DL for Masters, WiSe 2025/26 – week 2 exercise tasks

Prof. Alexander Binder

Week 02: PyTorch and a first true neural network

Task 1– Numpy to Pytorch and back

1.1

- Lade ein Bild mit Hilfe von PIL.Image in ein PIL.Image objekt
- wandle es in ein numpy array um
- print the shape of it
- print the dtype of the numpy array (es sollte $uint8$ sein)
- danach wandle das numpy array in einen PyTorch Tensor um
- aendere den dtype des PyTorch Tensors auf float32
- berechne den Mittelwert und Standardabweichung fuer R,G und B Kanaele separat in PyTorch.
- Bonus: Falls eine GPU genutzt wird, schiebe den Tensor auf die GPU, pruefe wieviel GPU Speicher PyTorch nutzt

1.2

- erstelle eine zufaellige matrix als PyTorch tensor mit shape $(7, 15, 2)$ durch Ziehung aus einer binomialverteilung mit $n = 30, p = 0.75$ mit Hilfe von `https://pytorch.org/docs/stable/distributions.html`
- Berechne den mittelwert dieser matrix über $dim = 1$
- wandle diese beiden in numpy arrays um

Task 2 – NNs old school (simple)

Old school: 1990s style

- take `fmnist_pytorch_logreg_class.py`, rename it.
- take in there the class

  areallyoldschoolneuralnet
- define 3 layers `torch.nn.Linear`, with 300,100 and (number of classes) many outputs in the above class. If you got the slowest notebook in the whole city, just use two layers with 300 and (number of classes) many neurons.
  - they cannot be defined in the `def forward(self,x):` but must be put into the class constructor, because you need their trainable parameters to be persistent across `.forward(x)` calls
  - after the first two linear layers apply a ReLU. See the PyTorch documentation or stackoverflow
- train it, measure the test performance. it should get better than the logistic regression model.

Broadcasting

> Programming using Broadcasting is a fundamental skill used in many data science packages (Numpy, JAX, Tensorflow, Pytorch).

```
a= torch.full((2,3),3.)
b= torch.full((5,1,3),3.)
c= a+b
```

What will `c.shape` be ?

`https://pytorch.org/docs/stable/notes/broadcasting.html`

same holds for many other element-wise binary operators like $-, *, /, **n$

$$
\begin{aligned}
a =& torch.ones((4)) \\
b =& torch.ones((1,4)) \\
& torch.add(a,b) && \rightarrow (1,4) \\
a =& torch.ones((4)) \\
b =& torch.ones((4,1)) \\
& torch.add(a,b) && \rightarrow (4,4)!!! \\
a =& torch.ones((3)) \\
b =& torch.ones((4,1)) \\
& torch.add(a,b) && \rightarrow (4,3) \\
a =& torch.ones((3)) \\
b =& torch.ones((1,4)) \\
& torch.add(a,b) && \rightarrow ERR
\end{aligned}
$$

Example:

start	after insert	after copying
(4,1)	(4,1)	(4,4)
(4)	(1,4)	(4,4)

- smaller tensor gets filled **from the left** with singleton dimensions until he has same dimensionality as larger tensor, as if `.unsqueeze(0)` would be applied again and again

More examples:

start	after insert	after copying
(1,3)	(1,3)	(1,3)
(3)	(1,3)	(1,3)

start	after insert	after copying
(2,3)	(1,2,3)	(5,2,3)
(5,1,3)	(5,1,3)	(5,2,3)

start	after insert	after copying
(1,7)	(1,1,1,7)	(5,2,3,7)
(5,2,3,7)	(5,2,3,7)	(5,2,3,7)

start	after insert	after copying
(4,1)	(1,4,1)	ERR
(2,3,7)	(2,3,7)	ERR

if broadcasting is too mind-boggling (why you dont do an exercise to have your heart beat faster), then apply `.unsqueeze(dim)` on your tensor, until both tensors have the same number of dimension axes. The only thing what is done then, is copying along $dim = 1$ axes.

Why do we need broadcasting?

- it is **MUCHHH** faster than nested for-loops
- avoid for loops in favor of broadcasting or built-in torch operations whenver possible
- for loops are good for test cases to ensure that code works correct, less suitable for serious model training or inference

for-loops = inexperienced coder (or very sleepy prof needs exercises to get done ;))

how Broadcasting works

1– the smaller tensor gets filled **from the left** with singleton dimensions until he has same dimensionality as larger tensor, as if `.unsqueeze(0)` would be applied again and again

2– then check whether they are compatible – they are incompatible if in one dimension both tensors have sizes > 1 which are not equal. if they are incompatible, you will get an error.

3– whenever a dimension with size 1 meets a dimension with a size $k > 1$, then the smaller vector is replicated/copied $k-1$ times in this dimension until he reaches in this dimension size k and your actual operation is applied

Task 3 – Broadcasting (really simple)

Which of these shapes are compatible under broadcasting for a simple binary operator like addition or multiplication? If they are, what is the resulting shape

- $(3, 1, 3), (2, 3, 3)$
- $(4, 1), (3, 1, 1, 5)$
- $(3), (3, 1, 1, 5)$
- $(1, 4), (7, 1)$
- $(6, 3, 1, 7), (2, 7)$
- $(6, 3, 1, 7), (2, 1, 7)$
- $(1, 2, 3, 1, 6), (8, 1, 3, 2, 6)$
- $(2, 5, 1, 7), (9, 2, 3, 2, 1)$

Task 4 – Broadcasting (still simple)

4.1

Compute for a vector $a = (a_0, a_1, a_2, a_3, a_4)$ of length 5 and a vector $b = (b_0, b_1, b_2)$ of length 3 the set of all $a_i + b_k$ using PyTorch broadcasting.

The result should be a matrix of shape $(5, 3)$ such that

$$c[i, k] = a_i + b_k \ .$$

Implement it as a function.

Hinweis/hint: use `Tensor.unsqueeze(dim)` before multiplying.

Note: the inputs should have a 1-dimensional shape vector for the pytorch tensors:
len(a.shape)= 1, len(b.shape)= 1

Wir werden das kommende Mal eine kompliziertere Variante davon programmieren, die etwas sinnvolles als Algorithmus tut, welche Broadcasting nutzt.

4.2

Compute for a tensor a of shape (K, L, M) and a tensor b of shape (K, M, N) the set of all

$$c[k, l, m, n] = a[k, l, m] * b[k, m, n]$$

using PyTorch broadcasting.

Implement it as a function.

Test the algorithm it for two randomized tensors a, b with `a.shape=(3,2,5), a.shape=(3,5,4)`
Hinweis/hint: use `Tensor.unsqueeze(dim)` before multiplying