1 Backpropagation (2pts)

Consider a MLP consisting of N input units, K hidden units and N output units. The activations are computed as follows:

$$\begin{aligned} \boldsymbol{z} &= \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)} \\ \boldsymbol{h} &= \sigma(\boldsymbol{z}) \\ \boldsymbol{y} &= \boldsymbol{x} + \boldsymbol{W}^{(2)}\boldsymbol{h} + \boldsymbol{b}^{(2)} \end{aligned}$$

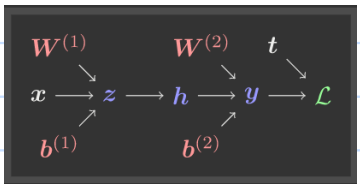
where $\sigma(\cdot)$ is the sigmoid function, applied element wise. The loss $\mathcal{L}$ will involve both $\boldsymbol{h}$ and $\boldsymbol{y}$:

$$\begin{aligned} \mathcal{L} &= \mathcal{S} + \mathcal{R} \\ \mathcal{S} &= \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{s}\|_2^2 \\ \mathcal{R} &= \boldsymbol{r}^T\boldsymbol{h} \end{aligned}$$

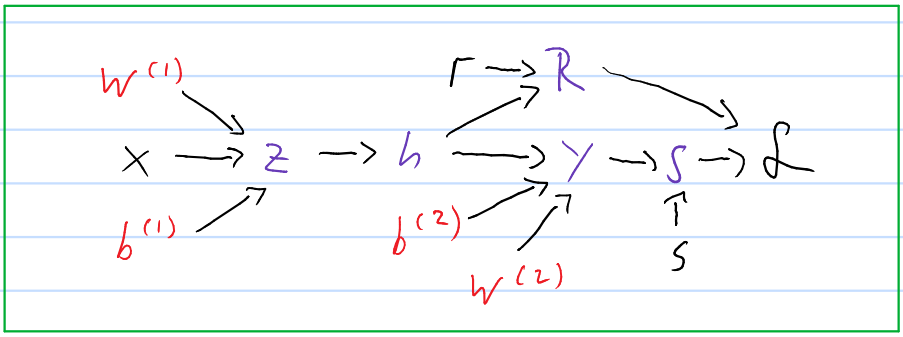
where $\boldsymbol{r}$ and $\boldsymbol{s}$ are given vectors.

1. (**1pt**) Draw the computation graph relating $\boldsymbol{x}$, $\boldsymbol{z}$, $\boldsymbol{h}$, $\boldsymbol{y}$, $\mathcal{R}$, $\mathcal{S}$ and $\mathcal{L}$
2. (**1pt**) Derive the backpropagation equations for computing $\frac{\partial \mathcal{L}}{\partial \boldsymbol{x}}$. You may use σ' to denote the derivation of the sigmoid function, so you do not have to write it out explicitly.

1. From the lecture slides:



So here with different $\mathcal{L}$:



2. I'm using the denominator layout convention (as usual in ML) so $\frac{\partial \mathcal{L}}{\partial \boldsymbol{v}}$ is a column vector, if $\boldsymbol{v}$ is a column vector and $\mathcal{L}$ is a scalar.

$$\mathcal{L}' = 1, \quad R' = \frac{\partial \mathcal{L}}{\partial R} = 1 = S', \quad y' = \mathcal{L}'\left(R' \underbrace{\frac{\partial R}{\partial y}}_{=0} + S' \frac{\partial S}{\partial y}\right) = \frac{\partial S}{\partial y} = y - s$$

column vector!

