CS 230: Deep Learning

Hargen Zheng

December 22, 2023

Contents

Part I

Neural Networks and Deep Learning

Chapter 1

Introduction

Why it matters?

- AI is the new Electricity.
- Electricity had once transformed countless industries: transportation, manufacturing, healthcare, communications, and more.
- AI will now bring about an equally big transformation.

Courses in this sequence (Specialization):

1. Neural Networks and Deep Learning
2. Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization
3. Structuring your Machine Learning project
4. Convolutional Neural Networks (CNNs)
5. Natural Language Processing: Building sequence models (RNNs, LSTM)

1.1 What is a Neural Network?

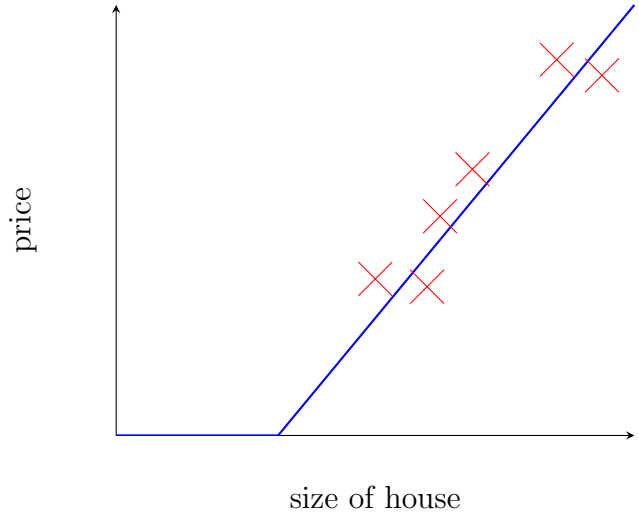

Figure 1.1: Housing Price Prediction

In the neural network literature, this function appears by a lot. This function is called a ReLU function, which stands for rectified linear units.

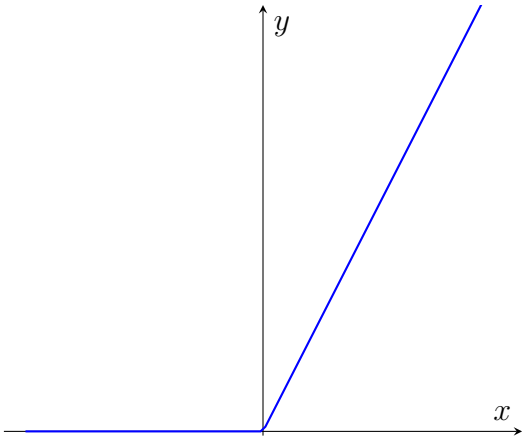

Figure 1.2: ReLU function: $\max\{0, x\}$

With the size of houses in square feet or square meters and the price of the house, we can fit a function to predict the price of a house as a function of its size.

This is the simplest neural network. We have the size of a house as input x, which goes into a node (a single "neuron"), and outputs the price y.

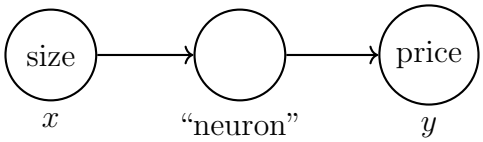

Figure 1.3: Simple Neural Network of Housing Example

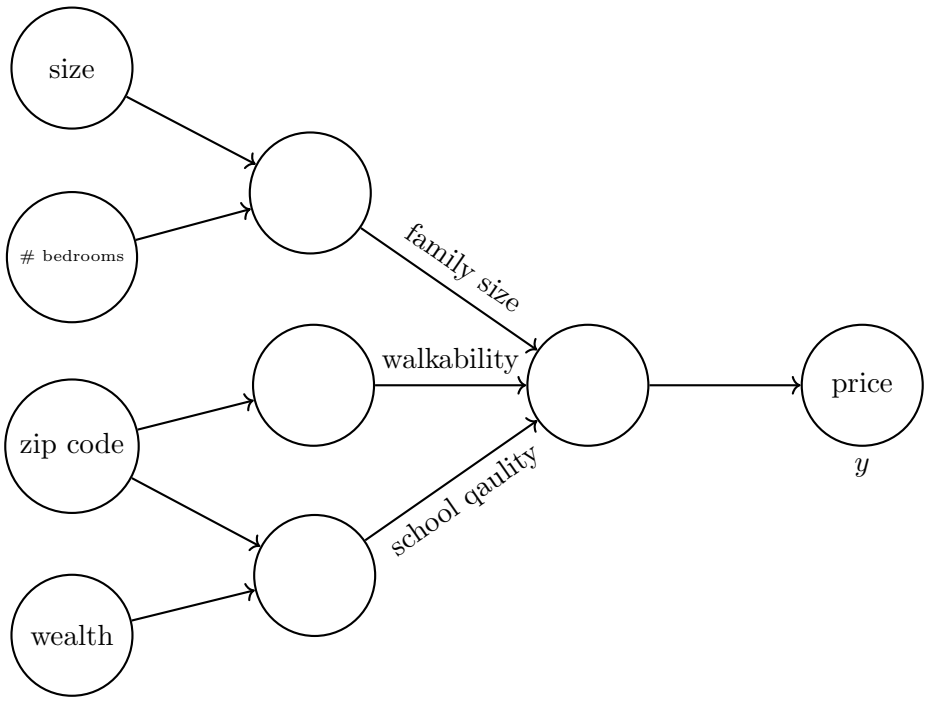

Figure 1.4: Slightly Larger Neural Network

Suppose that, instead of predicting the price of a house just from the size, we also have other features, such as the number of bedrooms, zip code, and wealth. The number of bedrooms determines whether or not a house can fit one's family size; Zip code tells walkability; and zip code and wealth tells how good the school quality is. Each of the circle could be ReLU or other nonlinear function.

We need to give the neural network the input x and the output y for a number of examples in the training set and, for all the things in the middle, neural network will figure out by itself.

