Mehrdimensionales Datenfitting mit linearer Regression

Stefan Volz
Fakultät für angewandte Natur- und Geisteswissenschaften
Hochschule für angewandte Wissenschaften Würzburg-Schweinfurt
Studiengang Technomathematik
Matrikelnummer: 3519001
stefan.volz@student.fhws.de

20. Dezember 2019, Wintersemester 2019

Abstract

In vielen Anwendungen ist es nötig/vorteilhaft aus Messdaten ein Mathematisches Modell zu entwickeln, welches möglichst Messungenauigkeiten ausgleicht bzw. Vorhersagen zulässt. Ziel der Arbeit ist die Implementierung von linearer Regression als Funktion höherer Ordnung, welche mittels Gradientenabstieg eine Fehlerminimierung des Modells im Bezug auf gegebene Messdaten durchführt. Die Implementierung erfolgt in Julia.

Inhaltsverzeichnis

1 Konventionen und Generelles

1.1 Wahl der Programmiersprache und Abhängigkeiten

Die Implementierung erfolgt in Julia[1]. Julia ist eine hochperformante, stark und dynamisch typisierte, überwiegend imperative Programmiersprache mit Fokus auf wissenschaftlichem Rechnen; ist jedoch im Gegensatz zu z.B. MATLAB als General Purpose Sprache zu verstehen. Die Arbeit benutzt Julia in Version 1.2.0. Die Abhängigkeiten beschränken sich auf das `Plots` Package[2].

Die Wahl der Sprache fiel auf Julia, da es nativ eine sehr gute Unterstützung für Matrizen mitbringt was für die Implementierung als sehr vorteilhaft angesehen wurde. Außerdem erlaubt der gute Unicode-Support es, Identifier so zu wählen, dass diese nah an der Fachliteratur sind.

1.2 Quellcode

Beim Code wurde darauf geachtet, die Typannotationen[3] von Julia zu nutzen, da diese zum einfacheren Verständnis des Codes beitragen und außerdem der Performance zuträglich sind. Identifier wurden größtenteils so gewählt, dass sie sich mit [Bis09] decken - teils wurden auch Bezeichner aus [Lip06] übernommen.

1.3 Text dieser Arbeit

Im Text der Arbeit werden - in Anlehnung an [Bis09] - folgende Konventionen genutzt:

Typ	Beschreibung	Beispiel
vektorielle Größen	fettgedruckte Kleinbuchstaben	$\mathbf{w}$
einzelne Elemente eines Vektors	indizierte Kleinbuchstaben	w_j
Matrizen und Mengen	Großbuchstaben	A
Hyperparameter des Modells	griechische Kleinbuchstaben	γ
sonstige Parameter	lateinische Kleinbuchstaben	x
Code listings und Quellcode-Referenzen	monospace font	`code`

Einzelne Indizes an Matrixen wie z.B. A_j sind als Zeilenindizes zu verstehen, sodass A_j die j-te Zeile von A ist. In einigen Fällen wurde zwecks Konsistenzwahrung mit [Bis09] von diesen Konventionen abgewichen.

1.4 Variablenbezeichner

Einige der wichtigsten Bezeichner sind hier aufgeführt:

Bezeichner	Beschreibung
d	Dimension eines Eingabevektors
k	Dimension eines Zielwertsvektors
M	Anzahl der Modellparameter
N	Anzahl der Trainingsdatensätze
X	Trainingseingabematrix
T	Trainingszielmatrix
$\mathbf{w}$	Modellparameter

2 Grundlegende Implementierung

2.1 Problemformulierung

Seien $d, k, M, N \in \mathbb{N}$ mit Messdaten $X \in \mathbb{R}^{d\times N}$ und zugehörigen Zielwerten $T \in \mathbb{R}^{k\times N}$ gegeben. Gesucht werden die Parameter $\mathbf{w}$ eines Modells $y(\mathbf{w}, \mathbf{x})$ mit $y \in \mathbb{R}^M \times \mathbb{R}^d \to \mathbb{R}^k$ welche das Minimierungsproblem

$$\min_{\mathbf{w}\in\mathbb{R}^M} \sum_{i=1}^{N} E(y(\mathbf{w}, X_i), T_i),$$

[1] `https://julialang.org/`
[2] `https://docs.juliaplots.org/`
[3] `https://docs.julialang.org/en/v1/manual/types/`

im Bezug auf eine Fehlerfunktion $E : \mathbb{R}^k \times \mathbb{R}^k \to \mathbb{R}$ lösen.

