Deep Learning

Single Layer Neural Networks

Logistics

- Homework 1 due next week
- Submit on gradescope
- Have you started talking to your classmates about the final project yet?
- Today: Single layer neural networks.
  - Universal Approximation

A few preliminary notations:

- L_2 regularization is the same as weight decay discussed last class: the regularizer is
$$r(x) = \lambda ||x||_2^2$$
- L_1 regularization is the same as the Lasso:
$$r(x) = \lambda ||x||_1$$

Last Time

- SGD: general purpose training algorithm.
  - SGD always reaches points where the gradient is small, as long as the learning rate is set properly.
  - The proper value for the learning rate is usually not known, so in practice we often either guess or use some more complicated scheme.
    - Last class we considered only constant learning rates. In practice we usually decay the learning rate. This is because the "optimal" constant learning rate depends on the number of iterations.
- Overfitting: if you have too little data, or test too many parameter/hypothesis values, then you might end up over fitting.
  - Combat this with regularization, early stopping.

Neural Networks

xkcd.com

Layers/Modules/Blocks

- "neural network" refers to a family of models or hypothesis classes.
- These models are created by composing different functional "blocks".
- Typical model: $y \approx f(x; \theta)$
- Composed model: $y \approx f(g(x; \theta_g); \theta_f)$
- This can get very complicated:

$$y \approx k([h([f(f(x; \theta_{f_1}); \theta_{f_2}), f(x; \theta_{f_1})]; \theta_h), x]\, ; \theta_k)$$

Why compose?

- Allows for expressivity:
  - Ex. generating all polynomials from quadratics
- $f\big(x,(a,b,c)\big) = a + bx + cx^2$
- $f\Big(f\big(x,(a,b,c)\big),(a',b',c')\Big) = a'' + b''x + c''x^2 + d''x^3 + e''x^4$
- Separation of concerns (just like in programming): different layers will do different things.

A few issues

1. Write/think about this in a more reasonable way.

2. How to train the model?

3. How to pick which functions to compose?

4. Why would anyone subject themselves to all this in the first place?

A few issues

1. Write/think about this in a more reasonable way.

   Next few lectures (and generally throughout the course)

2. How to train the model? SGD.

3. How to pick which functions to compose?

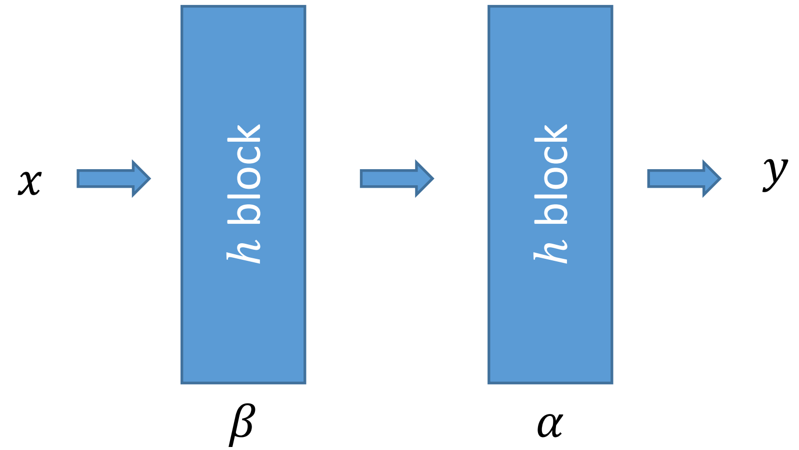
   More of an art than a science. We will cover many examples this semester.

4. Why would anyone subject themselves to all this in the first place?

   It can work really well!

Block Diagrams

- Example: $y = \alpha_0 + \alpha_1(\beta_0 + \beta_1 x^2)^2$
- Can also write as: $y = h(h(x;\beta);\alpha)$ where $h(z;\theta) = \theta_0 + \theta_1 z^2$
- We will use a block diagram or flowchart:



Rules of the Blocks

- Each block can have multiple inputs and outputs
  - Most of the time there is just one (vector valued) input and one (vector valued) output.
- When a block has a single input and output, it is often called a “layer”.
- If a layer is not directly connected to the output, it is called a "hidden layer".

What goes in a block?

- Something easy/efficient to compute (in this class).
- Something I can take a derivative of.
- Something expressive enough to ~~make up for our lack of domain knowledge~~ represent the unknown form of the function you are trying to approximate.

Form of a Typical Block

$$f(x; M, b) = \sigma(Mx + b)$$

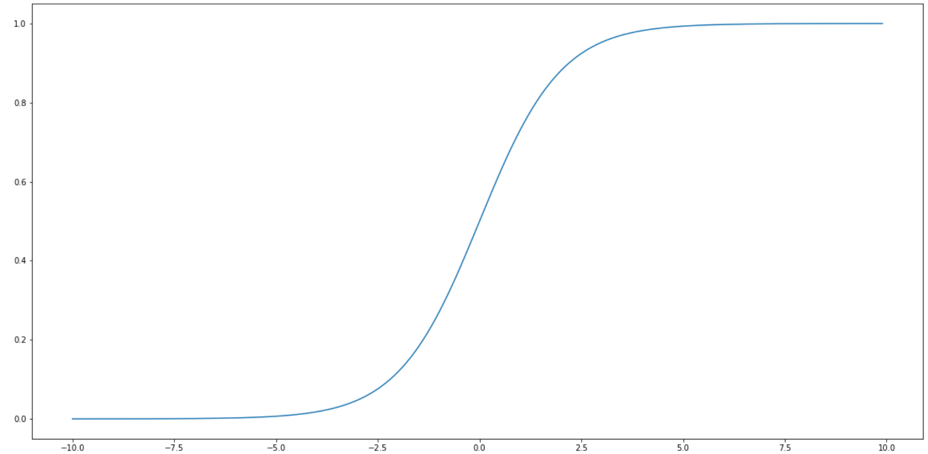
- $x \in \mathbb{R}^d$ is a vector, $M \in \mathbb{R}^{h \times d}$ is a matrix, and $b \in \mathbb{R}^h$ is a vector (called the *bias*).
- x is the *input* of the block, while $\theta = (M, b)$ are the parameters of the block.
- $\sigma: \mathbb{R}^h \to \mathbb{R}^h$ is a *coordinate-wise, non-linear, differentiable*.
  - Coordinate-wise means σ applies a common function to each coordinate.
  - Why is it important the σ be non-linear?
- σ is called the *activation function*.

Sigmoid activation

- A classical choice for σ is $\sigma\colon \mathbb{R}^h \to \mathbb{R}^h$ defined by:

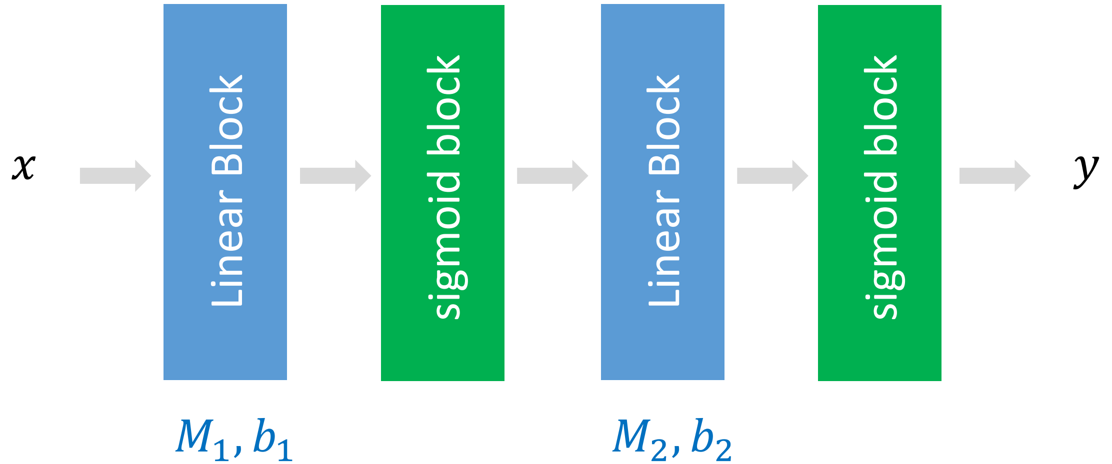
$$\sigma(z)[i] = \frac{\exp(z[i])}{1 + \exp(z[i])}$$

- $\sigma(z)[i]$ indicates the ith coordinate of $\sigma(z)$.
- This is the *sigmoid* activation function.



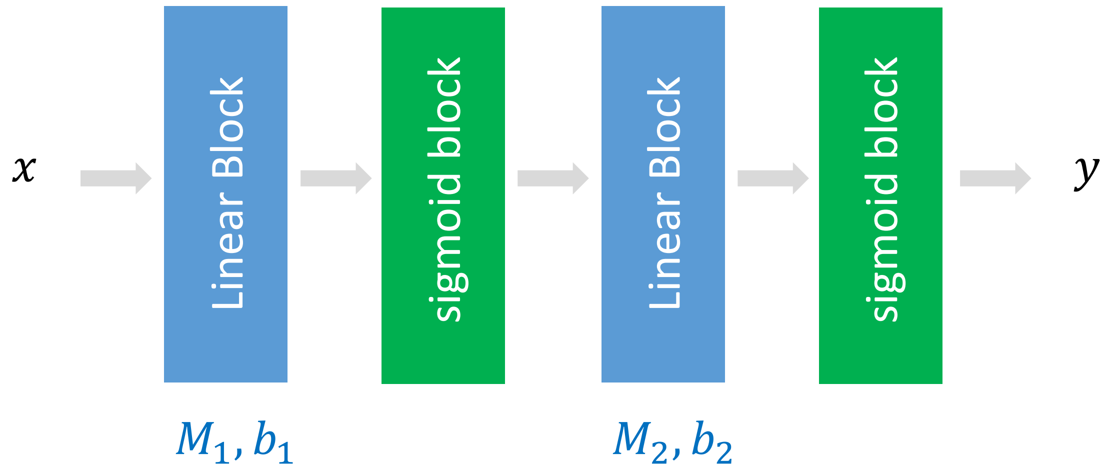
Single Hidden-Layer Network

- Two types of blocks: sigmoid block, and a linear block:
- “Linear block” means linear function, no activation.
- Example: $y = M_2\sigma(M_1 x + b_1) + b_2$

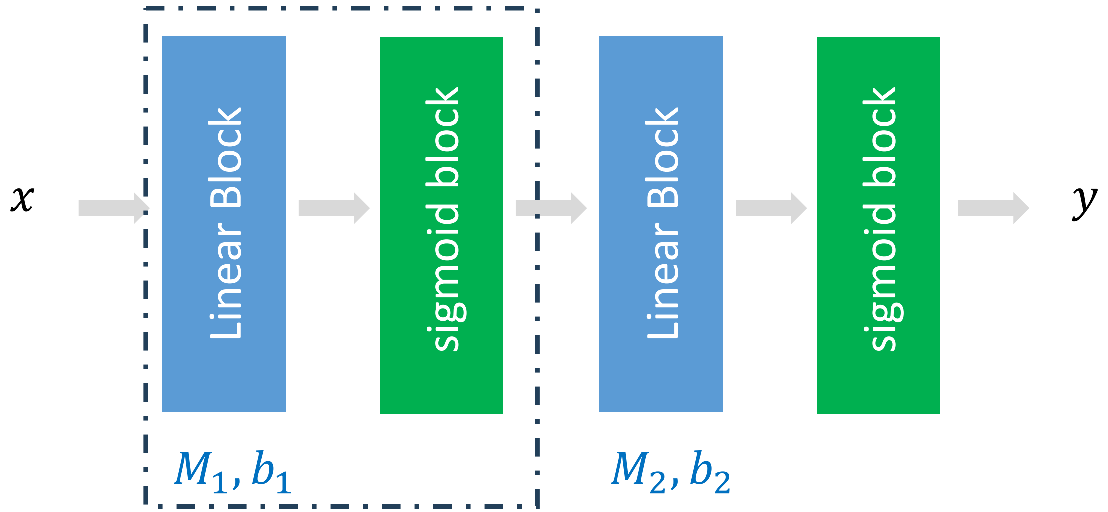


Block Diagrams are not Unique

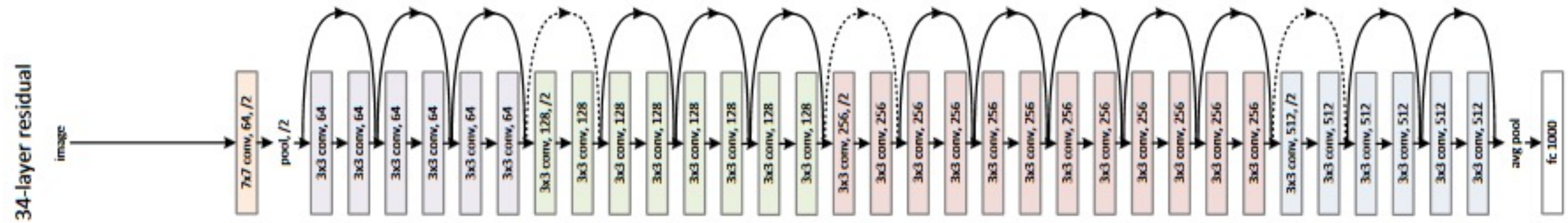
One way:



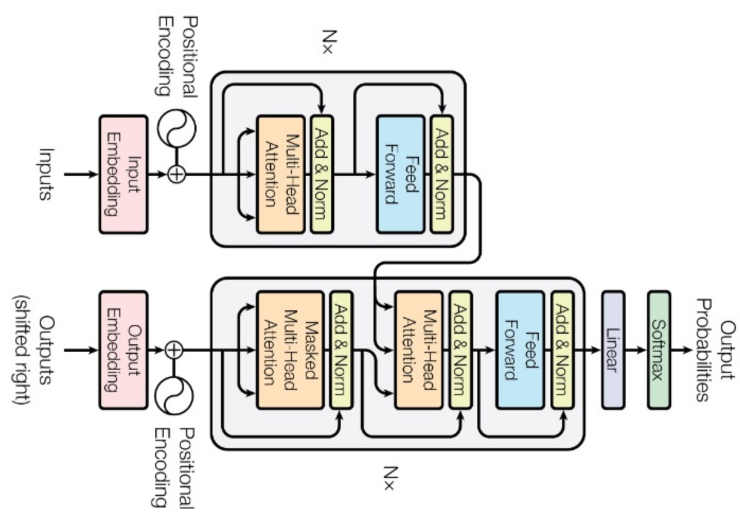
Another way:



Even block diagrams can get complicated…



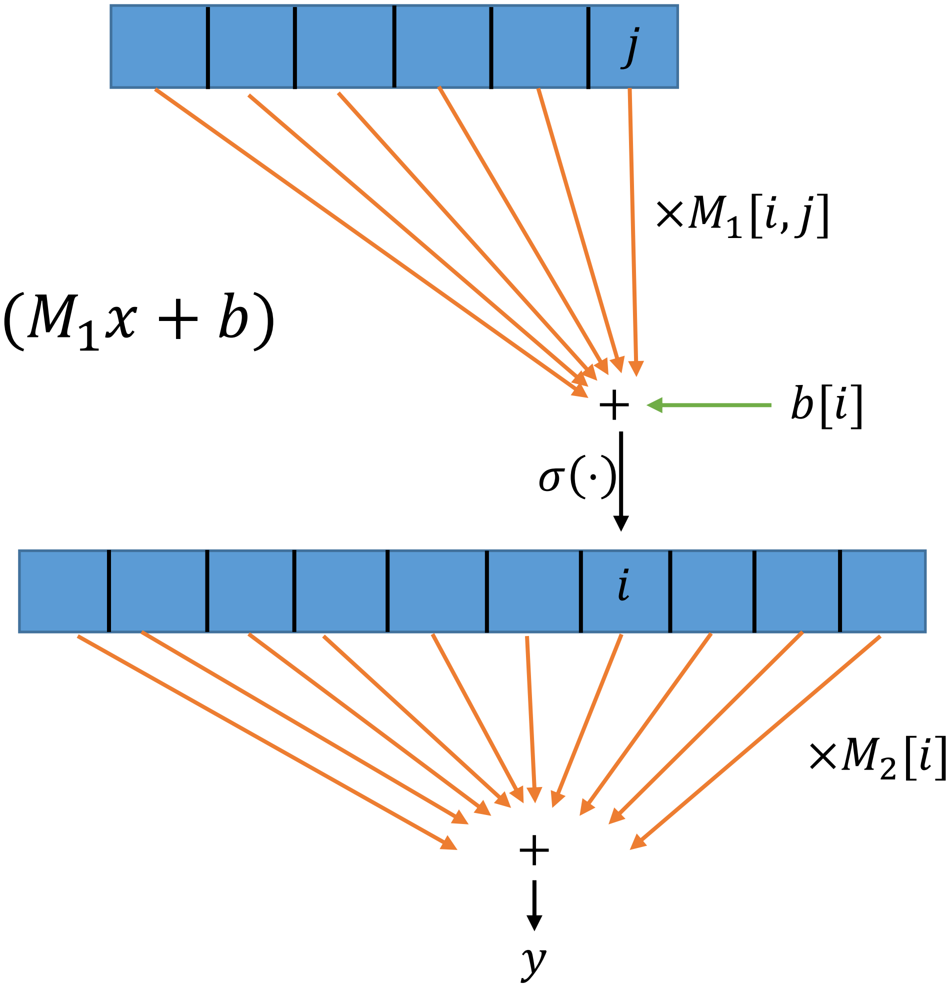
ResNet: A successful image recognition network we will cover later.



Transformer: a successful sequence/text modeling network we will also cover later.

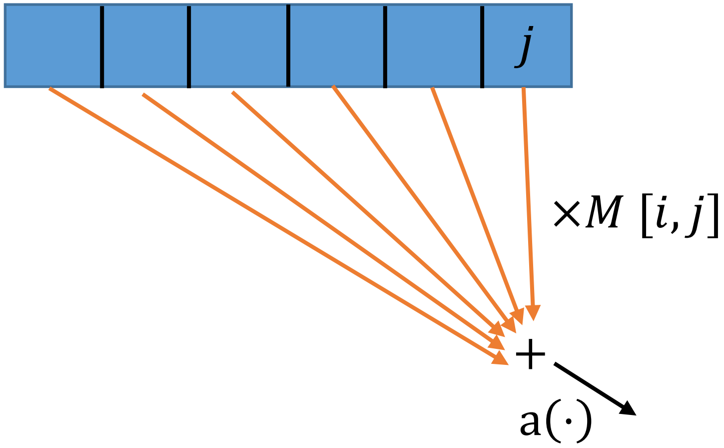
Another Visualization

$$y = M_2\sigma(M_1 x + b)$$



Why this form?

- It's (relatively) easy to write, code, differentiate.
- It is very expressive (more on this in a moment).
- Originally, there is some inspiration from actual biology.
- Each individual entry $\sigma(\sum_i M[i,j]x[i])$ is called a "neuron"



Actual Neurons

- Neuron cells take "inputs" from dendrites and produce "outputs" at the end of the axon.
- The "input" and "output" are electrical voltages. Each dendrite receives a rescaled version of the output of another neuron's axon.
- These dendritic inputs are approximately summed in the soma.
- If this sum is large enough, the neuron produces a large voltage at the end of the axon.

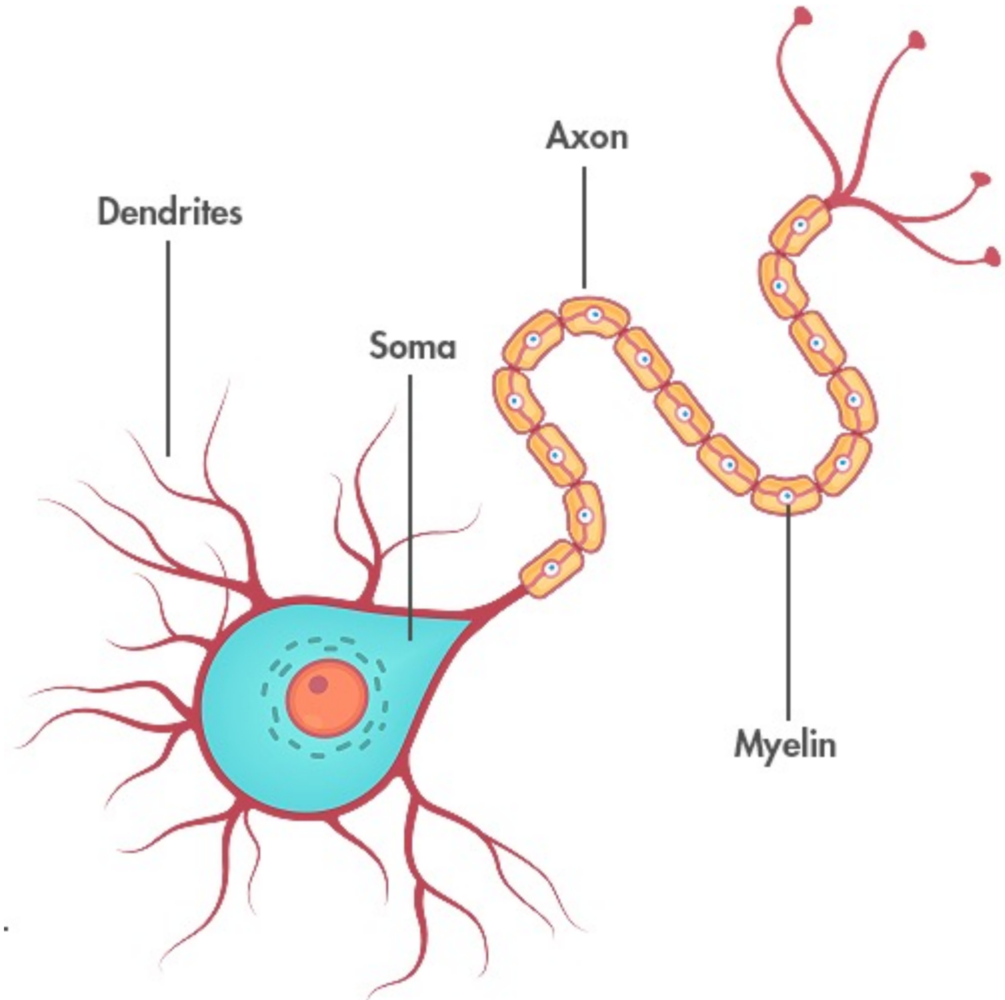
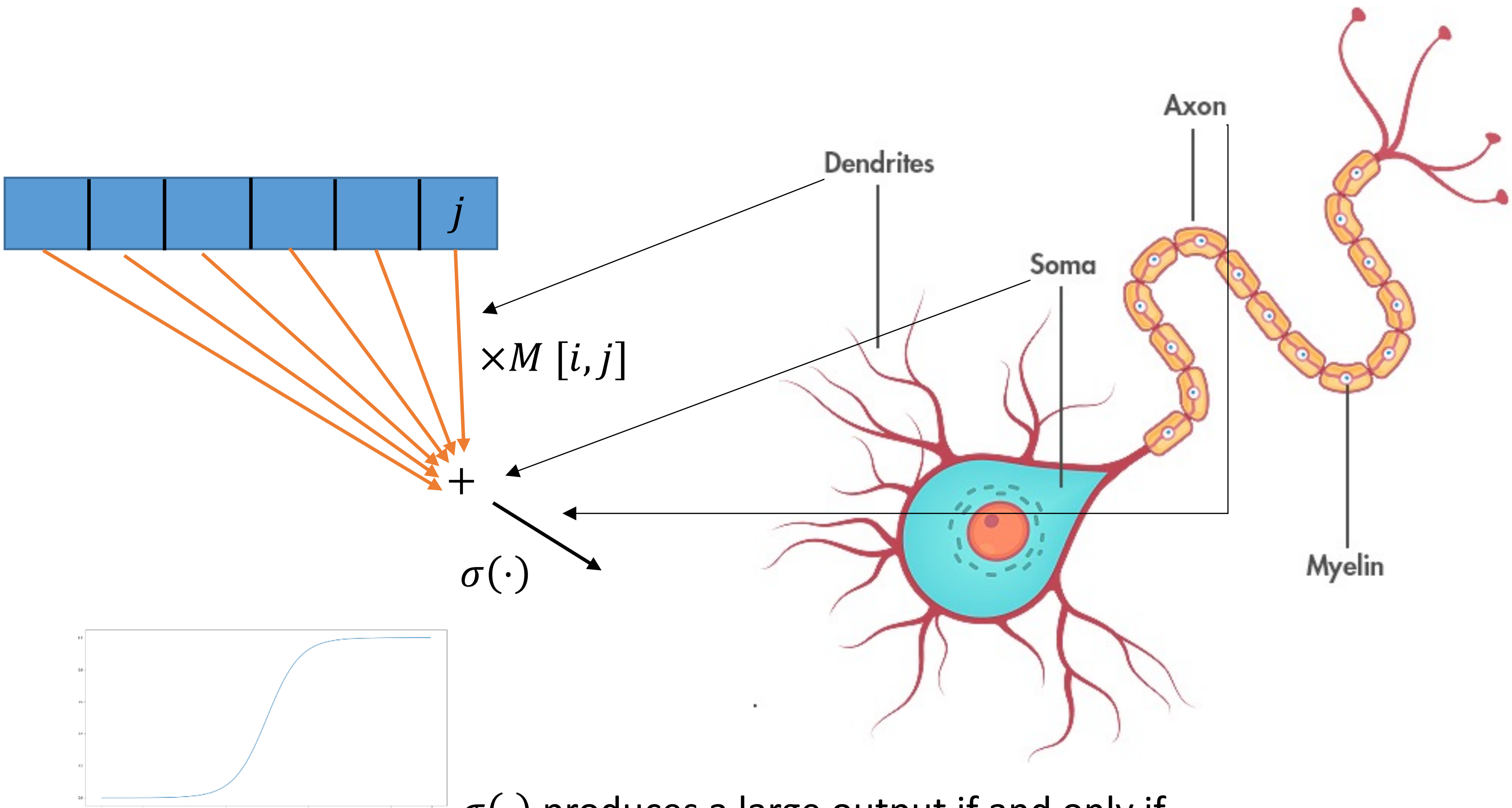