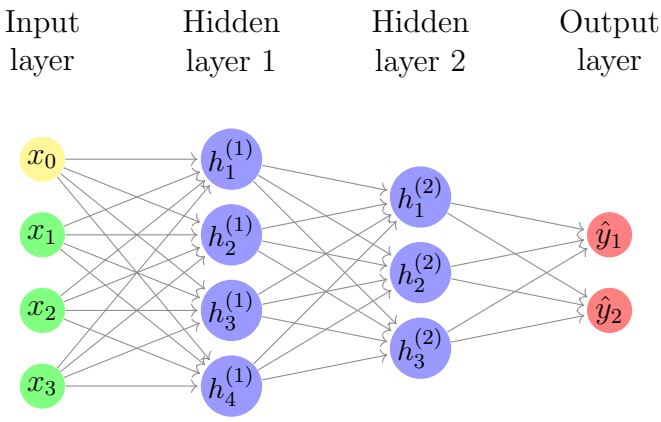

Figure 1.5: General Picture of Neural Network

1.2 Supervised Learning with Neural Networks

Supervised Learning Examples.

Input(x)	Output(y)	Application	Type of NN
Home features	Price	Real Estate	Standard
Ad, user info	Click on ad? (0/1)	Online Advertising	Standard
Image	Object $(1, \ldots, 1000)$	Photo tagging	CNN
Audio	Text transcript	Speech recognition	RNN
English	Chinese	Machine translation	RNN
Image, Radar info	Position of other cars	Autonomous driving	Custom

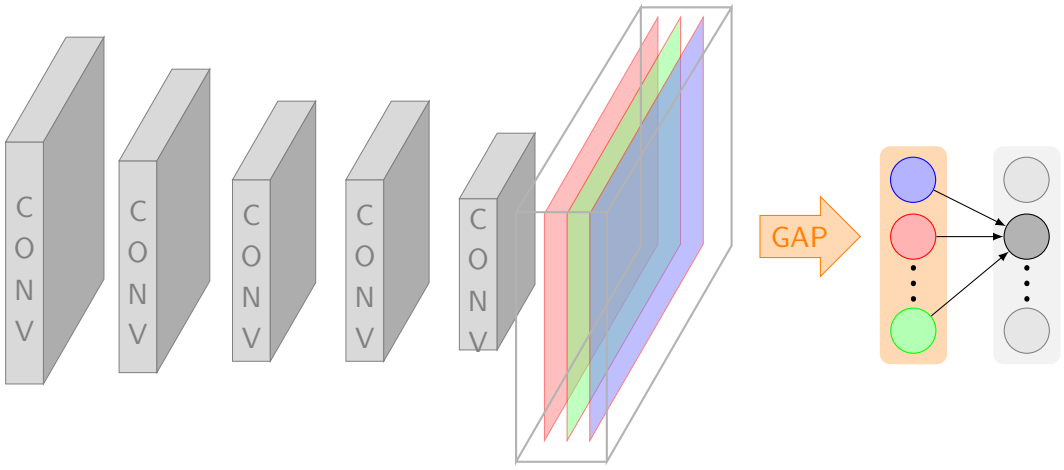

Figure 1.6: Convolutional Neural Network

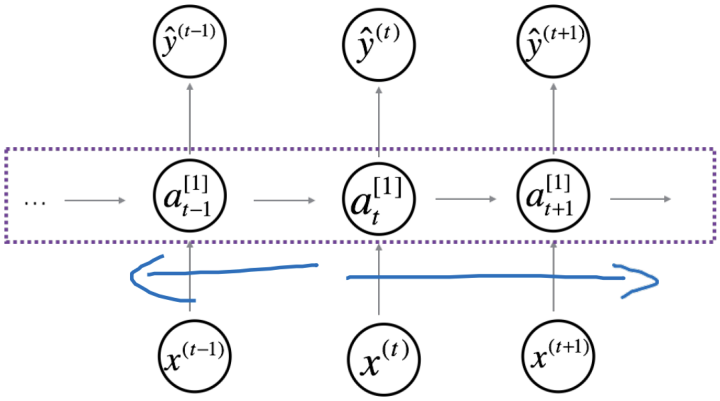

Figure 1.7: Recurrent Neural Network

There are two types of data: Structured Data and Unstructured Data. We could have the following examples for each.

Size	#bedrooms	···	Price(1000$s)
2104	3		400
1600	3		330
2400	3		369
⋮	⋮		⋮
3000	4		540

User Age	Ad ID	···	Click
41	93242		1
80	93287		0
18	87312		1
⋮	⋮		⋮
27	71244		1

Table 1.1: Structured Data

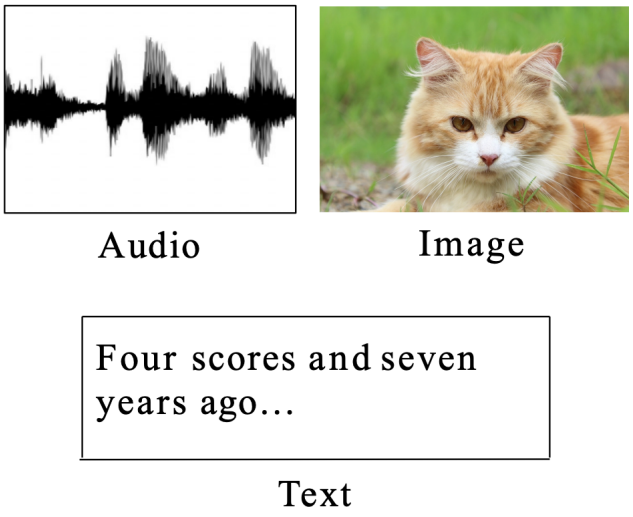

Figure 1.8: Unstructured Data

Historically, it has been much harder for computers to make sense of unstructured data compared to structured data. Thanks to the neural networks, computers are better at interpreting unstructured data as well, compared to just a few years ago.

