Conditions and loops

Kang Ik Kevin Cho

Conditions ¶

IfElse vs Switch

- Both ops build a condition over symbolic variables.
- `IfElse` takes a *boolean* condition and two variables as inputs.
- `Switch` takes a *tensor* as condition and two variables as inputs. `switch` is an elementwise operation and is thus more general than `ifelse`.
- Whereas `switch` evaluates both *output* variables, `ifelse` is lazy and only evaluates one variable with respect to the condition.

theano

Table Of Contents

Previous topic

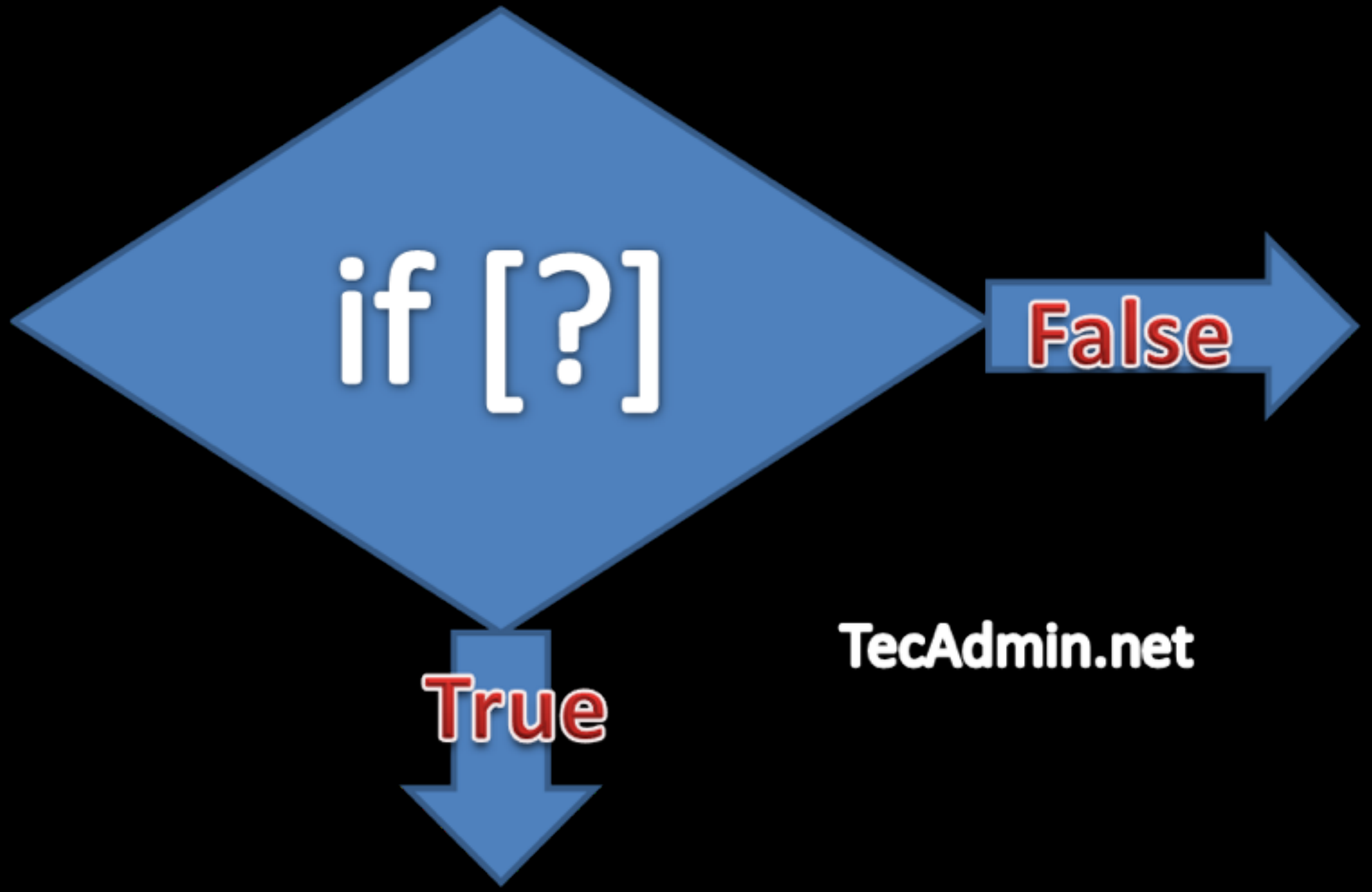
Loading and Saving

Next topic

All from theano documentation

Conditions

- IfElse
  - 둘 중 하나만 실행
- Switch
  - 모두 실행 후 선택



```python
from theano import tensor as T
from theano.ifelse import ifelse
import theano, time, numpy

a,b = T.scalars('a', 'b')
x,y = T.matrices('x', 'y')
```

```
a,b = T.scalars('a', 'b')
x,y = T.matrices('x', 'y')
```

