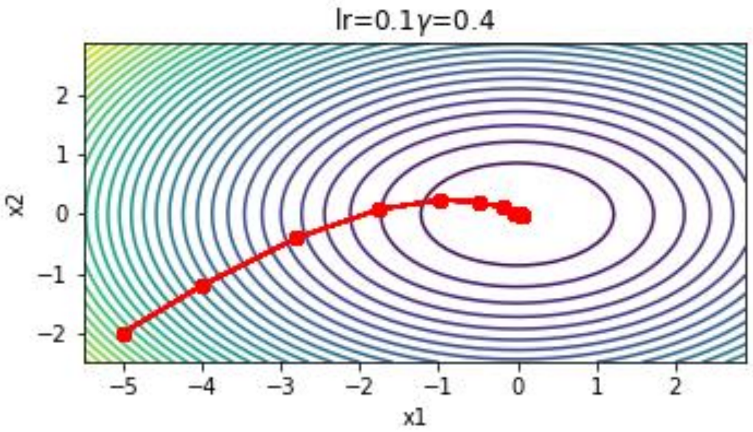
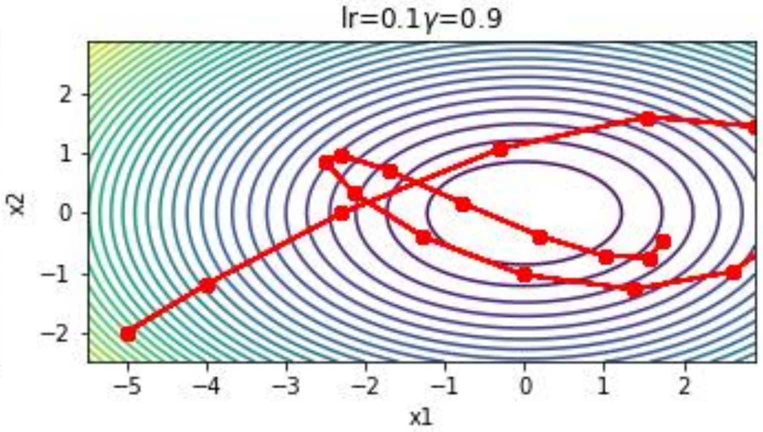
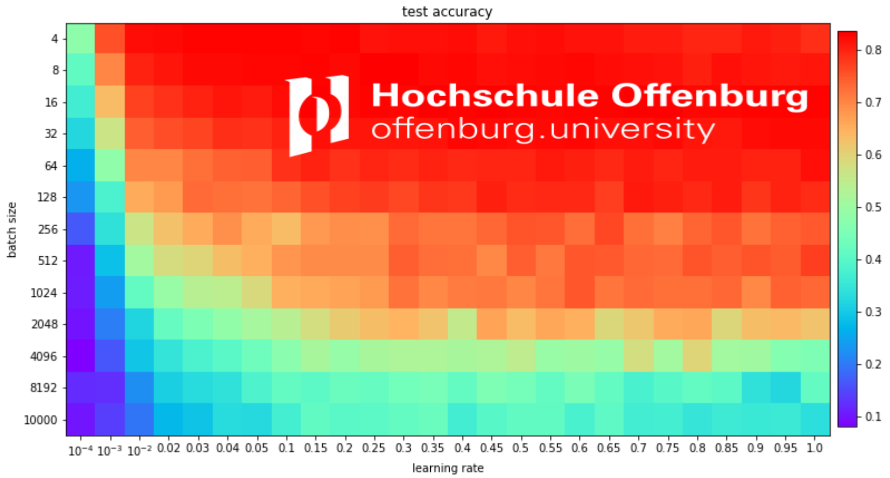






Übersicht über Stochastic Gradient Descent Varianten

Von Dennis Bystrow

Für Seminar Deep Learning WS-19/20

INFM – Fakultät Elektrotechnik, Medizintechnik und Informatik

Hochschule Offenburg

Inhalt

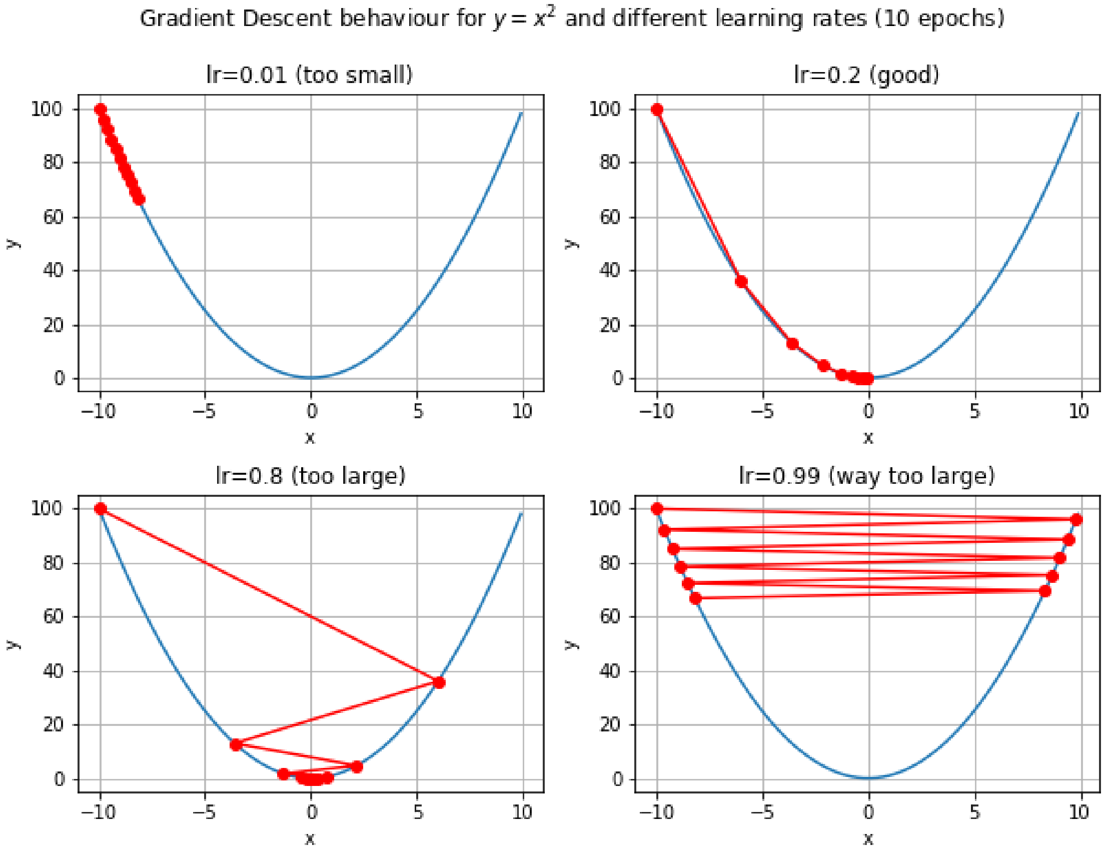
- Was ist Gradient Descent?
- (kurze) Einordnung im Deep Learning
- Stochastic Gradient Descent (SGD) Varianten + Experimente
  - Mini-Batch SGD
  - Momentum
  - Adagrad
  - RMSprop
- Zusammenfassung

Was ist Gradient Descent?

