Print to PDF

MLX

MLX is a NumPy-like array framework designed for efficient and flexible machine learning on Apple silicon, brought to you by Apple machine learning research.

The Python API closely follows NumPy with a few exceptions. MLX also has a fully featured C++ API which closely follows the Python API.

The main differences between MLX and NumPy are:

- **Composable function transformations**: MLX has composable function transformations for automatic differentiation, automatic vectorization, and computation graph optimization.
- **Lazy computation**: Computations in MLX are lazy. Arrays are only materialized when needed.
- **Multi-device**: Operations can run on any of the supported devices (CPU, GPU, ...)

The design of MLX is inspired by frameworks like PyTorch, Jax, and ArrayFire. A notable difference from these frameworks and MLX is the *unified memory model*. Arrays in MLX live in shared memory. Operations on MLX arrays can be performed on any of the supported device types without performing data copies. Currently supported device types are the CPU and GPU.

Install

Build and Install

Usage

Quick Start Guide

Lazy Evaluation

Unified Memory

Indexing Arrays

Saving and Loading Arrays

Function Transforms

Skip to main content

Conversion to NumPy and Other Frameworks

Using Streams

Examples

Linear Regression

Multi-Layer Perceptron

LLM inference

Python API Reference

Array

Devices and Streams

Operations

Random

Transforms

FFT

Linear Algebra

Metal

Neural Networks

Optimizers

Tree Utils

C++ API Reference

Operations

Further Reading

Developer Documentation

Next
Build and Install ›

Print to PDF

Build and Install

Contents

- Python Installation
- Build from source

Python Installation

MLX is available on PyPI. All you have to do to use MLX with your own Apple silicon computer is

```
pip install mlx
```

To install from PyPI you must meet the following requirements:

- Using an M series chip (Apple silicon)
- Using a native Python >= 3.8
- macOS >= 13.5

Note

MLX is only available on devices running macOS >= 13.5 It is highly recommended to use macOS 14 (Sonoma)

MLX is also available on conda-forge. To install MLX with conda do:

```
conda install conda-forge::mlx
```

Troubleshooting

Skip to main content

Probably you are using a non-native Python. The output of

```
python -c "import platform; print(platform.processor())"
```

should be `arm`. If it is `i386` (and you have M series machine) then you are using a non-native Python. Switch your Python to a native Python. A good way to do this is with Conda.

Build from source

Build Requirements

- A C++ compiler with C++17 support (e.g. Clang >= 5.0)
- cmake – version 3.24 or later, and `make`
- Xcode >= 15.0 and macOS SDK >= 14.0

> **Note**
>
> Ensure your shell environment is native `arm`, not `x86` via Rosetta. If the output of `uname -p` is `x86`, see the troubleshooting section below.

Python API

To build and install the MLX python library from source, first, clone MLX from its GitHub repo:

```
git clone git@github.com:ml-explore/mlx.git mlx && cd mlx
```

Make sure that you have pybind11 installed. You can install `pybind11` with `pip`, `brew` or `conda` as follows:

```
pip install "pybind11[global]"
conda install pybind11
brew install pybind11
```

Then simply build and install it using pip:

Skip to main content

```
env CMAKE_BUILD_PARALLEL_LEVEL="" pip install .
```

For developing use an editable install:

```
env CMAKE_BUILD_PARALLEL_LEVEL="" pip install -e .
```

To make sure the install is working run the tests with:

```
pip install ".[testing]"
python -m unittest discover python/tests
```

Optional: Install stubs to enable auto completions and type checking from your IDE:

```
pip install ".[dev]"
python setup.py generate_stubs
```

C++ API

Currently, MLX must be built and installed from source.

Similarly to the python library, to build and install the MLX C++ library start by cloning MLX from its GitHub repo:

```
git clone git@github.com:ml-explore/mlx.git mlx && cd mlx
```

Create a build directory and run CMake and make:

```
mkdir -p build && cd build
cmake .. && make -j
```

Run tests with:

```
make test
```

Install with:

```
make install
```

Skip to main content

Note that the built `mlx.metallib` file should be either at the same directory as the executable statically linked to `libmlx.a` or the preprocessor constant `METAL_PATH` should be defined at build time and it should point to the path to the built metal library.

Build Options

Option	Default
MLX_BUILD_TESTS	ON
MLX_BUILD_EXAMPLES	OFF
MLX_BUILD_BENCHMARKS	OFF
MLX_BUILD_METAL	ON
MLX_BUILD_PYTHON_BINDINGS	OFF

Note

If you have multiple Xcode installations and wish to use a specific one while building, you can do so by adding the following environment variable before building

```
export DEVELOPER_DIR="/path/to/Xcode.app/Contents/Developer/"
```

Further, you can use the following command to find out which macOS SDK will be used

```
xcrun -sdk macosx --show-sdk-version
```

Troubleshooting

Metal not found

You see the following error when you try to build:

```
error: unable to find utility "metal", not a developer tool or in PATH
```

Skip to main content

To fix this, first make sure you have Xcode installed:

```
xcode-select --install
```

Then set the active developer directory:

```
sudo xcode-select --switch /Applications/Xcode.app/Contents/Developer
```

x86 Shell

If the ouptut of `uname -p` is `x86` then your shell is running as x86 via Rosetta instead of natively.

To fix this, find the application in Finder (`/Applications` for iTerm, `/Applications/Utilities` for Terminal), right-click, and click "Get Info". Uncheck "Open using Rosetta", close the "Get Info" window, and restart your terminal.

Verify the terminal is now running natively the following command:

```
$ uname -p
arm
```

Also check that cmake is using the correct architecture:

```
$ cmake --system-information | grep CMAKE_HOST_SYSTEM_PROCESSOR
CMAKE_HOST_SYSTEM_PROCESSOR "arm64"
```

If you see `"x86_64"`, try re-installing `cmake`. If you see `"arm64"` but the build errors out with "Building for x86_64 on macOS is not supported." wipe your build cahce with `rm -rf build/` and try again.

Previous
MLX

Next
Quick Start Guide

Print to PDF

Quick Start Guide

Contents

- Basics
- Function and Graph Transformations

Basics

Import `mlx.core` and make an **`array`**:

```
>> import mlx.core as mx
>> a = mx.array([1, 2, 3, 4])
>> a.shape
[4]
>> a.dtype
int32
>> b = mx.array([1.0, 2.0, 3.0, 4.0])
>> b.dtype
float32
```

Operations in MLX are lazy. The outputs of MLX operations are not computed until they are needed. To force an array to be evaluated use **`eval()`**. Arrays will automatically be evaluated in a few cases. For example, inspecting a scalar with **`array.item()`**, printing an array, or converting an array from **`array`** to **`numpy.ndarray`** all automatically evaluate the array.

```
>> c = a + b    # c not yet evaluated
>> mx.eval(c)  # evaluates c
>> c = a + b
>> print(c)     # Also evaluates c
array([2, 4, 6, 8], dtype=float32)
>> c = a + b
>> import numpy as np
>> np.array(c)   # Also evaluates c
array([2., 4., 6., 8.], dtype=float32)
```

See the page on Lazy Evaluation for more details.

Skip to main content

Function and Graph Transformations

MLX has standard function transformations like `grad()` and `vmap()`. Transformations can be composed arbitrarily. For example `grad(vmap(grad(fn)))` (or any other composition) is allowed.

```
>> x = mx.array(0.0)
>> mx.sin(x)
array(0, dtype=float32)
>> mx.grad(mx.sin)(x)
array(1, dtype=float32)
>> mx.grad(mx.grad(mx.sin))(x)
array(-0, dtype=float32)
```

Other gradient transformations include `vjp()` for vector-Jacobian products and `jvp()` for Jacobian-vector products.

Use `value_and_grad()` to efficiently compute both a function's output and gradient with respect to the function's input.

Previous
Build and Install

Next
Lazy Evaluation

Lazy Evaluation

Contents

- Why Lazy Evaluation
- When to Evaluate

Why Lazy Evaluation

When you perform operations in MLX, no computation actually happens. Instead a compute graph is recorded. The actual computation only happens if an `eval()` is performed.

MLX uses lazy evaluation because it has some nice features, some of which we describe below.

Transforming Compute Graphs

Lazy evaluation let's us record a compute graph without actually doing any computations. This is useful for function transformations like `grad()` and `vmap()` and graph optimizations.

Currently, MLX does not compile and rerun compute graphs. They are all generated dynamically. However, lazy evaluation makes it much easier to integrate compilation for future performance enhancements.

Only Compute What You Use

In MLX you do not need to worry as much about computing outputs that are never used. For example:

```
def fun(x):
    a = fun1(x)
```

Skip to main content

```
y, _ = fun(x)
```

Here, we never actually compute the output of `expensive_fun`. Use this pattern with care though, as the graph of `expensive_fun` is still built, and that has some cost associated to it.

Similarly, lazy evaluation can be beneficial for saving memory while keeping code simple. Say you have a very large model `Model` derived from **`mlx.nn.Module`**. You can instantiate this model with `model = Model()`. Typically, this will initialize all of the weights as `float32`, but the initialization does not actually compute anything until you perform an **`eval()`**. If you update the model with `float16` weights, your maximum consumed memory will be half that required if eager computation was used instead.

This pattern is simple to do in MLX thanks to lazy computation:

```
model = Model() # no memory used yet
model.load_weights("weights_fp16.safetensors")
```

When to Evaluate

A common question is when to use **`eval()`**. The trade-off is between letting graphs get too large and not batching enough useful work.

For example:

```
for _ in range(100):
    a = a + b
    mx.eval(a)
    b = b * 2
    mx.eval(b)
```

This is a bad idea because there is some fixed overhead with each graph evaluation. On the other hand, there is some slight overhead which grows with the compute graph size, so extremely large graphs (while computationally correct) can be costly.

Luckily, a wide range of compute graph sizes work pretty well with MLX: anything from a few tens of operations to many thousands of operations per evaluation should be okay.

