UNIVERSITY OF CATANIA
DEPARTMENT OF ECONOMICS AND BUSINESS
MASTER DEGREE IN DATA SCIENCE FOR MANAGEMENT

Bruno Casella

Federated Transfer Learning using Network Composition

THESIS

Supervisor:
Prof. Sebastiano Battiato
Co-supervisor:
Prof. Valerio Giuffrida

September 2021

In memory of my grandmother.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Machine learning (ML) is the study of powerful techniques that can learn from experience. As a machine learning algorithm accumulates more experience, typically in the form of observational data or interactions with an environment, its performance improves.
Today, **Artificial Intelligence** (AI) is a thriving field with many practical applications and active research topics. We look to intelligent software to automate routine labor, understand speech or images, make diagnoses in medicine and support basic scientific research. The true challenge to artificial intelligence proved to be solving the tasks that are easy for people to perform but hard for people to describe formally—problems that we solve intuitively, that feel automatic, like recognizing spoken words or faces in images. The solution to this more intuitive problems is to allow computers to learn from experience and understand the world in terms of a hierarchy of concepts, with each concept defined in terms of its relation to simpler concepts. By gathering knowledge from experience, this approach avoids the need for human operators to formally specify all of the knowledge that the computer needs. The hierarchy of concepts allows the computer to learn complicated concepts by building them out of simpler ones. If we draw a graph showing how these concepts are built on top of each other, the graph is deep, with many layers. For this reason, we call this approach to AI **Deep Learning** (DL).
Ironically, abstract and formal tasks that are among the most difficult mental undertakings for a human being are among the easiest for a computer. Computers have long been able to defeat even the

best human chess player, but are only recently matching some of the abilities of average human beings to recognize objects or speech. A person's everyday life requires an immense amount of knowledge about the world. Much of this knowledge is subjective and intuitive, and therefore difficult to articulate in a formal way. Computers need to capture this same knowledge in order to behave in an intelligent way. One of the key challenges in artificial intelligence is how to get this informal knowledge into a computer.
The difficulties faced by systems relying on hard-coded knowledge suggest that AI systems need the ability to acquire their own knowledge, by extracting patterns from raw data. This capability is known as machine learning. The introduction of machine learning allowed computers to tackle problems involving knowledge of the real world and make decisions that appear subjective. A simple machine learning algorithm called *logistic regression* can determine whether to recommend cesarean delivery. A simple machine learning algorithm called *naive Bayes* can separate legitimate e-mail from spam e-mail. DL has demonstrated to be far superior to traditional ML methods in a variety of tasks, through learning features directly learn from data. However, high classification accuracy is not enough especially in safety-critical contexts like the medical or the defense ones.
Different issues often come into play, posing several challenges: few annotated data, class-imbalance and privacy (e.g., due to data privacy regulations, it is often unfeasible to collect and share data in a centralised data lake).
A technique called **Federated learning** (FL) may address these challenges by training models where the data is spread across nodes, sharing only weights with the central aggregation node. This approach is only applicable in the context of data with either common features or common samples under a federation. In reality, however, the set of such common entities may be small, making a federation less attractive and leaving the majority of the non-overlapping data under-utilized.
An other frequent problem is reusing or transferring information from previously learned tasks for the learning of new tasks. This problem is tackled with some techniques of **Transfer Learning** (TL). For

example, if you have any experience programming with Java, C or any other programming languages, you're already familiar with concepts like loops, recursion, objects and etc. If you then try to pick up a new programming language like python, you don't need to learn these concepts again, you just need to learn the corresponding syntax. Or to take another example, if you have played table tennis a lot it will help you learn tennis faster as the strategies in these games are similar.

a) Transferring Learned knowledge from Java to Python.

b) Transferring Learned knowledge Table Tennis to Tennis.

Figure 1.1: Real-life Transfer Learning example

In this thesis, is proposed a new technique of **Federated Transfer Learning** (FTL), a special case of FL, in which two datasets differ in the feature space. This can be applied to different datasets but similar in their nature, e.g. data collected from companies in the same sector, but with differences in the nature of business. These enterprises, also physically far in the world, share only a small overlap in feature space, and their datasets differ both in samples and in feature space.

Our proposed method aims to address the limitations of existing federated learning approaches, and leveraging transfer learning to provide solutions for the entire sample and feature space under a federation. The general idea of our proposed method is to combine the weights of two different neural networks trained on two different machines and datasets, but similar in their nature, and check how the performances change after the aggregation.

The experiments and the drafting of the thesis has been carried out together with my colleague of Management Engineering Alessio Barbaro Chisari, under the supervision of Prof. Sebastiano Battiato and Prof. Valerio Giuffrida, a Lecturer in Data Science at Edinburgh Napier University. In particular, it was expected a period of stay in Scotland at the Edinburgh Napier University, however it was not possible due to the CoViD-19 limitations.

Organization of the dissertation

The dissertation is organized as follows.
In Chapter 2 well be given the main notions about artificial intelligence, machine learning and deep learning in order to understand the results presented in the dissertation.
In Chapter 3 will be illustrated some existing techniques of Federated Learning, Transfer Learning and Federated Transfer Learning.
In Chapter 4 will be presented the proposal of a new technique of Federated Transfer Learning experimented in this dissertation.
In Chapter 5 will be described all the experiments done in order to achieve good results.
Finally, in Chapter 6 will be drawn the conclusions and will be given some hints towards future developments.

Chapter 2

Background

In this chapter will be provided background knowledge to set the stage for the subsequent chapters. The structure of this chapter taken from the Ph.D. thesis of Ruder S. [32] and from the book *Dive Into Deep Learning* [33]. The reader will be introduced to Artificial Intelligence and Machine Learning, that build mathematical models from data, and their most elementary methods, reviewing also the cornerstones of the techniques introduced, based on fundamentals of probability and information theory. In particular we will go into details of the type of Deep Learning models that will be mostly used in this thesis, that are Neural Networks. Finally, an overview of common tasks and famous architectures in Machine Learning will be provided.

2.1 Machine Learning

In this section, we introduce the reader to Machine Learning, which builds mathematical models from data. Lot of the concepts that will be introduced in this section, will be reused throughout the thesis, because they are the building blocks of many models, such as advanced neural network-based methods. In particular, the problem of generalization will be frequently addressed, because we want to create models that generalize to other domains and tasks.

First, we are going to introduce some core components that will follow us around, no matter what kind of machine learning problem we take on:

- The *data* that we can learn from.
- A *model* of how to transform the data.
- An *objective function* that quantifies how well (or badly) the model is doing.
- An *algorithm* to adjust the model's parameters to optimize the objective function.
- The goal of Machine Learning, the *generalization*.
- The technique of *regularization*.