1.3 Why is Deep Learning Taking off?

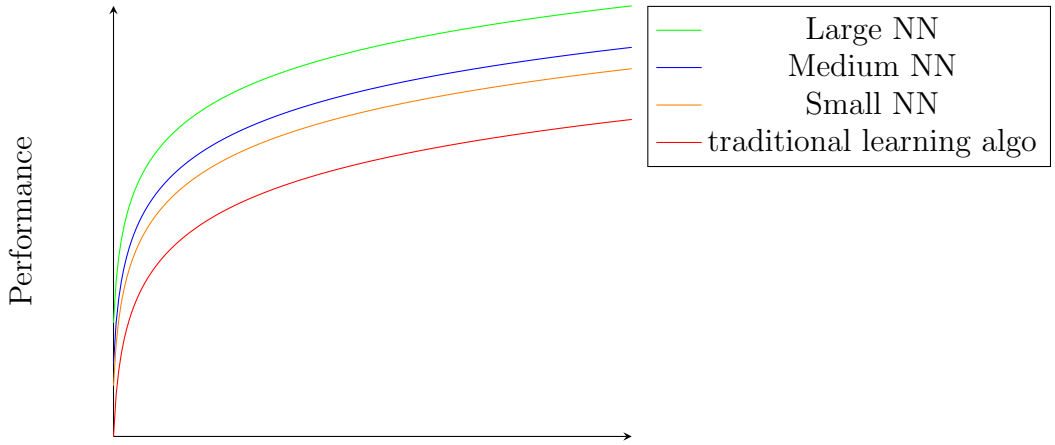

Figure 1.9: Scale drives deep learning progress

In the regime of small number of training sets, the relative ordering of the algorithms is not very well defined. If you don't have a lot of training data, it is often up to your skill at hand engineering features that determines performance. When we have large training sets – large labeled data regime in the right, we more consistently see large neural networks dominating the other approaches.

Chapter 2

Neural Networks Basics

2.1 Logistic Regression as a Neural Network

When implementing a neural network, you usually want to process entire training set without using an explicit for-loop. Also, when organizing the computation of a neural network, usually you have what's called a forward pass or forward propagation step, followed by a backward pass or backward propagation step.

2.1.1 Notation

Each training set will be comprised of m training examples:

$$\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}.$$

We denote m_{train} to be the number of training examples in the training set, and m_{test} to be the number of test examples. The ith training example is represented by a pair $(x^{(i)}, y^{(i)})$, where $x^{(i)} \in \mathbb{R}^{n_x}$ and $y^{(i)} \in \{0, 1\}$. To put all of the training examples in a more compact notation, we define matrix X as

$$X = \begin{bmatrix} | & | & & | \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(m)} \\ | & | & & | \end{bmatrix}$$

where $X \in \mathbb{R}^{n_x \times m}$. Similarly, we define Y as

$$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & \cdots & y^{(m)} \end{bmatrix}$$

where $Y \in \mathbb{R}^{1 \times m}$.

2.1.2 Binary Classification

Figure 2.1: Cat Image

You might have an input of an image and want to output a label to recognize this image as either being a cat, in which case you output 1, or non-cat, in which case you output 0. We use y to denote the output label.

To store an image, computer stores three separate matrices corresponding to the red, green, and blue color channels of this image. If the input image is 64 pixels by 64 pixels, then you would have 3 64×64 matrices corresponding to the red, green, and blue pixel intensity values for the image. We could define a feature vector x as follows:

$$x = \begin{bmatrix} | & | & | \\ red & green & blue \\ | & | & | \end{bmatrix}.$$

Suppose each matrix has dimension 64×64, then

$$n_x = 64 \times 64 \times 3 = 12288.$$

2.1.3 Logistic Regression

Given an input feature $x \in \mathbb{R}^{n_x}$, we want to find a prediction $\hat{y}$, where we want to interpret $\hat{y}$ as the probability of $y = 1$ for a given set of input features x.

$$\hat{y} = P(y = 1|x).$$

Define the weights $w \in \mathbb{R}^{n_x}$ and bias term $b \in \mathbb{R}$ in the logistic regression. It turns out, when implementing the neural networks, it is better to keep parameters w and b separate, instead of putting them together as $\theta \in \mathbb{R}^{n_x+1}$.

In linear regression, we would use $\hat{y} = w^T x + b$. However, this is not good for binary classification because the prediction could be much bigger than 1 or even negative, which does not make sense for probability. Therefore, we apply the sigmoid function, where

$$\hat{y} = \sigma(w^T x + b) = \sigma(z).$$

The formula of the sigmoid function is given as follows:

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

If z is very large, then e^{-z} will be close to zero, so $\sigma(z) \approx 1$. Conversely, if z is very small or a very large negative number, then $\sigma(z) \approx 0$.

The graph of the sigmoid function is given below:

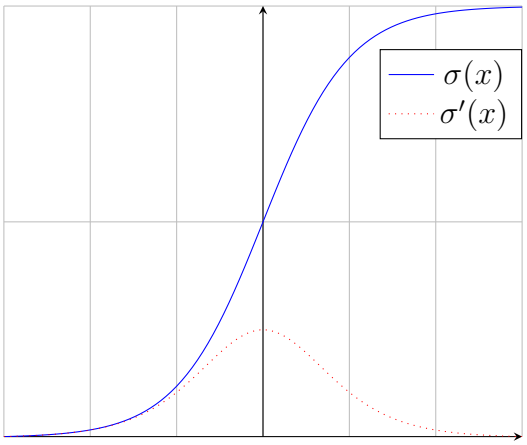

Figure 2.2: Sigmoid Function

2.1.4 Logistic Regression Cost Function

Logistic regression is a supervised learning algorithm, where given training examples $\{(x^{(1)}, y^{(1)}, \ldots, (x^{(m)}, y^{(m)})\}$, we want to find parameters w and b such that $\hat{y}^{(i)} \approx y^{(i)}$.

