Will Feng C++ autograd example (#745) ad775ac · 4 years ago History

Code Blame 191 lines (147 loc) · 5.86 KB Raw

```cpp
#include <torch/torch.h>
#include <iostream>

using namespace torch::autograd;

void basic_autograd_operations_example() {
  std::cout << "====== Running: \"Basic autograd operations\" ======" << std::endl;

  // Create a tensor and set ``torch::requires_grad()`` to track computation with it
  auto x = torch::ones({2, 2}, torch::requires_grad());
  std::cout << x << std::endl;

  // Do a tensor operation:
  auto y = x + 2;
  std::cout << y << std::endl;

  // ``y`` was created as a result of an operation, so it has a ``grad_fn``.
  std::cout << y.grad_fn()->name() << std::endl;

  // Do more operations on ``y``
  auto z = y * y * 3;
```

```
22      auto out = z.mean();
23
24      std::cout << z << std::endl;
25      std::cout << z.grad_fn()->name() << std::endl;
26      std::cout << out << std::endl;
27      std::cout << out.grad_fn()->name() << std::endl;
28
29      // ``.requires_grad_( ... )`` changes an existing tensor's ``requires_grad`` flag in-place.
30      auto a = torch::randn({2, 2});
31      a = ((a * 3) / (a - 1));
32      std::cout << a.requires_grad() << std::endl;
33
34      a.requires_grad_(true);
35      std::cout << a.requires_grad() << std::endl;
36
37      auto b = (a * a).sum();
38      std::cout << b.grad_fn()->name() << std::endl;
39
40      // Let's backprop now. Because ``out`` contains a single scalar, ``out.backward()``
41      // is equivalent to ``out.backward(torch::tensor(1.))``.
42      out.backward();
43
44      // Print gradients d(out)/dx
45      std::cout << x.grad() << std::endl;
46
47      // Now let's take a look at an example of vector-Jacobian product:
48      x = torch::randn(3, torch::requires_grad());
49
50      y = x * 2;
51      while (y.norm().item<double>() < 1000) {
52        y = y * 2;
53      }
54
55      std::cout << y << std::endl;
56      std::cout << y.grad_fn()->name() << std::endl;
57
```

```
    // If we want the vector-Jacobian product, pass the vector to ``backward`` as argument:
    auto v = torch::tensor({0.1, 1.0, 0.0001}, torch::kFloat);
    y.backward(v);

    std::cout << x.grad() << std::endl;

    // You can also stop autograd from tracking history on tensors that require gradients
    // either by putting ``torch::NoGradGuard`` in a code block
    std::cout << x.requires_grad() << std::endl;
    std::cout << x.pow(2).requires_grad() << std::endl;

    {
      torch::NoGradGuard no_grad;
      std::cout << x.pow(2).requires_grad() << std::endl;
    }

    // Or by using ``.detach()`` to get a new tensor with the same content but that does
    // not require gradients:
    std::cout << x.requires_grad() << std::endl;
    y = x.detach();
    std::cout << y.requires_grad() << std::endl;
    std::cout << x.eq(y).all().item<bool>() << std::endl;
  }

  void compute_higher_order_gradients_example() {
    std::cout << "====== Running \"Computing higher-order gradients in C++\" ======" << std::endl;

    // One of the applications of higher-order gradients is calculating gradient penalty.
    // Let's see an example of it using ``torch::autograd::grad``:

    auto model = torch::nn::Linear(4, 3);

    auto input = torch::randn({3, 4}).requires_grad_(true);
    auto output = model(input);

    // Calculate loss
```

```
    auto target = torch::randn({3, 3});
    auto loss = torch::nn::MSELoss()(output, target);

    // Use norm of gradients as penalty
    auto grad_output = torch::ones_like(output);
    auto gradient = torch::autograd::grad({output}, {input}, /*grad_outputs=*/{grad_output}, /*create_graph=*/true)[0];
    auto gradient_penalty = torch::pow((gradient.norm(2, /*dim=*/1) - 1), 2).mean();

    // Add gradient penalty to loss
    auto combined_loss = loss + gradient_penalty;
    combined_loss.backward();

    std::cout << input.grad() << std::endl;
  }

  // Inherit from Function
  class LinearFunction : public Function<LinearFunction> {
   public:
    // Note that both forward and backward are static functions

    // bias is an optional argument
    static torch::Tensor forward(
        AutogradContext *ctx, torch::Tensor input, torch::Tensor weight, torch::Tensor bias = torch::Tensor()) {
      ctx->save_for_backward({input, weight, bias});
      auto output = input.mm(weight.t());
      if (bias.defined()) {
        output += bias.unsqueeze(0).expand_as(output);
      }
      return output;
    }

    static tensor_list backward(AutogradContext *ctx, tensor_list grad_outputs) {
      auto saved = ctx->get_saved_variables();
      auto input = saved[0];
      auto weight = saved[1];
      auto bias = saved[2];
```

```
130
131         auto grad_output = grad_outputs[0];
132         auto grad_input = grad_output.mm(weight);
133         auto grad_weight = grad_output.t().mm(input);
134         auto grad_bias = torch::Tensor();
135         if (bias.defined()) {
136           grad_bias = grad_output.sum(0);
137         }
138
139         return {grad_input, grad_weight, grad_bias};
140       }
141     };
142
143   class MulConstant : public Function<MulConstant> {
144    public:
145     static torch::Tensor forward(AutogradContext *ctx, torch::Tensor tensor, double constant) {
146       // ctx is a context object that can be used to stash information
147       // for backward computation
148       ctx->saved_data["constant"] = constant;
149       return tensor * constant;
150     }
151
152     static tensor_list backward(AutogradContext *ctx, tensor_list grad_outputs) {
153       // We return as many input gradients as there were arguments.
154       // Gradients of non-tensor arguments to forward must be `torch::Tensor()`.
155       return {grad_outputs[0] * ctx->saved_data["constant"].toDouble(), torch::Tensor()};
156     }
157   };
158
159   void custom_autograd_function_example() {
160     std::cout << "====== Running \"Using custom autograd function in C++\" ======" << std::endl;
161     {
162       auto x = torch::randn({2, 3}).requires_grad_();
163       auto weight = torch::randn({4, 3}).requires_grad_();
164       auto y = LinearFunction::apply(x, weight);
165       y.sum().backward();
```

```
166
167         std::cout << x.grad() << std::endl;
168         std::cout << weight.grad() << std::endl;
169       }
170       {
171         auto x = torch::randn({2}).requires_grad_();
172         auto y = MulConstant::apply(x, 5.5);
173         y.sum().backward();
174
175         std::cout << x.grad() << std::endl;
176       }
177     }
178
179     int main() {
180       std::cout << std::boolalpha;
181
182       basic_autograd_operations_example();
183
184       std::cout << "\n";
185
186       compute_higher_order_gradients_example();
187
188       std::cout << "\n";
189
190       custom_autograd_function_example();
191     }
```