PyTorch Workshop

Nate Young[1] Humza Iqbal[1]

[1]Machine Learning at Berkeley

October 25 2017

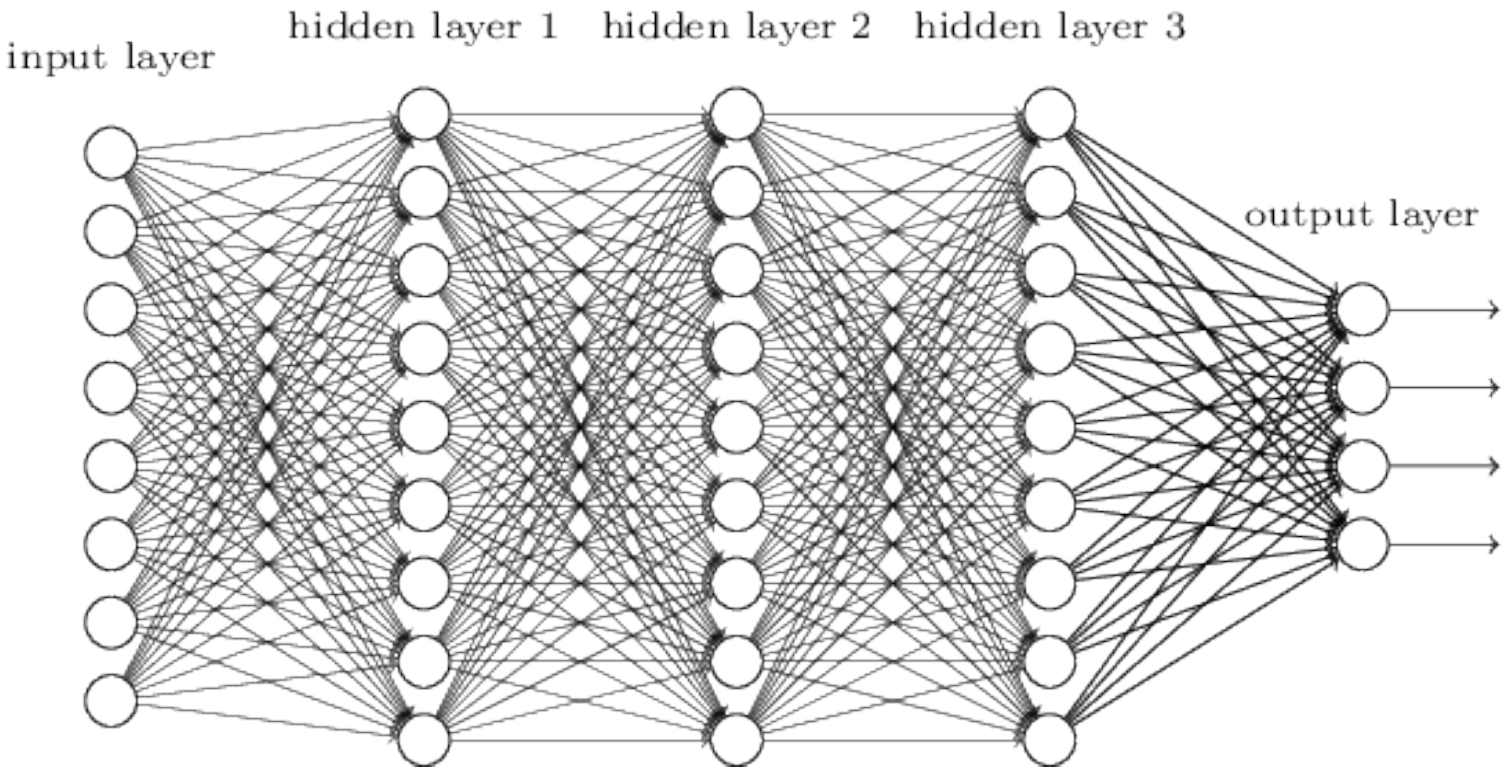
Who this workshop is for

- People who know the general ideas of deep learning and deep learning frameworks.
- People who want a ground up introduction to PyTorch

Basic Background

$$\vec{f}(\vec{x}) = \text{softmax}(W_3 \cdot \text{relu}(W_2 \cdot \text{relu}(W_1 \cdot \vec{x})))$$

or:



We want a framework that makes sense

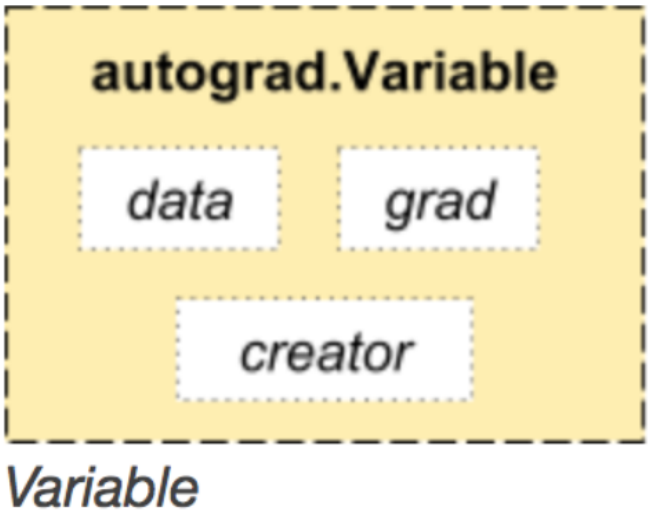
- We want to be able to define a general computation structure
- We want the framework to differentiate through the structure for us.
- We want the way of defining the structure to be as general as possible
- We want the way of defining the structure to be modular, so you can have weird graphs and do things like transfer learning.
- We want very little boilerplate for common tasks.

Define by Run

- In PyTorch the graph structure is decided at run time, in TensorFlow on the other hand the graph structure is decided before the code is compiled.
- The advantage to define by run is more flexibility for dynamic inputs.

PyTorch Objects

- Tensors hold data like numpy arrays (but can do operations on a GPU)
- Variables hold Tensors for data and gradients (not placeholders; there are no placeholders)



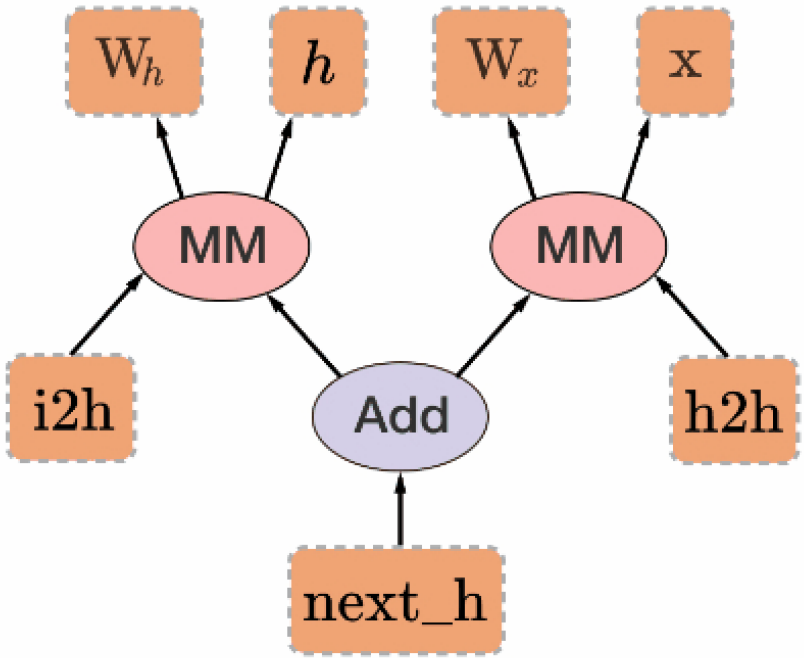
Computation graph

A graph is created on the fly

```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
```



PyTorch MNIST Example (Model)

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x)
```

PyTorch MNIST Example (Training)

```
model = Net()
if args.cuda:
    model.cuda()

optimizer = optim.SGD(model.parameters(), lr=args.lr, momentum=args.momentum)

def train(epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        if args.cuda:
            data, target = data.cuda(), target.cuda()
        data, target = Variable(data), Variable(target)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % args.log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.data[0]))
```

Good things about PyTorch

- You define your models by writing normal code within subclasses of torch.nn.Module.
- Define by run makes everything very pythonic
- As long as you don't do anything weird in Module.forward(), Module.backward() will work just fine without you touching it.
- Most operations make perfect sense; heavily numpy-inspired.
- Includes Sequential and other Module subclasses to make the common things simple (we really didn't have to write as much code).

Tensorflow MNIST (Model)

```
def inference(images, hidden1_units, hidden2_units):
  """Build the MNIST model up to where it may be used for inference.

