Homework 4 - Math Foundation for Machine Learning Engineers

kipngeno koech - bkoech

November 11, 2024

1 Warm Up Derivatives (10 Points)

Compute the first and second derivatives of the following functions:

1. $f(x) = \mathbf{M}^\top \mathbf{M}\mathbf{x}$, where $\mathbf{x} \in \mathbb{R}^m$ and $\mathbf{M} \in \mathbb{R}^{m \times m}$. What happens if $\mathbf{M} = \mathbf{M}^\top$? the first derivative is:

$$\frac{\partial f(x)}{\partial x} = \mathbf{M}^\top \mathbf{M}$$

the second derivative is:

$$\frac{\partial^2 f(x)}{\partial x^2} = 0$$

if $\mathbf{M} = \mathbf{M}^\top$:

$$f(x) = \mathbf{M}^\top \mathbf{M}\mathbf{x} = \mathbf{M}\mathbf{M}\mathbf{x} = \mathbf{M}^2\mathbf{x}$$

then the first derivative is:

$$\frac{\partial f(x)}{\partial x} = \mathbf{M}^2$$

the second derivative is:

$$\frac{\partial^2 f(x)}{\partial x^2} = 0$$

2. $f(\mathbf{X}) = \text{tr}(\mathbf{M}\mathbf{X})$, where $\mathbf{M} \in \mathbb{R}^{m \times n}$ and $\mathbf{X} \in \mathbb{R}^{n \times m}$, and tr is the trace of a square matrix.

$$f(\mathbf{X}) = \text{tr}(\mathbf{M}\mathbf{X}) = \text{tr}(\mathbf{X}\mathbf{M})$$

$$f(\mathbf{X}) = \text{tr}(\mathbf{X}\mathbf{M}) = \sum_{i=1}^{n} \sum_{j=1}^{m} \mathbf{X}_{ij}\mathbf{M}_{ji}$$

the first derivative is:

$$\frac{\partial f(\mathbf{X})}{\partial \mathbf{X}} = \mathbf{M}^\top$$

the second derivative is:

$$\frac{\partial^2 f(\mathbf{X})}{\partial \mathbf{X}^2} = 0$$

3. $f(\mathbf{X}) = \text{tr}(\mathbf{X}\mathbf{X}^\top)$, where $\mathbf{X} \in \mathbb{R}^{n \times n}$, and tr is the trace of a square matrix.

$$f(\mathbf{X}) = \text{tr}(\mathbf{X}\mathbf{X}^\top) = \text{tr}(\mathbf{X}^\top\mathbf{X})$$

$$f(\mathbf{X}) = \text{tr}(\mathbf{X}^\top\mathbf{X}) = \sum_{i=1}^{n} \sum_{j=1}^{n} \mathbf{X}_{ij}^2$$

the first derivative is:

$$\frac{\partial f(\mathbf{X})}{\partial \mathbf{X}} = 2\mathbf{X}$$

the second derivative is:

$$\frac{\partial^2 f(\mathbf{X})}{\partial \mathbf{X}^2} = 2$$

4. $f(\mathbf{X}) = \mathbf{a}^\top \mathbf{X}\mathbf{b}$, where $\mathbf{X} \in \mathbb{R}^{m \times n}$, $\mathbf{a} \in \mathbb{R}^m$ and $\mathbf{b} \in \mathbb{R}^n$.

$$f(\mathbf{X}) = \mathbf{a}^\top \mathbf{X}\mathbf{b} = \mathbf{b}^\top \mathbf{X}^\top \mathbf{a}$$

$$f(\mathbf{X}) = \mathbf{b}^\top \mathbf{X}^\top \mathbf{a} = \sum_{i=1}^{n} \sum_{j=1}^{m} \mathbf{X}_{ij} \mathbf{a}_j \mathbf{b}_i$$

the first derivative is:

$$\frac{\partial f(\mathbf{X})}{\partial \mathbf{X}} = \mathbf{a}\mathbf{b}^\top$$

the second derivative is:

$$\frac{\partial^2 f(\mathbf{X})}{\partial \mathbf{X}^2} = 0$$

5. $f(\mathbf{X}) = \mathbf{a}^\top \mathbf{X}^\top \mathbf{X}\mathbf{b}$, where $\mathbf{X} \in \mathbb{R}^{n \times m}$, $\mathbf{a} \in \mathbb{R}^m$ and $\mathbf{b} \in \mathbb{R}^m$.

$$f(\mathbf{X}) = \mathbf{a}^\top \mathbf{X}^\top \mathbf{X}\mathbf{b} = \mathbf{b}^\top \mathbf{X}^\top \mathbf{X}^\top \mathbf{a}$$

$$f(\mathbf{X}) = \mathbf{b}^\top \mathbf{X}^\top \mathbf{X}^\top \mathbf{a} = \sum_{i=1}^{m} \sum_{j=1}^{n} \mathbf{X}_{ji}^2 \mathbf{a}_j \mathbf{b}_i$$

the first derivative is:

$$\frac{\partial f(\mathbf{X})}{\partial \mathbf{X}} = 2\mathbf{X}\mathbf{b}\mathbf{a}^\top$$

the second derivative is:

$$\frac{\partial^2 f(\mathbf{X})}{\partial \mathbf{X}^2} = 2\mathbf{b}\mathbf{a}^\top$$

2 Ridge Regression (20 Points)

In ordinary least squares (OLS) regression, we seek to minimize the sum of squared residuals:

$$\min_{\boldsymbol{\beta}} \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2 \tag{1}$$

where $\mathbf{y}$ is the vector of observed target values and $\mathbf{X}$ is the matrix of input features.

We have shown in class and during recitation that the optimal parameters for OLS regression can be given by:

$$\boldsymbol{\beta} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y} \tag{2}$$

However, OLS can have several issues:

- The matrix $\mathbf{X}^\top \mathbf{X}$ may be non-invertible or ill-conditioned when features are correlated (multi-collinearity).
- OLS can lead to overfitting, especially when the number of features is large relative to the number of observations.
- OLS is sensitive to outliers since the cost function is based on squared residuals.

2.1 Ridge Regression Solution (10 Points)

Ridge regression modifies the OLS objective function by adding a penalty term:

$$\min_{\boldsymbol{\beta}} \left(\|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2 + \lambda \|\boldsymbol{\beta}\|^2 \right) \tag{3}$$

where λ is a regularization parameter that controls the strength of the penalty. Show that the optimal parameters $\boldsymbol{\beta}$ is:

$$\boldsymbol{\beta} = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{y} \tag{4}$$

where $\mathbf{I}$ is the identity matrix. This ensures that the solution is always unique (since $\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}$ is always invertible) and less sensitive to collinear features or outliers.

$$\min_{\boldsymbol{\beta}} \left(\|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2 + \lambda \|\boldsymbol{\beta}\|^2 \right)$$

$$\min_{\boldsymbol{\beta}} \left((\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^\top (\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) + \lambda \boldsymbol{\beta}^\top \boldsymbol{\beta} \right)$$

$$\min_{\boldsymbol{\beta}} \left(\mathbf{y}^\top \mathbf{y} - \mathbf{y}^\top \mathbf{X}\boldsymbol{\beta} - \boldsymbol{\beta}^\top \mathbf{X}^\top \mathbf{y} + \boldsymbol{\beta}^\top \mathbf{X}^\top \mathbf{X}\boldsymbol{\beta} + \lambda \boldsymbol{\beta}^\top \boldsymbol{\beta} \right)$$

$$\min_{\boldsymbol{\beta}} \left(\mathbf{y}^\top \mathbf{y} - 2\mathbf{y}^\top \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\beta}^\top \mathbf{X}^\top \mathbf{X}\boldsymbol{\beta} + \lambda \boldsymbol{\beta}^\top \boldsymbol{\beta} \right)$$

$$\frac{\partial}{\partial \boldsymbol{\beta}} \left(\mathbf{y}^\top \mathbf{y} - 2\mathbf{y}^\top \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\beta}^\top \mathbf{X}^\top \mathbf{X}\boldsymbol{\beta} + \lambda \boldsymbol{\beta}^\top \boldsymbol{\beta} \right) = 0$$

$$-2\mathbf{X}^\top \mathbf{y} + 2\mathbf{X}^\top \mathbf{X}\boldsymbol{\beta} + 2\lambda \boldsymbol{\beta} = 0$$

$$\mathbf{X}^\top \mathbf{X}\boldsymbol{\beta} + \lambda \boldsymbol{\beta} = \mathbf{X}^\top \mathbf{y}$$

$$(\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I})\boldsymbol{\beta} = \mathbf{X}^\top \mathbf{y}$$

$$\boldsymbol{\beta} = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{y}$$

2.2 Coding Ridge Regression (10 Points)

Complete the Ridge Regression code in the attached notebook and report the effect of changing the bias parameter.

3 Artificial Neural Network (60 Points)

In our recent recitation, we discussed artificial neural networks (ANNs) as complex functions that represent the intricate architecture of the human brain's neural network. These functions are subject to optimization to improve performance, a task often achieved through backpropagation and gradient descent, wherein the network adjusts its weights to minimize error and enhance accuracy in tasks such as classification and prediction.

We can summarize optimizing the model parameters (weights and biases) as follows:

Learning in deep neural networks using Gradient Descent can be summarized as follows:

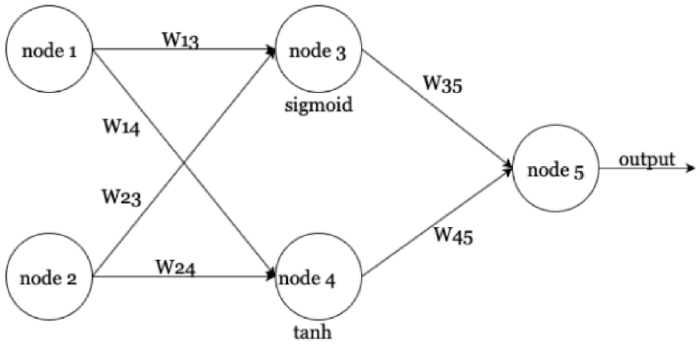
1. **Initialization:** Initialize the network's weights and biases, often randomly or by some heuristic.
2. **Forward Pass:** Input data is passed through the network. Each neuron computes a weighted sum of the inputs and a bias, followed by an activation function.
3. **Loss Calculation:** The network's prediction is compared to the true outcome using a loss function to calculate the error.
4. **Backward Pass (Backpropagation):** Calculate the gradient of the loss function with respect to each parameter using the chain rule.
5. **Gradient Descent:** Adjust the parameters in the opposite direction of the gradient to minimize the error.
6. **Update Parameters:** Update the weights by the equation:

   $$w_{\text{new}} = w_{\text{old}} - \alpha \cdot \nabla_w L,$$

   where α is the learning rate, $\nabla_w L$ is the gradient of the loss with respect to the weights, and w represents the weights.
7. **Iteration:** Repeat the forward pass, loss calculation, backpropagation, and parameter update until the loss converges or a stopping criterion is met.

