Activation Functions Explained

POSTECH CVLab 김승욱

Main ref: Papers with Code
Sub ref: stackoverflow, blogs, etc,
https://mlfromscratch.com/activation-functions-explained/#/

Why do we need activation functions?

The purpose of an activation function is to
introduce non-linearity into the output of a neuron

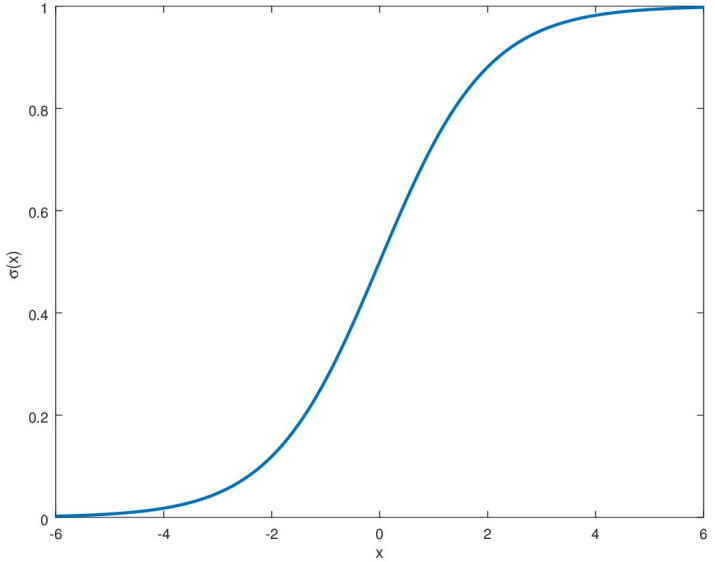
A neural network without an activation function is essentially just a **linear regression model** (this week's seminar on representation learning), no matter how many hidden layers we attach in the neural net.

This presentation covers:

- Sigmoid
- Tanh
- ReLU
- Leaky ReLU
- ELU
- PReLU
- Swish
- Hard Swish
- FReLU

Sigmoid

- Exists between 0 to 1.
- Especially used for models where we have to predict the probability of an output (ex Binary Classification)



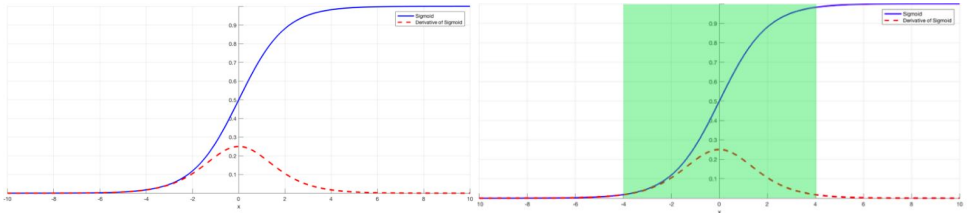
Advantages:

- Easy to understand (used mostly in shallow networks)
- Doesn't blow up activation

Known disadvantages:

- Gradient vanishing problem

-> squishes a large input space to a small space between 0 and 1. Large change in the input will cause a small change in the output - derivative becomes small.
-> n hidden layers -> n small derivatives multiplied together (chain rule)
-> gradient decreases exponentially as backprop down to initial layers
-> **BATCHNORM can solve this problem** by normalizing the input so that |x| does not reach the outer edges of the sigmoid function.



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

Sigmoid

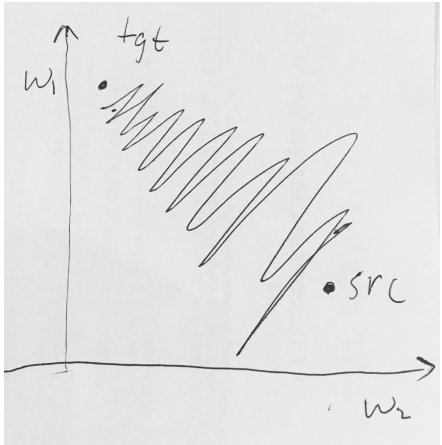
Known disadvantages (continued):

- Non-zero centered

-> Output after applying sigmoid is always positive

-> During gradient descent, the gradient on the weights during backprop will ALWAYS be either positive **or** negative.

