Introduction to deep learning

Lecture 3

Sam van Gool
vangool@irif.fr

Master 2 informatique
Université Paris Cité

21 January 2025

Today's lecture

1. Recap
2. Softmax regression and SGD: implementation
3. Vectorization
4. Going deep: Activation functions

Last week

CM:

- Softmax regression

TP:

Reminder: Course webpage: https://samvangool.net/iap/

Last week

CM:

- Softmax regression
- Intuition for *cross entropy loss*

TP:

Reminder: Course webpage: `https://samvangool.net/iap/`

Last week

CM:

- Softmax regression
- Intuition for *cross entropy loss*
- Different forms of *gradient descent*:

TP:

Reminder: Course webpage: https://samvangool.net/iap/

Last week

CM:

- Softmax regression
- Intuition for *cross entropy loss*
- Different forms of *gradient descent*:
  - naive (pass through all the data before making an update)

TP:

Reminder: Course webpage: https://samvangool.net/iap/

Last week

CM:

- Softmax regression
- Intuition for *cross entropy loss*
- Different forms of *gradient descent*:
  - naive (pass through all the data before making an update)
  - stochastic minibatch.

TP:

Reminder: Course webpage: https://samvangool.net/iap/

Last week

CM:

- Softmax regression
- Intuition for *cross entropy loss*
- Different forms of *gradient descent*:
  - naive (pass through all the data before making an update)
  - stochastic minibatch.

TP:

- Implementing *softmax regression* in Numpy

Reminder: Course webpage: https://samvangool.net/iap/

Last week

CM:

- Softmax regression
- Intuition for *cross entropy loss*
- Different forms of *gradient descent*:
  - naive (pass through all the data before making an update)
  - stochastic minibatch.

TP:

- Implementing *softmax regression* in Numpy
- Over-fitting a single batch

Reminder: Course webpage: https://samvangool.net/iap/

Last week

CM:

- Softmax regression
- Intuition for *cross entropy loss*
- Different forms of *gradient descent*:
  - naive (pass through all the data before making an update)
  - stochastic minibatch.

TP:

- Implementing *softmax regression* in Numpy
- Over-fitting a single batch
- Training on a full dataset, achieving $> 80\%$ accuracy.

Reminder: Course webpage: https://samvangool.net/iap/

TO DO last week

- Finish TP2.

TO DO last week

- Finish TP2.
- Read [Z, pp. 82-88, pp. 126-148], [P, Sections 5.1–5.2].

TO DO last week

- Finish TP2.
- Read [Z, pp. 82-88, pp. 126-148], [P, Sections 5.1–5.2].
- Do [Z, Section 4.4.7], Exercise 1 and [P, Problem 5.2].

TO DO last week

- Finish TP2.
- Read [Z, pp. 82-88, pp. 126-148], [P, Sections 5.1–5.2].
- Do [Z, Section 4.4.7], Exercise 1 and [P, Problem 5.2].
- If you need a reminder of Linear Algebra and Calculus: read [Z, Section 2.3 and 2.4], or [P, Appendix B].

Today's lecture

[Z, Section 4.4.7], Exercise 1

Numeric stability of `softmax` (see question 4(b) on TP2).

See notebook.

[P, Problem 5.2]

The loss function in the case of binary classification.

See notebook.

General remarks about the labs (TP)

- The lab exercises (TP) are an **essential** part of the course!

General remarks about the labs (TP)

- The lab exercises (TP) are an **essential** part of the course!
- Next week's TP 4 will be handed in and graded (25% of your grade).

General remarks about the labs (TP)

- The lab exercises (TP) are an **essential** part of the course!
- Next week's TP 4 will be handed in and graded (25% of your grade).
- In today's TP, you will make last week's solution more efficient using vectorization.

A (slow) implementation of softmax regression

See notebook.

Today's lecture

Broadcasting

- Numpy arrays have a `shape` attribute.

Broadcasting

- Numpy arrays have a `shape` attribute.
- Broadcasting happens when we perform an operation on two numpy arrays that do not have the same shape.

Broadcasting example

Example

(See notebook.)

```
A = np.array([[1.],[2.],[3.],[4.]]) # shape: (4,1)
B = np.array([[5.,-5.,5.,-5.,5.]])  # shape: (1,5)
A + B
```

Images generated by Google ImageFX, input prompts: "An unhappy mathematician, putting his head against a blackboard" and "A happy computer, which can perform any computation you ask of it".

