Edugrad

Ling 575j: Deep Learning for NLP
C.M. Downey

Edugrad, intro

- https://github.com/shanest/edugrad
- Minimal re-implementation of PyTorch API, for educational purposes
  - Forward/backward API for operations
    - Automatic differentiation via backprop
  - Dynamic computation graph
- Why? Modern DL libraries have so much additional cruft that you cannot chase back lots of method calls to their implementations.
  - E.g. what *really* happens when you call `loss.backward()`?
- NB: no performance optimizations, no GPU usage, etc. in edugrad

Edugrad: Tensor

- Tensor: wrapper around a numpy array (stored in .value attribute)
  - value: np array
  - grad: current gradient! (Set to 0 initially, populated during back propagation)
- Primary operators overloaded: +, -, ** (raise to a power)
  - More on implementation of those in a second

```
>>> import numpy as np
>>> from edugrad.tensor import Tensor
>>> t1 = Tensor(np.array([[1, 2], [3, 4]]))
>>> t2 = Tensor(np.array([[1, 2], [3, 4]]))
>>> t1 + t2
<edugrad.tensor.Tensor object at 0x7f97a81d5940>
>>> (t1 + t2).value
array([[2, 4],
       [6, 8]])
```

Edugrad: Operation

- Operation: defines forward/backward
  - **In forward/backward: np arrays, *not* Tensors**
- @tensor_op:
  - Takes an Operation, turns it into a method that takes Tensor arguments and returns Tensor outputs
    - And which *builds the computation graph dynamically*
  - @: decorator; equivalent to: add = tensor_op(add)
- Basic ops provided:
  - https://github.com/shanest/edugrad/blob/master/edugrad/ops.py

```python
@tensor_op
class add(Operation):
    @staticmethod
    def forward(ctx, a, b):
        return a + b

    @staticmethod
    def backward(ctx, grad_output):
        return grad_output, grad_output
```

```
>>> from edugrad.ops import add
>>> add(t1, t2).value
array([[2, 4],
       [6, 8]])
```

Edugrad: nn.Module

- edugrad.nn.Module:
  - As in PyTorch, basic model class
  - Stores parameters [accessed via .parameters()]
  - Can be nested (modules within modules)
  - Implements `forward`
- Defining a custom module:
  - Sub-class nn.Module
  - Initialize params in __init__
  - Implement custom forward method

Edugrad: Linear Module example

```python
class Linear(Module):
    def __init__(
        self,
        input_size: int,
        output_size: int,
        bias: bool = True,
    ):
        """A Linear module computes defines weights W, optionally biases b, and
        computers wX + b.

        Weight vector will have shape (input size, output size)

        Args:
            input_size: dimension of input vectors
            output_size: dimension of output vectors
            initializer: how to initialize weights and biases
            bias: whether or not to include the bias term; not needed for, e.g. embeddings
        """
        super(Linear, self).__init__()
        scale = 1 / np.sqrt(input_size)
        self.weight = Tensor(uniform_initializer((input_size, output_size), scale=scale), name="W")
        self.has_bias = bias
        if self.has_bias:
            # biases initialize to 0
            self.bias = Tensor(uniform_initializer((output_size,), scale=scale), name="b")
```

Always do this first!!

Define parameters

Edugrad: Linear Module

```python
def forward(self, inputs: Tensor):
    mul_node = ops.matmul(inputs, self.weight)
    if self.has_bias:
        # NOTE: this is a hack-ish way of handling shape issues with biases
        expanded_biases = ops.copy_rows(self.bias, num=inputs.value.shape[0])
        return ops.add(mul_node, expanded_biases)
    return mul_node
```

Edugrad: Basic Training Demo

- https://github.com/shanest/edugrad/blob/master/examples/toy_half_sum/main.py
  - Trains an MLP on f(x) = sum(x)/2 for bit vectors x
- MLP as a nn.Module:
- NB: don't hard-code hyper-parameters like this :)

```python
class MLP(nn.Module):
    def __init__(self, input_size, output_size):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_size, 32)
        self.fc2 = nn.Linear(32, 32)
        self.output = nn.Linear(32, output_size)

    def forward(self, inputs):
        hidden = edugrad.ops.relu(self.fc1(inputs))
        hidden = edugrad.ops.relu(self.fc2(hidden))
        return self.output(hidden)
```

Training Loop

```python
model = MLP(input_size, 1)
optimizer = edugrad.optim.SGD(model.parameters(), lr=1e-3)
train_iterator = edugrad.data.BatchIterator(batch_size=batch_size)

for epoch in range(num_epochs):
    total_loss = 0.0
    for batch in train_iterator(inputs, targets):
        predicted = model(batch.inputs)
        loss = edugrad.ops.mse_loss(predicted, batch.targets)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        total_loss += loss.value
    print(f"Epoch {epoch} loss: {total_loss / train_iterator.num_batches}")
```