Deep Learning

Grundlagen Grundbegriffe Grundübungen

Alfatraining Kurs 2025-08 Dozent: Karsten Flügge

Themen des Tages

Tag 5

Momentum

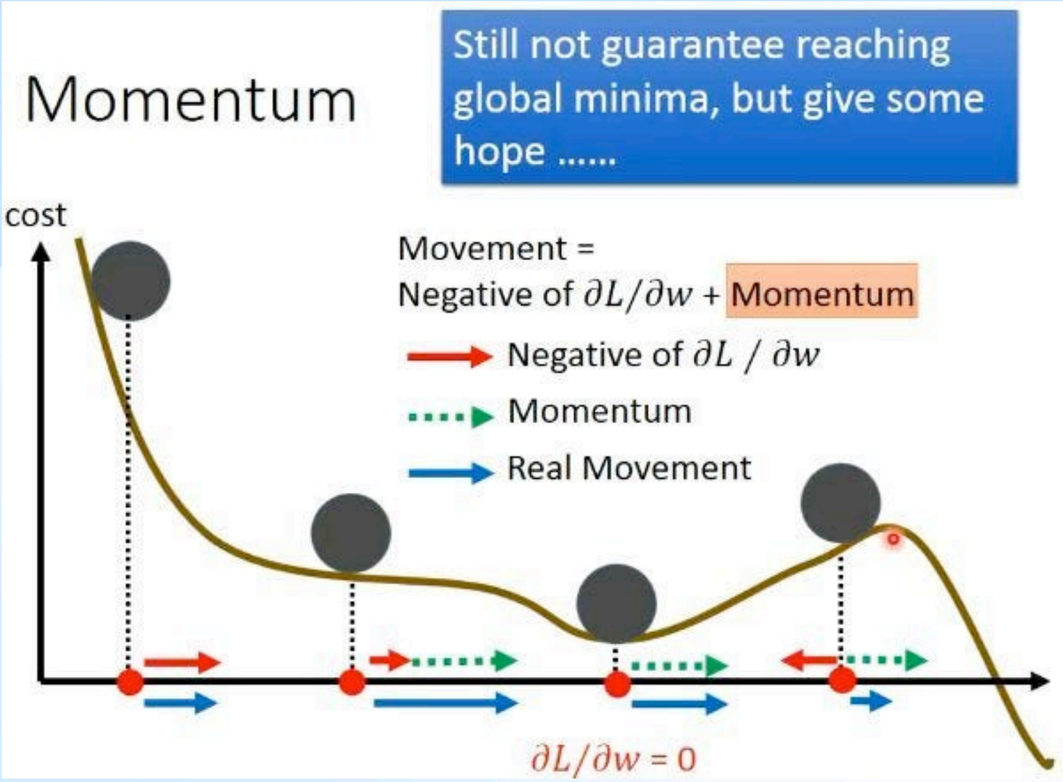
Lernrate

Adam Optimizer

Momentum

Wie eine Kugel welche beim Herabrollen am Berg Geschwindigkeit und **Schwung** aufnimmt und somit über kleine Kuhlen und Hindernisse drüber hüpft, sowie Ebenen schneller durchrollt.

So wird beim **Gradientenabstieg mit Momentum** schnellere Konvergenz erreicht, sowie das stecken bleiben in lokalen Minima vermieden.

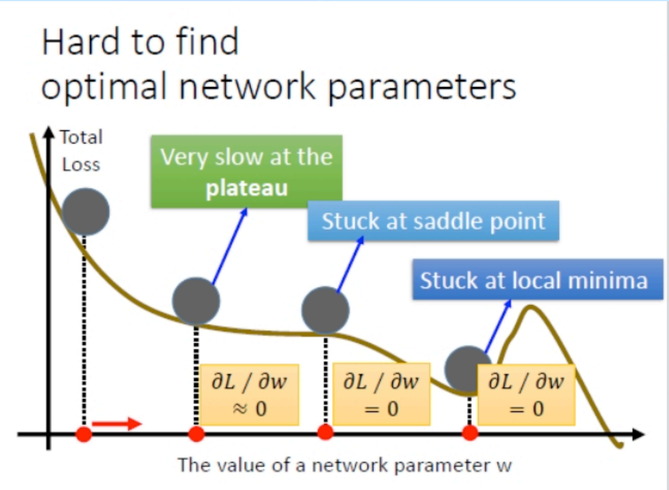


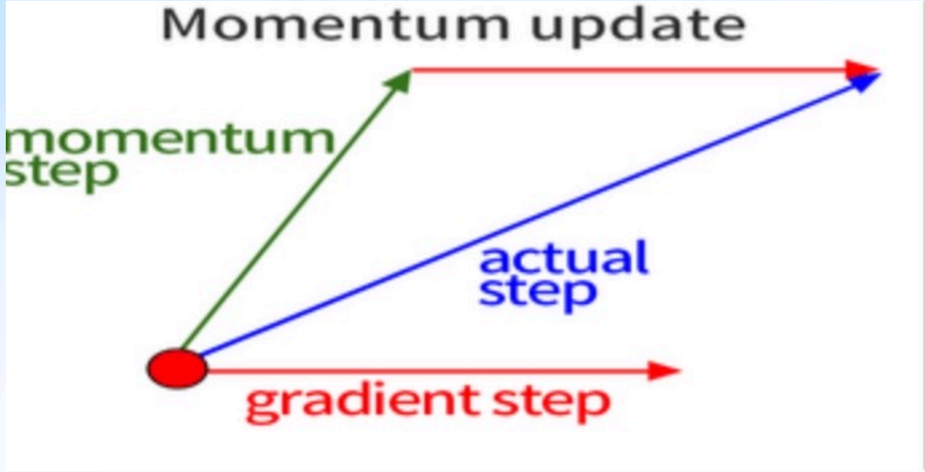
Momentum

Wie eine Kugel welche beim Herabrollen am Berg Geschwindigkeit und **Schwung** aufnimmt und somit über kleine Kuhlen und Hindernisse drüber hüpft, sowie Ebenen schneller durchrollt.

