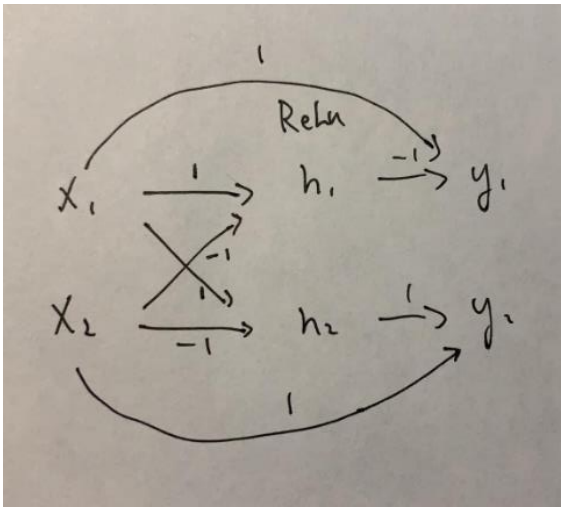
1 Hard-Coding Networks

1.1 Verify Sort

$$\mathbf{W}^{(1)} = \begin{bmatrix} -1 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 1 \end{bmatrix} \mathbf{b}^{(1)} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \mathbf{W}^{(2)} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \mathbf{b}^{(2)} = 3$$

1.2 Perform Sort

First, we can make a modular compare and swap network for 2 input, which is also known as a bubble sort for 2 elements and it works as follows:



Note that this will sort two numbers in ascending order i.e. $y1 \leq y2$.Then, we can expand this to sorting 4 numbers. We know that the worst case for bubble sort for 4 numbers is having a descending order input, which requires a total of 6 swaps. Therefore, we can create a neural network with 6 modules of the bubble sort provided above which compares the inputs in the way the bubble sort works. For example, first compare x1, x2, then compare new x2 with x3, etc.

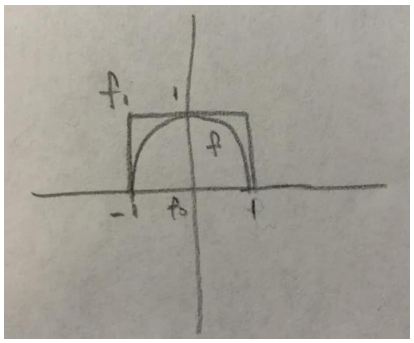
1.3 Universal Approximation Theorem

1.3.1 The answer is basically a modified verify sort but with only 3 elements and output h if $a \leq x \leq b$, and output 0 otherwise. Therefore, we can pick:

$$n = 2, \mathbf{W}^{(0)} = \begin{bmatrix} -1 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix}, \mathbf{b}^{(0)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \mathbf{W}^{(1)} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \mathbf{b}^{(1)} = -2$$

We can obtain value of h by multiplying the 1 output by the network like the verify sort network. Thus, we achieved a bump function that outputs h if x is between a and b.

1.3.2 We can make a bump function that covers f, as shown in the graph below:



Where $h_1 = 1, a_1 = -1, b_1 = -1$ because we want the bump to match with the peak of f which is 1. On the other hand, we also want to make sure the boundary matches which is -1 to 1.

Then we can easily calculate $\|f - \hat{f}_1\|$ and $\|f - \hat{f}_0\|$:

$$\|f - \hat{f}_1\| = \|\hat{f}_1 - \hat{f}_0\| - \|f - \hat{f}_0\| = 2 - \int_{-1}^{1} -x^2 + 1 dx = 2 - \frac{4}{3} = \frac{2}{3}$$

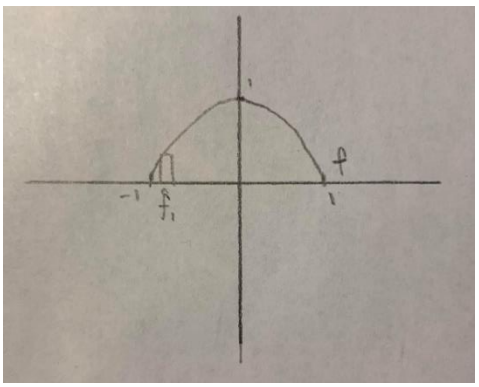
$$\|f - \hat{f}_0\| = \frac{4}{3}$$

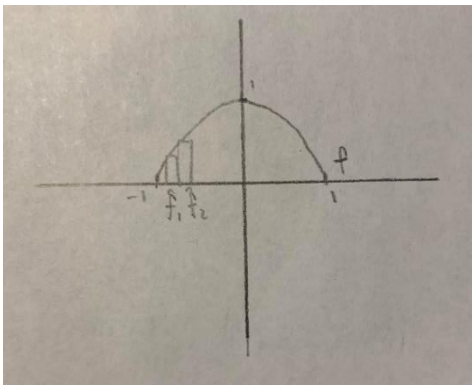
Thus, $\|f - \hat{f}_1\| < \|f - \hat{f}_0\|$.