a less than b

```
z_switch = T.switch(T.lt(a, b), T.mean(x), T.mean(y))
z_lazy = ifelse(T.lt(a, b), T.mean(x), T.mean(y))
```

```
a,b = T.scalars('a', 'b')
x,y = T.matrices('x', 'y')
```

```
f_switch = theano.function([a, b, x, y], z_switch,
                           mode=theano.Mode(linker='vm'))
f_lazyifelse = theano.function([a, b, x, y], z_lazy,
                               mode=theano.Mode(linker='vm'))
```

```python
val1 = 0.
val2 = 1.
big_mat1 = numpy.ones((10000, 1000))
big_mat2 = numpy.ones((10000, 1000))

n_times = 10
```

```python
tic = time.clock()
for i in xrange(n_times):
    f_switch(val1, val2, big_mat1, big_mat2)
print 'time spent evaluating both values %f sec' % (time.clock() - tic)

tic = time.clock()
for i in xrange(n_times):
    f_lazyifelse(val1, val2, big_mat1, big_mat2)
print 'time spent evaluating one value %f sec' % (time.clock() - tic)
```

Switch : 두개 다돌림

```
$ python ifelse_switch.py
time spent evaluating both values 0.6700 sec
time spent evaluating one value 0.3500 sec
```

IfElse : 하나만 돌림

Loop

- *Scan*
  - like a for loop
  - advantages
    - minimizes GPU transfers
    - faster than python for loop
    - takes memory into account
  - types
    - Reduction
    - map

```python
result = 1
for i in xrange(k):
  result = result * A
```

1. the initial value
2. accumulation
3. unchanging variables

```python
result = 1
for i in xrange(k):
  result = result * A
```

1.	the initial value	outputs_info
2.	accumulation	automatic
3.	unchanging variables	non_sequences

The equivalent Theano code

```
k = T.iscalar("k")
A = T.vector("A")
```

The equivalent Theano code

function

```
# Symbolic description of the result
result, updates = theano.scan(fn=lambda prior_result, A: prior_result * A,
                              outputs_info=T.ones_like(A),
                              non_sequences=A,
                              n_steps=k)
```

initial value

unchanging variable

The equivalent Theano code

```
# We only care about A**k, but scan has provided us with A**1 through A**k.
# Discard the values that we don't care about. Scan is smart enough to
# notice this and not waste memory saving them.
final_result = result[-1]

# compiled function that returns A**k
power = theano.function(inputs=[A,k], outputs=final_result, updates=updates)

print power(range(10),2)
print power(range(10),4)
```

for x in a_list 처럼

```
coefficients = theano.tensor.vector("coefficients")
x = T.scalar("x")

max_coefficients_supported = 10000
```

for x in a_list 처럼

indicates that it doesn't need to pass the prior result to fn

```
# Generate the components of the polynomial
components, updates = theano.scan(fn=lambda coefficient, power, free_variable: coefficient * (free_variable ** power),
                                  outputs_info=None,
                                  sequences=[coefficients, theano.tensor.arange(max_coefficients_supported)],
                                  non_sequences=x)
# Sum them up
polynomial = components.sum()

# Compile a function
calculate_polynomial = theano.function(inputs=[coefficients, x], outputs=polynomial)

# Test
test_coefficients = numpy.asarray([1, 0, 2], dtype=numpy.float32)
test_value = 3
```

does not change

power - like python's enumerate

```
print calculate_polynomial(test_coefficients, test_value)
print 1.0 * (3 ** 0) + 0.0 * (3 ** 1) + 2.0 * (3 ** 2)
```

Simple accumulation into a scalar, ditching lambda

```
up_to = T.iscalar("up_to")

# define a named function, rather than using lambda
def accumulate_by_adding(arange_val, sum_to_date):
    return sum_to_date + arange_val
seq = T.arange(up_to)
```

Simple accumulation into a scalar, ditching lambda

```
outputs_info = T.as_tensor_variable(np.asarray(0, seq.dtype))
scan_result, scan_updates = theano.scan(fn=accumulate_by_adding,
                                        outputs_info=outputs_info,
                                        sequences=seq)
triangular_sequence = theano.function(inputs=[up_to], outputs=scan_result)
```

```
# test
some_num = 15
print(triangular_sequence(some_num))
```

Another simple example

```
location = T.imatrix("location")
values = T.vector("values")
output_model = T.matrix("output_model")

def set_value_at_position(a_location, a_value, output_model):
    zeros = T.zeros_like(output_model)
    zeros_subtensor = zeros[a_location[0], a_location[1]]
    return T.set_subtensor(zeros_subtensor, a_value)
```

Another simple example

```
result, updates = theano.scan(fn=set_value_at_position,
                              outputs_info=None,
                              sequences=[location, values],
                              non_sequences=output_model)
```

```
assign_values_at_positions = theano.function(inputs=[location, values, output_model], outputs=result)
```

Another simple example

```python
# test
test_locations = numpy.asarray([[1, 1], [2, 3]], dtype=numpy.int32)
test_values = numpy.asarray([42, 50], dtype=numpy.float32)
test_output_model = numpy.zeros((5, 5), dtype=numpy.float32)
print(assign_values_at_positions(test_locations, test_values, test_output_model))
```

