Improving Domain-specific Transfer Learning Applications for Image Recognition and Differential Equations

Alessandro Saverio Paticchio

Student Id: 894092

Tommaso Scarlatti

Student Id: 897651

Supervisor: Prof. Marco Brambilla

Advisor: Prof. Pavlos Protopapas

Dipartimento di Elettronica, Informazione e Bioingegneria

Politecnico di Milano

Master of Science in Computer Science and Engineering

June 2020

Se sei la persona più intelligente della stanza, sei nella stanza sbagliata.

Ringraziamenti

Quando ho iniziato il mio percorso accademico avevo un sentimento misto di ambizione e timore per quello che sarebbe stato: stavo facendo un passo lungo, forse troppo, ed in salita. Con il tempo e con l'impegno, ne ho poi fatti tanti di passi, e quasi sempre in avanti.
All'inizio e durante il cammino ho avuto al mio fianco - fortunatamente - tante persone, che mi hanno spesso indicato la via, mi hanno dato supporto quando ero stanco, hanno festeggiato con me i piccoli traguardi.

Il più grande dei grazie va al mio tesoro più prezioso: Mamma, Papà e Irene. In questi anni ho fatto dei sacrifici che sono briciole rispetto a quelli che voi avete fatto per me. Siete la mia più potente energia, il mio asso nella manica, il mio porto sicuro. Siete sempre con me e io sono sempre con voi.

Grazie ad Aya, la mia tifosa numero uno, che dice di non capirmi ma che in realtà mi conosce meglio di quanto non mi conosca io. Sei una delle persone più belle che la vita mi ha fatto incontrare. Sei fantastica. E no, there's no sarcasm here.

Grazie ai *ghei ingenui* e ai *grifoni malvagi dorati*: Castro, GioDero, Rubbo e Davide. Sono contento di aver vissuto con voi gli anni da sbarbatello e di ritornare adolescente ogni volta che ci vediamo: rendete i miei ritorni a Lecce calorosi, accoglienti e spassosi.

Grazie a Santa e Tricky: condividere con voi gli anni del Poli è stata una fortuna inestimabile, sarete sempre il mio punto di riferimento e spero che il nostro cammino insieme continuerà domani. Siete degli amici veri.

Grazie a chi ho incontrato in CdM: Carlotta, GioFer, Tex, Frank, Dacciarmeniamer, Nick Star, Luca, Ksenija, Flavio e Lorenzo. Voi siete la vera ricchezza del Collegio ed il motivo per cui io la chiamo Casa. Siete stati ogni giorno ispirazione, risate e un'ottima scusa per bersi una birra.

Grazie a Zio Giacomo e Zio Francesco, la mia casa lontano da casa. Mi avete preso sotto le vostre ali protettive e mi avete dato le giuste dritte per cavarmela in una nuova vita.

Grazie agli *amici ruba-bici*: Gaia, Sabino, Assunta, Fede e Edo. Siete stati la mia famiglia in Svezia e il calore che mancava in un paese nordico.

Grazie ai ragazzi di *Polimi Data Scientists*: siete uno dei frutti più belli dei miei anni a Milano. Continuate a spaccare tutto.

Grazie ai miei relatori prof. Marco Brambilla e prof. Pavlos Protopapas: la vostra conoscenza e i vostri consigli ci hanno guidato in maniera sicura nel difficile mondo della ricerca.

E infine, grazie al mio compagno di avventure Tommaso: te l'avrò detto un milione di volte, ma mi ritengo fortunatissimo ad aver lavorato con te, e spero che succederà ancora. Ma oltre che un compagno di tesi sei stato un compagno di risate, di sfortune, di vittorie: un amico.

- Alessandro

Ringraziamenti

Ho agognato questo momento per due lunghi anni e mezzo e, sarò sincero, avrei preferito viverlo in un contesto totalmente differente. Purtroppo mi trovo ad affrontare una crisi inaspettata, dettata da cause ancora a me sconosciute, a cui cercherò di rispondere come ho sempre fatto: lavorando sodo e con costanza, per uscirne più forte di prima e per poter, un giorno, guardarmi indietro con una consapevolezza rinnovata.

I primi momenti al Politecnico li ricordo come momenti di estremo spaesamento e indaeguatezza. A tal proposito mi viene in mente una frase provocatoria che da sempre mi è cara: "*Se sei la persona più intelligente della stanza, sei nella stanza sbagliata*". Ed è proprio lì, nei giorni più duri, che ho capito di essere nella stanza giusta, nel momento giusto.

Questa tesi è dedicata a tutti voi che mi avete aiutato in questo lungo cammino, fatto di vittorie, così come di sconfitte. Questa tesi è dedicata a voi che avete gioito con me nei momenti di festa e sofferto con me nei momenti più tristi, a voi, insomma, che ci siete sempre stati.

Il mio ringraziamento più grande va a mia Mamma ed a mio Babbo. Ho sempre cercato di ripagare gli enormi sforzi che avete fatto per me con i risultati accademici, ma mai come adesso sento il bisogno di esternare l'immenso bene che vi voglio. Siete la mia ancora, i miei pilastri. Siete tutto per me.

Grazie a mio Zio Giovanni, che da sempre mi sostiene indistintamente e che per me è come un fratello maggiore.

Grazie alla Nonna Tina, che ahimè, non è riuscita a vedere la fine di questo percorso. Ti porterò sempre nel mio cuore.

Grazie ai miei Zii e Cugini di Roma. La vostra guida durante la mia infanzia è stata di fondamentale importanza, e non vi siete mai fatti mancare una parola di incoraggiamento nel

momento del bisogno.

Grazie alla mia cara, dolce, Lavinia, che nonostante la distanza non mi ha mai fatto mancare l'amore e il supporto di cui avevo bisogno, cercando di comprendermi fin oltre ogni limite.

Grazie a tutti i FdM, in particolare a Brasa, Spiga, Difa e Frago. Non riuscirò mai fino in fondo a dimostrarvi il bene che vi voglio.

Grazie a tutta la *Gang of Five* per questi due anni di straordinaria convivenza insieme. La Gang è per sempre.

Grazie a GM ed al Lory, fedeli commilitoni di battaglie accademiche e compagni di notti insonni. Con voi l'università ha assunto decisamente un altro sapore.

Grazie all'affiatata compagine dei Creamy Fireflies. Insieme abbiamo superato sfide ostili, consumato copiose confezioni di Kinder Cards, ed alla fine siamo riusciti a scalare Vancouver.

Un immenso ringraziamento va alla vecchia guardia di Firenze, sempre pronta a risollevarmi e rinsavirmi nei weekend, ed ai nuovi, speciali, amici di Milano, con cui ho condiviso momenti unici ed indimenticabili.

Vorrei ringraziare i miei due relatori: il Prof. Marco Brambilla e il Prof. Pavlos Protopapas. Il vostro supporto, la vostra guida, e i vostri continui feedback sono stati fondamentali per lo sviluppo e per la stesura della tesi. I tre mesi di lavoro ad Harvard sono stati un'esperienza estremamente formativa ed emozionante.

Infine, last but not least, vorrei ringraziare Alessandro: collega, compagno, amico. Sarò sincero, senza di te non sarei mai riuscito a superare i mesi in America ed a portare a termine questo lavoro. Hai subito e le mie lamentele e miei momenti più bui, ma spero che tu abbia anche gioito delle mie risate. Confido nel fatto che le noste strade si incrocino ancora nella vita.

- Tommaso

Abstract

In recent years, deep neural networks have become an indispensable tool for a wide range of applications, on which they have achieved extremely high predictive accuracy, in many cases, on par with human performance. These models led to great improvements in state-of-the-art results of many difficult tasks, such as image classification, speech recognition, or natural language processing. A considerable huge amount of data is a fundamental, necessary condition for training deep learning architectures, since is in their nature to be extremely *data hungry* models. Another factor that must be taken into account is that deep learning requires high-performance computational resources and very long training times.

An approach that helps overcoming the problem of computational cost is *transfer learning*, that consists of leveraging the knowledge acquired by a model, trained on a source task, to solve a target task, saving time and energy. This thesis explores the field of transfer learning in two very different scenarios: image recognition and resolution of differential equations. In both cases, we investigated previous research works in the literature, trying to improve and extend proposed techniques on one hand, and developing new ideas and new approaches on the other. In the image recognition task, which is a supervised learning scenario, we focus on the problem of data impact in a transfer learning setting. In this scenario, we developed different criteria to select a subsample (i.e. perform a data selection) of the target dataset, in order to train in a smarter and faster way. We tested the different criteria on a variety of combinations of datasets, distortions and models, finding that results are poorly generalizable. In the scenario of resolution of differential equations, instead, we have no *actual data*, hence we focused on the problem of the perturbation of the initial conditions and the parameters of the equations, investigating how transfer learning helps with this particular type of distortions, and proposing new network architectures. We show how transfer learning accelerates the resolution of several systems of differential equations and that it becomes even more helpful with our modifications to the source-trained network.

Abstract

Negli ultimi anni, le reti neurali profonde sono diventate un indispensabile strumento per un'ampia gamma di applicazioni, in cui hanno raggiunto un'alta precisione predittiva, spesso alla pari della performance umana. Questi modelli hanno portato a grandi miglioramenti allo stato dell'arte di molti problemi complessi, come classificazione di immagini, riconoscimento vocale o elaborazione del linguaggio naturale. Una considerevolmente grande quantità di dati è una condizione necessaria e fondamentale per allenare architetture di apprendimento profondo, dato che è nella loro natura essere modelli estrememamente *affamati di dati*. Un altro fattore di cui tener conto è che l'apprendimento profondo richiede risorse computazionali ad alte performance e tempi di allenamento molto lunghi.
Un appproccio che aiuta a superare il problema del costo computazionale è il *trasferimento dell'apprendimento*, che consiste nello sfruttare la conoscenza acquisita da un modello, allenato su un problema origine, per risolvere un problema obiettivo, risparmiando tempo ed energia. Questa tesi esplora il campo del trasferimento dell'apprendimento in due scenari molto differenti: riconoscimento di immagini e risoluzione di equazioni differenziali. In entrambi i casi, abbiamo investigato precedenti lavori di ricerca in letteratura, cercando sia di migliorare ed estendere le tecniche proposte, sia di sviluppare nuove idee e nuovi approcci. Nell'ambito del riconoscimento di immagini, che è uno scenario di apprendimento supervisionato, ci focalizziamo sul problema dell'impatto dei dati in un contesto di trasferimento dell'apprendimento. In tale ambito, abbiamo sviluppato diversi criteri per selezionare un sottoinsieme (i.e. effettuare una selezione dei dati) nel dataset obiettivo, per allenare in modo più intelligente e veloce. Abbiamo testato diversi criteri su una varietà di combinazioni di dataset, distorsioni e modelli, scoprendo che i risultati non sono generalizzabili. Nell'ambito della risoluzione di equazioni differenziali, invece, non abbiamo *veri dati*, perciò ci siamo focalizzati sul problema della perturbazione delle condizioni iniziali e dei parametri dell'equazione, investigando come il trasferimento dell'apprendimento può aiutare con questo particolare tipo di distorsioni, e proponendo nuove architetture per le reti. Mostriamo come il trasferimento dell'apprendimento accelera la risoluzione di molti sistemi di equazioni differenziali e che diventa ancora più vantaggioso con le nostre modifiche alla rete allenata sul problema d'origine.

Contents

List of Figures

Chapter 1

Introduction

1.1 Context and Problem Statement

In recent years, deep neural networks have become an indispensable tool for a wide range of applications, on which they have achieved extremely high predictive accuracy, in many cases, on par with human performance. These models led to great improvements in state-of-the-art results of many difficult tasks, such as image classification, speech recognition, or natural language processing. In particular, for computer vision tasks, the ease of design for such networks has established DNNs as the *go-to* solution. This was possible thanks to numerous open source deep learning libraries that have been developed in the last decade [7], [13].

The reasons why deep learning gained this terrific success are manifold. The information age has generated an exponential increase in the amount of digital data being stored, and consequently, the number of large scale carefully annotated datasets is increased as well [22], [72]. A considerable huge amount of data is a fundamental, necessary condition for training deep learning architectures, since is in their nature to be extremely *data hungry* models. Another factor that must be taken into account is that deep learning requires high-performance computational resources and very long training times. Indeed, in the last years, we have witnessed a huge increase in the computational power of state-of-the-art machines, together with the rise of cloud-based computing. Furthermore, in order to address their needs, new hardware, specifically designed for deep learning, have been developed, aiming at accelerating training and performances of neural networks, keeping the power consumption low [32].

Modern deep neural networks take weeks or even months to train across multiple GPUs on very large datasets. For this reason, it is common to use pre-trained models whenever is possible. A pre-trained model is a model that has been trained on a dataset, usually large, and that can be used as a starting initialization point for training another model on a different task.

If the new task to solve is similar to the original one, this approach can save computational time and achieve a better performance compared to training the model from scratch. This technique, also known as *finetuning*, is collocated in the field of *transfer learning*, a peculiar research field in machine learning whose aim is to find a way to take advantage of the knowledge acquired by a given model (source) on a given task and use it as a resource to solve a different task with a different model (target).

In this context, the question we pose ourselves is the following: exploring transfer learning, can we find smarter techniques to transfer the knowledge already acquired? In other words, operating in a transfer learning setting, can we find a way to reduce further the computational footprint? Can we find a way to improve the convergence and the final accuracy of our target model?

1.2 Proposed Solution

This thesis explores the field of transfer learning in two very different scenarios: image recognition and resolution of differential equations. In both cases, we investigated previous research works in the literature, trying to improve and extend proposed techniques on one hand, and developing new ideas and new approaches on the other.

In the image recognition task, which is a supervised learning problem, we focused on the problem of data impact in a transfer learning setting. In particular, we show the effect of *dataset shift*, namely when the probability distribution of data from source to target changes, on the accuracy of different models. We explored two different settings: a target model trained on a different, but similar, dataset, and a target model trained on a distorted version of the dataset on which the source model was trained on. In this two cases, we developed different criteria to select a subsample (i.e. perform a data selection) of the target dataset, in order to train in a smarter and faster way. We tested the different approaches on a variety of combinations of datasets, distortions, and models, finding that results are poorly generalizable. Furthermore, the criteria we implemented introduce an overhead due to the selection process which must be taken into account when comparing the computational costs.

For what concerns the resolution of differential equations, the scenario here is completely diverse. In this case we want to use a neural network to solve a differential equation (or a system of them). Our work is based on an approach proposed by [45], that explains how to solve one single system of differential equation, that, together with a combination of a set of initial conditions, form a *Cauchy problem*. These conditions are usually pre-imposed to the network, and solutions learnt strictly depend on them. In other words, if conditions are modified, solutions changes accordingly. In this particular scenario, transfer

learning techniques can be employed: the knowledge acquired to solve the equations on a particular set of conditions can be used to to solve equations on a different set. Our proposed solution introduces a new set of approaches to perform transfer learning in this scenario, which go beyond the classical finetuning of the target model. We show that, applying slight modifications to the network, we can gain a huge benefit in terms of computational time required.

1.3 Structure of the Thesis

The structure of the thesis is organized as follows:

- **Background** defines and explains the background knowledge and concepts on which this thesis is based on.
- **Related Work** provides an overview of other studies in the literature that addressed the same problem that we have covered.
- **Methodology** explains the methodology we followed in our research, describing each step in details.
- **Implementation** describes the major technologies, libraries and tools we employed, along with deployment strategies.
- **Experiments** is devoted to show the outcomes of the experiments performed to validate our approaches.
- **Conclusion** wraps up the discussion with concluding remarks and advises possible future works to be carried on.

Chapter 2

Background

In this section we will briefly describe the core concepts on which our thesis is based. We will start by introducing the machine learning field, the tasks it solves and the techniques it uses. We will then approach deep learning, starting from the theory behind Feed Forward Neural Networks. An introduction to Convolutional Neural Networks and Autoencoders will follow, highlighting the power of these architectures in many applications. In section 2.3 and 2.4 we will present differential equations and dynamical systems respectively, focusing on the relation between each other. Finally, we present transfer learning and its benefits when applied to deep learning models.

2.1 Machine Learning

In a broader way, this research is collocated in the field of Machine Learning. Machine Learning is the area of study of algorithms and statistical models that computers exploit to perform specific tasks, without using explicit instructions, but relying on patterns and inference captured from data. Indeed, Machine Learning algorithms learn from data, extracting features and taking decisions upon them. [46] provides a formal definition of a learning algorithm:

A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

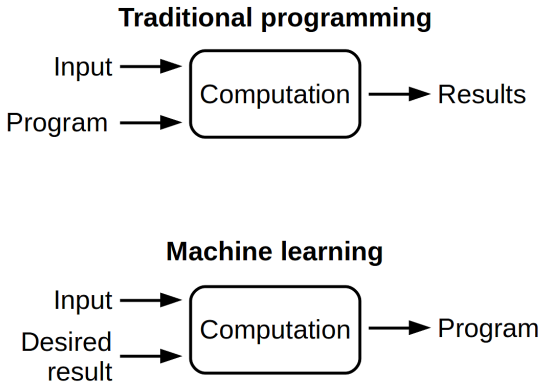


Figure 2.1 Intuitive comparison between Machine Learning and traditional programming.

Figure 2.1 is a widely used example to compare traditional computer programming to Machine Learning: the former will ask the programmer for an input and a set of instructions to follow in order to achieve the result, the latter will be fed with input and a desired output and will take care of understanding how to go from one to the other.

In the following pages we will provide the main concepts of Machine Learning to give a better understanding of the scientific knowledge we relied on.

2.1.1 Supervised and Unsupervised Learning

In Machine Learning, we call supervised learning a problem in which the goal is to learn a mapping between a set of inputs x and a set of outputs y.
This goal is usually accomplished by starting from a dataset D = $\{(x_i, y_i)\}_{i=1}^{N}$, which is a set of N couples (x_i, y_i), where the vector **x** represents the input features (also called attributes or covariates) and the **y** is a single output feature. The algorithm will learn the patterns underlying in the features space **X**, in order to predict the target feature **y**. The vector **x** can be of any type and have any complex form: sequence of numbers, strings, images, videos etc., and likewise **y** can be of any form, but most methods assume that the target variable can either have a value among a finite set, i.e. $y \in \{1...C\}$, or a real scalar value.
In the first case, we deal with a classification problem, as the target can assume one among a finite set of classes, whereas in the second case we deal with a regression problem.
On the opposite side, unsupervised learning problems will not predict any target variable, but will instead learn patterns from data, relying on a dataset in the following form: D =

$\{(x_i)_{i=1}^N\}$. We will not go into further details in this case as it is not in the scope of our project.

2.1.2 Models

Generally, the output of a Machine Learning algorithm is a model, which learns from data and will make prediction on new and unseen inputs.
There are two main types of models: parametric and non-parametric. [63] explains that parametric models are the ones that summarize the data with a set of parameters of fixed size, that consists of the mapping between the covariates and the output.
Instead, in non-parametric models you do not worry about the number of parameters, as it is potentially infinite and grows as the amount of data increases. They are typically more flexible, but will become very computationally heavy as the number of data explodes.
Furthermore, each model is characterized by two types of prediction errors: bias and variance. The bias is the difference between the average prediction of a model and the true value it should predict, and it is typically large when your model is too simple to capture enough information from your data, while the variance measures how your error fluctuates with respect to small variations in your learning data, and it is large when your model has learned too much from the training data and is not able to generalize on new ones.
When choosing a model, you will have to make a trade-off between these two errors.

2.1.3 Training, Validation and Testing

The typical Machine Learning flow passes by two fundamental steps that will be here delineated: training and testing. Before showing the algorithm all the data you have, you should split those into two separate sets: training data and test data, whose size are typically 80% and 20% respectively.
After this splitting, you may want to further split your training data, reserving another 20% to the validation set, whose use will be explained later.
In the training phase, a Machine Learning algorithm is fed with a set of data, the training set, that will help in understanding the structure and the distribution of the data, so that the model learns how to map each input to each output. Basically, during the training phase, the algorithm minimizes a loss (or cost) function, which is a measure of how good the model performs in the task is meant to solve. There are many types of loss functions and they differ from problem to problem. Some concrete examples are:

- Binary Cross Entropy Loss - $L := -\sum_i^N (\hat{y}_i \log(y_i) + (1-\hat{y}_i)\log(1-y_i))$

- Cross Entropy Loss - $L := -\sum_i \sum_{c=1}^{M} \hat{y}_{i,c} \log(y_{i,c})$, with M classes
- Mean Square Error - $L := \frac{1}{N} \sum_i^N (y_i - \hat{y}_i)^2$

Cross Entropy Loss is used for classification problems, whereas Mean Square Error is used for regression problems, but each setting may require a particular loss, as already stated.

Generally, your algorithm may have some hyperparameters, which cannot directly learn from data, but instead are decided a priori, before the training phase. The hyperparameters should be tuned to obtain the best performance, and that is why we need the validation set: a typical approach is to train your model with a set of different hyperparameters and then check your model performance on the validation set, selecting the one that yields the best results.

After the training phase, the algorithm outputs a model, which has been fit on the training data and which can easily make predictions on input points it has already seen during the training phase. However, it would be good to verify if the trained model performs well also on unseen data points. Indeed, the testing phase is exactly an assessment of your model on new data. This is why it is extremely important to separate training and testing at the very beginning of your process, so that your model will see the testing data only when the training phase is complete, and will not be biased in the prediction.

Based on the results of the training phase, you can make conclusions on how the process has been carried on. Certainly the best scenario is when the performance on testing data is comparable to the performance on training data, yet unfortunately this is not always the case. For instance, there are cases in which your model performs quite well on your training set, but poorly on your testing set: this is called overfitting, and it happens when your model cannot generalize well, going from known to unseen data points.
However, if the performance on your training data and testing data are comparable, but poor, you may want either to train your data for some more time or to increase the complexity of your model, so that it can capture the dynamics of your data better.

2.2 Deep Learning

Deep Learning is a subset of Machine Learning where artificial neural networks, algorithms that mimic the human brain, process and learn from large amounts of data, creating patterns to make decisions. The power of Deep Learning, with respect to traditional Machine Learning,

is the ability to automatically discover a hierarchy of features to be used for a variety of tasks. Traditional ML, on the contrary, requires these features to be provided manually by programmers, which is a time-consuming and error-prone operation, and it requires additional domain knowledge.

Deep learning provides a powerful framework for machine learning tasks. By adding more layers and more units within a layer, a deep network can represent functions of increasing complexity. Most tasks that consist of mapping an input vector to an output vector, can be accomplished via deep learning. In contrast to ML, DL needs high-end machines and considerably big amounts of training data to deliver accurate results.

So why is deep learning called deep? It is because of the structure of modern artificial neural networks. Back in the 80s, neural networks were only a few layers deep [30] [56] as it was not computationally feasible to build larger networks due to the limited computational power of the state of the art hardware. Nowadays, it is common to have neural networks with a considerably huge amount of layers and neurons, thanks to the advances in the technological field.

2.2.1 Feed Forward Neural Networks

Deep feedforward networks, also often called feedforward neural networks, or multilayer perceptrons (MLPs), are the fundamental deep learning models. Their goal is to approximate a function $\mathbf{f}$ that maps input $\mathbf{x}$ to output $\mathbf{y}$. [26]

A feedforward network defines a mapping $\mathbf{y} = f(\mathbf{x}, \theta)$ and learns the value of the parameters θ that result in the best possible function approximation. The architecture is composed of multiple units, called neurons, organized in layers and connected to form an acyclic structure. These models are called feedforward because the information only travels forward in the neural network and there are no feedback connections in which outputs of the model are fed back into itself.

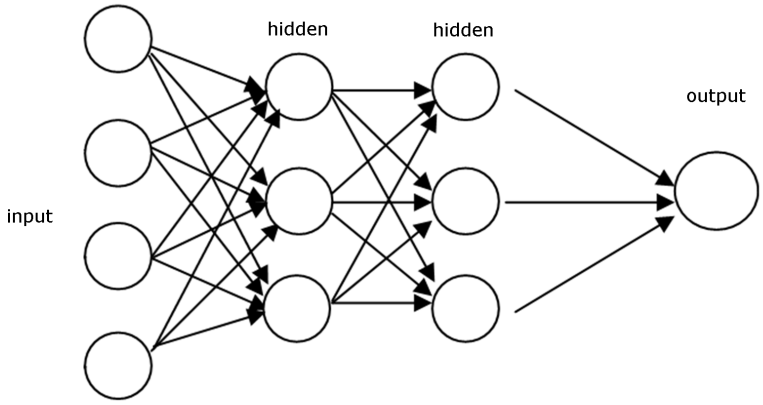


Figure 2.2 Feed Forward Neural Network with four layers: an input layer with four neurons, two hidden layers with three neurons each, and an output layer with a single neuron.

Feedforward neural networks are called **networks** because they are typically represented by composing together many different functions. The model's acyclic graph structure describes how these different functions are composed together. Functions are connected to form a chain, and the length of this chain defines the depth of the model. As mentioned above, a layer is a collection of neurons operating together at a specific depth within a neural network. We can differentiate between three types of layers:

- *Input layer:* is responsible for receiving raw data used as input to the model.
- *Hidden layer(s):* these layers reside in between the input and the output layer. The word "hidden" implies that they are not visible to the external systems and are "private" to the neural network. Their job is to extract from the inputs a hierarchy of features that can be used by the output layers.
- *Output layer:* is the last layer of a Feed Forward Neural Network. It is responsible for producing an output $f^*(x, \theta)$ as close as possible to the real output $\mathbf{y}$.

Finally, these networks are called **neural** because they are loosely inspired by neuroscience, sharing similarities with with the biological structure of the brain. The fundamental unit of computation of an artificial neural network, the artificial neuron, is a mathematical model of a biological neuron, which is composed of dendrites, axon, synapses and of the cell body.

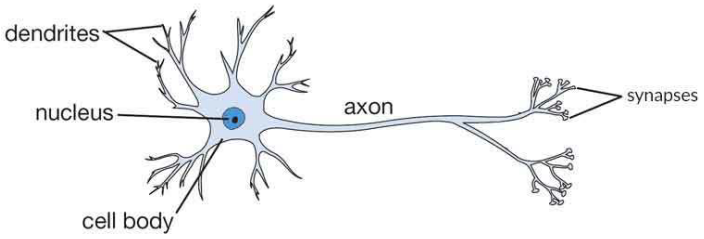


Figure 2.3 Graphical representation of a biological neuron.

- *Dendrites* are in charge of collecting electrical signals from the axons of other neurons, either inhibitory or excitatory.
- *Synapses* between the dendrite and axons modulate electrical signals in various amounts.
- A neuron fires an output signal through its *axon* only when the total strength of the input signals exceed a certain threshold. This output is then fed into other neurons of the network.

The artificial neuron resembles the biological one: the electrical signals are represented as a vector of numerical values. Each element of this vector is multiplied by a weight, in the same way synapses modulate the electrical signal. Finally, a weighted sum of the inputs with respect to the weights is computed. If this sum is above a given threshold called "bias", the neuron's activation function produces a positive output. More formally, given an input vector **x**, the artificial neuron output **y** is given by:

$$y = g(\sum_{i=1}^{I} x_i w_i - b)$$

where w_i is the weight of the i-th input x_i, b is the bias or treshold and g is the activation function of the neuron. These concepts were first introduced by Rosenblatt's perceptron in 1958, which enabled the training of a single neuron in an iterative manner. [54]

2.2.2 Activation Functions

The activation function g plays a crucial role in the learning ability of the neuron. If the activation function is not applied, the output signal becomes a simple linear function. Linear functions are only single-grade polynomials and therefore, a non-activated neural network will act as a linear regression with limited learning power. If the output signal of each neuron is a simple linear function, we cannot expect our network to capture underlying non-linear

patterns in real-world, complex data such as: images, videos, texts and sounds. Using a non-linear activation function allows to design artificial neural networks as universal function approximators.

