Deep Learning Framework

Weijie Zhao

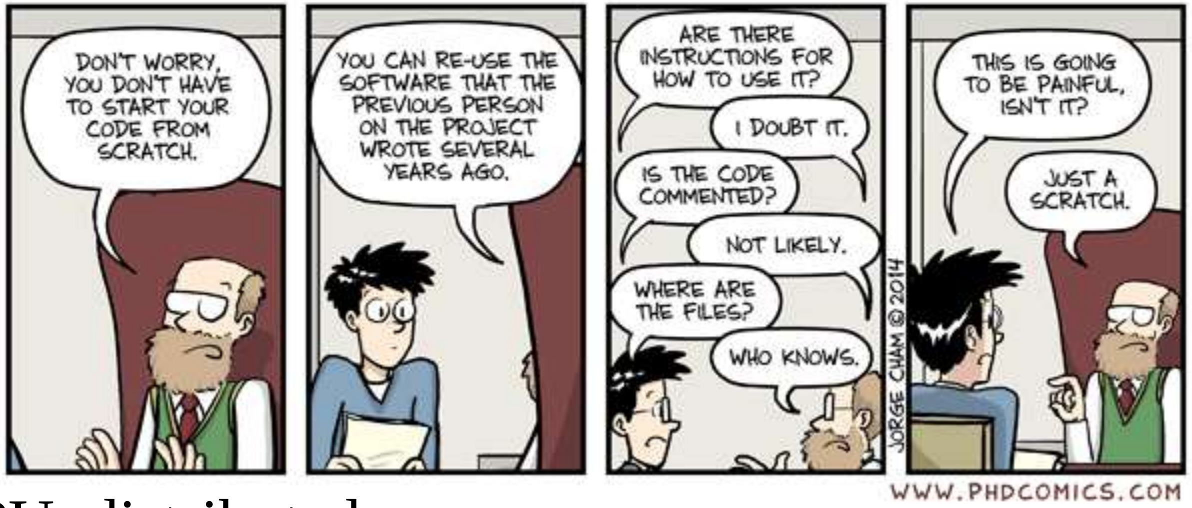
03/18/2024

Why Deep Learning Framework?

Deep Learning ~~Framework~~

- Data pre-processing
- Training
  - Tensor operations
  - Gradient computations
  - Parallelism: multi-threading, GPU, distributed
- Deployment
  - Latency
  - Model compression
  - Heterogeneous devices: server, laptop, mobile device, etc.

Deep Learning ~~Framework~~



- Data pre-processing
- Training
  - Tensor operations
  - Gradient computations
  - Parallelism: multi-threading, GPU, distributed
- Deployment
  - Latency
  - Model compression
  - Heterogeneous devices: server, laptop, mobile device, etc.

Without a framework, you have to do all these mess for each model!

Data Loading

```
import torch
from torch.utils.data import Dataset
from torchvision import datasets
from torchvision.transforms import ToTensor

training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)
```

Model

```
# Define model
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10)
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits

model = NeuralNetwork()
```

Train and Test

```
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), (batch + 1) * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

```
def test(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.eval()
    test_loss, correct = 0, 0
    with torch.no_grad():
        for X, y in dataloader:
            X, y = X.to(device), y.to(device)
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) ==
y).type(torch.float).sum().item()
    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%,
     Avg loss: {test_loss:>8f} \n")
```

Train and Test

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)

epochs = 5
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train(train_dataloader, model, loss_fn, optimizer)
    test(test_dataloader, model, loss_fn)
```

```
Epoch 1 -------------------------------
loss: 2.313734 [ 64/60000]
loss: 2.293157 [ 6464/60000]
loss: 2.279998 [12864/60000]
loss: 2.268756 [19264/60000]
loss: 2.255548 [25664/60000]
loss: 2.232167 [32064/60000]
loss: 2.245508 [38464/60000]
loss: 2.221311 [44864/60000]
loss: 2.217233 [51264/60000]
loss: 2.175452 [57664/60000]
Test Error: Accuracy: 27.8%, Avg loss: 2.171504
```

