We've been busy at work unifying core abstractions between what we've historically called PyTorch and Caffe2. This work allows to have zero-copy tensor sharing between PyTorch and Caffe2 codebases for POD (e.g., float, int, etc.) contiguous tensors. This functionality makes it easier to put together PyTorch and Caffe2 code!

During this process, we attempted to unify functionality from both PyTorch and Caffe2 code, so that existing code on both sides keeps working. However, in some cases, PyTorch and Caffe2 differed, and we had to come down on one side or another. This note describes some of the differences (primarily affecting the Caffe2 APIs).

Most of these changes are already part of PyTorch 1.0 (the rest are in master); the purpose of this note is to help you understand how to write your code differently to take advantage of these new changes.

High level

- **There are two "tensor wrappers", at::Tensor and caffe2::Tensor which expose different APIs around the same implementation class and thus support zero-copy conversion.** The wrappers present the traditional APIs historically used by PyTorch and Caffe2 respectively.
    - **Primary motivation:** Caffe2 supports non-POD-valued tensors (for example, a tensor whose individual elements are std::string, or perhaps even more unusually, individual elements are themselves tensors) and this is the functionality we don't want to preserve and address the use cases differently. PyTorch tensors support strided views on underlying storage, while existing Caffe2 code doesn't expect non-contiguous tensors. Two separate wrappers allows us to introduce explicit API boundary with clear conversion point that enforces the contract. For regular POD-valued contiguous tensors, this conversion effectively consists of an optional refcount bump and thus is free or very cheap (Tensor is a pointer underneath: more on this below). Either wrapper can be used to modify the underlying Tensor, however to avoid weird behaviors it's best not to modify tensor metadata of a tensor in-place actively if both wrappers are held.
    - **Secondary motivation:** We considered immediately merging the two original APIs, but PyTorch Tensor and Caffe2 Tensor have fairly different API philosophies (for example, historically Caffe2 tensors are initialized using constructors, whereas PyTorch tensors are initialized using factory functions), and we don't want deprecated usage patterns (e.g., `Tensor*` and partially initialized tensors--discussed in more detail below) to leak into code using the modern patterns. So, having two interfaces lets us maintain BC, while we fix old use-sites. However, in places where it makes sense, we have converged the APIs of the two classes, and intend to maintain this convergence for as long as there are two wrappers.
    - **C++ example:** Converting from at::Tensor to caffe2::Tensor: `caffe2::Tensor c2_tensor(at_tensor)`;
    - **C++ example:** Converting from caffe2::Tensor to at::Tensor: `at::Tensor at_tensor(c2_tensor)`.
    - **Python examples:** (works on both CPU and GPU; this is NOT in 1.0, you must be on master to get this functionality)
        - Retrieve a PyTorch tensor from a Caffe2 workspace: `workspace.FetchTorch('x')`
        - Turn a blob into a PyTorch tensor (in this example, the blob comes from a workspace): `workspace.Workspace.current_blobs['x'].to_torch()`
        - Feed a PyTorch tensor into a Caffe2 workspace: `workspace.FeedBlob('x', torch.randn(5))`

- **Tensor is now a pointer type.** Think of it as being equivalent to `shared_ptr<TensorImpl>` (where TensorImpl what historically you thought of as caffe2::Tensor; it's the struct that contains the metadata for the tensor). If you are a native C++ programmer, this behavior is unusual; fortunately, we expect this to be the only visible C++ class in PyTorch 1.0 which has this behavior.

    - **Motivation:** This change is the product of a lot of discussion. Writing Python in C++ gives some of the original motivation, but there are also some common failure modes in the pointer-based world. File an issue if you want us to say more about this topic.
    - **Corollary:** Given that Tensor is a pointer already, this means Tensor* should be avoided. There are a number of legacy APIs which are spelled this way (for example, `Operator::Output`), but new code should avoid taking a pointer to Tensor. `const Tensor&`, on the other hand, is still idiomatic, because it avoids reference count bumps and at use-sites is interchangeable with Tensor.
    - **Corollary:** `const Tensor&` doesn't mean “immutable tensor”, you can still mutate a tensor with this type; const `Tensor&` means you have a const reference to a mutable tensor. (A lot of people find this surprising, but it’s an unavoidable consequence of making Tensor a pointer type.)
    - **Corollary:** The method `data()` no longer returns a const pointer when Tensor is const; it now always returns a non-const pointer.
    - **Corollary:** It is unsafe to pass a Tensor to another thread, if either thread intends to subsequently mutate the tensor. A tensor that was an input to Predictor counts as such a tensor, as Predictor in-place resizes tensors when it processes the next input. (More subtly: you can now create two Tensors which share underlying storage; this is also unsafe if the Tensors are used in different threads.)
    - **Example:** `Tensor x = y` doesn't create a copy of the tensor metadata or structure; x and y refer to exactly the same tensor.
        - To make a deep copy, `Tensor x = y.clone()` (for `at::Tensor`) or `Tensor x = y.Clone()` (for `caffe2::Tensor`).
        - The copy constructor behavior of `caffe2::Tensor` is a new development - historically copy constructor was not supported. To avoid potential misuse in new call-sites, we temporarily disabled copy constructor on `caffe2::Tensor` wrapper, use `Tensor x = y.UnsafeSharedInstance()` instead. The copy constructor is enabled on `at::Tensor` wrapper.

