Neural Networks and Deep Learning

by DeepLearningAI (YouTube)

1. What is a neural network?

1.1. Neuron

The purpose of a neuron is to accept input, perform computations and return the result.

For example, A neuron can input the size of house from some sample data and predict the price of the house using that data. While it may be convenient to fit the data to a line (linear regression), most statistics follow a non-linear curve.

One non-linear relationship can be described by the **Rectified Linear Unit** (ReLU) function. This function returns a straight line when the input is positive, but zero otherwise.

$$\mathrm{ReLU}(x) := \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$$

1.2. Neural Network

A neural network comprises many neurons arranged in different layers. Some of these layers include:

- **Input** (x): All inputs to the network are provided in this layer.
- **Hidden**: These layers are responsible for all the computation in a neural network.
- **Output** (y): The result of all the computations from the hidden layer are returned in the output layer.

If every neuron in one layer is connected to every other neuron in its subsequent layer for every pair of consecutive layers in the network, the network is said to be *densely connected*.

If given given enough data about x and y, a neural network is really good at determining accurate functions that map x to y. This is discussed further in §3.

2. Supervised Learning

2.1. Types of Neural Network Architectures

Different neural network architectures have different applications. Some of them are listed as follows:

- **Standard neural networks (NNs)**: These are used for working with a finite set of data. They follow the same architecture as described in §1.2.
- **Recurrent neural networks (RNNs)**: These are specialized for data that is input as a one-dimensional sequence, *i.e.* over a period of time, such as text or audio.
- **Convolutional neural networks (CNNs)**: These are used for inputting and performing computations on image data.
- **Hybrid networks**: These networks are used for complex data, such as image as well as audio or text.

2.2. Types of data

There are two kinds of data a neural network operates on, **Structured** and **Unstructured** data.

Structured data	Unstructured data
Structured data refers to data that is organized in a tabular database. The features of a structured dataset can be easily understood by a computer.	Unstructured data refers to data whose features cannot be easily interpreted by a computer.
An example of a structured dataset is the analytics of an advert. In structured data, each parameter and statistic has a well-defined value.	Examples of unstructured dataset include audio, image or text, where the meaningful features need to be extracted instead (such as audio cues, pixel data or words in a text).

The objective of deep learning is to make the interpretation and computation of unstructured data easier with technologies such as *Speech Recognition*, *Image Recognition* and *Natural Language Processing*.

3. Recent Improvements in Deep Learning

When we plot the performance of deep learning algorithms with respect to the amount of labelled data, we observe that traditional learning models such as Support Vector Machines and Logistic Regression tend to plateau (horizontal asymptote).

However, it is recently observed that larger neural networks, consisting of numerous neurons and connections, have shown a better performance increase with large amounts of data. Some of the reasons are discussed below:

3.1. Change from sigmoid to ReLU

Traditional algorithms used sigmoid as their activation functions. The problem with sigmoid is that at really large absolute values, the gradient (slope) of the sigmoid function is very close to zero, which causes the algorithm *Gradient descent* to compute very slowly.

This issue is resolved since newer algorithms use the ReLU function (discussed in §1.1.) which has a constant non-zero gradient for positive values.

4. Binary Classification

Binary classification means to separate input data into two separate categories.

Input features are denoted by a vector $\mathbf{x}$, while output labels are represented by y (0 or 1). The number of dimensions of the input feature vector x is denoted by n_x.

$$\therefore n_x := \|\mathbf{x}\|$$

4.1. Notation

A single training example is represented by a pair $(\mathbf{x}, y)$, where $\mathbf{x} \in \mathbb{R}^{n_x}$ and $y \in \{0, 1\}$.

Our training set consists of m training examples. The i^{th} training example is represented by $(\mathbf{x}^{(i)}, y^{(i)})$.

The entire training set could be encapsulated by a matrix $X \in \mathcal{M}_{n_x \times m}(\mathbb{R})$ as follows:

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

The labels can similarly be encapsulated by a matrix $Y \in \mathcal{M}_{1 \times m}(\mathbb{R})$:

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

5. Logistic Regression

Logistic Regression is an algorithm used to solve the problem of binary classification.

Given an input feature vector $\mathbf{x}$, we want the algorithm to output the prediction $\hat{y}$, which is an estimate of the label (0 or 1). It is defined as,

$$\hat{y} := P(y = 1 \mid \mathbf{x})$$

We use the parameters $\mathbf{w} \in \mathbb{R}^{n_x}$ and $b \in \mathbb{R}$ in logistic regression.

Since $\hat{y}$ represents probabilities, we want it to be within 0 and 1. To overcome this problem, we use the sigmoid function.

