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 a 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.

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:  BestAvailable,
  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.

Running examples

Many additional Burn examples available in the examples directory. To run one, please refer to the example's README.md for the specific command to execute.

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. 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 my-first-burn-model
```

As mentioned previously, this will initialize your `my-first-burn-model` project directory with a `Cargo.toml` and a `src/main.rs` file.

In the `Cargo.toml` file, add the `burn` dependency with `train` and `wgpu` features.

```
[package]
name = "my-first-burn-model"
version = "0.1.0"
edition = "2021"

[dependencies]
burn = { version = "0.12.1", features = ["train", "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::{
    config::Config,
    module::Module,
    nn::{
        conv::{Conv2d, Conv2dConfig},
        pool::{AdaptiveAvgPool2d, AdaptiveAvgPool2dConfig},
        Dropout, DropoutConfig, Linear, LinearConfig, ReLU,
    },
    tensor::{backend::Backend, Tensor},
};

#[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**

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(),
        }
    }
}
```

▶ 🦀 **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 as input values for the parameter 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 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.

```rust
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:

```rust
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. In particular, there are Hugging Face dataset utilities that allow to download and store data from Hugging Face into an SQLite database for extremely efficient data streaming and storage. For this guide, we will use the MNIST dataset provided by Hugging Face.

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.

```rust
use burn::{
    data::{dataloader::batcher::Batcher, dataset::vision::MNISTItem},
    tensor::{backend::Backend, Data, ElementConversion, Int, Tensor},
};

pub struct MNISTBatcher<B: Backend> {
    device: B::Device,
}

impl<B: Backend> MNISTBatcher<B> {
    pub fn new(device: B::Device) -> Self {
        Self { device }
    }
}
```

This codeblock defines a batcher struct with the device in which the tensor should be sent before being passed to the model. Note that the device is an associative type of the `Backend` trait since not all backends expose the same devices. 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<MNISTItem, MNISTBatch<B>> for MNISTBatcher<B> {
    fn batch(&self, items: Vec<MNISTItem>) -> MNISTBatch<B> {
        let images = items
            .iter()
            .map(|item| Data::<f32, 2>::from(item.image))
            .map(|data| Tensor::<B, 2>::from_data(data.convert(), &self.device))
            .map(|tensor| tensor.reshape([1, 28, 28]))
            // Normalize: make 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(
                Data::from([(item.label as i64).elem()]),
                &self.device
            ))
            .collect();

        let images = Tensor::cat(images, 0).to_device(&self.device);
        let targets = Tensor::cat(targets, 0).to_device(&self.device);

        MNISTBatch { images, targets }
    }
}
```

