TECHNISCHE UNIVERSITÄT DRESDEN

Technische Universität Dresden
Bereich Mathematik und Naturwissenschaften
Fakultät Physik

Institut füer Kern- und Teilchenphysik
Emmy-Noether-Nachwuchsgruppe

Bachelor-Arbeit
zur Erlangung des Hochschulgrades
Bachelor of Science
im Studiengang
Physik

Anwendung von maschinellem Lernen zur Regression am Beispiel des Diphoton-Prozesses

vorgelegt von
Andreas Weitzel
geboren am 10.08.1999 in Fulda

eingereicht am 24.05.2021

Dokument erstellt mit pdfLaTeX.

Erstgutachter:	Dr. Frank Siegert
Zweitgutachter:	Prof. Dr. Arno Streassner

Zusammenfassung

<Zusammenfassung deutsch> In dieser Arbeit werden Sie die beste Bachelor-Arbeit der Welt finden.

Abstract

<Abstract english>

Inhaltsverzeichnis

1. Einleitung

2. Diphoton-Prozess

2.1. Partonischer Diphoton-Prozess

Im folgenden wir der Diphoton-Prozess auf partonischer Ebene behandelt. Explizit bedeutet dies die Interaktion $q\overline{q} \to \gamma\gamma$. Unser Ziel ist die Bestimmung der differentiellen Wirkungsquerschnitte, die wir über Fermis goldene Regel berechnen wollen. Dazu benötigen wir die lorenz-invarianten Matrixübergangselemente der grundlegenden Reaktion. Das heißt wir erstellen zunächst alle möglichen Feynman-Diagramme und ein Diagramm, das die Kinematik des Prozesses zeigt.

Hier befinden sich die Feynman-Diagramme des u- und t-Kanals!

Die Feynman-Regeln der QED liefern dann die folgenden Beiträge der Kanäle: t-Kanal:

$$\mathcal{M}_t = \overline{\nu}\left(p_1\right)\left(-iQ_q e\gamma^\mu\right)\epsilon^*_\mu\left(p_3\right)\left(\frac{\gamma^\mu\left(p_{1,\mu}-p_{3,\mu}\right)}{\left(p_1-p_3\right)^2}\right)\left(-iQ_q e\gamma^\nu\right)\epsilon^*_\nu\left(p_4\right)u\left(p_2\right) \tag{2.1}$$

u-Kanal:

$$\mathcal{M}_u = \overline{\nu}\left(p_1\right)\left(-iQ_q e\gamma^\mu\right)\epsilon^*_\mu\left(p_4\right)\left(\frac{\gamma^\mu\left(p_{1,\mu}-p_{4,\mu}\right)}{\left(p_1-p_4\right)^2}\right)\left(-iQ_q e\gamma^\nu\right)\epsilon^*_\nu\left(p_3\right)u\left(p_2\right) \tag{2.2}$$

Dabei wurden die Massen der Quarks vernachlässigt.

Wir nutzen die Notation $\gamma^\mu p_\mu = \not{p}$ und die Mandelstam-Variablen und vereinfachen damit die Ausdrücke Gleichung 2.1 und Gleichung 2.2 zu:

$$\mathcal{M}_t = -\frac{Q_q^2 e^2}{t}\left[\overline{\nu}(p_1)\gamma^\mu\epsilon^*_\mu(p_3)\left(\not{p}_1-\not{p}_3\right)\gamma^\nu\epsilon^*_\nu(p_4)u(p_2)\right] \tag{2.3}$$

$$\mathcal{M}_u = -\frac{Q_q^2 e^2}{u}\left[\overline{\nu}(p_1)\gamma^\mu\epsilon^*_\mu(p_4)\left(\not{p}_1-\not{p}_4\right)\gamma^\nu\epsilon^*_\nu(p_3)u(p_2)\right] \tag{2.4}$$

Explizites Einsetzen der Vierervektoren aus der kinematischen Skizze im Schwerpunktsystem, in dem beide einlaufenden Quarks jeweils einen Impuls p besitzen, führt auf:

$$t = (p_1-p_3)^2 = -4p^2\cos^2\left(\frac{\theta}{2}\right) \quad \text{und} \quad u = (p_1-p_4)^2 = -4p^2\sin^2\left(\frac{\theta}{2}\right) \tag{2.5}$$

Wir können nun das gesamt Matrixelement berechnen, indem wir die Anteile des u- und t-Kanals summieren:

$$\begin{aligned}\mathcal{M} = \mathcal{M}_u + \mathcal{M}_t &= \mathcal{F}\left[\overline{\nu}(p_1)\left(\frac{\Gamma_t}{\cos^2\left(\frac{\theta}{2}\right)} + \frac{\Gamma_u}{\sin^2\left(\frac{\theta}{2}\right)}\right)u(p_2)\right] \\ &= \mathcal{F}\left[\overline{\nu}(p_1)\Gamma u(p_2)\right]\end{aligned} \tag{2.6}$$

Wobei wir die Ersetzungen **??** gewählt haben.

$$\begin{gathered}\Gamma_t = \gamma^\mu \epsilon^*_\mu(p_3)(\not{p}_1 - \not{p}_3)\gamma^\nu \epsilon^*_\nu(p_4) \quad \text{und} \quad \Gamma_u = \gamma^\mu \epsilon^*_\mu(p_4)(\not{p}_1 - \not{p}_4)\gamma^\nu \epsilon^*_\nu(p_3) \\ \text{sowie} \quad \mathcal{F} = \frac{Q_q^2 e^2}{4p^2} \quad \text{und} \quad \Gamma = \frac{\Gamma_t}{\cos^2\left(\frac{\theta}{2}\right)} + \frac{\Gamma_u}{\sin^2\left(\frac{\theta}{2}\right)} \\ \cos^2\left(\frac{\theta}{2}\right) = a \quad \text{und} \quad \sin^2\left(\frac{\theta}{2}\right) = b\end{gathered} \tag{2.7}$$

Um das gemittelte Quadrat des Betrags nun zu berechnen, müssen wir Polarisation und Helizität der Photonen summieren, sowie durch die Anzahl der Anfangszustände der eingehenden Quarks teilen. Die Quarks können drei verschiedene Farbzustände und jeweils zwei verschiedene Helizitäten annehmen. Insgesamt liefern die Anfangszustände also einen Faktor 1/12:

$$\left\langle |\mathcal{M}|^2 \right\rangle = \frac{1}{12}\sum_{Hel.}\sum_{Pol.} |\mathcal{M}|^2 \tag{2.8}$$

Um die Summe über die Helizitäten auszuführen, verwenden wir Casimirs Trick:

$$\sum_{Hel.} |\mathcal{M}|^2 = \mathcal{F}^2 \sum_{Hel.} \left[\overline{\nu}(p_1)\Gamma u(p_2)\right]\left[\overline{\nu}(p_1)\Gamma u(p_2)\right]^* = \mathcal{F}^2 \ \mathrm{Tr}\left[\Gamma \not{p}_2 \overline{\Gamma} \not{p}_1\right] \tag{2.9}$$

Wobei $\overline{\Gamma} = \gamma^0 \Gamma^\dagger \gamma^0 = \frac{\overline{\Gamma}_t}{a} + \frac{\overline{\Gamma}_u}{b}$ die Dirac-Adjungierte bezeichnet. Für die Dirac-adjungierten $\overline{\Gamma}_t, \overline{\Gamma}_u$ ergibt sich:

$$\overline{\Gamma}_t = \gamma^\nu \epsilon_\nu(p_4)(\not{p}_1 - \not{p}_3)\gamma^\mu \epsilon_\mu(p_3) \quad \text{und} \quad \overline{\Gamma}_u = \gamma^\nu \epsilon_\nu(p_3)(\not{p}_1 - \not{p}_4)\gamma^\mu \epsilon_\mu(p_4) \tag{2.10}$$

?? wird damit zu:

$$\mathcal{F}^2 \ \mathrm{Tr}\left[\Gamma \not{p}_2 \overline{\Gamma} \not{p}_1\right] = \mathcal{F}^2 \ \mathrm{Tr}\left[\frac{1}{a^2}\Gamma_t \not{p}_2 \overline{\Gamma}_t \not{p}_1 + \frac{1}{ab}\Gamma_t \not{p}_2 \overline{\Gamma}_u \not{p}_1 + \frac{1}{ba}\Gamma_u \not{p}_2 \overline{\Gamma}_t \not{p}_1 + \frac{1}{b^2}\Gamma_u \not{p}_2 \overline{\Gamma}_u \not{p}_1\right] \tag{2.11}$$

Wobei wir die Terme von links nach rechts nach folgendem Schema benennen:

$$T_{ij} = \mathcal{F}^2 \ \mathrm{Tr}\left[\frac{1}{ij}\Gamma(i) \not{p}_2 \overline{\Gamma}(j) \not{p}_1\right] \quad \text{mit} \quad i, j \in \{a, b\} \tag{2.12}$$

Wir evaluieren nun diese Terme. Wir beginnen mit den Fällen $i = j$:

$$\begin{aligned} T_{aa} &= \frac{1}{a^2}\ \mathrm{Tr}\left[\gamma^\mu \epsilon^*_\mu(p_3)(\not{p}_1 - \not{p}_3)\gamma^\nu \epsilon^*_\nu(p_4)\not{p}_2\gamma^\nu \epsilon_\nu(p_4)(\not{p}_1 - \not{p}_3)\gamma^\mu \epsilon_\mu(p_3)\not{p}_1\right] \\ &= \frac{1}{a^2}\epsilon^*_\mu(p_3)\epsilon_\mu(p_3)\epsilon^*_\nu(p_4)\epsilon_\nu(p_4)\ \mathrm{Tr}\left[-2\gamma^\mu(\not{p}_1 - \not{p}_3)\not{p}_2(\not{p}_1 - \not{p}_3)\gamma^\mu\not{p}_1\right] \\ &= \frac{4\epsilon}{a^2}\ \mathrm{Tr}\left[(\not{p}_1 - \not{p}_3)\not{p}_2(\not{p}_1 - \not{p}_3)\not{p}_1\right] \\ &= \frac{32\epsilon}{a^2}(p_3 \cdot p_2)(p_3 \cdot p_1) \end{aligned} \tag{2.13}$$

Wobei die Abkürzung $\epsilon = \epsilon^*_\mu(p_3)\epsilon_\mu(p_3)\epsilon^*_\nu(p_4)\epsilon_\nu(p_4)$ verwendet wurde. Es folgt analog:

$$T_{bb} = \frac{32\epsilon}{b^2}(p_4 \cdot p_2)(p_4 \cdot p_1) \tag{2.14}$$

Für $i \neq j$ ergibt sich:

$$\begin{aligned} T_{ab} &= \frac{1}{ab}\ \mathrm{Tr}\left[\gamma^\mu \epsilon^*_\mu(p_4)(\not{p}_1 - \not{p}_4)\gamma^\nu \epsilon^*_\nu(p_3)\not{p}_2\gamma^\nu \epsilon_\nu(p_4)(\not{p}_1 - \not{p}_3)\gamma^\mu \epsilon_\mu(p_3)\not{p}_1\right] \\ &= \frac{\epsilon}{ab}\ \mathrm{Tr}\left[\gamma^\mu(\not{p}_1 - \not{p}_4)\gamma^\nu\not{p}_2\gamma^\nu(\not{p}_1 - \not{p}_3)\gamma^\mu\not{p}_1\right] \quad \text{hier noch Fehler!} \\ &= ... \\ &= \frac{16\epsilon}{ab}\left[(p_1 \cdot p_2)\left[-2\,(p_1 \cdot p_4) + (p_3 \cdot p_4)\right] - (p_1 \cdot p_3)(p_2 \cdot p_4) + (p_2 \cdot p_3)(p_1 \cdot p_4)\right] \end{aligned} \tag{2.15}$$

und analog:

$$T_{ba} = \frac{16\epsilon}{ab}\left[(p_1 \cdot p_2)\left[-2\,(p_1 \cdot p_3) + (p_3 \cdot p_4)\right] - (p_1 \cdot p_4)(p_2 \cdot p_3) + (p_1 \cdot p_3)(p_2 \cdot p_4)\right] \tag{2.16}$$

Beim Einsetzen der expliziten Vierervektoren aus **??**, fällt auf, dass $T_{ab} + T_{ba} = 0$. Wir haben nun die Summe über die Helizitäten ausgeführt und können damit **??** umschreiben zu:

$$\left\langle |\mathcal{M}|^2 \right\rangle = \frac{\mathcal{F}^2}{12}\sum_{Pol.} 32\epsilon\left(\frac{1}{a^2}(p_3 \cdot p_2)(p_3 \cdot p_1) + \frac{1}{b^2}(p_4 \cdot p_2)(p_4 \cdot p_1)\right) \tag{2.17}$$

Um die Summe über die verschiedenen Polarisationen auszuführen, verwenden wir die Vollständigkeitsrelation von realen Photonen:

$$\sum_{Pol.} \epsilon^\mu \epsilon^{*\nu} = -g^{\mu\nu} \tag{2.18}$$

Damit erhalten wir:

$$\begin{aligned} \left\langle |\mathcal{M}|^2 \right\rangle &= \frac{8}{3}\mathcal{F}^2\left(\frac{1}{a^2}(p_3 \cdot p_2)(p_3 \cdot p_1) + \frac{1}{b^2}(p_4 \cdot p_2)(p_4 \cdot p_1)\right) \\ &= \frac{2}{3}Q_q^4 e^4\left[\frac{1 - \cos^2(\theta)}{\cos^4\left(\frac{\theta}{2}\right)} + \frac{1 - \cos^2(\theta)}{\sin^4\left(\frac{\theta}{2}\right)}\right] \\ &= \frac{4}{3}Q_q^4 e^4 \frac{1 + \cos^2(\theta)}{\sin^2(\theta)} \end{aligned} \tag{2.19}$$

Wir wollen unser Ergebnis noch in Abh. der Pseudo-Rapidität angeben, da sich diese additiv unter Lorentz-Transformationen verhält und wir im Verlauf der Arbeit noch den hadronischen Prozess besprechen werden und sich das Schwerpunktsystem der Partonen von dem der Hadronen unterscheidet. Sie ist definiert als $\eta = -\ln\left(\tan\left(\frac{\theta}{2}\right)\right)$. Die Umformung gelingt am einfachsten mithilfe der Identität $\cos(\theta) = \tanh(\eta)$.

$$\left\langle |\mathcal{M}|^2 \right\rangle = \frac{4}{3} Q_q^4 e^4 \cosh(2\eta) \tag{2.20}$$

2.2. Differentieller Wirkungsquerschnitt des partonischen Prozesses

Um aus dem mittleren Betragsquadrat des Übergangsmatrixelementes einen Wirkungsquerschnitt berechnen zu können, bemühen wir Fermis goldene Regel für einen Prozess $1 + 2 \to 3 + 4$.

$$\sigma = \frac{(2\pi)^4}{2E_1 2E_2 (v_1 + v_2)} \int \left\langle |\mathcal{M}|^2 \right\rangle \delta(E_1 + E_2 - (E_3 + E_4)) \delta^3(\vec{p}_1 + \vec{p}_2 - \vec{p}_3 - \vec{p}_4) \frac{\mathrm{d}^3\vec{p}_3}{(2\pi)^3 2E_3} \frac{\mathrm{d}^3\vec{p}_4}{(2\pi)^3 2E_4} \tag{2.21}$$

Wir betrachten Ausdruck **??** im Schwerpunktsystem, in dem also gilt $E_1 = E_2$ sowie $\vec{p}_1 + \vec{p}_2 = 0$. Wir führen den Flussfaktor $F = 2E_1 2E_2 (v_1 + v_2)$ und nutzen dessen lorentz-invariante Form $F = 4\left[(p_1 p_2)^2 - m_1^2 m_2^2\right]^{\frac{1}{2}} \approx 4(p_1 p_2) = s$. Wir können mithilfe der Delta-Distribution der Impulse das Integral über $\vec{p}_3$ oder $\vec{p}_4$ auswerten und ersetzen die verbleibende Integration durch eine Integration über das Raumwinkelelement $\mathrm{d}^3\vec{p} = |\vec{p}|^2 \mathrm{d}|\vec{p}|\mathrm{d}\Omega$. Wir erhalten schließlich:

$$\sigma = \frac{1}{64\pi^2 s} \int \left\langle |\mathcal{M}|^2 \right\rangle \mathrm{d}\Omega = \frac{1}{32\pi s} \int \left\langle |\mathcal{M}|^2 \right\rangle \sin(\theta) \mathrm{d}\theta \tag{2.22}$$

Dabei konnten wir die $\mathrm{d}\phi$-Integration durchführen, da das Übergangsmatrixelement keine ϕ-Abhängigkeit zeigt. Für den differentiellen Wirkungsquerschnitt $\frac{\mathrm{d}\sigma}{\mathrm{d}\theta}$ ergibt sich dann:

$$\frac{\mathrm{d}\sigma}{\mathrm{d}\theta} = \frac{Q_q^4 e^4}{24\pi s} \frac{1 + \cos^2(\theta)}{\sin(\theta)} \tag{2.23}$$

daraus ergibt sich leicht der differentielle Wirkungsquerschnitt in Abhängigkeit von η

$$\frac{\mathrm{d}\sigma}{\mathrm{d}\eta} = \frac{\mathrm{d}\theta}{\mathrm{d}\eta} \frac{\mathrm{d}\sigma}{\mathrm{d}\theta} = \frac{Q_q^4 e^4}{48\pi s} (1 + \tanh^2(\eta)) \tag{2.24}$$

2.3. Hadronischer Diphoton Prozess

Der in **??** behandelte Prozess ist zwar sehr nützlich, spiegelt jedoch nicht die wahre Natur des Diphoton-Prozesses wider. In unserer Welt sind die Quarks durch das sogenannte Confinement in ihrem gegenseitigen Potential eingesperrt und kommen

somit nicht als freie Teilchen vor. Das Schwerpunktsystem aus **??** lässt sich also experimentell nicht erreichen. In Wahrheit müssen wir hier Hadronen betrachten, die die jeweiligen Quarks enthalten, die dann annihilieren sollen. In unserem Fall behandeln wir hierbei Protonen, die aus zwei up-Quarks und zwei down-Quarks bestehen. Lassen wir zwei Protonen in beispielsweise einem Speicherring mit genügend hohen Energien aufeinanderprallen, wird die Substruktur des Protons aufgelöst und die Konstituenten des Protons können miteinander interagieren. Bei diesen Interaktionen können die Quarks dann als freie Teilchen betrachtet werden. Wir untersuchen also explizit die Interaktion $pp \to \gamma\gamma$.

