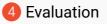
PyTorch CHEAT SHEET =



2 Modelldefinition





Allgemein

PyTorch ist ein Open Source Machine Learning Framework. Zur Repräsentation und Verarbeitung von Daten wird torch. Tensor, eine mehrdimensionale Matrix, verwendet. Zentrales Element aller neuronalen Netze in PyTorch ist das Autograd-Paket, welches automatische Differenzierung für alle Operationen auf Tensoren bietet.

import torch

import torch.nn as nn

from torchvision import datasets, models, transforms

import torch.nn.functional as F Funktionssammlung

Basispaket

Neuronale Netze

Beliebte Datensets, Architekturen und Bildtransformationen

torch.randn(*size) torch.Tensor(L) tnsr.view(a, b, ...)

Zufälliger Tensor Tensor aus Liste L Tensor in Größe (a, b, ...) umformen

Gradient soll berechnet requires_grad=True

werden

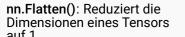
Laver



nn.Linear(m, n): Fully Connected Layer (oder Dense Layer) von m auf n Neuronen



nn.ConvXd(m. n. s): X-dimensionaler Convolution Layer von m auf n Kanäle mit Kernelgröße s; $X \in \{1, 2, 3\}$





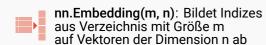
nn.MaxPoolXd(s): X-dimensionaler Pooling Layer mit Kernelgröße s; $X \in \{1, 2, 3\}$



nn.Dropout(p=0.5): Setzt während des Trainings zufällig Eingaben auf 0; hilft Overfitting zu vermeiden



nn.BatchNormXd(n): Normalisiert einen X-dimensionalen Èingabebatch mit n Features; $X \in \{1, 2, 3\}$





nn.RNN/LSTM/GRU: Recurrent Networks verbinden Neuronen einer Schicht mit Neuronen derselben oder vorheriger Schichten

torch.nn bietet zudem viele weitere Bausteine.

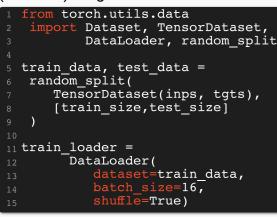
"State of the art"-Architekturen gibt es unter https://paperswithcode.com/sota

Daten laden

Ein Datensatz wird durch eine Klasse repräsentiert, die von **Dataset** erbt (Liste von (Features, Label)-Tupeln).

Mit DataLoader erfolgt das Laden strukturunabhängig in Batches.

Üblicherweise wird der Datensatz in Trainings- (z.B. 80%) und Testdaten (z.B. 20%) aufgeteilt.



Aktivierungsfunktionen

Zu den häufigsten Aktivierungsfunktionen zählen ReLU, Sigmoid und Tanh. Daneben existieren noch eine Reihe weiterer Aktivierungsfunktionen.

nn.ReLU() erzeugt ein nn.Module etwa für Sequential Modelle. F.relu() ist nur Aufruf der ReLU Funktion etwa für eine Verwendung in der forward Methode.



nn.ReLU() oder F.relu()

Ausgabe zwischen 0 und ∞, häufigste Aktivierungsfunktion



nn.Sigmoid() oder F.sigmoid()

Ausgabe zwischen 0 und 1, oft für Wahrscheinlichkeiten verwendet



nn.Tanh() oder F.tanh()

Ausgabe zwischen -1 und 1, oft für zwei Klassen verwendet

Modeldefinition class Net(nn. Module):

Es gibt verschiedene Wege ein neuronales Netz in PyTorch zu definieren, z.B. mit nn.Sequential (a), als Klasse (b) oder auch als Kombination aus beidem.

```
model = nn.Sequential(
 nn.Conv2D(,,,,,)
 nn.MaxPool2D(

)
 nn.ReLU()
 nn.Flatten()
 nn.Linear( , )
```

```
def __init__():
   super(Net, self).__init__()
    self.conv
        = nn.Conv2D( , , , )
     self.pool
        = nn.MaxPool2D( )
    self.fc = nn.Linear( , )
  def forward(self, x):
    x = self.pool(
          F.relu(self.conv(x))
    x = x.view(-1, \blacksquare)
    x = self.fc(x)
    return x
model = Net()
```

Modell speichern/laden

model = torch.load('PATH') Modell laden torch.save(model, 'PATH') Modell speichern

Üblicherweise werden nur die Modellparameter, nicht das ganze Modell gespeichert: model.state_dict()

```
torch.save(model.state dict(), 'params.ckpt')
model.load state dict(
                 torch.load('params.ckpt')
```

GPU Training

device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

Ist eine GPU mit CUDA-Unterstützung verfügbar, werden die Berechnungen mit dem Befehl model.to(device) bzw. inputs, labels = data[0].to(device), data[1].to(device) an die GPU mit ID 0 übergeben.

import torch.optim as optim

Training

nn.BCELoss

VERLUSTFUNKTION

PyTorch bietet bereits verschiedene Möglichkeiten den Fehler zu berechnen, u.a.:

Mittlerer absoluter Fehler nn.L1Loss

nn.MSELoss Mittlere quadratische Abweichung (L2Loss) Kreuzentropie, u.a. für Single-Label, nn.CrossEntropyLoss

unausgewogene Trainingsdaten

Binäre Kreuzentropie, u.a. für Multi-Label

oder Autoencoder

OPTIMIERUNGSALGORITHMEN (torch.optim)

Optimierungsalgorithmen werden genutzt, um im Gradientenabstiegsverfahren Gewichte zu aktualisieren und die Lernrate dynamisch anzupassen, u.a.:

optim.SGD Stochastic Gradient Descent optim.Adam Adaptive Moment Estimation

optim.Adagrad Adaptive Gradient optim.RMSProp Root Mean Square Prop

```
correct = 0 # Korrekt klassifiziert
total = 0 # Insgesamt klassifiziert
model.eval()
    h torch.no grad():
   for data in test loader:
     inputs, labels = data
    outputs = model(inputs)
     , predicted = torch.max(outputs.data, 1
    total += labels.size(0) # Batchgröße
    correct += (predicted==labels)
                         .sum().item()
4print('Accuracy: %s' % (correct/total))
```

loss fn = nn.CrossEntropyLoss() # Algorithmus für Gradientenabstiegsverfahren optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9) 10# Mehrfach über Datensatz iterieren 11for epoch in range(2): model.train() # Trainingsmodus aktivieren for i, data in enumerate(train loader, 0): # data ist Batch aus [inputs, labels] inputs, labels = data # Gradienten nullen optimizer.zero grad() # Ausgabe berechnen

outputs = model(inputs)

loss.backward()

optimizer.step()

Evaluation

Mit der Evaluation wird geprüft, ob das Modell zufriedenstellende Ergebnisse auf bisher zurückgehaltenen Daten liefert. Je nach Ziel werden unterschiedliche Metriken wie bspw. acurracy, precision, recall, F1 oder BLEU verwendet.

model.eval() Aktiviert Evaluationsmodus, manche Layer verhalten sich hier anders

torch.no_grad() Deaktiviert Autograd, reduziert

Speicherbedarf, beschleunigt Rechnungen

Fehler bestimmen & zurückpropagieren

Gewichte / Lernrate aktualisieren

loss = loss fn(outputs, labels)