$$h' = \underbrace{\frac{\partial y}{\partial h}}_{\text{Jacobian}} \cdot y' + R' \frac{\partial R}{\partial h} = W^{(2)}(y - s) + r$$

$$z' = h' \odot \frac{\partial h}{\partial z} = \left(W^{(2)}(y - s) + r\right) \odot \sigma'(z)$$

is a column vector

elementwise because h is an element wise function of z

$$x' = \underbrace{\frac{\partial z}{\partial x}}_{\text{Jacobian}} z' = W^{(1)} z' = W^{(1)} \cdot \left[\left(W^{(2)}(y - s) + r\right) \odot \sigma'(z)\right] = \frac{\partial \mathcal{L}}{\partial x}$$

regular matrix multiplication

2 Vector-Jacobian Products (2pts)

Consider the function $f : \mathbb{R}^n \to \mathbb{R}^n$ where $f(\boldsymbol{x}) = \boldsymbol{v}\boldsymbol{v}^T\boldsymbol{x}$, and $\boldsymbol{v} \in \mathbb{R}^{n\times 1}$ and $\boldsymbol{x} \in \mathbb{R}^{n\times 1}$. Here, we will explore the relative costs of evaluating Jacobians and vector-Jacobian products. We denote the Jacobian of f with respect to $\boldsymbol{x}$ as $\boldsymbol{J} \in \mathbb{R}^{n\times n}$.

1. **(0.5 pt)** Compute $\boldsymbol{J}$ for $n = 3$ and $\boldsymbol{v}^T = [1, 2, 3]$ and write down all its values.
2. **(0.5 pt)** What is the time and memory cost of evaluating the Jacobian of the function f as a function of n?
3. **(1.0 pt)** Describe how to evaluate $\boldsymbol{J}^T\boldsymbol{y}$ where $\boldsymbol{y} \in \mathbb{R}^n$ with a time and memory cost that is linear in n. Then, compute $\boldsymbol{z} = \boldsymbol{J}^T\boldsymbol{y}$ where $\boldsymbol{v}^T = [1, 2, 3]$ and $\boldsymbol{y} = [1, 1, 1]^T$.

it's constant?

1.

$$J = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} \\ \frac{\partial f_3}{\partial x_1} & \frac{\partial f_3}{\partial x_2} & \frac{\partial f_3}{\partial x_3} \end{pmatrix} = vv^T = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} (1\ 2\ 3) = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{pmatrix}$$

2. Not quite sure what is asked here. f is a linear function and therefore its Jacobian is a constant (see above), independent of x. So evaluating it comes at no computational cost at all, the matrix just needs to be saved in memory.

Or did you mean computing J?

Evaluation: $J(x)$ for $\frac{\partial J}{\partial x} = 0$ has no cost!

3. $J^T y$ $vv^T y$, so instead of computing $(vv^T)^T y = vv^T y$ (J) it is more efficient to compute the inner product $v^T y = t$ and then the scalar product $t \cdot v$. Both operations have linear time and memory cost, so the entire evaluation still has linear cost as well.

For $v^T = (1, 2, 3)$, $y^T = (1, 1, 1)$:

$$z = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} (1, 2, 3) \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} 6 = \begin{pmatrix} 6 \\ 12 \\ 18 \end{pmatrix}$$

Double check: $J^T y = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 6 \\ 12 \\ 18 \end{pmatrix}$ ✓

3 Linear Regression (3pts)

In this problem we will explore some curious properties of gradient descent and its variants for learning over-parameterized models. We will use linear regression for this purpose but some of these properties also hold for deep neural networks.

Given n pairs of input data with d features and scalar label $(\boldsymbol{x}_i, t_i) \in \mathbb{R}^d \times \mathbb{R}$, we wish to find a linear model $f(\boldsymbol{x}) = \hat{\boldsymbol{w}}^T \boldsymbol{x}$ with $\hat{\boldsymbol{w}} \in \mathbb{R}^d$ that minimizes the squared error of prediction on the training samples defined below. For concise notation, denote the data matrix $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ and the corresponding label vector $\mathbf{t} \in \mathbb{R}^n$. The training objective is to minimize the following loss:

$$\min_{\hat{\boldsymbol{w}}} \frac{1}{n} \sum_{i=1}^{n} (\hat{\boldsymbol{w}}^T \boldsymbol{x}_i - t_i)^2 = \min_{\hat{\boldsymbol{w}}} \frac{1}{n} \|\boldsymbol{X}\hat{\boldsymbol{w}} - \boldsymbol{t}\|_2^2 \tag{1}$$

Assume that $\boldsymbol{X}$ is full rank: $\boldsymbol{X}^T\boldsymbol{X}$ is invertible when $n > d$, and $\boldsymbol{X}\boldsymbol{X}^T$ is invertible otherwise. Note that when $d > n$, the problem is *underdetermined*, i.e., there are less training samples than parameters to be learned. This is analogous to learning an *overparameterized* model, which is common when training of deep neural networks.