▶ 🦀 **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 `Data` struct. Burn provides the `Data` 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. 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 = CrossEntropyLoss::new(None).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.

```rust
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,
}

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

    B::seed(config.seed);

    let batcher_train = MNISTBatcher::<B>::new(device.clone());
    let batcher_valid = MNISTBatcher::<B::InnerBackend>::new(device.clone());

    let dataloader_train = DataLoaderBuilder::new(batcher_train)
        .batch_size(config.batch_size)
        .shuffle(config.seed)
        .num_workers(config.num_workers)
        .build(MNISTDataset::train());

    let dataloader_test = DataLoaderBuilder::new(batcher_valid)
        .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])
        .num_epochs(config.num_epochs)
        .build(
            config.model.init::<B>(),
            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 then initialize our dataloaders using our previously created batcher. Since no automatic differentiation is needed during the validation phase, the backend used for the corresponding batcher is `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, and the only remaining task is saving the trained weights using the `CompactRecorder`. This recorder employs the `MessagePack` format with `gzip` compression, `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`.

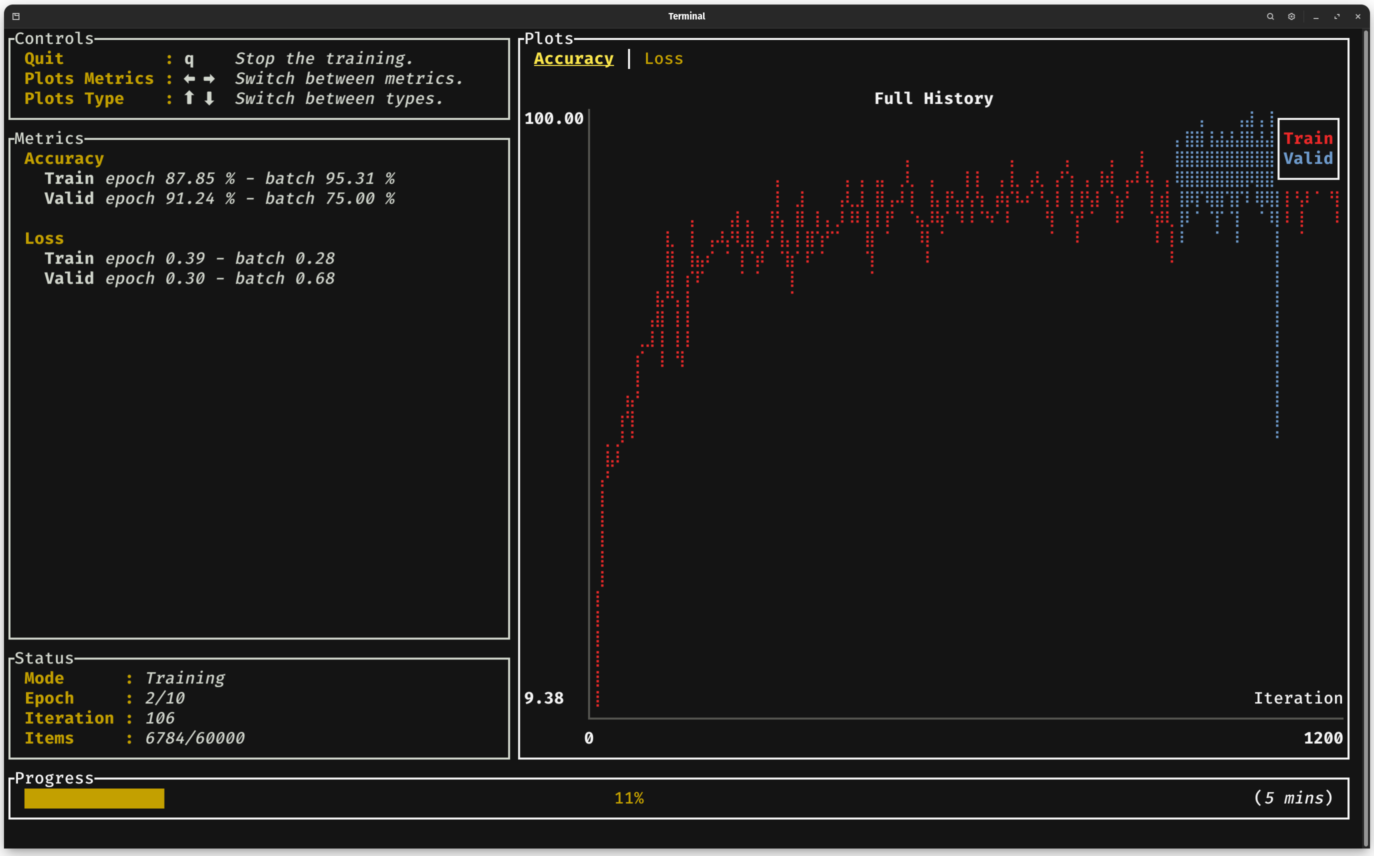
```rust
use burn::optim::AdamConfig;
use burn::backend::{Autodiff, Wgpu, wgpu::AutoGraphicsApi};
use guide::model::ModelConfig;

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

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

▶ 🦀 **Packages, Crates and Modules**

In this example, 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. By leaving the graphics API as `AutoGraphicsApi`, it should automatically use an API available on your machine. 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.

When running the example, we can see the training progression through a basic CLI dashboard:



```
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

Progress
                         11%                          (5 mins)
```

Inference

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

For loading a model primed for inference, it is of course more efficient to directly load the weights into the model, bypassing the need to initially set arbitrary weights or worse, weights computed from a Xavier normal initialization only to then promptly replace them with the stored weights. With that in mind, let's create a new initialization function receiving the record as input. This new function can be defined alongside the `init` function for the `ModelConfig` struct in `src/model.rs`.

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

It is important to note that the `ModelRecord` was automatically generated thanks to the `Module` trait. It allows us to load the module state without having to deal with fetching the correct type manually. Everything is validated when loading the model with the record.

Now let's create a simple `infer` method in a new file `src/inference.rs` which we will use to load our trained model.

```rust
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");
    let record = CompactRecorder::new()
        .load(format!("{artifact_dir}/model").into(), &device)
        .expect("Trained model should exist");

    let model = config.model.init_with::<B>(record);

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

    println!("Predicted {} Expected {}", predicted, 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 before sending it to the wanted device for inference. 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!

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:

```
// correct: Tensor is 1-Dimensional with 5 elements
let tensor_1 = Tensor::<Backend, 1>::from_floats([1.0, 2.0, 3.0, 4.0, 5.0]);

// incorrect: let tensor_1 = Tensor::<Backend, 5>::from_floats([1.0, 2.0, 3.0,
4.0, 5.0]);
// 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 `Data` struct as input. The `Data` struct has two fields: value & shape. 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]);

