Pytorch Activation Implementation

Pytorch Activation Function Implementations

Open in Colab

Almost all activations in pytorch:

```
...
from .activation import Threshold, ReLU, Hardtanh, ReLU6, Sigmoid, Tanh, \
    Softmax, Softmax2d, LogSoftmax, ELU, SELU, CELU, GELU, Hardshrink, LeakyReLU, \
    LogSigmoid, Softplus, Softshrink, MultiheadAttention, PReLU, Softsign, Softmin, \
    Tanhshrink, RReLU, GLU, Hardsigmoid, Hardswish, SiLU, Mish
...
from .adaptive import AdaptiveLogSoftmaxWithLoss
...
```

Like given source code, there are 30 activation modules in pytorch.

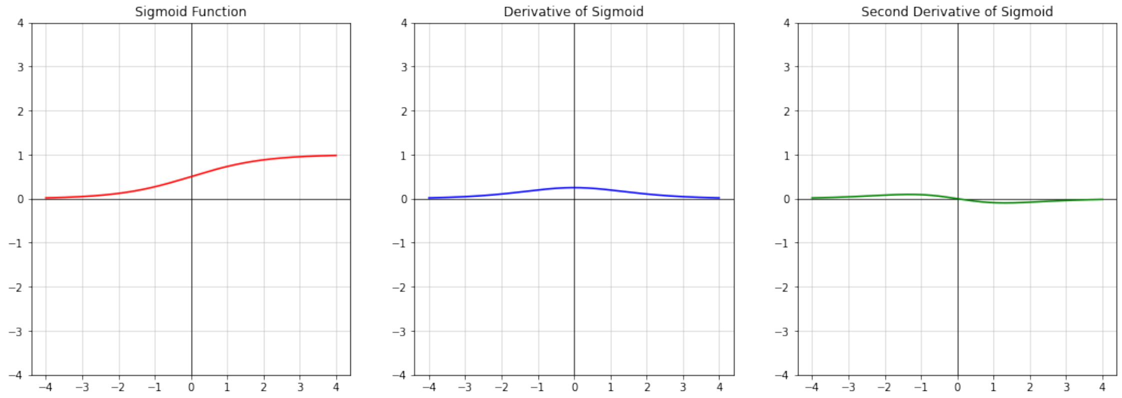
In this document, almost all activations are implemented except below two:

- AdaptiveLogSoftmaxWithLoss
  - from <Efficient softmax approximation for GPUs> by Edouard Grave, Armand Joulin, Moustapha Cissé, David Grangier, and Hervé Jégou
  - see `torch.nn.modules.adaptive`
- MultiheadAttention
  - from <Attention Is All You Need>
  - used for Transformers

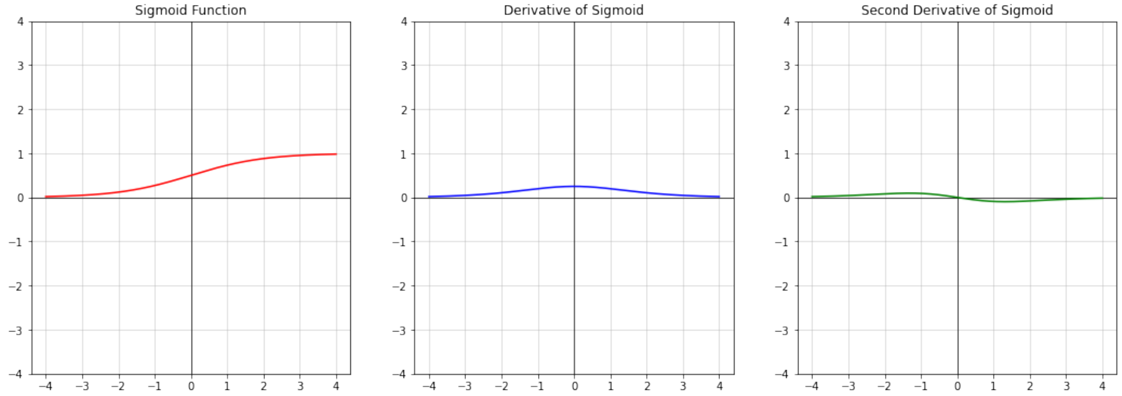
1 Sigmoid

$$\text{Sigmoid}(x) = \sigma(x) = \frac{1}{1 + \exp(-x)}$$

```
@plot_activation_module()
class Sigmoid(ActivationModule):

    def forward(self, x):
        return 1. / (1. + torch.exp(-x))
```



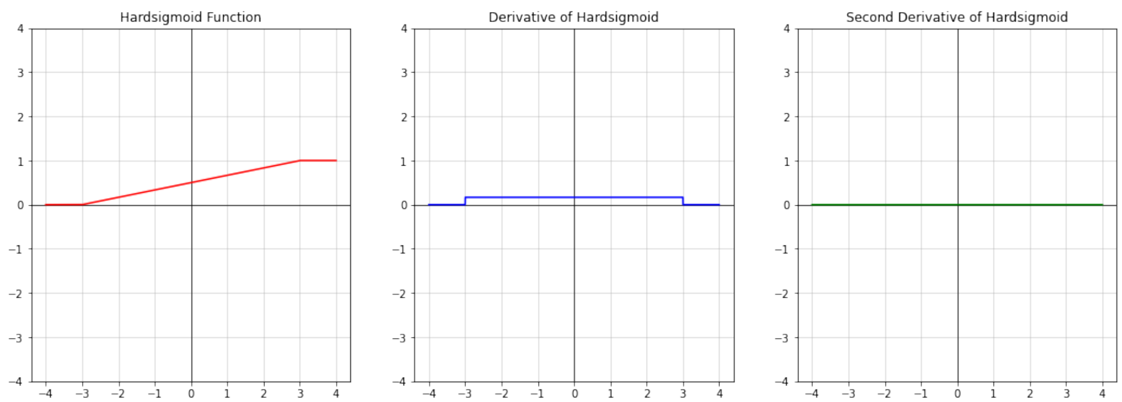

```
@plot_activation_function
class SigmoidFunction(torch.autograd.Function):

    @staticmethod
    def forward(ctx, input):
        output = 1. / (1. + input.neg().exp())
        ctx.save_for_backward(output)
        return output

    @staticmethod
    def backward(ctx, grad_output):
        output, = ctx.saved_tensors
        return grad_output * output.sub(1.).neg().mul(output)
```



2 Hardsigmoid

$$\text{Hardsigmoid}(x) = \begin{cases} 0 & \text{if } x \le -3, \\ 1 & \text{if } x \ge +3, \\ x/6 + 1/2 & \text{otherwise} \end{cases}$$

