14.332.435/16.332.530
Introduction to Deep Learning

Lecture 6
Code Case Study

Yuqian Zhang

Department of Electrical and Computer Engineering

Python Software Foundation [US] | https://www.python.org/doc/

Python | PSF | Docs | PyPI | Jobs | Community

python™

Donate | Search | GO | Socialize

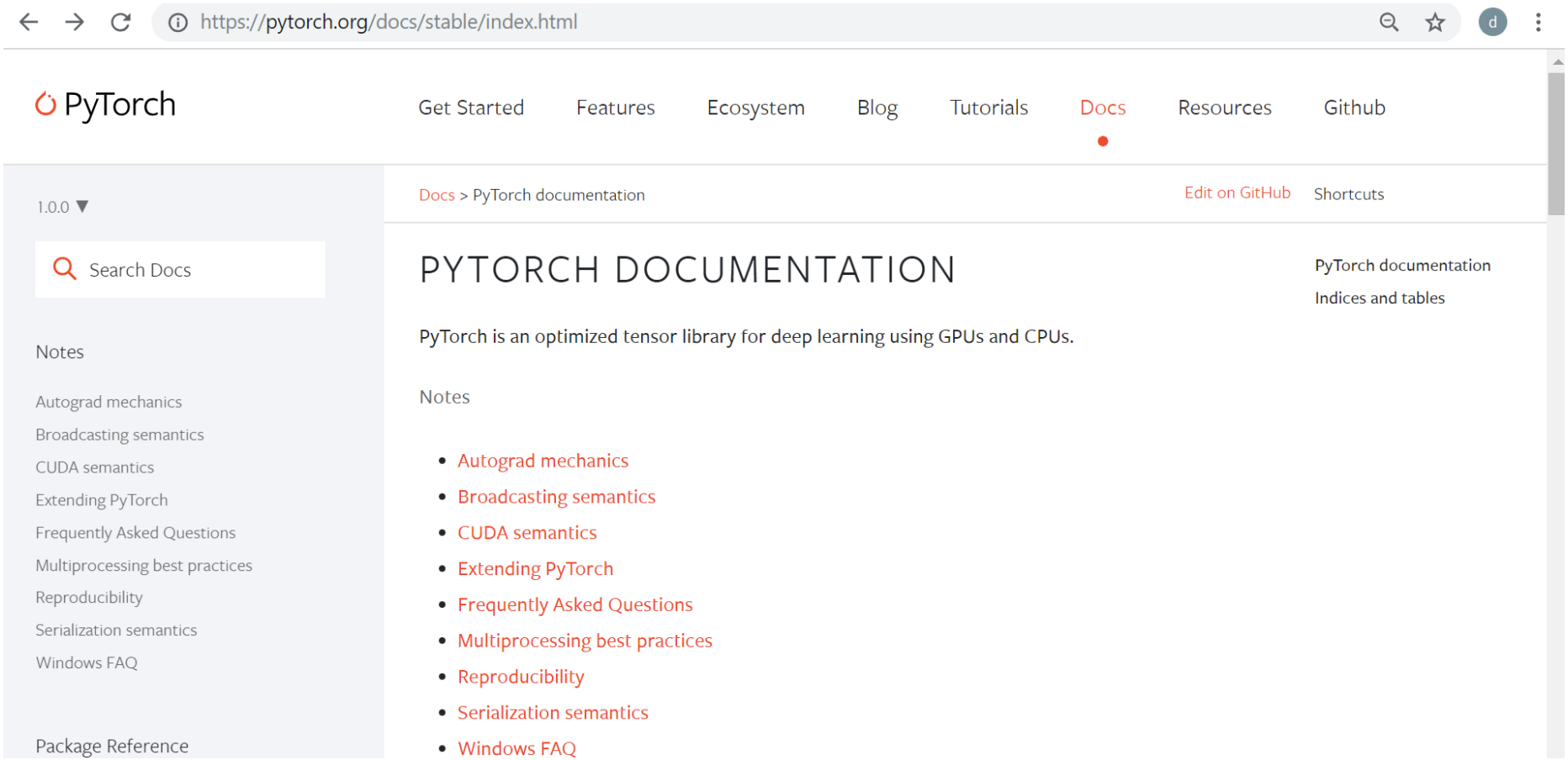
About | Downloads | Documentation | Community | Success Stories | News | Events

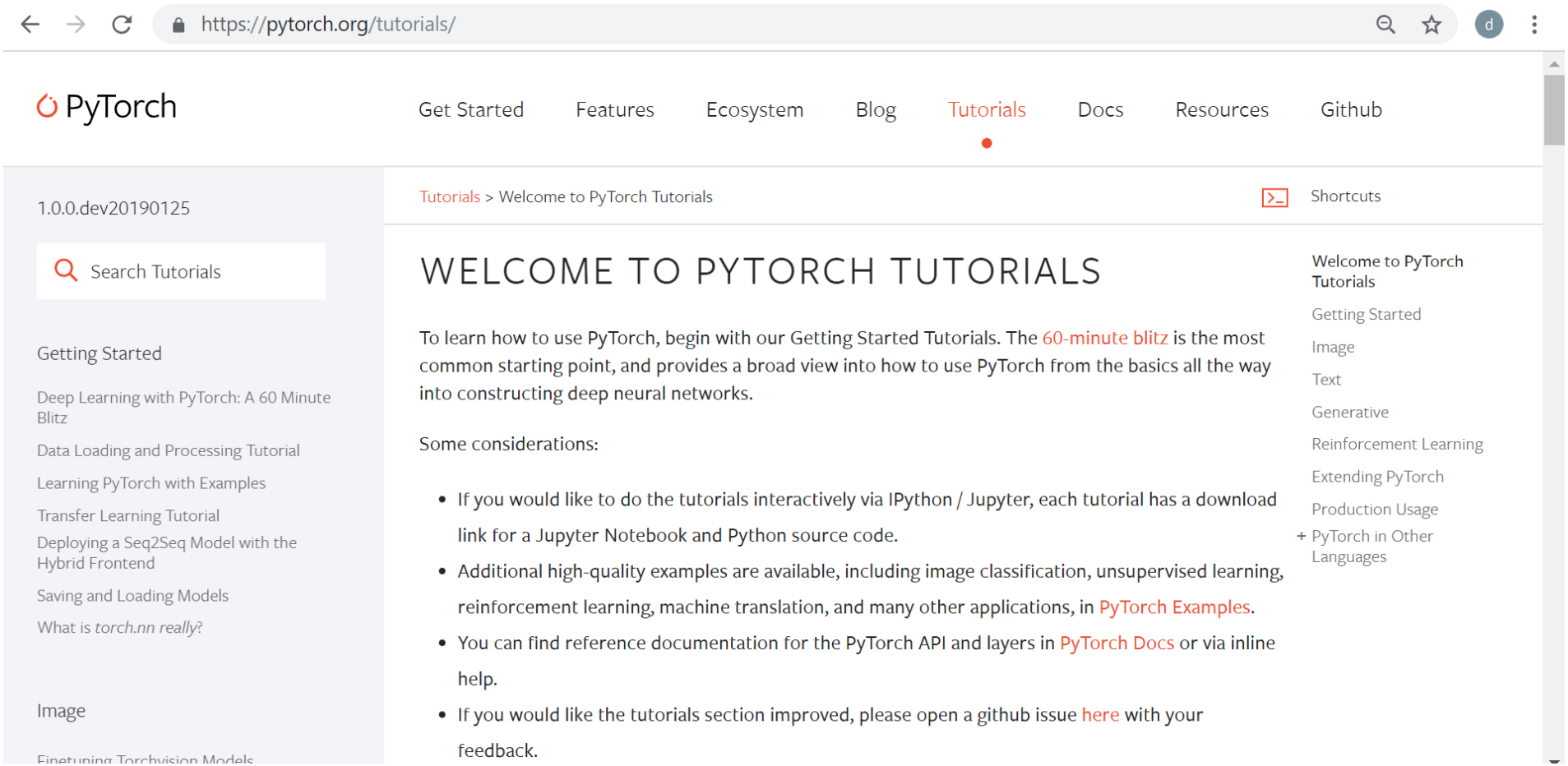
Browse the docs online or download a copy of your own.