Since the network is composed of stacked layers of neurons, the output of each layer becomes the input of the following one, and thanks to this chain structure, the output function will be a composition of all the non-linear functions learned by its layers. Formally speaking, given a set of input x and the function f learned by a feed forward neural network with L layers we have that:

$$f(\mathbf{x}) = f^L(f^{L-1}(...f^0(\mathbf{x})))$$

where f^l is the non-linear function learned by layer l, f^L and f^0 are the functions learned by the output and input layers respectively. Every activation function takes a single number and performs a certain fixed mathematical operation on it. For the output layer, the activation function is chosen depending on the output needed. On the other hand, for hidden layers, there is no universal rule for choosing a particular activation function. Here is a list of the most common activation functions adopted in neural networks:

- **ReLU** (Rectified Linear Unit): it is the most common activation function nowadays.

$$g(z) = ReLU(z) = max\{0, z\}$$

- **Sigmoid**: the sigmoid activation function, also called the logistic function, was the default activation through the early 1990s. It takes a real-valued number and "squashes" it into range between 0 and 1.

$$g(z) = \sigma(z) = \frac{1}{1+e^{-z}}$$

- **Tanh** (Hyperbolic tangent): the hyperbolic tangent squashes a real-valued number to the range $[-1, 1]$.

$$g(z) = tanh(z) = 2\sigma(2z) - 1$$

2.2.3 Training in Neural Networks

Training a neural network is not much different from training any other machine learning model. The main difference between traditional machine learning models and neural networks is that the nonlinearity of the activation functions of each neuron causes the loss function to become non-convex. Therefore, neural networks are usually trained by using an

iterative framework, called gradient descent, together with an optimization algorithm called backpropagation. They will be both described in details in the following paragraphs.

As any other machine learning model, a neural network needs to know whether it is performing well or not during the training phase, and therefore it needs a performance metric. This metric is usually referred to as the cost function or loss, already illustrated in the previous paragraphs. The idea behind gradient-based optimization techniques is the following: compute the gradient of the loss with respect to model parameters and then leverage the gradient itself to update the parameters. Generally speaking, a gradient is a vector-valued function that represents the slope of the tangent of the graph of the function, pointing the direction of the greatest rate of increase of the function. Since we want to minimize the loss function, we update the parameters in the direction opposite to the one pointed by the gradient. This will result in a reduction of the loss function value, which implies that the difference of the current behavior of the network and the expected one is reduced as well. In this way we can improve the performance of our model.

Deep learning models require a huge amount of data to reach good performances on the task they have been trained for. Classical gradient descent method uses the whole training set to perform gradient computation and an update of the weights. This whole process is called a cycle or training epoch. Since the dataset is an aggregation of data points, it can be called as a batch. Hence, this process is also known as **Batch Gradient Descent**. This can become easily unfeasible when dealing with deep learning models: large datasets often cannot be held in RAM, and even if they can, it is an extremely computationally expensive operation for just one update of the weights. Therefore, variants of gradient descent method has been developed. They basically differ on the amount of data used to calculate the gradient at each step:

- **Stochastic Gradient Descent**: stochastic gradient descent (SGD) performs a parameter update for each observation. Instead of using the whole training set, it just needs one sample to perform the parameters update. This computational advantage is leveraged by performing many more iterations of SGD, making many more steps than conventional batch gradient descent. One drawback is that due to its stochastic approach, this algorithm is less regular than the previous one. This algorithm may also result in a local minimum but not in the global minimum.

- **Mini-batch Gradient Descent**: it is a combination of both batch gradient descent and stochastic gradient descent. Mini-batch gradient descent performs an update for a random batch of observations. This algorithm reduces the noise occurred in the

Stochastic Gradient Descent and is more efficient than Batch Gradient Descent, that is why it is the algorithm of choice for training deep learning models.

So far we have seen different variants of gradient-based methods, but how is the gradient of the loss practically computed in a neural network model? We recall that the gradient of the loss is a vector-valued function containing the partial derivatives with respect to all the parameters. Computing such derivatives may seem like an extremely complex problem because of the many connections and dependencies among the various parameters; luckily, thanks to an algorithm called **backpropagation** [40], which relies on the chain rule of composite functions, this task can be accomplished by reusing a lot of computation.

Given a feed forward neural network with L layers, three adjacent layers are shown in the figure below.

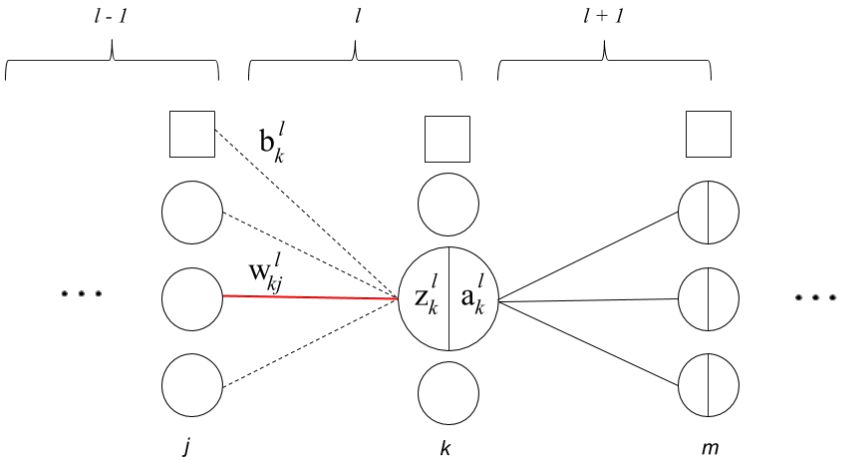


Figure 2.4 Three generic adjacent layers in the network, derivative is taken with respect to the weight shown in red. [1]

The input sum of a neuron k in layer l is defined as:

$$z_k^l = \sum_j w_{kj}^l a_j^{l-1} + b_k^l$$

The activation function of neuron k is:

$$a_j^l = g(z_j^l)$$

where g can be any of the nonlinear functions mentioned above. Finally, the input sum of a neuron m in layer $l+1$ is:

$$z_m^{l+1} = \sum_k w_{mk}^{l+1} a_k^l + b_m^{l+1}$$

Given the cost function (loss), of the network C, we define, the *error signal* of a neuron k in layer l as:

$$\delta_k^l = \frac{\partial C}{\partial z_k^l} = \left(\sum_m \frac{\partial C}{\partial z_m^{l+1}} w_{mk}^{l+1} \right) g'(z_k^l)$$

this is a measure of how much the total cost changes when the input sum of the neuron is changed. The last expansion of the formula is given by the chain rule of composite functions: the error signal for a neuron k at level l depends on the error signals of all the neurons at level $l+1$, since they are all connected to each other. So we have a recursive formula for the error signal of each neuron:

$$\delta_k^l = \left(\sum_m \delta_m^{l+1} w_{mk}^{l+1} \right) g'(z_k^l)$$

For what concerns the biases of the network, the gradient of the cost function with respect to them is simply the error signal:

$$\frac{\partial C}{\partial b_k^l} = \frac{\partial C}{\partial z_k^l} \frac{\partial z_k^l}{\partial b_k^l} = \delta_k^l$$

The backpropagation algorithm works as follow. First of all, in order to use the recursive formula, we need a forward pass to compute the neuron activation a^L of the last layer of the network. In this way, we can compute the error signal δ^L and propagate it backwards, calculating all the error signals of the network in a recursive manner. Using backpropagation in gradient-based learning algorithms, we can compute the parameters updates to minimize the loss function in an iterative fashion. Given a batch of training samples of size N, weights are updated according to the delta rule:

$$w^l \to w^l - \eta \frac{1}{N} \sum_{n=1}^{N} \delta^{n,l} (a^{n,l-1})^T$$

$$b^l \to b^l - \eta \frac{1}{N} \sum_{n=1}^{N} \delta^{n,l}$$

where the tuning parameter η is called *learning rate*. This parameter influences to what extent newly acquired information overrides old information, and therefore it represents the speed at which a machine learning model *learns*.

2.2.4 Convolutional Neural Networks

Convolutional Neural Networks [39] (CNNs) are a specialized kind of neural network for processing data that has a known, grid-like topology. Examples include time-series data, which can be thought of as a 1D grid taking samples at regular time intervals, and image data, which can be thought of as a 2D grid of pixels. Over the past decade, Deep CNNs have become one of the most used and successful algorithms in a variety of tasks of Computer Vision (CV), including image classification [64], object detection [66] and semantic image segmentation [43]. The name "convolutional neural network" refers to the fact that the network employs a specialized kind of linear operation called **convolution**. As stated in [26]:

Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.

In the following paragraphs we will first describe what convolution is and then we will shed light on the the motivation behind using it in a neural network. We will then describe an operation called **pooling**, which almost all convolutional networks employ.

Convolutional layers

Convolution is an operation on two functions of a real-valued argument that produces a third function expressing how the shape of one is modified by the other. Let be x and w two real-valued functions. The resulting function s of the convolution of x and w is defined as:

$$s(t) = \int x(a)w(t-a)da$$

The convolution operation is typically denoted with an asterisk in this way:

$$s(t) = (x * w)(t)$$

If we now assume that x and w are defined only over a set of integers, we can define the discrete convolution as:

$$s(t) = \sum_{a=-\infty}^{+\infty} x(a)w(t-a)$$

The input of a convolutional neural network is usually a multidimensional array of data and the kernel is usually a multidimensional array of parameters that are adapted by the learning algorithm. We will refer to these multidimensional arrays as **tensors**.

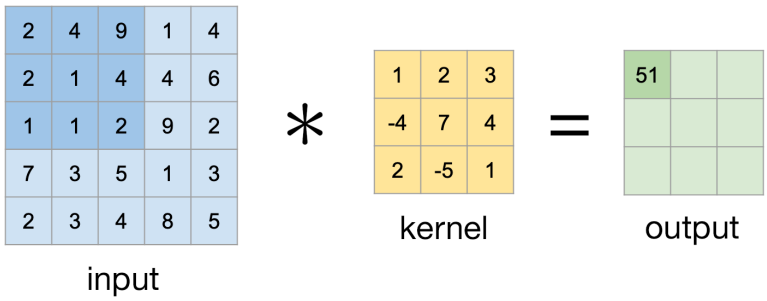


Figure 2.5 Convolution operation implemented by a 2D convolutional layer with kernel size of 3 and stride of 1 in both directions.

The convolutional layer is the core building block of a CNN and it does most of the computational heavy lifting. Figure 2.5 shows how the convolution operation is implemented in a convolutional layer. Basically, we apply dot products between a receptive field (input) and a filter (kernel) on all the dimensions. The outcome of this operation is a single integer of the output volume (feature map). Then we slide the filter over the same input image by a stride parameter and compute again the dot products between the new receptive field and the same filter. We repeat this process until we go through the entire input image. For each layer there are three parameters to be chosen: the **kernel size**, the **stride** and the **padding** [24]. The stride is the amount by which the kernel shifts after each dot product with the overlapping area of the input. Padding refers to the number of pixel added to the frame of the image to allow for more space for the kernel to cover. In this way, it is possible to preserve the original size of the input image after the convolution.

Convolution is based on three important underlying ideas that have determined their success in a variety of fields in the last decade: sparse interactions, parameter sharing and equivariant representations.

- **Sparse interactions**: (also referred as sparse connectivity or sparse weights) refers to the usage of a kernel which has a smaller size than the input image. This means that in each convolutional layer we need to store fewer parameters with respect to a fully connected one. In this way we have a two-fold benefit: the memory requirements of the model are reduced and its statistical efficiency is improved as well.

- **Parameter sharing**: refers to using the same parameter for more than one function in a model. In a traditional fully connected layer, each element of the weight matrix is used exactly once when computing the output of a layer. It is multiplied by one element of the input and then never revisited. In a convolutional neural net, instead, each weight of the kernel is used multiple times across the input image during the convolution operation. Convolution is thus dramatically more efficient than dense matrix multiplication in terms of the memory requirements and statistical efficiency.
- **Equivariance**: equivariance to translation is a property of convolutional layers that is caused by the particular way in which parameters are shared. A function is said to be equivariant if a modification of the input reflects in the output in the same way. For instance, with images, convolution creates a 2-D map of where certain features appear in the input. If the object is moved in the input, its representation will move the same amount in the output, preserving feature locality. Convolution is not naturally equivariant to some other transformations, such as changes in the scale or rotation of an image. Other mechanisms are necessary for handling these kinds of transformations, such as image augmentation techniques [51].

Pooling layers

A typical layer of a CNN is made of three stages: in the first stage, several convolutions are performed in parallel to generate a set of linear activations. In the second stage, linear activations are passed through a nonlinear function (detector stage). Finally, in the third stage, a pooling function modifies the output further. The main function of the pooling stage is to reduce the spatial size of the representation and therefore the amount of parameters and computation in the network.

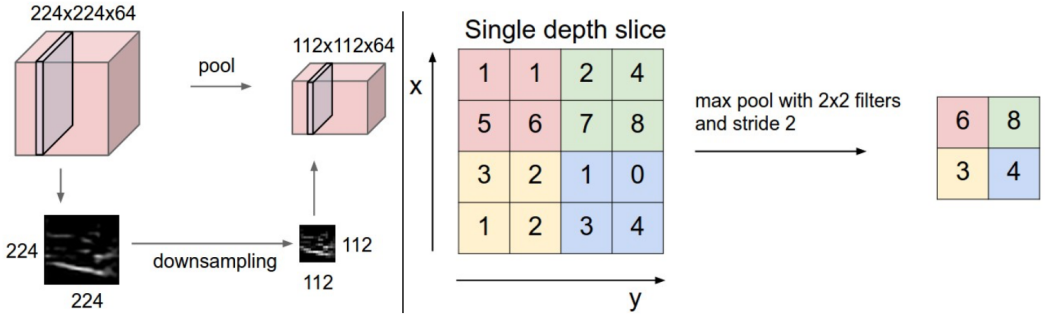


Figure 2.6 Left: input volume of size [224x224x64] is pooled with filter size 2, stride 2 into output volume of size [112x112x64]. Notice that the volume depth is preserved. Right: max pool operation using a 2x2 kernel over a 2-D input [9].

As we can see in figure 2.6 above, the pooling layer operates independently on every depth slice of the input, preserving the volume depth. There are many different variants of pooling, the most common form is a pooling layer using the *MAX* operation, with filters of size 2x2 and applied with a stride of 2. In all cases, pooling helps to make the representation become approximately invariant to small translations of the input, improving noise robustness and control overfitting.

Network architectures

Convolutional networks have played an important role in the history of deep learning. The classical CNN architecture employs one or more blocks containing the aforementioned three stages and then one or more fully connected layers. Basically, everything before the fully connected part can be thought as a sequence of feature extractors organized in a hierarchical way. Then, the fully connected part uses the high-level features extracted to perform the desired task. This kind of architecture was popularized by LeNet-5 [41], which was used on large scale to automatically classify hand-written digits on bank cheques in the United States.

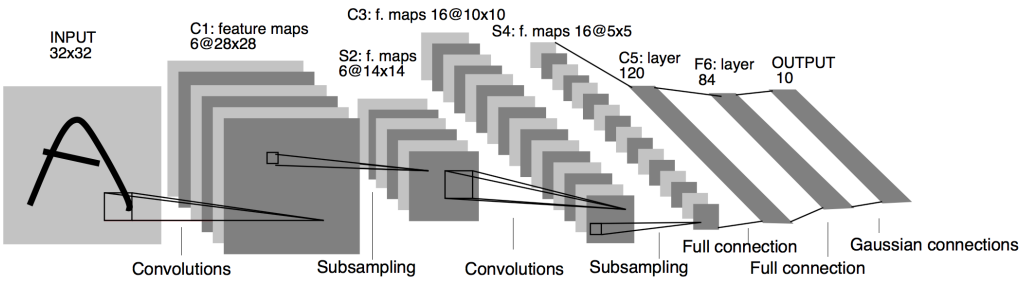


Figure 2.7 LeNet-5 architecture as published in the original paper [41].

As we have seen so far, convolutional networks provide a smart way to specialize neural networks to work with data that has a clear grid-structured topology. The current intensity of commercial interest in CNN began when *Krizhevsky et al.* (2012) [37] won the ImageNet object recognition challenge [57]. Since then, the main focus in the field was to make deeper and deeper networks, in order to improve the state of the art accuracy in a variety of problems (and contests). Models that have gained popularity throughout the last decade are: "AlexNet" (2012) [37], "ZF Net" (2013) [74], "GoogLeNet" (2014) [65], "VGG Net" (2014) [60] and "ResNet" (2015) [28]. The latter features special *skip connections* and a heavy use of batch normalization, and it is currently by far the state of the art for Convolutional Neural Network models.

2.3 Differential Equations

Now that the modellistic part has been explained, we will proceed by illustrating the field of differential equations, where we applied some of the models described. In the following paragraphs we will provide the basic concepts of the theory of the topic, together with some real examples.

2.3.1 Definitions

The history of differential equation has root in the XVII century, thanks to the contribution of Isaac Newton, who introduced the first types of differential equations [48], and the Bernoulli brothers. In the scientific research world, they are used in many applications to model real dynamical systems, hence they attract a lot of interest from the whole scientific community. Indeed, they describe how a given quantity varies, in relation to the quantity itself. From now on, we will describe the fundamentals concepts behind the theory of the differential equations, whose extended content is outlined in [19]. We define **ordinary differential equation (ODE) of order n** a relation of the form:

$$F(t, y(t), y'(t), y''(t), \ldots, y^{(n)}(t)) = 0, \text{with } F : \mathbb{R}^{n+2} \supseteq U \to \mathbb{R} \tag{2.1}$$

We call it *ordinary* as the unknown is the function of only one variable, and the *order* of a differential equation is the maximum order of derivation that appears in it. Instead, we call **partial differential equation** (PDE) if it is a differential equation that contains unknown multivariable functions and their partial derivatives. If in equation 2.1 we can make the maximum order derivative explicit, we can say that the equation is written in the **normal** form. For homogeneity, all the following definitions will assume that equation 2.1 is written in normal form. If equation 2.1 is a first degree polynomial, we say that the equation is **linear**, with the following general form:

$$a_0(t)y^{(n)}(t) + a_1(t)y^{(n-1)}(t) + \cdots + a_{n-1}(t)y'(t) + a_n(t)y(t) = b(t)$$

The evolution during time of many physical systems can be described by mean of linear differential equations, hence there is a strong and extensive theory about them. On the opposite side, **non-linear** equations are solvable with very few methods and can have very weird behaviors over large time intervals.

We define **solution** of equation 2.1 a function $\varphi = \varphi(t)$, defined and differentiable n-times in an interval $I \subseteq \mathbb{R}$ such that $(t, \varphi(t), \ldots, \varphi^{(n-1)}(t)) \in D$ and such that:

$$\varphi^{(n)}(t) = f(t, \varphi(t), \varphi'(t), \ldots, \varphi^{(n-1)(t)}) \ \forall t \in I$$

meaning the n-derivative of φ is a function of time and all the derivatives of order $(0, \ldots, n-1)$. We can think of variables whose change is affected by some other variables too, hence forming relations between them to form a system of equations. **Systems of ordinary differential equations** in multiple unknowns, each of them expressed as a function of a single variable, are defined as follows:

$$\begin{cases} \dot{y}_1 = f_1(t, y_1, y_2, \ldots y_n) \\ \dot{y}_2 = f_2(t, y_1, y_2, \ldots y_n) \\ \vdots \\ \dot{y}_n = f_n(t, y_1, y_2, \ldots y_n) \end{cases}$$

$$\text{shortly } \dot{\mathbf{y}} = f(t, \mathbf{y}(t))$$

If all the components of f_i are linear in y, then the system is linear too.

2.3.2 Solutions of differential equations

Given a differential equation:

$$y'(t) = f(t)$$

it has infinite solutions of type $y(t) = \int f(t)dt + c, \ c \in \mathbb{R}$. Hence, for an equation $\dot{y} = f(t, y)$, an infinite number of solutions exist, all represented by a parametric family $y = \varphi(t, c)$ called **general integral**. Nonetheless, we are more interested in finding solutions satisfying some additional conditions. Regarding this, we introduce the **Cauchy problem**, which is defined as:

for scalar equations of order n: finding y of class C^n such that

$$\begin{cases} y^{(n)}(t) = f(t, y_1(t), y_2(t), \ldots y_n(t)) \\ y(\tau) = \xi_0 \\ y'(\tau) = \xi_1 \\ \vdots \\ y^{(n)}(\tau) = \xi_n \end{cases}$$

$$\tau, \xi_0, \ldots, \xi_{n-1} \in \mathbb{R}.$$

for systems: finding a vector $\mathbf{y}$ of class C^1 such that:

$$\begin{cases} \dot{\mathbf{y}} = f(t, \mathbf{y}(t)) \\ \mathbf{y}(\tau) = \xi \end{cases}$$
$$\tau \in \mathbb{R} \text{ and } \xi \in \mathbb{R}^n.$$

Equations $y^j(\tau) = \xi_j$ and $\mathbf{y}(\tau) = \xi$ are called **boundary conditions** (or initial conditions if $\tau = 0$), and in this case the solutions are locally defined around τ. Basically, the Cauchy problem consists of searching for a solution y, in the whole family, that satisfies all the boundary conditions imposed by the problem.
The local existence of a solution was proven by Cauchy and Peano, under the assumption of continuity and limitedness. Consequently, Picard and Lindelöf showed that, for a lipschitz-continuous function y, the solution is also guaranteed to be unique.

2.4 Dynamical Systems

As outlined in the previous section, differential equations are used in many applications to model real dynamical systems. In the following paragraphs we will give a brief overview of what dynamical systems are and we will provide some examples as well.

2.4.1 Introduction

The term dynamic refers to phenomena that produce time-changing patterns: characteristics of the pattern at one time are interrelated with those at other times. This term is nearly synonymous with time-evolution, or pattern of change. Dynamical system can be thought as the mathematical prescription for evolving the state of a physical system in time. They deal with the evolution of systems, trying to predict the future of these systems or processes and understanding the limitations of these predictions. A more formal definition [52] is the following:

A dynamical system consists of a phase (or state) space P and a family of transformations $\varphi(t) : P \to P$, where the time t may be either discrete, $t \in \mathbb{Z}$, or continuous, $t \in \mathbb{R}$. For arbitrary states $x \in P$ the following must hold:

1. $\varphi_0(x) = x$ *identity*
2. $\varphi_s(\varphi_t(x)) = \varphi_{t+s}(x)\ \forall t, s \in \mathbb{R}$ *additivity*

At any given time, a dynamical system has a state given by a tuple of real numbers (a vector) that can be represented by a point in an appropriate state space. The *evolution rule* of the dynamical system is a function that describes what future states follow from the current state, and it can be either deterministic or stochastic. It is worth mentioning that dynamical systems may exhibit a completely unpredictable behavior, which might seem to be random, despite the fact that they are fundamentally deterministic. This seemingly unpredictable behavior has been called chaos, and systems which exhibit a chaotic behavior are called *chaotic systems*.

2.4.2 Examples

In this paragraph we will present some examples of dynamical systems described by differential equations.

Simple harmonic oscillator

A simple harmonic oscillator is an oscillator that, when displaced from its equilibrium position, experiences a restoring force F proportional to the displacement x, also known as Hook's law [29]: $\vec{F} = -k\vec{x}$. It consists of a mass m, which experiences a single force F, pulling the mass in the direction of the point $x = 0$ and depends only on the position x of the mass and a constant k. Balance of forces for the system reads:

$$F = ma = m\frac{d^2x}{dt^2} = m\ddot{x} = -kx$$

Solving this ordinary, second order differential equation leads to a function that describes the motion of the mass of the oscillator:

$$x(t) = Acos(\omega t + \phi)$$

where $\omega = \sqrt{\frac{k}{m}}$, A is the amplitude of the sinusoidal function and ϕ is the phase, which describes the starting point on the sine wave.

RC circuit

A resistor–capacitor circuit (RC circuit), is an electric circuit composed of resistors and capacitors driven by a voltage or current source. This type of circuit exhibits a large number of important types of behaviour that are fundamental to much of analog electronics. In particular, it is able to act as a passive filter. The simplest RC circuit is made of a capacitor

and a resistor in parallel. The capacitor will discharge its stored energy through the resistor throughout time. The voltage across the capacitor, which is time dependent, can be found by using Kirchhoff's current law. In fact, the current passing through the resistor must equal the current discharging the capacitor. Formally:

$$C\frac{dV}{dt} + \frac{V}{R} = 0$$

where C is the capacitance of the capacitor. This first order linear differential equation has the following solution:

$$V(t) = V_0 e^{-\frac{t}{RC}}$$

where V_0 is the capacitor voltage at time $t = 0$. In this particular deterministic dynamical system, the voltage V is the only state variable, therefore the phase space is one-dimensional and the system is called a *phase line*.

2.4.3 Hamilton's equations

With classical mechanics (often referred to as Newtonian mechanics), we refer to physical concepts employed and mathematical methods used to describe the motion of bodies under the influence of a system of forces. The earliest development of these concepts is devoted to Isaac Newton and Gottfried Wilhelm Leibniz, back in the 17th century. Classical mechanics proved to guarantee extremely accurate results when dealing with large objects with a speed not close to the speed of light.

An equivalent but more abstract reformulation of classical mechanics was proposed by William Rowan Hamilton in 1833, and it is known as *Hamiltonian mechanics*. In Hamiltonian mechanics, a physical system is described by a set of coordinates $\mathbf{r} = (\mathbf{q}, \mathbf{p})$ and its evolution through time is described by a system of two first order differential equations, the so called *Hamilton's equations*:

$$\begin{cases} \frac{\partial \mathbf{p}}{\partial t} = -\frac{\partial H}{\partial \mathbf{q}} \\ \frac{\partial \mathbf{q}}{\partial t} = \frac{\partial H}{\partial \mathbf{p}} \end{cases} \tag{2.2}$$

where $H = H(\mathbf{q}, \mathbf{p}, t)$ is the *Hamiltonian*, which often corresponds to the total energy of the system [25] and, for a closed system, is the sum of potential and kinetic energy. In classical mechanics, time evolution of both position and velocity of a system is determined using Newton's second law, once the total force applied to the system has been computed. On the other hand, in Hamiltonian mechanics time evolution is obtained by calculating the Hamiltonian H of the system in the canonical coordinates and then inserting it into the system of equations 2.2.

2.5 Transfer Learning

Transfer learning is a peculiar research field in Machine Learning whose aim is to find a way to take advantage of the knowledge acquired by a given model on a given task T_s, called source task, and use it as a resource to solve a different task T_t [26].
[50] provides a formal definition:

Given a source domain D_s and learning task T_s, a target domain D_t and learning task T_t , transfer learning aims to help improve the learning of the target predictive function f_t in D_t using the knowledge in D_s and T_s, where $D_s \neq D_t$, or $T_s \neq T_t$

The most typical approach in transfer learning is using a pre-trained model as a starting point at the beginning of your training phase, so that you can leverage some of the patterns already captured by the baseline and converge faster to your desired extrapolated knowledge.
It is therefore extremely beneficial in cases in which your training phase requires a lot of time, as it may speed up the process by providing an initialization closer to the desired result, or when your training set is not large enough, so that you can leverage features extrapolated from a model trained on a fairly similar and larger dataset.

