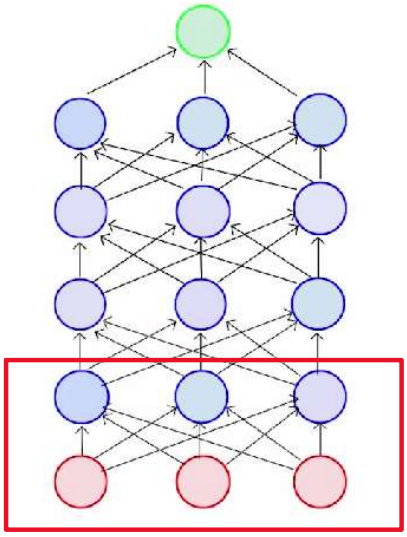
Unsupervised Pre-Training



Consider this deep neural network, We will train the weights only b/w the first two layers x & h_1.

Goal: Reconstruct x such that error will be minimum.

$$\min \frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{n} (\hat{x}_{ij} - x_{ij})^2$$

What if $\hat{x} = x$?

It means h_1 has perfectly captured all the important information about x.

This is referred to as unsupervised as it does not involve the label y & only the data x.

We will now repeat this process again by adding one more layer h_2.

Goal: Reconstruct h_1 such that error will be minimum

Now h_2 will be an abstract representation of h_1 similar to h_1 was of x $\Rightarrow$ h_2 is a deeper abstract representation of x

We keep on repeating this until all L layers are able to reconstruct the previous one perfectly.

Coming back to our original problem statement, given x & y learn the relationship b/w them: $f(x)$.

Goal: $\arg\min \frac{1}{} \sum^{m} (y_i - \hat{f}(x_i))^2$

Goal: $\arg\min \frac{1}{m} \sum_{i=1}^{m} (y_i - \hat{f}(x_i))^2$

We'll not be randomly initializing weights, but use the weights we learned from the above unsupervised pre-training.

Why does this greedy unsupervised training of weights work better?

Better optimization?

OR

Better regularization?

Optimization vs Regularization

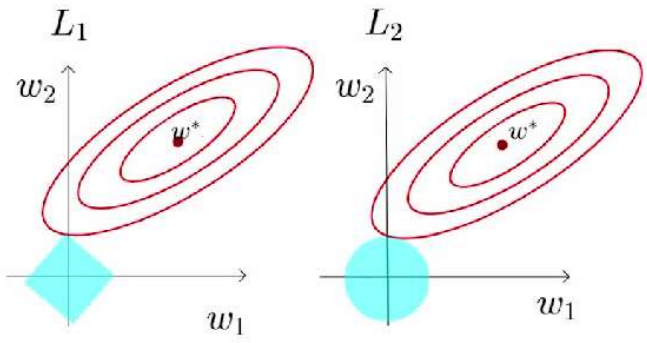
What is the optimization problem that we are trying to solve?

$$\text{minimize } L(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(y_i - \hat{f}(x)\right)^2$$

Given a large capacity of network in last layer, then $L(\theta)$ goes to 0 even w/o pre-training. However, this is computationally expensive.

If the capacity of the network is small, unsupervised pre-training helps.

What does regularization do? It constraints the weights to certain region of the parameter space.



L1 regularization constraints most weights to be 0

L2 regularization prevents most weights from taking larger values

Unsupervised pre-training constraints the weights to lie in only certain regions of the parameter space.

Specifically, regions where the characteristics of the data are captured well.

$$\Omega(\theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{\tilde{}}\left(x_{ij} - \hat{x}_{ij}\right)^2$$

$$J_L(\theta) = \frac{1}{m} \sum_{i=1}^{} \sum_{j=1}^{} (\quad)$$

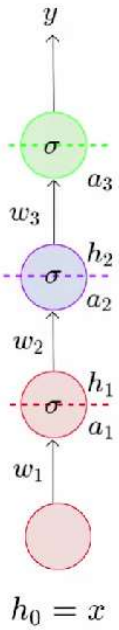
We can consider this unsupervised objective as an additional constraint on the optimization problem

Some experiments have also shown that pre-training is more robust to random initializations.

Better Activation Functions

What makes Deep Neural Networks powerful?

