Machine Learning
ITCS 4156

Python Stack
Linear Algebra and Optimization in NumPy
Computation Graphs in PyTorch

Razvan C. Bunescu
Department of Computer Science @ CCI
rbunescu@uncc.edu
Python Programming Stack for Deep Learning

• **Python** = object-oriented, interpreted, scripting language.
  – imperative programming, with functional programming features.

• **NumPy** = package for powerful N-dimensional arrays:
  – sophisticated (broadcasting) functions.
  – useful linear algebra, Fourier transform, and random number capabilities.

• **SciPy** = package for numerical integration and optimization.

• **Matplotlib** = comprehensive 2D and 3D plotting library.
Python Programming Stack for Deep Learning

- **PyTorch** = a wrapper of NumPy that enables the use of GPUs and automatic differentiation:
  - **Tensors** similar to NumPy’s ndarray, but can also be used on GPU.

- **Jupyter Notebook** = a web app for creating documents that contain live code, equations, visualizations and markdown text.

- **Anaconda** = an open-source distribution of Python and Python packages:
  - Package versions are managed through Conda.
  - Install all packages above using Anaconda / Conda install.
Anaconda Install

- **Anaconda**: Installation instructions for various platforms can be found at: [https://docs.anaconda.com/anaconda/install/](https://docs.anaconda.com/anaconda/install/)
  - For Mac and Linux users, the system PATH must be updated after installation so that ‘conda’ can be used from the command line.
    - **Mac OS X**:
      - For bash users: `export PATH=~/anaconda3/bin:$PATH`
      - For csh/tcsh users: `setenv PATH ~/anaconda3/bin:$PATH`
    - **For Linux**:
      - For bash users: `export PATH=~/anaconda3/bin:$PATH`
      - For csh/tcsh users: `setenv PATH ~/anaconda3/bin:$PATH`
  - It is recommend the above statement be put in the `~/.bashrc` or `~/.cshrc` file, so that it is executed every time a new terminal window is open.
  - To check that conda was installed, running “conda list” in the terminal should list all packages that come with Anaconda.
A number of tools and libraries that we will use can be configured from Anaconda:

- PyTorch can be installed from Anaconda, with ‘conda’ from the command line:
  - The actual command line depends on the platform as follows:
    - Using the GUI on pytorch.org, choose the appropriate OS, conda, Python 3.6, CUDA or CPU version.
import numpy as np

- np.array()
  - indexing, slices.
- ndarray.shape, .size, .ndim, .dtype, .T
- np.zeros(), np.ones(), np.arange(), np.eye()
  - dtype parameter.
  - tuple (shape) parameter.
- np.reshape(), np.ravel()
- np.amax(), np.maximum(), np.sum(), np.mean(), np.std()
  - axis parameter, also np.ndarray
- np.stack(), np.[hv]stack(), np.column_stack(), np.split()
- np.exp(), np.log(),
- https://docs.scipy.org/doc/numpy/user/quickstart.html
NumPy: Broadcasting

- Broadcasting describes how numpy treats arrays with different shapes during arithmetic operations.
- The smaller array is “broadcast” across the larger array so that they have compatible shapes, subject to broadcasting rules:
  - NumPy compares their shapes element-wise.
  - It starts with the trailing dimensions, and works its way forward.
  - Two dimensions are compatible when:
    - they are equal, or one of them is 1.

- [https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html](https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html)
Other Numpy Functions

- np.dot(), np.vdot()
  - also np.ndarray.
- np.outer(), np.inner()

- import numpy.random as random:
  - randn(), randint(), uniform()

- import numpy.linalg as la:
  - la.norm(), la.det(), la.matrix_rank(), np.trace()
  - la.eig(), la.svd()
  - la.qr(), la.cholesky()

- https://docs.scipy.org/doc/numpy/reference/routines.linalg.html
Logistic Regression: Vectorization

- **Version 1**: Compute gradient component-wise.

\[
\nabla E(w) = \sum_{n=1}^{N} (h_n - t_n) x_n^T
\]

```python
grad = np.zeros(K)
for n in range(N):
    h = sigmoid(w.dot(X[:, n]))
    temp = h - t[n]
    for k in range(K):
        grad(k) = grad(k) + temp * X[k,n]
```
Logistic Regression: Vectorization

- **Version 2**: Compute gradient, partially vectorized.

\[
\nabla E(w) = \sum_{n=1}^{N} (h_n - t_n) x_n^T
\]

```python
grad = np.zeros(K)
for n in range(N):
    grad = grad + (sigmoid(w.dot(X[:, n])) - t[n]) * X[:, n]
```

Logistic Regression: Vectorization

- **Version 3**: Compute gradient, vectorized.