Ein Problem, das wir nun beachten müssen, ist dass die Partonen im Proton nicht still sitzen, sondern sich bewegen. Auch befinden sich im Proton durchgehend Quark-Antiquark-Paare, die durch den Zerfall der Austauschteilchen der starken Wechselwirkung, den Gluonen, entstehen. Wir nennen diese Quarks See-Quarks und die Quarks die permanent im Hadron sitzen und seine Quantenzahlen ausmachen Valenzquarks. Wir können also nicht stumpf jedem Konstituentenquark 1/3 des Gesamtimpulses des Protons zuordnen, sondern müssen uns intensiver mit den Impulsen auseinandersetzen. Wir formulieren unser Modell hierbei in einem System, in dem das Proton eine sehr hohe Energie $E \gg m_p$ hat. In diesem Bezugssystem können wir die Masse des Protons im Vergleich zu seiner kinetischen Energie vernachlässigen. Wir schreiben den Vierervektor eines Protons zu $p_p = (E, 0, 0, E)$, legen also seinen Impuls parallel zur z-Achse. Wir können nun einem Parton einen unbestimmten Bruchteil ξ des Impulses zuordnen und damit seinen Vierervektor in **??** ausdrücken:

$$p_q = (\xi E, 0, 0, \xi E) = \xi p_p \tag{2.25}$$

Findet bei einer Interaktion ein Impulsübertrag q statt, so wird $\xi p_p \to (\xi p_p + q)$. Wir betrachten nun die invariante Masse beider Zustände(**??**)

$$(\xi p_p)^2 = m_q^2 \quad \text{und} \quad (\xi p_p + q)^2 = (\xi p_p)^2 + 2\xi p_p \cdot q + q^2 = m_q^2 \tag{2.26}$$

Mithilfe von **??** können wir nun die Identifikation **??** durchführen:

$$2\xi p_p \cdot q + q^2 = 0 \quad \Rightarrow \quad \xi = \frac{-q^2}{2p_p \cdot q} = x \tag{2.27}$$

Das x in **??** ist hierbei die Bjorken-Skalenvariable. Diese repräsentiert also bei hohen Proton-Impulsen den Impulsbruchteil, den ein Parton im Proton trägt.

Es ist im vornherein nun nicht klar, mit welchem Impulsbruchteil x ein Parton in die jeweilige Interaktion geht. Es ist also nicht möglich, die Reaktion $pp \to \gamma\gamma$ im Schwerpunktsystem der jeweils interagierenden Partonen zu beschreiben. Wir begeben uns also in das Schwerpunktsystem der kollidierenden Protonen und bedienen uns den sogenannten Partondichtefunktionen $f_{i,h}\left(x, Q^2\right)$. Diese PDFs beschreiben die Wahrscheinlichkeitsdichte, bei einer Energieskala $Q^2 = -q^2$, das entsprechende Parton i mit dem Impulsbruchteil x im Hadron h zu finden. Sie können nicht aus ersten Prinzipien abgeleitet werden und müssen experimentell bestimmt werden.

Wir wollen nun die Partondichtefunktionen des Protons nutzen, um einen Ausdruck für den totalen Wirkungsquerschnitt $pp \to \gamma\gamma$ zu finden. Kennen wir den totalen Wirkungsquerschnitt eines partonischen Prozesses zwischen den Partonen i und j, bei den festgelegten Impulsbruchteilen x_1 und x_2 und der Energieskala Q^2 (wir nennen diesen $\tilde{\sigma}_{i,j}(x_1, x_2, Q^2)$), dann können wir mithilfe der PDFs den totalen Wirkungsquerschnitt $\sigma_{i,j}$ für die Reaktion der Partonen i und j bei dem Zusammenstoß von zwei Protonen berechnen. In **??** ist die Kennzeichnung des Hadrons vernachlässigt.

$$\sigma_{i,j} = \int f_i(x_1, Q^2) f_j(x_2, Q^2) \tilde{\sigma}_{i,j}(x_1, x_2, Q^2) \mathrm{d}x_1 \mathrm{d}x_2 \tag{2.28}$$

Angewendet auf den Fall des Diphoton-Prozesses, bei dem Quark q und Antiquark $\overline{q}$ miteinander in Interaktion treten müssen, lässt sich der gesamte Wirkungsquerschnitt als Summe der Wirkungsquerschnitte der möglichen Partonen auffassen. Wir summieren dabei in (**??**) nicht über Antiteilchen.

$$\sigma = \sum_q \left(\sigma_{q,\overline{q}} + \sigma_{\overline{q},q}\right) \tag{2.29}$$

In Abschnitt **??** haben wir bereits die (differentiellen) Wirkungsquerschnitte für den partonischen Prozess σ_p im Schwerpunktsystem der Konstituenten berechnet. wir können $\tilde{\sigma}_{q,\overline{q}}(x_1, x_2, Q^2)$ also wie in **??** schreiben.

$$\tilde{\sigma}_{q,\overline{q}}(x_1, x_2, Q^2) = \int \frac{\mathrm{d}\sigma_p}{\mathrm{d}\eta}\left(x_1, x_2, Q^2\right) \mathrm{d}\eta \tag{2.30}$$

Wir müssen nun beachten, dass Gleichung **??** im Schwerpunktsystem der Partonen geschrieben ist. Praktisch ist es nur realisierbar, die Pseudorapidität im Schwerpunktsystem der Protonen zu messen. Diese unterscheiden sich offensichtlich, sobald $x_1 \neq x_2$ gilt. Für diesen Fall, haben wir die Pseudorapidität eingeführt, da sich diese unter Lorentztransformation additiv verhält. Weiterhin müssen wir uns um die Abhängigkeit der Mandelstam-varialbe s von x_1, x_2 kümmern, die das Quadrat der Schwerpunktsenergie der Partonen darstellt. Nach **??** gilt für die Partonen q und $\overline{q}$ mit den Impulsbruchteilen x_1 und x_2 im Schwerpunktsystem der beiden Hadronen **??**.

$$p_q = (x_1 E, 0, 0, x_1 E) \quad \text{und} \quad p_{\overline{q}} = (x_2 E, 0, 0, -x_2 E) \tag{2.31}$$

Mithilfe von **??** lässt sich die Schwerpunktenergie leicht in Abhängigkeit der Impulsbruchteile und der Strahlenergie E darstellen (**??**).

$$s = 2\sqrt{x_1 x_2} E \tag{2.32}$$

Im folgenden werden Variablen im Laborsystem ungestrichen und Variablen im Schwerpunktsystem der Partonen gestrichen benannt. Wie bereits erwähnt, verhält sich die Rapidität additiv bei Inertialsystemwechsel. Explizit heißt das, bewegt sich

das Schwerpunktsystem der Partonen mit der Geschwindigkeit β vom Laborsystem weg, berechnet sie sich nach **??**.

$$\eta' = \eta + \frac{1}{2}\ln\left(\frac{1-\beta}{1+\beta}\right) \quad \Rightarrow \quad \frac{\mathrm{d}\eta'}{\mathrm{d}\eta} = 1 \tag{2.33}$$

Wir kennen den differentiellen Wirkungsquerschnitt im bewegten System und möchten diesen nun in das Laborsystem transformieren (**??**).

$$\frac{\mathrm{d}\sigma_p}{\mathrm{d}\eta} = \frac{\mathrm{d}\eta'}{\mathrm{d}\eta}\frac{\mathrm{d}\sigma_p}{\mathrm{d}\eta'} = \frac{Q_q^4 e^4}{48\pi s}(1+\tanh^2(\eta')) \tag{2.34}$$

Die Geschwindigkeit β ergibt sich mit den Dreierimpulsen $\mathbf{p}$ zu **??**.

$$\beta = \frac{|\mathbf{p}_q + \mathbf{p}_{\overline{q}}|}{m_q + m_{\overline{q}}} = \frac{(x_1 - x_2)E}{(x_1 + x_2)E} = \frac{x_1 - x_2}{x_1 + x_2} \tag{2.35}$$

Setzen wir die gefunden Ausdrücke für s, η', und β in **??** ein, erhalten wir mit $Q^2 = 2x_1x_2E^2$ insgesamt für die differentiellen Wirkungsquerschnitt im Laborsystem **??**.

$$\frac{\mathrm{d}\sigma_p}{\mathrm{d}\eta}\left(x_1, x_2, Q^2, q\right) = \frac{Q_q^4 e^4}{96\pi Q^2}\left(1+\tanh^2\left(\eta + \frac{1}{2}\ln\left(\frac{x_2}{x_1}\right)\right)\right) \tag{2.36}$$

Setzen wir **??** rekursiv in **??**,**??** ein, erhalten wir insgesamt für den totalen und dreifach differentiellen Wirkungsquerschnitt des hadronischen Prozesses **??** und **??**.

$$\sigma = \sum_q \int \left[f_q(x_1, Q^2) f_{\overline{q}}(x_2, Q^2) + f_{\overline{q}}(x_1, Q^2) f_q(x_2, Q^2)\right] \frac{\mathrm{d}\sigma_p}{\mathrm{d}\eta} \mathrm{d}x_1 \mathrm{d}x_2 \mathrm{d}\eta \tag{2.37}$$

$$\frac{\mathrm{d}^3\sigma}{\mathrm{d}x_1 \mathrm{d}x_2 \mathrm{d}\eta} = \sum_q \left[f_q(x_1, Q^2) f_{\overline{q}}(x_2, Q^2) + f_{\overline{q}}(x_1, Q^2) f_q(x_2, Q^2)\right] \frac{\mathrm{d}\sigma_p}{\mathrm{d}\eta} \tag{2.38}$$

3. Maschinelles Lernen und tiefe neuronale Netzwerke

3.1. Motivation

Die Berechnung eines differentiellen Wirkungsquerschnitts eines Prozesses aus den zugrundeliegenden Feynman-Diagrammen, kann schnell sehr kompliziert werden. Oft sind diese Aufgaben analytisch nicht mehr lösbar, sodass numerische Methoden bemüht werden müssen. Diese fortgeschrittenen Methoden können in der Praxis sehr rechenaufwändig sein und viele Ressourcen beanspruchen. Algorithmen, die maschinelles Lernen verwenden, können je nach Typ und Komplexität jedoch sehr effizient und im Vergleich mit herkömmlichen numerischen Methoden signifikant schneller sein. Ein machine learning (ML)-Algorithmus ist zwar nicht in der Lage, den differentiellen Wirkungsquerschnitt numerisch aus den zugrundeliegenden Feynman-Diagrammen in erster Instanz zu berechnen, er kann die Funktion jedoch durch die Vorarbeit eines rechenaufwändigeren Algorithmus erlernen. Die praktische Anwendung hierbei liegt darin, mit resourcenfressenden numerischen Algorithmen zunächst eine ausreichende Anzahl an Punkten des differentiellen Wirkungsquerschnittes zu berechnen, mit diesen dann anschließende den ML-Algorithmus zu trainieren und im Endeffekt den ML-Algorithmus weiterzuverwenden, um eine größere Anzahl an Punkten zu berechnen.

Im Folgenden werden wir die Möglichkeiten eines solchen Einsatzes von ML-Algorithmen untersuchen und evaluieren. Wir beschränken uns dabei auf "überwachtes Lernen"von "Tiefen neuronalen Netzwerken".

3.2. Einführung in Maschinelles Lernen

Zunächst einmal ist der Begriff "Maschinelles Lernenßehr breit gefächert und kann eine Vielzahl an Dingen bedeuten. Allgemein wird hier lediglich aus Erfahrung, beispielsweise Messwerte, ein statistisches Modell entwickelt, das im Anschluss verallgemeinert werden kann. Der Algorithmus hat die Gesetzmäßigkeiten und Muster hinter den Lerndaten erkannt und kann diese nun anwenden. Die Anwendungsmöglichkeiten von statistischen Modellen sind quasi unbegrenzt, werden im Kontext von ML jedoch in zwei Gebiete unterteilt:

- Klassifizierung und
- Regression

Klassifizierung bezeichnet dabei den Vorgang, dass Eigenschaften oder "Featuresïn den Algorithmus gefüttert werden und dieser anschließend das Element, zu dem die Features gehören, eine bestimmte Gruppe zuweist. Die definierende Eigenschaft eines Elements, nach dem wir auch klassifizieren wollen wird auch "Label"genannt. Ein Beliebtes Beispiel hierfür ist die Kennzeichnung einer E-Mail mit dem Label SSpamöder "kein Spam", basiert auf dem "Bag of WordsModell, in dem jedes Wort in der E-Mail einem Feature der Nachricht entspricht. Regression ist die in unserem Fall nützlichere Verwendung von maschinellem Lernen. Während bei der Klassifikation die Ausgabe eine Gruppe ist, wird bei der Regression eine reelle Zahl ausgegeben. Prinzipiell wird lediglich eine Funktion, die gerne hochdimensional sein darf, erlernt. Ähnlich zu einer linearen Regression kann man so durch einige Messwerte eine Funktionsgleichung aufstellen und anschließend zwischen den Werten, oder auch außerhalb, interpolieren. Entsprechend soll die Regression für jede Art von Funktion funktionieren.

Es gibt einige Lernarten für ML-Algorithmen. Die Wichtigsten, die ich zumindest kurz erwähnen möchte, sind:

- Überwachtes Lernen (supervised learning)
- Unüberwachtes Lernen (unsupervised learning)

Das Unüberwachte Lernen findet hierbei seine Anwendung vor Allem in der Klassifikation. Hierbei werden nur die ungelabelten Features eingelesen, um anschließend von der Maschine eine Klassifikation entwickeln zu lassen. Der ML-Algorithmus soll also eine Möglichkeit entwickeln, die eingegeben Werte zu klassifizieren.

Für uns relevant ist das Überwachte lernen, wobei die Trainingsdaten mit Labels versehen sind. Explizit bedeutet das, dass die Eingabewerte wie η, x_1, x_2 die Features sind, die mit den Labels, dem Wrikungsquerschnitt $\frac{\mathrm{d}\sigma}{\mathrm{d}\eta}$, versehen sind. Je nachdem wie nah die Ausgabe der Maschine nun am richtigen Ergebnis liegt, kann der Algorithmus bestraft werden und so lernen.