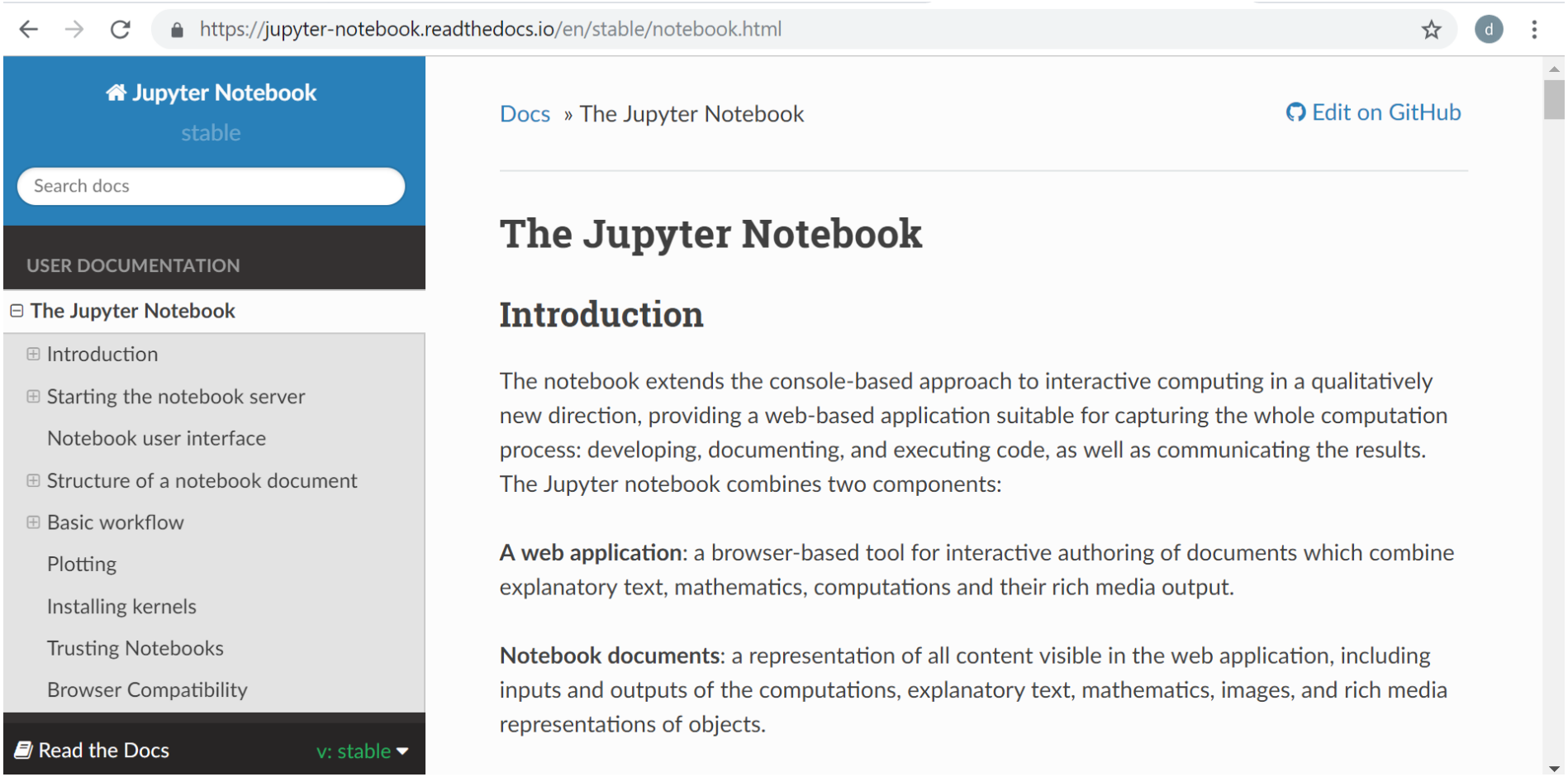
Python's documentation, tutorials, and guides are constantly evolving.
Get started here, or scroll down for documentation broken out by type and subject.

Python 3.x Docs | Python 2.x Docs

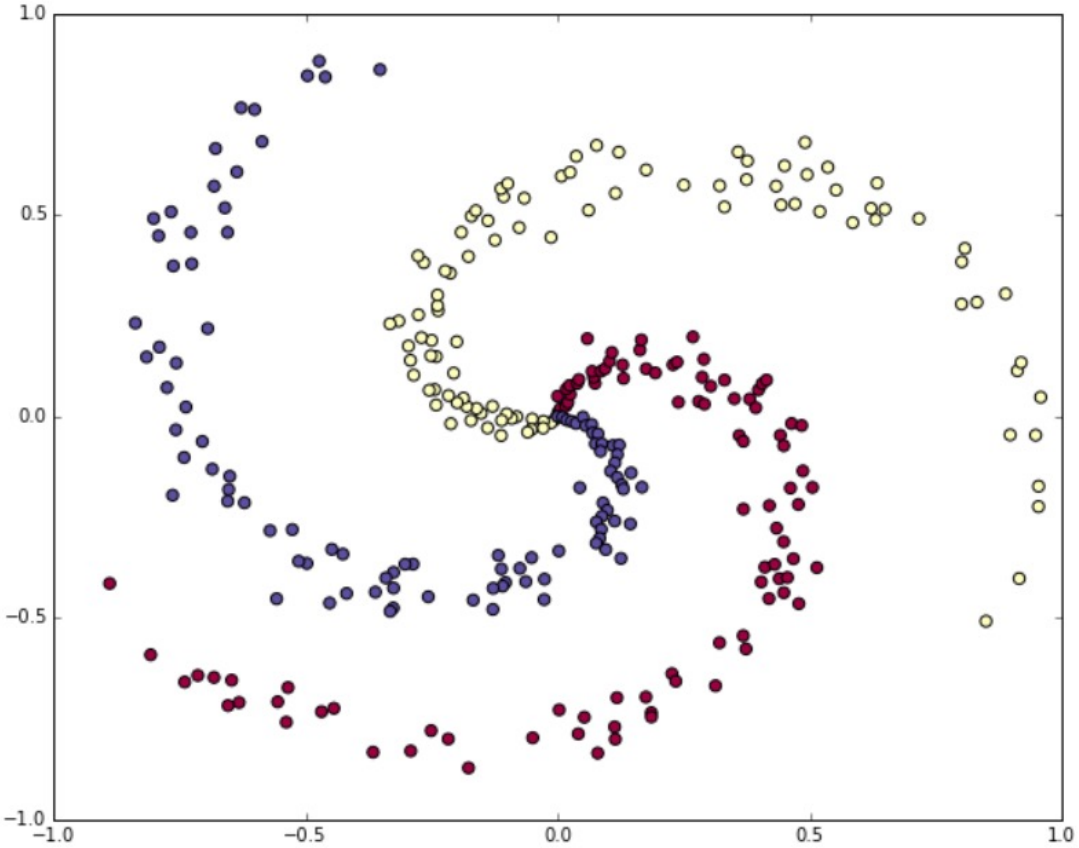
See also Documentation Releases by Version


