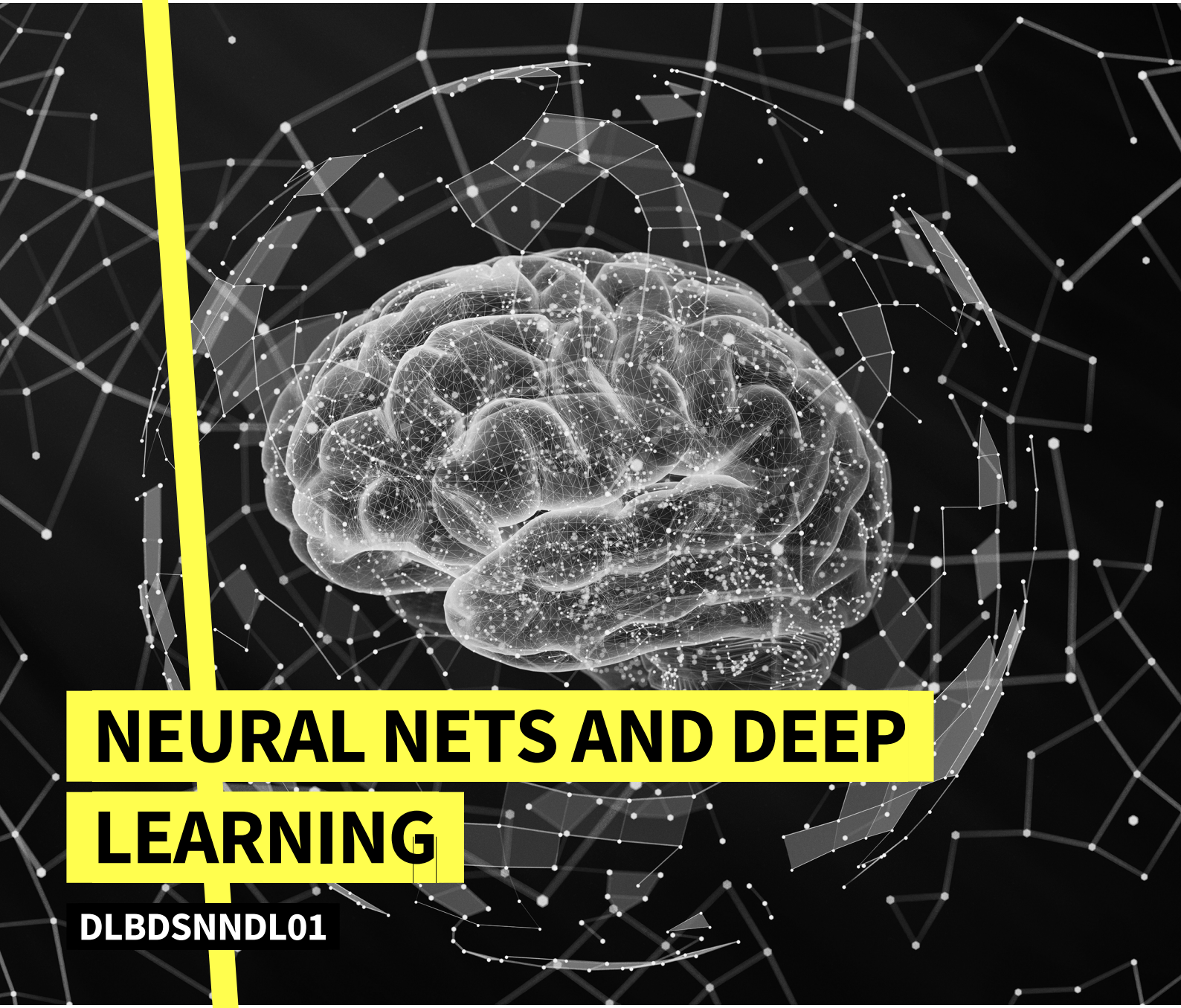
Course Book



NEURAL NETS AND DEEP LEARNING

DLBDSNNDL01

iu
INTERNATIONAL
UNIVERSITY OF
APPLIED SCIENCES

NEURAL NETS AND DEEP LEARNING

MASTHEAD

Publisher:
IU Internationale Hochschule GmbH
IU International University of Applied Sciences
Juri-Gagarin-Ring 152
D-99084 Erfurt

Mailing address:
Albert-Proeller-Straße 15-19
D-86675 Buchdorf
media@iu.org
www.iu.de

DLBDSNNDL01
Version No.: 001-2023-0130
Evis Plaku

MODULE DIRECTOR

PROF. DR. THOMAS ZÖLLER

Mr. Zöller teaches in the field of data science at IU International University of Applied Sciences. He focuses on the fields of advanced analytics and artificial intelligence and their key role in digital transformation.

After studying computer science with a minor in mathematics at the University of Bonn, Mr. Zöller received his doctorate with a thesis in the field of machine learning in image processing. This was followed by several years of application-oriented research, including time spent at the Fraunhofer Society. Throughout his professional career, Mr. Zöller has worked in various positions focusing on the fields of business intelligence, advanced analytics, analytics strategy, and artificial intelligence, while also gaining experience in the areas of defense technology, logistics, trade, finance, and automotive.

TABLE OF CONTENTS

NEURAL NETS AND DEEP LEARNING

INTRODUCTION

WELCOME

SIGNPOSTS THROUGHOUT THE COURSE BOOK

This course book contains the core content for this course. Additional learning materials can be found on the learning platform, but this course book should form the basis for your learning.

The content of this course book is divided into units, which are divided further into sections. Each section contains only one new key concept to allow you to quickly and efficiently add new learning material to your existing knowledge.

At the end of each section of the digital course book, you will find self-check questions. These questions are designed to help you check whether you have understood the concepts in each section.

For all modules with a final exam, you must complete the knowledge tests on the learning platform. You will pass the knowledge test for each unit when you answer at least 80 percent of the questions correctly.

When you have passed the knowledge tests for all the units, the course is considered finished and you will be able to register for the final assessment. Please ensure that you complete the evaluation prior to registering for the assessment.

Good luck!

BASIC READING

Aggarwal, C. C. (2018). *Neural networks and deep learning: A textbook*. Springer.

Chollet, F. (2017). *Deep learning with Python*. Manning Publications.

Géron, A. (2019). *Hands-on machine learning with Scikit-Learn, Keras and TensorFlow: Concepts, tools, and techniques to build intelligent systems* (2nd ed.). O'Reilly.

FURTHER READING

UNIT 1

Kaur, M. (2021). *An introduction to machine learning in a nutshell.* 10th International Conference on System Modeling & Advancement in Research Trends (SMART) System Modeling & Advancement in Research Trends (SMART), pp. 17–22.

Kriegeskorte, N. & Tal, G. (2019). Neural network models and deep learning. *Current Biology, 29*(7), 231–236.

UNIT 2

Isomura, T., & Friston, K. (2020). Reverse-engineering neural networks to characterize their cost functions. *Neural Computation*, 32(11), 2085–2121.

Lu, J. (2022). *Gradient descent, stochastic optimization, and other tales.*

UNIT 3

Bejani, M. M., & Ghatee, M. (2021). A systematic review on overfitting control in shallow and deep neural networks. *Artificial Intelligence Review: An International Science and Engineering Journal*, *54*(8), 6391–6438.

Watanabe, S., & Yamana, H. (2022). Overfitting measurement of convolutional neural networks using trained network weights. *International Journal of Data Science and Analytics,* 1–18.

UNIT 4

Cong, S., & Zhou, Y. (2022). A review of convolutional neural network architectures and their optimizations. *Artificial Intelligence Review: An International Science and Engineering Journal*, 1–65.

Zhang, H. (2022). A review of convolutional neural network development in computer vision. *EAI Endorsed Transactions on Internet of Things*, *7*(28), 1–11.

UNIT 5

Alqushaibi, A., Abdulkadir, S. J., Rais, H. M., & Al-Tashi, Q. (2020). *A review of weight optimization techniques in recurrent neural networks*. 2020 International Conference on Computational Intelligence (ICCI), Computational Intelligence (ICCI), pp. 196–201.

Kaur, M., & Mohta, A. (2019). *A review of deep learning with recurrent neural network.* 2019 International Conference on Smart Systems and Inventive Technology (ICSSIT), pp. 460–465.

LEARNING OBJECTIVES

Intelligent machines have long been a dream of humanity. The possibility of developing a computer system that learns by itself is not solely associated with sci-fi movies but is now part of a global momentum marking the unprecedented rise of artificial intelligence, machine learning, and deep learning.

The objective of the **Neural Nets and Deep Learning** course book is to provide a path that takes you from understanding fundamental concepts of machine learning, neural networks, and deep learning to starting to build your own projects that solve complex tasks intelligently.

You will be introduced to core machine learning approaches before carefully investigating neural networks and deep learning methodologies. You will expand your knowledge and gain practical experience with the architectures and topologies of the most common neural networks while analyzing their fundamental building blocks.

You will learn state-of-the-art techniques to fine-tune neural networks to enhance their performance and increase their efficiency while addressing challenging problems in a principled manner. You will broaden your understanding and gain more practical experience as you experiment with convolutional and recurrent neural networks.

This overview of frameworks, techniques, programming, logic, and a quest of always asking “why” will prepare you with the necessary tools to begin a journey that not only equips you with the skills to understand how intelligent computer systems operate, but also to start building your own.

UNIT 1

INTRODUCTION TO NEURAL NETWORKS

STUDY GOALS

On completion of this unit, you will be able to ...

- understand what machine learning is and which problems it aims to solve.
- explain what neural networks are and outline their fundamental building blocks.
- distinguish between deep and shallow neural networks.
- analyze various machine learning categories, such as supervised learning, unsupervised learning, semi-supervised learning, and reinforcement learning.
- develop a simple neural network used to classify images of hand-written digits.

1. INTRODUCTION TO NEURAL NETWORKS

Case Study

Anne, a neuroscientist, is fascinated by the human brain. She studies the ability of the brain to process thousands of impulses from the outside world and almost instantaneously translate those stimuli into meaningful information and insight. She is dazzled by the ability of the brain to process visual information, recognize patterns, and change and adapt to unknown situations based upon previous experiences. Anne always wonders how our brains guide us to run toward an ice-cream shop and to run away from a fast-approaching car, and in both cases to do it without much effort. Even in new situations, the brain seems to know, as if from experience, what is the right course of actions.

Anne's friend, Lukas, is a software engineer. As Anne discusses the complex abilities of the human brain, he introduces her to the fast growing and impactful field of machine learning. Lukas draws similarities between the human brain and neural networks in their ability to solve difficult tasks that require intelligence. Computers, too, can learn from past experiences, without needing to be told what to do through a rigid set of rules, but rather by continuously adapting and improving their performance as they gather more information.

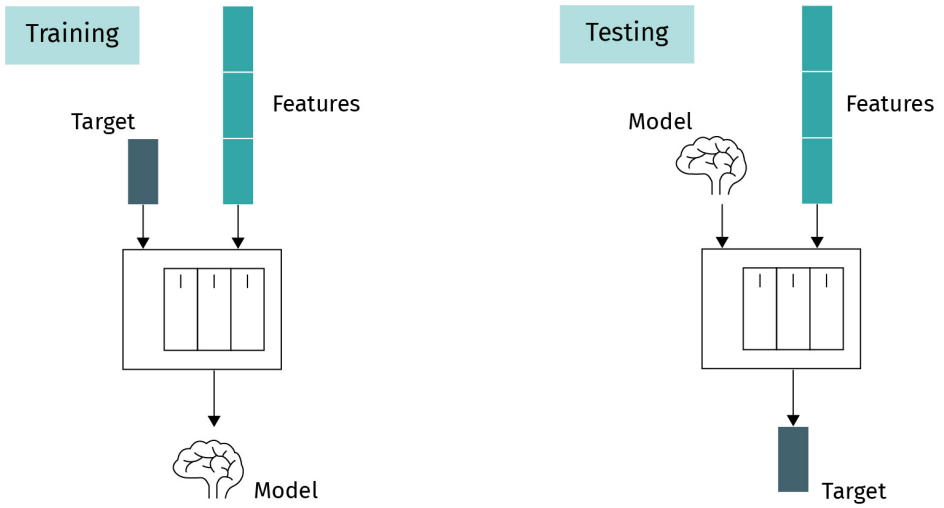
Anne is intrigued by the ability of computers to mimic and expand the capabilities of the human brain. She wants to understand how a computer can learn to make informed decisions based on the input data it receives.

Deep learning
a machine learning framework that uses neural networks to learn from experience

Machine learning is a popular and important branch of artificial intelligence that relies on the use of algorithms and data to solve complex tasks. A machine learning system learns through experience and gradually improves its performance. **Deep learning**, a subfield of machine learning aims to learn from large amounts of data relying on algorithms inspired by the functionalities of the human brain. The goal is to process high-volume data received as input and to apply carefully designated algorithms to recognize underlying relationships and draw out meaningful patterns.

Machine learning typically begins with a training phase during which the algorithm aims to explain the characteristics of the training data by building a model that best fits them. This model is then used to produce an output, typically when encountering data that it has not observed before. The figure below provides an illustration.

Figure 1: The Machine Learning Model

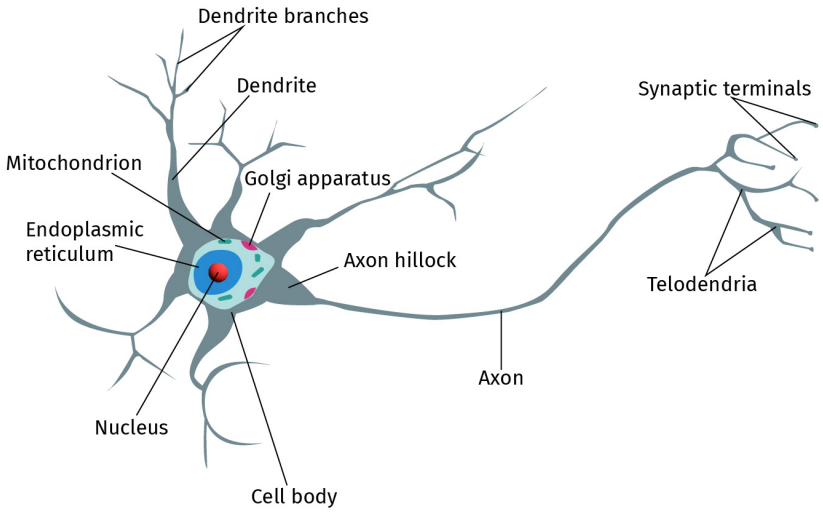


Source: Evis Plaku (2023) based on Amer, (2022).

1.1 The Biological Brain

The ability of the human brain to process complex information has inspired whole fields of computer sciences. Artificial neural networks emerged as a learning paradigm that aims to implement a simplified model of a biological neuron (McCulloch & Pitts, 1943). The biological neuron, the fundamental unit of the brain, is composed of a cell body containing the nucleus, in addition to several branching extensions, known as dendrites, and a very large extension, called the axon. The human brain contains billions of neurons, each receiving short electrical impulses called signals. Synapses allow neurons to communicate with one another by transmitting messages. A neuron fires a signal when it receives a sufficient number of messages from the set of interconnected neurons (McCulloch & Pitts, 1943). An illustration of a neuron is provided below.

Figure 2: Anatomy of a Biological Neuron



Source: BruceBlaus (2013). CC BY 3.0.

Individual neurons are part of a vast network containing billions of neurons, each connected to thousands of others. Seemingly simple neurons, when combined as parts of a large network, can carry out complex computations.

Early advances of machine learning and deep learning were focused on mimicking the abilities of the human brain. These resemblances, however, are mostly conceptual. Artificial neural networks are not strict representations of biological neurons, similar to how airplanes, although inspired by birds, do not flap their wings when they fly.

1.2 Building Blocks of Neural Networks

Artificial Neural Network
An ANN is a mathematical model used to recognize patterns and solve complex problems.

An **artificial neural network** (ANN) is a computing system that aims to identify underlying patterns and relationships in a set of input data. It has the ability to adapt well to changing input and to gradually improve its performance as it gains more information. To achieve this objective, a neural network is trained to identify a mapping between input data to the desired output. The following are core components of an artificial neural network.

The Artificial Neuron

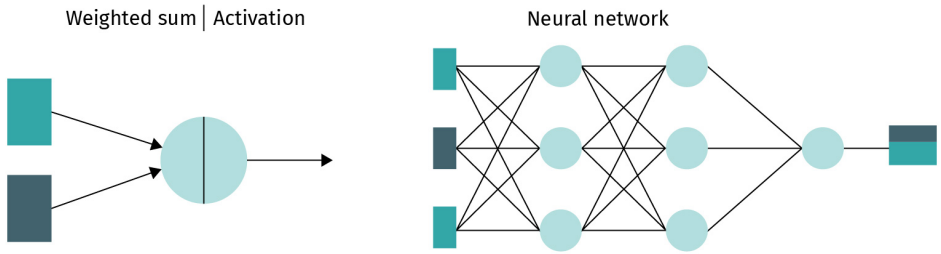
An artificial neuron is the central element of a neural network, often known as a node or a unit. Each neuron receives input from other nodes or external sources. These signal inputs are not of equal importance. Their relevance relative to other inputs is measured and defined through multiplicative values known as weights. A neuron considers the weighted sum of all receiving inputs and transforms it into an output using a special function commonly known as an activation function. An activation function is a critical part of the

design of a neural network as it decides if a specific neuron should be activated or not. It helps the network to learn complex patterns in the input data by making possible the transformation of input signals of previous neurons to outputs that will be used by subsequent neurons in the network.

Layers: The Building Blocks of Deep Learning

Layers are the core building blocks of a neural network. Their purpose is to process input data and output that data into a form that is more useful and beneficial for the task at hand. A neural network contains many layers. Typically, each network has three types of layers: an input layer, an output layer, and at least one intermediate layer often known as hidden layer(s). Each layer extracts a representation out of the input data and feeds it to subsequent layers. Layers are combined into a network, similar to how LEGO bricks are combined into forming complex structures. The figure below provides a simple illustration of a single neuron and a neural network.

Figure 3: A Simple Representation of a Neuron and a Network



Source: Evis Plaku (2023), based on Amer (2022).

The Model

Layers are combined into a network to allow it to map and represent the interactions between input and output data. Layers are most commonly arranged into a linear stack that maps input to output, but a wide variety of network architectures and topologies exist. The learning model defines the methodology used by the network to identify how to map input data to output targets. This model is then used to categorize data that it has not observed before. The efficacy of a model is based upon these results.

Loss Functions and Optimizers

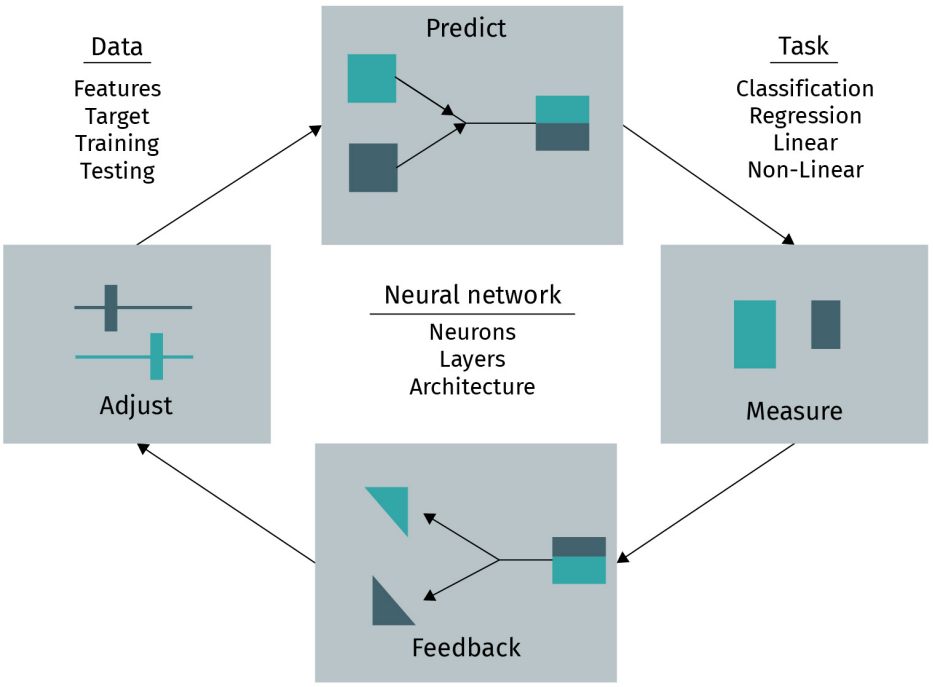
Equipping a neural network with learning capabilities requires careful consideration of two other fundamental components:

1. The objective of the neural network is to minimize the loss function during the learning process.
2. The optimization method determines how the neural network will be updated to accommodate the changes suggested by the loss function and, thus, enable learning.

The choice of the loss function and the optimizer function is of extreme importance in the success rate of the neural network.

The relationship between the core parts of a neural network is visually presented in the figure below. The neural network is fed input data together with their relative weight importance. A set of layers chained together transforms this input and maps it to an estimated prediction. The loss function compares the prediction of the network with the true value of target output data and yields a measure of how well the network can match the expected results. This estimated score is then used by the optimizer to update the network weights and adjust their relevance accordingly. This learning procedure is a continuous cycle of the above steps until the loss function is minimized, thus allowing the network to learn a model that explains the data well.

Figure 4: Neural Network Learning Process



Source: Evis Plaku (2023), based on Amer (2022).

Implementing our First Neural Network

Let's start to get some hands-on experience by implementing our first neural network. The ability to quickly recognize handwritten digits comes easily to humans. Yet, it is an important and difficult task for a computer. Trying to write a series of rigid rules to capture the various shapes of digits would be an almost impossible quest. However, like humans, neural networks, can learn how to recognize handwritten digits by experience. We will expose a neural network to thousands of examples of handwritten digits and train it to learn to identify unique features and qualities that it can use for effective classification.

To facilitate implementation, we will use a popular Python library known as Keras. Because Keras contains a lot of built-in functionalities that we can take advantage, it is able to provide an interface that facilitates the process of building artificial neural networks.

Our objective is to classify images of handwritten digits into ten categories (numbers from zero to 9). We will use the classic dataset MNIST, which contains a large set of 60,000 images that can be used during the training procedure, and an additional set of 10,000 images that we will use to test the efficiency of our model (Deng, 2012). The figure below provides an illustration.

Figure 5: MNIST Dataset Sample Digits

Source: Yann LeCun and Corinna Cortes(n.d.). CC 3.0.

The MNIST dataset is preloaded in Keras. When using it, we will divide the entire dataset into images used for training (`train_images` and `train_labels`) and images used for testing (`test_images` and `test_labels`). The annotated labels represent the true classification of handwritten digits, i.e., the class they belong to (e.g., our figure contains numbers 4, 0, 7, 3).

Loading the MNIST dataset in Keras

```
from keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels)
                              = mnist.load_data()
```

Note that there is a one-to-one mapping between each image and its true label category, i.e., the class the image belongs to.

Our approach is simple: First, we will use the training data as input to the neural network. The objective of the network is to identify underlying patterns and relationships and to be able to associate images with their labels. Once the learning procedure is complete, we will ask the network to provide a classification for images that it has not encountered before, that is, the testing images. We will verify the accuracy of our network by matching the classifications outputted by the network with the true labels of the test data.

Building the model

A key component of a neural network is the layer, a module that processes the data it receives by extracting representations of them and outputing the data in a more meaningful representation. Taking advantage of Keras' built-in functionalities, we can import and use models and layers that are already implemented. In particular:

```
from keras import models
from keras import layers

network = models.Sequential()
network.add(layers.Dense(512, activation='relu',
                         input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))
```

We are using a sequential model of dense layers. In this model each layer is fully connected to its neighbor layer, that is to say, all neurons of a layer are connected to all neurons of the neighbor layer. The input layer contains 784 neurons. It takes as input image data encoded with 28 by 28 pixels. The hidden layer contains 512 neurons, as observed in the above code. The activation function used is rectified linear activation function (ReLU) which is a linear function that will output the input data directly if it is positive and will output zero otherwise. The rectified linear activation function (ReLU) is widely used in neural networks since it is easy to use and achieves good performance in practice. Considering that our objective is to map each digit to one of the ten possible categories, then the second layer will return an array of ten probability estimations to measure that. A function known as "softmax" is used in such cases to convert a vector of numbers to a vector of probabilities. Specifically, it outputs ten values, one for each class. The output is normalized, that is, the sum of all probabilities equals one.

The compilation steps

Before making the network ready for training, we choose the loss function, the optimizer, and the metrics, as discussed previously. These elements form the compilation steps. RMSprop optimizer is an optimization technique commonly used in neural networks. Its objective is to reduce the overall loss and improve the accuracy of the classification by continuously modifying weights and other attributes of the network.

