit $n = 1$ weder eine ersichtliche Verbesserung der Bitrate noch des PSNRs mit sich bringt, wo hingegen bei allen anderen Werten für n eine Steigerung im PSNR zu vermerken ist. Der Grund, weshalb sich nur das PSNR ändert, nicht aber die Bitrate, ist der, dass wir nur den Bild-Decoder trainieren. Dieser bekommt die Nachricht, die der Bild-Encoder codiert hat. Da das Trainieren der letzten n Schichten keinen Einfluss auf den Bild-Encoder hat, bleibt dieser unverändert und codiert mit der selben Bitrate wie vor dem Fine-Tuning. Da aber die Nachricht des Bild-Encoders durch das Fine-Tuning vom Bild-Decoder nun besser decodiert werden kann, beobachten wir einen Anstieg im PSNR. Wir sehen, dass wir unseren besten Wert für das PSNR beim Fine-Tuning der letzten 7 Schichten erreichen, was darauf zurückführbar ist, dass in diesem Fine-Tuning mit 1.653.772 Parametern die meisten Parameter dieser Untersuchung trainiert wurden und für ein Netz-Update zu übertragen wären.

Wir entscheiden uns repräsentativ dafür, die Methode des Fine-Tunings der letzten 7 Schichten genauer zu untersuchen. Wir untersuchen das Fine-Tuning des Netzes D/70 mit den Qualitätspunkten von 1 bis 6 mit dem vollen Trainingsdatensatz $\mathcal{D}^{\text{train}*/7}$. Dabei wollen wir den Versatz dieser Netze zum Baseline-Netz D/70 ermitteln und zwischen diesen die BD-Rate ermitteln, um diese Methode mit anderen in dieser Arbeit behandelten Methoden vergleichen zu können. Wir wählen sechs Qualitätspunkte, da wir somit bei der Berechnung der BD-Rate weniger anfällig für Ausreißer sind.

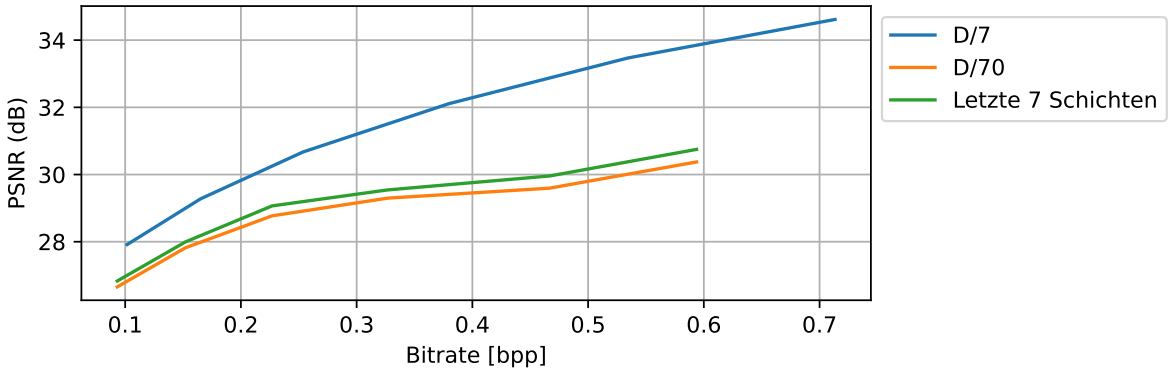


Abbildung 5.6.: Performanz auf $\mathcal{D}^{\text{Kodak}}$: Kurven über die Qualitätspunkte des schwachen Baseline-Netzes D/70, des D/70-Netzes nach dem Fine-Tuning der letzten 7 Schichten mit dem Trainingsdatensatz $\mathcal{D}^{\text{train}*/7}$, und des starken Baseline-Netzes D/7.

Die Ergebnisse dieser Untersuchung sehen wir in Abbildung 5.6. Wie bereits beschrieben, ist auch hier erkennbar, dass es keinen Versatz bei der Bitrate gibt, jedoch einen im PSNR. Dieser fällt im Vergleich zum Fine-Tuning des gesamten Netzes (D/7) jedoch sehr gering aus. Die ermittelte BD-Rate unserer Methode in Bezug auf das Netz vor dem Fine-Tuning (D/70) ergibt sich zu -12,6011%, was bedeutet, dass bei demselben PSNR das Netz nach dem Fine-Tuning eine 12,6011% geringere Bitrate benötigt, als vor dem Fine-Tuning (D/70).

5.3.2. Fine-Tuning im Hyperprior

Untersucht wird, wie sich die Performanz unseres Baseline-Netzes D/70 ändert, wenn es mit vollem Trainingsdatensatz ($\mathcal{D}^{\text{train}*/7}$) in unterschiedlichen Blöcken des Hyperpriors nachtrainiert wird. Dazu werden alle, außer die gerade untersuchten Schichten, im Netz eingefroren, so dass nur diese im Fine-Tuning lernen.

In Tabelle 5.7 sind die Blöcke, die wir untersuchen wollen, nach dem Fine-Tuning mit Qualitätspunkt 0 gezeigt. Wir sehen, dass wir bei allen Blöcken, welche wir untersuchen wollen, im PSNR Einbußen haben, sich die Bitrate aber bei jedem Fine-Tuning-Netz im Vergleich zur Baseline (D/70) verringert. Wir sehen auch, das sich trotz der 819.456 Parameter im Kontext-Modell die Bitrate und das PSNR nach dem Fine-Tuning am wenigsten ändern. Beim Entropie-Parameter ändern sich nach dem Fine-Tuning die Bitrate und das PSNR am stärksten, obwohl er mit 451.697 Parametern die wenigsten besitzt. Er besitzt auch die niedrigste Bitrate, dafür aber auch unter allen Fine-Tuning-Netzen das niedrigste PSNR, was ihn somit nicht unbedingt zur besten Wahl für ein Netz-Update macht.

Um nun unsere vier Fine-Tuning-Methoden besser vergleichen zu können, nutzen wir die

	Netz	Trainingsdaten	PSNR (dB)	Bitrate [bpp]	Anzahl Parameter
1	D/70	$\mathcal{D}^{\text{train}*/70}$	**29,30**	0,2844	-
2	D/70 || FT(Entropie-Parameter)	$\mathcal{D}^{\text{train}*/70}$ || $\mathcal{D}^{\text{train}*/7}$	29,16	**0,2769**	**451.697**
3	D/70 || FT(Hyper-Encoder)	$\mathcal{D}^{\text{train}*/70}$ || $\mathcal{D}^{\text{train}*/7}$	29,22	0,2829	737.920
4	D/70 || FT(Hyper-Decoder)	$\mathcal{D}^{\text{train}*/70}$ || $\mathcal{D}^{\text{train}*/7}$	29,19	0,2826	2.729.792
5	D/70 || FT(Kontext-Modell)	$\mathcal{D}^{\text{train}*/70}$ || $\mathcal{D}^{\text{train}*/7}$	29,24	0,2832	819.456

Tabelle 5.7.: Performanz auf $\mathcal{D}^{\text{Kodak}}$ bei **Fine-Tuning der Residual-Blöcke**: Metriken PSNR, Bitrate und Anzahl der Parameter, welche bei einem Netzupdate übertragen werden müssen. Der Validierungsdatensatz ist stets $\mathcal{D}^{\text{val}*/7}$ und trainiert wurde der Qualitätspunkt 0. Die Notation Netzteil1 || Netzteil2 bedeutet, dass erst Netzteil1 trainiert wird, dann fixiert und um Netzteil2 ergänzt, und schließlich in einem zweiten Trainingsschritt Netzteil2 ein Update erfährt. FT() bedeutet ein Fine-Tuning derjenigen Blöcke/Schichten, welche in den Klammern stehen.

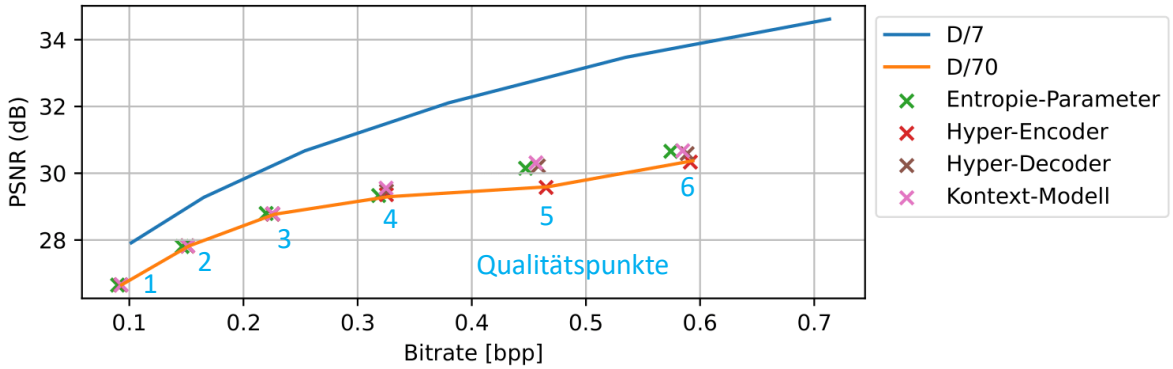


Abbildung 5.7.: Performanz auf $\mathcal{D}^{\text{Kodak}}$: Kurven über die Qualitätspunkte des schwachen Baseline-Netzes D/70, des D/70-Netzes nach dem Fine-Tuning der in der Legende stehenden Blöcke des Hyperpriors mit dem Trainingsdatensatz $\mathcal{D}^{\text{train}*/7}$, und des starken Baseline-Netzes D/7.