Consider a deep neural network where we replace the sigmoid in each layer by simple linear transformation.



$a_1 = W_1 x$

$h_1 = a_1 = W_1 x$

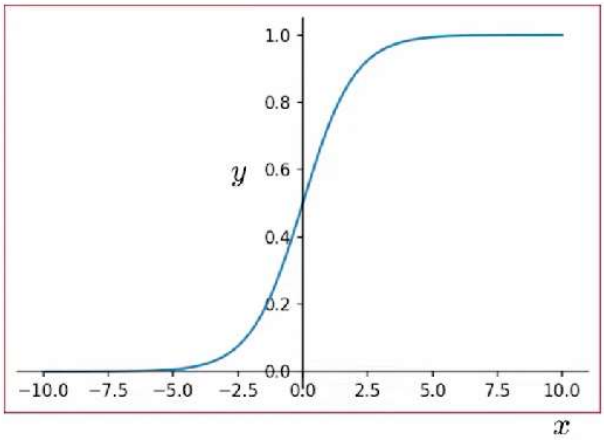
$\therefore y = (w_4 * (w_3 * (w_2 * (w_1 * x))))$

$\therefore$ This is just a linear transformation of x.

In other words, y is constrained to learning linear decision boundaries

$\therefore$ The activation functions are what which give neural networks their power.

Sigmoid



$\sigma(x) = \frac{1}{1+e^{-x}}$, $[0,1]$

$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1-\sigma(x))$

Saturated neurons: Gradients vanish.

Why would neurons get saturated?

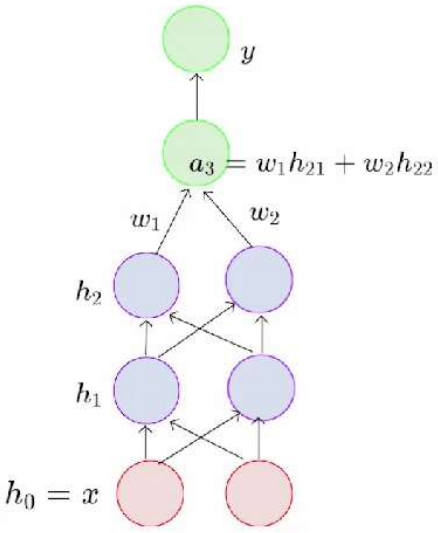
Why would neurons get saturated?

When weights are initialized to very high values.

$\sigma\left(\sum_{i=1}^{n} w_i x_i\right) \rightarrow 1$ ∴ it'll get saturated.

& training will get stalled.

Sigmoids are not centered. Why is this a problem?

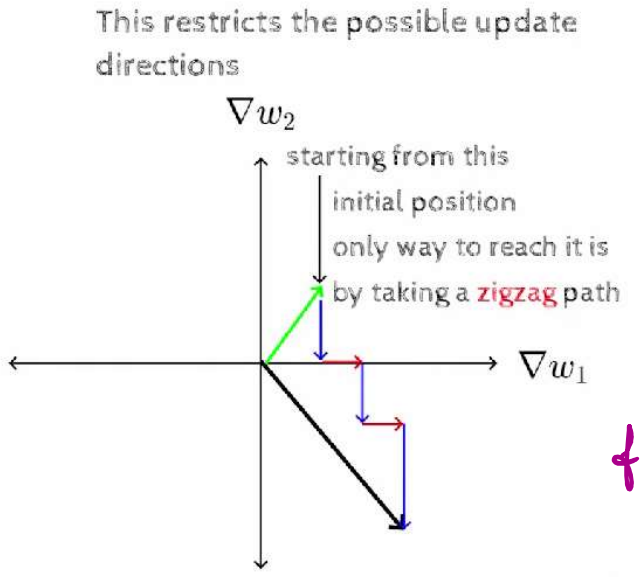


$$\nabla w_1 = \frac{\partial L(w)}{\partial y} \cdot \frac{\partial y}{\partial h_3} \frac{\partial h_3}{\partial a_3} \cdot h_{21}$$

$$\nabla w_2 = \frac{\partial L(w)}{\partial y} \cdot \frac{\partial y}{\partial h_3} \frac{\partial h_3}{\partial a_3} \cdot h_{22}$$

$h_{21}, h_{22} \in [0,1]$ ie. positive.

