4. PyTorch

17 juli

Under dagens lektion kommer vi att gå igenom hur man kan träna ett neuralt nätverk i Python med hjälp av Pytorch. Vi kommer att träna ett neuralt nätverk att känna igen handskrivna siffror från det välkända MNIST datasetet. För att åstadkomma detta behövs inte mycket kod, särskilt ifall man använder sig utav många inbyggda pytorchfunktioner. Vi kommer dock inte att göra detta. Då det är kul att lära sig saker, kommer ni att implementera vissa av dessa saker själva, så att ni får en bättre förståelse över hur det egentligen fungerar.

Intsallera Pytorch

Pytorch finns som ett Pythonpaket och går lätt att installera på sin dator. Om ni har en dator som kör Linux eller MacOS så räcker det med att skriva `pip3 install torch` i terminalen (förutsatt att man har pip installerat först, vilket brukar finnas). Ifall ni har en windows dator, kan det vara lite svårare att komma igång, och vi rekommenderar därför att ni kör på en av skoldatorerna.

Ett lätt sätt att verifiera att PyTorch har installerats korrekt är att köra ett pythonprogram med raden `import torch`. Ifall det fungerar så är ni redo att börja koda.

Vi kommer även att använda oss utav ett par andra bibliotek under lektionen. För dagens lektion behöver ni även ladda ner torchvision, vilket ni gör genom att köra `pip3 install torchvision`.

Tensorer

Innan vi börjar med att träna ett neuralt nätverk så är det bra att ha lite kunskap om tensorer, som används lite överallt av pytorch. En tensor kan man se som en n-dimensionell matris. Har man en 1-dimensionell tensor så är det i princip en vektor. Är det en 2-dimensionell tensor så är det en matris, men det kan även ha fler dimensioner än så.

Här är ett par exempel på att skapa tensorer. Testa att printa ett par av de här alternativen för att se vad de gör.

```
import torch

a = torch.tensor([[1,2],[3,4]])
b = torch.zeros(5)
c = torch.zeros((5, 6))
d = torch.arange(10)
e = torch.ones(8)
```

Vill man använda en tensor, så kan använda den ungefär som om det vore en lista. Dvs ifall man har en 1-dimensionell tensor funkar:

```
import torch
a = torch.arange(5)
print(a[2])
```

Har man en flerdimensionell tensor kan man skriva:

```
import torch
# Skapa en tensor med 6 rader med siffrorna 0-3
a = torch.tensor([list(range(4) for _ in range(6)])
single_row = a[2]
single_entry = a[2, 3]
two_rows = a[1:3]
```

Notera att man alltså kan indexera med ett komma mellan varje dimension, och behandla varje dimension som om det vore en lista.

Varje tensor som man skapar har en viss storlek, som man kan se genom att skrive `variable_name.shape`, och det går även att manipulera storleken lätt, exempelvis så här:

```
import torch
a = torch.arange(24)
print(a.shape, a)
b = a.reshape((4, 6))
print(b.shape, b)
c = b.unsqueeze(0)
print(c.shape, c)
d = c.squeeze()
print(d.shape, d)
```

1. Skapa en två dimensionell tensor som har 5 rader och 6 kolumner. Tensorn bör innehålla talen 0-5 på första raden, 6-11 på andra raden osv.

   Tips: `reshape` är en användbar metod.

2. Skriv ut de mellersta två kolumnerna av matrisen du skapade i förra uppgiften. Dvs kolumnerna med index 2 och 3.

   Tips: indexering/slicing

Det finns även massor med andra användbara funktioner för att manipulera tensorer. Ett tips när man håller på med deep learning är att se till att storleken på sina tensorer är korrekt. En vanlig bugg är nämligen att man skapar tensorer som till exempel har omvänd storlek (t.ex. (5, 6) istället för (6, 5)) eller att den har fel antal dimensioner (t.ex. (1, 5, 6) istället för (5, 6)). Därför tycker jag att det ofta är värt att printa storleken av tensorer så att man kan se till så att de är rätt.

Förberedning av träningsdatan

För att träna ett neuralt nätverk måste vi ladda ner datasetet och läsa in det i programmet. Detta kan vi lätt åstadkomma med i princip en rad kod, då det redan finns i torchvision biblioteket:

```
import torchvision
image_directory = 'images/'
dataset = torchvision.datasets.MNIST(image_directory,
                                     download=True,
                                     transform=torchvision.transforms.ToTensor())
```

Första gången ni kör det här kommer datasetet att sparas ner till mappen som ni gav som `image_directory`. Nerladdningen kommer dock bara att ske första gången. Följande gånger ni kör programmet kommer bilderna att läsas in från datorn istället.

Dataset innehåller en hel del information om datasetet. Det viktigaste, som vi kommer att använda oss utav är `dataset.data` - en tensor som innehåller all bilddata, och `dataset.targets` - en tensor med vad varje bild är för typ av siffra.

3. Skapa två partitioner av datasetet: en partition som vi kommer att använda för att träna nätverket, och en partition som vi kommer att använda för att säkerställa att nätverket inte har överfittat till träningsdatan. Träningspartitionen brukar bestå utav en majoritet av datan, men exakt hur mycket är upp till er.

Tips: Det kan vara lätt att loopa över alla exempel i datasetet och lägga de i en temporär lista. Sen på slutet kan man skapa konvertera listan till en tensor.

