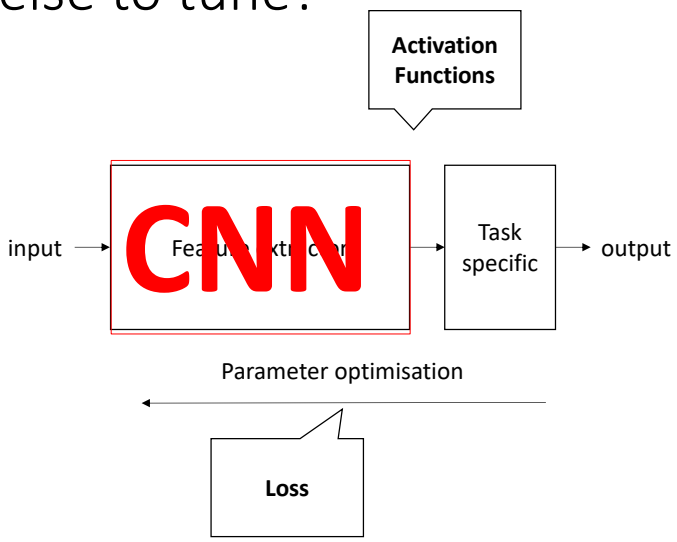
Deep Learning – Activation Functions

Bernhard Kainz

Let’s talk about activation functions

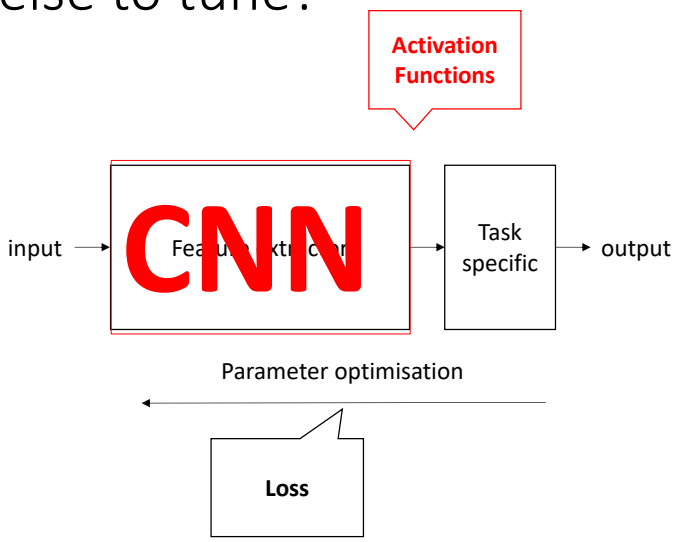
Where else to tune?



Deep Learning – Bernhard Kainz

Except network depth one can also tweak two other areas.
One is how the neuronal in network activate given a certain number of inputs
The other one is how you define the Error or Loss that's backpropagated.
A lot of research has gone into these, probably a bit more and more recently into losses and a bit less and more during the early days into activation functions.

Where else to tune?



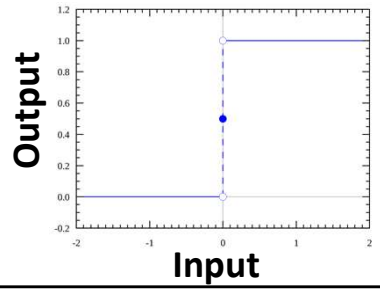
Except network depth one can also tweak two other areas.
One is how the neuronal in network activate given a certain number of inputs
The other one is how you define the Error or Loss that's backpropagated.
A lot of research has gone into these, probably a bit more and more recently into losses and a bit less and more during the early days into activation functions.

Activation functions

- Consider a neuron,

$$\text{Input} = \sum (w_i \cdot input) + b_i$$

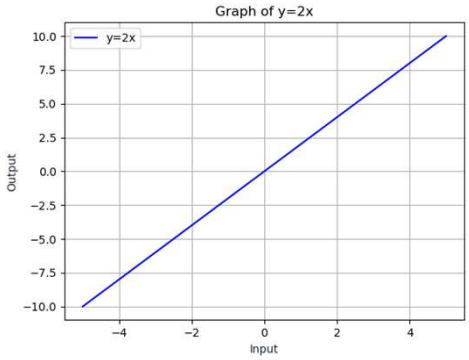
- Naïve activation:
  - If the value of Y is above a certain value, declare it activated.
  - Not differentiable, no backprop