- **Idiomatic construction of tensors uses factory functions rather than constructors.**
    - **Motivation:** There are two reasons for this. First, this matches Python APIs like Numpy and PyTorch, which don't have a notion of C++ style constructor. Second, this API eliminate “partially initialized states”, e.g., a Tensor which knows its size but not its dtype.
    - **Example:** `Tensor x = empty({2, 2});` instead of `Tensor x(CPU); ...`
    - **Example:** When writing operator code: `auto x = XOutput(0, {2, 2}, dtype<int64_t>())`; instead of `auto* x_ptr = Output(0); ...`. Previously, idiomatic use was to call `Output()` to get a pointer to an uninitialized Tensor which the operator would subsequently initialized; the new method `XOutput()` automatically handles initialization for you. `XOutput` returns a Tensor while Output returns a `Tensor*`; there is also a new Output overload which returns an old-style `Tensor*` rather than a `Tensor`. (The intent is eventually to eliminate `Output` entirely and rename `XOutput` to `Output`; unfortunately, there are still places where it is more convenient to return `Output*`)
- **PyTorch code leverages views with shared storage.** While `caffe2::Tensor` supported sharing the underlying data buffer (via `ShareData` and `ShareExternalPointer`) this functionality wasn't used much and many call sites implicitly assumed exclusive ownership. PyTorch's tensor semantics rely heavily on numpy-style ability to create multiple aliasing views on a single tensor. It provides powerful user API; however, it also means that generic systems code may need to keep potential shared storage in mind.
    - For an introduction to views, check out the PyTorch documentation on view. https://pytorch.org/docs/stable/tensors.html#torch.Tensor.view

- **We’ve introduced a new reference type for sizes() and strides(): ArrayRef (with a well known synonym, IntArrayRef = ArrayRef).** `ArrayRef<T>` is a non-owning, read-only reference to a contiguous array of `T`. You can think of it as a raw pointer plus a size.
    - **Motivation:** `ArrayRef`, as a value type, replaces conventional use of `const std::vector<T>&` to refer to an array by reference. The primary benefit of using `ArrayRef` is that it is container agnostic: an `ArrayRef` can validly refer to the data of a vector, an initializer list, or any other array-like type which stores its elements contiguously in memory (e.g., `c10::util::SmallVector`). An interface which returns a reference to a vector commits to internally representing the array as a vector; an interface which returns `ArrayRef` can use whatever internal representation makes sense for itself.
    - **Example:** Previously, `caffe2::Tensor::dims()` returned a `const std::vector<int64_t>&`; it now returns an `ArrayRef<int64_t>`. Most common operations (e.g., `size()`, `operator[]`) still work. To make a copy of the ArrayRef as a vector, call the `vec()` method on it (previously, this would have happened implicitly).
    - **Warning:** As ArrayRef is non-owning, it generally should not be stored in a struct. Also, take care when returning an ArrayRef from a function: make sure users of your API understand that it is a reference type, and thus they cannot hold onto it beyond the lifetime of the originating vector.

Examples: Before and after

There are a number of common idioms from Caffe2, which now have new spellings. This section covers some of the most important changes.

Constructing tensors

Before:

```
caffe2::Tensor x(CPU);
x.Resize(size);
float* x_data = x->mutable_data<float>();
```

After:

```
caffe2::Tensor x = caffe2::empty(size, dtype<float>().device(CPU));
float* x_data = x->data<float>();
```

Retrieving output in operator

Before:

```
caffe2::Tensor* output = Output(0);
output->Resize(size);
float* output_data = output->mutable_data<float>();
```

After:

```
// deprecated version (but requires less code changes)
caffe2::Tensor* output = Output(0, size, dtype<float>());
float* output_data = output->data<float>();

// new version
```

```
caffe2::Tensor output = XOutput(0, size, dtype<float>());
float* output_data = output.data<float>();
```

NB: Eventually, we intend for Output to return a `Tensor`, rather than a `Tensor*`.

