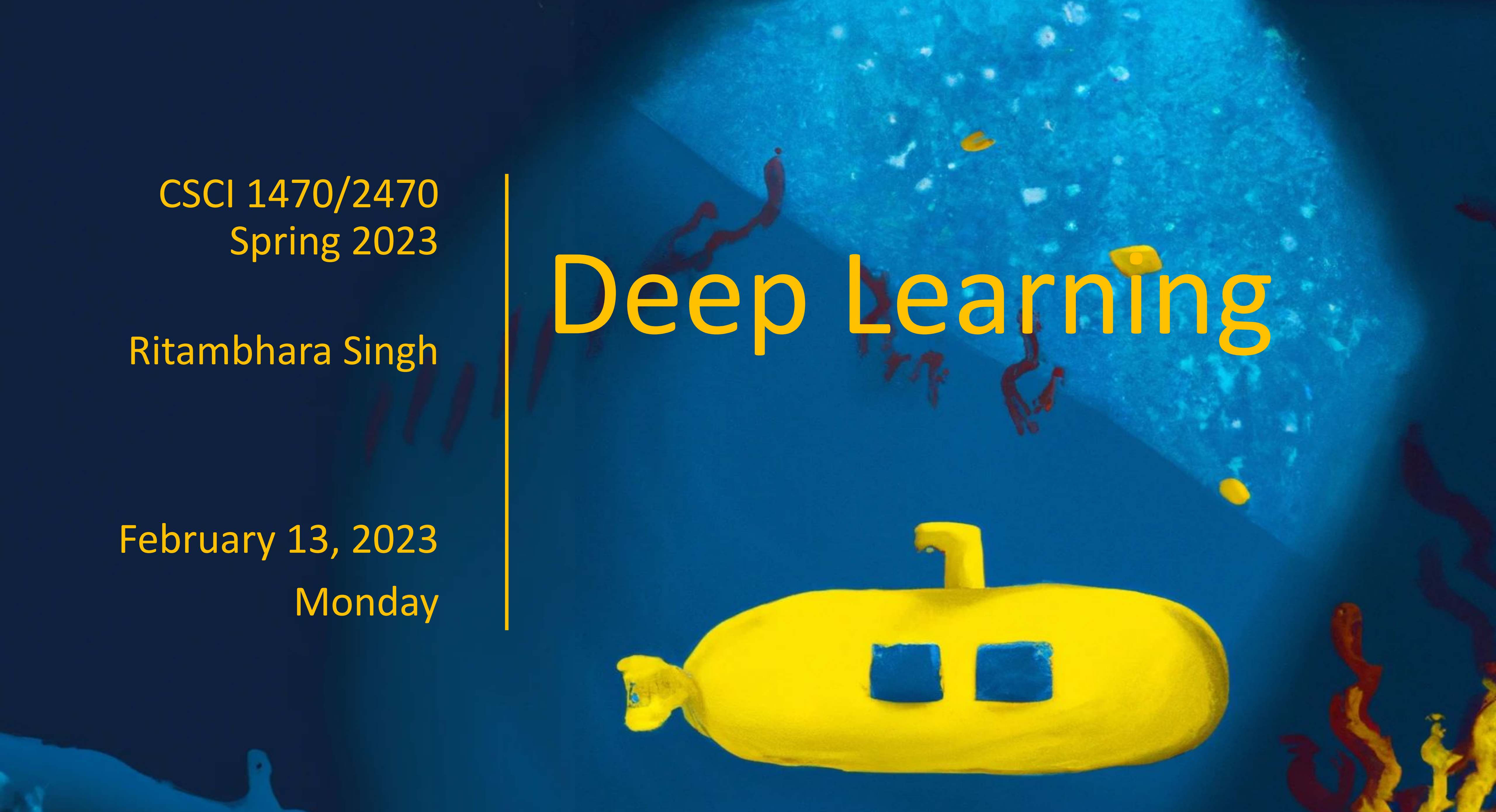


Deep Learning

CSCI 1470/2470
Spring 2023

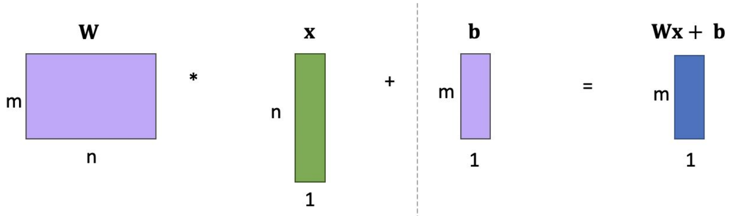
Ritambhara Singh

February 13, 2023
Monday

DALL-E 2 prompt “a painting of deep underwater with a yellow submarine in the bottom right corner”

Recap

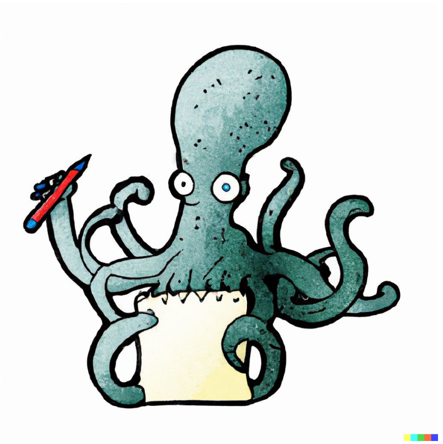
Neural networks as matrix operations



$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\\10 & 11 & 12\end{bmatrix} + \begin{bmatrix}100 & 200 & 300\end{bmatrix} = \begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\\10 & 11 & 12\end{bmatrix} + \begin{bmatrix}100 & 200 & 300\\100 & 200 & 300\\100 & 200 & 300\\100 & 200 & 300\end{bmatrix} = \begin{bmatrix}101 & 202 & 303\\104 & 205 & 306\\107 & 208 & 309\\110 & 211 & 312\end{bmatrix}$$

Broadcasting

Batching and Broadcasting



Intro to Tensorflow

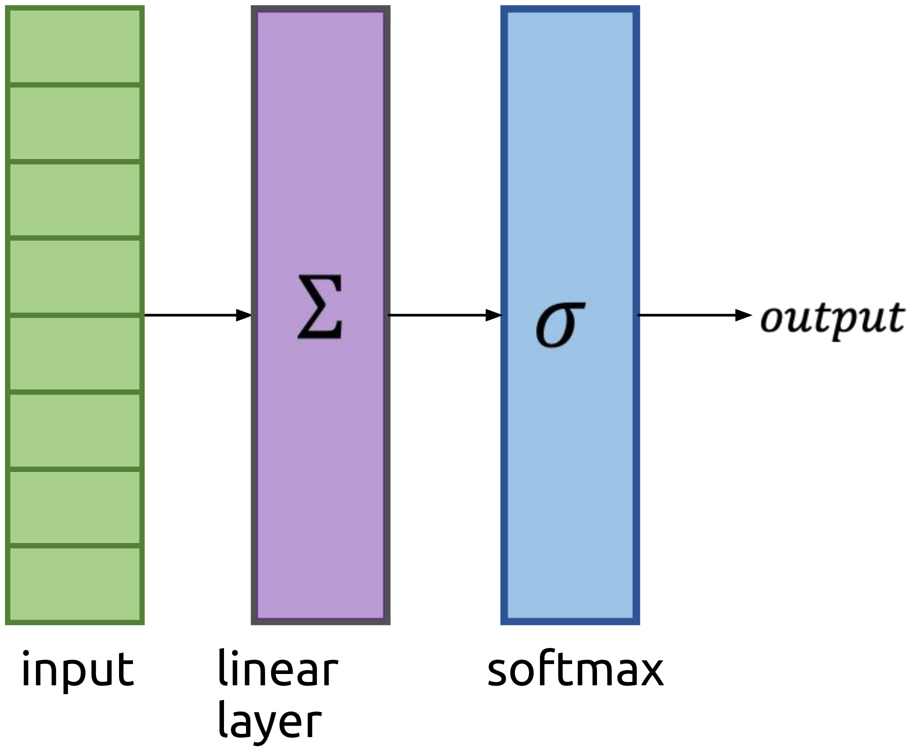
Today's goal – learn to build multi-layer neural networks

(1) Adding more layers to the network

(2) Introducing non-linearity (Activation functions)

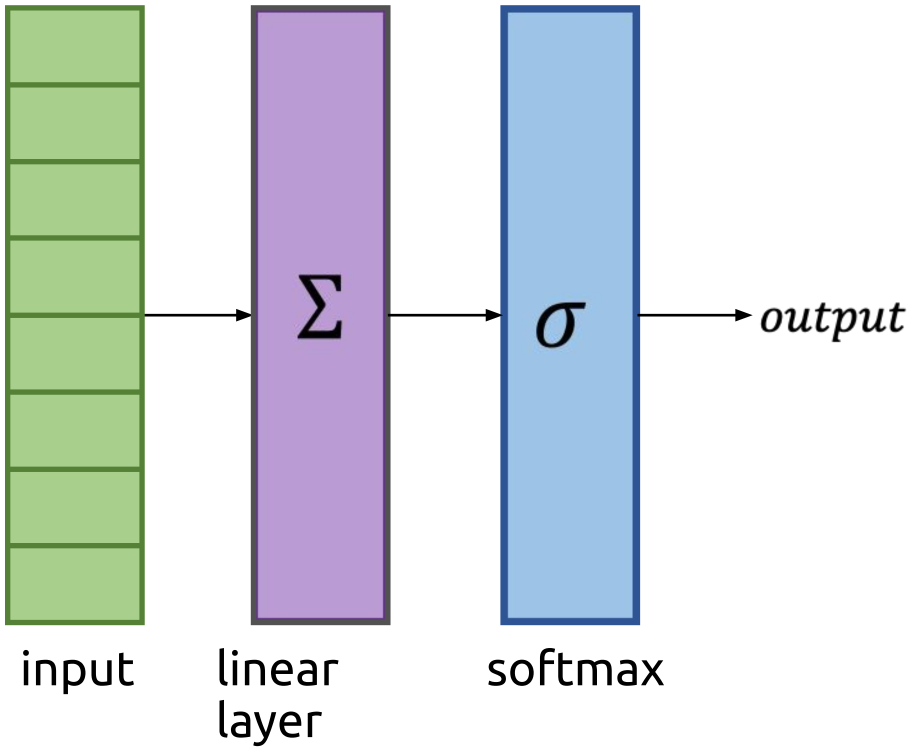
(3) Multi-layer neural network with non-linearity

