Introduction to Supervised Learning and Regression

HL Mathematics IA

Hassan Khurram

Updated: December 2020

Contents

1 Introduction

1.1 Background: Artificial Intelligence

From emerging technologies in self-driving automobiles to personal assistants on our phones, AI is the future. The book *Life 3.0: Being Human in the Age of Artificial Intelligence* by Max Tegmark motivated me to appreciate the gravity of the ever-expanding capacity of AI. With various perspectives that define our ethical and philosophical bounds on its presence in technology, the book and a series of documentaries captivated my interest in this technology. Since my realization sparked amidst the rise of AI, I seek to pursue a field in machine intelligence.

What does 'intelligence' mean in view of artificial life forms? Max Tegmark, an author and physicist at MIT stated that "Intelligence is the ability to achieve complex goals," in broad terms that concern both artificial and biological abilities. (Tegmark) But how does this relate to mathematics, particularly in the fields of calculus and linear algebra? AI and machine learning stem from optimization concepts in calculus. What was initially a daunting concept became conceptualized as the aim to minimize a function used to increase the accuracy of a neural network, or an algorithm that carries out such 'learning'. The network achieves the closest values possible to the actual, desired outputs. The grasp of these concepts enabled me to realize the real-world importance of the calculus optimization problems from class.

1.2 Aim and Rationale

My exploration aims to use functions to express the cost of the outputs—or the predicted housing prices compared to the actual values of prices—used for training a neural network in **supervised learning**. Supervised learning is a process where a neural network, or an algorithm that computes an output given input values adjusts according to how different its output is compared to the actual values. It attempts to model a function to relate inputs to outputs using known examples to learn this relationship. So, in this exploration, my models use regression (for continuous variables) with examples of housing prices, where the price is dependent on multiple variables such as lot size, number of rooms, number of bathrooms, and living area. **Vector calculus** and matrices are used to determine the value of the cost in each 'iteration', or example in a sample of housing data. Since data points are shuffled at random, this method is known as **stochastic gradient descent**, with the aims to show that cost (loss) tends to descend as the number of iterations increases. In other words, the network 'trains' to become more accurate in giving outputs closer to the target, or real values of housing prices. (Ng, 2012)

Note: "Cost" refers to the mean square error, or a function describing the difference between the network's estimated value and actual price of the houses. "Price" refers to the money value (\$) of houses.

2 AI Concepts

2.1 Neural Nets

In artificial intelligence, neural networks—which roughly emulate our understandings of neuroscience related to machine learning—are represented as a layered structure of neurons. Simply, it takes an input of values (in the input layer) for a given number of variables named features and computes the output of the network in the output layer neurons. Mathematically speaking, each layer consists of neurons that store numbers as activations, or the outputs dependent upon the weights and activations of all neurons in a preceding layer (except for input layer neurons). An activation is a number stored by a single neuron in the network. Also, weights linking the neurons may be thought of as the strengths of the connections, or 'synapses' between neurons. First, forward propagation is carried out to obtain the output values from the neural net, which is going from one layer to the next (to the right). (Kriegeskorte, 2019)

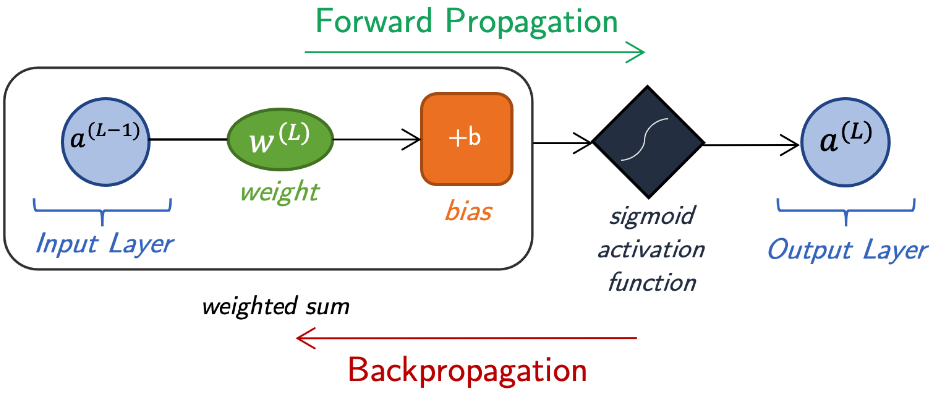

Figure 1: The perceptron: the most basic neural network of two neurons (one in each layer)

Note: The superscript notation with parentheses is not an exponent, but used for indexing the neuron's layer.

$$a^{(L)} = \sigma\left(\left(a^{(L-1)}\right)\left(w^{(L)}\right) + b\right) \tag{1}$$

This equation above is used to calculate the weighted sum, or the output expressed in terms of all inputs, or features (variables) in the input layer. In this case, the input, $a^{(L-1)}$, which is the activation of the 'input layer' neuron, is multiplied by a weight, $w^{(L)}$, and added by a bias, b. In the figure above, the output of the output layer neuron is $a^{(L)}$, when the weighted sum is the input of an activation function: this is represented by the σ (sigma) symbol, which is later discussed. Hence, 'learning'/'training' in machine intelligence is defined as a computer's manipulation of both the weights and biases of every neuron in the neural net to the most optimal values as parameters. Where forward propagation is used to compute cost (error) values in terms of a training set, backpropagation is used to decrease the output of the cost function C(w_1, ..., w_n), where there are n number of weights and biases in total for the neural network linking the neurons. (Le, 2015)

2.2 Activation functions

Activation functions are applied to the weighted sum outputs, or the activations of neurons in neural nets to a defined range. One activation function will be used in calculations of forward and backpropagation, the latter of which is used for gradient descent, or the process of machine 'learning', later explained.

2.2.1 Sigmoid (Logistic) Function

The sigmoid activation function is commonly used in neural networks. They are used for the activations of all neurons in classification problems. The rationale pertaining to the use of this function is later explained in detail. The function is defined as $y = \sigma(z)$ with the range of $y \in (0, 1)$. The 'z' variable of the sigmoid function is equal to the weighted sum to a single layer.

$$\lim_{z\to\infty} \sigma(z) = 1 \tag{2}$$

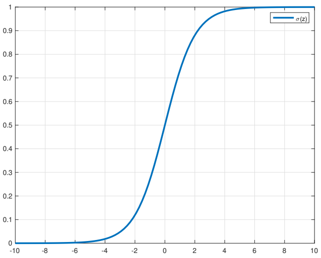

Figure 2: $\sigma\left(z\right)=\frac{1}{1+e^{-z}}$

$$\lim_{z\to-\infty}\sigma\left(z\right)=0 \tag{3}$$

Also, the derivative of this function, which will be used later in partial derivatives in terms of each weight and bias, is:

$$\sigma'\left(z\right)=\frac{e^{-z}}{\left(1+e^{-z}\right)^2} \tag{4}$$

2.3 Gradient Descent Problem

2.3.1 Problem and Cost Functions

The cost function's value, related to the difference between prediction and actual values, is to be minimized. In single variable calculus, the minimum of a function is graphically represented at the point where the first derivative of the function is equal to zero, and the second derivative is positive, used to directly solve for the local minimum at one point in the plane. However, since the cost function depends on many weights and biases, multivariate calculus (explained later) is used. In this case, a randomized starting point is set before iterative calculations are computed and changes are applied to all the weights and biases in a direction that reduces the cost function's value closer to a local minimum in space, as the global minimum cannot directly be solved for. Testing data (features) are used as inputs in a neural network to result in output (activation), or prediction compared to desired outputs in a defined cost function, using the mean square loss/cost function used commonly for regression back-propagation. (Strang, 2018)

$$C\left(w_1,\ldots,w_n\right)=\frac{1}{2}\left(a^{(L)}-y\right)^2 a^{(L)}=activation\,in\,L \tag{5}$$

$y=actual\ value\ of\ the\ output$

Here, the mean square loss function is used as errors may be negative, and so the squares of the errors are taken for each output in a data training set to give a non negative value of the error. Also, a factor of $\frac{1}{2}$ is multiplied with the square loss term so that the derivative function does not contain a factor. Overall, the problem investigated is based on iteratively minimizing this cost value, C. (Ng, 2012)

3 Introducing Vector Calculus

The use of matrices that follow were largely introduced to me in grade 10 mathematics. I was inspired by the idea of involving my exploration with matrices, intertwined with applied calculus. So, I now introduce multivariate functions and calculus. When dealing with multivariate functions, in which the output, say f, depends on more than one variable and not just

x, it is appropriate to think of the derivative of a function in terms of vectors. When taking the derivative in terms of each variable at a time, the rate of change of the function is taken in terms of only one variable, while the other variables are considered constant. Such is expressed as a partial derivative, where derivatives are also to be taken in terms of other variables present in the function's arguments (Dawkins 2018).

