"PyTorch - Variables, functionals and Autograd."

Feb 9, 2018

Variables

A **Variable** wraps a Tensor. It supports nearly all the API's defined by a Tensor. Variable also provides a *backward* method to perform backpropagation. For example, to backpropagate a loss function to train model parameter x, we use a variable $loss$ to store the value computed by a loss function. Then, we call *loss.backward* which computes the gradients $\frac{\partial loss}{\partial x}$ for all trainable parameters. PyTorch will store the gradient results back in the corresponding variable x.

Create a 2x2 Variable to store input data:

```
import torch
from torch.autograd import Variable

# Variables wrap a Tensor
x = Variable(torch.ones(2, 2), requires_grad=True)
# Variable containing:
# 1  1
# 1  1
# [torch.FloatTensor of size 2x2]
```

requires_grad indicates whether a variable is trainable. By default, *requires_grad* is False in creating a Variable. If one of the input to an operation requires gradient, its output and its subgraphs will also require gradient. To fine tune just part of a pre-trained model, we can set *requires_grad* to False at the base but then turn it on at the entrance of the subgraphs that we want to retrain.

```
param.requires_grad = True
```

Functions

We can add an operation to create another variable:

```
y = x + 2            # Create y from an operation
# Variable containing:
# 3  3
# 3  3
# [torch.FloatTensor of size 2x2]

z = torch.add(x, y)  # Same as z = x + y
```

We can add more operations:

```
z = y * y * 2

out = z.mean()
# Variable containing:
# 2
# [torch.FloatTensor of size 1]
```

PyTorch executes and Variables and operations immediately. In TensorFlow, the execution is delayed until we execute it in a session later.

Compute gradient

Autograd is a PyTorch package for the differentiation for all operations on Tensors. It performs the backpropagation starting from a variable. In deep learning, this variable often holds the value of the cost function. *backward* executes the backward pass and computes all the backpropagation gradients automatically. We access indvidual gradient through the attributes *grad* of a variable. *x.grad* below returns a 2x2 gradient tensor for $\frac{\partial out}{\partial x}$.

```
out.backward()

print(x.grad)
# Variable containing:
# 3 3
# 3 3
# [torch.FloatTensor of size 2x2]
```

To check the resule, we compute the gradient manually:

$$
\begin{aligned}
\frac{\partial out}{\partial x_i} &= \frac{1}{4}\sum_j \frac{\partial z_j}{\partial x_i} \\
&= \frac{1}{4}\sum_j \frac{\partial 2y_j^2}{\partial x_i} \\
&= \frac{1}{4}\sum_j 4y_j \frac{\partial y_j}{\partial x_i} \\
&= \sum_j (x_j + 2)\frac{\partial (x_j + 2)}{\partial x_i} \\
&= x_i + 2 && \frac{\partial x_j}{\partial x_i} = 0 \text{ if } i \neq j \\
&= 3 && \text{for } x_i = 1
\end{aligned}
$$

Dynamic computation graph

In PyTorch, the variables and functions build a dynamic graph of computation. For every variable operation, it creates at least a single Function node that connects to functions that created a Variable. The attribute *grad_fn* of a variable references the function that creates the variable. x has no function but any variable created by an operation will have a function.

```
x = Variable(torch.ones(2, 2), requires_grad=True)
y = x + 1

print(x.grad_fn)     # None

print(y.grad_fn)     # The Function that create the Variable y
# <AddBackward0 object at 0x102995438>
```

When *backward* is called, it follows backwards with the links created in the graph to backpropagate the gradient.

Dynamic vs Static computation graph (PyTorch vs TensorFlow)

The TensorFlow computation graph is static. Operation executions are delayed until the graph is completed. TensorFlow defines a graph first with placeholders. Once all operations are added, we execute the graph in a session by feeding data into the placeholders. The computation graph is static because it cannot be changed afterwards. We can repeat this process with different batch of data but the graph remains the same.

By design, PyTorch uses a dynamic computation graph. Whenever we create a variable or operations, it is executed immediately. We can add and execute operations anytime before *backward* is called. *backwards* follows the graph backward to compute the gradients. Then the graph will be disposed. (the retain_graph flag can override this behavior but rarely suggested.) For the training data in the next training iteration, a new graph is created. We can use the same code to create the same structure, or create a graph with different operations. In NLP, we deal with variable length sentences. Instead of padding the sentence to a fixed length, we create graphs with different number of LSTM cells based on the sentence's length.

