ΔΗΜΟΚΡΙΤΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΡΑΚΗΣ
"ΤΜΗΜΑ ΠΟΛΙΤΙΚΩΝ ΜΗΧΑΝΙΚΩΝ"

Διατμηματικό Μεταπτυχιακό Πρόγραμμα Σπουδων
"Εφαρμοσμένα Μαθηματικά"

Ο υπογραφόμενος δηλώνω υπεύθυνα ότι η παρούσα διατριβή είναι αποκλειστικά δικό μου έργο και υπεβλήθη στο Δ.Π.Μ.Σ. για να καλύψει μέρος των προαπαιτούμενων για την απόκτηση του Μεταπτυχιακού τίτλου στην Οργάνωση και Διοίκηση Τεχνικών Συστημάτων. Δηλώνω υπεύθυνα ότι, κατά την εκπόνηση της διατριβής, σεβάσθηκα τους όρους της Ακαδημαϊκής Ακεραιότητας και της Ερευνητικής Ηθικής και απέφυγα κάθε πράξη που συνιστά λογοκλοπή. Γνωρίζω, τέλος, ότι η λογοκλοπή μπορεί να τιμωρηθεί με αφαίρεση του Μεταπτυχιακού μου Τίτλου Σπουδών.

Υπογραφή ΜΦ

Ξάνθη Νοέμβριος 2024

Abstract

The following thesis is an extensive study of various basic but also advanced neural networks concepts, with particular emphasis on their architecture, learning techniques and many real life applications. The thesis begins by introducing both Artificial Neural Networks (ANNs) and the basic learning paradigms, and discusses central architecture variants and their behavior across a range of tasks. Explores convolutional neural networks (CNN) and particularly CNN-associated structures of LeNet, AlexNet, VGG and ResNet.

Next, the thesis introduces recurrent neural networks (RNNs), explaining what a recurrent layer is and then describing Long Short-Term Memory (LSTM) / Gated Recurrent Units (GRUs). Following that, the frequent use of RNNs in natural language processing, machine translation and time series forecasting are also discussed where we find out its capability on sequential data.

Then, we moves on to unsupervised learning with auto encoders, explaining their encoder-decoder framework and where they are applied in data compression and noise reduction. It also gives a comprehensive manual for Generative Adversarial Networks (GANs) that examines the adversarial process between generator and discriminator models.

Finally, the thesis discusses a state-of-the-art neural network architecture: Transformer model that is developed to improve long short-term memory architecture for encoding complex sequential data with better representation. It breaks down the principles of transformers and their various architectural relating to training. Consequent to deep learning, this provides an organized context about neural networks and their history as well as what may lie beyond for neural networks.

Keywords : Neural Networks, Deep Learning, Artificial Intelligence

Περίληψη

Η παρούσα διατριβή αποτελεί μια εκτενή μελέτη διάφορων βασικών, αλλά και προχωρημένων εννοιών των νευρωνικών δικτύων, με ιδιαίτερη έμφαση στην αρχιτεκτονική τους, τις τεχνικές εκμάθησης και πολλές εφαρμογές στον πραγματικό κόσμο. Η διατριβή ξεκινά με την εισαγωγή των Τεχνητών Νευρωνικών Δικτύων (ΤΝΔ) και των βασικών παραδειγμάτων εκμάθησης, και συζητά τις κεντρικές παραλλαγές της αρχιτεκτονικής τους και τη συμπεριφορά τους σε ένα εύρος εργασιών. Εξετάζει τα συνελικτικά νευρωνικά δίκτυα(CNN) και, ιδιαίτερα, τις παραλλαγές που σχετίζονται με τα CNN, όπως τα LeNet, AlexNet, VGG καιResNet.

Στη συνέχεια, η διατριβή εισάγει τα αναδρομικά νευρωνικά δίκτυα (RNN), εξηγώντας τι είναι μια αναδρομική σχέση και περιγράφοντας στη συνέχεια τη Μνήμη Μακράς-Σύντομης Διάρκειας (LSTM) / Μονάδες Αναδρομικών Πυλών (GRU). Ακολουθώντας αυτό, συζητείται η συχνή χρήση των RNN σε επεξεργασία φυσικής γλώσσας, μηχανική μετάφραση και πρόβλεψη χρονοσειρών, όπου εξετάζεται η ικανότητά τους σε δεδομένα ακολουθίας.

Στη συνέχεια, προχωράμε στην μη-επιβλεπόμενη εκμάθηση (unsupervised learning) με αυτόματους κωδικοποιητές (AutoEncoders), εξηγώντας το πλαίσιο κωδικοποιητή-αποκωδικοποιητή (Encoder - Decoder) και τις εφαρμογές τους σε συμπίεση δεδομένων και μείωση θορύβου. Παρέχεται επίσης ένας ολοκληρωμένος οδηγός για τα Γενετικά Αντιπαλούμενα Δίκτυα (GANs), που εξετάζει τη διαδικασία αντιπαλότητας μεταξύ των μοντέλων γεννήτριας και διακρίβωσης.

Τέλος, η διατριβή συζητά μια σύγχρονη αρχιτεκτονική νευρωνικών δικτύων: το μοντέλο Transformer, που αναπτύχθηκε για να βελτιώσει την αρχιτεκτονική της μνήμης μακράς-σύντομης διάρκειας για την κωδικοποίηση πολύπλοκων δεδομένων ακολουθίας με καλύτερη αναπαράσταση. Εξηγεί τις αρχές των transformers και τις διάφορες αρχιτεκτονικές τους σχετικές με την εκπαίδευση. Ως συνέπεια της βαθιάς εκμάθησης, αυτό παρέχει ένα οργανωμένο πλαίσιο για τα νευρωνικά δίκτυα και την ιστορία τους, καθώς και για το τι μπορεί να υπάρχει στο μέλλον για τα νευρωνικά δίκτυα.

Λέξεις κλειδιά Νευρωνικά Δίκτυα, Βαθεια Μάθηση, Τεχνητή Νοημοσύνη

Acronyms

M-P = McCullogh and Pitts

ANN = Artificial Neural Networks

NN = Neural Networks

MLP = Multi Layer Perceptron

PCA = Principal Component Analysis

MSE = Mean Squared Error

AI = Artificial Intelligence

CNN = Convolutional Neural Networks

RNN = Recurrent Neural Networks

GAN = Generative Adversarial Networks

RGB = Red Green Blue

ILSVRC = ImageNet Large Scale Visual Recognition Challenge

LRN = Local Response Normalization

VGG = Visual Geometry Group

LSTM = Long Short-Term Memory

GRU = Gated Recurrent Units

CTC = Connectionist Temporal Classification

BPTT = Back Propagation Through Time

NLP = Natural Language Proccessing

MRI = Magnetic Resonance Imaging

DCGAN = Deep Convolutional Generative Adversarial Network

WGAN = Wasserstein Generative Adversarial Networks

BPTT = Back Propagation Through Time

LLM = Large Language Model

BERT = Bidirectional Encoder Representations Transformers

MLM = Masked Language Modeling

GPT = Generative Pre-trained Transformer

ViT = Vision Transformers

Acknowledgements

I would like to express my gratitude to my thesis advisor Christos Shoinas, for the valuable help throughout the entire process of writing my thesis.

Dedication

The thesis is dedicated to my son Panagiotis

Contents

List of Tables

List of Figures

Chapter 1

Introduction to Neural Networks

1.1 Introduction to Artificial Neuron (ANNs)

First, let us briefly explain the basic function of biological neurons so that we can see the analogy between the Artificial and the Biological neuron and understand how scientists were inspired by the physical operations to construct an artificial neuron.

Neurons are the fundamental units that comprise the nervous system. They are special cells that except from the common structures that share with all other eukaryotic cells have also some unique structures like axons and dendrites.

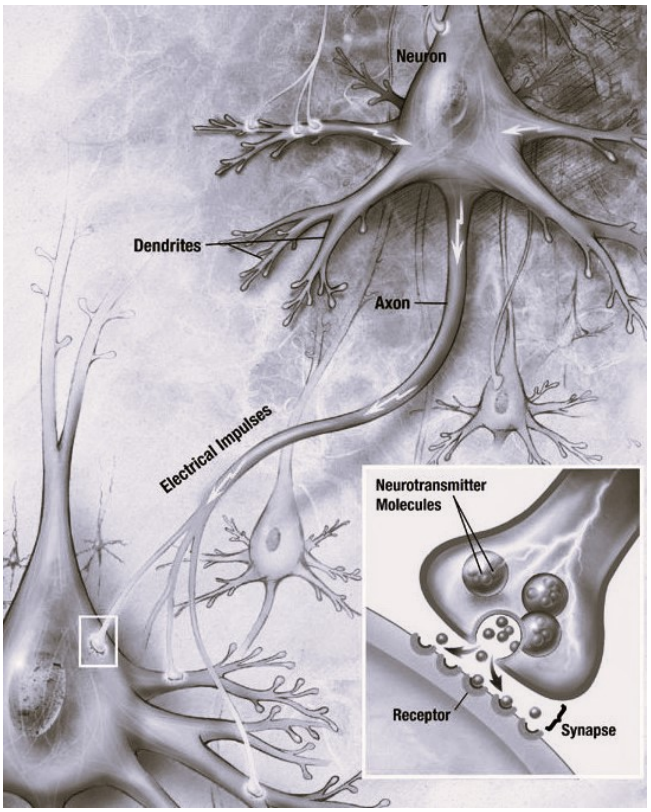


Figure 1.1: Neuron

The neuron's body, sometimes referred to as soma, is the core section of a neuron that contains the nucleus and is the place where most of the protein synthesis occurs.

Dendrites are short, branching fibers that extend from the body of a neuron, providing an increased surface area for receiving incoming information (receptor,post-synaptic)[27].

An axon is a long, cable-like, projection of soma, which conducts action potentials (electrical impulses) from the cell body down to the axon terminal. A vital sub-component of an axon

is called the axon hillock and is important because of the high concentration of voltage-gated sodium. Whenever action potentials are generated, they are generated here and move down the axon.

The Axon terminal is the ending area of the neuron axon (pre-synaptic area) where neurotransmitters are released. A small gap between the pre-synaptic membrane and post-synaptic membrane is called a synapse and is a crucial part of neurons.

Dendrites receive electrical impulses from the neighboring neurons and stimulate soma. If the stimulation surpasses a threshold, an action potential is triggered from the axon hillock, and through the axon, the signal, travels to the axon terminal where an amount of neurotransmitters is released (proportional to electrical signal) into the synapse (Figure 1.1).

The breakthrough in the area of Neural Networks was made by two great scientists, Warren McCulloch and Walter Pitts that were working on a formulation of neuronal function as binary logic.

McCulloh and Pitts first introduced (1943)[28] a mathematical model of a neuron that sums, the weighted, binary, inputs and produces an output if the summation value exceeds a threshold (Logical Threshold Unit). In their model the weights, that were applied to inputs, could be either 0 or 1. A negative weight applied on an 1 input makes the contribution to the neuron inhibitory and a positive one excitatory.

"The inhibitory synapse may be of such a kind as to produce a substance which raises the threshold of the neuron, or it may be so placed that the local disturbance produced by its excitation opposes the alteration induced by the otherwise excitatory synapses". [28]

The original McCulloch and Pitts model lacked built-in learning capabilities, requiring manual programming for specific tasks.

Inputs: $\{ x_1 \dots x_n \}$

Weights: $\{ w_1 \dots w_n \}$

Bias: b, Threshold: θ

sum: $\sum_{i=1}^{n} w_i x_i + bias$

$$\phi(sum) = \begin{cases} 1 & \text{if } sum \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

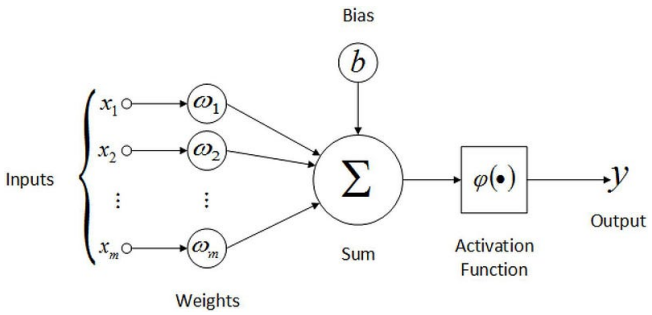


Figure 1.2: McCulloch and Pitts model
[1]

Many scientists disagree with directly comparing biological and artificial neurons because of their significant differences. Still, despite these arguments, we can arbitrarily make an analogy of biological and artificial neurons just for a more comprehensive presentation of artificial neurons. The dendrites can be matched with inputs, the weights with synapses, the summation with the soma, the activation function with the axon hillock, and the output with the axon terminal.

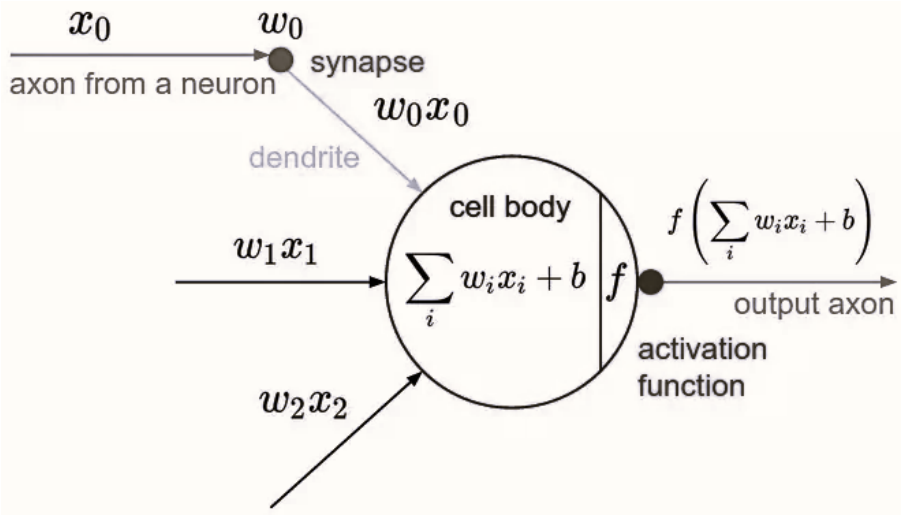


Figure 1.3: Artificial to Physical Neuron Analogy
[24]

Although ANNs have drastically evolved during the last decades, the principles of the model laid out by McCulloch and Pitts (inputs, weighted connections, summation, activation function) remained fundamental. They are used by the most complex big and modern Artificial Neural Networks.

1.2 Fundamental concepts

The McCulloch and Pitts neuron serves as the foundational computational unit, possessing limited capabilities. However, they argued that interconnected neurons could collectively compute any computable function, with a specific emphasis on Boolean functions. These interconnected neurons form the basis of Artificial Neural Networks (ANNs).

In 1958 Frank Rosenblatt[35] proposed an artificial neural network model called **Perceptron**, where the inputs and weights could be any real number, outputs are binary and the activation function is a threshold logic unit. Originally, the perceptron model had multiple inputs and outputs and was single-layered, meaning no other neurons intervened between inputs and outputs.

Later in 1986, a **Multi-Layered Perceptron** was presented by Rumelhart D.,Hinton G. & Williams.R.[36], including extra neurons between inputs and outputs. Multi-Layred Perceptrons model consists of the cornerstone of modern Artificial Neuron Networks.