- Optimierungsproblem lösen, z.B. minimiere $\mathrm{f(x)} = x^2$
- Von einem Startpunkt aus der Ableitung (Gradient) folgen

$$x_{t+1} = x_t - \eta \nabla f(x_t)$$

- Learning rate η bestimmt die Schrittweite
- Hohe Ableitung an Stelle x_t → großer Schritt
- Kleine Ableitung an Stelle x_t → kleiner Schritt



Was ist Gradient Descent?

- Was wenn man mehrere Variablen hat?
- Z.B. minimiere: $f(x) = x_1^2 + 2x_2^2$

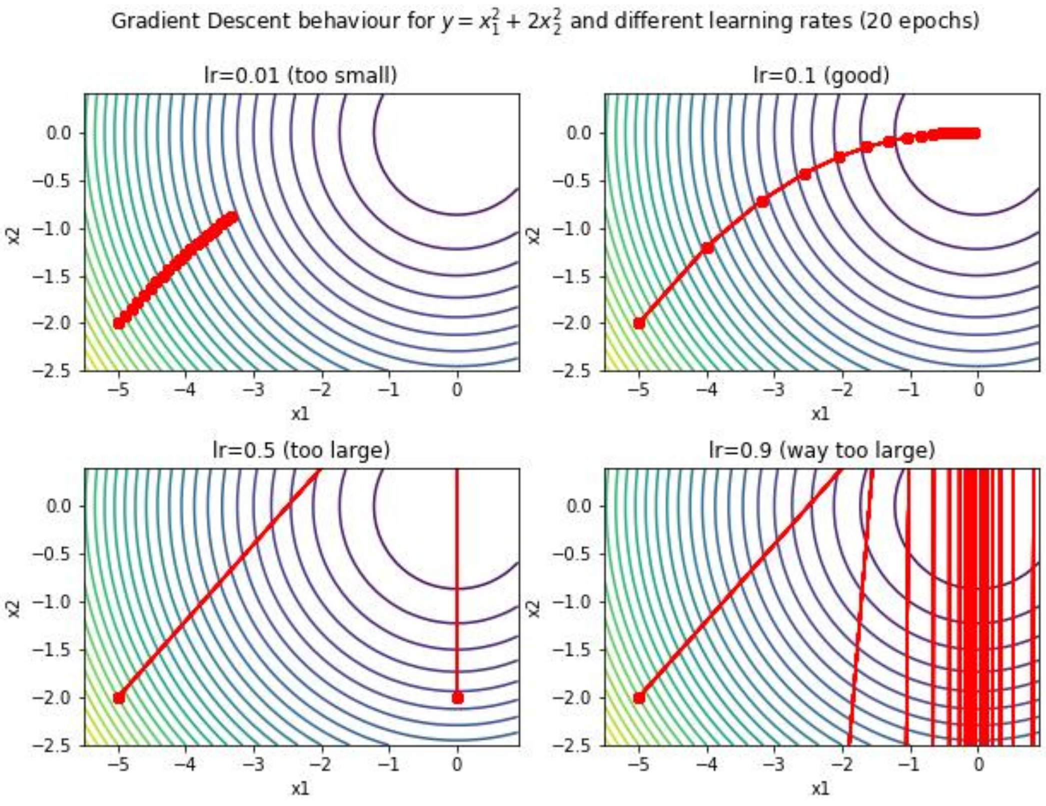
- Gradient: Vektor der partiellen Ableitungen

$$\nabla f = \begin{bmatrix} 2x_1 \\ 4x_2 \end{bmatrix}$$

- Wähle Startpunkt: z.B. P(x1 = -5 | x2 = -2)
- Parameterupdate mit P:

$$x_{1'} = x_1 - \eta \nabla_{x1} f(x_1)$$

$$x_{2'} = x_2 - \eta \nabla_{x2} f(x_2)$$



Einordnung beim Deep Learning

- Parameter sind die Menge der Gewichte des Netzwerks: θ (genau genommen auch bias)
- Cost Function (Objective Function) die optimiert werden soll: $J(\theta)$
- Gradient: $\nabla_\theta J(\theta)$

- Gradient Descent beim Training eines Netzwerks:
  1. Forward Pass mit einem Trainingsbild
  2. Gradienten werden bei der Backpropagation je Schicht berechnet
  3. Für die restlichen Trainingsbilder wiederholen → 1.

  Danach mit dem Gradienten (über alle Trainingsbilder) die Parameter Updaten (Gewichte und bias)

Mini-Batch Stochastic Gradient Descent

„Die Erkenntnis des SGDs ist, dass der Gradient ein Erwartungswert ist. Der Erwartungswert kann anhand einer kleinen Menge von Stichproben näherungsweise geschätzt werden.“

- Deep Learning, Goodfellow et al.