We call this a define-by-run framework. which the backpropagation is based on what has been running in the graph. Since we start a new graph for every iteration, the backpropagation path can be different for each iteration.

Example

PyTorch provides functions to make training easier without processing the raw data of the gradients directly. This is demonstrated here.

But to tie all the APIs together, here is an example in doing backpropagation manually.

```
import torch
from torch.autograd import Variable

dtype = torch.FloatTensor
N, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(N, D_in).type(dtype), requires_grad=False)
y = Variable(torch.randn(N, D_out).type(dtype), requires_grad=False)

w1 = Variable(torch.randn(D_in, H).type(dtype), requires_grad=True)
w2 = Variable(torch.randn(H, D_out).type(dtype), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)

    loss = (y_pred - y).pow(2).sum()
    print(t, loss.data[0])

    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```

```
    w1.grad.data.zero_()
    w2.grad.data.zero_()
```

TensorFlow example

Just for comparison, here is the equivalent code in TensorFlow:

```
import tensorflow as tf
import numpy as np

N, D_in, H, D_out = 64, 1000, 100, 10

x = tf.placeholder(tf.float32, shape=(None, D_in))
y = tf.placeholder(tf.float32, shape=(None, D_out))

w1 = tf.Variable(tf.random_normal((D_in, H)))
w2 = tf.Variable(tf.random_normal((H, D_out)))

h = tf.matmul(x, w1)
h_relu = tf.maximum(h, tf.zeros(1))
y_pred = tf.matmul(h_relu, w2)

loss = tf.reduce_sum((y - y_pred) ** 2.0)

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

learning_rate = 1e-6
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    x_value = np.random.randn(N, D_in)
    y_value = np.random.randn(N, D_out)
    for _ in range(500):
                # Run the same graph with different batch of training da
        loss_value, _, _ = sess.run([loss, new_w1, new_w2],
                                    feed_dict={x: x_value, y: y_value})
        print(loss_value)
```

Extending Function

We can create custom function to add operations to the computation graph. Here is an example to build your own ReLU function:

```
import torch
from torch.autograd import Variable


class MyReLU(torch.autograd.Function):

    @staticmethod
    def forward(ctx, input):
        ctx.save_for_backward(input)
        return input.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_output):
        input, = ctx.saved_tensors
        grad_input = grad_output.clone()
        grad_input[input < 0] = 0
        return grad_input


dtype = torch.FloatTensor
N, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(N, D_in).type(dtype), requires_grad=False)
y = Variable(torch.randn(N, D_out).type(dtype), requires_grad=False)

# Create random Tensors for weights, and wrap them in Variables.
w1 = Variable(torch.randn(D_in, H).type(dtype), requires_grad=True)
w2 = Variable(torch.randn(H, D_out).type(dtype), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    # To apply our Function, we use Function.apply method. We alias thi
    relu = MyReLU.apply
    y_pred = relu(x.mm(w1)).mm(w2)

    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data

    w1.grad.data.zero_()
    w2.grad.data.zero_()
```

Access data

We can access the raw data of a variable with *data*.

```
x = torch.randn(3)
x = Variable(x, requires_grad=True)

y = x * 2
while y.data.norm() < 100:
    y = y * 2

print(y)
# Variable containing:
#  48.8215
# 162.7583
# -69.1980
# [torch.FloatTensor of size 3]
```

Backward (non-scalar output)

out below is a scalar and we do not need to specify any parameters for *backward*. By default, we backpropagate a gradient of 1.0 back.

```
out = z.mean()
out.backward()    # Same as out.backward(torch.FloatTensor([1.0]))
```

y below is a Tensor of size 3. *backward* requires a Tensor to specify each backpropagation gradient if the variable is not a scalar. To match each element of *y*, *gradients* needs to match the size of *y*. In some situtation, the gradient values are computed from the model predictions and the true labels.

```
gradients = torch.FloatTensor([0.1, 1.0, 0.0001])
y.backward(gradients)

print(x.grad)
# Variable containing:
#  6.4000               - backpropagate gradient of 0.1
# 64.0000               - backpropagate gradient of 1.0
#  0.0064
# [torch.FloatTensor of size 3]
```

channel_panel • 9 months ago

instead of:
`# Variable containing:
2`
shouldn't it be 18?

1 • Reply • Share ›

Marcin • 9 months ago

Great article!

• Reply • Share ›