Most numerical computations have an iterative outer loop (e.g. the iteration in stochastic gradient descent). A natural and usually efficient place to use `eval()` is at each iteration

Skip to main content

Here is a concrete example:

```
for batch in dataset:

    # Nothing has been evaluated yet
    loss, grad = value_and_grad_fn(model, batch)

    # Still nothing has been evaluated
    optimizer.update(model, grad)

    # Evaluate the loss and the new parameters which will
    # run the full gradient computation and optimizer update
    mx.eval(loss, model.parameters())
```

An important behavior to be aware of is when the graph will be implicitly evaluated. Anytime you `print` an array, convert it to an **`numpy.ndarray`**, or otherwise access it's memory via **`memoryview`**, the graph will be evaluated. Saving arrays via **`save()`** (or any other MLX saving functions) will also evaluate the array.

Calling **`array.item()`** on a scalar array will also evaluate it. In the example above, printing the loss (`print(loss)`) or adding the loss scalar to a list (`losses.append(loss.item())`) would cause a graph evaluation. If these lines are before `mx.eval(loss, model.parameters())` then this will be a partial evaluation, computing only the forward pass.

Also, calling **`eval()`** on an array or set of arrays multiple times is perfectly fine. This is effectively a no-op.

> **⚠ Warning**
>
> Using scalar arrays for control-flow will cause an evaluation.

Here is an example:

```
def fun(x):
    h, y = first_layer(x)
    if y > 0:  # An evaluation is done here!
        z  = second_layer_a(h)
    else:
        z  = second_layer_b(h)
    return z
```

Using arrays for control flow should be done with care. The above example works and can even be used with gradient transformations. However, this can be very inefficient if

Skip to main content

Previous
Quick Start Guide

Next
Unified Memory

Unified Memory

Contents

- A Simple Example

Apple silicon has a unified memory architecture. The CPU and GPU have direct access to the same memory pool. MLX is designed to take advantage of that.

Concretely, when you make an array in MLX you don't have to specify its location:

```
a = mx.random.normal((100,))
b = mx.random.normal((100,))
```

Both `a` and `b` live in unified memory.

In MLX, rather than moving arrays to devices, you specify the device when you run the operation. Any device can perform any operation on `a` and `b` without needing to move them from one memory location to another. For example:

```
mx.add(a, b, stream=mx.cpu)
mx.add(a, b, stream=mx.gpu)
```

In the above, both the CPU and the GPU will perform the same add operation. The operations can (and likely will) be run in parallel since there are no dependencies between them. See Using Streams for more information the semantics of streams in MLX.

In the above `add` example, there are no dependencies between operations, so there is no possibility for race conditions. If there are dependencies, the MLX scheduler will automatically manage them. For example:

```
c = mx.add(a, b, stream=mx.cpu)
d = mx.add(a, c, stream=mx.gpu)
```

In the above case, the second `add` runs on the GPU but it depends on the output of the first `add` which is running on the CPU. MLX will automatically insert a dependency

Skip to main content

between the two streams so that the second `add` only starts executing after the first is complete and `c` is available.

A Simple Example

Here is a more interesting (albeit slightly contrived example) of how unified memory can be helpful. Suppose we have the following computation:

```
def fun(a, b, d1, d2):
  x = mx.matmul(a, b, stream=d1)
  for _ in range(500):
      b = mx.exp(b, stream=d2)
  return x, b
```

which we want to run with the following arguments:

```
a = mx.random.uniform(shape=(4096, 512))
b = mx.random.uniform(shape=(512, 4))
```

The first `matmul` operation is a good fit for the GPU since it's more compute dense. The second sequence of operations are a better fit for the CPU, since they are very small and would probably be overhead bound on the GPU.

If we time the computation fully on the GPU, we get 2.8 milliseconds. But if we run the computation with `d1=mx.gpu` and `d2=mx.cpu`, then the time is only about 1.4 milliseconds, about twice as fast. These times were measured on an M1 Max.

Previous
Lazy Evaluation

Next
Indexing Arrays

Print to PDF

Indexing Arrays

Contents

- Differences from NumPy
- In Place Updates

For the most part, indexing an MLX `array` works the same as indexing a NumPy `numpy.ndarray`. See the NumPy documentation for more details on how that works.

For example, you can use regular integers and slices (`slice`) to index arrays:

```
>>> arr = mx.arange(10)
>>> arr[3]
array(3, dtype=int32)
>>> arr[-2]  # negative indexing works
array(8, dtype=int32)
>>> arr[2:8:2] # start, stop, stride
array([2, 4, 6], dtype=int32)
```

For multi-dimensional arrays, the `...` or `Ellipsis` syntax works as in NumPy:

```
>>> arr = mx.arange(8).reshape(2, 2, 2)
>>> arr[:, :, 0]
array(3, dtype=int32)
array([[0, 2],
       [4, 6]], dtype=int32
>>> arr[..., 0]
array([[0, 2],
       [4, 6]], dtype=int32
```

You can index with `None` to create a new axis:

```
>>> arr = mx.arange(8)
>>> arr.shape
[8]
>>> arr[None].shape
[1, 8]
```

You can also use an `array` to index another `array`:

Skip to main content

```
>>> arr = mx.arange(10)
>>> idx = mx.array([5, 7])
>>> arr[idx]
array([5, 7], dtype=int32)
```

Mixing and matching integers, `slice`, `...`, and `array` indices works just as in NumPy.

Other functions which may be useful for indexing arrays are `take()` and `take_along_axis()`.

Differences from NumPy

> **Note**
>
> MLX indexing is different from NumPy indexing in two important ways:
>
> - Indexing does not perform bounds checking. Indexing out of bounds is undefined behavior.
> - Boolean mask based indexing is not yet supported.

The reason for the lack of bounds checking is that exceptions cannot propagate from the GPU. Performing bounds checking for array indices before launching the kernel would be extremely inefficient.

Indexing with boolean masks is something that MLX may support in the future. In general, MLX has limited support for operations for which outputs *shapes* are dependent on input *data*. Other examples of these types of operations which MLX does not yet support include `numpy.nonzero()` and the single input version of `numpy.where()`.

In Place Updates

In place updates to indexed arrays are possible in MLX. For example:

```
>>> a = mx.array([1, 2, 3])
>>> a[2] = 0
>>> a
array([1, 2, 0], dtype=int32)
```

Just as in NumPy, in place updates will be reflected in all references to the same array:

Skip to main content

```
>>> a = mx.array([1, 2, 3])
>>> b = a
>>> b[2] = 0
>>> b
array([1, 2, 0], dtype=int32)
>>> a
array([1, 2, 0], dtype=int32)
```

Transformations of functions which use in-place updates are allowed and work as expected. For example:

```
def fun(x, idx):
    x[idx] = 2.0
    return x.sum()

dfdx = mx.grad(fun)(mx.array([1.0, 2.0, 3.0]), mx.array([1]))
print(dfdx)  # Prints: array([1, 0, 1], dtype=float32)
```

In the above `dfdx` will have the correct gradient, namely zeros at `idx` and ones elsewhere.

Previous
Unified Memory

Next
Saving and Loading Arrays

Saving and Loading Arrays

Print to PDF

MLX supports multiple array serialization formats.

Serialization Formats

Format	Extension	Function	Notes
NumPy	`.npy`	`save()`	Single arrays only
NumPy archive	`.npz`	`savez()` and `savez_compressed()`	Multiple arrays
Safetensors	`.safetensors`	`save_safetensors()`	Multiple arrays
GGUF	`.gguf`	`save_gguf()`	Multiple arrays

The `load()` function will load any of the supported serialization formats. It determines the format from the extensions. The output of `load()` depends on the format.

Here's an example of saving a single array to a file:

```
>>> a = mx.array([1.0])
>>> mx.save("array", a)
```

The array `a` will be saved in the file `array.npy` (notice the extension is automatically added). Including the extension is optional; if it is missing it will be added. You can load the array with:

```
>>> mx.load("array.npy", a)
array([1], dtype=float32)
```

Here's an example of saving several arrays to a single file:

```
>>> a = mx.array([1.0])
>>> b = mx.array([2.0])
>>> mx.savez("arrays", a, b=b)
```

Skip to main content

```
>>> mx.load("arrays.npz")
{'b': array([2], dtype=float32), 'arr_0': array([1], dtype=float32)}
```

In this case `load()` returns a dictionary of names to arrays.

The functions `save_safetensors()` and `save_gguf()` are similar to `savez()`, but they take as input a `dict` of string names to arrays:

```
>>> a = mx.array([1.0])
>>> b = mx.array([2.0])
>>> mx.save_safetensors("arrays", {"a": a, "b": b})
```

Previous
Indexing Arrays

Next
Function Transforms

Print to PDF

Function Transforms

Contents

- Automatic Differentiation
- Automatic Vectorization

MLX uses composable function transformations for automatic differentiation, vectorization, and compute graph optimizations. To see the complete list of function transformations check-out the API documentation.

The key idea behind composable function transformations is that every transformation returns a function which can be further transformed.

Here is a simple example:

```
>>> dfdx = mx.grad(mx.sin)
>>> dfdx(mx.array(mx.pi))
array(-1, dtype=float32)
>>> mx.cos(mx.array(mx.pi))
array(-1, dtype=float32)
```

The output of `grad()` on `sin()` is simply another function. In this case it is the gradient of the sine function which is exactly the cosine function. To get the second derivative you can do:

```
>>> d2fdx2 = mx.grad(mx.grad(mx.sin))
>>> d2fdx2(mx.array(mx.pi / 2))
array(-1, dtype=float32)
>>> mx.sin(mx.array(mx.pi / 2))
array(1, dtype=float32)
```

Using `grad()` on the output of `grad()` is always ok. You keep getting higher order derivatives.

Any of the MLX function transformations can be composed in any order to any depth. See the following sections for more information on automatic differentiation and automatic vectorization. For more information on `compile()` see the compile documentation.

Skip to main content

Automatic Differentiation

Automatic differentiation in MLX works on functions rather than on implicit graphs.

> **Note**
>
> If you are coming to MLX from PyTorch, you no longer need functions like `backward`, `zero_grad`, and `detach`, or properties like `requires_grad`.

