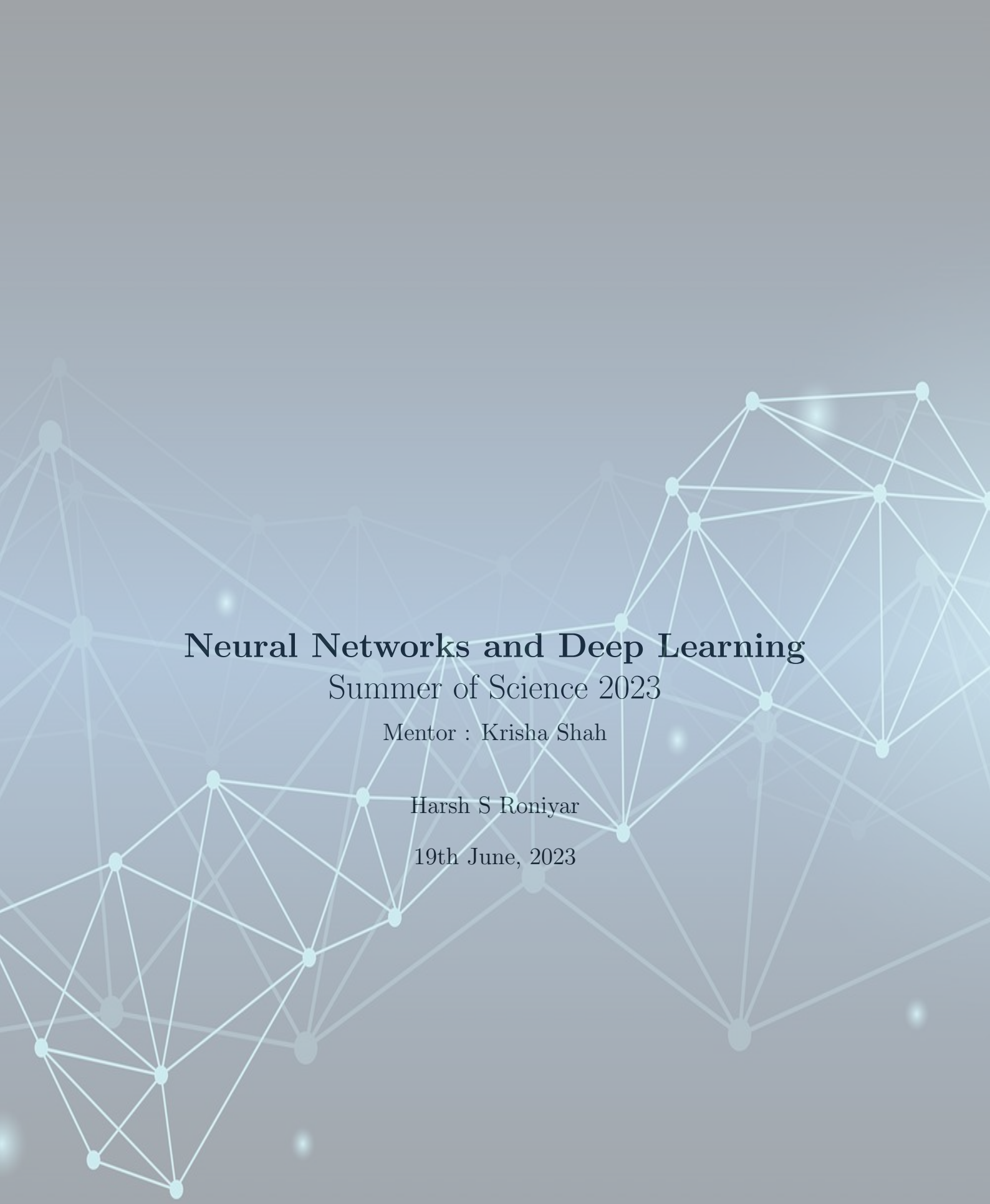


Neural Networks and Deep Learning

Summer of Science 2023

Mentor : Krisha Shah

Harsh S Roniyar

19th June, 2023

Abstract

Neural networks and deep learning have revolutionized the field of artificial intelligence and have become indispensable tools in various domains. This report provides an overview of the fundamental concepts, advancements, and applications of neural networks and deep learning.

