BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

CLASSIFICATION WITH NEURAL NETWORKS IN THE KERAS ENVIRONMENT

KLASIFIKACE POMOCÍ NEURONOVÝCH SÍTÍ V PROSTŘEDÍ KERAS

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR — MICHAL PYŠÍK
AUTOR PRÁCE

SUPERVISOR — Ing. VLADIMÍR BARTÍK, Ph.D.
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

145041

Institut:	Department of Information Systems (UIFS)
Student:	**Pyšík Michal**
Programme:	Information Technology
Specialization:	Information Technology
Title:	**Classification with Use of Neural Networks in the Keras Environment**
Category:	Data Mining
Academic year:	2022/23

Assignment:

1. Get acquainted with the topic of classification, especially using various types of neural networks.
2. Study the possibilities of the Keras environment in detail, especially with regard to different types and topologies of neural networks.
3. In agreement with the supervisor, select suitable datasets and classification tasks for the purpose of experiments with this library.
4. Implement an experimental application and perform experiments showing the properties of individual neural networks.
5. Evaluate the achieved results and other possibilities for extending this project.

Literature:

- Han, J., Kamber, M.: Data Mining: Concepts and Techniques. Third Edition. Morgan Kaufmann Publishers, 2012, 703 p., ISBN 978-0-12-381479-1.
- Chollet, F.: Keras: the Python Deep Learning API [online]. 2021 [cit. 2021-10-01]. Available at: [https://keras.io/]

Requirements for the semestral defence:
No requirements.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Bartík Vladimír, Ing., Ph.D.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	10.5.2023
Approval date:	24.10.2022

Abstract

This thesis examines the problematics of classification using artificial neural networks within use of the Keras framework, a high-level deep learning API for the Python programming language. The aim of the thesis is to discover the diverse options Keras has to offer in the field of classification, and to compare different types and topologies of artificial neural networks in the form of experiments on selected datasets, complemented by a simple experimental application whose main purpose is to provide an interface for these experiments.

Abstrakt

Tato práce zkoumá problematiku klasifikace pomocí umělých neuronových sítí s využitím knihovny Keras, poskytující vysokoúrovňové rozhraní pro práci s umělými neuronovými sítěmi v programovacím jazyce Python. Cílem práce je prozkoumat rozsáhlé možnosti této knihovny v oblasti klasifikace a porovnat různé typy a topologie umělých neuronových sítí formou experimentů na vybraných datasetech, což je doplněno jednoduchou experimentální aplikací sloužící především jako rozhraní pro tyto experimenty.

Keywords

Keras, Tensorflow, classification, artificial neural networks, types of neural networks, topologies of neural networks, comparison of neural networks, multilayer perceptrons, convolutional neural networks, recurrent neural networks, experiments

Klíčová slova

Keras, Tensorflow, klasifikace, umělé neuronové sítě, typy neuronových sítí, topologie neuronových sítí, porovnání neuronových sítí, vícevrstvé perceptrony, konvoluční neuronové sítě, rekurentní neuronové sítě, experimenty

Reference

PYŠÍK, Michal. *Classification with neural networks in the Keras environment.* Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Vladimír Bartík, Ph.D.

Rozšířený abstrakt

Klasifikace je jedním z nejběžnějších problémů v oblasti strojového učení. Cílem klasifikačního problému je přiřadit jednotlivá vstupní data do předem definovaných kategorií (tříd). To lze provést například použitím umělých neuronových sítí (dále jen neuronové sítě), které jsou inspirovány biologickými neuronovými sítěmi a jsou běžně využívány v mnoha doménách, včetně klasifikace. Vzhledem k tomu, že takové (nejen klasifikační) problémy můžou nabývat mnoha dosti odlišných podob, jako je například klasifikace obrázků či mluvené řeči, existuje více specializovaných typů těchto sítí. Pro práci s neuronovými sítěmi v dnešní době již není třeba znát podrobně jejich vnitřní mechanismy, ale lze využít jedno z vysokoúrovňových rozhraní pro práci s nimi, kam mimo jiné spadá také knihovna Keras.

Cílem této bakalářské práce je prozkoumat rozsáhlé možnosti prostředí Keras v oblasti klasifikace pomocí neuronových sítí a porovnat vlastnosti různých typů a topologií neuronových sítí formou experimentů na vybraných datasetech. Jelikož se různé typy neuronových sítí v mnoha ohledech výrazně liší, není jejich porovnání snadným problémem. Z těchto důvodů byly vybrány tři odlišné datasety, kdy každý dataset svou charakteristikou odpovídá jednomu ze tří hlavních druhů neuronových sítí (vícevrstvý perceptron, konvoluční neuronová síť, rekurentní neuronová síť). Pro každý ze tří experimentů byly individuálně vytvořeny čtyři neuronové sítě, a to nejen jedna od každého ze tří typů, ale dále ještě jedna odpovídající svým typem charakteristice daného experimentu, za účelem porovnání vícera topologií neuronových sítí. Jako rozhraní pro průběh experimentů slouží jednoduchá experimentální aplikace napsána v programovacím jazyce Python s využitím knihovny Keras, která zahrnuje mnoho funkcí umožňující důkladně monitorovat průběh trénování, a vyhodnocení (nejen) daných neuronových sítí.

Práce začíná úvodem do problematiky klasifikace představením klíčových pojmů, metrik pro měření výkonnosti klasifikačního modelu, a stručným popisem vybraných alternativních klasifikačních algoritmů. Poté jsou představeny umělé neuronové sítě, a to způsobem vhodným i pro čtenáře neznalého v dané problematice. Tato část je postupně rozvinuta od nejjednodušších stavebních bloků (neuronů) až po rozdíly mezi jednotlivými typy neuronových sítí. Dále následuje kapitola věnující se samotné knihovně Keras, která je strukturována chronologicky v souladu s vytvářením vlastních klasifikačních modelů. U každého kroku jsou popsány odlišné přístupy a možnosti, které zde Keras nabízí.

Samotné experimenty začínají jejich přípravou, včetně analýzy jednotlivých datasetů, způsobů interpretace odlišných druhů dat jednotlivými typy neuronových sítí, a výběrem architektur použitých sítí. Zde jsou také odůvodněny různá rozhodnutí, která jsou klíčová pro průběh experimentů a jejich přínos. Po stručném popisu návrhu a implementace experimentální aplikace následuje popis průběhu jednotlivých experimentů, především formou popisu grafů vygenerovaných pomocí experimentání aplikace, a jejich vyhodnocení formou jednotlivých závěrů a konečným shrnutím. Ačkoli by nebylo vhodné dělat příliš vážné závěry čistě na základě relativní výkonnosti jednotlivých modelů, jelikož může být silně ovlivněna příliš mnoha faktory, byly demonstrovány jednotlivé vlastnosti odlišných typů neuronových sítí a jejich využitelnost v oblastech neobvyklých pro daný typ sítě. V rámci experimentů byly také nalezeny určité neočekávané poznatky. Po vyvození závěrů byly dále navrženy vhodné možnosti pro rozšíření této práce, především v kontextu moderních architektur neuronových sítí a aktuálních trendů v této oblasti.

Classification with neural networks in the Keras environment

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Ing. Vladimír Bartík, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Michal Pyšík
May 3, 2023

Acknowledgements

I would like to express my gratitude towards my supervisor, Ing. Vladimír Bartík, Ph.D., who was always willing to consult the state of the work with me and whose suggestions helped greatly towards reaching the thesis' final state. I would also like to thank my family and friends for their continuous support throughout my studies.

Contents

Chapter 1

Introduction

In recent years, the field of artificial intelligence has seen a significant rise in popularity, thanks to advancements in technology and the availability of large datasets. One of the most popular and successful computational models in this field are artificial neural networks, inspired by the function of real biological brains. These models have been widely used in various applications, whether it be simple ones, like digit classification, or complex natural language processing applications, like AI assistants.

Classification is a fundamental task in machine learning, where the goal is to predict the class label of a given input, by classifying the input object into one of a finite number of classes based on its features. In this sense, an object is nothing more a set of numerical values, but they do not always represent its features directly. In the case of, for example, images or audio files, the features have to be extracted first. This is usually learned and done by the model itself, at least in the case of deep learning. Artificial neural networks have proven themselves to be especially helpful when it comes to complex non-linear tasks, due to their ability to handle complex relationships between variables and their ability to learn from large amounts of data. There are many different types of neural networks, each suitable for problems of a different nature.

To start solving problems using neural networks today, there's no longer a need for learning all the mathematical principles of their internal mechanism. Any of the high-level deep-learning APIs currently available, which include the Keras library, can be used instead. The topic of this thesis revolves around utilization of the Keras library in the field of classification using neural networks. The main goals are to introduce neural networks in a way that even a reader inexperienced in the problematics can understand, to explore Keras' distinct options in the context of classification using neural networks, and to compare different types and topologies of neural networks through thoughtful experiments.

The theoretical part of the thesis introduces the concepts of classification (chapter 2) and artificial neural networks (chapter 3) from the ground up. Chapter 4 then introduces the Keras library in the form of a chronological use case guide. Finally, the practical part consists of preparation of the experiments in chapter 5, brief description of the experimental application (chapter 6), that serves as an interface for the experiments, and ends with the course and evaluation of the experiments in chapter 7.

Chapter 2

Classification

This chapter is an introduction to the problematics of classification, one of the most common kinds of tasks in machine learning—a field of study concerned with algorithms that can learn to solve many different kinds of tasks by learning from data. The goal of a classification task is to assign discrete class labels to objects from the problem domain, for example differentiating between pictures of cats and dogs. Classification falls into the category of supervised learning, since the model learns to predict classes of objects by training on labeled datasets. Some other common types of machine learning problems include regression (supervised learning), which is about predicting a continuous values, and clustering (unsupervised learning), whose goal is to group instances into unlabeled categories based on their similarity.

Section 2.1 explains basic classification terminology and types, into which can be different classification tasks divided [38], followed by section 2.2, which introduces miscellaneous evaluation metrics for measuring a classification model's performance. Besides artificial neural networks, the primary subject of this thesis, explained later in chapter 3, I considered it appropriate to also mention a few simpler classification algorithms [31] in section 2.3, that can sometime be more effective in the cases of simpler problems.

2.1 Terminology and types of tasks

This short section explains some of the technical terms that are crucial to understand before reading further.

Basic terminology

- **Classifier** – An algorithm used to map objects from the input set of data to specific classes. Classifiers can range from simple algorithms all the way to deep neural networks consisting of many hidden layers.
- **Model** – A classification model learns to draw some conclusion from the input data given for training, which can be later utilized to predict class labels of new data. Simply said, the model consists of the selected classifier and the experience gained by training on labeled data.
- **Feature** – An individual measurable property of a phenomenon. Features need to be transformed into numerical representations before being fed to the classifier.

- **Instance** – A feature vector, an n-dimensional vector of numerical values that represent some object. It is referred to as a sample in the context of a dataset. Some examples can be a single row in a table of data, or a multi-dimensional vector, where the feature values might correspond to the pixels of an image.
- **Dataset** – A collection of instances (samples), that is a subset of a given feature space[1]. Datasets are used for training, validating and testing the model.
- **Model training** – The purpose of training is to build the best mathematical representation of the relationship between data features and the target labels. This is done by feeding a labeled training dataset to the model, which then adjusts itself to make more accurate predictions, by trying to minimize the value of a loss function, which computes the distance between the expected output and the model's prediction.
- **Model evaluation** – Ways to evaluate the performance of the model. A quantifiable measure is called a metric.

Types of tasks

- **Binary classification** – Classification tasks that have only two class labels. Examples include predicting whether an email is a spam based on the most occurring words, or distinguishing males from females based on their favorite hobbies.
- **Multi-class classification** – Tasks with more than two class labels. Each instance is assigned only one label. A good example may be recognizing the species of the animal in a photo of a single animal.
- **Multi-label classification** – Each instance may be assigned more than one class label, including none or all of them. For example, classifying which kinds of fruit are contained in images of fruit bowls.
- **Balanced/imbalanced classification** – The distribution of classes in a training dataset may not always be uniform. A very good example is detection of a certain medical condition in patients. In these cases, other classification metrics become prioritized over plain accuracy, as false detection of the condition in a healthy patient usually leads to less damage than misclassifying a sick patient as healthy.

2.2 Evaluating a model's performance

There are many different evaluation metrics that can be used to measure a classification model's performance [16]. The simplest and also the most commonly used one is accuracy, which is calculated simply as the number of correct predictions divided by the sum of all predictions.

All classification metrics are based on the counts of true positives (TP) and true negatives (TN), corresponding to correct predictions, together with false positives (FP) and false negatives (FN), corresponding to incorrect predictions. A representation of the above parameters in a matrix format is called a confusion matrix (also known as an error matrix), an exemplary one is shown as table 2.1. In this context, we assume a binary classification problem with classes marked as *positive* and *negative* (those can for example represent the

[1]Feature space is the set of all possible values for a chosen set of features from given data.

presence of some disease in a patient), but some of these metrics can also be extended for use in multi-class and multi-label classification in a few possible ways.

		Predicted	
		P	**N**
Actual	**P**	TP = 302	FN = 15
	N	FP = 23	TN = 277

Table 2.1: A confusion matrix of binary classification.

Some of these metrics, apart from accuracy (2.1), are precision (2.2), which is the ability to identify only the relevant data points, recall (2.3), referred to as sensitivity in the statistics domain, which is the ability to find all relevant cases within a dataset, and specificity (2.4), which is based on the same principle as sensitivity, but in terms of actual negatives.

$$accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{2.1}$$

$$precision = \frac{TP}{TP + FP} \tag{2.2}$$

$$recall/sensitivity = \frac{TP}{TP + FN} \tag{2.3}$$

$$specificity = \frac{TN}{TN + FP} \tag{2.4}$$

Let's also take a look at some of the more advanced metrics. The F$_1$ score (2.5) is the harmonic mean of precision and recall. High F$_1$ score can be expected from models that can successfully classify *positive* cases, while not going overboard and marking too many *negative* cases as *positive*.

$$F_1 = \frac{2}{precision^{-1} + recall^{-1}} = \frac{2 \cdot precision \cdot recall}{precision + recall} \tag{2.5}$$

In cases where precision and recall are not given equal importance, either weighted-F$_1$ score or plotting a PR or ROG curve may be of use. A PR (precision-recall) curve is a curve that appears in a graph plotted such that the x-axis represents recall and the y-axis represents precision, showing the tradeoff between precision and recall at different thresholds. A ROC (receiver operating characteristic) curve is the same, but with x-axis representing the false positive rate (FPR; $FPR = 1 - specificity$) and the y-axis representing the true positive rate (TPR; $TPR = recall$), illustrating the diagnostic ability of a binary classifier as its discrimination threshold is varied. In both of these cases, we aim to maximize the area under the curve (AUC). Due to the absence of true negatives in its equation, a PR curve is considered more useful than a ROC curve when dealing with imbalanced classification with excess of the *negative* class [34], where correctly detecting *positive* cases is the main goal, such as a virus test or cancer detection. A ROC curve might be a better choice in cases where both classes are given the same importance (but one of the classes is still the main point of interest), since ROC considers true negatives as well. Both a PR curve and a ROC curve are plotted in figure 2.1.

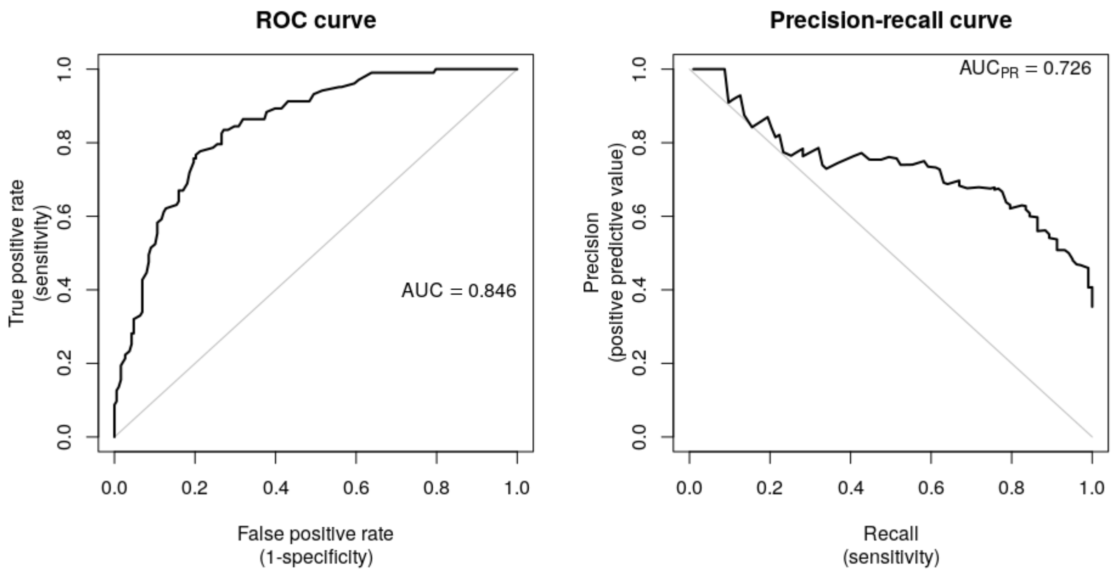


Figure 2.1: Example plots of a ROC curve and a PR curve [2].

2.3 Other classification algorithms

Before the introduction to artificial neural networks, this section briefly introduces a few other classification algorithms.

Logistic regression

Similar to how linear regression predicts continuous values by assuming a linear relationship between dependent and independent variables (useful for regression tasks), logistic regression divides elements into two groups based on a set probability threshold (often called decision boundary), which makes it a suitable technique for binary classification.

Instead of assuming that the data follow a linear function, they are modeled using a sigmoid function (Figure 2.2).The range of this function, the interval between 0 and 1, can then represent the probability of an instance belonging to a certain class. This probability then rounds to either 0 or 1 depending on whether it lies above or below the set threshold.

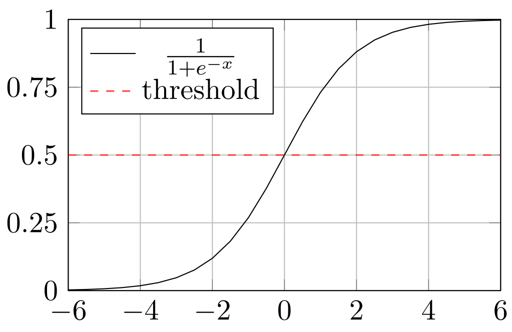


Figure 2.2: Graph of a sigmoid function and a threshold set to 0.5.

Naive Bayes classifiers

Naive Bayes classifiers are a family of probabilistic machine learning algorithms based on the Bayes theorem (2.6). They always assume that all the features of an instance being classified are independent of each other, and their contribution to the final outcome is equal. These assumptions are generally not present in the real world, hence the name “Naive”.

$$P(A|B) = \frac{P(B|A)\ P(A)}{P(B)} \tag{2.6}$$

At first, the dataset is converted into frequency tables, counting the numbers of occurrences of different feature values. From this table, a likehood table is generated by finding the propabilities of given features. After that, the Bayes theorem is used to calculate the posterior probabilities of the object belonging to a certain class based on its features. There are three types of Naive Bayes classifiers—Multinomial, Benoulli and Gaussian.

Decision trees

Decision trees model the classification or regression process in the form of a tree structure. Trees used for classification, with a single discrete class label, are called classification trees. They consist of decision nodes with two or more branches split according to a certain feature value and leaf nodes representing classes. An example can be seen in Figure 2.3.

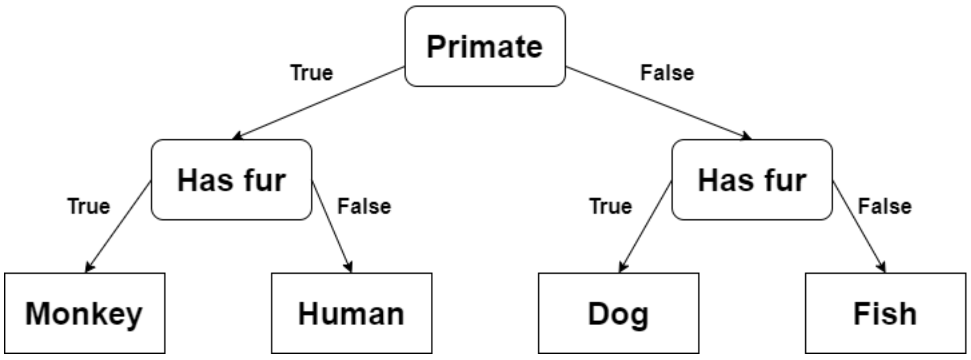


Figure 2.3: A very simple classification tree.

A decision tree is built using the ID3 algorithm [26], which utilizes entropy and information gain. Entropy is used to calculate the homogenity of a sample. The more equally divided a sample is, the higher its entropy. The information gain depends on the decrease in entropy after each splitting of the dataset on a feature. At each step, the goal is to find the feature that returns the highest information gain.

Random forest is an algorithm made out of multiple randomly created decision trees, where each tree-node works on a random subset of features and the final output is combined from outputs of the individual trees. As declared by François Chollet in [6, p. 16], “random forests are applicable to a wide rage of problems—you could say that they’re almost always the second-best algorithm for any shallow machine learning tasks”.

k-nearest neighbors algorithm

This algorithm classifies new cases based on a similarity measure to already labeled cases. The new instance inherits the label most common among its k-nearest neighbors measured by a distance function, where $k \geq 1$. Larger k value generally increases the precision by reducing the overall noise, but the optimal value for most datasets was found to be between 3 and 10. The number of dimensions of the space the labeled instances occupy is equal to the number of input features. Individual feature measures should also be normalized, so that their contribution to the outcome is ideally the same.

Support vector machines

A support vector machine (SVM) finds the hyperplane in an N-dimensional space (where N is the number of input features) separating the two classes and maximizing the margin between them. Support vectors are data points that are closer to the hyperplane and influence its position and orientation. For 2 input features, the classes are separated by a line, for 3 of them, a the line is replaced by a plane and this goes on into higher dimensions as the number of features increases. For data that's not linearly separable, one can use a non-linear SVM, which utilizes Kernel functions that transform non-linear spaces into higher-dimensional linear spaces.

To classify a new sample, we take the output of a linear function. If the output is greater than 1, the first class is predicted, and if the output is smaller than -1, the other class is predicted. The interval between -1 and 1 acts only as a margin, in contrast to how logistic regression works.

Chapter 3

Artificial neural networks

Artificial neural networks (ANNs), usually simply called neural networks (NNs), are computational systems inspired by the function of biological neural networks. They are at the heart of deep learning algorithms, a subset of machine learning algorithms. Neural networks can be used for many kinds of tasks, including classification, pattern recognition or detection, clustering, and many others. Some of the most notable real world applications include facial recognition, voice recognition, weather forecasting, or studying the behavior of social media users.

This chapter serves as a gradual introduction to NNs, which begins by introducing their basic building blocks in section 3.1, moving up to their structure and behavior in section 3.2. Section 3.3 covers the principles of their training and common problems to look out for while training them to solve classification problems. The last section, 3.4, explains the differences between the main NN types and their use cases.

3.1 Artificial neuron

Artificial neurons [35, p. 5] are the basic building blocks of every artificial neural network. A biological neuron is made out of dendrites, which act as the input vector, cell body (soma), which acts as the summation function, and axon, which gets its signal from the summation behavior inside the soma and transmits the signal to other cells in the body including other neurons. An artificial neuron (Figure 3.1) is a mathematical function basically mimicking a biological neuron. It takes the sum of its one or more inputs, each individually multiplied by its assigned weight, and then adds a constant value called bias to the sum. The sum is then passed through a function known as an activation function (also called transfer function), which "fires" to pass the information to subsequent neurons.

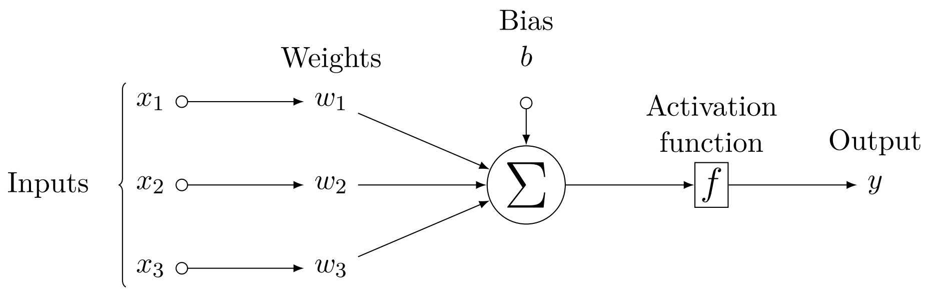


Figure 3.1: Example of an artificial neuron with three inputs.

