Attention Mechanism for Human Motion Prediction

By
AMAL FAHAD AL-AQEL
Umm Al-Qura University
Makkah, Saudi Arabia
2020

Submitted to the Faculty of the
College of Computer Science and Information System of
the
Umm Al-Qura University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
April 2020

Attention Mechanism for Human Motion Prediction

Signature of Author		……..………………
Committee Member		Signature and Date
Dr. ………..………………	(Chairman)	………..………………
Dr. ………..………………	(Member)	………..………………
Dr. ………..………………	(Member)	………..………………
Dr. ………..………………	(Member)	………..………………
Dr. ………..………………	(External Examiner)	………..………………

Date of Degree : [April 2020]

ACKNOWLEDGEMENTS

First and foremost, I would like to thank Allah Almighty for giving me the strength and means to undertake this thesis. Also, I would like to express my sincere gratitude to my advisor dr.Murtaza Khan for the continuous support of my thesis. I cannot express enough thanks to my family and my colleagues for their continued support and encouragement.

ABSTRACT

Full Name	:	Amal Fahad Al-Aqel
Thesis Title	:	Bidirectional Recurrent Neural Networks for Human Motion Prediction
Major Field	:	Computer Vision
Date of Degree	:	[April 2020]

Human motion prediction aims to forecast the most likely future frames of motion conditioned on a given sequence of frames. Because of its importance to many applications especially robotics, human motion prediction has received a lot of interest and has become an active area of research. Recently, deep learning methods have been dominant in many tasks due to their successful results. Particularly, Recurrent Neural Networks (RNNs) have shown excellent performance on human motion prediction task and other tasks that depend on sequential data, where preserving the order of the sequence items is crucial. The well-known Sequence-to-Sequence (Seq2Seq) architectures have been used for sequence learning where two RNNs namely the encoder and the decoder work cooperatively to transform one sequence to another. In the context of neural machine translation, the use of attention decoders yields state-of-the-art results. This work attempts to assess quantitatively the use of a bidirectional encoder and an attention decoder in human motion prediction. The experiments of this work have shown that using attention decoder has achieved state-of-the-art results after 160 milliseconds of motion prediction. In contrast with earlier works, the quality of predictions doesn’t deteriorate and remains stable even after more than 1 second of motion prediction.

ملخص الرسالة

الاسم الكامل: [أمل فهد العقل**]**

عنوان الرسالة: [التنبؤ بالحركة البشرية باستخدام الشبكات العصبية التكرارية**]**

التخصص: [هندسة وعلوم الحاسب الآلي**]**

تاريخ الدرجة العلمية:

يهدف التنبؤ بالحركة البشرية إلى توقع إطارات الحركة المستقبلية الأكثر احتمالاً والمشروطة بسلسلة معطاة من إطارات الحركة. نظراً لأهمية التنبؤ بالحركة البشرية لكثير من التطبيقات، لا سيما علم الروبوتات، فقد حظي هذا المجال باهتمام كبير وأصبح مجالاً نشطاً للبحث. في الآونة الأخيرة، سادت أساليب التعلم العميق في العديد من المهام نظراً لنتائجها الناجحة. أظهرت الشبكات العصبية التكرارية على وجه الخصوص أداءً ممتازاً في مهمة التنبؤ بالحركة البشرية وغيرها من المهام خاصة تلك التي تعتمد على البيانات المتسلسلة، حيث يعد الحفاظ على ترتيب العناصر أمراً بالغ الأهمية. تم استخدام البنى المعروفة باسم (Seq2Seq) في تعلم التسلسل حيث يعمل نوعان من الشبكات العصبية التكرارية وهما المشفر والمفسر بشكل تعاوني بهدف تحويل سلسلة ما إلى سلسلة أخرى. في سياق الترجمة الآلية العصبية، أسفر استخدام المشفرات المعتمدة على تقنية الانتباه إلى أحدث النتائج. يحاول هذا العمل أن يقيم بشكل كمي استخدام المشفر ثنائي الاتجاه والمفسر المعتمد على تقنية الانتباه في التنبؤ بالحركة البشرية. أوضحت التجارب التي أجريت في هذا العمل أن المفسر المعتمد على تقنية الانتباه قاد إلى تحقيق أحدث النتائج بعد 160 جزءاً من الثانية من التنبؤ بالحركة. على عكس الأعمال السابقة، فإن جودة التنبؤات لا تتدهور وتظل مستقرة حتى بعد مضي أكثر من ثانية واحدة من التنبؤ بالحركة.

PUBLISHED WORK

A paper entitled “Attention Mechanism for Human Motion Prediction” was presented at the 2020 3rd International Conference on Computer Applications Information Security (ICCAIS20).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION AND LITERATURE REVIEW

1.1 Human Motion Prediction

Humans are blessed with a remarkable ability to make accurate short-term predictions about their surroundings based on past observations (Gui et al., 2018). Crossing a crowded street is one among many other tasks that would be very challenging without our capacity of understanding human movements and anticipating their most likely actions in the near future (Martinez et al., 2017).

Given a set of 3D poses or skeletons, the goal of human motion prediction is to forecast the most likely future frames of human motion based on the given sequence as illustrated in Figure 1.1 (Tang et al., 2018; Martinez et al., 2017).

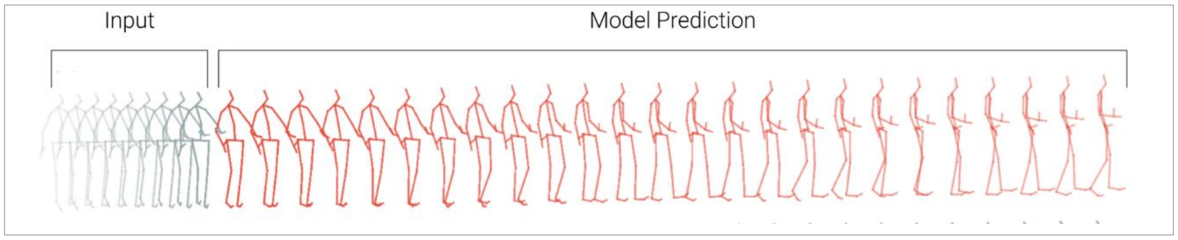


Figure 1.1: Motion prediction task. The gray colored sequence is the input sequence and the red colored sequence is the output sequence or the prediction (Martinez et al., 2017).

Human motion prediction is essential for achieving the goal of robotic intelligence where robots are supposed not only to have a notion of human motion but also to be able to predict their movements, resulting in a seamless interaction between humans and machines (Tang et al., 2018).

Many situations involve human-robot interaction such as handshaking during socialization or handing tools to a surgeon during an operation. For the interaction to be successful, the robot is supposed to recognize and forecast limbs’ pose and position precisely so that it can provide a rapid and proper response (Tang et al., 2018). Human motion forecasting proved to be important for plenty of tasks including action detection, action recognition and action analysis in computer vision, body pose estimation as well as motion synthesis in computer graphics, virtual and augmented reality, etc. (Pavllo at al., 2018). Humans by nature are very flexible and can perform complex movements that are subject not only to the physical laws but also to the intentions of the moving person. Thus, human motion is inherently highly stochastic and non-deterministic which makes the task of modeling human motion very challenging. (Martinez et al., 2017). Specifically, many future poses are of high probability for the same set of observed sequence poses, thus making the task of long-term prediction very complex and non-trivial (Pavllo et al., 2018).

Commonly, the literature refers to the task of long-term motion prediction as *motion generation* which is of special interest to the computer graphics community specifically for the animation industry. On the other hand, the task of short-term motion prediction is commonly referred to as *motion prediction* which mostly concerns the community of computer vision. (Pavllo et al., 2018). The former is harder to validate quantitatively, therefore a qualitative metric, specifically human judgment is crucial while the latter can be validated quantitatively typically by measuring the mean-squared error in the angle space (Martinez et al., 2017).

1.2 Recurrent Neural Networks and Motion Prediction

Deep learning methods have proved to be successful for many tasks including pattern recognition and human motion prediction (Pavllo et al., 2019).

Particularly, Recurrent neural networks (RNNs) have shown good performance in predicting future 3D human poses not only in the short-term motion prediction

(Fragkiadaki et al., 2015) but also in the long-term motion generation (Martinez et al., 2017).

Unlike traditional methods which require expert knowledge about human motion upfront, RNNs like other deep learning methods can be trained to automatically learn representations that generalize to novel tasks depending on the used network structure and the task at hand (Butepage et al., 2017). RNNs excel at processing sequential data because of their ability to capture temporal dependencies between elements of a sequence. Unlike traditional neural networks known as feedforward neural networks (FNNs), RNNs have internal loops to persist information allowing them to remember the context of previously seen inputs.

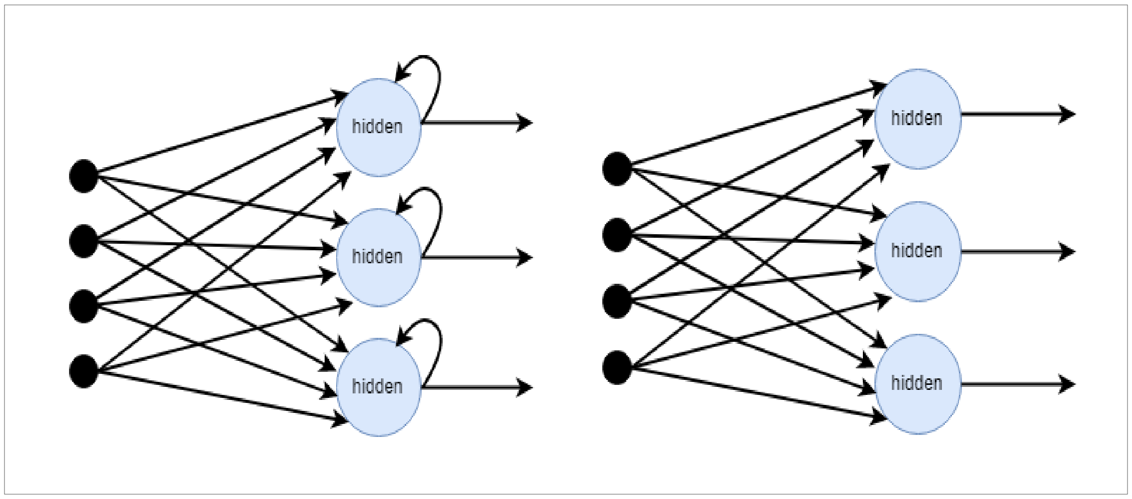


Figure 1.2: RNN vs FNN.
(left) Recurrent neural network (RNN). (Right) Feed forward neural network (FNN). Figure adopted from (Mulder et al., 2015).

FNNs process inputs at each iteration independently meaning that there is no context to be preserved. i.e. FNNs lack any form of memory. In the context of language modeling, machine translation as an example, FNNs can take a fixed number of previous words to predict the incoming word, a drawback that results from how FNNs are built. As a result, all words seen in previous iterations are forgotten even though those necessary to detect the next word. However, RNNs theoretically can preserve arbitrary context lengths. (Mulder et al., 2015). Figure 1.2 illustrates how RNNs differ from FNNs. RNNs will be explored in depth in chapter 3.

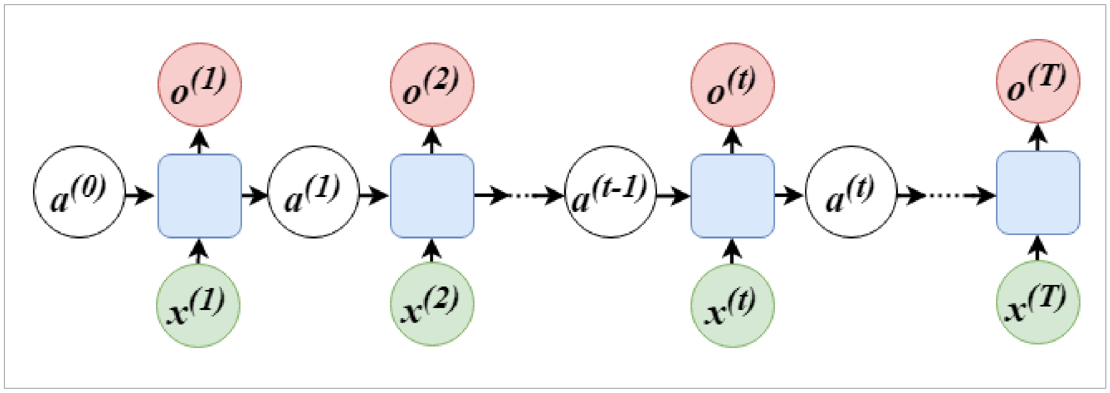


Figure 1.3: Unidirectional RNN.

Some applications may require information not only from the past of the input sequence but also from the future. One can consider the case of speech recognition, if there happens to be a word with two different interpretations that both look plausible, it might be crucial to take into account future words alongside with past words to determine the current word (Goodfellow et al., 2016). Bidirectional RNN (BRNN) was invented to fulfill that need (Schuster & Paliwal, 1997).

In a nutshell, BRNN combines two RNNs, one of them processes the sequence in the forward direction starting from the beginning of the sequence (forward through time) while the other one processes the sequence in the backward direction starting from the end of the sequence (backward through time) (Goodfellow et al., 2016). Figures 1.3 and 1.4 show how a unidirectional RNN differs from a bidirectional RNN. BRNNs will be discussed further in chapter 3, section 3.9.

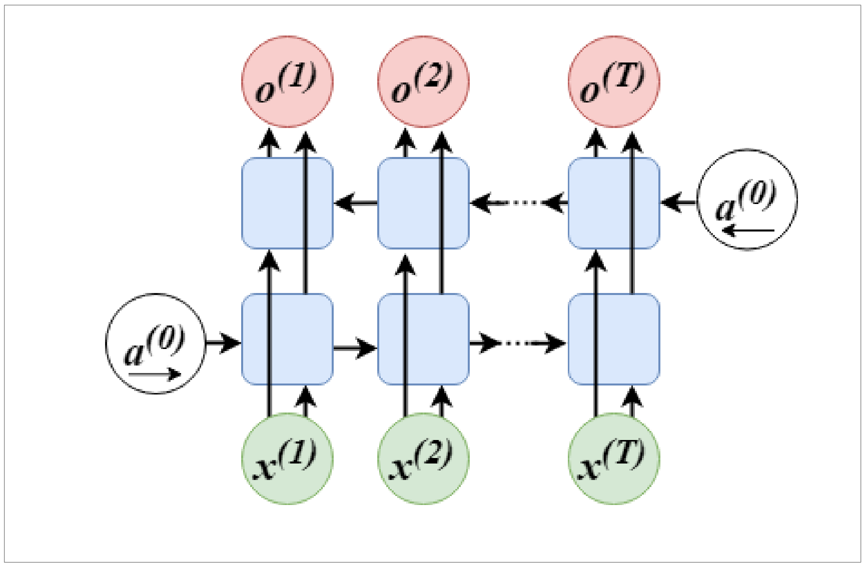


Figure 1.4: Bidirectional RNN (BRNN).
Figure adapted from (Amidi & Amidi).

1.3 Motion Data

The task of modeling human motion relies often on motion data acquired through motion capture technology. Mainly, Motion capture (MOCAP) is a cost-effective technique used to acquire highly realistic motion data by recording the performance of live actors to be converted later to mathematical representation and consequently applied to a mathematical model (Dean, 2016). In general, MOCAP uses an articulated model or *skeleton* that expresses the human joint chain which imposes certain parent-child relationships between joints (Wang et al., 2014). Among several MOCAP datasets that are publicly available (Sigal et al., 2009; Ionescu et al., 2014; Carnegie Mellon University - motion capture library), Human3.6M is currently the largest dataset with 3.6 Million accurate 3D human poses obtained by recording 15 different activities of 11 professional actors using accurate marker-based motion capture system namely Vicon system (Ionescu et al., 2014). Two common pose parametrizations considered in the literature are provided by the Human3.6M including *relative 3D joint positions representation* and *Kinematic representation* with a full skeleton of 32 joints for both representations (Ionescu et al., 2014; Ionescu et al., 2011).

As in previous works of Martinez et al., (2017) and Fragkiadaki et al., (2015), Human3.6M dataset is used in this work to assess quantitatively the use of bidirectional recurrent Neural Networks (BRNNs) in human motion prediction task.

1.4 Overview of Motion Capture Data

Conventional methods of character animation are very time consuming and require skillful animators to pose a character with the aid of specialized animation software. As a result, a need for better alternatives arises to fulfill the increasing demands of the animation industry. MOCAP technology provides a solution that creates better looking 3D animation in a shorter amount of time.

A comprehensive definition is given by Dyer, Martin, & Zulauf, (1995) which states that MOCAP:

> *involves measuring an object's position and orientation in physical space, then recording that information in a computer-usable form. Objects of interest include human and non-human bodies, facial expressions, camera or light positions, and other elements in a scene.*

The process of capturing motion is accomplished by using either physical information or image information provided by sensors to reconstruct the joints of the skeleton. Depending on the used techniques, MOCAP could be classified to sensor-based and image-based motion capture (Mulder et al., 2015). Sensor-based MOCAP involves the use of physical sensors including but not limited to inertial, optical and pressure sensors whereas image-based MOCAP involves the use of single or multiple cameras to capture human motion by acquiring information from colored or depth images (Mulder et al., 2015).

An extensive presentation of different motion capture techniques can be found in (Mulder et al., 2015). Another classification divides MOCAP into marker-based and marker-less depending on the tracking technology (72.b Documentation with motion capture, 2012). Generally, marker-based MOCAP offers a way to acquire animations directly from live actors by attaching markers to the body of the subject who wears a motion-capturing suite as shown in Figure 1.5, and then record the motion by tracking key points in the space over time and finally, converting them to 3D digital form (Meredith & Maddock, 2001). Figure 1.5 shows a set of markers being attached to the actor’s body. The less common alternative is the marker-less MOCAP which doesn’t involve the use of any artificial enhancements of the object or the environment other than the sensors (72.b Documentation with motion capture, 2012).

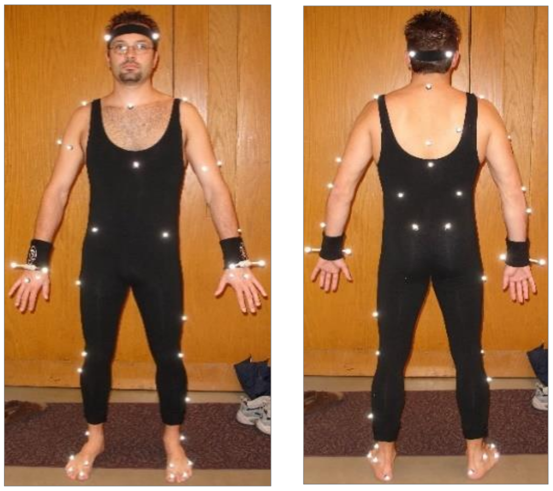


Figure 1.5: Marker-based MOCAP.
An actor wearing a motion-capturing suite with a set of markers attached to his body (Carnegie Mellon University - motion capture library - info).

1.2.3 **MOCAP File Formats**

Before mentioning the most common MOCAP file formats, it would be useful to define some terminologies that are necessary to understand these formats.

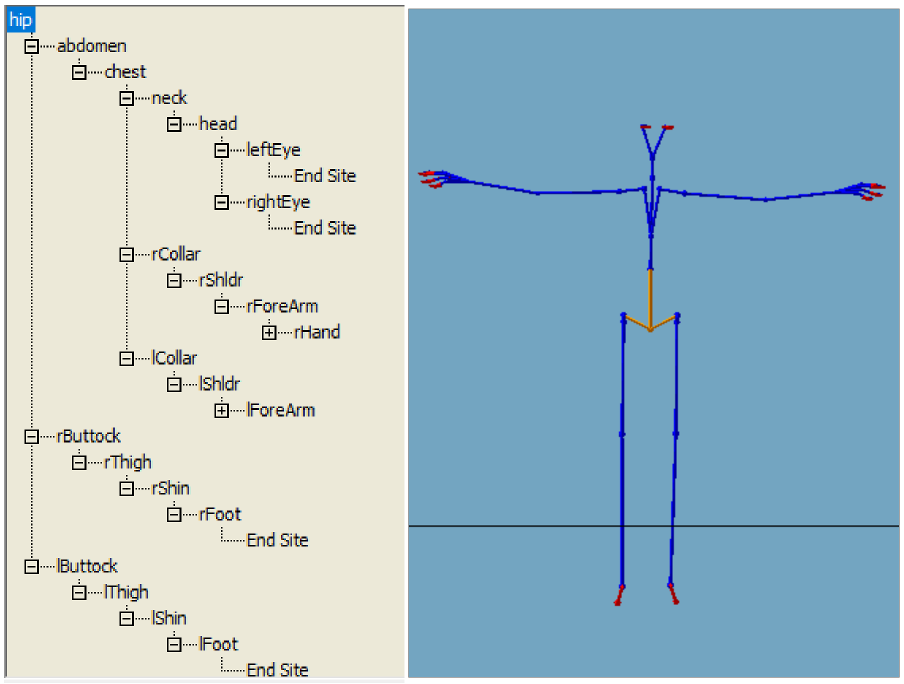
Skeleton: The entire character that motion data is applied to make animation. The skeleton consists of a set of bones. Figure 1.6 shows a hierarchical skeleton alongside the hierarchy of the bones (Meredith & Maddock, 2001).

Bone or joint: The smallest entity in the motion that is subject to individual translational or/and rotational changes throughout the animation. Bones are connected by joints which are related to each other by a parent-child relationship. The movement of the joints down the hierarchy i.e. children joints, is affected by the movement of joints higher in the hierarchy i.e. parent joints (Meredith & Maddock, 2001). The root joint as illustrated in Figure 1.6 is the hip joint. The hierarchical structure of the of the bones is shown to the left of Figure 1.6. Different datasets have different number of joints. For example, the number of joints composing the skeleton of Human3.6M is 32 while it's 57 in CMU dataset.

Degree of freedom / channel: Translation and rotation changes can be applied to joints over time to generate animation. These changes represent degree of freedoms (DOFs) of joints. Usually, a joint may have between 1 to 6 DOFs (Khan, et al., 2017).

Frame: Any animation is composed of a set of frames that when played consecutively generate motion. A single frame of motion contains channel DOF data for every bone in the skeleton (Meredith & Maddock, 2001).

MOCAP data comes into different formats. One of the most common formats is Biovision Hierarchy (BVH) format with .bvh extension. It was developed originally at Biovision and gained its popularity since then due to its simplicity (72.b Documentation with motion capture, 2012).
Any BVH file is comprised of 2 sections, the hierarchy section which contains information about the hierarchy of the skeleton and the initial pose as shown in Figure 1.7 and the motion section which contains information about the channels for every joint as shown in Figure 1.8.



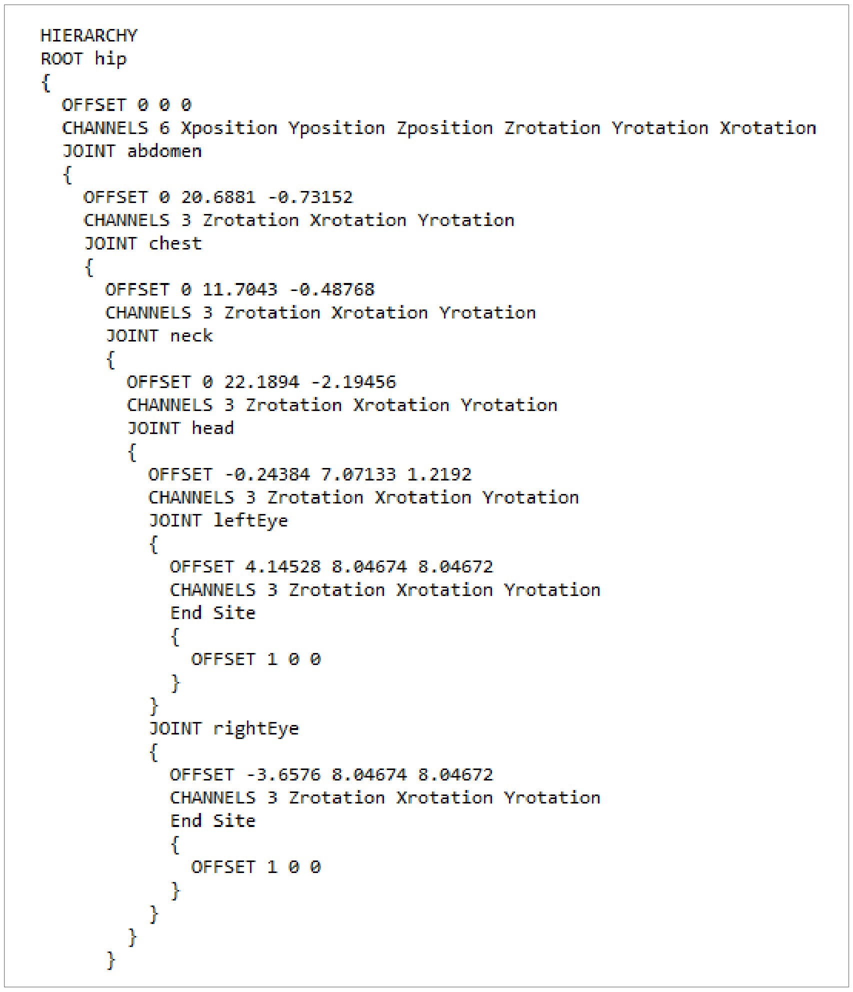
```
hip
⊟····abdomen
│    ⊟····chest
│    │    ⊟····neck
│    │    │    ⊟····head
│    │    │         ⊟····leftEye
│    │    │         │    └·····End Site
│    │    │         ⊟····rightEye
│    │    │              └·····End Site
│    │    ⊟····rCollar
│    │    │    ⊟····rShldr
│    │    │         ⊟····rForeArm
│    │    │              ⊞····rHand
│    │    ⊟····lCollar
│    │         ⊟····lShldr
│    │              ⊞····lForeArm
⊟····rButtock
│    ⊟····rThigh
│         ⊟····rShin
│              ⊟····rFoot
│                   └·····End Site
⊟····lButtock
     ⊟····lThigh
          ⊟····lShin
               ⊟····lFoot
                    └·····End Site
```

Figure 1.6: An articulated body model or skeleton illustrates the hierarchy of bones. The root i.e. hip bone is highlighted in orange.

Specifically, the hierarchy section starting with HIERARCHY keyword is used to determine the structure of the skeleton based on the joint's hierarchy. ROOT keyword that comes afterwards refers to a common root joint which indicates the start of the skeletal structure. Following the root is a chain of joints connected to each other with a parent-child relationship and encapsulated within a pair of curly braces where each child joint is proceeded with JOINT keyword.

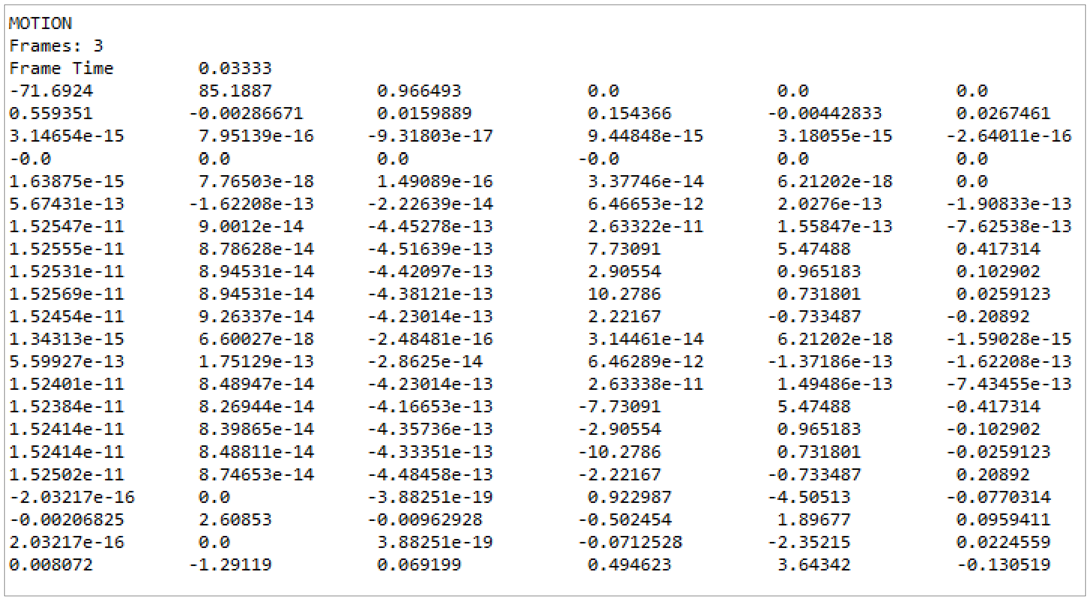
However, the end of the chain is indicated with (End site) keyword which refers to an end-effector that is, the last joint in a chain with no children where offset values of this joint indicates the bones' length and orientation.

Each joint has a position indicated by the three numbers following OFFSET keyword. These numbers represent x y z relative positions or translations of a joint with respect to its parent. In the case of the root joint, these numbers represent a global position. Furthermore, offset values determine implicitly the length and orientation of the parent bone. CHANNELS keyword specifies the DOFs of a joint. The order of channels for every joint matches the order of data in the motion section of the file. Particularly, the first 6 values in the motion section correspond to the channels defined for the root joint in the hierarchy section in the same order they appear with and the next 3 values correspond to the channels of the child joint that comes next in the hierarchy in the same order they appear with and so forth.



```
HIERARCHY
ROOT hip
{
  OFFSET 0 0 0
  CHANNELS 6 Xposition Yposition Zposition Zrotation Yrotation Xrotation
  JOINT abdomen
  {
    OFFSET 0 20.6881 -0.73152
    CHANNELS 3 Zrotation Xrotation Yrotation
    JOINT chest
    {
      OFFSET 0 11.7043 -0.48768
      CHANNELS 3 Zrotation Xrotation Yrotation
      JOINT neck
      {
        OFFSET 0 22.1894 -2.19456
        CHANNELS 3 Zrotation Xrotation Yrotation
        JOINT head
        {
          OFFSET -0.24384 7.07133 1.2192
          CHANNELS 3 Zrotation Xrotation Yrotation
          JOINT leftEye
          {
            OFFSET 4.14528 8.04674 8.04672
            CHANNELS 3 Zrotation Xrotation Yrotation
            End Site
            {
              OFFSET 1 0 0
            }
          }
          JOINT rightEye
          {
            OFFSET -3.6576 8.04674 8.04672
            CHANNELS 3 Zrotation Xrotation Yrotation
            End Site
            {
              OFFSET 1 0 0
            }
          }
        }
      }
```

Figure 1.7: The hierarchy section of a BVH file with (.bvh) extension.



```
MOTION
Frames: 3
Frame Time      0.03333
-71.6924        85.1887         0.966493        0.0             0.0             0.0
0.559351        -0.00286671     0.0159889       0.154366        -0.00442833     0.0267461
3.14654e-15     7.95139e-16     -9.31803e-17    9.44848e-15     3.18055e-15     -2.64011e-16
-0.0            0.0             0.0             -0.0            0.0             0.0
1.63875e-15     7.76503e-18     1.49089e-16     3.37746e-14     6.21202e-18     0.0
5.67431e-13     -1.62208e-13    -2.22639e-14    6.46653e-12     2.0276e-13      -1.90833e-13
1.52547e-11     9.0012e-14      -4.45278e-13    2.63322e-11     1.55847e-13     -7.62538e-13
1.52555e-11     8.78628e-14     -4.51639e-13    7.73091         5.47488         0.417314
1.52531e-11     8.94531e-14     -4.42097e-13    2.90554         0.965183        0.102902
1.52569e-11     8.94531e-14     -4.38121e-13    10.2786         0.731801        0.0259123
1.52454e-11     9.26337e-14     -4.23014e-13    2.22167         -0.733487       -0.20892
1.34313e-15     6.60027e-18     -2.48481e-16    3.14461e-14     6.21202e-18     -1.59028e-15
5.59927e-13     1.75129e-13     -2.8625e-14     6.46289e-12     -1.37186e-13    -1.62208e-13
1.52401e-11     8.48947e-14     -4.23014e-13    2.63338e-11     1.49486e-13     -7.43455e-13
1.52384e-11     8.26944e-14     -4.16653e-13    -7.73091        5.47488         -0.417314
1.52414e-11     8.39865e-14     -4.35736e-13    -2.90554        0.965183        -0.102902
1.52414e-11     8.48811e-14     -4.33351e-13    -10.2786        0.731801        -0.0259123
1.52502e-11     8.74653e-14     -4.48458e-13    -2.22167        -0.733487       0.20892
-2.03217e-16    0.0             -3.88251e-19    0.922987        -4.50513        -0.0770314
-0.00206825     2.60853         -0.00962928     -0.502454       1.89677         0.0959411
2.03217e-16     0.0             3.88251e-19     -0.0712528      -2.35215        0.0224559
0.008072        -1.29119        0.069199        0.494623        3.64342         -0.130519
```

Figure 1.8: The motion section of a BVH file with (.bvh) extension. Channel information of the first frame of motion is shown.

