Math powers - activate!

Learning activation function

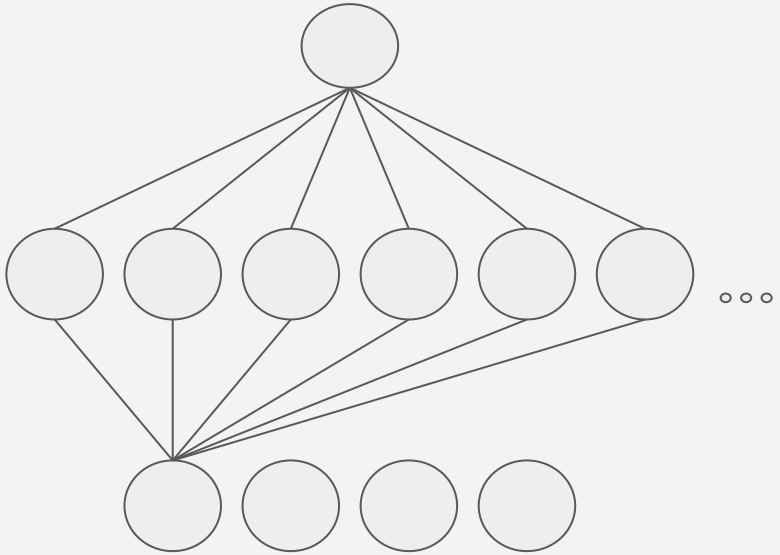
General approximation theorem

One hidden layer with non-polynomial activation functions can approximate any function given sufficient width

Activation is required for neural network to not collapse to simple matrix multiplication:

$$h = W_1 x + b_1$$
$$y = W_2 h + b_2$$
$$y = W_2(W_1 x + b_1) + b_2 = W_2 W_1 x + W_2 b_1 + b_2$$

General approximation via neural network



$$Y = \sigma(W_2\sigma(W_1x + b_1) + b_2)$$

$$Y = \frac{1}{1 + exp(-W_2\frac{1}{1+exp(-W_1x-b_1)} - b_2)}$$

But how can such non-linearities approximate relatively simple functions?

Experiment one: approximations given width

$$f(\vec{x}) = x_0^3 + x_1^2 - x_2 + 3x_3 - 2x_4$$

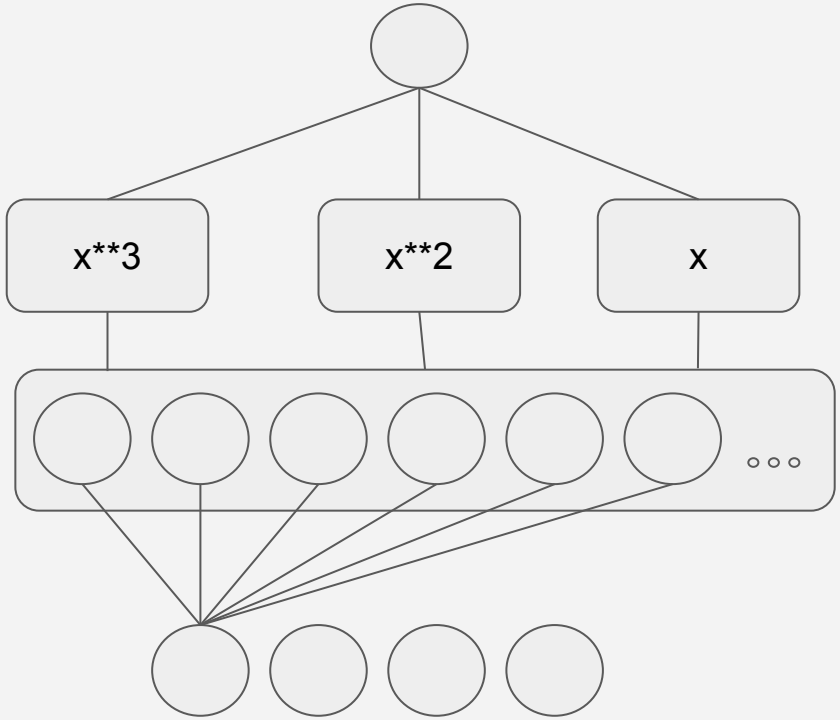
Using sigmoid activation

Number of hidden units	Epochs before termination	Validation MSE
5	40	0.027176
25	282	0.019558
100	640	0.019294
1000	2	1.632126
5	1000 (no stopping)	0.023175
1000	1000 (no stopping)	0.031281