-> The gradient updates go too far in different directions which makes optimization harder. **Can only change direction by ZIGZAGGING**



can also be solved with **batchnorm**, which normalizes the data to be 0-centered.

Batch normalization + Sigmoid seems to solve all the problems of sigmoid?

:: reference) Then why is ReLU so widespread?

ReLU was studied before BatchNorm was introduced. Now with the right tricks, the comparison between Sigmoid and ReLU is less clear

-> ReLU empirically worked well before “historically”, and continues to work well now -> a safe default option.

-> ReLU is faster to compute, as well as its derivative.

-> **Empirically**, ReLU requires relaxed tricks to make the network train, whereas sigmoid is more fiddly (more fragile)

-> Good enough: In many domains, other activations are no better/only slightly better than ReLU.

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

Tanh (Hyperbolic Tangent)

- Exists between -1 to 1.
- **Provides stronger gradients** (steeper)
  -> Faster convergence
- Zero-centered

Known disadvantages:

- Gradient vanishing problem (still exists!)

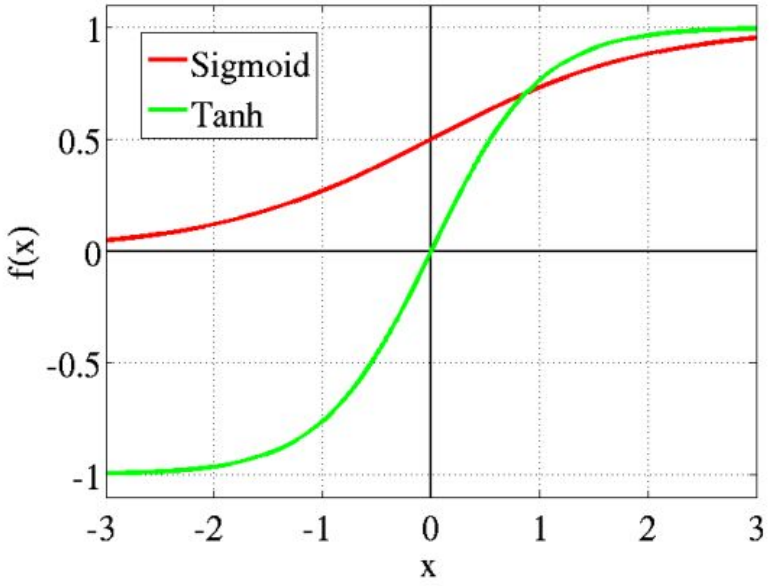
-> squishes a large input space to a small space between **-1** and 1.

Model sparsity: The less complex the model is during optimization, the faster it will converge, and the more likely it is to find a mathematical optimum in time.
Complexity: number of unimportant neurons in your model. The fewer of them, the better ("sparser") your model is.

- Sigmoid and Tanh produce non-sparse models

-> Their neurons pretty much ALWAYS produce an output value
-> output either cannot be zero, or is zero with very low probability.

Design choice: Do you need your gradients to be stronger?: Tanh
Even with batchnorm, stronger gradients near 0