So wird beim **Gradientenabstieg mit Momentum** schnellere Konvergenz erreicht, sowie das stecken bleiben in lokalen Minima vermieden.

Es wird die **vorherige Bewegungsrichtung und Geschwindigkeit** gespeichert und zum neuen Gradienten **hinzu addiert**





Momentum "Impuls" p = **m** * v (petere = mass * **velocity**) "Schwung"
Impulse "Stoß" J = Δp

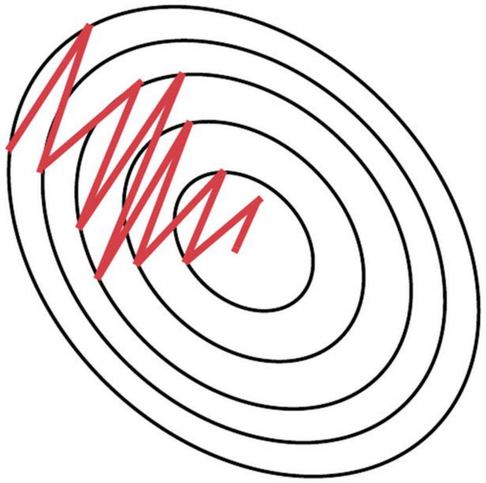
SGD with Momentum

```
momentum = 0.9  # Hyperparameter
```

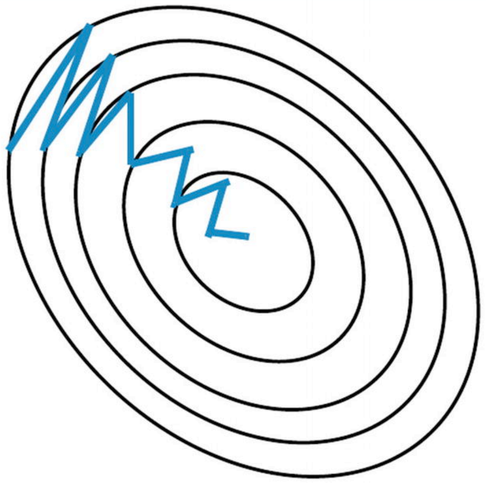
```
optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=momentum)
```

optimizer = **tf.keras**.optimizers.SGD(learning_rate=learning_rate, momentum=momentum)

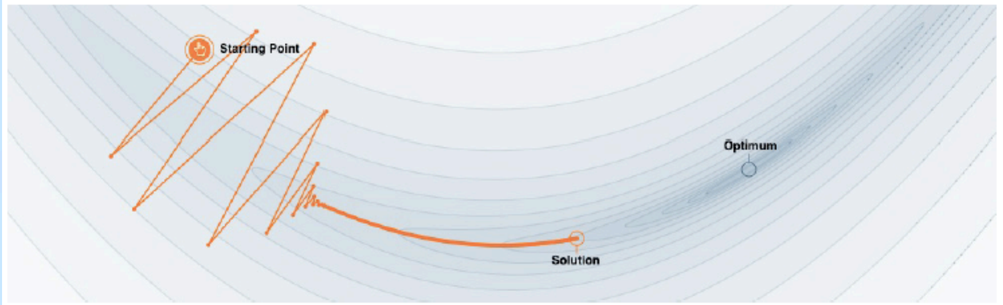
⚠️💡 **müssen wir aber gar nicht lernen, es folgt besseres …**



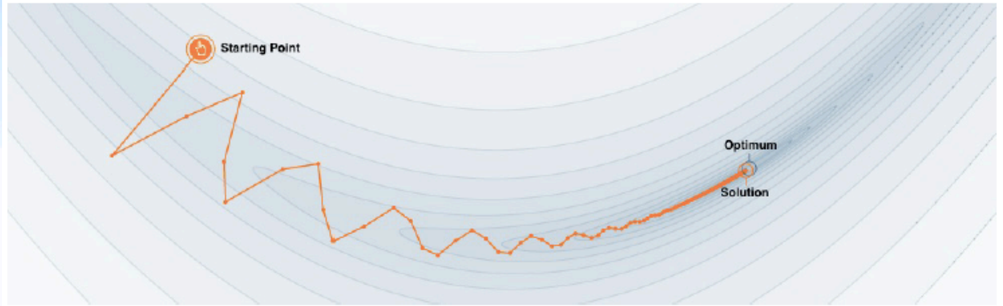
Stochastic Gradient Descent **withhout** Momentum



Stochastic Gradient Descent **with** Momentum



(a) The SGD trajectory: large oscillation and bad convergence.



(b) The Momentum trajectory: small oscillation and better convergence.

