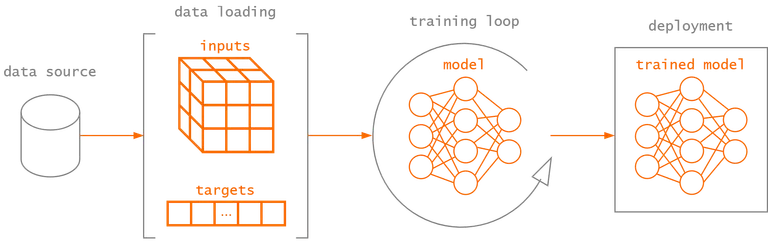
# Data and Data loading

**Introduction**

Building a deep learning model to tackle a particular task involves several steps. Whether the objective is to classify images, translating texts or predicting a continuous variable, the typical machine learning or deep learning process remains the same. Given a data source specific to the problem at hand, a deep learning algorithm automatically discovers useful representations from raw data to perform the defined objective. To tackle this problem, we need a few things: a source of training data (i.e., dataset), a model to train and a way to estimate the parameters of the model so that it will give good predictions, which is encapsulated by the training loop. This simplified process is illustrated in the figure below, where the last step is to deploy the trained model to perform predictions on new data.



High Level Process of Building a Deep Learning Model

In order to achieve this, Burn provides key components that serve as the building blocks of the framework and your future projects. In this series, we'll provide an overview of these components and draw parallels with the help of the example from our basic workflow guide. The guide trains a simple convolutional neural network to recognize handwritten digits using the MNIST dataset.

For the first part of this series, we will cover data loading with Burn.

**Dataset & Data Loading**

At its core, a dataset is a collection of data typically related to a specific analysis or processing task. The data modality can vary depending on the task, but most datasets primarily consist of images, texts, audio or videos.

For the first part of this series, we will cover data loading with Burn.

As mentioned earlier, this data source represents an integral part of machine learning to successfully train a model. Thus, it is essential to provide a convenient and performant API to handle your data. Since this process varies wildly from one problem to another, it is defined as a trait that should be implemented on your type. The dataset trait is quite simple:

pub trait Dataset<I>: Send + Sync {

fn get(&self, index: usize) -> Option<I>;

fn len(&self) -> usize;

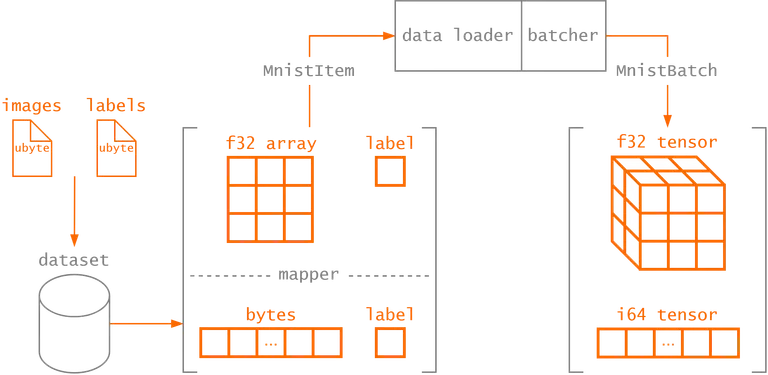
}

The trait assumes a fixed-length set of items that can be randomly accessed in constant time. Burn also provides transformations that can be applied lazily to modify one or multiple datasets, such as the MapperDataset transformation which is used later in this post. More details are available in the book.

During training, the dataset is used to access the data samples and, for most use cases in supervised learning, their corresponding ground-truth labels. The Dataset trait implementation is responsible to retrieve the data from its source, usually some sort of data storage. At this point, the dataset could be naively iterated over to provide the model a single sample to process at a time, but this is not very efficient.

Instead, we collect multiple samples that the model can process as a batch to fully leverage modern hardware (e.g., GPUs - which have impressing parallel processing capabilities). Since each data sample in the dataset can be collected independently, the data loading is typically done in parallel to further speed things up. In this case, we parallelize the data loading using a multi-threaded BatchDataLoader to obtain a sequence of items from the Dataset implementation. Finally, the sequence of items is combined into a batched multidimensional array (i.e., tensor[[4](https://burn.dev/blog/building-blocks-dataset/" \l "reference-4)]) that can be used as input to a model with the Batcher trait implementation. Other tensor operations can be performed during this step to prepare the batch data, as we will demonstrate in the following section. The process is illustrated in the figure below for the MNIST dataset, which we'll see in the next section.