MSE is not used as the loss function because when learning parameters, the optimization problem becomes non-convex, where we end up with multiple local optima, so gradient descent may not find a global optimum. The loss function is defined to measure how good our output $\hat{y}$ is when the true label is y. Instead, the loss (error) function for logistic regression could be represented by

$$\mathcal{L}(\hat{y}, y) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})).$$

The intuition behind is that when considering the squared error cost function, we want the squared error to be as small as possible. Similarly, we want to loss function for logistic regression to be as small as possible.

If $y = 1$, then the loss function ends up with

$$\mathcal{L}(\hat{y}, 1) = -\log(\hat{y}).$$

In this case, we want $\log(\hat{y})$ to be large $\Leftrightarrow$ want $\hat{y}$ to be large.

If $y = 0$, then

$$\mathcal{L}(\hat{y}, 0) = -\log(1 - \hat{y}).$$

In this case, we want $\log(1 - \hat{y})$ to be large $\Leftrightarrow$ want $\hat{y}$ to be small.

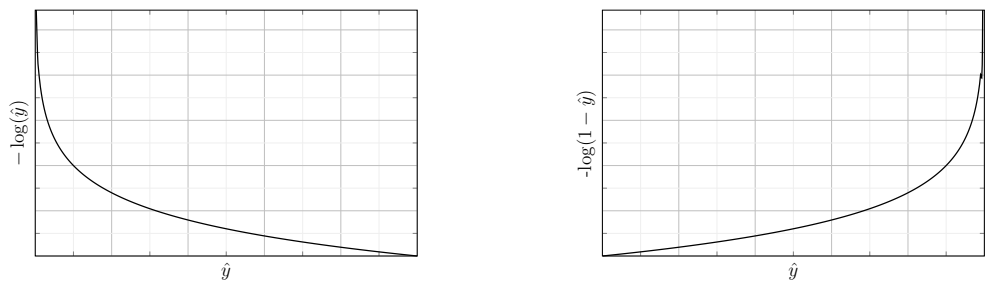

Figure 2.3: A binary classifier's loss function based on if $y = 1$ or $y = 0$.

The loss function is defined with respect to a single training example. The cost function measures how well the model is doing on the entire training set, which is given as

$$\begin{aligned} J(w,b) &= \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) \\ &= -\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}\log(\hat{y}^{(i)}) + (1-y^{(i)})\log(1-\hat{y}^{(i)})). \end{aligned}$$

Interpretation of the Cost Function. We interpret $\hat{y}$ to be the probability of $y = 1$ given a set of input features x. This means that

- If $y = 1$, $P(y|x) = \hat{y}$
- If $y = 0$, $P(y|x) = 1 - \hat{y}$

We can summarize the two equations as follows:

$$P(y|x) = \hat{y}^{y}(1-\hat{y})^{1-y}.$$

Since the log function is a strictly monotonically increasing function, maximizing $\log(P(y|x))$ would give a similar result as maximizing $P(y|x)$.

$$\begin{aligned} \log(P(y|x)) &= \log(\hat{y}^{y}(1-\hat{y})^{1-y}) \\ &= y\log(\hat{y}) + (1-y)\log(1-\hat{y}) \\ &= -\mathcal{L}(\hat{y}, y). \end{aligned}$$

Therefore, minimizing the loss means maximizing the probability.

The above was for one training example. For m examples, we can express the probability as follows:

$$\begin{aligned} \log(P(\text{labels in training set})) &= \log(\prod_{i=1}^{m} P(y^{(i)}|x^{(i)})) \\ &= \sum_{i=1}^{m}\log(P(y^{(i)}|x^{(i)}) \\ &= -\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) \end{aligned}$$

The cost is defined as

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}).$$

We get rid of the negative sign as now we want to minimize the cost. $\frac{1}{m}$ is added in the front to make sure our quantities are at a better scale. To minimization of the cost function carries out maximum likelihood estimation with the logistic regression model, under the assumption that our training examples are i.i.d (independent and identically distributed).

2.1.5 Gradient Descent

We want to find w, b that minimize $J(w, b)$. Our cost function $J(w, b)$ is a single big bowl, which is a convex function, which is one of the huge reasons why we use this particular cost function J for logistic regression.

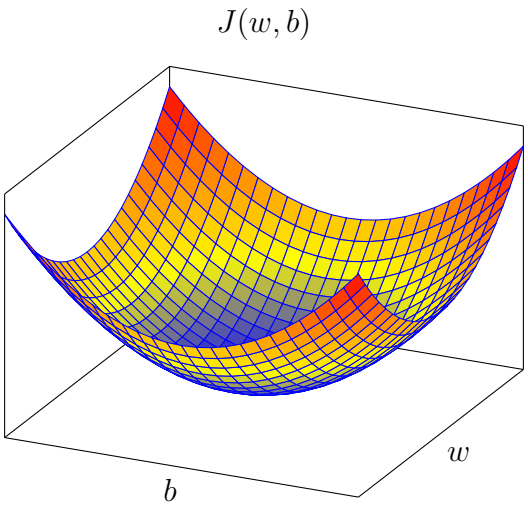

Figure 2.4: Convex Cost Function

For logistic regression, almost any initialization method works. Though random initialization works, people don't usually do that for logistic regression. Instead, we generally initialize the weights to 0. Gradient descent takes a step in the steepest downhill direction. After iterations, we converge to the global optimum or get close to the global optimum. In one dimension, for the parameter w, we can define the gradient descent algorithm as follows

$$w := w - \alpha \frac{\partial J(w, b)}{\partial w}$$
$$b := b - \alpha \frac{\partial J(w, b)}{\partial b}$$

where α is the learning rate that control the size of steps and $\frac{\partial J(w)}{\partial w}, \frac{\partial J(w,b)}{\partial b}$ are the changes we make for the parameter w, b.

2.1.6 Computation Graph

The computations of neural network are organized in terms of a forward pass step, in which we compute the output of the neural network, followed by a backward propagation step, which we use to compute the gradients. The computation graph explains why it is organized this way.