https://pytorch.org/docs/stable/index.html
PyTorch
Get Started
Features
Ecosystem
Blog
Tutorials
Docs
Resources
Github
1.0.0 ▼
Search Docs
Notes
Autograd mechanics
Broadcasting semantics
CUDA semantics
Extending PyTorch
Frequently Asked Questions
Multiprocessing best practices
Reproducibility
Serialization semantics
Windows FAQ
Package Reference
Docs > PyTorch documentation
Edit on GitHub
Shortcuts
PYTORCH DOCUMENTATION
PyTorch documentation
Indices and tables
PyTorch is an optimized tensor library for deep learning using GPUs and CPUs.
Notes
Autograd mechanics
Broadcasting semantics
CUDA semantics
Extending PyTorch
Frequently Asked Questions
Multiprocessing best practices
Reproducibility
Serialization semantics
Windows FAQ


https://pytorch.org/tutorials/
PyTorch
Get Started
Features
Ecosystem
Blog
Tutorials
Docs
Resources
Github
1.0.0.dev20190125
Search Tutorials
Getting Started
Deep Learning with PyTorch: A 60 Minute Blitz
Data Loading and Processing Tutorial
Learning PyTorch with Examples
Transfer Learning Tutorial
Deploying a Seq2Seq Model with the Hybrid Frontend
Saving and Loading Models
What is torch.nn really?
Image
Tutorials > Welcome to PyTorch Tutorials
Shortcuts
WELCOME TO PYTORCH TUTORIALS
To learn how to use PyTorch, begin with our Getting Started Tutorials. The 60-minute blitz is the most common starting point, and provides a broad view into how to use PyTorch from the basics all the way into constructing deep neural networks.
Some considerations:
If you would like to do the tutorials interactively via IPython / Jupyter, each tutorial has a download link for a Jupyter Notebook and Python source code.
Additional high-quality examples are available, including image classification, unsupervised learning, reinforcement learning, machine translation, and many other applications, in PyTorch Examples.
You can find reference documentation for the PyTorch API and layers in PyTorch Docs or via inline help.
If you would like the tutorials section improved, please open a github issue here with your feedback.
Welcome to PyTorch Tutorials
Getting Started
Image
Text
Generative
Reinforcement Learning
Extending PyTorch
Production Usage
+ PyTorch in Other Languages


https://jupyter-notebook.readthedocs.io/en/stable/notebook.html
Jupyter Notebook
stable
Search docs
USER DOCUMENTATION
The Jupyter Notebook
Introduction
Starting the notebook server
Notebook user interface
Structure of a notebook document
Basic workflow
Plotting
Installing kernels
Trusting Notebooks
Browser Compatibility
Read the Docs
v: stable
Docs » The Jupyter Notebook
Edit on GitHub
The Jupyter Notebook
Introduction
The notebook extends the console-based approach to interactive computing in a qualitatively new direction, providing a web-based application suitable for capturing the whole computation process: developing, documenting, and executing code, as well as communicating the results. The Jupyter notebook combines two components:
A web application: a browser-based tool for interactive authoring of documents which combine explanatory text, mathematics, computations and their rich media output.
Notebook documents: a representation of all content visible in the web application, including inputs and outputs of the computations, explanatory text, mathematics, images, and rich media representations of objects.

A Toy Example using Softmax



Library Import

```
# A bit of setup
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'
```

https://docs.scipy.org/doc/numpy-1.15.0/user/quickstart.html

SciPy.org Sponsored By ENTHOUGHT

Scipy.org | Docs | NumPy v1.15 Manual | NumPy User Guide

index | next | previous

Quickstart tutorial

Prerequisites

Before reading this tutorial you should know a bit of Python. If you would like to refresh your memory, take a look at the Python tutorial.

If you wish to work the examples in this tutorial, you must also have some software installed on your computer. Please see http://scipy.org/install.html for instructions.

The Basics

NumPy's main object is the homogeneous multidimensional array. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers. In NumPy dimensions are called *axes*.

For example, the coordinates of a point in 3D space `[1, 2, 1]` has one axis. That axis has 3 elements in it, so we say it has a length of 3. In the example pictured below, the array has 2 axes. The first axis

Table Of Contents

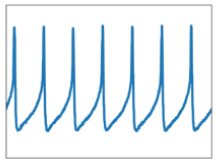
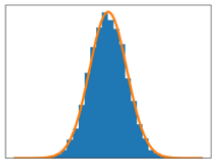
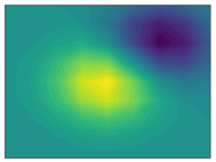
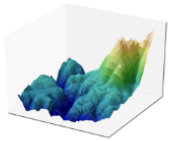
https://matplotlib.org/3.0.2/index.html

matplotlib

Version 3.0.2

Fork me on GitHub

Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shells, the Jupyter notebook, web application servers, and four graphical user interface toolkits.

Matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc., with just a few lines of code. For examples, see the sample plots and thumbnail gallery.

For simple plotting the `pyplot` module provides a MATLAB-like interface, particularly when combined with IPython. For the power user, you have full control of line styles, font properties, axes properties, etc, via an object oriented interface or via a set of

