Vanishing/exploding gradients

Now that we have a better view of how backward propagation of gradients work, we are equipped to understand the difficulties of training the weights.

Until the problems were understood, and solutions found, the evolution of Deep Learning was extremely slow.

Let's summarize back propagation up until this point

- We compute the loss gradient $\mathcal{L}'_{(l)} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}_{(l)}}$ of each layer l in descending order
- The backward step to compute the loss gradient of the preceding layer is:
    - $\mathcal{L}'_{(l-1)} = \mathcal{L}'_{(l)} \frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(l-1)}}$

When we derived back propagation, we didn't look "inside" of the "local gradient " $\frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(l-1)}}$

We will do so now.

Let's look more deeply into the term $\frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(i-1)}}$

$$
\begin{aligned}
\frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(l-1)}} &= \frac{\partial a_{(l)}(f_{(l)}(\mathbf{y}_{(l-1)}, \mathbf{W}_{(l)}))}{\partial \mathbf{y}_{(l-1)}} & \text{(def. of } \mathbf{y}_{(l)}\text{)} \\
&= \frac{\partial a_{(l)}(f_{(l)}(\mathbf{y}_{(l-1)}, W_{(l)}))}{\partial f_{(l)}(\mathbf{y}_{(l-1)}, \mathbf{W}_{(l)})} \frac{\partial f_{(l)}(\mathbf{y}_{(l-1)}, \mathbf{W}_{(l)})}{\partial \mathbf{y}_{(l-1)}} & \text{(chain rule)} \\
&= a'_{(l)} f'_{(l)}
\end{aligned}
$$

where we define

$$
\begin{aligned}
a'_{(l)} &= \frac{\partial a_{(l)}(f_{(l)}(\mathbf{y}_{(l-1)}, \mathbf{W}_{(l)}))}{\partial f_{(l)}(\mathbf{y}_{(l-1)}, \mathbf{W}_{(l)})} & \text{derivative of } a_{(l)}(\ldots) \text{ wrt } f_{(l)}(\ldots) \\
f'_{(l)} &= \frac{\partial f_{(l)}(\mathbf{y}_{(l-1)}, W_{(l)})}{\partial \mathbf{y}_{(l-1)}} & \text{derivative of } f_{(l)}(\ldots) \text{ wrt } \mathbf{y}_{(l-1)}
\end{aligned}
$$

$a'_{(l)}$ is the derivative of activation function $a_{(l)}$ with respect to its input.

- we saw graphs of this when we introduced activation functions
- see below

We won't explicitly write it out other than to observe $a'_{(l)} \in [0, 1]$.

Substituting the value of the loss gradient into the backward update rule:

$$\begin{aligned}\mathcal{L}'_{(l-1)} &= \mathcal{L}'_{(l)} \frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(l-1)}} \\ &= \mathcal{L}'_{(l)} a'_{(l)} f'_{(l)}\end{aligned}$$

We now do the same for the $\mathcal{L}'_{(l)}$ term expanding

$$\begin{aligned}\mathcal{L}'_{(l)} &= \mathcal{L}'_{(l+1)} \frac{\partial \mathbf{y}_{(l+1)}}{\partial \mathbf{y}_{(l)}} \\ &= \mathcal{L}'_{(l+1)} a'_{(l+1)} f'_{(l+1)}\end{aligned}$$

Substituting this back into $\mathcal{L}'_{(l-1)}$

$$\begin{aligned}\mathcal{L}'_{(l-1)} &= \mathcal{L}'_{(l+1)} a'_{(l+1)} f'_{(l+1)} a'_{(l)} f'_{(l)} \\ &= a'_{(l+1)} a'_{(l)} f'_{(l+1)} f'_{(l)} && \text{re-arranging terms}\end{aligned}$$

Hopefully, you can see that if iterate through single backward steps, we can derive an expression for the loss gradient at layer l in terms of the loss gradient of the final layer K:

Since

$$\mathcal{L}'_{(l)} = \mathcal{L}'_{(l+1)} \frac{\partial \mathbf{y}_{(l+1)}}{\partial \mathbf{y}_{(l)}}$$

we get

$$\mathcal{L}'_{(l)} = \mathcal{L}'_{(L+1)} \prod_{l'=l+1}^{L} a'_{(l')} f'_{(l')}$$