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

// Initialization using from_floats (Recommended for f32 ElementType)
// Will be converted to Data internally. `.convert()` not needed as from_floats()
defined for fixed ElementType
let tensor_3 = Tensor::<Backend, 1>::from_floats([1.0, 2.0, 3.0]);

// 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(Data::from(&arr[0..3]).convert());

// Initialization from a custom type

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

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

The `.convert()` method for Data struct is called to ensure that the data's primitive type is consistent across all backends. With `.from_floats()` method the ElementType is fixed as f32 and therefore no convert operation is required across backends. This operation can also be done at element wise level as: `let tensor_6 = Tensor::<B, 1, Int>::from_data(Data::from([(item.age as i64).elem()])`. The `ElementConversion` trait however needs to be imported for the element wise operation.

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]);
    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]);
    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::empty(shape, device)`	`torch.empty(shape, device=device)`
`tensor.dims()`	`tensor.size()`
`tensor.shape()`	`tensor.shape`
`tensor.reshape(shape)`	`tensor.view(shape)`
`tensor.flatten(start_dim, end_dim)`	`tensor.flatten(start_dim, end_dim)`
`tensor.squeeze(dim)`	`tensor.squeeze(dim)`
`tensor.unsqueeze()`	`tensor.unsqueeze(0)`
`tensor.unsqueeze_dim(dim)`	`tensor.unsqueeze(dim)`
`tensor.slice(ranges)`	`tensor[(*ranges,)]`
`tensor.slice_assign(ranges, values)`	`tensor[(*ranges,)] = values`
`tensor.narrow(dim, start, length)`	`tensor.narrow(dim, start, length)`
`tensor.chunk(num_chunks, dim)`	`tensor.chunk(num_chunks, dim)`
`tensor.device()`	`tensor.device`
`tensor.to_device(device)`	`tensor.to(device)`
`tensor.repeat(2, 4)`	`tensor.repeat([1, 1, 4])`
`tensor.equal(other)`	`x == y`
`Tensor::cat(tensors, dim)`	`torch.cat(tensors, dim)`
`tensor.into_data()`	N/A
`tensor.to_data()`	N/A
`Tensor::from_data(data, device)`	N/A
`tensor.into_primitive()`	N/A
`Tensor::from_primitive(primitive)`	N/A
`Tensor::stack(tensors, dim)`	`torch.stack(tensors, dim)`

Numeric Operations

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

Burn	PyTorch Equivalent
`tensor.into_scalar()`	`tensor.item()` (for single-element tensors)
`tensor + other` or `tensor.add(other)`	`tensor + other`
`tensor + scalar` or `tensor.add_scalar(scalar)`	`tensor + scalar`
`tensor - other` or `tensor.sub(other)`	`tensor - other`
`tensor - scalar` or `tensor.sub_scalar(scalar)`	`tensor - scalar`
`tensor / other` or `tensor.div(other)`	`tensor / other`
`tensor / scalar` or `tensor.div_scalar(scalar)`	`tensor / scalar`
`tensor * other` or `tensor.mul(other)`	`tensor * other`
`tensor * scalar` or `tensor.mul_scalar(scalar)`	`tensor * scalar`
`tensor.powf(other)` or `tensor.powi(intother)`	`tensor.pow(other)`
`tensor.powf_scalar(scalar)` or `tensor.powi_scalar(intscalar)`	`tensor.pow(scalar)`
`-tensor` or `tensor.neg()`	`-tensor`
`Tensor::zeros(shape)`	`torch.zeros(shape)`
`Tensor::zeros(shape, device)`	`torch.zeros(shape, device=device)`
`Tensor::ones(shape, device)`	`torch.ones(shape, device=device)`
`Tensor::full(shape, fill_value, device)`	`torch.full(shape, fill_value, device=device)`
`tensor.mean()`	`tensor.mean()`
`tensor.sum()`	`tensor.sum()`
`tensor.mean_dim(dim)`	`tensor.mean(dim)`
`tensor.sum_dim(dim)`	`tensor.sum(dim)`
`tensor.equal_elem(other)`	`tensor.eq(other)`
`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)`

Burn	PyTorch Equivalent
`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_where(mask, value_tensor)`	`torch.where(mask, value_tensor, tensor)`
`tensor.mask_fill(mask, value)`	`tensor.masked_fill(mask, value)`
`tensor.gather(dim, indices)`	`torch.gather(tensor, dim, indices)`
`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.argmax(dim)`	`tensor.argmax(dim)`
`tensor.max()`	`tensor.max()`
`tensor.max_dim(dim)`	`tensor.max(dim)`
`tensor.max_dim_with_indices(dim)`	N/A
`tensor.argmin(dim)`	`tensor.argmin(dim)`
`tensor.min()`	`tensor.min()`
`tensor.min_dim(dim)`	`tensor.min(dim)`
`tensor.min_dim_with_indices(dim)`	N/A
`tensor.clamp(min, max)`	`torch.clamp(tensor, min=min, max=max)`
`tensor.clamp_min(min)`	`torch.clamp(tensor, min=min)`
`tensor.clamp_max(max)`	`torch.clamp(tensor, max=max)`
`tensor.abs()`	`torch.abs(tensor)`
`tensor.triu(diagonal)`	`torch.triu(tensor, diagonal)`
`tensor.tril(diagonal)`	`torch.tril(tensor, diagonal)`

Float Operations

Those operations are only available for `Float` tensors.