SGD with momentum

SGD without momentum

Update der Gewichte:
Newton Verfahren

```
w = w + schrittweite * ableitung
```

SGD:

```
w = w + learning_rate * gradient
```

SGD with momentum
v0 = 0 # Anfangsgeschwindigkeit
v = learning_rate * gradient

Merken / Mitschleifen der alten Geschwindigkeit

v = gamma * v_alt + learning_rate * gradient

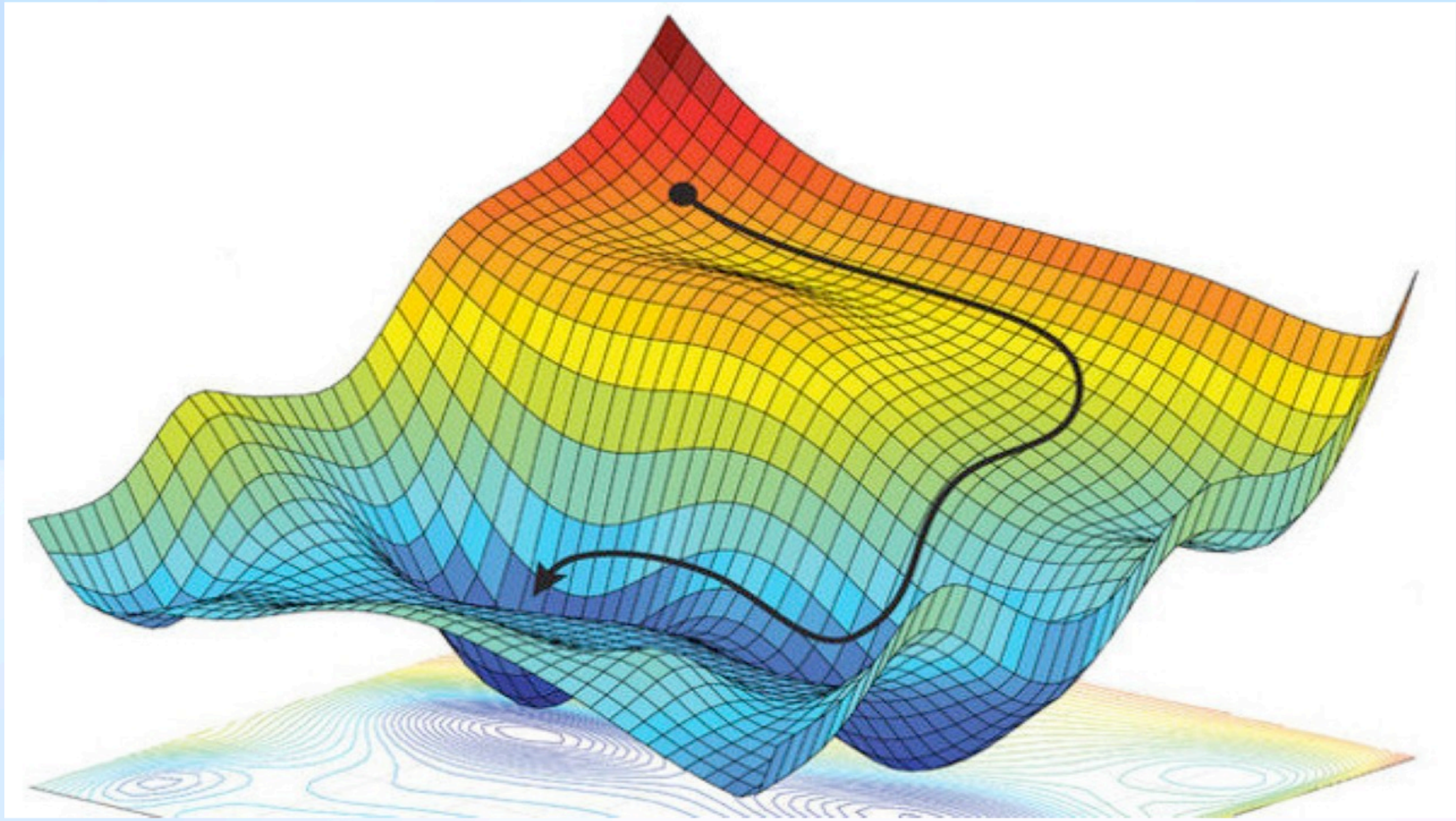
```
w = w + v       update der Gewichte
```

gamma ist ein neuer Hyperparameter und entspricht etwa der **Masse** in p = **m** * v
gamma ist 'momentum' Hyperparameter zum steuern wie schnell wir Momentum aufbauen wollen

Exponential moving average

```
z.B. gamma = 1/2
v1 = learning_rate * gradient
v2 = 1/2 v1 + 0 ( angenommen es gab nur einmal am anfang einen Gradientenimpuls)
v3 = 1/2 v2 = 1/4   => exponentieller Verfall der alten Geschwindigkeit
```

Pause



Aufgabe

`tag5/momentum.py` code aufrufen und verstehen

Aufgabe

In Perceptron.py vom Tag 2 den Momentum Mechanismus einfügen
(in backward methode)

Hyperparameter gamma:
wie viel wird von alter Geschwindigkeit beibehalten

Pause

Lernrate

Mit dem Hyperparameter "Lernrate" kann man Steuern wie schnell oder aggressiv das Netz neue Informationen absorbiert. konkret bedeutet das wie stark der Gradient beim Update die vorherigen Gewichte beeinflussen soll.

Im Trivialfalle welchen wir bereits kennen gelernt haben ist das einfach der Faktor mit welchem der Gradient abgezogen wird:

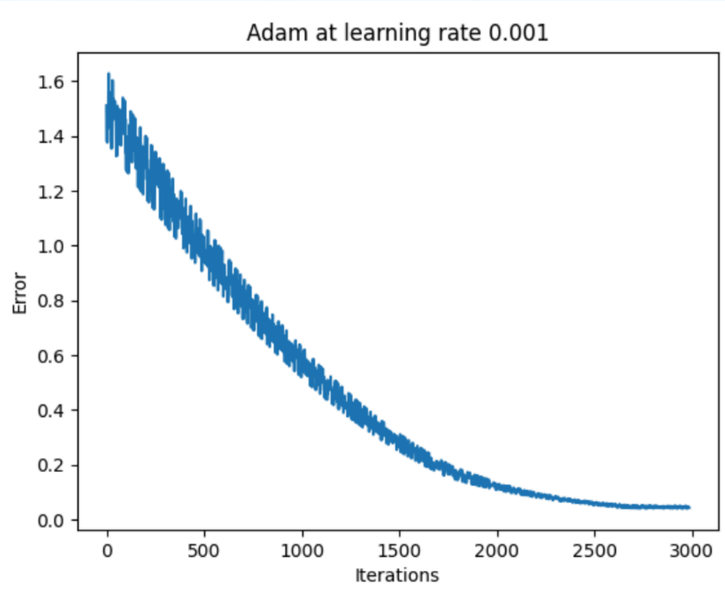
```
def sgd_update(weights, learning_rate, gradient):
    """Update weights using a gradient and learning rate."""
    return [w - learning_rate * g for w, g in zip(weights, gradient)]
```