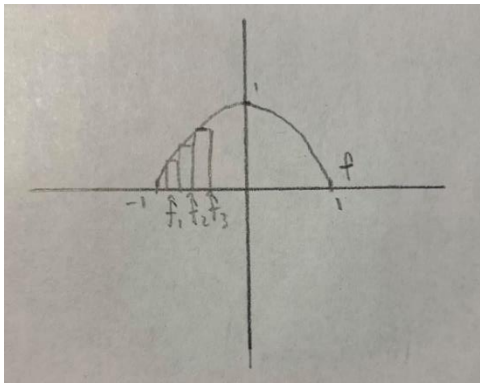
1.3.3 Since we have N bump functions to approximate f. We first divide distance between the two x-intercepts which is 2 by N and call this as $\Delta x = \frac{2}{N}$. Then we know how wide each bump function is. We can start from the left end where $a_0 = -1, b_0 = a_0 + \Delta x, h_0 = f\left(a_0 + \frac{\Delta x}{2}\right)$. Notice that this is very similar as approximating integral using rectangles and taking the midpoint value as the height of the rectangle. Then, we can continue adding more bump functions with the following rule:

$$a_{i+1} = b_i, b_{i+1} = a_{i+1} + \Delta x, h_{i+1} = f\left(a_{i+1} + \frac{\Delta x}{2}\right)$$

Notice that $\|f - \hat{f}_{i+1}\| < \|f - \hat{f}_i\|$ because we are adding more rectangles to approximate the function better as we can see from the following graphs showing $\hat{f}_1, \hat{f}_2, \hat{f}_3$. We can see that the differences between the rectangles and f are getting smaller.