2.1.1 Data

Obviously, you can not do Data Science without data. Generally, when we talk of data, we are concerned with a collection of examples. In order to work with data usefully, we typically need to come up with a suitable numerical representation. Each example (or data point, data instance, sample) typically consists of a set of attributes called *features* (or covariates), from which the model must make its predictions. In supervised learning problems, the thing to predict is a special attribute that is designated as the *label* (or target).
Formally, each input if typically represented as a vector $x \in R^d$ of d *features*, where each feature contains the value for a particular attribute of the data and each example is assumed to be drawn independently from the data generating distribution $\hat{p}$. An entire dataset can be seen as a matrix $X \in R^{n \times d}$ containing n examples, one example in each row.
If our work domain is related to image data, we can have photographs and each individual photo might constitute an example, each represented by an ordered list of numerical values corresponding to the brightness of each pixel. A 200×200 color photograph would consist of 200×200×3=120000 numerical values, corresponding to the brightness of the red, green, and blue (RGB) channels for each spatial location.
In another traditional task, we might try to predict whether or not a patient will survive, given a standard set of features such as age,

vital signs, and diagnoses.
When every example is characterized by the same number of numerical values, we say that the data consist of *fixed-length* vectors and we describe the constant length of the vectors as the *dimensionality* of the data. Fixed-length can be a convenient property, because it means one less thing to worry about during the creation of the model.
It is not easy to represent data as fixed-length vectors. For example we can not expect images from Internet to all show up with the same resolution or shape; we might consider cropping them all to a standard size, but we risk losing information in the cropped out portions. Moreover, the domain of text data presents other problems. For example reviews in e-commerce sites such as Amazon or TripAdvisor can be of different lengths; some are short: "very good!", but others can go on for pages. Deep Learning overcomes traditional methods of Machine Learning, because it can handle varying-length data.
Generally, the more data we have, the easier our job becomes. When we have more data, we can train more powerful models and rely less heavily on pre-conceived assumptions.
Finally, we need the *right* data; it is not enough to have a high quantity of data and process them in a clever way, because if the data contain mistakes, or if the chosen features are not useful for the prediction of target of interest, learning will fail.

2.1.2 Models

Most machine learning involves transforming the data in some sense. With the term *model*, we indicate the computation process of ingesting data of one type, and returning predictions of a possibly different type. In particular, we are interested in statistical models that can be estimated from data.
A point estimator $\hat{\theta}$ is any function of the data that seeks to model the true underlying parameter θ^* of the data: $\hat{\theta} = g(X)$
As the data is assumed to be generated from a random process and $\hat{\theta}$ is a function of the data, $\hat{\theta}$ is itself a random variable.
The simplest example of a *point estimator* that maps from inputs to

outputs is *linear regression*, which aims to solve a regression problem. Linear regression models a conditional probability distribution $p(y|\mathbf{x})$: it takes as input a vector $\mathbf{x} \in R^d$ and aims to predict the value of a scalar $y \in R$ using a vector $\theta \in R^d$ of *weights* or *parameters* and an *intercept* or *bias* term $b \in R$:

$$\hat{y}(x;\theta) = \theta^T x + b \tag{2.1}$$

where $\hat{y}$ is the predicted value of y. The mapping from features to prediction is an *affine function*, i.e. a linear function plus a constant.

Linear regression can be applied also to classification; in this case we talk of *logistic regression*. For example, in case of binary classification we have two classes, class 0 and class 1. The output of linear regression can be squashed into a probability, in the interval (0,1), using the *sigmoid* σ, which is the following function:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

The probability produced by logistic regression is then calculated as follows:

$$\hat{p}(y = 1|x;\theta) = \hat{y} = \sigma(\theta^T x) \tag{2.3}$$

Clearly, specifying the probability of one of these classes determines the probability of the other class, because the output random variable follows a Bernoulli distribution.
In case of multi-class classification, we learn a separate set of weights $\theta_i \in \theta$ for the label y_i of the i-th class. Then, we apply the *softmax* function to squash the values to obtain a categorical distribution:

$$\hat{p}(y_i|x;\theta) = \frac{e^{\theta_i^T x}}{\sum_{j=1}^{C} e^{\theta_j^T x}} \tag{2.4}$$

where the denominator is the so-called *partition function* that normalizes the distribution by summing over the scores for all C classes.

2.1.3 Objective functions

We said that Machine Learning is like learning from experience. With learning we mean improving at some task over time, but we have to

establish what constitutes an improvement, in order to improve the model.
Formally, we need to have measures of how good (or bad) our models are. These measures are called *objective functions*. By convention, we usually define objective functions so that lower is better. This is merely a convention. You can take any function for which higher is better, and turn it into a new function that is qualitatively identical but for which lower is better by flipping the sign. Because lower is better, these functions are sometimes called *loss functions*.
Before of talking of the loss functions, we have to introduce some concepts about *information theory*.

Information theory

Information theory was originally proposed to study the information contained in signals passed through a noisy channel, and it is now a building block of Machine Learning. Some measures from information theory will be used to characterize the discrepancy between two probability distributions in terms of the information that they encode, providing us a measure of "goodness" of the probability distribution that our model is learning compared to the empirical distribution of the data. The basis of information theory is the *self-information*, a measure which determine the information content of an event $x \in X$:

$$I(x) = -\log P(x) \tag{2.5}$$

where log is the natural logarithm.
Another fundamental measure is the *Shannon entropy* that give us the expectation of information when an event x is drawn from a probability distribution P:

$$H(x) = E_{x\sim P}[I(x)] = -E_{x\sim P}[logP(x)] \tag{2.6}$$

Basically, this is a measure of the uncertainty contained in a probability distribution.
We can define the *Kullback-Leibler divergence* (KL divergence) extending the concept of entropy to two distributions, measuring the

relative entropy between P(x) with respect to another probability distribution Q(x):

$$D_{KL}(P||Q) = E_{x\sim P}[log\frac{P(x)}{Q(x)}] = E_{x\sim P}[logP(x) - logQ(x)] \quad (2.7)$$

It is always non-negative and zero if and only if the two distribution are equal.
A quantity closely related to KL divergence is the *cross-entropy*, defined as:

$$H(P, Q) = -E_{x\sim P}[logQ(X)] \quad (2.8)$$

Now we have the fundamental tools of information theory to proceed the discussion about objective functions.

