Deep Learning

Grundlagen Grundbegriffe Grundübungen

Alfatraining Kurs 2024-08 Dozent: Karsten Flügge

Themen des Tages

Tag 3

Tensoren
Bibliotheken
Regression vs. Klassifikation
Lernkurven

Tensoren

Tensoren verallgemeinern das Konzept von 1-D Vektoren und 2-D Matrizen auf 3-D und n-dimensionale Daten

0-D "einzelne Zahl", Faktor
1-D Vektoren
2-D Matrizen, z.B. Bild (Höhe*Breite)
3-D Arrays von Matrizen, z.B. Bild × Farbe oder SW-Bild × Zeit
4-D Arrays von Arrays von … z.B. Bild × Farbe × Zeit = Video
5-D z.B. 'batch' Liste von Videos

💡 In Deep Learning werden ALLE Daten zum Berechnen in Tensoren umgewandelt
(Texte -> Worte -> Buchstaben / Tokens -> Zahlen)

💡 Generell üblich in AI: Trainingsdaten werden zu '**batches**' "Stapel" zusammengefasst um mehrere Trainingselemente gleichzeitig parallel zu trainieren. Damit erhält man immer aus n-Tensoren **n+1**-Tensoren

💡 In der Mathematik haben Tensoren noch Zusatzstrukturen/Axiome, welche in der KI aber meist nicht benutzt oder benötigt werden. Der Begriff kommt aus der Physik, wo man in Materialien in jedem Punkt mehrere Spannungen (Tensionen) in mehrere Richtungen haben kann.

Tensoren

Die **Werte** eines Tensors nennt man **Parameter**

Diese können **fest** sein oder (temporär) variabel also **trainierbar.**

Die **Ergebnisse** einer Berechungs-Schicht nennt man **Aktivierungen** oder **Zustände**

Bei der **Initialisierung** erhalten die Tensoren **zufällige** Werte, allerdings kann die Wahl der Startwerte (grossen) Einfluss auf das Lernverhalten haben. Standard: Normal / Gleichverteilung geteilt durch Länge des Tensors.

💡 In tieferen Schichten der Netze sind Tensoren oft völlig abstrakt, dass heisst was genau die **Parameter** der Tensoren darstellen ist bei Konstruktion unklar, die **Zustände** lassen sich aber gegebenenfalls rekonstruieren "Grossmutterneuron"

Tensoren

Warum heisst tensorflow **tensor**flow?

Tensoren kennen wir:
0-d "Skalar" Zahl
1-d "Vektor" Liste von Zahlen
2-d "Matrizen" Liste von Liste von Zahlen
3-d "Tensor" "Blob"
4-d …

Beispiel von Tensoren
2-d **Bild** Breite*Höhe
3-d Batch **Stapel** von Schwarzweiss bildern
2-d Blätter * Breite + Blüten * Breite
3-d Bild*Farbe "Farb Kanal eigene Dimension"
4-d Batch **Stapel** von Farbbildern
4-d **Video** Farbbild*Zeit
4x-d Video Farbbild*Zeit + Ton ?

R^n + R^m = R^(n+m) (1,2,3)+(4,5) = (1,2,3,4,5) Basis e1… e5
R^n * R^m = R^(n*m) wie in einer Matrix Basis e11 e12 e13 e21 e22 e23

Bibliotheken

Warum Bibliotheken?

Warum nicht 'einfach' eigenes Framework schreiben?

Bibliotheken

Warum Bibliotheken?

Optimierung der Rechenoptionen, automatisch **parallel** auf GPU
Standardisierung der Rechenoperationen
Durch**getestet** auf Fehler

Convenience:
Viele Schritte sind **abstrahiert** und implizit, z.B.
Implizite Initialisierung

💡 Trotzdem kann es sehr lehrreich und sogar nützlich sein, Aufgaben ohne Bibliotheken zu lösen. Z.B. hat Karpathy GPT 2 in einer einzigen Datei nachimplementiert.

Bibliotheken

Welche Bibliotheken?

- **Basis Bibliotheken:**
  - **numpy** für Mathematik (Lineare Algebra)
  - matplotlib für **Visualisierung**