Wir betrachten zunächst nur Probleme für die $k = 1$ gilt und bezeichnen daher die Zielwerte mit $\mathbf{t}$.

2.2 Lineare Regression

2.2.1 Mathematische Grundlagen

Bei linearer Regression handelt es sich um eine Methode des überwachten Lernens.

Definition 2.1. Lineare Regressions Modelle sind Modelle, welche sich im Bezug auf ihre Modellparameter linear verhalten[Bis09, S. 137f]. Sie stellen Funktionen der Form $y : \mathbb{R}^M \times \mathbb{R}^d \to \mathbb{R}^k, (\mathbf{w}, \mathbf{x}) \mapsto y(\mathbf{w}, \mathbf{x})$ dar. Die Anzahl der Modellparameter ist gegeben durch $M \in \mathbb{N}$, die Anzahl an Eingangsgrößen durch $d \in \mathbb{N}$ und die der Zielgrößen durch $k \in \mathbb{N}$.

Anmerkung 2.2. Dies bedeutet nicht, dass sie auch zwingend linear im Bezug auf die Eingangsvariablen sein müssen.

Das einfachste lineare Regressions Modell ist eine Linearkombination der Eingangsvariablen

$$y(\mathbf{w}, \mathbf{x}) = w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_D x_D,$$

diese spiegeln jedoch oftmals nicht die zugrunde-liegende Verteilung der realen Messwerte wider, und limitieren ein Modell mit d Eingangsvariablen auf $d+1$ Modellparameter. Daher werden Basisfunktionen eingeführt und Lineare Regressionsmodelle als Linearkombination dieser gebildet.

Definition 2.3. Eine Funktion $\Phi_j : \mathbb{R}^d \to \mathbb{R}^k, \mathbf{x} \mapsto \Phi_j(\mathbf{x})$ bezeichnen wir als Basisfunktion.

Anmerkung 2.4. Im Code werden Basisfunktionen als Funktionen $\Phi : \mathbb{N} \times \mathbb{R}^d \to \mathbb{R}^k, (j, \mathbf{x}) \mapsto \Phi(j, \mathbf{x})$ implementiert.

Mit diesen Basisfunktionen ergibt sich als Modellgleichung

$$y(\mathbf{w}, \mathbf{x}) = w_0 + \sum_{j=1}^{M-1} w_j \Phi_j(\mathbf{x}),$$

wobei der Parameter w_0 auch als Bias-Parameter bezeichnet wird, da er einen festen Offset der Daten ermöglicht. Im Code werden wir diesen Bias-Parameter als "normalen" Parameter betrachten und als kleinsten Index in $\mathbf{w}$ 1 wählen. Ein Eingangsvariablenunabhängiger Offset ist leicht durch eine angepasste Definition der Basisfunktion erreichbar:

$$\Phi : (j, \mathbf{x}) \mapsto \begin{cases} 1, & j = 1, \\ \Phi_j(\mathbf{x}), & sonst. \end{cases}$$

Damit vereinfacht sich die Darstellung der Modellgleichung zu

$$y(\mathbf{w}, \mathbf{x}) = \sum_{j=1}^{M} w_j \Phi(j, \mathbf{x}), \tag{1}$$

2.2.2 Implementierung

Auf Basis dieser Formulierungen können wir mit der Implementierung beginnen. Hierzu definieren wir uns zunächst eine Funktion Σ um alle im Code vorkommenden Summen zu abzudecken.

Listing 1 Funktion Σ

```
        "Sum from k=`from` to `to` of `a(k)`"
        Σ(from::Integer, to::Integer, a::Function, zero = 0) =
          mapreduce(a, (+), from:to; init = zero)
```

Die Funktion ist hierbei eine Implementierung von $\sum_{k=\text{from}}^{\text{to}} a(k)$. Bei `mapreduce` handelt es sich um eine eingebaute Funktion welche (hier) erst `a` über eine Sequenz mappt und anschließend diese Sequenz mittels `(+)`

reduziert/faltet. Dabei ist `(+)` der eingebaute Additionsoperator. Der optionale Parameter `zero` erlaubt es die Funktion auch für nicht-skalare Summen(bzw. jegliche Typen für die eine Implementierung zu `+` existiert) zu nutzen.

