**Codereview um den Prozess hinter der Handschrifterkennung zu verstehen**

import torch

import torch.nn as nn

import torch.nn. functional as F

import torch.optim as optim

from torch.autograd import Variable

from torchvision import datasets, transforms

kwargs = {'num\_workers': 1, 'pin\_memory': True}

train\_data =

torch.utils.data.DataLoader(

datasets.MNIST('data', train=True,download=True,

transform=transforms.Compose([transforms.ToTensor(),

transforms.Normalize((0.1307,) , (0.3081,))])), batch\_size=64, shuffle=True, \*\*kwargs)

test\_data =

torch.utils.data.DataLoader(datasets.MNIST('data', train=False,

transform=transforms.Compose([transforms.ToTensor(),

transforms.Normalize((0.1307,) , (0.3081,))])), batch\_size=64, shuffle=True, \*\*kwargs)

train\_data & test\_data benutzen denselben Datensatz. In der Praxis werden jedoch verschiedene Datensätze benutzt, damit sie auch beim Test auf neue Daten gut abschneiden.

class Netz(nn.Module):

    def \_\_init\_\_(self):

        super(Netz, self).\_\_init\_\_()

        self.conv1 = nn.Conv2d(1,10, kernel\_size=5)

        self.conv2 = nn.Conv2d(10,20, kernel\_size=5)

        self.conv\_dropout = nn.Dropout2d()

        self.fc1 = nn.Linear(320, 60)

        self.fc2 = nn.Linear(60, 10)

Es werden hier die Faltungsschichten mit Conv2d erstellt. Diese wurden speziell für die Verarbeitung mit zweidimensionalen Daten wie Bildern erstellt. Man sieht das es 1 Eingangsneuron gibt. Dieses Eingangsneuron enthält das Komplette Bild. Dieses Eingangsneuron arbeitet gleichzeitig auch nur mit Graufstufen. Würden wir bspw. eine 2 einfügen, wären die Berechnungen Fehlerhaft. Eine 3 würde wieder Sinn ergeben, wenn man mit einem farbigen Bild, also rbg Features(Muster) erkennen möchte. Die 10 sind die Ausgangsneuronen dieses Layers. Das Bild wird in 10 Regionen aufgeteilt. Keine Region hat mit der anderen zu tun. Mehr Ausgangsneuronen würden komplexere Muster ergeben. Die kernel\_size ermöglicht es über das Bild zu wandern. Es wird quasi eine Kopie der Größe 5x5 (NICHT Pixel) der Fläche des Bildes gemacht. Beispiel: der kernel fängt oben links im Bild an und erstellt einen Abdruck. Es wandert 1 Pixel nach rechts und macht erneut einen Abdruck.

Da neuronale Netze dazu neigen Werte auswendig zu lernen, muss man das neuronale Netz auch vergessen lassen. Diese Funktion tut dies. Der Standardwert liegt bei p = 0,5, was 50% vergessen wird.

nn.Dropout2d(p= 0.3) = 30% vergessen usw…

Die Daten aus self.conv2 = nn.Conv2d(10,20, kernel\_size=5) werden an

self.fc1 = nn.Linear(320, 60) weitergegeben und abgeflacht(räumliche Merkmale transformiert in Vektor). Die 320 ist nun ein Vektor mit 320 Eingangsneuronen. Die Fully Connected Schicht ermöglicht somit die Backpropagation-Algorithmus zu verwenden.Ohne transformation/abflachen nicht möglich.