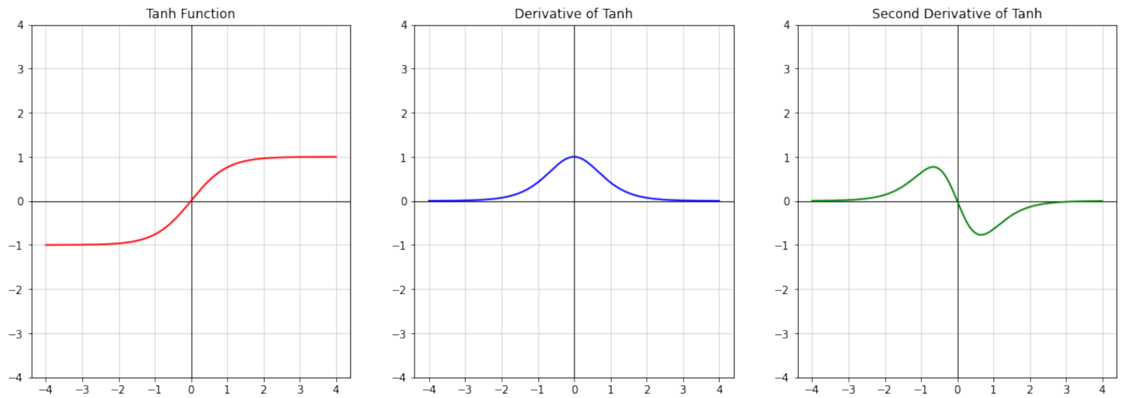
```
@plot_activation_module()
class Hardsigmoid(ActivationModule):

    @staticmethod
    def forward(x):
        return (x / 6. + .5).clamp(0., 1.)
```



3 Tanh

$$\text{Tanh}(x) = \tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

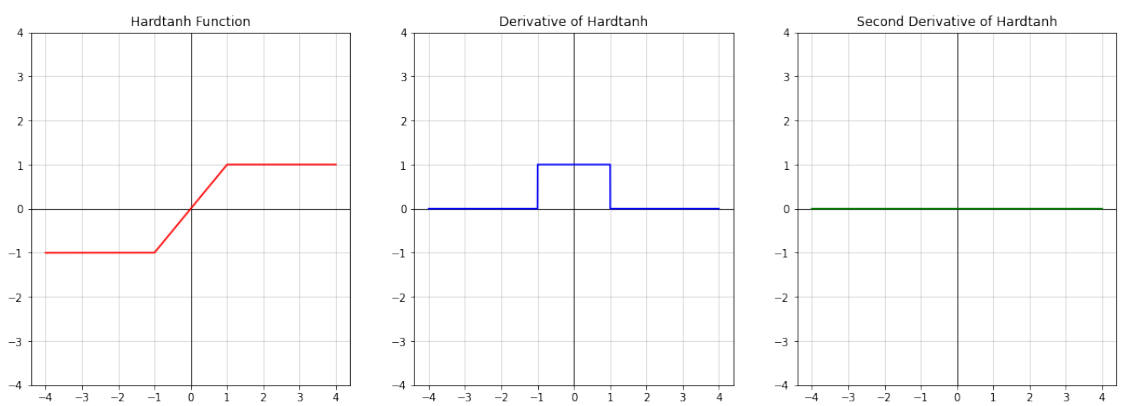
```
@plot_activation_module()  # basic activation for RNN / LSTM
class Tanh(ActivationModule):

    def forward(self, x):
        # return torch.sigmoid(2. * x) * 2. - 1.
        pos = torch.exp(x)
        neg = torch.exp(-x)
        return (pos - neg) / (pos + neg)
```



4 Hardtanh

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

```
@plot_activation_module()
class Hardtanh(ActivationModule):

    def forward(self, x):
        return x.clamp(-1., 1.)
```

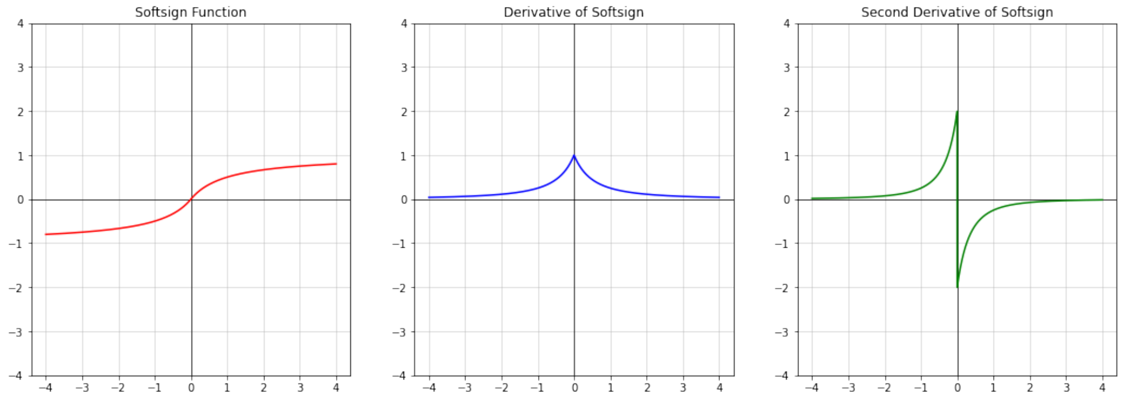


5 Softsign

$$\text{SoftSign}(x) = \frac{x}{1 + |x|}$$

```
@plot_activation_module()
class Softsign(ActivationModule):
```

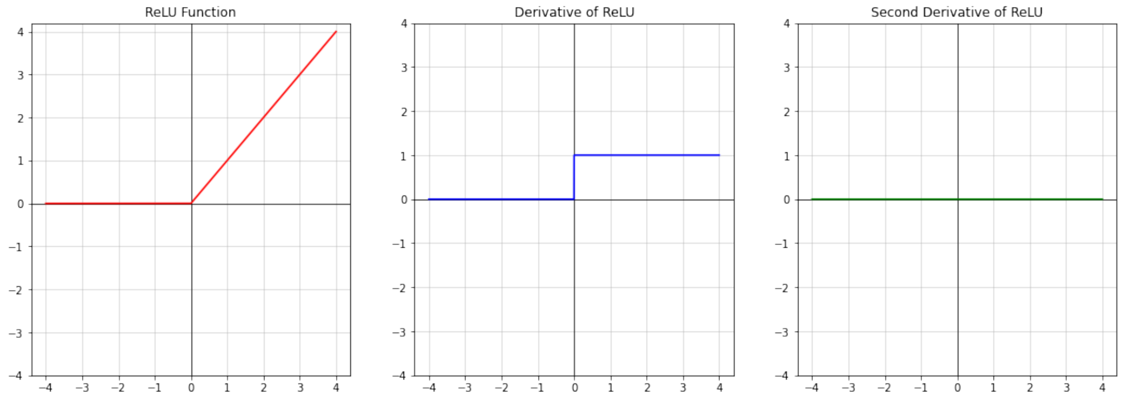
```
    def forward(self, x):
        return x / (1 + torch.abs(x))
```