Qualitätspunkte 1 bis 6, da wir mit der höheren Anzahl an Qualitätspunkten weniger anfällig für Ausreißer sind und später eine genauere BD-Rate berechnen können.

In Abbildung 5.7 sind die von uns ermittelten Qualitätspunkte der vier Fine-Tuning-Methoden dargestellt. Es ist zu sehen, dass an den meisten Qualitätspunkten kaum eine Verbesserung der Performaz stattfindet, bis auf die Qualitätspunkte 5 und 6. An diesen kann man für das Fine-Tuning vom Entropie-Parameter, dem Hyper-Decoder und dem Kontext-Modell eine gute Verbesserung der Performanz feststellen. Dabei liegt das Fine-Tuning vom Kontext-Modell mit einer BD-Rate von -10,0277% in Bezug zum Netz D/70 vorne, nach ihm kommt das vom Entropie-Parameter mit einer BD-Rate von -9,4889%. Danach kommt das Fine-Tuning vom Hyper-Decoder mit einer BD-Rate von -8,8766%

	Netz	Trainingsdaten	PSNR (dB)	Bitrate [bpp]	Anzahl Parameter
1	D/70	$\mathcal{D}^{\text{train}*/70}$	29,30	**0,2844**	-
2	D/70 || FT(Res.-Blöcke)	$\mathcal{D}^{\text{train}*/70}$ || $\mathcal{D}^{\text{train}*/7}$	29,86	0,2967	**2.066.176**
3	D/70 || FT(Alle Blöcke)	$\mathcal{D}^{\text{train}*/70}$ || $\mathcal{D}^{\text{train}*/7}$	**29,88**	0,3012	6.925.056

Tabelle 5.8.: Performanz auf $\mathcal{D}^{\text{Kodak}}$ bei **Fine-Tuning der Resisual-Blöcke**: Metriken PSNR, Bitrate und Anzahl der Parameter, welche bei einem Netzupdate übertragen werden müssen. Der Validierungsdatensatz ist stets $\mathcal{D}^{\text{val}*/7}$ und trainiert wurde der Qualitätspunkt 0. Die Notation Netzteil1 || Netzteil2 bedeutet, dass erst Netzteil1 trainiert wird, dann fixiert und um Netzteil2 ergänzt, und schließlich in einem zweiten Trainingsschritt Netzteil2 ein Update erfährt. FT(Res.-Blöcke) bedeutet, dass jeder ResBlock (Abbildung 4.2a) im Fine-Tuning lernt. FT(Alle Blöcke) bedeutet, dass jeder ResBlock (Abbildung 4.2a), ResBlockStride (Abbildung 4.2b) und ResBlockUp (Abbildung 4.2c) im Fine-Tuning lernt.

und zuletzt das vom Hyper-Encoder, welcher mit Abstand die geringste BD-Rate von -0,4201% hat.

5.3.3. Fine-Tuning der Residual-Blöcke

Untersucht wird, wie sich die Performanz unseres Baseline-Netzes D/70 ändert, wenn es mit vollem Trainingsdatensatz ($\mathcal{D}^{\text{train}*/7}$) in den Residual-Blöcken nachtrainiert wird. Dazu werden alle Schichten, außer die in den Residual-Blöcken, im Netz eingefroren, so dass nur diese im Fine-Tuning lernen. Die Residual-Blöcke sind interessant, da wir, wie in Abschnitt 4.3.1 erklärt, unsere Adaptoren später dort einfügen wollen. Interessant ist daher auch, die Residual-Blöcke ohne den Einsatz von Adaptoren zu verbessern, indem sie vollständig im Fine-Tuning lernen.

In Tabelle 5.8 sind die beiden Methoden im Fine-Tuning gezeigt, einmal werden nur alle ResBlock-Blöcke (Abbildung 4.2a) trainiert, das wird im Folgenden "Res.-Blöcke" genannt und einmal werden alle Blöcke ResBlock (Abbildung 4.2a), ResBlockStride (Abbildung 4.2b) und ResBlockUp (Abbildung 4.2c) trainiert, das wird im Folgenden mit "Alle Blöcke" betitelt. Wir sehen in der Tabelle, welche sich für den Qualitätspunkt 0 ergeben hat, dass bei beiden Methoden keine Verringerung der Bitrate stattfindet, sondern beide diese erhöhen. Zu sehen ist aber auch, dass das PSNR bei beiden Methoden steigt, jedoch die beiden Methoden im Fine-Tuning untereinander nur einen sehr geringen Unterschied beim PSNR haben, obwohl das Fine-Tuning von "Res.-Blöcke" 2.066.176 Parameter lernt und das von "Alle Blöcke" sogar 6.925.056 Parameter lernt. Für diesen großen Unterschied in der Parameter-Anzahl scheint der Unterschied im PSNR sehr gering zu sein. Es fällt ebenfalls auf, dass die Bitrate trotz der vergleichsweise vielen Parameter vom Fine-Tuning von "Alle Blöcke" höher ist, als die von "Res.-Blöcke".

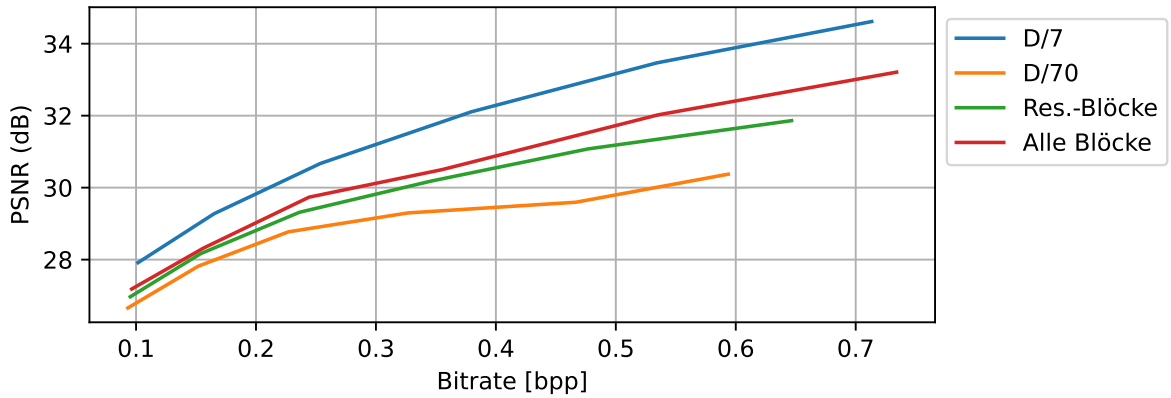


Abbildung 5.8.: Performanz auf $\mathcal{D}^{\text{Kodak}}$: Kurven über die Qualitätspunkte des schwachen Baseline-Netzes D/70, des D/70-Netzes nach dem Fine-Tuning aller Aufkommen von ResBlock mit dem Trainingsdatensatz $\mathcal{D}^{\text{train}*/7}$ ("Res.-Blöcke"), des D/70-Netzes nach dem Fine-Tuning aller Aufkommen von ResBlock, ResBlockStride und ResBlockUp mit dem Trainingsdatensatz $\mathcal{D}^{\text{train}*/7}$ ("alle Blöcke"), und des starken Baseline-Netzes D/7.

Um nun unsere beiden Fine-Tuning-Methoden besser vergleichen zu können, nutzen wir die Qualitätspunkte 1 bis 6, da wir mit der höheren Anzahl an Qualitätspunkten weniger anfällig für Ausreißer sind und später eine genauere BD-Rate berechnen können.

In Abbildung 5.8 sehen wir zum einen die Kurven über die jeweils sechs Qualitätspunkte unserer beiden Fine-Tuning-Methoden verglichen mit denen unserer Netze D/70 und D/7. Zu erkennen ist, dass das Fine-Tuning von "Alle Blöcke" besser ist, als das Fine-Tuning von "Res.-Blöcke". Dennoch besitzt die Methode "Alle Blöcke" mehr als drei mal so viele Parameter wie "Res.-Blöcke". Dies wirft die Frage auf, ob eine vergleichbar kleine Qualitätssteigerung von "Res.-Blöcke" zu "Alle Blöcke" mit der vielfachen Anzahl an Parametern zu rechtfertigen ist. Die BD-Rate des Fine-Tunings von "Res.-Blöcke" beträgt -23,9474% und die von "Alle Blöcke" beträgt -30,7742% in Bezug auf D/70.