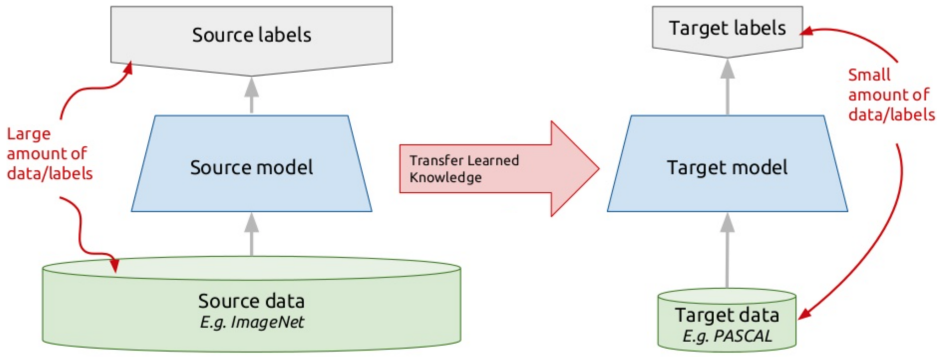


Figure 2.8 An example of Transfer Learning setting: a model pre-trained on ImageNet, a widely known and large images dataset, is exploited to learn from PASCAL, another image dataset with way less data. [12].

Furtherly, depending on the type of problem to solve, there are different cases of transfer learning [50]:

- Inductive transfer learning: $T_s \neq T_t$, no matter if $D_s \neq D_t$ or $D_s = D_t$
- Transductive transfer learning: $T_s = T_t$ and $D_s \neq D_t$
- Unsupervised transfer learning: $T_s \neq T_t$, no matter if $D_s \neq D_t$ or $D_s = D_t$, but T_s and T_t are related

From the above definitions, it is intuitive that transfer learning is often used in case of dataset shift, that is when $P_s(y,x) \neq P_t(y,x)$ [47].
This method is widely used in Deep Learning, since the training usually requires plenty of time, therefore a network already initialized with some knowledge is more likely to get to the desired result faster than training a model from scratch.
The most typical approach, called fine-tuning, takes a pre-trained network and resumes its training with a new target training set, thus skipping the initialization of the weights. It is possible to choose whether to fine-tune the whole network, or to "freeze" some layers and only train some of them, depending on how many data you have at disposal.
In Convolutional Neural Networks, early layers tend to capture generic features, which can be similar across different domains, while later layers learn features which are more dataset-dependent, thus those are usually the most sensitive when fine-tuning [73].

Chapter 3

Related Work

In this chapter, we will go through the literature of scientific research that constitutes the basis of our thesis and gave us the direction to pursue.

Transfer learning is a major topic in machine learning and, in neural networks, it generally entails initializing a recipient neural network using some of the weights from a donor neural network that was previously trained on a related task. This protocol, proposed by Yosinski et al. [73], gained research attention and led to the development of different techniques to refine it, aiming at improving the performance of the recipient network.

In the first section, we will present the main papers that leverage data selection in a transfer learning scenario, which inspired us for our proposed work in the images context. Deep learning is a setting in which the more data you have the better it is, and not much has been researched on trying to reduce the amount of data to feed the algorithm. The lack of literature in this specific field of deep learning was what convinced us to work on it.

In the second section, we will give a brief overview of different techniques developed to use neural networks for solving differential equation, and transfer learning approaches applied in this scenario. In particular, we focused on dynamical systems governed by a system of differential equations. Again, the lack of literature in this scenario led us to investigate transfer learning techniques in this specific field as well.

3.1 Impact of Data on Transfer Learning

Starting from the assumption that transfer learning is a commonly applied method in Machine Learning, for its applicability and for the performance it yields, [49] tackles the problem of the continuous increasing of dataset sizes, especially in the Deep Learning field.
The authors question whether it is true that the more data you have, the better your result will be, and find out that this is not always the case. Indeed, they compare results of transfer

learning on different subsets of training data and find out that the best performance is obtained when some irrelevant samples are discarded. Furthermore, they propose a weighted technique, namely domain adaptive transfer learning, which exploits importance weights computed on the target set. Their method modifies slightly the loss function to minimize, to take into account the (dis)similarity between source domain and target domain, by weighting each sample with respect to the ratio of the distribution of the respective label in the target and in the source datasets.

[55] has similar conclusions: they state that transfer learning is often helpful, but it can harm your result if the source data and the target data are too different. Their experiment involved training a hierarchical Naive Bayes algorithm with different data sources, noticing that the more the sources are dissimilar, the worse the model performs.

[76] introduce a reinforcement learning-based framework, namely L2TL, to improve transfer learning on a target task by a peculiar extraction of information from a source dataset. Their flow trains a policy model by looking at a performance metric on a validation set, which will then output weights for each source class adaptively.

[27] faces the problem of negative transfer, that is the process of using knowledge which will actually hinder your performance, rather than improving it. This paper proposes a novel technique to accurately select those samples that will cause negative transfer and stop the transfer at the top performance gain. Their method exploits the sum of the Rademacher distribution to estimate the class noise rate of transferred data. Transferred data having high probability of being labeled wrongly are removed to reduce noise accumulation.

[71] also contributed to the research in the negative transfer field, and also adopted a weighted mechanism by mean of a Generative Adversarial Network discriminator to perform the density ratio estimation, used in the loss function minimization.

[14] investigates the behaviour of different CNNs in the transfer learning fiels, by studying how a choice of a particular pre-trained CNN model can alter your final result. Given a target task, they rank different models pre-trained on different source task, in order to select which one is more suitable to the given problem, by evaluating the similarity between the source and the target datasets.

[69] introduced a selection technique to accurately pick a subset of your training data that will improve the performance of your Deep Learning model, specifically devoting their experiments on Convolutional Neural Networks. The algorithm is then adapted to a transfer learning setting by [70], and detailed below.

Algorithm 1: Algorithm to obtain an optimized training set for you transfer learning problem [70].

input :

- f_θ: pretrained network from source domain
- X: training set in target domain
- V: validation set in target domain

output : $X\prime$: optimized training set in target domain

for *i = 1 to len(X)* **do**
  for *i = 1 to len(V)* **do**
    use f_θ to compute $I_{loss}(X_i, V_j)$,
    where $I_{loss}(X_i, V_j) = -\nabla_\theta L(x_j) H_\theta^{-1} \nabla_\theta L(x_i)$
    and $H_\theta^{-1} = (\frac{1}{N} \sum_i^N \nabla_\theta^2 L(x_i))^{-1}$
  end
  if $\sum_j I_{loss}(X_i, V_j) > 0$ **then**
    remove X_i from X
end
$X\prime$ is obtained

The proposed algorithm is used in an image recognition setting, where a Convolutional Neural Network model is used to compute the loss function and its derivatives. The approach aims not only at selecting the most impactful samples of a target training set, but also to eliminate some harmful samples, obtaining a better prediction accuracy when those negative samples are excluded from the training.
Nonetheless, this approach has the non negligible drawback of the computational complexity, both in terms of space and time, as one should compute the I_{loss} for each training sample and, in the meantime, store the inverted Hessian matrix, whose size is $n_\theta * n_\theta$, where n_θ is the number of parameters of the network, including weights and biases, which is enormous in a CNN.

3.2 Neural Networks for Solving Differential Equations

The idea to solve differential equations using neural networks was first proposed by Lagaris et al. [38]. They used the assumption that $u(x) = A(x) + F(x, N(x))$, where A and F are carefully designed to satisfy given boundary conditions and N is a neural network trained to

minimize the following loss function:

$$L = \left\| G(x, u(x), \nabla u(x), \nabla^2 u(x)) \right\|$$

where $G(x, u(x), \nabla u(x), \nabla^2 u(x)) = 0$ is the differential equation to be solved. The employment of a neural architecture for solving differential equations has many attractive features, such as: a reduced number of model parameters with respect to any other solution technique, the fact that the method is generalizable, and can be applied to ODEs, systems of ODEs and to PDEs, and the possibility to implement it on parallel architectures.

Recently, studying the evolution of dynamical systems (and therefore solving the differential equations that govern them) has become a significant trend in scientific research. Neural networks have leveraged the exponential increase in the amount of digital data available to explore and forecast the future behavior of complex dynamical systems. Several number of studies have been conducted in a data-driven scenario, for discovering differential equations and finding approximate solutions for those equations. [16], for instance, developed an approach for using NNs for learning optimized approximations to PDEs, working in a supervised setting based on actual solutions to the known underlying equations. [68], instead, introduced a fully data-driven forecasting method for high dimensional, chaotic systems using recurrent neural networks.

In addition to data-driven studies, equation-driven unsupervised NNs have been used to solve both ODEs and PDEs. These type of networks are trained in a completely unsupervised setting, hence do not need any ground truth data. Essentially, the loss function depends only on the solutions obtained by the neural network. [61] developed an approach for solving high dimensional PDEs by approximating the solution with a deep neural network. The network is trained to jointly satisfy the differential operator, initial condition, and boundary conditions. [45] developed a Hamiltonian neural network architecture that is used to solve DE systems. The proposed network speeds up the convergence to the solution with respect to previous NN DE solver and once optimized, it satisfies Hamilton's equations over the entire temporal domain. The Hamiltonian network proved to be more numerically precise and robust for solving dynamical equations than standard semi-implicit schemes such as a symplectic Euler integrator. A crucial role in the performance of the network is played by the form of the parametric solution employed:

$$\hat{\mathbf{z}}(t) = \mathbf{z}(0) + f(t)\mathbf{N}(t)$$

where $\hat{\mathbf{z}}$ is the solution vector discovered by the NN, $\mathbf{z}(0)$ is the initial state vector, and $\mathbf{N}(t) \in \mathbb{R}^D$ is a vector of D outputs of the network. The parametric function $f(t)$ has the following form:

$$f(t) = 1 - e^{-t}$$

and it enforces the initial conditions in the parametric solutions, i.e. $\hat{\mathbf{z}}(0) = \mathbf{z}(0)$ when $f(0) = 0$. Another benefit of the proposed architecture is that individual outputs share all the weights except those in the output layer, allowing correlations between the outputs.

As we outlined in the previous paragraphs, many solutions have been proposed to solve differential equations using neural networks, by means of different approaches and architectures. On the other hand, very little has been done for what concerns domain-specific application of transfer learning in this context. As stated in the introduction of the chapter, Yosinski et al. [73] developed an experimental protocol in a transfer learning setting for quantifying the generality of neural network layers. They defined generality as the extent to which a layer from a network trained on some task A can be used for another task B. With their approach, they successfully confirmed the generality of the first layers of image-based CNNs. In the context of NN for solving differential equations, [44] extends the work of Yosinski et al. proposing a new methodology, studying layer generality across a continuously parametrized set of tasks (given by a family of BVPs), not strictly limited to a binary comparison of two of them. They found that deeper layers become successively more specific to the problem parameter, and that network width can play an important role in determining layer generality.

Chapter 4

Methodology

Now that the basic foundation of our research has been outlined, we will continue with a detailed explanation of our methodology, going step by step into the main parts of our research and detailing each one of them. The structure of this chapter is divided into two main parts: in the first, we will illustrate the main techniques that we adopted to perform data selection in transfer learning on a supervised problem, namely image recognition; in the second, we will outline the experiments we made in the field of DEs, showing how to improve transfer learning, going beyond the simple finetuning.

4.1 Deep Transfer Learning in Image Recognition

4.1.1 Pre-trained Model on a Source Dataset

In chapter 2 the main concepts related to transfer learning have been explained. In our research we focus on Deep Learning models, and in particular on Convolutional Neural Networks, often used for computer vision tasks, and Feed Forward Neural Networks.
As mentioned, transfer learning is applied in contexts in which we have a source domain D_s, a target domain D_t, and a pre-trained model which learnt the mapping between the features and the target of the source domain. In our work, we analyze the problem of **covariate shift**, a particular case of dataset shift, whose definition is the following [47]:

Given a learning problem, where the set of features is defined as x and the target variable is y, we defined covariate shift as $P_s(y|x) = P_t(y|x)$ *and* $P_s(x) \neq P_t(x)$*, where* $P_i(\cdot)$ *is the probability distribution of the data, either in the source domain or in the target domain.*

CNNs and (most of) Neural Networks in general are discriminative models, i.e. when trained, they learn the conditional probability distribution $P(y|x)$, which is the mapping between the covariates and the target variables. This entails that, in the case of a distribution shift in $P(x)$, the mapping learnt by a network will not be adequate anymore and the performance will be harmed. In our specific case, we focus on models which will be trained on source dataset D_s and then apply those same models to a different target dataset D_t. In order to make the discussion more formal, from now on we will name B a neural model trained on the data coming from a source dataset D_s, and will refer to it using "baseline" and "pre-trained model" as synonyms.

4.1.2 Impact of Dataset Shift

Once a model B has undergone a training process, it apprehended the probability distribution $P_s(y|x)$, and will perform as expected on data points coming from the same distribution.
Nonetheless, when fed with data points coming from different distributions, the results might be different from expected.
In order to simulate the distribution change, we exploited an autoencoder architecture, outlined in chapter 2 and which will be furtherly described in the following chapters.
Autoencoders have the capability of reproducing an output as equal as possible to the input they are fed with, by first compressing it in a latent space and then decompressing it. Hence, let us call x_s the clean version of a sample coming from D_s. A is an autoencoder trained on D_s, capable of reproducing accurately points coming from this dataset. x_l is the output of the latent layer of $A(x_s)$, i.e. when the autoencoder is fed with a clean sample, and x_t the output of the last layer of $A(x_s)$.
The distortion that we applied works as follow:

Algorithm 2: Algorithm to obtain a distorted version of a clean dataset D_s.

input :

- D_s: clean dataset
- A: autoencoder trained on D_s
- c: shift constant

output : D_t: distorted dataset

Initialize D_t as an empty dataset
for x_s *in* D_s **do**
    Feed A with x_s: $A(x_s)$
    Get x_l, the output of the latent layer of $A(x_s)$
    Apply $x_l = x_l + c$
    Get x_t, the output of the final layer of $A(x_s)$
    Add x_t to D_t
end
D_t is obtained

This transformation is meant to apply a distribution shift in the dataset D_s, which will harm the performance of B on the target dataset. From now on, we will refer to this particular shift as **embedding shift**.

When applying the embedding shift to images, they get visually distorted proportionally to the shift constant c, as you can see in figure 4.1. In particular, the higher the c, the more the image is distorted and less recognizable, both from a human eye and a neural model. Furthermore, we observed that even setting $c = 0$ resulted in a consistent drop of performance, due to the non-perfect reconstruction of the autoencoder.

By looking at the examples, you can see that, when increasing the shift constant, the network gets more easily fooled and less confident about the output.

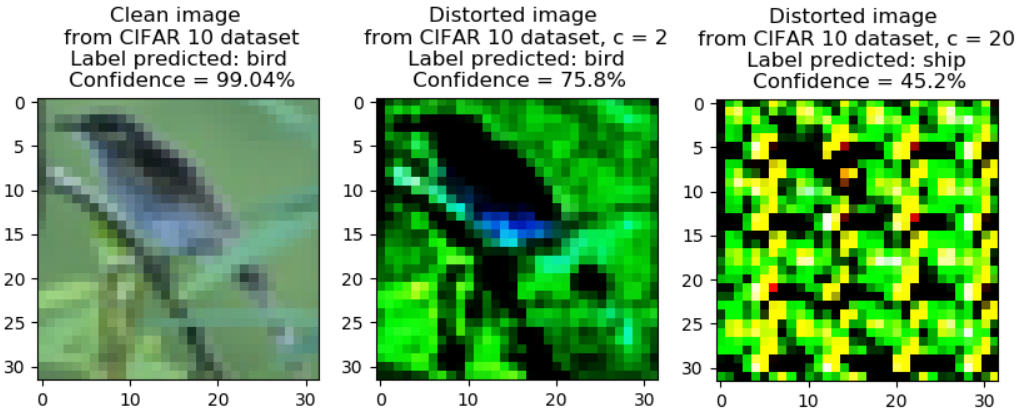


Figure 4.1 In this figure we compare a single image - sampled from CIFAR 10 dataset - and two distortions applied on it. First of all, the higher the shift constant c is, the less the image is recognizable: indeed, with c=20, the image has no meaning anymore. Furthermore, both the accuracy and the confidence level of the network decrease accordingly.

The reasons why we chose this specific distortion type are multiple and here listed:

- It alters the condition of the input and the relative output of a predictive model substantially, so that a concrete transfer learning setting can be exploited to make experiments. Indeed, if the distortion were not strong enough, the model performance would not be altered and thus there would be nothing to "transfer".
- It is tunable. The choice of the shift constant c can make the distortion weak or strong, allowing us to experiment with different levels of transformation.
- In the Computer Vision literature, there is plenty of papers concerning application of noise to images [23], [75], [17]. Nonetheless, noise is a type of distortion that tends to spoil the distribution randomly, and networks trained on noisy data will likely overfit. Instead, applying an embedding shift will actually transform the data consistently, in such a way that if two samples x_s^1, x_s^2 are similar (meaning they likely have the same label), their representation in the latent space x_l^1, x_l^2 will also be similar, likewise x_t^1, x_t^2.

Together with this hand-crafted dataset shift, we will propose the application of the methodologies on another type of distribution variation, which is using a totally new (and non distorted) dataset as target task, with the exactly same labels of the source dataset. More details will be given in section 6.

4.1.3 Data Selection

Transfer learning itself is a widely used method in Machine Learning because it oftentimes speeds up the convergence to the desired result and can even lead you to better performance. Indeed, one of the drawbacks of Deep Learning is the computational resource that it requires to train a model, that is why a transfer learning approach can help, by providing a better starting point for the training phase. In particular, in computer vision tasks, the power required is huge, due to the size of the data points and to the large number of parameters of the trainable networks.
On top of this, the research focuses on the problem of data impact when the user has to face a transfer learning problem: we individuated different criteria to select a subsample of the target dataset D_t, and will analyze the behaviour of the baseline B, that already contains knowledge extracted from D_s, when the training is resumed by using a particular subset $\hat{D}_t$ of the data. The result will be a finetuned model $\tilde{B}$.

Error-driven Approach

The first method that we took into consideration was selecting the samples which the network is wrong the most with. The baselines that we trained, as will be illustrated, were all fit by minimizing the Cross Entropy Loss, mentioned in chapter 2 and here reported for better readability:

$$\text{Cross Entropy Loss - } L := -\sum_i \sum_{c=1}^{M} \hat{y}_{i,c} \log(y_{i,c}) \text{ , with M classes}$$

This metric measures the performance of classification models in multi-class problems. Given a sample x_i, the higher is $L(x_i)$, the more the network is far from classifying it correctly. Given a baseline B, trained on a dataset D_s, this approach selects the samples coming from D_t that cause the network to be wrong the most.
After the selection, the baseline B is then used as a starting point and then is trained on a subset $\hat{D}_t$. The flow is outlined in the following algorithm:

Algorithm 3: Error-driven approach to apply transfer learning from D_s to D_t .

input :

- D_t: distorted dataset
- B: baseline model, trained on the clean dataset D_s
- p: percentage of data to select

Initialize losses-vector E
for x_t *in* D_t **do**
    Feed B with x_t: $B(x_t)$
    Compute loss $L(x_t)$
    Add $L(x_t)$ to E
end
E = *argsort*(E)
Select a subset $\hat{D}_t$, by retaining the first p% of E
Resume training of B with $\hat{D}_t$

The rationale behind this **error-driven** approach is that supposedly, once a model corrects the most critical points - or at least it gets closer to be right - the other ones should follow, and their loss should be minimized as well.

Entropy-driven Approach

The idea behind this approach is based on **active learning**, a special case of machine learning which relies on the assumption that a model is expected to achieve a greater accuracy if it is allowed to accurately pick the data from which it can learn [58]. Starting from this, one of the approaches widely used in literature is *uncertainty sampling*.
Uncertainty sampling is a technique that peculiarly selects a subset of the training data to improve the model performance, giving precedence to the ones which the model is more doubtful about. In our specific classification task, the uncertainty can be seen as a low-confidence by our model in assigning a specific class to a sample. In order to quantify it, we used **information entropy**, that is calculated as:

$$H(x) = -\sum_{m}^{M} p(y = m|x) \log p(y = m|x) \tag{4.1}$$

where x is a single sample, y is the target variable and $m \in (1,\ldots,M)$ are the possible labels. Information entropy, often just entropy, is a fundamental quantity in information theory associated to any random variable, which can be interpreted as the average level of "information", "surprise", or "uncertainty", introduced by Shannon in 1948 [59].
In order to compute this value, as you can see, we need to have a probability distribution of the classes to assign to a given single sample, this is achieved by attaching a softmax classification layer to the network [26]. In our specific case, we adapted 4.1 to the output of the final softmax layer of the baseline network B, as follows:

$$H(x) = -\sum_{m}^{M} p(y = m|B(x)) \log p(y = m|B(x)) \tag{4.2}$$

where B(x) represents the final M-dimensional output of the network.
In fact, the **entropy-driven** approach will feed the baseline network B with the target training set D_t, and will select a subset of samples according to how much B is uncertain about their classification.
By simply looking at the formulation of entropy, it is intuitive to see which samples are more likely to be selected according to this criterion. Given a sample x and its output $B(x)$, when the latter is an array whose values are quite similar between each other, the network is very uncertain about which is the label to assign, and indeed the entropy will be high. Instead, when there is a value that dominates the others (namely close to 1), the entropy of the output vector will be close to zero.
It is also possible to give a geometric explanation of the entropy-driven approach. In order to visualize the selection, we created a simple scenario using a two-dimensioanl synthethized dataset and a binary problem to solve with a simple perceptron. In figure 4.2, you can see how the decision boundary of the model and the entropy are correlated, namely the entropy is proportional to the proximity of a sample to the decision boundary: the closer it is, the more confused the model is about that sample.

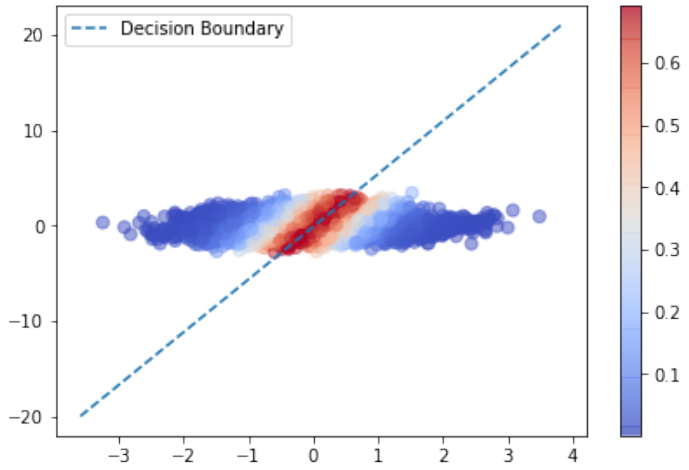


Figure 4.2 This figure visualizes the distribution of a 2-dimensional points set on the space. The dashed line is a decision boundary fit on those same data. As the colorbar suggests, the closest the points are to the boundary, the higher the entropy is.

Intuitively, an entropy-driven selection should select samples whose distribution *shapes* the decision boundary and hence should speed up the learning.
The selection process works by feeding the baseline B with the whole target training set and evaluate each single prediction, hence retaining only the samples whose predictions yield the highest entropy. The amount of samples is regulated by a parameter p, which represents the percentage of data points to select.
The selection is validated by the fact that B is trained on a source domain D_s that, to some extent, resembles D_t. It is worth noticing that this approach is unsupervised, in the sense that it does not need any label to be used, so it could be a doable technique when we do not have labels in the target domain D_t, and would like to know which samples we should give the precedence for labeling and use for train.

Subset recomputation
As explained, the training of a Neural Network is made by feeding the training set into the network for multiple epochs, and each epoch is composed by several updates of the surface that partitions the space in as many regions as many classes there are in the problem. Therefore, even after one single epoch, the decision boundary learnt from the model changes significantly.
In such a situation, the most entropic samples will change as well. In order to take this into account, we expanded the process by recomputing the most entropic sample every T epochs, for n_{rec} times. This approach is meant to capture the evolution of the network, and prevents overfitting on the first computed subset.

Anyways, this approach perturbates the shape of the loss function, once every T epochs, and subsequently the local minima of the network will move. The parameter that regulates how much the network performance is perturbed is, in fact, T. Indeed, at each recomputation i, the procedure will select a subset $\hat{D}_{t,i}$, which will be used for training for T epochs. Naturally, the larger T is, the better the model will fit $\hat{D}_{t,i}$, and those samples will see their entropy decreasing as the training goes on. Hence, if T is large, the set $\hat{D}_{t,i+1}$ will be quite different from $\hat{D}_{t,i}$, resulting in a strong perturbation of the loss function. On the other hands, if T is small, the perturbation will be softer.

Differential Approach

The following proposed method is a modification of a technique proposed by [70], whose work has been outlined in chapter 3. As mentioned, the authors claim that they manage to find a subset of the target data that improve the accuracy of the model. Nonetheless, their approach is extremely computational heavy, both in space and time, as already illustrated previously. Indeed, at our best effort, we did not manage to replicate those same results with the computational power at our disposal, which sheds doubt on the benefit of the proposed method. We tried to adapt that method to simpler problems and simpler networks, but the approach did not yield any result better than a random selection, oftentimes even worse.
The **differential approach** aims at selecting those samples that supposedly should lead to an expected generalization error reduction. This is done by simplifying the algorithm proposed by [70] as follows:

Algorithm 4: Algorithm to obtain an optimized training set for you transfer learning problem, inspired to [70].

input :

- B: pretrained network from source domain
- X_t: training set in target domain
- V_t: validation set in target domain

output : $X_t\prime$: optimized training set in target domain

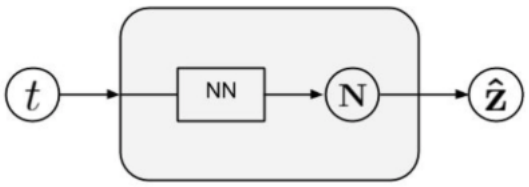
```
for i = 1 to len(X_t) do
    for i = 1 to len(V_t) do
        use B to compute I_loss(X_{t,i}, V_{t,j}),
          where I_loss(X_{t,i}, V_{t,j}) = ∇_θ L(X_{t,i}) ∇_θ L(X_{t,j})
    end
    if Σ_j I_loss(X_{t,i}, V_{t,j}) < 0 then
        remove X_{t,i} from X_t
end
X_t′ is obtained
```

The algorithm 4 calculates the gradient of the loss with respect to the parameters of the network in a training point x, obtaining a vector yielding the direction in which the loss function has the steepest ascent, and performs the dot-product with all the gradients of the loss function computed in the validation points $\mathbf{v}$, summing over all of them.
As commonly known, the dot-product is a measure of similarity between two vectors: if positive, the two vectors point towards similar directions, if negative it is the opposite, if zero the two vectors are orthogonal. From this, it derives that algorithm 4 will retain only the training samples which will impact in the loss in the same way as the validation samples would do, supposedly resulting in a better generalization error.

4.2 Deep Transfer Learning for Differential Equations

4.2.1 Baseline Method

In the recent past, literature about resolution of Differential Equations (DEs) with Neural Networks has been explored [53], [16]. In particular, we made our first steps in this field starting from a methodology proposed by [45], whose architecture is shown in figure 4.3.



$$\hat{\mathbf{z}}(t) = \mathbf{z}(0) + f(t)\mathbf{N}(t)$$
$$f(t) = 1 - e^{-t}$$

Figure 4.3 The network architecture proposed by [45] to solve differential equations.