Trainingsbild: x
Label: y
SGD: $\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta; \mathrm{x}, \mathrm{y})$

- pro zufälligem Trainingsbild ein Update der Gewichte **ODER**
- Die Trainingsdaten in zufällige Batches aufteilen und pro Batch ein Update der Gewichte
- Typische Batch Size zwischen ~16 und ~500

→ Loss hat hohe Varianz

→ Geringere Varianz beim Loss. Man sampelt eine größere Menge. Effizienter als normaler GD

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```
[Externe Abb. 1]

Experimente

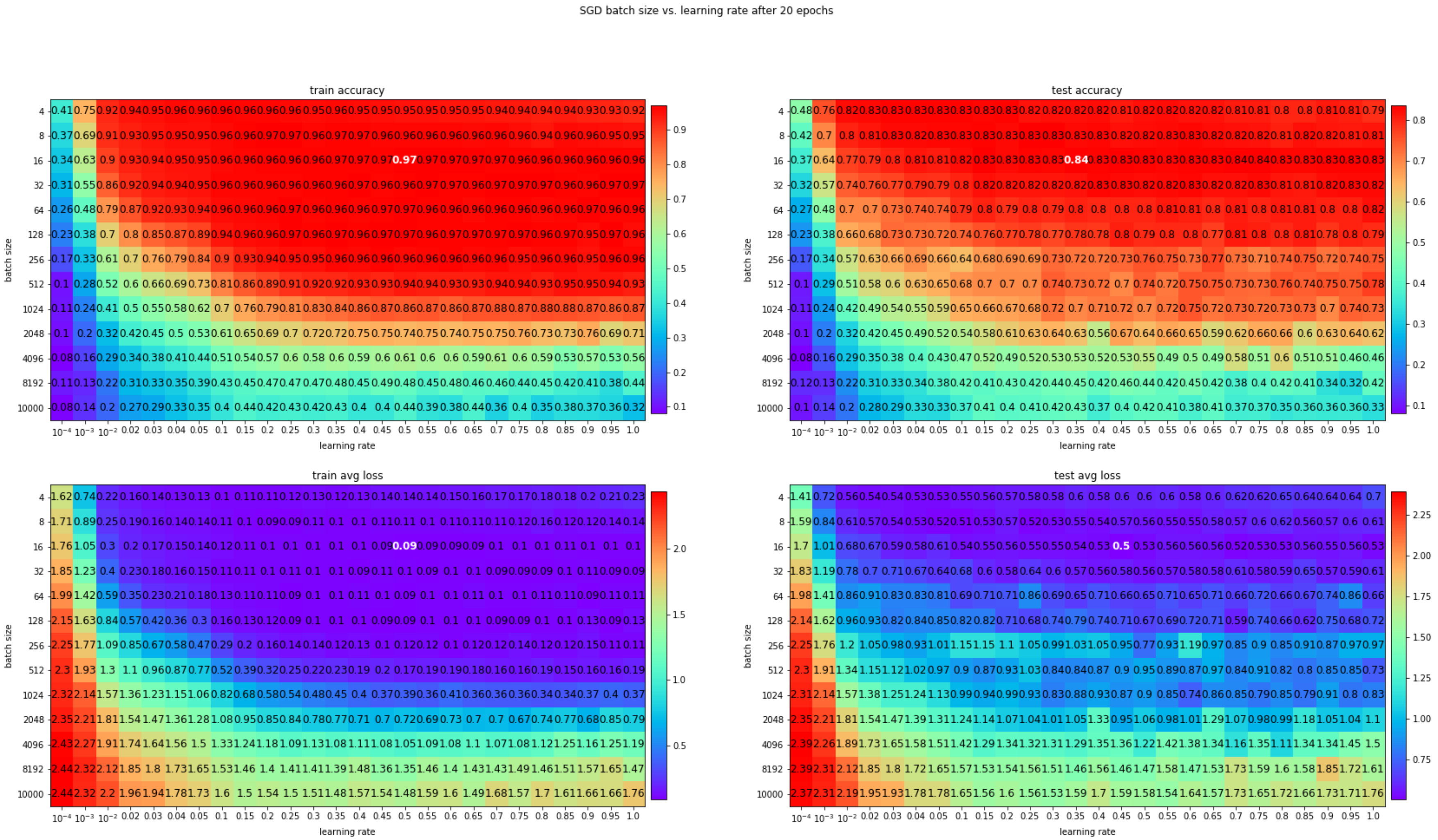
- Data Set: CIFAR-10
  - 32x32 Farbbilder mit 10 Kategorien (bird, cat, frog, ship,…)
  - 6‘000 Bilder pro Kategorie
  - 50‘000 Trainingsbilder und 10‘000 Testbilder
  - Bestes (bekannte) Ergebnis: 99.3% Test Accuracy (Kolesnikov et al. Dezember 2019)
- Verwendetes Netzwerk: Resnet20
- Verwendetes Framework: PyTorch
- Cost Function: CrossEntropyLoss
- Jeweils 20 Epochen trainiert
- Auf Google Colab