The report begins with an introduction to neural networks, explaining their structure, operation, and key components, such as neurons, layers, and activation functions. The report examines the training process of neural networks and discusses important techniques such as gradient descent, forward and back-propagation, and vectorization. Then, the report goes on to implement Shallow and Deep Neural Networks

My learning so far is based on Andrew Ng's Neural Networks and Deep Learning Course and from the book Deep Learning by Goodfellow et al.

Contents

1 Introduction

A neural network is a computational model inspired by the structure and function of the human brain. It is a type of machine learning algorithm that is designed to recognize patterns and make predictions based on input data. Neural networks consist of interconnected nodes, called neurons, organized into layers.

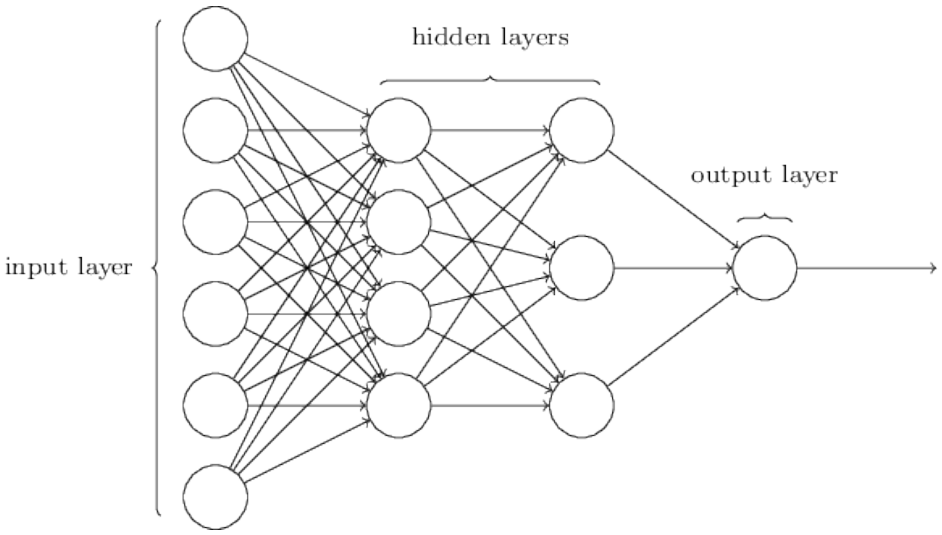


Figure 1: Multi-layer Perceptron (Building block of Neural Networks)

The basic building block of a neural network is the artificial neuron, also known as a perceptron. Each neuron takes in one or more inputs, applies weights to those inputs, sums them up, and then applies an activation function to produce an output. The activation function determines whether the neuron should "fire" and pass its output to the next layer of neurons.

Broad paradigms in Neural Networks (in general, Machine Learning) are

- Supervised Learning
- Unsupervised Learning
- Sequential Learning

1.1 Supervised Learning

In supervised learning the algorithm learns from labeled examples or training data. In supervised learning, the input data is accompanied by corresponding target labels or desired outputs. The goal is to train a model that can learn the mapping between the input data and the target labels, enabling it to make accurate predictions on new, unseen data.

1.2 Unsupervised Learning

In unsupervised learning the algorithm learns patterns and structures in the data without any explicit labels or target outputs. Unlike supervised learning, the input data in unsupervised learning is unlabeled, and the algorithm explores the data to find inherent patterns, relationships, or groupings.

1.3 Reinforcement (Sequential) Learning

Reinforcement learning is a type of machine learning where an agent learns to make sequential decisions in an environment to maximize a cumulative reward signal. The agent interacts with the environment, takes actions, receives feedback in the form of rewards or penalties, and learns the optimal values to achieve long-term goals.