Remember what does an artificial neuron does. Simply put, it calculates a "weighted sum" of its input, adds a bias and then decides whether it should be activated not.
Now, the value of Y can be anything ranging from -inf to +inf. The neuron really doesn't know the bounds of the value. So how do we decide whether the neuron should fire or not?
We decided to add "activation functions" for this purpose

Linear activation

- $Output = c \cdot x$
- Constant gradient, no relationship to x during backprop



Deep Learning – Bernhard Kainz

Ok, so we want something to give us intermediate (analog) activation values rather than saying “activated” or not (binary).
A straight line function where activation is proportional to input.
This way, it gives a range of activations, so it is not binary activation.
It is a constant gradient and the descent is going to be on constant gradient. If there is an error in prediction, the changes made by back propagation is constant and not depending on the change in input delta(x)

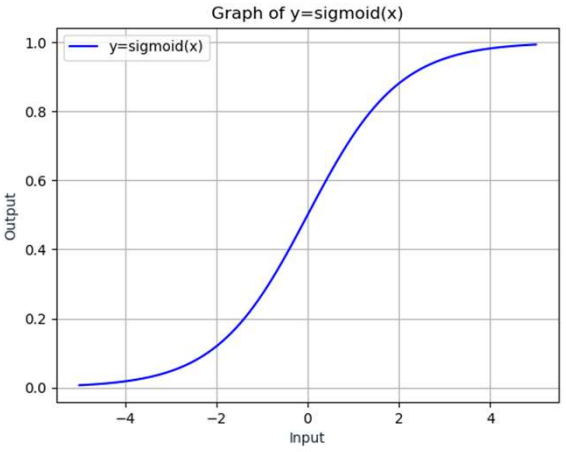
Even worse for fully connected layers. Each layer is activated by a linear function. That activation in turn goes into the next level as input and the second layer calculates weighted sum on that input and it in turn, fires based on another linear activation function.
No matter how many layers we have, if all are linear in nature, the final activation function of last layer is nothing but just a linear function of the input of first layer!

That means these N linearly activated layers can be replaced by a single layer.

No matter how we stack, the whole network is still equivalent to a single layer with linear activation

Sigmoid function

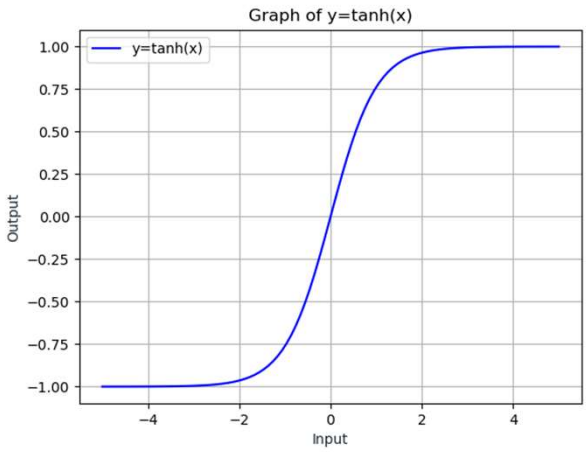
- $Output = \frac{1}{1+e^{-input}}$
- Nonlinear



looks smooth and “step function like” and you have seen this in previous ML courses.
it is nonlinear in nature., Great. Now we can stack layers.
What about non binary activations? It will give an analog activation unlike step function.
It has a smooth gradient too.
between X values -2 to 2, Y values are very steep. Which means, any small changes in the values of X in that region will cause values of Y to change significantly. That means this function has a tendency to bring the Y values to either end of the curve.
It tends to bring the activations to either side of the curve (above x = 2 and below x = -2 for example). Making clear distinctions on prediction.
towards either end of the sigmoid function, the Y values tend to respond very less to changes in X. What does that mean? The gradient at that region is going to be small. It gives rise to a problem of “vanishing gradients”.
This meant the network may refuse to learn further or is drastically slow.

tanh function

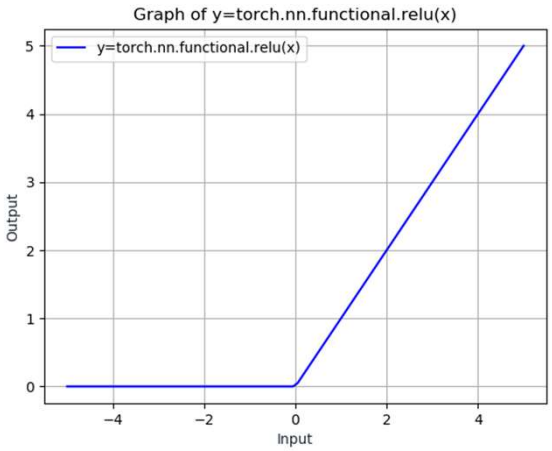
- $Output = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
- Scaled sigmoid



this has characteristics similar to sigmoid that we discussed above. It is nonlinear in nature, so great we can stack layers! It is bound to range (-1, 1) so no worries of activations blowing up.
the gradient is stronger for tanh than sigmoid (derivatives are steeper).
One advantage of tanh is that we can expect the output to be close to have a zero-mean. This has advantages for the weights that follow, because they see positive and negative values and therefore tend to converge faster.
Deciding between the sigmoid or tanh will depend on your requirement of gradient strength. Unfortunately if you stack a lot of these you cannot learn very efficiently. One also needs to be very careful about normalisation.