Auf die 320 kommt man, wenn man print(x.size()) nutzt. Das Ergebnis dessen ist ein torch.size([64, 20, 4, 4]). 64 ist die Batch\_size (Erinnerung: Batch sagt aus das z.B. 64 Bilder pro Trainingsprozess genutzt werden). Die 20 da unsere Ausgangsneuronenauszahl bei 20 liegt. Die 4, 4 ergeben sich durch den kernel\_size = 5, ändert man die kernel\_size, ändert sich auch der „Abdruckbereich“, heißt wir haben einen Abdruck von 4x4 Pixel. 4x4x20 ergeben 320. Die print(x.size()) werden im def forward erstellt und erst dann kann man auch die self.fc1 = nn.Linear(320, 60) setzen. Tut man dies nicht, hat man die Gesamtzahl der Neuronen verändert und verfälscht die Berechnungen.

def forward(self, x):

        x = self.conv1(x)

        x = F.max\_pool2d(x, 2)

        x = F.relu(x)

        x = self.conv2(x)

        x = self.conv\_dropout(x)

        x = F.max\_pool2d(x, 2)

        x = F.relu(x)

        #print(x.size())

        #x = torch.flatten(x, start\_dim=1)

        #print(x.shape)

        x = x.view(-1, 320)

        x = F.relu(self.fc1(x))

        x = self.fc2(x)

        return F.log\_softmax(x, dim=0)

Die forward-Methode eines neuronalen Netzwerks nehmen die Neuronen die Eingabedaten entgegen und stellen mithilfe ihrer Gewichtungen Beziehungen zwischen diesen Daten her.

Die Gewichtungen werden während des Trainingsprozesses angepasst, um die relevanten Muster und Beziehungen in den Daten zu erfassen.

Die Ausgaben der Neuronen dienen dann als Eingaben für die nächsten Schichten des Netzwerks.

def forward(self, x): wird immer wieder mit out = model(data) aufgerufen. Normalerweie werden funktionen so aufgerufen mode.forward(data), jedoch beinhaltet module(data )einen automatischen aufruf der forward methode. Es ist so vorgegeben.

Die aktuellen Eingabedaten x = self.conv1(x) durchlaufen nochmal den Konstruktor. So werden relevante Merkmale und deren Gewichtungen erneut extrahiert(festgelegt).

Bei x = F.max\_pool2d(x, 2) wird quasi das Bild nochmal auf seine wichtigsten Daten reduziert. Es ähnelt der Bildkomprimierung. x sind unsere Daten und die 2 sind 2x2 Matritzen. Aus dieser Matrix wird dann jeweils der höchste Wert genommen.

**Mit x = F.relu(x) werden nichtlineare Beziehungen herausgefunden und verknüpft. Bei dem Beispiel mit den Ziffern erkennt das Neuronale Netz die Ziffern, aber kann z.B. nicht zuordnen zu welcher Person die Handschrift gehört. Die Beziehungen können anhand von Merkmalen mit x = F.relu(x) herausgefunden werden und erlernt werden. Wichtig ist, in der ReLU-Funktion werden keine Gewichtungen angepasst.**

Als nächstes wird x = self.conv2(x) erneut durchgeführt aus der Klasse Netz def \_\_init\_\_. Das sorgt dafür das Räumlichkeiten (kanten, formen) und komplexere Merkmale besser erkennen innerhalb dieser Daten/Bildes.

Danach werden wieder mit x = self.conv\_dropout(x) Neuronen zurückgesetzt.

x = F.max\_pool2d(x, 2) & x = F.relu(x) wiederholen sich Sinngemäß.

print(x.size()) oder x = torch.flatten(x, start\_dim=1) oder print(x.shape)

ist wichtig, wenn man herausfinden möchte, was die korrekte Eingangsneuronenzahl ist zum Erstellen des Vektors und um damit die Fully Connected Schicht richtig anwenden zu können, also die Backpropagation.

Bei der Methode x = x.view(-1, 320) werden die einzelnen Dimensionen quasi verschmolzen. torch.size([64, 20, 4, 4]) waren unsere 4 Dimensionen und mit x = x.view(-1, 320) wurde es zu einer einzigen Dimension. Eigentlich sind es 2 Dimensionen, da die batch\_size auch eine Dimension ist, jedoch als separat betrachtet wird. Die -1 ermöglicht uns die batch-size zu ermitteln. Sie ist zwar ursprünglich auf 64 gesetzt, aber es kann durch **Zufälliges Mischen, Änderungen im Dataloader-Code oder Datensatzende erreicht Probleme geben. Man könnte auch 3 Dimensionen Formen mit x = x.view(-1, 20, 16). Also die Funktion ist flexible.**

Danach wiederholt sich der Vorgang wieder etwas. Die relu Funktion wird wieder angewendet, um nichtlineare Beziehungen herzustellen und innerhalb dessen werden die Eingangsneuronen und Ausgangsneuronen Linear angepasst. Unsere Parameter wie die 320 sind festgelegt, aber nicht die daten innerhalb dieser Dimension , also unser x in seinen Gewichtungen.