Image: https://www.neuroskills.com/brain-injury/neuroplasticity/neuronal-firing/

Actual Neurons



$\sigma(\cdot)$ produces a large output if and only if the sum of the inputs is large

Some terminology:

- Individual coordinates at the output of a layer are sometimes called "neurons", or "units"
- For any scalar function $\sigma: \mathbb{R} \to \mathbb{R}$, we write an activation function $\sigma: \mathbb{R}^h \to \mathbb{R}^h$ by applying σ to each coordinate of the input.
- The arrangement and types of layers used in a neural network model are called its "architecture"
- The "size" of a layer usually indicates the dimension of the output of the layer.
- This is also called the "width" of the layer.

Width vs Depth

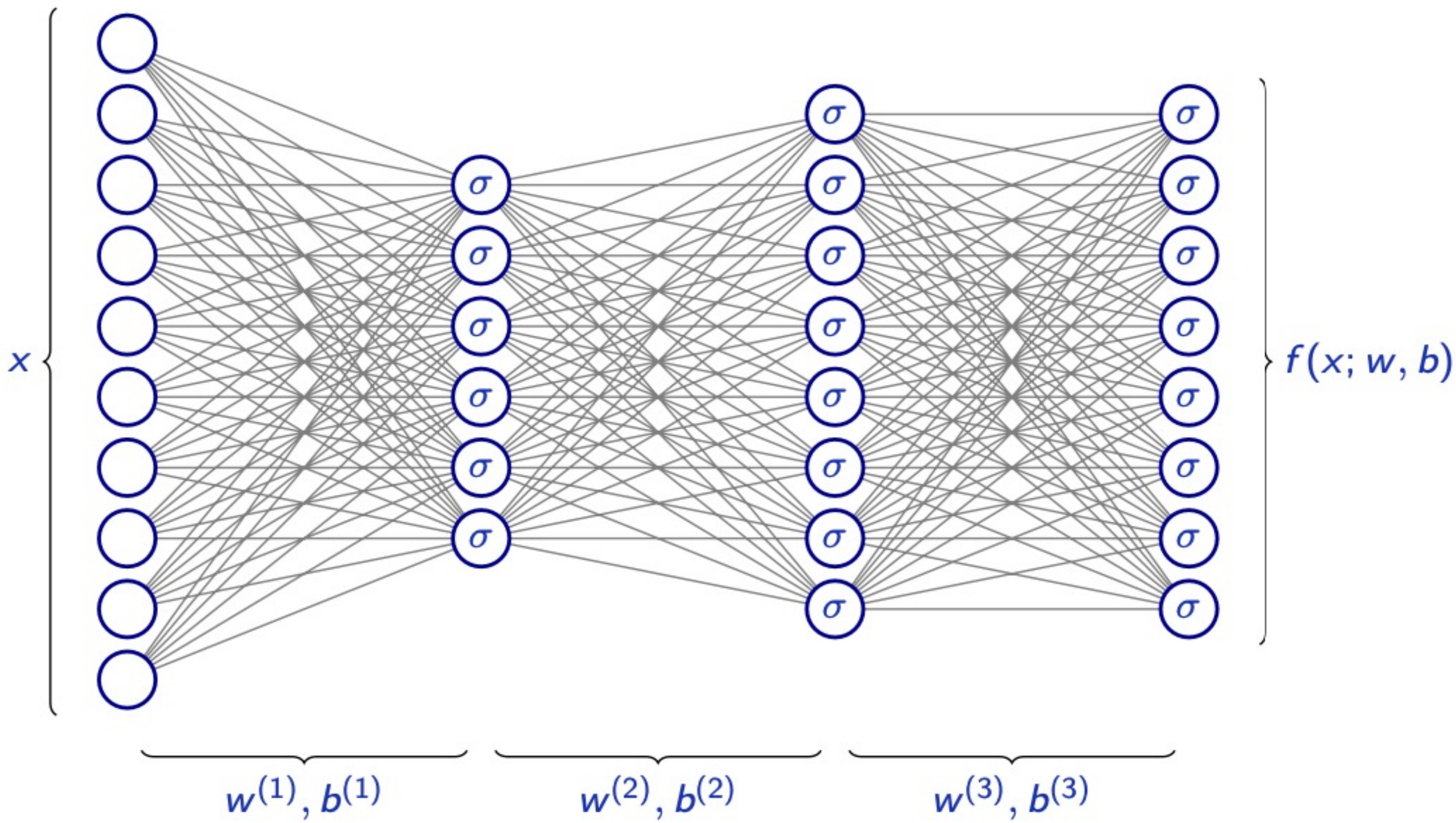


Image credit: Francois Fleuret , lecture 3.4. Multi-Layer Perceptrons

Common Activation Functions

- Sigmoid:
$$\sigma(x) = \frac{\exp(x)}{1 + \exp(x)}$$
- Rectified Linear Unit (ReLU):
$$\sigma(x) = \max(0, x)$$
- Softplus:
$$\sigma(x) = \log(1 + \exp(x))$$
- See https://en.wikipedia.org/wiki/Rectifier_(neural_networks) for a zoo of others.

