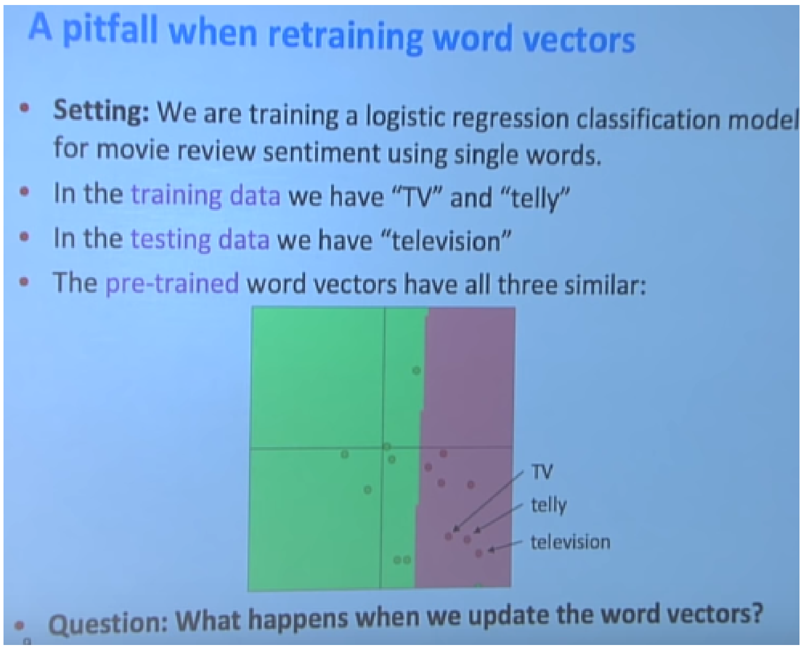
Model which trains Word vectors also:

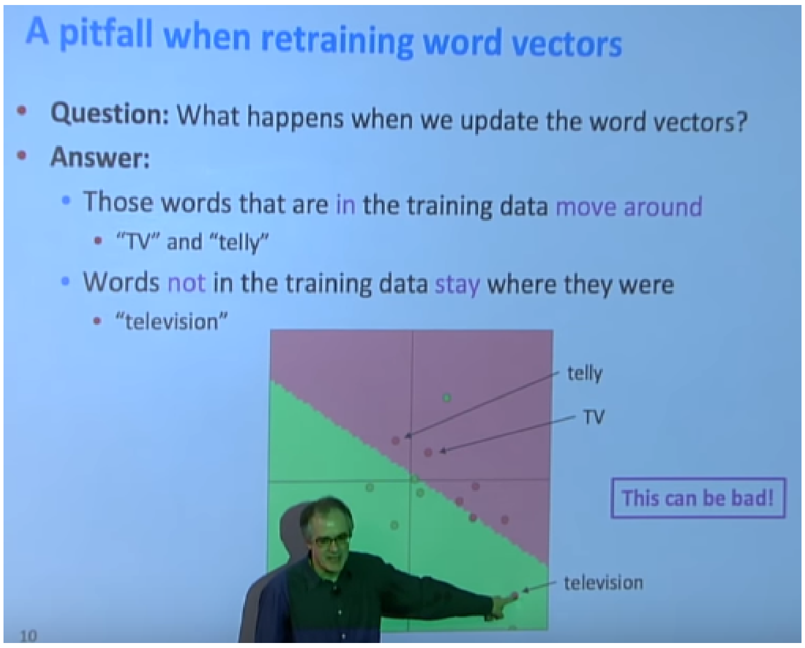
Pushing the gradients into our word vectors has advantages like for example in a case where we train a nn for Named Entity Recognition, if the model is allowed to work around with the word vectors, it can move the word “in” such that every time it sees the word the resulting probability of the center word being a place increases. But it also has its disadvantages.



