Deep Learning with JAX

Introduction

Marco Forgione

IDSIA USI-SUPSI, Lugano, Switzerland

Getting started

JAX has an interface that behaves like numpy (or MATLAB). Let's compute:

$$\begin{bmatrix}1 & 1\end{bmatrix} + \begin{bmatrix}1 & 2\end{bmatrix} = \begin{bmatrix}2 & 3\end{bmatrix}$$

Getting started

JAX has an interface that behaves like numpy (or MATLAB). Let's compute:

$$\begin{bmatrix}1 & 1\end{bmatrix} + \begin{bmatrix}1 & 2\end{bmatrix} = \begin{bmatrix}2 & 3\end{bmatrix}$$

MATLAB

```
a = ones(1, 2);
b = [1.0, 2.0];
a + b
```

```
[2 3]
```

JAX

```
a = jnp.ones((2,))
b = jnp.array([1.0, 2.0])
a + b
```

```
Array([2., 3.], dtype=float32)
```

Getting started

JAX has an interface that behaves like numpy (or MATLAB). Let's compute:

$$[1 \quad 1] + [1 \quad 2] = [2 \quad 3]$$

MATLAB

```
a = ones(1, 2);
b = [1.0, 2.0];
a + b
```

```
[2 3]
```

JAX

```
a = jnp.ones((2,))
b = jnp.array([1.0, 2.0])
a + b
```

```
Array([2., 3.], dtype=float32)
```

- JAX is slightly more verbose. The price of a general-purpose language like Python.

Getting started

JAX has an interface that behaves like numpy (or MATLAB). Let's compute:

$$\begin{bmatrix}1 & 1\end{bmatrix} + \begin{bmatrix}1 & 2\end{bmatrix} = \begin{bmatrix}2 & 3\end{bmatrix}$$

MATLAB

```
a = ones(1, 2);
b = [1.0, 2.0];
a + b
```

```
[2 3]
```

JAX

```
a = jnp.ones((2,))
b = jnp.array([1.0, 2.0])
a + b
```

```
Array([2., 3.], dtype=float32)
```

- JAX is slightly more verbose. The price of a general-purpose language like Python.
- JAX allows you define a generic 1D vector (neither a row or a column).

Getting started

JAX has an interface that behaves like numpy (or MATLAB). Let's compute:

$$\begin{bmatrix}1 & 1\end{bmatrix} + \begin{bmatrix}1 & 2\end{bmatrix} = \begin{bmatrix}2 & 3\end{bmatrix}$$

MATLAB

```
a = ones(1, 2);
b = [1.0, 2.0];
a + b
```

```
[2 3]
```

JAX

```
a = jnp.ones((2,))
b = jnp.array([1.0, 2.0])
a + b
```

```
Array([2., 3.], dtype=float32)
```

- JAX is slightly more verbose. The price of a general-purpose language like Python.
- JAX allows you define a generic 1D vector (neither a row or a column).
- A few more subtle differences...

Function definition

- Define the function:

$$f(p, x) : \mathbb{R}^2 \times \mathbb{R} \mapsto \mathbb{R} = p_1 x + p_2$$

Function definition

- Define the function:

$$f(p, x) : \mathbb{R}^2 \times \mathbb{R} \mapsto \mathbb{R} = p_1 x + p_2$$

MATLAB

```
function y = f(p, x)
    y = p(1) * x + p(2);
end
```

JAX

```
def f(p, x):
    y = p[0] * x  + p[1]
    return y
```

We shall interpret f as the model structure, p as the parameter, x as the input.

Function definition

- Define the function:

$$f(p,x) : \mathbb{R}^2 \times \mathbb{R} \mapsto \mathbb{R} = p_1 x + p_2$$

MATLAB

```
function y = f(p, x)
    y = p(1) * x + p(2);
end
```

JAX

```
def f(p, x):
    y = p[0] * x  + p[1]
    return y
```

We shall interpret f as the model structure, p as the parameter, x as the input.

- Apply f with "true" $p^o = [1.0 \;\; 2.0]$ input data point $x = 0.5$

Function definition

- Define the function:

$$f(p, x) : \mathbb{R}^2 \times \mathbb{R} \mapsto \mathbb{R} = p_1 x + p_2$$

MATLAB

```
function y = f(p, x)
    y = p(1) * x + p(2);
end
```

JAX

```
def f(p, x):
    y = p[0] * x  + p[1]
    return y
```

We shall interpret f as the model structure, p as the parameter, x as the input.