The given architecture is the composition of a feed-forward neural network that takes a time step $t \in [t_0, t_{final}]$ as input, and a parametrization that produces as output the following:

$$\hat{\mathbf{z}}(t) = z(0) + f(t)\mathbf{N}(t) \tag{4.3}$$

where $f(t) = 1 - e^{-t}$ and $\mathbf{N}(t)$ is the raw output of the network.
The loss function used to solve this problem is dependent on the differential equation(s) you want to solve, and is defined as an MSE between the numerical and analytical time-derivatives of $\hat{\mathbf{z}}(t)$, where the numerical derivative is computed by mean of automatic differentiation.
As an example, given that the differential equation we want to solve is:

$$\frac{\partial \mathbf{z}}{\partial t} = \mathbf{z}^2$$

the MSE loss function will be:

$$L = \frac{1}{K}\sum_{n=1}^{K}\left(\dot{\hat{\mathbf{z}}}^{(n)} - \left(\hat{\mathbf{z}}^{(n)}\right)^2\right)^2 \tag{4.4}$$

where $\mathbf{z}^{(n)} = \mathbf{z}(t_n)$ and K is the total number of points in $[t_0, t_{final}]$ used for the optimization of the network.
What should be noticed is that the loss 4.4 does not contain terms of a ground truth: everything is computed by mean of the output of the network and the parametrization. Hence, this method, is totally unsupervised.
As we know from chapter 2, DEs have families of solution, but we usually want to obtain one of them, basing on an initial condition $\mathbf{z}(0)$. In fact, the purpose of the parametrization 4.3 is used to force the network to predict $\hat{\mathbf{z}}(0) = \mathbf{z}(0)$ for $t = 0$, and to evolve the system starting from there. Furthermore, this particular problem does not suffer from the overfitting

problem: there is no training set and test set, as we only want to know the solution of the equation in the interval $[t_0, t_{final}]$, hence the lower the training loss goes, the better it is.

4.2.2 Perturbation of the Initial Conditions

An important characteristic of the proposed methodology is that it is totally data-less: inputs of the network 4.3 are not data coming from a given distribution, hence we cannot properly refer to concepts like $P(x)$ or $P(y|x)$. In this case, Neural Networks are used as an optimization tool to solve a single equation or a system of them. Since there is no dataset, there is not even a *dataset shift*. Nonetheless, we know that, with the proposed architecture, you can solve only one Cauchy problem, for one or more DEs and one initial condition. If we change the initial condition $\mathbf{z}(0)$ to $\tilde{\mathbf{z}}(0)$, the solution of the network will be wrong, as it has no mean to generalize for all the possible boundary constraints.

In our work we investigated the **perturbation of the initial conditions** of a Cauchy problem, solvable with a Deep Learning architecture. Indeed, we found out that, if you have a network B, trained to solve a Cauchy problem P (with initial conditions $\mathbf{z}(0)$) , you can leverage the knowledge of B to solve $\tilde{P}$: namely, you can solve a Cauchy problem with initial conditions $\tilde{\mathbf{z}}(0) = \mathbf{z}(0) + \delta$, where δ is a perturbation, of various entity, on the initial condition. A more intuitive definition of the problem is defined below:

$$B \text{ solving } P = \begin{cases} \dot{\mathbf{z}} = \mathbf{z}(t) \\ \mathbf{z}(0) = \xi \end{cases} \xrightarrow[\text{transfer}]{\text{knowledge}} \tilde{B} \text{ solving } \tilde{P} = \begin{cases} \dot{\mathbf{z}} = \mathbf{z}(t) \\ \tilde{\mathbf{z}}(0) = \xi + \delta \end{cases}$$

4.2.3 Learning More than One Solution

Transfer learning allows us to *jump* among different Cauchy Problems in a smart, efficient and fast way. Nonetheless, with this architecture we can only solve one Cauchy Problem at a time. For this reason, we slightly modified the architecture, by firstly adding another input: the **initial conditions**.

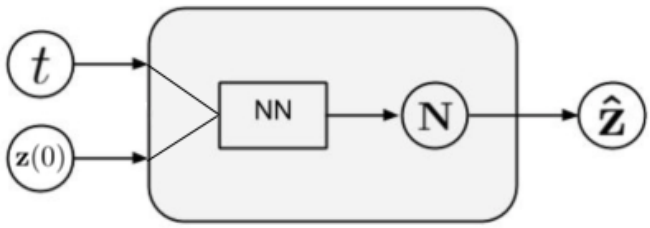


Figure 4.4 Network architecture to learn the solution of a DE (or a system of DEs) for multiple inputs.

This new architecture, shown in figure 4.4 is meant to learn the solution of multiple Cauchy Problems, each of them with a different initial conditions. Specifically this network is able to generalize the solution for $\mathbf{z}(0) \in [\xi_{min}, \xi_{max}]$, as we will show in section 6. In this specific scenario, the transfer learning problem becomes:

$$B \text{ solving } P = \begin{cases} \dot{\mathbf{z}} = \mathbf{z}(t) \\ \mathbf{z}(0) \in [\xi_{min}, \xi_{max}] \end{cases} \xrightarrow[\text{transfer}]{\text{knowledge}} \tilde{B} \text{ solving } \tilde{P} = \begin{cases} \dot{\mathbf{z}} = \mathbf{z}(t) \\ \mathbf{z}(0) \in [\xi_{min} + \delta, \xi_{max} + \delta] \end{cases}$$

Finally, we added another input to the network: the **parameters** θ. Again, this addition gives the network the possibility to generalize among a family of Cauchy Problems, which differ from each other according to initial conditions and parameters.

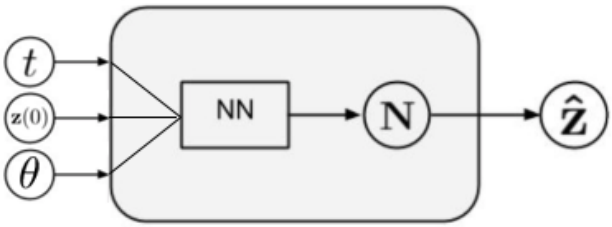


Figure 4.5 Architecture to learn the solution of a DEs for multiple inputs and multiple parameters.

In this scenario, the transfer learning problem becomes:

$$B \text{ solving } P = \begin{cases} \dot{\mathbf{z}} = \mathbf{z}(\theta, t) \\ \mathbf{z}(0) \in [\xi_{min}, \xi_{max}] \\ \theta \in [\theta_{min}, \theta_{max}] \end{cases} \xrightarrow[\text{transfer}]{\text{knowledge}} \tilde{B} \text{ solving } \tilde{P} = \begin{cases} \dot{\mathbf{z}} = \mathbf{z}(\theta, t) \\ \mathbf{z}(0) \in [\xi_{min} + \delta, \xi_{max} + \delta] \\ \theta \in [\theta_{min} + \varepsilon, \theta_{max} + \varepsilon] \end{cases}$$

4.2.4 Use Cases

The listed methodology has been applied on two dynamical systems. In particular, we focused on two non-linear systems of DEs: the Nonlinear oscillator, that describes the dynamics of a an-harmonic oscillator, and the SIR model, that describes mathematically the evolution of an epidemy.

Nonlinear oscillator

In section 2.4.2 we introduced the linear (harmonic) oscillator. The balance of forces for the system leads to the following ordinary, second order differential equation:

$$F = ma = m\frac{d^2x}{dt^2} = m\ddot{x} = -kx$$

We considered the one dimensional nonlinear (an-harmonic) oscillator with the Hamiltonian expressed in this way:

$$H = \frac{p^2}{2} + \frac{x^2}{2} + \lambda\frac{x^4}{4}$$

where the natural frequency and the mass of the oscillator are both considered to be unity. Furthermore, the parameter λ governs the nonlinearity of the oscillator, hence if $\lambda = 0$ we are in the harmonic case. The Hamiltonian H corresponds to the total energy E of the system, and the associated equations of motion are the following:

$$\begin{cases} \dot{x} = p \\ \dot{p} = -(x + x^3) \end{cases}$$

where x and p are the position and the velocity of the oscillator respectively. Furthermore, being the only two state variables of the system, the phase space is two-dimensional and the system is called a *phase plane*. In this case, we write explicitly the initial conditions as:

$$\mathbf{z}(0) = \mathbf{x}(0), \mathbf{p}(0)$$

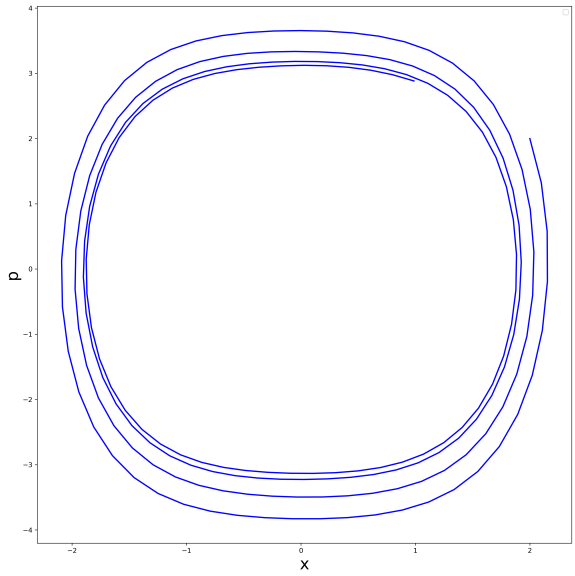


Figure 4.6 Phase space of the nonlinear oscillator for initial conditions: $x(0) = 2.0, p(0) = 2.0$.

SIR model

The SIR (Susceptible-Infected-Recovered) model is a widely used scheme to mathematically characterize the spreading of infectious disease. It divides the population in 3 compartments:

S : the number of susceptible people
I : the number of infected people
R : the number of recovered people

The model makes the population flow in these three compartments, according to the following scheme:

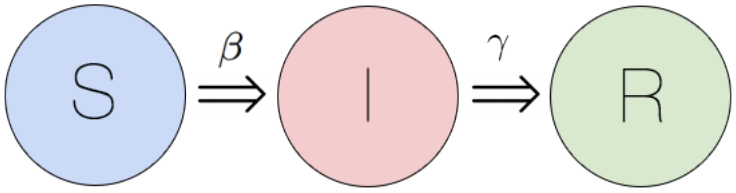


Figure 4.7 Diagram that describes how the individuals move between the compartments in the SIR model.

Hence, each member of the population will go through all the steps, the parameters β regulates how fast you go from Susceptible to Infected and γ regulates regulates how fast you go from Infected to Recovered.

$$\begin{cases} \dot{S} = -\frac{\beta SI}{N} \\ \dot{I} = \frac{\beta SI}{N} - \gamma I \\ \dot{R} = \gamma I \end{cases}$$

$$\mathrm{N = S + I + R}$$

Firstly, you can notice that $\frac{\partial S}{\partial t} + \frac{\partial I}{\partial t} + \frac{\partial R}{\partial t} = 0$, hence the quantity $N = S + I + R$, which is the total population, remains constant. Secondly, the dynamics strictly depends on the parameters β, γ. In particular, you can calculate the basic reproduction ratio $R_0 = \frac{\beta}{\gamma}$, which is the expected number of new infections generated by a single infected individual in a situation where all the people are susceptible [15]. This ratio is a measurement of how much the virus will spread and if, eventually, there will be a total outbreak or not. In this case, we write explicitly the initial conditions and parameters as:

$$\mathbf{z}(0) = \mathbf{S}(0), \mathbf{I}(0), \mathbf{R}(0)$$
$$\theta = \beta, \gamma$$

Chapter 5

Implementation

Now that the main methods we developed have been illustrated, we will go deeper and outline how we actually implemented them. In particular, we will show the types of technologies we relied on, focusing on most critical aspects we faced during the implementation. Furthermore, we will give a brief overview of our deployment choices, driven by the computational requirements of the tasks to solve.

5.1 Source Code

Our entire code base is written in Python [5], a programming language born in 1991, with its first release, and then upgraded multiple times up to version 3.7.6, in 2018 [3]. Python is a high-level programming language, object-oriented and widely used for applications development, scripting, numerical computations and system testings. Being a high-level programming language, it lets the programmer focus on the main core of the problem logic, offering a good level of abstraction. Furthermore, this feature makes the code easier to understand and to maintain.
Another key for Python success is its platform-independence. A script written in Python is compiled into bytcode at first, (a low-level platform independent representation of the source code), and then interpreted by the Python Virtual Machine (PVM). Thanks to this, it can be used to create stand-alone platform-independent applications.
Python is also an open source programming language, and has a variety of compatible frameworks, libraries and tools you can leverage for your software development. In particular, in the last decade Python has become the *go to* option for Machine Learning and AI projects, given the plenty of libraries developed for this purposes.
One of them is PyTorch [5], an open source machine learning library developed by Facebook, which we leveraged to build, train and test our networks. It is a powerful and easy-to-

learn library, widely used for the development of Deep Learning related projects. PyTorch implements dynamic computational graphs, a very useful tool for programmers, who can manipulate the network at runtime, facilitating the model optimization. This feature gives PyTorch a major advantage over other machine learning frameworks, which treat neural networks as static objects [8]. Furthermore, PyTorch offers low-level API that allows a good modularization and specification of the various training phases of a deep learning model.
A central class of PyTorch is `torch.Tensor`, a multidimensional array, easy to manipulate and that can be used on a GPU for accelerate computing. Regarding the training, it was a crucial point tracking the process for all the networks at run time. To do so, we took advantage of Tensorboard [11], a utility to visualize the results of machine learning experiments and keep track of important metrics such as loss and accuracy. Tensorboard is a toolkit developed for Tensorflow, a machine learning library developed by Google, but it is compatible with PyTorch, quick to install and easy to insert in your code.
Together with PyTorch, we leveraged Numpy [4], another open source library that adds support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. Its main class is a powerful N-dimensional array, that is very similar to PyTorch's Tensor, hence it was quick passing objects through the two libraries. Unfortunately, Numpy arrays cannot make use of GPUs and therefore it was not totally suitable for our tasks. Training deep neural networks, indeed, is a computationally heavy process, and GPUs can significantly speed up the computation, that is why we had to leverage PyTorch for this. Nonetheless, we used Numpy for many calculations which could be made on CPU.

5.1.1 Networks Building and Training

PyTorch allows an easy and deep customization of a network architecture, in such a way that the programmer can either find what is needed in the library or re-implement some methods from scratch.
To construct the networks, we used the `torch.nn` package that, together with Autograd (which will be introduced later), define deep learning models and differentiate them. In particular, the `nn.Module` is the class that represents the network itself. This class is totally customizable, and allows the implementation of any network the programmer needs, by stacking layers and attaching activation functions.
Given the high flexibility of PyTorch, we could also manipulate the training process as we wanted: the library does not offer any pre-built `fit` or `train` method, but many tutorials are available to implement those on your own and tailor the process on your needs. This was

crucial given that, in many experiments, we had to pause the train and change the data to feed the network with.

5.1.2 Automatic Differentiation

The fundamental of a neural network training is back-propagation, which consists of the calculation of the gradient of the loss function with respects to all the parameters of the model and the consequent update of these. In PyTorch, this is achieved by its automatic differentiation engine, namely Autograd [6], which provides classes and functions implementing automatic differentiation of arbitrary scalar valued functions. Autograd builds a computational graph that connects all the tensors involved in an operation via operators, to keep track of the connections between all the variables and be able to compute derivatives by mean of the chain rule.
The functions used by Autograd are usually transparent to the programmer when training a network, as the simple command `loss.backward()`, usually written in the training methods, will take care of everything. Nonetheless, we had to compute derivatives in many steps of our projects, such as when implementing the differential approach or when computing the time derivatives in the resolution of DEs. This was made possible by the following Autograd function:

```
torch.autograd.grad(outputs, inputs, grad_outputs=None,
retain_graph=None, create_graph=False, only_inputs=True,
allow_unused=False)
```

which computes and returns the sum of gradients of outputs with respect to the inputs, both of PyTorch Tensor type. Hence, this is the main function we leveraged to compute specific derivatives in the our project.

5.1.3 Loss Customization

PyTorch offers many pre-built classes and functions implementing the most common loss functions for Deep Learning: Cross Entropy Loss, MSE Loss, L1 Loss and many others. Indeed, we made us of `torch.nn.functional.cross_entropy` to train our classification networks. However, for the DEs resolutions, we needed our own loss functions, which were different for the two problems (Nonlinear oscillator and SIR) and, of course, not available in the library.
Fortunately, PyTorch allows the programmer to code a customized loss function too, in the

form of a method that takes as input a set of tensors and outputs the result of the calculation as another tensor. After the computation, it is straightforward to call the method `backward()` on the output tensor, to calculate the gradients through the backward graph, passing by all the nodes (which are weights and biases, in this case), traceable from the output and store them in their `grad` attribute.
Finally, `optimizer.step()` will change the values of all the nodes, according to the just computed gradients.

5.2 Deployment

In this section we introduce our deployment strategies during the development of our thesis. The main factor that drove our choices is the computational complexity (time and space) of the tasks we were required to solve. Another important factor that we took into account is the overhead of moving code and data to a different system, under the assumption that it may be changed frequently.

5.2.1 Local and Remote Deployment

For the development of our thesis, we heavily relied on our local resources (i.e. our laptops), which have the following specifications:

- **Apple MacBook Pro**
  - Intel(R) Core(TM) i7-4558U CPU @ 2.80GHz
  - 8,00 GB RAM DDR3
  - macOS Mojave
- **Dell XPS 13-9360**
  - Intel(R) Core(TM) i5-8250U CPU @ 1.80GHz
  - 8,00 GB RAM DDR3
  - Windows 10

We used PyCharm as our IDE (Integrated Development Environment) of choice to define and structure our project. The IDE offers a total integration to version control systems, such as Git. We also used Jupyter Notebook for fast prototyping and easy visualization, especially in the preliminary phases. Jupyter Notebook is an open-source web application that allows users to create and share documents that contain live code, equations, visualizations and

narrative text. It is a very versatile application, which is used in a variety of tasks: data cleaning and transformation, numerical simulation, machine learning, and much more [34].

Although the majority of tasks was developed and run on local machines, we needed a much more powerful architecture to train our models in a reasonable amount of time. For this reason, thanks to the support of the Instistute of Applied Computational Sciences at Harvard University, we assembled a server machine with two GPUs, which was then connected to the internal network of the university. We named it "Aeneid". Specifications are the following:

- **Aeneid**
  - Intel(R) Xeon(R) CPU E5-1620 v3 @ 3.50GHz
  - 64,00 GB RAM DDR4
  - GPU-0: Nvidia GeForce GTX 1080
  - GPU-1: Nvidia GeForce RTX 2080 Ti
  - Ubuntu 18.04.3 LTS

Using CUDA [21], we leverage the GPUs of the machine to speed up our computational time by a factor of 100x on training of convolutional networks. In order to deploy our code on the remote machine in the most efficient way we used Git, a distributed version-control system for tracking changes in source code during software development.

Debugging of deep learning models is a difficult and time-consuming task. Running models on a remote machine makes debugging even more complicated. For this reason, using PyCharm, we debugged our models using a remote Python interpreter, an interpreter that is located on the server machine. In this way, it was possible to perform debugging operations through the graphical interface of PyCharm on our local machines, while the real debugging was taking place on the remote machine.

5.2.2 Cloud-based Deployment

In addition to the proposed solutions in the previous section, we relied on a cloud-based environment known by the name of Google Colaboratory [2]. Colaboratory is a free Jupyter notebook environment that requires no setup and runs entirely in the cloud. The advantage of Colab is that such notebook is run on the cloud, enabling the possibility of writing and executing code, saving and sharing the analyses, and accessing powerful computing resources. Everything is performed on the cloud, with a web-based interface, and it is totally free. Colab

is also integrated with Google Drive, allowing you to share, comment, and collaborate on the same document and the same data with multiple people. Despite all these benefits, the massive advantage of Google Colab over Jupyter proved to be the free accessibility to GPU and TPU based computation, which provided a massive speedup especially in the image context, in which we had to train deep convolutional neural networks. One possible drawback is that the system limits the amount of computational resources allocated after a period of non-stop usage. Furthmore, if the user is inactive for more than 30 minutes, it disconnects automatically. For these reasons, Colab is not the way to go for long and heavy computations.

Chapter 6

Experiments

In this chapter, we will dive deep into the experiments we have performed on different datasets to show the results of our methodology and implementation. Following the structure of section 4, likewise this section is divided in two parts: we will first describe the results obtained from the implementation of the methodologies to perform data selection in a transfer learning scenario, and then the analysis will continue by illustrating the applications of the methodologies in the resolution of DEs with deep learning architectures.

6.1 Data Selection for Deep Transfer Learning in Image Recognition

This section will guide the reader through all the outcomes of our experiments, where different methodologies of data selection have been applied. We will first describe the experimental settings, including datasets, shifts and architecture used, and then we will illustrate the results of the techniques implemented.

6.1.1 Experimental Settings

Datasets

The datasets we have chosen are datasets of images, which require convolutional neural networks to perform classification tasks. Two of them are made of images of small size and with only one channel, which made them suitable to be used as input for feed-forward neural networks too. Here is the list of the datasets employed:

- **CIFAR-10:** the CIFAR-10 dataset [35] is is one of the most widely used datasets for machine learning research. It contains 60,000 32x32 color images in 10 different

classes. The 10 different classes represent airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. There are 6,000 images of each class. Classes are completely mutually exclusive (e.g. there is no overlap between automobiles and trucks).

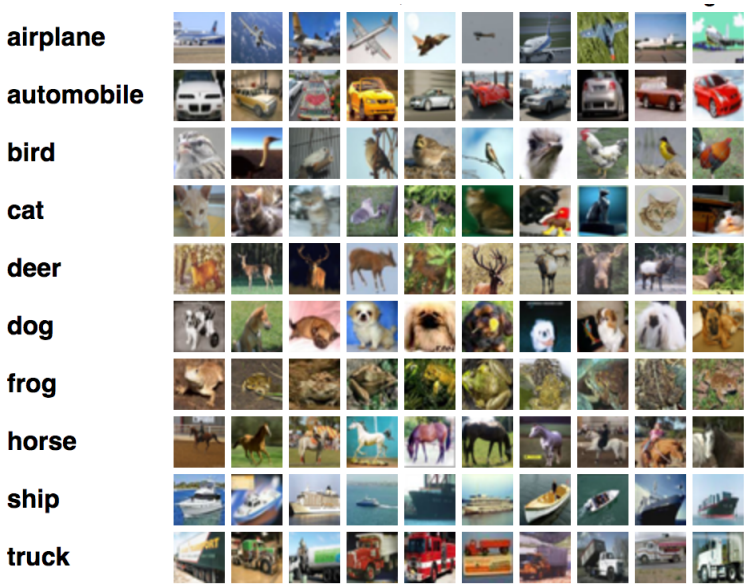


Figure 6.1 Samples images from the CIFAR-10 dataset.

- **MNIST** the MNIST database [42] is a large database of handwritten digits that is commonly used for training various image processing systems. It is a very lightweight dataset which is suitable for trying learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing. The dataset is made of digits written by high school students and employees of the United States Census Bureau which have been normalized to fit into a 28x28 pixel bounding box.

- **USPS** USPS [31] is an image database for handwritten text recognition research. It contains digital images of approximately 5000 city names, 5000 state names, 10000 ZIP Codes, and 50000 alphanumeric characters. Each image was scanned from mail in a working post office at 300 pixels/in in 8-bit gray scale. Furthermore, the database is divided into explicit training and testing sets to facilitate the sharing of results among researchers as well as performance comparisons.

Shifts

As stated in section 2.5, the most typical approach in a transfer learning scenario is using a pre-trained model on a source domain D_s and task T_s and use it as a starting point to

boost the learning of a target domain D_t and task T_t. In the images context, we adopted different types of image distortions in order to generate newly and artificially created datasets with different levels of severity. In this way, it was possible to arbitrarily manipulate the differences between source and target dataset, and therefore study the impact of transfer learning in a more controlled setting. In the following experiments, in order to present results as uniform as possible, we made use of just one type of distortion, which is explained in details below. The other types of distortion implemented are listed in appendix A.

- **Embedding shift**: the embedding shift uses an autoencoder, an architecture which is able to learn a compressed representation of the input in a latent space, called *embedding*. In the case of images, it learns a representation which has typically more than three channels and a reduced dimension. In order to generated a distorted image, we forwarded the pristine version as input to the autoencoder, we learnt its embedding and then we applied an additive shift to each value of the tensor. The decoder will try then to reconstruct the original image starting from a modified embedding, which will cause some prediction errors depending on the severity of the shift. Actually, the prediction errors are also present with no shift at all, as the reconstruction of the autoencoder cannot be pure: in this case, we will refer to it as *plain embedding shift*.

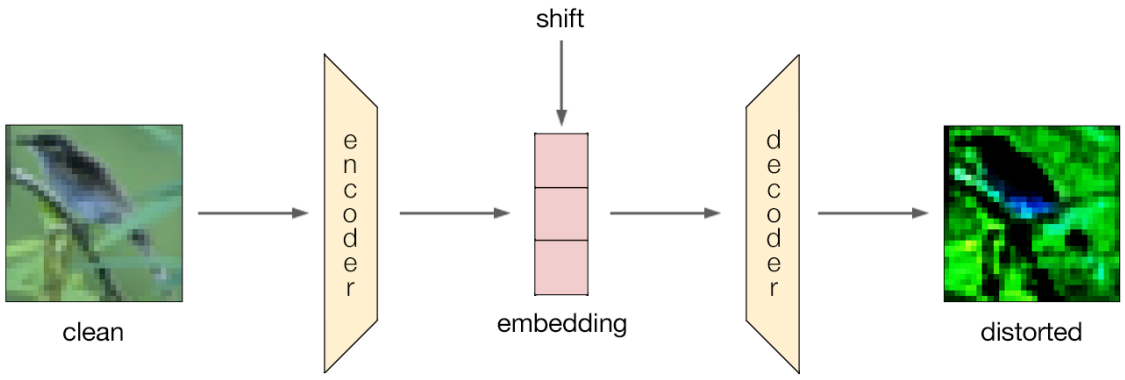


Figure 6.2 Embedding shift distortion mechanism. A clean image from the CIFAR-10 dataset with the label "bird" is fed into the autoencoder. A shift $c = 2$ is applied to the embedding, resulting in a distorted decoded image. Due to this distortion, we observe an average drop in the confidence of our models of 40%.

As mentioned in section 4, we will apply the illustrated techniques on another type of distribution shift, which is the covariate shift between the MNIST dataset and the USPS dataset. In fact, both of them are dataset containing hand-written digits, nonetheless a network well-trained on one of them will not be able to correctly classify also data points sampled from the other, due to the change in the covariates distribution.

Network Architectures

For this task, we implemented architectures with either only convolutional layers, or only feed-forward layers, or with both convolutional and feed-forward layers.

Due to the differences in terms of features and number of classes to predict among the different datasets adopted, we used mainly two architectures to test our approaches. For all the digit-based datasets and CIFAR-10 we used a simple convolutional neural network with two pairs of convolutional and max pooling layers, followed by a fully connected layer with a final softmax [18] layer. This architecture is very similar to the original *LeNet* developed by *LeCun et al.* in 1998 [41]. In figure 6.3 we can see the vanilla feed-forward neural network which was used for digit-based datasets, such as the MNIST.

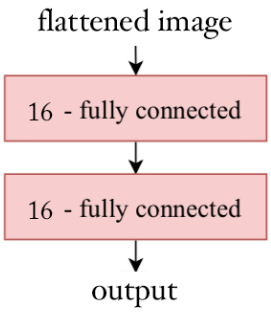


Figure 6.3 Network architecture for digit-based datasets. Fully connected layers are parameterized by *d*-fc, where *d* represents the dimensionality of the output space.

We chose to try this type of architecture as well because it proved to achieve good performances in the digit classification tasks. One-channel images have been flattened (and eventually resized) before feeding the network. The other network architecture employed can be observed in figure 6.4.

