Data science from scratch

Yasser Hifny

Data science from scratch

Yasser Hifny

وَمَنْ أَحْسَنُ قَوْلًا مِمَّنْ دَعَا إِلَى اللَّهِ وَعَمِلَ صَالِحًا
وَقَالَ إِنَّنِي مِنَ الْمُسْلِمِينَ

سورة فصلت الآية ٣٣

And who is better in speech than one who invites to Allah and does righteousness and says, "Indeed, I am of the Muslims."

Dedication.

For my father and mother

Contents

1 Optimization for Data Science 1

1.1 Derivative . . . 2
1.2 Gradient Descent . . . 4
1.2.1 Examples . . . 6
1.3 Gradient Descent using Taylor's Series . . . 8
1.4 Gradient Descent Limitations . . . 9
1.4.1 Adaptive learning Rate . . . 12
1.5 Assignment . . . 14

2 Input Representation 15

3 Linear Regression Networks 17

3.1 The model . . . 18
3.2 Learning problem . . . 19
3.2.1 Numerical solution . . . 22
3.3 Example . . . 22
3.4 Assignment . . . 24

4 Binary Classification Networks 25

4.1 The model 26
4.2 Learning problem 27
4.3 Classification decision 31
4.4 Example 32
4.5 Assignment 34

5 Multiclass Classification Networks 35

5.1 The model 36
5.2 Learning problem 38
5.3 Classification decision 40
5.4 Example 42
5.5 Assignment 43

6 Multilabel Classification Networks 44

6.1 Model 45
6.2 Learning problem 47
6.3 Example 47
6.4 Assignment 49

7 Deep Neural Networks 50

7.1 Motivation 51
7.2 Model 53
7.3 Learning via Backpropagation 55

7.3.1 Forward Propagation 55
7.3.2 Backward Propagation 55
7.4 Example 59
7.5 Assignment 61

8 Convolutional Networks 62

9 Recurrent Neural Networks 64

9.1 Model 65
9.2 Learning problem 68
9.3 The difficulty of training simple RNN 70
9.4 Long Short-Term Memory networks 74
9.4.1 Vanishing/Exploding Gradients with LSTMs 75
9.5 Example 76
9.6 Assignment 79

1. Optimization for Data Science

Calculus is one course you can come with to your parents and say, I am dropping it. And they'll understand.

— Bill Cosby

Gradient-based optimization is an essential tool for the field of data science. This chapter addresses the basic elements of calculus used to understand gradient-based methods such as gradient descent.

1.1 Derivative

The derivative in calculus is a way of measuring the rate of change of a function at a certain point. It can also be interpreted as the slope of the line that is tangent to the function's curve at that point. The derivative of a function $f(x)$ can be denoted by $f'(x)$ or $\frac{df(x)}{dx}$, where x is the input variable[1]. The derivative mathematically can be defined as follows:

$$f'(x) = \frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{1.1}$$

if the limit exists. This means that the function is not differentiable at those points and has no derivative there. A function is not differentiable at a point if it is not continuous at that point. For example, the function $f(x) = |x|$ is not continuous at $x = 0$, so it has no derivative there.

[1] A partial derivative is a derivative of a function of several variables with respect to one of those variables, while keeping the others constant. For example, if $f(x_1, x_2)$is a function of x_1 and x_2, then the partial derivative of f with respect to x_1 is denoted by $\frac{\partial f}{\partial x_1}$ and it is obtained by differentiating f with respect to x_1 and treating x_2 as a constant. Similarly, the partial derivative of f with respect to x_2 is denoted by $\frac{\partial f}{\partial x_2}$ and it is obtained by differentiating f with respect to x_2 and treating x_1 as a constant. Partial derivatives are used to measure the rate of change of a function along a specific direction or axis

Most data science algorithms are formulated as a minimization of a loss or objective function with respect to certain variables. To find a minimum of a function, there are different methods depending on the type and complexity of the function. Some of the common methods are:

- Sketching the function: This method involves plotting the graph of the function and visually identifying the lowest point on the graph. For example, the one-variable quadratic function has one global minimum[2] at $x = 1$:

$$f(x) = (x - 1)^2 \tag{1.2}$$

  where it is easy to find the minimum by inspecting the plot. However, this method is useful for simple functions that can be easily graphed, but it may not be accurate or feasible for more complicated functions.

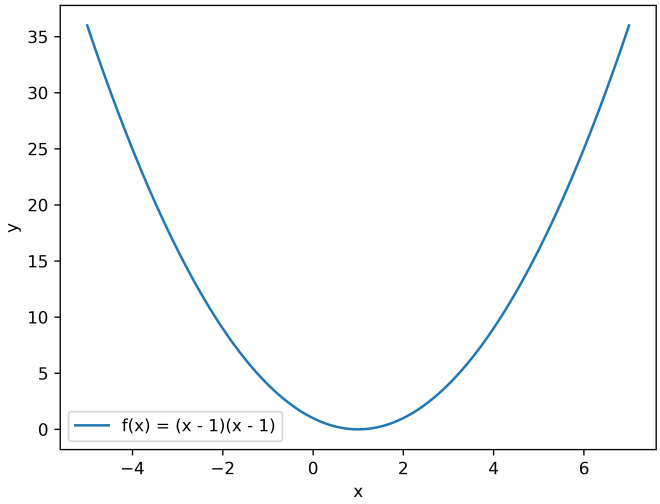


Figure 1.1: A plot of a simple quadratic function has a global minimum.

- Finding analytical solution: This method involves using calculus to find the derivative of the function and setting it equal to zero. This gives the critical points of the function, where the slope is zero or undefined. Figure 1.2 shows graphically why we set the first derivative to zero where the slope at the maximum and minimum is a horizontal line (i.e. the slope is zero).

[2]The difference between local and global minimum of a function is that a local minimum is the point where the function value is smaller than (or equal to) the function values at nearby points, while a global minimum is the point where the function value is the smallest among all points in the domain. A function can have multiple local minima, but only one global minimum.

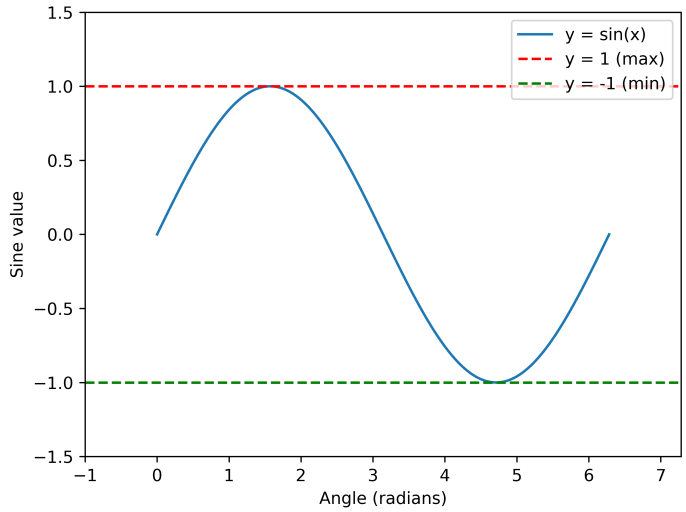


Figure 1.2: The slope at the maximum and minimum is a horizontal line (i.e. the slope is zero).

Then, using the second derivative test or the first derivative test, we can determine which critical points are local minima, local maxima, or neither. To find the minimum of the quadratic function in Equation (1.2), we differentiate it with respect to x and set the derivative to zero as follows (i.e. using the power rule):

$$f'(x) = 2(x - 1) = 0 \tag{1.3}$$

Hence, the minimum happens at $x = 1$. In practice, this method for finding the minimum of a function does not scale well with the amount of training data commonly seen in data science problems.

- Using gradient descent method: The gradient descent method is an iterative optimization algorithm that is used to find the minimum of a function by moving in the opposite direction of the gradient (or the slope) of the function at each point. This method will be detailed in the next section.

1.2 Gradient Descent

The gradient descent method involves starting from an initial guess and iteratively updating it by moving in the opposite direction of the gradient (the vector of partial derivatives) of the function. The gradient gives the direction of steepest ascent, so

moving against it will lead to a descent. The step size is determined by a learning rate parameter that controls how fast or slow the algorithm converges. This method is useful for finding a local minimum of a function that may not have an analytical solution or may be too complex to solve by calculus. However, this method does not guarantee finding the global minimum of the function, and it may depend on the choice of initial guess and learning rate.
The gradient descent method works as follows (assuming the function has one variable only):

1. Start with an initial guess $x = x^{(0)}$ for the parameters of the function that need to be optimized.

2. Calculate the gradient of the function with respect to the parameters at the current point $g(x) = \frac{df(x)}{dx}|_{x^{(0)}}$.

3. Update the parameters by subtracting a fraction of the gradient from the current values. The fraction is called the learning rate and it controls how big or small the steps are.
$$x^{(1)} = x^{(0)} - \eta g(x) \tag{1.4}$$
where $\eta > 0$ is the learning rate. When the gradient is ascending, it means that the function is increasing in that direction. Therefore, we move against the gradient direction to find a lower point on the function. This way, we hope to eventually reach a local minimum of the function. When descending, it means that the function is decreasing in that direction. Therefore, we move along the gradient direction to find a lower point on the function. This way, we hope to eventually reach a local minimum of the function as well. This behavior is shown in Figure 1.3. In the next section, we show mathematically why we need to subtract a fraction of the gradient from the current values.

4. Repeat steps 2 and 3 until the gradient is close to zero or a maximum number of iterations is reached.

The gradient descent method is widely used in machine learning and data science to train models by minimizing a loss function that measures the difference between the predicted and actual outputs. The gradient descent method can be applied to different types of functions, such as linear, quadratic, or non-linear functions. There are also different variants of the gradient descent method, such as batch gradient descent, stochastic gradient descent, mini-batch gradient descent, and momentum gradient descent, that differ in how they calculate and update the gradients.

•
http://www2.gcc.edu/dept/math/faculty/BancroftED/teaching/handouts/differentiation.pdf

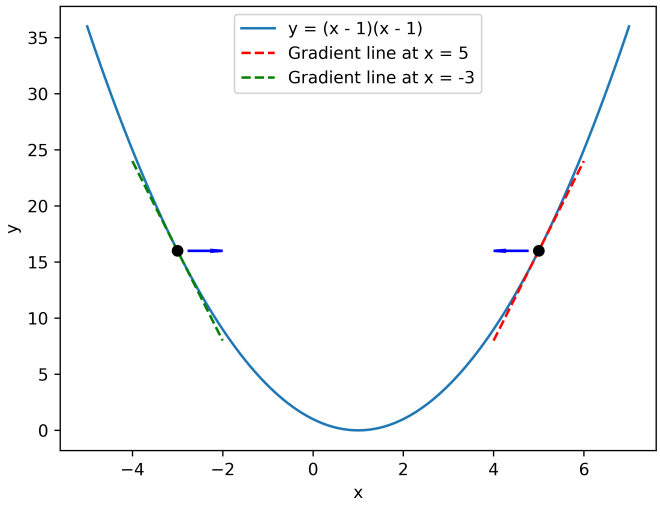


Figure 1.3: Function plot with gradient lines (ascending at $x = 5$ and descending at $x = -3$) and arrows pointing towards the minimum.

1.2.1 Examples

In this subsection, I will use Python code to illustrate how the gradient descent algorithm works and how it can be applied to different functions of one variable or two variables.

The gradient descent algorithm is a method to find the minimum of a function by taking small steps in the direction of the steepest decrease. To apply this algorithm to the function $f(x) = (x-1)^2$, we need to first find its derivative, which is $g(x) = 2(x-1)$. The algorithm starts with an initial guess for x =3, and computes the value of $g(x)$ at $x = 3$. Then it updates x by subtracting an $\eta g(x)$ from it. This gives a new value for x that is closer to the minimum of the function. The algorithm repeats this process until it converges to a value of x that makes $g(x)$ very close to zero or a maximum number of epochs is reached. This value of $x = 1$ is the minimum of the quadratic function $f(x) = (x-1)^2$. The described algorithm can be implemented as a Python code as follows:

```
def grad(x):
    return 2.0 * (x-1.0)

x = 3.0
eta = 0.001
epochs = 50000
for i in range(epochs):
```

```
    x -= eta * grad(x)

print(x)
```

Listing 1.1: Python example for finding the minimum of a quadratic function in one variable.

On the other hand, we can use two different learning rates η_{x1}, η_{x2} for the function $f(x1, x2) = (x1 - 2)^2 + 10 * (x2 + 3)^2$ because the function has different scales and curvatures along the $x1$ and $x2$ directions. If we use a single learning rate for both variables (e.g. $\eta_{x1} = \eta_{x2} = 0.1$), we might encounter aconvergence problem. By using different learning rates for each variable $\eta_{x1} = 0.1$ and $\eta_{x2} = 0.05$, we can adjust the step size according to the shape of the function and find the minimum more efficiently and accurately[3]. A Python implementation to find the minimum of this function is:

```
#Define the function of two variables
def f(x1, x2):
    return (x1 - 2)**2 + 10.0 * ((x2 + 3)**2)

#Define the partial derivatives of the function
def df_dx1(x1, x2):
    return 2 * (x1 - 2)

def df_dx2(x1, x2):
    return 20.0 * (x2 + 3)

#Define the learning rates for each variable
eta_x1 = 0.1 # Learning rate for x1
eta_x2 = 0.05 # Learning rate for x2

#Define the initial values for x1 and x2
x1 = 0.0
x2 = 0.0

#Define the tolerance for convergence
epsilon = 0.000001

#Define a variable to store the previous value of the function
prev_f = f(x1, x2)

#Start the gradient descent loop
steps = 0
while True:
    steps += 1
    #Update x1 and x2 using the gradient and the learning rates
    x1 = x1 - eta_x1 * df_dx1(x1, x2)
    x2 = x2 - eta_x2 * df_dx2(x1, x2)
```

[3] You can play with the learning rates to study the convergence properties.

```
33
34      #Compute the current value of the function
35      curr_f = f(x1, x2)
36
37      #Check if the function value has decreased sufficiently
38      if abs(curr_f - prev_f) < epsilon:
39          break # Exit the loop
40
41      #Update the previous value of the function
42      prev_f = curr_f
43
44  #Print the final values of x1 and x2 and the minimum value of the
        function
45  print("x1 =", x1)
46  print("x2 =", x2)
47  print("f(x1, x2) =", curr_f)
48  print("steps for convergence =", steps)
```

Listing 1.2: Python example for finding the minimum of a quadratic function in two variables.

The adaptive learning rate is a technique that adjusts the learning rate dynamically based on the progress of the gradient descent algorithm. The idea is to use a larger learning rate when the function is far from the minimum and a smaller learning rate when the function is close to the minimum. This way, we can speed up the convergence and avoid overshooting or oscillating. One method to implement the adaptive learning rate is to use the Hessian matrix, which is the matrix of second-order partial derivatives of the function. The Hessian matrix captures the curvature of the function and can be used to scale the gradient vector according to the shape of the function. By using the inverse of the Hessian matrix as a multiplier for the gradient vector, we can obtain a more accurate direction and step size for each iteration of the gradient descent algorithm. The next section will detail these techniques.

1.3 Gradient Descent using Taylor's Series

In mathematics, Taylor's series can be used to make a first-order approximation to a scalar loss function $f(\mathbf{x})$ around the current point vector $\mathbf{x}^{(t)} \in R^d$ given the first derivative of the function at that point:

$$f(\mathbf{x}^{(t+1)}) \approx f(\mathbf{x}^{(t)}) + (\mathbf{x}^{(t+1)} - \mathbf{x}^{(t)})^T \mathbf{g}, \tag{1.5}$$

where $\mathbf{g}$ is the gradient vector at the point $\mathbf{x}^{(t)}$. It can be written as well as follows:

$$f(\mathbf{x}^{(t)} + \Delta\mathbf{x}) \approx f(\mathbf{x}^{(t)}) + \Delta\mathbf{x}^T \mathbf{g}, \tag{1.6}$$

where $\Delta\mathbf{x} = \mathbf{x}^{(t+1)} - \mathbf{x}^{(t)}$. In order to decrease the loss function $f(\mathbf{x}^{(t)} + \Delta\mathbf{x})$, the term $\Delta\mathbf{x}^T\mathbf{g}$ has to be a negative value. Hence,

$$\Delta\mathbf{x}^T\mathbf{g} < 0 \tag{1.7}$$

Let's consider the cosine of angle between the two vectors $\Delta\mathbf{x}^T$ and $\mathbf{g}$:

$$\cos\theta = \frac{\Delta\mathbf{x}^T\mathbf{g}}{|\Delta\mathbf{x}^T||\mathbf{g}|} \tag{1.8}$$

$\cos\theta$ lies between -1 and 1 i.e. $-1 \leq \cos\theta \leq +1$. Hence,

$$-|\Delta\mathbf{x}^T||\mathbf{g}| \leq \Delta\mathbf{x}^T\mathbf{g} \leq |\Delta\mathbf{x}^T||\mathbf{g}| \tag{1.9}$$

Now we want the dot product to be as negative as possible (so that loss can be as low as possible). We can set the dot product to be $-|\Delta\mathbf{x}^T||\mathbf{g}|$ where $cos\theta$ has to be equal to -1 corresponds to $\theta = 180^o$. Therefore,

$$\Delta\mathbf{x} = -\mathbf{g} \tag{1.10}$$

This result explains why we move in the opposite direction of the gradient as we described in Section 1.2.
Intuitively, since the first-order approximation is good only for small $\Delta\mathbf{x}$, we want to choose a small $\eta > 0$ to make $\Delta\mathbf{x}$ small in magnitude. η is called the learning rate. Hence,

$$\Delta\mathbf{x} = -\eta\mathbf{g} \tag{1.11}$$

1.4 Gradient Descent Limitations

The gradient of a function at a specific point represents the direction of the steepest descent of the function at that point. In other words, it points in the direction in which the function decreases most rapidly. On the other hand, the contours[4] of a function are curves along which the function has the same value, so there is no change in the function value as you move along the contour.
Since the gradient points in the direction of maximum decrease, it is orthogonal (perpendicular) to the direction along which there is no change in the function value, which is the contour. This is true for any scalar function.
One limitation of the gradient descent is the zigzag effect. The zigzag effect occurs because the gradient at each point points in the direction of the steepest descent

[4]A contour of a function (e.g. Rosenbrock function) is a curve connecting points with the same function value. A contour is defined mathematically as the set of points (x, y) such that $f(x, y) = c$, where c is a constant.

when moving downhill and may not necessarily point directly toward the minimum. As the algorithm moves along the steepest slope, it overshoots the direction of the minimum, and in the next step, it must correct its course. This leads to a back-and-forth zigzagging pattern as the algorithm iteratively converges to the minimum as shown in Figure 1.4. This zigzag effect slows down the convergence of the algorithm.