- Apply f with "true" $p^o = [1.0 \;\; 2.0]$ input data point $x = 0.5$

```
p_o = [1.0, 2.0];
x = 0.5;
f(p_o, x)
```

2.5000

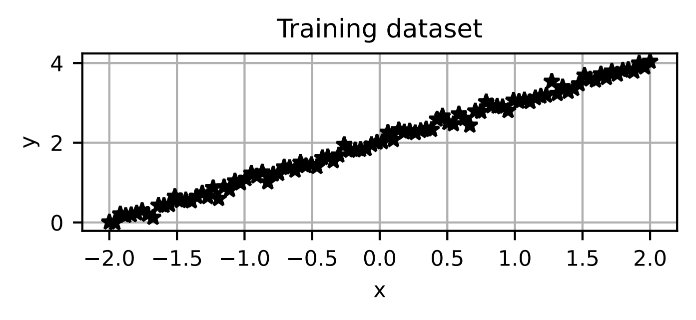
```
p_o = jnp.array([1.0, 2.0])
x = jnp.array(0.5)
y = f(p_o, x); y
```

Array(2.5, dtype=float32)

Linear regression dataset

- Apply f with $p = p^o$ to $N = 100$ linearly spaced points in $[-2 \;\; 2]$, add noise.

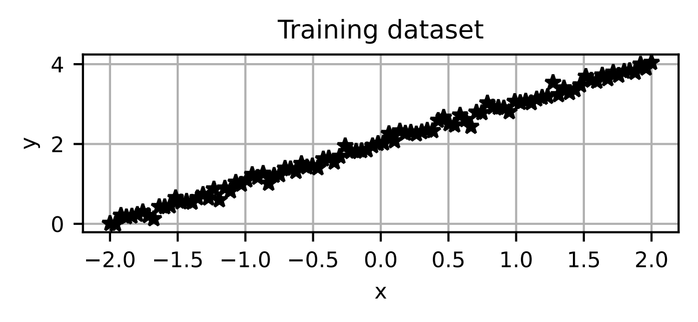
```
N = 100
x = jnp.linspace(-2, 2, N)
y = f(p_o, x) + jr.normal(key_e, (N,)) * 0.1
```



Linear regression dataset

- Apply f with $p = p^o$ to $N = 100$ linearly spaced points in $[-2 \;\; 2]$, add noise.

```
N = 100
x = jnp.linspace(-2, 2, N)
y = f(p_o, x) + jr.normal(key_e, (N,)) * 0.1
```



Function `f` works both with scalar and vector input `x`. Useful feature...

Loss definition

The Mean Squared Error (MSE) loss is:

$$\mathcal{L}(p, y, x) : \mathbb{R}^{n_p} \times \mathbb{R}^N \times \mathbb{R}^N \mapsto \mathbb{R} = \frac{1}{N} \sum_{i=1}^{N} \left(y_i - f(p, y_i, x_i)\right)^2$$

with $n_p = 2$ parameters.

Loss definition

The Mean Squared Error (MSE) loss is:

$$\mathcal{L}(p, y, x) : \mathbb{R}^{n_p} \times \mathbb{R}^N \times \mathbb{R}^N \mapsto \mathbb{R} = \frac{1}{N} \sum_{i=1}^{N} \left(y_i - f(p, y_i, x_i)\right)^2$$

with $n_p = 2$ parameters.

We exploit that `f` can process a vector input `x`:

```
def loss_fn(p, y, x):
    y_hat = f(p, x) # works with vector x
    loss = jnp.mean((y - y_hat) ** 2)
    return loss
```

Loss definition

The Mean Squared Error (MSE) loss is:

$$\mathcal{L}(p, y, x) : \mathbb{R}^{n_p} \times \mathbb{R}^N \times \mathbb{R}^N \mapsto \mathbb{R} = \frac{1}{N} \sum_{i=1}^{N} \left(y_i - f(p, y_i, x_i)\right)^2$$

with $n_p = 2$ parameters.

We exploit that `f` can process a vector input `x`:

```
def loss_fn(p, y, x):
    y_hat = f(p, x) # works with vector x
    loss = jnp.mean((y - y_hat) ** 2)
    return loss
```

```
p_hat = jr.normal(key_p, shape=(2,))
loss_fn(p_hat, y, x)
```

```
Array(1.7047384, dtype=float32)
```

Automatic differentiation in JAX

- For gradient-based optimization, we need the gradient:

$$\nabla_1 \mathcal{L}(p, y, x) : \mathbb{R}^{n_p} \times \mathbb{R}^N \times \mathbb{R}^N \mapsto \mathbb{R}^{n_p},$$

  i.e. the derivative of $\mathcal{L}$ with respect to its first argument: p.