5.4. Fine-Tuning mit Adaptoren

In diesem Abschnitt wird der bereits in Abschnitt 4.3 eingeführte Adaptor an die ebenfalls beschriebenen Stellen des Baseline-Netzes D/70 zum Zeitpunkt des Fine-Tunings eingesetzt und trainiert. Dabei unterscheiden wir zwei Methoden zum Einsatz der Adaptoren. Die erste, welche in Abschnitt 5.4.1 untersucht wird, nutzt nur den ResBlockAdaptor (Abbildung 4.3a). Bekommt ein Baseline-Netz diesen Adaptor eingesetzt, wird es im Folgenden Adaptornetz 1 genannt. Die zweite Methode, welche in Abschnitt 5.4.2 untersucht wird, nutzt ResBlockAdaptor (Abbildung 4.3a), ResBlockStrideAdaptor (Abbildung 4.3b) und ResBlockUpAdaptor (Abbildung 4.3c). Baseline-Netze, welche diese

	Netz (+Adaptor)	Trainingsdaten	PSNR (dB)	Bitrate [bpp]	Anzahl Parameter
1	D/70	$\mathcal{D}^{\text{train}*/70}$	29,19	**0,2844**	11.833.149
2	D/70 +Adap	$\mathcal{D}^{\text{train}*/70}$	29,57	0,2952	12.207.677
3	D/7	$\mathcal{D}^{\text{train}*/7}$	31,54	0,3280	11.833.149
4	D/7 +Adap	$\mathcal{D}^{\text{train}*/7}$	**31,63**	0,3279	12.207.677
5	D/70 || +Adap	$\mathcal{D}^{\text{train}*/70}$ || $\mathcal{D}^{\text{train}*/7}$	29,52	0,2917	**374.528**
6	D/70 +Adap || Adap	$\mathcal{D}^{\text{train}*/70}$ || $\mathcal{D}^{\text{train}*/7}$	29,80	0,2964	**374.528**

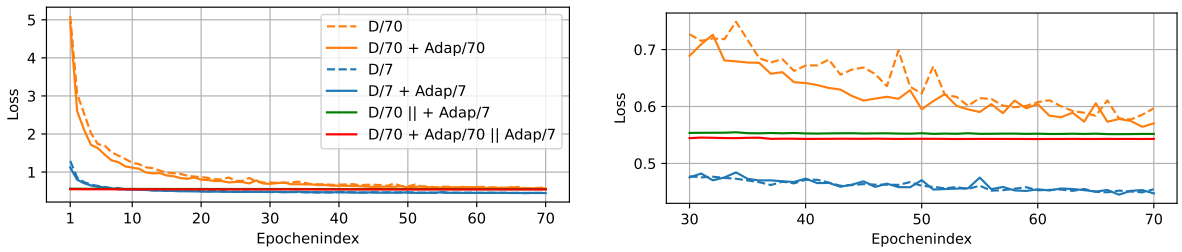
Tabelle 5.9.: Performanz auf $\mathcal{D}^{\text{Kodak}}$: Metriken PSNR, Bitrate und Anzahl der Parameter, welche bei einem Netzupdate übertragen werden müssen. Der Validierungsdatensatz ist stets $\mathcal{D}^{\text{val}*/7}$ und trainiert wurde der Qualitätspunkt 0. Die Notation Netzteil1 || Netzteil2 bedeutet, dass erst Netzteil1 trainiert wird, dann fixiert und um Netzteil2 ergänzt, und schließlich in einem zweiten Trainingsschritt Netzteil2 ein Update erfährt, in diesem Fall werden nur die Parameter von Netzteil2 gezählt. Verwendet wurde aus Abbildung 4.3a der ResBlockAdaptor($3 \times 3, 2, N$).

Adaptoren eingesetzt bekommen, werden im Folgenden auch Adaptornetz 2 genannt.

5.4.1. Adaptornetz 1

Im Folgenden wollen wir untersuchen, welchen Effekt das Fine-Tuning von nachträglich eingefügten Adaptoren auf die Performanz des Baseline-Netzes hat. Dazu wird hier als der Adaptor (Abbildung 4.4) wie in Abbildung 4.3a gezeigt in die bereits bestehenden ResBlock-Blöcke des Bild-Encoders und -Decoders eingefügt.

In Tabelle 5.9 sind verschiedene Netze gezeigt, welche mit unterschiedlichen Trainingsdaten trainiert wurden, alle sind dabei Netze mit Qualitätspunkt 0. In Zeile 1 wird das Baseline-Netz mit $\mathcal{D}^{\text{train}*/70}$ trainiert und entspricht damit dem Netz D/70. In Zeile 2 wurde das Baseline-Netz um die Adaptoren ergänzt und dann mit $\mathcal{D}^{\text{train}*/70}$ trainiert. Ähnlich sind auch die Zeilen 3 und 4, nur dass sich hier der Trainingsdatensatz ändert, hier wird nun $\mathcal{D}^{\text{train}*/7}$ genutzt. Das Netz in Zeile 3 entspricht dabei dem Netz D/7. Besonders wichtig sind die Zeilen 5 und 6, da diese ein Fine-Tuning abdecken. In Zeile 5 wird das Baseline-Netz D/70 aus Zeile 1 um die Adaptoren ergänzt und anschließend werden *nur* die Adaptoren mit $\mathcal{D}^{\text{train}*/7}$ im Fine-Tuning weiter trainiert. Ähnliches passiert in Zeile 6: Das Netz D/7 aus Zeile 2, welches bereits Adaptoren besitzt, erfährt auch ein Fine-Tuning, bei dem ebenfals nur die Adaptoren lernen. Wir sehen, dass das Netz D/70 (Zeile 1) von allen die niedrigste Bitrate besitzt und dass durch das Training von Netz und Adaptoren mit vollem Datensatz das Netz in Zeile 4 das beste PSNR zeigt. Jedoch trainieren die Netze in Zeilen 1 und 4 alle Schichten im Netz, wo hingegen die Netze in Zeilen 5 und 6 nur die Adaptoren trainieren und somit die wenigsten Parameter



(a) Gesamtes Training über 70 Epochen.

(b) Fokus auf die Epochen 30 bis 70.

Abbildung 5.9.: **Loss**-Kurven auf $\mathcal{D}^{\text{val}*/7}$ beim Training der Modelle aus Tabelle 5.9. Bei Netzen, bei denen erst die Baseline und dann der Adaptor trainiert wurde, wird nur die Loss-Kurve des Adaptortrainings abgebildet, so dass alle Netze nach Epoche 70 mit dem Training fertig sind und verglichen werden können.

von allen trainieren.

Die Tabelle beschreibt zwei verschiedene Stories. Die erste ist, dass das Netz D/70 (Zeile 1) die Adaptoren angehängt bekommt, welche im Fine-Tuning trainiert werden (Zeile 5), so dass bei Netz-Updates für andere D/70-Netze nur diese Adaptoren übertragen werden müssen. Die andere Story ist, dass das Netz bereits die Adaptoren besitzt und Adaptoren und Netz gemeinsam auf $\mathcal{D}^{\text{train}*/70}$ trainiert wurden (Zeile 2) und nun im Fine-Tuning diese Adaptoren mit weiteren Daten ($\mathcal{D}^{\text{train}*/7}$) weiter lernen (Zeile 6).

In Abbildung 5.9 ist das Training beziehungsweise das Fine-Tuning der Netze aus Tabelle 5.9 zu sehen. Es ist zu erkennen, dass D/70, was für das Baseline-Netz steht, welches auf $\mathcal{D}^{\text{train}*/70}$ trainiert wurde und D/70 +Adap/70, was für das Baseline-Netz mit eingefügten Adaptoren steht welche gemeinsam auf $\mathcal{D}^{\text{train}*/70}$ trainiert wurden, immer sehr nah beieinander liegen. Ebenso bei D/7 und D/7 +Adap/7, welche gemeinsam auf $\mathcal{D}^{\text{train}*/7}$ trainiert wurden. Bei den beiden Fine-Tuning-Kurven ist kaum ein Lernen zu sehen, dennoch besitzen sie einen niedrigeren Loss, als ihr ursprüngliches Netz vor dem Fine-Tuning (D/70 und D/70 +Adap/70).

Im Folgenden wird nur noch die erste der bereits genannten Stories weiterverfolgt. Wir untersuchen also, wie wir durch das Anhängen und Fine-Tuning von Adaptoren an das Netz D/70 dieses verbessern können und dieselben Adaptoren an andere D/70-Netze als Update übertragen können, um diese auf die gleiche Performanz anzuheben.

Nun untersuchen wir, welchen Effekt die einstellbaren Parameter v und R der Adaptoren auf die Performanz des Netzes haben. Dafür wählen wir die Parameter aus folgenden Mengen: $v \in \{3, 5, 7\}$ und $R \in \{2, 4, 8\}$. Wir erzeugen für jede Kombination aus v und R die resultierenden Adaptoren und fügen sie in unser D/70-Netz ein. Im Anschluss werden (nur!) die Adaptoren im Fine-Tuning mit $\mathcal{D}^{\text{train}*/7}$ trainiert.

Die Loss-Kurven der Netze und ihre Einstellungen für v und R sind in Abbildung 5.10

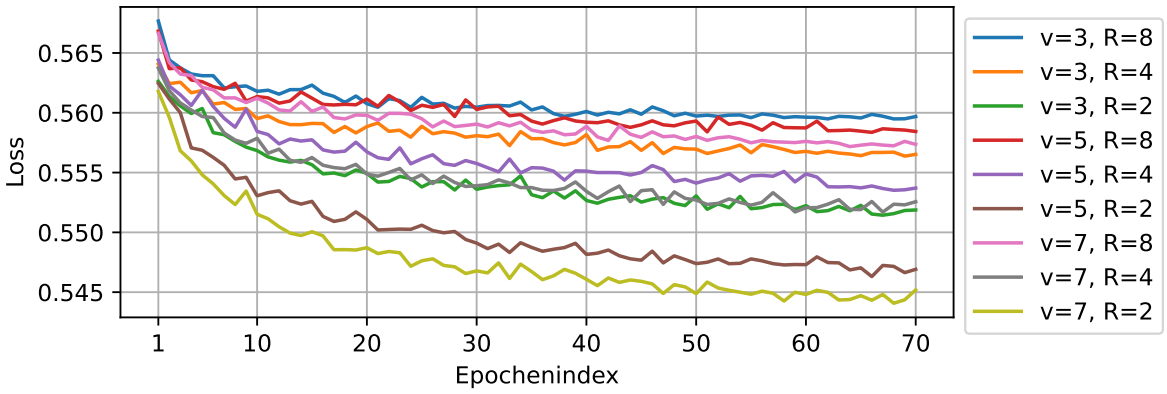


Abbildung 5.10.: **Loss**-Kurven des Adaptornetzes 1 auf $\mathcal{D}^{\text{val}*/7}$ beim Fine-Tuning der Adaptoren eines vorherigen D/70-Netzes, mit unterschiedlichen Einstellungen für v und R.

dargestellt. Dabei ist zu sehen, dass alle Netze im Fine-Tuning lernen und dass ein erhöhtes v zu einem niedrigeren Loss und ein erhöhtes R zu einem erhöhten Loss führen.