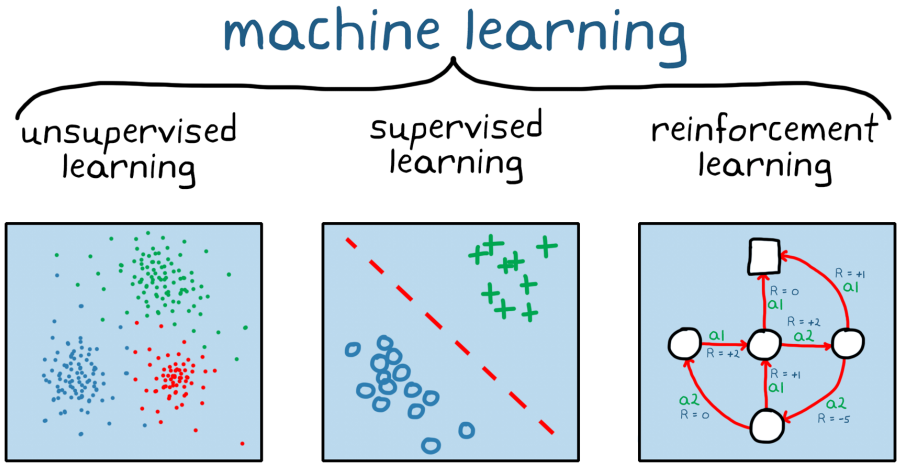


Figure 2: Unsupervised v/s Supervised v/s Reinforcement Learning

In the subsequent sections below, we will be only discussing 'Supervised' Learning with Neural Networks.

Some examples of Neural Networks :

1. Standard NN
2. Convolution NN
3. Recurrent NN

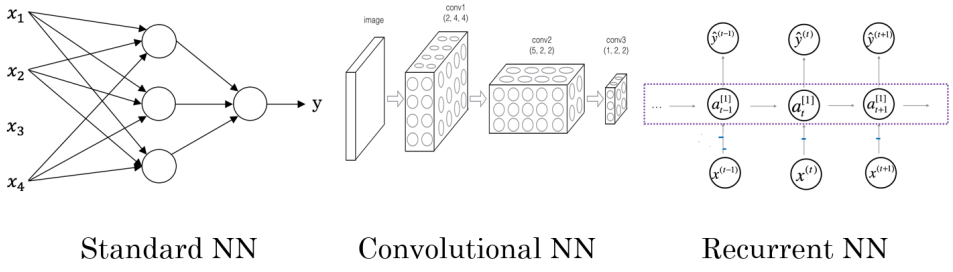


Figure 3: Examples of Neural Networks

2 Basics of Neural Network Programming

2.1 Logistic Regression as a Neural Network

2.1.1 Binary Classification

As is evident, binary classification is a supervised learning algorithm that categorizes observations into two different classes (typically 0 and 1). So primarily, when we want to model Binary Classification, we need outputs from the model to lie between 0 and 1.

2.1.2 Logistic Regression

Logistic regression is a learning algorithm used in a supervised learning problem when the output y are all either zero or one. The goal of logistic regression is to minimize the error between its predictions and training data.Logistic regression is modeled using the sigmoid curve which gives outputs between 0 and 1, which is representative of the probability of it being 1.
i.e. $\hat{y} = P(y = 1|x)$ where x is our input layer and $\hat{y}$ is the predicted value and $0 \leq \hat{y} \leq 1$

The parameters used in Logistic Regression are:

- The input features vector: $x \in \mathbb{R}^{n_x}$, where n_x is the number of features[1]
- The training label: $y \in 0, 1$
- The weights: $w \in \mathbb{R}^{n_x}$, where n_x is the number of features
- The threshold: $b \in \mathbb{R}$
- The output: $\hat{y} = \sigma(w^T x + b)$
- Sigmoid function: $s = \sigma(z) = 1/(1 + e^{-z})$