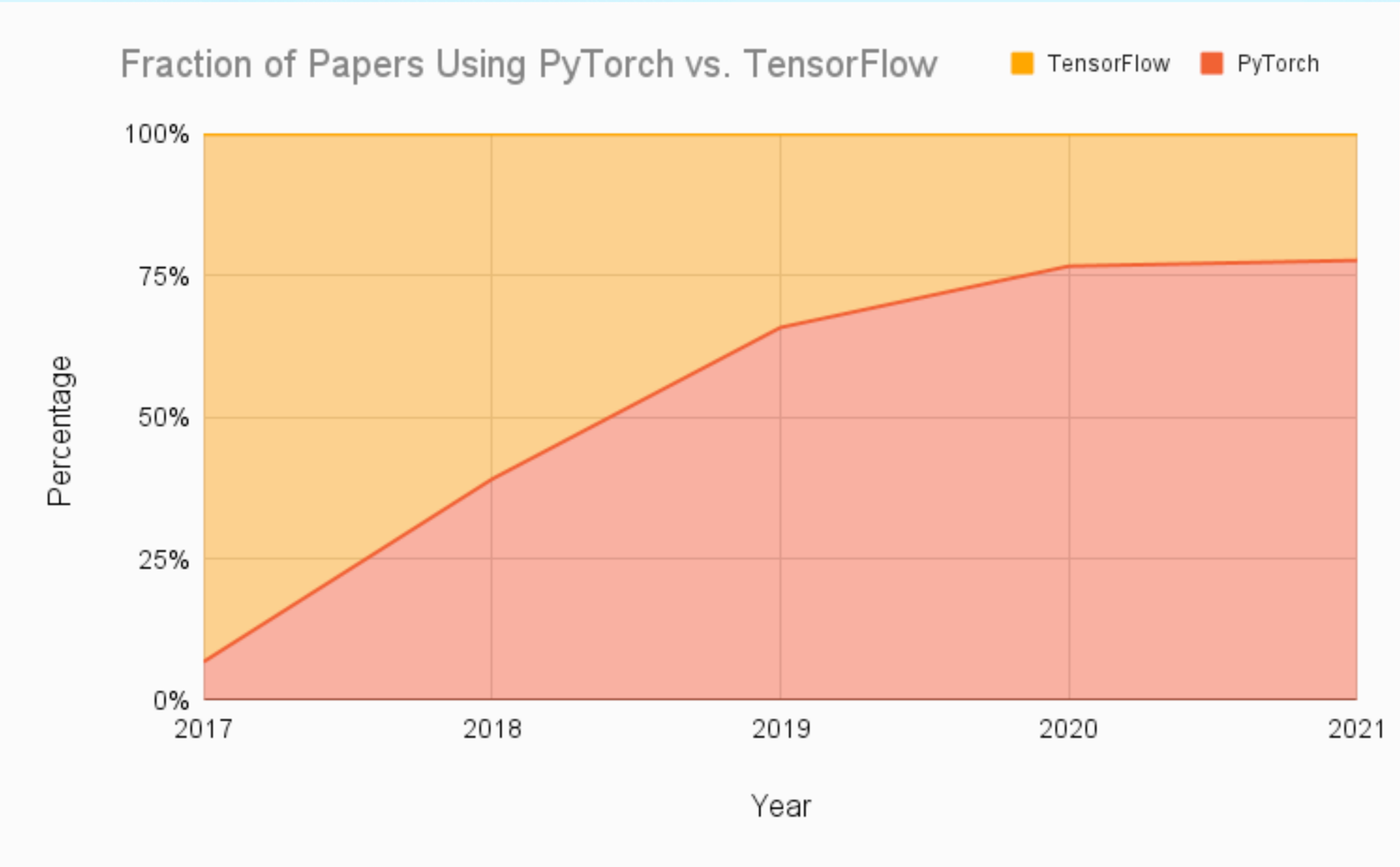
- Veraltete / spezielle Bibliotheken:
  - pandas für Datenmanipulation und Analyse
  - scipy für wissenschaftliche Berechnungen
  - sklearn, ...
  - caffe, theano (wird nicht mehr aktiv entwickelt)

⚠ Diese wollen wir vermeiden und statt dessen alles mit Deep Learning lösen:

- **Deep Learning Bibliotheken:**
  - **pytorch (meta) Linux Foundation**
  - **tensorflow 1.0 => 2.0 (google)**
  - **keras (wrapper) spacy NLP cinc**
  - **jax**

- Speziell
  - **transformers** von Hugging Face (convenience für NLP, LLM, textuelle Netze)

Bibliotheken



2024 wohl > 95% pytorch
Zukunft: jax? mojo?

Bibliotheken

Warum hat **pytorch** tensorflow praktisch abgelöst?

Imperative Programmierung

tensorflow hat ursprünglich einen **Graphen** erzeugt, welcher dann auf der GPU ausgeführt wurde.
mittlerweile kann man aber in pytorch und tensorflow frei wählen,
ob das Modell **direkt** imperativ oder indirekt (per graph) berechnet/ausgeführt werden soll.

Vermurkste tensorflow **APIs**

Der Übergang von tensorflow 1 zu tensorflow 2 war eine Katastrophe.
Die API hat sich ständig geändert. Keras wurde integriert und wieder extegriert ...

Tensor Bibliotheken

tensorflow und py**torch** haben **Tensoren** als Grundbaustein

Grundlegende Operationen:
Anlegen

```
import torch # PyTorch ! pip install torch !
t0 = torch.tensor(1.0) # Scalar
t1 = torch.tensor([1.0, 2.0, 3.0]) # Vector
data = [[1, 2],[3, 4]]
t2 = torch.tensor(data) # Matrix …
```

Umwandeln (von nach numpy / array)

```
torch.tensor(np_array) oder
torch.from_numpy(np_array)
tensor.numpy()
```

Form / Shape

```
print(f"Shape of tensor: {tensor.shape}")
```

Datentyp

```
print(f"Datatype of tensor: {tensor.dtype}")
```

Operationen wie in numpy:
Indexing tensor[0], tensor[:, 0], tensor[..., -1] ...
Slicing extraktion von Subdaten
Reordering ändern der shape
Joining torch.**cat**([tensor, tensor, tensor], dim=1)
torch.**stack** along a ***new*** dimension

Named Tensors

💡 **Benamte Tensoren in pytorch**

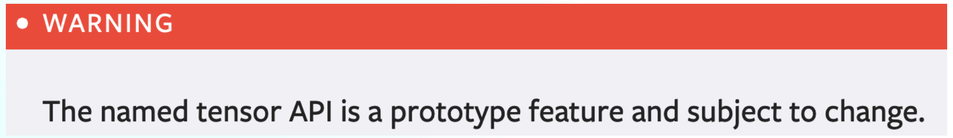
Ähnlich **Einheiten / Dimensionen** in der Physik kann man Tensoren auch einen Namen mitgeben.

💡 Idealerweise sollten sich damit viele Fehler vermeiden lassen

```
torch.zeros(2, 3).names = ('width', 'height')
# https://pytorch.org/docs/stable/named_tensor.html
```

💡 Automatisches '**alignment**' statt 'manuelles herumschaufeln von Dimensionen'

Leider ist dieses Feature noch nicht komplett konzeptioniert und implementiert:



Element / Broadcasting

Broadcasting
t * 5
Behält die shape!