The MOTION section of the file starts with number of frames and frame's duration or frame rate and finally the channel data for each bone as the appear in the hierarchy section of the file. In fact, the channel data is nothing more than the animation data of each bone through time. An older MOCAP format is the Biovision Action File (BVA) with .bva extension which is similar to BVH file in many ways but with key differences, the most important of which is that BVA can't store motion for a hierarchical skeleton. The motion of a child bone doesn't depend on the motion of any other joints (Lander, 1998). Another common format is the Hierarchical Translation Rotation (HTR) format with .htr extension which was developed by Motion Analysis company with the aim of solving some problems of BVH files (Du et al., 2016). Acclaim motion file is yet another popular format designed by Acclaim gaming company. This format consists of two files. The first file is the Acclaim Skeleton File ASF with .asf extension which contains information about the hierarchy and the initial pose of the skeleton. The second file is the Acclaim Motion Capture (AMC) file with .amc extension which is used to save motion data of the skeleton. This separation between the hierarchy information and the motion data is useful since only one ASF file can be used in a motion capture session with multiple AMC files (Lander, 1998). Coordinate 3D (C3D) is a binary format defined by the National Institute of Health mainly to be used in the biomechanical research. Unlike previously described formats which merely contain information about 3D positions and orientations, C3D supports wide diversity of data that is useful for the biomechanical research (72.b Documentation with motion capture, 2012).

1.5 **Literature Review**

The scope of this work is focused primarily on the use of deep learning methods specifically recurrent neural networks (RNNs) in the task of human motion prediction. This section will briefly review the recent works that employ deep learning to predict human motion with a special focus on RNNs. The reviewed works mainly rely on motion data acquired from motion capture MOCAP techniques. It's worth mentioning that the literature generally refers to the process of producing motion for less than 500ms as short-term prediction which is the main concern of this research. Otherwise, if the generated motion exceeds 500ms it's referred to as long-term generation or synthesis. For skeleton-based tasks such as action recognition, deep learning approaches have outperformed traditional methods (Tang et al., 2018). An early work of Taylor et al., (2007) suggested a conditional restricted Boltzmann machine (CRBM) to model motion where sampling is needed for inference (Martinez et al., 2017). The model was experimented with few motions including walking, jogging and running (Butepage et al., 2017).Encoder-Recurrent-Decoder (ERD) architecture for human motion prediction was proposed by Fragkiadaki et al., (2015). The proposed model is an RNN that integrates a nonlinear encoder and a decoder before and after the recurrent layers. To ease accumulated errors during training which eventually lead to predicting unrealistic motion, the authors of Fragkiadaki, et al., (2015) suggested the use of noise scheduling by adding increasing amounts of random noise gradually to the input data during training. Although noise scheduling has alleviated the problem to some degree, it's hard to tune noise scheduling in practice (Martinez et al., 2017).

Another architecture, namely 3-layer long short-term memory network (LSTM-3LR) was proposed in the same study (Fragkiadaki, et al., 2015). ERD and LSTM-3LR both consist of concatenated Long Short-Term Memory LSTM units, but the former has a non-linear space encoder for data preprocessing. Recently, the authors of Martinez et al. (2017) have further extended the work of Fragkiadaki et al. (2015) by proposing a significantly simpler RNN architecture with one Gated Recurrent Unit (GRU) instead of using the more complicated LSTM unit which was used in previous works (Jain et al., 2016; Fragkiadaki et al., 2015). To address the problem of discontinuity between the last frame of the input sequence and the first frame of the predicted sequence, authors of Martinez et al., (2017) decided to model velocities instead of absolute joint angles by deploying a residual architecture that models first-order motion derivatives. The use of a residual architecture leads to smoother and more accurate predictions. Earlier, the work of Jain et al., (2016) tried to combine high level spatio-temporal graphs (st-graphs) with RNNs by developing a structural RNN (SRNN) which transforms an st-graph into a trainable and scalable RNNs mixture. By combining st-graphs with RNNs, authors aimed at exploiting the power of RNNs at modeling sequential data and to compensate for their lack of an intuitive spatio-temporal structure that can capture spatio-temporal relations between joints and hence, is suitable to represent skeletal data. As in Fragkiadaki et al., (2015), noise scheduling was employed to lessen the effect of accumulated errors. As a result, SRNN generates plausible motions for actions like eating. For more challenging actions like discussion, SRNN doesn't generate unrealistic motions (Martinez et al., 2017). In the previously mentioned works (Martinez et al., 2017; Fragkiadaki et al., 2015; Taylor et al., 2007; Butepage et al., 2017; Jain et al., 2016), the traditional Euclidean loss is used to measure the distance between the prediction and the ground truth during training. However, a novel loss function, namely the geodesic loss, was proposed by Gui et al., (2018) to replace the Euclidean loss. Generally, 3D rotations between joint angles are used to represent motion frames. Unlike Euclidean distance, the geodesic distance exploits the geometric structure of 3D rotations and thus, avoids inaccuracies in predictions and convergence to mean pose after longer time periods that follow from the use of the Euclidean distance as a loss function. Inspired by generative adversarial networks (GANs), the authors introduced two global discriminators to validate the predicted motion while casting the predictor as a generator. Basically, the generator is a decoder-encoder network while both discriminators are RNNs. The first global discriminator is a fidelity discriminator that validates the overall plausibility of the predicted motion whereas the second one is a continuity discriminator that validates the coherence of the predicted sequence with the input sequence. The recent work of Pavllo et al., (2019), uses Quaternions representation to represent joint angles, which was overlooked by previously mentioned works. Because Euler angles representation suffers from singularities and discontinuities, using it will ultimately lead to the notorious problem of exploding gradients which makes the training very difficult or impossible. Alternatively, Exponential maps representation was used in previous works which alleviates these issues to some degree but doesn't eliminate them. In addition to using Quaternions, the authors suggested a loss function that penalizes absolute joint position errors instead of joint angle errors. Additionally, both convolutional and recurrent architectures were investigated and evaluated on short-term prediction and long-term generation.

Chapter 2

DEEP LEARNING

2.1 Introduction

Artificial Intelligence, machine learning, deep learning, despite being related and sometimes overlapping concepts, they are not identical indeed. Different researchers may have different views about how these fields are related. Figures 2.1 and 2.2 present different views of the relations between these fields.

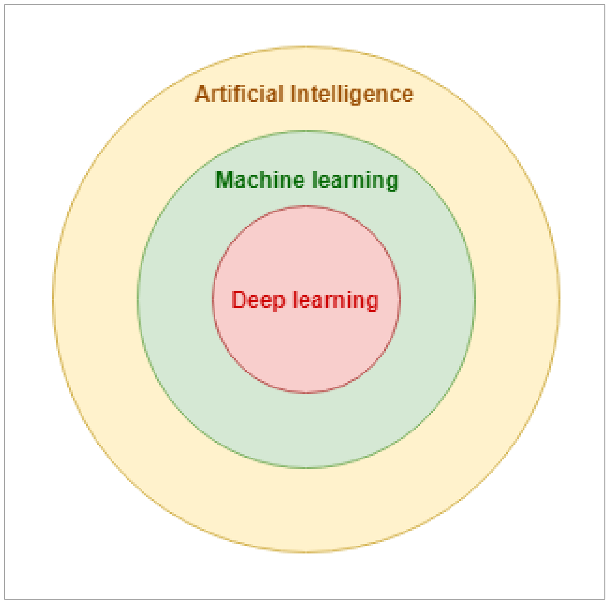


Figure 2.1: The relationship between artificial intelligence, machine learning, and deep learning according to Chollet. Figure adapted from (Chollet, 2017a).

Artificial intelligence or AI could be defined as "*the effort to automate intellectual tasks normally performed by humans*." (Chollet, 2017a). Even if one chose to go with the opinion that considers machine learning a part of AI, there are many approaches in AI that don't involve any learning.

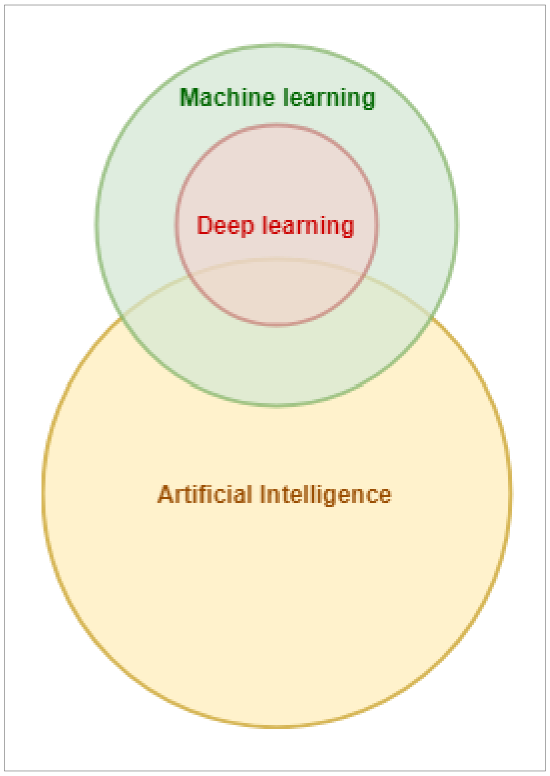


Figure 2.2: The relationship between Artificial intelligence, machine learning and deep learning according to Trask. Figure adapted from (Trask, 2019).

For example, early chess programs include merely hardcoded rules without any learning mechanisms. Machine learning, on the other hand, seeks to grant computers the ability to learn without being explicitly programmed to do so (Trask, 2019). Figure 2.3 points out the difference between classical programming and machine learning.

The former is about taking as input handcrafted rules and data that will be processed according to these rules to get answers as output whereas the latter is about taking data and answers as input to get the rules. These rules are used then with new data to get the expected answers. Classical programming or alternatively symbolic AI, provides solution to well-defined, logical problems but fails to solve more complex problems such as language translation, image and speech recognition etc. Without being explicitly programmed, a machine learning program is trained by being exposed to many training examples to learn the rules by capturing certain patterns in the training data. At its essence, machine learning observes a pattern and aims to imitate that pattern either directly or indirectly.

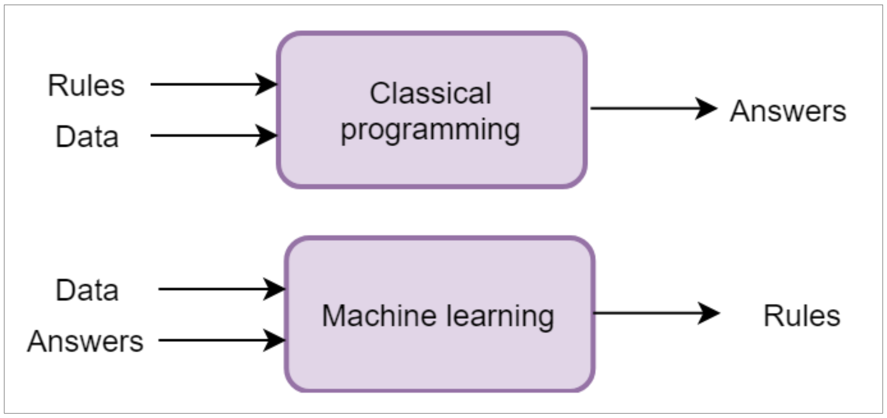
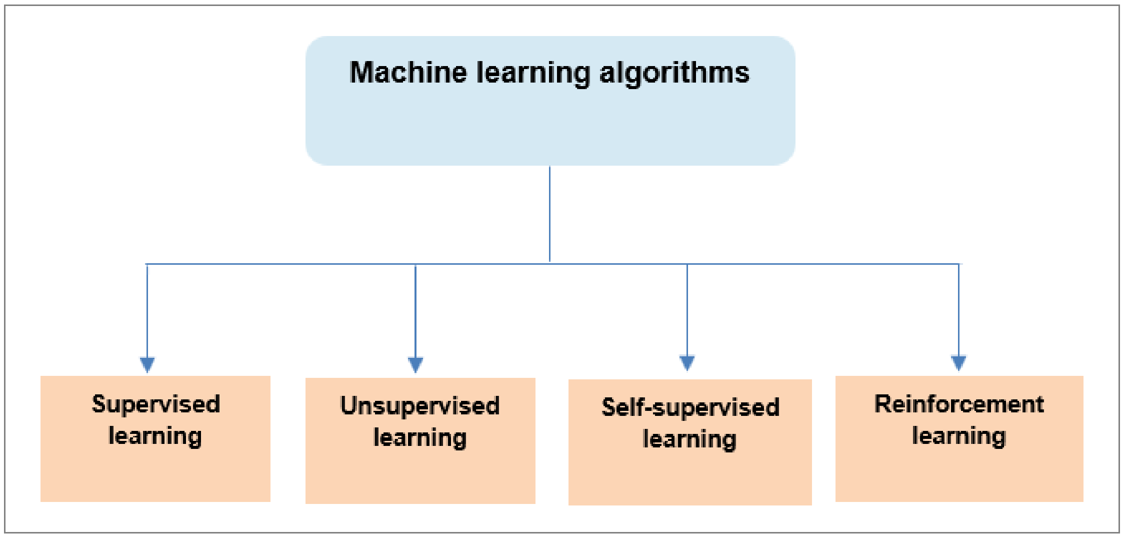


Figure 2.3: Machine learning vs classical programming.
Figure adapted from (Chollet, 2017a).

2.2 Machine learning vs deep learning

Mainly, machine learning learns how to map an input image for example, to some output or a target such as a label or a class. This is done by looking at too many training examples of inputs and targets (Chollet, 2017a). Deep learning resembles machine learning in the learning process but with a distinction. When it comes to deep learning, the learning is done through several successive layers which work as filters that purify the incoming information to become increasingly meaningful. This is done using models loosely inspired by the human brain known as neural networks. Before delving into the fundamental building blocks of deep learning, the neural networks, a brief overview of the categories of machine learning algorithms will be provided next.

2.3 **Machine learning categories**



e

Figure 2.4: Machine learning algorithms.

Sometimes, these categories may overlap and may not be perfectly separable, nonetheless, they are useful to give a broader view of different kinds of machine learning algorithms. The following sections present briefly the categories of machine learning.

a) Supervised learning:

Most machine learning algorithms used successfully in industry today fall under the supervised learning category (Chollet, 2017a). Supervised learning aims at learning to predict a known target (usually human-annotated labels) given an input.

Usually, the input or the feature is denoted as x and the target or the label is denoted as y. For the i_{th} example in a dataset, (x_i, y_i) refers to the (feature, target) pair of the i_{th} example from the dataset. A dataset is a collection of n examples each of which consists of an (x_i, y_i) pair (Zhang et al., 2019). Given a set of labeled examples (x_i, y_i) of inputs and correct labels, the goal of a supervised learning algorithm is to learn a model f that maps an input x_i to a prediction $f(x_i) = \hat{y}_i$ by adjusting a set of parameters θ during the training phase to get the prediction as close possible to the correct label. Most supervised learning algorithms are either classification or regression tasks (Chollet, 2017a).

What differentiate a regression task from a classification task is the type of the output. A regression task aims at learning a mapping f from an input x to a continuous real valued y. A classic example of a regression problem is predicting house prices given a set of features such as the square footage, number of rooms, etc. The target y may be any arbitrary number in some range. However, for a classification problem y may take only a small number of discrete values. Recognizing a handwritten digit is a classification task in which the goal is to look at an image as a set of pixels and predict which class or digit the image belongs among a set of discrete values (classes) (Zhang, et al., 2019).

b) Unsupervised learning:

Unlike supervised learning, no correct answer is provided to the model to be duplicated. The task of an unsupervised algorithm is to explore a dataset and attempt to find some patterns in the data (Trask, 2019). Clustering is a well-known example of unsupervised learning where the algorithm attempts to group the data into labeled clusters by exploring correlations that present in the data (Chollet, 2017a; Trask, 2019). According to (Trask, 2019), any unsupervised task can be seen as a form of clustering. Grouping some users based on their browsing activities is an example of a clustering problem (Zhang, et al., 2019).

c) Self-supervised learning:

This is a recent branch of machine learning in which it attempts to learn a mapping from pairs of inputs and outputs. Like supervised learning, the learning is supervised by the labels but with a key difference that the labels are no longer human-annotated but rather are generated from the dataset automatically without human intervention (Chollet, 2017a). An example of a self-supervised learning is the Autoencoder network which aims at learning a compact representation of the input so that it can reconstruct the input unmodified using that representation.

d) Reinforcement learning:

This branch of machine learning algorithms addresses the case of an agent that interacts with its environment over time and at each timestep t the agent learns to choose actions that will maximize some reward r_t. Applications may include robotics and AI for video games (Zhang, et al., 2019).

2.4 Neural Networks

As mentioned before, deep learning uses neural networks to learn meaningful representations of the input data successively through a bunch of layers that work as a multistage distillation and purification system of the incoming information to make it as closer as possible to the final output (Chollet, 2017a). Figure 2.5 shows a deep neural network with 4 layers where each layer is trying to learn a useful representation of data. The representations are getting closer to the target as they approach the final output. As presented in Figure 2.6, a neural network is a universal approximator of a function f that maps an input (an image of a handwritten digit) to a target (a label). For example, the handwritten classifier $y = f(x)$ is a neural network that maps an input to a category. The classifier in Figure 2.6 defines a mapping $y = f(x; \theta)$ that maps an input x to a category y. Given enough training data and computational time, the neural network of the classifier can learn through training process the optimal set of parameters θ that result in the best possible approximation to the function f (Goodfellow et al., 2016).

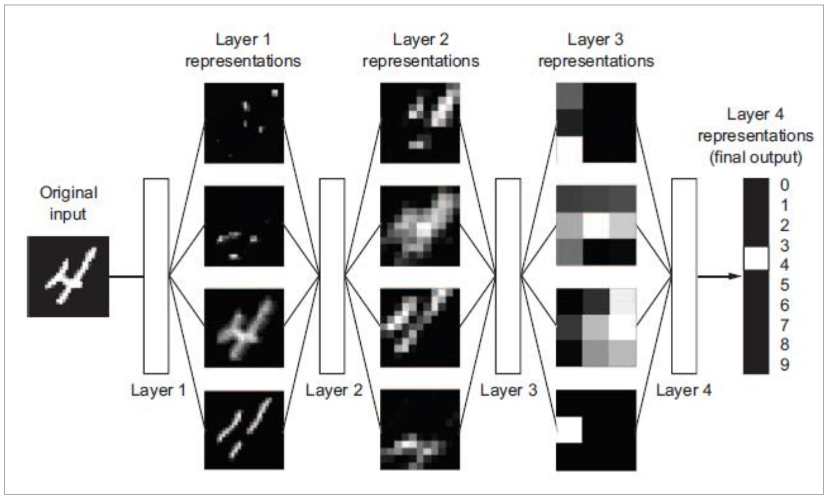


Figure 2.5: Deep neural network for handwritten digit classification (Chollet, 2017a).

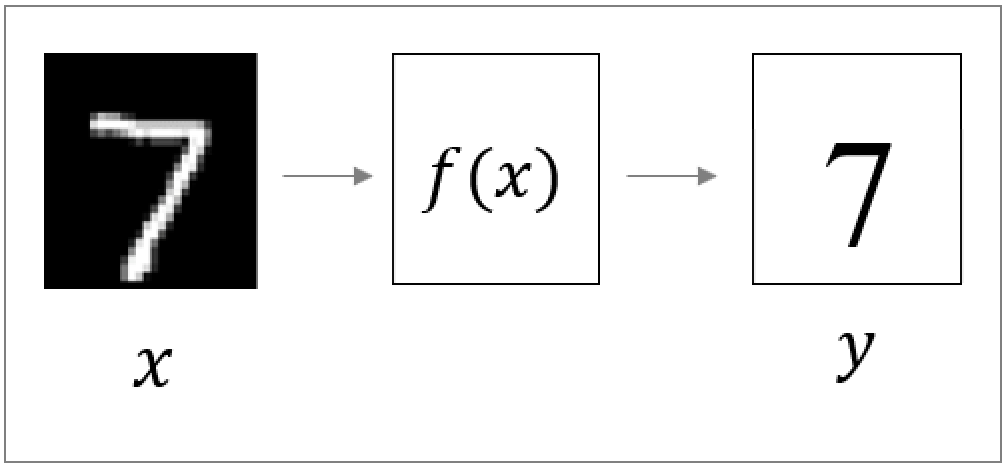


Figure 2.6: A neural network for handwritten digit classification shown as a function that maps an input to a target.

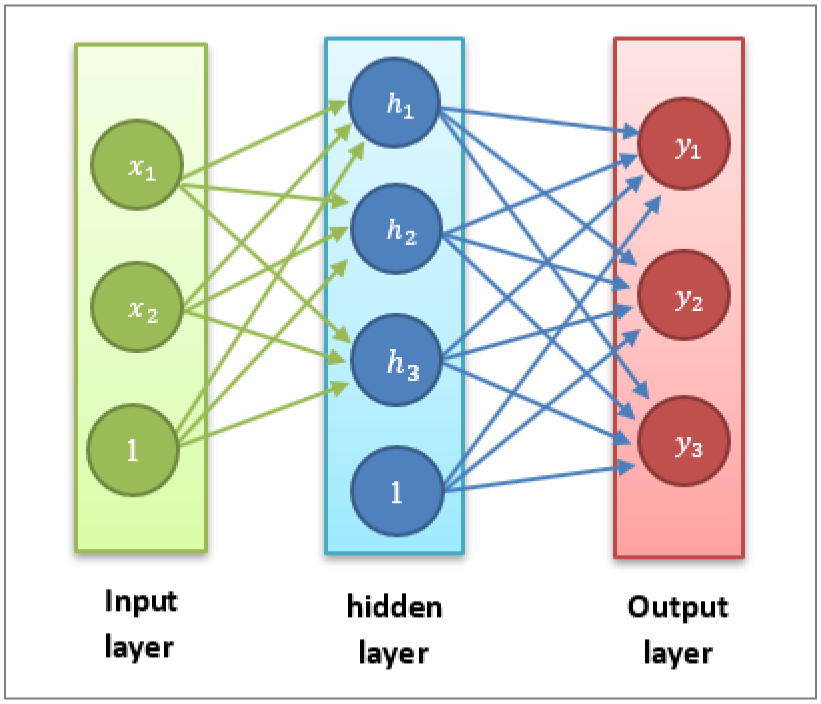


Figure 2.7: A neural network with three layers.

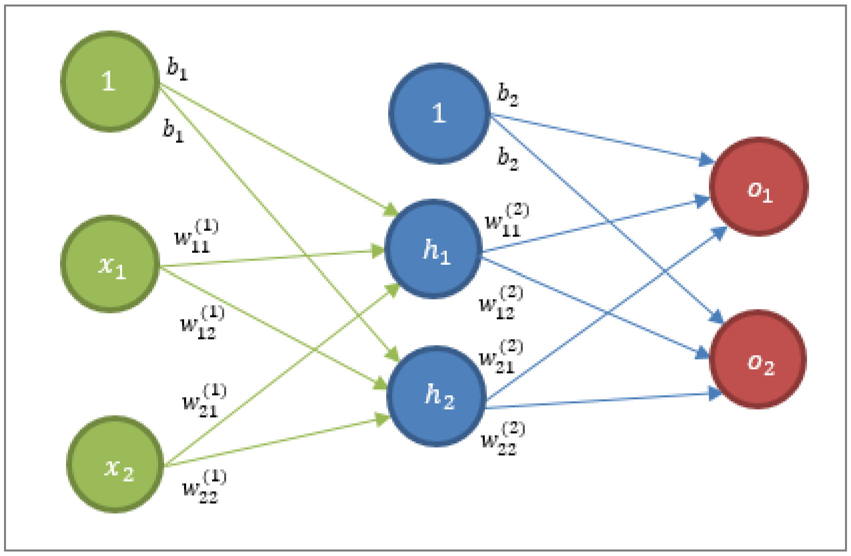


Figure 2.8: A simple neural network.

Each layer in a neural network is a simple data transformation that's controlled by a set of weights. A simple neural network is shown in Figure 2.7 which consists of 3 layers, an input layer, a hidden layer and an output layer with 3, 4 and 3 nodes respectively. The input layer doesn't involve calculations, it just passes the information to the next layer.

For the sake of clarity, a simplified version of the network in Figure 2.7 is shown in Figure 2.8 and will be used to illustrate the processes executed by a neural network. A single *node* or a neuron such as h_1, is the basic unit of computation in a neural network which takes an input and generates an output (known as the *activation* of the node) which in turn is used to calculate the inputs to the nodes of the next layer o_1 and o_2 in this case. Every node from one layer is connected to every other node from the next layer and these connections represent *weights* that are given based on the relative importance of an input to other inputs. For example, the input feature $x_1 \in \mathbb{R}$ that is connected to the node h_1 is attached to weight $w_{11}^{(1)} \in \mathbb{R}$. Notice that the superscripts used for weights in Figure 2.8 is in the form $w_{ij}^{k} \in \mathbb{R}$ which means that the weight is connecting the i_{th} node in the k_{th} layer to the j_{th} node in the $k_{th} + 1$ layer. $b_1 \in \mathbb{R}$ and $b_2 \in \mathbb{R}$ are the *biases*.

The bias is like the intercept in a linear equation. In fact, the bias is similar to a regular weight, but it's always multiplied by a constant activation which is 1. Weights and biases are trainable parameters that need to be tuned during the training process to get the *prediction* of the network as close as possible to the *target*. The reason that both weights and biases are needed for a successful learning will be mentioned shortly after discussing the activation functions. The final outputs of the network (predictions) are the activations of the output nodes $o_1$and o_2. To acquire the activation of a single hidden unit such as h_1, a linear combination (weighted sum) of the inputs x_1 and x_2 and the weights $w_{11}^{(1)}$, $w_{21}^{(1)}$ and the bias b_1 need to be calculated, followed by a non-linearity (*activation function*) σ_1 as shown in the following equations:

$$h_{1(in)} = w_{11}^{(1)} x_1 + w_{21}^{(1)} x_2 + b_1 \tag{2.1}$$

$$h_{1(out)} = \sigma_1 \left(h_{1(in)}\right) \tag{2.2}$$

Where:

$w_{11}^{(1)}$and $w_{21}^{(1)}$	:weights associated to the input features x_1 and x_2 respectively.
x_1 and x_2	:input features of single training example.
b_1	:bias of the hidden unit.
$h_{1(in)}$	:input to the hidden unit h_1.
σ_1	:activation function of all hidden units.
$h_{1(out)}$	:output of the hidden unit h_1 (the activation of h_1).

In the same manner, the following equations describe how h_2 is calculated:

$$h_{2(in)} = w_{12}^{(1)} x_1 + w_{22}^{(1)} x_2 + b_1 \tag{2.3}$$

$$h_{2(out)} = \sigma_1 \left(h_{2(in)}\right) \tag{2.4}$$

Before showing how the final output of the network is acquired, it should be noted that the prediction of the network is usually referred to as $\hat{y}$ (here, the symbol $o_{(out)}$ will be used but it is the same thing as $\hat{y}$ which might be used in other places in this document). The target of the network is referred to usually as y. As for the first prediction of the network $\hat{y}_1$ or $o_{1(out)}$, the following formulas show how to get the first prediction:

$$o_{1(in)} = w_{11}^{(2)} h_{1(out)} + w_{21}^{(2)} h_{2(out)} + b_2 \tag{2.5}$$

$$o_{1(out)} = \sigma_2\left(o_{1(in)}\right) \tag{2.6}$$

Where:

$w_{11}^{(2)}, w_{21}^{(2)}$	: weights associated to the outputs of the hidden units h_1 and h_2
$h_{1(out)}$, $h_{2(out)}$	: outputs (the activations) of the hidden units
b_1	: bias of the output unit
$o_{1(in)}$	: input to the output unit
σ_2	: activation function of the output units
$o_{1(out)}$	: first output of the neural network (the first prediction)

Similarly, $\hat{y}_2$ or $o_{2(out)}$ is given by the following equations:

$$o_{2(in)} = w_{12}^{(2)} h_{1(out)} + w_{22}^{(2)} h_{2(out)} + b_2 \tag{2.6}$$

$$o_{2(out)} = \sigma_2\left(o_{2(in)}\right) \tag{2.7}$$

2.4.1 **Activation Functions**

Each node in the neural network involves a linear transformation $wx + b$ followed by a non-linearity σ known as the activation function. Activation functions are crucial in neural networks because they allow the network to learn much richer representations of input data. Without using activation functions to break the linearity, the network will learn only linear transformations of the input data and there would be no benefit from stacking more layers as they can't extend the learned representations (Chollet, 2017a). Some of the most popular activation functions will be presented here along with their graphs, formulas and derivatives. Derivatives will be given since differentiating activation functions is an essential step in the training process of a neural network as will be discussed in the next section. One of the most widely used activation function for hidden units is the Rectified linear unit (RELU). As presented in Figure 2.9, RELU is a simple and efficient function that is defined as $max(0, x)$. It gives the output x if it is positive, otherwise it turns it into 0. Mathematically, RELU function is described as follows:

$$f(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases} \tag{2.8}$$

The derivative of the RELU is undefined at 0. For other values it is given below:

$$\acute{f}(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x > 0 \end{cases} \tag{2.9}$$

Other choices of the activation function include the Sigmoid function and the hyperbolic tangent function Tanh. Figure 2.10 shows the plot of the Sigmoid function and the following formula describes the Sigmoid function mathematically:

$$f(x) = \frac{1}{1+e^{-x}} \tag{2.10}$$

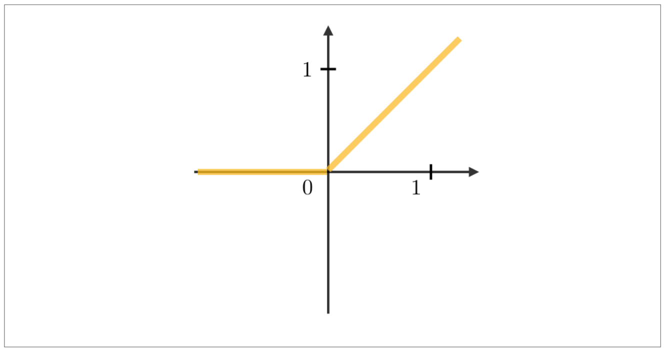


Figure 2.9: Rectified linear unit (RELU) (Amidi & Amidi).

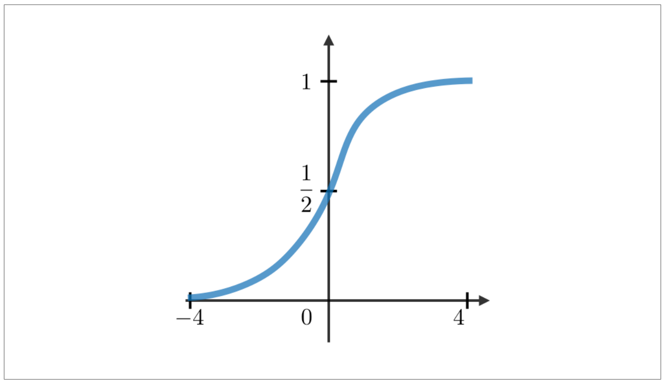


Figure 2.10: Sigmoid function (Amidi & Amidi).

As shown in Figure 2.10, the Sigmoid function takes values in the range $(-\infty, +\infty)$ and outputs values in the range (1,0) such that $f(x)$ approaches 1 as x approaches $+\infty$ whereas $f(x)$ approaches 0 as x approaches $-\infty$. Sigmoid has a nice derivative which is given by the following formula:

$$\acute{f}(x) = \frac{1}{1+e^{-x}} \cdot \left(1 - \frac{1}{1+e^{-x}}\right) \tag{2.11}$$

Hyperbolic tangent function (Tanh) is another function that might be used as an activation function to ca the hidden units to break the linearity. The plot of Tanh is shown in Figure 2.11. Below is the equation that defines Tanh mathematically:

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.12}$$

Tanh is a rescaling of the Sigmoid, it outputs values in the range (-1, 1). Tanh derivative is given by the following equation:

$$\acute{f}(x) = 1 - \left(\frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}\right)^2 \tag{2.13}$$

RELU is almost always used as an activation function in hidden units whereas Sigmoid and Tanh are less commonly used. As for the activation functions used in the output layer, Sigmoid is a natural choice for binary classification problems. For multi-class classification problems, the Softmax function, a generalization of the Sigmoid, is used to produce a probability distribution in which the outputs sum to 1.