Categorical cross entropy
a loss function used for tasks where one instance belongs to one of many possible categories

For the loss function, we are using **categorical cross entropy** which is typically used for multi-class classification models, as it is our case (images will be classified to one of the ten possible categories). The loss function will measure how well our neural network is modeling the training data. We will monitor the efficacy of our model using the "accuracy" metric, that is, the fraction of the images that were classified correctly. All the above can be easily implemented, as follows:

```
network.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])
```

Pre-processing the input data

Before training the model, it is important to pre-process the input data so it matches the form required by the network. We will reshape the gray-scale image data in the form expected by the network and normalize them. Our training images are stored in an array of shapes (60000, 28, 28) that uses integer data in the [0, 255] interval to represent pixel brightness. We will transform it into an array of the same shape, but we will store floating point numbers in the interval [0, 1]. This can be achieved by the following Python code:

```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
```

As a final pre-processing step, we encode the labels. Since artificial neural networks understand only numbers and not categories, we convert categories to numbers. Through this process, each label of our input data is assigned to a unique integer value, as shown below by taking advantage of Keras built-in functionalities:

```
from tensorflow.keras.utils import to_categorical
train_labels = to_categorical(train_labels)
test_labels  = to_categorical(test_labels)
```

Training the model

We are now ready to train our first neural network. Keras has a built-in functionality that allows it to fit the model to the training data, that is, it investigates and identifies underlying relationships in the training data so it can build a model that best explains input data. To train the model, we use the method “fit” which takes as parameters the image input data and their associated labels, in addition to the number of epochs and the batch size. Epochs denote the number of passes that our neural network has completed in the entire training dataset. Batch size determines the number of training examples used in one iteration. In our case, we are setting the number of epochs to five and are feeding input data 128 images at a time.

```
network.fit(train_images, train_labels, epochs=5,
            batch_size=128)
```

Expected output:

```
Epoch 1/5
469/469 [==============================]
loss: 0.2523 - accuracy: 0.9273
Epoch 2/5
469/469 [==============================]
loss: 0.1007 - accuracy: 0.9698
```

```
Epoch 3/5
469/469 [==============================]
loss: 0.0671 - accuracy: 0.9798
Epoch 4/5
469/469 [==============================]
loss: 0.0483 - accuracy: 0.9858
Epoch 5/5
469/469 [==============================]
loss: 0.0369 - accuracy: 0.9888
```

As observed, this simple neural network quickly achieves an accuracy of 98.88 percent on the training data. Our objective, however, is to test the performance of our model in a separate set of data, which we denoted as "test data." To do so, we call the method evaluate and feed it as parameters the test images and their associated labels to measure the performance of our model.

```
test_loss, test_acc = network.evaluate(test_images,
                                       test_labels)
print('test_acc:', test_acc)
```

Expected output

```
test_acc: 0.9785
```

As one can observe, our model performs well on the testing data, scoring an accuracy of 97.85 percent. It is common for machine learning models to perform slightly better on the data used during the training procedure in comparison to data used during testing.

This illustrative example shows that with only a few lines of Python code we can build and train a simple neural network that can be used to solve an important problem, such as efficiently classifying handwritten digits.

1.3 Deep Versus Shallow Networks

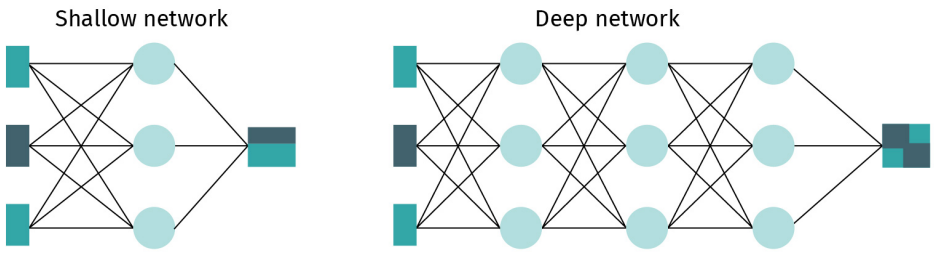
Neural networks can be built using various structures and topologies. A network with a hidden layer is considered a shallow network, in contrast to a deeper neural network that contains several hidden layers. A network with one single hidden layer can produce reasonable results for a variety of problems. Shallow networks can even be effective when modeling complex functions if they have a sufficient number of neurons.

Deep networks, however, have a higher parameter efficiency. Given the same amount of training data, a deep network is typically able to achieve a better performance while using exponentially fewer neurons than a shallow network. In addition, many practical problems contain data that are structured and hierarchical. Deep networks take advantage of their many layers to decompose the problem at hand into smaller portions, thus, learning

the model more quickly and efficiently. An illustration by Géron (2019) asks you to imagine that you've been asked to draw a forest using some drawing software but you are not allowed to copy and paste. It would require a tremendous amount of time and patience to draw each leaf individually, and then to arrange leaves into branches, branches into trees, and finally, to arrange the trees into a forest. But if you were allowed to copy and paste some features, you would be able to take advantage of the repetitive structure of the forest and you would be drawing it in no time (Géron, 2019).

Similarly, a deep network containing several hidden layers can create various representations of the input data at each layer, which can be useful for learning. Moreover, deep networks allow the breakdown of various aspects of the learning procedure into smaller and more manageable chunks. This allows the network to converge faster to an optimal solution, and it also improves the ability of the network to efficiently estimate previously unseen data. The figure below provides a simplified illustration of two types of networks.

Figure 6: Shallow Versus Deep Networks

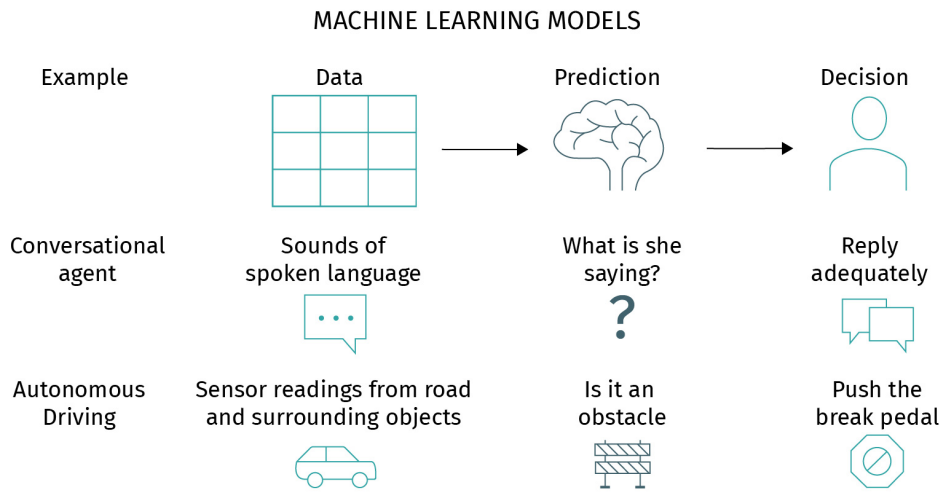


Source: Evis Plaku (2023).

Machine Learning Categories

Machine learning is a broad field. Though learning can take place in many different forms, three main phases are usually observed: (1) the computer agent receives data as input, (2) a prediction is made using the trained model, and (3) a decision or an action follows. An illustration is provided in the figure below.

Figure 7: Machine Learning Categories



Source: Evis Plaku (2023).

According to Géron (2019), learning categories are often defined based on the following:

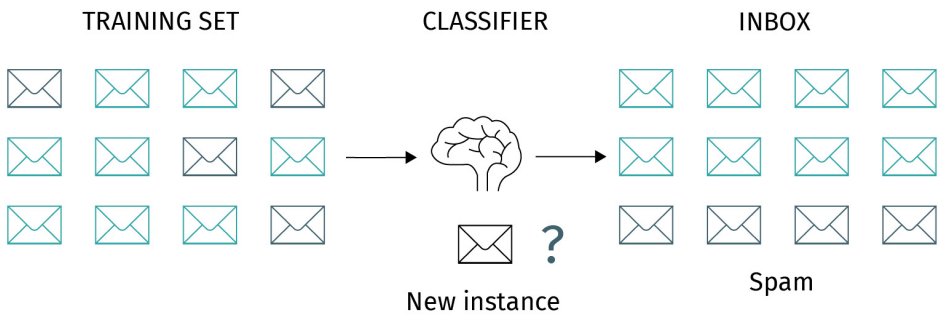
- how the computer system is trained. Four major categories known as supervised learning, unsupervised learning, semi-supervised learning, and reinforcement learning differ in the amount and type of supervision provided.
- how the computer system can learn. Considering that input data are often high-volume, two major types of learning take place known as batch and online learning. In batch learning, the computer system requires all training data for learning to take place. In online learning, new data are incrementally fed into the learning system, either individually or in small chunks, known as mini batches.

1.4 Supervised Learning

Supervised learning maps input data to categories by learning a model based on labeled training data

Supervised learning is the most common case. Given a set of example training data, the objective is to map input data to known targets, often annotated by a (human) supervisor to denote the expected results. Most examples of deep learning, such as image classification, speech recognition, separating spam emails, and others, fall into the category of supervised learning. The figure below provides an illustration.

Figure 8: Supervised Learning: Classification

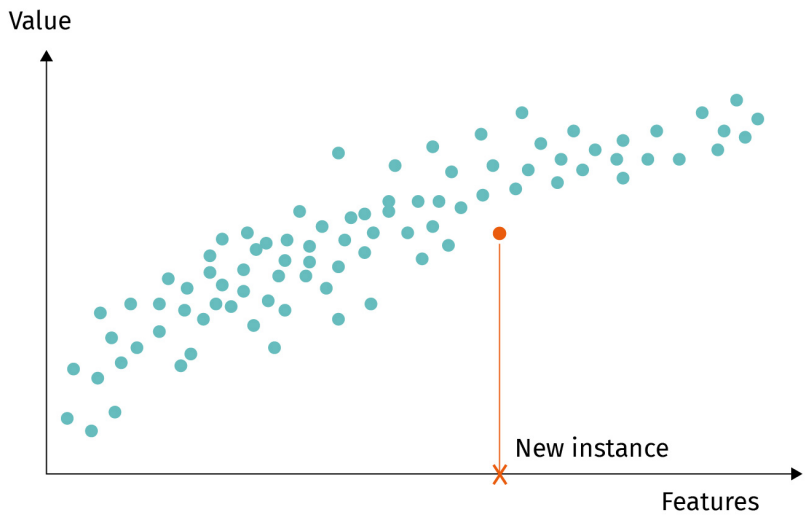


Source: Evis Plaku (2023).

Classification is a typical supervised learning task. The spam filter is a good example of such a problem. A computer system is trained with thousands of emails along with their class category, i.e., spam or not spam. The objective of the system is to correctly classify new emails and decide if they should move to the inbox folder or be flagged as spam.

Another typical task of supervised learning is known as regression. The goal is to predict a numerical value, for example, the price of a house, based on a set of features (e.g., location, number of rooms, and area). Training the system requires feeding the learning algorithms with a vast number of examples of house prices and the associated properties for each house. Once trained, the system will produce a model that can be used to effectively estimate the price of a house not encountered before, given its set of features. The figure below provides an illustration.

Figure 9: Supervised Learning: Regression



Source: Evis Plaku (2023), based on Géron (2019).

Other important supervised learning algorithms include:

Linear regression

Linear regression is a well-known algorithm, widely used in statistics and numerous machine learning tasks. Linear regression models aim to identify a linear relationship between the input features and the output data.

Logistic regression

Logistic regression is a classification algorithm that maps input variables to a discrete outcome. It is commonly used for binary outcomes (e.g., to decide if sample data fits better in one category or in another).

K-nearest neighbors

Supervised learning classifiers rely on the notion of proximity to make predictions. The key assumption is that similar data inputs will be found close to one another. K-nearest neighbors can be used both for classification and regression.

Support vector machines (SVM)

SVMs are a learning model that classies data into categories by performing non-linear classification. Data instances are considered points in space and the objective is to identify groups of input data with maximum distance to the decision boundary.

Decisions trees and random forests

Decision trees classify input data by branching them through a series of binary "this or that" conditions similar to a flowchart. Random forests are built from a series of decision trees and utilize ensemble learning, i.e. They address complex problems by combining many classifiers.

Neural networks

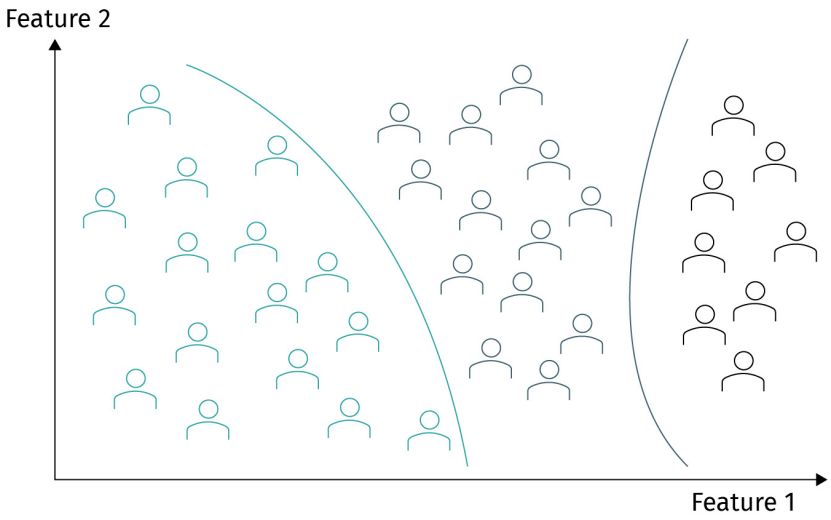
Neural networks aim to identify underlying relationships and patterns in the input data through a process that mimics certain aspects of the way the human brain operates.

Unsupervised Learning

Unsupervised learning uses machine learning algorithms to find structure in unlabeled data

As the name suggests, **unsupervised learning** algorithms do not require human (or otherwise) help to label the input data used during the training procedure. Rather, the system learns without a teacher. These algorithms work by discovering hidden patterns or by grouping the data into a set of related items while evaluating their properties. Unsupervised learning algorithms are commonly used in the field of data analytics for purposes of data visualization or to better understand the data at hand. The figure below provides an illustration.

Figure 10: Unsupervised Learning: Clustering



Source: Evis Plaku (2023).

Imagine you have developed a brand-new website and are eager to measure visitors' traffic and discover potential target groups of similar visitors. Without any prior help or human intervention, an unsupervised learning algorithm can take as input your data and identify common patterns or groups, such as young readers who usually access your website in the evening, or technical blogs that are usually read by people with a higher education degree. An intuitive example of such clustering is represented in the figure above. Common categories of unsupervised learning algorithms include the following:

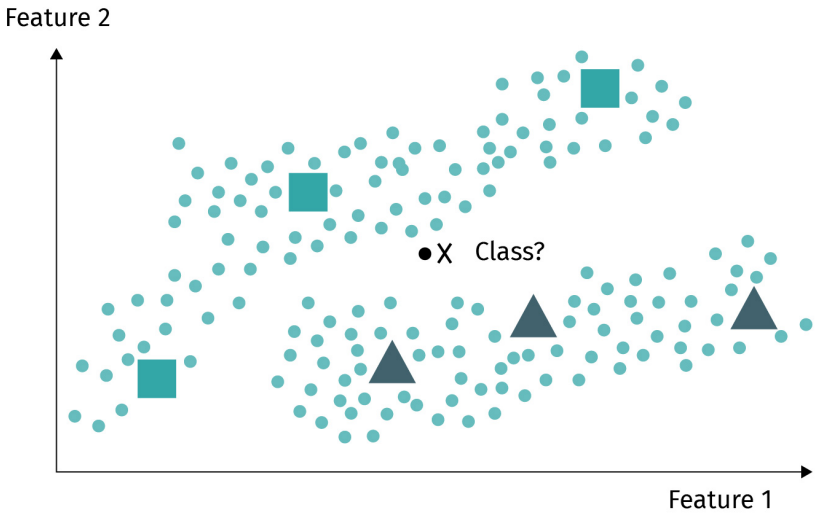
- Clustering is a technique that aims to divide input data into groups consisting of similar samples. Popular approaches include
  - K-means
  - hierarchical cluster analysis
- Visualization and dimensionality reduction commonly refers to the process of reducing the number of input variables in a set of input data without significantly affecting the quality of data and for facilitation visualization. Most used algorithms include
  - principal component analysis (PCA)
  - kernel PCA
  - locally linear embedding
- Anomaly and novelty detection are techniques used to identify if new data belong to existing observations, groups of data, or if they are significantly different from current data and need to be cast out as outliers. Popular algorithms include
  - one class support vector machines
  - isolation forest

Semi-Supervised Learning

Semi-supervised learning
a combination of supervised and unsupervised learning

Semi-supervised learning offers a happy trade-off between supervised and unsupervised learning. The training procedure uses a small set of labeled data for the classification process and a larger, unlabeled set for grouping features. A good example of this kind of algorithm is the photo recognition software on your phone. The unsupervised part of learning clusters all photos into groups denoting that a specific person appears in several photos. To correctly recognize the identity of that person, all it requries is a small set of labeled data (classification). The system is then able to pinpoint that person in all the rest of the photos. The figure below provides an illustration. The small round circles represent data points, while triangles and squares denote two separate classes. When a new instance is encountered, the learning model will first group it into one of the possible clusters and then decide on the class category it belongs to.

Figure 11: Semi-Supervised Learning



Source: Evis Plaku (2023), based on Géron, (2019).

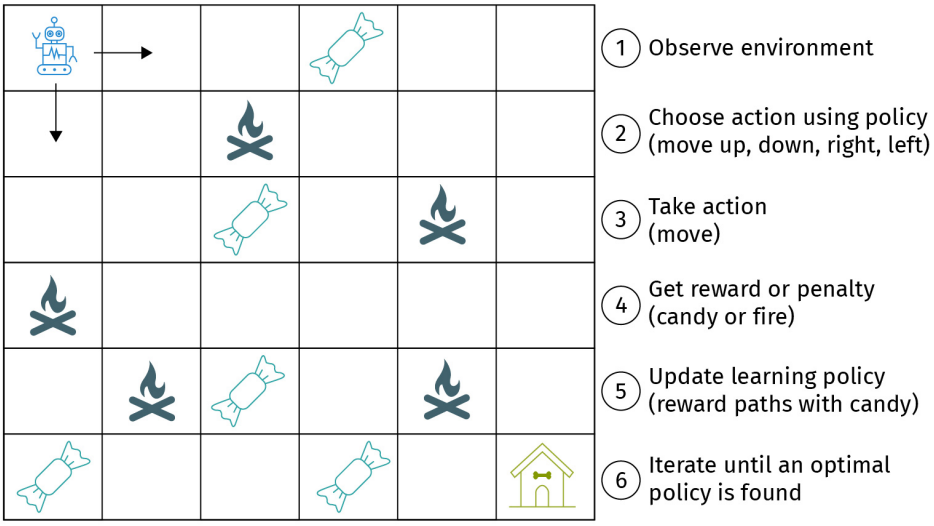
1.5 Reinforcement Learning

Reinforcement learning
trains a model through trial and error, rewarding desired behavior and punishing undesired behaviors

Reinforcement learning is a machine learning model that trains a computer agent by rewarding desired behaviors and punishing undesired ones. Given an environment, the task of the computer agent is to perceive and interpret the environment and then take action. Considering the desired outcome, these actions will either be rewarded or punished. The agent will learn, by trial and error, an optimal strategy, known as a policy. The policy defines what action the agent should choose in each situation.

For example, consider the figure below. A robot should learn how to move from a source destination to a goal destination while situated in a given environment. It uses its sensors to observe the environment and selects an action using its policy. Once the action is taken, the agent will either be rewarded or punished depending on the outcome of the action. This will lead to an update of the learning policy to accommodate the experience of the agent. These steps will be repeated until an optimal course of action is found.

Figure 12: Reinforcement Learning



Source: Evis Plaku (2023).

Positive values (e.g., rewards) will be assigned to desired behaviors to encourage the agent. Negative values will be associated with undesired actions. The goal of the agent is to maximize the long-term reward to achieve an optimal solution.

Reinforcement learning differs from supervised learning. In supervised learning, the training data are annotated with the true labels so the model is trained using the expected outputs. In reinforcement learning, however, there is no training set, and the agent is expected to learn solely from experience. Through trial and error. Reinforcement learning finds wide applications in fields, such as gaming, personalized recommendations, and robotics.

SUMMARY

Machine learning is focused on using data and algorithms to solve complex tasks by relying on learning from experience. The notion of artificial neural networks, inspired by biological neural networks, can perform complex computations and process information through a vast number of interconnected nodes, forming various layers that transmit informa-

tion between them until it matches the desired output. Deep learning is a machine learning technique based on artificial neural networks that aims to identify underlying relationships and patterns in the input data.

Shallow and deep networks were discussed by highlighting their differences in terms of architecture and learning paradigms. We explored how shallow networks contain only one hidden layer, while deep networks are composed of several hidden layers that allow them to generally achieve better performance on many complex tasks. We also demonstrated how to build a simple neural network used to classify handwritten digits.

Finally, various types of learning methodologies were introduced. Supervised learning is a technique where learning takes place through a training procedure that uses labeled input data. In contrast, unsupervised learning does not require annotated data, but aims to cluster and group input data according to their intrinsic features. Another approach, known as semi-supervised learning, is considered an intermediate ground between the two because it uses a small portion of labeled data to guide classification and a larger portion of unlabeled data for clustering. Reinforcement learning is a more novel learning paradigm where the computer agent learns solely by experience. Through trial and error, the agent takes action and is either rewarded or punished depending on the outcome, with the objective of exploring an optimal policy by maximizing a long-term reward.

UNIT 2

FEED-FORWARD NETWORKS

STUDY GOALS

On completion of this unit, you will be able to ...

- understand what a feed-forward neural network is.
- distinguish between different network topologies and cost functions.
- explain how the backpropagation and gradient descent algorithm help a neural network to train a model.
- build a simple feed-forward neural network used for classifying image data.
- describe how batch normalization can improve the performance of a network.

2. FEED-FORWARD NETWORKS

Case Study

Anne is a neuroscientist and Lukas is a software engineer. Anne and Lukus are also interested in fashion and they have agreed to maintain the website of a local store that displays clothing items grouped into several categories. They want to find a way that will allow them to organize thousands of images of clothing items quickly and effectively into their respective categories. Doing that manually requires tremendous time and effort.

Anne has a basic understanding of neural networks and she suggests that a machine learning algorithm can be used to automate this process. Lukas agrees that they can train a computer system to learn how to classify these items automatically. First, the existing set of clothing items that are already organized into categories should be used to train a **feed-forward neural network** so that it learns a model that maps images to their respective categories. Then, this model can be used to classify previously unseen items, thus saving Anne and Lukas a lot of time and effort.

Feed-forward neural network
A feed-forward neural network is an artificial network in which connections do not form a cycle.

2.1 Architecture and Weight Initialization

Feed-forward neural networks are the backbone of deep learning, widely used to address numerous challenging problems such as pattern recognition, computer vision, stock market prediction, and many others. Commonly known as a multi-layered network of neurons, feed-forward neural networks are thus named because information is processed only in the forward direction.

A feed-forward artificial neural network is composed from a stack of layers, consisting of an input layer, an output layer, and at least one hidden layer. The input layer is responsible for accepting and processing input data before passing them through to the next layer for further processing. Typically, the number of neurons in this layer is equal to the number of attributes of the input data. The hidden layer(s) transform the received input and pass it to subsequent layers. The activation function takes into consideration the strength of the connections between neurons and the **bias** measure and decides the traversal of neurons to subsequent layers. This process is repeated until an output is generated in the final output layer. The most fundamental architectures of feed-forward neural networks are presented below.

Bias
a constant that is added to the product of inputs and weights

The Perceptron

One of the most basic artificial neural network (ANN) architectures is known as **the perceptron**. Based on a model known as threshold logic unit (TLU), the perceptron consists of binary input values, connection weights, a bias, and an activation function. Weights are estimations that determine the strength of the connection between neurons. They play an important role as they determine the influence that input neurons will have on the target

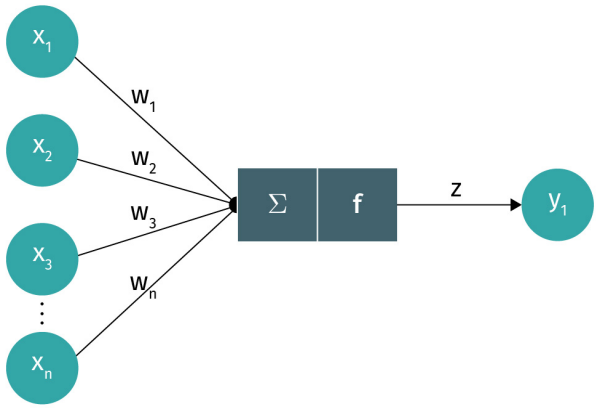
output. The bias is an additional term added to the neurons, typically to help the model fit the given data better by allowing it to shift the activation function. The TLU computes a weighted sum of the inputs, adds the bias, and applies an activation function to that sum to generate the result. The mathematical equation representing it and a visual illustration are provided below.

The perceptron
This is the most basic architecture of a neural network. It is designated to take a number of binary inputs and produce a binary output.

$$z = f(b + x \cdot w) = f(b + \sum_{i=1}^{n} x_i \cdot w_i)$$

where x_i denotes an instance of the input data, w_i the respective weights for $i \in \{1,\ n\}$, b the bias and f the activation function.

Figure 13: Perceptron Model



Source: Evis Plaku (2023), based on Géron (2019).

A single TLU is commonly used to perform a simple linear binary classification outputting a result belonging to one of two possible categories. As one can observe, input data are received by the input neurons. The network then computes the weighted sum of input neurons. Weights measure the strength of the connection between neurons and define how much influence the input will have on the desired output. The activation function defines how these weighted sums will be transformed from one layer to another, thus allowing the network to model relations between input and output data.