Starting with an example, first consider a (2 variable) function $f(x, y) = 2x^2 + 3y$. In this case, f is dependent upon both variables x and y. If the partial derivative is taken in terms of x first, then variable y is considered a constant, while taking the derivative in terms of x using the same principle of the power rule from single-variable calculus. Hence,

$$\frac{\partial}{\partial x} f(x, y) = 4x + 0 = 4x \tag{6}$$

Note: $\frac{\partial}{\partial x}$ is the notation for the partial derivative, in terms of variable x.

Following the same method, taking the partial derivative in terms of y requires the variable x to be dealt as a constant. So,

$$\frac{\partial}{\partial y} f(x, y) = 0 + 3 = 3 \tag{7}$$

When considering the gradient of a bi-variate function, or $f(x, y)$, the following vector notation is utilized to denote the partial derivatives, or the rates of change in f in terms of both dimensions: x and y. The Jacobian expresses a multivariate function's first-order partial derivatives in terms of all variables:

$$(x, y) = \begin{bmatrix} \frac{\partial}{\partial x} f(x, y) \\ \frac{\partial}{\partial y} f(x, y) \end{bmatrix} \tag{8}$$

$$\therefore\ (x, y) = \begin{bmatrix} 4x \\ 3 \end{bmatrix} \tag{9}$$

What this vector gradient represents is the direction of steepest ascent, or where the output, f, increases most quickly in terms of both inputs. Visualising this vector gradient function, the following 3-dimensional graph represents the vector notation of the gradient function ∇f. (Dawkins, 2018)

If a function h has n variables, such that $h(x_1, \ldots, x_n)$, then its Jacobian, a matrix representing all first-order partial derivatives, will be expressed as,

$$(x_1,\ \ldots,\ x_n) = \begin{bmatrix} \frac{\partial}{\partial x_1} h(x_1,\ \ldots,\ x_n) \\ \vdots \\ \frac{\partial}{\partial x_n} h(x_1,\ \ldots,\ x_n) \end{bmatrix} \tag{10}$$

4 Regression Supervised Learning: A Network for Housing Prices

4.1 Looking at Data

In this exploration, I use the available data of housing prices in Seattle, Washington from Kaggle, and thus use a model of a supervised, regression machine learning algorithm that models a multivariate function. Such relates to my own search for a future residence at university. In a training data set, the different variables are named as 'features' or inputs to a network of neurons that determines the values of the output. The graph in figure 4, however, is a set of 30 randomized points from the data in only a two dimensional plane. (Kaggle)

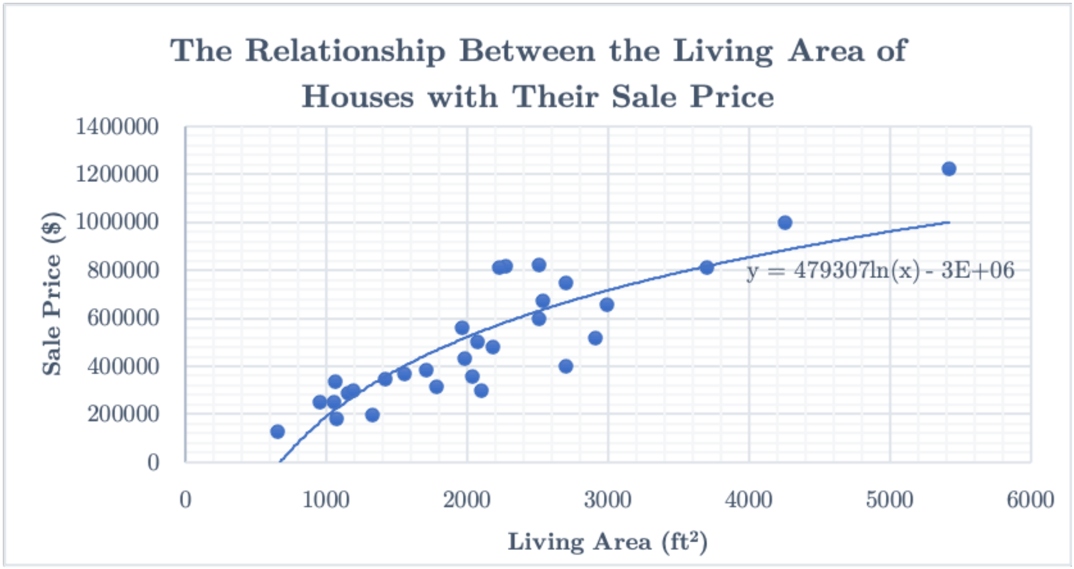

Figure 3: A graph with non-linear regression

In this exploration, I use the available data of housing prices in Seattle, Washington (Kaggle), and thus use a model of a supervised, regression machine learning algorithm that models a multivariate function. Such relates to my own search for a future residence at university. In a training data set, the different variables are named as 'features' or inputs to a network of neurons that determines the values of the output. The graph in figure 4, however, is a set of 30 randomized points from the data in only a two dimensional plane.

Logarithmic regression can be used to model the functions that represent the relationship between one variable, say the living area—as visualized in figure 4 (created using Excel)—with the selling price in dollars ($). But when more than a single independent variable is introduced, such as lot size, number of bedrooms/bathrooms and living area for a data set with 30 examples, the trend cannot be visually represented after solving for the parameters of a function with 4 independent variables. Hence, using a trained neural network would be viable to input a vector of 4 features (variables) in the network so as to output a value that is later passed on to a cost function, that is minimized with each data point, or iteration. In other words, the weights and biases are the parameters for the learning function used for regression, which are updated in gradient descent after each training example.

4.2 A Theoretical Neural Network to Model Continuous Values: Housing Prices

The model of the neural network for regression will involve a single neuron in the output layer for a single output value, or the predicted housing price. The number of neurons in the input layer will be equal to the number of variables in the feature (input layer) vector. The network will involve a non-linear activation function on the hidden layer neurons, or neurons in layers between the inputs and output. Also for regression, the activation function will not be applied to the output layer neuron, since sigmoid functions are bounded on $(0, 1)$, and thus do not possess ranges suitable to output larger predicted housing price values. In general, the hidden layers will be included to take into account the complexity of non-linear, multivariate regression using neural nets in machine learning algorithms, where the activation functions apply to the weighted sums. (Ng, 2012)

Furthermore, as detailed by a paper from Columbia University, the process starts with "weights randomly initialized and then adjusted in many small steps to bring the network closer to the desired behavior." (Kriegeskorte, 2019) In figure 7, there is one hidden layer for the feed-forward, shallow artificial neural network. The hidden layer is included since the activation functions on the neurons in this layer have linear weighted sums (expressed as $a^{(L-1)}(w^L)+b$) followed by a 'squashing' non-linearity to result in the activations accounting for non-linear regression. The single hidden layer network is able to "approximate any function that contains a continuous mapping from one finite space to another", or features to labels. Also, the number of hidden layer neurons is 3, between the number of input and output neurons for clarity. I established the following setup of the synapses and neurons below:

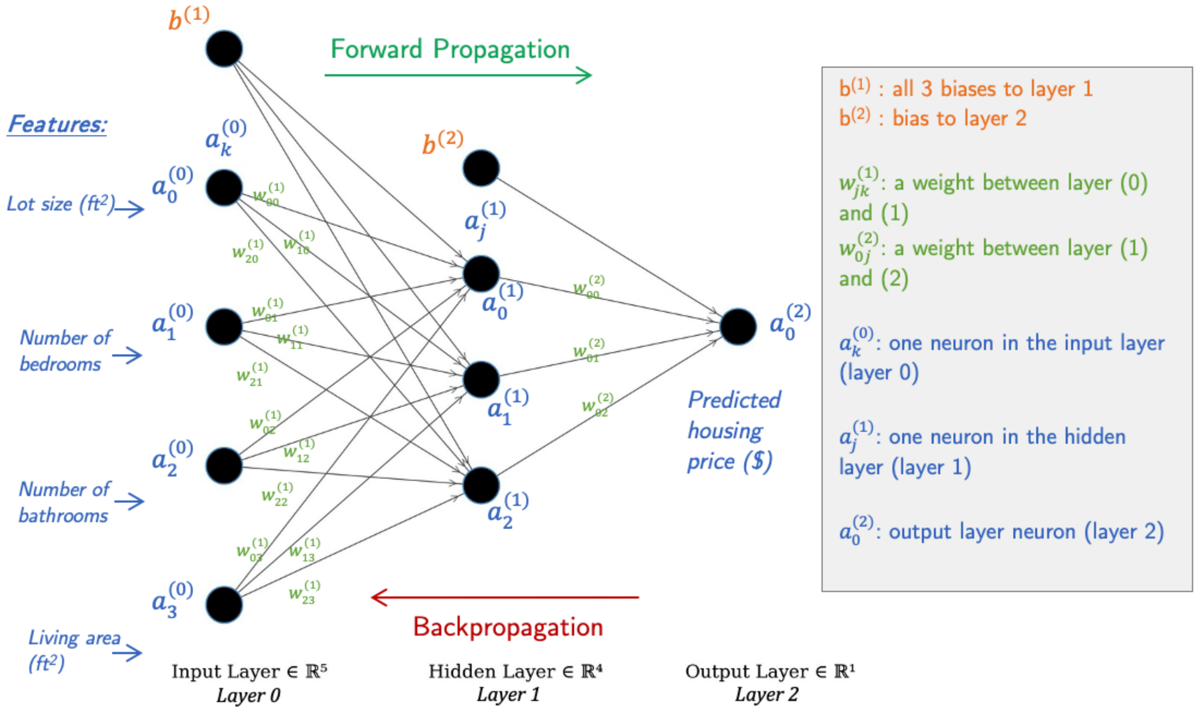

Figure 4: Neural Network Model. This visualization was created using "alexlenail".