3.1 Chain Rule and Backpropagation (20 Points)

Consider a network structured with two input nodes, two hidden nodes, and a single output node. Nodes 3 and 4 use the sigmoid and tanh functions, respectively. The remaining nodes are designed with linear activation, meaning that their outputs are directly proportional to their inputs.



We will follow the procedures described in the previous section.

1. **Initialization:** The weights are assigned as follows:
   The inputs of node 1 and node 2 are 1.0 and -1.0 respectively.

   $$W_{13} = 1, \quad W_{14} = -1, \quad W_{23} = 2, \quad W_{24} = -3, \quad W_{35} = 2, \quad W_{45} = -1.$$

   The hyperbolic tangent function is given by:

   $$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

   The sigmoid function is:

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

   There is no bias term in this network.

2. **Forward Pass (5 Points):** Compute the output of node 5.

   Node 3: $a_3 = \sigma(W_{13} \cdot x_1 + W_{23} \cdot x_2)$ Node 4: $a_4 = \tanh(W_{14} \cdot x_1 + W_{24} \cdot x_2)$ Node 5: $a_5 = W_{35} \cdot a_3 + W_{45} \cdot a_4$

   Node 3: $a_3 = \sigma(1 \cdot 1 + 2 \cdot -1)$ Node 4: $a_4 = \tanh(-1 \cdot 1 - 3 \cdot -1)$ Node 5: $a_5 = 2 \cdot a_3 + -1 \cdot a_4$

   Node 3: $a_3 = \sigma(1 - 2)$ Node 4: $a_4 = \tanh(-1 + 3)$ Node 5: $a_5 = 2 \cdot a_3 + -1 \cdot a_4$

   Node 3: $a_3 = \sigma(-1)$ Node 4: $a_4 = \tanh(2)$ Node 5: $a_5 = 2 \cdot a_3 + -1 \cdot a_4$

   Node 3: $a_3 = \dfrac{1}{1 + e^{-x}}$ Node 4: $a_4 = \dfrac{e^{2x} - 1}{e^{2x} + 1}$ Node 5: $a_5 = 2 \cdot a_3 + -1 \cdot a_4$

   Node 3: $a_3 = \dfrac{1}{1 + e^{1}}$ Node 4: $a_4 = \dfrac{e^{4} - 1}{e^{4} + 1}$ Node 5: $a_5 = 2 \cdot a_3 + -1 \cdot a_4$

   Node 3: $a_3 = \dfrac{1}{1 + 2.71828}$ Node 4: $a_4 = \dfrac{54.59815 - 1}{54.59815 + 1}$ Node 5: $a_5 = 2 \cdot a_3 + -1 \cdot a_4$

   Node 3: $a_3 = 0.26894$ Node 4: $a_4 = 0.96403$ Node 5: $a_5 = 2 \cdot 0.26894 + -1 \cdot 0.96403$

   Node 3: $a_3 = 0.26894$ Node 4: $a_4 = 0.96403$ Node 5: $a_5 = 0.53788 - 0.96403$

   Node 5: $a_5 =$ **-0.42614473734**

3. **Loss Calculation (2 Points):** The loss function is $L(y, \hat{y}) = (\hat{y} - y)^2$, where y is the output of node 5 (answer from previous part) and $\hat{y}$ is the desired output. In this part, we let $\hat{y} = 1.0$. Compute the loss of the network.

   $$L = (\hat{y} - y)^2 = (1.0 - (-0.42615))^2 = (1.42615)^2 = \mathbf{2.03388881183}$$

4. **Backpropagation (8 Points):** Calculate the gradient of the loss function with respect to each parameter using the chain rule. Remember the parameters are normally the weights and biases, but here we don't use a bias so the parameters are the weights. So for each weight, write down the derivative with respect to that weight

$$\frac{\partial L}{\partial W_{35}}, \quad \frac{\partial L}{\partial W_{45}}, \quad \frac{\partial L}{\partial W_{13}}, \quad \frac{\partial L}{\partial W_{14}}, \quad \frac{\partial L}{\partial W_{23}}, \quad \frac{\partial L}{\partial W_{24}}$$

derivative w.r.t to node 5:

$$\frac{\partial L}{\partial a_5} = -2 \cdot (\hat{y} - y) = -2 \cdot (1.0 - (-0.42615)) = \textbf{-2.8522894747}$$

derivative w.r.t to W_{35}:

$$\frac{\partial L}{\partial W_{35}} = \frac{\partial L}{\partial a_5} \cdot \frac{\partial a_5}{\partial W_{35}} = -2.8522894747 \cdot a_3 = -2.852 \cdot 0.26894142137 = \textbf{-0.76709878548}$$

derivative w.r.t to W_{45}:

$$\frac{\partial L}{\partial W_{45}} = \frac{\partial L}{\partial a_5} \cdot \frac{\partial a_5}{\partial W_{45}} = -2.8522894747 \cdot a_4 = -2.8522894747 \cdot 0.96402758008 = \textbf{-2.74968571998}$$

derivative w.r.t to node 4:

$$\frac{\partial L}{\partial a_4} = \frac{\partial L}{\partial a_5} \cdot \frac{\partial a_5}{\partial a_4} = -2.8522894747 \cdot W_{45} = -2.8522894747 \cdot -1 = \textbf{2.8522894747}$$

derivative w.r.t to tanh:

$$\frac{\partial L}{\partial tanh} = \frac{\partial L}{\partial a_4} \cdot \frac{\partial a_4}{\partial tanh} = 2.8522894747 \cdot \tanh'(W_{14} \cdot x_1 + W_{24} \cdot x_2)$$

$$= 2.8522894747 * (1 - 0.96402758008^2) = 2.8522894747 * (0.07065082485) = \textbf{0.20151660408}$$

derivative w.r.t to W_{14}:

$$\frac{\partial L}{\partial W_{14}} = \frac{\partial L}{\partial tanh} \cdot \frac{\partial a_4}{\partial W_{14}} = 0.20151660408 \cdot x_1 = 0.20151660408 \cdot 1 = \textbf{0.20151660408}$$

derivative w.r.t to W_{24}:

$$\frac{\partial L}{\partial W_{24}} = \frac{\partial L}{\partial tanh} \cdot \frac{\partial a_4}{\partial W_{24}} = 0.20151660408 \cdot x_2 = 0.20151660408 \cdot -1 = \textbf{-0.20151660408}$$

derivative w.r.t to node 3:

$$\frac{\partial L}{\partial a_3} = \frac{\partial L}{\partial a_5} \cdot \frac{\partial a_5}{\partial a_3} = -2.8522894747 \cdot W_{35} = -2.8522894747 \cdot 2 = \textbf{-5.7045789494}$$

N/B: the derivative of the sigmoid function is $\sigma'(x) = \sigma(x)(1 - \sigma(x))$
derivative w.r.t to sigmoid:

$$\frac{\partial L}{\partial \sigma} = \frac{\partial L}{\partial a_3} \cdot \frac{\partial a_3}{\partial \sigma} = -5.7045789494 \cdot \sigma'(W_{13} \cdot x_1 + W_{23} \cdot x_2)$$

$$= -5.7045789494 \cdot 0.26894142137 \cdot (1 - 0.26894142137) = -5.7045789494 \cdot 0.26894142137 \cdot 0.73105857863$$

$$= \textbf{-1.12158829557}$$

derivative w.r.t to W_{13}:

$$\frac{\partial L}{\partial W_{13}} = \frac{\partial L}{\partial \sigma} \cdot \frac{\partial a_3}{\partial W_{13}} = -1.12158829557 \cdot x_1 = -1.12158829557 \cdot 1 = \textbf{-1.12158829557}$$

derivative w.r.t to W_{23}:

$$\frac{\partial L}{\partial W_{23}} = \frac{\partial L}{\partial \sigma} \cdot \frac{\partial a_3}{\partial W_{23}} = -1.12158829557 \cdot x_2 = -1.12158829557 \cdot -1 = \textbf{1.12158829557}$$

5. **Gradient Descent and Parameter Update (5 Points):**Use the gradient descent optimization technique to update the parameters. Since we are minimizing the loss, we will go in the opposite direction of the gradient.

$$w_{\text{new}} = w_{\text{old}} - \alpha \cdot \nabla_w L,$$

use learning rate $\alpha = 0.1$.

After one iteration, report the weight values in the table below (keep only 2 decimal digits): example in table below
updating weights:
updating W_{13}:

$$W_{13_{\text{new}}} = W_{13_{\text{old}}} - \alpha \cdot \frac{\partial L}{\partial W_{13}} = 1 - 0.1 \cdot -1.12158829557 = \textbf{1.11215882956}$$

updating W_{14}:

$$W_{14_{\text{new}}} = W_{14_{\text{old}}} - \alpha \cdot \frac{\partial L}{\partial W_{14}} = -1 - 0.1 \cdot 0.20151660408 = \textbf{-1.02015166041}$$

updating W_{23}:

$$W_{23_{\text{new}}} = W_{23_{\text{old}}} - \alpha \cdot \frac{\partial L}{\partial W_{23}} = 2 - 0.1 \cdot 1.12158829557 = \textbf{1.88784117044}$$

updating W_{24}:

$$W_{24_{\text{new}}} = W_{24_{\text{old}}} - \alpha \cdot \frac{\partial L}{\partial W_{24}} = -3 - 0.1 \cdot -0.20151660408 = \textbf{-2.97984833959}$$

updating W_{35}:

$$W_{35_{\text{new}}} = W_{35_{\text{old}}} - \alpha \cdot \frac{\partial L}{\partial W_{35}} = 2 - 0.1 \cdot -0.76709878548 = \textbf{2.07670987855}$$

updating W_{45}:

$$W_{45_{\text{new}}} = W_{45_{\text{old}}} - \alpha \cdot \frac{\partial L}{\partial W_{45}} = -1 - 0.1 \cdot -2.74968571998 = \textbf{-0.72503142800}$$

W_{13}	W_{14}	W_{23}	W_{24}	W_{35}	W_{45}
1.11	-1.02	1.88	-2.98	2.08	-0.73

3.2 Checkpoint 1

1. Are you comfortable calculating the derivatives ? **Yes**
2. Are you comfortable applying the gradient descent ? **Yes**
3. Are you comfortable completing the previous section ? **Yes**

If your answer is NO, please post on piazza, attend office hour for a TA to help you understand. Do not move to next section. You might see this in your exam.

If your answer is Yes, congratulations, you learned some of the basics of neural network and it will help you in the Introduction to deep learning and Introduction to machine learning course.