Burn Data Loading Process



**Leading By Example**

Now that we have a high-level map of the data loading components in Burn, let's see how we can define the data loading process for the MNIST dataset. Although we have conveniently implemented the MnistDataset[[5](https://burn.dev/blog/building-blocks-dataset/" \l "reference-5)] used in the guide, we'll go over its implementation to demonstrate how the Dataset and Batcher traits are used.

Make sure you have all the required imports to follow along.

use std::{

fs::{create\_dir\_all, File},

io::{Read, Seek, SeekFrom},

path::{Path, PathBuf},

};

use burn::{

backend::{wgpu::WgpuDevice, Wgpu},

data::{

dataloader::{batcher::Batcher, DataLoaderBuilder},

dataset::{

transform::{Mapper, MapperDataset},

Dataset, InMemDataset,

},

},

tensor::{backend::Backend, Int, Tensor, TensorData},

};

use burn\_common::network::downloader::download\_file\_as\_bytes;

use flate2::read::GzDecoder;

use serde::{Deserialize, Serialize};

Additionally, your Cargo.toml should look like this.

[package]

name = "building\_blocks\_dataset"

version = "0.1.0"

edition = "2021"

[dependencies]

burn = { version = "0.14.0", features = ["dataset", "wgpu"] }

burn-common = { version = "0.14.0", features = ["network"] }

serde = { version = "1.0.206", features = ["derive"] }

dirs = "5.0.1"

flate2 = "1.0.31"

The MNIST dataset of handwritten digits has a training set of 60,000 examples and a test set of 10,000 examples. A single item in the dataset is represented by a 28 × 28 pixels black-and-white image (stored as raw bytes) with its corresponding label (a digit between 0 and 9). When loading the data from its source, a single item can be represented by the MnistItemRaw struct.

#[derive(Deserialize, Debug, Clone)]

struct MnistItemRaw {

pub image\_bytes: Vec<u8>,

pub label: u8,

}

With single-channel images of such low resolution, the entire training and test sets can be loaded in memory at once. Therefore, we leverage the already existing InMemDataset[[6](https://burn.dev/blog/building-blocks-dataset/" \l "reference-6)] to retrieve the raw images and labels data. At this point, the image data is still just a bunch of bytes, but we want to retrieve the structured image data in its intended form. For that, we can define a MapperDataset that transforms the raw image bytes to a 2D array image (which we convert to float while we're at it).

const WIDTH: usize = 28;

const HEIGHT: usize = 28;

*/// MNIST item.*

#[derive(Deserialize, Serialize, Debug, Clone)]

pub struct MnistItem {

*/// Image as a 2D array of floats.*

pub image: [[f32; WIDTH]; HEIGHT],

*/// Label of the image.*

pub label: u8,

}

struct BytesToImage;

impl Mapper<MnistItemRaw, MnistItem> *for* BytesToImage {

*/// Convert a raw MNIST item (image bytes) to a MNIST item (2D array image).*

fn map(&self, item: &MnistItemRaw) -> MnistItem {

*// Ensure the image dimensions are correct.*

debug\_assert\_eq!(item.image\_bytes.len(), WIDTH \* HEIGHT);

*// Convert the image to a 2D array of floats.*

let mut image\_array = [[0f32; WIDTH]; HEIGHT];

*for* (i, pixel) in item.image\_bytes.iter().enumerate() {

let x = i % WIDTH;

let y = i / HEIGHT;

image\_array[y][x] = \*pixel as f32;

}

MnistItem {

image: image\_array,

label: item.label,

}

}

}

type MappedDataset = MapperDataset<InMemDataset<MnistItemRaw>, BytesToImage, MnistItemRaw>;

*/// The MNIST dataset.*

pub struct MnistDataset {

dataset: MappedDataset,

}

To construct the MnistDataset, the data source must be parsed into the expected MappedDataset type. We automatically download the data files from the web and parse the files to retrieve the images and labels. The code is included for completeness, but just know that each image is retrieved as a list of bytes and each digit label as an 8-bit unsigned integer. Thus, we have a Vec<Vec<u8>> to represent images and a Vec<u8> to represent labels. Since both the train and test sets use the same file format, we can separate the functionality to load the train() and test() dataset.

