CS-667 Advanced Machine Learning

Nazar Khan

PUCIT

Python, Automatic Differentiation and TensorFlow

Python

- A high-level, open-source, interpreted programming language.
- Highly extendable.
- Very powerful, very readable – almost like pseudocode.
- A *must-know* language for machine learning.

```
def qsort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return qsort(left) + middle + qsort(right)

print(qsort([3,6,8,10,1,2,1]))
```

Python Basics

- Basic data types
  - Integers and floats
  - Booleans
  - Strings
- Containers
  - Lists (slicing, loops, comprehensions)
  - Sets
  - Tuples
  - Dictionaries
- Functions
- Classes

These topics are covered in detail in notebooks 00 till 07 at `https://gitlab.erc.monash.edu.au/andrease/Python4Maths`

Numeric Computing

Numpy package for scientific computing. Install via `conda install numpy`.

- Arrays
- Data types
- Array math
- Broadcasting

Matplotlib package for MATLAB-like plots. Install via `conda install matplotlib`.

- Plotting, Subplots, Images

These topics are covered in detail in the tutorial at `http://cs231n.github.io/python-numpy-tutorial/`. The corresponding notebook python_numpy.ipynb can be found in the course folder on \\printsrv.

Automatic Differentiation (AD)

- Set of techniques to numerically evaluate the derivative of a function *specified by a computer program*.
- Analytic or symbolic differentiation evaluates the derivative of a function *specified by a math expression*.
- Also called *algorithmic differentiation* or *computational differentiation*.
- Backpropagation is a special case of AD.

Modern machine learning frameworks (TensorFlow, Theano, PyTorch) employ AD. The programmer only needs to implement the loss function. Derivatives are handled automatically.

Types of AD

- AD is just unrolling of the chain rule.
- Can be unrolled in two ways,
  1. Forward:

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial w_{n-1}} \frac{\partial w_{n-1}}{\partial x} = \frac{\partial y}{\partial w_{n-1}} \left(\frac{\partial w_{n-1}}{\partial w_{n-2}} \frac{\partial w_{n-2}}{\partial x} \right)$$
$$= \frac{\partial y}{\partial w_{n-1}} \left(\frac{\partial w_{n-1}}{\partial w_{n-2}} \left(\frac{\partial w_{n-2}}{\partial w_{n-3}} \frac{\partial w_{n-3}}{\partial x} \right) \right) = \cdots$$

  2. Reverse:

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial w_1} \frac{\partial w_1}{\partial x} = \left(\frac{\partial y}{\partial w_2} \frac{\partial w_2}{\partial w_1} \right) \frac{\partial w_1}{\partial x}$$
$$= \left(\left(\frac{\partial y}{\partial w_3} \frac{\partial w_3}{\partial w_2} \right) \frac{\partial w_2}{\partial w_1} \right) \frac{\partial w_1}{\partial x} = \cdots$$

- The idea is to accumulate required derivatives.

Forward Accumulation AD

- Consider the function

$$\begin{aligned} z = f(x_1, x_2) &= x_1 x_2 + \sin x_1 \\ &= w_1 w_2 + \sin w_1 \\ &= w_3 + w_4 = w_5 \end{aligned}$$

- Consider derivatives with respect to x_1. Let $\dot{w}_i = \frac{\partial w_i}{\partial x}$.
- For computing $\frac{\partial z}{\partial x_1}$, we first compute the *seed values*