The most basic example is taking the gradient of a scalar-valued function as we saw above. You can use the **`grad()`** and **`value_and_grad()`** function to compute gradients of more complex functions. By default these functions compute the gradient with respect to the first argument:

```
def loss_fn(w, x, y):
   return mx.mean(mx.square(w * x - y))

w = mx.array(1.0)
x = mx.array([0.5, -0.5])
y = mx.array([1.5, -1.5])

# Computes the gradient of loss_fn with respect to w:
grad_fn = mx.grad(loss_fn)
dloss_dw = grad_fn(w, x, y)
# Prints array(-1, dtype=float32)
print(dloss_dw)

# To get the gradient with respect to x we can do:
grad_fn = mx.grad(loss_fn, argnums=1)
dloss_dx = grad_fn(w, x, y)
# Prints array([-1, 1], dtype=float32)
print(dloss_dx)
```

One way to get the loss and gradient is to call `loss_fn` followed by `grad_fn`, but this can result in a lot of redundant work. Instead, you should use **`value_and_grad()`**. Continuing the above example:

```
# Computes the gradient of loss_fn with respect to w:
loss_and_grad_fn = mx.value_and_grad(loss_fn)
loss, dloss_dw = loss_and_grad_fn(w, x, y)

# Prints array(1, dtype=float32)
print(loss)

# Prints array(-1, dtype=float32)
print(dloss_dw)
```

Skip to main content

You can also take the gradient with respect to arbitrarily nested Python containers of arrays (specifically any of `list`, `tuple`, or `dict`).

Suppose we wanted a weight and a bias parameter in the above example. A nice way to do that is the following:

```
def loss_fn(params, x, y):
   w, b = params["weight"], params["bias"]
   h = w * x + b
   return mx.mean(mx.square(h - y))

params = {"weight": mx.array(1.0), "bias": mx.array(0.0)}
x = mx.array([0.5, -0.5])
y = mx.array([1.5, -1.5])

# Computes the gradient of loss_fn with respect to both the
# weight and bias:
grad_fn = mx.grad(loss_fn)
grads = grad_fn(params, x, y)

# Prints
# {'weight': array(-1, dtype=float32), 'bias': array(0, dtype=float32)}
print(grads)
```

Notice the tree structure of the parameters is preserved in the gradients.

In some cases you may want to stop gradients from propagating through a part of the function. You can use the `stop_gradient()` for that.

Automatic Vectorization

Use `vmap()` to automate vectorizing complex functions. Here we'll go through a basic and contrived example for the sake of clarity, but `vmap()` can be quite powerful for more complex functions which are difficult to optimize by hand.

> **Warning**
>
> Some operations are not yet supported with `vmap()`. If you encounter an error like: `ValueError: Primitive's vmap not implemented.` file an issue and include your function. We will prioritize including it.

A naive way to add the elements from two sets of vectors is with a loop:

Skip to main content

```
def naive_add(xs, ys):
    return [xs[i] + ys[:, i] for i in range(xs.shape[1])]
```

Instead you can use `vmap()` to automatically vectorize the addition:

```
# Vectorize over the second dimension of x and the
# first dimension of y
vmap_add = mx.vmap(lambda x, y: x + y, in_axes=(1, 0))
```

The `in_axes` parameter can be used to specify which dimensions of the corresponding input to vectorize over. Similarly, use `out_axes` to specify where the vectorized axes should be in the outputs.

Let's time these two different versions:

```
import timeit

print(timeit.timeit(lambda: mx.eval(naive_add(xs, ys)), number=100))
print(timeit.timeit(lambda: mx.eval(vmap_add(xs, ys)), number=100))
```

On an M1 Max the naive version takes in total `0.390` seconds whereas the vectorized version takes only `0.025` seconds, more than ten times faster.

Of course, this operation is quite contrived. A better approach is to simply do `xs + ys.T`, but for more complex functions `vmap()` can be quite handy.

Previous
Saving and Loading Arrays

Next
Compilation

Print to PDF

Compilation

Contents

- Basics of Compile
- Example Speedup
- Debugging
- Pure Functions
- Compiling Training Graphs
- Transformations with Compile

MLX has a `compile()` function transformation which compiles computation graphs. Function compilation results in smaller graphs by merging common work and fusing certain operations. In many cases this can lead to big improvements in run-time and memory use.

Getting started with `compile()` is simple, but there are some edge cases that are good to be aware of for more complex graphs and advanced usage.

Basics of Compile

Let's start with a simple example:

```
def fun(x, y):
    return mx.exp(-x) + y

x = mx.array(1.0)
y = mx.array(2.0)

# Regular call, no compilation
# Prints: array(2.36788, dtype=float32)
print(fun(x, y))

# Compile the function
compiled_fun = mx.compile(fun)

# Prints: array(2.36788, dtype=float32)
print(compiled_fun(x, y))
```

Skip to main content

The output of both the regular function and the compiled function is the same up to numerical precision.

The first time you call a compiled function, MLX will build the compute graph, optimize it, and generate and compile code. This can be relatively slow. However, MLX will cache compiled functions, so calling a compiled function multiple times will not initiate a new compilation. This means you should typically compile functions that you plan to use more than once.

```
def fun(x, y):
    return mx.exp(-x) + y

x = mx.array(1.0)
y = mx.array(2.0)

compiled_fun = mx.compile(fun)

# Compiled here
compiled_fun(x, y)

# Not compiled again
compiled_fun(x, y)

# Not compiled again
mx.compile(fun)(x, y)
```

There are some important cases to be aware of that can cause a function to be recompiled:

- Changing the shape or number of dimensions
- Changing the type of any of the inputs
- Changing the number of inputs to the function

In certain cases only some of the compilation stack will be rerun (for example when changing the shapes) and in other cases the full compilation stack will be rerun (for example when changing the types). In general you should avoid compiling functions too frequently.

Another idiom to watch out for is compiling functions which get created and destroyed frequently. This can happen, for example, when compiling an anonymous function in a loop:

```
a = mx.array(1.0)
# Don't do this, compiles lambda at each iteration
for _ in range(5):
    mx.compile(lambda x: mx.exp(mx.abs(x)))(a)
```

Skip to main content

Example Speedup

The `mlx.nn.gelu()` is a nonlinear activation function commonly used with Transformer-based models. The implementation involves several unary and binary element-wise operations:

```
def gelu(x):
    return x * (1 + mx.erf(x / math.sqrt(2))) / 2
```

If you use this function with small arrays, it will be overhead bound. If you use it with large arrays it will be memory bandwidth bound. However, all of the operations in the `gelu` are fusible into a single kernel with `compile()`. This can speedup both cases considerably.

Let's compare the runtime of the regular function versus the compiled function. We'll use the following timing helper which does a warm up and handles synchronization:

```
import time

def timeit(fun, x):
    # warm up
    for _ in range(10):
        mx.eval(fun(x))

    tic = time.perf_counter()
    for _ in range(100):
        mx.eval(fun(x))
    toc = time.perf_counter()
    tpi = 1e3 * (toc - tic) / 100
    print(f"Time per iteration {tpi:.3f} (ms)")
```

Now make an array, and benchmark both functions:

```
x = mx.random.uniform(shape=(32, 1000, 4096))
timeit(nn.gelu, x)
timeit(mx.compile(nn.gelu), x)
```

On an M1 Max the times are 15.5 and 3.1 milliseconds. The compiled `gelu` is five times faster.

Note

As of the latest MLX, CPU functions are not fully compiled. Compiling CPU functions can still be helpful, but won't typically result in as large a speedup as

Skip to main content

Debugging

When a compiled function is first called, it is traced with placeholder inputs. This means you can't evaluate arrays (for example to print their contents) inside compiled functions.

```
@mx.compile
def fun(x):
    z = -x
    print(z)  # Crash
    return mx.exp(z)

fun(mx.array(5.0))
```

For debugging, inspecting arrays can be helpful. One way to do that is to globally disable compilation using the `disable_compile()` function or `MLX_DISABLE_COMPILE` flag. For example the following is okay even though `fun` is compiled:

```
@mx.compile
def fun(x):
    z = -x
    print(z) # Okay
    return mx.exp(z)

mx.disable_compile()
fun(mx.array(5.0))
```

Pure Functions

Compiled functions are intended to be *pure*; that is they should not have side effects. For example:

```
state = []

@mx.compile
def fun(x, y):
    z = x + y
    state.append(z)
    return mx.exp(z)

fun(mx.array(1.0), mx.array(2.0))
# Crash!
print(state)
```

After the first call of `fun`, the `state` list will hold a placeholder array. The placeholder

Skip to main content

array results in a crash.

You have two options to deal with this. The first option is to simply return `state` as an output:

```
state = []

@mx.compile
def fun(x, y):
   z = x + y
   state.append(z)
   return mx.exp(z), state

 _, state = fun(mx.array(1.0), mx.array(2.0))
 # Prints [array(3, dtype=float32)]
 print(state)
```

In some cases returning updated state can be pretty inconvenient. Hence, **`compile()`** has a parameter to capture implicit outputs:

```
from functools import partial

state = []

# Tell compile to capture state as an output
@partial(mx.compile, outputs=state)
def fun(x, y):
    z = x + y
    state.append(z)
    return mx.exp(z), state

fun(mx.array(1.0), mx.array(2.0))
# Prints [array(3, dtype=float32)]
print(state)
```

This is particularly useful for compiling a function which includes an update to a container of arrays, as is commonly done when training the parameters of a **`mlx.nn.Module`**.

Compiled functions will also treat any inputs not in the parameter list as constants. For example:

```
state = [mx.array(1.0)]

@mx.compile
def fun(x):
    return x + state[0]

# Prints array(2, dtype=float32)
print(fun(mx.array(1.0)))
```

Skip to main content

```
# Update state
state[0] = mx.array(5.0)

# Still prints array(2, dtype=float32)
print(fun(mx.array(1.0)))
```

In order to have the change of state reflected in the outputs of `fun` you again have two options. The first option is to simply pass `state` as input to the function. In some cases this can be pretty inconvenient. Hence, **`compile()`** also has a parameter to capture implicit inputs:

```
from functools import partial
state = [mx.array(1.0)]

# Tell compile to capture state as an input
@partial(mx.compile, inputs=state)
def fun(x):
    return x + state[0]

# Prints array(2, dtype=float32)
print(fun(mx.array(1.0)))

# Update state
state[0] = mx.array(5.0)

# Prints array(6, dtype=float32)
print(fun(mx.array(1.0)))
```

Compiling Training Graphs

This section will step through how to use **`compile()`** with a simple example of a common setup: training a model with **`mlx.nn.Module`** using an **`mlx.optimizers.Optimizer`** with state. We will show how to compile the full forward, backward, and update with **`compile()`**.