1. **Deriving the Gradient (0.5pt)** Write down the gradient of the loss, w.r.t. the learned parameter vector $\hat{\boldsymbol{w}}$.
2. **Underparameterized Model (0.5pt)** = overdetermined
   (a) First consider the underparameterized $d < n$ case. Write down the solution obtained by gradient descent assuming training converges. Show your work. Is the solution unique?
   (b) Assume that the ground truth labels are generated by a linear target: $t_i = \boldsymbol{w}^{*T} \boldsymbol{x}_i$. Show that the solution in part (a) achieves perfect generalization when $d < n$, i.e., $\forall \boldsymbol{x} \in \mathbb{R}^d$, $(\boldsymbol{w}^{*T}\boldsymbol{x} - \hat{\boldsymbol{w}}^T\boldsymbol{x})^2 = 0$.

$X \in \mathbb{R}^{n \times d}$

$\Rightarrow X^T X \in \mathbb{R}^{d \times d}$, $X X^T \in \mathbb{R}^{n \times n}$

1.

$$\mathcal{L} = \frac{1}{n} \|Xw - t\|_2^2 = \frac{1}{n} (Xw - t)^T (Xw - t) = \frac{1}{n} (w^T X^T X w - t^T X w + t^T t - w^T X^T t)$$

constant factor, does not affect minimum

$$\Rightarrow \frac{\partial \mathcal{L}}{\partial w} = \frac{2}{n} (X^T X w - X^T t)$$

2. Gradient descent: $w_n = w_{n-1} - \alpha \left. \frac{\partial \mathcal{L}}{\partial w} \right|_{w = w_{n-1}}$, with some learning rate α and initial w_0.

So for absolute convergence we want: $w^* := w_{n^*} = w_{n^*-1}$, which is the case when we reached the minimum where

$$\left. \frac{\partial \mathcal{L}}{\partial w} \right|_{w = w^*} = 0 = X^T X w^* - X^T t$$

which leads to the well-known normal equations:

$$X^T X w^* = X^T t \overset{(a)}{\Rightarrow} w^* = (X^T X)^{-1} X^T t$$

if $(X^T X)^{-1}$ exists; X^+ is the pseudo inverse

Clearly, w^* is unique if it exists since it is simply a linear transformation of t.

(b) for $t_i = w^{*T} x_i$ we get $t = X w^*$ and w^* is the solution:

$$\hat{w} = (X^T X)^{-1} X^T t = (X^T X)^{-1} X^T X w^* = w^* \quad \checkmark$$

So clearly: $(w^{*T} x - \hat{w}^T x)^2 = (w^{*T} x - w^{*T} x)^2 = 0 \quad \forall x$

$= w^{*T}$

3. **Overparameterized Model: 2D Example (1pt)**

(a) Now consider the overparameterized $d > n$ case. We will first illustrate that there exists multiple empirical risk minimizers. For simplicity let $n = 1$ and $d = 2$. Choose $x_1 = [2, 1]$ and $t_1 = 2$, i.e., the one data point and all possible $\hat{w}$ lie on a 2D plane. Show that there exists infinitely many $\hat{w}$ satisfying $\hat{w}^T x_1 = t_1$ on a real line. Write down the equation of the line.