Variation: If you said `output->ResizeLike(input);` say `XOutput(..., input.sizes(), ...);` instead

Reading sizes into a vector

Before:

```
std::vector<int64_t> sizes = tensor.dims();
sizes[0] = 1;
```

After:

```
std::vector<int64_t> sizes = tensor.dims().vec(); // auto is also ok
sizes[0] = 1;
```

Pointer to a tensor

Before:

```
void do_some_stuff(caffe2::Tensor* tensor_ptr) {
  auto* p = tensor_ptr->mutable_data<T>();
  ...
}
```

After:

```
void do_some_stuff(const caffe2::Tensor& tensor) {
  auto* p = tensor_ptr.data<T>();
  ...
}
```

Caching tensors

A common idiom in Caffe2, owing to the fact that operators are stateful, is caching a tensor as a member in a class, constructing it if it is not already initialized, and otherwise reusing the previous buffer. We introduced a new method on Operator, ReinitializeTensor, precisely for this case.

Before:

```
template <typename Context>
class CosineSimilarityOp : public Operator<Context> {
  bool RunOnDevice() {
    // ...
    aux_.Resize(N, M);
    float* aux_data = aux_.mutable_data<float>();
    // ...
  }
private:
  Tensor aux_{Context::GetDeviceType()};
}
```

After:

```
template <typename Context>
class CosineSimilarityOp : public Operator<Context> {
  bool RunOnDevice() {
    // ...
    ReinitializeTensor(&aux_, {N, M}, at::dtype<float>
().device(Context::GetDeviceType()));
    float* aux_data = aux_.data<float>();
    // ...
  }
private:
  Tensor aux_;
}
```

Abridged API reference

Below, we give abridged API reference for caffe2::Tensor, Operator and ArrayRef. These APIs are not complete, but contain the most commonly used operations / notably changed methods. (at::Tensor is not included, because it was left substantially unchanged during unification. You can read its full API here.)

Caffe2 Tensor: (full API)

```
namespace caffe2 {

class Tensor {
public:
  // Converting methods

  explicit Tensor(at::Tensor);    // conversion from at::Tensor
  explicit operator at::Tensor(); // conversion to at::Tensor

  // "Copy" methods

  Tensor(const Tensor&) = delete; // disabled to avoid confusion; see below
  Tensor UnsafeSharedInstance(); // pointer copy (shares metadata and data)

  Tensor Alias() const; // shallow copy; copies metadata but not data
                        // NB: you probably DON'T want this method
                        // y = x.Alias() is equivalent to what
                        // y.ShareData(x) used to do
  Tensor Clone() const; // deep copy; copies metadata and data
                        // y = x.Clone() is equivalent to what
                        // y.CopyFrom(x) used to do

  // Renamed methods:

  int64_t dim();              // previously called ndim()
  int64_t numel();            // previously called size()
  IntArrayRef sizes();       // previously called dims()
  TypeMeta dtype();          // previously called meta()
  int64_t size(int64_t dim); // previously called dim()

  template <typename T> T* data(); // returns mutable pointer now
}

// Create a tensor with uninitialized data (see below about TensorOptions)
Tensor empty(IntArrayRef dims, TensorOptions options);

}
```

TensorOptions is a class with a number of helper functions for its construction. It has a number of implicit constructors to make working with it more convenient; you can see a number of examples and how to use it in the PyTorch C++ documentation. Operator: (full API; this class was adjusted to add some new methods that allow you to work with tensors without materializing uninitialized Tensors)

```
namespace caffe2 {

template <typename Context>
class Operator : OperatorBase {
  // Allocate (or retrieve) the tensor corresponding to the idx
  // output of an operator, with sizes and dtype as specified
  // by sizes and options.  The device of the tensor is inferred
  // from the Context parameter of Operator.
  Tensor XOutput(int idx, IntArrayRef sizes, TensorOptions options);

  // Similar to XOutput, but it returns a more backwards compatible
  // Tensor*.
  Tensor* Output(int idx, IntArrayRef sizes, TensorOptions options);
}

}
```

ArrayRef: (full API)

```
namespace c10 {

// Non-owning, read-only reference to an array, like folly::Range
template <typename T>
class ArrayRef {
public:
  ArrayRef();                        // empty list
  ArrayRef(const std::vector<T>&); // implicit conversion from vector
  ArrayRef(const std::initializer_list<T>&); // implicit conversion from {1,2,3}

  bool empty() const;                 // is list zero-length?
  size_t size() const;                // length of list
  bool equals(ArrayRef) const;        // list equality
  const T& operator[](size_t) const; // indexing (bounds checked)
  std::vector<T> vec() const;         // copy into vector
}
}
```