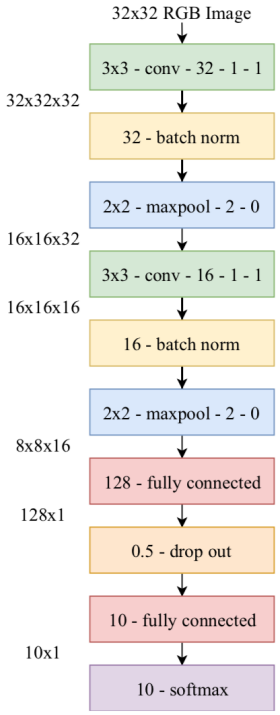


Figure 6.4 Network architecture used in the image recognition context for CIFAR 10 and digit-based datasets. Convolutional layers are parameterized by *kxk*-conv-*d*-*s*-*p*, where *kxk* is the spatial extent of the filter, d is the number of output filters in a layer, s represents the filter stride and p indicates the zero-padding. Max-pooling layers are parameterized as *kxk*-maxpool-*s*-*p*, where *s* is the spatial stride and *p* indicates the implicit zero padding. Batch normalization layers are parameterized by *d*-bn, where *d* is the number of features in the layer. Finally, fully connected layers are parameterized by *d*-fc, where *d* represents the dimensionality of the output space.

For what concerns the training phase, we chose cross-entropy loss for all the three architectures, minimized through Adam optimizer, with the standard hyperparameters stated in the original paper [33], and with a batch size of 64.

6.1.2 Baselines and Distortion Effect

The problem of image classification is vastly known, hence the discussion in this section will not be long. Here below we show the accuracy obtained by the training of the network on the CIFAR 10 dataset, on the training set and validation set. The patience hyperparameter for early stopping was set at 15 epochs.

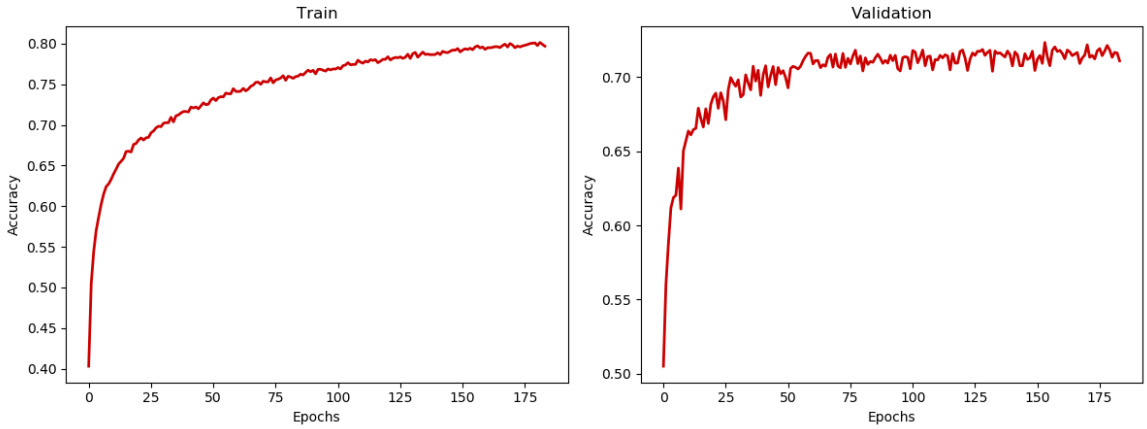


Figure 6.5 Accuracy trend of the training on clean CIFAR 10 dataset. *Left*: Training accuracy. *Right*: Validation accuracy.

We can see that the training stopped with a validation accuracy of about 70%: we then tested the model on a test set, getting a final accuracy of 71.35%.
On top of this result, we then applied a plain embedding shift on the test set and evaluated the accuracy again. The result was an accuracy on the distorted test set of 45.5%.

Here instead, we show the baseline training on the MNIST dataset. Also in this case we employed early stopping with a patience of 15 epochs.

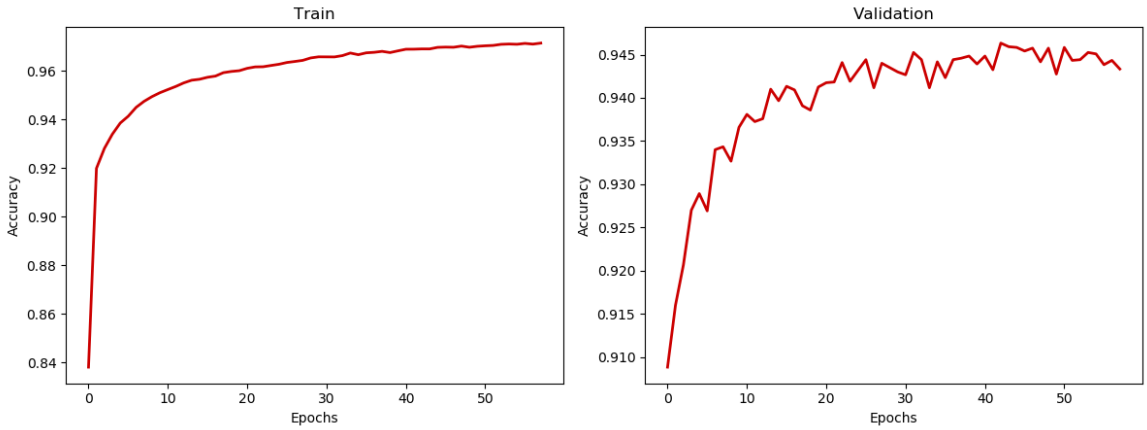


Figure 6.6 Accuracy trend of the training on MNIST dataset. *Left*: Training accuracy. *Right*: Validation accuracy.

We obtained a final test accuracy of 94.91% on the test set, and a drop to 41.2% when using the same baseline to classify the USPS dataset.

6.1.3 Results and Discussion

In the following sections we will show the results of the implementation of the methodologies described to perform optimized data selection. The discussion will continue by dedicating a specific section for each of the techniques applied, where we will explain in detail the applications of the procedures, giving explanations and comments about the outcomes.

In the experiments on the error-driven and entropy-driven approaches, we decided to make a selection (whose structure has been illustrated in section 4.1.3) with two specific reduction of the target dataset: 25% and 50%. Once the subset is extracted, we furtherly split it into an 80% training and 20% validation, in order to monitor the progress of the network and prevent overfitting.

Figure 6.7 In this figure, as an example, we see how we perform the selection of the subset: first, we sample 25% of the target dataset, according to a given criterion - e.g. top 25% for entropy, bottom 25% for entropy, random 25%. Then, we furtherly split the subset to get a validation set.

Error-driven Approach

In this section we will show the results obtained from our first approach implemented: the error-driven approach. As we described in section 4.1.3, the error-driven approach selects a subset of samples based on their single contribution to the loss function of the network to finetune. The rationale behind this approach is that critical points should be prioritized during training, in order to achieve a faster convergence and a higher accuracy. The assumption is that once the critical points have been corrected, the other ones should follow too.

We tested the proposed approach in two different scenarios, in which in both of them we applied the early stopping technique. The first scenario is a model pre-trained on a clean version of the CIFAR 10 dataset and finetuned on the distorted version of the same dataset obtained by applying a plain embedding shift. As shown in figures 6.8 and 6.9, we chose two specific reduction of the target dataset: 25% and 50%. The two figures exhibit the same

behavior: the random selection outperforms the top selection on the test set. Focusing on the training curve, we can see that the bottom selection basically does not learn anything, being stopped before the other two due to early stopping. It behaves as expected, since it is trained on the examples on which the model is correct. On the test set, the random outperforms the top selection, which has a very low accuracy. One possible explanation is that the top selection focuses too much on the examples which are wrong the most for the model, failing to capture the real distribution of the data. In order to understand deeply why this kind of selection was not working, we applied the approach to a two-dimensional synthetized dataset. We discovered that, in some extreme cases, the decision boundary learnt by the model was diametrically opposite to the right one, and therefore the final accuracy was very low.

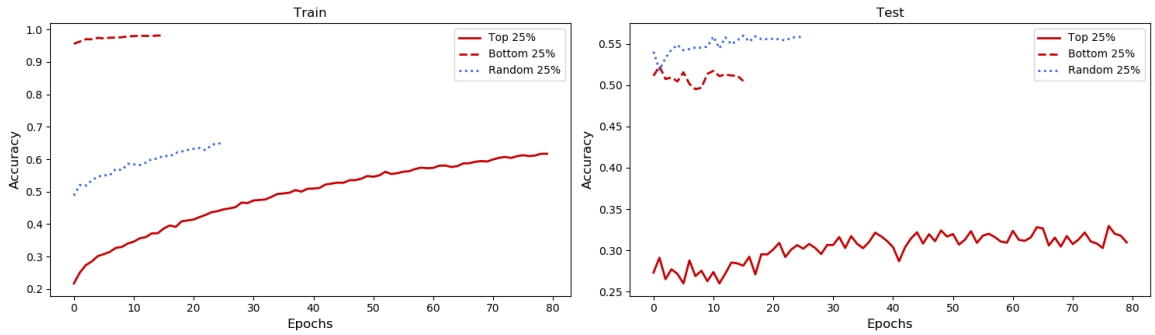


Figure 6.8 Accuracy trend of a finetuned model on distorted CIFAR 10, with plain embedding shift, retaining 50% of the dataset. Samples selected according to error-driven criterion.

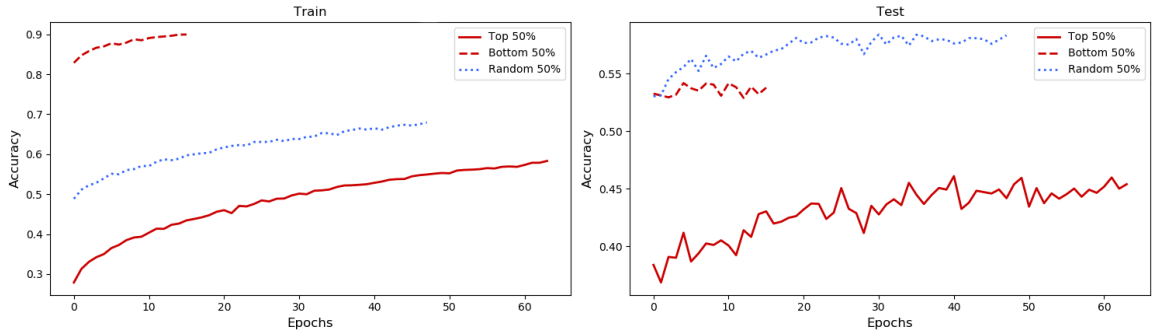


Figure 6.9 Accuracy trend of a finetuned model on distorted CIFAR 10, with plain embedding shift, retaining 50% of the dataset. Samples selected according to error-driven criterion.

The second scenario is a pre-trained model on the MNIST dataset and finetuned on the USPS dataset, retaining 50% of the dataset. As we can see in figure 6.10, the random outperforms again the top selection.

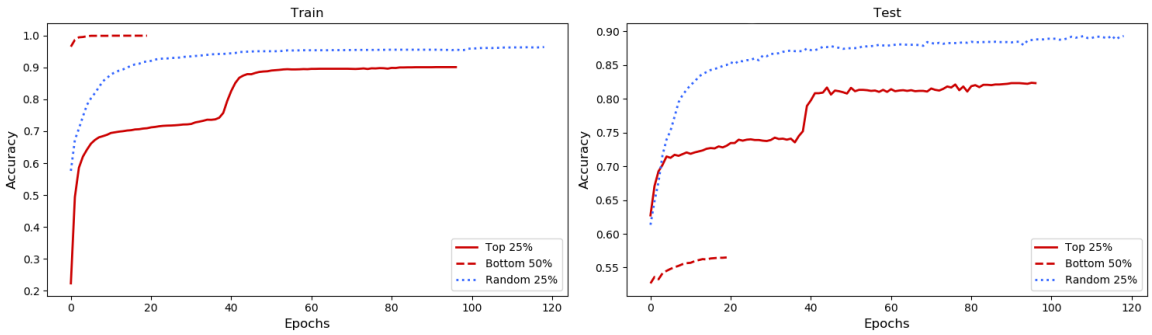


Figure 6.10 Accuracy trend of a pre-trained model on MNIST, finetuned on USPS, retaining 50% of the dataset. Samples selected according to error-driven criterion.

Entropy-driven Approach

We outlined the rationale behind entropy-driven approach in section 4. The entropy-driven approach proposes to finetune the baseline network with a selection of data guided by the confidence of the predictive model. Indeed, the assumption of this procedure is that the network should be more likely wrong when fed with those samples, as they are very close to the decision boundary, possibly in a region of the hyper-plane different from the one where all the other samples with the same label are located.

Here below we show the outcome of an experiment carried on a baseline model pretrained on a CIFAR 10 dataset, which is then distorted with a plain embedding shift, and used for finetuning. The trend of the accuracies on the training set and on the test set are shown below.

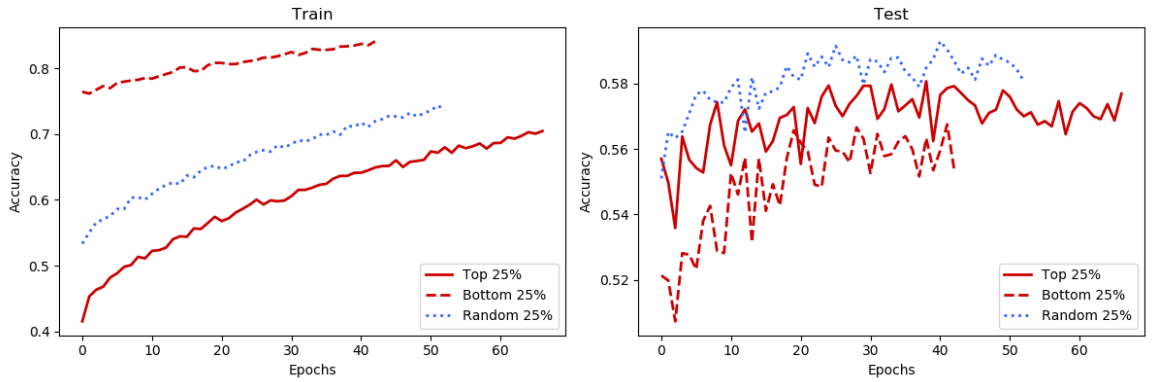


Figure 6.11 Accuracy trend of a finetuned model on distorted CIFAR 10, with plain embedding shift, retaining 25% of the dataset. Samples selected according to entropy-driven criterion.

In this experiment, we carefully selected the top 25% and the bottom 25% samples, according to their entropy, and compare the selection with a random 25% subset. Focusing on

the training curve, we can see that, as expected, the model finetuned with the most entropic samples is the one that starts from the lowest point, as the initialization of the network is not the best to classify those points. On the other hand, the bottom 25% selection yields a flatter training curve: these points are the ones that cause less troubles to the baseline model, hence the training starts from a very high accuracy.
Moving our attention to the test curve, the situation is very different: the bottom 25% curve is the worst at generalizing over all the dataset, performing significantly worse than the others. On the other hand, despite not having the best performance in training, the random selection is the one that outperforms the others.
Given the strong reduction of the dataset size, there is a significant difference in the prediction accuracy between training and test, for all the three selections, caused by overfitting. Nonetheless, similar conclusions can be made when enlarging the retained subset size to 50%.

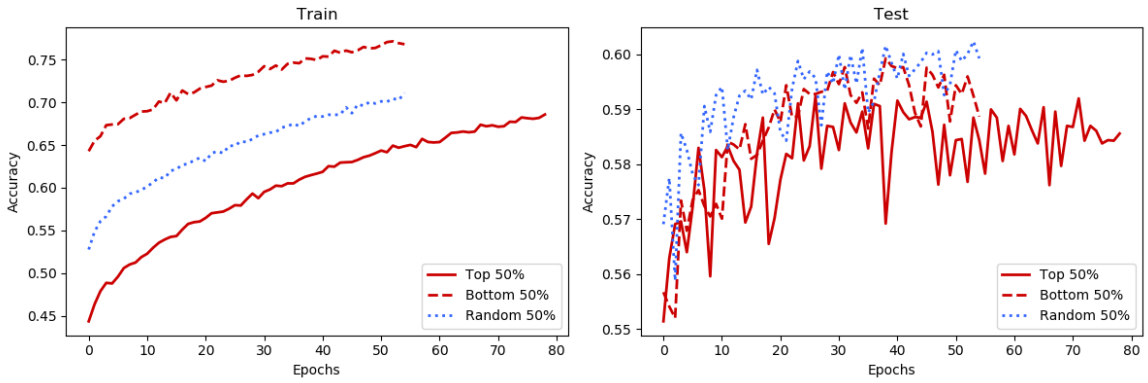


Figure 6.12 Accuracy trend of a finetuned model on distorted CIFAR 10, with plain embedding shift, retaining 50% of the dataset. Samples selected according to entropy-driven criterion.

Although the overfitting effect is less evident, training with the most entropic samples does not lead to any significant benefit. It is also noticeable a closer similarity of performance between the top 50% and bottom 50% curves: as the size of the subset goes increasing, the difference between the two subsets decreases.

Similar results are obtained when we use the baseline pre-trained on the MNIST dataset and finetuned on the USPS dataset. Here below we show an experiment where the 50% of the samples are retained.

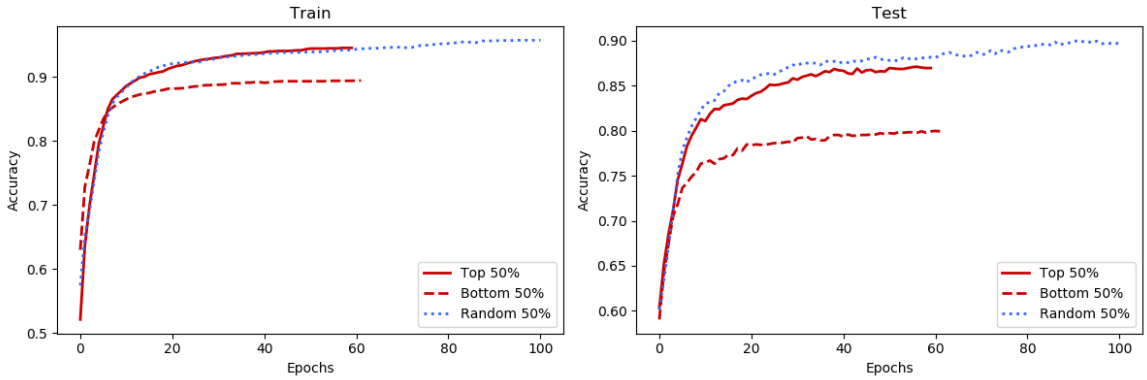


Figure 6.13 Accuracy trend of a finetuned model on USPS dataset, retaining 50% of the dataset. Samples selected according to entropy-driven criterion.

Also in this case, despite a smoother and clearer improvement in the test set, still the model trained with the most entropic samples does not outperform the random selection. These experiments prove the partial ineffectiveness of sampling points according to their classification entropy, which is a widely used approach in literature, in this specific context of image recognition.

Subset recomputation

The further step that we took was driven by the following intuition: if the decision boundary of a network moves in the hyperspace, the most entropic samples will change as well, as the training goes on. In order to tackle this change, we modified the approach to take into account the network's hyperplane change, by recomputing the entropy-driven subset multiple times in the training procedure. As highlighted in section 4, this is done once every T epochs, training for N_{epochs}.

Here below we show the results of an experiment made with the same settings of figures 6.11, 6.12. We set $N_{epochs} = 50$, meaning the training takes 50 epochs, and $T = 5$, meaning every 5 epochs the subset is recomputed.

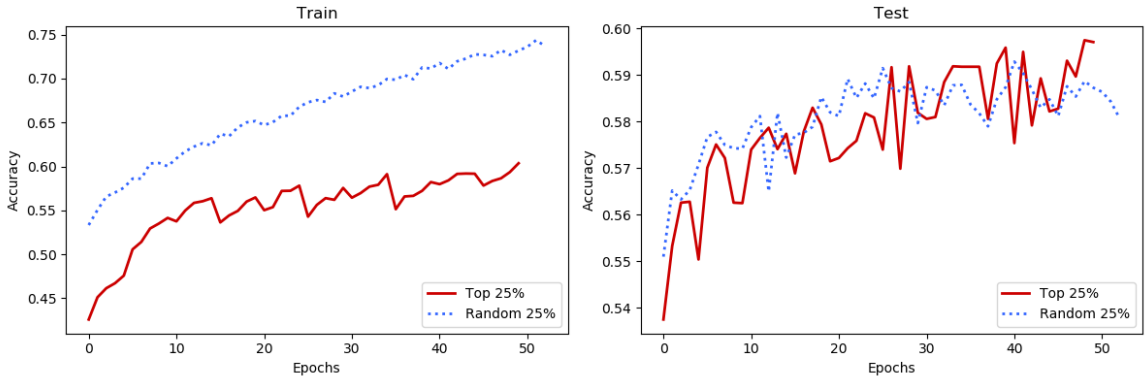


Figure 6.14 Accuracy trend of the entropy-driven approach on CIFAR 10 dataset, with subset recomputation - 25%.

We compare this approach with the same random sampling of figure 6.11. We observe a similar behaviour in the training curve, with noticeable accuracy drops every 5 epochs, due to the resampling, that get steeper and steeper as the model is finetuned: the change of the subset causes a shift in the distribution to learn from the network and a change in the shape of the loss function. We tried with different values of T, with no significant differences or improvements.
With respect to training with a fixed subset, we notice a sensible difference in the test accuracy curve, that reaches a higher accuracy. Nonetheless, this method still does not outperform the selection of a random subset of the data: they look like having the same generalization error, but this procedure is more computationally expensive, as the time complexity of recomputation of the subset is not negligible.

Different results are instead obtained with the finetuning on the USPS dataset. Here we show the results of the subset recomputation (25% and 50%) every 5 epochs, with a total training epochs of 50, comparing it with a random selection trained with early stopping.

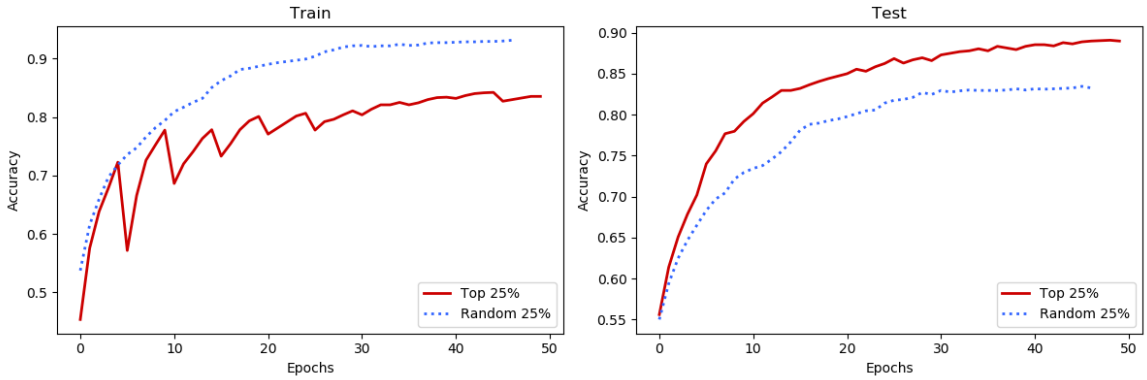


Figure 6.15 Accuracy trend of the entropy-driven approach on USPS dataset, with subset recomputation - 25%

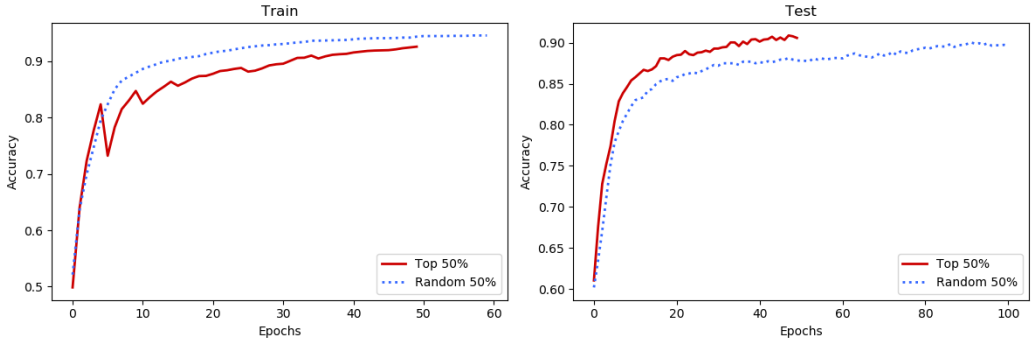


Figure 6.16 Accuracy trend of the entropy-driven approach on USPS dataset, with subset recomputation - 50%.

In both the training curves of figures 6.15, 6.13 we can see significant drops, every time the subset is recomputed. However, in this case the test curve shows better outcomes: not only the recomputation of the subset reaches higher accuracy, but it is also faster than the random selection at getting those results, even with just the 25% of the dataset.

The reason why this happens is that, in this case, the network is smaller, hence it does not overfit on the single subsets in the few epochs of sub-training, thus giving an overall better generalization error. This network architecture, in fact, is way less parametrized than the Convolutional Network used in the previous experiments. Indeed, cutting the dataset so sharply will hinder the performance of a large model, namely with a great number of parameters, that in principle needs a lot of data to generalize.

As a conclusion, despite leading to better final outcomes in some cases, this method does not look generalizable for all the possible problems. Furthermore, it is not guaranteed to

converge: there will always be a subset of most entropic samples to draw every N epochs, that will cause strong oscillations in the accuracy of the training set (or validation).

Differential Approach

The last method we explored is the one described by algorithm 4. This procedure is meant to explore the target training set in order to search for the samples which will expectedly improve the generalization error of your network. This is done by computing, for each training sample, the following quantity, with $X_{t,i}$ coming from the target training set, $V_{t,j}$ coming from the target validation set:

$$\sum_j I_{loss}(X_{t,i}, V_{t,j}), \text{ where } I_{loss}(X_{t,i}, V_{t,j}) = \nabla_\theta L(X_{t,i}), \nabla_\theta L(V_{t,j})$$

I_{loss} is a measure of similarity between the gradients of the loss computed in the points $X_{t,i}$, $V_{t,j}$. Therefore fixing a training sample $X_{t,i}$ and computing the sum of all $I_{loss}(X_{t,i}, V_{t,j})$ will measure if its impact on the loss will be beneficial to minimize the loss of the validation set. Indeed, we only retain a training sample $X_{t,i}$ if $\sum_j I_{loss}(X_{t,i}, V_{t,j}) > 0$.
Since the proposed method entails a significant computational complexity, both in time and space, in order to test this selection criterion, we worked with a reduced CIFAR 10 dataset, keeping only 4 classes ("airplane", "automobile", "bird" and "cat"), hence having a total number of samples of 16000. We had to retrain the architectures previously outlined on a clean dataset, obtaining a final test accuracy of 87%.
For this experiment, we distorted the dataset with an embedding shift, with $c = 2$, resulting in a new target test accuracy of 61%. Once again, we split the dataset in a 80%-20% training and test, and then another equal split for the training, to get a validation set. We then used the target training set to perform the selection: out of 10240, 9849 were retained.
To compare the results, we show here the train and validation curves of a model finetuned for 40 epochs, with samples selected with the differential criterion, compared with a random selection containing the exact same amount of samples.

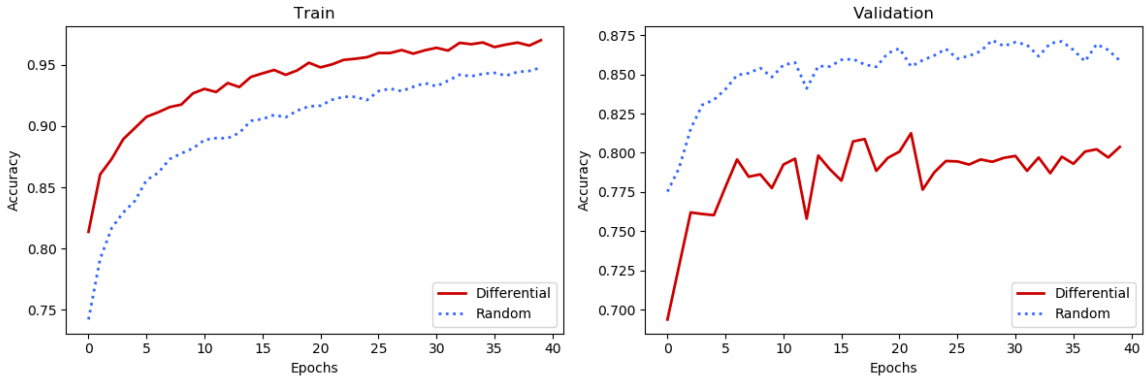


Figure 6.17 Accuracy trend of differential approach on distorted CIFAR 10.