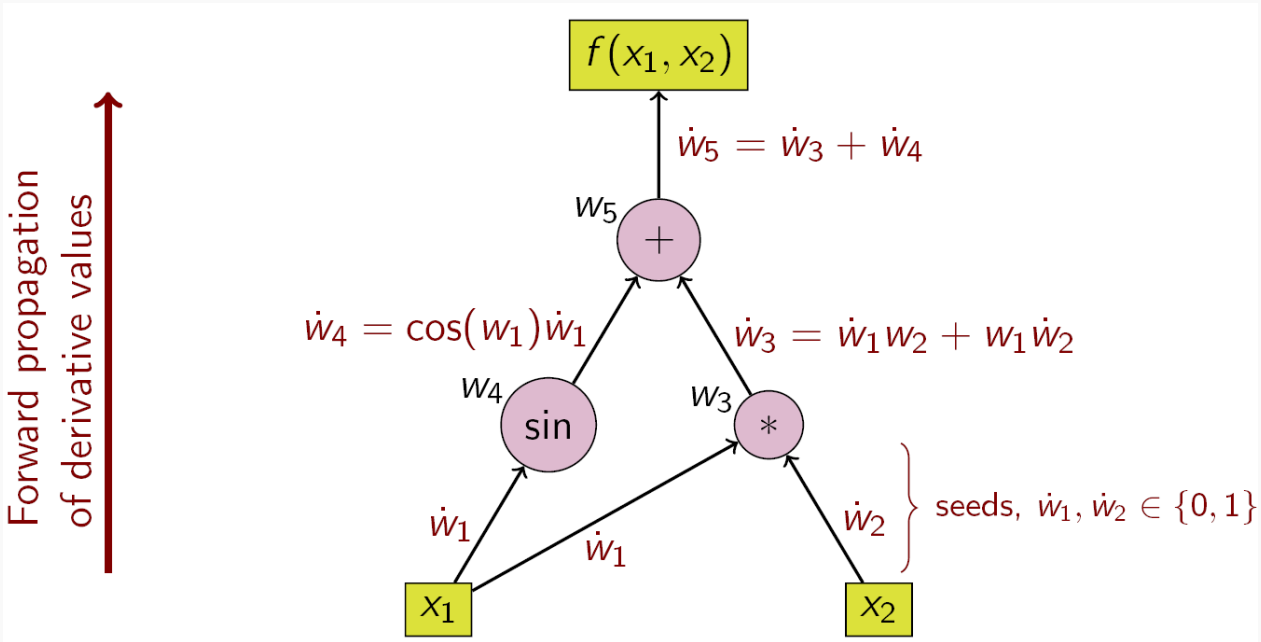
$$\dot{w}_1 = \frac{\partial x_1}{\partial x_1} = 1$$

$$\dot{w}_2 = \frac{\partial x_2}{\partial x_1} = 0$$

- These seed values can be propagated using the *chain rule*.

Forward Accumulation AD

Operations to compute value	Operations to compute derivative
$w_1 = x_1$	$\dot{w}_1 = 1$ (seed)
$w_2 = x_2$	$\dot{w}_2 = 0$ (seed)
$w_3 = w_1 \cdot w_2$	$\dot{w}_3 = w_2 \cdot \dot{w}_1 + w_1 \cdot \dot{w}_2$
$w_4 = \sin w_1$	$\dot{w}_4 = \cos w_1 \cdot \dot{w}_1$
$w_5 = w_3 + w_4$	$\dot{w}_5 = \dot{w}_3 + \dot{w}_4$



Forward Accumulation AD

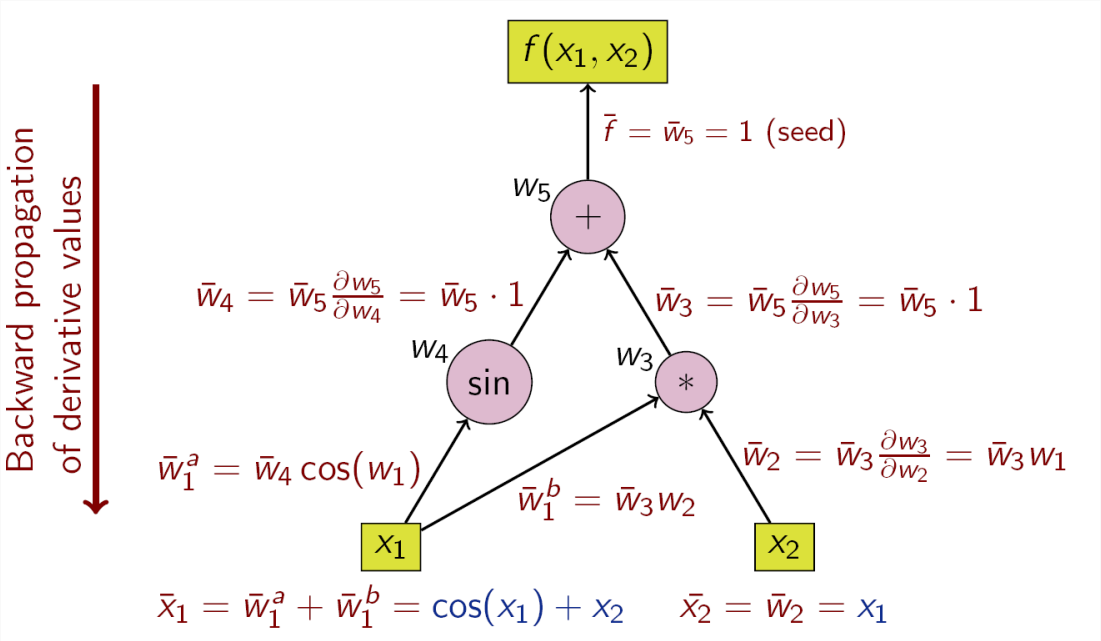
- For computing $\frac{\partial z}{\partial x_2}$, propagate *again* with seed values $\dot{w}_1 = 0$ and $\dot{w}_2 = 1$.
- Number of forward *sweeps* is equal to number of inputs.
- So forward AD is efficient when output size is much larger than input size.

