# Overview

Welcome to The Burn Book 👋

This book will help you get started with the Burn deep learning framework, whether you are an advanced user or a beginner. We have crafted some sections for you:

- Basic Workflow: From Training to Inference: We'll start with the fundamentals, guiding you through the entire workflow, from training your models to deploying them for inference. This section lays the groundwork for your Burn expertise.
- Building Blocks: Dive deeper into Burn's core components, understanding how they fit together. This knowledge forms the basis for more advanced usage and customization.
- Saving & Loading Models: Learn how to easily save and load your trained models.
- Custom Training Loop: Gain the power to customize your training loops, fine-tuning your models to meet your specific requirements. This section empowers you to harness Burn's flexibility to its fullest.
- Importing Models: Learn how to import ONNX and PyTorch models, expanding your compatibility with other deep learning ecosystems.
- Advanced: Finally, venture into advanced topics, exploring Burn's capabilities at their peak. This section caters to those who want to push the boundaries of what's possible with Burn.

Throughout the book, we assume a basic understanding of deep learning concepts, but we may refer to additional material when it seems appropriate.

# Why Burn?

Why bother with the effort of creating an entirely new deep learning framework from scratch when PyTorch, TensorFlow, and other frameworks already exist? Spoiler alert: Burn isn't merely a replication of PyTorch or TensorFlow in Rust. It represents a novel approach, placing significant emphasis on making the right compromises in the right areas to facilitate exceptional flexibility, high performance, and a seamless developer experience. Burn isn't a framework specialized for only one type of application, it is designed to serve as a versatile framework suitable for a wide range of research and production uses. The foundation of Burn's design revolves around three key user profiles:

**Machine Learning Researchers** require tools to construct and execute experiments efficiently. It's essential for them to iterate quickly on their ideas and design testable experiments which can help them discover new findings. The framework should facilitate the swift implementation of cutting-edge research while ensuring fast execution for testing.

**Machine Learning Engineers** are another important demographic to keep in mind. Their focus leans less on swift implementation and more on establishing robustness, seamless deployment, and cost-effective operations. They seek dependable, economical models capable of achieving objectives without excessive expense. The whole machine learning workflow —from training to inference— must be as efficient as possible with minimal unpredictable behavior.

**Low level Software Engineers** working with hardware vendors want their processing units to run models as fast as possible to gain competitive advantage. This endeavor involves harnessing hardware-specific features such as Tensor Core for Nvidia. Since they are mostly working at a system level, they want to have absolute control over how the computation will be executed.

The goal of Burn is to satisfy all of those personas!

# Getting Started

Burn is a deep learning framework in the Rust programming language. Therefore, it goes without saying that one must understand the basic notions of Rust. Reading the first chapters of the Rust Book is recommended, but don't worry if you're just starting out. We'll try to provide as much context and reference to external resources when required. Just look out for the 🦀 **Rust Note** indicators.

## Installing Rust

For installation instructions, please refer to the installation page. It explains in details the most convenient way for you to install Rust on your computer, which is the very first thing to do to start using Burn.

## Creating a Burn application

Once Rust is correctly installed, create a new Rust application by using Rust's build system and package manager Cargo. It is automatically installed with Rust.

▸ 🦀 **Cargo Cheat Sheet**

In the directory of your choice, run the following:

```
cargo new my_burn_app
```

This will initialize the `my_burn_app` project directory with a `Cargo.toml` file and a `src` directory with an auto-generated `main.rs` file inside. Head inside the directory to check:

```
cd my_burn_app
```

Then, add Burn as a dependency:

```
cargo add burn --features wgpu
```

Finally, compile the local package by executing the following:

```
cargo build
```

That's it, you're ready to start! You have a project configured with Burn and the WGPU backend, which allows to execute low-level operations on any platform using the GPU.

> When using one of the `wgpu` backends, you may encounter compilation errors related to recursive type evaluation. This is due to complex type nesting within the `wgpu` dependency chain.
>
> To resolve this issue, add the following line at the top of your `main.rs` or `lib.rs` file:
>
> ```
> #![recursion_limit = "256"]
> ```
>
> The default recursion limit (128) is often just below the required depth (typically 130-150) due to deeply nested associated types and trait bounds.

## Writing a code snippet

The `src/main.rs` was automatically generated by Cargo, so let's replace its content with the following:

```
use burn::tensor::Tensor;
use burn::backend::Wgpu;

// Type alias for the backend to use.
type Backend = Wgpu;

fn main() {
    let device = Default::default();
    // Creation of two tensors, the first with explicit values and the second
one with ones, with the same shape as the first
    let tensor_1 = Tensor::<Backend, 2>::from_data([[2., 3.], [4., 5.]],
&device);
    let tensor_2 = Tensor::<Backend, 2>::ones_like(&tensor_1);

    // Print the element-wise addition (done with the WGPU backend) of the two
tensors.
    println!("{}", tensor_1 + tensor_2);
}
```

▸ 🦀 **Use Declarations**

▸ 🦀 **Generic Data Types**

By running `cargo run`, you should now see the result of the addition:

```
Tensor {
  data:
[[3.0, 4.0],
 [5.0, 6.0]],
  shape:  [2, 2],
  device:  DefaultDevice,
  backend:  "wgpu",
  kind:  "Float",
  dtype:  "f32",
}
```

While the previous example is somewhat trivial, the upcoming basic workflow section will walk you through a much more relevant example for deep learning applications.

## Using `prelude`

Burn comes with a variety of things in its core library. When creating a new model or using an existing one for inference, you may need to import every single component you used, which could be a little verbose.

To address it, a `prelude` module is provided, allowing you to easily import commonly used structs and macros as a group:

```
use burn::prelude::*;
```

which is equal to:

```
use burn::{
    config::Config,
    module::Module,
    nn,
    tensor::{
        backend::Backend, Bool, Device, ElementConversion, Float, Int, Shape,
Tensor,
        TensorData,
    },
};
```

> For the sake of simplicity, the subsequent chapters of this book will all use this form of importing except in the Building Blocks chapter, as explicit importing aids users in grasping the usage of particular structures and macros.

# Examples

In the next chapter you'll have the opportunity to implement the whole Burn `guide` example yourself in a step by step manner.

Many additional Burn examples are available in the examples directory. Burn examples are organized as library crates with one or more examples that are executable binaries. An example can then be executed using the following cargo command line in the root of the Burn repository:

```
cargo run --example <example name>
```

To learn more about crates and examples, read the Rust section below.

▸ 🦀 **About Rust crates**

The following additional examples are currently available if you want to check them out:

| Example | Description |
|---|---|
| Custom CSV Dataset | Implements a dataset to parse CSV data for a regression task. |
| Regression | Trains a simple MLP on the California Housing dataset to predict the median house value for a district. |
| Custom Image Dataset | Trains a simple CNN on custom image dataset following a simple folder structure. |
| Custom Renderer | Implements a custom renderer to display the `Learner` progress. |
| Image Classification Web | Image classification web browser demo using Burn, WGPU and WebAssembly. |
| MNIST Inference on Web | An interactive MNIST inference demo in the browser. The demo is available online. |
| MNIST Training | Demonstrates how to train a custom `Module` (MLP) with the `Learner` configured to log metrics and keep training checkpoints. |
| Named Tensor | Performs operations with the experimental `NamedTensor` feature. |

| Example | Description |
| --- | --- |
| ONNX Import Inference | Imports an ONNX model pre-trained on MNIST to perform inference on a sample image with Burn. |
| PyTorch Import Inference | Imports a PyTorch model pre-trained on MNIST to perform inference on a sample image with Burn. |
| Text Classification | Trains a text classification transformer model on the AG News or DbPedia datasets. The trained model can then be used to classify a text sample. |
| Text Generation | Trains a text generation transformer model on the DbPedia dataset. |
| Wasserstein GAN MNIST | Trains a WGAN model to generate new handwritten digits based on MNIST. |

For more information on each example, see their respective `README.md` file. Be sure to check out the examples directory for an up-to-date list.

> Note that some examples use the `datasets` library by HuggingFace to download the datasets required in the examples. This is a Python library, which means that you will need to install Python before running these examples. This requirement will be clearly indicated in the example's README when applicable.

# Guide

This guide will walk you through the process of creating a custom model built with Burn. We will train a simple convolutional neural network model on the MNIST dataset and prepare it for inference.

For clarity, we sometimes omit imports in our code snippets. For more details, please refer to the corresponding code in the `examples/guide` directory. We reproduce this example in a step-by-step fashion, from dataset creation to modeling and training in the following sections. It is recommended to use the capabilities of your IDE or text editor to automatically add the missing imports as you add the code snippets to your code.

> Be sure to checkout the git branch corresponding to the version of Burn you are using to follow this guide.
>
> The current version of Burn is `0.18` and the corresponding branch to checkout is `main`.

The code for this demo can be executed from Burn's base directory using the command:

```
cargo run --example guide
```

## Key Learnings

- Creating a project
- Creating neural network models
- Importing and preparing datasets
- Training models on data
- Choosing a backend
- Using a model for inference

# Model

The first step is to create a project and add the different Burn dependencies. Start by creating a new project with Cargo:

```
cargo new guide
```

As mentioned previously, this will initialize your `guide` project directory with a `Cargo.toml` and a `src/main.rs` file.

In the `Cargo.toml` file, add the `burn` dependency with `train`, `vision` and `wgpu` features. Since we disable the default features, we also want to enable `std`, `tui` (for the dashboard) and `fusion` for wgpu. Then run `cargo build` to build the project and import all the dependencies.

```
[package]
name = "guide"
version = "0.1.0"
edition = "2024"

[dependencies]
# Disable autotune default for convolutions
burn = { version = "~0.18", features = ["std", "tui", "train", "vision",
"wgpu", "fusion"], default-features = false }
# burn = { version = "~0.18", features = ["train", "vision", "wgpu"] }
```

Our goal will be to create a basic convolutional neural network used for image classification. We will keep the model simple by using two convolution layers followed by two linear layers, some pooling and ReLU activations. We will also use dropout to improve training performance.

Let us start by defining our model struct in a new file `src/model.rs`.

```rust
use burn::{
    nn::{
        conv::{Conv2d, Conv2dConfig},
        pool::{AdaptiveAvgPool2d, AdaptiveAvgPool2dConfig},
        Dropout, DropoutConfig, Linear, LinearConfig, Relu,
    },
    prelude::*,
};

#[derive(Module, Debug)]
pub struct Model<B: Backend> {
    conv1: Conv2d<B>,
    conv2: Conv2d<B>,
    pool: AdaptiveAvgPool2d,
    dropout: Dropout,
    linear1: Linear<B>,
    linear2: Linear<B>,
    activation: Relu,
}
```

There are two major things going on in this code sample.

1. You can create a deep learning module with the `#[derive(Module)]` attribute on top of a struct. This will generate the necessary code so that the struct implements the `Module` trait. This trait will make your module both trainable and (de)serializable while adding related functionalities. Like other attributes often used in Rust, such as `Clone`, `PartialEq` or `Debug`, each field within the struct must also implement the `Module` trait.

   ▸ 🦀 **Trait**

   ▸ 🦀 **Derive Macro**

2. Note that the struct is generic over the `Backend` trait. The backend trait abstracts the underlying low level implementations of tensor operations, allowing your new model to run on any backend. Contrary to other frameworks, the backend abstraction isn't determined by a compilation flag or a device type. This is important because you can extend the functionalities of a specific backend (see backend extension section), and it allows for an innovative autodiff system. You can also change backend during runtime, for instance to compute training metrics on a cpu backend while using a gpu one only to train the model. In our example, the backend in use will be determined later on.

   ▸ 🦀 **Trait Bounds**

Note that each time you create a new file in the `src` directory you also need to explicitly add this module to the `main.rs` file. For instance after creating the `model.rs`, you need to add the following at the top of the main file:

```
mod model;
```

Next, we need to instantiate the model for training.

```
#[derive(Config, Debug)]
pub struct ModelConfig {
    num_classes: usize,
    hidden_size: usize,
    #[config(default = "0.5")]
    dropout: f64,
}

impl ModelConfig {
    /// Returns the initialized model.
    pub fn init<B: Backend>(&self, device: &B::Device) -> Model<B> {
        Model {
            conv1: Conv2dConfig::new([1, 8], [3, 3]).init(device),
            conv2: Conv2dConfig::new([8, 16], [3, 3]).init(device),
            pool: AdaptiveAvgPool2dConfig::new([8, 8]).init(),
            activation: Relu::new(),
            linear1: LinearConfig::new(16 * 8 * 8,
self.hidden_size).init(device),
            linear2: LinearConfig::new(self.hidden_size,
self.num_classes).init(device),
            dropout: DropoutConfig::new(self.dropout).init(),
        }
    }
}
```

At a glance, you can view the model configuration by printing the model instance:

```
#![recursion_limit = "256"]
mod model;

use crate::model::ModelConfig;
use burn::backend::Wgpu;

fn main() {
    type MyBackend = Wgpu<f32, i32>;

    let device = Default::default();
    let model = ModelConfig::new(10, 512).init::<MyBackend>(&device);

    println!("{model}");
}
```

Output:

```
Model {
  conv1: Conv2d {stride: [1, 1], kernel_size: [3, 3], dilation: [1, 1], groups:
1, padding: Valid, params: 80}
  conv2: Conv2d {stride: [1, 1], kernel_size: [3, 3], dilation: [1, 1], groups:
1, padding: Valid, params: 1168}
  pool: AdaptiveAvgPool2d {output_size: [8, 8]}
  dropout: Dropout {prob: 0.5}
  linear1: Linear {d_input: 1024, d_output: 512, bias: true, params: 524800}
  linear2: Linear {d_input: 512, d_output: 10, bias: true, params: 5130}
  activation: Relu
  params: 531178
}
```

▸ 🦀 **References**

When creating a custom neural network module, it is often a good idea to create a config alongside the model struct. This allows you to define default values for your network, thanks to the `Config` attribute. The benefit of this attribute is that it makes the configuration serializable, enabling you to painlessly save your model hyperparameters, enhancing your experimentation process. Note that a constructor will automatically be generated for your configuration, which will take in as input values the parameters which do not have default values: `let config = ModelConfig::new(num_classes, hidden_size);`. The default values can be overridden easily with builder-like methods: (e.g `config.with_dropout(0.2);`)

The first implementation block is related to the initialization method. As we can see, all fields are set using the configuration of the corresponding neural network's underlying module. In this specific case, we have chosen to expand the tensor channels from 1 to 8 with the first layer, then from 8 to 16 with the second layer, using a kernel size of 3 on all dimensions. We also use the adaptive average pooling module to reduce the dimensionality of the images to an 8 by 8 matrix, which we will flatten in the forward pass to have a 1024 (16 * 8 * 8) resulting tensor.

Now let's see how the forward pass is defined.

```
impl<B: Backend> Model<B> {
    /// # Shapes
    ///   - Images [batch_size, height, width]
    ///   - Output [batch_size, num_classes]
    pub fn forward(&self, images: Tensor<B, 3>) -> Tensor<B, 2> {
        let [batch_size, height, width] = images.dims();

        // Create a channel at the second dimension.
        let x = images.reshape([batch_size, 1, height, width]);


        let x = self.conv1.forward(x); // [batch_size, 8, _, _]
        let x = self.dropout.forward(x);
        let x = self.conv2.forward(x); // [batch_size, 16, _, _]
        let x = self.dropout.forward(x);
        let x = self.activation.forward(x);

        let x = self.pool.forward(x); // [batch_size, 16, 8, 8]
        let x = x.reshape([batch_size, 16 * 8 * 8]);
        let x = self.linear1.forward(x);
        let x = self.dropout.forward(x);
        let x = self.activation.forward(x);

        self.linear2.forward(x) // [batch_size, num_classes]
    }
}
```

For former PyTorch users, this might feel very intuitive, as each module is directly incorporated into the code using an eager API. Note that no abstraction is imposed for the forward method. You are free to define multiple forward functions with the names of your liking. Most of the neural network modules already built with Burn use the `forward` nomenclature, simply because it is standard in the field.

Similar to neural network modules, the `Tensor` struct given as a parameter also takes the Backend trait as a generic argument, alongside its dimensionality. Even if it is not used in this specific example, it is possible to add the kind of the tensor as a third generic argument. For example, a 3-dimensional Tensor of different data types(float, int, bool) would be defined as following:

```
Tensor<B, 3> // Float tensor (default)
Tensor<B, 3, Float> // Float tensor (explicit)
Tensor<B, 3, Int> // Int tensor
Tensor<B, 3, Bool> // Bool tensor
```

Note that the specific element type, such as `f16`, `f32` and the likes, will be defined later with the backend.

# Data

Typically, one trains a model on some dataset. Burn provides a library of very useful dataset sources and transformations, such as Hugging Face dataset utilities that allow to download and store data into an SQLite database for extremely efficient data streaming and storage. For this guide though, we will use the MNIST dataset from `burn::data::dataset::vision` which requires no external dependency.

To iterate over a dataset efficiently, we will define a struct which will implement the `Batcher` trait. The goal of a batcher is to map individual dataset items into a batched tensor that can be used as input to our previously defined model.

Let us start by defining our dataset functionalities in a file `src/data.rs` . We shall omit some of the imports for brevity, but the full code for following this guide can be found at `examples/guide/` directory.

```
use burn::{
    data::{dataloader::batcher::Batcher, dataset::vision::MnistItem},
    prelude::*,
};


#[derive(Clone, Default)]
pub struct MnistBatcher {}
```

This batcher is pretty straightforward, as it only defines a struct that will implement the `Batcher` trait. The trait is generic over the `Backend` trait, which includes an associated type for the device, as not all backends expose the same devices. As an example, the Libtorch-based backend exposes `Cuda(gpu_index)` , `Cpu` , `Vulkan` and `Metal` devices, while the ndarray backend only exposes the `Cpu` device.

Next, we need to actually implement the batching logic.

```rust
#[derive(Clone, Debug)]
pub struct MnistBatch<B: Backend> {
    pub images: Tensor<B, 3>,
    pub targets: Tensor<B, 1, Int>,
}

impl<B: Backend> Batcher<B, MnistItem, MnistBatch<B>> for MnistBatcher {
    fn batch(&self, items: Vec<MnistItem>, device: &B::Device) -> MnistBatch<B>
{
        let images = items
            .iter()
            .map(|item| TensorData::from(item.image).convert::<B::FloatElem>())
            .map(|data| Tensor::<B, 2>::from_data(data, device))
            .map(|tensor| tensor.reshape([1, 28, 28]))
            // Normalize: scale between [0,1] and make the mean=0 and std=1
            // values mean=0.1307,std=0.3081 are from the PyTorch MNIST example
            // https://github.com/pytorch/examples/
blob/54f4572509891883a947411fd7239237dd2a39c3/mnist/main.py#L122
            .map(|tensor| ((tensor / 255) - 0.1307) / 0.3081)
            .collect();

        let targets = items
            .iter()
            .map(|item| {
                Tensor::<B, 1, Int>::from_data([(item.label as
i64).elem::<B::IntElem>()], device)
            })
            .collect();

        let images = Tensor::cat(images, 0);
        let targets = Tensor::cat(targets, 0);

        MnistBatch { images, targets }
    }
}
```

### ▸ 🦀 Iterators and Closures

In the previous example, we implement the `Batcher` trait with a list of `MnistItem` as input and a single `MnistBatch` as output. The batch contains the images in the form of a 3D tensor, along with a targets tensor that contains the indexes of the correct digit class. The first step is to parse the image array into a `TensorData` struct. Burn provides the `TensorData` struct to encapsulate tensor storage information without being specific for a backend. When creating a tensor from data, we often need to convert the data precision to the current backend in use. This can be done with the `.convert()` method (in this example, the data is converted backend's float element type `B::FloatElem`). While importing the `burn::tensor::ElementConversion` trait, you can call `.elem()` on a specific number to

convert it to the current backend element type in use.

# Training

We are now ready to write the necessary code to train our model on the MNIST dataset. We shall define the code for this training section in the file: `src/training.rs`.

Instead of a simple tensor, the model should output an item that can be understood by the learner, a struct whose responsibility is to apply an optimizer to the model. The output struct is used for all metrics calculated during the training. Therefore it should include all the necessary information to calculate any metric that you want for a task.

Burn provides two basic output types: `ClassificationOutput` and `RegressionOutput`. They implement the necessary trait to be used with metrics. It is possible to create your own item, but it is beyond the scope of this guide.

Since the MNIST task is a classification problem, we will use the `ClassificationOutput` type.

```
impl<B: Backend> Model<B> {
    pub fn forward_classification(
        &self,
        images: Tensor<B, 3>,
        targets: Tensor<B, 1, Int>,
    ) -> ClassificationOutput<B> {
        let output = self.forward(images);
        let loss = CrossEntropyLossConfig::new()
            .init(&output.device())
            .forward(output.clone(), targets.clone());

        ClassificationOutput::new(loss, output, targets)
    }
}
```

As evident from the preceding code block, we employ the cross-entropy loss module for loss calculation, without the inclusion of any padding token. We then return the classification output containing the loss, the output tensor with all logits and the targets.

Please take note that tensor operations receive owned tensors as input. For reusing a tensor multiple times, you need to use the `clone()` function. There's no need to worry; this process won't involve actual copying of the tensor data. Instead, it will simply indicate that the tensor is employed in multiple instances, implying that certain operations won't be performed in place. In summary, our API has been designed with owned tensors to optimize performance.

Moving forward, we will proceed with the implementation of both the training and validation

steps for our model.

```
impl<B: AutodiffBackend> TrainStep<MnistBatch<B>, ClassificationOutput<B>> for
Model<B> {
    fn step(&self, batch: MnistBatch<B>) ->
TrainOutput<ClassificationOutput<B>> {
        let item = self.forward_classification(batch.images, batch.targets);

        TrainOutput::new(self, item.loss.backward(), item)
    }
}

impl<B: Backend> ValidStep<MnistBatch<B>, ClassificationOutput<B>> for Model<B>
{
    fn step(&self, batch: MnistBatch<B>) -> ClassificationOutput<B> {
        self.forward_classification(batch.images, batch.targets)
    }
}
```

Here we define the input and output types as generic arguments in the `TrainStep` and `ValidStep`. We will call them `MnistBatch` and `ClassificationOutput`. In the training step, the computation of gradients is straightforward, necessitating a simple invocation of `backward()` on the loss. Note that contrary to PyTorch, gradients are not stored alongside each tensor parameter, but are rather returned by the backward pass, as such: `let gradients = loss.backward();`. The gradient of a parameter can be obtained with the grad function: `let grad = tensor.grad(&gradients);`. Although it is not necessary when using the learner struct and the optimizers, it can prove to be quite useful when debugging or writing custom training loops. One of the differences between the training and the validation steps is that the former requires the backend to implement `AutodiffBackend` and not just `Backend`. Otherwise, the `backward` function is not available, as the backend does not support autodiff. We will see later how to create a backend with autodiff support.