ReLU

- $Output = \max(0, x)$
- Efficient
- combinations of ReLU and ReLU are non linear
- Bound: [0, inf)
- dying ReLu problem

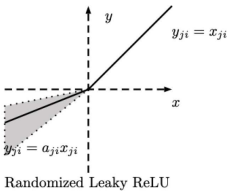


It gives an output x if x is positive and 0 otherwise.
At first look this would look like having the same problems of linear function, as it is linear in positive axis. First of all, ReLu is nonlinear in nature. And combinations of ReLu are also non linear!
it is a good approximator. Any function can be approximated with combinations of ReLu so this means we can stack layers.
It is not bound and the range of ReLu is [0, inf). This means it can blow up the activation.
the sparsity of the activation can also be a problem in networks. Imagine a big neural network with a lot of neurons. Using a sigmoid or tanh will cause almost all neurons to fire in an analog way.
hat means almost all activations will be processed to describe the output of a network. This is costly.
We would ideally want a few neurons in the network to not activate and thereby making the activations sparse and efficient.
RELU allows this.
Imagine a network with random initialized weights and almost 50% of the network yields 0 activation because of the characteristic of ReLu (output 0 for negative values of x). This means fewer neurons are firing (sparse activation) and the network is faster.
Because of the horizontal line in ReLu(for negative X), the gradient can go towards 0. For activations in that region of ReLu, gradient will be 0 because of which the weights will not get adjusted during descent. That means, those neurons which go into that state

will stop responding to variations in error/ input. This is called dying ReLu problem.
This problem can cause several neurons to just die and not respond making a substantial part of the network passive.

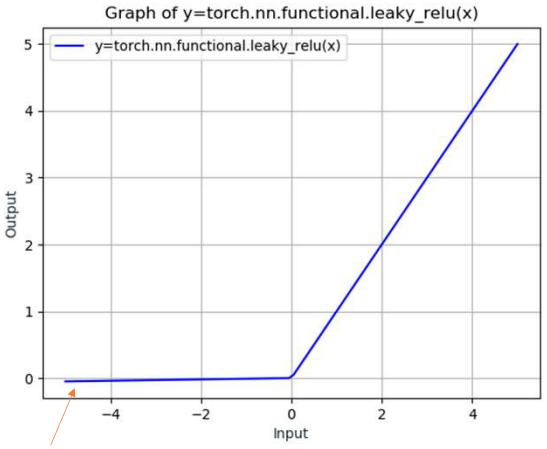
Leaky ReLU

- $Output = \begin{cases} x & for\ x \geq 0 \\ e.g.\ 0.01 \cdot x & for\ x < 0 \end{cases}$
- Mitigates dying ReLu



Randomized Leaky ReLU

atcold.github.io



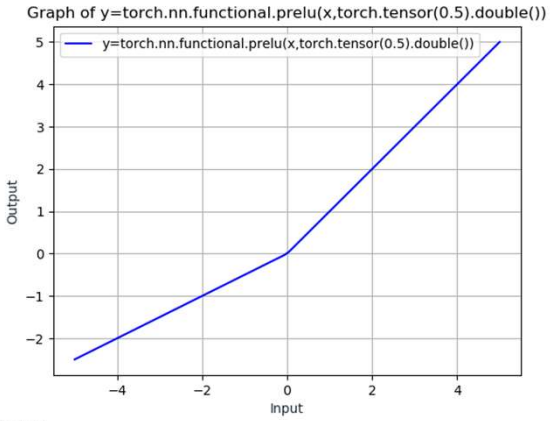
Deep Learning – Bernhard Kainz

There are variations in ReLu to mitigate this issue by simply making the horizontal line into non-horizontal component . for example y = 0.01x for x<0 will make it a slightly inclined line rather than horizontal line. This is leaky ReLu. There are other variations too. The main idea is to let the gradient be non zero and recover during training eventually.

ReLu is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations. That is a good point to consider when we are designing deep neural nets.