Quick search

Go

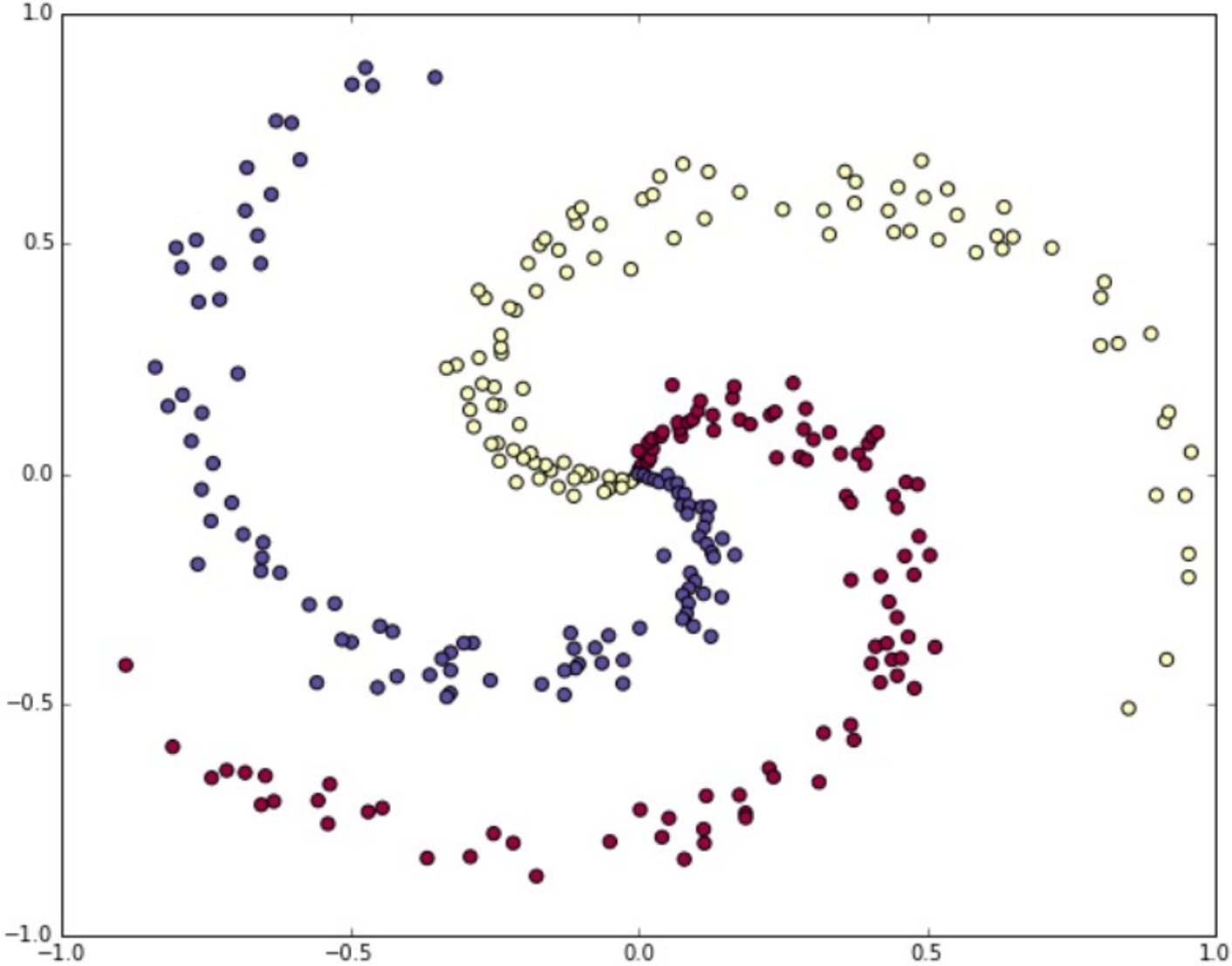
Matplotlib 3.0 is Python 3 only.

For Python 2 support, Matplotlib 2.2.x will be continued as a LTS release and updated with bugfixes until January 1, 2020.

Data Preparation and Visualization

```
np.random.seed(0)
N = 100 # number of points per class
D = 2 # dimensionality
K = 3 # number of classes
X = np.zeros((N*K,D))
y = np.zeros(N*K, dtype='uint8')
for j in xrange(K):
  ix = range(N*j,N*(j+1))
  r = np.linspace(0.0,1,N) # radius
  t = np.linspace(j*4,(j+1)*4,N) + np.random.randn(N)*0.2 # theta
  X[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
  y[ix] = j
fig = plt.figure()
plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral)
plt.xlim([-1,1])
plt.ylim([-1,1])
#fig.savefig('spiral_raw.png')
```

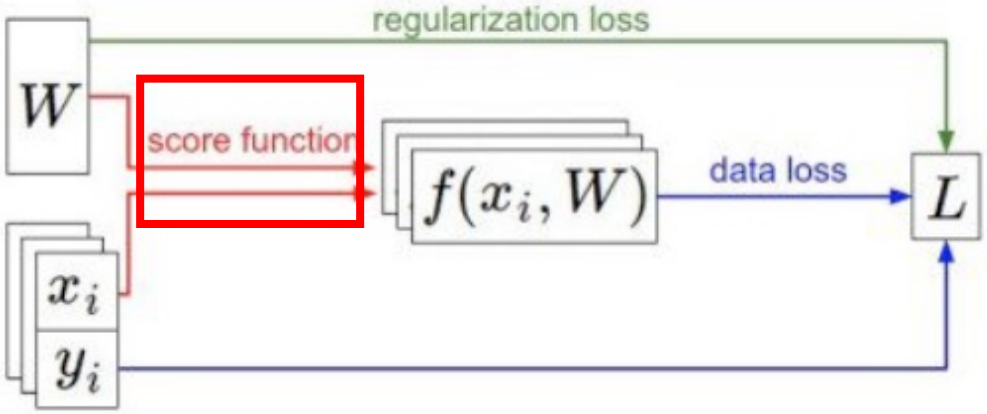
Normally we would want to preprocess the dataset so that each feature has zero mean and unit standard deviation, but in this case the features are already in a nice range from -1 to 1, so we skip this step.


1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0

Train a Softmax Classifier

```
# initialize parameters randomly
W = 0.01 * np.random.randn(D,K)
b = np.zeros((1,K))

# some hyperparameters
step_size = 1e-0
reg = 1e-3 # regularization strength

# gradient descent loop
num_examples = X.shape[0]
for i in xrange(200):
```

Within the Loop (FW)



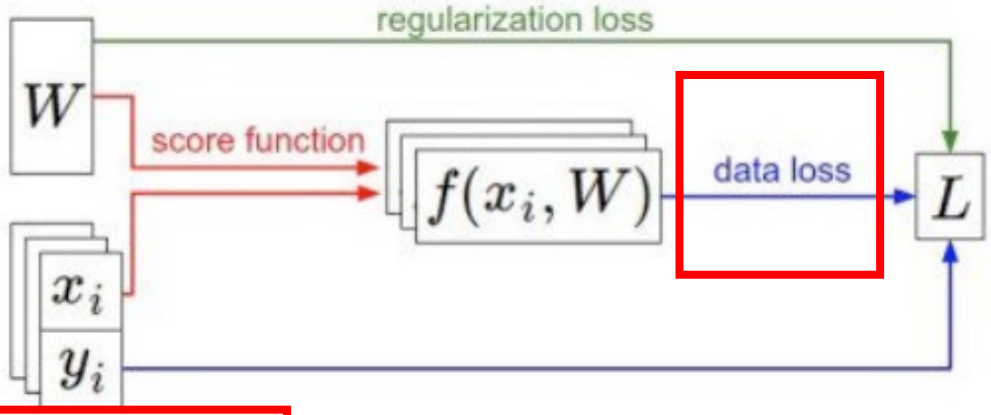

```
# evaluate class scores, [N x K]
scores = np.dot(X, W) + b

# compute the class probabilities
exp_scores = np.exp(scores)
probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True) # [N x K]

# compute the loss: average cross-entropy loss and regularization
corect_logprobs = -np.log(probs[range(num_examples),y])
data_loss = np.sum(corect_logprobs)/num_examples
reg_loss = 0.5*reg*np.sum(W*W)
loss = data_loss + reg_loss
if i % 10 == 0:
  print "iteration %d: loss %f" % (i, loss)
```

$$\mathbf{score} = \mathbf{Wx} + \mathbf{b}$$

$$L_{data} = -\frac{1}{N}\left[\sum_{i=1}^{N}\sum_{k=1}^{K} \mathbf{1}\{\boldsymbol{y_i} = \boldsymbol{k}\} log \frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right]$$

$$\boldsymbol{L_{reg}} = \sum \boldsymbol{W_i^2} \qquad L = \boldsymbol{L_{data}} + \boldsymbol{L_{reg}}$$

Within the Loop (FW)