The mechanism of an artificial neuron can be described mathematically:

$$y = f\left(b + \sum_{i=1}^{n} x_i w_i\right) = f\left(\sum_{i=0}^{n} x_i w_i\right) \tag{3.1}$$

where x_i is the i-th input of the neuron out of its n total inputs, w_i is the weight assigned to the i-th input, b is bias, often represented by bias input $x_0 = 1$ with weight $w_0 = b$, f is an activation function and y is the neuron's output.

The perceptron algorithm, invented in 1958 at the Cornell Aeronautical Laboratory by Frank Rosenblatt [27], is considered to be the first artificial neural network, although it only consists of a single neuron with a binary step activation function (see figure 3.2).

Activation function

Activation function is the part of a neuron that makes the final decision whether it should "fire" or not. It can be as simple as a binary step (activates above certain threshold) function or a linear function, but only when a non-linear function is used, then a two-layer neural network can be proven to be a universal function approximator [7]. When multiple layers use a linear activation function, the entire network is equivalent to a single-layer model, since the output of each layer is a linear combination of the inputs, and hence, the overall output can be expressed as a single linear equation.

There are many different activation functions, each better suited for different scenarios. Some of the most common ones include sigmoid—activation function of choice for binary classification, already seen in Figure 2.2, tanh (hyperbolic tangent), which usually finds applications in recurrent neural networks (introduced later in section 3.4) for natural language processing and speech recognition, ReLU (Rectified Linear Unit), which outperforms both sigmoid and tanh in computational speed, and softmax, used to build multi-class classifiers. Some of the mentioned activation functions are shown in Figure 3.2.

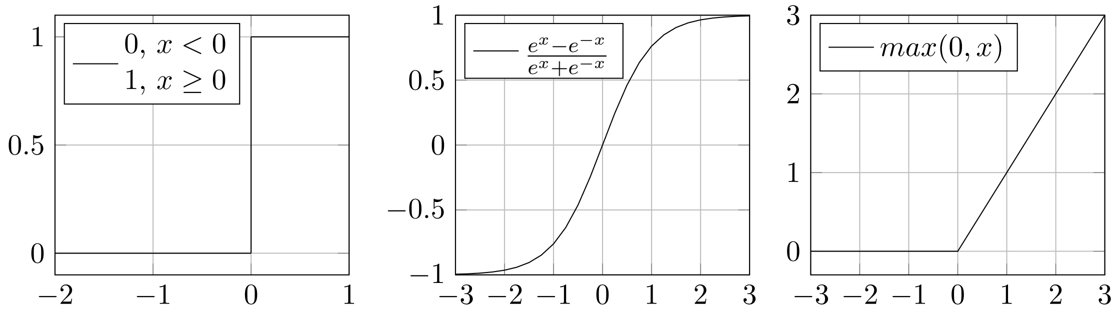


Figure 3.2: Graphs of binary step (left), tanh (middle) and ReLU (right) activation functions.

3.2 Artificial neural network

A collection of interconnected artificial neurons can form an artificial neural network [35, p. 6]. An ANN is comprised of layers of neurons including the input layer, output layer, and any number of layers between them, called hidden layers. Neural networks can be visualised with graphs, where each node represents a neuron and edges represent connections between them, as shown in Figure 3.4. The certain ways in which individual neurons can be connected are called topologies.

At first, the input layer takes an input in the form of numerical data. The hidden layers, present between input layer and output layer, perform all the calculations to find hidden features and patterns in the data. The input therefore goes through a series of transformations through succeeding layers, which finally results in an output conveyed using the output layer. In contrast with traditional computing models, an NN acts as a black box, in the sense that one cannot know what abstractly happens inside, since the network learns by exposure to labeled data and "stores the gained knowledge" by tweaking its parameters (weights of the inputs of its neurons). Knowledge in the form of many floating point numbers is obviously not very human-readable.

Topologies of neural networks

The topology (also called architecture or structure) of a neural network plays a fundamental role in its functionality and performance [11]. When building an NN to solve a specific problem, there is no single topology best suited for it, and while there are some empirically derived guidelines one can follow, it is mostly a matter of trial and error. One thing to keep in mind is that while adding more layers allows an NN to learn more complex patterns, it makes the network harder to train and much more prone to overfitting [6, p. 104], a very common problem explained later near the end of section 3.3.

One way to classify different NN topologies is distinguishing between feed-forward neural networks (FFNNs) and recurrent neural networks (RNNs), whose direct comparison is shown in figure 3.3. In a FFNN, the information flows in only one direction, so any layer can only gain information from the previous layers. In contrast, RNNs (often called feedback NNs) can have signals traveling in both directions by introducing loops—edges going from a given neuron to a neuron in either some previous layer or the current layer, which can include the neuron itself. These feedback loops introduce the concept of a network's memory, which is why these networks are used for processing sequential data, where context matters a lot, such as recognizing spoken sequences, detecting the next word/letter in a text, or even music composition[1].

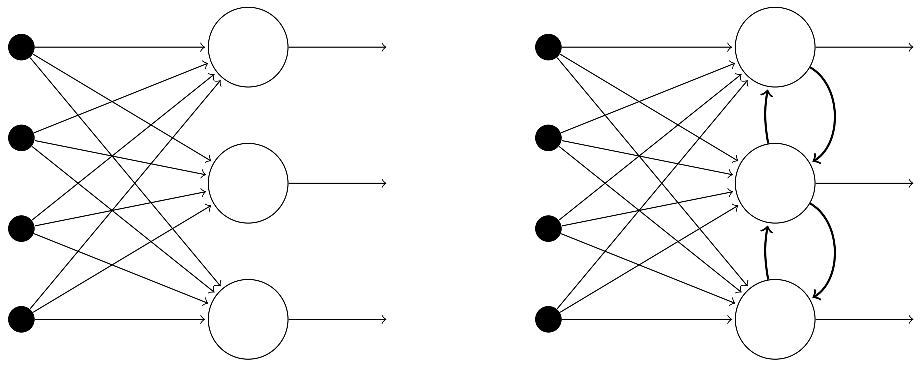


Figure 3.3: The comparison between feed-forward (left) and recurrent (right) neural networks. While FFNNs allow the signal to travel one way only, RNNs contain feedback loops.

[1]Note that in the context of music composition, the term recurrent neural networks refers to their use as generative models, where the network is trained to generate new music samples based on a given input sequence

With the advancement in digital technologies in the recent years, the demand for analyzing complex, high dimensional, and noise-contaminated datasets has risen quite a lot. This led to a rapid development of deep learning, part of a broader family of machine learning methods utilizing NNs. The adjective “deep” refers to the use of multiple layers in the network, which is then referred to as a deep neural network (DNN). While not being a formally defined rule, an ANN should contain at least two hidden layers to be called a DNN. The complexity of these networks makes them a very powerful tool for solving real-life problems. An example of a DNN is visualized in figure 3.4.

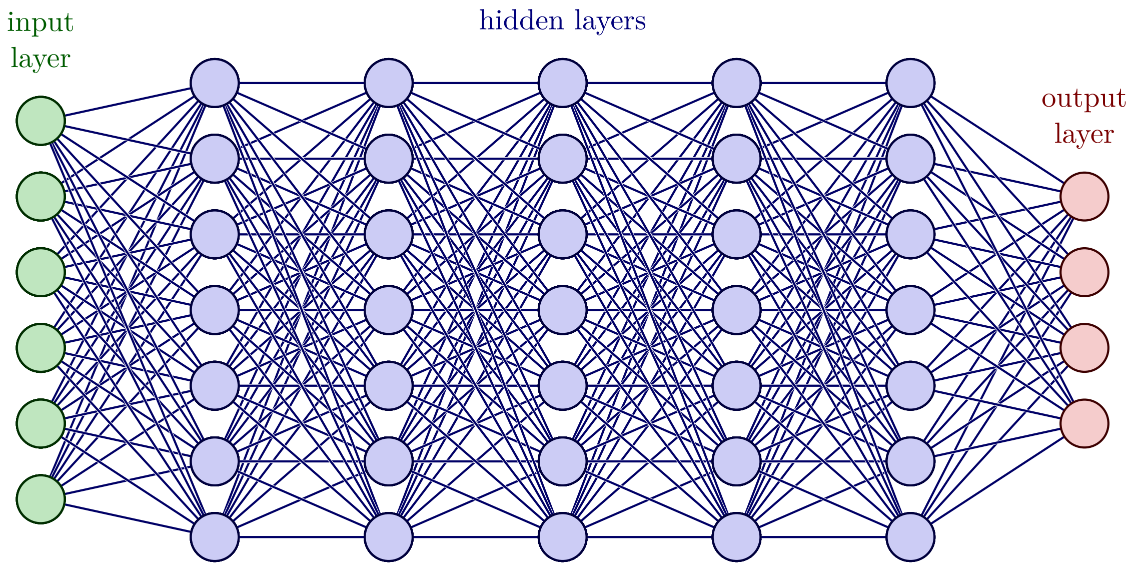


Figure 3.4: A dense deep neural network. The original source code for generating this figure using TikZ[2]was taken from [24].

Tensor operations

“Much as any computer program can be ultimately reduced to a small set of binary operations on binary inputs (AND, OR, NOT, and so on), all transformations learned by artificial neural networks can be reduced to a handful of tensor operations applied to tensors of numeric data” [6, p. 38]. Tensors are the basic data structure used not only by NNs, but by all current machine learning systems.

Most students and researchers are probably familiar with scalars, vectors and matrices. Those three structures can be referred to as 0D, 1D and 2D tensors (in this order), since tensors are a generalization of matrices to an arbitrary number of dimensions, as demonstrated in figure 3.5. In the context of tensors, a dimension is often called an axis or the tensor’s rank. The key attributes of a tensor are its number of axes, its shape (dimensions along each individual axis), and in the context of programming—the data type of its contained data. Even complex data can be broken down into a tensor—a dataset consisting of multiple videos can be represented as a 5D tensor, whose axes represent a sample’s order, number of the frame, x-coordinate (width), y-coordinate (height), and (color) channels.

[2]https://www.ctan.org/pkg/pgf

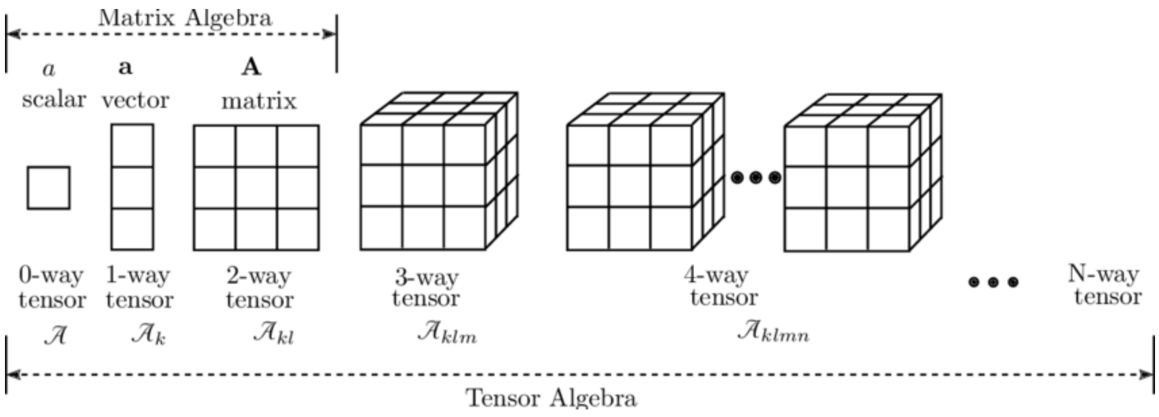


Figure 3.5: Tensors as generalizations of scalars, vectors and matrices [32].

Tensors can be added, multiplied and so on. Each layer of an NN can be thought of as a function that takes a tensor of a certain rank as its input, and returns a tensor of a certain rank as its output. Since all tensor operations can also be interpreted geometrically, the goal of a classification algorithm is basically just finding a chain of tensor operations that "uncrumbles" the feature space in a way that separates the classes.

3.3 Training a neural network

Training a neural network is the optimization process of finding the appropriate weights of its connections, so that the difference between the network's output and the expected output is minimal. This is possible thanks to a feedback loop algorithm called backpropagation (short for backward propagation of errors), which is the essence of NN training.

Though it is quite useful to know the basics of the mathematical foundation behind the training mechanism (explained right in the following subsection), a programmer doesn't have to worry too much about it when using a high level deep learning API such as Keras [4]. From the practical standpoint, it's more important to know how to preprocess data, split a dataset for training, validation and testing, build a model with reasonable architecture comprised of layers with activation functions fitting the given type of task, etc. Some guidelines for these problems are spread throughout this chapter, and mainly through the next chapter 4, which explores some of the extensive options Keras has to offer.

Loss function

Before diving into the training process itself, let's introduce a way of measuring the difference between the expected output and the model's prediction. This is the role of a loss function (synonymous with cost function or error function, but cost sometimes refers to the average loss over the entire training dataset). It maps an event or values of one or more variables onto a real number representing some "cost" associated with the event called loss, which we aim to minimize.

The most common loss for classification is cross-entropy, which measures the performance of a classification model whose output is a probability value between 0 and 1. Binary cross-entropy (3.2), a special case of cross-entropy with only two classes, is not only the most commonly used loss for binary classification, but is also often used for multi-label classification. Assume that $y_i \in \{0, 1\}$ marks the correct class of the i-th training sample and p_i is the predicted probability of the sample belonging to class 1.

$$L_{BCE}(p_i, y_i) = -(y_i \log(p_i) + (1 - y_i) \log(1 - p_i)) \tag{3.2}$$

The general version of cross-entropy, used for multi-class classification, is often called categorical cross-entropy (3.3). Assume that M is the total number of classes, $\vec{\boldsymbol{y}}_i$ is a vector of size M that indicates to which class the sample belongs, $\vec{\boldsymbol{p}}_i$ is a vector of predicted probabilities for each class, $y_{i,c}$ is a binary indicator whether sample i belongs to class c and $p_{i,c}$ is the corresponding predicted probability.

$$L_{CCE}(\vec{\boldsymbol{p}}_i, \vec{\boldsymbol{y}}_i) = -\sum_{c=1}^{M} y_{i,c} \log(p_{i,c}) \tag{3.3}$$

Other classification losses include hinge loss (3.4), developed primarily for SVM model evaluation, which apart from wrong predictions also penalizes right predictions that are not confident, negative loglikelihood loss, and KL/JS divergence. Note that in the case of hinge loss (in the version for binary classfication shown bellow), $y_i \in \{-1, 1\}$ and $p_i \in (-1, 1)$.

$$L_H(p_i, y_i) = max(0, 1 - y_i p_i) \tag{3.4}$$

Gradient descent

The gradient of a function of n variables can be interpreted as a vector pointing in the direction of the fastest growth of the function. In the case of an NN, n is equal to the total number of weights in the network. Adjusting the network's weights is done according to the gradient descend method, since we always want to minimize the loss function with the goal of finding its global minimum.

Let's demonstrate this on a simple example of training a linear model consisting of a single neuron without an activation function [30]. Training on each sample can be thought of as finding solutions for linear equation (3.5) with variables $a_0, a_1, ..., a_n$.

$$y = a_0 x_0 + a_1 x_1 + \cdots + a_n x_n \tag{3.5}$$

The model's guess based on its current weights can then be written down as equation (3.6).

$$\hat{y} = w_0 x_0 + w_1 x_1 + \cdots + w_n x_n \tag{3.6}$$

We calculate the cost (3.7) between the expected result and the model's prediction. For the sake of the example, let's use the squared error[3] function, because it's easy to differentiate.

$$C = \frac{1}{2}(y - \hat{y})^2 = \frac{1}{2}(y - w_0 x_0 - w_1 x_1 - \ldots - w_n x_n)^2 \tag{3.7}$$

We calculate the gradient of the cost (3.8), which is nothing more than a vector consisting of its partial derivatives each with respect to an individual weight (3.9).

$$\nabla C(\vec{\boldsymbol{w}}) = \left\langle \frac{\partial C}{\partial w_0}, \frac{\partial C}{\partial w_1}, \ldots, \frac{\partial C}{\partial w_n} \right\rangle \tag{3.8}$$

$$\frac{\partial C}{\partial w_i} = \frac{1}{2} \cdot 2(y - w_0 x_0 - w_1 x_1 - \ldots - w_n x_n)(-x_i) \tag{3.9}$$

[3]Mean squared error (MSE) is the most commonly used loss function for regression.

Moving to the final step (3.10), the weights are updated by subtracting the calculated gradient multiplied by the learning rate (denoted by η), a tuning parameter that determines the step size at each iteration. While the learning rate can be as simple as a constant, there are many optimization algorithms that scale it dynamically to ensure faster and more reliable convergence towards the function's global minimum [28].

$$\vec{\boldsymbol{w}} \leftarrow \vec{\boldsymbol{w}} - \eta \cdot \nabla C(\vec{\boldsymbol{w}}) \iff \forall i \in \{0, 1, ..., n\} : w_i \leftarrow w_i - \eta \cdot \frac{\partial C}{\partial w_i} \tag{3.10}$$

These steps should be repeated until the algorithm converges. This special case of calculating the cost for every iteration is called stochastic gradient descent. In practice, the cost is usually calculated as the average loss over batches of training samples and sometimes even over entire epochs[4].

Backpropagation

Backpropagation, proposed back in 1986 by David E. Rumelhart [29], is an efficient method of computing gradients in directed graphs of computations, such as multi-layer neural networks. When training an NN, every iteration consists of two passes—the forward pass and the backward pass. In the forward pass, the data is fed to the input layer, goes through the hidden layers and finally at the output layer, the network's prediction gets produced, based on which the network's error can be calculated via the loss function. In the backward pass, the flow is reversed so that the error gets propagated from the output layer all the way back to the input layer, while updating the weights in each layer. This process of propagating the error backwards is called backward propagation, or simply backpropagation.

The following interpretation of the backpropagation algorithm 3.1 draws inspiration from [8] and [33]. To understand the notation, $x_{ij}^{(l)}$ is the i-th input of neuron j in layer l and $w_{ij}^{(l)}$ is the associated weight. The weighted sum computed by neuron j in layer l is denoted by $z_j^{(l)}$, which then transforms to the neuron's output $y_j^{(l)}$ by passing through its activation function f. Since this time the neurons have an activation function, as wasn't the case in the example in previous subsection, it was appropriate to break down the expressions inside step 4 using the chain rule, which states that $f((g(x))' = f'(g(x))g'(x)$.

[4] An epoch is a single iteration over the entire training dataset.

Algorithm 3.1 Backpropagation

1. **Initialization**

   Set all weights within the network to some small initial value.

   $w_{ij}^{(l)}(0) = \textit{some small initial value}$

   Set the iteration counter to 0.

   $p = 0$

2. **Forward pass**

   Calculate the output of every neuron j in each layer l starting from the first hidden layer and ending with the output layer (the input layer only passes raw input).

   $$y_j^{(l)}(p) = f\left(z_j^{(l)}(p)\right) = f\left(\sum_{i=0}^{n} w_{ij}^{(l)}(p) x_{ij}^{(l)}(p)\right)$$

   Note that n is the number of inputs of neuron j and $w_{0j}x_0$ is its bias term.

3. **Error calculation**

   Calculate the error $C(p)$ between the expected output for the current sample and the output obtained from the neuron(s) in the output layer.

   If the training data is split into batches, repeat step 2 for every sample in the current batch and then calculate $C(p)$ as the average loss across the batch.

4. **Backward pass**

   First, calculate the derivatives of the error in terms of the weights between the last hidden layer and the output layer.

   $$\frac{\partial C}{\partial w_{ij}^{(out)}} = \frac{\partial C}{\partial y_j^{(out)}} \frac{\partial y_j^{(out)}}{\partial z_j^{(out)}} \frac{\partial z_j^{(out)}}{\partial w_{ij}^{(out)}}$$

   Continue calculating derivatives of the error in terms of the weights between all the remaining pairs of neighboring layers going from right to left.

   $$\frac{\partial C}{\partial w_{ij}^{(l)}} = \frac{\partial C}{\partial y_j^{(l)}} \frac{\partial y_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}} = \sum_{k \in K} \left(\frac{\partial C}{\partial y_k^{(l+1)}} \frac{\partial y_k^{(l+1)}}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial y_j^{(l)}} \right) \frac{\partial y_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}}$$

   Note that K is the set of all neurons in layer $l+1$ connected to neuron j.

   Update all weights within the network.

   $$w_{ij}^{(l)}(p+1) = w_{ij}^{(l)}(p) - \eta \cdot \frac{\partial C}{\partial w_{ij}^{(l)}}$$

5. **Repetition**

   Increase the iteration counter.

   $p \leftarrow p + 1$

   Continue with step 2, until the target number of iterations is reached or until the error decreases below a certain threshold.

Common problems

Training a neural network is not so straightforward, since there many pitfalls one can find himself in. Some problems can be caused by inappropriately chosen learning rate—a value too small slows down the progress, while a value too high introduces oscillations and instabilities leading to divergence. It is also not uncommon to get stuck in a local minimum while performing gradient descent (since hardly any loss function is convex), in which case it is recommended to introduce some element of randomness.

On a larger scale, the most common problem is probably overfitting [6, p. 104]. An NN is trained on a dataset which is meant to represent the problem it is being trained to solve, but since the dataset contains only finite number of samples, it can only represent a subset of the problem space. Overfitting occurs when the model becomes too accustomed to its training data. It fails to generalize the problem and performs poorly when presented with new data. It is caused by the model memorizing patterns in irrelevant information (noise) within the dataset, which usually happens when the model trains for too long on sample data or when the model is too complex. Some techniques to prevent overfitting (other than getting more training data) include reducing the network's capacity by removing some hidden layers or reducing the number of neurons in them, and applying regularization, which adds additional cost to the loss function for larger weights. Three most popular regularization techniques are L1 and L2 regularization[5], and also introducing dropout layers, which randomly remove certain features by randomly setting some inputs of a layer to zero.

The opposite phenomenon to overfitting is underfitting [6, p. 104], which occurs when the model is unable to accurately capture the relationship between input and output variables, leading to high error rates on both the training set and unseen data. This usually happens when there is simply not enough training data or the model is too simple. Techniques to reduce underfitting include increasing the model's complexity, increasing the number of features by performing feature engineering, removing unwanted noise from the data or simply increasing the duration of the training. Underfitting is usually easier to identify than overfitting, since an overfitted model reaches high accuracy in training. A visual interpretation of both underfitting and overfitting can be seen in figure 3.6.

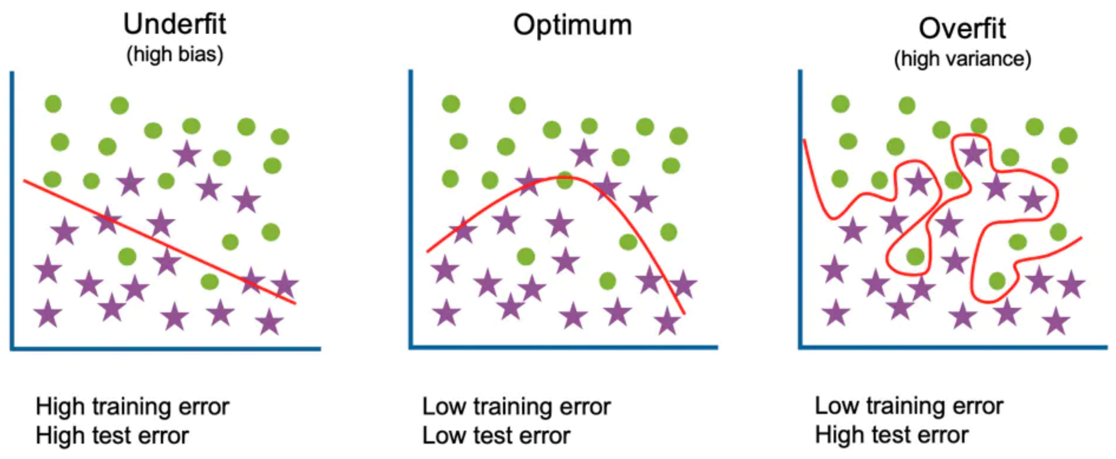


Figure 3.6: Visual interpretation of underfitting and overfitting [17].

[5]Penalizing the total loss by either absolute values (L1) or squares (L2) of the weights.

3.4 Main types of neural networks

This section explains the differences between the three main ANN types.

Multi-layer perceptron

Multi-layer perceptron (MLP) refers to a standard feedforward neural network whose three or more layers are fully connected [6] and (except for the input layer) utilize some non-linear activation function. A topology of an MLP may look like the one previously shown in figure 3.4. Note that the "perceptrons" of which MLPs are composed of are not perceptrons in the strictest possible sense, but rather ordinary artificial neurons, since they can employ arbitrary activation functions (not limited to threshold-based functions).