So if the earlier parts of ∇w_1 & ∇w_2 are +ve then $\nabla w_1, \nabla w_2$ are +ve & vice versa.

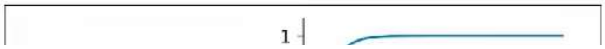


Sigmoids are computationally expensive.

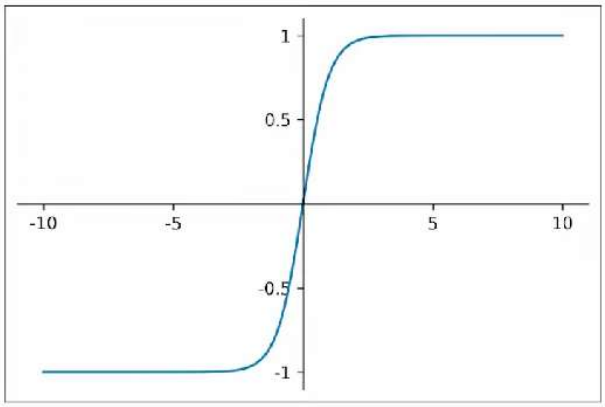
Because of exponential computation

Let us see another activation function

tanh



$[-1, 1]$



$[-1, 1]$

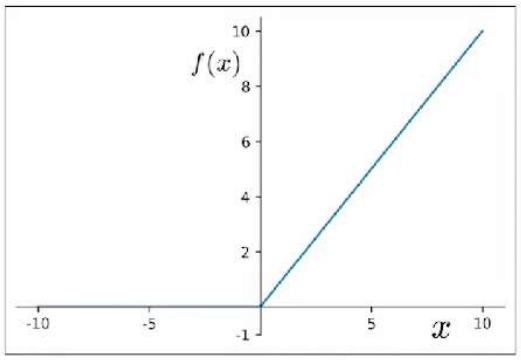
$\frac{\partial \tanh(x)}{\partial x} = (1 - \tanh^2(x))$

Zero centered

The gradient still vanishes at saturation.

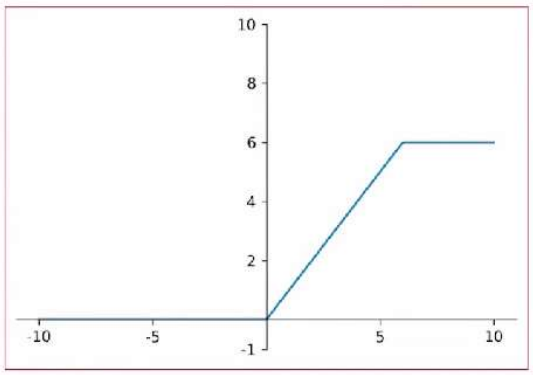
Computationally expensive as well.

ReLU



$f(x) = \max(0, x)$

- One of the most popular activation functions
- It is a non-linear function
- We can combine two ReLU units to recover a piecewise linear approximation of the scaled sigmoid function



$f(x) = \max(0, x) - \max(0, x-6)$

- Also called ReLU6 (↑ max value)

Advantages of ReLU

- Does not saturate
- Computationally effective
- converges faster than Sigmoid/tanh

$$\frac{\partial ReLU(x)}{\partial x} = \begin{cases} 0 & \text{if } x<0 \\ 1 & \text{if } x>0 \end{cases}$$

What would happen if at some point a large gradient causes the bias b to be updated to a large negative value ? $b << 0$

$\therefore w_1 x_1 + w_2 x_2 + b < 0$ [if $b << 0$]

$\therefore$ The neuron would output 0 [dead neuron]

even $\frac{\partial h_1}{\partial a_1}$ would be 0

The weights won't be updated

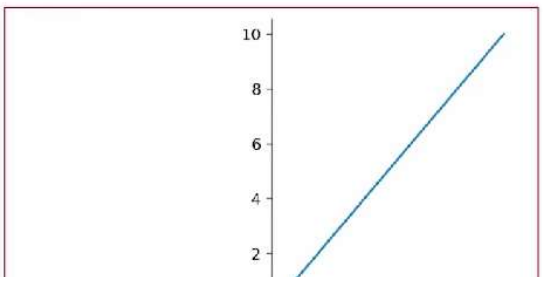
as $\nabla w_1 = \frac{\partial L(\theta)}{\partial y} \frac{\partial y}{\partial a_2} \frac{\partial a_2}{\partial h_1} \frac{\partial h_1}{\partial a_1} \frac{\partial a_1}{\partial w_1}$

In practice, a large fraction of ReLU units can die if the learning rate is set ~~too~~ high.