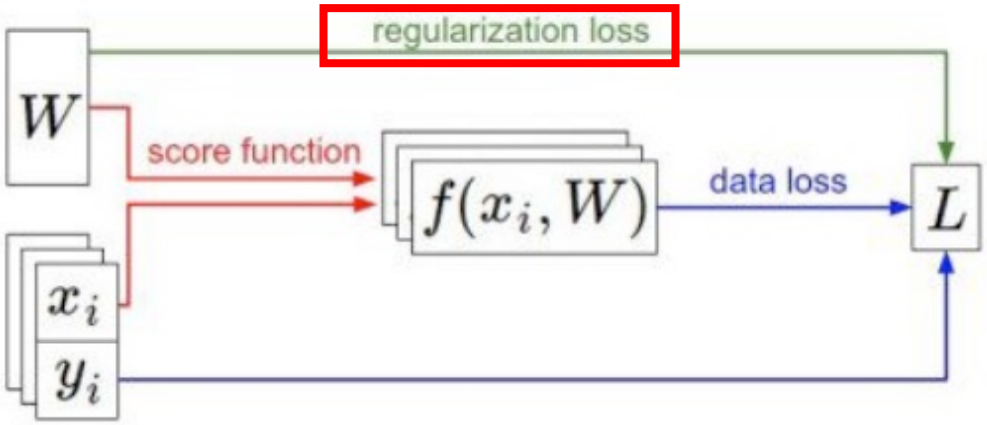

```
# evaluate class scores, [N x K]
scores = np.dot(X, W) + b

# compute the class probabilities
exp_scores = np.exp(scores)
probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True) # [N x K]

# compute the loss: average cross-entropy loss and regularization
corect_logprobs = -np.log(probs[range(num_examples),y])
data_loss = np.sum(corect_logprobs)/num_examples
reg_loss = 0.5*reg*np.sum(W*W)
loss = data_loss + reg_loss
if i % 10 == 0:
  print "iteration %d: loss %f" % (i, loss)
```

$$\mathbf{score} = \mathbf{Wx} + \mathbf{b}$$

$$L_{data} = -\frac{1}{N}\left[\sum_{i=1}^{N}\sum_{k=1}^{K} \mathbf{1}\{\boldsymbol{y_i} = \boldsymbol{k}\} log \frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right]$$

$$\boldsymbol{L_{reg}} = \sum \boldsymbol{W_i^2} \qquad L = \boldsymbol{L_{data}} + \boldsymbol{L_{reg}}$$

Within the Loop (FW)



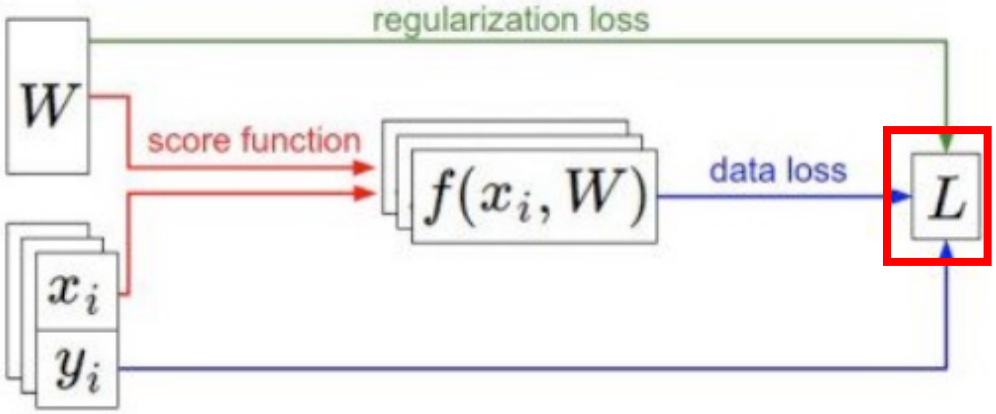

```
# evaluate class scores, [N x K]
scores = np.dot(X, W) + b

# compute the class probabilities
exp_scores = np.exp(scores)
probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True) # [N x K]

# compute the loss: average cross-entropy loss and regularization
corect_logprobs = -np.log(probs[range(num_examples),y])
data_loss = np.sum(corect_logprobs)/num_examples
reg_loss = 0.5*reg*np.sum(W*W)
loss = data_loss + reg_loss
if i % 10 == 0:
  print "iteration %d: loss %f" % (i, loss)
```

$$\mathbf{score} = \mathbf{Wx} + \mathbf{b}$$

$$L_{data} = -\frac{1}{N}\left[\sum_{i=1}^{N}\sum_{k=1}^{K} \mathbf{1}\{\boldsymbol{y_i} = \boldsymbol{k}\} log \frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right]$$

$$\boldsymbol{L_{reg}} = \sum \boldsymbol{W_i^2} \qquad L = \boldsymbol{L_{data}} + \boldsymbol{L_{reg}}$$

Within the Loop (FW)




```
# evaluate class scores, [N x K]
scores = np.dot(X, W) + b

# compute the class probabilities
exp_scores = np.exp(scores)
probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True) # [N x K]

# compute the loss: average cross-entropy loss and regularization
corect_logprobs = -np.log(probs[range(num_examples),y])
data_loss = np.sum(corect_logprobs)/num_examples
reg_loss = 0.5*reg*np.sum(W*W)
loss = data_loss + reg_loss
if i % 10 == 0:
  print "iteration %d: loss %f" % (i, loss)
```

$$\mathbf{score} = \mathbf{Wx} + \mathbf{b}$$

$$L_{data} = -\frac{1}{N}\left[\sum_{i=1}^{N}\sum_{k=1}^{K} \mathbf{1}\{\boldsymbol{y_i} = \boldsymbol{k}\} log \frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right]$$

$$\boldsymbol{L_{reg}} = \frac{\mathbf{1}}{\mathbf{2}}\sum \boldsymbol{W_i^2} \qquad L = \boldsymbol{L_{data}} + \boldsymbol{L_{reg}}$$

Within the Loop (BW)

$$p_i = \frac{e^{a_i}}{\sum_k e^{a_k}} \qquad L = -\log(p_j)$$

```
# compute the gradient on scores
dscores = probs
dscores[range(num_examples),y] -= 1
dscores /= num_examples

# backpropate the gradient to the parameters (W,b)
dW = np.dot(X.T, dscores)
db = np.sum(dscores, axis=0, keepdims=True)

dW += reg*W # regularization gradient

# perform a parameter update
W += -step_size * dW
b += -step_size * db
```

$$\frac{\partial L}{\partial a_i} = p_i - \mathbf{1}(i = j)$$

$$\frac{\partial L}{\partial p_j} = -\frac{1}{p_j}$$

$$\frac{\partial p_i}{\partial a_j} = \begin{cases} p_i(1 - p_j) & if \quad i = j \\ -p_j . p_i & if \quad i \neq j \end{cases}$$

$$p_i = \frac{e^{a_i}}{\sum_{k=1}^{N} e_k^a}$$

?

$$\frac{\partial p_i}{\partial a_j} = \frac{\partial \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}}}{\partial a_j}$$

$$f(x) = \frac{g(x)}{h(x)}$$

$$f'(x) = \frac{g\prime(x)h(x) - h\prime(x)g(x)}{h(x)^2}$$

If $i = j$,

$$\begin{aligned}
\frac{\partial \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}}}{\partial a_j} &= \frac{e^{a_i}\sum_{k=1}^{N} e^{a_k} - e^{a_j}e^{a_i}}{\left(\sum_{k=1}^{N} e^{a_k}\right)^2} \\
&= \frac{e^{a_i}\left(\sum_{k=1}^{N} e^{a_k} - e^{a_j}\right)}{\left(\sum_{k=1}^{N} e^{a_k}\right)^2} \\
&= \frac{e^{a_j}}{\sum_{k=1}^{N} e^{a_k}} \times \frac{\left(\sum_{k=1}^{N} e^{a_k} - e^{a_j}\right)}{\sum_{k=1}^{N} e^{a_k}} \\
&= p_i(1 - p_j)
\end{aligned}$$