Since an MLP accepts a vector (1D tensor) of numerical features as input, data is mostly provided in a tabular format, such as csv [7] files or spreadsheets. In these datasets, rows separate individual instances, while columns separate individual features, including the class label in the case of classification problems. MLPs can also work with more complicated data, such as images, text data, or timeseries data, if there is a sensible way to convert the data into a vector. For instance, a 28x28 pixel grayscale image can be converted to a vector consisting of 784 numerical features representing individual pixels. The main problem with using MLPs for image classification is that they can only recognize global patterns. For example, an MLP can be trained to classify different shapes if they're all similarly scaled and located in the same part of an image, but fails to classify them once they're presented in different scales and locations.

Convolutional neural network

A convolutional neural network (CNN/ConvNet) is an NN type most commonly used for processing image data [25], since unlike an MLP, it has the ability to recognize local patterns in multidimensional data. A CNN looks for spatial relations (relations between nearby pixels) instead of only looking at an image as a whole—this property is called translation invariance and allows the network to recognize objects regardless of their position in an image. Capturing the spatial and temporal dependencies in an image is done through the application of relevant filters.

The central part of any CNN, that also gave this NN type its name, is the convolutional layer. Convolution is a mathematical operation, in this sense refering to the process of adding each element of an image to its local neighbors, weighted by a convolutional kernel, as shown in figure 3.7. A convolutional layer contains a set of kernels, parameters of which are to be learned throughout the training. Each kernel is used for detecting a specific feature in the data, creating a feature map. A structure of multiple kernels stacked together is called a filter. Two important terms when considering convolution are padding—a process of adding layers of zeros around the image so that the kernel can overlap the image in every possible position which has an element of the original image at the kernel's center, so that the feature map keeps the size of the input image, and stride—the number of elements traversed between steps. A stride of more than 1 obviously downsamples the image even if it's padded along the edges.

[6] A layer that is fully connected with its preceding layer is called a dense layer.

[7] A CSV (comma-separated values) file is a text file that has a specific format which allows data to be saved in a table structured format.

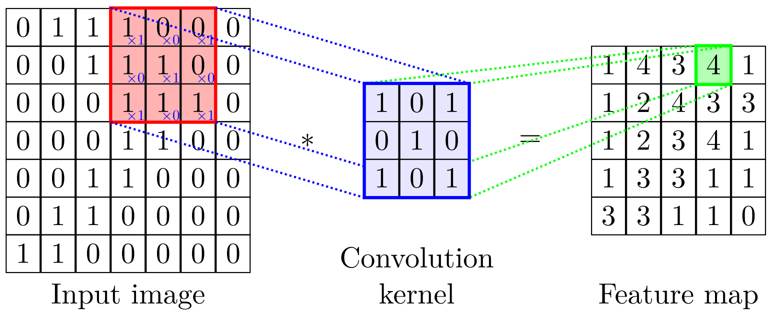


Figure 3.7: Convolution of a 7x7 image with a 3x3 kernel (no padding, stride set to 1) [37].

Although convolutional layers summarize the presence of features in an input image, the feature maps they produce are sensitive to location of the features in the image. A common approach for reaching local translation invariance is to downsample the feature map. Although this can be easily achieved by changing the stride, a more robust and common approach is to introduce pooling layers. Pooling transforms groups of elements (called patches) in a feature map into single elements, based on the selected pooling operation—the most common ones are maximum pooling (max pooling), which selects the highest value in the patch, and average pooling, which calculates the average of the values. Both of these pooling operations are shown in figure 3.8. Some non-linear function (e.g. ReLU) is usually applied to a feature map before it gets passed to a pooling layer. It should also be noted that while pooling can help the network become more translation invariant, it does not address its variance to other transformations such as rotations or changes in scale of an object, and even though some special CNN architectures addressing this issue were already proposed [20], the usual way of combating this issue is to simply include slightly modified (rotated, mirrored, etc.) copies of images already contained in the dataset when training—this is part of a technique called data augmentation.

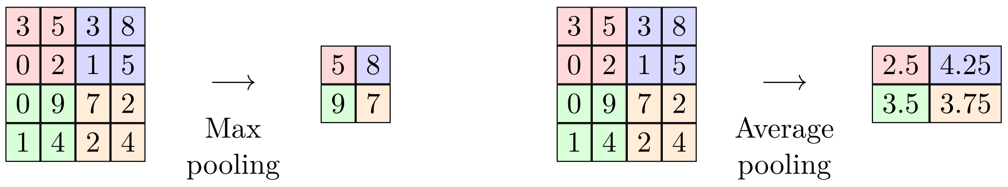


Figure 3.8: Application of 2x2 max pooling and 2x2 average pooling to a 4x4 feature map.

The overall architecture of a CNN can be broken down into two parts, as can be seen in figure 3.9. The first part, responsible for feature extraction, is made of convolutional layers, each usually followed by a pooling layer. The first convolutional layer is usually used to detect low-level features, such as horizontal or vertical lines, but the deeper a convolutional layer is, the higher-level features it can recognize, such as entire objects or even facial expressions. The second part of a CNN, responsible for the classification itself, consists of dense layers and essentially behaves as a standard MLP. Feature maps usually have more than one dimension, but a dense layer only accepts a vector as input. To solve this problem, we can introduce a flatten layer, which transforms all the final pooled feature maps into a single large vector.

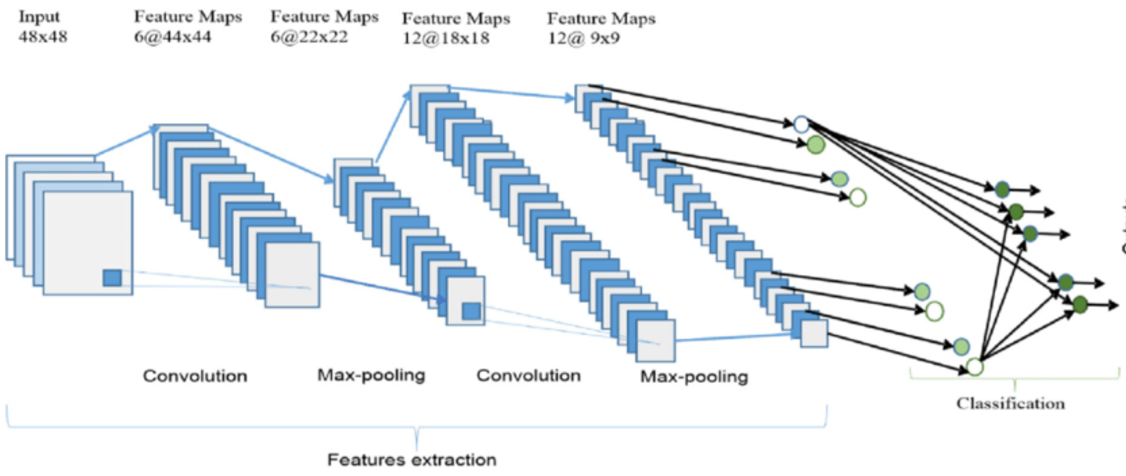


Figure 3.9: Architecture of a CNN composed of an input layer, multiple alternating convolution and max-pooling layers, one dense layer and an output layer [1, Figure 9].

Recurrent neural network

Sometimes we might want to work with sequential data (e.g. time series), where the chronological order of elements creates a certain context. Most prominent is natural language processing, which includes both text (as sequences of either words or characters) and speech. For this kind of problem, let's consider a special NN type that has a sequential memory, which provides the ability to remember preceding elements and take them into consideration. This is called a recurrent neural network (RNN), and its concept is based on David Rumelhart's work in 1986 [29]. Apart from sequence classification and regression, RNNs are also great for building generative models that require a sequential output, such as generating text or even handwriting.

In an RNN, information gets passed not only to succeeding layers, but also backwards through loops, as was previously shown in figure 3.3 when mentioning RNNs in the context of NN topologies. Each recurrent cell has its hidden state, which is calculated as a function of its previous hidden state and its input. This means that the network as a whole has a hidden state, that strongly affects its response to a given input.

A huge problem RNNs face is their short-term memory caused by the vanishing gradient problem [14], from which they tend to suffer more than other NN architectures the more steps they process. When a FFNN is trained using backpropagation, the backpropagated error signal typically decreases exponentially as it propagates further from the output layer, making the weight adjustments in those layers less prominent. All neurons that participated in the calculation of the output get updated—this gets very complicated in the case of an RNN, where it's not just the neurons preceeding the output layer, but all of the neurons involved back in time, creating a long path for the error to propagate through. This diminishes the effect of earlier inputs on the network's training. To overcome this problem, two specialised versions of RNN were created.

Long Short Term Memory (LSTM), capable of remembering long sequences for a long period of time, was introduced by Hochreiter and Schmidhuber in 1997 [15]. The new hidden state of a LSTM cell is calculated not only from the previous hidden state of the input, but also from information stored in the long term memory. There is a total of three gates that LSTM uses—at each timestep, the input gate decides what information will be stored in long term memory, the forget gate decides which of this information will be kept or discarded, and the output gate calculates the new hidden state.

Gated recurrent unit (GRU), introduced in 2014 by Kyunghyun Cho [3], is designed similarly as LSTM, but it aims to solve the same problem by incorporating an operating mechanism that consists of the update gate, responsible for determining the amount of previous information getting passed to the next state, and the reset gate, which decides how much of the past information is going to be neglected (forgotten). At first, it takes the input, and the previous hidden state multiplied by the reset gate's output, which results in something called the candidate's hidden state. This state, together with the update gate, is then used to calculate the current hidden state. The cells of a standard RNN, LSTM and GRU are all shown in figure 3.10.

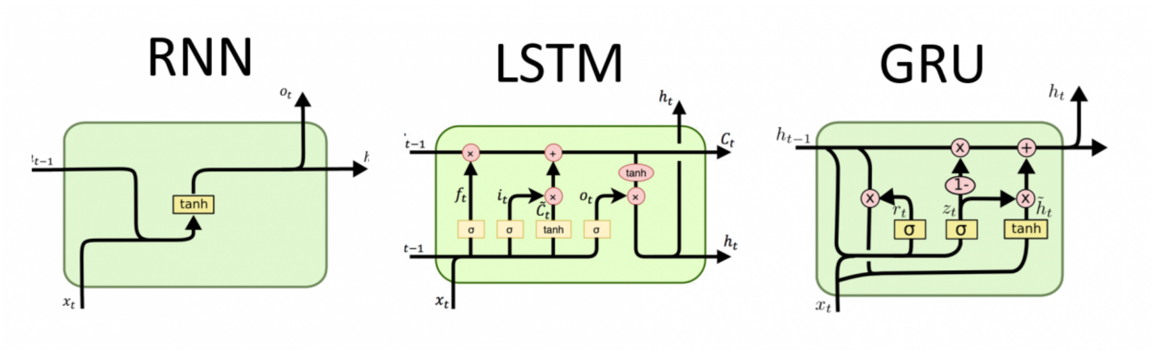


Figure 3.10: Comparison of standard RNN cell, LSTM cell and GRU cell [21].

Chapter 4

The Keras library

Keras is an open source library that provides a high-level interface for working with artificial neural networks in Python[1]. It uses the TensorFlow[2] library as its backend[3], for which it provides a high-level API that makes deep learning more accessible to a wider range of users through its ease of use. Keras, written in Python, was developed with a focus on enabling fast experimentation, with the core idea that “being able to go from idea to result as fast as possible is key to doing good research” [4].

Since the range of options Keras has to offer is very extensive, the goal of this chapter is to explore some of its main offerings in the context of classification, in the form of a guide starting with a model’s building phase in section 4.1, followed by it’s compilation in section 4.2, its training in section 4.3, and finally its testing and general usage in final section 4.4. Each section briefly introduces the many choices a user has along the way of solving a given classification problem using Keras. Much of the information in this chapter is drawn from the official Keras documentation [4], and from the book Deep Learning in Python [6] written by François Chollet, the author of Keras himself.

4.1 Building a model

There are three ways to create a model in Keras. The most straightforward way is to use the `Sequential` class, which groups a linear stack of layers into a sequential model with a single input and a single output. The second option is using the Keras functional API, which is more flexible and capable of creating models with non-linear topologies, shared layers, and even multiple inputs and outputs. It provides an easy-to-use way of creating directed acyclic graphs of layers. A direct comparison between using the `Sequential` class and using the functional API is shown in listing 4.1. The last option is to use model subclassing, by writing custom subclasses of the `Model` class and the `Layer` class.

```
import tensorflow as tf

model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(16, activation='relu', input_shape=(12,)))
```

[1]https://www.python.org/

[2]https://www.tensorflow.org/

[3]Up until version 2.3, Keras could also be configured to utilize a different library, such as Theano or CNTK. Since version 2.4, only TensorFlow is supported. Keras has also been embedded as part of the TensorFlow package since the release of TensorFlow 2.0.

```
model.add(tf.keras.layers.Dense(8, activation='relu'))
model.add(tf.keras.layers.Dense(1, activation='sigmoid'))

input_layer = tf.keras.Input(shape=(12,))
hidden_layer = tf.keras.layers.Dense(16, activation='relu')(input_layer)
hidden_layer = tf.keras.layers.Dense(8, activation='relu')(hidden_layer)
output_layer = tf.keras.layers.Dense(1, activation='sigmoid')(hidden_layer)
equivalent_model = tf.keras.Model(inputs=input_layer, outputs=output_layer)
```

Listing 4.1: An MLP binary classification model built using the `Sequential` class and an equivalent model built using the functional API.

Layers and activation functions

A model is series of layers grouped into a `Model` object with training and inference features. Each layer consists of a tensor-in tensor-out computation function and usually also some state (the layer's weights). The most basic is probably the `Dense` layer, which contains `units` number of neurons, each connected to every neuron in the previous layer. All the available layers can be grouped into the following categories:

- **Core layers**: `Dense`, `Input` object, `Embedding`, `Activation`, `Masking`, `Lambda`
- **Convolution layers**: `Conv1D`, `Conv2D`, `Conv3D`, `SeparableConv2D`, `DepthwiseConv2D`, `Conv2DTranspose`, etc.
- **Pooling layers**: `MaxPooling1D`, `AveragePooling2D`, `GlobalMaxPooling2D`, `GlobalAveragePooling3D`, etc.
- **Recurrent layers**: `SimpleRNN`, `LSTM`, `GRU`, base `RNN`, `ConvLSTM2D`, etc.
- **Other layer categories**: Preprocessing (e.g. `TextVectorization`), normalization (e.g. `BatchNormalization`), regularization (e.g. `Dropout`), attention, reshaping (e.g. `Reshape`, `Flatten`), merging, locally-connected layers, and activation layers

Every layer type accepts a different set of arguments, through which can the layer's behavior be modified. These arguments range from obvious ones like the number of units or the activation function, to very specific ones like the kernel regularizer function (which introduces weight decay). Many of the arguments, including the activation function and the kernel regularizer, can be passed either as an object instance, whose parameters can be tweaked, or as a string identifier, in which case the default values will be used. Both of these options are demonstrated in listing 4.2.

Activation function is without a doubt one of the most prominent layer attributes. Keras `activations` module offers these built-in functions: `relu`, `sigmoid`, `softmax`, `softplus`, `softsign`, `tanh`, `selu`, `elu` and `exponential`. Activation functions can not only be passed as an object instance or a string identifier, but also added as a separate layer (see listing 4.2), which is also the only was to use activations that maintain a state—for instance `PReLU`, and some other advanced activation functions such as `LeakyReLU`.

In a classification model, the hidden layers usually utilize either `relu`, `sigmoid` or `tanh` activation function. The output layer, however, depends on the given type of classification. We use a `Dense` layer containing either a single neuron with `sigmoid` for binary classification, *number of classes* neurons with `softmax` for multi-class classification as, shown at the end of listing 4.2, or *number of classes* neurons with `sigmoid` for multi-label classification.

```
model = tf.keras.Sequential()
model.add(tf.keras.layers.Conv2D(
    input_shape=(30, 30, 3), # 30x30 RGB image (3 color channels)
    filters=64, kernel_size=(3, 3), strides=(1, 1), padding='same',
    kernel_regularizer=tf.keras.regularizers.L2(0.001),
    kernel_initializer='zeros', bias_regularizer='l1',
    bias_initializer=tf.keras.initializers.Zeros()
))
model.add(tf.keras.layers.LeakyReLU(alpha=0.1))
... # Pooling, Flatten, Dense, etc.
model.add(tf.keras.layers.Dense(units=6, activation='softmax'))
```

Listing 4.2: A CNN six-class classification model with specifically parameterized layers.

Keras models can be saved to a file by the `save_model()` function (or `Model`'s `save()` method) and loaded by the `load_model()` function. The same can be done with only the current weights instead of the whole model, by the `Model`'s `save_weights()` and `load_weights()` methods. The model's summary, which contains information about the consecutive layers (e.g. layer type, number of trainable parameters), can be printed using `Model`'s `summary()` method. Models can also be plotted and saved as an image using the `plot_model()` function from Keras utilities.

4.2 Compiling a model

Once a model is built, the `Model`'s `compile()` method is used to configure it for training. Apart from some very specific optional arguments, we specify the optimizer and the loss function to be used for training, and a list of metrics to be evaluated during training and testing. All of these can be again passed either as a object instance or as an string identifier (default parameters), an example can be seen in listing 4.3.

```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
    loss=tf.keras.losses.BinaryCrossEntropy(),
    metrics=['accuracy', tf.keras.metrics.FalsePositives()])
```

Listing 4.3: Configuring a Keras model for training using its `compile` method.

Optimizer

An optimizer is an algorithm that dictates how the model's attributes, such as weights and learning rate, are updated in response to the output of the loss function. The simplest optimizer Keras offers is `SGD`, which is the stochastic gradient descent algorithm, with an optional `momentum` hyperparameter that accelerates the descent in the relevant direction.

The other available optimizers tend to be more complicated, since they all belong to the family of adaptive optimizers [28], introduced to solve the issues of gradient descent. This includes `Adagrad`, which adapts the learning rate to perform more significant updates for rarely occurring features, `Adadelta` and `RMSprop`, both of which improve the previous algorithm by taking a fixed number of past gradients into consideration, and `Adam`—the overall most commonly used optimizer, which adds to the advantages of all these previous algorithms by storing an exponentially decaying average of past gradients (similar to momentum). The remaining Keras optimizers are `Adamax`, `Nadam` and `Ftrl`.

Loss

As already explained in section 3.3, the purpose of a loss function is to compute the quantity that a model seeks to minimize during training. The built-in losses Keras offers can be divided into three categories: Probabilistic losses, regression losses, and hinge losses for ”maximum-margin” classification. Since this thesis revolves around classification, we can neglect regression losses, apart from maybe mentioning `MeanSquaredError` and `MeanAbsoluteError` since these two are very common.

Cross-entropy based losses can be found in the probabilistic category. For binary classification, we use `BinaryCrossentropy`, while for multi-class classification, we can use either `CategoricalCrossentropy` in combination with label-encoded (integer) class labels, or `SparseCategoricalCrossentropy` with one-hot-encoded class labels (both of these encoding techniques will be explained in the next section 4.3). The probabilistic losses also include `KLDivergence` (Kullback–Leibler divergence), which measures a very similar quality to cross-entropy, and `Poisson`, used for regression with discrete variables.

The last category includes three versions of the hinge loss function: standard `Hinge`, `SquaredHinge` and `CategoricalHinge`. The usage of these losses in Keras is quite rare, since they are mostly used for SVMs and only very rarely for NNs.

Metrics

A metric is a function that is used to evaluate the performance of a model, but unlike loss functions, the results from evaluating a metric are not used when training the model. The built-in metrics can be grouped into six categories. Three of these categories correspond to the three previously listed loss categories, since any loss function can also be used as a metric. Also, there is only a single metric in the image segmentation category—`MeanIoU` (mean Intersection over Union), used to measure the accuracy of an object detector.

Probably the most common are accuracy metrics, which tell us how often the model predicted the correct outcome. When the metric `Accuracy` is specified, Keras detects the output shape and automatically determines which type of accuracy shall be used (but it can also be specified manually). These include `BinaryAccuracy`, which calculates how often predictions match binary labels (binary classification), `CategoricalAccuracy` and `SparseCategoricalAccuracy`, which calculate how often predictions match either one-hot labels or integer labels respectively (multi-class classification), and finally `TopKCategoricalAccuracy` and `SparseTopKCategoricalAccuracy`, which compute how often targets are in the top `K` predictions (multi-class classification).

The last category contains classification metrics based on true/false positives and negatives, implementing the metrics defined in section 2.2. The simplest of these metrics are naturally `TruePositives`, `TrueNegatives`, `FalsePositives` and `FalseNegatives`, which are self-explanatory, while the most complex one is probably `AUC`, which approximates the AUC (Area under the curve) of the ROC or PR curves. The remaining metrics are `Precision`, `Recall`, `PrecisionAtRecall`, which computes the best precision where $recall \geq specified\ value$, and `SensitivityAtSpecificity` complemented by `SpecificityAtSensitivity`, which compute the best sensitivity where $specificity \geq specified\ value$ and vice versa.

4.3 Training a model

A Keras model can be trained either by calling its `fit()` method, which trains the model for a fixed number of epochs, or the `train_on_batch()` method, which runs a single gradient update on a single batch of data. The arguments of the `fit()` method include the input data (unlabeled samples), target data (corresponding class labels), `batch_size` (number of samples per gradient update), `epochs` (number of epochs to train the model for), and some others. Keras is usually combined with other Python libraries, since the input and target data passed to the `fit()` method can be in the form of Numpy[4] arrays, TensorFlow tensors, Pandas[5] DataFrames, or some of the few other more specialized formats, including a dictionary mapping input names to the corresponding array/tensors, suitable for models that have named inputs.

Part of the training data can be used for the model's validation using either the `validation_split` or `validation_data` argument of the `fit()` method. A validation dataset is a sample of data held back from training the model, giving out an unbiased estimate of the model's performance, which is especially useful for monitoring the generalizability of the model and detecting overfitting. The `fit()` method returns a `History` object, whose `History.history` attribute is a record of training loss and metrics values at successive epochs, as well as validation loss and metrics values (if applicable). Value of the `verbose` argument controls how much information will be printed at the end of each epoch, such as the values of loss and metrics (also for validation if applicable), or even an animated progress bar.

Keras also offers a few built-in datasets provided by the `tf.keras.datasets` module. These include famous datasets such as the MNIST[6] handwritten digits classification dataset [9], the IMDB movie review sentiment classification dataset[7] [22], the CIFAR10 and CIFAR100 small images classification datasets[8], and three other datasets. Training a model on a built-in dataset using the `fit()` method is shown in listing 4.4.

```
# Loading the training and testing data
(X_train, y_train), (X_test, y_test) = tf.keras.datasets.mnist.load_data()
# Training the model
model.fit(X_train, y_train, batch_size=32, epochs=80, validation_split=0.1)
```

Listing 4.4: Training a Keras model on the built-in MNIST handwritten digits classification datasets using the `Model`'s `fit()` method.

Data preprocessing

Since a dataset is usually a set of measurements of some phenomena, the data often has to be transformed into a form that an NN can accept as input. This includes encoding non-numerical categorical data into numerical values and reshaping the input—for example, a 2D CNN usually expects an input of shape `(rows, cols, channels)`. Reshaping can be done in advance (often using Numpy), or by including one or more `Reshape` layers in the model's architecture. Even though Keras offers some utilities (e.g. the `to_categorical`

[4]https://numpy.org/
[5]https://pandas.pydata.org/
[6]http://yann.lecun.com/exdb/mnist/
[7]https://ai.stanford.edu/~amaas/data/sentiment/
[8]http://www.cs.toronto.edu/~kriz/cifar.html

function), external libraries including more advanced preprosessing tools, such as scikit-learn[9], are usually used alongside Keras.

There are three main techniques for encoding categorical data. Label encoding (also called integer or ordinal encoding) represents each of n total categories as an integer (either from 1 to n or 0 to $n-1$). Label encoding should only be used in cases where there is some kind of hierarchical relationship between the categories. If we tried to encode, let's say, a *color* attribute in that way, the network would learn some unintended relationship based on the order in which the colors were encoded, such as $red < green < yellow$. If no relationship is present between the categories, one-hot encoding should be used, which means transforming each value into a vector of length n, with all of its components set to 0 except for the single one representing the given category, which is set to 1. The direct comparison of label and one-hot encoding on a weather feature example is shown in table 4.1. The last technique is called learned embedding (or simply embedding) and is based on mapping each category to a distinct vector, whose properties are adapted while training the NN, which allows for miscellaneous relationships between the categories to be learned. This technique is often referred to as word embedding, since it was originally developed to provide a distributed representation for words (e.g. allowing similar words to have similar vector representations).