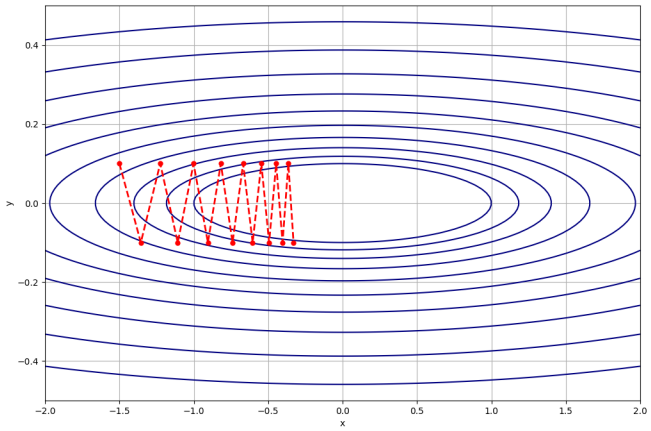


Figure 1.4: The zigzag effect of the gradient descent algorithm.

Zigzag effect of gradient descent does not happen for the loss functions that have circular contours or equal curvature in all dimensions or directions. For example, a function like $f(x, y) = x^2 + y^2$ has circular contours and does not have zigzag effect. A straight line to the minimum for these functions is followed by gradient descent as shown in Figure 1.5.

Hessian-based optimization methods, like Newton's method or Quasi-Newton methods (such as BFGS and L-BFGS), make use of second-order information to help guide the optimization process more effectively [1]. They use the Hessian matrix or its approximation to adjust the step size and direction, which can reduce the zigzagging effect and potentially lead to faster convergence. Using Taylor's expansion, the second-order approximation is given by

$$f(\mathbf{x}) \approx f(\mathbf{x}^{(t)}) + (\mathbf{x} - \mathbf{x}^{(t)})^T \mathbf{g} + \frac{1}{2}(\mathbf{x} - \mathbf{x}^{(t)})^T \mathbf{H}(\mathbf{x} - \mathbf{x}^{(t)}), \tag{1.12}$$

where $\mathbf{H}$ is the Hessian matrix at the point $\mathbf{x}^{(t)}$. The local Hessian matrix is a symmetric matrix and is defined by

$$\mathbf{H} \equiv \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}_i \partial \mathbf{x}_j}|_{\mathbf{x}^{(t)}} \tag{1.13}$$

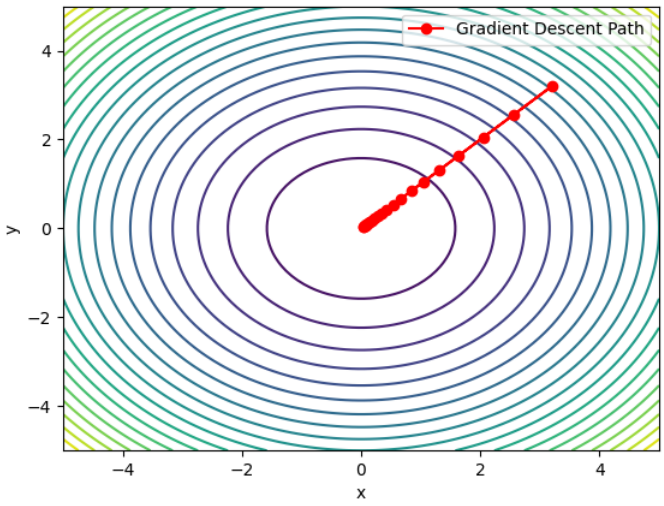


Figure 1.5: The zigzag effect of the gradient descent algorithm does not happen for the functions that have circular contours.

The basic idea of Newton's method is to minimize the quadratic approximation of the cost function $f(\mathbf{x})$ around the current point $\mathbf{x}^{(t)}$. Equation (1.12) can be rewritten as follows:

$$\Delta f(\mathbf{x}^{(t)}) = f(\mathbf{x}) - f(\mathbf{x}^{(t)}) \approx \Delta \mathbf{x}^T \mathbf{g} + \frac{1}{2} \Delta \mathbf{x}^T \mathbf{H} \Delta \mathbf{x} \tag{1.14}$$

Differentiating the above equation with respect $\Delta\mathbf{x}$ and setting the output to zero to get the minimum:

$$\mathbf{g} + \mathbf{H}\Delta\mathbf{x} = 0, \tag{1.15}$$

Hence, the Newton's update rule is given by

$$\Delta\mathbf{x} = -\eta \mathbf{H}^{-1} \mathbf{g} \tag{1.16}$$

When the matrix $\mathbf{H}$ equals to the identity matrix[5] (i.e. taking the same step in each direction), we reach the gradient descent update rule described in Equation (1.11). It's important to note that while Hessian-based methods can help overcome the zigzagging effect, they often come with their own challenges, such as increased computational complexity and the need to compute or approximate the Hessian matrix. In practice, these trade-offs need to be considered when selecting an optimization algorithm for a particular problem [2].

[5]An identity matrix is a square matrix in which all the elements of the principal diagonal are ones and all other elements are zero.

Adaptive learning rate methods overcome the zigzag effect in the gradient descent algorithm as well. They are addressed in the next subsection.

1.4.1 Adaptive learning Rate

Adaptive learning rate methods overcome the zigzag effect in gradient descent by adjusting the learning rate for each parameter during the optimization process. These methods take into account the history of gradients, the magnitude of the gradients, or both, to determine an appropriate learning rate for each parameter. As a result, adaptive learning rate methods can effectively navigate the loss surface and reduce oscillations and zigzagging.
Some popular adaptive learning rate methods include:

- **Momentum**: Momentum can reduce the zigzag problem by accumulating a vector that smooths out the gradient updates and aligns them with a consistent direction. Hence, it converges faster and more reliably than the gradient descent [3]. The momentum with gradient descent algorithm updates the parameters $\mathbf{x}$ as follows:

$$m^{(t)} = \beta m^{(t-1)} + (1-\beta) g^{(t)} \tag{1.17}$$

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \eta m^{(t)} \tag{1.18}$$

  where $g^{(t)}$ is the gradient of the loss function at time step t, $m^{(t)}$ is the vector that accumulates the past gradients, β is the momentum coefficient, and η is the learning rate.

- **AdaGrad**: AdaGrad accumulates the squared gradients for each parameter in a diagonal matrix and uses this information to adapt the learning rate for each parameter. Parameters with larger accumulated squared gradients have their learning rate reduced, while those with smaller accumulated squared gradients have their learning rate increased. This makes AdaGrad well-suited for problems with sparse gradients or features that occur with varying frequency [4]. The AdaGrad algorithm updates the variables $\mathbf{x}$ as follows:

$$v^{(t)} = v^{(t-1)} + g^{(t)}.g^{(t)} \tag{1.19}$$

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \frac{\eta}{\sqrt{v^{(t)}} + \epsilon} g^{(t)} \tag{1.20}$$

  where $g^{(t)}$ is the gradient of the loss function at time step t, $v^{(t)}$ is the sum of the squares of the gradients up to time step t, η is the learning rate, and ϵ is a small constant to prevent division by zero

- **RMSprop**: RMSprop is an improvement over Adagrad that uses an exponentially decaying average of the squared gradients instead of the cumulative sum. This makes RMSprop more robust to situations where the accumulated squared gradients can grow indefinitely, causing the learning rate to shrink too much [5]. The RMSprop algorithm updates the variables $\mathbf{x}$ as follows:

$$v^{(t)} = \beta v^{(t-1)} + (1-\beta) g^{(t)} . g^{(t)} \tag{1.21}$$

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \eta \frac{g^{(t)}}{\sqrt{v^{(t)}} + \epsilon} \tag{1.22}$$

  where $g^{(t)}$ is the gradient of the loss function at time step t, $v^{(t)}$ is the estimate of the second moment of the gradients, β is the decay rate for the moment, η is the learning rate, and ϵ is a small constant to prevent division by zero.

- **Adam**: Adam combines the ideas of RMSprop and momentum-based methods. It computes the first moment (mean) and the second moment (uncentered variance) of the gradients, and uses these moments to adapt the learning rate for each parameter. This helps Adam to navigate the loss surface more effectively, reducing zigzagging and achieving faster convergence [6]. The Adam algorithm updates the variables $\mathbf{x}$ as follows:

$$m^{(t)} = \beta_1 m^{(t-1)} + (1-\beta_1) g^{(t)} \tag{1.23}$$

$$v^{(t)} = \beta_2 v^{(t-1)} + (1-\beta_2) g^{(t)} . g^{(t)} \tag{1.24}$$

$$\hat{m}^{(t)} = \frac{m^{(t)}}{1-\beta_1^t} \tag{1.25}$$

$$\hat{v}^{(t)} = \frac{v^{(t)}}{1-\beta_2^t} \tag{1.26}$$

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \eta \frac{\hat{m}^{(t)}}{\sqrt{\hat{v}^{(t)}} + \epsilon} \tag{1.27}$$

  where $g^{(t)}$ is the gradient of the loss function at time step t, $m^{(t)}$ and $v^{(t)}$ are the estimates of the first and second moments of the gradients, $\hat{m}^{(t)}$ and $\hat{v}^{(t)}$ are the bias-corrected estimates[6], β_1 and β_2 are the decay rates for the

[6] The Adam algorithm suffers from a bias problem due to the initialization of the first and second moment estimates at zero. This means that the algorithm tends to underestimate the true values of these moments at the beginning of the training, leading to inaccurate gradient updates. To overcome this problem, the Adam algorithm uses a bias correction term that divides each moment estimate by a factor that accounts for the decay rates of the moments. This way, the algorithm can adjust for the bias and converge faster and more reliably.

moments, η is the learning rate, and ϵ is a small constant to prevent division by zero. Although the Adam algorithm is the state-of-the-art learning algorithm, the algorithms suffer from worse generalization[7] performance than stochastic gradient descent despite their faster training speed [7]. Hence, a practical recipe for training is to start with Adam for a few epochs and then switch to the gradient descent algorithm. In addition, the Adam algorithm accumulates two statistics for each variable or parameter during the optimization process.

1.5 Assignment

Using Adam algorithm, find the minimum of the function $f(x1, x2) = (x1 - 2)^2 + 10 * (x2 + 3)^2$.

[7] The generalization is measured using the test set performance.

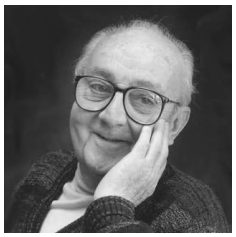
2. Input Representation

Your Content

Now, I am become Death, the destroyer of worlds.

— Robert Oppenheimer

3. Linear Regression Networks



All models are wrong, but some are useful.

— George Box

In this chapter, we will introduce one of the most fundamental and widely used methods in statistics, machine learning, and data science: linear regression. Linear regression is a technique that allows us to model the relationship between input variables (also called predictors or features) and output variables (also called responses or targets) using a linear function.

3.1 The model

Linear regression is a method of finding the best linear relationship between input variables and output variables as shown in Figure 3.1. The input variables may be float, binary, or integer values and the output variables must be **real** or **float** values[1]. For example, if the input has 3 dimensions or variables and the output has 2 dimensions, then the relation between the inputs and outputs in the model will look like this:

$$\begin{aligned} y_1 &= w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + b_1 \\ y_2 &= w_{21}x_1 + w_{22}x_2 + w_{23}x_3 + b_2 \end{aligned} \tag{3.1}$$

where each output is connected to all inputs as shown in Figure3.2. Hence, Equation (3.1) represents a fully connected or dense network. Equation (3.1) can be written in matrix form as well:

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{3.2}$$

[1]House price prediction is an example for predicting a float variable (i.e. house price) given the input features. The input features could be the number of rooms, area, number of bathrooms,... etc.

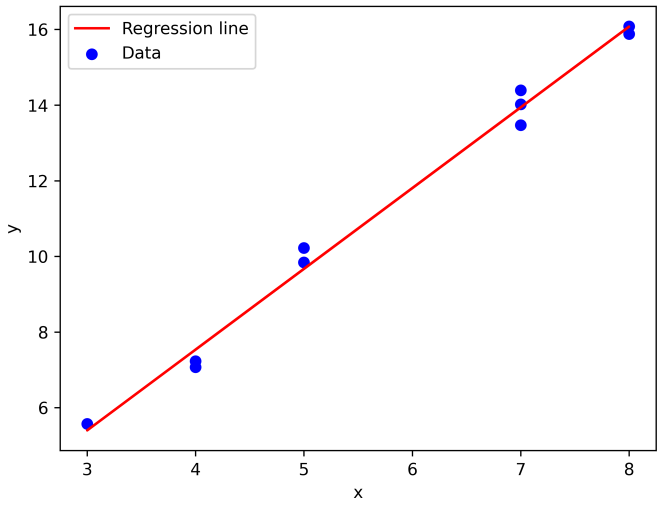


Figure 3.1: Linear function between one input variable and one output variable fitted using linear regression model.

Generally, linear regression single-layer network can be written as:

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b} \tag{3.3}$$

where $\mathbf{y} \in R^K$ is a vector of output variables, $\mathbf{x} \in R^d$ is a vector of input variables (each variable is a feature), $\mathbf{W} \in R^{KxD}$ and $\mathbf{b} \in R^K$ is a bias vector. The $\mathbf{W}$ and $\mathbf{b}$ are called parameters and they are estimated during the training phase using the training data.
Finding the optimal values for $\mathbf{W}$ and $\mathbf{b}$ using the training data is the subject of the next section.

3.2 Learning problem

Given a training data $(\mathbf{x}_1, \mathbf{t}_1), (\mathbf{x}_2, \mathbf{t}_2), \ldots, (\mathbf{x}_N, \mathbf{t}_N)$, the goal of the learning algorithm is to estimate the values of the $\mathbf{W}$ and $\mathbf{b}$ where $\mathbf{x} \in R^d$ and $\mathbf{t} \in R^K$. In supervised learning settings, we define an objective function to measure how close the $\mathbf{t}$ to its predicted value $\mathbf{y}$ over all the training data N. Concretely, we define E as follows:

$$E^n(\mathbf{W}, \mathbf{b}) = \frac{1}{2}||\mathbf{y}^n - \mathbf{t}^n|| = \frac{1}{2}\sum_{k=1}^{K}(y_k^n - t_k^n)^2 \tag{3.4}$$

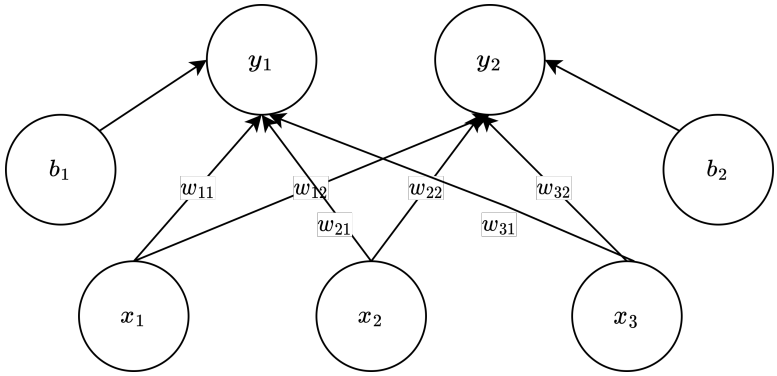


Figure 3.2: Linear regression network that has an input with 3 nodes or variables and the output has 2 nodes.

and

$$E(\mathbf{W}, \mathbf{b}) = \frac{1}{N} \sum_{n=1}^{N} E^n \tag{3.5}$$

where E is known as Mean Square Error (MSE) loss function and n is an index for the training sample $(\mathbf{x}_n, \mathbf{t}_n)$. Since $\mathbf{y}$ is a float vector, MSE loss function is a suitable objective function for linear regression. In order to estimate the values of $\mathbf{W}$ and $\mathbf{b}$ parameters, we minimize the loss function with respect to the parameters. The loss function measures how well the linear model fits the data, and the parameters are the coefficients and biases of the linear model. By minimizing the loss function, we can find the optimal values of the parameters that make the best predictions for the output variable.