Recall that equation (2.1) defined a linear regression model. In order to learn the weights θ we can minimize the *error* of the model, which represents a measure of how far is the model's prediction $\hat{y}$ from the true value y. A common error measure is the *Mean Squared Error*, defined as:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \quad (2.9)$$

The MSE can be viewed also as the cross-entropy between the empirical distribution and a Gaussian model.
A common way to find the solution to this problem is using the method of *least squares*. With a matrix notation:

$$X\theta = y \quad (2.10)$$

Using the *normal equation*

$$X^T X \quad (2.11)$$

we can minimize the sum of the squared differences between the two members of the following equation, yielding to the desired parameters θ

$$\begin{aligned} X^T X\hat{\theta} &= X^T y \\ \hat{\theta} &= (X^T X)^{-1} X^T y \end{aligned} \quad (2.12)$$

In case of multi-class classification, as written before, the logistic regression produce probability given by eq. (2.3) which uses the

sigmoid defined in eq. (2.2).
To learn the weights, we can minimize the average cross-entropy over all examples in our data:

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n} H(p, \hat{p}; x_i) \tag{2.13}$$

The cross-entropy in the previous equation is defined between the empirical conditional probability $p(y|x)$ and the probability of our model $\hat{p}(y|x;\theta)$ for each example x:

$$H(p, \hat{p}; x) = -\sum_{i=1}^{C} p(y_i|x) log\hat{p}(y_i|x;\theta) \tag{2.14}$$

In case of binary classification this last equation becomes simpler:

$$H(p, \hat{p}; x) = -(1-y)log(1-\hat{y}) - ylog\hat{y} \tag{2.15}$$

There are many other different loss function, which can be useful for different application, like the *Triplet Loss* useful in some image recognition tasks, or the *hinge loss* used in *Support Vector Machines* and for NLP tasks. We will not define these and other loss functions because throughout the experiments of the thesis it will be used only Cross-entropy, Binary Cross-Entropy and Mean Squared Error.
In contrast to linear regression with MSE, there is typically no closed-form solution to obtain the optimal weights for most loss functions. Instead, we iteratively minimize the error of our model using an algorithm known as *gradient descent*, that will be introduced in the next paragraph.

2.1.4 Optimization Algorithms

Once we have got some data source and representation, a model, and a well-defined objective function, we need an algorithm capable of searching for the best possible parameters for minimizing the loss function. Popular optimization algorithms for deep learning are based on an approach called *gradient descent*. In short, at each step, this method checks to see, for each parameter, which way the training set loss would move if you perturbed that parameter just a small

amount. It then updates the parameter in the direction that may reduce the loss. Let's see the details.
Gradient descent is a method used to minimize a cost function $J(\theta)$. The model's parameters $\theta \in R^d$ are updated in the opposite direction of the *gradient* $\nabla_\theta J(\theta)$ of the function. The gradient is a vector that contains all the partial derivatives $\frac{\partial}{\partial \theta_i} J(\theta)$. The i-th element of the gradient is the partial derivative of $J(\theta)$ wrt θ_i.
The formula to update the parameters is the following:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta) \tag{2.16}$$

where η is the *learning rate*, a parameter that determines the amplitude of an update of the parameters. It is a very important hyperparameter when we have to train a model. To guarantee the convergence of the algorithm, the learning rate is often reduced over the course of training. Typical value of learning rates are orders of $10^{-}n$ with n=1,2,3...
Previously we saw that in order to learn the weights we minimize the average cross-entropy over all examples in our data. In general, we minimize the expected value of a loss function over the empirical distribution of our data:

$$J(\theta) = E_{x,y \sim p_{data}} L(x, y, \hat{y}, \theta) = \frac{1}{n} \sum_{i=1}^{n} L(x, y, \hat{y}, \theta) \tag{2.17}$$

Thus, the gradient $\nabla_\theta J(\theta)$ is equal to:

$$\nabla_\theta J(\theta) = \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta L(x_i, y_i, \hat{y}_i, \theta) \tag{2.18}$$

This approach is called *batch gradient descent*. The problem is that it is an expensive method, indeed, before an update we have to compute the gradient for all examples in the data. An alternative approach is the *stochastic gradient descent* which iterates through the data, computes the gradient, and performs an update for each example i (so, basically it uses a batch size of 1):

$$\nabla_\theta J(\theta) = \nabla_\theta L(x_i, y_i, \hat{y}_i, \theta) \tag{2.19}$$

This approach is cheaper than the batch gradient descent, but the problem now is that the resulting gradient estimate is less accurate. For this reason, the most common approach is the *mini-batch gradient descent*, that is a stochastic gradient descent with mini-batches, and which computes the gradient over a *mini-batch* of m examples:

$$\nabla_\theta J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L(x_i, y_i, \hat{y}_i, \theta) \tag{2.20}$$

with the mini-batch size m that ranges from 2 to a few hundred and enables training of large models on datasets with hundreds of thousand of examples. Typical values of this hyperparameter are powers of 2, for memory reasons, generally 64, 128, 256, 512 or 1024. So, the batch size defines the number of samples that will be propagated through the network.
For instance, let's say you have 1050 training samples and you want to set up a batch size equal to 100. The algorithm takes the first 100 samples (from 1st to 100th) from the training dataset and trains the network. Next, it takes the second 100 samples (from 101st to 200th) and trains the network again. We can keep doing this procedure until we have propagated all samples through of the network. Problem might happen with the last set of samples. In our example, we've used 1050 which is not divisible by 100 without remainder. The simplest solution is just to get the final 50 samples and train the network, or to discard them.
Advantages of using a batch size lower than the number of all samples:

- It requires less memory. Since you train the network using fewer samples, the overall training procedure requires less memory. That's especially important if you are not able to fit the whole dataset in your machine's memory.

- Typically networks train faster with mini-batches. That's because we update the weights after each propagation. In our example we've propagated 11 batches (10 of them had 100 samples and 1 had 50 samples) and after each of them we've updated our

network's parameters. If we used all samples during propagation we would make only 1 update for the network's parameter.

Disadvantages of using a batch size < number of all samples:

- The smaller the batch the less accurate the estimate of the gradient will be. In the figure below, you can see that the direction of the mini-batch gradient (green color) fluctuates much more in comparison to the direction of the full batch gradient (blue color).

