

Deep Learning

Tutorial 2 – NumPy and the Coding Framework

Naama Pearl

Autonomous Vision Group

University of Tübingen

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN



Tübingen AI Center
tuebingen.ai



e l i a s
European Laboratory for Learning and Intelligent Systems

Outline

- ▶ Quiz
- ▶ Arrays for a batch of data
- ▶ Broadcasting
- ▶ Multiplying multi-dimensional arrays
- ▶ The weight gradient of a layer
- ▶ The Educational Framework (EDF)

Setup

Exercise 0

1. Download `tut_02.ipynb` from the course website and copy it into the `code` folder of sheet 1 (the notebook imports `edf.py` from there)
2. Open a terminal in that folder and run
 - ▶ `conda activate deep_learning_ex_1`
 - ▶ `jupyter lab`
3. Open `tut_02.ipynb` and run the first cell (`Shift + Enter`)

Arrays for a Batch of Data

A batch of data is an array (tensor)

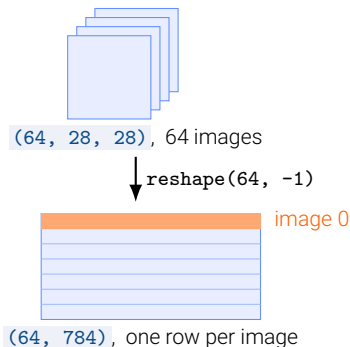
- ▶ A NumPy array has a **shape**, with one size per axis
- ▶ In deep learning, the samples are stacked into a **batch** along **axis 0**

```
import numpy as np
imgs = np.random.rand(64, 28, 28)  # 64 images

imgs.shape
imgs.ndim

X = imgs.reshape(64, -1)  # one row per image

X.shape
X.ndim
```



Indexing and slicing

- ▶ `X[i, j]` selects one entry, with one index per axis
- ▶ `X[start:end]` slices the range `start, ..., end-1` from that axis
- ▶ `step` is optional. `X[start:end:step]` returns the same range with jumps of `step`
- ▶ Each part can be dropped, the default is `start=0`, `end=size of the axis`, `step=1`
 - ▶ `:` alone selects the whole axis

```
# X is (64, 784), as before
```

```
X[0, 0]
```

```
X[0].shape
```

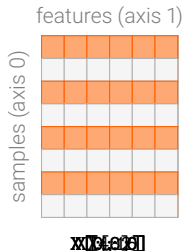
```
X[:, 0].shape
```

```
X[0:16].shape
```

```
X[48:].shape
```

```
X[:, :2].shape
```

- ▶ A single index removes its axis, a range keeps it



Elementwise operations and the matrix product

```
x = np.array([-1., 2., 3.])    # (3,)  
w = np.array([0.5, -1., 2.])  # (3,)  
  
np.exp(x)  
np.maximum(0, x)  
x * w
```

- ▶ `+`, `*`, `np.exp` and `np.maximum` work entry by entry
- ▶ In a network, the **activation functions** (sigmoid, ReLU) are elementwise

```
# X is (64, 784), as before  
A = np.random.randn(784, 10)    # (784, 10)  
x @ w  
(X @ A).shape
```

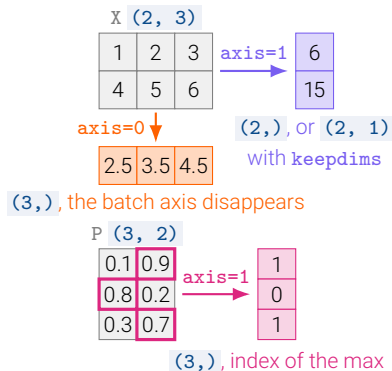
Reductions along an axis

```
X = np.array([[1., 2., 3.],
              [4., 5., 6.]])    # 2 samples, 3 features

X.mean()
X.mean(axis=0)
X.sum(axis=1)
X.sum(axis=1, keepdims=True)

P = np.array([[0.1, 0.9], [0.8, 0.2], [0.3, 0.7]]) # (3, 2)

np.argmax(P, axis=1)
```