To start, here is the simple example without any compilation:

```
import mlx.core as mx
import mlx.nn as nn
import mlx.optimizers as optim

# 4 examples with 10 features each
x = mx.random.uniform(shape=(4, 10))

# 0, 1 targets
y = mx.array([0, 1, 0, 1])
```

Skip to main content

```
# SGD with momentum
optimizer = optim.SGD(learning_rate=0.1, momentum=0.8)

def loss_fn(model, x, y):
    logits = model(x).squeeze()
    return nn.losses.binary_cross_entropy(logits, y)

loss_and_grad_fn = nn.value_and_grad(model, loss_fn)

# Perform 10 steps of gradient descent
for it in range(10):
    loss, grads = loss_and_grad_fn(model, x, y)
    optimizer.update(model, grads)
    mx.eval(model.parameters(), optimizer.state)
```

To compile the update we can put it all in a function and compile it with the appropriate input and output captures. Here's the same example but compiled:

```
import mlx.core as mx
import mlx.nn as nn
import mlx.optimizers as optim
from functools import partial

# 4 examples with 10 features each
x = mx.random.uniform(shape=(4, 10))

# 0, 1 targets
y = mx.array([0, 1, 0, 1])

# Simple linear model
model = nn.Linear(10, 1)

# SGD with momentum
optimizer = optim.SGD(learning_rate=0.1, momentum=0.8)

def loss_fn(model, x, y):
    logits = model(x).squeeze()
    return nn.losses.binary_cross_entropy(logits, y)

# The state that will be captured as input and output
state = [model.state, optimizer.state]

@partial(mx.compile, inputs=state, outputs=state)
def step(x, y):
    loss_and_grad_fn = nn.value_and_grad(model, loss_fn)
    loss, grads = loss_and_grad_fn(model, x, y)
    optimizer.update(model, grads)
    return loss

# Perform 10 steps of gradient descent
for it in range(10):
    loss = step(x, y)
    # Evaluate the model and optimizer state
    mx.eval(state)
```

Skip to main content

> **Note**
>
> If you are using a module which performs random sampling such as `mlx.nn.Dropout()`, make sure you also include `mx.random.state` in the `state` captured by `compile()`, i.e. `state = [model.state, optimizer.state, mx.random.state]`.

> **Note**
>
> For more examples of compiling full training graphs checkout the MLX Examples GitHub repo.

Transformations with Compile

In MLX function transformations are composable. You can apply any function transformation to the output of any other function transformation. For more on this, see the documentation on function transforms.

Compiling transformed functions works just as expected:

```
grad_fn = mx.grad(mx.exp)

compiled_grad_fn = mx.compile(grad_fn)

# Prints: array(2.71828, dtype=float32)
print(grad_fn(mx.array(1.0)))

# Also prints: array(2.71828, dtype=float32)
print(compiled_grad_fn(mx.array(1.0)))
```

> **Note**
>
> In order to compile as much as possible, a transformation of a compiled function will not by default be compiled. To compile the transformed function simply pass it through `compile()`.

You can also compile functions which themselves call compiled functions. A good practice is to compile the outer most function to give `compile()` the most opportunity to optimize the computation graph:

Skip to main content

```
@mx.compile
def inner(x):
    return mx.exp(-mx.abs(x))

def outer(x):
    inner(inner(x))

# Compiling the outer function is good to do as it will likely
# be faster even though the inner functions are compiled
fun = mx.compile(outer)
```

Previous
Function Transforms

Next
Conversion to NumPy and Other Frameworks

Conversion to NumPy and Other Frameworks

Contents

- PyTorch
- JAX
- TensorFlow

MLX array implements the Python Buffer Protocol. Let's convert an array to NumPy and back.

```
import mlx.core as mx
import numpy as np

a = mx.arange(3)
b = np.array(a) # copy of a
c = mx.array(b) # copy of b
```

Note

Since NumPy does not support `bfloat16` arrays, you will need to convert to `float16` or `float32` first: `np.array(a.astype(mx.float32))`. Otherwise, you will receive an error like: `Item size 2 for PEP 3118 buffer format string does not match the dtype V item size 0.`

By default, NumPy copies data to a new array. This can be prevented by creating an array view:

```
a = mx.arange(3)
a_view = np.array(a, copy=False)
print(a_view.flags.owndata) # False
a_view[0] = 1
print(a[0].item()) # 1
```

Skip to main content

While this is quite powerful to prevent copying arrays, it should be noted that external changes to the memory of arrays cannot be reflected in gradients.

Let's demonstrate this in an example:

```
def f(x):
    x_view = np.array(x, copy=False)
    x_view[:] *= x_view # modify memory without telling mx
    return x.sum()

x = mx.array([3.0])
y, df = mx.value_and_grad(f)(x)
print("f(x) = x² =", y.item()) # 9.0
print("f'(x) = 2x !=", df.item()) # 1.0
```

The function `f` indirectly modifies the array `x` through a memory view. However, this modification is not reflected in the gradient, as seen in the last line outputting `1.0`, representing the gradient of the sum operation alone. The squaring of `x` occurs externally to MLX, meaning that no gradient is incorporated. It's important to note that a similar issue arises during array conversion and copying. For instance, a function defined as `mx.array(np.array(x)**2).sum()` would also result in an incorrect gradient, even though no in-place operations on MLX memory are executed.

PyTorch

> **Warning**
>
> PyTorch Support for **`memoryview`** is experimental and can break for multi-dimensional arrays. Casting to NumPy first is advised for now.

PyTorch supports the buffer protocol, but it requires an explicit **`memoryview`**.

```
import mlx.core as mx
import torch

a = mx.arange(3)
b = torch.tensor(memoryview(a))
c = mx.array(b.numpy())
```

Conversion from PyTorch tensors back to arrays must be done via intermediate NumPy arrays with `numpy()`.

Skip to main content

JAX

JAX fully supports the buffer protocol.

```
import mlx.core as mx
import jax.numpy as jnp

a = mx.arange(3)
b = jnp.array(a)
c = mx.array(b)
```

TensorFlow

TensorFlow supports the buffer protocol, but it requires an explicit `memoryview`.

```
import mlx.core as mx
import tensorflow as tf

a = mx.arange(3)
b = tf.constant(memoryview(a))
c = mx.array(b)
```

Previous
Compilation

Next
Using Streams

Using Streams

Contents

- Specifying the `Stream`

Specifying the `Stream`

All operations (including random number generation) take an optional keyword argument `stream`. The `stream` kwarg specifies which `Stream` the operation should run on. If the stream is unspecified then the operation is run on the default stream of the default device: `mx.default_stream(mx.default_device())`. The `stream` kwarg can also be a `Device` (e.g. `stream=my_device`) in which case the operation is run on the default stream of the provided device `mx.default_stream(my_device)`.

Previous
Conversion to NumPy and Other Frameworks

Next
Linear Regression

Print to PDF

Linear Regression

Let's implement a basic linear regression model as a starting point to learn MLX. First import the core package and setup some problem metadata:

```
import mlx.core as mx

num_features = 100
num_examples = 1_000
num_iters = 10_000  # iterations of SGD
lr = 0.01  # learning rate for SGD
```

We'll generate a synthetic dataset by:

1. Sampling the design matrix `X`.
2. Sampling a ground truth parameter vector `w_star`.
3. Compute the dependent values `y` by adding Gaussian noise to `X @ w_star`.

```
# True parameters
w_star = mx.random.normal((num_features,))

# Input examples (design matrix)
X = mx.random.normal((num_examples, num_features))

# Noisy labels
eps = 1e-2 * mx.random.normal((num_examples,))
y = X @ w_star + eps
```

We will use SGD to find the optimal weights. To start, define the squared loss and get the gradient function of the loss with respect to the parameters.

```
def loss_fn(w):
    return 0.5 * mx.mean(mx.square(X @ w - y))

grad_fn = mx.grad(loss_fn)
```

Start the optimization by initializing the parameters `w` randomly. Then repeatedly update the parameters for `num_iters` iterations.

```
w = 1e-2 * mx.random.normal((num_features,))
```

Skip to main content

```
    grad = grad_fn(w)
    w = w - lr * grad
    mx.eval(w)
```

Finally, compute the loss of the learned parameters and verify that they are close to the ground truth parameters.

```
loss = loss_fn(w)
error_norm = mx.sum(mx.square(w - w_star)).item() ** 0.5

print(
    f"Loss {loss.item():.5f}, |w-w*| = {error_norm:.5f}, "
)
# Should print something close to: Loss 0.00005, |w-w*| = 0.00364
```

Complete linear regression and logistic regression examples are available in the MLX GitHub repo.

Previous
Using Streams

Next
Multi-Layer Perceptron

Print to PDF

Multi-Layer Perceptron

In this example we'll learn to use `mlx.nn` by implementing a simple multi-layer perceptron to classify MNIST.