Single Layer Fully Connected Feed Forward Neural Network



This network can achieve ~90% accuracy on the MNIST test set

Single Layer Fully Connected Feed Forward Neural Network

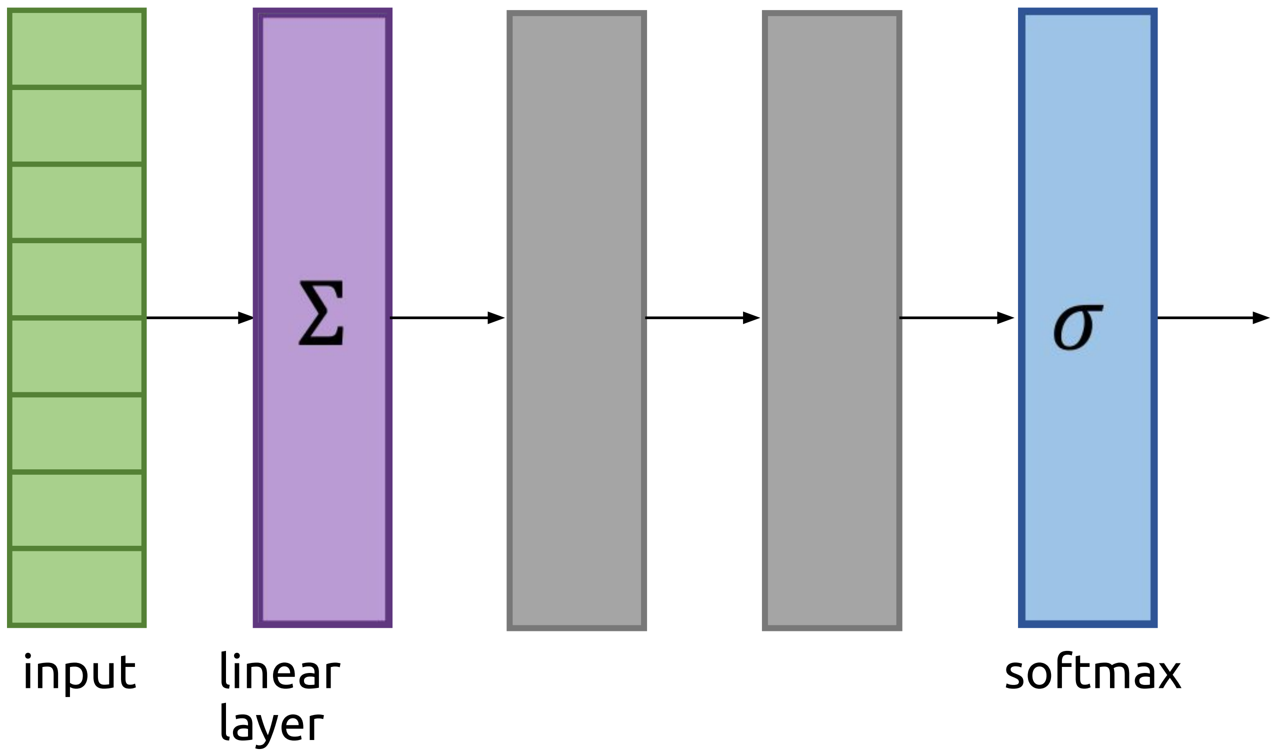


This network can achieve ~90% accuracy on the MNIST test set

How can we do better?

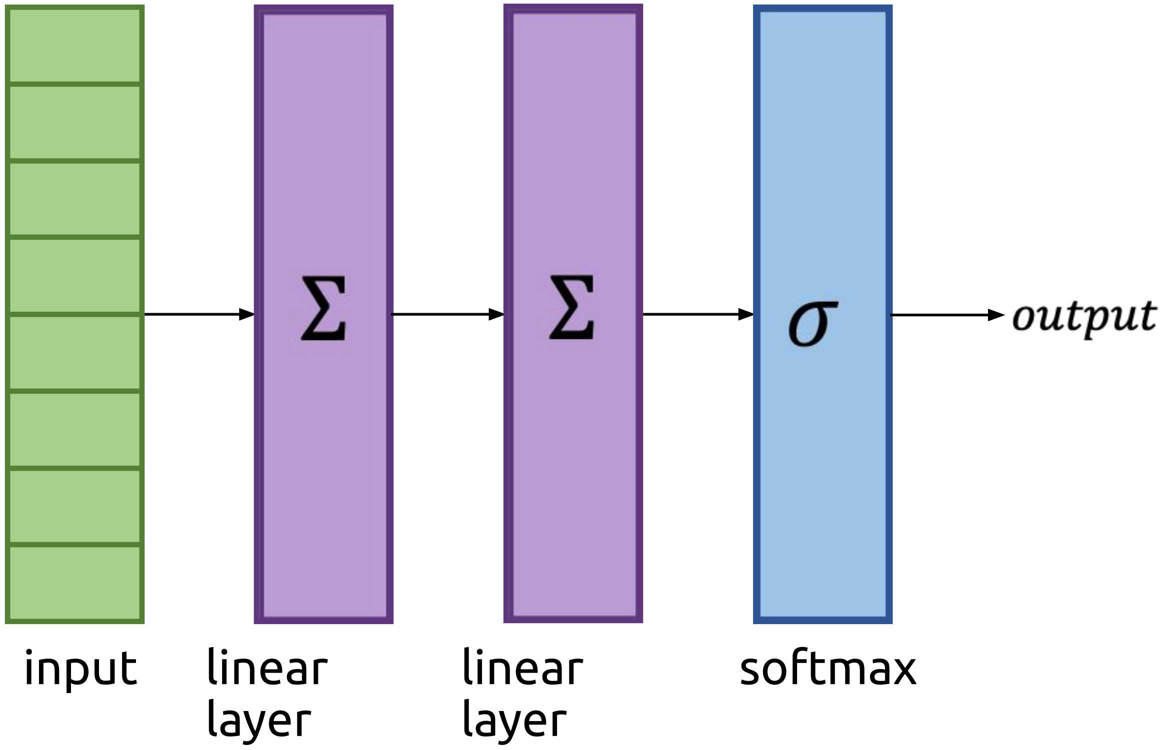
Go deeper!

Multi-layer Neural Networks

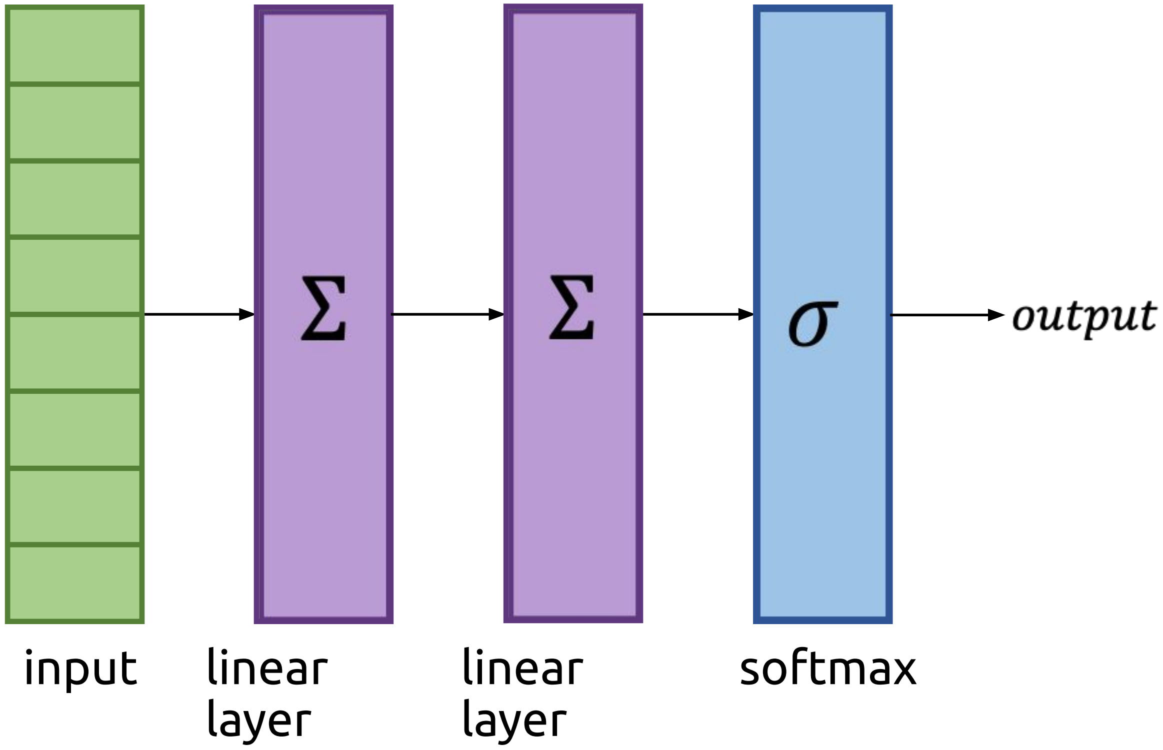


- Each new layer adds another function to the network
  - $f\left(g\left(h(\ldots z(x)\ldots)\right)\right)$
  - More composed functions → can represent more complex computations
- Each new layer has its own tunable parameters
  - More parameters to tune → can capture more complex patterns in the data

One Way to Make a Multi-layer Network



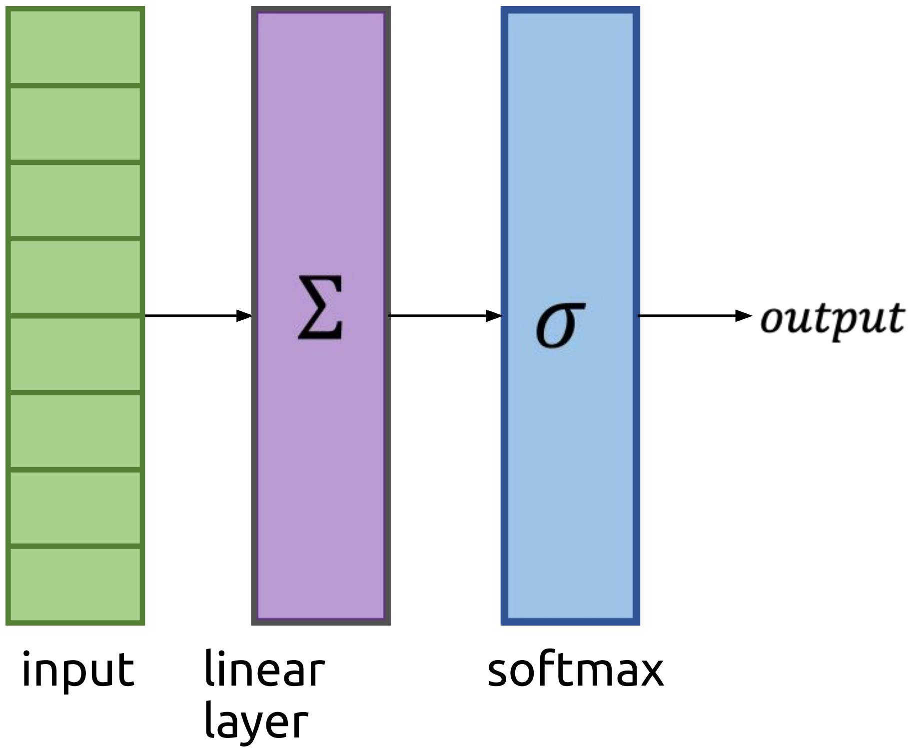
One Way to Make a Multi-layer Network



Obvious idea: just stack more linear layers

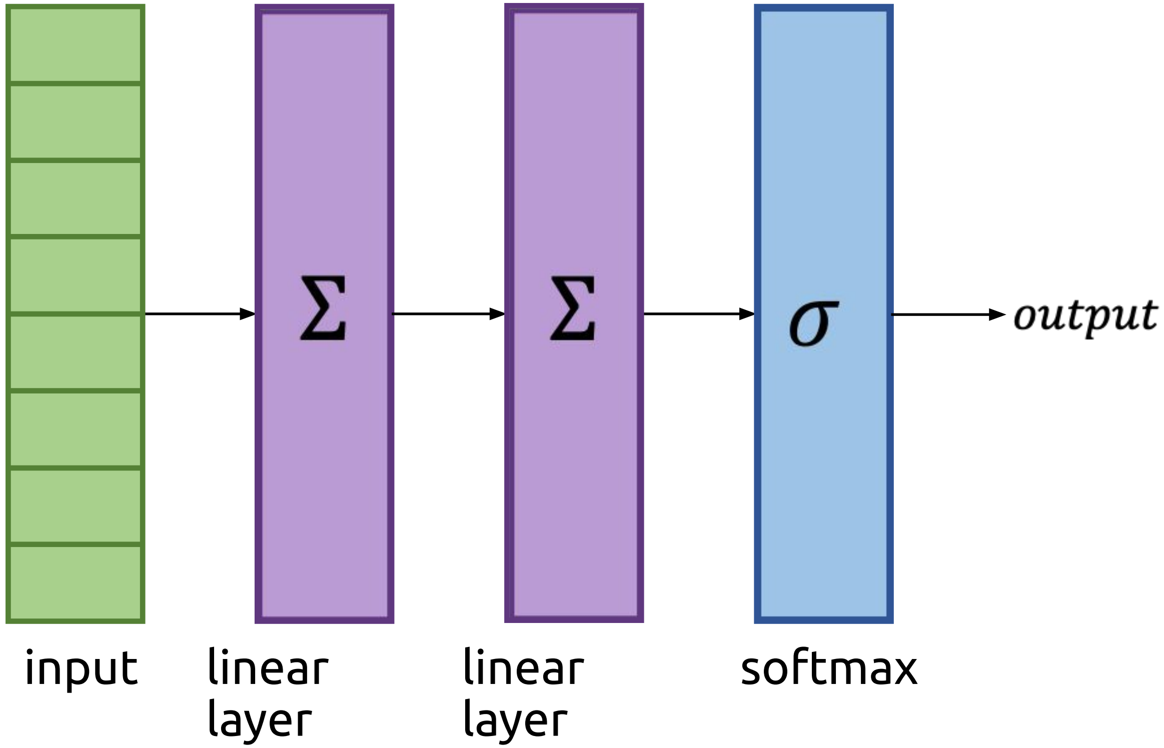
Let’s examine the consequences of this design decision...

