PyTorch® and Gradients

The proliferation of Deep Learning (DL) toolkits (DLT), such as PyTorch or Autograd, have brought a number of easy to use techniques to mathematics computations. In particular, Deep Learning toolkits have a well developed ability to compute gradients *(i.e. derivatives)* that mathematicians and computer scientists can use for modeling and solving non-machine learning problems. This is an evolving series of notes that uses PyTorch and HIPS (Harvard Intelligent Probabilistic Systems Group) Autograd to solve increasingly interesting problems. All of the material included in this series of notes can be found at Gradient Methods, so feel free to explore the Colaboratory notebooks to see more details. The goal is to leverage the DLT to solve numerical Differential Equations and Transfer Learning.

Gradient Computation

As a first step, we need to compute gradients for functions. Given that DL back-propagation uses the chain rule to compute gradients, we fully expect that we should be able to leverage this infrastructure to compute general gradients for functions. Setting up our problem, given a function $f(x)$ and a set of points $\{x_i\}$, we want to compute the set $\{f'(x_i)\}$. The solution for this is fairly simple with these four steps:

- Define your function so it operates on a tensor and outputs a tensor
- Define a loss function to be **L = sum(**$\{f(x_i)\}$**)**
- Use the DLT grad function to compute **f' = grad(**L**)**
- Define a convenience function that outputs both $f(\{x_i\})$ and $f'(\{x_i\})$

In code, for Autograd, a typical example is

```
'''Python
def f(x):
  y = x*x + 4.0
  return y
def loss_f(x):
  loss = np.sum(f(x))
  return loss
f_p = grad(loss_f)
def g(x):
  return f(x), f_p(x)
#Compute points
y, y_p = g(x_vals)
```

```
'''
```

The key for using the DLT for function and derivative computation is understanding the loss function. The DLT takes the derivative relative to a loss, but using a sum means that the derivative of relative to each point is exactly what we want.
As the next code shows, this can easily be extended to second derivatives -

```
'''Python
def f(x):
  y = x*x + 4.0
  return y
def loss_f(x):
  loss = np.sum(f(x))
  return loss
f_p = grad(loss_f)
def loss_fp(x):
  loss = np.sum(f_p(x))
  return loss
f_pp = grad(loss_fp)
def g(x):
  return f(x), f_p(x)
def h(x):
  return f(x), f_p(x), f_pp(x)
'''
```

For PyTorch, the code is similar, but it uses the backward function to compute the gradient

```
'''Python
def f(x):
  y = x * x + 2.0
  return y
def loss_f(x):
  z = f(x).sum()
  return z
def f_p(x):
  z = loss_f(x)
  z.backward()
  return x.grad
'''
```

For a more complete set of examples, consider reviewing "0-Automatic_Differentiation.ipynb" in Gradient Methods. Overall DLTs provide robust capabilities for automatic numerical differentiation; therefore, this provides a good start for solving interesting numerical problems. The next step is solving linear inverse problems.

Solving Linear Inverse Problems

Many numerical problems eventually are written so that it reduces to solving a Linear Algebra problem $Ax = b$; however, computing the inverse of a matrix is often numerically unstable. In many cases, this is rewritten to use iterative methods using this method

Minimize the value of Loss = $(||Ax - b||_2)^2$

Or within the framework of DLT, minimize the average of this Loss definition, denoted by *mle*,

This minimization is so common that it is a standard capability within DLT frameworks and they all have methods for efficient computation of the *mle*.

The Colaboratory file Function Minimization within the above repository has some simple examples for using the gradient methods to minimize functions. The highlights are summarized in the following paragraphs.

As a starting point, PyTorch includes many methods for minimizing a loss function in the *"optim"* package, including:

- SGD - Stochastic Gradient Descent
- Adam - Adam Paper
- Adadelta
- Adagrad
- …

Each of these have strengths and weaknesses, so there is no single method for all problems. I normally use SGD, Adam or a combination of these for solving problems. For these snippets, we use the function $x^2 + y^2 + 4$. These code snippets show how easy the DLT is to use for solving minimization problems:

```
'''Python
offset = torch.Tensor([4.]).double()
#This computes a stopping criteria
def compute_loss_error(l1, l2):
  return np.abs(l1 - l2)/np.max([l1,l2])
#This computes the loss function
def loss_f(in_pos):
  y = in_pos.pow(2).sum() + offset
  return y
#define the first points for "guess"
x1 = torch.Tensor(std_val * (np.random.randn(1,N) +
mean_val),device=device).double()
```

```
x1.requires_grad = True

...

while i<=max_iter and adj_error > iter_stop:
  optimizer.zero_grad()
  loss = loss_f(x1)
  loss.backward()
  optimizer.step()
  if i % 10 == 0:
    curr_loss = loss_f(x1).data.numpy()[0]
    adj_error = compute_loss_error(last_loss,curr_loss)
    last_loss = curr_loss
  i = i + 1

'''
```

The complete code is in the "Function Minimization" Colaboratory file, but the minimization function is summarized by these steps

```
'''Python
  optimizer.zero_grad()
  loss = loss_f(x1)
  loss.backward()
  optimizer.step()
'''
```

At each step, we

- Zero the gradient
- Compute the loss
- Compute the gradient using backward
- Take a gradient step

The only difference between using SGD and Adam is in the definition of the optimizer. These are summarized below:

```
'''Python
#SGD optimizer
optimizer = torch.optim.SGD([x1], lr=.1)
#Adam optimizer
optimizer = torch.optim.Adam([x1], lr=4., betas=(.5,.9))
''
```

Otherwise the processing is the same.

Of interest for some problems is using multiple different solvers at different stages of the minimization problem To illustrate this, the "Function Minimization" Colaboratory file considers the function $x^6 + y^6 + z^6 + w^6 + 4$. The basic SGD algorithm fails with *nan* values, but the Adam algorithm manages to converge. Furthermore, we improve the convergence rate by considering multiple algorithms as seen below:

```
'''Python
optimizer = torch.optim.Adam([x1], lr=.04, betas=(.5,.9))
optimizer2 = torch.optim.Adam([x1], lr=.04, betas=(.9,.95))
optimizer3 = torch.optim.Adam([x1], lr=.04, betas=(.9,.999))

...

while i<=max_iter and adj_error > iter_stop:
  if adj_error > .1:
    optimizer.zero_grad()
    loss = loss_f(x1)
    loss.backward()
    optimizer.step()
  elif adj_error > .01:
    optimizer2.zero_grad()
    loss = loss_f(x1)
    loss.backward()
    optimizer2.step()
  else:
    optimizer3.zero_grad()
    loss = loss_f(x1)
    loss.backward()
    optimizer3.step()
  if i % 10 == 0:
    curr_loss = loss_f(x1).data.numpy()[0]
    adj_error = compute_loss_error(last_loss,curr_loss)
    last_loss = curr_loss
  i = i + 1
'''
```

The integration of these solvers into PyTorch, makes it easy to use this type of capability while solving problems