Learning Rate vs. Batch Size

Stetige Learning rates von 0.0001 bis 1.0
Batch sizes von 4 bis 10‘000

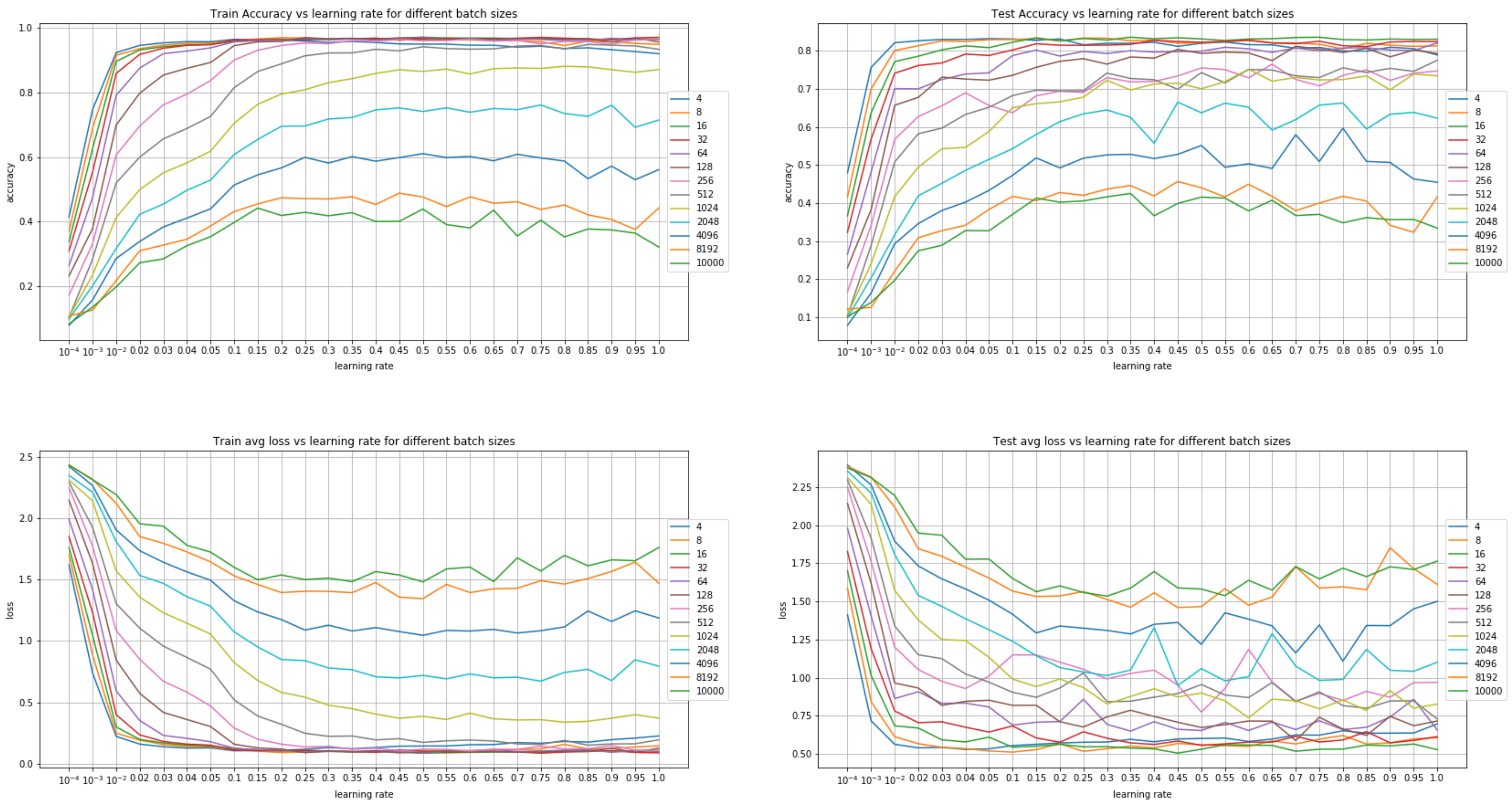
Beste Kombination (test acc)
→ lr = 0.35 batch_size = 16

→ Kleine Batch size und niedrigere learning rate führen zu besseren Ergebnissen als zu hohe batch size mit jeder anderen learning rate. Hier scheint die batch size eine höhere Auswirkung auf das Ergebnis zu haben als die Learning rate



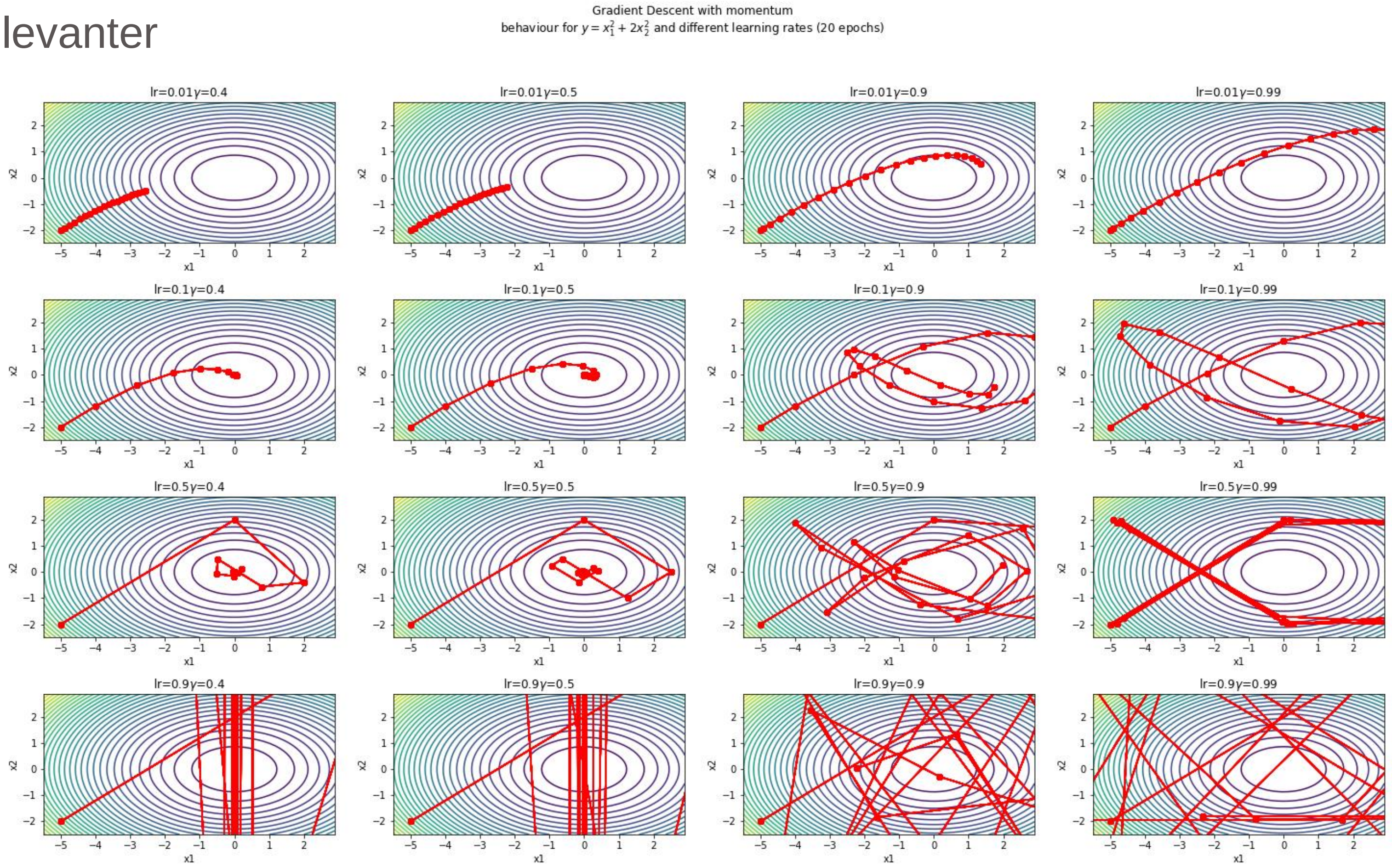
Learning Rate vs. Batch Size

SGD batch size vs learning rate after 20 epochs

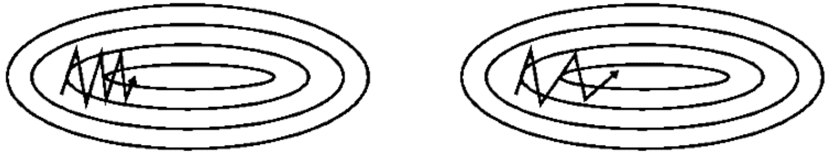


SGD mit Momentum

- Momentum gibt SGD eine Trägheit
- Parameter die beim vorigen Update relevanter waren (θ_{t-1}), werden noch relevanter
- $\theta_{t+1} = \theta_t - \gamma\theta_{t-1} + \eta\nabla_\theta J(\theta)$
- Hohes γ: Hohe Trägheit