▸ 🦀 **Generic Type Constraints in Method Definitions**

Let us move on to establishing the practical training configuration.

```rust
#[derive(Config)]
pub struct TrainingConfig {
    pub model: ModelConfig,
    pub optimizer: AdamConfig,
    #[config(default = 10)]
    pub num_epochs: usize,
    #[config(default = 64)]
    pub batch_size: usize,
    #[config(default = 4)]
    pub num_workers: usize,
    #[config(default = 42)]
    pub seed: u64,
    #[config(default = 1.0e-4)]
    pub learning_rate: f64,
}

fn create_artifact_dir(artifact_dir: &str) {
    // Remove existing artifacts before to get an accurate learner summary
    std::fs::remove_dir_all(artifact_dir).ok();
    std::fs::create_dir_all(artifact_dir).ok();
}

pub fn train<B: AutodiffBackend>(artifact_dir: &str, config: TrainingConfig,
device: B::Device) {
    create_artifact_dir(artifact_dir);
    config
        .save(format!("{artifact_dir}/config.json"))
        .expect("Config should be saved successfully");

    B::seed(config.seed);

    let batcher = MnistBatcher::default();

    let dataloader_train = DataLoaderBuilder::new(batcher.clone())
        .batch_size(config.batch_size)
        .shuffle(config.seed)
        .num_workers(config.num_workers)
        .build(MnistDataset::train());

    let dataloader_test = DataLoaderBuilder::new(batcher)
        .batch_size(config.batch_size)
        .shuffle(config.seed)
        .num_workers(config.num_workers)
        .build(MnistDataset::test());

    let learner = LearnerBuilder::new(artifact_dir)
        .metric_train_numeric(AccuracyMetric::new())
        .metric_valid_numeric(AccuracyMetric::new())
        .metric_train_numeric(LossMetric::new())
        .metric_valid_numeric(LossMetric::new())
        .with_file_checkpointer(CompactRecorder::new())
        .devices(vec![device.clone()])
```

```
            .num_epochs(config.num_epochs)
            .summary()
            .build(
                config.model.init::<B>(&device),
                config.optimizer.init(),
                config.learning_rate,
            );

        let model_trained = learner.fit(dataloader_train, dataloader_test);

        model_trained
            .save_file(format!("{artifact_dir}/model"), &CompactRecorder::new())
            .expect("Trained model should be saved successfully");
    }
```

It is a good practice to use the `Config` derive to create the experiment configuration. In the `train` function, the first thing we are doing is making sure the `artifact_dir` exists, using the standard rust library for file manipulation. All checkpoints, logging and metrics will be stored under this directory. We initialize the dataloaders using the previously created batcher. Since no automatic differentiation is needed during the validation phase, the `learner.fit(...)` method defines the necessary backend bounds on the data loader for `B::InnerBackend` (see Backend). The autodiff capabilities are available through a type system, making it nearly impossible to forget to deactivate gradient calculation.

Next, we create our learner with the accuracy and loss metric on both training and validation steps along with the device and the epoch. We also configure the checkpointer using the `CompactRecorder` to indicate how weights should be stored. This struct implements the `Recorder` trait, which makes it capable of saving records for persistency.

We then build the learner with the model, the optimizer and the learning rate. Notably, the third argument of the build function should actually be a learning rate *scheduler*. When provided with a float as in our example, it is automatically transformed into a *constant* learning rate scheduler. The learning rate is not part of the optimizer config as it is often done in other frameworks, but rather passed as a parameter when executing the optimizer step. This avoids having to mutate the state of the optimizer and is therefore more functional. It makes no difference when using the learner struct, but it will be an essential nuance to grasp if you implement your own training loop.

Once the learner is created, we can simply call `fit` and provide the training and validation dataloaders. For the sake of simplicity in this example, we employ the test set as the validation set; however, we do not recommend this practice for actual usage.

Finally, the trained model is returned by the `fit` method. The trained weights are then saved using the `CompactRecorder`. This recorder employs the `MessagePack` format with half precision, `f16` for floats and `i16` for integers. Other recorders are available, offering

support for various formats, such as `BinCode` and `JSON`, with or without compression. Any backend, regardless of precision, can load recorded data of any kind.

# Backend

We have effectively written most of the necessary code to train our model. However, we have not explicitly designated the backend to be used at any point. This will be defined in the main entrypoint of our program, namely the `main` function defined in `src/main.rs`.

```
use crate::{model::ModelConfig, training::TrainingConfig};
use burn::{
    backend::{Autodiff, Wgpu},
    optim::AdamConfig,
};

fn main() {
    type MyBackend = Wgpu<f32, i32>;
    type MyAutodiffBackend = Autodiff<MyBackend>;

    let device = burn::backend::wgpu::WgpuDevice::default();
    let artifact_dir = "/tmp/guide";
    crate::training::train::<MyAutodiffBackend>(
        artifact_dir,
        TrainingConfig::new(ModelConfig::new(10, 512), AdamConfig::new()),
        device.clone(),
    );
}
```

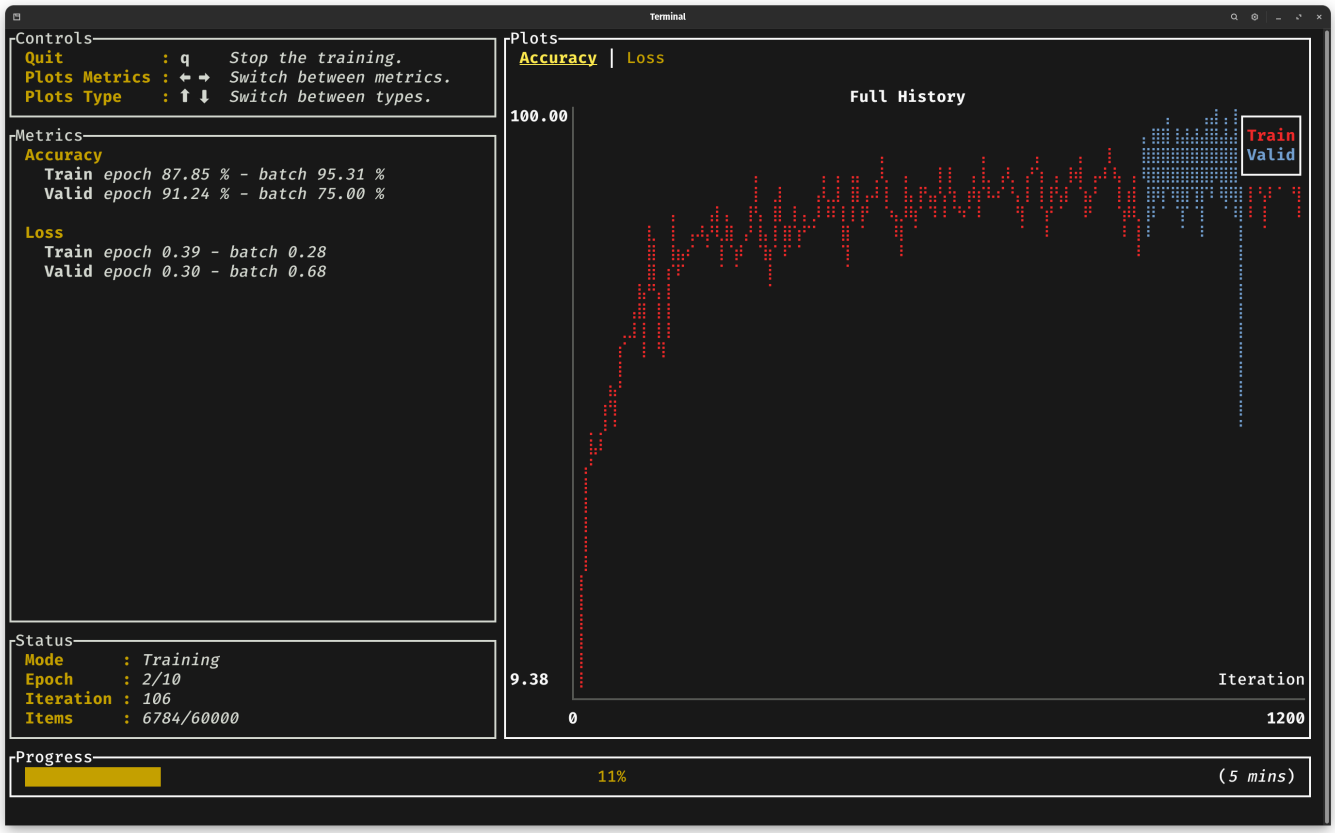
In this code snippet, we use the `Wgpu` backend which is compatible with any operating system and will use the GPU. For other options, see the Burn README. This backend type takes the graphics API, the float type and the int type as generic arguments that will be used during the training. The autodiff backend is simply the same backend, wrapped within the `Autodiff` struct which imparts differentiability to any backend.

We call the `train` function defined earlier with a directory for artifacts, the configuration of the model (the number of digit classes is 10 and the hidden dimension is 512), the optimizer configuration which in our case will be the default Adam configuration, and the device which can be obtained from the backend.

You can now train your freshly created model with the command:

```
cargo run --release
```

When running your project with the command above, you should see the training progression through a basic CLI dashboard:


Terminal
Controls
Quit : q Stop the training.
Plots Metrics : ← → Switch between metrics.
Plots Type : ↑ ↓ Switch between types.
Metrics
Accuracy
Train epoch 87.85 % - batch 95.31 %
Valid epoch 91.24 % - batch 75.00 %
Loss
Train epoch 0.39 - batch 0.28
Valid epoch 0.30 - batch 0.68
Status
Mode : Training
Epoch : 2/10
Iteration : 106
Items : 6784/60000
Plots
Accuracy | Loss
Full History
100.00
9.38
Train
Valid
Iteration
0
1200
Progress
11%
(5 mins)

# Inference

Now that we have trained our model, the next natural step is to use it for inference.

You need two things in order to load weights for a model: the model's record and the model's config. Since parameters in Burn are lazy initialized, no allocation and GPU/CPU kernels are executed by the `ModelConfig::init` function. The weights are initialized when used for the first time, therefore you can safely use `config.init(device).load_record(record)` without any meaningful performance cost. Let's create a simple `infer` method in a new file `src/inference.rs` which we will use to load our trained model.

```
pub fn infer<B: Backend>(artifact_dir: &str, device: B::Device, item:
MnistItem) {
    let config = TrainingConfig::load(format!("{artifact_dir}/config.json"))
        .expect("Config should exist for the model; run train first");
    let record = CompactRecorder::new()
        .load(format!("{artifact_dir}/model").into(), &device)
        .expect("Trained model should exist; run train first");

    let model = config.model.init::<B>(&device).load_record(record);

    let label = item.label;
    let batcher = MnistBatcher::default();
    let batch = batcher.batch(vec![item], &device);
    let output = model.forward(batch.images);
    let predicted = output.argmax(1).flatten::<1>(0, 1).into_scalar();

    println!("Predicted {predicted} Expected {label}");
}
```

The first step is to load the configuration of the training to fetch the correct model configuration. Then we can fetch the record using the same recorder as we used during training. Finally we can init the model with the configuration and the record. For simplicity we can use the same batcher used during the training to pass from a MnistItem to a tensor.

By running the infer function, you should see the predictions of your model!

Add the call to `infer` to the `main.rs` file after the `train` function call:

```
    crate::inference::infer::<MyBackend>(
        artifact_dir,
        device,
        burn::data::dataset::vision::MnistDataset::test()
            .get(42)
            .unwrap(),
    );
```

The number `42` is the index of the image in the MNIST dataset. You can explore and verify them using this MNIST viewer.

# Conclusion

In this short guide, we've introduced you to the fundamental building blocks for getting started with Burn. While there's still plenty to explore, our goal has been to provide you with the essential knowledge to kickstart your productivity within the framework.

# Building Blocks

In this section, we'll guide you through the core elements that make up Burn. We'll walk you through the key components that serve as the building blocks of the framework and your future projects.

As you explore Burn, you might notice that we occasionally draw comparisons to PyTorch. We believe it can provide a smoother learning curve and help you grasp the nuances more effectively.

# Backend

Nearly everything in Burn is based on the `Backend` trait, which enables you to run tensor operations using different implementations without having to modify your code. While a backend may not necessarily have autodiff capabilities, the `AutodiffBackend` trait specifies when autodiff is needed. This trait not only abstracts operations but also tensor, device, and element types, providing each backend the flexibility they need. It's worth noting that the trait assumes eager mode since burn fully supports dynamic graphs. However, we may create another API to assist with integrating graph-based backends, without requiring any changes to the user's code.

Users are not expected to directly use the backend trait methods, as it is primarily designed with backend developers in mind rather than Burn users. Therefore, most Burn userland APIs are generic across backends. This approach helps users discover the API more organically with proper autocomplete and documentation.

# Tensor

As previously explained in the model section, the Tensor struct has 3 generic arguments: the backend B, the dimensionality D, and the data type.

```
Tensor<B, D>           // Float tensor (default)
Tensor<B, D, Float>    // Explicit float tensor
Tensor<B, D, Int>      // Int tensor
Tensor<B, D, Bool>     // Bool tensor
```

Note that the specific element types used for `Float`, `Int`, and `Bool` tensors are defined by backend implementations.

Burn Tensors are defined by the number of dimensions D in its declaration as opposed to its shape. The actual shape of the tensor is inferred from its initialization. For example, a Tensor of size (5,) is initialized as below:

```
let floats = [1.0, 2.0, 3.0, 4.0, 5.0];

// Get the default device
let device = Default::default();

// correct: Tensor is 1-Dimensional with 5 elements
let tensor_1 = Tensor::<Backend, 1>::from_floats(floats, &device);

// incorrect: let tensor_1 = Tensor::<Backend, 5>::from_floats(floats,
&device);
// this will lead to an error and is for creating a 5-D tensor
```

## Initialization

Burn Tensors are primarily initialized using the `from_data()` method which takes the `TensorData` struct as input. The `TensorData` struct has two public fields: `shape` and `dtype`. The `value`, now stored as bytes, is private but can be accessed via any of the following methods: `as_slice`, `as_mut_slice`, `to_vec` and `iter`. To retrieve the data from a tensor, the method `.to_data()` should be employed when intending to reuse the tensor afterward. Alternatively, `.into_data()` is recommended for one-time use. Let's look at a couple of examples for initializing a tensor from different inputs.

```
// Initialization from a given Backend (Wgpu)
let tensor_1 = Tensor::<Wgpu, 1>::from_data([1.0, 2.0, 3.0], &device);

// Initialization from a generic Backend
let tensor_2 = Tensor::<Backend, 1>::from_data(TensorData::from([1.0, 2.0,
3.0]), &device);

// Initialization using from_floats (Recommended for f32 ElementType)
// Will be converted to TensorData internally.
let tensor_3 = Tensor::<Backend, 1>::from_floats([1.0, 2.0, 3.0], &device);

// Initialization of Int Tensor from array slices
let arr: [i32; 6] = [1, 2, 3, 4, 5, 6];
let tensor_4 = Tensor::<Backend, 1,
Int>::from_data(TensorData::from(&arr[0..3]), &device);

// Initialization from a custom type

struct BodyMetrics {
    age: i8,
    height: i16,
    weight: f32
}

let bmi = BodyMetrics{
        age: 25,
        height: 180,
        weight: 80.0
    };
let data  = TensorData::from([bmi.age as f32, bmi.height as f32, bmi.weight]);
let tensor_5 = Tensor::<Backend, 1>::from_data(data, &device);
```

## Ownership and Cloning

Almost all Burn operations take ownership of the input tensors. Therefore, reusing a tensor multiple times will necessitate cloning it. Let's look at an example to understand the ownership rules and cloning better. Suppose we want to do a simple min-max normalization of an input tensor.

```
let input = Tensor::<Wgpu, 1>::from_floats([1.0, 2.0, 3.0, 4.0], &device);
let min = input.min();
let max = input.max();
let input = (input - min).div(max - min);
```

With PyTorch tensors, the above code would work as expected. However, Rust's strict

ownership rules will give an error and prevent using the input tensor after the first `.min()` operation. The ownership of the input tensor is transferred to the variable `min` and the input tensor is no longer available for further operations. Burn Tensors like most complex primitives do not implement the `Copy` trait and therefore have to be cloned explicitly. Now let's rewrite a working example of doing min-max normalization with cloning.

```
let input = Tensor::<Wgpu, 1>::from_floats([1.0, 2.0, 3.0, 4.0], &device);
let min = input.clone().min();
let max = input.clone().max();
let input = (input.clone() - min.clone()).div(max - min);
println!("{}", input.to_data());// Success: [0.0, 0.33333334, 0.6666667, 1.0]

// Notice that max, min have been moved in last operation so
// the below print will give an error.
// If we want to use them for further operations,
// they will need to be cloned in similar fashion.
// println!("{:?}", min.to_data());
```

We don't need to be worried about memory overhead because with cloning, the tensor's buffer isn't copied, and only a reference to it is increased. This makes it possible to determine exactly how many times a tensor is used, which is very convenient for reusing tensor buffers or even fusing operations into a single kernel (burn-fusion). For that reason, we don't provide explicit inplace operations. If a tensor is used only one time, inplace operations will always be used when available.

# Tensor Operations

Normally with PyTorch, explicit inplace operations aren't supported during the backward pass, making them useful only for data preprocessing or inference-only model implementations. With Burn, you can focus more on *what* the model should do, rather than on *how* to do it. We take the responsibility of making your code run as fast as possible during training as well as inference. The same principles apply to broadcasting; all operations support broadcasting unless specified otherwise.

Here, we provide a list of all supported operations along with their PyTorch equivalents. Note that for the sake of simplicity, we ignore type signatures. For more details, refer to the full documentation.

## Basic Operations

Those operations are available for all tensor kinds: `Int`, `Float`, and `Bool`.

| Burn | PyTorch Equivalent |
|---|---|
| `Tensor::cat(tensors, dim)` | `torch.cat(tensors, dim)` |
| `Tensor::empty(shape, device)` | `torch.empty(shape, device=device)` |
| `Tensor::from_primitive(primitive)` | N/A |
| `Tensor::stack(tensors, dim)` | `torch.stack(tensors, dim)` |
| `tensor.all()` | `tensor.all()` |
| `tensor.all_dim(dim)` | `tensor.all(dim)` |
| `tensor.any()` | `tensor.any()` |
| `tensor.any_dim(dim)` | `tensor.any(dim)` |
| `tensor.chunk(num_chunks, dim)` | `tensor.chunk(num_chunks, dim)` |
| `tensor.split(split_size, dim)` | `tensor.split(split_size, dim)` |
| `tensor.split_with_sizes(split_sizes, dim)` | `tensor.split([split_sizes], dim)` |
| `tensor.device()` | `tensor.device` |
| `tensor.dtype()` | `tensor.dtype` |
| `tensor.dims()` | `tensor.size()` |
| `tensor.equal(other)` | `x == y` |
| `tensor.expand(shape)` | `tensor.expand(shape)` |
| `tensor.flatten(start_dim, end_dim)` | `tensor.flatten(start_dim, end_dim)` |
| `tensor.flip(axes)` | `tensor.flip(axes)` |
| `tensor.into_data()` | N/A |
| `tensor.into_primitive()` | N/A |
| `tensor.into_scalar()` | `tensor.item()` |
| `tensor.narrow(dim, start, length)` | `tensor.narrow(dim, start, length)` |
| `tensor.not_equal(other)` | `x != y` |
| `tensor.permute(axes)` | `tensor.permute(axes)` |
| `tensor.movedim(src, dst)` | `tensor.movedim(src, dst)` |
| `tensor.repeat_dim(dim, times)` | `tensor.repeat(*[times if i == dim else 1 for i in range(tensor.dim())])` |
| `tensor.repeat(sizes)` | `tensor.repeat(sizes)` |

| Burn | PyTorch Equivalent |
|---|---|
| `tensor.reshape(shape)` | `tensor.view(shape)` |
| `tensor.roll(shfts, dims)` | `tensor.roll(shifts, dims)` |
| `tensor.roll_dim(shift, dim)` | `tensor.roll([shift], [dim])` |
| `tensor.shape()` | `tensor.shape` |
| `tensor.slice(ranges)` | `tensor[(*ranges,)]` |
| `tensor.slice_assign(ranges, values)` | `tensor[(*ranges,)] = values` |
| `tensor.slice_fill(ranges, value)` | `tensor[(*ranges,)] = value` |
| `tensor.slice_dim(dim, range)` | N/A |
| `tensor.squeeze(dim)` | `tensor.squeeze(dim)` |
| `tensor.swap_dims(dim1, dim2)` | `tensor.transpose(dim1, dim2)` |
| `tensor.to_data()` | N/A |
| `tensor.to_device(device)` | `tensor.to(device)` |
| `tensor.transpose()` | `tensor.T` |
| `tensor.unsqueeze()` | `tensor.unsqueeze(0)` |
| `tensor.unsqueeze_dim(dim)` | `tensor.unsqueeze(dim)` |
| `tensor.unsqueeze_dims(dims)` | N/A |

## Numeric Operations

Those operations are available for numeric tensor kinds: `Float` and `Int`.