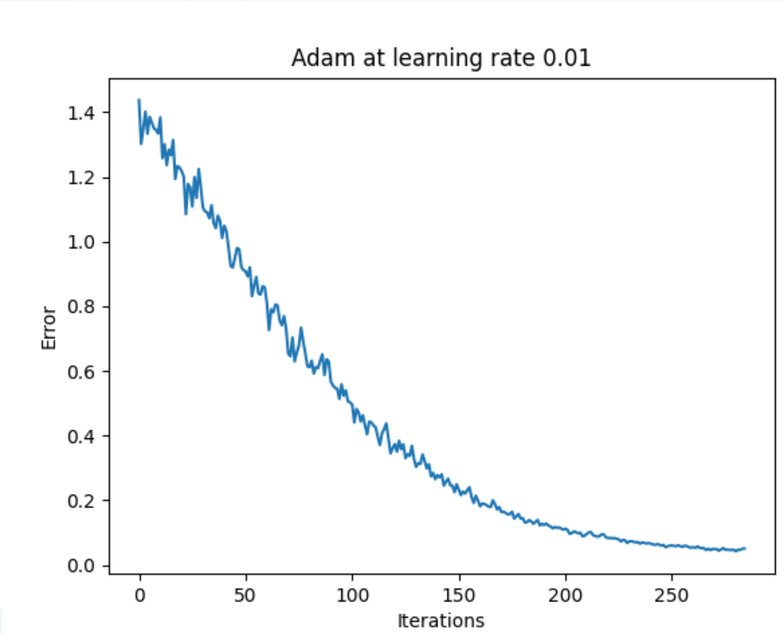
Nun da wir Momentum kennen gelernt haben kann man natürlich auch beeinflussen wie stark der Impuls beim Gradienten Abstieg sein soll. Prinzipiell kann man später alles mögliche steuern.

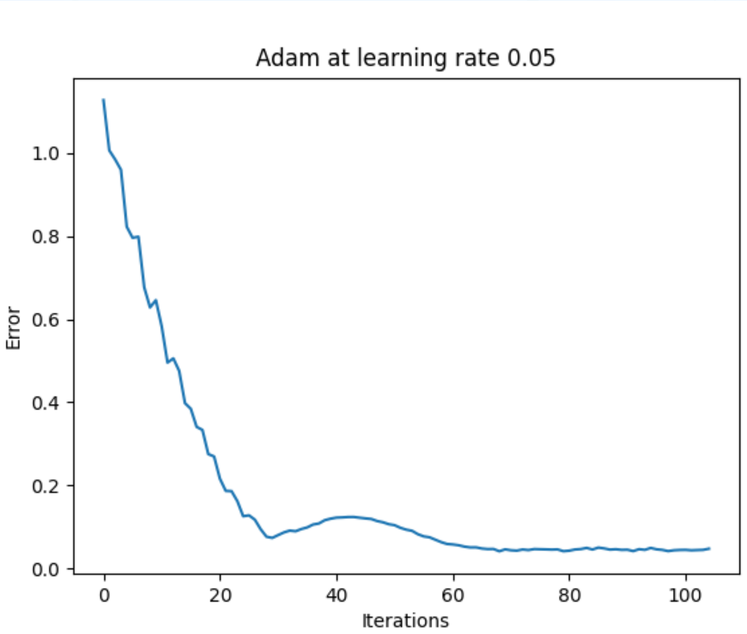
Learning rate

Gute Lernrate typischerweise zwischen 0.001 und 0.1 (hängt natürlich vom Problem ab)

"mehr hilft mehr" gilt nur am Anfang (beachte Anzahl der benötigten Iterationen≈Epochen!)



