CS224n: NLP with Deep Learning

Lecture 4: Backpropagation and Computational Graphs

Matrix Gradients

We can analyse the weights individually, before going back to the matrix form: W_{ij}:

- i refers to the position of the hidden layer output
- j refres to the position of the input layer

Thus, W_{ij} only contributes to z_i, and not to the other z_k

Tips

Tip 1

- Carefully define variables at each new step
- Keep track of their dimensions all the time

Tip 2

Chain rule

Tip 3

Derivng softmax:

- separate the cases according to whether we are calculating the softmax of the correct class or not

Tip 4

- Start by doing element-wise derivation, before going global witht the matrix derivation

Tip 5

Use the Shape Convention

- Error message δ at ahidden layer has the same dimensionality as that hidden layer

Words

During training, gradient descent pushes words around

- Always use available "pre-trained" word vectors

Should I update my own word vectors?

- If we have a small training set:
  No
- If our training set is large:
  Yes

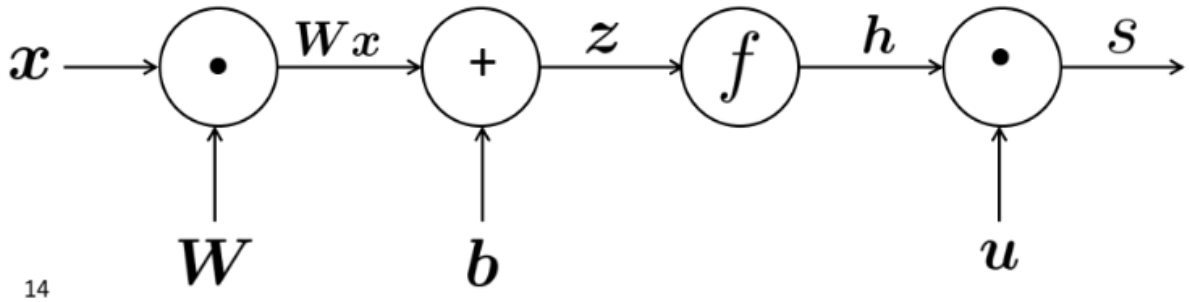
Computational Graphs & Backprop

Forward Propagation

Computation Graphs and Backpropagation

- We represent our neural net equations as a graph
  - Source nodes: inputs
  - Interior nodes: operations
  - Edges pass along result of the operation

$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad (\text{input})$$

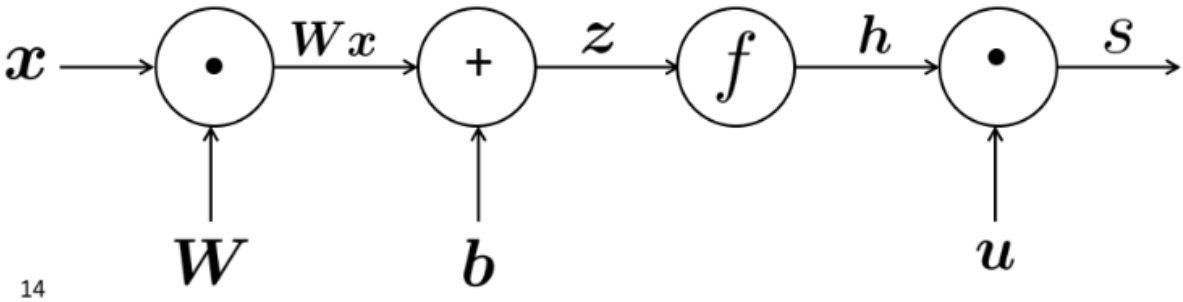


Back Propagation

Computation Graphs and Backpropagation

- We represent our neural net equations as a graph
  - Source nodes: inputs
  - Interior nodes: operations
  - Edges pass along result of the operation

$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad (\text{input})$$



Using the chain rule, we know that:

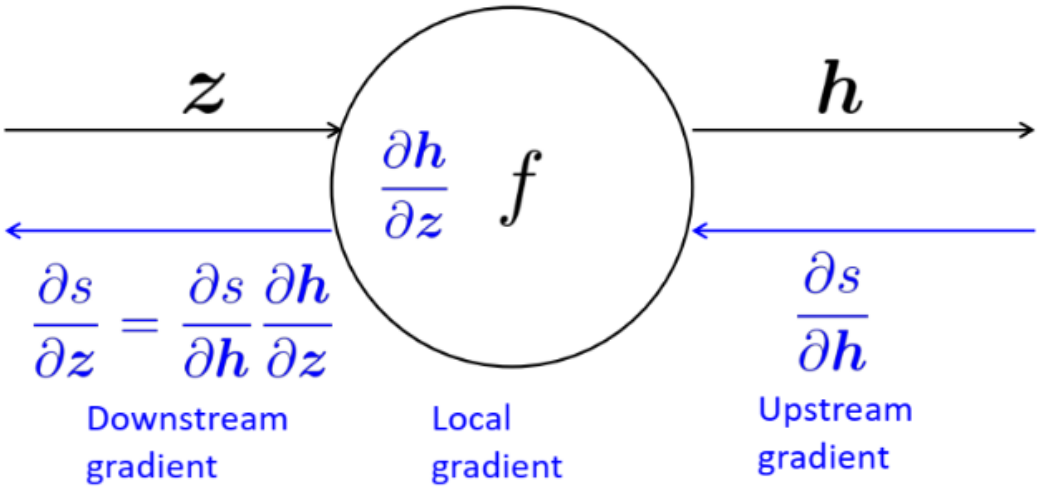
[downstream gradient] = [upstream gradient] x [local gradient]

Backpropagation: Single Node

- Each node has a **local gradient**
  - The gradient of it's output with respect to it's input

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

- [downstream gradient] = [upstream gradient] x [local gradient]