(b) We know that multiple empirical risk minimizers exist in overparameterized linear regression and there is only one true solution. Thus, it seems unlikely that gradient descent will generalize if it returns an arbitrary solution. However, we will show that gradient descent tends to find a certain solution with desirable properties. This phenomenon, known as *implicit regularization*, helps explain the success of gradient-based methods to train overparameterized models, like deep neural networks. First consider the 2-dimensional example in the previous part: starting from zero initialization i.e., $\hat{w}(0) = \mathbf{0}$, what is the direction of the gradient? You should write down a unit-norm vector. Does the direction change along the trajectory? Based on this geometric intuition, which solution - along the line of solutions - does gradient descent find? Provide a pictorial sketch or a short description of your reasoning.

(c) Give a geometric argument that among all the solutions on the line, the gradient descent solution from the previous part has the smallest Euclidean norm. *Hint:* Use the Pythogream Theorem.

3. (a) The equation of the line is $w^T x_1 = t_1 = w_x x_{1x} + w_y x_{1y}$

Or re-arranged: $w_y = \frac{t_1}{x_{1y}} - \frac{x_{1x}}{x_{1y}} w_x = 2 - 2 w_x$ for $x_1^T = (2,1)$, $t_1 = 2$

So one of the components of w is a free parameter. Choosing any arbitrary value for either w_x or w_y determines the value of the other to satisfy $w^T x_1 = t_1$.

any $w = \frac{t_1 x_1}{\|x_1\|_2^2} + \alpha \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} x_1 = \frac{2}{5}\begin{pmatrix} 2 \\ 1 \end{pmatrix} + \alpha \begin{pmatrix} 1 \\ -2 \end{pmatrix}$, $\alpha \in \mathbb{R}$ is a solution!

(b) Here: $X = x_1^T$, since $n = 1$. So the gradient for $w_0 = 0$ is:

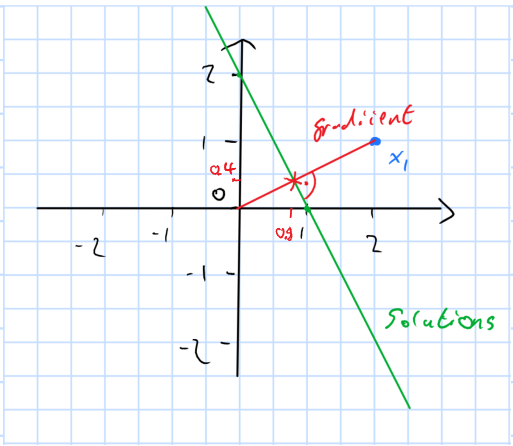
$$\left.\frac{\partial f}{\partial w}\right|_{w=0} = \frac{1}{n}\left(2X^T X w - 2X^T t\right) = -2x_1 t$$

So the direction of x_1. Normalized: $\frac{x_1}{\|x_1\|_2} = \frac{1}{\sqrt{5}}\begin{pmatrix} 2 \\ 1 \end{pmatrix}$

The direction of the gradient does not change along that trajectory:

$\alpha \in \mathbb{R}$: $\left.\frac{\partial f}{\partial w}\right|_{w=\alpha x_1} = 2x_1 x_1^T \alpha x - 2x_1 t = 2\alpha x_1 x_1^T x_1 - 2x_1 t$

$= \underbrace{2\left(\alpha \|x_1\|_2^2 - 1\right)}_{\text{some other constant}} x_1$, still points in x_1 direction!



The solution obtained by gradient descent will be $\frac{t}{\|x_1\|^2} x_1 = \frac{2}{5} x_1 = (0.8, 0.4)^T$. The closest point on the solution line to the starting point $w_0 = 0$.

(c) Gradient descent will always find the solution closest to the starting position w_0. Which is the intersection of $\frac{\epsilon_1 x_1}{\|x_1\|_2^2} + \alpha \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} x_1$ with $w_0 + \beta x_1$, $\alpha, \beta \in \mathbb{R}$

$\in$ range(x_1) $\quad$ $\in$ null(x_1)

so: $w^* = \frac{\epsilon_1 x_1}{\|x_1\|_2^2} + \alpha^* \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} x_1 = w_0 + \beta^* x_1$

It is easy to see, that to minimize the Euclidean norm $\|w^*\|_2^2$, $\alpha^* = 0$.

