PyTorch and Automatic Differentiation

David I. Inouye

Main functionalities

1. Automatic gradient calculations
2. GPU acceleration (probably won't cover in class)
3. Neural network functions (simplify things a good deal)

(PyTorch has a very nice tutorial that covers more basics: https://pytorch.org/tutorials/beginner/basics/intro.html)

```
import numpy as np
import torch # PyTorch library
import scipy.stats
import matplotlib.pyplot as plt
import seaborn as sns
# To visualize computation graphs
# See: https://github.com/szagoruyko/pytorchviz
# Uncomment the following line to install on Google colab
#%pip install -U git+https://github.com/szagoruyko/pytorchviz.git@master
from torchviz import make_dot, make_dot_from_trace
sns.set()
%matplotlib inline
```

PyTorch: Some basics of converting between NumPy and Torch

See link below for more information:

https://pytorch.org/tutorials/beginner/former_torchies/tensor_tutorial.html#numpy-bridge

```
1  # Torch and numpy
2  x = torch.linspace(-5,5,10)
3  print(x)
4  print(x.dtype)
5  print('NOTE: x is float32 (torch default is float32)')
6  x_np = np.linspace(-5,5,10)
7  y = torch.from_numpy(x_np)
8  print(y)
9  print(y.dtype)
10 print('NOTE: y is float64 (numpy default is float64)')
11 print(y.float().dtype)
12 print('NOTE: y can be converted to float32 via `float()`')
13 print(x.numpy())
14 print(y.numpy())
```

```
tensor([-5.0000, -3.8889, -2.7778, -1.6667, -0.5556,  0.5556,
1.6667,  2.7778,
         3.8889,  5.0000])
torch.float32
NOTE: x is float32 (torch default is float32)
tensor([-5.0000, -3.8889, -2.7778, -1.6667, -0.5556,  0.5556,
1.6667,  2.7778,
         3.8889,  5.0000], dtype=torch.float64)
torch.float64
NOTE: y is float64 (numpy default is float64)
torch.float32
NOTE: y can be converted to float32 via `float()`
[-5.         -3.8888888  -2.7777777  -1.6666665  -0.55555534
0.55555534
  1.6666665   2.7777777   3.8888888   5.        ]
[-5.         -3.88888889 -2.77777778 -1.66666667 -0.55555556
0.55555556
  1.66666667  2.77777778  3.88888889  5.        ]
```

Torch can be used to do simple computations just like numpy

```python
# Explore gradient calculations
x = torch.tensor(5.0)
y = 3*x**2 + x
print(x, x.grad)
print(y)
```

```
tensor(5.) None
tensor(80.)
```

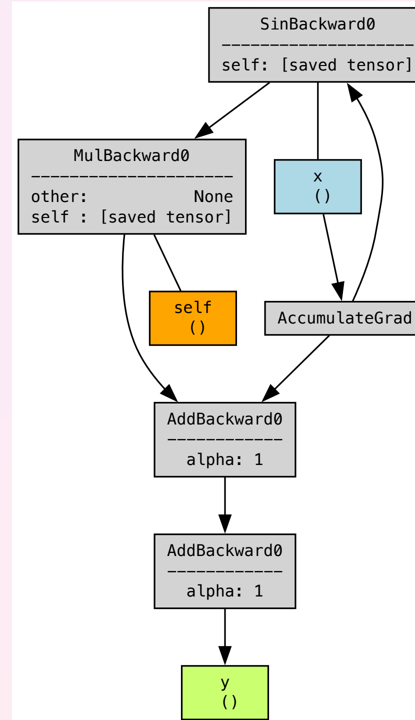
PyTorch automatically creates a computation graph for computing gradients if `requires_grad=True`

- IMPORTANT: You must set `requires_grad=True` for any torch tensor for which you will want to compute the gradient (usually model parameters)
  - These are known as the "leaf nodes" or "input nodes" of a gradient computation graph
  - Note that some leaf nodes will not need gradient (e.g., constant matrices like the training data)