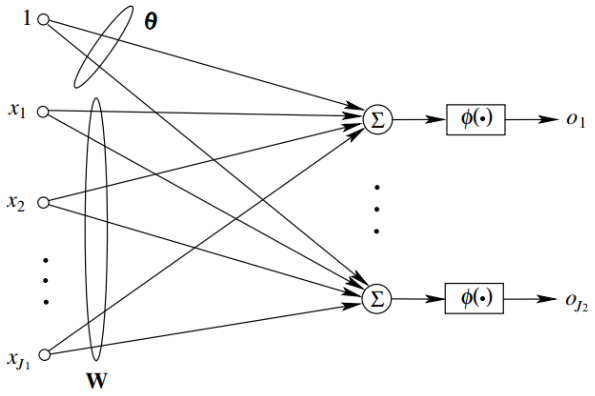


Figure 1.4: Single Layered Perceptron
[24]

Given that training data are fed to perceptron sequentially, the learning method can described mathematically as follows:

Consider the training data sequence denoted by k, x_n be the input vector, $w_{m,n}$ the weight matrix, b is the bias, ϕ is the activation function, y_m the output vector, e_m the error vector and t_m the target value vector then:

$$\vec{y_m} = \phi(\vec{w_n}^T \cdot \vec{x_n} + b)$$

$$\vec{e_m} = \vec{y_m} - \vec{t_m}$$

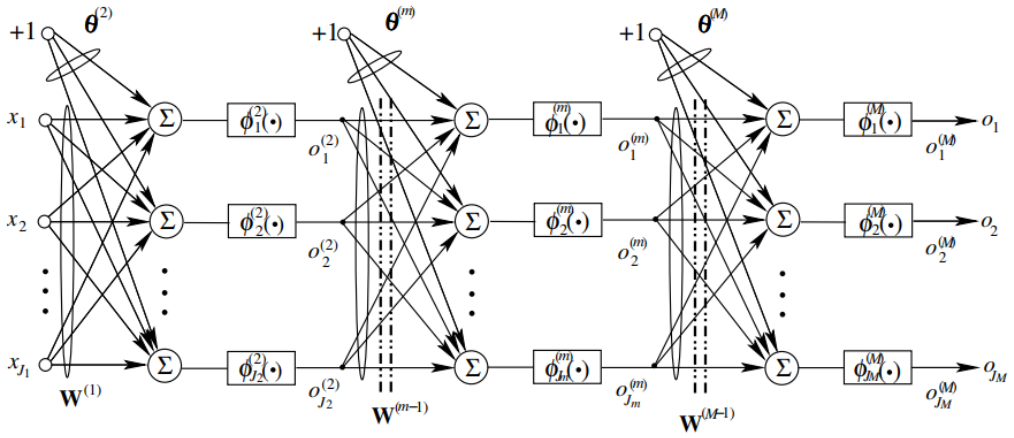


Figure 1.5: Multi Layered Perceptron
[24]

The most simple architecture is the Feed-forward Neural Network where artificial neurons are connected, forming a net of neurons where the info flows from input to output without making any loops. The structure of interconnected neurons is organized into layers, Input, Output, and Hidden Layers. The Hidden Layers receive inputs from the preceding layer and generate outputs that serve as inputs for the subsequent layer.

Later in 1989 George Cybenko published the Cybenko theorem[4] also known as the universal approximation theorem for feedforward neual networks. This theorem states that a feedforward neural network with a single hidden layer containing a finite number of neurons (or units) can approximate any continuous function on a compact input domain to any desired degree of accuracy, given a sufficiently large number of neurons in the hidden layer. We omit the proof of the theorem because this thesis is focused on the results of the theorem.

1.3 Learning Concepts

Donald Hebb in 1949 was the first to propose a weight adjustment method, in his book *The Organization of Behavior*[14]. Hebb proposed a learning rule also known as *Hebbian learning*,

where if a pre-synaptic neuron cell stimulates repeatedly and persistently a post-synaptic neuron cell, then the synapses between them are strengthened.

This mathematically can be described as

$$\Delta w_{i_j} = x_i \cdot y_j$$

, where x is the input vector, y is the y is the output vector, and w is the weight between x,y.

Although Hebb's rule is a primitive way of making neural networks learn, his groundbreaking proposal laid the foundation for learning techniques in neural networks. After its publication, more and more learning techniques were developed and presented.

Marvin Minsky and Seymour Papert published "Perceptrons"[29] highlighting the limitations of single-layer perceptrons, particularly their inability to solve non-linearly separable problems like the XOR problem. This led to a temporary decline in neural network research.

In 1986, Rumelhart, Hinton, and Williams reintroduced the backpropagation algorithm[36], which updates the weights of neurons by utilizing the gradient descent method to minimize error. Backpropagation algorithm was firstly introduced by Paul Werbos[42] and it is considered as one of the most historic innovations in the field of Neural Networks. It computes the gradient of a loss function with respect to the weights of the network for a single input–output example.

Below we will present an example of how backpropagation acts on a feedforward Neural Network. Suppose we have a Neural Network with n inputs (X_n), k neuron cells in the hidden layer (H_k), and m outputs (Y_m). First, we are going to present the forward propagation by calculating the output with respect to the input and weights. Assume that the activation function is f. Then the output of each neuron in the hidden layer (H) is

$$net_k = \sum_{i=1}^{n} W_{ki} \cdot X_i + \text{bias}$$

$$H_k = f\left(net_k\right)$$

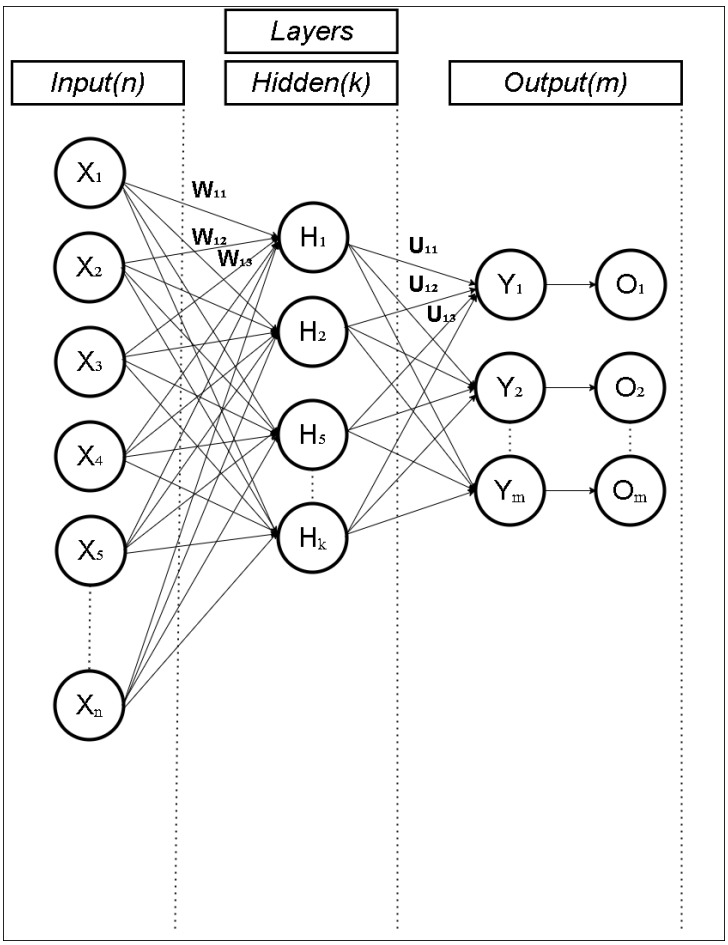


Figure 1.6: Neural Network

In the same manner, the output of each neuron in the output layer will be

$$net_m = \sum_{j=1}^{k} U_{mj} \cdot H_j + \text{bias}$$

$$Y_m = f\left(net_m\right)$$

and

$$O_m = f\left(Y_m\right)$$

At this point, given the inputs, we have used the Neural Network to produce an output. The output contains an error, which is the difference between the predicted output and the actual output. This error is calculated using the so-called Loss Function.

$$E = \frac{1}{m}\sum_{z=1}^{m} Y_z - \hat{Y}^2$$

There are many different Loss functions; in our example, we use the MSE (mean squared error).

Backpropagation uses the chain rule to efficiently calculate the gradients in respect to output, hidden layer 1 etc. Assuming that δ_{ym} is the gradient of the m_{th} output

$$\delta_{ym} = \frac{\partial E}{\partial O_m} \cdot \frac{\partial O_m}{\partial Y_m} = \frac{\partial E}{\partial f(Y_m)} \cdot \frac{\partial f(Y_m)}{\partial Y_m} = \frac{\partial(\frac{1}{m}\sum_{z=1}^{m} f(Y_z) - f(\hat{Y})^2)}{\partial f(Y_m)} \cdot \frac{\partial f(Y_m)}{\partial Y_m}$$
$$= (f(Y_m) - \hat{Y}) \cdot f'(Y_m)$$

.

Now we know the gradient of a certain output lets say j we will use it to find the $\frac{\partial E}{\partial U_{ij}}$ (weight which connects H_i to Y_j) which shows the contribution of this weight to the total error and actually is the weight upadate $\Delta U_{ij} = \frac{\partial E}{\partial U_{ij}}$

$$\frac{\partial E}{\partial U_{ij}} = \frac{\partial E}{\partial f(Y_j)} \cdot \frac{\partial f(Y_j)}{\partial Y_j} \cdot \frac{\partial Y_j}{\partial U_{ij}}$$

but we previously show that

$$\frac{\partial E}{f(Y_j)} \cdot \frac{f(Y_j)}{\partial Y_j} = \delta_{yj}$$

and with some basic arithmetic we can easily find that

$$\frac{\partial Y_j}{\partial U_{ij}} = H_i$$

because the fraction is in respect to U_{ij} and all terms except one ($\partial(H_i \cdot U_{ij})$) will be discarded and we will have $\frac{\partial(H_i \cdot U_{ij})}{\partial U_{ij}} = H_i$, so

$$\frac{\partial E}{\partial U_{ij}} = \delta_{yj} \cdot H_i$$

This can be written in a vector form that generalizes the update weight formula to:

$$\Delta\vec{U} = -a \cdot \vec{\delta_y}^{\mathrm{T}} \cdot \vec{H}$$

where a denotes the learning rate. Following the same technique we can calculate the gradients of Loss Function with respect to the hidden layer δ_h and we will have the formulas for $\vec{W}$

$$\Delta\vec{W} = -a \cdot \vec{\delta_h}^{\mathrm{T}} \cdot \vec{X}$$

In order everything to be crystal clear and explained, the calculation of δ_h (gradient) is presented below. It is worth mentioning that in our previous calculations when we refered to H_i we were talking about the neuron H_i after activation function. In the following section neuron after activation wll be refered to as $f(H_i)$ and pre activation as H_i. Let, for example find the gradient of Loss function with respect to hidden layer neuron $f(H_2)$. Using the chain rule, we sum over all contributions from each output neuron Y to H_2

$$\frac{\partial E}{\partial H_2} = \sum_{i=1}^{k} \frac{\partial E}{\partial Y_i} \cdot \frac{\partial Y_i}{\partial H_2}$$

We previously show that $\frac{\partial E}{\partial Y_i} = \delta_{oi}$, and we must find $\frac{\partial Y_i}{\partial H_2}$.

Since from forward pass we have

$$Y_i = \sum_{j=1}^{m} U_{ij} \cdot f(H_j) + bias_j$$

the partial derivative of Y_i with respect to H_2 will be

$$\frac{\partial Y_i}{\partial H_2} = U_{ij} \cdot f'(H_2)$$

Putting all together we get the final formula for gradient of Losss function with respect to hidden layer

$$\delta_{hi} = (\sum_{j=1}^{k} \delta_{oj} \cdot U_{ji}) \cdot f'(H_i)$$

After backpropagation calculates the gradients, *Gradient Descent* algorithm uses the concept of following the steepest descent to minimize a function. It relies on gradients and learning rate to update parameters (weights and biases) in the direction that will decrease the loss.

Together, gradient descent and backpropagation form the core of training neural networks.

The above example is only demonstating one method of learning, the supervised learning. Means that network predicts an output but it needs the actual (labeled, correct, ground truth) output to adjust the weights and be trained. There are three main types of learning in neural networks: supervised learning, unsupervised learning, and reinforcement learning.

Unsupervised learning focuses on analyzing unlabeled data to uncover hidden patterns. It utilizes specialized techniques like Principal Component Analysis (PCA) to extract information about the data. While often associated with neural networks, unsupervised learning can be applied with other algorithms as well. One common application of unsupervised learning is pattern recognition.

Reinforcement learning uses agents that interact with their environment. A neural network processes information from the environment and guides the agent's actions. The outcomes of these actions, along with the agent's experience, are used to update the network's internal parameters, allowing the agent to learn and improve its behavior over time.

1.4 Applications of neural networks

Neural networks are used in a variety of applications, such as computer vision, natural language processing, recommendation systems, and many others. Their ability to learn complex patterns from data makes them exceptionally powerful tools in these fields.

In computer vision, neural networks can accurately identify and classify objects within im-

ages and videos, enabling advancements in facial recognition, medical imaging, and automated surveillance systems. In natural language processing, they facilitate sophisticated understanding and generation of human language, powering tasks such as translation, sentiment analysis, and chatbots that interact seamlessly with users.

Recommendation systems leverage neural networks to analyze user behavior and preferences, providing personalized content and product suggestions. For example, streaming services and online retailers use these systems to enhance user experience and drive engagement.

In speech recognition, neural networks are critical for converting spoken language into text with high accuracy. This technology is fundamental for virtual assistants, transcription services, and hands-free control of devices, significantly improving accessibility and convenience.

Game playing has also seen remarkable advancements through neural networks, particularly in the development of AI that can learn and master complex games. These AI systems have outperformed human champions in games like chess, Go, and various video games, demonstrating their capability for strategic thinking and adaptation.

Medical diagnosis is another crucial application area where neural networks excel. They assist in analyzing medical images, identifying diseases, and predicting patient outcomes with precision, supporting healthcare professionals in making informed decisions and improving patient care.

Autonomous driving represents one of the most transformative uses of neural networks. These systems process vast amounts of sensor data to navigate and make real-time decisions on the road, aiming to enhance safety and efficiency in transportation.

Overall, the versatility and transformative impact of neural networks across various industries underscore their importance in driving innovation and improving technological capabilities.

1.5 Thesis organization

This thesis is a study that digs into the analysis of various neural network techniques.

We begin by establishing the foundation with a review of perceptrons, the building blocks of neural networks. Following this, we explore more advanced architectures, analyzing Convolutional Neural Networks (CNNs) for their ability to extract spatial features in data, particularly useful for tasks like image recognition.

Next, we examine Recurrent Neural Networks (RNNs) and their strength in handling sequential data, making them valuable for tasks like language processing.

Autoencoders and Generative Adversarial Networks (GANs) will then be explored, investigating their capabilities in data compression, dimensionality reduction, and even generating entirely new data.

Finally, we will delve into the recent advancements with Transformers, a powerful architecture redefining the state-of-the-art in various natural language processing tasks.

Chapter 2

Convolutional Neural Networks (CNNs)

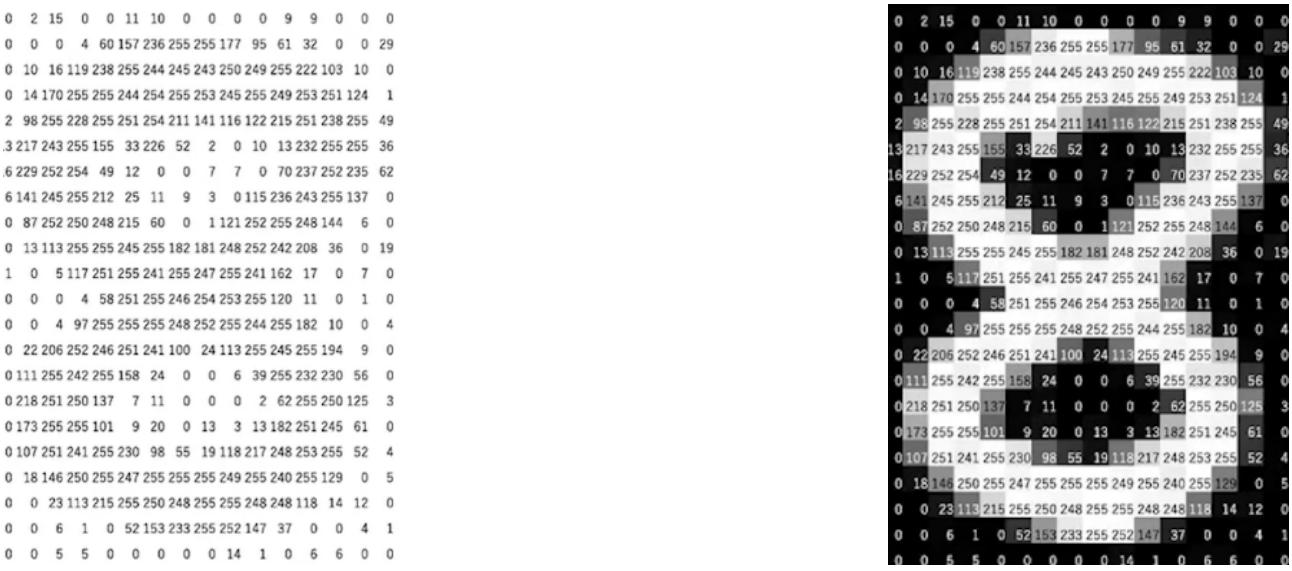
CNNs are a strong tool in artificial intelligence and they are constantly evolving. They are mostly used in computer vision for tasks like image recognition, object detection, and image classification. They find applications in medical imaging, autonomous driving, facial recognition, and others.