- $\text{axis}=k$ is the axis that **disappears**. `keepdims=True` keeps it with size 1

`(B,)` and `(B, 1)` are different shapes

```
y = np.array([1., 0., 1.])
```

```
y.shape
```

```
y.reshape(-1, 1).shape
```

```
y[:, None].shape
```

```
Y = y.reshape(-1, 1)
```

```
Y[:, 0].shape
```

1	0	1
---	---	---

`(3,)`, a vector

1
0
1

`(3, 1)`, a column

- The numbers are the same, but `+` and `*` behave differently (next part)

Notebook, part 1

Exercise 1

- ▶ Fill in the shapes in the comments
- ▶ Write the code
- ▶ Every task prints **correct** when it is solved

Now

1.2 Predict the shapes of slices

More practice

- 1.1 Flatten the images into a matrix **X**
- 1.3 Cut the top left corner out of every image
- 1.4 Compute the mean pixel of every image
- 1.5 Turn the labels into a column, and back

Reminder

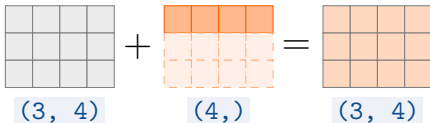
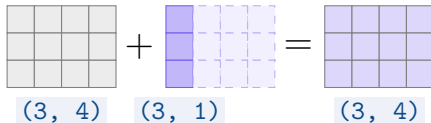
- ▶ `X.shape`, `imgs.reshape(64, -1)`
- ▶ `X[i, j]`, `X[start:end:step]`, `X[:, 0]`
- ▶ `X.mean(axis=1, keepdims=True)`
- ▶ `y[:, None]`, `y.reshape(-1, 1)`
- ▶ `Y[:, 0]`

Broadcasting

The broadcasting rule

Operations like + and * also work for arrays of **different shapes**.

- ▶ Compare the shapes **from the right**
- ▶ An array with fewer axes gets axes of size 1 on the left, `(4,)` becomes `(1, 4)`
- ▶ Two sizes fit if they are **equal** or one of them is **1**
- ▶ An axis of size 1 is repeated to the other size



The repeated copies (dashed) are not stored. NumPy only computes as if they were there.

Broadcasting in a layer

A layer for a batch, from `edf.py | Affine.forward`

```
self.value = np.matmul(self.x.value, self.w.A.value) + self.w.b.value  
# the addition broadcasts b over the batch.
```

Code `x.value @ w.A.value + w.b.value`

Meaning samples @ weights + bias

Shapes `(B, D)` @ `(D, K)` + `(K,)` → `(B, K)`

► B samples with D features each, and a layer with K outputs

► `(B, K)` + `(K,)` adds the same bias to every sample



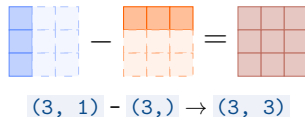
`(B, K)` + `(K,)` → `(B, K)`

here $B = 4$, $K = 3$

Subtracting $(B,)$ from $(B, 1)$

```
y_hat = np.array([[1.], [2.], [3.]]) # (3, 1)
y = np.array([1., 2., 3.])          # (3,)

d = y_hat - y
np.mean(d ** 2)
d = y_hat - y.reshape(-1, 1)
np.mean(d ** 2)
```



- ▶ $(3, 1)$ and $(3,)$ fit the rule, so the result is $(3, 3)$ with all differences $\hat{y}_i - y_j$
- ▶ The loss is wrong, but the training still runs
- ▶ Checking shapes in the code helps, for example `assert d.shape == y_hat.shape`

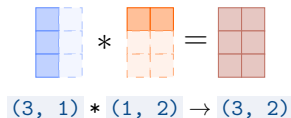
Introducing a new axis

```
a = np.array([1., 2., 3.])      # (3,)  
c = np.array([10., 20.])      # (2,)
```

```
a[:, None].shape
```

```
c[None, :].shape
```

```
a[:, None] * c[None, :]
```



- ▶ `None` (or `np.newaxis`) inserts an axis of size 1 at its position
- ▶ The matrix of all products $a_i c_j$ is the **outer product** ac^T
- ▶ The same works with more axes (next part)

