Python for Data processing

# Lecture 3:
# Arrays, tensors and computations - Part III

Kosta Rozen

# What we already know

**A lot about NumPy arrays:**

- creation
- indexing
- universal functions
- linear algebra
- best practices
- I/O

# This lecture

**PyTorch:**

- basics
- operations
- gradients
- logistic regression

# Why NumPy is not enough

**NumPy arrays are great but:**

- they work only on CPU
- they provide only basic building blocks

**For deep learning:**

- CPU/GPU/TPU/?
- gradients

# PyTorch

- **tensors** provide the same operations as NumPy arrays
- work on CPU/**GPU**/**TPU**
- provide **autogradients**
- **deep learning** building blocks
- efficient **data loading**
- **deployment**

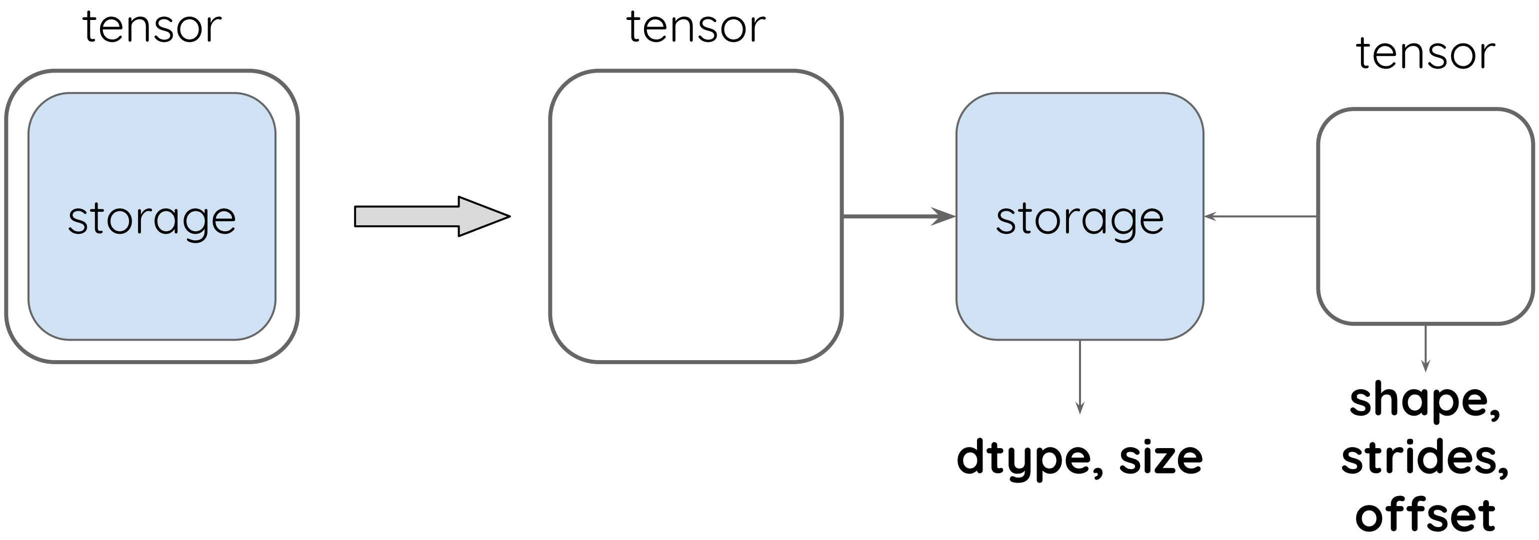
# PyTorch tensors

# Tensors

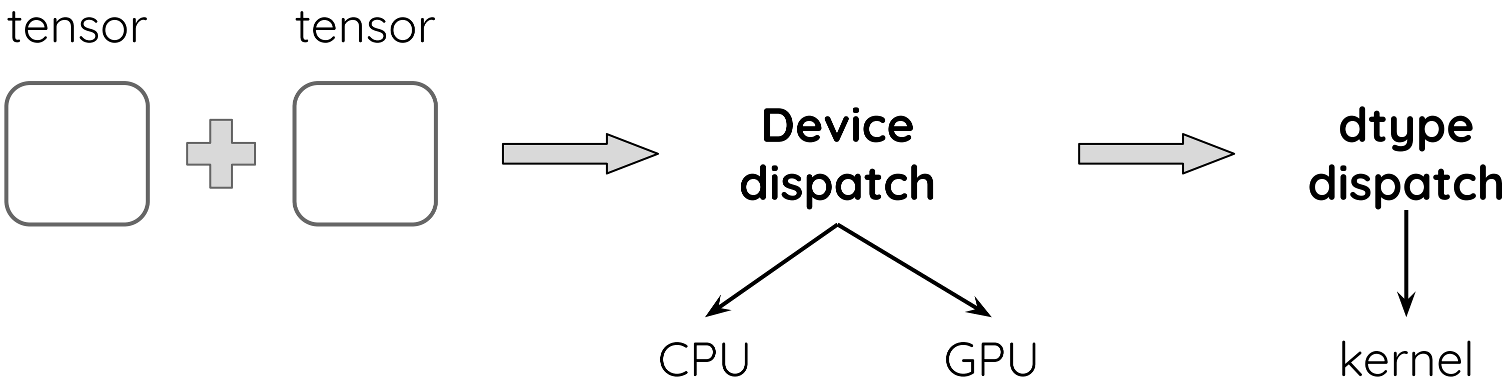
- similar to arrays, provide the same computational facilities
- can **share** data
- can live on **different devices**