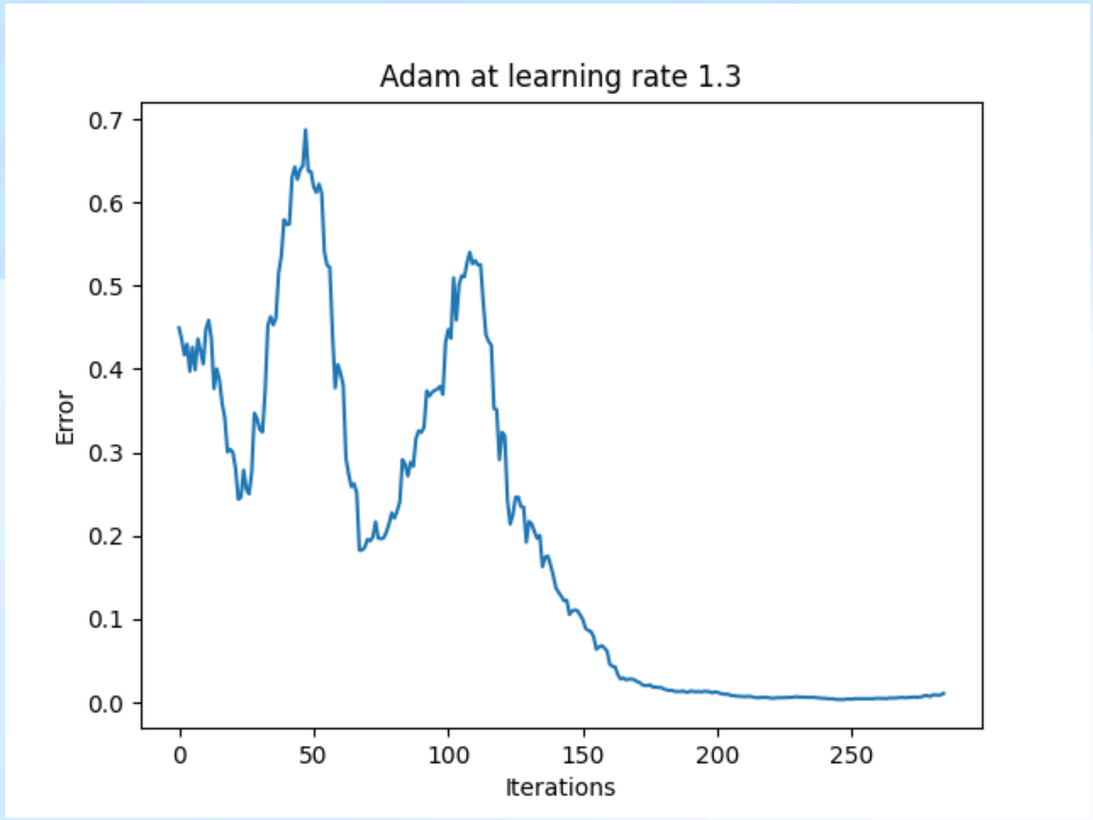


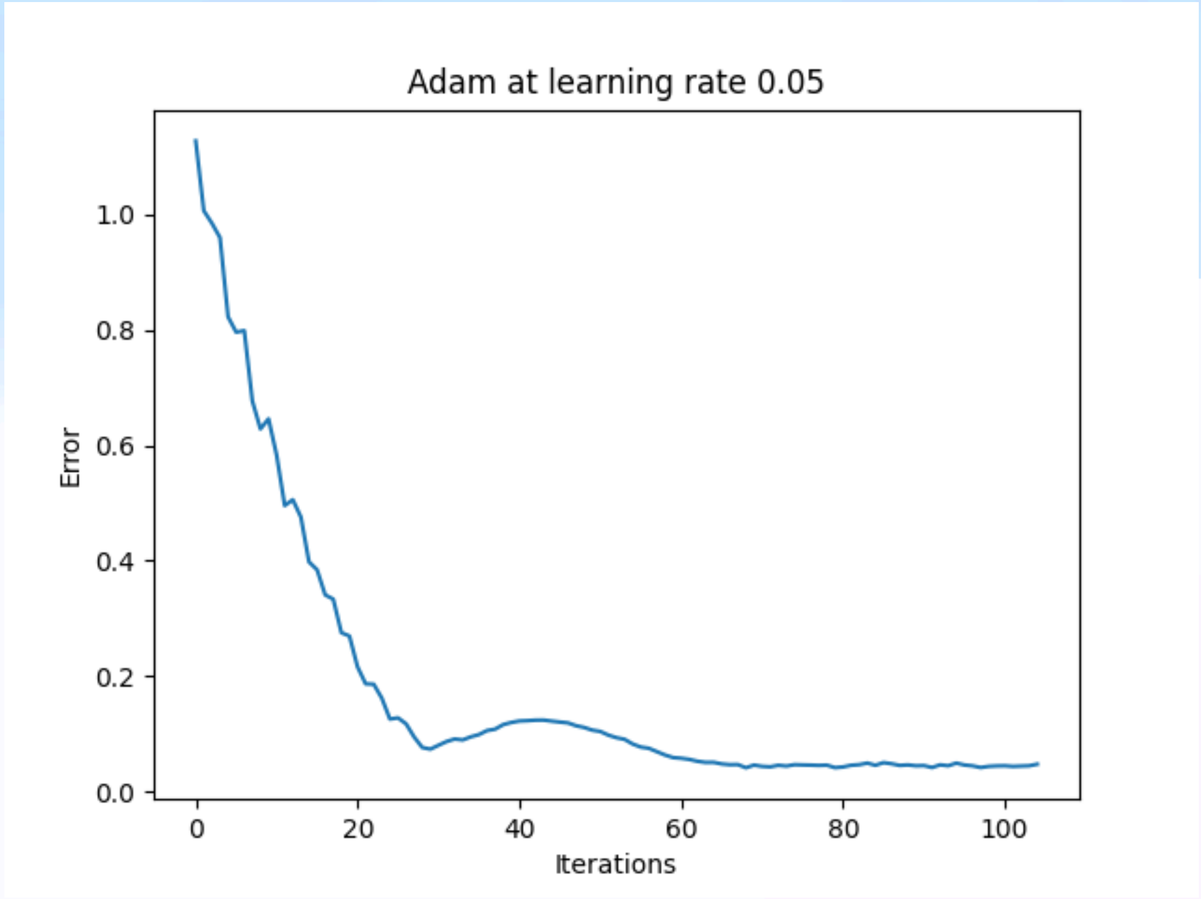


Chaos

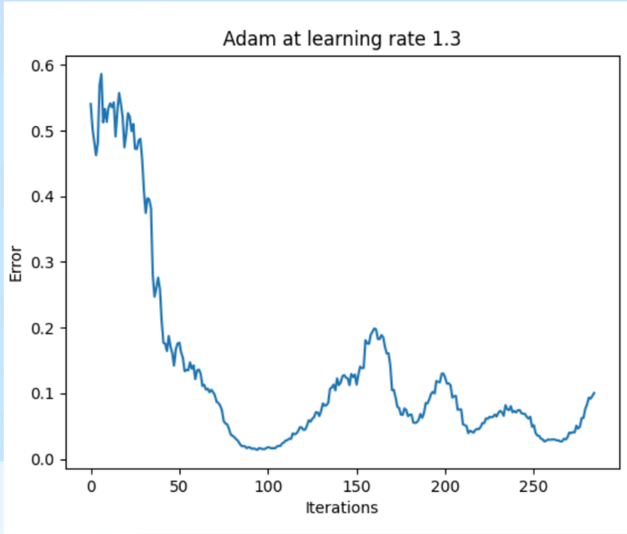
Lernkurve kann bei tiefen Netzwerken auch bei guter Lernrate "chaotisch" werden.

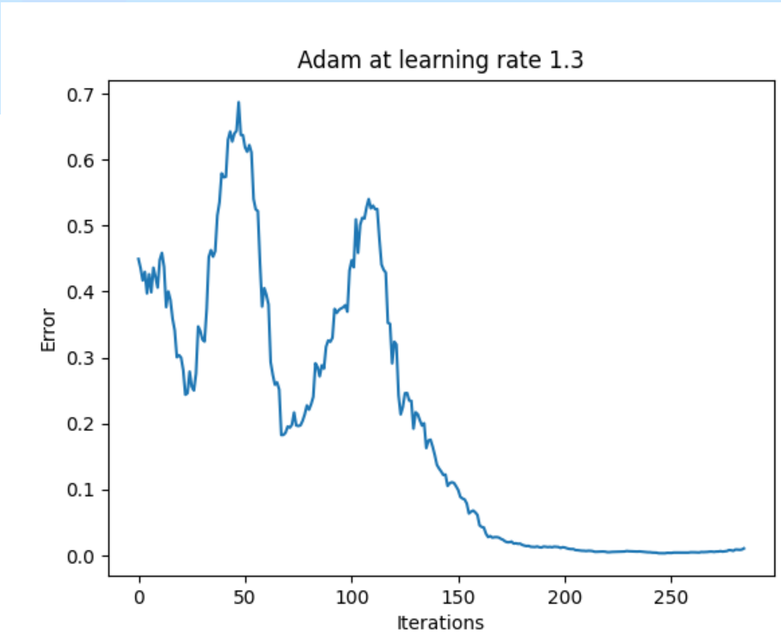
Bei kleinsten Änderungen der Lernrate kann die Lernkurve ganz verschieden aussehen!