4.3 Backpropagation for Gradient Descent with an Activation Function

For each iteration, supervised learning is used to 'train' the neural net. The purpose is to minimize the cost function in stochastic (random) gradient descent, in which the backpropagation procedure evaluates the gradients of the cost function. In stochastic gradient descent, the batch size, or the number of examples per forward and backward pass that defines one iteration, is one. In each iteration, the weights and biases values are updated. The features are the inputs of forward propagation before calculating the cost. Then, the first partial derivatives of the cost function are calculated and represented as a matrix (Jacobian) that, with respect to each weight and bias, provides both the magnitude and direction (sign) of change to update the weights and biases (parameters) in each iteration. (Rocca, 2018) Hence, backpropagation is essentially using matrix derivatives that express, using the chain rule, the cost function as the composition of the function in terms of the weights and biases. It is important to realize that the output neuron's activations depend on the value of the features.

5 The Neural Network's Functions

5.1 Deriving Equations for Forward Propagation:

The sigmoid function allows for 'logistic regression' to take place within the hidden layers using the non-linear activation function to model an unknown function for the relationship between all the variables and housing prices. Each element of the matrices (array of numbers) represent either a parameter or activation to or from the neurons in each layer. The following equations, with the sigmoid activation function applied to the weighted sum to layer (1) neurons, will later be used to find the first partial derivatives useful in backpropagation. For forward propagation, the process of finding the network's output, $a_0^{(2)}$, employs the following equation involving all neurons in each layer. In terms of all of the weights and biases to first solve for the outputs, or activations of layer (1) neurons, the following matrix equation for forward propagation from layer (0) to (1) is:

$$a^{(1)} = w^{(1)} \cdot a^{(0)} + b^{(1)} \tag{11}$$

$$a^{(1)} = \begin{bmatrix} a_0^{(1)} \\ a_1^{(1)} \\ a_2^{(1)} \end{bmatrix} = \sigma \left(\begin{bmatrix} w_{00}^{(1)} a_0^{(0)} + w_{01}^{(1)} a_1^{(0)} + w_{02}^{(1)} a_2^{(0)} + w_{03}^{(1)} a_3^{(0)} + b_0^{(1)} \\ w_{10}^{(1)} a_0^{(0)} + w_{11}^{(1)} a_1^{(0)} + w_{12}^{(1)} a_2^{(0)} + w_{13}^{(1)} a_3^{(0)} + b_1^{(1)} \\ w_{20}^{(1)} a_0^{(0)} + w_{21}^{(1)} a_1^{(0)} + w_{22}^{(1)} a_2^{(0)} + w_{23}^{(1)} a_3^{(0)} + b_2^{(1)} \end{bmatrix} \right) \tag{12}$$

However, the sigmoid activation function is not applied to the output layer, which is a continuous value of housing price. A linear "activation" function is instead used, such that the output, $a_0^{(2)} = z^{(2)}$, where $z^{(2)}$ is the weighted sum to the output layer neuron. Thus, the forward propagation equation to the layer (2) neuron is:

$$a^{(2)} = w^{(2)} \cdot a^{(1)} + b^{(2)} a^{(2)} = \left(\begin{bmatrix} w_{00}^{(2)} & w_{01}^{(2)} & w_{02}^{(2)} \end{bmatrix} \begin{bmatrix} a_0^{(1)} \\ a_1^{(1)} \\ a_2^{(1)} \end{bmatrix} + \begin{bmatrix} b_0^{(2)} \end{bmatrix} \right) \tag{13}$$

$$\begin{bmatrix} a_0^{(2)} \end{bmatrix} = \left(\begin{bmatrix} w_{00}^{(2)} a_0^{(1)} + w_{01}^{(2)} a_1^{(1)} + w_{02}^{(2)} a_2^{(1)} + b_0^{(2)} \end{bmatrix} \right) \tag{14}$$

5.2 Deriving Equations for Backpropagation:

To correlate with my neural network, I derived the following equations, extending on the models for basic neural networks researched. First, the partial derivative in terms of a single weight is considered, which will be a part of the matrix-vector that represents the overall cost function for backpropagation used for 30 examples/iterations (30 houses). The cost function C(...) is defined in terms of all the weights and biases present in the neural network. (Urtasun et al., 2015) Hence, the cost function is defined as:

$$C\left(w_{00}^{(1)}, \ldots, w_{jk}^{(1)},\ w_{0j}^{(2)}, b_j^{(1)}, b_0^{(2)} \right) = \frac{1}{2} \left(a_0^{(2)} \ - \ y \right)^2 \tag{15}$$

To represent such with matrices, the **Jacobian** shows all the values of the first partial derivatives of the cost function with respect to each weight and bias. The number of partial derivatives is equal to the total number of weights and biases. The cost function is defined in terms of 4 matrices, for the weights and biases between each of the 3 layers. The Jacobians are:

$$\frac{\partial C}{\partial w^{\mathbf{(2)}}} = \begin{bmatrix} \frac{\partial C}{\partial w_{\mathbf{00}}^{\mathbf{(2)}}} \\ \frac{\partial C}{\partial w_{\mathbf{01}}^{\mathbf{(2)}}} \\ \frac{\partial C}{\partial w_{\mathbf{02}}^{\mathbf{(2)}}} \end{bmatrix} \tag{16}$$

$$\frac{\partial C}{\partial w^{(\mathbf{1})}} = \begin{bmatrix} \frac{\partial C}{\partial w_{\mathbf{00}}^{(\mathbf{1})}} & \frac{\partial C}{\partial w_{\mathbf{01}}^{(\mathbf{1})}} & \frac{\partial C}{\partial w_{\mathbf{02}}^{(\mathbf{1})}} & \frac{\partial C}{\partial w_{\mathbf{03}}^{(\mathbf{1})}} \\ \frac{\partial C}{\partial w_{\mathbf{10}}^{(\mathbf{1})}} & \frac{\partial C}{\partial w_{\mathbf{11}}^{(\mathbf{1})}} & \frac{\partial C}{\partial w_{\mathbf{12}}^{(\mathbf{1})}} & \frac{\partial C}{\partial w_{\mathbf{13}}^{(\mathbf{1})}} \\ \frac{\partial C}{\partial w_{\mathbf{20}}^{(\mathbf{1})}} & \frac{\partial C}{\partial w_{\mathbf{21}}^{(\mathbf{1})}} & \frac{\partial C}{\partial w_{\mathbf{22}}^{(\mathbf{1})}} & \frac{\partial C}{\partial w_{\mathbf{23}}^{(\mathbf{1})}} \end{bmatrix} \qquad \frac{\partial C}{\partial b^{(\mathbf{2})}} = \begin{bmatrix} \frac{\partial C}{\partial b_{\mathbf{0}}^{(\mathbf{2})}} \end{bmatrix} \tag{17}$$

$$\frac{\partial C}{\partial b^{(\mathbf{1})}} = \begin{bmatrix} \frac{\partial C}{\partial b_{\mathbf{0}}^{(\mathbf{2})}} \\ \frac{\partial C}{\partial b_{\mathbf{1}}^{(\mathbf{2})}} \\ \frac{\partial C}{\partial b_{\mathbf{2}}^{(\mathbf{2})}} \end{bmatrix} \tag{18}$$

These derivatives, in terms of each weight and bias in the network, may be thought of as the representative vectors in a space with 4 input variables and one output (house price), and thus 5 dimensions, where there is the steepest descent towards a minimum value of the cost function at a point in space; hence the name of "gradient descent". Each single partial derivative term in each of these 4 Jacobians is solved for below.