Notebook, part 2

Exercise 2

- ▶ Fill in the shapes in the comments
- ▶ Write the code
- ▶ Every task prints `correct` when it is solved

Now

2.1 Predict the shapes of broadcasts

2.4 Find the bug in `squared_error`

More practice

2.2 Center the images

2.3 Scale every image to a largest pixel of 1

2.5 Compute the table of all products, with `None`

Reminder

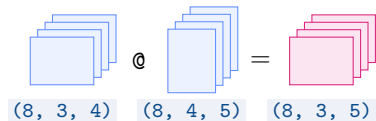
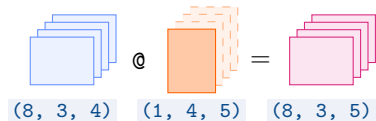
- ▶ Shapes from the right, sizes equal or 1
- ▶ `X - X.mean(axis=0)` `(B, D)` - `(D,)`
- ▶ `imgs.max(axis=(1, 2), keepdims=True)`
- ▶ `y.reshape(-1, 1)`
- ▶ `a[:, None] * c[None, :]`

Multiplying Multi-Dimensional Arrays

Three kinds of products

	What it does	Example
$A * B$	entry by entry, shapes broadcast	$(B, K) * (K,) \rightarrow (B, K)$
$A @ B$, 2 axes	rows times columns	$(B, D) @ (D, K) \rightarrow (B, K)$
$A @ B$, 3+ axes	a stack of matrix products	$(8, 3, 4) @ (8, 4, 5) \rightarrow (8, 3, 5)$

- ▶ With 3 or more axes, the **last two** axes are the matrices
- ▶ The leading axes **broadcast**. $(8, 3, 4) @ (1, 4, 5)$ (or $(4, 5)$) uses the same 4×5 matrix for all 8



Notebook, part 3

Exercise 3

- ▶ Fill in the shapes in the comments
- ▶ Write the code
- ▶ Every task prints **correct** when it is solved

Now

3.1 Predict the shapes of products

More practice

3.2 Compute the forward pass of a layer with a sigmoid

Reminder

▶ $(B, D) @ (D, K) \rightarrow (B, K)$

▶ $(8, 3, 4) @ (8, 4, 5) \rightarrow (8, 3, 5)$

The Weight Gradient of a Layer

The weight gradient of a layer, one sample

$$\mathbf{x} (D,) \quad \mathbf{A} (D, K) \quad \mathbf{b} (K,) \quad y = \mathbf{x} @ \mathbf{A} + \mathbf{b} (K,)$$

- ▶ We want $\partial L / \partial \mathbf{A}$, the gradient of the loss for the weights
- ▶ $\mathbf{g} = \partial L / \partial y (K,)$ is already given, from the layer above
- ▶ By the chain rule, what is left is how y depends on $\mathbf{A}$, $\partial y / \partial \mathbf{A}$
- ▶ The entry $\mathbf{A}[\mathbf{d}, \mathbf{k}]$ connects input $\mathbf{d}$ to output $\mathbf{k}$, with $\partial y_k / \partial A_{dk} = x_d$
- ▶ So its gradient is $\mathbf{x}[\mathbf{d}] * \mathbf{g}[\mathbf{k}]$, the input times the gradient of the output
- ▶ All entries at once are a table of all products, the **outer product** of $\mathbf{x}$ and $\mathbf{g}$ (slide 14)

Code $\mathbf{x}[:, \text{None}] * \mathbf{g}[\text{None}, :]$

Meaning inputs * gradients of the outputs

Shapes $(D, 1) * (1, K) \rightarrow (D, K)$



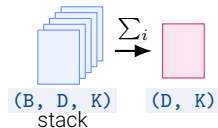
$$(D, 1) * (1, K), \quad D = 4, K = 3$$

The weight gradient of a layer for a batch

$$X (B, D) \quad A (D, K) \quad b (K,) \quad Y = X @ A + b (B, K)$$

$G (B, K)$ comes from the layer above, one row g per sample

- ▶ Sample i gives the outer product of $X[i]$ and $G[i]$ (D, K)
- ▶ We **stack** them along the batch axis into (B, D, K)
- ▶ The gradient for A for the whole batch is the **sum** of the stack over the samples