Listing 2 Funktion `y`

```
"""Linear Regression
# Args:
    w: Parameters
    Φ(j, x): Basis function of type (Int, Vector{T}) → T
    x: Input vector
"""
function y(
  w::Vector{<:Number},
  Φ::(T where T <: Function),
  x::Vector{<:Number})::Number
    Σ(1, size(w)[1], j→w[j] * Φ(j, x))
end
```

Diese Implementierung der Funktion `y` passt 1:1 zur mathematischen Formulierung in Gleichung 1[4].

2.3 Die Fehlerfunktion

Als nächstes benötigen wir eine Möglichkeit um den Fehler des Systems zu ermitteln - sodass wir diesen im nächsten Schritt minimieren können. Die genutzte Fehlerfunktion ist die des quadratischen Fehlers. Nach [Bis09, S. 140f] ergibt sich die Fehlerfunktion

$$E_D := \frac{1}{2}\sum_{n=1}^{N}(\mathbf{t}_n - y(\mathbf{w}, \mathbf{x}))^2. \tag{2}$$

Mit dieser Definition können wir nun unser Minimierungsproblem wie folgt definieren:

$$\min_{\mathbf{w}\in\mathbb{R}^M} \frac{1}{2}\sum_{n=1}^{N}(\mathbf{t}_n - y(\mathbf{w}, X_n))^2.$$

Im nächsten Schritt werden wir ein Verfahren implementieren, das diese Minimierung durchführt.

2.4 Gradientenabstiegsverfahren

2.4.1 Mathematische Grundlagen

Das Gradientenabstiegsverfahren ist ein numerisches Verfahren mit dem sich allgemeine Optimierungsprobleme lösen lassen. Beim Gradientenabstiegsverfahren bestimmen wir den Gradienten von E_D im Bezug auf die Modellparameter $\mathbf{w}$ - diesen Gradienten bezeichnen wir mit $\nabla_{\mathbf{w}} E_D$. Hierzu bestimmen wir zunächst die partielle Ableitungen von E_D nach allen w_k aus $\mathbf{w}$.

$$\frac{\partial E_D}{\partial \mathbf{w}_k} = \frac{\partial}{\partial \mathbf{w}_k}\frac{1}{2}\sum_{n=1}^{N}(\mathbf{t}_n - y(\mathbf{w}, \mathbf{x}))^2 = -\sum_{n=1}^{N}\Phi(k, X_n)\cdot(\mathbf{t}_n - y(\mathbf{w}, \Phi, X_n))$$

Hieraus folgt dann für den Gradienten:

$$\nabla_{\mathbf{w}} E_D = \begin{pmatrix} \frac{\partial E_D}{\partial \mathbf{w_1}} \\ \vdots \\ \frac{\partial E_D}{\partial \mathbf{w_M}} \end{pmatrix} =: \begin{pmatrix} e_1 \\ \vdots \\ e_M \end{pmatrix}.$$

[4]Die in Listing 2 genutzte Schreibweise `Vector{<:Number}` bedeutet *Vektor eines Typen T, wobei T ein Subtyp des Abstrakten Typs Number ist.* Diese hat gegenüber `Vector{Number}` den Vorteil, dass sie gewisse Optimierungen ermöglicht.

Für die komponentenweise Berechnung dieses Gradienten ergibt sich nach [Lip06, S. 95f]

$$\frac{1}{M} ppuuuuuuuuuuuuuh$$

Um diesen Gradienten zu implementieren, beginnen wir mit der Implementierung der partiellen Ableitung:

Listing 3 Funktion `∂E_D∂w_k`

```
    """Derivative of E_D with respect to w_k
    # Args:
        Φ(k, x_n): Basis function
        X: Set of inputs x_n where x_n is an input vector to Φ
        t: corresponding target values for each x_n
        k: Index for w_k in respect to which the derivative is taken
        w: Parameters
    """
    function ∂E_D∂w_k(
      Φ::Function,
      X::Matrix{<:Number},
      t::Vector{<:Number},
      w::Vector{<:Number},
      k::Integer)::Number
        N = size(t)[1]
        - Σ(1, N, n→Φ(k, X[n, :]) * (t[n] - y(w, Φ, X[n,:])))
    end
```

2.4.2 Einzeliteration des Gradientenabstiegs

Hiermit können wir eine Iteration des Gradientenabstiegsalgorithmus wie folgt implementieren.