Automatic differentiation in JAX

- For gradient-based optimization, we need the gradient:

$$\nabla_1 \mathcal{L}(p, y, x) : \mathbb{R}^{n_p} \times \mathbb{R}^N \times \mathbb{R}^N \mapsto \mathbb{R}^{n_p},$$

  i.e. the derivative of $\mathcal{L}$ with respect to its first argument: p.

- The main point of JAX (PyTorch, ...) is **automatic differentiation**. Efficient & effortless numerical evaluation of derivatives of interest!

Automatic differentiation in JAX

- For gradient-based optimization, we need the gradient:

$$\nabla_1 \mathcal{L}(p, y, x) : \mathbb{R}^{n_p} \times \mathbb{R}^N \times \mathbb{R}^N \mapsto \mathbb{R}^{n_p},$$

  i.e. the derivative of $\mathcal{L}$ with respect to its first argument: p.

- The main point of JAX (PyTorch, ...) is **automatic differentiation**. Efficient & effortless numerical evaluation of derivatives of interest!

```
grad_fn = jax.grad(loss_fn, 0) # gradient wrt 1st agrument
```

Automatic differentiation in JAX

- For gradient-based optimization, we need the gradient:

$$\nabla_1 \mathcal{L}(p, y, x) : \mathbb{R}^{n_p} \times \mathbb{R}^N \times \mathbb{R}^N \mapsto \mathbb{R}^{n_p},$$

  i.e. the derivative of $\mathcal{L}$ with respect to its first argument: p.

- The main point of JAX (PyTorch, ...) is **automatic differentiation**. Efficient & effortless numerical evaluation of derivatives of interest!

```
grad_fn = jax.grad(loss_fn, 0) # gradient wrt 1st agrument
```

The function `grad_fn` can be called on arbitraty arguments:

```
grad_fn(p_hat, y, x)
```

```
Array([-1.0732304, -2.4366264], dtype=float32)
```

Fitting a model in JAX

With automatic differentiation, setting up gradient descent is a piece of cake.

```
p_hat = jr.normal(key_p, shape=(2,))
lr = 1e-2 # learning rate

for i in range(200):
    g = grad_fn(p_hat, y, x)
    p_hat = p_hat - lr * g
```

$$\hat{p}^{k+1} = \hat{p}^{k} - \lambda \nabla_1 \mathcal{L}(\hat{p}^{k}, y, x)$$

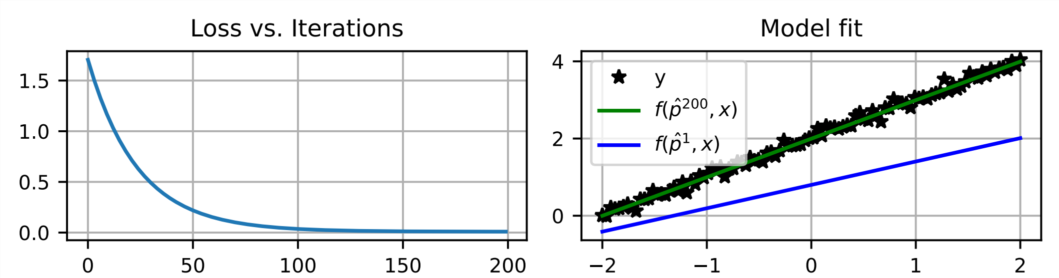
Fitting a model in JAX

With automatic differentiation, setting up gradient descent is a piece of cake.

```
p_hat = jr.normal(key_p, shape=(2,))
lr = 1e-2 # learning rate

for i in range(200):
    g = grad_fn(p_hat, y, x)
    p_hat = p_hat - lr * g
```

$$\hat{p}^{k+1} = \hat{p}^{k} - \lambda \nabla_1 \mathcal{L}(\hat{p}^{k}, y, x)$$



Exercises

- Open the accompanying notebook: `code/01_jax_intro.ipynb`
- Familiarize with the code & environment
- Verify that the optimized $\hat{p}$ is close to p^o
- Save and visualize the loss vs. iteration
- Verify that the gradient returned by JAX is correct