Single-Layer Network (in math)



$$\sigma\left(\begin{bmatrix} w_2 & b_2 \end{bmatrix} \begin{bmatrix} x \\ 1 \end{bmatrix}\right)$$

Multi-Layer Network (in math)



$$\sigma\left(\begin{bmatrix} w_2 & b_2 \end{bmatrix}\left(\begin{bmatrix} w_1 & b_1 \end{bmatrix}\begin{bmatrix} x \\ 1 \end{bmatrix}\right)\right)$$

Let's simplify this a bit...

Simplifying multi-layer math...

$$\sigma\left(\begin{bmatrix} w_2 & b_2 \end{bmatrix}\left(\begin{bmatrix} w_1 & b_1 \end{bmatrix}\begin{bmatrix} x \\ 1 \end{bmatrix}\right)\right)$$

Simplifying multi-layer math...

$$\sigma\left(\begin{bmatrix} w_2 & b_2 \end{bmatrix}\left(\begin{bmatrix} w_1 & b_1 \end{bmatrix}\begin{bmatrix} x \\ 1 \end{bmatrix}\right)\right)$$

Apply associativity...

$$\sigma\left(\left(\begin{bmatrix} w_2 & b_2 \end{bmatrix}\begin{bmatrix} w_1 & b_1 \end{bmatrix}\right)\begin{bmatrix} x \\ 1 \end{bmatrix}\right)$$

Multiply the matrices...

$$\sigma\left(\begin{bmatrix} w_{12} & b_{12} \end{bmatrix}\begin{bmatrix} x \\ 1 \end{bmatrix}\right)$$

Simplifying multi-layer math...

$$\sigma\left(\begin{bmatrix} w_2 & b_2 \end{bmatrix}\left(\begin{bmatrix} w_1 & b_1 \end{bmatrix}\begin{bmatrix} x \\ 1 \end{bmatrix}\right)\right)$$

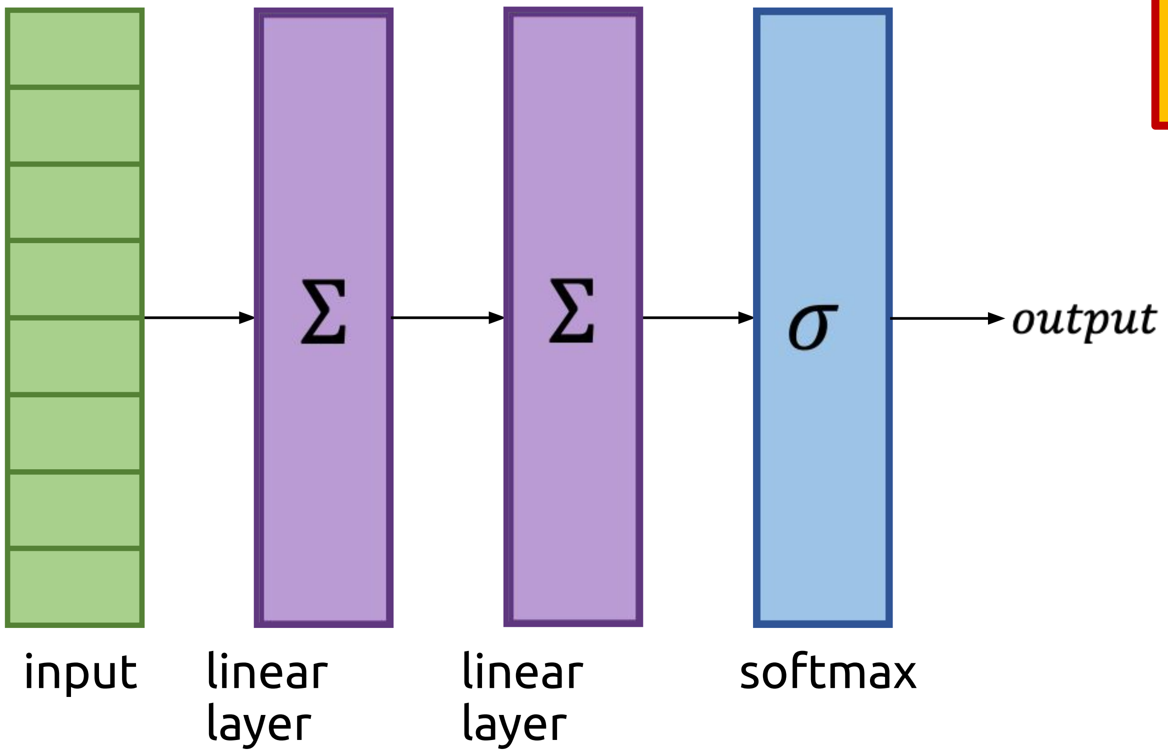
Apply associativity...

$$\sigma\left(\left(\begin{bmatrix} w_2 & b_2 \end{bmatrix}\begin{bmatrix} w_1 & b_1 \end{bmatrix}\right)\begin{bmatrix} x \\ 1 \end{bmatrix}\right)$$

Multiply the matrices...

Same as a one-layer network $\longrightarrow$ $$\sigma\left(\begin{bmatrix} w_{12} & b_{12} \end{bmatrix}\begin{bmatrix} x \\ 1 \end{bmatrix}\right)$$

Takeaway: Stacking Linear Layers Isn't Enough



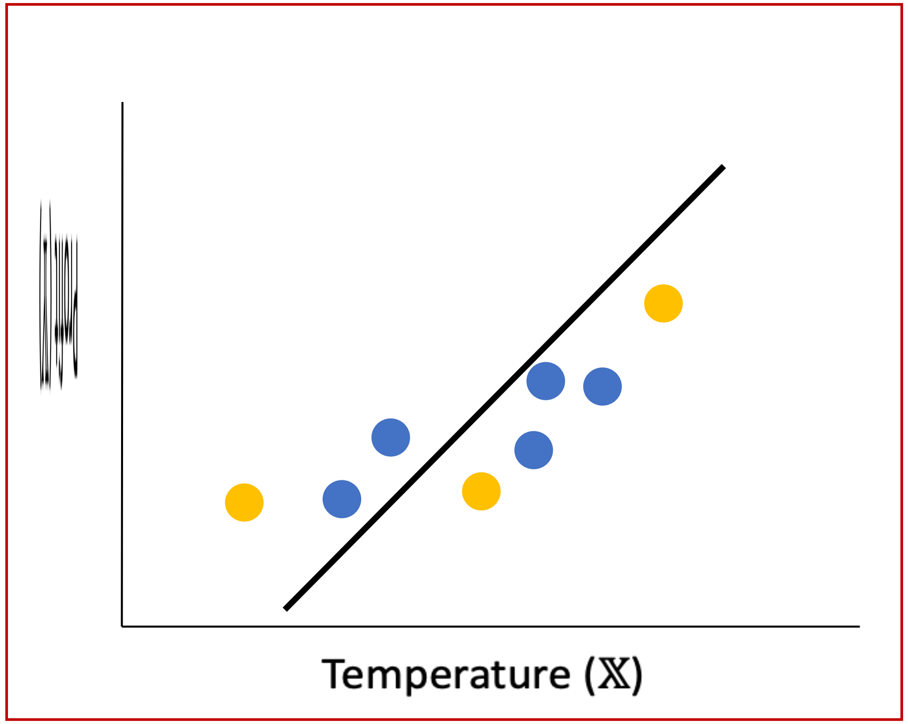
Combination of linear functions is another linear function.

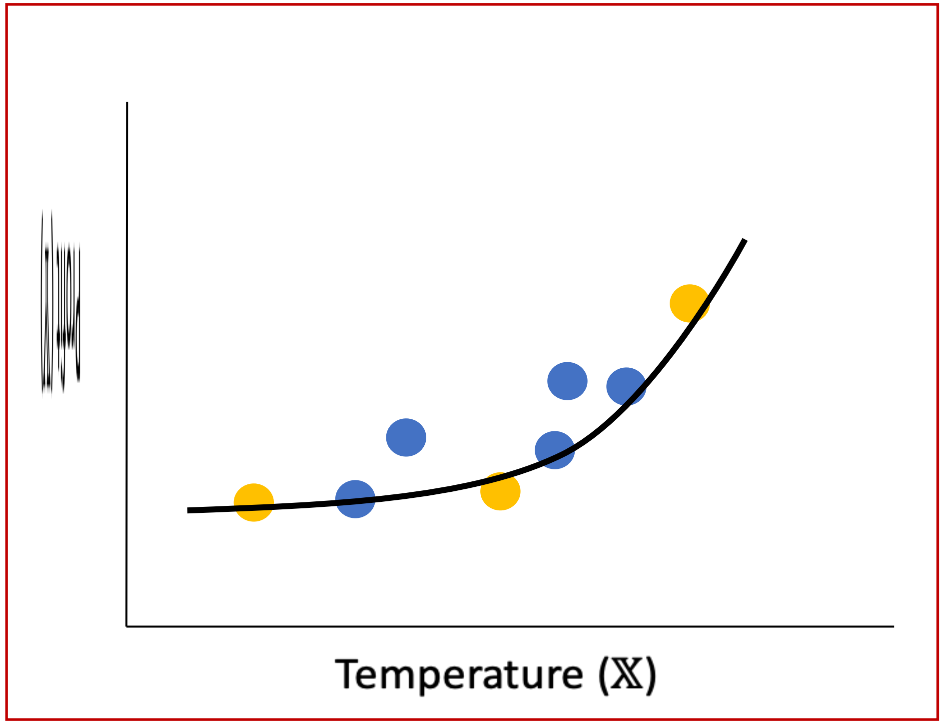
Why is this a problem?

Any questions?