Listing 4 Funktion `gradient_descent_iteration`

```
    """One iteration of the gradient descent algorithm
    # Args:
        ∂E_D∂w_k: Partial derivative of error function with respect to
            the k-th parameter
        X: Column vector of inputs x_n where x_n is an input vector to the
            error function
        t: corresponding target values for each x_n
        w: Parameters
        η: Learning rate
    """
    function gradient_descent_iteration(
      ∂E_D∂w_k::Function,
      X::Matrix{<:Number},
      t::Vector{<:Number},
      w::Vector{<:Number},
      η::Number)::Vector{<:Number}
        M = size(w)[1]
        ∇w = zero(w)
        for j = 1:M
            ∂E_D∂w_jk(k) = ∂E_D∂w_k(X, t, w, k)
            ∇w += collect(map(∂E_D∂w_jk, 1:M))
        end
        w - η * ∇w
    end
```

Innerhalb der Funktion iterieren wir über alle Indizes j, bilden die partielle Ableitung nach $\mathbf{w}_k$[5] und berechnen ihren Wert für alle Komponenten des Ergebnisvektors. Der Gradient ist dann die Summe all dieser Vektoren. In der letzten Zeile der Funktion machen wir einen "Schritt" in Richtung des Gradienten - steigen auf der Fehlerkurve also ab. Die Schrittweite wird hierbei durch den Hyperparameter η gesteuert. Dieser als Lernrate bezeichnete Parameter steuert im Grunde genommen die Konvergenzgeschwindigkeits des Gradientenabstiegsverfahrens.

2.4.3 Hauptfunktion

Auf Basis dieser Funktion können wir nun den Rest des Algorithmus in der Funktion `gradient_descent` umsetzen.

[5] Achtung: Die hier genutzte Funktion ist nicht die des globalen Scopes, sondern ein Parameter der Funktion.

Listing 5 Funktion `gradient_descent`

```
    """Minimize function E_D(X, t, w)
    # Args:
        ∂E_D∂w_k: Partial derivative of error function with respect to
            the k-th parameter
        X: Column vector of inputs x_n where x_n is an input vector to the
            error function
        t: corresponding target values for each x_n
        w: Initial parameters - usually `randn(M)`
        η: Learning rate
        M: Number of model parameters
        iters: Number of iterations
    """
    function gradient_descent(
      ∂E_D∂w_k::Function,
      X::Matrix{<:Number},
      t::Vector{<:Number},
      η::Number,
      M::Integer,
      iters::Integer,
      w::Vector{<:Number})::Vector{<:Number}
        ∇w = zero(w)
        for i = 1:iters
            w = gradient_descent_iteration(∂E_D∂w_k, X, t, w, η)
        end
        w
    end
```

Diese Funktion ruft `iters`-mal die Hilsfunktion `gradient_descent_iteration` auf und updated die Parameter mit dem Ergebnis dieser Aufrufe. Nach Ende der Optimierung gibt sie die vermeintlich optimal Werte der Parameter zurück.

Der Aufruf von `gradient_descent` erfolgt aus der Funktion `fit_linear_model` heraus.

Listing 6 Funktion `gradient_descent`

```
    """Find regression model
    # Args:
        Φ: Basis Function
        X: Set of inputs x_n where x_n is an input vector to Φ
        t: corresponding target values for each x_n
        η: learning rate with which to train
        M: Number of model parameters
        iters: Number of iterations
        optimizer: Parameter to select optimizer that's used
    # Fails:
        On unknown optimizers or error inside the optimizer
    """
    function fit_linear_model(
      Φ::Function,
      X::Matrix{<:Number},
      t::Vector{<:Number},
      η::Number,
      M::Integer,
      iters::Integer,
      optimizer = :gradient_descent)::Tuple{Function,Number}
        if optimizer == :gradient_descent
            w = gradient_descent(
              (X, t, w, k)→∂E_D∂w_k(Φ, X, t, w, k),
              X, t, η, M, iters, randn(M), ε, γ)
            residual_error = E_D(Φ, X, t, w)
            (x→y(w, Φ, x), residual_error)
        else
            error("Invalid optimizer")
        end
    end
```