6 ReLU

$$\mathrm{ReLU}(x) = (x)^{+} = \max(0, x)$$

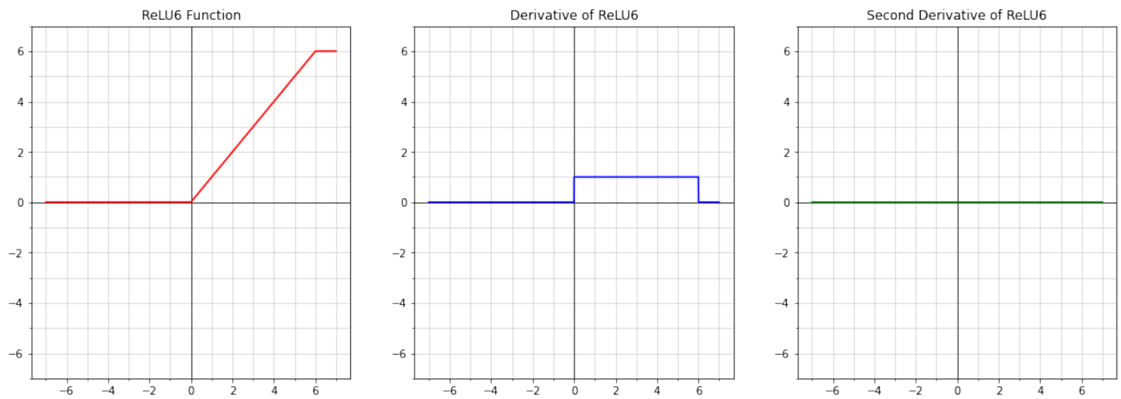
```
@plot_activation_module()
class ReLU(ActivationModule):  # Rectified Linear Unit

    def forward(self, x):  # zeros_like: zero-filled tensor which has same shape with x
        return x.clamp(0.)
```



7 ReLU6

$$\mathrm{ReLU6}(x) = \min(\max(0, x), 6)$$

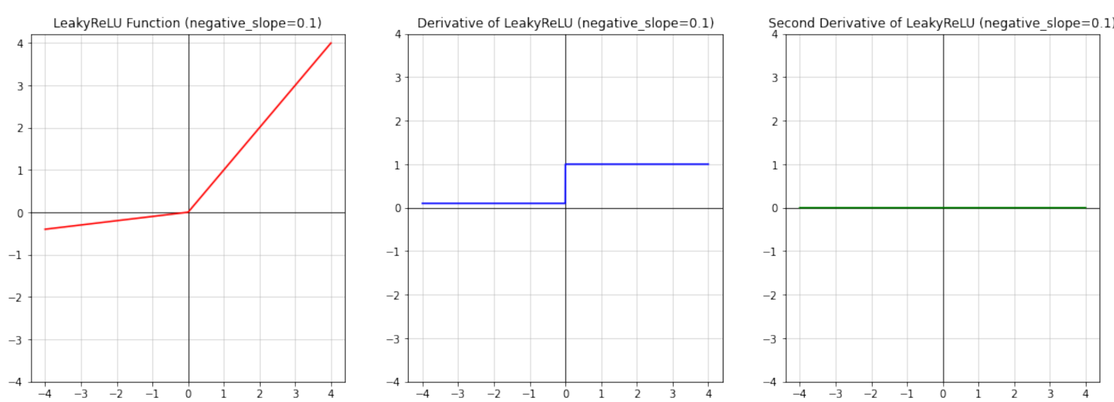
```python
@plot_activation_module(True)
class ReLU6(ActivationModule):   # Rectified Linear Unit

    def forward(self, x):
        return torch.clamp(x, 0., 6.)
```



8 LeakyReLU

$$\text{LeakyReLU}(x) = \max(0, x) + \text{negative\_slope} * \min(0, x)$$

```python
@plot_activation_module(negative_slope=1e-1)
class LeakyReLU(ActivationModule):   # Leaky - Rectified Linear Unit

    def __init__(self, negative_slope=1e-2):
        super().__init__()
        self.negative_slope = negative_slope

    def forward(self, x):
        return torch.where(x >= 0., x, x * self.negative_slope)
```

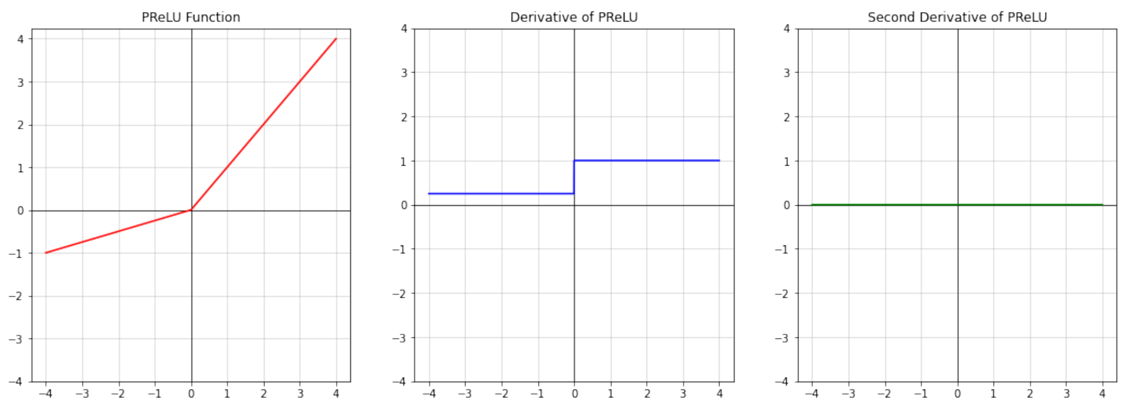


9 PReLU

$$\mathrm{PReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ ax, & \text{otherwise} \end{cases}$$

Here a is a learnable parameter.

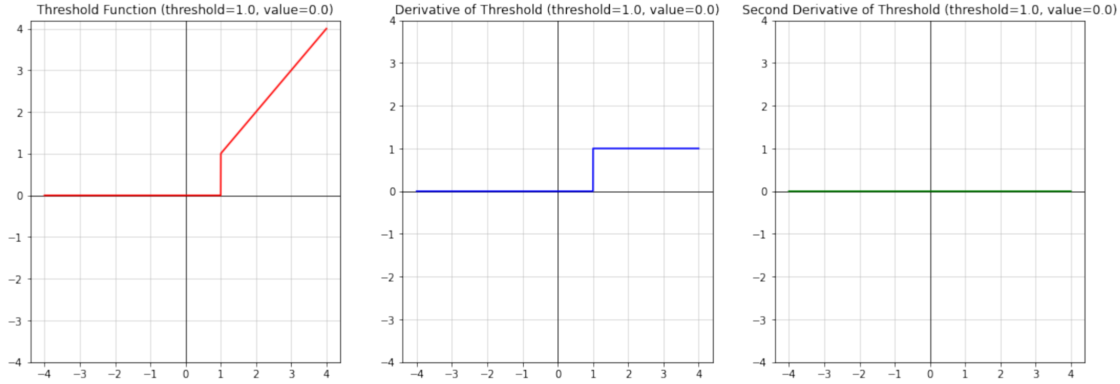
```
@plot_activation_module()
class PReLU(ActivationModule):   # Parametric Rectified Linear Unit

    def __init__(self, a=.25):
        super().__init__()
        self.weight = nn.Parameter(torch.tensor(a))

    def forward(self, x):
        return torch.where(x >= 0., x, x * self.weight)
```