Okay let's compute and show the computation graph

```
# Explore gradient calculations
x = torch.tensor(5.0, requires_grad=True)
# A constant input tensor that does not require gradient
c = torch.tensor(3.0)
y = c*torch.sin(x) + x + c
print(x, x.grad)
print(y)
make_dot(
    y, dict(x=x, c=c, y=y),
    show_attrs=True, show_saved=True)
```

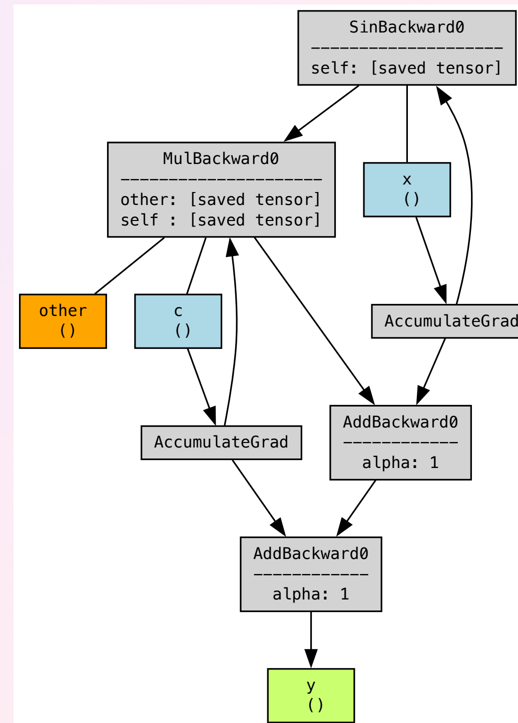
```
tensor(5., requires_grad=True) None
tensor(5.1232, grad_fn=<AddBackward0>)
```



Okay let's compute and show the computation graph

```
 1  # Explore gradient calculations
 2  x = torch.tensor(5.0, requires_grad=True)
 3  # Change to compute grad over this variable too
 4  c = torch.tensor(3.0, requires_grad=True)
 5  y = c*torch.sin(x) + x + c
 6  print(x, x.grad)
 7  print(y)
 8  make_dot(
 9      y, dict(x=x, c=c, y=y),
10      show_attrs=True, show_saved=True)
```

```
tensor(5., requires_grad=True) None
tensor(5.1232, grad_fn=<AddBackward0>)
```



We can even do loops

```python
# We can even do loops
x = torch.tensor(1.0, requires_grad=True)
y = x
for i in range(3):
    y = y*(y+1)
print(x, x.grad)
print(y)
make_dot(
    y, dict(x=x, y=y),
    show_attrs=True, show_saved=True)
```

```
tensor(1., requires_grad=True) None
tensor(42., grad_fn=<MulBackward0>)
```

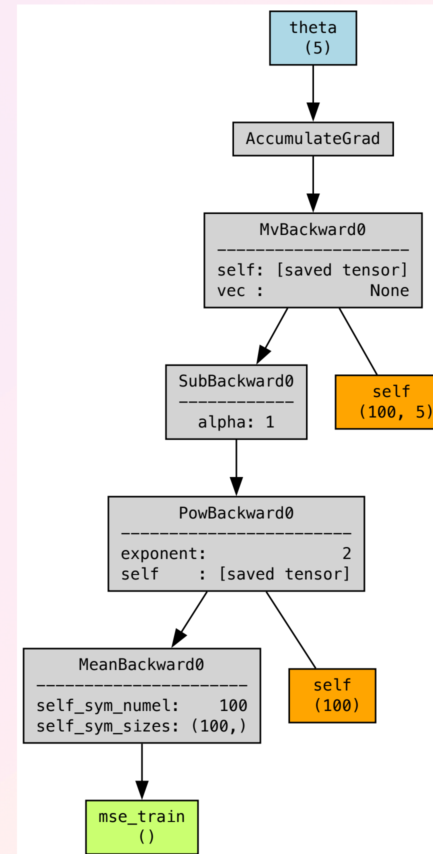
Let’s do this for a more complex ML example