$\therefore$ It is advised to initialize bias to a positive value (0.01)

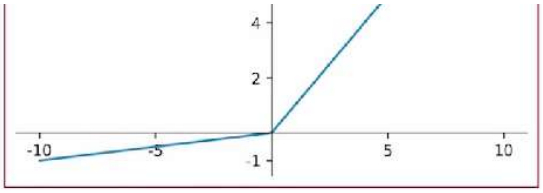
Solution ?

Leaky ReLU



$f(x) = \max(0.1x, x)$

- No saturation
- Will not die ($0.1x$ will always ensure at least a small gradient



- Will not die ($0.1x$ will always ensure at least a small gradient will flow through)
- Computationally efficient
- Close to zero centered o/p.

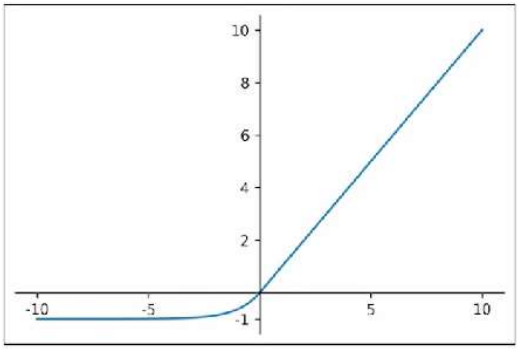
Parametric ReLU

$f(x) = (\alpha x, x)$

α -> parameter of the model.

α will get updated during backpropagation

Exponential Linear Unit (ELU)



$f(x) = x$ if $x > 0$

$= ae^x - 1$ if $x \leq 0$

- All benefits of ReLU
- $ae^x - 1$ ensures at least some gradient will flow through.
- Close to zero centered o/p
- Expensive

MaxOut

In bagging, the (sub)set of training samples seen by each of the models does not change across epochs. Therefore, the model weights get optimized for those samples.

We do not have that luxury with Dropouts. Especially for deep models where the number of sub-models is exponentially high.

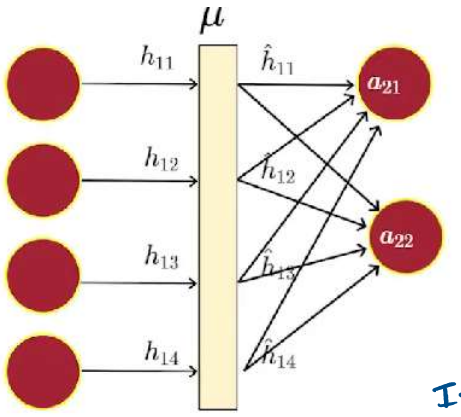
Each sub-model is samples rarely & sees only some parts of the training data.

Therefore, we want the updates to be largeer.

We can't set the learning rate as very high as parameters are shared across the models.

What is the solution?

Dropout



$\mu \sim$ Bernoulli (p) (Mask)

$p = 0.5$

$a_{21} = w_{11}\hat{h}_{11} + w_{14}\hat{h}_{14}$

$a_{22} = w_{21}\hat{h}_{21} + w_{24}\hat{h}_{24}$

If ReLU activation was used &

$0 < a_{21} < a_{22}$ ← reacting more positively to training sample as o/p is higher