Sample number	**Default feature**	**Label encoded feature**	**One-hot encoded feature**
1	`"sunny"`	1	[1, 0, 0]
2	`"cloudy"`	2	[0, 1, 0]
3	`"sunny"`	1	[1, 0, 0]
4	`"rainy"`	3	[0, 0, 1]

Table 4.1: Direct comparison of label encoding and one-hot encoding.

Further preprocessing can be done to optimize the training process, generally leading to better results [18]. This includes techniques like data augmentation (explained in the CNN subsection of section 3.4), normalization, standardization and batch normalization. Normalization refers to scaling the values from different ranges to a common range, usually the interval $(0, 1)$, while standardization refers to transforming the data such that the mean of the data is equal to zero and the standard deviation to one. Batch normalization (`BatchNormalization` layer in Keras) is based on normalising each batch individually, and is often used after convolutional layers.

It is extremely important to use different data for training, validation and testing to check how well the model is generalized and to detect overfitting in time. While there is no optimal dataset split percentage, since it depends on the given problem, three standard ways to split a dataset (training|validation|testing) are 80%|10%|10%, 70%|15%|15% and 60%|20%|20%. In general, the larger a dataset is, the lesser proportion of the data is required for testing and validation. However, much more important than finding good split percentages is to ensure that these three datasets are about equally balanced with respect to the distinct classes and feature values.

[9] `https://scikit-learn.org`

Callbacks

In Keras, a callback is an object that can perform actions at various stages of training/testing (before/after a batch, at the start/end of an epoch, etc.). There are several built-in callbacks, such as `ModelCheckpoint`, which periodically saves the trained model (or its weights) to a file, `CSVLogger`, which streams epoch results to a CSV file, or `EarlyStopping`, which stops the training when a monitored metric had stopped improving.

To create a custom callback, we simply create a subclass of the `Callback` class and override a set of methods called at various stages of training, testing and predicting, such as `on_(train|test|predict)_begin()`, `on_(train|test|predict)_batch_end()`, `on_epoch_begin()`, etc. A list of callbacks can be passed to the `fit()` method (and also to the `evaluate()` and `predict()` methods introduced in the following section 4.4) via the `callbacks` argument, as shown in listing 4.5.

```
# Custom callback that prints a message after the training is finished
class CustomCallback(tf.keras.callbacks.Callback):
    def on_train_end(self, logs=None):
        print("Training is complete, have a nice day!")

my_callbacks = [tf.keras.EarlyStopping(patience=2), CustomCallback()]
model.fit(X_train, y_train, epochs=30, callbacks=my_callbacks)
```

Listing 4.5: A class for a custom callback, and training a Keras model using its `fit()` method while utilizing multiple callbacks, including the custom one.

4.4 Testing and using a model

The `Model`'s `evaluate()` method is used to test/evaluate a model's performance. Its arguments include the testing data, the corresponding class labels, `batch_size`, and a few others, including `callbacks`. There are obviously no epochs, since each iteration would show the same results. The `evaluate()` method, shown in listing 4.6, returns the loss value and metrics for the model in test mode (weights don't get updated), and also prints the relevant information (controlled by the `verbose` argument). Alternatively, the `test_on_batch()` method can be used instead, to test the model on a single batch of data.

```
scores = model.evaluate(X_test, y_test, return_dict=True)
# return_dict=True => metrics are stored in a dictionary instead of a list
print("Test accuracy: ", scores['accuracy'])
```

Listing 4.6: Evaluating a model's accuracy on a testing dataset.

The `Model`'s `predict()` method generates output predictions for given input samples, as shown in listing 4.7. It accepts similar arguments to the `evaluate()` method, except for class labels, since those are returned by the method as a Numpy array(s) of predictions. For a small number of inputs that fit in one batch (marked as `X`), we can also use the `Model`'s `__call__()` method as `predictions = model(X)`, or its `predict_on_batch()` method.

```
predictions = model.predict(X)
# the predicted labels are stored in a list
for i in range(len(predictions)):
    print("Class prediction for ", i+1, ". sample:", predictions[i])
```

Listing 4.7: Using a model to predict class labels for a given set of data.

Apart from custom NN architectures created directly by the user, Keras also offers some pre-built deep learning models, along with pre-trained weights, as Keras Applications, available from the `tensorflow.keras.applications` module. They can be used for prediction, feature extraction, and fine-tuning[10]. These models include famous CNN architectures, such as three different versions of Inception including Xception [5] (Extreme version of Inception), used in the exemplary use case shown in listing 4.8, multiple versions of ResNet [13], many version of EfficientNet [36], and a few others.

```
from tensorflow.keras.applications import xception
from tensorflow.keras.preprocessing.image import load_img, img_to_array
import numpy as np

# loading weights from training on the ImageNet dataset
model = xception.Xception(weights="imagenet", include_top=True)
# loading and preprocessing hamster image
img = load_img('hamster.png', target_size=(299,299))
img = img_to_array(img)
img = np.expand_dims(img, axis=0)
# making model predictions (top=3 => show 3 most probable classes)
x = xception.preprocess_input(img)
preds = model.predict(x)
print("Predicted: ", xception.decode_predictions(preds, top=3)[0])
# Predicted: [('n02342885', 'hamster', 0.9340777),
# ('n03794056', 'mousetrap', 0.0025085167),
# ('n07714990', 'broccoli', 0.002150822)]
```

Listing 4.8: Classifying an image of a hamster by using the Xception [5] NN architecture along with weights pre-trained on the ImageNet[11]dataset

[10]Finetuning means taking weights of a trained neural network and using it as initialization for a new model being trained on data from the same domain. It is primarily used to speed up the training process and overcome small dataset size.

[11]https://www.image-net.org

Chapter 5

Preparation of the experiments

Now that NNs and Keras have been sufficiently introduced, it's time to move on to the practical part of the thesis. The objective of the experiments is to choose 3 different datasets, each suited for a different type of NNs (tabular for MLPs, image for CNNs, sequential for RNNs), create an NN(s) of each type for each of the datasets, compare their performances, and experimentally find out how often can an NN, that's not intuitively suited for a given task at first glance, outperform the one that is.

Since comparing NNs of different types is a bit like comparing apples to oranges, this chapter focuses on preparation of the experiments in such a way, that the different NNs can actually utilize their signature abilities sensibly to their advantage. This heavily depends on the choice of the datasets in section 5.1, preprocessing of the data in section 5.2, which dictates how the data will be interpreted by the individual types of NNs, and finally, the NNs' architectures in section 5.3.

5.1 Choosing and analyzing the datasets

This section introduces the chosen datasets, along with the reasons for choosing them and some brief analysis of their compositions.

5.1.1 Tabular data - MiniBooNE particle identification

MiniBooNE (Booster Neutrino Experiment) is a Cherenkov detector experiment at Fermilab[1] designed to observe neutrino oscillations. The MiniBooNE particle identification dataset had been obtained from the UCI Machine Learning Repository [10] and contains records of 130,064 events (instances), each consisting of 50 individual particle measurements (real numbers), and a binary label that marks whether the instance was a signal event (1) or a background event (0). Since there are only 36,499 (≈ 28 %) signal events, compared to the remaining 93,565 (≈ 72 %) background events, one might call this an imbalanced binary classification problem.

Due to the nature of the experiment the dataset captures, the features (particle measurements) in the scope of a single event are obviously not really independent of each other, so the RNN and 1D–CNN might be able to make use of their unique abilities to identify some hidden patterns the MLPs are unable to see.

[1] `https://www.fnal.gov/`

5.1.2 Image data - Fashion-MNIST

Fashion-MNIST [39] is a dataset consisting of 60,000 training and 10,000 test grayscale images of clothes, each belonging in one of the 10 distinct clothing categories shown in figure 5.1. It was developed as a modern drop-in replacement for the original MNIST dataset, from which it had inherited the same image resolution (28x28), number of target classes, and even the total number of samples and the distribution of target classes. The data was extracted from Zalando's[2] arcticle images and the entire dataset is built into Keras in the exact same manner as the standard MNIST.

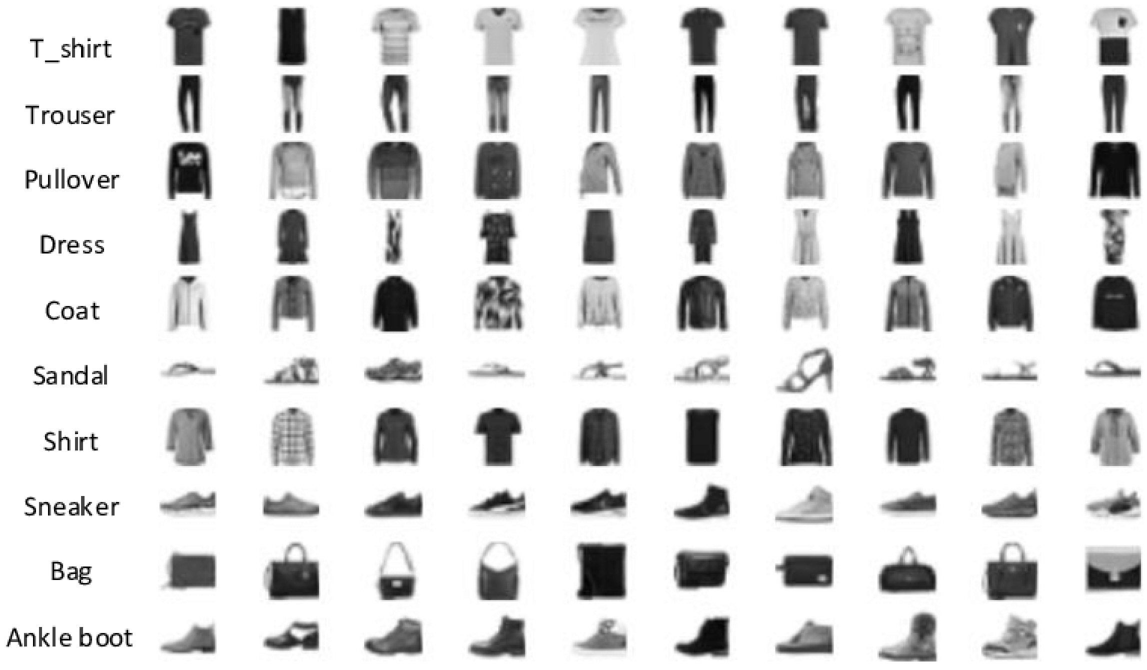


Figure 5.1: Sample images from the Fahion-MNIST dataset [19, Figure 8].

Searching for patterns among the different clothing categories should be undoubtedly more challenging than among digits, so Fashion-MNIST might as well be the better choice for benchmarking NNs built using modern technologies. While MNIST is still often considered to be the go-to introduction dataset for getting into image classification, it was first introduced back in 1998 when the average NN had drastically weaker capabilities than today. The distribution of the target classes has a discrete uniform distribution, so each of the 10 classes makes up exactly 10 % of both the training set and the test set.

5.1.3 Sequential data - IMDB movie review sentiment classification

The IMDB movie review sentiment classification dataset[3] [22] is a set of 25,000 different reviews obtained from the Internet Movie Databased (IMDb[4]), each binary labeled as either positive (1) or negative (0). It is a famous dataset very commonly used for natural language processing or text analytics, that's also built into Keras. A sample review is shown in listing 5.1.

[2]https://github.com/zalandoresearch/fashion-mnist
[3]https://ai.stanford.edu/~amaas/data/sentiment/
[4]https://www.imdb.com/

```
lavish production values and solid performances in this straightforward
adaption of jane [OOV] satirical classic about the marriage game within
and between the classes in [OOV] 18th century england northam and paltrow
are a [OOV] mixture as friends who must pass through [OOV] and lies to
discover that they love each other good humor is a [OOV] virtue which goes
a long way towards explaining the [OOV] of the aged source material which
has been toned down a bit in its harsh [OOV] i liked the look of the film
and how shots were set up and i thought it didn't rely too much on [OOV]
of head shots like most other films of the 80s and 90s do very good results.
```

Listing 5.1: A sample positive review from the IMDB movie review sentiment classification dataset converted into words (OOV is an out-of-vocabulary token representing an unknown word).

There's a total of 50,000 reviews, out of which 25,000 are part of the train set and the other half belongs in the test set. The Keras implementation of this dataset also comes with options like only considering the X most common words, ignoring the Y most common words (e.g. the word 'the'), etc. Exactly half of the reviews are positive (applies to both train/test set) and each review is a sequence of anywhere from 7 to 2,494 (not necessarily distinct) words, with the average being ≈ 234.76 words. Each sentence is a list of indexes (integers). The more often a word is found in the reviews overall, the lower its index in the words' dictionary.

5.2 Data preprocessing

Each dataset has to be preprocessed for each type of NNs in a way that will still be sensible, while also demonstrating the network's unique abilities. This section describes this process for all 9 combinations of tasks and NN types. Most of these adjustments will be done by the networks themselves via preprocessing layers (`Reshape, Flatten,` etc.). The first axis of the input shapes, which indicates the batch size, will not be listed here and is implicitly set to `None` to obtain a variable batch size.

5.2.1 Tabular data - MiniBooNE particle identification

Each sample is a vector of 50 different numerical (floating point) features and has shape `(50)`.

- **MLP**, `input_shape=(num_features)` – Nothing to be done, each sample will have shape `(50)`, to be interpreted as 50 separate numerical features.
- **1D CNN**, `input_shape=(timesteps, num_features)` – Each sample will be reshaped to `(50, 1)`, to be interpreted as 50 timesteps of a single feature, since interpreting samples as a single timestep of 50 features would somewhat degrade the network to an MLP.
- **RNN**, `input_shape=(timesteps, num_features)` – The exact same shape and rationalization behind it as in the 1D CNN case.

5.2.2 Image data - Fashion-MNIST

Each sample is a 2D tensor of shape `(28, 28)` containing integer values between 0 and 255. For all networks, the values will be converted to `float32` and normalized between 0 and 1.

- **MLP**, `input_shape=(num_features)` – Each sample will be reshaped to `(784)`, so that the value of each pixel is interpreted as a separate feature.
- **2D CNN**, `input_shape=(height, width, channels)` – Each sample will be reshaped to `(28, 28, 1)`, to be interpreted as a single-channel (grayscale) image.
- **RNN**, `input_shape=(timesteps, num_features)` – Each sample will be reshaped to `(28, 28, 1)`, so that each row of pixels will be interpreted as a separate feature, whose value changes over 28 timesteps.

5.2.3 Sequential data - IMDB movie review sentiment classification

Each sample is a variable-sequence of integer indexes (words label encoded through the dictionary). For all NN types, only the 10,000 most commonly occurring words will be considered.

- **MLP**, `input_shape=(num_features)` – Each sample will be vectorized into a vector of length 10,000, that's made of binary values indicating whether each word occurs in the given review or not. This leads to the MLP getting less information on input, but may also lead to a surprisingly good performance if there's a strong correlation between occurrences of certain words and the positivity/negativity of the review. The shape of each sample will then be `(10_000)`.
- **1D CNN**, `input_shape=(timesteps, num_features)` – The words will be embedded, which means converting them to a fixed-length (in this case 32) vector of floating point values. The more similar two words are, the lower the euclidean distance between their corresponding vectors. This gives the network much more context to work with than one-hot encoding the words (vectors of length 10,000 comprised of 0s and a single 1). The reviews will be cut-off/padded to only the first 500 words, so the shape of each sample will then be `(500, 32)`.
- **RNN**, `input_shape=(timesteps, num_features)` – The exact same shape and rationalization behind it as in the 1D CNN case. Unlike CNNs, RNNs have the ability to work with sequences of variable lengths, but for the sake of the experimental comparison, they will still be kept at 500 words.

5.3 Choosing the neural network architectures

The obvious problem is that there's no definitive way of labeling two or more NNs of different types as equally powerful, since they're so fundamentally different in the ways they function. Iteratively building NNs by directly measuring their performance and aiming for very similar results would somewhat devaluate the results of the experiments, so I've decided to settle on a different approach—aiming for (approximately) the same number of trainable parameters. This metric is sometimes used to measure an NN's "learning power", since it provides more flexibility in approximating the function that divides the feature space. There are still many factors to consider, as these parameters could be "invested" into parts of the network that have only minor impact on performance, so I've done quite a lot of testing and tweaks before deciding on the final architectures.

There are 4 NNs per experiment—one of each type to compare their core differences, and one extra network best suited for the given task (e.g. CNN for image classification) to also compare different topologies of NNs of the same type. For each experiment, there is a reference number X of trainable parameters obtained by rounding the number of trainable parameters of one network that's used as a reference point, and all the other networks must stay within 10 % error, meaning the interval $(X - \frac{X}{10}, X + \frac{X}{10})$. The NNs were also regularized by including `Dropout` layers with rates tuned for reaching similar training and validation performance (loss and accuracy).

The graphs of all 12 architectures can be found in Appendix B. The naming scheme of the networks is the name of the given data type and the given NN type, separated by an underscore (e.g. Sequence_MLP). The additional network for each experiment is denoted by an extra **x**, which stands for **extra** (e.g. Image_CNNx).

Below are some details about each chosen NN architecture, categorized by the corresponding experiment type. The input layer, and the output layer, which contains either a single neuron with `sigmoid` activation (binary classification – tabular, sequential) or *number of classes* neuron with `softmax` (multiclass classification – Image), are implicit (not mentioned in the architecture details), just as the dropout layers. All models are compiled with `BinaryCrossentropy` or `CategoricalCrossentropy` (in which case the categorical labels are one-hot encoded first) loss, and `Adam` (tabular, image) or `RMSprop` (sequential) optimizer.

5.3.1 Tabular data

This problem is a lot less difficult then the other two, so the NN architectures are rather simple. The reference number of trainable parameters (7,500) had been set just through some experimentation.

- **Tabular_MLP** – A single hidden (`Dense`) layer with 150 neurons and `Relu` activation (also used in all further mentioned `Dense` and `Conv` layers), so the network is expected to learn many simple patterns but struggle in drawing some more complex conclusions.

- **Tabular_MLPx** – The same as Tabular_MLP, but has 3 hidden layers, each with 50 neurons. These two networks were created dependently on each other for a width versus depth comparison of MLPs.

- **Tabular_CNN** – A single `Conv1D` layer with 32 filters, kernel size 3 (each convolution step is based on 3 subsequential timesteps), and stride of 1, also followed by a `MaxPooling1D` layer with pool size of 2. These layers are followed by a `Flatten` layer and a `Dense` layer with 10 neurons, to further reinforce the classification based on the extracted features.

- **Tabular_RNN** – A single `LSTM` layer with 42 units. LSTM had been chosen as the referential building block for all recurrent architectures in the thesis because of its commonness in the context of modern NN architectures.

5.3.2 Image data

The reference number of trainable parameters (240,000) is inherited from the **Image_CNN** and **Image_CNNx** architectures, which have been taken from a Kaggle notebook[5] shared by Gabriel Preda.

- **Image_MLP** – Two `Dense` layers with 224 neurons each, followed by a third `Dense` layer containing 112 neurons. This architecture seems fairly balanced in terms of width and depth of the hidden layers.
- **Image_CNN** – Three subsequent couples of a `Conv2D` and a `MaxPooling2D` layer with a 3x3 kernel size, 2x2 pool size and an increasing number of filters (32, 64, 128), to extract features of an increasing complexity level, followed by a `Flatten` layer and a `Dense` layer with 128 neurons, to strongly reinforce the classification, which is based on the high-level features.
- **Image_CNNx** – Exactly the same as **Image_CNN**, but not regularized at all (no `Dropout` layers). This network was added to demonstrate the need for regularization by direct comparison to the regularized network.
- **Image_RNN** – A `LSTM` layer consisting of 180 units, followed by another one with 90 units and `Dense` layer with 90 neurons. The stacked `LSTM` layers (with the first one set to `return_sequences=True`) allow the network to learn more complex (deeper) recurrent patterns.

5.3.3 Sequential data

The reference number of trainable parameters (370,000), inherited from the **Sequential_RNN** architecture, is the largest out of the three experiments, although a huge part of the "parameter budget" is "invested" either into word embedding via the `Embedding` layer (CNN, RNNs), or having the vectorized sequences at input (MLP).

- **Sequential_MLP** – Only two succeeding `Dense` layers with 36 neurons each. This might seem like very little for such a large "parameter budget", but let's not forget that the input layer is 10,000 neurons wide, so the number of connections adds up quickly.
- **Sequential_CNN** – The `Embedding` layer is followed by a `Conv1D` layer with 64 filters and kernel size of 3, followed by a `GlobalMaxPooling1D` layer and a `Dense` layer with 256 neurons.
- **Sequential_RNN** – The `Embedding` layer is followed by a single `LSTM` layer with 100 units.
- **Sequential_RNNx** – The `Embedding` layer is followed by two `LSTM` layers with 80 and 40 units respectively (the first one is also set to `return_sequences=True`), and a 40-neuron `Dense` layer. This network was created mainly for a depth versus width comparison of RNNs.

[5] https://www.kaggle.com/code/gpreda/cnn-with-tensorflow-keras-for-fashion-mnist

Chapter 6

The experimental application

The main role of the application is to provide a very easy-to-understand (graphical) interface for training and testing the chosen NNs on the corresponding datasets, while still providing enough modularity to enable slightly more experienced users to change the NN architectures and possibly even the datasets, wihout having to dissect the entire source code.

Section 6.1 specifies the functionality that the application should provide, the important implementation details are then noted in section 6.2.

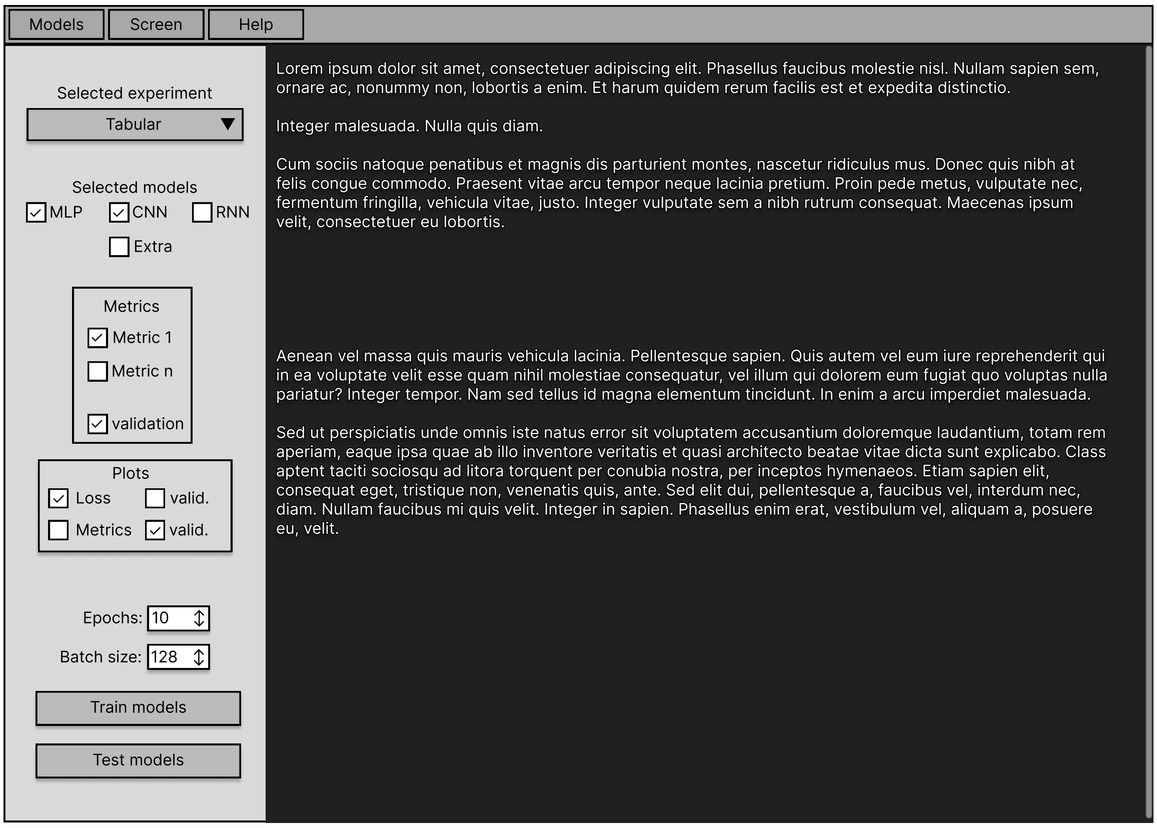
6.1 Concept and requirements

This section is divided into the functional requirements, stating what functionality the application should provide, and a wireframe, the concept of how the graphical user interface should look like (mainly the overall layout).