In Tabelle 5.10 sind die aus den Einstellungen für v und R resultierenden Parameter der Adaptoren aus dem Adapternetz 1 gezeigt. Zu erkennen ist, dass ein größerer Wert für v in mehr Parametern resultiert und ein größerer Wert für R in weniger Parametern resultiert. Dies ist auch zu erwarten, da v einen Kernel mit $v \times v$ beschreibt und R der Reduktionsparameter ist, durch den die Eingangs-Feature-Map-Anzahl geteilt wird (genauer beschrieben in Abschnitt 4.3.2). Mit einem $v = 3$ und einem $R = 8$ erreichen wir in diesem Fall die niedrigste Anzahl an untersuchten Parametern.

In Tabelle 5.11 sehen wir für die unterschiedlichen Einstellungen von v und R in den Adaptoren des Adaptornetzes 1 die resultierenden Werte für das PSNR nach dem Fine-Tuning. Wir sehen, dass hier eine Erhöhung von v zu einem erhöhten PSNR und eine Erhöhung von R zu einem niedrigeren PSNR führt. In Kombination unserer Ergebnisse aus Tabelle 5.10 lässt sich schließen, dass mehr lernbare Parameter in allen untersuchten Fällen zu einem höheren PSNR führen. Das höchste PSNR erreicht die Einstellung $v = 7$ und $R = 2$ mit 29,67 dB.

In Tabelle 5.12 ist die erreichte Bitrate des Adaptornetzes 1 nach dem Fine-Tuning für die verschiedenen Werte für v und R der Adaptoren gezeigt. Zu erkennen ist, das hier mit einem Erhöhen von v die Bitrate ebenfalls erhöht wird, und bei einem Erhöhen von R die Bitrate abnimmt. In Kombination unserer Ergebnisse aus Tabelle 5.10 lässt sich schließen, dass mehr lernbare Parameter zu einer erhöhten Bitrate führen, und weniger lernbare Parameter die Bitrate reduzieren. Die niedrigste Bitrate bekommen wir für $v = 3$ und $R = 8$ mit durchschnittlichen 0,2869 bpp. Dieser Wert liegt immer noch über dem des D/70-Netzes (0,2844 bpp).

Aus diesen Experimenten resultiert nun, dass v die Parameteranzahl steigen und R die Parameteranzahl sinken lässt. Dabei führen mehr lernbare Parameter in den Adaptoren

	$R = 2$	$R = 4$	$R = 8$
$v = 3$	374.528	123.200	**45.920**
$v = 5$	833.280	237.888	74.592
$v = 7$	1.521.408	409.920	117.600

Tabelle 5.10.: **Anzahl der Adaptorparameter** des Adaptornetzes 1 bei Einsatz des vorgeschlagenen Blocks ResBlockAdaptor($v \times v, R, N$) (Abbildung 4.3a)

	$R = 2$	$R = 4$	$R = 8$
$v = 3$	29,52	29,43	29,38
$v = 5$	29,62	29,49	29,40
$v = 7$	**29,67**	29,55	29,43

Tabelle 5.11.: **PSNR** (dB) des Adaptornetzes 1 auf $\mathcal{D}^{\text{Kodak}}$ bei verschiedenen Werten für v und R unter Einsatz des vorgeschlagenen Blocks ResBlockAdaptor($v \times v, R, N$) (Abbildung 4.3a). Als Modell wurde das zuerst auf $\mathcal{D}^{\text{train}*/70}$ trainierte Baseline-Modell genommen, welches im Anschluss eingefroren, um die Adaptoren ergänzt und auf $\mathcal{D}^{\text{train}*/7}$ weiter trainiert wurde.

	$R = 2$	$R = 4$	$R = 8$
$v = 3$	0,2917	0,2882	**0,2869**
$v = 5$	0,2948	0,2903	0,2878
$v = 7$	0,2965	0,2912	0,2885

Tabelle 5.12.: **Bitrate** [bpp] des Adaptornetzes 1 auf $\mathcal{D}^{\text{Kodak}}$ bei verschiedenen Werten für v und R unter Einsatz des vorgeschlagenen Blocks ResBlockAdaptor($v \times v, R, N$) (Abbildung 4.3a). Als Modell wurde das zuerst auf $\mathcal{D}^{\text{train}*/70}$ trainierte Baseline-Modell genommen, welches im Anschluss eingefroren, um die Adaptoren ergänzt und auf $\mathcal{D}^{\text{train}*/7}$ weiter trainiert wurde.

zu einer Verbesserung des PSNRs aber auch gleichzeitig zu Einbußen in der Bitrate. Mit $v = 7$ und $R = 2$ wird der beste Wert für das PSNR und mit $v = 3$ und $R = 8$ die besten Werte für die Bitrate erreicht. Um einen Tradeoff zwischen den beiden Metriken einzugehen, entscheiden wir uns dafür, das Adaptornetz 1 mit Adaptoren der Parameter $v = 3$ und $R = 2$ auszustatten. Im Folgenden gilt nun, dass Adaptornetz 1, wenn nicht anders beschrieben, immer mit Adaptoren der Einstellung $v = 3$ und $R = 2$ im ResBlock ausgestattet wurde und im Fine-Tuning nur diese Adaptoren gelernt haben.

	Netz (+Adaptor)	Trainingsdaten	PSNR (dB)	Bitrate [bpp]	Anzahl Parameter
1	D/70	$\mathcal{D}^{\text{train}*/70}$	29,19	**0,2844**	11.833.149
2	D/70 +Adap	$\mathcal{D}^{\text{train}*/70}$	29,56	0,2961	12.528.701
3	D/7	$\mathcal{D}^{\text{train}*/7}$	31,54	0,3280	11.833.149
4	D/7 +Adap	$\mathcal{D}^{\text{train}*/7}$	**31,57**	0,3271	12.528.701
5	D/70 || +Adap	$\mathcal{D}^{\text{train}*/70}$ || $\mathcal{D}^{\text{train}*/7}$	29,60	0,2957	**695.552**
6	D/70 +Adap || Adap	$\mathcal{D}^{\text{train}*/70}$ || $\mathcal{D}^{\text{train}*/7}$	29,90	0,2972	**695.552**

Tabelle 5.13.: Performanz auf $\mathcal{D}^{\text{Kodak}}$ bei **Nutzung aller drei Adaptoren nach Abbildung 4.3**: Metriken PSNR, Bitrate und Anzahl der Parameter, welche bei einem Netzupdate übertragen werden müssen. Der Validierungsdatensatz ist stets $\mathcal{D}^{\text{val}*/7}$ und trainiert wurde der Qualitätspunkt 0. Die Notation Netzteil1 || Netzteil2 bedeutet, dass erst Netzteil1 trainiert wird, dann fixiert und um Netzteil2 ergänzt, und schließlich in einem zweiten Trainingsschritt Netzteil2 ein Update erfährt, in diesem Fall werden nur die Parameter von Netzteil2 gezählt. Verwendet wurden die Blöcke ResBlockAdaptor$(3 \times 3, 2, N)$ (Abbildung 4.3a), ResBlockStrideAdaptor$(3 \times 3, 2, N, S)$ (Abbildung 4.3b) und ResBlockUpAdaptor$(3 \times 3, 2, N, U)$ (Abbildung 4.3c).

5.4.2. Adaptornetz 2

Im Folgenden wollen wir untersuchen, welchen Effekt das Fine-Tuning von einer größeren Anzahl nachträglich eingefügter Adaptoren auf die Performanz des Baseline-Netzes hat. Dazu wird hier als der Adaptor (Abbildung 4.4) wie in Abbildung 4.3a, Abbildung 4.3b und Abbildung 4.3c, in die bereits bestehenden Blöcke ResBlock, ResBlockStride und ResBlockUp eingefügt.

In Tabelle 5.13 sind verschiedene Netze (alle mit dem Qualitätspunkt 0) nach ihrem Training beziehungsweise nach ihrem Fine-Tuning gezeigt. In Zeile 1 ist das Baseline-Netz D/70 zu sehen und in Zeile 3 das D/7-Netz. In der Zeile 2 wird das Baseline-Netz um die Adaptoren ergänzt und anschließend mit $\mathcal{D}^{\text{train}*/70}$ und in Zeile 4 mit $\mathcal{D}^{\text{train}*/7}$ trainiert. Das Netz aus Zeile 1 wird in Zeile 5 um Adaptoren ergänzt, welche im Fine-Tuning mit dem vollen Datensatz $\mathcal{D}^{\text{train}*/7}$ trainiert werden. Ähnlich geschiet dies auch mit dem Netz aus Zeile 2. Da dieses jedoch schon Adaptoren enthält, müssen sie nicht mehr hinzugefügt werden. Dennoch werden in Zeile 6 im Fine-Tuning mit dem vollen Datensatz $\mathcal{D}^{\text{train}*/7}$ nur diese Adaptoren trainiert. Da in Zeile 5 und 6 nur die Adaptoren lernen, sehen wir hier auch die niedrigste Anzahl an Parametern (für die Netz-Update-Nachricht), welche im Training/Fine-Tuning lernen. Wir sehen im Allgemeinen sehr ähnliche Resultate wie im Experiment mit dem Adapternetz 1 aus Abschnitt 5.4.1. Auch hier besitzt das D/70-Netz (Zeile 1) die niedrigste Bitrate und das Netz aus Zeile 4, welches die Adaptoren bereits vor dem Lernen eingesetzt bekommt und direkt auf

dem vollen Datensatz $\mathcal{D}^{\text{train}*/7}$ trainiert wird, das beste PSNR.