The issue is that, since

$$0 \leq a'_{(l)} \leq \max_z a'_{(l)}(z)$$

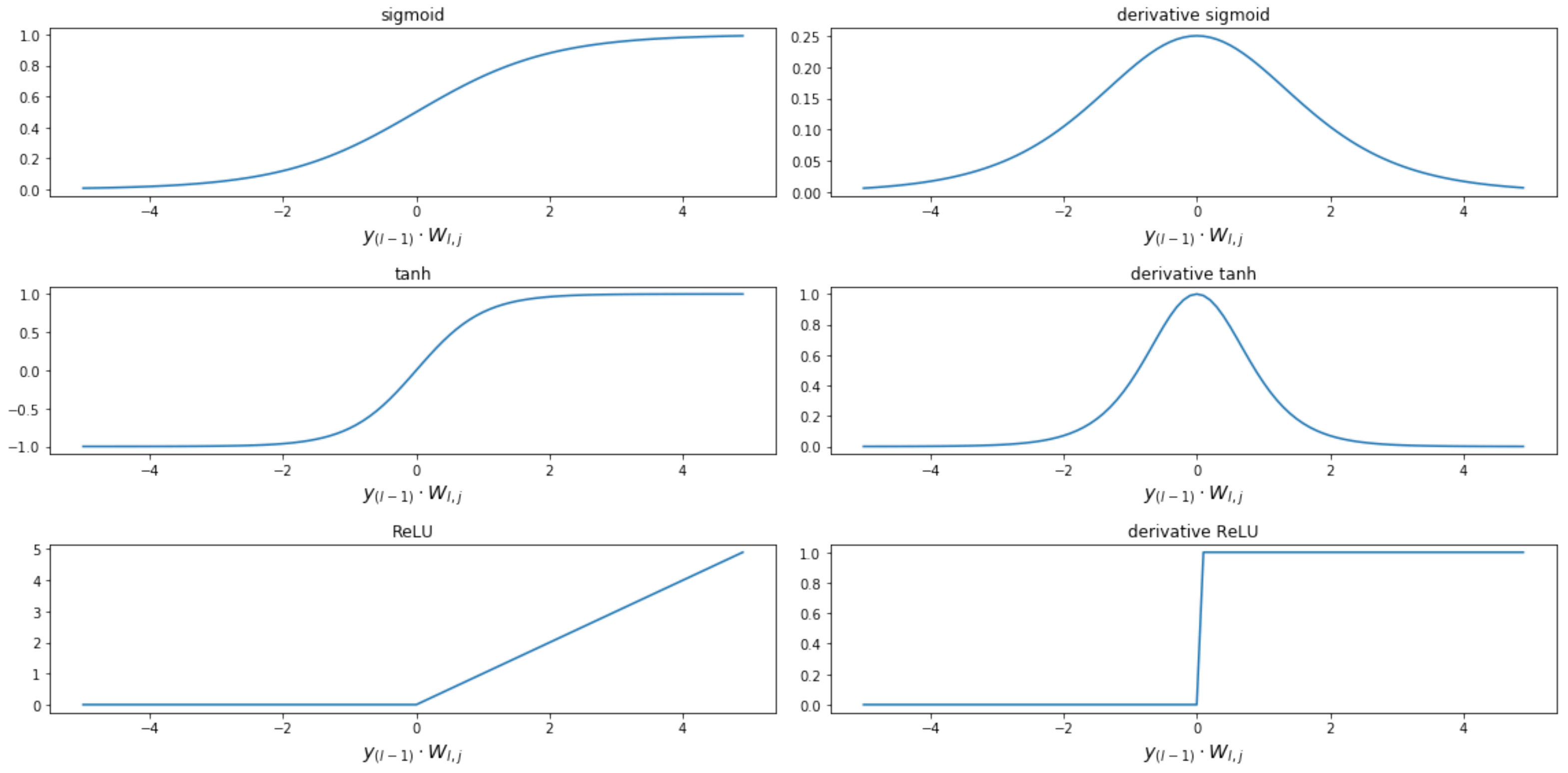
the product

$$\prod_{l'=i+1}^{K} a'_{(l')}$$

can be increasingly small as the number of layers K grows, if $\max_z a'_{(l)}(z) < 1$.

A plot of activation functions and their derivatives follows.

```
In [5]: fig, _ = nnh.plot_activations( np.arange(-5,5, 0.1) )
```



Thus, unless offset by the $f'_{(l)}$ terms, $\mathcal{L}'_{(l)}$ will quickly diminish to 0 as K decreases, i.e., as we seek to compute $\mathcal{L}'_{(l)}$ for layers l closest to the input.

This means

$$\frac{\partial \mathcal{L}}{\partial W_{(l)}} \quad = \quad \frac{\partial \mathcal{L}}{\partial \mathbf{y}_{(l)}} \frac{\partial \mathbf{y}_{(l)}}{\partial W_{(l)}} \quad = \quad \mathcal{L}'_{(l)} \frac{\partial \mathbf{y}_{(l)}}{\partial W_{(l)}}$$

will approach 0.