$$tanh\,(x) = 2 \cdot \sigma\,(2x) - 1$$

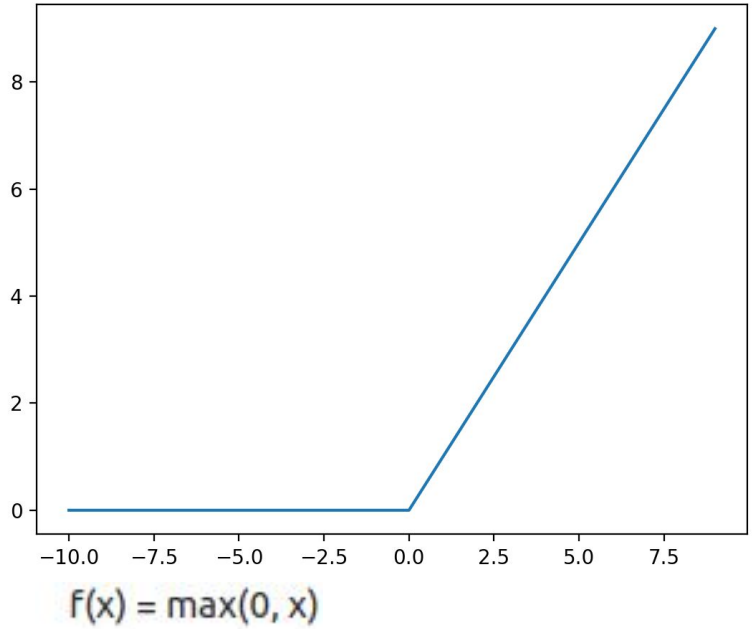
Sigmoidal units saturate across most of their domain-they saturate to a high value when z is very positive, saturate to a low value when z is very negative, and are **only strongly sensitive to their input** when z is near 0.

ReLU (Rectified Linear Unit)

Nearly linear, preserve many of the properties that make linear models easy to optimize with gradient-based methods. Also preserve many of the properties that make linear models generalize well.
The most preferred default Activation Function

Advantages:
- Computational Simplicity (faster)
- Representational sparsity: accelerate learning, simplify model
- Based on the principle that models are easier to optimize if their behaviour is closer to linear.

Known disadvantages:
- **Dying ReLU**
-> gradient is 0 whenever the unit is not active. This could lead to cases where a unit never activates as a gradient-based optimization algorithm will not adjust the weights of a unit that never activates initially
-> leaky ReLU allow small negatives values when the input is less than 0.
-> PReLU adaptively learns the parameters of rectifiers
- **Exploding output**
-> Infinite inputs produce infinite outputs
- **Others**
-> Small values (even non-positive ones) may capture patterns underlying the



ReLU (Rectified Linear Unit)

Known tips:

- A smaller bias input value

-> Conventional 1.0 to around 0.1 initialization

- “He weight initialization”

-> More suitable for ReLU and its extensions, unlike Xavier initialization

(Xavier initialization assumes that activations are linear)

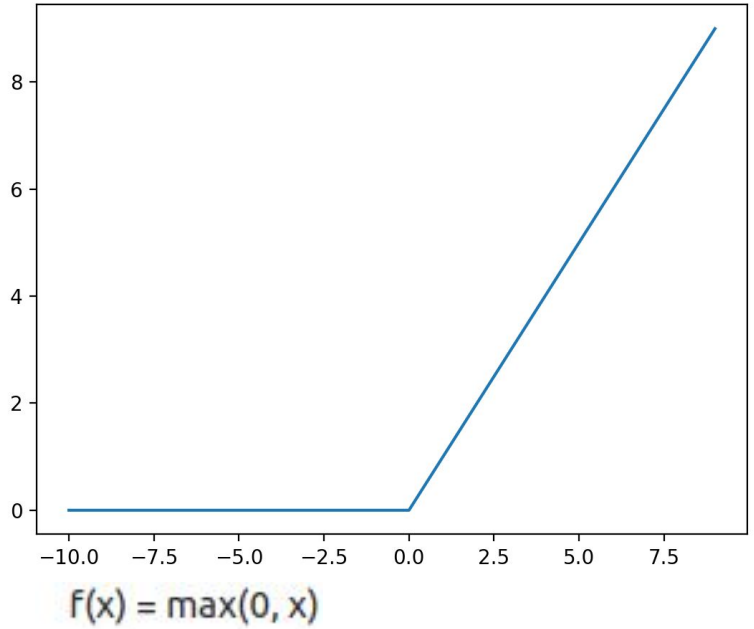
- Scale Input data (Batchnorm?)