Below is a simple linear regression error computation for a random model

```
# Data
# A simple tensor example
rng = torch.manual_seed(42)
X_train = torch.randn(100, 5) #.requires_grad_(True)
y_train = torch.mean(X_train, axis=1)

# Model
theta = torch.randn(5).requires_grad_(True)
y_pred = torch.matmul(X_train, theta)

# Error
mse_train = torch.mean((y_train - y_pred)**2)
make_dot(
    mse_train,
    dict(X_train=X_train, mse_train=mse_train, theta=theta),
    show_attrs=True, show_saved=True
)
```



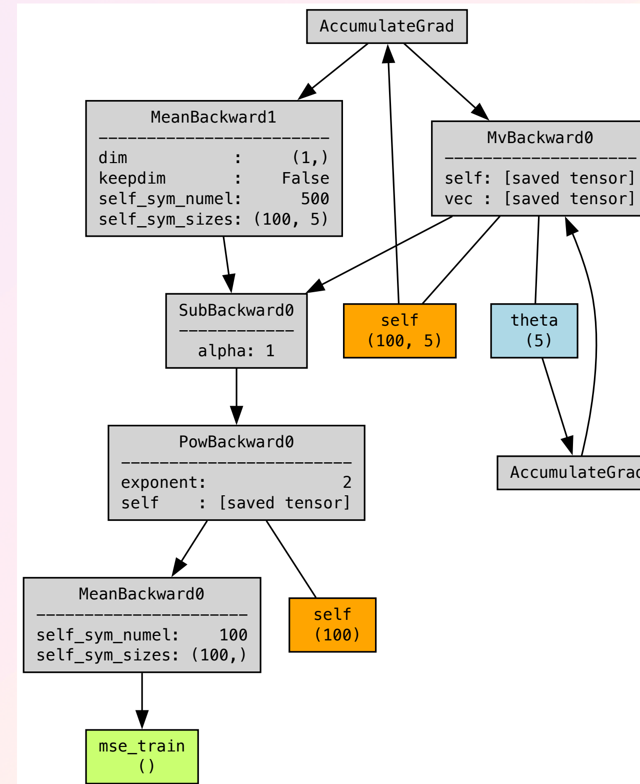
While only the parameters should "require_grad" in usual ML optimization, you could compute gradients for other inputs (e.g., creating adversarial examples via optimization)

```python
# A simple tensor example
# Data
rng = torch.manual_seed(42)
X_train = torch.randn(100, 5).requires_grad_(True)
y_train = torch.mean(X_train, axis=1)

# Model
theta = torch.randn(5).requires_grad_(True)
y_pred = torch.matmul(X_train, theta)

# Error
mse_train = torch.mean((y_train - y_pred)**2)

make_dot(
    mse_train,
    dict(X_train=X_train, mse_train=mse_train, theta=theta),
    show_attrs=True, show_saved=True
)
```



Now we can automatically compute gradients via backward call

Note that tensor has grad_fn for doing the backwards computation

```python
x = torch.tensor(5.0, requires_grad=True)
# A constant input tensor that does not require gradient
c = torch.tensor(3.0)
y = c*torch.sin(x) + x + c
print(x, x.grad)
print(y)

y.backward()
print(x, x.grad)
print(y)
```

```
tensor(5., requires_grad=True) None
tensor(5.1232, grad_fn=<AddBackward0>)
tensor(5., requires_grad=True) tensor(1.8510)
tensor(5.1232, grad_fn=<AddBackward0>)
```

A call to `backward` will free up the **implicit** computation graph (i.e., removed saved tensors)

```
try:
    y.backward()
    print(x, x.grad)
    print(y)
except Exception as e:
    print(e)
```

```
Trying to backward through the graph a second time (or directly
access saved tensors after they have already been freed). Saved
intermediate values of the graph are freed when you call
.backward() or autograd.grad(). Specify retain_graph=True if
you need to backward through the graph a second time or if you
need to access saved tensors after calling backward.
```