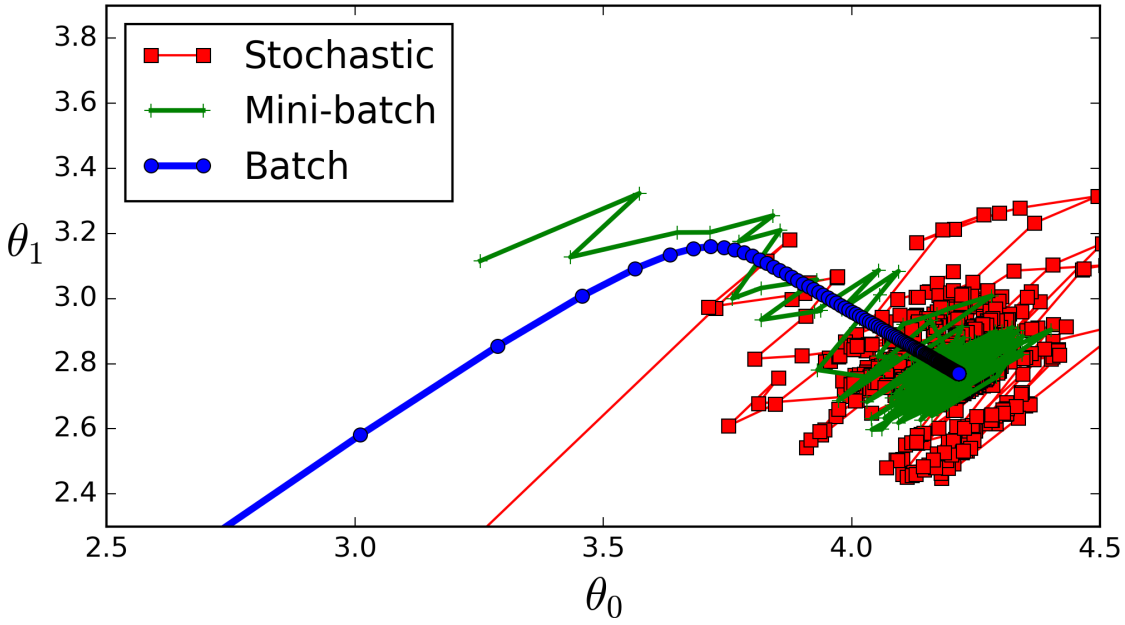


Figure 2.1: Batch size comparison

2.1.5 Generalization

The goal of Machine Learning is to generalize, which means training a model that performs well on new and previously unseen inputs. To this end, the available data X is typically split into 3 parts:

- **Training set**
  [9] A set of examples used for learning: to fit the parameters of the classifier. We would use the training set to find the "optimal" weights.

- **Validation set**
  A set of examples used to tune the hyper-parameters of a classifier. We would use the validation set to find the "optimal" values of hyper-parameters or determine a stopping point for the algorithm.

- **Test set**
  A set of examples used only to assess the performance of a fully-trained classifier. We would use the test to estimate the error rate after we have chosen the final model. After assessing the final model on the test set, you must not tune the model any further.

We might wonder why we have to separate validation and test sets. The reason is that the error rate estimate of the final model on validation data will be biased (smaller than the true error rate) since the validation set is used to select the final model. After assessing the final model on the test set, you must not tune the model any further.

During training, we compute the *training error*, the error of the model on the training set, which we try to minimize. However, we are interested in the *test error*, also said *generalization error*, that measures the model's performance on the test set, which the model has never seen before. This is also the main difference between the fields of Machine Learning and Optimization: the first one tries to minimize the generalization error, while the latter tries to minimize the training error.

All the sets are assumed to be *i.i.d.* which means *independent and identically distributed*. Indeed, the samples of each dataset are independent from each other and the datasets are identically distributed, in the sense that they are drawn from the same probability distribution. However, it can happen that for example training data are taken from Internet, for example high resolution images, while validation and test set are uploaded by the users, and these images can be blurried.
If the model is not able to obtain a low error on the training set, it is said to have high *bias*, which means that we made some wrong

assumptions in the learning algorithm that causes the loss of relevant relations in the data. Instead, if the gap between the training error and the test error is too large, the model has high *variance*, which means it is sensitive to small fluctuations and noise in the training data.
This dichotomy is knows as *bias-variance trade-off*.
Most used splits are 60% for training, 20% for validation and the remaining 20% for test set, or 70-15-15, or 80-10-10. Today with millions of data, the trend is to have validation and test sets much smaller, because the goal of validation is to check which algorithm works better, and you might not need 20% for example.
The trade-off between bias and variance is easy to understand, but difficult to master. Recall that bias is the difference between the average prediction of our model and the correct value which we are trying to predict, and variance is the variability of model prediction for a given data point or a value which tells us spread of our data. [10].
Mathematically, let the variable we are trying to predict as Y and other covariates as X. We assume there is a relationship between the two such that:

$$Y = f(X) + e \tag{2.21}$$

Where e is the error term, normally distributed with a mean of 0.
If we create a model $\hat{f}(X)$ of f(X) using linear regression or any other modeling technique, then the expected squared error at a point x is:

$$Err(x) = E[(Y - \hat{f}(x))^2] \tag{2.22}$$

The error $Err(x)$ can be decomposed in:

$$\begin{aligned} Err(x) &= (E[\hat{f}(x)] - f(x))^2 + E[(\hat{f}(x) - E[\hat{f}(x)])^2] + \sigma_e^2 \\ Err(x) &= Bias^2 + Variance + IrreducibleError \end{aligned} \tag{2.23}$$

We can see that Err(x) is the sum of $Bias^2$, Variance, and Irreducible Error. Irreducible error is the error that can't be reduced by creating good models. It is a measure of the amount of noise in our data. Here it is important to understand that no matter how good we make our model, our data will have certain amount of noise or irreducible

error that can not be removed.
In *supervised learning*, **underfitting** happens when a model is unable to capture the underlying pattern of the data. These models usually have high bias and low variance. It happens when we have very less amount of data to build an accurate model or when we try to build a linear model with a nonlinear data. Also, these kind of models are very simple to capture the complex patterns in data like Linear and logistic regression.
In supervised learning, **overfitting** happens when our model captures the noise along with the underlying pattern in data. It happens when we train our model a lot over noisy dataset. These models have low bias and high variance. These models are very complex.

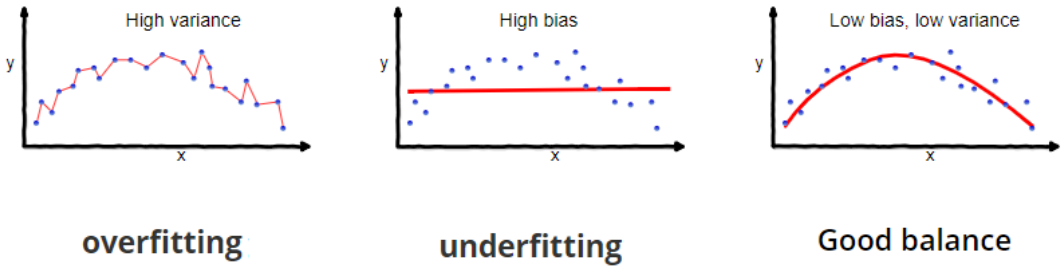


Figure 2.2: Underfitting, balance and overfitting

To build a good model, we need to find a good balance between bias and variance such that it minimizes the total error Err(x).