4. Skapa en klass som representerar ditt neurala nätverk. Här ser ni ett exempel på hur klassen kan se ut:

```
class Model(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.forward_layer = ...

    def forward(self, x):
        return self.forward_layer(x)
```

Tills allting är på plats räcker det med att modellen har ett enda lager. Typiskt sett brukar en modell ha flera lager (ibland väldigt många), men det kan ni lägga in senare, när allting fungerar om ni vill. Ett exempel på ett lager ni kanske vill använda är `torch.nn.Linear`.

`forward` funktionen går att anropa väldigt lätt genom att skriva anropa modellen som om det vore en funktion. T.ex:

```
model = Model()
input_data = ... # Bilddatan från datasetet.
output = model(input_data)
```

5. Loopa över träningsdatasetet och träna modellen på en bild åt gången. Exempel:

```
training_data = ...
training_targets = ...
```

```
model = ...
optimizer = torch.optim.SGD(model.parameters())
for image_data, target in zip(training_data, training_targets):
    optimizer.zero_grad()
    output = model(image_data)
    loss = ...
    loss.backward()
```

Som ni ser så finns det ett par saker som ni måste lägga till. Dels så skapar vi ett `optimizer` objekt som är ansvarig för att uppdatera nätverkets vikter. I början av varje iteration så måste vi se till att anropa `optimizer.zero_grad()` för att derivatorna som beräknades i förra iterationen inte ska räknas med igen i den här iterationen. Sist gör vi en forward pass genom att anropa modellens forward funktion, beräknar lossen, och kör `loss.backward()` som uppdaterar parametrerna i nätverket.

6. Lägg till kod så att ni under träningen kan se hur stor loss modellen har. Ett bra sätt är att printa medelvärdet av de n senaste iterationerna och göra det med jämna mellanrum. Starta en träning och se hur lossen förändras. Förhoppningsvis ser ni att lossen går ner med tiden. Om inte, så kan det vara ett tecken på att det finns nån bugg i koden.

7. Lägg till kod som utvärderar modellen på valideringsdatasetet. För detta kommer vi inte att beräkna en loss som vi gjort tidigare. Istället kommer vi endast köra forward funktionen, och jämföra modellens output med korrekt svar. Hur många procent av gångerna gissar modellen på samma sak som det korrekta svaret efter att ni har tränat modellen?

8. Spara modellen till en fil och läs in den igen. På så sätt kan ni stänga av programmet och fortsätta träna modellen igen, utan att behöva börja om från början. Det här kan spara massor med tid.

   Tips: kolla upp `torch.save`, `torch.load`.

Förbättringar

Nu när vi har skrivit ihop alla de viktigaste delarna av programmet, så är det dags att försöka förbättra modellen. Målet är att den ska få så hög precision som möjligt på valideringssettet. Precisionen påverkas av massor av faktorer, och nu ska vi ta en titt på vissa av de faktorerna.

9. Modifiera koden så att vi loopar igenom datasettet flera gångar. Att vi går igenom hela datasettet kallas för en epoch, och ofta räcker det inte bara med en epoch. Gör så att modellen tränar i ett par epocher. Efter varje epoch, printa precisionen på valideringssettet. Detta kan hjälpa med att avgöra hur många epocher som är rimligt att ha. När precisionen slutar att gå ner så behöver vi inte träna längre.

10. Testa att lägga till ett par till lager i nätverket, särskilt ifall ni bara har ett lager för tillfället. Exakt hur många lager som är bäst och hur stora de bör vara är inte

uppenbart. Det kan vara värt att testa ett par olika alternativ för att se vad som ger bäst resultat.

11. Hittills har vi bara tränat på en bild åt gången. En förbättring som vi kan använda oss utav är att träna med en batch åt gången. Dvs att modellen samtidigt tränar på ett antal bilder. Modifiera träningsloopen så att modellen tar in *batch_size* stycken bilder åt gången, där ni väljer ett rimligt värde på *batch_size*. Ofta brukar man använda en tvåpotens, men man måste inte göra det.

Extrauppgifter

12. Rita egna siffror och se ifall nätverket gissar rätt på vad de liknar. Ni kan antingen rita bilderna i t.ex. paint, gimp osv, eller fota på ett papper. För detta så måste ni även skriva ihop kod för att läsa in bilderna och konvertera de till rätt format. Dvs svartvitt, rätt storlek, och som en tensor.

13. Skriv ihop ett program som visualiserar vilka siffror som nätverket gör fel på. Finns det något gemensamt för de bilderna som nätverket gör fel på? Eller är det ganska slumpartat?

14. Experimentera med att applicera transformer på bilderna under träningen. Syftet med detta är att få in mer variation i datasettet, för att försöka få en högre precision på valideringssettet. Det finns en mängd transformer i `torchvision.transforms` som går att använda. Bland annat skulla man kunna testa att rotera bilderna med en liten, slumpad vinkel, lägga till en blurreffekt på vissa bilder, eller att flytta vissa bilder lite i sidled/höjdled.

15. Läs på om olika optimizers och hur de fungerar. Ett känt exempel är `torch.optim.Adam`.

Final boss

Ifall ni känner er sugna på en utmaning så får ni testa att implementera ännu mer från scratch. Dvs hela forward propagation- och backpropagationberäkningarna, uppdatering av nätverkets parametrar etc.