Due: 15 Oct 2020

1. **Linear LS.** Consider the linear least squares problem

$$\min_{x \in \mathbb{R}^d} \frac{1}{2}\|Ax - b\|_2^2$$

where $A \in \mathbb{R}^{n \times d}$ and has rank $k \leq \min\{n, d\}$. Prove that the solution to this problem is given by

$$x^* = V_k \Sigma_k^{-1} U_k^\intercal b$$

where U_k and V_k are the matrices consisting of the first k columns of U and V and Σ_k is the top-left k-by-k submatrix of Σ in an SVD of A.

Solution. Applying the definitions, we have that

$$U = \begin{bmatrix} U_k & \tilde{U} \end{bmatrix}, V = \begin{bmatrix} V_k & \tilde{V} \end{bmatrix}, \Sigma = \begin{bmatrix} \Sigma_k & 0 \\ 0 & 0 \end{bmatrix}.$$

The factor of $\frac{1}{2}$ does not affect the optimization problem, so we will drop it in future manipulations. Additionally, notice that since U and V are orthogonal, any subset of their columns forms an orthogonal matrix. In particular, U_k and V_k are orthogonal. Next, recalling that multiplications by orthogonal matrices do not affect the values of norms, we manipulate the objective:

$$\|Ax - b\|_2^2 = \|U^\intercal(Ax - b)\|_2^2 = \|U^\intercal(U\Sigma V^\intercal x - b)\|_2^2 = \|\Sigma V^\intercal x - U^\intercal b\|_2^2.$$

Replace all matrices with their blocks forms:

$$\|\Sigma V^\intercal x - U^\intercal b\|_2^2 = \left\| \begin{bmatrix} \Sigma_k & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} V_k^\intercal \\ \tilde{V}^\intercal \end{bmatrix} x - \begin{bmatrix} U_k^\intercal \\ \tilde{U}^\intercal \end{bmatrix} \right\|_2^2 = \|\Sigma_k V_k^\intercal x - U_k^\intercal b\|_2^2 + \|\tilde{U}^\intercal b\|_2^2.$$

The second term of this equation is what cannot be controlled by changing x, i.e., the distance between x and the column space of A. The minimizer x^* is the vector that sends the first term of this equation to 0:

$$\begin{aligned} \Sigma_k V_k^\intercal x^* - U_k^\intercal b &= 0 \\ \implies \Sigma_k V_k^\intercal x^* &= U_k^\intercal b \\ \implies V_k^\intercal x^* &= \Sigma_k^{-1} U_k^\intercal b \\ \implies x^* &= V_k \Sigma_k^{-1} U_k^\intercal b \end{aligned}$$

is the minimizer, as desired.

2. **Constrained minimization.** Consider the constrained minimization problem

$$\min_{x \in \mathbb{R}^d} \frac{1}{2}\|Ax - b\|_2^2 \qquad \text{subject to} \qquad t - \|x\|_1 \geq 0$$

where t is a given constant. Set up a quadratic programming problem to solve this problem. Do it using the decomposition $x = x_+ - x_-$ where x_+ and x_- are the vectors with nonnegative components discussed in class. You do not need to solve it.

Solution. First, note that by decomposing x as above, we obtain that $\|x\|_1 = \|x_+\|_1 + \|x_-\|_1$. As such, the condition $t - \|x\|_1 \leq 0$ becomes $\|x_+\|_1 + \|x_-\|_1 \leq t$, which can be written in matrix form as

$$\begin{bmatrix} \mathbb{1}_d & \mathbb{1}_d \end{bmatrix} \begin{bmatrix} x_+ \\ x_- \end{bmatrix} = \mathbb{1}_{2d} \begin{bmatrix} x_+ \\ x_- \end{bmatrix} \leq t$$

where $\mathbb{1}_{2d}$ denotes the vector containing $2d$ ones. Additionally, all of the components of x_+ and x_- must be greater than or equal to 0 due to the definition of this decomposition of x. To incorporate this specification into the constraints, we note that $x_+ \geq 0$ (elementwise) is equivalent to $-x_+ \leq 0$ (elementwise) and write it in matrix form:

$$-I_{2d} \begin{bmatrix} x_+ \\ x_- \end{bmatrix} \leq 0$$

where I_{2d} is the $(2d) \times (2d)$ identity. Combining these matrices gives us the following matrix form of the constraints:

$$\begin{bmatrix} \mathbb{1}_{2d} \\ -I_{2d} \end{bmatrix} \begin{bmatrix} x_+ \\ x_- \end{bmatrix} \leq \begin{bmatrix} t \\ 0 \end{bmatrix}$$

where the bottom $-I_{2d}$ is a $(2d) \times (2d)$ identity matrix occupying the entire lower block of the matrix.