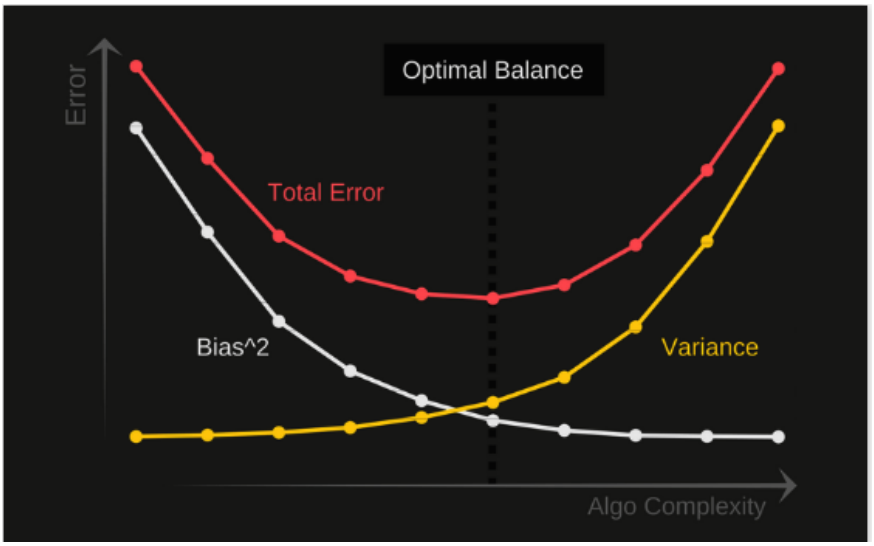


Figure 2.3: Optimal balance

An optimal balance of bias and variance would never overfit or underfit the model.

We can try different things if our model suffers from high bias or variance. A basic recipe to follow:

- 1^{st} question: the model has high bias? Look at the training performance, and if there is no high bias you could try a more complex model, or train longer.
- 2^{nd} question: there is high variance? We look at validation performance. To solve high variance problems, we can try training with more data or some technique of *regularization*.

If you suspect your model is overfitting, so it has high variance, one of the first thing you can do is regularization.

2.1.6 Regularization

Regularization is a way to modify a model's capacity to encourage the model to prefer certain functions in its hypothesis space over others. The most common way to achieve this is by adding a regularization term $\Sigma(\theta)$ to the cost function $J(\theta)$:

$$J(\theta) = MSE + \lambda\Sigma(\theta) \tag{2.24}$$

where λ controls the strength of the regularization. If $\lambda = 0$, we impose no restriction. As λ grows larger, the preference that we impose on the algorithm becomes more prominent. So, λ is an hyperparameter that you have to tune.
The most popular forms of regularization leverage common vector norms. *l1* regularization places a penalty on the l1 norm, i.e. the sum of the absolute values of the weights and is defined as follows:

$$\Sigma(\theta) = ||\theta||_1 = \sum_i |\theta_i| \tag{2.25}$$

where $\theta_i \in R$. l1 regularization is also known as *lasso* (least absolute shrinkage and selection operator) and is the most common way to induce sparsity in a solution as the l1 norm will encourage most weights to become 0, and this can be useful to compress the model and consume less memory.
However, *l2* norm is the most used. It is defined as:

$$\Sigma(\theta) = ||\theta||_2^2 \tag{2.26}$$

where $||\theta||_2^2 = \sqrt{\sum_i \theta_i^2}$ is the *Euclidean norm*, or l2 norm. Somewhat counter-intuitively, l2 regularization thus seeks to minimize the squared l2 norm as in practice, the squared l2 norm is often more computationally convenient to work with than the l2 norm. For instance, derivatives of the squared l2 norm with respect to each element of θ depend only on the corresponding element, while derivatives of the l2 norm depend on the entire vector.
For example, let's apply l2 regularization using logistic regression. You want to minimize the cost function J, $\min_{w,b} J(w,b)$ with $J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^i, y^i)$ $w \in R^n, b \in R$. To add regularization to logistic regression, we add a term to the ost function:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^i, y^i) + \frac{\lambda}{2m}||w||_2^2 \tag{2.27}$$

with m that is the number of training samples.
Note that we considered only the weights w, and not the bias b. However, you could add $\frac{\lambda}{2m}b^2$, that here is omitted for simplicity.
l2 regularization is also called *weight decay* because during the update

phase (2.16), here $\nabla_\theta J(\theta)$ is increased, so in the formula, θ will be lower:

$$\theta = \theta - \frac{\eta\lambda}{m}\theta - \eta \cdot \nabla_\theta J(\theta) = \theta(1 - \frac{\eta\lambda}{m}) - \eta \cdot \nabla_\theta J(\theta) \tag{2.28}$$

So, weight decay is like the ordinary gradient descent, but also multiplying θ by $1 - \frac{\eta\lambda}{m}$.
The reason why regularization reduces overfitting is in the λ hyperparameter. If it is high, then the weights will be close to 0 for a lot of units of the model (in particular, *hidden units* if your model is a neural network), which means you are zeroing out the impact of them, resulting in a simpler model.

2.2 Neural Networks

Neural networks (NN) are a computational model made of artificial "neurons", inspired by a simple biological neural network. In this section, we will give an overview of the fundamental building blocks used in neural networks. Neural networks can be seen as compositions of functions. In fact, we can view the basic machine learning models described so far, linear regression and logistic regression, as simple instances of a neural network.
Recall that the functions used for multi-class logistic regression are:

$$\begin{aligned} y &= f(x) = Wx + b \\ g(y) &= softmax(y) \end{aligned} \tag{2.29}$$

where $W \in R^{Cxd}$, $x \in R^d$, $b \in R^C$, $y \in R^C$, with C that is the number of classes and d is the dimensionality of the input. W is the matrix of weights.
Logistic regression can be seen as a composition of the functions f and g: g(f(x)) where f(·) is an affine function and g(·) is an *activation function*, in this case the softmax function.
A neural network is a composition of multiple such affine functions interleaved with non-linear activation functions. The softmax and sigmoid functions are common functions used at the final or *output layer* of a neural network to obtain a categorical distribution. Non-output layers are referred to as *hidden layers*.