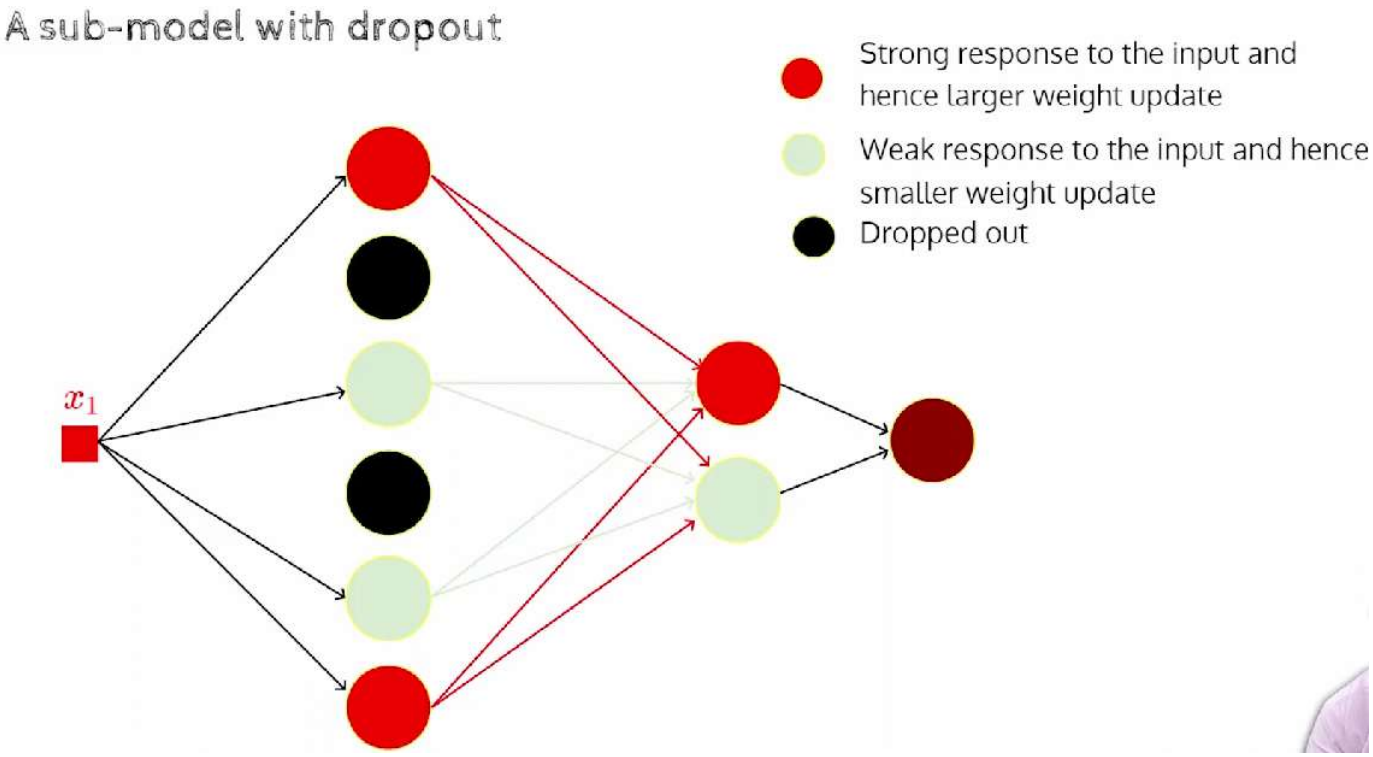
training sample as o/p is higher

$\therefore a_{22}$ is more sensitive to the training sample than a_{21}.