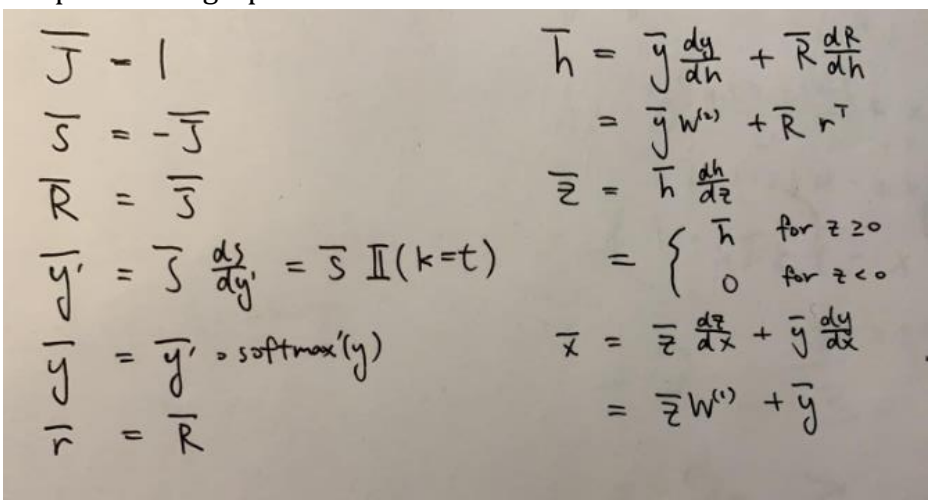




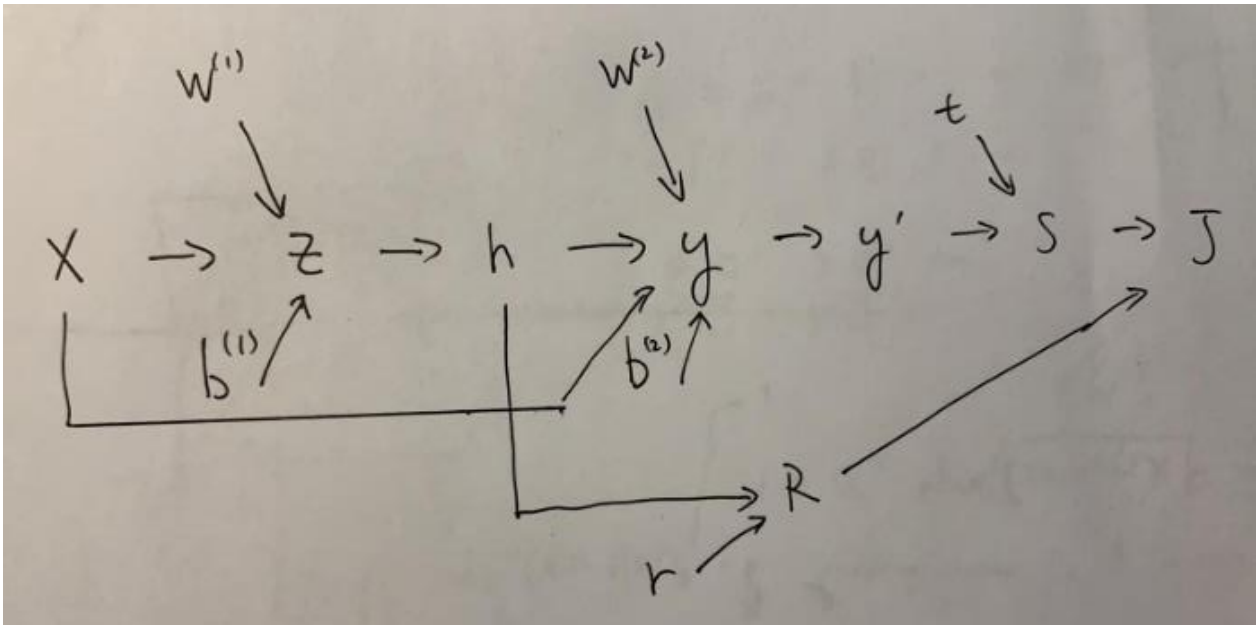
2 Backprop

2.1 Computational Graph

2.1.1 The computational graph is as follows:



2.1.2 The equations are as follows:



2.2 Vector-Jacobian Products

2.2.1 The Jacobian matrix is as follows:

$$J = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{bmatrix} = \mathbf{v}\mathbf{v}^{\mathbf{T}}$$

2.2.2 The time is $\Omega(n^2)$ and memory cost is $O(n^2)$ because there are n^2 entries to calculate and n^2 entries to store to memory.

2.2.3 We can perform such action in linear time if we multiply the matrices in a different order:

$$z = J^{\mathrm{T}}y = \mathbf{v}\mathbf{v}^{\mathrm{T}}y = \mathbf{v} \cdot (\mathbf{v}^{\mathrm{T}} \cdot y)$$

So, we will first get a 1x1 product from $(\mathbf{v}^{\mathrm{T}} \cdot y)$, then a 3x1 product from $\mathbf{v} \cdot (\mathbf{v}^{\mathrm{T}} \cdot y) = z$. Therefore, either of the operations can be done in linear time and memory cost since we did not get any temporary square matrix like what we got from 2.2.1. Thus, we can evaluate $J^{\mathrm{T}}y$ in linear time and memory cost.

3 Linear Regression

3.1 Deriving the gradient as follows:

$$\frac{d}{d\widehat{\mathbf{w}}} \min_{\widehat{\mathbf{w}}} \frac{1}{n} \|X\widehat{\mathbf{w}} - \mathbf{t}\|_2^2 = \frac{2}{n} X^{\mathrm{T}}(X\widehat{\mathbf{w}} - \mathbf{t}) = \frac{2}{n}(X^{\mathrm{T}}X\widehat{\mathbf{w}} - X^{\mathrm{T}}\mathbf{t})$$

3.2 Underparameterized Model

3.2.1 By setting the gradient to 0, we get the following:

$$\frac{2}{n}(X^{\mathrm{T}}X\widehat{\mathbf{w}} - X^{\mathrm{T}}\mathbf{t}) = 0$$
$$X^{\mathrm{T}}X\widehat{\mathbf{w}} - X^{\mathrm{T}}\mathbf{t} = 0$$
$$X^{\mathrm{T}}X\widehat{\mathbf{w}} = X^{\mathrm{T}}\mathbf{t}$$
$$\widehat{\mathbf{w}} = (X^{\mathrm{T}}X)^{-1}X^{\mathrm{T}}\mathbf{t}$$

Note that $X^{\mathrm{T}}X$ is invertible as given since $n > d$.