They were first introduced in the late 1980s by Yann LeCun[25] and his colleagues at AT&T Bell Labs. Their pioneering work was demonstrated in the 1989 paper "Backpropagation Applied to Handwritten Zip Code Recognition", where they showed how gradient-based learning algorithms could train CNNs to recognize handwritten digits.

In the next sections, we will analyze how a computer store images, the new concepts introduced by LeCun's work (such as convolutional layers and pooling layers) and finally some specialized architectures of CNNs and their application.

2.1 Digital images

A digital greyscale image, is a matrix where each element of the matrix represents the intensity o light at that point.



(a) Values of a matrix of a greyscale image [39]

(b) Representation of values of a matrix [39]

Figure 2.1: Greyscale image matrices

One the other hand in a digital color image each element of the matrix represents a colour. Colors in digital images are often represented using three components: red, green, and blue (RGB). These components can be thought of as coordinates in a three-dimensional color space. In this sense, colors can be represented as 3D vectors where each dimension corresponds to the intensity of one of the RGB components.

Images are typically very large and it is insufficient to train neural networks directly using raw images. We need to extract the more critical information from images because this help us to reduce the dimensionality of the image data, while preserving important information. This technique is called *Feature Extraction*, it makes the training process more efficient and allows the network to learn relevant patterns and features, such as edges, textures, and shapes, which are crucial for tasks like image recognition and classification.

In CNNs feature extraction happens (producing feature maps) in convolutional layer where a kind of filtering of the image is happening in order to detect patterns, edges, textures etc. Pooling layer acts on feature maps extracted by convolutional layer and it's aim is to reduce the dimension of the data, to retain the most important information while reducing the amount of computation and prevent the network from overfitting.

2.2 Convolutional Layers

As it was mentioned before, convolution layer purpose is to extract the features of the input data (image). But which are the feaures of an image? We can divide feature to Locals and Globals. Local feature is a characteristic of a small region of an image and could be the color, the texture, a shape or an edge. Gloabal features are relevant to larger portions of an image and could be the distribution of pixel intensities in that portion (histogram), or the overall shape, size, aspect ratio, orientation (morphological feature), or color distribution in that larger area of an image. Back to the concept of convolutional layer, where we are going to explain how the afforementioned features are extracted. The mechanism uses an entity called kernel or filter, which is actually a 3D matrix. The dimensions of the matrix vary and depend on color channels

and features to be extracted. To be more specific color channel affects depth and features affect height and width.

The kernel is dot-producted with the values of the image starting at the left top corner. The result of the product is stored to an array called feature map. Afterwards kernel is moving one column right and again is dot-producted with the images values and result stored to feature map. This procedure continues until we reach the end of the image to the right, then kernel moves one row to the bottom and repeats the procedure. When kernel reaches the bottom right corner of image, the operation of feature extraction has come to an end and the result is a feature map.

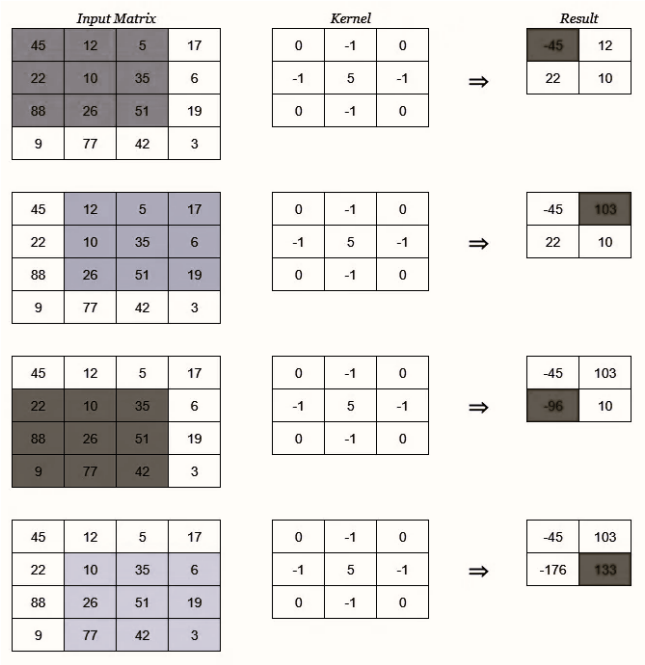


Figure 2.2: Kernel movement
[19]

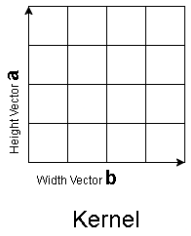
It's worth mentioning some basic terms that are very common when it comes to feature extraction and these are (except those referred previously) *Stride* and *Padding*. Stride is the step size of the kernel as it moves across the input. Previously for simplicity reasons we accepted that kernel moves only one column at a time. Next is the padding that has to do with zeros added to the input image to ensure that feature map has the same spatial dimensions as the input.

After describing how feature extraction acts on input data, the two basic questions that haven't

answered yet and deliberately left until now, are about what are the values that a kernel should consist of and why, and also why dot-producting? Firstly we are going to answer the second question about dot-product. By dot-producting kernel with a part of an image, actually we check if the certain part of the image is similar to kernel. If we write down the dot-product formula

$$\vec{\alpha} \cdot \vec{\beta} = \sum_{i=1}^{n} a_i b_i = |\alpha||\beta| \cos\theta$$

we can detect a gap in what it has been written about dot-product between kernel and an image portion, since a kernel it is actually a vector space produced by the height and width vectors (assume greyscale image where each element of image matrix is a number rather than vector), the same applies to image. So it's like trying to dot-product between vector spaces. In order to find it we reshape the matrices to vectors (flatten) and know can execute the operation.



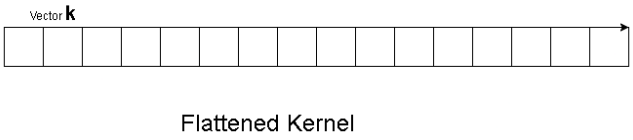


Figure 2.3: Kernel Reshaping

In color images, the afforementioned operation is executed on each color and the results are summed. Dot-product formula indicates that it comprehends $\cos\theta$ which actually shows the cosine similarity between the two vectors. The reason cosine similarity is not used is because it adds computation overhead (as we must divide with product of the two vectors magitude)

$$\cos\theta = \frac{\sum_{i=1}^{n} a_i b_i}{|\alpha||\beta|}$$

and because it is observed that in some cases can vanish the gradients (through normalization).

Finally, we can admit that dot-product detects a *Relative Similarity between vectors* and thats why it is used for feature extraction.

Now that dot-product is explained we can analyze kernel design. Kernels are typically designed to have a specific pattern they want to detect. Ideally, the elements corresponding to the "important" features in the kernel will have higher weights compared to others. In other words, if we create a kernel representing an edge, it will pass through the input image to detect for an edge. This can be repeated for shapes,corners etc, ending up having deconstruct an image to feature maps that hold all the important features of the specific image.

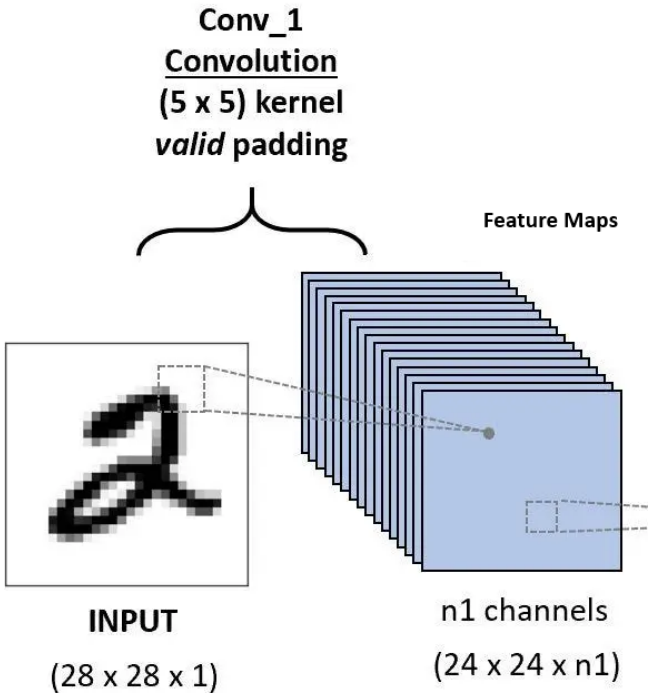


Figure 2.4: Convoluting
[37]

Mathematically, a single kernel action on Image can be described by the following formula:

$$\sum_{m=0}^{k1}\sum_{n=0}^{k2} I_{i+m,j+n} * K_{m,n}$$

Where I is the image matrix with dimensions i,j and K is the kernel matrix with dimensions (k1,k2). This formula represents the cross correlation and is used by almost every machine learning and deep learning library. It is used over the dot product formula that we have

mentioned before, because in dot product we need to flip the kermel matrix to get the desired functionallity since the dot prduct formula for the same matrices is given by the formula:

$$\sum_{m=0}^{k1}\sum_{n=0}^{k2} I_{i-m,j-n} * K_{m,n}$$

Kernels are a part of the network that are affected by the learning procedure. At the begining kernel values (considered as weights) are random numbers which later when error is back-propagated through the network, are updated in relation to how much each kernel value contributed to the overall error.

Finally a special referal to activation functions must be written, which are typically applied element-wise to each element of the feature map. They are used to introduce non-linearity in to the network in order to learn more complex patterns. Although they are often presented as a separate layer (activation Layer), they are typically considered part of the convolutional layer in implementation.

Before closing this section, we are going to sum up the characteristics of the convolutional layer and we will explain why it is crucial to know how these values affect the whole network training. The distinguished values related to convolutional layer are the

- Convolutional Layer Parameters:
  - Number of Kernels
  - Kernel Size (eg. 32×32)
  - Stride
  - Padding
  - Activation Function

and are of great importance, since they are the fundamental parameters to set up CNNs using software tools like *Keras, TensorFlow* etc. It will be more clear when we are going to analyze some very famous variants of CNNs and how they used these special parameters.

2.3 Pooling Layers

Pooling layers are primarily used for subsampling the output of convolutional layers (feature maps) to reduce the dimensions of input data. This reduction achieves lower computational load, decreased memory usage, and fewer parameters, which limits the risk of overfitting. Similar to convolutional layers, pooling layers use a filter (kernel) with defined stride and padding. However, the pooling filter doesn't contain learnable weights. Instead, it defines an area of the input data (feature map) from which a single value will be extracted, while other values in this area are dropped.

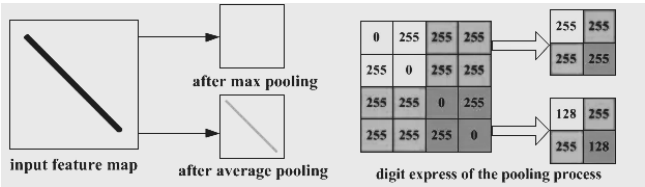


Figure 2.5: Convoluting
[32]

There are two main types of pooling:

- *Max Pooling*: selects the highest value from the area marked by the filter.
- *Average Pooling*: computes the average value of the marked area.

The filter passes through the feature map in the same manner as in convolutional layers (using stride and padding), producing a significantly reduced map. For this reason, pooling is considered a dimensionality reduction technique. This contrasts with convolutional layers which, while also reducing dimensionality to some extent, primarily serve for feature extraction.

2.4 Architecture Summary

In this section we are going to see CNN with a more abstract perspective and not focus specifically in Layers or techniques. Firstly the flow of information will be presented and next we are

going to make a try to explain the affection of crucial prameters to the output of the CNN. We will use an example of training set containing images to analyze the basic operations happening in a CNN.

An image comes as an input to a CNN. The convolutional layer firstly it is going to extract the features, although randomnly for the first time sicnce weights of kernel at the first pass are randomly selected. The kernels of convolutional layer will produce feature maps to which pooling layer will be apllied to reduce dimensionality. The output of the pooling layer is flattenend and used as an input to fully connected feed forward network. This is the end of forward propagation procedure of a CNN.

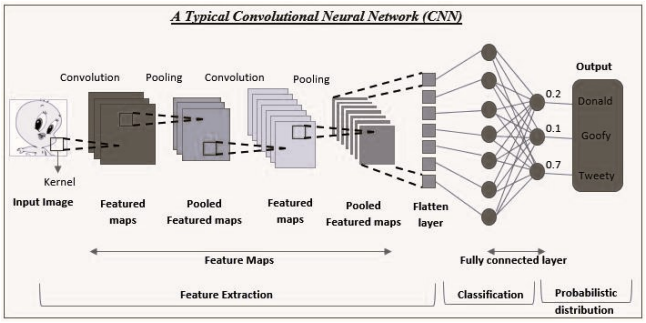


Figure 2.6: Convoluting
[32]

After some sample input data have forward passed the network, and the total error has been averaged, it is time to use that averaged error to update weights using the back-propagation algorithm. This procedure for feed forward networks has thoroughly explained in section 1 so we are going to see how back propagation is applied to the convolutional and pooling layers which are acting between input data and flattend feature map. The functionallity of a layer is identical to others, so, showing how training applies in on layer, it can very easily expanded to all layers. We can define *Input feature map* as X, *Output feature map* as O, K as the kernel weights that are applied to input feature map in order to produce output feature map and E the Loss function.

The formula of the Loss function is:

$$E = \sum_{i=0}^{H'-1} \sum_{j=0}^{W'-1} O_{i,i} - \hat{Y}_{i,j}^{\ 2}$$

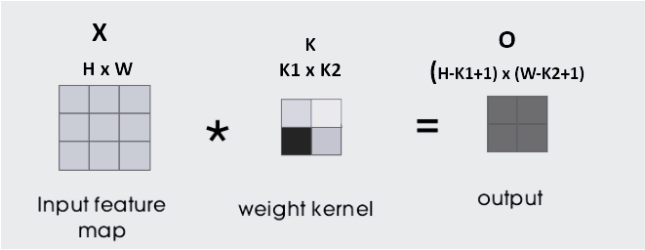


Figure 2.7: A layer in CNN
[21]

where $H' = H - K1 + 1$ and $W' = W - K2 + 1$ The formula giving the gradients of Loss function with respect to kernel weight is:

$$\frac{\partial E}{\partial K_{m,n}} = \sum_{i=0}^{H'-1} \sum_{j=0}^{W'-1} \frac{\partial E}{\partial Oi,j} \cdot \frac{\partial Oi,j}{\partial K_{m,n}}$$

The term $\frac{\partial E}{\partial Oi,j}$ is the gradient of Output $\delta_O i,j$ which has already been calculated. So we need to find $\frac{\partial Oi,j}{\partial K_{m,n}}$. Given that in forward pass we have

$$O_{i,j} = \sum_{i=0}^{H'-1} \sum_{j=0}^{W'-1} X_{i+m,Jn+n} \cdot K_{m,n}$$

then

$$\frac{\partial Oi,j}{\partial K_{m,n}} = X_{i+m,j+n} \delta_O i,j$$

So the final formula for gradient of Loss function with respect to krnel weight is:

$$\frac{\partial E}{\partial K_{m,n}} = \sum_{i=0}^{H'-1} \sum_{j=0}^{W'-1} \delta_O i,j \cdot X_{i+m,j+n}$$

and the update rule for kernel weights is:

$$K_{m,n}^{t+1} = K_{m,n}^{t} - \eta \cdot \sum_{i=0}^{H'-1} \sum_{j=0}^{W'-1} \delta_O i,j \cdot X_{i+m,j+n}$$

where t denotes the layer.

2.5 Specialized CNN Architectures (LeNet, AlexNet, VGG, ResNet)

In previous sections we touch on the core of the CNNs and explain how the basic mechanism works. Now we are going to quote some famous variations of the basic structure of CNN.

LeNet-5 Architecture it is maybe the most famous CNN architecture, and it was introduced by Yann Lecun[26]. It is mainly used for hadwritten digit recognition.