| Burn | PyTorch Equivalent |
|---|---|
| `Tensor::eye(size, device)` | `torch.eye(size, device=device)` |
| `Tensor::full(shape, fill_value, device)` | `torch.full(shape, fill_value, device=device)` |
| `Tensor::ones(shape, device)` | `torch.ones(shape, device=device)` |
| `Tensor::zeros(shape, device)` | `torch.zeros(shape, device=device)` |
| `tensor.abs()` | `torch.abs(tensor)` |
| `tensor.add(other)` or `tensor + other` | `tensor + other` |

| Burn | PyTorch Equivalent |
|---|---|
| `tensor.add_scalar(scalar)` or `tensor + scalar` | `tensor + scalar` |
| `tensor.all_close(other, atol, rtol)` | `torch.allclose(tensor, other, atol, rtol)` |
| `tensor.argmax(dim)` | `tensor.argmax(dim)` |
| `tensor.argmin(dim)` | `tensor.argmin(dim)` |
| `tensor.argsort(dim)` | `tensor.argsort(dim)` |
| `tensor.argsort_descending(dim)` | `tensor.argsort(dim, descending=True)` |
| `tensor.bool()` | `tensor.bool()` |
| `tensor.clamp(min, max)` | `torch.clamp(tensor, min=min, max=max)` |
| `tensor.clamp_max(max)` | `torch.clamp(tensor, max=max)` |
| `tensor.clamp_min(min)` | `torch.clamp(tensor, min=min)` |
| `tensor.contains_nan()` | N/A |
| `tensor.div(other)` or `tensor / other` | `tensor / other` |
| `tensor.div_scalar(scalar)` or `tensor / scalar` | `tensor / scalar` |
| `tensor.equal_elem(other)` | `tensor.eq(other)` |
| `tensor.full_like(fill_value)` | `torch.full_like(tensor, fill_value)` |
| `tensor.gather(dim, indices)` | `torch.gather(tensor, dim, indices)` |
| `tensor.greater(other)` | `tensor.gt(other)` |
| `tensor.greater_elem(scalar)` | `tensor.gt(scalar)` |
| `tensor.greater_equal(other)` | `tensor.ge(other)` |
| `tensor.greater_equal_elem(scalar)` | `tensor.ge(scalar)` |
| `tensor.lower(other)` | `tensor.lt(other)` |
| `tensor.lower_elem(scalar)` | `tensor.lt(scalar)` |
| `tensor.lower_equal(other)` | `tensor.le(other)` |
| `tensor.lower_equal_elem(scalar)` | `tensor.le(scalar)` |
| `tensor.mask_fill(mask, value)` | `tensor.masked_fill(mask, value)` |

| Burn | PyTorch Equivalent |
|---|---|
| `tensor.mask_where(mask, value_tensor)` | `torch.where(mask, value_tensor, tensor)` |
| `tensor.max()` | `tensor.max()` |
| `tensor.max_abs()` | `tensor.abs().max()` |
| `tensor.max_abs_dim(dim)` | `tensor.abs().max(dim, keepdim=True)` |
| `tensor.max_dim(dim)` | `tensor.max(dim, keepdim=True)` |
| `tensor.max_dim_with_indices(dim)` | N/A |
| `tensor.max_pair(other)` | `torch.Tensor.max(a,b)` |
| `tensor.mean()` | `tensor.mean()` |
| `tensor.mean_dim(dim)` | `tensor.mean(dim, keepdim=True)` |
| `tensor.min()` | `tensor.min()` |
| `tensor.min_dim(dim)` | `tensor.min(dim, keepdim=True)` |
| `tensor.min_dim_with_indices(dim)` | N/A |
| `tensor.min_pair(other)` | `torch.Tensor.min(a,b)` |
| `tensor.mul(other)` or `tensor * other` | `tensor * other` |
| `tensor.mul_scalar(scalar)` or `tensor * scalar` | `tensor * scalar` |
| `tensor.neg()` or `-tensor` | `-tensor` |
| `tensor.not_equal_elem(scalar)` | `tensor.ne(scalar)` |
| `tensor.ones_like()` | `torch.ones_like(tensor)` |
| `tensor.one_hot(num_classes)` | `torch.nn.functional.one_hot` |
| `tensor.one_hot_fill(num_classes, on_value, off_value, axis)` | N/A |
| `tensor.pad(pads, value)` | `torch.nn.functional.pad(input, pad, value)` |
| `tensor.powf(other)` or `tensor.powi(intother)` | `tensor.pow(other)` |
| `tensor.powf_scalar(scalar)` or `tensor.powi_scalar(intscalar)` | `tensor.pow(scalar)` |
| `tensor.prod()` | `tensor.prod()` |

| Burn | PyTorch Equivalent |
|---|---|
| `tensor.prod_dim(dim)` | `tensor.prod(dim, keepdim=True)` |
| `tensor.rem(other)` or `tensor % other` | `tensor % other` |
| `tensor.scatter(dim, indices, values)` | `tensor.scatter_add(dim, indices, values)` |
| `tensor.select(dim, indices)` | `tensor.index_select(dim, indices)` |
| `tensor.select_assign(dim, indices, values)` | N/A |
| `tensor.sign()` | `tensor.sign()` |
| `tensor.sort(dim)` | `tensor.sort(dim).values` |
| `tensor.sort_descending(dim)` | `tensor.sort(dim, descending=True).values` |
| `tensor.sort_descending_with_indices(dim)` | `tensor.sort(dim, descending=True)` |
| `tensor.sort_with_indices(dim)` | `tensor.sort(dim)` |
| `tensor.sub(other)` or `tensor - other` | `tensor - other` |
| `tensor.sub_scalar(scalar)` or `tensor - scalar` | `tensor - scalar` |
| `scalar - tensor` | `scalar - tensor` |
| `tensor.sum()` | `tensor.sum()` |
| `tensor.sum_dim(dim)` | `tensor.sum(dim, keepdim=True)` |
| `tensor.topk(k, dim)` | `tensor.topk(k, dim).values` |
| `tensor.topk_with_indices(k, dim)` | `tensor.topk(k, dim)` |
| `tensor.tril(diagonal)` | `torch.tril(tensor, diagonal)` |
| `tensor.triu(diagonal)` | `torch.triu(tensor, diagonal)` |
| `tensor.zeros_like()` | `torch.zeros_like(tensor)` |

## Float Operations

Those operations are only available for `Float` tensors.

| Burn API | PyTorch Equivalent |
|---|---|

| Burn API | PyTorch Equivalent |
|---|---|
| `tensor.cast(dtype)` | `tensor.to(dtype)` |
| `tensor.ceil()` | `tensor.ceil()` |
| `tensor.cos()` | `tensor.cos()` |
| `tensor.cosh()` | `tensor.cosh()` |
| `tensor.erf()` | `tensor.erf()` |
| `tensor.exp()` | `tensor.exp()` |
| `tensor.floor()` | `tensor.floor()` |
| `tensor.from_floats(floats, device)` | N/A |
| `tensor.from_full_precision(tensor)` | N/A |
| `tensor.int()` | Similar to `tensor.to(torch.long)` |
| `tensor.is_close(other, atol, rtol)` | `torch.isclose(tensor, other, atol, rtol)` |
| `tensor.is_finite()` | `torch.isfinite(tensor)` |
| `tensor.is_inf()` | `torch.isinf(tensor)` |
| `tensor.is_nan()` | `torch.isnan(tensor)` |
| `tensor.log()` | `tensor.log()` |
| `tensor.log1p()` | `tensor.log1p()` |
| `tensor.matmul(other)` | `tensor.matmul(other)` |
| `tensor.random(shape, distribution, device)` | N/A |
| `tensor.random_like(distribution)` | `torch.rand_like()` only uniform |
| `tensor.recip()` or `1.0 / tensor` | `tensor.reciprocal()` or `1.0 / tensor` |
| `tensor.round()` | `tensor.round()` |
| `tensor.sin()` | `tensor.sin()` |
| `tensor.sinh()` | `tensor.sinh()` |
| `tensor.sqrt()` | `tensor.sqrt()` |
| `tensor.tan()` | `tensor.tan()` |
| `tensor.tanh()` | `tensor.tanh()` |
| `tensor.to_full_precision()` | `tensor.to(torch.float)` |
| `tensor.var(dim)` | `tensor.var(dim)` |
| `tensor.var_bias(dim)` | N/A |

| Burn API | PyTorch Equivalent |
|---|---|
| `tensor.var_mean(dim)` | N/A |
| `tensor.var_mean_bias(dim)` | N/A |

## Int Operations

Those operations are only available for `Int` tensors.

| Burn API | PyTorch Equivalent |
|---|---|
| `Tensor::arange(5..10, device)` | `tensor.arange(start=5, end=10, device=device)` |
| `Tensor::arange_step(5..10, 2, device)` | `tensor.arange(start=5, end=10, step=2, device=device)` |
| `tensor.bitwise_and(other)` | `torch.bitwise_and(tensor, other)` |
| `tensor.bitwise_and_scalar(scalar)` | `torch.bitwise_and(tensor, scalar)` |
| `tensor.bitwise_not()` | `torch.bitwise_not(tensor)` |
| `tensor.bitwise_left_shift(other)` | `torch.bitwise_left_shift(tensor, other)` |
| `tensor.bitwise_left_shift_scalar(scalar)` | `torch.bitwise_left_shift(tensor, scalar)` |
| `tensor.bitwise_right_shift(other)` | `torch.bitwise_right_shift(tensor, other)` |
| `tensor.bitwise_right_shift_scalar(scalar)` | `torch.bitwise_right_shift(tensor, scalar)` |
| `tensor.bitwise_or(other)` | `torch.bitwise_or(tensor, other)` |
| `tensor.bitwise_or_scalar(scalar)` | `torch.bitwise_or(tensor, scalar)` |
| `tensor.bitwise_xor(other)` | `torch.bitwise_xor(tensor, other)` |
| `tensor.bitwise_xor_scalar(scalar)` | `torch.bitwise_xor(tensor, scalar)` |
| `tensor.float()` | `tensor.to(torch.float)` |
| `tensor.from_ints(ints)` | N/A |
| `tensor.int_random(shape, distribution, device)` | N/A |
| `tensor.cartesian_grid(shape, device)` | N/A |

### Bool Operations

Those operations are only available for `Bool` tensors.

| Burn API | PyTorch Equivalent |
|---|---|
| `Tensor::diag_mask(shape, diagonal)` | N/A |
| `Tensor::tril_mask(shape, diagonal)` | N/A |
| `Tensor::triu_mask(shape, diagonal)` | N/A |
| `tensor.argwhere()` | `tensor.argwhere()` |
| `tensor.bool_and()` | `tensor.logical_and()` |
| `tensor.bool_not()` | `tensor.logical_not()` |
| `tensor.bool_or()` | `tensor.logical_or()` |
| `tensor.float()` | `tensor.to(torch.float)` |
| `tensor.int()` | `tensor.to(torch.long)` |
| `tensor.nonzero()` | `tensor.nonzero(as_tuple=True)` |

### Quantization Operations

Those operations are only available for `Float` tensors on backends that implement quantization strategies.

| Burn API | PyTorch Equivalent |
|---|---|
| `tensor.quantize(scheme, qparams)` | N/A |
| `tensor.dequantize()` | N/A |

## Activation Functions

| Burn API | PyTorch Equivalent |
|---|---|
| `activation::gelu(tensor)` | `nn.functional.gelu(tensor)` |
| `activation::hard_sigmoid(tensor, alpha, beta)` | `nn.functional.hardsigmoid(tensor)` |
| `activation::leaky_relu(tensor, negative_slope)` | `nn.functional.leaky_relu(tensor, negative_slope)` |
| `activation::log_sigmoid(tensor)` | `nn.functional.log_sigmoid(tensor)` |

| Burn API | PyTorch Equivalent |
|---|---|
| `activation::log_softmax(tensor, dim)` | `nn.functional.log_softmax(tensor, dim)` |
| `activation::mish(tensor)` | `nn.functional.mish(tensor)` |
| `activation::prelu(tensor,alpha)` | `nn.functional.prelu(tensor,weight)` |
| `activation::quiet_softmax(tensor, dim)` | `nn.functional.quiet_softmax(tensor, dim)` |
| `activation::relu(tensor)` | `nn.functional.relu(tensor)` |
| `activation::sigmoid(tensor)` | `nn.functional.sigmoid(tensor)` |
| `activation::silu(tensor)` | `nn.functional.silu(tensor)` |
| `activation::softmax(tensor, dim)` | `nn.functional.softmax(tensor, dim)` |
| `activation::softmin(tensor, dim)` | `nn.functional.softmin(tensor, dim)` |
| `activation::softplus(tensor, beta)` | `nn.functional.softplus(tensor, beta)` |
| `activation::tanh(tensor)` | `nn.functional.tanh(tensor)` |

## Grid Functions

| Burn API | PyTorch Equivalent |
|---|---|
| `grid::meshgrid(tensors, GridIndexing::Matrix)` | `torch.meshgrid(tensors, indexing="ij") |
| `grid::meshgrid(tensors, GridIndexing::Cartesian)` | `torch.meshgrid(tensors, indexing="xy") |

## Linalg Functions

| Burn API | PyTorch Equivalent |
|---|---|
| `linalg::vector_norm(tensors, p, dim)` | `torch.linalg.vector_norm(tensor, p, dim) |

## Displaying Tensor Details

Burn provides flexible options for displaying tensor information, allowing you to control the level of detail and formatting to suit your needs.

## Basic Display

To display a detailed view of a tensor, you can simply use Rust's `println!` or `format!` macros:

```
let tensor = Tensor::<Backend, 2>::full([2, 3], 0.123456789,
&Default::default());
println!("{}", tensor);
```

This will output:

```
Tensor {
  data:
[[0.12345679, 0.12345679, 0.12345679],
 [0.12345679, 0.12345679, 0.12345679]],
  shape:  [2, 3],
  device:  Cpu,
  backend:  "ndarray",
  kind:  "Float",
  dtype:  "f32",
}
```

## Controlling Precision

You can control the number of decimal places displayed using Rust's formatting syntax:

```
println!("{:.2}", tensor);
```

Output:

```
Tensor {
  data:
[[0.12, 0.12, 0.12],
 [0.12, 0.12, 0.12]],
  shape:  [2, 3],
  device:  Cpu,
  backend:  "ndarray",
  kind:  "Float",
  dtype:  "f32",
}
```

## Global Print Options

For more fine-grained control over tensor printing, Burn provides a `PrintOptions` struct and a `set_print_options` function:

```
use burn::tensor::{set_print_options, PrintOptions};

let print_options = PrintOptions {
    precision: Some(2),
    ..Default::default()
};

set_print_options(print_options);
```

Options:

- `precision` : Number of decimal places for floating-point numbers (default: None)
- `threshold` : Maximum number of elements to display before summarizing (default: 1000)
- `edge_items` : Number of items to show at the beginning and end of each dimension when summarizing (default: 3)

### Checking Tensor Closeness

Burn provides a utility function `check_closeness` to compare two tensors and assess their similarity. This function is particularly useful for debugging and validating tensor operations, especially when working with floating-point arithmetic where small numerical differences can accumulate. It's also valuable when comparing model outputs during the process of importing models from other frameworks, helping to ensure that the imported model produces results consistent with the original.

Here's an example of how to use `check_closeness` :

```rust
use burn::tensor::{check_closeness, Tensor};
type B = burn::backend::NdArray;

let device = Default::default();
let tensor1 = Tensor::<B, 1>::from_floats(
    [1.0, 2.0, 3.0, 4.0, 5.0, 6.001, 7.002, 8.003, 9.004, 10.1],
    &device,
);
let tensor2 = Tensor::<B, 1>::from_floats(
    [1.0, 2.0, 3.0, 4.000, 5.0, 6.0, 7.001, 8.002, 9.003, 10.004],
    &device,
);

check_closeness(&tensor1, &tensor2);
```

The `check_closeness` function compares the two input tensors element-wise, checking their absolute differences against a range of epsilon values. It then prints a detailed report showing the percentage of elements that are within each tolerance level.

The output provides a breakdown for different epsilon values, allowing you to assess the closeness of the tensors at various precision levels. This is particularly helpful when dealing with operations that may introduce small numerical discrepancies.

The function uses color-coded output to highlight the results:

- Green [PASS]: All elements are within the specified tolerance.
- Yellow [WARN]: Most elements (90% or more) are within tolerance.
- Red [FAIL]: Significant differences are detected.

This utility can be invaluable when implementing or debugging tensor operations, especially those involving complex mathematical computations or when porting algorithms from other frameworks. It's also an essential tool when verifying the accuracy of imported models, ensuring that the Burn implementation produces results that closely match those of the original model.

# Autodiff

Burn's tensor also supports auto-differentiation, which is an essential part of any deep learning framework. We introduced the `Backend` trait in the previous section, but Burn also has another trait for autodiff: `AutodiffBackend`.

However, not all tensors support auto-differentiation; you need a backend that implements both the `Backend` and `AutodiffBackend` traits. Fortunately, you can add auto-differentiation capabilities to any backend using a backend decorator: `type MyAutodiffBackend = Autodiff<MyBackend>`. This decorator implements both the `AutodiffBackend` and `Backend` traits by maintaining a dynamic computational graph and utilizing the inner backend to execute tensor operations.

The `AutodiffBackend` trait adds new operations on float tensors that can't be called otherwise. It also provides a new associated type, `B::Gradients`, where each calculated gradient resides.

```
fn calculate_gradients<B: AutodiffBackend>(tensor: Tensor<B, 2>) ->
B::Gradients {
    let mut gradients = tensor.clone().backward();

    let tensor_grad = tensor.grad(&gradients);         // get
    let tensor_grad = tensor.grad_remove(&mut gradients); // pop

    gradients
}
```

Note that some functions will always be available even if the backend doesn't implement the `AutodiffBackend` trait. In such cases, those functions will do nothing.

| Burn API | PyTorch Equivalent |
|---|---|
| `tensor.detach()` | `tensor.detach()` |
| `tensor.require_grad()` | `tensor.requires_grad()` |
| `tensor.is_require_grad()` | `tensor.requires_grad` |
| `tensor.set_require_grad(require_grad)` | `tensor.requires_grad(False)` |

However, you're unlikely to make any mistakes since you can't call `backward` on a tensor that is on a backend that doesn't implement `AutodiffBackend`. Additionally, you can't retrieve the gradient of a tensor without an autodiff backend.

# Difference with PyTorch

The way Burn handles gradients is different from PyTorch. First, when calling `backward`, each parameter doesn't have its `grad` field updated. Instead, the backward pass returns all the calculated gradients in a container. This approach offers numerous benefits, such as the ability to easily send gradients to other threads.

You can also retrieve the gradient for a specific parameter using the `grad` method on a tensor. Since this method takes the gradients as input, it's hard to forget to call `backward` beforehand. Note that sometimes, using `grad_remove` can improve performance by allowing inplace operations.

In PyTorch, when you don't need gradients for inference or validation, you typically need to scope your code using a block.

```
# Inference mode
torch.inference():
   # your code
   ...

# Or no grad
torch.no_grad():
   # your code
   ...
```

With Burn, you don't need to wrap the backend with the `Autodiff` for inference, and you can call `inner()` to obtain the inner tensor, which is useful for validation.

```
/// Use `B: AutodiffBackend`
fn example_validation<B: AutodiffBackend>(tensor: Tensor<B, 2>) {
    let inner_tensor: Tensor<B::InnerBackend, 2> = tensor.inner();
    let _ = inner_tensor + 5;
}

/// Use `B: Backend`
fn example_inference<B: Backend>(tensor: Tensor<B, 2>) {
    let _ = tensor + 5;
    ...
}
```

### Gradients with Optimizers

We've seen how gradients can be used with tensors, but the process is a bit different when working with optimizers from `burn-core`. To work with the `Module` trait, a translation step is required to link tensor parameters with their gradients. This step is necessary to easily

support gradient accumulation and training on multiple devices, where each module can be forked and run on different devices in parallel. We'll explore deeper into this topic in the Module section.

# Module

The `Module` derive allows you to create your own neural network modules, similar to PyTorch. The derive function only generates the necessary methods to essentially act as a parameter container for your type, it makes no assumptions about how the forward pass is declared.

```
use burn::module::Module;
use burn::tensor::backend::Backend;

#[derive(Module, Debug)]
pub struct PositionWiseFeedForward<B: Backend> {
    linear_inner: Linear<B>,
    linear_outer: Linear<B>,
    dropout: Dropout,
    gelu: Gelu,
}

impl<B: Backend> PositionWiseFeedForward<B> {
    /// Normal method added to a struct.
    pub fn forward<const D: usize>(&self, input: Tensor<B, D>) -> Tensor<B, D>
{
        let x = self.linear_inner.forward(input);
        let x = self.gelu.forward(x);
        let x = self.dropout.forward(x);

        self.linear_outer.forward(x)
    }
}
```

Note that all fields declared in the struct must also implement the `Module` trait.

## Tensor

If you want to create your own module that contains tensors, and not just other modules defined with the `Module` derive, you need to be careful to achieve the behavior you want.

- `Param<Tensor<B, D>>` : If you want the tensor to be included as a parameter of your modules, you need to wrap the tensor in a `Param` struct. This will create an ID that will be used to identify this parameter. This is essential when performing module optimization and when saving states such as optimizer and module checkpoints. Note that a module's record only contains parameters.

- `Param<Tensor<B, D>>.set_require_grad(false)` : If you want the tensor to be included as a parameter of your modules, and therefore saved with the module's weights, but you don't want it to be updated by the optimizer.
- `Tensor<B, D>` : If you want the tensor to act as a constant that can be recreated when instantiating a module. This can be useful when generating sinusoidal embeddings, for example.

## Methods

These methods are available for all modules.

| Burn API | PyTorch Equivalent |
|---|---|
| `module.devices()` | N/A |
| `module.fork(device)` | Similar to `module.to(device).detach()` |
| `module.to_device(device)` | `module.to(device)` |
| `module.no_grad()` | `module.require_grad_(False)` |
| `module.num_params()` | N/A |
| `module.visit(visitor)` | N/A |
| `module.map(mapper)` | N/A |
| `module.into_record()` | Similar to `state_dict` |
| `module.load_record(record)` | Similar to `load_state_dict(state_dict)` |
| `module.save_file(file_path, recorder)` | N/A |
| `module.load_file(file_path, recorder)` | N/A |

Similar to the backend trait, there is also the `AutodiffModule` trait to signify a module with autodiff support.

| Burn API | PyTorch Equivalent |
|---|---|
| `module.valid()` | `module.eval()` |

# Visitor & Mapper

As mentioned earlier, modules primarily function as parameter containers. Therefore, we naturally offer several ways to perform functions on each parameter. This is distinct from PyTorch, where extending module functionalities is not as straightforward.

The `map` and `visitor` methods are quite similar but serve different purposes. Mapping is used for potentially mutable operations where each parameter of a module can be updated to a new value. In Burn, optimizers are essentially just sophisticated module mappers. Visitors, on the other hand, are used when you don't intend to modify the module but need to retrieve specific information from it, such as the number of parameters or a list of devices in use.