3.2.2 Assume that ground truth labels are generated by a linear target, we get:

$$\mathbf{t} = X\mathbf{w}^*$$

Since they share the same target, we can substitute above into the equation of $\widehat{\mathbf{w}}$

$$\widehat{\mathbf{w}} = (X^{\mathrm{T}}X)^{-1}X^{\mathrm{T}}X\mathbf{w}^* = \mathbf{w}^*$$

Therefore, we can say that:

$$\forall x \in \mathbb{R}^d, \left(\mathbf{w}^{*\mathrm{T}}\mathbf{x} - \widehat{\mathbf{w}}^{\mathrm{T}}\mathbf{x}\right)^2 = 0$$

Thus, we can conclude that the solution achieves perfect generalization for $d < n$.

3.3 Overparameterized Model: 2D Example

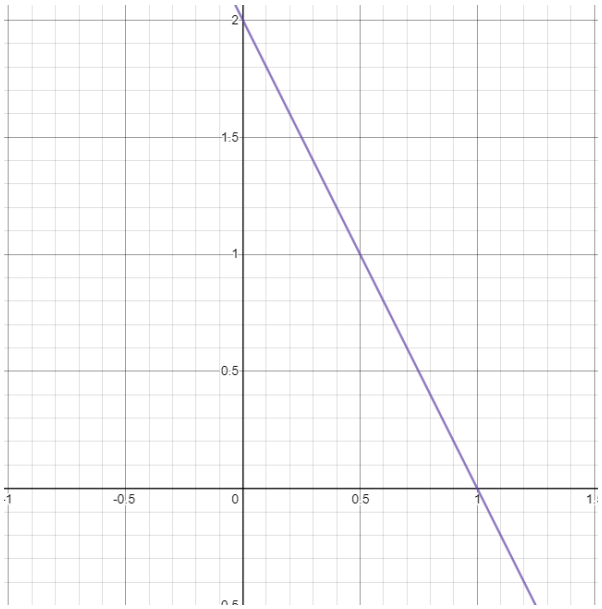
3.3.1 We want to show the following:

$$\widehat{\mathbf{w}}^{\mathrm{T}}\mathbf{x}_1 = y_1$$

Since we are given $n = 1, d = 2$. We can expand the weight matrix as follows:

$$[w_1 \quad w_2]\begin{bmatrix}2\\1\end{bmatrix} = 2w_1 + w_2 = 2 = t_1$$
$$w_2 = 2 - 2w_1$$

We have an equation of line which is as follow:

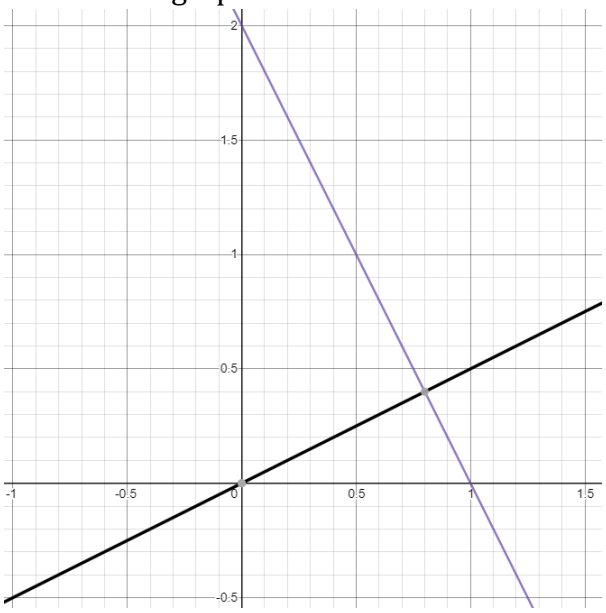


3.3.2 We can now substitute $\mathbf{x_1}$ into the gradient we obtained and set it to 0:

$$\frac{2}{n}X^{\mathrm{T}}(X\hat{\mathbf{w}} - \mathbf{t}) = 2[2 \quad 1]\begin{bmatrix}2\\1\end{bmatrix}\hat{\mathbf{w}} - 2[2 \quad 1]2$$
$$= 10\hat{\mathbf{w}} - [8 \quad 4] = 0$$
$$\hat{\mathbf{w}} = [0.8 \quad 0.4]$$