Apparently, this selection leads to a result which is diametrically opposed to the expectations: the differential-driven subset has a good performance on the training set, both in terms of convergence and absolute value of accuracy, resulting instead in a modest result in the validation set: once again, a random subset looks like better capturing the distribution of the data and has a better generalization error.

Similar results are obtained when learning the USPS distribution. The procedure was exactly the same, with a total of 3982 samples retained, out of 5832.

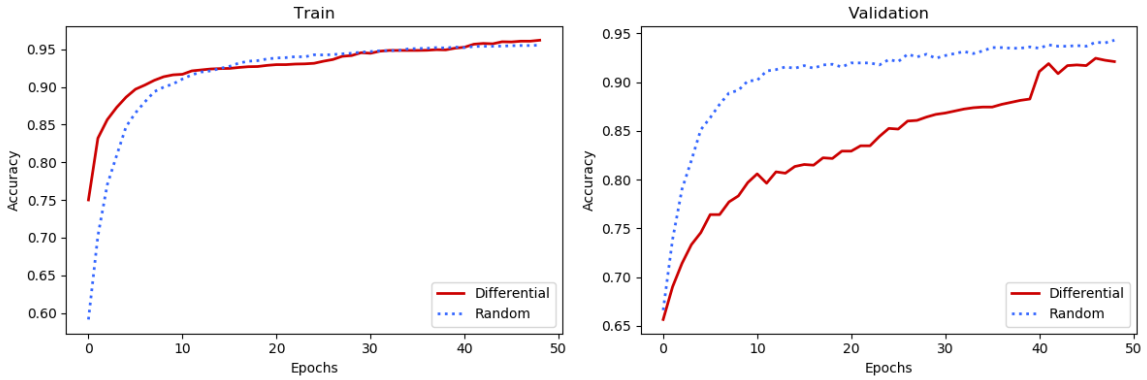


Figure 6.18 Accuracy trend of differential approach on USPS.

Indeed, this selection criterion only takes into account one training sample at a time, disregarding the effect on the loss function when optimized through multiple training samples all together, which is what happens when training with batched gradient descent. Hence this will lead to an approximate selection which will not actually capture the relations between multiple data points, resulting in poor generalization performance.

6.2 Deep Transfer Learning for Differential Equations

The discussion will proceed by explaining in detail how the approaches of section 4.1.3 have been implemented on the two problems: Nonlinear oscillator and SIR model.
Following the scheme of the previous section, we firstly illustrate the experimental settings, which are significantly different than the image recognition ones, and then we will dive into the results of the research.

6.2.1 Experimental Settings

Datasets

One of the main characteristics of this problem is that it does not require data at all. As you can see from the architectures shown in chapter 4, the inputs of the networks are time-steps, initial conditions and parameters. Nonetheless, these are just numbers that we can draw from a pre-specified interval, different for each one of the inputs.
It is intuitive that, the bigger each interval is, the more points the network needs to be able to generalize over all the interval. More details will be given when showing the effective results of our experiments. To have an additional confirm of the validity of the methods, along with the solution learnt by the networks, we will oftentimes plot also the solution provided by the function `odeint` of the Scipy library [10], a well known and efficient solver for Ordinary Differential Equations.

Distortions

Differently from the image recognition case, in this setting we do not have any special distortion, but just a vector space shift of initial conditions or parameters, that vary according to the problem.

- **Nonlinear oscillator**: in the phase space of the nonlinear oscillator the initial conditions correspond to the total initial energy of the system. For values of $x(0)$ and $p(0)$ between 0 and 1 the motion of the system is similar to the one of the simple harmonic oscillator, with elliptical trajectories. Moving out from this region, the motion starting deviate from the behavior of the simple harmonic oscillator and the trajectories shapes are closer to rhombuses. It is worth mentioning that initial conditions in the harmonic region are easier to learn for the network, i.e. the convergence is faster. Therefore, since the difficulty of learning a trajectory starting from an initial point is not the same

for all points in the phase space, the same amount of perturbation added to a given initial condition can lead to different scenarios.

- **SIR model**: for the SIR case, the initial conditions correspond to the distribution of the population at the start of the epidemic: going towards high $\mathbf{I}(0)$ will result in a greater number of infected people at time-zero, vice-versa happens when going towards low $\mathbf{I}(0)$. Instead, shifting the parameters means changing the basic reproduction ratio R_0: the combined shift of β and γ will determine if, in the target task, the virus will behave as in the source task or not. In particular, as already mentioned, β models the contagion rate of the virus and γ is the recovery rate, therefore going towards high β and small γ will result in a severe epidemic in the considered time interval, vice-versa will result instead in a smoother situation where the virus spreads slowly and the people recover fast.

Network Architectures

Solving differential equations which describe dynamical systems is a very different task with respect to image classification. In this case, the setting is completely unsupervised: we are feeding the network sampling time instants from a specified time interval, and therefore we do not need convolutional layers anymore, since data do not exhibit a grid-like topology as images do. Our proposed architecture in this context is a simple feed-forward neural network with 2 and 4 hidden layers with 50 neurons each, and a number of inputs and outputs that may vary according to the task to solve. Concerning the activation functions, for the networks used to solve the Nonlinear oscillator equations we used the *sin* activation function, whereas we used the *sigmoid* for the SIR system.

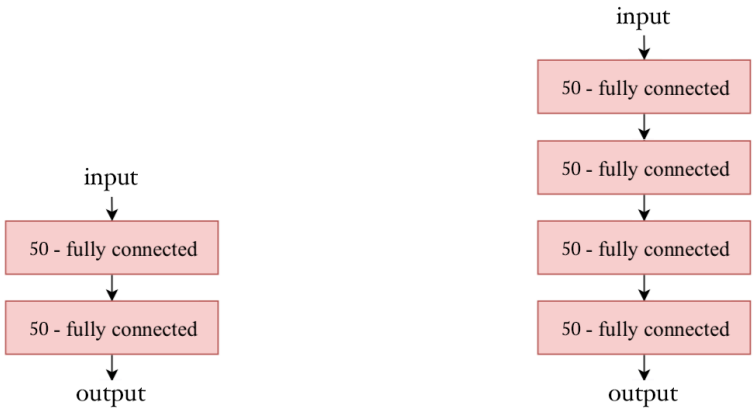


Figure 6.19 Networks used in the differential equation context. Fully connected layers are parameterized by d-fc, where d represents the dimensionality of the output space. *Left:* network with 2 hidden layers with 50 neurons each. *Right:* network with 4 hidden layers with 50 neurons each.

6.2.2 Baselines and Perturbation effect

In the following we will show the results of the baselines training for the two problems considered. Please notice that in this case the only quantity that is interesting is merely the loss function: the lower it is, the better the network is good at approximating the exact solution of the equation. Furthermore, as already mentioned, the loss function is different between the two problems, as the equations to solve are different.
For each problem we will show baseline results for learning on **fixed initial conditions**: in this setting, the only input of the network is the time step t, the initial condition is fixed and enforced by the parametrization $\mathbf{z}(t) = \mathbf{z}(0) + (1 + e^{-t})\mathbf{N}(t)$. Consequently, we will show numerically and visually the impact of the perturbation of the initial conditions on the solution provided by the neural model.

Nonlinear oscillator

In section 4.2.4 we introduced the dynamical system of the nonlinear oscillator, its Hamiltonian, and the system of differential equations that govern it. The network adopted to solve the problem is shown in figure 6.19 on the left, The phase space of the oscillator consists of two degrees of freedom, with $\mathbf{z} = (x, p)^T$. Accordingly, we adopted the feed-forward neural network shown in figure 6.19 with 2 hidden layers with 50 neurons each. The network has two outputs $\mathbf{N} = (N_1, N_2)^T$ used to parametrize the approximate solutions $\hat{\mathbf{z}} = (\hat{x}, \hat{p})^T$. The loss function L is defined according to the equations of motion:

$$\begin{cases} \dot{x} = p \\ \dot{p} = -(x + x^3) \end{cases}$$

and it is expressed in the following form:

$$L = \frac{1}{K} \sum_{n=0}^{K} \left[\left(\dot{\hat{x}}^{(n)} - \hat{p}^{(n)} \right)^2 + \left(\dot{\hat{p}}^{(n)} + \hat{x}^{(n)} + \left(\hat{x}^{(n)} \right)^3 \right)^2 \right]$$

In order to train the network, we initialized a grid with $K = 200$ time points equally spaced in the time interval $t = [0, 4\pi]$. At the beginning of each epoch, we perturb all the time points by using a random term obtained by a normal distribution with zero mean and a standard deviation of 0.06π. In this way it is like we are continuously sampling in the time interval, helping the network discovering the true trajectory in the phase space. The network was trained for $5 \cdot 10^4$ epochs by using Adam optimizer [33] with a fixed learning rate of $8 \cdot 10^{-4}$.

Here we will show the trajectories in the phase space found by the network at the end of the optimization process, as long with the logarithm of the loss function during the training phase. We chose to show two solutions of two different initial conditions, one in the linear oscillator case ($\lambda = 0$) and one in the nonlinear oscillator case ($\lambda = 1$). The chosen configurations C_1 and C_2 are the following:

- $C_1 : \mathbf{x}(0) = 1.0, \mathbf{p}(0) = 1.0$
- $C_2 : \mathbf{x}(0) = 4.0, \mathbf{p}(0) = 2.5$

In figure 6.20 we can observe the trajectory of the solution of the linear oscillator starting from configuration C_1 and the trend of the LogLoss during training.

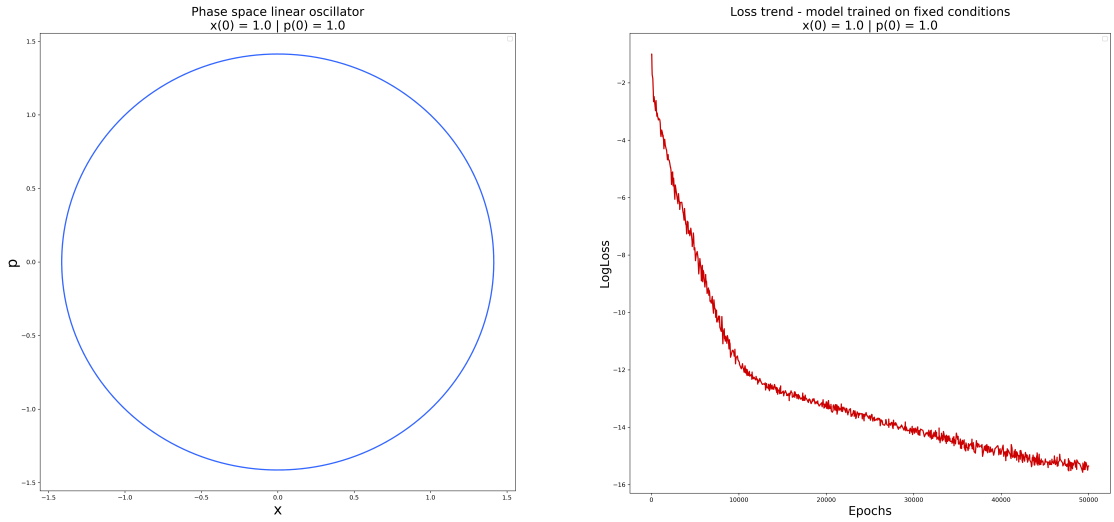


Figure 6.20 Linear oscillator $(\lambda = 0)$ trained starting from initial conditions $\mathbf{x}(0) = 1.0, \mathbf{p}(0) = 1.0$. *Left:* trajectory of the solution in the two-dimensional phase space. *Right:* logloss trend for $50K$ epochs.

In the linear oscillator case, the trajectory in the phase space is always a circle, with a radius determined by both x and p. The energy of the system, in fact, is represented by the area of the trajectory, and, as stated in in section 4.2.4, the Hamiltonian represents the energy of the system as well. For what concerns the LogLoss, at the end of the training it reaches a value around -15, which corresponds to a real loss around $1 \cdot 10^{-7}$, a very good result.

In figure 6.21 we can observe the results of training the network in the nonlinear oscillator case, starting from configuration C_2. The trajectory has an elliptical shape in center region and then it starts diverging. The LogLoss at the end of the training is higher with respect to the linear case. This is because the problem to solve is more difficult and the network struggles to learn this more complex trajectory.

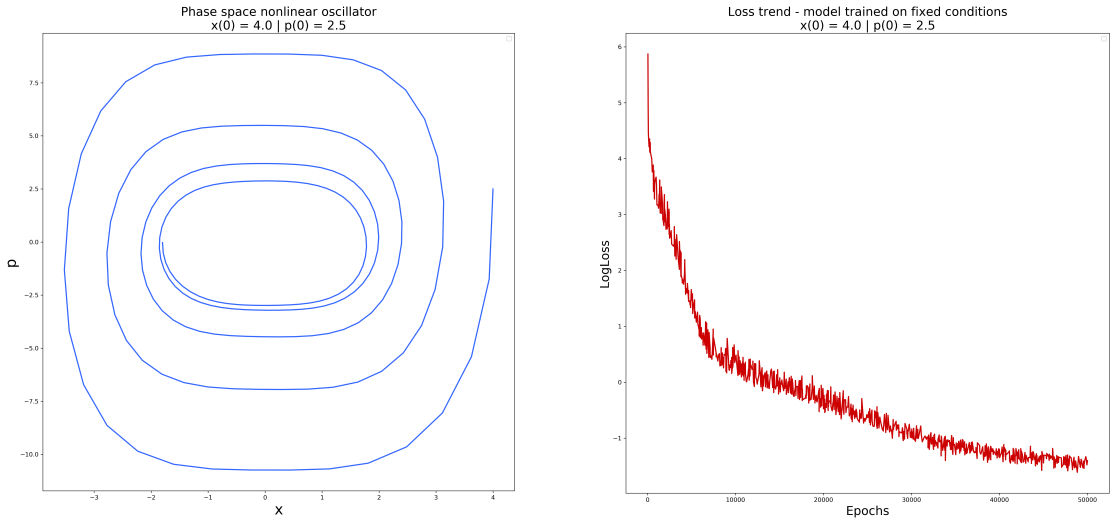


Figure 6.21 Nonlinear oscillator ($\lambda = 1$) trained starting from initial conditions $\mathbf{x}(0) = 4.0, \mathbf{p}(0) = 2.5$. *Left:* trajectory of the solution in the two-dimensional phase space. *Right:* LogLoss trend for $50K$ epochs.

In the previous paragraphs we showed how the neural network to solve the nonlinear oscillator problem has been trained. Now we show how perturbing the initial conditions deteriorates the accuracy of the solution. Here below we plot the solutions obtained by a pre-trained model on configuration C_3 on two configurations in which initial conditions have been perturbed:

- $C_3 : \mathbf{x}(0) = 1.5, \mathbf{p}(0) = 1.5 \rightarrow \tilde{C}_3 : \mathbf{x}(0) = 1.6, \mathbf{p}(0) = 1.6$
- $C_3 : \mathbf{x}(0) = 1.5, \mathbf{p}(0) = 1.5 \rightarrow \tilde{C}_3 : \mathbf{x}(0) = 2.0, \mathbf{p}(0) = 2.0$

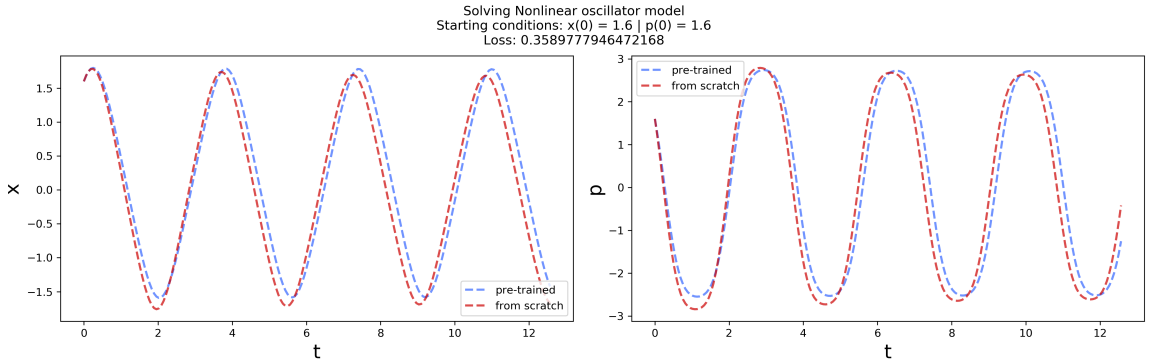


Figure 6.22 Network perturbed solution for the Nonlinear oscillator model, with $\mathbf{x}(0) = 1.6, \mathbf{p}(0) = 1.6$.

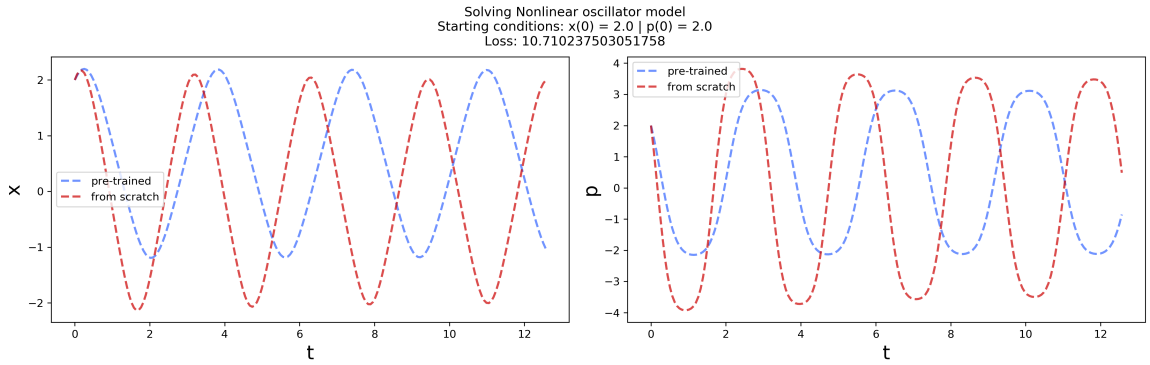


Figure 6.23 Network perturbed solution for the Nonlinear oscillator model, with $\mathbf{x}(0) = 2.0, \mathbf{p}(0) = 2.0$.

We can see clearly in figure 6.22 and 6.23 how different levels of severity in the perturbation affect the solution of the pre-trained model. We used a model trained from scratch on the perturbed configurations as a ground truth for the comparison of the results. We can see also how the loss of the model increases dramatically from the first to the second case, due to the huge perturbation applied.

SIR model

For the resolution of this system of DE the network architecture is basically the same, with 2 layers and 50 neurons each. The loss function to be minimized L is defined according to the dynamics of the system:

$$\begin{cases} \dot{S} = -\frac{\beta SI}{N} \\ \dot{I} = \frac{\beta SI}{N} - \gamma I \\ \dot{R} = \gamma I \end{cases}$$

$$\mathrm{N = S + I + R}$$

and expressed in the following form:

$$L = \frac{1}{K}\sum_{n=0}^{K}\left[\left(\dot{\hat{S}}^{(n)} + \frac{\beta \hat{S}^{(n)}\hat{I}^{(n)}}{N}\right)^2 + \left(\dot{\hat{I}}^{(n)} - \frac{\beta \hat{S}^{(n)}\hat{I}^{(n)}}{N} + \gamma \hat{I}^{(n)}\right)^2 + \left(\dot{\hat{R}}^{(n)} - \gamma \hat{I}^{(n)}\right)^2\right] \quad (6.1)$$

where n represents a time step in the interval and each squared term represents one of the three terms of the dynamical system.

To make it easier to learn, and without loss of generalization, we assumed that the three variables S, I, R are percentages, therefore we will refer to them as relative amount of the total population, hence fixing $N = 1$.
Moreover, in the training, we have always fixed $\mathbf{R}(0) = 0$ to describe better real epidemics phenomena, where, at time zero, there are usually no recovered people. Nonetheless, all the experiments are replicable relaxing this constraint.
From this and from the fact that the sum of the three variables must be 1, in all the experiments we have just fixed (or varied) $\mathbf{S}(0)$, changing the number of infected accordingly as $\mathbf{I}(0) = 1 - \mathbf{S}(0)$.

For the training of the network, we sampled $K = 2500$ time points equally spaced in the time interval $t = [0, 20]$, also perturbing the points at each epoch with the same noise used for the Nonlinear oscillator case, raising the standard deviation to 0.15π. For this case, the convergence of the network was quite fast and required training for just $1 \cdot 10^3$ epochs, by using Adam optimizer [33], with a fixed learning rate of $8 \cdot 10^{-4}$. Here we show two solutions, for two combinations of initial conditions and parameters, summarized in configurations C_1 and C_2:

- $C_1 : \mathbf{S}(0) = 0.8, \mathbf{I}(0) = 0.2, \mathbf{R}(0) = 0, \beta = 0.8, \gamma = 0.2$
- $C_2 : \mathbf{S}(0) = 0.3, \mathbf{I}(0) = 0.7, \mathbf{R}(0) = 0, \beta = 0.2, \gamma = 0.5$

We will show the solution found by the optimization, along with the LogLoss, so that the results are easier to visualize and understand.

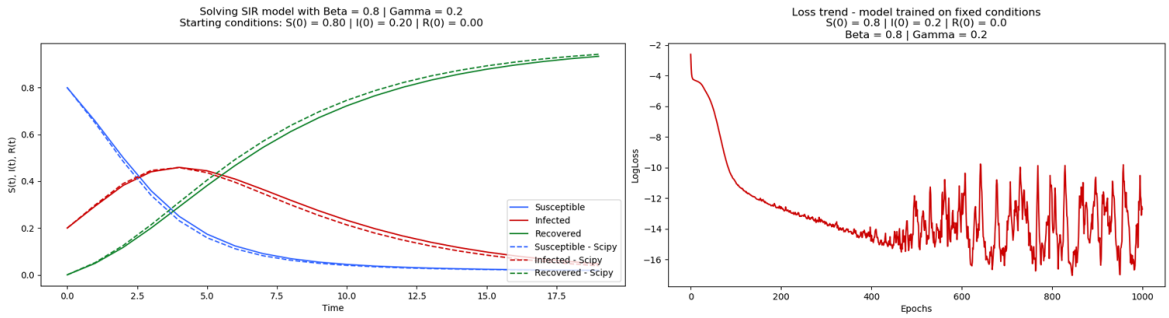


Figure 6.24 Network training for the SIR model, with configuration C_1. *Left*: Solution found by the network. *Right*: Loss trend of the training-

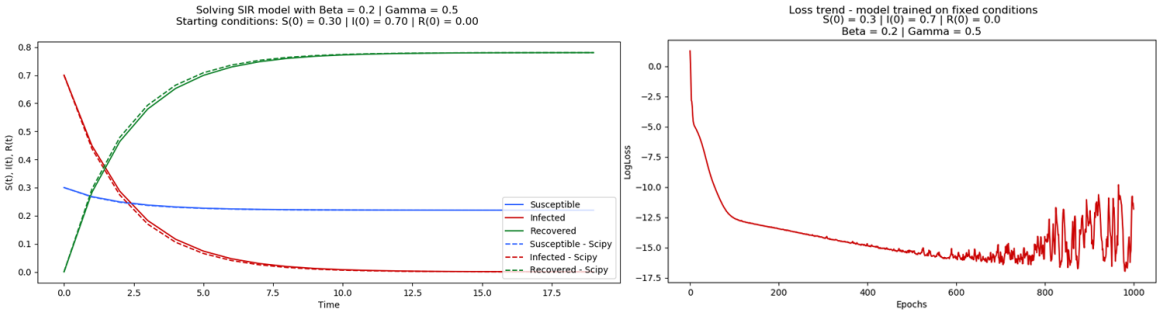


Figure 6.25 Network training for the SIR model, with configuration C_2. *Left*: Solution found by the network. Right: Loss trend of the training.

In figures 6.24, 6.25, we plotted our solution together with the one provided by Scipy `odeint`, displayed in dashed lines: as you can see, they are quite close and almost everywhere overlapping. Indeed, both the loss functions reach a LogLoss around -13, which means a real loss around $1 \cdot 10^{-6}$, that is a really accurate result. From both the plots 6.24, 6.25, we see what we expected from the description of the parameters: figure 6.24 shows a peak of the virus, given that there is a consistent number of infected at time zero and β is large, instead in figure 6.25 we encounter a progressive disappearance of the virus, with no outbreaks, due to the larger γ.
The oscillations of the loss functions are due to the optimization algorithm of Adam, which is not stable when the gradients get really small, as it divides them by the exponential moving average of the their square, which may tend to zero and make the updates of the weights explode, escaping from the minima. To overcome this issue, we kept track of the model with the best loss during the training and saved it.

Now that the baselines have been illustrated, we will show how perturbing the initial conditions impact heavily on the accuracy of the solution. Here below we show the solutions of the system provided by a model trained on configuration C_1, when initial conditions are perturbed as follows (β, γ remain the same):

- $C_1 : \mathbf{S}(0) = 0.8, \mathbf{I}(0) = 0.2, \mathbf{R}(0) = 0 \rightarrow \tilde{C}_1 : \mathbf{S}(0) = 0.78, \mathbf{I}(0) = 0.22, \mathbf{R}(0) = 0$
- $C_1 : \mathbf{S}(0) = 0.8, \mathbf{I}(0) = 0.2, \mathbf{R}(0) = 0 \rightarrow \tilde{C}_1 : \mathbf{S}(0) = 0.7, \mathbf{I}(0) = 0.3, \mathbf{R}(0) = 0$

where the first one corresponds to a perturbation of 2.5% on the initial conditions, while the second one is a perturbation of 12.5% (computed with respect to $\mathbf{S}(0)$, since $\mathbf{I}(0) = 1 - \mathbf{S}(0)$ and $\mathbf{R}(0) = 0$.

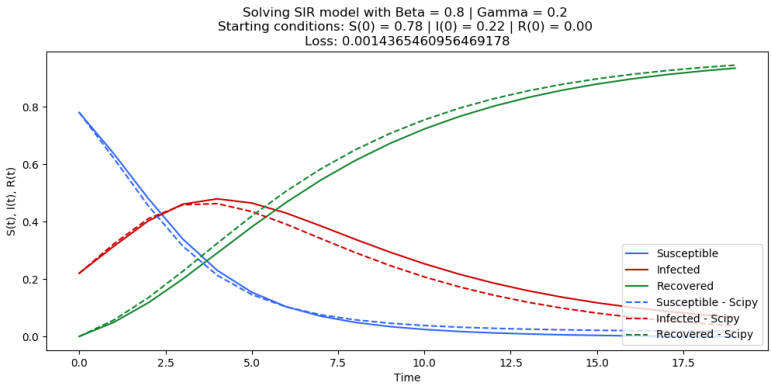


Figure 6.26 Network perturbed solution for the SIR model, with $\mathbf{S}(0) = 0.78, \mathbf{I}(0) = 0.22, \mathbf{R}(0) = 0.0$, $\beta = 0.8, \gamma = 0.2$.