As described in Chapter 1, linear models with MSE loss function (i.e quadratic loss function) has a unique solution. Moreover, it can be found analytically. Let's assume a multiple regression problem where we have multiple input variables and one output variable. In matrix form, the loss function is given by

$$E(\theta) = \frac{1}{2N}(X\theta - t)^T(X\theta - t) \tag{3.6}$$

where t is the output matrix of size $N \times 1$, X is the input matrix of size $N \times d + 1$ where we add the bias as an extra feature that has a value $= 1$, $\theta = \{\mathbf{W}, \mathbf{b}\}$ is the parameters matrix of size $d + 1 \times 1$. Ignoring the constant term $\frac{1}{2N}$ since it does not affect the optimization results and simplifying the cost function:

$$E(\theta) = \theta^T X^T X\theta - \theta^T X^T t - t^T X\theta + t^T t \tag{3.7}$$

Since $\theta^T X^T t = t^T X\theta =$ scalar,

$$E(\theta) = \theta^T X^T X\theta - 2t^T X\theta + t^T t \tag{3.8}$$

To find the optimal value of θ, we compute the first derivative of the cost function with respect to the parameters and set it to zero.

$$\frac{\partial E(\theta)}{\partial \theta} = 2X^T X\theta - 2X^T t = 0 \tag{3.9}$$

It can be simplified

$$X^T X\theta = X^T t \tag{3.10}$$

Hence, the analytical solution is given by

$$\theta = (X^T X)^{-1} X^T t \tag{3.11}$$

However, finding the values of the parameters $\mathbf{W}$ and $\mathbf{b}$ using the analytical solution does not scale well with the amount of training data. Hence, we will focus on the gradient descent solution in this chapter. The gradient of MSE loss function with respect to the parameters $\mathbf{W}$ and $\mathbf{b}$ can be computed as follows:

$$\frac{\partial E^n(\mathbf{W},\mathbf{b})}{\partial w_{rs}} = \frac{\partial E^n(\mathbf{W},\mathbf{b})}{\partial y_r}\frac{\partial y_r}{\partial w_{rs}} = (y_r^n - t_r^n)x_s^n \tag{3.12}$$

where w_{rs} is the element (r, s) of the matrix $\mathbf{W}$. Hence,

$$\frac{\partial E(\mathbf{W},\mathbf{b})}{\partial w_{rs}} = \frac{1}{N}\sum_{n=1}^{N}(y_r^n - t_r^n)x_s^n \tag{3.13}$$

and the gradient of the loss function with respect to the bias variable b_r is given by

$$\frac{\partial E(\mathbf{W},\mathbf{b})}{\partial b_r} = \frac{1}{N}\sum_{n=1}^{N}(y_r^n - t_r^n) \tag{3.14}$$

Using the gradient descent, the matrix of the weights and the bias vector can be updated as follows:

$$\begin{aligned} w_{rs}^{t+1} &= w_{rs}^t - \eta\frac{\partial E(\mathbf{W},\mathbf{b})}{\partial w_{rs}} \\ b_r^{t+1} &= b_r^t - \eta\frac{\partial E(\mathbf{W},\mathbf{b})}{\partial b_r} \end{aligned} \tag{3.15}$$

where η is the learning rate. The algorithm can update the parameters after the gradient over the whole training set is accumulated. To scale the problem to a large amount of training data, we use a variant called mini-batch stochastic gradient to estimate the parameters of the linear regression model. This algorithm will be described in the next subsection.

3.2.1 Numerical solution

Mini-batch stochastic gradient descent is a variation of gradient descent that updates the parameters of the linear regression model using a subset of the data (called a mini-batch) at each iteration. It is a numerical solution to find the global minimum of the quadratic loss function with respect to the parameters.
The idea is to reduce the computational cost and memory usage of gradient descent, while still achieving a good convergence rate.
The algorithm works as follows:

- Initialize the parameters $\mathbf{W}$ and $\mathbf{b}$ randomly or with zeros.
- Divide the data into small batches of equal size (for example, 32 or 64).
- Repeat until convergence or a maximum number of epochs:
  - For each batch:
    - Compute the predictions of the linear model for the batch.
    - Use Equation (5.7) to compute the loss function for the batch (i.e. mean squared error) .
    - Compute the gradients of the loss function with respect to the parameters for the batch using Equation (3.13) and Equation (3.14).
    - Update the parameters using Equation (4.13).
  - Return the final parameters.

The advantages of mini-batch gradient descent are:

- It can handle large datasets that do not fit in memory.
- It can exploit parallelism and vectorization to speed up computations

but it requires tuning the batch size and the learning rate hyperparameters.

3.3 Example

To illustrate the linear regression algorithm, Table 3.3 with 10 rows of random training data was created. The first three columns represent input features (x_1, x_2, x_3) and the last two columns represent output targets (t_1, t_2):

```
1
2 import numpy as np
3 from sklearn.metrics import mean_squared_error
4
5 # Training data
6 X = np.array([
```

Table 3.1: Synthesized data for linear regression modeling. The input variables are 3 and the output variables are 2 and they are float variables.

x_1	x_2	x_3	t_1	t_2
0.23	0.87	0.95	2.74	1.82
0.34	0.56	0.29	1.42	2.61
0.98	0.68	0.05	3.12	2.49
0.51	0.24	0.64	1.63	1.95
0.42	0.75	0.49	2.84	2.37
0.89	0.11	0.93	2.17	1.68
0.14	0.91	0.25	1.96	2.55
0.76	0.41	0.53	2.29	2.24
0.66	0.04	0.08	1.27	1.35
0.45	0.82	0.76	2.86	2.74

```
    [0.23, 0.87, 0.95],
    [0.34, 0.56, 0.29],
    [0.98, 0.68, 0.05],
    [0.51, 0.24, 0.64],
    [0.42, 0.75, 0.49],
    [0.89, 0.11, 0.93],
    [0.14, 0.91, 0.25],
    [0.76, 0.41, 0.53],
    [0.66, 0.04, 0.08],
    [0.45, 0.82, 0.76],
])
T = np.array([
    [2.74, 1.82],
    [1.42, 2.61],
    [3.12, 2.49],
    [1.63, 1.95],
    [2.84, 2.37],
    [2.17, 1.68],
    [1.96, 2.55],
    [2.29, 2.24],
    [1.27, 1.35],
    [2.86, 2.74],
])

# Analytical solution
def analytical_solution(X, T):
    X_b = np.c_[np.ones((X.shape[0], 1)), X]
    theta = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(T)
    return theta

theta_analytical = analytical_solution(X, T)
```

```
38 print("Analytical solution coefficients:", theta_analytical)
39
40 # Gradient descent
41 def gradient_descent(X, T, eta=0.1, iterations=100000):
42     m, n = X.shape
43     theta = np.random.randn(n + 1, T.shape[1])
44     X_b = np.c_[np.ones((m, 1)), X]
45     for i in range(iterations):
46         gradients = 2 / m * X_b.T.dot(X_b.dot(theta) - T)
47         theta -= eta * gradients
48     return theta
49
50 theta_gradient_descent = gradient_descent(X, T)
51 print("Gradient descent coefficients:", theta_gradient_descent)
52
53 # Comparing results
54 Y_analytical = np.c_[np.ones((X.shape[0], 1)), X].dot(
       theta_analytical)
55 Y_gradient_descent = np.c_[np.ones((X.shape[0], 1)), X].dot(
       theta_gradient_descent)
56
57 mse_analytical = mean_squared_error(T, Y_analytical)
58 mse_gradient_descent = mean_squared_error(T, Y_gradient_descent)
59
60 print("Mean Squared Error - Analytical Solution:", mse_analytical)
61 print("Mean Squared Error - Gradient Descent:", mse_gradient_descent
       )
```

Listing 3.1: Python example for finding the linear regression parameters using the gradient descent algorithm. The analytical solution is provided as well.

Since linear regression has a unique solution, both analytical and gradient descent methods have the same result.

3.4 Assignment

Predict house price using the Keras deep learning library and Google colab. To load the dataset, use the method: tf.keras.datasets.boston_housing.load_data.

4. Binary Classification Networks

Logistic regression is a powerful tool for modeling the relationship between a categorical response variable and some explanatory variables. It is especially useful when the response variable has only two possible outcomes, such as success/failure, yes/no, or healthy/sick.

— Alan Agresti

In this chapter, we will introduce a probabilistic binary classification model[1] that estimates the probability of an outcome that can only be one of two values, such as yes or no, based on one or more predictor variables. Hence, it can be used to make a binary classifier by choosing a threshold value and classifying inputs with probability greater than the threshold as one class, and below the threshold as the other class

4.1 The model

Logistic regression is a classification method that is used to predict the relationship between a binary dependent variable and one or more independent variables. As shown in Figure 4.1, the network has one output variable and many input variables. The input variables may have float, binary, or integer values, and the output **probability variable** is a float bounded between 0.0 and 1.0. For example, if the input has 3 dimensions or variables and only one output variable, then the relation between the inputs and the output in the model will look like this:

$$z = w_1 x_1 + w_2 x_2 + w_3 x_3 + b \tag{4.1}$$

[1] Also known as logistic regression in the literature.

where the intermediate z is connected to all inputs and has a float value. Hence, Equation (4.1) represents a fully connected or dense network. It can be written in matrix form as well:

$$z = \begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + b \tag{4.2}$$

Generally, logistic regression single-layer network can be written as:

$$z = \mathbf{w}^T \mathbf{x} + b \tag{4.3}$$

where $z \in R$ is a variable, $\mathbf{x} \in R^d$ is a vector of input variables (each variable is a feature), $\mathbf{w} \in R^d$ and b is a bias. The $\mathbf{w}$ and b are called parameters and they are estimated during the training phase using the training data.
In addition, the output y is computed in Equation (4.4) using a sigmoid transformation or activation function. It is used to map z to a float bounded between 0.0 and 1.0 as shown in Figure 4.2[2]. Naturally, the binary activation function can transform the input to output that has a value 0 or 1 as shown in Figure 4.3. However, this function is not continuous and not differentiable at 0. This explains why sigmoid was selected as a transformation method for binary classification. The sigmoid activation function is a continuous and differentiable function. Hence, we can compute the gradient of a loss function based on that activation function.

$$y = \frac{1}{1 + e^{-z}} \tag{4.4}$$

Finding the optimal values for $\mathbf{w}$ and b using the training data is the subject of the next section.

4.2 Learning problem

Given a training data $(\mathbf{x}_1, t_1), (\mathbf{x}_2, t_2), \ldots, (\mathbf{x}_N, t_N)$, the goal of the learning algorithm is to estimate the values of the $\mathbf{w}$ and b where $\mathbf{x} \in R^d$ and $t \in \{1, 0\}$. We define an objective function to measure how close the t to its predicted value y over all the training data N. Concretely, we define E as follows:

$$E^n(\mathbf{w}, b) = -(t^n \log p^n + (1 - t^n) \log(1 - p^n)) \tag{4.5}$$

and

$$E(\mathbf{w}, b) = \frac{1}{N} \sum_{n=1}^{N} E^n \tag{4.6}$$

[2]It is so called because its graph is "S-shaped".

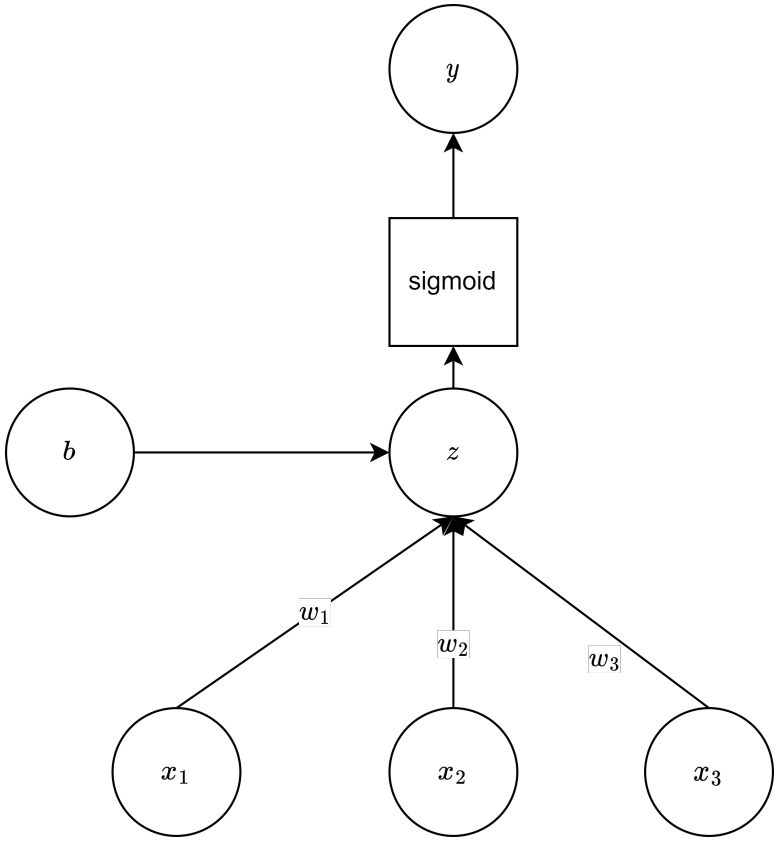


Figure 4.1: A logistic regression model (i.e. binary classification model) must have only one output variable and optional count of input variables.

where E is known as the binary cross-entropy (BCE) loss function and n is an index for the training sample $(\mathbf{x}_n, t_n)$. The loss function is plotted in Figure 4.4. The loss increases exponentially as the predicted probability of the true class gets closer to zero. Since y is a float value between 0 and 1, the BCE loss function is a matching objective function for logistic regression. In order to estimate the values of $\mathbf{w}$ and b parameters, we minimize the loss function with respect to the parameters.
The logistic regression model with BCE loss function has a unique solution[3]. The gradient of BCE loss function with respect to the parameters $\mathbf{w}$ and b can be computed as follows:

$$\frac{\partial E^n(\mathbf{w}, b)}{\partial w_i} = \frac{\partial E^n(\mathbf{w}, b)}{\partial y^n} \frac{\partial y^n}{\partial z^n} \frac{\partial z^n}{\partial w_i} \tag{4.7}$$

where w_i is the element i of the vector $\mathbf{w}$. The three terms can be computed as

[3]It can not be found analytically.

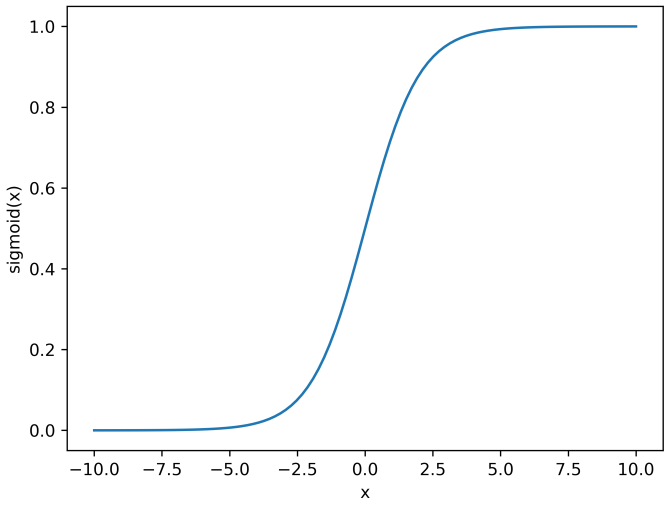


Figure 4.2: Sigmoid activation function.

follows:

$$\begin{aligned}\frac{\partial E^n(\mathbf{w}, b)}{\partial y^n} &= -\frac{t^n}{y^n} + \frac{1-t^n}{1-y^n} \\ &= \frac{-(1-y^n)t + y^n(1-t^n)}{y^n(1-y^n)} \\ &= \frac{(y^n - t^n)}{y^n(1-y^n)}\end{aligned} \tag{4.8}$$

Using Equation 4.4, the gradient of the output y^n with respect to z^n:

$$\begin{aligned}\frac{\partial y^n}{\partial z^n} &= \frac{0 - e^{-z^n}(-1)}{(1+e^{-z^n})^2} \\ &= \frac{1}{(1+e^{-z^n})} \cdot \frac{e^{-z^n}}{(1+e^{-z^n})} \\ &= y^n \cdot \frac{e^{-z^n} + 1 - 1}{(1+e^{-z^n})} \\ &= y^n(1 - \frac{1}{(1+e^{-z^n})}) \\ &= y^n(1-y^n)\end{aligned} \tag{4.9}$$

And $\frac{\partial z^n}{\partial w_i}$ is given by

$$\frac{\partial z^n}{\partial w_i} = x_i^n \tag{4.10}$$

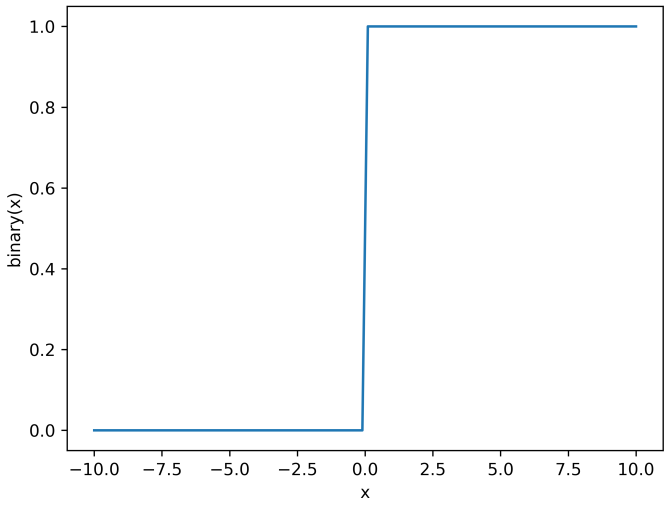


Figure 4.3: Binary activation function.