PReLU

- $Output = \begin{cases} x & for\ x \geq 0 \\ a \cdot x & for\ x < 0 \end{cases}$
- a is learnable
- Either one or a separate a is used for each input channel

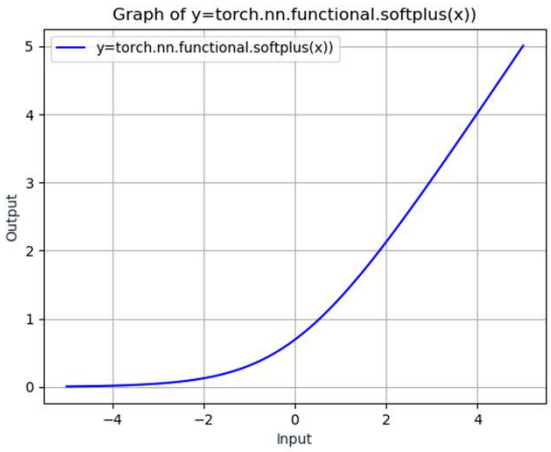


Here a is a learnable parameter. When called without arguments, pytorch uses a single parameter a across all input channels. If called with number of channels as input argument, a separate a is used for each input channel.

What's interesting about the ReLus is that they are scale invariant.
You can multiply the signal by a value and the output will not be changed, except the scale. So these are equivariant to scale. There is only one linearity.

SoftPlus

- $Output = \frac{1}{\beta} \cdot \log\left(1 + e^{(\beta \cdot x)}\right)$
- Smooth approximation of ReLU
- Output always positive
- Numerical stability:
  use linear if
  $(\beta \cdot x) > threshold$

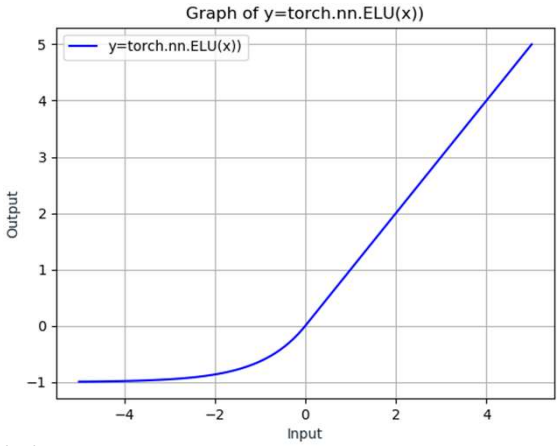


Functions like these here are affected by the amplitude of the input signal. They are non-linear throughout.

Softplus is a smooth approximation of the ReLU function. A differentiable version of ReLU.
It has a scale parameter beta. The higher beta, the more this function will look like a ReLU.
It can be used to constrain the output of a unit to always be positive.
For numerical stability the pytorch implementation reverts to the linear function when input times beta > a threshold

ELU

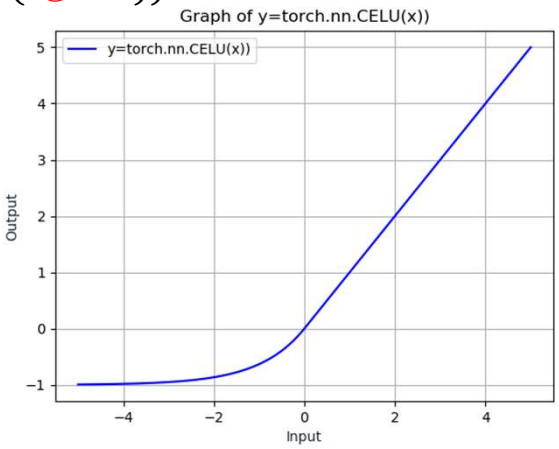
- $Output = \max(0, x) + \min(0, \alpha \cdot (e^x - 1))$
- Element-wise



Softplus like functions can be parametrized in various ways.
Another soft version of RELU. You use Relu as a basis and add a small constant that makes it smooth.
One difference is that this one here can become negative, unlike the RELu.
That may have an advantage but very much depends on the application it is used for.
Allows the network to make the average of the output zero, which can help convergence.