10 Threshold

$$y = \begin{cases} x, & \text{if } x > \text{threshold} \\ \text{value}, & \text{otherwise} \end{cases}$$

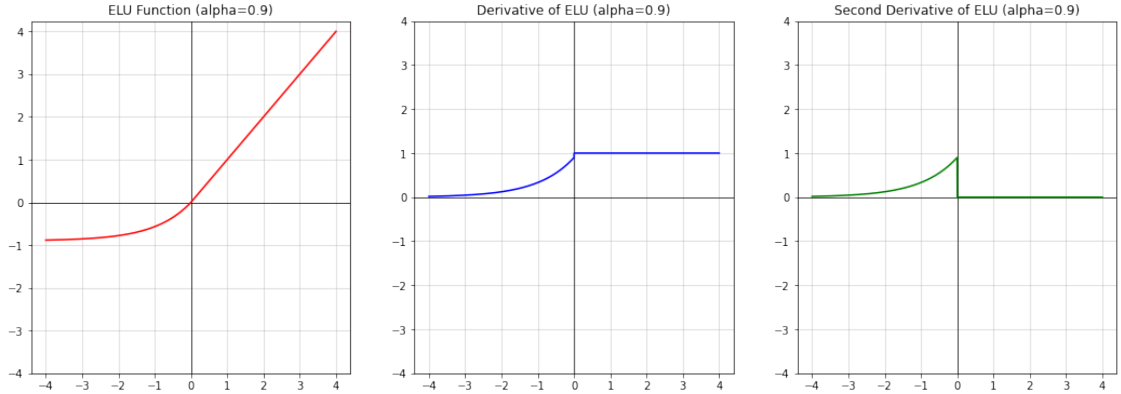
```
@plot_activation_module(threshold=1., value=0.)   # ThresholdReLU: value=0.
class Threshold(ActivationModule):

    def __init__(self, threshold=1., value=0.):
        super().__init__()
        self.threshold = threshold
        self.value = value

    def forward(self, x):
        return x.masked_fill(x <= self.threshold, self.value)
```



11 ELU

$$\mathrm{ELU}(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha * (\exp(x) - 1), & \text{if } x \leq 0 \end{cases}$$

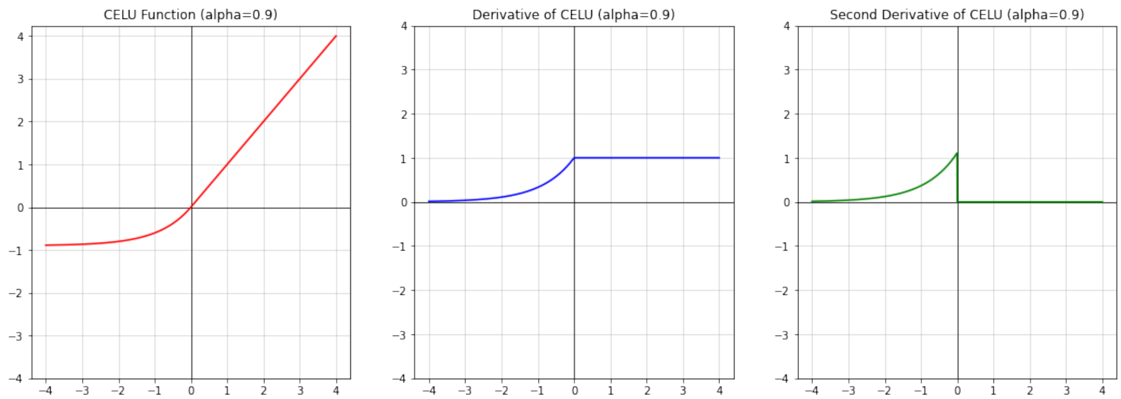
```
@plot_activation_module(alpha=.9)
class ELU(ActivationModule):   # Exponential Linear Unit

    def __init__(self, alpha=1.):
        super().__init__()
        self.alpha = alpha

    def forward(self, x):
        return torch.where(x >= 0., x, (torch.exp(x) - 1.) * self.alpha)
```



12 CELU

$$\mathrm{CELU}(x) = \max(0, x) + \min(0, \alpha * (\exp(x/\alpha) - 1))$$

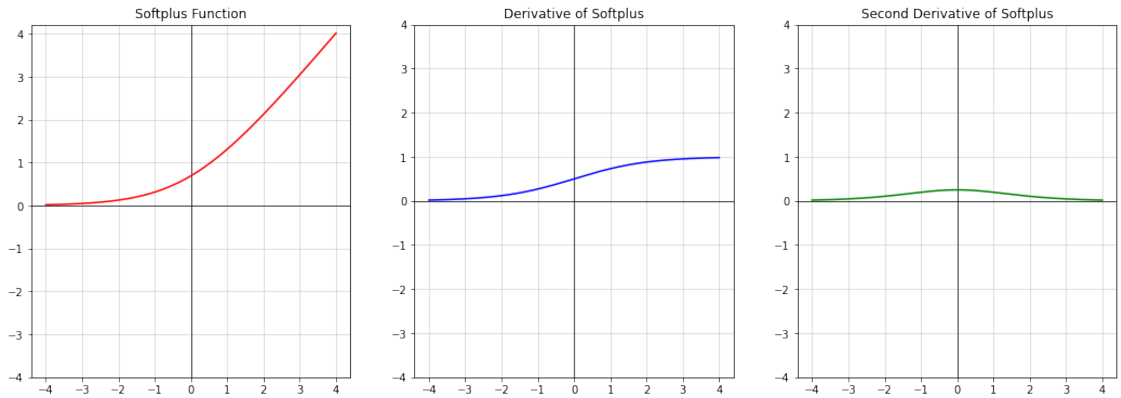
```python
@plot_activation_module(alpha=.9)
class CELU(ActivationModule):  # Continuously differentiable ELU

    def __init__(self, alpha=1.):
        super().__init__()
        self.alpha = alpha

    def forward(self, x):
        return torch.where(x >= 0., x, (torch.exp(x / self.alpha) - 1.) * self.alpha)
```