Im Weiteren verfolgen wir nur noch die Story, in der wir das Netz D/70 (Zeile 1) um Adaptoren ergänzen und nur diese im Fine-Tuning mit dem gesamten Datensatz $\mathcal{D}^{\text{train}*/7}$ trainieren (Zeile 5). Wir können nun die im Fine-Tuning gelernten Adaptoren in jedes andere D/70-Netz, welches gleich trainiert wurde, einsetzen und dieselbe Performanz wie das Netz, aus dem die Adaptoren stammen, erhalten.

Wir wollen nun den Einfluss der Einstellungen für v und R unserer Adaptoren auf die Performanz untersuchen. Dazu wählen wir $v \in \{3, 5, 7\}$ und $R \in \{2, 4, 8\}$. Wir erzeugen für jede Kombination aus v und R die resultierenden Apaptoren und fügen sie in unser D/70-Netz ein. Im Anschluss werden die Adaptoren im Fine-Tuning mit $\mathcal{D}^{\text{train}*/7}$ trainiert.

In Tabelle 5.14 ist die Anzahl der Parameter der Adaptoren für die unterschiedlichen Einstellungen von v und R gezeigt. Zu sehen ist, dass bei einem Erhöhen von v die Anzahl der Parameter steigt und bei einem Erhöhen von R abnimmt. Dieser Zusammenhang ergibt sich aus dem Aufbau des Adaptors, da v eine Einstellung für einen Kernel der Größe $v \times v$ ist und R ein Reduktionparameter für die Anzahl der Feature Maps innerhalb des Adaptors ist. Wir sehen, dass die Kombination aus $v = 3$ und $R = 8$ die wenigsten und die Einstellung $v = 7$ und $R = 2$ die meisten Parameter benötigt.

In Tabelle 5.15 sind die für die unterschiedlichen Einstellungen von v und R resultierenden Werte des PSNRs nach dem Fine-Tuning gezeigt. Zu sehen ist, dass die Kombination aus $v = 7$ und $R = 2$ mit 29,78 dB das beste PSNR hervorbringt. Es ist auch zu sehne, dass sich mit dem Erhöhen von v das PSNR erhöht und mit dem Erhöhen von R das PSNR sinkt. In Kombination mit unseren Ergebnissen aus Tabelle 5.14 lässt sich schließen, dass mehr lernbare Parameter zu einem höheren PSNR führen.

In Tabelle 5.16 sind die für die unterschiedlichen Einstellungen von v und R resultierenden Bitraten nach dem Fine-Tuning gezeigt. Der beste Wert mit 0,2874 bpp wird dabei bei einer Einstellung von $v = 3$ und $R = 2$ erreicht. Allgemein ist festzustellen, dass mit einem höheren v die Bitrate steigt und bei einem höheren R die Bitrate sinkt. In Kombination mit unseren Ergebnissen aus Tabelle 5.14 lässt sich schließen, dass mehr lernbare Parameter zu einer höheren Bitrate führen.

Wir sehen, dass es keine beste Wahl für die Parameter v und R gibt, da es immer einen Tradeoff zwischen PSNR, Bitrate und Anzahl der Parameter gibt. Wir würden für die Wahl $v = 7$ und $R = 2$ das beste PSNR erhalten, aber dafür die schlechteste Bitrate und die meisten Parameter. Würden wir uns für $v = 3$ und $R = 8$ entscheiden, würden wir die wenigsten Parameter und die beste Bitrate haben, aber auch das schlechteste PSNR. Wir entscheiden uns für die weiteren Untersuchungen in dieser Arbeit für die Parameter $v = 3$ und $R = 2$ im Adaptornetz 2, da diese in dieser Untersuchung einen guten Tradeoff zwischen PSNR, Bitrate und Parameteranzahl aufweisen. Im Folgenden gilt also, dass das Adapternetz 2 ein D/70-Netz ist, welches durch den Einsatz der Blöcke ResBlockAdaptor$(3 \times 3, 2, N)$ (Abbildung 4.3a), ResBlockStrideAdaptor$(3 \times 3, 2, N, S)$

	$R = 2$	$R = 4$	$R = 8$
$v = 3$	695.552	228.800	**85.280**
$v = 5$	1.547.520	441.792	138.528
$v = 7$	2.825.472	761.280	218.400

Tabelle 5.14.: **Anzahl der Adaptorparameter** des Adaptornetzes 2 bei Einsatz der vorgeschlagenen Blöcke ResBlockAdaptor($v \times v, R, N$) (Abbildung 4.3a), ResBlockStrideAdaptor($v \times v, R, N, S$) (Abbildung 4.3b) und ResBlockUpAdaptor($v \times v, R, N, U$) (Abbildung 4.3c).

	$R = 2$	$R = 4$	$R = 8$
$v = 3$	29,60	29,49	29,39
$v = 5$	29,74	29,58	29,44
$v = 7$	**29,78**	29,61	29,48

Tabelle 5.15.: **PSNR** (dB) des Adaptornetzes 2 auf $\mathcal{D}^{\text{Kodak}}$ bei verschiedenen Werten für v und R unter Einsatz der vorgeschlagenen Blöcke ResBlockAdaptor($v \times v, R, N$) (Abbildung 4.3a), ResBlockStrideAdaptor($v \times v, R, N, S$) (Abbildung 4.3b) und ResBlockUpAdaptor($v \times v, R, N, U$) (Abbildung 4.3c). Als Modell wurde das zuerst auf $\mathcal{D}^{\text{train}*/70}$ trainierte Baseline-Modell genommen, welches im Anschluss eingefroren, um die Adaptoren ergänzt und auf $\mathcal{D}^{\text{train}*/7}$ weiter trainiert wurde.

	$R = 2$	$R = 4$	$R = 8$
$v = 3$	0,2957	0,2905	**0,2874**
$v = 5$	0,2996	0,2939	0,2887
$v = 7$	0,3021	0,2945	0,2895

Tabelle 5.16.: **Bitrate** [bpp] des Adaptornetzes 2 auf $\mathcal{D}^{\text{Kodak}}$ bei verschiedenen Werten für v und R unter Einsatz der vorgeschlagenen Blöcke ResBlockAdaptor($v \times v, R, N$) (Abbildung 4.3a), ResBlockStrideAdaptor($v \times v, R, N, S$) (Abbildung 4.3b) und ResBlockUpAdaptor($v \times v, R, N, U$) (Abbildung 4.3c). Als Modell wurde das zuerst auf $\mathcal{D}^{\text{train}*/70}$ trainierte Baseline-Modell genommen, welches im Anschluss eingefroren, um die Adaptoren ergänzt und auf $\mathcal{D}^{\text{train}*/7}$ weiter trainiert wurde.

(Abbildung 4.3b) und ResBlockUpAdaptor($3 \times 3, 2, N, U$) (Abbildung 4.3c) mit den Parametern $v = 3$ und $R = 2$ Adaptoren angehängt bekommt. Nur diese Adaptoren lernen im Fine-Tuning auf dem gesamten Datensatz $\mathcal{D}^{\text{train}*/7}$.

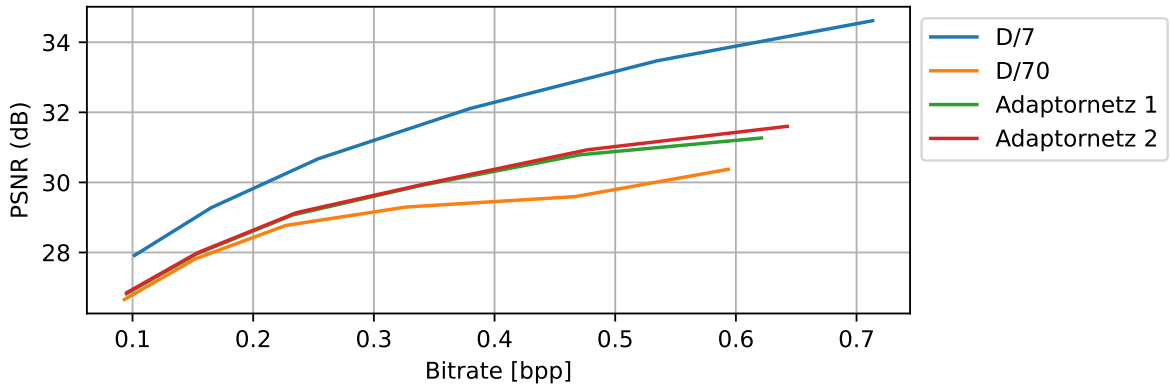


Abbildung 5.11.: Performanz auf $\mathcal{D}^{\text{Kodak}}$: Kurven über die Qualitätspunkte des schwachen Baseline-Netzes D/70, des starken Baseline-Netzes D/7, und D/70-Netze nach dem Einfügen und dem Fine-Tuning der Adaptoren mit dem Trainingsdatensatz $\mathcal{D}^{\text{train}*/7}$. "Adaptornetz 1" fügt den Adaptor (Abbildung 4.4) in ResBlock ein (Abbildung 4.3a). "Adaptornetz 2" fügt den Adaptor in ResBlock (Abbildung 4.3a), ResBlockStride (Abbildung 4.3b) und ResBlockUp (Abbildung 4.3c) ein.