Linear functions may not be sufficient

- Root cause of our problem: a composition of linear functions is still linear

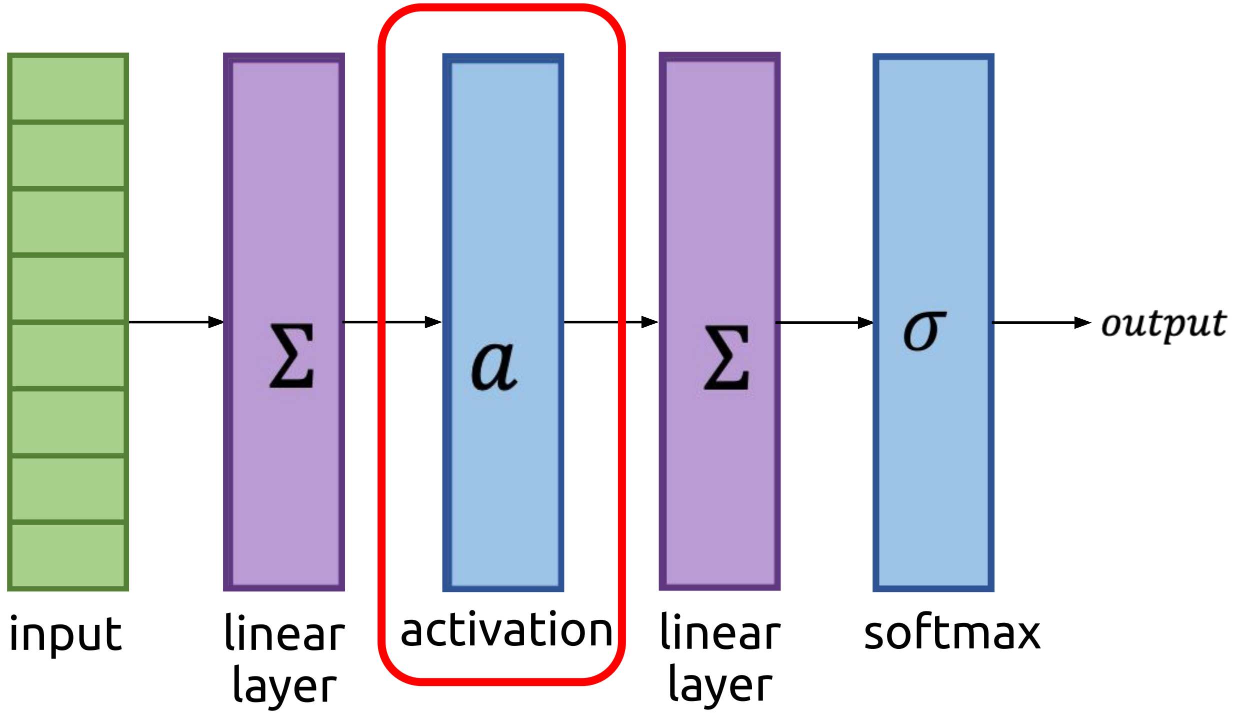





HALP
LOLCATS.COM

Incorporate non-linearity - Activation Functions

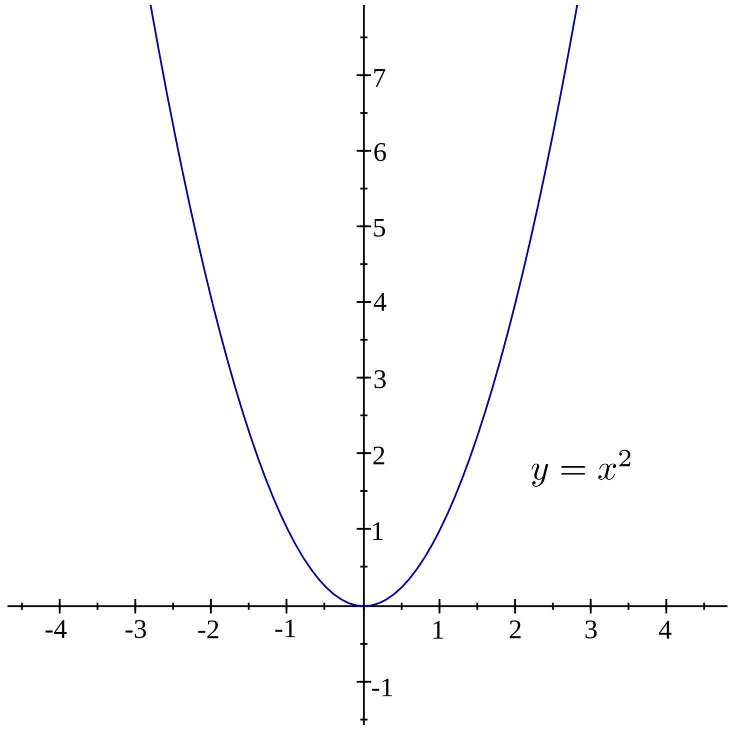
- Root cause of our problem: a composition of linear functions is still linear
- Need some kind of *nonlinear* function between each linear layer.
- Called an ***activation function***
  - Origin of the name: a neuron “activates” if it gets enough electrochemical input



What is a good activation function?

Can you think of a simple non-linear function?

- How about $a(x) = x^2$?
  - Linear → Quadratic
  - Let's examine the consequences of this design decision
  - In particular, let's look at what happens to the gradient



Recall: Single-layer network gradient

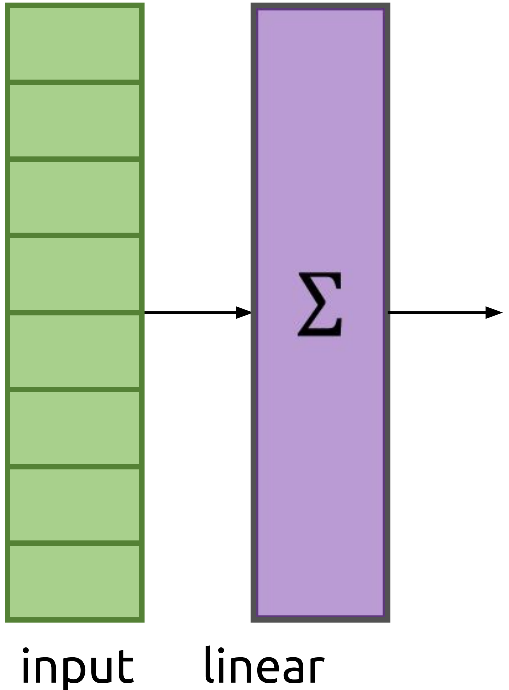
- Let’s look at the partial derivative of logits $\frac{\sigma\iota_j}{\partial w_{j,i}}$

Recall: Single-layer network gradient

- Let's look at the partial derivative of logits $\frac{\partial l_j}{\partial w_{j,i}}$

Recall:

$$l_j = W_{j,0}x_0 + W_{j,1}x_1 + \cdots + W_{j,k}x_k + b_j$$
$$= \sum_k W_{j,k}\, x_k + b_j$$



Recall: Single-layer network gradient

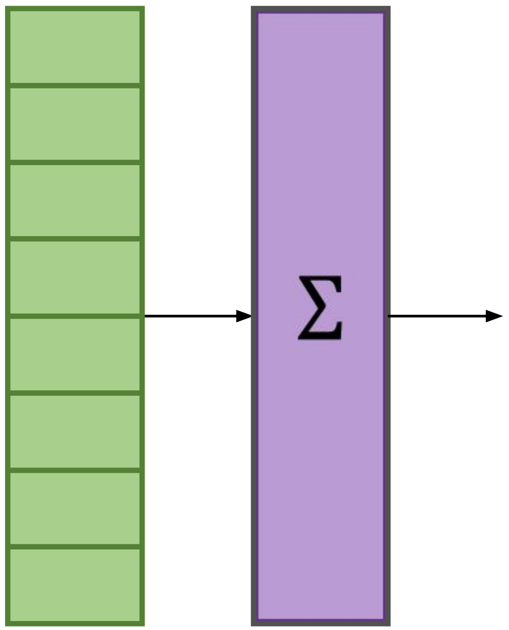
- Let's look at the partial derivative of logits $\frac{\partial l_j}{\partial w_{j,i}}$

Recall:

$$l_j = W_{j,0}x_0 + W_{j,1}x_1 + \cdots + W_{j,k}x_k + b_j$$
$$= \sum_k W_{j,k}\, x_k + b_j$$

So:

$$\frac{\partial \sum_k W_{j,k}\, x_k + b_k}{\partial w_{j,i}} = x_i$$



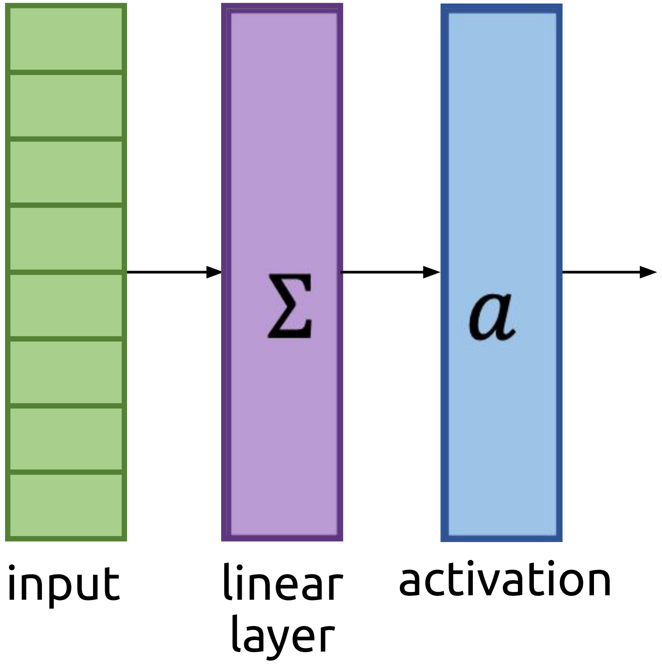
input linear

Now add our activation function

-

Let $a(l_j)$ or $a_j = (l_j)^2$

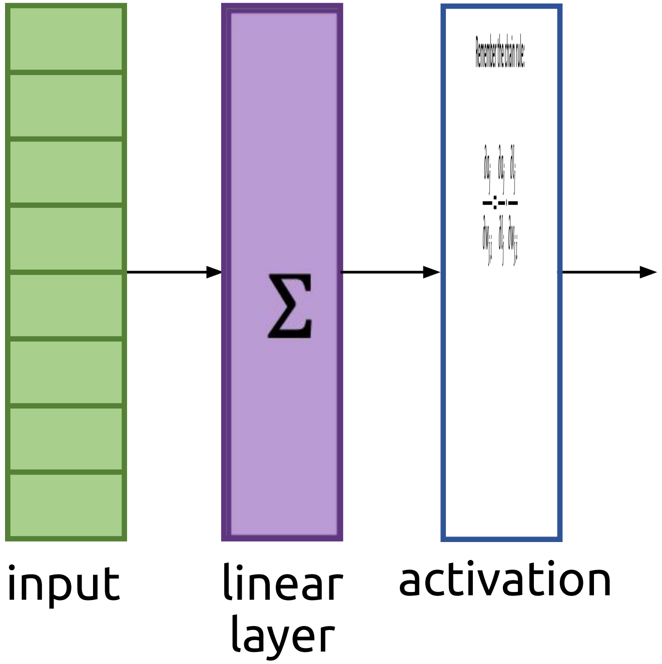
Our goal is to calculate $\frac{\partial a_j}{\partial w_{j,i}}$



Now add our activation function

- Remember the chain rule:

$$\frac{\partial a_j}{\partial w_{j,i}} = \frac{\partial a_j}{\partial l_j} \cdot \frac{\partial l_j}{\partial w_{j,i}}$$