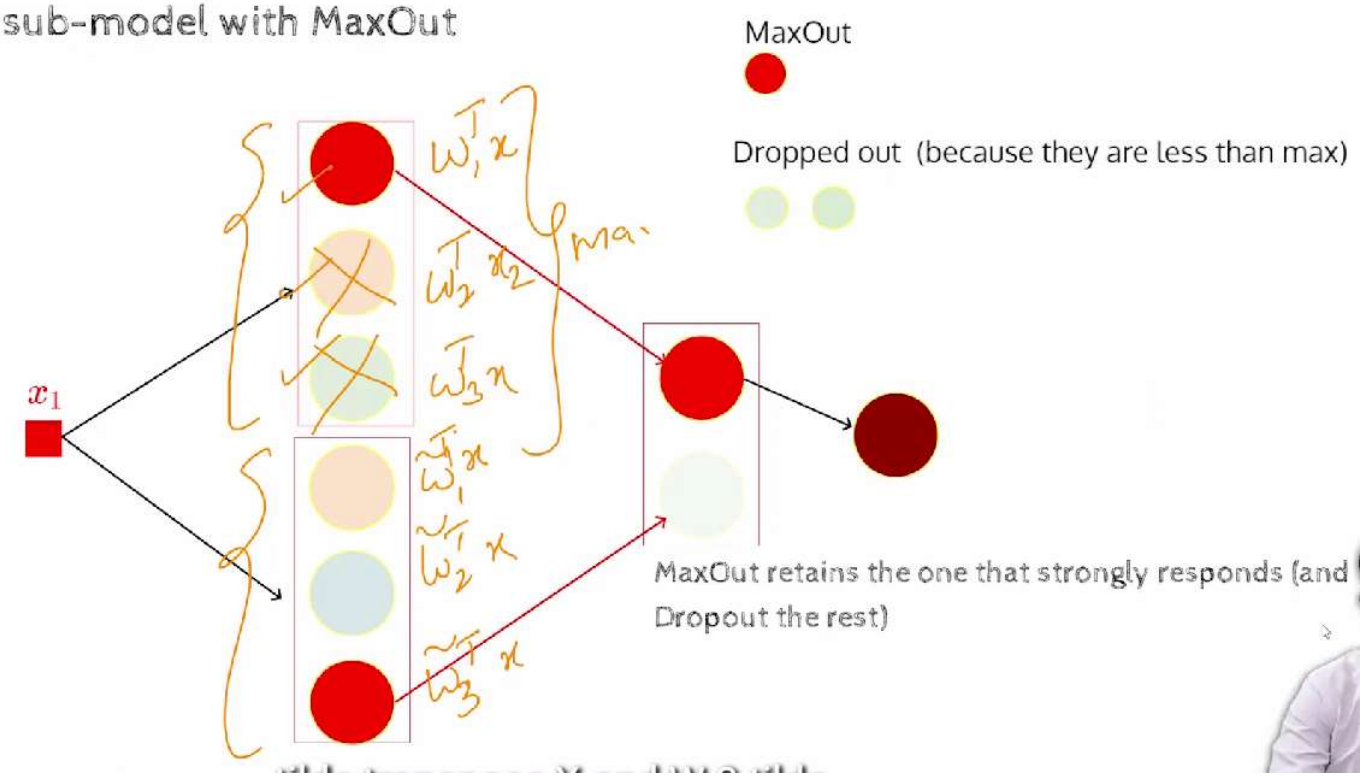
We can acknowledge this by the following:

$$\max(a_{21}, a_{22})$$

$$\therefore \frac{\partial h_{21}}{\partial a_{21}} = 0 \quad \& \quad \frac{\partial h_{21}}{\partial a_{22}} = 1$$