For $i \neq j$,

$$\begin{aligned}
\frac{\partial \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}}}{\partial a_j} &= \frac{0 - e^{a_j}e^{a_i}}{\left(\sum_{k=1}^{N} e^{a_k}\right)^2} \\
&= \frac{-e^{a_j}}{\sum_{k=1}^{N} e^{a_k}} \times \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}} \\
&= -p_j . p_i
\end{aligned}$$

$$\frac{\partial p_i}{\partial a_j} = \begin{cases} p_i(1 - p_j) & if \quad i = j \\ -p_j . p_i & if \quad i \neq j \end{cases}$$

Evaluation Accuracy

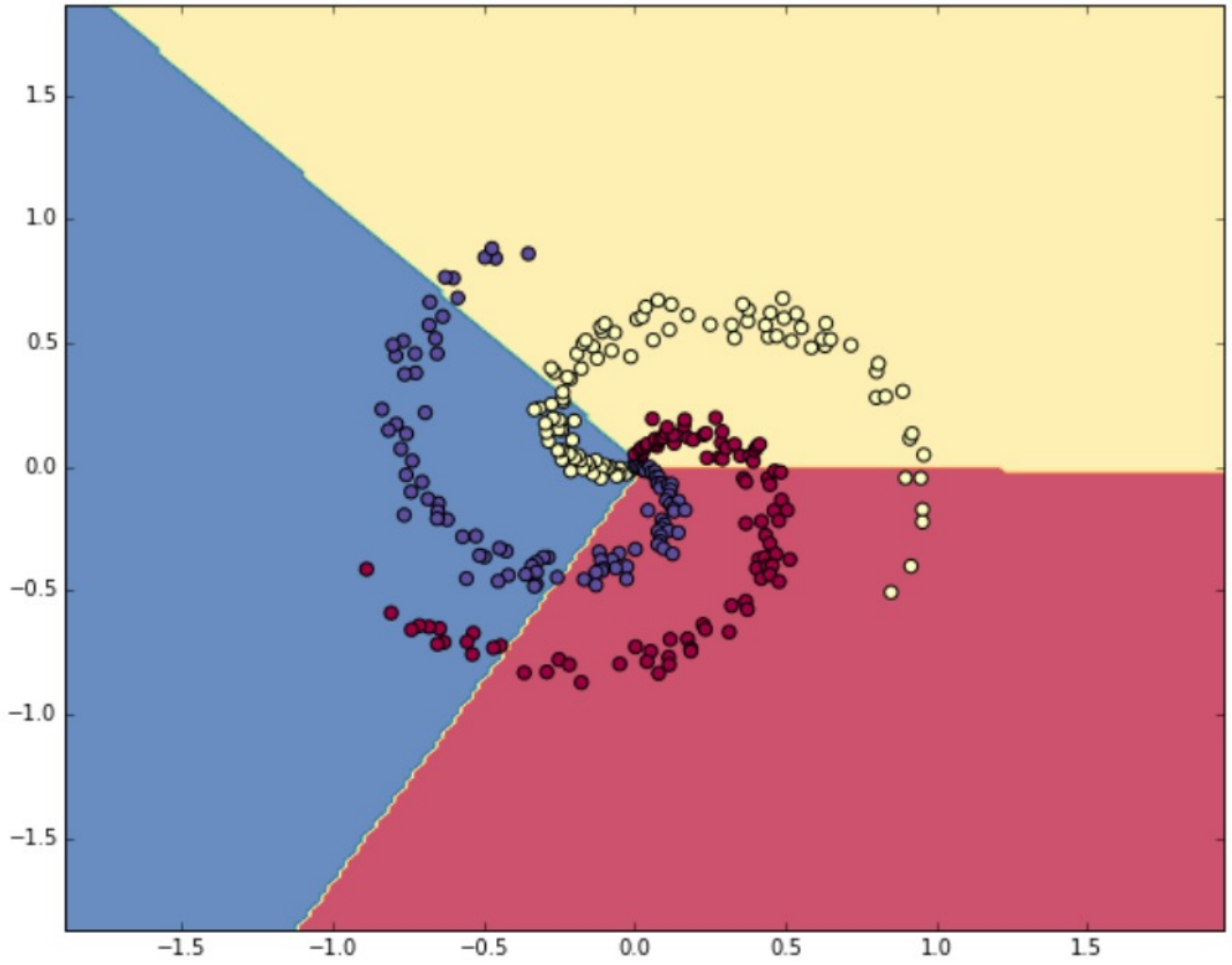
```
# evaluate training set accuracy
scores = np.dot(X, W) + b
predicted_class = np.argmax(scores, axis=1)
print 'training accuracy: %.2f' % (np.mean(predicted_class == y))
```

```
training accuracy: 0.49
```

```
iteration 0: loss 1.096956
iteration 10: loss 0.917265
iteration 20: loss 0.851503
iteration 30: loss 0.822336
iteration 40: loss 0.807586
iteration 50: loss 0.799448
iteration 60: loss 0.794681
iteration 70: loss 0.791764
iteration 80: loss 0.789920
iteration 90: loss 0.788726
iteration 100: loss 0.787938
iteration 110: loss 0.787409
iteration 120: loss 0.787049
iteration 130: loss 0.786803
iteration 140: loss 0.786633
iteration 150: loss 0.786514
iteration 160: loss 0.786431
iteration 170: loss 0.786373
iteration 180: loss 0.786331
iteration 190: loss 0.786302
```

Visualization of Decision Boundary

```
# plot the resulting classifier
h = 0.02
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))
Z = np.dot(np.c_[xx.ravel(), yy.ravel()], W) + b
Z = np.argmax(Z, axis=1)
Z = Z.reshape(xx.shape)
fig = plt.figure()
plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral, alpha=0.8)
plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral)
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
```


1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

Example using Neural Network

- Adding one hidden layer on prior example

```
# initialize parameters randomly
h = 100 # size of hidden layer
W = 0.01 * np.random.randn(D,h)
b = np.zeros((1,h))
W2 = 0.01 * np.random.randn(h,K)
b2 = np.zeros((1,K))

# some hyperparameters
step_size = 1e-0
reg = 1e-3 # regularization strength

# gradient descent loop
num_examples = X.shape[0]
for i in xrange(10000):
```

Within the Loop (FW)

```
# evaluate class scores, [N x K]
hidden_layer = np.maximum(0, np.dot(X, W) + b) # note, ReLU activation
scores = np.dot(hidden_layer, W2) + b2

# compute the class probabilities
exp_scores = np.exp(scores)
probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True) # [N x K]

# compute the loss: average cross-entropy loss and regularization
correct_logprobs = -np.log(probs[range(num_examples),y])
data_loss = np.sum(correct_logprobs)/num_examples
reg_loss = 0.5*reg*np.sum(W*W) + 0.5*reg*np.sum(W2*W2)
loss = data_loss + reg_loss
if i % 1000 == 0:
  print "iteration %d: loss %f" % (i, loss)
```