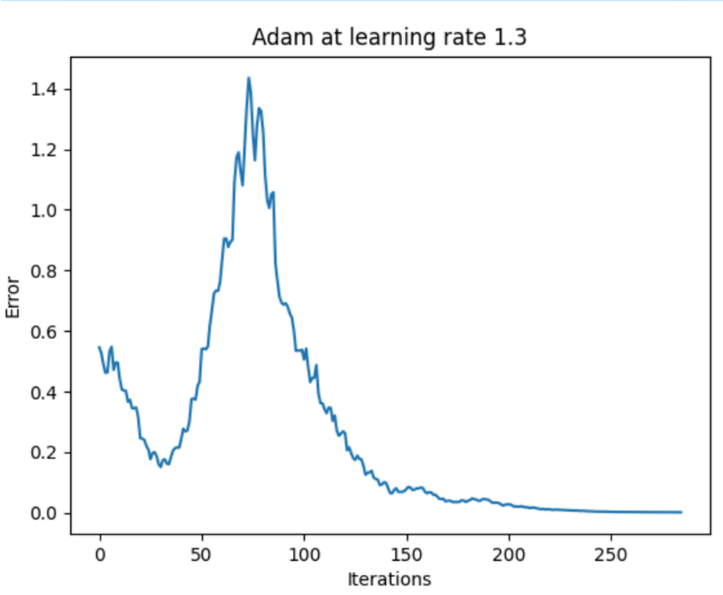




Learning rate







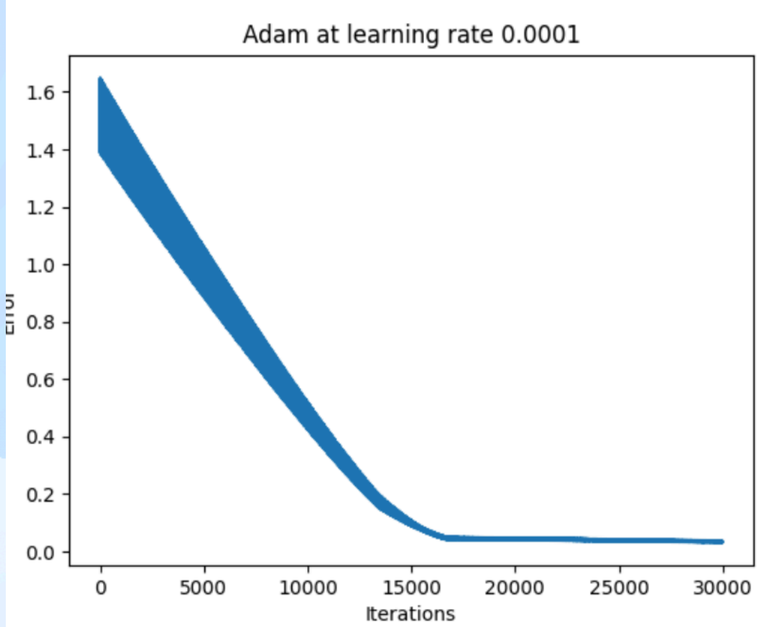
⚠️ **Katastrophales Vergessen** bei **zu hoher Lernrate** möglich

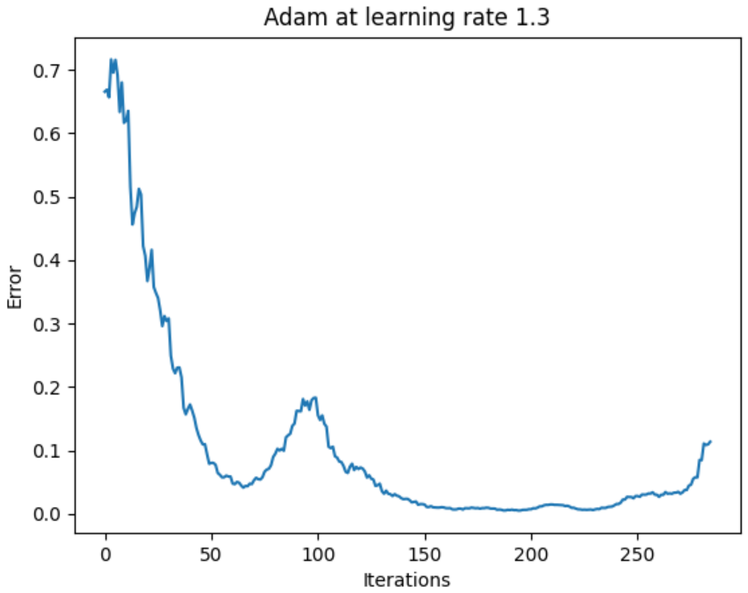
💡 **"System fängt sich wieder"** typische Beobachtung auch bei guten Lernraten!

⚠️ **Chaotisches** Verhalten sogar bei konstanter Lernrate:
Lernkurven zeigen oft chaotisches Verhalten, also grosse Variationen bei kleinsten Änderungen

Learning rate

Zu niedrige Lernrate = **sehr langsames** Lernen:
Zu hohe Lernrate = **keine Konvergenz** oder **Katastrophales Vergessen**





Learning rate

Quintessenz:

Learning rate typischerweise zwischen 0.0001 und 0.1 NIE über 1.