Burn API	PyTorch Equivalent
`tensor.exp()`	`tensor.exp()`
`tensor.log()`	`tensor.log()`
`tensor.log1p()`	`tensor.log1p()`
`tensor.erf()`	`tensor.erf()`
`tensor.sqrt()`	`tensor.sqrt()`
`tensor.recip()`	`tensor.reciprocal()`
`tensor.cos()`	`tensor.cos()`
`tensor.sin()`	`tensor.sin()`
`tensor.tanh()`	`tensor.tanh()`
`tensor.from_floats(floats, device)`	N/A
`tensor.int()`	Similar to `tensor.to(torch.long)`
`tensor.zeros_like()`	`torch.zeros_like(tensor)`
`tensor.ones_like()`	`torch.ones_like(tensor)`
`tensor.random_like(distribution)`	`torch.rand_like()` only uniform
`tensor.one_hot(index, num_classes, device)`	N/A
`tensor.transpose()`	`tensor.T`
`tensor.swap_dims(dim1, dim2)`	`tensor.transpose(dim1, dim2)`
`tensor.matmul(other)`	`tensor.matmul(other)`
`tensor.var(dim)`	`tensor.var(dim)`
`tensor.var_bias(dim)`	N/A
`tensor.var_mean(dim)`	N/A
`tensor.var_mean_bias(dim)`	N/A
`tensor.random(shape, distribution, device)`	N/A
`tensor.to_full_precision()`	`tensor.to(torch.float)`
`tensor.from_full_precision(tensor)`	N/A

Int Operations

Those operations are only available for `Int` tensors.

Burn API	PyTorch Equivalent
`tensor.from_ints(ints)`	N/A
`tensor.float()`	Similar to `tensor.to(torch.float)`

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)`

Bool Operations

Those operations are only available for `Bool` tensors.

Burn API	PyTorch Equivalent
`tensor.float()`	Similar to `tensor.to(torch.float)`
`tensor.int()`	Similar to `tensor.to(torch.long)`
`tensor.not()`	`tensor.logical_not()`

Activation Functions

Burn API	PyTorch Equivalent
`activation::gelu(tensor)`	Similar to `nn.functional.gelu(tensor)`
`activation::log_sigmoid(tensor)`	Similar to `nn.functional.log_sigmoid(tensor)`
`activation::log_softmax(tensor, dim)`	Similar to `nn.functional.log_softmax(tensor, dim)`
`activation::mish(tensor)`	Similar to `nn.functional.mish(tensor)`
`activation::quiet_softmax(tensor, dim)`	Similar to `nn.functional.quiet_softmax(tensor, dim)`
`activation::relu(tensor)`	Similar to `nn.functional.relu(tensor)`
`activation::sigmoid(tensor)`	Similar to `nn.functional.sigmoid(tensor)`
`activation::silu(tensor)`	Similar to `nn.functional.silu(tensor)`
`activation::softmax(tensor, dim)`	Similar to `nn.functional.softmax(tensor, dim)`
`activation::softplus(tensor, beta)`	Similar to `nn.functional.softplus(tensor, beta)`
`activation::tanh(tensor)`	Similar to `nn.functional.tanh(tensor)`

Autodiff

Burn's tensor also supports autodifferentiation, 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 autodifferentiation 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.

```rust
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::nn;
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 tensor in the module.
    fn visit<const D: usize>(&mut self, id: &ParamId, tensor: &Tensor<B, D>);
}

/// Module mapper trait.
pub trait ModuleMapper<B: Backend> {
    /// Map a tensor in the module.
    fn map<const D: usize>(&mut self, id: &ParamId, tensor: Tensor<B, D>) ->
Tensor<B, D>;
}
```

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.
`LayerNorm`	`nn.LayerNorm`
`GroupNorm`	`nn.GroupNorm`
`Dropout`	`nn.Dropout`
`GELU`	`nn.GELU`
`Linear`	`nn.Linear`

Burn API	PyTorch Equivalent
`Embedding`	`nn.Embedding`
`Relu`	`nn.ReLU`

Convolutions

Burn API	PyTorch Equivalent
`Conv1d`	`nn.Conv1d`
`Conv2d`	`nn.Conv2d`
`ConvTranspose1d`	`nn.ConvTranspose1d`
`ConvTranspose2d`	`nn.ConvTranspose2d`