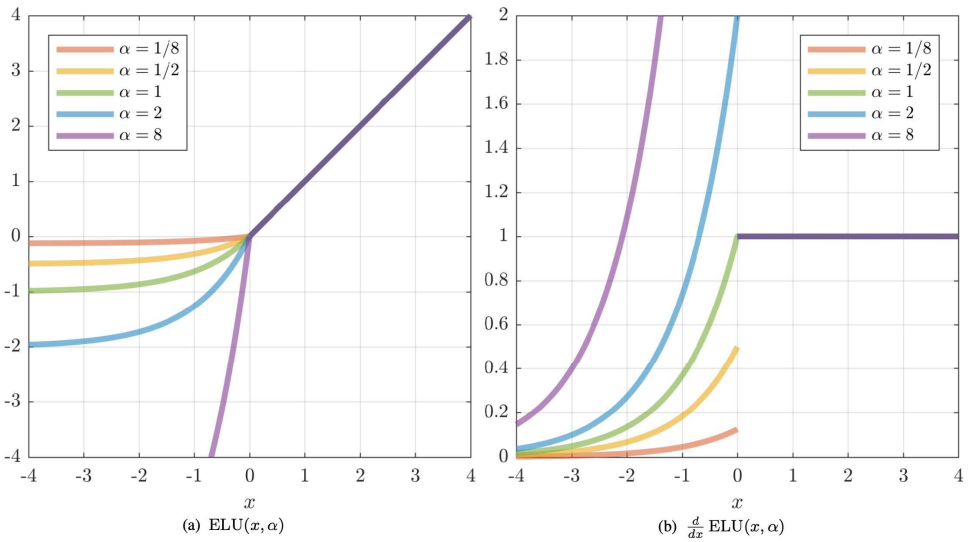
CELU

- $Output = \max(0, x) + \min(0, \alpha \cdot (e^{\frac{x}{\alpha}} - 1))$
- Element-wise
- Barron (2017) https://arxiv.org/abs/1704.07483



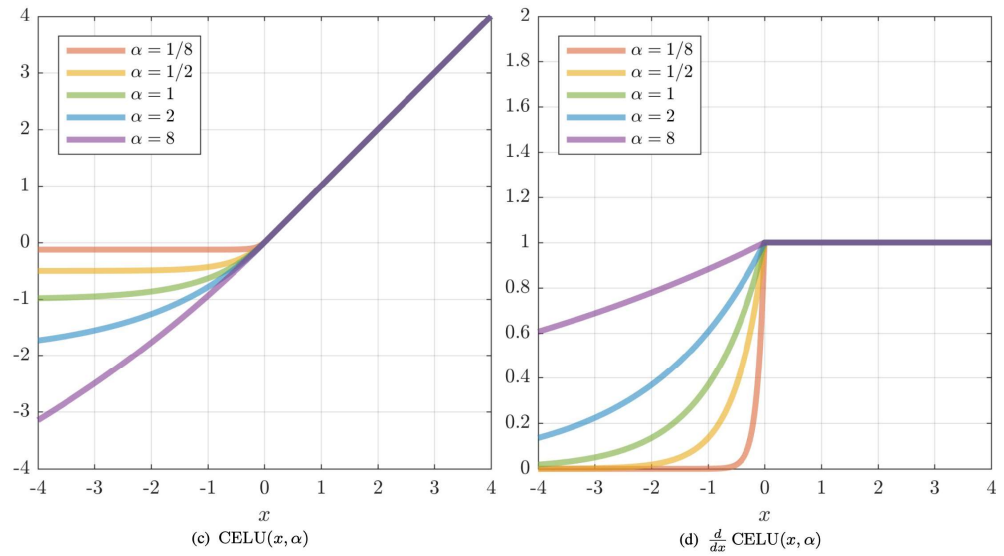
Same for CELU
This one is continuously differentiable if alpha is not equal to one. The CELU is C1 continuous.

CELU



(a) $\mathrm{ELU}(x, \alpha)$ (b) $\frac{d}{dx}\mathrm{ELU}(x, \alpha)$

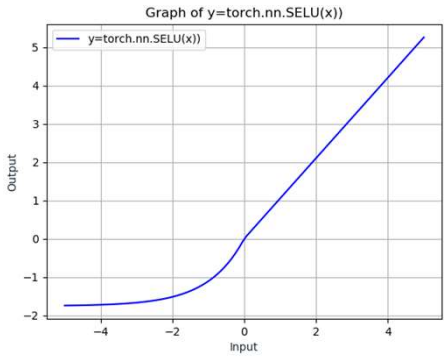
CELU



(c) CELU(x, α) (d) $\frac{d}{dx}$ CELU(x, α)

SELU

- $Output = scale \cdot (\max(0, x) + \min(0, \alpha \cdot (e^x - 1)))$
- with α = 1.6732632423543772848170429916717 and scale = 1.0507009873554804934193349852946