- provide **declarative** computations

→let's try it out!

# Tensors and storage



# Devices and computations



# View, copies, reshaping

PyTorch is a bit more elaborated:

- `view`: always returns a view or fails
- `reshape`: returns either view or new tensor
- depends on contiguity constraints

# Gradients

In deep learning we need **gradients**:

- to calculate updates to network parameters (weights)
- no way to do that in NumPy
- an easy go in PyTorch (a bit more elaborated in Tensorflow)

$$L\left(a_{ij}\right) \text{ (scalar) } \rightarrow \frac{\partial L}{\partial a_{ij}} \text{ (tensor)}$$

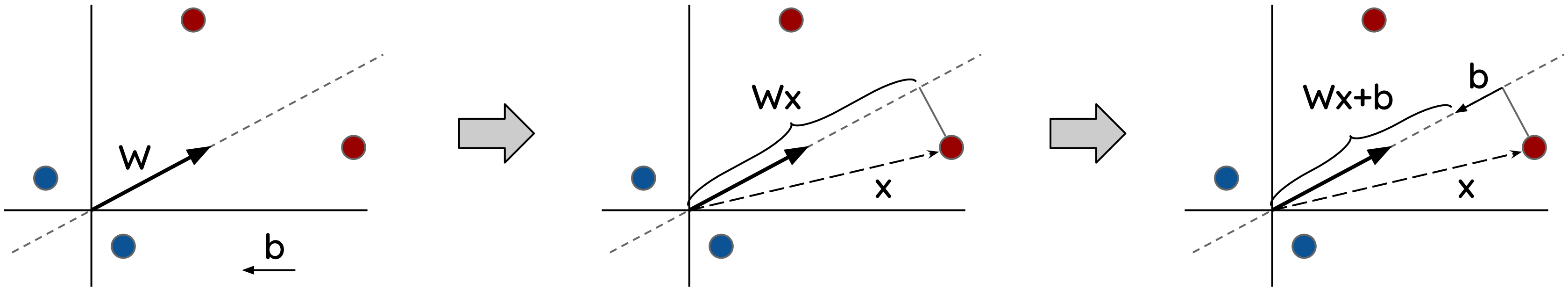
→let's try it out!

# Logistic regression with PyTorch

# Logistic regression: setup

- **two-dimensional** input (created with `make_blobs` from `sklearn.datasets`)
- **binary** classification with **linear** decision boundary
- output: **sigmoid**
- from **scratch**