So any $w_0 = \frac{\epsilon_1 x_1}{\|x_1\|_2^2} - \beta^* x_1$ will find the solution with minimal Euclidean norm: $\hat{w} = \frac{\epsilon_1 x_1}{\|x_1\|_2^2} = \min_w \|w^*\|_2^2$

Just like in part (b) with $w_0 = 0$ $\left(\beta^* = \frac{\epsilon_1}{\|x_1\|_2^2}\right)$.

Simple argument in words: Starting at w_0 GD will always find the solution closest to w because the gradient always points from w to the hyperplane of solutions (perpendicular to the hyperplane), so $w_0 = 0$ will always lead to the solution with minimal Euclidean norm.

In fact, any $w_0 = \alpha x_1$ will converge to that solution (see part 4).

4. **Overparameterized Model: General Case (1pt)**

(a) Now we will generalize the previous geometric insight developed to general $d > n$. Show that gradient descent from zero initialization i.e., $\hat{w}(0) = \mathbf{0}$ finds a unique minimizer if it converges. Write down the solution and show your work. *Hint:* Recall the result on the direction of the gradient in the previous 2D example, which suggests that the gradient vector is always spanned by the rows of $\boldsymbol{X}$. What does this tell you about the gradient descent solution?

(b) Given the gradient descent solution from the previous part $\hat{w}$ and another zero-loss solution $\hat{w}_1$, evaluate $(\hat{w} - \hat{w}_1)^T \hat{w}$. Use this quantity to show that among all the empirical risk minimizers for $d > n$, the gradient descent solution has the smallest Euclidean norm.

In the case of $d > n$ and X of full rank, $Xw = t$ is underdetermined and has infinitely many solutions.
Any solution must have the form: $w = \{\hat{w} + w_\perp \mid w_\perp \in Null(X^T)\}$
where $\hat{w} = X^T(X X^T)^{-1} t$ is the only solution in $range(X^T)$ because XX^T has full rank!
(Easy to verify: $X\hat{w} = X X^T(XX^T)^{-1} t = t$)

It is easy to see this is the least norm solution, because any other solution w must satisfy:

$$Xw = X(\hat{w} + (w - \hat{w})) = X\hat{w} + X(w - \hat{w}) = t \Rightarrow X(w - \hat{w}) = 0$$

So $w - \hat{w} \in null(X)$ and $\|w\|_2 = \|\hat{w} + (w - \hat{w})\|_2 \geq \|\hat{w}\|_2$
because: $\hat{w} \perp (w - \hat{w})$

In words: $\hat{w} = X^T(X X^T)^{-1} t$ is the only solution of $Xw = t$ in the range of X^T. Adding any vector from null(X) to this unique solution also gives a solution, and increases the norm.
($range(X^T)$ is the span of the rows of X)

(a) Now, to show that GD converges to $\hat{w}$ for $w_0 = 0$:

Again: $L = \frac{1}{n}\|Xw - t\|_2^2 \overset{1.}{\Rightarrow} \frac{\partial L}{\partial w} = \frac{2}{n}(X^T X w - X^T t)$
$= \frac{2}{n} X^T (Xw - t)$

so $\left.\frac{\partial L}{\partial w}\right|_{w=0} = -\frac{2}{n} X^T t \in range(X^T)$ ($range(X^T)$ ← some vector)

and also: $X^T X w \in range(X^T) \ \forall w$. So $\frac{\partial L}{\partial w} \in range(X^T)$.

So for any $w_0 \in range(X^T)$, any gradient descent update will stay in $range(X^T)$ and the only solution in $range(X^T)$ is $\hat{w}$ as shown above. Therefore, if GD converges, it converges to $\hat{w}$ when starting in $range(X^T)$, for example by choosing $w_0 = 0$.