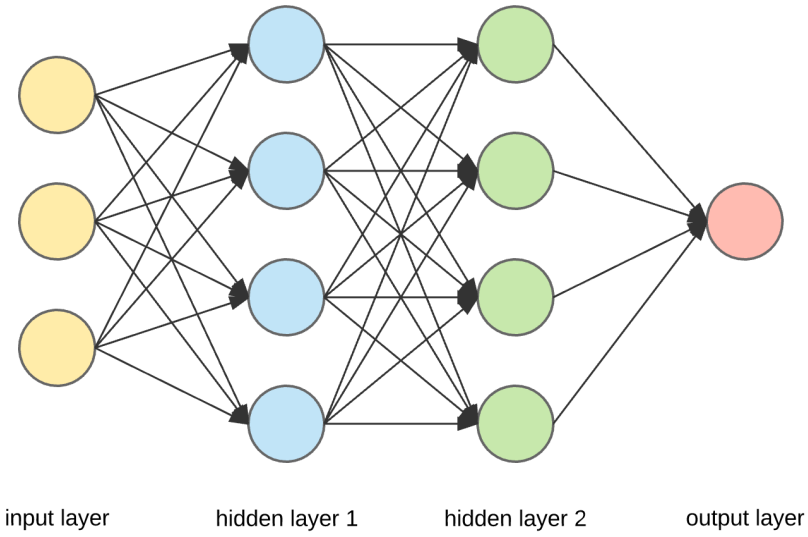


Figure 2.4: An example of a Neural Network

Linear regression can be seen as a neural network without a hidden layer and a linear activation function, the identity function $f(x) = x$, while logistic regression employs a non-linear activation function. Neural networks are typically named according to the number of hidden layers. A model with one hidden layer is known as a one-layer *feed-forward neural network*, which is also known as a *multilayer perceptron* (MLP), and its formulation is:

$$\begin{aligned} h &= \sigma_1(W_1 x + b_1) \\ y &= softmax(W_2 h + b_2) \end{aligned} \tag{2.30}$$

where σ_1 is the activation function of the first hidden layer. Note that each layer is parameterized with its own weight matrix W and bias vector b. The representation of a NN with n inputs, p hidden neurons and one output can be the following:

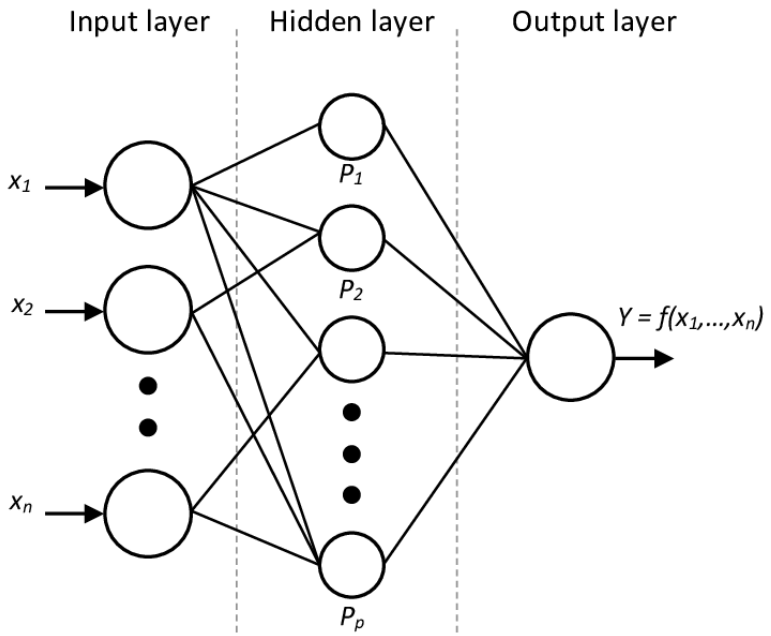


Figure 2.5: An example of a MLP

Computing the output of one layer, e.g. h that is fed as input to subsequent layers, which eventually produce the output of the entire network y is known as *forward propagation*. As a composition of linear functions can be expressed as another linear function, the expressiveness of deep neural networks mainly comes from its non-linear activation functions. Let's see the most used activation functions.

2.2.1 Activation functions

Until now we have seen the sigmoid function (2.2):

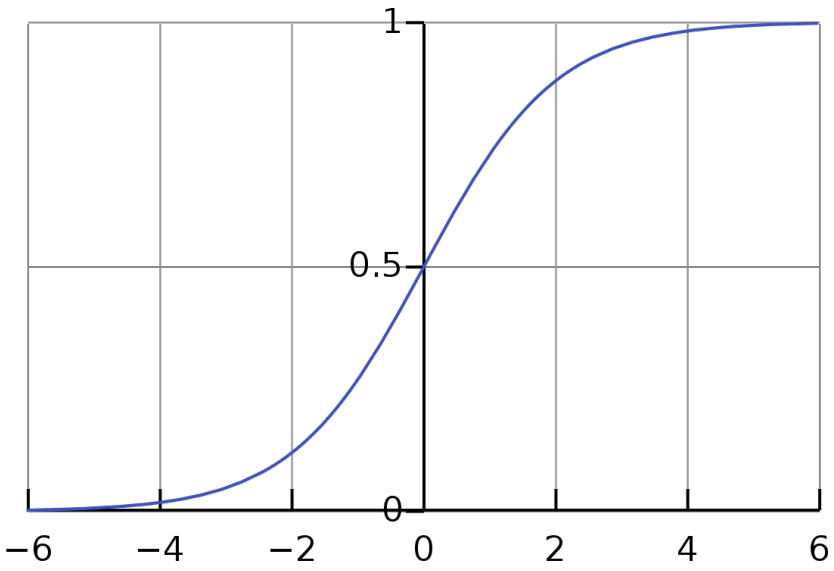


Figure 2.6: Sigmoid function

Another famous activation function is the *hyperbolic tangent*, defined as:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.31}$$