Code `X[:, :, None] * G[:, None, :] .sum(axis=0)`

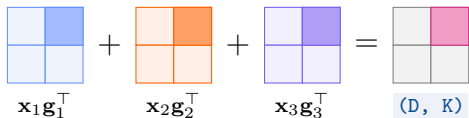
Meaning inputs * gradients from above sum over the batch

Shapes $(B, D, 1)$ * $(B, 1, K)$ $\rightarrow (B, D, K) \rightarrow (D, K)$

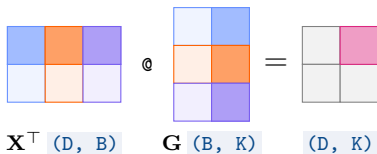
- ▶ `edf.py | Affine.backward` computes the stack, `edf.py | Parameter.addgrad` sums it
- ▶ $X.T @ G$ gives the same result, faster. $@$ sums over the shared axis B

Broadcasting with a sum and the matrix product

one outer product per sample, then `.sum(axis=0)`


$$\mathbf{x}_1 \mathbf{g}_1^\top + \mathbf{x}_2 \mathbf{g}_2^\top + \mathbf{x}_3 \mathbf{g}_3^\top = \text{matrix } (D, K)$$

`X.T @ G`


$$\mathbf{X}^\top (D, B) @ \mathbf{G} (B, K) = \text{matrix } (D, K)$$

- ▶ $B = 3$ samples, $D = K = 2$. Every color is one sample
- ▶ Entry (d, k) of the sum is $\sum_i X_{id} G_{ik}$, one product per sample
- ▶ In $\mathbf{X}^\top @ \mathbf{G}$, entry (d, k) is the dot product of row d of $\mathbf{X}^\top$ and column k of $\mathbf{G}$
- ▶ The dot product runs over the shared axis B , so it is the same sum $\sum_i X_{id} G_{ik}$
- ▶ `@` adds the products as it goes and never stores the stack (B, D, K)

Notebook, part 4

Exercise 4

- ▶ Fill in the shapes in the comments
- ▶ Write the code
- ▶ Every task prints `correct` when it is solved

Now

4.2 The batch. Compute the stack of outer products, its sum, and the same gradient with one matrix product

More practice

4.1 The gradient for **A** of one sample, as an outer product

Reminder

- ▶ $A[d, k]$ connects input d to output k
- ▶ Its gradient is $x[d] * g[k]$
- ▶ $x[:, None] * g[None, :]$ $(D, 1) * (1, K)$
- ▶ The batch: one outer product per sample, summed over the batch

The Educational Framework (EDF)

The edf.py file

[edf.py](#)

EDF stands for **Educational Framework**, a small deep learning library written with NumPy.

Globals	<code>DT = np.float32</code>
settings, lists	<code>learning_rate = .001</code>
	<code>def clear_compgraph(): # CompNodes = [], Parameters = []</code>
Run	<code>def Forward(): # all nodes, in the order they were created</code>
	<code>def Backward(loss): # all gradients to 0, all nodes in reverse order</code>
	<code>def UpdateParameters(): # value -= learning_rate * grad</code>
Data	<code>class Input: # holds the data in .value</code>
	<code>class Parameter: # a learnable array, updated by UpdateParameters</code>
	<code>class AffineParams: # A and b of a layer</code>
Nodes	<code>class CompNode: # the base class, a node adds itself to CompNodes</code>
	<code>class Affine(CompNode): # also Sigmoid, Softmax, CrossEntropyLoss</code>

The order of creation is the order of the forward pass, since a node needs its inputs when it is created.

`Backward()` starts with $L.grad = 1/B$, so every gradient is a mean over the batch.