A pitfall when retraining word vectors

- **Setting:** We are training a logistic regression classification model for movie review sentiment using single words.
- In the training data we have “TV” and “telly”
- In the testing data we have “television”
- The pre-trained word vectors have all three similar:

- **Question: What happens when we update the word vectors?**

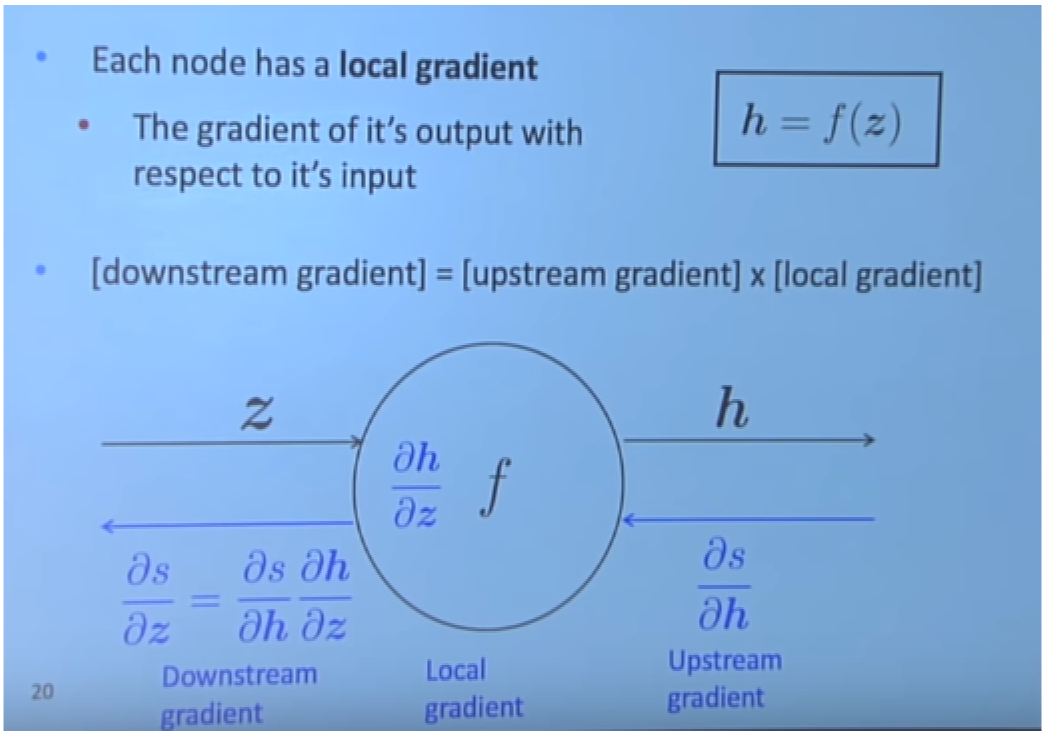


A pitfall when retraining word vectors

- **Question:** What happens when we update the word vectors?
- **Answer:**
  - Those words that are in the training data move around
    - “TV” and “telly”
  - Words not in the training data stay where they were
    - “television”

Moving around the word vectors is usually done in the case of training a model for translation as it is not hard to get a large corpus of translated words and since we have a lot of words, the above mentioned disadvantage vanishes.

Downstream, Local and upstream gradient terms:

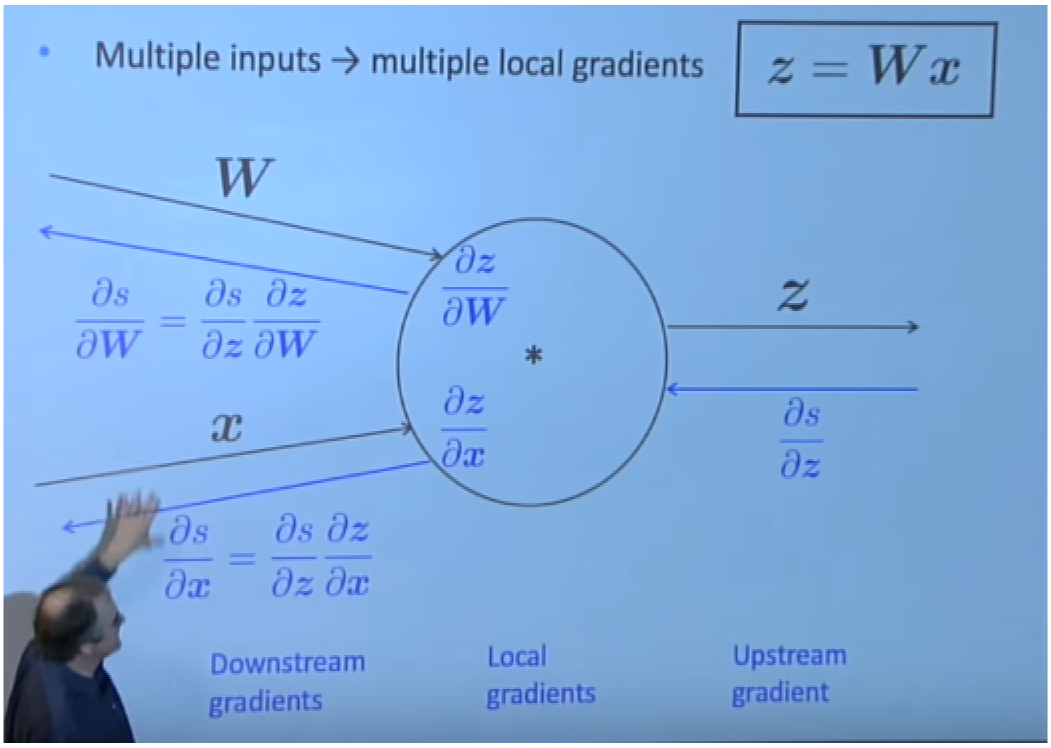


- Each node has a **local gradient**
  - The gradient of it's output with respect to it's input
- [downstream gradient] = [upstream gradient] x [local gradient]

$$\boxed{h = f(z)}$$

$$\frac{\partial s}{\partial z} = \frac{\partial s}{\partial h}\frac{\partial h}{\partial z}$$

z
h
$\frac{\partial h}{\partial z}$
f
$\frac{\partial s}{\partial h}$
Downstream gradient
Local gradient
Upstream gradient
20



- Multiple inputs → multiple local gradients

$$\boxed{z = Wx}$$

$$\frac{\partial s}{\partial W} = \frac{\partial s}{\partial z}\frac{\partial z}{\partial W}$$

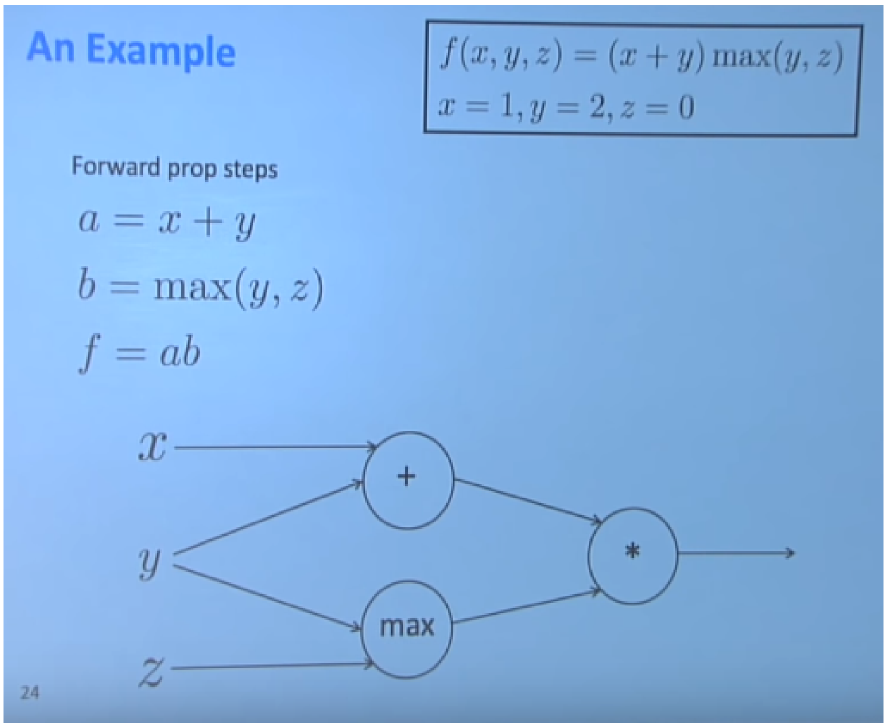
$$\frac{\partial s}{\partial x} = \frac{\partial s}{\partial z}\frac{\partial z}{\partial x}$$

W
x
$\frac{\partial z}{\partial W}$
$*$
$\frac{\partial z}{\partial x}$
z
$\frac{\partial s}{\partial z}$
Downstream gradients
Local gradients
Upstream gradient

This downstream gradient given above is the upstream gradient for the node that comes left of this (the one that comes before it in forward propagation).