To get the objective function, we expand $f(x) = \frac{1}{2}\|Ax - b\|_2^2$ and simplify it.

$$\begin{aligned} f(x) &= \frac{1}{2}\|Ax - b\|_2^2 = \frac{1}{2}(Ax-b)^\intercal(Ax-b) \\ &= \frac{1}{2}(x^\intercal A^\intercal - b^\intercal)(Ax - b) \\ &= \frac{1}{2}x^\intercal A^\intercal Ax - x^\intercal A^\intercal b - b^\intercal Ax + b^\intercal b \\ &= \frac{1}{2}x^\intercal A^\intercal Ax - 2b^\intercal Ax + b^\intercal b \\ &= \frac{1}{2}x^\intercal A^\intercal Ax - 2b^\intercal Ax. \end{aligned}$$

We pause to note here that ignoring the constant term $b^\intercal b$ is permissible, as it does not affect the optimization problem. Continuing, we apply the decomposition:

$$\begin{aligned} f(x) &= \frac{1}{2}(x_+ - x_-)^\intercal A^\intercal A(x_+ - x_-) - 2b^\intercal A(x_+ - x_-) \\ &= \frac{1}{2}\begin{bmatrix} x_+^\intercal & x_-^\intercal \end{bmatrix} \begin{bmatrix} A^\intercal A & -A^\intercal A \\ -A^\intercal A & A^\intercal A \end{bmatrix} \begin{bmatrix} x_+ \\ x_- \end{bmatrix} - 2b^\intercal Ax_+ + 2b^\intercal Ax_- \\ &= \frac{1}{2}\begin{bmatrix} x_+^\intercal & x_-^\intercal \end{bmatrix} \begin{bmatrix} A^\intercal A & -A^\intercal A \\ -A^\intercal A & A^\intercal A \end{bmatrix} \begin{bmatrix} x_+ \\ x_- \end{bmatrix} + 2\begin{bmatrix} -b^\intercal A & b^\intercal A \end{bmatrix} \begin{bmatrix} x_+ \\ x_- \end{bmatrix}. \end{aligned}$$

So the final quadratic programming problem is

$$f(x) = \frac{1}{2}\begin{bmatrix} x_+^\intercal & x_-^\intercal \end{bmatrix} \begin{bmatrix} A^\intercal A & -A^\intercal A \\ -A^\intercal A & A^\intercal A \end{bmatrix} \begin{bmatrix} x_+ \\ x_- \end{bmatrix} + 2\begin{bmatrix} -b^\intercal A & b^\intercal A \end{bmatrix} \begin{bmatrix} x_+ \\ x_- \end{bmatrix} \text{ subject to}$$
$$\begin{bmatrix} \mathbb{1}_{2d} \\ -I_{2d} \end{bmatrix} \begin{bmatrix} x_+ \\ x_- \end{bmatrix} \leq \begin{bmatrix} t \\ 0 \end{bmatrix}.$$

3. **Build your own.** Reproduce any example from Section IV of the given paper of your choice except for Problem 5 (which was reproduced for us). The choice of the architecture of the NN, optimization method, and software is up to you. All examples in the paper are solved using NNs with only one hidden layer. There are codes for Problem 5 posted on ELMS under Files/Codes/NLLS. In your report, write the setup for the optimization problem and specify the optimization method and software developed or used. Also, compare your solution with the exact solution and with the numerical solution in the paper.

Solution. I chose to reproduce problem 1, which is

$$\frac{d\psi}{dx} + (x + \frac{1+3x^2}{1+x+x^3})\psi = x^3 + 2x + x^2\frac{1+3x^2}{1+x+x^3}.$$

To do this, I used the trial function $\psi_t(x) = 1 + xN(x,w)$ as suggested in the paper, where w is a vector of weights and biases for the neural network, and loss function

$$\ell(w) = \sum_{i=1}^{n} \left[\frac{d\psi}{dx}(x_i) - f(x_i, \psi_t(x_i)) \right]^2$$

where n is the number of mesh points and f is the forcing function. The derivative of the trial function is $\frac{d\psi}{dx} = N(x,w) + xN_x(x,w)$ by the product rule; I used a finite difference to compute this (discussed later). Let H be the number of nodes I used in my neural network; then the architecture of my neural network is

$$N(x,w) = v^\intercal \sigma(Wx + u)$$

where

- $v \in \mathbb{R}^H$ is a vector of weights
- $W \in \mathbb{R}^H$ is a vector of weights
- $u \in \mathbb{R}^H$ is a vector of biases
- $\sigma(z)$ is a nonlinear activation function; I tested tanh, sigmoid, and ReLU.