Since this term is used in the update to $W_{(l)}$, we won't learn weights for the earliest layers.

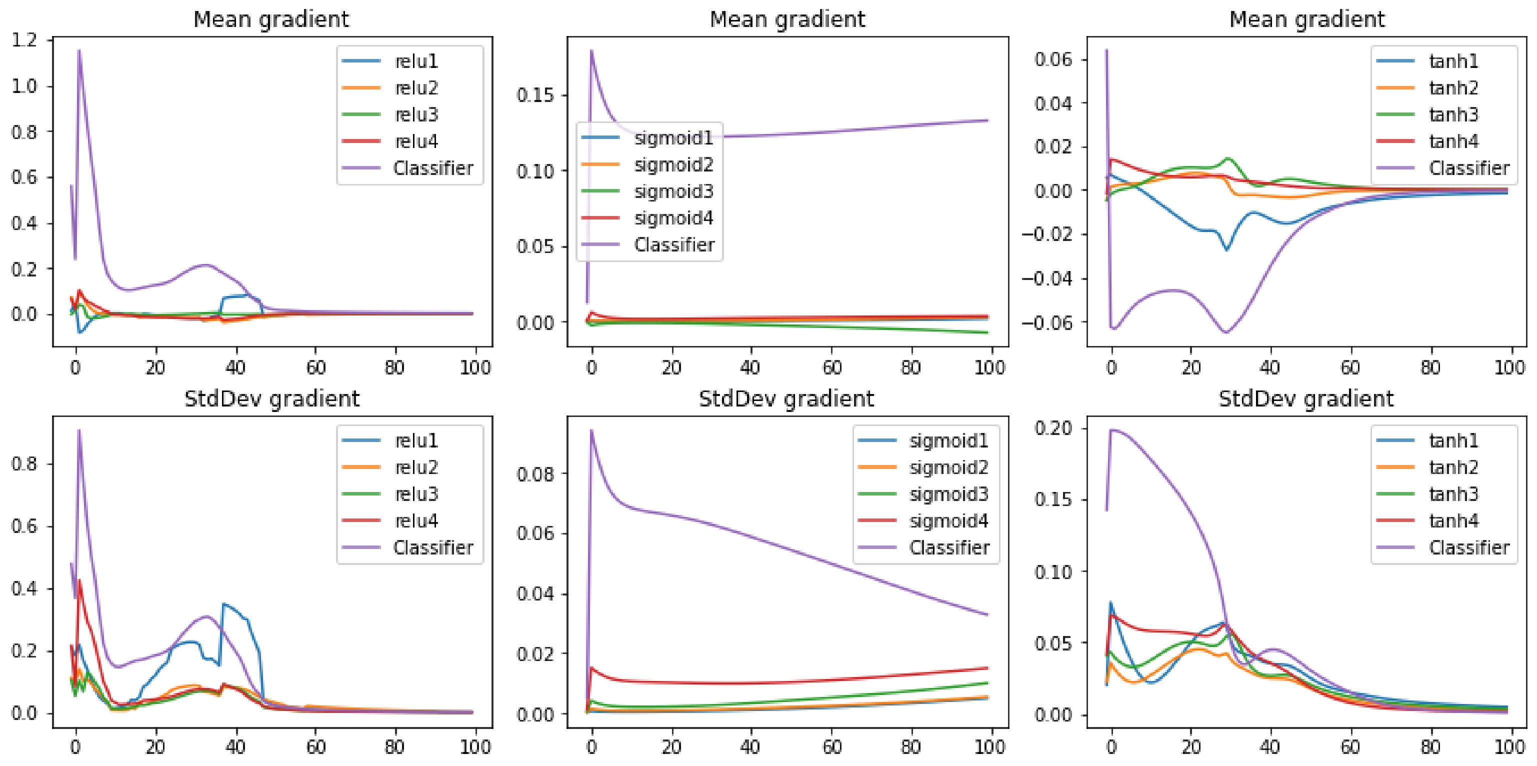
Re-visiting our running example

- we compute the (average/variance across units) gradient value
- for each layer in each epoch

Focusing on Sigmoid column of the chart below

- we see that mean gradients are essentially 0 throughout training for non-Classifier layers
    - mean 0, very small standard deviation:
        - almost all samples from this distribution are near 0
- note that the maximum value of the Sigmoid's derivative is 0.25
    - so the Sigmoid activation is very exposed to Vanishing gradients

RELU	SIGMOID	TANH



The gradient behavior across epochs for the networks using ReLU or Tanh are somewhat better

- small mean, but larger variance. Hence the gradient for many activations is non-zero.
- Observe that the maximum derivative for these activations is around 1.