You can implement your own mapper or visitor by implementing these simple traits:

```
/// Module visitor trait.
pub trait ModuleVisitor<B: Backend> {
    /// Visit a float tensor in the module.
    fn visit_float<const D: usize>(&mut self, id: ParamId, tensor: &Tensor<B,
D>);
    /// Visit an int tensor in the module.
    fn visit_int<const D: usize>(&mut self, id: ParamId, tensor: &Tensor<B, D,
Int>);
    /// Visit a bool tensor in the module.
    fn visit_bool<const D: usize>(&mut self, id: ParamId, tensor: &Tensor<B, D,
Bool>);
}

/// Module mapper trait.
pub trait ModuleMapper<B: Backend> {
    /// Map a float tensor in the module.
    fn map_float<const D: usize>(&mut self, id: ParamId, tensor: Tensor<B, D>)
-> Tensor<B, D>;
    /// Map an int tensor in the module.
    fn map_int<const D: usize>(&mut self, id: ParamId, tensor: Tensor<B, D,
Int>) -> Tensor<B, D, Int>;
    /// Map a bool tensor in the module.
    fn map_bool<const D: usize>(&mut self, id: ParamId, tensor: Tensor<B, D,
Bool>) -> Tensor<B, D, Bool>;
}
```

Note that the trait doesn't require all methods to be implemented as they are already defined to perform no operation. If you're only interested in float tensors (like the majority of use cases), then you can simply implement `map_float` or `visit_float`.

For example, the `ModuleMapper` trait could be implemented to clamp all parameters into the range `[min, max]`.

```rust
/// Clamp parameters into the range `[min, max]`.
pub struct Clamp {
    /// Lower-bound of the range.
    pub min: f32,
    /// Upper-bound of the range.
    pub max: f32,
}

// Clamp all floating-point parameter tensors between `[min, max]`.
impl<B: Backend> ModuleMapper<B> for Clamp {
    fn map_float<const D: usize>(
        &mut self,
        _id: burn::module::ParamId,
        tensor: burn::prelude::Tensor<B, D>,
    ) -> burn::prelude::Tensor<B, D> {
        tensor.clamp(self.min, self.max)
    }
}

// Clamp module mapper into the range `[-0.5, 0.5]`
let mut clamp = Clamp {
    min: -0.5,
    max: 0.5,
};
let model = model.map(&mut clamp);
```

If you want to use this during training to constrain your model parameters, make sure that the parameter tensors are still tracked for autodiff. This can be done with a simple adjustment to the implementation.

```rust
impl<B: AutodiffBackend> ModuleMapper<B> for Clamp {
    fn map_float<const D: usize>(
        &mut self,
        _id: burn::module::ParamId,
        tensor: burn::prelude::Tensor<B, D>,
    ) -> burn::prelude::Tensor<B, D> {
        let is_require_grad = tensor.is_require_grad();

        let mut tensor = Tensor::from_inner(tensor.inner().clamp(self.min,
self.max));

        if is_require_grad {
            tensor = tensor.require_grad();
        }

        tensor
    }
}
```

# Module Display

Burn provides a simple way to display the structure of a module and its configuration at a glance. You can print the module to see its structure, which is useful for debugging and tracking changes across different versions of a module. (See the print output of the Basic Workflow Model example.)

To customize the display of a module, you can implement the `ModuleDisplay` trait for your module. This will change the default display settings for the module and its children. Note that `ModuleDisplay` is automatically implemented for all modules, but you can override it to customize the display by annotating the module with `#[module(custom_display)]`.

```rust
#[derive(Module, Debug)]
#[module(custom_display)]
pub struct PositionWiseFeedForward<B: Backend> {
    linear_inner: Linear<B>,
    linear_outer: Linear<B>,
    dropout: Dropout,
    gelu: Gelu,
}

impl<B: Backend> ModuleDisplay for PositionWiseFeedForward<B> {
    /// Custom settings for the display of the module.
    /// If `None` is returned, the default settings will be used.
    fn custom_settings(&self) -> Option<burn::module::DisplaySettings> {
        DisplaySettings::new()
            // Will show all attributes (default is false)
            .with_show_all_attributes(false)
            // Will show each attribute on a new line (default is true)
            .with_new_line_after_attribute(true)
            // Will show the number of parameters (default is true)
            .with_show_num_parameters(true)
            // Will indent by 2 spaces (default is 2)
            .with_indentation_size(2)
            // Will show the parameter ID (default is false)
            .with_show_param_id(false)
            // Convenience method to wrap settings in Some()
            .optional()
    }

    /// Custom content to be displayed.
    /// If `None` is returned, the default content will be used
    /// (all attributes of the module)
    fn custom_content(&self, content: Content) -> Option<Content> {
        content
            .add("linear_inner", &self.linear_inner)
            .add("linear_outer", &self.linear_outer)
            .add("anything", "anything_else")
            .optional()
    }
}
```

# Built-in Modules

Burn comes with built-in modules that you can use to build your own modules.

## General

| Burn API | PyTorch Equivalent |
|---|---|
| `BatchNorm` | `nn.BatchNorm1d`, `nn.BatchNorm2d` etc. |
| `Dropout` | `nn.Dropout` |
| `Embedding` | `nn.Embedding` |
| `Gelu` | `nn.Gelu` |
| `GroupNorm` | `nn.GroupNorm` |
| `HardSigmoid` | `nn.Hardsigmoid` |
| `InstanceNorm` | `nn.InstanceNorm1d`, `nn.InstanceNorm2d` etc. |
| `LayerNorm` | `nn.LayerNorm` |
| `LeakyRelu` | `nn.LeakyReLU` |
| `Linear` | `nn.Linear` |
| `Prelu` | `nn.PReLu` |
| `Relu` | `nn.ReLU` |
| `RmsNorm` | *No direct equivalent* |
| `SwiGlu` | *No direct equivalent* |
| `Interpolate1d` | *No direct equivalent* |
| `Interpolate2d` | *No direct equivalent* |

## Convolutions

| Burn API | PyTorch Equivalent |
|---|---|
| `Conv1d` | `nn.Conv1d` |
| `Conv2d` | `nn.Conv2d` |
| `Conv3d` | `nn.Conv3d` |
| `ConvTranspose1d` | `nn.ConvTranspose1d` |
| `ConvTranspose2d` | `nn.ConvTranspose2d` |
| `ConvTranspose3d` | `nn.ConvTranspose3d` |
| `DeformConv2d` | `torchvision.ops.DeformConv2d` |

## Pooling

| Burn API | PyTorch Equivalent |
|---|---|

| Burn API | PyTorch Equivalent |
|---|---|
| `AdaptiveAvgPool1d` | `nn.AdaptiveAvgPool1d` |
| `AdaptiveAvgPool2d` | `nn.AdaptiveAvgPool2d` |
| `AvgPool1d` | `nn.AvgPool1d` |
| `AvgPool2d` | `nn.AvgPool2d` |
| `MaxPool1d` | `nn.MaxPool1d` |
| `MaxPool2d` | `nn.MaxPool2d` |

## RNNs

| Burn API | PyTorch Equivalent |
|---|---|
| `Gru` | `nn.GRU` |
| `Lstm` / `BiLstm` | `nn.LSTM` |
| `GateController` | *No direct equivalent* |

## Transformer

| Burn API | PyTorch Equivalent |
|---|---|
| `MultiHeadAttention` | `nn.MultiheadAttention` |
| `TransformerDecoder` | `nn.TransformerDecoder` |
| `TransformerEncoder` | `nn.TransformerEncoder` |
| `PositionalEncoding` | *No direct equivalent* |
| `RotaryEncoding` | *No direct equivalent* |

## Loss

| Burn API | PyTorch Equivalent |
|---|---|
| `BinaryCrossEntropyLoss` | `nn.BCELoss` |
| `CosineEmbeddingLoss` | `nn.CosineEmbeddingLoss` |
| `CrossEntropyLoss` | `nn.CrossEntropyLoss` |
| `HuberLoss` | `nn.HuberLoss` |
| `MseLoss` | `nn.MSELoss` |

| Burn API | PyTorch Equivalent |
| --- | --- |
| `PoissonNllLoss` | `nn.PoissonNLLLoss` |

# Learner

The burn-train crate encapsulates multiple utilities for training deep learning models. The goal of the crate is to provide users with a well-crafted and flexible training loop, so that projects do not have to write such components from the ground up. Most of the interactions with `burn-train` will be with the `LearnerBuilder` struct, briefly presented in the previous training section. This struct enables you to configure the training loop, offering support for registering metrics, enabling logging, checkpointing states, using multiple devices, and so on.

There are still some assumptions in the current provided APIs, which may make them inappropriate for your learning requirements. Indeed, they assume your model will learn from a training dataset and be validated against another dataset. This is the most common paradigm, allowing users to do both supervised and unsupervised learning as well as fine-tuning. However, for more complex requirements, creating a custom training loop might be what you need.

## Usage

The learner builder provides numerous options when it comes to configurations.

| Configuration | Description |
|---|---|
| Training Metric | Register a training metric |
| Validation Metric | Register a validation metric |
| Training Metric Plot | Register a training metric with plotting (requires the metric to be numeric) |
| Validation Metric Plot | Register a validation metric with plotting (requires the metric to be numeric) |
| Metric Logger | Configure the metric loggers (default is saving them to files) |
| Renderer | Configure how to render metrics (default is CLI) |
| Grad Accumulation | Configure the number of steps before applying gradients |
| File Checkpointer | Configure how the model, optimizer and scheduler states are saved |
| Num Epochs | Set the number of epochs |
| Devices | Set the devices to be used |
| Checkpoint | Restart training from a checkpoint |

| Configuration | Description |
| --- | --- |
| Application logging | Configure the application logging installer (default is writing to `experiment.log`) |

When the builder is configured at your liking, you can then move forward to build the learner. The build method requires three inputs: the model, the optimizer and the learning rate scheduler. Note that the latter can be a simple float if you want it to be constant during training.

The result will be a newly created Learner struct, which has only one method, the `fit` function which must be called with the training and validation dataloaders. This will start the training and return the trained model once finished.

Again, please refer to the training section for a relevant code snippet.

## Artifacts

When creating a new builder, all the collected data will be saved under the directory provided as the argument to the `new` method. Here is an example of the data layout for a model recorded using the compressed message pack format, with the accuracy and loss metrics registered:

```
├── experiment.log
├── checkpoint
│   ├── model-1.mpk.gz
│   ├── optim-1.mpk.gz
│   └── scheduler-1.mpk.gz
│   ├── model-2.mpk.gz
│   ├── optim-2.mpk.gz
│   └── scheduler-2.mpk.gz
├── train
│   ├── epoch-1
│   │   ├── Accuracy.log
│   │   └── Loss.log
│   └── epoch-2
│       ├── Accuracy.log
│       └── Loss.log
└── valid
    ├── epoch-1
    │   ├── Accuracy.log
    │   └── Loss.log
    └── epoch-2
        ├── Accuracy.log
        └── Loss.log
```

You can choose to save or synchronize that local directory with a remote file system, if desired. The file checkpointer is capable of automatically deleting old checkpoints according to a specified configuration.

# Metric

When working with the learner, you have the option to record metrics that will be monitored throughout the training process. We currently offer a restricted range of metrics.

| Metric | Description |
|---|---|
| Accuracy | Calculate the accuracy in percentage |
| TopKAccuracy | Calculate the top-k accuracy in percentage |
| Precision | Calculate precision in percentage |
| Recall | Calculate recall in percentage |
| FBetaScore | Calculate $F_\beta$ score in percentage |
| AUROC | Calculate the area under curve of ROC in percentage |
| Loss | Output the loss used for the backward pass |
| CPU Temperature | Fetch the temperature of CPUs |
| CPU Usage | Fetch the CPU utilization |
| CPU Memory Usage | Fetch the CPU RAM usage |
| GPU Temperature | Fetch the GPU temperature |
| Learning Rate | Fetch the current learning rate for each optimizer step |
| CUDA | Fetch general CUDA metrics such as utilization |

In order to use a metric, the output of your training step has to implement the `Adaptor` trait from `burn-train::metric`. Here is an example for the classification output, already provided with the crate.

```rust
/// Simple classification output adapted for multiple metrics.
#[derive(new)]
pub struct ClassificationOutput<B: Backend> {
    /// The loss.
    pub loss: Tensor<B, 1>,

    /// The output.
    pub output: Tensor<B, 2>,

    /// The targets.
    pub targets: Tensor<B, 1, Int>,
}

impl<B: Backend> Adaptor<AccuracyInput<B>> for ClassificationOutput<B> {
    fn adapt(&self) -> AccuracyInput<B> {
        AccuracyInput::new(self.output.clone(), self.targets.clone())
    }
}

impl<B: Backend> Adaptor<LossInput<B>> for ClassificationOutput<B> {
    fn adapt(&self) -> LossInput<B> {
        LossInput::new(self.loss.clone())
    }
}
```

# Custom Metric

Generating your own custom metrics is done by implementing the `Metric` trait.

```rust
/// Metric trait.
///
/// # Notes
///
/// Implementations should define their own input type only used by the metric.
/// This is important since some conflict may happen when the model output is
adapted for each
/// metric's input type.
pub trait Metric: Send + Sync {
    /// The input type of the metric.
    type Input;

    /// The parameterized name of the metric.
    ///
    /// This should be unique, so avoid using short generic names, prefer using
the long name.
    ///
    /// For a metric that can exist at different parameters (e.g., top-k
accuracy for different
    /// values of k), the name should be unique for each instance.
    fn name(&self) -> String;

    /// Update the metric state and returns the current metric entry.
    fn update(&mut self, item: &Self::Input, metadata: &MetricMetadata) ->
MetricEntry;
    /// Clear the metric state.
    fn clear(&mut self);
}
```

As an example, let's see how the loss metric is implemented.

```rust
/// The loss metric.
#[derive(Default)]
pub struct LossMetric<B: Backend> {
    state: NumericMetricState,
    _b: B,
}

/// The loss metric input type.
#[derive(new)]
pub struct LossInput<B: Backend> {
    tensor: Tensor<B, 1>,
}


impl<B: Backend> Metric for LossMetric<B> {
    type Input = LossInput<B>;

    fn update(&mut self, loss: &Self::Input, _metadata: &MetricMetadata) ->
MetricEntry {
        let [batch_size] = loss.tensor.dims();
        let loss = loss
            .tensor
            .clone()
            .mean()
            .into_data()
            .iter::<f64>()
            .next()
            .unwrap();

        self.state.update(
            loss,
            batch_size,
            FormatOptions::new(self.name()).precision(2),
        )
    }

    fn clear(&mut self) {
        self.state.reset()
    }

    fn name(&self) -> String {
        "Loss".to_string()
    }
}
```

When the metric you are implementing is numeric in nature, you may want to also implement the `Numeric` trait. This will allow your metric to be plotted.

```rust
impl<B: Backend> Numeric for LossMetric<B> {
    fn value(&self) -> f64 {
        self.state.value()
    }
}
```

# Config

When writing scientific code, you normally have a lot of values that are set, and Deep Learning is no exception. Python has the possibility to define default parameters for functions, which helps improve the developer experience. However, this has the downside of potentially breaking your code when upgrading to a new version, as the default values might change without your knowledge, making debugging very challenging.

With that in mind, we came up with the Config system. It's a simple Rust derive that you can apply to your types, allowing you to define default values with ease. Additionally, all configs can be serialized, reducing potential bugs when upgrading versions and improving reproducibility.

```rust
use burn::config::Config;

#[derive(Config)]
pub struct MyModuleConfig {
    d_model: usize,
    d_ff: usize,
    #[config(default = 0.1)]
    dropout: f64,
}
```

The derive also adds useful `with_` methods for every attribute of your config, similar to a builder pattern, along with a `save` method.

```rust
fn main() {
    let config = MyModuleConfig::new(512, 2048);
    println!("{}", config.d_model); // 512
    println!("{}", config.d_ff); // 2048
    println!("{}", config.dropout); // 0.1
    let config =  config.with_dropout(0.2);
    println!("{}", config.dropout); // 0.2

    config.save("config.json").unwrap();
}
```

## Good practices

By using the config type it is easy to create new module instances. The initialization method should be implemented on the config type with the device as argument.

```
impl MyModuleConfig {
    /// Create a module on the given device.
    pub fn init<B: Backend>(&self, device: &B::Device) -> MyModule {
        MyModule {
            linear: LinearConfig::new(self.d_model, self.d_ff).init(device),
            dropout: DropoutConfig::new(self.dropout).init(),
        }
    }
}
```

Then we could add this line to the above `main`:

```
use burn::backend::Wgpu;
let device = Default::default();
let my_module = config.init::<Wgpu>(&device);
```

# Record

Records are how states are saved with Burn. Compared to most other frameworks, Burn has its own advanced saving mechanism that allows interoperability between backends with minimal possible runtime errors. There are multiple reasons why Burn decided to create its own saving formats.

First, Rust has serde, which is an extremely well-developed serialization and deserialization library that also powers the `safetensors` format developed by Hugging Face. If used properly, all the validations are done when deserializing, which removes the need to write validation code. Since modules in Burn are created with configurations, they can't implement serialization and deserialization. That's why the record system was created: allowing you to save the state of modules independently of the backend in use extremely fast while still giving you all the flexibility possible to include any non-serializable field within your module.

**Why not use safetensors?**

`safetensors` uses serde with the JSON file format and only supports serializing and deserializing tensors. The record system in Burn gives you the possibility to serialize any type, which is very useful for optimizers that save their state, but also for any non-standard, cutting-edge modeling needs you may have. Additionally, the record system performs automatic precision conversion by using Rust types, making it more reliable with fewer manual manipulations.

It is important to note that the `safetensors` format uses the word *safe* to distinguish itself from Pickle, which is vulnerable to Python code injection. On our end, the simple fact that we use Rust already ensures that no code injection is possible. If your storage mechanism doesn't handle data corruption, you might prefer a recorder that performs checksum validation (i.e., any recorder with Gzip compression).

# Recorder

Recorders are independent of the backend and serialize records with precision and a format. Note that the format can also be in-memory, allowing you to save the records directly into bytes.

| Recorder | Format | Compression |
|---|---|---|
| DefaultFileRecorder | File - Named MessagePack | None |

| Recorder | Format | Compression |
|---|---|---|
| NamedMpkFileRecorder | File - Named MessagePack | None |
| NamedMpkGzFileRecorder | File - Named MessagePack | Gzip |
| BinFileRecorder | File - Binary | None |
| BinGzFileRecorder | File - Binary | Gzip |
| JsonGzFileRecorder | File - Json | Gzip |
| PrettyJsonFileRecorder | File - Pretty Json | Gzip |
| BinBytesRecorder | In Memory - Binary | None |

Each recorder supports precision settings decoupled from the precision used for training or inference. These settings allow you to define the floating-point and integer types that will be used for serialization and deserialization.

| Setting | Float Precision | Integer Precision |
|---|---|---|
| `DoublePrecisionSettings` | `f64` | `i64` |
| `FullPrecisionSettings` | `f32` | `i32` |
| `HalfPrecisionSettings` | `f16` | `i16` |

Note that when loading a record into a module, the type conversion is automatically handled, so you can't encounter errors. The only crucial aspect is using the same recorder for both serialization and deserialization; otherwise, you will encounter loading errors.

**Which recorder should you use?**

- If you want fast serialization and deserialization, choose a recorder without compression. The one with the lowest file size without compression is the binary format; otherwise, the named MessagePack could be used.
- If you want to save models for storage, you can use compression, but avoid using the binary format, as it may not be backward compatible.
- If you want to debug your model's weights, you can use the pretty JSON format.
- If you want to deploy with `no-std`, use the in-memory binary format and include the bytes with the compiled code.

For examples on saving and loading records, take a look at Saving and Loading Models.

# Dataset

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.

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 similar to the dataset abstract class in PyTorch:

```rust
pub trait Dataset<I>: Send + Sync {
    fn get(&self, index: usize) -> Option<I>;
    fn len(&self) -> usize;
}
```

The dataset trait assumes a fixed-length set of items that can be randomly accessed in constant time. This is a major difference from datasets that use Apache Arrow underneath to improve streaming performance. Datasets in Burn don't assume *how* they are going to be accessed; it's just a collection of items.

However, you can compose multiple dataset transformations to lazily obtain what you want with zero pre-processing, so that your training can start instantly!

## Transformation

Transformations in Burn are all lazy and modify one or multiple input datasets. The goal of these transformations is to provide you with the necessary tools so that you can model complex data distributions.

| Transformation | Description |
|---|---|
| `SamplerDataset` | Samples items from a dataset. This is a convenient way to model a dataset as a probability distribution of a fixed size. |
| `ShuffledDataset` | Maps each input index to a random index, similar to a dataset sampled without replacement. |
| `PartialDataset` | Returns a view of the input dataset with a specified range. |
| `MapperDataset` | Computes a transformation lazily on the input dataset. |

| Transformation | Description |
|---|---|
| `ComposedDataset` | Composes multiple datasets together to create a larger one without copying any data. |
| `WindowsDataset` | Dataset designed to work with overlapping windows of data extracted from an input dataset. |

Let us look at the basic usages of each dataset transform and how they can be composed together. These transforms are lazy by default except when specified, reducing the need for unnecessary intermediate allocations and improving performance. The full documentation of each transform can be found at the API reference.