Training a perceptron

In a simple perceptron model learning takes place following an algorithm proposed by Rosenblatt (1958) thatrelies on the assumption that the connection between two neurons will grow stronger if one neuron triggers the other. This rule is the backbone of learning in the perceptron model. When the perceptron receives a training instance, it makes a prediction, that is then compared against the true target value. An error term measures the difference between the prediction of the network and the true target. This helps adjust the connection weights accordingly so that the neural network can aim to minimize the overall error and strengthen only those connections which achieve this objective. The mathematical equation enforcing this rule is presented as

$$w_{i,\ j}^{\text{next step}} = w_{i,\ j} + \eta \left(y_j - \widehat{y_j}\right) \cdot x_i$$

where

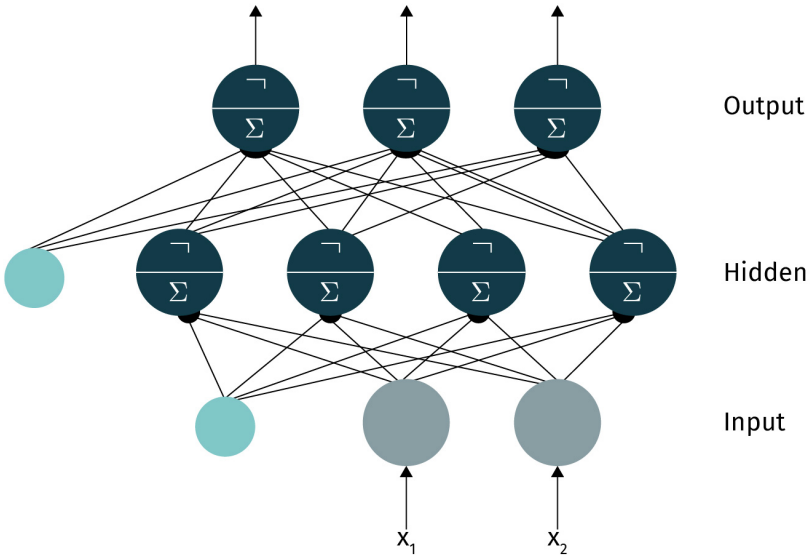
- $w_{i,\ j}$ represents the weight of the connection between neuron i and neuron j.
- x_i is the i[th] input value of the training data.
- $\widehat{y_j}$ is the output of the j[th] output neuron of the training data.
- y_j is the target output of the j[th] output neuron of the training data.
- η is the learning rate.

Multi-Layered Perceptron

Multi-layered perceptron
A multi-layered perceptron is a fully connected multi-layer neural network.

Note that this basic perceptron model is typically bounded to output linear models only and, therefore, can encounter serious difficulties in many practical situations involving non-linearity and complex patterns. To overcome the challenges of a simple perceptron model to address complex problems, **multi-layered perceptron** models emerged as an alternative. A multi-layered perceptron consists of an input layer, one or more layers of TLUs known as hidden layers and a final TLU layer known as output layer. All the layers except the output layer contain a bias neuron. Each layer is fully connected to the subsequent one. A multi-layer perceptron can be effectively trained to learn how to model complex behaviors. The figure below provides an illustration of such architecture.

Figure 14: Multi-Layer Perceptron Model



Source: Evis Plaku (2023), based on Géron (2019).

2.2 Cost Functions

Once a feed-forward neural network is trained, we are interested in investigating how well our model behaves when encountering input data that it have not been seen before. A cost function is a measure of "how well" the neural network does during the training phase. Choosing a well-defined and appropriate cost function is an important aspect when designing a neural network. The cost function is denoted as a single value that rates the overall performance of the neural network. It is typically in the form

$$C\big(W,\ B,\ S^i,\ E^i\big)$$

where

- W denotes the set of connection weights of the network.
- B denotes the set of biases of the network.
- S^i denotes the input of a single training sample.
- E^i denotes the expected output of a sample i.

According to Nielsen (2018), the cost function is required to satisfy the following two properties:

1. The cost function C is expected to be the average of cost functions for individual training examples, as in $C = \frac{1}{n} \cdot \Sigma_x C_x$ for training examples x. By enforcing such requirements, it is possible to run algorithms that facilitate the learning phase by continuously updating weights and tweaking parameters until the optimal solution is found.
2. The cost function must not be dependent on any activation values of the network besides the output values. This restriction is imposed to allow the network to backpropagate information from the output layer to previous layers, so that weights and other parameters are updated accordingly.

Overall, the key objective of the cost function is to quantify the error measured as the difference between the predicted values and the true expected values. The choice of the cost functions depends on the type of machine learning problem (e.g., regression or classification tasks) and specific features of the problem at hand. The most common cost functions are presented below.

Cost Functions for Regression Problems

Mean absolute error (MAE) measures the average error observed between predictions of the model and the expected results, mathematically expressed as

$$MAE = \frac{1}{n} \cdot \sum_{i=1}^{n} |y_i - x_i|$$

where for each training instance i, the predicted value and the expected value are denoted, respectively, by y_i and x_i for all n samples in the dataset.

Mean squared error (MSE)

The MSE is a widely used cost function that measures the error as the squared difference between the predicted and expected values. Instead of calculating the absolute values of the differences as in MAE, the MSE cost function calculates the square of individual error measures. It is mathematically expressed as

$$MSE = \frac{1}{n} \cdot \sum_{i=1}^{n} (y_i - \overline{y}_i)^2$$

Root mean squared error (RMSE)

An extension of the MSE error, the RMSE, estimates the error of the cost function as the square root of the sum of squared differences between the predictions produced by the model and the actual expected values. RMSE is mathematically expressed as

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N} (Predicted_i - Expected_i)^2}{N}}$$

Cost Functions for Classification Problems

Entropy
a measure of the average amount of information required to represent an event when considering all possible outcomes

The cross-entropy cost function is commonly used in classification problems to measure the difference between two probability distributions. It is built from the idea of **entropy** in information theory. In our context, it is used to measure the performance of the classification model when the output is a probability value between zero and 1, i.e., it measures the probability of a testing instance belonging to one of the target categories. The cross-entropy cost function, commonly denoted as H is mathematically expressed as

$$H(x) = \sum_{i=1}^{n} p(x) \cdot \log q(x)$$

where $p(x)$ denotes the probability associated with the true labels, and while $q(x)$ denotes the probability associated with the estimate of the model for all n testing samples. In practice, the cross-entropy function works best when the data is normalized.

Categorical cross-entropy

The categorical cross-entropy function is commonly used when an input data can be classified to only one category. An example of that is classifying handwritten digits and matching them to only one of the ten categories representing digits (numbers zero to 9). Another special version of categorical cross-entropy, known as binary cross-entropy, is used in cases of a binary classification, that is, the output belongs to one of two possible classes, for example, a classification between cats and dogs.

2.3 Backpropagation and Gradient Descent

Gradient descent is a popular and widely used algorithm that is able to find optimal solutions to a wide range of problems. The objective is to continuously tweak learning parameters until the associated cost function is minimized. To make learning possible, we need to find a set of weights such that the cost function of our model is minimized. The gradient descent algorithm is initialized with a set of parameters (weights and biases), and it attempts to fine tune these parameters. The learning algorithm moves in the desired direction by continually computing the gradient (i.e., derivative) which marks the slope of the cost function. To minimize the cost, the learning algorithm moves in the opposite direction of the gradient.

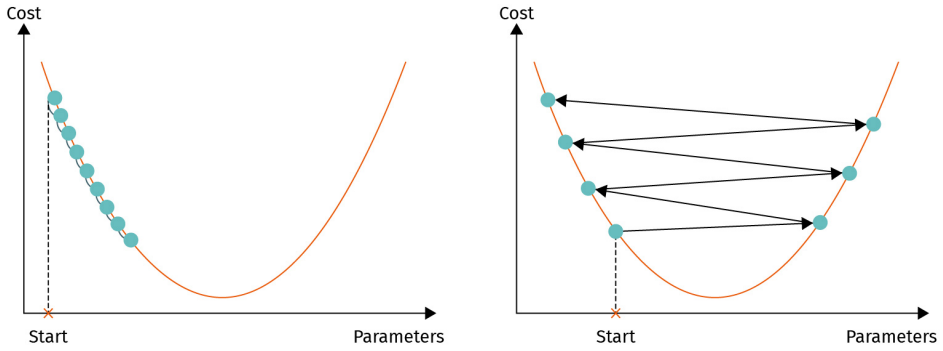
Gradient descent is an optimization technique used to train a machine learning model that minimizes a cost function.

To get a better sense of the gradient descent algorithm, imagine that you are lost on a mountain hike and would like to reach the lowest point as fast as possible. Without a map, a good strategy you can follow is to quickly check the terrain and then take a downward step in the direction where the land's slope is the steepest. Repeating this process often enough will allow you to descend fast to the lowest point, thus reaching a safe valley. The gradient descent algorithm operates in a similar way. Without a map, that is, without knowing the value of the function for every possible set of parameters, the best strategy is to move in the direction in which the cost function reduces. To do that, the following four steps are iteratively repeated:

1. Weights are initialized randomly.
2. The gradients of the cost function are calculated while considering the parameters of the model.
3. Weights are then updated by an amount that is proportional to the calculated gradients.
4. This process is repeated until the cost function is minimized or a termination criterion is met.

Denoting the set of parameters as the algorithm starts by initializing with random values. Then, a step at a time is taken while aiming to reduce the cost function to a minimum. The size of the steps play an important role in the efficiency and effectiveness of the algorithm. Using small steps would require many iterations and a longer time until convergence to the minimum. Conversely, using larger steps might cause the gradient to oscillate from one direction to another direction, ending up in a place that is possibly worse, and causing the algorithm to diverge. The figure below provides an illustration of both cases.

Figure 15: Gradient Descent Learning Rates

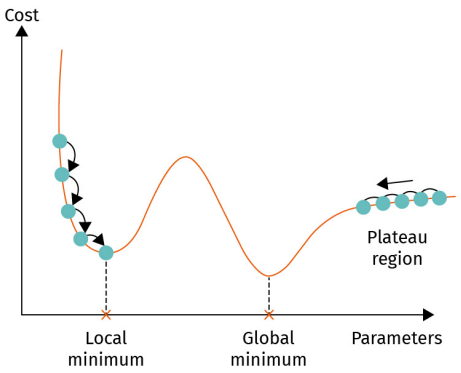


Source: Evis Plaku (2023), based on Géron (2019).

The gradient descent algorithm faces important challenges, especially in cases where the cost function is not regular. Note that a function is regular if it is defined everywhere and has a finite derivative at any point. That makes convergence to a global minimum difficult. Note that the updates of the weights are only affected by the learning rate and the gradient at that moment. Previous steps are not considered. That causes problems when the gradient is computed at high-level areas with little change in steepness, commonly known as plateau areas.

In such scenarios, the update of weights is minimal and might even be potentially zero. That causes a disruption of the learning process and makes the network stagnant. Consider the illustration provided below. In the plateau region, the update of weights will lead to minuscule changes and the gradient will move very slowly, or even get stuck. Another issue is illustrated on the left part of the figure, where the gradient can enter a region where it finds only a local minimum without having the possibility of converging to the global minimum.

Figure 16: Gradient Descent Pitfalls



Source: Evis Plaku (2023), based on Géron (2019).

For the learning process to take place effectively, it is obligatory to search a vast space of possibilities for a model that identifies the correct set of parameters that minimize the cost function. Tweaking the parameters until you find the correct ones is a difficult task that becomes even more complex when the input data contain many features. The search resembles trying to find a needle in a hundreds-dimensional haystack. The job of the gradient descent algorithm is to efficiently move toward the bottom of the haystack because that is where the needle is located.

Variants of Gradient Descent Algorithm

There are multiple variants of the gradient descent algorithm. They usually differ in the amount of data that is used to calculate the gradient. Considering that the computational cost grows exponentially fast with large inputs of data, it is important to choose an algorithm that can perform efficiently. Below are summarized the most-used variations of the gradient descent algorithm.

Batch gradient descent

The batch gradient descent algorithm computes the gradient of the cost function over the complete training set of input data, at every single step. As a result, this algorithm is very expensive computationally, which makes the learning process slow. The gradient of the cost function is computed with regard to each parameter of the model, say θ_j. It measures by how much the cost function be affected when parameter θ_j changes. This is known as a partial derivate.

Stochastic gradient descent

Batch gradient descent is very slow because it needs to use the entire training set to compute the gradient for every step. Stochastic gradient descent follows an opposite strategy. Instead of choosing the entire input dataset, only one instance is selected at random. This drastically improves computational performance but comes with the additional cost of introducing irregularity to the algorithms, which will be able to jump from one direction to another quite often.

Repeating this process long enough will allow the gradient to eventually get close to the minimum, however, it will not stop there. Forcing the algorithm to stop means that we will find a good-enough solution, but probably not the optimal one.

The stochastic gradient descent algorithm has been proven to work well in practice. Due to its random and irregular nature, the algorithm behaves well in situations where a local minimum is encountered, because it jumps toward another direction, thus escaping the local minimum.

A common improvement of pure gradient descent algorithm is to gradually reduce the learning rate. It is typical to start with large steps that allow the algorithm to make notable progress in reasonable time and avoid local minimums, and then to gradually reduce the step size to help the algorithm converge on a global minimum.

Mini-batch gradient descent

The mini-batch gradient descent algorithm emerged as a happy middle ground between batch and stochastic gradient descent. A set of random instances known as mini-batch are used to compute the gradient. The algorithm generally behaves better than stochastic gradient descent when it is around a minimum, as it makes more steps close to and around it. However, it suffers more than stochastic gradient descent when encountering a local minimum.

The Backpropagation Algorithm

Backpropagation
an algorithm used to fine-tune the weights of the neural network in the learning process

Backpropagation is a central mechanism that equips multi-layered neural networks with the ability to learn. When a neural network is built, weights determine the strength of connections between neurons of the network. As information is transmitted through the network, such weights are updated and when the final layer is reached, the network makes a prediction of the relation between the input features and the desired output. The difference between the predicted value and the actual one is considered a loss/error. The key objective of the backpropagation algorithm is to transmit information in the opposite, i.e., backward direction so that it allows the weights to correct, gradually minimizing the error term and improving the efficacy of predictions.

Ever since Rumelhart et al. (1986) introduced the backpropagation learning algorithm, it has become a standard and efficient way of training a neural network. The backpropagation algorithm is an effective technique that continuously tweaks the connection weights and biases of the network with the objective of gradually reducing the overall error. It computes the gradient of the error of the network for every parameter of the model. That provides information on how to fine-tune weights of connected neurons and the associated biases so that the overall error is minimized, and the network converges to an optimal solution. The algorithm contains two main phases: a forward pass and a backward pass. The main steps are as follows:

1. The algorithm starts by using one mini-batch at a time from the full set of training data. The process is repeated multiple times. Each pass is commonly referred to as an epoch.
2. The input layer receives the mini-batch and passes it through to the first hidden layer. For each instance of the mini batch, the backpropagation algorithm computes the outputs of all the neurons in this hidden layer.
3. The intermediate output results are transformed through the whole set of hidden layers, each receiving information from the previous layer, processing it, and outputting the results to the subsequent layer until the last layer, namely the output layer performs the same process. This is the forward pass. Note that intermediate results are also stored, as they are needed for the backward pass of the algorithm.
4. Once the forward pass is complete, the output error of the network is calculated using a loss function that measures the difference between the desired and the actual output.

5. This measure of the error is then distributed to each connection of the output to determine how much they influenced the overall error. This process is repeated backwards from the output layer up until the input layer. This reverse step helps measure the influence that network weights have on the current result.
6. When the backward pass is completed, the algorithm performs a gradient descent step to fine-tune all the connection weights in the network using the error gradients computed from the previous steps.

In summary, the algorithm works through a forward pass in which a prediction is made given the training data, and then an equally important backward pass that measures the role played by each connection weight in the overall error. The gradient descent step is used to tweak connection weights to reduce the error.

It is important to note that, for the backpropagation algorithm to work effectively in practice, all the connection weights of the hidden layers must be initialized at random. Doing otherwise will severely impact the training process, causing it to possibly fail. Imagine the opposite, that is, initializing all weights and biases to the same value, say zero. Then, all the neurons of a layer will be identical to one another. Since the backpropagation algorithm will affect them in the same way, they will continue to remain identical. That means that the network will behave as if it has only one neuron per layer, despite having hundreds of them. Initializing connection weights at random provides an opportunity to train a network that is not symmetrical, thus empowering the backpropagation algorithm to train a diverse team of neurons into efficiently learning a generalized model from the input data.

Activation functions used in the backpropagation algorithm

An activation function defines how the weighted sum of inputs will be transformed to an output in any given layer of the neural network. The choice of an activation function is an important step of building a network. When implementing the gradient descend algorithm, the activation function should have a derivative that never becomes zero, thus allowing the gradient to make progress at every single step. Several activation functions have been proven to work well in many practical applications:

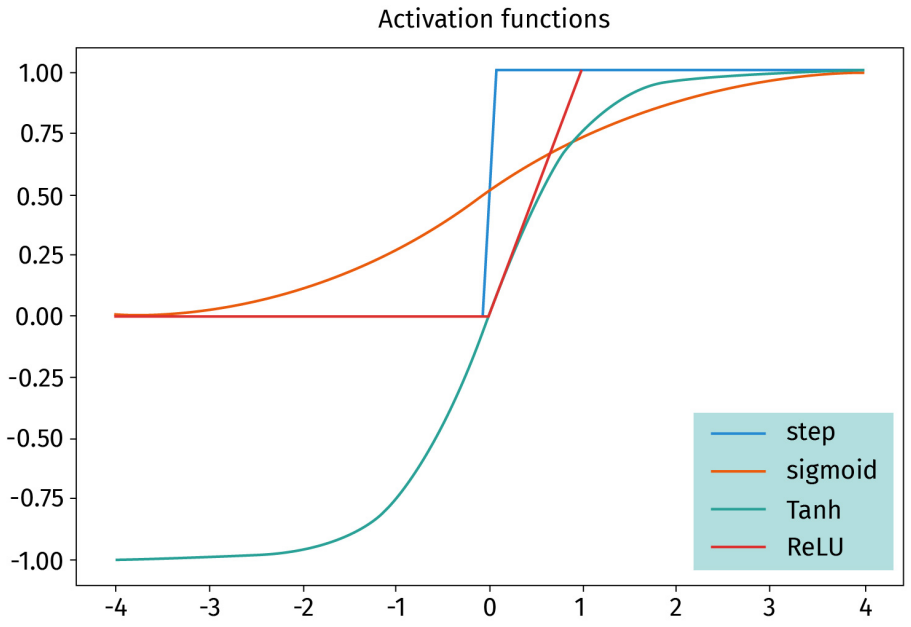
- The step function is commonly used in the perceptron model. It uses a threshold value and, if the input sum is above that threshold, then the output will be a certain value, say 1, and another value, say 0, otherwise.
- The logistic function, otherwise known as the sigmoid function, is commonly used to add non-linearity to the learning model. Its key characteristics is that it can map any real-value input to an output between zero and 1. It is mathematically defined as $\sigma(z) = \frac{1}{1+e^{-z}}$. When plotted, the function forms an "S" shape, as denoted in the figure below.
- A disadvantage of the sigmoid function is that the function gets flat if z is a large positive or negative value. The hyperbolic tangent function is an alternative example of a function that is continuous and differentiable. The output values however range between -1 and 1. The advantage is that negative inputs will be mapped to strong nega-

tive values and zero inputs will be mapped to values near zero in the tangent graph. The hyperbolic tangent function is mathematically defined as: $\tanh(z) = 2\sigma(2z) - 1$ for $\sigma(z) = \frac{1}{1+e^{-z}}$.

- The rectified linear unit function, mathematically defined as $ReLU(z) = \max(0,\ z)$ is a function that tends to work quite well in practice and is very fast to compute. The function is continuous, but, unlike the sigmoid or the hyperbolic tangent function, is not differentiable when $z = 0$ and the derivative of the function is zero when z is negative.

The figure below provides a visualization of these commonly used family of activation functions and their derivatives.

Figure 17: Activation Functions and Their Derivatives



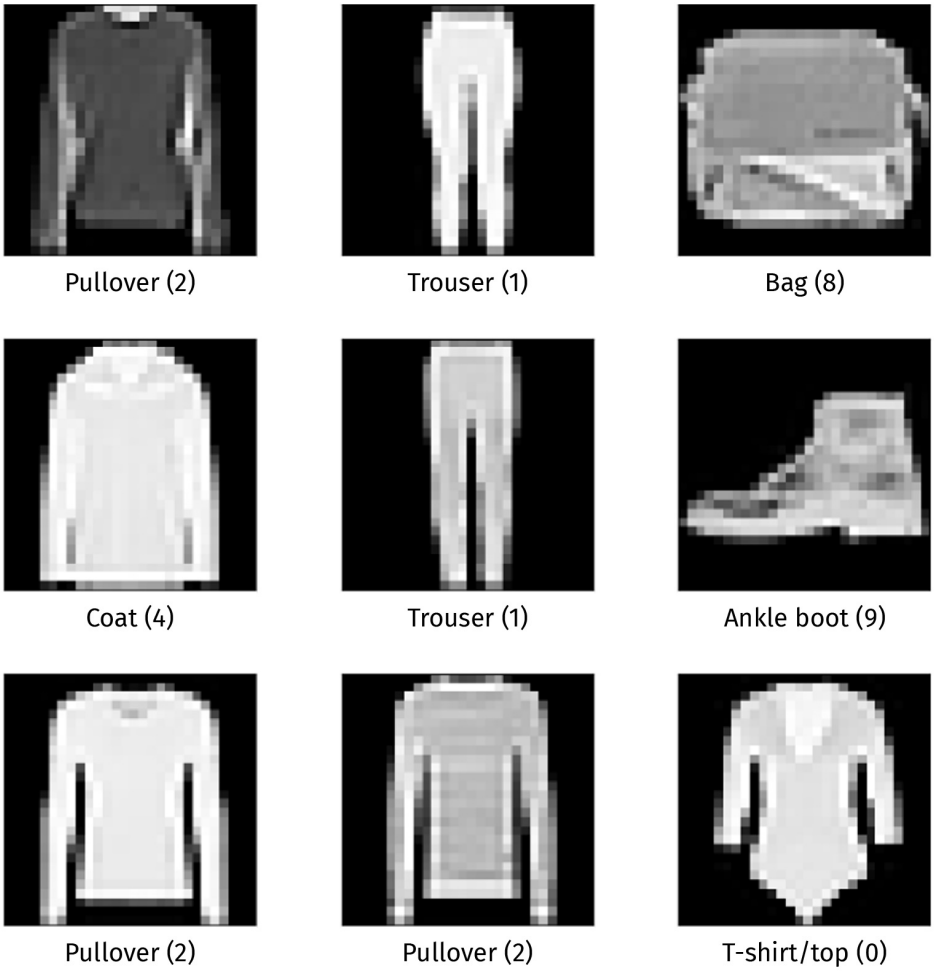
Source: Evis Plaku (2023).

Note that our networks are a chain of layers that transform input values to desired target values. The role of activation functions is crucial because it allows us to introduce non-linearity in our inputs. Alternatively, without using such functions, all the transformations would be linear (chaining several linear functions still yield a linear function) and that would not be effective in learning complex models that are non-linear. That is often the case of the models that are learned by neural networks.

Implementation: Building an Image Classifier Neural Network

We will now demonstrate how to build a neural network that can be used to classify images of items of clothing. We will use a popular dataset known as Fashion MNIST which consists of 70,000 grayscale images of 28 by 28 pixels each, categorized in ten separate classes, each representing a fashion item (Xiao et al., 2017). The figure below provides an illustration.

Figure 18: Samples from Fashion MNIST Dataset



Source: Evis Plaku (2023), based on Zalando (2017).