13 Softplus

$$\text{Softplus}(x) = \frac{1}{\beta} * \log(1 + \exp(\beta * x))$$

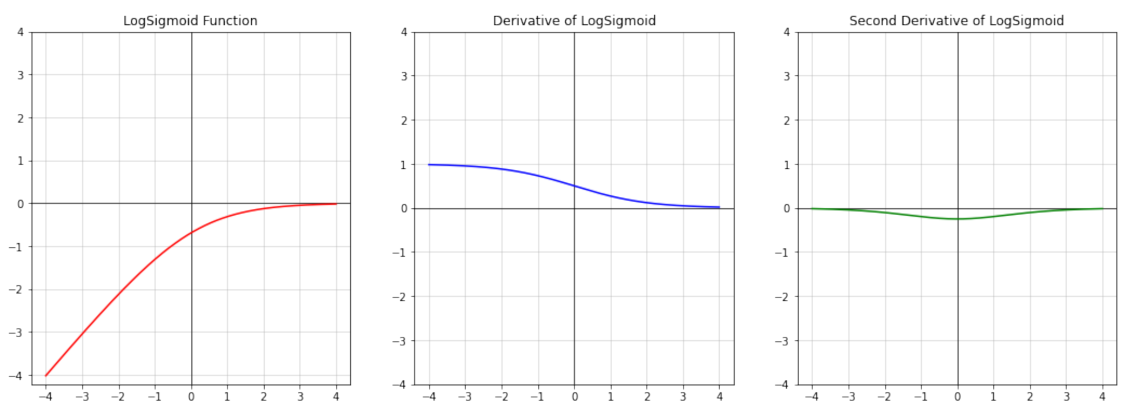
```python
@plot_activation_module()  # Derivative becomes sigmoid
class Softplus(ActivationModule):

    def __init__(self, beta=1.):
        super().__init__()
        self.beta = beta

    def forward(self, x):
        return torch.log(torch.exp(x * self.beta) + 1.)
```



14 LogSigmoid

$$\text{LogSigmoid}(x) = \log\left(\frac{1}{1+\exp(-x)}\right)$$

```
@plot_activation_module()
class LogSigmoid(ActivationModule):

    def forward(self, x):
        return x.sigmoid().log()
```

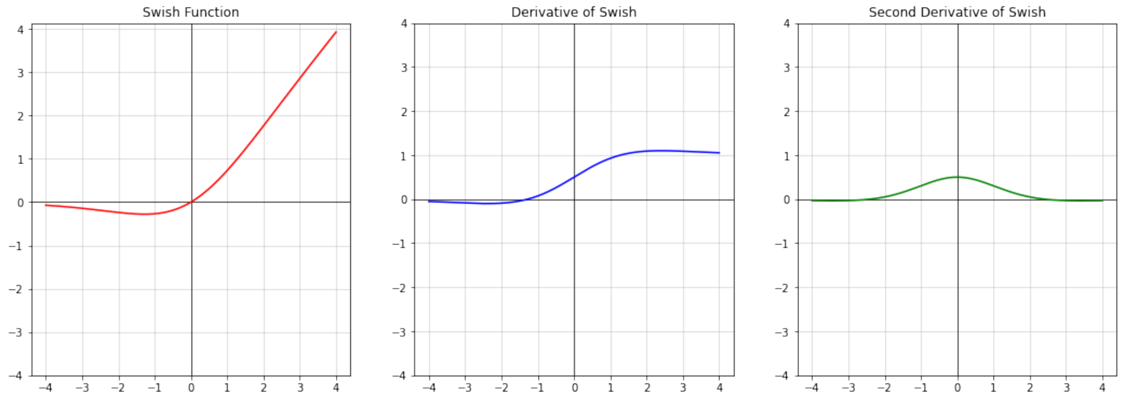


15 Swish

$$\text{Swish}(x) = \text{silu}(x) = x * \sigma(x), \text{where } \sigma(x) \text{ is the logistic sigmoid.}$$

```
@plot_activation_module()
class Swish(ActivationModule):  # same as nn.SiLU: Sigmoid Linear Unit
```

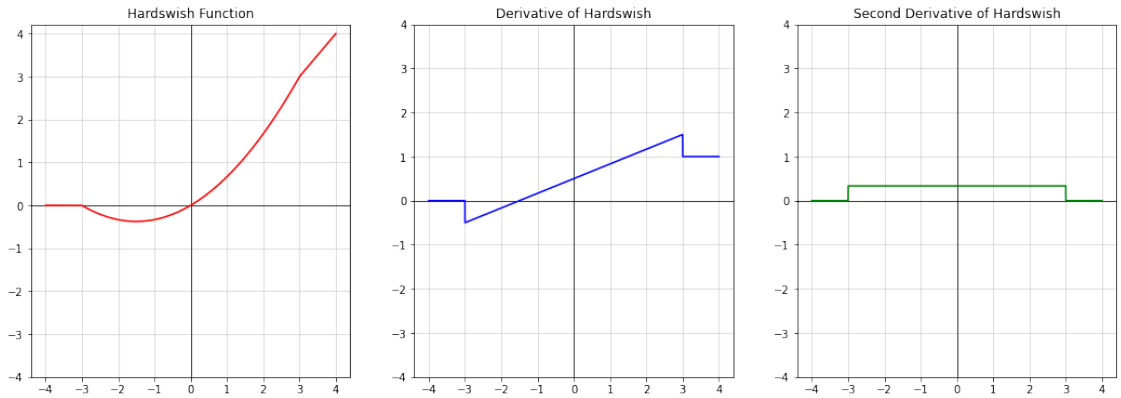
```
    def __init__(self, alpha=1.):
        super().__init__()
        self.alpha = alpha

    def forward(self, x):
        return x * torch.sigmoid(x * self.alpha)
```



16 Hardswish

$$\text{Hardswish}(x) = \begin{cases} 0 & \text{if } x \leq -3, \\ x & \text{if } x \geq +3, \\ x \cdot (x+3)/6 & \text{otherwise} \end{cases}$$

```
@plot_activation_module()
class Hardswish(ActivationModule):   # mobilenetv3

    def forward(self, x):
        return torch.where(
            torch.logical_and(-3. < x, x < 3.),
            x * (x + 3.) / 6.,   # when: -3 < x < 3
            x.relu()
        )
```