An easy example

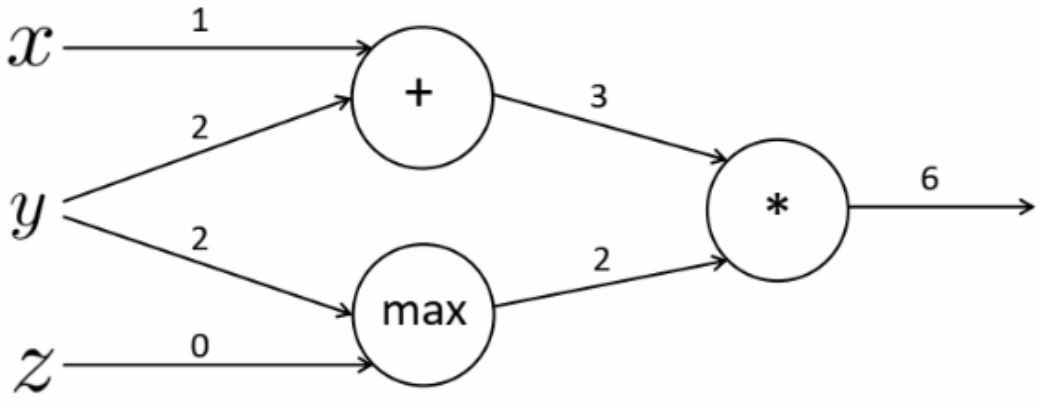
Forward

An Example

$$f(x, y, z) = (x + y)\max(y, z)$$
$$x = 1, y = 2, z = 0$$

Forward prop steps

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



Backward

An Example

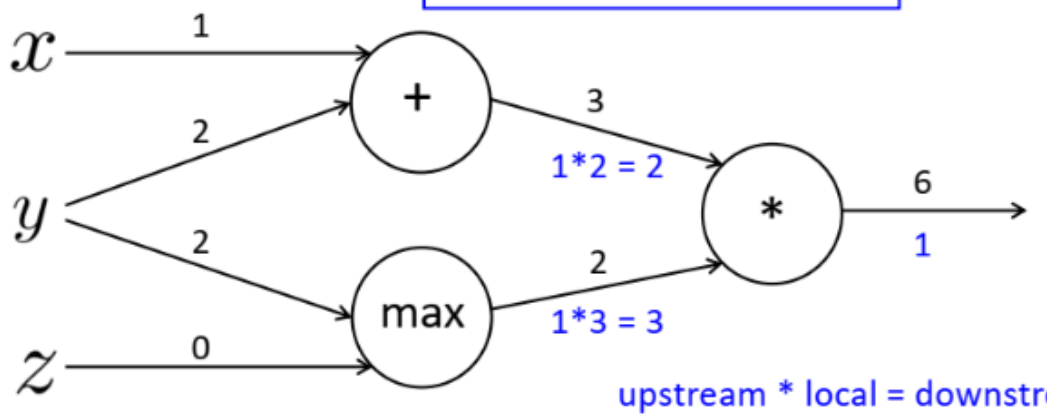
$$f(x, y, z) = (x + y)\max(y, z)$$
$$x = 1, y = 2, z = 0$$

Forward prop steps

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

Local gradients

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



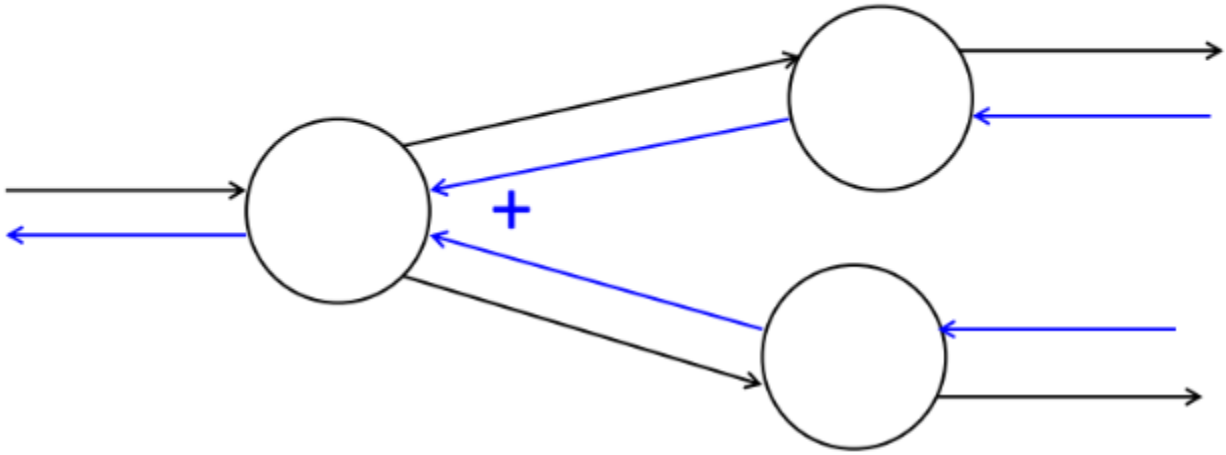
upstream * local = downstream

Don't forget to sum the 2 gradients for y !

Indeed, if we change y a little bit, it is going to impact our result f in 2 ways, one coming through the a path, the other one through the b path

Hence, the need to account for both these paths by summing the impacts of y on them

Gradients sum at outward branches



$$a = x + y$$
$$b = \max(y, z)$$
$$f = ab$$
$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial a}\frac{\partial a}{\partial y} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial y}$$

35

What can we learn from this example?

Node intuitions

- $+$ distributes the upstream gradient among all the inputs
- max routes the upstream gradient to a single input:
  It sends the gradient to the direction of the max inputn while other inputs don't get any
- $*$ switches the upstream gradient

Computing gradients efficiently

- Compute them all at once
- by starting from the output, even if we only need one!