3.3 Bonus (5 Points)

Run another iteration (5 points) Use the updated weights of previous section. However the inputs of node 1 and node 2 are 2.0 and -2.0 respectively, the desired output is 2. What is your predicted output, and what is your updated weights ?
weights to use:

$$W_{13} = 1.11, \quad W_{14} = -1.02, \quad W_{23} = 1.88, \quad W_{24} = -2.98, \quad W_{35} = 2.08, \quad W_{45} = -0.73$$

inputs:

$$x_1 = 2.0, \quad x_2 = -2.0, \quad \hat{y} = 2.0$$

Forward Pass:

Node 3: $a_3 = \sigma(W_{13} \cdot x_1 + W_{23} \cdot x_2)$ Node 4: $a_4 = \tanh(W_{14} \cdot x_1 + W_{24} \cdot x_2)$ Node 5: $a_5 = W_{35} \cdot a_3 + W_{45} \cdot a_4$

Node 3: $a_3 = \sigma(1.11 \cdot 2.0 + 1.88 \cdot -2.0)$ Node 4: $a_4 = \tanh(-1.02 \cdot 2.0 - 2.98 \cdot -2.0)$ Node 5: $a_5 = 2.08 \cdot a_3 + -0.73 \cdot a_4$

Node 3: $a_3 = \sigma(2.22 - 3.76)$ Node 4: $a_4 = \tanh(-2.04 + 5.96)$ Node 5: $a_5 = 2.08 \cdot a_3 + -0.73 \cdot a_4$

Node 3: $a_3 = \sigma(-1.54)$ Node 4: $a_4 = \tanh(3.92)$ Node 5: $a_5 = 2.08 \cdot a_3 + -0.73 \cdot a_4$

Node 3: $a_3 = \dfrac{1}{1 + e^{-1.54}}$ Node 4: $a_4 = \dfrac{e^{7.84} - 1}{e^{7.84} + 1}$ Node 5: $a_5 = 2.08 \cdot a_3 + -0.73 \cdot a_4$

Node 3: $a_3 = \dfrac{1}{1 + 4.65235195767}$ Node 4: $a_4 = \dfrac{2455.029 - 1}{2455.029 + 1}$ Node 5: $a_5 = 2.08 \cdot a_3 + -0.73 \cdot a_4$

Node 3: $a_3 = \dfrac{1}{5.65235195767}$ Node 4: $a_4 = \dfrac{2454.029}{2456.029}$ Node 5: $a_5 = 2.08 \cdot a_3 + -0.73 \cdot a_4$

Node 3: $a_3 = 0.1776535274$ Node 4: $a_4 = 0.999212971$ Node 5: $a_5 = 2.08 \cdot 0.1776535272 + -0.73 \cdot 0.999121971$

Node 5: $a_5 = -0.3598839702$

Loss Calculation:

$$L = (\hat{y} - y)^2 = (2.0 - (-0.3598839702))^2 = (2.3598839702)^2 = 5.5698839702$$

Backpropagation:

$$\frac{\partial L}{\partial y} = -2 \cdot (\hat{y} - y) = -2 \cdot (2.0 - (-0.3598839702)) = -2 \cdot 2.3598839702 = \textbf{-4.719679405}$$

$$\frac{\partial L}{\partial W_{35}} = \frac{\partial L}{\partial a_5} \cdot \frac{\partial a_5}{\partial W_{35}} = -4.719679405 \cdot a_3 = -4.719679405 \cdot 0.1776535274 = \textbf{-0.838422073}$$

$$\frac{\partial L}{\partial W_{45}} = \frac{\partial L}{\partial a_5} \cdot \frac{\partial a_5}{\partial W_{45}} = -4.719679405 \cdot a_4 = -4.719679405 \cdot 0.999212971 = \textbf{-4.715996686}$$

$$\frac{\partial L}{\partial a_4} = \frac{\partial L}{\partial a_5} \cdot \frac{\partial a_5}{\partial a_4} = -4.719679405 \cdot W_{45} = -4.719679405 \cdot -0.73 = \textbf{3.445365966}$$

$$\frac{\partial L}{\partial tanh} = \frac{\partial L}{\partial a_4} \cdot \frac{\partial a_4}{\partial tanh} = 3.445365966 \cdot \tanh'(W_{14} \cdot x_1 + W_{24} \cdot x_2)$$

$$= 3.445365966 \cdot (1 - 0.999212971^2) = 3.445365966 \cdot 0.0001573996059 = \textbf{0.000542394}$$

$$\frac{\partial L}{\partial W_{14}} = \frac{\partial L}{\partial tanh} \cdot \frac{\partial a_4}{\partial W_{14}} = 0.000542394 \cdot x_1 = 0.000542394 \cdot 2 = \textbf{0.001084788}$$

$$\frac{\partial L}{\partial W_{24}} = \frac{\partial L}{\partial tanh} \cdot \frac{\partial a_4}{\partial W_{24}} = 0.000542394 \cdot x_2 = 0.000542394 \cdot -2 = \textbf{-0.001084788}$$

$$\frac{\partial L}{\partial a_3} = \frac{\partial L}{\partial a_5} \cdot \frac{\partial a_5}{\partial a_3} = -4.719679405 \cdot W_{35} = -4.719679405 \cdot 2.08 = \textbf{-9.816933162}$$

$$\frac{\partial L}{\partial \sigma} = \frac{\partial L}{\partial a_3} \cdot \frac{\partial a_3}{\partial \sigma} = -9.816933162 \cdot \sigma'(W_{13} \cdot x_1 + W_{23} \cdot x_2)$$

$$= -9.816933162 \cdot 0.1776535274 \cdot (1 - 0.1776535274) = -9.816933162 \cdot 0.1776535274 \cdot 0.8223464726 = \textbf{-1.441073}$$

$$\frac{\partial L}{\partial W_{13}} = \frac{\partial L}{\partial \sigma} \cdot \frac{\partial a_3}{\partial W_{13}} = -1.441073 \cdot x_1 = -1.441073 \cdot 2 = \textbf{-2.882146}$$

$$\frac{\partial L}{\partial W_{23}} = \frac{\partial L}{\partial \sigma} \cdot \frac{\partial a_3}{\partial W_{23}} = -1.441073 \cdot x_2 = -1.441073 \cdot -2 = \textbf{2.882146}$$

Gradient Descent and Parameter Update:

$$weight_{\text{new}} = weight_{\text{old}} - \alpha \cdot \nabla_w L$$

$$\alpha = 0.1$$

$$W_{13_{\text{new}}} = W_{13_{\text{old}}} - \alpha \cdot \frac{\partial L}{\partial W_{13}} = 1.11 - 0.1 \cdot -2.882146 = \textbf{1.3982146}$$

$$W_{14_{\text{new}}} = W_{14_{\text{old}}} - \alpha \cdot \frac{\partial L}{\partial W_{14}} = -1.02 - 0.1 \cdot 0.001084788 = \textbf{-1.019891521}$$

$$W_{23_{\text{new}}} = W_{23_{\text{old}}} - \alpha \cdot \frac{\partial L}{\partial W_{23}} = 1.88 - 0.1 \cdot 2.882146 = \textbf{1.5917854}$$

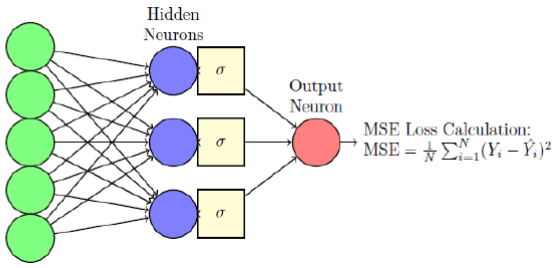
$$W_{24_{\text{new}}} = W_{24_{\text{old}}} - \alpha \cdot \frac{\partial L}{\partial W_{24}} = -2.98 - 0.1 \cdot -0.001084788 = \textbf{-2.979891521}$$

$$W_{35_{\text{new}}} = W_{35_{\text{old}}} - \alpha \cdot \frac{\partial L}{\partial W_{35}} = 2.08 - 0.1 \cdot -0.838422073 = \textbf{2.1638422073}$$

$$W_{45_{\text{new}}} = W_{45_{\text{old}}} - \alpha \cdot \frac{\partial L}{\partial W_{45}} = -0.73 - 0.1 \cdot -4.715996686 = \textbf{-0.2584003314}$$

W_{13}	W_{14}	W_{23}	W_{24}	W_{35}	W_{45}
1.40	-1.02	1.59	-2.98	2.16	-0.26

3.4 One Layer MLP (40 Points)



In a one-layer Multi-Layer Perceptron (MLP) with a single hidden layer, we represent the weights as matrices to efficiently handle the numerous connections between layers. For a network with m input neurons and n hidden neurons, the weights and biases are defined as follows:

- Weights of the Hidden Layer ($\mathbf{W}_h$): An $m \times n$ matrix.
- Biases of the Hidden Layer ($\mathbf{b}_h$): An n-dimensional vector.
- Weights of the Output Layer ($\mathbf{W}_o$): An $n \times k$ matrix (assuming k output neurons).
- Biases of the Output Layer ($\mathbf{b}_o$): A k-dimensional vector.

The sigmoid activation function is defined as:

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

Its derivative, which is used in the computation of the gradient, is:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

The Mean Squared Error (MSE) loss function is:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(Y_i - \hat{Y}_i)^2$$

where N is the number of samples, Y is the true output, and $\hat{Y}$ is the predicted output.
forward pass

$$\mathbf{h} = \sigma(\mathbf{X}\mathbf{W}_h + \mathbf{b}_h)$$

Write down the gradients of the MSE with respect to the following parameters (15 points): the gradient of the loss w.r.t to the predicted output $\hat{Y}$ is:

$$\frac{\partial MSE}{\partial \hat{Y}} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial}{\partial \hat{Y}}(Y_i - \hat{Y}_i)^2$$

$$\frac{\partial MSE}{\partial \hat{Y}} = \frac{1}{N}\sum_{i=1}^{N} -2(Y_i - \hat{Y}_i) = -\frac{2}{N}\sum_{i=1}^{N}(Y_i - \hat{Y}_i)$$

1. Gradient with respect to $\mathbf{W}_o$:

$$\frac{\partial MSE}{\partial \mathbf{W}_o}$$

the forward equation here is:

$$\hat{Y} = \hat{\mathbf{h}}\mathbf{W}_o + \mathbf{b}_o$$

so the gradient of the loss w.r.t to the weight $\mathbf{W}_o$ is:

$$\frac{\partial MSE}{\partial \mathbf{W}_o} = \frac{\partial MSE}{\partial \hat{Y}} \cdot \frac{\partial \hat{Y}}{\partial \mathbf{W}_o}$$

$$\frac{\partial MSE}{\partial \mathbf{W}_o} = -\frac{2}{N}\sum_{i=1}^{N}(Y_i - \hat{Y}_i) \cdot \frac{\partial}{\partial \mathbf{W}_o}(\hat{\mathbf{h}}_i\mathbf{W}_o + \mathbf{b}_o)$$

$$\frac{\partial MSE}{\partial \mathbf{W}_o} = -\frac{\mathbf{2}}{\mathbf{N}}\sum_{\mathbf{i=1}}^{\mathbf{N}}(\mathbf{Y_i} - \hat{\mathbf{Y}}_\mathbf{i}) \cdot \hat{\mathbf{h}}_\mathbf{i}$$