Experiment one: approximations given width

$$f(\vec{x}) = x_0^3 + x_1^2 - x_2 + 3x_3 - 2x_4$$

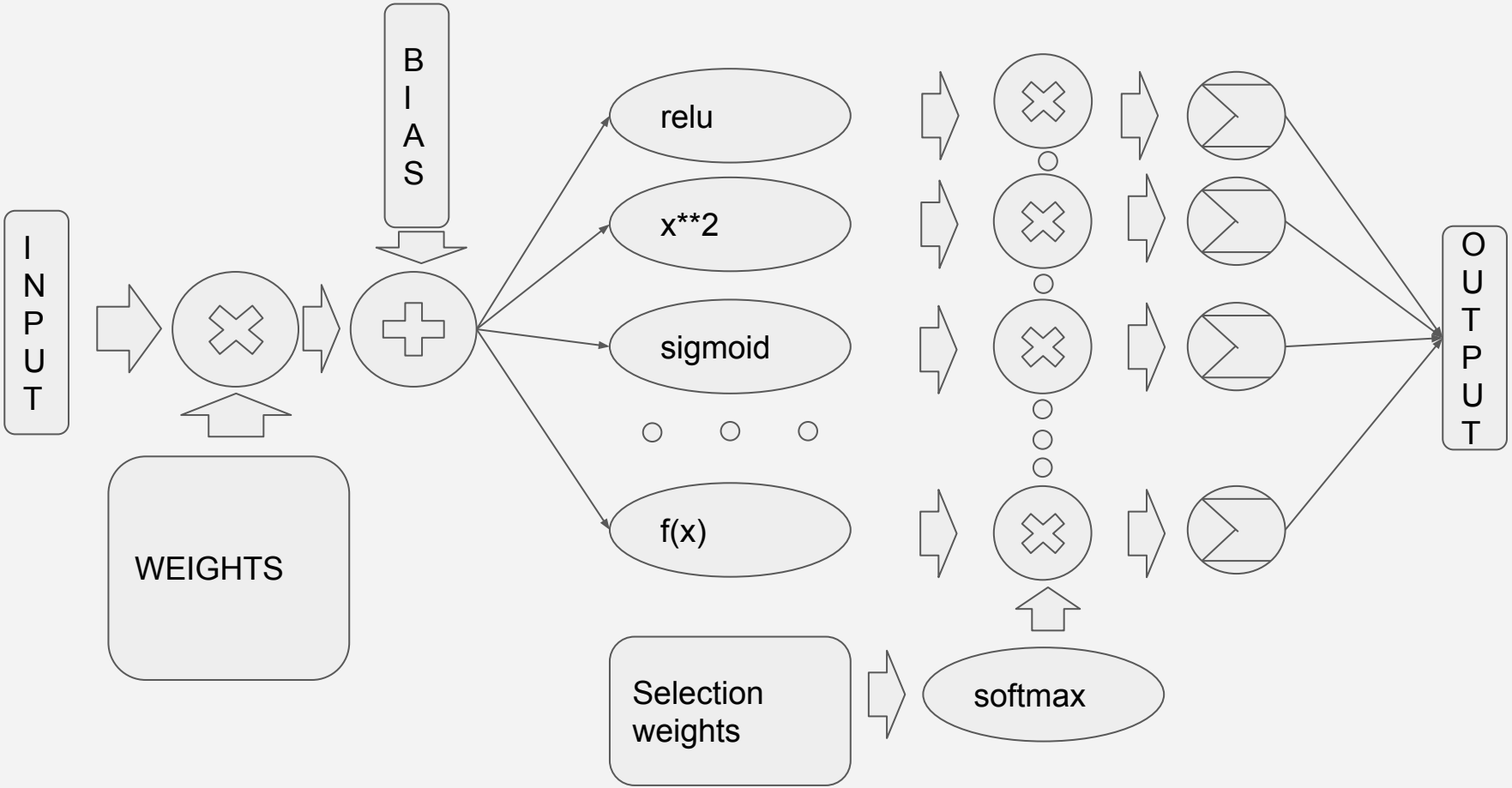
Number of hidden units	Epochs before termination	Validation MSE
4	15	0.018579
5	7	0.024485
25	11	0.030832
100	41	0.018022
4	1000 (no stopping)	5.562163e-05



