Neural Nets &
Theano Python Library

1: Neural Nets

Neural Nets

- Linear combination of features

  $b + w_1 * x_1 + w_2 * x_2 + \ldots + w_p * x_p$
- Non-linear activation (e.g. **wx** -> [-1, 1])

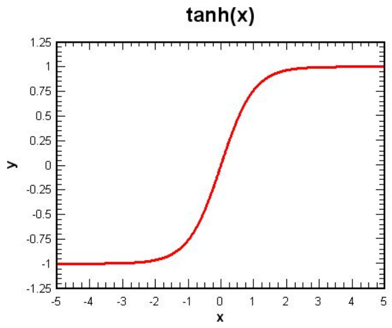
  $f(b + w_1 * x_1 + w_2 * x_2 + \ldots + w_p * x_p)$
- Pass outputs on as inputs (multi-layer nets)
  - This gives non-linear decision boundaries
  - This gives a non-convex problem :(

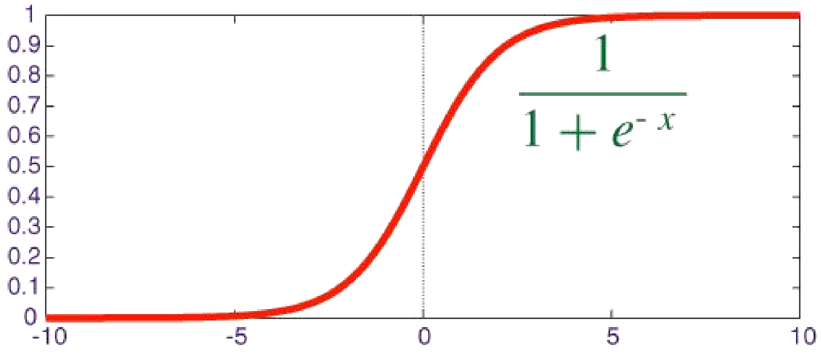
Neural Nets

- Linear combination of features

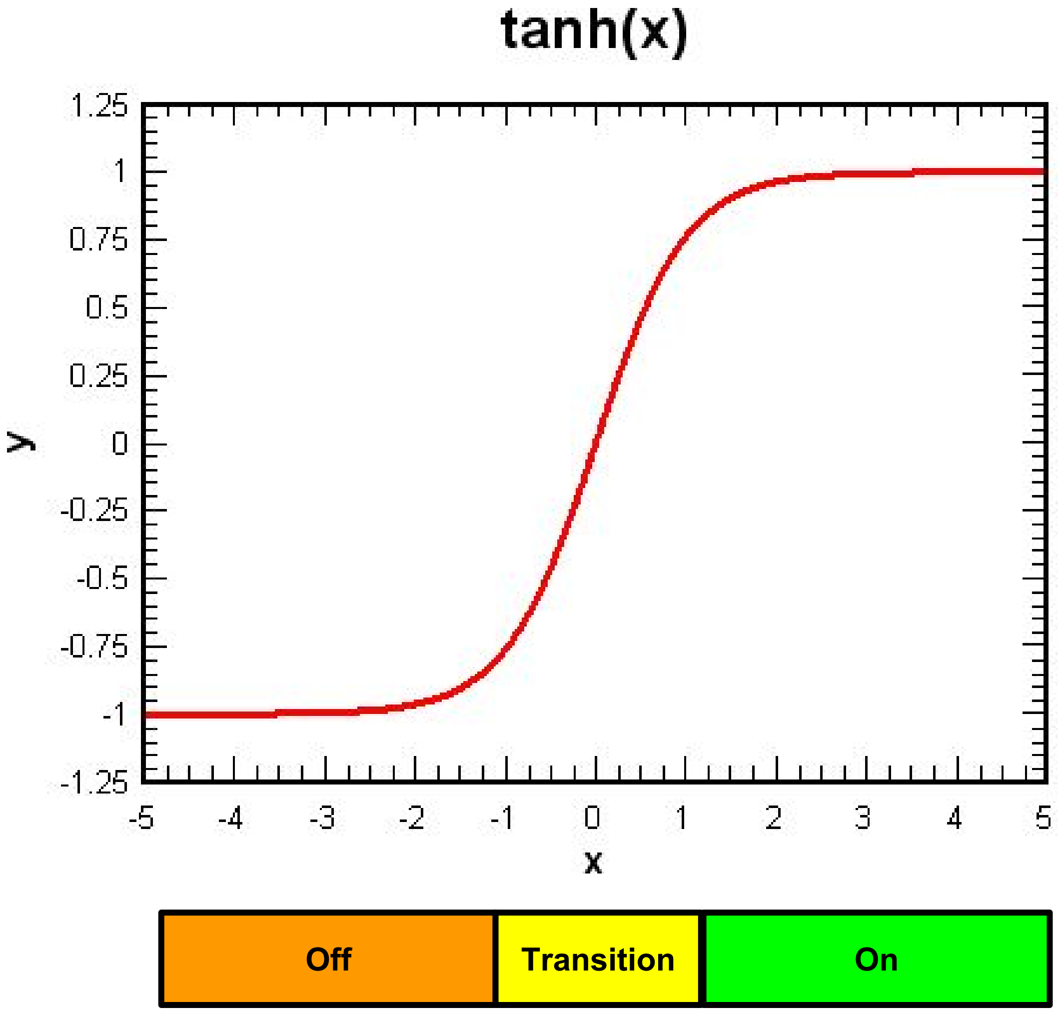
  b + w_1 * x_1 + w_2 * x_2 + … + w_p * x_p