I began by coding this network in Python from scratch (i.e., only Numpy) using L-BFGS as an optimizer, but ran into a slew of errors which I was unable to resolve. As such, I decided to cut my losses and learn PyTorch. It was not too hard to get a basic one-layer neural network set up with PyTorch; the challenge was computing the derivative $\frac{dN}{dx}$ accurately.

To compute the derivative of the trial function for the loss calculation, I initially tried `torch.autograd.grad`, but was unable to get it to work—the network did not accurately predict the training set. This process was made much harder given that I was learning PyTorch for the first time to do this project! After unsuccessfully pursuing this for a few hours, I elected to switch to the following centered finite difference:

$$\frac{dN}{dx}(x) \approx \frac{N(x+\delta, w) - N(x-\delta, w)}{2\delta}$$

and used $\delta = 10^{-6}$ in my computations. I found this to immediately work very well.

For the optimization routine, I stuck to stochastic gradient descent with a learning rate of 0.01. I found that turning the learning rate higher than that consistently caused the gradients to blow up and not provide useful information.

My default settings were to set $n = 10$, $H = 100$, $\delta = 10^{-6}$, learning rate $= 0.01$, and number of epochs $= 1000$. Using those values, I trained the network on the interval $[0,1]$ and predicted the interval $[1,10]$ (Figures 1 and 2).

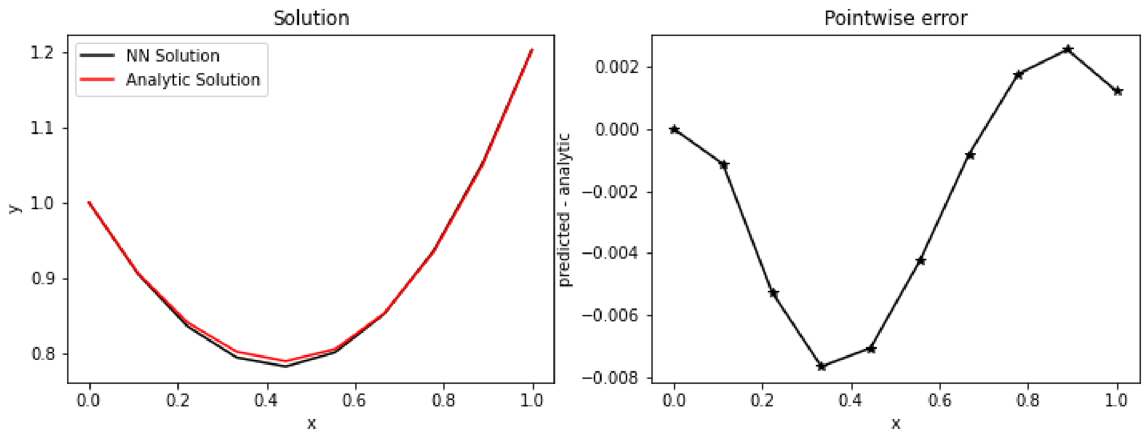

Figure 1: Results of training the network using tanh with default settings.

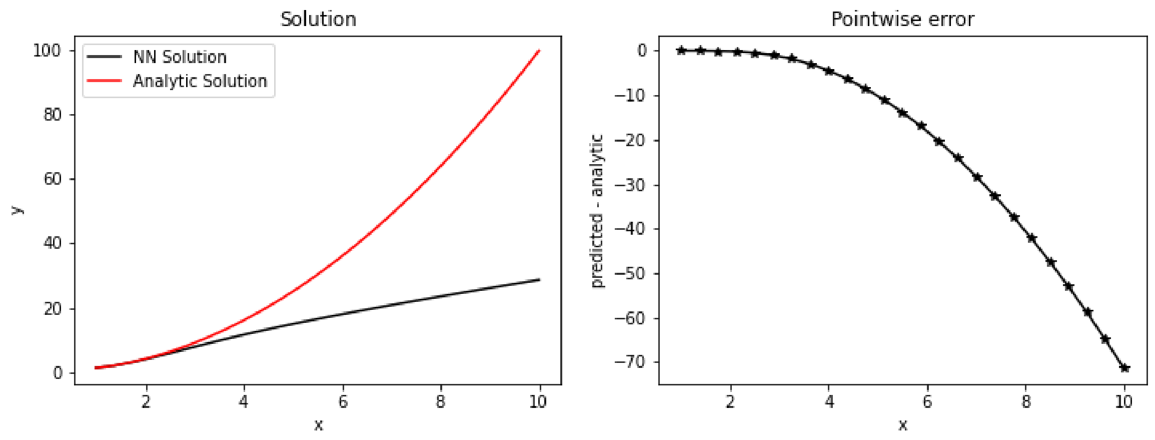

Figure 2: Results of training the network using tanh with default settings.