Universal Approximation

,

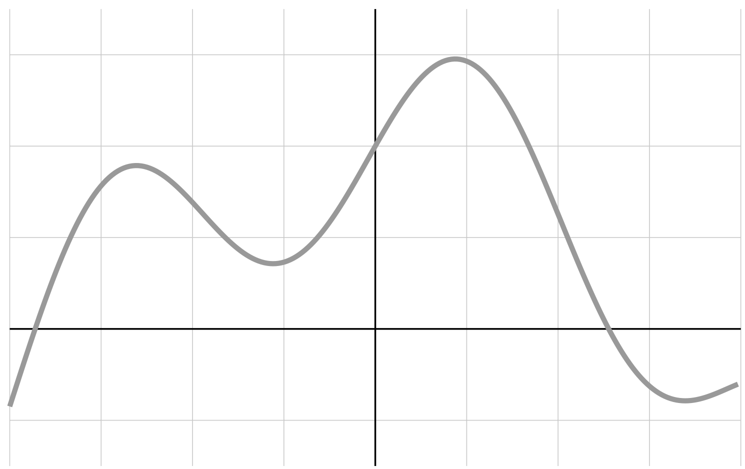


Image credit: Francois Fleuret , lecture 3.4. Multi-Layer Perceptrons

$$f(x) = \sigma(w_1 x + b_1)$$

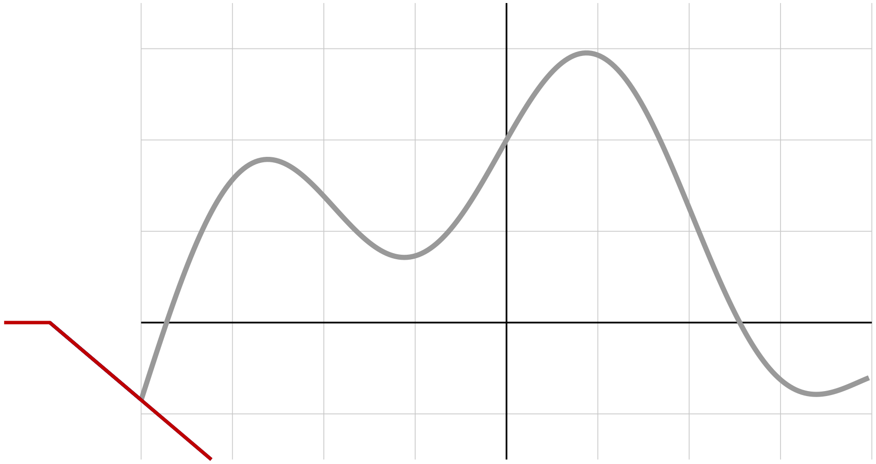


Image credit: Francois Fleuret , lecture 3.4. Multi-Layer Perceptrons

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2)$$

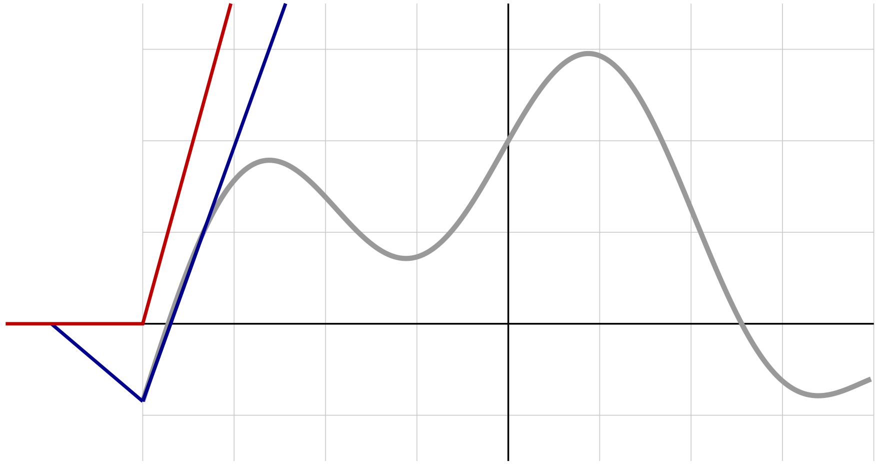


Image credit: Francois Fleuret , lecture 3.4. Multi-Layer Perceptrons

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3)$$

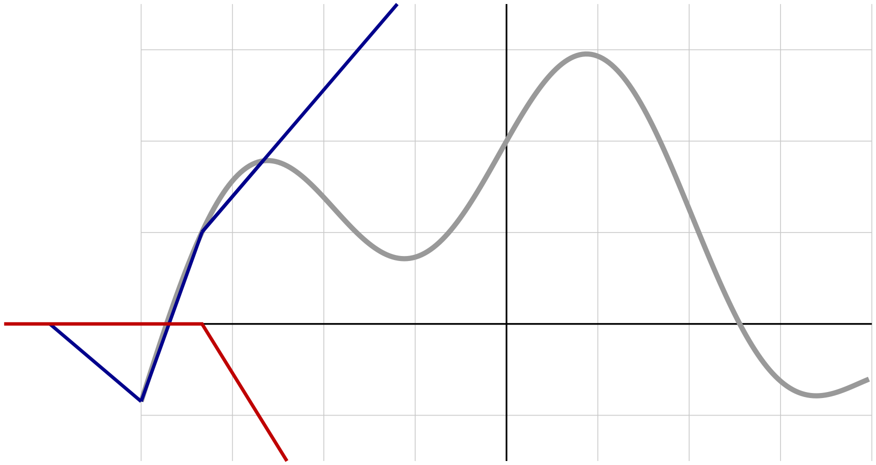


Image credit: Francois Fleuret , lecture 3.4. Multi-Layer Perceptrons

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \dots$$

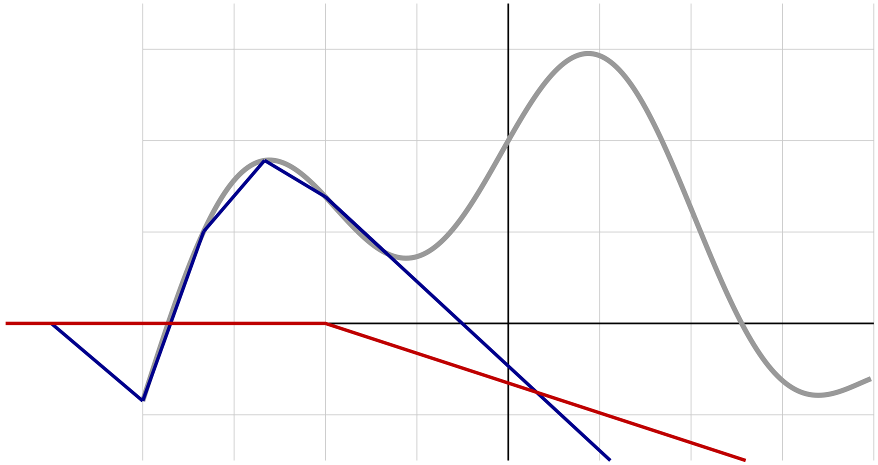


Image credit: Francois Fleuret , lecture 3.4. Multi-Layer Perceptrons

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \dots$$

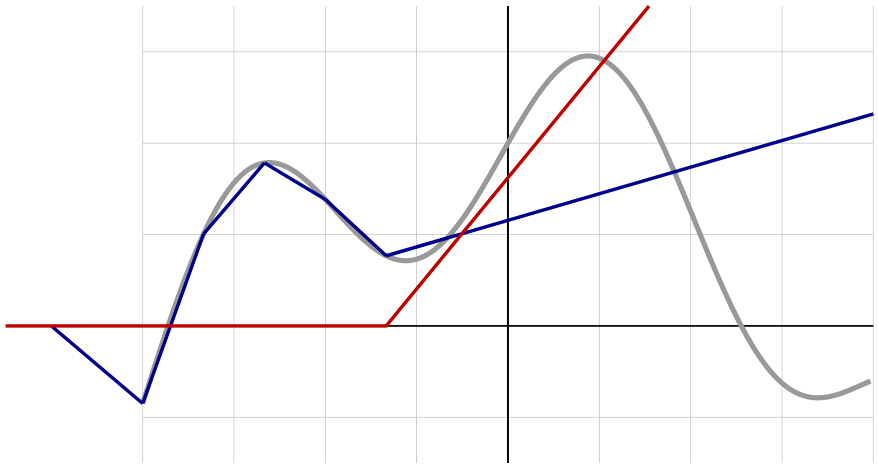


Image credit: Francois Fleuret , lecture 3.4. Multi-Layer Perceptrons

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \dots$$

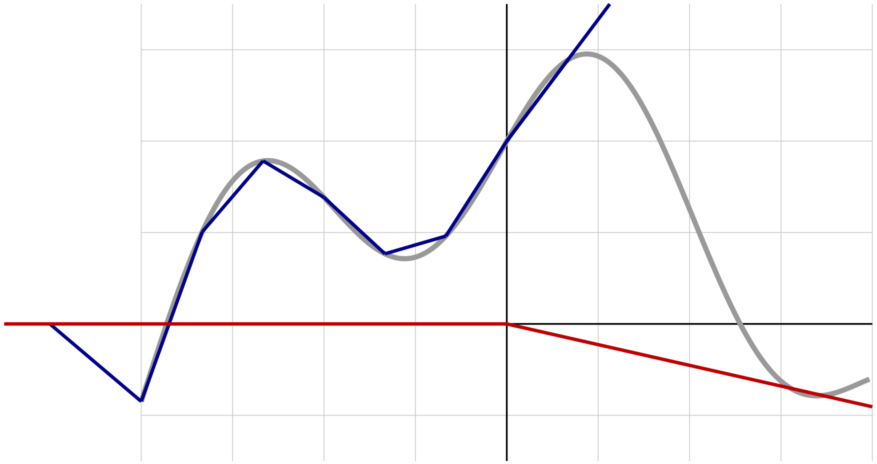


Image credit: Francois Fleuret , lecture 3.4. Multi-Layer Perceptrons

We are going to prove it for sigmoid

- **Universal Approximation**: For any smooth function $f: [0,1]^d \to \mathbb{R}$ and any $\epsilon > 0$ there exists a value for h and a setting of M_1 and M_2 such that
$$|f(x) - M_2\sigma(M_1 x)| \leq \epsilon$$
for all $x \in [0,1]^d$.
- **Food for thought**: Is the same true for $[-10,100]^d \to \mathbb{R}$?

Universal Approximation

- *No matter what the underlying function is,* we can approximate it with a sufficiently wide neural network.
- **Food for thought:** Why shouldn't we just set $h = 1000000000000$ all the time?
  - Computational complexity: $d \times h$
  - Overfitting!
  - If you have some domain knowledge, you might be able to design a more efficient architecture.

Proving Universal Approximation

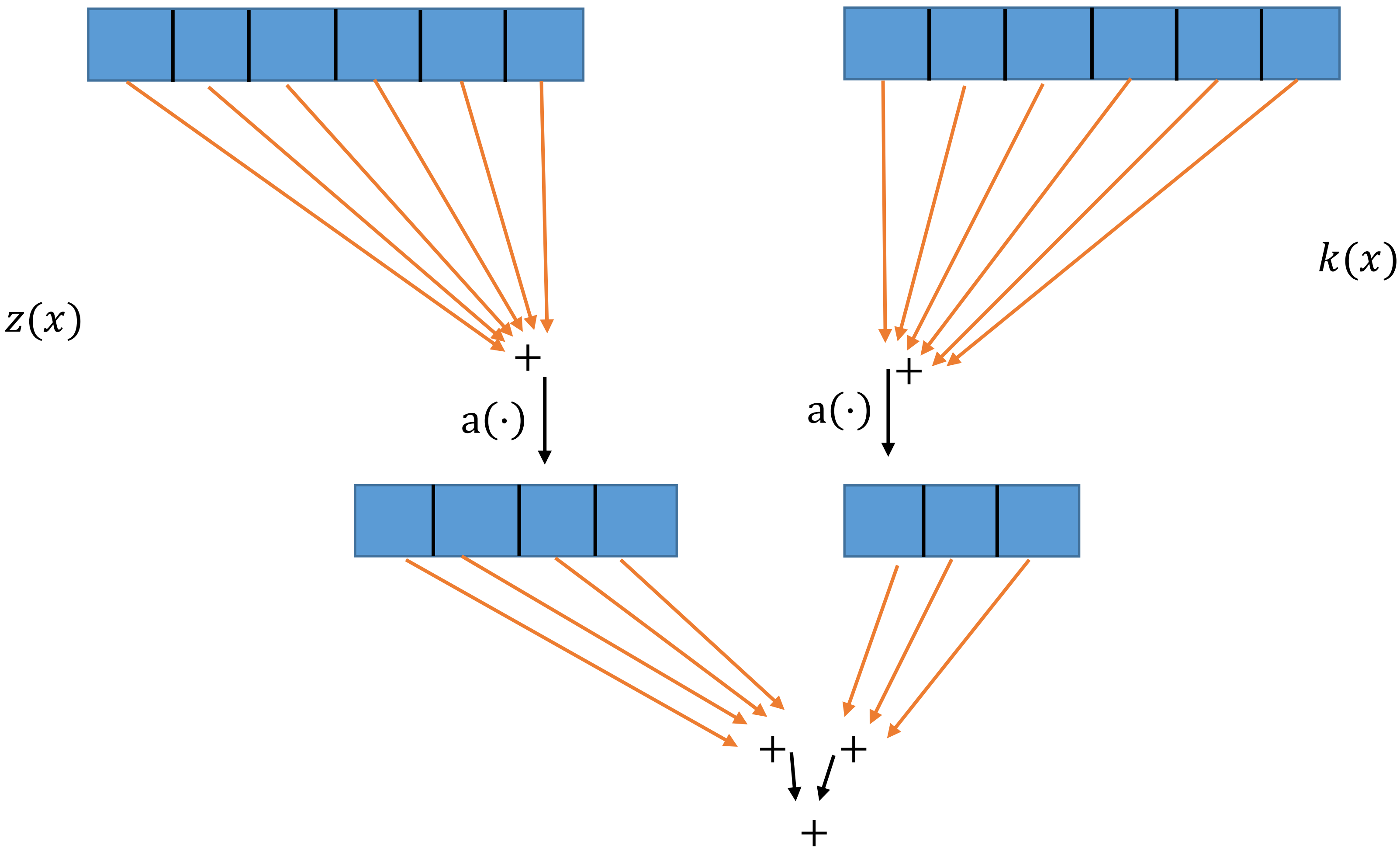
- We will prove this for the sigmoid activation in class. You will show it for the ReLU activation on the homework.
  - **Food for thought:** It is actually true for *any* activation that is not a polynomial (why isn't it true for polynomials?).
- The proof has two steps:

1. First, suppose we want to learn a function $f: [0,1] \to \mathbb{R}$ rather than $f: [0,1]^d \to \mathbb{R}$.
2. Use an argument involving Fourier series to extend from 1-dimension to d-dimensions