- Non-linear activation

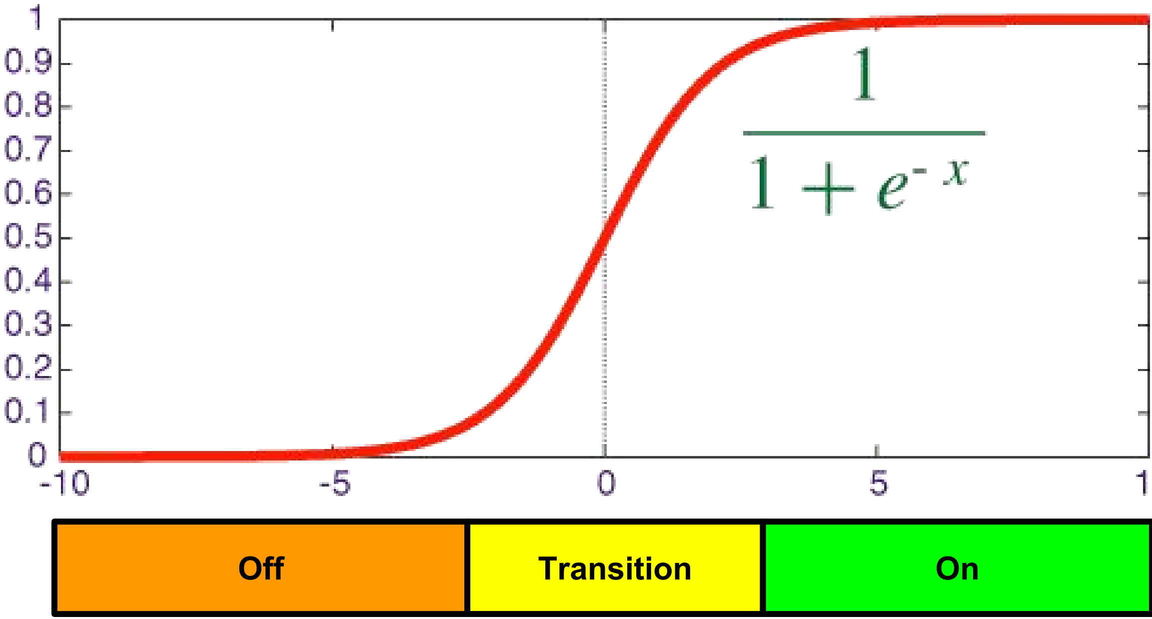




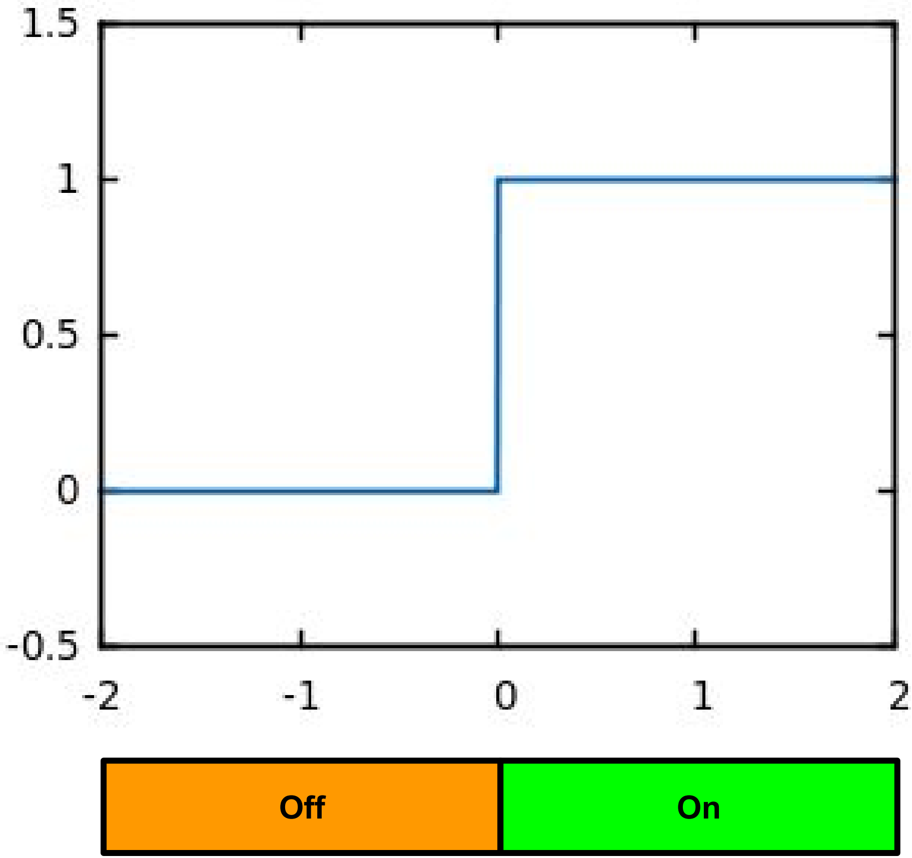
- Non-linear activation:
    - Once linear combination exceeds a