Wann nützlich?
→ Bei stark ungleichen Gradienten je Parameter (schmale Schlucht) und Plateau



[Externe Abb. 2]

Experimente

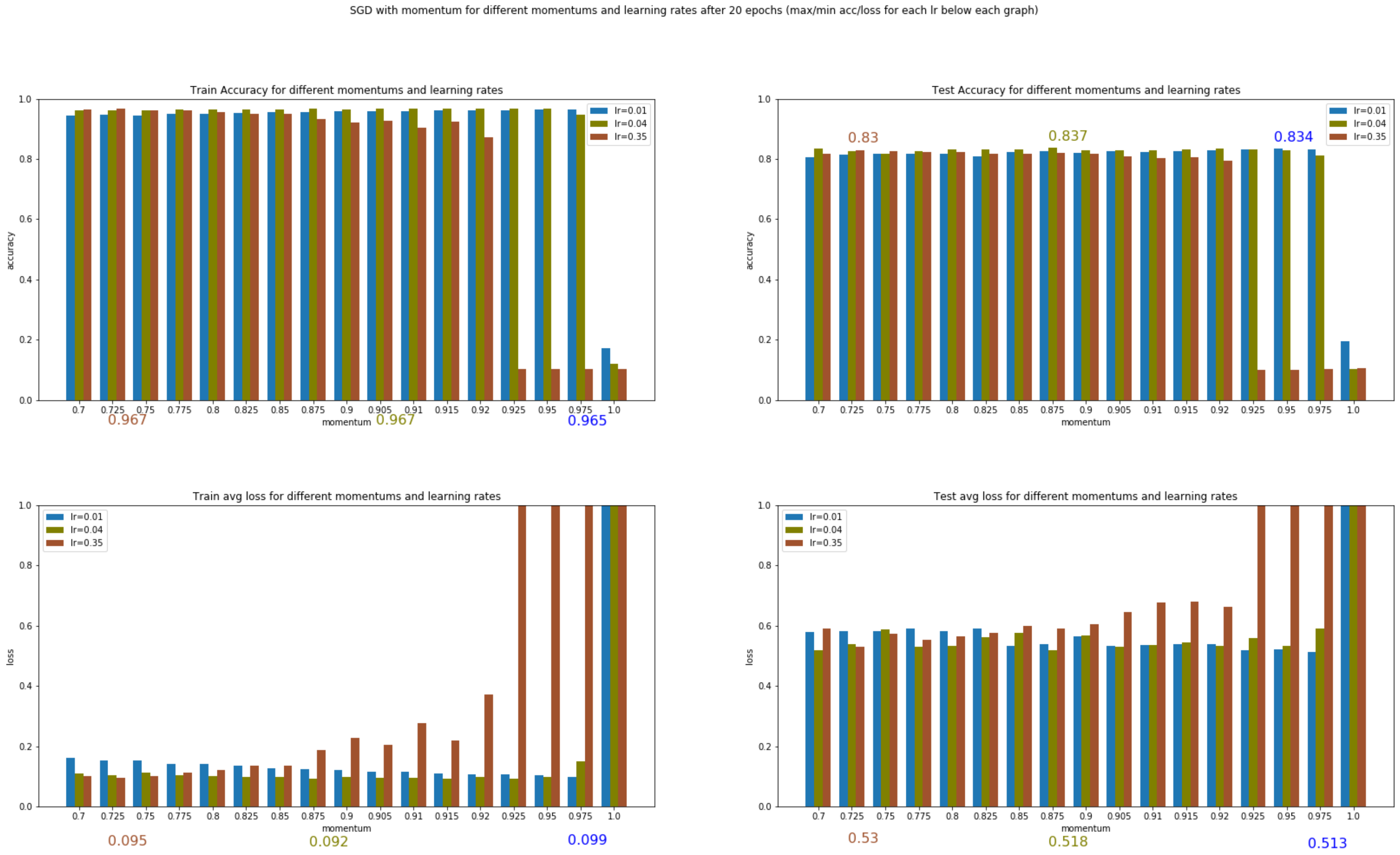
- Learning rates: 0.01, 0.04, 0.35
- Gamma: 0.7 bis 1.0

- Beste Kombination (test acc)
- → lr = 0.04 gamma = 0.875

→ Ein hohes Gamma (hier 0.925) und relativ hohe Learning rate (hier 0.35) sorgen für sehr schlechtes Ergebnis

→ Also zu viel Trägheit

→ Geringere Learning rate und Gamma sehr nahe 0.9 machen am meisten Sinn



AdaGrad

- Was wenn manche Parameter weniger wichtig sind als andere?
- Was bei einem dünn besetzten Datensatz?

→ **Learning Rate pro Parameter anpassen**

- Seltene Features → Parameter erhält größeres Update
- Häufige Features → Parameter erhält kleineres Update

→ Man akkumuliert die Summe der bisherigen quadrierten Gradienten für einen Parameter: G_t