Layer	Type	Maps	Size	Kernel Size	Stride	Activation
Out	Fully Connected	---	10	---	---	RBF
F6	Fully Connected	---	84	---	---	tanh
C5	Convolution	120	1x1	5x5	1	tanh
S4	Avg Pooling	16	5x5	2x2	2	tanh
C3	Convolution	16	10x10	5x5	1	tanh
S2	Avg Pooling	6	14x14	2x2	2	tanh
C1	Convolution	6	28x28	5x5	1	tanh
In	Input	1	32x32	---	---	---

Figure 2.8: LenNet-5

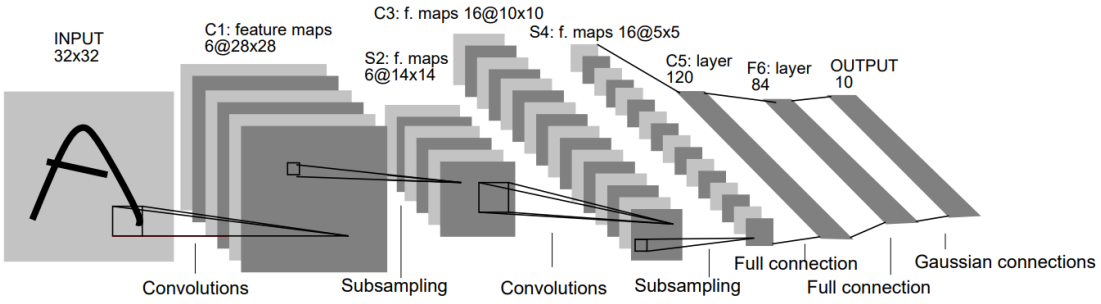


Figure 2.9: LenNet-Architecture

It worths mentioning some extra details of the Lenet-5:

- Inputs are zero padded to 32×32 from 28×28 and are also normalized before fed into the network

- The averge pooling layers are slightly more complex. The average values are multiplied by a learnable coeficient and then a lernable bias is added to the result. Finally activation function is applied.

- In output layer each neuron is calculating the square of the Euclidean distance between input and weights (of output layer) instead of multiplying weights vector with inputs.

AlexNet Architecture developed by Alex Krizhevsky, Ilya Sutskever and Geoffrey Hinton[23] and won the 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC). It is very similar to LeNet but it is also bigger and deeper. In this architecture firstly introduced the stacking of convolutional layers in contrast to previous architectures that used pooling and convlutional one after the other. Two features make AlexNet stand out. The first is a regularization technique

Layer	Type	Maps	Size	Kernel Size	Stride	Activation
Out	Fully Connected	--	1000	--	--	Softmax
F10	Fully Connected	--	4096	--	--	ReLU
F9	Fully Connected	--	4096	--	--	ReLU
S8	Max Pooling	256	3x3	3x3	2	--
C7	Convolution	256	13x13	3x3	2	ReLU
C6	Convolution	384	13x13	3x3	2	ReLU
C5	Convolution	384	13x13	3x3	2	ReLU
S4	Max Pooling	256	13x13	3x3	2	--
C3	Convolution	256	27x27	5x5	1	ReLU
S2	Max Pooling	96	27x27	3x3	2	--
C1	Convolution	96	55x55	11x11	4	ReLU
In	Input	3	227x227	---	---	---

Figure 2.10: AlexNet-Architecture

called *dropout*, where in every training step (forward and backward pas) a randomly selected set of neurons are deactivated. It is like different NNs are trained on each training step making the resulting net seen as an average of smaller NNs.

The other special feature is called LRN (Local Response Normalization) and is applied in

Layer C1 and C3 after the activation function. In this technique a hyper parameter is defining neighborhoods of feature maps. Neurons with stronger activation output reduce the output of the neurons in the neighbor feature maps at the same position. *GoogleNet Architecture* developed by Christian Szegedy[40] from Google Research and won the ILSVRC 2014. The characteristic of this architecture is that it is much deeper from the previous. But the innovation is that it was the first to introduce the *inception modules*.

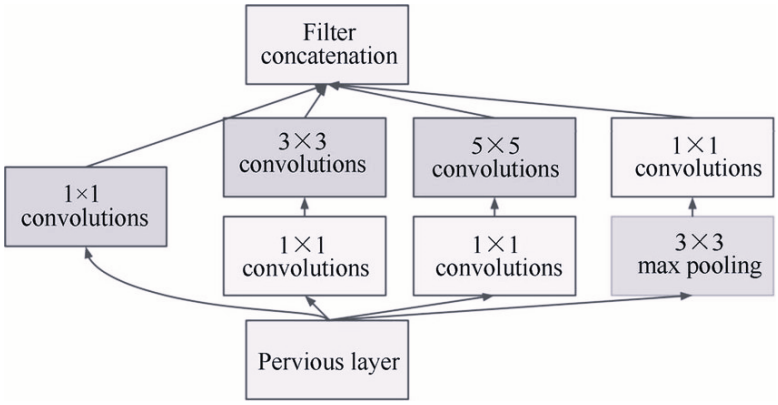


Figure 2.11: Inception module

These modules are the building block of GoogleNet and their contribution to the CNN is to make it even more deeper. Their functionality is quite simple. They copy the output of previous layer and distribute it to multiple convolutional and pooling layers which use such a stride and padding, that all produce feature maps of same dimension. Lastly the produced feature maps are concatenated to one.

VGGNet Architecture developed by Karen Simonyan and Andrew Zisserman from the Visual Geometry Group (VGG)[38]. The architecture is very simple. It follows a pattern of two or three convolutional layers and then a pooling layer and then again until the end where it is fully connected net with 2 hidden layers.

ResNet Architecture developed by Kaiming He and won the ILSVRC 2015[13]. It is based on blocks that include some convolutional and pooling layers, and these blocks are stacked to form the ResNet. But that's also a simple CNN architecture. The thing that makes that CNN unique is that inside each block, the input is copied and added to the output. If we name the target function $h(x)$, actual output $f(x)$ and input x then instead of trying to model the $h(x)$

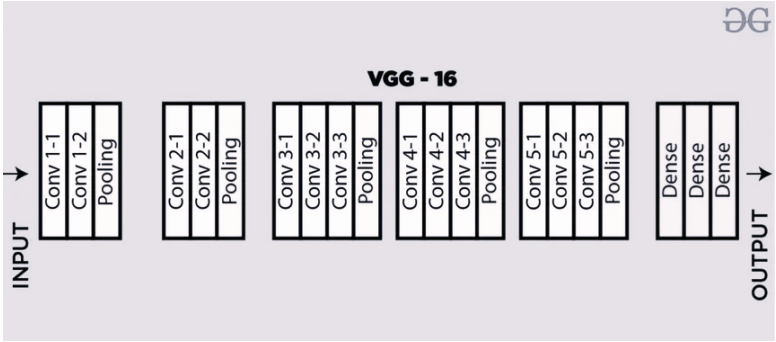


Figure 2.12: VGGNet

it is better to model $h(x) - x$ (residual between input and desired output). With that change we can see much better results in vanishing of gradients and also faster learning since most of the times weights start with values close to zero know with residual blocks input is copied to output so weights getting larger faster.

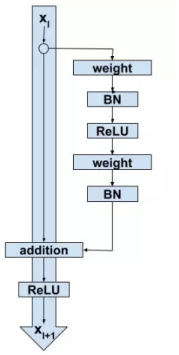


Figure 2.13: Residual Block

Chapter 3

Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) are a type of artificial neural network designed to handle sequential data, such as time series, text, speech etc. Unlike traditional neural networks, which process inputs independently, RNNs have connections that loop back on themselves, allowing them to maintain information across different steps in a sequence. This looping structure enables RNNs to remember past inputs, making them suitable for tasks where the order of inputs matters. For example, in language processing, RNNs can consider previous words to predict the next word in a sentence. However, training RNNs can be difficult due to issues like the vanishing gradient problem, which makes it hard for the network to learn long-term dependencies. To overcome these challenges, more advanced versions like Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs) have been created. These improvements help RNNs perform better and more reliably, making them essential tools in many fields that work with sequential data.

3.1 Introduction to recurrent layers

Recurrent Neural Networks (RNNs) started from the feedback neural networks created in the 1980s by John Hopfield's work[18] on Hopfield networks. Although different in design, Hopfield networks showed how recurrent connections could store and retrieve patterns, laying the groundwork for later RNN research.

During the late 1980s and early 1990s, two important RNN architectures were introduced: Jordan networks[20] and Elman networks[7]. Jordan networks included a context layer that received feedback from the output layer, while Elman networks had a context layer that received feedback from the hidden layer. These designs helped improve the way RNNs handled sequences of data. These early models were groundbreaking but had trouble learning long-term patterns because of the vanishing gradient problem[30].

In 1997, Sepp Hochreiter and Jürgen Schmidhuber introduced Long Short-Term Memory (LSTM) networks[17], which used memory cells and gates to fix the vanishing gradient issue and remember information for longer periods. Building on these early ideas, RNNs became popular in

the 2000s for speech recognition and natural language processing. A major achievement came in 2014 when Baidu(Chinese Technology Company)[12] used RNNs trained with Connectionist Temporal Classification (CTC)[10] to greatly improve speech recognition accuracy. In 2015, Google reduced speech recognition errors by 49% using similar methods[5]. In 2014 Kyunghyun Cho proposed Gated Recurrent Unit [3] which has simpler architecture and keep all the functionality from LSTM.

3.2 Vanilla RNN

All the neural networks we have discussed so far, information flows in one direction: from the input layer to the output layer. These are called Feed Forward Networks. As the name suggests, data moves forward through the network, starting from the input and reaching the output without going back.

In Recurrent Neural Networks (RNNs), things are a bit different. Neurons in one layer not only receive inputs from the previous layer but also use their own previous outputs.

The key point to understand how RNNs work is that an input in RNN is not value but a sequence of values. Therefore, each input splits in *timesteps* that are the individuals values of the sequence of one input. These sequences include the concept of time, so one RNN input contains multiple timesteps that one follows the other.

The equation of forward pass is quite similar to feed forward networks if we just add the weighted inputs from previous timestep

$$\vec{y_m} = \phi(\vec{W_{in}} \cdot \vec{x_t} + \vec{W_{rec}} \cdot \vec{h_{t-1}} + b)$$

where $\vec{W_{in}}$ is the vector of weights act on input, $\vec{W_{rec}}$ is the vector of weights act on recurrent input (previous timestep), ϕ is activation function and b is bias.

Next we are going to present how these networks are trained using a variation of backpropagation algorithm the BPTT (Back Propagation Through Time) but before doing so we must

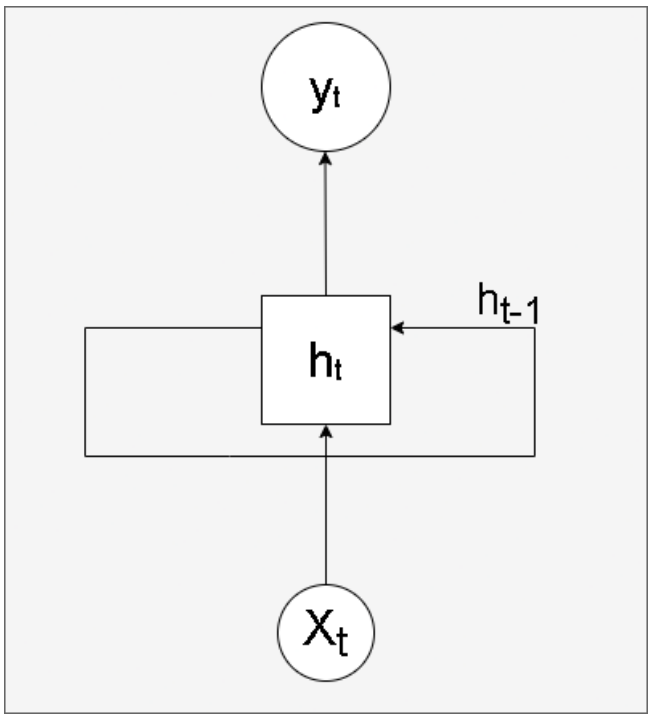


Figure 3.1: Folded RNN

touch on the unfolding procedure of an RNN.

This is important in order to explain how BPTT works. So unfolding an RNN is nothing more than analyzing all the timesteps for a current input individualy. Let us present a showcase of a simplfied model consisting of only one neuron.

At first forward pass occured, $h_1 = \phi(W_{x1} \cdot x_1 + W_{h0} \cdot h0)$ and then output is $y_1 = \phi(W_{o1} \cdot h_1)$.

$$h_2 = \phi(W_{x2} \cdot x_2 + W_{h1} \cdot h1), y_2 = \phi(W_{o2} \cdot h_2)$$

$$h_3 = \phi(W_{x3} \cdot x_3 + W_{h2} \cdot h2), y_3 = \phi(W_{o3} \cdot h_3)$$

At this point we have calculated the outputs for each step or for each member of the sequence. Now it is time to back propagate the error. To do so, we need to use a *Loss function* for BPTT. This is a very crucial part of BPTT because actually we decide which will be the Error that will be back propagated and will help us update the weights accordingly.

The last part is to update the weights for both output, hidden and input Layer. Assume our

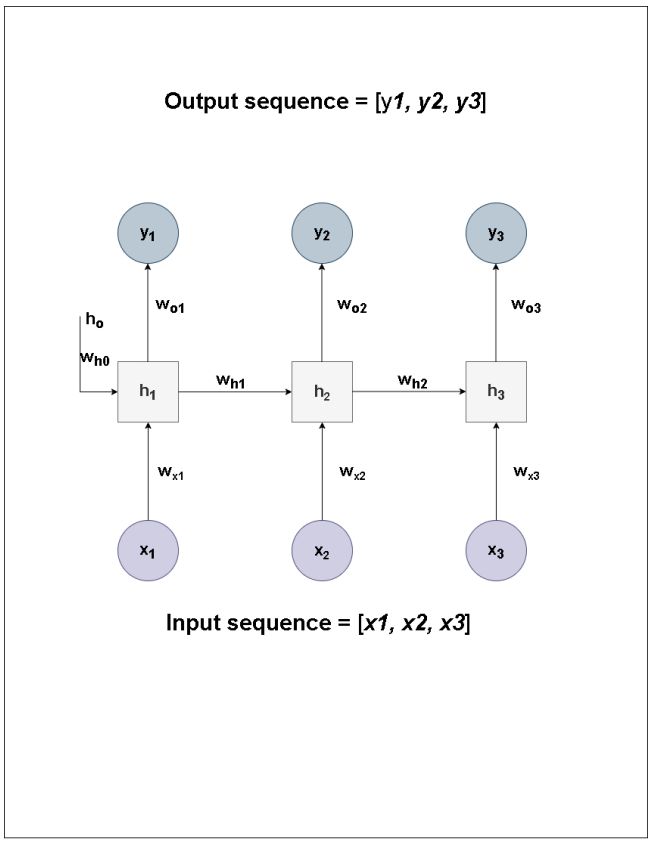


Figure 3.2: Unfolded RNN- Seq to Seq

Loss Function is L, then gradients of output layer are calculated by the formulas:

$$\frac{\partial L}{\partial y_1}, \frac{\partial L}{\partial y_2}, \frac{\partial L}{\partial y_3}$$

Gradients of hidden layer:

$$\frac{\partial L}{\partial h_1}, \frac{\partial L}{\partial h_2}, \frac{\partial L}{\partial h_3}$$

Knowing the gradients, weights can be updated using the following formulas:

Output Layer

$$W_{o-new} = W_{o-old} - \eta \cdot \sum_{t=1}^{3} \frac{\partial L}{\partial W_o(t)}$$

Hidden Layer

$$W_{h-new} = W_{h-old} - \eta \cdot \sum_{t=1}^{3} \frac{\partial L}{\partial W_h(t)}$$

Input Layer

$$W_{i-new} = W_{i-old} - \eta \cdot \sum_{t=1}^{3} \frac{\partial L}{\partial W_i(t)}$$

The RNNs are capable of taking an input sequence and produce a output sequence (sequence to sequence). Supose we could use a sequence of stock market prices as an input, and we could predict an output sequence.

They can also be fed with an input sequence and produce a single output vector (sequence to vector). An example could be an RNN that take a sequence of words as inputs and predicts the next word. The sequence to vector and vector to sequence RNNs belong to a category called Encoder-Decoder that will explained in the next chapter.