There a lot's of variations of this with many different parameters,
Scaled Exponential Linear Units have the advantage of internal normalization. The shown parameters are the solution to a fixed point problem where they assume mean of 0 and variance of 1 for all weights in a network.
When it comes to neural networks, normalization can happen in three different places.
First, there is input normalization. One example is to scale the pixel values of grey-scale pictures (0–255) to values between zero and one.
Second, and this time specifically relevant for neural networks, there is batch normalization.
Third, there is internal normalization, and this is where SELU's magic is happening. The main idea is that each layer preserves the mean and variance from the previous layer.
The activation function needs both positive and negative values for *y* to shift the mean.
A gradient very close to zero can be used to decrease the variance.
So the cause for vanishing gradients in other activation functions is a necessary characteristic for internal normalization.
So SELUs can never die.

SELU

- Klambauer 2017 et al.: https://arxiv.org/abs/1706.02515

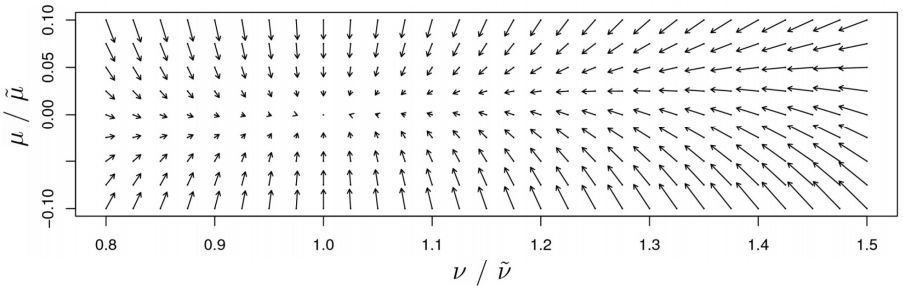
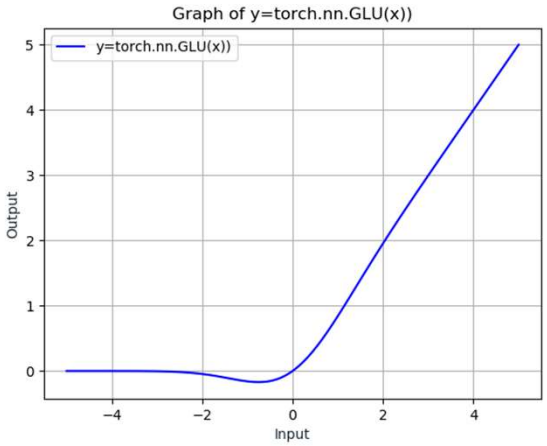


Figure 2: For $\omega = 0$ and $\tau = 1$, the mapping g of mean μ (x-axis) and variance ν (y-axis) to the next layer's mean $\tilde{\mu}$ and variance $\tilde{\nu}$ is depicted. Arrows show in which direction (μ, ν) is mapped by $g : (\mu, \nu) \mapsto (\tilde{\mu}, \tilde{\nu})$. The fixed point of the mapping g is $(0, 1)$.

GELU

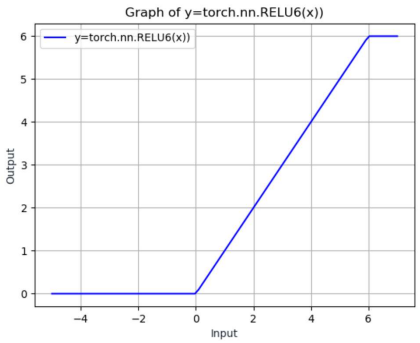
- $Output = x \cdot \Phi(x)$
- $\Phi(x)$ is the Cumulative Distribution Function for Gaussian Distribution.



Some for example can be related to Gaussian distributions. This is useful for regularization of the network.

ReLU6

- $Output = \min(\max(0, x), 6)$

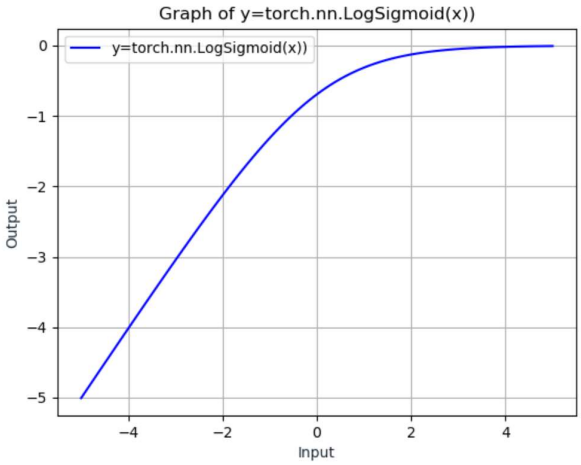


Deep Learning – Bernhard Kainz

RELu can also be made saturating
6 is a bit random, any parametrization is possible.
It looks a bit like a hard sigmoid or tanh function