As a first step import the MLX packages we need:

```
import mlx.core as mx
import mlx.nn as nn
import mlx.optimizers as optim

import numpy as np
```

The model is defined as the `MLP` class which inherits from **`mlx.nn.Module`**. We follow the standard idiom to make a new module:

1. Define an `__init__` where the parameters and/or submodules are setup. See the Module class docs for more information on how **`mlx.nn.Module`** registers parameters.
2. Define a `__call__` where the computation is implemented.

```
class MLP(nn.Module):
    def __init__(
        self, num_layers: int, input_dim: int, hidden_dim: int, output_dim: i
    ):
        super().__init__()
        layer_sizes = [input_dim] + [hidden_dim] * num_layers + [output_dim]
        self.layers = [
            nn.Linear(idim, odim)
            for idim, odim in zip(layer_sizes[:-1], layer_sizes[1:])
        ]

    def __call__(self, x):
        for l in self.layers[:-1]:
            x = mx.maximum(l(x), 0.0)
        return self.layers[-1](x)
```

We define the loss function which takes the mean of the per-example cross entropy loss. The `mlx.nn.losses` sub-package has implementations of some commonly used loss functions.

```
def loss_fn(model, X, y):
```

Skip to main content

```
    return mx.mean(nn.losses.cross_entropy(model(X), y))
```

We also need a function to compute the accuracy of the model on the validation set:

```
def eval_fn(model, X, y):
    return mx.mean(mx.argmax(model(X), axis=1) == y)
```

Next, setup the problem parameters and load the data. To load the data, you need our mnist data loader, which we will import as *mnist*.

```
num_layers = 2
hidden_dim = 32
num_classes = 10
batch_size = 256
num_epochs = 10
learning_rate = 1e-1

# Load the data
import mnist
train_images, train_labels, test_images, test_labels = map(
    mx.array, mnist.mnist()
)
```

Since we're using SGD, we need an iterator which shuffles and constructs minibatches of examples in the training set:

```
def batch_iterate(batch_size, X, y):
    perm = mx.array(np.random.permutation(y.size))
    for s in range(0, y.size, batch_size):
        ids = perm[s : s + batch_size]
        yield X[ids], y[ids]
```

Finally, we put it all together by instantiating the model, the **mlx.optimizers.SGD** optimizer, and running the training loop:

```
# Load the model
model = MLP(num_layers, train_images.shape[-1], hidden_dim, num_classes)
mx.eval(model.parameters())

# Get a function which gives the loss and gradient of the
# loss with respect to the model's trainable parameters
loss_and_grad_fn = nn.value_and_grad(model, loss_fn)

# Instantiate the optimizer
optimizer = optim.SGD(learning_rate=learning_rate)

for e in range(num_epochs):
```

Skip to main content

```
        # Update the optimizer state and model parameters
        # in a single call
        optimizer.update(model, grads)

        # Force a graph evaluation
        mx.eval(model.parameters(), optimizer.state)

    accuracy = eval_fn(model, test_images, test_labels)
    print(f"Epoch {e}: Test accuracy {accuracy.item():.3f}")
```

Note

The `mlx.nn.value_and_grad()` function is a convenience function to get the gradient of a loss with respect to the trainable parameters of a model. This should not be confused with `mlx.core.value_and_grad()`.

The model should train to a decent accuracy (about 95%) after just a few passes over the training set. The full example is available in the MLX GitHub repo.

Previous
Linear Regression

Next
LLM inference

LLM inference

Contents

- Implementing the model
- Converting the weights
- Weight loading and benchmarking
- Scripts

MLX enables efficient inference of large-ish transformers on Apple silicon without compromising on ease of use. In this example we will create an inference script for the Llama family of transformer models in which the model is defined in less than 200 lines of python.

Implementing the model

We will use the neural network building blocks defined in the `mlx.nn` module to concisely define the model architecture.

Attention layer

We will start with the llama attention layer which notably uses the RoPE positional encoding. [1] In addition, our attention layer will optionally use a key/value cache that will be concatenated with the provided keys and values to support efficient inference.

Our implementation uses `mlx.nn.Linear` for all the projections and `mlx.nn.RoPE` for the positional encoding.

```
import mlx.core as mx
import mlx.nn as nn

class LlamaAttention(nn.Module):
    def __init__(self, dims: int, num_heads: int):
        super().__init__()
```

Skip to main content

```
        self.rope = nn.RoPE(dims // num_heads, traditional=True)
        self.query_proj = nn.Linear(dims, dims, bias=False)
        self.key_proj = nn.Linear(dims, dims, bias=False)
        self.value_proj = nn.Linear(dims, dims, bias=False)
        self.out_proj = nn.Linear(dims, dims, bias=False)

    def __call__(self, queries, keys, values, mask=None, cache=None):
        queries = self.query_proj(queries)
        keys = self.key_proj(keys)
        values = self.value_proj(values)

        # Extract some shapes
        num_heads = self.num_heads
        B, L, D = queries.shape

        # Prepare the queries, keys and values for the attention computation
        queries = queries.reshape(B, L, num_heads, -1).transpose(0, 2, 1, 3)
        keys = keys.reshape(B, L, num_heads, -1).transpose(0, 2, 1, 3)
        values = values.reshape(B, L, num_heads, -1).transpose(0, 2, 1, 3)

        # Add RoPE to the queries and keys and combine them with the cache
        if cache is not None:
            key_cache, value_cache = cache
            queries = self.rope(queries, offset=key_cache.shape[2])
            keys = self.rope(keys, offset=key_cache.shape[2])
            keys = mx.concatenate([key_cache, keys], axis=2)
            values = mx.concatenate([value_cache, values], axis=2)
        else:
            queries = self.rope(queries)
            keys = self.rope(keys)

        # Finally perform the attention computation
        scale = math.sqrt(1 / queries.shape[-1])
        scores = (queries * scale) @ keys.transpose(0, 1, 3, 2)
        if mask is not None:
            scores = scores + mask
        scores = mx.softmax(scores, axis=-1)
        values_hat = (scores @ values).transpose(0, 2, 1, 3).reshape(B, L, -

        # Note that we return the keys and values to possibly be used as a c
        return self.out_proj(values_hat), (keys, values)
```

Encoder layer

The other component of the Llama model is the encoder layer which uses RMS normalization [2] and SwiGLU. [3] For RMS normalization we will use `mlx.nn.RMSNorm` that is already provided in `mlx.nn`.

```
class LlamaEncoderLayer(nn.Module):
    def __init__(self, dims: int, mlp_dims: int, num_heads: int):
```

Skip to main content

```
        self.attention = LlamaAttention(dims, num_heads)

        self.norm1 = nn.RMSNorm(dims)
        self.norm2 = nn.RMSNorm(dims)

        self.linear1 = nn.Linear(dims, mlp_dims, bias=False)
        self.linear2 = nn.Linear(dims, mlp_dims, bias=False)
        self.linear3 = nn.Linear(mlp_dims, dims, bias=False)

    def __call__(self, x, mask=None, cache=None):
        y = self.norm1(x)
        y, cache = self.attention(y, y, y, mask, cache)
        x = x + y

        y = self.norm2(x)
        a = self.linear1(y)
        b = self.linear2(y)
        y = a * mx.sigmoid(a) * b
        y = self.linear3(y)
        x = x + y

        return x, cache
```

Full model

To implement any Llama model we simply have to combine `LlamaEncoderLayer` instances with an **`mlx.nn.Embedding`** to embed the input tokens.

```
class Llama(nn.Module):
    def __init__(
        self, num_layers: int, vocab_size: int, dims: int, mlp_dims: int, nu
    ):
        super().__init__()

        self.embedding = nn.Embedding(vocab_size, dims)
        self.layers = [
            LlamaEncoderLayer(dims, mlp_dims, num_heads) for _ in range(num_
        ]
        self.norm = nn.RMSNorm(dims)
        self.out_proj = nn.Linear(dims, vocab_size, bias=False)

    def __call__(self, x):
        mask = nn.MultiHeadAttention.create_additive_causal_mask(x.shape[1])
        mask = mask.astype(self.embedding.weight.dtype)

        x = self.embedding(x)
        for l in self.layers:
            x, _ = l(x, mask)
        x = self.norm(x)
        return self.out_proj(x)
```

Skip to main content

Note that in the implementation above we use a simple list to hold the encoder layers but using `model.parameters()` will still consider these layers.

Generation

Our `Llama` module can be used for training but not inference as the `__call__` method above processes one input, completely ignores the cache and performs no sampling whatsoever. In the rest of this subsection, we will implement the inference function as a python generator that processes the prompt and then autoregressively yields tokens one at a time.

```
class Llama(nn.Module):
    ...

    def generate(self, x, temp=1.0):
        cache = []

        # Make an additive causal mask. We will need that to process the pro
        mask = nn.MultiHeadAttention.create_additive_causal_mask(x.shape[1])
        mask = mask.astype(self.embedding.weight.dtype)

        # First we process the prompt x the same way as in __call__ but
        # save the caches in cache
        x = self.embedding(x)
        for l in self.layers:
            x, c = l(x, mask=mask)
            cache.append(c)  # <--- we store the per layer cache in a
                             #      simple python list
        x = self.norm(x)
        y = self.out_proj(x[:, -1])  # <--- we only care about the last logi
                                     #      that generate the next token
        y = mx.random.categorical(y * (1/temp))

        # y now has size [1]
        # Since MLX is lazily evaluated nothing is computed yet.
        # Calling y.item() would force the computation to happen at
        # this point but we can also choose not to do that and let the
        # user choose when to start the computation.
        yield y

        # Now we parsed the prompt and generated the first token we
        # need to feed it back into the model and loop to generate the
        # rest.
        while True:
            # Unsqueezing the last dimension to add a sequence length
            # dimension of 1
            x = y[:, None]

            x = self.embedding(x)
            for i in range(len(cache)):
                # We are overwriting the arrays in the cache list. When
```

Skip to main content

```
            # old cache the moment it is not needed anymore.
            x, cache[i] = self.layers[i](x, mask=None, cache=cache[i])
        x = self.norm(x)
        y = self.out_proj(x[:, -1])
        y = mx.random.categorical(y * (1/temp))

        yield y
```

Putting it all together

We now have everything we need to create a Llama model and sample tokens from it. In the following code, we randomly initialize a small Llama model, process 6 tokens of prompt and generate 10 tokens.

```
model = Llama(num_layers=12, vocab_size=8192, dims=512, mlp_dims=1024, num_h

# Since MLX is lazily evaluated nothing has actually been materialized yet.
# We could have set the `dims` to 20_000 on a machine with 8GB of RAM and th
# code above would still run. Let's actually materialize the model.
mx.eval(model.parameters())

prompt = mx.array([[1, 10, 8, 32, 44, 7]])  # <-- Note the double brackets b
                                            #     have a batch dimension eve
                                            #     though it is 1 in this cas

generated = [t for i, t in zip(range(10), model.generate(prompt, 0.8))]

# Since we haven't evaluated anything, nothing is computed yet. The list
# `generated` contains the arrays that hold the computation graph for the
# full processing of the prompt and the generation of 10 tokens.
#
# We can evaluate them one at a time, or all together. Concatenate them or
# print them. They would all result in very similar runtimes and give exactl
# the same results.
mx.eval(generated)
```

Converting the weights

This section assumes that you have access to the original Llama weights and the SentencePiece model that comes with them. We will write a small script to convert the PyTorch weights to MLX compatible ones and write them in a NPZ file that can be loaded directly by MLX.