Diese Funktion stellt im Grunde genommen nur einen Treiber für den Optimizer da. Sie initialisiert die Modellparameter zu Beginn mit einem Vektor aus normalverteilten Zufallszahlen[Lip06] aus dem Intervall $[-1, 1]$. Nach der Optimierung gibt sie das vollendete Modell als Closure[6] zurück, was dem die Funktion Aufrufenden ermöglicht, Modellaussagen in Abhängigkeit eines Eingangsvektors zu erhalten. Außerdem bestimmt sie den Restfehler des Modells und gibt auch diesen zurück. Der Parameter `optimizer` ist bisher noch nicht in Benutzung, da nur ein Optimizer implementiert ist.

2.4.4 Abbruchkriterium

Die Grundimplementierung wird mit einem einfachen Abbruchkriterium abgeschlossen. Hierbei wird zu jeder Iteration geprüft, ob die Norm der Differenz der Parametervektoren von zwei aufeinanderfolgenden Iterationen kleiner als ein neuer Hyperparameter ε ist - in Formeln ausgedrückt: es wird geprüft ob $||\mathbf{w} - \mathbf{w}'||_2 < \varepsilon$ gilt. Außerdem wird geprüft ob einer der Parameter zu $\pm\infty$ divergiert (und das Verfahren somit "fehlgeschlagen") ist oder durch einen Rechenfehler ein `NaN`[7] in $\mathbf{w}$ aufgetaucht ist. In all diesen Fällen wird der Algorithmus vorzeitig abgebrochen. Sofern kein Fehler vorliegt werden die Modellparameter zurückgegeben[8]. Außerdem ändert sich der Rückgabewert der Funktion diesbezüglich, dass sie nun zurückgibt wieviele Iterationen tatsächlich durchlaufen wurden.

[6] Eine Closure ist eine anonyme Funktion welche intern eine Referenz auf ihren Erstellungskontext hält. Hier wird sie eingesetzt um `y` mit $\mathbf{w}$ und Φ partiell zu evaluieren.

[7] *Not a Number* - spezieller Wert von IEEE floats

[8] Der Punkt im Funktionsaufruf in Listing 7 sorgt dafür, dass die Funktion vektorisiert/*pointwise* aufgerufen wird (siehe `https://docs.julialang.org/en/v1/manual/functions/`).

Listing 7 Funktion `gradient_descent` mit einfachem Abbruchkriterium

```
    """Minimize function E_D(X, t, w)
    # Args:
        ∂E_D∂w_k: Partial derivative of error function with respect to
            the k-th parameter
        X: Column vector of inputs x_n where x_n is an input vector to the
            error function
        t: corresponding target values for each x_n
        w: Initial parameters - usually `randn(M)`
        η: Learning rate
        M: Number of model parameters
        iters: Number of iterations
        ε: Gradient descent stops once the difference between two iterations
            (w and w') is less than ε

    # Fails:
        Fails on encountering NaN in computation or on Divergence to Inf
    """
    function gradient_descent(
      ∂E_D∂w_k::Function,
      X::Matrix{<:Number},
      t::Vector{<:Number},
      η::Number,
      M::Integer,
      iters::Integer,
      w::Vector{<:Number},
      ε = 10e-12::Number)::Tuple{Vector{<:Number},Integer}
        did_iters = 0
        for i = 1:iters
            did_iters += 1
            w_old = w
            w = gradient_descent_iteration(∂E_D∂w_k, X, t, w, η, ∇w, γ)
            if any(isnan.(w))
              error("Encountered NaN")
            end
            if any(isinf.(w))
                error("Divergence in calculation")
            end
            if norm(w_old - w) < ε
                break
            end
        end
        (w, did_iters)
    end
```

Für den Parameter ε wird eine arbiträr gewählte Standartbelegung von $10 \cdot 10^{-12}$ gesetzt.