Customized Layer

- What if we want to propose a new layer?

```
class Square(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x):
        # Because we are saving one of the inputs use `save_for_backward`
        # Save non-tensors and non-inputs/non-outputs directly on ctx
        ctx.save_for_backward(x)
        return x**2

    @staticmethod
    def backward(ctx, grad_out):
        # A function support double backward automatically if autograd
        # is able to record the computations performed in backward
        x, = ctx.saved_tensors
        return grad_out * 2 * x
```

Long Long Term Memory

```
class LLTM(torch.nn.Module):
    def __init__(self, input_features, state_size):
        super(LLTM, self).__init__()
        self.input_features = input_features
        self.state_size = state_size

        self.weights = torch.nn.Parameter(
            torch.empty(3 * state_size, input_features + state_size))
        self.bias = torch.nn.Parameter(torch.empty(3 * state_size))
        self.reset_parameters()

    def reset_parameters(self):
        stdv = 1.0 / math.sqrt(self.state_size)
        for weight in self.parameters():
            weight.data.uniform_(-stdv, +stdv)
```

```
def forward(self, input, state):
        old_h, old_cell = state
        X = torch.cat([old_h, input], dim=1)

        gate_weights = F.linear(X, self.weights, self.bias)
        gates = gate_weights.chunk(3, dim=1)

        input_gate = torch.sigmoid(gates[0])
        output_gate = torch.sigmoid(gates[1])
        # Here we use an ELU instead of the usual tanh.
        candidate_cell = F.elu(gates[2])

        # Compute the new cell state.
        new_cell = old_cell + candidate_cell * input_gate
        # Compute the new hidden state and output.
        new_h = torch.tanh(new_cell) * output_gate

        return new_h, new_cell
```

Forward

```
std::vector<at::Tensor> lltm_forward(
    torch::Tensor input,
    torch::Tensor weights,
    torch::Tensor bias,
    torch::Tensor old_h,
    torch::Tensor old_cell) {
  auto X = torch::cat({old_h, input}, /*dim=*/1);

  auto gate_weights = torch::addmm(bias, X, weights.transpose(0, 1));
  auto gates = gate_weights.chunk(3, /*dim=*/1);

  auto input_gate = torch::sigmoid(gates[0]);
  auto output_gate = torch::sigmoid(gates[1]);
  auto candidate_cell = torch::elu(gates[2], /*alpha=*/1.0);

  auto new_cell = old_cell + candidate_cell * input_gate;
  auto new_h = torch::tanh(new_cell) * output_gate;

  return {new_h,
          new_cell,
          input_gate,
          output_gate,
          candidate_cell,
          X,
          gate_weights};
}
```

Backward

```
std::vector<torch::Tensor> lltm_backward(
    torch::Tensor grad_h,
    torch::Tensor grad_cell,
    torch::Tensor new_cell,
    torch::Tensor input_gate,
    torch::Tensor output_gate,
    torch::Tensor candidate_cell,
    torch::Tensor X,
    torch::Tensor gate_weights,
    torch::Tensor weights) {
  .....

  return {d_old_h, d_input, d_weights, d_bias, d_old_cell};
}
```

Python Binding

```
PYBIND11_MODULE(TORCH_EXTENSION_NAME, m) {
  m.def("forward", &lltm_forward, "LLTM forward");
  m.def("backward", &lltm_backward, "LLTM backward");
}
```

Just-In-Time Compile

```
from torch.utils.cpp_extension import load

lltm_cpp = load(name="lltm_cpp", sources=["lltm.cpp"])

class LLTMFunction(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input, weights, bias, old_h, old_cell):
        outputs = lltm_cpp.forward(input, weights, bias, old_h, old_cell)
        new_h, new_cell = outputs[:2]
        variables = outputs[1:] + [weights]
        ctx.save_for_backward(*variables)

        return new_h, new_cell

    @staticmethod
    def backward(ctx, grad_h, grad_cell):
        outputs = lltm_cpp.backward(
            grad_h.contiguous(), grad_cell.contiguous(), *ctx.saved_tensors)
        d_old_h, d_input, d_weights, d_bias, d_old_cell = outputs
        return d_input, d_weights, d_bias, d_old_h, d_old_cell
```