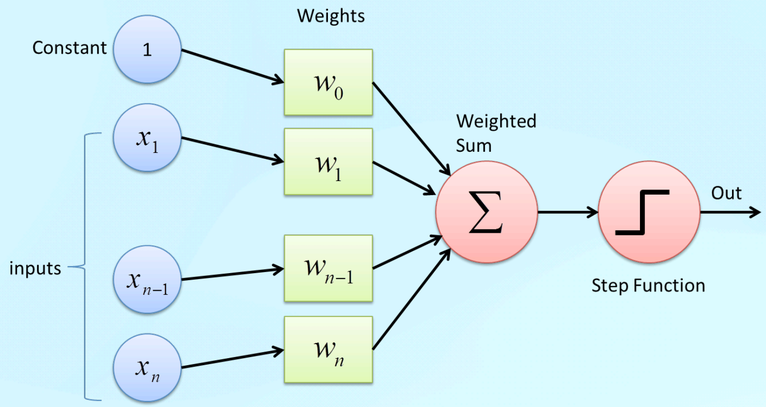
element-wise product
t0 * t1
Behält die shape!

tensor multiplication, contraction
wie Matrix Multipliation:
t0 @ t1

Ändert die shape!

Breiter Input : Batch

Batch: berechne die Aktivierung von mehreren Inputs gleichzeitig!



Stapel von Input Vektoren wird parallel ans Netz gegeben.

Damit y = W*X mit Inputs als Matrix gesammelt (statt y = W*x input vector x)

Oder allgemeiner:
n-dimensionale Trainingsdaten werden als
(n+1)-dimensionale Trainingsgruppen (sogenannter batch) zusammengefasst

⚠ **Achtung, unsere Bibliotheken operieren implizit immer auf batches!**
Dass heisst, wir müssen unsere Trainingsdaten immer als Gruppe übergeben.

nicht model(x1) sondern model([x1,x2 …]) äußere Liste ist batch

Breiter Input : Batch

Batch: berechne die Aktivierung von mehreren Inputs gleichzeitig!
Stapel von Input Vektoren wird parallel ans Netz gegeben.

n-dimensionale Trainingsdaten werden als
(n+1)-dimensionale Trainingsgruppen (sogenannter batch) zusammengefasst

⚠ **Achtung, unsere Bibliotheken operieren implizit immer auf batches!**
Dass heisst, wir müssen unsere Trainingsdaten immer als Gruppe übergeben.

nicht model(x1) sondern model([x1,x2 …]) äußere Liste ist batch

```
# -1 heisst automatisch shape schlussfolgern
# 1 heisst listen der Länge 1
x=[x1, x2, x3, ...]
x_view = x.view(-1, 1) # wrap each element into a list [[x1], [x2], [x3], ...]
same :
x_view = x.view(len(x), 1) # wrap each element into a list [[x1], [x2], [x3], ...]

im wesentlichen das selbe wie reshape(-1,1)
```

Pause

Perceptron in libraries

Unser Perceptron heisst

in pytorch **nn.Linear** ohne Aktivierung
in tensorflow **layers.Dense** **(from tensorflow.keras import layers)**

layers.Dense(FEATURES, activation='relu') mit Aktivierung immer als Parameter

```
layers.Dense(FEATURES, activation=tf.nn.relu(…), input_shape=(1,)),  ReLU activation als String oder Objekt
```

Das sind **generelle Synonyme** fürs **Perceptron:**

Fully Connected, Linear, Dense … Layer (Schicht)

Aufgabe

pytorch und tensorflow

sine.py trainieren lassen!

sine.py als shared colab:
https://colab.research.google.com/drive/1SUz0_rCMvnCe4xV2yxM6e8-lr1Z-sVuK

pytorch und tensorflow tensoren anlegen und manipulieren

Was sind Unterschiede?

tensorflow Funktionen meist **ausserhalb** tf.reshape(tensor,shape=…)
pytorch Funktionen meist **innerhalb** tensor.reshape(…)

```
tensor.T   transpose in pytorch
t0 @ t1    matmul in pytorch!
```

Regression vs. Klassifikation

Regression:
direkte Vorhersage von einem Wert

Klassifikation:
Vorhersage von einer Klasse
(z.B. einem Wertebereich 25-30 Jahre)

Regression vs. Klassifikation

Regression: Eine Methode zur Vorhersage **kontinuierlicher Werte**. Das Ziel ist es, eine Funktion zu finden, die eine Beziehung zwischen Eingabedaten (Features) und einem kontinuierlichen Zielwert modelliert.

- Beispiel: Vorhersage des Preises eines Hauses basierend auf Größe, Lage, etc.

Klassifikation: Eine Methode zur Vorhersage **diskreter Klassenlabels**. Das Ziel ist es, Eingabedaten einer oder mehreren vordefinierten Kategorien zuzuordnen.

- Beispiel: Klassifikation von E-Mails in „Spam“ oder „Kein Spam“.

Regression vs. Klassifikation

Vorteile und Nachteile

- **Regression**:
  - Vorteile: Gut geeignet für Probleme mit kontinuierlichen Zielwerten, flexible Modellierung von Beziehungen.
  - Nachteile: Kann schwer zu interpretieren sein, besonders bei komplexen Modellen; Anfällig für Ausreißer.
  - schwerer zu trainieren

- **Klassifikation**:
  - Vorteile: Klare Entscheidung zwischen Klassen; leichter zu interpretieren; oft effizienter bei eindeutigen Kategorien.
  - Nachteile: Funktioniert nicht gut bei kontinuierlichen Zielwerten; kann bei einer großen Anzahl von Klassen komplizierter werden.