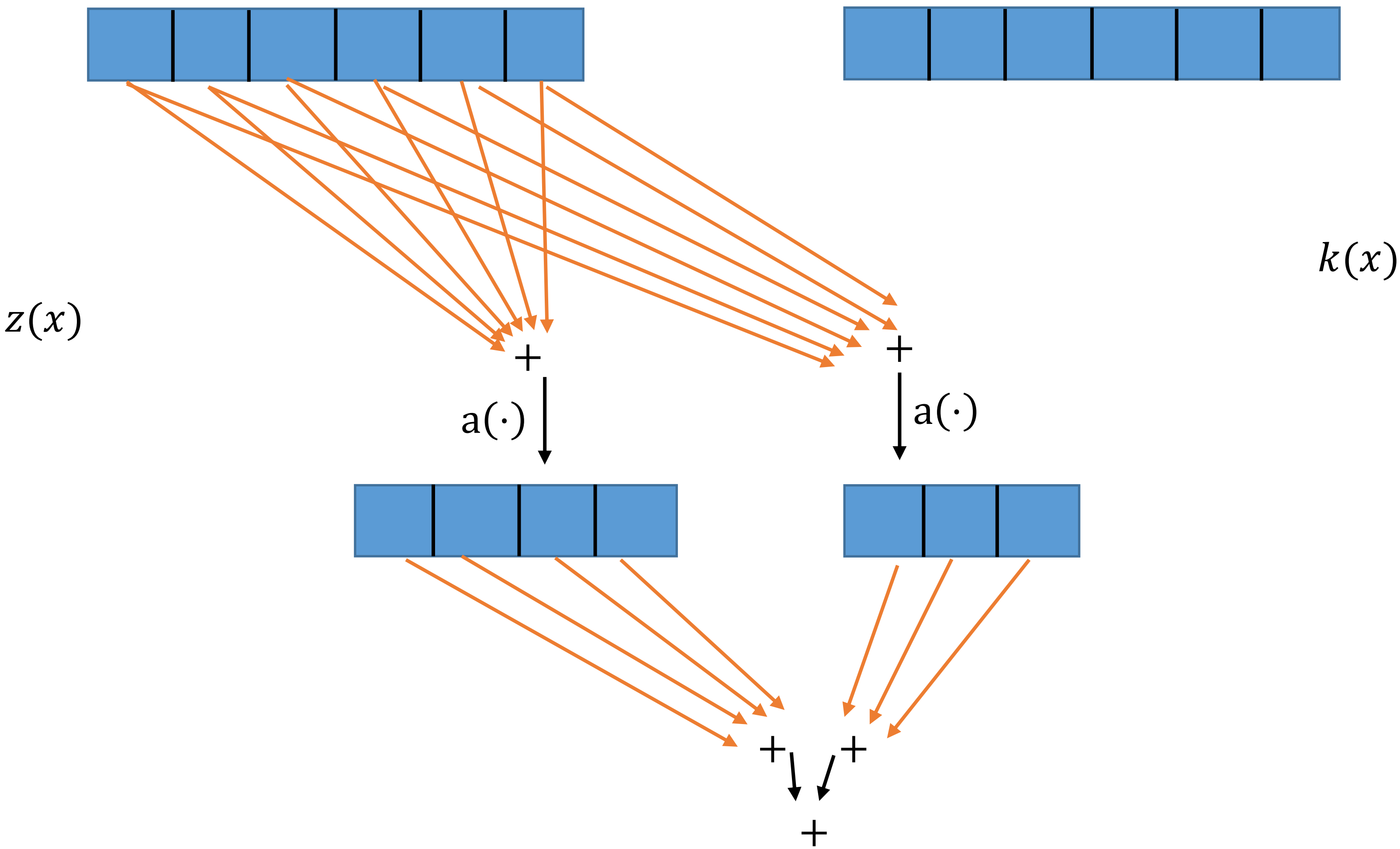
Key fact: We can add networks

- Adding two networks can be written as a single network:
$$z(x) = M_2\sigma(M_1x + b_z)$$
$$k(x) = W_2\sigma(W_1x + b_k)$$
- Want to write $z(x) + k(x)$ as a single network.
- Define $D_1 = \begin{pmatrix} M_1 \\ W_1 \end{pmatrix}$, the matrix given by stacking M_1 on top of W_1. Define $D_2 = (M_2, W_2)$ and $b = (b_z, b_k)$. Then
$$z(x) + k(x) = D_2\sigma(D_1x + b)$$

Key Fact: We can add networks

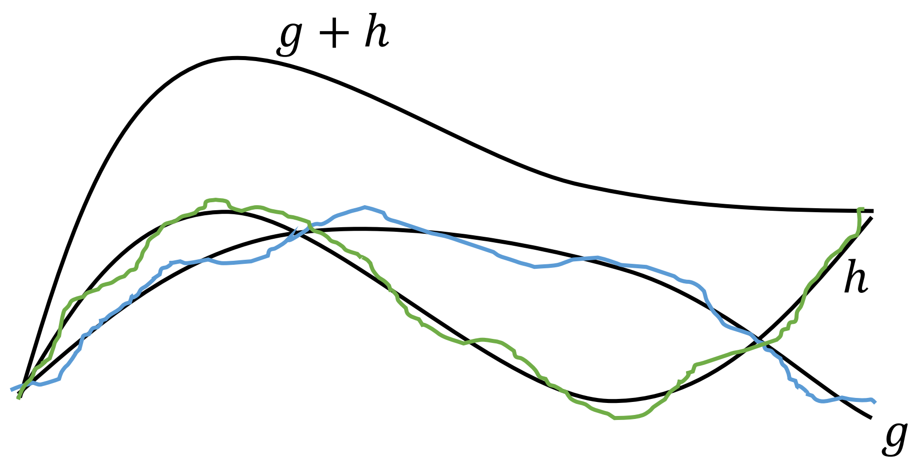


Key Fact: We can add networks



Step 1: $f: [-1,1] \rightarrow \mathbb{R}$

- If g and h can both be approximated to within $\frac{\epsilon}{2}$, then $g + h$ can also be approximated to within ϵ.



- Just add the approximations!
- What happens to the width of the resulting network?

Step 1: $f: [-1,1] \to \mathbb{R}$

- If $M_{g,1}, M_{g,2}$ and $M_{h,1}, M_{h,2}$ are such that
$$\left|g(x) - M_{g,2}\sigma\left(M_{g,1}x + b_g\right)\right| \leq \epsilon$$
$$\left|h(x) - M_{h,2}\sigma\left(M_{h,1}x + b_h\right)\right| \leq \epsilon$$
- Then if $k(x) = cg(x) + dh(x)$,
$$\left|k(x) - \left(cM_{g,2}\sigma\left(M_{g,1}x + b_g\right) + dM_{h,2}\sigma(M_{h,1}x + b_g\right)\right| \leq (|c| + |d|)\epsilon$$
- We increased the width (roughly doubling it).
- If g and h can be approximated with any ϵ, then so can k.

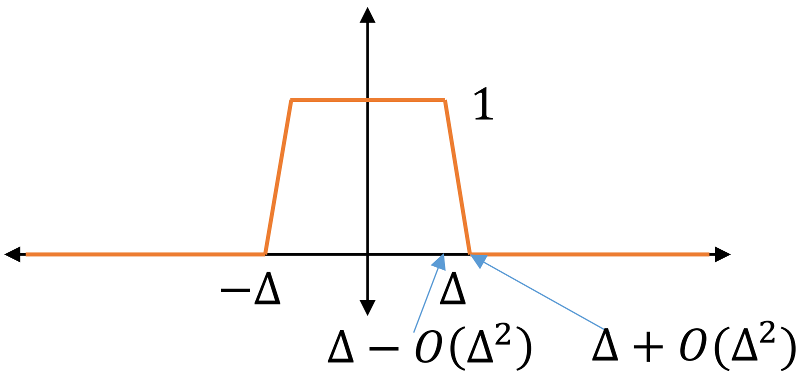
Strategy for 1D functions:

- Design a special class of "simple functions", and show that we can approximate the simple functions well.
- Show that *any* function can be written as a linear combination of simple functions.

- For example, might try simple functions are $\{x, x^2, x^3, x^4, \dots\}$.
  - This would allow us to make any polynomial.
  - Is that good enough?
    - No. Some functions are never close to their Taylor expansions.

Step 1: $f : [-1,1] \to \mathbb{R}$

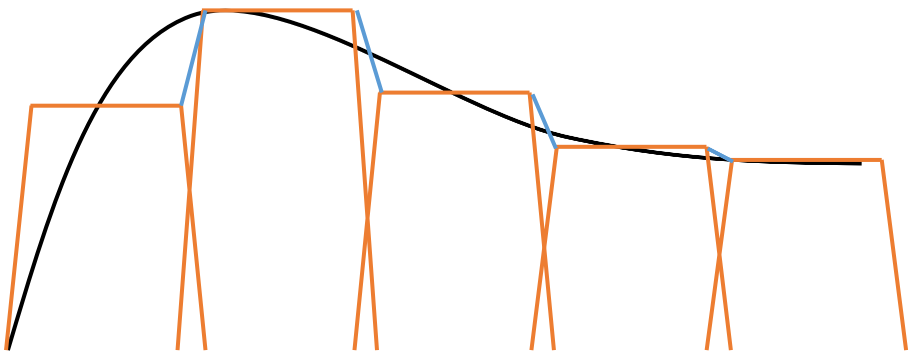
Show that for any Δ, a function like:



Can be approximated well

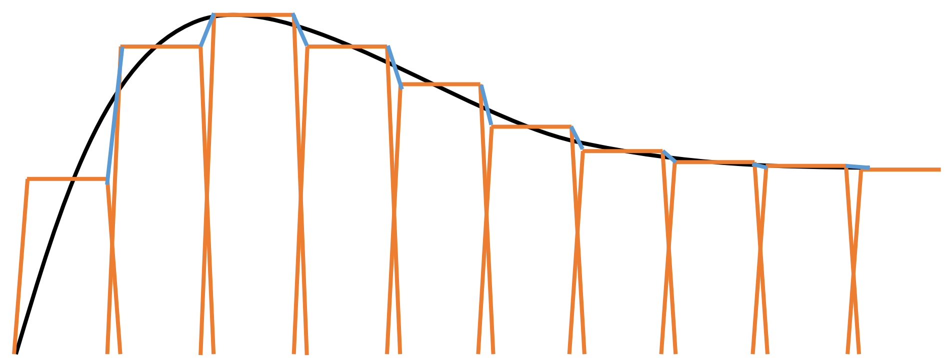
Step 1: $f: [0,1] \to \mathbb{R}$

- Many shifted and scaled copies of this "bump function" can be used to approximate f:



Step 1: $f: [-1,1] \to \mathbb{R}$

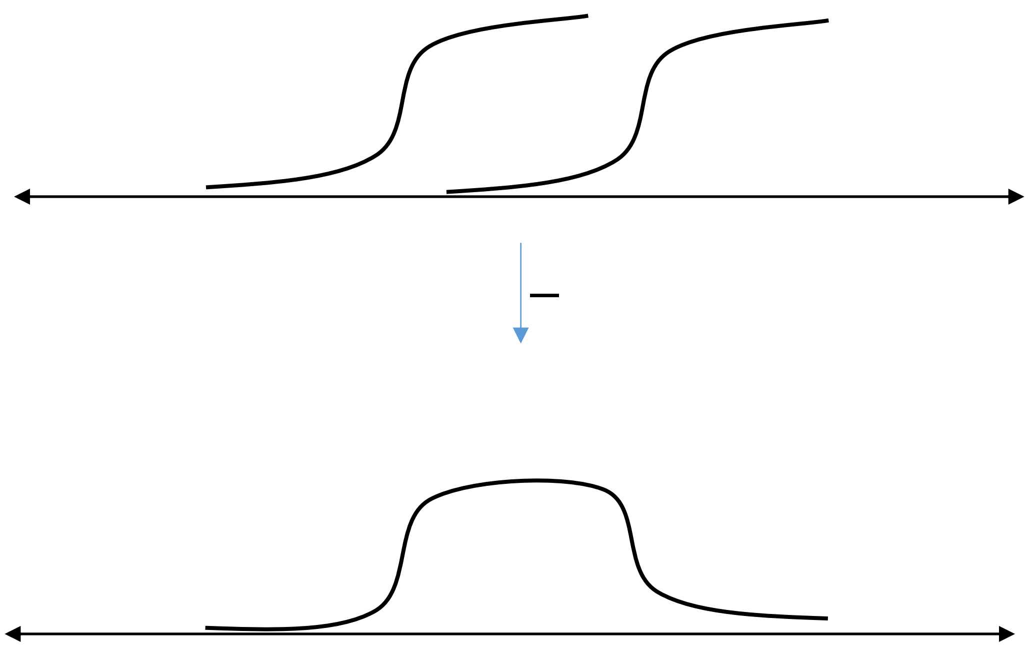
- Many shifted and scaled copies of this "bump function" can be used to approximate f:



Make Δ smaller…

Approximating the bump function

- What does a single neuron's output look like?