3 Fortgeschrittene Features

3.1 Momentum Verfahren/Konjugierter Gradientenabstieg

Das Momentum Verfahren ist eine Adaption des Gradientenabstiegs, welche die Konvergenzgeschwindigkeit erhöhen soll. Das Ziel ist es, auf flachen Bereichen der Fehlerkurve die Abstiegsgeschwindigkeit zu erhöhen und sie in steilen Passagen zu verringern. Das Verfahren lässt sich als ein Ball der eine Kurve hinabrollt visualisieren - dieser verhält sich bei Änderungen der Kurve mit einer gewissen Trägheit. Um eine solche

Trägheit zu realisieren wird ein Momentum-Term hinzugefügt. Dieser Term ist das Produkt des Gradienten der vorhergehenden Iteration und eines neuen Hyperparameters $\gamma \in [0, 1)$ - dem Trägheitsfaktor. Der Gradient $\nabla_{\mathbf{w}}\hat{E_D}$ in jeder Iteration ergibt sich dann aus dem eigentlichen Gradienten dieser Iteration $\nabla_{\mathbf{w}}E'_D$ sowie dem der vorhergehenden Iteration $\nabla_{\mathbf{w}}E_D$ multipliziert mit dem Momentum-Faktor γ.

$$\nabla_{\mathbf{w}}\hat{E}_D = \nabla_{\mathbf{w}}E'_D + \gamma\nabla_{\mathbf{w}}E_D$$

Zur Implementierung müssen lediglich γ und $\nabla_{\mathbf{w}}E_D$(im Code `∇w_prior`) als neue Parameter hinzugefügt werden und die Rückgabe so angepasst, dass in jeder Iteration ein 2-Tupel aus neuem Parametervektor und aktuellem Gradienten zurückgegeben wird. Dementsprechend müssen auch die call-sites in `gradient_descent` sowie `fit_linear_model` angepasst werden. Da die Änderungen an den callsites trivial sind ist hier nur `gradient_descent_iteration` aufgeführt.

Listing 8 Funktionen `gd_iteration` und `gd` mit konjungiertem Gradientenabstieg

```
    """One iteration of the gradient descent algorithm
    # Args:
        ∂E_D∂w_k: Partial derivative of error function with respect to
            the k-th parameter
        X: Column vector of inputs x_n where x_n is an input vector to the
            error function
        t: corresponding target values for each x_n
        w: Parameters
        η: Learning rate
        ∇w_prior: Gradient of parameters from prior iteration
        γ: Momentum factor
    """
    function gradient_descent_iteration(
      ∂E_D∂w_k::Function,
      X::Matrix{<:Number},
      t::Vector{<:Number},
      w::Vector{<:Number},
      η::Number,
      ∇w_prior::Vector{<:Number},
      γ::Number)::Tuple{Vector{<:Number},Vector{<:Number}}
        M = size(w)[1]
        ∇w = γ * ∇w_prior
        for j = 1:M
            ∂E_D∂w_jk(k) = ∂E_D∂w_k(X, t, w, k)
            ∇w += collect(map(∂E_D∂w_jk, 1:M))
        end
        (w - η * ∇w, ∇w)
    end
```

Nach [Lip06, S. 110] wird γ mit einem Wert von 0.9 vorbelegt. Die wohl größte Problematik des konjugierten Gradientenabstiegs ist, dass der Fall auftreten kann, dass der Momentum-Term betragsmäßig größer als der aktuelle Gradient ist, jedoch das umgekehrte Vorzeichen besitzt. In diesem Fall würde sich der Fehler des Systems vergrößern. Aufgrund dessen kann hier keine Konvergenz garantiert werden. Ein kurzer Test (mit konstanter Lernrate und $\varepsilon = 10E - 12$) mit nur wenigen Datenpunkten zeigt folgende Daten:

γ	k bei Abbruch nach k Iterationen	Restfehler
0.0	50544	6.701991676725787
0.5	28837	6.701991676725787
0.9	5635	6.701991676725783

Also gibt es durchaus Fälle, in denen durch dieses Verfahren eine enorme Steigerung bei der Konvergenzgeschwindigkeit erzielt wird. Jedoch sehen die Verläufe der Kurven von Restfehler und Gradient wie folgt aus:

