Inference Mode

`c10::InferenceMode` is a new RAII guard analogous to `NoGradMode` to be used when you are certain your operations will have no interactions with autograd (e.g. model training). Compared to `NoGradMode`, code run under this mode gets better performance by disabling autograd related work like view tracking and version counter bumps. However, tensors created inside `c10::InferenceMode` has more limitation when interacting with autograd system as well.

`InferenceMode` can be enabled for a given block of code. Inside `InferenceMode` all newly allocated (non-view) tensors are marked as inference tensors. Inference tensors:

- do not have a version counter so an error will be raised if you try to read their version (e.g., because you saved this tensor for backward).
- are immutable outside `InferenceMode`. So an error will be raised if you try to: - mutate their data outside InferenceMode. - mutate them into `requires_grad=True` outside InferenceMode. To work around you can make a clone outside `InferenceMode` to get a normal tensor before mutating.

A non-view tensor is an inference tensor if and only if it was allocated inside `InferenceMode`. A view tensor is an inference tensor if and only if the tensor it is a view of is an inference tensor.

Inside an `InferenceMode` block, we make the following performance guarantees:

- Like `NoGradMode`, all operations do not record `grad_fn` even if their inputs have `requires_grad=True`. This applies to both inference tensors and normal tensors.
- View operations on inference tensors do not do view tracking. View and non-view inference tensors are indistinguishable.
- Inplace operations on inference tensors are guaranteed not to do a version bump.

For more implementation details of `InferenceMode` please see the RFC-0011-InferenceMode.

Migration guide from `AutoNonVariableTypeMode`

In production use of PyTorch for inference workload, we have seen a proliferation of uses of the C++ guard `AutoNonVariableTypeMode` (now `AutoDispatchBelowADInplaceOrView`), which disables autograd, view tracking and version counter bumps. Unfortunately, current colloquial of this guard for inference workload is unsafe: it's possible to use `AutoNonVariableTypeMode` to bypass PyTorch's safety checks and result in silently wrong results, e.g. PyTorch throws an error when tensors saved for backwards are subsequently mutated, but mutation happens inside `AutoNonVariableTypeMode` will silently bypass the check and returns wrong gradient to users.

When current users of `AutoNonVariableTypeMode` think about migrating, the following steps might help you decide the best alternatives:

1. Users trying to run workload in inference only mode (like loading a pretrained JIT model and run inference in C++ runtime) should add `c10::InferenceMode guard` to guard all operations on tensors (including model loading). See an inference workload example below:

```
c10::InferenceMode guard;
model.load_jit(saved_model);
auto inputs = preprocess_tensors(data);
auto out = model.forward(inputs);
auto outputs = postprocess_tensors(out);
```

Note `c10::InferenceMode` offers a drop in replacement for `AutoNonVariableTypeMode` which preserves the performance characteristics of `AutoNonVariableTypeMode`. But they also have some differences that users should pay additional attention to:

   - Both guards affects tensor execution process to skip work not related to inference, but `InferenceMode` also affects tensor creation while `AutoNonVariableTypeMode` doesn't. In other words, tensors created inside `InferenceMode` are marked as inference tensors so that certain limitation can be applied after exiting `InferenceMode`.
   - Enabled/disabled `InferenceMode` states can be nested while `AutoNonVariableTypeMode` only allows enabled state..

```
{
  InferenceMode guard(true);
  // InferenceMode is on
  {
    InferenceMode guard(false);
    // InferenceMode is off
  }
  // InferenceMode is on
}
// InferenceMode is off
```

2. Users trying to implement a customized kernel who wants to redispatch under `Autograd` dispatch keys should use

`AutoDispatchBelowADInplaceOrView` instead. Note `AutoDispatchBelowADInplaceOrView` is just a new name of `AutoNonVariableTypeMode` since it explains the guard's functionality better. We're deprecating `AutoNonVariableTypeMode` and it'll be removed in 1.10 release. See customized kernel `ROIAlignFunction` in `pytorch/vision` for an example:

```
class ROIAlignFunction : public torch::autograd::Function<ROIAlignFunction> {
 public:
  static torch::autograd::variable_list forward(
      torch::autograd::AutogradContext* ctx,
      const torch::autograd::Variable& input,
      const torch::autograd::Variable& rois,
      double spatial_scale,
      int64_t pooled_height,
      int64_t pooled_width,
      int64_t sampling_ratio,
      bool aligned) {
    ctx->saved_data["spatial_scale"] = spatial_scale;
    ctx->saved_data["pooled_height"] = pooled_height;
    ctx->saved_data["pooled_width"] = pooled_width;
    ctx->saved_data["sampling_ratio"] = sampling_ratio;
    ctx->saved_data["aligned"] = aligned;
    ctx->saved_data["input_shape"] = input.sizes();
    ctx->save_for_backward({rois});
    // Used to be at::AutoNonVariableTypeMode g;
    at::AutoDispatchBelowADInplaceOrView guard;
    auto result = roi_align(
        input, rois, spatial_scale, pooled_height,
        pooled_width, sampling_ratio, aligned);
    return {result};
  }
```

Customized inplace & view kernels need some special handling in addition to the guard above, see custom kernel tutorial for more details.