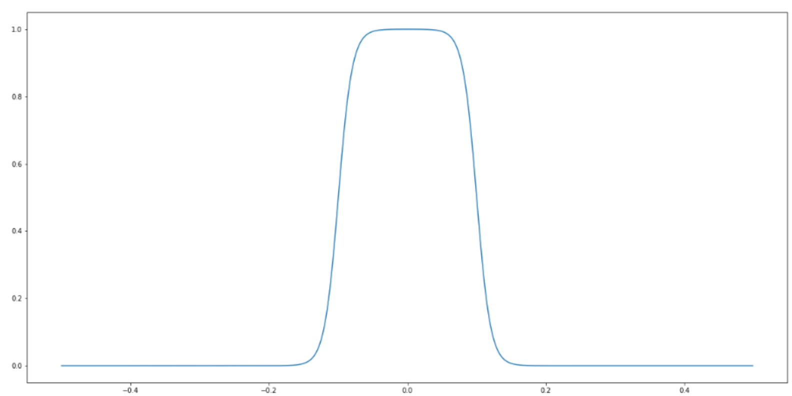


Approximating the bump function

$$\hat{B}_\Delta(x) = \sigma\left(\frac{x+\Delta}{\Delta^2}\right) - \sigma\left(\frac{x-\Delta}{\Delta^2}\right)$$

- $\hat{B}_\Delta(x)$ satisfies:
- $\hat{B}_\Delta(x) \in [1-2\epsilon, 1]$ for $|x| \leq \Delta + \log(\epsilon)\Delta^2$
- $\hat{B}_\Delta(x) \in [0, 2\epsilon]$ for $|x| \geq \Delta - \log(\epsilon)\Delta^2$

$\hat{B}_{0.1}(x)$



Approximating $f(x)$ given bump functions

- You will do this on the homework!

Putting it together for 1D functions:

Single NN can represent bump function

+

Any function is *linear combination* of bumps

+

Combination of single NNs can be represented by single NN

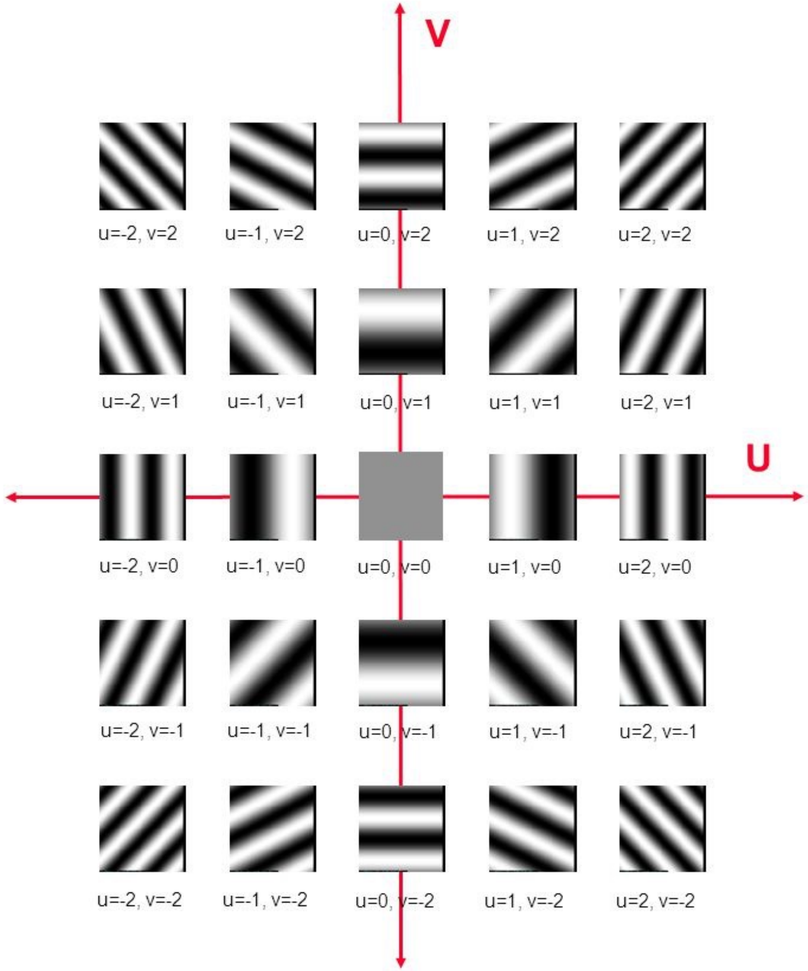
We can write any function as a single layer NN

Step 2: $f: [-1,1]^d \to \mathbb{R}$

1. First, we show that we can approximate $\sin(\omega \cdot x)$ and $\cos(\omega \cdot x)$ for any ω.
2. Next, we show that any function can be written as a combination of $\sin(\omega \cdot x)$ and $\cos(\omega \cdot x)$.
   - The trigonometric functions act like "bump" functions on the high-dimensional space.

- Why don't we just use actual "bump" functions?
  - It's much harder to show how to approximate d-dimensional bump functions than for one-dimension.

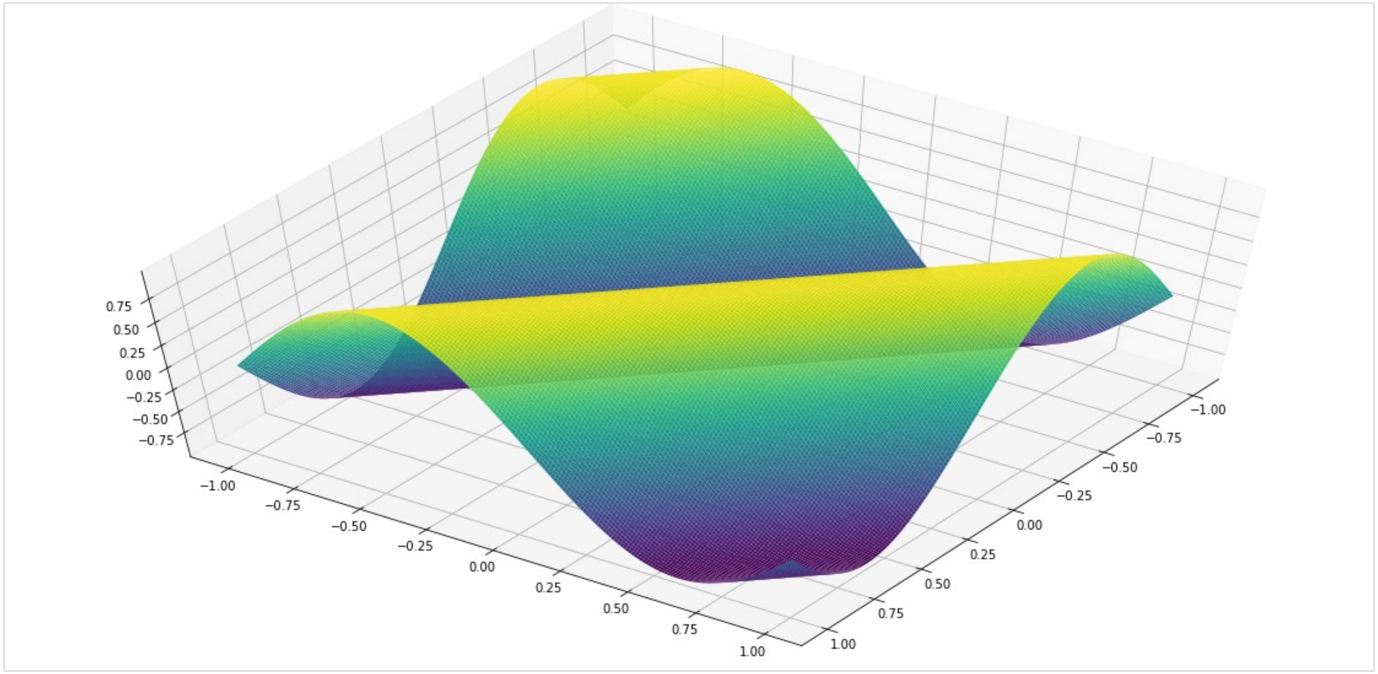
Example: Fourier in 2D



Play with this this out: https://thepythoncodingbook.com/2021/08/30/2d-fourier-transform-in-python-and-fourier-synthesis-of-images/

Image credit: https://slideplayer.com/slide/5665338/

$\sin(\omega \cdot x)$ is a 1D function… from a certain point of view.



$$z \coloneqq \frac{\omega}{||\omega||_2} \cdot x \quad \Rightarrow \quad z \in [-1,1]$$

$$s(z) \coloneqq \sin(||\omega||_2 z)$$

$$\sin(\omega \cdot x) = s(z)$$

Approximating $\sin(\omega \cdot x)$

1. We can approximate $s(x) = \sin(||\omega||_2 x)$ to arbitrary precision on $[-1,1]$ (because we can approximate *anything* on $[-1,1]$).
$$s(x) \approx M_2\sigma(M_1 x + b)$$
2. Next: $\sin(\omega \cdot x) = s\left(\frac{\omega}{||\omega||_2} \cdot x\right)$.
3. Observe that $\frac{\omega}{||\omega||_2} \cdot x \in [-1,1]$ because $x \in [-1,1]^d$
4. Therefore:
$$\sin(\omega \cdot x) \approx M_2\sigma\left(\frac{M_1 \omega^\top}{||\omega||_2} x + b\right)$$

Step 2: $f: [-1,1]^d \to \mathbb{R}$

- Using the Fourier expansion, we can write:
- $f(x) = \int_\omega s_\omega \sin(\omega \cdot x) + c_\omega \cos(\omega \cdot x)\, d\omega$
- We approximate the integral using a Reimann sum:
- $f(x) \approx \sum_\omega \delta\, (s_\omega \sin(\omega \cdot x) + c_\omega \cos(\omega \cdot x))$
- The more terms we include in the Reimann sum, the better the approximation is.
- Include enough terms to get an ϵ accurate approximation to f.
- Each term can be written as a single-layer NN.

Step 2: $f: [-1,1]^d \rightarrow \mathbb{R}$

1. We can approximate $\sin(\omega \cdot x)$ and $\cos(\omega \cdot x)$ for any ω to within $O(\epsilon)$ with a single layer neural network.
2. The function f can be approximated to within $O(\epsilon)$ by a linear combination of $\sin(\omega \cdot x)$ and $\cos(\omega \cdot x)$.
3. Since linear combinations of single-layer networks are also single-layer networks, f can be approximated to within $O(\epsilon)$ by a single-layer network.

Universal Approximation may not be enough

- Just because it is *possible* to express your function using a neural network does not mean that you can actually *find* such an approximation.