-> Without data scaling, the weights of the neural network can grow large.

-> This makes the network unstable and increases the generalization error

- Using weight penalty

-> Since ReLU is unbounded in the positive domain.



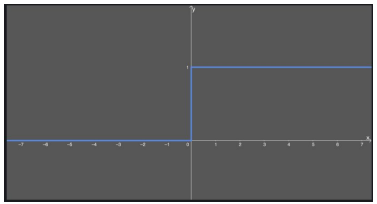
Leaky ReLU (YOLO, GAN…)

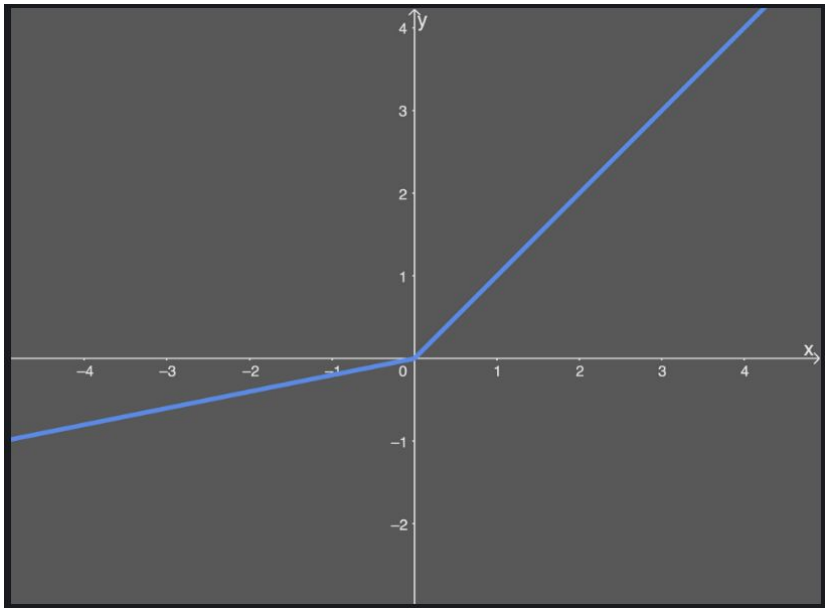
Alpha value typically between 0.1 and 0.3
Solves the **dead ReLU** problem
-> Values of gradients can no longer be stuck at 0

Still have to deal with exploding gradients,
the network does not learn the alpha value
-> There may be an optimal alpha value for this task

Negative outputs: helps the network nudge weights and biases in the right directions

Becomes a linear function when differentiated-> a con?





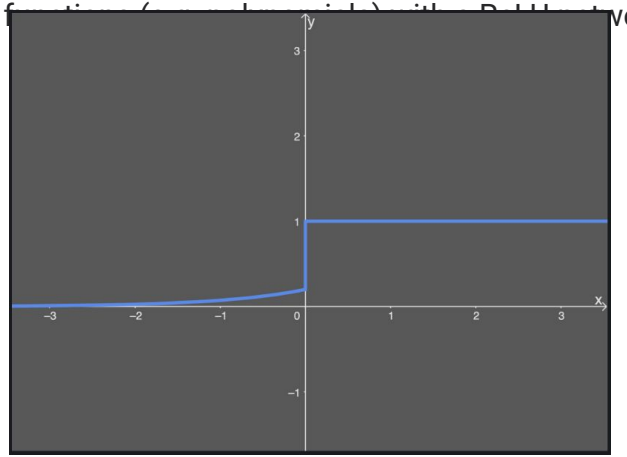
$$\mathrm{LReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases} \qquad \mathrm{LReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ \alpha & \text{if } x \leq 0 \end{cases}$$