Broadcasting example

Example

(See notebook.)

```
A = np.array([[1.],[2.],[3.],[4.]]) # shape: (4,1)
B = np.array([[5.,-5.,5.,-5.,5.]])  # shape: (1,5)
A + B
```

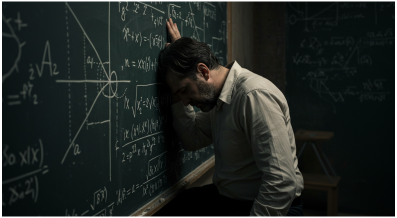
Images generated by Google ImageFX, input prompts: "An unhappy mathematician, putting his head against a blackboard" and "A happy computer, which can perform any computation you ask of it".

Broadcasting example

Example

(See notebook.)

```
A = np.array([[1.],[2.],[3.],[4.]]) # shape: (4,1)
B = np.array([[5.,-5.,5.,-5.,5.]])  # shape: (1,5)
A + B
```



The Mathematician: Not allowed!!

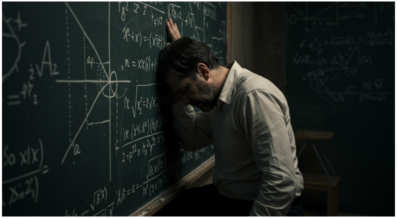
Images generated by Google ImageFX, input prompts: "An unhappy mathematician, putting his head against a blackboard" and "A happy computer, which can perform any computation you ask of it".

Broadcasting example

Example

(See notebook.)

```
A = np.array([[1.],[2.],[3.],[4.]]) # shape: (4,1)
B = np.array([[5.,-5.,5.,-5.,5.]])  # shape: (1,5)
A + B
```



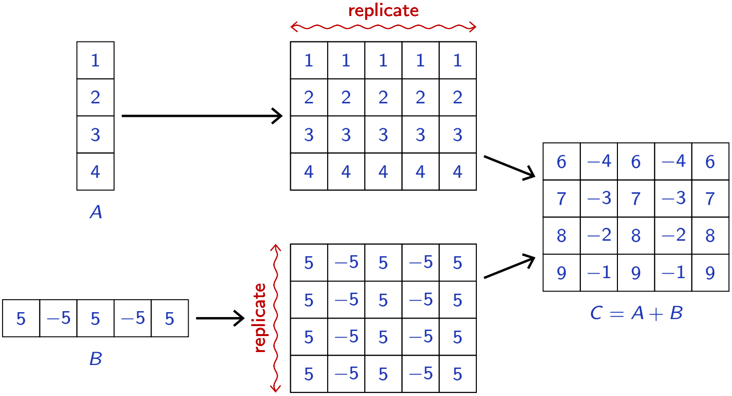
The Mathematician: Not allowed!!

Numpy: This is fine.

Images generated by Google ImageFX, input prompts: "An unhappy mathematician, putting his head against a blackboard" and "A happy computer, which can perform any computation you ask of it".

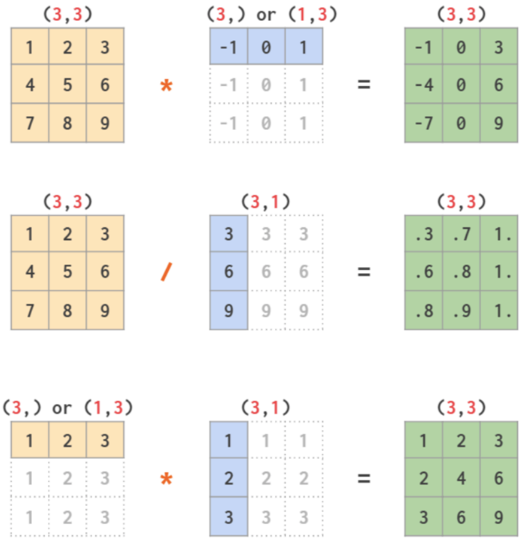
Broadcasting example visualized

```
A = torch.tensor([[1.], [2.], [3.], [4.]])
B = torch.tensor([[5., -5., 5., -5., 5.]])
C = A + B
```



Example & image credit: François Fleuret's Deep Learning Course.

Broadcasting visualized



(From https://towardsdatascience.com/broadcasting-in-numpy-58856f926d73)

Broadcasting in practice: vectorized batch processing

- We represented our dataset as a 60000×784 matrix.