Solving $\frac{\partial C}{\partial w^{(2)}}$ Weights from layer (1) to (2)

In the cost function $C(\dots)$, the derivative of y (actual housing price) with respect to $^{(2)}$ is 0 since it is a fixed value in an example from the data set used. To find the cost function's derivative in terms of one of the weights from layer (1) to (2), or from neuron j in layer (1) to neuron 0 in layer (2), the following chain rule using partial derivatives is applied:

$$\frac{\partial C}{\partial \mathbf{w}_{\mathbf{0}j}^{(\mathbf{2})}} = \frac{\partial C}{\partial a_{\mathbf{0}}^{(\mathbf{2})}} \frac{\partial a_{\mathbf{0}}^{(\mathbf{2})}}{\partial z^{(\mathbf{2})}} \frac{\partial z^{(\mathbf{2})}}{\partial w_{\mathbf{0}j}^{(\mathbf{2})}} \tag{19}$$

$$\mathbf{C}\left(\dots\right) = \frac{\mathbf{1}}{\mathbf{2}}\left(\mathbf{a}_{\mathbf{0}}^{(\mathbf{2})} \ - \ \mathbf{y}\right)^{\mathbf{2}} \tag{20}$$

The mean square cost function is used after each single iteration.

$$\frac{\partial C}{\partial a_{\mathbf{0}}^{(\mathbf{2})}} = (a_{\mathbf{0}}^{(\mathbf{2})} - y) \tag{21}$$

Derivative is taken for one training example.

$$\frac{\partial C}{\partial \mathbf{w}_{\mathbf{0j}}^{(\mathbf{2})}} = (\mathbf{a}_{\mathbf{0}}^{(\mathbf{2})} - \mathbf{y}) \frac{\partial \mathbf{a}_{\mathbf{0}}^{(\mathbf{2})}}{\partial \mathbf{z}^{(\mathbf{2})}} \frac{\partial \mathbf{z}^{(\mathbf{2})}}{\partial \mathbf{w}_{\mathbf{0j}}^{(\mathbf{2})}} \tag{22}$$

In the derivative of the cost function, C, with respect to the weight, $w_{0j}^{(2)}$, the magnitude of the change in the cost function is proportional to the difference between the network's output $a_0^{(2)}$ and the target value, y. Hence, if a training example's prediction or activation on the output layer is nearer to the actual value, y, then the change in the parameters, or weights is smaller. Moreover, since the activation function is linear for the output layer as the values on the output are unbounded, the following is defined:

$$a_0^{(2)} = z^{(2)} = \left(\sum_{j=0}^{2} \left(w_{0j}^{(2)}\right)\left(a_j^{(1)}\right)\right) + \ b^{(2)} \tag{23}$$

$$\frac{\partial C}{\partial \mathbf{w}_{\mathbf{0j}}^{(\mathbf{2})}} = (\mathbf{a}_{\mathbf{0}}^{(\mathbf{2})} - \mathbf{y}) \frac{\partial \mathbf{a}_{\mathbf{0}}^{(\mathbf{2})}}{\partial \mathbf{z}_{\mathbf{0}}^{(\mathbf{2})}} \frac{\partial \mathbf{z}^{(\mathbf{2})}}{\partial \mathbf{w}_{\mathbf{0j}}^{(\mathbf{2})}} \tag{24}$$

$$a_0^{(2)} = z^{(2)} \therefore \frac{\partial a_0^{(2)}}{\partial z_0^{(2)}} = 1. \tag{25}$$

$$\frac{\partial \mathbf{z^{(2)}}}{\partial \mathbf{w_{0\mathit{j}}^{(2)}}} = a_j^{\mathbf{(1)}} \tag{26}$$

In the summation expression above, the following derivative is taken in terms of a single weight only, so all other terms (from other layer (1) neurons) are constants.

$$\boxed{\therefore \frac{\partial C}{\partial w_{0j}^{(2)}} = (a_0^{(2)} - y)(a_j^{(1)})} \tag{27}$$

Solving $\frac{\partial C}{\partial b^{(2)}}$ Biases to layer (2)

$$\frac{\partial C}{\partial b^{(2)}} = \frac{\partial C}{\partial a_0^{(2)}} \frac{\partial a_0^2}{\partial z_0^{(2)}} \frac{\partial z_0^{(2)}}{\partial b^{(2)}} \tag{28}$$

$$\frac{\partial C}{\partial a_0^{(2)}} = (a_0^{(2)} - y) \tag{29}$$

The power rule is used on the same mean square cost function.

$$\frac{\partial C}{\partial b^{(2)}} = (a_0^{(2)} - y) \frac{\partial a_0^{(2)}}{\partial z^{(2)}} \frac{\partial z_0^{(2)}}{\partial b^{(2)}} \tag{30}$$

$$a_0^{(2)} = z^{(2)} = \left(\sum_{j=0}^{2} \left(w_{0j}^{(2)} \right) \left(a_j^{(1)} \right) \right) + \ b^{(2)} \tag{31}$$

To layer (2) there is no activation function applied to the weighted sum, $z^{(2)}$.

$$\frac{\partial C}{\partial b^{(2)}} = (a_0^{(2)} - y)(1) \frac{\partial z_0^{(2)}}{\partial b^{(2)}} \tag{32}$$

$$a_0^{(2)} = z^{(2)} \therefore \frac{\partial a_0^{(2)}}{\partial z_0^{(2)}} = 1. \tag{33}$$

$$\frac{\partial z_0^{(2)}}{\partial b^{(2)}} = 1 \tag{34}$$

In solving for $\frac{\partial z_0^{(2)}}{\partial b^{(2)}}$, the term $\sum_{j=0}^{2} \left(w_{0j}^{(2)} \right) \left(a_j^{(1)} \right)$ is a constant, of which its derivative in terms of $b^{(2)}$ is 0. Hence,

$$\boxed{\therefore \frac{\partial C}{\partial b^{(2)}} = (a_0^{(2)} - y)} \tag{35}$$

Solving $\frac{\partial C}{\partial w^{(1)}}$: Weights from layer (0) to (1)

In order to solve for the derivative in terms of $w_{jk}^{(1)}$, a single weight from neuron k in layer (0) to neuron j in layer (1), the multivariate chain rule is used. The cost function, C, is determined by the value of the output of the neural network, $a_0^{(2)}$, which is determined by the value of the weighted sum $z_0^{(2)}$, determined by the activation of one neuron in the previous layer. This activation, $a_j^{(1)}$, is determined by the weighted sum, $z_j^{(1)}$ to the neuron that is linked with the single weight the partial derivative is taken in terms of.

$$\frac{\partial C}{\partial \mathbf{w}_{jk}^{\mathbf{(1)}}} = \frac{\partial C}{\partial \mathbf{a_0^{(2)}}} \frac{\partial \mathbf{a_0^{(2)}}}{\partial \mathbf{z_0^{(2)}}} \frac{\partial \mathbf{z_0^{(2)}}}{\partial \mathbf{a}_j^{\mathbf{(1)}}} \frac{\partial \mathbf{a}_j^{\mathbf{(1)}}}{\partial \mathbf{z}_j^{\mathbf{(1)}}} \frac{\partial \mathbf{z}_j^{\mathbf{(1)}}}{\partial \mathbf{w}_{jk}^{\mathbf{(1)}}} \tag{36}$$

The multivariate chain rule is used.

$$a_j^{(\mathbf{1})} = \sigma\left(z_j^{(\mathbf{1})}\right) = \sigma\left(\left(\sum_{k=\mathbf{0}}^{\mathbf{3}}\left(w_{jk}^{(\mathbf{1})}\right)\left(a_k^{(\mathbf{0})}\right)\right) + \mathbf{b}_j^{(\mathbf{1})}\right) \tag{37}$$

The weighted sum is the input of the sigmoid function to output the activation of neuron j in layer (1).

$$\mathbf{a}_{\mathbf{0}}^{(\mathbf{2})} = z^{(\mathbf{2})} = \left(\sum_{j=\mathbf{0}}^{\mathbf{2}}\left(w_{\mathbf{0}j}^{(\mathbf{2})}\right)\left(a_j^{(\mathbf{1})}\right)\right) + \mathbf{b}^{(\mathbf{2})} \tag{38}$$

The activation of the layer (2) neuron is the sum of the product of weights and activations from layer (1) neurons, added with a bias.

$$\frac{\partial C}{\partial \mathbf{w}_{jk}^{(\mathbf{1})}} = (a_{\mathbf{0}}^{(\mathbf{2})} - y)(\mathbf{1})(w_{\mathbf{0}j}^{(\mathbf{2})})\left(\sigma\prime\left(z_{\mathbf{0}}^{(\mathbf{1})}\right)\right)(a_k^{(\mathbf{0})}) \tag{39}$$

$$\sigma'\left(z_{\mathbf{0}}^{(\mathbf{1})}\right) = \frac{\mathbf{e}^{-z_j^{(\mathbf{1})}}}{\left(\mathbf{1} + \mathbf{e}^{-z_j^{(\mathbf{1})}}\right)^{\mathbf{2}}} \tag{40}$$