Bei der return F.log\_softmax(x, dim = 0) Methode werden die „rohen“ Aktivierungswerte x in Wahrscheinlichkeiten umgewandelt. Mit dieser Wahrscheinlichkeit lässt sich sagen, wie erfolgreich ein Neuron in Zukunft in seiner Lösung ist. Bessere Werte werden dabei stärker betont. Es wird dabei eine zusammengefasste Wahrscheinlichkeitsbeteiligung aller Neuronen zu ihren Ergebnissen erstellt. Das Modell lernt das z.b. Neuron 5 besser in seiner Lösung ist als Neuron 2.

Somit wäre Neuron 5 korrekter in seiner Klassifikation als Neuron 2.

0 ist die Batch\_size Dimension.

Als Randnotiz warum bspw. die log\_softmax(x) Methode verwendet wird und warum nicht die sigmoid oder eine nn.MSELoss() Funktion? Man muss wissen welche Art von Typ von Daten man vor sich hat und was das Ziel dahinter ist. Jede Funktion kann einen Erfolg erzielen, aber bei der einen Methode fällt sie geringer, bei der anderen größer aus. Aus Erfahrungswerten lernt man, wann welche Methode besser geeignet ist.

model = Netz()

model.cuda()

optimizer = optim.SGD(model.parameters(), lr = 0.1, momentum = 0.8)

model.parameters() gibt dem Optimizer Zugriff auf alle Gewichte , Bias und Gradienten des Modells, damit er diese während des Trainings optimieren kann.

Mit der Lernrate (lr) gehe ich große oder kleine Schritte in die Richtung des Minimus der Verlustfunktion und mit dem Momentum ist mein Schwung gemeint, mit dem ich die Geschwindigkeit bestimme, wie schnell dieser Schritt getan wird. Zu hohe Lernraten können dazu führen, dass über das Minimum hinweggesprungen wird, während zu niedrige Lernraten zu langsamem Fortschritt oder lokalen Minima führen können. Das Momentum hilft dabei, in Richtung des Minimums zu beschleunigen und mögliche Hindernisse zu überwinden.

def train(epoch):

    model.train()

    for batch\_id, (data, target) in enumerate(train\_data):

        data = data.cuda()

        target = target.cuda()

        optimizer.zero\_grad()

        out = model(data)

        loss = F.nll\_loss(out, target)

        loss.backward()

        optimizer.step()

        # print(f'Train Epoch: {epoch} [{batch\_id \* len(data)}/{len(train\_data.dataset)} ({100. \* batch\_id / len(train\_data):.0f}%)]\tLoss: {loss.item():.6f}')

Ähnelt stark einem mehrdimensionalen Array:

for batch\_id, (data, target) in enumerate(train\_data): Innerhalb unserer train\_data befindet sich ein Paket: batch\_id (0 wäre unser erstes Paket(0 ist unser Index), 1 nächstes Paket), diese wiederum enthält zwei weitere Pakete: Paket: data und paket: target. In data finden wir eine mehrdimensionale Matrix (abhängig von Anzahl der Achsen). In target finden wir die Lösung, die erarbeitet werden soll in der Form eines Vektors. Das Ziel innerhalb der def\_train(epoch) ist es, alle Daten von Data in unser neuronales Netz hinzuzufügen out = model(data). Das Netz weiß erstmal nicht welchen Lernprozess es verfolgen soll mit den Daten Data und hat zufällige Gewichtungen. Hinterher wird dem Netz target gezeigt . Diese beinhalten die Lösungen, also das Ziel in der unser Netz trainieren soll loss = F.nll\_loss(out, target). Das Netz merkt sich die Fehler und passt seine Gewichtungen an loss.backward(). optimizer.zero\_grad() sorgt dafür das der Gradient auf null gesetzt wird, damit es keine Akkumulierungen gibt beim Gradienten. Es ist eine Vorbereitung auf den nächsten Vorwärts- & Rückwärtslauf.