Suppose we want the compute the function

$$J(a, b, c) = 3(a + bc).$$

We can break down the computation into three steps:

1. $u = bc$
2. $v = a + u$
3. $J = 3v$

This could be represented by the computation graph below:

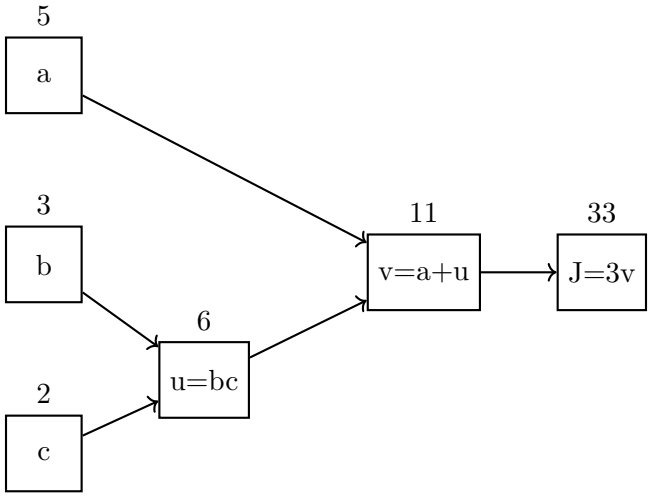

Figure 2.5: Computation Graph for $J(a, b, c) = 3(a + bc)$

Suppose $a = 5, b = 3, c = 2$, then $J(a, b, c) = 3(a + bc) = 3(5 + 3 \times 2) = 33$.

2.1.7 Derivatives with a Computation Graph

Consider the computation graph 2.5. Suppose we want to compute the derivative of J with respect to v. Since we know $J = 3v$, we can take the derivative as follows:

$$\frac{\partial J}{\partial v} = \frac{\partial}{\partial v}(3v) = 3.$$

Suppose then we want to compute the derivative of J with respect to a. We can apply the chain rule as follows:

$$\frac{\partial J}{\partial a} = \frac{\partial J}{\partial v} \cdot \frac{\partial v}{\partial a} = 3 \cdot \frac{\partial}{\partial a}(a + u) = 3 \cdot 1 = 3.$$

For brevity, we could use "da'' to represent $\frac{\partial J}{\partial a}$. Similarly, we can use "dv'' to represent $\frac{\partial J}{\partial v}$.

To find the derivative of J with respect to u, we could do similar computation as $\frac{\partial J}{\partial a}$ as follows:

$$\frac{\partial J}{\partial u} = \frac{\partial J}{\partial v} \cdot \frac{\partial v}{\partial u} = 3 \cdot 1 = 3.$$

$\frac{\partial J}{\partial b}$ could be compute as follows:

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial u} \cdot \frac{\partial u}{\partial b} = 3 \cdot c = 3 \cdot 2 = 6.$$

Similarly, we can compute

$$\frac{\partial J}{\partial c} = \frac{\partial J}{\partial u} \cdot \frac{\partial u}{\partial c} = 3 \cdot b = 3 \cdot 3 = 9.$$

The most efficient way to compute all these derivatives is through a right-to-left computation following the direction of the arrows below:

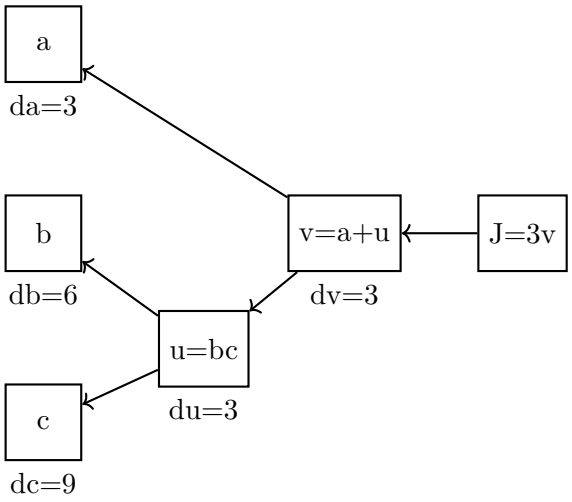

Figure 2.6: Backward Propagation for $J(a, b, c) = 3(a + bc)$

2.1.8 Logistic Regression Gradient Descent

This section will cover how to compute derivatives to implement gradient descent for logistic regression.

To recap, logistic regression is set up as follows:

$$\begin{aligned} z &= w^T x + b \\ \hat{y} &= a = \sigma(z) \\ \mathcal{L}(a, y) &= -(y \log(a) + (1 - y) \log(1 - a)) \end{aligned}$$

This can be represented by the computation graph below:

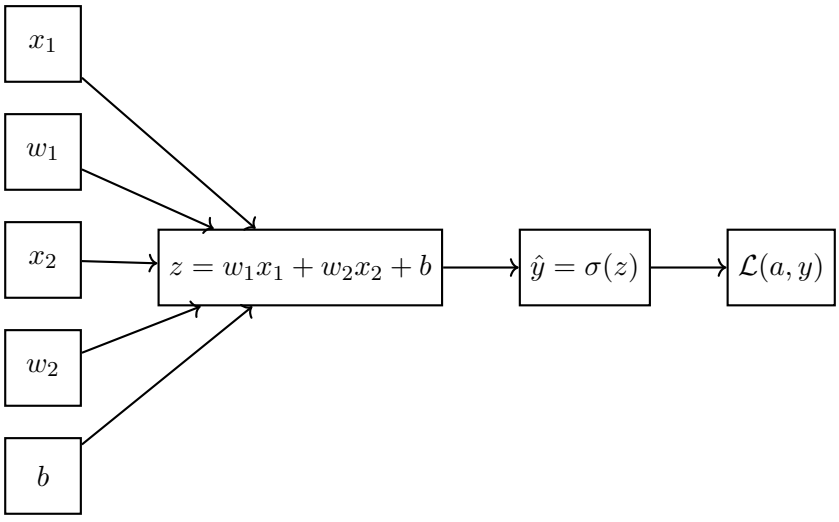

Figure 2.7: Computation Graph for Logistic Regression

In this case, we can compute "da" as follows:

$$da = \frac{\partial \mathcal{L}(a,y)}{\partial a} = -y \cdot \frac{1}{a} - (1-y) \cdot \frac{1}{1-a} \cdot (-1) = -\frac{y}{a} + \frac{1-y}{1-a}.$$