5.4.3. Vergleich der Adaptornetze

In Abbildung 5.11 werden die Netze Adaptornetz 1 aus Abschnitt 5.4.1 und Adaptornetz 2 aus Abschnitt 5.4.2 miteinander verglichen. Abgebildet sind hier die Qualitätspunkte 1 bis 6 für jedes der dargestellten Netze. Wir sehen das Netz D/70 vor dem Fine-Tuning und Einfügen der Adaptoren, dann die Adaptornetze 1 und 2, sowie das Netz D/7, das wie bisher eine obere Qualitätsgrenze markiert. Es ist zu sehen, dass für niedrige Bitraten (von 0,1 bis ca. 0,23 bpp) das Netz D/70 und die beiden Adptornetze sehr nah beieinander liegen und das Netz D/7 auch einen geringeren Abstand zu diesen hat, als für höhere Bitraten. Bei höheren Bitraten sehen wir, dass sich die beiden Adaptornetze mehr von dem D/70-Netz abheben, also eine bessere Performanz aufweisen. Die beiden Adaptornetze liegen über die 6 Qualitätspunkte verteilt immer sehr nah beieinander, was bedeutet, dass ihre Performanz sehr ähnlich ist.

Um die beiden Verfahren besser miteinander vergleichen zu können, berechnen wir die BD-Rate beider Adaptornetze im Bezug zum Netz D/70. Bei dem Adaptornetz 1 kommen wir auf eine BD-Rate von -17,1569% und bei dem Adapternetz 2 auf -20,9734%. Zum Vergleich, das Netz D/7, welches direkt mit dem vollen Datensatz $\mathcal{D}^{\text{train}*/7}$ trainiert wurde, hat eine BD-Rate von -50,2060%. Man könnte meinen, dass das Adaptornetz 2 eine bessere BD-Rate als das Adaptornetz 1 hat, dass man diese Methode auswählen sollte, jedoch sollte dabei beachtet werden, dass das Adaptornetz 2 auch knapp doppelt so viele Parameter im Fine-Tuning lernt wie das Adaptornetz 1. Insgesamt lernt Adaptornetz 2 nämlich 695.552 Parameter im Fine-Tuning und das Adaptornetz 1 kommt auf gerade einmal 374.528 Parameter. Daher sehen wir uns mit einem Tradeoff konfrontiert,

Methode	BD-Rate	#Parameter	$\frac{\text{BD-Rate}}{\#\text{Parameter}} \cdot 10^6$
D/7	**-50,2060%**	11.833.149	-4,2428%
Letzte 7 Schichten	-12,6011%	1.653.772	-7,6196%
Res.-Blöcke	-23,9474%	2.066.176	-11,5902%
Alle Blöcke	**-30,7742%**	6.925.056	-4,4439%
Entropie-Parameter	-9,4889%	451.697	-21,0072%
Hyper-Encoder	-0,4201%	737.920	-0,5771%
Hyper-Decoder	-8,8766%	2.729.792	-3,2517%
Kontext-Modell	-10,0277%	819.456	-12,2370%
Adaptornetz 1	-17,1569%	**374.528**	**-45,8094%**
Adaptornetz 2	-18,1668%	695.552	-26,1185%
D/70	0,000%	-	0,0000%

Tabelle 5.17.: Vergleich der unterschiedlichen Methoden aus den Abschnitten 5.3 und 5.4. Zahl der *zu lernenden* Parameter: #Parameter.

der uns eine etwas größere Performanzsteigerung für fast doppelt so viele Parameter bietet.

5.5. Gesamtschau der Methoden

Im Folgenden werden die bereits behandelten Methoden miteinander verglichen, um zu ermitteln, welche sich für ein *parametersparsames und leistungssteigerndes* Fine-Tuning am besten eignen. Daraus lässt sich schließen, welche Methoden für ein bitrateneffizientes Update am besten geeignet sind, da wenn diese Methoden nur wenige Parameter lernen, nur diese und nicht das gesamte Netz für ein Update übertragen werden müssen.

In Tabelle 5.17 sehen wir alle bereits behandelten Methoden. Es wird angegeben, wie die Methode heißt, welche BD-Rate sie in Bezug zum Netz D/70 besitzt und wie viele Parameter bei dieser Methode gelernt werden müssen. Es wird auch angegeben, in welchem Verhältnis BD-Rate und die Anzahl der zu lernenden (und daher zu übertragenen!) Parameter stehen.

In Tabelle 5.17 ist zu sehen, dass die Methode D/7, in der wir das Netz neu auf dem gesamten Datensatz $\mathcal{D}^{\text{train}*/7}$ trainieren und in *einem* Update, wie es klassischerweise geschieht, das gesamte Netz für das Update verwenden, die beste BD-Rate erreicht. Dabei haben wir aber auch mit Abstand die meisten *zu lernenden* Parameter (#Parameter), die unser Update sehr umfangreich machen würden. Die Methode "Alle Blöcke" aus Abschnitt 5.3.3 zeigt die nächstbeste BD-Rate, gefolgt von "Res.-Blöcke" aus demselben Abschnitt. Zu sehen ist aber, dass auch hier viele Parameter verwendet werden. Die

wenigsten Parameter verwendet die Methode Adapternetz 1. Es ist naheliegend zu behaupten, dass alle Methoden, welche mehr Parameter als das Adaptornetz 1 benötigen und dabei eine schlechtere BD-Rate als diese erzielen, objektiv schlechter als das Adaptornetz 1 sind. Methoden die darunter fallen sind das Fine-Tuning der letzten 7 Schichten aus Abschnitt 5.3.1 und alle Methoden aus Abschnitt 5.3.2, also die Methoden Entropie-Parameter, Hyper-Encoder, Hyper-Decoder und Kontext-Modell. *Wir sehen also, dass das Fine-Tuning mit Adaptoren objektiv besser ist, als das Fine-Tuning der letzten 7 Schichten und das einzelner Blöcke im Hyperprior.*

Für die verbleibenden Methoden D/7, Res.-Blöcke, Alle Blöcke, Adaptornetz 1 und Adaptornetz 2 gilt, dass wenn eine Methode mehr lernende Parameter besitzt, dass diese auch eine bessere BD-Rate besitzt. Dies macht das weitere objektive Ausschließen von Methoden schwierig. Um zwischen diesen Methoden jedoch trotzdem abwägen zu können, welche für ein Update besser geeignet ist, bringen wir die BD-Rate mit der Anzahl der zu lernenden Parameter jeder Methode ins Verhältnis (letze Spalte Tabelle 5.17). Dabei wird berechnet, wie viel BD-Rate wir in jeder Methode pro 1.000.000 zu übertragener Parameter umsetzen. Diese Kennzahl bedeutet nicht, dass ein lineares Skalieren der Methoden möglich ist, sie dient lediglich der Vergleichbarkeit der Methoden untereinander. Wir sehen, dass obwohl die Methode D/7 die beste BD-Rate hat, sie wegen ihrer vielen Parameter in diesem Maß schlecht abschneidet. Zu sehen ist, dass bei diesem Maß die Methode Adapternetz 1 mit großem Abschtand vorne liegt, gefolgt von der Methode Adapternetz 2 und und Entropie-Parameter. Nach diesem Maß schneidet die Methode von Adaptornetz 1, abgesehen von Adaptonetz 2, mehr als doppelt so gut wie die restlichen Methoden ab. *Damit sehen wir, dass das Verwenden von Adaptoren in einem Update eine große Leistungssteigerung für seine wenigen Parameter, welche übertragen werden müssen, mit sich bringt.*

5.6. Update des Netzes

Auf Grundlage der Ergebnisse aus Abschnitt 5.5 führen wir in diesem Abschnitt Updates auf dem Netz D/70 nur mit den Methoden Adaptornetz 1 und Adaptornetz 2 durch, da wir diese für am besten bewertet haben.

Für das Update werden die Adaptoren aus den Adapornetzen separiert und in einem Netz, welches nur diese enthält, gespeichert. Dieses Netz wird mit Hilfe des NNCodec von Becking et al. [31] komprimiert. Dabei wird der Quantisierungsparameter auf den in Abschnitt 5.2.2 ermittelten Wert qp= -60 gesetzt, während alles sonst bei den Standardeinstellungen bleibt. Das nun komprimierte Netz (bzw. der komprimierte Netzteil) wird versendet und vom Empfänger wieder dekomprimiert. Dieser fügt die erhaltenen Adaptoren nun an die richtigen Stellen in sein D/70-Netz ein, um das Update dieses Netzes zu vollenden.

In Abbildung 5.12 ist gezeigt, wo die Qualitätspunkte des D/70-Netzes nach dem Up-

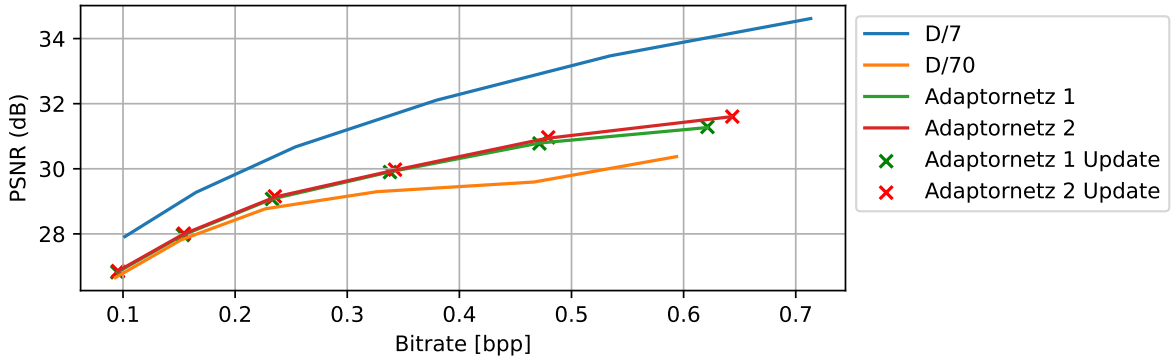


Abbildung 5.12.: Performanz auf $\mathcal{D}^{\text{Kodak}}$: Kurven über die Qualitätspunkte des schwachen Baseline-Netzes D/70, des starken Baseline-Netzes D/7, und D/70 Netze nach dem Einfügen und updaten von Adaptoren. Adaptornetz 1 Update beschreibt das Netz D/70 nach dem Update durch die komprimiert übertragenen Adaptoren aus Adaptornetz 1, selbiges gilt für Adaptornetz 2 Update mit den Adaptoren aus dem Adaptornetz 2.