Example:

Find the numerical values of df/dx, df/dy, df/dz in the example given below:



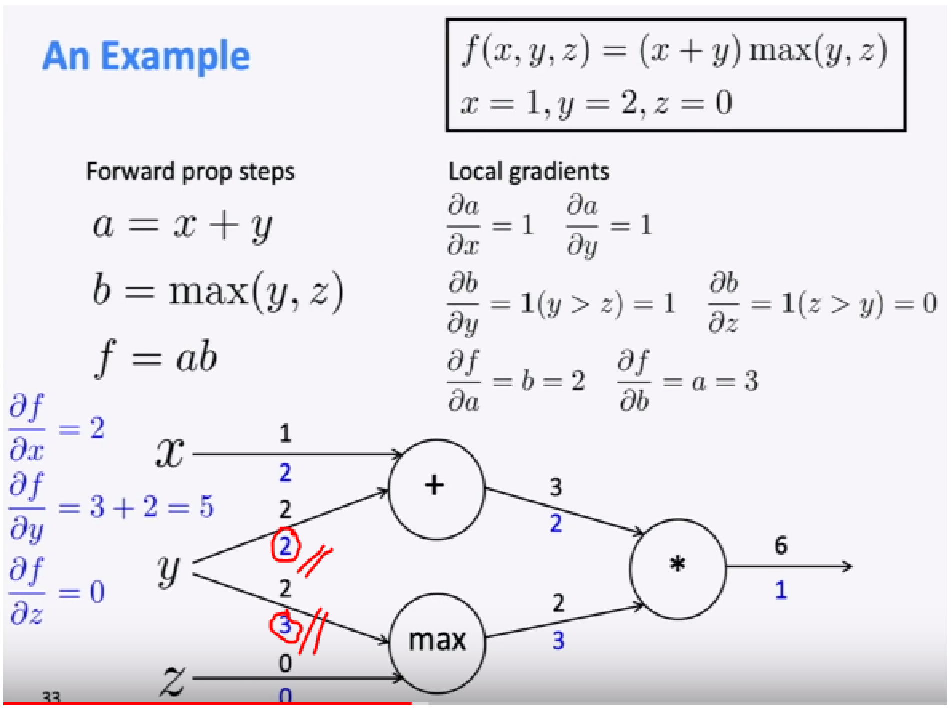
An Example

$$\boxed{\begin{array}{l} f(x,y,z) = (x+y)\max(y,z) \\ x = 1, y = 2, z = 0 \end{array}}$$

Forward prop steps

$$a = x + y$$
$$b = \max(y,z)$$
$$f = ab$$

Solution:



An Example

$$\boxed{\begin{array}{l} f(x,y,z) = (x+y)\max(y,z) \\ x = 1, y = 2, z = 0 \end{array}}$$

Forward prop steps

$$a = x + y$$
$$b = \max(y,z)$$
$$f = ab$$

Local gradients

$$\frac{\partial a}{\partial x} = 1 \quad \frac{\partial a}{\partial y} = 1$$
$$\frac{\partial b}{\partial y} = \mathbf{1}(y > z) = 1 \quad \frac{\partial b}{\partial z} = \mathbf{1}(z > y) = 0$$
$$\frac{\partial f}{\partial a} = b = 2 \quad \frac{\partial f}{\partial b} = a = 3$$

$$\frac{\partial f}{\partial x} = 2$$
$$\frac{\partial f}{\partial y} = 3 + 2 = 5$$
$$\frac{\partial f}{\partial z} = 0$$

Here we have two values for backpropagation of y, What do we do ? We add both as y causes change in f through both the ways independently.