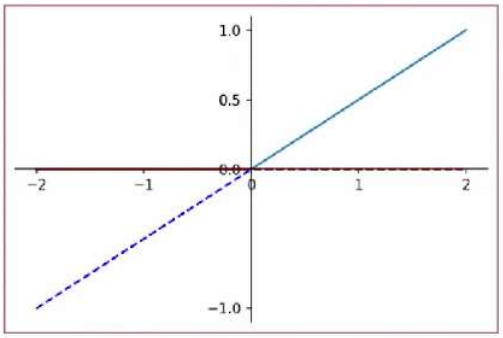


A sub-model with MaxOut

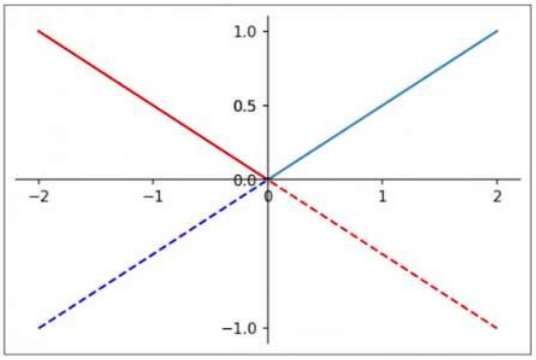


Maxout Neuron : $\max(w_1 x + b_1, \cdots, w_n x + b_n)$

$max(0x + 0, w_2^T x + b_2)$



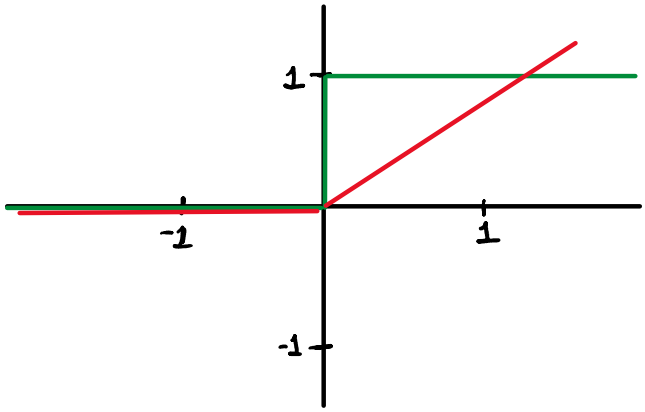
$max(0.5x, -0.5x)$



Generalizes ReLU and Leaky ReLU

GELU to SILU

GELU (Gaussian Error Linear Unit)



hard thresholding

$$f(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x < 0 \end{cases}$$

$$f(x) = \begin{cases} 1 \cdot x & \text{if } x > 0 \\ 0 & \text{if } x < 0 \end{cases}$$

ReLU

In both activation function, output depends upon sign of input.

$$f(x) = \begin{cases} 1 \cdot x & \text{if } x > 0 \\ 0 \cdot x & \text{if } x < 0 \end{cases}$$

Dropout does the same Except, it does it stochastically

$$M(x) = \begin{cases} 1 \cdot x & , p \\ 0 \cdot x & , 1-p \end{cases}$$

Combining these properties into an activation function:

$$f(x) = m \cdot x$$

$$m \sim \text{Bernoulli}(\phi(x)) \quad , \quad \phi(x) \in [0,1]$$

$\phi(x)$ can be logistic but the more natural choice is cdf of standard normal distribution.

$$E[f(x)] = E[m \cdot x] = \phi(x) \cdot 1 \cdot x + (1 - \phi(x)) \cdot 0 \cdot x$$

$$= x\phi(x)$$

$$= x\,\phi(x)$$

$$= P(X \leq x) \cdot x$$

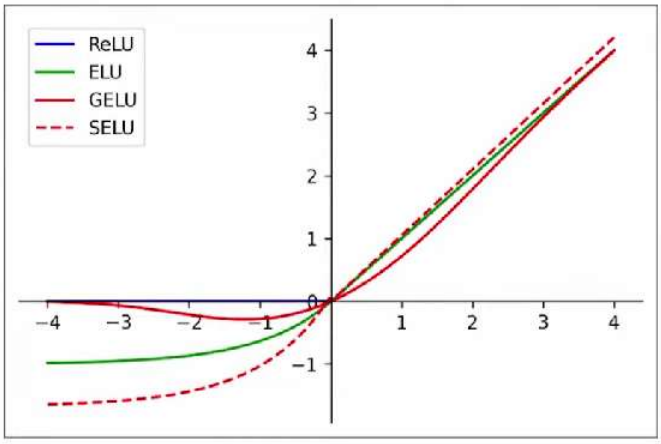
CDF of Gaussian $(\mu=0, \sigma=1)$ is computed w/ error function. The approximation of error function is given by:

$$\approx 0.5x\left(1 + \tanh\left[\frac{\sqrt{2}}{\sqrt{\pi}}\left(x + 0.044715x^3\right)\right]\right)$$

GELU $\quad f(x) \approx x \cdot \sigma(1.702x)$

SELU (Scaled Exponential Linear Unit)

It centers the output activations from each layer to zero mean & unit variance. Called self normalizing.



$$f(x) = \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha e^x - \alpha & \text{if } x \leq 0 \end{cases}$$

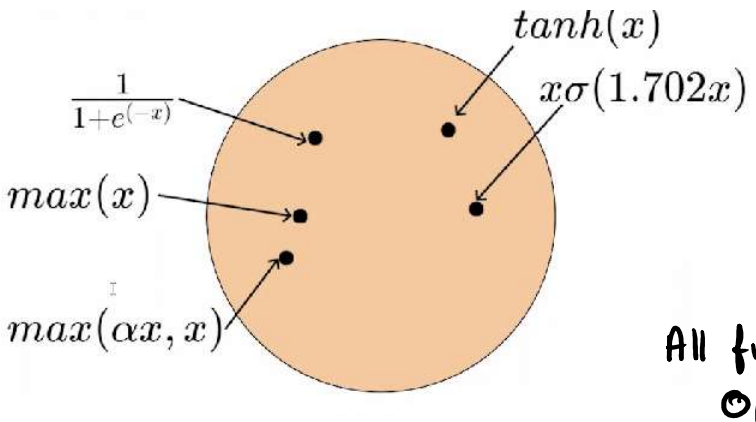
$\lambda \approx 1.0507009$,
$\lambda \approx 1.673263$