Functional requirements

- A main (text) screen that shows important information/logs about the train/test progress and feedback for the user interaction.
- Select the active dataset. This also determines the current experiment context.
- Select which of the 4 models are active at the moment. Only the currently selected models are considered for any kind of interaction at the given time.
- Train the selected models for a selected number of epochs with a selected batch size, test the selected models with a selected batch size.
- Select the metrics that should be used for training and testing, while allowing each metric to be also used for validation.
- Plot the progress of the (validation) loss and selected (validation) metrics during training, plot the confusion matrices after testing.
- Save the current weights and load saved weights of the currently selected models, in the context of the currently selected experiment.
- Utilities for smoothing the user experience, like the ability to clear the (text) screen or to save its current state as a text file (log).
- Provide enough modularity for easy NN architecture and dataset changes.

Wireframe



Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Phasellus faucibus molestie nisl. Nullam sapien sem, ornare ac, nonummy non, lobortis a enim. Et harum quidem rerum facilis est et expedita distinctio.

Integer malesuada. Nulla quis diam.

Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quis nibh at felis congue commodo. Praesent vitae arcu tempor neque lacinia pretium. Proin pede metus, vulputate nec, fermentum fringilla, vehicula vitae, justo. Integer vulputate sem a nibh rutrum consequat. Maecenas ipsum velit, consectetuer eu lobortis.

Aenean vel massa quis mauris vehicula lacinia. Pellentesque sapien. Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur? Integer tempor. Nam sed tellus id magna elementum tincidunt. In enim a arcu imperdiet malesuada.

Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Etiam sapien elit, consequat eget, tristique non, venenatis quis, ante. Sed elit dui, pellentesque a, faucibus vel, interdum nec, diam. Nullam faucibus mi quis velit. Integer in sapien. Phasellus enim erat, vestibulum vel, aliquam a, posuere eu, velit.

Figure 6.1: Wireframe of the application's GUI. The wireframe was created in Figma[1].

6.2 Implementation

This section first describes some implementational details of the application logic, and then notes some details about the application's graphical user interface.

Application logic

Each time the user selects an experiment, a new instance of the `Experiment` class is created. Among its attributes is an array of the 4 corresponding models, the data type, and the dataset, which is further represented by an instance of the `Dataset` class, that encapsulates the data split into train and test sets, and holds additional information, like the number of classes and the shape of a sample. This class can be easily modified to exchange one of the datasets for a different one, although some more complex ones, like the IMDB movie review sentiment classification, have to also store additional information (e.g. both the embedded and the one-hot encoded version of the data). The `app/datasets` folder can be used to store external datasets, although by default, it only contains the MiniBooNE particle identification dataset, since the other two are loaded directly from Keras.

[1]https://www.figma.com/

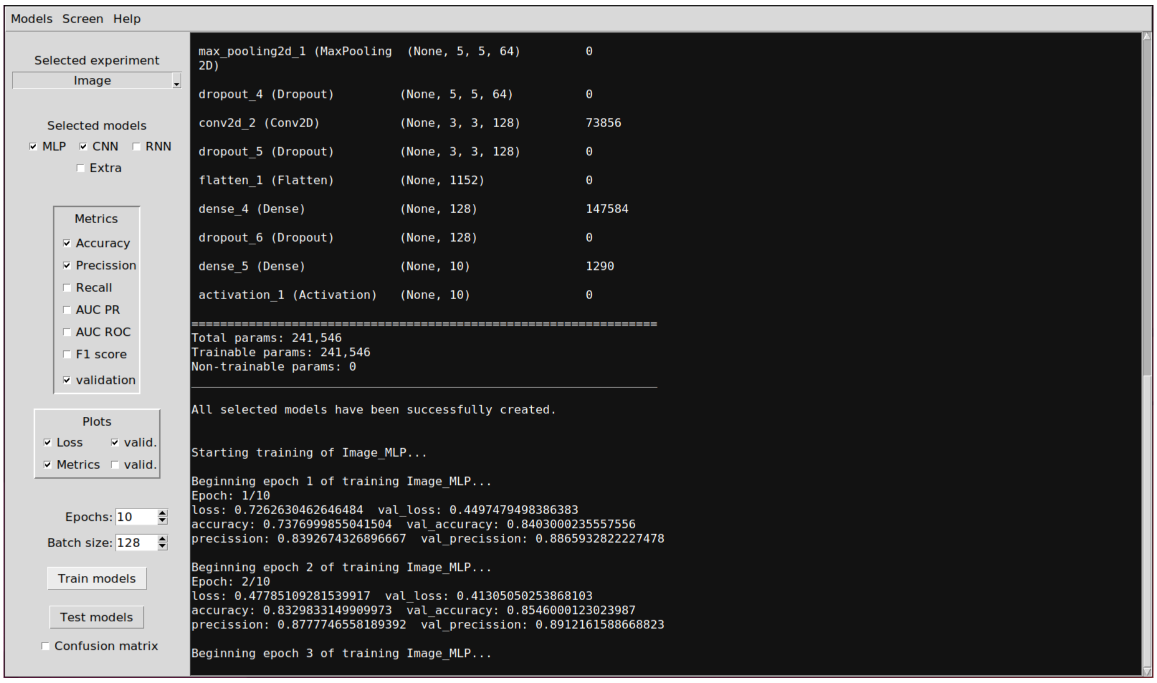
The NN architectures can be changed by modifying the `create_network` function, just remember that reshaping is left to the models themselves by including reshaping layers. There are custom callbacks for training and testing, inheriting from the Keras' `Callback` class, whose purpose is to present selected progress information to the user in a more minimalistic way. Plots of loss and metrics are handled by Matplotlib[2], while Keras handles all the machine learning backend and its other utilities are also used heavily throughout the entire application.

Graphical user interface

The application's graphical user interface (GUI) is built using the Tkinter[3] framework. The entire application is then encapsulated in an instance of the `View` class, whose attributes include even the current `Experiment` instance. All user interaction is handled via callbacks of the GUI's interactive elements and the standard output is redirected to the text screen.

Smoothness of the user experience had been taken very seriously. For example, even though the selected models are created each time a dataset is chosen, along with showing their architecture summaries, when the user selects additional models before beginning training or testing, these models are additionally silently created (no summary shown). Each action with the potential to discard unsaved progress, whether it be saving/loading stored weights or clearing the text screen is guarded by a prompt that informs the user about the possible consequences and gives him the option to cancel the action.

The finished GUI is shown in figure 6.2. The **Models** button at the top opens a drop-down menu for saving and loading stored weights, and the **Screen** button opens a drop-down menu for clearing the screen or saving its current state to a text file.



Selected experiment
Image

Selected models
✓ MLP ✓ CNN ☐ RNN
☐ Extra

Metrics
✓ Accuracy
✓ Precission
☐ Recall
☐ AUC PR
☐ AUC ROC
☐ F1 score
✓ validation

Plots
✓ Loss ✓ valid.
✓ Metrics ☐ valid.

Epochs: 10
Batch size: 128

☐ Confusion matrix

```
 max_pooling2d_1 (MaxPooling  (None, 5, 5, 64)         0
 2D)

 dropout_4 (Dropout)         (None, 5, 5, 64)          0

 conv2d_2 (Conv2D)           (None, 3, 3, 128)         73856

 dropout_5 (Dropout)         (None, 3, 3, 128)         0

 flatten_1 (Flatten)         (None, 1152)              0

 dense_4 (Dense)             (None, 128)               147584

 dropout_6 (Dropout)         (None, 128)               0

 dense_5 (Dense)             (None, 10)                1290

 activation_1 (Activation)   (None, 10)                0

=================================================================
Total params: 241,546
Trainable params: 241,546
Non-trainable params: 0
_________________________________________________________________

All selected models have been successfully created.


Starting training of Image_MLP...

Beginning epoch 1 of training Image_MLP...
Epoch: 1/10
loss: 0.7262630462646484  val_loss: 0.4497479498386383
accuracy: 0.7376999855041504  val_accuracy: 0.8403000235557556
precission: 0.8392674326896667  val_precission: 0.8865932822227478

Beginning epoch 2 of training Image_MLP...
Epoch: 2/10
loss: 0.47785109281539917  val_loss: 0.41305050253868103
accuracy: 0.8329833149909973  val_accuracy: 0.8546000123023987
precission: 0.8777746558189392  val_precission: 0.8912161588668823

Beginning epoch 3 of training Image_MLP...
```

Figure 6.2: The experimental application's GUI created using Tkinter.

[2]https://matplotlib.org/
[3]https://docs.python.org/3/library/tkinter.html

Chapter 7

Course and evaluation of the experiments

The description of each experiment's course starts with a reasoned choice of the batch size, the number of epochs, and the selected metrics. The results are then presented, mainly in the form of generated graphs, and some conclusions are made based on the visible trends and patterns. Each dataset had been only split into train and test subsets, with the test set also being used as the validation set, and validation happening every training epoch, so testing a model at any given time is expected to yield results very similar to the validation phase of its most recent training epoch.

This chapter is split into sections 7.1, 7.2 and 7.3, corresponding to the individual experiments. All measured values are rounded to four (training) or eight (testing) decimal places for the sake of the readability.

7.1 Tabular data classification

This experiment could be considered an anomaly detection problem, since we're more focused on detecting a rare event (less commonly occurring positive class) and less concerned about the nagative cases. For this reason, most of the available metrics (see figure 6.2) had been measured, as they're mostly based on precision and recall, both of which are very relevant for this kind of problem. The only metric to be left out is area under the ROC curve, as ROC curves can sometimes be misleadingly optimistic in imbalanced problems, while misclassifying most of the minority class cases.

A rule of thumb for choosing the number of epochs for training on data with extracted features is to start with the number of features multiplied by 3 (in this case $50 * 3 = 150$). After some experimentation with larger numbers of epochs, I decided to settle on this value as the visible trends didn't really change and the improvement progress stagnated heavily after that point. The batch size had been set to 256, which should be large enough to average out random fluctuations given the size of the dataset.

7.1.1 Results

First, the graphs of the training metrics' progress are shown and described, then the testing phase is presented via the confusion matrices and a table of the measured metrics.

Training phase

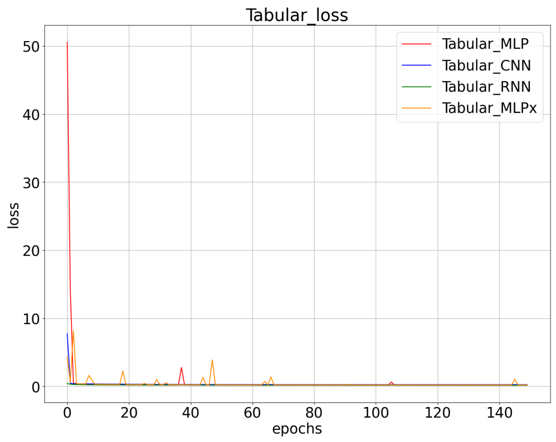


Figure 7.1: Training loss progress.

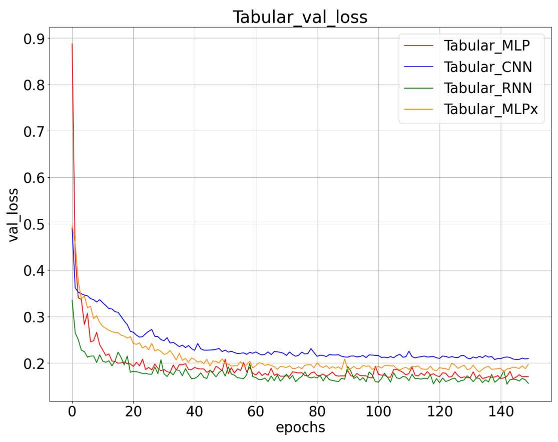


Figure 7.2: Validation loss progress.

Since the graph of training loss (figure 7.1) is very "zoomed out", most of the following information was derived from the training logs. The most prominent progress happened in the first four epochs, where all models reached training loss of about 0.4. `Tabular_CNN` ended up being the worst with 0.2535 loss after the entire training, while the best performing was surprisingly `Tabular_RNN` with loss of 0.1635, closely followed by `Tabular_MLP` (0.1774). Somewhere in the middle ended up `Tabular_MLPx` with 0.2093 loss.

All models seem to generalize very well, as quite unexpectedly, most models' validation loss (figure 7.2) was about 0.4 to 0.8 lower than its training loss. `Tabular_RNN` took this trend even further with an impressive 0.2096 validation loss. The relative performance of the models is consistent in terms of training and validation loss.

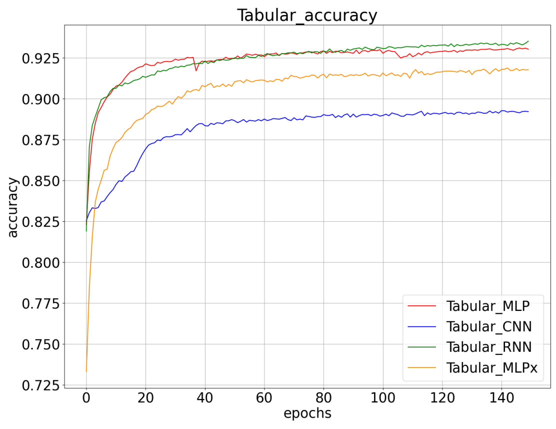


Figure 7.3: Training accuracy progress.

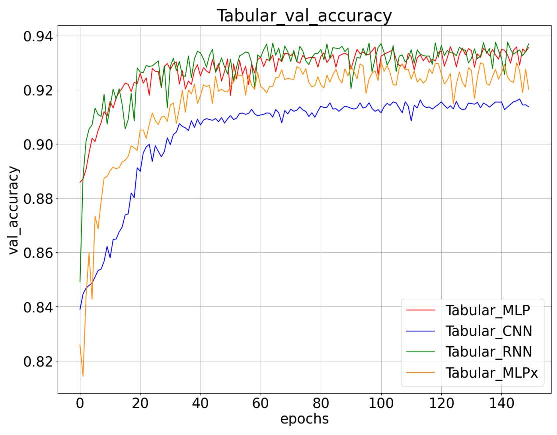


Figure 7.4: Validation accuracy progress.

All models reached at least 0.875 training (figure 7.3) and 0.9 validation (figure 7.4) accuracy in the first 20 epochs, and drastically slowed down their improvement after that point. Even though `Tabular_MLP` performed the best in the beginning, it was, again, closely outperformed by `Tabular_RNN`, which ended up with 0.935 training and 0.9369 validation accuracy. The worst performing was again `Tabular_CNN` with 0.892 training and 0.9137 validation accuracy. Just as was the case with loss, all models showed better validation performance than training performance and the models' relative performance is basically the same.

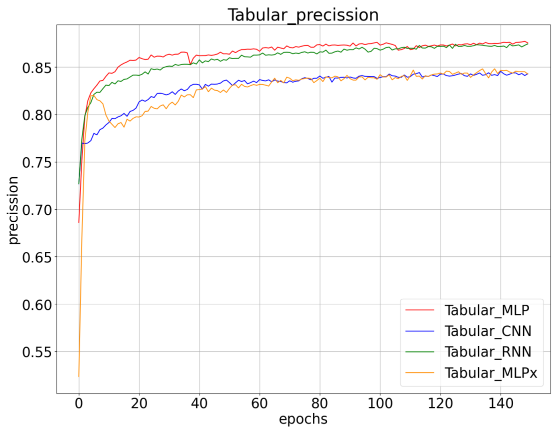


Figure 7.5: Training precision progress.

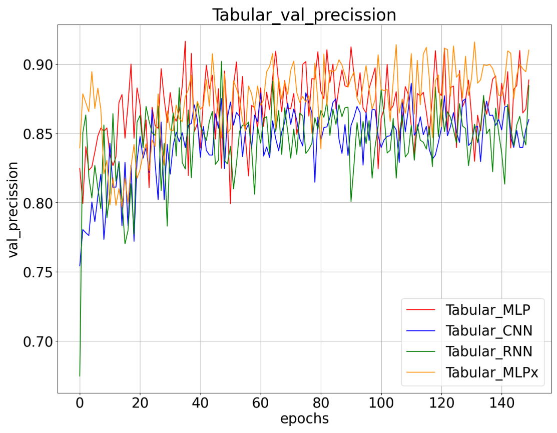


Figure 7.6: Validation precision progress.

In both cases, precision split the models into two pairs. In the case of training precision (figure 7.5), the better pair consists of `Tabular_RNN` and `Tabular_MLP`, which reached 0.8755 precision, while the worse pair converged to a value about 0.035 lower, `Tabular_MLPx` reaching the worst training precision of only 0.8426.

Interestingly enough, in validation precision (figure 7.6, whose graph seems to be quite noisy because of many spikes), `Tabular_MLPx` was, along with `Tabular_MLP`, part of the better performing pair, and the only model to end up with above 0.9 precision, with a value of 0.9102. It also proved itself as the most consistent model, as its spikes didn't dip as low as the ones of all the other models.

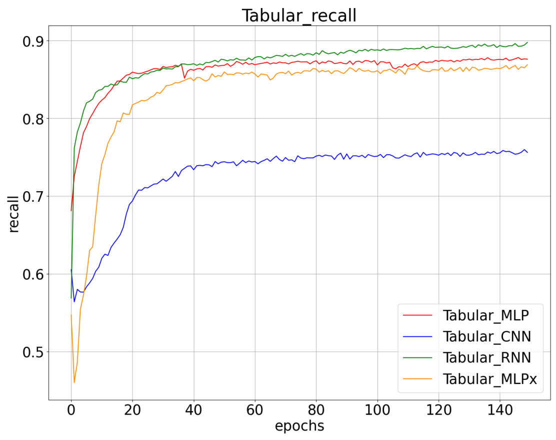


Figure 7.7: Training recall progress.

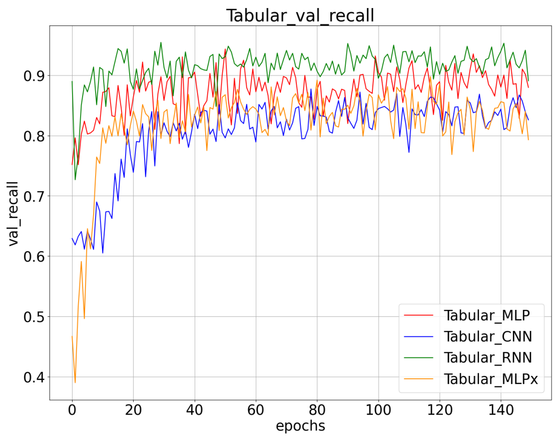


Figure 7.8: Validation recall progress.

Both training (figure 7.7) and validation (figure 7.8) recall were visibly dominated by `Tabular_RNN`, reaching values of 0.8979 and 0.8908 respectively, although the validation recall dipped in the last epoch and peaked at 0.9546 during the 30th epoch. Not too far behind were the MLPs, both reaching about 0.87 ± 0.005 training recall. Horrible performance can be seen from `Tabular_CNN`, which started stagnating at a value of about 0.75, although it tightly outperformed `Tabular_MLPx` in validation, where it also peaked during the 75th epoch at a more "respectable" value of 0.8479.

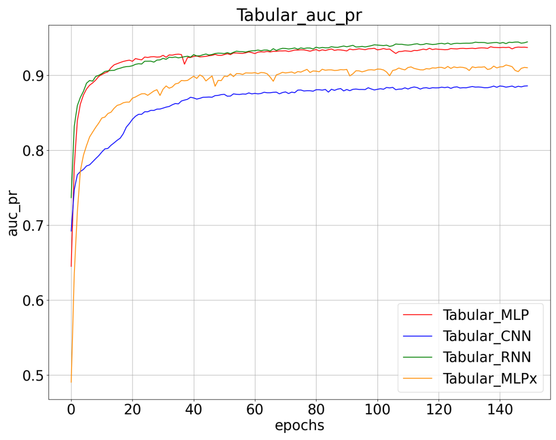


Figure 7.9: Training AUC PR progress.

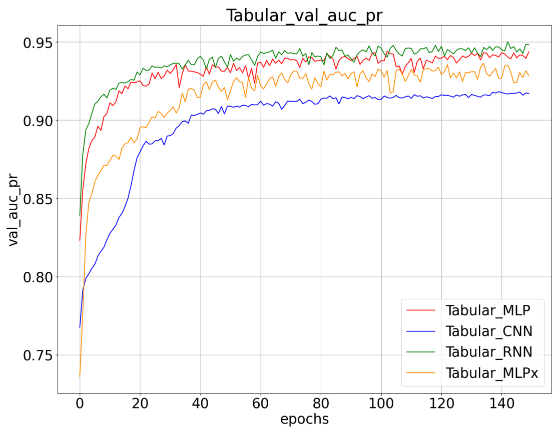


Figure 7.10: Validation AUC PR progress.

The AUC PR (Area unded the precision-recall curve) graphs look very similar to the accuracy graphs. This applies to the AUC PR graph (figure 7.9) compared to figure 7.3, as well as the validation AUC PR graph (figure 7.10) compared to figure 7.4. The best performer, `Tabular_RNN`, reached 0.9442 training and 0.9482 validation AUC PR in the last epoch, while the worst one, `Tabular_CNN`, only reached values of 0.8855 and 0.9171 respectively.

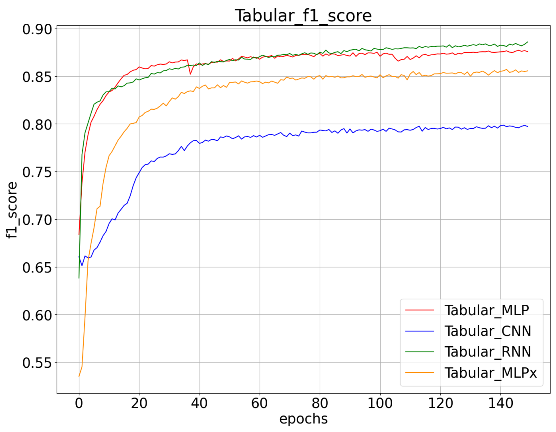


Figure 7.11: Training F_1 score progress.

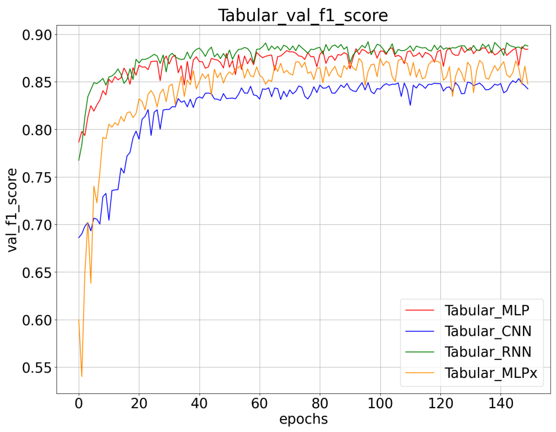


Figure 7.12: Validation F_1 score progress.

Measures of the F_1 score are, again, very similar to both accuracy and AUC PR. The most visible difference is the training F_1 score (figure 7.11) of `Tabular_CNN`, as F_1 score strongly punishes the trade-off between precision and recall, in which `Tabular_CNN` struggled a lot (see figure 7.7). Overall, the highest training F_1 score (0.8859) in the last epoch was, again, obtained by `Tabular_RNN`, while `Tabular_CNN` only reached a value of 0.7973.

Results of the validation F1 score (figure 7.12) seem more balanced, as in the last epoch, `Tabular_RNN` measured 0.8876 and `Tabular_CNN` measured 0.8426, which is not too far behind given the difference of their training F_1 scores.

Testing phase

Please note that this is the only experiment where the test set was resampled from the dataset between training and testing, so it doesn't directly correspond to the validation set.

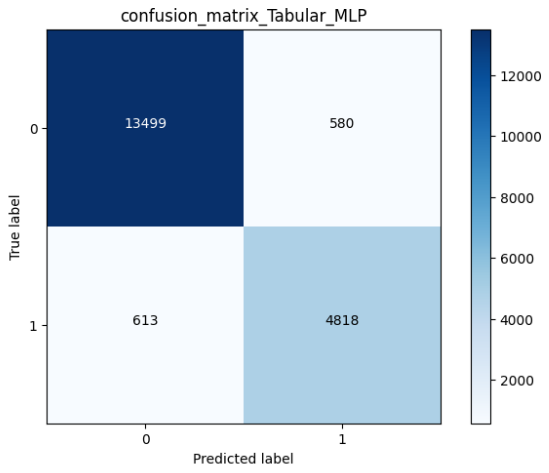


Figure 7.13: Confusion matrix of `Tabular_MLP`.