Methode	Nachricht [KB] unkomprimiert	Nachricht [KB] komprimiert
D/7	50.610	15.609
Adaptornetz 1	1.498	**568**
Adaptornetz 2	2.782	1.056

Tabelle 5.18.: Durchschnittliche Nachrichtengröße der Netzupdatemethoden in unkomprimierter Form und in durch NNCodec [31] komprimierter Form.

date durch die komprimiert übertragenen Adaptoren liegen. Es ist zu sehen, dass die Qualitätspunkte der D/70-Netze, welche ein Update erfahren haben, genau auf der Kurve des Netzes liegen, von dem sie die Adaptoren erhalten haben. Das bezeugt, dass die D/70-Netze, nachdem sie ein solches Update erhalten haben, eine fast gleiche Performanz besitzen, wie die Netze, aus denen sie die Adaptoren bekommen haben.

Die dabei entstehenden wichtigen Kennzahlen sind in Tabelle 5.18 gezeigt. Zu sehen ist, dass das unkomprimierte Netz D/7, welches die klassische Methode von Netz-Updates beschreibt, bei der das gesamte Netz lernt und versendet wird, eine durchschnittliche Nachrichtengröße von 50.610 KB besitzt. In komprimierter Form werden aus den 50.610 KB nur noch 15.609 KB. Nutzen wir jedoch unsere Methode mit den Adaptoren, erzielen wir eine durchschnittliche Nachrichtengröße der Adaptoren aus Adaptornetz 1 von nur 1.498 KB. Im Vergleich zu den 50.610 KB des Netzes D/7 ist diese Methode um einiges bitratensparsamer. Dasselbe gilt für die durchschnittliche Nachrichtengröße der Adaptoren aus dem Adaptornetz 2, diese benötigt unkomprimiert 2.782 KB. Wird dieses Netz nun vor dem Versenden komprimiert, besitzt es nur noch eine durchschnittliche Nachrichtengröße von 1.056 KB. Werden die Adaptoren aus dem Adaptornetz 1 komprimiert

bevor die versendet werden, beträgt ihre durchschnittliche Nachrichtengröße nur noch 568 KB. *Diese 568 KB von Adaptornetz 1 sind im Vergleich zu den 15.609 KB, die bei der klassischen Methode (D/7) benötigt werden, nur noch gut 3,6% der Nachrichtengröße des Updates.*

6. Zusammenfassung und Ausblick

Im Folgenden fassen wir unsere Ergebnisse und Erkentnisse aus dieser Arbeit zusammen und geben einen Ausblick dazu, welche Untersuchungen zu diesem Thema noch von Interesse sein könnten.

Wir haben untersucht, wie wir ein bitrateneffizientes Update eines neuronalen Netzes realisieren können. Dabei wollten wir ein schwaches Netz, welches auf einem reduzierten Datensatz trainiert wurde, zu einem performanteren Netz updaten, ohne dabei unsere Daten-Domäne zu ändern. Das Update gestaltet sich dabei so, dass ein schwach trainiertes Netz in einem Fine-Tuning mit angefügten Adaptoren mit vollem Datensatz trainiert wird, die Gewichte des schwach trainierten Netzes jedoch fixiert werden. Die nachtrainierten Adaptoren werden im Anschluss für das Netz-Update genutzt, somit wird nicht das gesamte Netz, sondern nur wenige Adaptoren im Update versendet. Die Gewichte dieser Adaptoren werden jedoch vor dem Versenden noch komprimiert und beim Empfänger, welcher die Adaptoren in sein Netz an gleicher Stelle einbaut, dekomprimiert. Durch das Einsetzen an gleicher Stelle in ein gleiches schwach trainiertes Netz wird somit sende- und empfängerseitig ein Netz mit ähnlicher Performanz zur Verfügung gestellt.

Der Einsatz der neu entwickelten Adaptoren für ein Update eines Faltungsnetzes zur Bildcodierung hat in unseren Untersuchungen zu einer Verbesserung der Performanz (BD-Rate in Bezug auf das Netz vor dem Update) des Netzes nach dem Update geführt, wobei es nur einen Bruchteil der Parameter des gesamten Netzes für das Update benötigt. Dies macht unseren Ansatz zu einem parametersparsamen und leistungssteigernden Verfahren für Netz-Updates.

Wir haben den Einsatz unserer Adaptoren mit dem Fine-Tuning weniger bereits bestehender Schichten und deren Nutzen für ein Netz-Update verglichen. Dabei stellte sich heraus, dass der Einsatz unserer Adaptoren in einer Gegenüberstellung von BD-Rate (bezüglich des Netzes vor dem Update) und Anzahl der zu übertragenen Parameter besser abschneidet, als alle von uns untersuchten Methoden des Fine-Tunings ausgewählter bereits bestehender Schichten.

In einem weiteren Experiment haben wir gezeigt, dass ein Update durch unsere Adaptoren bitrateneffizienter ist als ein klassisches Netz-Update mittels Versenden des gesamten, vollständig nachtrainierten Netzes. Dazu haben wir ein Netz, welches komprimiert als Update dient, mit unserem Ansatz verglichen und erzielten eine Reduktion der Update-Nachrichtengröße von ca. 15.609 KB auf nur ca. 568 KB bei Verwendung unserer Adaptoren. Dadurch konnten wir unser Update auf nur gut 3,6% der Nachrich-

tengröße klassischer Netz-Updates reduzieren und behalten dabei immer noch gut ein Drittel der Performanzsteigerung (BD-Rate in Bezug auf das Netz vor dem Update) der klassischen Methode. Dies macht unseren Ansatz zu einer bitrateneffizienten Methode, eine signifikante Steigerung der Performanz (im PSNR oder der BD-Rate) zu erzielen.

Untersuchungen, die weitere möglicherweise interessante Ergebnisse erbringen können, wären zum Beispiel der Einsatz des Adaptors wie in Adaptornetz 1 und 2, jedoch nur im Encoder oder nur im Decoder. Auch interessant wäre eine Untersuchung, wie sich die Adaptoren bewähren, wenn sie seriell statt wie in dieser Arbeit parallel eingesetzt werden. Auch der Einsatz des Adaptors von Chen et al. [4] in unsere Adaptornetze 1 und 2 wäre eine aufschlussreiche Untersuchung mit der unsere Adaptoren und die von Chen et al. noch besser vergleichbar werden. Auch eine Gegenprobe mit dem Adaptor, wie ihn Shen et al. [5] eingesetzt haben, wobei unser Adaptor seriell und nur im Decoder eingefügt wird, wäre für einen Vergleich unserer und der bestehenden Methoden durchaus noch interessant.

Literaturverzeichnis

[1] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems*, 36, 2024.

[2] Tianyi Zhang, Felix Wu, Arzoo Katiyar, Kilian Q. Weinberger, and Yoav Artzi. Revisiting Few-sample BERT Fine-Tuning, 2021.

[3] Rabeeh Karimi Mahabadi, James Henderson, and Sebastian Ruder. Compacter: Efficient Low-Rank Hypercomplex Adaptor Layers. *Advances in Neural Information Processing Systems*, 34:1022–1035, 2021.

[4] Hao Chen, Ran Tao, Han Zhang, Yidong Wang, Xiang Li, Wei Ye, Jindong Wang, Guosheng Hu, and Marios Savvides. Conv-Adaptor: Exploring Parameter Efficient Transfer Learning for ConvNets. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, pages 1551–1561, Seattle, WA, Washington, June 2024.

[5] Sheng Shen, Huanjing Yue, and Jingyu Yang. Dec-Adaptor: Exploring Efficient Decoder-Side Adaptor for Bridging Screen Content and Natural Image Compression. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 12887–12896, Paris, France, October 2023.

[6] Kyuheon Kim. White Paper on Neural Network Coding. `https://isotc.iso.org/livelink/livelink/open/jtclsc29ag3`, 2024.

[7] Peter Vogel. *Signaltheorie und Kodierung*, chapter Quellencodierung, page 280. Springer Berlin Heidelberg, 1999.

[8] Dirk W. Hoffmann. *Einführung in die Informations- und Codierungstheorie*, chapter Arithmetische Codierung, pages 254–259. Springer Berlin Heidelberg, 2014.

[9] Dirk W. Hoffmann. *Einführung in die Informations- und Codierungstheorie*, chapter Entropiecodierungen, pages 247–251. Springer Berlin Heidelberg, 2014.

[10] Peter Vogel. *Signaltheorie und Kodierung*, chapter AD-Umsetzung, pages 243–246. Springer Berlin Heidelberg, 1999.

[11] Robert M. Gray and Allen Gersho. *Vector Quantization and Signal Compression*, chapter Nonuniform Quantization and Companding, pages 156–161. Springer New York, NY, 1991.

[12] Robert M. Gray and Allen Gersho. *Vector Quantization and Signal Compression*, chapter Vector Coding, pages 307–688. Springer New York, NY, 1991.