Zur konkreten Durchführung von maschinellem Lernen gibt es verschiedene Modelle, zum Beispiel Lineare Regression, also das Minimieren der Abweichung einer Linearen Funktion von den Messwerten, logistische Regression, Entscheidungsbaum-Lernen, Support Vector Machines, etc. Die meisten überwachten Lernalgorithmen sind jedoch "flach", betreiben also ßhallowlearning. Hierbei wird die Ausgabe direkt aus den eingehenden Features berechnet. Die wohlbekannte Ausnahme bilden tiefe neuronale Netze, hierbei spricht man auch von "Deeplearning. Der Unterschied besteht hierbei dabei, dass aus den Eingangswerten zunächst sozusagen Zwischenergebnisse berechnet werden, die dann als Eingangswerte für die nächste Instanz des Modells dienen. Die einzelnen Instanzen, die jeweils Eingangswerte entgegennehmen und Zwischenwerte oder gegebenfalls auch Endwerte ausgeben, werden die Schichten des neuronalen Netzwerks genannt. Im Fall von neuronalen Netzen, besteht eine Schicht aus verschiedenen sogenannten Neuronen, deren Benennung von den Neuronen im biologischen Nervensystem inspiriert ist. Jedes Neuron stellt hierbei eine Funktion dar, die Ausgangswerte von der vorhergehenden Schicht entgegennimmt

und eine reelle Zahl zurückgibt. Diese reelle Zahl wird dann wiederum an die nächste Schicht von Neuronen weitergegeben. Klassischerweise würde die letzte Schicht dann nur noch ein Neuron enthalten, dessen Ausgabe dann das Label repräsentiert. Die am häufigsten genutzte Art von neuronalen Netzen sind vollständig verbundene Feedforward-Netze. Hierbei bedeutet vollständig verbunden, dass jedes Neuron der Schicht $l-1$ seine Ausgabe an jedes Neuron der Schicht l weitergibt. In anderen Arten von neuronalen Netzen kann es vorkommen, dass nur bestimmte Neuronen miteinander verbunden sind, beispielsweise könnte also ein Neuron der Schicht l nur die Hälfte aller Ausgaben der Neuronen der Schicht $l-1$ berücksichtigen. Ein anderes macht dann Gebrauch von der anderen Hälfte. Feedforward bedeutet, dass die einzelnen Schichten in einer festen Reihenfolge aufgebaut sind und es keine "Loopsöder ähnliches gibt. Auch wir werden uns in dieser Arbeit mit der gewöhnlichsten Art von Neuronalen Netzen beschäftigen.

Im folgenden beschäftigen wir uns also damit einen überwachten Regressionsalgorithmus mit tiefen Neuronalen Netzen zu entwickeln, der eine gegebenfalls hochdimensionale Funktion erlernen kann und damit die aufwändige numerische Berechnung von differentiellen Wirkungsquerschnitten effizienter machen kann.

3.3. Neuronale Netze

In **??** haben wir bereits erwähnt, was neuronale Netze sind und wie sie in etwa funktionieren. In diesem Abschnitt gehen wir noch einmal tiefer in die Theorie von neuronalen Netzen. In Abbildung ist ein prinzipieller Aufbau von einem sogenannten mehrschichtigen Perzeptron gezeigt. Hier sind alle Knoten von zwei aufeinanderfolgenden Schichten miteinander verbunden und die Layers sind in einer festen Reihenfolge angeordnet. Es handelt sich also um ein vollständig verbundenes Feedforward-Netz.

Die erste Schicht nimmt die Features unseres Messwertes entgegen und gibt die die gleiche Anzahl an Ausgangswerten, wie der Layer auch Neuronen hat, wieder aus. Jede Node/Neuron stellt hierbei eine lineare Funktion von den Anfangswerten da, die in der Praxis mittels einer Vektormultiplikation dargestellt werden kann:

$$y_l^n = \mathbf{w_{ln}} \cdot \mathbf{y_{l-1}} + b_{ln} \tag{3.1}$$

Hierbei bezeichnet l die Nummer des Layers und n ist der Index des jeweiligen Neurons innerhalb einer Schicht. Für den ersten Layer gilt $\mathbf{y_0} = \mathbf{x}$. Vektoren sind hierbei fettgedruckt (lieber irgendwo am Anfang hinschreiben). Hierbei bezeichnet w_{ln} die Gewicht oder "Weights"der Node und b_{ln} das sogenannte "Bias". In unserem vollständig verbundenen Netz erhalten wir also pro Node eine lineare Gleichung der Form **??**. Insgesamt können wir die Rechenoperation, die in einem Layer stattfindet also als Matrixmultiplikation formulieren. Die Vektoren $\mathbf{w_{ln}}$ werden hierbei zu den Zeilen der Matrix $\mathbf{W_l}$, die y_l^n sowie die b_{ln} fassen wir in Vektoren zusammen. In **??** ist die Matrixmultiplikation niedergeschrieben.

$$\mathbf{y_l} = \mathbf{M_l} \cdot \mathbf{y_l} + \mathbf{b_l} \tag{3.2}$$

Im letzten Layer findet der Output statt. Wie in den meisten Fällen, so kann auch in unserem Fall der letzte Layer als Schicht mit einem Neuron aufgefasst werden und der Funktionswert y dieses Neurons ist das Label unserer Anfangs eingefütterten Features. Aus der hier kurz aufgefassten Mathematik geht direkt hervor, dass neuronale Netze, die lediglich aus einem Layer bestehen, auch nur lineare statistische Modelle liefern. Ist die Funktion, die wir voraussagen wollen also nicht linear, sollten wir ein tieferes Neuronales Netzwerk verwenden, dass seine Nichtlinearität durch die aufeinandergestapelten Layers erhält. Sind die Gewichte der Layer $\mathbf{M_l}$ gut abgestimmt und ist das Netzwerk tief genug, kann man viele Funktionen sehr gut und effizient nähern. Dafür müssen wir behandeln, wie man denn gute Gewichte erhält. Dazu wird ein Optimierungsproblem gelöst. Konkret ist dieses Problem in **??** gezeigt.

$$C(\mathbf{M}, \mathbf{b}) = \frac{1}{N} \sum_{i=1}^{N} \left(y^{(i)} - \tilde{y}^{(i)} \right)^2 \tag{3.3}$$

Hierbei bezeichnet N die Menge unserer Trainingsdaten und $\tilde{y}$ sind die wahren Labels, die uns im Fall des überwachten Lernens ja zur Verfügung stehen. Die Funktion C wird in der Praxis Kostenfunktion ("Cost-Function") oder Verlustfunktion ("Loss-Function") genannt und ist eine hochdimensionale Funktion, abhängig von allen Gewichten $\mathbf{M}$ und Bias $\mathbf{b}$. In **??** ist die Kostenfunktion durch die mittlere quadratische Abweichung repräsentiert. Theoretisch kann als Kostenfunktion jedoch jede Funkion genommen werden, die die Abweichung zwischen den wahren Labels und den Predictions beschreibt. In der Anwendung nutzt man die Funktion, die im jeweiligen Anwendungsfall das beste Ergebnis erzielt. Das Minimum oder zumindest ein lokales Minimum dieser Funktion analytisch zu berechnen, ist nicht mehr möglich, daher haben sich über die Zeit eine Reihe an Möglichkeiten etabliert, sich einem Minimum zu Nähern. In der Praxis reicht es in fast allen Fällen ein lokales Minimum der Kostenfunktion zu finden. Die beliebteste Methode ist wohl der "Gradient-descent". Hierbei wird numerisch der Gradient der Kostenfunktion berechnet und dessen negative auf die trainierbaren Variablen des Modells angewendet. Der Gradient wird dabei jeweils als mittel über einen sogenannten Batch berechnet. In anderen Worten werden gleichzeitig mehrere Objekte durch das Netz geführt und am Ende die jeweiligen Gradienten gemittelt. Man hofft so sich mit jedem Batch dem lokalen oder sogar globalen Minimum der Kostenfunktion zu nähern. Für die praktische Durchführung haben sich in der Zeit verschiedene Etablierungen entwickelt, wie Adam, Stochastic Gradient Descent (SDG) oder RMSprop.

3.4. Training und Hyperparameter

Hat man sich nun auf ein vollständig verbundenes Feedforward-Netz festgelegt, gibt es immer noch viele Parameter, die optimiert werden müssen. Man nennt Parameter, die der Programmierer im Vornherein festlegen muss und die nicht vom Algorithmus erlernt werden, Hyperparameter. Wir sprechen im folgenden über die Hyperparameter:

- Anzahl der Layer und Nodes
- Kostenfunktion
- Aktivierungsfunktion der Neuronen
- Initialisierung der Gewichte
- Optimizer(Lernart) oder learning-rate(Lernrate)
- Batch-Größe
- Trainingsepochen
- Normalisierung
- Regularisierung
- Dropout

Zunächst sollte man sich Gedanken machen, wie die Struktur des Netzes aussehen soll. Man legt also die Anzahl der **Layer und Nodes** fest. Tiefere neuronale Netze mit größeren Anzahlen an Neuronen sind in der Lage kompliziertere Sachverhalte genauer zu lernen. Die Anzahl der trainierbaren Parameter nimmt jedoch exponentiell mit der Zahl der Schichten zu. Daher sollte man bestrebt sein, das vorliegende Problem mit möglichst wenigen Layern zu lösen. Ähnlich verhält es sich mit der Anzahl an Neuronen pro Layer. Ein gewöhnliches Problem, auf das man trifft, wenn man willkürlich zu große neuronale Netze für simple Probleme verwendet ist das der Überanpassung(siehe **??**). Das Netz passt sich hierbei zu stark an die Messdaten an, was am Ende dazu führt, dass sich die Generalisierungsfähigkeit des Netzes verschlechtert. Ein gewöhnliches Beispiel ist, wenn ein linearer Zusammenhang zwischen unseren Messwerte vorliegt, ein komplexes Netz darin jedoch ein Polynom hoher Ordnung sieht, welches eventuell die Trainingsdaten perfekt abdeckt, im Allgemeinen jedoch ungenauer wäre als ein simpleres lineares Modell.

Ein weiterer wichtiger Grundpfeiler eines ML-Algorithmus ist die bereits angesprochene **Kostenfunktion**. Beliebt sind hierbei die mittlere quadratische oder absolute Abweichung oder die Kreuzentropie. In high-level APIs wie scikit-learn oder Keras in Tensorflow sind noch weitere verschiedene Kostenfunktionen bereits integriert, von denen man dann nur noch diejenige finden muss, die im vorliegenden Fall am besten performed. Man hat sozusagen die Qual der Wahl.

Da jedes Neuron eine lineare Funktion der Aktivierungen der vorausgehenden Neuronen darstellt, ist es im vornherein nicht bestimmt in welchem Bereich der numerische Wert der Ausgabe der Neuronen liegt. Daher wird häufig die Ausgabe eines Neurons mit einer sogenannten **Aktivierungsfunktion** reguliert. Das hat den Vorteil, dass manche Neuronen nicht sehr große Werte nehmen und damit fast alleinig die endgültige Ausgabe bestimmen, sondern alle Neuronen "gleichberechtigt"von der darauffolgenden Schicht berücksichtigt werden. Beliebte Funktionen sind hier die

Sigmoid-Funktion, der tanh, oder ReLU. ReLU beispielsweise, setzt die Aktivierung eines Neurons auf Null, wenn dessen Ausgabe negativ ist, und gibt den ursprünglichen Funktionswert zurück, wenn die Aktivierung der Knoten positiv ist.

Irgendwo muss das Netz ja anfangen zu lernen, daher müssen seine Gewichte am Anfang irgendwie festgelegt(**initialisert**) werden. Auch hier haben sich über die Zeit verschiedene Möglichkeiten entwickelt, wie simpel alle Gewichte zunächst bei Null zu belassen, zufällige Werte nahe an Null zu nehmen, komplett zufällige Werte oder gar die Initialisierung an die Aktivierungsfunktion anzupassen. So ist zum Beispiel HeNormal an die Aktivierungsfunktion ReLU angepasst und soll zu besseren Ergebnissen führen.

Wie bereits angesprochen, wird zum Trainieren der neuronalen Netze eine Art des "gradient descent"verwendet. Am liebsten wird hierbei der Adam-Algorithmus verwendet. Jeder Art von gradient-descent muss jedoch eine **learning-rate** übergeben werden. Diese ist prinzipiell ein Faktor, mit dem der am Ende berechnete Gradient skaliert wird, bevor dieser auf die Gewichte angewendet wird. Intuitiv führt eine größere learning-rate zu schneller lernenden Netzen, kann jedoch andererseits auch dazu führen, dass man über die Minima hinausschießt und der Algorithmus im Endeffekt nicht konvergiert. Ein stabileres, aber rechenaufwändigeres und damit langsameres, Lernen erreicht man mit kleinen learning-rates.

Die **Batch-Größe** beschreibt, wie viele Objekte auf einen Schlag vom neuronalen Netz behandelt werden. In der Praxis fügt man jediglich dem Tensor der Eingangswerte eine weitere Dimension hinzu, die dann den Ausmaß der Batch-Größe hat. Das neuronale Netz kann dann alle Objekte parallel berechnen. Die Batch-Größe kann große Einflüsse auf die Trainingszeit und das Lernverhalten haben. Große Batch-Größen können schneller bearbeitet werden und führen zu kürzeren Trainingszeiten. Zu große Trainingszeiten verschlechtern die Stabilität des Lernvorgangs.

Die Anzahl **Trainingsepochen** beschreibt, wie oft während des Lernvorgangs über die Trainingsdaten gegangen wird. Nach einer Epoche werden die Trainingsdaten gegebenenfalls durchmischt und wieder eingefüttert. Es kann nun so lange nachkorrigiert werden, bis die Kostenfunktion nicht mehr merklich abnimmt und man ein lokales Minimum erreicht hat. Oft ist es nicht nötig sehr große Epochenzahlen zu verwenden, da dies meist keine großen Auswirkungen mehr hat und weiterhin zur Überanpassung führen kann. Praktisch beobachtet man nach jeder Epoche die Kostenfunktion angewendet auf die Trainingsdaten und auf die Testdaten. Hat man einen Punkt erreicht, ab dem der Verlust der Trainingsdaten zwar noch sinkt, der Verlust der Testdaten aber wieder steigt, ist das ein klares Zeichen für Überanpassung.

Hat man Eingabewerte, deren numerische Reichweite stark auseinandergeht, kann es sich lohnen die Eingabewert zu **normalisieren**. Das bedeutet, alle Features auf ein festgelegtes Intervall, zum Beispiel $I = [1, 0]$ anzupassen. So wird verhindert, dass einem Feature mit großem numerischen Wert nicht zu viel Bedeutung zugeordnet wird.

Ein Werkzeug zur Bekämpfung der schon besprochenen Überanpassung ist die **Regularisierung** des Modells. Diese soll die Komplexität des Modells moderieren,

indem das Optimierungsproblem modifiziert wird. Die modifizierte Version von ?? ist in ?? zu sehen. Die in ?? zu sehende Regularisierung wird L-Regularisierung genannt und $|\mathbf{M}|$ ist die p-Norm des Modells. Für gewöhnlich nutzt man die L1- oder L2-Regularisierung. Mit dem richtigen tuning der Parameter können mit tiefen neuronalen Netzen komplexe Zusammenhänge gut genähert werden ohne es zu Überanpassung kommen zu lassen. Wichtig ist hierbei, dass die Konstante L so zu wählen, dass die Genauigkeit des Algorithmus nicht auf Kosten der Regularisierung leidet. Sprich ist L zu groß, kümmert sich der Algorithmus nur noch darum, die Norm zu minimieren und findet im schlimmsten Fall ein lokales Minimum in dem alle Gewichte Null sind.

$$C(\mathbf{M}, \mathbf{b}) = \left[\mathrm{L}\|\mathbf{M}\|^p + \frac{1}{N} \sum_{i=1}^{N} \left(y^{(i)} - \tilde{y}^{(i)} \right)^2 \right] \quad \text{mit} \quad \|\mathbf{M}\|^p = \sum_{l,n} \|\mathbf{w}_{ln}\|^p \tag{3.4}$$

Eine weitere Möglichkeit zur Verhinderung von Überanpassung ist das Einfügen von Dropout-Layern. Diese setzen zufällig die Ausgaben von Neuronen auf Null und simulieren somit quasi einen Ausfall dieser Nodes. Dies soll die Generalisierungsfähigkeit des ML-Algorithmus erhöhen und Überanpassung verhindern.

3.5. Transfer-Learning

Das Trainieren von neuronalen Netzen kann teilweise sehr Kosten- und Zeitintensiv sein. Auch werden häufig große Mengen an Daten benötigt, um eine brauchbare Genauigkeit zu erhalten. Es ist jedoch möglich, bei beiden hier genannten Aspekten, eine Abkürzung zu nutzen und das Training signifikant effizienter zu machen. Diese Abkürzung nennt sich **Transfer-Learning**. Die Grundidee des Transfer-Learning ist denkbar einfach. Ähnlich zu einem Menschen, soll hierbei der ML-Algorithmus seine bereits erlerntes wissen, auf eine geringfügig andere Situation übertragen. Im Gegensatz zum Menschen, besitzt die Maschine jedoch kein Bewusstsein und ist nicht in der Lage, dies ohne Anweisung zu tun. Man muss den Algorithmus also noch mit Daten aus der neuen Situation trainieren. Trotzdem bringt das Transfer-Learning die beiden angesprochenen Vorteile:

- Höherer Start, höhere Asymptote und höhere Steigung
- signifikant weniger Messwerte benötigt, um brauchbare Ergebnisse zu erreichen

Das Transfer-Learning wird also vor allem eingesetzt, wenn man entweder einen Algorithmus trainieren möchte, der ansonsten rechenintensiv über längere Zeit trainieren müsste, oder über zu wenige Daten besitzt, um ein Modell von Grund auf zu trainieren. Wir profitieren von beiden Aspekten, denn wir können zwar theoretisch beliebig viele Punkt erzeugen und für jede Situation einen kompletten eigenständigen Algorithmus trainieren, es ist jedoch rechnerisch effizienter weniger Punkte zu erzeugen und zusätzlich nicht von Null zu beginnen. Konkret werden wir im Laufe dieser Thesis das Transfer-Learning verwenden, um die differentiellen Wirkungsquerschnitt

berechnet mit einem PDF-Set, auf selbige, berechnet mit einem anderen PDF-Set, zu übertragen.