# LR breakdown

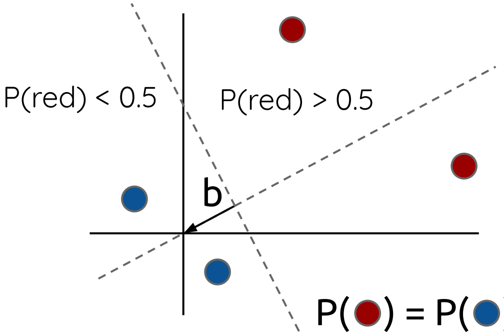



Scalar product of W and x: $Wx = W_0x_0+W_1x_1$

Add bias b: $W_0x_0+W_1x_1 + b$

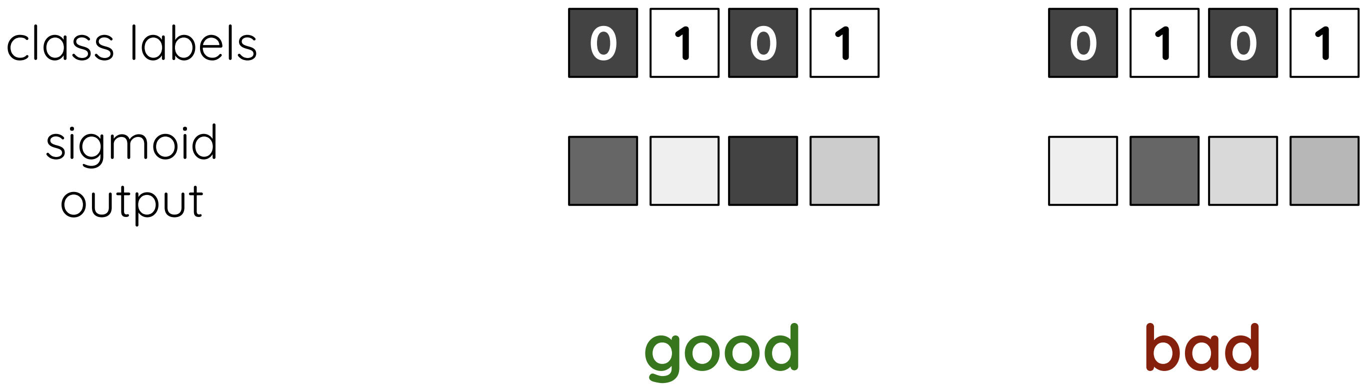
P(●) = σ(Wx+b)

Apply Sigmoid function:
If Wx+b > 0, then P(●) > 0.5
If Wx+b < 0, then P(●) < 0.5




P(●) = P(●) - decision boundary

# Log loss




**We want: probability** ↓ for class 0, probability ↑ for class 1

# Log loss function

Log loss is -1 * the log-likelihood function you learned in probability course as part of max-likelihood method for parameter estimation:

True value of sample k:
**1 for red** or **0 for blue**

Predicted probability
of sample k being **red**

Predicted probability
of sample k being **blue**

$$L = -\frac{1}{N}\sum_k (y_k \log(\hat{y}_k) + (1 - y_k)\log(1 - \hat{y}_k))$$

Number of
samples

$$\hat{y}_i = \sigma(X_{ik}W_k + b)$$

1 if sample
k is **blue**

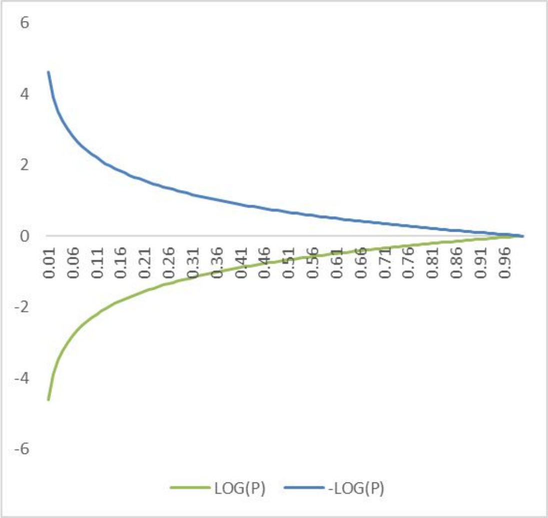
# Log loss function

$$L = -\frac{1}{N}\sum_k \left(y_k \log(\hat{y}_k) + (1 - y_k)\log(1 - \hat{y}_k)\right)$$

$$\hat{y}_i = \sigma(X_{ik}W_k + b)$$

Log parts of the function represent the cost of error when estimating probability:

- When we're right (p=1), log(p) is 0
- When we're wrong (small p), log(p) is negative and -log(p) is positive.
- The smaller p is, the larger the error (-log) will be.




→let's try it out!

# What we’ve learned

- PyTorch tensors and gradients
- how to perform simple gradient descent

# Assignment

- explore PyTorch tensor operations

questions?