(b) I already showed this. But again:

$$X\vec{w}_1 = X(\vec{w} + (\vec{w}_1 - \vec{w})) = X\vec{w} + X(\vec{w}_1 - \vec{w}) = t \Rightarrow X(\vec{w}_1 - \vec{w}) = 0$$

which means that for any solution $\vec{w}_1 \neq \vec{w}$, $(\vec{w} - \vec{w}_1)$ is orthogonal to $\vec{w}$:

$$(\vec{w} - \vec{w}_1)^T \vec{w} = 0 = \vec{w}^T\vec{w} - \vec{w}_1^T\vec{w} \Rightarrow \vec{w}_1^T\vec{w} = \vec{w}^T\vec{w}$$

So: $(\vec{w} - \vec{w}_1)^T(\vec{w} - \vec{w}_1) = \vec{w}^T\vec{w} - 2\vec{w}_1^T\vec{w} + \vec{w}_1^T\vec{w}_1$

$(= \|\vec{w} - \vec{w}_1\|_2^2)$ $= \vec{w}_1^T\vec{w}_1 - \vec{w}^T\vec{w} = \|\vec{w}_1\|_2^2 - \|\vec{w}\|_2^2$

$$\Rightarrow \|\vec{w}_1\|_2^2 = \|\vec{w}\|_2^2 + \|\vec{w} - \vec{w}_1\|_2^2 \geq \|\vec{w}\|_2^2$$

Equality only for $\|\vec{w} - \vec{w}_1\|_2 = 0$ so $\vec{w} = \vec{w}_1$.

which means $\vec{w}$ is indeed the least norm solution.

□

5. Benefit of Overparameterization: Visualize and compare underparameterized with overparameterized polynomial regression (see attached jupyter notebook). Include your code snippets for the `fit_poly` function in the write-up. Does overparameterization (higher degree polynomial) always lead to overfitting, i.e., larger test error?

In [3]:
```
def fit_poly(X, d, t):
    X_expand = poly_expand(X, d=d, poly_type=poly_type)
    if d > n:
        W = X_expand.T@np.linalg.inv(X_expand@X_expand.T)@t
    else:
        W = np.linalg.inv(X_expand.T@X_expand)@X_expand.T@t
    return W
```

d | val_loss

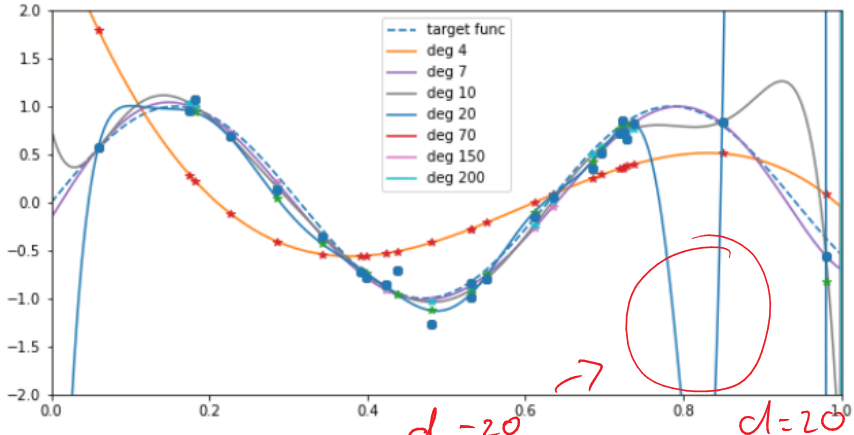
```
 2 0.23473638175555447
 3 0.19020505096352697
 4 0.24537346900176515
 5 0.029631242075101965
 7 0.002650135121507636
10 0.06641133900488616
15 9.582366859521946
20 3312.0732386744157
30 122.37598629957265
50 122.37598629957265
70 122.37598629957265
100 122.37598629957265
150 122.37598629957265
200 122.37598629957265
```

best!

worst!