The derivative of the activation $a_j^{(1)}$ with respect to $z_j^{(1)}$ is the derivative of the sigmoid function, as calculated before.

$$\boxed{\therefore \frac{\partial C}{\partial w_{jk}^{(1)}} = (a_0^{(2)} - y)(w_{0j}^{(2)})\left(\frac{e^{-z_j^{(1)}}}{\left(1 + e^{-z_j^{(1)}}\right)^2}\right)(a_k^{(0)})} \tag{41}$$

Solving $\frac{\partial C}{\partial b^{(1)}}$ Biases to layer (1)

The multivariate chain rule is similarly used to solve for the derivative of the cost function with respect to one of the three biases to layer (1) neurons. Hence,

$$\frac{\partial C}{\partial b_j^{(1)}} = \frac{\partial C}{\partial a_0^{(2)}}\frac{\partial a_0^{(2)}}{\partial z_0^{(2)}}\frac{\partial z_0^{(2)}}{\partial a_j^{(1)}}\frac{\partial a_j^{(1)}}{\partial z_j^{(1)}}\frac{\partial z_j^{(1)}}{\partial b_j^{(1)}} \tag{42}$$

$$a_j^{(\mathbf{1})} = \sigma\left(z_j^{(\mathbf{1})}\right) \tag{43}$$

$$z_j^{(\mathbf{1})} = \left(\sum_{k=\mathbf{0}}^{\mathbf{3}}\left(w_{jk}^{(\mathbf{1})}\right)\left(a_k^{(\mathbf{0})}\right)\right) + \mathbf{b}_j^{(\mathbf{1})} \tag{44}$$

The activation on each layer (1) neuron is the output of the sigmoid function with an input of the weighted sum, $z_j^{(\mathbf{1})}$, to the neuron concerned.

$$\mathbf{z}_{\mathbf{0}}^{(\mathbf{2})} = z^{(\mathbf{2})} = \left(\sum_{j=\mathbf{0}}^{\mathbf{2}}\left(w_{\mathbf{0}j}^{(\mathbf{2})}\right)\left(a_j^{(\mathbf{1})}\right)\right) + \mathbf{b}^{(\mathbf{2})} \tag{45}$$

This is the weighted sum to the layer (2) neuron.

$$\frac{\partial \mathbf{z}_j^{(\mathbf{1})}}{\partial \mathbf{b}_j^{(\mathbf{1})}} = \mathbf{1} \tag{46}$$

The derivative of $z_j^{(1)}$ with respect to a single bias to a layer 1 neuron is 1, since the weighted sum excluding the bias is a constant.

$$\frac{\partial C}{\partial \mathbf{b}_j^{(\mathbf{1})}} = (a_{\mathbf{0}}^{(\mathbf{2})} - y)(\mathbf{1})(w_{\mathbf{0}j}^{(\mathbf{2})})(\sigma'\left(\mathbf{z}^{(\mathbf{1})}\right))(\mathbf{1}) \tag{47}$$

$$\boxed{\therefore \frac{\partial C}{\partial \mathbf{b}_j^{(\mathbf{1})}} = (a_{\mathbf{0}}^{(\mathbf{2})} - y)(w_{\mathbf{0}j}^{(\mathbf{2})})\left(\frac{e^{-z_j^{(\mathbf{1})}}}{\left(\mathbf{1} + e^{-z_j^{(\mathbf{1})}}\right)^{\mathbf{2}}}\right)} \tag{48}$$

$\sigma'\left(z^{(1)}\right)$ is calculated before.

5.3 Training Example 1

5.3.1 Forward Propagation with Example 1:

The first example from the training data may be used to first calculate the output of the network using forward propagation, before using backwards propagation to calculate the updated weights that result in a lower cost value for subsequent iterations. The weights and biases are initialized with random values. The following numbers were the outputs of the code written in Python 3.0.

```
import random
W = round(random.uniform(0, 0.001),10)
print(W)
```

The weights and biases were randomized in the range (0, 0.001) as in the sigmoid activation function, the gradient is close to zero when z, the weighted sum, is large. All the randomized initial weights and biases are:

$$\begin{bmatrix} b_{\mathbf{0}}^{(\mathbf{1})} \\ b_{\mathbf{1}}^{(\mathbf{1})} \\ b_{\mathbf{2}}^{(\mathbf{1})} \end{bmatrix} = \begin{bmatrix} \mathbf{0.0003656608} \\ \mathbf{0.0007430702} \\ \mathbf{0.0004790847} \end{bmatrix} \quad \begin{bmatrix} b_{\mathbf{0}}^{(\mathbf{2})} \end{bmatrix} = [\mathbf{0.0007412303}] \tag{49}$$

$$\begin{bmatrix} w_{00}^{(1)} & w_{01}^{(1)} & w_{02}^{(1)} & w_{03}^{(1)} \\ w_{10}^{(1)} & w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} \\ w_{20}^{(1)} & w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} \end{bmatrix} = \begin{bmatrix} 0.0001579 & 0.0006046 & 0.0001568 & 0.0009584 \\ 0.0004433 & 0.0005851 & 0.0007728 & 0.0004575 \\ 0.0003029 & 0.0001045 & 0.0009119 & 0.0002420 \end{bmatrix} \tag{50}$$

$$\begin{bmatrix} w_{00}^{(2)} & w_{01}^{(2)} & w_{02}^{(2)} \end{bmatrix} = \begin{bmatrix} 0.0007504129 & 0.0004124064 & 0.0006434057 \end{bmatrix} \tag{51}$$

The feature vector, or the input variables of the first example house to the input layer is written as:

$$\begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ a_2^{(0)} \\ a_3^{(0)} \end{bmatrix} = \begin{bmatrix} \textit{lot size } (ft^2) \\ \textit{number of bedrooms} \\ \textit{number of bathrooms} \\ \textit{living area } (ft^2) \end{bmatrix} = \begin{bmatrix} 11200 \\ 4 \\ 2.5 \\ 2180 \end{bmatrix} \tag{52}$$

Layer 1 Operations (activations)

Note: The values calculated from here are noted to 3 decimal places. The values are stored in the GDC and are not rounded. The quoted values for the prices (\$) are rounded to 2 decimal places. The following computations are solved for on the TI-84 GDC.

$$\begin{bmatrix} a_0^{(1)} \\ a_1^{(1)} \\ a_2^{(1)} \end{bmatrix} = \sigma\left(\begin{bmatrix} w_{00}^{(1)}a_0^{(0)} + w_{01}^{(1)}a_1^{(0)} + w_{02}^{(1)}a_2^{(0)} + w_{03}^{(1)}a_3^{(0)} + b_0^{(1)} \\ w_{10}^{(1)}a_0^{(0)} + w_{11}^{(1)}a_1^{(0)} + w_{12}^{(1)}a_2^{(0)} + w_{13}^{(1)}a_3^{(0)} + b_1^{(1)} \\ w_{20}^{(1)}a_0^{(0)} + w_{21}^{(1)}a_1^{(0)} + w_{22}^{(1)}a_2^{(0)} + w_{23}^{(1)}a_3^{(0)} + b_2^{(1)} \end{bmatrix}\right) = \sigma\left(\begin{bmatrix} 3.861... \\ 5.966... \\ 3.923... \end{bmatrix}\right) \tag{53}$$

Here, the sigmoid function is applied:

$$\begin{bmatrix} a_0^{(1)} \\ a_1^{(1)} \\ a_2^{(1)} \end{bmatrix} = \begin{bmatrix} 0.979... \\ 0.997... \\ 0.980... \end{bmatrix} \tag{54}$$

Layer 2 Operations (activation, neural net's output)

$$\begin{bmatrix} a_0^{(2)} \end{bmatrix} = \left(\begin{bmatrix} w_{00}^{(2)}a_0^{(1)} + w_{01}^{(2)}a_1^{(1)} + w_{02}^{(2)}a_2^{(1)} + b_0^{(2)} \end{bmatrix}\right)\begin{bmatrix} a_0^{(2)} \end{bmatrix} \tag{55}$$

$$\begin{bmatrix} a_0^{(2)} \end{bmatrix} = \$0.00251... \approx \$0.00 \tag{56}$$

Calculated above is the network's predicted housing price based on the input features. As the network has not at all been 'trained' yet, the output value on the first iteration is far-off, in which the mean square cost value is large.

$$C(\ldots) = \frac{1}{2}(0.00251... - 480500)^2 = 1.15 \times 10^{11} \tag{57}$$

This cost value is used in backpropagation (below) to solve for the partial derivatives used to update weights and biases

5.3.2 Backpropagation with Example 1

In this neural network, the weights and biases are updated after each iteration which consists of a single training example.