We can give both terms the same denominator and clean up as follows:

$$\frac{\partial \mathcal{L}(a,y)}{\partial a} = \frac{-y(1-a)}{a(1-a)} + \frac{a(1-y)}{a(1-a)} = \frac{-y + ay + a - ay}{a(1-a)} = \frac{a-y}{a(1-a)}.$$

Then, we can go backwards and compute $\frac{\partial \mathcal{L}(a,y)}{\partial z}$. We know that the derivative of a sigmoid function has the form $\frac{\partial}{\partial z}\sigma(z) = \sigma(z)(1-\sigma(z))$. In this case $\sigma(z) = a$, as we have defined, so $\frac{\partial a}{\partial z} = a(1-a)$. Therefore,

$$dz = \frac{\partial \mathcal{L}(a,y)}{\partial z} = \frac{\partial \mathcal{L}}{\partial a} \cdot \frac{\partial a}{\partial z} = \frac{a-y}{a(1-a)} \times a(1-a) = a - y.$$

Finally, we do the backward propagation for the input layer as follows:

$$\begin{aligned}
\frac{\partial \mathcal{L}(a,y)}{\partial w_1} &= x_1 \cdot \frac{\partial \mathcal{L}(a,y)}{\partial z} = x_1 \cdot (a-y) \\
\frac{\partial \mathcal{L}(a,y)}{\partial w_2} &= x_2 \cdot \frac{\partial \mathcal{L}(a,y)}{\partial z} = x_2 \cdot (a-y) \\
\frac{\partial \mathcal{L}(a,y)}{\partial x_1} &= w_1 \cdot \frac{\partial \mathcal{L}(a,y)}{\partial z} = w_1 \cdot (a-y) \\
\frac{\partial \mathcal{L}(a,y)}{\partial x_2} &= w_2 \cdot \frac{\partial \mathcal{L}(a,y)}{\partial z} = w_2 \cdot (a-y) \\
\frac{\partial \mathcal{L}(a,y)}{\partial b} &= 1 \cdot \frac{\partial \mathcal{L}(a,y)}{\partial z} = (a-y)
\end{aligned}$$

2.1.9 Gradient Descent on m examples

Recall that the cost function for m training examples is given by

$$J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y),$$

where $a^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^T x^{(i)} + b)$.

The cost function is the average of loss values for each training example. It turns out that derivative works similarly and we can compute the derivative of $J(w,b)$ with respect to w_1 as follows:

$$\frac{\partial}{\partial w_1} J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial w_i} \mathcal{L}(a^{(i)}, y^{(i)}) = \frac{1}{m} \sum_{i=1}^{m} x_1 \cdot (a^{(i)} - y^{(i)}).$$

Now, let's formalize the algorithm for the logistic regression gradient descent.

To implement logistic regression this way, we have two for-loops. The first for-loop is to iterate through all training examples. The second for-loop is to iterate through all the features. In the example above, we only have features $w^{(1)}, w^{(2)}$. If we have n features instead, then we need to iterate through all of them.

However, when implementing deep learning algorithms, the explicit for-loop makes the algorithms less efficient. The vectorization technique allows us to get rid of the explicit for-loops, which allows us to scale to larger datasets.

Algorithm 1 Gradient Descent for Logistic Regression

Initialize $J = 0; dw_1 = 0; dw_2 = 0; db = 0$
for $i = 1$ to m **do**
  $z^{(i)} \leftarrow w^T x^{(i)} + b$
  $a^{(i)} \leftarrow \sigma(z^{(i)})$
  $J \leftarrow J + [-(y^{(i)} \log(a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)}))]$
  $dz^{(i)} \leftarrow a^{(i)} - y^{(i)}$ ▷ Superscript refers to one training example
  $dw_1 \leftarrow dw_1 + x_1^{(i)} dz^{(i)}$ ▷ Here we assume $n_x = 2$
  $dw_2 \leftarrow dw_2 + x_2^{(i)} dz^{(i)}$ ▷ No superscript i as it's accumulative
  $db \leftarrow db + dz^{(i)}$
end for
$J \leftarrow J/m$
$dw_1 \leftarrow dw_1/m; dw_2 \leftarrow dw_2/m; db \leftarrow db/m$

2.2 Python and Vectorization

2.2.1 Vectorization

The aim of vectorization is to remove explicit for-loops in code. In the deep learning era, we often train on relatively large dataset because that's when deep learning algorithms tend to shine. In deep learning, the ability to perform vectorization has become a key skills.

Recall that in logistic regression, we compute $z = w^T x + b$, where $w, x \in \mathbb{R}^{n_x}$. For non-vectorized implementation, we compute z as

$$z = \sum_{i=1}^{n_x} w[i] \times x[i], ; z+ = b$$

The vectorized implementation is very fast:

$$z = np.dot(w, x) + b.$$

SIMD (single instruction multiple data), in both GPU and CPU, enables built-in functions, such as $np.dot()$, to take much better advantage of parallelism to do computations much faster.

Neural network programming guideline: whenever possible, avoid explicit for-loops.