Das Transfer-Learning wird bereichsübergreifend im maschinellen Lernen verwendet. Im folgenden werden wir kurz den konkreten Ablauf für neuronale Netze besprechen. Der gewöhnliche Ablauf beinhaltet dabei:

- Zunächst wird ein sogenanntes Source-Model an einer Source-Datenmenge bis zur Konvergenz trainiert.
- Als nächstes erstellt man eine (viel) kleinere Datenmenge an Zielwerten.
- Man entfernt die oberste oder einige der oberen Schichten (sprich der Output-Layer und wenige darunterliegende Layer)
- Die Gewichte der restlichen Layer werden zunächst eingefroren, um nicht durch große Gradienten zerstört zu werden
- Wir ersetzen die entfernten Schichten mit neuen, trainierbaren Neuronen
- Schließlich trainieren wir das neue Modell an unserer kleinern Datenmenge
- Optional kommt als letztes das sogenannte Fine-Tuning, bei dem die eingefrorenen Gewichte wieder aufgetaut werden

Das Fine-Tuning kann dabei essentiell sein, um nocheinmal bedeutende Verbesserungen zu bewirken. Man sollte jedoch bei kleinen learning-rates bleiben.

3.6. Implementierung mit Keras und Tensorflow

Die Implementierung des ML-Algorithmus wird in dieser Arbeit mit der open-source Python-Bibliothek TensorFlow und Keras stattfinden. Keras fungiert hierbei als eine high-level API für TensorFlow. Keras ist hierbei schon so weit entwickelt, dass man gut funktionierende ML-Algorithmen in wenigen Zeilen schreiben kann. Man muss prinzipiell nur die Architektur des neuronalen Netzes eintragen, Werte für Optimizer und Loss festlegen und die Trainingsdaten in der passenden Form präparieren. Es ist mit Keras jedoch auch möglich, auf eine tieferes Level zu gehen und die Layer-Klassen selbst zu schreiben. So ist man flexibler und kann direkt nachvollziehen, was in in jeder Schicht passiert. Entscheidet man sich Custom-Layer zu benutzen, wird auch der Trainingsalgorithmus selbst geschrieben. Selbst wenn man sich jedoch auf der Low-Level-Ebene von Keras befindet, kann man trotzdem noch von den Optimizern und Kostenfunktionen, die bereits in TensorFlow eingebunden sind, profitieren. Ein Beispiel für einen einfachen ML-Algorithmus mit der high-level API Keras ist in **??** gezeigt. Die Schwierigkeit besteht also primär nicht darin, ein tiefes neuronales Netzwerk zu bauen und zu trainieren, sondern im Tuning der Hyperparameter. Diese müssen, wie bereits erwähnt, vom Programmierer im vornherein festgelegt werden und können nicht vom Algotihmus erlernt werden. Es gibt leider keine Methode,

immer die perfekten Hyperparameter zu finden, über die Zeit haben sich jedoch die Möglichkeiten der Grid-Search und des Random-Search durchgesetzt. Bei ersterem wird hierbei systematisch immer ein Hyperparameter in einem festgelegten Intervall variiert und der Lernvorgang durchgeführt. Genauso variiert man alle Hyperparameter, die man Anpassen möchte und erstellt so ein "Gitter". Am Ende wählt man die Hyperparameter, die das beste Ergebnis geliefert haben. Da es vergleichsweise viele Hyperparameter gibt, die man festlegen muss, kann das schnell dazu führen, dass der Trainingsvorgang sehr oft durchgeführt werden muss und damit die Suche sehr rechenaufwändig wird. In der Praxis wird also häufig der Random-Search bevorzugt, bei dem alle Hyperparameter vor jedem Durchlauf einen zufälligen Wert aus einem vorher festgesetzten Pool annehmen. Man sucht nun so lange, bis man ein zufriedenstellendes Ergebnis erreicht hat.

3.7. Monte-Carlo-Integration

In der Physik, insbesondere der Teilchenphysik, ist es oft nötig, die Integrale von komplexen Funktionen auszuwerten. So auch in unserem Fall, wenn wir den totalen Wirkungsquerschnitt aus den differentiellen erhalten wollen. Oft genug, sind diese Integrale nicht mehr analytisch lösbar und es bedarf numerischen Methoden, um ein Ergebnis zu erhalten. Herkömmliche Integrationsmethoden weisen dabei, bezogen auf ihre Konvergenz, generell eine Abhängigkeit der Dimensionalität des Integrals auf. Hier kommt der Große Vorteil der Monte-Carlo-Integration (MC-Integration) ins Spiel, denn diese Methode konvergiert immer mit einer Geschwindigkeit von $\frac{1}{N}$, wobei N die Anzahl an ausgewerteten Funktionspunkten ist. Monte-Carlo-Methoden machen hierbei Gebrauch von dem Gesetz der Großen Zahlen und lösen die Integrale mittels Wahrscheinlichkeitstheorie.

Wir betrachten eine Funktion $f : \Omega \subseteq \mathbb{R}^n \to \mathbb{R}, x \mapsto f(x)$ und definieren ihren Erwartungswert μ auf Ω als **??**.

$$\langle f \rangle = \frac{1}{\|\Omega\|} \int_\Omega f(x) \mathrm{d}x \tag{3.5}$$

Wir wenden nun das Gesetz der Großen Zahlen an und finden somit einen Schätzer für den Erwartungswert von f (**??**).

$$\langle \tilde{f} \rangle = \frac{1}{N} \sum_{i=1}^{N} f(x_i) \quad \text{mit} \quad \lim_{N\to\infty} \frac{1}{N} \sum_{i=1}^{N} f(x_i) = \mu(f) \tag{3.6}$$

Legen wir also ein unendlich dichtes Gitter und evaluieren f an jeder Stelle des Gitters, können wir einen exakten Ausdruck für das Integral aus **??** finden. Da dies praktisch nicht möglich ist, wird oft der Schätzer als genäherten Wert für den Erwartungswert $\tilde{\mu} \approx \mu$ verwendet und das Integral näherungsweise berechnet. In MC-Simulationen werden nun die Funktionswerte x_i zufällig gezogen, da die Anzahl an

benötigten Punkten für ein dicht liegendes Gitter exponentiell mit der Dimensionalität des Integrals ansteigt. Wir können die Geschwindigkeit der Konvergenz unserer Näherung erhöhen, wenn wir in **??** eine produktive Eins in Form einer Wahrscheinlichkeitsdichte $\rho : \Omega \subseteq \mathbb{R}^n \to \mathbb{R}_{\geq 0}, x \mapsto \rho(x)$ mit $\int_\Omega \rho(x)\mathrm{d}x = 1$ einführen (**??**)

$$I = \int_\Omega f(x)\mathrm{d}x = \int_\Omega \frac{f(x)}{\rho(x)}\rho(x)\mathrm{d}x = \mu_\rho\left(\frac{f}{\rho}\right) \tag{3.7}$$

Dabei stellt $\mu_\rho\left(\frac{f}{\rho}\right)$ den Erwartungswert von $\frac{f}{q}$ unter der Bedingung dar, dass die x_i nach der Wahrscheinlichkeitsverteilung $\rho(x)$ gezogen werden. Der Schätzer ergibt sich dann zu **??**.

$$I \approx \tilde{\mu}_\rho\left(\frac{f}{\rho}\right) = \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{\rho(x_i)} \tag{3.8}$$

Die Konvergenz der MC-Simulation ist am Schnellsten, wenn sich die Varianz von **??** minimiert. Die Varianz ist gegeben durch **??**

$$\mathrm{Var}\left(\frac{f}{\rho}\right) = \left\langle\left(\frac{f}{\rho} - \left\langle\frac{f}{\rho}\right\rangle\right)^2\right\rangle = \left\langle\left(\frac{f}{\rho}\right)^2\right\rangle - \left\langle\frac{f}{\rho}\right\rangle^2 = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{f(x_i)}{\rho(x_i)}\right)^2 - I^2 \tag{3.9}$$

Die Varianz minimiert sich also, wenn jeder Summand aus **??** gleich groß ist. Der Vorgang die Wahrscheinlichkeitsdichte ρ an die Form unserer zu integrierenden Funktion f anzupassen, nennt man **Importance Sampling**. Hierbei zieht man absichtlich die x_i mit höheren Wahrscheinlichkeiten aus den Regionen, in denen auch f den größten Beitrag liefert.

Wir werden im Folgenden simple Monte-Carlo-Methoden und das Importance Sampling verwenden, um aus unseren differentiellen Wirkungsquerschnitten die totalen Wirkungsquerschnitte zu erhalten. Man kann die Varianz und damit die Konvergenz der Simulation noch weiter verbessern, indem man beispielsweise das sogenannte **Stratified Sampling** mit dem Importance Sampling kombiniert. Beim Stratified Sampling wird Ω in l Teilmengen unterteilt und aus jeder Teilmenge eine festgelegte Anzahl n an x_i gezogen. Der Effekt des Stratified Sampling ist am Größten, wenn jede Teilmenge die gleiche Varianz liefert.

4. Anwendung von Maschinellem auf den Diphoton Prozess

4.1. Partonischer Diphoton-Prozess

4.1.1. Modell und Hyperparameter

Wir beginnen damit den ML-Algorithmus auf das einfache Beispiel des partonischen Diphoton-Prozess aus **??** anzuwenden. Wie wir gesehen haben, lässt sich der Wirkungsquerschnitt auch prima analytisch berechnen. Die Näherung mit machinelearning dient in diesem Falle also eher dem Lerneffekt und der Veranschaulichung. Da wir den analytischen Ausdruck des differtiellen Wirkungsquerschnitts kennen (siehe **??** und **??**), können wir uns unsere Trainings- und Testdaten sehr simpel selbst generieren. Wir nutzen hierbei die Python-Bibliothek Pandas, um die generierten Arrays für $\frac{\mathrm{d}\sigma}{\mathrm{d}\theta}$ und $\frac{\mathrm{d}\sigma}{\mathrm{d}\eta}$ abzuspeichern und einzulesen. Wir trainieren den ML-Algorithmus mit ca. 60000 Daten in geeigneten Wertebereichen. Explizit sind das für θ der Bereich $[\epsilon, \pi - \epsilon]$ und für η der Bereich $[-3, 3]$. Wir werden die Performance der Modelle für verschiedene ϵ evaluieren, da die analytische Funktion an den Rändern des Intervalls für $\epsilon = 0$ Polstellen hat und eine Ausgabe des Netzes von ∞ allein schon konzeptionell nicht möglich ist und sich auch nicht mit der numerischen Natur von Computern verträgt. Was die Architektur des neuronalen Netzes angeht, entscheiden wir uns für ein simples Netz mit einer bestimmten Anzahl an hidden Layers mit der gleichen Anzahl an Neuronen.

Modell für $\frac{\mathrm{d}\sigma}{\mathrm{d}\eta}$:
Der differentielle Wirkungsquerschnitt in Abhängigkeit der Pseudo-Rapidität ist eine sehr gutartige Funktion ohne Pol- oder Sprungstellen oder Ähnliches. **??** reduziert sich, bei Vernachlässigung von Vorfaktoren und Verschiebungen, von der Komplexität auf einen $tanh^2$, dessen Wertebereich sich über $[0, 1)$ erstreckt und damit schon von vornherein normiert ist. Wir behandeln den Vorfaktor mit einer Skalierung der Funktionswerte, auf die wir später noch weiter eingehen werden. Für diese vergleichsweise einfache Aufgabe können wir simpel die Hyperparameter raten und das Ergebnis auswerten. Wir wählen die in **??** gezeigten Werte. Im Folgenden werden wir nicht zwischen den Hyperparametern, die die Architektur und ähnliches des Netzes bestimmen und den Trainingsparametern, die das Training beeinflussen, differenzieren. Das Training an sich wird von den in Keras leicht einzubauenden Callbacks bestimmt. Wir werden im folgenden die Callbacks verwenden:

- **LearningRateScheduler**: Ein Ablaufplan wird festgelegt, der für jede Epoche

Hyperparameter	Wert
Anzahl Layer	2
Anzahl Units	64
Loss-Funktion	Mean-Absolute-Error
Optimizer	Adam
Aktivierungsfunktion	ReLU
Kernel-Initializer	HeNormal
Bias-Initializer	Zeros
Learning-rate	0.005
Batch-Größe	128
Max. Epochen	300
Anzahl Trainingspunkte	10000

Tabelle 4.1.: Hyperparameter des Modells $\frac{d\sigma}{d\eta}$

die zu verwendende Learning-Rate bestimmt.

- **ReduceLROnPlateau**: Erzielt das Training bezogen auf eine bestimmte Metrik nicht einen gewissen Fortschritt, wird die Learning-Rate reduziert.

- **EarlyStopping**: Erzielt das Training bezogen auf eine bestimmte Metrik für eine gewisse Zeit keinen Mindestfortschritt, wird das Training gestoppt.

Die Wahl der genauen Konfiguration der Callbacks ist in **??** festgehalten. Die gelernte Funktion im Vergleich mit den analytischen Werten ist in **??** gezeigt. Die Werte überlagern sich recht gut, sodass man auf den ersten Blick keinen Unterschied feststellen kann. Betrachtet man das Verhältnis, erkennt man dass sich der Unterschied auf ca. 0.1%. Diese Genauigkeit ist mit den hier verwendeten State-of-the-Art Hyperparametern für das einfache Problem auch zu erwarten.

Modell für $\frac{\mathrm{d}\sigma}{\mathrm{d}\theta}$:
Der Wirkungsquerschnitt in Abhängigkeit von θ unterscheidet sich vom vorherigen Modell durch seine Polstellen. Da Computer schlecht mit Polstellen umgehen können, müssen wir den Trainingsbereich auf $[\epsilon, \pi - \epsilon]$ einschränken. Aus physikalischer Sicht ist das legitim, da die Polstellen im Strahlengang des Speicherrings liegen und damit nicht messbar sind. Viele Detektoren können Pseudo-Rapiditäten bis zu $|\eta| \leq 2.5$ messen, was einem $\epsilon \approx 0.163$ entspricht. Man kann dem Modell den Umgang mit den Polstellen erleichtern, in dem man die Labels(also den differentiellen Wirkungsquerschnitt) auf das Intervall $[-1, 1]$ normiert. Da gute Modelle hier nicht mehr trivial gefunden werden können, greifen wir auf eine automatische, zufällige Suche zurück (Random-Search), um nicht einzelne Hyperparameter per Hand ausprobieren zu müssen. Die Such-Parameter mit Ergebnis sind in **??** festgehalten. Es fällt auf, dass die Architektur des Modells um ein vielfaches komplizierter ist, als die vorhergehende. Einerseits ist dies aufgrund der Polstellen zu erwarten und andererseits ist

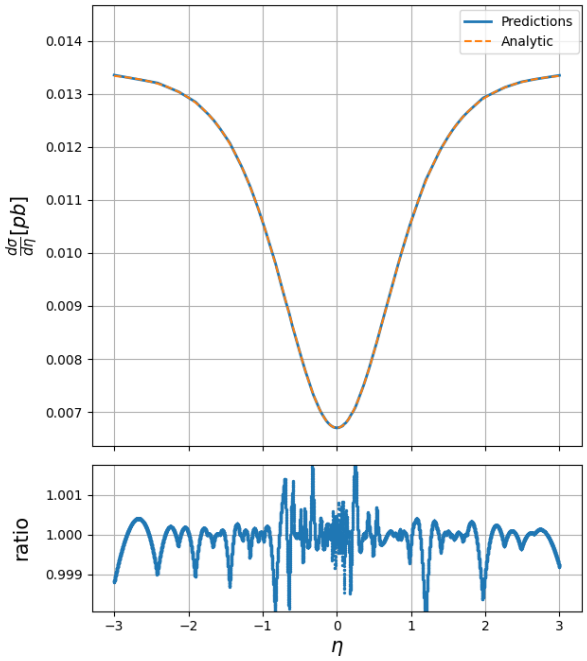


Abbildung 4.1.: machine learning predictions vs analtisch berechnete Werte Eta

Hyperparameter	Pool	Best Config
Anzahl Layer	$\{1, 2, 3, 4\}$	4
Anzahl Units	$\{32, 64, 128, 256\}$	128
Loss-Funktion	MAE, MSE, Huber	MAE
Optimizer	Adam, RMSprop, SGD	Adam
Aktivierungsfunktion	ReLU, Leaky-ReLU, Sigmoid	Leaky-ReLU
Learning-rate	$\{10^{-2}, 5 \cdot 10^{-3}, 10^{-3}, 10^{-4}\}$	$5 \cdot 10^{-3}$
Batch-Größe	$\{64, 128, 512, 768, 2048\}$	128
Label-Normalisierung	{keine, $[-1, 1]$}	$[-1, 1]$
Max. Epochen	200	
Anzahl Trainingspunkte	60000	