2. Gradient with respect to $\mathbf{b}_o$:

$$\frac{\partial MSE}{\partial \mathbf{b}_o}$$

$$\frac{\partial MSE}{\partial \mathbf{b}_o} = \frac{\partial MSE}{\partial \hat{Y}} \cdot \frac{\partial \hat{Y}}{\partial \mathbf{b}_o}$$

$$\frac{\partial MSE}{\partial \mathbf{b}_o} = -\frac{2}{N}\sum_{i=1}^{N}(Y_i - \hat{Y}_i) \cdot \frac{\partial}{\partial \mathbf{b}_o}(\hat{\mathbf{h}}_i\mathbf{W}_o + \mathbf{b}_o)$$

$$\frac{\partial MSE}{\partial \mathbf{b}_o} = -\frac{2}{N}\sum_{i=1}^{N}(Y_i - \hat{Y}_i) \cdot 1 = -\frac{\mathbf{2}}{\mathbf{N}}\sum_{\mathbf{i=1}}^{\mathbf{N}}(\mathbf{Y_i} - \hat{\mathbf{Y}}_\mathbf{i})$$

3. Gradient with respect to $\mathbf{W}_h$:

$$\frac{\partial MSE}{\partial \mathbf{W}_h}$$

the forward equation for the hidden layer is: $\hat{\mathbf{h}} = \sigma(h)$ where $h = \mathbf{X}\mathbf{W}_h + \mathbf{b}_h$
Gradient of the M.S.E loss with respect to the output of the hidden layer $\hat{h}$ is:

$$\frac{\partial MSE}{\partial \hat{\mathbf{h}}} = \frac{\partial MSE}{\partial \hat{Y}} \cdot \frac{\partial \hat{Y}}{\partial \hat{\mathbf{h}}}$$

$$\frac{\partial MSE}{\partial \hat{\mathbf{h}}} = \frac{\partial MSE}{\partial \hat{Y}} \cdot \frac{\partial \hat{Y}}{\partial \hat{\mathbf{h}}} = \frac{\partial MSE}{\partial \hat{Y}} \cdot \mathbf{W}_o$$

derivative of the loss w.r.t to sigmoid:

$$\frac{\partial MSE}{\partial \sigma} = \frac{\partial MSE}{\partial \hat{\mathbf{h}}} \cdot \frac{\partial \hat{\mathbf{h}}}{\partial \sigma}$$

but the derivative of the sigmoid function is $\sigma'(x) = \sigma(x)(1 - \sigma(x))$, so :

$$\frac{\partial \hat{\mathbf{h}}}{\partial \sigma} = \hat{\mathbf{h}} \cdot (1 - \hat{\mathbf{h}})$$

so:

$$\frac{\partial \hat{\mathbf{h}}}{\partial \sigma} = \frac{\partial MSE}{\partial \hat{\mathbf{h}}} \cdot \frac{\partial \hat{\mathbf{h}}}{\partial \sigma} = \frac{\partial MSE}{\partial \hat{\mathbf{h}}} \cdot \hat{\mathbf{h}} \cdot (1 - \hat{\mathbf{h}})$$

finally, the gradient of the loss w.r.t to the weight $\mathbf{W}_h$ is:

$$\frac{\partial MSE}{\partial \mathbf{W}_h} = \frac{\partial MSE}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial \mathbf{W}_h}$$

$$\frac{\partial \sigma}{\partial \mathbf{W}_h} = \frac{\partial \sigma}{\partial h} \cdot \frac{\partial h}{\partial \mathbf{W}_h}$$

$$\frac{\partial \sigma}{\partial \mathbf{W_h}} = \mathbf{X}$$

the final derivative is:

$$\frac{\partial MSE}{\partial \mathbf{W}_h} = \frac{\partial MSE}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial \mathbf{W}_h} = \frac{\partial MSE}{\partial \hat{\mathbf{h}}} \cdot \frac{\partial \hat{\mathbf{h}}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial \mathbf{W}_h}$$

$$= \frac{\partial MSE}{\partial \hat{\mathbf{h}}} \cdot \hat{\mathbf{h}} \cdot (1 - \hat{\mathbf{h}}) \cdot \mathbf{X}$$

where $\frac{\partial MSE}{\partial \hat{\mathbf{h}}}$ is defined above

4. Gradient with respect to $\mathbf{b}_h$:

$$\frac{\partial MSE}{\partial \mathbf{b}_h}$$

$$\frac{\partial MSE}{\partial \mathbf{b}_h} = \frac{\partial MSE}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial \mathbf{b}_h}$$

$$\frac{\partial \sigma}{\partial \mathbf{b}_h} = 1$$

the final derivative is:

$$\frac{\partial MSE}{\partial \mathbf{b}_h} = \frac{\partial MSE}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial \mathbf{b}_h} = \frac{\partial MSE}{\partial \hat{\mathbf{h}}} \cdot \hat{h} \cdot (1 - \hat{h})$$

where $\frac{\partial MSE}{\partial \hat{\mathbf{h}}}$ is defined above

Write down the gradient descent update rules for the parameters using a learning rate of 0.1 (5 points).

$$w_{\text{new}} = w_{\text{old}} - \alpha \cdot \nabla_w L,$$
$$b_{\text{new}} = b_{\text{old}} - \alpha \cdot \nabla_b L,$$

where w represents the weights and b represents the biases.

For the $\mathbf{W}_h$ and $\mathbf{W}_o$ matrices, the update rules are:

$$\mathbf{W}_{h_{\text{new}}} = \mathbf{W}_{h_{\text{old}}} - \alpha \cdot \nabla_{\mathbf{W}_h} L,$$
$$\mathbf{W}_{o_{\text{new}}} = \mathbf{W}_{o_{\text{old}}} - \alpha \cdot \nabla_{\mathbf{W}_o} L,$$

for the $\mathbf{b}_h$ and $\mathbf{b}_o$ vectors, the update rules are:

$$\mathbf{b}_{h_{\text{new}}} = \mathbf{b}_{h_{\text{old}}} - \alpha \cdot \nabla_{\mathbf{b}_h} L,$$
$$\mathbf{b}_{o_{\text{new}}} = \mathbf{b}_{o_{\text{old}}} - \alpha \cdot \nabla_{\mathbf{b}_o} L,$$

Complete the MLP codes in the starter notebook (20 points).

Bonus (10 points): Add Another layer hidden layer to MLP. and complete the codes in the 2 Layer MLP. You should still use sigmoid activation on the new layer. What is the effect of adding another layer ?

Answer: The effect of adding another layer is that the model can capture more complex relationships in the data. The additional layer allows the model to learn more complex features in the data, which can help improve the performance of the model. However, adding more layers can also lead to overfitting since it will learn data-specific features. It is important to carefully tune the hyperparameters of the model to prevent overfitting when adding more layers.

The gradients: w.r.t to the output layer:

$$\frac{\partial MSE}{\partial \hat{Y}} = -\frac{2}{N}\sum_{i=1}^{N}(Y_i - \hat{Y}_i)$$

$$\frac{\partial MSE}{\partial \mathbf{W}_o} = -\frac{\mathbf{2}}{\mathbf{N}}\sum_{\mathbf{i=1}}^{\mathbf{N}}(\mathbf{Y_i} - \hat{\mathbf{Y}}_\mathbf{i}) \cdot \hat{\mathbf{h}}_\mathbf{i}$$

$$\frac{\partial MSE}{\partial \mathbf{b}_o} = -\frac{\mathbf{2}}{\mathbf{N}}\sum_{\mathbf{i=1}}^{\mathbf{N}}(\mathbf{Y_i} - \hat{\mathbf{Y}}_\mathbf{i})$$

$$\frac{\partial MSE}{\partial \mathbf{h}_2} = \frac{\partial MSE}{\partial \hat{Y}} \cdot \frac{\partial \hat{Y}}{\partial \mathbf{h}_2} = \frac{\partial MSE}{\partial \hat{Y}} \cdot \mathbf{W}_o$$

derivatives w.r.t to the hidden layer 2:

$$\frac{\partial MSE}{\partial \mathbf{h}_2} = \frac{\partial MSE}{\partial \hat{Y}} \cdot \frac{\partial \hat{Y}}{\partial \mathbf{h}_2} = \frac{\partial MSE}{\partial \hat{Y}} \cdot \mathbf{W}_o$$

it has a sigmoid activation function, so the derivative of the loss w.r.t to the sigmoid is:

$$\frac{\partial MSE}{\partial \sigma} = \frac{\partial MSE}{\partial \mathbf{h}_2} \cdot \frac{\partial \mathbf{h}_2}{\partial \sigma} = \frac{\partial MSE}{\partial \mathbf{h}_2} \cdot \mathbf{h}_2 \cdot (1 - \mathbf{h}_2)$$

derivative w.r.t to weight $\mathbf{W}_h 2$:

$$\frac{\partial MSE}{\partial \mathbf{W}_h 2} = \frac{\partial MSE}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial \mathbf{W}_h 2} = \frac{\partial MSE}{\partial \mathbf{h}_2} \cdot \mathbf{h}_2 \cdot (1 - \mathbf{h}_2) \cdot \mathbf{h}_1$$

derivative w.r.t to bias $\mathbf{b}_h 2$:

$$\frac{\partial MSE}{\partial \mathbf{b}_h 2} = \frac{\partial MSE}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial \mathbf{b}_h 2} = \frac{\partial MSE}{\partial \mathbf{h}_2} \cdot \mathbf{h}_2 \cdot (1 - \mathbf{h}_2)$$

derivative w.r.t to the hidden layer 1:

$$\frac{\partial MSE}{\partial \mathbf{h}_1} = \frac{\partial MSE}{\partial \mathbf{h}_2} \cdot \frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1} = \frac{\partial MSE}{\partial \mathbf{h}_2} \cdot \frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1} = \frac{\partial MSE}{\partial \mathbf{h}_2} \cdot \mathbf{W}_{h_2}$$

derivative w.r.t to hidden layer 1 weights: it has a sigmoid activation function, so the derivative of the loss w.r.t to the sigmoid is:

$$\frac{\partial MSE}{\partial \sigma} = \frac{\partial MSE}{\partial \mathbf{h}_1} \cdot \frac{\partial \mathbf{h}_1}{\partial \sigma} = \frac{\partial MSE}{\partial \mathbf{h}_2} \cdot \mathbf{W}_{h_2} \cdot \mathbf{h}_1 \cdot (1 - \mathbf{h}_1)$$

derivative w.r.t to weight $\mathbf{W}_h 1$:

$$\frac{\partial MSE}{\partial \mathbf{W}_h 1} = \frac{\partial MSE}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial \mathbf{W}_h 1} = \frac{\partial MSE}{\partial \mathbf{h}_1} \cdot \mathbf{h}_1 \cdot (1 - \mathbf{h}_1) \cdot \mathbf{X}$$

derivative w.r.t to bias $\mathbf{b}_h 1$:

$$\frac{\partial MSE}{\partial \mathbf{b}_h 1} = \frac{\partial MSE}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial \mathbf{b}_h 1} = \frac{\partial MSE}{\partial \mathbf{h}_1} \cdot \mathbf{h}_1 \cdot (1 - \mathbf{h}_1)$$

4 Evaluations (10 Points)

In class, we discussed solving a linear regression using the pseudo-inverse, which gives the same result as minimizing the residual sum of squares. In the starter notebook, I have provided a solution to the linear regression problem using SVD and the pseudo-inverse. Compare the root mean square error (RMSE) of this method, the ridge regression method, and the one-layer MLP. Why do you think the errors are different ? Which one has the lowest error and why ?