5.1. Sigmoid function

The sigmoid is a logistic function that is defined as follows:

$$\sigma : \mathbb{R} \to (0,1)$$

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

The sigmoid is used to convert a set of real values to within the range $(0,1)$. This is illustrated by the following limits:

$$\lim_{z\to\infty} \sigma(z) = \lim_{z\to\infty} \frac{1}{1+e^{-z}} = \frac{1}{1+0} = 1$$

$$\lim_{z\to-\infty} \sigma(z) = \lim_{z\to-\infty} \frac{1}{1+e^{-z}} = \lim_{z\to\infty} \frac{1}{1+e^{z}} = 0$$

Therefore, we can define the output prediction $\hat{y}$ using sigmoid as follows:

$$\hat{y} := \sigma(\mathbf{w}^T \mathbf{x}^{(i)} + b)$$

where $\mathbf{w}^T$ refers to the *transpose* of $\mathbf{w}$, where $\mathbf{w}^T \in \mathcal{M}_{1\times n_x}(\mathbb{R})$.

Note: We could also write $\hat{y} := \sigma(\mathbf{w} \cdot \mathbf{x}^{(i)} + b)$, where '$\cdot$' represents the scalar (dot) product.

5.2. Error functions

Given a training set $\{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \ldots, (\mathbf{x}^{(m)}, y^{(m)})\}$, we want a model that can predict $\hat{y}^{(i)} \approx y^{(i)}$.

For brevity, let us define $z^{(i)} := \mathbf{w}^T \mathbf{x}^{(i)} + b$ such that $\hat{y}^{(i)} := \sigma(z)$.

5.2.1 Loss function

A loss function is applied to a single training example.

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

The loss function defined above is minimized if $\hat{y}^{(i)} \approx y^{(i)}$, which is the objective of our model. For an ideal model, $\hat{y}^{(i)} = y^{(i)} \implies \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = 0 \; \forall \, i$.

5.2.2. Cost function

A cost function is used to measure the loss across many training examples.

$$J(\mathbf{w}, b) := \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m} \sum_{i=1}^{m} [y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log{(1 - \hat{y}^{(i)})}]$$

5.3. Gradient Descent

The objective of gradient descent is to find values for w and b so as to minimize the cost function described in §5.2.2.

Repeat until minima is reached:

1. $w_i := w_i - \alpha \frac{\partial J}{\partial w_i}(\mathbf{w}, b) \; \forall \, i$
2. $b := b - \alpha \frac{\partial J}{\partial b}(\mathbf{w}, b)$

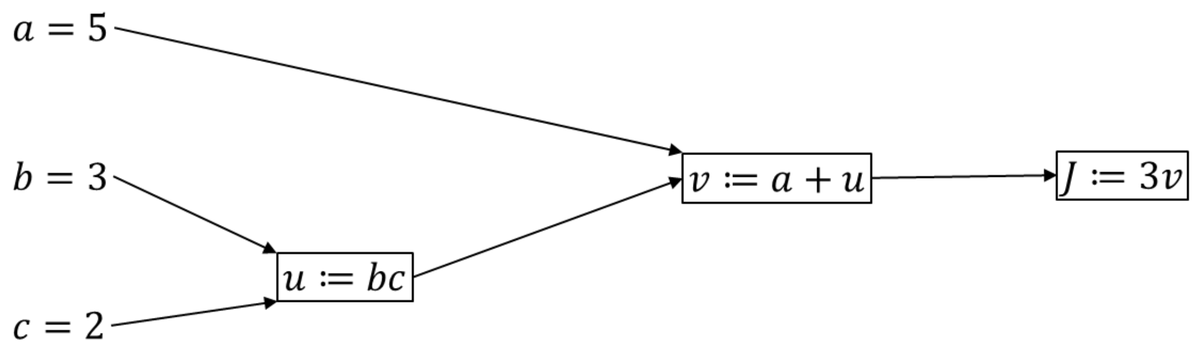
The computation of gradient descent is discussed in §5.4.2.

5.4. Computation Graph

Consider the function $J(a, b, c) = 3(a + bc)$. To compute $J(5, 3, 2)$, we have to perform the following steps:

1. Compute $u := bc = 3 \cdot 2 = 6.$
2. Compute $v := a + u = 5 + 6 = 11.$
3. Compute $J := 3v = 3(11) = 33.$

These computations can be represented by the following graph:



Propagating forward the graph, we can compute J using the values of a, b and c.