Skip to main content

```
import argparse
from itertools import starmap

import numpy as np
import torch

def map_torch_to_mlx(key, value):
    if "tok_embedding" in key:
        key = "embedding.weight"

    elif "norm" in key:
        key = key.replace("attention_norm", "norm1").replace("ffn_norm", "no

    elif "wq" in key or "wk" in key or "wv" in key or "wo" in key:
        key = key.replace("wq", "query_proj")
        key = key.replace("wk", "key_proj")
        key = key.replace("wv", "value_proj")
        key = key.replace("wo", "out_proj")

    elif "w1" in key or "w2" in key or "w3" in key:
        # The FFN is a separate submodule in PyTorch
        key = key.replace("feed_forward.w1", "linear1")
        key = key.replace("feed_forward.w3", "linear2")
        key = key.replace("feed_forward.w2", "linear3")

    elif "output" in key:
        key = key.replace("output", "out_proj")

    elif "rope" in key:
        return None, None

    return key, value.numpy()


if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Convert Llama weights to M
    parser.add_argument("torch_weights")
    parser.add_argument("output_file")
    args = parser.parse_args()

    state = torch.load(args.torch_weights)
    np.savez(
        args.output_file,
        **{k: v for k, v in starmap(map_torch_to_mlx, state.items()) if k is
    )
```

Weight loading and benchmarking

After converting the weights to be compatible to our implementation, all that is left is to load them from disk and we can finally use the LLM to generate text. We can load numpy

Skip to main content

To create a parameter dictionary from the key/value representation of NPZ files we will use the `mlx.utils.tree_unflatten()` helper method as follows:

```
from mlx.utils import tree_unflatten

model.update(tree_unflatten(list(mx.load(weight_file).items())))
```

`mlx.utils.tree_unflatten()` will take keys from the NPZ file that look like `layers.2.attention.query_proj.weight` and will transform them to

```
{"layers": [..., ..., {"attention": {"query_proj": {"weight": ...}}}]}
```

which can then be used to update the model. Note that the method above incurs several unnecessary copies from disk to numpy and then from numpy to MLX. It will be replaced in the future with direct loading to MLX.

You can download the full example code in mlx-examples. Assuming, the existence of `weights.pth` and `tokenizer.model` in the current working directory we can play around with our inference script as follows (the timings are representative of an M1 Ultra and the 7B parameter Llama model):

```
$ python convert.py weights.pth llama-7B.mlx.npz
$ python llama.py llama-7B.mlx.npz tokenizer.model 'Call me Ishmael. Some ye
[INFO] Loading model from disk: 5.247 s
Press enter to start generation
------
, having little or no money in my purse, and nothing of greater consequence
------
[INFO] Prompt processing: 0.437 s
[INFO] Full generation: 4.330 s
```

We observe that 4.3 seconds are required to generate 100 tokens and 0.4 seconds of those are spent processing the prompt. This amounts to a little over **39 ms per token**.

By running with a much bigger prompt we can see that the per token generation time as well as the prompt processing time remains almost constant.

```
$ python llama.py llama-7B.mlx.npz tokenizer.model 'Call me Ishmael. Some ye
[INFO] Loading model from disk: 5.247 s
Press enter to start generation
------
```

Skip to main content

```
[INFO] Prompt processing: 0.579 s
[INFO] Full generation: 4.690 s
$ python llama.py --num-tokens 500 llama-7B.mlx.npz tokenizer.model 'Call me
[INFO] Loading model from disk: 5.628 s
Press enter to start generation
------
take his eyes from the ground. “What is it you are waiting for?” said I. “I
------
[INFO] Prompt processing: 0.633 s
[INFO] Full generation: 21.475 s
```

Scripts

Download the code

The full example code is available in mlx-examples.

[1] Su, J., Lu, Y., Pan, S., Murtadha, A., Wen, B. and Liu, Y., 2021. Roformer: Enhanced transformer with rotary position embedding. arXiv preprint arXiv:2104.09864.

[2] Zhang, B. and Sennrich, R., 2019. Root mean square layer normalization. Advances in Neural Information Processing Systems, 32.

[3] Shazeer, N., 2020. Glu variants improve transformer. arXiv preprint arXiv:2002.05202.

Previous
Multi-Layer Perceptron

Next
Array

Print to PDF

Array

Skip to main content

`array`	An N-dimensional array object.
`array.astype`(self, dtype[, stream])	Cast the array to a specified type.
`array.item`(self)	Access the value of a scalar array.
`array.tolist`(self)	Convert the array to a Python `list`.
`array.dtype`	The array's `Dtype`.
`array.ndim`	The array's dimension.
`array.shape`	The shape of the array as a Python tuple.
`array.size`	Number of elements in the array.
`Dtype`	An object to hold the type of a `array`.
`array.abs`(self, *[, stream])	See `abs()`.
`array.all`(self[, axis, keepdims, stream])	See `all()`.
`array.any`(self[, axis, keepdims, stream])	See `any()`.
`array.argmax`(self[, axis, keepdims, stream])	See `argmax()`.
`array.argmin`(self[, axis, keepdims, stream])	See `argmin()`.
`array.cos`(self, *[, stream])	See `cos()`.
`array.dtype`	The array's `Dtype`.
`array.exp`(self, *[, stream])	See `exp()`.
`array.log`(self, *[, stream])	See `log()`.
`array.log1p`(self, *[, stream])	See `log1p()`.
`array.logsumexp`(self[, axis, keepdims, stream])	See `logsumexp()`.
`array.max`(self[, axis, keepdims, stream])	See `max()`.
`array.mean`(self[, axis, keepdims, stream])	See `mean()`.

Skip to main content

`array.prod`(self[, axis, keepdims, stream])	See `prod()`.
`array.reciprocal`(self, *[, stream])	See `reciprocal()`.
`array.reshape`(self, *args[, stream])	Equivalent to `reshape()` but the shape can be passed either as a tuple or as separate arguments.
`array.round`(self, /[, decimals, stream])	See `round()`.
`array.rsqrt`(self, *[, stream])	See `rsqrt()`.
`array.sin`(self, *[, stream])	See `sin()`.
`array.split`(self, indices_or_sections[, ...])	See `split()`.
`array.sqrt`(self, *[, stream])	See `sqrt()`.
`array.square`(self, *[, stream])	See `square()`.
`array.sum`(self[, axis, keepdims, stream])	See `sum()`.
`array.transpose`(self, *args[, stream])	Equivalent to `transpose()` but the axes can be passed either as a tuple or as separate arguments.
`array.T`	Equivalent to calling `self.transpose()` with no arguments.
`array.var`(self[, axis, keepdims, ddof, stream])	See `var()`.

Previous
LLM inference

Next
mlx.core.array

mlx.core.array

class mlx.core.**array**

An N-dimensional array object.

__init__(*self: array*, *val: Union[scalar, list, tuple, ndarray, array]*, *dtype: Optional[Dtype]* = *None*)

Methods

Skip to main content

`__init__`(self, val[, dtype])	
`abs`(self, *[, stream])	See `abs()`.
`all`(self[, axis, keepdims, stream])	See `all()`.
`any`(self[, axis, keepdims, stream])	See `any()`.
`argmax`(self[, axis, keepdims, stream])	See `argmax()`.
`argmin`(self[, axis, keepdims, stream])	See `argmin()`.
`astype`(self, dtype[, stream])	Cast the array to a specified type.
`cos`(self, *[, stream])	See `cos()`.
`cummax`(self[, axis, reverse, inclusive, stream])	See `cummax()`.
`cummin`(self[, axis, reverse, inclusive, stream])	See `cummin()`.
`cumprod`(self[, axis, reverse, inclusive, stream])	See `cumprod()`.
`cumsum`(self[, axis, reverse, inclusive, stream])	See `cumsum()`.
`diag`(self[, k, stream])	Extract a diagonal or construct a diagonal matrix.
`diagonal`(self[, offset, axis1, axis2, stream])	See `diagonal()`.
`exp`(self, *[, stream])	See `exp()`.
`flatten`(self[, start_axis, end_axis, stream])	See `flatten()`.
`item`(self)	Access the value of a scalar array.
`log`(self, *[, stream])	See `log()`.
`log10`(self, *[, stream])	See `log10()`.
`log1p`(self, *[, stream])	See `log1p()`.
`log2`(self, *[, stream])	See `log2()`.

Skip to main content

`max`(self[, axis, keepdims, stream])	See `max()`.
`mean`(self[, axis, keepdims, stream])	See `mean()`.
`min`(self[, axis, keepdims, stream])	See `min()`.
`moveaxis`(self, source, destination, *[, stream])	See `moveaxis()`.
`prod`(self[, axis, keepdims, stream])	See `prod()`.
`reciprocal`(self, *[, stream])	See `reciprocal()`.
`reshape`(self, *args[, stream])	Equivalent to `reshape()` but the shape can be passed either as a tuple or as separate arguments.
`round`(self, /[, decimals, stream])	See `round()`.
`rsqrt`(self, *[, stream])	See `rsqrt()`.
`sin`(self, *[, stream])	See `sin()`.
`split`(self, indices_or_sections[, axis, stream])	See `split()`.
`sqrt`(self, *[, stream])	See `sqrt()`.
`square`(self, *[, stream])	See `square()`.
`squeeze`(self[, axis, stream])	See `squeeze()`.
`sum`(self[, axis, keepdims, stream])	See `sum()`.
`swapaxes`(self, axis1, axis2, *[, stream])	See `swapaxes()`.
`tolist`(self)	Convert the array to a Python `list`.
`transpose`(self, *args[, stream])	Equivalent to `transpose()` but the axes can be passed either as a tuple or as separate arguments.
`var`(self[, axis, keepdims, ddof, stream])	See `var()`.

Skip to main content

`T`	Equivalent to calling `self.transpose()` with no arguments.
`at`	Used to apply updates at the given indices.
`dtype`	The array's `Dtype`.
`itemsize`	The size of the array's datatype in bytes.
`nbytes`	The number of bytes in the array.
`ndim`	The array's dimension.
`shape`	The shape of the array as a Python tuple.
`size`	Number of elements in the array.