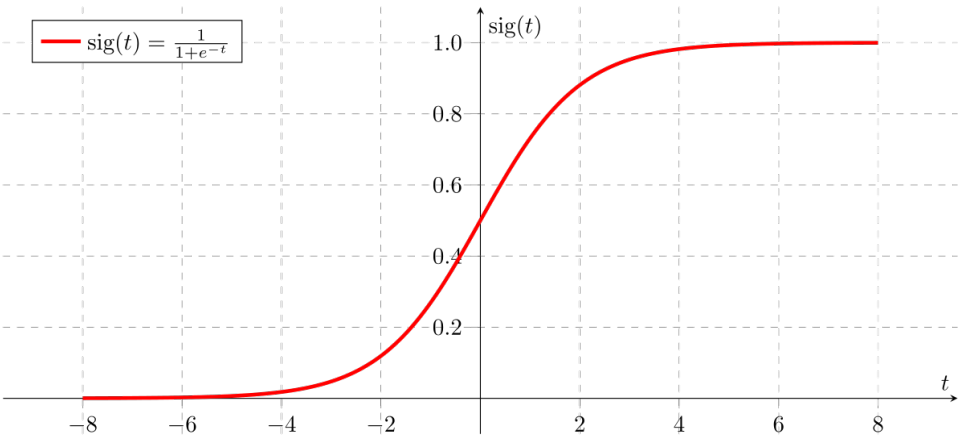


Figure 4: Sigmoid Function

[1]Input features vector for an image.
Images of size $(num_{px}, num_{px}, 3)$, where 3 represents the 3 *RGB* channels, are flattened into single vectors of shape $(num_{px} * num_{px} * 3, 1)$.
Here, $n_x = num_{px} * num_{px} * 3$.

2.1.3 Loss(Error) and Cost Functions

Loss function computes the error for a single training example (discrepancy between $\hat{y^{(i)}}$ and $y^{(i)}$), whereas the Cost function is the average of the loss function of the entire training set.

Loss Function:

$L\left(\hat{y}^{(i)}, y^{(i)}\right) = -\left(y^{(i)} \log\left(\hat{y}^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - \hat{y}^{(i)}\right)\right.$

- If $y^{(i)} = 1 : L\left(\hat{y}^{(i)}, y^{(i)}\right) = -\log\left(\hat{y}^{(i)}\right)$ where $\log\left(\hat{y}^{(i)}\right)$ and $\hat{y}^{(i)}$ should be close to 1
- If $y^{(i)} = 0 : L\left(\hat{y}^{(i)}, y^{(i)}\right) = -\log\left(1 - \hat{y}^{(i)}\right)$ where $\log\left(1 - \hat{y}^{(i)}\right)$ and $\hat{y}^{(i)}$ should be close to 0

Cost Function:

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

A good point to note here is that by choosing $J(w, b)$ as defined above, we have made it a convex function, thus eliminating possibility of multiple optimums. It will be good to remind ourselves of our goal at this point, which is to minimize w and b.

2.2 Gradient Descent

Gradient Descent is an optimization algorithm commonly used in neural networks for updating the weights and biases during the training process.

In a neural network, the goal is to minimize a cost or loss function that measures the discrepancy between the predicted output and the actual output. Gradient Descent iteratively adjusts the network's parameters (weights and biases) in the direction of steepest descent of the cost function to find its minimum.

The general update rule for Gradient Descent in a neural network can be expressed as follows:

$$\theta_{t+1} = \theta_t - \alpha \cdot \nabla J(\theta_t)$$

where:

- θ_{t+1} represents the updated parameter values at the $t + 1$ iteration.
- θ_t represents the current parameter values at the t iteration.
- α (alpha) is the learning rate, which determines the step size for each iteration. It is a hyperparameter that needs to be carefully tuned.
- $\nabla J(\theta_t)$ is the gradient of the cost function J with respect to the parameters θ_t. The gradient represents the direction and magnitude of the steepest ascent of the cost function.

To compute the gradient $\nabla J(\theta_t)$ for the parameters θ_t, we typically use the backpropagation algorithm, which calculates the gradient recursively layer by layer, starting from the output layer and propagating the errors backward through the network.

