

PyTorch CHEAT SHEET









1 Daten laden 2 Modelldefinition 3 Training 4 Evaluation

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.

<code>import torch</code>	Basispaket	<code>torch.randn(*size)</code>	Zufälliger Tensor
<code>import torch.nn as nn</code>	Neuronale Netze	<code>torch.Tensor(L)</code>	Tensor aus Liste L
<code>from torchvision import datasets, models, transforms</code>	Beliebte Datensets, Architekturen und Bildtransformationen	<code>tnsr.view(a, b, ...)</code>	Tensor in Größe (a, b, ...) umformen
<code>import torch.nn.functional as F</code>	Funktionssammlung	<code>requires_grad=True</code>	Gradient soll berechnet werden

Layer

 <code>nn.Linear(m, n)</code> : Fully Connected Layer (oder Dense Layer) von m auf n Neuronen	 <code>nn.ConvXd(m, n, s)</code> : X-dimensionaler Convolution Layer von m auf n Kanäle mit Kernelgröße s; $X \in \{1, 2, 3\}$
 <code>nn.Flatten()</code> : Reduziert die Dimensionen eines Tensors auf 1	 <code>nn.MaxPoolXd(s)</code> : X-dimensionaler Pooling Layer mit Kernelgröße s; $X \in \{1, 2, 3\}$
 <code>nn.Dropout(p=0.5)</code> : Setzt während des Trainings zufällig Eingaben auf 0; hilft Overfitting zu vermeiden	 <code>nn.BatchNormXd(n)</code> : Normalisiert einen X-dimensionalen Eingabebatch mit n Features; $X \in \{1, 2, 3\}$
 <code>nn.Embedding(m, n)</code> : Bildet Indizes aus Verzeichnis mit Größe m auf Vektoren der Dimension n ab	 <code>nn.RNN/LSTM/GRU</code> : 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.

```
1 from torch.utils.data
2 import Dataset, TensorDataset,
3     DataLoader, random_split
4
5 train_data, test_data =
6     random_split(
7         TensorDataset(inps, tgts),
8         [train_size, test_size]
9     )
10
11 train_loader =
12     DataLoader(
13         dataset=train_data,
14         batch_size=16,
15         shuffle=True)
```

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.

 <code>nn.ReLU()</code> oder <code>F.relu()</code> Ausgabe zwischen 0 und ∞ , häufigste Aktivierungsfunktion	 <code>nn.Sigmoid()</code> oder <code>F.sigmoid()</code> Ausgabe zwischen 0 und 1, oft für Wahrscheinlichkeiten verwendet
 <code>nn.Tanh()</code> oder <code>F.tanh()</code> Ausgabe zwischen -1 und 1, oft für zwei Klassen verwendet	

Modelldefinition

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(1, 1, 1)
    nn.MaxPool2D(1)
    nn.ReLU()
    nn.Flatten()
    nn.Linear(1, 1)
)
```

```
class Net(nn.Module):
    def __init__():
        super(Net, self).__init__()

        self.conv
            = nn.Conv2D(1, 1, 1)

        self.pool
            = nn.MaxPool2D(1)

        self.fc = nn.Linear(1, 1)

    def forward(self, x):
        x = self.pool(
            F.relu(self.conv(x))
        )

        x = x.view(-1, 1)

        x = self.fc(x)

        return x

model = Net()
```

Training

VERLUSTFUNKTION

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

<code>nn.L1Loss</code>	Mittlerer absoluter Fehler
<code>nn.MSELoss</code>	Mittlere quadratische Abweichung (L2Loss)
<code>nn.CrossEntropyLoss</code>	Kreuzentropie, u.a. für Single-Label, unausgewogene Trainingsdaten
<code>nn.BCELoss</code>	Binäre Kreuzentropie, u.a. für Multi-Label oder Autoencoder

OPTIMIERUNGSGRUNDALGORITHMEN (torch.optim)

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

<code>optim.SGD</code>	Stochastic Gradient Descent
<code>optim.Adam</code>	Adaptive Moment Estimation
<code>optim.Adagrad</code>	Adaptive Gradient
<code>optim.RMSProp</code>	Root Mean Square Prop

```
1 correct = 0 # Korrekt klassifiziert
2 total = 0 # Insgesamt klassifiziert
3
4 model.eval()
5 with torch.no_grad():
6     for data in test_loader:
7         inputs, labels = data
8         outputs = model(inputs)
9         _, predicted = torch.max(outputs.data, 1)
10        total += labels.size(0) # Batchgröße
11        correct += (predicted==labels)
12                               .sum().item()
13
14 print('Accuracy: %s' % (correct/total))
```

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()`

```
1 torch.save(model.state_dict(), 'params.ckpt')
2 model.load_state_dict(
3     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.

```
1 import torch.optim as optim
2
3 # Verlustfunktion / Loss Function
4 loss_fn = nn.CrossEntropyLoss()
5
6 # Algorithmus für Gradientenabstiegsverfahren
7 optimizer = optim.SGD(model.parameters(),
8                        lr=0.001, momentum=0.9)
9
10 # Mehrfach über Datensatz iterieren
11 for epoch in range(2):
12     model.train() # Trainingsmodus aktivieren
13     for i, data in enumerate(train_loader, 0):
14         # data ist Batch aus [inputs, labels]
15         inputs, labels = data
16
17         # Gradienten nullen
18         optimizer.zero_grad()
19
20         # Ausgabe berechnen
21         outputs = model(inputs)
22         # Fehler bestimmen & zurückpropagieren
23         loss = loss_fn(outputs, labels)
24         loss.backward()
25         # Gewichte / Lernrate aktualisieren
26         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. accuracy, 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