Backpropagating in terms of the weights to layer (2)

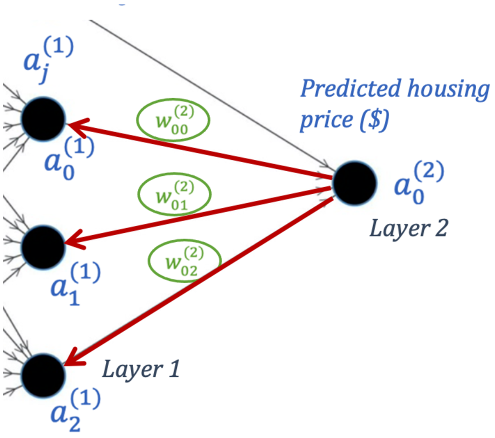

Figure 5

In order to solve for the partial derivatives in terms of the weights to layer 2, I derived the following equation:

$$\frac{\partial C}{\partial \mathbf{w}_{\mathbf{0}j}^{(\mathbf{2})}} = (a_{\mathbf{0}}^{(\mathbf{2})} - y)(a_j^{(\mathbf{1})}) \tag{58}$$

Here, $a_0^{(2)}$ is the output neuron's activation (prediction for housing price ($)) for the first training example, where the desired output is y. $a_j^{(1)}$ represents the activation, or output of a single neuron j in layer (1). This partial derivative is solved with respect to the three weights to layer (2), with j = 0, 1, 2 in the following computations.

$$\frac{\partial C}{\partial w^{(2)}} = \begin{bmatrix} \frac{\partial C}{\partial w_{00}^{(2)}} \\ \frac{\partial C}{\partial w_{01}^{(2)}} \\ \frac{\partial C}{\partial w_{02}^{(2)}} \end{bmatrix} = \begin{bmatrix} -470597.132... \\ -479271.935... \\ -471180.267... \end{bmatrix} \tag{59}$$

$$\frac{\partial C}{\partial w^{(2)}} = \begin{bmatrix} \frac{\partial C}{\partial w_{00}^{(2)}} \\ \frac{\partial C}{\partial w_{01}^{(2)}} \\ \frac{\partial C}{\partial w_{02}^{(2)}} \end{bmatrix}^T = \begin{bmatrix} -470597.132... & -479271.935... & -471180.267... \end{bmatrix} \tag{60}$$

Note: The Jacobian matrix is transposed (rows and columns exchange), as represented by T, so that the dimensions of the matrix (1×3) are the same as that of the matrix of weights for $w^{(2)}$.

Updating Weights to Layer (2)

After I calculated the partial derivatives with respect to each of the weights (the rates of changes of the cost function), the weights are updated using the following equations with respect to each weight in the matrix. In using an update function, a learning rate (η)—a constant multiplied by the cost function's derivative with respect to a weight—would update the network's weights. Accordingly, a research paper on Deep Learning by Quoc V states that learning rates from 0.01 to 0.1 are a "very good start". (Le, 2015) The updates for each weight, **using a learning rate of 0.01**—a lower rate is used to prevent the model from overshooting the optimal weights—are carried out with the following formula adapted from Le's paper, with appropriate notations:

$$\acute{w}_{(0j)}^{(2)} = w_{0j}^{(2)} - \left(\eta \frac{\partial C}{\partial w_{0j}^{(2)}} \right) \tag{61}$$

$$\acute{w}_{0j}^{(\mathbf{2})} = any\ single\ updated\ weight\ from\ layer\ (1)\ to\ layer\ (\mathbf{2}). \tag{62}$$

Hence,

$$\acute{w}^{(2)} = \begin{bmatrix} w_{00}^{(2)} - \left(\eta \frac{\partial C}{\partial w_{00}^{(2)}} \right) & w_{01}^{(2)} - \left(\eta \frac{\partial C}{\partial w_{01}^{(2)}} \right) & w_{02}^{(2)} - \left(\eta \frac{\partial C}{\partial w_{02}^{(2)}} \right) \end{bmatrix} \tag{63}$$

$$\acute{w}^{(\mathbf{2})} = \begin{bmatrix} \mathbf{4705.972}... & \mathbf{4792.719}... & \mathbf{4711.803}... \end{bmatrix} \tag{64}$$

Note: These updated weights would only apply to the second iteration (using the second example).

Backpropagating in terms of the bias to layer (2)

$$\frac{\partial C}{\partial b_0^{(2)}} = (a_0^{(2)} - y) = [-480499.997...] \tag{65}$$

Updating Biases to Layer (2)

$$\acute{b}_0^{(2)} \;=\; b_0^{(2)} - \left(\eta\frac{\partial C}{\partial b_0^{(2)}}\right) = [\mathbf{4805.000}...] \tag{66}$$

Backpropagating in terms of the weights to layer (1)

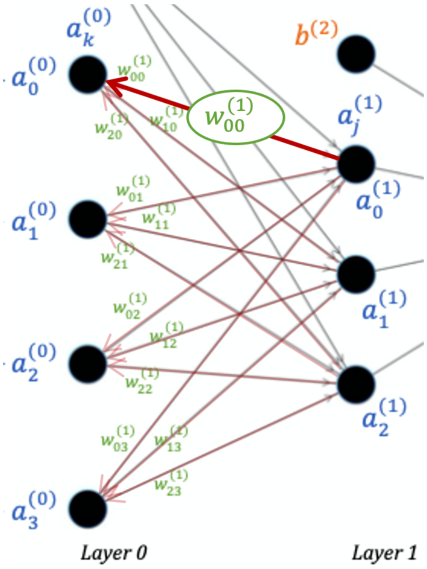

Figure 6

$$\frac{\partial C}{\partial w_{jk}^{(1)}} = (a_0^{(2)} - y)(w_{0j}^{(2)})\left(\sigma'\left(z_0^{(1)}\right)\right)(a_k^{(0)}) \tag{67}$$

The weight $w_{0j}^{(2)}$, is taken from the initial randomized weights to layer (2), and not from the updated weights above. Updated weights will be used in the forward pass using Example 2.

Solving $\frac{\partial C}{\partial w_{00}^{(1)}}$, with $j = 0, k = 0$ (neuron 0 in layer 0 to neuron 0 in layer 1):

$$\frac{\partial C}{\partial w_{00}^{(1)}} = (a_0^{(2)} - y)\left(w_{00}^{(2)}\right)\left(\sigma'\left(z_0^{(1)}\right)\right)(a_0^{(0)}) \tag{68}$$

The weighted sum (summation of the product of layer 1 neuron activations, or the features of the house, and weights) to layer 1 is:

$$z_0^{(1)} = a_0^{(0)}w_{00}^{(1)} + a_1^{(0)}w_{01}^{(1)} + a_2^{(0)}w_{02}^{(1)} + a_3^{(0)}w_{03}^{(1)} + b_0^{(1)} = 3.861... \tag{69}$$

The derivative of the sigmoid function with the weighted sum, $z_0^{(1)}$ is:

$$\sigma'\left(z_0^{(1)}\right) = \frac{e^{-z_0^{(1)}}}{\left(1+e^{-z_0^{(1)}}\right)^2} \;=\; 0.020... \tag{70}$$

$$\frac{\partial C}{\partial w_{00}^{(1)}} = (0.00251... - \;480500)\,(0.000750...)\,(0.020...)\,(11200) = \;-81514.552... \tag{71}$$

The 11 other partial derivatives in the matrix of partial derivatives below with respect to all the weights between layers (0) to (1) are calculated repeating the same method. Hence,

$$\frac{\partial C}{\partial w^{(1)}} = \begin{bmatrix} \frac{\partial C}{\partial w_{00}^{(1)}} & \frac{\partial C}{\partial w_{01}^{(1)}} & \frac{\partial C}{\partial w_{02}^{(1)}} & \frac{\partial C}{\partial w_{03}^{(1)}} \\ \frac{\partial C}{\partial w_{10}^{(1)}} & \frac{\partial C}{\partial w_{11}^{(1)}} & \frac{\partial C}{\partial w_{12}^{(1)}} & \frac{\partial C}{\partial w_{13}^{(1)}} \\ \frac{\partial C}{\partial w_{20}^{(1)}} & \frac{\partial C}{\partial w_{21}^{(1)}} & \frac{\partial C}{\partial w_{22}^{(1)}} & \frac{\partial C}{\partial w_{23}^{(1)}} \end{bmatrix} = \begin{bmatrix} -81514.552 & -29.112 & -18.195 & -15866.225 \\ -5657.864 & -15.999 & -1.262 & -1101.262 \\ -65856.593 & -23.520 & -14.700 & -12818.515 \end{bmatrix} \tag{72}$$

These all represent the factor of the changes to each of the weights represented in this Jacobian matrix.