Within the Loop (BW)

```
# compute the gradient on scores
dscores = probs
dscores[range(num_examples),y] -= 1
dscores /= num_examples

# backpropate the gradient to the parameters
# first backprop into parameters W2 and b2
dW2 = np.dot(hidden_layer.T, dscores)
db2 = np.sum(dscores, axis=0, keepdims=True)
# next backprop into hidden layer
dhidden = np.dot(dscores, W2.T)
# backprop the ReLU non-linearity
dhidden[hidden_layer <= 0] = 0
# finally into W,b
dW = np.dot(X.T, dhidden)
db = np.sum(dhidden, axis=0, keepdims=True)

# add regularization gradient contribution
dW2 += reg * W2
dW += reg * W

# perform a parameter update
W += -step_size * dW
b += -step_size * db
W2 += -step_size * dW2
b2 += -step_size * db2
```

Within the Loop (BW)

```
# compute the gradient on scores
dscores = probs
dscores[range(num_examples),y] -= 1
dscores /= num_examples

# backpropate the gradient to the parameters
# first backprop into parameters W2 and b2
dW2 = np.dot(hidden_layer.T, dscores)
db2 = np.sum(dscores, axis=0, keepdims=True)
# next backprop into hidden layer
dhidden = np.dot(dscores, W2.T)
# backprop the ReLU non-linearity
dhidden[hidden_layer <= 0] = 0
# finally into W,b
dW = np.dot(X.T, dhidden)
db = np.sum(dhidden, axis=0, keepdims=True)

# add regularization gradient contribution
dW2 += reg * W2
dW += reg * W

# perform a parameter update
W += -step_size * dW
b += -step_size * db
W2 += -step_size * dW2
b2 += -step_size * db2
```

Within the Loop (BW)

```
# compute the gradient on scores
dscores = probs
dscores[range(num_examples),y] -= 1
dscores /= num_examples

# backpropate the gradient to the parameters
# first backprop into parameters W2 and b2
dW2 = np.dot(hidden_layer.T, dscores)
db2 = np.sum(dscores, axis=0, keepdims=True)
# next backprop into hidden layer
dhidden = np.dot(dscores, W2.T)
# backprop the ReLU non-linearity
dhidden[hidden_layer <= 0] = 0
# finally into W,b
dW = np.dot(X.T, dhidden)
db = np.sum(dhidden, axis=0, keepdims=True)

# add regularization gradient contribution
dW2 += reg * W2
dW += reg * W

# perform a parameter update
W += -step_size * dW
b += -step_size * db
W2 += -step_size * dW2
b2 += -step_size * db2
```

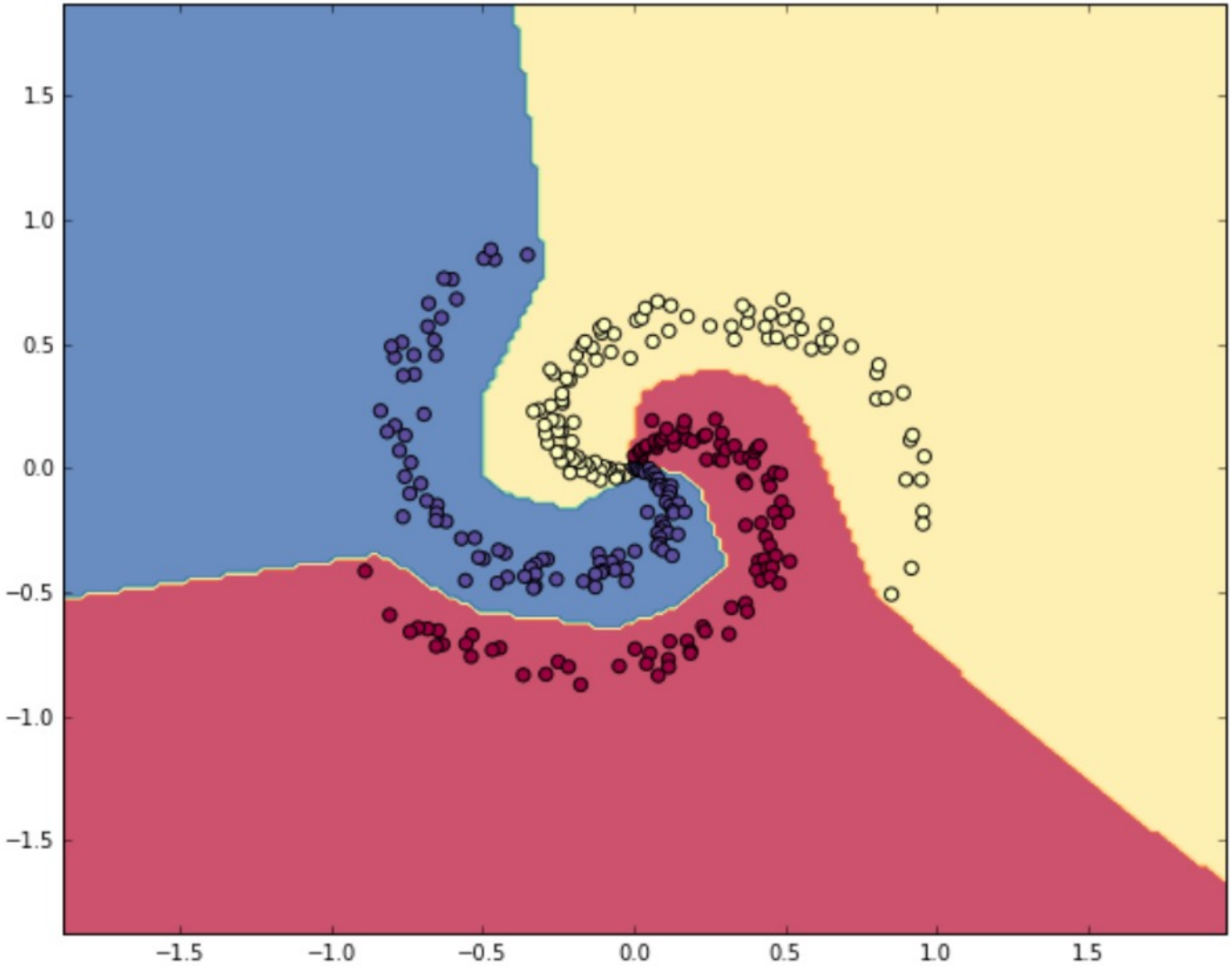
Within the Loop (BW)

```
# compute the gradient on scores
dscores = probs
dscores[range(num_examples),y] -= 1
dscores /= num_examples

# backpropate the gradient to the parameters
# first backprop into parameters W2 and b2
dW2 = np.dot(hidden_layer.T, dscores)
db2 = np.sum(dscores, axis=0, keepdims=True)
# next backprop into hidden layer
dhidden = np.dot(dscores, W2.T)
# backprop the ReLU non-linearity
dhidden[hidden_layer <= 0] = 0
# finally into W,b
dW = np.dot(X.T, dhidden)
db = np.sum(dhidden, axis=0, keepdims=True)

# add regularization gradient contribution
dW2 += reg * W2
dW += reg * W

# perform a parameter update
W += -step_size * dW
b += -step_size * db
W2 += -step_size * dW2
b2 += -step_size * db2
```

```python
# evaluate training set accuracy
hidden_layer = np.maximum(0, np.dot(X, W) + b)
scores = np.dot(hidden_layer, W2) + b2
predicted_class = np.argmax(scores, axis=1)
print 'training accuracy: %.2f' % (np.mean(predicted_class == y))
```

```
training accuracy: 0.98
```

```
iteration 0: loss 1.098744
iteration 1000: loss 0.294946
iteration 2000: loss 0.259301
iteration 3000: loss 0.248310
iteration 4000: loss 0.246170
iteration 5000: loss 0.245649
iteration 6000: loss 0.245491
iteration 7000: loss 0.245400
iteration 8000: loss 0.245335
iteration 9000: loss 0.245292
```