Using shared variables - Gibbs sampling

```
W = theano.shared(W_values) # we assume that ``W_values`` contains the
                            # initial values of your weight matrix

bvis = theano.shared(bvis_values)
bhid = theano.shared(bhid_values)

trng = T.shared_randomstreams.RandomStreams(1234)
```

Using shared variables - Gibbs sampling

```
def OneStep(vsample) :
    hmean = T.nnet.sigmoid(theano.dot(vsample, W) + bhid)
    hsample = trng.binomial(size=hmean.shape, n=1, p=hmean)
    vmean = T.nnet.sigmoid(theano.dot(hsample, W.T) + bvis)
    return trng.binomial(size=vsample.shape, n=1, p=vmean,
                         dtype=theano.config.floatX)

sample = theano.tensor.vector()
```

```
values, updates = theano.scan(OneStep, outputs_info=sample, n_steps=10)

gibbs10 = theano.function([sample], values[-1], updates=updates)
```

link shared variables and their updated value

Simple example about the updates

```
a = theano.shared(1)
values, updates = theano.scan(lambda: {a: a+1}, n_steps=10)
```

```
b = a + 1
c = updates[a] + 1
f = theano.function([], [b, c], updates=updates)

print(b)
print(c)
print(a.get_value())
```

Shared variables into non_sequences

```python
W = theano.shared(W_values) # we assume that ``W_values`` contains the
                            # initial values of your weight matrix

bvis = theano.shared(bvis_values)
bhid = theano.shared(bhid_values)

trng = T.shared_randomstreams.RandomStreams(1234)

# OneStep, with explicit use of the shared variables (W, bvis, bhid)
def OneStep(vsample, W, bvis, bhid):
    hmean = T.nnet.sigmoid(theano.dot(vsample, W) + bhid)
    hsample = trng.binomial(size=hmean.shape, n=1, p=hmean)
    vmean = T.nnet.sigmoid(theano.dot(hsample, W.T) + bvis)
    return trng.binomial(size=vsample.shape, n=1, p=vmean,
                         dtype=theano.config.floatX)

sample = theano.tensor.vector()

# The new scan, with the shared variables passed as non_sequences
values, updates = theano.scan(fn=OneStep,
                              outputs_info=sample,
                              non_sequences=[W, bvis, bhid],
                              n_steps=10)

gibbs10 = theano.function([sample], values[-1], updates=updates)
```

Shared variables into non_sequences

```
# The new scan, using strict=True
values, updates = theano.scan(fn=OneStep,
                              outputs_info=sample,
                              non_sequences=[W, bvis, bhid],
                              n_steps=10,
                              strict=True)
```

“but also looking back more than one step.”

Theano scan

Recurrent Neural Network with Scan

$$x(n) = \tanh(Wx(n-1) + W_1^{in}u(n) + W_2^{in}u(n-4) + W^{feedback}y(n-1))$$
$$y(n) = W^{out}x(n-3)$$

```python
def oneStep(u_tm4, u_t, x_tm3, x_tm1, y_tm1, W, W_in_1, W_in_2,  W_feedback, W_out):

  x_t = T.tanh(theano.dot(x_tm1, W) + \
               theano.dot(u_t,   W_in_1) + \
               theano.dot(u_tm4, W_in_2) + \
               theano.dot(y_tm1, W_feedback))
  y_t = theano.dot(x_tm3, W_out)

  return [x_t, y_t]
```

Recurrent Neural Network with Scan

```
W = T.matrix()
W_in_1 = T.matrix()
W_in_2 = T.matrix()
W_feedback = T.matrix()
W_out = T.matrix()

u = T.matrix() # it is a sequence of vectors
x0 = T.matrix() # initial state of x has to be a matrix, since
                # it has to cover x[-3]
y0 = T.vector() # y0 is just a vector since scan has only to provide
                # y[-1]
```

Recurrent Neural Network with Scan

```
([x_vals, y_vals], updates) = theano.scan(fn=oneStep,
                                          sequences=dict(input=u, taps=[-4,-0]),
                                          outputs_info=[dict(initial=x0, taps=[-3,-1]), y0],
                                          non_sequences=[W, W_in_1, W_in_2, W_feedback, W_out],
                                          strict=True)
     # for second input y, scan adds -1 in output_taps by default
```

Conditional ending of Scan

```
def power_of_2(previous_power, max_value):
    return previous_power*2, theano.scan_module.until(previous_power*2 > max_value)

max_value = T.scalar()
values, _ = theano.scan(power_of_2,
                        outputs_info = T.constant(1.),
                        non_sequences = max_value,
                        n_steps = 1024)

f = theano.function([max_value], values)

print f(45)
```

Optimizing Scan’s performance

- Minimizing Scan usage
- Explicitly passing inputs of the inner function to scan
- Deactivating garbage collecting in Scan
- Graph optimizations