Gradients accumulate, i.e., sum, from multiple backward calls

```python
x = torch.tensor(5.0, requires_grad=True)
for i in range(2):
    y = 3*x**2
    y.backward()
    print(x, x.grad)
    print(y)
```

```
tensor(5., requires_grad=True) tensor(30.)
tensor(75., grad_fn=<MulBackward0>)
tensor(5., requires_grad=True) tensor(60.)
tensor(75., grad_fn=<MulBackward0>)
```

Thus, must zero gradients before calling `backward()`

```python
# Thus if before calling another gradient iteration,
#  zero the gradients
x.grad.zero_()
print(x, x.grad)

# Now that gradient is zero, we can do again
y = 3*x**2
y.backward()
print(x, x.grad)
print(y)
```

```
tensor(5., requires_grad=True) tensor(0.)
tensor(5., requires_grad=True) tensor(30.)
tensor(75., grad_fn=<MulBackward0>)
```

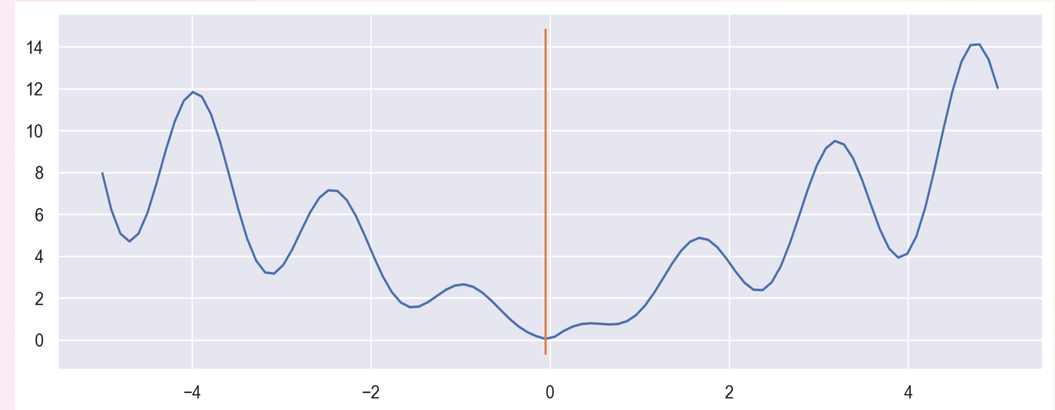
More complicated gradients example

```python
x = torch.arange(5, dtype=torch.float32).requires_grad_(True)
y = torch.mean(torch.log(x**2+1)+5*x)
y.backward()
print(y)
print(x)
print('Grad', x.grad)
```

```
tensor(11.4877, grad_fn=<MeanBackward0>)
tensor([0., 1., 2., 3., 4.], requires_grad=True)
Grad tensor([1.0000, 1.2000, 1.1600, 1.1200, 1.0941])
```

Now let's optimize a non-convex function (pretty much all DNNs)

```
1 def objective(theta):
2     return theta*torch.cos(4*theta) + 2*torch.abs(theta)
3
4 theta = torch.linspace(-5, 5, steps=100)
5 y = objective(theta)
6 theta_true = float(theta[np.argmin(y)])
7 plt.figure(figsize=(12,4))
8 plt.plot(theta.numpy(), y.numpy())
9 plt.plot(theta_true * np.ones(2), plt.ylim())
```