cutoff dramatic change in behavior



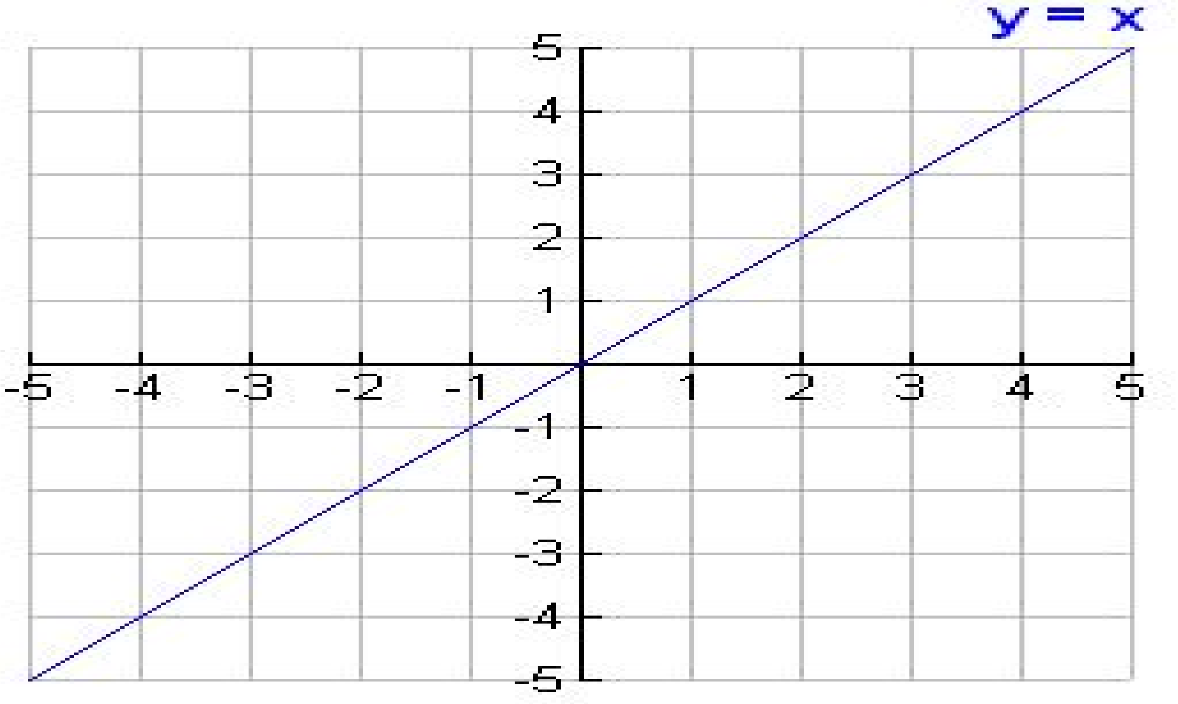
- Non-linear activation:
  - Once linear combination exceeds a cutoff dramatic change in behavior