Tabelle 4.2.: Parameter der Random-Search für $\frac{d\sigma}{d\theta}$ mit Ergebnis

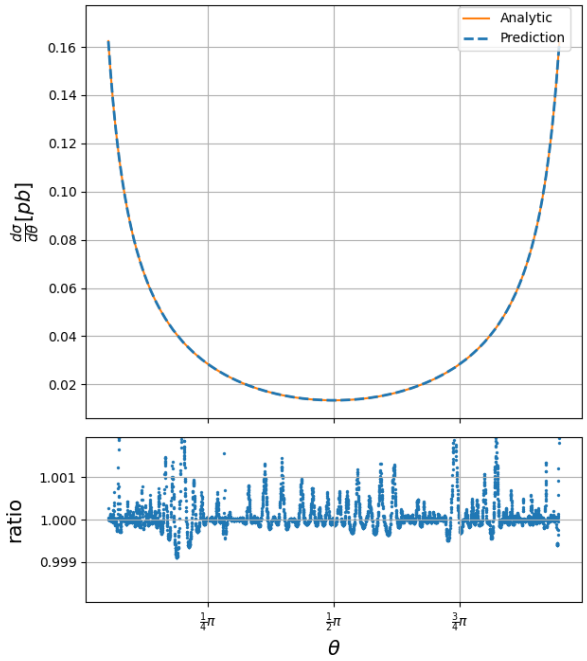


Abbildung 4.2.: Predictions vs Analytisch auf relevantem Theta-Intervall

die Möglichkeit des Overfitten durch die große Anzahl an Trainingspunkten weitgehend ausgeschlossen und daher nur natürlich, dass komplexere Modelle genauere Ergebnisse erzielen. Die Performance des Modells ist in **??** gezeigt. Die Präzision ist trotz der komplizierteren Funktion mit **??** zu vergleichen. Durch den Random-Search konnte also ein vergleichsweise passendes Modell gefunden werden.

Wir wollen nun betrachten, wie sich das Modell auf einem Interval $[\epsilon', \pi - \epsilon']$ mit $\epsilon' < \epsilon$ schlägt. Wir vergleichen dies mit einem Modell, dass zwar mit den gleichen Hyperparametern, jedoch auf $[\epsilon', \pi - \epsilon']$ trainiert wurde. Für ein drittes Modell sind die Trainingsdaten nach einer Verteilung generiert, die der Form von $\frac{d\sigma}{d\theta}$ ähnelt (Importance Sampling). Die Vergleiche sind in **??** und in **??** für $\epsilon' = 0.01$ gezeigt. Wie zu erwarten weicht das ursprüngliche schnell von der analytischen Funktion ab. Man erkennt, dass dem Modell zwar die Tendenz bekannt ist, der genaue Verlauf jedoch rasch unbekannt wird. Man könnte vermuten, dass der Maschine zwar die Steigung bekannt ist, alle weiteren Ableitungen jedoch die Komplexität des Modells übertreffen. Die beiden anderen Modelle zeigen akzeptable Leistung auch nahe an den Polstellen. In **??** lässt sich schlecht beurteilen, ob das importance Sampling Wirkung zeigt. Lediglich im Verhältnis kann man erahnen, dass das mit importance gesampelten Trainingsdaten trainierte Netz an den Polstellen besser und im Zentrum schlechter angepasst ist. In **??** a) wird diese Vermutung bestätigt, auch wenn die Auswirkungen nur vergleichsweise klein sind. Einen größeren Effekt sieht man in **??** b). Durch die große Zahl an Trainingsdaten in a) sind schon genug Punkte nahe

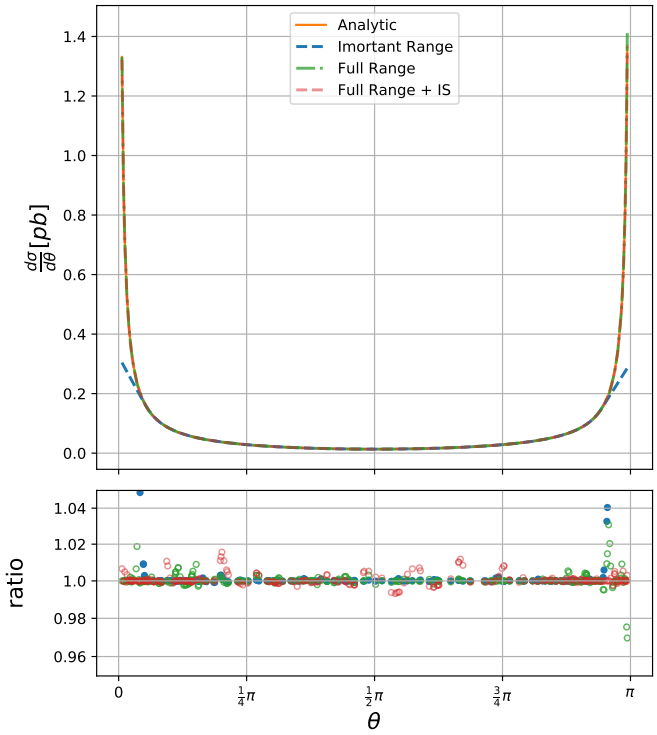
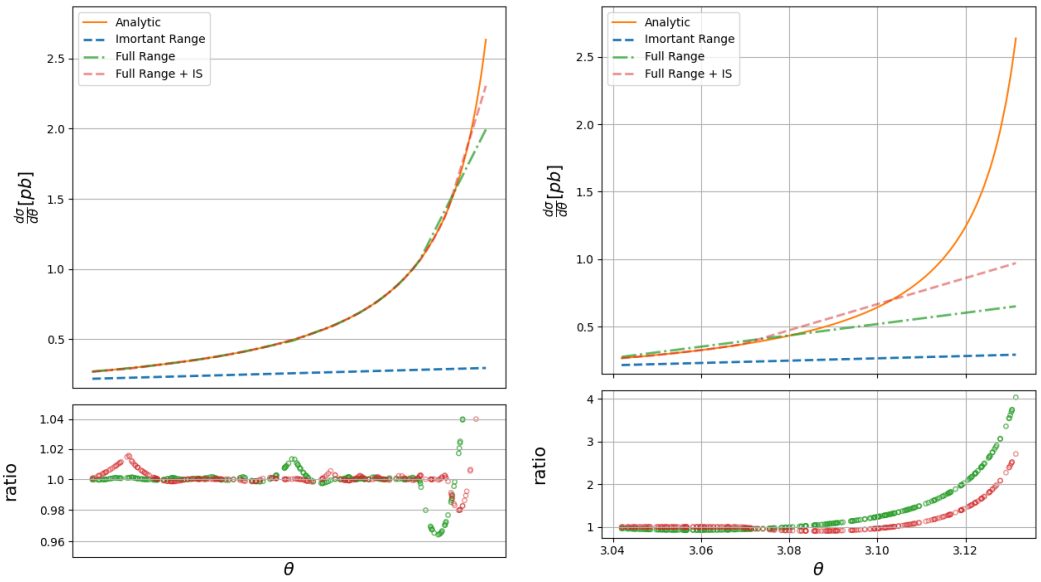


Abbildung 4.3.: machine learning predictions vs analtisch berechnete Werte

an der Polstelle vorhanden und die zusätzlichen Werte bringen nur einen kleinen (absoluten) Mehrwert. Ist man jedoch begrenzt in seiner Verfügung über Trainingsdaten oder möchte die Trainigszeit minimieren und trotzdem brauchbare Ergebnisse erhalten, kann importance sampling jedoch helfen. Man sollte jedoch im Hinterkopf behalten, dass man hierbei einen Kompromiss eingeht und die Verlässlichkeit in den Bereichen, die durch das sampling vernachlässigt werden, abnimmt. In **??** sind noch einmal der MAPE (Mean-Absolute-Percentage-Error) der verschiedenen Modell für verschiende Testdatensets gezeigt. Hier wird noch einmal deutlich, dass das importance sampling vor allem nützlich ist, wenn die Bereiche von Funktionen besonders wichtig sind, in denen die Funktion auch einen hohen Funktionswert besitzt. Da wir den differentiellen Wirkungsquerschnitt letztendlich benutzen wollen, um den totalen Wirkungsquerschnitt zu berechnen, ist dies für uns genau der Fall. Allgemein geht das Annähern eines beliebigen Integranden mittels maschinellem Lernen Hand in Hand mit anschließender Monte-Carlo Integration. Die Verteilung, die wir benutzen, um die Form des Wirkungsquerschnittes anzunähern, ist die ein Polynom vierten Grades und in **??** gezeigt.



(a) Modelle mit 60000 Traningspunkten (b) Modelle mit 10000 Trainingspunkten

Abbildung 4.4.: Performance des Netzes für Randpunkte

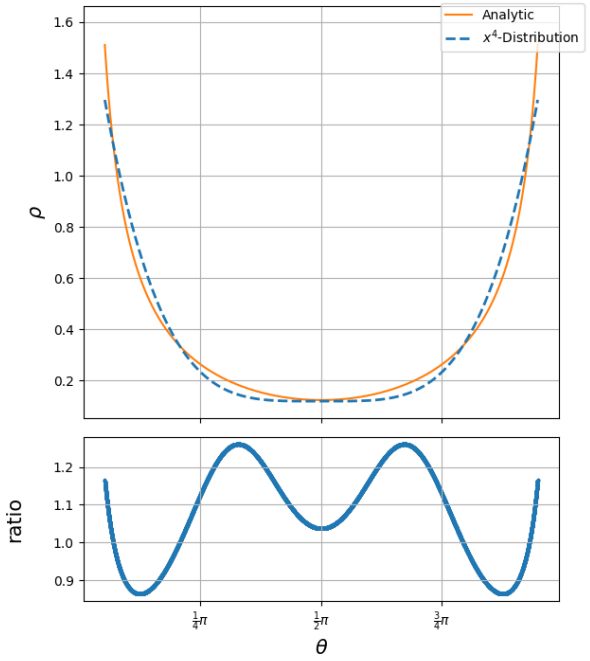


Abbildung 4.5.: simples Importance Sampling, das die analytische Funktion annähern soll

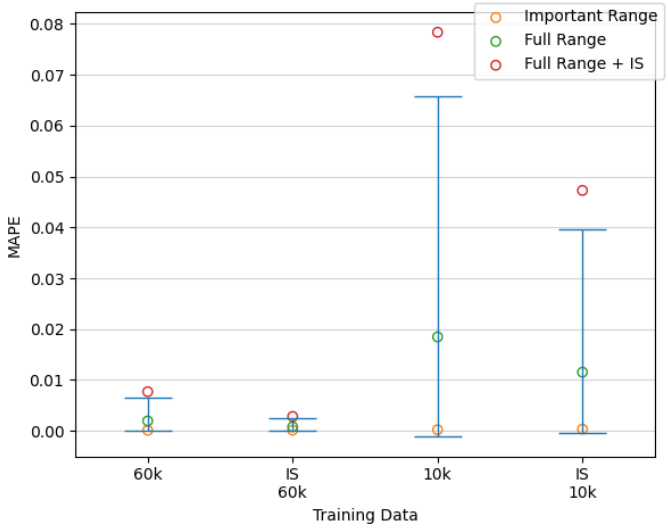


Abbildung 4.6.: Vergleich der Performance von verschiedenen Theta-Modellen, die mit verschiedenen Datenmengen trainiert wurden. IS: Importance sampling. Auf der x-Achse sind die Trainings-Datenmengen und in der Legende die Test-Datenmengen

Bereich	Cut
Photon-Energie	$\lvert p_T\rvert > 40$ GeV
Photon Winkel	$\lvert\eta_{\gamma,\tilde{\gamma}}\rvert < 2.37$ ohne ¸$1.37 < \lvert\eta_{\gamma,\tilde{\gamma}}\rvert < 1.52$
Impulsbruchteil	$x_{1,2} < 0.7$

Tabelle 4.3.: Event-Selektion für den Diphoton-Prozess in Leading-Order am ATLAS - Detektor

4.2. Hadronischer Diphoton-Prozess

4.2.1. cuts and stuff

Im Gegensatz zu den im Vorhergehenden betrachteten Prozessen, ist die Reaktion $pp \to \gamma\gamma$ beobachtbar und messbar. Wir wollen den Wirkungsquerschnitt am Beispiel einer Messung im ATLAS-Detektor behandeln. Da kein Detektor perfekt ist, müssen wir die Events nach messbaren und nicht-detektierbaren Ereignissen einteilen. Wir wenden also Cuts auf unsere generierten Phasenraumpunkte an und trainieren unseren Algorithmus nur an solchen Messpunkten, die auch praktisch detektierbar wären. Die verwendeten Cuts sind angelehnt an **??** und aufgelistet in **??**. Dabei sind γ und $\tilde{\gamma}$ die Bezeichnungen für die beiden Photonen. p_T bezeichnet dabei den Impuls der produzierten Photonen transversal zum Strahlengang. Da die Quarks, die im Prozess beteiligt sind in unserem Modell nur einen Impuls in Strahlrichtung haben, folgt aus der Impulserhaltung direkt $p_{T,\gamma} = p_{T,\tilde{\gamma}}$. ATLAS kann keine beliebig spitzen

Winkel messen, daher betrachten wir nur Photonen mit einer Pseudo-Rapidität von $\eta < 2.37$. Der Detektor besteht aus zwei Teilen, wobei sich der eine Teil wie ein Zylindermantel um den Strahl legt und der andere die Deckel darstellt. Zwischen diesen Teilen befindet sich ein Spalt, in dem nicht gemessen werden kann, daher verwerfen wir auch Ereignisse mit $1.37 < |\eta_{\gamma,\tilde{\gamma}}| < 1.52$. Wie wir später sehen werden, machen die Partondichtefunktionen und damit auch der dreifach differentielle Wirkungsquerschnitt ab ca. $x \approx 0.7$ einen Bogen und fällt extrem schnell zu Null hin ab. Zum totalen Wirkunsquerschnitt, der letztendlich unser Ziel darstellt, trägt dieser Bereich so gut wie nicht mehr bei. Weiterhin vereinfacht es es dem neuronalen Netz extrem, wenn er diesen Phasenraumbereich mit extrem kleinen Labels nicht mehr erlernen muss.
Wir müssen weiterhin beachten, dass wir beide Photonen messen können müssen, damit wir den Prozess identifizieren können. Da wir im Schwerpunktsystem der Protonen messen, unterscheiden sich die Pseudo-Rapiditäten der beiden Photonen. Wir müssen die Cuts in η also für beide Photonen sicherstellen. Messen wir sowohl η_γ als auch $\eta_{\tilde{\gamma}}$ in Bewegungsrichtung des Quarks mit Impulsbruchteil x_1, berechnet sich η_γ aus dem η' der Photonen im Schwerpunktsystem der Quarks nach **??**.

$$\eta_\gamma = \eta' - \frac{1}{2}\ln\left(\frac{x_2}{x_1}\right) \quad \text{sowie} \quad \eta_{\tilde{\gamma}} = -\eta' - \frac{1}{2}\ln\left(\frac{x_2}{x_1}\right) \tag{4.1}$$

Intuitiver ist es jedoch, wenn $\eta_\gamma = \eta_{\tilde{\gamma}} frx_1 = x_2$ gelten würde, anstatt $\eta_\gamma = -\eta_{\tilde{\gamma}}$, sprich wenn wir $\eta_{\tilde{\gamma}}$ in Bewegungsrichtung von x_2 messen würden. Dabei transformiert sich $\eta_{\tilde{\gamma}} \to -\eta_{\tilde{\gamma}}$ und wir finden **??**.

$$\eta_{\tilde{\gamma}} = \eta' + \frac{1}{2}\ln\left(\frac{x_2}{x_1}\right) \quad \Rightarrow \quad \eta_{\tilde{\gamma}} = \eta_\gamma + \frac{1}{2}\ln\left(\frac{x_2^2}{x_1^2}\right) \tag{4.2}$$

4.2.2. Suchprozess

Den Integranden den wir nun erlernen möchten (**??**) ist nun nicht mehr eindimensional, sondern dreidimensional. Dieser Unterschied fällt zwar zunächst als erstes auf, es ist jedoch ein anderer Faktor, der das neuronale Netz stärker beeinflusst. Die Partondichtefunktionen, die den dreidimensionalen Wirkungsquerschnitt bestimmen, fallen exponentiell mit ihren Impulsbruchteilen x_1 und x_2 ab und besitzen Polstellen für $x \to 0$. Damit man numerisch mit den Partondichtefunktionen auch noch an der Stelle Null arbeiten kann, werden diese ab einem gewissen $x_{min} \approx 10^{-9}]$ eingefroren. Praktisch heißt das jetzt, dass sich unsere Labels zwischen ca 30 Größenordnungen bewegen. Mit derartigen Veränderungen können neuronale Netze nicht arbeiten, da diese Formen allein schon konzeptionell nicht zu erreichen sind. Das wird intuitiv, wenn man sich das neuronale Netz als eine Reihe an hintereinander ausgeführten Matrixmultiplikationen vorstellt, die nur ihre Linearität dank den Aktivierungsfunktionen verlieren. Die Sensitivität auf kleine Veränderungen in x ist so nur begrenzt zu reproduzieren. Hinzu kommt, dass eine gängige Loss-Funktion wie der Mean-Squared- oder Mean-Absolute-Error offensichtlich nur Punkte mit hohen Wirkungsquerschnitten berücksichtigen und damit die Ergebnisse schon ab einem kleinen x

unbrauchbar werden würden. Abhilfe kann hierbei die oft für ähnliche Probleme verwendete Loss-Funktion „Mean-Squared-Logarithmic-Error(MSLE)“ schaffen (siehe **??**). Wir sehen, dass es beim MSLE nicht mehr auf die Größe der Abweichung ankommt, sondern das Verhältnis der Werte. Damit ist gesichert, dass keine Bereiche des Phasenraumes komplett vernachlässigt werden. Um nicht mit negativen Werten zu arbeiten, transformiert man meistens $y \to y + 1$. Diese Form hilft uns jedoch nicht weiter, da der Großteil unserer numerischen Werten in herkömmlich verwendeten Einheiten sprich $1/\mathrm{GeV}^2$ und pb viel kleiner als eins ist. Bemühen wir die Taylor-Entwicklung des Logarithmus um eins, fällt auf, dass $\ln(1+x) \approx x$ gilt und wir somit effektiv nur einen ineffizienteren Mean-Squared-Error implementieren.