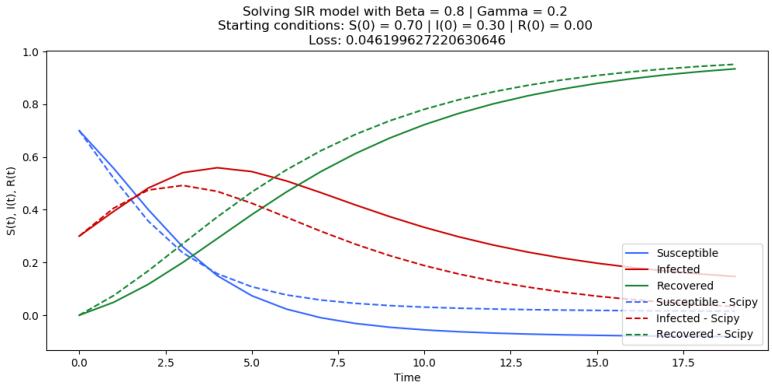


Figure 6.27 Network perturbed solution for the SIR model, with $\mathbf{S}(0) = 0.7, \mathbf{I}(0) = 0.3, \mathbf{R}(0) = 0.0$, $\beta = 0.8, \gamma = 0.2$.

As expected, the solution are visually wrong - especially compared to the one provided by the Scipy library. Furthermore, the impact of the distortion is quantifiable by looking at the values of the loss yielded by those solutions: for a perturbation of 2.5% we result in a loss of order $1 \cdot 10^{-3}$, while for a perturbation of 12.5% we even end up having a loss of $1 \cdot 10^{-2}$. This proves that, even with small perturbations, the network is not able to generalize, as the values of the loss are too high to consider the system as solved.

6.2.3 Results and Discussion

In the following sections we will apply the methodologies illustrated in chapter 4, showing their effectiveness both with the Nonlinear oscillator DEs and the SIR model. In particular, we will firstly show the effectiveness of transfer learning on the basic scenario, namely when the network is trained on fixed initial conditions.

Consequently, we will show how the network can be used to learn solutions of multiple Cauchy problems and how transfer learning techniques can be used to *move* in the space of all the possible solutions.
To give more details, we will vary the initial conditions both for the Nonlinear oscillator and the SIR model. Additionally, for the latter we will also vary the parameters of the system.
In section 4.2 we mentioned the possibility of generalizing the solutions over a set of boundary conditions and parameters, hence, for a more clear discussion, we define these two settings:

- **Bundle of initial conditions**: in this setting, the inputs of the network are the time step t and an initial condition $\mathbf{z}(0)$ drawn an interval (or bundle) $[\xi_{min}, \xi_{max}]$, hence the initial condition is variable..
- **Bundle of initial conditions and parameters**: in this setting, the inputs of the network are the time step t, an initial condition $\mathbf{z}(0)$ drawn from an interval (or bundle) $[\xi_{min}, \xi_{max}]$, and a set of parameters θ, drawn from an interval (or bundle) $[\theta_{min}, \theta_{max}]$. The initial condition is variable, likewise the coefficients of the system of DEs, also coming as input, therefore the loss function will have variable coefficients too.

Nonlinear oscillator

For the Nonlinear oscillator system, we decided to finetune starting from a pre-trained model on condition C_1. In this initial condition, the behavior of the oscillator is similar to the one of the linear oscillator, leading to trajectories with the shape of ellipses. We will then shift to two different conditions, which differ by the severity of the distortion applied to C_1. The two shifts can be summarized as follows:

- $C_1 : \mathbf{x}(0) = 1.0, \mathbf{p}(0) = 1.0 \rightarrow \ \mathrm{C}_2 : \mathbf{x}(0) = 1.05, \mathbf{p}(0) = 1.05$
- $C_1 : \mathbf{x}(0) = 1.0, \mathbf{p}(0) = 1.0 \rightarrow \ \mathrm{C}_3 : \mathbf{x}(0) = 2.5, \mathbf{p}(0) = 2.0$

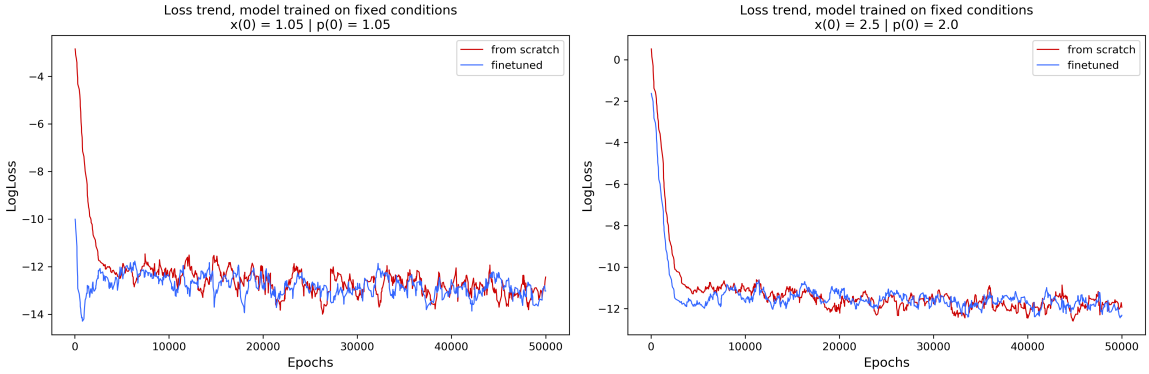


Figure 6.28 Scratch and finetuning comparison for nonlinear oscillator on fixed initial conditions. *Left*: networks trained on C_2. *Right*: networks trained on C_3.

Figure 6.28 above shows the loss trend of training from scratch compared to finetuning in the two aforementioned cases. In the first case, the distortion is very small, and it is equally applied to both the dimensions. We can observe that finetuning starts from a lower loss and therefore can help to converge faster. In the second case, two types of more severe distortions are applied independently on the two dimensions. We can see how the two lines are closer to each other, meaning that the impact of finetuning is less evident compared to the first case, but it still can help. We can observe that the final loss in the first case is slightly lower then the one of the second case. This is due to the fact that condition C_3 has a more complex trajectory with respect to C_2, and therefore the problem is intrinsically more difficult for the network.

Bundle of initial conditions

In this setting, we trained the network to learn a predefined bundle of initial conditions in a jointly manner. The training of the network is performed feeding the network with randomly sampled initial conditions from the predefined bundle, along with perturbed time steps, as explained in details in section 4.2.3. We show the effectiveness of the method by plotting the solution of the network on an initial condition selected inside the aforementioned bundle. Specifically, the model was trained on the following bundle of initial conditions:

$$C_4 : \mathbf{x}(0) \in [1.0, 1.2], \mathbf{p}(0) \in [1.0, 1.2]$$

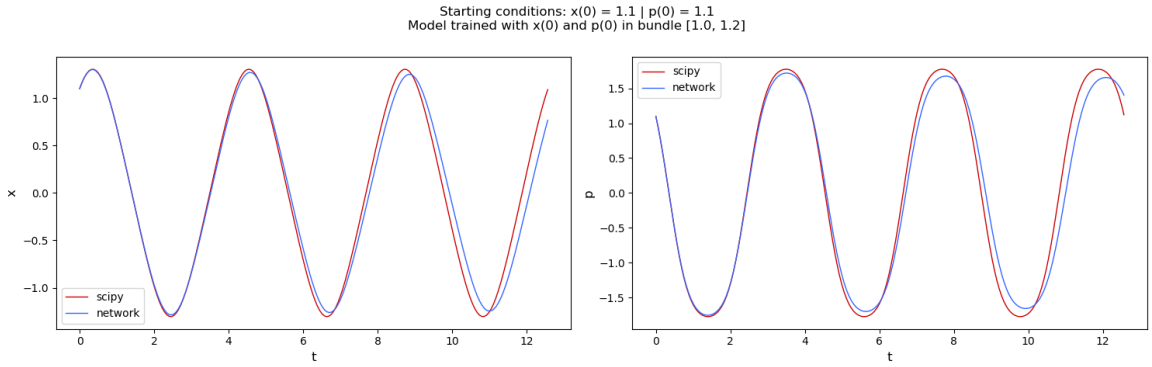


Figure 6.29 Network solution for the Nonlinear oscillator, with $\mathbf{x}(0) = 1.1, \mathbf{p}(0) = 1.1$, model trained on bundle of initial conditions C_4. *Left:* solution for $\mathbf{x}$. *Right:* solution for $\mathbf{p}$.

Figure 6.29 compares the solution found by network on an initial condition inside the bundle, with the one found by the SciPy solver. The network learns almost perfectly the solution, even if it was not trained directly on the selected initial condition. We obtained similar results for every initial condition sampled inside the bundle. Therefore, using initial conditions as input leads to a huge benefit: with just one training process we can learn multiple Cauchy problems at once.

We now show the results of our transfer learning approach in the bundle setting. Specifically, we analyzed the following settings, choosing two levels of perturbation:

- $C_4 : \mathbf{x}(0) \in [1.0, 1.2], \mathbf{p}(0) \in [1.0, 1.2] \rightarrow \ C_5 : \mathbf{x}(0) \in [1.5, 1.7], \mathbf{p}(0) \in [1.5, 1.7]$
- $C_4 : \mathbf{x}(0) \in [1.0, 1.2], \mathbf{p}(0) \in [1.0, 1.2] \rightarrow \ C_6 : \mathbf{x}(0) \in [2.2, 2.5], \mathbf{p}(0) \in [2.0, 2.2]$

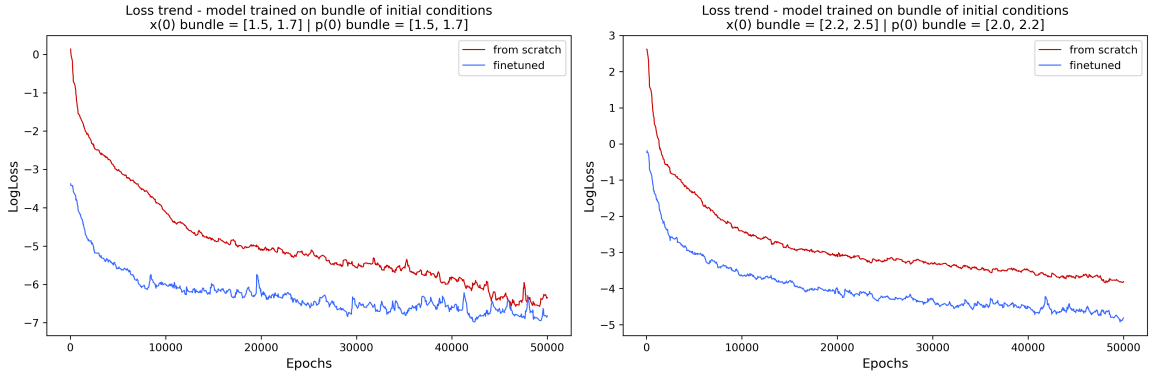


Figure 6.30 Scratch and finetuning comparison of the pre-trained Nonlinear oscillator on C_4, on new configurations C_5 and C_6. *Left*: networks trained on C_5. *Right*: networks trained on C_6.

Figure 6.30 shows the power of our approach applied to two very different cases. In the case of C_5, the new bundle to learn is the same for both x and p, and is not too far from the bundle condition C_4 on which the model was pre-trained. We can see how the finetuned model starts from a very lower loss and converges faster than the model trained from scratch on the new bundle. The latter, indeed, starts from a random initialization of the weights, reaching the loss of the finetuned model after $50K$ epochs.

In the second scenario, shifting from C_4 to C_6, we obtain again promising results. In this case, the new bundle to learn is different in x and p, it is further from C_4, and it is a little bit wider then C_5. Nonetheless, the finetuned model starts again from a very lower loss, and after $50K$ epochs it is still considerably below the loss of the model trained from scratch.

SIR model

In this epidemiologic system, we firstly applied a finetuning technique on the model trained on configuration C_1, shifting towards two very different configurations. In the application of transfer learning on the following cases - and on the consequent ones - we did not find any significant difference on the long term between a finetuned model or a model trained from scratch in terms of loss reached. Nonetheless, we found a great difference in terms of speed of convergence, hence we show the training trend for the first epochs. In particular, we analyzed the following two cases:

- $C_1 : \mathbf{S}(0) = 0.8, \beta = 0.8, \gamma = 0.2 \rightarrow \ \mathrm{C}_2 : \mathbf{S}(0) = 0.7, \beta = 0.8, \gamma = 0.2$
- $C_1 : \mathbf{S}(0) = 0.8, \beta = 0.8, \gamma = 0.2 \rightarrow \ \mathrm{C}_3 : \mathbf{S}(0) = 0.1, \beta = 0.8, \gamma = 0.2$

for brevity, we omitted $\mathbf{I}(0)$, that is always fixed as $1 - \mathbf{S}(0)$, and $\mathbf{R}(0) = 0$. Noticeably, configuration C_2 has an initial condition closer to C_1: it corresponds of a perturbation of just 12.5%. On the other hand, C_3 depicts an initial situation strongly different from C_1. Here below we plot the loss in the first 150 epochs of the finetuning process, as well as the loss curve if the network is randomly initialized, namely it is trained from scratch.

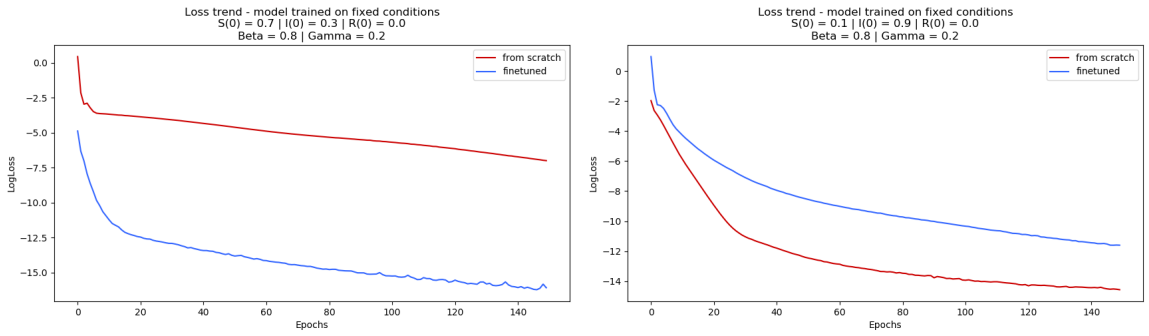


Figure 6.31 Scratch and finetuning comparison on configuration. *Left*: networks trained on C_2. *Right*: networks trained on C_3.

The two settings have very different behaviour: when learning configuration C_2, it is extremely beneficial starting from a model finetuned on C_1, being it a target domain very close to the source. On the other hand, if the shift is strong, as when learning C_3, the finetuning is even worse than retraining totally from scratch, at least in the first epochs.
These results, which have been replicated with other configurations, confirm that transfer learning will be extremely useful to solve fastly several Cauchy Problems, slightly perturbing the one where the baseline network is trained on.

Bundle of initial conditions

In this case, the problem is slightly more complicated, as expected: the network has to generalize over a set of possible initial conditions. Nonetheless, using the exactly same architecture and training hyperparameters of the fixed initial conditions scenario, the network is able to generalize well, converging to good results in approximately the same amount of epochs. In order to train, for each time step t, we randomly pick an initial condition $\mathbf{S}(0)$, drawn from a predefined bundle. The loss minimization, not shown for brevity, reached an error of $1 \cdot 10^{-5}$, which is higher than the previous case, but still a good result.
To show the effectiveness of the method, we here plot two network solutions, generated by the exact same model trained on the following setting:

$$C_4 : \mathbf{S}(0) \in [0.7, 0.9], \beta = 0.8, \gamma = 0.2$$

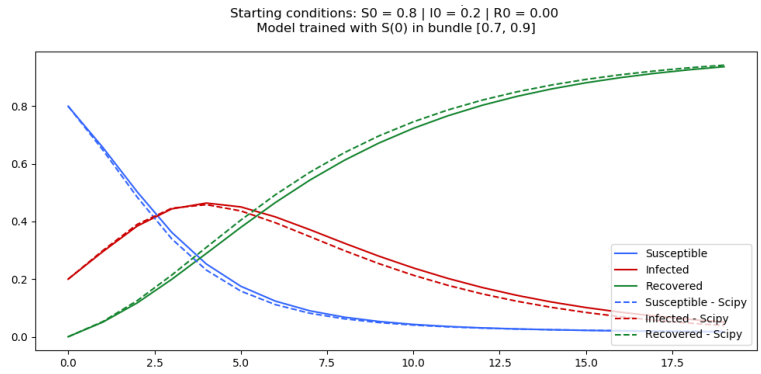


Figure 6.32 Network solution for the SIR model, with $\mathbf{S}(0) = 0.8, \mathbf{I}(0) = 0.2, \mathbf{R}(0) = 0.0$, $\beta = 0.8, \gamma = 0.2$, model trained on bundle of initial conditions.

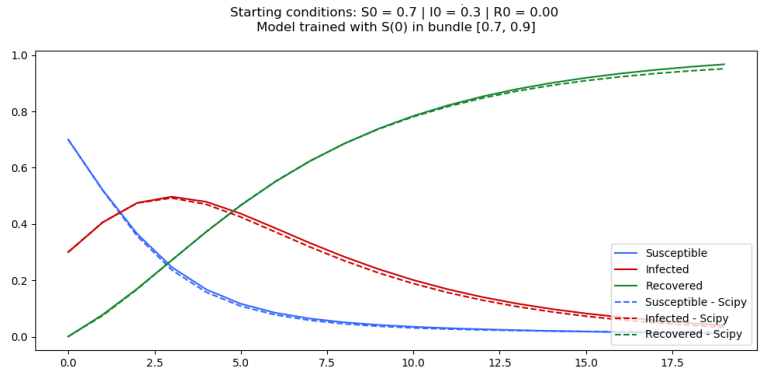


Figure 6.33 Network solution for the SIR model, with $\mathbf{S}(0) = 0.7, \mathbf{I}(0) = 0.3, \mathbf{R}(0) = 0.0$, $\beta = 0.8, \gamma = 0.2$, model trained on bundle of initial conditions.

More experiments have been done and they all led to results similar to the ones here displayed. The just listed experiments show effectiveness in terms of learning the solution of multiple Cauchy problems altogether. The discussion will continue by showing how transfer learning is even more effective in this case.
We used this last trained model as source domain and explored the following two target domains:

- $C_4 : \mathbf{S}(0) \in [0.7, 0.9]$, $\beta = 0.8, \gamma = 0.2 \rightarrow$ $C_5 : \mathbf{S}(0) \in [0.45, 0.65]$, $\beta = 0.8, \gamma = 0.2$
- $C_4 : \mathbf{S}(0) \in [0.7, 0.9]$, $\beta = 0.8, \gamma = 0.2 \rightarrow$ $C_6 : \mathbf{S}(0) = [0.05, 0.15]$, $\beta = 0.8, \gamma = 0.2$

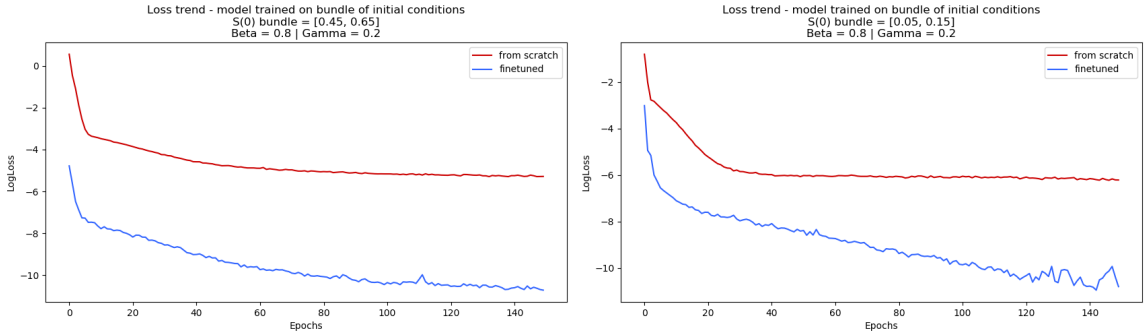


Figure 6.34 Scratch and finetuning comparison on configurations C_5, C_6. *Left*: networks trained on C_5. *Right*: networks trained on C_6.

The great benefit of learning on a bundle of initial conditions can be seen in figure 6.34. In this figure, we plotted again the trend of the loss for the first 150 epochs when finetuning on a pre-trained model, along with scratch training. The network trained on C_6 starts from an error greater than the one trained on C_5: this is reasonable, as its initial conditions are quite far from the one of the source domain. Nonetheless the training process can still benefit from the knowledge acquired from the network on the source task, resulting in the finetuned model reaching an error lower than the model trained from scratch. Instead, with no surprise, learning C_5 is significantly slower if starting from a random initialization of the weights, given the strong similarity between source and target tasks.

Bundle of initial conditions and parameters

Finally, due to the dependence of the system on β and γ, we vary those parameters as well within a range. In order to train, for each time step t of the training set, we sample a random tuple $(\mathbf{S}(0), \beta, \gamma)$, whose elements are drawn from a predefined bundle, and feed the network with it.

Given the complexity of this problem, we increased the network capacity, using 4 hidden layers, and we trained for $1 \cdot 10^4$ epochs, with $K = 2500$ points in the interval $t \in [0, 20]$.

Below we show two solutions of the system, obtained with the same network trained on the following configuration:

$$C_7 : \mathbf{S}(0) \in [0.7, 0.9], \beta \in [0.65, 0.85], \gamma \in [0.15, 0.3]$$

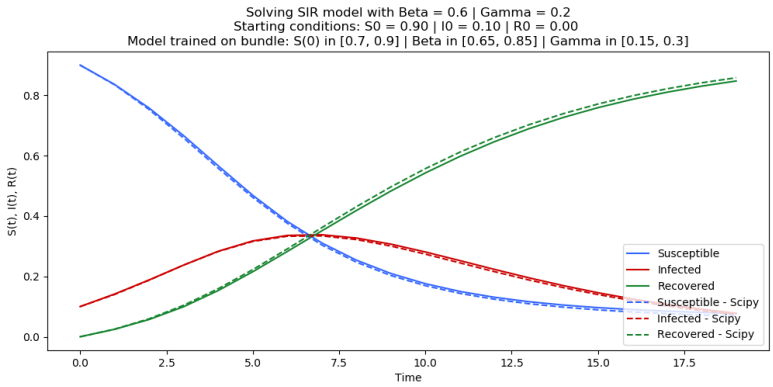


Figure 6.35 Network solution for the SIR model, with $\mathbf{S}(0) = 0.9, \mathbf{I}(0) = 0.1, \mathbf{R}(0) = 0.0$, $\beta = 0.6, \gamma = 0.2$, model trained on bundle of initial conditions and parameters.

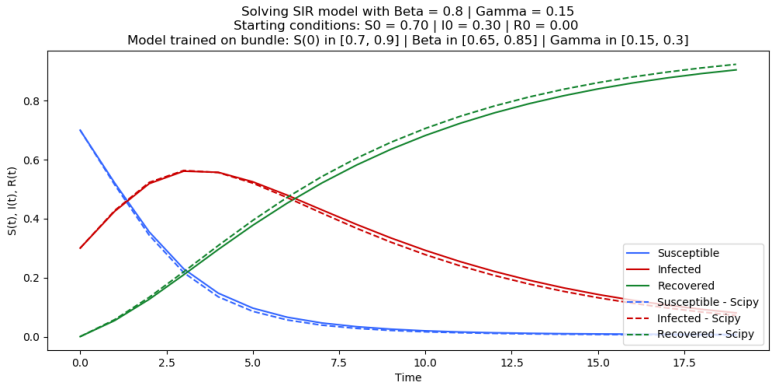


Figure 6.36 Network solution for the SIR model, with $\mathbf{S}(0) = 0.7, \mathbf{I}(0) = 0.3, \mathbf{R}(0) = 0.0$, $\beta = 0.8, \gamma = 0.15$, model trained on bundle of initial conditions and parameters.

As you can see, despite the two solutions are quite different from each other, the network approximates both of them. Indeed, also in this scenario we reached a loss of about $1 \cdot 10^{-5}$. Lastly, the application of transfer learning in this scenario concerns the two following problems:

- $C_7 : \mathbf{S}(0) \in [0.7, 0.9], \beta \in [0.65, 0.85], \gamma \in [0.15, 0.3] \rightarrow C_8 : \mathbf{S}(0) \in [0.45, 0.65], \beta \in [0.4, 0.6], \gamma \in [0.4, 0.6]$
- $C_7 : \mathbf{S}(0) \in [0.7, 0.9], \beta \in [0.65, 0.85], \gamma \in [0.15, 0.3] \rightarrow C_9 : \mathbf{S}(0) = [0.1, 0.3], \beta \in [0.2, 0.4], \gamma \in [0.7, 0.9]$

We show here the results of the training for the first 300 epochs, again comparing the finetuned case with the scratch case.

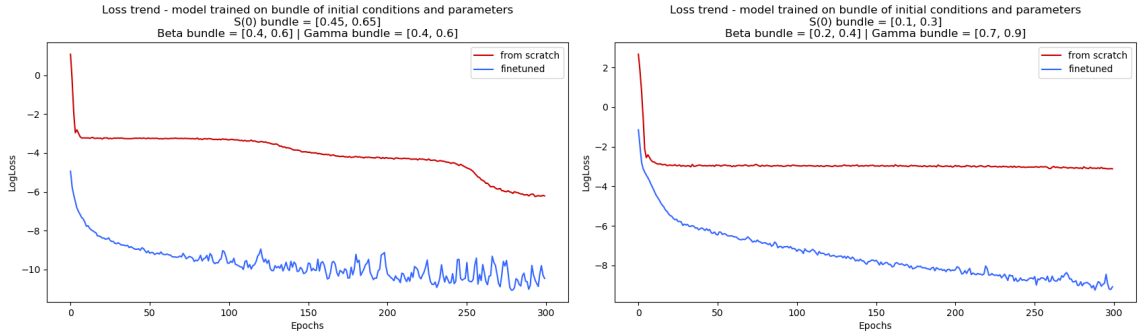


Figure 6.37 Scratch and finetuning comparison on configurations C_8, C_9. *Left*: networks trained on C_8. *Right*: networks trained on C_9.

Similarly to what we saw in figure 6.34, also in this case transfer learning is easier and fast. In fact, both C_8 and C_9 are easily learnt by the network, thanks to its highly generalization power.

6.2.4 Bundle Loss Analysis

Finally, to have a measure of the learning capacity of the architecture, we employed transfer learning to monitor how the LogLoss of the training varies according to the variation of the bundle size. To do so, we wanted to give the network full flexibility, hence we replaced the input $\mathbf{S}(0)$ with the tuple $(\mathbf{I}(0), \mathbf{R}(0))$ and forcing $\mathbf{S}(0) = 1 - \mathbf{I}(0) - \mathbf{R}(0)$.

Firstly, let us have a look at the loss distribution within and outside the bundle itself, using a network trained on:

- C_{10} : $\mathbf{I}(0) \in [0.2, 0.4], \mathbf{R}(0) \in [0.1, 0.3], \beta \in [0.4, 0.8], \gamma \in [0.3, 0.7]$

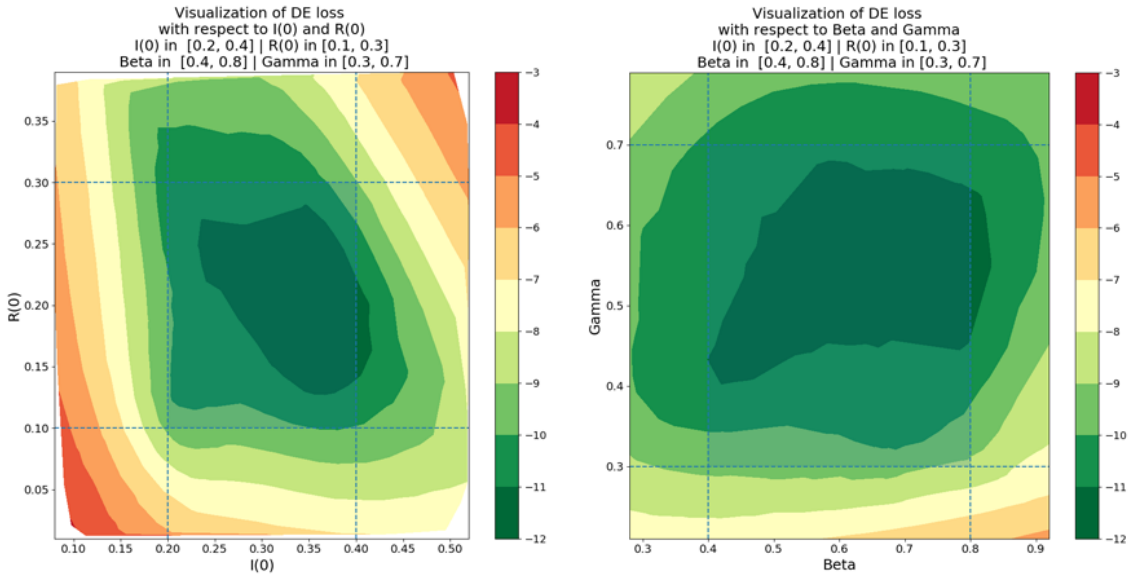


Figure 6.38 LogLoss distribution within and outside the bundle of C_{10}, measured by the colorbar.