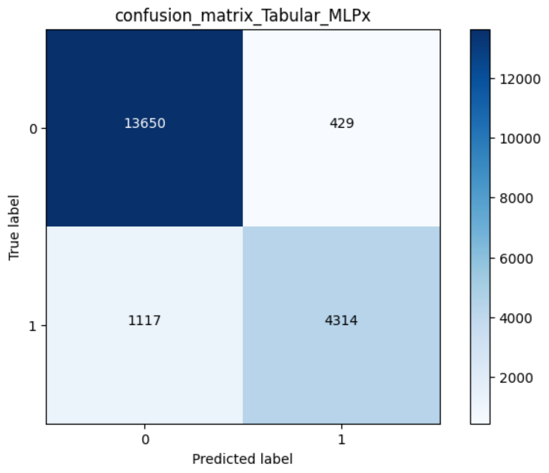


Figure 7.14: Confusion matrix of `Tabular_MLPx`.

The confusion matrices of the two MLPs (figures 7.13 and 7.14) reveal that `Tabular_MLP`, with it's lesser depth and wider layers, was more prone to predicting the less occurring positive label than its deeper counterpart, especially when the actual label was positive. Its positive predictions make up 27.6679 %, while Tabular_MLPx's make up only 24.3106 % of all predicted labels (the actual occurrence of the positive class in the test set was 27.837 %).

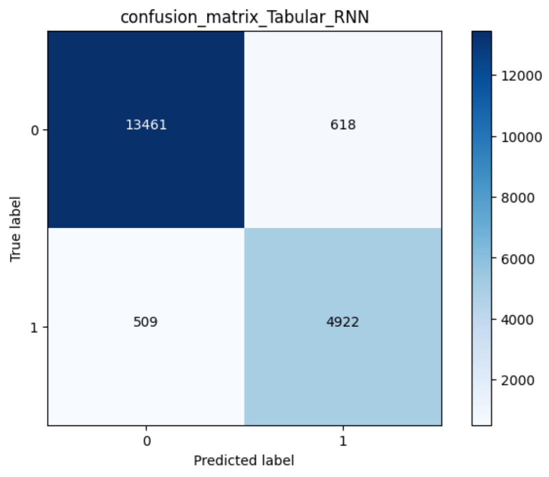


Figure 7.15: Confusion matrix of Tabular_RNN.

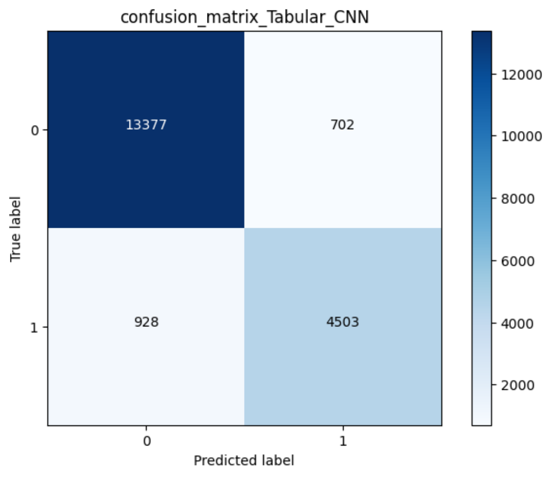


Figure 7.16: Confusion matrix of Tabular_CNN.

As its confusion matrix (figure 7.15) shows, Tabular_RNN improved on Tabular_MLP's amount of TPs by 104, for the cost of only 38 more FPs. Its positive predictions make up 28.3957 %, making it the only overly optimistic model, as well as the best performing.

Even though Tabular_CNN (figure 7.16) counts less FNs than Tabular_MLPx, the count is still unimpressive as well as its FP count, making it the worst overall performing model in terms of rounded predictions, with a 26.6786% positive prediction rate.

Metric	Tabular_MLP	Tabular_CNN	Tabular_RNN	Tabular_MLPx
Loss	0.16712075	0.20510150	0.14320368	0.19452372
Accuracy	0.93885189	0.92765248	0.93251324	0.92957455
Precision	0.89255279	0.87909085	0.88230193	0.88848990
Recall	0.88712943	0.85812926	0.87417912	0.85421652
AUC PR	0.94987130	0.93610001	0.94286144	0.93956620
F1 score	0.88983280	0.86848354	0.87822169	0.87101614

Table 7.1: Loss and metrics values measured during the testing phase of the tabular data experiment.

Table 7.1 shows all values measured in testing. Not that surprisingly, Tabular_MLP beat Tabular_MLPx by every metric (and loss). What's interesting is that while Tabular_RNN measured lower loss than Tabular_MLP, it got slightly outperformed by every single metric. The overall worst performing model seems to be Tabular_CNN, mainly due to its low precision and recall (low AUC PR and F_1 score are direct consequences of that). Even though Tabular_MLPx measured very slightly lower recall, it "redeemed" itself by the measured precision.

7.1.2 Conclusion

The comparison of MLPs with more width (number of neurons in a layer), represented by Tabular_MLP, and those with more depth (more layers), represented by Tabular_MLPx, is pretty straightforward in this case, as `Tabular_MLP` outperformed the latter by almost every measure, although usually not by a too large margin. It can be concluded that at least on this scale and in this kind of problem, an increase in width should precede stacking more layers, that also leads to slower training and unnecessary complexity.

As for the other NN types, `Tabular_RNN` proved itself as a very competent alternative to MLPs for when the features measure the same kind of phenomena, happening just at different time or space points. Not only were its results very impressive, but also consistent, and the model seemed to learn very quickly. The 1D convolutional network, `Tabular_CNN`, performed poorly compared to the other models, but a very possible explanation might be that its bottleneck was the small convolutional kernel size (3), limiting its ability to recognize global patterns, as opposed to the MLPs, working only globally, and `Tabular_RNN`, which was able to store much more context in its recurrent memory. It is very much possible that increasing the kernel size would've had a strong positive impact on the model's performance, as 1D convolutional networks are very commonly used for handling problems of similar nature.

7.2 Image data classification

The only metric to be measured in this balanced multi-class image classification experiment is accuracy, as the other available metrics (see figure 6.2) would have to be measured with respect to a specific class, but in this case, all classes share an equal importance (any important characteristics of the trained models will still be visible in the confusion matrices).

The number of epochs had been set to 200, which is quite a lot, but some interesting changes in the graphs were happening around the 150th epoch. The batch size had been kept at 32, as small mini-batches are very common for computer vision problems.

7.2.1 Results

Again, first the training results and then the testing results are presented.

Training phase

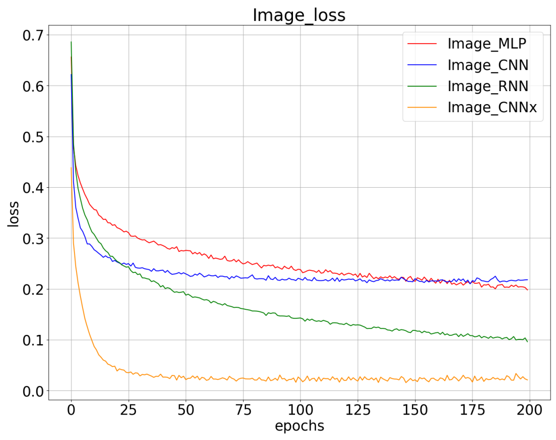


Figure 7.17: Training loss progress.

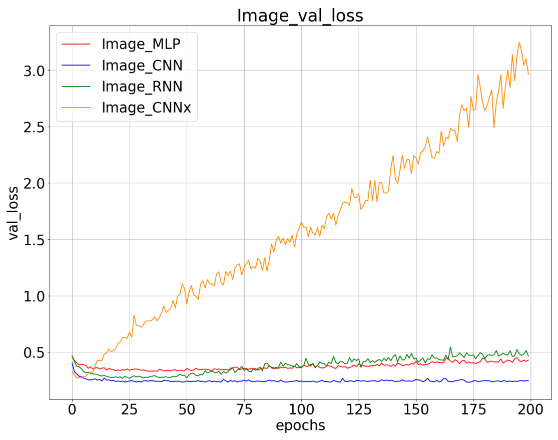


Figure 7.18: Validation loss progress.

First thing to notice in the training (figure 7.17) and validation (figure 7.18) loss graphs is how quickly `Image_CNNx`, the only unregularized model, reached training loss of only about 0.025 (0.021 in the final epoch). This is obviously a case of strong overfitting, as its validation loss was steadily increasing, peaking at 3.246 in the 196th epoch, and would probably keep increasing even more if the training continued. In the contrary, the training loss of `Image_CNN`, which shares the same architecture but heavily regularized, basically converged to values of about 0.215. This paid off well, as its validation loss reached values as small as 0.2326 (49th epoch) very quickly, outperforming the other models effortlessly.

The battle between `Image_RNN` and `Image_MLP` ended up in favor of the former, as in the final epoch, `Image_RNN` reached training loss about twice as low (0.0961 vs 0.198) for the cost of only about 10 % higher validation loss (0.4636 vs 0.4295).

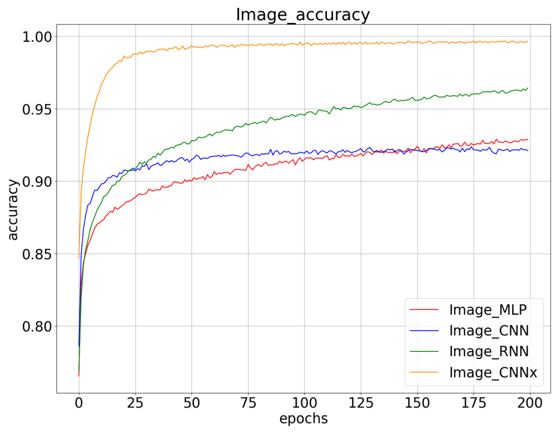


Figure 7.19: Training accuracy progress.

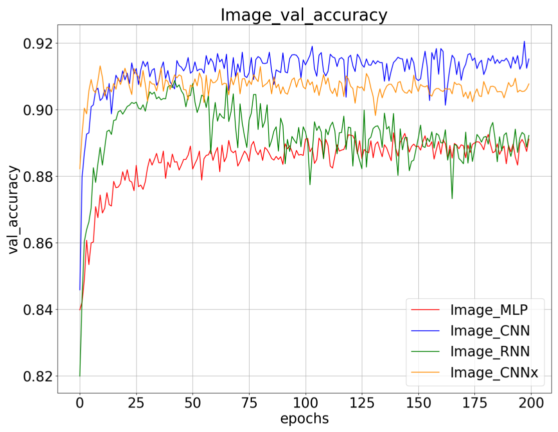
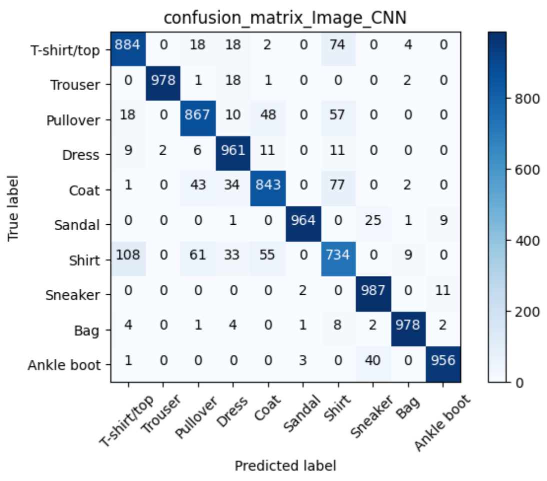


Figure 7.20: Validation accuracy progress.

The unregularized `Image_CNNx` seems to had reached training accuracy (figure 7.19) of around 0.99 (peaking at 0.9968 during the 147th epoch) extremely quickly. Just as was the case with loss, the second most accurate model was `Image_RNN`, with a training accuracy of 0.9643 in the last epoch. The main CNN representative, `Image_CNN`, plateaud very quicky when reaching values of around 0.92, and it even got outperformed by `Image_MLP` at about the 150th epoch and onwards.

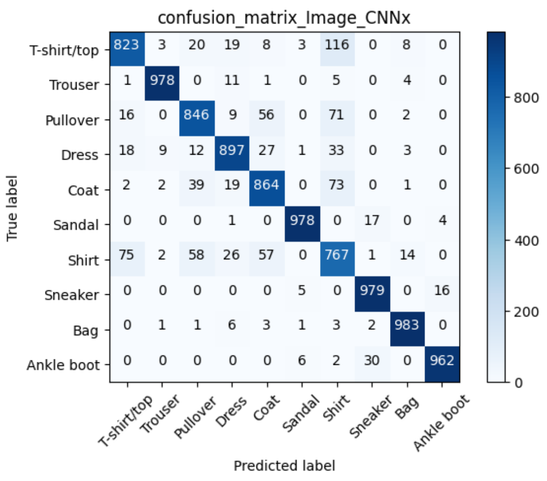
When it comes to validation accuracy (figure 7.20), `Image_CNN` proved itself as the best, although its performance doesn't seem great enough to excuse the poor training performance, given that it may had been intuitively expected to perform the best. It ended up at 0.9152 (last epoch), but `Image_CNNx` was really close behind with a value of 0.9077. This was really unexpected, as one would assume the accuracy would decrease much quicker, complementing the loss increase due to overfitting, but it seemed to be almost unaffected. At about the 45th epoch, `Image_RNN` started showing obvious signs of overfitting, as its validation accuracy went from around 0.905 down to around 0.89, more or less matching `Image_MLP`, which was the worst performing model in terms of accuracy.

Testing phase



True label \ Predicted label	T-shirt/top	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle boot
T-shirt/top	884	0	18	18	2	0	74	0	4	0
Trouser	0	978	1	18	1	0	0	0	2	0
Pullover	18	0	867	10	48	0	57	0	0	0
Dress	9	2	6	961	11	0	11	0	0	0
Coat	1	0	43	34	843	0	77	0	2	0
Sandal	0	0	0	1	0	964	0	25	1	9
Shirt	108	0	61	33	55	0	734	0	9	0
Sneaker	0	0	0	0	0	2	0	987	0	11
Bag	4	0	1	4	0	1	8	2	978	2
Ankle boot	1	0	0	0	0	3	0	40	0	956

Figure 7.21: Confusion matrix of `Image_CNN`.



True label \ Predicted label	T-shirt/top	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle boot
T-shirt/top	823	3	20	19	8	3	116	0	8	0
Trouser	1	978	0	11	1	0	5	0	4	0
Pullover	16	0	846	9	56	0	71	0	2	0
Dress	18	9	12	897	27	1	33	0	3	0
Coat	2	2	39	19	864	0	73	0	1	0
Sandal	0	0	0	1	0	978	0	17	0	4
Shirt	75	2	58	26	57	0	767	1	14	0
Sneaker	0	0	0	0	0	5	0	979	0	16
Bag	0	1	1	6	3	1	3	2	983	0
Ankle boot	0	0	0	0	0	6	2	30	0	962

Figure 7.22: Confusion matrix of `Image_CNNx`.

To help interpret the following confusion matrices (figures 7.21 and 7.22), remember that a perfectly accurate model would have 1,000s on the diagonal and 0s everywhere else, as there are precisely 1,000 samples of each class in the test set.

Even at first glance, it can be seen that both models seem to have problems with classifying shirts. Not only are they (mainly `Image_CNN`) likely to classify a shirt as something similar, like a T-shirt/top or a coat, but they (mainly the more overfitted `Image_CNNx`) also often tend to classify those as a shirt.

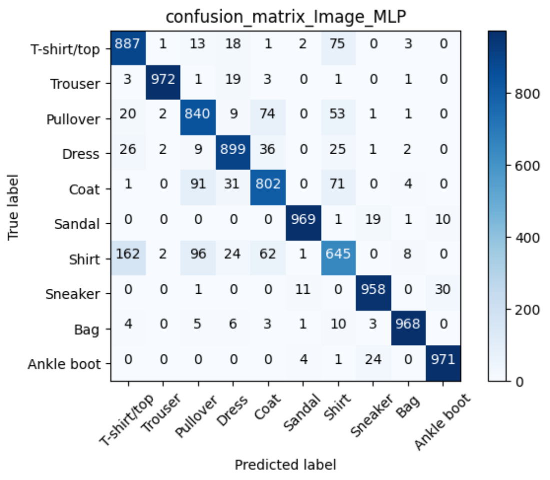


Figure 7.23: Confusion matrix of `Image_MLP`.

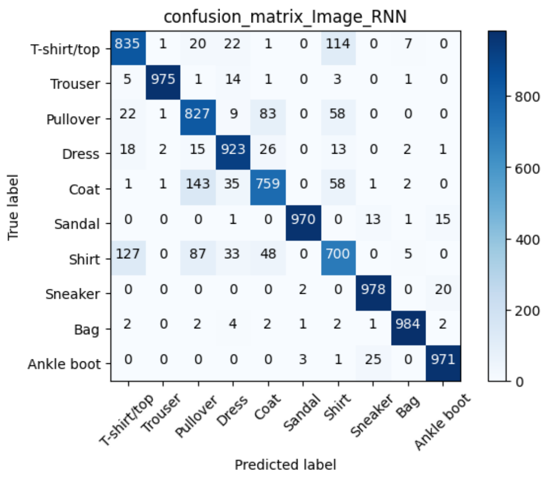


Figure 7.24: Confusion matrix of `Image_RNN`.

Similar pattern can be seen in the confusion matrices of the worse performing models (figures 7.23 and 7.24). Even more prominent than in the previous case is the problem with classifying shirts and coats. `Image_MLP` classified only 645 shirts correctly, predicting 162 of them to be a T-shirt/top, and `Image_RNN` managed to classify 143 coats as a pullover. When it comes to clothes for the lower body, and accessories, the two models performed almost as well as the convolutional models (and were slightly better at classifying ankle boots for some reason).

Metric	`Image_MLP`	`Image_CNN`	`Image_RNN`	`Image_CNNx`
Loss	0.42950425	0.24873139	0.46358573	2.96167660
Accuracy	0.89109999	0.90315002	0.89950001	0.90154999

Table 7.2: Loss and metrics values measured during the testing phase of the image data experiment.

Table 7.2 shows almost nothing new. All models seem to perform just about the same in terms of accuracy, except for `Tabular_MLP`, that scored about 0.01 lower than the rest.

7.2.2 Conclusion

The steadily increasing loss of `Image_CNNx` was very much expected, but its great test accuracy was very surprising. This demonstrates that when the data is split uniformly between classes in the train set and small enough batches are used (smaller batches introduce noise that has a regularizing effect [23]), the negative impacts of overfitting reduce drastically.

I would advise against using MLPs for image classification outside of educational purposes, due to their inability to recognize local patterns. Even though the worst performing model, `Image_MLP`, didn't perform that much more poorly than the rest, keep in mind that the images were very small, grayscale, quite simple, and standardized.

Although not as magnificently as in the tabular experiment (section 7.2), RNNs somewhat proved themselves again as a possible viable alternative, this time to CNNs. But one can wonder how well this scales to more complex problems, as my understanding of `Image_RNN`'s mechanism somewhat reminded me of a barcode scanner, scanning the grayscale image from left to right in parallel with respect to the rows of pixels (but please take this statement with a grain of salt). One also has to decide between interpreting rows of pixels as features and columns as timesteps, or the other way around—this is also dependent on the given problem and should be expected to impact the model's performance.

7.3 Sequential data classification

Again, the only metric to be measured in this experiment is accuracy, as for the other available metrics (see figure 6.2), the positive class would have to be the main point of interest (or the class labels could get flipped to relate the metrics to the negative class), but I figured it might be a better idea to consider positive and negative movie reviews equally (both the train/test sets are also perfectly balanced in terms of class distribution).

Given that training these models (especially the RNNs) was more time demanding than the previous experiments, the training was done over 100 epochs, which should yield results scaled similarly to the previous experiments. The batch size had been set to 64—an optimal batch size for training LSTM-based models in most cases [12].

7.3.1 Results

Once again, first the training results and then the testing results are presented.

Training phase

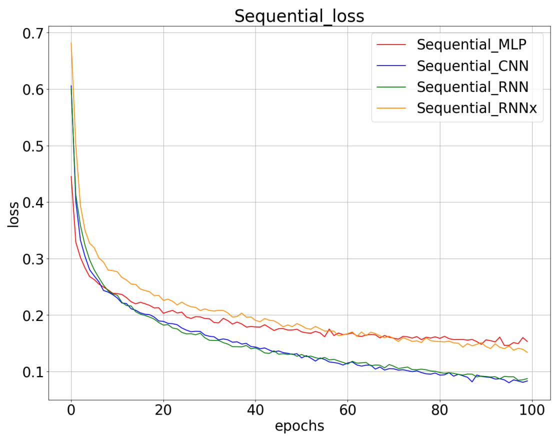


Figure 7.25: Training loss progress.

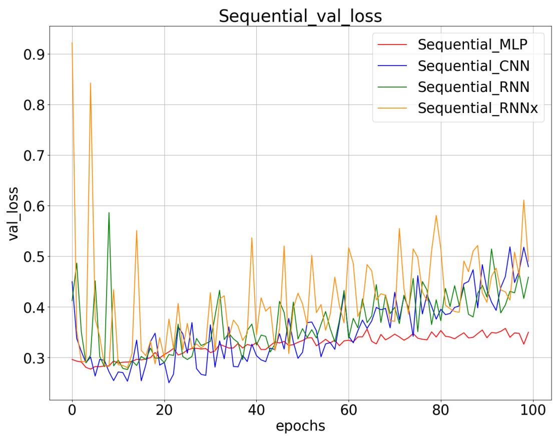


Figure 7.26: Validation loss progress.

It is clearly visible from both training loss (figure 7.25) and validation loss (figure 7.26) graphs, that the more complicated `Sequential_RNNx` performed objectively worse than `Sequential_RNN`. Not only did its loss decrease slow down quite early, but there also seems to be more prominent and unstable overfitting (based on the validation loss).

`Image_CNN`'s performance seems to be just as good as `Sequential_RNN`'s in both cases. It measured 0.0834 loss in the last epoch, a bit lower than `Sequential_RNN`'s 0.0877, but its validation loss was hovering around slightly higher values during the last few epochs. Very surprising was the training performance of `Sequential_MLP`, as it performed similarly to `Sequential_RNNx` in terms of loss, and much better that all the other models in terms of validation loss (0.35 during the last epoch, other models measured between 0.45 and 0.49).

This is very impressive, since it only had limited training information available (vectorized words) compared to the other models (embedded words).

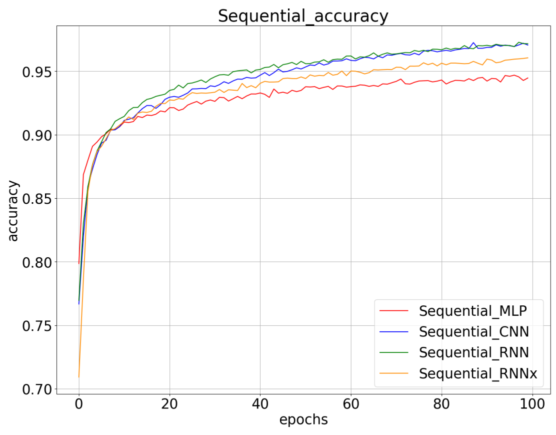


Figure 7.27: Training accuracy progress.

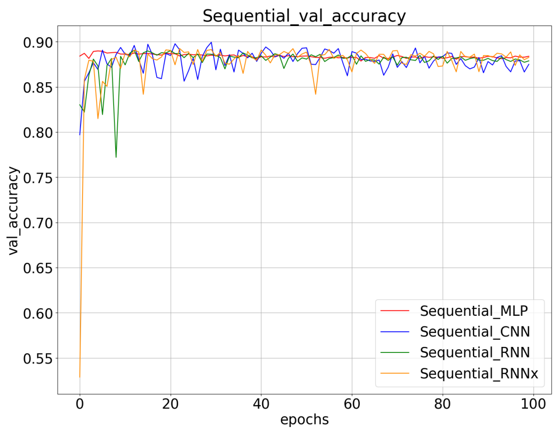


Figure 7.28: Validation accuracy progress.

In terms of training accuracy (figure 7.27), the best performing (0.9716, 100th epoch) was `Sequential_RNN`, but `Sequential_CNN` kept staying extremely close (0.9706) and based on the curves, it seems that it maybe could have outran `Sequential_CNN` if the training continued. The worst training accuracy was measured by `Sequential_MLP` (0.9446, 100th epoch), but it was still quite good.

All models performed quite similarly in terms of validation accuracy (figure 7.28), although `Sequential_RNN`'s and `Sequential_MLP`'s graphs are more consistent (less spikes). During the last epoch, all models measured between 0.875 (`Sequential_CNN`) and 0.8839 (`Sequential_MLP`) validation accuracy.

Testing phase

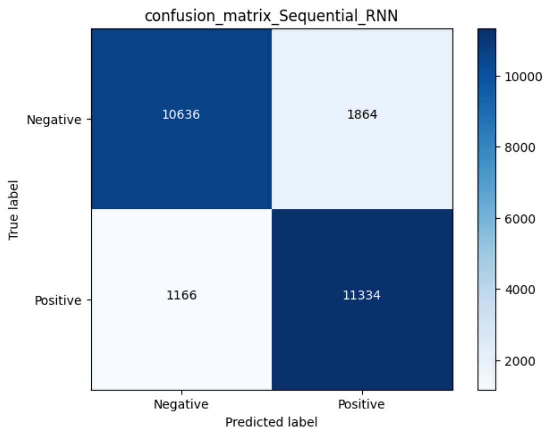


Figure 7.29: Confusion matrix of `Sequential_RNN`.