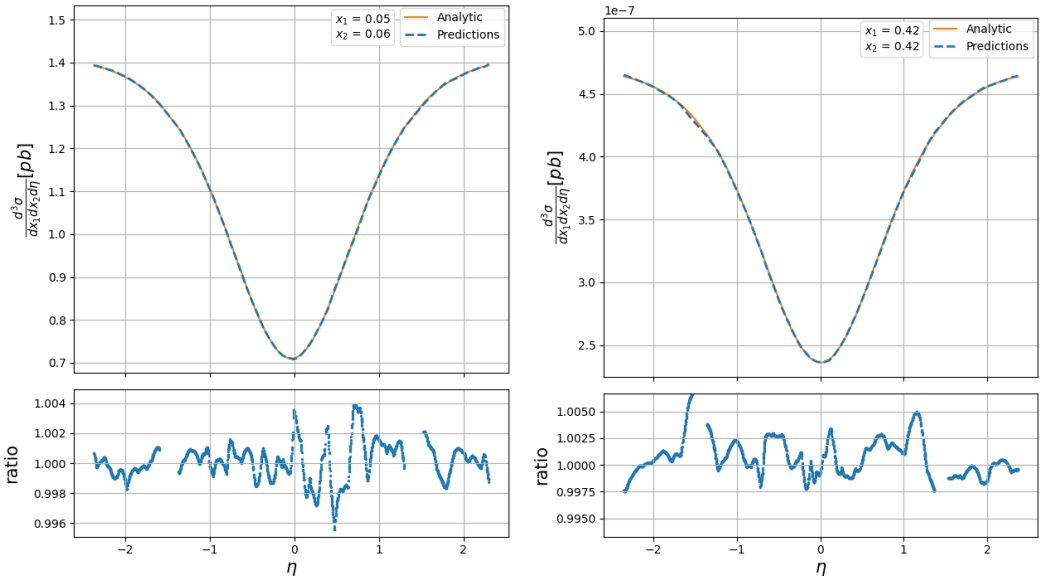
$$C\left(\mathbf{M}, \mathbf{b}\right) = \frac{1}{N} \sum_{i=1}^{N} \left(\ln\left(y^{(i)}\right) - \ln\left(\tilde{y}^{(i)}\right) \right)^2 = \frac{1}{N} \sum_{i=1}^{N} \left(\ln\left(\frac{y^{(i)})}{\tilde{y}^{(i)}} \right) \right)^2 \tag{4.3}$$

Hier kommt die Skalierung, die wir in den beiden vorhergehenden Modellen bereits verwendet haben, ins Spiel. Die Skalierung beseitigt außerdem das „Dying-ReLU“-Problem, dass im Zusammenhang mit numerisch kleinen Labels auftreten kann. Es geht hierbei um einen großen Gradienten nach Überschätzung der Funktionswerte seitens des Netzes, der auf die Neuronen angewendet wird und die Parameter so verändert, dass das Neuron in der Zukunft nur noch Null zurückgeben wird. Da im Bereich der ReLU unter Null auch die Ableitung der Loss-Funktion Null ist, kann sich die Node somit nicht regenerieren. Das passiert bei sehr kleinen Labels schon nach den ersten Batches, da die durch Initialisierung der Gewichte der Layer, das Modell zu Beginn des Trainings einen mehr oder weniger zufälligen Wert zurückgibt. Man müsste also die Initialisierung der Gewichte perfekt auf die vorliegenden Daten abstimmen, falls das überhaupt im vorliegenden Fall möglich ist. Es ist also naheliegend auf die Skalierung der Funktionswerte zurückzugreifen. Praktisch wird das bei uns durch einen Transformator-Objekt realisiert, der die vorliegenden Labels so skaliert, dass der kleinste Funktionswert auf eins abgebildet wird. Die Skalierungskonstante hängt offensichtlich von den vorliegenden Phasenraumpunkten ab und ist damit von Datenset zu Datenset unterschiedlich. Daher erstellen wir im folgenden für jeden für jedes Modell einen zugehörigen Transformator, der nach dem Training mit dem Modell abgespeichert wird und bei Bedarf wieder initialisiert werden kann. Die Skalierungskonstante ist wichtig, damit wir später unabhängig vom Trainingsdatenset die Predictions unseres Modells auf die wirklichen Werte zurückskalieren können. Weitere Möglichkeiten um mit dem „Dying-ReLU“-Problem umzugehen sind die Verwendung von Aktivierungsfunktionen mit nicht-verschwindender Ableitung wie Leaky-ReLU oder ELU, die den Nodes ermöglichen soll, sich zu regenerieren, oder die Übergabe eines „Clipvalue/Clipnorm“-Parameters an den Optimizer, der den Gradienten reguliert. Die Skalierung in Kombination mit dem Mean-Squared-Logarihmic-Error macht eine Anwendung von tiefen neuronalen Netzen auf das vorliegende Problem überhaupt erst möglich. Wir werden später die Transformation der vorliegenden Daten und deren Effekt auf das Lernverhalten der Netze genauer untersuchen. Da wir nun bereits einen Transformator verwenden, müssen wir das bilden

Hyperparameter	Pool	Best Config
(Units, Nr. of Layers)	$\{(256,5),(512,3),(64,7),(1024,2),(128,6)\}$	$(256,5)$
Loss-Funktion	MAE, MSE, Huber	MAE
Optimizer	Adam, RMSprop	Adam
Aktivierungsfunktion	ReLU, Leaky-ReLU, Sigmoid, ELU, tanh	Leaky-ReLU
Learning-rate	$\{10^{-2}, 5\cdot10^{-3}, 10^{-3}, 10^{-4}\}$	10^{-2}
Batch-Größe	$\{256,128,512,768,1024\}$	256
Basis 10	True, False	True
Label-Normalisierung	$\{\text{keine}, [-1,1]\}$	keine
Feature-Normal.	True, False	True
Skalierung	True	
Logarithmus	True	
Max. Epochen	100	
Trainingspunkte	4.000.000	

Tabelle 4.4.: Hyperparameter Pools eines erfolgreichen Random-Search mit bester Konfiguration für den dreidimensionalen differentiellen Wirkungsquerschnitt des Diphoton Prozesses

des Logarithmus auch nicht mehr auf die Loss-Function abwälzen. Wir können nun das Netz direkt den Logarithmus der Wirkungsquerschnitte lernen lassen und haben alle nötigen Informationen zur Rücktransformation der Ausgaben des Netzes im Transformator gespeichert. So können wir den Komfort der in Keras implementierten Loss-Funktionen verwenden und müssen keine eigenen Implementationen kreieren, sobald wir beispielsweise einen Mean-Absolute-Logarithmic-Error verwenden wollen. Die Label-Normalisierung aus dem vohergehenden Modell wurde praktisch auch im Transformator umgesetzt.
Selbst mit den eingeführten Transformierungen ist das Erlernen jedoch immer noch keine triviale Aufgabe. Zur Hyperparameteroptimierung verwenden wir wieder einen Random Search. Für die Hyperparameter wurden mittlereweile zwar Algorithmen entwickelt, die effizienter sein sollen als ein Random Search, in dieser Arbeit konnte jedoch keine Verbesserung festgestellt werden. Konkret ausprobiert wurden ein "Bayesian Searchünd der "Hyperband-Tuner", wobei die Implementierung mit Keras-Tuner vollzogen wurde. Bayesian Search benutzt eine Objecitv-Funktion, die aus den bereits getesteten Hyperparametern eine Vorhersage für vielversprechende neue Hyperparameter-Kombinationen abgibt. Hyperband trainiert eine große Zahl an Modellen für wenige Epochen und sucht aus diesen die besten Kandidaten zum Weitertrainieren aus. Dies wird stufenweise durchgeführt, bis man in der Theorie mit einer Hand voll gut funktionierender Hyperparameter zurückbleibt. Die Hyperparameter einer erfolgreichen Suche sind in **??** aufgelistet. Hierbei muss man beachten, dass die



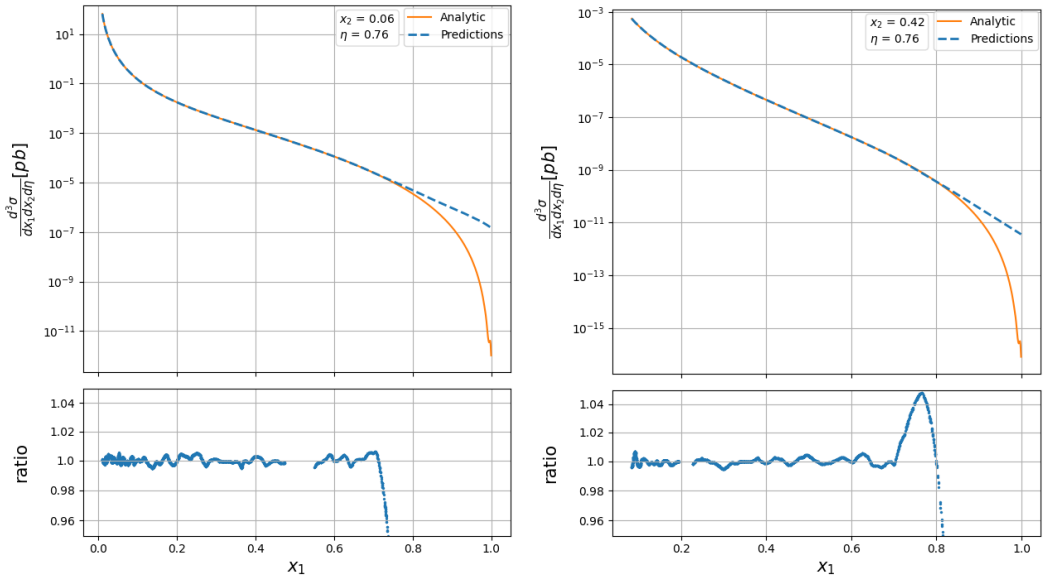
(a) (b)

Abbildung 4.7.: Schnitte des differentiellen Wirkungsquerschnitts in η

Trainingspunkte zufällig generiert sind nach den Verteilungen in **??**.

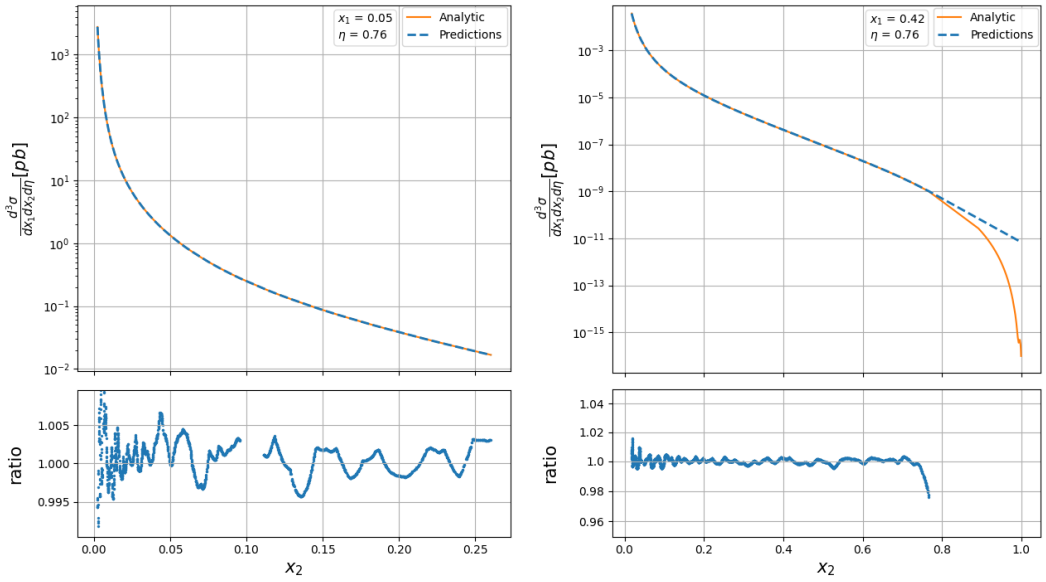
$$\rho(x) = \frac{1}{(x+\alpha)\ln\left(\frac{x_{max}+\alpha}{x_{min}+\alpha}\right)} \qquad \text{mit} \qquad \alpha = 0.005$$
$$\rho(\eta) = \begin{cases} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(\eta-\eta_{max})^2}{2\sigma^2}\right) & \text{für} \quad 0 \leq \eta \leq \eta_{max} \\ \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(\eta+\eta_{max})^2}{2\sigma^2}\right) & \text{für} \quad -\eta_{max} \leq \eta < 0 \end{cases} \qquad \text{mit} \quad \sigma = 1.5 \tag{4.4}$$

Nach der Generation müssen wir jedoch die Cuts aus **??** anwenden. Bei gegebenen Parametern und Cuts beläuft sich dabei die Sample-Effizienz auf $\approx 40\%$. Wir trainieren also mit weniger Daten, als es auf den ersten Blick wirkt. Das Sampling entspricht einem Importance-Sampling, dessen Nutzen schon im vorigen Abschnitt besprochen wurde. In den Abbildungen **??**, **??**,... sind Schnitte des dreidimensionalen Wirkungsquerschnittes an verschiedenen Phasenpunkten gezeigt. Wir sehen, dass sich unsere intensive Behandlung der Hyperparameter und der Daten-Transformationen bezahlt gemacht hat. Wir verzeichnen an den meisten Stellen eine maximale Abweichung von lediglich 0.5%. Für Ausnahmefälle beträgt die Abweichung bis zu $\approx 1\%$. Es lässt sich leicht der Moment erkennen, ab dem die x-Werte gefiltert wurden. Wie schon im Modell für $\frac{d\sigma}{d\theta}$, verläuft die Vorhersage des Modells linear weiter und entfernt sich somit von den analytischen Werten. Für große x wird unser Modell also den Wirkungsquerschnitt gegebenenfalls um Größenordnungen überschätzen. Da jedoch nur der integrierte Wirkungsquerschnitt über x_1, x_2 überhaupt messbar ist und dieser sich nicht merklich durch diese Abweichung beeinflussen lässt, ist es gerechtfertigt in



(a) (b)

Abbildung 4.8.: Schnitte des differentiellen Wirkungsquerschnitts in x_1



(a) (b)

Abbildung 4.9.: Schnitte des differentiellen Wirkungsquerschnitts in x_2

diesem Phasenraumbereich eine so große Abweichung zu besitzen.