Note that the classes are diverse and the set of clothes belonging to a particular category contain a large variety of features. We will take advantage of Keras library and preload the dataset directly from it. Loading the dataset as

```
from keras.datasets import fashion_mnist
fashion_mnist = keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) =
                              fashion_mnist.load_data()
```

We split the dataset into a training set and a testing set, and we also create a validation set. Our neural network will rely on the gradient descent algorithm to learn a model out of the input data. We will also scale the features to a zero to one range by dividing each pixel by 255.0. This allows to represent the color as floats (0.0 to 1.0) rather than integers (0 to 255):

```
X_valid, X_train = X_train_full[:5000] / 255.0,
                   X_train_full[5000:] / 255.0
y_valid, y_train = y_train_full[:5000],
                   y_train_full[5000:]
```

To better understand the classification categories, we can encode them with meaningful class names:

```
class_names = ["T-shirt/top", "Trouser", "Pullover",
               "Dress", "Coat", "Sandal", "Shirt",
               "Sneaker", "Bag", "Ankle boot"]
```

Building the neural network

We will build our neural network using a sequential model that is a stack of layers connected sequentially to one another. The first layer will be a Flatten layer which has the objective of preprocessing the input data by converting the 28 by 28 pixels of each image into a one-dimensional array. Next, we will use two hidden layers. The first is a dense hidden layer composed of 300 neurons. The activation function of choice is the rectified linear activation function (ReLU) function. This layer also contains the weights matrix denoting the connection weights between the neurons of the layer and their input. Another dense hidden layer will be added that contains 100 neurons and also uses the ReLU activation function. Finally, since our classification problem aims to categorize clothing items in ten disparate class, we will add an output layer with ten neurons using the softmax activation function to better capture the fact that the categories are exclusive:

```
model = keras.models.Sequential([
             keras.layers.Flatten(input_shape=[28, 28]),
             keras.layers.Dense(300, activation="relu"),
             keras.layers.Dense(100, activation="relu"),
             keras.layers.Dense(10, activation="softmax")
```

Building the model

Now that the model is created, we need to compile and run it. To do so, we need to determine the loss function and the optimizer. In addition, we will also define the metrics that will be used during training and evaluation. Note that each instance of our input data belongs to an exclusive label category (10 not-overlapping classes). We will use the "sparse_categorical_crossentropy" cost function and we will train the model using the stochastic gradient descent algorithm, i.e., we will implement the backpropagation algorithm described earlier. In the following code, the Stochastic Gradient Descent algorithm is denoted by "sgd."

```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

Training and evaluating the model

The model is now ready to be trained. The training algorithm will be fed the input features along with the target categories, in addition to the number of epochs we will use for the training procedure. To carefully investigate how our model will behave on data it has not encountered before, we will provide an additional set of validation data:

```
model_history = model.fit(X_train, y_train, epochs=40,
                  validation_data=(X_valid, y_valid))
```

Output

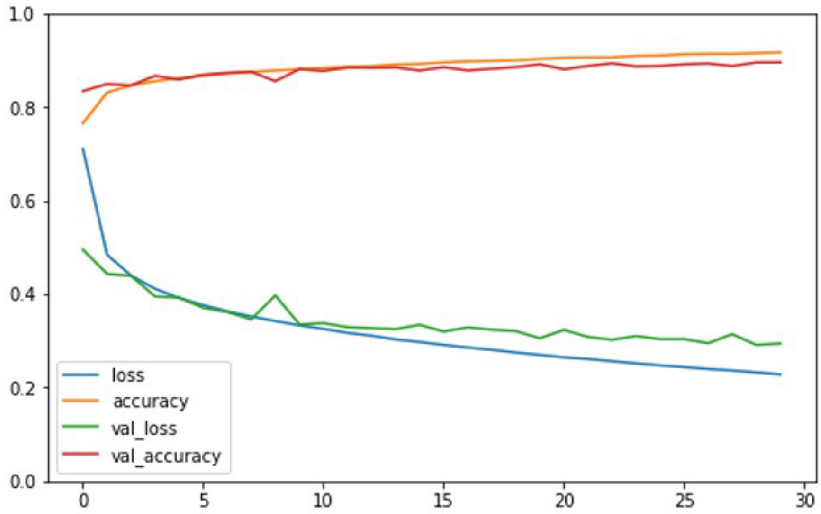
```
Epoch 30/30
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2265
- accuracy: 0.9180 - val_loss: 0.3276 - val_accuracy: 0.8828
```

The model is now trained. The accuracy of the model is approximately 92 percent on the training data and 88 percent on the validation data. To get a better understanding of how the model learned during each epoch, we can plot the history of training and validation accuracy and error. We rely on pandas and matplotlib libraries to achieve that:

```
import pandas as pd
import matplotlib.pyplot as plt
pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.gca().set_ylim(0, 1)
plt.show()
```

As the figure below denotes, both the training and the validation accuracy grow significantly during the training procedure while the training and the validation loss decrease steadily. The training and validation lines walk closely to one another, which means that the model behaves well enough even with data that it has not encountered before.

Figure 19: Training and Validation Accuracy and Loss



Source: Evis Plaku (2023).

This example demonstrates that, by taking advantage of Keras library and its built-in models and functions, we can build, train, evaluate, and test a feed-forward neural network that implements the stochastic gradient descent algorithm and backpropagation while using typical cost and activation functions in an easy-to-understand manner.

2.4 Batch Normalization

Training deep forward neural networks that are composed of many layers is challenging. The key objective of the training phase is to produce a model that generalizes well to new data. Training, however, can become highly sensitive to the initial distribution of connection weights. Also, the weight update can cause the distribution of inputs to layers to change. This can make the network chase a target that is continuously moving.

To address such issues, normalization is used as a vast category of techniques that aims to make samples that are fed to a machine learning model more similar to one another. Standardizing the input helps the model to generalize with previously unseen data, but it also has the effect of significantly reducing the number of iterations required to train the model. A common example of normalization is the assumption that the training data are distributed according to a normal (Gaussian) distribution which is centered and scaled accordingly. For example, by subtracting the mean of the data from each data point and dividing them by their standard deviation, it causes the data to follow a normal distribution centered on zero with a unit standard deviation.

Batch normalization is a normalization technique that continuously normalizes the input data during the training phase, even as the mean and variance of data changes during the training phase. An extra layer is added in the feed-forward network with the purpose of standardizing the inputs received from previous layers before transmitting them to the subsequent layers. Since the input is typically trained in batches, the normalization process is also performed in batches, and not in single inputs.

Batch Normalization
a technique used to standardize the inputs received by a neural network's layers

Batch normalization helps with gradient propagation, but it also helps to speed up the training process and smoothens the loss function as demonstrated by Ioffe and Szegedy (2015). The process of batch normalization adds a layer of complexity to the model because of the extra operations that need to be performed. This comes with an additional computation cost; however, since convergence is typically faster with batch normalization, the technique provides overall benefits during the training phase of the network.

Implementing Batch Normalization

The Keras library provides built-in, easy-to-use functionalities for implementing batch normalization. Typically, a batch normalization layer is added before or after calling the activation function of a hidden layer. To start the training phase with normalized input, a batch normalization layer can also be added as the first layer of the model, as shown in the following example.

```
model = keras.models.Sequential([
keras.layers.Flatten(input_shape=[28, 28]),
keras.layers.BatchNormalization(),
keras.layers.Dense(300, activation="relu",
                        kernel_initializer="he_normal"),
keras.layers.BatchNormalization(),
keras.layers.Dense(100, activation="relu",
                   kernel_initializer="he_normal"),
keras.layers.BatchNormalization(),
keras.layers.Dense(10, activation="softmax")
])
```

The example is an illustration of a simple model containing only two hidden layers. The effects of batch normalization will significantly increase as the network becomes deeper containing a larger number of hidden layers.

SUMMARY

In this unit, you learned that feed-forward neural networks are widely used to address a variety of challenging problems. You were presented with fundamental neural network architectures while discussing the role of each component. A simple perceptron model and the multi-layered perceptron were presented as examples of such networks.

We investigated the importance of cost functions in measuring how well the neural network behaved during the training phase and presented key characteristics of good cost functions, in addition to discussing commonly used functions for regression and classification tasks. Cost functions such as the mean absolute error, mean squared error, root mean squared error, cross entropy and categorical cross-entropy were presented and defined.

You also learned that the backpropagation and the gradient descent algorithm can be used effectively to facilitate the learning process of our network by continuously tweaking parameters until a nearly optimal solution is found that minimizes the cost function. Variants of the gradient descent algorithm such as batch gradient descent, stochastic gradient descent and batch gradient descent were discussed, in addition to common activation functions. An implementation of a neural network used to classify images denoting clothing items in ten disparate categories was also provided.

Finally, you learned that batch normalization is a generalization technique that helps a neural network not only learn how to explain the training data, but also to be able to effectively make predictions on data it has not encountered before.

UNIT 3

OVERTRAINING AVOIDANCE

STUDY GOALS

On completion of this unit, you will be able to ...

- understand the concept of overtraining a machine learning model.
- distinguish between different regularization techniques.
- describe common regularization strategies, such as early stopping, l1 and l2 regularization, dropout, and weight pruning.
- analyze the effects of regularization techniques in improving the performance of a machine learning model.

3. OVERTRAINING AVOIDANCE

Case Study

Anne and Lukas are eager to expand their practical knowledge on neural networks. Anne is a neuroscientist and Lukas is a software engineer. They have carefully gathered a large dataset containing images of clothing items separated into various categories. They want to train a neural network to discover underlying patterns and relations in the input data and then use this model to classify a large number of previously unseen images of clothing items.

To achieve this objective, Anne and Lukas extensively increase the number of training iterations and use all the image data they have in the training procedure. Once the training is over, something unusual happens. The model is able to predict the training data with a very high accuracy. Yet, when tested against image data of clothing items that it has not encountered before, the model behaves poorly. It is as if the model learned to memorize the entire training set, but it gets easily disoriented when observing new instances. In such cases, the classification error is high.

After carefully researching and investigating the issue, Anne and Lukas learned the hard way that what occurred is a common problem affecting many machine learning models, commonly known as overtraining. They are now very interested in learning how to deal with such a problem and be able to overcome it.

3.1 What is Overtraining?

A machine learning model leverages input data to train a model that is able to learn underlying patterns and relationships such that it can effectively map input features to target outputs. It happens that when the model is too complex or it trains for too long, it can start learning irrelevant information, or noise. The model then starts to “memorize” the training data, i.e., it fits the training data too closely. This can be a problematic issue because the model will suffer when it encounters new data.

Recall that the objective of the training phase is to adjust the model and its parameters to achieve the best performance possible for the given data. This process is commonly known as optimization. However, when a machine learning model is built, the ultimate purpose is to effectively use it on a set of data that it has not encountered before. This process is known as generalization. There are situations in which the model behaves well on the training data, but it does not generalize well to previously unseen data. This issue is known as **overfitting**.

Overfitting results when a machine learning model predicts training examples with

Optimization and generalization are often correlated. The objective is to gradually improve the accuracy of prediction both in the training and the testing data. Though this happens quite often in the earlier iteration steps of the learning phase as the neural net-

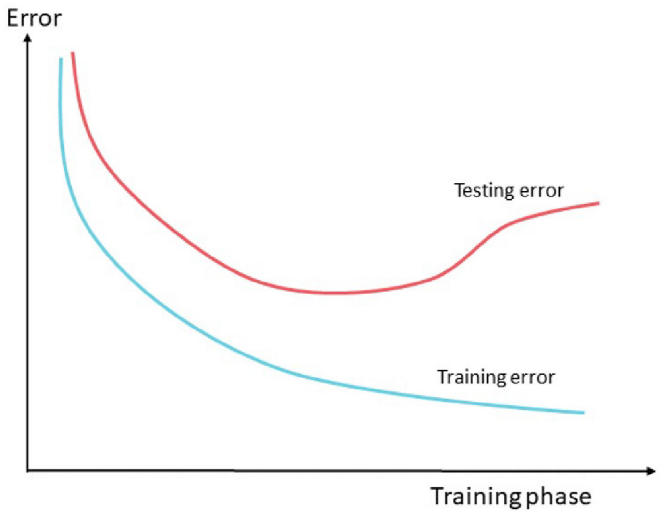
work attempts to model relevant patterns, it might happen that generalization stops improving, or worse, degrades, while the training accuracy continues to grow. When this happens, the model is said to be overtraining.

high accuracy but performs poorly with previously unseen data

In other words, the neural network and the machine learning model in general, is exploring the training data so deeply that it is starting to learn patterns that are too specific only for the training data but are either irrelevant or misleading for new data. Imagine if we attempt to memorize the solution to a difficult math problem, but do not understand the underlying mathematical rules that can help us to solve other similar problems.

Similarly, when overfitting, the neural network is memorizing only the training data; it does not behave well when it encounters input data that it has not seen before, often leading to inaccurate predictions. In such cases, the error measuring the accuracy of prediction steadily decreases for the training set, but when encountering new testing data the error typically begins to stall for a while before noticeably increasing as more data are encountered. The figure below provides an illustration of such a scenario.

Figure 20: Overfitting Illustration



Source: Evis Plaku (2023).

Deep neural networks, and complex machine learning models in general, are often prone to the risk of detecting very subtle patterns in the data, or the noise itself. When these events happen, generalizing to new instances will be problematic.

Underfitting the training data

Underfitting happens when the model cannot capture substantial relationships between input features and target output which results in high error rates

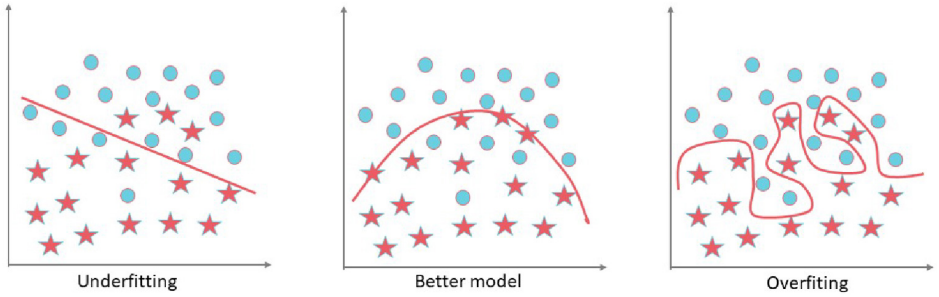
When training a machine learning model, one can encounter the opposite problem of overfitting, that is, **underfitting**. It occurs when the model is too simplistic to learn and discover meaningful underlying patterns in the data it observes. In such cases, the nature of the data is far more complex than the model; therefore, predictions will be inaccurate, even on the training data.

Underfitting usually occurs when the input features are not significant enough, thus the network is unable to determine a relevant relationship between input and output data. A model can be underfitted if its complexity is too low or the model is not trained for the necessary amount of time. Insufficient availability of training data is another reason for underfitting. Several strategies might be followed to deal with underfitting, including the following:

- identifying a more powerful model that usually contains a larger number of parameters
- exploring the features of the learning algorithm to identify features that can have a larger impact on the accuracy of the model
- removing noise from the training data
- increasing the number of training iterations

The graphs below provides an illustration of how different models can potentially fit against previously unobserved data. The left graph demonstrates a model that underfits the data. In such a case, both the training and testing error are high. The right graph shows a far too-complex model that overfits the training data. The middle graph presents the middle-ground; a more optimal model that has a low training error, but that also generalizes well to new data.

Figure 21: Underfitting, Overfitting, and a More Accurate Model



Source: Evis Plaku (2023).

The bias-variance tradeoff

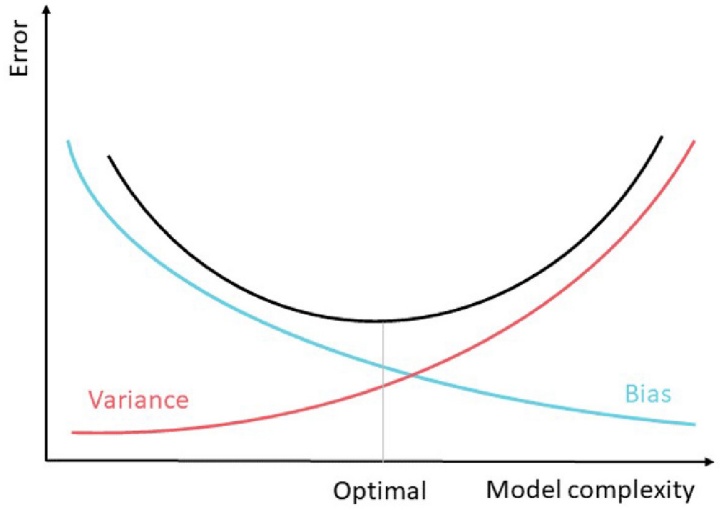
As shown in the work of Sammut (2010), the generalization error that often occurs in neural networks, and machine learning models in general, can be expressed in terms of three different types of errors:

1. Bias denotes an error that occurs when wrong assumptions are made for the input data. For example, assuming the input features have a linear relationship when, in fact, the relationship is more complex, leads to a bias error. A model that has a high bias tends to underfit the training data.
2. Variance expresses the error related to the excessive sensitivity of the model on the training data as a result of representing minor variations in input. Such an error occurs more often when the model is far too complex, thus leading to an overtrained model.
3. Irreducible error denotes the noise associated with the data itself. Detecting and/or removing outliers in the input data or improving the quality of the received data (for example, data received from sensors) might lead to reducing this type of error.

Training a neural network, or a machine learning model in general, is often a process of calibrating the bias variance trade-off. Typically, when the complexity of the model is increased, the variance is increased as well, while the bias error is reduced. Conversely, when the complexity of a model is reduced, the variance is also reduced, but the bias is increased. The figure below provides a typical illustration of the bias-variance tradeoff.

The blue line denotes the bias that decreases as the model becomes more complex, while the red line denotes the variance which grows with the growing complexity of the model.

Figure 22: Bias Variance Trade-Off



Source: Evis Plaku (2023).

Four distinct cases can be identified using possible combinations of variance and bias errors. A model with low variance and low bias is always desired. On the other hand, the opposite of that scenario, namely a model with high bias and high variance, is one that needs to be avoided at all cases. Models that exhibit low variance, but high bias are typi-

cally models that suffer from underfitting, while a model with high variance and low bias usually overfits. Building a neural network requires careful investigation in order to decide which tradeoff between bias and variance might be the more effective.

Fighting the issue of overtraining is a crucial phase in building effective neural networks. The next sections provide insight in some of the most common techniques that address overtraining.

3.2 Early Stopping

Early stopping
a regularization technique used to avoid overtraining by interrupting training if the validation error starts to increase

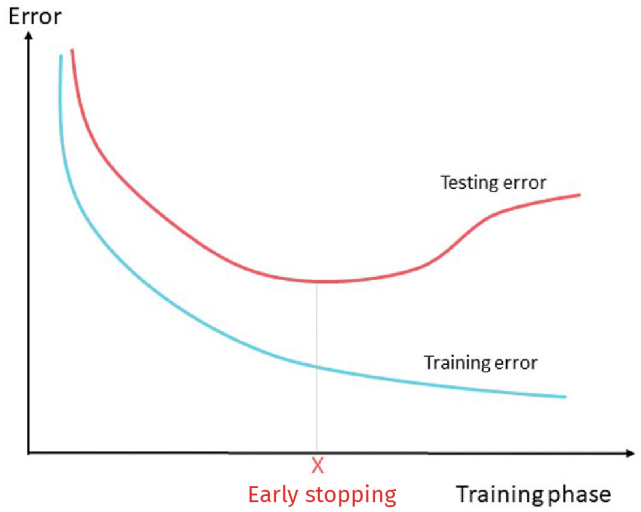
Overtraining is combatted using a general set of techniques referred to as regularization. **Early stopping** is one of these strategies. The idea of early stopping is simple: during the training phase, the objective of the machine learning model is to gradually improve the accuracy of prediction, and, therefore, reduce the overall error. To effectively estimate its performance, the learned model is tested against a validation set composed of previously unseen data.

When it is observed that the validation error reaches a stalling point and then steadily increases, the algorithm stops iterating at the critical point. This strategy is known as early stopping.

To determine the stopping point, the training error and the validation error are monitored closely. Though the training error typically decreases steadily, there comes a moment where the validation error stops decreasing and starts increasing. That is usually the point where the model starts to overfit the training data. Stopping the training phase will allow the model to have a lower variance and a better generalization.

The graph below provides an illustration. The blue line denotes the error measured on the training data. As one can observe, the training error is gradually reduced during the training phase. The red line illustrates how the validation error stops decreasing after a certain point and starts to increase. When implementing the early stopping regularization strategy, we can stop the learning process as soon as the validation error reaches the minimum in order to fight overtraining.

Figure 23: Early Stopping Regularization



Source: Evis Plaku (2023).

Early stopping implementation in Keras

In practice, early stopping is often implemented by allowing a neural network to train for an arbitrarily large number of epochs and then monitoring closely the performance of the model. The training procedure will stop as soon as there is no sign of improvement on the validation error.

In Keras, early stopping can be implemented by taking advantage of callbacks which provide a way to interact automatically with the training model. Callbacks allow to specify the performance measure that will be used to monitor the training process and interrupt it when the trigger signal occurs. An example is shown in the code below.

```
model.fit(train_X, train_y,
validation_split=0.3,
callbacks=EarlyStopping(monitor='val_loss')
)
```

Note that the "monitor" argument determines the performance measure that will be used as a criterion for terminating the training phase. In this particular case, the validation loss will be used as a performance measure metric. Another argument could be used to determine if the chosen metric is to increase (i.e., maximize) or decrease (i.e., minimize). For example, we would seek to minimize the validation loss, but to increase the validation accuracy.

Once the chosen metric stops improving, the training procedure will stop. To identify the stopping point, i.e., the epoch number in which the training procedure stopped, we can use an additional argument known as “verbose” similarly to the example code shown below.

```
EarlyStopping(monitor='val_loss', mode='min', verbose=1)
```

In practice, it is often the case that the very first sign of no improvement in the chosen metric is not the best point to stop the training phase. Consider the case where the model moves slowly in a plateau region (there is no sign of improvement for a while; in some cases, it can also get a little worse before it gets better in terms of the loss function). To account for such cases, an additional parameter known as “patience” can be set up. It denotes the number of epochs we will patiently wait – even though the model does not show any sign of improvement. The code below shows an example:

```
EarlyStopping(monitor='val_loss', mode='min', verbose=1,
              patience=50)
```

Determining the exact number of steps to wait may vary based on the given problem. It often helps to visualize the performance measure. Another important aspect to consider in practice is the rate of improvement. Typically, we might be interested in avoiding very minor changes as improvement, and set a specific parameter known as “min_delta” to determine the satisfactory improvement rate. For example,

```
EarlyStopping(monitor='val_accuracy', mode='max',
              min_delta=1)
```

3.3 L1 and L2 Regularization

Occam’s razor is an important philosophical principle that states that given two different explanations for a specific situation, the simplest one is more likely to be correct. A similar principle finds wide implementation when building machine learning models. Given input training data and a desired target, there might exist several models that explain the training data. The underlying principle is that simpler models are less likely to suffer from overfitting compared to more complex models.

The goal is to develop algorithms that perform well both on the training data and new previously unseen samples. The so-called regularization techniques refer to a set of methods that help to lower the over-complexity of the model and to develop simpler models that promote generalization, thus preventing overfitting. Regularization techniques play a crucial role in improving the performance of machine learning models, and neural networks in particular. An effective regularization technique will identify a favorable tradeoff between bias and variance, yielding a model with significant reduction in variance, but that does not overly increase bias.

In our context, a simpler model is one that has fewer parameters. Therefore, with the objective of fighting overfitting, a good strategy would be to enforce constraints on the complexity of the model, in order to favor simpler models. To achieve that, a small penalty is typically added to the model's weight parameters to force weights to take smaller values, which, in turn, makes their distribution more regular. This process is commonly known as weights regularization. There exist two types of weights regularization.

L1 Regularization

Weights decay and weights regularization are techniques that penalize the neural network for having large weights, and therefore add a cost to the training loss. **L1 regularization** occurs when the added cost is a fraction of the absolute value of the weight coefficients. More formally, we denote the objective function as $f(X,\ y,\ \ W)$, where $X,\ y$ and W represent respectively the inputs, output and weight parameters. A hyperparameter, say λ, will control the extent of the added cost. In particular, L1 regularization can be defined as

L1 regularization
a technique used to fight overtraining by helping control the complexity of the model by focusing on the total number of features

$$L_1(X,\ y,\ W) = f(X,\ y,\ W) + \lambda \cdot \sum_{i=1}^{n} |w_i|$$

That is, L1 regularization on the model parameter is defined as the sum of absolute values of individual parameters. The L1 regularization technique results in a solution that is more sparse. In our context, sparsity refers to some parameters having a value of zero. In other words, it means that L1 regularization can lead to a neural network whose neurons only use a subset of their most important inputs and are not significantly affected by "noisy" or irrelevant inputs.

L1 regularization aims to optimize the parameters of a neuron or a layer. It is relatively easy to be implemented and has shown to provide robust solutions when dealing with outliers in the input data.

In the context of a neural network, the L1 regularization technique affects all the weights of the network. What that means is that the network will undergo a continuous transformation while its learnable parameters are tweaked to fight overfitting and produce a model that generalizes well and is able to perform effectively when facing new data it has not encountered before.

L2 Rregularization

L2 regularization, also known as "weight decay,"is the most common form of regularization techniques. The strategy behind L2 regularization is to drive the weights closer to the origin by adding a regularization term in the objective function. The cost that is added is proportional to the square of values of the weight coefficients. More formally, following a similar representation as above, L2 regularization can be defined as

L2 regularization
a technique used to fight overtraining by helping control the complexity of the model, focusing on the weight of features

$$L_2(X,\ y,\ W) = f(X,\ y,\ W) + \lambda \cdot \sum_{i=1}^{n} |w_i^2|$$

Note that when calculating the square of the parameters, the resulting values will always be positive numbers, and ,therefore, their sum will also be a positive number that will never go down to zero. As one can observe, L2 regularization penalizes the squared mag-

nitude of all parameters favoring dispersed weight vectors. In more practical terms, this encourages the neural network to rely on all its inputs, even if some of them have a small influence, rather than relying heavily on a minor set of inputs.

L2 regularization is proven to behave well in practice when most, or all, of the input features influence the output target and the network's weights are approximately of the same size. In contrast with L1 regularization, weights are not decayed to zero, though they become relatively small. Though L2 regularization might not be robust to outliers, it is an effective technique used to help learn complex data patterns.

Comparison of L1 and L2 Regularization

Both L1 and L2 regularization techniques share the common trait of helping the neural network perform well, not only on the training data, but also on new, previously unseen inputs. They aim to reduce the test error while not increasing the training error and improving the generalization capabilities of the model.

A key distinction between these two regularization techniques is the fact that L2 regularization encourages weights toward zero (yet, not exactly zero), while L1 regularization encourages weight values to be zero, thus resulting in a solution that is more sparse. Observe that smaller weights help by lowering the influence of hidden neurons. Therefore, the neural network learns to neglect those neurons, while its overall complexity gets reduced. L1 regularization generates models that are simpler and more interpretable but suffer when they have to learn complex data patterns.

L2 regularization, conversely, is used more effectively to learn complex data patterns. It performs better in cases where a fully connected network better represents the relationship between the output variable and the input features. L2 regularization is usually more efficient computationally.

Both techniques are important tools in fighting overtraining. In fact, both methods are often combined in practice to yield better results. In their work, Zou and Hastie (2005) introduced a new regularization technique, known as elastic net, which is a linear combination of L1 and L2 regularization aiming to take advantage of the major benefits of each. Elastic net regularization offers a sparse model with high prediction accuracy while encouraging grouping of strongly correlated predictors.

In summary, L1 and L2 regularization help control the complexity of the model, which is not merely a matter of identifying the right size of the network with the right number of parameters. Rather, it is aimed to find a model that is regularized appropriately, robust to noises or irrelevant information, that generalizes well, and can perform effectively, both on training input and previously unencountered data.

Implementation of L1 and L2 Regularization

L1 and L2 regularizations can be implemented in Python by taking advantage of the built-in functions of the Keras library. Typically, a kernel regularizer is added to the network's layers, as demonstrated in the code below:

```
from keras import regularizers

model = models.Sequential()

model.add(layers.Dense(16,
kernel_regularizer=regularizers.l2(0.001),
activation='relu', input_shape=(10000,)))

model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
activation='relu'))

model.add(layers.Dense(1, activation='sigmoid'))
```

This code shows a sequential model that contains three dense, i.e., fully connected layers, using the rectified linear unit (relu) activation function in the first two layers and the sigmoid function in the last layer. Note that "l2(0.001)" means that l2 weight regularization is being implemented by adding to the total loss of a network a coefficient of 0.001 in the weights' matrix of that respective layer. Weight regularization has been proven an effective technique in many practical applications resulting in models that are more resistant to overtraining.

3.4 Dropout

Another technique used for fighting overtraining has been proposed by Srivastava et. al. (2014), commonly known as **dropout**. The idea is to introduce a probability of dropping out, i.e., totally ignoring, a particular set of neurons in the hidden units during a step of the training phase. Note that such neurons might be activated again at other steps of the training procedure.

Dropout
a regularization technique where randomly selected neurons are ignored during the training phase

Though the strategy might seem simplistic and arbitrary, it has proven to be very effective in many practical situations. The authors illustrated the dropout technique with the example of how banks continuously move their tellers to random positions, so that it would be more difficult for tellers to cooperate to defraud the bank. In the context of neural networks, randomly removing a set of neurons at each iteration step would help to avoid the creation of random patterns that do not have any significance in mapping the input data to the target output.

Dropout in a Neural Network

Training a neural network with the addition of the dropout technique prohibits neurons to adapt only with their neighbors and forces them to be as impactful as possible. This way, as the dropout continues removing random neurons at each step, the neural network learns to not depend only on a particular set of neurons and becomes stronger, more resilient, and more robust. Thus, the network is generally less affected by minor changes in the input, and, overall, learns how to better adapt and generalize.

To better understand the effect of the dropout technique in fighting overtraining, note the following: because of the random dropout, a unique neural network is generated at each step of the training phase. Since each neuron can either be present or absent, then the number of possible networks grows exponentially. Of course, such networks are not independent, because they share a large number of weights, but still, the multitude of potential networks are different. This allows the resulting neural network to be treated as an average ensemble of all these different networks. That can be a practical advantage in combatting overtraining as it produces more robust neural networks.

Implementing the Dropout Technique in Keras

The dropout technique is relatively easily implemented using the Keras library. During the training phase, a dropout layer is added which randomly drops out some inputs by setting them to zero. When the training phase is completed, inputs are passed to subsequent layers. The following code, as shown in Géron (2019) applies a dropout rate of 20 percent before every dense layer of the neural network:

```
model = keras.models.Sequential([
keras.layers.Flatten(input_shape=[28, 28]),
keras.layers.Dropout(rate=0.2),
keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
keras.layers.Dropout(rate=0.2),
keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
keras.layers.Dropout(rate=0.2),
keras.layers.Dense(10, activation="softmax")
])
```

In practice, it is not always straightforward to know which dropout rate to choose for a better efficacy of the neural network. A good practice is to increase the dropout rate if the model starts to overfit the training data, and to decrease the dropout rate if the model underfits the training data. Furthermore, another common practice is to keep the dropout rate in proportion to the size of the layer, i.e., a larger dropout for a larger layer.

Adding extra dropout layers comes with an additional computational cost. However, since the dropout technique typically enhances the performance of the neural network, it is worth the time and the effort.

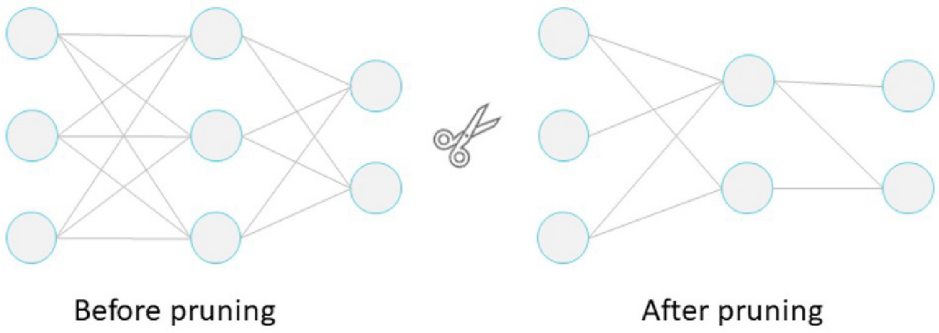
3.5 Weight Pruning

Pruning
a regularization technique that fights overtraining by shrinking a neural network while minimizing the loss in accuracy and performance

Neural networks typically increase in size especially when aiming to learn complex patterns. This growth, however, often comes with an additional computational cost. **Pruning** a neural network is a regularization technique which aims to compress the size of a neural network while attempting to minimize the losses in the accuracy or the performance of the network. The ultimate goal of the pruning process is to convert a large network to a smaller one while aiming to keep the same accuracy and performance.

In principle, pruning is the process of shrinking a neural network by removing parts of it. These parts might include individual parameters, such as weights, or even larger units, such as neurons. Knowing what and how to prune becomes an issue of crucial importance. Typically, there are two different modes to prune a neural network. First, we can prune the weights of a network. This is achieved by setting some of the weights to zero. This technique decreases the number of parameters of the network while keeping the architecture the same. Alternatively, a second way to prune a neural network would be to remove entire nodes. This causes a change in the architecture of the neural network while aiming to maintain the same level of accuracy. The figure below provides an illustration. A fully connected neural network is pruned both in terms of removing some connection weights, but also entire nodes. As one can observe, after pruning, the neural network contains fewer parameters and fewer nodes.

Figure 24: Pruning a Neural Network



Source: Evis Plaku (2023).

How to prune

Extensive research has focused on addressing the issue of how to prune. Pruning based on weights has become popular as it is a technique that does not affect the structure of the network. It requires, however, to deal with sparse computations because of setting some of the parameters to zero. Conversely, pruning the nodes allows dense computations but comes with the cost of changing the structure of the network, which can potentially damage the accuracy of the model. Regardless of which pruning technique is chosen, the key objective remains the same: to remove the less important parameters or nodes so the accuracy and performance of the network is not affected. General guidelines commonly involved when deciding what to prune are listed below:

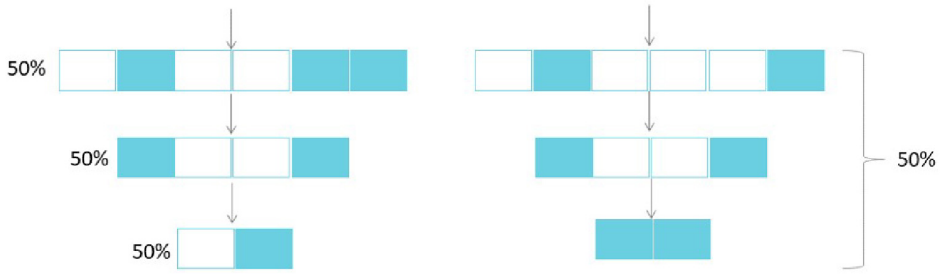
- Determine the significance of each neuron.
- Rank neurons according to their significance, assuming there is a strictly defined measure for comparing neurons according to their relevance.
- Prune the least significant neurons.
- Decide when to stop pruning based upon a termination condition.

A common pruning technique, known as weight magnitude criterion, is based on the weight value. Weights that are close to zero (comparison based upon a threshold) are removed from the network. Another criterion that is used for pruning is based upon how often a neuron is activated during the training phase. The assumption is that neurons that are rarely activated are less likely to have a significant impact in building the model, therefore, they are good candidates to be removed. Other techniques remove neurons that seem to be redundant. That means that if two neurons in a certain layer have very similar, if not identical, weights and activations, they are behaving almost the same way. Removing them from the network should not diminish the learning capabilities of the network.

One other aspect that needs to be considered is to decide how to apply the chosen criterion for pruning. Two choices are typically available: global or local pruning. Global pruning means that the selected pruning criterion will be applied globally to all parameters of the network. Oftentimes, this technique leads to better results, though it suffers from cases when a certain layer can collapse by being pruned completely.

An alternative to avoid these issues is to perform local pruning, which allows to prune only at layer-level, that is, pruning at the same rate at each layer. The figure below provides an illustration. Observe that the left part of the figure, which denotes local pruning, applies the same pruning rate to each layer. The right side of the figure illustrates how global pruning applies the selected criteria to the neural network as a whole.

Figure 25: Local Versus Global Pruning



Source: Evis Plaku (2023).

When to prune?

Another important aspect in the pruning process is deciding when to prune. A common approach is “train, prune and fine-tune.” As the name inplies, this strategy performs the pruning process after the training phase is completed. However, it might be often the case that the performance of the model suffers from (additional) pruning. To address such an issue, the neural network is trained again for a few extra iterations so it can recover from the loss caused by the pruning process. The last two steps can be iterated, while the per-

formance of the model is monitored closely. In practice, iterating has shown to improve the overall performance of the model, but it comes with the additional cost of increased computation and training time.

SUMMARY

In this unit, you learned that overtraining is a common pitfall for machine learning algorithms. It happens in the cases where the model tries to fit the training data entirely, as if it memorizes not only patterns and relations in the data, but also random fluctuations and noise. These models behave poorly when applied to a different set of data that it has not encountered before. Such models are said to not be able to generalize well to previously encountered instances.

We investigated a set of approaches used to fight overtraining, known as regularization techniques, and discussed their effects in improving the performance of a machine learning model.

You learned that early stopping is a regularization technique that continuously measures the overall error of prediction against a validation set and interrupts the training phase when it observes that the validation error increases, even though the training error might continue to decrease. In addition, you were presented with L1 and L2 regularization strategies which aim to combat overtraining based on the underlying principle that simpler models are less likely to overfit. Such techniques penalize models that have a larger number of parameters.

You were also presented another regularization technique, dropout, which introduces a probability of eliminating certain neurons during a step of the training phase. The goal is to allow neurons to be as impactful as possible and to make the network stronger, more resilient, and more robust. Pruning was also introduced as a regularization technique that aims to compress a network by removing only those connections or neurons that do not play an important role in the learning process of the network, and thus, will not (significantly) affect its performance.

UNIT 4

CONVOLUTIONAL NEURAL NETWORKS

STUDY GOALS

On completion of this unit, you will be able to ...

- understand the inspiration behind convolutional neural networks and their many applications.
- explain the role of different types of layers in a convolutional neural network.
- learn the purpose and importance of the architectural design of a convolutional neural network.
- analyze popular convolutional neural networks and highlight their key advantages.

4. CONVOLUTIONAL NEURAL NETWORKS

Case Study

Lukas is a software engineer and his friend Anne is a neuroscientist. “Look around you,” Anne said to Lukas. “What do you see? Is it merely shapes or symbols, or is it more? How amazing it is that we can recognize the objects that surround us instantaneously, with little to no effort, and then after detecting them, we convey meaning. I know that computers can do the same,” she continued. “They can observe basic shapes, colors, and textures and then learn how to assemble it all together into detecting and recognizing whole objects.”

convolutional neural networks
a deep learning algorithm commonly used to identify and recognize objects in image data

Lukas felt a bit surprised, but soon he understood that she was talking about **convolutional neural networks** (CNN): a special class of neural networks which is most commonly applied to analyze visual information. “CNNs are the “eyes” of the computer,” Lukas said, “and I want to learn more about them. I want to learn what they are composed of, and how we can build CNNs to learn how to recognize objects. I wonder,” Lukas added, “if we can effectively do that for large sets of images that contain a wide range of objects.”

4.1 Motivation and Applications

Since the rise of artificial intelligence (AI), a lot of work has focused on building computer models that are able to effectively perform tasks that, for humans, might seem trivial. The ability to quickly recognize a cute dog in a picture is something that all humans proudly do almost effortlessly. Yet, it was not until recently that a computer program could reliably identify the cute dog, or any other specific object in an image.

Convolutional neural networks (CNNs) are a type of neural network that have been inspired from the study of the brain’s visual cortex and have been used to recognize objects in image data for several decades now. However, in the last few years, due to the significant increase in computers’ computational power and the large availability of training data, CNNs have been used extensively and with remarkable results in many complex tasks.

Convolutional neural networks have a lot of similairities to common neural networks. Their core unit is the neuron and they both aim to use training data to learn a model that identifies patterns and underlying relationships between input features and target outputs, while continuously tweaking the network’s parameters as they learn. So, then, what is different? Why are convolutional neural networks so popular?

Regular neural networks typically suffer from a rapid increase of the number of parameters when the number of layers in the network is increased. This adds an important computational burden. Moreover, tuning a large number of parameters becomes a huge task. Convolutional neural networks perform more effectively not only in terms of computa-

tional time, but also in their ability to diminish the number of parameters without affecting the quality of the model. This is a remarkable achievement, especially when dealing with image data, because the values at each pixel of the image are considered as a feature.

Furthermore, convolutional neural networks have demonstrated noteworthy abilities in better "understanding" abstract concepts in image data. CNNs work well in gradually constructing a model that detects a particular object by moving from rudimentary shapes, such as lines or arcs, to learning how to represent more generic concepts, such as a particular piece of the object until being able to fully recognize it.

Another advantage of CNNs is that they learn how to perform feature extraction effectively. The key assumption is that the input data are images that encode certain specific properties into the architecture of the network. The CNN then assigns the relative importance (in terms of weights and biases) to various aspects of the image and further processes them to generate invariant features that are passed from one layer to another with the aim of achieving one final output that allows to correctly classify the object.

Convolutional neural networks generally require less preprocessing of the input data compared to other classification algorithms and have demonstrated good abilities to learn intrinsic characteristics of image input data. CNNs have been successful in addressing many complex computer vision problems, including the following:

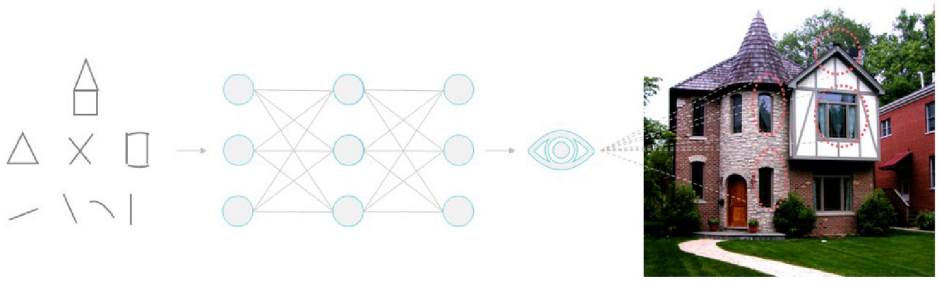
- object detection: CNNs have been effectively applied to recognize objects in an image by classifying them based upon key features and/or patterns identified in the image. The detected objects vary from a large and wide range, including everyday items, animals, or objects that can be used in drones or self-driving cars.
- face and facial emotion recognition: A lot of CNN frameworks have been equipped with the ability to recognize faces within an image by detecting various features, such as eyes, nose, and mouth with great accuracy. CNNs have been used to reduce the amount of distortion in a face and even to help differentiate between various facial expressions that represent emotions, such as happiness or anger.
- self-driving autonomous cars: The development of autonomous self-driving cars is a challenging task. CNNs have helped in the process of detecting traffic signs or obstacles lying in the road. In this context, convolutional neural networks have been used in integration with other techniques, such as reinforcement learning to help the model learn how to respond appropriately when encountering this visual information.
- words recognition, prediction, and translation: A common application of convolutional neural network is that of training a model to distinguish handwritten characters in various languages and contexts. Important progress is being made in the ability of CNNs to even predict the next word that might follow in a particular sentence. CNNs have also been used to facilitate automatic translation between languages with a high degree of accuracy.
- medical analysis: A common implementation of CNNs include detecting abnormalities in medical images. Convolutional neural networks that have been trained on large medical image datasets have shown good accuracy in identifying such abnormalities, in some cases, even outperforming human doctors.

4.2 Convolution and Image Filtering

The work of David H. Hubel and Torsten Wiesel on the structure of the brain's visual cortex gained them a Nobel prize in Medicine in 1981. They performed a series of experiments on cats and monkeys and demonstrated that many neurons only react to a visual stimulus that is located in a limited region of the visual field (Hubel & Wiesel, 1959;1968). Furthermore, the authors showed that some neurons react only to images of horizontal lines, while others react to lines with different orientations. They also observed that some neurons react more vividly to complex patterns that are formed as a result of combinations from lower-level patterns. This architecture empowers the visual cortex to detect all sorts of complex patterns in the visual field.

The figure below provides a simplified illustration. An object (in this case, a house) is part of a larger image. The dotted lines represent the local receptive fields of the visual cortex. The objective is to extract features from the received visual information, build upon those features, and construct different layers of abstractions, such that a convolutional neural network is finally able to process the information it receives in such a way such that it can learn how to effectively classify the object as a house.

Figure 26: Local Receptive Fields in the Visual Cortex



Source: Evis Plaku (2023) based on Géron (2019).

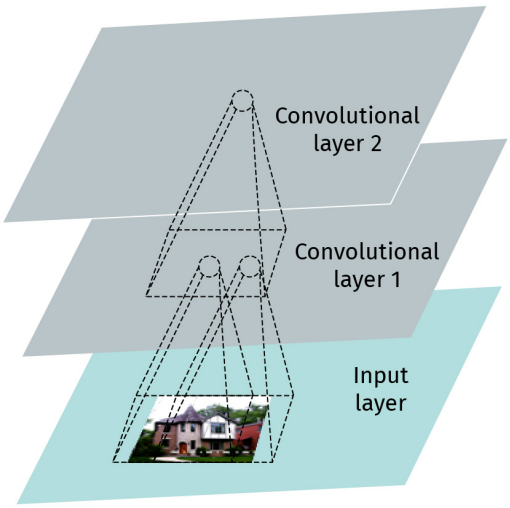
Convolutional neural networks, in contrast with common neural networks, have three main types of layers, the convolutional layer, pooling layer, and fully connected layer. The objective of the CNN is to reduce the image data it receives into a simpler form that is easier to process without losing features that are important for achieving a high accuracy prediction. Typically, each convolutional neural network contains a convolutional layer as the first layer of the network, which is then followed by other convolutional or pooling layers. The fully connected layer is usually the final layer of the CNN. As the image data gets processed through the layers of the CNN, classification starts with simple features (e.g., color and edges) and builds up to identify larger elements (e.g., shapes) and larger abstractions until the whole object is finally recognized.

Convolutional Layer

Convolutional layer
As the main building block of a CNN, it contains a set of filters, parameters of which are to be learned throughout training

The **convolutional layer** is a crucial building block of a CNN. Given an image input data, the neurons of the first convolutional layer are connected only to pixels of the image that are in their respective receptive field, and not to every single pixel of the image. Similarly, neurons of the second convolutional layer are only connected to a small receptive area coming in from the first layer. This hierarchical structure allows the network to first concentrate on a small set of low-level features and then learn how to assemble them together into more general features in the subsequent layers. The image below provides an illustration. The input image data is processed via two convolutional layers where each layer is represented in two dimensions.

Figure 27: CNN Layers With Rectangular Local Receptive Fields



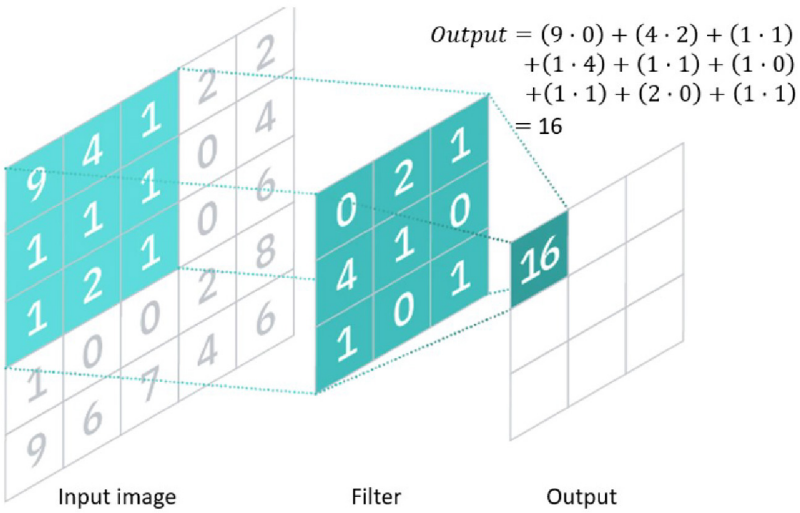
Source: Evis Plaku (2023), based onGéron (2019).

The convolutional layer works by relying on several key components such as the input data, a filter, and a feature map. Typically, the input data is an image, for example, a color image composed of a matrix of pixels represented in three dimensions: width, height, and depth which correspond to the red-green-blue values of an image. The filter is a kind of feature detector that moves across the image data checking if the feature is present in the respective local receptive field. Each filter has a relatively small width and height and it moves through the image covering also its full depth. For example, a filter might have a size of 5 x 5 x 3 meaning that it has a width and height of five pixels, and a width of three pixels corresponding to the three-color channels.

A process known as convolution represents the filter moving across the image data to check if the feature is present. In particular, the filter is applied to a portion of the image data and computes the dot product between the respective pixels of the image and the filter itself. It then slides to another part of the image and repeats the same procedure. At the end, this yields the aggregated results in the form of a two-dimensional map called a

feature map, or activation map. Eventually, the convolutional neural network will learn filters that activate when they encounter some type of visual features. These visual features usually denote very simple properties in the first layers and more complex and general ones in the other layers. When the whole process terminates, we will have a separate set of activation maps produced by these filters. The activation maps are stacked together to generate a final output.

Figure 28: Example of Applying a Filter to an Image Array of Weights



Source: Evis Plaku (2023).

The figure above provides an illustration. Part of an input image is represented as a two-dimensional array of weights, as shown in the left-hand side of the figure. A 3 x 3 filter matrix represents in our context the dimensions of the receptive field. This filter is applied to the highlighted area of the image producing the dot product as a result. This process will be repeated until the filter is applied throughout the whole image input data. Note that an output value in the feature maps does not connect to each pixel in the image data. In fact, it only connects to the respective receptive field that is applying the filter.

This characteristic of connecting the convolutional layers only partially is also called local connectivity. The depth axis of the filter is usually the same as the depth dimension of the input volume. However, the weight and height are only mapped partially, not fully. This asymmetry in spatial dimensions is a key element of convolutional neural networks.

Spatial arrangement

So far, we have discussed the structure of the convolution layer, but have not yet explained how the neurons are arranged in the output volume. The three hyperparameters that determine the size of the output volume are as follows:

1. The first hyperparameter is the depth of the output volume. It denotes the number of filters to be used. For example, if we use three distinct filters then we would create three different feature maps leading to a depth of three.
2. The second hyperparameter is known as the **stride**. It determines the step size that we use to slide through in the input matrix while using the filter. Typically, we use a stride of one or two. Note that the larger the stride, the smaller the output.
3. It is often desirable to have the same dimensions of the input and output. To achieve that, we add a padding to the input volume by surrounding it with zeroes around the border. This procedure allows the filters to fit the input image and it is controlled by another hyperparameter, known as **padding**.

Stride
The stride is a parameter of the CNN's filter that determines the amount of movement over the image data.

Padding
refers to the number of pixels added to an image when it is being processed by the filter of a CNN

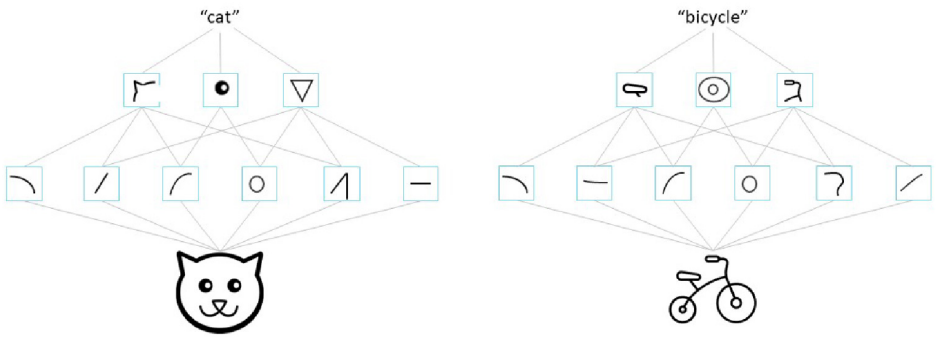
There are three types of padding:

1. Valid padding, which is also known as zero padding, assumes that all dimensions are valid so that the input image data can be covered by the specified filter and stride.
2. Same padding, which denotes a case where the output produced is of the same size as the input.
3. Full padding, which increases the output size by surrounding it with zeros alongside the borders.

Convolution layers have a key distinctive feature compared to fully connected layers that are typical for regular neural networks. Dense layers aim to learn global patterns from their input feature space, while convolution layers focus on learning local patterns. An image, for example, can be broken down into smaller pieces that convey patterns and meaning, such as edges, textures and so on.

These local patterns are not affected by translation of position. What that means in practice is that if a CNN is able to learn how to identify a part of an object (for example, the nose of a cat) in the lower part of an image, it is able to detect it in all other parts of the image as well. This increases efficiency because fewer training instances are needed to learn a model that generalizes well. Additionally, CNNs are able to learn hierarchical patterns that move from small units, such as edges or corners, to bigger levels of abstractions (for example, body parts) until the object is finally detected as a whole. The figure below provides an illustration.

Figure 29: Visual Spatial Hierarchy



Source: Evis Plaku (2023).

Consider, for example, two different objects: a cat and a bicycle. If our convolutional neural network is trying to detect the object contained in an image, it might consider the object as the sum of its parts. For example, a bicycle would be comprised of wheels, pedals, frame, etc., while a cat will be comprised of ears, nose, and eyes, to name a few. Initially, a convolution layer would be able to detect only low-level patterns, such as lines, edges or shapes, but then, at higher-level layers it would discover and identify some parts of the object (for example the nose of a cat or the wheel of a bicycle). Moving up in the hierarchy of layers would empower the convolutional neural network to assemble these smaller parts into higher-level representations and finally be able to detect the whole object.

Pooling Layer

Pooling layer
The pooling layer is a component of the CNN whose objective is to progressively reduce the spatial dimensions and number of parameters in the network.

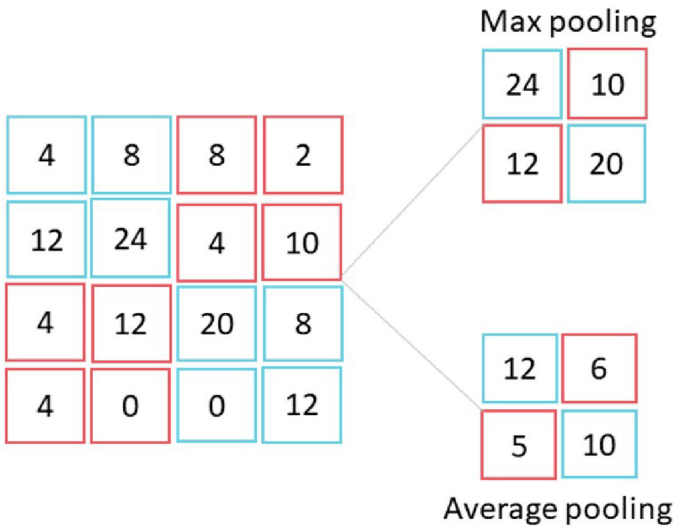
The key objective of a **pooling layer** is to reduce the dimensions of the input image to lighten the computational burden and to reduce the number of parameters used. Continuously shrinking input images also helps control overtraining. Similarly to the convolution layer, the pooling layer also operates with a filter that slides across the input image. This filter, however, does not have any weights. Instead, it identifies the neurons located with the receptive field and aggregates the inputs using an aggregation function such as the maximum of the mean. This gives rise to two different types of pooling:

1. **Max pooling**: When the filter slides across the input image, it selects the pixel with the maximum value and sends it to the output array. The filter then moves across to swipe the whole input image repeating the same procedure. This type of pooling is used more commonly than average pooling.
2. **Average pooling**: This calculates the average value among the pixels identified by the filter in its local receptive field and sends the selected value to the result array.

Regardless of the type of pooling used, a lot of information is lost. Yet, this process is beneficial for the CNN because it helps reduce the complexity of the network and fights overfitting.

The figure below provides an illustration. Given a small representation of an image matrix, a 2 x 2 filter slides across the image, as denoted by the different colored boxes. Max pooling selects the pixel with the highest value, while average pooling calculates the average of pixels in each box (representing the local receptive field).

Figure 30: Types of Pooling



Source: Evis Plaku (2023).

Fully Connected Layer

Fully connected layer aims to compile the data extracted from the previous layers to form the final output

The main objective of the **fully connected layer** is to empower the neural network to learn non-linear relationships that map input features into the desired target. In convolutional neural networks, as in regular ones, a fully connected layer is the one whose neurons have full connections to the neurons of the previous layers.

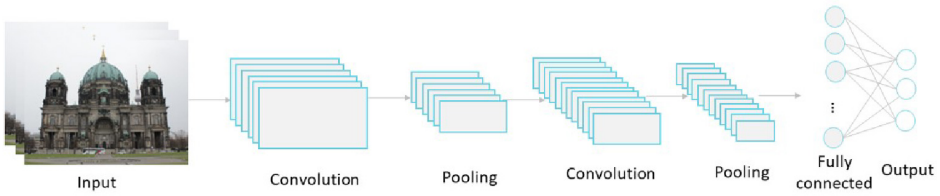
A convolutional neural network might have more than one fully connected layer. The first fully connected layer receives input (from the final pooling or convolution layer) and flattens it. What flattening means is that it unrolls the output received as a three-dimensional matrix into a one-dimensional vector and then applies the activation function. The activation function continuously transforms weights and biases, allowing the network to tweak parameters and increase its learning capabilities. Next, the role of the last layer is to output the final probabilities that an instance belongs to any of the classification categories. For that reason, such a layer typically uses a “softmax” activation function that assigns an estimated probability ranging from zero to 1.

4.3 CNN Architecture

The architecture of a convolutional neural network plays a crucial role in the ability of the network to learn complex patterns and effectively classify new instances. Though there are many variations and architectures of CNNs, they share common traits. A typical CNN contains a stack of a few convolution layers, followed by at least one pooling layer, and then again, several convolution layers followed by pooling layer(s). This arrangement might be repeated a few times before a few fully connected layers empower the network to learn non-linear relationships and output a model that classifies the input data it receives.

The general objective of these architectures is to continuously shrink the image size (this diminishes the computational load and the number of parameters used) as we progress in the stack of layers, but to do that without compromising the ability of the network to learn meaningful patterns. Because of the convolution layers, a CNN is able to go deeper and deeper in extracting patterns and it can move from learning basic properties to learning more complex abstractions until a full representation and classification of the object is achieved.

Figure 31: Architecture of a Convolutional Neural Network



Source: Evis Plaku (2023).

The figure above presents an illustration of a convolutional neural network containing several convolution layers followed by pooling layers, and then again, the same arrangement before containing a few fully-connected layers and a final layer that outputs the prediction of the model.

Implementing a CNN

We can implement a simple CNN by taking advantage of Keras built-in functionalities. For instance, we can use such a network to classify instances of the famous fashion MNIST dataset containing clothing items categories in ten different classes as follows (Xiao et al., 2017) (Géron, 2019):