5.4.1. Computing Derivatives

To find the gradient of J with respect to the variables a, b and c, we must first find the gradient of J with respect to the variable it is directly dependent on (v). In turn, we should then compute the gradient of v with respect to the variables it is dependent on.

Therefore, we can derive the gradient for J by *propagating backwards* through the computation graph:

$$\frac{\partial J}{\partial a}(a,b,c) = \frac{\mathrm{d}J}{\mathrm{d}v} \cdot \frac{\partial v}{\partial a}$$

$$\frac{\partial J}{\partial b}(a,b,c) = \frac{\mathrm{d}J}{\mathrm{d}v} \cdot \frac{\partial v}{\partial u} \cdot \frac{\partial u}{\partial b}$$

$$\frac{\partial J}{\partial c}(a,b,c) = \frac{\mathrm{d}J}{\mathrm{d}v} \cdot \frac{\partial v}{\partial u} \cdot \frac{\partial u}{\partial c}$$

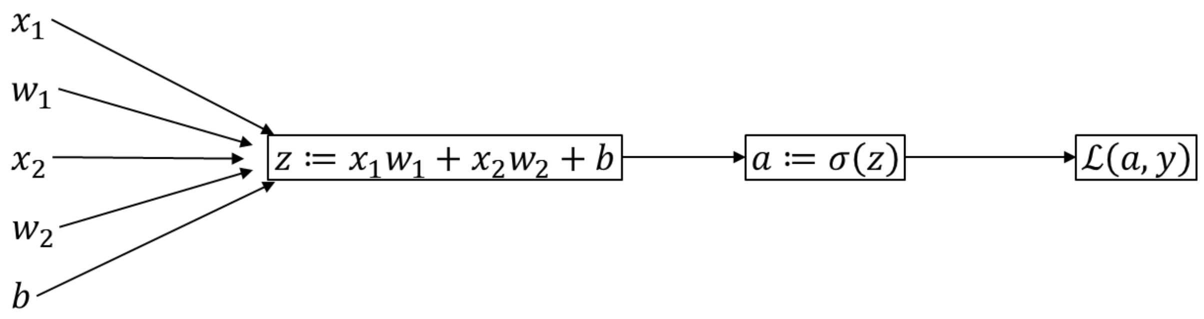
This is known as the *chain rule* of differentiation.

5.4.2. Computing Gradient Descent for one training example

We have the following variables for our logistic regression model:

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

We can visualize the gradient descent algorithm using a computation graph as follows:



The objective of gradient descent is to minimize the loss function $\mathcal{L}$ by traversing the slope of steepest descent. To compute the gradient, we would have to differentiate the loss function with respect to one of the input variables.

From §5.4.1., we know that we can perform differentiation using the chain rule, which is achieved by propagating backwards through the computation graph.

Upon differentiating, we get the following results:

$$\mathrm{d}a := \frac{\partial \mathcal{L}}{\partial a}(a, y) = -\frac{y}{a} + \frac{1-y}{1-a}$$

$$\mathrm{d}z := \frac{\partial \mathcal{L}}{\partial z}(a, y) = \frac{\partial \mathcal{L}}{\partial a}(a, y) \cdot \frac{\mathrm{d}a}{\mathrm{d}z}(z) = a - y$$

$$\mathrm{d}w_1 := \frac{\partial \mathcal{L}}{\partial w_1}(a, y) = x_1 \frac{\partial \mathcal{L}}{\partial z}(a, y) = x_1(a - y) = x_1 \mathrm{d}z$$

$$\mathrm{d}w_2 := \frac{\partial \mathcal{L}}{\partial w_2}(a, y) = x_2 \frac{\partial \mathcal{L}}{\partial z}(a, y) = x_2(a - y) = x_2 \mathrm{d}z$$

$$\mathrm{d}b := \frac{\partial \mathcal{L}}{\partial b}(a, y) = 1 \cdot \frac{\partial \mathcal{L}}{\partial z}(a, y) = a - y$$

The convention during programming is to denote the derivative of the loss function with respect to parameter 't' as '$\mathrm{d}t$'.