Hier loss.backward() geben die Gradienten an, wie sich der Verlust ändert, wenn sich die Gewichtungen geringfügig ändern. Der optimizer.step() aktualisiert die Parameter mit den neuen Gewichtungen.

def test():

    model.eval()

    loss = 0

    correct = 0

    for data, target in test\_data:

        with torch.no\_grad():

            data = Variable(data.cuda())

        target = Variable(target.cuda())

        out = model(data)

        loss += F.nll\_loss(out, target, reduction='sum').item()

        prediction = out.data.max(1, keepdim=True)[1]

        correct += prediction.eq(target.data.view\_as(prediction)).cpu().sum()

    loss = loss / len(test\_data.dataset)

    print('Durchschnittsloss: ', loss)

    print('Genauigkeit: ', 100.\*correct/len(test\_data.dataset))

model.train() ist wichtig, um das Neuronale Netz in den Trainingsmodus zu versetzen. Damit wird dann auch die Dropout-Schicht aktiviert, um das Over Fitting zu reduzieren. Die Batch-Normalisierung wird auf die Minibatches angewendet. Mit model.eval() möchte man verhindern das Neuronen ausgeschalten werden und die Batch-Normalisierung auf den gesamten Datensatz verwendet wird.

loss = 0 & correct = 0 werden benutzt, um die korrekten Vorhersagen und die falschen Vorhersagen anzusammeln.

with torch.no\_grad(): sorgt dafür, dass während dieser Vorwärtsdurchläufe keine Parameteraktualisierungen durchführen.

Der Aufruf out = model(data) führt die Methode forward durch das neuronale Netzwerk aus. Während dieses Durchlaufs werden die Gewichtungen und Biases verwendet, um die Eingabedaten durch das Netzwerk zu transformieren. Die Gewichtungen wurden während des Trainingsprozesses angepasst.

loss += F.nll\_loss(out, target, reduction='sum').item() Es wird sich der Verlust zwischen den aktuellen Vorhersagen out und den Zielwerten target angesehen. Dabei wird jeder Verlust pro Durchgang summiert mit reduction='sum'

.item() wird verwendet um den Verlust als Gleitkommazahl darzustellen.

prediction = out.data.max(1, keepdim=True)[1] extrahiert die Vorhersagen aus out mit dem höchsten Wahrscheinlichkeitswert(auch Maximum genannt). Die erste 1 bezieht sich darauf, worauf der max-Operator angewendet wird. In dem Fall auf Dimension 1. Die letzte 1 sorgt das die Anzahl an Dimensionen beibehalten soll. Max-Operator erklärt anhand eines Beispiels:

tensor = torch.tensor([[1, 3, 5],

                       [7, 2, 4],

                       [6, 8, 9]])

Wird mit dem Max-Operator zu.

Maximalwerte je Zeile: tensor([[5],

                               [7],

                               [9]])

Indizes der Maximalwerte je Zeile: tensor([[2],

                                           [0],

                                           [2]])

Mit correct += prediction.eq(target.data.view\_as(prediction)).cpu().sum()

Überprüfen wir ob die vorhergesagten werte (prediction) mit den Zielwerten (target.data.view) übereinstimmen. Mit eq() lassen wir uns die Werte mit True oder False angeben und alle booleschen True Werte werden mit sum() summiert. Mit cpu() wird die Rechnung auf den Cpu verschoben.

Am Ende werden die summierten korrekten vorhersagen durch die Gesamtlänge der Datenpunkte dividiert.

if \_\_name\_\_ == '\_\_main\_\_':

    for epoch in range(0, 4):

        train(epoch)

        test()