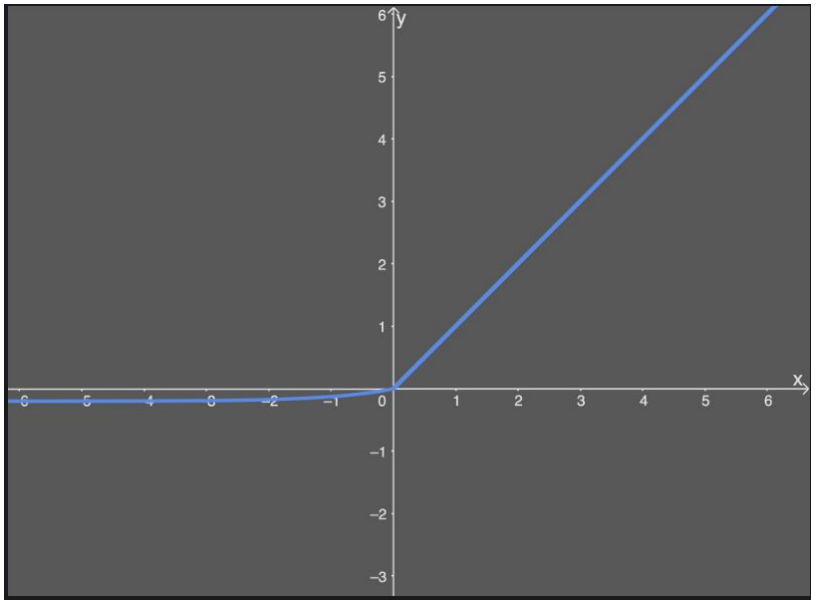
ELU (Not as famous)

Alpha value typically between 0.1 and 0.3

More computationally expensive due to the **exponential** operation
Still has nonlinearities(partly) when differentiated -> a pro?

“It is difficult to generate nonlinearity (in the sense of “curvature”) with (piecewise) linearity (e.g. ReLU, Leaky ReLU). Hence it is difficult to effectively approximate smooth f[illegible]work.”
-quora





$$\mathrm{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x < 0 \end{cases}$$

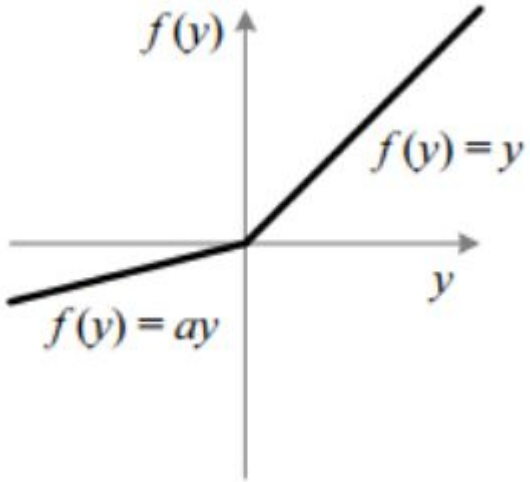
$$\mathrm{ELU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ \mathrm{ELU}(x) + \alpha & \text{if } x \leq 0 \end{cases}$$

PReLU (Kaiming He, 2015)

Alpha value is a **learnable parameter**
Number of slope parameters to be learned ==
Number of layers (or sum of all channels in every layer)
-> Negligible compared to number of weights/biases to be learned