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

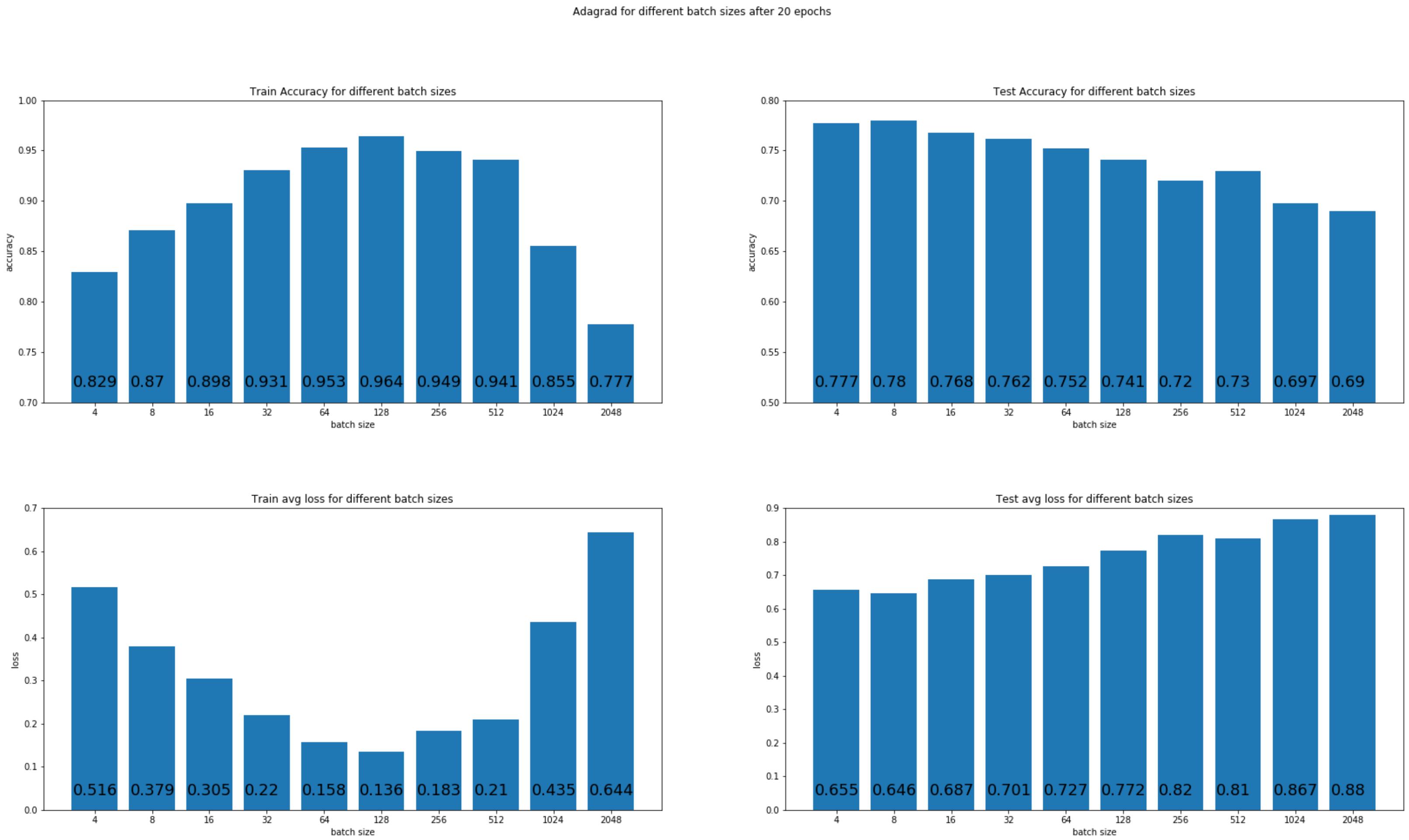
- Learning rate muss nicht mehr getunt werden
- „lr = 0.01 sollte idR. ausreichen“
- Problem: Summe steigt immer weiter und learning rate wird extrem klein

  → Kann deshalb irgendwann nicht mehr weiterlernen

Experimente

- Mit PyTorch Defaults gestartet
- lr = 0.01
- batch sizes von 4 bis 2048 getestet

- Beste Kombination (test acc)
- → batch_size = 8



→ Schlechtere accuracy erreicht als erwartet. Weitere Tests mit anderen initialen Learning Rates.

→ Datensatz ist gleichverteilt. Kann per Parameter adapted learning rate hier überhaupt einen signifikanten Effekt haben?

RMSProp

- Beseitigt AdaGrads Problem aussterbender learning rates
- → bisherige quadrierte Gradienten werden nicht einfach nur aufsummiert
- → Mit exponential weighted moving average anstatt nur der Summe G_t der quadrierten Gradienten

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

$$E[g^2]_t = 0.9 E[g^2]_{t-1} + 0.1 g_t^2$$

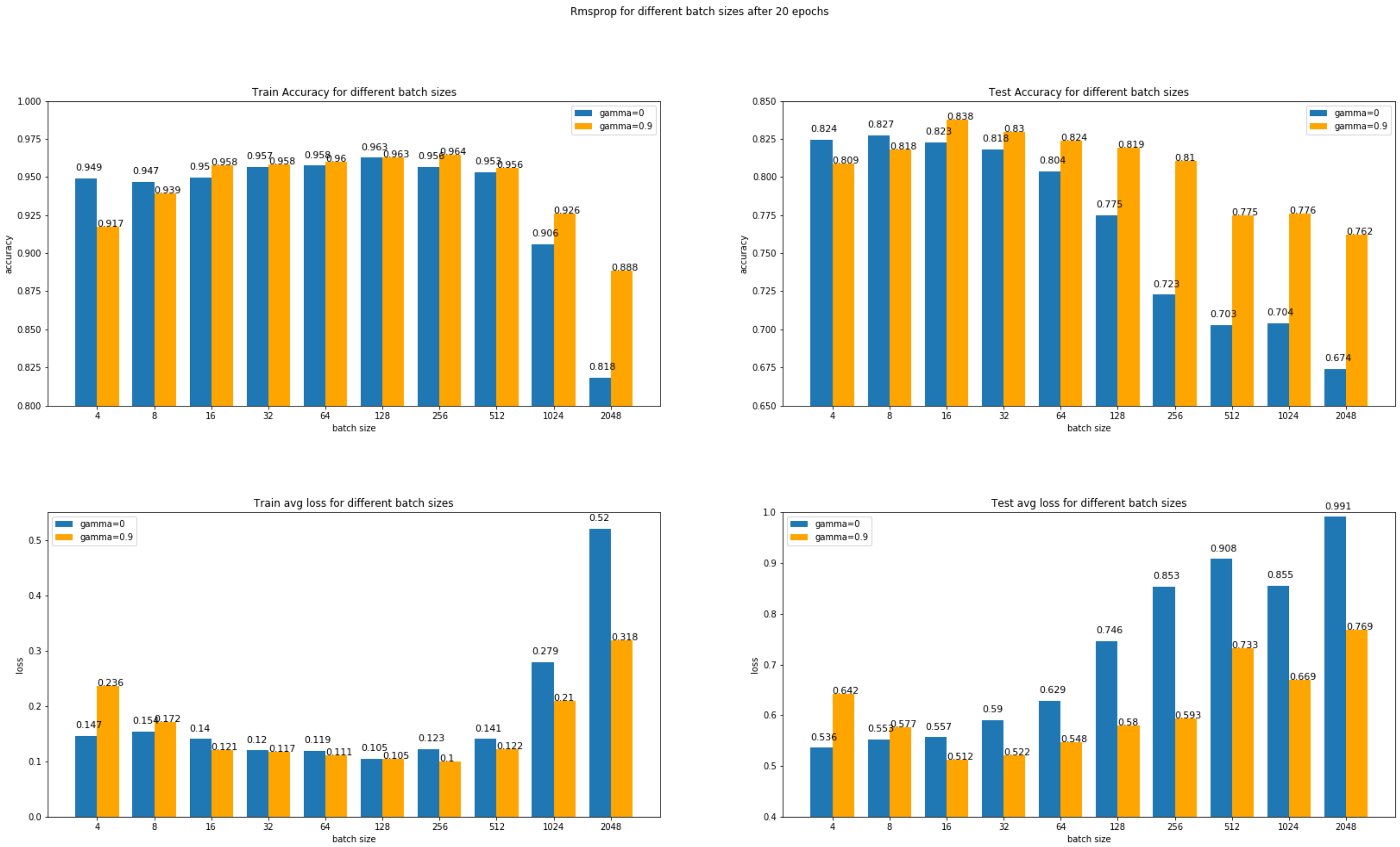
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