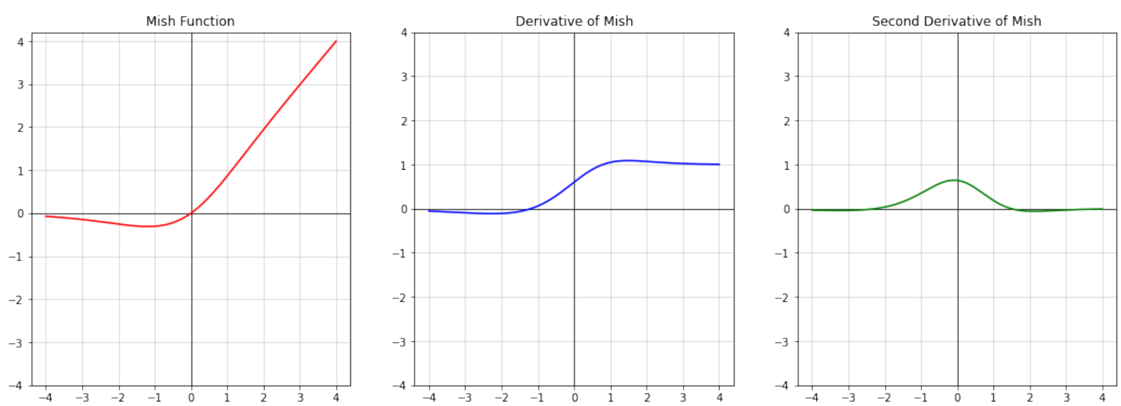


17 Mish

Mish is not implemented in pytorch older versions.

$$\text{Mish}(x) = x * \text{Tanh}(\text{Softplus}(x))$$

```python
@plot_activation_module()
class Mish(ActivationModule):

    def forward(self, x):
        softplus_x = torch.log(torch.exp(x) + 1.)
        return x * torch.tanh(softplus_x)
```



18 GELU

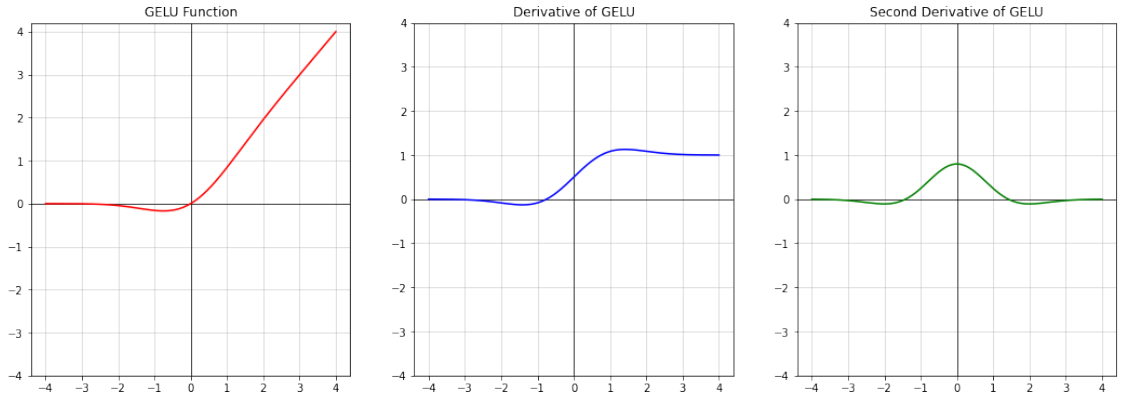
$$\text{GELU}(x) = x * \Phi(x)$$

where $\Phi(x)$ is the Cumulative Distribution Function for Gaussian Distribution.

With Error Function, CDF can be reduced to:

$$\Phi(x) = \frac{1}{2}\left[\operatorname{erf}\left(\frac{x}{\sqrt{2}}\right) + 1\right]$$

```
@plot_activation_module()
class GELU(ActivationModule):  # Gaussian Error Linear Unit

    def gaussian_cdf_function(self, x):
        sqrt_two = math.sqrt(2.)
        phi_x = (torch.erf(x / sqrt_two) + 1.) / 2.
        return phi_x

    def forward(self, x):
        return x * self.gaussian_cdf_function(x)
```

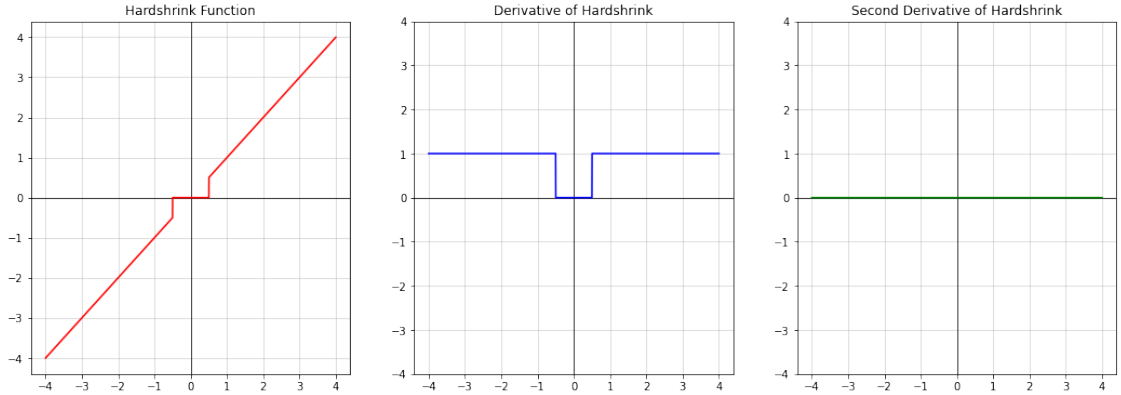


19 Hardshrink

$$\text{HardShrink}(x) = \begin{cases} x, & \text{if } x > \lambda \\ x, & \text{if } x < -\lambda \\ 0, & \text{otherwise} \end{cases}$$

```
@plot_activation_module()
class Hardshrink(ActivationModule):

    def __init__(self, lambd=.5):
        super().__init__()
        self.lambd = lambd

    def forward(self, x):
        x = x.pow(1.)
        return x.where(
            torch.logical_or(x > self.lambd, x < -self.lambd),
            torch.zeros_like(x)  # same as 0
```

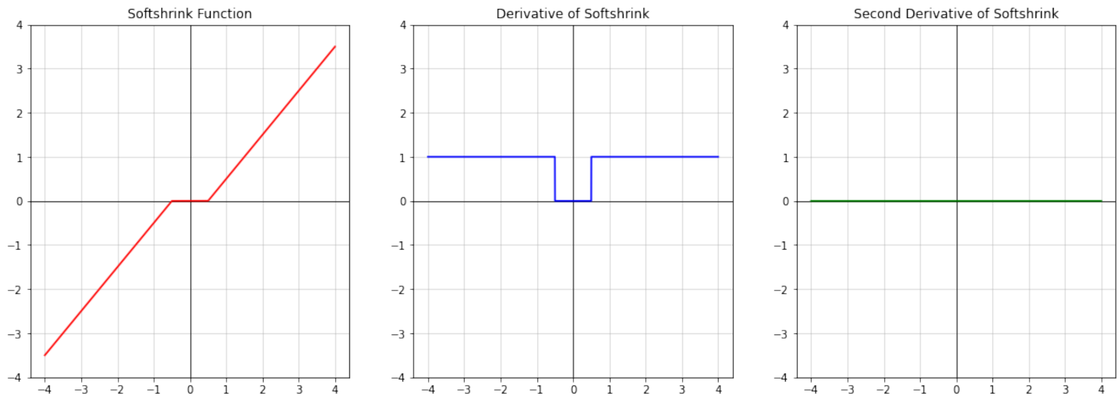
```
        )
```



20 Softshrink

$$\text{SoftShrinkage}(x) = \begin{cases} x - \lambda, & \text{if } x > \lambda \\ x + \lambda, & \text{if } x < -\lambda \\ 0, & \text{otherwise} \end{cases}$$