5.4.3. Computing Gradient Descent for m training examples

Recalling the cost function J,

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

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

It can be observed that the cost function J is simply the average of the loss function $\mathcal{L}$. Therefore,

$$\frac{\partial J}{\partial w_1}(\mathbf{w}, b) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial \mathcal{L}}{\partial w_1}(a^{(i)}, y^{(i)})$$

From the previous section, we derived the derivative of the loss function. Therefore,

$$\frac{\partial J}{\partial w_1}(\mathbf{w}, b) = \frac{x_1}{m} \sum_{i=1}^{m} a^{(i)} - y^{(i)}$$

Here is an algorithm snippet for n_x features:

```
1. **Set** $J := 0$
2. **For** $i := 1$ to $n_x$:
    1. **Set** $\mathrm{d}w_i := 0$
3. **Set** $\mathrm{d}b := 0$
4. **For** $i := 1$ **to** $m$:
    1. **Set** $z^{(i)} := \mathbf{w}^T\mathbf{x}^{(i)} + b$
    2. **Set** $a^{(i)} := \sigma(z^{(i)})$
    3. **Decrement** $J$ **by** $y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log (1 - a^{(i)})$
    4. **Set** $\mathrm{d}z^{(i)} := a^{(i)} - y^{(i)}$
    5. **For** $j := 1$ to $n_x$:
        1. **Increment** $\mathrm{d}w_j$ **by** $x_j^{(i)}\mathrm{d}z^{(i)}$
    6. **Increment** $\mathrm{d}b$ **by** $\mathrm{d}z^{(i)}$
5. **Set** $J := J \div m$
6. **For** $i := 1$ to $n_x$:
    1. **Set** $\mathrm{d}w_i := \mathrm{d}w_i \div m$
    2. **Decrement** $w_i$ by $\alpha \cdot \mathrm{d}w_i$
7. **Set** $\mathrm{d}b := \mathrm{d}b \div m$
8. **Decrement** $b$ by $\alpha \cdot \mathrm{d}b$
```

where α is known as the *learning rate*.

This algorithm does work, but it contains too many explicit `for` loops, which slows down the algorithm by a lot, especially considering larger data sets. To overcome the inefficiency of explicit `for` loops, we shall use a technique known as *Vectorization*.

5.5. Vectorization

5.5.1. Scalar Product

In Python, we can declare two random arrays $\mathbf{a} \in \mathbb{R}^{1{,}000{,}000}$ and $\mathbf{b} \in \mathbb{R}^{1{,}000{,}000}$ using NumPy as follows:

```python
import numpy as np
a = np.random.randn(1_000_000)
b = np.random.randn(1_000_000)
```

To obtain the scalar product of $\mathbf{a}$ and $\mathbf{b}$, we *could* use a `for` loop:

```
c: int = 0
for i in range(1_000_000):
        c += a[i] * b[i]
```

However, this process takes too much time (~ 400 ms). A better approach would be to *vectorize* the process by using the built-in NumPy `dot()` function.

```
c: int = np.dot(a, b)
```

This process takes much less time (~ 2 ms) than an explicit `for` loop.

5.5.2. Linear Transformation

Consider computing the matrix-vector product of a matrix $A \in \mathcal{M}_{m\times n}(\mathbb{R})$ and a vector $\mathbf{v} \in \mathbb{R}^n$. It *could* be computed in Python using nested for loops:

```
u = np.zeros((n, 1))
for i in range(m):
        for j in range(n):
                u += A[i][j] + v[j]
```

However, this code has a time complexity of $O(n^2)$ and runs much slower than a vectorized approach:

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

5.5.3. Operating on every vector element

Consider exponentiating every element of a vector $\mathbf{v} \in \mathbb{R}^n$. A non-vectorized approach would be as follows:

```
u = np.zeros((n, 1))
for i in range(n):
        u[i] = math.exp(v[i])
```

A much faster vectorized solution would be as follows:

```
u = math.exp(v)
```

5.5.4. Vectorizing Gradient Descent

The gradient descent algorithm mentioned in §5.4.3. uses many explicit `for` loops, which could be eliminated by vectorizing $\mathrm{d}\mathbf{w}$:

```
1. Set $J := 0$
2. Set $\mathrm{d}\mathbf{w} := \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \in \mathbb{R}^{n_x}$
3. Set $\mathrm{d}b := 0$
4. For $i := 1$ to $m$:
    1. Set $z^{(i)} := \mathbf{w}^T \mathbf{x}^{(i)} + b$
    2. Set $a^{(i)} := \sigma(z^{(i)})$
    3. Decrement $J$ by $(y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log (1 - a^{(i)}))$
    4. Set $\mathrm{d}z^{(i)} := a^{(i)} - y^{(i)}$
    5. Increment $\mathrm{d}\mathbf{w}$ by $\mathbf{x}^{(i)} \mathrm{d}z^{(i)}$
    6. Increment $\mathrm{d}b$ by $\mathrm{d}z^{(i)}$
5. Set $J := J \div m$
6. Set $\mathrm{d}\mathbf{w} := \mathrm{d}\mathbf{w} \div m$
7. Decrement $\mathbf{w}$ by $\alpha \cdot \mathrm{d}\mathbf{w}$
8. Set $\mathrm{d}b := \mathrm{d}b \div m$
9. Decrement $b$ by $\alpha \cdot \mathrm{d}b$
```

5.5.5. Vectorizing Logistic Regression

After vectorizing $\mathrm{d}\mathbf{w}$ in §5.5.4, we still have a `for` loop iterating over all m training examples.