Pooling

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`	`nn.LSTM`
`GateController`	*No direct equivalent*

Transformer

Burn API	PyTorch Equivalent
`MultiHeadAttention`	`nn.MultiheadAttention`

Burn API	PyTorch Equivalent
`TransformerDecoder`	`nn.TransformerDecoder`
`TransformerEncoder`	`nn.TransformerEncoder`
`PositionalEncoding`	*No direct equivalent*

Loss

Burn API	PyTorch Equivalent
`CrossEntropyLoss`	`nn.CrossEntropyLoss`
`MSELoss`	`nn.MSELoss`

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

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
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.

```
/// Metric trait.
///
/// 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.
///
/// The only exception is for metrics that don't need any input, setting the
associated type
/// to the null type `()`.
pub trait Metric: Send + Sync {
    /// The input type of the metric.
    type Input;

    /// Updates 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 loss =
f64::from_elem(loss.tensor.clone().mean().into_data().value[0]);

        self.state
            .update(loss, 1, FormatOptions::new("Loss").precision(2))
    }

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

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
#[derive(Config)]
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 pattern it is easy to create instances from this config. Therefore, initialization methods should be implemented on the config struct.

```rust
impl MyModuleConfig {
    /// Create a module with random weights.
    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(),
        }
    }

    /// Create a module with a record, for inference and fine-tuning.
    pub fn init_with(&self, record: MyModuleRecord<B>) -> MyModule {
        MyModule {
            linear: LinearConfig::new(
                self.d_model,
                self.d_ff,
            ).init_with(record.linear),
            dropout: DropoutConfig::new(self.dropout).init(),
        }
    }
}
```

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

```rust
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 Message Park	None
NamedMpkFileRecorder	File - Named Message Park	None

Recorder	Format	Compression
NamedMpkGzFileRecorder	File - Named Message Park	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 message park 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

Most deep learning training being done on datasets –with perhaps the exception of reinforcement learning–, it is essential to provide a convenient and performant API. 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.
`ComposedDataset`	Composes multiple datasets together to create a larger one without copying any data.

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 dataset_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.

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.

Sources

For now, there is only one dataset source 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.

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.

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(device);
    let mut optim = config.optimizer.init();

    // Create the batcher.
    let batcher_train = MNISTBatcher::<B>::new(device.clone());
    let batcher_valid = MNISTBatcher::<B::InnerBackend>::new(device.clone());

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

    let dataloader_test = DataLoaderBuilder::new(batcher_valid)
        .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:

```rust
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).forward(output.clone(),
batch.targets.clone());
            let accuracy = accuracy(output, batch.targets);

            println!(
                "[Train - Epoch {} - Iteration {}] Loss {:.3} | Accuracy {:.3} %",
                iteration,
                epoch,
                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).forward(output.clone(),
batch.targets.clone());
            let accuracy = accuracy(output, batch.targets);

            println!(
                "[Valid - Epoch {} - Iteration {}] Loss {} | Accuracy {}",
                iteration,
                epoch,
                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`.

```rust
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.

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 smp_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
  .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

While the first approach is very straightforward, it does require the model to already be initialized. If instead you would like to skip the initialization and directly load the weights into the modules of your model, you can create a new initialization function. Let's take the following model definition as a simple example.

```
#[derive(Module, Debug)]
pub struct Model<B: Backend> {
    linear_in: Linear<B>,
    linear_out: Linear<B>,
    activation: ReLU,
}
```

Similar to the basic workflow inference example, we can define a new initialization function which initializes the different parts of our model with the record values.

```
impl<B: Backend> Model<B> {
    /// Returns the initialized model using the recorded weights.
    pub fn init_with(record: ModelRecord<B>) -> Model<B> {
        Model {
            linear_in: LinearConfig::new(10, 64).init_with(record.linear_in),
            linear_out: LinearConfig::new(64, 2).init_with(record.linear_out),
            activation: ReLU::new(),
        }
    }

    /// Returns the dummy model with randomly initialized weights.
    pub fn new(device: &Device<B>) -> Model<B> {
        let l1 = LinearConfig::new(10, 64).init(device);
        let l2 = LinearConfig::new(64, 2).init(device);
        Model {
            linear_in: l1,
            linear_out: l2,
            activation: ReLU::new(),
        }
    }
}
```

Now, let's save a model that we can load later. In the following snippets, we use `type MyBackend = NdArray<f32>` but you can use whatever backend you like.