*// CVDF mirror of http://yann.lecun.com/exdb/mnist/*

const URL: &str = "https://storage.googleapis.com/cvdf-datasets/mnist/";

const TRAIN\_IMAGES: &str = "train-images-idx3-ubyte";

const TRAIN\_LABELS: &str = "train-labels-idx1-ubyte";

const TEST\_IMAGES: &str = "t10k-images-idx3-ubyte";

const TEST\_LABELS: &str = "t10k-labels-idx1-ubyte";

impl MnistDataset {

*/// Creates a new train dataset.*

pub fn train() -> Self {

Self::new("train")

}

*/// Creates a new test dataset.*

pub fn test() -> Self {

Self::new("test")

}

fn new(split: &str) -> Self {

*// Download dataset*

let root = MnistDataset::download(split);

*// Parse data as vector of images bytes and vector of labels*

let images = MnistDataset::read\_images(&root, split);

let labels = MnistDataset::read\_labels(&root, split);

*// Collect as vector of MnistItemRaw*

let items: Vec<\_> = images

.into\_iter()

.zip(labels)

.map(|(image\_bytes, label)| MnistItemRaw { image\_bytes, label })

.collect();

let dataset = InMemDataset::new(items);

let dataset = MapperDataset::new(dataset, BytesToImage);

Self { dataset }

}

*/// Read images at the provided path for the specified split.*

*/// Each image is a vector of bytes.*

fn read\_images<P: AsRef<Path>>(root: &P, split: &str) -> Vec<Vec<u8>> {

let file\_name = *if* split == "train" {

TRAIN\_IMAGES

} *else* {

TEST\_IMAGES

};

let file\_name = root.as\_ref().join(file\_name);

*// Read number of images from 16-byte header metadata*

let mut f = File::open(file\_name).unwrap();

let mut buf = [0u8; 4];

let \_ = f.seek(SeekFrom::Start(4)).unwrap();

f.read\_exact(&mut buf)

.expect("Should be able to read image file header");

let size = u32::from\_be\_bytes(buf);

let mut buf\_images: Vec<u8> = vec![0u8; WIDTH \* HEIGHT \* (size as usize)];

let \_ = f.seek(SeekFrom::Start(16)).unwrap();

f.read\_exact(&mut buf\_images)

.expect("Should be able to read image file header");

buf\_images

.chunks(WIDTH \* HEIGHT)

.map(|chunk| chunk.to\_vec())

.collect()

}

*/// Read labels at the provided path for the specified split.*

fn read\_labels<P: AsRef<Path>>(root: &P, split: &str) -> Vec<u8> {

let file\_name = *if* split == "train" {

TRAIN\_LABELS

} *else* {

TEST\_LABELS

};

let file\_name = root.as\_ref().join(file\_name);

*// Read number of labels from 8-byte header metadata*

let mut f = File::open(file\_name).unwrap();

let mut buf = [0u8; 4];

let \_ = f.seek(SeekFrom::Start(4)).unwrap();

f.read\_exact(&mut buf)

.expect("Should be able to read label file header");

let size = u32::from\_be\_bytes(buf);

let mut buf\_labels: Vec<u8> = vec![0u8; size as usize];

let \_ = f.seek(SeekFrom::Start(8)).unwrap();

f.read\_exact(&mut buf\_labels)

.expect("Should be able to read labels from file");

buf\_labels

}

*/// Download the MNIST dataset files from the web.*

*/// Panics if the download cannot be completed or the content*

*/// of the file cannot be written to disk.*

fn download(split: &str) -> PathBuf {

*// Dataset files are stored un the burn-dataset cache directory*

let cache\_dir = dirs::home\_dir()

.expect("Could not get home directory")

.join(".cache")

.join("burn-dataset");

let split\_dir = cache\_dir.join("mnist").join(split);

*if* !split\_dir.exists() {

create\_dir\_all(&split\_dir).expect("Failed to create base directory");

}

*// Download split files*

*match* split {

"train" => {

MnistDataset::download\_file(TRAIN\_IMAGES, &split\_dir);

MnistDataset::download\_file(TRAIN\_LABELS, &split\_dir);

}

"test" => {

MnistDataset::download\_file(TEST\_IMAGES, &split\_dir);

MnistDataset::download\_file(TEST\_LABELS, &split\_dir);