- **SamplerDataset**: This transform can be used to sample items from a dataset with (default) or without replacement. Transform is initialized with a sampling size which can be bigger or smaller than the input dataset size. This is particularly useful in cases where we want to checkpoint larger datasets more often during training and smaller datasets less often as the size of an epoch is now controlled by the sampling size. Sample usage:

```
type DbPedia = SqliteDataset<DbPediaItem>;
let dataset: DbPedia = HuggingfaceDatasetLoader::new("dbpedia_14")
        .dataset("train").
        .unwrap();

let dataset = SamplerDataset<DbPedia, DbPediaItem>::new(dataset, 10000);
```

- **ShuffledDataset**: This transform can be used to shuffle the items of a dataset. Particularly useful before splitting the raw dataset into train/test splits. Can be initialized with a seed to ensure reproducibility.

```
let dataset = ShuffledDataset<DbPedia, DbPediaItem>::with_seed(dataset, 42);
```

- **PartialDataset**: This transform is useful to return a view of the dataset with specified start and end indices. Used to create train/val/test splits. In the example below, we show how to chain ShuffledDataset and PartialDataset to create splits.

```
// define chained dataset type here for brevity
type PartialData = PartialDataset<ShuffledDataset<DbPedia, DbPediaItem>>;
let len = dataset.len();
let split == "train"; // or "val"/"test"

let data_split = match split {
            "train" => PartialData::new(dataset, 0, len * 8 / 10), // Get first
80% dataset
            "test" => PartialData::new(dataset, len * 8 / 10, len), // Take
remaining 20%
            _ => panic!("Invalid split type"),                      // Handle
unexpected split types
        };
```

- **MapperDataset**: This transform is useful to apply a transformation on each of the items of a dataset. Particularly useful for normalization of image data when channel means are known.
- **ComposedDataset**: This transform is useful to compose multiple datasets downloaded from multiple sources (say different HuggingfaceDatasetLoader sources) into a single bigger dataset which can be sampled from one source.
- **WindowsDataset**: This transform is useful to create overlapping windows of a dataset. Particularly useful for sequential Time series Data, for example when working with an LSTM.

## Storage

There are multiple dataset storage options available for you to choose from. The choice of the dataset to use should be based on the dataset's size as well as its intended purpose.

| Storage | Description |
|---|---|
| `InMemDataset` | In-memory dataset that uses a vector to store items. Well-suited for smaller datasets. |
| `SqliteDataset` | Dataset that uses SQLite to index items that can be saved in a simple SQL database file. Well-suited for larger datasets. |
| `DataframeDataset` | Dataset that uses Polars dataframe to store and manage data. Well-suited for efficient data manipulation and analysis. |

# Sources

For now, there are only a couple of dataset sources available with Burn, but more to come!

## Hugging Face

You can easily import any Hugging Face dataset with Burn. We use SQLite as the storage to avoid downloading the model each time or starting a Python process. You need to know the format of each item in the dataset beforehand. Here's an example with the dbpedia dataset.

```
#[derive(Clone, Debug, serde::Serialize, serde::Deserialize)]
pub struct DbPediaItem {
    pub title: String,
    pub content: String,
    pub label: usize,
}

fn main() {
    let dataset: SqliteDataset<DbPediaItem> =
HuggingfaceDatasetLoader::new("dbpedia_14")
        .dataset("train") // The training split.
        .unwrap();
}
```

We see that items must derive `serde::Serialize`, `serde::Deserialize`, `Clone`, and `Debug`, but those are the only requirements.

## Images

`ImageFolderDataset` is a generic vision dataset used to load images from disk. It is currently available for multi-class and multi-label classification tasks as well as semantic segmentation and object detection tasks.

```
// Create an image classification dataset from the root folder,
// where images for each class are stored in their respective folder.
//
// For example:
// root/dog/dog1.png
// root/dog/dog2.png
// ...
// root/cat/cat1.png
let dataset = ImageFolderDataset::new_classification("path/to/dataset/
root").unwrap();
```

```rust
// Create a multi-label image classification dataset from a list of items,
// where each item is a tuple `(image path, labels)`, and a list of classes
// in the dataset.
//
// For example:
let items = vec![
    ("root/dog/dog1.png", vec!["animal".to_string(), "dog".to_string()]),
    ("root/cat/cat1.png", vec!["animal".to_string(), "cat".to_string()]),
];
let dataset = ImageFolderDataset::new_multilabel_classification_with_items(
    items,
    &["animal", "cat", "dog"],
)
.unwrap();


// Create a segmentation mask dataset from a list of items, where each
// item is a tuple `(image path, mask path)` and a list of classes
// corresponding to the integer values in the mask.
let items = vec![
    (
        "path/to/images/image0.png",
        "path/to/annotations/mask0.png",
    ),
    (
        "path/to/images/image1.png",
        "path/to/annotations/mask1.png",
    ),
    (
        "path/to/images/image2.png",
        "path/to/annotations/mask2.png",
    ),
];
let dataset = ImageFolderDataset::new_segmentation_with_items(
    items,
    &[
        "cat", // 0
        "dog", // 1
        "background", // 2
    ],
)
.unwrap();
```

```
// Create an object detection dataset from a COCO dataset. Currently only
// the import of object detection data (bounding boxes) is supported.
//
// COCO offers separate annotation and image archives for training and
// validation, paths to the unpacked files need to be passed as parameters:

let dataset = ImageFolderDataset::new_coco_detection(
    "/path/to/coco/instances_train2017.json",
    "/path/to/coco/images/train2017"
)
.unwrap();
```

### Comma-Separated Values (CSV)

Loading records from a simple CSV file in-memory is simple with the `InMemDataset`:

```
// Build dataset from csv with tab ('\t') delimiter.
// The reader can be configured for your particular file.
let mut rdr = csv::ReaderBuilder::new();
let rdr = rdr.delimiter(b'\t');

let dataset = InMemDataset::from_csv("path/to/csv", rdr).unwrap();
```

Note that this requires the `csv` crate.

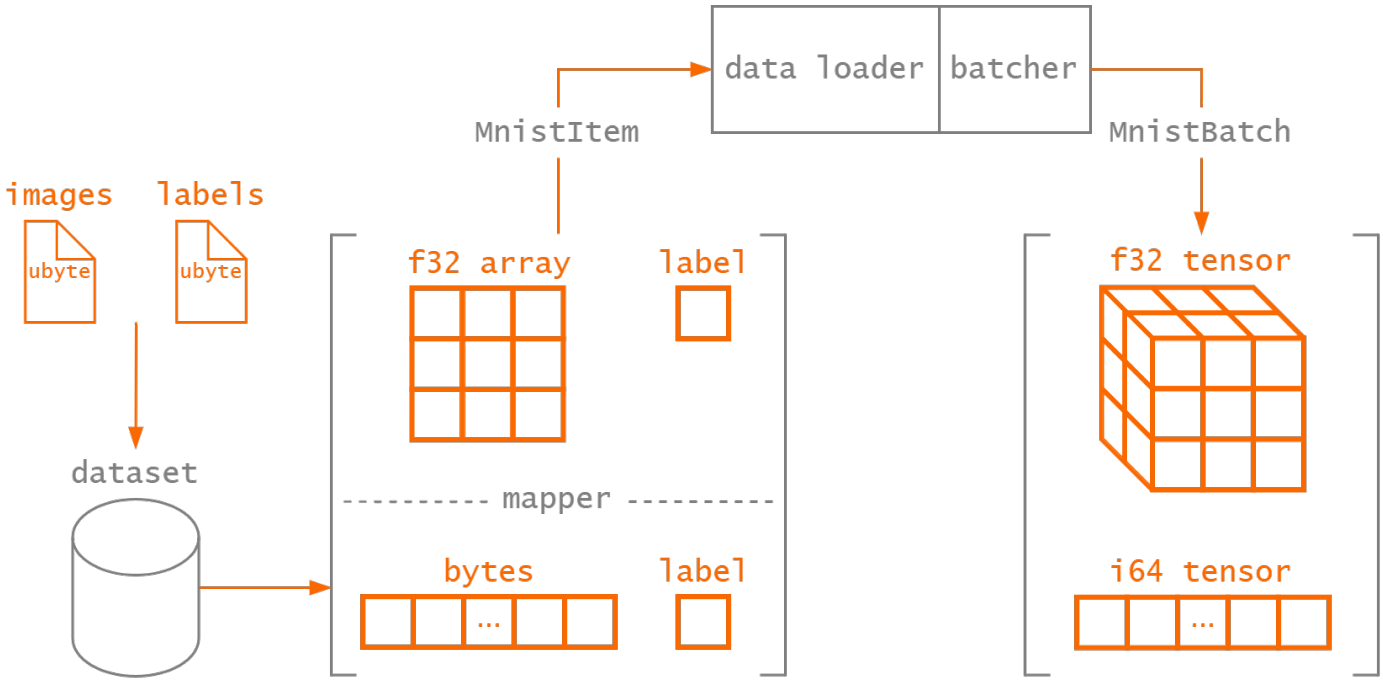
#### What about streaming datasets?

There is no streaming dataset API with Burn, and this is by design! The learner struct will iterate multiple times over the dataset and only checkpoint when done. You can consider the length of the dataset as the number of iterations before performing checkpointing and running the validation. There is nothing stopping you from returning different items even when called with the same `index` multiple times.

## How Is The Dataset Used?

During training, the dataset is used to access the data samples and, for most use cases in supervised learning, their corresponding ground-truth labels. Remember that 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 tensor 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 is done in the basic workflow guide. The process is illustrated in the figure below for the MNIST dataset.




Although we have conveniently implemented the `MnistDataset` used in the guide, we'll go over its implementation to demonstrate how the `Dataset` and `Batcher` traits are used.

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 $28 \times 28$ pixels black-and-white image (stored as raw bytes) with its corresponding label (a digit between $0$ and $9$). This is defined by the `MnistItemRaw` struct.

```
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` 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;

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>;

pub struct MnistDataset {
    dataset: MappedDataset,
}
```

To construct the `MnistDataset`, the data source must be parsed into the expected `MappedDataset` type. Since both the train and test sets use the same file format, we can separate the functionality to load the `train()` and `test()` dataset.

```
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: Vec<Vec<u8>> = MnistDataset::read_images(&root, split);
        let labels: Vec<u8> = 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();

        // Create the MapperDataset for InMemDataset<MnistItemRaw> to transform
        // items (MnistItemRaw -> MnistItem)
        let dataset = InMemDataset::new(items);
        let dataset = MapperDataset::new(dataset, BytesToImage);

        Self { dataset }
    }

}
```

Since the `MnistDataset` simply wraps a `MapperDataset` instance with `InMemDataset`, we can easily implement the `Dataset` trait.

```
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`, which we already went over in the basic workflow guide. The `Batcher` takes a list of `MnistItem` retrieved by the dataloader as input and returns a batch of images as a 3D tensor along with their targets.

# Custom Training Loops

Even though Burn comes with a project dedicated to simplifying training, it doesn't mean that you have to use it. Sometimes you may have special needs for your training, and it might be faster to just reimplement the training loop yourself. Also, you may just prefer implementing your own training loop instead of using a pre-built one in general.

Burn's got you covered!

We will start from the same example shown in the basic workflow section, but without using the `Learner` struct.

```rust
#[derive(Config)]
pub struct MnistTrainingConfig {
    #[config(default = 10)]
    pub num_epochs: usize,
    #[config(default = 64)]
    pub batch_size: usize,
    #[config(default = 4)]
    pub num_workers: usize,
    #[config(default = 42)]
    pub seed: u64,
    #[config(default = 1e-4)]
    pub lr: f64,
    pub model: ModelConfig,
    pub optimizer: AdamConfig,
}

pub fn run<B: AutodiffBackend>(device: &B::Device) {
    // Create the configuration.
    let config_model = ModelConfig::new(10, 1024);
    let config_optimizer = AdamConfig::new();
    let config = MnistTrainingConfig::new(config_model, config_optimizer);

    B::seed(config.seed);

    // Create the model and optimizer.
    let mut model = config.model.init::<B>(&device);
    let mut optim = config.optimizer.init();

    // Create the batcher.
    let batcher = MnistBatcher::default();

    // Create the dataloaders.
    let dataloader_train = DataLoaderBuilder::new(batcher.clone())
        .batch_size(config.batch_size)
        .shuffle(config.seed)
        .num_workers(config.num_workers)
        .build(MnistDataset::train());

    let dataloader_test = DataLoaderBuilder::new(batcher)
        .batch_size(config.batch_size)
        .shuffle(config.seed)
        .num_workers(config.num_workers)
        .build(MnistDataset::test());

    ...
}
```

As seen with the previous example, setting up the configurations and the dataloader hasn't changed. Now, let's move forward and write our own training loop:

```
pub fn run<B: AutodiffBackend>(device: B::Device) {
    ...

    // Iterate over our training and validation loop for X epochs.
    for epoch in 1..config.num_epochs + 1 {
        // Implement our training loop.
        for (iteration, batch) in dataloader_train.iter().enumerate() {
            let output = model.forward(batch.images);
            let loss = CrossEntropyLoss::new(None, &output.device())
                .forward(output.clone(), batch.targets.clone());
            let accuracy = accuracy(output, batch.targets);

            println!(
                "[Train - Epoch {} - Iteration {}] Loss {:.3} | Accuracy {:.3}
%",
                epoch,
                iteration,
                loss.clone().into_scalar(),
                accuracy,
            );

            // Gradients for the current backward pass
            let grads = loss.backward();
            // Gradients linked to each parameter of the model.
            let grads = GradientsParams::from_grads(grads, &model);
            // Update the model using the optimizer.
            model = optim.step(config.lr, model, grads);
        }

        // Get the model without autodiff.
        let model_valid = model.valid();

        // Implement our validation loop.
        for (iteration, batch) in dataloader_test.iter().enumerate() {
            let output = model_valid.forward(batch.images);
            let loss = CrossEntropyLoss::new(None, &output.device())
                .forward(output.clone(), batch.targets.clone());
            let accuracy = accuracy(output, batch.targets);

            println!(
                "[Valid - Epoch {} - Iteration {}] Loss {} | Accuracy {}",
                epoch,
                iteration,
                loss.clone().into_scalar(),
                accuracy,
            );
        }
    }
}
```

In the previous code snippet, we can observe that the loop starts from epoch `1` and goes up

to `num_epochs`. Within each epoch, we iterate over the training dataloader. During this process, we execute the forward pass, which is necessary for computing both the loss and accuracy. To maintain simplicity, we print the results to stdout.

Upon obtaining the loss, we can invoke the `backward()` function, which returns the gradients specific to each variable. It's important to note that we need to map these gradients to their corresponding parameters using the `GradientsParams` type. This step is essential because you might run multiple different autodiff graphs and accumulate gradients for each parameter id.

Finally, we can perform the optimization step using the learning rate, the model, and the computed gradients. It's worth mentioning that, unlike PyTorch, there's no need to register the gradients with the optimizer, nor do you have to call `zero_grad`. The gradients are automatically consumed during the optimization step. If you're interested in gradient accumulation, you can easily achieve this by using the `GradientsAccumulator`.

```
let mut accumulator = GradientsAccumulator::new();
let grads = model.backward();
let grads = GradientsParams::from_grads(grads, &model);
accumulator.accumulate(&model, grads); ...
let grads = accumulator.grads(); // Pop the accumulated gradients.
```

Note that after each epoch, we include a validation loop to assess our model's performance on previously unseen data. To disable gradient tracking during this validation step, we can invoke `model.valid()`, which provides a model on the inner backend without autodiff capabilities. It's important to emphasize that we've declared our validation batcher to be on the inner backend, specifically `MnistBatcher<B::InnerBackend>`; not using `model.valid()` will result in a compilation error.

You can find the code above available as an example for you to test.

## Multiple optimizers

It's common practice to set different learning rates, optimizer parameters, or use different optimizers entirely, for different parts of a model. In Burn, each `GradientParams` can contain only a subset of gradients to actually apply with an optimizer. This allows you to flexibly mix and match optimizers!

```
// Start with calculating all gradients
let grads = loss.backward();

// Now split the gradients into various parts.
let grads_conv1 = GradientParams::from_module(&mut grads, &model.conv1);
let grads_conv2 = GradientParams::from_module(&mut grads, &model.conv2);

// You can step the model with these gradients, using different learning
// rates for each param. You could also use an entirely different optimizer
here!
model = optim.step(config.lr * 2.0, model, grads_conv1);
model = optim.step(config.lr * 4.0, model, grads_conv2);

// For even more granular control you can split off individual parameter
// eg. a linear bias usually needs a smaller learning rate.
if let Some(bias) == model.linear1.bias {
    let grads_bias = GradientParams::from_params(&mut grads, &model.linear1,
&[bias.id]);
    model = optim.step(config.lr * 0.1, model, grads_bias);
}

// Note that above calls remove gradients, so we can just get all "remaining"
gradients.
let grads = GradientsParams::from_grads(grads, &model);
model = optim.step(config.lr, model, grads);
```

## Custom Type

The explanations above demonstrate how to create a basic training loop. However, you may find it beneficial to organize your program using intermediary types. There are various ways to do this, but it requires getting comfortable with generics.

If you wish to group the optimizer and the model into the same structure, you have several options. It's important to note that the optimizer trait depends on both the `AutodiffModule` trait and the `AutodiffBackend` trait, while the module only depends on the `AutodiffBackend` trait.

Here's a closer look at how you can create your types:

**Create a struct that is generic over the backend and the optimizer, with a predefined model.**

```
struct Learner<B, O>
where
    B: AutodiffBackend,
{
    model: Model<B>,
    optim: O,
}
```

This is quite straightforward. You can be generic over the backend since it's used with the concrete type `Model` in this case.

**Create a struct that is generic over the model and the optimizer.**

```
struct Learner<M, O> {
    model: M,
    optim: O,
}
```

This option is a quite intuitive way to declare the struct. You don't need to write type constraints with a `where` statement when defining a struct; you can wait until you implement the actual function. However, with this struct, you may encounter some issues when trying to implement code blocks to your struct.

```
impl<B, M, O> Learner<M, O>
where
    B: AutodiffBackend,
    M: AutodiffModule<B>,
    O: Optimizer<M, B>,
{
    pub fn step(&mut self, _batch: MnistBatch<B>) {
        //
    }
}
```

This will result in the following compilation error:

```
1. the type parameter `B` is not constrained by the impl trait, self type, or
predicates
   unconstrained type parameter [E0207]
```

To resolve this issue, you have two options. The first one is to make your function generic over the backend and add your trait constraint within its definition:

```
#[allow(dead_code)]
impl<M, O> Learner2<M, O> {
    pub fn step<B: AutodiffBackend>(&mut self, _batch: MnistBatch<B>)
    where
        B: AutodiffBackend,
        M: AutodiffModule<B>,
        O: Optimizer<M, B>,
    {
        //
    }
}
```

However, some people may prefer to have the constraints on the implementation block itself. In that case, you can make your struct generic over the backend using `PhantomData<B>`.

**Create a struct that is generic over the backend, the model, and the optimizer.**

```
struct Learner3<B, M, O> {
    model: M,
    optim: O,
    _b: PhantomData<B>,
}
```

You might wonder why `PhantomData` is required. Each generic argument must be used as a field when declaring a struct. When you don't need the generic argument, you can use `PhantomData` to mark it as a zero sized type.

These are just some suggestions on how to define your own types, but you are free to use any pattern that you prefer.

# Saving and Loading Models

Saving your trained machine learning model is quite easy, no matter the output format you choose. As mentioned in the Record section, different formats are supported to serialize/deserialize models. By default, we use the `NamedMpkFileRecorder` which uses the MessagePack binary serialization format with the help of rmp_serde.

```
// Save model in MessagePack format with full precision
let recorder = NamedMpkFileRecorder::<FullPrecisionSettings>::new();
model
    .save_file(model_path, &recorder)
    .expect("Should be able to save the model");
```

Note that the file extension is automatically handled by the recorder depending on the one you choose. Therefore, only the file path and base name should be provided.

Now that you have a trained model saved to your disk, you can easily load it in a similar fashion.

```
// Load model in full precision from MessagePack file
let recorder = NamedMpkFileRecorder::<FullPrecisionSettings>::new();
model = model
    .load_file(model_path, &recorder, device)
    .expect("Should be able to load the model weights from the provided file");
```

**Note:** models can be saved in different output formats, just make sure you are using the correct recorder type when loading the saved model. Type conversion between different precision settings is automatically handled, but formats are not interchangeable. A model can be loaded from one format and saved to another format, just as long as you load it back with the new recorder type afterwards.

## Initialization from Recorded Weights

The most straightforward way to load weights for a module is simply by using the generated method load_record. Note that parameter initialization is lazy, therefore no actual tensor allocation and GPU/CPU kernels are executed before the module is used. This means that you can use `init(device)` followed by `load_record(record)` without any meaningful performance cost.

```
// Create a dummy initialized model to save
let device = Default::default();
let model = Model::<MyBackend>::init(&device);

// Save model in MessagePack format with full precision
let recorder = NamedMpkFileRecorder::<FullPrecisionSettings>::new();
model
    .save_file(model_path, &recorder)
    .expect("Should be able to save the model");
```

Afterwards, the model can just as easily be loaded from the record saved on disk.

```
// Load model record on the backend's default device
let record: ModelRecord<MyBackend> =
NamedMpkFileRecorder::<FullPrecisionSettings>::new()
    .load(model_path.into(), &device)
    .expect("Should be able to load the model weights from the provided file");

// Initialize a new model with the loaded record/weights
let model = Model::init(&device).load_record(record);
```

## No Storage, No Problem!

For applications where file storage may not be available (or desired) at runtime, you can use the `BinBytesRecorder`.

In the previous examples we used a `FileRecorder` based on the MessagePack format, which could be replaced with another file recorder of your choice. To embed a model as part of your runtime application, first save the model to a binary file with `BinFileRecorder`.

```
// Save model in binary format with full precision
let recorder = BinFileRecorder::<FullPrecisionSettings>::new();
model
    .save_file(model_path, &recorder)
    .expect("Should be able to save the model");
```

Then, in your final application, include the model and use the `BinBytesRecorder` to load it.

Embedding the model as part of your application is especially useful for smaller models but not recommended for very large models as it would significantly increase the binary size as well as consume a lot more memory at runtime.