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

// Directly initialize a new model with the loaded record/weights
let model = Model::init_with(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.

```
// 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

The Burn project supports the import of models from various frameworks, emphasizing efficiency and compatibility. Currently, it handles two primary model formats:

1. ONNX: Facilitates direct import, ensuring the model's performance and structure are maintained.

2. PyTorch: Enables the loading of PyTorch model weights into Burn's native model architecture, ensuring seamless integration.

Import ONNX Model

Why Importing Models is Necessary

In the realm of deep learning, it's common to switch between different frameworks depending on your project's specific needs. Maybe you've painstakingly fine-tuned a model in TensorFlow or PyTorch and now you want to reap the benefits of Burn's unique features for deployment or further testing. This is precisely the scenario where importing models into Burn can be a game-changer.

Traditional Methods: The Drawbacks

If you've been working with other deep learning frameworks like PyTorch, it's likely that you've exported model weights before. PyTorch, for instance, lets you save model weights using its `torch.save()` function. Yet, to port this model to another framework, you face the arduous task of manually recreating the architecture in the destination framework before loading in the weights. Not only is this method tedious, but it's also error-prone and hinders smooth interoperability between frameworks.

It's worth noting that for models using cutting-edge, framework-specific features, manual porting might be the only option, as standards like ONNX might not yet support these new innovations.

Enter ONNX

ONNX (Open Neural Network Exchange) is designed to solve such complications. It's an open-standard format that exports both the architecture and the weights of a deep learning model. This feature makes it exponentially easier to move models between different frameworks, thereby significantly aiding interoperability. ONNX is supported by a number of frameworks including but not limited to TensorFlow, PyTorch, Caffe2, and Microsoft Cognitive Toolkit.

Advantages of ONNX

ONNX stands out for encapsulating two key elements:

1. **Model Information**: It captures the architecture, detailing the layers, their connections, and configurations.
2. **Weights**: ONNX also contains the trained model's weights.

This dual encapsulation not only simplifies the porting of models between frameworks but also allows seamless deployment across different environments without compatibility concerns.

Burn's ONNX Support: Importing Made Easy

Understanding the important role that ONNX plays in the contemporary deep learning landscape, Burn simplifies the process of importing ONNX models via an intuitive API designed to mesh well with Burn's ecosystem.

Burn's solution is to translate ONNX files into Rust source code as well as Burn-compatible weights. This transformation is carried out through the burn-import crate's code generator during build time, providing advantages for both executing and further training ONNX models.

Advantages of Burn's ONNX Approach

1. **Native Integration**: The generated Rust code is fully integrated into Burn's architecture, enabling your model to run on various backends without the need for a separate ONNX runtime.

2. **Trainability**: The imported model is not just for inference; it can be further trained or fine-tuned using Burn's native training loop.

3. **Portability**: As the model is converted to Rust source code, it can be compiled into WebAssembly for browser execution. Likewise, this approach is beneficial for no-std embedded devices.

4. **Optimization**: Rust's compiler can further optimize the generated code for target architectures, thereby improving performance.

Sample Code for Importing ONNX Model

Below is a step-by-step guide to importing an ONNX model into a Burn-based project:

Step 1: Update `build.rs`

Include the `burn-import` crate and use the following Rust code in your `build.rs`:

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

fn main() {
    // Generate Rust code from the ONNX model file
    ModelGen::new()
        .input("src/model/mnist.onnx")
        .out_dir("model/")
        .run_from_script();
}
```

Step 2: Modify `mod.rs`

Add this code to the `mod.rs` file located in `src/model`:

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

Step 3: Utilize Imported Model

Here's how to use the imported model in your application:

```rust
mod model;

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

fn main() {
    // Initialize a new model instance
    let device = NdArrayDevice::default();
    let model: Model<NdArray<f32>> = Model::new(&device);

    // Create a sample input tensor (zeros for demonstration)
    let input = tensor::Tensor::<NdArray<f32>, 4>::zeros([1, 1, 28, 28], &device);

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

    // Print the output
    println!("{:?}", output);
}
```

Working Examples

For practical examples, please refer to:

1. MNIST Inference Example
2. SqueezeNet Image Classification

By combining ONNX's robustness with Burn's unique features, you'll have the flexibility and power to streamline your deep learning workflows like never before.

🚨**Note**: `burn-import` crate is in active development and currently supports a limited set of ONNX operators.

PyTorch Model

Introduction

Whether you've trained your model in PyTorch or you want to use a pre-trained model from PyTorch, you can import them into Burn. Burn supports importing PyTorch model weights with `.pt` file extension. Compared to ONNX models, `.pt` files only contain the weights of the model, so you will need to reconstruct the model architecture in Burn.

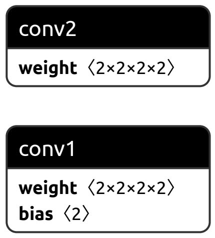
How to export a PyTorch model

If you have a PyTorch model that you want to import into Burn, you will need to export it first, unless you are using a pre-trained published model. To export a PyTorch model, you can use the `torch.save` function.