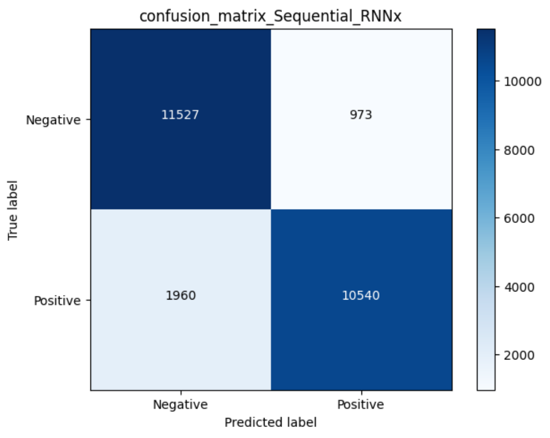


Figure 7.30: Confusion matrix of `Sequential_RNNx`.

Judging from the confusion matrices, it seems that the recurrent models have developed opposite biases. While `Sequential_RNN` (figure 7.29), having only one LSTM-based layer, is more prone to misclassifying a negative review as positive, the deeper (stacked)

`Sequential_RNNx` (7.30) seems to have an even stronger bias, but this time towards false negatives.

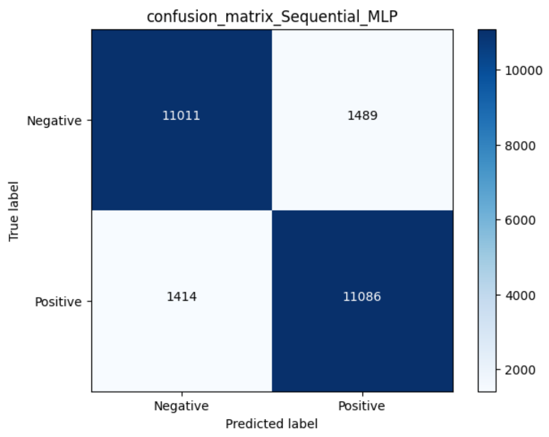


Figure 7.31: Confusion matrix of `Sequential_MLP`.

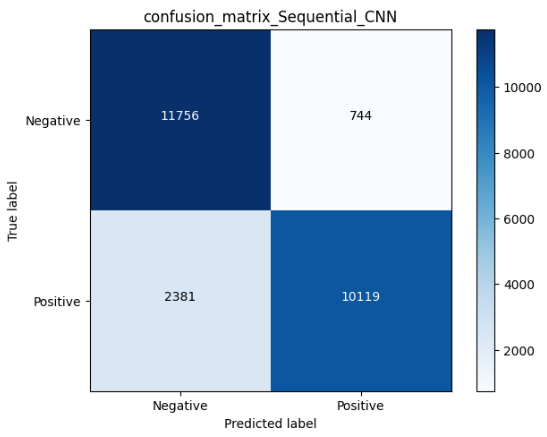


Figure 7.32: Confusion matrix of `Sequential_CNN`.

Since the only information available for `Sequential_MLP` (figure 7.31) is the occurrence (0/1) of each word, I really expected it to strongly associate certain words with a certain class and build up a strong bias. The opposite is actually true, as it is surprisingly the least biased model.

On the contrary, `Sequential_CNN` (figure 7.32) is more than three times as likely to predict a false negative than to predict a false positive, making it the most biased model of the experiment.

Metric	`Sequential_MLP`	`Sequential_CNN`	`Sequential_RNN`	`Sequential_RNNx`
Loss	0.34997347	0.47943094	0.45871839	0.48976046
Accuracy	0.88388002	0.87944001	0.87922668	0.88009000

Table 7.3: Loss and metrics values measured during the testing phase of the sequential data experiment.

Table 7.3 confirms that the performance of `Sequential_RNN` and `Sequential_CNN` is roughly the same. `Sequential_RNNx` has worse test performance than `Sequential_MLP`, even though it's training performance was not marginally better, which is quite bad for a recurrent model considering the nature of the experiment (natural language processing).

7.3.2 Conclusion

The convolutional model, `Sequential_CNN`, managed to reach the same levels of performance as the better one of the recurrent models (`Sequential_RNN`), while only having convolutional kernels of size 3. It should also be mentioned that none of the recurrent models had recurrent dropout (setting some of the “remembered” context to zero, instead of the input), as that massively slowed down their training when I experimented with it, so I decided to only use standard dropouts with very high rates, that could have possibly negatively impacted the performance of both models.

Comparing the two recurrent models, there was really no criteria by which the stacked `Sequential_RNNx` would beat the “simpler” model, so I assume that unless the problem is quite complex, adding additional layers may have a negative impact, while simply adding more recurrent cells usually shouldn't cause much harm and should be preferred.

What attracts the most attention is the performance of `Sequential_MLP`. It demonstrated that using less information might be even beneficial in cases where the ocurrence of certain tokens (words) strongly characterizes the classes. Word embedding could have been used for training this model as well, in which case each word-vector would be simply interpreted as 32 individual features, but the input layer would be massively wide and I also wanted to try something different, which, I suppose, actually paid off.

7.4 Summary

It is crucial to realize that comparing the models based solely on performance shouldn't be the only focus, as there are just too many factors that can influence the results. The models were regularized heavily to generalize well, but for example, once model A shows better training performance, but worse validation/test performance than model B, it can imply that model B was just regularized a bit more. One should also focus on the visible characteristics of the models shown throughout the experiments.

What the experiments demonstrated very well is that when presented with a problem, one should not simply choose the "canonical" NN type suited for the problem, but also consider the alternatives based on analyzing the problem first. The best examples seem to be `Tabular_RNN` from the first experiment and `Sequential_CNN` from the third experiment. The third experiment also demonstrated (through `Sequential_MLP`'s performance), that sometimes, it might be worth to consider a simplification of the data's interpretation, as it might be even beneficial for dealing with some shortcomings (the class bias in this case).

In summary, usually all models performed fairly well regardless of the type of the problem (I would consider `Tabular_CNN` in the first problem as the worst exception), but keep in mind that the problems were rather simple, so this trend is not guaranteed to transfer to marginally larger scales of most modern practical problems, that are often solved by models with complex architectures and billions of trainable parameters.

Chapter 8

Conclusion

The first goal of the thesis was to introduce classification and artificial neural networks, preferably to a reader with no prior knowledge of the problematics. The important classification terminology and metrics, understanding of which is crucial for understanding the rest of the work, were explained clearly, and also some other common classification algorithms were briefly introduced. Artificial neural networks were explained very gradually, starting from a single neuron, all the way to the three main ANN types. Modern architectural advancements (e.g. transformers) were either left out completely or just very briefly mentioned, to really focus on the understanding of the fundamentals.

The second goal was to introduce Keras in the context of classification with ANNs. The given section was very practically focused, and ordered in the same way a user would approach when building a classification model. The code snippets often purposely demonstrated different ways of performing a certain action, and while some parts of the text might seem overfilled with Keras' diverse options (loss, metrics, etc.), those options were introduced practically, to guide the reader's choices when building his own models.

The third and last goal was to perform experiments showing the differences between the three main ANN types. Since the original ideas for the experiments were lacking in many aspects, there was a need for a way to compare all three ANN types on a problem, that unbiasedly favors none of them. This was solved by performing an individual experiment for each of the corresponding data types, and the requirement for comparing different ANN topologies was met by always introducing an additional ANN of the given type. The preparation of the experiments was crucial in this case, and all choices, including selecting the number of trainable parameters as the performance-similarity benchmark, were backed up by an explanation. The experimental application is rather simple and focused mainly on its main purpose, but also provides many utilities for a smooth practical usage.

Even though a straight performance comparison of different ANN types is almost impossible, since there are simply too many factors having possibly detrimental effects on the results to consider, the experiments demonstrated the usefulness of the individual ANN types on problems where they may have not been considered normally. Some interesting, unexpected findings emerged throughout the course of the experiments too.

A very appropriate way of extending this work might be a continuation focusing on modern ANN architectures and even on specific models, possibly with an emphasis on generative models, which have been rapidly gaining popularity by the time of finishing this thesis. This could also incorporate inventing some creative ways of comparing such models. Some other possibilities include comparing different classification algorithms (not only ANNs) or extending the work to a different problem domain (e.g. regression), but today's circumstances create many great opportunities favoring the first approach.

Bibliography

[1] Alom, M. Z., Taha, T., Yakopcic, C., Westberg, S., Sidike, P. et al. A State-of-the-Art Survey on Deep Learning Theory and Architectures. *Electronics*. march 2019, vol. 8, p. 292. DOI: 10.3390/electronics8030292.

[2] Barbosa, A. *Area under the precision-recall curve* [online]. Jan 2020 [cit. 2022-06-09]. Available at: https://modtools.wordpress.com/2020/01/17/area-under-the-precision-recall-curve/.

[3] Cho, K., Merrienboer, B. van, Bahdanau, D. and Bengio, Y. *On the Properties of Neural Machine Translation: Encoder-Decoder Approaches*. arXiv, 2014. DOI: 10.48550/ARXIV.1409.1259. Available at: https://arxiv.org/abs/1409.1259.

[4] Chollet, F. et al. *Keras: the Python Deep Learning API* [online]. 2021 [cit. 2022-05-03]. Available at: https://keras.io.

[5] Chollet, F. Xception: Deep Learning with Depthwise Separable Convolutions. *CoRR*. 2016, abs/1610.02357. Available at: http://arxiv.org/abs/1610.02357.

[6] Chollet, F. *Deep learning with Python*. 1st ed. Shelter Island, NY: Manning, 2018. ISBN 978-1-61729-443-3.

[7] Cybenko, G. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*. Springer. 1989, vol. 2, no. 4, p. 303–314.

[8] Dalecký Š. *Neuro-fuzzy systémy*. Brno, CZ, 2014. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/16598/.

[9] Deng, L. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*. IEEE. 2012, vol. 29, no. 6, p. 141–142.

[10] Dua, D. and Graff, C. *UCI Machine Learning Repository*. 2017. Available at: http://archive.ics.uci.edu/ml.

[11] Fiesler, E. and Beale, R. Neural network topologies. *The Handbook of Neural Computation, E. Fiesler and R. Beale (Editors-in-Chief), Oxford University Press and IOP Publishing*. Citeseer. 1996.

[12] Gures, C. *Selecting Optimal LSTM Batch Size* [online]. Medium, Mar 2020 [cit. 2023-04-05]. Available at: https://medium.com/@canerkilinc/selecting-optimal-lstm-batch-size-63066d88b96b.

[13] He, K., Zhang, X., Ren, S. and Sun, J. Deep Residual Learning for Image Recognition. *CoRR*. 2015, abs/1512.03385. Available at: http://arxiv.org/abs/1512.03385.

[14] Hochreiter, S. Untersuchungen zu dynamischen neuronalen Netzen. *Diploma, Technische Universität München*. 1991, vol. 91, no. 1.

[15] Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural computation*. MIT Press. 1997, vol. 9, no. 8, p. 1735–1780.

[16] Hossin, M. and Sulaiman, M. N. A review on evaluation metrics for data classification evaluations. *International journal of data mining & knowledge management process*. Academy & Industry Research Collaboration Center (AIRCC). 2015, vol. 5, no. 2, p. 1.

[17] IBM Cloud Education . *Overfitting* [online]. March 2021 [cit. 2022-05-13]. Available at: https://www.ibm.com/cloud/learn/overfitting.

[18] Isik, F., Ozden, G. and Kuntalp, M. Importance of data preprocessing for neural networks modeling: The case of estimating the compaction parameters of soils. *Energy Educ Sci Technol Part A: Energy Sci Res*. april 2012, vol. 29, p. 463–474.

[19] Kheradpisheh, S. R., Mirsadeghi, M. and Masquelier, T. BS4NN: Binarized Spiking Neural Networks with Temporal Coding and Learning. *CoRR*. 2020, abs/2007.04039. Available at: https://arxiv.org/abs/2007.04039.

[20] Li, S. *Rotation Invariance Neural Network*. arXiv, 2017. DOI: 10.48550/ARXIV.1706.05534. Available at: https://arxiv.org/abs/1706.05534.

[21] Lopez, D. *RNN, LSTM & GRU: Recurrent Neural Network (RNN), Long-Short Term Memory (LSTM) & Gated Recurrent Unit (GRU)* [online]. dprogrammer, april 2019 [cit. 2022-05-23]. Available at: http://dprogrammer.org/rnn-lstm-gru.

[22] Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y. et al. Learning Word Vectors for Sentiment Analysis. In: *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Portland, Oregon, USA: Association for Computational Linguistics, June 2011, p. 142–150. Available at: http://www.aclweb.org/anthology/P11-1015.

[23] Masters, D. and Luschi, C. Revisiting Small Batch Training for Deep Neural Networks. *CoRR*. 2018, abs/1804.07612, p. 1. Available at: http://arxiv.org/abs/1804.07612.

[24] Neutelings, I. *Neural networks* [online]. TikZ.net, april 2022 [cit. 2022-05-01]. Available at: https://tikz.net/neural_networks/.

[25] O'Shea, K. and Nash, R. *An Introduction to Convolutional Neural Networks*. arXiv, 2015. DOI: 10.48550/ARXIV.1511.08458. Available at: https://arxiv.org/abs/1511.08458.

[26] Quinlan, J. R. Induction of decision trees. *Machine learning*. Springer. 1986, vol. 1, no. 1, p. 81–106.

[27] Rosenblatt, F. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review.* American Psychological Association. 1958, vol. 65, no. 6, p. 386.

[28] Ruder, S. *An overview of gradient descent optimization algorithms.* arXiv, 2016. DOI: 10.48550/ARXIV.1609.04747. Available at: https://arxiv.org/abs/1609.04747.

[29] Rumelhart, D. E., Hinton, G. E. and Williams, R. J. Learning representations by back-propagating errors. *Nature.* Nature Publishing Group. 1986, vol. 323, no. 6088, p. 533–536.

[30] Sahu, V. *Power of a Single Neuron* [online]. Medium, june 2018 [cit. 2022-05-06]. Available at: https://towardsdatascience.com/power-of-a-single-neuron-perceptron-c418ba445095.

[31] Sayad, S. *Classification* [online]. Dr. Saed Sayad, may 2015 [cit. 2022-04-07]. Available at: https://www.saedsayad.com/classification.htm.

[32] Shulga, D., Morozov, O., Roth, V., Friedrich, F. and Hunziker, P. Tensor B-Spline Numerical Methods for PDEs: a High-Performance Alternative to FEM. arXiv. 2019. DOI: 10.48550/ARXIV.1904.03057. Available at: https://arxiv.org/abs/1904.03057.

[33] Slávka, M. *Playing Gomoku with Neural Networks.* Brno, CZ, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/21764/.

[34] Sofaer, H. R., Hoeting, J. A. and Jarnevich, C. S. The area under the precision-recall curve as a performance metric for rare binary events. *Methods in Ecology and Evolution.* Wiley Online Library. 2019, vol. 10, no. 4, p. 565–577.

[35] Suzuki, K. *Artificial Neural Networks.* Rijeka: IntechOpen, april 2011. Available at: https://doi.org/10.5772/644.

[36] Tan, M. and Le, Q. V. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. *CoRR.* 2019, abs/1905.11946. Available at: http://arxiv.org/abs/1905.11946.

[37] Veličković, P. *TikZ/2D Convolution* [online]. GitHub, 2016. Available at: https://github.com/PetarV-/TikZ/tree/master/2D%20Convolution.

[38] Waseem, M. *How To Implement Classification In Machine Learning?* [online]. Edureka, march 2022 [cit. 2022-04-04]. Available at: https://www.edureka.co/blog/classification-in-machine-learning/.

[39] Xiao, H., Rasul, K. and Vollgraf, R. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. *CoRR.* 2017, abs/1708.07747. Available at: http://arxiv.org/abs/1708.07747.

Appendix A

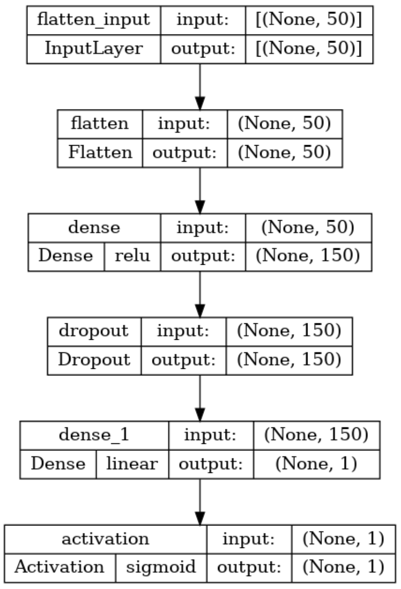
Contents of the included storage media

- **`/app/`** – The experimental application (source code, datasets, working directories, instructions).
- **`/experiments_results/`** – Results of the experiments generated by the experimental application.
- **`/model_plots/`** – Model plots of the neural network architectures used in the experiments.
- **`/text/`** – LaTeXsource codes of this thesis.
- **`xpysik00-thesis.pdf`** – This thesis in PDF format.

Appendix B

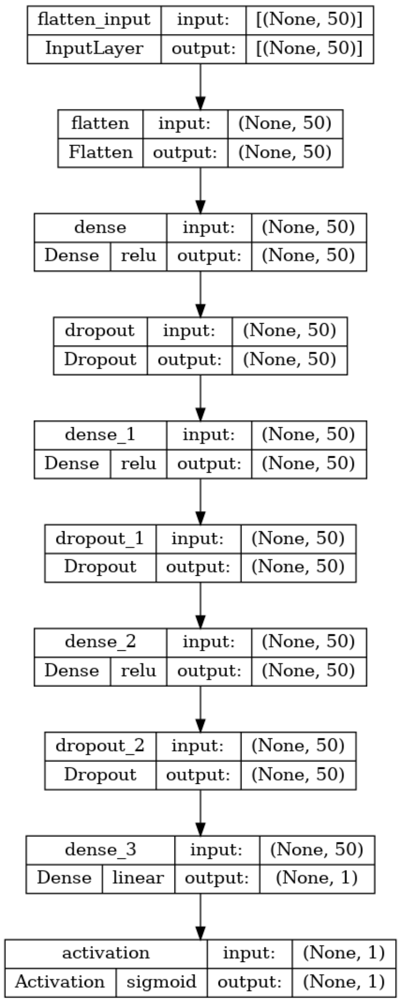
Model plots of the used neural network architectures

B.1 Tabular data - MiniBooNE particle identification



flatten_input | input: | [(None, 50)]
InputLayer | output: | [(None, 50)]
flatten | input: | (None, 50)
Flatten | output: | (None, 50)
dense | input: | (None, 50)
Dense | relu | output: | (None, 150)
dropout | input: | (None, 150)
Dropout | output: | (None, 150)
dense_1 | input: | (None, 150)
Dense | linear | output: | (None, 1)
activation | input: | (None, 1)
Activation | sigmoid | output: | (None, 1)

Figure B.1: Tabular_MLP



flatten_input | input: | [(None, 50)]
InputLayer | output: | [(None, 50)]
flatten | input: | (None, 50)
Flatten | output: | (None, 50)
dense | input: | (None, 50)
Dense | relu | output: | (None, 50)
dropout | input: | (None, 50)
Dropout | output: | (None, 50)
dense_1 | input: | (None, 50)
Dense | relu | output: | (None, 50)
dropout_1 | input: | (None, 50)
Dropout | output: | (None, 50)
dense_2 | input: | (None, 50)
Dense | relu | output: | (None, 50)
dropout_2 | input: | (None, 50)
Dropout | output: | (None, 50)
dense_3 | input: | (None, 50)
Dense | linear | output: | (None, 1)
activation | input: | (None, 1)
Activation | sigmoid | output: | (None, 1)

Figure B.2: Tabular_MLPx

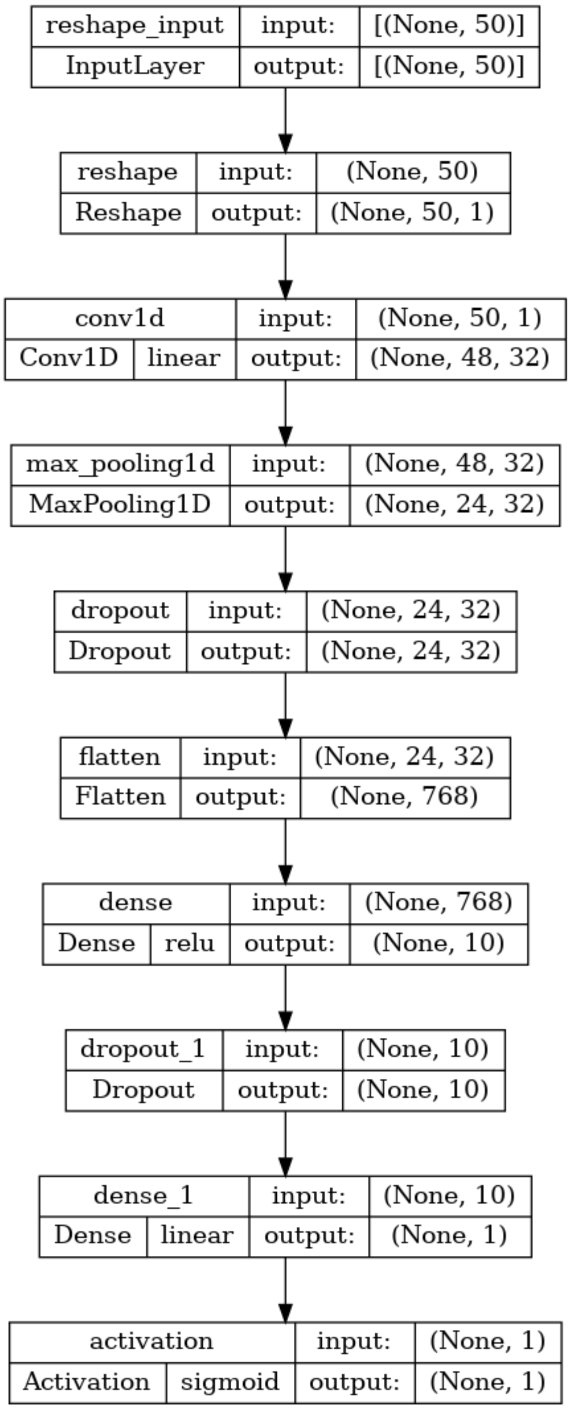

reshape_input | input: | [(None, 50)]
InputLayer | output: | [(None, 50)]
reshape | input: | (None, 50)
Reshape | output: | (None, 50, 1)
conv1d | input: | (None, 50, 1)
Conv1D | linear | output: | (None, 48, 32)
max_pooling1d | input: | (None, 48, 32)
MaxPooling1D | output: | (None, 24, 32)
dropout | input: | (None, 24, 32)
Dropout | output: | (None, 24, 32)
flatten | input: | (None, 24, 32)
Flatten | output: | (None, 768)
dense | input: | (None, 768)
Dense | relu | output: | (None, 10)
dropout_1 | input: | (None, 10)
Dropout | output: | (None, 10)
dense_1 | input: | (None, 10)
Dense | linear | output: | (None, 1)
activation | input: | (None, 1)
Activation | sigmoid | output: | (None, 1)

Figure B.3: Tabular_CNN

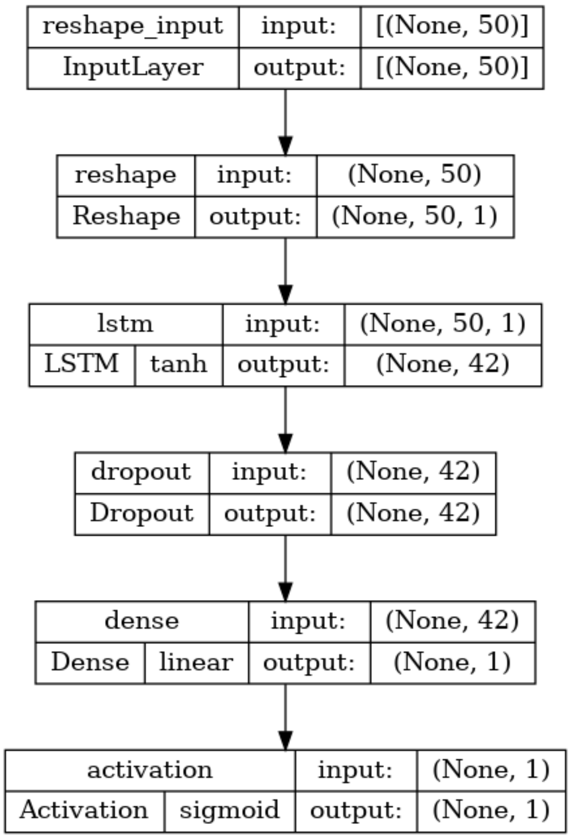

reshape_input | input: | [(None, 50)]
InputLayer | output: | [(None, 50)]
reshape | input: | (None, 50)
Reshape | output: | (None, 50, 1)
lstm | input: | (None, 50, 1)
LSTM | tanh | output: | (None, 42)
dropout | input: | (None, 42)
Dropout | output: | (None, 42)
dense | input: | (None, 42)
Dense | linear | output: | (None, 1)
activation | input: | (None, 1)
Activation | sigmoid | output: | (None, 1)