Regression vs. Klassifikation

Beispiele und Anwendungen

- **Regression**:
  - Preisvorhersage (z.B. Immobilienpreise, Aktienkurse)
  - Vorhersage der Temperatur
  - Berechnung von Umsatzprognosen

- **Klassifikation**:
  - Bildklassifikation (z.B. Katze vs. Hund)
  - Krankheitsdiagnose (z.B. gesund vs. krank)
  - Sentiment-Analyse (z.B. positive vs. negative Bewertung)

- **Ausserhalb des Schemas:**
- Bilderstellung
- Textverarbeitung …

Regression vs. Klassifikation

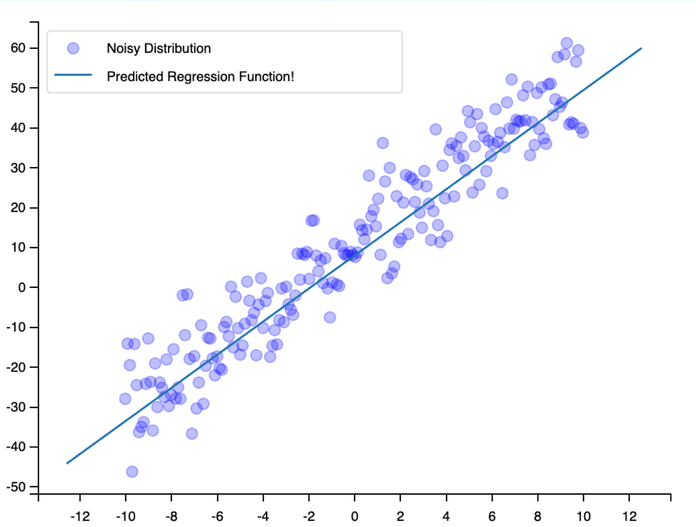
Regression:
Haben wir bereits gemacht

Klassifikation:
über sogenannte Softmax Aktivierung und passendem Fehlermaß: CrossEntropy

Linear Regression

Nur anwenden was wir bisher gemacht haben:

model(x) = w*x + w0 (nicht mal Vektor!)



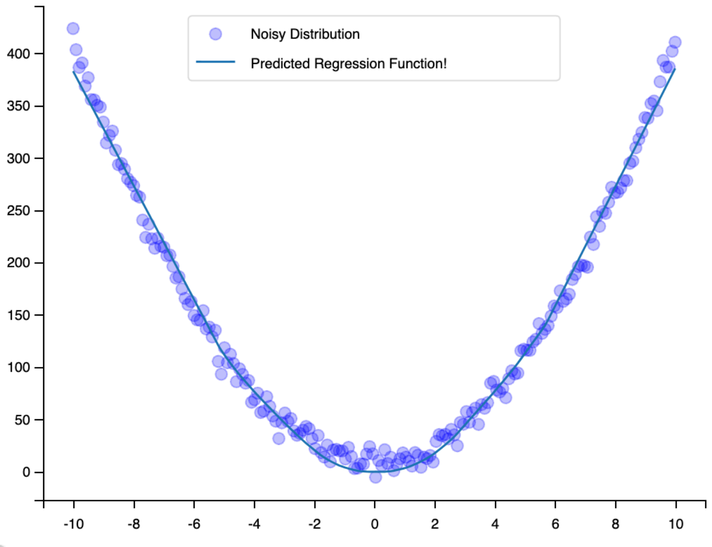
trivial_regression.py

Nonlinear Regression

Nur anwenden was wir bisher gemacht haben:

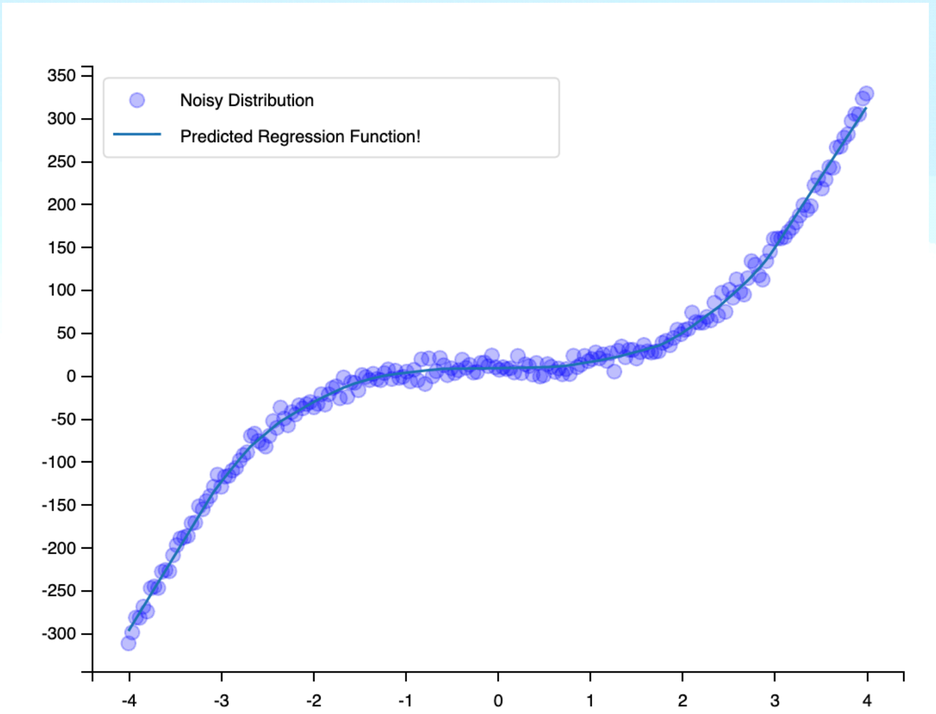
model(x) = f(w*x + w0) (nicht mal Vektor!)