Aufgabe:

Hyperparameter Suche mit allem Konstant, nur Lernrate variiert,
zB zwischen 0.0000001 und 10
Vergleiche die **plots**

Fortgeschritten: verwende einen **Scheduler**

Adaptive learning rate

Adaptive Lernrate

Eine fortgeschrittene Lerntechnik ist mit **hoher Lernrate** zu **beginnen** und diese dann **sukzessive zu reduzieren.**

```
import torch.optim.lr_scheduler
scheduler = lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)

# … in training loop:
   …
   optimizer.step()
   scheduler.step(loss)
```

💡 Um nicht zu lange of Ebenen zu verbringen kann man auch gelegentlich schockartig die Lernrate kurz senken:

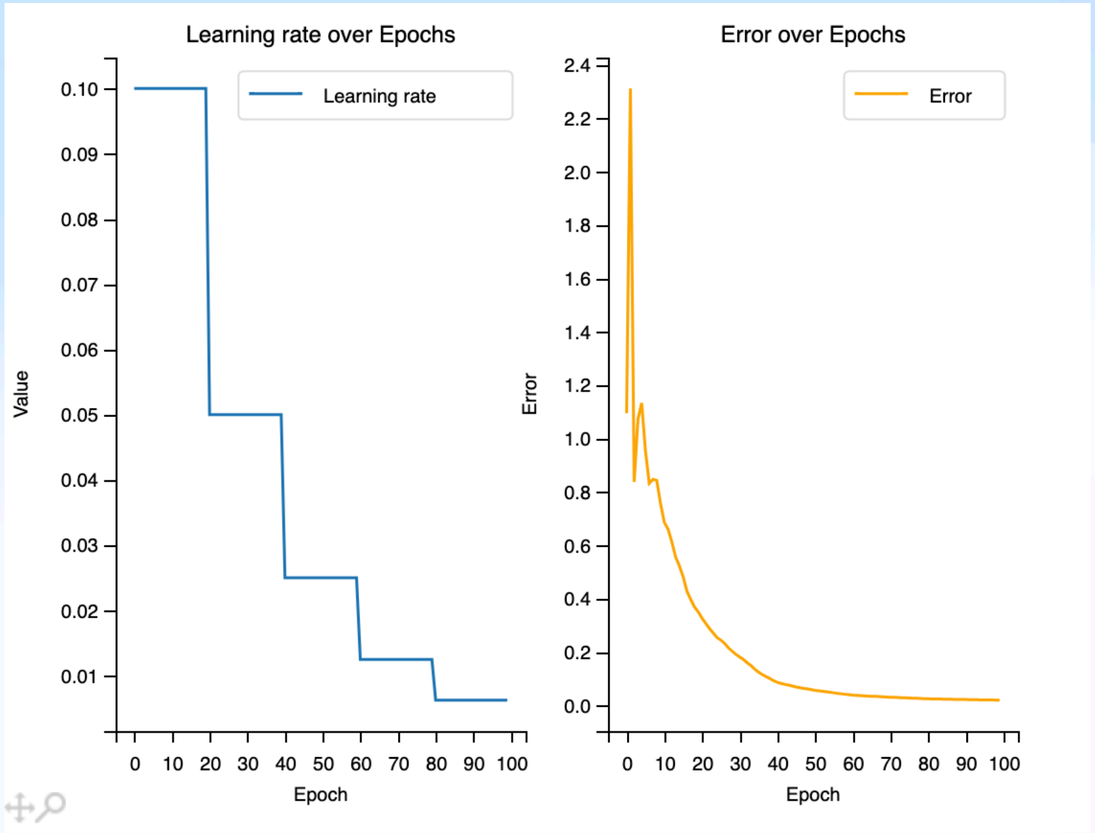
```
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.1, patience=10, verbose=True)
```

💡 Um aus lokalen Minima auszubrechen kann man auch gelegentlich schockartig die Lernrate kurz stark erhöhen (Alkohol;).

Adaptive learning rate

Die Lernrate kann man über Hyperparameter steuern

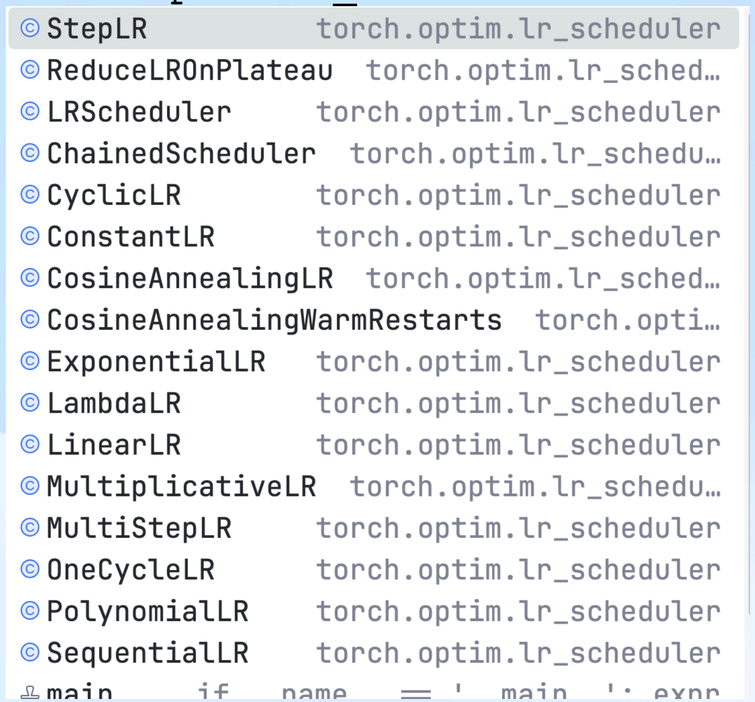
```
initial_learning_rate = 0.1
decay_at_steps = 20 # alle 20 Schritte:
learning_rate_decay_factor = 0.5 # halbiere lr

optimizer = optim.SGD(model.parameters(), lr=initial_learning_rate)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=decay_at_steps, gamma=learning_rate_decay_factor)
```



Adaptive learning rate

Wie immer gibt es verschiedenste Ansätze, je nach individuellen Anforderungen:

```
torch.optim.lr_scheduler.
```