4.2.3. Vergleiche

Wir wollen nun noch einmal direkt die Performance der einzelnen Hyperparameter an unserem Problem untersuchen. Dafür benutzen wir die Konfiguration, die wir im Vorhergehenden durch den Random Search gefunden haben und variieren für jedes Training nur ein Hyperparameter. Für die Anzahl an Neuronen und Layern, die stark korreliert sind, ist die Veränderung eines einzelnen der beiden Parameter nur bedingt interessant. Daher vergleichen wir auch verschiedene Formen bzw. Architekturen des Netzes, sprich verschiedene Kombinationen von Anzahl an Layern und Units. Die Modell werden nach dem Mean-Absolute-Percentage Error eines Test-Datestets beurteilt, das genauso gesampelt ist wie die Trainingsdaten. Wir trainieren jedes Modell fünf mal mit unterschiedlichen Initialisierungen an zwei Millionen gesampelten Phasenraumpunkten, um die statistische Schwankung der Güte des Modells einschätzen zu können. Die eingezeichneten Fehlerbalken sollen die Schwankung verdeutlichen und sind kein Maß dafür, welchen Fehler das Netz in der Praxis erreichen kann.

Loss-Funktion: In **??** a) ist der Vergleich von drei verschiedenen Kostenfunktionen gezeigt. Für Probleme mit stark variierenden Labels setzt sich der Mean-Absolute-Error durch, da dieser nicht so sensitiv auf Ausreißer oder, in unserem Fall, Polstellen ist. Der Huber-Loss, der eine Kombination des linearen Fehlers und des quadratischen Fehlers darstellt, schlägt sich insgesamt besser als der reine quadratische Fehler, kann insgesamt jedoch nicht mit dem linearen Fehler mithalten. Das quadratische Verhalten des Fehlers für kleine Abweichungen scheint für unser Problem nicht von Vorteil zu sein.

Optimizer: Der Vergleich des Optimizers **??** b) überrascht, da generell der Adam-Optimizer in Literatur und auch erfahrungsgemäß die besten Ergebnisse liefert. Betrachtet man die Fehlerbalken, scheint es als, als würde für unser Problem RMSprop konstant etwas bessere Ergebnisse erzielen. Allerdings zeigt unsere Trainingsreihe für Adam einen Ausreißer. Um ein definites Ergebnis zu erhalten, welcher Optimizer für unser vorliegendes Problem besser geeignet is, müsste man größere Versuchsreihen aufnehmen. Nichtsdestotrotz sollte man RMSprop nicht von vornherein abschreiben, denn er ist ein Versuch Wert. Der normale Stochastic-Gradient-Descent erzielt signifikant schlechtere Ergebnisse.

Trainingsdaten: Die Größe des Sets an Trainingsdaten ist in **??** c) verglichen. Wie erwartet nimmt der Fehler des Modells mit der Zahl an vorhandenen Trainingsdaten ab. Das gleiche gilt voraussichtlich für die Unsicherheit auf dem Ergebnis, auch wenn in unserem Fall das Modell, das mit den meisten Trainingsdaten einen Ausreißer zeigt. Man erkennt jedoch auch, dass die Performance des Modells konvergiert und mehr als vier Millionen gesampelte Daten keinen signifikanten Beitrag mehr leisten.

Learning-Rate: An dem Vergleich der Learning-Rates, der in **??** gezeigt ist, kann man ein interessantes Verhalten des Netzes erkennen. Für eine Learning-Rate von $1 \cdot 10^{-3}$ verändert sich der mittlere Fehler so gut wie überhaupt nicht. Das deutet darauf hin, dass wir unabhängig von unserer Initialisierung bei dieser Learning-Rate das gleiche lokale Minimum finden. Die Abweichung der Performance wird danach mit der Learning-Rate größer, wir finden nun höhere und tiefere lokale Minima. Bei einer zu kleinen anfänglichen Lernrate, bleiben wir schon früh in einem hohen Minima stecken und können keine brauchbaren Ergebnisse erzielen. Wir können daraus schlussfolgern, dass es gut sein kann, seine anfängliche Learning-Rate etwas größer zu initialisieren, als man intuitiv für richtig halten würde. Dadurch kann man eine größere Menge an lokalen Minima erkunden, insofern man über die benötigte Rechenleistung verfügt. Beachte jedoch, dass dieser Ansatz nur in Kombination mit einem Zeitplan zur Reduzierung der Lernrate funktioniert und das "Dying-ReLU„ verstärken oder sogar auslösen kann.

Daten-Transformationen: Welche Daten-Transformationen für unser Problem funktionieren, ist in **??** gezeigt. Ich möchte an dieser Stelle noch einmal die Wichtigkeit dieser Transformationen hervorheben. Während man in der Literatur viel über die Normalisierung oder das Reskalieren der Features liest, werden die Labels oft von Transformationen ausgenommen. Für spezielle Regressionsprobleme, wie es bei uns vorliegt, können diese jedoch der Schlüssel dazu sein, überhaupt konvergierende Modelle zu erhalten. **??** zeigt, dass verschiedene Implementationen brauchbare Ergebnisse liefern und es wichtiger ist, überhaupt die Skalierung und die Anwendung des Logarithmus zu verwenden. Trainingsläufe ohne Skalierung und Logarithmus sind nicht aufgeführt, da der Fehler nicht vergleichbar ist. Ein konvergierendes Modell ohne Logarithmus kann erhalten werden, wenn man anstelle des Logarithmus die Label-Normalisierung verwendet.

Achitektur: Die Architektur in **??** zu vergleichen ist interessant, da man sehen kann, dass eine passende Architektur zum Problem effektiver ist als die Komplexität des Modells. Das Modell mit den wenigsten zu trainierenden Parametern zeigt im Vergleich bessere Leistung als das komplexeste Modell. Die Modelle (128, 6), (256,5), (384, 4) zeigen unabhängig von ihren trainierbaren Parameter sehr gute Genauigkeit. Wir konnten also durch Orientierung am besten gefundenen Modell ein effizienteres finden, indem wir die Architektur ein wenig variiert haben. Das Modell (64, 7) zeigt trotz wenigen Freiheitsgraden akzeptable Genauigkeit.

Anzahl an Layer und Units pro Layer: Sowohl für die Anzahl an Layer und der Units pro Layer verläuft nach einer Kurve, die ihr Minimum bei unseren optimalen Parametern hat. Der Schritt zum jeweils simpleren Modell ist klein und kann bei Bedarf einen Kompromiss zwischen Geschwindigkeit und Genauigkeit darstellen.

Aktivierungsfunktionen: Die Abwandlungen der ReLU-Funktion zeigen sehr gute Ergebnisse, einsehbar in **??**. Die gewöhnliche ReLU ist überraschenderweise etwas

abgeschlagen. Eine plausible Erklärung hierfür liefert wiederum das "Dying ReLU"-Problem in Kombination mit unserer vergleichsweise hohen anfänglichen Lernrate.

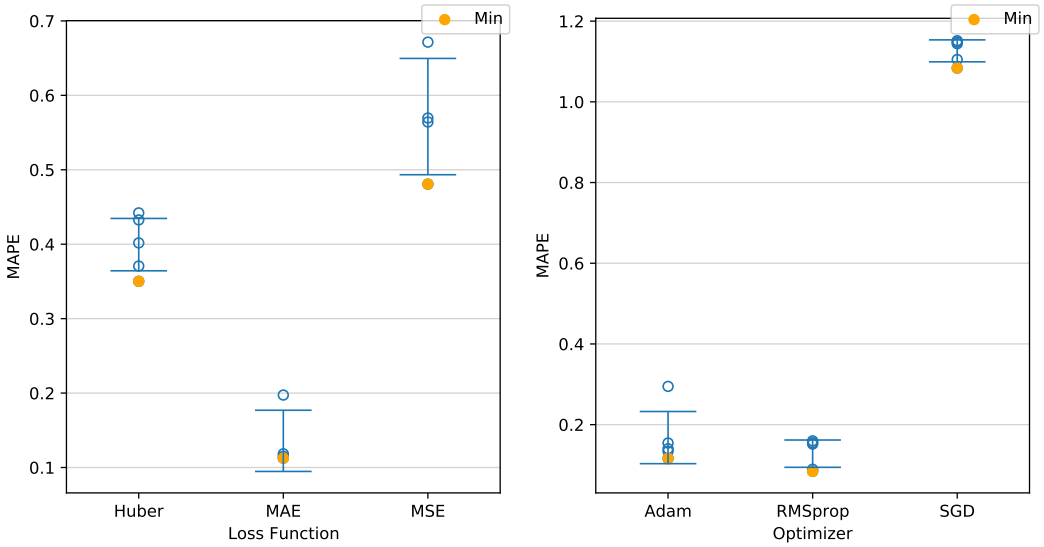
Batch-Sizes: Auch die Batch-Sizes (siehe **??**) haben große Auswirkungen auf das Lernverhalten des Modells. Ein Trainingsvorgang ist bei größerer Batch-Size mit passender Hardware zwar schneller, zeigt jedoch eindeutig größere Abweichungen.

4.3. Reweweight zwischen Fits der Partondichtefunktionen

4.3.1. ...

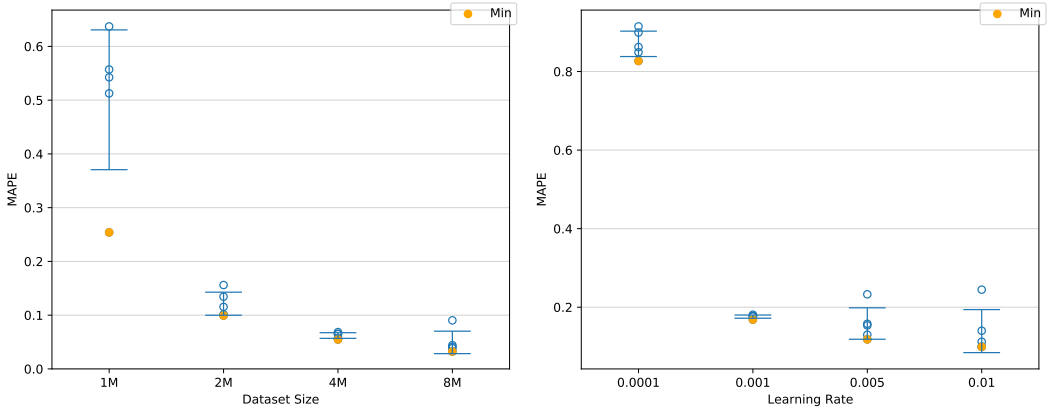
4.4. Transfer-Learning zwischen verschiedenen Fits der Partondichtefunktionen

4.4.1. ...



(a) Vergleich für verschiedene Loss-Funktionen

(b) Vergleich für verschiedene Optimizer RMSprop,SGD mit momentum = 0.1



(c) Vergleich für verschiedene Anzahl an Trainigspunkten

(d) Vergleich für verschiedene Anfangs-Lernraten

Abbildung 4.10.: Vergleich von Hyperparametern (I), MAPE: Mean-Absolute-Percentage-Error

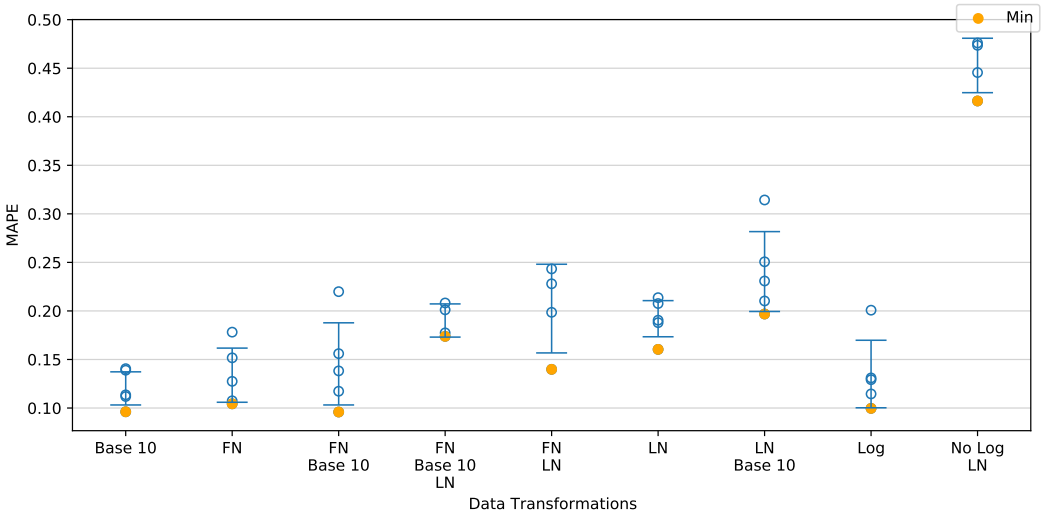


Abbildung 4.11.: wichtig: die Daten-Transformationen ohne die überhaupt nichts geht
Base 10: Daten werden mit Logarithmus zur Basis 10 transformiert
FN: Feature-Normalization
LN: Label-Normalization
Log: Nur Scaling+Logarithmus
No Log: Nur Scaling

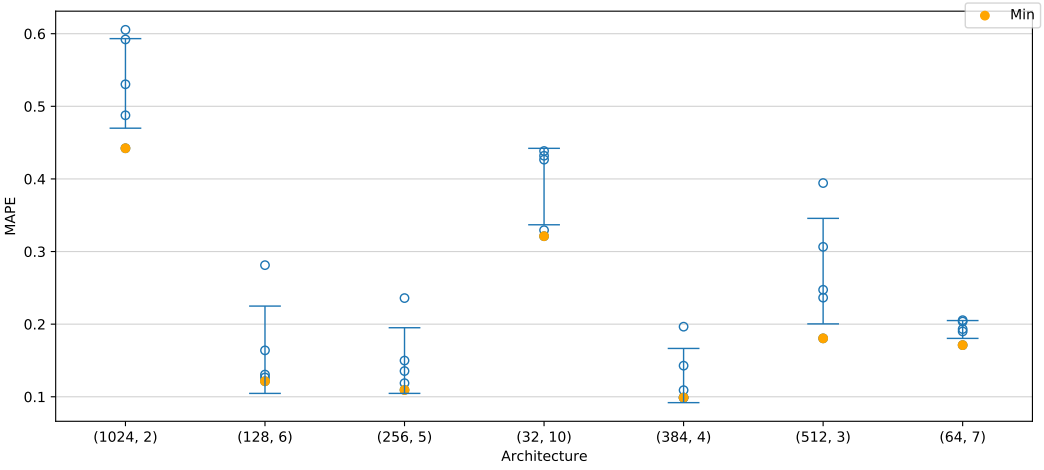
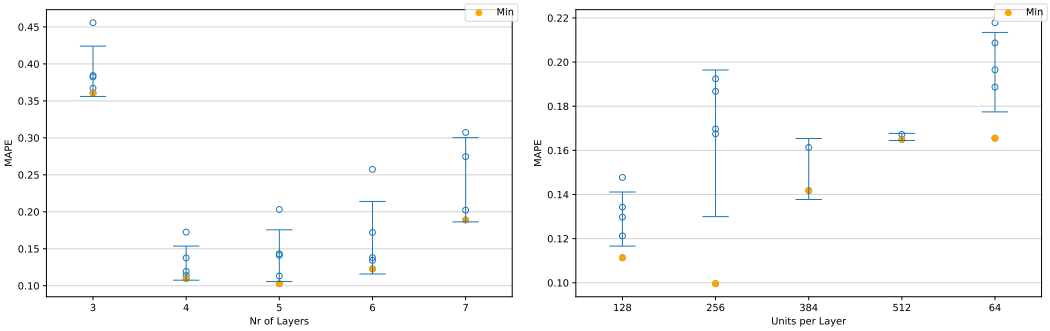
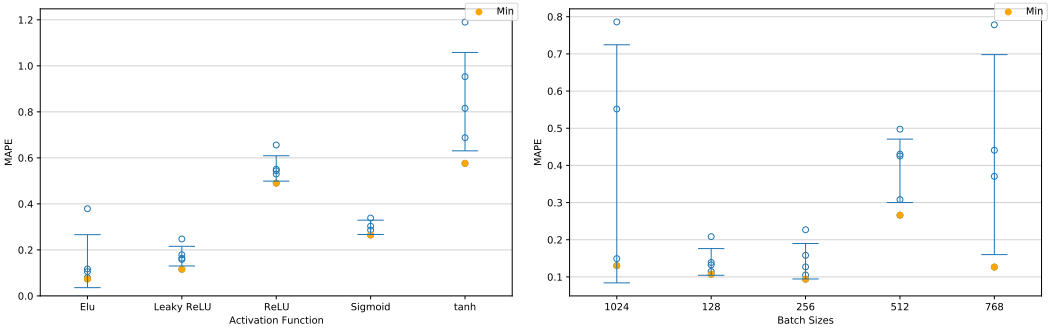


Abbildung 4.12.: Vergleich für verschiedene Modell-Architekturen.
x-Labels in (Units, Nr of Layers)