We can vectorize $z^{(i)} \; \forall \, i$ by constructing a matrix $Z \in \mathcal{M}_{1 \times m}(\mathbb{R})$ as follows:

$$Z := \begin{bmatrix} z^{(1)} & z^{(2)} & \dots & z^{(m)} \end{bmatrix}$$

Using matrix X from §4.1., We can then define each $z^{(i)}$ compactly as follows:

$$Z := \mathbf{w}^T \cdot X + [b \quad b \quad \cdots \quad b]$$

where $[b \quad b \quad \cdots \quad b] \in \mathcal{M}_{1 \times m}(\mathbb{R})$.

The NumPy command would look like this:

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

Note: Although b is a scalar, NumPy will automatically *broadcast* it to a row vector of `shape` $(1, m)$ and add it to the result of `np.dot(w.T, X)`.

We can similarly vectorize $a^{(i)}$ as a matrix $A \in \mathcal{M}_{1\times m}(\mathbb{R})$:

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

5.5.6. Vectorizing Gradient Computation

From §5.4.2., we know that,

$$\mathrm{d}z^{(i)} := a^{(i)} - y^{(i)}$$

We can vectorize the above expression by constructing a new matrix $\mathrm{d}Z \in \mathcal{M}_{1\times m}(\mathbb{R})$ using the matrices A and Y from §5.5.5. and §4.1. respectively:

$$\begin{aligned} \mathrm{d}Z &:= \begin{bmatrix} \mathrm{d}z^{(1)} & \mathrm{d}z^{(2)} & \cdots & \mathrm{d}z^{(m)} \end{bmatrix} \\ &:= \begin{bmatrix} a^{(1)} - y^{(1)} & a^{(2)} - y^{(2)} & \cdots & a^{(m)} - y^{(m)} \end{bmatrix} \\ &:= A - Y \end{aligned}$$

Recalling definitions of $\mathrm{d}\mathbf{w}$ and $\mathrm{d}b$,

$$\begin{aligned} \mathrm{d}\mathbf{w} &:= \frac{1}{m}\sum_{i=1}^{m} \mathbf{x}^{(i)} \mathrm{d}z^{(i)} \\ &= \frac{1}{m} \begin{bmatrix} \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(m)} \end{bmatrix} \begin{bmatrix} \mathrm{d}z^{(1)} \\ \mathrm{d}z^{(2)} \\ \vdots \\ \mathrm{d}z^{(m)} \end{bmatrix} \\ &:= \frac{1}{m} X \, \mathrm{d}Z^T \\ \mathrm{d}b &:= \frac{1}{m}\sum_{i=1}^{m} \mathrm{d}z^{(i)} = \frac{1}{m}\sum \mathrm{d}Z \end{aligned}$$

Therefore, `dw` and `db` can be computed using the following code:

```
dw = np.dot(X, dZ.T) / m
db = np.sum(dZ) / m
```

5.5.7. Optimized Logistic Regression

The following algorithm describes one iteration of gradient descent with learning rate α which has been optimized using vectorization:

1. **Set** $Z := \mathbf{w}^T \cdot X + b$
2. **Set** $A := \sigma(Z)$
3. **Set** $J := -\frac{1}{m} \sum (Y \log A + (1 - Y) \log (1 - A))$
4. **Set** $\mathrm{d}Z := A - Y$
5. **Set** $\mathrm{d}\mathbf{w} := \frac{1}{m} X \, \mathrm{d}Z^T$
6. **Set** $\mathrm{d}b := \frac{1}{m} \sum \mathrm{d}Z$
7. **Decrement** $\mathbf{w}$ by $\alpha \cdot \mathrm{d}\mathbf{w}$
8. **Decrement** b by $\alpha \cdot \mathrm{d}b$

Note: It is assumed that variables are broadcasted to the correct shape whenever required.

5.6. Kernelization

Many of the scalable deep learning implementations are done on a Graphics Processing Unit (GPU).

Both the GPU and the CPU have parallelization instructions, which are sometimes called SIMD (Single Instruction, Multiple Data) Instructions. The GPU is much better designed for this kind of parallel processing.

Parallelism can allow NumPy to be much more efficient at processing large volumes of data.

6. Neural Networks