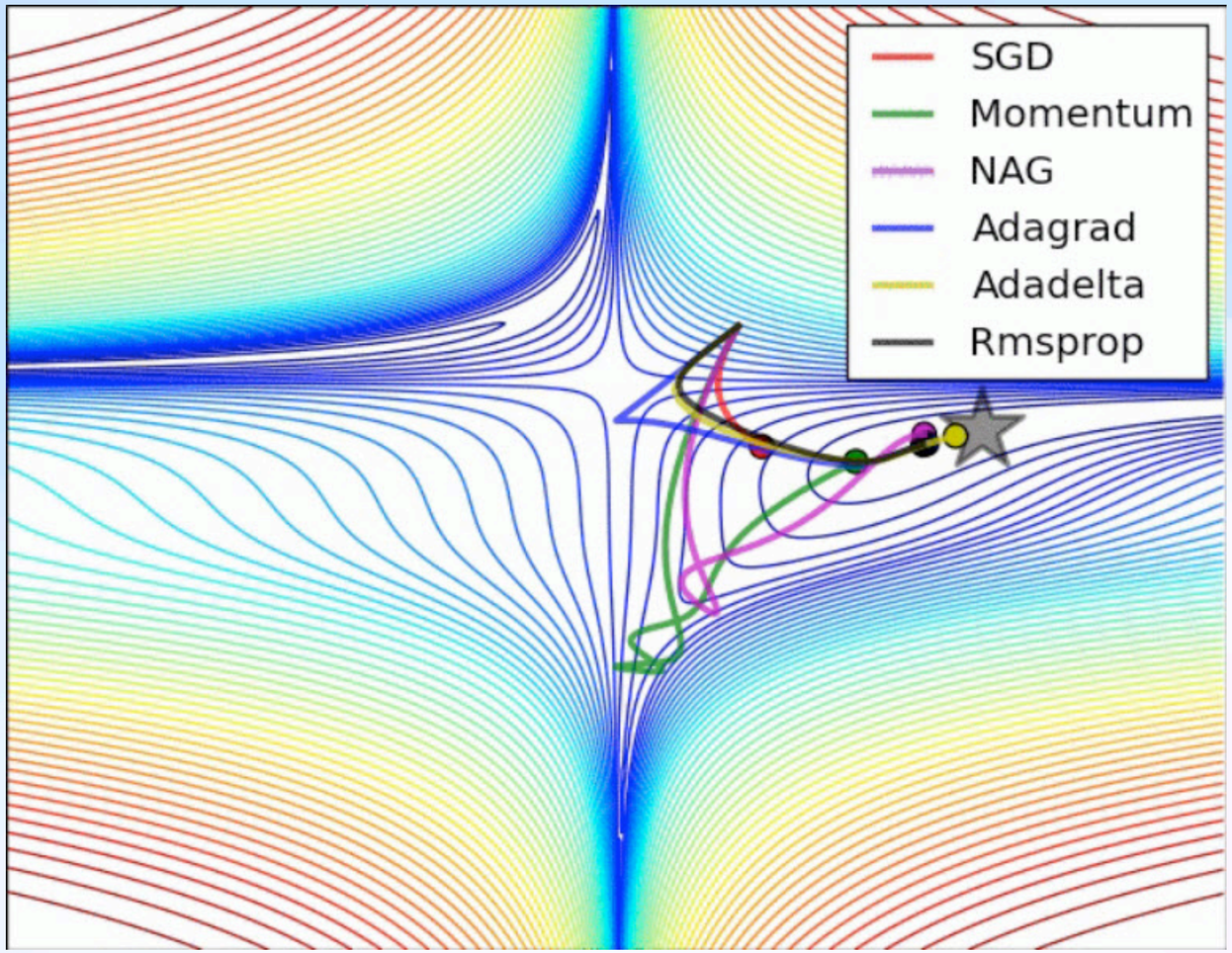


Aufgabe:

Verwende unser Perceptron für lineare Regression

Verschiedene Optimizer im Vergleich

Hier sieht man wie einfaches Momentum übers Ziel hinausschiesst:



AdaM (Adaptive Moment Estimation)

Beim naivem Momentum kann es sein, dass wir zu viel Momentum aufbauen uns 'aus der Kurve fliegen'.
Ausserdem ist die Schrittweite in alle Richtungen gleich groß.

Idee: wir passen die **Lernrate pro Gewicht** adaptiv an:

Dazu merken wir uns zusätzlich zum normalen Momentum noch das sogenannte **zweite Momentum:**

u = 0
u = gamma2 * u + **g^2 (element-weise quadrierter Gradient)**
wir verlieren Richtung und haben nur noch **Magnitude**

Ausserdem benutzen wir statt der Lernrate das gamma doppelt:
Damit steuern wir, wie groß der **Beitrag** vom alten Wert / vom neuen Wert sein soll:
vorher
v = gamma * v + **lr** * g =>
v = gamma * **v** + **(1 - gamma)** * **g**

u = gamma * u + (1 - gamma) * g^2

Wir teilen per Gewicht unsere Geschwindigkeit durch die Magnitude

Update der Gewichte:
lern_rate_per_w = lr / (√u + ε) (ε falls √u ≈ 0 ist)
w = w + v * lern_rate_per_w

Adam (Adaptive Moment Estimation)

Anstatt die Momentum-Methode selbst zu implementieren, verwenden wir einen sehr **erfolgreich**en Optimierer welcher in allen Bibliotheken implementiert ist: den **Adam Optimizer.**

Zusätzlich zum Momentum implementiert für uns der Adam Optimizer weitere Tricks:

Adam berechnet individuelle adaptive Lernraten für verschiedene Parameter anhand von Schätzungen der ersten und zweiten Momente der Gradienten:

- Adam: Verwendet sowohl den ersten Moment (Mittelwert) als auch den
  - **zweiten Moment** (quadrierte Mittelwerte "unkorrigierte **Varianz**") der Gradienten.
- Adam: **Bias-Korrektur**, um der Tatsache entgegenzuwirken, dass die anfänglichen Momentenschätzungen zu klein sind und so zu langsamen updates führen würden. "Anfangs-Schwung"
- Adam: Passt die **Lernraten** für jeden Parameter **individuell** an.

Diese Erweiterungen von SGD wurden in Papern vorgestellt nämlich als **AdaGrad** und **RMSProp**

Aufgabe:

Visualisiere MNist Daten, outlier und Gewichte

```
# weights = model.state_dict()['weight'].cpu().numpy()
# weights = weights.reshape(10, 28, 28)
img = mnist_test.data[0].numpy()
plt.imshow(img, cmap='gray')
plt.show()
```