Updated Weights to Layer (1):

$\acute{w}_{jk}^{(1)} = new,\ updated\ weights\ from\ layer\ \ (1)\ \ to\ \ (2)\,.$
Maintaining the same learning rate of $\eta = 0.01$, the following updates to the weights are applied:

$$\acute{w}^{(1)} = \begin{bmatrix} w_{00}^{(1)} - \left(\eta \frac{\partial C}{\partial w_{00}^{(1)}}\right) & w_{01}^{(1)} - \left(\eta \frac{\partial C}{\partial w_{01}^{(1)}}\right) & w_{02}^{(1)} - \left(\eta \frac{\partial C}{\partial w_{02}^{(1)}}\right) & w_{03}^{(1)} - \left(\eta \frac{\partial C}{\partial w_{03}^{(1)}}\right) \\ w_{10}^{(1)} - \left(\eta \frac{\partial C}{\partial w_{10}^{(1)}}\right) & w_{11}^{(1)} - \left(\eta \frac{\partial C}{\partial w_{11}^{(1)}}\right) & w_{12}^{(1)} - \left(\eta \frac{\partial C}{\partial w_{12}^{(1)}}\right) & w_{13}^{(1)} - \left(\eta \frac{\partial C}{\partial w_{13}^{(1)}}\right) \\ w_{20}^{(1)} - \left(\eta \frac{\partial C}{\partial w_{20}^{(1)}}\right) & w_{21}^{(1)} - \left(\eta \frac{\partial C}{\partial w_{21}^{(1)}}\right) & w_{22}^{(1)} - \left(\eta \frac{\partial C}{\partial w_{22}^{(1)}}\right) & w_{23}^{(1)} - \left(\eta \frac{\partial C}{\partial w_{23}^{(1)}}\right) \end{bmatrix} \tag{73}$$

$$\acute{w}^{(\mathbf{1})} = \begin{bmatrix} \mathbf{815.145}... & \mathbf{0.2917}... & \mathbf{0.182}... & \mathbf{158.663}... \\ \mathbf{56.579}... & \mathbf{0.160}... & \mathbf{0.013}... & \mathbf{11.013}... \\ \mathbf{658.566}... & \mathbf{0.235}... & \mathbf{0.147}... & \mathbf{128.185}... \end{bmatrix} \tag{74}$$

Backpropagating in terms of the biases to layer (1)

$$\frac{\partial C}{\partial b^{(\mathbf{1})}} = \begin{bmatrix} \frac{\partial C}{\partial b_{\mathbf{0}}^{(\mathbf{1})}} \\ \frac{\partial C}{\partial b_{\mathbf{1}}^{(\mathbf{1})}} \\ \frac{\partial C}{\partial b_{\mathbf{2}}^{(\mathbf{1})}} \end{bmatrix} \tag{75}$$

$$\frac{\partial C}{\partial \mathbf{b_0^{(1)}}} = (a_{\mathbf{0}}^{(\mathbf{2})} - y)(w_{\mathbf{00}}^{(\mathbf{2})})\left(\sigma\prime\left(\mathbf{z_0^{(1)}}\right)\right) \tag{76}$$

$$\sigma'\left(\mathbf{z_0^{(1)}}\right) = \left(\frac{e^{-z_{\mathbf{0}}^{(\mathbf{1})}}}{\left(\mathbf{1} + e^{-z_{\mathbf{0}}^{(\mathbf{1})}}\right)^{\mathbf{2}}}\right) = \mathbf{0.020}... \tag{77}$$

$$\frac{\partial C}{\partial \mathbf{b_0^{(1)}}} = -\mathbf{7.278}... \tag{78}$$

Repeating the same method, the two other partial derivatives are:

$$\frac{\partial C}{\partial \mathbf{b_1^{(1)}}} = -\mathbf{0.505}\dots \tag{79}$$

$$\frac{\partial C}{\partial \mathbf{b_2^{(1)}}} = -\mathbf{5.880}... \tag{80}$$

Updating Biases to Layer (1)

$\acute{b}_j^{(1)} = new,\ updated\ biases\ to\ layer\ \ (1)\ \ neurons.$

$$\acute{b}^{(\mathbf{1})} = \begin{bmatrix} \acute{b}_{\mathbf{0}}^{(\mathbf{1})} \\ \acute{b}_{\mathbf{1}}^{(\mathbf{1})} \\ \acute{b}_{\mathbf{2}}^{(\mathbf{1})} \end{bmatrix} = \begin{bmatrix} b_{\mathbf{0}}^{(\mathbf{1})} - \left(\eta \frac{\partial C}{\partial b_{\mathbf{0}}^{(\mathbf{1})}}\right) \\ b_{\mathbf{1}}^{(\mathbf{1})} - \left(\eta \frac{\partial C}{\partial b_{\mathbf{1}}^{(\mathbf{1})}}\right) \\ b_{\mathbf{2}}^{(\mathbf{1})} - \left(\eta \frac{\partial C}{\partial b_{\mathbf{2}}^{(\mathbf{1})}}\right) \end{bmatrix} = \begin{bmatrix} \mathbf{0.073}... \\ \mathbf{0.005}... \\ \mathbf{0.059}... \end{bmatrix} \tag{81}$$

5.4 Training Example 2

5.4.1 Forward Propagation with Example 2

The house for the second example (feature vector) is:

$$\begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ a_2^{(0)} \\ a_3^{(0)} \end{bmatrix} = \begin{bmatrix} \textit{lot size } (ft^2) \\ \textit{number of bedrooms} \\ \textit{number of bathrooms} \\ \textit{living area } (ft^2) \end{bmatrix} = \begin{bmatrix} 10050 \\ 3 \\ 1 \\ 1060 \end{bmatrix} \tag{82}$$

Layer 1 Operations (activations)

Using the updated weights and biases previously calculated, the method for solving for the estimated housing price in forward propagation is repeated in the following. The sigmoid function below has been applied to the new weighted sum. Moreover, the values of the activations to layer (1) in the matrix below approaching 1 as the weighted sums approach infinity are caused by 'vanishing gradients', or that the gradient approaches 0 as the weighted sum increases after each iteration.

$$\begin{bmatrix} a_0^{(1)} \\ a_1^{(1)} \\ a_2^{(1)} \end{bmatrix} = \begin{bmatrix} 1.00 \\ 1.00 \\ 1.00 \end{bmatrix} \tag{83}$$

Layer 2 Operations (activation, neural net's output)

Again, the method used for forward propagation to the output layer's activation is repeated this second iteration.

$$\begin{bmatrix} a_0^{(2)} \end{bmatrix} \approx \$19015.50 \tag{84}$$

After the first iteration, where both forward and backpropagation was performed on one example, the value of the error decreases after the weights are adjusted, as for the second example's cost function value. Note, however, that the effect of the different value of the target price in the second example compared to the first example on the cost value is evaluated in the conclusion. This cost value is to be used in backpropagation.

$$C\left(\ldots\right) = \frac{1}{2}\left(19015.495... - 335000\right)^2 = 4.99 \times 10^{10} \tag{85}$$

6 Further Iterations

6.1 Method and Improving the Model with Keras

To scale up this process for 4600 training examples, I wrote Python code using the *Keras* library integrated with *TensorFlow*, which apply the fundamentals of gradient descent detailed so far. The code (as provided in Appendix A) used to train on the data set was modified considerably.

- The feature vector was instead scaled to values with the range $(0, 1)$ so that, say the weights connected to the feature node with higher values (like the lot area) were updated on a similar scale of other features.

- The **ReLU activation function** was employed for neurons on both the hidden and output layers. As a piece-wise function, positive weighted sums preserve their values, while negative values are zero.
- The `Adam` optimizer with a learning rate of 0.1 was chosen for more efficient learning.
- The loss function used was `mae` (mean absolute error), explained in the next section.

Additionally, the data was split into a shuffled training and validation set. This distinction was made on the basis of optimizing parameter values and the model selection, respectively. As detailed in the conclusion, the noise with the loss largely concerned with the former, simplified approach, was addressed by setting the **batch size** to be 15 rather than 1. This is the number of samples patched and processed at a time, before the weights and biases were updated. Also, the number of **epochs** was 400, the number of times the model trained on the data by passing through all examples in the data set.