The Softmax function is described by the following formula:

$$f_i(\vec{x}) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} \quad \text{for } i = 1, \dots, J \tag{2.14}$$

Expectedly, the Softmax has a similar derivative to the Sigmoid as shown by the following formula:

$$\acute{f}_i(\vec{x}) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} \cdot \left(1 - \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}\right) \quad \text{for } i = 1, \dots, J \tag{2.15}$$

However, for regression problems, usually a linear function is used as the activation of the output layer. Now that the notion of activation functions becomes clear, it is appropriate to answer the following question: why both weights and biases are needed in a neural network? The reason is that changing the weights affect the activation function differently than changing the bias. Changing the weights will affect the steepness of the curve while changing the bias would shift the curve either to the right or to the left.

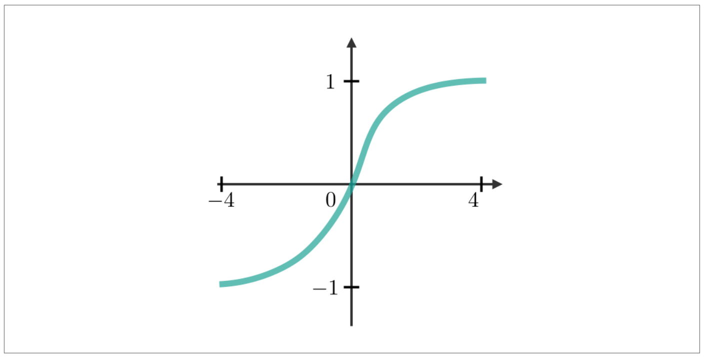


Figure 2.11: Hyperbolic tangent function (Tanh) (Amidi & Amidi).

Figures 2.12 and 2.13 show the effect of changing weights and biases on the Sigmoid function curve respectively.

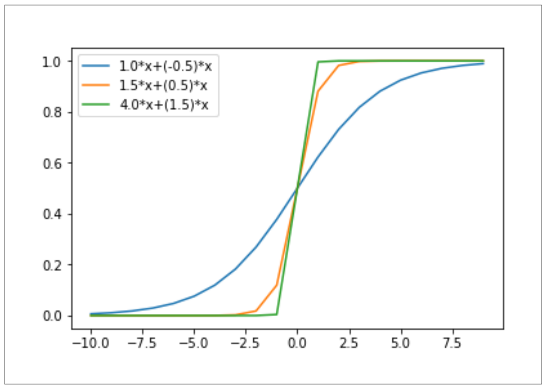


Figure 2.12: The effect of changing weights on the Sigmoid function curve (Bisht, 2018).

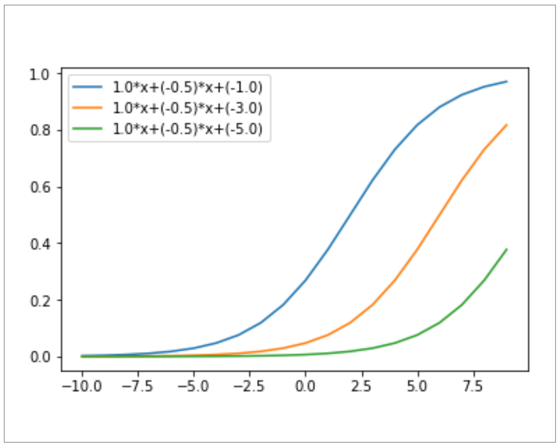


Figure 2.13: The effect of changing bias on the Sigmoid function curve (Bisht, 2018).

2.4.2 **Neural Networks in Action**

This section illustrates the procedure taken by a neural network during the training phase. The neural network in Figure 2.8 will be used as an example for this illustration. The neural network starts with a *forward pass* traversing the nodes from left to right to find the final outputs (the predictions). By the end of the forward pass, the predictions $o_{1(out)}$ and $o_{2(out)}$ or $\hat{y}_1$ and $\hat{y}_2$ would be calculated as shown previously in section 2.4 (equations 2.1 to 2.7). The network calculates the error by measuring how far its prediction is from the correct labels (targets) y_1 and y_2. This is done using a *cost function.* Sometimes, the *cost function* is referred to as the *objective function* or *error function*. However, the *loss function* term refers to the error of one training example where the c*ost function* refers to the average error over all training examples. The *cost function* is the average of the losses as follows:

$$L = \frac{1}{n}\sum_{i}^{n} L_i\left(y_{i,}\hat{y}_i\right) \tag{2.16}$$

Where:

L : total cost

L_i : loss of the of the i_{th} example

n : total number of examples

y_i : correct label (target) of the i_{th} example

$\hat{y}_i$: network’s final output (prediction) of the i_{th} example

Mean squared error or MSE, is one of the most widely used cost functions which works best for regression problems. MSE is given by the following equation:

$$L = \frac{1}{2n}\sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{2.17}$$

For classification problems, another popular cost function is used which is known as the cross-entropy cost function. For a binary classification problem, the cross-entropy cost function is given by the following equation:

$$L = -\frac{1}{n}\sum_{i=1}^{n} y_i\, log(\hat{y}_i) + (1 - y_i)log(1 - \hat{y}_i) \tag{2.18}$$

For the more general case of the multiclassification problems, the generalization of the cross-entropy function is given by the following formula:

$$L = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m} y_{ij}\, log\left(\hat{y}_{ij}\right) \tag{2.19}$$

Obviously, the larger the error the worse the prediction. At the beginning of training, the network is naïve and will produce large errors. In order to get the prediction of the network as close to the target as possible, the cost function i.e. the error needs to be minimized and here comes the role of *gradient descent*, an iterative optimization algorithm of finding the minimum of a function.

Gradient descent minimizes a function by iteratively following the direction of the steepest descent given by the negative of the gradient of that function. Figure 2.14 shows a graph of the cost function $L(w)$ vs the weight w. The given cost function in Figure 2.14 is assumed to be a single variable function to simplify the explanation but usually cost functions are multivariable functions with many parameters to optimize. The goal of the gradient descent is to minimize the cost function $L(w)$ through finding $\acute{L}(w)$, the derivative of $L(w)$ w.r.t the weight w.

The gradient is nothing but a generalization of the derivative where it is used for multivariable functions. It is a vector ∇L that points to the direction of the greatest increase of the function (local maxima). Intuitively, the negative of the gradient $-\nabla L$ points to the direction of the greatest decrease (local minima). For a multivariable function $L(w_1, \dots w_n)$, the gradient ∇L is a vector where each component is a partial derivative as follows:

$$\nabla L = \begin{bmatrix} \frac{\partial L}{\partial w_1} \\ \cdot \\ \cdot \\ \cdot \\ \frac{\partial L}{\partial w_n} \end{bmatrix} \tag{2.20}$$

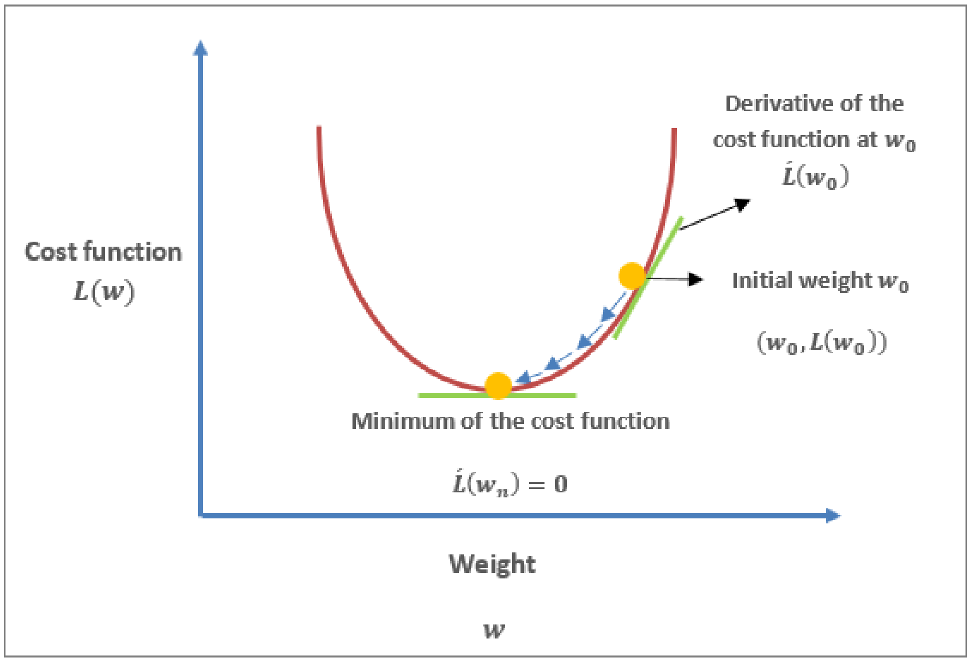

Derivative of the cost function at w_0 $\acute{L}(w_0)$
Cost function $L(w)$
Initial weight w_0
$(w_0, L(w_0))$
Minimum of the cost function
$\acute{L}(w_n) = 0$
Weight
w

Figure 2.14: Gradient descent algorithm.
Figure adapted from (Gradient Descent with Momentum, 2019).

The following are the steps of the gradient descent algorithm as shown in Figure 2.14:

1) Start with some initial weight w_0 (initialized randomly or assigned to 0).
2) Compute the derivative of the cost function $\acute{c}(w_0)$ at w_0.
3) Update the weight $w_i = w_i - \alpha\acute{L}(w_i)$ where α is the learning rate that determines the size of the gradient descent step (how big the step to take).
4) Repeat the steps until convergence or for a defined number of iterations.

Generally, the gradient descent update step according to the notation used in Figure 2.8 is as follows:

$$w_{ij}^{k} = \grave{w}_{ij}^{k} - \alpha \frac{\partial L}{\partial w_{ij}^{k}} \tag{2.21}$$

Where:

w_{ij}^{k} : new weight that connects the i_{th} node from the k_{th} layer to the j_{th} node from the $k_{th}+1$ layer

$\grave{w}_{ij}^{k}$: old weight that connects the i_{th} node from the k_{th} layer to the j_{th} node from the $k_{th}+1$ layer

L : cost function

α : learning rate

An effective way of calculating all partial derivatives in a neural network in a linear time – regardless the size of the network – is known as *backpropagation* (short for backward propagation of errors). It's an implementation of the chain rule of derivatives that aims at calculating the gradient of the cost function w.r.t the network's weights. The following is the chain rule which computes the derivative of composite functions:

$$\frac{d}{dx}[f(g(x))] = \acute{f}\big(g(x)\big).\acute{g}(x) \tag{2.22}$$

For example, to update the weight $w_{11}^{(2)}$in the network of Figure 2.8 the following update rule is used:

$$w_{11}^{(2)} = w_{11}^{(2)} - \alpha\frac{\partial L}{\partial\, w_{11}^{(2)}} \tag{2.23}$$

Where c, is the cost function and $\frac{\partial L}{\partial\, w_{11}^{(2)}}$ is the partial derivative of the cost function w.r.t the weight $w_{11}^{(2)}$. Knowing that c is a function of the prediction $o_{1(out)}$ or $\hat{y}_1$ (the quantity that's relevant to $w_{11}^{(2)}$), and that $h_{1(out)}$ and $o_{1(out)}$ or $\hat{y}_1$ in Figure 2.8 are given by equations 2.1, 2.2, 2.5 and 2.6, how to calculate $\frac{\partial L}{\partial\, w_{11}^{(2)}}$? Here comes the role of backpropagation. Using the chain rule, $\frac{\partial L}{\partial\, w_{11}^{(2)}}$ is obtained as follows:

$$\frac{\partial L}{\partial\, w_{11}^{(2)}} = \frac{\partial L}{\partial o_{1(out)}}\ \frac{\partial o_{1(out)}}{\partial o_{1(in)}}\ \frac{\partial o_{1(in)}}{\partial w_{11}^{(2)}} \tag{2.24}$$

By finding the quantity $\frac{\partial L}{\partial\, w_{11}^{(2)}}$ which quantifies how much a change in $w_{11}^{(2)}$ would affect the cost function c, the next step is to update $w_{11}^{(2)}$ accordingly as shown in equation (2.21). Below is a concrete example with real numbers to show how a neural network operates (Mazur, 2017). The notation used for this example as shown in Figure 2.15 is different than that used in Figure 2.8 to aid in illustration. All numbers have been rounded.

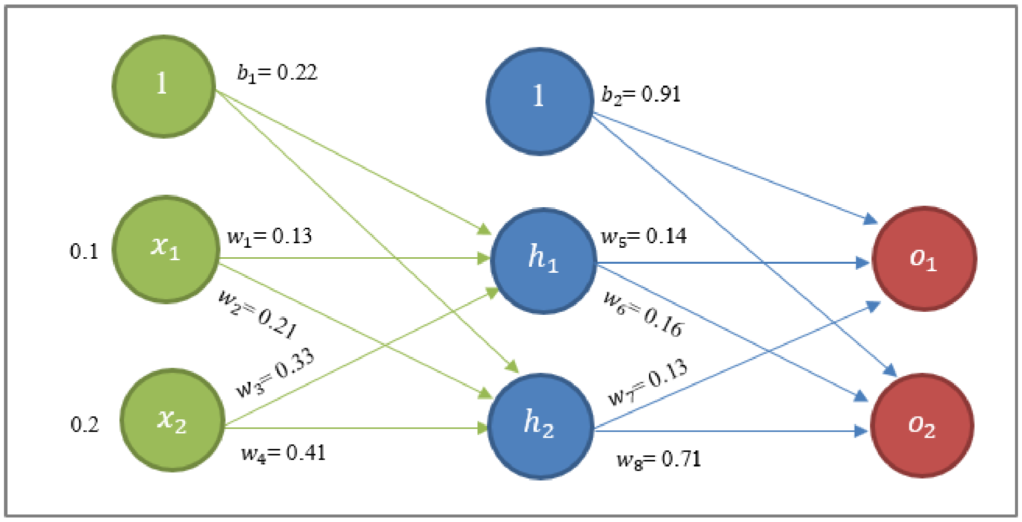


Figure 2.15: : A neural network with real numbers.

The given neural network has the following information:

Input features: x_1= 0.1 and x_2= 0.2

Targets: o_1= 0.5 and o_2= 0.3

Activation function for hidden layer: σ = Sigmoid **-** given in equation 2.10.

Activation function for output layer: σ = Sigmoid

Cost function: mean squared error **(MSE) -** given in equation 2.17.

Learning rate: α = 0.1

The following are the steps taken by a network during training:

1) Initialize weights and biases randomly.

2) Forward pass: traverse the nodes from left to right (from the input layer up to the output layer) and calculate the following quantities:

   From the input layer to the hidden layer:

   $h_{1(in)} = w_1 * x_1 + w_3 * x_2 + b_1$ = 0.13*(0.1) + 0.33*(0.2) + 0.22 = 0.299

   $h_{1(out)} = \sigma(0.299) = \frac{1}{1+e^{-0.299}} = 0.426$

   $h_{2(in)} = w_2 * x_1 + w_4 * x_2 + b_1$ = 0.21*(0.1) + 0.41*(0.2) + 0.22 = 0.323

   $h_{2(out)} = \sigma(0.323) = 0.580$

   From the hidden layer to the output layer:

   $o_{1(in)} = w_5 * h_{1(out)} + w_7 * h_{2(out)} + b_2$ = 0.14*(0.429) + 0.13*(0.580) + 0.91 = 1.680

   $o_{1(out)} = \sigma(1.680) = 0.843$

   $o_{2(in)} = w_6 * h_{1(out)} + w_8 * h_{2(out)} + b_2$ = 0.16*(0.429) + 0.71*(0.580) + 0.91 = 1.390

   $o_{2(out)} = \sigma(1.390) = 0.801$

3) Calculate the error (cost) of the network using the cost function as follows:

   The error (cost) of the first output:

   $L_{o1} = \frac{1}{2}\left(y_1 - o_{1(out)}\right)^2 = \frac{1}{2}(0.5 - 0.843)^2 = 0.059$

   The error (cost) of the second output:

   $L_{o2} = \frac{1}{2}\left(y_2 - o_{2(out)}\right)^2 = \frac{1}{2}(0.3 - 0.801)^2 = 0.126$

   The total error:

   $L = L_{o1} + L_{o2}$ = 0.059 + 0.126 = 0.185

4) Backward pass: traverse the nodes from right to left (from output layer up to the input layer) to adjust the weights. This involves calculating the partial derivative of the cost function w.r.t each weight. The partial derivative quantifies how much a change in the weight affects the total error so that the weight is adjusted accordingly later. This is done through backpropagation which is an application of the chain rule in calculus. Backpropagation involves finding the following quantities:

a) From the output to the hidden layer:

$\frac{\partial L}{\partial w_5}$, $\frac{\partial L}{\partial w_6}$, $\frac{\partial L}{\partial w_7}$,$\frac{\partial L}{\partial w_8}$ and $\frac{\partial L}{\partial b_2}$.

- $\boldsymbol{\frac{\partial L}{\partial w_5}}$:

$$\frac{\partial L}{\partial w_5} = \frac{\partial L}{\partial o_{1(out)}} * \frac{\partial o_{1(out)}}{\partial o_{1(in)}} * \frac{\partial o_{1(in)}}{\partial w_5}$$

$$L = \frac{1}{2}\left(y_1 - o_{1(out)}\right)^2 + \frac{1}{2}\left(y_2 - o_{2(out)}\right)^2$$

$$\frac{\partial L}{\partial o_{1(out)}} = 2 * \frac{1}{2}\left(y_1 - o_{1(out)}\right)^{2-1} * (0-1) + 0 = -\left(y_1 - o_{1(out)}\right)$$

$$= o_{1(out)} - y_1 = 0.843 - 0.5 = 0.343$$

Knowing that: $o_{1(out)} = \frac{1}{1+e^{-o_{1(in)}}}$, it follows that:

$$\frac{\partial o_{1(out)}}{\partial o_{1(in)}} = \frac{1}{1+e^{-o_{1(in)}}} * \left(1 - \frac{1}{1+e^{-o_{1(in)}}}\right) = o_{1(out)} * (1 - o_{1(out)})$$

$$= 0.843 * (1 - 0.843) = 0.132$$

Knowing that: $o_{1(in)} = w_5 * h_{1(out)} + w_7 * h_{2(out)} + b_2$, it follows that:

$$\frac{\partial o_{1(in)}}{\partial w_5} = 1 * h_{1(out)} + 0 = h_{1(out)} = 0.426$$

$$\frac{\partial L}{\partial w_5} = 0.343 * 0.132 * 0.426 = 0.019$$

- $\boldsymbol{\frac{\partial L}{\partial w_6}}$:

$$\frac{\partial L}{\partial w_6} = \frac{\partial L}{\partial o_{2(out)}} * \frac{\partial o_{2(out)}}{\partial o_{2(in)}} * \frac{\partial o_{2(in)}}{\partial w_6}$$

$$\frac{\partial L}{\partial o_{2(out)}} = o_{2(out)} - y_2 = 0.801 - 0.3 = 0.501$$

$$\frac{\partial o_{2(out)}}{\partial o_{2(in)}} = o_{2(out)} * \left(1 - o_{2(out)}\right) = 0.801 * (1 - 0.801) = 0.159$$

Knowing that: $o_{2(in)} = w_6 * h_{1(out)} + w_8 * h_{2(out)} + b_2$, it follows that:

$$\frac{\partial o_{2(in)}}{\partial w_6} = h_{1(out)} = 0.426$$

$$\frac{\partial L}{\partial w_6} = 0.501 * 0.159 * 0.426 = 0.0339$$

- $\boldsymbol{\frac{\partial L}{\partial w_7}}$:

$$\frac{\partial L}{\partial w_7} = 0.0263$$

- $\boldsymbol{\frac{\partial L}{\partial w_8}}$:

$$\frac{\partial L}{\partial w_8} = 0.0462$$

$$\frac{\partial L}{\partial b_2} = \frac{\partial L_{o1}}{\partial b_2} + \frac{\partial L_{o2}}{\partial b_2}$$

$$\frac{\partial L_{o1}}{\partial b_2} = \frac{\partial L_{o1}}{\partial o_{1(out)}} * \frac{\partial o_{1(out)}}{\partial o_{1(in)}} * \frac{\partial o_{1(in)}}{\partial b_2} = 0.343 * 0.132 * 1 = 0.0453$$

$$\frac{\partial L_{o2}}{\partial b_2} = 0.501 * 0.159 * 1 = 0.0797$$

$$\frac{\partial c}{\partial b_2} = 0.0453 + 0.0797 = 0.125$$

b) From hidden layer to input layer: $\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \frac{\partial L}{\partial w_3}, \frac{\partial L}{\partial w_4}$ and $\frac{\partial L}{\partial b_1}$

- $\boldsymbol{\frac{\partial L}{\partial w_1}}$:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial h_{1(out)}} * \frac{\partial h_{1(out)}}{\partial h_{1(in)}} * \frac{\partial h_{1(in)}}{\partial w_1}$$

Each of the three quantities comprising $\frac{\partial L}{\partial w_1}$ will be found separately:

- **Quantity (1)** $\boldsymbol{\frac{\partial L}{\partial h_{1(out)}}}$**:**

$$\frac{\partial L}{\partial h_{1(out)}} = \frac{\partial L_{o1}}{\partial h_{1(out)}} + \frac{\partial L_{o2}}{\partial h_{1(out)}}$$

$\boldsymbol{\frac{\partial c_1}{\partial h_{1(out)}}}$ **:**

$$\frac{\partial L_{o1}}{\partial h_{1(out)}} = \frac{\partial L_{o1}}{\partial o_{1(out)}} * \frac{\partial o_{1(out)}}{\partial o_{1(in)}} * \frac{\partial o_{1(in)}}{\partial h_{1(out)}}$$

$$\frac{\partial L_{o1}}{\partial o_{1(out)}} * \frac{\partial o_{1(out)}}{\partial o_{1(in)}} = 0.343 * 0.132 = 0.0453$$

$$\frac{\partial o_{1(in)}}{\partial h_{1(out)}} = w_5 = 0.14$$

$$\frac{\partial L_{o1}}{\partial h_{1(out)}} = 0.0453 * 0.14 = 0.006$$

$\boldsymbol{\frac{\partial L_{o2}}{\partial h_{1(out)}}}$ **:**

$$\frac{\partial L_{o2}}{\partial h_{1(out)}} = 0.501 * 0.159 * 0.16 = 0.012$$

$\boldsymbol{\frac{\partial L}{\partial h_{1(out)}}}$ **:**

$$\frac{\partial L}{\partial\, h_{1(out)}} = 0.006 + 0.012 = 0.018$$

- **Quantity (2)** $\frac{\boldsymbol{\partial h_{1(out)}}}{\boldsymbol{\partial\, h_{1(in)}}}$:

As shown previously: $h_{1(out)} = \frac{1}{1+e^{h_{1(in)}}}$

$$\frac{\partial h_{1(out)}}{\partial\, h_{1(in)}} = \frac{1}{1+e^{h_{1(in)}}} * \left(1 - \frac{1}{1+e^{h_{1(in)}}}\right) = h_{1(out)} * \left(1 - h_{1(out)}\right)$$

$$= 0.426 * (1 - 0.426) = 0.245$$

- **Quantity (3)** $\frac{\boldsymbol{\partial h_{1(in)}}}{\boldsymbol{\partial\, w_1}}$:

$$\frac{\partial h_{1(in)}}{\partial\, w_1} = x_1 = 0.1$$

- **Now we can find** $\frac{\boldsymbol{\partial L}}{\boldsymbol{\partial\, w_1}}$:

$$\frac{\partial L}{\partial\, w_1} = 0.018 * 0.245 * 0.1 = 0.000441$$

Other quantities can be found following the same approach:

$$\frac{\partial L}{\partial\, w_2} = 0.063 * 0.244 * 0.1 = 0.0015372$$

$$\frac{\partial L}{\partial\, w_3} = 0.018 * 0.245 * 0.2 = 0.000882$$

$$\frac{\partial L}{\partial\, w_4} = 0.063 * 0.244 * 0.2 = 0.0030744$$

$$\frac{\partial L}{\partial\, b_1} = \frac{\partial L_{o1}}{\partial\, b_1} + \frac{\partial L_{o2}}{\partial\, b_1}$$

$$\frac{\partial L_{o1}}{\partial\, b_1} = \frac{\partial L_{o1}}{\partial\, o_{1(out)}} * \frac{\partial o_{1(out)}}{\partial\, o_{1(in)}} * \frac{\partial o_{1(in)}}{\partial\, h_{1(out)}} * \frac{\partial\, h_{1(out)}}{\partial\, h_{1(in)}} * \frac{\partial\, h_{1(in)}}{\partial\, b_1}$$

$$= 0.343 * 0.132 * 0.14 * 0.245 * 1 = 0.00156$$

$$\frac{\partial L_{o2}}{\partial\, b_1} = \frac{\partial L_{o2}}{\partial\, o_{2(out)}} * \frac{\partial o_{2(out)}}{\partial\, o_{2(in)}} * \frac{\partial o_{2(in)}}{\partial\, h_{2(out)}} * \frac{\partial\, h_{2(out)}}{\partial\ \ h_{2(in)}} * \frac{\partial\, h_{2(in)}}{\partial\ \ b_1}$$

$$= 0.501 * 0.159 * 0.71 * 0.2436 * 1 = 0.0138$$

$$\frac{\partial L}{\partial\, b_1} = 0.0016 + 0.0138 = 0.0154$$

5) Update the weights using the formula: $w = \grave{w} - \alpha \frac{\partial L}{\partial\, w}$ as shown previously in equation 2.19.

$$w_1 = 0.13 - 0.1 * 0.000441 \quad = 0.130$$

$$w_2 = 0.21 - 0.1 * 0.0015372 \quad = 0.210$$

$$w_3 = 0.33 - 0.1 * 0.000882 \quad = 0.330$$

$$w_4 = 0.41 - 0.1 * 0.0030744 \quad = 0.410$$

$$w_5 = 0.14 - 0.1 * 0.019 \quad = 0.1381$$

$$w_6 = 0.16 - 0.1 * 0.0339 \quad = 0.1566$$

$$w_7 = 0.13 - 0.1 * 0.0263 \quad = 0.1274$$

$$w_8 = 0.71 - 0.1 * 0.0462 \quad = 0.7054$$

$$b_1 = 0.22 - 0.1 * 0.01533 \quad = 0.2185$$

$$b_2 = 0.91 - 0.1 * 0.125 \quad = 0.8975$$

2.4.3 **Vectorization in Neural Networks**

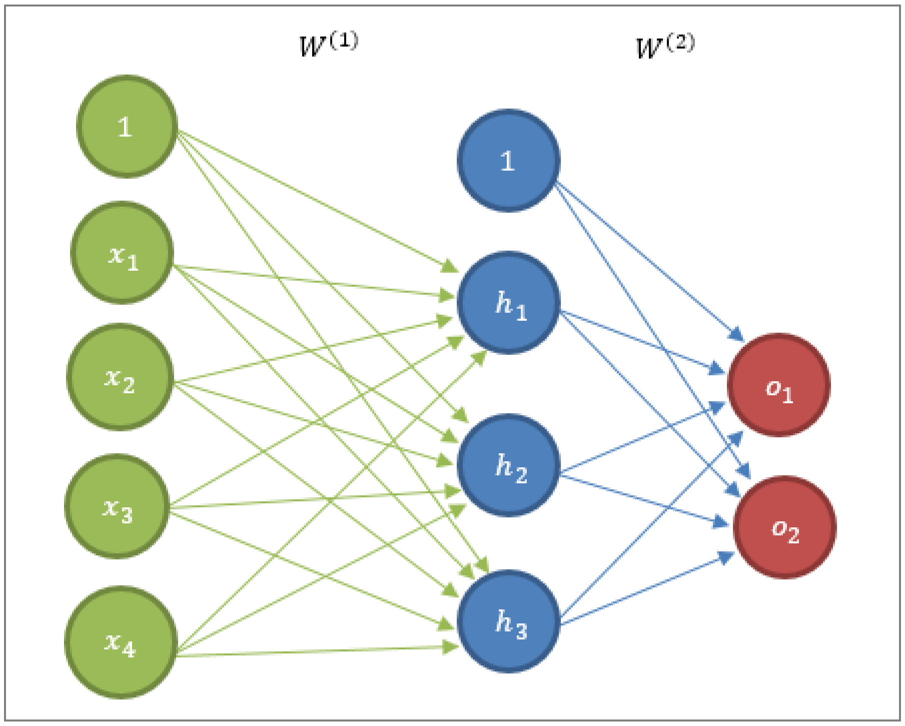


Figure 2.16: A simple neural network

Vectorization means to implement an algorithm so that a vector of values is processed at once rather than processing a single value at a time. In its essence, vectorization is the process of eliminating for loops and replacing them with matrix operations. In practice, vectorization is adopted when implementing neural networks since it allows the use of matrix operations that exploit parallelization capabilities of modern CPUs and GPUs resulting in a much faster implementation than the case of using for loops. A simple neural network like previously shown networks is given in Figure 2.16 and will be used to illustrate vectorization concept in neural networks. For one training example, the vectorized forward propagation step is as follows:

- From the input layer to the hidden layer:

$$h_{(out)} = \sigma\left(\begin{bmatrix} w_{11}^{(1)} & w_{21}^{(1)} & w_{31}^{(1)} & w_{41}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} & w_{32}^{(1)} & w_{42}^{(1)} \\ w_{13}^{(1)} & w_{23}^{(1)} & w_{33}^{(1)} & w_{43}^{(1)} \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_1 \\ b_1 \\ b_1 \end{bmatrix}\right) = \sigma\left(\begin{bmatrix} h_{1(in)} \\ h_{2(in)} \\ h_{3(in)} \end{bmatrix}\right) = \begin{bmatrix} h_{1(out)} \\ h_{2(out)} \\ h_{3(out)} \end{bmatrix}$$

Which can be represented as follows:

$$h_{(in)} = W^{(1)} * X + b_1 \tag{2.23}$$

$$h_{(out)} = \sigma(h_{(in)}) \tag{2.24}$$

Where:

$W^{(1)} \in \mathbb{R}^{h\times n}$	: weight matrix from the input to the hidden layer
$X \in \mathbb{R}^{n\times 1}$	: input features vector
$b_1 \in \mathbb{R}^{n\times 1}$	: bias vector from the input layer to the hidden layer
$h_{(in)} \in \mathbb{R}^{h\times 1}$	: input vector to the hidden unit
$h_{(out)} \in \mathbb{R}^{h\times 1}$	: output vector of the hidden unit
σ	: activation function
n	: number of input features
h	: number of hidden units

- From the hidden layer to the output layer:

$$o_{(out)} = \sigma\left(\begin{bmatrix} w_{11}^{(2)} & w_{21}^{(2)} & w_{31}^{(2)} \\ w_{12}^{(2)} & w_{22}^{(2)} & w_{32}^{(2)} \end{bmatrix} * \begin{bmatrix} h_{1(out)} \\ h_{2(out)} \\ h_{3(out)} \end{bmatrix} + \begin{bmatrix} b_2 \\ b_2 \\ b_2 \end{bmatrix} \right) = \sigma\left(\begin{bmatrix} o_{1(in)} \\ o_{2(in)} \end{bmatrix} \right) = \begin{bmatrix} o_{1(out)} \\ o_{2(out)} \end{bmatrix}$$

Which can be represented as follows:

$$o_{(in)} = W^{(2)} * h_{(out)} + b_2 \tag{2.25}$$

$$o_{(out)} = \sigma(o_{(in)}) \tag{2.26}$$

Where:

$W^{(2)} \in \mathbb{R}^{q \times h}$	: weight matrix from the hidden to the output layer
$h_{(out)} \in \mathbb{R}^{h \times 1}$	: output vector of the hidden unit
$b_2\, \mathbb{R}^{h \times 1}$	: bias vector from the hidden layer to the output layer
$o_{(in)} \in \mathbb{R}^{q \times 1}$	: input vector to the output unit
$o_{(out)} \in \mathbb{R}^{q \times 1}$	: output vector of the output unit
σ	: activation function
n	: number of input features
h	: number of hidden units
q	: number of output units

Chapter 3

RECURRENT NEURAL NETWORKS

3.1 **Introduction**

A human reader processes a sentence word by word while keeping memories – internal model or state – of what has been seen so far. The internal state is built from past information and is constantly updated as new information is being processed. This incremental processing of a sequence – a sentence in this case – results in a robust representation of the meaning conveyed by the sentence (Chollet, 2017a). Similarly, Recurrent neural networks (RNNs) use the same strategy, though in a much simpler fashion. They process sequences element by element through an internal loop while keeping a state of what has been processed so far as shown in Figure 3.1.