```rust
// Include the model file as a reference to a byte array
static MODEL_BYTES: &[u8] = include_bytes!("path/to/model.bin");

// Load model binary record in full precision
let record = BinBytesRecorder::<FullPrecisionSettings>::default()
    .load(MODEL_BYTES.to_vec(), device)
    .expect("Should be able to load model the model weights from bytes");

// Load that record with the model
model.load_record(record);
```

This example assumes that the model was already created before loading the model record. If instead you want to skip the random initialization and directly initialize the weights with the provided record, you could adapt this like the previous example.

# Importing Models

Burn supports importing models from other frameworks and file formats, enabling you to use pre-trained weights in your Burn applications.

## Supported Formats

Burn currently supports three primary model import formats:

| Format | Description | Use Case |
|---|---|---|
| **ONNX** | Open Neural Network Exchange format | Direct import of complete model architectures and weights from any framework that supports ONNX export |
| **PyTorch** | PyTorch weights (.pt, .pth) | Loading weights from PyTorch models into a matching Burn architecture |
| **Safetensors** | Hugging Face's model serialization format | Loading a model's tensor weights into a matching Burn architecture |

# Importing ONNX Models in Burn

## Introduction

As deep learning evolves, interoperability between frameworks becomes crucial. Burn, a modern deep learning framework in Rust, provides robust support for importing models from other popular frameworks. This section focuses on importing ONNX (Open Neural Network Exchange) models into Burn, enabling you to leverage pre-trained models in your Rust-based deep learning projects.

## Why Import Models?

Importing pre-trained models offers several advantages:

1. **Time-saving**: Skip the resource-intensive process of training models from scratch.
2. **Access to state-of-the-art architectures**: Utilize cutting-edge models developed by researchers and industry leaders.
3. **Transfer learning**: Fine-tune imported models for your specific tasks, benefiting from knowledge transfer.
4. **Consistency across frameworks**: Maintain consistent performance when moving between frameworks.

## Understanding ONNX

ONNX (Open Neural Network Exchange) is an open format designed to represent machine learning models with these key features:

- **Framework agnostic**: Provides a common format that works across various deep learning frameworks.
- **Comprehensive representation**: Captures both the model architecture and trained weights.
- **Wide support**: Compatible with popular frameworks like PyTorch, TensorFlow, and scikit-learn.

This standardization allows seamless movement of models between different frameworks

and deployment environments.

# Burn's ONNX Support

Burn's approach to ONNX import offers unique advantages:

1. **Native Rust code generation**: Translates ONNX models into Rust source code for deep integration with Burn's ecosystem.
2. **Compile-time optimization**: Leverages the Rust compiler to optimize the generated code, potentially improving performance.
3. **No runtime dependency**: Eliminates the need for an ONNX runtime, unlike many other solutions.
4. **Trainability**: Allows imported models to be further trained or fine-tuned using Burn.
5. **Portability**: Enables compilation for various targets, including WebAssembly and embedded devices.
6. **Backend flexibility**: Works with any of Burn's supported backends.

# ONNX Compatibility

Burn requires ONNX models to use **opset version 16 or higher**. If your model uses an older version, you'll need to upgrade it using the ONNX version converter.

## Upgrading ONNX Models

There are two simple ways to upgrade your ONNX models to the required opset version:

Option 1: Use the provided utility script:

```
uv run --script https://raw.githubusercontent.com/tracel-ai/burn/refs/heads/
main/crates/burn-import/onnx_opset_upgrade.py
```

Option 2: Use a custom Python script:

```
import onnx
from onnx import version_converter, shape_inference

# Load your ONNX model
model = onnx.load('path/to/your/model.onnx')

# Convert the model to opset version 16
upgraded_model = version_converter.convert_version(model, 16)

# Apply shape inference to the upgraded model
inferred_model = shape_inference.infer_shapes(upgraded_model)

# Save the converted model
onnx.save(inferred_model, 'upgraded_model.onnx')
```

## Step-by-Step Guide

Follow these steps to import an ONNX model into your Burn project:

### Step 1: Update `build.rs`

First, add the `burn-import` crate to your `Cargo.toml`:

```
[build-dependencies]
burn-import = "~0.18"
```

Then, in your `build.rs` file:

```
use burn_import::onnx::ModelGen;

fn main() {
    ModelGen::new()
        .input("src/model/my_model.onnx")
        .out_dir("model/")
        .run_from_script();
}
```

This generates Rust code from your ONNX model during the build process.

### Step 2: Modify `mod.rs`

In your `src/model/mod.rs` file, include the generated code:

```
pub mod my_model {
    include!(concat!(env!("OUT_DIR"), "/model/my_model.rs"));
}
```

### Step 3: Use the Imported Model

Now you can use the imported model in your code:

```
use burn::tensor;
use burn_ndarray::{NdArray, NdArrayDevice};
use model::my_model::Model;

fn main() {
    let device = NdArrayDevice::default();

    // Create model instance and load weights from target dir default device
    let model: Model<NdArray<f32>> = Model::default();

    // Create input tensor (replace with your actual input)
    let input = tensor::Tensor::<NdArray<f32>, 4>::zeros([1, 3, 224, 224],
&device);

    // Perform inference
    let output = model.forward(input);

    println!("Model output: {:?}", output);
}
```

## Advanced Configuration

The `ModelGen` struct provides several configuration options:

```
ModelGen::new()
    .input("path/to/model.onnx")
    .out_dir("model/")
    .record_type(RecordType::NamedMpk)
    .half_precision(false)
    .embed_states(false)
    .run_from_script();
```

- `record_type` : Defines the format for storing weights (Bincode, NamedMpk,

NamedMpkGz, or PrettyJson).
- `half_precision` : Reduces model size by using half-precision (f16) for weights.
- `embed_states` : Embeds model weights directly in the generated Rust code (requires record type `Bincode` ).

## Loading and Using Models

Depending on your configuration, you can load models in several ways:

```
// Create a new model instance with device
// (initializes weights randomly and lazily; load weights via `load_record`
afterward)
let model = Model::<Backend>::new(&device);

// Load from a file
// (file type should match the record type specified in `ModelGen`)
let model = Model::<Backend>::from_file("path/to/weights", &device);

// Load from embedded weights (if embed_states was true)
let model = Model::<Backend>::from_embedded(&device);

// Load from the output directory with default device (useful for testing)
let model = Model::<Backend>::default();
```

## Troubleshooting

Common issues and solutions:

1. **Unsupported ONNX operator**: Check the list of supported ONNX operators. You may need to simplify your model or wait for support.

2. **Build errors**: Ensure your `burn-import` version matches your Burn version and verify the ONNX file path in `build.rs` .

3. **Runtime errors**: Confirm that your input tensors match the expected shape and data type of your model.

4. **Performance issues**: Try using the `half_precision` option to reduce memory usage or experiment with different `record_type` options.

5. **Viewing generated files**: Find the generated Rust code and weights in the `OUT_DIR`

directory (usually `target/debug/build/<project>/out`).

## Examples and Resources

For practical examples, check out:

1. MNIST Inference Example
2. SqueezeNet Image Classification

These demonstrate real-world usage of ONNX import in Burn projects.

## Conclusion

Importing ONNX models into Burn combines the vast ecosystem of pre-trained models with Burn's performance and Rust's safety features. Following this guide, you can seamlessly integrate ONNX models into your Burn projects for inference, fine-tuning, or further development.

The `burn-import` crate is actively developed, with ongoing work to support more ONNX operators and improve performance. Stay tuned to the Burn repository for updates!

---

🚨 **Note**: The `burn-import` crate is in active development. For the most up-to-date information on supported ONNX operators, please refer to the official documentation.

# PyTorch Model

## Introduction

Burn supports importing model weights from PyTorch, whether you've trained your model in PyTorch or want to use a pre-trained model. Burn supports importing PyTorch model weights with `.pt` and `.safetensors` file extensions. Compared to ONNX models, these files only contain the weights of the model, so you will need to reconstruct the model architecture in Burn.

This guide demonstrates the complete workflow for exporting models from PyTorch and importing them into Burn. You can also refer to this Transitioning From PyTorch to Burn tutorial for importing a more complex model.

## Exporting Models to PyTorch Format

To export a PyTorch model correctly, you need to save only the model weights (state_dict) using the `torch.save` function, not the entire model.

### Example: Exporting a PyTorch Model

```python
import torch
import torch.nn as nn

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(2, 2, (2,2))
        self.conv2 = nn.Conv2d(2, 2, (2,2), bias=False)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        return x

if __name__ == "__main__":
    # Set seed for reproducibility
    torch.manual_seed(42)

    # Initialize model and ensure it's on CPU
    model = Net().to(torch.device("cpu"))

    # Extract model weights dictionary
    model_weights = model.state_dict()

    # Save only the weights, not the entire model
    torch.save(model_weights, "conv2d.pt")
```

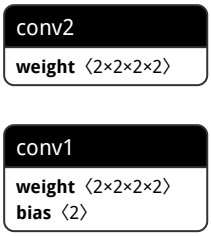
If you accidentally save the entire model instead of just the weights, you may encounter errors during import like:

```
Failed to decode foobar: DeserializeError("Serde error: other error:
Missing source values for the 'foo1' field of type 'BarRecordItem'.
Please verify the source data and ensure the field name is correct")
```

## Verifying the Export

You can verify your exported model by viewing the `.pt` file in Netron, a neural network visualization tool. A properly exported weights file will show a flat structure of tensors, while an incorrectly exported file will display nested blocks representing the entire model architecture.

When viewing the exported model in Netron, you should see something like this:




# Importing PyTorch Models into Burn

Importing a PyTorch model into Burn involves two main steps:

1. Defining the model architecture in Burn
2. Loading the weights from the exported PyTorch model

## Step 1: Define the Model in Burn

First, you need to create a Burn model that matches the architecture of the model you exported:

```rust
use burn::{
    nn::conv::{Conv2d, Conv2dConfig},
    prelude::*,
};

#[derive(Module, Debug)]
pub struct Net<B: Backend> {
    conv1: Conv2d<B>,
    conv2: Conv2d<B>,
}

impl<B: Backend> Net<B> {
    /// Create a new model.
    pub fn init(device: &B::Device) -> Self {
        let conv1 = Conv2dConfig::new([2, 2], [2, 2])
            .init(device);
        let conv2 = Conv2dConfig::new([2, 2], [2, 2])
            .with_bias(false)
            .init(device);
        Self { conv1, conv2 }
    }

    /// Forward pass of the model.
    pub fn forward(&self, x: Tensor<B, 4>) -> Tensor<B, 4> {
        let x = self.conv1.forward(x);
        self.conv2.forward(x)
    }
}
```

## Step 2: Load the Weights

You have two options for loading the weights:

### Option A: Load Dynamically at Runtime

This approach loads the PyTorch file directly at runtime, requiring the `burn-import` dependency:

```rust
use crate::model;
use burn::record::{FullPrecisionSettings, Recorder};
use burn_import::pytorch::PyTorchFileRecorder;

type Backend = burn_ndarray::NdArray<f32>;

fn main() {
    let device = Default::default();

    // Load weights from PyTorch file
    let record = PyTorchFileRecorder::<FullPrecisionSettings>::default()
        .load("./conv2d.pt".into(), &device)
        .expect("Should decode state successfully");

    // Initialize model and load weights
    let model = model::Net::<Backend>::init(&device).load_record(record);
}
```

### Option B: Pre-convert to Burn's Binary Format

This approach converts the PyTorch file to Burn's optimized binary format during build time, removing the runtime dependency on `burn-import`:

```rust
// This code would go in build.rs or a separate tool

use crate::model;
use burn::record::{FullPrecisionSettings, NamedMpkFileRecorder, Recorder};
use burn_import::pytorch::PyTorchFileRecorder;

type Backend = burn_ndarray::NdArray<f32>;

fn convert_model() {
    let device = Default::default();

    // Load from PyTorch
    let recorder = PyTorchFileRecorder::<FullPrecisionSettings>::default();
    let record = recorder
        .load("./conv2d.pt".into(), &device)
        .expect("Should decode state successfully");

    // Save to Burn's binary format
    let recorder = NamedMpkFileRecorder::<FullPrecisionSettings>::default();
    recorder
        .record(record, "model.mpk".into())
        .expect("Failed to save model record");
}

// In your application code
fn load_model() -> Net<Backend> {
    let device = Default::default();

    // Load from Burn's binary format
    let record = NamedMpkFileRecorder::<FullPrecisionSettings>::default()
        .load("./model.mpk".into(), &device)
        .expect("Should decode state successfully");

    Net::<Backend>::init(&device).load_record(record)
}
```

---

**Note**: For examples of pre-converting models, see the `examples/import-model-weights` directory in the Burn repository.

---

## Extract Configuration

In some cases, models may require additional configuration settings, which are often included in a `.pt` file during export. The `config_from_file` function from the `burn-import` cargo package allows for the extraction of these configurations directly from the `.pt` file.

```rust
use std::collections::HashMap;

use burn::config::Config;
use burn_import::pytorch::config_from_file;

#[derive(Debug, Config)]
struct NetConfig {
    n_head: usize,
    n_layer: usize,
    d_model: usize,
    some_float: f64,
    some_int: i32,
    some_bool: bool,
    some_str: String,
    some_list_int: Vec<i32>,
    some_list_str: Vec<String>,
    some_list_float: Vec<f64>,
    some_dict: HashMap<String, String>,
}

fn main() {
    let path = "weights_with_config.pt";
    let top_level_key = Some("my_config");
    let config: NetConfig = config_from_file(path, top_level_key).unwrap();
    println!("{:#?}", config);

    // After extracting, it's recommended you save it as a json file.
    config.save("my_config.json").unwrap();
}
```

# Troubleshooting and Advanced Features

## Key Remapping for Different Model Architectures

If your Burn model structure doesn't match the parameter names in the PyTorch file, you can remap keys using regular expressions:

```
let device = Default::default();
let load_args = LoadArgs::new("tests/key_remap/key_remap.pt".into())
    // Remove "conv" prefix, e.g. "conv.conv1" -> "conv1"
    .with_key_remap("conv\\.(.*)", "$1");

let record = PyTorchFileRecorder::<FullPrecisionSettings>::default()
    .load(load_args, &device)
    .expect("Should decode state successfully");

let model = Net::<Backend>::init(&device).load_record(record);
```

## Debugging with Key Inspection

To help with troubleshooting import issues, you can enable debugging to print the original and remapped keys:

```
let device = Default::default();
let load_args = LoadArgs::new("tests/key_remap/key_remap.pt".into())
    // Remove "conv" prefix, e.g. "conv.conv1" -> "conv1"
    .with_key_remap("conv\\.(.*)", "$1")
    .with_debug_print(); // Print the keys and remapped keys

let record = PyTorchFileRecorder::<FullPrecisionSettings>::default()
    .load(load_args, &device)
    .expect("Should decode state successfully");

let model = Net::<Backend>::init(&device).load_record(record);
```

Here is an example of the output:

```
Debug information of keys and tensor shapes:
---
Original Key: conv.conv1.bias
Remapped Key: conv1.bias
Shape: [2]
Dtype: F32
---
Original Key: conv.conv1.weight
Remapped Key: conv1.weight
Shape: [2, 2, 2, 2]
Dtype: F32
---
Original Key: conv.conv2.weight
Remapped Key: conv2.weight
Shape: [2, 2, 2, 2]
Dtype: F32
---
```

### Automatic Handling of Non-Contiguous Indices

The PyTorchFileRecorder automatically handles non-contiguous indices in model layer names. For example, if the source model contains indices with gaps:

```
"model.layers.0.weight"
"model.layers.0.bias"
"model.layers.2.weight"  // Note the gap (no index 1)
"model.layers.2.bias"
"model.layers.4.weight"
"model.layers.4.bias"
```

The recorder will automatically reindex these to be contiguous while preserving their order:

```
"model.layers.0.weight"
"model.layers.0.bias"
"model.layers.1.weight"  // Reindexed from 2
"model.layers.1.bias"
"model.layers.2.weight"  // Reindexed from 4
"model.layers.2.bias"
```

### Partial Model Loading

You can selectively load weights into a partial model, which is useful for:

- Loading only the encoder from an encoder-decoder architecture
- Fine-tuning specific layers while initializing others randomly
- Creating hybrid models combining parts from different sources

### Specifying the Top-Level Key for state_dict

Sometimes the `state_dict` is nested under a top-level key along with other metadata. In this case, you can specify the top-level key in `LoadArgs`:

```rust
let device = Default::default();
let load_args = LoadArgs::new("tiny.en.pt".into())
    .with_top_level_key("my_state_dict");

let record = PyTorchFileRecorder::<FullPrecisionSettings>::default()
    .load(load_args, &device)
    .expect("Should decode state successfully")
```

### Support for Enum Modules

The PyTorchFileRecorder supports models containing enum modules with new-type variants. The enum variant is automatically selected based on the enum variant type, allowing for flexible model architectures.

## Current Known Issues

1. Candle's pickle does not currently unpack boolean tensors.

# Safetensors Model

## Introduction

Burn supports importing model weights from the Safetensors format, a secure and efficient alternative to pickle-based formats. Whether you've trained your model in PyTorch or you want to use a pre-trained model that provides weights in Safetensors format, you can easily import them into Burn.

This guide demonstrates the complete workflow for exporting models to Safetensors format and importing them into Burn.

## Exporting Models to Safetensors Format

To export a PyTorch model to Safetensors format, you'll need the `safetensors` Python library. This library provides a simple API for saving model weights in the Safetensors format.

### Example: Exporting a PyTorch Model

```python
import torch
import torch.nn as nn
from safetensors.torch import save_file

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(2, 2, (2,2))
        self.conv2 = nn.Conv2d(2, 2, (2,2), bias=False)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        return x

if __name__ == "__main__":
    # Set seed for reproducibility
    torch.manual_seed(42)

    # Initialize model and ensure it's on CPU
    model = Net().to(torch.device("cpu"))

    # Extract model weights dictionary
    model_weights = model.state_dict()

    # Save to Safetensors format
    save_file(model_weights, "conv2d.safetensors")
```

### Verifying the Export

You can verify your exported model by viewing the `.safetensors` file in Netron, a neural network visualization tool. A correctly exported file will display a flat structure of tensors, similar to a PyTorch `.pt` weights file.

## Importing Safetensors Models into Burn

Importing a Safetensors model into Burn involves two main steps:

1. Defining the model architecture in Burn
2. Loading the weights from the Safetensors file

### Step 1: Define the Model in Burn

First, you need to create a Burn model that matches the architecture of the model you exported:

```
use burn::{
    nn::conv::{Conv2d, Conv2dConfig},
    prelude::*,
};

#[derive(Module, Debug)]
pub struct Net<B: Backend> {
    conv1: Conv2d<B>,
    conv2: Conv2d<B>,
}

impl<B: Backend> Net<B> {
    /// Create a new model.
    pub fn init(device: &B::Device) -> Self {
        let conv1 = Conv2dConfig::new([2, 2], [2, 2])
            .init(device);
        let conv2 = Conv2dConfig::new([2, 2], [2, 2])
            .with_bias(false)
            .init(device);
        Self { conv1, conv2 }
    }

    /// Forward pass of the model.
    pub fn forward(&self, x: Tensor<B, 4>) -> Tensor<B, 4> {
        let x = self.conv1.forward(x);
        self.conv2.forward(x)
    }
}
```

## Step 2: Load the Weights

You have two options for loading the weights:

### Option A: Load Dynamically at Runtime

This approach loads the Safetensors file directly at runtime, requiring the `burn-import` dependency:

```rust
use crate::model;
use burn::record::{FullPrecisionSettings, Recorder};
use burn_import::safetensors::SafetensorsFileRecorder;

type Backend = burn_ndarray::NdArray<f32>;

fn main() {
    let device = Default::default();

    // Load weights from Safetensors file
    let record = SafetensorsFileRecorder::<FullPrecisionSettings>::default()
        .load("./conv2d.safetensors".into(), &device)
        .expect("Should decode state successfully");

    // Initialize model and load weights
    let model = model::Net::<Backend>::init(&device).load_record(record);
}
```

### Option B: Pre-convert to Burn's Binary Format

This approach converts the Safetensors file to Burn's optimized binary format during build time, removing the runtime dependency on `burn-import`:

```rust
// This code would go in build.rs or a separate tool

use crate::model;
use burn::record::{FullPrecisionSettings, NamedMpkFileRecorder, Recorder};
use burn_import::safetensors::SafetensorsFileRecorder;

type Backend = burn_ndarray::NdArray<f32>;

fn convert_model() {
    let device = Default::default();

    // Load from Safetensors
    let recorder = SafetensorsFileRecorder::<FullPrecisionSettings>::default();
    let record = recorder
        .load("./conv2d.safetensors".into(), &device)
        .expect("Should decode state successfully");

    // Save to Burn's binary format
    let recorder = NamedMpkFileRecorder::<FullPrecisionSettings>::default();
    recorder
        .record(record, "model.mpk".into())
        .expect("Failed to save model record");
}

// In your application code
fn load_model() -> Net<Backend> {
    let device = Default::default();

    // Load from Burn's binary format
    let record = NamedMpkFileRecorder::<FullPrecisionSettings>::default()
        .load("./model.mpk".into(), &device)
        .expect("Should decode state successfully");

    Net::<Backend>::init(&device).load_record(record)
}
```

---

**Note**: For examples of pre-converting models, see the `examples/import-model-weights` directory in the Burn repository.

---

# Advanced Configuration Options

## Framework-Specific Adapters

When importing Safetensors models, you can specify an adapter type to handle framework-specific tensor transformations. This is crucial when importing models from different ML frameworks, as tensor layouts and naming conventions can vary:

```
let device = Default::default();

// Create load arguments with framework-specific adapter
let load_args = LoadArgs::new("model.safetensors".into())
    .with_adapter_type(AdapterType::PyTorch); // Default adapter

// Load with the specified adapter
let record = SafetensorsFileRecorder::<FullPrecisionSettings>::default()
    .load(load_args, &device)
    .expect("Should decode state successfully");
```

#### Available Adapter Types

| Adapter Type | Description |
|---|---|
| **PyTorch** (default) | Automatically applies PyTorch-specific transformations:<br>- Transposes weights for linear layers<br>- Renames normalization parameters (weight→gamma, bias→beta) |
| **NoAdapter** | Loads tensors directly without any transformations<br>- Useful when importing from frameworks that already match Burn's tensor layout |

# Troubleshooting and Advanced Features

## Key Remapping for Different Model Architectures

If your Burn model structure doesn't match the parameter names in the Safetensors file, you can remap keys using regular expressions:

```rust
let device = Default::default();

// Create load arguments with key remapping
let load_args = LoadArgs::new("model.safetensors".into())
    // Remove "conv" prefix, e.g. "conv.conv1" -> "conv1"
    .with_key_remap("conv\\.(.*)", "$1");

let record = SafetensorsFileRecorder::<FullPrecisionSettings>::default()
    .load(load_args, &device)
    .expect("Should decode state successfully");

let model = Net::<Backend>::init(&device).load_record(record);
```

## Debugging with Key Inspection

To help with troubleshooting import issues, you can enable debugging to print the original and remapped keys:

```rust
let device = Default::default();

// Enable debug printing of keys
let load_args = LoadArgs::new("model.safetensors".into())
    .with_key_remap("conv\\.(.*)", "$1")
    .with_debug_print();  // Print original and remapped keys

let record = SafetensorsFileRecorder::<FullPrecisionSettings>::default()
    .load(load_args, &device)
    .expect("Should decode state successfully");
```

## Automatic Handling of Non-Contiguous Indices

The SafetensorsFileRecorder automatically handles non-contiguous indices in model layer names. For example, if the source model contains indices with gaps:

```
"model.layers.0.weight"
"model.layers.0.bias"
"model.layers.2.weight"  // Note the gap (no index 1)
"model.layers.2.bias"
"model.layers.4.weight"
"model.layers.4.bias"
```

The recorder will automatically reindex these to be contiguous while preserving their order:

```
"model.layers.0.weight"
"model.layers.0.bias"
"model.layers.1.weight"  // Reindexed from 2
"model.layers.1.bias"
"model.layers.2.weight"  // Reindexed from 4
"model.layers.2.bias"
```

## Partial Model Loading

You can selectively load weights into a partial model, which is useful for:

- Loading only the encoder from an encoder-decoder architecture
- Fine-tuning specific layers while initializing others randomly
- Creating hybrid models combining parts from different sources

## Support for Enum Modules

The SafetensorsFileRecorder supports models containing enum modules with new-type variants. The enum variant is automatically selected based on the enum variant type, allowing for flexible model architectures.

# Models and Pre-Trained Weights

The `models` repository contains definitions of different deep learning models with examples for different domains like computer vision and natural language processing.

This includes image classification models such as `MobileNetV2`, `SqueezeNet` and `ResNet`, object detection models such as `YOLOX` and language models like `BERT` and `RoBERTa`.

Be sure to check out the up-to-date collection of models to get you started. Pre-trained weights are available for every supported architecture in this collection. You will also find a spotlight of community contributed models.

# Quantization (Beta)

Quantization techniques perform computations and store tensors in lower precision data types like 8-bit integer instead of floating point precision. There are multiple approaches to quantize a deep learning model categorized as:

- Post-training quantization (PTQ)
- Quantization aware training (QAT)

In post-training quantization, the model is trained in floating point precision and later converted to the lower precision data type.

There are two types of post-training quantization:

1. Static quantization: quantizes the weights and activations of the model. Quantizing the activations statically requires data to be calibrated (i.e., recording the activation values to compute the optimal quantization parameters with representative data).
2. Dynamic quantization: quantized the weights ahead of time (like static quantization) but the activations are dynamically at runtime.

Sometimes post-training quantization is not able to achieve acceptable task accuracy. This is where quantization aware training comes into play, as it models the effects of quantization during training. Quantization errors are thus modeled in the forward and backward passes using fake quantization modules, which helps the model learn representations that are more robust to the reduction in precision.

> Quantization support in Burn is currently in active development.
>
> It supports the following modes on some backends:
>
> - Static per-tensor quantization to signed 8-bit integer ( `i8` )
>
> No integer operations are currently supported, which means tensors are dequantized to perform the operations in floating point precision.

## Module Quantization

Quantizing the weights of your model after training is quite simple. We have access to the weight tensors and can collect their statistics, such as the min and max value when using `MinMaxCalibration` , to compute the quantization parameters.