\[
\nabla E(w) = \sum_{n=1}^{N} (h_n - t_n)x_n^T
\]

\[
\text{grad} = X @ (\text{sigmoid}(w . \text{dot}(X)) - t)
\]

```python
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```
import scipy

- scipy.sparse.coo_matrix()
  groundTruth = coo_matrix((np.ones(numCases, dtype = np.uint8),
                           (labels, np.arange(numCases)))).toarray()

- scipy.optimize:
  - scipy.optimize.fmin_l_bfgs_b()
    theta, _, _ = fmin_l_bfgs_b(softmaxCost, theta,
                               args = (numClasses, inputSize, decay, images, labels),
                               maxiter = 100, disp = 1)
  - scipy.optimize.fmin_cg()
  - scipy.minimize

https://docs.scipy.org/doc/scipy-0.10.1/reference/tutorial/optimize.html
Towards PyTorch: Graphs of Computations

• A function $J$ can be expressed by the **composition** of **computational elements** from a given set:
  – logic operators.
  – logistic operators.
  – multiplication and additions.

• The function is defined by a **graph of computations**:
  – A directed acyclic graph, with one node per computational element.
  – Depth of architecture = depth of the graph = longest path from an input node to an output node.
Logistic Regression as a Computation Graph

Inference = Forward Propagation

Learning = Backward Propagation
Neural Network as a Computation Graph

Inference = Forward Propagation

Learning = Backward Propagation
What is PyTorch

- A wrapper of **NumPy** that enables the use of GPUs.
  - **Tensors** similar to NumPy’s ndarray, but can also be used on GPU.

- A flexible deep learning platform:
  - Deep Neural Networks built on a tape-based **autograd** system:
    - Building neural networks using and replaying a tape recorder.
    - **Reverse-mode auto-differentiation** allows changing the network at runtime:
      - The computation graph is created on the fly.
      - Backpropagation is done on the dynamically built graph.

[http://pytorch.org/about/]
Automatic Differentiation

https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html

- Deep learning packages offer automatic differentiation.

- PyTorch has the **autograd** package:
  - `torch.Tensor` the main class; `torch.Function` class also important.
  - When `requires_grad = True`, it tracks all operations on this tensor (e.g. the parameters).
  - An acyclic graph is build **dynamically** that encodes the history of computations, i.e. compositions of functions.
    - TensorFlow compiles **static** computation graphs.
  - To compute the gradient, call `backward()` in a scalar valued Tensor (e.g. the `loss`).
Tensors

- PyTorch **tensors** support the same operations as NumPy.
  - Arithmetic.
  - Slicing and Indexing.
  - Broadcasting.
  - Reshaping.
  - Sum, Max, Argmax, …

- PyTorch tensors can be converted to NumPy tensors.
- NumPy tensors can be converted to PyTorch tensors.

http://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html
Autograd

- The **autograd** package provides automatic differentiation for all operations on Tensors.
  - It is a *define-by-run* framework, which means that the gradient is defined by how your code is run:
    - Every single **backprop** iteration can be different.

- **autograd.Tensor** is the central class of the package.
  - Once you finish your computation you can call `.backward()` and have all the gradients computed automatically.

http://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html
A **Tensor** `v` has three important attributes:

- `v.data` holds the raw tensor value.

- `v.grad` is another Tensor which accumulates the gradient w.r.t. `v`:
  
  The gradient of what?
  
  - The gradient of any variable `u` that uses `v` on which we call `u.backward()`.

  - [http://pytorch.org/docs/master/autograd.html](http://pytorch.org/docs/master/autograd.html)

- `v.grad_fn` stores the **Function** that has created the Tensor `v`:

  - [http://pytorch.org/docs/master/autograd.html](http://pytorch.org/docs/master/autograd.html)
Multivariate Chain Rule for Differentiation

- Multivariate Chain Rule:

\[ f = f(g_1(x), g_2(x), \ldots, g_n(x)) \]

\[ \frac{\partial f}{\partial x} = \sum_{i=1}^{n} \frac{\partial f}{\partial g_i} \frac{\partial g_i}{\partial x} \]

- Example 2:

\[ \text{loss}(x) = (h_1(x) - h_2(x))^2 \]
\[ h_1(x) = 2g_1(x) + 1 \]
\[ h_2(x) = 2g_1(x) + g_2(x) \]
\[ g_1(x) = 3x \]
\[ g_2(x) = x^2 + x \]
PyTorch

• Install using Anaconda:
  – conda install pytorch torchvision -c pytorch
  – http://pytorch.org

• Install from sources:
  – https://github.com/pytorch/pytorch#from-source

• Tutorials:
  – http://pytorch.org/tutorials/
  – http://pytorch.org/tutorials/beginner/pytorch_with_examples.html