Hence, Equation (4.7) can be written using the three terms computed above as follows:

$$\begin{aligned}\frac{\partial E^n(\mathbf{w}, b)}{\partial w_i} &= \frac{(y^n - t^n)}{y^n(1 - y^n)} y^n(1 - y^n) x_i^n \\ &= (y^n - t^n) x_i\end{aligned} \tag{4.11}$$

and the gradient of the loss function with respect to the bias variable b is given by

$$\frac{\partial E^n(\mathbf{w}, b)}{\partial b} = (y^n - t^n) \tag{4.12}$$

Using the gradient descent, the vector of the weights and the bias term can be updated as follows:

$$\begin{aligned}w_i^{t+1} &= w_i^t - \eta \frac{1}{N} \sum_{n=1}^{N} (y^n - t^n) x_i^n \\ b^{t+1} &= b^t - \eta \frac{1}{N} \sum_{n=1}^{N} (y^n - t^n)\end{aligned} \tag{4.13}$$

where η is the learning rate. Using mini-batch stochastic gradient descent, it is possible to learn the parameters efficiently for large datasets. The algorithm is very similar to the one described in section 3.2.1.

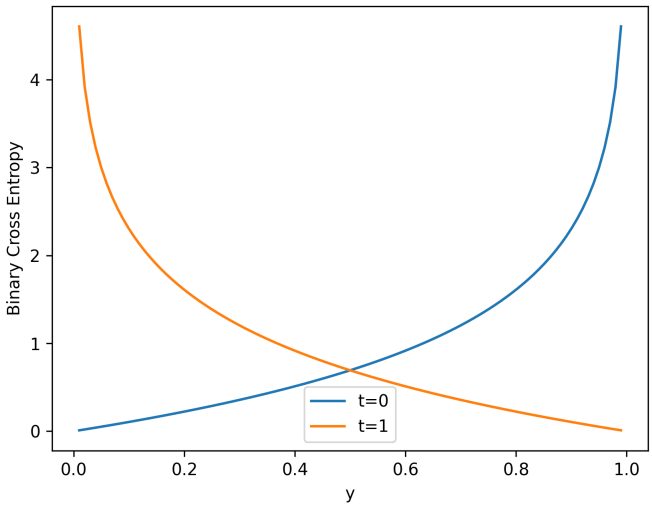


Figure 4.4: Binary cross-entropy objective function.

4.3 Classification decision

The output of the learning algorithm is to estimate the values of the $\mathbf{w}$ and b parameters. Hence, they can be used at the test time for prediction. Since y is a float value between 0 and 1, it can not be used directly to classify the input samples to 0 or 1. To overcome this problem, we define a threshold for decision

$$\hat{y} = \begin{cases} 1 & \text{if } y \geq 0.5 \\ 0 & \text{if } y < 0.5 \end{cases}$$

where $\hat{y}$ is the final decision either 0 or 1. Logistic regression separates the two classes with a linear decision boundary. The linear decision boundary of logistic regression is the set of all points $\mathbf{x}$ that satisfy:

$$P(y = 1|\mathbf{x}) = P(y = 0|\mathbf{x}) = \frac{1}{1 + e^{-z}} = \frac{1}{2} \tag{4.14}$$

then

$$z = w_1 x_1 + w_2 x_2 + \cdots + w_d x_d + b = 0 \tag{4.15}$$

For two-dimensional data

$$w_1 x_1 + w_2 x_2 + b = 0 \tag{4.16}$$

Hence,

$$x_2 = -\frac{b}{w_2} - \frac{w_1}{w_2}x_1 \tag{4.17}$$

For example, the two-dimensional AND gate[4] is shown in Table 4.4. The first two columns represent input features (x_1, x_2) and the last column columns represent output targets t. The decision boundary is shown in Figure 4.5.

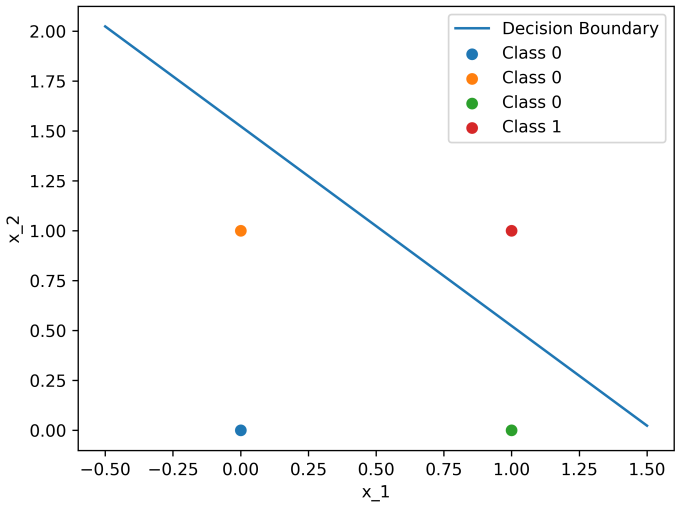


Figure 4.5: The decision boundary for the two-dimensional "AND" gate.

4.4 Example

A Python code is provided to illustrate the logistic regression learning algorithm and plot the decision boundary:

```
1
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  def sigmoid(z):
6      return 1 / (1 + np.exp(-z))
7
8  def predict(X, w):
9      return sigmoid(np.dot(X, w))
10
```

[4]The AND gate is known in logic design literature.

Table 4.1: Two-dimensional AND gate truth table.

x_1	x_2	t
0	0	0
0	1	0
1	0	0
1	1	1

```
11 def cost_function(X, t, w):
12     N = len(t)
13     y = predict(X, w)
14     E = -1/N * (np.dot(t.T, np.log(y)) + np.dot((1-t).T, np.log(1-y)
       ))
15     return E
16
17 def gradient_descent(X, t, w, alpha, iterations):
18     m = len(t)
19     E_history = []
20     for i in range(iterations):
21         y = predict(X, w)
22         w = w - alpha * (1/m) * np.dot(X.T, (y-t))
23         E_history.append(cost_function(X, t, w))
24     return w, E_history
25
26 X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
27 t = np.array([[0], [0], [0], [1]])
28
29 m = len(t)
30 X = np.hstack((np.ones((m, 1)), X))
31
32 n = X.shape[1]
33 w = np.zeros((n, 1))
34
35 alpha = 0.01
36 iterations = 100000
37
38 w_final, E_history = gradient_descent(X, t, w, alpha, iterations)
39
40 print (predict(X, w_final))
41
42 x_1 = np.linspace(-0.5, 1.5)
43 x_2 = -(w_final[0] + w_final[1]*x_1) / w_final[2]
44
45 plt.plot(x_1,x_2,label='Decision Boundary')
46 plt.scatter(0,0,label='Class 0')
47 plt.scatter(0,1,label='Class 0')
48 plt.scatter(1,0,label='Class 0')
49 plt.scatter(1,1,label='Class 1')
```

```
plt.xlabel('x_1')
plt.ylabel('x_2')
plt.legend()
plt.savefig('decision_boundary_and.png')
```

Listing 4.1: Python example for plotting the logistic regression decision boundary for AND Problem.

Since logistic regression has a unique solution, running the script several times will lead to the same result.

4.5 Assignment

Spam email detection (binary classification task) using the Keras deep learning library and Google colab. To load the dataset, visit https://archive.ics.uci.edu/ml/datasets/spambase.

5. Multiclass Classification Networks

When we make inferences based on incomplete information, we should draw them from that probability distribution that has the maximum entropy permitted by the information we do have.

— E. T. Jaynes

In this chapter, we will introduce the probabilistic multiclass or multinomial classification algorithm[1]. It aims to classify the input samples into one of three or more classes (for two classes only see Chapter 4 of binary classification).

5.1 The model

A multiclass softmax classifier is a supervised learning algorithm that can handle multiple classes. It assigns a probability to each class based on the input features and the learned parameters. The input variables may be float, binary, or integer values and the output variables must be categorical variables or class labels. Categorical variables or class labels in multiclass problem setting are usually encoded using one-hot vectors. A one-hot vector is a vector that has only one element with a value of 1 and the rest are 0. For example, if there are three classes A, B, C and D, we can use the following one-hot vectors to represent them:

A: [1, 0, 0]
B: [0, 1, 0]
C: [0, 0, 1]

The advantage of using one-hot vectors is that they can be easily used with models that output probabilities for each class.

[1] It is known as the softmax [8] or maximum entropy classifier [9]

If the input has 4 dimensions or variables and the output has 3 classes, then the relation between the inputs and outputs in the model will look like this:

$$\begin{aligned} z_1 &= w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + w_{14}x_4 + b_1 \\ z_2 &= w_{21}x_1 + w_{22}x_2 + w_{23}x_3 + w_{24}x_4 + b_2 \\ z_3 &= w_{31}x_1 + w_{32}x_2 + w_{33}x_3 + w_{34}x_4 + b_3 \end{aligned} \tag{5.1}$$

where each output is connected to all inputs as shown in Figure 5.1. Hence, Equation (5.1) represents a fully connected or dense network. Equation (5.1) can be written in matrix form as well:

$$\begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{5.2}$$

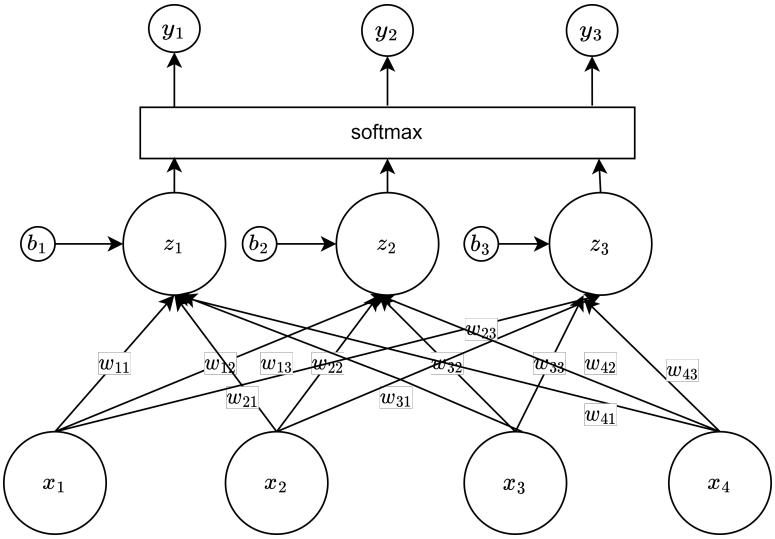


Figure 5.1: Multiclass classification network that has an input with 4 nodes or variables and the output has 3 nodes or classes.

Generally, softmax regression single-layer network can be written as:

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b} \tag{5.3}$$

$$\mathbf{y} = \text{softmax}(\mathbf{z}) \tag{5.4}$$

and the elements of the softmax y_i are given by

$$y_i = \frac{e^{z_i}}{\sum_{c=1}^{K} e^{z_c}} \text{ for } i = 1, \ldots, K \tag{5.5}$$

where $\mathbf{y} \in R^K$ is a vector of output classes or variables (the softmax transform is a normalized function, meaning that all elements of the output vector $\mathbf{y}$ are in the range (0, 1) and sum up to 1), $\mathbf{x} \in R^d$ is a vector of input variables (each variable is a feature), $\mathbf{W} \in R^{Kxd}$ and $\mathbf{b} \in R^K$ is a bias vector. The $\mathbf{W}$ and $\mathbf{b}$ are called parameters and they are estimated during the training phase using the training data.
Finding the optimal values for $\mathbf{W}$ and $\mathbf{b}$ using the training data is the subject of the next section.

5.2 Learning problem

Given a training data $(\mathbf{x}_1, \mathbf{t}_1), (\mathbf{x}_2, \mathbf{t}_2), \ldots, (\mathbf{x}_N, \mathbf{t}_N)$, the goal of the learning algorithm is to estimate the values of the $\mathbf{W}$ and $\mathbf{b}$ where $\mathbf{x} \in R^d$ and $\mathbf{t}$ is a one-hot vector of length K can be mathematically denoted as an element of the set $0, 1^K$ that has exactly one element equal to 1 and the rest equal to 0. In supervised learning settings, we define an objective function to measure how close the $\mathbf{t}$ to its predicted value $\mathbf{y}$ over all the training data N. Concretely, we define E as follows:

$$E^n(\mathbf{W}, \mathbf{b}) = -\sum_{k=1}^{K} t_k^n \log y_k^n \tag{5.6}$$

and

$$E(\mathbf{W}, \mathbf{b}) = \frac{1}{N} \sum_{n=1}^{N} E^n \tag{5.7}$$

where E is known as categorical cross-entropy loss function and n is an index for the training sample $(\mathbf{x}_n, \mathbf{t}_n)$. Since $\mathbf{y}$ is a float vector (its elements float between 0 and 1), the categorical cross-entropy loss function is a suitable objective function for softmax regression. In order to estimate the values of $\mathbf{W}$ and $\mathbf{b}$ parameters, we minimize the loss function with respect to the parameters. The loss function measures how well the softmax model fits the data, and the parameters are the coefficients and biases of the model. By minimizing the loss function, we can find the optimal values of the parameters that make the best predictions for the output classed. The softmax regression model with the categorical cross-entropy loss function has a unique solution[2].
In order to derive the gradient of the objective function with respect to the parameters of the softmax classifier, we will need the derivative of the softmax function with respect to its inputs. The gradient of softmax with respect to its inputs is a matrix

[2] It can not be found analytically.

known as the Jacobian matrix[3]. It is KxK matrix and it is given by:

$$\begin{pmatrix} \frac{\partial y_1}{\partial z_1} & \frac{\partial y_1}{\partial z_2} & \frac{\partial y_1}{\partial z_3} & \cdots & \frac{\partial y_1}{\partial z_K} \\ \frac{\partial y_2}{\partial z_1} & \frac{\partial y_2}{\partial z_2} & \frac{\partial y_2}{\partial z_3} & \cdots & \frac{\partial y_2}{\partial z_K} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_K}{\partial z_1} & \frac{\partial y_K}{\partial z_2} & \frac{\partial y_K}{\partial z_3} & \cdots & \frac{\partial y_K}{\partial z_K} \end{pmatrix} \tag{5.8}$$

When $k = i$ (i.e. diagonal elements of the matrix):

$$\begin{aligned} \frac{\partial y_k}{\partial z_i} &= \frac{e^{z_i} \sum_j e^{z_j} - e^{z_k} e^{z_i}}{(\sum_j e^{z_j})^2} \\ &= \frac{e^{z_i}}{\sum_j e^{z_j}} \frac{\sum_j e^{z_j} - e^{z_k}}{\sum_j e^{z_j}} \\ &= y_i(1 - y_k) \\ &= y_i - y_i y_k \end{aligned} \tag{5.9}$$

and when $k \neq i$:

$$\frac{\partial y_k}{\partial z_i} = \frac{-e^{z_i} e^{z_k}}{(\sum_j e^{z_j})^2} = -\frac{e^{z_i}}{\sum_j e^{z_j}} \frac{e^{z_k}}{\sum_j e^{z_j}} = -y_i y_k \tag{5.10}$$

Hence, the two equations can be combined into one equation:

$$\frac{\partial y_k}{\partial z_i} = y_i(\delta_{ik} - y_k) \tag{5.11}$$

where Kronecker delta δ_{ik} is defined as follows:

$$\delta_{ik} = \begin{cases} 0 \text{ when } i \neq k \\ 1 \text{ when } i = k \end{cases} \tag{5.12}$$

It can be written using two separate matrices:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{z}} = \begin{pmatrix} y_1 & 0 & 0 & \cdots & 0 \\ 0 & y_2 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & y_K \end{pmatrix} - \begin{pmatrix} y_1 y_1 & y_1 y_2 & y_1 y_3 & \cdots & y_1 y_K \\ y_2 y_1 & y_2 y_2 & y_2 y_3 & \cdots & y_2 y_K \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ y_K y_1 & y_K y_2 & y_K y_3 & \cdots & y_K y_K \end{pmatrix} \tag{5.13}$$

Using Equation (5.11), the

$$E^n(\mathbf{W}, \mathbf{b}) = -\sum_{k=1}^{K} t_k^n \log y_k^n \tag{5.14}$$

[3]The softmax function is a vector and differentiating a vector with respect to a vector of parameters generates a matrix.

$$\begin{aligned}
\frac{\partial E(\mathbf{W},\mathbf{b})}{\partial w_{ij}} &= -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K}\frac{\partial E^n(\mathbf{W},\mathbf{b})}{\partial y_k^n}\frac{\partial y_k^n}{\partial z_i^n}\frac{\partial z_i^n}{\partial w_{ij}} \\
&= -\frac{1}{N}\sum_{n=1}^{N}\left(\sum_{k=1}^{K}\frac{t_k^n}{y_k^n}y_i^n(\delta_{ik}-y_k^n)\right)x_j^n \\
&= \frac{1}{N}\sum_{n=1}^{N}\left(\sum_{k=1}^{K}\frac{t_k^n}{y_k^n}y_i^n(y_k^n-\delta_{ik})\right)x_j^n \\
&= \frac{1}{N}\sum_{n=1}^{N}\left(y_i^n\sum_{k=1}^{K}t_k^n - y_i^n\sum_{k=1}^{K}\frac{t_k^n}{y_k^n}\delta_{ik}\right)x_j^n \\
&= \frac{1}{N}\sum_{n=1}^{N}\left(y_i^n - t_i^n\right)x_j^n,
\end{aligned} \tag{5.15}$$

where $\sum_{k=1}^{K} t_k^n = 1.0$. Similarly, the gradient of the cross-entropy loss function with respect to the bias b_i:

$$\frac{\partial E(\mathbf{W},\mathbf{b})}{\partial b_i} = \frac{1}{N}\sum_{n=1}^{N}\left(y_i^n - t_i^n\right) \tag{5.16}$$

Using the gradient descent, the vector of the weights and the bias term can be updated as follows:

$$\begin{aligned}
w_{ij}^{t+1} &= w_{ij}^t - \eta\frac{1}{N}\sum_{n=1}^{N}(y_i^n - t_i^n)x_j^n \\
b_i^{t+1} &= b_i^t - \eta\frac{1}{N}\sum_{n=1}^{N}(y_i^n - t_i^n)
\end{aligned} \tag{5.17}$$

where η is the learning rate. Using mini-batch stochastic gradient descent, it is possible to learn the parameters efficiently for large datasets. The algorithm is very similar to the one described in section 3.2.1.