So that we don't compute the same gradients again

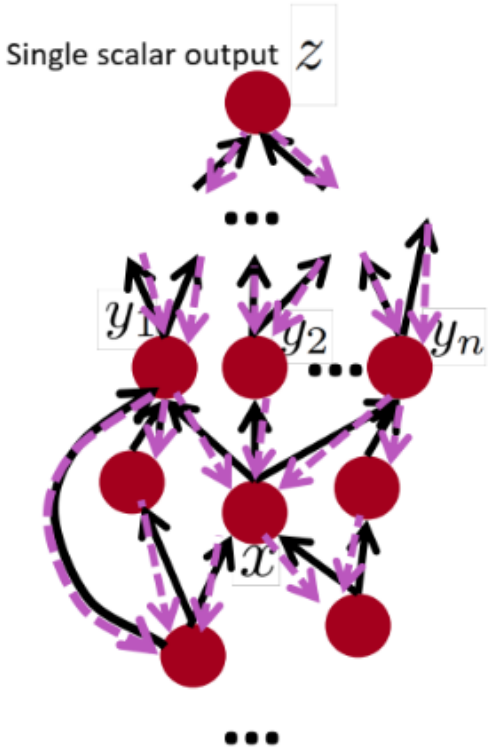
General Method

1. Forward prop in topological order
   (ie calculate the ones that depend on others after these are finished)
2. Back prop

- Initialize output gradient to 1
- Visit nodes in reverse order
- Compute their gradients using their node successors (ie coming from the ouput)

$$\boxed{\frac{\partial f}{\partial x} = \sum_{i=1}^{n} \frac{\partial f}{\partial y_i} \cdot \frac{\partial y_i}{\partial x}}$$

Back-Prop in General Computation Graph



1. Fprop: visit nodes in topological sort order
   - Compute value of node given predecessors
2. Bprop:
   - initialize output gradient = 1
   - visit nodes in reverse order:
     Compute gradient wrt each node using gradient wrt successors
     $\{y_1, y_2, \ldots y_n\}$ = successors of x

$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

Done correctly, big O() complexity of fprop and bprop is **the same**

In general our nets have regular layer-structure and so we can use matrices and Jacobians...

42

Big O() complexity of forward prop, and of backprop !

It is **not** a super complex calculation to run

Implementation Structure

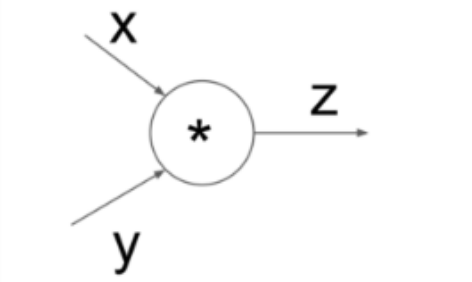
Backprop Implementations

```
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
```


- Don't forget to store the values at each node, even during the forward prop!

Here is an example of why it is useful to do so:

Implementation: forward/backward API



(x,y,z are scalars)

```
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]
```


When we use a Deep Learning framework, somebody has already done this work for us !!

Forward and backward prop have already been implemented!

It is just experimenting, playing with Legos, without needing to understand the underlying math (doesn't mean we shouldn't!)

- But we have to do this if we want to do research, and to create our own algorithms

Gradient checking: Numerical Gradient

- We estimate the slope $f'(x)$ by wiggling x a bit in both directions:
  we just use $f(x-h)$ and $f(x+h)$

- Easy to implement, thus useful for an easy check

- However, very slow: we have to evaluate our functions on millions of parameter combinations
  ==> Thus, not used in practice for computation

- Hence, just implement it using a *if* statement to (de)activate it for debugging