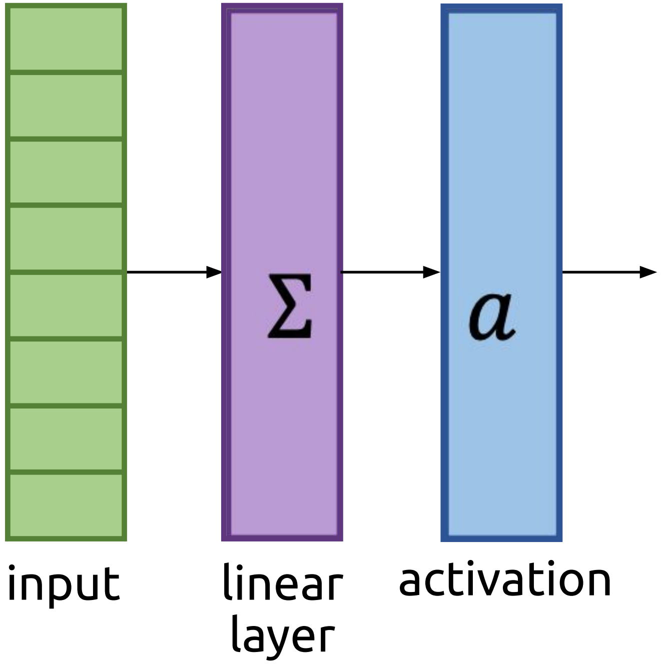


Now add our activation function

-

$$\frac{\partial a_j}{\partial w_{j,i}} = \frac{\partial a_j}{\partial l_j} \cdot \frac{\partial l_j}{\partial w_{j,i}}$$

$$\frac{\partial a_j}{\partial w_{j,i}} = \frac{\partial \left(l_j\right)^2}{\partial l_j} \cdot x_i$$



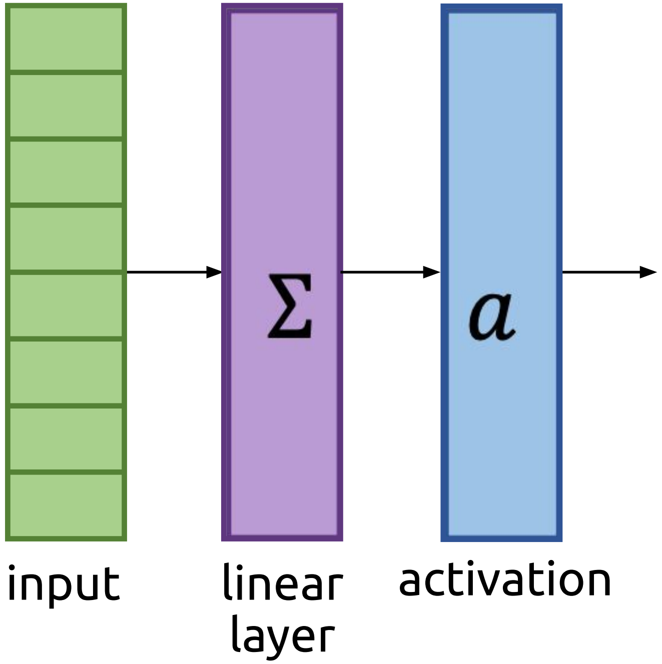
Now add our activation function

•

$$\frac{\partial a_j}{\partial w_{j,i}} = \frac{\partial a_j}{\partial l_j} \cdot \frac{\partial l_j}{\partial w_{j,i}}$$

$$\frac{\partial a_j}{\partial w_{j,i}} = \frac{\partial (l_j)^2}{\partial l_j} \cdot x_i$$

$$\frac{\partial a_j}{\partial w_{j,i}} = 2l \cdot x_i$$



Uh oh, we have a problem...

- Previous Gradient

$$\frac{\partial l_j}{\partial w_{j,i}} = x_i$$

- New Gradient

$$\frac{\partial a_j}{\partial w_{j,i}} = 2l \cdot x_i$$

New gradient is, in general, ***larger*** in magnitude
With more layers, gradient gets bigger and bigger...

Known as the ***Exploding Gradient Problem***

Consequences of Exploding Gradients

Remember the update rule for SGD:

$$\Delta w_{j,i} = -\alpha \cdot \frac{\partial L}{\partial w_{j,i}}$$

So if our gradient gets really big, we need a very small learning rate α

$a(x) = x^2$: ***Not*** a good activation function!

Any questions?


HALP

The Sigmoid Activation Function

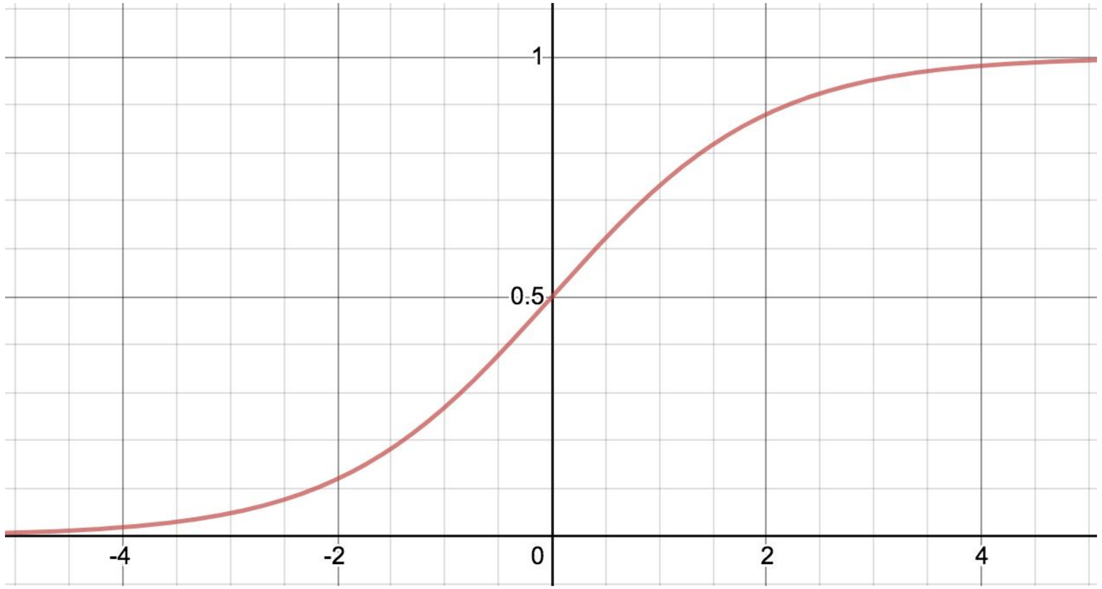
Have I mentioned this function before?

•

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

The Sigmoid Activation Function

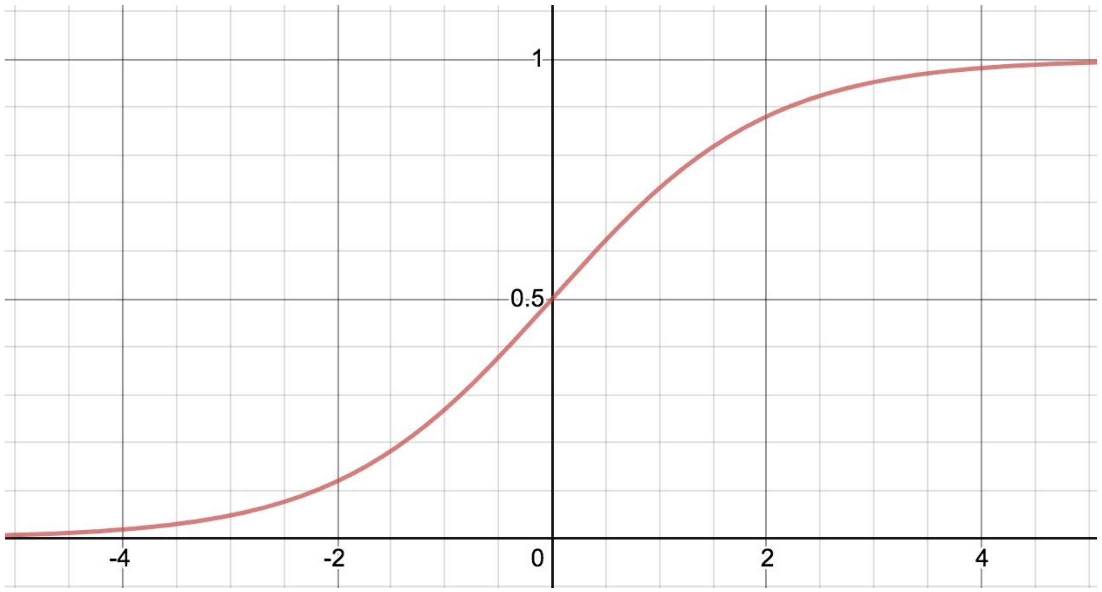
- Historically very popular activation function
- **Takes real value and squashes it to range between 0 and 1**
  - i.e. $\sigma(x)\colon\ \mathbb{R} \rightarrow (0,1)$



$$\sigma(x) = \frac{1}{1+e^{-x}}$$

The Sigmoid Activation Function

- Large negative numbers become 0 and large positive numbers become 1
- **Bounded**: guarantees gradient cannot grow without bound!!



$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Another “Sigmoidal” function: Tanh

•

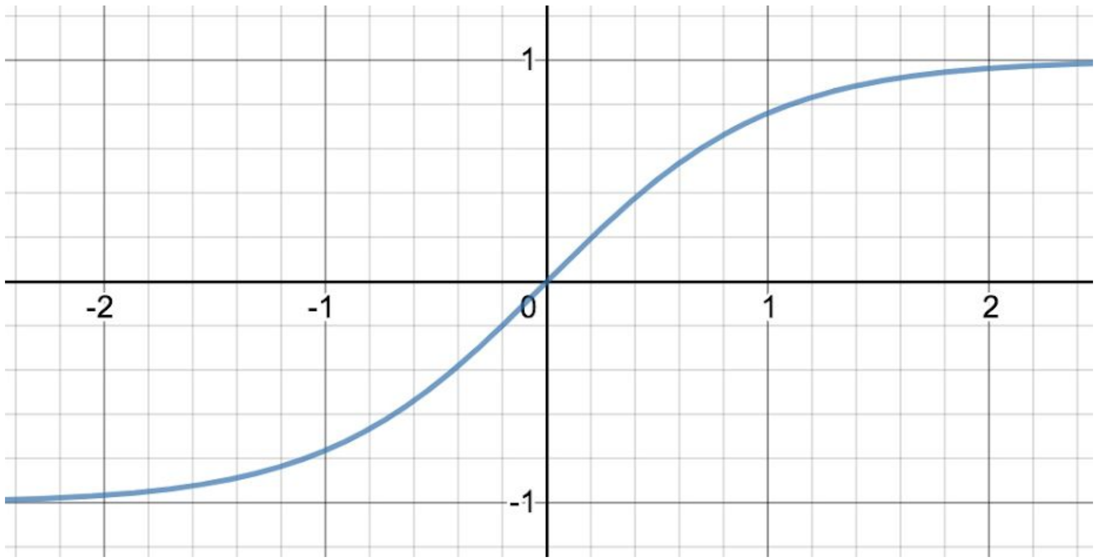
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^x}$$

The hyperbolic tangent function

Tanh

Do you see any issues with these functions? (Think about the gradients!)