5.3 Classification decision

The output of the learning algorithm is to estimate the values of the $\mathbf{W}$ and $\mathbf{b}$ parameters. Hence, they can be used at the test time for prediction. The class with the highest probability is the predicted class. To find this winner class for a given

sample:

$$\begin{aligned}\hat{y} &= \arg \max_{1 \le k \le K} (y_k) \\ &= \arg \max_{1 \le k \le K} (z_k)\end{aligned} \tag{5.18}$$

The softmax classifier separates the classes with linear decision boundaries. To find the decision boundary for two-dimensional data with three classes, let:

$$\begin{aligned}z_1 &= w_{11}x_1 + w_{12}x_2 + b_1 \\ z_2 &= w_{21}x_1 + w_{22}x_2 + b_2 \\ z_3 &= w_{31}x_1 + w_{32}x_2 + b_3\end{aligned} \tag{5.19}$$

We have three decision boundaries between classes 1 and 2, 1 and 3, and 2 and 3. The linear decision boundary between classes 1 and 2 must satisfy:

$$w_{11}x_1 + w_{12}x_2 + b_1 = w_{21}x_1 + w_{22}x_2 + b_2 \tag{5.20}$$

then

$$(w_{11} - w_{21})x_1 + (w_{12} - w_{22})x_2 = b_2 - b_1 \tag{5.21}$$

For example, the decision boundaries are shown in Figure 5.2 (details in the next section).

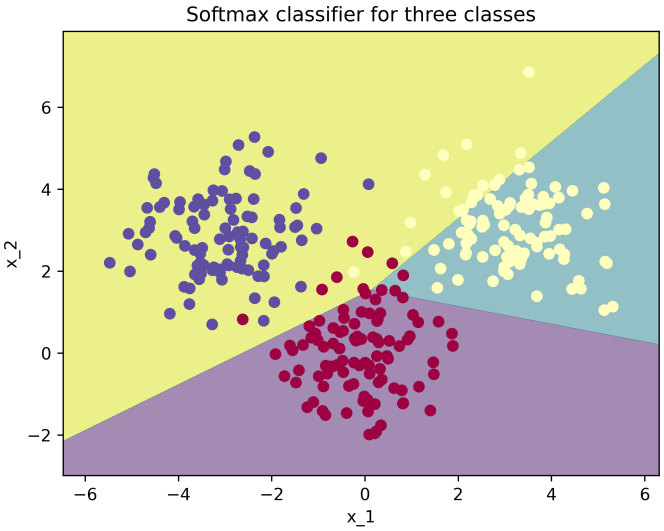


Figure 5.2: The decision boundaries between three classes for the two-dimensional data.

5.4 Example

A Python code is provided to illustrate the learning algorithm of the softmax classifier and plot the decision boundaries between three classes:

```
1
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  def softmax(x):
6      x = x - np.max(x, axis=1, keepdims=True)
7      exp_x = np.exp(x)
8      return exp_x / np.sum(exp_x, axis=1, keepdims=True)
9
10 def cross_entropy(y_true, y_pred):
11     y_pred = np.clip(y_pred, 1e-12, 1 - 1e-12)
12     return -np.mean(np.sum(y_true * np.log(y_pred), axis=1))
13
14 def gradient_descent(X, y_true, y_pred, w, b, learning_rate):
15     m = X.shape[0]
16     dw = (1/m) * np.dot(X.T, (y_pred - y_true))
17     db = (1/m) * np.sum(y_pred - y_true, axis=0)
18     w = w - learning_rate * dw
19     b = b - learning_rate * db
20     return w, b
21
22 np.random.seed(42)
23 X0 = np.random.multivariate_normal(mean=[0, 0], cov=[[1, 0], [0,
       1]], size=100)
24 y0 = np.array([[1, 0, 0]] * 100)
25 X1 = np.random.multivariate_normal(mean=[3, 3], cov=[[1, 0], [0,
       1]], size=100)
26 y1 = np.array([[0, 1, 0]] * 100)
27 X2 = np.random.multivariate_normal(mean=[-3, 3], cov=[[1, 0], [0,
       1]], size=100)
28 y2 = np.array([[0, 0, 1]] * 100)
29
30 X = np.concatenate((X0, X1, X2), axis=0)
31 y_true = np.concatenate((y0, y1, y2), axis=0)
32
33 w = np.random.randn(2, 3)
34 b = np.random.randn(3)
35
36 epochs = 100
37 learning_rate = 0.01
38
39 for epoch in range(epochs):
40     y_pred = softmax(np.dot(X, w) + b)
41     loss = cross_entropy(y_true, y_pred)
42     if epoch % 10 == 0:
```

```
        print(f"Epoch {epoch}, Loss: {loss:.4f}")
    w, b = gradient_descent(X, y_true, y_pred, w, b, learning_rate)

# Plotting the decision boundaries
x_min = X[:, 0].min() - 1
x_max = X[:, 0].max() + 1
y_min = X[:, 1].min() - 1
y_max = X[:, 1].max() + 1

xx, yy = np.meshgrid(np.arange(x_min,x_max,.01),np.arange(y_min,
    y_max,.01))
Z=np.argmax(softmax(np.dot(np.c_[xx.ravel(), yy.ravel()], w) + b),
    axis=1).reshape(xx.shape)

plt.contourf(xx ,yy ,Z ,alpha=.5)

plt.scatter(X[:,0],X[:,1],c=np.argmax(y_true,axis=1), cmap=plt.cm.
    Spectral)
plt.xlabel("x_1")
plt.ylabel("x_2")

plt.show()
plt.savefig("softmax_classifier.png", dpi =600)
```

Listing 5.1: Python example for plotting the softmax decision boundaries for three classes problem.

Since softmax regression has a unique solution, running the script several times will lead to the same result.

5.5 Assignment

Implement an optical recognition of handwritten digits (multiclass classification task) using the Keras deep learning library and Google colab. To load the dataset, visit https://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits.

6. Multilabel Classification Networks

Multilabel classification is a classification task where each input sample can be assigned multiple labels. For instance, a given image may contain both a cat and a dog and should be annotated both with the "cat" label and the "dog" label.

— François Chollet

In this chapter, we introduce the multilabel classification algorithm where each instance can belong to more than one class at the same time. For example, a document can have multiple topics, such as politics, religion, and education. The multilabel classification is different from the multiclass classification where each data point or instance must belong to one class only.

6.1 Model

Multilabel classification is a variant of the classification problem where multiple nonexclusive labels may be assigned to each instance. For example, suppose you want to classify a movie based on its genres. A movie can belong to more than one genre, such as comedy, romance, and action. In this case, the input is the movie and the output is a set of genres that describe the movie (three genre in total). For example, a movie that is only comedy would have an output of:

$$t = [1\ 0\ 0]$$

A movie that is both romance and action would have an output of:

$$t = [0\ 1\ 1]$$

A movie that has no genres would have an output of:

$$t = [0\ 0\ 0]$$

This output is called a multi-hot vector, where each element corresponds to a label and has a value of 0 or 1 depending on whether the label is present or not. A movie can have any combination of genres, so there are $2^3 = 8$ possible outputs for this problem.

One way to approach this problem is to transform it into binary classification problems, where each label is predicted independently by a binary logistic regression classifier. The target of such classifiers can be represented as a multi-hot vector, where each element corresponds to a label and has a value of 0 or 1 depending on whether the label is predicted or not. The model can be written as:

$$\begin{aligned} y &= \left[y_1\ y_2 \vdots y_L\right] \\ &= \left[\sigma(\mathbf{w}_1^T\mathbf{x} + b_1)\ \sigma(\mathbf{w}_2^T\mathbf{x} + b_2) \vdots \sigma(\mathbf{w}_L^T\mathbf{x} + b_L)\right] \end{aligned} \tag{6.1}$$

where σ is the sigmoid function, $\mathbf{x}$ is the input vector, $\mathbf{w}_i$ and b_i are the weight vector and bias term for the i-th label, L is the number of labels. An example of multilabel network is shown in Figure 6.1.

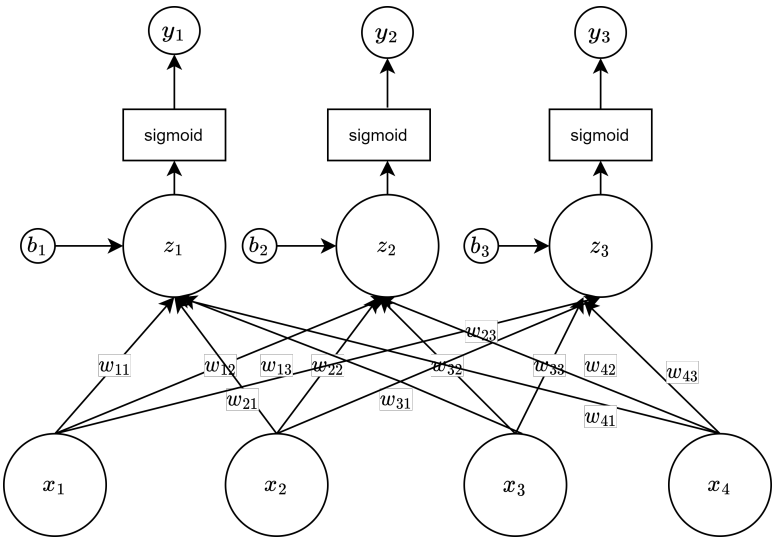


Figure 6.1: Multilabel classification network that has an input with 4 nodes or variables and the output has 3 nodes or classes.

6.2 Learning problem

The objective function for multilabel classification should be the summation of the binary cross-entropy losses for each label. Binary cross-entropy is a special case of categorical cross-entropy where the target is 0 or 1. It measures how well a model predicts a binary outcome. The binary cross-entropy loss for a single label can be written as:

$$E(t, y) = -t \log(y) - (1 - t) \log(1 - y) \tag{6.2}$$

where t is the true label (0 or 1) and y is the predicted probability (between 0 and 1).
The objective function for multilabel classification can be obtained by summing the binary cross-entropy losses for each label:

$$E(\mathbf{t}, \mathbf{y}) = -\sum_{i=1}^{L} [t_i \log(y_i) + (1 - t_i) \log(1 - y_i)] \tag{6.3}$$

where L is the number of labels, $\mathbf{t}$ is the true multi-hot vector, and $\mathbf{y}$ is the predicted probability vector.
This objective function can be used to train a multilabel classifier by minimizing it with respect to the model parameters (see Chapter 4 for details about gradient computation).
A summary of the learning problems discussed so far is shown in the following table:

Model	linear regression	binary classification	multiclass classification	multilabel classification
activation	linear	sigmoid	softmax	sigmoid
# of output nodes	$K \geq 1$	K=1	$K \geq 3$	$K \geq 2$
objective function	mean square error	binary cross entropy	categorical cross entropy	K binary cross entropy

Table 6.1: A summary for the learning algorithms.

6.3 Example

A Python code is provided to illustrate the learning algorithm of the multilabel classifier for three classes:

```
1
2 import numpy as np
```

```
3
4  def sigmoid(x):
5      return 1 / (1 + np.exp(-x))
6
7  def sigmoid_derivative(x):
8      return x * (1 - x)
9
10 class LinearNN:
11     def __init__(self, n_inputs, n_outputs):
12         self.weights = np.random.rand(n_inputs, n_outputs)
13         self.bias = np.random.rand(n_outputs)
14
15     def train(self, X, T, epochs, lr):
16         for epoch in range(epochs):
17             # forward propagation
18             y_pred = self.predict(X)
19
20             # compute gradient
21             d_weights = np.dot(X.T, (y_pred - T) *
    sigmoid_derivative(y_pred))
22             d_bias = np.sum((y_pred - T) * sigmoid_derivative(y_pred
    ), axis=0)
23
24             # update weights and bias
25             self.weights -= lr * d_weights
26             self.bias -= lr * d_bias
27
28             if epoch % 100 == 0:
29                 loss = np.mean(-T*np.log(y_pred) - (1-T)*np.log(1-
    y_pred))
30                 print(f'Loss at epoch {epoch}: {loss}')
31
32     def predict(self, X):
33         return sigmoid(np.dot(X, self.weights) + self.bias)
34
35 if __name__ == "__main__":
36     X = np.array([[0, 0, 1],
37                   [0, 1, 1],
38                   [1, 0, 1],
39                   [1, 1, 1]])
40     T = np.array([[0, 0, 1],
41                   [1, 0, 0],
42                   [0, 1, 0],
43                   [1, 1, 1]])
44
45     model = LinearNN(X.shape[1], T.shape[1])
46     model.train(X, T, epochs=1000000, lr=0.001)
47
48     print("Model weights:")
49     print(model.weights)
```

```
50     print("Model biases:")
51     print(model.bias)
52     print("Predictions on training data:")
53     print(np.round(model.predict(X)))
```

Listing 6.1: Python example for mutlilabel classifier. The classifier has three output classes.

6.4 Assignment

https://www.kaggle.com/datasets/shivanandmn/multilabel-classification-dataset

7. Deep Neural Networks

I have never claimed that I invented backpropagation. David Rumelhart invented it independently long after people in other fields had invented it. It is true that when we first published we did not know the history so there were previous inventors that we failed to cite. What I have claimed is that I was the person to clearly demonstrate that backpropagation could learn interesting internal representations and that this is what made it popular.

— Geoffrey E. Hinton

7.1 Motivation

Deep neural networks are composed of multiple layers of neurons that can learn complex patterns and features from the input data. However, if each layer of neurons uses a linear activation function, such as $g(z) = z$, then the output of the network will be just a linear combination of the inputs, regardless of how many layers there are. Assume a network with L layers and the output layer is a regression problem, then the output of the network is given by

$$\begin{aligned} y &= W^L W^{L-1} \dots W^1 x \\ &= Wx \end{aligned} \tag{7.1}$$

This means that the network will not be able to capture any non-linear relationships or interactions among the input variables, and will fail to model complex phenomena that do not follow linearity.

Table 7.1: Two-dimensional XOR gate truth table.

x_1	x_2	t
0	0	0
0	1	1
1	0	1
1	1	0

To overcome this limitation, we use non-linear activation functions, such as sigmoid, tanh, relu, etc. (see Figure 7.1), that can introduce non-linearity into the network.

$$y = W^L g(W^{L-1} \dots g(W^2 g(W^1 x + b^1) + b^2) + b^{L-1}) + b^L \tag{7.2}$$

Non-linear activation functions can map the input to a different range or domain, such as (0, 1) for sigmoid or (-1, 1) for tanh, and can create non-linear decision boundaries via non-linear combinations of the weights and inputs. Non-linear activation functions can also help the network to avoid saturation or vanishing gradients, which are problems that occur when the derivative of the activation function becomes very small or zero, and prevent the network from learning effectively.
By using non-linear activation functions in deep neural networks, we can enable the network to learn more expressive and powerful representations of the input data, and to approximate any continuous function

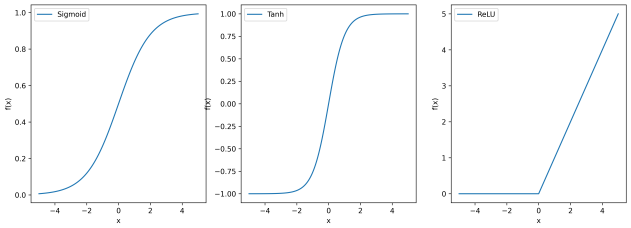


Figure 7.1: Three different activation functions commonly used in deep networks.

One of the functions that can not be solved using linear decision boundaries is the XOR gate that is used in design logic (see 7.1). If there is a linear decision boundary can solve the XOR problem, then its equation should be given by

$$x_1 w_1 + x_2 w_2 \lesseqgtr_0^1 \theta \tag{7.3}$$

where θ is a threshold. Since the points $(1, 0)$ and $(0, 1)$ lie on the same side of decision boundary, then

$$\begin{aligned} w_1 &> \theta \\ w_2 &> \theta \end{aligned} \tag{7.4}$$

on the other hand, the point (1, 1) lies on the other side of decision boundary:

$$w_1 + w_2 < \theta \tag{7.5}$$

which is a contradiction of Equation (7.4). Hence, such a linear decision boundary does not exist. This means that there is no single line that can separate the four points that represent the XOR table into two classes. Hence, the XOR problem can not be solved using linear networks. To solve the XOR problem, a non-linear neural network with at least one hidden layer is needed.
The output layer is the final layer in the network where the desired predictions are obtained. Depending on the type and goal of the task, the output layer can have different activation functions and loss functions.
For example, if the task is a regression problem, where the output is a continuous value, such as predicting the price of a house, then the output layer can have a linear activation function, such as g(x) = x, and a mean squared error loss function, which measures the difference between the predicted and actual values.
If the task is a binary classification problem, where the output is either 0 or 1, such as predicting whether an email is spam or not, then the output layer can have a sigmoid activation function, such as $g(x) = 1/(1 + exp(-x))$, and a binary cross-entropy loss function, which measures the probability of the correct class.
If the task is a multi-class classification problem, where the output is one of several possible classes, such as predicting the type of animal in an image, then the output layer can have a softmax activation function, such as $g(x) = \exp(x)/\sum(\exp(x))$, and a categorical cross-entropy loss function, which measures the probability of the correct class. An example of a deep neural network (DNN) is shown in Figure 7.2.
By having a flexible output layer, deep neural networks can adapt to different types of tasks and produce accurate and meaningful predictions.