mit nichtlinearem f.
Gegebenenfalls mehrere Schichten stapeln



Aufgabe

Regression an Daten die 5*x^3 + 10 verteilt sind

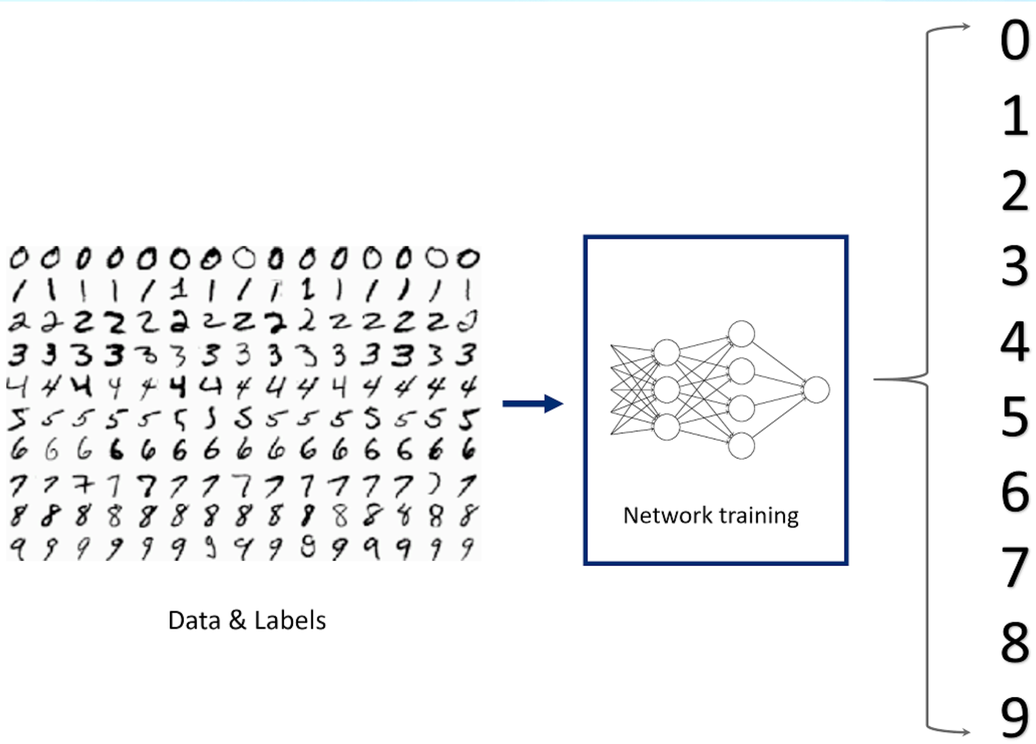


MNIST Klassifizierung

MNIST (Modified National Institute of Standards and Technology) ist ein weit verbreitetes "Hello World " **Dataset** in der Welt des maschinellen Lernens, speziell im Bereich der Bildverarbeitung und der Entwicklung von tiefen neuronalen Netzen. Es wird oft als Einstieg in das Training von **Bildklassifikationsmodellen** verwendet, sowie zum Testen ob eine neue Architektur grundsätzlich mächtig ist.

Überblick und Definition:

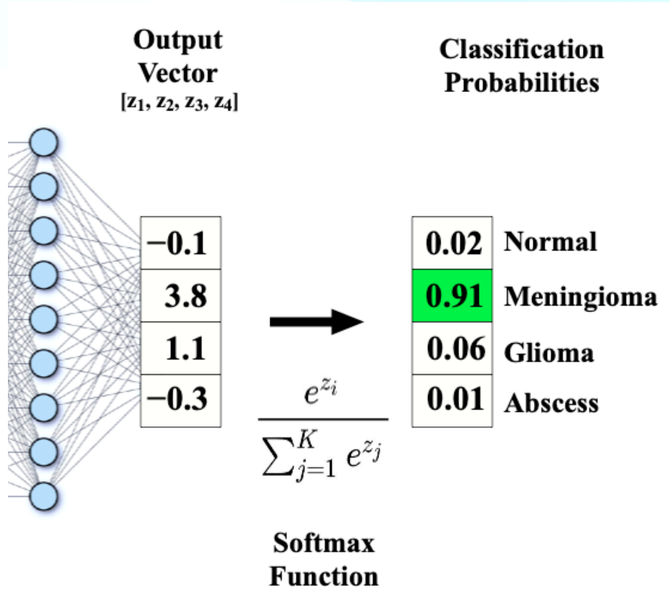
- **MNIST-Dataset**: Eine Sammlung von handgeschriebenen Ziffern von 0 bis 9, insgesamt 70.000 Graustufenbilder (**28x28 Pixel**).
  - **Trainingsdaten**: 60.000 Bilder.
  - **Testdaten**: 10.000 Bilder.
- **Ziel**: Die Klassifikation der Ziffern anhand der Bilddaten.
- 



Softmax

Idee: normiere die Outputs so dass die Summe 1 ist
(wie bei Wahrscheinlichkeitsverteilungen)

Der Topf mit dem höchsten Wert entspricht der wahrscheinlichsten 'Klasse'.



