PyTorch 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
(https://pytorch.org/tutorials/beginner/basics/intro.html)

In [1]:

```python
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
(https://pytorch.org/tutorials/beginner/former_torchies/tensor_tutorial.html#numpy-bridge)
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)
c = torch.tensor(3.0)  # A constant input tensor that does not require grad
# y = c*x**2 + x+c
y = c*torch.sin(x) + x + c
print(x, x.grad)
print(y)
made_dot(y, dict(x=x, c=c, y=y), show_attr=True, show_saved=True)
In [5]: # Explore gradient calculations
x = torch.tensor(5.0, requires_grad=True)
c = torch.tensor(3.0, requires_grad=True)  # Change to compute grad over th
#y = c*x**2 + x+c
y = c*torch.sin(x) + x + c
print(x, x.grad)
print(y)
made_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>)

Out[5]:
In [6]: # 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>)

Out[6]:
Let's do this for a more complex ML example

Below is a simple linear regression error computation for a random model
In [7]:
    # 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)  # Average of x features

    # 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)

Out[7]:

```
theta
(5)

AccumulateGrad

MvBackward
self: [saved tensor]
vec : None

SubBackward0
alpha: 1

PowBackward0
exponent: 2
self : [saved tensor]

MeanBackward0
self_numel: 100
self_sizes: (100,)

mse_train ()
```
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)
# 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)  # Average of x features

## 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)
Now we can automatically compute gradients via backward call

Note that tensor has grad_fn for doing the backwards computation

```
In [9]:
x = torch.tensor(5.0, requires_grad=True)
c = torch.tensor(3.0)  # A constant input tensor that does not require grad
#y = c*x**2 + x+c
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)

```
In [10]:
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 variables 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 variables after calling backward.

Gradients accumulate, i.e., sum, from multiple backward calls
```python
In [11]:
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
In [12]:
# 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
In [13]:
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)
Let's use simple gradient descent on this function.

```python
In [14]:
def objective(theta):
    return theta*torch.cos(4*theta) + 2*torch.abs(theta)

theta = torch.linspace(-5, 5, steps=100)
y = objective(theta)
theta_true = float(theta[np.argmin(y)])
plt.figure(figsize=(12,4))
plt.plot(theta.numpy(), y.numpy())
plt.plot(theta_true * np.ones(2), plt.ylim())
```

Out[14]: [<matplotlib.lines.Line2D at 0x7fab91014100>]

![Graph](image-url)
In [15]:
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)

def visualize_results(theta_arr, obj_arr, objective, theta_true=None, vis_arr=None):
    if vis_arr is None:
        vis_arr = np.linspace(np.min(theta_arr), np.max(theta_arr))
    fig = plt.figure(figsize=(12, 4))
    plt.plot(vis_arr, [objective(torch.tensor(theta)).numpy() for theta in vis_arr], label='Gradient steps')
    plt.plot(theta_arr, obj_arr, 'o-', label='Gradient steps')
    if theta_true is not None:
        plt.plot(np.ones(2)*theta_true, plt.ylim(), label='True theta')
        plt.plot(np.ones(2)*theta_arr[-1], plt.ylim(), label='Final theta')
    plt.legend()

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

visualize_results(theta_hat_arr, obj_arr, objective, theta_true=theta_true,
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/
In [16]:

```python
import torch

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold inputs and outputs
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

# Use the nn package to define our model and loss function.
model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out),
)
loss_fn = torch.nn.MSELoss(reduction='sum')

# Use the optim package to define an Optimizer that will update the weights
# the model for us. Here we will use Adam; the optim package contains many
# optimization algorithms. The first argument to the Adam constructor tells
# optimizer which Tensors it should update.
learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

for t in range(500):
    # Forward pass: compute predicted y by passing x to the model.
    y_pred = model(x)

    # Compute and print loss.
    loss = loss_fn(y_pred, y)
    if t % 100 == 99:
        print(t, loss.item())

    # Before the backward pass, use the optimizer object to zero all of the
    # gradients for the variables it will update (which are the learnable
    # weights of the model). This is because by default, gradients are
    # accumulated in buffers (i.e. not overwritten) whenever .backward() is
    # called. Checkout docs of torch.autograd.backward for more details.
    optimizer.zero_grad()

    # Backward pass: compute gradient of the loss with respect to model
    # parameters
    loss.backward()

    # Calling the step function on an Optimizer makes an update to its
    # parameters
    optimizer.step()
```

A few more details autograd and backward()
function

Jacobian

\[ J = \begin{bmatrix}
\frac{\partial y_1}{\partial x_1} & \ldots & \frac{\partial y_1}{\partial x_n} \\
\vdots & \ddots & \vdots \\
\frac{\partial y_m}{\partial x_1} & \ldots & \frac{\partial y_m}{\partial x_n}
\end{bmatrix} \]

Backward computes Jacobian transpose vector product

\[ J^T \cdot v = \begin{bmatrix}
\frac{\partial y_1}{\partial x_1} & \ldots & \frac{\partial y_1}{\partial x_n} \\
\vdots & \ddots & \vdots \\
\frac{\partial y_m}{\partial x_1} & \ldots & \frac{\partial y_m}{\partial x_n}
\end{bmatrix} \begin{bmatrix}
\frac{\partial l}{\partial y_1} \\
\vdots \\
\frac{\partial l}{\partial y_m}
\end{bmatrix} = \begin{bmatrix}
\frac{\partial l}{\partial x_1} \\
\vdots \\
\frac{\partial l}{\partial x_n}
\end{bmatrix} \]

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

Example: \( y = b^T x, z = \exp(y) \)

- \( J_z = [[\frac{dz}{dy}]], v = [1], J_z^T v = \frac{dz}{dy} \)
- \( J_y = [\frac{dy}{dx_1} \frac{dy}{dx_2} \ldots \frac{dy}{dx_5}]^T, v = \frac{dz}{dy}, J_y^T v = [\frac{dz}{dx_1} \frac{dz}{dx_2} \ldots \frac{dz}{dx_5}]^T = \nabla_x z(x) \)

In [17]:
\[ x = (2.0 \ast \text{torch.ones}(5).\text{float()})\text{.requires_grad(}\text{True}) \]
\[ b = \text{torch.arange}(5).\text{float()} \]
\[ y = \text{torch.dot}(b, x) \]
\[ y\text{.retain_grad()} \]
\[ z = \text{torch.log}(y) \]
\[ z\text{.retain_grad()} \]
\[ z\text{.backward()} \]

```python
def print_grad(a):
    print(a, a.grad)

print_grad(z)
p = print_grad(y)
p = print_grad(x)
```

tensor(2.9957, grad_fn={<LogBackward>}) tensor(1.)
tensor(20., grad_fn={<DotBackward>}) tensor(0.0500)
tensor([2., 2., 2., 2., 2.], requires_grad=True) tensor([0.0000, 0.0500, 0.1000, 0.1500, 0.2000])