Reverse Accumulation AD

- Fix the *dependent variable* and compute the derivative w.r.t each sub-expression recursively.
$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial w_1}\frac{\partial w_1}{\partial x} = \left(\frac{\partial y}{\partial w_2}\frac{\partial w_2}{\partial w_1}\right)\frac{\partial w_1}{\partial x} = \left(\left(\frac{\partial y}{\partial w_3}\frac{\partial w_3}{\partial w_2}\right)\frac{\partial w_2}{\partial w_1}\right)\frac{\partial w_1}{\partial x} = \cdots$$
- Define the *adjoint* $\bar{w}_i = \frac{\partial y}{\partial w_i}$ as the derivative w.r.t sub-expression w_i.
- Notice the similarity with $\delta_j = \frac{\partial E}{\partial a_j}$ in back-propagation.

Reverse Accumulation AD

Operations to compute value	Operations to compute derivative
$z_5 = w_5$	$\bar{w}_5 = 1$ (seed)
$w_5 = w_3 + w_4$	$\bar{w}_4 = \bar{w}_5$
$w_5 = w_3 + w_4$	$\bar{w}_3 = \bar{w}_5$
$w_3 = w_1 \cdot w_2$	$\bar{w}_2 = \bar{w}_3 \cdot w_1$
$w_4 = \sin w_1$ and $w_3 = w_1 \cdot w_2$	$\bar{w}_1 = \bar{w}_3 \cdot w_2 + \bar{w}_4 \cdot \cos w_1$



Reverse Accumulation AD

- Number of reverse *sweeps* is equal to number of outputs.
- So reverse AD is efficient when input size is much larger than output size.

AD in Python

- A Python package called *Autograd* implements reverse mode automatic differentiation.
- Elementary operations such as $+, \sin, x^k$ *etc.* are *overloaded* by also computing their derivates $1, \cos, kx$ *etc.*.
- If required, user-defined complex functions and their derivative implementations can be *registered* with Autograd.

Logistic Regression via Automatic Differentiation

```
import pylab
import sklearn.datasets
import autograd.numpy as np
from autograd import grad

# Generate the data
train_X, train_y = sklearn.datasets.make_moons(500, noise=0.1)

# Define the activation, prediction and loss functions for Logistic Regression
def activation(x):
    return 0.5*(np.tanh(x) + 1)

def predict(weights, inputs):
    return activation(np.dot(inputs, weights))

def loss(weights):
    preds = predict(weights, train_X)
    label_probabilities = np.log(preds) * train_y + np.log(1 - preds) * (1 - train_y)
    return -np.sum(label_probabilities)

# Compute the gradient of the loss function
gradient_loss = grad(loss)

# Set the initial weights
weights = np.array([1.0, 1.0])

# Steepest Descent
loss_values = []
learning_rate = 0.001
```

Logistic Regression via Automatic Differentiation

```
for i in range(100):
    loss_values.append(loss(weights))
    step = gradient_loss(weights)
    weights -= step * learning_rate

# Plot the decision boundary
x_min, x_max = train_X[:, 0].min() - 0.5, train_X[:, 0].max() + 0.5
y_min, y_max = train_X[:, 1].min() - 0.5, train_X[:, 1].max() + 0.5
x_mesh, y_mesh = np.meshgrid(np.arange(x_min, x_max, 0.01), np.arange(y_min, y_max, 0.01))
Z = predict(weights, np.c_[x_mesh.ravel(), y_mesh.ravel()])
Z = Z.reshape(x_mesh.shape)
cs = pylab.contourf(x_mesh, y_mesh, Z, cmap=pylab.cm.Spectral)
pylab.scatter(train_X[:, 0], train_X[:, 1], c=train_y, cmap=pylab.cm.Spectral)
pylab.colorbar(cs)

# Plot the loss over each step
pylab.figure()
pylab.plot(loss_values)
pylab.xlabel("Steps")
pylab.ylabel("Loss")
pylab.show()
```