alpha == 0 : ReLU
alpha(fixed) > 0 : Leaky ReLU
Learnable parameter: PReLU

a: usually initialized to 0.25



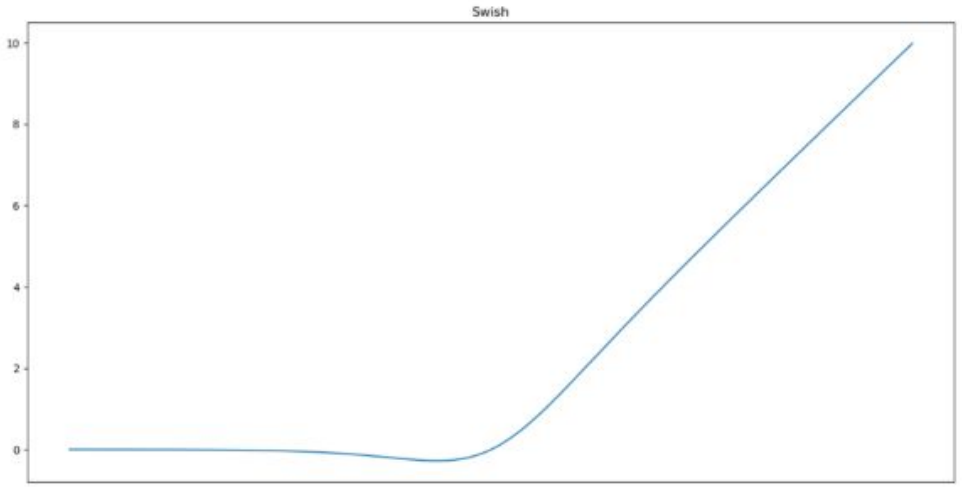
$$f(y_i) = \begin{cases} y_i, & \text{if } y_i > 0 \\ a_i y_i, & \text{if } y_i \leq 0 \end{cases}$$

Swish (Google Brain, EfficientNet/EfficientDet)

Smooth function, does not change direction abruptly like ReLU variations

Non-monotonic: Does not remain stable or move in one direction.
->authors write in their paper that it is in fact this property which separates Swish from most other activation functions

“extensive experiments show that **Swish consistently matches or outperforms ReLU on deep networks** applied to a variety of challenging domains such as image classification and machine translation”
-> Unlike other proposed variants, which were inconsistent



$$f(x) = x * sigmoid(x)$$
$$= x * (1 + e^{-x})^{-1}$$

Swish (Google Brain, EfficientNet/EfficientDet)

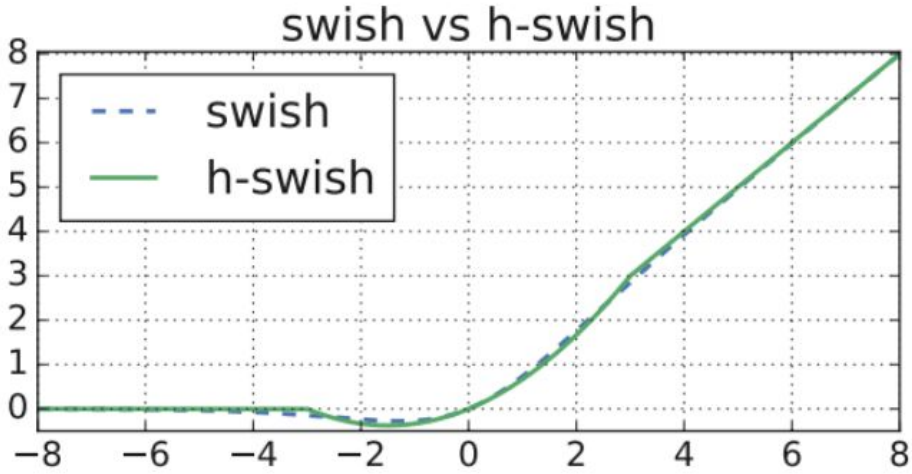
Authors' observations, and attempts to explain the consistently superior performance of Swish

- First, it is bounded below. Swish therefore benefits from sparsity similar to ReLU. Very negative weights are simply zeroed out.
- Second, it is unbounded above. This means that for very large values, the outputs do not saturate to the maximum value (i.e., to 1 for all the neurons). According to the authors of the Swish paper, this is what set ReLU apart from the more traditional activation functions.
- Third, separating Swish from ReLU, the fact that it is a smooth curve means that its output landscape will be smooth. This provides benefits when optimizing the model in terms of convergence towards the minimum loss.
- Fourth, small negative values are zeroed out in ReLU (since $f(x) = 0$ for $x < 0$). However, those negative values may still be relevant for capturing patterns underlying the data, whereas large negative values may be zeroed out (for reasons of sparsity, as we saw above). The smoothness property and the values of $f(x) < 0$ for $x \approx 0$ yield this benefit. This is a clear win over ReLU.