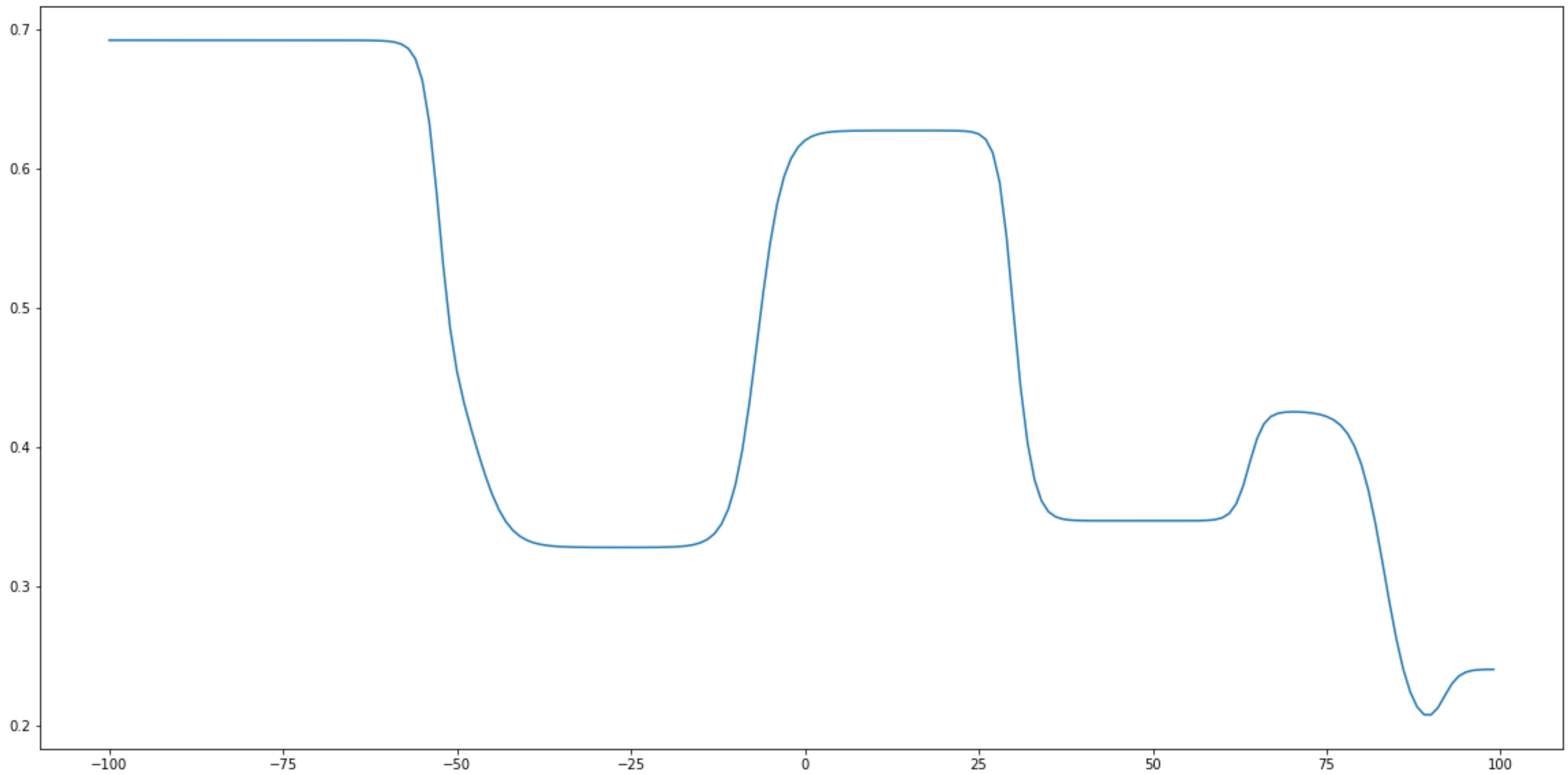
Example

```python
import torch
from torch import nn
```

```python
class SingleLayerNet(nn.Module):
    def __init__(self, layer_size):
        super(SingleLayerNet, self).__init__()
        self.Linear1 = nn.Linear(1,layer_size)
        self.activation = nn.Sigmoid()
        self.Linear2 = nn.Linear(layer_size,1)


    def forward(self, x):
        out = self.Linear1(x)
        out = self.activation(out)
        out = self.Linear2(out)
        return out

```

```python
net = SingleLayerNet(layer_size=10)
outputs = net(x_values)
```

10 units

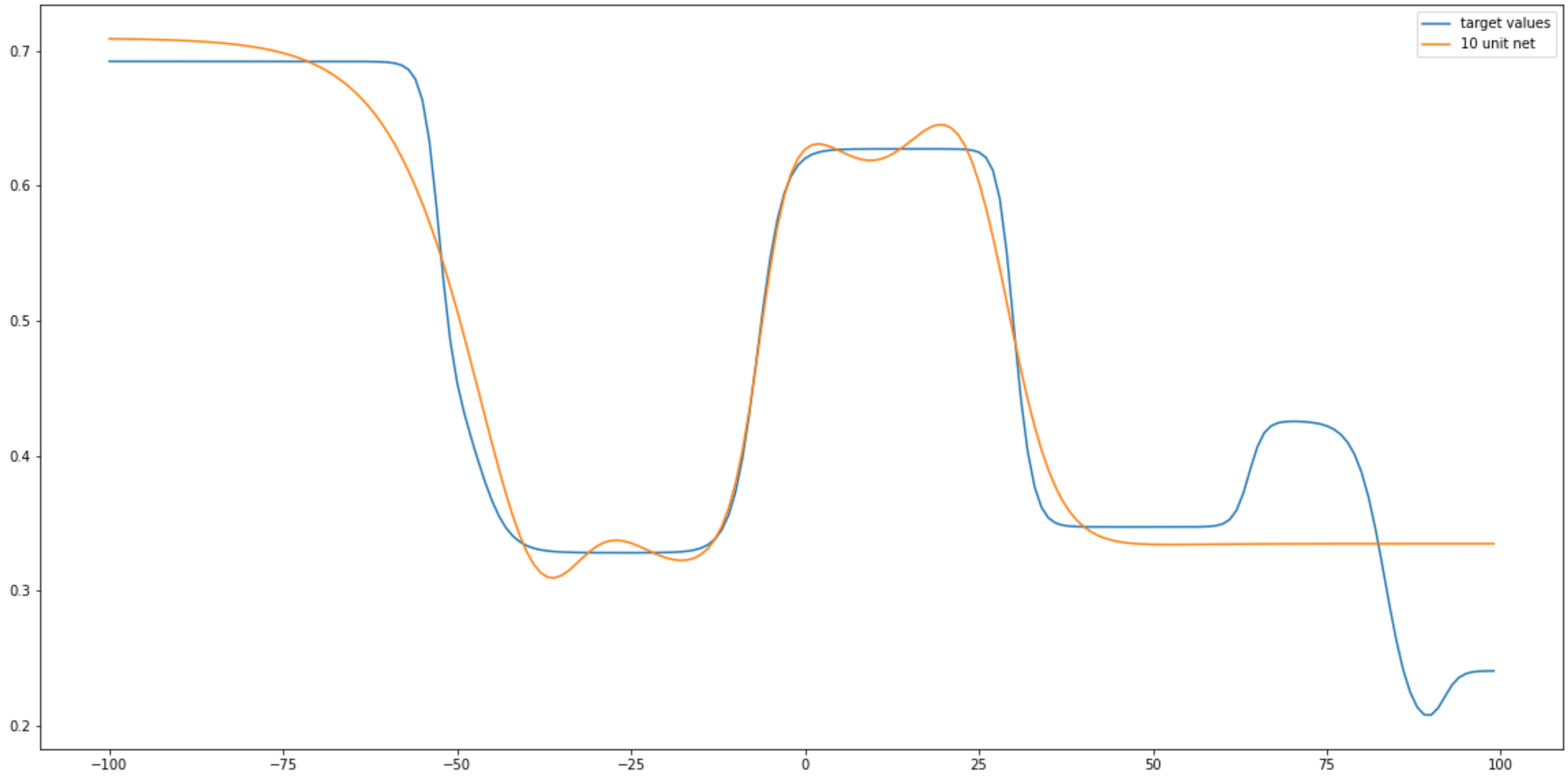


Train a new net with 10 hidden units

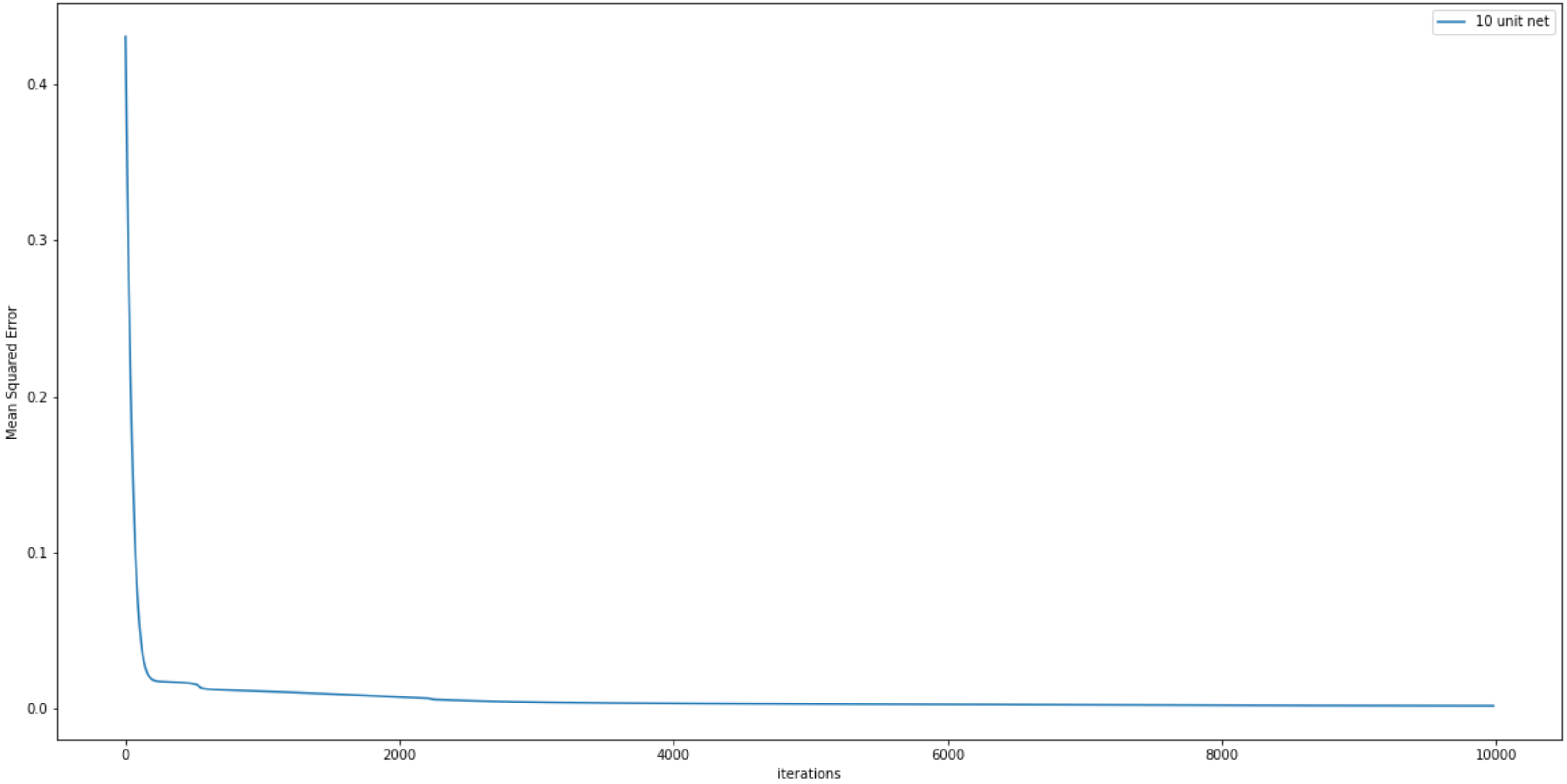
```
def train(net, num_steps=10000):
    loss_func = nn.MSELoss() # mean squared error: (y-\hat y)^2
    losses = [] # should eventually be a list of floats
    optimizer = torch.optim.Adam(net.parameters())
    for t in range(num_steps):
        optimizer.zero_grad() # has to do with computing the gradient

        predictions = net(xvals)
        loss_val = loss_func(predictions, targets)

        loss_val.backward() # has to do with computing gradients

        optimizer.step()

        # lossval is a pytorch Tensor object. lossval.item() extracts the
        # underlying float.
        losses.append(loss_val.item())
    return losses