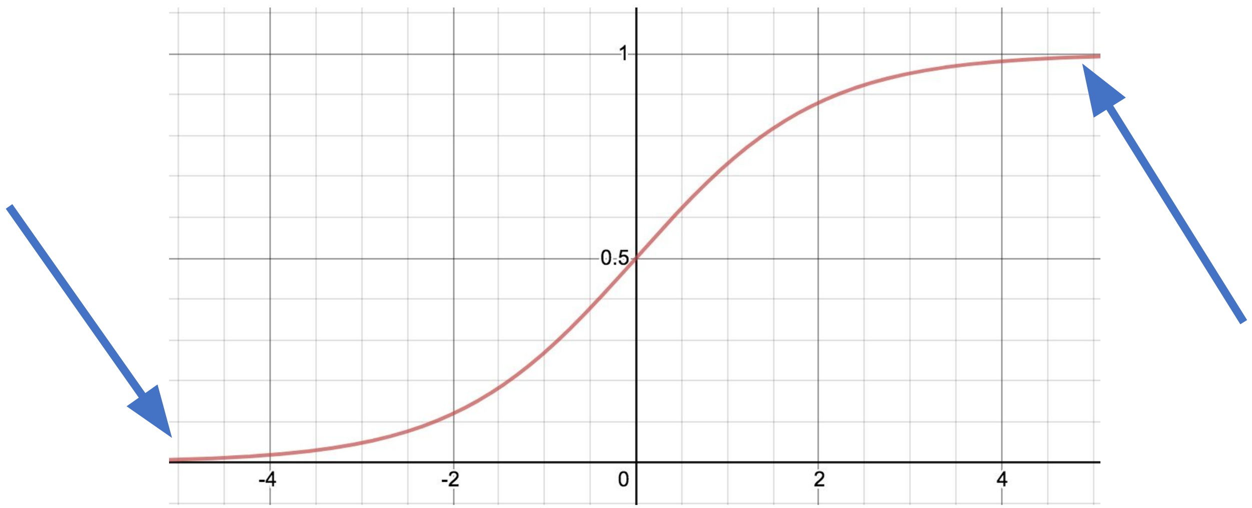
- Output range: $[-1,1]$
  - Versus sigmoid $[0,1]$
- Somewhat desirable property of keeping the signal that passes through the network “centered” around zero.



$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{x}}$$

But we're still not out of the woods...

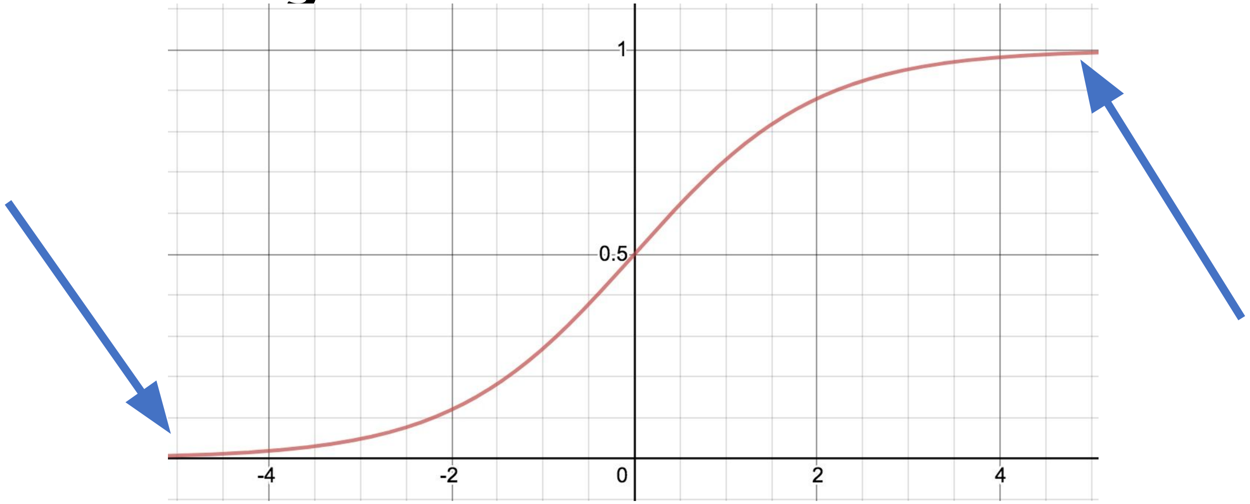
- The bounded-ness of these functions is a double-edged sword
  - Why? Being bounded means that the function has ***asymptotes***, which have zero derivative in the limit.



$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

But we’re still not out of the woods...

- So, our derivatives don’t grow out of control...
- ...but the price we pay is that they approach zero, and ***the network stops learning***
- Known as the ***Vanishing Gradient Problem***



$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Consequences of Vanishing Gradients

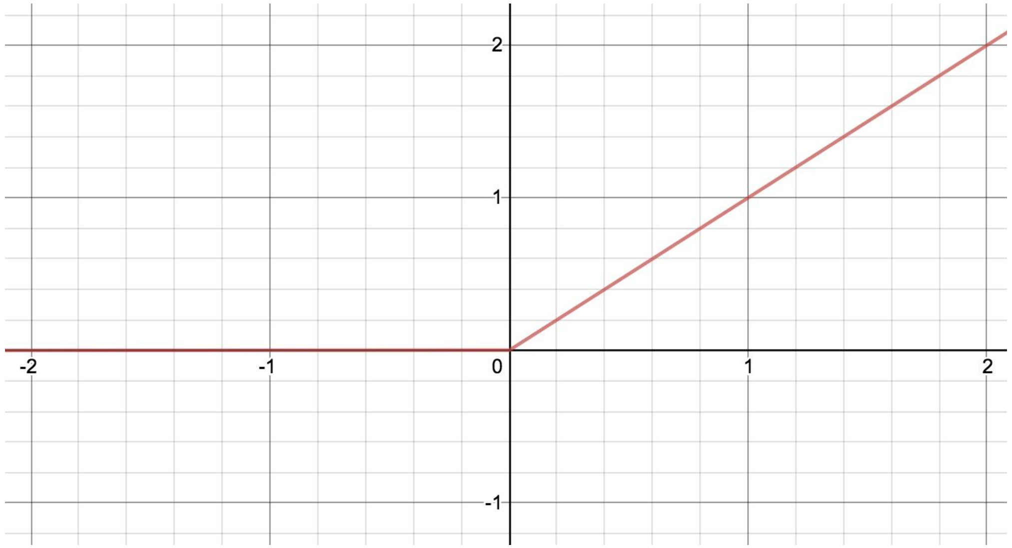
- Problem is exacerbated by stacking multiple layers (gradients shrink more the deeper you go)
- Led to the belief that in practice, neural nets could only ever be a few layers deep...

Any questions?


halp!!
quickmeme.com

Enter the ***Rectifier Function***

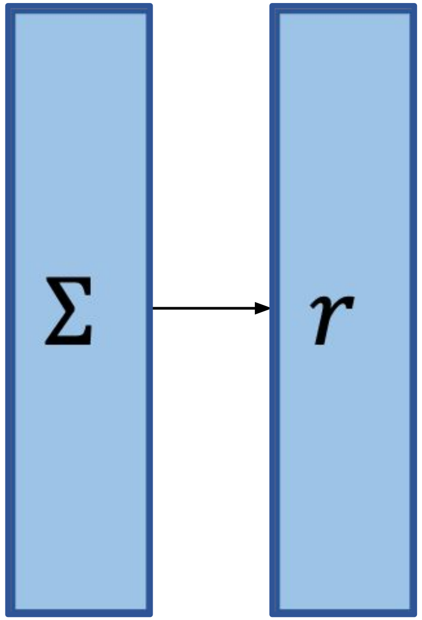
- Nonlinear — cannot be represented as: $a(x) + b$



$$f(x) = \begin{cases} x, & x > 0 \\ 0, & else \end{cases}$$

More commonly known as ReLU

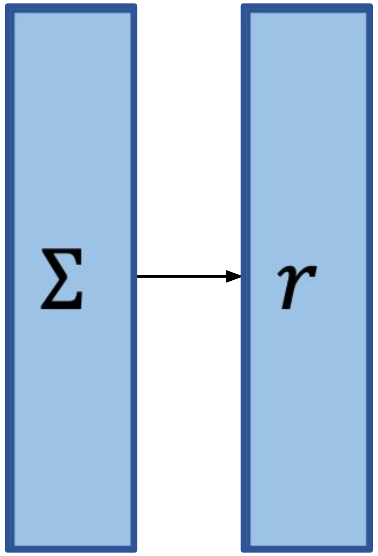
- ***Re***ctified ***L***inear ***U***nit
  - Technically: Linear layer followed by the rectifier function
  - But in most contexts, you will see the rectifier function called "ReLU"



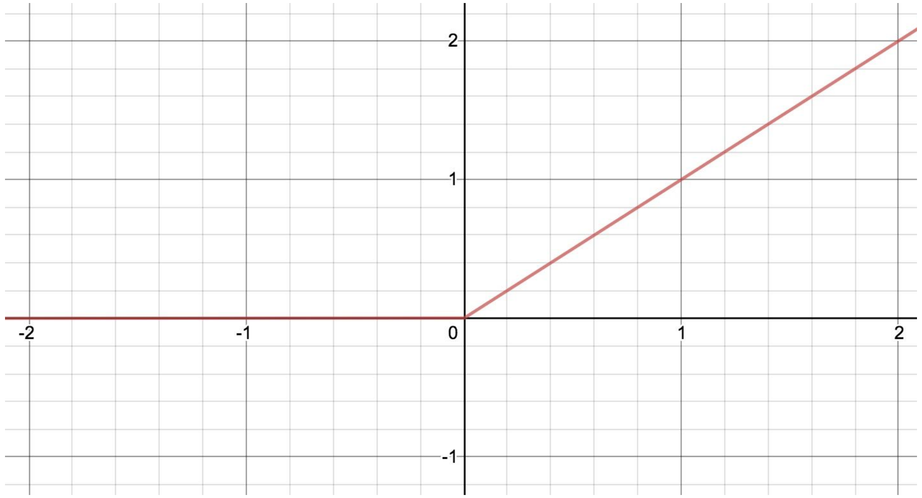
Rectified Linear Unit

Advantages of ReLU

- Does not suffer from vanishing ***or*** exploding gradients!
- Super computationally efficient (avoids the exp calls in sigmoid/tanh)
- Most popular, de-facto ‘standard’ activation function



Rectified Linear Unit



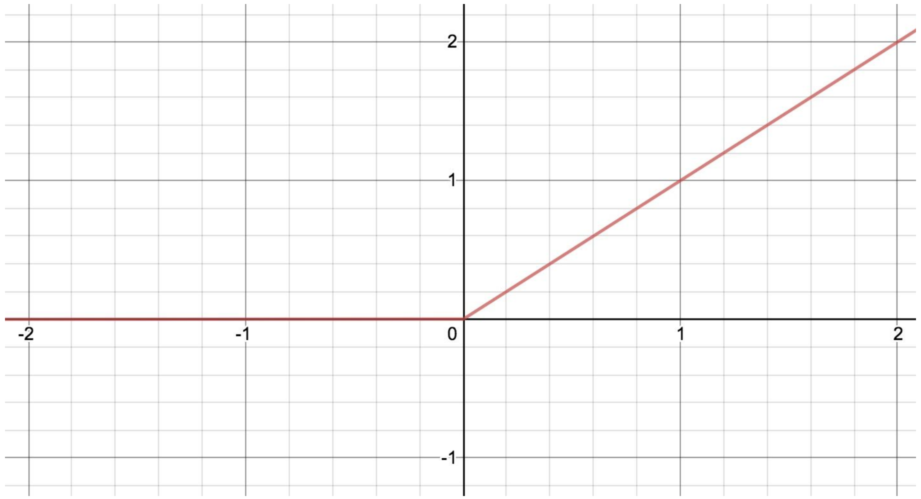
$$\mathrm{ReLU}(x) = \max(0, x)$$

$$f(x) = \begin{cases} x, & x > 0 \\ 0, & else \end{cases}$$

But not even ReLU is perfect...