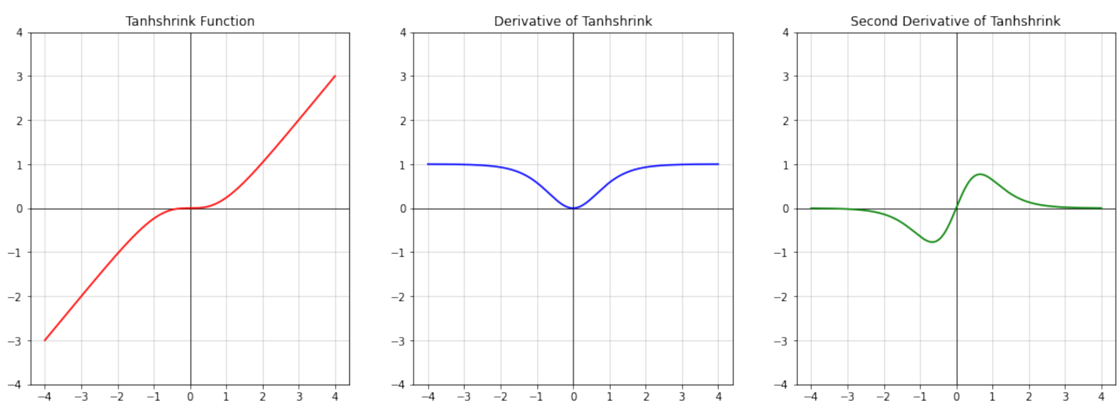
```
@plot_activation_module()
class Softshrink(ActivationModule):

    def __init__(self, lambd=.5):
        super().__init__()
        self.lambd = lambd

    def forward(self, x):
        x = x.pow(1.)
        return torch.where(
            x > self.lambd, # 1st condition
            x - self.lambd,
            torch.where(
                x < -self.lambd,  # 2st condition
                x + self.lambd,
                torch.zeros_like(x)  # same as 0
            )
        )
```



21 Tanhshrink

$$\text{Tanhshrink}(x) = x - \tanh(x)$$

```
@plot_activation_module()
class Tanhshrink(ActivationModule):

    def forward(self, x):
        return x - torch.tanh(x)
```



22 RRelu

$$\text{RReLU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ ax & \text{otherwise} \end{cases}$$

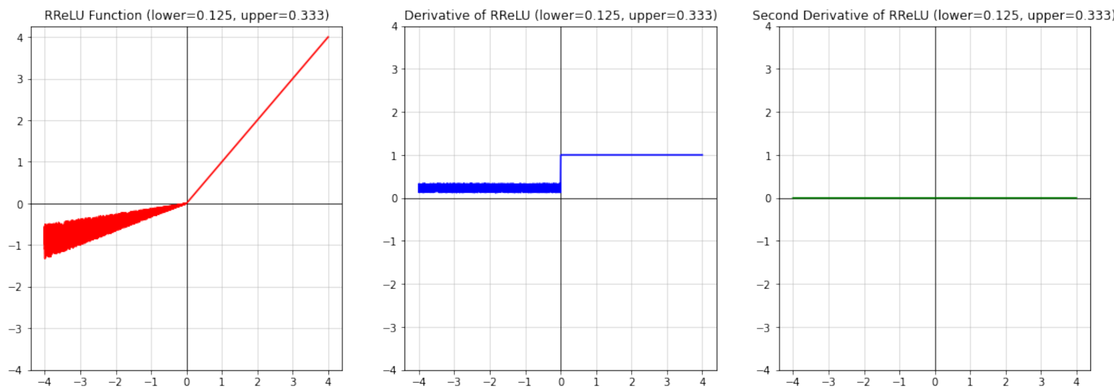
where a is randomly sampled from uniform distribution $\mathcal{U}(\text{lower}, \text{upper})$.

C++ Source in `Activation.cpp`

```cpp
template <typename scalar_t>
inline void _rrelu_with_noise_train(
    Tensor& output,
    const Tensor& input,
    const Tensor& noise,
    const Scalar& lower_,
    const Scalar& upper_,
    c10::optional<Generator> generator) {
  // ...
  for (const auto i : c10::irange(input.numel())) {
    if (input_data[i] <= 0) {
      at::uniform_real_distribution<double> uniform(lower, upper);
      const scalar_t r = (scalar_t)uniform(gen);
      output_data[i] = input_data[i] * r;
      noise_data[i] = r;  // save for backward
    } else {
      noise_data[i] = 1;  // save for backward
      output_data[i] = input_data[i];
    }
  }
  // ...
}

Tensor& rrelu_with_noise_out_cpu(const Tensor& self,
    const Tensor& noise,
    const Scalar& lower,
    const Scalar& upper,
    bool training,
    c10::optional<Generator> generator,
    Tensor& output) {
  if (training) {
    // ...
    _rrelu_with_noise_train<scalar_t>(
      output, self.contiguous(), noise, lower, upper, generator
    );
    // ...
    return output;
  } else {
    // ...
    auto negative = (lower_tensor + upper_tensor) / 2;
    Scalar negative_slope = negative.item();
    return at::leaky_relu_out(output, self, negative_slope);
  }
}
```

```python
# explicitly add bounds to plot
@plot_activation_module(lower=1. / 8, upper=round(1. / 3, 3))
class RReLU(ActivationModule):  # Randomized ReLU

    def __init__(
        self,
        lower=1. / 8,
        upper=1. / 3
```

```
    ):
        super().__init__()
        self.lower = lower
        self.upper = upper

    def forward(self, input):
        if self.training:
            return torch.where(
                input > 0., input,
                # rand_like: [0, 1) -> make it as distribution of [lower ,upper)
                input * (torch.rand_like(input) * (self.upper-self.lower) + self.lower)
            )
        else:
            return self.leaky_relu(input)

    def leaky_relu(self, input):
        negative_slope = (self.lower + self.upper) / 2
        return input.where(input < 0., input * negative_slope)
```