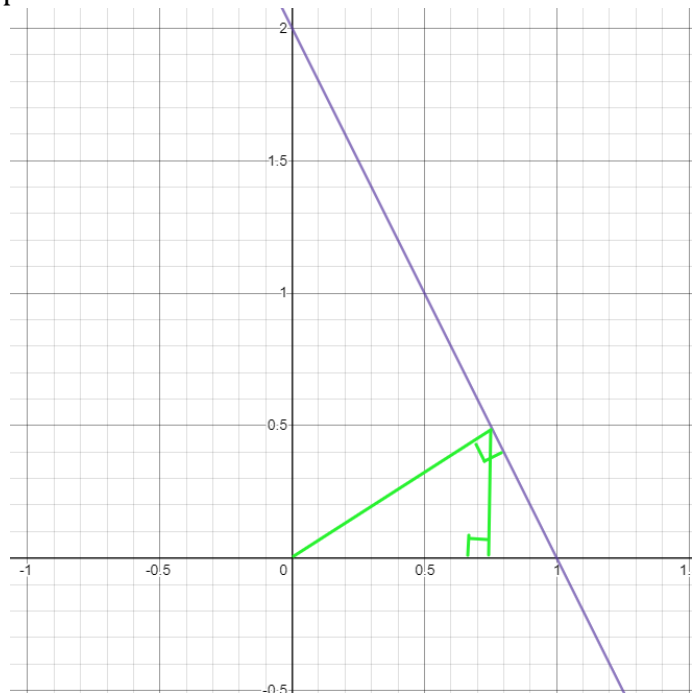
And then we can turn it into a unit vector $\hat{\mathbf{w}} = \left[\frac{0.8}{\sqrt{0.8}} \quad \frac{0.4}{\sqrt{0.8}}\right]$

If we plot the line onto the same graph as above:



We can see that it is perpendicular to the line of solution we obtained from before. Therefore, we can say that the direction will not change along the trajectory because this is the shortest Euclidean distance towards the line of solution if we start from 0. Therefore, based on this intuition, I believe that the gradient descent will find $\hat{\mathbf{w}} = [0.8 \quad 0.4]$ as the solution.

3.3.3 By the Pythagoras theorem, we can find the line that would create a right triangle shown in the graph below:



The line that started from the origin is exactly the gradient descent solution we found, and we know this line has the smallest Euclidean norm because the distance is orthogonal to the solution of line. Thus, there cannot exist another line that is shorter. We can further show it below:

The equation that we are trying to minimize is as follows by Pythagoras Theorem:

$$f = \sqrt{x^2 + (2 - 2x)^2}$$

Note that this is same as minimizing the expression inside the square root:

$$f = x^2 + (2 - 2x)^2$$

If we take the derivative of it, we get the following:

$$\frac{d}{dx} x^2 + (2 - 2x)^2 = 2x + 2(2 - 2x)(-2) = 10x - 8$$

Note that this gradient is exactly the line we drew in 3.3.2.

Therefore, we can conclude that the solution from previous part has the smallest Euclidean norm.

3.4 <u>Overparameterized Model: General Case</u>

3.4.1 To Generalize the overparameterized model, we need to show gradient descent from zero initialization finds a unique minimizer if it converges.

We know that $\widehat{\mathbf{w}}$ must be a linear combination of some constants with the input. So, let $\widehat{\mathbf{w}} = X^{\mathrm{T}}C$ for some constant matrix C.

We can now substitute this into the following:

$$\frac{2}{n} X^{\mathrm{T}}(X\widehat{\mathbf{w}} - \mathbf{t}) = 0$$

We can drop the 2 and n term because it does not help us find a unique minimizer

$$X^{\mathrm{T}}(XX^{\mathrm{T}}C - \mathbf{t}) = 0$$
$$XX^{\mathrm{T}}(XX^{\mathrm{T}}C - \mathbf{t}) = X \cdot 0$$

Since we know XX^{T} is invertible when $n < d$

$$(XX^{\mathrm{T}}C - \mathbf{t}) = (XX^{\mathrm{T}})^{-1}X \cdot 0 = 0$$
$$C = (XX^{\mathrm{T}})^{-1}\mathbf{t}$$

Substituting C back into $\hat{\mathbf{w}}$ gives us the following:

$$\hat{\mathbf{w}} = X^{\mathrm{T}}(XX^{\mathrm{T}})^{-1}\mathbf{t}$$

Note that this is indeed unique.