  Args:
    images: Images placeholder, from inputs().
    hidden1_units: Size of the first hidden layer.
    hidden2_units: Size of the second hidden layer.

  Returns:
    softmax_linear: Output tensor with the computed logits.
  """
  # Hidden 1
  with tf.name_scope('hidden1'):
    weights = tf.Variable(
        tf.truncated_normal([IMAGE_PIXELS, hidden1_units],
                            stddev=1.0 / math.sqrt(float(IMAGE_PIXELS))),
        name='weights')
    biases = tf.Variable(tf.zeros([hidden1_units]),
                         name='biases')
    hidden1 = tf.nn.relu(tf.matmul(images, weights) + biases)
  # Hidden 2
  with tf.name_scope('hidden2'):
    weights = tf.Variable(
        tf.truncated_normal([hidden1_units, hidden2_units],
                            stddev=1.0 / math.sqrt(float(hidden1_units))),
        name='weights')
    biases = tf.Variable(tf.zeros([hidden2_units]),
                         name='biases')
    hidden2 = tf.nn.relu(tf.matmul(hidden1, weights) + biases)
  # Linear
  with tf.name_scope('softmax_linear'):
    weights = tf.Variable(
        tf.truncated_normal([hidden2_units, NUM_CLASSES],
                            stddev=1.0 / math.sqrt(float(hidden2_units))),
        name='weights')
    biases = tf.Variable(tf.zeros([NUM_CLASSES]),
                         name='biases')
    logits = tf.matmul(hidden2, weights) + biases
  return logits
```

Tensorflow MNIST (Loss)

```
def loss(logits, labels):
  """Calculates the loss from the logits and the labels.

  Args:
    logits: Logits tensor, float - [batch_size, NUM_CLASSES].
    labels: Labels tensor, int32 - [batch_size].

  Returns:
    loss: Loss tensor of type float.
  """
  labels = tf.to_int64(labels)
  cross_entropy = tf.nn.sparse_softmax_cross_entropy_with_logits(
      labels=labels, logits=logits, name='xentropy')
  return tf.reduce_mean(cross_entropy, name='xentropy_mean')
```

Tensorflow MNIST (Training)

```
def training(loss, learning_rate):
  """Sets up the training Ops.

  Creates a summarizer to track the loss over time in TensorBoard.

  Creates an optimizer and applies the gradients to all trainable variables.

  The Op returned by this function is what must be passed to the
  `sess.run()` call to cause the model to train.

  Args:
    loss: Loss tensor, from loss().
    learning_rate: The learning rate to use for gradient descent.

  Returns:
    train_op: The Op for training.
  """
  # Add a scalar summary for the snapshot loss.
  tf.summary.scalar('loss', loss)
  # Create the gradient descent optimizer with the given learning rate.
  optimizer = tf.train.GradientDescentOptimizer(learning_rate)
  # Create a variable to track the global step.
  global_step = tf.Variable(0, name='global_step', trainable=False)
  # Use the optimizer to apply the gradients that minimize the loss
  # (and also increment the global step counter) as a single training step.
  train_op = optimizer.minimize(loss, global_step=global_step)
  return train_op
```

Installation

- Available on Linux and OSX only
- No differing compilations so no compiling it to support cuda
- The Great Thing About Pytorch Is That It Just Works!(... but sometimes it doesn't)

Problems with PyTorch

- Comparatively little support, few StackOverflow questions, etc. You may be one of the first to get any particular error.
- Fewer specific operations; gap will close
- Lots of researchers (but not all) are using pytorch
- Tensorboard is not *officially* supported yet
- More difficult for production, due to scaling and speed (but you can convert models from pytorch to other frameworks
- Slower, in some cases
- Use PyTorch for all your research purposes now. It is in beta, but it is the future.

Demos: Dynamic net

Demos: Nearest Neighbor

Demos: Custom Scipy Op