Implementation:

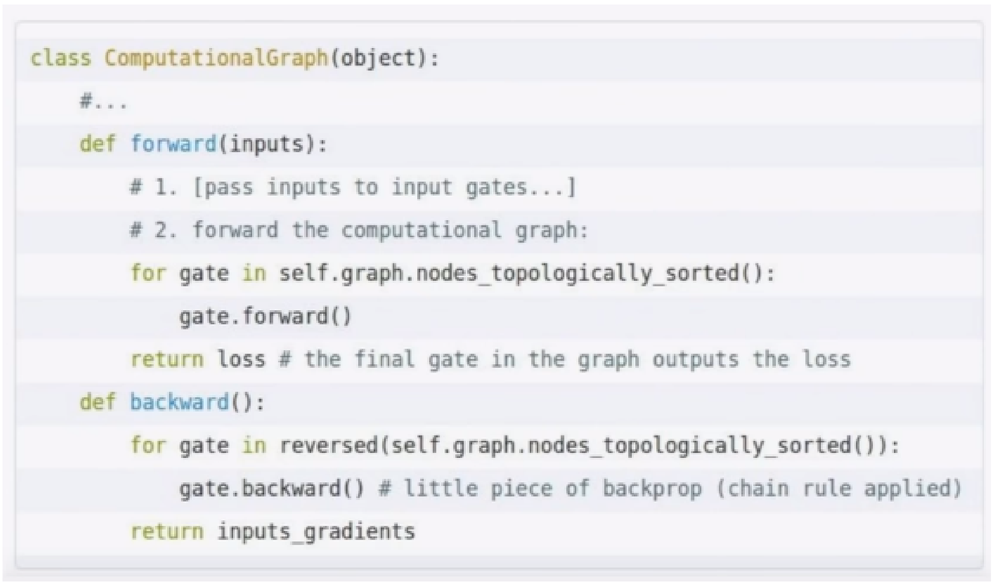
Automatic differentiation:

It is the method of calculation of back prop automatically from the symbolic expression of forward prop. It was tried to be implemented in an older framework but newer frameworks like pytorch and tensorflow leave it behind for manual implementation.

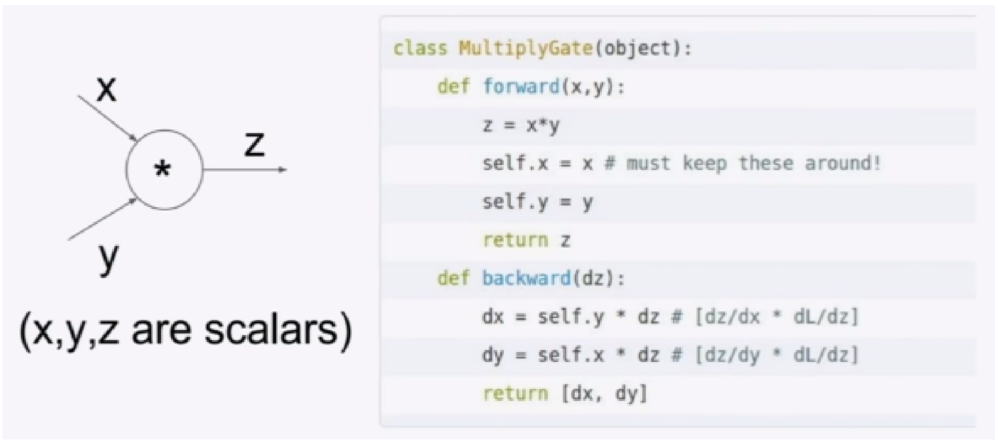
Manual differentiation:

In this method we have to define nodes with their local gradients and frameworks will do the rest of the work for us.