Previous
Array

Next
mlx.core.array.astype

Print to PDF

mlx.core.array.astype

array.astype(*self: array*, *dtype: Dtype*, *stream: Union[None, Stream, Device] = None*) → array

Cast the array to a specified type.

Parameters::

- **dtype** (*Dtype*) – Type to which the array is cast.
- **stream** (*Stream*) – Stream (or device) for the operation.

Returns::

The array with type `dtype`.

Return type::

array

Previous
mlx.core.array

Next
mlx.core.array.item

mlx.core.array.item

array.item(*self: array*) → object

Access the value of a scalar array.

Returns::

Standard Python scalar.

Previous
mlx.core.array.astype

Next
mlx.core.array.tolist

mlx.core.array.tolist

Print to PDF

array.tolist(*self: array*) → object

Convert the array to a Python `list`.

Returns::

The Python list.

If the array is a scalar then a standard Python scalar is returned.

If the array has more than one dimension then the result is a nested list of lists.

The value type of the list corresponding to the last dimension is either `bool`, `int` or `float` depending on the `dtype` of the array.

Return type::

list

Previous
mlx.core.array.item

Next
mlx.core.array.dtype

mlx.core.array.dtype

property array.**dtype**

The array's `Dtype`.

Previous
mlx.core.array.tolist

Next
mlx.core.array.ndim

mlx.core.array.ndim

property array.**ndim**

The array's dimension.

Previous
mlx.core.array.dtype

Next
mlx.core.array.shape

Neural Networks

Contents

- Quick Start with Neural Networks
- The Module Class
- Value and Grad

Writing arbitrarily complex neural networks in MLX can be done using only `mlx.core.array` and `mlx.core.value_and_grad()`. However, this requires the user to write again and again the same simple neural network operations as well as handle all the parameter state and initialization manually and explicitly.

The module `mlx.nn` solves this problem by providing an intuitive way of composing neural network layers, initializing their parameters, freezing them for finetuning and more.

Quick Start with Neural Networks

```
import mlx.core as mx
import mlx.nn as nn

class MLP(nn.Module):
    def __init__(self, in_dims: int, out_dims: int):
        super().__init__()

        self.layers = [
            nn.Linear(in_dims, 128),
            nn.Linear(128, 128),
            nn.Linear(128, out_dims),
        ]

    def __call__(self, x):
        for i, l in enumerate(self.layers):
            x = mx.maximum(x, 0) if i > 0 else x
            x = l(x)
        return x

# The model is created with all its parameters but nothing is initialized
# yet because MLX is lazily evaluated
mlp = MLP(2, 10)
```

Skip to main content

```
print(params["layers"][0]["weight"].shape)

# Printing a parameter will cause it to be evaluated and thus initialized
print(params["layers"][0])

# We can also force evaluate all parameters to initialize the model
mx.eval(mlp.parameters())

# A simple loss function.
# NOTE: It doesn't matter how it uses the mlp model. It currently captures
#       it from the local scope. It could be a positional argument or a
#       keyword argument.
def l2_loss(x, y):
    y_hat = mlp(x)
    return (y_hat - y).square().mean()

# Calling `nn.value_and_grad` instead of `mx.value_and_grad` returns the
# gradient with respect to `mlp.trainable_parameters()`
loss_and_grad = nn.value_and_grad(mlp, l2_loss)
```

The Module Class

The workhorse of any neural network library is the `Module` class. In MLX the `Module` class is a container of `mlx.core.array` or `Module` instances. Its main function is to provide a way to recursively **access** and **update** its parameters and those of its submodules.

Parameters

A parameter of a module is any public member of type `mlx.core.array` (its name should not start with `_`). It can be arbitrarily nested in other `Module` instances or lists and dictionaries.

`Module.parameters()` can be used to extract a nested dictionary with all the parameters of a module and its submodules.

A `Module` can also keep track of "frozen" parameters. See the `Module.freeze()` method for more details. `mlx.nn.value_and_grad()` the gradients returned will be with respect to these trainable parameters.

Updating the Parameters

MLX modules allow accessing and updating individual parameters. However, most times

Skip to main content

`Module.update()`.

Inspecting Modules

The simplest way to see the model architecture is to print it. Following along with the above example, you can print the `MLP` with:

```
print(mlp)
```

This will display:

```
MLP(
  (layers.0): Linear(input_dims=2, output_dims=128, bias=True)
  (layers.1): Linear(input_dims=128, output_dims=128, bias=True)
  (layers.2): Linear(input_dims=128, output_dims=10, bias=True)
)
```

To get more detailed information on the arrays in a `Module` you can use `mlx.utils.tree_map()` on the parameters. For example, to see the shapes of all the parameters in a `Module` do:

```
from mlx.utils import tree_map
shapes = tree_map(lambda p: p.shape, mlp.parameters())
```

As another example, you can count the number of parameters in a `Module` with:

```
from mlx.utils import tree_flatten
num_params = sum(v.size for _, v in tree_flatten(mlp.parameters()))
```

Value and Grad

Using a `Module` does not preclude using MLX's high order function transformations (`mlx.core.value_and_grad()`, `mlx.core.grad()`, etc.). However, these function transformations assume pure functions, namely the parameters should be passed as an argument to the function being transformed.

There is an easy pattern to achieve that with MLX modules

Skip to main content

```
model = ...

def f(params, other_inputs):
    model.update(params)  # <---- Necessary to make the model use the passed
    return model(other_inputs)

f(model.trainable_parameters(), mx.zeros((10,)))
```

However, `mlx.nn.value_and_grad()` provides precisely this pattern and only computes the gradients with respect to the trainable parameters of the model.

In detail:

- it wraps the passed function with a function that calls `Module.update()` to make sure the model is using the provided parameters.
- it calls `mlx.core.value_and_grad()` to transform the function into a function that also computes the gradients with respect to the passed parameters.
- it wraps the returned function with a function that passes the trainable parameters as the first argument to the function returned by `mlx.core.value_and_grad()`

`value_and_grad`(model, fn)	Transform the passed function `fn` to a function that computes the gradients of `fn` wrt the model's trainable parameters and also its value.

Module
- mlx.nn.Module.training
- mlx.nn.Module.state
- mlx.nn.Module.apply
- mlx.nn.Module.apply_to_modules
- mlx.nn.Module.children
- mlx.nn.Module.eval
- mlx.nn.Module.filter_and_map
- mlx.nn.Module.freeze
- mlx.nn.Module.leaf_modules
- mlx.nn.Module.load_weights
- mlx.nn.Module.modules
- mlx.nn.Module.named_modules

Skip to main content

mlx.nn.Module.save_weights
mlx.nn.Module.train
mlx.nn.Module.trainable_parameters
mlx.nn.Module.unfreeze
mlx.nn.Module.update
mlx.nn.Module.update_modules
Layers
mlx.nn.ALiBi
mlx.nn.AvgPool1d
mlx.nn.AvgPool2d
mlx.nn.BatchNorm
mlx.nn.Conv1d
mlx.nn.Conv2d
mlx.nn.Dropout
mlx.nn.Dropout2d
mlx.nn.Dropout3d
mlx.nn.Embedding
mlx.nn.GELU
mlx.nn.GroupNorm
mlx.nn.GRU
mlx.nn.InstanceNorm
mlx.nn.LayerNorm
mlx.nn.Linear
mlx.nn.LSTM
mlx.nn.MaxPool1d
mlx.nn.MaxPool2d
mlx.nn.Mish
mlx.nn.MultiHeadAttention
mlx.nn.PReLU
mlx.nn.QuantizedLinear
mlx.nn.RMSNorm

Skip to main content

mlx.nn.RNN
mlx.nn.RoPE
mlx.nn.SELU
mlx.nn.Sequential
mlx.nn.SiLU
mlx.nn.SinusoidalPositionalEncoding
mlx.nn.Softshrink
mlx.nn.Step
mlx.nn.Transformer
mlx.nn.Upsample
Functions
mlx.nn.elu
mlx.nn.gelu
mlx.nn.gelu_approx
mlx.nn.gelu_fast_approx
mlx.nn.glu
mlx.nn.hardswish
mlx.nn.leaky_relu
mlx.nn.log_sigmoid
mlx.nn.log_softmax
mlx.nn.mish
mlx.nn.prelu
mlx.nn.relu
mlx.nn.relu6
mlx.nn.selu
mlx.nn.sigmoid
mlx.nn.silu
mlx.nn.softmax
mlx.nn.softplus
mlx.nn.softshrink
mlx.nn.step

Skip to main content

Loss Functions
- mlx.nn.losses.binary_cross_entropy
- mlx.nn.losses.cosine_similarity_loss
- mlx.nn.losses.cross_entropy
- mlx.nn.losses.gaussian_nll_loss
- mlx.nn.losses.hinge_loss
- mlx.nn.losses.huber_loss
- mlx.nn.losses.kl_div_loss
- mlx.nn.losses.l1_loss
- mlx.nn.losses.log_cosh_loss
- mlx.nn.losses.margin_ranking_loss
- mlx.nn.losses.mse_loss
- mlx.nn.losses.nll_loss
- mlx.nn.losses.smooth_l1_loss
- mlx.nn.losses.triplet_loss

Initializers
- mlx.nn.init.constant
- mlx.nn.init.normal
- mlx.nn.init.uniform
- mlx.nn.init.identity
- mlx.nn.init.glorot_normal
- mlx.nn.init.glorot_uniform
- mlx.nn.init.he_normal
- mlx.nn.init.he_uniform

Previous
mlx.core.metal.set_cache_limit

Next
mlx.nn.value_and_grad

Print to PDF

mlx.nn.value_and_grad

mlx.nn.value_and_grad(*model: Module*, *fn: Callable*)

Transform the passed function `fn` to a function that computes the gradients of `fn` wrt the model's trainable parameters and also its value.

Parameters::

- **model** (*Module*) – The model whose trainable parameters to compute gradients for
- **fn** (*Callable*) – The scalar function to compute gradients for

Returns::

A callable that returns the value of `fn` and the gradients wrt the trainable parameters of `model`

Previous
Neural Networks

Next
Module

Print to PDF

Module

class mlx.nn.Module

Base class for building neural networks with MLX.

All the layers provided in `mlx.nn.layers` subclass this class and your models should do the same.