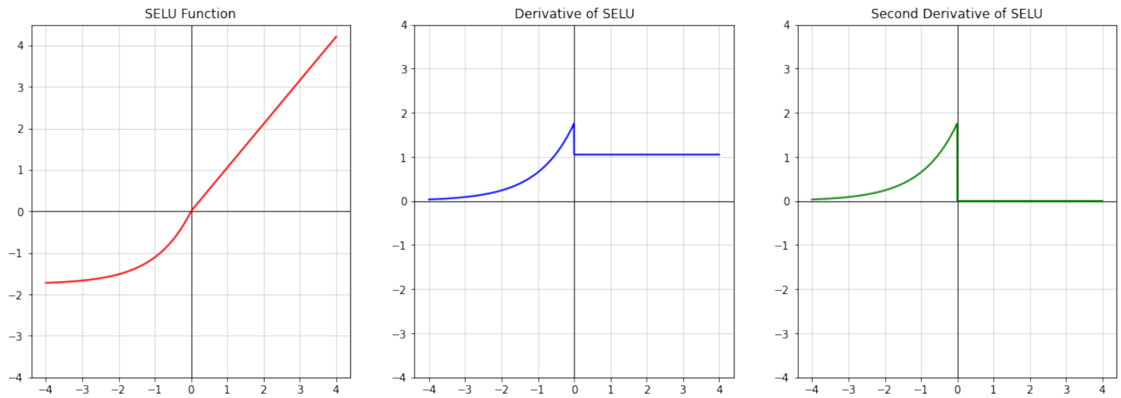
23 SELU

$$\mathrm{SELU}(x) = \mathrm{scale} * (\max(0, x) + \min(0, \alpha * (\exp(x) - 1)))$$

C++ Source in `Activation.cpp`

```
static const double SELU_ALPHA = 1.6732632423543772848170429916717;
static const double SELU_SCALE = 1.0507009873554804934193349852946;
```

```
@plot_activation_module()
class SELU(ActivationModule):  # Scaled ELU

    # Make it as final
    alpha = property(lambda self: 1.6732632423543772848170429916717)
    scale = property(lambda self: 1.0507009873554804934193349852946)

    def forward(self, x):
        return torch.where(x >= 0., x, (torch.exp(x) - 1.) * self.alpha) * self.scale
```



24 GLU

$$\mathrm{GLU}(a,b) = a \otimes \sigma(b)$$

where a is the first half of the input matrices and b is the second half.

C++ Source in `Activation.cu`

```
void glu_kernel(TensorIteratorBase& iter) {
  AT_DISPATCH_FLOATING_TYPES_AND2(kHalf, kBFloat16, iter.dtype(), "glu_cuda", [&]() {
    using acc_t = at::acc_type<scalar_t, true>;
    gpu_kernel(iter, [] GPU_LAMBDA (scalar_t a_, scalar_t b_) -> scalar_t {
      const acc_t a = a_;
      const acc_t b = b_;
      const acc_t one = acc_t(1);
      const acc_t sigmoid = one / (one + std::exp(-b));
      return a * sigmoid;
    });
  });
}
```

```
class GLU(ActivationModule):   # Gated Linear Unit

    def __init__(self, dim=-1):
        super().__init__()
        self.dim = dim

    def forward(self, input):
        a, b = input.chunk(2, dim=self.dim)
        return a * b.sigmoid()
```

25 Softmax

$$\mathrm{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

```python
class Softmax(ActivationModule):

    # Implicit Inference of Dim
    _get_softmax_dim = staticmethod(lambda ndim: 0 if ndim in (0, 1, 3) else 1)

    def __init__(self, dim=None):
        super().__init__()
        self.dim = dim

    def forward(self, input):
        dim = self.dim
        if dim is None:
            dim = self._get_softmax_dim(input.ndim)
        input = input - input.amin(dim).unsqueeze(dim)
        input_exp = input.exp()
        return input_exp / input_exp.sum(dim).unsqueeze(dim)
```

```python
class Softmax2d(Softmax):

    def __init__(self):
        super().__init__(dim=1)

    def forward(self, input):
        assert input.ndim == 4, 'Softmax2d requires a 4D tensor as input'
        return super().forward(input)
```

26 Softmin

$$\text{Softmin}(x_i) = \frac{\exp(-x_i)}{\sum_j \exp(-x_j)}$$

```python
class Softmin(Softmax):

    def forward(self, input):
        return super().forward(-input)
```

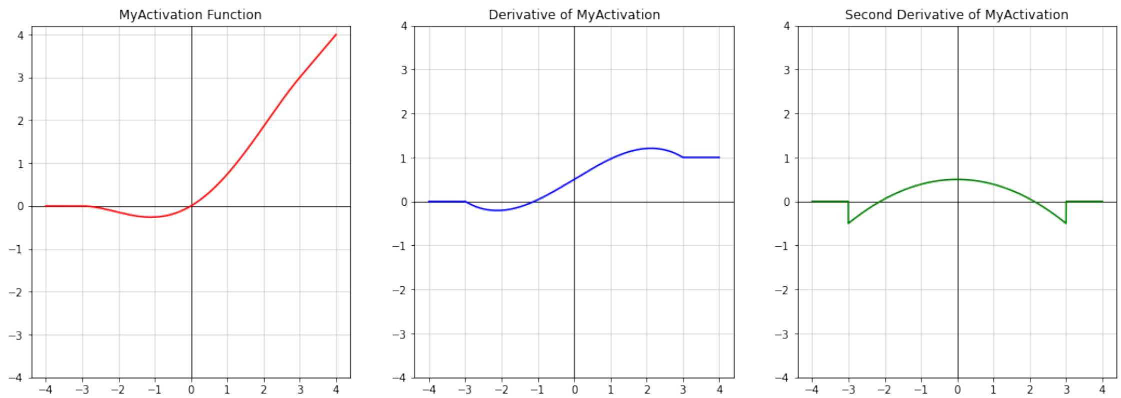
27 LogSoftmax

$$\text{LogSoftmax}(x_i) = \log\left(\frac{\exp(x_i)}{\sum_j \exp(x_j)}\right)$$

```python
class LogSoftmax(Softmax):

    def forward(self, input):
        return super().forward(input).log()
```

28 Continuously differentiable HardSwish

```
@plot_activation_module()
class MyActivation(ActivationModule):  # Continuously differentiable HardSwish

    def forward(self, x):
        return torch.where(
            torch.logical_and(-3. < x, x < 3.),
            x * (x + 3.) * (x + 3.) * (-x + 6.) / 108.,  # when: -3 < x < 3
            x.relu()
        )
```



Summary

All Modules are:

```
ModuleList(
  (0): CELU()
  (1): ELU()
  (2): GELU()
  (3): GLU()
  (4): Hardshrink()
  (5): Hardsigmoid()
  (6): Hardswish()
  (7): Hardtanh()
  (8): LeakyReLU()
  (9): LogSigmoid()
  (10): LogSoftmax()
  (11): Mish()
  (12): MyActivation()
  (13): PReLU()
  (14): RReLU()
  (15): ReLU()
  (16): ReLU6()
  (17): SELU()
  (18): Sigmoid()
  (19): Softmax()
  (20): Softmax2d()
  (21): Softmin()
  (22): Softplus()
  (23): Softshrink()
  (24): Softsign()
  (25): Swish()
  (26): Tanh()
  (27): Tanhshrink()
  (28): Threshold()
)
```