Feed-forward neural networks (FNNs) make a blind assumption about the data to be processed, i.e. FNNs assume that the data is independently and identically distributed. However, this assumption doesn't hold in most cases. This text is an example of a sequential data in which words are written in some order. When these words are permuted randomly, it would be impossible to interpret the meaning of the text. The audio signal of a speech, image frames in a video and stock prices are some examples of sequential data that FNNs fail to deal with. Moreover, the task at hand may not only take a sequence as input but rather may require completing the sequence such as predicting the stock market (Zhang et al., 2019). To overcome FNNs shortcomings, RNNs were designed with the aim of handling sequential data. RNNs achieved promising results in many tasks specifically those with variable input and output lengths such as machine translation, image captioning and handwriting recognition (Lipton, 2015; Chung et al., 2014). Specifically, RNNs are a rich class of neural networks specialized for processing sequential data that is often of variable lengths. They can process much longer sequences than other neural network architectures (Goodfellow et al., 2016).

According to (Graves, 2013), RNNs are dynamic models that can generate sequences of music, text and motion capture data. Moreover, RNNs have shown good performance is motion prediction (Martinez et al., 2017).

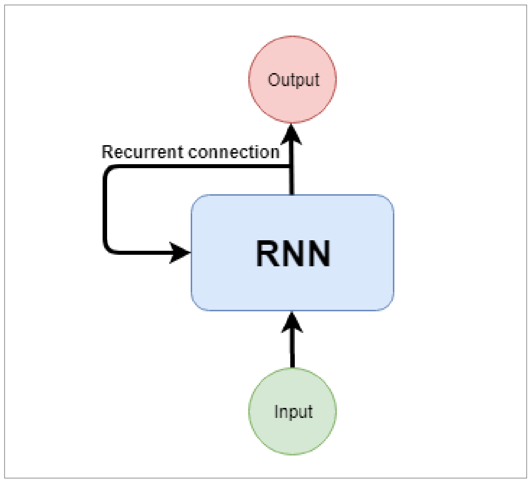


Figure 3.1: A recurrent network with a loop. Figure adapted from (Chollet, 2017a).

RNN extends the standard FNN to handle variable length sequences by using a recurrent hidden state. At any timestep, the activation of the hidden state depends on the activation of hidden state at the previous timestep (Chung et al., 2014).

RNNs operate on a sequence of values $x^{(1)}, \ldots, x^{(T)}$. This sequence is a vector $x^{(t)}$ where t is the timestep index ranging from 1 to T. The timestep index does not necessarily represent the time as in the real world but rather it may represent merely a position in the sequence. RNNs employ the principle of parameter sharing. As the name indicates, parameter sharing means sharing parameters or weights across several timesteps. Equation 3.1 describes an RNN generally (Goodfellow et al., 2016).

$$a^{(t)} = f\left(a^{(t-1)}, x^{(t)}; W\right) \tag{3.1}$$

Where:

$a^{(t)}$: state of the network at time step t (current activation)

$a^{(t-1)}$: state of the network at the previous time step $t-1$ (old activation)

$x^{(t)}$: input at the current time step t

W : set of weights or parameters

f : a non-linear activation function that can be as simple as Sigmoid or as complex as long short-term memory unit (LSTM) which will be discovered later in this chapter (Cho K. , et al., 2014b).

Clearly as indicated by equation 3.1, the state $a^{(t)}$ (current activation) encodes information about the complete past sequence because it's calculated using the state of the previous time step $a^{(t-1)}$ (previous activation) which in turn encodes information about the sequence from earlier time steps. Additionally, equation 3.1 is recursive (recurrent) as $a^{(t)}$ points back to $a^{(t-1)}$. The principle of parameter sharing means that the same set of parameters W is used across several time steps. An illustration of the equation is shown in Figure 3.2 where the network is depicted from two perspectives. To the left of the figure is the circuit diagram of an RNN that uses an input x to calculate the state a which is passed forward through time. The black box represents a single time step. To the right of the figure is the unfolded version of the same RNN where each node represents an instance of a single time step. Typically, RNNs would have an output layer to use the state a to make predictions (Goodfellow et al., 2016). Figure 3.2 represents an RNN universally without specifying neither the output of the network nor the choice of activation function for hidden units.

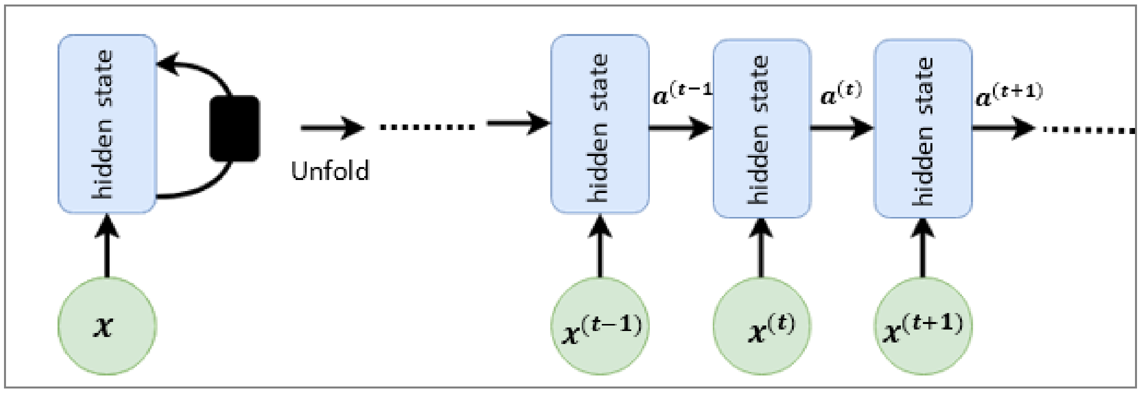


Figure 3.2: RNN with no outputs. (left) a circuit diagram of the RNN. (right) the same RNN but unfolded.
Figure adapted fron Goodfellow et al., (2016).

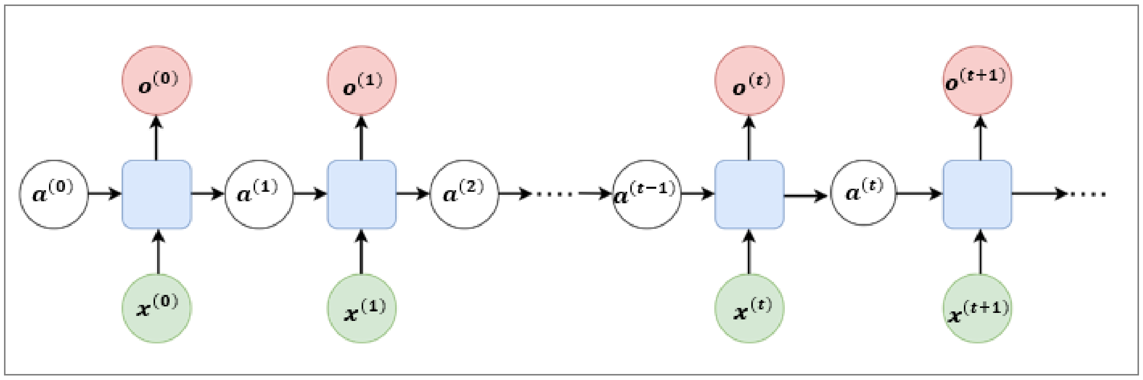


Figure 3.3: Traditional RNN.
Figure adapted from (Olah, Understanding LSTM Networks, 2015).

Figure 3.3 gives more detailed representation of a traditional RNN (vanilla RNN) where $a^{(t)}$ is the activation at time step *t*, $x^{(t)}$ is the input at time step *t* and $o^{(t)}$ is the output at time step *t*. $a^{(t)}$ and $o^{(t)}$ are calculated as described by equations 3.2 and 3.3 respectively (Amidi & Amidi).

$$a^{(t)} = g_1\left(W_a a^{(t-1)} + W_x x^{(t)} + b_a\right) \tag{3.2}$$

$$o^{(t)} = g_2\left(W_y a^{(t)} + b_y\right) \tag{3.3}$$

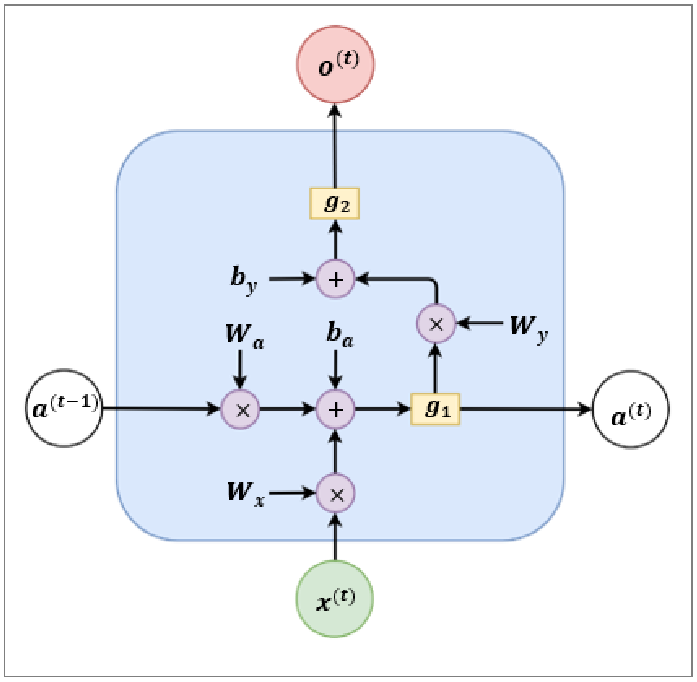


Figure 3.4: Vanilla RNN block.

Where:

b	: number of examples in the mini-batch $x^{(t)}$ (batch size)
n	: number of features
h	: number of hidden units
q	: number of output units
$x^{(t)} \in \mathbb{R}^{b \times n}$	: mini-batch input at current time step *t*
$a^{(t)} \in \mathbb{R}^{b \times h}$	: hidden state (activation) at current time step *t*
$a^{(t-1)} \in \mathbb{R}^{b \times h}$	: hidden state (activation) at previous time step *t-1*
$o^{(t)} \in \mathbb{R}^{b \times q}$	: output at time step *t*

$W_a \in \mathbb{R}^{h \times h}$, $W_x \in \mathbb{R}^{n \times h}$ and $W_y \in \mathbb{R}^{h \times q}$: weight matrices

$b_a \in \mathbb{R}^{1 \times h}$ and $b_y \in \mathbb{R}^{1 \times q}$: biases

g_1 and g_2 : activation functions

3.2 **RNN Activation Functions**

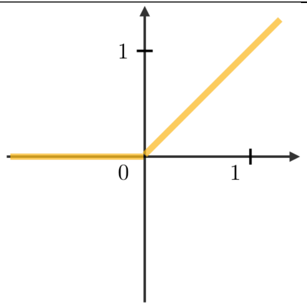
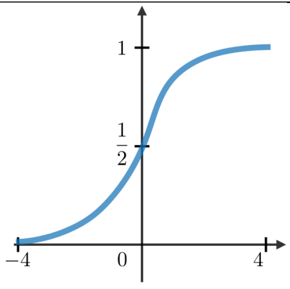
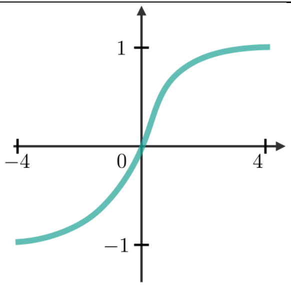
RELU	**Sigmoid**	**Tanh**
$\sigma(x) = \max(0, x)$	$\sigma(x) = \dfrac{1}{1 + e^{-x}}$	$\sigma(x) = \dfrac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$
		

Table 1: Common choices of activation functions used in RNNs.
(Amidi & Amidi).

Common choices of activation functions are shown in Table 1. Figures 3.2 and 3.3 depict RNN units as a black box. The calculations carried out by an RNN unit as described by equations 3.2 and 3.3 are illustrated visually in Figure 3.4.

3.3 **RNN Loss Function**

Generally, the loss function is a measure of how close a neural network is to the ideal weights. RNN loss function is defined as the sum of losses over all time steps as shown in the following equation:

$$L_{total}(\hat{y}, y) = \sum_{t=1}^{T_y} L_t\left(\hat{y}^{(t)}, y^t\right) \tag{3.4}$$

3.4 **RNNs in Action**

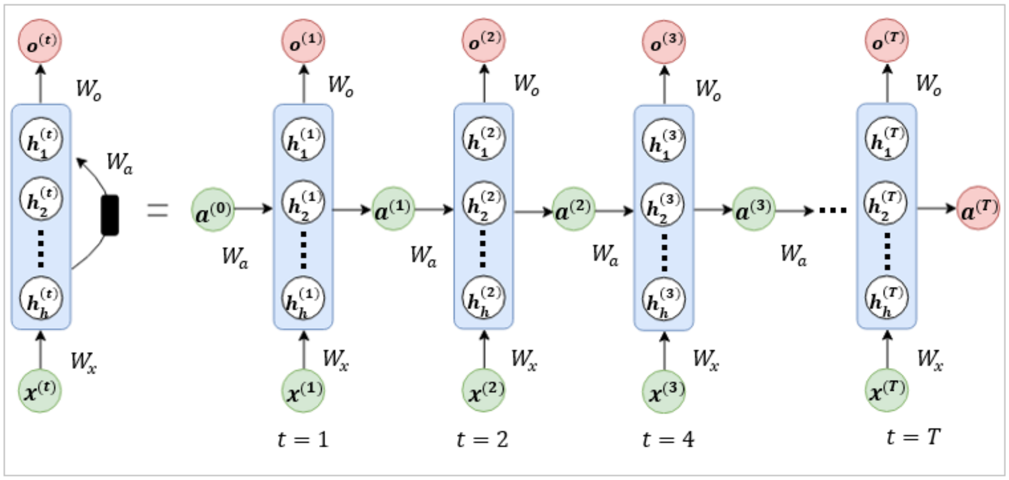


Figure 3.5: Rolled RNN diagram vs unrolled RNN diagram.
(left): rolled version or circuit diagram of a vanilla RNN. (right): unrolled version or unfolded diagram of the same vanilla RNN. Figure adapted from (Chen G. , 2016).

This section shows a simple example of an RNN in action including one forward pass followed by a backward pass (Khuong, 2019; Chen, 2016). As shown previously, RNNs can be viewed either as a circuit diagram or as an unfolded diagram as shown below.
Figure 3.5 shows a simple RNN, also known as vanilla RNN where the blue blocks are the *hidden states* which can be thought of as an activation function that acts on each circle

inside the block that is the *hidden node*. $h_i^{(t)}$ is the i_{th} hidden node at timestep t. Each hidden node performs a linear calculation then an activation function g_1 is applied to all nodes to get a vector of activations $a^{(t)}$. At each timestep t, the RNN takes the output of the hidden state at the previous timestep $t-1$ along with the input vector $x^{(t)}$ at the current timestep t. This allows the RNN to accumulate information about the whole sequence and thus keep a memory of what it has seen before. However, at timestep 0, there is no previous hidden state, therefore $a^{(0)}$ would be a vector of 0s. As pointed out earlier, RNNs use the concept of parameter sharing meaning that all weight matrices are shared across several timesteps. i.e. W_a, W_x and W_o are the same throughout the whole sequence. The following equations describe Figure 3.5 mathematically:

$$a^{(t)} = g_1(W_a\, a^{(t-1)} + W_x x^{(t)}) \tag{3.5}$$

$$o^{(t)} = g_2(W_o a^{(t)}) \tag{3.6}$$

Where:

n	: number of input features (length of input vector $x^{(t)}$)
h	: number of hidden nodes (length of activation vector $a^{(t)}$)
q	: number of output nodes (length of output vector $\hat{y}^{(t)}$)
$a^{(t)} \in \mathbb{R}^h$	: vector of activation values at the current timestep $t-1$ (the output of the hidden state at the current timestep)
$a^{(t-1)} \in \mathbb{R}^h$	: vector of activation values at the previous timestep $t-1$ (the output of the hidden state at the previous timestep)
$x^{(t)} \in \mathbb{R}^n$	: vector of input features at the current timestep t
$o^{(t)} \in \mathbb{R}^q$	: output vector (prediction) at timestep t
$a^{(t)} \in \mathbb{R}^h$	: vector of activation values at the current timestep t (the output of the hidden state of the current timestep).

$W_a \in \mathbb{R}^{h \times h}$	: weight matrix multiplied by $a^{(t-1)}$ to get $a^{(t)}$
$W_x \in \mathbb{R}^{h \times n}$	: weight matrix multiplied by $x^{(t)}$ to get $a^{(t)}$
$W_o \in \mathbb{R}^{q \times h}$	: weight matrix multiplied by $a^{(t)}$ to get $\hat{y}^{(t)}$
g_1	: activation function of hidden nodes
g_2	: activation function of output nodes

N	**i**	**c**	**e**
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

Table 2: One-hot encoding for categorical variables.

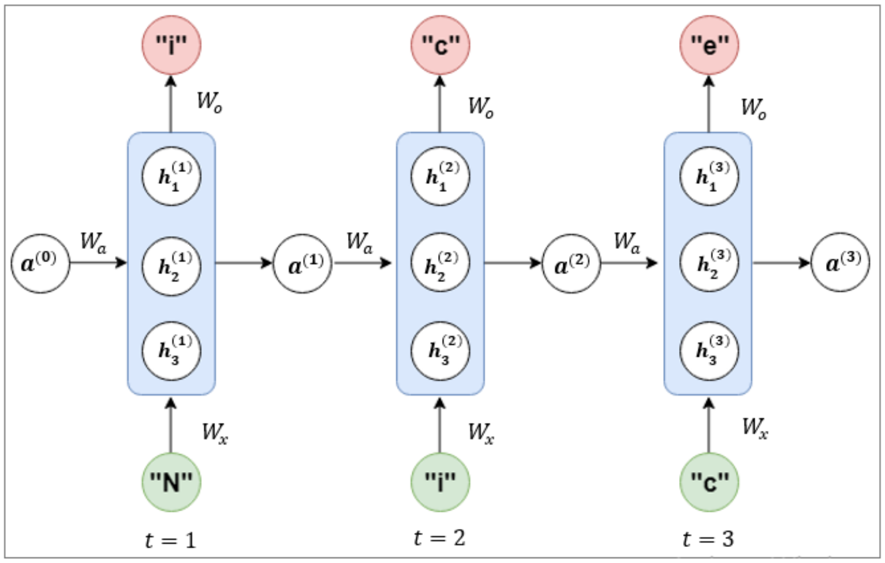


Figure 3.6: A simple RNN that is to be trained to predict the last letter "e" from the word "Nice" given the previous three letters.

Below is given a superficial example of an RNN. The goal is to train the RNN to predict the last letter of the word “Nice” given the previous three letters. As shown in Figure 3.6, The given RNN has 3 hidden nodes. It’s a multi-classification problem since there are only 4 letters to predict “N”, “i”, “c” and “e”. Of course, Neural networks can deal with numerical values only and hence each letter is represented as a one-hot encoded vector. This means that each letter is assigned a vector of length that equals the number of classes, 4 in this case where one element of that vector is 1 and the rest are 0s. One-hot encoding is used to convert categorical variables to a form that can be passed to neural networks. Categorical variables are those that represent names or labels and don’t have a specific ordering or ranking associated with them. Table 2 illustrates how each letter is represented with one-hot encoding.

Generally, the training phase of the given RNN would look like the following:

- Initialize weight matrices W_a, W_x and W_o randomly.
- Make a forward pass to find the predictions.
- Calculate the loss for each timestep and then the cost for all timesteps.
- Make a backward pass to find the gradients.
- Update the weights according to the gradients.
- Repeat the steps until convergence or for a defined number of iterations.

One forward pass could be described mathematically as follows:

$$a_t = g_1\left(\begin{bmatrix} w_{a,11} & w_{a,12} & w_{a,13} \\ w_{a,21} & w_{a,22} & w_{a,23} \\ w_{a,31} & w_{a,32} & w_{a,33} \end{bmatrix}\begin{bmatrix} a_1^{(t-1)} \\ a_2^{(t-1)} \\ a_3^{(t-1)} \end{bmatrix} + \begin{bmatrix} w_{x,11} & w_{x,12} & w_{x,13} & w_{x,14} \\ w_{x,21} & w_{x,22} & w_{x,23} & w_{x,24} \\ w_{x,31} & w_{x,32} & w_{x,32} & w_{x,34} \end{bmatrix}\begin{bmatrix} x_1^{(t)} \\ x_2^{(t)} \\ x_3^{(t)} \\ x_4^{(t)} \end{bmatrix}\right)$$

$$= \begin{bmatrix} a_1^{(t)} \\ a_2^{(t)} \\ a_3^{(t)} \end{bmatrix} \qquad (3.7)$$

$$o_t = g_2\left(\begin{bmatrix} w_{o,11} & w_{o,12} & w_{o,13} \\ w_{o,21} & w_{o,22} & w_{o,23} \\ w_{o,31} & w_{o,32} & w_{o,33} \\ w_{o,41} & w_{o,42} & w_{o,43} \end{bmatrix}\begin{bmatrix} a_1^{(t)} \\ a_2^{(t)} \\ a_3^{(t)} \end{bmatrix}\right) = \begin{bmatrix} o_1^{(t)} \\ o_2^{(t)} \\ o_3^{(t)} \\ o_4^{(t)} \end{bmatrix} \tag{3.8}$$

After the forward pass, the loss for each timestep is computed followed by the cost which is the sum of all losses. Since the task at hand is a multiclassification problem, the used loss would be the cross-entropy. The cross-entropy loss at timestep *t* is as follows:

$$L^{(t)}\left(y^{(t)}, o^{(t)}\right) = -\, y^{(t)} log(o^{(t)}) \tag{3.9}$$

Whereas the cross-entropy cost of all timesteps is as follows:

$$L(y, o) = -\, \sum_{t=1}^{t=T} y^{(t)} log(o^{(t)}) \tag{3.10}$$

Now it's the time of the backward pass which aims at calculating the gradients using backpropagation through time BPTT. BPTT is an application of backpropagation that is used in RNNs (Zhang et al., 2019). It's trickier than usual backpropagation since the sequence might be too long and hence the dependencies of weights span over the whole sequence. This is true due to the principle of parameter sharing where the same weight matrices are used over several timesteps. When calculating the gradients of the weights w.r.t the cost function, the weight dependencies are the same length as the sequence itself which makes it a tricky and computationally intensive task, especially for long sequences. The gradients of W_a, W_x and W_o w.r.t to the cost function are as follows:

1. $\frac{\partial L}{\partial\, w_o}$:

Knowing that:

$L_t(y_t, o_t) = -\, y_t log(o_t)$

$o^{(t)} = \sigma_2(W_o a^{(t)})$

It follows that:

$$\frac{\partial L^{(t)}}{\partial\, w_o} = \frac{\partial L^{(t)}}{\partial o^{(t)}}\frac{\partial o^{(t)}}{\partial\, w_o}$$

$$\frac{\partial L}{\partial\, w_o} = \frac{\partial L^{(1)}}{\partial\, w_o} + \frac{\partial L^{(2)}}{\partial\, w_o} + \frac{\partial L^{(3)}}{\partial\, w_o} = \sum_{t=1}^{t=T} \frac{\partial L^{(t)}}{\partial\, w_o}$$

2. $\frac{\partial L}{\partial\, w_a}$ **:**

Knowing that:

$$L^{(t)}\left(y^{(t)}, o^{(t)}\right) = -\, y^{(t)} log(o^{(t)})$$

$$o^{(t)} = \sigma_2(W_o a^{(t)})$$

$$a^{(t)} = \sigma_1(W_a\, a^{(t-1)} + W_x x^{(t)})$$

It follows that:

$$\frac{\partial L^{(t)}}{\partial\, w_o} = \frac{\partial L^{(t)}}{\partial o^{(t)}} \frac{\partial o^{(t)}}{\partial\, a^{(t)}} \frac{\partial a^{(t)}}{\partial\, W_a}$$

But $a^{(t-1)}$ also contains W_a and hence the chain rule is applied to $a^{(t-1)}$ recursively until reaching $a^{(0)}$

$$\frac{\partial L^{(t)}}{\partial\, w_a} = \frac{\partial L^{(t)}}{\partial o^{(t)}} \frac{\partial o^{(t)}}{\partial\, a^{(t)}} \frac{\partial a^{(t)}}{\partial\, W_a} + \frac{\partial L^{(t)}}{\partial o^{(t)}} \frac{\partial o^{(t)}}{\partial\, a^{(t)}} \frac{\partial a^{(t)}}{\partial\, a^{(t-1)}} \frac{\partial a^{(t-1)}}{\partial\ \, W_a} +..$$

$$+ \frac{\partial L^{(t)}}{\partial o^{(t)}} \frac{\partial o^{(t)}}{\partial\, a^{(t)}} \frac{\partial a^{(t)}}{\partial\, a^{(0)}} \frac{\partial a^{(0)}}{\partial\ \, W_a}$$

$$\frac{\partial L}{\partial\, w_a} = \sum_{t=1}^{t=T} \sum_{k=0}^{k=t} \frac{\partial L^{(t)}}{\partial o^{(t)}} \frac{\partial o^{(t)}}{\partial\, a^{(t)}} \frac{\partial a^{(t)}}{\partial\, a^{(k)}} \frac{\partial a^{(k)}}{\partial\ \, W_a}$$

Similarly, $\frac{\partial L}{\partial\, w_x}$ is:

$$\frac{\partial L}{\partial\, w_x} = \sum_{t=1}^{t=T} \sum_{k=0}^{k=t} \frac{\partial L^{(t)}}{\partial o^{(t)}} \frac{\partial o^{(t)}}{\partial\, a^{(t)}} \frac{\partial a^{(t)}}{\partial\, a^{(k)}} \frac{\partial a^{(k)}}{\partial W_x}$$

Finally, to update weights proportionally to the gradients, the following update rule is applied:

$$W_i = W_i - \alpha \frac{\partial L}{\partial W_i} \tag{3.11}$$

Where:

W_i : weight matrices W_a, W_x and W_o

α : learning rate

3.5 **Vanishing and Exploding Gradients**

Although RNNs have impressive achievements in many areas, almost none of them were achieved by vanilla RNNs. Rather they were achieved by RNNs that use sophisticated recurrent units. It has been reported that training vanilla RNNs to capture long-term dependencies is hard due to the vanishing and exploding gradients problem (Chung et al., 2014). In practice, vanilla RNNs experience the problem of vanishing or exploding gradients which occurs during backpropagation in which gradients become extremely small or extremely large. Both cases affect learning because vanishing gradients (very small gradients) would rather lead to a model that barely if not at all learns during training whereas exploding gradients (very large gradients) would crash the model and produce lots of not-a-numbers (NaNs) (Trask, 2019). The problem of vanishing and exploding gradients is not limited to RNNs, it occurs in a very deep traditional feed-forward neural networks as well (Chollet, 2017a). The problem is rooted in the fact that computing gradients in RNNs involves repetitive matrix multiplication. To understand the effect of repetitive multiplication, imagine what would happen if some weight w is multiplied by itself several times. Depending on the magnitude of w, the product will either explode or vanish (Goodfellow et al., 2016). The question then arises, why doesn't forward propagation - which involves repetitive matrix multiplication - suffer from that problem? Unlike backpropagation, forward propagation uses activation functions forcing the values to stay within a specific range (Trask, 2019). In his book, "Grokking Deep Learning", Trask provides a synthesized example to show the results of a backpropagation loop for Sigmoid and RELU functions when used as activation functions. The example shows how the gradients become extremely small and extremely large for Sigmoid and RELU respectively. For RELU, the reason for the gradient explosion is matrix multiplication whereas for Sigmoid the reason for the gradient vanishing is because the derivative of the Sigmoid is very flat at the tails of the function (Trask, 2019, p. 273).

Generally, this issue was tackled by researchers from two different perspectives. One perspective is concerned with improving the learning algorithm itself such as using gradient clipping technique, a simple solution to the exploding gradient problem which been in use by practitioners (Goodfellow et al., 2016). Gradient clipping will be explored further in section 3.7. The other perspective is concerned with modifying the architecture of the network itself either by designing more sophisticated recurrent units such long short-term memory (LSTM) and gated recurrent unit (GRU), or by using residual connections between layers. LSTM and GRU layers were invented to conquer the vanishing/exploding gradient problem (Chollet, 2017a). Section 3.6 is devoted to LSTM and GRU. The residual connections will be covered in section 3.7.

3.6 **Gated RNNs**

Backpropagation in RNNs involves propagating gradients over many timesteps causing the gradients to either vanish in most cases or explode occasionally but with severe damage to the optimization. Gated RNNs are the most effective sequence models in use today including long short-term memory (LSTM) and gated recurrent unit (GRU) (Goodfellow et al., 2016).

3.6.1 **Long Short-Term Memory - LSTM**

Introduced by Hochreiter and Schmidhuber back in 1997, LSTM aims at modeling long sequences by providing a remedy to the vanishing gradient problem. LSTM has been improved by many others ever since. Essentially, what LSTM does is that it provides a way of saving information for later use (Chollet, 2017a). Both vanilla RNN and LSTM have a hidden state which is denoted by $a^{(t)}$. However, LSTM differs from vanilla RNN in the presence of the memory cell or cell state - the part surrounded by the orange dashed line in Figure 3.8. The memory cell is the core of LSTM which acts as a long-term memory that captures long-term dependencies. It is analogs to a conveyer belt that runs beside the sequence to carry relevant information at any timestep to be used in later timesteps as needed (Chollet, 2017a). In its essence, LSTM is designed to capture relevant information that is useful for later timesteps. To appreciate the ability of LSTM to capture only relevant information in long sequences, suppose that the task at hand is to classify a movie review as positive or negative and then rank the movie based on that review. To excel in such a task, the most informative and expressive words - even if they appear early in the sequence - need to be captured as they provide a strong clue about the sentiment of the review while insignificant neutral words need to be discarded as they don't hold precious information that may help in the classification. In addition to the memory cell, there are different gates in the LSTM module that regulate the flow of information by deciding which relevant information to be kept and thus to be taken to the conveyer belt (memory cell) for later use and which irrelevant information to be thrown.

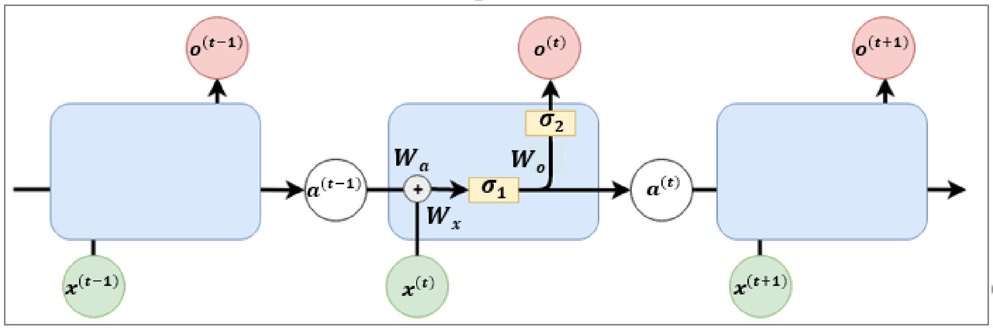


Figure 3.7: Traditional RNN module - vanilla RNN.
Figure adapted from (Olah, Understanding LSTM Networks, 2015).

These gates can be thought of as real gates that could be fully opened, partially opened or closed meaning that these gates control not only the kind of information to be kept or thrown but also how much of information should be remembered or forgotten ("A numerical example of LSTMs," 2017).

Mainly, LSTM gates have control over the memory cell from which they can add or remove information (Olah, Understanding LSTM Networks, 2015). These gates are learned during training to become successful at capturing relevant information needed later for prediction and throwing irrelevant information (Nguyen, 2019). Usually, LSTM has three gates as shown in Figure 3.8. Every gate consists of a fully connected layer with a Sigmoid activation followed by an element-wise multiplication ("A numerical example of LSTMs," 2017). As mentioned earlier, Sigmoid function produces values between 0 and 1 where 0 indicates pass nothing through the gate and 1 indicates pass everything through the gate.