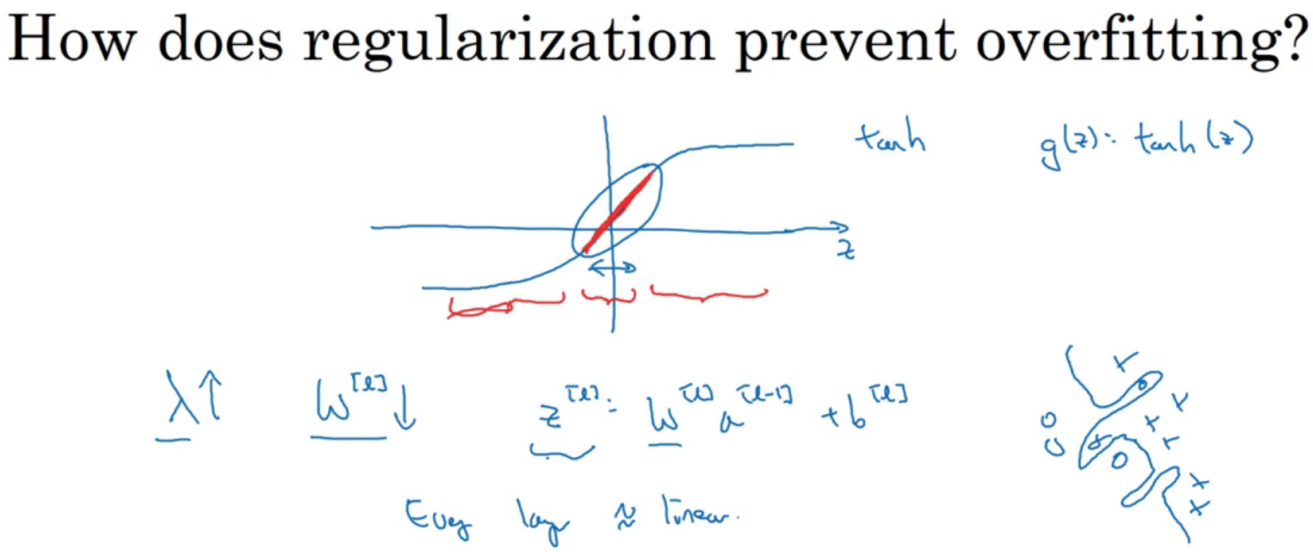
- Much less needed now, when we can just throw layers easily using well-written frameworks

---

Stuff to know

Regularization

- Add a L2 regularization so that our parameters don't get too big
  => Prevent overfitting
  - As our weights are quite small, the values that are inputted to our activation functions (such as tanh, sigmoid...) will not be saturated
  - They will instead be in the linear part of the activation function, and hence the neural network will output a combination of linear models, ie a linear model
  - Hence, the overfitting prevention: staying close to a linear (thus simple) decision boundary



- Regularization is even more important in Deep Learning than in Classical Machine Learning, because Deep Learning has much more parameters

- On the other hand, L1-regularization will compress several weights to 0, and thus operate a feature selection

Vectorization

- Vectors and other higher-dimensional objects:
  - 1D: Vectors
  - 2D: Matrices

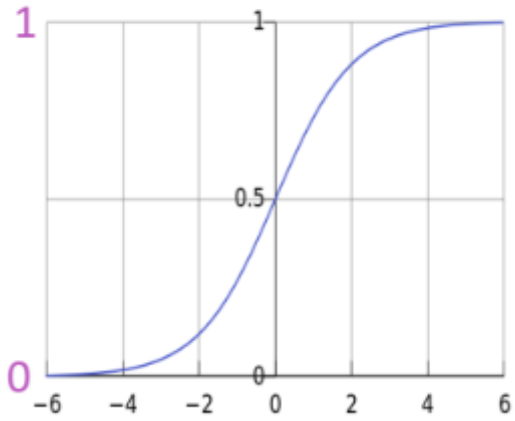
    - 3D: Tensors

- Always use vector and matrix implementations to compute much faster
  (as opposed to *for*-loops)

Non-linearities

- Sigmoid (logistic function)
    - Used a lot in the beginnings of Neural Nets
      Not much used now!

$$f(z) = \frac{1}{1 + \exp(-z)}.$$



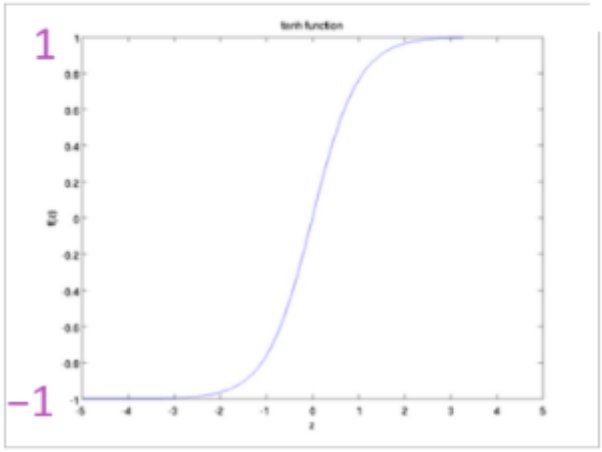
- tanh
    - tanh is just a rescaled and shifted sigmoid!

