

MLX-RS: Basic Operations

This documentation provides a comprehensive guide to using the **mlx-rs** API for array operations, building neural networks (MLP and CNN), and implementing training loops.

1. Basic Array Operations

The foundation of **mlx-rs** is the `Array` struct. Operations in MLX are **lazy**, meaning they are recorded in a computation graph and only executed when explicitly evaluated.

Core Concepts

- **Lazy Evaluation:** Use `.eval()` or `Array::eval_all()` to trigger GPU computation.
- **Dtype:** Specifies the data type (e.g., `Float32`, `Int32`).
- **Device Management:** Set the default device (CPU/GPU) at the start of your program.

Common Methods

Method	Arguments	Description
<code>from_slice<T></code>	<code>data: &[T]</code> , <code>shape: &[i32]</code> , <code>dtype: Dtype</code>	Creates an array from a Rust slice.
<code>add</code>	<code>other: &Array</code>	Element-wise addition.
<code>multiply</code>	<code>other: &Array</code>	Element-wise multiplication.
<code>matmul</code>	<code>other: &Array</code>	Matrix multiplication (supports broadcasting).
<code>reshape</code>	<code>shape: &[i32]</code>	Returns a new array with a different shape.
<code>eval</code>	None	Triggers execution for the specific array.

Example implementation

Rust

```

use mlx::{Array, Dtype, Result, Device, DeviceType};

fn main() -> Result<()> {
    let gpu = Device::new(DeviceType::Gpu);
    gpu.set_default()?;

    // Initialize arrays
    let a = Array::from_slice(&[1.0, 2.0, 3.0, 4.0], &[4],
Dtype::Float32)?;
    let b = Array::from_slice(&[5.0, 6.0, 7.0, 8.0], &[4],
Dtype::Float32)?;

    // Operations (Lazy)
    let sum = a.add(&b)?;
    let dot = a.matmul(&b)?; // Dot product for 1D arrays

    // Compute on GPU
    sum.eval()?;

    println!("Sum: {}", sum);
    Ok(())
}

```

2. Multi-Layer Perceptron (MLP)

For building neural networks, use the `nn` module. This involves defining a model, an optimizer, and a training loop using automatic differentiation.

Key API Components

- **Sequential:** A container to stack layers.
- **Linear:** `Linear::new(input_dims: i32, output_dims: i32, bias: bool, key: &Array)`
- **value_and_grad:** A transform that returns both the output of a function (loss) and the gradients of the inputs.
- **Adam:** A standard optimizer.

Implementation Guide

When using `value_and_grad`, you must link the "tracer" arrays provided by the transform back to your model parameters using `update_parameters(p)`.

```
let model = RefCell::new(Sequential::new(vec![
    Box::new(Linear::new(10, 32, true, &key)?),
    Box::new(ReLU::new()),
    Box::new(Linear::new(32, 5, true, &key)?),
]));

let mut optimizer = Adam::new(1e-3, &model.borrow().parameters_owned())?;
```

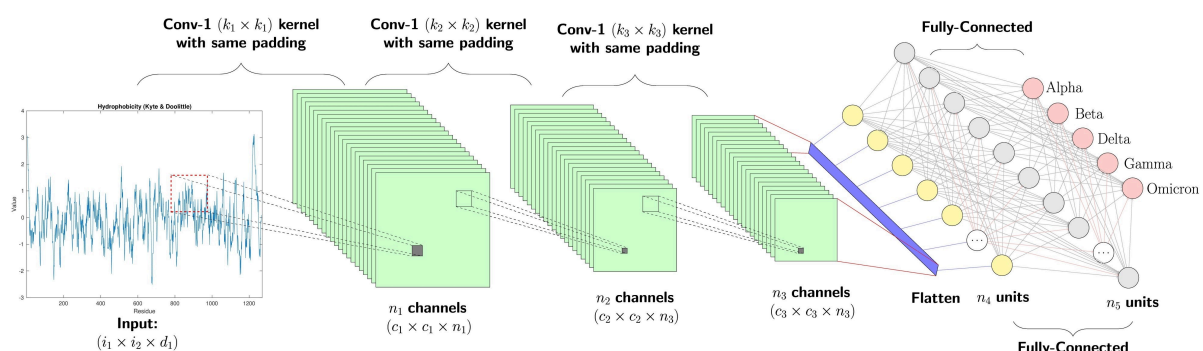
3. Convolutional Neural Networks (CNN)

CNNs introduce spatial operations. In **mlx-rs**, `Conv2d` expects inputs in the shape `[Batch, Height, Width, Channels]`.

Conv2d Arguments

`Conv2d::new(in_channels, out_channels, kernel_size, stride, padding, dilation, groups, bias, key)`

- **kernel_size / stride / padding:** Takes `[i32; 2]` (e.g., `[3, 3]`).
- **Flatten:** Flattens a multi-dimensional tensor into a 2D tensor `[Batch, Features]`.



Shutterstock

Handling One-Hot Targets

If your labels are integers, convert them to One-Hot encoding to provide a "hard target" for the `cross_entropy` loss function.

```
// Create labels [0, 9] for 8 images
let raw_labels = Array::random_uniform(&[8], 0.0, 10.0, Dtype::Float32, &key)?.cast(Dtype::Int32)?;
let range = Array::arange(0.0, 10.0, 1.0, Dtype::Float32)?;

// Broadcast comparison to create [8, 10] one-hot matrix
let targets = raw_labels.reshape(&[8, 1])?
    .equal(&range.reshape(&[1, 10])?)?
    .cast(Dtype::Float32)?;
```

4. The Training Loop Logic

Because **mlx-rs** utilizes the C-pointer architecture of MLX, updating parameters follows a specific sequence:

1. **Capture Parameters:** Get the current weights as a vector.
2. **Trace Gradients:** Use `value_and_grad`. Inside the closure, update the model with the tracer arrays `p`.
3. **Optimize:** Call `optimizer.update`. You must pass a vector of mutable references: `params.iter_mut().collect()`.
4. **Sync Model:** Call `model.update_parameters(¶ms)` to write the new weights back into the model layers.
5. **Eval:** Call `Array::eval_all` on parameters and the loss to flush the computation to the GPU.

Training Loop

```
for step in 0..1000 {
    let mut params = model.borrow().parameters_owned();

    let (loss, grads) = transforms::value_and_grad(|p: &[Array]| {
        let mut m = model.borrow_mut();
        m.update_parameters(p);
```

```
        let logits = m.forward(&x)?;  
        cross_entropy(&logits, &targets)  
    }, &params)?;  
  
    optimizer.update(params.iter_mut().collect(), grads)?;  
    model.borrow_mut().update_parameters(&params);  
  
    Array::eval_all(&[&params, &[loss.clone()]].concat())?;  
}
```