So can we teach computer to figure things out?

```
>>> model.layers
[<keras.engine.topology.InputLayer at 0x7f5f79ebd320>,
 <keras.layers.core.Dense at 0x7f5f79ebd3c8>,
 <keras.layers.core.Lambda at 0x7f5f79ebd5c0>,
 <keras.layers.core.Lambda at 0x7f5f79ebd4a8>,
 <keras.layers.core.Lambda at 0x7f5f79ebd470>,
 <keras.layers.merge.Concatenate at 0x7f5f79ebd518>,
 <keras.layers.core.Dense at 0x7f5f79ebd4e0>]
>>> model.layers[1].get_weights()
[array([[ 0.6324068 ,  0.21015163, -0.797554  ,  0.2908216 ],
        [-0.39017132, -0.17748564, -0.8185287 , -0.30828226],
        [-0.08740603, -0.65033466, -0.01853351,  0.02315693],
        [-0.08319189, -0.6517701 , -0.02518974,  0.02785753],
        [ 0.01850492,  0.01924918,  0.0201706 , -1.0621451 ]],
       dtype=float32),
 array([-1.2548941 , -0.25056088, -0.08851553, -0.90485483], dtype=float32)]
>>> model.layers[-1].get_weights()
[array([[-0.9552365 ], [ 0.10199025],
        [-0.1944016 ], [-0.25859004],
        [-0.98726046], [-0.5750056 ],
        [ 0.3844548 ], [-1.1067214 ],
        [ 0.36572674], [-0.16890422],
        [-0.15650085], [ 0.33446547]], dtype=float32), array([1.085738],
dtype=float32)]
```

Activation learning layer - description



Experimental evaluation - MNIST

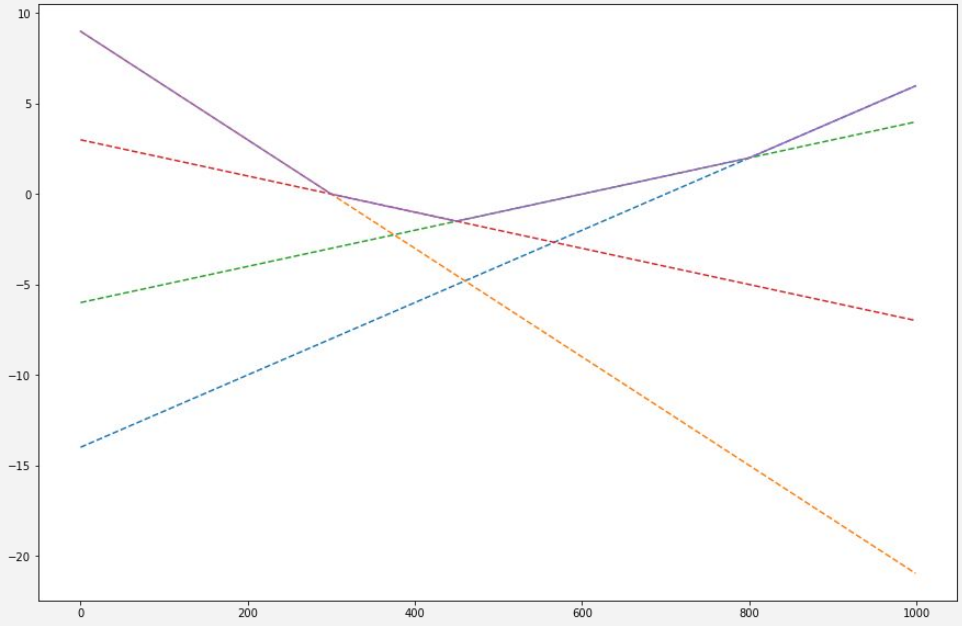
Evaluation on the most overused dataset in the world.

- Softmax
- Dense
- Activation of choice
- Dense
- Activation of choice
- Conv2D with strides
- Input