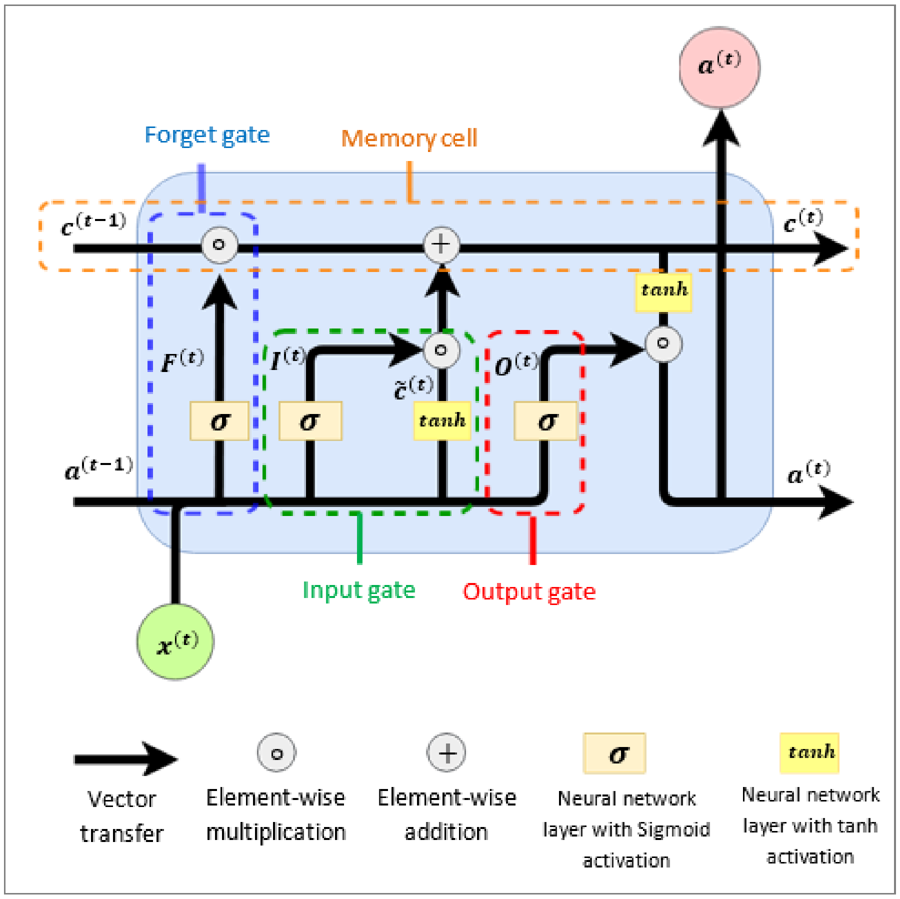


Figure 3.8: Long-short term memory (LSTM).
Figure adapted from (Olah, Understanding LSTM Networks, 2015; Nguyen, 2019).

To summarize, the memory cell in LSTM represents the long-term memory which is getting updated first through the forget gate where irrelevant information is removed, and second through the input gate where new information is added. $a^{(t)}$ represents the working memory (short-term memory) where LSTM keeps only the immediately useful information from the memory cell (Chen E. , 2017). Notice that $a^{(t)}$ usually is referred to as the hidden state. Below, each gate is presented briefly.

Forget gate: As the name indicates, this is the part of LSTM responsible for deciding which irrelevant information to be abandoned (and by how much).

The following formula shows the output $F^{(t)}$ of the forget gate (Olah, Understanding LSTM Networks, 2015):

$$F^{(t)} = \sigma(W_f . \left[a^{(t-1)}, x^{(t)}\right] + b_f) \tag{3.12}$$

Where:

σ	: Sigmoid function
W_f	: weight matrix of the forget gate
$a^{(t-1)}$	: activation vector of the previous timestep *t-1* (output of the previous timestep)
$x^{(t)}$	: input vector of the current timestep *t*
$\left[a^{(t-1)}, x^{(t)}\right]$	: concatenation of the two vectors
b_f	: bias vector

By looking at $a^{(t-1)}$ and $x^{(t)}$ then applying Sigmoid, the forget gate outputs a number in the range 0 - 1 for every number in $c^{(t-1)}$, the old memory cell (long-term memory). Doing so, the forget gate discards irrelevant information from $c^{(t-1)}$ where 0 indicates forgetting entirely while 1 indicates the opposite (Olah, Understanding LSTM Networks, 2015). After calculating $F^{(t)}$ as shown above, $F^{(t)}$ is getting multiplied by the old memory cell $c^{(t-1)}$. This symbol $\circ$ represents element-wise multiplication (Hadamard product) in which every element in the first matrix or vector is getting multiplied by the corresponding element in the second matrix or vector as shown in the following equation:

$$\begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix} \circ \begin{bmatrix} b_1 & b_2 \\ b_3 & b_4 \end{bmatrix} = \begin{bmatrix} a_1 . b_1 & a_2 . b_2 \\ a_3 . b_3 & a_4 . b_4 \end{bmatrix} \tag{3.13}$$

Input gate: This gate consists of two parts. The first part shown in Figure 3.9 is the one responsible for creating new candidate values vector $\tilde{c}^{(t)}$ learned from the input vector $x^{(t)}$ which can be added to the new memory cell $c^{(t)}$ to become a part of the long-term memory (Chen E. , 2017). This part of the input gate is represented mathematically by the following equation (Olah, Understanding LSTM Networks, 2015):

$$\tilde{c}^{(t)} = tanh\left(W_c.\left[a^{(t-1)}, x^{(t)}\right] + b_c\right) \tag{3.14}$$

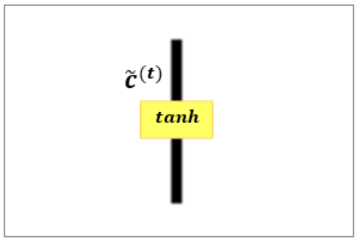


Figure 3.9: First part of the input gate.

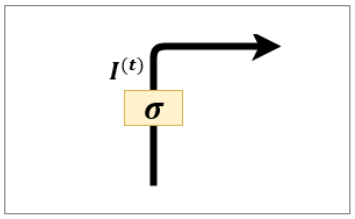


Figure 3.10: Second part of the input gate.

The other part shown in Figure 3.10 is responsible for determining which values of the candidate vector $\tilde{c}^{(t)}$ worth to be added to the new memory cell (long-term memory) $c^{(t)}$ (Chen E. , 2017). The following equation describes this part of the input gate mathematically (Olah, Understanding LSTM Networks, 2015):

$$I^{(t)} = \sigma(W_i.\left[a^{(t-1)}, x^{(t)}\right] + b_i) \tag{3.15}$$

Finally, these two parts of the input gate are getting combined through element-wise multiplication as shown in Figure 3.11.

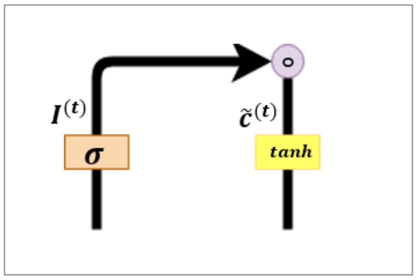


Figure 3.11: The two parts of the input gate combined through element-wise multiplication.

As shown previously, the forget gate determines which irrelevant information that is no longer needed to be memorized and thus need to be dropped from the memory cell. This is achieved through multiplying $F^{(t)}$ with the old memory cell $c^{(t-1)}$. After forgetting irrelevant information from the old memory cell, the following step of updating the memory cell is to multiply $I^{(t)}$ with $\tilde{c}^{(t)}$. This means that the new candidate values $\tilde{c}^{(t)}$ are getting scaled by $I^{(t)}$ according to their importance. Finally, to finish updating the memory cell, the input gate and the forget gate are getting combined as follows (Olah, Understanding LSTM Networks, 2015):

$$c^{(t)} = F^{(t)} \circ c^{(t-1)} + I^{(t)} \circ \tilde{c}^{(t)} \tag{3.16}$$

Output gate: This part outputs the activation $a^{(t)}$ of the LSTM cell. $a^{(t)}$ be thought of as the working memory (short-term memory) of LSTM. The output gate learns to focus on the parts of the long-term memory that are immediately useful (Chen E. , 2017). The following equations illustrate how $a^{(t)}$ is obtained (Olah, Understanding LSTM Networks, 2015):

$$O^{(t)} = \sigma(W_o . \left[a^{(t-1)}, x^{(1)}\right] + b_o) \tag{3.17}$$

$$a^{(t)} = O^{(t)} \circ tanh\ (\ c^{(t)}) \tag{3.18}$$

3.6.2 Gated Recurrent Unit – GRU

More recently, gated recurrent unit (GRU) was proposed in 2014 (Cho et al., 2014b).

Compared with LSTM, GRU doesn't have separate memory cells. Moreover, GRU uses two gates instead of three. Although GRU has a simpler structure than LSTM, the former has outperformed the latter empirically in some cases (Chung et al., 2014). Figure 3.12 is an illustration of a GRU cell that consists of two gates, a reset gate and an update gate. At every timestep, the GRU cell takes the current input $x^{(t)}$ and the old activation of the previous timestep $t-1$, that is $a^{(t-1)}$ as an input to produce the new activation $a^{(t)}$ of the current timestep *t*. The reset gate controls how much of the old activation $a^{(t-1)}$ should be included in the candidate activation $\tilde{a}^{(t)}$. Afterward, the update gate decides the amount in which the old activation $a^{(t-1)}$ and the candidate activation $\tilde{a}^{(t)}$ should participate in the calculation of the new activation $a^{(t)}$. Below, a brief illustration of the GRU gates is given.

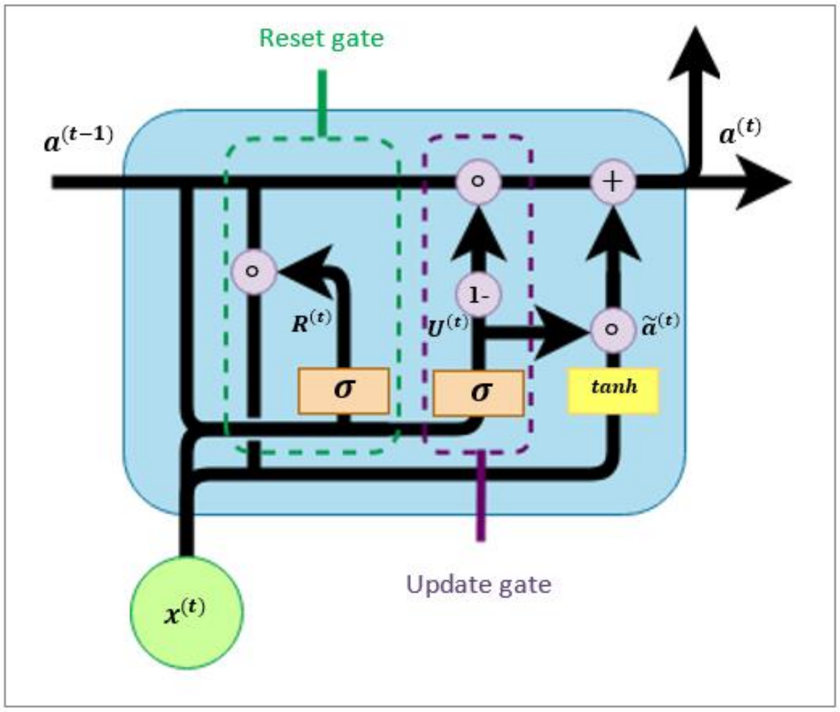


Figure 3.12: Gated Recurrent Unit (GRU).
Figure adapted from (Nguyen, 2019; Drakos, 2019).

Reset gate $\mathbf{R}^{(t)}$: The role of the reset gate is to control how much of the old activation $a^{(t-1)}$ should be forgotten. i.e. how much of the old activation $a^{(t-1)}$ should be included when calculating the candidate activation $\tilde{a}^{(t)}$. When $R^{(t)} \approx 0$, it means forget the old activation $a^{(t-1)}$ and look at $x^{(t)}$ only to form the candidate activation $\tilde{a}^{(t)}$. When $R^{(t)} \approx$

1, it means remember the old activation $a^{(t-1)}$ and use it beside $x^{(t)}$ to form the candidate activation $\tilde{a}^{(t)}$ (Chung et al., 2014). Below is the equation of the reset gate (Drakos, 2019):

$$R^{(t)} = \sigma(W_r.\left[a^{(t-1)}, x^{(t)}\right] + b_r) \tag{3.19}$$

Update gate $\mathbf{U^{(t)}}$: The update gate controls the amount of the update that should be made to the new activation. i.e. $U^{(t)}$ decides the amount of which the old activation $a^{(t-1)}$ and the candidate activation $\tilde{a}^{(t)}$ should participate in the calculation of the new activation $a^{(t)}$. When $U^{(t)} \approx 0$, it means keep the old activation $a^{(t-1)}$ and use it as the new activation $a^{(t)}$ while discarding the candidate activation $\tilde{a}^{(t)}$. When $U^{(t)} \approx 1$, it means discard the old activation $a^{(t-1)}$ and use the candidate activation as the new activation $a^{(t)}$. All gates shown previously have similar formulas with different weights and biases and the reset gate is no exception as shown below (Drakos, 2019):

$$U^{(t)} = \sigma(W_u.\left[a^{(t-1)}, x^{(t)}\right] + b_u) \tag{3.18}$$

Two quantities are involved in the calculation of the candidate activation $\tilde{a}^{(t)}$, one of them is $x^{(t)}$ and the other one is $a^{(t-1)}$ which is controlled by $R^{(t)}$ as shown below (Drakos, 2019):

$$\tilde{a}^{(t)} = tanh\left(W_a.\left[R^{(t)} \circ a^{(t-1)}, x^{(t)}\right] + b_a\right) \tag{3.20}$$

The final output of the GRU is the new activation $a^{(t)}$ which is a linear interpolation between two values $a^{(t-1)}$ and $\tilde{a}^{(t)}$ (Chung et al., 2014).
The following equation shows how $a^{(t)}$ is obtained (Drakos, 2019):

$$a^{(t)} = (1 - U^{(t)}) \circ a^{(t-1)} + U^{(t)} \circ \tilde{a}^{(t)} \tag{3.21}$$

3.7 **Residual Connections**

The idea of residual connections, also known as skip connections, was used by Lin et al., (1996) following the work of Lang & Hinton, (1988) about delays in FNNs. (Goodfellow et al., 2016). Later, a Microsoft research team won the 1st place on the ILSVRC 2015 classification task using residual nets with 125 layers depth. Afterward, the team has published a technical report of their experiments on the ImageNet test set. (He et al., 2015). Their work provided a piece of empirical evidence that the use of residual connections has improved the performance of very deep convolutional neural networks. Specifically, the authors have observed that stacking more layers doesn't improve performance, on the contrary, it leads to higher training and testing errors, which is counter-intuitive since a deeper model is supposedly able to learn the representations learned by the shallower model, i.e. the former is expected to have, at least, the same error as the latter.

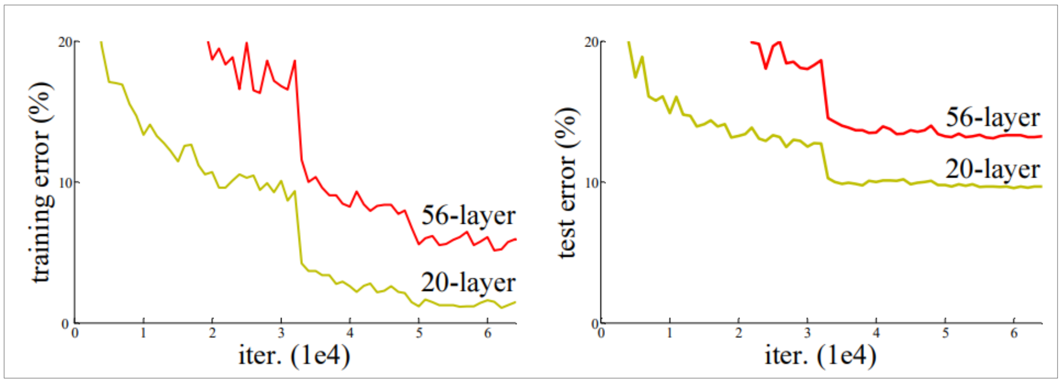


Figure 3.13: (left) training error (right) testing error.
The plots show training and testing errors of 20 layers (yellow curve) and 56 layers (red curve) convolutional networks without residual connections on CIFAR-10. Deeper network has higher training and testing errors (He et al., 2015).

The abovementioned problem is not caused by overfitting since the training error of the deeper model is higher than that of the shallower model. Therefore, the work of He et al., (2015) introduced the residual connections as a solution to this problem. The reason behind the success of the residual connections is unclear but it's empirically evident. A residual connection is a connection that skips one or more layers. Figure 3.14 shows a residual block with a residual connection that bypasses every two layers

In Figure 3.14, the connection takes the output of a previous layer x and adds it to the output of the next two layers $f(x)$. The output of the residual block is $f(x) + x$. The connection is nothing more than an identity mapping in which x passes unchanged.

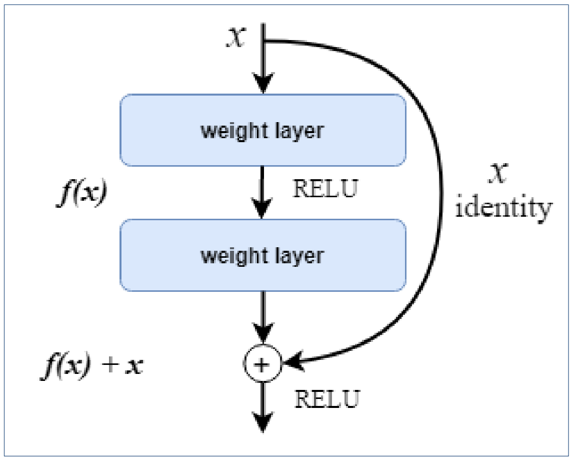


Figure 3.14: The residual block used by He et al., (2015).

Let's say that the two layers are supposed to learn a mapping h, i.e. the original mapping to be learned is h. Hence, x is the input and $h(x)$ is the output of these layers.

However, by introducing the residual connection as shown in Figure 3.14, the desired mapping to be learned by the layers is no longer $h(x)$, rather, its $f(x)$.

Hence, it follows logically that:

1) Desired mapping is $f(x) = h(x) - x$.
2) Original mapping is $h(x) = f(x) + x$.

The authors hypothesized that learning the residual mapping $f(x)$ is easier than learning the original mapping $h(x)$ by providing the following example:

If the original mapping to be learned were the identity mapping, i.e. $h(x) = x$, then it is easier to push the residual $f(x)$ to 0 than to let the stacked layers learn the identity mapping without a residual connection. To elaborate on the example, when the original mapping is the identity, $h(x) = x$ then it follows that, with the presence of the residual connection, the desired mapping is $f(x) = h(x) - x = x - x = 0$. Hence, the layers will try to push $f(x)$ towards 0 rather than trying to learn the identity mapping.

3.8 **Gradient Clipping**

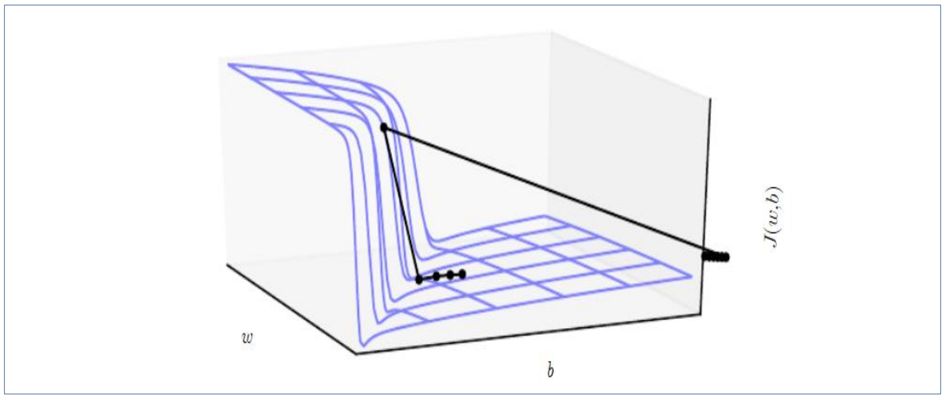


Figure 3.15: Cost function surface of highly non-linear models.
The cost function $J(w, b)$ is plotted as a function of its parameters w and b showing the error surface with a wall that indicates an abrupt change in the error. In RNNs, Repetitive multiplication of weights results in exremely steep regions (walls) as shown here in the error surface of the cost function $J(w, b)$. These When approached by gradient descent during optimization, these walls may cause the gradient descent to update the parameters with values very far from the optimal solution (Goodfellow et al., 2016).

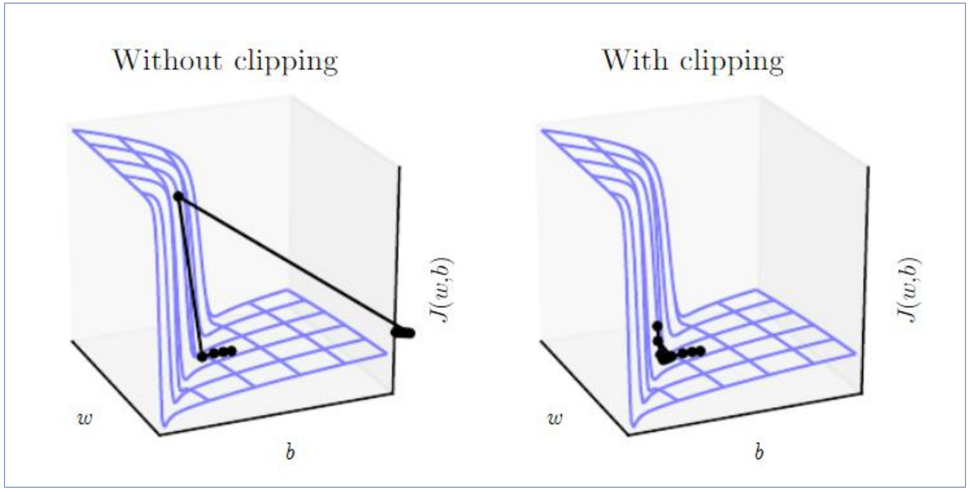


Figure 3.16: The effect of using gradient clipping.
(left): Without gradient clipping, gradient descent step moves from the valley to the wall (it moves suddenly from a low error region to a high error region) resulting in a high gradient that updates the parameters with new values outside the axes of the plot. (right): With gradient clipping, even though the gradient descent step ascends the wall, the step size isn't too big to update the parameters with values far away from the solution (Goodfellow et al., 2016).

Usually, highly nonlinear models such as RNNs and deep FNNs have cost functions with very steep regions (walls) as shown in Figure 3.15. These regions result from repeated multiplication of parameters (weights) thus have large gradients. Avoiding steep regions is essential since getting close to them during optimization my cause the gradient descent to make an update of the parameters leading them to a region far away from the desired solution and thus losing the progress that have been made so far (Goodfellow et al., 2016). Gradient clipping is a simple mechanism to deal with exploding gradient problem. Two variations of gradient clipping were suggested with minor differences between them (Pascanu et al., 2012; Uení et al., 2012). Figure 3.7 shows the effect of using gradient clipping. When the gradient is large, the update of the parameters may lead to a region where the cost function is larger. Therefore, the step size must be small enough to prevent the update from taking a big step upward the wall. As illustrated in figure 3.7, gradient clipping prevents gradient descent from overshooting the minima by keeping the gradient less than some threshold as follows (Pascanu et al., 2012):

If $\|g\| > threshold$ then:

$$g = \frac{g \times threshold}{\|g\|} \tag{3.12}$$

Where, $\|g\|$ is the norm of the gradient vector g which can be possibly the L-1 norm of g, that is $\|g\|_1$ or the L-2 norm of g, that is $\|g\|_2$.
Simply, The L-1 norm of a vector $x \in \mathbb{R}^n$ with elements $x = (x_1, x_2, \dots, x_n)$ is the sum of its elements magnitudes as follows:

$$\|x\|_1 = \sum_{i=1}^{n} |x_i| \tag{3.14}$$

The L-2 norm is the Euclidean distance and is expressed mathematically as follows:

$$\|x\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2} \tag{3.13}$$

3.9 **RNN Architectures**

RNN may have different architectures, each of which fulfills the need of a specific application. The variations of RNNs input and output sizes in different applications led to the emergence of a rich family of RNNs architectures. Below is a brief presentation of RNNs architectures.

One-to-one neural network: $T_x = T_o = 1$. As illustrated in Figure 3.15, $a^{(0)}$ is the activation at time step 0, x and o are the input and output vectors respectively. This is the vanilla mode of processing where no RNN is used. One-to-one network takes one input vector x and generates one output vector o. This architecture is used in diverse applications such as image classification.

One-to-many RNN: $T_x = 1$ and $T_o > 1$. Music generation is an example of one-to-many RNN where the task is to generate original musical compositions that humans expect to hear. Another application is image captioning in which a single image is processed, and the output is an annotation of that image. Figure 3.16 demonstrates One-to-many RNN architecture where the RNN takes one input which is image for image captioning application or maybe nothing as the case of music generation and generates output at each time step.

Many-to-one RNN: $T_x > 1$ and $T_y = 1$. This architecture is useful for the task of sentiment classification where a sentence is classified as conveying negative or positive sentiment. For example, a text review is given as an input and the output is supposed to be an integer that represents the rating of the reviewed item according to the written review. One-to-one RNN is shown in Figure 3.17. The RNN takes input at each time step (a word in sentiment classification) and generates a single output (the rating).

Many-to-many RNN (input and output have the same length): $T_x = T_o$. A well-known application that uses this architecture is Named-entity recognition or NER. As the name suggests, NER seeks to locate named entities in a text and classify these names into a set of predefined categories such as organizations, person names, locations etc.

The output would be the same input text where named entities are highlighted and annotated (Wikipedia, 2019). Figure 3.18 shows many-to-many RNN where the input is the same size as the output.

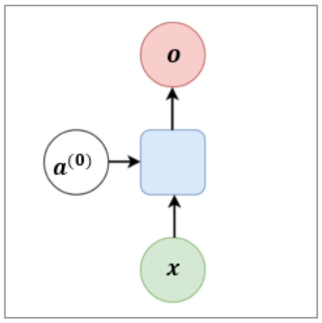


Figure 3.16: One-to-one RNN. Figure adapted from (Amidi & Amidi).

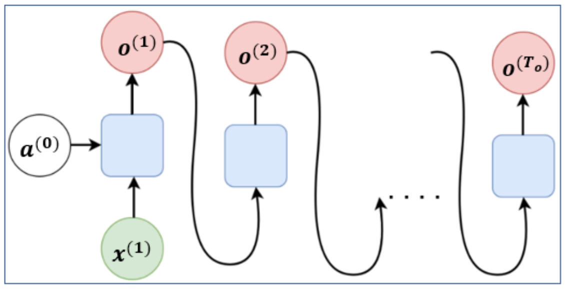


Figure 3.17: One-to-many RNN.
Figure adapted from (Amidi & Amidi).

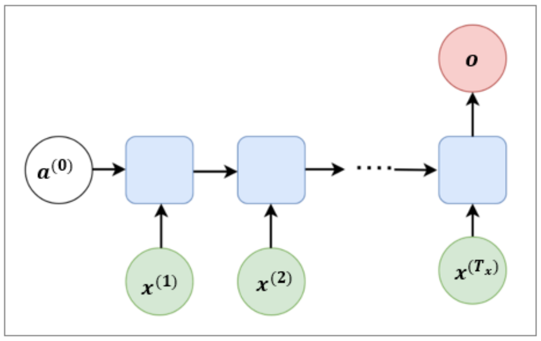


Figure 3.18: Many-to-one RNN.
Figure adapted from (Amidi & Amidi).

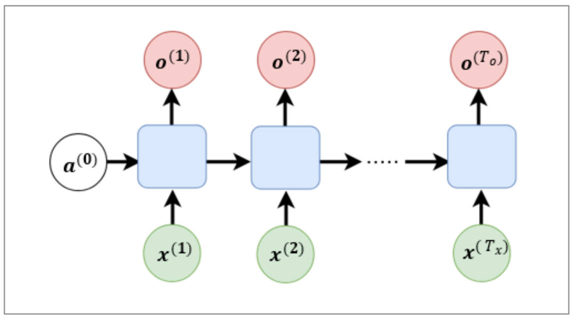


Figure 3.19: Many-to-many RNN, $T_x = T_o$.
Figure adapted from (Amidi & Amidi).

Many-to-many RNN (input and output have variable lengths): $T_x \neq T_o$. The task of machine translation is one of the most famous applications that uses this architecture. A sentence is translated from the source language to the target language. Clearly, input and output sequences would not be of the same length. This architecture is illustrated in Figure 2.19.

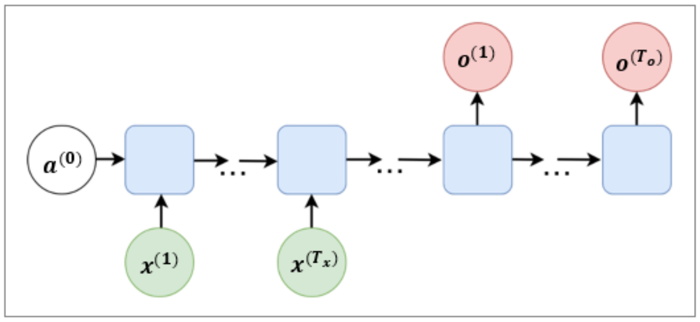


Figure 3.20: Many-to-many RNN, $T_x \neq T_o$.
Figure adapted from (Amidi & Amidi).

3.10 Bidirectional Recurrent Neural Networks

All previously presented architectures process information in one direction, meaning that the hidden state $a^{(t)}$ at any time step t encapsulates information about the past $x^{(1)}, \dots, x^{(t-1)}$ and the present $x^{(t)}$. However, some applications might need to process the *whole sequence* to output a prediction $o^{(t)}$ (Goodfellow et al., 2016). Consider the following example, which is taken from (Zhang et al., 2019).

```
1. I am _____
2. I am _____ very hungry.
3. I am _____ very hungry, I could eat three chickens.
```

If the task were to fill the blanks, we might choose different words such as ‘happy’, ‘not’ and ‘very’. Apparently, the end of the sentence provides important information about which word to choose. A model that doesn’t make use of such information will lose its’ predictive power. Named-entity recognition NER is another example where longer-range context is critical (e.g. does Brown refer to a person or to the color?). To address the need of processing the whole sequence, Bidirectional RNNs (BRNNs) were invented (Schuster & Paliwal, 1997).

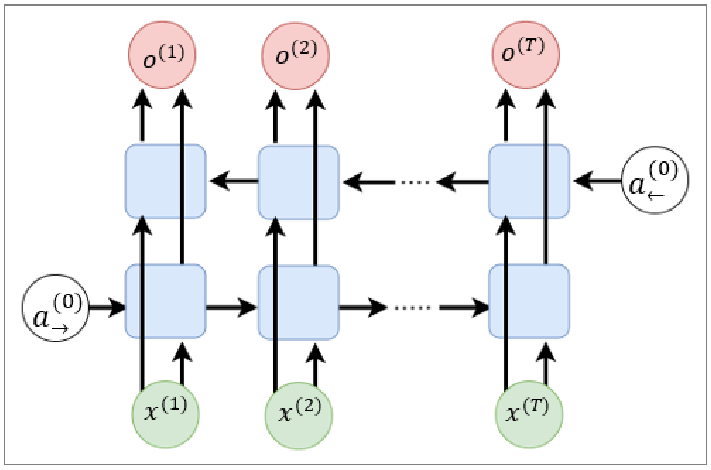


Figure 3.21 Bidirectional RNN.