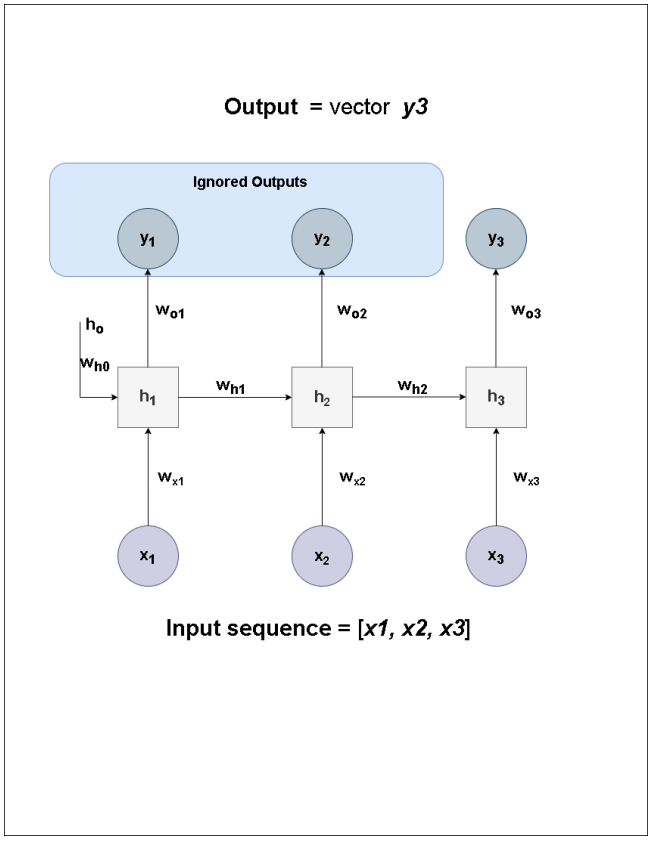


Figure 3.3: Seq to Vec

In conclusion,Vanilla Recurrent Neural Networks is the very basic concept of how we can add some *memory* to neural nets. From what we explained so far it can be understood that Vanilla RNN have limitation on learning long term dependencies in sequences. They need to get deeper and deeper, and usually this strategy drives network to vanish gradients.

The really interesting part of RNNs are the following sections where we explain how scientists achieve to tackle the long-term memory limitation without only creating deeper networks.

3.3 Long Short-Term Memory (LSTM)

The LSTM networks are a very sophisticated version of RNN and add complexity but only in the cell Layer. If we consider the cell as a blackbox then LSTMs are very close of how Vanilla RNNs work. The real novelty is isolated inside cell. But in LSTM networks, cell is a whole architecture by itself for which we are going to discuss in the current section.

So, in contrast to Vanilla RNN which they also weight the input from previous step and process it, LSTMs introduce a whole cell logic which contains gates to let the info flow or block the info from flowing, extra activation functions and element-wise multiplication.

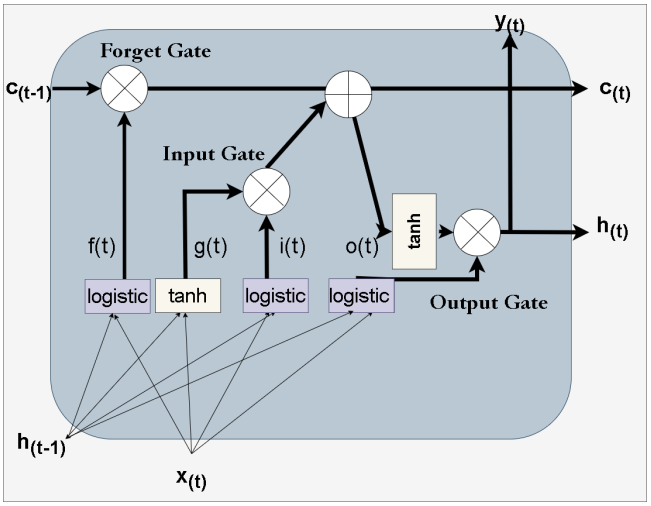


Figure 3.4: LSTM Cell

In an LSTM, the information passed to the next time step consists of two values, *c(t)* and *h(t)*, rather than just one as in a Vanilla RNN. The *c(t)* represents the long-term state, while the *h(t)* represents the short-term state.

The short-term state, *h(t)*, functions similarly to the hidden state in a Vanilla RNN, storing and manipulating recent information. In contrast, the long-term state, *c(t)*, maintains information accumulated throughout the entire training process, effectively capturing the long-term dependencies.

An important component of the LSTM cell is its three gates: *Input*, *Forget*, and *Output*. These gates are responsible for storing or discarding information in the long-term state (*c(t)*) or the

short-term state (*h(t)*). Although LSTMs might initially appear more complex than Vanilla RNNs, their functionality is straightforward.

After the reference to the crucial parts of LSTMs we are ready to explain the flow of info and how the aforementioned components contribute to the overall process.

The current input and previous short-term state are connected to a fully connected layer of nodes. The first node is connected to Forget Gate and the fourth node is connected to Output Gate. The second and third node are connected to Input gate.

The first node of fully Connected layer takes the current input and previous short-term state as arguments, applies weights and the results pass from an *logistic* activation function.*logistic* function produces an output from 0–1.

$$f_{(t)} = \sigma({W_{xf}}^T x_{(t)} + {W_{hf}}^T h_{(t-1)})$$

The $f_{(t)}$ is the output of the *logistic* activation function (0–1).

The second node in the same manner, weights the current input and previous short-term state and the result passed from a *tanh*.

$$g_{(t)} = \tanh({W_{xg}}^T x_{(t)} + {W_{hg}}^T h_{(t-1)})$$

Third and fourth node are identical to node one but with dfferent weights.

$$i_{(t)} = \sigma({W_{xi}}^T x_{(t)} + {W_{hi}}^T h_{(t-1)})$$

$$o_{(t)} = \sigma({W_{xo}}^T x_{(t)} + {W_{ho}}^T h_{(t-1)})$$

Now let's move to the gates of an LSTM cell that control the flow of info in the cell. The *Forget-Gate* takes the output from $f_{(t)}$ and the previous long-term state and performs an element wise

multiplication (Hadamard product) between them.

$$k_{(t)} = f_{(t)} \times h_{(t-1)}$$

An element wise multiplication is also performed between $g_{(t)}$ and $i_{(t)}$, representing the *Input-Gate*.

$$l_{(t)} = g_{(t)} \times i_{(t-1)}$$

The purpose of the Forget-Gate controls what should be removed from long-term memory, since $0 \leq f_{(t)} \leq 1$. A value close to 1 means the information is retained, and a value close to 0 means it is discarded.

So, Input-Gate controls the amount of contribution of the current inputs to the cell state. The sigmoid node acts as a filter that decides how much of the new information should be allowed to enter the cell state. In the Input-Gate, the tanh node processes the new input data to create a vector of candidate values (sometimes referred to as "candidate cell state") that could be added to the cell state. These candidate values are scaled between -1 and 1 to allow both positive and negative changes to the cell state, which is important for adjusting the cell state dynamically based on the input sequence.

The Output-Gate determines what part of the cell state should be outputted to the next hidden state. After the cell state is updated with the new information from the input gate and modified by the forget gate, the output gate uses a sigmoid activation function to decide which parts of the cell state are relevant for the next hidden state.

We have just explained the Forward Pass of an LSTM cell. In order cell to be trained, we need to update the weights of the four nodes. Suppose we use as *Loss Function* the *Mean Squared Error* function

$$E = \frac{1}{n} \sum_{t=1}^{n} Y_t - \hat{Y_t}^2$$

We will show the procedure of weight updating for one timestep but the methodology is identical for any timestep. Firstly update the weights of o_t, so we are trying to calculate $\frac{\partial E}{\partial W_{xo,ho}}$. Applying the chain rule we have:

$$\frac{\partial E}{\partial W_{xo,ho}} = \frac{\partial E}{\partial Y_t} \cdot \frac{\partial Y_t}{\partial o_{(t)}} \cdot \frac{\partial(o_{(t)})}{\partial({W_{xo}}^T x_{(t)} + {W_{ho}}^T h_{(t-1)})} \cdot \frac{\partial {W_{xo}}^T x_{(t)} + {W_{ho}}^T h_{(t-1)}}{\partial W_{xo}}$$

The formula is only for weights of x_t for obvious reasons but the same formula applies also for the weights of h_{t-1} by changing the W_{xo} with W_{ho}. Lets calculate the four fractions seperately.

$$\frac{\partial E}{\partial Y_t} = \frac{2}{n} \cdot (\hat{Y}_t - Y_t)$$

$$\frac{\partial Y_t}{\partial o_{(t)}} = \frac{\partial(o_{(t)} \cdot \tanh(c_t))}{\partial o_{(t)}} = \tanh(c_t)$$

$$\frac{\partial o_{(t)}}{\partial {W_{xo}}^T x_{(t)} + {W_{ho}}^T h_{(t-1)}} = o_{(t)} \cdot (1 - o_{(t)})$$

$$\frac{\partial {W_{xo}}^T x_{(t)} + {W_{ho}}^T h_{(t-1)}}{\partial W_{xo}} = x_{(t)}$$

if we combine the results we the following type for an output weight:

$$\delta_{xo} = \frac{2}{n} \cdot (\hat{Y}_t - Y_t) \cdot \tanh(c_t) \cdot o_{(t)} \cdot (1 - o_{(t)}) \cdot x_{(t)}$$

And finally the rule for the weight update

$$\Delta W_{xo} = -\eta \cdot \delta_{xo}$$

Next will be the weights of forget gate. The following formula implementing the chain rule shows explicitly how we get the final result.

$$\frac{\partial E}{\partial W_{xf,hf}} = \frac{\partial E}{\partial H_t} \cdot \frac{\partial H_t}{\partial C_{(t)}} \cdot \frac{\partial C_{(t)}}{\partial f_{(t)}} \cdot \frac{\partial f_{(t)}}{\partial W_{xf}}$$

Where,

$$\frac{\partial E}{\partial H_t} = \frac{2}{n} \cdot (\hat{Y}_t - Y_t)$$

$$\frac{\partial H_t}{\partial C_{(t)}} = o_t \cdot (1 - \tanh^2(C_t))$$

$$\frac{\partial C_{(t)}}{\partial f_{(t)}} = C_{t-1}$$

$$\frac{\partial f_{(t)}}{\partial W_{xf}} = f_{(t)} \cdot (1 - f(t))$$

and know the same rule applies to the input gate for weights of $g_{(t)}$ and $i_{(t)}$

$$\frac{\partial E}{\partial W_{xg,hg}} = \frac{\partial E}{\partial H_t} \cdot \frac{\partial H_t}{\partial C_{(t)}} \cdot \frac{\partial C_{(t)}}{\partial g_{(t)}} \cdot \frac{\partial g_{(t)}}{\partial W_{xg}} =$$

$$\frac{\partial E}{\partial W_{xg,hg}} = \frac{2}{n} \cdot (\hat{Y}_t - Y_t) \cdot o_t \cdot (1 - \tanh^2(C_t)) \cdot i_{(t)} \cdot (1 - g_{(t)}^2)$$

$$\frac{\partial E}{\partial W_{xi,hi}} = \frac{\partial E}{\partial H_t} \cdot \frac{\partial H_t}{\partial C_{(t)}} \cdot \frac{\partial C_{(t)}}{\partial i_{(t)}} \cdot \frac{\partial i_{(t)}}{\partial W_{xi}} =$$

$$\frac{\partial E}{\partial W_{xi,hi}} = \frac{2}{n} \cdot (\hat{Y}_t - Y_t) \cdot o_t \cdot (1 - \tanh^2(C_t)) \cdot g_{(t)} \cdot i_{(t)}(1 - i_{(t)})$$

3.4 Gated Recurrent Unit (GRU)

The GRU (Gated Recurrent Unit) networks are simpler compared to LSTMs. While both GRU and LSTM are improvements of standard RNNs, GRUs have a less complex structure. In GRUs, there are fewer gates and fewer operations, making them faster and easier to train.

One of the main differences is that GRUs do not have separate long-term and short-term states like LSTMs. Instead, GRUs use a single state, called the hidden state $h(t)$, to hold both recent and long-term information.

GRUs have only two gates: the *Update Gate* and the *Reset Gate*. These gates help the model decide when to keep the old information and when to update it with new data.

Let's take a closer look at these gates:

The update gate controls how much of the previous hidden state $h(t-1)$ should be carried to the next time step. It works by taking the current input and the previous hidden state, applying weights, and passing the result through a *sigmoid* function. The sigmoid function outputs a value between 0 and 1.

$$z_{(t)} = \sigma({W_{xz}}^T x_{(t)} + {W_{hz}}^T h_{(t-1)})$$

If $z_{(t)}$ is close to 1, most of the previous hidden state is kept. If $z_{(t)}$ is close to 0, more of the new information is used.

The reset gate controls how much of the previous hidden state should be "forgotten" when processing new information. Like the update gate, the reset gate also takes the current input and previous hidden state, applies weights, and passes the result through a sigmoid function.

$$r_{(t)} = \sigma({W_{xr}}^T x_{(t)} + {W_{hr}}^T h_{(t-1)})$$

When $r_{(t)}$ is close to 0, the GRU forgets most of the previous hidden state, and when $r_{(t)}$ is close to 1, it keeps more of the past.

Next, the GRU calculates a candidate hidden state $\tilde{h}(t)$. This candidate hidden state is similar to the LSTM's candidate cell state, and it uses a *tanh* function to scale the values between -1 and 1. The reset gate controls how much of the previous hidden state contributes to this candidate state.

$$\tilde{h}_{(t)} = \tanh({W_{xh}}^T x_{(t)} + r_{(t)} \cdot {W_{hh}}^T h_{(t-1)})$$

Finally, the hidden state $h(t)$ is updated using a mix of the previous hidden state and the candidate hidden state. The update gate decides how much of each to use. If the update gate $z(t)$ is close to 1, the hidden state remains mostly the same. If it is close to 0, the hidden state

is replaced by the candidate hidden state.

$$h_{(t)} = (1 - z_{(t)}) \cdot h_{(t-1)} + z_{(t)} \cdot \tilde{h}_{(t)}$$

Finally, the update gate $z(t)$ decides how much of the old information should be carried to the next step, and the reset gate $r(t)$ decides how much of the past should be forgotten when calculating the candidate hidden state.

Now let's look at how to update the weights during training:

If we we use a *Mean Squared Error (MSE)* loss function:

$$E = \frac{1}{n} \sum_{t=1}^{n} Y_t - \hat{Y_t}^2$$

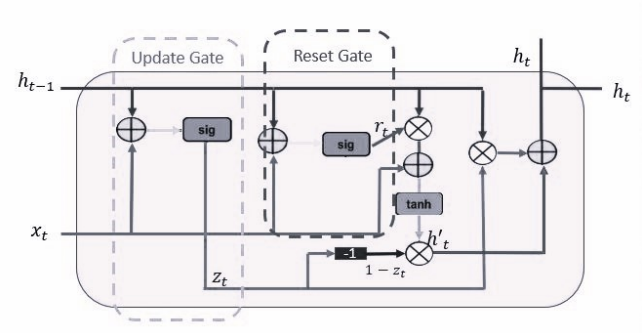


Figure 3.5: GRU cell
[31]

We can apply the *chain rule* to compute the gradients of the loss with respect to the weights. Let's first consider the weights of the *update gate* (W_{xz} and W_{hz}):

The derivative of the loss with respect to the update gate weights is calculated as:

$$\frac{\partial E}{\partial W_{xz,hz}} = \frac{\partial E}{\partial h_{(t)}} \cdot \frac{\partial h_{(t)}}{\partial z_{(t)}} \cdot \frac{\partial z_{(t)}}{\partial ({W_{xz}}^T x_{(t)} + {W_{hz}}^T h_{(t-1)})}$$

We can break this down step by step:

The gradient of the loss with respect to the output Y_t is:

$$\frac{\partial E}{\partial Y_t} = \frac{2}{n} \cdot (\hat{Y}_t - Y_t)$$

The gradient of the hidden state $h_{(t)}$ with respect to the update gate $z_{(t)}$ is:

$$\frac{\partial h_{(t)}}{\partial z_{(t)}} = \tilde{h}_{(t)} - h_{(t-1)}$$

The derivative of the update gate activation $z_{(t)}$ with respect to its pre-activation input is:

$$\frac{\partial z_{(t)}}{\partial (W_{xz}{}^T x_{(t)} + W_{hz}{}^T h_{(t-1)})} = z_{(t)} \cdot (1 - z_{(t)})$$

Finally, the derivative of the weighted sum for W_{xz} is:

$$\frac{\partial (W_{xz}{}^T x_{(t)} + W_{hz}{}^T h_{(t-1)})}{\partial W_{xz}} = x_{(t)}$$

Combining these steps, we get the update for the weights of the update gate:

$$\delta_{xz} = \frac{2}{n} \cdot (\hat{Y}_t - Y_t) \cdot (\tilde{h}_{(t)} - h_{(t-1)}) \cdot z_{(t)} \cdot (1 - z_{(t)}) \cdot x_{(t)}$$

And in the same way for the weights W_{hz}, we replace $x_{(t)}$ with $h_{(t-1)}$.