Answer:

The RMSE of the linear regression using the pseudo-inverse is **7.387891796775459**. The RMSE of the ridge regression is **7.379693013437837**. The RMSE of the one-layer MLP is **13.113716195628152**.

The errors are different because the methods used to solve the regression problem are different. The linear regression using the pseudo-inverse to solve the regression problem, and the ridge regression method uses L2 regularization to add a penalty to the loss, which helps to reduce overfitting, while the one-layer MLP is a neural network method which involves calculating weights. The one-layer MLP is a more complex model that can capture non-linear relationships in the data. The one-layer MLP has the highest error because it is a more complex model that can capture non-linear relationships in the data which cannot be captured in a one-layer mlp. The ridge regression method has the lowest error because it adds a regularization term to the loss function, which helps prevent overfitting.

I have provided a keras implementation of a deeper neural network with 4 layers and it ouputs the lowers RMSE error. However, there are some techniques added here like dropout, weight initialization, different activation function, and Adam optimizer (we used SGD optimizer in previous question). If you are interested in knowing more about deep neural networks, then you should consider taking the course introduction to deep learning offered here at CMU by Prof Bhiksha.

Homework4

November 11, 2024

[2]:
```
import numpy as np
import pandas as pd


from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

import warnings
warnings.filterwarnings('ignore')
```

[3]:
```
df = pd.read_csv("Real estate.csv")
df.rename({'Ouse_price_of_unit_area': 'House_price_of_unit_area'}, axis = 1,␣
 ↪inplace = True)
df.drop("No", axis = 1, inplace = True)
column_maping = {}
for i in df.columns:
    new_column = i[3:].capitalize().replace(' ', '_')
    column_maping[i] = new_column
# Now we will rename the column using the dictinary
df.rename(columns = column_maping, inplace = True)
df.rename({'Ouse_price_of_unit_area': 'House_price_of_unit_area'}, axis = 1,␣
 ↪inplace = True)
X  = df.drop(['Transaction_date', "House_price_of_unit_area"], axis = 1)
y = df['House_price_of_unit_area']
```

[4]:
```
# Do not change the code below
scaler = StandardScaler()
X_scale = scaler.fit_transform(X)
X_scale = np.asarray(X_scale)
y = np.asarray(y)
X_train, X_test, y_train, y_test = train_test_split(X_scale, y, test_size = 0.
 ↪2, random_state = 42)
# Do not change the code Above

# Add a column of ones to X_train and X_test to account for the bias term␣
 ↪(intercept)
X_train_b = np.c_[np.ones((X_train.shape[0], 1)), X_train]
```

```
X_test_b = np.c_[np.ones((X_test.shape[0], 1)), X_test]

# Perform SVD decomposition on the training data
U, s, VT = np.linalg.svd(X_train_b, full_matrices=False)

# Create diagonal matrix for Sigma
S_diag = np.diag(s)

# Compute the pseudo-inverse of the training data
X_train_pinv = VT.T @ np.linalg.inv(S_diag) @ U.T

# Calculate the weights (regression coefficients), including the bias term␣
 ↪(intercept)
w = X_train_pinv @ y_train

# Make predictions using the testing set
y_pred = X_test_b @ w

se = (y_pred-y_test) ** 2
mse = se.mean()
rmse = mse**0.5
print(f"Root Mean Squre Error using SVD {rmse}")
```

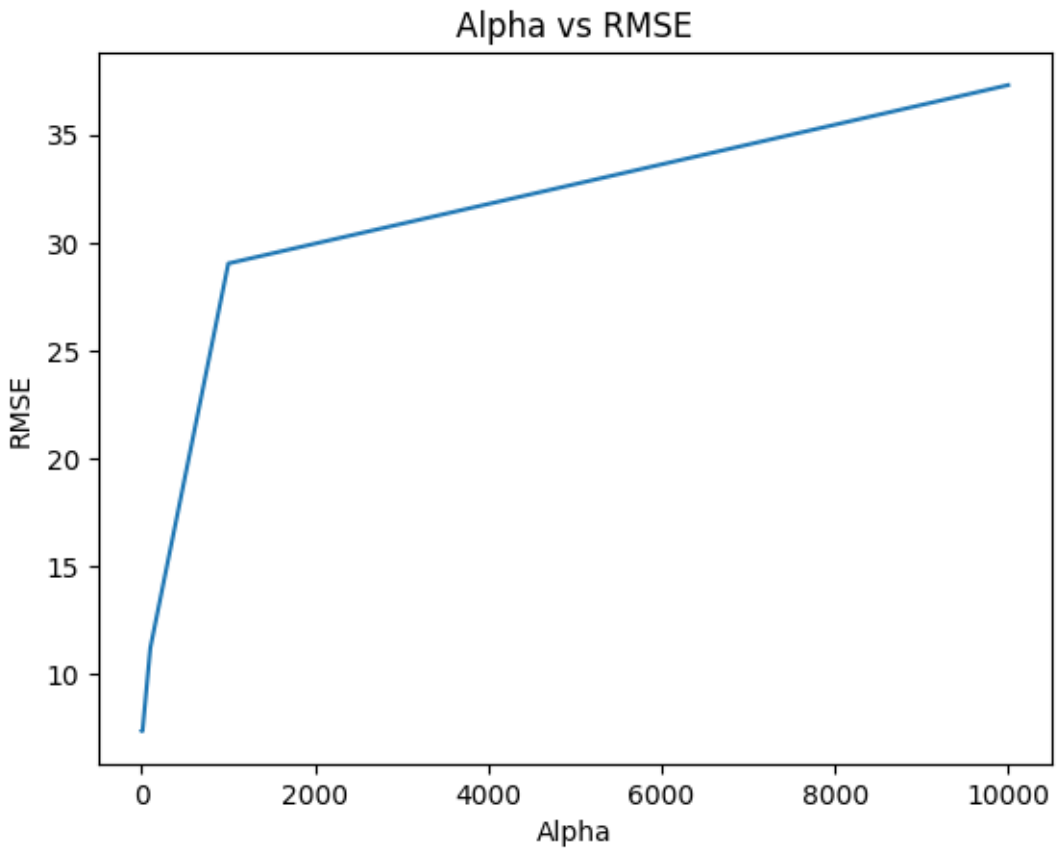
```
Root Mean Squre Error using SVD 7.387891796775459
```

[11]:
```
import matplotlib.pyplot as plt
class Model(object):
    """
     Ridge Regression.
    """

    def fit(self, X, y, alpha=0):
        """
        Fits the ridge regression model to the training data.

        Arguments
        ----------
        X: nxp matrix of n examples with p independent variables
        y: response variable vector for n examples
        alpha: regularization parameter.
        """

        intercept = np.ones((len(X),1))
        X_b = np.c_[intercept,X]

        I = np.identity(X_b.shape[1])
```

```
        betha_optim = np.linalg.inv(X_b.T.dot(X_b) + alpha*I).dot(X_b.T).dot(y)
        self.betas = betha_optim
        return betha_optim

    def predict(self, X):
        """
        Predicts the dependent variable of new data using the model.

        Arguments
        ----------
        X: nxp matrix of n examples with p covariates

        Returns
        ----------
        response variable vector for n examples
        """
               # Your code here
        X_predictor = np.c_[np.ones((X.shape[0], 1)), X]
        self.predictions = X_predictor.dot(self.betas)
        return self.predictions

    def rmse(self, X, y):
        """
        Returns the RMSE(Root Mean Squared Error) when the model is validated.

        Arguments
        ----------
        X: nxp matrix of n examples with p covariates
        y: response variable vector for n examples

        Returns
        ----------
        RMSE when model is used to predict y
        """
        y_predict = self.predict(X=X)
        se = (y_predict-y) ** 2
        mse = se.mean()
        rmse = mse**0.5
        return rmse

# Do not change the code below
scaler = StandardScaler()
X_scale = scaler.fit_transform(X)
X_scale = np.asarray(X_scale)
y = np.asarray(y)
X_train, X_test, y_train, y_test = train_test_split(X_scale, y, test_size = 0.
 ↪2, random_state = 42)
```

```python
my_model = Model()
#! TODO: USE DIFFERENT SETS OF ALPHA VALUES TO FIND THE BEST ALPHA
alphas = [0.01, 0.1, 1, 10, 100, 1000, 10000]
RMSES = []
for alpha in alphas:
    my_model.fit(X=X_train, y=y_train, alpha=alpha)
    rmse = my_model.rmse(X=X_test, y=y_test)
    RMSES.append(rmse)
    print(f"Root Mean Squre Error Ridge for alpha: {alpha} is {rmse}")
    print(my_model.rmse(X=X_test, y=y_test))

# plot different alpha values against RMSE
plt.plot(alphas, RMSES)
plt.xlabel('Alpha')
plt.ylabel('RMSE')
plt.title('Alpha vs RMSE')
plt.show()
```

```
Root Mean Squre Error Ridge for alpha: 0.01 is 7.387798950430715
7.387798950430715
Root Mean Squre Error Ridge for alpha: 0.1 is 7.38697328534338
7.38697328534338
Root Mean Squre Error Ridge for alpha: 1 is 7.379693013437837
7.379693013437837
Root Mean Squre Error Ridge for alpha: 10 is 7.39620540310947
7.39620540310947
Root Mean Squre Error Ridge for alpha: 100 is 11.238394958870039
11.238394958870039
Root Mean Squre Error Ridge for alpha: 1000 is 29.033427420684916
29.033427420684916
Root Mean Squre Error Ridge for alpha: 10000 is 37.290048013604675
37.290048013604675
```