Once the gradient is computed, we multiply it by the learning rate α and subtract the result from the current parameter values θ_t to obtain the updated parameter values θ_{t+1}.

The process of iteratively applying this update rule continues until convergence or until a predefined number of iterations is reached. At convergence, the parameters ideally reach a point where the cost function is minimized, resulting in a well-trained neural network (as seen in the Graph at points A and B).

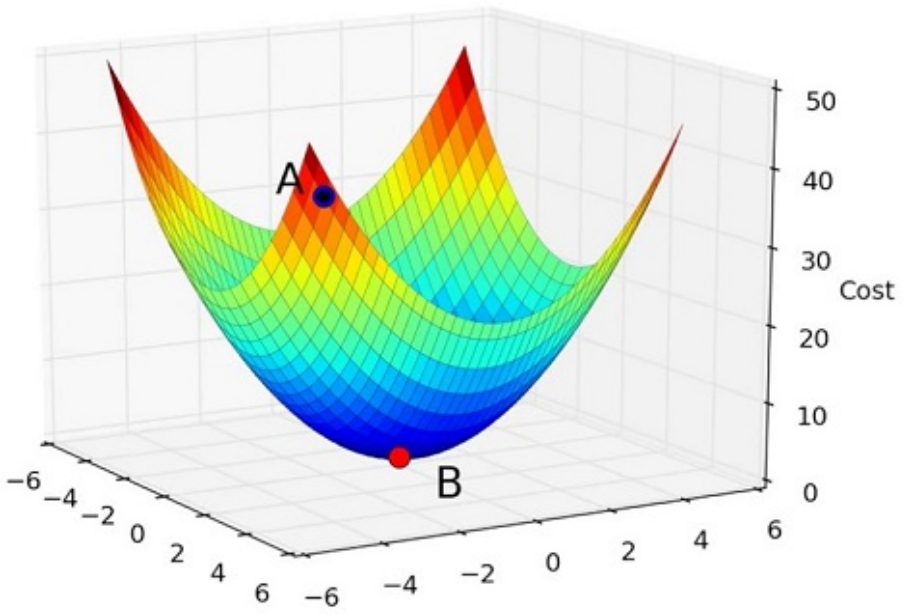


Figure 5: Gradient Descent Optimization

2.3 Computation Graph

A computation graph is a visual representation of the computations performed in a neural network. It breaks down complex mathematical operations into smaller, interconnected nodes, allowing for efficient computation and automatic differentiation.

In a neural network, computations are typically organized into layers consisting of nodes or neurons. Each neuron performs a computation by applying a linear transformation followed by a non-linear activation function.

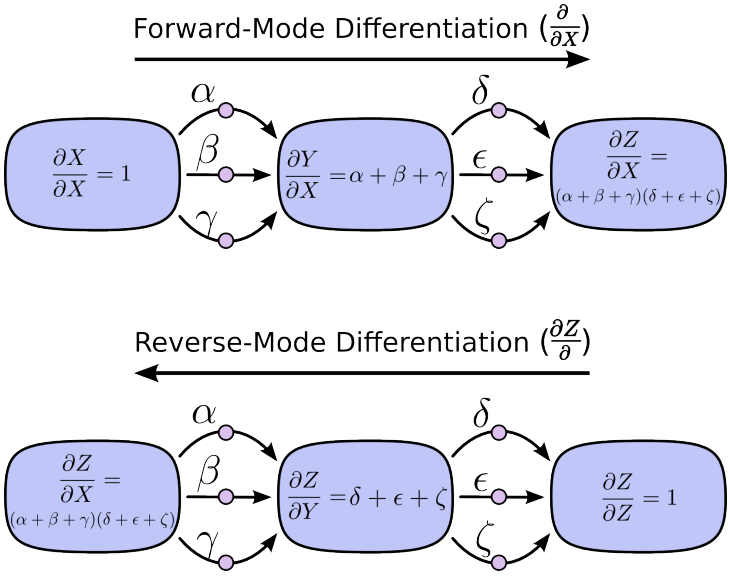
Consider a simple neural network with two input features x_1 and x_2 and a single output y. The computation graph for this network can be represented as follows:

$$z = w_1 x_1 + w_2 x_2 + b$$
$$a = \sigma(z)$$
$$y = f(a)$$

where:

- z represents the weighted sum of the inputs plus a bias term. This is the linear transformation applied by the neuron.
- w_1 and w_2 are the weights associated with the input features.
- b is the bias term.
- $\sigma(\cdot)$ is the activation function, which introduces non-linearity to the computation. It could be a function like the sigmoid function or the rectified linear unit (ReLU) or the leaky rectified linear unit (Leaky ReLU) function.
- a represents the activation of the neuron, which is obtained by applying the activation function to the linear transformation output z.
- $f(\cdot)$ represents the final output function that produces the desired output y based on the activation a.

The computation graph provides a clear and structured representation of the computations performed in the neural network. It allows for efficient forward propagation, where inputs flow through the graph to produce outputs, and also facilitates backward propagation, which is used for computing gradients during the training process.



2.3.1 Forward and Back Propagation

During forward propagation, input values are assigned to the corresponding nodes in the graph, and computations are performed layer by layer until the final output is obtained. During backward propagation (also known as backpropagation), gradients are computed by recursively applying the chain rule to propagate the gradients from the output layer to the input layer. These gradients are then used to update the weights and biases through optimization algorithms such as Gradient Descent, as explained in the previous response.

An example to help visualise it[2]

[2]A wonderful description of the same can be found here: https://colah.github.io/posts/2015-08-Backprop/

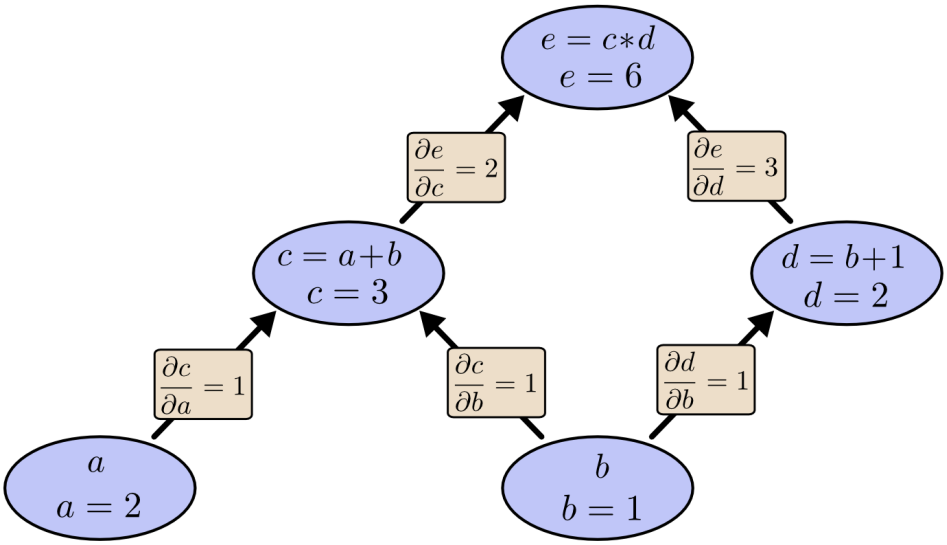


Figure 6: An example of a Computation Graph

The computation graph plays a crucial role in neural networks by providing a structured representation of the computations and enabling efficient and accurate training through automatic differentiation.

2.4 Vectorization

In deep learning, we deal with very large datasets. Hence, a non-computationally-optimal function can become a huge bottleneck in our algorithm and can result in a model that takes ages to run.

To make sure that our code is computationally efficient, we will use vectorization.

An example to demonstrate the difference between the following implementations of the dot/outer/elementwise product is given in the following pages.