}

\_ => panic!("Invalid split specified {}", split),

};

split\_dir

}

*/// Download a file from the MNIST dataset URL to the destination directory.*

*/// File download progress is reported with the help of a [progress bar](indicatif).*

fn download\_file<P: AsRef<Path>>(name: &str, dest\_dir: &P) -> PathBuf {

*// Output file name*

let file\_name = dest\_dir.as\_ref().join(name);

*if* !file\_name.exists() {

*// Download gzip file*

let bytes = download\_file\_as\_bytes(&format!("{URL}{name}.gz"), name);

*// Create file to write the downloaded content to*

let mut output\_file = File::create(&file\_name).unwrap();

*// Decode gzip file content and write to disk*

let mut gz\_buffer = GzDecoder::new(&bytes[..]);

std::io::copy(&mut gz\_buffer, &mut output\_file).unwrap();

}

file\_name

}

}

As mentioned earlier, the MnistDataset simply wraps a MapperDataset instance with InMemDataset. This makes the Dataset trait implementation quite straightforward.

impl Dataset<MnistItem> *for* MnistDataset {

fn get(&self, index: usize) -> Option<MnistItem> {

self.dataset.get(index)

}

fn len(&self) -> usize {

self.dataset.len()

}

}

The only thing missing now is the Batcher. We'll define the MnistBatcher with the device on which the tensor should be sent before being passed to the model. Note that the device is an associative type of the Backend trait since not all backends expose the same devices.

#[derive(Clone)]

pub struct MnistBatcher<B: Backend> {

device: B::Device,

}

impl<B: Backend> MnistBatcher<B> {

pub fn new(device: B::Device) -> Self {

Self { device }

}

}

Next, we need to actually implement the batching logic. The batch(items) method takes a list of MnistItem retrieved by the dataloader as input and returns a batch containing the images in the form of a 3D tensor, along with the targets tensor that contains the indices of the correct digit class.

#[derive(Clone, Debug)]

pub struct MnistBatch<B: Backend> {

pub images: Tensor<B, 3>,

pub targets: Tensor<B, 1, Int>,

}

impl<B: Backend> Batcher<MnistItem, MnistBatch<B>> *for* MnistBatcher<B> {

fn batch(&self, items: Vec<MnistItem>) -> MnistBatch<B> {

let images = items *// take items Vec<MnistItem>*

.iter() *// create an iterator over it*

.map(|item| TensorData::from(item.image)) *// for each item, convert the image to float32 data struct*

.map(|data| Tensor::<B, 2>::from\_data(data, &self.device)) *// for each data struct, create a tensor on the device*

.map(|tensor| tensor.reshape([1, HEIGHT, WIDTH])) *// for each tensor, reshape to the image dimensions [C, H, W]*

.map(|tensor| ((tensor / 255) - 0.1307) / 0.3081) *// for each image tensor, apply normalization*

.collect(); *// collect the values into a new vector of tensors*

let targets = items

.iter()

.map(|item| {

Tensor::<B, 1, Int>::from\_data(TensorData::from([item.label as i32]), &self.device)

})

.collect();

let images = Tensor::cat(images, 0).to\_device(&self.device);

let targets = Tensor::cat(targets, 0).to\_device(&self.device);

MnistBatch { images, targets }

}

}

To achieve this, we parse the image array into a TensorData struct provided by Burn to encapsulate tensor storage information without being specific for a backend. The image is then reshaped to the image dimensions [1, HEIGHT, WIDTH] with a single channel and normalized according to the MNIST dataset statistics. Each vector of images and targets are then concatenated and the MnistBatch is returned.

At this point, we now have everything in place to load the MNIST data for our application. If you want to test it out, you can use the DataLoaderBuilder with the MnistDataset and MnistBatcher we just defined to iterate over the batches.

pub fn main() {

type MyBackend = Wgpu<f32, i32>;

let batch\_size = 32;

let num\_workers = 4;

*// Create a default Wgpu device*

let device = WgpuDevice::default();

*// Create a data loader for the MNIST test data*

let batcher = MnistBatcher::<MyBackend>::new(device);

let dataloader = DataLoaderBuilder::new(batcher)

.batch\_size(batch\_size)

.num\_workers(num\_workers)

.build(MnistDataset::test());

*// Iterate over the MNIST test data batches*

*for* (iteration, batch) in dataloader.iter().enumerate() {

println!(

"[Iteration {}] Images {:?} | Targets {:?}",

iteration,

batch.images.dims(),

batch.targets.dims(),

);

}

}

This completes our tour of Burn's data loading components.