Figure B.4: Tabular_RNN

B.2 Image data - Fashion-MNIST

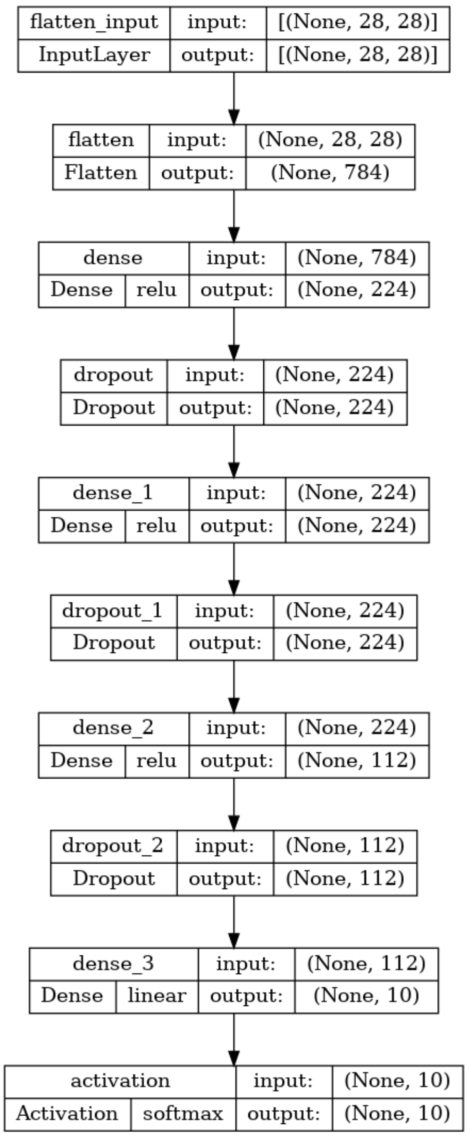

flatten_input | InputLayer | input: [(None, 28, 28)] | output: [(None, 28, 28)]
flatten | Flatten | input: (None, 28, 28) | output: (None, 784)
dense | Dense | relu | input: (None, 784) | output: (None, 224)
dropout | Dropout | input: (None, 224) | output: (None, 224)
dense_1 | Dense | relu | input: (None, 224) | output: (None, 224)
dropout_1 | Dropout | input: (None, 224) | output: (None, 224)
dense_2 | Dense | relu | input: (None, 224) | output: (None, 112)
dropout_2 | Dropout | input: (None, 112) | output: (None, 112)
dense_3 | Dense | linear | input: (None, 112) | output: (None, 10)
activation | Activation | softmax | input: (None, 10) | output: (None, 10)

Figure B.5: Image_MLP

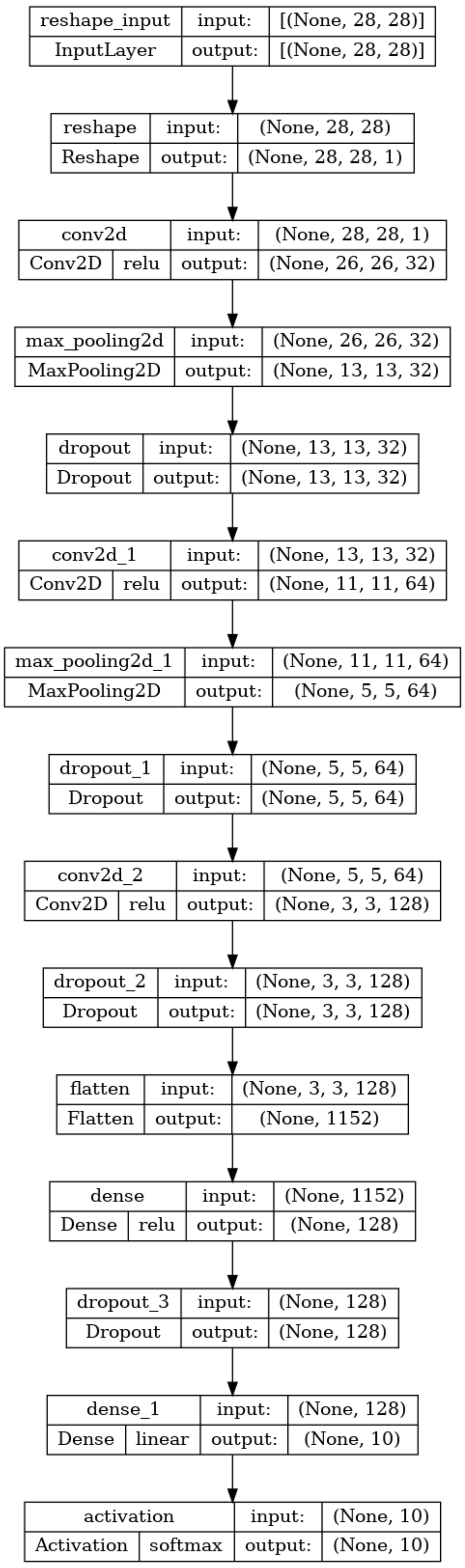

reshape_input | InputLayer | input: [(None, 28, 28)] | output: [(None, 28, 28)]
reshape | Reshape | input: (None, 28, 28) | output: (None, 28, 28, 1)
conv2d | Conv2D | relu | input: (None, 28, 28, 1) | output: (None, 26, 26, 32)
max_pooling2d | MaxPooling2D | input: (None, 26, 26, 32) | output: (None, 13, 13, 32)
dropout | Dropout | input: (None, 13, 13, 32) | output: (None, 13, 13, 32)
conv2d_1 | Conv2D | relu | input: (None, 13, 13, 32) | output: (None, 11, 11, 64)
max_pooling2d_1 | MaxPooling2D | input: (None, 11, 11, 64) | output: (None, 5, 5, 64)
dropout_1 | Dropout | input: (None, 5, 5, 64) | output: (None, 5, 5, 64)
conv2d_2 | Conv2D | relu | input: (None, 5, 5, 64) | output: (None, 3, 3, 128)
dropout_2 | Dropout | input: (None, 3, 3, 128) | output: (None, 3, 3, 128)
flatten | Flatten | input: (None, 3, 3, 128) | output: (None, 1152)
dense | Dense | relu | input: (None, 1152) | output: (None, 128)
dropout_3 | Dropout | input: (None, 128) | output: (None, 128)
dense_1 | Dense | linear | input: (None, 128) | output: (None, 10)
activation | Activation | softmax | input: (None, 10) | output: (None, 10)

Figure B.6: Image_MLPx

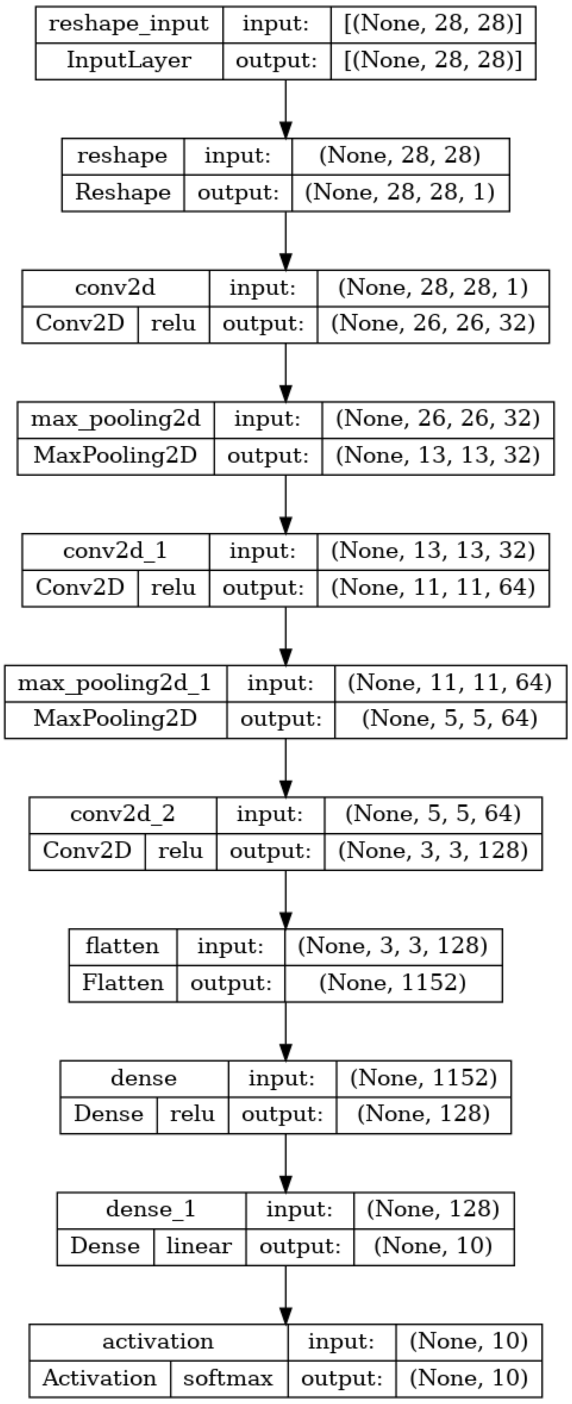

reshape_input | input: | [(None, 28, 28)]
InputLayer | output: | [(None, 28, 28)]
reshape | input: | (None, 28, 28)
Reshape | output: | (None, 28, 28, 1)
conv2d | input: | (None, 28, 28, 1)
Conv2D | relu | output: | (None, 26, 26, 32)
max_pooling2d | input: | (None, 26, 26, 32)
MaxPooling2D | output: | (None, 13, 13, 32)
conv2d_1 | input: | (None, 13, 13, 32)
Conv2D | relu | output: | (None, 11, 11, 64)
max_pooling2d_1 | input: | (None, 11, 11, 64)
MaxPooling2D | output: | (None, 5, 5, 64)
conv2d_2 | input: | (None, 5, 5, 64)
Conv2D | relu | output: | (None, 3, 3, 128)
flatten | input: | (None, 3, 3, 128)
Flatten | output: | (None, 1152)
dense | input: | (None, 1152)
Dense | relu | output: | (None, 128)
dense_1 | input: | (None, 128)
Dense | linear | output: | (None, 10)
activation | input: | (None, 10)
Activation | softmax | output: | (None, 10)

Figure B.7: Image_CNN

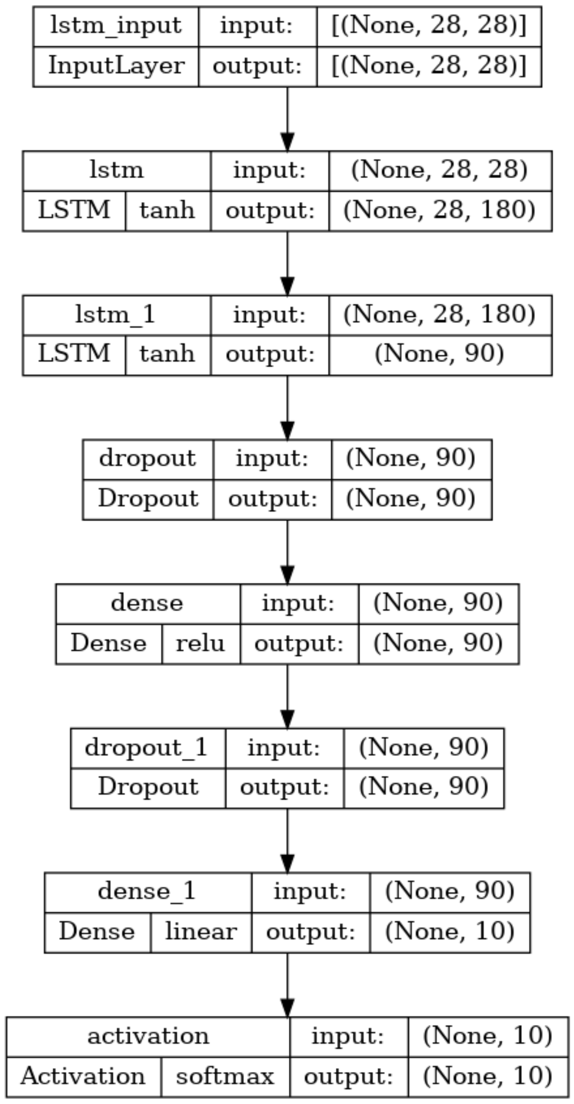

lstm_input | input: | [(None, 28, 28)]
InputLayer | output: | [(None, 28, 28)]
lstm | input: | (None, 28, 28)
LSTM | tanh | output: | (None, 28, 180)
lstm_1 | input: | (None, 28, 180)
LSTM | tanh | output: | (None, 90)
dropout | input: | (None, 90)
Dropout | output: | (None, 90)
dense | input: | (None, 90)
Dense | relu | output: | (None, 90)
dropout_1 | input: | (None, 90)
Dropout | output: | (None, 90)
dense_1 | input: | (None, 90)
Dense | linear | output: | (None, 10)
activation | input: | (None, 10)
Activation | softmax | output: | (None, 10)

Figure B.8: Image_RNN

B.3 Sequential data - IMDB movie review sentiment classification

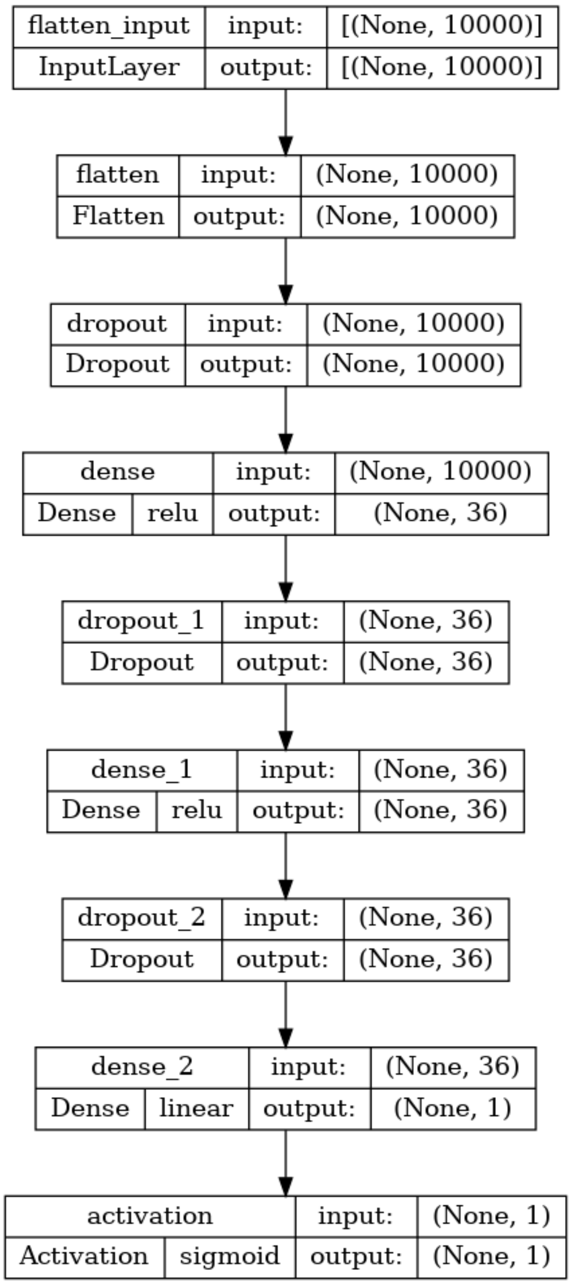

flatten_input | InputLayer | input: [(None, 10000)] | output: [(None, 10000)]
flatten | Flatten | input: (None, 10000) | output: (None, 10000)
dropout | Dropout | input: (None, 10000) | output: (None, 10000)
dense | Dense | relu | input: (None, 10000) | output: (None, 36)
dropout_1 | Dropout | input: (None, 36) | output: (None, 36)
dense_1 | Dense | relu | input: (None, 36) | output: (None, 36)
dropout_2 | Dropout | input: (None, 36) | output: (None, 36)
dense_2 | Dense | linear | input: (None, 36) | output: (None, 1)
activation | Activation | sigmoid | input: (None, 1) | output: (None, 1)

Figure B.9: Sequential_MLP

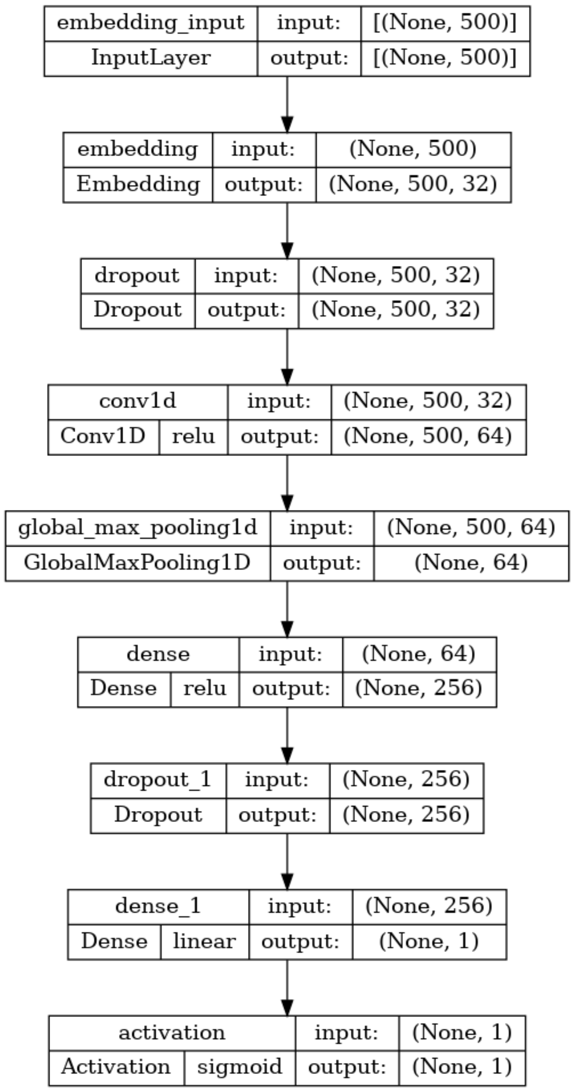
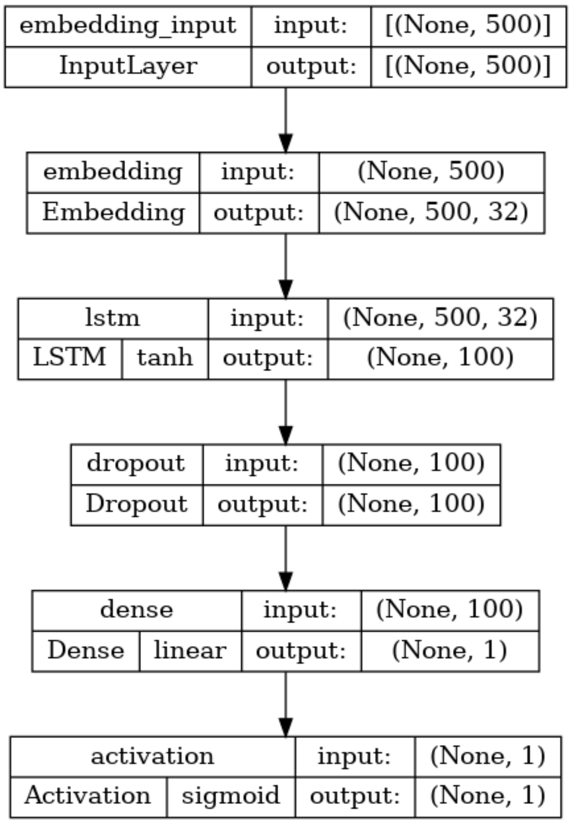
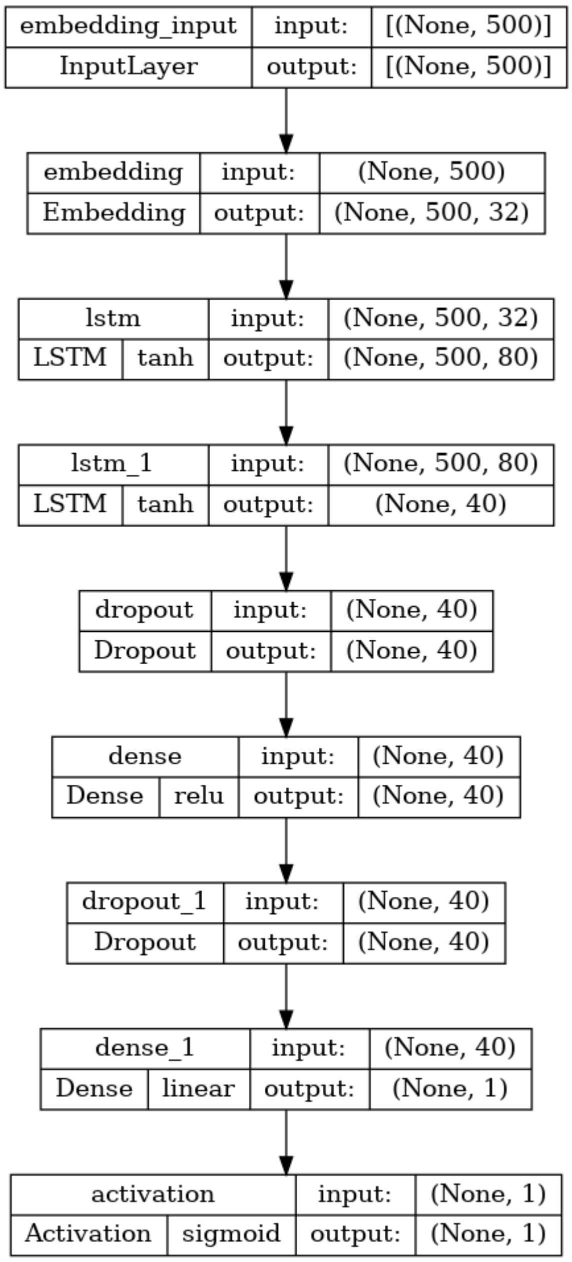

embedding_input | InputLayer | input: [(None, 500)] | output: [(None, 500)]
embedding | Embedding | input: (None, 500) | output: (None, 500, 32)
dropout | Dropout | input: (None, 500, 32) | output: (None, 500, 32)
conv1d | Conv1D | relu | input: (None, 500, 32) | output: (None, 500, 64)
global_max_pooling1d | GlobalMaxPooling1D | input: (None, 500, 64) | output: (None, 64)
dense | Dense | relu | input: (None, 64) | output: (None, 256)
dropout_1 | Dropout | input: (None, 256) | output: (None, 256)
dense_1 | Dense | linear | input: (None, 256) | output: (None, 1)
activation | Activation | sigmoid | input: (None, 1) | output: (None, 1)

Figure B.10: Sequential_CNN



embedding_input | input: | [(None, 500)]
InputLayer | output: | [(None, 500)]
embedding | input: | (None, 500)
Embedding | output: | (None, 500, 32)
lstm | input: | (None, 500, 32)
LSTM | tanh | output: | (None, 100)
dropout | input: | (None, 100)
Dropout | output: | (None, 100)
dense | input: | (None, 100)
Dense | linear | output: | (None, 1)
activation | input: | (None, 1)
Activation | sigmoid | output: | (None, 1)

Figure B.11: Sequential_RNN



embedding_input | input: | [(None, 500)]
InputLayer | output: | [(None, 500)]
embedding | input: | (None, 500)
Embedding | output: | (None, 500, 32)
lstm | input: | (None, 500, 32)
LSTM | tanh | output: | (None, 500, 80)
lstm_1 | input: | (None, 500, 80)
LSTM | tanh | output: | (None, 40)
dropout | input: | (None, 40)
Dropout | output: | (None, 40)
dense | input: | (None, 40)
Dense | relu | output: | (None, 40)
dropout_1 | input: | (None, 40)
Dropout | output: | (None, 40)
dense_1 | input: | (None, 40)
Dense | linear | output: | (None, 1)
activation | input: | (None, 1)
Activation | sigmoid | output: | (None, 1)

Figure B.12: Sequential_RNNx