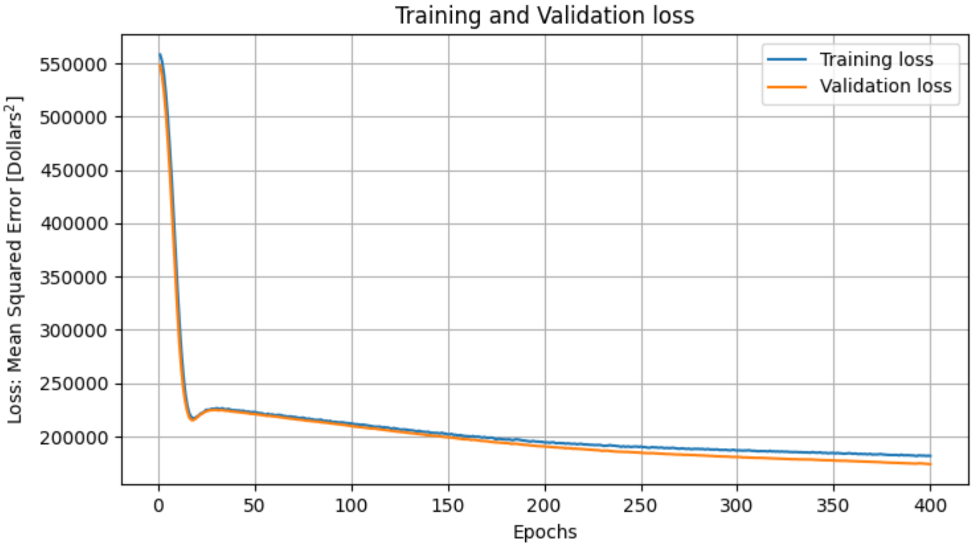

Figure 7: Graph of loss against epochs in learning

6.2 Results of the Neural Network

Matplotlib was used to visualize the resulting loss in the training and validation data over 400 epochs. The figure above is purposed to model the changes in the value of the cost function, whereby the neural network becomes more 'optimal' through 'learning', or in computing each iteration to update weights and biases. This is represented by an initially rapidly decreasing value of the cost function before converging. One interesting result is the slight increase in the loss after about 30 iterations, before decreasing at a slower rate. Notably, the rate of decrease of the cost function also decreases with the number of epochs.

```
Epoch 400/400
197/197 [==============================] - 0s 1ms/step
- loss: 307026296832.0000
- mae: 176583.6875
- mse: 307026296832.0000
- val_loss: 71021813760.0000
- val_mae: 167710.5312 - val_mse: 71021813760.0000
Final Root Mean Squared Error (RMSE) =  $420.218
```

The *metrics* passed to the `model.compile` function to prepare the network for training include `"mse"`, or the mean squared error cost function for m examples in each mini-batch, is represented by

$$C(...) = \frac{1}{2m}\left(\sum_{1}^{m}\left(a_0^{(2)} - y\right)^2\right) \tag{86}$$

Another metric is `"mae"`, or the mean absolute error cost function,

$$C(...) = \frac{1}{m}\left(\sum_{1}^{m}\left|a_0^{(2)} - y\right|\right) \tag{87}$$

6.3 Interpreting Results

In the second iteration of the original neural network, the changes to the weights and biases to layer (1) became more insignificant. Consistent with the result in equation (83), this problem in machine learning is known as 'vanishing gradients', where the increasing value of the weighted sum passed through the sigmoid activation function approaches an output of 1. This corresponds to where the horizontal asymptote (y = 1) in the sigmoid function lies. This means that there are decreased rates of change of the cost with respect to each of the weights and biases to layer (1). Eventually, the only significant adjustments to the weights and biases were for those linked to the output layer neuron (layer 2).

Moreover, the descent of the cost value would be 'noisy' due to stochastic (random) gradient descent. Greater noise, or a larger average spread of data (variance) from the mean cost value is expected in using this method involving single examples randomly selected. (Rocca, 2018) In the improved, **mini-batched gradient descent** algorithm whose result is shown in figure 7, the graph depicts minimal noise. However, as the loss converges to 177,000 [dollars^2] (the square of the final RMSE), the noise in the data begins to increase slightly. The model still vastly improves the representation of learning over varied housing price values being used. Here, the cost function is modified, as it is computed over $m = 15$ shuffled training examples using equation (87).

7 Conclusion

7.1 Limitations and Reflection

One concern with this algorithm is that the computed values for the negative gradients and the local minimum value of the cost function may differ from the global minimum, leaving the possibility of a more accurate model to map features to labels. This was caused by the randomization of the values of weights and biases in the first example before backpropagation. In the case of the mini-batch algorithm implementation, the continually decreasing result in figure 7 after 400 epochs may indicate under-fitting. The implementation of time-based learning rate schedules may address this result for faster training, where learning rates may decrease as training proceeds.

With the finalized weights and biases over 4600 training examples, my neural net achieved the aim to both compute the cost values, which on average, decreased with successive iterations. Thus, the network gradually increased its accuracy at predicting housing prices relative to the initial model (before the first iteration). The model, however, excluded other parameters such as location, age, and condition. By the limitations posed by the availability of data, the model would not closely model the housing price of unseen housing data, as its loss converges to a relatively high RMSE value.

7.2 Beyond the Exploration

Software modules employing deep neural networks are rapidly emerging into various fields. The applications of linear algebra that stem from concepts of matrices and calculus coalesce as the backbone of AI, which vastly enhanced my knowledge and devotion for the wide-ranging field of data science. Software development may be utilized to include this search for homes and their prices in other cities, and more advanced ML models may be applied to data concerning a wide range of fields including medicine and computer vision.

References

[1] "33. Neural Nets and the Learning Function." Lecture by Gilbert Strang, MIT OpenCourseWare, YouTube, 16 May 2019, www.youtube.com/watch?v=L3-WFKCW-tY.

[2] Choromanska, Anna, et al. "The Loss Surfaces of Multilayer Networks." The Loss Surfaces of Multilayer Networks, NYU: Courant Institute of Mathematical Sciences, 2015, proceedings.mlr.press/v38/choromanska15.pdf.

[3] Dawkins, Paul. "Calculus III - Chain Rule." Partial Derivatives, 2018, tutorial.math.lamar.edu/Classes/CalcIII/ChainRule.aspx.

[4] Heaton, Jeff. The Number of Hidden Layers. Heaton Research, 1 June 2017, www.heatonresearch.com/2017/06/01/hidden-layers.html.

[5] Kriegeskorte, Nikolaus. Neural Network Models and Deep Learning – a Primer for Biologists. Columbia University, 2019, arxiv.org/pdf/1902.04704.pdf.

[6] Ng, Andrew. CS229 Lecture Notes. Stanford University, 2012, sgfin.github.io/files/notes/CS229_Lecture_Notes.pdf.

[7] Rocca, Joseph. A Gentle Journey from Linear Regression to Neural Networks. Towards Data Science, 13 July 2019, towardsdatascience.com/a-gentle-journey-from-linear-regression-to-neural-networks-68881590760e.

[8] Shree. "House Price Prediction." Kaggle, 26 Aug. 2018, www.kaggle.com/shree1992/housedata.

[9] Tegmark, Max. Life 3.0: Being Human in the Age of Artificial Intelligence. Penguin Books, 2018.

[10] Urtasun, Raquel, and Rich Zemel. CSC 411: Lecture 10: Neural Networks I. 2015, University of Toronto.

Appendix A Code

```
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.utils import shuffle
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn import preprocessing
from sklearn.model_selection import cross_val_score, KFold
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.wrappers.scikit_learn import KerasRegressor
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import matplotlib as mpl
```

```
mpl.use('tkagg')


# February 24, 2019

filename = 'filename.csv'
names0 = ['sqft_lot','bedrooms','bathrooms','sqft_living']
samples = np.array(pd.read_csv(filename, names=names0, skiprows=[0],
 ↪  header=None)).astype(float)

where_are_NaNs = np.isnan(samples) # replaces NaNs with zeros
samples[where_are_NaNs] = float(0)

names1 = ['price']
labels =  np.array(pd.read_csv(filename, names=names1, skiprows=[0],
 ↪  header=None)).astype(float)
where_are_NaNs = np.isnan(labels)
labels[where_are_NaNs] = float(0)
y = labels

min_max_scaler = preprocessing.MinMaxScaler()
X = min_max_scaler.fit_transform(samples.astype(np.float))

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
 ↪  random_state=1)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
 ↪  test_size=0.2, random_state=1)

# neural network model (fully/densely connected layers)
model = Sequential([
    Dense(units=3, input_shape=(samples.shape[1],), activation='relu'),
    Dense(units=1, activation='relu'),])
model.summary()

num_epochs = 400

# training
model.compile(optimizer=Adam(learning_rate=0.1), loss='mse', metrics=['mae',
 ↪  'mse'])

history = model.fit(X_train, y_train, epochs=num_epochs, batch_size=15,
 ↪  validation_data=(X_val, y_val), shuffle=True)


# PLOTTING:

loss_train = history.history['mae']
loss_val = history.history['val_mae']
rmse_final = np.sqrt(loss_train[-1])
print("Final Root Mean Squared Error = ", rmse_final)
```

```
epochs = np.arange(1,num_epochs+1)

plt.plot(epochs, loss_train, label='Training loss')
plt.plot(epochs, loss_val,label='Validation loss')

plt.title('Training and Validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss: Mean Squared Error [Dollars$^2$]')
plt.grid()
plt.legend()
plt.show()
```