- We said that the zero-derivative asymptotes of sigmoid were a problem...

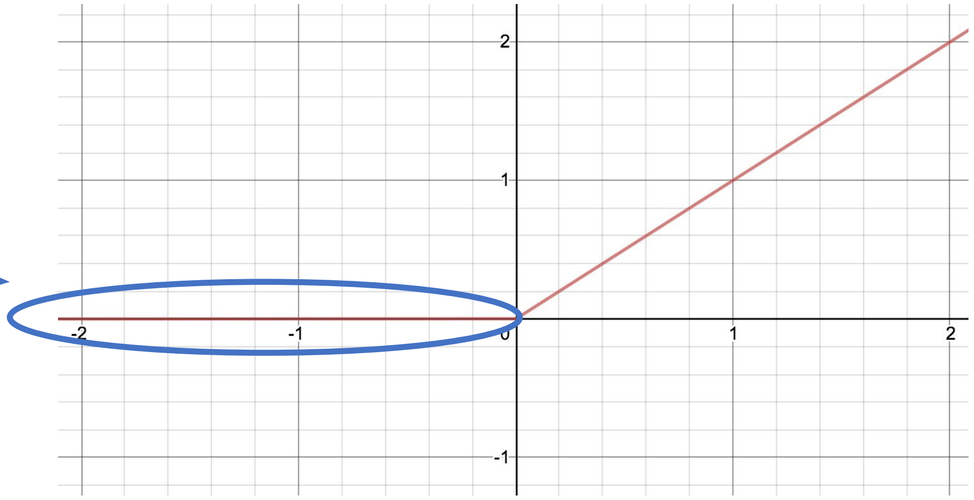
Do we see any issues here?



$$f(x) = \begin{cases} x, & x > 0 \\ 0, & else \end{cases}$$

But not even ReLU is perfect...

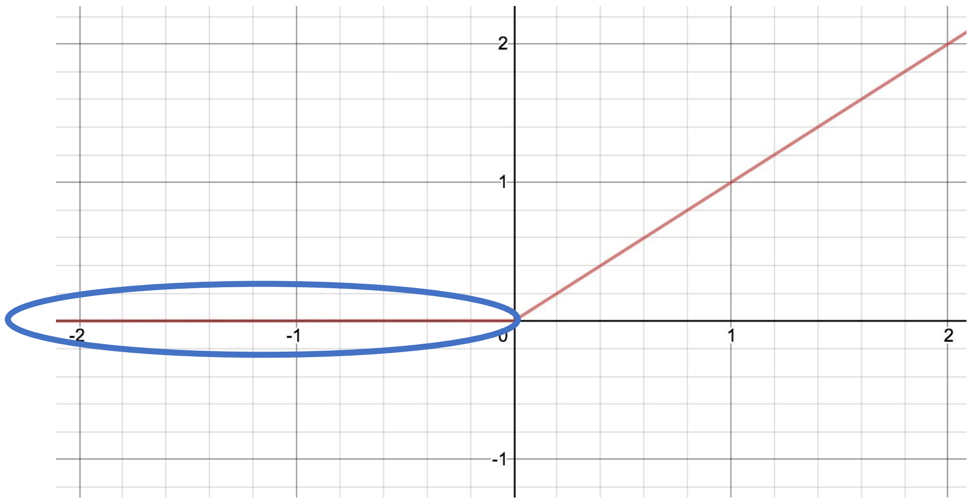
- We said that the zero-derivative asymptotes of sigmoid were a problem...
- Check out this huge zero-derivative region
- Effectively: layers that feed into this activation ***don't learn anything*** if they feed negative values



$$f(x) = \begin{cases} x, & x > 0 \\ 0, & else \end{cases}$$

But not even ReLU is perfect...

- Not such a big deal if the previous layer just occasionally produces negative values
  - Some people even claim this as a "feature," in that the resulting 'sparse activations' in the network more closely resemble what the human brain does
- But what if the previous layer ***always*** produces negative values?
- Is this even possible?

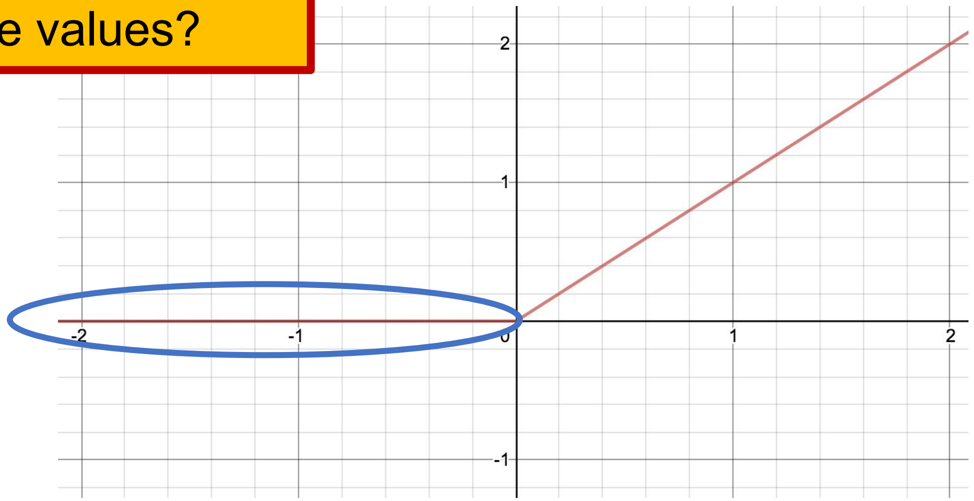


$$f(x) = \begin{cases} x, & x > 0 \\ 0, & else \end{cases}$$

But not even ReLU is perfect...

- The value fed into ReLU:
  - $l_j = \sum_k W_{j,k}\, x_k + b_j$

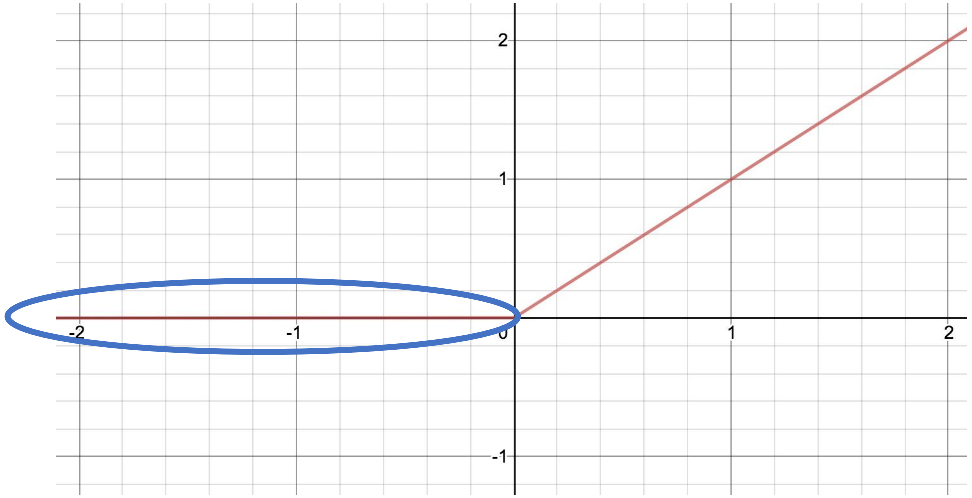
Thinking activity: How could we always get negative values?



$$f(x) = \begin{cases} x, & x > 0 \\ 0, & else \end{cases}$$

But not even ReLU is perfect...

- The value fed into ReLU:
  - $l_j = \sum_k W_{j,k} x_k + b_j$
- If our inputs x_k are bounded (e.g. [0,1]), then the following is possible:
  - The weights have small magnitude
  - The bias is a large negative number
- In this case, l_j will always be negative!

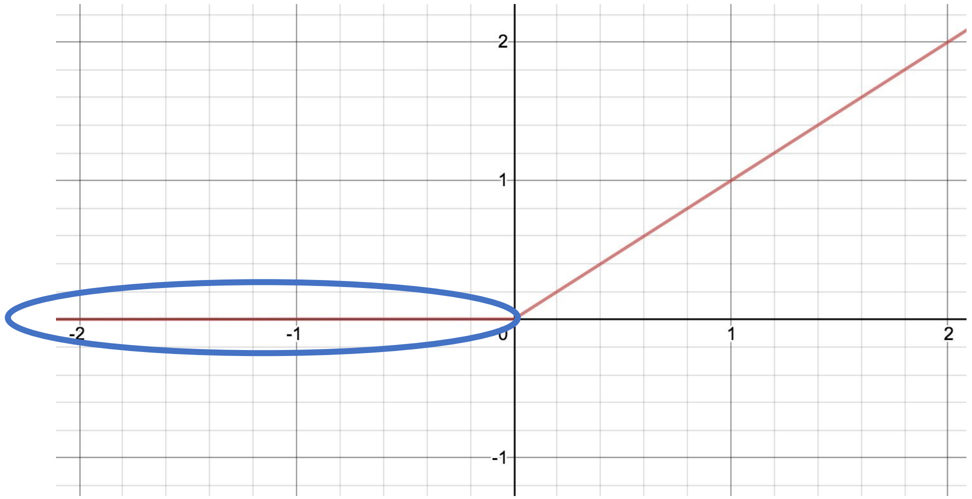


$$f(x) = \begin{cases} x, & x > 0 \\ 0, & else \end{cases}$$

Known as the ***Dead ReLU problem***

But not even ReLU is perfect...

- Does this ever happen in practice?
  - Yes! A large gradient update can 'accidentally' knock the parameters into a state where this happens.
  - Known cases where as much as 40% of the network suffers from this



$$f(x) = \begin{cases} x, & x > 0 \\ 0, & else \end{cases}$$

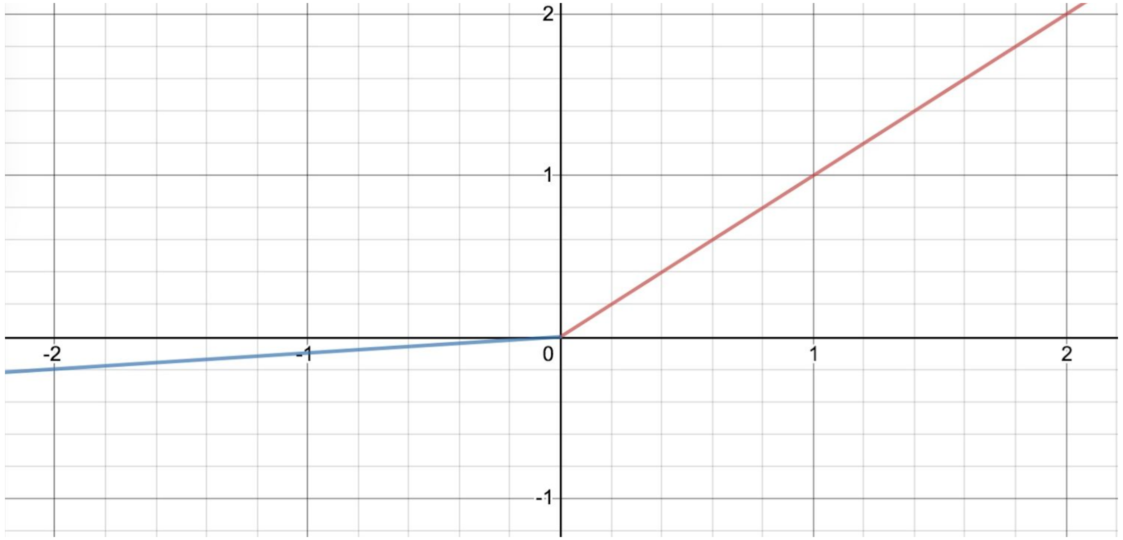
Known as the ***Dead ReLU problem***


HALP
LOLCATS.COM

Leaky ReLU

- Fix — we give a tiny positive slope for negative inputs
- Some activation "leaks" through the barrier

$$f(x) = \begin{cases} x, & x > 0 \\ ax, & else \end{cases}$$



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

Other Activation Functions

Why use other activation functions?