- Standardwerte für gamma und learning rate:
  - $\gamma = 0.9$
  - $\eta = 0.001$
- Wenn $\gamma = 0$ dann nur exponential moving average

Experimente

- lr = 0.001
- batch sizes von 4 bis 2048
- γ: 0 und 0.9
  - Nicht gewichtetes vs. gewichtetes Mittel
- Beste Kombination (test acc):
- $\gamma = 0.9$ batch_size = 16

→ Gewichtetes Mittel ($\gamma = 0.9$) sorgt bei höherer batch size für sehr viel bessere Ergebnisse als ungewichtetes Mittel



Zusammenfassung der Ergebnisse (test acc) für CIFAR-10 Datensatz

- Insgesamt 440 Trainingsläufe über jeweils 20 Epochen
  - **Momentum test acc 0.8412: lr = 0.04 batch_size = 16 gamma = 0.9**
  - RMSprop test acc 0.8377: lr = 0.001 gamma = 0.9
  - SGD test acc 0.8363: lr = 0.35 batch_size = 16
  - AdaGrad test acc 0.7799: batch_size = 8

- Nach 100 Epochen mit den jeweils besten Parametern
  - **Momentum mit LR-Scheduling (Cosine Annealling) 0.8564**
  - Momentum 0.8425
  - SGD 0.8388
  - RMSprop 0.8319
  - AdaGrad 0.7720

Vielen Dank für die Aufmerksamkeit!

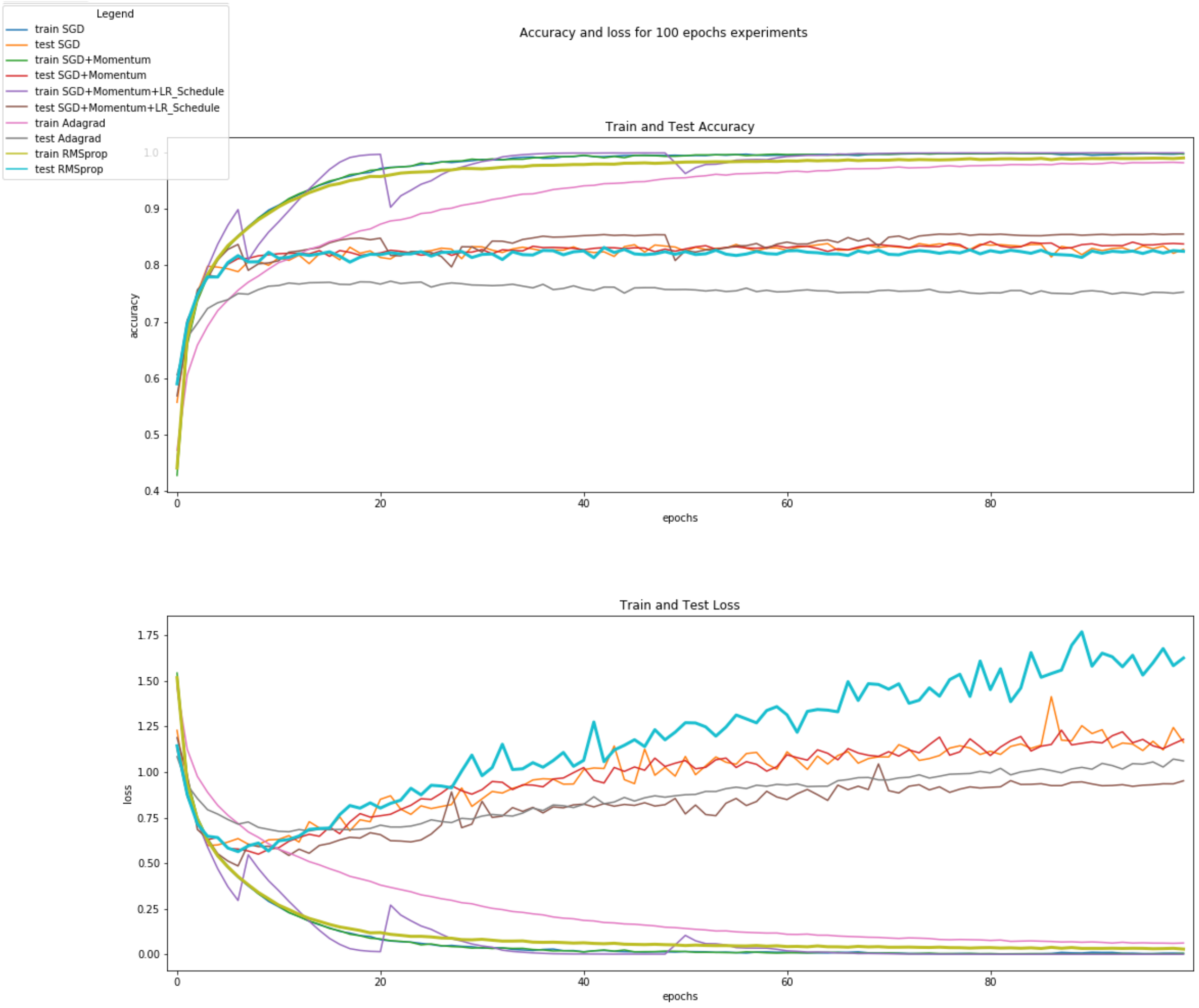
Öffentliches github repository: https://github.com/Dens49/seminar-deeplearning-sgd