If we want to compute $u = Av$, then by the definition of matrix multiplication,

$$u_i = \sum_j A_{ij} v_j.$$

The vectorized implementation would be

$$u = np.dot(A, v).$$

Suppose we want to apply the exponential operation on every element of a vector

$$v = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}.$$

We can apply the vectorization technique as follows:

$$u = np.exp(v).$$

Similarly, we have $np.log(v), np.abs(v), np.maximum(v, 0), v**2$, etc. to apply the vectorization technique. We can revise the original gradient descent algorithm and use the vectorization technique as follows

Algorithm 2 Gradient Descent for Logistic Regression with Vectorization

Initialize $J = 0; dw = np.zeros((n_x, 1)); db = 0$
for $i = 1$ to m **do**
  $z^{(i)} \leftarrow w^T x^{(i)} + b$
  $a^{(i)} \leftarrow \sigma(z^{(i)})$
  $J \leftarrow J + [-(y^{(i)} \log(a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)}))]$
  $dz^{(i)} \leftarrow a^{(i)} - y^{(i)}$
  $dw \leftarrow dw + x^{(i)} dz^{(i)}$ ▷ Get rid of the for-loop
  $db \leftarrow db + dz^{(i)}$
end for
$J \leftarrow J/m$
$dw \leftarrow dw/m; db \leftarrow db/m$

2.2.2 Vectorizing Logistic Regression

For logistic regression, we need to compute $a^{(i)} = \sigma(z^{(i)})$ m times, where $z^{(i)} = w^T x^{(i)} + b$. There is a way to compute without using explicit for-loops. Recall that we define

$$X = \begin{bmatrix} | & | & & | \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(m)} \\ | & | & & | \end{bmatrix},$$

where $X \in \mathbb{R}^{n_x \times m}$.

We want to compute all $z^{(i)}$'s at the same time.

From math, we know that

$$\begin{aligned} Z &= \begin{bmatrix} z^{(1)} & z^{(2)} & \cdots & z^{(m)} \end{bmatrix} \\ &= w^T X + \begin{bmatrix} b & b & \cdots & b \end{bmatrix} \\ &= \begin{bmatrix} w^T x^{(1)} + b & w^T x^{(2)} + b & \cdots & w^T x^{(m)} + b \end{bmatrix} \end{aligned}$$

In order to implement this in NumPy, we can do

$$Z = np.dot(w.T, X) + b,$$

where $b \in \mathbb{R}$. Python will expand this real number to a $1 \times m$ vector by **broadcasting**.

Similarly, we can stack lower case $a^{(i)}$'s horizontally to obtain capital A as follows:

$$A = \begin{bmatrix} a^{(1)} & a^{(2)} & \cdots & a^{(m)} \end{bmatrix} = \sigma(Z).$$

With Z being a $1 \times m$ vector, we can apply the sigmoid function to each element of Z by

$$A = \frac{1}{1 + np.exp(-Z)}.$$

2.2.3 Vectorizing Logistic Regression's Gradient Computation

Recall that we compute individual dz's as $dz^{(i)} = a^{(i)} - y^{(i)}$ for the ith training example. Now, let's define

$$dZ = \begin{bmatrix} dz^{(1)} & dz^{(2)} & \cdots & dz^{(m)} \end{bmatrix},$$

where $dz^{(i)}$'s are stacked horizontally and $dZ \in \mathbb{R}^{1\times m}$.

Since we have

$$A = \begin{bmatrix} a^{(1)} & \cdots & a^{(m)} \end{bmatrix}, \; Y = \begin{bmatrix} y^{(1)} & \cdots & y^{(m)} \end{bmatrix}$$

we can compute dZ as

$$dZ = A - Y = \begin{bmatrix} a^{(1)} - y^{(1)} & a^{(2)} - y^{(2)} & \cdots & a^{(m)} - y^{(m)} \end{bmatrix}.$$

We can compute db with vectorization as

$$\begin{aligned} db &= \frac{1}{m}\sum_{i=1}^{m} dz^{(i)} \\ &= \frac{1}{m} \times np.sum(dZ). \end{aligned}$$

Similarly, we can compute dw with vectorization as

$$\begin{aligned} dw &= \frac{1}{m} X dZ^T \\ &= \frac{1}{m} \begin{bmatrix} | & | & & | \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(m)} \\ | & | & & | \end{bmatrix} \begin{bmatrix} dz^{(1)} \\ \vdots \\ dz^{(m)} \end{bmatrix} \\ &= \frac{1}{m} \begin{bmatrix} x^{(1)} dz^{(1)} + \cdots + x^{(m)} z^{(m)} \end{bmatrix}, \end{aligned}$$

where $dw \in \mathbb{R}^{n_x \times 1}$.

2.2.4 Broadcasting in Python

The following matrix shows the calories from Carbs, Proteins, Fats in 100g of different foods:

	Apples	Beef	Eggs	Potatoes
Carb	56.0	0.0	4.4	68.0
Protein	1.2	104.0	52.0	8.0
Fat	1.8	135.0	99.0	0.9

Suppose we want to calculate the percentage of calories from Carb, Protein, and Fat for each of the four foods. For example, for apples, the percentage of calories for Carb is

$$\frac{56.0}{59.0} \approx 94.9\%.$$

We could do this without explicit for-loops. Suppose the matrix is denoted by A, where $A \in \mathbb{R}^{3\times 4}$.

To sum each column vertically, we can use

$$cal = A.sum(axis = 0).$$

$axis = 0$ refers to the vertical columns inside the matrix, whereas $axis = 1$ will sum horizontally. The $cal = cal.reshape(1, 4)$ operation would convert s to a one by four vector, which is very cheap to call as it's runtime is $\mathcal{O}(1)$.

Then, we can compute percentages by

$$percentage = 100 * A/(cal.reshape(1, 4)),$$

where we divide a $(3, 4)$ matrix by a $(1, 4)$ vector. This will divide each of the three elements in a column by the value in the corresponding column of $cal.reshape(1, 4)$.

A few more examples follows are provided below. Suppose we want to add 100 to every element of the vector $\begin{bmatrix}1\\2\\3\\4\end{bmatrix}$. We can do

$$\begin{bmatrix}1\\2\\3\\4\end{bmatrix} + 100.$$

Python will convert the number 100 into the vector

$$\begin{bmatrix}100\\100\\100\\100\end{bmatrix}$$

and so

$$\begin{bmatrix}1\\2\\3\\4\end{bmatrix} + 100 \Rightarrow \begin{bmatrix}1\\2\\3\\4\end{bmatrix} + \begin{bmatrix}100\\100\\100\\100\end{bmatrix} = \begin{bmatrix}101\\102\\103\\104\end{bmatrix}.$$

If we add a (m, n) matrix by a $(1, n)$ vector, Python will copy the vector m times to convert it into an (m, n) matrix, and then add two matrices together.