LogSigmoid

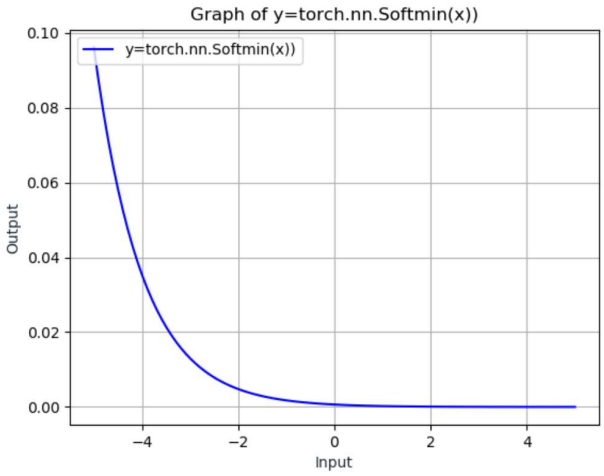
- $Output = \log(\frac{1}{1+e^{-x}})$
- Element-wise



Logsigmoid is mostly used in cost functions, not really as activation function
It's a useful function to have in a loss function and we'll see later on how this works.

Softmin

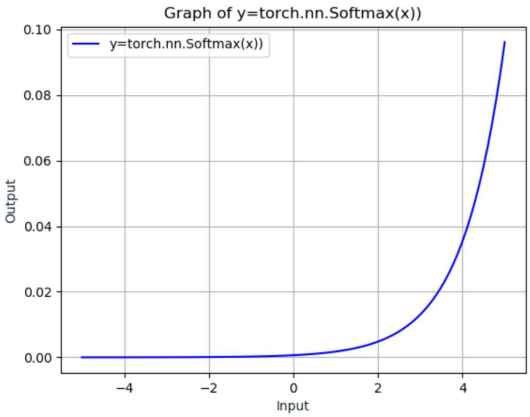
- $Output = \frac{e^{-x_i}}{\sum_j e^{-x_j}}$
- Applies the Softmin function to an n-dimensional input Tensor rescaling them so that the elements of the n-dimensional output Tensor lie in the range [0,1] and sum up to 1.



These are multi-dimensional non-linearities. Vector n in vector n out. In these the x can be seen as Energies or penalties instead of scores if you like. These are a good way to turn a couple of numbers that look a bit like a probability distribution.

Softmax

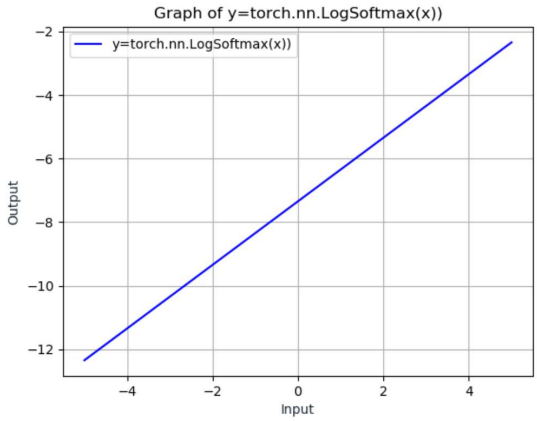
- $Output = \frac{e^{x_i}}{\sum_j e^{x_j}}$
- Applies the Softmax function to an n-dimensional input Tensor rescaling them so that the elements of the n-dimensional output Tensor lie in the range [0,1] and sum up to 1.



This one is probably the most common one, you might have already come across. Very commonly used to convert network logits to final class probability scores.

LogSoftmax

- $Output = \log(\frac{e^{x_i}}{\sum_j e^{x_j}})$
- Applies the log(Softmax) function to an n-dimensional input Tensor