Softmax Klassifizierung

In PyTorch wird **Klassifizierung** in einer einzigen Zeile gehandhabt:

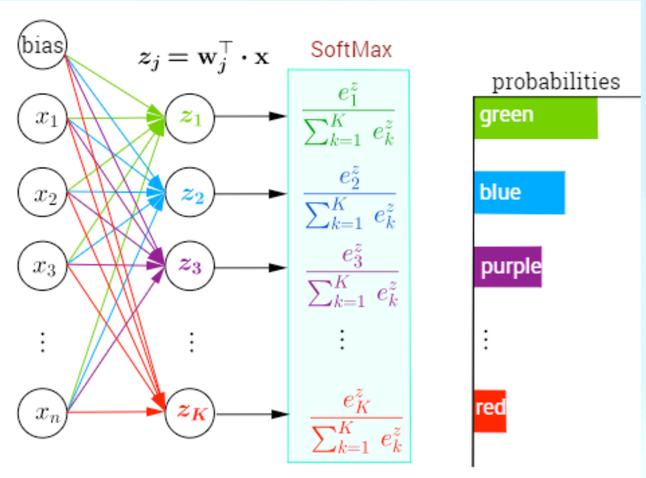
```
criterion = torch.nn.CrossEntropyLoss()  # 💡 Softmax is internally computed.
```

Der Rest der Trainingsfunktion ist generisch:

```
for epoch in range(training_epochs):
    accuracy()
    for X, Y in data_loader:
        optimizer.zero_grad() # reset gradients to zero
        hypothesis = model(X.view(-1, 28 * 28))
        cost = criterion(hypothesis, Y) # Verlustfunktion
        cost.backward() # backpropagation, ziehe Gradienten ab
        optimizer.step() #
```

Fehler

CrossEntropy für '**Klassen**' bewertet zB wie unsere Zahlen 0…9 in verschiedenen **Töpfen** landen.
Hierzu werden am Ende alle Töpfe durch die Summe der Aktivierungen geteilt, was man **Softmax** (weiches maximum) nennt. Das erinnert an **Wahrscheinlichkeitsvariablen**, deren **Summe stets 1** ergeben.



10 Zielvariablen haben Zustände 0/1, praktisch dazwischen [0,1] (ja ist Ziffer '7', oder nicht, oder vielleicht)

Eine weiche (arg)maximum Funktion schaut, in welchem der Töpfe der größte Wert ist.
Ein gutes Maß zum Schauen, ob die Werte in unseren Töpfen p_i unseren gewünschten Werten y_i entsprechen ist

Normal: $y_i * \bar{y}_i$ solange bei einem Zustand beide 1 sind, kommt 1 raus, was gewünscht/anzustreben ist.
wir formulieren es als Minimierungsproblem um:

CrossEntropy (logistische **Verlustfunktion**) $-\sum y_i \log(p_i)$
(log ist monoton, man verliert nur Skalierung, lässt sich leichter ableiten!)
Der Cross-Entropy-Verlust ist effektiv, weil er große Strafen für sicher vorhergesagte falsche Antworten verhängt, was die Konvergenzgeschwindigkeit des Lernprozesses beschleunigt.

Regression vs. Klassifikation

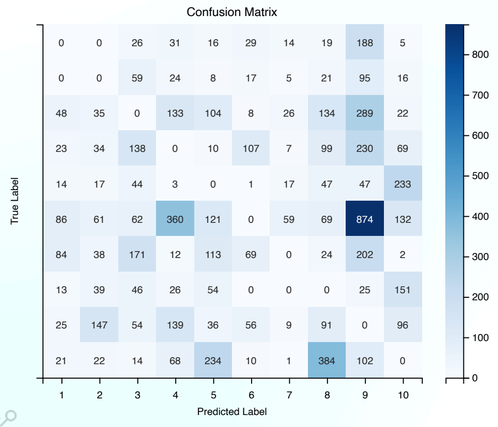
Weitere Aspekte

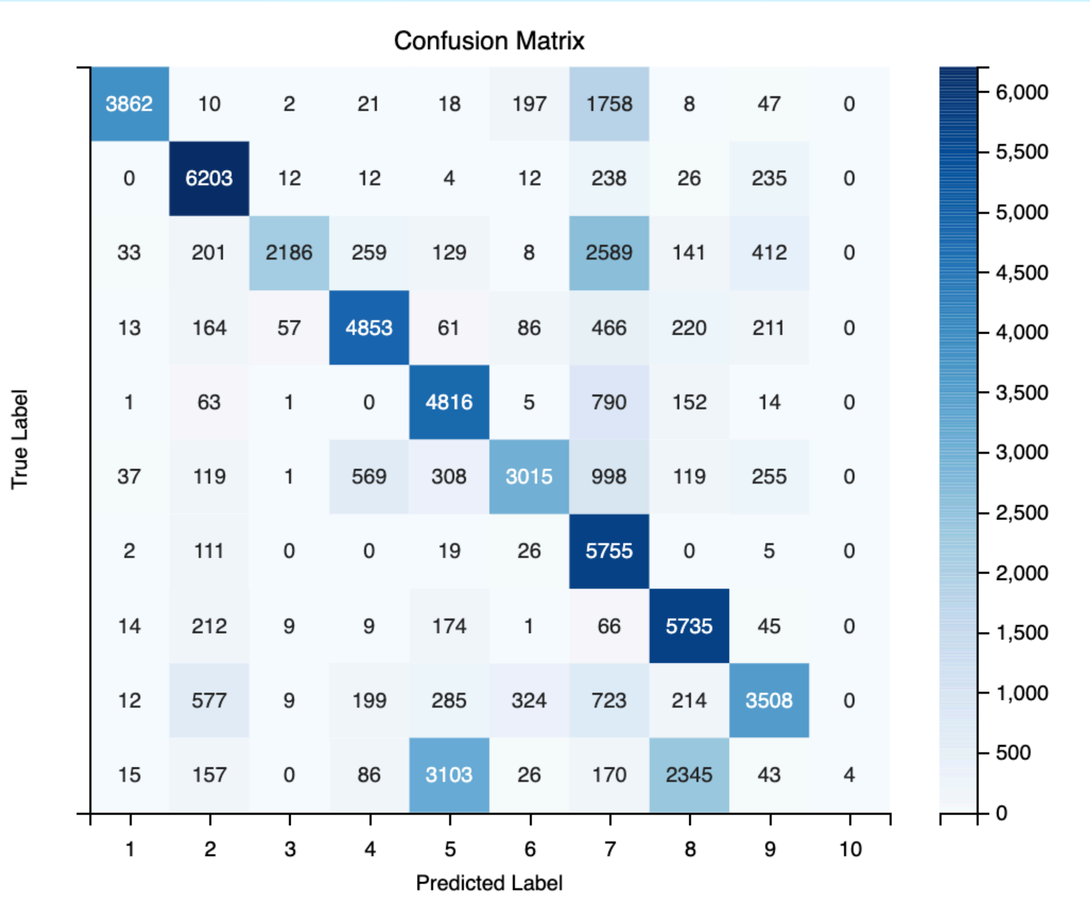
- **Metriken**:
  - Regression: Mean Squared Error (MSE), Root Mean Squared Error (RMSE), R^2-Score
  - Klassifikation: **Accuracy**, Precision, Recall, F1-Score
- **Modelle**:
  - Regression: Lineare Regression, Lasso, Ridge, Neuronale Netze
  - Klassifikation: Logistische Regression, Entscheidungsbäume, Support Vector Machines, Random Forest, **Neuronale Netze**
- **Datenverteilung**:
  - Regression: Normalerweise kontinuierlich verteilt.
  - Klassifikation: Diskrete Klassen, oft gleichmäßig oder ungleich verteilt.