Manual Implementation code

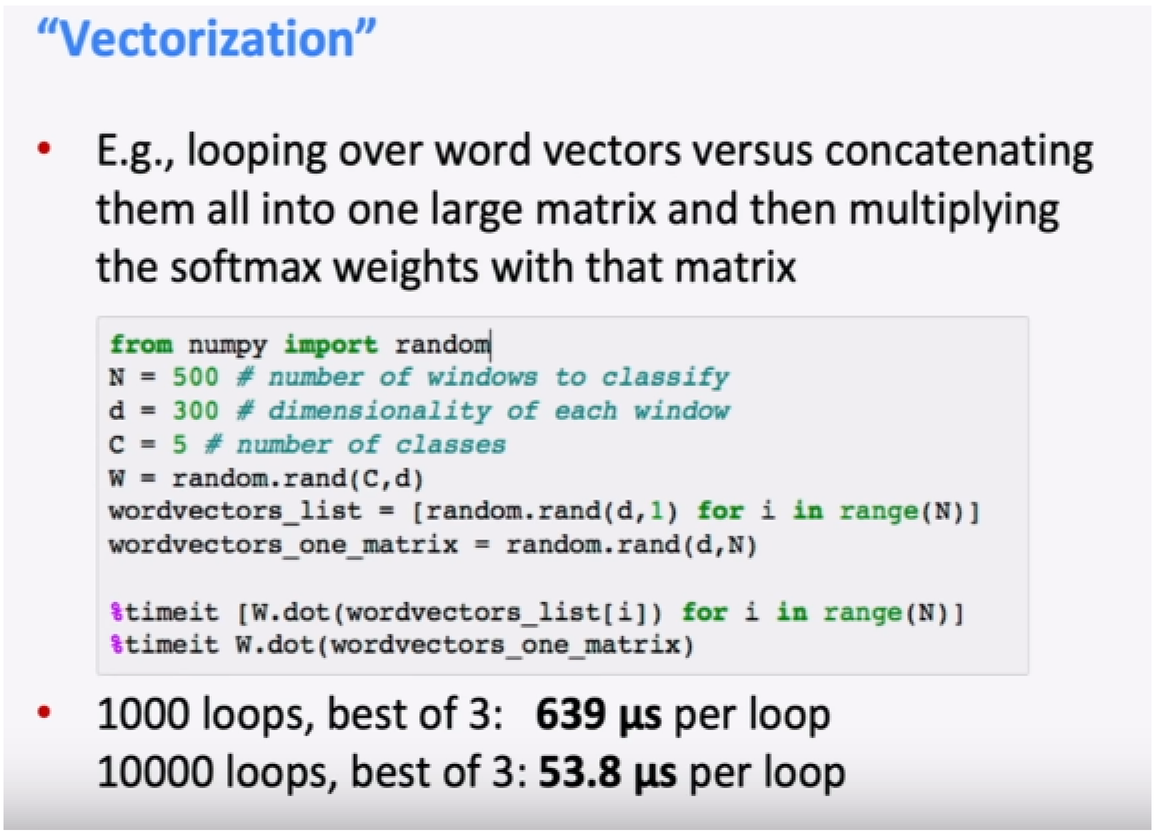


```
class ComputationalGraph(object):
    #...
    def forward(inputs):
        # 1. [pass inputs to input gates...]
        # 2. forward the computational graph:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss # the final gate in the graph outputs the loss
    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward() # little piece of backprop (chain rule applied)
        return inputs_gradients
```



```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        self.x = x # must keep these around!
        self.y = y
        return z
    def backward(dz):
        dx = self.y * dz # [dz/dx * dL/dz]
        dy = self.x * dz # [dz/dy * dL/dz]
        return [dx, dy]
```

Important Points:

1. Don't forget regularization to prevent overfitting.
2. 

"Vectorization"