Q: why use nonlinear activation instead of a *step activation* (no transition)?



Q: why use nonlinear activation instead of *linear activation*?



...even more off	Off	Transition	On	...even more on

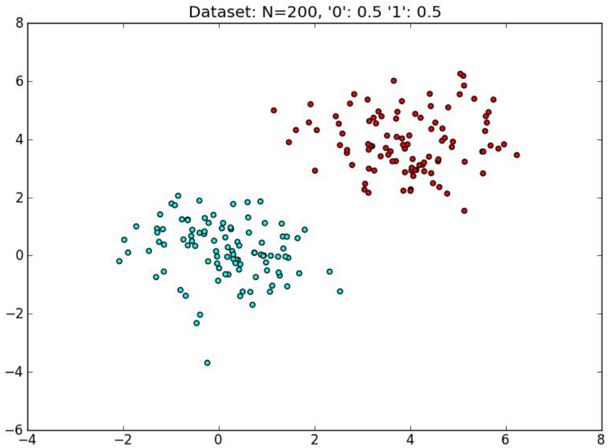
Example: Logistic Regression

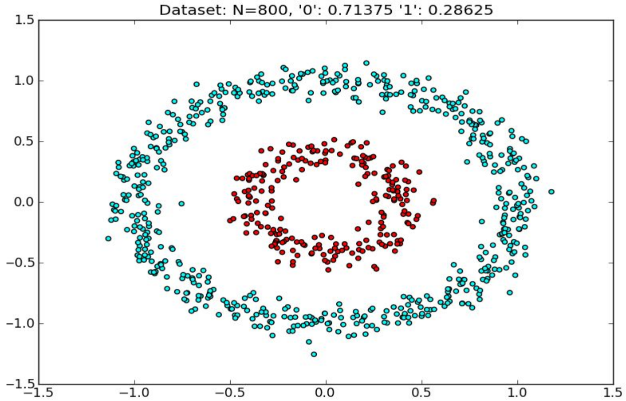
Linear combination of features (log-odds), passed into sigmoid activation function to get *p*.

$$\ell odds = \log \frac{p}{1-p} = x_1\beta_1 + \ldots + x_m\beta_m$$
$$\implies p = \exp(\ell odds)/(1 + \exp(\ell odds)))$$

Layering for nonlinearity

- Single layer nets (e.g. logistic regression) can make linear decisions:





Linearly separable (you can draw a straight line to separate these two colors)