The same approach can be applied to compute the weight updates for the *reset gate* and the *candidate hidden state*.

As conclusion we can say that, GRU works similarly to LSTM but with fewer gates and operations, making it simpler and faster to train. The weight updates follow the same principles as LSTMs but with fewer terms to compute.

.

3.5 Applications of RNNs: natural language processing (NLP), machine translation, time series forecasting

RNNs (Recurrent Neural Networks) are widely used for handling sequential data, which makes them useful for a wide range of real-world applications. One of the most well known areas where RNNs are widely used is *Natural Language Processing (NLP)*. In NLP, RNNs help machines understand and generate human language. For example, they are used in tasks like text generation, where the network generates sentences by predicting the next word based on the previous ones. RNNs are also used in speech recognition, where they take as input audio signals to convert speech into text. Another key application of RNNs is in *machine translation*, where a computer learns to translate text from one language to another. By analyzing the sequence of words in the source language and generating the relative words in the target language, RNNs enable automatic translation services like Google Translate to improve their accuracy. In addition to language-related tasks, RNNs are also extensively used *time series forecasting*. Time series forecasting involves predicting future events based on past data points, and RNNs fit exactly at this because they remember patterns in the data. This application is important in industries such as finance, weather prediction, stock market forecasting, and even healthcare, where predicting future trends based on historical data is important for decision-making.

Chapter 4

Autoencoders

Autoencoders are a type of artificial neural network used to learn efficient, compact data representations, unsupervised.

A compressor takes input data and attempts to compress it into a smaller form and from that compressed version, we should be able to reconstruct the original version of your data.

This network consists of two parts: the *encoder* and the *decoder*. The encoder is responsible for converting the input to a low-dimensional representation, typically referred to as the *latent space* and the decoder in turn takes this compressed data trying to transform it into an output that is most similar with the input. The goal is that the autoencoder learns important features of the data and still minimizes the loss between the original input and reconstructed output. Typically, autoencoders are used for tasks like image denoising or dimensionality reduction.

One of the most important properties of autoencoders is that they are trained not to pay attention to noise or any other information that might be irrelevant and instead put emphasis on predicting important components like original input.

4.1 Unsupervised learning with autoencoders

Autoencoders come from the work of Geoffrey Hinton and his team in the 1980s. Research on neural networks and unsupervised learning had precursors in the work of autoencoders, where they were applied to problems such as dimensionality reduction and feature learning. One of the most significant papers related to Autoencodes, "Reducing the Dimensionality of Data with Neural Networks" — Hinton and Salakhutdinov (2006)[16] showed that deep autoencoders could do much better than traditional methods such as PCA in capturing complex, non-linear patterns in data.

In the recent years, during the revival of neural networks, autoencoders got a bit more popular again. These deep architectures enabled them to learn deeper or more complex data representations. A notable paper: "Auto-encoding Variational Bayes" by Kingma and Welling (2013)[22] dealt with generating new data from a probabilistic perspective using Variational Autoencoders

(VAEs).

In summary, the training of autoencoder is not very different from that of conventional multi-layer perceptrons (MLPs) except for their objectives and structure. In MLPs, the objective is typically to learn how to map inputs to specific outputs (e.g. classification or regression) while in autoencoders the task we want to perform is reconstructing the input itself.

In an autoencoder:

Encoder: It takes input data and runs them through a series of layers which maps it to much smaller dimensions/latent space.

Decoder: Reconstructed the original input from this compressed representation.

The training is done focussig on minimizing the difference between the original input and the reconstructed output (using a loss function for instance mean squared error). On the other hand, MLPs are trained to minimise the error between predictions and actual target labels.

However, autoencoders are unsupervised learning since they do not need labeled data like MLPs. In addition, one of the main differences between autoencoders and MLPs is that autoencoders regularly consist of bottleneck layers (small layers in the middle of the network) which forces learning a more compressed, shorter representation for the data comparing to an MLP. One major difference how autoencoders are trained as compared to traditional MLP is in the training and reconstruction set-up.

4.2 Encoder-decoder architecture

The Encoder-Decoder architecture is the backbone of autoencoders and is f basic importance. This architecture consists of two main parts: the encoder, which compresses the input into a latent space, and the decoder, which reconstructs the input from this compressed representation. We can mathematically describe the process as follows:

Let $\vec{x} \in \mathbb{R}^n$ represent the input data, and let the encoder function be denoted by f_e. The encoder

maps the input to a latent representation $\vec{z} \in \mathbb{R}^m$ (where $m < n$) through the equation:

$$\vec{z} = f_e(\vec{x}) = \phi(\vec{W_e} \cdot \vec{x} + \vec{b_e}),$$

where $\vec{W_e}$ are the weights, $\vec{b_e}$ is the bias, and ϕ is an activation function such as ReLU or sigmoid.

Next, the decoder, denoted by g_d, attempts to reconstruct the original input from $\vec{z}$. The output $\hat{\vec{x}}$ is given by:

$$\hat{\vec{x}} = g_d(\vec{z}) = \psi(\vec{W_d} \cdot \vec{z} + \vec{b_d}),$$

where $\vec{W_d}$ and $\vec{b_d}$ are the weights and biases of the decoder, and ψ is another activation function. The goal of training an autoencoder is to minimize the reconstruction error, by using a loss

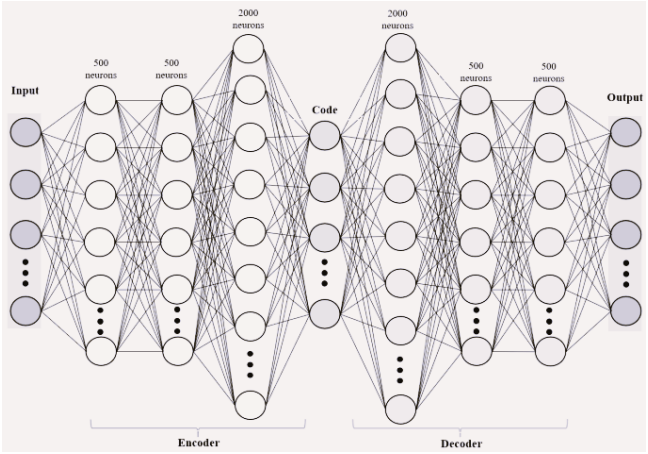


Figure 4.1: Autoencoder
[2]

function like the Mean Squared Error (MSE):

$$E(\vec{y}, \hat{\vec{y}}) = \frac{1}{n} \sum_{i=1}^{n} (\vec{y_i} - \hat{\vec{y}_i})^2.$$

Minimizing this error adjusts the weights $\vec{W_e}$ and $\vec{W_d}$ to ensure that the decoded output $\hat{\vec{x}}$ closely look ike the original input $\vec{x}$.

One important application of the Encoder-Decoder architecture is denoising. In denoising autoencoders, the model is trained not only to reconstruct the input but to *remove noise* from

corrupted input data. The process works by adding noise to the original input $\vec{x}$, resulting in a noisy version $\vec{x_n}$. The encoder receives $\vec{x_n}$ as input, and the decoder tries to reconstruct the clean version $\vec{x}$:

$$\vec{z} = f_e(\vec{x_n}), \quad \hat{\vec{x}} = g_d(\vec{z}).$$

The loss function here is still the MSE, but now it measures the difference between the clean input $\vec{x}$ and the reconstructed output $\hat{\vec{x}}$. The training minimizes this difference, enabling the autoencoder to remove noise from data in practice.

Denoising autoencoders are used in tasks such as image denoising, where an image is corrupted by noise (e.g., Gaussian noise) and the autoencoder learns to reconstruct a clean version of the image. Mathematically, the noisy input $\vec{x_n}$ could be represented as:

$$\vec{x_n} = \vec{x} + \vec{\epsilon},$$

where $\vec{\epsilon}$ is the noise added to the original input $\vec{x}$. The training process encourages the network to discard $\vec{\epsilon}$ while retaining the structure of $\vec{x}$.

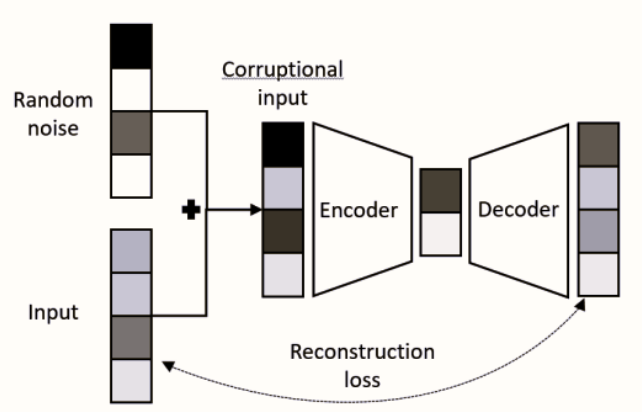


Figure 4.2: Denoising
[15]

Autoencoders are naturally a very useful tool for *compression*, where the encoder reduces the input to a lower-dimensional latent space $\vec{z}$. In this case, the size of $\vec{z}$ is much smaller than the input $\vec{x}$, meaning the encoder captures the most important features of the data in fewer dimensions. The decoder reconstructs the input using only this compressed representation.

The success of the compression is evaluated by how well the decoder can reconstruct the original input from the compressed latent vector. If $\vec{z}$ effectively captures the core structure of $\vec{x}$, the reconstruction will be accurate even though $\vec{z}$ has fewer dimensions than $\vec{x}$.

Compression through autoencoders is used in data storage and transmission. For example, in image compression, an image with high resolution can be reduced to a much smaller latent vector $\vec{z}$ and later reconstructed with minimal loss of quality. The encoder and decoder act as encoding and decoding mechanisms, which can be applied in practical tasks like sending compressed data over networks, as mentioned earlier.

Another core function of the Encoder-Decoder architecture is *dimensionality reduction*. Similar to PCA (Principal Component Analysis), autoencoders reduce the dimensionality of data, but they do so more effectively because they can capture non-linear relationships in the data. The encoder compresses high-dimensional input $\vec{x}$ into a smaller latent space $\vec{z}$, allowing us to work with fewer dimensions while preserving the structure of the data. Let's consider the case where $\vec{x}$ is an image with 1024 pixels, and $\vec{z}$ is a latent vector with only 100 values. The encoder function f_{θ_e} maps $\vec{x}$ to $\vec{z}$, where the most important features of the image are retained. The reconstruction error is minimized by adjusting θ_e and θ_d, ensuring that $\vec{z}$ effectively represents $\vec{x}$ with fewer dimensions. This ability to reduce dimensions while retaining important information is useful in many machine learning tasks, such as feature extraction for classification or clustering.

Dimensionality reduction with autoencoders is widely used in applications like data visualization and exploratory analysis, where reducing data to 2D or 3D helps visualize complex patterns that are not easily understood in higher dimensions.

4.3 Applications of autoencoders

Encoder-Decoder has many applications in practice. And in image and audio processing, autoencoders are frequently used for *denoising*, or removing noise from images/audio while preserving important features. For example, in medical imaging, denoising autoencoders can

remove noise from MRI or X-ray images, improving their visual appearance so that they can be more easily interpreted for diagnostic purposes.

Another key area is that of [-compression]. Image Compression: Autoencoders are widely used within image compression demands, where high-resolution images should be compressed to gift an everyday size for storage or communication. Used for size get back on bandwidth bottleneck such as mobile devices and web apps where we need to reduce the data without losing quality.

Autoencoders are commonly used in the context of large high-dimensional datasets (e.g., genomics or financial data) for *dimensionality reduction*. They speed up the process of training machine learning models by limiting the number of features to be considered, preventing overfitting.

Chapter 5

Generative Adversarial Networks (GANs)

GAN s were popularized by Ian Goodfellow and his team in 2014[9]. The fundamental idea in this breakthrough was two neural networks, the GAN generator and discriminator that was competing its other. The generator creates data, and the discriminator is trained to determine if that data it has just received (from either the real dataset or from itself) is real or generated. The generator gradually becomes better and produces data more closer to real, which the discriminator finds difficult to separate from.

That competitive process is the difference between a GAN and other networks, as both sets of parameters change with the actions of each other. Moreover, GANs still evolve and mature, with several seminal papers that push the state of the art each year even further.

Subsequent to Goodfellow, Radford, introduced the Deep Convolutional GAN (DCGAN) in 2015[33] that made GANs significantly better dealing with image data. This was a historic event for GANs: The function of convolutional layers enabled models to produce developments of far greater detail than ever before, moving towards understanding how pixels are arranged and related in images, which is important for creating new images from scratch.

However, Arjovsky created a more stable training scheme with the use of the Jensen-Shannon divergence to construct discriminator losses for GANs. Introduced in 2017 is the Wasserstein GAN (WGAN)[11], by Arjovsky, which only roughly satisfy an adversarial loss. By using a different distance metric, WGANs stabilize the training process and produce better quality generated outputs. Karras also published on Progressive Growing of GANs that same year where the model could generate high resolution images by starting generation with images sized small and increase progressively in size whilst training.

Each of these has expanded the original GAN framework to improve model stability, efficiency, and capacity for producing higher dimensional data. More recently GANs have been used for a variety of creative and practical applications like generating photo-realistic images, deepfakes or art creations

5.1 Generative and Discriminative Models

Models in data science are typically split into one of two classes — generative and discriminative models. These seem like categories with different angles of tackling the problem of understanding data.

Discriminative models learn how to separate different groups or categories in a dataset by focusing on what makes them different. In formal terms, for a given input x, a discriminative model aims to model the conditional probability $p(y|x)$, where y represents the label of that particular input. This model learns the mapping from input to their corresponding labels directly, where it involves classification tasks. Examples of such models include logistic regression, SVMs, and most neural networks designed for classification tasks These models are trained to minimize a loss function, such as cross-entropy, which is high when a class label y for a given input x prediction is wrong.

Generative models, on the other hand, aim to model the joint probability distribution of the data, $p(x, y)$. From this joint distribution, both conditional probability distributions $p(y|x)$ and $p(x|y)$ can be derived. Obtaining one should not compromise the ability to derive the other.

Generative models moreover not only attempt to model $x \to y$ but also try to model $p(x)$, the original data distribution and thus learning about the structure of that domain. Generative models use a learned distribution to generate new instances of data by sampling from it. At a more technical level, a generative model learns the joint distribution $p(x, y) = p(y|x)p(x)$, where by marginalizing over y, it allows to create new samples of x.

Mathematically, this can be written as:

$$p(x) = \sum_{y} p(x, y)$$

or in the case of continuous data:

$$p(x) = \int p(x, y)\, dy$$

5.2 GAN architecture: generator and discriminator

A GAN is a type of neural network that pits two networks against each other: the *generator* and the *discriminator*. Therefore, both networks have opposite directions in which they are trained but the goals of each side are different.

Generator: The generator G will sample from the same distribution as true data. It receives a random noise vector $\mathbf{z}$ which is an input from the prior distribution $p(\mathbf{z})$ (usually uniform or normal). The generator then converts this random noise into a synthetic sample of data $G(\mathbf{z})$ (to make it similar to real data $\mathbf{x}$)

The discriminator D, on the other hand, is a binary classifier which must output 0 for samples that are real and 1 for samples that are fake. A real data sample x from the true data distribution $p(\mathbf{x})$, or a fake sample generated by the generator G: G(x) The discriminator is supposed to produce a estimation if the input data is real or generated. In a more formal language, it returns $D(x)$ — the probability that the input is real.

The generator is trained to increase the probability of falsely predicting by the discriminator, which means that it tries to generate data that cannot be distinguished from genuine input data—otherwise known as fooling the discriminative model.

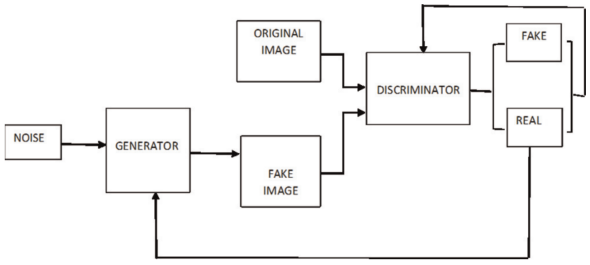


Figure 5.1: General architecture of a GAN [8]

In the case of GANs, we often mention "true data distribution" $p(\mathbf{x})$ a lot but it is quite confusing to understand why because at the end of the day our models can only operate on dataset. The dataset is the one that we have and it consists of real life data points. This dataset, however, is just a small snippet of the huge different sorts of examples that might occur in the real world.