A `Module` can contain other `Module` instances or `mlx.core.array` instances in arbitrary nesting of python lists or dicts. The `Module` then allows recursively extracting all the `mlx.core.array` instances using `mlx.nn.Module.parameters()`.

In addition, the `Module` has the concept of trainable and non trainable parameters (called “frozen”). When using `mlx.nn.value_and_grad()` the gradients are returned only with respect to the trainable parameters. All arrays in a module are trainable unless they are added in the “frozen” set by calling `freeze()`.

```
import mlx.core as mx
import mlx.nn as nn

class MyMLP(nn.Module):
    def __init__(self, in_dims: int, out_dims: int, hidden_dims: int = 16
        super().__init__()

        self.in_proj = nn.Linear(in_dims, hidden_dims)
        self.out_proj = nn.Linear(hidden_dims, out_dims)

    def __call__(self, x):
        x = self.in_proj(x)
        x = mx.maximum(x, 0)
        return self.out_proj(x)

model = MyMLP(2, 1)

# All the model parameters are created but since MLX is lazy by
# default, they are not evaluated yet. Calling `mx.eval` actually
# allocates memory and initializes the parameters.
mx.eval(model.parameters())

# Setting a parameter to a new value is as simply as accessing that
# parameter and assigning a new array to it.
model.in_proj.weight = model.in_proj.weight * 2
mx.eval(model.parameters())
```

Skip to main content

`Module.training`	Boolean indicating if the model is in training mode.
`Module.state`	The module's state dictionary

Methods

`Module.apply`(map_fn[, filter_fn])	Map all the parameters using the provided `map_fn` and immediately update the module with the mapped parameters.
`Module.apply_to_modules`(apply_fn)	Apply a function to all the modules in this instance (including this instance).
`Module.children`()	Return the direct descendants of this Module instance.
`Module.eval`()	Set the model to evaluation mode.
`Module.filter_and_map`(filter_fn[, map_fn, ...])	Recursively filter the contents of the module using `filter_fn`, namely only select keys and values where `filter_fn` returns true.
`Module.freeze`(*[, recurse, keys, strict])	Freeze the Module's parameters or some of them.
`Module.leaf_modules`()	Return the submodules that do not contain other modules.
`Module.load_weights`(file_or_weights[, strict])	Update the model's weights from a `.npz`, a `.safetensors` file, or a list.
`Module.modules`()	Return a list with all the modules in this instance.
`Module.named_modules`()	Return a list with all the modules in this instance and their name with dot notation.
`Module.parameters`()	Recursively return all the `mlx.core.array` members of this Module as a dict of dicts and lists.

Skip to main content

`Module.save_weights`(file)	Save the model's weights to a file.
`Module.train`([mode])	Set the model in or out of training mode.
`Module.trainable_parameters`()	Recursively return all the non frozen `mlx.core.array` members of this Module as a dict of dicts and lists.
`Module.unfreeze`(*[, recurse, keys, strict])	Unfreeze the Module's parameters or some of them.
`Module.update`(parameters)	Replace the parameters of this Module with the provided ones in the dict of dicts and lists.
`Module.update_modules`(modules)	Replace the child modules of this `Module` instance with the provided ones in the dict of dicts and lists.

Previous
mlx.nn.value_and_grad

Next
mlx.nn.Module.training

mlx.nn.Module.training

property Module.**training**

Boolean indicating if the model is in training mode.

Previous
Module

Next
mlx.nn.Module.state

Print to PDF

mlx.nn.Module.state

property Module.**state**

The module's state dictionary

The module's state dictionary contains any attribute set on the module including parameters in `Module.parameters()`

Unlike `Module.parameters()`, the `Module.state` property is a reference to the module's state. Updates to it will be reflected in the original module.

Previous
mlx.nn.Module.training

Next
mlx.nn.Module.apply

Print to PDF

Layers

Skip to main content

`ALiBi`()	
`AvgPool1d`(kernel_size[, stride, padding])	Applies 1-dimensional average pooling.
`AvgPool2d`(kernel_size[, stride, padding])	Applies 2-dimensional average pooling.
`BatchNorm`(num_features[, eps, momentum, ...])	Applies Batch Normalization over a 2D or 3D input.
`Conv1d`(in_channels, out_channels, kernel_size)	Applies a 1-dimensional convolution over the multi-channel input sequence.
`Conv2d`(in_channels, out_channels, kernel_size)	Applies a 2-dimensional convolution over the multi-channel input image.
`Dropout`([p])	Randomly zero a portion of the elements during training.
`Dropout2d`([p])	Apply 2D channel-wise dropout during training.
`Dropout3d`([p])	Apply 3D channel-wise dropout during training.
`Embedding`(num_embeddings, dims)	Implements a simple lookup table that maps each input integer to a high-dimensional vector.
`GELU`([approx])	Applies the Gaussian Error Linear Units.
`GroupNorm`(num_groups, dims[, eps, affine, ...])	Applies Group Normalization [1] to the inputs.
`GRU`(input_size, hidden_size[, bias])	A gated recurrent unit (GRU) RNN layer.
`InstanceNorm`(dims[, eps, affine])	Applies instance normalization [1] on the inputs.

Skip to main content

`LayerNorm`(dims[, eps, affine])	Applies layer normalization [1] on the inputs.
`Linear`(input_dims, output_dims[, bias])	Applies an affine transformation to the input.
`LSTM`(input_size, hidden_size[, bias])	An LSTM recurrent layer.
`MaxPool1d`(kernel_size[, stride, padding])	Applies 1-dimensional max pooling.
`MaxPool2d`(kernel_size[, stride, padding])	Applies 2-dimensional max pooling.
`Mish`()	Applies the Mish function, element-wise.
`MultiHeadAttention`(dims, num_heads[, ...])	Implements the scaled dot product attention with multiple heads.
`PReLU`([num_parameters, init])	Applies the element-wise parametric ReLU.
`QuantizedLinear`(input_dims, output_dims[, ...])	Applies an affine transformation to the input using a quantized weight matrix.
`RMSNorm`(dims[, eps])	Applies Root Mean Square normalization [1] to the inputs.
`ReLU`()	Applies the Rectified Linear Unit.
`RNN`(input_size, hidden_size[, bias, ...])	An Elman recurrent layer.
`RoPE`(dims[, traditional, base, scale])	Implements the rotary positional encoding.
`SELU`()	Applies the Scaled Exponential Linear Unit.
`Sequential`(*modules)	A layer that calls the passed callables in order.
`SiLU`()	Applies the Sigmoid Linear Unit.
`SinusoidalPositionalEncoding`(dims[, ...])	Implements sinusoidal positional

Skip to main content

`Softshrink`([lambd])	Applies the Softshrink function.
`Step`([threshold])	Applies the Step Activation Function.
`Transformer`(dims, num_heads, ...)	Implements a standard Transformer model.
`Upsample`(scale_factor[, mode, align_corners])	Upsample the input signal spatially.

Previous
mlx.nn.Module.update_modules

Next
mlx.nn.ALiBi

Functions

Layers without parameters (e.g. activation functions) are also provided as simple functions.

Skip to main content

`elu`(x[, alpha])	Applies the Exponential Linear Unit.
`gelu`(x)	Applies the Gaussian Error Linear Units function.
`gelu_approx`(x)	An approximation to Gaussian Error Linear Unit.
`gelu_fast_approx`(x)	A fast approximation to Gaussian Error Linear Unit.
`glu`(x[, axis])	Applies the gated linear unit function.
`hardswish`(x)	Applies the hardswish function, element-wise.
`leaky_relu`(x[, negative_slope])	Applies the Leaky Rectified Linear Unit.
`log_sigmoid`(x)	Applies the Log Sigmoid function.
`log_softmax`(x[, axis])	Applies the Log Softmax function.
`mish`(x)	Applies the Mish function, element-wise.
`prelu`(x, alpha)	Applies the element-wise parametric ReLU.
`relu`(x)	Applies the Rectified Linear Unit.
`relu6`(x)	Applies the Rectified Linear Unit 6.
`selu`(x)	Applies the Scaled Exponential Linear Unit.
`sigmoid`(x)	Applies the sigmoid function.
`silu`(x)	Applies the Sigmoid Linear Unit.
`softmax`(x[, axis])	Applies the Softmax function.
`softplus`(x)	Applies the Softplus function.
`softshrink`(x[, lambd])	Applies the Softshrink activation function.
`step`(x[, threshold])	Applies the Step Activation Function.
`tanh`(x)	Applies the hyperbolic tangent function.

Previous
mlx.nn.Upsample

Next
mlx.nn.elu

Print to PDF

Loss Functions

Skip to main content

`binary_cross_entropy`(inputs, targets[, ...])	Computes the binary cross entropy loss.
`cosine_similarity_loss`(x1, x2[, axis, eps, ...])	Computes the cosine similarity between the two inputs.
`cross_entropy`(logits, targets[, weights, ...])	Computes the cross entropy loss.
`gaussian_nll_loss`(inputs, targets, vars[, ...])	Computes the negative log likelihood loss for a Gaussian distribution.
`hinge_loss`(inputs, targets[, reduction])	Computes the hinge loss between inputs and targets.
`huber_loss`(inputs, targets[, delta, reduction])	Computes the Huber loss between inputs and targets.
`kl_div_loss`(inputs, targets[, axis, reduction])	Computes the Kullback-Leibler divergence loss.
`l1_loss`(predictions, targets[, reduction])	Computes the L1 loss.
`log_cosh_loss`(inputs, targets[, reduction])	Computes the log cosh loss between inputs and targets.
`margin_ranking_loss`(inputs1, inputs2, targets)	Calculate the margin ranking loss that loss given inputs x_1, x_2 and a label y (containing 1 or -1).
`mse_loss`(predictions, targets[, reduction])	Computes the mean squared error loss.
`nll_loss`(inputs, targets[, axis, reduction])	Computes the negative log likelihood loss.
`smooth_l1_loss`(predictions, targets[, beta, ...])	Computes the smooth L1 loss.
`triplet_loss`(anchors, positives, negatives)	Computes the triplet loss for a set of anchor, positive, and negative samples.

Previous
mlx.nn.tanh

Nex
mlx.nn.losses.binary_cross_entrop