(a) Vergleich für verschiedene Zahlen an Layern

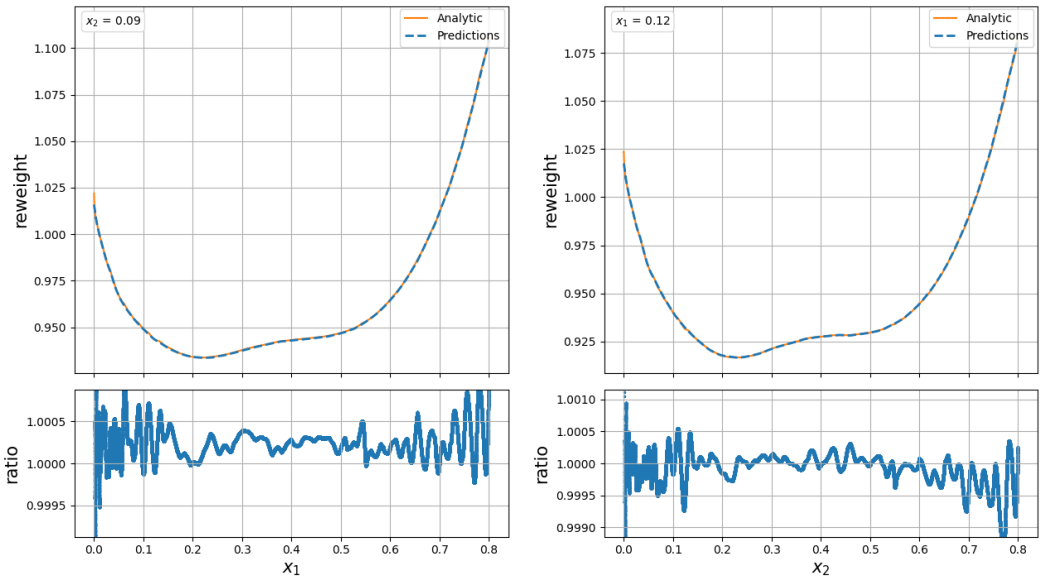
(b) Vergleich für verschiedene Zahlen an Units



(c) Vergleich für verschiedene Activation-Functions

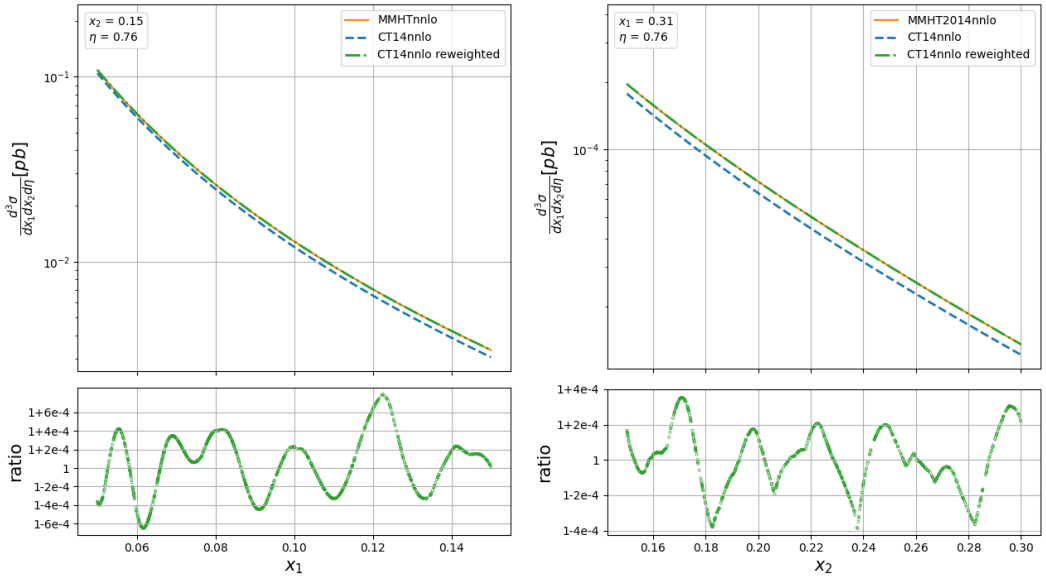
(d) Vergleich für verschiedene Batch-Sizes

Abbildung 4.13.: Vergleich von Hyperparametern (II), MAPE: Mean-Absolute-Percentage-Error

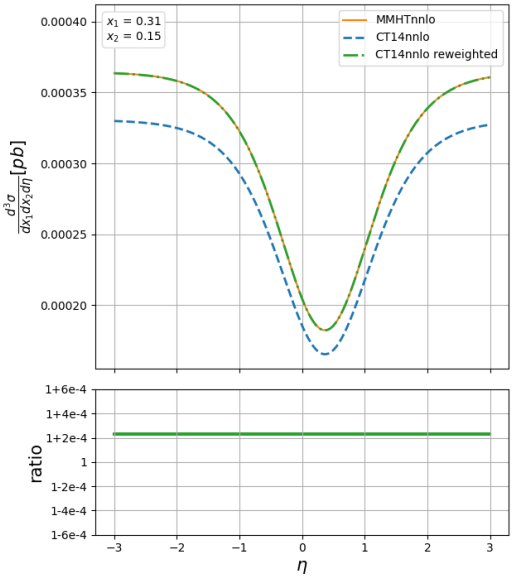


(a) Schnitt in x_1 (b) Schnitt in x_2

Abbildung 4.14.: Vlt noch 2 3D-plots, einmal die analytischen Reweights, gelernte Reweights werden aber gleich aussehen bei der geringen abweichung



(a) Schnitt in x_1 (b) Schnitt in x_2



(c) Schnitt in η

Abbildung 4.15.: Reweight von CT14nnlo auf MMHT2014nnlo mittels gelernten Weigths

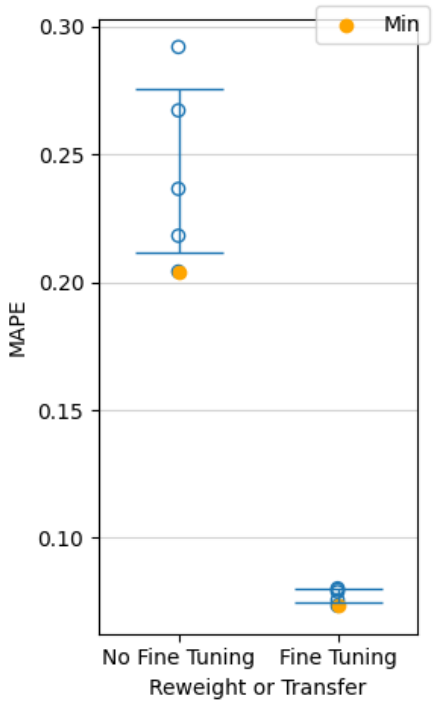
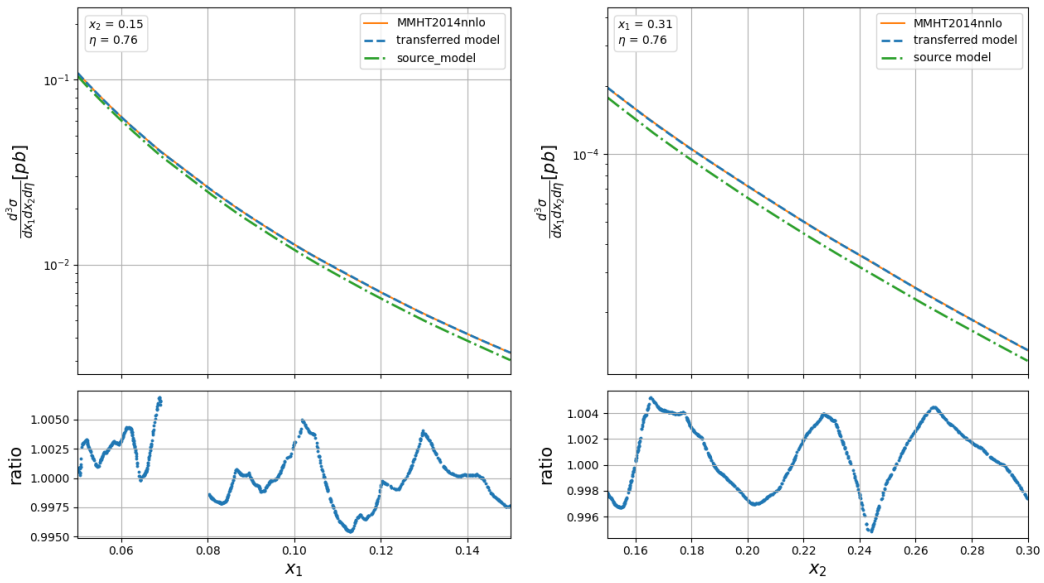
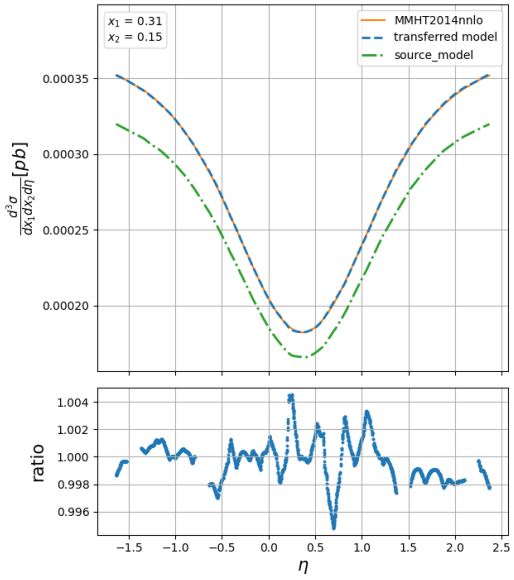


Abbildung 4.16.: Vergleich von Transfer Learning mit und ohne Fine-Tuning, Vergleich mit gereweithenem source model kommt noch

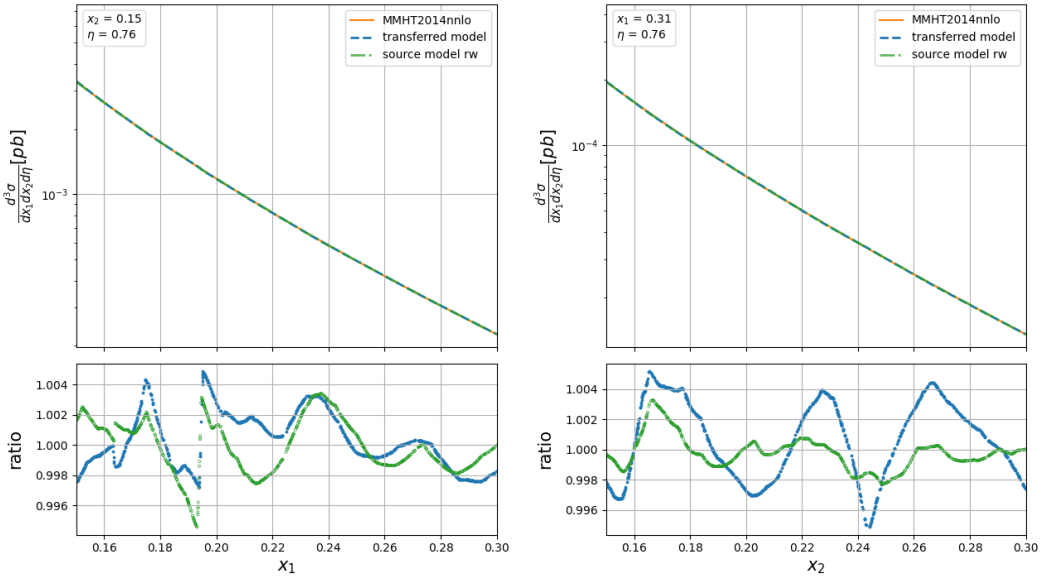


(a) Schnitt in x_1 (b) Schnitt in x_2



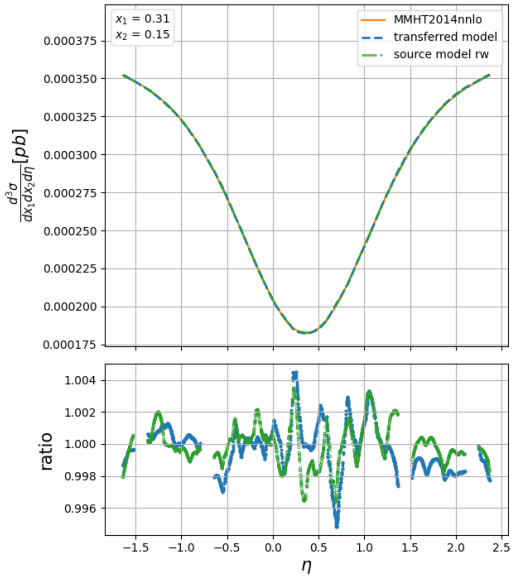
(c) Schnitt in η

Abbildung 4.17.: Transferiertes Model von Source Model zum Transfer model



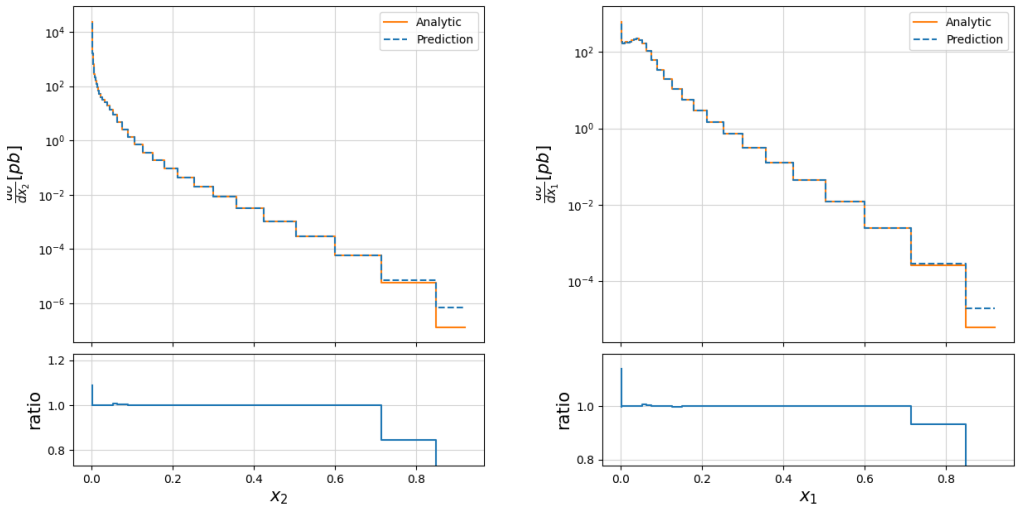
(a) Schnitt in x_1

(b) Schnitt in x_2

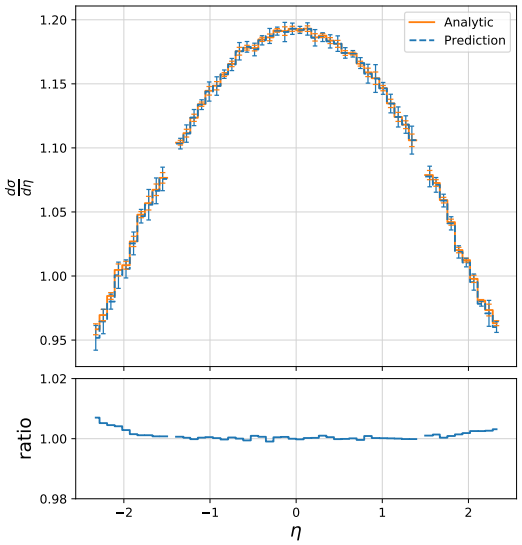


(c) Schnitt in η

Abbildung 4.18.: Vergleich transferiertes Modell, gereweightetes Source Model
rw: reweighted



(a) (b)



(c)

Abbildung 4.19.: MC-Integrationen über zwei Freiheitsgrade

5. Zusammenfassung und Diskussion

5.1. Zusammenfassung

5.2. Diskussion

5.3. Ausblick

A. Anhang

A.1. Grafiken

A.2. Source-Code

Callback	Config
LearningRateScheduler	nach einem Offset von 10 Epochen, wird die Learning-Rate nach jeder Epoche um 5% reduziert, bis diese auf $5 \cdot 10^{-8}$ abgefallen ist.
ReduceLROnPlateau	Fällt der Loss nach einer Epoche nicht um mindestens $2 \cdot 10^{-6}$, wird die Learning-Rate um 50% reduziert.
EarlyStopping	Fällt der Loss in drei aufeinanderfolgenden Epochen nicht um mindestens $2 \cdot 10^{-7}$ ab, wird der Trainingsvorgang gestoppt.

Danksagung

Danke an Christial Wiel.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit im Rahmen der Betreuung am Institut füer Kern- und Teilchenphysik ohne unzulässige Hilfe Dritter verfasst habe und alle verwendeten Quellen als solche gekennzeichnet habe.

Ort, Datum Unterschrift