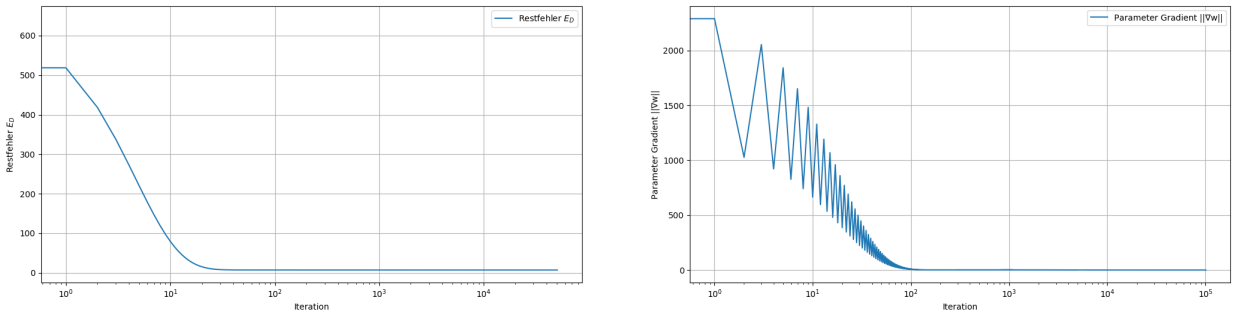

Abbildung 1: Beispielfit mit $\gamma = 0$

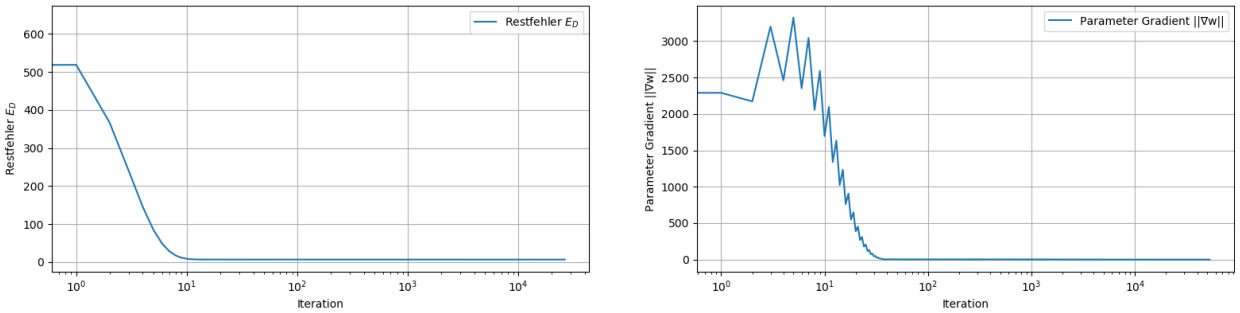

Abbildung 2: Beispielfit mit $\gamma = 0.5$

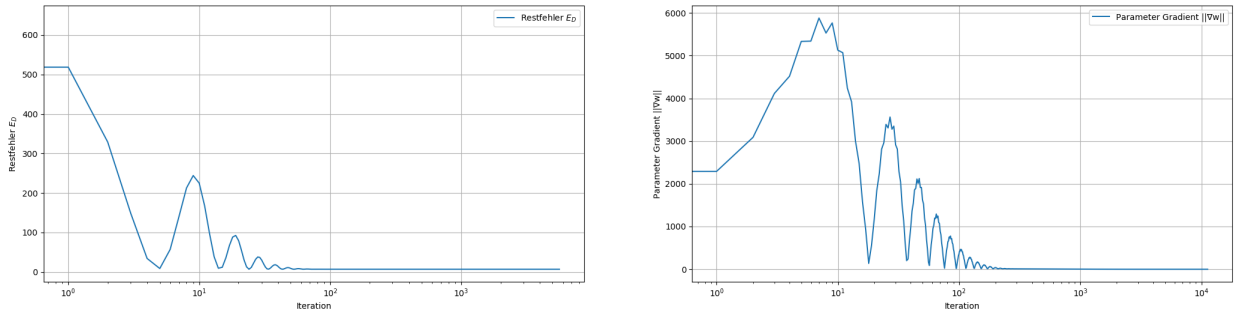

Abbildung 3: Beispielfit mit $\gamma = 0.9$

Es ist klar zu sehen, dass der oben genannte 'worst case' im Fall $\gamma = 0.9$ eingetreten ist. Das vermeintlich bessere Konvergenzverhalten lag daran, dass ein sehr geringes ϵ gewählt wurde. Erhöhen wir ϵ auf beispielsweise $10e-3$, dann ergeben sich die folgenden Werte:

γ	k bei Abbruch nach k Iterationen	Restfehler
0.0	24	10.216203809068343
0.5	14	6.813835951850837
0.9	29	36.63725946510655

beziehungsweise folgende Graphen:

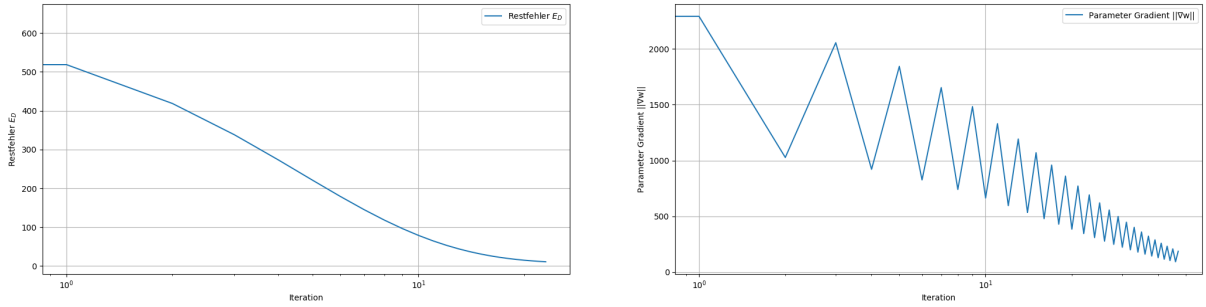

Abbildung 4: Beispielfit mit $\gamma = 0$

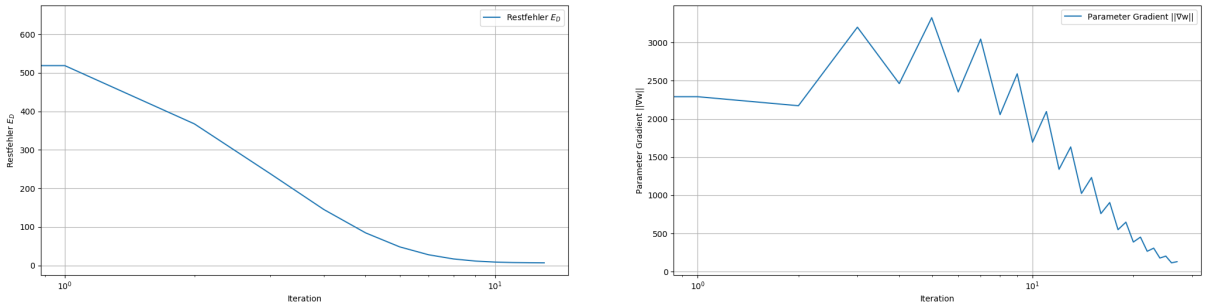

Abbildung 5: Beispielfit mit $\gamma = 0.5$

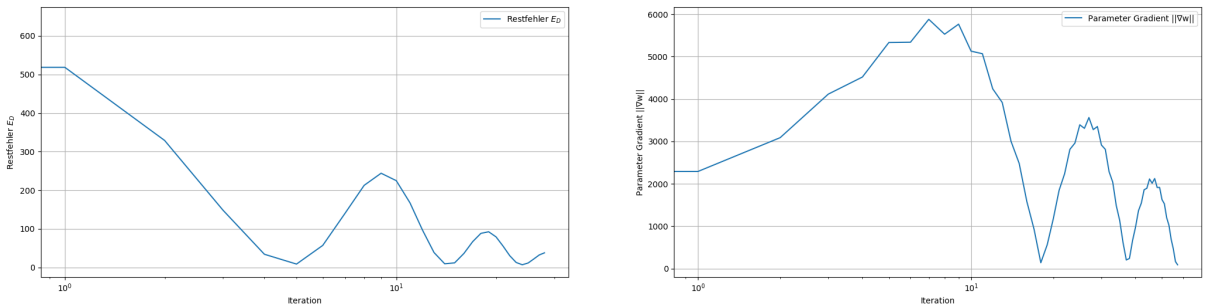

Abbildung 6: Beispielfit mit $\gamma = 0.9$

Das Momentumverfahren wirkt sich hier also teils positiv und teils negativ aus.

3.2 Adaptive Lernrate

3.3 Weight Decay

4 Beispiele

Literatur

[Bis09] Bishop, Christopher M.: *Pattern recognition and machine learning*. Springer-Verlag, 2009. – ISBN 978–1–4939–3843–8

[Lip06] Lippe, Wolfram-Manfred: *Soft Computing*. Springer-Verlag, 2006. – ISBN 978–3–540–20972–0