TensorFlow

- Open source library for numerical computation using data flow graphs.
- Graph nodes represent mathematical operations and edges represent multidimensional data arrays (tensors) that flow between them.
- Deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device without rewriting code.
- Install via *conda install tensorflow*
- Datasets easily accessible through TF `https://github.com/tensorflow/datasets/blob/master/docs/datasets.md`
- Extremely easy high-level API of TensorFlow called Keras[1].
- Useful to understand the low-level APIs[2] as well.

[1] `https://www.tensorflow.org/guide/keras`
[2] `https://www.tensorflow.org/guide/low_level_intro`

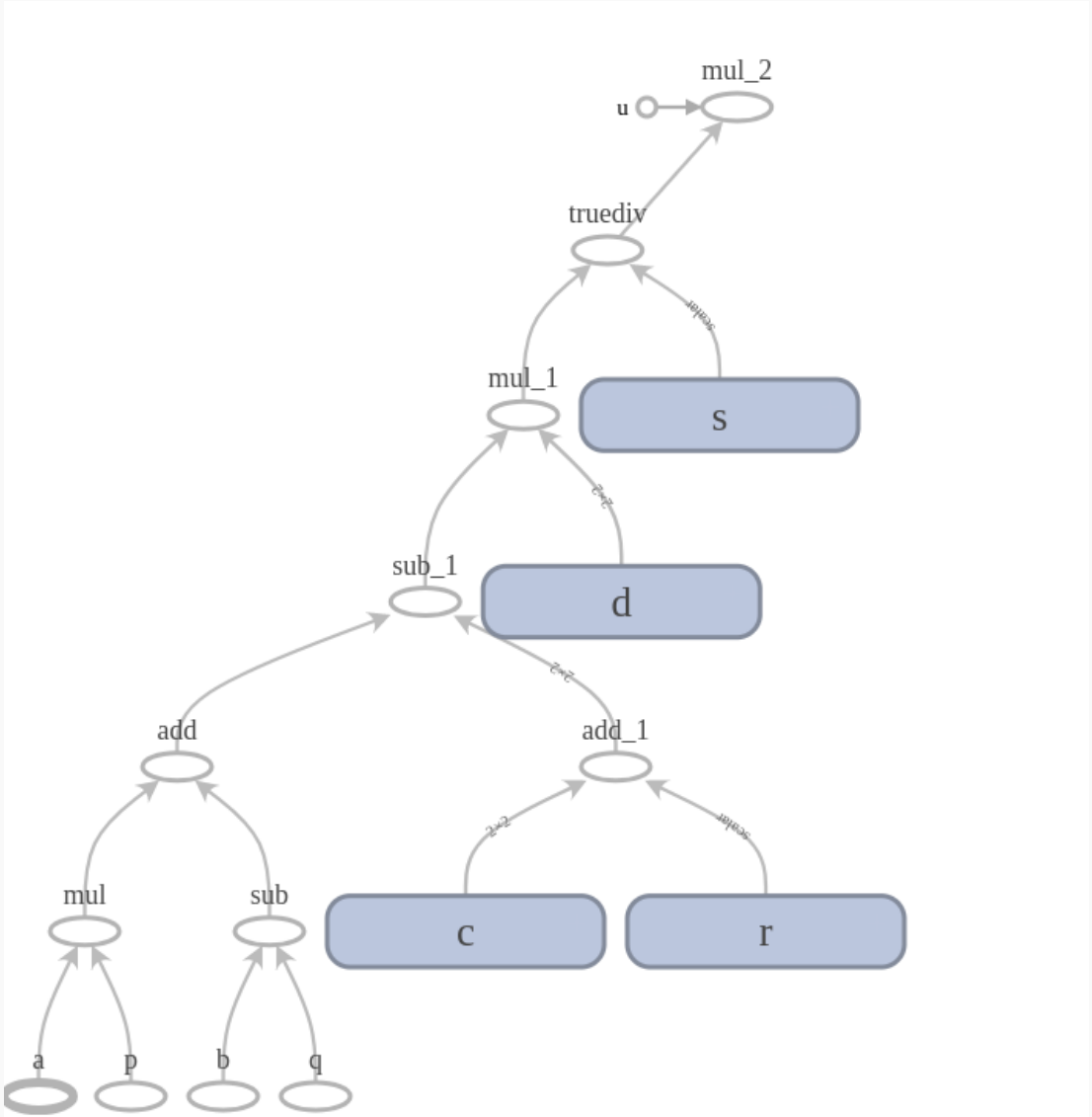
Keras

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(512, activation=tf.nn.relu),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