Broadcasting in practice: vectorized batch processing

- We represented our dataset as a 60000×784 matrix.
- We then processed each line separately, for example, for every batch, we repeat the following line 64 times:

  ```
  probs = softmax(x @ W + b)
  ```

Broadcasting in practice: vectorized batch processing

- We represented our dataset as a 60000×784 matrix.
- We then processed each line separately, for example, for every batch, we repeat the following line 64 times:

  ```
  probs = softmax(x @ W + b)
  ```
- Numpy (and Pytorch) work *faster* than Python loops.

Broadcasting in practice: vectorized batch processing

- We represented our dataset as a 60000×784 matrix.
- We then processed each line separately, for example, for every batch, we repeat the following line 64 times:

  `probs = softmax(x @ W + b)`
- Numpy (and Pytorch) work *faster* than Python loops.
- We can **vectorize** the calculation `softmax(x @ W + b)`.

Batch matrix multiplication

- Instead of repeating `x @ W + b` for every row `x` separately, we can take advantage of broadcasting, and do it at once for the entire batch:

```
x.shape # (64, 784)
W.shape # (784, 10)
b.shape # (1, 10)
(x @ W).shape # (64, 10)
(x @ W + b).shape # (64, 10)
```

Broadcasting details

1. If one of the arrays has fewer dimensions than the other, then it is reshaped by adding dimension 1 on the left.

Broadcasting details

1. If one of the arrays has fewer dimensions than the other, then it is reshaped by adding dimension 1 on the left.
2. For every remaining dimension mismatch, *if one of the dimensions is equal to 1*, then it is expanded by replicating elements.

Broadcasting details

1. If one of the arrays has fewer dimensions than the other, then it is reshaped by adding dimension 1 on the left.
2. For every remaining dimension mismatch, *if one of the dimensions is equal to 1* , then it is expanded by replicating elements.
3. If there is a mismatch where both have a dimension not equal to 1, **fail**.

Broadcasting details

1. If one of the arrays has fewer dimensions than the other, then it is reshaped by adding dimension `1` on the left.
2. For every remaining dimension mismatch, *if one of the dimensions is equal to 1*, then it is expanded by replicating elements.
3. If there is a mismatch where both have a dimension not equal to `1`, **fail**.

Reflection

When can we compute `A + B`?

Broadcasting details

1. If one of the arrays has fewer dimensions than the other, then it is reshaped by adding dimension `1` on the left.
2. For every remaining dimension mismatch, *if one of the dimensions is equal to 1*, then it is expanded by replicating elements.
3. If there is a mismatch where both have a dimension not equal to 1, **fail**.

Reflection

When can we compute `A + B`?

- `A.shape = (10)`, `B.shape = (5,1)`

Broadcasting details

1. If one of the arrays has fewer dimensions than the other, then it is reshaped by adding dimension `1` on the left.
2. For every remaining dimension mismatch, *if one of the dimensions is equal to 1*, then it is expanded by replicating elements.
3. If there is a mismatch where both have a dimension not equal to 1, **fail**.

Reflection

When can we compute `A + B`?

- `A.shape = (10)`, `B.shape = (5,1)`
- `A.shape = (10)`, `B.shape = (1,5)`

Broadcasting details

1. If one of the arrays has fewer dimensions than the other, then it is reshaped by adding dimension `1` on the left.
2. For every remaining dimension mismatch, *if one of the dimensions is equal to 1*, then it is expanded by replicating elements.
3. If there is a mismatch where both have a dimension not equal to `1`, **fail**.

Reflection

When can we compute `A + B`?

- `A.shape = (10)`, `B.shape = (5,1)`
- `A.shape = (10)`, `B.shape = (1,5)`
- `A.shape = (1,10)`, `B.shape = (1,10,1)`

Today's lecture

ReLU

- The last additional ingredient: rectified linear unit (ReLU):

$$\mathrm{ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0. \end{cases}$$

ReLU

- The last additional ingredient: rectified linear unit (ReLU):

$$\mathrm{ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \le 0. \end{cases}$$

- Briefly: $\mathrm{ReLU}(x) = \max(x, 0)$. ReLU is an *activation function*.