Let’s use simple gradient descent on this function

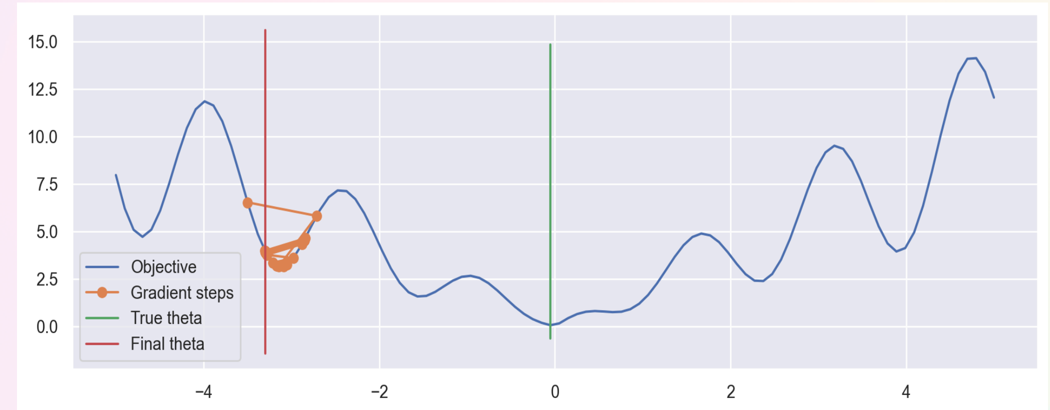
```
1  # Just a function for visualizing results
2  def visualize_results(theta_arr, obj_arr, objective, theta_tr
3      if vis_arr is None:
4          vis_arr = np.linspace(np.min(theta_arr), np.max(theta
5      fig = plt.figure(figsize=(12,4))
6      plt.plot(
7          vis_arr,
8          [
9              objective(torch.tensor(theta)).numpy()
10             for theta in vis_arr
11         ],
12         label='Objective')
13     plt.plot(theta_arr, obj_arr, 'o-', label='Gradient steps'
14     if theta_true is not None:
15         plt.plot(np.ones(2)*theta_true, plt.ylim(),
16             label='True theta')
17     plt.plot(np.ones(2)*theta_arr[-1], plt.ylim(),
18             label='Final theta')
19     plt.legend()
20     plt.show()
```

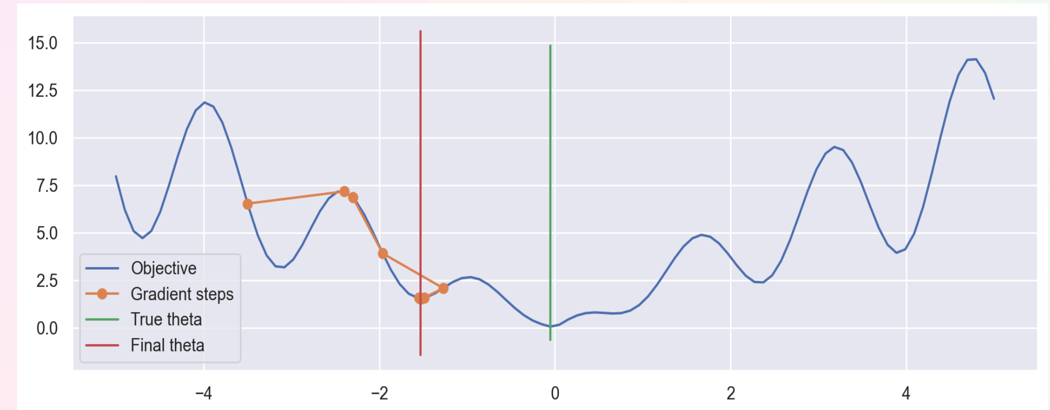
Let's use simple gradient descent on this function

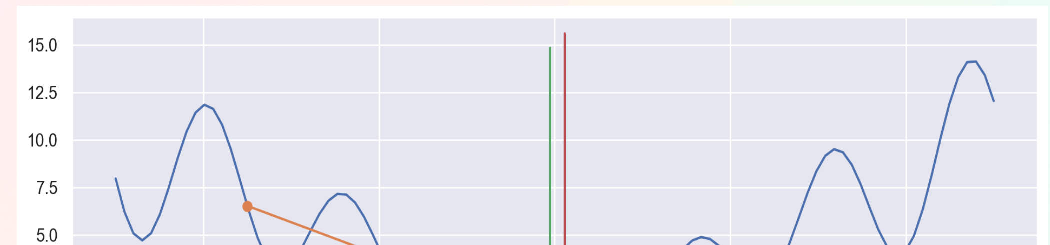
```
def gradient_descent(objective, step_size=0.05,
        max_iter=100, init=0):
    # Initialize
    theta_hat = torch.tensor(init, requires_grad=True)
    theta_hat_arr = [theta_hat.detach().numpy().copy()]
    obj_arr = [objective(theta_hat).detach().numpy()]
    # Iterate
    for i in range(max_iter):
        # Compute gradient
        if theta_hat.grad is not None:
            theta_hat.grad.zero_()
        out = objective(theta_hat)
        out.backward()

        # Update theta in-place
        with torch.no_grad():
            theta_hat -= step_size * theta_hat.grad
        theta_hat_arr.append(
            theta_hat.detach().numpy().copy())
        obj_arr.append(objective(theta_hat).detach().numpy())
    return np.array(theta_hat_arr), np.array(obj_arr)

# 0.05 doesn't escape, 0.07 does, 0.15 gets much closer
for step_size in [0.05, 0.07, 0.15]:
    theta_hat_arr, obj_arr = gradient_descent(
        objective, step_size=step_size,
        init=-3.5, max_iter=100)
```