Confusion Matrix

Um mein Gefühl dafür zu bekommen welche Daten falsch klassifiziert wurden ist es auf hilfreich eine so genannte Konfusions Matrix anzeigen zu lassen:

Im Beispiel von MNIST sieht man zum Beispiel dass die Zahl 7 öfters mit der Zahl 1 und 3 verwechselt wurde:



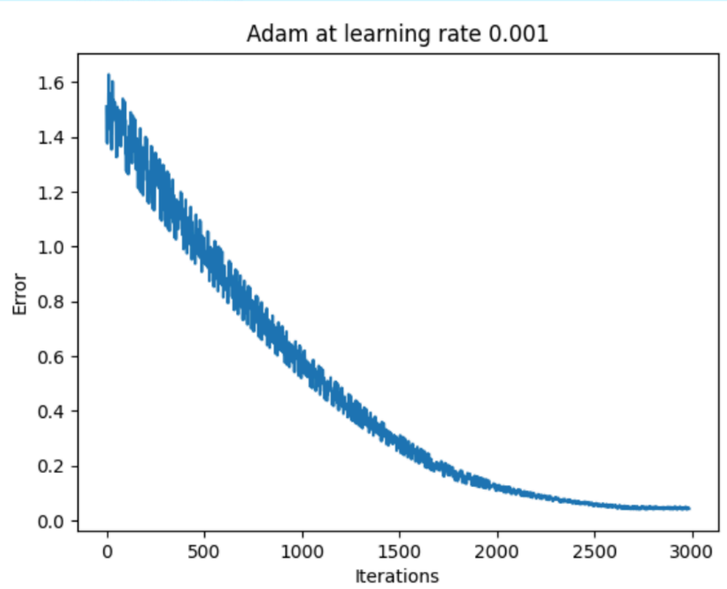


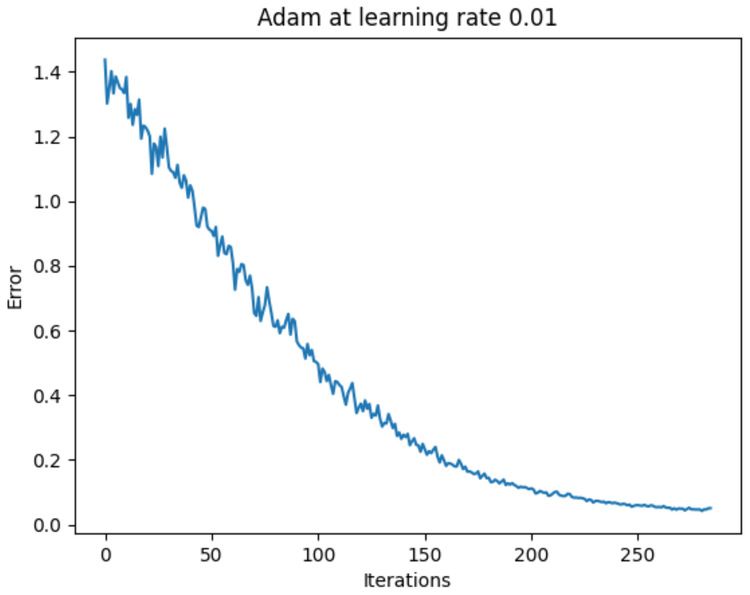
Pause

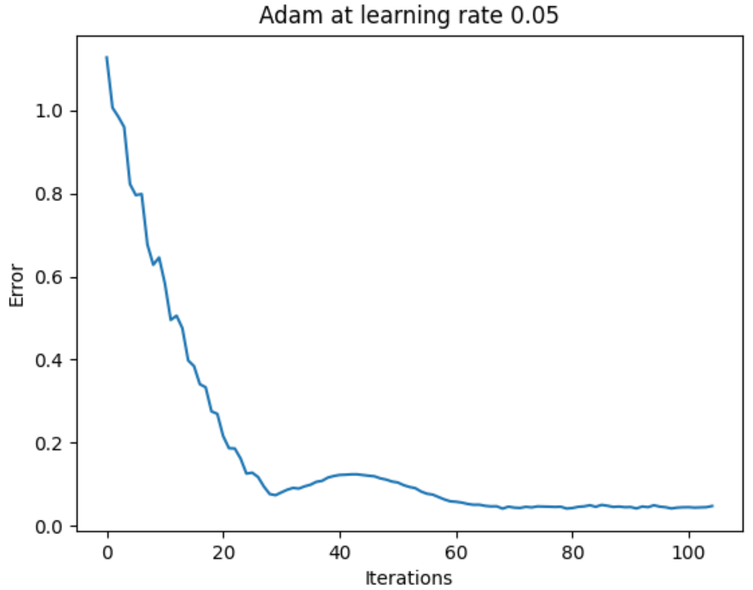
Lernkurven

Man kann und sollte den **Lernfortschritt** (absteigende Fehlerrate) **verfolgen.**
In jedem Fall per print(), aber idealer- und typischer weise auch mit **plots**