That is, if our dataset contains many cats, it will represent only a small fraction of all possible cat photos in the world. The 'true data distribution' $p(\mathbf{x})$ is refered to the way all of possible actual world data are stretched or distributed not just that few examples we notice in our dataset. Although the model has only seen about this dataset, it tries to learn the underlying pattern or structure that a sample is taken from "the true data distribution".

This concept is quite important in GANs, because the generator must be able to generate completely new data that looks real and not just replicate the training examples. For this, it has to assimilate the global shape of how actual data looks like. The discriminator does his job by determining if a sample is REAL or FAKE, and the generator tries to hack it by generating data which makes the generator to generate real-world like-data who comes from the true data distribution.

Now the model learns to generalize outside of what we trained it on, by exposing itself to the actual data distribution while training and generating more realistic new data.

This concept of a competition between the generator and the discriminator is fundamentally how GANs are trained.

5.3 Training GANs

Training a Generative Adversarial Network (GAN) involves two main parts: the generator G and the discriminator D, each trained with different goals. The discriminator tries to tell the difference between real and fake data, while the generator tries to fool the discriminator by producing fake data that looks real. Let's break down the process into forward pass, backpropagation, and the error functions.

In each training step, the forward pass for both the discriminator and generator happens in the following way:

The discriminator D receives a real data sample x from the dataset. The discriminator outputs a probability $D(x)$ that indicates how real the sample is. The closer $D(x)$ is to 1, the more

confident the discriminator is that x is real.

The generator G takes in a random noise vector z, sampled from a prior distribution $p(z)$ (often normal distribution $\mathcal{N}(0,1)$). Then it produces a fake data sample $G(z)$. This fake data is then passed to the discriminator. The discriminator outputs $D(G(z))$, a probability that indicates how real the fake sample appears. The closer $D(G(z))$ is to 0, the more confident the discriminator is that the data is fake.

GAN training is modeled as a minimax game. The discriminator is trained to maximize its ability to correctly classify real and fake samples, while the generator is trained to minimize the discriminator's ability to distinguish real from fake data.

The discriminator is trained using the following loss function:

$$L_D = -\left(\mathbb{E}_x[\log D(x)] + \mathbb{E}_z[\log(1 - D(G(z)))]\right)$$

The minus symbol in the formula for the discriminator loss L_D is used because the objective is to maximize the discriminator's performance in distinguishing between real and fake data, but most optimization algorithms (like gradient descent) are designed to minimize a function. To convert a maximization problem into a minimization problem (so we can use gradient descent), we apply a negative sign.

Moreover, we must refer to symbol E which shows the *Expected value*. Supose we select a batch of data from the dataset we have to train GAN. Supose this batch cotains n datapoints with real data and and the same amount (n) for fake data.

The first part $\log D(x)$ is the loss for real samples, encouraging $D(x)$ to be close to 1.

For the real data, *Discriminator* calculates the expected value:

$$L_{\text{real}} = \mathbb{E}_x[\log D(x)] = -\frac{1}{n}\sum_{i=1}^{n}\log(D(x_i))$$

The second part $\log(1 - D(G(z)))$ is the loss for fake samples, encouraging $D(G(z))$ to be close

to 0. For the fake data it calculates:

$$L_{\text{fake}} = \mathbb{E}_z[\log(1 - D(G(z)))] = -\frac{1}{n}\sum_{i=1}^{n}\log(1 - D(G(z_i)))$$

The total loss for the discriminator is the sum of the real and fake data losses:

$$L_D = L_{\text{real}} + L_{\text{fake}}$$

The generator is trained to maximize the probability that the discriminator classifies its fake data as real:

$$L_G = -\mathbb{E}_z[\log(D(G(z)))]$$

Here, $\log(D(G(z)))$ is the loss for the generator, which aims to make $D(G(z))$ close to 1, meaning the generator successfully fooled the discriminator.

The discriminator's weights are updated by minimizing the discriminator loss L_D. This is done using gradient descent. The gradients are computed for both real and fake data separately:

$$\frac{\partial L_D}{\partial W_D} = \frac{\partial}{\partial W_D}\left(-\log D(x) - \log(1 - D(G(z)))\right)$$

where

$$\frac{\partial}{\partial W_D}\left(-\log D(x)\right) = -\frac{1}{D(x)} \cdot \frac{\partial D(x)}{\partial W_D}$$

This uses the chain rule, where $\frac{\partial D(x)}{\partial W_D}$ is the derivative of the discriminator's output with respect to its parameters W_D.

And

$$\frac{\partial}{\partial W_D}\left(-\log(1 - D(G(z)))\right) = \frac{1}{1 - D(G(z))} \cdot \frac{\partial D(G(z))}{\partial W_D}$$

Again, this uses the chain rule, and $\frac{\partial D(G(z))}{\partial W_D}$ is the gradient of the discriminator's output on fake data with respect to its parameters.

Here, W_D represents the discriminator's weights, which are updated to make it better at dis-

tinguishing between real and fake samples.

After computing these gradients, the discriminator's parameters W_D are updated using gradient descent. The update rule is:

$$W_{D_{new}} = W_{D_{old}} - \eta \cdot \frac{\partial L_D}{\partial W_D}$$

The generator's weights are updated by minimizing the generator loss L_G. The goal is to fool the discriminator, so the gradients are computed as follows:

$$\frac{\partial L_G}{\partial W_G} = \frac{\partial}{\partial W_G} \left(-\log(D(G(z)))\right)$$

which expands to

$$\frac{\partial}{\partial W_G} \left(-\log D(G(z))\right) = -\frac{1}{D(G(z))} \cdot \frac{\partial D(G(z))}{\partial W_G}$$

As in *Descriminator* $\frac{\partial D(G(z))}{\partial W_G}$ is calculated using chain rule.

W_G represents the generator's weights, which are updated to produce more realistic data.

The update rule is:

$$W_{G_{new}} = W_{G_{old}} - \eta \cdot \frac{\partial L_G}{\partial W_G}$$

In each training iterationhe discriminator receives both real data and fake data and uses the outputs from both to compute its total loss. Both the real and fake data contribute to the discriminator's loss, and this total loss is then used to update its weights W_D After updating the discriminator, the generator produces new fake data. The generator then computes its loss L_G, and updates its weights W_G to minimize that loss, aiming to fool the discriminator more effectively.The discriminator is not updated during this step, it simply provides feedback by evaluating the generator's fake data. This process is repeated over many iterations until both networks reach a balance where the generator creates convincing data and the discriminator cannot easily tell real from fake data.

5.4 Applications of GANs: image generation, text generation, style transfer

GANs (Generative Adversarial Networks) find wide usage across various domains where generation of good quality synthetic data is a requirement. The best-known examples is image generation. This technique learns how to generate realistic images from a dataset of real images. An iconic illustration is the generation of non-existent human faces employed in art and design. Autodesk is also using GANs to generate high-resolution satellite images,create product mockups and even create images that mimic a certain art style.

GANs have also shown promise for text generation. Generating cohesive, meaningful text sequences is difficult because of the highly structured nature's language, but there are improvements such as the SeqGAN that deal with at least this issue. They were also not able generate individual sentences and paragraphs or even whole news articles which, for example could be very useful in creative writing, content production or conversational AI like chatbots.

Although, GANs are also very powerful in this regard such as style transfer, where it learns how a visual style of image can be transferred onto another image. In particular, a GAN can turn a picture into something that appears like a photo of a painting. CycleGAN is a famous type of this sort which can ultimately translate one domain's images to another (horse2zebra, summer2winter, etc.) without using paired training data.

Moreover, GANs can be employed in data augmentation which creates more training samples to help machine learning models perform better and are notably used in areas such as medical imaging where annotated data is scarce. They are also utilized in super-resolution problem, that involves generating sets of images that all transform low res images to high-res version which can be applied in surveillance, satellite imaging and video enhancement.

Chapter 6

Transformers

This chapter will cover basic concepts on transformers, which are the basic deep learning empirical technique, specially in NLP community. We will also see the mathematics behind the Transformer Model, various architecture differences and training over multiple GPUs with particular emphasis on operations in non-passthrough layers. The groudbreaking publication was back in 2017 from Ashish Vaswani, a computer scientist, with the title *Attention Is All You Need*[41] shaping with that way the future of Large Language Models.

6.1 Core Principles of Transformers

At the core of Transformer lies an attention mechanism, which is the most important component and make Transformers a unique architecture. More specific transformers contain a self-attention mechanism, that actually tries to find dependencies between tokens. The rest of this section will focus on how self-attention works and will also explain how the rest of the components frame a transformer.

Imagine we have an input (sentence) which is composed by n tokens (words). First step is to split all tokens and to transform every token into a d-dimensional vector x_i where i is the id of the token in the sequence and d the size of the newly costructed vector.

These vectors are called *Embeddings* and represent the linguistic features for any word of a vocabulary. Nowadays, due to the evolution of NLP techniques a whole family of tools (like Word2Vec, Glove,FastText etc) has been created that are converting words into embeddings. These tools provide an out-of-the-box easy solution for Transformers. Actually they take the word and give you 300 component vector (count changes) on each field that you have about a particular word e.g. “is verb”, or “represents an animal” and so on.

The second step the attention mechanism follows, is to aplly, three weight matrices that are learnable, to each token embeddings. But what are these 3 weight matrices. The first matrix W_Q represents the *query*, that is a question on token, “what information should i attend to”? Next, the *key* matrix W_K is associated with the question “Does this token contains relevant information”. And last matrix *value* are the actual data need to pass to next step. Self-attention

mechanism calculates three new vectors for every token in the sequence referred as Query (Q_i), Key (K_i) and Value (V_i). These vectors are computed by taking a dot product of the input x_i and various learned weight matrices W_Q, W_K, and W_V

$$Q_i = x_i W_Q,\ K_i = x_i W_K,\ V_i = x_i W_V$$

Note that these weights (W_Q, W_K and W_V) are initialized randomly, they are updated as we train the model and that their size is the same as embeddings size. With our Q,K and V we can now apply attention scores. To perform attention for a pair of tokens, their query and key vectors are multiplied to produce the dot product. The idea is that tokens with queries and keys on the same coordinate (high dot product) should attend to each other.

The score of attention between the i-th and j-th token is defined as:

$$AttentionScore(i,j) = \frac{Q_i \dot{K}_j^T}{\sqrt{(d_k)}}$$

Where d_k is the dimensionality of the key vectors. Division by $\sqrt{d_k}$ prevents the dot products from growing too large in magnitude, which can lead to extremely small gradients. This is also known as 'scaled dot-product attention'.

After calculating the attention scores, we normalize them using a "softmax" function as seen below. The softmax function that takes the raw attention scores will result probabilities that are indicating how to focus on each token:

$$\alpha_{ij} = \text{softmax}\left(\frac{Q_i \cdot K_j^T}{\sqrt{d_k}}\right) = \frac{\exp(Q_i K_j^T/\sqrt{d_k})}{\sum_{j=1}^{n} \exp(Q_i K_j^T/\sqrt{d_k})}$$

The *softmax* produces a probability distribution that actually normalizes the attention scores with the particularity of rewarding high attention scores to even more higher probabilities. As, the $\sum_{j=1}^{n} \exp(Q_i K_j^T/\sqrt{d_k})$ in the denominator implies, *softmax* function is applied per row (per-token normalization), representing with that way how much the i-th token should attend

to the j-th word.

Then we compute the output for each token as a 'weighted sum' of all value vectors V_j in the sequence, the self-attention output can be computed as:

$$Output_i = \sum_{j=1}^{n} \alpha_{ij} V_j$$

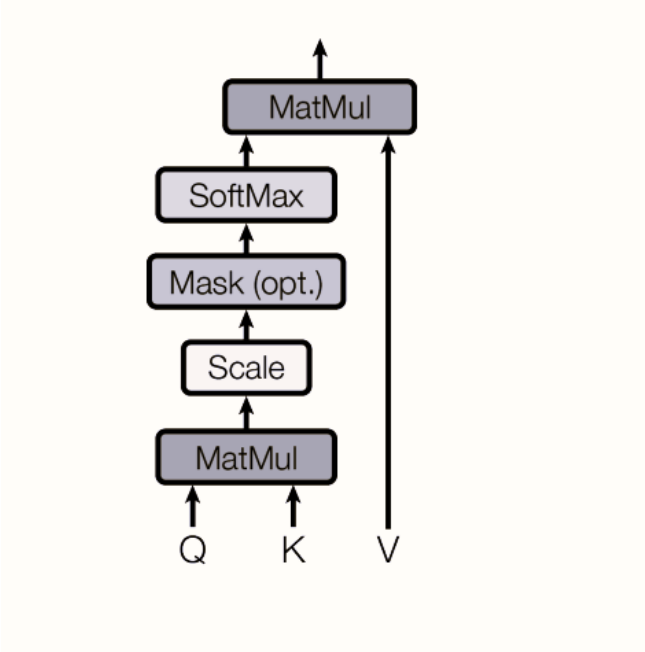


Figure 6.1: Single-Head Attention
[41]

But in practice, transformers employ multi-head attention instead of single-head attention. This boils down to running the attention mechanism h number of times (i.e. with a different set of learned weights) and then concatinating the results,

$$\text{Multi-Head Attention}(Q, K, V) = (\text{head}_1 || \text{head}_2 || \ldots || \text{head}_h)\dot{W}_O$$

Where each attention head is calculated as explained above, and W_O is an additional learned weight matrix applied after concatenating the heads to linearly project them. It worths mentioning that in multi-head attention dimensionality of weights (W_Q, W_K and W_V) are changed to d_k/h, h is the number of heads. Consequently we have multiple smaller weight matrices for

query,key and value.

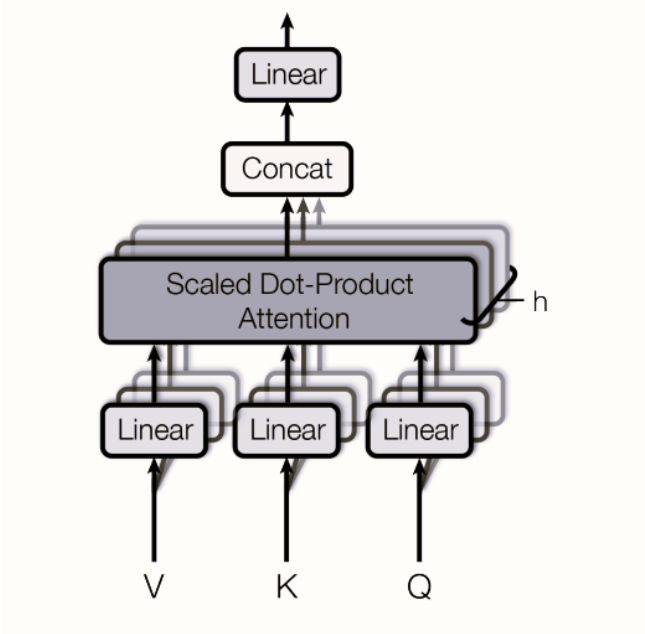


Figure 6.2: Multi-Head Attention [41]

After analyzing the self-attention mechanism let's move and see the "whole picture", how the transformer is constructed. One way transformers tackle the "gradient vanishing" issue is to add the current input to output of the multi-head attention (residual connection) a concept that firstly introduced in Convolutional Network *RestNet*. Morever, another caution with transformers is that there is no concept of order of sequences within the last self-attention layer. To fix alternately, a form of "positional encodings" are included in the input embeddings. These encodings are actually relativistic position encodings so the model knows where each token is in relation to other tokens.

Normally positional encoding is defined via different frequencies of sine and cosine functions.

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right)$$

pos is the position of the token in the sequence i, dimension d is the embedding size.

The last overall input to the Transformer model is the addition of the word embeddings and positional encodings:

$$x_i\text{input} = x_i + PE_i$$

This feeds the model both an informing about what toke and were in the sequence is it.