In this plot we can see how the loss behaves as a function of initial conditions and parameters. For the plot on the left, we fixed (β, γ) at the center of their bundle, i.e. $\beta = 0.6$ and $\gamma = 0.5$, and solved the differential equations with initial conditions sampled randomly inside and outside the bundle. On the contrary, for the right-hand plot, we fixed $\mathbf{I}(0) = 0.3$ and $\mathbf{R}(0) = 0.2$ and sampled β and γ. We can see that the networks performance is excellent at the core of the bundle, highlighted from the blue square, and getting worse as you move outside, passing from a LogLoss of -12 to a LogLoss of -9 just outside from the known area. Finally, if you go way far from the bundle, as expected the LogLoss worsens significantly.

Knowing how the loss behaves in the surroundings of the bundle, we exploited transfer learning to understand how much is our explorability capability, namely how big our bundle can be. To do so, we trained a network on the following configuration, where the interval of each variable is sized 0.1.

- $C_{11} : \mathbf{I}(0) \in [0.1, 0.2], \mathbf{R}(0) \in [0.1, 0.2], \beta \in [1.0, 1.1], \gamma \in [1.0, 1.1]$

At this point, for each of the 4 variables, we increased incrementally the bundle size, keeping the others fixed, to discover which ones are more easily learnable from the network and which ones are more difficult. In fact, in the increment of the bundle size and in the learning of the new solutions, we started from the models previously trained, to save computational time and reach faster convergence.

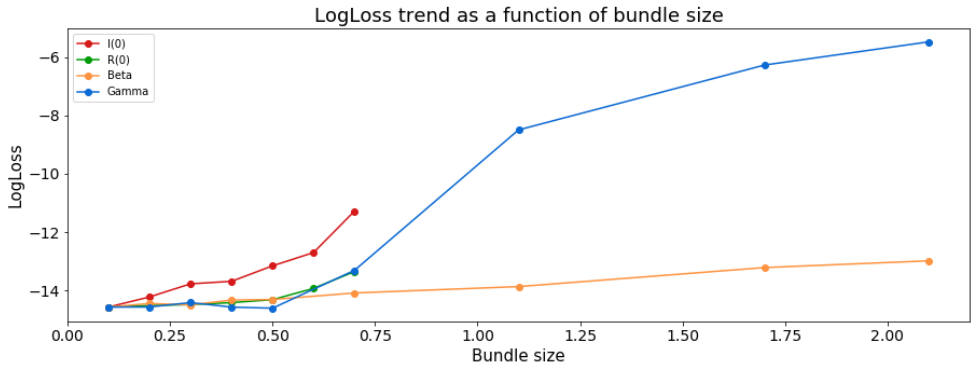


Figure 6.39 LogLoss as a function of the bundle size.

This plot shows that for 3 variables out of three the LogLoss remains reasonably low for significantly large bundles, which is an excellent result. Indeed, the only variable that causes trouble to the model is γ: if the bundle size exceeds 1.0 - and the other bundle sizes remain 0.1 - the network is not able to solve the task.

6.2.5 Possible Applications

Given the possibility of learning over a bundle of initial conditions and parameters, the question we posed ourselves is the following: if we can solve the DEs system for a set of parameters, can we know which exact $\mathbf{I}(0)$, $\mathbf{R}(0)$, β and γ fits better a real trend? To answer this question, we developed the following methodology: let us call B a model trained on a bundle of initial conditions $[\xi_{min}, \xi_{max}]$ and parameters $[\theta_{min}, \theta_{max}]$, T a subset of a trajectory coming from the SIR system, that we know it is generated by solving the SIR system for an unknown $\bar{\mathbf{z}}(0) \in [\xi_{min}, \xi_{max}]$ and $\bar{\theta} \in [\theta_{min}, \theta_{max}]$. Our goal is to discover $(\bar{\mathbf{z}}(0), \bar{\theta})$, by minimizing the difference between the solution of the model and T. To be more formal, we want to find $\hat{\theta}$, such that:

$$\hat{\mathbf{z}}(0), \hat{\theta} = \underset{\mathbf{z}(0),\theta}{\arg\min} \underbrace{\frac{1}{K} \sum_{n=1}^{K} \left(\hat{T}^{(n)} - T^{(n)} \right)^2}_{MSE} \tag{6.2}$$

where K is the number of total points of T, presumably small, $T^{(n)}$ is the value of the trajectory at $t = n$. This is a supervised problem that is solvable by a minimization of a MSE loss function, with respect to $(\mathbf{z}(0), \theta)$. On this purpose, we developed the procedure illustrated below:

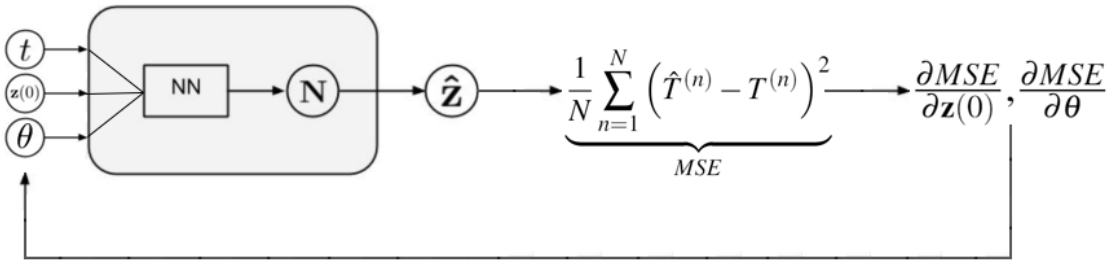


Figure 6.40 Optimization procedure to find $\bar{\mathbf{z}}(0), \bar{\theta}$.

To estimate $(\hat{\mathbf{z}}(0), \hat{\theta})$, we feed the network 4.5 with a random tuple $(\mathbf{z}(0), \theta)$, coming from the bundle where the net has been trained, along with the time-steps t whose trajectories is contained in T, to compute the solution of the DE. Then, we compute the *MSE* as in equation 6.2 and run an optimization procedure with respect to the inputs of the network - $\mathbf{z}(0), \theta$ -, computing the gradient of the *MSE* and updating the inputs accordingly. In our specific case, we adapted the methodology to the SIR model, hence:

$$\mathbf{z}(0) = \mathbf{I}(0), \mathbf{R}(\mathbf{0})$$
$$\theta = \beta, \gamma$$

and define the following metric, to measure the difference between the estimated parameters and the real parameters:

$$Score = \left(\hat{\mathbf{I}}(0) - \bar{\mathbf{I}}(0)\right)^2 + \left(\hat{\mathbf{R}}(0) - \bar{\mathbf{R}}(0)\right)^2 + \left(\hat{\beta} - \bar{\beta}\right)^2 + (\hat{\gamma} - \bar{\gamma})^2$$

We run the optimization algorithm with $K = 4$ equally spaced points belonging to T, for 100 epochs, using Adam optimizer and a learning rate of $1 \cdot 10^{-3}$. This particular problem is very susceptible to local minima, given the limited number of points we use to optimize, but oftentimes it is just necessary to re-run the (very fast) minimization algorithm to find the best result, in term of minimum *MSE*.

To show the effectiveness of the method, we sampled 100 different tuples $(\bar{\mathbf{I}}(0), \bar{\mathbf{R}}(0), \bar{\beta}, \bar{\gamma})$ as unknowns to find.

Similarly to what we have done in the bundle exploration, we wanted to discover how the *score* estimation is affected if the trajectories to fit are generated by models trained on variables outside the bundle where the analyzed network has been trained.

Here we show some results of the optimization, by plotting the unknowns in the $(\beta, \gamma) - space$ and in the $(\mathbf{I}(0), \mathbf{R}(0)) - space$: similarly to figure 6.38, we either fixed the initial conditions or the parameters at the center of the bundle. The color of the areas represents the *score*

obtained after the minimization, i.e. how good the initial conditions and the parameters are estimated.

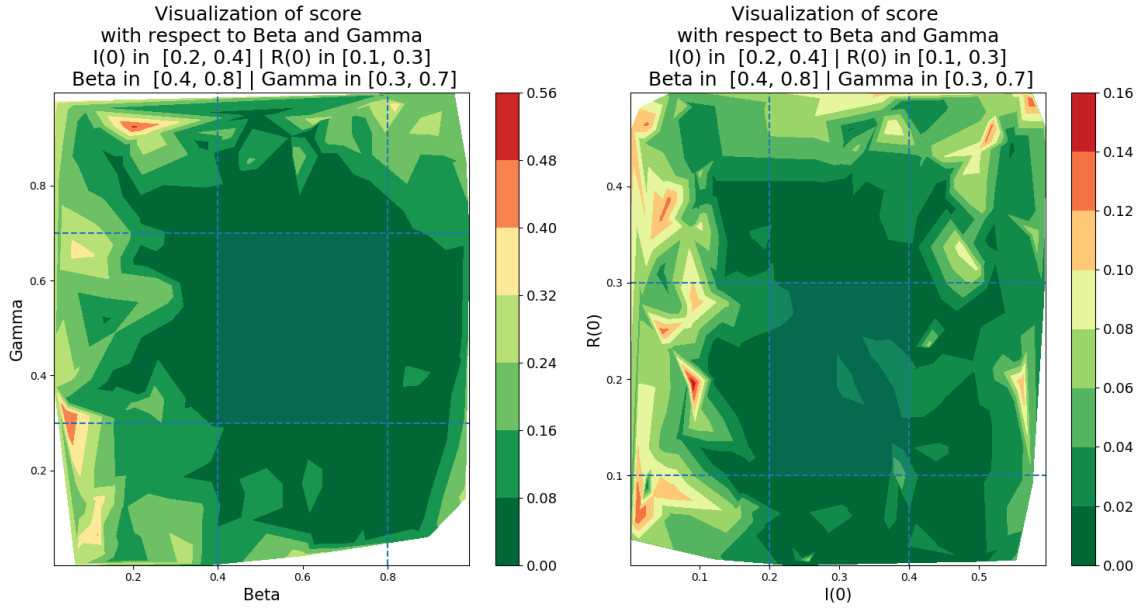


Figure 6.41 Score results for three different networks.

In figure 6.41 you can see the results of 100 optimization runs for a network trained on C_{10}, where the blue-area delimits the bundle of parameters where the model has been trained. As expected, in those areas the algorithm is quite accurate in the estimation, likewise in a significant portion of space outside, going toward worse score values as it ends up far from the bundle. The method just outlined is meant to be a solution to the problem of finding the parameters of a model, given very few data points. In this scenario, we have the general form of an equation, namely the SIR system, but we are able to find the specific parameters with a small dataset. This is doable thanks to the power of neural networks, which are able to learn analytical solutions of the system, namely they can find a function that approximates the solution, and thus provide analytical derivatives.

Chapter 7

Conclusion

This thesis explored the field on transfer learning in two very different use cases: image recognition and resolution of differential equations. These two scenarios, the first supervised and the second unsupervised, led to different techniques and different outcomes. In the previous chapter we showed the results of the experiments carried on, with some comments. Now we want to recap the entire process and suggest some bits of discussion for future works.

7.1 Summary of the Results

In both scenarios we applied transfer learning to adapt the Deep Learning models to solve new and different tasks, at various levels. The diverse method yielded different results in the two cases, in terms of performance and generalizability of the techniques proposed.
In the image recognition task, we explored three criteria (error-driven, entropy-driven and differential) to select a part of the target dataset to use for the finetuning, to understand whether one of them can yield better (or comparable) performance with respect to use the whole data available. Additionally to the shown ones, we run many more experiments, with different networks architecture, different datasets and hyperparameters - especially regularization techniques to prevent the overfitting of the network on a subset of the data. We decided to include only the listed results as they are the most illustrative and explanatory, so that the behaviour of the network is as clear as possible.
Out of all the experiments, it seems problematic finding a portion of the data that reflects accurately the distribution of the entire information at disposal: if this does not happen (as in some of the cases listed), a network trained on a biased dataset will never perform better than a network trained on a random selection. Furthermore, in the implementation, we often encountered problems of complexity, both in space and time, which is an important factor to

take into account: if the scope of a given technique is reducing the size of dataset to save computational resources for the training, the data selection procedure itself should be light and fast as well, otherwise the benefit is minimal. Based on the results, we realized that it is hard to deprive Deep Learning models of data, as it is in their nature to be *data hungry*. Nonetheless, we do believe that techniques to portion the data in a smart way to boost the finetuning of the network exist.
On the other hand, the transfer learning methodologies applied in the field of differential equations resulted in better outcomes, in terms of speed of convergence of the training process and generalization of the networks. The power of the methodologies, indeed, relies in the great amount of computational power you can save by exploiting them: firstly, we proved that transfer learning allows a fast exploration of the various Cauchy problems, secondly (and thirdly) we changed the architecture as to provide the network with a better generalization capability, and turned out that transfer learning is even more beneficial in that case.
Furthermore, the new architectures open to a new method for estimation of equation parameters, illustrated in section 6.2.5, thanks to the function approximation ability of Neural Networks.

7.2 Future Works

Future developments of this work are many, in both the scenarios we explored.
Starting from the images problem, there is still room for research in order to find a technique which is generalizable and computationally light.
To keep on investigating this field, we would suggest to study the field of uncertainty sampling in Deep Learning, as it was the one who gave us the better results (even though not in all the cases), also in term of energy usage.
The DEs task opens instead a multitude of applications. Firstly, we applied our techniques on two systems of ODE, hence it would be interesting to explore other dynamical systems and perhaps modify the architecture to solve also systems of Partial Differential Equations.
Concerning the perturbation of initial conditions, and likewise of parameters and relative bundle, we did not run an analysis of the amount of finetuning required in relation to the intensity of the distortion. In other words, we did not investigate thoroughly how each different distortion impacts on the network and how much it should be adjusted to get the correct solution again.
Lastly, the parameter estimation described in section 6.2.5 can be potentially used to describe the behaviour of real phenomena which can be approximated with systems of DEs. Indeed, an epidemic phenomenon can be easily characterized by mean of this methodology and few

real-data points, to use as supervised information in the optimization procedure. Hence, this technique can be generalized for several dynamical systems and help to describe them with small effort and limited knowledge.

Bibliography

[1] Backpropagation from the beginning. https://medium.com/@erikhallstrm/backpropagation-from-the-beginning-77356edf427d, accessed 10/02/2020.

[2] Google colaboratory. https://colab.research.google.com, accessed 11/04/2019.

[3] History of python. https://en.wikipedia.org/wiki/History_of_Python, accessed 13/03/2020.

[4] Numpy website. https://numpy.org/, accessed 13/03/2020.

[5] Python documentation. https://www.python.org/doc/, accessed 13/03/2020.

[6] Pytorch autograd. https://pytorch.org/docs/stable/autograd.html.

[7] Pytorch website. https://pytorch.org/, accessed 13/03/2020.

[8] Pytorch website. https://heartbeat.fritz.ai/10-reasons-why-pytorch-is-the-deep-learning-framework-of- accessed 13/03/2020.

[9] Schema of the max pooling operation. http://cs231n.github.io/convolutional-networks/, accessed 17/02/2020.

[10] Scipy odeint documentation. https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.odeint.html, accessed 10/023/2020.

[11] Tensorboard website. https://www.tensorflow.org/tensorboard, accessed 13/03/2020.

[12] Transfer learning example. https://www.slideshare.net/xavigiro/transfer-learning-d2l4-insightdcu-machine-learning-workshop-2017, accessed 09/02/2020.

[13] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[14] Muhammad Afridi, Arun Ross, and Erik Shapiro. On automated source selection for transfer learning in convolutional neural networks. *Pattern Recognition*, 73, 07 2017.

[15] N.T.J. Bailey. *The Mathematical Theory of Infectious Diseases and Its Applications*. Mathematics in Medicine Series. Griffin, 1975.

[16] Yohai Bar-Sinai, Stephan Hoyer, Jason Hickey, and Michael P Brenner. Learning data-driven discretizations for partial differential equations. *Proceedings of the National Academy of Sciences*, 116(31):15344–15349, 2019.

[17] Tejas S. Borkar and Lina J. Karam. Deepcorrect: Correcting DNN models against image distortions. *CoRR*, abs/1705.02406, 2017.

[18] John S. Bridle. Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 211–217. Morgan-Kaufmann, 1990.

[19] Sandro Salsa Carlo Domenico Pagani. *Analisi Matematica 2*. Zanichelli, 2014.

[20] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and André van Schaik. EMNIST: an extension of MNIST to handwritten letters. *CoRR*, abs/1702.05373, 2017.

[21] Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[22] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, June 2009.

[23] Samuel F. Dodge and Lina J. Karam. Understanding how image quality affects deep neural networks. *CoRR*, abs/1604.04004, 2016.

[24] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning, 2016.

[25] H. Goldstein, C.P. Poole, and J.L. Safko. *Classical Mechanics*. Addison Wesley, 2002.

[26] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[27] Lin Gui, Ruifeng Xu, Qin Lu, Jiachen Du, and Yu Zhou. Negative transfer detection in transductive transfer learning. *International Journal of Machine Learning and Cybernetics*, 9, 02 2017.

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[29] Robert Hooke. *Lectures de potentia restitutiva, or, Of spring [microform] : explaining the power of springing bodies : to which are added some collections / by Robert Hooke*. Printed for J. Martyn London, 1678.

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

[31] Jonathan J Hull. A database for handwritten text recognition research. *CoRR*, abs/1702.05373, 1994.

[32] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, Richard C. Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *CoRR*, abs/1704.04760, 2017.

[33] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[34] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E. Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B. Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and et al. Jupyter notebooks - a publishing format for reproducible computational workflows. In *ELPUB*, 2016.

[35] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research).

[36] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-100 (canadian institute for advanced research).

[37] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[38] I.E. Lagaris, A. Likas, and D.I. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE Transactions on Neural Networks*, 9(5):987–1000, 1998.

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

[40] Yann Lecun. A theoretical framework for back-propagation. In D. Touretzky, G. Hinton, and T. Sejnowski, editors, *Proceedings of the 1988 Connectionist Models Summer School, CMU, Pittsburg, PA*, pages 21–28. Morgan Kaufmann, 1988.

[41] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

[42] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[43] Xiaolong Liu, Zhidong Deng, and Yuhan Yang. Recent progress in semantic image segmentation. *CoRR*, abs/1809.10198, 2018.

[44] Martin Magill, Faisal Qureshi, and Hendrick W. de Haan. Neural networks trained to solve differential equations learn general representations, 2018.

[45] Marios Mattheakis, David Sondak, Akshunna S. Dogra, and Pavlos Protopapas. Hamiltonian neural networks for solving differential equations, 2020.

[46] Tom Mitchell. Machine learning, international edition. In *McGraw-Hill Series in Computer Science*, 1997.

[47] Jose G. Moreno-Torres, Troy Raeder, RocíO Alaiz-RodríGuez, Nitesh V. Chawla, and Francisco Herrera. A unifying view on dataset shift in classification. 45(1), 2012.

[48] I. Newton and D.T. Whiteside. *The Mathematical Papers of Isaac Newton: Volume 3*. The Mathematical Papers of Sir Isaac Newton. Cambridge University Press, 2008.

[49] Jiquan Ngiam, Daiyi Peng, Vijay Vasudevan, Simon Kornblith, Quoc V. Le, and Ruoming Pang. Domain adaptive transfer learning with specialist models. *CoRR*, abs/1811.07056, 2018.

[50] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Trans. on Knowl. and Data Eng.*, 22(10):1345–1359, October 2010.

[51] Luis Perez and Jason Wang. The effectiveness of data augmentation in image classification using deep learning. *CoRR*, abs/1712.04621, 2017.

[52] Lawrence Perko. *Differential Equations and Dynamical Systems*. Springer-Verlag, Berlin, Heidelberg, 1991.

[53] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378, 11 2018.

[54] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386, 1958.

[55] Michael T. Rosenstein, Zvika Marx, Leslie Pack Kaelbling, and Thomas G. Dietterich. To transfer or not to transfer. In *In NIPS'05 Workshop, Inductive Transfer: 10 Years Later*, 2005.

[56] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536, 1986.

[57] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[58] Burr Settles. Active learning literature survey. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2009.

[59] Claude E. Shannon. A mathematical theory of communication. *Bell Syst. Tech. J.*, 27(3):379–423, 1948.

[60] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[61] Justin Sirignano and Konstantinos Spiliopoulos. Dgm: A deep learning algorithm for solving partial differential equations. *Journal of Computational Physics*, 375:1339–1364, Dec 2018.

[62] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net, 2014.

[63] Peter Norvig Stuart Russell. Artificial intelligence: A modern approach (3rd edition). In *Prentice Hall*, 2002.

[64] Farhana Sultana, Abu Sufian, and Paramartha Dutta. Advancements in image classification using convolutional neural network. *CoRR*, abs/1905.03288, 2019.

[65] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[66] Christian Szegedy, Alexander Toshev, and Dumitru Erhan. Deep neural networks for object detection. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 2553–2561. Curran Associates, Inc., 2013.

[67] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 2020.

[68] Pantelis R. Vlachas, Wonmin Byeon, Zhong Y. Wan, Themistoklis P. Sapsis, and Petros Koumoutsakos. Data-driven forecasting of high-dimensional chaotic systems with long short-term memory networks. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 474(2213):20170844, May 2018.

[69] Tianyang Wang, Jun Huan, and Bo Li. Data dropout: Optimizing training data for convolutional neural networks. *CoRR*, abs/1809.00193, 2018.

[70] Tianyang Wang, Jun Huan, and Michelle Zhu. Instance-based deep transfer learning. *CoRR*, abs/1809.02776, 2018.

[71] Zirui Wang, Zihang Dai, Barnabás Póczos, and Jaime Carbonell. Characterizing and avoiding negative transfer. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 11293–11302, 2019.

[72] Jianxiong Xiao, James Hays, Krista A. Ehinger, Aude Oliva, and Antonio Torralba. Sun database: Large-scale scene recognition from abbey to zoo. In *CVPR*, pages 3485–3492. IEEE Computer Society, 2010.

[73] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3320–3328. Curran Associates, Inc., 2014.

[74] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. *CoRR*, abs/1311.2901, 2013.

[75] Yiren Zhou, Sibo Song, and Ngai-Man Cheung. On classification of distorted images with deep convolutional neural networks. *CoRR*, abs/1701.01924, 2017.

[76] Linchao Zhu, Sercan O. Arik, Yi Yang, and Tomas Pfister. Learning to transfer learn, 2019.

Appendix A

Additional Experimental Settings

This appendix is devoted to present all the experimental settings adopted in the image recognition task, which have been tried during the development of the thesis and which have not been included in section 6.1.

A.1 Datasets

In this section we will list additional datasets that have been used to test our approaches:

- **CIFAR-100:** the CIFAR-100 dataset [36] is just like the CIFAR-10, except it has 100 classes containing 600 images each. The 100 classes in the CIFAR-100 are grouped into 20 superclasses. Each image comes with a "fine" label (the class to which it belongs) and a "coarse" label (the superclass to which it belongs).
- **EMNIST** the EMNIST dataset [20] is a set of handwritten character digits derived from the NIST Special Database 19. The dataset constitute a more challenging classification tasks with respect the MNIST. It involves digits and letters (both lowercase and uppercase), and it shares the same image structure and parameters as the original MNIST task, allowing for direct compatibility with all existing classifiers and systems.
- **Synthetized Data**: in addition to all the datasets presented, we decided to synthetize our own datasets using SciPy [67], a Python library for scientific computing. In this way it was possible to have total control on the features of the data, which led to a deeper investigation of the results of our proposed techniques. Furthermore, using a synthetized dataset sped up the runtime of our experiments, and gave the chance to visualize data points without leveraging on techniques for data dimensionality reduction.

A.2 Distortions

Together with embedding shift, the following list of distortions have been adopted to to generate new datasets with different levels of severity:

- **Additive White Gaussian Noise (AWGN)**: it is commonly used to model additive noise encountered during image acquisition and transmission. It is *additive*, because it is added to any noise that might be intrinsic of the considered image. *White*, instead, refers to the fact that it has uniform power across the whole frequency band. Finally, *Gaussian*, describes that the probability distribution of the noise samples is Gaussian with a zero mean in time domain.
- **Gaussian blur**: it is often encountered during image acquisition and compression. It represents a distortion that eliminates high frequency discriminative object features like edges and contours. The visual effect of this blurring technique is a smooth blur resembling that of viewing the image through a translucent screen.
- **Color shift**: it is a simple, additive noise. It consists in an integer value added to each pixel value in just one of the RGB (Red, Blue, Green) channels. Given a shift s and a value of a pixel in a particular channel p, the new value will be $p \rightarrow (p+s)(mod256)$.

A.3 Architectures

For CIFAR 100, the convolutional network presented in section 6.1 proved to be insufficient for the classification task, and therefore we used a fully-convolutional network with a higher number of layers. It consists of only convolutional layers with a final 100-way softmax layer and it is heavily based on the All-Conv Net proposed by *Springenberg et al.* [62], with the addition of batch normalization units after each convolutional layer.

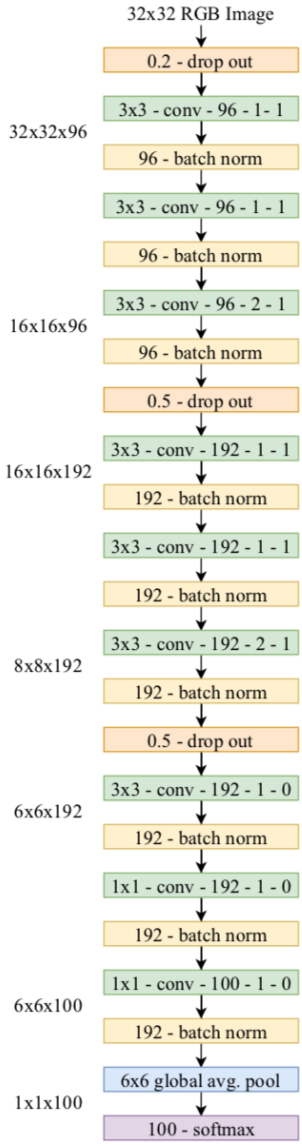


Figure A.1 Network architecture used in the image recognition context for CIFAR 100 dataset. Convolutional layers are parameterized by *kxk*-conv-*d*-*s*-*p*, where *kxk* is the spatial extent of the filter, d is the number of output filters in a layer, s represents the filter stride and p indicates the zero-padding. Max-pooling layers are parameterized as *kxk*-maxpool-*s*-*p*, where *s* is the spatial stride and *p* indicates the implicit zero padding. Batch normalization layers are parameterized by *d*-bn, where *d* is the number of features in the layer. Finally, fully connected layers are parameterized by *d*-fc, where *d* represents the dimensionality of the output space.