Classical Implementation

```
import time

x1 = [9, 2, 5, 0, 0, 7, 5, 0, 0, 0, 9, 2, 5, 0, 0]
x2 = [9, 2, 2, 9, 0, 9, 2, 5, 0, 0, 9, 2, 5, 0, 0]

### CLASSIC DOT PRODUCT OF VECTORS IMPLEMENTATION ###
tic = time.process_time()
dot = 0

for i in range(len(x1)):
    dot += x1[i] * x2[i]
toc = time.process_time()
print ("dot = " + str(dot) + "\n ----- Computation time = " + str(1000 * (toc - tic))
↪  + "ms")

### CLASSIC OUTER PRODUCT IMPLEMENTATION ###
tic = time.process_time()
outer = np.zeros((len(x1), len(x2))) # we create a len(x1)*len(x2) matrix with only
↪  zeros

for i in range(len(x1)):
    for j in range(len(x2)):
        outer[i,j] = x1[i] * x2[j]
toc = time.process_time()
print ("outer = " + str(outer) + "\n ----- Computation time = " + str(1000 * (toc -
↪  tic)) + "ms")

### CLASSIC ELEMENTWISE IMPLEMENTATION ###
tic = time.process_time()
mul = np.zeros(len(x1))

for i in range(len(x1)):
    mul[i] = x1[i] * x2[i]
toc = time.process_time()
print ("elementwise multiplication = " + str(mul) + "\n ----- Computation time = " +
↪  str(1000 * (toc - tic)) + "ms")

### CLASSIC GENERAL DOT PRODUCT IMPLEMENTATION ###
W = np.random.rand(3,len(x1)) # Random 3*len(x1) numpy array
tic = time.process_time()
gdot = np.zeros(W.shape[0])

for i in range(W.shape[0]):
    for j in range(len(x1)):
        gdot[i] += W[i,j] * x1[j]
toc = time.process_time()
print ("gdot = " + str(gdot) + "\n ----- Computation time = " + str(1000 * (toc -
↪  tic)) + "ms")
```

Output:

```
dot = 278
 ----- Computation time = 0.1567150000001405ms
outer = [[81. 18. 18. 81.  0. 81. 18. 45.  0.  0. 81. 18. 45.  0.  0.]
 [18.  4.  4. 18.  0. 18.  4. 10.  0.  0. 18.  4. 10.  0.  0.]
 [45. 10. 10. 45.  0. 45. 10. 25.  0.  0. 45. 10. 25.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [63. 14. 14. 63.  0. 63. 14. 35.  0.  0. 63. 14. 35.  0.  0.]
 [45. 10. 10. 45.  0. 45. 10. 25.  0.  0. 45. 10. 25.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [81. 18. 18. 81.  0. 81. 18. 45.  0.  0. 81. 18. 45.  0.  0.]
 [18.  4.  4. 18.  0. 18.  4. 10.  0.  0. 18.  4. 10.  0.  0.]
 [45. 10. 10. 45.  0. 45. 10. 25.  0.  0. 45. 10. 25.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]]
 ----- Computation time = 0.24711300000035408ms
elementwise multiplication = [81.  4. 10.  0.  0. 63. 10.  0.  0.  0. 81.  4. 25.  0.
↪  0.]
 ----- Computation time = 0.15952099999960723ms
gdot = [33.67290389 24.88257566 17.1213738 ]
 ----- Computation time = 0.23116099999986872ms
```