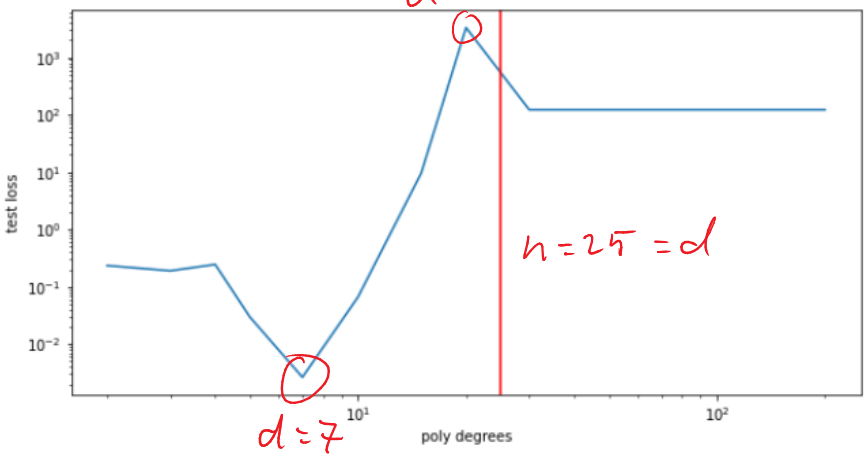
stable for d > n

for n = 25



d = 20

d = 20 < n = 25



Stable result for larger d because of implicit regularization!

Overparameterization does not always make the result worse here, because of the implicit L_2-regularization. However, generally one should always lean towards a smaller degree for polynomial interpolation, as clearly evident here.

4 Linear Regression and Optimization (3pts)
In this question we will study the solutions to linear regression from the variants of gradient descent that we saw in class.

1. **Stochastic Gradient Descent (1pt)** Recall that SGD estimates the gradient from a single randomly sampled training example and updates the parameters until convergence. In the overparameterized $d > n$ case, assume that SGD also converges to a solution $\hat{\boldsymbol{w}}$ such that $\boldsymbol{X}\hat{\boldsymbol{w}} = \boldsymbol{t}$. Show that the SGD solution is identical to the minimum norm solution $\boldsymbol{w}^*$ obtained by gradient descent, i.e., $\hat{\boldsymbol{w}} = \boldsymbol{w}^*$. *Hint:* Is $\boldsymbol{x}_i$ contained in the span of $\boldsymbol{X}$? Do the update steps SGD ever leave the span of $\boldsymbol{X}$?
2. **Mini-Batch SGD (1pt)** Recall that SGD estimates the gradient from mini-batches of data $\mathbf{B} \in \mathbb{R}^{b \times d}$, where $1 < b << n$ and $\boldsymbol{B}$ is taken from the rows of $\boldsymbol{X}$. In this case, does mini-batch SGD obtain the minimum norm solution on convergence?

1. Again: We have already shown in Problem 3 that

$w^* = X^T(X X^T)^{-1} t$ is the only solution in range(X^T) and that it is the least norm solution.

So here we choose some $\tilde{X} = \begin{bmatrix} \tilde{X}_1^T \\ \vdots \\ \tilde{X}_b^T \end{bmatrix} \in \mathbb{R}^{b \times d}$

where the $\tilde{X}_i^T$ are also rows of X. So: range$(\tilde{X}^T) \subseteq$ range(X^T).
And the gradient of the new loss function

$$\tilde{L} = \frac{1}{b} \| \tilde{X} \hat{w} - t \|_2^2 ,$$

$$\frac{\partial \tilde{L}}{\partial w} = \frac{2}{b} (\tilde{X}^T \tilde{X} w - \tilde{X}^T t) \in \text{range}(\tilde{X}^T) \subseteq \text{range}(X^T)$$

is still constraint to range(X^T). So if SD converges, the only solution it can converge to is still w^*, the least norm solution.

2. Same as in 1. just that we approximate ∇L:

$$\frac{\partial L}{\partial w} \approx \frac{1}{m} \sum_{i=1}^{m} \frac{\partial \tilde{L}}{\partial w} \Big|_{\tilde{X} = \text{random choice of } b \text{ rows from } X} .$$

So the approx gradient is a superposition of vectors (gradients) that are all elements of range(X^T). Therefore, the approximated gradient is itself still element of range(X^T) and can again only find the single solution in that set, which is the least norm solution w^*.