	ReLU	Adaptive
Accuracy	0.98036	0.97345
Final loss	0.00302	0.00412
Epochs	7.0	4.4

Maxout

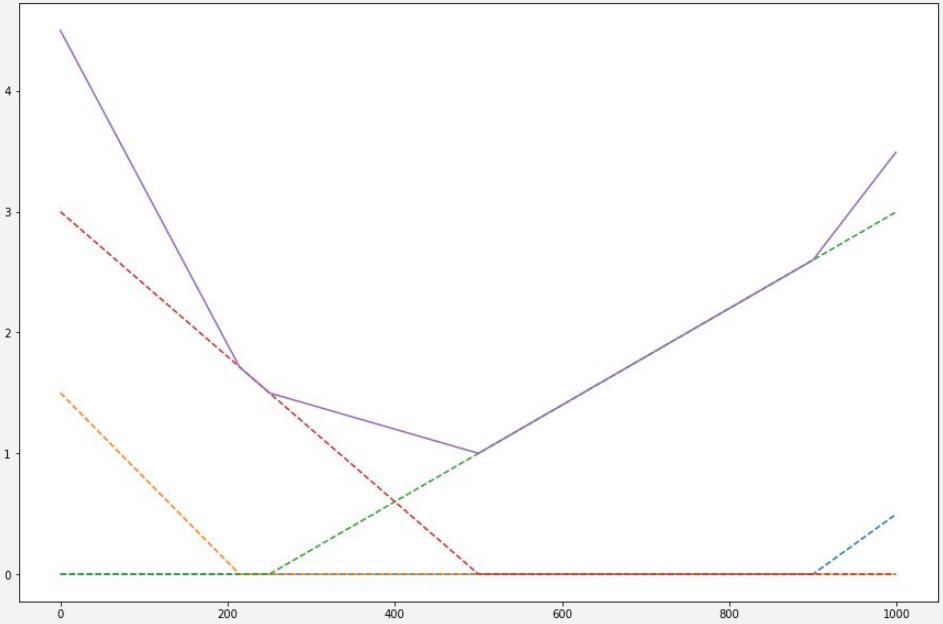
$$f(\vec{x}) = \max_{i \in I} a_i \odot \vec{x} + b_i$$



	ReLU	Maxout (1 spl)
Accuracy	0.98036	0.9743
Final loss	0.00302	0.00387
Epochs	7.0	6.0

Adaptive piecewise linear units

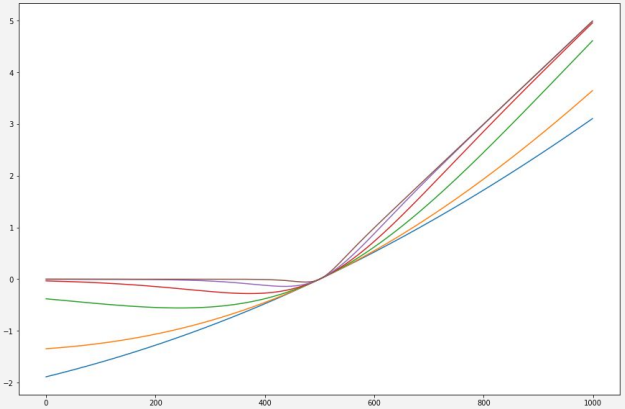
$$f(\vec{x}) = ReLU(\vec{x}) + \sum_{i \in S} ReLU(\vec{a_i} \odot \vec{x} + \vec{b_i})$$



	ReLU	APL (1 sample)
Accuracy	0.98036	0.9798
Final loss	0.00302	0.00305
Epochs	7.0	6

Swish activation

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



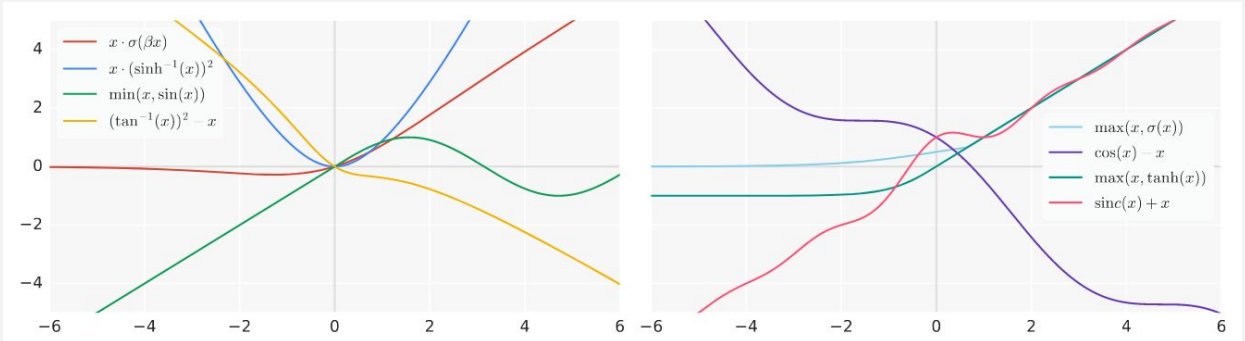


Figure 3: The top novel activation functions found by the searches. Separated into two diagrams for visual clarity. Best viewed in color.

Conclusions

-> It enables better representation
-> but in majority of the cases is impractical
-> could make use if we want to test some transforms of the input variables