Here is an example of how to export 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

def main():
    torch.manual_seed(42)  # To make it reproducible
    model = Net().to(torch.device("cpu"))
    model_weights = model.state_dict()
    torch.save(model_weights, "conv2d.pt")
```

Use Netron to view the exported model. You should see something like this:



How to import a PyTorch model

1. Define the model in Burn:

```rust
use burn::{
    module::Module,
    nn::conv::{Conv2d, Conv2dConfig},
    tensor::{backend::Backend, Tensor},
};

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

impl<B: Backend> Net<B> {
    /// Create a new model from the given record.
    pub fn new_with(record: NetRecord<B>) -> Self {
        let conv1 = Conv2dConfig::new([2, 2], [2, 2])
            .init_with(record.conv1);
        let conv2 = Conv2dConfig::new([2, 2], [2, 2])
            .with_bias(false)
            .init_with(record.conv2);
        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)
    }
}
```

2. Load the model weights from the exported PyTorch model (2 options):

   a) *Dynamically*, but this requires burn-import runtime 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();
    let record = PyTorchFileRecorder::<FullPrecisionSettings>::default()
        .load("./conv2d.pt".into(), &device)
        .expect("Should decode state successfully");

    let model = model::Net::<Backend>::new_with(record);
}
```

   b) *Pre-converted* to Burn's binary format:

```rust
// Convert the PyTorch model to Burn's binary format in
// build.rs or in a separate executable. Then, include the generated file
// in your project. See `examples/pytorch-import` for an example.

use crate::model;

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

type Backend = burn_ndarray::NdArray<f32>;

fn main() {
    let device = Default::default();
    let recorder = PyTorchFileRecorder::<FullPrecisionSettings>::default()
    let record: model::NetRecord<B> = recorder
        .load("./conv2d.pt".into(), &device)
        .expect("Should decode state successfully");

    // Save the model record to a file.
    let recorder = NamedMpkFileRecorder::<FullPrecisionSettings>::default();
    recorder
        .record(record, "MY_FILE_OUTPUT_PATH".into())
        .expect("Failed to save model record");
}

/// Load the model from the file in your source code (not in build.rs or
script).
fn load_model() -> Net::<Backend> {
    let device = Default::default();
    let record = NamedMpkFileRecorder::<FullPrecisionSettings>::default()
        .load("./MY_FILE_OUTPUT_PATH".into(), &device)
        .expect("Should decode state successfully");

    Net::<Backend>::new_with(record)
}
```

Troubleshooting

Adjusting the source model architecture

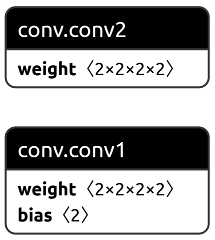
If your target model differs structurally from the model you exported, `PyTorchFileRecorder` allows changing the attribute names and the order of the attributes. For example, if you exported a model with the following structure:

```python
class ConvModule(nn.Module):
    def __init__(self):
        super(ConvModule, 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

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv = ConvModule()

    def forward(self, x):
        x = self.conv(x)
        return x
```

But you need to import it into a model with the following structure:

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

Which produces the following weights structure (viewed in Netron):



You can use the `PyTorchFileRecorder` to change the attribute names and the order of the attributes by specifying a regular expression (See regex::Regex::replace) to match the attribute name and a replacement string in `LoadArgs`:

```
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>::new_with(record);
```

Loading the model weights to a partial model

`PyTorchFileRecorder` enables selective weight loading into partial models. For instance, in a model with both an encoder and a decoder, it's possible to load only the encoder weights. This is done by defining the encoder in Burn, allowing the loading of its weights while excluding the decoder's.

Current known issues

1. Candle's pickle library does not currently function on Windows due to a Candle bug.
2. Candle's pickle does not currently unpack boolean tensors.

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.

```rust
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:

```rust
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 specificity of each implementation will be covered by the examples provided in this section. Currently, we only have one example, but more are yet to come!