After that, I tested out the three activation functions to see if they made a difference. From there, I learned I should have actually been using a ReLU activation the whole time (Figure 3)!

I tried adding another layer but have as of yet been unsuccessful—there are some implementation details I have yet to work out. The attached code has the beginnings of my attempt, however.

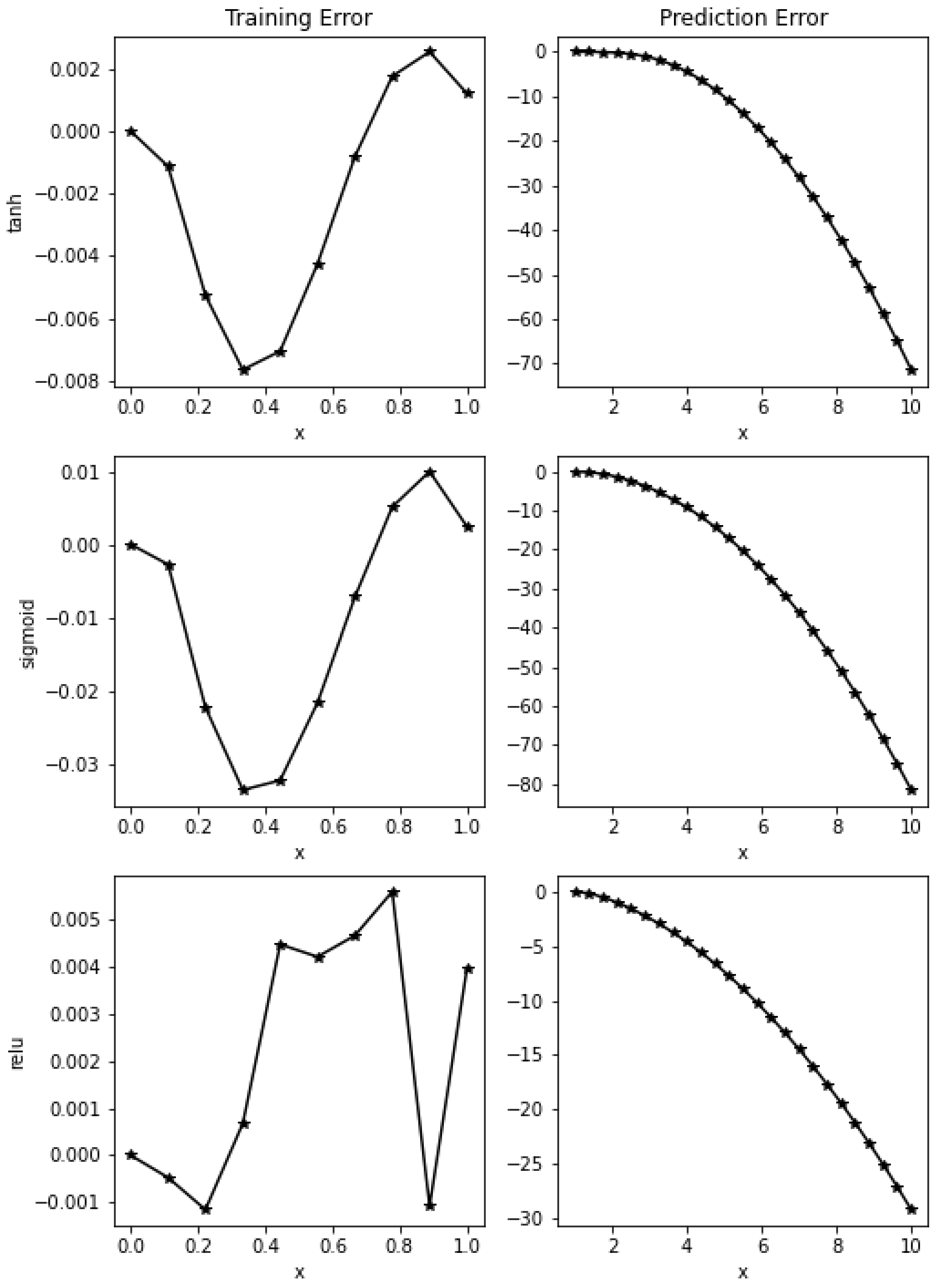

Figure 3: Training and prediction error for various activation functions. Note that the y axes are all different.