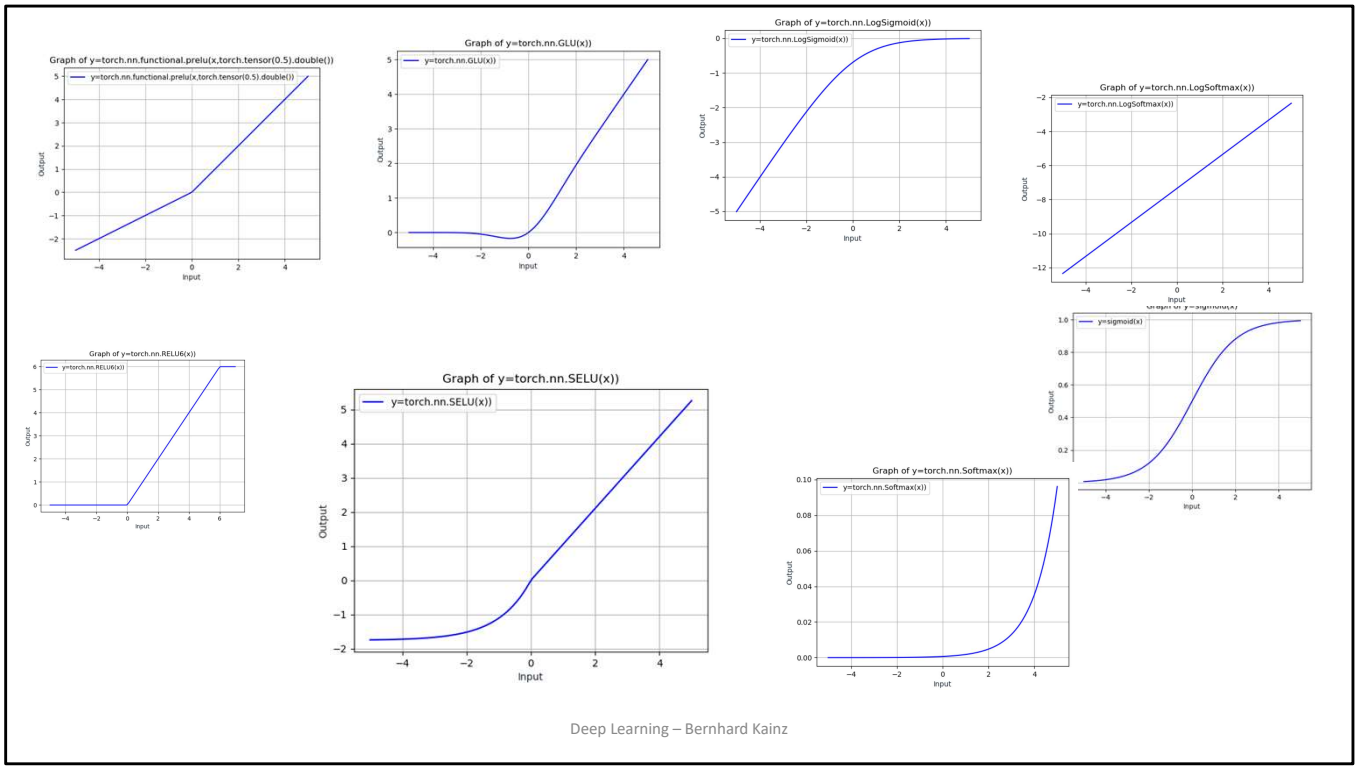


There is also a log version of softmax, This can also be useful as a piece of a loss function.

Periodic activations, SIREN

- Difficult convergence properties for general problems
- Has been used for implicit representations (find a continuous function that represents sparse input data, e.g. an image)
- https://vsitzmann.github.io/siren/

$$\Phi(\mathbf{x}) = \mathbf{W}_n \left(\phi_{n-1} \circ \phi_{n-2} \circ \ldots \circ \phi_0\right)(\mathbf{x}) + \mathbf{b}_n, \quad \mathbf{x}_i \mapsto \phi_i(\mathbf{x}_i) = \sin(\mathbf{W}_i \mathbf{x}_i + \mathbf{b}_i).$$



Now, which activation functions to use. The answer is simple… (42)
Does that mean we just use ReLu for everything we do? Or sigmoid or tanh? Well, yes and no. When you know the function you are trying to approximate has certain characteristics, you can choose an activation function which will approximate the function faster leading to faster training process. For example, a sigmoid works well for a classifier (see the graph of sigmoid, doesn't it show the properties of an ideal classifier?) because approximating a classifier function as combinations of sigmoid is easier than maybe ReLu, for example. Which will lead to faster training process and convergence. You can use your own custom functions too!. If you don't know the nature of the function you are trying to learn, then maybe i would suggest start with ReLu, and then work backwards. ReLu works most of the time as a general approximator!

What do we learn from this?

- Which function to use depends on the nature of the targeted problem.
- Most often you will be fine with ReLUs for classification problems. If the network does not converge, use leakyReLUs or PReLUs, etc.
- Tanh is quite ok for regression and continuous reconstruction problems.
- The representative power of you training set will usually outweigh the contribution of a smartly chosen activation function.