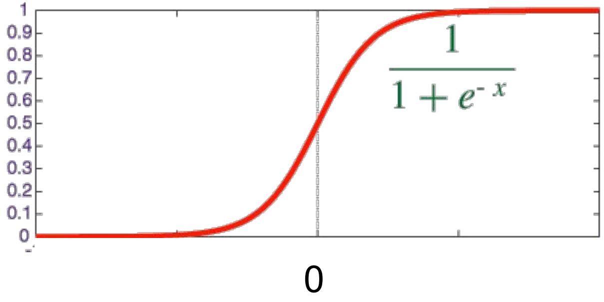
Not linearly separable (you can't draw a straight line to separate the two colors)

Layering for nonlinearity

- Multi-layer nets can approximate any function, i.e. can do any nonlinear separation.
  - Downside: the optimization problem is nonconvex
  - Downside: you may need *lots* of nodes / layers

Neural Net OR

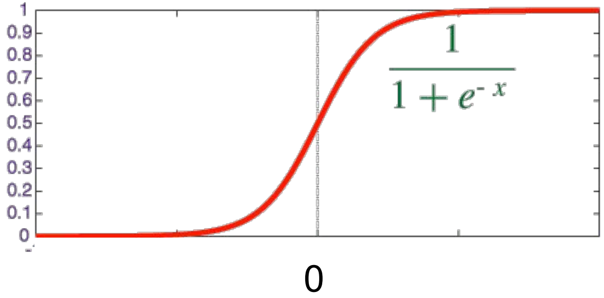
bias
(intercept) **= -15**



	X1	X2	Y
	0	0	0
	0	1	1
	1	0	1
	1	1	1
w	?	?	

Neural Net OR

bias **(intercept)** **= -15**

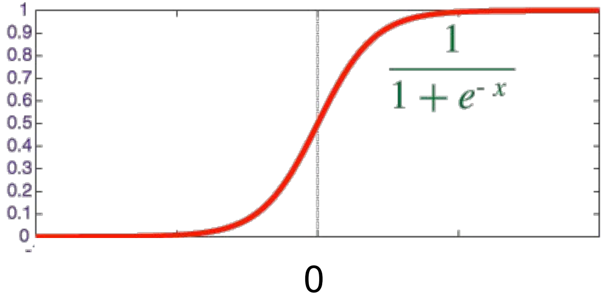


	X1	X2	WX + bias	Y
	0	0	-15	~0
	0	1	5	~1
	1	0	5	~1
	1	1	25	~1
w	+20	+20		

Either X being active enough to turn on Y

Neural Net AND

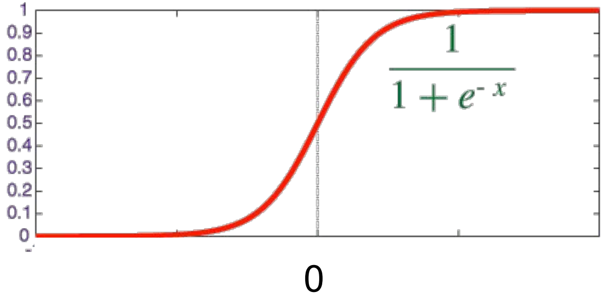
bias (intercept) = -15



	X1	X2	Y
	0	0	0
	0	1	0
	1	0	0
	1	1	1
w	?	?	

Neural Net AND

bias
(intercept) **= -15**



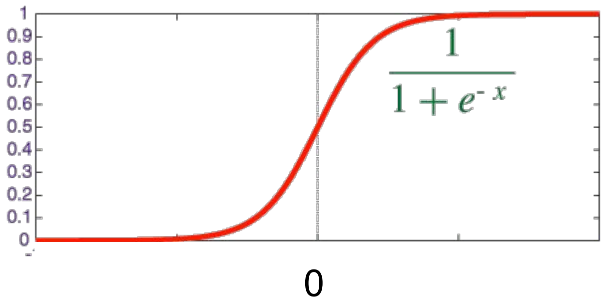
	X1	X2	WX + bias	Y
	0	0	-15	~0
	0	1	-5	~0
	1	0	-5	~0
	1	1	5	~1
w	+10	+10		