```

```
net = SingleLayerNet(layer_size=10)
losses = train(net)
```

The fit is mediocre

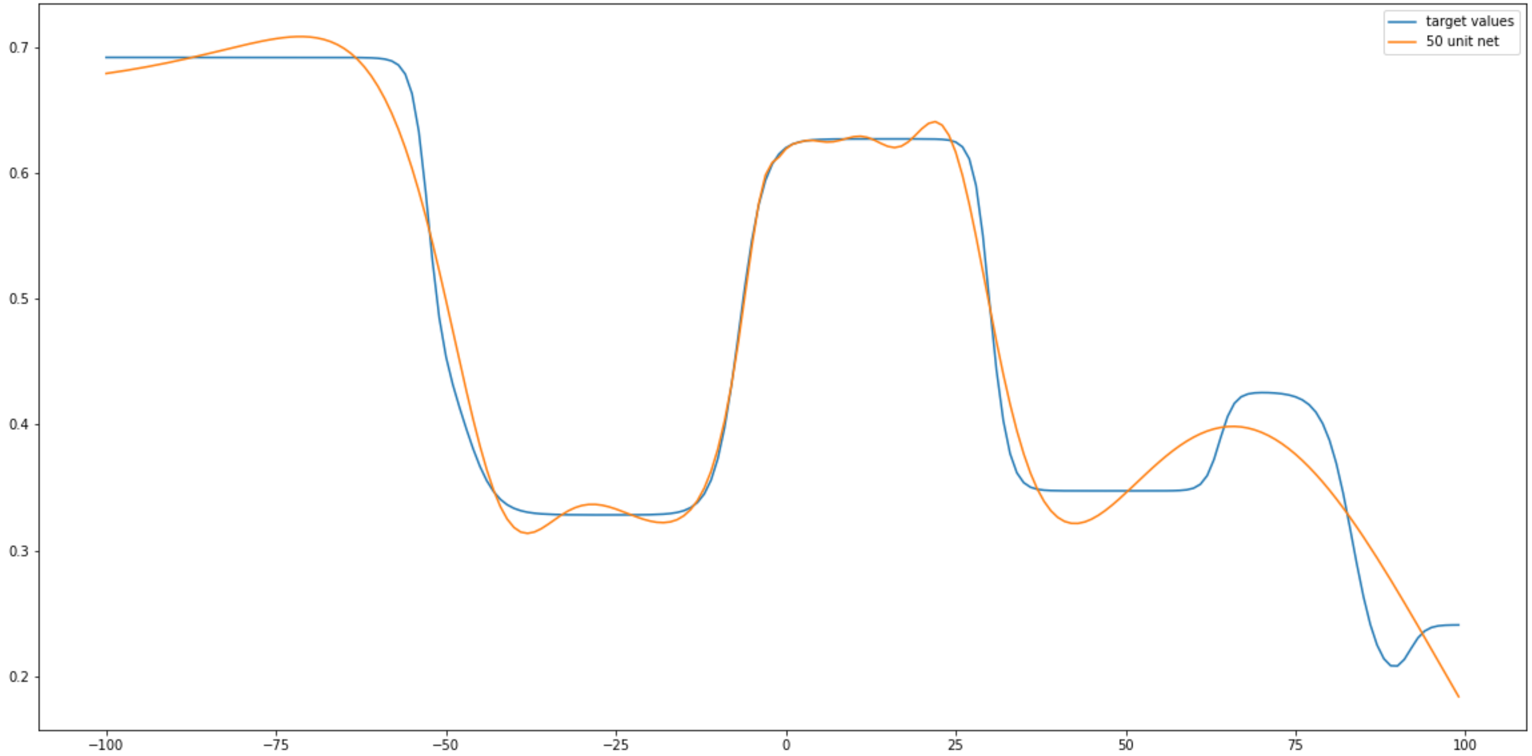


Loss is converged...



Maybe try a bigger net?

```
big_net = SingleLayerNet(layer_size=50)
losses = train(big_net)
```

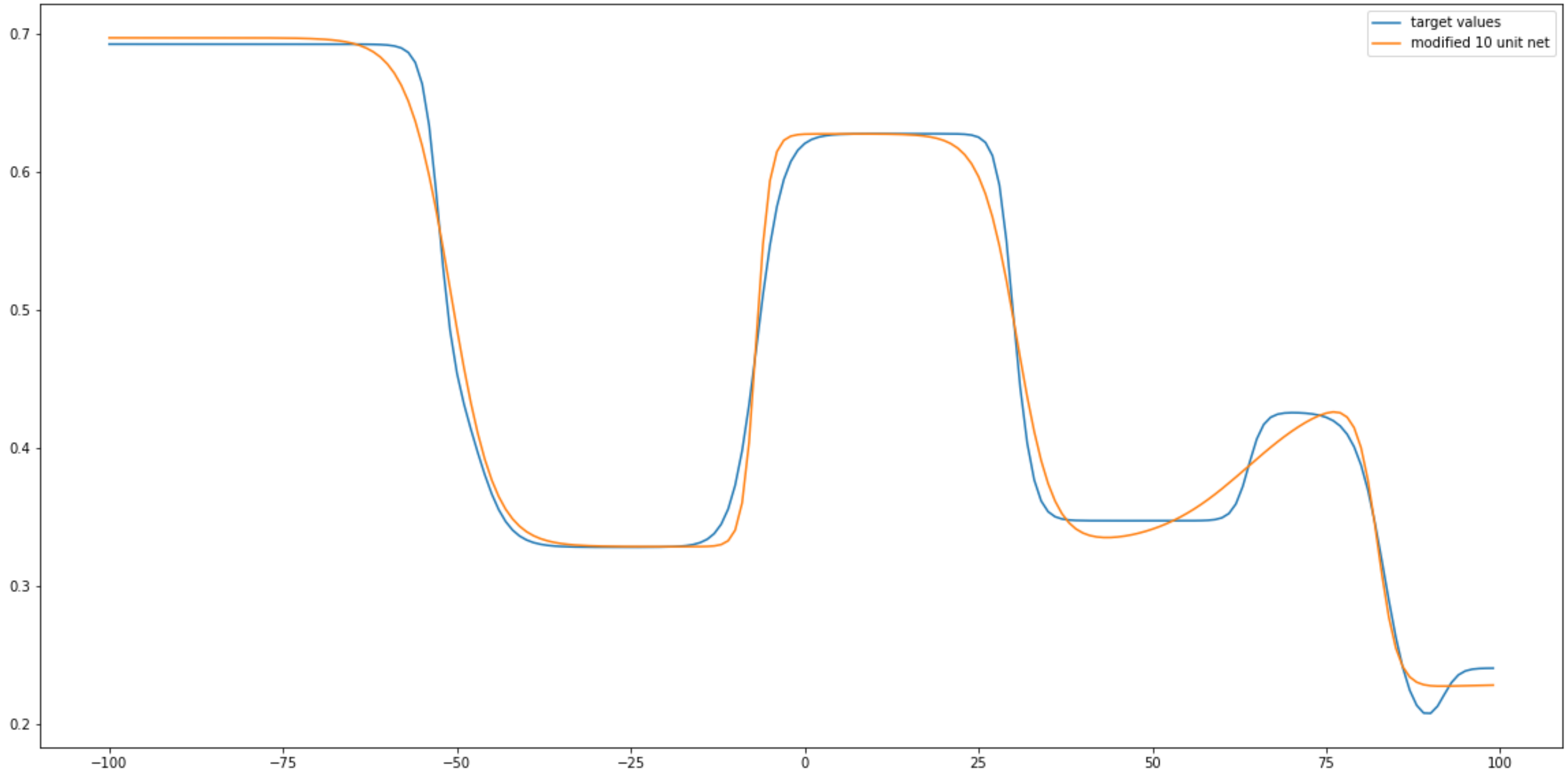


Change the initialization

```python
class SingleLayerNetTryTwo(nn.Module):
    def __init__(self, layer_size):
        super(SingleLayerNetTryTwo, self).__init__()
        self.Linear1 = nn.Linear(1,layer_size)

        # force the initialization of the first linear layer to have widely
        # distributed biases - similar to how I made the original net.
        with torch.no_grad():
            self.Linear1.bias *= 80

        self.activation = nn.Sigmoid()
        self.Linear2 = nn.Linear(layer_size,1)


    def forward(self, x):
        out = self.Linear1(x)
        out = self.activation(out)
        out = self.Linear2(out)
        return out
```

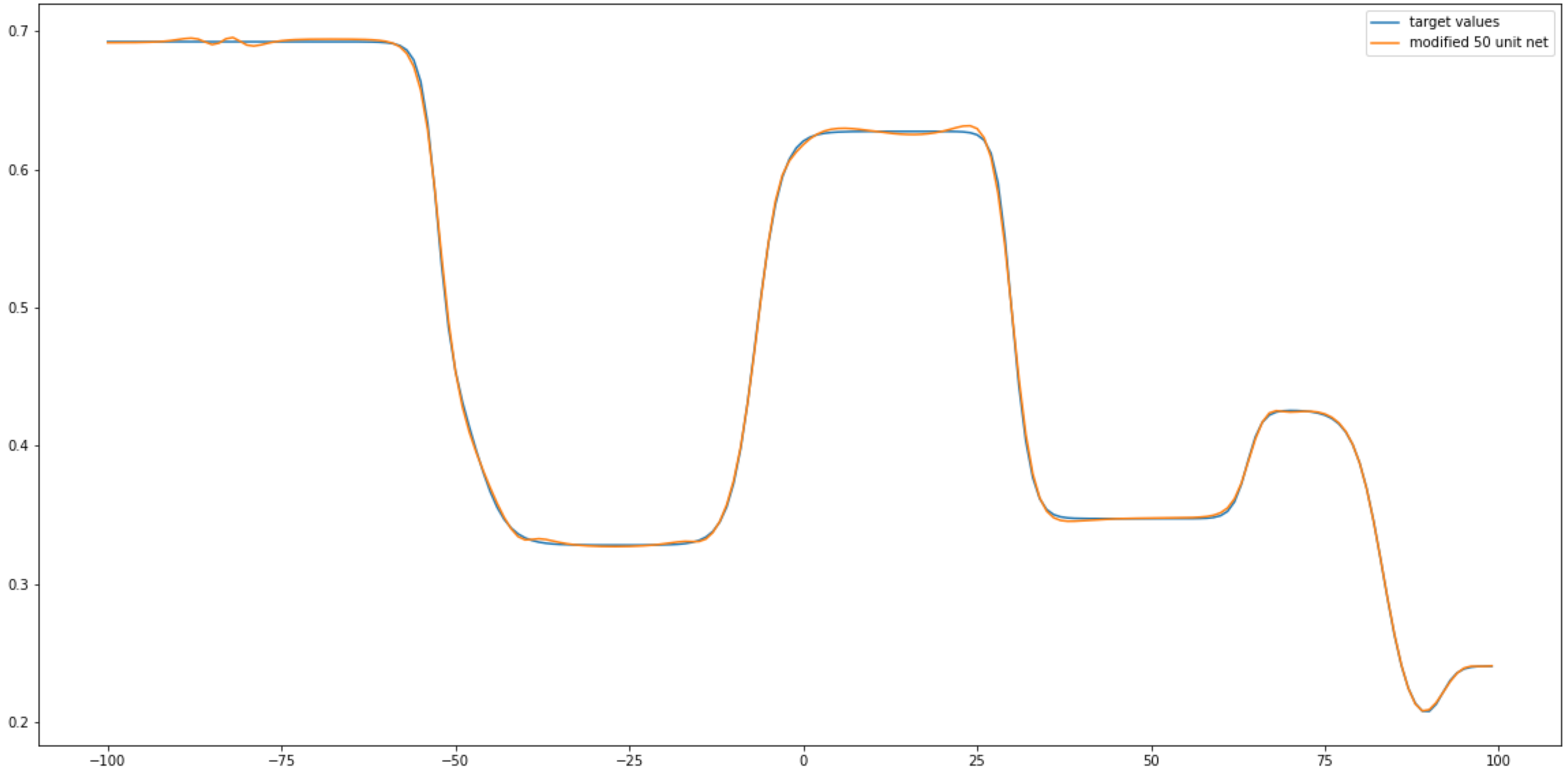
The fit is better!

```
modified_net = SingleLayerNetTryTwo(layer_size=10)
modified_losses = train(modified_net)
```



Final attempt: bigger net with better initialization

```
modified_big_net = SingleLayerNetTryTwo(layer_size=50)
losses_modified_big_net = train(modified_big_net)
```



Lessons Learned

- Just because the network *can* express a function (it has enough *capacity*) doesn't mean it *will* express the function.
- Adding more capacity (*overparameterizing*) can make it easier to learn hard functions.
  - This might also lead to overfitting... make sure to check if this is happening!
- Initialization/normalization of data can matter a lot!
  - If you know something about your data, you might be able to help your model train.