BRNNs have proven to be successful in many applications including handwriting recognition, speech recognition and bioinformatics (Goodfellow et al., 2016). Figure 4.8 shows a BRNN. Unlike traditional RNN, BRNNs constitute of two RNNs. One is a forward RNN that processes the sequence in the forward direction and computes a forward state. The other is a backward RNN that processes the sequence in the backward direction and computes a backward state. Thus, BRNN computes two hidden states, forward and backward hidden states at each time step *t*. For a given time step *t*, the mini-batch input is $x^{(t)} \in \mathbb{R}^{b\times n}$ where b refers to the number of examples in the mini-batch (batch size) and n refers to the number of features. Forward and backward hidden states are $a_{\rightarrow}^{(t)} \in \mathbb{R}^{b\times h}$ and $a_{\leftarrow}^{(t)} \in \mathbb{R}^{b\times h}$ where h refers to the number of hidden units. At time step t, forward and backward hidden states (activations) are computed as follows:

$$a_{\rightarrow}^{(t)} = \sigma_1\left(W_a^f a^{(t-1)} + W_x^f x^{(t)} + b_a^f\right) \tag{3.14}$$

$$a_{\leftarrow}^{(t)} = \sigma_1 \left(W_a^b a^{(t-1)} + W_x^b x^{(t)} + b_a^b \right) \tag{3.15}$$

Where:

$x^{(t)} \in \mathbb{R}^{b \times n}$: mini-batch input at current time step *t*

$a^{(t)} \in \mathbb{R}^{b \times h}$: hidden state (activation) at current time step *t*

$a^{(t-1)} \in \mathbb{R}^{b \times h}$: hidden state (activation) at previous time step *t-1*

$a_{\rightarrow}^{(t)} \in \mathbb{R}^{b \times h}$: forward hidden state at current time step *t*

$a_{\leftarrow}^{(t)} \in \mathbb{R}^{b \times h}$: backward hidden state at current time step *t*

$W_a^f \in \mathbb{R}^{n \times h}$, $W_x^f \in \mathbb{R}^{h \times h}$, $W_a^b \in \mathbb{R}^{n \times h}$ and $W_x^b \in \mathbb{R}^{h \times h}$: the weight matrices

$b_a^f \in \mathbb{R}^{1 \times h}$ and $b_a^b \in \mathbb{R}^{1 \times h}$: the biases

σ_1 : activation function

b : number of examples in the input mini-batch $x^{(t)}$ (batch size)

n : number of features

h : number of hidden units

q : number of output units

Afterwards, forward and backward hidden states, $a_{\rightarrow}^{(t)}$ and $a_{\leftarrow}^{(t)}$ are concatenated to form $a^{(t)} \in \mathbb{R}^{b \times 2h}$.

Finally, the output is computed as in traditional RNNs:

$$o^{(t)} = \sigma_2\left(W_o a^{(t)} + b_o\right) \tag{3.16}$$

Where:

$o^{(t)} \in \mathbb{R}^{b \times q}$	: output at time step t
$W_o \in \mathbb{R}^{2h \times q}$	: weight matrix
$b_o \in \mathbb{R}^{1 \times q}$	: bias
σ_2	: activation function

3.11 Encoder-Decoder Sequence-to-Sequence Architecture

Many applications involve processing sequences of variable lengths. Machine translation, video captioning, speech recognition, and question answering among many others may have input and output sequences of different lengths (Goodfellow et al., 2016). Two pioneering works were first to propose an RNN architecture that maps two sequences of variable-lengths, Cho K. et al., (2014b) and Sutskever et al., (2014).

The authors of both works referred to the proposed architecture as encoder-decoder or sequence-to-sequence respectively (Goodfellow et al., 2016). Generally, a sequence-to-sequence (Seq2Seq) or encoder-decoder model aims at learning to convert an input sequence from one domain (e.g. source language) to an output sequence from another domain (e.g. target language) (Chollet, 2017b).

A (Seq2Seq) model is composed of two RNNs:

a) An *encoder* that takes the input sequence element by element and produce a fixed-length vector, namely the *context vector* C. Typically, the last hidden state of the *encoder* is used as the context vector (Goodfellow et al., 2016).
b) A *decoder* that uses the context vector C either as its initial hidden state or by connecting it to the hidden units at each timestep. Both ways can be used in combination (Goodfellow et al., 2016).

In fact, the encoder-decoder architecture is a many-to-many RNN where the length of the input sequence doesn't necessarily match that of the output sequence, $T_x \neq T_o$. The *encoder* is a regular RNN that processes the input sequence x sequentially and updates its hidden state (activation) $a_d^{(t)}$ at each timestep *t* as follows (Cho K. , et al., 2014b):

$$a_e^{(t)} = f\left(a_e^{(t-1)}, x^{(t)}\right) \tag{3.18}$$

When the *encoder* scans the whole input sequence, the activation of the last timestep *t* represents the context vector C. Ideally, the context vector summarizes the entire input sequence (Cho K. , et al., 2014b). However, the *decoder's* activation is slightly different, since its activation $a_d^{(t)}$ at timestep *t* depends on the previous activation $a_d^{(t-1)}$, the previous output $o^{(t-1)}$, and the context vector C as follows (Cho K. , et al., 2014b):

$$a_d^{(t)} = f\left(a_d^{(t-1)}, o^{(t-1)}, C\right) \tag{3.19}$$

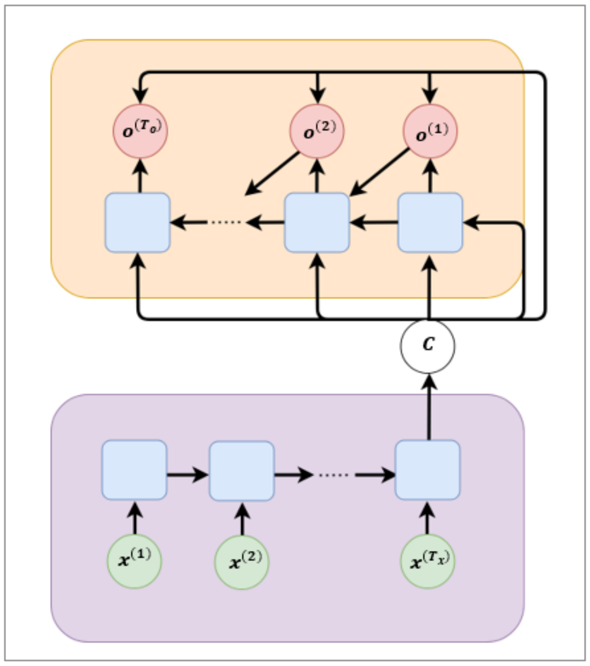


Figure 3.22: Encoder-decoder architecture.
Figure adapted from (Goodfellow et al., 2016).

3.12 Attention Mechanism

The notion of attention has gained popularity in diverse learning applications. In the context of machine translation, the work of Bahdanau et al., (2015) was the first to exploit the power of attention (Luong et al., 2015). Authors of Bahdanau et al., (2015) argue that the use of a fixed-length context vector in Seq2Seq models is a bottleneck, especially for long sequences. Instead of putting the burden on the context vector alone to encode the entire input sequence, the attention allows the decoder at every step of decoding to selectively focus on (attend to) different parts of the input sequence that are more relevant to predicting the current part of the output sequence.

Unlike attention decoder, classic decoder ignores all hidden states of the encoder and uses the last hidden state (context vector) alone during decoding which is obviously a waste of precious information about the input sequence encoded by the overlooked hidden states. However, attention decoder takes all hidden states into consideration. Based on the architecture used by Bahdanau et al., (2015), the work of Luong et al., (2015) suggested a simpler version of attention which has achieved the state-of-the-art results in the neural machine translation task. Below is an illustration the attention mechanism as described in Luong et al., (2015).

At every decoding step t:

1) Calculate energy $e^{(j)}$:

$$e^{(j)} = f_1(a_d^{(t-1)}, a_e^{(j)}) \tag{3.20}$$

Where:

$e^{(j)}$	: j_{th} energy
$a_d^{(t-1)}$	: previous decoder's activation
$a_e^{(j)}$	: j_{th} encoder's activation
f_1	: a linear layer

2) Calculate attention weights $aw^{(j)}$ by taking the Softmax of the j_{th} energy $e^{(j)}$ over all energies:

$$aw^{(j)} = \frac{exp\ (e^{(j)})}{\sum_{k=1}^{T} exp(e^{(k)})} \tag{3.21}$$

Where:

T : length of the source sequence

3) Calculate the context vector c:

$$c = \sum_{j=1}^{T} aw^{(j)} a_e^{(j)} \tag{3.22}$$

4) Calculate the current decoder's activation $a_d^{(t)}$:

$$a_d^{(t)} = rnn(a_d^{(t-1)}, o_d^{(t-1)}, c) \tag{3.23}$$

Where:

$a_d^{(t-1)}$: previous decoder's activation

$o_d^{(t-1)}$: previous decoder's output

c : current context vector

rnn : RNN function

5) Calculate the decoder's current output $o_d^{(t)}$ depends on 3 quantities as follows:

$$o_d^{(t)} = f_2(o_d^{(t-1)}, a_d^{(t)}, c) \tag{3.20}$$

Where:

f_2 : a linear layer

Chapter 4

MATERIALS AND METHODS

4.1 Materials

4.1.1 Human3.6 Dataset

As mentioned earlier, Human3.6 dataset was used in the experimentation of this work as in previous works. Currently, Human3.6 is the largest publicly available motion dataset with 3.6 million 3D human poses (Ionescu et al., 2011; Ionescu et al., 2014). It has 50Hz framerate - 50 frames per second (fps) and it provides motion data for 7 professional actors (subjects) performing 15 different activities. Human3.6 uses an articulated skeleton with 32 joints. Furthermore, Human3.6 provides motion data in several formats including image data, time-of-flight data, scanner data, and pose data. Primarily, pose data includes two parametrizations, joint positions and joint angles. As the adopted dataset in this work comes in pose data format, this format will be explored in more detail in the next section.

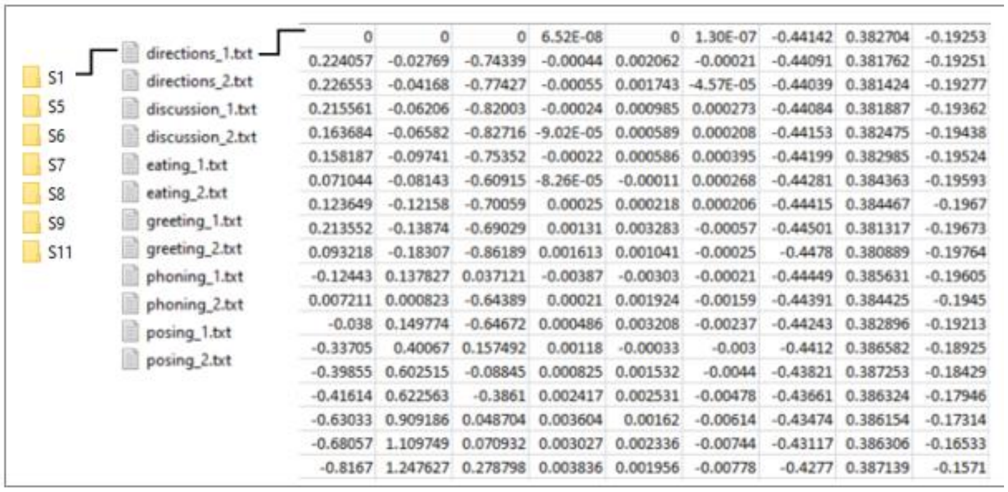
4.1.2 Human3.6 Pose Parametrizations

Motion data parametrization is the task of converting motion data raw format to a numerical format suitable for data analysis (Du, Manns, Herrmann, & Fischer, 2016). Originally, Human3.6 dataset is available in two parametrizations, relative 3D joint positions (R3DJP) and Kinematic representation (KR). These representations are common in the literature, each one has its own strengths and weaknesses. (R3DJP) representation uses the 3D cartesian positions directly to represent the joints of the skeleton. This representation doesn't preserve the skeleton structure and thus it requires constraints to be imposed on the skeleton (bone-length constraints) to make sure that the distances between joints remain fixed throughout the animation (Komura, Habibie, Schwarz, & Holden, 2017; Du et al., 2016).

Imposing these constraints comes with its own cost as it requires non-linear optimization (Komura et al., 2017). On the other hand, (KR) representation depends on the angles of the joints rather than their positions. Specifically, (KR) uses the translation (position) and orientation of the root joint as well as the relative orientations of other joints (Du et al., 2016). (KR) connects joints through a parent-child relationship to form an articulated skeleton. The movement of a joint is relative to its parent. (KR) requires forward kinematics to obtain 3D joints positions which involves non-linear transformations and trigonometric functions (Komura et al., 2017). Unlike (R3DJP), (KR) preserves the structure of the skeleton and thus it is most commonly used in motion analysis tasks. Also, it is the standard representation for most (MOCAP) datasets (Du et al., 2016). The dataset adopted in this work uses (KR) representation.

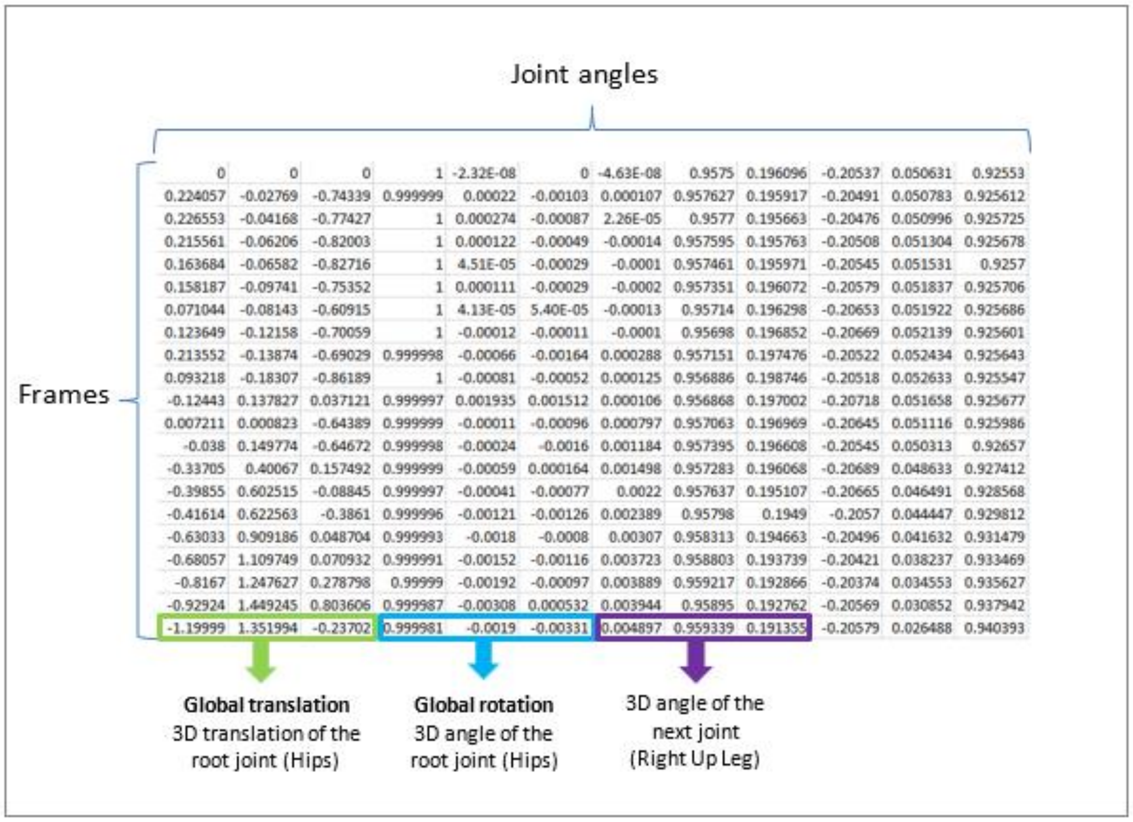
4.1.3 **Human3.6 Preprocessing**

Following earlier works (Jain et al., 2016; Martinez et al., 2017; Fragkiadaki et al., 2015; Pavllo et al., 2019), this work uses a preprocessed version of H3.6 dataset provided by (Jain et al., 2016). The preprocessing of the dataset follows the approach of (Taylor et al., 2007) in which the original raw angles were converted from Euler angle to exponential maps representation alongside with a special preprocessing applied to the root joint. The preprocessed H3.6 dataset provides motion data for 7 professional actors or subjects performing 15 different activities including: walking, eating, smoking, discussion, directions, greeting, phoning, posing, purchases, sitting, sitting down, taking photo, waiting, walking dog and walking together. The motion data of every subject contains 15 actions where each action contains 2 sub-actions, i.e. the same action is performed twice by the subject resulting in 30 actions for each subject. The dataset is arranged as shown in Figure 4.1. The motion data for each subject is stored in a folder such that the name of the folder consists of a capital “S” followed by the number of the subject. For example, the folder of the 1st subject has the name “S1”. Other folders are given names in the same manner.



0	0	0	6.52E-08	0	1.30E-07	-0.44142	0.382704	-0.19253
0.224057	-0.02769	-0.74339	-0.00044	0.002062	-0.00021	-0.44091	0.381762	-0.19251
0.226553	-0.04168	-0.77427	-0.00055	0.001743	-4.57E-05	-0.44039	0.381424	-0.19277
0.215561	-0.06206	-0.82003	-0.00024	0.000985	0.000273	-0.44084	0.381887	-0.19362
0.163684	-0.06582	-0.82716	-9.02E-05	0.000589	0.000208	-0.44153	0.382475	-0.19438
0.158187	-0.09741	-0.75352	-0.00022	0.000586	0.000395	-0.44199	0.382985	-0.19524
0.071044	-0.08143	-0.60915	-8.26E-05	-0.00011	0.000268	-0.44281	0.384363	-0.19593
0.123649	-0.12158	-0.70059	0.00025	0.000218	0.000206	-0.44415	0.384467	-0.1967
0.213552	-0.13874	-0.69029	0.00131	0.003283	-0.00057	-0.44501	0.381317	-0.19673
0.093218	-0.18307	-0.86189	0.001613	0.001041	-0.00025	-0.4478	0.380889	-0.19764
-0.12443	0.137827	0.037121	-0.00387	-0.00303	-0.00021	-0.44449	0.385631	-0.19605
0.007211	0.000823	-0.64389	0.00021	0.001924	-0.00159	-0.44391	0.384425	-0.1945
-0.038	0.149774	-0.64672	0.000486	0.003208	-0.00237	-0.44243	0.382896	-0.19213
-0.33705	0.40067	0.157492	0.00118	-0.00033	-0.003	-0.4412	0.386582	-0.18925
-0.39855	0.602515	-0.08845	0.000825	0.001532	-0.0044	-0.43821	0.387253	-0.18429
-0.41614	0.622563	-0.3861	0.002417	0.002531	-0.00478	-0.43661	0.386324	-0.17946
-0.63033	0.909186	0.048704	0.003604	0.00162	-0.00614	-0.43474	0.386154	-0.17314
-0.68057	1.109749	0.070932	0.003027	0.002336	-0.00744	-0.43117	0.386306	-0.16533
-0.8167	1.247627	0.278798	0.003836	0.001956	-0.00778	-0.4277	0.387139	-0.1571

Figure 4.1: Preprocessed H3.6 dataset arrangement.


Joint angles
Frames
Global translation
3D translation of the root joint (Hips)
Global rotation
3D angle of the root joint (Hips)
3D angle of the next joint (Right Up Leg)

0	0	0	1	-2.32E-08	0	-4.63E-08	0.9575	0.196096	-0.20537	0.050631	0.92553
0.224057	-0.02769	-0.74339	0.999999	0.00022	-0.00103	0.000107	0.957627	0.195917	-0.20491	0.050783	0.925612
0.226553	-0.04168	-0.77427	1	0.000274	-0.00087	2.26E-05	0.9577	0.195663	-0.20476	0.050996	0.925725
0.215561	-0.06206	-0.82003	1	0.000122	-0.00049	-0.00014	0.957595	0.195763	-0.20508	0.051304	0.925678
0.163684	-0.06582	-0.82716	1	4.51E-05	-0.00029	-0.0001	0.957461	0.195971	-0.20545	0.051531	0.9257
0.158187	-0.09741	-0.75352	1	0.000111	-0.00029	-0.0002	0.957351	0.196072	-0.20579	0.051837	0.925706
0.071044	-0.08143	-0.60915	1	4.13E-05	5.40E-05	-0.00013	0.95714	0.196298	-0.20653	0.051922	0.925686
0.123649	-0.12158	-0.70059	1	-0.00012	-0.00011	-0.0001	0.95698	0.196852	-0.20669	0.052139	0.925601
0.213552	-0.13874	-0.69029	0.999998	-0.00066	-0.00164	0.000288	0.957151	0.197476	-0.20522	0.052434	0.925643
0.093218	-0.18307	-0.86189	1	-0.00081	-0.00052	0.000125	0.956886	0.198746	-0.20518	0.052633	0.925547
-0.12443	0.137827	0.037121	0.999997	0.001935	0.001512	0.000106	0.956868	0.197002	-0.20718	0.051658	0.925677
0.007211	0.000823	-0.64389	0.999999	-0.00011	-0.00096	0.000797	0.957063	0.196969	-0.20645	0.051116	0.925986
-0.038	0.149774	-0.64672	0.999998	-0.00024	-0.0016	0.001184	0.957395	0.196608	-0.20545	0.050313	0.92657
-0.33705	0.40067	0.157492	0.999999	-0.00059	0.000164	0.001498	0.957283	0.196068	-0.20689	0.048633	0.927412
-0.39855	0.602515	-0.08845	0.999997	-0.00041	-0.00077	0.0022	0.957637	0.195107	-0.20665	0.046491	0.928568
-0.41614	0.622563	-0.3861	0.999996	-0.00121	-0.00126	0.002389	0.95798	0.1949	-0.2057	0.044447	0.929812
-0.63033	0.909186	0.048704	0.999993	-0.0018	-0.0008	0.00307	0.958313	0.194663	-0.20496	0.041632	0.931479
-0.68057	1.109749	0.070932	0.999991	-0.00152	-0.00116	0.003723	0.958803	0.193739	-0.20421	0.038237	0.933469
-0.8167	1.247627	0.278798	0.99999	-0.00192	-0.00097	0.003889	0.959217	0.192866	-0.20374	0.034553	0.935627
-0.92924	1.449245	0.803606	0.999987	-0.00308	0.000532	0.003944	0.95895	0.192762	-0.20569	0.030852	0.937942
-1.19999	1.351994	-0.23702	0.999981	-0.0019	-0.00331	0.004897	0.959339	0.191355	-0.20579	0.026488	0.940393

Figure 4.2: Part of a motion data file from the adopted dataset.

ID	Name	parent	Children
0	Hips	0	1, 6, 11
1	Right Up Leg	0	2
2	Right Leg	1	3
3	Right Foot	2	4
4	Right Toe Base	3	5
5	Site	4	-
6	Left Up Leg	0	7
7	Left Leg	6	8
8	Left Foot	7	9
9	Left Toe Base	8	10
10	Site	9	-
11	Spine	0	12
12	Spine 1	11	13, 16, 24
13	Neck	12	14
14	Head	13	15
15	Site	14	-
16	Left Shoulder	12	17
17	Left Arm	16	18
18	Arm	17	19
19	Left Hand	18	20, 22
20	Left Hand Thumb	19	21
21	Site	20	-
22	Left wrist End	19	23
23	Site	22	-
24	Right Shoulder	12	25
25	Right Arm	24	26
26	Right Fore Arm	25	27
27	Right Hand	26	28, 30
28	Right Hand Thumb	27	29
29	Site	28	-
30	Right wrist End	27	31
31	Site	30	-

Table 3: Joints of Human3.6 skeleton.
The ID, name, parent and children of each joint are shown.

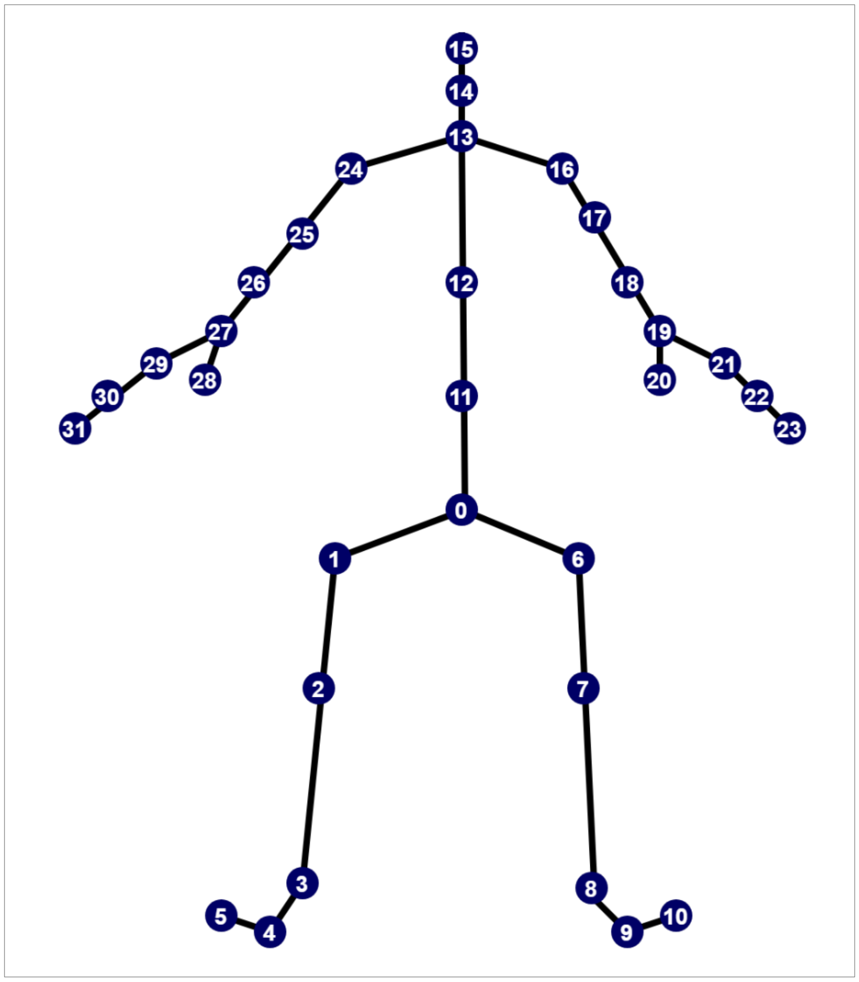


Figure 4.3: : H3.6 skeleton where joints are numbered according to their IDs given in Table 3.

Following previous works, subjects 1, 6, 7, 8, 9 and 11 were used for training and subject 5 was used for testing. As mentioned before, every subject has 2 sub-actions for the same action. For example, subject 1 has two files for the "directions" action, directions_1.txt and directions_2.txt. Figure 2.4 shows part of a motion data in the adopted dataset. For any motion data file in the dataset, "directions_1.txt" for example, the rows represent the number of frames of motion which vary from one file to another whereas the columns represent joint angles in exponential maps representation. The file resembles the comma-separates-values (CSV) format in which the comma is used as a delimiter. Every 3 elements in a row is related to a single joint except the first 3 elements, i.e. the first 3 elements represent 3D position information of the root joint “Hips”, the next 4 elements represent 3D joint angle information of the root joint and the next 4 elements represent 3D joint angle information of the next joint in the hierarchy “Right Up Leg” and so forth. The order in which joint angles appear corresponds to the order of joints in Table 3. As the angles are given in exponential maps representation, the total number of columns is 69 for all files. This is because the total number of joints is 32 where each joint angle is represented by 3 elements thus the total number of columns is the number of joints multiplied with 3 in addition to the first 3 elements of the root 3D position, i.e. (number of joint = 32) * (exponential maps elements = 3) + (root 3D position =3) = 69. In this work, Human3.6 dataset is down sampled by 2 as in previous works i.e. frame rate becomes 25 fps instead of 50 fps.

4.2 **Methods**

Before breaking down the architecture into its constituting building blocks, it is useful to introduce some important ideas as a preface to the incoming material. Some important terminologies are given below.

- *Sequence*: A set of sequential frames of motion.
- *Source sequence* (*input sequence*): The sequence that will be provided as input to the *encoder* during training and testing.
- *Target sequence* (*output sequence*): The sequence that the *decoder* will predict.
- *Teacher forcing:* Training the decoder by feeding the ground-truth at every decoding step (the ground-truth of the *decoder's* output of the previous timestep is fed as input to the next timestep).
- *Sampling:* Training the decoder by feeding its own predictions at every decoding step (the *decoder's* output of the previous timestep is fed as input to the next timestep).
- *Scheduled sampling:* A combination of *teacher forcing* and *sampling* in which the decoder is fed the ground-truth at some steps and its own predictions at other steps.
- *Time major:* The data is time major when its 0th dimension is the length of the sequence be it the source or target sequence. It's called *time major* because the length of the sequence represents the number of timesteps.

- *Batch major:* The data is time major when its 0th dimension is the batch size.

Since the motion dataset is down sampled to 25 frames/second, feeding 50 frames to the model is equivalent to feeding 2 seconds of motion and predicting 10 frames is equivalent to predicting 400 milliseconds.

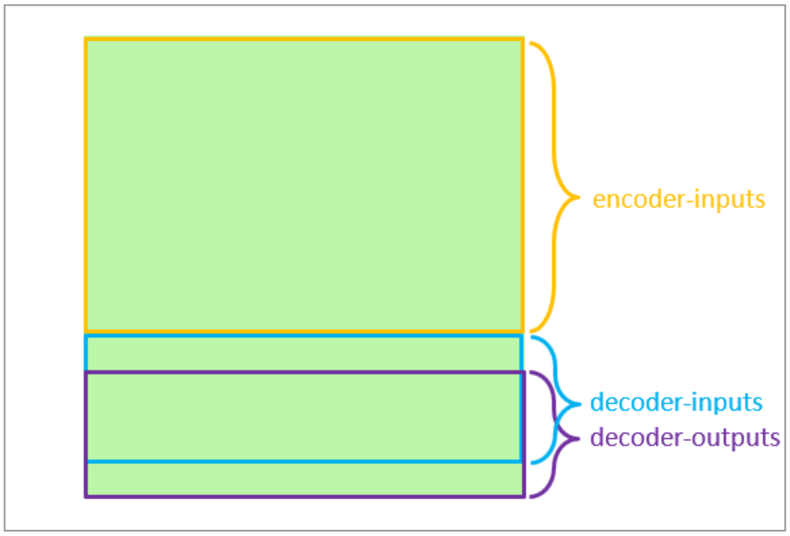


Figure 4.4: The division of encoder-inputs, decoder-inputs and decoder-outputs sequences.
Number of *encoder-inputs* frames = (input-sequence length) - 1. Number of *decoder-inputs* frames = output-sequence length. Number of *decoder-outputs* frames = output-sequence length. The *decoder-outputs* sequence is shifted one timestep ahead of the *decoder-inputs*.

During training, the Seq2Seq model needs 3 sequences which are extracted from the dataset. The extraction of theses sequences is illustrated in Figure 4.4. These sequences are as follows:

a) *encoder-inputs*: The sequence that is fed to the *encoder* during training as well as testing. All frames of the *source sequence* are taken as *encoder-inputs* except the last frame which is fed to the decoder at the first step of decoding. Hence, *encoder-inputs* is 1 frame less than the *source sequence*.

b) *decoder-inputs:* The sequence that is fed as input to the *decoder*. During training, this sequence will be always the ground-truth if pure *teacher forcing* is used. If pure *sampling* is used, it will be always *decoder's* predictions. However, if *scheduled sampling* is used, *decoder-inputs* is either the ground-truth or the *decoder's* predictions.

This is applicable only during training, during testing however, *decoder-inputs* is always *decoder's* predictions. *decoder-inputs* has the same length of the *target sequence*.

c) *decoder-outputs:* The *decoder's* desired output (ground-truth) that the *decoder's* prediction will be compared against. *decoder-outputs* length is the same as the *source sequence* length. *decoder-outputs* sequence is shifted one timestep ahead of the *decoder-inputs.*

The abovementioned sequences namely *encoder-inputs*, *decoder-inputs*, and *decoder-outputs* are supplied to the model ahead of training. During testing, *encoder-inputs* is only the sequence to be supplied to the model.

4.2.1 **Model Architecture**

This work uses a Seq2Seq model with an *encoder* and a *decoder* as shown Figure 4.5. The *encoder* can be either a unidirectional RNN or a bidirectional RNN as shown in Figures 4.6 and 4.7 respectively. However, the *decoder* is always a unidirectional RNN with attention which either uses *teacher forcing* or *sampling* during training as shown in Figures 4.8 and 4.9 respectively.