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} + \begin{bmatrix}100 & 200 & 300\end{bmatrix} \Rightarrow \begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} + \begin{bmatrix}100 & 200 & 300\\100 & 200 & 300\end{bmatrix} = \begin{bmatrix}101 & 202 & 303\\104 & 205 & 306\end{bmatrix}.$$

Now, suppose we add a (m, n) matrix by a $(m, 1)$ vector. Then, Python will copy the vector n times horizontally and form a (m, n) matrix.

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} + \begin{bmatrix}100\\200\end{bmatrix} \Rightarrow \begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} + \begin{bmatrix}100 & 100 & 100\\200 & 200 & 200\end{bmatrix} = \begin{bmatrix}101 & 102 & 103\\204 & 205 & 206\end{bmatrix}.$$

2.2.5 A Note on Python/NumPy Vectors

Let's set

$$a = np.random.randn(5),$$

which creates five random Gaussian variables stored in array a. It turns out if we do $print(a.shape)$, we get $(5,)$. This is called a **rank one array**, which is neither a row vector nor a column vector. In this case, $a.T$ and a look the same. When we call $np.dot(a, a^T)$, we will get a single number, instead of a matrix generated by the outer product.

When doing neural network computations, it is recommended not to use rank one array. Instead, we can create a $(5, 1)$ vector as follows:

$$a = np.random.randn(5, 1),$$

which generates a column vector. Now, if we call $a.T$, we will get a row vector. This way, the behaviors of the vector are easier to understand.

When dealing with a matrix with unknown dimensions, we can use an assertion statement to make sure the shape is as desired:

$$assert(a.shape == (5, 1)).$$

Also, if we ultimately encounter the rank one array, we can always reshape it to the column vector as follows:

$$a = a.reshape((5, 1)),$$

so it behaves more consistently as a column vector, or a row vector.

Chapter 3

Shallow Neural Networks

3.1 Neural Networks Overview

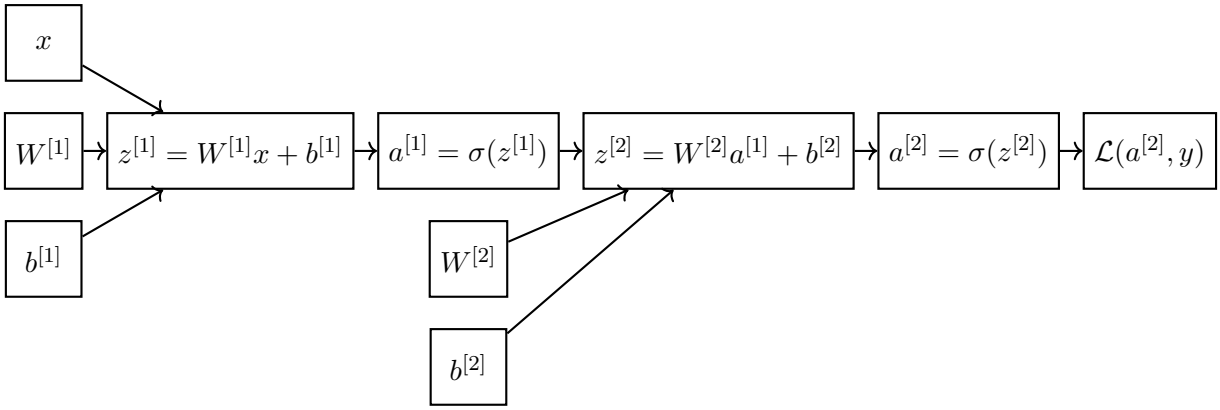

Figure 3.1: Shallow Neural Network

After forward pass, we do backward propagation to compute the derivatives

$$da^{[2]}, dz^{[2]}, dW^{[2]}, db^{[2]}, da^{[1]}, dz^{[1]}, dW^{[1]}, db^{[1]}, dx.$$

This is basically taking logistic regression and repeating it twice.

3.2 Neural Network Representation

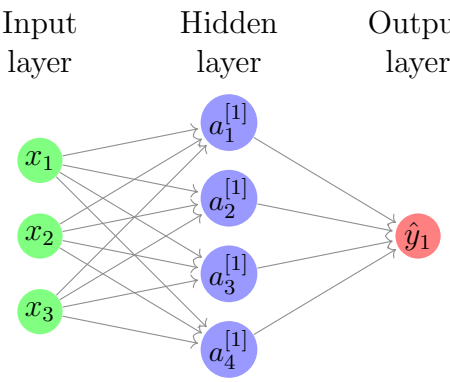

Figure 3.2: Neural Network Representation

In a neural network that we train with supervised learning, the training set contains values of the inputs x as well as the target outputs y. The term "Hidden layer" refers to the fact that in the training set, the true values for those nodes in the middle are not observed.

Alternatively, we can use $a^{[0]}$, which stands for the activation, to represent the input layer X. Then, $a^{[1]}$ can be used to represent the hidden layer, where

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix}.$$

Finally, the output layer could be denoted by $a^{[2]}$.

The figure 3.2 is called 2 layer Neural Network. The reason is that when we count layers in neural networks, we do not count the input layer, so hidden layer is layer 1 and output layer is layer 2. Input layer is referred to as layer 0. For the hidden layer, there are parameters $w^{[1]}, b^{[1]}$ associated with it, where $w^{[1]} \in \mathbb{R}^{4\times 3}$ and $b^{[1]} \in \mathbb{R}^{4\times 1}$. Similarly, the output layer has parameters $w^{[2]}, b^{[2]}$ associated with it, where $w^{[2]} \in \mathbb{R}^{1\times 4}$ and $b^{[2]} \in \mathbb{R}^{1\times 1}$.