Need both X to be active to turn on Y

Neural Net XOR

can't do it with one layer

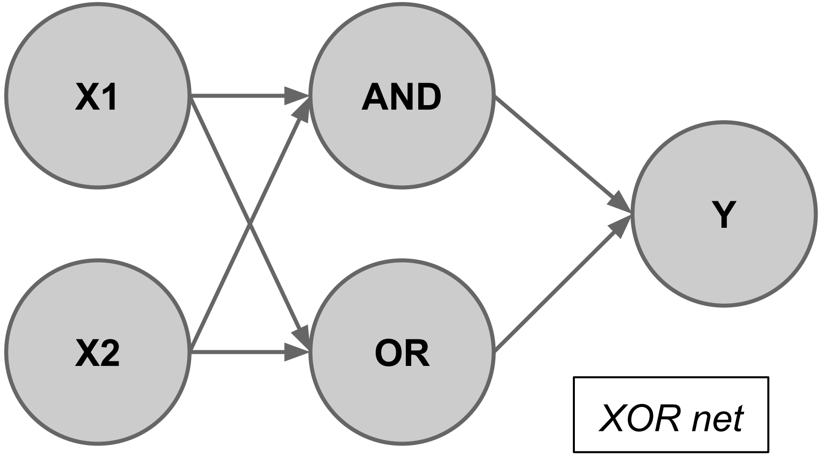
bias (intercept) = -15



	X1	X2	Y
	0	0	0
	0	1	1
	1	0	1
	1	1	0
w	?	?	

Neural Net XOR

*insert layer that does AND, OR (**we know how to make AND and OR operations!**)*



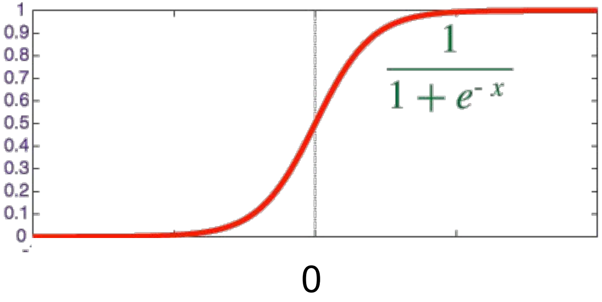
W

X1	X2	X1& X2	X1| X2	Y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0
		?	?	

Neural Net XOR

assume we have outputs of AND and OR layer...

bias (intercept) = -15

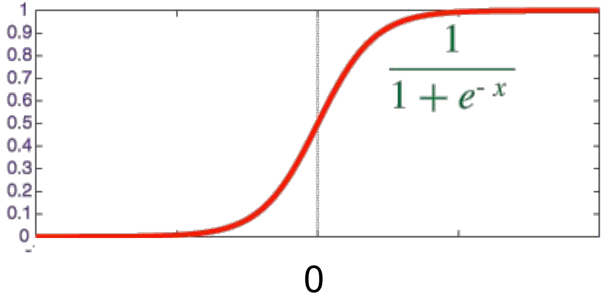


	A1:= X1&X2	A2:= X1|X2	Y
	0	0	0
	0	1	1
	0	1	1
	1	1	0
w	?	?	

Neural Net XOR

assume we have outputs of AND and OR layer...

bias (intercept) = -15



	A1:= X1&X2	A2:= X1|X2	wA + bias	Y
	0	0	-15	~0
	0	1	5	~1
	0	1	5	~1
	1	1	-5	~0
w	-10	+20		

The OR alone can turn Y on, but the AND disables it.

Neural net computation

- In practice: many layers, many nodes, very non-convex
  - Lots of calculations
    - Many nodes
    - Try lots of starting values
  - Parallel computing (most nodes don’t interact with each other)
    - e.g. for XOR, could compute AND, OR in parallel

Backpropagation briefly

- Perceptron:
  - Find error, update parameters so error might get fixed
- Back-propagation:
  - Neural nets fit using gradient descent
  - BP also finds an error, at the output layer, then it sends it backwards through the neural net

2: Theano

Theano

- It looks weird, why? Basically, speed:
  - Python (numpy) is not the most efficient language
  - Python is not parallelized
- Fitting neural nets:
  - Lots of computation
  - Better to be parallel (e.g. use a GPU / map-reduce)

Aside: why GPUs?