7.2 Model

Consider a neural network model with two hidden layers and an output layer. The activation function for all layers, including the output layer, is the sigmoid function, ideal for a multi-class classification problem with K classes.
Let's denote:

- x: Input data vector.
- $W^{[1]}, W^{[2]}, W^{[3]}$: Weight matrices for Hidden Layer 1, Hidden Layer 2, and Output Layer, respectively.
- $b^{[1]}, b^{[2]}, b^{[3]}$: Bias vectors for the corresponding layers.
- $z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$: Linear transformation for each layer l.

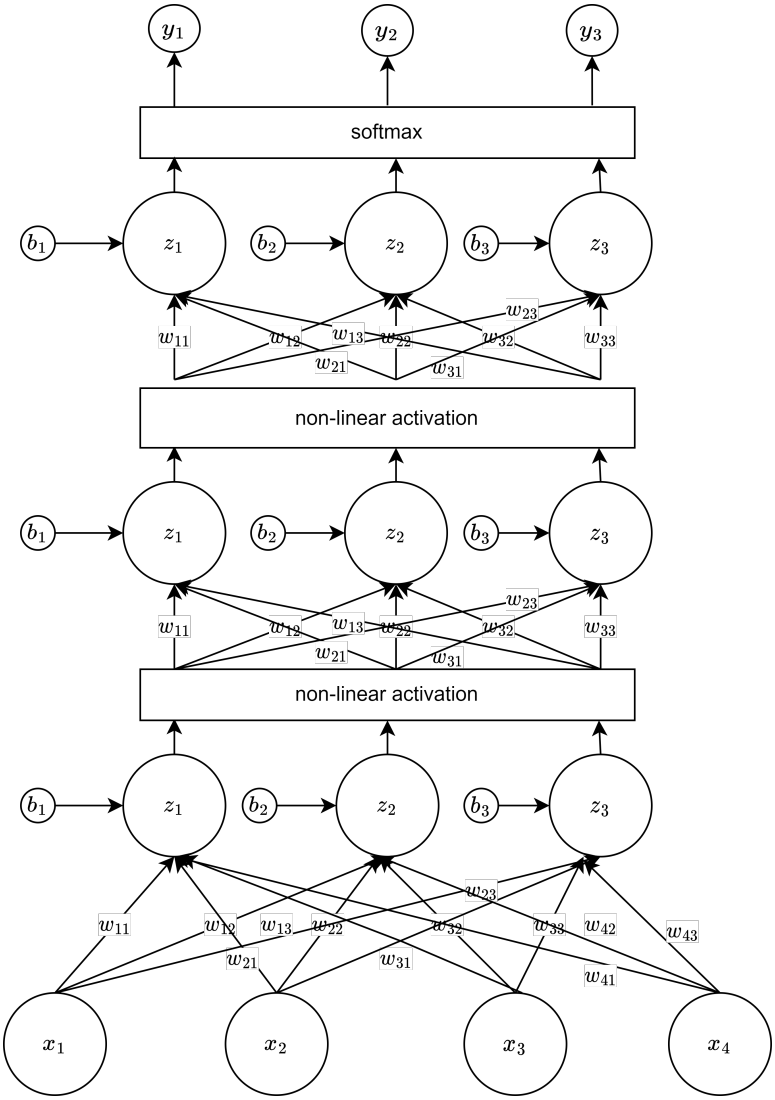


Figure 7.2: A deep neural network consists of 2 hidden layers (each layer has 3 nodes) with non-linear activation such as sigmoid function. The output layer has 3 nodes and a softmax activation to support a multi-class classification problem. The input layer has 4 nodes. The parameters – Please note the index of weight matrices and biases were removed in the figure for simplicity.– are the weights $W^{[3]}$, $W^{[2]}$, $W^{[1]}$ and the biases $b^{[3]}$, $b^{[2]}$, $b^{[1]}$.

- $a^{[l]} = \text{sigmoid}(z^{[l]})$: Sigmoid activation for each hidden layer.
- $a^{[L]} = \text{softmax}(z^{[l]})$: Softmax activation for the output layer.

7.3 Learning via Backpropagation

The backpropagation algorithm is fundamental for training artificial neural networks, and particularly deep learning networks. It is responsible for optimizing the weights in the network by computing gradients, allowing the network to learn from the error made during prediction.
The algorithm is based on a simple principle: it feeds input data forward through the network, computes the error by comparing the predicted and actual outputs, and then propagates this error backward through the network, adjusting the weights along the way. This iterative process is performed over multiple epochs and effectively trains the network.

7.3.1 Forward Propagation

First, we perform forward propagation to compute the activations for each layer:
1. Hidden Layer 1:

$$z^{[1]} = W^{[1]}\boldsymbol{x} + b^{[1]}$$
$$a^{[1]} = \text{sigmoid}(z^{[1]})$$

2. Hidden Layer 2:

$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$a^{[2]} = \text{sigmoid}(z^{[2]})$$

3. Output Layer:

$$z^{[3]} = W^{[3]}a^{[2]} + b^{[3]}$$
$$a^{[3]} = \text{softmax}(z^{[3]})$$

7.3.2 Backward Propagation

Next, we compute the gradients and propagate them back through the network:
The output layer uses the softmax activation function. We can write the softmax activation function as:

$$a_i^{[3]} = \frac{e^{z^{[3]}i}}{\sum_{j=1}^{K} e^{z_j^{[3]}}} \tag{7.6}$$

Where the denominator sums over all K classes.
The cost function is cross-entropy loss, which we can write as:

$$E = -\sum_{i=1}^{K} t_i \log a_i^{[3]} \tag{7.7}$$

Where the sum goes over all K classes.
Based on equation (5.15), the gradients are give by:

$$\frac{\partial E}{\partial z} = \frac{\partial a}{\partial z}\frac{\partial E}{\partial a} \tag{7.8}$$

.

$$\frac{\partial E}{\partial a} = \begin{pmatrix} -t_1/a_1 \\ \vdots \\ -t_n/a_n \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ -1/a_j \\ \vdots \\ 0 \end{pmatrix} \tag{7.9}$$

where j is the index of the correct class. In addition, $\frac{\partial \mathbf{a}}{\partial \mathbf{z}}$ is given by

$$\frac{\partial \mathbf{a}}{\partial \mathbf{z}} = \begin{pmatrix} a_1 & 0 & 0 & \cdots & 0 \\ 0 & a_2 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_K \end{pmatrix} - \begin{pmatrix} a_1a_1 & a_1a_2 & a_1a_3 & \cdots & a_1a_K \\ a_2a_1 & a_2a_2 & a_2a_3 & \cdots & a_2a_K \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_Ka_1 & a_Ka_2 & a_Ka_3 & \cdots & a_Ka_K \end{pmatrix} \tag{7.10}$$

Hence,

$$\frac{\partial E}{\partial z} = \frac{1}{a_j}\left[\begin{pmatrix} a_1a_j \\ \vdots \\ a_ja_j \\ \vdots \\ -a_Ka_j \end{pmatrix} - \begin{pmatrix} 0 \\ \vdots \\ a_j \\ \vdots \\ 0 \end{pmatrix}\right] = a - t \tag{7.11}$$

1. Output Layer:

$$\delta^{[3]} = a^{[3]} - t \tag{7.12}$$

$$\frac{\partial E}{\partial W^{[3]}} = \delta^{[3]}(a^{[2]})^T \tag{7.13}$$

$$\frac{\partial E}{\partial b^{[3]}} = \delta^{[3]} \tag{7.14}$$

2. Hidden Layer 2:

$$\delta^{[2]} = (W^{[3]})^T \delta^{[3]} \circ a^{[2]} \circ (1 - a^{[2]}) \tag{7.15}$$

$$\frac{\partial E}{\partial W^{[2]}} = \delta^{[2]} (a^{[1]})^T \tag{7.16}$$

$$\frac{\partial E}{\partial b^{[2]}} = \delta^{[2]} \tag{7.17}$$

3. Hidden Layer 1:

$$\delta^{[1]} = (W^{[2]})^T \delta^{[2]} \circ a^{[1]} \circ (1 - a^{[1]}) \tag{7.18}$$

$$\frac{\partial E}{\partial W^{[1]}} = \delta^{[1]} \boldsymbol{x}^T \tag{7.19}$$

$$\frac{\partial E}{\partial b^{[1]}} = \delta^{[1]} \tag{7.20}$$

Here, $\circ$ denotes element-wise multiplication, which is also known as the Hadamard product. Note that these updates are performed after each iteration or epoch. Finally, the weights and biases are updated using gradient descent:

$$W^{[l]} = W^{[l]} - \eta \frac{1}{N} \sum_{n=1}^{N} \frac{\partial E^n}{\partial W^{[l]}}, \tag{7.21}$$

$$b^{[l]} = b^{[l]} - \eta \frac{1}{N} \sum_{n=1}^{N} \frac{\partial E^n}{\partial b^{[l]}}, \tag{7.22}$$

where η is the learning rate.
This process is repeated for multiple epochs until the network is adequately trained. The delta term $\delta^{[l]}$ represents the error or gradient of the loss function with respect to the pre-activation value $z^{[l]}$ of layer l, i.e.,

$$\delta^{[l]} = \frac{\partial E}{\partial z^{[l]}} \tag{7.23}$$

The exact computation of δ differs based on the layer. For the output layer (layer 3):
We use the chain rule of calculus,

$$\delta^{[3]} = \frac{\partial E}{\partial z^{[3]}} = \frac{\partial E}{\partial a^{[3]}} \cdot \frac{\partial a^{[3]}}{\partial z^{[3]}} \tag{7.24}$$

$$\delta^{[3]} = a^{[3]} - t$$

For the hidden layer 2, the delta term is computed using the delta term of the next layer (output layer) and the weight matrix that connects them. This is because the loss function depends on $z^{[2]}$ indirectly through $z^{[3]}$. The derivative of the loss function with respect to $z^{[2]}$ is:

$$\frac{\partial E}{\partial z^{[2]}} = \frac{\partial E}{\partial z^{[3]}} \frac{\partial z^{[3]}}{\partial z^{[2]}}$$

Using the fact that $z^{[3]} = W^{[3]}a^{[2]} + b^{[3]}$, where $W^{[3]}$ and $b^{[3]}$ are the weight matrix and bias vector of layer 3, respectively, and applying the chain rule, we get:

$$\frac{\partial E}{\partial z^{[2]}} = \frac{\partial E}{\partial z^{[3]}} \frac{\partial (W^{[3]}a^{[2]} + b^{[3]})}{\partial z^{[2]}}$$

The derivative of $(W^{[3]}a^{[2]} + b^{[3]})$ with respect to $z^{[2]}$ is simply $W^{[3]}$, since $a^{[2]}$ depends on $z^{[2]}$ and $b^{[3]}$ does not. Therefore, we can simplify the expression as:

$$\frac{\partial E}{\partial z^{[2]}} = \frac{\partial E}{\partial z^{[3]}} W^{[3]}$$

Using the fact that $\frac{\partial E}{\partial z^{[3]}} = \delta^{[3]}$, we can write this in vector form as:

$$\delta^{[2]} = (W^{[3]})^T \delta^{[3]} \tag{7.25}$$

However, this is not the final form of $\delta^{[2]}$, since we have not taken into account the effect of the activation function of layer 2. Recall that $a^{[2]} = g(z^{[2]})$, where g is the activation function of layer 2. To account for this, we need to multiply $\delta^{[2]}$ by the derivative of g with respect to $z^{[2]}$, which is denoted by $g'(z^{[2]})$. This gives us:

$$\delta^{[2]} = (W^{[3]})^T \delta^{[3]} \circ g'(z^{[2]}) \tag{7.26}$$

where $\circ$ denotes element-wise multiplication, also known as the Hadamard product. If we assume that g is a sigmoid function, such as $g(z) = \frac{1}{1+e^{-z}}$, then $g'(z) = g(z)(1 - g(z))$, and we can simplify the expression as:

$$\delta^{[2]} = (W^{[3]})^T \delta^{[3]} \circ a^{[2]} \circ (1 - a^{[2]}) \tag{7.27}$$

For the hidden layer 1, the delta term is computed in a similar way as for hidden layer 2, using the delta term of the next layer (hidden layer 2) and the weight matrix that connects them.
The back-propagation algorithm is illustrated in Figure 7.3.

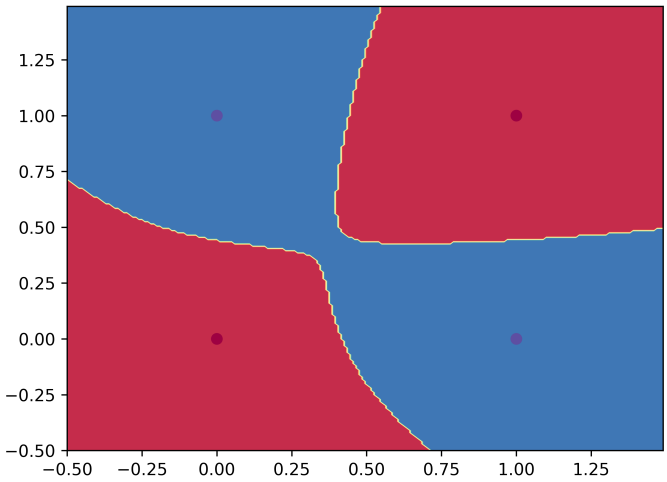


Figure 7.3: The back-propagation.

7.4 Example

A Python code is provided to illustrate the learning algorithm of the XOR problem and plot the decision boundary between the two classes:

```
1
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  def sigmoid(x):
6      return 1 / (1 + np.exp(-x))
7
8  def sigmoid_derivative(x):
9      return x * (1 - x)
10
11 # XOR input
12 X = np.array([[0,0], [0,1], [1,0], [1,1]])
13 # XOR output
14 Y = np.array([[0],[1],[1],[0]])
15
16 np.random.seed(0)
17
18 # initialize weights and biases randomly with mean 0
19 w0 = 2*np.random.random((2,4)) - 1
20 b0 = 2*np.random.random((1,4)) - 1
21 w1 = 2*np.random.random((4,1)) - 1
```

```
b1 = 2*np.random.random((1,1)) - 1

# Learning rate
lr = 0.1
# Number of iterations
epochs = 10000

for epoch in range(epochs):
    # Forward propagation
    l0 = X
    l1 = sigmoid(np.dot(l0, w0) + b0)
    l2 = sigmoid(np.dot(l1, w1) + b1)

    # Backward propagation
    l2_error = Y - l2
    l2_delta = l2_error * sigmoid_derivative(l2)

    l1_error = l2_delta.dot(w1.T)
    l1_delta = l1_error * sigmoid_derivative(l1)

    # Update weights and biases
    w1 += l1.T.dot(l2_delta) * lr
    b1 += np.sum(l2_delta, axis=0, keepdims=True) * lr
    w0 += l0.T.dot(l1_delta) * lr
    b0 += np.sum(l1_delta, axis=0, keepdims=True) * lr

# Plot decision boundary
h = 0.01
x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min,
    y_max, h))

l0 = np.c_[xx.ravel(), yy.ravel()]
l1 = sigmoid(np.dot(l0, w0) + b0)
l2 = sigmoid(np.dot(l1, w1) + b1)

# threshold the output
Z = (l2 > 0.5).astype(int)

Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral)
plt.scatter(X[:, 0], X[:, 1], c=Y[:, 0], cmap=plt.cm.Spectral)

plt.savefig('xor_decision_boundary.png', dpi=600)
```

Listing 7.1: Python example for plotting the XOR decision boundary.

In Figure 7.4, the 2D plot for the XOR problem results in a decision boundary that separates the space into four regions. Depending on how the network learns, the

shape of the boundary may vary, but it will always be able to separate [0, 0] and [1, 1] from [0, 1] and [1, 0].

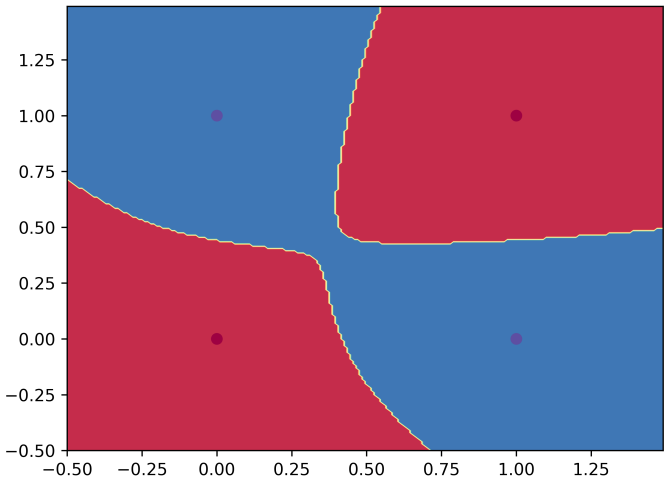


Figure 7.4: The decision boundaries between two classes for the two-dimensional data XOR problem.