$$tanh(z) = 2\ logistic(2z) - 1$$

    - It is now symmetrical between [-1,1], which helps a lot for Neural Nets
    - However, its derivatives are quite expensive to compute!

tanh

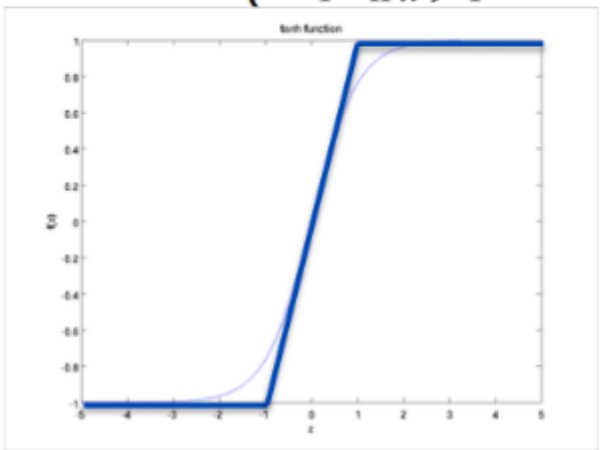
$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}},$$



- Hard tanh
  - The involved math is easier than with tanh
    Thus faster to compute!
  - The gradient becomes 0 when it gets at either end of the function, so things go "dead"
    Thus, try to stay in the middle part long enough!
    - If even such a simple function works, why not go even more simple? 😁

hard tanh

$$\text{HardTanh}(x) = \begin{cases} -1 & \text{if } x < -1 \\ x & \text{if } -1 <= x <= 1 \\ 1 & \text{if } x > 1 \end{cases}$$



- ReLU
  - Trains fast
  - Performs well !

What everyone uses Now !

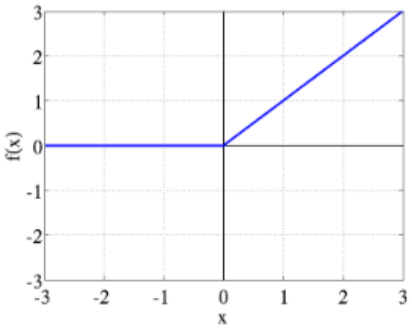
Non-linearities: The new world order

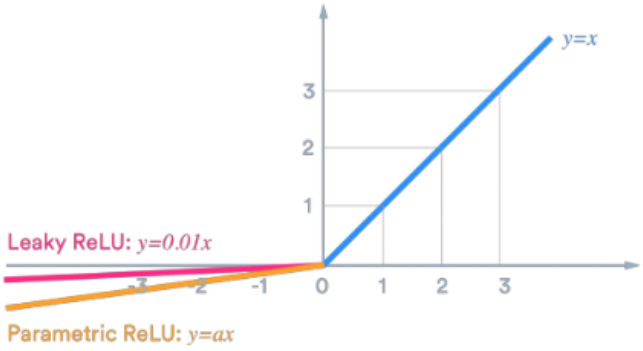
ReLU (rectified linear unit) hard tanh

Leaky ReLU

Parametric ReLU

$$\text{rect}(z) = \max(z, 0)$$





- For building a feed-forward deep network, the first thing you should try is ReLU — it trains quickly and performs well due to good gradient backflow

Parameter Initialization

- Initialize weights with small random values to break symmetry
  - Small so that we stay in the middle zones of the activation functions
- Use Xavier initialzation

Optimizers

- Usually, plain SGD works fine
- For more complex neural nets, we can use more sophisticated "adaptive' optimizers:
  - Adagrad
  - RMSprop
  - **Adam**: a safe starter choice !
  - ...

Learning Rates

- First, try different powers of 10 to get the right order of magnitude

- Better results by decreasing the learning rate as we train
  - By hand: halve learning rate very *k* epochs
  - Using a formula, for example an exponential decaying learning rate

- Use a higher learning rate with fancier optimizers, as it will shrink rapidly