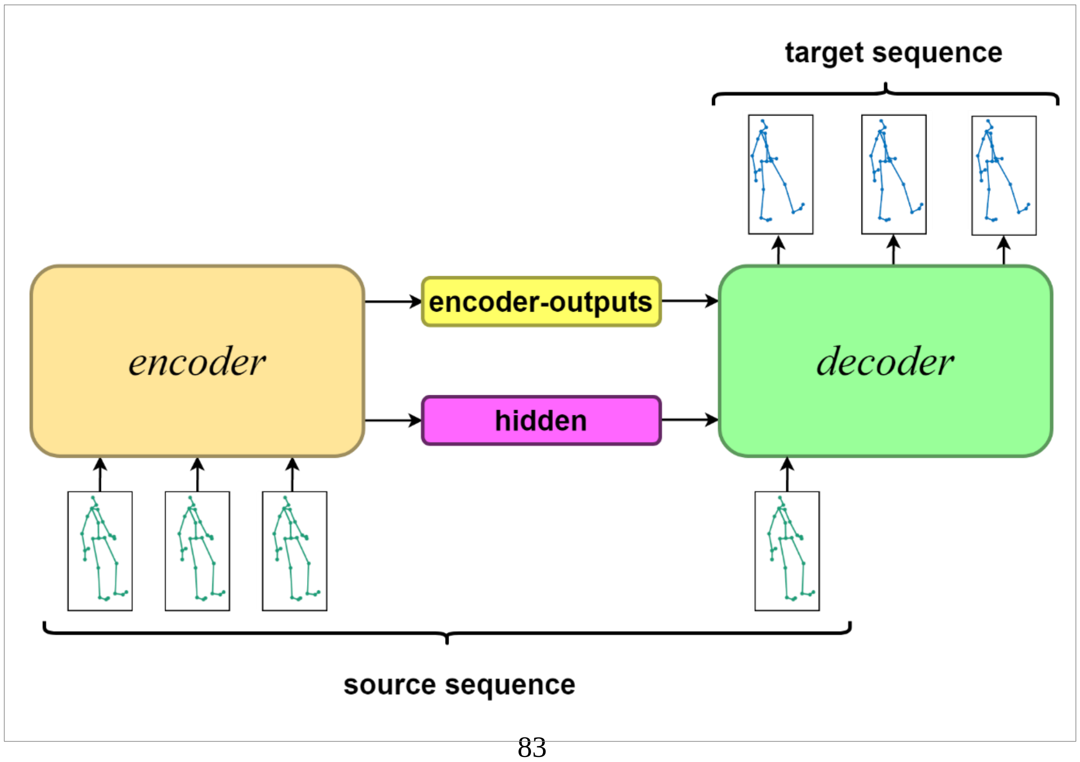


Figure 4.5: The Seq2Seq architecture used in this work.

At every iteration, the Seq2Seq model starts training by supplying the *encoder* with the *source sequence* (*encoder-inputs)* which is shown as green skeletons in Figure 4.5. At every step of encoding, the *encoder* will consume the provided sequence frame by frame and produce a hidden state (activation) at each step. Upon finishing the encoding of the *source sequence*, the *encoder* provides two outputs; the *encoder-outputs* which comprises the hidden states of all steps and the *hidden* which is the hidden state of the last step. Since this Seq2Seq model uses an attention *decoder*, both *hidden* and *encoder-outputs* are needed. However, when a classic *decoder* is used (one that doesn't employ any form of attention), only *hidden* is needed for decoding hence *encoder-outputs* is discarded.

At first step of decoding, the attention *decoder* takes 3 input sequences namely *encoder-outputs*, *hidden* and the last frame of the *source sequence* (serves as *decoder-inputs*). The *hidden* is used to initialize the hidden state of the *decoder* thus, the *decoder* is conditioned on the *source sequence* since *hidden* is supposed to encode the whole *source sequence*. In the subsequent steps of decoding, the hidden state of the decoder is updated, and *decoder-inputs* becomes either the previous prediction (*sampling*) or the ground-truth of the previous output (*teacher forcing*). The role of *encoder-outputs* is to obtain *attention weights* which will be illustrated in detail later.

4.2.2 **Encoder**

As mentioned earlier, this work has experimented with a unidirectional *encoder* and its bidirectional counterpart to assess their performance quantitatively. Both *encoder's* architectures used in this work have a single GRU layer and both will be explored in this section. Figure 4.6 illustrates a unidirectional *encoder*, its inputs (*encoder-inputs*), and its outputs (*encoder-outputs* and *hidden*). The dimensions of inputs and outputs are shown in *time major* format where *S* is the length of source sequence, *B* is the batch size and *H* is he hidden size. Since the unidirectional *encoder* processes the *source sequence* in one direction, it produces a single hidden state for the last step (*hidden*) and a single hidden state for every step (*encoder-outputs*).

However, the *hidden* of the bidirectional *encoder* shown in Figure 4.7 contains 2 hidden states; the last step's hidden state of the forward RNN (blue blocks in Figure 4.7), and the first step's hidden state of the backward RNN (purple blocks in Figure 4.7).

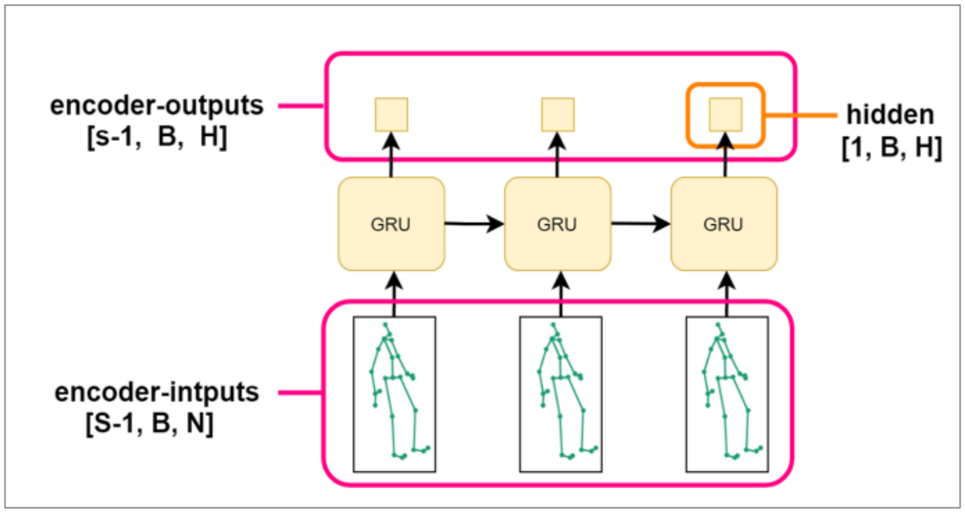


Figure 4.6: A unidirectional *encoder*.
S = length of source sequence, B = batch size, H = hidden size, and N = input size (No. of features).

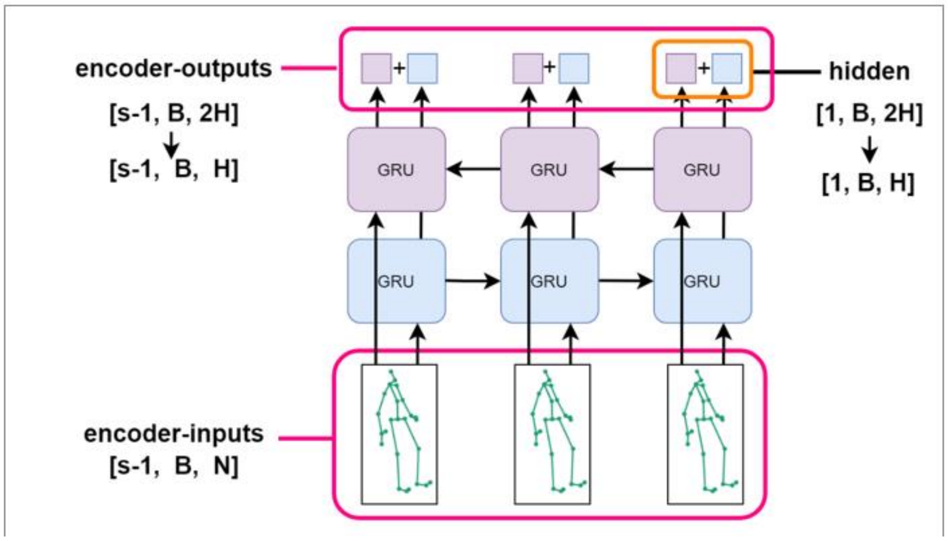


Figure 4.7: A bidirectional *encoder*.
S = length of source sequence, B = batch size, H = hidden size, and N = input size (No. of features).

The forward RNN processes the source sequence in the original order and the backward RNN processes the sequence in the reversed order. Ultimately, forward and backward hidden states (blue and purple squares) are combined (added) and the last dimension of *hidden* is reduced consequently to become like that of the unidirectional *encoder*. The same is true for *encoder-outputs* produced by the bidirectional *encoder*. Reducing the dimensions of the bidirectional *encoder's* outputs is crucial since the unidirectional *decoder* expects the dimensions of *hidden* and *encoder-outputs* to be like those of the unidirectional *encoder*.

4.2.3 **Decoder**

The decoder used in the Seq2Seq model of this work is a unidirectional attention *decoder*. Figures 4.8 and 4.9 show the same attention *decoder*. The blue diamond represents the steps of attention which will be illustrated later whereas the red trapezoid represents a projection layer that projects the output of the GRU cell from hidden size back to input size. Figure 4.8 shows the attention *decoder* when trained with *teacher forcing* approach for a single iteration where the *decoder* is fed the ground-truth (green skeleton) at every decoding step. Figure 4.9 shows the same attention *decoder* when trained with *sampling* approach for a single iteration where the *decoder* is fed the ground-truth (green skeleton) at the first step of decoding only and is fed its own predictions (blue skeleton) at the remaining steps.

Pure *teacher forcing* is when the attention *decoder* sees ground-truth in all training iterations whereas pure *sampling* is when the attention *decoder* sees its own predictions in all training iterations. *Scheduled sampling* is when the *decoder* sees the ground-truth in some iterations and its own predictions in others. It is possible for the *decoder* that uses *scheduled sampling* to have different probabilities for *teacher forcing* and *sampling* e.g. 60% of *teacher forcing* and 40% of *sampling*.

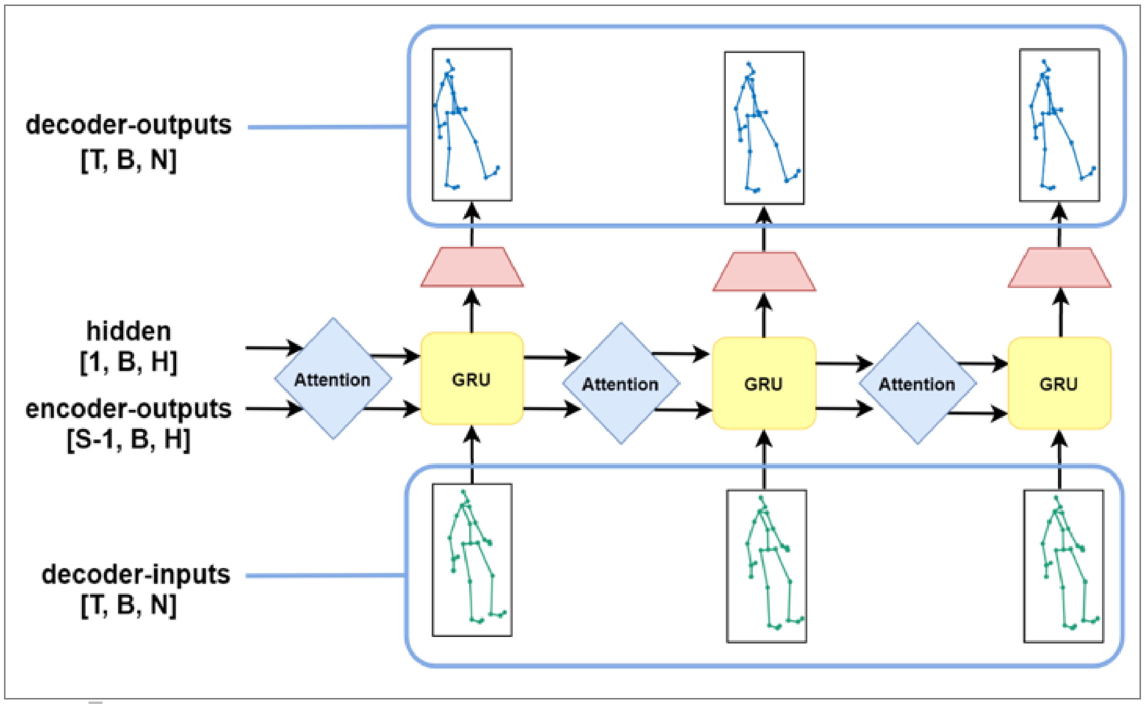


Figure 4.8: Attention *decoder* with *sampling*.
T = length of target sequence, B = batch size, H = hidden size, and N = input size (No. of features).

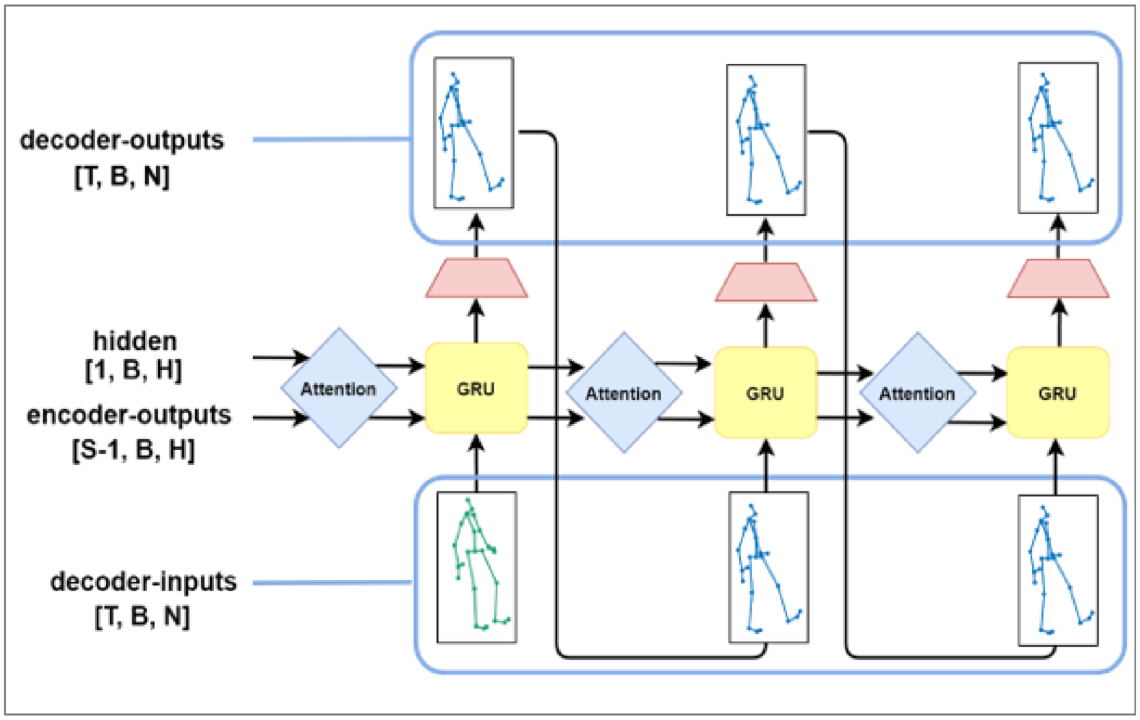


Figure 4.9: Attention *decoder* with *sampling*.
T = length of target sequence, B = batch size, H = hidden size, and N = input size (No. of features).

4.2.4 Model Implementation

4.2.4.1 High-level overview of the implementation

The data preprocessing part is mostly adopted from Martinez et al., (2017) whereas the attention implementation is adopted from (Robertson, 2019). Below are the steps of the human motion prediction task as implemented in this work.

1. **Hyperparameters initialization**

Since the Seq2Seq model requires a predefined set of hyperparameters, the following variables are defined upfront creating the model:

- **Length of the source sequence**: number of frames to feed into the *encoder*.
- **Length of the target sequence**: number of frames that the *decoder* must predict.
- **Size of the hidden layer**: number of hidden units for each layer.
- **Number of hidden layers**.
- **Maximum L-2 norm:** threshold after which the gradient is clipped.
- **Batch size.**
- **Learning rate.**
- **Learning decay factor:** multiplicative factor to decay the learning rate.
- **Learning rate decay step:** number of steps after which to decay the learning rate.
- **Teacher forcing ratio:** 0 means the decoder uses pure *sampling* for training, 0.5 means it uses *scheduled sampling* and 1 means it uses pure *teacher forcing*.
- **Number of iterations:** number of training iterations.
- **Evaluate step:** number of iterations to after which evaluation takes place.

Usually, it is useful to reduce the learning rate as the training progresses. This is known as learning rate decay or scheduling. The intuition behind this idea is that as the training progresses, the model optimizes its parameters and becomes closer to the optimal solution thus reducing the learning rate prevents overshooting the minima and helps the model to converge. The decay takes place every (**Learning rate decay step**) according to the following equation:

$$\alpha = \grave{\alpha}\gamma \tag{4.1}$$

Where:

α : new learning rate

$\grave{\alpha}$: old learning rate

γ : learning rate decay factor

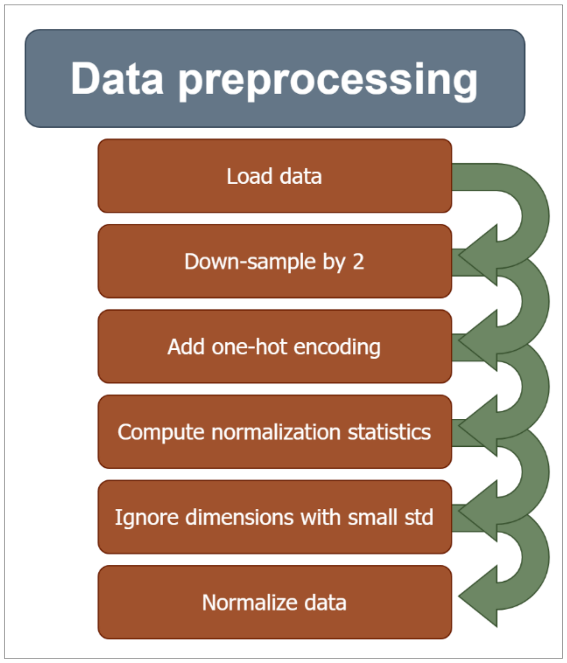


Figure 4.10: Data preprocessing stages.

2. Data preprocessing

The same preprocessing is applied to data during training and testing. The steps of the preprocessing are highlighted in Figure 4.10. These steps are presented in more detail below.

- **Load data:** Convert data from (CSV) format to float Numpy array.
- **Down-sample data by 2:** Take even rows and discard odd rows.
- **Add one-hot encoding:** Append the one-hot encoding to the data.

- **Compute normalization statistics:** Find the mean and the standard deviation (std) of the whole dataset.
- **Ignore dimensions with very small std:** Columns with $std < 1e - 4$ will not be used in the training since they don't provide helpful information.
- **Normalize data:** Subtract the mean and divide by the std.

3. Extract encoder-outputs of SRNN seeds

The same ground truth sequences that were used for evaluation and testing by Jain et al., (2016) in their proposed architecture, that is the structural recurrent neural network (SRNN), are un-normalized and converted to Euler angles to be used in evaluation and testing later. This will ensure that the results are comparable to previous researches. i.e. this stage will reproduce the same SRNN's seed sequences (the ground truth of the expected output sequences) taken from the testing set, i.e. subject 5. As mentioned before, subject 5 was used in previous works for testing. This stage constitutes of the following steps:

a) **Extract SRNN seed sequences:**

For each action of subject 5, the total number of seed sequences is 8, in which 4 sequences are taken from sub-action 1 and the other 4 are taken from sub-action 2. Figure 4.11 shows how the seed sequences are extracted from any sub-action of subject 5. Each seed sequence is then assigned to *encoder-inputs*, *decoder-inputs* and *decoder-outputs* as shown previously in Figure 4.4. At this point, only *decoder-outputs* are needed as they will be used in evaluation later to be compared against the model predictions.

b) **Un-normalize SRNN** *decoder-outputs*

The *decoder-outputs* will be un-normalized so that it can be used in evaluation.
Un-normalization recover original data from the normalized data as follows:

- Recover the ignored dimensions.
- Multiply by the standard deviation of motion data
- Add the mean of motion data

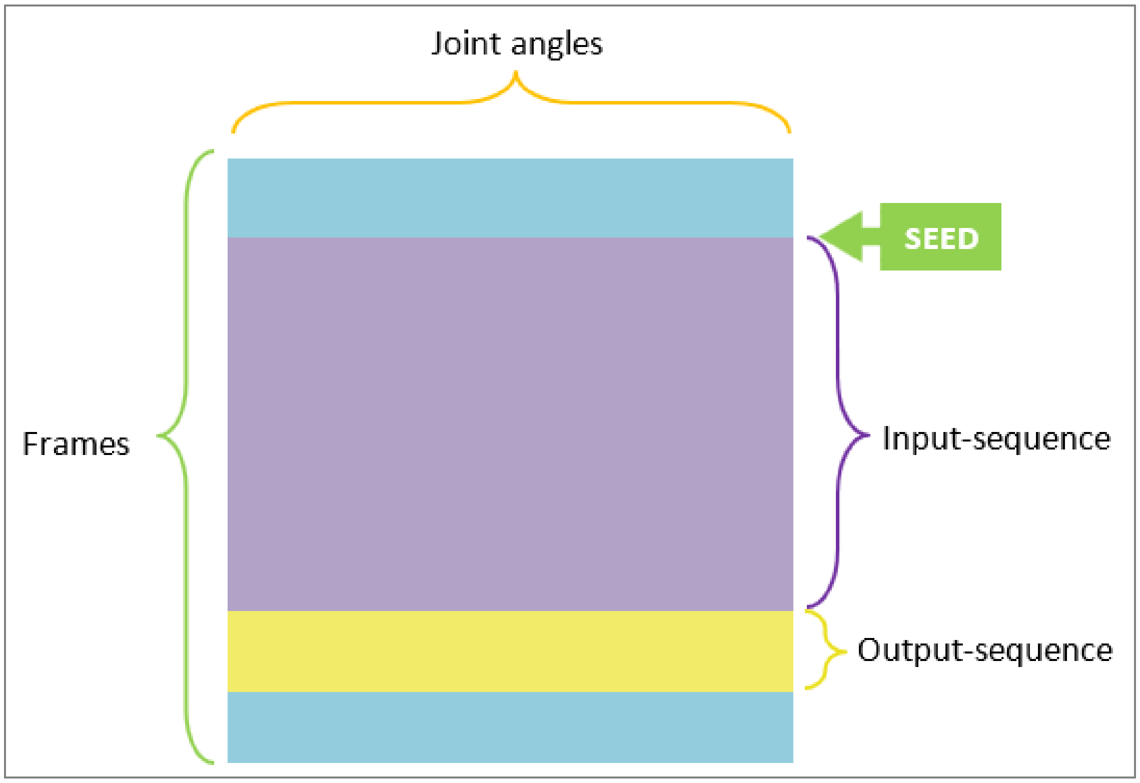


Figure 4.11: The extraction of SRNN sequences.
The blue block is any sub-action of subject 5. The purple block starts at frame number = SEED and ends at frame number = SEED + (input sequence length). The yellow block starts at frame number = SEED + (input sequence length) + 1 and ends at frame number = SEED + (input sequence length) + 1 + (output sequence).

c) **Convert SRNN *decoder-outputs* to Euler angle representation**

SRNN *decoder-outputs* are converted from exponential maps to Euler angles.

4. Start the training loop

Before starting the training loop, the following steps are executed:

1. Get a random batch from the training set consisting of *encoder-inputs*, *decoder-inputs* and *decoder-outputs*. These sequences follow the same pattern shown in Figure 4.5. The dimensionality of each sequence is given below:

encoder-inputs =

```
(batch size, input sequence length – 1, number of normalized of joint angles).
```

decoder-inputs =

`(batch size, output sequence length , number of normalized of joint angles).`

decoder-outputs =

`(batch size, output sequence length , number of normalized of joint angles).`

Figures 4.12 and 4.13 illustrate the dimensionality of the abovementioned sequences. The *decoder-outputs* is shifted one timestep ahead of the *decoder-inputs* as shown previously in Figure 4.11.

2. Feed the encoder with *encoder-inputs, decoder-inputs* and *decoder-outputs*.
3. Get *encoder-outputs* (hidden states of all steps) and *encoder-hidden* (hidden state of the last step).
4. If the random number < teacher forcing ratio:
    1. For every element in *target-sequence*:
        - ❖ Pass *encoder-outputs, decoder-hidden* and *decoder-inputs* (will be always the ground-truth since it uses teacher forcing).
        - ❖ Perform attention steps as shown in Figure 4.12
        - ❖ Get *prediction* and updated *decoder-hidden*
        - ❖ Add *prediction* to previous *predictions*

5. else:
    2. For every element in *target-sequence*:
        - ❖ Feed the decoder with *encoder-outputs, decoder-hidden* and *decoder-inputs* (it will be the ground-truth only at 1st steps but for other steps it will be the prediction since the decoder uses sampling).
        - ❖ Perform attention steps as shown in Figure 4.12
        - ❖ Get *prediction* and updated *decoder-hidden*
        - ❖ Add *prediction* to previous *predictions*

6. *step-loss* = (*predictions* – *decoder-outputs*)^2
7. *step-loss* = mean*(step-loss)*
8. Backward propagate *step-loss*
9. Perform optimizer step for encoder and decoder

10. Print the step loss every *k* steps.
11. Do learning decay every *l* steps.

If this is a validation step (*current step % m == 0*):

- Validate on random sequences:
    1. Get a random batch exactly as done previously under the training step but this time the batch is taken from **testing data** i.e. subject 5.
    2. Feed the encoder with *encoder-inputs, decoder-inputs* and *decoder-outputs*.
    3. Get *encoder-outputs* (hidden states of all steps) and *encoder-hidden* (hidden state of the last step).
    4. For every element in *target-sequence*:
        - ❖ Feed the decoder with *encoder-outputs, decoder-hidden* and *decoder-inputs* (it will be the ground-truth only at 1st steps but for other steps it will be the prediction since the decoder uses sampling).
        - ❖ Perform attention steps as shown in Figure 4.12
        - ❖ Get *prediction* and updated *decoder-hidden*
        - ❖ Add *prediction* to previous *predictions*
    5. *step-loss* = (*predictions* – *decoder-outputs*)^2
    6. *step-loss* = mean*(step-loss)*
    7. *validation-loss* = *step-loss*

- Validate on SRNN sequences:

  For every action:
    1. Get the 8 SRNN seed sequences as done previously in step 3 of the training phase, but this time *encoder-inputs, decoder-inputs* and *decoder-outputs* are needed.
    2. Feed the encoder with *encoder-inputs*, *decoder-inputs* and *decoder-outputs*.
    3. Get *encoder-outputs* (hidden states of all steps) and *encoder-hidden* (hidden state of the last step)
    4. For every element in *target-sequence*:

- ❖ Feed the decoder *encoder-outputs, decoder-hidden* and *decoder-inputs* (it will be the ground-truth only at 1st steps but for other steps it will be the prediction since the decoder uses sampling)
- ❖ Perform attention steps as shown in Figure 4.14
- ❖ Get *prediction* and updated *decoder-hidden*
- ❖ Add *prediction* to previous *predictions*

5. *step-loss* = (*predictions* – *decoder-outputs*)^2
6. *step-loss* = mean*(step-loss)*
7. *SRNN-loss* = *step-loss*
8. Return the 8 predictions of the model.
9. Un-normalize the predictions.
10. Convert the predictions from exponential maps to Euler angles.
11. Obtain the mean-squared-error between the ground truth and the predictions as shown in Figure 4.15.
    - o For every sequence, compute the squared sum of errors for every frame (each sequence will have 1 value for every frame).
    - o Compute the mean of the squared sums for all the 8 sequences.

5. Print the results

The following information are printed to the screen:

For every *n* steps:

- Current step.
- Current step loss (training loss).

For every *m* steps:

- Global step.
- Learning rate.
- Step time in milliseconds (total steps time / number of steps).
- Average training loss(training loss / number of steps).
- SRNN loss.

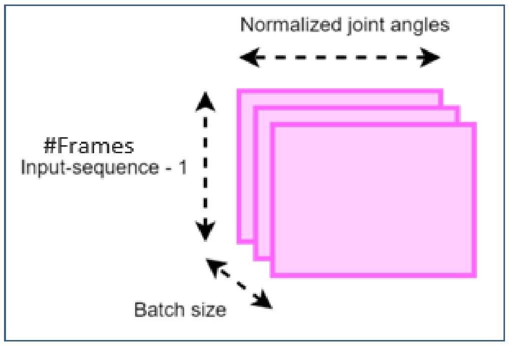


Figure 4.12: ***encoder-inputs*** **dimensions**

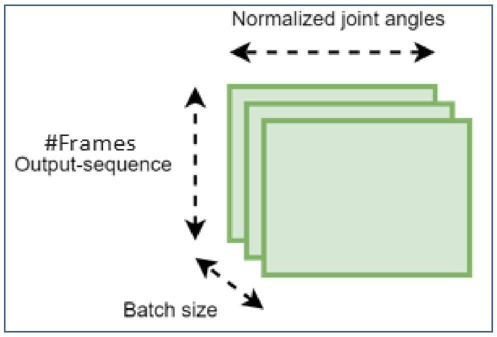


Figure 4.13: ***decoder-inputs*** **and** ***decoder-outputs*** **dimensions.**

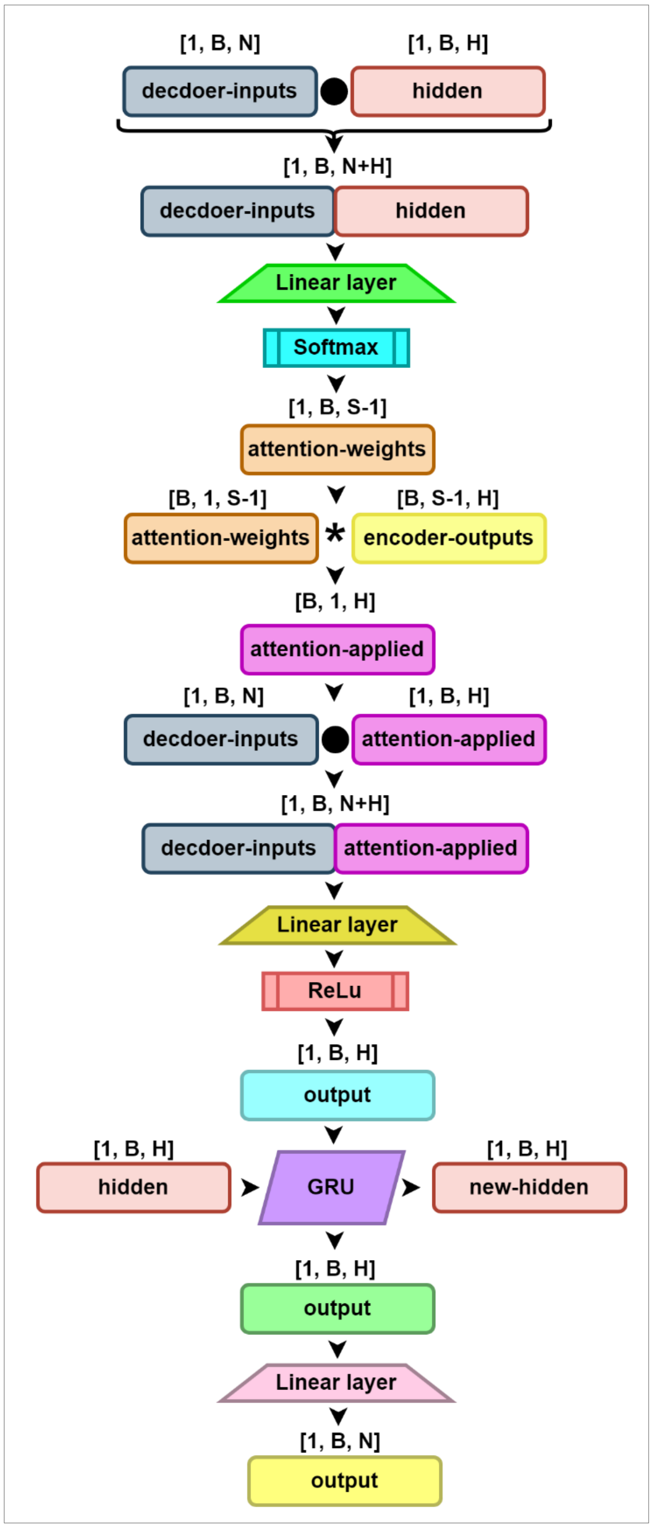


Figure 4.14: Attention steps.