7.5 Assignment

Implement an optical recognition of handwritten digits (multiclass classification task) using deep neural networks. The Keras deep learning library and Google colab can be used for this task. To load the dataset, visit `https://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits`.

8. Convolutional Networks

Deep Learning is far, far more than old-style neural nets with more than a couple of layers. DL is an "architectural language" with enormous flexibility. The bestiary of DL architectural elements has diversified tremendously over the last decade.

— Yann LeCun

Your Content

9. Recurrent Neural Networks

Recurrent nets are in principle capable to store past inputs to produce the currently desired output. Because of this property recurrent nets are used in time series prediction and process control. Practical applications involve temporal dependencies spanning many time steps, e.g. between relevant inputs and desired outputs. In this case, however, gradient based learning methods take too much time. The extremely increased learning time arises because the error vanishes as it gets propagated back.

— Sepp Hochreiter

Recurrent neural networks, and in particular, Long Short-Term Memory (LSTM) networks, have revolutionized the field of sequence modeling. LSTMs provide a powerful framework for capturing long-range dependencies in sequential data, enabling machines to understand and generate complex sequences with remarkable fluency and precision. In this chapter, we will introduce RNNs and LSTMs and address the difficulty to train these networks.

9.1 Model

A simple RNN is a type of artificial neural network that is designed to process sequential data. It has a recurrent structure that allows it to maintain an internal state (hidden state) that captures information from previous steps in the sequence. Unlike linear dynamical systems, RNNs are nonlinear models and can learn complex

patterns and dependencies in sequential data. The hidden state of an RNN is updated at each time step based on the current input and the previous hidden state, using nonlinear activation functions. RNNs possess a recurrent connection enabling them to store information about past inputs within their hidden state.
Unlike feedforward neural networks, which process data in a single forward pass and have no memory of past inputs, RNNs have a recurrent connection that allows them to maintain information about previous inputs in their hidden state. A dynamical system may be defined by:

$$h_t = f_h(X_t, h_{t-1}) \tag{9.1}$$
$$y_t = f_o(h_t) \tag{9.2}$$

A simple RNN [10] comprises three layers: an input layer, a hidden layer, and an output layer as shown in Figure 9.1. The input layer receives sequential data during each time step, the hidden layer retains memory and processes the sequential information, and the output layer generates the final output or prediction.

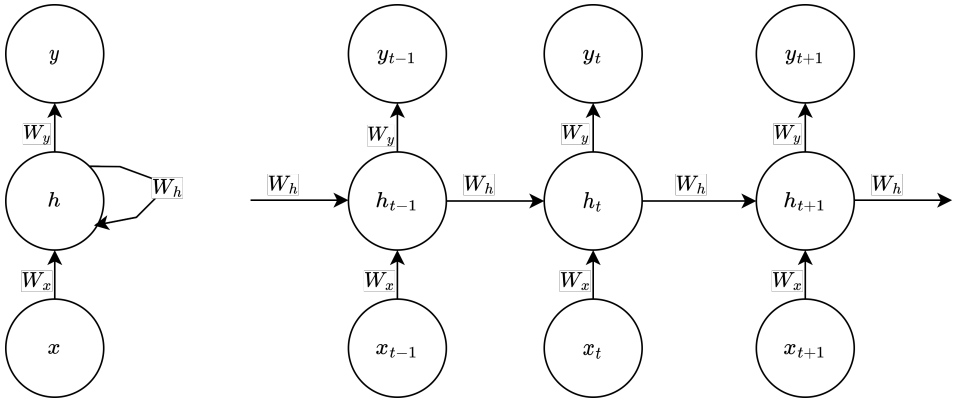


Figure 9.1: An unfolded simple recurrent neural network. The matrices W_x, W_h, W_y are shared between time steps. Unfolding a simple RNN helps to illustrate how the hidden state is passed along and updated as the network processes sequential data. It provides a visual representation of the temporal dependencies and the flow of information within the RNN.

Let's define the forward pass equations of the simple RNN model:

- Hidden state activation:

$$z_t^h = W_h h_{t-1} + W_x x_t + b_h \tag{9.3}$$
$$h_t = \tanh(z_t^h) \tag{9.4}$$

- Output activation:

$$z_t^y = W_y h_t + b_y \tag{9.5}$$
$$y_t = \text{softmax}(z_t^y) \tag{9.6}$$

The hidden activation function is tanh and its derivative is $1 - \tanh^2(x)$. The output activation function is softmax, which for a specific class i is defined as:

$$\text{softmax}(z_i^y) = \frac{e^{z_i^y}}{\sum_j e^{z_j^y}} \tag{9.7}$$

Where y represents the output and l represents the target. The activations of the hidden state and output are z^h and z^y respectively. The weight matrix connecting the hidden layer to the output is W_y.
Let's denote d_{neurons} as the number of neurons in the hidden layer and d_{inputs} as the number of input features. Then:

- x_t is a vector of size d_{inputs} containing the inputs at time t.
- h_{t-1} is a d_{neurons} vector containing the hidden state of the previous time-step. At the first time step, $t = 0$, there are no previous hidden state, so $h_{t-1} = 0$.
- W_x is a $d_{\text{neurons}} \times d_{\text{inputs}}$ matrix containing the connection weights between input and the hidden layer.
- W_h is a $d_{\text{neurons}} \times d_{\text{neurons}}$ matrix containing the connection weights between two hidden layers.
- W_y is a $d_{output} \times d_{\text{neurons}}$ matrix containing the connection weights between the hidden layer and the output.
- b_h is a vector of size d_{neurons} containing each neuron's bias term.
- b_y is a vector of size d_{output} containing the bias term of the output layer.
- y_t is a vector of size d_{output} containing the layer's outputs at time step t.

RNNs are designed to handle sequential data with varying lengths, dividing the input data into multiple time steps, where each step represents an element of the sequence.
A distinguishing feature of RNNs is the recurrent connection, which allows the network to retain information across time steps. At each time step, the hidden layer takes input from the current step as well as the hidden state from the previous step, enabling the hidden state to carry information from past steps. Moreover, the same set of weights and biases is used at each time step, enabling effective processing of sequences with different lengths.

9.2 Learning problem

RNNs are trained using backpropagation through time (BPTT), an extension of the standard backpropagation algorithm that accounts for the recurrent nature of the network. BPTT calculates gradients and updates the weights and biases to minimize the discrepancy between the predicted output and the target value. For the backward pass, we will compute the gradients of the loss with respect to the activations and parameters. Let's define the loss at each time step as:

$$E_t = -\sum_i l_{ti} \log y_{ti} \tag{9.8}$$

This is the categorical cross-entropy loss for softmax outputs. The overall loss is given by

$$E = \sum_{t=1}^{T} E_t(l_t, y_t) \tag{9.9}$$

$$= -\sum_{t}^{T} l_t \log y_t \tag{9.10}$$

$$= -\sum_{t=1}^{T} l_t \log[\text{softmax}(y_t)] \tag{9.11}$$

Firstly, let's define the quantities δ_t^y and δ_t^h:

$$\delta_t^y = \frac{\partial E_t}{\partial z_t^y} \tag{9.12}$$

To get the gradient with respect to the pre-activation z_t^y, we need to apply the chain rule to differentiate L_t:

$$\frac{\partial E_t}{\partial z_t^y} = \frac{\partial E_t}{\partial y_t} \times \frac{\partial y_t}{\partial z_t^y} \tag{9.13}$$

This derivative is calculated as the derivative of the loss function with respect to the output, multiplied by the derivative of the softmax activation function. Using the properties of the softmax function and the cross-entropy loss (i.e. see Chapter 5), the above simplifies to:

$$\delta_t^y = y_t - l_t \tag{9.14}$$

Now let's define δ_t^h:

$$\delta_t^h = \frac{\partial E_t}{\partial z_t^h} \tag{9.15}$$

This derivative includes all paths through the computational graph where z_t^h affects the loss function E. This is calculated using the chain rule as:

$$\delta_t^h = \frac{\partial E_t}{\partial h_t}\frac{\partial h_t}{\partial z_t^h} = \left(\frac{\partial E_t}{\partial z_t^y}\frac{\partial z_t^y}{\partial h_t} + \frac{\partial E_t}{\partial z_{t+1}^h}\frac{\partial z_{t+1}^h}{\partial h_t}\right)\frac{\partial h_t}{\partial z_t^h} \tag{9.16}$$

Substituting δ_t^y and δ_{t+1}^h and the derivative of the tanh function:

$$\delta_t^h = \left(\delta_t^y {W_y}^T + \delta_{t+1}^h {W_h}^T\right) \odot (1 - \tanh^2(z_t^h)) \tag{9.17}$$

This equation is the form of the backpropagation through time (BPTT) for hidden states in RNN. The $\odot$ denotes the Hadamard product (element-wise multiplication). The term $1 - \tanh^2(z_t^h)$ is the derivative of the tanh activation function. The terms $\delta_t^y {W_y}^T$ and $\delta_{t+1}^h {W_h}^T$ represent the influence of the error at the output at time t and the influence of the error at the hidden state at time $t+1$ respectively. They are both backpropagated to the hidden state at time t through the respective weight matrices W_y and W_h.
For an input sequence of length T, the gradient accumulations are given by
For the weight matrix W_y:

$$\frac{\partial E}{\partial W_y} = \sum_{t=1}^{T}\frac{\partial E_t}{\partial z_t^y}\frac{\partial z_t^y}{\partial W_y} = \sum_{t=1}^{T}\delta_t^y \cdot h_t^T \tag{9.18}$$

For the bias b_y:

$$\frac{\partial E}{\partial b_y} = \sum_{t=1}^{T}\frac{\partial E_t}{\partial z_t^y} = \sum_{t=1}^{T}\delta_t^y \tag{9.19}$$

For the recurrent weight matrix W_h:

$$\frac{\partial E}{\partial W_h} = \sum_{t=1}^{T}\frac{\partial E_t}{\partial z_t^h}\frac{\partial z_t^h}{\partial W_h} = \sum_{t=1}^{T}\delta_t^h \cdot h_{t-1}^T \tag{9.20}$$

For the input weight matrix W_x:

$$\frac{\partial E}{\partial W_x} = \sum_{t=1}^{T}\frac{\partial E_t}{\partial z_t^h}\frac{\partial z_t^h}{\partial W_x} = \sum_{t=1}^{T}\delta_t^h \cdot x_t^T \tag{9.21}$$

For the bias b_h:

$$\frac{\partial E}{\partial b_h} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial z_t^h} = \sum_{t=1}^{T} \delta_t^h \tag{9.22}$$

After accumulating the gradients over the entire sequence, you can use an optimization algorithm (like SGD, Adam, etc.) to update the weights and biases:

$$W = W - \eta \frac{\partial E}{\partial W} \tag{9.23}$$

$$b = b - \eta \frac{\partial E}{\partial b} \tag{9.24}$$

where η is the learning rate.
Simple RNNs face a challenge known as the vanishing or exploding gradients problem [11]. This arises when gradients in the network become excessively small (vanishing gradients) or large (exploding gradients) while processing lengthy sequences. This issue hampers the training process and the model's ability to retain long-term dependencies.

9.3 The difficulty of training simple RNN

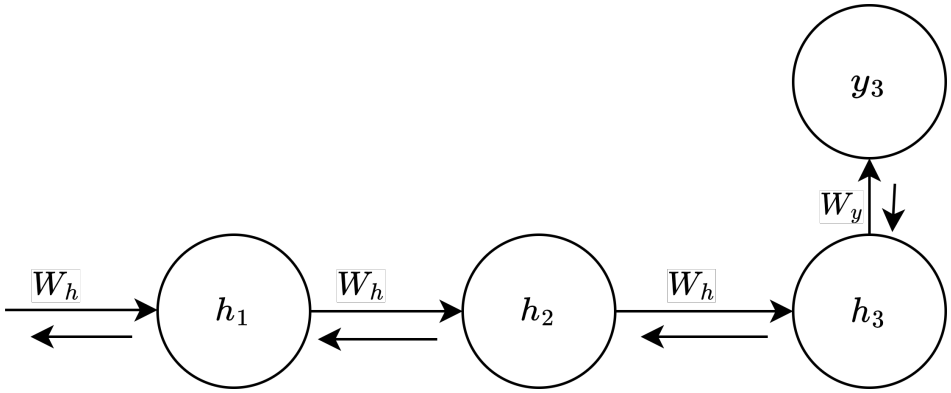


Figure 9.2: Propagating gradients through the unfolded RNN. The memory unit, h_t, is a function of its previous memory unit h_{t-1}. Hence, we differentiate h_3 with h_2 and h_2 with h_1.

In the following Figure 9.2, we have three-time steps. Then

$$\frac{\partial E_3}{\partial W_h} = \frac{\partial E_3}{\partial y_3}\frac{\partial y_3}{\partial h_3}\frac{\partial h_3}{\partial W_h} + \frac{\partial E_3}{\partial y_3}\frac{\partial y_3}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial W_h} + \frac{\partial E_3}{\partial y_3}\frac{\partial y_3}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial W_h}, \tag{9.25}$$

where the first term is a direct application of the chain rule. However, we have to take into consideration the previous time steps. So, we differentiate the cost function with respect to memory units h_2 as well as h_1 taking into consideration the weight matrix W_h. Please note that a memory unit h_t is a function of its previous memory unit h_{t-1} according to the recursive formulation ($h_t = \tanh(W_h h_{t-1} + W_x x_t + b_h)$). Hence, we differentiate h_3 with h_2 and h_2 with h_1.
Hence, at the time-step t, we can compute the gradient and further use backpropagation through time from t to 1 to compute the overall gradient with respect to W_h:

$$\frac{\partial E_t}{\partial W_h} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W_h}, \tag{9.26}$$

and $\frac{\partial h_t}{\partial h_k}$ can be computed using a chain rule. It can be written as follows

$$\frac{\partial E_t}{\partial W_h} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \left(\prod_{j=k}^{t-1} \frac{\partial h_{j+1}}{\partial h_j} \right) \frac{\partial h_k}{\partial W_h} \tag{9.27}$$

where

$$\prod_{j=k}^{t-1} \frac{\partial h_{j+1}}{\partial h_j} = \frac{\partial h_t}{\partial h_k} = \frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial h_{t-2}} \cdots \frac{\partial h_{k+1}}{\partial h_k}, \tag{9.28}$$

Since we differentiate a vector function with respect to a vector, the result is a matrix (called the Jacobian matrix) whose elements are all point-wise derivatives. Aggregate the gradients with respect to W_h over the whole time-steps with back-propagation, we can finally yield the following gradient with respect to W_h:

$$\frac{\partial E}{\partial W_h} = \sum_{t=1}^{T} \sum_{k=1}^{t} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W_h} \tag{9.29}$$

Let's take the norms[1] of these Jacobians $\left\| \frac{\partial h_{j+1}}{\partial h_j} \right\|$ and let $\phi_h = \tanh(W_h h_{t-1} + W_x x_t + b_h)$:

$$\left\| \frac{\partial h_{j+1}}{\partial h_j} \right\| \leq \|W_h\| \left\| diag(\phi_h'(h_j)) \right\| \tag{9.30}$$

In this equation, we set γ_W, the largest eigenvalue associated with $\|W_h\|$ as its upper bound, while γ_h largest eigenvalue associated with

$$\left\| diag(\phi_h'(h_j)) \right\|$$

[1]The 2-norm may be interpreted as an absolute value, of the Jacobian matrix.

as its corresponding upper-bound.
Depending on the chosen activation function ϕ_h, the derivative ϕ'_h will be upper bounded by different values. For hyperbolic tangent function as shown in Figure 9.3, we have $\gamma_h = 1$ while for sigmoid function, we have $\gamma_h = 0.25$. Thus, the chosen upper bounds γ_W and γ_h end up being a constant term resulting from their product:

$$\left\| \frac{\partial h_{j+1}}{\partial h_j} \right\| \leq \gamma_W \gamma_h \tag{9.31}$$

The gradient $\frac{\partial h_{t+1}}{\partial h_k}$ is a product of Jacobian matrices that are multiplied many times, $t - k$ times in our case:

$$\left\| \frac{\partial h_{t+1}}{\partial h_k} \right\| = \left\| \prod_{j=k}^{t} \frac{\partial h_{j+1}}{\partial h_j} \right\| \leq (\gamma_W \gamma_h)^{t-k} \tag{9.32}$$

The equation can be paraphrased as follows: The magnitude of the partial derivative of the hidden state at time step $t + 1$ with respect to the hidden state at time step k is equal to the magnitude of the product of the partial derivatives of consecutive hidden states from time step k to time step t. This magnitude is bounded by the value of the product of two factors, γ_W and γ_h, raised to the power of $t - k$. When the sequence becomes longer, meaning there is a larger distance between time steps t and k, the equation shows that the value can either become very small or very large quickly. This violates the assumption of locality in gradient descent. The outcome depends on the value of gamma: if it is large, the gradient can explode (become very large), and if it is small, the gradient can vanish (become very small). These problems highlight that when the gradient vanishes, it implies that the earlier hidden states do not significantly influence the later hidden states. In other words, the network fails to learn long-term dependencies, as the information from earlier time steps becomes negligible or irrelevant in the later ones.
Assume $\gamma_h = 1$ then if the norm of the weight matrix W_h is less than 1, each time we multiply the gradient by W_h, the magnitude of the gradient decreases. Imagine multiplying a number by 0.5 repeatedly - the result gets smaller and smaller. This is why $||W_h|| < 1$ is a sufficient[2] condition for the gradients to vanish. It guarantees vanishing gradients because multiplying by a number less than one repeatedly will always cause the result to tend towards zero. However, it's not a necessary condition because there are other reasons gradients might vanish, such as saturating activation functions.