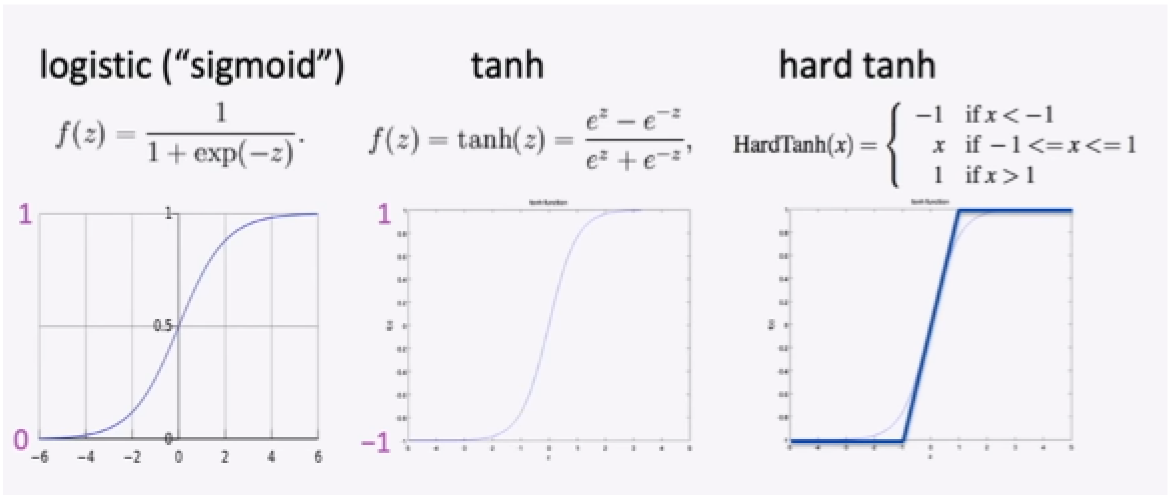
- E.g., looping over word vectors versus concatenating them all into one large matrix and then multiplying the softmax weights with that matrix

```python
from numpy import random
N = 500 # number of windows to classify
d = 300 # dimensionality of each window
C = 5 # number of classes
W = random.rand(C,d)
wordvectors_list = [random.rand(d,1) for i in range(N)]
wordvectors_one_matrix = random.rand(d,N)

%timeit [W.dot(wordvectors_list[i]) for i in range(N)]
%timeit W.dot(wordvectors_one_matrix)
```

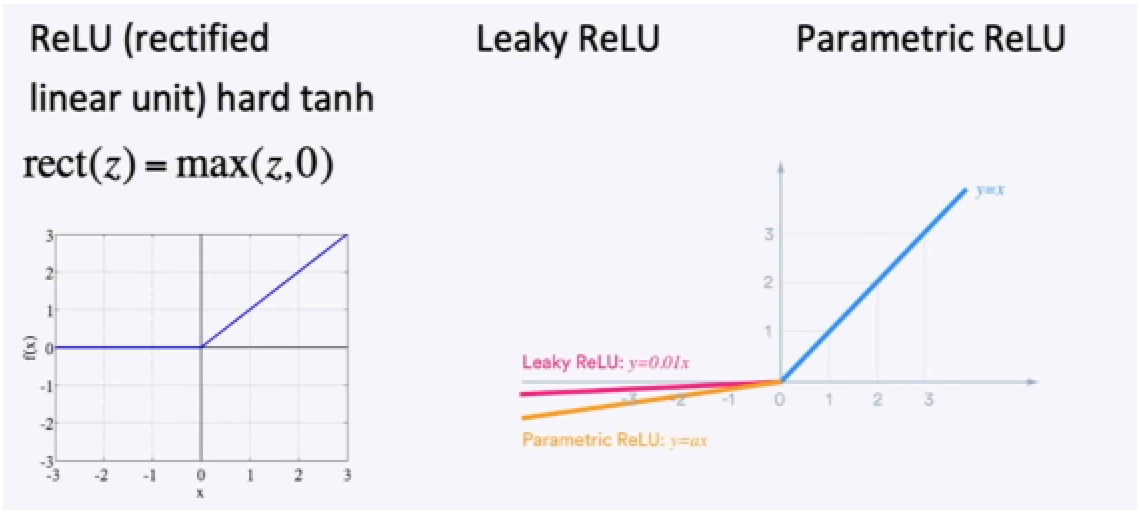
- 1000 loops, best of 3: **639 µs** per loop
  10000 loops, best of 3: **53.8 µs** per loop

3. Important non Linearity functions:



Hard tanh was found because tanh is computationally expensive.

4. ReLU is actually one of the simplest nonlinear functions:



5. We initialize randomly and not zeros just to break the symmetry.
   For it commonly used is xavier initialization.