Solche **plots** kann man einfach **selber erstellen** oder professionell mit Bibliotheken wie **tensorboard** wobei der zusätzliche Komfort / Nutzen den Aufwand nur selten deckt?







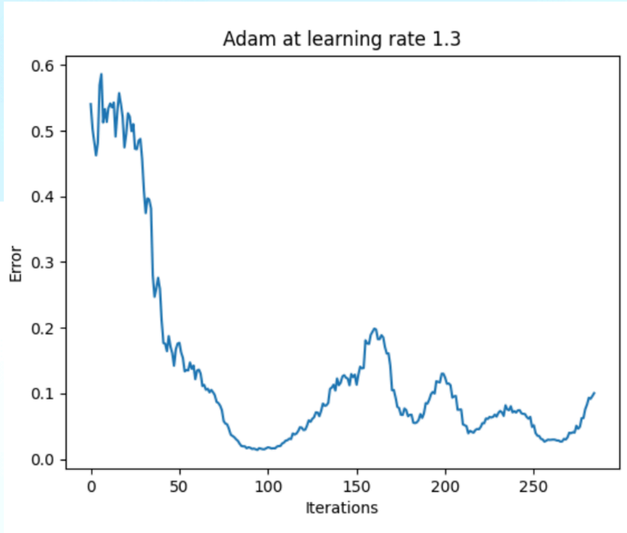
Lernkurven

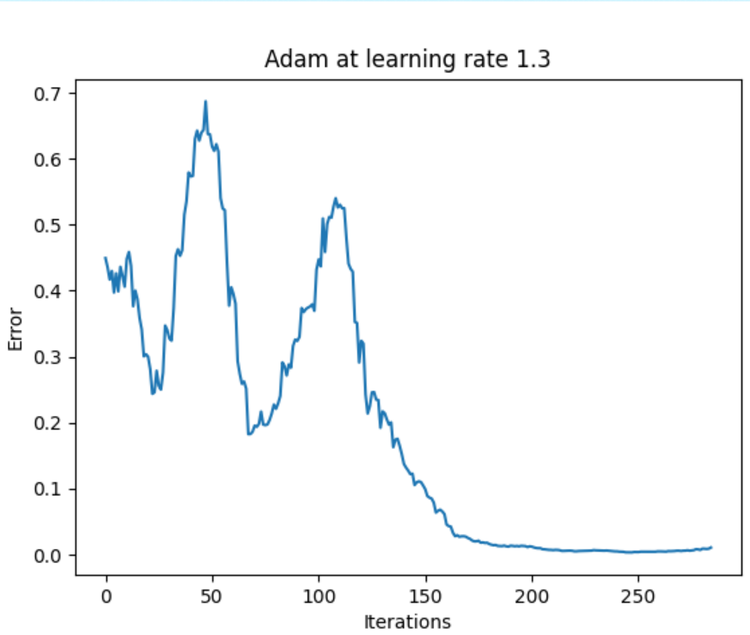
Wall time / compute

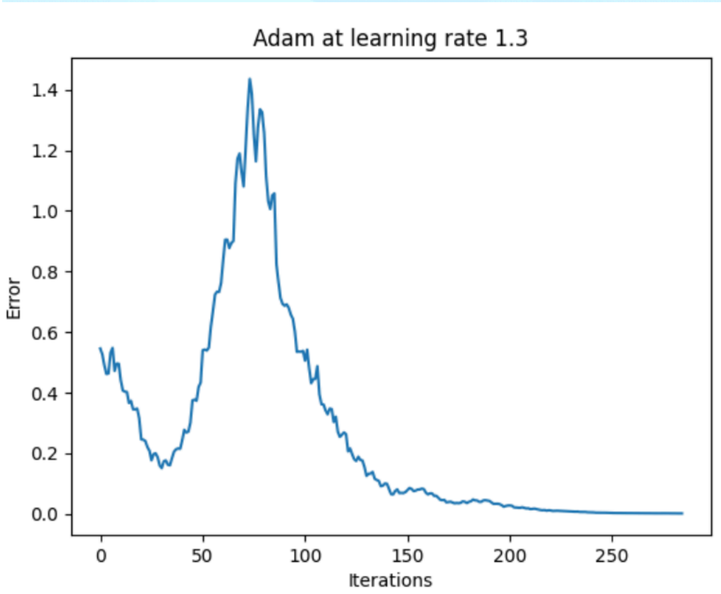
Schlechte Lernkurven

'**Schlechte**' Lernkurven zeigen **keine konsistente Reduktion** oder keinen steten Abfall der **Fehlerrate.**

⚠️ Auch unter idealen Lernbedingungen können manchmal **Lernrückschritte** vorkommen!







Gute Lernkurven

'**Gute**' Lernkurven zeigen einen möglichst **steten Abfall.**

⚠️ Auch unter idealen Lernbedingungen können manchmal **Lernrückschritte** vorkommen!

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