[6]:
```python
import numpy as np

# Sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Derivative of sigmoid
def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))

# Mean Squared Error loss
def mse_loss(y_true, y_pred):
    return ((y_true - y_pred) ** 2).mean()

def mse_loss_derivative(y_true, y_pred):
    return -2 * (y_true - y_pred)/ y_true.size

# Forward pass
def forward_pass(x, W_h1, b_h1, W_o, b_o):
```

```
    #TODO what is the output of the hidden layer
    a_h1 =  np.dot(x, W_h1) + b_h1
    #TODO apply activation to the output of the hidden layer
    z_h1 = sigmoid(a_h1)
    #TODO use ouput of activation as input to the output layer (it is just␣
 ↪similar to first layer but we don't apply activation)
    y_pred = np.dot(z_h1, W_o) + b_o
    return y_pred, z_h1, a_h1

# Backward pass
def backward_pass(x, y_true, y_pred, z_h1, a_h1, W_h1, W_o):
    # Derivative of loss with respect to y_pred
    #TODO What type of loss function are we using, what is it's derivative ?
    dL_dy_pred = mse_loss_derivative(y_true, y_pred)


    # Gradients for output layer
    #TODO remember the output layer is just a dense layer
    dL_dW_o = np.dot(z_h1.T, dL_dy_pred)

    dL_db_o = np.sum(dL_dy_pred, axis=0, keepdims=True)


    # Derivative of loss with respect to z_h1
    #TODO derivative with respect to the output of activation layer

    dL_dz_h1 = np.dot(dL_dy_pred, W_o.T)


    # Derivative of loss with respect to a_h1
    #TODO derivative with respect to the activation layer
    dL_da_h1 = dL_dz_h1 * sigmoid_derivative(a_h1)

    # Gradients for hidden layer
    dL_dW_h1 = np.dot(x.T, dL_da_h1)
    dL_db_h1 = np.sum(dL_da_h1, axis=0, keepdims=True)


    return dL_dW_h1, dL_db_h1, dL_dW_o, dL_db_o



scaler = StandardScaler()
X_scale = scaler.fit_transform(X)
X_scale = np.asarray(X_scale)
y = np.asarray(y)
```

```
X_train, X_test, y_train, y_test = train_test_split(X_scale, y, test_size = 0.
 ↪2, random_state = 42)


y_train = y_train.reshape((y_train.shape[0],1))


# Network architecture
input_size = 5 # Number of features
hidden_layer_size = 20 # Number of neurons in layer
output_size = 1 # predicted variable

# Initial random weights and biases for each layer
W_h1 = np.random.randn(input_size, hidden_layer_size) * 0.001
b_h1 = np.zeros((1, hidden_layer_size))
W_o = np.random.randn(hidden_layer_size, output_size) * 0.001
b_o = np.zeros((1, output_size))

learning_rate = 0.2

#To save the weights which give the lowest loss
lowest_loss = float('inf')
best_weights = None


for i in range(100):
    # Forward pass to get predictions
    y_pred, z_h1, a_h1 = forward_pass(X_train, W_h1, b_h1, W_o, b_o)
    loss = mse_loss(y_train, y_pred)

    if loss < lowest_loss:
        lowest_loss = loss
        # Save the best weights and biases
        best_weights = (W_h1.copy(), b_h1.copy(), W_o.copy(), b_o.copy())

    # Backward pass to get gradients

    dL_dW_h1, dL_db_h1, dL_dW_o, dL_db_o = backward_pass(X_train, y_train,␣
 ↪y_pred, z_h1, a_h1, W_h1, W_o)

    # Now you would use the gradients to update the weights and biases
    W_h1 -= learning_rate * dL_dW_h1
    b_h1 -= learning_rate * dL_db_h1
    W_o -= learning_rate * dL_dW_o
    b_o -= learning_rate * dL_db_o
```

```
W_h1_best, b_h1_best, W_o_best, b_o_best = best_weights
y_pred_test, _, _ = forward_pass(X_test, W_h1_best, b_h1_best, W_o_best,␣
 ↪b_o_best)
se = (y_pred_test-y_test) ** 2
mse = se.mean()
rmse = mse**0.5
print(f"Root Mean Squre Error 1 Layer MLP {rmse}")

#(331, 1)
#Root Mean Squre Error 1 Layer MLP 15.69657819926082
```

```
Root Mean Squre Error 1 Layer MLP 13.113716211105924
```

Bonus Lets add one more hidden layer (10 Points)

0.0.1 You must write down all gradients and complete the code below to get full bonus points

[7]:
```
# Forward pass
def forward_pass(x, W_h1, b_h1, W_h2, b_h2, W_o, b_o):
    #TODO compute ouput of first hidden layer
    a_h1 = np.dot(x, W_h1) + b_h1
    z_h1 = sigmoid(a_h1) # apply activation to ouputs of first hidden layer

    #TODO compute ouput of second hidden layer
    a_h2 = np.dot(z_h1, W_h2) + b_h2
    z_h2 = sigmoid(a_h2) #apply activation to ouputs of first hidden layer

    # compute ouput of output layer, why don't we apply activation ?
    y_pred = np.dot(z_h2, W_o) + b_o

    return y_pred, z_h1, a_h1, z_h2, a_h2

# Backward pass
def backward_pass(x, y_true, y_pred, z_h1, a_h1, z_h2, a_h2, W_h1, W_h2, W_o):

    # Derivative of loss with respect to y_pred, what kind of loss are we using␣
 ↪?
    #TODO
    dL_dy_pred = mse_loss_derivative(y_true, y_pred)

    # Gradients for output layer
    #TODO
    dL_dW_o = np.dot(z_h2.T, dL_dy_pred)
    #TODO
    dL_db_o = np.sum(dL_dy_pred, axis=0, keepdims=True)
```

```
    # Derivative of loss with respect to z_h2
    #TODO
    dL_dz_h2 = np.dot(dL_dy_pred, W_o.T)

    # Derivative of loss with respect to a_h2
    #TODO
    dL_da_h2 =  dL_dz_h2 * sigmoid_derivative(a_h2)

    # Gradients for second hidden layer
    #TODO
    dL_dW_h2 =  np.dot(z_h1.T, dL_da_h2)
    #TODO
    dL_db_h2 =  np.sum(dL_da_h2, axis=0, keepdims=True)

    # Derivative of loss with respect to z_h1
    #TODO
    dL_dz_h1 =  np.dot(dL_da_h2, W_h2.T)

    # Derivative of loss with respect to a_h1
    #TODO
    dL_da_h1 =  dL_dz_h1 * sigmoid_derivative(a_h1)

    # Gradients for first hidden layer
    #TODO
    dL_dW_h1 =  np.dot(x.T, dL_da_h1)
    #TODO
    dL_db_h1 =  np.sum(dL_da_h1, axis=0, keepdims=True)

    return dL_dW_h1, dL_db_h1, dL_dW_h2, dL_db_h2, dL_dW_o, dL_db_o



# Random input and true output (modify these according to your dataset)
scaler = StandardScaler()
X_scale = scaler.fit_transform(X)
X_scale = np.asarray(X_scale)
y = np.asarray(y)
X_train, X_test, y_train, y_test = train_test_split(X_scale, y, test_size = 0.
  ↪2, random_state = 42)
y_train = y_train.reshape((y_train.shape[0],1))

# Network architecture
input_size = 5 # Number of features
hidden_layer1_size = 100
hidden_layer2_size = 20
output_size = 1
```

```
# Initial random weights and biases for each layer
W_h1 = np.random.randn(input_size, hidden_layer1_size) * 0.001
b_h1 = np.zeros((1, hidden_layer1_size))
W_h2 = np.random.randn(hidden_layer1_size, hidden_layer2_size) * 0.001
b_h2 = np.zeros((1, hidden_layer2_size))
W_o = np.random.randn(hidden_layer2_size, output_size) * 0.001
b_o = np.zeros((1, output_size))


learning_rate = 0.1

#Training loop
for i in range(200):
    # Forward pass to get predictions
    y_pred, z_h1, a_h1, z_h2, a_h2 = forward_pass(X_train, W_h1, b_h1, W_h2,␣
 ↪b_h2, W_o, b_o)
    #TODO Compute the loss
    # Backward pass to get gradients
    dL_dW_h1, dL_db_h1, dL_dW_h2, dL_db_h2, dL_dW_o, dL_db_o =␣
 ↪backward_pass(X_train, y_train, y_pred, z_h1, a_h1, z_h2, a_h2, W_h1, W_h2,␣
 ↪W_o)

    # Now you would use the gradients to update the weights and biases for each␣
 ↪layer
    #TODO
    W_h1 -= learning_rate * dL_dW_h1
    #TODO
    b_h1 -= learning_rate * dL_db_h1

    #TODO
    W_h2 -= learning_rate * dL_dW_h2
    #TODO
    b_h2 -= learning_rate * dL_db_h2
    #TODO
    W_o -= learning_rate * dL_dW_o
    #TODO
    b_o -= learning_rate * dL_db_o

y_pred, _, _, _, _ = forward_pass(X_test, W_h1, b_h1, W_h2, b_h2, W_o, b_o)
# print(y_pred)
se = (y_pred-y_test) ** 2
mse = se.mean()
rmse = mse**0.5
print(f"Root Mean Squre Error 2 Layer MLP {rmse}")
```

```
Root Mean Squre Error 2 Layer MLP 13.1137162111749
```