[2]The difference between necessary and sufficient conditions can be quite subtle. In mathematics, a necessary condition must be true for the given statement to be true, but it is not enough on its own to guarantee the statement is true. A sufficient condition, on the other hand, if true, guarantees the statement is true.

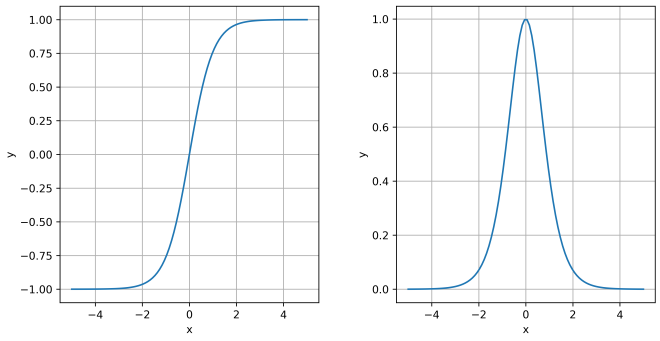


Figure 9.3: The hyperbolic tangent activation function and its derivative.

On the other hand, if the norm of the weight matrix W_h is greater than 1, each multiplication by W_h increases the magnitude of the gradient. Think about multiplying a number by 2 over and over - the result gets larger and larger. So $||W_h|| > 1$ is a necessary condition for gradient explosion - if the gradients are going to explode, there must be some factor making them bigger, such as a weight matrix with a norm greater than 1. But just because $||W_h|| > 1$ doesn't guarantee that the gradients will explode. Other factors might prevent this, such as gradient clipping or careful initialization of the weights. This is why $||W_h|| > 1$ is necessary, but not *sufficient*, for gradient explosion.

Remember that these are simplifications - in reality, the behavior of the gradients in an RNN will depend on a combination of many factors, including the specific sequences of inputs, the activation functions, and the structure of the network, in addition to the weight matrices. But considering the norms of the weight matrices provides some insight into the challenges faced when training RNNs, namely the problems of vanishing and exploding gradients.

The problem of vanishing gradients is not exclusive to recurrent neural networks (RNNs) but also occurs in deep feedforward neural networks. However, RNNs are typically deeper, which makes this issue more common in RNNs.

The vanishing or exploding gradients problem can hinder the effective capture of long-range dependencies by simple RNNs. To mitigate this issue, more advanced RNN architectures such as Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) were introduced. These architectures incorporate mechanisms that enable better control of information flow within the network, addressing the challenges associated with vanishing or exploding gradients.

9.4 Long Short-Term Memory networks

In practice, RNNs cannot capture long term dependencies due to the gradient exploding and vanishing problems [11]. LSTMs were introduced [12] to overcome the gradient vanishing problems and they are an essential component for many applications.

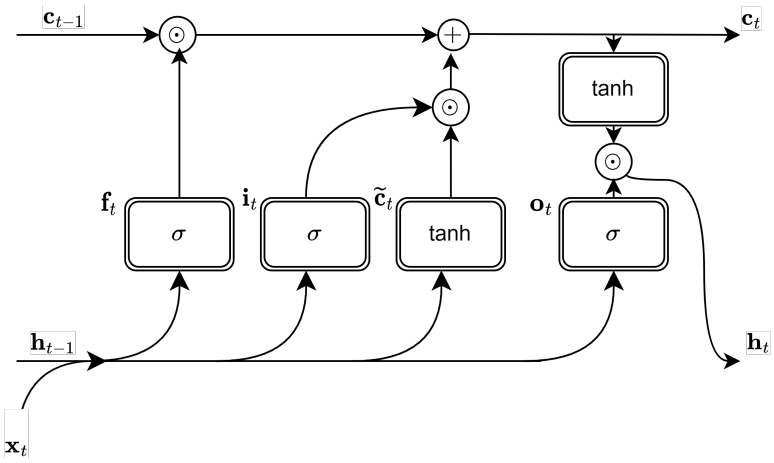


Figure 9.4: The hidden state computation in an LSTM network.

An LSTM network (see Figure 9.5) has a memory cell and three gating units: the input gate is used to control the amount of information to *add* to the current memory, the forget gate is used to control the amount of information to *remove* from the previous memory, and the output gate is used to control the amount of information to *output* from the current memory. These gates take as input the previous hidden state and the current input, and outputs a number between 0 and 1 (i.e. logistic function). The flow of information into or out of the memory is controlled by the multiplication of the output of these gates. The updates at each time step t are given by:

$$\mathbf{i}_t = \sigma(\mathbf{W}_i\mathbf{h}_{t-1} + \mathbf{U}_i\mathbf{x}_t) \tag{9.33}$$
$$\mathbf{f}_t = \sigma(\mathbf{W}_f\mathbf{h}_{t-1} + \mathbf{U}_f\mathbf{x}_t) \tag{9.34}$$
$$\mathbf{o}_t = \sigma(\mathbf{W}_o\mathbf{h}_{t-1} + \mathbf{U}_o\mathbf{x}_t) \tag{9.35}$$
$$\widetilde{\mathbf{c}}_t = \tanh(\mathbf{W}_c\mathbf{h}_{t-1} + \mathbf{U}_c\mathbf{x}_t) \tag{9.36}$$
$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \widetilde{\mathbf{c}}_t \tag{9.37}$$
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \tag{9.38}$$

where $\mathbf{i}_t$ is the input gate, $\mathbf{f}_t$ is the forget gate, $\mathbf{o}_t$ is the output gate, $\mathbf{c}_t$ is the memory cell, and $\mathbf{h}_t$ is the hidden state. $\odot$ denotes element-wise multiplication.

$\mathbf{W}_i$,$\mathbf{U}_i$, $\mathbf{W}_f$,$\mathbf{U}_f$, $\mathbf{W}_o$,$\mathbf{U}_o$,$\mathbf{W}_c$,$\mathbf{U}_c$, are weight matrices (parameters) of the LSTM network. A variant of LSTM known as bidirectional BiLSTM [13] allows the integration of both past and future information. It is a combination of two LSTMs in two directions: one operates in the forward direction and the other operates in the backward direction. Hence, each input word at time t is aware about the past and future contexts which may improve the results.
Similar to LSTMs, Gated Recurrent Units (GRUs) were developed to handle long term dependencies [14]. The gated recurrent units (GRUs) [15] which have the following forward updates:

$$\mathbf{z}_t = \sigma(\mathbf{W}_z\mathbf{h}_{t-1} + \mathbf{U}_z\mathbf{x}_t) \tag{9.39}$$

$$\mathbf{r}_t = \sigma(\mathbf{W}_r\mathbf{h}_{t-1} + \mathbf{U}_r\mathbf{x}_t) \tag{9.40}$$

$$\tilde{\mathbf{h}}_t = \tanh(\mathbf{W}_h(\mathbf{h}_t \odot \mathbf{r}_t) + \mathbf{U}_h\mathbf{x}_t) \tag{9.41}$$

$$\mathbf{h}_t = \mathbf{z}_t \odot \mathbf{h}_{t-1} + (1 - \mathbf{z}_t) \odot \tilde{\mathbf{h}}_t \tag{9.42}$$

where $\mathbf{z}_t$ is the update gate, $\mathbf{r}_t$ is the reset gates. $\mathbf{W}_z$,$\mathbf{U}_z$, $\mathbf{W}_r$,$\mathbf{U}_r$, $\mathbf{W}_h$,$\mathbf{U}_h$, are the parameters of the GRU networks.

9.4.1 Vanishing/Exploding Gradients with LSTMs

The cell state in the LSTMs is given by

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \widetilde{\mathbf{c}}_t \tag{9.43}$$

To find the derivative $\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}}$, we notice that $\mathbf{c}_t$ is a function of $\mathbf{f}_t$ (the forget gate), $\mathbf{i}_t$ (input gate) and $\widetilde{\mathbf{c}}_t$ (candidate input), and each of these being a function of $\mathbf{c}_{t-1}$ (since they are all functions of $\mathbf{h}_{t-1}$). Using the multivariate chain rule we get:

$$\begin{aligned}
\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}} &= \frac{\partial \mathbf{f}_t}{\partial \mathbf{c}_{t-1}} \cdot \mathbf{c}_{t-1} + \mathbf{f}_t + \frac{\partial \mathbf{i}_t}{\partial c_{t-1}} \cdot \widetilde{\mathbf{c}}_t + \frac{\partial \widetilde{\mathbf{c}}_t}{\partial c_{t-1}} \cdot \mathbf{i}_t \\
&= \sigma'(\mathbf{W}_f\mathbf{h}_{t-1} + \mathbf{U}_f\mathbf{x}_t) \cdot \mathbf{W}_f \cdot \mathbf{o}_{t-1} \odot \tanh(\mathbf{c}_{t-1}) \cdot \mathbf{c}_{t-1} \\
&+ \mathbf{f}_t \\
&+ \sigma'(\mathbf{W}_i\mathbf{h}_{t-1} + \mathbf{U}_i\mathbf{x}_t) \cdot \mathbf{W}_i \cdot \mathbf{o}_{t-1} \odot \tanh(\mathbf{c}_{t-1}) \cdot \widetilde{\mathbf{c}}_t \\
&+ \sigma'(\mathbf{W}_c\mathbf{h}_{t-1} + \mathbf{U}_c\mathbf{x}_t) \cdot \mathbf{W}_c \cdot \mathbf{o}_{t-1} \odot \tanh(\mathbf{c}_{t-1}) \cdot \mathbf{i}_t
\end{aligned} \tag{9.44}$$

Hence, the cell state gradient is an additive function of the four terms computed in the above equation. During the backpropagation, it is possible for these additive terms to have a value of $\vec{1}$. Therefore, using LSTMs, the neural network is trained to determine when the gradient should vanish and when it should be retained by adjusting the values of the four terms.
The LSTM design is not always sufficient to prevent the issue of exploding gradients. Therefore, in successful applications of LSTM, an additional technique called

gradient clipping is often employed. Gradient clipping helps mitigate the problem of exploding gradients by constraining the magnitude of the gradients during the training process.

9.5 Example

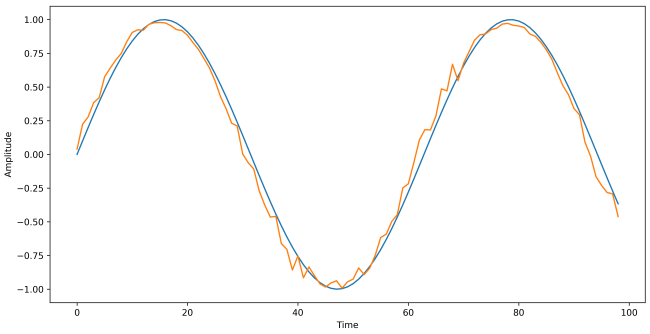


Figure 9.5: Sine wave prediction using RNN model. The curve with the orange color is the predicted one.

A Python code is provided to illustrate the learning algorithm of a regression problem based on the simple RNN model (Elman's network).

```
1
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  # Define the sine wave sequence
6  def generate_sequence(length):
7      freq = 0.1  # Frequency of the sine wave
8      x = np.arange(0, length)
9      y = np.sin(freq * x)
10     return y
11
12 # Define the ElmanRNN class
13 class ElmanRNN:
14     def __init__(self, input_size, hidden_size, output_size):
15         self.input_size = input_size
16         self.hidden_size = hidden_size
17         self.output_size = output_size
18
19         # Initialize the weights
20         self.W_xh = np.random.randn(hidden_size, input_size)
21         self.W_hh = np.random.randn(hidden_size, hidden_size)
```

```
        self.W_hy = np.random.randn(output_size, hidden_size)

        # Initialize the biases
        self.b_h = np.zeros((hidden_size, ))
        self.b_y = np.zeros((output_size, ))

    def forward(self, x):
        T = x.shape[0]
        self.h = np.zeros((T + 1, self.hidden_size))
        self.y = np.zeros((T, self.output_size))

        for t in range(T):
            self.h[t + 1] = np.tanh(np.dot(self.W_xh, x[t]) + np.dot
    (self.W_hh, self.h[t]) + self.b_h)
            self.y[t] = np.dot(self.W_hy, self.h[t + 1]) + self.b_y

        return self.y

    def backward(self, x, y, learning_rate):
        T = x.shape[0]
        dL_dW_xh = np.zeros_like(self.W_xh)
        dL_dW_hh = np.zeros_like(self.W_hh)
        dL_dW_hy = np.zeros_like(self.W_hy)
        dL_db_h = np.zeros_like(self.b_h)
        dL_db_y = np.zeros_like(self.b_y)
        dh_next = np.zeros_like(self.h[0])


        for t in reversed(range(T)):
            dL_dy = 2 * (self.y[t] - y[t])
            dL_dW_hy += np.dot(dL_dy.T, self.h[t+1].reshape(-1, 1).T
    )
            dL_db_y += dL_dy

            dh = np.dot(self.W_hy.T, dL_dy.T) + dh_next.reshape(-1,
    1)
            dh_raw = (1 - self.h[t+1] ** 2) * dh[0]
            dL_db_h += dh_raw

            dL_dW_hh += np.dot(dh_raw.reshape(-1, 1), self.h[t].
    reshape(1, -1))
            dL_dW_xh += np.dot(dh_raw.reshape(-1, 1), x[t].reshape
    (1, -1))

            dh_next = np.dot(self.W_hh.T, dh_raw.reshape(-1, 1))



        # Update weights and biases
        self.W_xh -= learning_rate * dL_dW_xh
```

```
        self.W_hh -= learning_rate * dL_dW_hh
        self.W_hy -= learning_rate * dL_dW_hy
        self.b_h -= learning_rate * dL_db_h
        self.b_y -= learning_rate * dL_db_y

    def train(self, x, y, learning_rate, num_epochs):
        for epoch in range(num_epochs):
            y_pred = self.forward(x)
            self.backward(x, y, learning_rate)

            if (epoch + 1) % 100 == 0:
                loss = np.mean((y_pred - y) ** 2)
                print(f'Epoch: {epoch+1}/{num_epochs}, Loss: {loss}')
    )

# Define the sequence length and generate the sine wave sequence
sequence_length = 100
sequence = generate_sequence(sequence_length)

# Prepare the training data
train_data = sequence[:-1]
train_target = sequence[1:]

# Reshape the training data for input to the Elman RNN
train_data = train_data.reshape(sequence_length-1, 1)
train_target = train_target.reshape(sequence_length-1, 1)

# Define hyperparameters
input_size = 1
hidden_size = 16
output_size = 1
learning_rate = 0.001
num_epochs = 10000

# Initialize the Elman RNN model
model = ElmanRNN(input_size, hidden_size, output_size)

# Train the model
model.train(train_data, train_target, learning_rate, num_epochs)

# Generate predictions for the sequence
predictions = model.forward(train_data)

# Plot the original sine wave and the predicted sine wave
plt.figure(figsize=(12, 6))
plt.plot(sequence[:-1], label='Original Sine Wave')  # Original sine
    wave
plt.plot(predictions, label='Predicted Sine Wave')  # Predicted sine
    wave
plt.xlabel('Time')  # X-axis label
```

```
plt.ylabel('Amplitude')  # Y-axis label
#plt.title('Original vs Predicted Sine Wave')  # Title of the plot
#plt.legend()  # Display legend
#plt.show()  # Display the plot
plt.savefig('rnn_sine.png', dpi=600)
```

Listing 9.1: Python example for sine wave prediction problem.

9.6 Assignment

The language modeling problem aims to predict the next word given the previous words. In this assignment, please use Keras toolkit to implement a language modeling algorithm. please use any text corpus for the learning algorithm.

Bibliography

[1] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, 1999.

[2] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proc. of IEEE*, 86(11):2278–2324, 1998.

[3] Ning Qian. On the momentum term in gradient descent learning algorithms. In *Neural networks: the official journal of the International Neural Network Society*, volume 12, pages 145–151, 1999.

[4] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7):2121–2159, 2011.

[5] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning lecture 6a: overview of mini-batch gradient descent.

[6] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[7] Pan Zhou, Jiashi Feng, Chao Ma, Caiming Xiong, Steven C. H. Hoi, and Weinan E. Towards theoretically understanding why SGD generalizes better than ADAM in deep learning. *CoRR*, abs/2010.05627, 2020.

[8] John S. Bridle. Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In Françoise Fogelman Soulié and Jeanny Hérault, editors, *Neurocomputing*, pages 227–236, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[9] E.T. Jaynes. On the rationale of maximum-entropy methods. *Proceedings of the IEEE*, 70(9):939–952, 1982.

[10] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

[11] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks, 2013.

[12] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[13] Alex Graves, Santiago Fernández, and Jürgen Schmidhuber. Bidirectional LSTM networks for improved phoneme classification and recognition. In *International Conference on Artificial Neural Networks*, pages 799–804. Springer, 2005.

[14] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics.

[15] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *NIPS 2014 Workshop on Deep Learning, December 2014*, 2014.