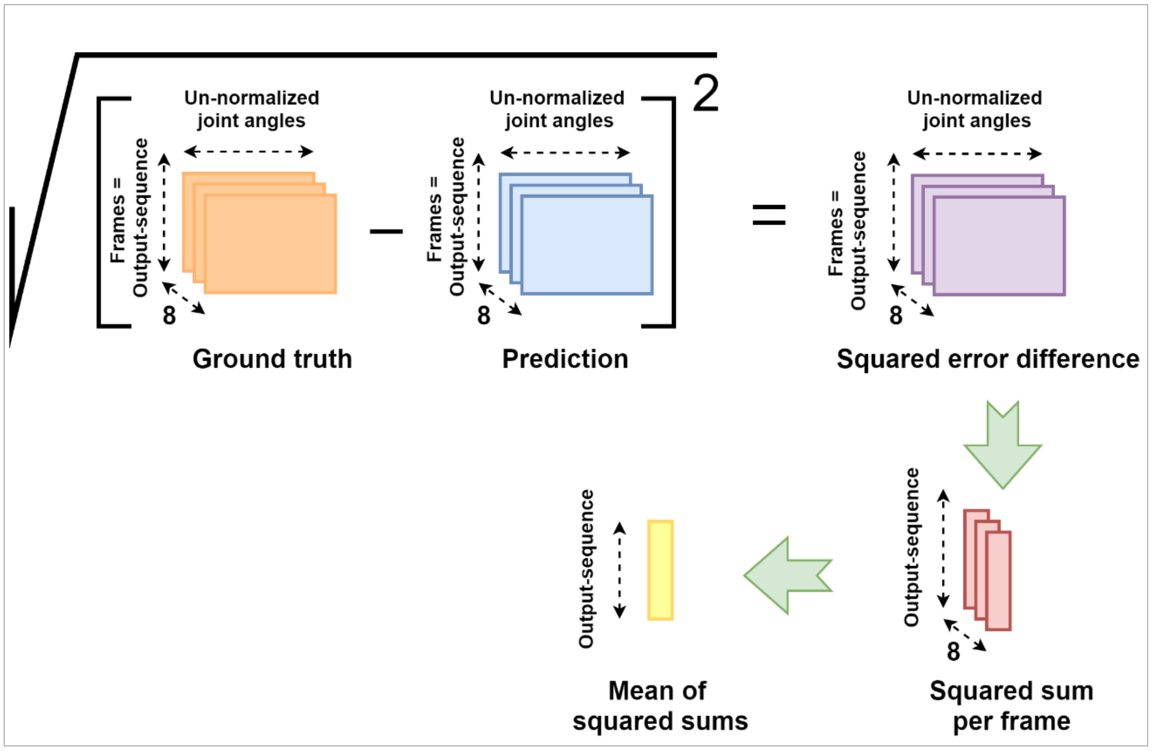


Figure 4.15: Finding the error between the ground truth and the prediction for a single action.

Chapter 5

EXPERIMENTS AND RESULTS

5.1 Experimental Setup

5.1.1 Environmental Specifications of the Experiment

a) Hardware

The experiments were conducted using Google Collaboratory, a research project of Google based on Jupyter notebook environment that's connected to a cloud-base runtime with a Tesla GPU. Table 4 shows environmental specifications of the experiment.

Hardware	Specifications
GPU	1xTesla K80, compute 3.7, having 2496 CUDA cores, 12GB GDDR5 VRAM
CPU	2xsingle core hyper threaded Xeon Processors @2.3Ghz
RAM	~12.6 GB Available
Disk	~33 GB Available

Table 4: Hardware specifications.

b) Software

Table 5 presents software specifications of the experimentations.

Software	Description	Version
Python	General-purpose programming language	3.6.9
PyTorch (Paszke, et al., 2019)	Deep learning framework	1.4.0

Table 5: Software specifications.

5.1.2 **Experimentations of this Work**

This work explores quantitatively the performance of the following 4 variations of the Seq2Seq model:

- **Uni-Enc-50:** Unidirectional encoder and attention decoder with 50% teacher forcing (the decoder sees ground-truth 50% of the time – *scheduled sampling*).
- **Bi-Enc-50:** Bidirectional encoder and attention decoder with 50% teacher forcing (the decoder sees ground-truth 50% of the time – *scheduled sampling*).
- **Uni-Enc-0:** Unidirectional encoder and attention decoder with 0% teacher forcing (the decoder sees its own predictions all the time – *sampling*).
- **Uni-Enc-100:** Unidirectional encoder and attention decoder with 100% teacher forcing (the decoder sees ground-truth all the time – *teacher forcing*).

One of this work's models namely **Bi-Enc-50**, uses a bidirectional *encoder* while other models use unidirectional encoders. However, common to all models is the architecture of the attention *decoder.* The abovementioned models differ in the way the *decoder* is trained. The number appended to the name of each model signifies the probability that the *decoder* is fed ground-truth or its own predictions during training. During training, the *decoder* in **Uni-Enc-50** and **Bi-Enc-50** models is fed the ground-truth 50% of the time and its own predictions 50% of the time. This is approach is known as *scheduled sampling*. For **Uni-Enc-0** model, the *decoder* is never fed the ground-truth, instead, it is fed its own predictions all the time which is a pure *sampling* approach. However, the decoder of **Uni-Enc-100 model** is fed the ground-truth all the time i.e. it is trained through *teacher forcing* approach. The motivation behind changing the percentages is to test the impact of different approaches of training including *teacher forcing, scheduled sampling,* and *sampling* on the performance. According to Martinez et al., (2017), training the decoder with pure *teacher forcing* may lead a network that is unable to recover from its own mistakes. Surprisingly, the findings of the experimentations done by this work are neutral and don't signify any difference between these approaches.

a) Hyperparameters

For all models, the hyperparameters used in the experimentations are given in Table 6. These are supplied by the user.

b) Architecture and algorithm implementation details

All experiments are conducted using a Seq2Seq architecture with a single GRU layer for the *encoder* and the *decoder*. The optimization algorithm is the Stochastic gradient descent (SGD) and the cost function is the Mean squared error (MSE). During training, the

Hyperparameter	Value
Learning rate	0.005
Learning rate decay factor	0.95
Number of steps after which the learning rate is decayed	10.000
Maximum L-2 norm after which the gradient is clipped	5
Batch size	16
Number of iterations	100.000
Number of hidden units	1024
Number of layers	1
Length of the input sequence - number of frames to be fed to the encoder	50
Length of the output sequence – number of frames to be predicted by the decoder	25

Table 6: Model's hyperparameters.

Architecture	Seq2Seq
Encoder	Unidirectional or bidirectional RNN
Decoder	Unidirectional RNN with attention
RNN cell	Gated recurrent unit (GRU)
Additional techniques	Residual connections
Cost function	Mean squared error (MSE)
Optimization algorithm	Stochastic gradient descent (SGD)

Table 7: Model's architectural and algorithmic details.

5.1.3 **Results**

Following previous works, mean errors are shown in Euler angles after 80ms, 160ms, 320ms and 400ms of motion. i.e. after 2, 4, 8 and 10 frames since the frame rate is 25 fps. Furthermore, the errors of this works models are also shown after 560ms and 1000ms i.e. after 14 and 25 frames respectively. The models of this work are compared against the following models:

- **ERD:** Encoder-Recurrent-Decoder by Fragkiadaki et al., (2015).
- **LSTM-3LR:** 3-layer long short-term memory network by Fragkiadaki et al., (2015).
- **SRNN:** Structural recurrent neural network by Jain et al., (2016).
- **Res. sup.**: Residual decoder with a unidirectional encoder by (Martinez et al., 2017).
- **AGED w/ adv+geo**: Adversarial geometry-aware encoder-decoder with frame-wise geodesic loss by Gui wt al., (2018).

Tables 8 and 9 show errors of all abovementioned models on walking, eating, smoking, and discussion actions. Tables 10, 11, 12, 13, 14, and 15 show errors for last two models since earlier models have not been tested on the remaining actions. The lowest error is shown in bold and the second lowest error is shown underlined. Table 15 shows average errors on all actions.

	Walking						Eating					
milliseconds	80	160	320	400	560	1000	80	160	320	400	560	1000
ERD	0.93	1.18	1.59	1.78	-	-	1.27	1.45	1.66	1.80	-	-
LSTM-3LR	0.77	1.00	1.29	1.47	-	-	0.89	1.09	1.35	1.46	-	-
SRNN	0.81	0.94	1.16	1.30	-	-	0.97	1.14	1.35	1.46	-	-
Res. sup.	0.27	0.47	0.67	0.73	-	-	0.23	0.39	0.62	0.78	-	-
AGED w/ adv+geo	**0.22**	0.36	0.55	0.67	**-**	**-**	**0.17**	0.28	0.51	0.64	-	-
Uni-Enc-50	**0.22**	**0.21**	0.24	0.24	0.22	0.24	0.25	0.23	0.20	0.21	0.20	0.25
Bi-Enc-50	**0.22**	**0.21**	**0.23**	0.24	0.23	0.24	0.19	0.20	0.20	**0.19**	0.19	0.23
Uni-Enc-0	**0.22**	0.22	0.25	0.26	0.25	0.25	0.22	0.21	0.20	0.20	0.19	0.25
Uni-Enc-100	0.23	**0.21**	**0.23**	**0.23**	0.23	0.23	0.21	**0.19**	**0.18**	**0.19**	0.18	0.24

Table 8: Mean squared errors in Euler angles of discussion and smoking actions.

	Smoking						Discussion					
milliseconds	80	160	320	400	560	1000	80	160	320	400	560	1000
ERD	1.66	1.95	2.35	2.42	-	-	2.27	2.47	2.68	2.76	-	-
LSTM-3LR	1.34	1.65	2.04	2.16	-	-	1.88	2.12	2.25	2.23	-	-
SRNN	1.45	1.68	1.94	2.08	-	-	1.22	1.49	1.83	1.93	-	-
Res. sup.	0.32	0.59	0.99	1.09	-	-	0.33	0.61	1.05	1.15	-	-
AGED w/ adv+geo	0.27	0.43	0.82	0.84	**-**	**-**	0.27	0.56	0.76	0.83	-	-
Uni-Enc-50	0.33	0.34	0.40	0.30	0.30	0.36	**0.23**	**0.29**	**0.24**	0.25	0.24	0.23
Bi-Enc-50	**0.29**	**0.29**	0.38	0.26	0.24	0.35	0.25	0.31	0.25	0.26	0.25	0.23
Uni-Enc-0	0.28	0.32	**0.38**	0.28	0.27	0.36	0.29	0.33	0.29	0.29	0.24	0.22
Uni-Enc-100	0.31	0.34	**0.41**	**0.30**	0.27	0.35	0.27	**0.27**	**0.24**	**0.24**	0.25	0.24

Table 9: Mean squared errors in Euler angles of discussion and smoking actions.

	Directions						Greeting					
milliseconds	80	160	320	400	560	1000	80	160	320	400	560	1000
Res. sup.	0.26	0.47	0.72	0.84	-	-	0.75	1.17	1.74	1.83	-	-
AGED w/ adv+geo	**0.23**	0.39	0.63	0.69	**-**	**-**	0.56	0.81	1.30	1.46	-	-
Uni-Enc-50	0.25	0.23	0.23	0.20	0.21	0.21	**0.28**	0.27	0.29	0.30	0.40	0.33
Bi-Enc-50	0.25	**0.22**	0.24	0.20	0.20	0.22	0.31	**0.25**	**0.25**	0.30	0.40	0.33
Uni-Enc-0	0.24	0.23	**0.22**	**0.19**	0.21	0.20	0.31	0.26	0.27	0.29	0.40	0.32
Uni-Enc-100	0.24	0.23	**0.22**	**0.19**	0.22	0.22	0.31	0.26	0.27	**0.28**	0.36	0.31

Table 10: Mean squared errors in Euler angles of directions and greeting actions.

	Phoning						Posing					
milliseconds	80	160	320	400	560	1000	80	160	320	400	560	1000
Res. sup.	0.23	0.43	0.69	0.82	-	-	0.36	0.71	1.22	1.48	-	-
AGED w/ adv+geo	**0.19**	**0.34**	**0.50**	**0.68**	**-**	**-**	**0.31**	0.58	1.12	1.34	-	-
Uni-Enc-50	0.52	0.42	0.86	0.86	0.41	0.30	0.42	0.37	0.36	0.33	0.44	0.35
Bi-Enc-50	0.54	0.48	0.88	0.90	0.44	0.32	0.40	**0.34**	0.31	0.32	0.37	0.36
Uni-Enc-0	0.53	0.47	0.87	0.88	0.43	0.32	0.38	0.35	0.33	0.34	0.39	0.37
Uni-Enc-100	0.58	0.48	0.87	0.88	0.42	0.30	0.40	0.36	**0.30**	**0.30**	0.40	0.36

Table 11: Mean squared errors in Euler angles of phoning and posing actions.

	Purchases						Sitting					
milliseconds	80	160	320	400	560	1000	80	160	320	400	560	1000
Res. sup.	0.51	0.97	1.07	1.16	-	-	**0.41**	1.05	1.49	1.63	-	-
AGED w/ adv+geo	0.46	0.78	1.01	1.07	**-**	**-**	**0.41**	0.76	1.05	1.19	-	-
Uni-Enc-50	0.42	0.38	0.34	0.34	0.37	0.41	0.49	0.45	0.48	0.44	0.46	0.44
Bi-Enc-50	0.39	0.33	0.32	0.29	0.30	0.36	0.42	0.45	0.44	0.43	0.45	0.41
Uni-Enc-0	0.38	**0.32**	0.30	0.28	0.32	0.37	0.43	**0.40**	**0.42**	**0.41**	0.40	0.40
Uni-Enc-100	**0.36**	**0.32**	**0.28**	**0.28**	0.31	0.37	0.46	0.43	0.44	0.43	0.43	0.39

Table 12: Mean squared errors in Euler angles of purchases and sitting actions.

	Sitting down						Taking photo					
milliseconds	80	160	320	400	560	1000	80	160	320	400	560	1000
Res. sup.	0.39	0.81	1.40	1.62	-	-	0.24	0.51	0.90	1.05	-	-
AGED w/ adv+geo	**0.33**	0.62	0.98	1.10	**-**	**-**	**0.23**	0.48	0.81	0.95	-	-
Uni-Enc-50	0.55	**0.60**	0.53	0.49	0.50	0.53	0.24	0.28	0.26	0.25	0.22	0.24
Bi-Enc-50	0.56	0.61	0.55	0.50	0.52	0.56	0.24	**0.26**	**0.23**	**0.22**	0.22	0.22
Uni-Enc-0	0.61	0.63	**0.51**	**0.48**	0.50	0.56	0.24	0.30	0.25	0.24	0.22	0.22
Uni-Enc-100	0.65	0.67	0.55	0.52	0.50	0.55	0.26	0.27	0.24	0.23	0.24	0.22

Table 13: Mean squared errors in Euler angles of sitting down and taking photo actions.

	Waiting						Walking dog					
milliseconds	80	160	320	400	560	1000	80	160	320	400	560	1000
Res. sup.	0.28	0.53	1.02	1.14	-	-	0.56	0.91	1.26	1.40	-	-
AGED w/ adv+geo	**0.24**	0.50	1.02	1.13	**-**	**-**	0.50	0.81	1.15	1.27	-	-
Uni-Enc-50	0.32	0.33	0.30	0.29	0.38	0.30	<u>0.37</u>	<u>0.42</u>	0.38	0.33	0.34	0.35
Bi-Enc-50	0.29	**0.28**	<u>0.27</u>	<u>0.27</u>	0.37	0.28	0.38	**0.38**	**0.32**	**0.31**	0.32	0.37
Uni-Enc-0	<u>0.27</u>	<u>0.29</u>	**0.26**	**0.26**	0.36	0.29	**0.36**	0.43	<u>0.35</u>	<u>0.32</u>	0.35	0.33
Uni-Enc-100	0.30	0.30	<u>0.27</u>	0.28	0.36	0.29	<u>0.37</u>	0.43	0.36	0.33	0.34	0.34

Table 14: Mean squared errors in Euler angles of waiting and walking dog actions.

	Walking together						Average					
milliseconds	80	160	320	400	560	1000	80	160	320	400	560	1000
Res. sup.	0.31	0.58	0.87	0.91	-	-	0.36	0.67	1.02	1.15	-	-
AGED w/ adv+geo	<u>0.23</u>	0.41	0.56	0.62	**-**	**-**	**0.31**	0.54	0.85	0.97	-	-
Uni-Enc-50	0.24	0.23	<u>0.22</u>	<u>0.22</u>	0.24	0.28	0.34	0.34	<u>0.36</u>	<u>0.34</u>	-	-
Bi-Enc-50	**0.22**	**0.21**	<u>0.22</u>	<u>0.22</u>	0.24	0.28	<u>0.33</u>	**0.32**	**0.34**	**0.33**	-	-
Uni-Enc-0	**0.22**	**0.21**	<u>0.22</u>	<u>0.22</u>	0.23	0.28	<u>0.33</u>	<u>0.33</u>	**0.34**	**0.33**	-	-
Uni-Enc-100	**0.22**	**0.21**	**0.20**	**0.21**	0.23	0.26	0.34	<u>0.33</u>	**0.34**	**0.33**	-	-

Table 15: Mean squared errors in Euler angles of walking together action and the average mean square errors for all actions.

5.1.4 **Discussion**

This work implemented 4 models namely **Uni-Enc-50**, **Bi-Enc-50**, **Uni-Enc-0**, and **Uni-Enc-100**.

All models use the same attention decoder but with varying probabilities of *teacher forcing* i.e. how much the decoder sees its own predictions as opposed to seeing ground-truths during training. Specifically, **Uni-Enc-50** and **Bi-Enc-50** use attention decoders that see ground-truth 50% of the time during training. **Uni-Enc-0** and **Uni-Enc-100** use attention decoders that see ground-truth 0% and 100% of the time during training respectively. Moreover, the work has experimented with two models, **Uni-Enc-50** and **Bi-Enc-50**, that use different versions of encoders, that is unidirectional and bidirectional respectively.

As it is evident from the results, errors after 80ms of this work's models are comparable to those of **Res. sup.** and **AGED w/ adv+geo**. However, as the time of prediction increases, the errors of earlier models deteriorate progressively unlike models of this work which show very stable prediction performance. The models used by this work have beaten earlier models on all actions except phoning where earlier models have lower prediction errors as shown in Table 11. Nevertheless, the errors produced by this work's models on phoning action have reduced noticeably after 560ms and 1000ms. Furthermore, this work's models' performance on all actions stays high even after 560ms and 1000ms, which is a strong indicator of the suitability of attention decoders not only for motion prediction (<500ms), but also for motion generation (>500ms). On average, models of this work have beaten earlier models with a large margin as shown in Table 4. According to the experimentations of this work, changing the percentage of teacher forcing (the probability that the decoder sees ground-truth during training) has no real impact on the results. Moreover, the use of a bidirectional encoder has no added benefit over its unidirectional counterpart. On the contrary of [2] and [5], residual connections and Quaternions have not improved the results but rather make them worse.

5.1.5 Conclusion

Learning a model that can generalize to different categories of human motion is a super-intensive task since human motion is highly variable and complex by nature. Nonetheless, many researchers are approaching the task relentlessly. Recent works have achieved good results on the task using deep learning methods. However, at the time of writing, the general model that can learn all kinds of human motion is still beyond the reach. For the first time, this work employs a bidirectional encoder to model human motion with an attention decoder to make predictions such that both are trained jointly on 15 different actions from the Human3.6 dataset. Prediction results after 80ms are comparable to those of previous works, however, after 160ms, models used by this work have much lower errors on all periodic and non-periodic actions except phoning. Even though, errors after 560ms and 1000ms on phoning action have reduced significantly and become less than those of previous works. According the findings of this work, the use of attention decoder has achieved state-of-the-art of human motion prediction after 160ms of motion prediction with very stable performance that doesn't deteriorate even after 1000ms of motion prediction which is not the case with earlier works. However, using a bidirectional encoder has no advantage over its unidirectional counterpart. Moreover, varying the percentages of teacher forcing i.e. training the decoder with 100%, 50%, or 0% of teacher forcing has no effect either.

5.1.6 **Future work**

Future work may include validating the predicted motion qualitatively, experimenting with different forms of attention, comparing the performance of LSTM against GRU, and using larger datasets of human motion.

REFRENCES

72.b Documentation with motion capture. (2012). Retrieved from https://pdfs.semanticscholar.org/fc18/0849a4171814aa7f9a0366305d791071227a.pdf

A numerical example of LSTMs. (2017, 6). Retrieved from https://statisticalinterference.wordpress.com/2017/06/01/lstms-in-even-more-excruciating-detail/

Amidi, S., & Amidi, A. (n.d.). Recurrent Neural Networks cheatsheet. Retrieved from https://stanford.edu/ shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks

Bahdanau, D., Cho, K., & Bengio, Y. (2015). Neural Machine Translation by Jointly Learning to Align and Translate. In Y. Bengio, & Y. LeCun (Ed.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings.* Retrieved from http://arxiv.org/abs/1409.0473

Bisht, H. S. (2018, 9). Effect of Bias in Neural Network. Retrieved from https://www.geeksforgeeks.org/effect-of-bias-in-neural-network/

Butepage, J., Black, M. J., Kragic, D., & Kjellstrom, H. (2017, 7). Deep Representation Learning for Human Motion Prediction and Classification. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* IEEE. doi:10.1109/cvpr.2017.173

Carnegie Mellon University - motion capture library. (n.d.). Retrieved from http://mocap.cs.cmu.edu/

Carnegie Mellon University - motion capture library - info. (n.d.). Retrieved from http://mocap.cs.cmu.edu/info.php

Chen, E. (2017, 5). Exploring LSTMs. Retrieved from http://blog.echen.me/2017/05/30/exploring-lstms/

Chen, G. (2016). A Gentle Tutorial of Recurrent Neural Network with Error Backpropagation. *CoRR, abs/1610.02583*. Retrieved from http://arxiv.org/abs/1610.02583

Cho, K., Merrienboer, B., Bahdanau, D., & Bengio, Y. (2014a). On the Properties of Neural Machine Translation: Encoder-Decoder Approaches. *CoRR, abs/1409.1259*. Retrieved from http://arxiv.org/abs/1409.1259

Cho, K., Merrienboer, B., Gülçehre, Ç., Bougares, F., Schwenk, H., & Bengio, Y. (2014b). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *CoRR, abs/1406.1078*. Retrieved from http://arxiv.org/abs/1406.1078

Chollet, F. (2017a). *Deep Learning with Python* (1st ed.). Greenwich, CT, USA: Manning Publications Co.

Chollet, F. (2017b, 9). The Keras Blog. Retrieved from https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html

Chung, J., Gülçehre, Ç., Cho, K., & Bengio, Y. (2014). Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *CoRR, abs/1412.3555*. Retrieved from http://arxiv.org/abs/1412.3555

Dean, C. J. (2016). *Manifold Learning Techniques for Editing Motion Capture Data.* Master's thesis, Victoria University of Wellington. Retrieved from https://pdfs.semanticscholar.org/9124/1839dbde7088ccf0e5ea5d86dada6293e34e.pdf

Drakos, G. (2019, 2). What is a Recurrent NNs and Gated Recurrent Unit (GRUS). Medium. Retrieved from https://medium.com/@george.drakos62/what-is-a-recurrent-nns-and-gated-recurrent-unit-grus-ea71d2a05a69

Du, H., Manns, M., Herrmann, E., & Fischer, K. (2016, 12). Joint Angle Data Representation for Data Driven Human Motion Synthesis. *Procedia CIRP, 41*, 746-751. doi:10.1016/j.procir.2015.12.096

Dyer, S., Martin, J.-P., & Zulauf, J. (1995). Motion Capture White Paper.

Fragkiadaki, K., Levine, S., Felsen, P., & Malik, J. (2015, 12). Recurrent Network Models for Human Dynamics. *2015 IEEE International Conference on Computer Vision (ICCV),* (pp. 4346-4354). doi:10.1109/ICCV.2015.494

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning.* MIT Press.

Google Colaboratory. (n.d.). Google. Retrieved from https://colab.research.google.com/notebooks/welcome.ipynb

Gradient Descent with Momentum. (2019, 9). Retrieved from https://kraj3.com.np/blog/2019/09/gradient-descent-with-momentum/

Graves, A. (2013). Generating Sequences With Recurrent Neural Networks. *CoRR, abs/1308.0850*. Retrieved from http://arxiv.org/abs/1308.0850

Gui, L.-Y., Wang, Y.-X., Liang, X., & Moura, J. M. (2018). Adversarial Geometry-Aware Human Motion Prediction. In V. Ferrari, M. Hebert, C. Sminchisescu, & Y. Weiss (Ed.), *Computer Vision -- ECCV 2018* (pp. 823-842). Cham: Springer International Publishing. Retrieved from https://www.amazon.com/Computer-Vision-Conference-September-Proceedings-ebook/dp/B07JB3HQ8Z?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=B07JB3HQ8Z

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep Residual Learning for Image Recognition. *CoRR, abs/1512.03385*. Retrieved from http://arxiv.org/abs/1512.03385

Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation, 9*, 1735-1780. doi:10.1162/neco.1997.9.8.1735

Ionescu, C., Li, F., & Sminchisescu, C. (2011, 11). Latent structured models for human pose estimation. *2011 International Conference on Computer Vision*, (pp. 2220-2227). doi:10.1109/ICCV.2011.6126500

Ionescu, C., Papava, D., Olaru, V., & Sminchisescu, C. (2014, 7). Human3.6M: Large Scale Datasets and Predictive Methods for 3D Human Sensing in Natural Environments. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 36*, 1325-1339. doi:10.1109/TPAMI.2013.248

Jain, A., Zamir, A. R., Savarese, S., & Saxena, A. (2016, 6). Structural-RNN: Deep Learning on Spatio-Temporal Graphs. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, (pp. 5308-5317). doi:10.1109/CVPR.2016.573

Khan, M. A., Arif, M., & Kamal, A. (2017, 2). Modeling and compression of motion capture data. *2017 Learning and Technology Conference (L T) - The MakerSpace: from Imagining to Making!*, (pp. 7-13). doi:10.1109/LT.2017.8088120

Khuong, B. (2019, 7). The Basics of Recurrent Neural Networks (RNNs). Towards AI. Retrieved from https://medium.com/towards-artificial-intelligence/whirlwind-tour-of-rnns-a11effb7808f

Komura, T., Habibie, I., Schwarz, J., & Holden, D. (2017). Data-Driven Character Animation Synthesis. In B. Müller, S. I. Wolf, G.-P. Brueggemann, Z. Deng, A. McIntosh, F. Miller, & W. S. Selbie (Eds.), *Handbook of Human Motion* (pp. 1-29). Cham: Springer International Publishing. doi:10.1007/978-3-319-30808-1_10-1

Lander, J. (1998). Working with motion capture file formats. Retrieved from http://www.darwin3d.com/gamedev/articles/col0198.pdf

Lang, K. J., & Hinton, G. E. (1988). *The development of the time-delay neural network architecture for speech recognition.* Carnegie-MellonUniversity. Retrieved from https://apps.dtic.mil/dtic/tr/fulltext/u2/a221540.pdf

Lin, T., Horne, B. G., Tiňo, P., & Giles, C. L. (1996). *Learning Long-Term Dependencies is Not as Difficult with NARX Recurrent Neural Networks.* USA: University of Maryland at College Park.

Lipton, Z. C. (2015). A Critical Review of Recurrent Neural Networks for Sequence Learning. *CoRR, abs/1506.00019*. Retrieved from http://arxiv.org/abs/1506.00019

Luong, M.-T., Pham, H., & Manning, C. D. (2015). Effective Approaches to Attention-based Neural Machine Translation. *CoRR, abs/1508.04025*. Retrieved from http://arxiv.org/abs/1508.04025

Martinez, J., Black, M. J., & Romero, J. (2017). On human motion prediction using recurrent neural networks. *CoRR, abs/1705.02445*. Retrieved from http://arxiv.org/abs/1705.02445

Mazur, M. (2017, 11). A Step by Step Backpropagation Example. Retrieved from https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

Mulder, W. D., Bethard, S., & Moens, M.-F. (2015). A survey on the application of recurrent neural networks to statistical language modeling. *Computer Speech & Language, 30,* 61-98. doi:https://doi.org/10.1016/j.csl.2014.09.005

Nguyen, M. (2019, 7). Illustrated Guide to LSTM's and GRU's: A step by step explanation. Towards Data Science. Retrieved from https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21

Olah, C. (2015, 9). Neural Networks, Types, and Functional Programming. Retrieved from https://colah.github.io/posts/2015-09-NN-Types-FP/

Olah, C. (2015, 8). Understanding LSTM Networks. Retrieved from https://colah.github.io/posts/2015-08-Understanding-LSTMs/#fnref1

Pascanu, R., Mikolov, T., & Bengio, Y. (2012). Understanding the exploding gradient problem. *CoRR, abs/1211.5063*. Retrieved from http://arxiv.org/abs/1211.5063

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., . . . Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. Alché-Buc, E. Fox, & R. Garnett (Eds.), *Advances in Neural Information Processing Systems 32* (pp. 8024-8035). Curran Associates, Inc. Retrieved from http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

Pavllo, D., Feichtenhofer, C., Auli, M., & Grangier, D. (2019). Modeling Human Motion with Quaternion-based Neural Networks. *CoRR, abs/1901.07677*. Retrieved from http://arxiv.org/abs/1901.07677

Pavllo, D., Grangier, D., & Auli, M. (2018). QuaterNet: A Quaternion-based Recurrent Model for Human Motion. *CoRR, abs/1805.06485*. Retrieved from http://arxiv.org/abs/1805.06485

Pons-Moll, G., Baak, A., Gall, J., Leal-Taixé, L., Müller, M., Seidel, H., & Rosenhahn, B. (2011, 11). Outdoor human motion capture using inverse kinematics and von mises-fisher sampling. *2011 International Conference on Computer Vision*, (pp. 1243-1250). doi:10.1109/ICCV.2011.6126375

Robertson, S. (2019, 9). NLP From Scratch: Translation with a Sequence to Sequence Network and Attention. Retrieved from https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html

Schuster, M., & Paliwal, K. K. (1997, 11). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing, 45*, 2673-2681. doi:10.1109/78.650093

Sigal, L., Balan, A. O., & Black, M. J. (2009, 8). HumanEva: Synchronized Video and Motion Capture Dataset and Baseline Algorithm for Evaluation of Articulated Human Motion. *International Journal of Computer Vision, 87*, 4-27. doi:10.1007/s11263-009-0273-6

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to Sequence Learning with Neural Networks. *CoRR, abs/1409.3215*. Retrieved from http://arxiv.org/abs/1409.3215

Tang, Y., Ma, L., Liu, W., & Zheng, W.-S. (2018). Long-Term Human Motion Prediction by Modeling Motion Context and Enhancing Motion Dynamic. *CoRR, abs/1805.02513*. Retrieved from http://arxiv.org/abs/1805.02513

Taylor, G. W., Hinton, G. E., & Roweis, S. T. (2007). Modeling Human Motion Using Binary Latent Variables. In B. Schölkopf, J. C. Platt, & T. Hoffman (Eds.), *Advances in Neural Information Processing Systems 19* (pp. 1345-1352). MIT Press. Retrieved from http://papers.nips.cc/paper/3078-modeling-human-motion-using-binary-latent-variables.pdf

Trask, A. (2019). *Grokking Deep Learning* (1st ed.). Greenwich, CT, USA: Manning Publications Co.

Uení, V., Brně, T. V., Multimédií, G. A., & Práce, D. (2012). Statistical Language Models Based on Neural Networks.

Wang, X., Chen, Q., & Wang, W. (2014). 3D Human Motion Editing and Synthesis: A Survey. *Computational and Mathematical Methods in Medicine, 2014*, 1-11. doi:10.1155/2014/104535

Wikipedia. (2019). Named-entity recognition --- Wikipedia, The Free Encyclopedia.

Xia, S., Gao, L., Lai, Y.-K., Yuan, M.-Z., & Chai, J. (2017, 5 01). A Survey on Human Performance Capture and Animation. *Journal of Computer Science and Technology, 32*, 536-554. doi:10.1007/s11390-017-1742-y

Zhang, A., Lipton, Z. C., Li, M., & Smola, A. J. (2019). *Dive into Deep Learning.*