Compilation

```
from setuptools import setup, Extension
from torch.utils import cpp_extension

setup(name='lltm_cpp',
      ext_modules=[cpp_extension.CppExtension('lltm_cpp',
['lltm.cpp'])],
      cmdclass={'build_ext': cpp_extension.BuildExtension})
```

Customized Hashing Layer

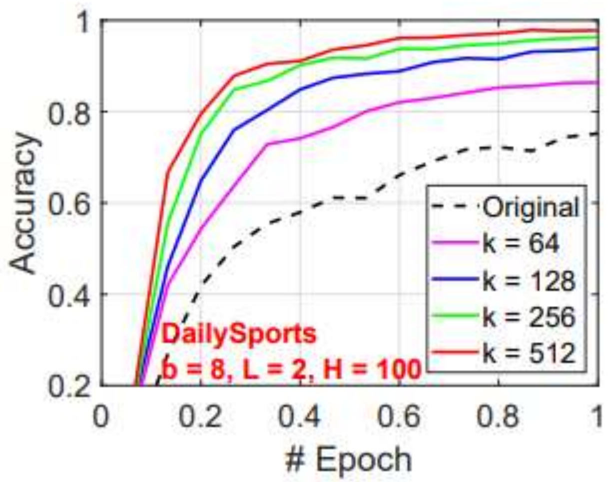
- Consistent Weighted Sampling

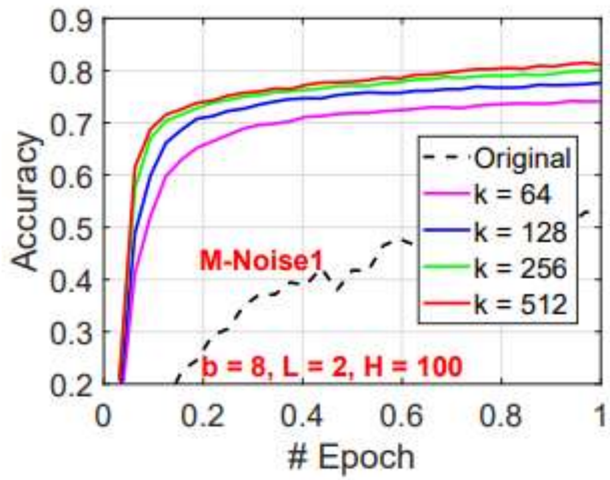
$$\textbf{min-max}: \quad K_{MM}(u,v) = \frac{\sum_{i=1}^{D} \min\{u_i,\ v_i\}}{\sum_{i=1}^{D} \max\{u_i,\ v_i\}}$$

Algorithm 1 Consistent Weighted Sampling (CWS)

Input: Data vector $u = (u_i \geq 0, i = 1 \text{ to } D)$
Output: Consistent uniform sample (i^*, t^*)

For i from 1 to D
$\quad r_i \sim Gamma(2,1), \ \ c_i \sim Gamma(2,1), \beta_i \sim Uniform(0,1)$
$\quad t_i \leftarrow \lfloor \frac{\log u_i}{r_i} + \beta_i \rfloor, \ y_i \leftarrow \exp(r_i(t_i - \beta_i)), a_i \leftarrow c_i/(y_i \exp(r_i))$
End For
$i^* \leftarrow arg\min_i \ a_i, \qquad t^* \leftarrow t_{i^*}$





Ping Li and Weijie Zhao. "GCWSNet: Generalized Consistent Weighted Sampling for Scalable and Accurate Training of Neural Networks." arXiv preprint arXiv:2201.02283 (2022).