3.4.2 Given a zero-loss solution $\hat{\mathbf{w}}_1$, we can evaluate the following:

$$(\hat{\mathbf{w}} - \hat{\mathbf{w}}_1)^{\mathrm{T}}\hat{\mathbf{w}} = (X^{\mathrm{T}}(XX^{\mathrm{T}})^{-1}\mathbf{t} - \hat{\mathbf{w}}_1)^{\mathrm{T}}\hat{\mathbf{w}}$$
$$= (\mathbf{t}^{\mathrm{T}}(XX^{\mathrm{T}})^{-1}X - \hat{\mathbf{w}}_1^{\mathrm{T}})\hat{\mathbf{w}} = (\mathbf{t}^{\mathrm{T}}(XX^{\mathrm{T}})^{-1}X - \hat{\mathbf{w}}_1^{\mathrm{T}})X^{\mathrm{T}}(XX^{\mathrm{T}})^{-1}\mathbf{t}$$
$$= \mathbf{t}^{\mathrm{T}}(XX^{\mathrm{T}})^{-1}XX^{\mathrm{T}}(XX^{\mathrm{T}})^{-1}\mathbf{t} - \hat{\mathbf{w}}_1^{\mathrm{T}}X^{\mathrm{T}}(XX^{\mathrm{T}})^{-1}\mathbf{t}$$
$$= \mathbf{t}^{\mathrm{T}}(XX^{\mathrm{T}})^{-1}\mathbf{t} - \hat{\mathbf{w}}_1^{\mathrm{T}}X^{\mathrm{T}}(XX^{\mathrm{T}})^{-1}\mathbf{t}$$

Note that $\hat{\mathbf{w}}_1^{\mathrm{T}}X = t$ because it is a zero-loss solution. Therefore:

$$= \mathbf{t}^{\mathrm{T}}(XX^{\mathrm{T}})^{-1}\mathbf{t} - \hat{\mathbf{w}}_1^{\mathrm{T}}X^{\mathrm{T}}(XX^{\mathrm{T}})^{-1}\mathbf{t} = 0$$

This also means that the gradient descent solution is orthogonal to the zero-loss solution. Thus, using similar arguments as 3.3.3, we can say that the gradient descent solution is orthogonal to all possible solutions, and so, the solution with the smallest Euclidean norm.

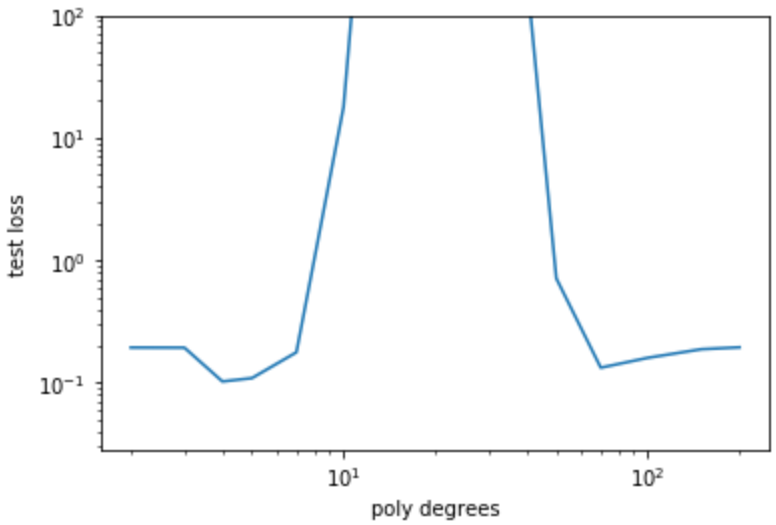
3.5 Benefit of Overparameterization

3.5.1 Here is the code snippet:

```
4  def fit_poly(X, d, t):
5    X_expand = poly_expand(X, d=d, poly_type = poly_type)
6    if d <= n:
7      W = (np.linalg.inv(X_expand.T @ X_expand) @ X_expand.T) @ t
8    else:
9      W = (X_expand.T @ np.linalg.inv(X_expand @ X_expand.T)) @ t
10   return W
```

And no, overparameterizing does not always lead to overfitting. If we look at the graph produced:

Vincent Fan



We can see that when the degrees reach around 10^2, the test loss is similar to lower degrees. Thus, we can say that it does not always lead to overfitting when d >> n.