```
// Quantization config
let mut quantizer = Quantizer {
    calibration: Calibration::MinMax,
    scheme: QuantizationScheme::PerTensor(QuantizationMode::Symmetric,
QuantizationType::QInt8),
};

// Quantize the weights
let model = model.quantize_weights(&mut quantizer);
```

---

Given that all operations are currently performed in floating point precision, it might be wise to dequantize the module parameters before inference. This allows us to save disk space by storing the model in reduced precision while preserving the inference speed.

This can easily be implemented with a `ModuleMapper`.

```
/// Module mapper used to dequantize the model params being loaded.
pub struct Dequantize {}

impl<B: Backend> ModuleMapper<B> for Dequantize {
    fn map_float<const D: usize>(
        &mut self,
        _id: ParamId,
        tensor: Tensor<B, D>,
    ) -> Tensor<B, D> {
        tensor.dequantize()
    }
}

// Load saved quantized model in floating point precision
model = model
    .load_file(file_path, recorder, &device)
    .expect("Should be able to load the quantized model weights")
    .map(&mut Dequantize {});
```

---

## Calibration

Calibration is the step during quantization where the range of all floating-point tensors is computed. This is pretty straightforward for weights since the actual range is known at *quantization-time* (weights are static), but activations require more attention.

To compute the quantization parameters, Burn supports the following `Calibration` methods.

| Method | Description |
| --- | --- |
| `MinMax` | Computes the quantization range mapping based on the running min and max values. |

## Quantization Scheme

A quantization scheme defines the quantized type, quantization granularity and range mapping technique.

Burn currently supports the following `QuantizationType` variants.

| Type | Description |
| --- | --- |
| `QInt8` | 8-bit signed integer quantization. |

Quantization parameters are defined based on the range of values to represent and can typically be calculated for the layer's entire weight tensor with per-tensor quantization or separately for each channel with per-channel quantization (commonly used with CNNs).

Burn currently supports the following `QuantizationScheme` variants.

| Variant | Description |
| --- | --- |
| `PerTensor(mode, type)` | Applies a single set of quantization parameters to the entire tensor. The `mode` defines how values are transformed, and `type` represents the target quantization type. |

### Quantization Mode

| Mode | Description |
| --- | --- |
| `Symmetric` | Maps values using a scale factor for a range centered around zero. |

# Advanced

In this section, we will go into advanced topics that extend beyond basic usage. Given Burn's exceptional flexibility, a lot of advanced use cases become possible.

Before going through this section, we strongly recommend exploring the basic workflow section and the building blocks section. Establishing a solid understanding of how the framework operates is crucial to comprehending the advanced concepts presented here. While you have the freedom to explore the advanced sections in any order you prefer, it's important to note that this section is not intended to be linear, contrary to preceding sections. Instead, it serves as a repository of use cases that you can refer to for guidance as needed.

# Backend Extension

Burn aims to be the most flexible deep learning framework. While it's crucial to maintain compatibility with a wide variety of backends, Burn provides the ability to extend the functionality of a backend implementation to suit your modeling requirements. This versatility is advantageous in numerous ways, such as supporting custom operations like flash attention or manually fusing operations for enhanced performance.

In this section, we will go into the process of extending a backend, providing multiple examples. But before we proceed, let's establish the fundamental principles that will empower you to craft your own backend extensions.

As you can observe, most types in Burn are generic over the Backend trait. This might give the impression that Burn operates at a high level over the backend layer. However, making the trait explicit instead of being chosen via a compilation flag was a thoughtful design decision. This explicitness does not imply that all backends must be identical; rather, it offers a great deal of flexibility when composing backends. The autodifferentiation backend trait (see autodiff section) is an example of how the backend trait has been extended to enable gradient computation with backpropagation. Furthermore, this design allows you to create your own backend extension. To achieve this, you need to design your own backend trait specifying which functions should be supported.

```
pub trait Backend: burn::tensor::backend::Backend {
    fn my_new_function(tensor: B::TensorPrimitive<2>) -> B::TensorPrimitive<2>
{
        // You can define a basic implementation reusing the Burn Backend API.
        // This can be useful since all backends will now automatically support
        // your model. But performance can be improved for this new
        // operation by implementing this block in specific backends.
    }
}
```

You can then implement your new custom backend trait for any backend that you want to support:

```
impl<E: TchElement> Backend for burn_tch::LibTorch<E> {
   fn my_new_function(tensor: TchTensor<E, 2>) -> TchTensor<E, 2> {
      // My Tch implementation
   }
}

impl<E: NdArrayElement> Backend for burn_ndarray::NdArray<E> {
    // No specific implementation, but the backend can still be used.
}
```

You can support the backward pass using the same pattern.

```
impl<B: Backend> Backend for burn_autodiff::Autodiff<B> {
    // No specific implementation; autodiff will work with the default
    // implementation. Useful if you still want to train your model, but
    // observe performance gains mostly during inference.
}

impl<B: Backend> Backend for burn_autodiff::Autodiff<B> {
   fn my_new_function(tensor: AutodiffTensor<E, 2>) -> AutodiffTensor<E, 2> {
      // My own backward implementation, generic over my custom Backend trait.
      //
      // You can add a new method `my_new_function_backward` to your custom
backend
      // trait if you want to invoke a custom kernel during the backward pass.
   }
}

impl<E: TchElement> Backend for burn_autodiff::Autodiff<burn_tch::LibTorch<E>>
{
   fn my_new_function(tensor: AutodiffTensor<E, 2>) -> AutodiffTensor<E, 2> {
      // My own backward implementation, generic over a backend implementation.
      //
      // This is another way to call a custom kernel for the backward pass that
      // doesn't require the addition of a new `backward` function in the
custom backend.
      // This is useful if you don't want all backends to support training,
reducing
      // the need for extra code when you know your model will only be trained
on one
      // specific backend.
   }
}
```

The specifics of each implementation will be covered by the examples provided in this section. The `cubecl` compiler frontend is the recommended method of implementing custom kernels, since it supports multiple backends, including `wgpu` and `CUDA`, and is the way first-party `burn` kernels are written.

# Custom `cubecl` Kernel

In this section, you will learn how to create your own custom operation by writing your own kernel with the cubecl compiler frontend. We will take the example of a common workflow in the deep learning field, where we create a kernel to fuse multiple operations together. Note that `burn` does this automatically, but a manual implementation might be more efficient in some cases. We will fuse a matmul kernel followed by an addition and the ReLU activation function, which is commonly found in various models. All the code can be found under the examples directory.

## Custom Backend Trait

First, we need to determine the type signature of our newly created operation by defining our custom backend traits. As we will use the associated type `TensorPrimitive` of the `Backend` trait, which encapsulates the underlying tensor implementation of the backend, we will use a type alias to avoid the ugly disambiguation with associated types.

```
/// We create our own Backend trait that extends the Burn backend trait.
pub trait Backend: burn::tensor::backend::Backend {
    fn fused_matmul_add_relu(
        lhs: FloatTensor<Self>,
        rhs: FloatTensor<Self>,
        bias: FloatTensor<Self>,
    ) -> FloatTensor<Self>;
}

/// We create our own AutodiffBackend trait that extends the Burn autodiff
backend trait.
pub trait AutodiffBackend: Backend + burn::tensor::backend::AutodiffBackend {}
```

In our project, we can use these traits instead of the `burn::tensor::backend::{Backend, AutodiffBackend}` traits provided by Burn. Burn's user APIs typically make use of the `Tensor` struct rather than dealing directly with primitive tensor types. Therefore, we can encapsulate our newly defined backend traits with functions that expose new operations while maintaining a consistent API.

```rust
/// We define our custom implementation using the added function on our custom
backend.
pub fn matmul_add_relu_custom<B: Backend>(
    lhs: Tensor<B, 3>,
    rhs: Tensor<B, 3>,
    bias: Tensor<B, 3>,
) -> Tensor<B, 3> {
    let output = B::fused_matmul_add_relu(
        lhs.into_primitive().tensor(),
        rhs.into_primitive().tensor(),
        bias.into_primitive().tensor(),
    );

    Tensor::from_primitive(TensorPrimitive::Float(output))
}

/// We define a reference implementation using basic tensor operations.
pub fn matmul_add_relu_reference<B: Backend>(
    lhs: Tensor<B, 3>,
    rhs: Tensor<B, 3>,
    bias: Tensor<B, 3>,
) -> Tensor<B, 3> {
    let x = lhs.matmul(rhs) + bias;

    activation::relu(x)
}
```

Note that we also provide a reference implementation for testing purposes, which allows us to easily validate our new implementation. While not mandatory, having a reference implementation can be valuable, especially in projects where creating a reference implementation solely using basic tensor operations is feasible.

## Forward Kernel

Now, let's proceed to write the fused kernel using the `cubecl` compiler frontend. To keep things simple, we'll create a straightforward matmul kernel without employing any intricate techniques. We won't delve into the details of the `cube` macro, but if you're interested to learn more, please see `cubecl` Book. The actual matmul, add and relu computations are found at the end, after an extensive prelude that serves to correctly map each compute unit to the data it is responsible for, with support for batches.

```rust
use cubecl::{cube, prelude::*};

#[cube(launch)]
pub fn fused_matmul_add_relu_kernel<F: Float>(
    lhs: &Tensor<F>,
    rhs: &Tensor<F>,
    bias: &Tensor<F>,
    output: &mut Tensor<F>,
) {
    let row = ABSOLUTE_POS_X;
    let col = ABSOLUTE_POS_Y;
    let batch = ABSOLUTE_POS_Z;

    let n_rows = output.shape(output.rank() - 2);
    let n_cols = output.shape(output.rank() - 1);
    let dim_k = rhs.shape(rhs.rank() - 1);

    if row >= n_rows || col >= n_cols {
        return;
    }

    let offset_output = batch * n_rows * n_cols;
    let mut offset_lhs = 0;
    let mut offset_rhs = 0;

    let batch_dims = output.rank() - 2;
    for dim in 0..batch_dims {
        offset_lhs += offset_output / output.stride(dim) % lhs.shape(dim) *
lhs.stride(dim);
        offset_rhs += offset_output / output.stride(dim) % rhs.shape(dim) *
rhs.stride(dim);
    }

    let mut sum = F::new(0.0);
    for k in 0..dim_k {
        let lhs_index = row * dim_k + k;
        let rhs_index = k * n_cols + col;

        sum += lhs[offset_lhs + lhs_index] * rhs[offset_rhs + rhs_index];
    }

    let out_index = row * n_cols + col;
    let index = offset_output + out_index;

    output[index] = F::max(sum + bias[index], F::new(0.0));
}
```

Now, let's move on to the next step, which involves implementing the remaining code to launch the kernel. We'll go into implementing our custom backend trait for the generic JIT backend. This automatically implements the trait for `burn-cuda`, `burn-wgpu` as well as fusion.

```rust
/// Implement our custom backend trait for the generic `CubeBackend`.
impl<R: CubeRuntime, F: FloatElement, I: IntElement> Backend for CubeBackend<R,
F, I> {
    fn fused_matmul_add_relu(
        lhs: FloatTensor<Self>,
        rhs: FloatTensor<Self>,
        bias: FloatTensor<Self>,
    ) -> FloatTensor<Self> {
        // Define cube dim, hardcoded for simplicity.
        let cube_dim = CubeDim { x: 16, y: 16, z: 1 };

        lhs.assert_is_on_same_device(&rhs);
        lhs.assert_is_on_same_device(&bias);

        // For simplicity, make sure each tensor is continuous.
        let lhs = into_contiguous(lhs);
        let rhs = into_contiguous(rhs);
        let bias = into_contiguous(bias);

        // Get the matmul relevant shapes.
        let ndims = lhs.shape.num_dims();
        let num_rows = lhs.shape.dims[ndims - 2];
        let num_cols = rhs.shape.dims[ndims - 1];

        // Compute shape of output, while tracking number of batches.
        let mut num_batches = 1;
        let mut shape_out = vec![0; ndims];
        for i in shape_out.clone().into_iter().take(ndims - 2) {
            shape_out[i] = usize::max(lhs.shape.dims[i], rhs.shape.dims[i]);
            num_batches *= shape_out[i];
        }
        shape_out[ndims - 2] = num_rows;
        shape_out[ndims - 1] = num_cols;
        let shape_out = Shape::from(shape_out);

        // Create a buffer for the output tensor.
        let buffer = lhs
            .client
            .empty(shape_out.num_elements() * core::mem::size_of::<F>());

        // Create the output tensor primitive.
        // Create the output tensor primitive.
        let output = CubeTensor::new_contiguous(
            lhs.client.clone(),
            lhs.device.clone(),
            shape_out,
            buffer,
            F::dtype(),
        );

        // Declare the wgsl workgroup with the number of cubes in x, y and z.
        let cubes_needed_in_x = f32::ceil(num_rows as f32 / cube_dim.x as f32)
```

```
as u32;
        let cubes_needed_in_y = f32::ceil(num_cols as f32 / cube_dim.y as f32)
as u32;
        let cube_count =
            CubeCount::Static(cubes_needed_in_x, cubes_needed_in_y, num_batches
as u32);

        // Execute lazily the kernel with the launch information and the given
buffers. For
        // simplicity, no vectorization is performed
        fused_matmul_add_relu_kernel::launch::<F, R>(
            &lhs.client,
            cube_count,
            cube_dim,
            lhs.as_tensor_arg::<F>(1),
            rhs.as_tensor_arg::<F>(1),
            bias.as_tensor_arg::<F>(1),
            output.as_tensor_arg::<F>(1),
        );

        // Return the output tensor.
        output
    }
}
```

In the preceding code block, we demonstrated how to launch the kernel that modifies the correct buffer. It's important to note that Rust's mutability safety doesn't apply here; the context has the capability to execute any mutable operation on any buffer. While this isn't a problem in the previous scenario where we only modify the newly created output buffer, it is wise to keep this in mind.

## Backward

Now that the custom backend trait is implemented for the JIT backend, you can use it to invoke the `matmul_add_relu_custom` function. However, calculating gradients is not yet possible at this stage. If your use case does not extend beyond inference, there is no need to implement any of the following code.

For the backward pass, we will leverage the backend implementation from `burn-autodiff`, which is actually generic over the backend. Instead of crafting our own `cubecl` kernel for the backward pass, we will use our fused kernel only for the forward pass, and compute the gradient using basic operations.

```
// Implement our custom backend trait for any backend that also implements our
custom backend trait.
impl<B: Backend, C: CheckpointStrategy> Backend for Autodiff<B, C> {
    fn fused_matmul_add_relu(
        lhs: FloatTensor<Self>,
        rhs: FloatTensor<Self>,
        bias: FloatTensor<Self>,
    ) -> FloatTensor<Self> {
        // Create our zero-sized type that will implement the Backward trait.
        #[derive(Debug)]
        struct FusedMatmulAddReluBackward;

        // Implement the backward trait for the given backend B, the node
gradient
        // with three other gradients to calculate (lhs, rhs, and bias).
        impl<B: Backend> Backward<B, 3> for FusedMatmulAddReluBackward {
            // Our state that we must build during the forward pass to compute
the backward pass.
            //
            // Note that we could improve the performance further by only
keeping the state of
            // tensors that are tracked, improving memory management, but for
simplicity, we avoid
            // that part.
            type State = (NodeID, NodeID, FloatTensor<B>, Shape);

            fn backward(
                self,
                ops: Ops<Self::State, 3>,
                grads: &mut Gradients,
                checkpointer: &mut Checkpointer,
            ) {
                // Get the nodes of each variable.
                let [node_lhs, node_rhs, node_bias] = ops.parents;
                // Fetch the gradient for the current node.
                let grad = grads.consume::<B>(&ops.node);

                // Set our state.
                let (lhs_state, rhs_state, output, shape_bias) = ops.state;
                let lhs: FloatTensor<B> =
checkpointer.retrieve_node_output(lhs_state);
                let rhs: FloatTensor<B> =
checkpointer.retrieve_node_output(rhs_state);

                // Fetch shapes of our tensor to support broadcasting.
                let shape_lhs = lhs.shape();
                let shape_rhs = rhs.shape();

                // Compute the gradient of the output using the already
existing `relu_backward`
                // function in the basic Burn backend trait.
                let grad_output = B::relu_backward(output, grad);
```

```
                // Compute the lhs gradient, which is the derivative of matmul
with support for
                // broadcasting.
                let grad_lhs = broadcast_shape::<B>(
                    B::float_matmul(grad_output.clone(),
B::float_transpose(rhs)),
                    &shape_lhs,
                );
                // Compute the rhs gradient, which is the derivative of matmul
with support for
                // broadcasting.
                let grad_rhs = broadcast_shape::<B>(
                    B::float_matmul(B::float_transpose(lhs),
grad_output.clone()),
                    &shape_rhs,
                );
                // The add derivative is only 1, so we just need to support
broadcasting to
                // compute the bias gradient.
                let grad_bias = broadcast_shape::<B>(grad_output, &shape_bias);

                // Register the gradient for each variable based on whether
they are marked as
                // `tracked`.
                if let Some(node) = node_bias {
                    grads.register::<B>(node.id, grad_bias);
                }
                if let Some(node) = node_lhs {
                    grads.register::<B>(node.id, grad_lhs);
                }
                if let Some(node) = node_rhs {
                    grads.register::<B>(node.id, grad_rhs);
                }
            }
        }

        // Prepare a stateful operation with each variable node and
corresponding graph.
        //
        // Each node can be fetched with `ops.parents` in the same order as
defined here.
        match FusedMatmulAddReluBackward
            .prepare::<C>([lhs.node.clone(), rhs.node.clone(),
bias.node.clone()])
            // Marks the operation as compute bound, meaning it will save its
            // state instead of recomputing itself during checkpointing
            .compute_bound()
            .stateful()
        {
            OpsKind::Tracked(mut prep) => {
                // When at least one node is tracked, we should register our
backward step.
```

```
                // The state consists of what will be needed for this
operation's backward pass.
                // Since we need the parents' outputs, we must checkpoint their
ids to retrieve
                // their node output at the beginning of the backward pass. We
can also save
                // utilitary data such as the bias shape. If we also need this
operation's output,
                // we can either save it in the state or recompute it.
                // during the backward pass. Here we choose to save it in the
state because it's a
                // compute bound operation.
                let lhs_state = prep.checkpoint(&lhs);
                let rhs_state = prep.checkpoint(&rhs);
                let bias_shape = bias.primitive.shape();

                let output = B::fused_matmul_add_relu(
                    lhs.primitive.clone(),
                    rhs.primitive.clone(),
                    bias.primitive,
                );

                let state = (lhs_state, rhs_state, output.clone(), bias_shape);

                prep.finish(state, output)
            }
            OpsKind::UnTracked(prep) => {
                // When no node is tracked, we can just compute the original
operation without
                // keeping any state.
                let output = B::fused_matmul_add_relu(lhs.primitive,
rhs.primitive, bias.primitive);
                prep.finish(output)
            }
        }
    }
}
```

The previous code is self-documented to make it clearer, but here is what it does in summary:

We define `fused_matmul_add_relu` within `Autodiff<B>`, allowing any autodiff-decorated backend to benefit from our implementation. In an autodiff-decorated backend, the forward pass must still be implemented. This is achieved using a comprehensive match statement block where computation is delegated to the inner backend, while keeping track of a state. The state comprises any information relevant to the backward pass, such as input and output tensors, along with the bias shape. When an operation isn't tracked (meaning there won't be a backward pass for this specific operation in the graph), storing a state becomes unnecessary, and we simply perform the forward computation.

The backward pass uses the gradient obtained from the preceding node in the computation graph. It calculates the derivatives for `relu` (`relu_backward`), add (no operation is required here, as the derivative is one), and `matmul` (another `matmul` with transposed inputs). This results in gradients for both input tensors and the bias, which are registered for consumption by subsequent operation nodes.

The only remaining part is to implement our autodiff-decorated backend trait for our JIT Backend.