Softplus

CLASS `torch.nn.Softplus(beta=1, threshold=20)` [SOURCE]

Applies the element-wise function:

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

SoftPlus is a smooth approximation to the ReLU function and can be used to constrain the output of a machine to always be positive.

For numerical stability the implementation reverts to the linear function for inputs above a certain value.

Hardshrink

CLASS `torch.nn.Hardshrink(lambd=0.5)` [SOURCE]

Applies the hard shrinkage function element-wise:

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

LogSigmoid

CLASS `torch.nn.LogSigmoid` [SOURCE]

Applies the element-wise function:

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

CELU

CLASS `torch.nn.CELU(alpha=1.0, inplace=False)` [SOURCE]

Applies the element-wise function:

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

More details can be found in the paper Continuously Differentiable Exponential Linear Units .

Parameters

- **alpha** – the α value for the CELU formulation. Default: 1.0
- **inplace** – can optionally do the operation in-place. Default: `False`

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input

Great PyTorch documentation here!

Reasons to use other activation functions

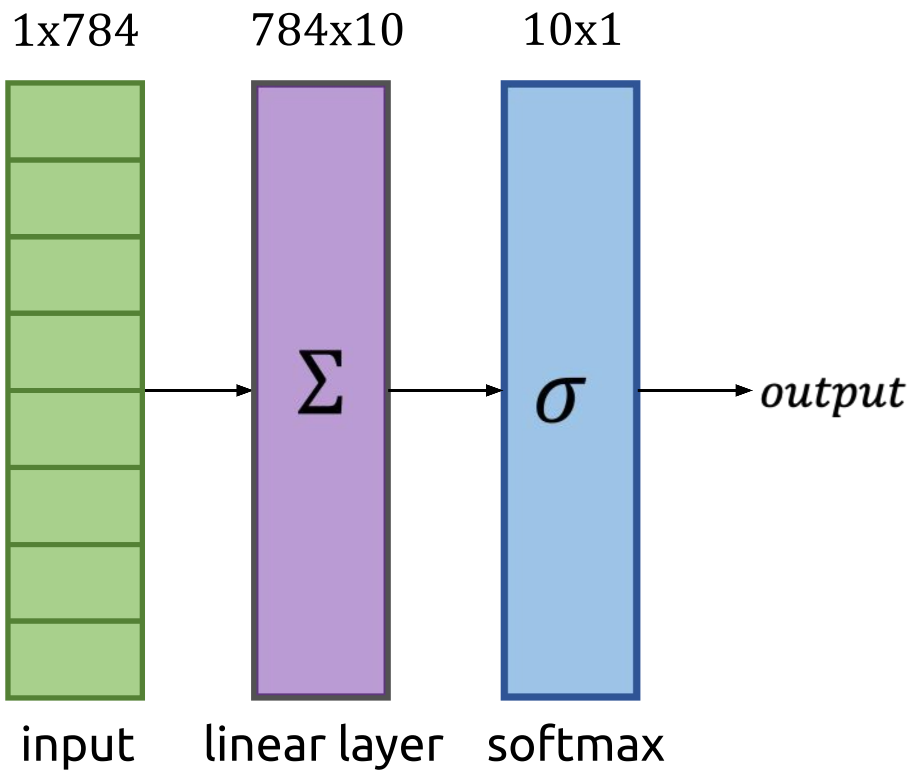
Any questions?

- Bounding network outputs to a particular range
  - Tanh: [-1, 1]
  - Sigmoid: [0,1]
  - Softplus: [0, ∞]

- Example: Predicting a person's age from other biological features
  - Age is a strictly positive quantity
  - We can help our network learn by restricting it to output only positive numbers
  - Use a Softplus activation on the output

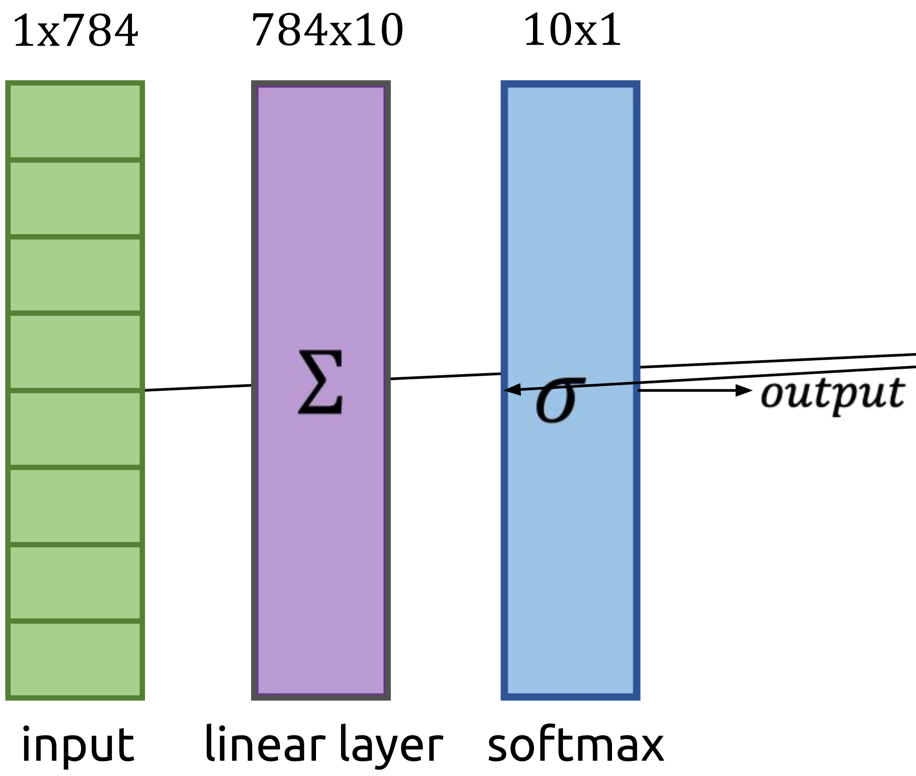
Building a multi-layer network

- Previously:

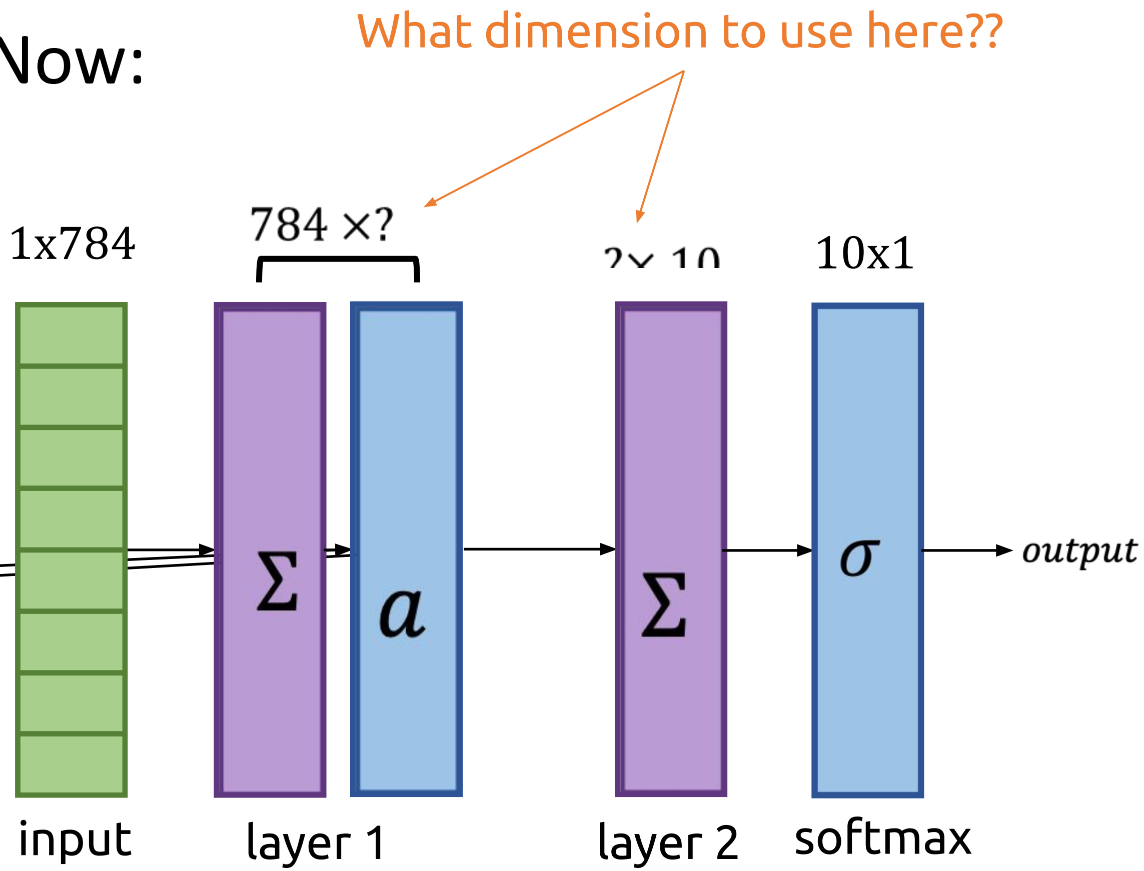


Consequences of adding activation layers

- Previously:



- Now:



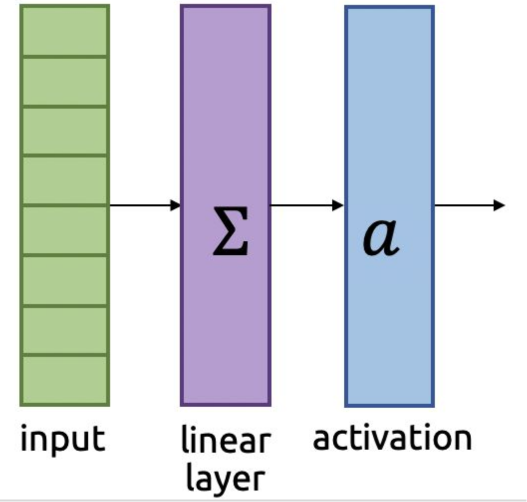
Recap

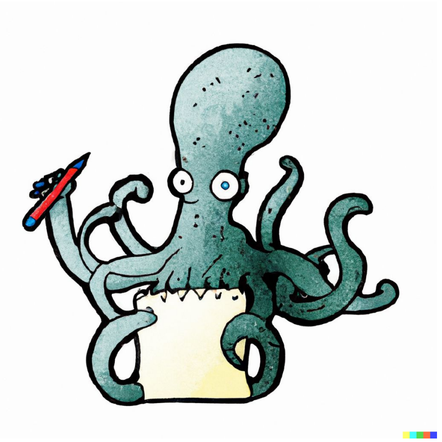
Stacking multiple layers

More layers more complicated function

Linear layers are not sufficient!

Need non-linearity





Activation functions

Exploding gradients

Vanishing gradients

ReLU, Leaky ReLU