3. **Adaptive Methods (1pt)** We will now consider the behavior of adaptive gradient descent methods, in particular the AdaGrad[a] method. Let w_i denote the $i-$th parameter. A scalar learning rate η is used. At time t for parameter i, the update step for AdaGrad is as follows:

$$w_{t+1,i} = w_{t,i} - \frac{\eta}{\sqrt{G_{t,i} + \epsilon}} \nabla_{w_i} \mathcal{L}(w_{t,i})$$

$$G_{t,i} = G_{t-1,i} + (\nabla_{w_i} \mathcal{L}(w))^2$$

The term ϵ is a fixed small scalar used for numerical stability. Intuitively, Adagrad can be thought of as adapting the learning rate in each dimension to efficiently move through badly formed curvatures.

Consider the overparameterized linear regression model ($d > n$). Assume that AdaGrad converges to a solution. Provide a proof or a counterexample for whether AdaGrad always obtains the minimum norm solution. *Hint:* Compute the 2D case i.e., $x_1^T = [2, 1]$, $w_0^T = [0, 0]$, and $t = [2]$.

Does this result hold true for other adaptive methods (RMSProp, Adam) in general? Yes!

[a] https://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf

My intuition is: No, because adjusting the components individually (not the entire gradient as a whole vector) allows the construction of vectors outside of range (X^T). Which also holds true for other adaptive methods that scale individual gradient components!

Counterexample: $X w = t \Leftrightarrow X_1^T w = t \Leftrightarrow (2,1) w = 2$

$$\Rightarrow w^* = X^T (X X^T)^{-1} t = X_1 (X_1^T X_1)^{-1} t$$

$$= \begin{pmatrix} 2 \\ 1 \end{pmatrix} \cdot \frac{1}{5} \cdot 2 = \frac{2}{5} X_1$$

is the least norm solution.

$$\frac{\partial \mathcal{L}}{\partial w} = 2 (X X_1^T w - X_1 t) = \begin{pmatrix} 8 & 4 \\ 4 & 2 \end{pmatrix} w - 4 X_1$$

In [1]:
```
import numpy as np
```

In [2]:
```
x1 = np.array([2, 1])
t = 2
```

In [3]:
```
def dL(w):
    return np.array([[8, 4], [4, 2]])@w - 4*x1
```

In [4]:
```
def ada_step(w, eta, G):
    grad = dL(w)
    w[0] -= eta/( np.sqrt(G[0]) )*grad[0]
    w[1] -= eta/( np.sqrt(G[1]) )*grad[1]

    G += grad**2
    return w, G
```

In [5]:
```
eta = 0.01
G = np.ones(2)
w_ada = np.zeros(2)
w_regular = np.zeros(2)

for i in range(1000000):
    w_ada, G = ada_step(w_ada, eta, G)
    w_regular -= eta*dL(w_regular)  # regular GD
```

In [9]:
```
print("Theoretical Solution:", 2/5*x1)
print("Gradient at theoretical Solution:", dL(2/5*x1))
print("GD Solution:", w_regular)
print("Gradient at GD Solution:", dL(w_regular))
print("Prediction of GD Solution:", x1@w_regular)
print("AdaGrad Solution:", w_ada)
print("Gradient at AdaGrad Solution:", dL(w_ada))
print("Prediction of Adagrad Solution:", x1@w_ada)
```

```
Theoretical Solution: [0.8 0.4]
Gradient at theoretical Solution: [0. 0.]
GD Solution: [0.8 0.4]
Gradient at GD Solution: [-5.32907052e-15 -2.66453526e-15]
Prediction of GD Solution: 1.9999999999999987
AdaGrad Solution: [0.68022007 0.63955986]
Gradient at AdaGrad Solution: [-6.37712105e-13 -3.18856053e-13]
Prediction of Adagrad Solution: 1.9999999999998406
```

≠ ≈ 0 ≈ 2 ≈ 0 ≈ 2 ✓

Clearly, AdaGrad converges to a different solution!