We can now diagnose one reason that training of early Deep Learning networks was difficult

- use of sigmoid activations were common (inspired by biology)
- if activations were very large/small, we are in a region where the sigmoid's derivatives are 0
- even when non-zero,the maximum of the derivative of the sigmoid is much smaller than 1
- the end result was that deep networks suffered from Vanishing Gradients

Why the ReLU became the common activation function

One half of the domain of the ReLU has non-zero (and, depending on slope, greater than 1) gradient.

- This makes it less susceptible to the Vanishing gradient problem.

Still: half of the domain of the ReLU has zero gradient, which may impede learning.

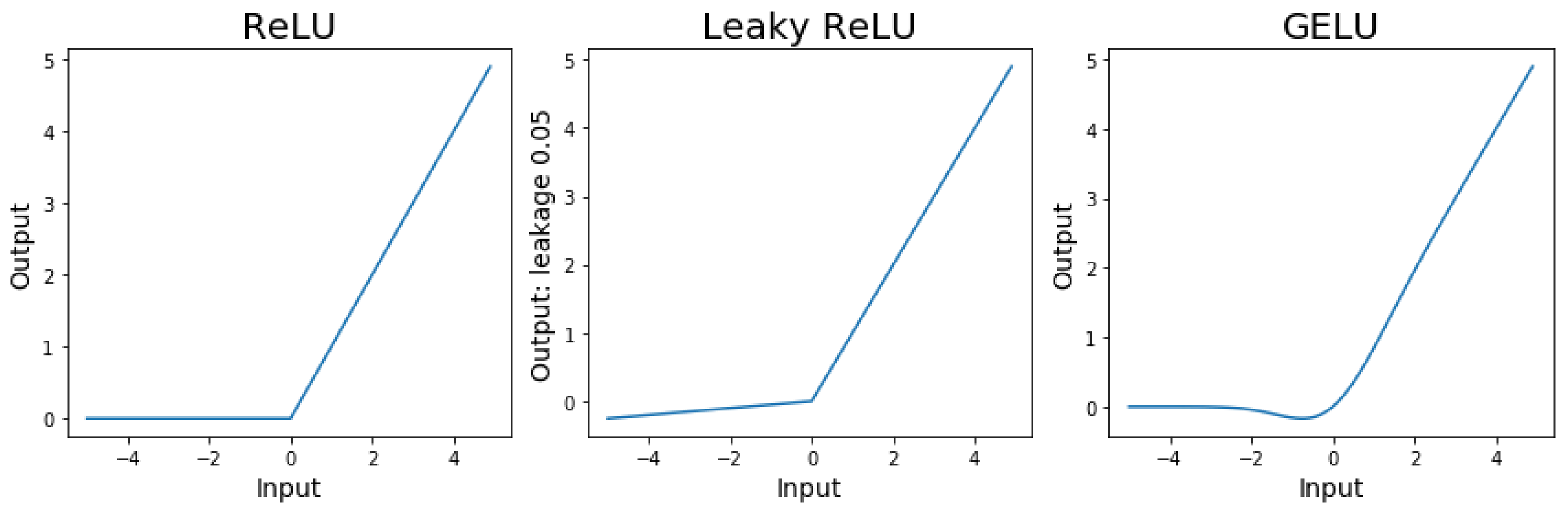
There are variants of ReLU

- constructed such that gradient is non-zero everywhere
- hopefully facilitating faster convergence in training

Some examples (and their plots below)

- Leaky ReLU
  $$\text{LeakyReLU}(x) = x * (x \ge 0) + x * \alpha * (x < 0)$$
  for small α (leakage) rate
- [Gaussian Error Linear Unit (GELU) (https://arxiv.org/pdf/1606.08415v3.pdf)](https://arxiv.org/pdf/1606.08415v3.pdf)
  $$\text{GELU}(x) = x * \text{NormCDF}(x)$$
  The belief is that this may facilitate learning.

```
In [6]: _= plot_RELU_variants()
```



Conclusion

Something seemingly as simple as taking derivatives turned out to have some important subtleties.

The problem of gradients either shrinking to zero or growing too large is a real problem

- It can still hinder the use of very deep (many layers) networks
- This is particularly a problem in Recurrent networks
    - The depth of the "unrolled loop" is the length of the input sequence

We will explore techniques to manage the issue of vanishing and exploding gradients.

```
In [7]: print("Done")
```

```
Done
```