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. 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.

```rust
/// We use a type alias for better readability.
pub type FloatTensor<B, const D: usize> = <B as
burn::tensor::backend::Backend>::TensorPrimitive<D>;

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

/// 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(),
        rhs.into_primitive(),
        bias.into_primitive(),
    );

    Tensor::from_primitive(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 thread to the data it is responsible of, with support for batches.

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

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

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

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

@group(0)
@binding(4)
var<storage, read> 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.

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

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

// Implement the dynamic kernel trait for our kernel type.
impl<E: FloatElement> DynamicKernel for FusedMatmulAddRelu<E> {
    fn source_template(self) -> SourceTemplate {
        // Extend our raw kernel with workgroup size information using the
        // `SourceTemplate` trait.
        FusedMatmulAddReluRaw::source_template()
            .register("workgroup_size_x", self.workgroup_size_x.to_string())
            .register("workgroup_size_y", self.workgroup_size_y.to_string())
            .register("elem", E::type_name())
    }

    fn id(&self) -> String {
        format!("{:?}", self)
    }
}
```

Subsequently, we'll go into implementing our custom backend trait for the WGPU backend.

```
/// Implement our custom backend trait for the existing backend `Wgpu`.
impl<G: GraphicsApi, F: FloatElement, I: IntElement> Backend for Wgpu<G, F, I> {
    fn fused_matmul_add_relu<const D: usize>(
        lhs: FloatTensor<Self, D>,
        rhs: FloatTensor<Self, D>,
        bias: FloatTensor<Self, D>,
    ) -> WgpuTensor<F, D> {
        // Define workgroup size, hardcoded for simplicity.
        let workgroup_size_x = 16;
        let workgroup_size_y = 16;

        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 num_rows = lhs.shape.dims[D - 2];
        let num_cols = rhs.shape.dims[D - 1];

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

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

        // Create the output tensor primitive.
        let output = WgpuTensor::new(lhs.context.clone(), shape_out, buffer);

        // Create the kernel.
        let kernel = FusedMatmulAddRelu::<F>::new(workgroup_size_x,
workgroup_size_y);

        // Build info buffer with tensor information needed by the kernel, such as
shapes and strides.
        let info = build_info(&[&lhs, &rhs, &output]);
        let info_buffer = lhs
            .context
            .create_buffer_with_data(bytemuck::cast_slice(&info));
        // Declare the wgsl workgroup with the number of blocks in x, y and z.
        let blocks_needed_in_x = f32::ceil(num_rows as f32 / workgroup_size_x as
f32) as u32;
        let blocks_needed_in_y = f32::ceil(num_cols as f32 / workgroup_size_y as
f32) as u32;
        let workgroup = WorkGroup::new(blocks_needed_in_x, blocks_needed_in_y,
num_batches as u32);

        // Execute lazily the kernel with the launch information and the given
buffers.
        lhs.client.execute(
            Box::new(DynamicKernel::new(kernel, workgroup)),
            &[
                &lhs.handle,
                &rhs.handle,
                &bias.handle,
                &output.handle,
                &info_handle,
            ],
        );

        // 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> Backend for Autodiff<B> {
    fn fused_matmul_add_relu<const D: usize>(
        lhs: FloatTensor<Self, D>,
        rhs: FloatTensor<Self, D>,
        bias: FloatTensor<Self, D>,
    ) -> FloatTensor<Self, D> {
        // Create our zero-sized type that will implement the Backward trait.
        #[derive(Debug)]
        struct FusedMatmulAddReluBackward<const D: usize>;

        // Implement the backward trait for the given backend B, the node gradient
being of rank D
        // with three other gradients to calculate (lhs, rhs, and bias).
        impl<B: Backend, const D: usize> Backward<B, D, 3> for
FusedMatmulAddReluBackward<D> {
            // The state that must be built 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 = (
                FloatTensor<B, D>,
                FloatTensor<B, D>,
                FloatTensor<B, D>,
                Shape<D>,
            );

            fn backward(self, ops: Ops<Self::State, 3>, grads: &mut Gradients) {
                // 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, D>(&ops.node);

                // Set the state.
                let (lhs, rhs, output, shape_bias) = ops.state;

                // Fetch shapes of the tensors to support broadcasting.
                let shape_lhs = B::shape(&lhs);
                let shape_rhs = B::shape(&rhs);

                // 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, D>(
                    B::matmul(grad_output.clone(), B::transpose(rhs)),
                    &shape_lhs,
                );
                // Compute the rhs gradient, which is the derivative of matmul
with support for
                // broadcasting.
                let grad_rhs = broadcast_shape::<B, D>(
                    B::matmul(B::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, D>(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, D>(node, grad_bias);
                }
                if let Some(node) = node_lhs {
                    grads.register::<B, D>(node, grad_lhs);
                }
                if let Some(node) = node_rhs {
                    grads.register::<B, D>(node, 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(
                [lhs.node, rhs.node, bias.node],
                [lhs.graph, rhs.graph, bias.graph],
            )
            .stateful()
        {
            OpsKind::Tracked(prep) => {
                // When at least one node is tracked, we should register our
backward step.
                // We compute the output and the state before finishing the
preparation.
```

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

                    let state = (lhs.primitive, rhs.primitive, 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<Wgpu<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.

It is worth noting that while the manual fusion of operations can be valuable, our future plans include the development of a backend extension that will automate this process. 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.