Hard Swish (Google Brain, *Searching for MobileNetV3, ICCV '19*)

Replaces the computationally expensive Sigmoid function with a piecewise linear analogue.

".. no discernible difference in accuracy, but multiple advantages from a deployment perspective."



$$\text{h-swish}(x) = x\frac{\text{ReLU6}(x+3)}{6}$$

ReLU6 :: ReLU upper-bounded at 6

Why 6? :: If you unbound the upper limit, you lose too many bits to the Q part of a Q.f number. Keeping the ReLUs bounded by 6 will let them take a max of 3 bits (upto 8) leaving 4/5 bits for .f

FReLU (Funnel ReLU, 2020)

“...extends ReLU and PReLU to a 2D activation by adding a negligible overhead of spatial condition.”

“The forms of ReLU and PReLU are
y = max(x, 0) and y = max(x, px) given p < 1, respectively,

while FReLU is in the form of y = max(x,T(x)),
where T(x) is the 2D spatial condition.

Moreover, the spatial condition achieves a pixel-wise modeling capacity in a simple way, capturing complicated visual layouts with regular convolutions.”

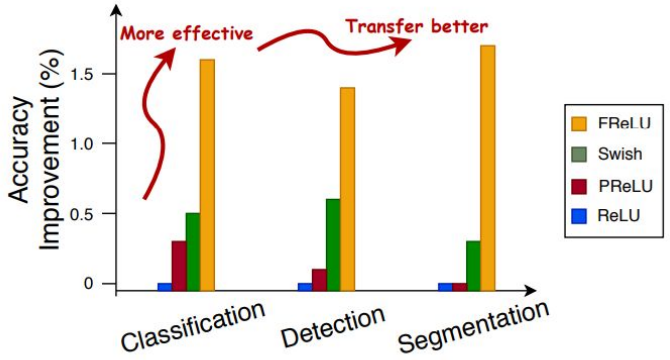


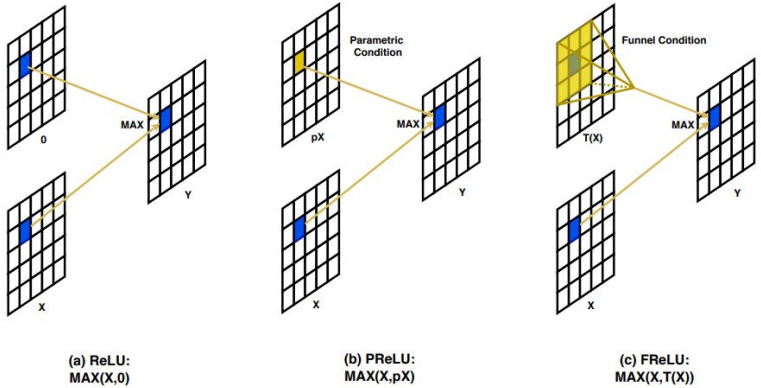
Fig. 1. Effectiveness and **generalization** performance. We set the ReLU network as the baseline, and show the *relative improvement* of accuracy on the three basic tasks in computer vision: image classification (Top-1 accuracy), object detection (mAP), and semantic segmentation (mean_IU). We use the ResNet-50 [15] as the backbone

FReLU (Funnel ReLU, 2020)

$$f(x_{c,i,j}) = max(x_{c,i,j}, \mathbb{T}(x_{c,i,j})) \tag{1}$$

$$\mathbb{T}(x_{c,i,j}) = x^{\omega}_{c,i,j} \cdot p^{\omega}_{c} \tag{2}$$

Here, $x_{c,i,j}$ is the input pixel of the non-linear activation $f(\cdot)$ on the c-th channel, at the 2-D spatial position (i, j); function $\mathbb{T}(\cdot)$ denotes the funnel condition, $x^{\omega}_{c,i,j}$ denotes a $k_h \times k_w$ **Parametric Pooling Window** centered on $x_{c,i,j}$, p^{ω}_{c} denotes the coefficient on this window which is shared in the same channel, and $(\cdot)$ denotes dot multiply.