Computation Graph, Session, Run

```
import numpy as np
import tensorflow as tf

a = tf.placeholder(tf.float32, name='a') #placeholder contains fixed value set before running the graph
b = tf.placeholder(tf.float32, name='b')
c = tf.Variable(initial_value=[[5.0,5.0],[5.0,5.0]], name='c') #variable can change its value
d = tf.Variable(initial_value=[[3.0,3.0],[3.0,3.0]], name='d')
p = tf.placeholder(tf.float32, name='p')
q = tf.placeholder(tf.float32, name='q')
r = tf.Variable(initial_value=3.0, name='r')
s = tf.Variable(initial_value=4.0, name='s')
u = tf.constant(5.0, name='u') #constant value is same for all runs of the graph

e = (((a * p) + (b - q) - (c + r )) * d/s) * u #the computation graph

#setup a session
sess = tf.Session()
writer = tf.summary.FileWriter("./temp") #for TensorBoard
writer.add_graph(sess.graph)

#run the session
sess.run(tf.global_variables_initializer())
result = sess.run(e,feed_dict={p:1.0, q:2.0, a:[[1,1],[1,1]],b:[[2,2],[2,2]]}) #feed the placeholders
print("Result: ", result)
```

Computation Graph, Session, Run



TensorBoard

- Collection of web applications for inspecting and understanding runs and graphs in TF.
- Useful for understanding extremely complex computations such as deep networks and debugging and optimizing them.
- Currently supports five visualizations:
  1. scalars
  2. images
  3. audio
  4. histograms
  5. graphs
- Run your code via `python tf_comp_graph.py`
- Start TensorBoard service via `tensorboard -logdir temp`.
- Open `localhost:6006` in your browser.
- Note that temp/ should only contain 1 summary.

Machine Learning in TensorFlow

- Linear regression
- Logistic regression
- Neural Network
- CNN

Two-Layer Neural Network in TensorFlow

```
import numpy as np
import tensorflow as tf

# Generate Random Data
examples = 1000
features = 100
x_data = np.random.randn(examples, features)
y_data = np.random.randn(examples,1)

# Define the Neural Network Model
hidden_layer_nodes = 10
X = tf.placeholder(tf.float32, shape=[None, features], name = "X")
y = tf.placeholder(tf.float32, shape=[None, 1], name = "y")
w1 = tf.Variable(tf.random_normal(shape=[features,hidden_layer_nodes]), name="w1")
b1 = tf.Variable(tf.random_normal(shape=[hidden_layer_nodes]), name="b1")
w2 = tf.Variable(tf.random_normal(shape=[hidden_layer_nodes,1]), name="w2")
b2 = tf.Variable(tf.random_normal(shape=[1,1]), name="b2")
hidden_output = tf.nn.relu(tf.add(tf.matmul(X, w1), b1), name="hidden_output")
y_hat = tf.nn.relu(tf.add(tf.matmul(hidden_output, w2), b2), name="y_hat")
loss = tf.reduce_mean(tf.square(y_hat - y), name="loss")

# Set up the gradient descent
learning_rate = 0.05
optimiser = tf.train.GradientDescentOptimizer(learning_rate)
train_step = optimiser.minimize(loss)

sess = tf.Session()
writer = tf.summary.FileWriter("./temp")
writer.add_graph(sess.graph)
```

Two-Layer Neural Network in TensorFlow

```
sess.run(tf.global_variables_initializer())

epochs = 5000
batch_size = 5

# Before Training
curr_loss = sess.run(loss, feed_dict={X:x_data, y:y_data})
print("Loss before training:", curr_loss)

for i in range(epochs):
    rand_index = np.random.choice(examples, size=batch_size)
    sess.run(train_step, feed_dict={X:x_data[rand_index], y:y_data[rand_index]})

# After Training
curr_loss = sess.run(loss, feed_dict={X:x_data, y:y_data})
print("Loss before training:", curr_loss)

# Loss before training: 42.431
# Loss before training: 0.976375
```