Building and training a network with EDF

a notebook with `import edf`

Setup

empty graph

```
import edf
edf.clear_compgraph()
```

Inputs

shape from `.value`

```
x = edf.Input(); x.value = X          # (B, D)
y = edf.Input(); y.value = labels     # (B,)
```

Layers

parameters, nodes

```
params_1 = edf.AffineParams(D, H)     # A (D, H) and b (H,)
params_2 = edf.AffineParams(H, K)     # A (H, K) and b (K,)
affine1 = edf.Affine(params_1, x)     # (B, H)
h = edf.Sigmoid(affine1)              # (B, H)
affine2 = edf.Affine(params_2, h)     # (B, K)
p = edf.Softmax(affine2)              # (B, K)
```

Loss

one per sample

```
L = edf.CrossEntropyLoss(p, y)       # (B,)
```

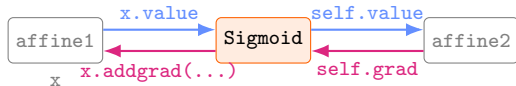
Training loop

repeat many times

```
for i in range(iterations):
    edf.Forward()          # all nodes, in creation order
    edf.Backward(L)        # all nodes, in reverse order
    edf.UpdateParameters() # value -= learning_rate * grad
```

An example node

edf.py

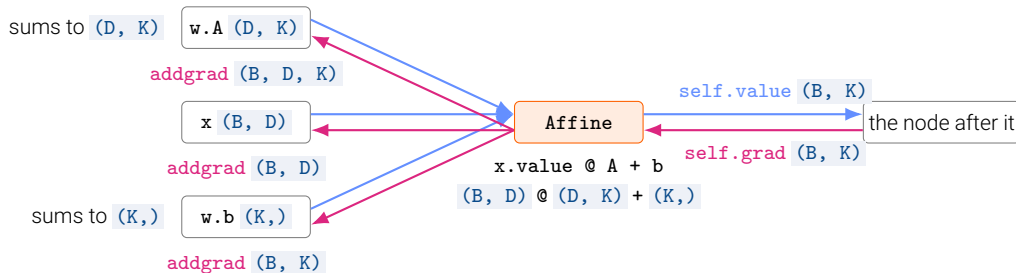


- ▶ `forward()` reads `x.value` and writes `self.value`
- ▶ `backward()` gets the complete `self.grad` $= \frac{\partial \mathcal{L}}{\partial \text{value}}$ from the nodes after it, and passes $\frac{\partial \mathcal{L}}{\partial \text{value}} \frac{\partial \text{value}}{\partial \mathbf{x}}$ to `x.addgrad`
- ▶ `addgrad` **adds**, because a node can feed several nodes (lecture 2)
- ▶ `Parameter.addgrad` also sums over the batch

```
class Sigmoid(CompNode):  
    def __init__(self, x):  
        CompNodes.append(self)  
        self.x = x  
  
    def forward(self):  
        self.value = 1 / (1 + np.exp(-self.x.value))  
  
    def backward(self):  
        self.x.addgrad(self.grad  
                        * self.value * (1 - self.value))
```

```
class Parameter:  
    def addgrad(self, delta):  
        self.grad += np.sum(delta, axis=0)
```

The Affine node

[edf.py](#)

```
def forward(self):
    self.value = np.matmul(self.x.value, self.w.A.value) + self.w.b.value
def backward(self):
    self.x.addgrad(np.matmul(self.grad, self.w.A.value.transpose()))
    self.w.b.addgrad(self.grad)
    self.w.A.addgrad(self.x.value[:, :, np.newaxis] * self.grad[:, np.newaxis, :])
```

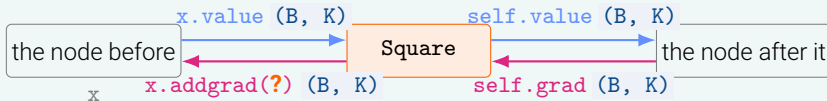
- `Parameter.addgrad` sums the gradients of the samples over the batch

Write a Square node

Exercise 5

Write the node `Square`, $y_{bk} = x_{bk}^2$ for every entry, x, y (B, K)

1. Write `forward()`
2. Compute $\frac{\partial y_{bk}}{\partial x_{bk}}$ and write `backward()` with one `addgrad` call
3. Run the check of the task



In `tut_02.ipynb`, **task 5.1**.