Putting it all together for ML models

- PyTorch has many helper functions to handle much of stochastic gradient descent or using other optimizers
- Example from https://pytorch.org/tutorials/beginner/examples_nn/two_layer_net_optim.html

Putting it all together for ML models

```
1  import torch
2
3  # N is batch size; D_in is input dimension;
4  # H is hidden dimension; D_out is output dimension.
5  N, D_in, H, D_out = 64, 1000, 100, 10
6
7  # Create random Tensors to hold inputs and outputs
8  x = torch.randn(N, D_in)
9  y = torch.randn(N, D_out)
10
11 # Use the nn package to define our model and loss function.
12 model = torch.nn.Sequential(
13     torch.nn.Linear(D_in, H),
14     torch.nn.ReLU(),
15     torch.nn.Linear(H, D_out),
16 )
17 loss_fn = torch.nn.MSELoss(reduction='sum')
18
19 # Use the optim package to define an Optimizer that will upda
20 # the model for us. Here we will use Adam; the optim package
21 # optimization algoriths. The first argument to the Adam cons
22 # optimizer which Tensors it should update.
23 learning_rate = 1e-4
24 optimizer = torch.optim.Adam(model.parameters(), lr=learning_
25 for t in range(500):
26     # Forward pass: compute predicted y by passing x to the m
```

```
99 71.03107452392578
199 1.639991283416748
299 0.0156068941578269
399 6.425016181310639e-05
499 8.307362975301658e-08
```

A few more details `autograd` and `backward()` function

See https://pytorch.org/tutorials/beginner/basics/autogradqs_tutorial.html for more details.

- Jacobian

$$J = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$

Backward computes Jacobian transpose vector product

$$J^T \cdot v = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix} \begin{pmatrix} \frac{\partial l}{\partial y_1} \\ \vdots \\ \frac{\partial l}{\partial y_m} \end{pmatrix} = \begin{pmatrix} \frac{\partial l}{\partial x_1} \\ \vdots \\ \frac{\partial l}{\partial x_n} \end{pmatrix}$$

Simplification is when output is scalar than the derivative is assumed to be 1

- Example: $y = b^T x, z = \exp(y)$
  - $J_z = [[]], v=[1], J_z^T v = $
  - $J_y = \begin{bmatrix} \frac{dy}{dx_1} & \frac{dy}{dx_2} & \dots & \frac{dy}{dx_5} \end{bmatrix}^T, v = \frac{dz}{dy}, J_y^T v = \begin{bmatrix} \frac{dz}{dx_1} & \frac{dz}{dx_2} & \dots & \frac{dz}{dx_5} \end{bmatrix}^T = \nabla_x z(x)$

Simplification is when output is scalar than the derivative is assumed to be 1

```python
x = (2.0 * torch.ones(5).float()).requires_grad_(True)
b = torch.arange(5).float()
y = torch.dot(b, x)
y.retain_grad()
z = torch.log(y)
z.retain_grad()
z.backward()

def print_grad(a):
    print(a, a.grad)
print_grad(y)
print_grad(x)
```

```
tensor(20., grad_fn=<DotBackward0>) tensor(0.0500)
tensor([2., 2., 2., 2., 2.], requires_grad=True)
tensor([0.0000, 0.0500, 0.1000, 0.1500, 0.2000])
```