- CPU: central processing unit.
  - Does program execution

- GPU: graphics processor unit.
  - Does calculations to render things for display on screen

Aside: why GPUs?

- CPU: central processing unit.
  - Few cores, each fast & “smart”, good @ serial tasks

- GPU: graphics processor unit.
  - Many cores, each slow & “dumb”, good @ parallel tasks

Aside: why GPUs?

- CPU: central processing unit.
    - Few cores, each fast & “smart”, serial
        - Smart: more features, e.g. you can run an OS

- GPU: graphics processor unit.
    - Many cores, each slow & “dumb”, parallel
        - Dumb: it can only do certain tasks well (linear algebra)

Aside: why GPUs?

- CPU: central processing unit.
  - Few cores, each fast & “smart”, serial
- GPU: graphics processor unit.
  - Many cores, each slow & “dumb”, parallel
  - **Better for neural nets: lots of simple, parallel calculations (in sum, faster than the CPU)**

Theano

- Allows construction of “more efficient code”
  - Alternatives: inline a bunch of C code

- Theano can talk to a GPU
  - Recall: GPUs are “dumb”, you usually have to write another language to use them

- For both cases, theano acts like a “foreign language interpreter”

Theano Code: *Symbolic Representation*

```
>>> x = T.dmatrix('x')
>>> s = 1 / (1 + T.exp(-x))
>>> logistic = function([x], s)
>>> logistic([[0, 1], [-1, -2]])
array([[ 0.5       ,  0.73105858],
       [ 0.26894142,  0.11920292]])
```

Theano Code: *Symbolic Representation*

```
>>> x = T.dmatrix('x')
>>> s = 1 / (1 + T.exp(-x))
>>> logistic = function([x], s)
```

I’m confused! Why do we define x and s this way? Where’s the data?

Theano Code: *Symbolic Representation*

```
>>> x = T.dmatrix('x')
```

Define input type

```
>>> s = 1 / (1 + T.exp(-x))
```

Define function behavior

```
>>> logistic = function([x], s)
```

Put it all together

Theano Code: *Optimization (1)*

```
w = theano.shared(np.asarray((np.random.randn(*(numFeatures,
numClasses))*.01)))
```

Weights

```
X = T.matrix()
Y = T.matrix()
```

Data

Theano Code: *Optimization (1)*

```
w = theano.shared(np.asarray((np.random.randn(*(numFeatures,
numClasses))*.01)))
```

Weights

Shared: can update

```
X = T.matrix()
Y = T.matrix()
```

Data

Tensor: cannot update

Theano Code: *Optimization (2)*

```
def model(X, w):
    return T.nnet.softmax(T.dot(X, w))
y_hat = model(X, w)
cost = T.mean(T.nnet.categorical_crossentropy(y_hat, Y))
```

Loss (“cost”)

```
gradient = T.grad(cost=cost, wrt=w)
update = [[w, w - gradient * alpha]]
```

Gradient Descent

Theano Code: *Optimization (2)*

```
def model(X, w):
    return T.nnet.softmax(T.dot(X, w))
y_hat = model(X, w)
cost = T.mean(T.nnet.categorical_crossentropy(y_hat, Y))

gradient = T.grad(cost=cost, wrt=w)
update = [[w, w - gradient * alpha]]
```

Loss ("cost")

Theano "knows" many functions

Theano "knows" the gradient of these functions

Gradient Descent

Theano Code: *Optimization (3)*

```
train = theano.function(inputs=[X, Y],
                        outputs=cost,
                        updates=update)
```

Theano ‘function’ set up:

1. Set inputs (data)
2. Set output (current loss; “cost”)
3. Set update rule (do gradient descent)

Things to try besides the notebook:

Use theano to write:

- An AND neural net

- An OR neural net

- An XOR net

Things to try besides the notebook:

AND, OR rough guide:

1. Copy the logistic regression code
2. What is the data?
    a. Two binary features
    b. Binary output
3. Don’t run gradient descent too long!
4. Instead of checking accuracy, see what weights you get.