(noch nicht vollständig)

Weitere Optimizer

- NAG (Nesterov Accelerated Gradient): „smartes“ Momentum
- AdaDelta: Extension to AdaGrad, ähnlich wie RMSprop aber unabhängig davon entwickelt
- Adam: Im Prinzip RmsProp + Momentum
- AdamW
- Nadam
- AdaMax
- Rprop
- AMSGrad
- SparseAdam
- ASGD
- LBFGS

Weitere Experimente
100 Epochen für SGD, Momentum, LR-Schedule, AdaGrad, RMSprop



Quellen, Formeln, Externe Abbildungen

- https://ruder.io/optimizing-gradient-descent/ als Hauptquelle
- https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1
- https://paperswithcode.com/sota/image-classification-on-cifar-10
- https://arxiv.org/abs/1912.11370v1 Paper zum CIFAR-10 Ergebnis
- https://courses.d2l.ai/berkeley-stat-157/units/optimization.html Erklärungen und Code zu (nicht-) konvexer Optimierung
- https://courses.d2l.ai/berkeley-stat-157/units/adam.html Erklärungen und Code zu Gradient Descent, Momentum, AdaGrad, RMSprop, Adam
- https://www.cs.toronto.edu/~kriz/cifar.html CIFAR-10 Data Set
- https://www.deeplearningbook.org/
- https://de.coursera.org/lecture/deep-neural-network/rmsprop-BhJlm
- Externe Abb. 1: https://ruder.io/optimizing-gradient-descent/
- Externe Abb. 2: https://ruder.io/content/images/2015/12/without_momentum.gif https://ruder.io/content/images/2015/12/with_momentum.gif
- Sämtliche mathematische Formeln sind übernommen aus oder orientieren sich an: https://ruder.io/optimizing-gradient-descent/

Adam

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\, g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)\, g_t^2$$

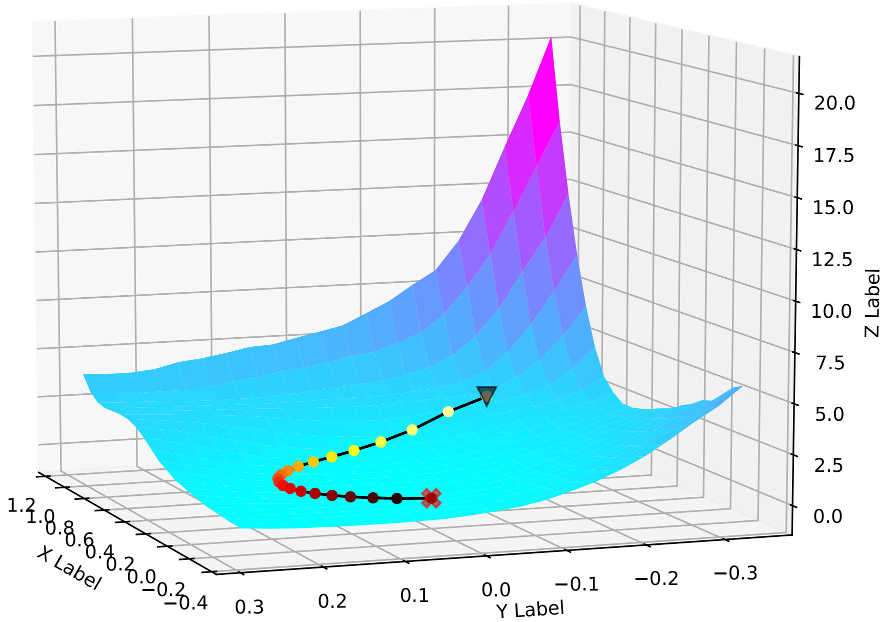
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

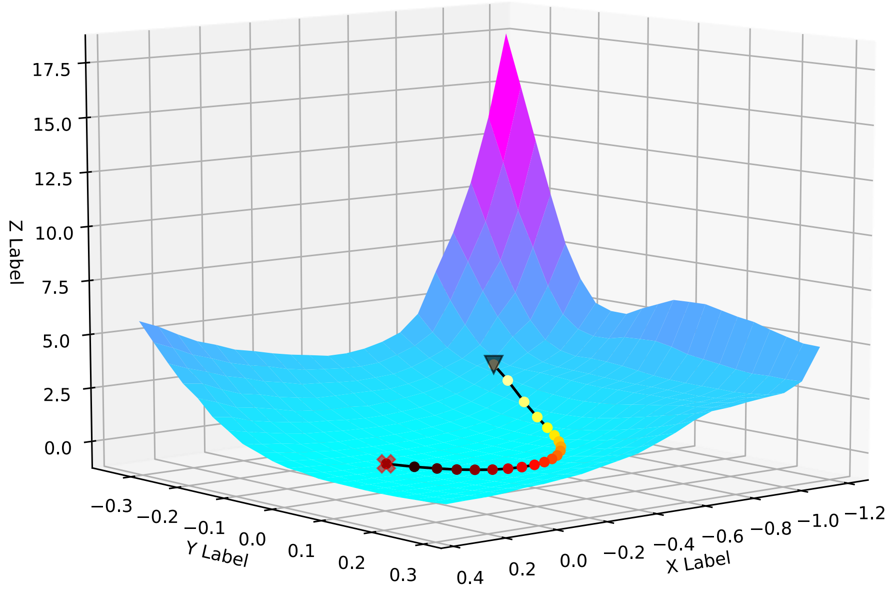
Visualisierungen mit GradVis

- https://github.com/cc-hpc-itwm/GradVis
- Visualisiert loss surface und Pfad dadurch zwei- und dreidimensional

SGD batch_size=16 lr=0.35



Momentum batch_size=16 lr=0.04 momentum=0.9



Cosine Annealling Learning Rate Schedule