```
from functools import partial

DefaultConv2D = partial(keras.layers.Conv2D,
kernel_size=3, activation='relu', padding="SAME")
```

```
model = keras.models.Sequential([

DefaultConv2D(filters=64, kernel_size=7,
              input_shape=[28, 28, 1]),

keras.layers.MaxPooling2D(pool_size=2),

DefaultConv2D(filters=128),
DefaultConv2D(filters=128),
keras.layers.MaxPooling2D(pool_size=2),

DefaultConv2D(filters=256),
DefaultConv2D(filters=256),
keras.layers.MaxPooling2D(pool_size=2),

keras.layers.Flatten(),
keras.layers.Dense(units=128, activation='relu'),
keras.layers.Dropout(0.5),
keras.layers.Dense(units=64, activation='relu'),
keras.layers.Dropout(0.5),
keras.layers.Dense(units=10, activation='softmax'),
])
```

Observe how the architecture of this simple CNN is mapped into code:

- The “partial” function allows us to wrap the hyperparameters we will commonly use throughout the network in the defined “DefaultConv2D” class.
- The first layer uses a large kernel size of 7 x 7 to swipe across the input images that have dimensions of 28 x 28 pixels and only one color channel (i.e., grayscale).
- The max pooling layers allows the network to shrink the input images by a factor of two, as denoted by the “pool_size” parameter.
- Then, a structure formed of two convolutional layers followed by a pooling layer is repeated twice. Note that if the image size were larger, we could have repeated this block more often to be able to reduce the dimensionality of the images and to go deeper into understanding underlying patterns.
- Another interesting pattern to denote is the fact that the number of filters increases as the network moves from the input to the output layer. Note that the number of basic features is relatively low (e.g., lines, circles, and irregular shapes), but as we progress toward the output there are many different ways how to combine them into forming object parts. Since we are shrinking the image dimensions by a factor of two, then increasing the number of feature maps by the same factor will not lead to an unmanageable computational load.
- Finally, the fully connected layer flattens the output, two dropout layers are added to combat overtraining and to help the model generalize before the last dense layer is added, which outputs the classification into the ten categories of the fashion MNIST dataset.

4.4 Popular Convolutional Networks

Remarkable advancements have been made over the years in using convolutional neural networks for image detection and classification. Researchers have developed a wide variety of architectures, which are often tested in a popular competition such as the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) (Russakovsky et al., 2014). In this competition, CNN models are tested against high volume datasets containing images of over 1,000 classes of objects, some of which have only very minor differences. Trying to correctly classify more than 100 different dog breeds seems like an impossible task for a human, and yet these CNN models have done an outstanding job. Some of the most popular CNN architectures are presented below.

LeNet-5

Created by LeCun et al. (1998), the LeNet-5 architecture is a classic example. It is widely used to classify images of hand-written digits like the famous MNIST dataset. It grew in popularity because the model is simple, straightforward, and effective. The architecture of LeNet-5 is composed of three sets of convolutional layers alternated with two sets of average pooling layers, and is then followed by two sets of fully-connected layers. The table below presents details of the most important components of LeNet-5 architecture.

Table 1: LeNet-5 Architecture

Layer	Type	Maps	Size	Kernel size	Activation
Out	Fully connected		10		RBF
F6	Fully connected		84		tanh
C5	Convolution	120	1 x 1	5 x 5	tanh
S4	Average pooling	16	5 x 5	2 x 2	tanh
C3	Convolution	16	10 x 10	5 x 5	tanh
S2	Average pooling	6	14 x 14	2 x 2	tanh
C1	Convolution	6	28 x 28	5 x 5	tanh
In	Input	1	32 x 32		

Source: Evis Plaku (2023).

Note the following important points:

- The input layer receives images composed of 28 x 28 pixels, but the images are padded so that they become 32 x 32 pixels. The input layer, however, is the only layer that uses padding. As the image data are processed in the next layers of the network, they continuously shrink in size.
- Layer C1 is a convolution layer containing six convolution kernels of dimensions 5 x 5. The activation map is 28 x 28.

- Next, the average pooling layer S2 generates six maps of size 14 x 14. Each cell in the function map is connected to a 2 x 2 neighbor at the respective function in layer C1.
- Similarly, the input image data gets processed in the next convolution and pooling layers. Note that according to the architecture proposed by LeCun et al., the average pooling layers behave in a slightly more complex manner. In particular, the mean of input neurons is multiplied by a learning coefficient and added to a bias term. These operations are performed for each map, and then the activation function is finally applied.
- The final output layer receives the information produced from the earlier steps and yields an output that estimates a probability that a particular input belongs to each of the ten possible digit classes.

AlexNet

The AlexNet architecture demonstrated remarkable success as it won the famous ILSRVC ImageNet challenge in 2012 by a large margin to its closest competitor (Krizhevsky et al., 2012). The architecture of AlexNet is similar to that of LeNet-5 in structure, but much larger and deeper. The presented model stacks convolution layers on top of each other, instead of putting a pooling layer on top of a convolution one, as we discussed in LeNet-5 architecture. The table below presents the architecture in more detail, including the rectified linear activation function (ReLU).

Table 2: AlexNet Architecture

Layer	Type	Maps	Size	Kernel size	Activation
Out	Fully connected		1000		Softmax
F9	Fully connected	4096			ReLU
F8	Fully connected	4096			ReLU
C7	Convolution	256	13 x 13	3 x 3	ReLU
C6	Convolution	384	13 x 13	3 x 3	ReLU
C5	Convolution	384	13 x 13	3 x 3	ReLU
S4	Max pooling	256	13 x 13	3 x 3	
C3	Convolution	256	27 x 27	5 x 5	ReLU
S2	Max pooling	96	27 x 27	3 x 3	
C1	Convolution	96	55 x 55	11 x 11	ReLU
In	Input	3 (RGB)	227 x 227		

Source: Evis Plaku (2023).

To fight overfitting (note that a model overfits when it behaves well while using training data, but poorly when encountering new data), Krizhevsky et al. applied two regularization techniques. First, they used the dropout technique by introducing a probability of 50

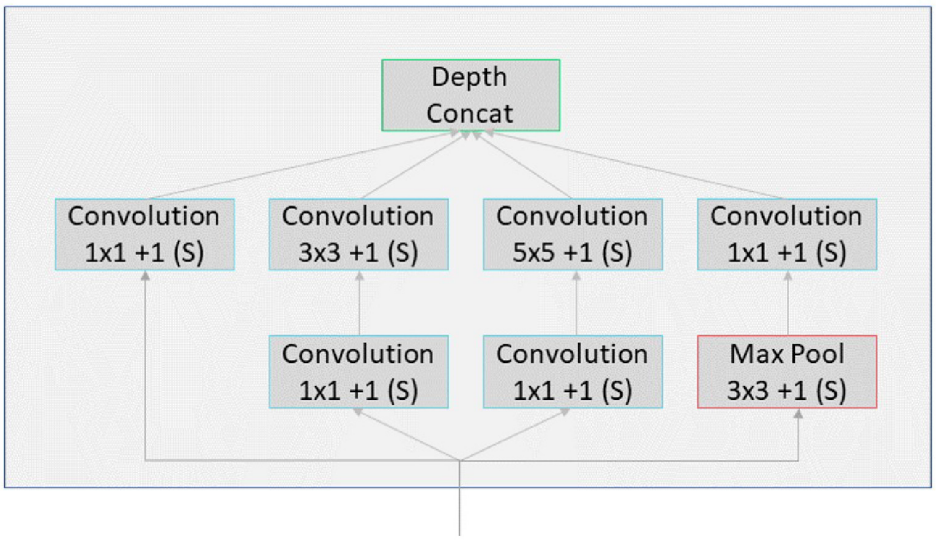
percent of simply ignoring a neuron during the training phase. Next, they performed a technique called data augmentation, which artificially increases the size of an image during the training phase by generating multiple variants of that image. These variations are created by randomly shifting the training images by various offsets, for example, changing lighting conditions. These techniques help the model to become more robust to changes and, therefore, to yield better results when tested against data it has not encountered before.

Another distinctive feature of AlexNet is the use of a normalization step called local response normalization. It aims to enforce a competitive activation of neurons by allowing stronger neurons to restrain other neurons in neighboring areas. This encourages the differentiation and specialization of various feature maps, empowering them to explore a wider range of features. This technique, too, helps with generalization.

GoogLeNet

The GoogLeNet architecture showed a great performance by winning the ILSVRC challenge in 2014. The model proposed Szegedy and his colleagues from Google Research was a much deeper network than the current state of the art (Szegedy et al., 2015). A fundamental advantage of the proposed model was the ability to use parameters more effectively than other architectures by introducing the notion of sub-networks, known as inception modules. The figure below illustrates the GoogLeNet architecture.

Figure 32: Inception Module of GoogLeNet Architecture



Source: Evis Plaku (2023).

The input data are transmitted to four different layers. The rectified linear unit (ReLU) activation function is used by all convolutional layers. Observe that the notation 1 x 1 + 1(S) means that the layer uses a kernel of size 1 x 1, with stride one and SAME padding. That

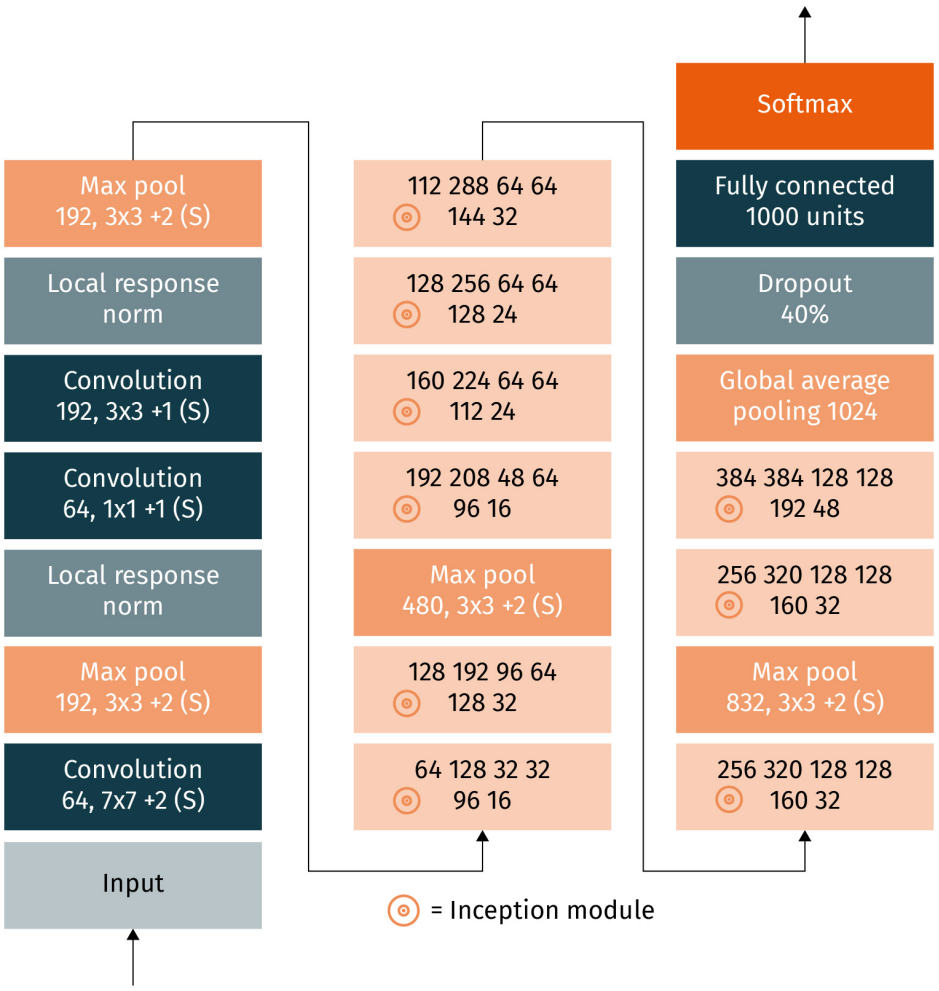
means that the outputs of the layer will have the same spatial dimensions as its inputs. To be able to better capture patterns in different scales, the GoogLeNet architecture uses different kernel sizes (1 x 1, 3 x 3, and 5 x 5) in the second set of convolution layers. Finally, the depth concat layer allows to concatenate all outputs along the depth dimension by stacking the activation feature maps from all convolutional layers.

An interesting element of the GoogLeNet architecture is the usage of layers with kernels that have dimensions of 1 x 1. These layers observe only one pixel at a time and therefore it is impossible for them to capture any features. Yet, they serve the following purposes:

- Kernels of size 1 x 1 are still able to capture patterns along the depth dimension, even though they cannot capture patterns in the width and height dimensions.
- They output fewer feature maps than their inputs, thus forming a kind of a bottleneck layer used to reduce the dimensionality of the input data, and therefore improve the performance of the model in terms of computational cost and number of parameters.
- Each pair of combined convolution layers (for example, a layer of size 1 x 1 with layer of size 3 x 3 or 5 x 5) can serve as a more powerful convolution layer that is able to capture even more complex patterns than each kernel individually.

The architecture of the GoogLeNet convolutional neural network is a deep model that includes nine inception modules as the ones described above. The ReLU activation function is used by all convolutional layers. An illustration of the architecture is presented in the figure below alongside some key characteristics of the CNN.

Figure 33: GoogLeNet Architecture



Source: Evis Plaku (2023).

Upon receiving the image input data, the GoogLeNet architecture reduces the computational load by dividing the area of the image by 16. This process is performed in the first two layers. Note that the first convolution layer uses a relatively large kernel size of 7 x 7 pixels in order to preserve a considerable amount of the initial information. As discussed above, the goal of the local response normalization layer is to empower the previous layers to learn a wide variety of features in different levels of abstraction.

The next two convolution layers serve as bottleneck layers to shrink the dimensionality of the input and to reduce the number of parameters needed for the model to learn the necessary patterns. Another local response normalization layer follows. Then, to speed up computations, a max pooling layer reduces the image size while aiming to pertain the same quality of features and information.

Once these operations are completed, the GoogLeNet architecture focuses on a large stack of nine inception modules that work with two pooling layers that help reduce dimensionality and speed up computations. The role of the inception modules is crucial in the architecture of GoogLeNet, as discussed above.

The role of the average pooling layer is to output the mean of each feature map and discard any spatial information left at this point. Note that input image data are typically of size 224 x 224 pixels for the GoogLeNet model. These dimensions are reduced in half by each of the five max pooling layers yielding images of size 7 x 7 pixels. Considering that this is a drastic reduction in the dimensionality of the image data, there is no need for this architecture to stack several fully connected layers on top, as was the case of the previously discussed architectures. This way the number of parameters used is considerably reduced.

Finally, the last layers perform a dropout regularization technique and use a fully connected layer with 1,000 units to account for the fact that there are 1,000 possible categorization classes. The role of the softmax function is to output the estimated probabilities of each input data belonging to each of the possible categories.

The GoogLeNet architecture is a highly effective and well-researched model. Several other architectures have been proposed as variants of it, which have achieved even better performance by further exploring and improving the inception modules.

ResNet

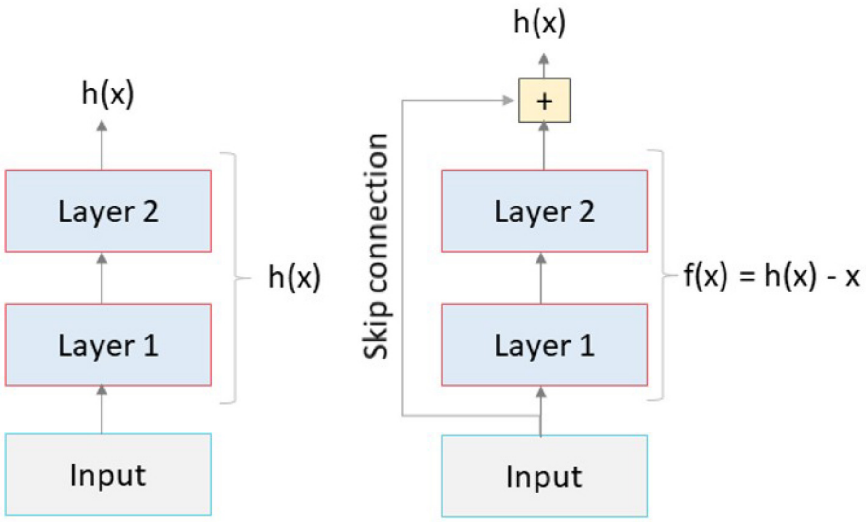
In 2015, the popular ILSVRC contest was won by a model proposed by Kaiming He and his colleagues, who used a Residual Network composed of 152 layers (He et al., 2016). Observing the history of the most successful architectures confirmed a trend: deeper models rely on effective architectures to actually use fewer parameters while getting better at detecting and classifying objects in image input data. Though usually deeper models are associated with more parameters, the key to achieving the drastic improvement in the context of ResNet is by using what the authors denote as a "skip connection" which allows to add the signal that contributes to a layer to the output of another layer that is located higher in the stack.

This approach addresses the challenges associated with training very deep networks by introducing the notion of residual blocks that allow to have direct connections that skip some layers of the model. Typically, the output of a layer is provided as input to the next layer. Residual blocks make possible to bypass several layers and connect two distant layers directly. This connection is known as skip connection or shortcut connection and allows to train deeper networks more effectively. The shortcut connection performs what is known as identity mapping, i.e., adding the output of these connections to the output of the stacked layers. Note that this process does not require any extra parameters, nor does it require more computational time.

To better understand the importance of residual blocks in speeding up and improving learning, consider the following. The objective of training a neural network is to be able to learn how to model a target function, say $h(x)$. But, if we add to the output of the network

the input x, then in fact we are skipping connections and the network should now learn how to model the function $f(x) = h(x) - x$ rather than simply $h(x)$. This process is known as residual learning. The figure below provides an illustration.

Figure 34: Residual Learning

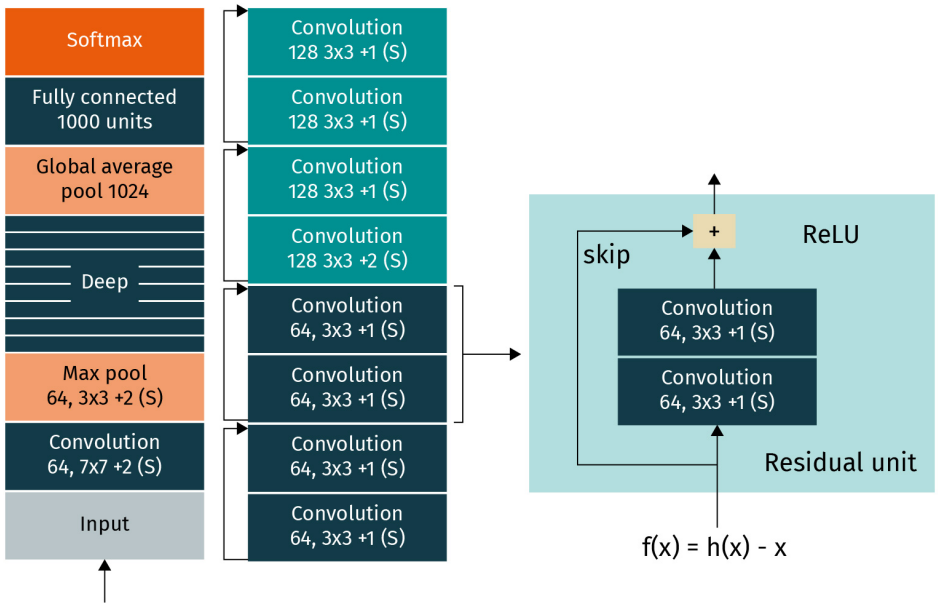


Source: Evis Plaku (2023), based on He et al. (2016).

The following highlights an important difference between the residual network and regular networks. When the weights of a regular network are close to zero (e.g., during initialization), the outputs will also be close to zero. However, if we add a skip connection, the network will now output a copy of its input. In other words, it will model the identity function. Since it is often the case that the target function is close to the identity function, the training process will be completed considerably faster. Another advantage of skip connections is that it allows the network to make progress in learning even in the cases when some layers have not yet started learning. Each residual unit can be considered a smaller network containing skip connections.

The figure below presents the ResNet architecture. Observe that it contains a lot of similarities to the GoogLeNet architecture, especially at its beginning and end. The middle part contains a large stack of residual units where each unit is made of two convolution layers using batch normalization and ReLU activation function.

Figure 35: ResNet Architecture



Source: Evis Plaku (2023). based on He et al. (2016).

SUMMARY

In this unit, you learned that convolutional neural networks emerged as a new type of neural networks most commonly applied to analyze image input data in order to detect and classify objects. We discussed how CNNs typically perform more effectively than regular neural networks in terms of computational time and by using fewer parameters. Convolutional neural networks have been successfully applied to a wide range of fields, including, but not limited to, object detection, medical analysis, face and facial emotion recognition, and self-driving autonomous cars.

We expanded our discussion of CNNs and provided insight into their architecture and the type of layers they are composed of. We addressed how the convolutional layer plays a crucial role in the CNN by continuously transforming the input image data to extract relevant features. The pooling layer then is used mainly to shrink the size of the feature maps and reduce the number of parameters needed by the model. The fully connected layers allow the network to compile the data extracted by the previous layers to form the final output classification.

We also investigated how the architecture of a convolutional neural network plays an important role in its effectiveness and the ability of the network to extract meaningful features. This can form various levels of

hierarchical abstractions until the whole object is effectively recognized and detected. We presented several state-of-the-art architectures that demonstrate the remarkable advancements that have been made over the years in using convolutional neural networks for image detection and classification.

UNIT 5

RECURRENT NEURAL NETWORKS

STUDY GOALS

On completion of this unit, you will be able to ...

- understand what recurrent neural networks are and how they work.
- know what memory cells are and the role they play in a recurrent neural network.
- identify the major challenges encountered when training recurrent neural networks.
- comprehend what a long short-term memory unit is and why it performs better on long sequences than a naive recurrent neural network.
- develop a simple recurrent neural network to process sequence data.

5. RECURRENT NEURAL NETWORKS

Case Study

Lukas is a software engineer who is experimenting with training a neural network to perform automatic translation. He finds this problem interesting, but challenging, as he is overwhelmed by the many flavors of grammatical rules and the peculiarities of the language. He is discussing this challenge with his friend, Anne, who is a neuroscientist.

“I think context is everything,” Lukus says. “I want to make the network remember the words and the sentences it has read before, but it seems to forget so easily. My network behaves like an inattentive reader: as soon as some new information comes up, everything that was read until that moment slips away from memory.”

“If only there was a way to help the network remember prior information,” Lukas continued. “This way, it would take into consideration past information and build up its understanding based on previous words and sentences. It would not throw everything away to start from scratch, but it would have a persistent memory.”

“I know what you need,” Anne answered. “You need to develop a recurrent neural network that saves information not only short term, but also works well when facing long sequences. Let's build and train it together.”

5.1 Recurrent Neurons

Imagine you are watching a baseball game. The batter hits a hard ball that flies high. You observe the left-fielder rushing in the direction of the ball, anticipating its trajectory, and adapting their movements accordingly until they finally catch the ball. In what appears to be a common baseball game movement, we can observe how tracking the ball’s movements requires “predicting” the near future by relying on past observations and actions. As humans, we inadvertently do it all the time when we finish each other's sentences or get an accurate gut feeling on how a movie ends.

Recurrent neural networks
a special type of neural network distinguished by their “memory,” which allows them to maintain a state of what the network has observed thus far

In this section, we will discuss **recurrent neural networks** (RNN), a class of neural networks that are built upon the same promise: to predict the future, at least to a certain point. More realistically, these special types of networks have been used successfully to analyze time series data, such as retail sales, stock prices, or rainfall measurements; on autonomous driving aiming to anticipate and avoid dangerous trajectories; on natural language processing tasks, such as automatic translation and sentiment analysis, and many other challenging real-world tasks.

What are Recurrent Neural Networks?

A major characteristic of regular feed-forward neural networks or convolutional neural networks is that they have no memory. As they receive data, these inputs are processed independently without remembering any intermediate information. However, in many practical cases it is important to recollect what came before. We, as humans, adopt this principle quite often. In fact, as you are reading this text, you are processing it word by word while (hopefully) creating an internal model that is built upon past information and is continuously updated when new information comes in.

RNNs embrace a similar principle. Their distinctive feature is that to predict an output, they rely on information from prior inputs while maintaining a state of what the network has observed up to that point. In fact, RNNs use a looping mechanism that allows information to flow from one step to the next. Hence, they can effectively process a sequence of information that affects the final output. RNNs possess a kind of internal memory that empowers them to make past information persist. For that reason, **memory in RNNs** are especially powerful in problems that involve sequential data and in scenarios where the context plays a critical role in predicting the outcome.

Memory in RNN
The memory in RNN refers to the ability of the network to store information from prior inputs persistentantly.

Because of their internal memory, recurrent neural networks can recollect crucial information about the input they receive at different timesteps and can learn how to form a much deeper, more meaningful understanding of the sequence of input data they are processing while leveraging on past inputs.

In traditional neural networks, each input shown to the network is processed independently, i.e., apart from the connection weights the network does not keep any state between inputs. If such a network, for example, needs to process a sequence or a temporal series of data, the entire sequence must be processed at once by the network. A recurrent neural network, however, will process the sequence by iterating through its elements and maintaining a state that contains information regarding what the network has observed thus far. This peculiar characteristic makes RNNs a powerful and robust type of neural networks in many practical applications, especially when involving sequential or temporal series of data.

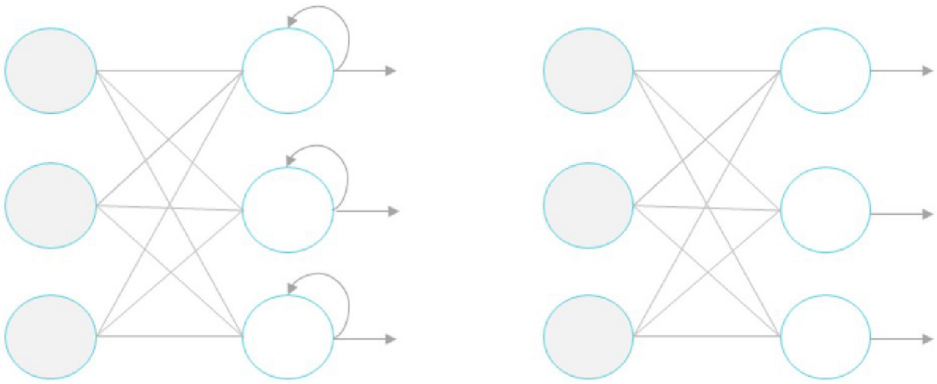
Information Flow in Recurrent Neural Networks

Neural networks are composed of nodes that process information through a set of connected layers. In a regular feed-forward neural network, information flows in forward direction (from the input layer toward the output layer, passing through hidden layers). These types of networks, however, do not have a notion of the order in time in which they receive information. They have no memory of the input they have received in previous timesteps. This can cause a serious limitation, especially when dealing with problems where, in order to predict what is coming next, it is critical to consider the state of information in several prior timesteps.

Recurrent neural networks, converesly, are structured in a way that facilitates the transmission of information through a cycle. When a given node needs to make a decision, it considers not only the current input, but also what it has learned from prior inputs. Because of their internal memory, RNNs are designed to pass information in two directions. This enables them to effectively process sequential and temporal data.

The figure below provides an illustration of the difference in information flow between recurrent neural networks and regular feed-forward neural networks. As observed, the node containing a cycle represents the ability of the RNN to allow information to persist, as in the form of a short-term memory. This memory stored in the hidden state of the network represents the "context" created from prior inputs.

Figure 36: Recurrent Neural Networks Versus Feed-Forward Neural Networks



Source: Evis Plaku (2023).

Architectural Types of Recurrent Neural Networks

While regular feed-forward neural networks usually map one input to one output, recurrent neural networks are not restricted in this regard. Instead, input and outputs of RNNs can differ in length. In this context, four common types of recurrent neural networks can be distinguished. This broad category empowers RNNs to be used effectively in many distinct scenarios, such as sentiment analysis, music generation, or automatic translation.

One-to-one

The simplest form is a RNN that maps one input to one output, called a one-to-one network. The input and output sizes are fixed. In this case, the RNN behaves similarly to a regular neural network.

One-to-many

A one-to-many RNN allows a single input to be mapped to multiple outputs. Such a network receives an input of a fixed length but produces a sequence of outputs. Wide applications of such networks are found in music generation and image captioning.

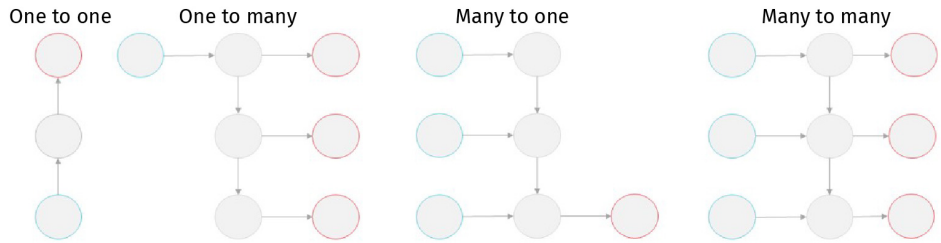
Many-to-one

Many-to-one refers to those types of recurrent neural networks where a series of inputs generate only one single output. A typical use of this type of network is sentiment analysis where pieces of text are analyzed and then classified to represent a sentiment that categorizes the opinions expressed in the text.

Many-to-many

A many-to-many recurrent neural network uses a series of input data and generates a series of outputs. The number of inputs and outputs can be the same size, leading to a subcategory commonly known as equal unit size; if the input and output sequences have different sizes, this leads to unequal unit size RNNs. The former finds applications in name-entity recognition problems (i.e., seeking to locate and classify named entities found in unstructured texts), while the latter is typically used in automatic machine translation. The figure below provides an illustration of the above cases.

Figure 37: Types of Recurrent Neural Networks



Source: Evis Plaku (2023).

Implementing a Simple RNN in Keras

The Keras library provides built-in functionalities for implementing a simple recurrent neural network. To illustrate the concept, we will build a simple RNN to address the Internet Movie Database (IMDb) movie review classification problem. Note that the dataset presented by Maas et al. (2011) contains IMDb movie reviews that are labeled according to the sentiment they represent (i.e., positive or negative).

Sentiment analysis is an important and challenging problem. In our context, note that the movie reviews are stored as a sequence of integers. These integers represent identification numbers that have been pre-assigned to individual words, while target labels are denoted by either zero (negative sentiment) or one (positive sentiment). This representation of pure textual information into a sequence of data makes the problem suitable to be addressed by a recurrent neural network.

As shown in Chollet (2017), before we can start building a recurrent neural network, we need to preprocess the data to determine the number of features to be used, their length, the batch size used for training, and other relevant aspects for the training phase. The data is processed as follows:

```
from keras.datasets import imdb
from keras.preprocessing import sequence
from keras.utils import pad_sequences
max_feat   = 10000
max_len    = 500
batch_size = 32

# loading the data
(X_train, y_train), (X_test, y_test) =
          imdb.load_data(num_words=max_features)

# shape of input train and test data
X_train = pad_sequences(X_train, maxlen=max_len)
X_test = pad_sequences(X_test, maxlen=max_len)

print('shape of input training data:', X_train.shape)
print(shape of input test data:', X_test.shape)
```

Note that we start by importing the necessary libraries to load the dataset to process it. Then, we determine the number of words to consider as features to 10,000. The maximum length of 500 denotes that after this many words we will cut off the text and not consider the rest of it. In addition, we determine a batch size of 32. After we load the dataset, we determine that the dimensions of the data used for training and for testing, both having a shape of 25,000 x 500.

Training the model with embedding and SimpleRNN layers

After loading the dataset and preprocessing the data, it is time to train a model using an embedding and a simple RNN layer. The role of the embedding layer is to transform each word into a fixed length vector of defined size. In order to deal with textual data, our recurrent neural network need to have the data transformed into numbers before the learning phase can start. Next, we use a simple RNN layer so that our network can identify patterns in the data and learn how to map input features to the target output. In addition, we also use a fully connected dense layer with the sigmoid function whose objective is to convert the output of the model in a probability score. This can be achieved using the code below.

```
from keras.layers import Dense

network = Sequential()
network.add(Embedding(max_feat, 32))
network.add(SimpleRNN(32))
network.add(Dense(1, activation='sigmoid'))
```

```
network.compile(optimizer='rmsprop', metrics=['acc'],
                loss='binary_crossentropy',)
history = model.fit(X_train, y_train, epochs=10,
                    batch_size=128, validation_split=0.2)
```

As one can observe, the model is trained using the root mean squared propagation, which is a variant extension of the gradient descent algorithm. Our chosen metric is the accuracy of prediction. The model is trained for ten epochs with a batch size of 128 and using a split of 20 percent for testing data.

Displaying the results

To get a better understanding of the results, we can plot the accuracy of the model with the training and the testing data and, similarly, we can also show the training and validation loss. To do so, we take advantage of the "matplotlib" library. We keep track of the history (steps and results) of the training procedure for the training and validation accuracy and loss. Then, as shown in Chollet (2017), the data is plotted against the number of epochs used:

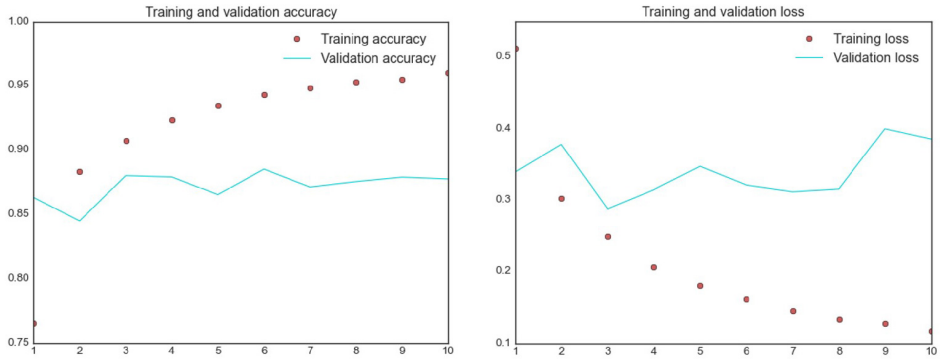
```
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)
plt.plot(epochs, acc, 'o', label='Training accuracy')
plt.plot(epochs, val_acc, label='Validation accuracy')
plt.title('Training and validation accuracy')
plt.legend()

plt.plot(epochs, loss, 'o', label='Training loss')
plt.plot(epochs, val_loss, label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
```

Figure 38: Training and Validation Accuracy and Loss



Source: Evis Plaku (2023).

The figure above demonstrates the results. Note that this simple recurrent neural network achieves a validation accuracy of approximately 85 percent, while, as expected, it performs better on the training data. Note that we only considered the first 500 words rather than the full sequence, which in some cases might be longer, and this might be an issue that could affect the performance of our model. Another, more fundamental issue is the fact that RNNs in their simple basic form suffer when processing long sequences, such as text. As we will discuss in the next sections, there are more advanced types of layers that generally yield a better performance.

Two Issues of Standard RNNs

Recurrent neural networks must deal with two major obstacles that influence the effectiveness of the learning process in an RNN. Both issues are related to "gradient" problems. Recall that the gradient is the derivative of a function. In simpler terms, the gradient measures how much the output of a function changed if the input changes just by a little bit. In our context, the gradient estimates the change in all weight parameters with regard to the change in the error. The higher the gradient, the steeper the slope of the function and therefore the faster a model can learn. However, a slope equal or close to zero would mean that the model is not learning.

Since recurrent neural networks are equipped with memory units, they establish a connection between the target output and input events that have occurred on earlier timesteps. However, in practice, it is difficult to correctly assign importance to far remote inputs. This leads to two important issues as discussined in the following paragraphs.

Exploding gradients

Exploding gradients occur when large error gradients accumulate, resulting in unreasonably large updates to the weights of the network

Exploding gradientsrefer to the problem where the learning algorithm assigns unreasonably high importance to weights. Large error gradients accumulate, and this leads to major updates of network weights during the training phase. It causes the network to become unstable and ultimately unable to learn well from the training data.

In practice, exploding gradients often occur by repeatedly multiplying gradient values larger than one causing an exponential growth that results in overflow, similar to the effect of the proverbial butterfly whose flapping wings cause a distant hurricane. Usually when a model suffers from exploding gradient it might show the following signs:

- The model is unstable, meaning that major changes occur from one update to another.
- The model is unable to learn meaningful patterns out of the training data.
- The model weights become unreasonably large during training phase and/or the error gradient values are regularly larger than 1.0 for most nodes.

When these signs occur, one needs to confirm the issue of exploding gradients. To address the challenge, gradients are often truncated or compressed. Typical measures include redesigning the network to have fewer layers or to use a smaller batch size during the training phase. A common approach in recurrent neural networks is to allow updating in more recent time-steps, otherwise known as truncating through time, which will be discussed in more detail in the upcoming sections. Limiting the size of the gradients during the training phase (commonly known as clipping) is another common technique that is used to deal with the exploding gradients problem.

Vanishing gradients

The learning process in a recurrent neural network requires propagating a feedback signal from the output back to earlier layers. Sending messages in the backward direction empowers the network to learn from its errors and adjust its weight parameters accordingly. However, since the recurrent neural network can be a deep network comprised of many layers, it may occur that the signal becomes weak or even gets entirely lost as it travels back. This issue is known as the **vanishing gradient** problem and might cause the network to become untrainable. Vanishing gradients can become too small (i.e., close to zero) and, therefore, it becomes difficult, if not impossible, for the network to learn meaningful patterns out of the input data.

Vanishing gradient
the problem of the recurrent neural network being unable to propagate useful gradient information from the output of the model back to the layers near the input of the model

The vanishing gradient problem is an important barrier for recurrent neural networks. To address this issue, weight initialization is proposed as a technique that algorithmically creates initial values that prevent the backpropagation algorithm from updating weights to unrealistically small values. Though many techniques have been tried out, a special type of recurrent neural networks, known as long short-term memory networks, have emerged as an effective and robust approach that deals with the vanishing gradient problem in a principled manner. This type of network is discussed in detail in the next section.

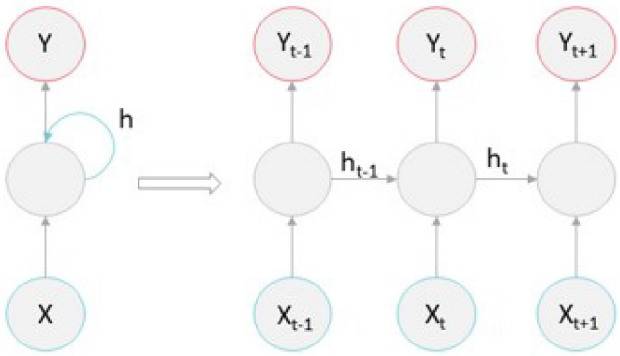
5.2 Memory Cells

Consider an idiom that comes unexpectedly in this text, such as “out of the blue.” For the idiom to make sense, one needs to not only understand the individual words, but also how they are aligned. A recurrent neural network, in this simple analogy, needs to consider a sequence of words and the position of each word before assembling all that information to make a prediction about the next word in the sequence.

A recurrent neural network has a form of memory. More formally, at a given time step t, the output of a recurrent neuron is dependent on inputs from previous timesteps. Because the recurrent neuron preserves information in a state across different timesteps, this state is often called a memory cell. Various cell types have been developed, ranging from basic cells to more advanced ones, as we will discuss in more detail in the following sections.

Let us denote a cell at a given timestep t as h_t where "h" stands for hidden. The state of a cell is a function of the state at a previous timestep, and the inputs received at the current timestep, as in $h_t = f(h_{t-1},\ X_t)$ where X_t denotes the inputs at timestep t and h_{t-1} represents the hidden state at the previous timestep. Let us denote the output at timestep t as Y_t. The figure below provides an illustration. In the context of our earlier example, the left side of the figure represents the network, which has built a model that is able to predict the entire phrase. The right side of the figure, however, represents how the expression is unrolled through several time-steps, each mapping (for instance) a single word.

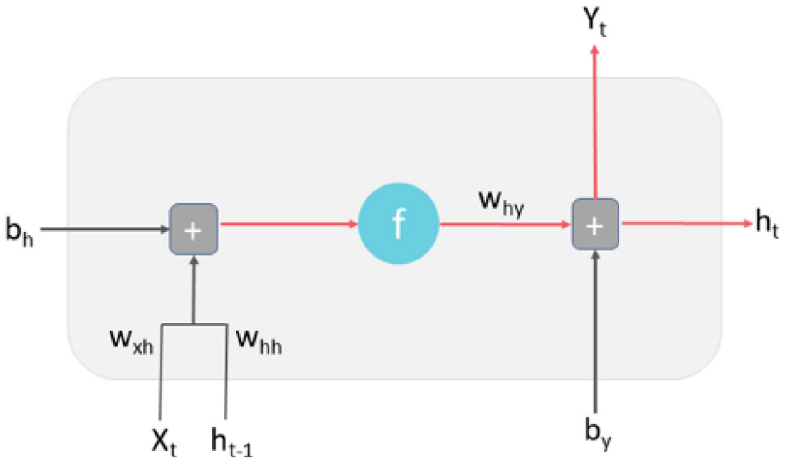
Figure 39: A Memory Cell with Hidden States



Source: Evis Plaku (2023).

The structure of a simple cell in the hidden layer of a recurrent neural network is exemplified by the following figure. Note that w is used to represent the weight coefficient matrix. For example, w_{xh} represents the weight coefficient matrix of the input layer to the hidden layer. The bias vector is denoted by b, as, for instance, b_h represents the bias vector of the hidden layer. Finally, the activation function used by the memory cell is denoted by f.

Figure 40: Simple RNN Cell Structure in the Hidden Layer



Source: Evis Plaku (2023).

As one can observe the resulting state h_t at timestep t is dependent on the state h_{t-1} at a previous timestep and the current inputs X_t are adjusted accordingly by the weight parameters and the bias term, which after applying the activation function, yields the output Y_t.

5.3 LSTMs

Basic recurrent neural networks are simplistic models that are rarely used in practice in their pure form. Although they were built on the premise of being able to retain information obtained many timesteps before, in practice many barriers prevent RNNs from learning these long-term dependencies. The vanishing gradient problem discussed earlier poses an important challenge. Moreover, the deeper the network, the more impenetrable it becomes for information to flow back smoothly and for the network to remember what happened in earlier remote steps.

To address this issue, German researchers Hochreiter and Schmidhuber (1997) proposed a robust solution by developing a variant of a recurrent neural network known as **long-short-term memory units (LSTM)**

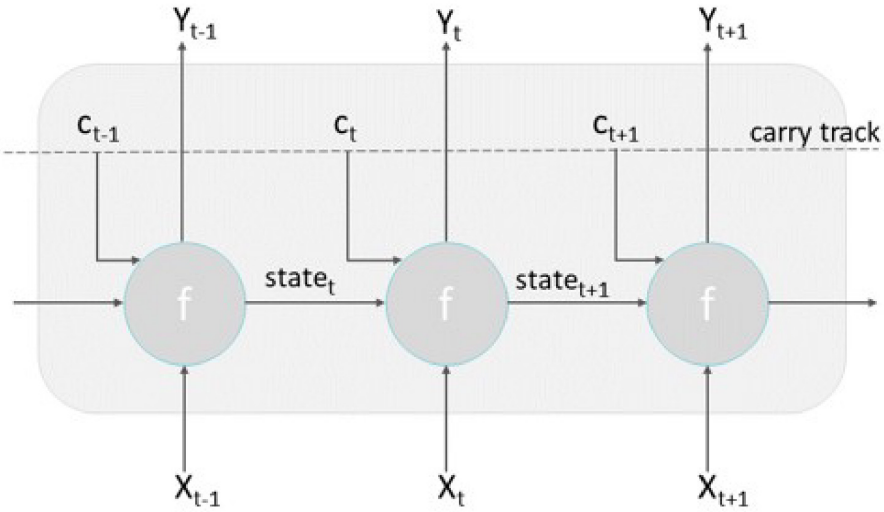
LSTM
A long short-term memory unit is a special type of recurrent neural network capable of successfully handling long-term dependencies

. LSTMs are a type of recurrent neural networks able to handle long-term dependencies. They preserve relevant information from earlier sequences and carry it forward in the network. These types of networks can even learn from events that have a significant time lag between them.

The image of a conveyor belt is often used to describe an LSTM. Information from a sequence that is being processed by the network can jump on the conveyor belt at any given point, and it is then transported to later timesteps before jumping off, all intact, and is used whenever it is needed by the network. An LSTM is able to save information, thus preventing it from gradually getting lost in the deep layers of the network.

Consider a simple LSTM cell, as shown in the figure below. Note that W_{out} and U_{out} denote output weight matrices. An LSTM carries information across various timesteps. In our illustration, we denote c_t by the information that is carried at timestep t. A cell will combine the input connection with the recurrent connection and the information it has carried, thus affecting the state that is being transmitted to the next timestep. The objective is to yield the next state and the output.

Figure 41: A LSTM With a Carry Track



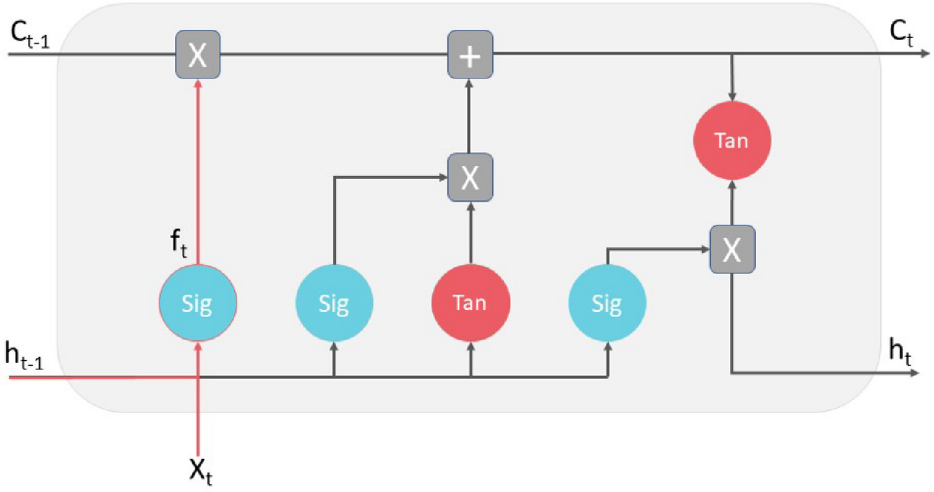
Source: Evis Plaku (2023) based on Chollet (2017).

The particularity of a long short-term memory unit is the way that the next state of the carried information is computed. The LSTM can add or remove information from the cell state based on regulations imposed by some special structures, known as gates. The role of gates is to decide if information will pass through or not. To achieve that, they typically use a sigmoid neural net layer. Recall that a sigmoid layer uses a sigmoid function that receives an input and outputs a result ranging from zero (let nothing pass) to one (let everything pass). Three distinct transformations are involved that are described by three types of gates that control the flow of information in the cell state.

The forget gate

An LSTM has to decide which information is relevant to keep from the prior cell state. The forget gate is charged with this task. The figure below provides an illustration. The arrows represent the information flow in the forget gate, while the rest of the figure demonstrates the other composing parts of the LSTM cell, whose discussion follows this section.

Figure 42: The Forget Gate



Source: Evis Plaku (2023).

To make a decision, the forget gate considers the input at a given timestep X_t and the hidden state h_{t-1} and applies on them the sigmoid function, which generates a result between zero and 1. Values of zero denote information that is discarded, while values of one shows information that is remembered. Values in between represent information that is partially remembered. The value of the forget gate, denoted by f_t will be used by the cell on future steps.

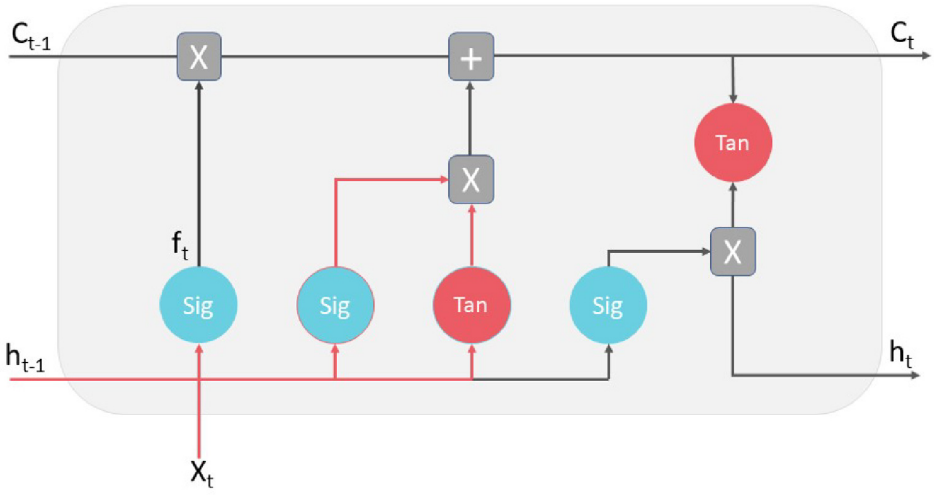
Consider as an example a system that aims to perform an automatic prediction on the next word that should come up in a sentence. If, for instance, the cell state contains information about the gender of a subject so that it can use the correct pronouns, when a new subject is encountered in the sentence, it might be a good decision to forget the gender of the previous subject.

The input gate and update state

The role of the input gate is to identify the elements that need to be added to the cell state and the long-term memory of the network. The input gate layer decides which values will be updated. To achieve that, a sigmoid function is applied on the current state X_t and the hidden state h_{t-1}, transforming the values in the range zero (not relevant) to one (relevant). In order to better regulate the network, the tangent function is also applied yielding

an output between -1 and 1. These outputs are multiplied together in a process in which the input gate is charged with the task of deciding what information is relevant to update in the current cell state of the LSTM unit. The figure below provides an illustration.

Figure 43: The Input and Update State



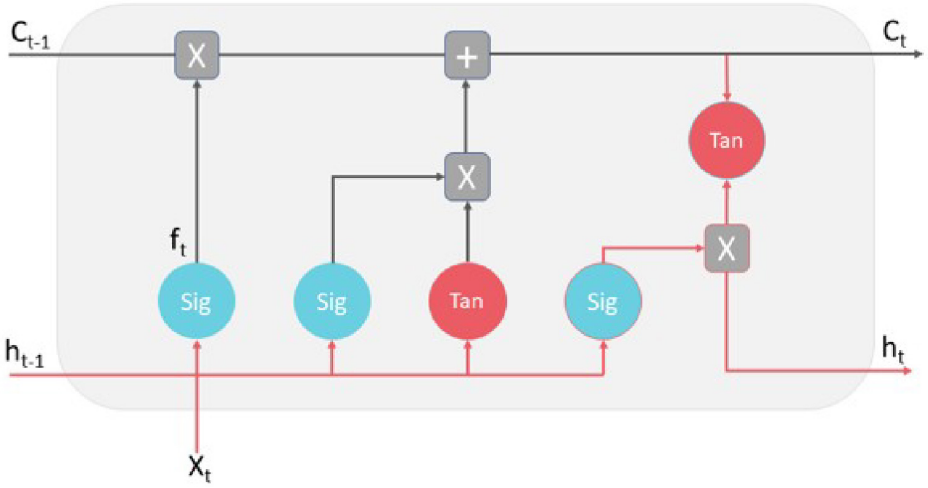
Source: Evis Plaku (2023).

At this point, the network has enough information to calculate the cell state and it is ready to store the information in it. When this transformation takes place, it allows to transition from the old cell C_{t-1} to the new oneC_t. The previous cell C_{t-1} is multiplied with the forget vector f_t and then the network performs pointwise addition with the input vector. This allows it to update the cell state to the values that are found relevant. Note that during this step, the old state does not carry out the information that we decided to forget. In the context of our example, we would discard the information about the gender of the previous subject and replace it with the gender of the new one.

The output gate

The objective of the output gate is to decide what to output from the memory cell. It contains information on previous inputs and decides the value of the next hidden state. The operations performed on the output gate start with a sigmoid function being applied to the hidden state and the current input. Then, the modified state is passed to the tangent function. The result is multiplied with the Sigmoid function, which decides what parts of the cell state to output. The figure below provides an illustration of the above operations, denoted by the red arrows.

Figure 44: The Output Gate



Source: Evis Plaku (2023).

We provide a filtered version of the cell state containing only the parts that we decided to keep. The new cell and the new hidden state carried out in the next steps of the process.

Implementing the LSTM Layer in Keras

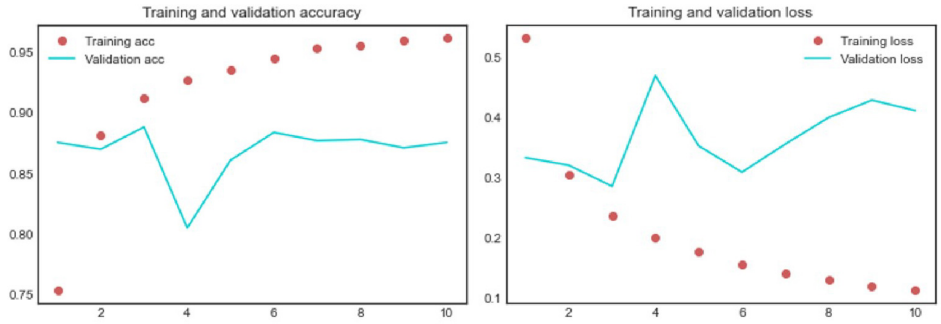
Earlier we showed how we can implement a simple RNN layer in Keras. Similarly, we can implement a more advanced layer, such as an LSTM, to tackle the same problem, namely the IMDB movie review classification. To do so, we update the code used previously to denote the fact that we are now using an LSTM layer. The other aspects of the model remain the same:

```
from keras.layers import LSTM

network = Sequential()
network.add(Embedding(max_feat, 32))
network.add(LSTM(32))
network.add(Dense(1, activation='sigmoid'))
```

Plotting the results as we did in the previous section, would show that, this time, we achieve an accuracy of 89 percent which is a better result. In contrast with a simple RNN cell, LSTMs suffer less from vanishing gradient problems.

Figure 45: LSTM Training and Validation Accuracy and Loss



Source: Evis Plaku (2023).

Gated Recurrent Unit Networks

Gated Recurrent Units
A GRU is an advancement of a standard RNN that leverages connections through a sequence of nodes to address the vanishing gradient problem and effectively perform machine learning tasks.

Gated recurrent units (GRU) are a type of recurrent neural network developed by Cho et.al (2014) to address the vanishing gradient problem. Similar to long short-term memory units, GRUs rely on gates to control information flow. However, they have a simpler architecture, use less memory, and typically require less time to train. GRUs do not have a cell state and they use the hidden state to pass on information. They are composed of two gate operating mechanisms commonly referred to as the update gate and the reset gate. Their key objective is to retain information from earlier timesteps and use that to decide what information should be passed to the output.

Update gate

The role of the update gate is to decide the amount of previous information that will be passed into the next state. More formally, at a given timestep t, the update gate z_t is calculated as

$$z_t = \sigma\Big(W^{(z)} \cdot x_t + U^z \cdot h_{t-1}\Big)$$

Note that the input x_t at timestep t is multiplied by its weight parameters $W^{(z)}$. Similarly, the information from the previous timesteps stored in h_{t-1} is multiplied by the respective weight parameters U^z. These results are added together, multiplied by a coefficient σ, and then a sigmoid activation function is applied to output results that fall between zero and 1. This way the update gate controls how much information from earlier timesteps should be passed on in future calculations of the model.

Reset gate

The purpose of the reset gate is to help the model the decide how much of the information from earlier timesteps it should forget. The formula for the reset gate is the same as the update gate:

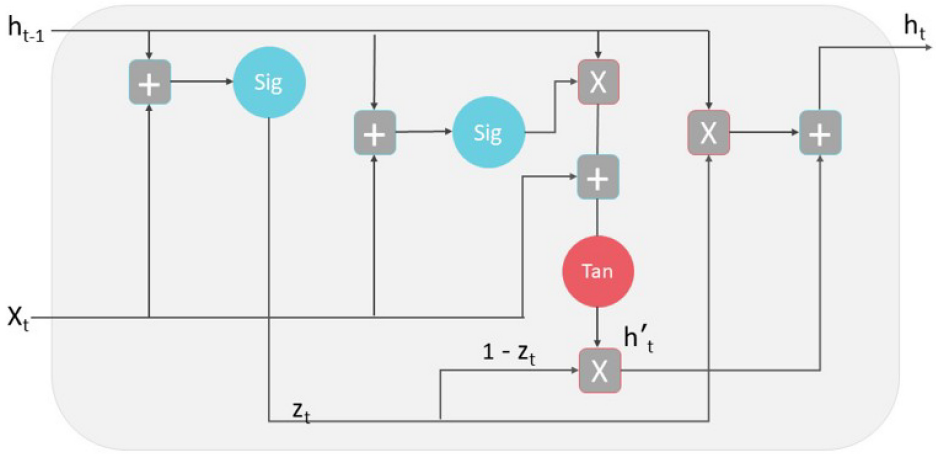
$$z_t = \sigma\Big(W^{(z)} \cdot x_t + U^z \cdot h_{t-1}\Big)$$

However, the key differences rely on the weights and the use of gates. The operations performed at this gate include multiplying the input x_t and the hidden state h_{t-1} with their respective weights, adding the results and applying the sigmoid function to squash the output.

Gates in action

The figure below provides an illustration of the update and reset gate operation in a GRU network.

Figure 46: Gated Recurrent Unit



Source: Evis Plaku (2023).

First, the reset gate determines which information from earlier timesteps will be stored in the new memory. Input vector x_t and hidden state h_{t-1} are multiplied with their respective weights and an element-wise multiplication is performed between the previous hidden state and the reset gate. The results are summed and a non-linear activation function is applied yielding the current memory content h'_t.

At the current timestep t, the network must calculate the hidden state h_t. To achieve that, it will heavily rely on the role of the update gate, because it holds information for the current unit and will transport it forward in the network. The update gate will decide which information to use from current memory content h_t and earlier timesteps h_{t-1}. In particular, the update gate and h_{t-1} perform element-wise multiplication and the result is summed with the product operation between the current memory content h'_t and $1 - z_t$, where z_t is the update gate at timestep t.

Let's illustrate these operations with an example. Assume the goal of our network is to consider movie reviews and determine the sentiment (i.e., positive or negative) associated with them. Consider this review: "*Home Alone* is a splendid movie. Leaving a child alone can be alarming. The movie evokes all sorts of scary nostalgia." As we can notice, the most relevant information is of the review is positioned in the beginning of the text. Without the ability to store past information, the network could be led to make wrong predictions. However, because of the architecture described, the model can learn to set the vector z_t close to one (hence, $1 - z_t$ will be close to 0) and therefore ignore a large portion of more recent text which explains the movie plot and is irrelevant for the prediction.

To summarize, GRUs allow the network model to store and filter information relying on the update and reset gates. That helps the network to learn because relevant information is kept through timesteps and does not vanish as new information comes in.

In practice, both LSTMs and GRUs are used widely. GRUs typically need less time to train and use fewer parameters, but LSTMs tend to perform better on larger datasets. Both types of recurrent neural networks are state-of-the-art approaches that can perform well in many complex problems.

5.4 Training RNNs: Unrolling Through Time

Backpropagation through time
BPTT is a gradient-based technique used for training certain types of recurrent neural networks, such as long short-term memory networks.

A common strategy used to train recurrent neural networks is **backpropagation through time** (BPTT). The key elements of this approach include unrolling the RNN (as we did earlier) and then applying the backpropagation algorithm.

Backpropagation is a standard algorithm in machine learning, which in our context is used to calculate the gradient of the error function with respect to the weights of the network. A neural network uses forward propagation to generate an output of the model. An error term measures if this output is correct or not. Backpropagation allows to move backwards through the neural network to compute the partial derivatives of the error. A gradient descent algorithm adjusts the weights of the network accordingly with the objective of minimizing an error function, thus enabling the neural network to learn during the training phase.

In our case, backpropagation through time is the application of the backpropagation algorithm to a recurrent neural network. Conceptually, we need to unroll all input timesteps to apply BPTT. At each timestep, the algorithm will consider one input and one output timestep and a copy of the recurrent neural network. At each timestep, the errors will be calculated, then accumulated together. In particular, if the training phase of the network starts at time t_0 and ends at time t_n then the total cost function is calculated as the sum over time of the standard error function E_t at each timestep t:

$$E_{total}(t_0, t_n) = \sum_{t=t_0}^{t_n} E(t)$$

When the network is rolled back up, the weights will be updated. Note that when the gradient descent considers the weight updates w_{ij} they will have contributions from each timestep:

$$\Delta w_{ij} = -\eta \frac{\delta\, E_{total}(t_o,\ t_n)}{w_{ij}} = -\ \eta \sum_{t=t_0}^{t_n} \frac{\delta E(t)}{\delta w_{ij}}$$

- The partial derivatives $\delta \frac{E}{w_{ij}}$ will be affected by multiple instances of each weight and will be dependent on the activations of input and hidden units at previous timesteps.
- The following key steps of the backpropagation through the time algorithm are repeated: Consider a sequence of input and output timesteps.
- Unroll the recurrent neural network and at each timestep calculate and accumulate the errors.
- Roll the network back up and update the weights accordingly.

Two main phases, commonly known as forward pass and backward pass, compose the backpropagation through the time algorithm.

Forward Pass

During the forward pass, the input vector x_t and hidden state from the previous timestep h_{t-1} are multiplied by the respective weight matrices, say $W^{(xh)}$ and $W^{(hh)}$, and then are summed together. A non-linear function, say $\tanh$, receives the result and generates the input for the next timestep. It processes the information for performing classification, outputting the result y_t. More formally, the equation can be written as

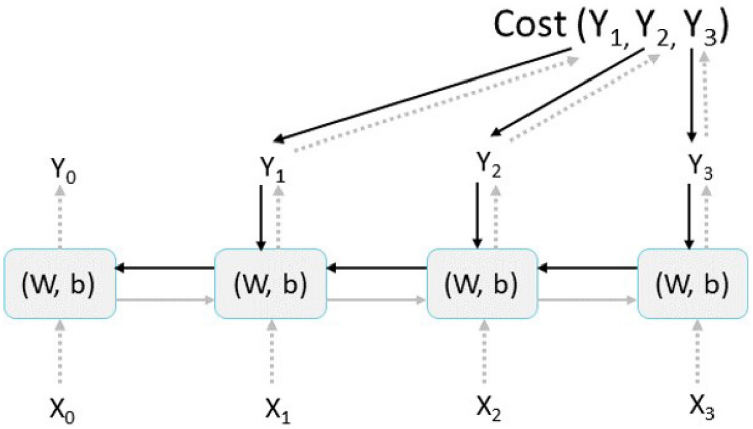
$$h_t = \tanh\Big(W^{(xh)} \cdot x_t + \ W^{(hh)} \cdot h_{t-1}\Big) \quad \text{and} \quad y_t = W^{(hy)} \cdot h_t$$

Backward Pass

The objective of the backward pass is to start from the end and move backwards to compute the gradient of the classification loss with respect to the weight matrix and the hidden state. Note that sequence data are processed one step at a time, while during the backward pass of the backpropagation algorithm the gradients move across timesteps. Hence, the name backpropagation through time.

The figure below provides an illustration. The $(W,\quad b)$ pair denotes the weights and biases. The dashed arrows represent the forward pass through the unrolled network. A cost function is used to evaluate the output sequence. The gradients of the cost function are propagated backwards through the unrolled network, as denoted in the figure by the solid arrows. The gradients computed during the BPTT algorithm are used to update the weights of the network. Note that the information about the gradients is transmitted backwards through all the outputs and not just the final output. In our example, it means that the cost function is computed by using three outputs of the network ($Y_1,\ Y_2, Y_3$), and therefore, the gradient will affect those, but not output Y_0.

Figure 47: Backpropagation Through Time



Source: Evis Plaku (2023).

Practical Considerations for the BPTT Algorithm

Due to their architectural characteristics, recurrent neural networks can become increasingly complex when unfolded. Keeping track of all components at multiple timesteps is burdensome. This is a problematic issue for most recurrent neural networks. Note that weights are updated at each timestep. Therefore, it requires a large storage to keep track of a complete history of changes of inputs and network states at earlier timesteps. In order to make the network computationally feasible, it might be helpful to ignore very early information, and, thus, to truncate the network after a certain number of timesteps.

The rationale behind this strategy is that the update of weights is affected less by contributions that come from timesteps further back in time, in comparison to more recent steps. This idea has led to an enhanced algorithm known as truncated backpropagation through time (TBPTT).

In TBPTT, the forward and backward passes are executed in chunks of sequences rather than the entire sequence. This helps reduce the computational complexity of the algorithm during the training phase. However, it might come with the cost of not being able to learn dependencies as long as the original BPTT algorithm because of the limit imposed by the truncation procedure.

In more detail, this modified version works by processing the sequence of inputs one timestep at a time and then every k_1 timestep, the BPTT update is performed back for a certain number of timesteps, for example, k_2. This periodic processing and backpropagation make the method a practical and effective way of training an RNN.

Note that if k_2 – the number of timesteps for which BPTT is performed – is sufficiently small, then the parameter update cost will not be computationally expensive. On the contrary, its hidden states have still been exposed to many timesteps and, therefore, contain useful information about earlier timesteps, which can be effectively used during the training procedure.

Observe that both parameters play an important role, because k_1 determines the number of forward-pass timesteps between updates. The frequency of weights updates affects how fast or slow the training phase will be. The k_2 parameter determines the number of steps for which to apply BPTT. Therefore, it should be large enough to be able to capture meaningful and relevant information in earlier timesteps, so that the network can effectively learn.

In comparison with BPTT, the following steps are repeated in the enhanced version of the truncated backpropagation through time algorithm:

1. Consider a sequence of of k_1 input and output timesteps.
2. Unroll the recurrent neural network and at each timestep calculate and accumulate the errors for k_2 timesteps.
3. Roll the network back up and update the weights accordingly.

The backpropagation through time algorithm and the enhanced version using truncation have proven to be effective approaches to train recurrent neural networks that address complex and challenging problems.

SUMMARY

In this unit, you learned that recurrent neural networks are a class of neural networks that rely on maintaining information from prior observations and using that for future predictions. We discussed how this distinctive characteristic makes RNNs especially powerful in many scenarios where the context plays an important role in predicting the outcome, such as in automatic translation, language modeling, text generation, speech recognition, and many others.

We also discussed the different types of recurrent neural networks and their composing parts. We presented the memory cells as core units of an RNN and identified how the issues of exploding gradients and vanishing gradients can deter and even prevent an RNN from learning effectively.

To address these challenges, the long short-term memory unit was then presented as a robust approach that addresses these issues and that empowers a network to learn effectively while remembering remote events and using that information to predict target outputs. In particu-

lar, we discussed the role of different gates such as forget, and the input and output gates, in transmitting information and deciding which are the most relevant parts that need to be propagated in the other steps.

Furthermore, we demonstrated how we can build a simple recurrent and LSTM layer using Keras. We discussed how a recurrent neural network can learn through the backpropagation through time algorithm. In addition, truncation was presented as an enhancement that provides computational convenience and leads to effective practical implementations of recurrent neural networks in many challenging domains.

BACKMATTER

LIST OF REFERENCES

Amer, M. (2022). *A visual introduction to deep learning* [Image]. kDimensions.

BruceBlaus (2013, September 30). *Multipolar Neuron* [Image]. Wikipedia Commons. CC BY 3.0. https://en.wikipedia.org/wiki/Multipolar_neuron

Cho, K., van Merrienboer, B., Bahdanau, D., & Bengio, Y. (2014). *On the properties of neural machine translation: Encoder-decoder approaches*. Proceedings of the Eighth Workshop on Syntax: Semantics and Structure in Statistical Translation, 103–111. https://aclanthology.org/W14-4012.pdf

Chollet, F. (2017). *Deep learning with Python*. Manning Publications.

Deng, M. (2012). The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, *29*(6), 141–142.

Géron, A. (2019). *Hands-on machine learning with Scikit-Learn, Keras and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly.

He, K., Zhang, X., Ren, S., & Sun, J. (2016). *Deep residual learning for image recognition* [Image]. IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 770–778. https://ieeexplore.ieee.org/document/7780459

Hochreiter, S., & Schmidhuber, Jürgen. (1997). Long short-term memory. *Neural Computation*, *9*(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Hubel, D. H., & Wiesel, T. N. (1959). Single unit activity in striate cortex of unrestrained cats. *Journal of Physiology, 147*(2), 226–238. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1357023/

Hubel, D. H., & Wiesel, T. N. (1968). Receptive fields and functional architecture of monkey striate cortex. *Journal of Physiology*, *195,* 215–243. http://10.1113/jphysiol.1968.sp008455

Ioffe, S., & Szegedy, C. (2015). *Batch normalization: Accelerating deep network training by reducing internal covariate shift*. https://arxiv.org/abs/1502.03167

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 1097–1105. https://papers.nips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). G*radient-based learning applied to document recognition*. Proceedings of the IEEE, 2278–2324. https://ieeexplore.ieee.org/document/726791

Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A., & Potts, C. (2011). L*earning word vectors for sentiment analysis*. ACL. https://aclanthology.org/P11-1015/

Mcculloch, W., & Pitts, W. (1943). A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, *5*, 127–147. https://www.cs.cmu.edu/~./epxing/Class/10715/reading/McCulloch.and.Pitts.pdf

Nielsen, M. A. (2018). *Neural networks and deep learning [misc].* Determination Press.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for the information storage and organization in the brain. *Psychological Review, 65*(6), 386–408. https://doi.org/10.1037/H0042519

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning internal representations by error propagation. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, 1,* 318–362. https://doi.org/10.1016/B978-1-4832-1446-7.50035-2

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., & Fei-Fei, L. (2014). *ImageNet large scale visual recognition challenge*. Springer. https://doi.org/10.1007/s11263-015-0816-y

Sammut, C. (2010). *Bias-variance trade-offs.* Encyclopedia of Machine Learning. Springer. pp. 110–110.

Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research 15(56)*, 1929–1958. https://jmlr.org/papers/v15/srivastava14a.html

Srivastava (2019). Dropout in a neural network [Image].

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. E., Anguelov, D., Erhan, D., Vanhoucke, V., & Rabinovich, A. (2015). *Going deeper with convolutions.* 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 1–9. https://ieeexplore.ieee.org/document/7298594

Xiao, H., Rasul, K., & Vollgraf, R. (2017). *Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms*. arXiv https://arxiv.org/abs/1708.07747v2

Yann LeCun and Corinaa Cortes (n.d.) *MNIST dataset sample digits* [Image] CC 3.0.

Zou, H., Hastie, T. (2005). Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society, 67(2)*, 301–320. https://www.jstor.org/stable/3647580

Zalando, (2017). *Samples from fashion MNIST dataset* [Image]. The MIT License.

LIST OF TABLES AND FIGURES

IU Internationale Hochschule GmbH
IU International University of Applied Sciences
Juri-Gagarin-Ring 152
D-99084 Erfurt

Mailing Address
Albert-Proeller-Straße 15-19
D-86675 Buchdorf

media@iu.org
www.iu.org

Help & Contacts (FAQ)
On myCampus you can always find answers to questions concerning your studies.