FReLU (Funnel ReLU, 2020)

All the regions $x^{\omega}_{c,i,j}$ in the same channel share the same coefficient p^{ω}_{c}, therefore, it only adds a slight additional number of parameters. The region represented by $x^{\omega}_{c,i,j}$ is a sliding window, the size is default set to a 3×3 square, and we set the 2-D padding to be 1, in this case,

$$x^{\omega}_{c,i,j} \cdot p^{\omega}_{c} = \sum_{i-1\leq h\leq i+1, j-1\leq w\leq j+1} x_{c,h,w} \cdot p_{c,h,w} \tag{3}$$

As a result, the pixel-wise condition makes the network has a pixel-wise modeling capacity, the function $max(\cdot)$ gives per-pixel a choice between *looking at the spatial context or not*. Formally, consider a network $\{F_1, F_2, ..., F_n\}$ with n FReLU layers, each FReLU layer F_i has a $k \times k$ parametric window. For brevity, we only analyze the FReLU layers regardless of the convolution layers. Because the max selection between 1×1 and $k \times k$, each pixel after F_1 has a *activate filed* set $\{1, 1 + r\}$ $(r = k - 1)$. After the F_n layer, the set becomes $\{1, 1 + r, 1 + 2r, ..., 1 + nr\}$, which gives more choices to each pixel and can approximate any layouts if n is sufficiently large.

Seems to have more theoretical/implementation details, but haven't read through it yet

FReLU (Funnel ReLU, 2020)

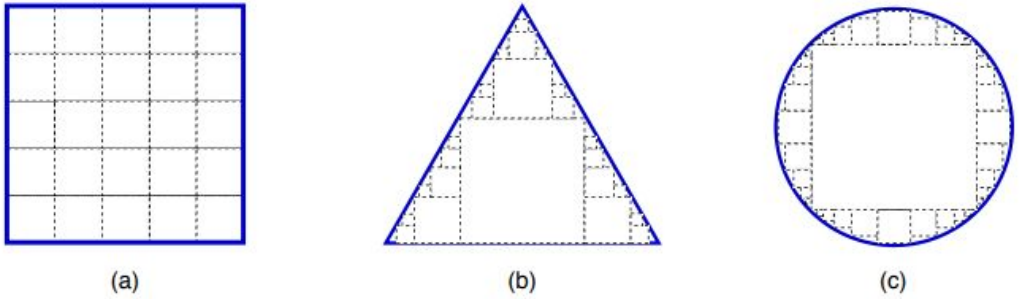


Fig. 3. Graphic depiction of how the per-pixel funnel condition can achieve *pixel-wise modeling capacity*. The distinct sizes of squares represent the distinct *activate fields* of each pixel in the top activation layers. (a) The normal activate field that has equal sizes of squares per-pixel, and can only describe the horizontal and vertical layouts. In contrast, the $max(\cdot)$ allows each pixel to choose *looking around or not* in each layer, after enough number of layers, they have many different sizes of squares. Therefore, the different sizes of squares can approximate (b) the shape of the oblique line, and (c) the shape of an arc, which are more common natural object layouts.

?

More activation functions

GeLU, SeLU, Maxout….

Conclusion (subjective)

- Sigmoid < Tanh
- ReLU is **faster** than Tanh
  - may vary in **performance** if tanh is used **with batchnorm** (always used nowadays)
- PReLU > Leaky ReLU > ReLU
- Hard Swish (faster) > Swish > PReLU
- FReLU > Hard Swish

If I were to experiment to choose my activation function:

- ReLU (baseline), Tanh, Hard Swish, PReLU, FReLU
- Total 5 choices