few suggested values

Automatic Search for Activation Functions

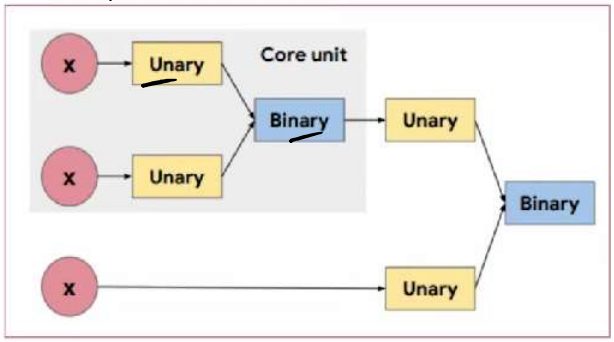
How long can we keep discovering these functions? Is there a better way of searching these activation functions?

functions?



Assume that the left circle is a search space for activation functions (scalar functions)

All functions are composed of two operations



∴ It is logical to assume that we can compose the functions in many combinations & see which composition gives me the best activation functions

We have arrived at the conclusion that any function of the below form is a good activation function.

$$f(x) = x\sigma(\beta x)$$

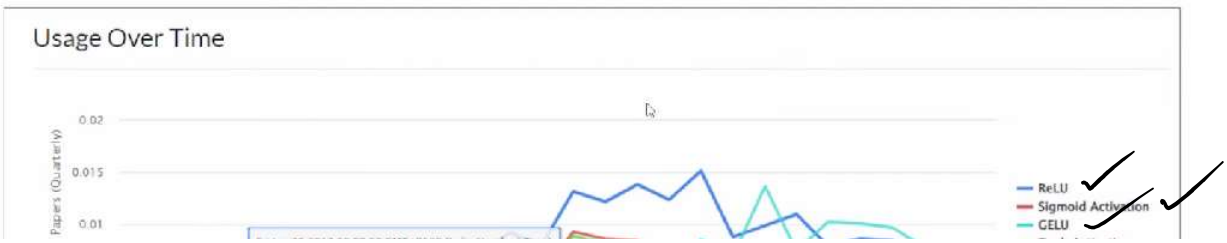
Named as SWISH

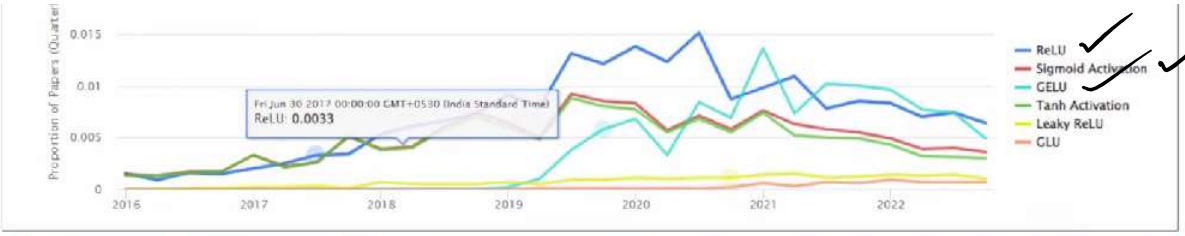
$\beta = 1.702$ or a learnable parameter (GELU)

$x\sigma(\beta x)$: SWISH

$x\sigma(1.702x)$: GELU

$x\sigma(x)$: SILU (Sigmoid-weighted Linear Unit)





Things to Remember

Sigmoids are bad

ReLU is more or less the standard unit for Convolutional Neural Networks

Can explore Leaky ReLU/Maxout/ELU

tanh sigmoids are still used in LSTMs/RNNs (we will see more on this later)

GELU is most commonly used in Transformer based architectures like BERT, GPT