```
impl<R: CubeRuntime, F: FloatElement, I: IntElement> AutodiffBackend
    for Autodiff<CubeBackend<R, F, I>>
{
}
```

## Conclusion

In this guide, we've implemented a fused kernel using the `cubecl` compiler frontend, enabling execution on any GPU and any `cubecl` backend. By delving into the inner workings of both the JIT backend and the autodiff backend, we've gained a deeper understanding of these systems.

While extending a backend may be harder than working with straightforward tensors, the benefits can be worth it. This approach enables the crafting of custom models with greater control over execution, which can potentially greatly enhance the performance of your models.

As we conclude this guide, we hope that you have gained insights into Burn's world of backend extensions, and that it will help you to unleash the full potential of your projects.

# Custom WGPU Kernel

In this section, you will learn how to create your own custom operation by writing your own kernel with the WGPU backend. We will take the example of a common workflow in the deep learning field, where we create a kernel to fuse multiple operations together. Note that `burn` does this automatically, but a manual implementation might be more efficient in some cases. We will fuse a matmul kernel followed by an addition and the ReLU activation function, which is commonly found in various models. All the code can be found under the examples directory.

## Custom Backend Trait

First, we need to determine the type signature of our newly created operation by defining our custom backend traits. As we will use the associated type `TensorPrimitive` of the `Backend` trait, which encapsulates the underlying tensor implementation of the backend, we will use a type alias to avoid the ugly disambiguation with associated types.

```
/// We create our own Backend trait that extends the Burn backend trait.
pub trait Backend: burn::tensor::backend::Backend {
    fn fused_matmul_add_relu(
        lhs: FloatTensor<Self>,
        rhs: FloatTensor<Self>,
        bias: FloatTensor<Self>,
    ) -> FloatTensor<Self>;
}

/// We create our own AutodiffBackend trait that extends the Burn autodiff
backend trait.
pub trait AutodiffBackend: Backend + burn::tensor::backend::AutodiffBackend {}
```

In our project, we can use these traits instead of the `burn::tensor::backend::{Backend, AutodiffBackend}` traits provided by Burn. Burn's user APIs typically make use of the `Tensor` struct rather than dealing directly with primitive tensor types. Therefore, we can encapsulate our newly defined backend traits with functions that expose new operations while maintaining a consistent API.

```rust
/// We define our custom implementation using the added function on our custom
backend.
pub fn matmul_add_relu_custom<B: Backend>(
    lhs: Tensor<B, 3>,
    rhs: Tensor<B, 3>,
    bias: Tensor<B, 3>,
) -> Tensor<B, 3> {
    let output = B::fused_matmul_add_relu(
        lhs.into_primitive().tensor(),
        rhs.into_primitive().tensor(),
        bias.into_primitive().tensor(),
    );

    Tensor::from_primitive(TensorPrimitive::Float(output))
}

/// We define a reference implementation using basic tensor operations.
pub fn matmul_add_relu_reference<B: Backend>(
    lhs: Tensor<B, 3>,
    rhs: Tensor<B, 3>,
    bias: Tensor<B, 3>,
) -> Tensor<B, 3> {
    let x = lhs.matmul(rhs) + bias;

    activation::relu(x)
}
```

Note that we also provide a reference implementation for testing purposes, which allows us to easily validate our new implementation. While not mandatory, having a reference implementation can be valuable, especially in projects where creating a reference implementation solely using basic tensor operations is feasible.

## Forward Kernel

Now, let's proceed to write the fused kernel using the WGSL shading language. To keep things simple, we'll create a straightforward matmul kernel without employing any intricate techniques. Although we won't delve into the details of the WGSL syntax, as it falls beyond the scope of this guide, we still provide the implementation below for readers who are curious. The actual matmul, add and relu computations are found at the end, after an extensive overhead whose use is to correctly map each compute unit to the data it is responsible of, with support for batches.

```
@group(0)
@binding(0)
var<storage, read_write> lhs: array<{{ elem }}>;

@group(0)
@binding(1)
var<storage, read_write> rhs: array<{{ elem }}>;

@group(0)
@binding(2)
var<storage, read_write> bias: array<{{ elem }}>;

@group(0)
@binding(3)
var<storage, read_write> output: array<{{ elem }}>;

@group(0)
@binding(4)
var<storage, read_write> info: array<u32>;

const BLOCK_SIZE = {{ workgroup_size_x }}u;

@compute
@workgroup_size({{ workgroup_size_x }}, {{ workgroup_size_y }}, 1)
fn main(
    @builtin(global_invocation_id) global_id: vec3<u32>,
    @builtin(local_invocation_index) local_idx: u32,
    @builtin(workgroup_id) workgroup_id: vec3<u32>,
) {
    // Indices
    let row = workgroup_id.x * BLOCK_SIZE + (local_idx / BLOCK_SIZE);
    let col = workgroup_id.y * BLOCK_SIZE + (local_idx % BLOCK_SIZE);
    let batch = global_id.z;

    // Basic information
    let dim = info[0];
    let n_rows = info[6u * dim - 1u];
    let n_cols = info[6u * dim];
    let K = info[5u * dim - 1u];

    // Returns if outside the output dimension
    if row >= n_rows || col >= n_cols {
        return;
    }

    // Calculate the corresponding offsets with support for broadcasting.
    let offset_output = batch * n_rows * n_cols;
    var offset_lhs: u32 = 0u;
    var offset_rhs: u32 = 0u;

    let batch_dims = dim - 2u;
    for (var b: u32 = 1u; b <= batch_dims; b++) {
```

```
        let stride_lhs = info[b];
        let stride_rhs = info[b + dim];
        let stride_output = info[b + 2u * dim];
        let shape_lhs = info[b + 3u * dim];
        let shape_rhs = info[b + 4u * dim];

        offset_lhs += offset_output / stride_output % shape_lhs * stride_lhs;
        offset_rhs += offset_output / stride_output % shape_rhs * stride_rhs;
    }

    // Basic matmul implementation
    var sum = 0.0;
    for (var k: u32 = 0u; k < K; k++) {
        let lhs_index = row * K + k;
        let rhs_index = k * n_cols + col;

        sum += lhs[offset_lhs + lhs_index] * rhs[offset_rhs + rhs_index];
    }

    let output_index = row * n_cols + col;
    let index = offset_output + output_index;

    // Add and ReLU
    output[index] = max(sum + bias[index], 0.0);
}
```

Now, let's move on to the next step, which involves implementing the remaining code to launch the kernel. The initial part entails loading the template and populating it with the appropriate variables. The `register(name, value)` method simply replaces occurrences of `{{ name }}` in the above WGSL code with some other string before it is compilated. In order to use templating utilities, you will have to activate the `template` feature of Burn in your `cargo.toml`.

```rust
// Source the kernel written in WGSL.
kernel_wgsl!(FusedMatmulAddReluRaw, "./kernel.wgsl");

// Define our kernel type with cube information.
#[derive(new, Debug)]
struct FusedMatmulAddRelu<E: FloatElement> {
    cube_dim: CubeDim,
    _elem: PhantomData<E>,
}

// Implement the dynamic kernel trait for our kernel type.
impl<E: FloatElement> KernelSource for FusedMatmulAddRelu<E> {
    fn source(&self) -> SourceTemplate {
        // Extend our raw kernel with cube size information using the
        // `SourceTemplate` trait.
        FusedMatmulAddReluRaw::new()
            .source()
            .register("workgroup_size_x", self.cube_dim.x.to_string())
            .register("workgroup_size_y", self.cube_dim.y.to_string())
            .register("elem", E::type_name())
            .register("int", "i32")
    }

    fn id(&self) -> cubecl::KernelId {
        cubecl::KernelId::new::<Self>().info(self.cube_dim)
    }
}
```

Subsequently, we'll go into implementing our custom backend trait for the WGPU backend. Note that we won't go into supporting the `fusion` feature flag in this tutorial, so we implement the trait for the raw `WgpuBackend` type.

```rust
/// Implement our custom backend trait for the existing backend `WgpuBackend`.
impl<F: FloatElement, I: IntElement> Backend for CubeBackend<WgpuRuntime, F, I>
{
    fn fused_matmul_add_relu(
        lhs: FloatTensor<Self>,
        rhs: FloatTensor<Self>,
        bias: FloatTensor<Self>,
    ) -> FloatTensor<Self> {
        // Define cube dim, hardcoded for simplicity.
        let cube_dim = CubeDim { x: 16, y: 16, z: 1 };

        lhs.assert_is_on_same_device(&rhs);
        lhs.assert_is_on_same_device(&bias);

        // For simplicity, make sure each tensor is continuous.
        let lhs = into_contiguous(lhs);
        let rhs = into_contiguous(rhs);
        let bias = into_contiguous(bias);

        // Get the matmul relevant shapes.
        let ndims = lhs.shape.num_dims();
        let num_rows = lhs.shape.dims[ndims - 2];
        let num_cols = rhs.shape.dims[ndims - 1];

        // Compute shape of output, while tracking number of batches.
        let mut num_batches = 1;
        let mut shape_out = vec![0; ndims];
        for i in shape_out.clone().into_iter().take(ndims - 2) {
            shape_out[i] = usize::max(lhs.shape.dims[i], rhs.shape.dims[i]);
            num_batches *= shape_out[i];
        }
        shape_out[ndims - 2] = num_rows;
        shape_out[ndims - 1] = num_cols;
        let shape_out = Shape::from(shape_out);

        // Create a buffer for the output tensor.
        let buffer = lhs
            .client
            .empty(shape_out.num_elements() * core::mem::size_of::<F>());

        // Create the output tensor primitive.
        let output = CubeTensor::new_contiguous(
            lhs.client.clone(),
            lhs.device.clone(),
            shape_out,
            buffer,
            F::dtype(),
        );

        // Create the kernel.
        let kernel = FusedMatmulAddRelu::<F>::new(cube_dim);
```

```
        // Build info buffer with tensor information needed by the kernel, such
as shapes and strides.
        let info = build_info::<_, F>(&[&lhs, &rhs, &output]);
        let info_handle = lhs.client.create(bytemuck::cast_slice(&info));

        // Declare the wgsl workgroup with the number of cubes in x, y and z.
        let cubes_needed_in_x = f32::ceil(num_rows as f32 / cube_dim.x as f32)
as u32;
        let cubes_needed_in_y = f32::ceil(num_cols as f32 / cube_dim.y as f32)
as u32;
        let cube_count =
            CubeCount::Static(cubes_needed_in_x, cubes_needed_in_y, num_batches
as u32);

        // Execute lazily the kernel with the launch information and the given
buffers.
        lhs.client.execute(
            Box::new(SourceKernel::new(kernel, cube_dim)),
            cube_count,
            vec![
                lhs.handle.binding(),
                rhs.handle.binding(),
                bias.handle.binding(),
                output.handle.clone().binding(),
                info_handle.binding(),
            ],
        );

        // Return the output tensor.
        output
    }
}
```

In the preceding code block, we demonstrated how to launch the kernel that modifies the correct buffer. It's important to note that Rust's mutability safety doesn't apply here; the context has the capability to execute any mutable operation on any buffer. While this isn't a problem in the previous scenario where we only modify the newly created output buffer, it is wise to keep this in mind.

## Backward

Now that the custom backend trait is implemented for the WGPU backend, you can use it to invoke the `matmul_add_relu_custom` function. However, calculating gradients is not yet possible at this stage. If your use case does not extend beyond inference, there is no need to implement any of the following code.

For the backward pass, we will leverage the backend implementation from `burn-autodiff`, which is actually generic over the backend. Instead of crafting our own WGSL kernel for the backward pass, we will use our fused kernel only for the forward pass, and compute the gradient using basic operations.

```
// Implement our custom backend trait for any backend that also implements our
custom backend trait.
//
// Note that we could implement the backend trait only for the Wgpu backend
instead of any backend that
// also implements our own API. This would allow us to call any function only
implemented for Wgpu
// and potentially call a custom kernel crafted only for this task.
impl<B: Backend, C: CheckpointStrategy> Backend for Autodiff<B, C> {
    fn fused_matmul_add_relu(
        lhs: FloatTensor<Self>,
        rhs: FloatTensor<Self>,
        bias: FloatTensor<Self>,
    ) -> FloatTensor<Self> {
        // Create our zero-sized type that will implement the Backward trait.
        #[derive(Debug)]
        struct FusedMatmulAddReluBackward;

        // Implement the backward trait for the given backend B, the node
gradient
        // with three other gradients to calculate (lhs, rhs, and bias).
        impl<B: Backend> Backward<B, 3> for FusedMatmulAddReluBackward {
            // Our state that we must build during the forward pass to compute
the backward pass.
            //
            // Note that we could improve the performance further by only
keeping the state of
            // tensors that are tracked, improving memory management, but for
simplicity, we avoid
            // that part.
            type State = (NodeID, NodeID, FloatTensor<B>, Shape);

            fn backward(
                self,
                ops: Ops<Self::State, 3>,
                grads: &mut Gradients,
                checkpointer: &mut Checkpointer,
            ) {
                // Get the nodes of each variable.
                let [node_lhs, node_rhs, node_bias] = ops.parents;
                // Fetch the gradient for the current node.
                let grad = grads.consume::<B>(&ops.node);

                // Set our state.
                let (lhs_state, rhs_state, output, shape_bias) = ops.state;
                let lhs: FloatTensor<B> =
checkpointer.retrieve_node_output(lhs_state);
                let rhs: FloatTensor<B> =
checkpointer.retrieve_node_output(rhs_state);

                // Fetch shapes of our tensor to support broadcasting.
                let shape_lhs = lhs.shape();
```

```
                let shape_rhs = rhs.shape();

                // Compute the gradient of the output using the already
existing `relu_backward`
                // function in the basic Burn backend trait.
                let grad_output = B::relu_backward(output, grad);

                // Compute the lhs gradient, which is the derivative of matmul
with support for
                // broadcasting.
                let grad_lhs = broadcast_shape::<B>(
                    B::float_matmul(grad_output.clone(),
B::float_transpose(rhs)),
                    &shape_lhs,
                );
                // Compute the rhs gradient, which is the derivative of matmul
with support for
                // broadcasting.
                let grad_rhs = broadcast_shape::<B>(
                    B::float_matmul(B::float_transpose(lhs),
grad_output.clone()),
                    &shape_rhs,
                );
                // The add derivative is only 1, so we just need to support
broadcasting to
                // compute the bias gradient.
                let grad_bias = broadcast_shape::<B>(grad_output, &shape_bias);

                // Register the gradient for each variable based on whether
they are marked as
                // `tracked`.
                if let Some(node) = node_bias {
                    grads.register::<B>(node.id, grad_bias);
                }
                if let Some(node) = node_lhs {
                    grads.register::<B>(node.id, grad_lhs);
                }
                if let Some(node) = node_rhs {
                    grads.register::<B>(node.id, grad_rhs);
                }
            }
        }

        // Prepare a stateful operation with each variable node and
corresponding graph.
        //
        // Each node can be fetched with `ops.parents` in the same order as
defined here.
        match FusedMatmulAddReluBackward
            .prepare::<C>([lhs.node.clone(), rhs.node.clone(),
bias.node.clone()])
            // Marks the operation as compute bound, meaning it will save its
            // state instead of recomputing itself during checkpointing
```

```
            .compute_bound()
            .stateful()
        {
            OpsKind::Tracked(mut prep) => {
                // When at least one node is tracked, we should register our
backward step.

                // The state consists of what will be needed for this
operation's backward pass.
                // Since we need the parents' outputs, we must checkpoint their
ids to retrieve their node
                // output at the beginning of the backward. We can also save
utilitary data such as the bias shape
                // If we also need this operation's output, we can either save
it in the state or recompute it
                // during the backward pass. Here we choose to save it in the
state because it's a compute bound operation.
                let lhs_state = prep.checkpoint(&lhs);
                let rhs_state = prep.checkpoint(&rhs);
                let bias_shape = bias.primitive.shape();

                let output = B::fused_matmul_add_relu(
                    lhs.primitive.clone(),
                    rhs.primitive.clone(),
                    bias.primitive,
                );

                let state = (lhs_state, rhs_state, output.clone(), bias_shape);

                prep.finish(state, output)
            }
            OpsKind::UnTracked(prep) => {
                // When no node is tracked, we can just compute the original
operation without
                // keeping any state.
                let output = B::fused_matmul_add_relu(lhs.primitive,
rhs.primitive, bias.primitive);
                prep.finish(output)
            }
        }
    }
}
```

The previous code is self-documented to make it clearer, but here is what it does in summary.

We define `fused_matmul_add_relu` within `Autodiff<B>`, allowing any autodiff-decorated backend to benefit from our implementation. In an autodiff-decorated backend, the forward pass must still be implemented. This is achieved using a comprehensive match statement block where computation is delegated to the inner backend, while keeping track of a state.

The state comprises any information relevant to the backward pass, such as input and output tensors, along with the bias shape. When an operation isn't tracked (meaning there won't be a backward pass for this specific operation in the graph), storing a state becomes unnecessary, and we simply perform the forward computation.

The backward pass uses the gradient obtained from the preceding node in the computation graph. It calculates the derivatives for `relu` (`relu_backward`), add (no operation is required here, as the derivative is one), and `matmul` (another `matmul` with transposed inputs). This results in gradients for both input tensors and the bias, which are registered for consumption by subsequent operation nodes.

The only remaining part is to implement our autodiff-decorated backend trait for our WGPU Backend.

```
impl<G: GraphicsApi, F: FloatElement, I: IntElement> AutodiffBackend for
Autodiff<WgpuBackend<G, F, I>>
{
}
```

## Conclusion

In this guide, we've implemented a fused kernel using the WGPU backend, enabling execution on any GPU. By delving into the inner workings of both the WGPU backend and the autodiff backend, we've gained a deeper understanding of these systems.

While extending a backend may be harder than working with straightforward tensors, the benefits can be worth it. This approach enables the crafting of custom models with greater control over execution, which can potentially greatly enhance the performance of your models.

As we conclude this guide, we hope that you have gained insights into Burn's world of backend extensions, and that it will help you to unleash the full potential of your projects.

# WebAssembly

Burn supports WebAssembly (WASM) execution using the `NdArray` and `WebGpu` backends, allowing models to run directly in the browser.

Check out the following examples:

- Image Classification Web
- MNIST Inference on Web

When targeting WebAssembly, certain dependencies require additional configuration. In particular, the `getrandom` crate requires explicit setting when using `WebGpu` .

Following the recommended usage, make sure to explicitly add the dependency with the `wasm_js` feature flag for your project.

```
[dependencies]
getrandom = { version = "0.3.2", default-features = false, features = [
    "wasm_js",
] }
```

You also need to set the `getrandom_backend` accordingly via the rust-flags. The flag can either be set by specifying the `rustflags` field in `.cargo/config.toml`

```
[target.wasm32-unknown-unknown]
rustflags = ['--cfg', 'getrandom_backend="wasm_js"']
```

Or by using the `RUSTFLAGS` environment variable:

```
RUSTFLAGS='--cfg getrandom_backend="wasm_js"'
```

This change is now explicitly required with latest versions of Burn, following the `getrandom` recommendations. This avoids potential issues for WASM developers who do not target Web targets.

# No Standard Library

In this section, you will learn how to run an onnx inference model on an embedded system, with no standard library support on a Raspberry Pi Pico. This should be universally applicable to other platforms. All the code can be found under the examples directory.

## Step-by-Step Guide

Let's walk through the process of running an embedded ONNX model:

### Setup

Follow the embassy guide for your specific environment. Once setup, you should have something similar to the following.

```
./inference
├── Cargo.lock
├── Cargo.toml
├── build.rs
├── memory.x
└── src
    └── main.rs
```

Some other dependencies have to be added

```
[dependencies]
embedded-alloc = "0.6.0" # Only if there is no default allocator for your chip
burn = { version = "0.18", default-features = false, features = ["ndarray"] } #
Backend must be ndarray

[build-dependencies]
burn-import = { version = "0.18" } # Used to auto generate the rust code to
import the model
```

### Import the Model

Follow the directions to import models.

Use the following ModelGen config

```
ModelGen::new()
    .input(my_model)
    .out_dir("model/")
    .record_type(RecordType::Bincode)
    .embed_states(true)
    .run_from_script();
```

## Global Allocator

First define a global allocator (if you are on a no_std system without alloc).

```
use embedded_alloc::LlffHeap as Heap;

#[global_allocator]
static HEAP: Heap = Heap::empty();

#[embassy_executor::main]
async fn main(_spawner: Spawner) {
    {
        use core::mem::MaybeUninit;
        const HEAP_SIZE: usize = 100 * 1024; // This is dependent on the model
size in memory.
        static mut HEAP_MEM: [MaybeUninit<u8>; HEAP_SIZE] =
[MaybeUninit::uninit(); HEAP_SIZE];
        unsafe { HEAP.init(&raw mut HEAP_MEM as usize, HEAP_SIZE) }
    }
}
```

## Define Backend

We are using ndarray, so we just need to define the NdArray backend as usual

```
use burn::{backend::NdArray, tensor::Tensor};

type Backend = NdArray<f32>;
type BackendDevice = <Backend as burn::tensor::backend::Backend>::Device;
```

Then inside the `main` function add

```
use your_model::Model;

// Get a default device for the backend
let device = BackendDevice::default();

// Create a new model and load the state
let model: Model<Backend> = Model::default();
```

### Running the Model

To run the model, just call it as you would normally

```
// Define the tensor
let input = Tensor::<Backend, 2>::from_floats([[input]], &device);

// Run the model on the input
let output = model.forward(input);
```

## Conclusion

Running a model in a no_std environment is pretty much identical to a normal environment. All that is needed is a global allocator.