[13] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems*, 25, 2012.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, Las Vegas, NV, USA, June 2016.

[15] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Nets. *Advances in Neural Information Processing Systems*, 27, 2014.

[16] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[17] Aaron Van den Oord, Nal Kalchbrenner, Lasse Espeholt, Oriol Vinyals, Alex Graves, et al. Conditional Image Generation with PixelCNN Decoders. *Advances in Neural Information Processing Systems*, 29, 2016.

[18] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-Based Learning Applied to Document Recognition. *In Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[19] Rüdige Seydel. *Höhere Mathematik im Alltag*, chapter Bildkompression und JPEG, pages 71–77. Springer Berlin Heidelberg, 2022.

[20] Barry G. Haskell and Arun N. Netravali. *Digital Pictures*, chapter Still Image Coding Standards - ISO JBIG and JPEG, pages 581–589. Springer New York, NY, 1995.

[21] Mohammed Ghanbari. *Standard Codecs: Image compression to advanced video coding*, chapter Coding of still pictures (JPEG and JPEG2000), pages 115–121. IET telecommunications series, 2011.

[22] Barry G. Haskell and Arun N. Netravali. *Digital Pictures*, chapter Still Image Coding Standards - ISO JBIG and JPEG, pages 571–581. Springer New York, NY, 1995.

[23] Fabian Mentzer, Eirikur Agustsson, Michael Tschannen, Radu Timofte, and Luc Van Gool. Conditional Probability Models for Deep Image Compression. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4394–4402, Salt Lake City, UT, USA, June 2018.

[24] Johannes Ballé, David Minnen, Saurabh Singh, Sung Jin Hwang, and Nick Johnston. Variational Image Compression with a Scale Hyperprior. *arXiv preprint arXiv:1802.01436*, pages 1–23, 2018.

[25] Zhengxue Cheng, Heming Sun, Masaru Takeuchi, and Jiro Katto. Learned Image Compression with Discretized Gaussian Mixture Likelihoods and Attention Modules. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7939–7948, Seattle, WA, USA, June 2020.

[26] Nick Johnston, Damien Vincent, David Minnen, Michele Covell, Saurabh Singh, Troy Chinen, Sung Jin Hwang, Joel Shor, and George Toderici. Improved Lossy Image Compression with Priming and Spatially Adaptive Bit Rates for Recurrent Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4385–4393, Salt Lake City, UT, USA, June 2018.

[27] Lucas Theis, Wenzhe Shi, Andrew Cunningham, and Ferenc Huszár. Lossy Image Compression with Compressive Autoencoders. In *Proceedings of International Conference on Learning Representations*, Toulon, Frankreich, April 2017.

[28] George Toderici, Damien Vincent, Nick Johnston, Sung Jin Hwang, David Minnen, Joel Shor, and Michele Covell. Full Resolution Image Compression with Recurrent Neural Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5306–5314, Honolulu, HI, USA, July 2017.

[29] David Minnen, Johannes Ballé, and George Toderici. Joint Autoregressive and Hierarchical Priors for Learned Image Compression. In *Proceedings of the Conference on Neural Information Processing Systems (NIPS)*, pages 10794–10803, Montréal, Canada, December 2018.

[30] M.W. Marcellin and T.R. Fischer. Trellis Coded Quantization of Memoryless and Gauss-Markov Sources. *IEEE Transactions on Communications*, 38(1):82–93, 1990.

[31] Daniel Becking, Paul Haase, Heiner Kirchhoffer, Karsten Müller, Wojciech Samek, and Detlev Marpe. NNCodec: An Open Source Software Implementation of the Neural Network Coding ISO/IEC Standard. In *Proceedings of the ICML Workshop Neural Compression: From Information Theory to Applications*, Hawaii, HI, USA, July 2023.

[32] Detlev Marpe, Heiko Schwarz, and Thomas Wiegand. Context-Based Adaptive Binary Arithmetic Coding in the H.264/AVC Video Compression Standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):620–636, 2003.

[33] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-Efficient Transfer Learning for NLP. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2790–2799. PMLR, Jun 2019.

[34] Minh Quang Pham, Josep-Maria Crego, François Yvon, and Jean Senellart. A Study of Residual Adaptors for Multi-Domain Neural Machine Translation. In *Proccedings of the Conference on Machine Translation*, pages 615–626, online, November 2020.

[35] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-Rank Adaptation of Large Language Models. *arXiv preprint arXiv:2106.09685*, 2021.

[36] Vladislav Lialin, Vijeta Deshpande, and Anna Rumshisky. Scaling Down to Scale Up: A Guide to Parameter-Efficient Fine-Tuning. *arXiv preprint arXiv:2303.15647*, 2023.

[37] Marcellin Michael W. Taubman, David S. *JPEG2000 Image Compression Fundamentals, Standards and Practice*, chapter Elementary Concepts, pages 3–8. Springer, 2002.

[38] Gisle Bjøntegaard. Calculation of Average PSNR Differences Between RD-Curves. *ITU SG16 Doc. VCEG-M33*, 2001.

[39] Hiroshi Akima. A New Method of Interpolation and Smooth Curve Fitting Based on Local Procedures. *Journal of the ACM (JACM)*, 17(4):589–602, 1970.

[40] Jonas Pfeiffer, Aishwarya Kamath, Andreas Rücklé, Kyunghyun Cho, and Iryna Gurevych. Adaptorfusion: Non-Destructive Task Composition for Transfer Learning. *arXiv preprint arXiv:2005.00247*, 2020.

[41] Jonas Pfeiffer, Andreas Rücklé, Clifton Poth, Aishwarya Kamath, Ivan Vulić, Sebastian Ruder, Kyunghyun Cho, and Iryna Gurevych. Adaptorhub: A Framework for Adapting Transformers. *arXiv preprint arXiv:2007.07779*, 2020.

[42] Jonas Pfeiffer, Ivan Vulić, Iryna Gurevych, and Sebastian Ruder. Mad-x: An Adaptor-Based Framework for Multi-Task Cross-Lingual Transfer. *arXiv preprint arXiv:2005.00052*, 2020.

[43] Tianfan Xue, Baian Chen, Jiajun Wu, Donglai Wei, and William T Freeman. Video Enhancement with Task-Oriented Flow. *International Journal of Computer Vision (IJCV)*, 127(8):1106–1125, 2019.

[44] Kodak Lossless True Color Image Suite. Online. Available: `http://r0k.us/graphics/kodak/`.

[45] Johannes Ballé, Valero Laparra, and Eero P Simoncelli. Density Modeling of Images using a Generalized Normalization Transformation. *arXiv preprint arXiv:1511.06281*, 2015.

[46] Jean Bégaint, Fabien Racapé, Simon Feltman, and Akshay Pushparaja. CompressAI: A PyTorch Library and Evaluation Platform for End-to-End Compression Research. *arXiv preprint arXiv:2011.03029*, November 2020.

[47] Diederik P Kingma. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*, 2014.

A. Notationsverzeichnis

Notation	Erklärung
A, B, C	Diskrete Symbole
b	Anzahl Bits im gesamten Bitstrom
b_i	Bias
$\mathbf{b}^H$	Bitstrom des Hyperpriors
$\mathbf{b}^R$	Haupt-Bitstrom
D	Parameter für tiefenweise Faltung (Aufteilung der Feature-Maps in D Blöcke)
Δ	Quantisierungsschrittweite
F	Fläche
h	Höhe (zum Beispiel einer Eingabe-Feature-Map)
H	Höhe des Originalbildes
Θ	Parameter eines neuronalen Netzes
$\hat{\Theta}$	Quantisierte Parameter eines neuronalen Netzes
$I(\hat{x})$	Informationsgehalt (von $\hat{x}$)
K	Kernelgröße ($K \times K$)
λ	Hyperparameter
$\mathcal{L}$	Verlust
M	Anzahl Features/Feature-Maps für die Eingabe
$\mathbf{M}$	Mittelwert-Tensor
N	Anzahl Features/Feature-Maps für die Eingabe/Ausgabe
$\mathcal{N}(\mathbf{M}, \mathbf{\Sigma})$	Gaußsche Verteilungsdichtefunktion der Entropie-Parameter
$P(\hat{x})$	Auftrittswahrscheinlichkeit (von $\hat{x}$)
qp	Quantisierungsparameter
R	Parameter zur Dimensionsveränderung
S	Stride
$\mathbf{\Sigma}$	Kovarianz-Tensor
v	Kernelgröße ($v \times v$)
w	Breite (zum Beispiel einer Eingabe-Feature-Map)
$w_i, w_{i,j}$	Gewicht
$w\prime_{i,j}$	Gewicht in einer maskierten Faltungsschicht, vor der Maskierung
x	Eingabewert aus $\mathbb{R}$ oder umfangreichen diskreten Wertebereich
$\mathbf{x}$	Tensor, zum Beispiel Bild
$\hat{x}$	Quantisierung von x (diskretes Symbol)
$\hat{\mathbf{x}}$	Quantisierung von Tensor $\mathbf{x}$
$X_i, X_{i,j}$	Eingabewerte einer Schicht

Notation	Erklärung
y	Wert aus $\mathbb{R}$ oder umfangreichen diskreten Wertebereich
$\mathbf{y}$	Tensor
$\hat{y}$	Quantisierung von y
$\hat{\mathbf{y}}$	Quantisierung von $\mathbf{y}$
$Y_i, Y_{i,j}$	Ausgabewerte einer Schicht
$\mathbf{z}$	Tensor
$\hat{\mathbf{z}}$	Quantisierung von $\mathbf{z}$