[8]:
```
%pip install keras
%pip install tensorflow

from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.callbacks import EarlyStopping


model = Sequential()


model.add(Dense(400, input_dim = 5, kernel_initializer = 'he_uniform', ␣
 ↪activation = 'relu')) #
model.add(Dropout(0.2))


model.add(Dense(400, input_dim = 5, kernel_initializer = 'he_uniform',␣
 ↪activation = 'relu')) #
model.add(Dropout(0.2))

model.add(Dense(400, kernel_initializer = 'he_uniform',activation = 'relu')) #
model.add(Dropout(0.2))

model.add(Dense(1, activation = 'linear'))

model.compile(loss = 'mean_squared_error', optimizer = 'adam')
model.summary()
```

```
Requirement already satisfied: keras in /usr/local/lib/python3.10/dist-packages
(3.4.1)
Requirement already satisfied: absl-py in /usr/local/lib/python3.10/dist-
packages (from keras) (1.4.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages
(from keras) (1.26.4)
Requirement already satisfied: rich in /usr/local/lib/python3.10/dist-packages
(from keras) (13.9.4)
Requirement already satisfied: namex in /usr/local/lib/python3.10/dist-packages
(from keras) (0.0.8)
Requirement already satisfied: h5py in /usr/local/lib/python3.10/dist-packages
(from keras) (3.12.1)
Requirement already satisfied: optree in /usr/local/lib/python3.10/dist-packages
(from keras) (0.13.0)
Requirement already satisfied: ml-dtypes in /usr/local/lib/python3.10/dist-
packages (from keras) (0.4.1)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-
packages (from keras) (24.1)
Requirement already satisfied: typing-extensions>=4.5.0 in
/usr/local/lib/python3.10/dist-packages (from optree->keras) (4.12.2)
```

```
Requirement already satisfied: markdown-it-py>=2.2.0 in
/usr/local/lib/python3.10/dist-packages (from rich->keras) (3.0.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in
/usr/local/lib/python3.10/dist-packages (from rich->keras) (2.18.0)
Requirement already satisfied: mdurl~=0.1 in /usr/local/lib/python3.10/dist-
packages (from markdown-it-py>=2.2.0->rich->keras) (0.1.2)
Requirement already satisfied: tensorflow in /usr/local/lib/python3.10/dist-
packages (2.17.0)
Requirement already satisfied: absl-py>=1.0.0 in /usr/local/lib/python3.10/dist-
packages (from tensorflow) (1.4.0)
Requirement already satisfied: astunparse>=1.6.0 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (1.6.3)
Requirement already satisfied: flatbuffers>=24.3.25 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (24.3.25)
Requirement already satisfied: gast!=0.5.0,!=0.5.1,!=0.5.2,>=0.2.1 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (0.6.0)
Requirement already satisfied: google-pasta>=0.1.1 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (0.2.0)
Requirement already satisfied: h5py>=3.10.0 in /usr/local/lib/python3.10/dist-
packages (from tensorflow) (3.12.1)
Requirement already satisfied: libclang>=13.0.0 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (18.1.1)
Requirement already satisfied: ml-dtypes<0.5.0,>=0.3.1 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (0.4.1)
Requirement already satisfied: opt-einsum>=2.3.2 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (3.4.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-
packages (from tensorflow) (24.1)
Requirement already satisfied:
protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<5.0.0dev,>=3.20.3
in /usr/local/lib/python3.10/dist-packages (from tensorflow) (3.20.3)
Requirement already satisfied: requests<3,>=2.21.0 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (2.32.3)
Requirement already satisfied: setuptools in /usr/local/lib/python3.10/dist-
packages (from tensorflow) (75.1.0)
Requirement already satisfied: six>=1.12.0 in /usr/local/lib/python3.10/dist-
packages (from tensorflow) (1.16.0)
Requirement already satisfied: termcolor>=1.1.0 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (2.5.0)
Requirement already satisfied: typing-extensions>=3.6.6 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (4.12.2)
Requirement already satisfied: wrapt>=1.11.0 in /usr/local/lib/python3.10/dist-
packages (from tensorflow) (1.16.0)
Requirement already satisfied: grpcio<2.0,>=1.24.3 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (1.64.1)
Requirement already satisfied: tensorboard<2.18,>=2.17 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (2.17.0)
Requirement already satisfied: keras>=3.2.0 in /usr/local/lib/python3.10/dist-
```

```
packages (from tensorflow) (3.4.1)
Requirement already satisfied: tensorflow-io-gcs-filesystem>=0.23.1 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (0.37.1)
Requirement already satisfied: numpy<2.0.0,>=1.23.5 in
/usr/local/lib/python3.10/dist-packages (from tensorflow) (1.26.4)
Requirement already satisfied: wheel<1.0,>=0.23.0 in
/usr/local/lib/python3.10/dist-packages (from astunparse>=1.6.0->tensorflow)
(0.44.0)
Requirement already satisfied: rich in /usr/local/lib/python3.10/dist-packages
(from keras>=3.2.0->tensorflow) (13.9.4)
Requirement already satisfied: namex in /usr/local/lib/python3.10/dist-packages
(from keras>=3.2.0->tensorflow) (0.0.8)
Requirement already satisfied: optree in /usr/local/lib/python3.10/dist-packages
(from keras>=3.2.0->tensorflow) (0.13.0)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.10/dist-packages (from requests<3,>=2.21.0->tensorflow)
(3.4.0)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-
packages (from requests<3,>=2.21.0->tensorflow) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.10/dist-packages (from requests<3,>=2.21.0->tensorflow)
(2.2.3)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.10/dist-packages (from requests<3,>=2.21.0->tensorflow)
(2024.8.30)
Requirement already satisfied: markdown>=2.6.8 in
/usr/local/lib/python3.10/dist-packages (from
tensorboard<2.18,>=2.17->tensorflow) (3.7)
Requirement already satisfied: tensorboard-data-server<0.8.0,>=0.7.0 in
/usr/local/lib/python3.10/dist-packages (from
tensorboard<2.18,>=2.17->tensorflow) (0.7.2)
Requirement already satisfied: werkzeug>=1.0.1 in
/usr/local/lib/python3.10/dist-packages (from
tensorboard<2.18,>=2.17->tensorflow) (3.1.2)
Requirement already satisfied: MarkupSafe>=2.1.1 in
/usr/local/lib/python3.10/dist-packages (from
werkzeug>=1.0.1->tensorboard<2.18,>=2.17->tensorflow) (3.0.2)
Requirement already satisfied: markdown-it-py>=2.2.0 in
/usr/local/lib/python3.10/dist-packages (from rich->keras>=3.2.0->tensorflow)
(3.0.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in
/usr/local/lib/python3.10/dist-packages (from rich->keras>=3.2.0->tensorflow)
(2.18.0)
Requirement already satisfied: mdurl~=0.1 in /usr/local/lib/python3.10/dist-
packages (from markdown-it-py>=2.2.0->rich->keras>=3.2.0->tensorflow) (0.1.2)
```

Model: "sequential"

```
Layer (type)                         Output Shape                ␣
↪Param #

dense (Dense)                        (None, 400)                   ␣
↪2,400

dropout (Dropout)                    (None, 400)                      ␣
↪  0

dense_1 (Dense)                      (None, 400)                 ␣
↪160,400

dropout_1 (Dropout)                  (None, 400)                      ␣
↪  0

dense_2 (Dense)                      (None, 400)                 ␣
↪160,400

dropout_2 (Dropout)                  (None, 400)                      ␣
↪  0

dense_3 (Dense)                      (None, 1)                        ␣
↪401


Total params: 323,601 (1.23 MB)

Trainable params: 323,601 (1.23 MB)

Non-trainable params: 0 (0.00 B)
```