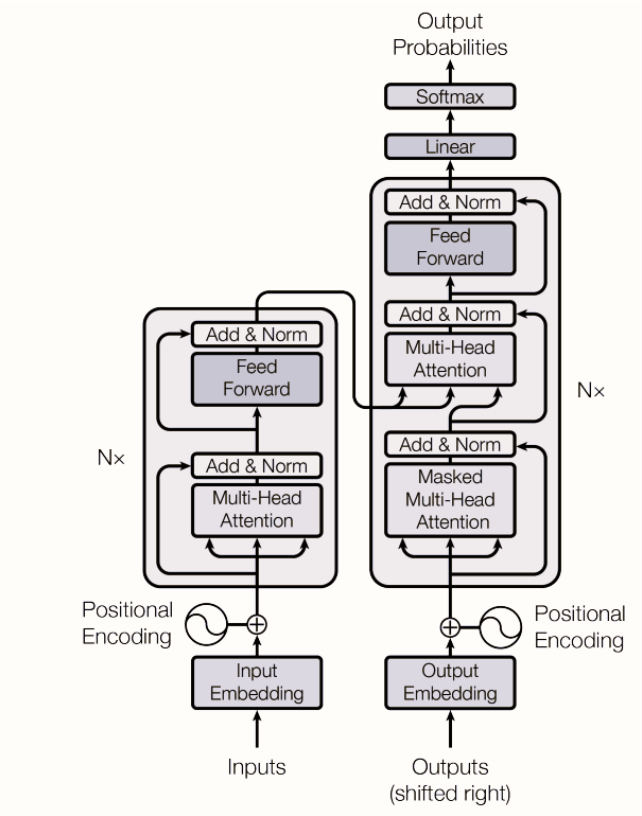


Figure 6.3: Transformer Architecture [41]

Transformers are composed of two main parts. One part is the *encoder* and the other is the *decoder*. In the encoder, the flow of information follows the below path:

- Encoder:
  - Input to Embeddings

  - Positional Embeddings addition
  - Multi-head attention
  - Residual connection
  - Feed Forward Neural Network

Now the decoder part is very similar to how encoder is processing the information, with the key difference that it takes as an input the "ground truth" when transformer is in training mode or the previously predicted token in inference mode. We must also refer to *Masked Multi-Head Attention* which is a copy of the convetional self-attention mechanism that finds attention scores of a token only to previous tokens, all the the other relations are having the value zero.

- Decoder:
  - Input to Embeddings
  - Positional Embeddings addition
  - Masked Multi-head attention
  - Residual connection
  - Input addition
  - Multi-head attention
  - Residual Connection
  - Feed Forward Neural Network
  - Residual Connection
  - Softmax

The final *softmax* function produces a probability distribution to the whole vocabulary. The token with the maximum probability is the predicted one that in training mode is added to the Decoder part as asn additipon to it's input.

6.2 Transformer Architecture Variations

The base Transformer architecture itself will be almost the same, with some versions targeting different kinds of works to do so. Every form of the architecture does a bit differently in order to maximize performance for certain tasks including language modeling, text generation, and long sequence handling.

For example, you may want to look at BERT: Bidirectional Encoder Representations from Transformers[6]

Since BERT looks at all the words in a sentence (context) — before and after, it is basically trying to predict the following word like a language model does. Such a model is known as "bidirectional".

The key mathematical difference in BERT when compared to the original Transformer is its pre-training objective. BERT is trained through a process called masked language modeling(MLM). In the case of MLM, a random word is masked from the input sequence and model is tasked to predict this missing word depending on context. Given the input sequence: $x_1, x_2, \ldots, x_n$ a random word x_j is chosen on which we can create a prediction task such that the model learns to predict this specific word x_j from sampling the rest of the sequeunce.

The loss function for MLM is:

$$\mathcal{L}_{MLM} = -\sum_{j=1}^{n} \log P(x_j | x_1, \ldots, [MASK], \ldots, x_n)$$

BERT is also pre-trained on another task known as "next sentence prediction" (NSP). This architecture involves a model being trained to answer whether or not two sentences occur in succession from the text.

Generative Pre-trained Transformer (GPT)[34]

GPT: Text Generation It leverages "causal (autoregressive) attention" so that every token can only attend to prior tokens. This is great for text generation, since we use the words preceding

a given word to predict that word.

GPT, mathematically — predicts the next word in s sequence by;

$$P(x_{i+1}|x_1, x_2, \ldots, x_i)$$

For this a simple "cross-entropy loss" is used to train it:

$$L_{textGPT} = -\sum_{i=1}^{n} logP(x_{i+1}|x_1, x_2, dots, x_i)$$

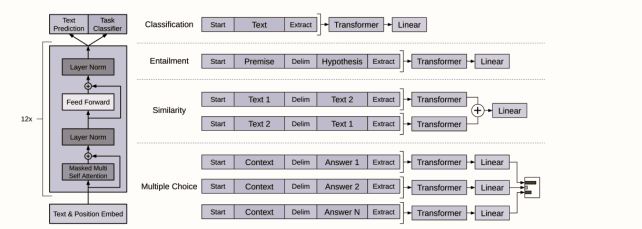


Figure 6.4: GPT
[34]

GPT can now fill in the next word conditioned upon what it has already predicted.

6.3 Training Transformers

The process of training a transformer is to adjust the values of its parameters so that it can minimize some loss function measuring how well the model's predictions match the actual outputs. Backpropagation and gradient descent (common practices in neural network training) are used to do these. In this post, we will delve into the aforementioned points and break down the aspects of training a transformer model.

The loss function captures the error between what our model predicts and the ground "truth/solution labels". The cross-entropy loss is the most popular loss function for tasks like language modeling or text generation. The following $y_1, y_2, \ldots, y_n$ are the sequence of true tokens and $\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_n$ is the predicted probability distribution over possible tokens.

$$\mathcal{L} = -\sum_{i=1}^{n} \log P(\hat{y}_i | x_1, x_2, \ldots, x_n)$$

$P(\hat{y}_i | x_1, x_2, \ldots, x_n)$, is the predicted probability of correct token y_i in position i for input sequence$x_1, x_2, \ldots, x_n$

If it can predict $\hat{y}_i$ correctly, the model decreases at least $-\log P(\hat{y}_i)$ and penalize lower probabilities assigned to the correct token more.

For tasks like machine translation or summarization, the loss can be calculated over sequences in which the model predicts an output sequence $\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m$ using an input sequence $x_1, x_2, \ldots, x_n$. The purpose is to decrease the error value between this sequence and the true sequence.

The above loss is calculated and subsequently the model uses "backpropagation" for updating the parameters (weight matrices W_Q, W_K, W_V, W_O from attention layers and others in the model). The idea is to get the loss smaller and smaller after every step.

In other words, in backpropagation we find this gradient by calculating the pointwise multiplication of this matrix of partial derivatives with a matrix representing $\frac{dy}{\partial \theta_i}$, which contains the upstream "signal" for layer i (see chain rule). In mathematical form, if our loss function is $\mathcal{L}$ and we want to update parameter θ, then the gradient is as follows:

$$\frac{\partial L}{\partial \theta}$$

The gradient gives us how much we expect the loss function to change with respect to a parameter θ.

After computing the gradients, we update our model's parameters with gradient descent. A simple rule update for training with gradient descent:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \frac{\partial L}{\partial \theta}$$

θ_{new} is the new parameter value.

θ_{old} is the old parameter value.

- η is the "learning rate" which control the how large steps are taking in you gradient descent process.

$\frac{\partial \mathcal{L}}{\partial \theta}$ is the gradient with respect to the loss of the parameter θ.

This update rule is used with all the parameters in a transformer model, like the attention weights, feed-forward layer weights or any other learned parameter.

Even though basic gradient descent can solve the problem, there are some advanced versions of it to enhance its convergence and stability, especially for large models like transformer. What are Some of The techniques Used for Optimization

Adam optimizer(Adaptive Moment Estimation,) has become the main optimizer for training transformers. It utilizes two fundamental concepts: " momentum and adaptive learning rates". Adam's update rule is more complicated than the previous gradient descent: This is the formula of *First Moment Estimate* m_t where β_1 is a factor and g_t is the gradient.

First Moment Estimate tries gives the appropriate importance to the direction that gradients moving rather to the gradient value itself.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

For the first iterations performance is might not be satisfactory due to the very small values of m_t so *Bias-Corrected First Moment Estimate* is introduced:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

Second Moment Estimate v_t do the same as First Moment Estimate but only for the magnitudes of the gradients.

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

And similarly *Bias-Corrected Second Moment Estimate* for Seconf Moment Esimate

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Finaly the formula of parameter update using ADAM optimizer retreive from:

$$\theta_{t+1} = \theta_t - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Where m_t is the moving average of the gradients, v_t is the moving average of the squared gradients, $\hat{m}_t$ and $\hat{v}_t$ are bias-corrected estimates of m_t and v_t and ϵ is a small constant added to the denominator to avoid division by zero. Utilizing Adam makes it easier to train the model because it adapts the learning rate for each parameter, making the training efficient and stable, especially on vast datasets and deep models.

6.4 Applications of Transformers

Some popular use case of transformers are *machine translation*, it means it can be used to translate a given text from one language to another. Other models achieved that, were "LSTM" and "GRU", but the performance of transformer architectures was significantly better and directly attributable to the mode of attention.

The translation process is equivalent to learning a mapping function $f : X \to Y$, where X and Y are the tokenized sequence in source language and target language respectively. The Transformer is an approach to training a neural network which can accept any combination of inputs and generate (within reason) any outputs. The provisioning differs from standard copy-learning by model the probability of the target sequence given the source sequence.

Text summarization, as the name implies it is to extract a summary small paragraph from for large content in the form of sentence. It is equivalent to a sequence-to-sequence problem where the input consists in a long document and the output is a shorter version that tries to capture

its main ideas. The transformer is trained to attend to useful parts of the original document when generating the summary. The objective function in text summarization is the same as machine translation: learning to predict the next word in the summary for a given document input.

For *question answering* models, a transformer is tasked with getting an input passage(is the segment of text from which the model is supposed to extract or infer the answer to a given question) of text and then some question over that text. You can treat this competition as a span prediction task for you to predict the passage part that has answer, or an unstructured data natural language processing (NLP) assignment. The problem can be modeled as finding the start and end position of the answer, mathematically.

In *Text generation* given an input, the transformers must generate long coherent and fluently written text. E.g., "GPT (Generative Pre-trained Transformer)" models are meant for conditional text generation, predicting the next token in a sequence given the previous tokens. In inference, autoregressive transformers such as GPT, produce text by sampling words from one at a time based on previously generated words.

In *sentiment analysis* by transformer model, we want to classify whether a given text is positive-negative-neutral. Again, this is generally cast as a "classification problem", such that the input sequence is fed through the transformer and you take the final hidden state h_{CLS} (the hidden state corresponding to the special [CLS] token in models like BERT).

Transformers have become very popular in other fields, such as:

Image Processing: Transformers can process image data as well, by transforming images into patches and considering these patches as tokens in a sequence; models such as "Vision Transformers (ViT)" use this approach. Similar to how it attends to words that appear in the sentence, the model now learns to attend different areas of an image.

Speech Recognition: Wav2Vec model apply transformers to speech data. In this case, we allow the model to process raw audio signals and learn to focus on relevant audio features when it learns to generate text transcripts.

Protein Folding: In biological science, transformers are used in models such as the "AlphaFold" that aims to predict protein structure from amino acid sequences. The attention mechanism enables the model to learn global dependencies along the protein sequence, resulting in very accurate 3D structure predictions.

All these applications are possible by using the strength of the transformer and its attention mechanism which allows that a model concentrate on relevant parts of the input data — be it text, image, speech or biological sequence.

Chapter 7

Conclusion

7.1 Contributions of the Study

The contributions of this study, is towards the understanding and interpretation of neural networks. It provides a well-explained idea of different neural networks like Convolutional NN Recurrent NN etc. It means taking it one layer at a time and understanding the complex parts of these mechanisms in those networks. Therefore, gives readers an insight into how these models operate internally and which applications they have on real applications. While many books will just tell you what to do with neural networks, this work really shows us how the inner workings of these models can be understood. This is very important for understanding why a model decides various aspects that can enhance transparency and validity.

We hope that this research helps narrow the gap between theory and practice, and serves as a more inclusive resource for those interested in understanding how neural networks work and what can they represent in the world.

7.2 Suggestions for Future Work

In addition to explaining how neural networks work at a high level, the study also opens up further research areas. One area for future work will be to enhance the operational efficiency of this model. Training neural networks — especially deep ones like Transformers (image) used in NLP — is computationally expensive and time consuming. Researchers can work on making these models faster without losing accuracy. However, they may utilise techniques like pruning or quantization to reduce their processing demands and yet remain effective. Another important agenda for future is interpretability of a neural network. While this study represents a step forward in demonstrating how we can assist understand neural networks more, there is still work to be done on making them interpretable. Create new tools or methodologies that show the way neural networks make up their decisions, could also make them more understandable for users, especially those with no profound knowledge of the technical details. This can help regain some trust get better utilization from these systems. Last but not least, in the future it may be interesting to investigate how we can combine various different architectures of neural networks

for better results. For instance, combining CNNs and RNNs together for tasks needing both imagery as well sequence data, or mixing GAN and Transformers to generate more advanced data, may spawn even more efficient use cases across industries — including healthcare, finance or entertainment.

With the work that is already out there, will give more leverage to future works in making neural networks robust and power efficient also becoming useful for a lot other things as well for broader applicability across a number of people, industries.

Bibliography

[1] Brilliant.org. Feed forward neural networks – intuition on forward propagation. https://www.analyticsvidhya.com/blog/2021/10/feed-forward-neural-networks-intuition-on-forward-propagation/, 2023. [Online; accessed 28-February-2024].

[2] D. Bank, N. Koenigstein, and R. Giryes. Autoencoders, 2021.

[3] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation, 2014.

[4] G. V. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2:303–314, 1989.

[5] Z. Danko. Neon prescription... or rather, new transcription for google voice. https://blog.google/products/voice/neon-prescription-or-rather-new-transcription-for-google-voice/, 2015. Accessed: 2024-09-17.

[6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

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

[8] D. O. Esan, P. A. Owolawi, and C. Tu. Generative adversarial networks: Applications, challenges, and open issues. In M. Domínguez-Morales, J. Civit-Masot, L. Muñoz-Saavedra, and R. Damaševičius, editors, *Deep Learning*, chapter 2. IntechOpen, Rijeka, 2023.

[9] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial networks, 2014.

[10] A. Graves. Connectionist temporal classification. In *vvv*, 2012.

[11] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. Courville. Improved training of wasserstein gans, 2017.

[12] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, and A. Y. Ng. Deep speech: Scaling up end-to-end speech recognition, 2014.

[13] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition, 2015.

[14] D. O. Hebb. *The Organization of Behavior: A Neuropsychological Theory*. John Wiley & Sons, New York, 1949.

[15] H. Hernández, E. Alberdi, A. Goti, and A. Oyarbide-Zubillaga. Application of the k-prototype clustering approach for the definition of geostatistical estimation domains. *Mathematics*, 11(3), 2023.

[16] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[17] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[18] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, 1982.

[19] A. Jain. All about convolutions, kernels, features in cnn. *Medium*, 2024.

[20] M. I. Jordan. Attractor dynamics and parallelism in a connectionist sequential machine, 1986.

[21] J. Kafunah. Backpropagation in convolutional neural networks. *DeepGrid*, 5 September 2016.

[22] D. P. Kingma and M. Welling. Auto-encoding variational bayes, 2022.

[23] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems*, 25, 01 2012.

[24] A. Kurenkov. A brief history of neural nets and deep learning. *Skynet Today*, 2020.

[25] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, Winter 1989.

[26] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. In *Intelligent Signal Processing*, pages 306–351. IEEE Press, 2001.

[27] Li, X., Tang, J., Zhang, Q. Power-efficient neural network with artificial dendrites. https://doi.org/10.1038/s41565-020-0722-5, 2020. [Online; accessed 20-February-2024].

[28] W. S. McCulloch and W. H. Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 1943.

[29] M. Minsky and S. Papert. A review of "perceptrons: An introduction to computational geometry" by marvin minsky and seymour papert. *Inf. Control.*, 17:501–522, 1970.

[30] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks, 2013.

[31] pluralsight.com. Lstm versus gru units in rnn. `https://www.pluralsight.com/resources/blog/guides/lstm-versus-gru-units-in-rnn`, 2020. [Retrieved 22:31, Nov 7, 2024].

[32] R. Qayyum. Introduction to pooling layers in cnn. *Medium*, 2022.

[33] A. Radford, L. Metz, and S. Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks, 2016.

[34] A. Radford and K. Narasimhan. Improving language understanding by generative pre-training. 2018.

[35] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[36] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[37] S. Saha. A guide to convolutional neural networks — the eli5 way. *SaturnCloud*, 2018.

[38] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv 1409.1556*, 09 2014.

[39] H. Singh. How do computers store images? *Analytics Vidhya*, 2023.

[40] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[41] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need, 2023.

[42] P. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Science. Thesis (Ph. D.). Appl. Math. Harvard University.* PhD thesis, 01 1974.