1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

Building 2-layer NN using Pytorch

Four types of methods

- using tensor representation

  --- manual fw, manual bw

- using auto gradient

  --- manual fw, automatic bw

- using nn module

  --- automatic fw, automatic bw

- customize the model

  --- define your own layer

Example

- One hidden layer
- Loss function is MSE (L2)

Using Tensor Representation

- Treat as tensors
- Similar to NumPy
- Can use GPU

```
# Code in file tensor/two_layer_net_tensor.py
import torch

device = torch.device('cpu')
# device = torch.device('cuda') # Uncomment this to run on GPU

# 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 input and output data
x = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)

# Randomly initialize weights
w1 = torch.randn(D_in, H, device=device)
w2 = torch.randn(H, D_out, device=device)

learning_rate = 1e-6
```

```python
for t in range(500):
  # Forward pass: compute predicted y
  h = x.mm(w1)
  h_relu = h.clamp(min=0)
  y_pred = h_relu.mm(w2)

  # Compute and print loss; loss is a scalar, and is stored in a PyTorch Tensor
  # of shape (); we can get its value as a Python number with loss.item().
  loss = (y_pred - y).pow(2).sum()
  print(t, loss.item())

  # Backprop to compute gradients of w1 and w2 with respect to loss
  grad_y_pred = 2.0 * (y_pred - y)
  grad_w2 = h_relu.t().mm(grad_y_pred)
  grad_h_relu = grad_y_pred.mm(w2.t())
  grad_h = grad_h_relu.clone()
  grad_h[h < 0] = 0
  grad_w1 = x.t().mm(grad_h)

  # Update weights using gradient descent
  w1 -= learning_rate * grad_w1
  w2 -= learning_rate * grad_w2
```

mm: matrix multiplication for batch processing

Using Autograd

- Pytorch can automatically compute gradient
- Must be differentiable

```
# Create random Tensors for weights; setting requires_grad=True means that we
# want to compute gradients for these Tensors during the backward pass.
w1 = torch.randn(D_in, H, device=device, requires_grad=True)
w2 = torch.randn(H, D_out, device=device, requires_grad=True)
```

```
y_pred = x.mm(w1).clamp(min=0).mm(w2)
```

Simplified writing

```
# Compute and print loss. Loss is a Tensor of shape (), and loss.item()
# is a Python number giving its value.
loss = (y_pred - y).pow(2).sum()
print(t, loss.item())

# Use autograd to compute the backward pass. This call will compute the
# gradient of loss with respect to all Tensors with requires_grad=True.
# After this call w1.grad and w2.grad will be Tensors holding the gradient
# of the loss with respect to w1 and w2 respectively.
loss.backward()
```

Automatic gradient calculation

```
# Update weights using gradient descent. For this step we just want to mutate
# the values of w1 and w2 in-place; we don't want to build up a computational
# graph for the update steps, so we use the torch.no_grad() context manager
# to prevent PyTorch from building a computational graph for the updates
with torch.no_grad():
  w1 -= learning_rate * w1.grad
  w2 -= learning_rate * w2.grad

  # Manually zero the gradients after running the backward pass
  w1.grad.zero_()
  w2.grad.zero_()
```

Using nn Module

```
# Use the nn package to define our model as a sequence of layers. nn.Sequential
# is a Module which contains other Modules, and applies them in sequence to
# produce its output. Each Linear Module computes output from input using a
# linear function, and holds internal Tensors for its weight and bias.
# After constructing the model we use the .to() method to move it to the
# desired device.
model = torch.nn.Sequential(
          torch.nn.Linear(D_in, H),
          torch.nn.ReLU(),
          torch.nn.Linear(H, D_out),
        ).to(device)

# The nn package also contains definitions of popular loss functions; in this
# case we will use Mean Squared Error (MSE) as our loss function. Setting
# reduction='sum' means that we are computing the *sum* of squared errors rather
# than the mean; this is for consistency with the examples above where we
# manually compute the loss, but in practice it is more common to use mean
# squared error as a loss by setting reduction='elementwise_mean'.
loss_fn = torch.nn.MSELoss(reduction='sum')
```

```python
for t in range(500):
  # Forward pass: compute predicted y by passing x to the model. Module objects
  # override the __call__ operator so you can call them like functions. When
  # doing so you pass a Tensor of input data to the Module and it produces
  # a Tensor of output data.
  y_pred = model(x)

  # Compute and print loss. We pass Tensors containing the predicted and true
  # values of y, and the loss function returns a Tensor containing the loss.
  loss = loss_fn(y_pred, y)
  print(t, loss.item())

  # Zero the gradients before running the backward pass.
  model.zero_grad()

  # Backward pass: compute gradient of the loss with respect to all the learnable
  # parameters of the model. Internally, the parameters of each Module are stored
  # in Tensors with requires_grad=True, so this call will compute gradients for
  # all learnable parameters in the model.
  loss.backward()

  # Update the weights using gradient descent. Each parameter is a Tensor, so
  # we can access its data and gradients like we did before.
  with torch.no_grad():
    for param in model.parameters():
      param.data -= learning_rate * param.grad
```

Customize Your Own Model

```
class TwoLayerNet(torch.nn.Module):
  def __init__(self, D_in, H, D_out):
    """
    In the constructor we instantiate two nn.Linear modules and assign them as
    member variables.
    """
    super(TwoLayerNet, self).__init__()
    self.linear1 = torch.nn.Linear(D_in, H)
    self.linear2 = torch.nn.Linear(H, D_out)

  def forward(self, x):
    """
    In the forward function we accept a Tensor of input data and we must return
    a Tensor of output data. We can use Modules defined in the constructor as
    well as arbitrary (differentiable) operations on Tensors.
    """
    h_relu = self.linear1(x).clamp(min=0)
    y_pred = self.linear2(h_relu)
    return y_pred
```

```
# Construct our model by instantiating the class defined above.
model = TwoLayerNet(D_in, H, D_out)

# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
loss_fn = torch.nn.MSELoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
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)
  print(t, loss.item())

  # Zero gradients, perform a backward pass, and update the weights.
  optimizer.zero_grad()
  loss.backward()
  optimizer.step()
```

Acknowledgement

Many materials of the slides of this course are adopted and re-produced from several deep learning courses and tutorials.

- Prof. Fei-fei Li, Stanford, CS231n: Convolutional Neural Networks for Visual Recognition (online available)
- Prof. Andrew Ng, Stanford, CS230: Deep learning (online available)
- Prof. Yanzhi Wang, Northeastern, EECE7390: Advance in deep learning
- Prof. Jianting Zhang, CUNY, CSc G0815 High-Performance Machine Learning: Systems and Applications
- Prof. Vivienne Sze, MIT, “Tutorial on Hardware Architectures for Deep Neural Networks”
- Pytorch official tutorial https://pytorch.org/tutorials/
- https://github.com/jcjohnson/pytorch-examples