[9]:
```
import os
os.environ["CUDA_VISIBLE_DEVICES"] = "-1"
scaler = StandardScaler()
X_scale = scaler.fit_transform(X)
X_scale = np.asarray(X_scale)
y = np.asarray(y)
X_train, X_test, y_train, y_test = train_test_split(X_scale, y, test_size = 0.
 ↪2, random_state = 42)
y_train = y_train.reshape((y_train.shape[0],1))
history = model.fit(X_train, y_train,  epochs = 200, validation_data=(X_test,␣
 ↪y_test),
```

```
                    callbacks = EarlyStopping(monitor = 'val_loss',patience =␣
↪40))
```

```
Epoch 1/200
11/11             4s 83ms/step -
loss: 986.4611 - val_loss: 187.3347
Epoch 2/200
11/11             0s 34ms/step -
loss: 179.3348 - val_loss: 156.9577
Epoch 3/200
11/11             1s 26ms/step -
loss: 128.8178 - val_loss: 97.9532
Epoch 4/200
11/11             1s 20ms/step -
loss: 113.4353 - val_loss: 100.7919
Epoch 5/200
11/11             0s 24ms/step -
loss: 98.3579 - val_loss: 89.4237
Epoch 6/200
11/11             1s 20ms/step -
loss: 139.2811 - val_loss: 83.5790
Epoch 7/200
11/11             0s 22ms/step -
loss: 95.1077 - val_loss: 73.6563
Epoch 8/200
11/11             0s 23ms/step -
loss: 102.2080 - val_loss: 79.5526
Epoch 9/200
11/11             0s 19ms/step -
loss: 88.3762 - val_loss: 65.0579
Epoch 10/200
11/11             0s 21ms/step -
loss: 73.7353 - val_loss: 61.5901
Epoch 11/200
11/11             0s 16ms/step -
loss: 102.1125 - val_loss: 59.2914
Epoch 12/200
11/11             0s 30ms/step -
loss: 91.2132 - val_loss: 64.8738
Epoch 13/200
11/11             1s 24ms/step -
loss: 97.1009 - val_loss: 59.5561
Epoch 14/200
11/11             0s 21ms/step -
loss: 93.7803 - val_loss: 59.5480
Epoch 15/200
11/11             0s 17ms/step -
loss: 93.1076 - val_loss: 58.1397
```

```
Epoch 16/200
11/11             0s 18ms/step -
loss: 88.8036 - val_loss: 54.3965
Epoch 17/200
11/11             0s 17ms/step -
loss: 70.2638 - val_loss: 49.3561
Epoch 18/200
11/11             1s 38ms/step -
loss: 84.1058 - val_loss: 50.6575
Epoch 19/200
11/11             1s 33ms/step -
loss: 70.0297 - val_loss: 54.6827
Epoch 20/200
11/11             1s 21ms/step -
loss: 88.8656 - val_loss: 55.3178
Epoch 21/200
11/11             1s 28ms/step -
loss: 67.8646 - val_loss: 51.2793
Epoch 22/200
11/11             0s 18ms/step -
loss: 76.7303 - val_loss: 61.0252
Epoch 23/200
11/11             0s 23ms/step -
loss: 89.9349 - val_loss: 53.9564
Epoch 24/200
11/11             0s 18ms/step -
loss: 77.2778 - val_loss: 43.7928
Epoch 25/200
11/11             0s 19ms/step -
loss: 87.9341 - val_loss: 51.7264
Epoch 26/200
11/11             1s 23ms/step -
loss: 73.4521 - val_loss: 50.7750
Epoch 27/200
11/11             1s 34ms/step -
loss: 85.6545 - val_loss: 51.1736
Epoch 28/200
11/11             1s 31ms/step -
loss: 89.2620 - val_loss: 68.4901
Epoch 29/200
11/11             1s 37ms/step -
loss: 74.7465 - val_loss: 51.0189
Epoch 30/200
11/11             0s 25ms/step -
loss: 68.6750 - val_loss: 44.9507
Epoch 31/200
11/11             0s 16ms/step -
loss: 71.4699 - val_loss: 50.2313
```

```
Epoch 32/200
11/11             0s 20ms/step -
loss: 88.5188 - val_loss: 44.1377
Epoch 33/200
11/11             0s 18ms/step -
loss: 67.7506 - val_loss: 46.9963
Epoch 34/200
11/11             0s 16ms/step -
loss: 72.2658 - val_loss: 45.3241
Epoch 35/200
11/11             0s 18ms/step -
loss: 89.9286 - val_loss: 44.4716
Epoch 36/200
11/11             0s 17ms/step -
loss: 80.5026 - val_loss: 51.0675
Epoch 37/200
11/11             0s 18ms/step -
loss: 63.0110 - val_loss: 42.5683
Epoch 38/200
11/11             0s 17ms/step -
loss: 57.0131 - val_loss: 41.4556
Epoch 39/200
11/11             0s 12ms/step -
loss: 81.6651 - val_loss: 41.3081
Epoch 40/200
11/11             0s 14ms/step -
loss: 74.9821 - val_loss: 45.5565
Epoch 41/200
11/11             0s 10ms/step -
loss: 73.2874 - val_loss: 61.6684
Epoch 42/200
11/11             0s 11ms/step -
loss: 64.2432 - val_loss: 44.2818
Epoch 43/200
11/11             0s 10ms/step -
loss: 90.6332 - val_loss: 42.1117
Epoch 44/200
11/11             0s 11ms/step -
loss: 70.9702 - val_loss: 50.3262
Epoch 45/200
11/11             0s 11ms/step -
loss: 79.0764 - val_loss: 57.3779
Epoch 46/200
11/11             0s 11ms/step -
loss: 97.1907 - val_loss: 43.1374
Epoch 47/200
11/11             0s 12ms/step -
loss: 104.1230 - val_loss: 41.1444
```

```
Epoch 48/200
11/11             0s 12ms/step -
loss: 72.1089 - val_loss: 39.1258
Epoch 49/200
11/11             0s 12ms/step -
loss: 76.6023 - val_loss: 67.6951
Epoch 50/200
11/11             0s 13ms/step -
loss: 75.2273 - val_loss: 38.8589
Epoch 51/200
11/11             0s 10ms/step -
loss: 95.9648 - val_loss: 42.3573
Epoch 52/200
11/11             0s 11ms/step -
loss: 61.2024 - val_loss: 44.3708
Epoch 53/200
11/11             0s 11ms/step -
loss: 77.4434 - val_loss: 48.9275
Epoch 54/200
11/11             0s 10ms/step -
loss: 72.5472 - val_loss: 47.1935
Epoch 55/200
11/11             0s 13ms/step -
loss: 71.6391 - val_loss: 50.4937
Epoch 56/200
11/11             0s 13ms/step -
loss: 72.6620 - val_loss: 48.8558
Epoch 57/200
11/11             0s 10ms/step -
loss: 78.5944 - val_loss: 55.0177
Epoch 58/200
11/11             0s 12ms/step -
loss: 68.6633 - val_loss: 42.5117
Epoch 59/200
11/11             0s 12ms/step -
loss: 56.7422 - val_loss: 37.6094
Epoch 60/200
11/11             0s 10ms/step -
loss: 58.6207 - val_loss: 38.5807
Epoch 61/200
11/11             0s 11ms/step -
loss: 97.7858 - val_loss: 56.3075
Epoch 62/200
11/11             0s 11ms/step -
loss: 97.2284 - val_loss: 48.2464
Epoch 63/200
11/11             0s 10ms/step -
loss: 59.8639 - val_loss: 39.2761
```

```
Epoch 64/200
11/11             0s 12ms/step -
loss: 67.8466 - val_loss: 50.1880
Epoch 65/200
11/11             0s 10ms/step -
loss: 71.4863 - val_loss: 40.2775
Epoch 66/200
11/11             0s 10ms/step -
loss: 83.0523 - val_loss: 45.2619
Epoch 67/200
11/11             0s 12ms/step -
loss: 58.9747 - val_loss: 49.9192
Epoch 68/200
11/11             0s 10ms/step -
loss: 65.9682 - val_loss: 39.6778
Epoch 69/200
11/11             0s 12ms/step -
loss: 99.1024 - val_loss: 40.7259
Epoch 70/200
11/11             0s 12ms/step -
loss: 69.9960 - val_loss: 47.4773
Epoch 71/200
11/11             0s 13ms/step -
loss: 74.9860 - val_loss: 51.0106
Epoch 72/200
11/11             0s 10ms/step -
loss: 55.5364 - val_loss: 37.8628
Epoch 73/200
11/11             0s 12ms/step -
loss: 57.7738 - val_loss: 41.2249
Epoch 74/200
11/11             0s 11ms/step -
loss: 55.6683 - val_loss: 38.5737
Epoch 75/200
11/11             0s 11ms/step -
loss: 85.8466 - val_loss: 73.6150
Epoch 76/200
11/11             0s 13ms/step -
loss: 101.9437 - val_loss: 47.3916
Epoch 77/200
11/11             0s 12ms/step -
loss: 59.1887 - val_loss: 42.2810
Epoch 78/200
11/11             0s 10ms/step -
loss: 79.0101 - val_loss: 42.5591
Epoch 79/200
11/11             0s 11ms/step -
loss: 60.5573 - val_loss: 39.3539
```

```
Epoch 80/200
11/11             0s 12ms/step -
loss: 85.0743 - val_loss: 47.3171
Epoch 81/200
11/11             0s 12ms/step -
loss: 52.8557 - val_loss: 41.1603
Epoch 82/200
11/11             0s 10ms/step -
loss: 74.5153 - val_loss: 36.3660
Epoch 83/200
11/11             0s 10ms/step -
loss: 76.8667 - val_loss: 38.8608
Epoch 84/200
11/11             0s 11ms/step -
loss: 85.8755 - val_loss: 39.9939
Epoch 85/200
11/11             0s 10ms/step -
loss: 59.3341 - val_loss: 39.6148
Epoch 86/200
11/11             0s 11ms/step -
loss: 77.3023 - val_loss: 42.0261
Epoch 87/200
11/11             0s 14ms/step -
loss: 56.8682 - val_loss: 38.1092
Epoch 88/200
11/11             0s 10ms/step -
loss: 55.7162 - val_loss: 41.7898
Epoch 89/200
11/11             0s 10ms/step -
loss: 62.8504 - val_loss: 42.8936
Epoch 90/200
11/11             0s 11ms/step -
loss: 54.4851 - val_loss: 41.7834
Epoch 91/200
11/11             0s 14ms/step -
loss: 56.8465 - val_loss: 39.0139
Epoch 92/200
11/11             0s 17ms/step -
loss: 65.4451 - val_loss: 38.8984
Epoch 93/200
11/11             0s 29ms/step -
loss: 60.2444 - val_loss: 41.7889
Epoch 94/200
11/11             1s 27ms/step -
loss: 79.2747 - val_loss: 46.0970
Epoch 95/200
11/11             1s 24ms/step -
loss: 56.7410 - val_loss: 41.3795
```

```
Epoch 96/200
11/11             0s 16ms/step -
loss: 55.5673 - val_loss: 37.1161
Epoch 97/200
11/11             0s 19ms/step -
loss: 66.8238 - val_loss: 40.3603
Epoch 98/200
11/11             0s 17ms/step -
loss: 88.2185 - val_loss: 54.7229
Epoch 99/200
11/11             0s 19ms/step -
loss: 73.9865 - val_loss: 66.6440
Epoch 100/200
11/11             0s 20ms/step -
loss: 88.4238 - val_loss: 44.2323
Epoch 101/200
11/11             0s 11ms/step -
loss: 61.8064 - val_loss: 41.4330
Epoch 102/200
11/11             0s 12ms/step -
loss: 59.4315 - val_loss: 42.0228
Epoch 103/200
11/11             0s 10ms/step -
loss: 82.1133 - val_loss: 45.4302
Epoch 104/200
11/11             0s 12ms/step -
loss: 55.7618 - val_loss: 36.3542
Epoch 105/200
11/11             0s 11ms/step -
loss: 49.7134 - val_loss: 42.4889
Epoch 106/200
11/11             0s 10ms/step -
loss: 59.3687 - val_loss: 45.5686
Epoch 107/200
11/11             0s 12ms/step -
loss: 78.7495 - val_loss: 42.4134
Epoch 108/200
11/11             0s 15ms/step -
loss: 50.0286 - val_loss: 49.7534
Epoch 109/200
11/11             0s 12ms/step -
loss: 67.2207 - val_loss: 43.6824
Epoch 110/200
11/11             0s 11ms/step -
loss: 69.1416 - val_loss: 37.8418
Epoch 111/200
11/11             0s 13ms/step -
loss: 79.1478 - val_loss: 38.2905
```

```
Epoch 112/200
11/11             0s 23ms/step -
loss: 74.2002 - val_loss: 41.7023
Epoch 113/200
11/11             1s 28ms/step -
loss: 57.7071 - val_loss: 43.4487
Epoch 114/200
11/11             1s 19ms/step -
loss: 49.6566 - val_loss: 37.9704
Epoch 115/200
11/11             0s 12ms/step -
loss: 55.2487 - val_loss: 38.8944
Epoch 116/200
11/11             0s 10ms/step -
loss: 65.2462 - val_loss: 43.7662
Epoch 117/200
11/11             0s 19ms/step -
loss: 58.9636 - val_loss: 41.4765
Epoch 118/200
11/11             0s 21ms/step -
loss: 53.6188 - val_loss: 41.2831
Epoch 119/200
11/11             1s 19ms/step -
loss: 68.3268 - val_loss: 38.9324
Epoch 120/200
11/11             0s 29ms/step -
loss: 65.8823 - val_loss: 39.5982
Epoch 121/200
11/11             0s 20ms/step -
loss: 89.0370 - val_loss: 41.0503
Epoch 122/200
11/11             0s 22ms/step -
loss: 51.6565 - val_loss: 41.2813
Epoch 123/200
11/11             0s 22ms/step -
loss: 50.5487 - val_loss: 43.7615
Epoch 124/200
11/11             0s 25ms/step -
loss: 53.0840 - val_loss: 41.4637
Epoch 125/200
11/11             1s 31ms/step -
loss: 47.4496 - val_loss: 36.3940
Epoch 126/200
11/11             0s 18ms/step -
loss: 94.7648 - val_loss: 55.7338
Epoch 127/200
11/11             0s 19ms/step -
loss: 69.1148 - val_loss: 65.4991
```

```
Epoch 128/200
11/11             0s 17ms/step -
loss: 63.3496 - val_loss: 37.4291
Epoch 129/200
11/11             0s 17ms/step -
loss: 63.0614 - val_loss: 37.4612
Epoch 130/200
11/11             0s 18ms/step -
loss: 67.0856 - val_loss: 38.8843
Epoch 131/200
11/11             0s 18ms/step -
loss: 80.2918 - val_loss: 43.9271
Epoch 132/200
11/11             0s 22ms/step -
loss: 79.5727 - val_loss: 55.0062
Epoch 133/200
11/11             0s 21ms/step -
loss: 70.1385 - val_loss: 38.4630
Epoch 134/200
11/11             0s 32ms/step -
loss: 50.2648 - val_loss: 44.9514
Epoch 135/200
11/11             1s 28ms/step -
loss: 70.8141 - val_loss: 45.0127
Epoch 136/200
11/11             0s 26ms/step -
loss: 54.2321 - val_loss: 42.3256
Epoch 137/200
11/11             1s 32ms/step -
loss: 60.5990 - val_loss: 41.5957
Epoch 138/200
11/11             1s 39ms/step -
loss: 69.2141 - val_loss: 42.1470
Epoch 139/200
11/11             1s 38ms/step -
loss: 50.8823 - val_loss: 51.2459
Epoch 140/200
11/11             0s 21ms/step -
loss: 58.0639 - val_loss: 38.5995
Epoch 141/200
11/11             1s 19ms/step -
loss: 55.5084 - val_loss: 38.1223
Epoch 142/200
11/11             0s 25ms/step -
loss: 42.5869 - val_loss: 42.7461
Epoch 143/200
11/11             0s 20ms/step -
loss: 75.5971 - val_loss: 43.5264
```

```
Epoch 144/200
11/11              0s 26ms/step -
loss: 53.2123 - val_loss: 52.3444
```

[10]:
```
print(f"RMSE error using deeper neural network {model.evaluate(X_test,␣
↪y_test)**0.5}")
```

```
3/3             0s 6ms/step - loss:
60.4568
RMSE error using deeper neural network 7.234945254168172
```