Vectorised Implementation

```
x1 = [9, 2, 5, 0, 0, 7, 5, 0, 0, 0, 9, 2, 5, 0, 0]
x2 = [9, 2, 2, 9, 0, 9, 2, 5, 0, 0, 9, 2, 5, 0, 0]

### VECTORIZED DOT PRODUCT OF VECTORS ###
tic = time.process_time()
dot = np.dot(x1,x2)
toc = time.process_time()
print ("dot = " + str(dot) + "\n ----- Computation time = " + str(1000 * (toc - tic))
↪  + "ms")

### VECTORIZED OUTER PRODUCT ###
tic = time.process_time()
outer = np.outer(x1,x2)
toc = time.process_time()
print ("outer = " + str(outer) + "\n ----- Computation time = " + str(1000 * (toc -
↪  tic)) + "ms")

### VECTORIZED ELEMENTWISE MULTIPLICATION ###
tic = time.process_time()
mul = np.multiply(x1,x2)
toc = time.process_time()
print ("elementwise multiplication = " + str(mul) + "\n ----- Computation time = " +
↪  str(1000*(toc - tic)) + "ms")

### VECTORIZED GENERAL DOT PRODUCT ###
tic = time.process_time()
dot = np.dot(W,x1)
toc = time.process_time()
print ("gdot = " + str(dot) + "\n ----- Computation time = " + str(1000 * (toc - tic))
↪  + "ms")
```

Output:

```
dot = 278
 ----- Computation time = 0.09885899999995118ms
outer = [[81 18 18 81  0 81 18 45  0  0 81 18 45  0  0]
 [18  4  4 18  0 18  4 10  0  0 18  4 10  0  0]
 [45 10 10 45  0 45 10 25  0  0 45 10 25  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [63 14 14 63  0 63 14 35  0  0 63 14 35  0  0]
 [45 10 10 45  0 45 10 25  0  0 45 10 25  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [81 18 18 81  0 81 18 45  0  0 81 18 45  0  0]
 [18  4  4 18  0 18  4 10  0  0 18  4 10  0  0]
 [45 10 10 45  0 45 10 25  0  0 45 10 25  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]]
 ----- Computation time = 0.1062459999998655ms
elementwise multiplication = [81  4 10  0  0 63 10  0  0  0 81  4 25  0  0]
 ----- Computation time = 0.06695800000011687ms
gdot = [26.70284694 22.76463722 19.81081097]
 ----- Computation time = 0.20383000000001594ms
```

The vectorized implementation is much cleaner and more efficient. For bigger vectors/matrices, the differences in running time become even bigger[3].

2.5 Implementing Logistic Regression

After going through all the basic components of a neural network, we can combine them in order to create our (basic) neural network model.
The main steps for building a Neural Network are:

1. Define the model structure (such as number of input features)
2. Initialize the model's parameters
3. Loop:
   (a) Calculate current loss (forward propagation)
   (b) Calculate current gradient (backward propagation)
   (c) Update parameters (gradient descent)

The following equations will help us to implement our model over 'n' number of training examples[4]

$$A = \sigma(w^T X + b) = (a^{(1)}, a^{(2)}, ..., a^{(m-1)}, a^{(m)}) \tag{1}$$

$$J = -\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}\log(a^{(i)}) + (1-y^{(i)})\log(1-a^{(i)})) \tag{2}$$

[3]**Note** that `np.dot()` performs a matrix-matrix or matrix-vector multiplication. This is different from `np.multiply()` and the $*$ operator (which is equivalent to $.*$ in Matlab/Octave), which performs an element-wise multiplication.

[4]**Note** In cases involving arithmetic operations, broadcasting in Python using NumPy arrays vectorizes the operation.

$$\frac{\partial J}{\partial w} = \frac{1}{m} X(A - Y)^T \tag{3}$$

$$\frac{\partial J}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} (a^{(i)} - y^{(i)}) \tag{4}$$

where,

- w is weights, a numpy array of size $(num_{px} * num_{px} * 3, 1)$
- b is bias, a scalar
- X is data of size $(num_{px} * num_{px} * 3, n)$
- Y is true "label" vector of size $(1, n)$

3 Future Plan Of Action

Timeline

Week 6
Shallow Neural Networks

Week 7
Deep Neural Networks
Refer Chap. 5, 6, 7 from Goodfellow[a]

[a]Optional

Week 8
Regularization and Hyperparameter Tuning
Final Report Preparation

Week 9
Summarization and Review
Discuss with mentor

References

1. A five-course deep learning specialization by Coursera
2. This is an amazing series on Neural Networks by 3Blue1Brown
3. This is a good book by Michael Nielsen on Neural Networks and Deep Learning