ReLU

- The last additional ingredient: rectified linear unit (ReLU):

$$\mathrm{ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0. \end{cases}$$

- Briefly: $\mathrm{ReLU}(x) = \max(x, 0)$. ReLU is an *activation function*.
- Its derivative is very easy to compute:

$$\frac{\partial \mathrm{ReLU}}{\partial x}(x_0) = \begin{cases} 1 & \text{if } x_0 \geq 0 \\ 0 & \text{if } x_0 < 0. \end{cases}$$

ReLU

- The last additional ingredient: rectified linear unit (ReLU):

$$\mathrm{ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0. \end{cases}$$

- Briefly: $\mathrm{ReLU}(x) = \max(x, 0)$. ReLU is an *activation function*.
- Its derivative is very easy to compute:

$$\frac{\partial \mathrm{ReLU}}{\partial x}(x_0) = \begin{cases} 1 & \text{if } x_0 \geq 0 \\ 0 & \text{if } x_0 < 0. \end{cases}$$

- Multi-layer perceptron = stack of several linear layers, with ReLU in between each layer.

Numeric stability trick

- We saw that, for numeric stability, instead of computing

$$\mathrm{softmax}(\mathbf{o})_j = \frac{o_j}{P} \text{ where } P = \sum_j \exp(o_j)$$

we compute, for a number $M \in \mathbb{R}$,

$$\mathrm{softmax}'(\mathbf{o})_j = \frac{o_j - M}{P} \text{ where } P = \sum_j \exp(o_j - M)$$

Numeric stability trick

- We saw that, for numeric stability, instead of computing

$$\mathrm{softmax}(\mathbf{o})_j = \frac{o_j}{P} \text{ where } P = \sum_j \exp(o_j)$$

we compute, for a number $M \in \mathbb{R}$,

$$\mathrm{softmax}'(\mathbf{o})_j = \frac{o_j - M}{P} \text{ where } P = \sum_j \exp(o_j - M)$$

Reflection

Why does this give the same answer?

Computing the gradient of softmax by hand

- We compute the logits $\mathbf{o} = \mathbf{x}^{\mathrm{T}} \cdot \boldsymbol{\Omega} + \boldsymbol{\beta}$.

Computing the gradient of softmax by hand

- We compute the logits $\mathbf{o} = \mathbf{x}^{\mathrm{T}} \cdot \boldsymbol{\Omega} + \boldsymbol{\beta}$.
- We compute the probability vector $\mathbf{p} = \mathrm{softmax}(\mathbf{o})$.

Computing the gradient of softmax by hand

- We compute the logits $\mathbf{o} = \mathbf{x}^{\mathrm{T}} \cdot \boldsymbol{\Omega} + \boldsymbol{\beta}$.
- We compute the probability vector $\mathbf{p} = \mathrm{softmax}(\mathbf{o})$.
- If the correct class is c, the loss on $\mathbf{x}$ is $\ell \overset{\mathrm{def}}{=} -\log p_c$.

Computing the gradient of softmax by hand

- We compute the logits $\mathbf{o} = \mathbf{x}^{\mathrm{T}} \cdot \boldsymbol{\Omega} + \boldsymbol{\beta}$.
- We compute the probability vector $\mathbf{p} = \mathrm{softmax}(\mathbf{o})$.
- If the correct class is c, the loss on $\mathbf{x}$ is $\ell \stackrel{\text{def}}{=} -\log p_c$.
- We saw last week the **gradient** of this new loss function:

$$\mathbf{p}' \stackrel{\text{def}}{=} \frac{\partial \ell}{\partial o_j} = \begin{cases} p_j & \text{if } j \neq c \\ p_j - 1 & \text{if } j = c. \end{cases}$$

Computing the gradient of softmax by hand

- We compute the logits $\mathbf{o} = \mathbf{x}^{\mathrm{T}} \cdot \boldsymbol{\Omega} + \boldsymbol{\beta}$.
- We compute the probability vector $\mathbf{p} = \mathrm{softmax}(\mathbf{o})$.
- If the correct class is c, the loss on $\mathbf{x}$ is $\ell \stackrel{\text{def}}{=} -\log p_c$.
- We saw last week the **gradient** of this new loss function:

$$\mathbf{p}' \stackrel{\text{def}}{=} \frac{\partial \ell}{\partial o_j} = \begin{cases} p_j & \text{if } j \neq c \\ p_j - 1 & \text{if } j = c. \end{cases}$$

Reflection

Why is this the gradient?

TODO for next week

- Finish TP3.

TODO for next week

- Finish TP3.
- Read [P, Chapter 3].

TODO for next week

- Finish TP3.
- Read [P, Chapter 3].
- Review the reflection questions on the last two slides above.

References

[P] Prince, S., **Understanding Deep Learning**, https://udlbook.com

[Z] Zhang, A. et al., **Dive into Deep Learning**, https://www.d2l.ai

Image sources:

- https://en.wikipedia.org/wiki/Omega
- https://en.wikipedia.org/wiki/Nevel_(instrument)