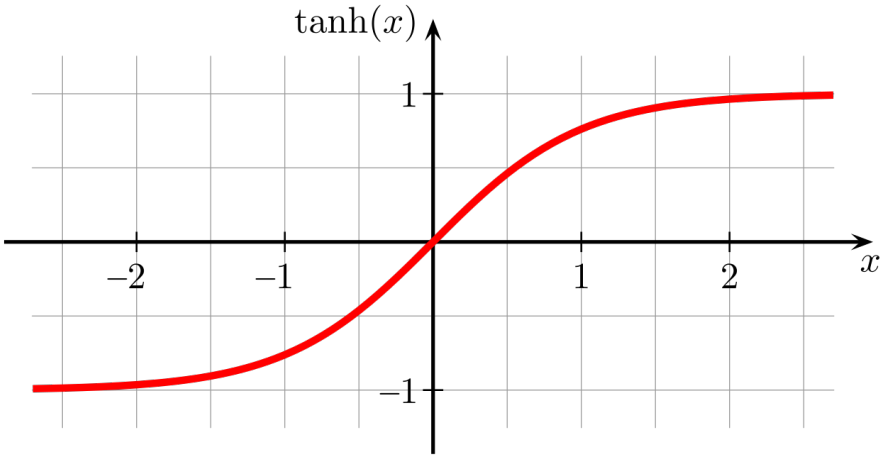


Figure 2.7: Hyperbolic tangent function

This almost works better than the sigmoid function because with values between -1 and +1, the mean of activations that come out of the hidden layer are close to mean 0, and sometimes when we

train an algorithm we might center the data, so with mean 0. In hyperbolic tangent is 0, instead in sigmoid is 0.5.
One exception is for the output layer because if y is either 0 or 1 then it makes sense for $\hat{y}$ to be a number, that you want to output, that is between 0 and 1 rather than between -1 and 1. So, one exception where I would use the sigmoid activation function is when you are using binary classification, for the output layer. So, the activation functions can be different for different layers.
One of the downsides of both sigmoid and tanh is that if x is either very large or very small, then the gradient of this function becomes very small because at the extremes of both these functions, the slope is close to 0. This can slow down the gradient descent.
So, another choice is the *Rectified Linear Unit*, ReLU, defined as:

$$ReLU = \sigma(x) = max(0, x) \tag{2.32}$$

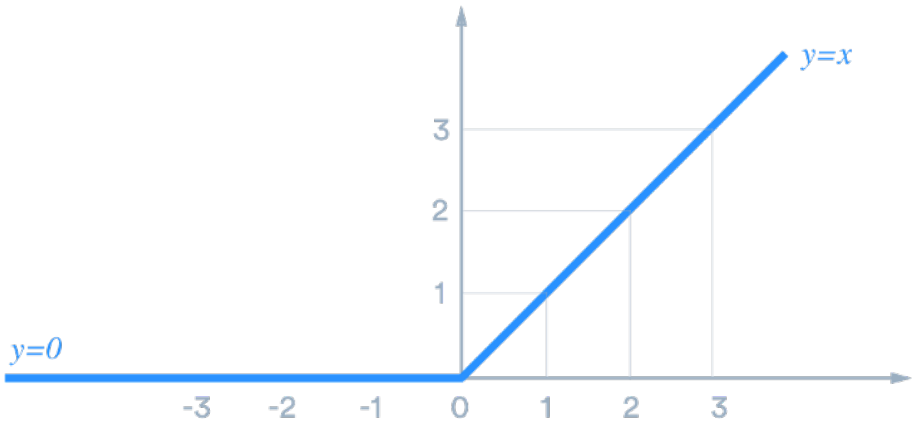


Figure 2.8: ReLU function

ReLU is the default option.
One disadvantage of ReLU is that the derivative is 0 when x is negative, so there is another version called *Leaky ReLU*, defined as:

$$LeakyReLU = \sigma(x) = max(ax, x) \tag{2.33}$$

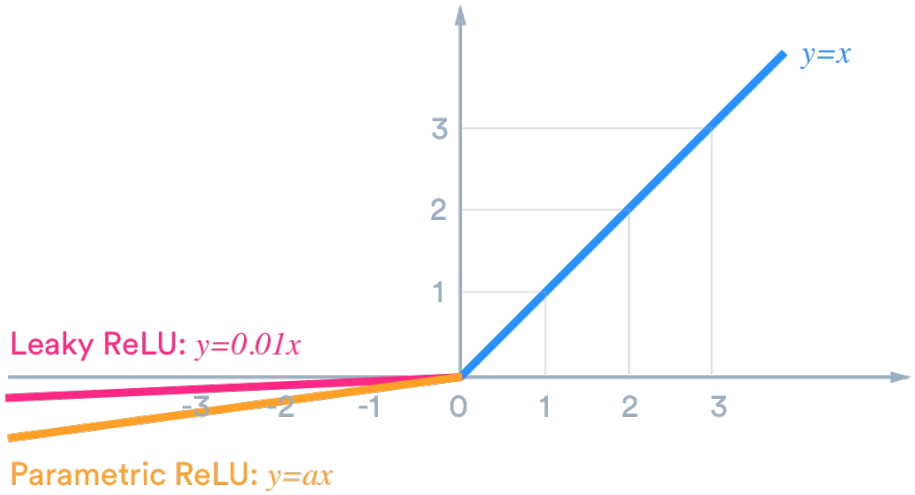


Figure 2.9: Leaky ReLU function

Advantages of both ReLU and Leaky ReLU is the fact that for the most time the derivative will not be 0, and this cause NN will learn faster.
These are the most used activation functions. Why we have to use activation functions? When we have deep neural networks, with lot of hidden layers, if you do not use an activation function, all these layers are just computing a linear function, so you might not have any hidden layers, and in general a linear hidden layer is more or less useless.

2.2.2 Normalization

During the training of a Neural Network, one technique that speed up training is the *normalization* of the inputs. Normalizing your inputs corresponds to 2 steps. The first step is to subtract the mean:

$$\begin{aligned} mean = \mu &= \frac{1}{m}\sum_{i=1}^{m} x^i \\ x &:= x - \mu \end{aligned} \tag{2.34}$$

and the second step is to normalize the variance, dividing x by the *standard deviation* σ:

$$\begin{aligned} \sigma^2 &= \frac{1}{m}\sum_{i=1}^{m}(x^i - \mu)^2 \\ x &:= \frac{x-\mu}{\sigma} \end{aligned} \tag{2.35}$$

Graphically, the result is this:

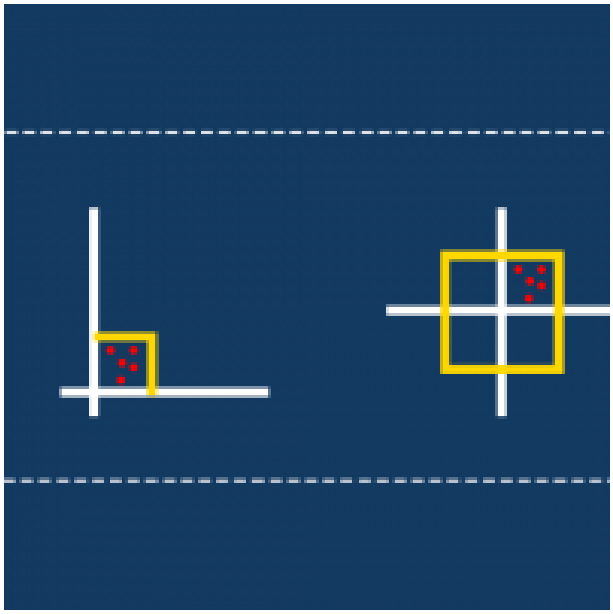


Figure 2.10: Normalization

If you normalize your training data, then you have to use the same μ and σ to normalize your test set. You do not want to normalize training and test differently.
The reason why we want to normalize input features is the following: Recall a generic cost function $J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^i, y^i)$. If you do not normalize the input features, then the cost function will look like a squashed out bowl. Considering only two input features, $x_1 and x_2$, and assuming that they are on different scales, for example $x_1 : 1, ..., 100$ and $x_2 : 0, ...1$ then the range for $w_1 and w_2$ will be different and so the function will be elongated. Whereas if you normalize features, then your cost function will look more symmetric. This means that if the inputs are not normalized you have to use a very small learning rate for gradient descent, because it may need a lot of steps to oscillate back before it finally finds the minimum, whereas in spherical contours, gradient descent can go straight to the so minimum:

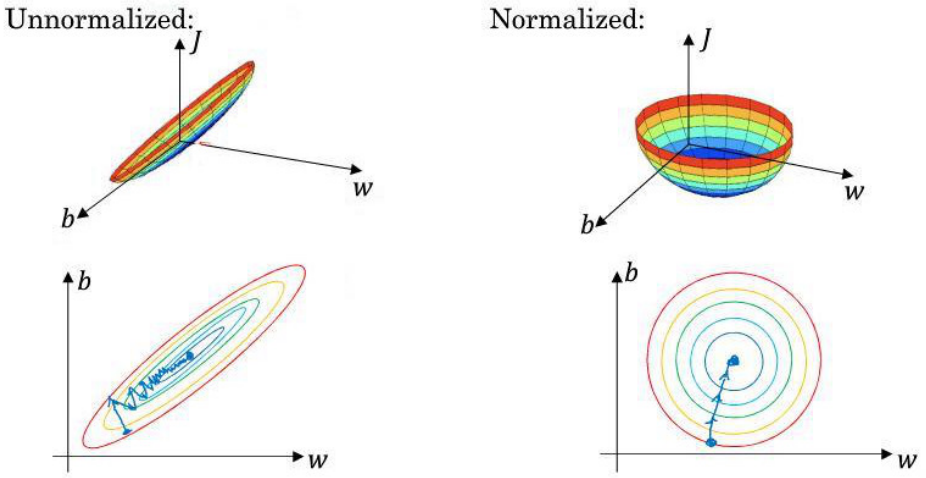


Figure 2.11: Why to normalize

Now, in deep learning we have deep models, like the one of Fig. 2.4. Here you have not just input features, but also activations. So, if you want to train the deeper parameters, then it would be nice if you can normalize the hidden mean and variance to make the training more efficient. This is what *batch normalization* does. Let's see how it works.

Given some intermediate values in your neural network, $z^1, ..., z^m$ from some hidden layer, you compute the mean $\mu = \frac{1}{m}\sum_{i=1}^{m} z^i$ and the variance $\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(z^i - \mu)^2$. So, finally you get $z^i_{norm} = \frac{z^i - \mu}{\sqrt{\sigma^2 + \epsilon}}$ which basically is the same formula of before, where you divide by standard deviation, but with the addition of $\epsilon > 0$ to avoid dividing by 0 in some situations.

Now z has mean 0 and standard deviation 1, but we do not want the hidden units to always have mean 0 and standard deviation 1. Maybe it makes sense for hidden units to have a different distribution; so, we compute this: $\tilde{z}^i = \gamma z^i_{norm} + \beta$ with γ and β that are learnable parameters. You will update both these parameters just as you update the weights of a NN. The effect of these parameters is that allows you to set the mean of $\tilde{z}$ to whatever you want it to be. By choosing the values of γ and β you are allowed to make hidden unit values to have other means and variances as well. This can be useful, for example, when you want data clustered with certain distribution in order to take advantage of the nonlinearity of the sigmoid function,

rather than having all the values in a linear regime.

2.2.3 Back-propagation

We saw that neural networks are trained with some algorithm of gradient descent. Each model has multiple layers, so calculating the gradient of the loss with respect to the parameters of each layer is not simple. A dynamic algorithm called *back-propagation*[19] is used to compute the gradients.
Back-propagation, which stands for backward propagation of the errors, is based on the *chain rule* of calculus, which given functions $y = g(x)$ and $z = f(g(x)) = f(y)$ defines the derivative $\frac{\partial z}{\partial x}$ of z with respect to x as the derivative of z with respect to y times the derivative of y with respect to x: $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$.
To understand how back-propagation works, we will see a very simple example.[20]
Suppose we have two inputs, $i_1 = 2$ and $i_2 = 3$ associated with an output $out = 1$, and some initial weights $w_1 = .11, w_2 = .21, w_3 = .12, w_4 = .08, w_5 = .14$ and $w_6 = .15$.

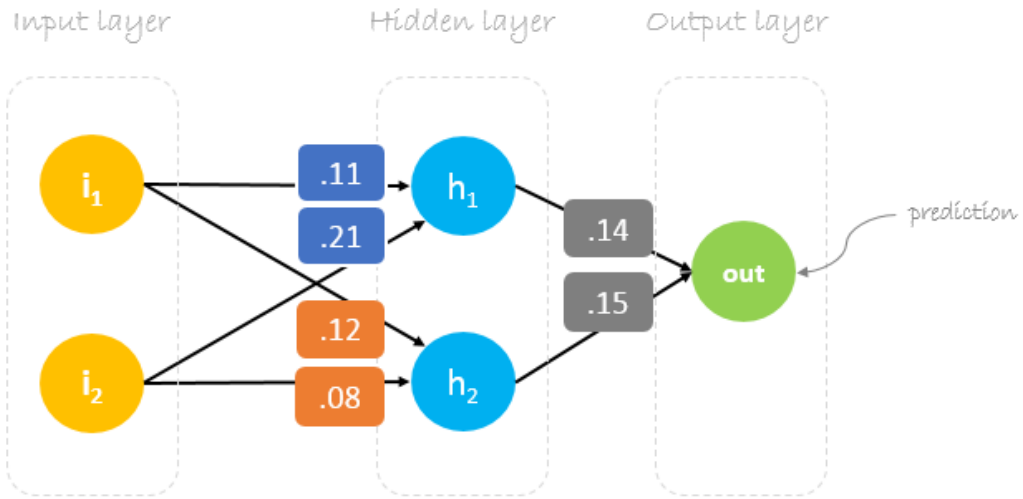


Figure 2.12: Example network

We will use given weights and inputs to predict the output. Inputs are multiplied by weights; the results are then passed forward to next

layer.

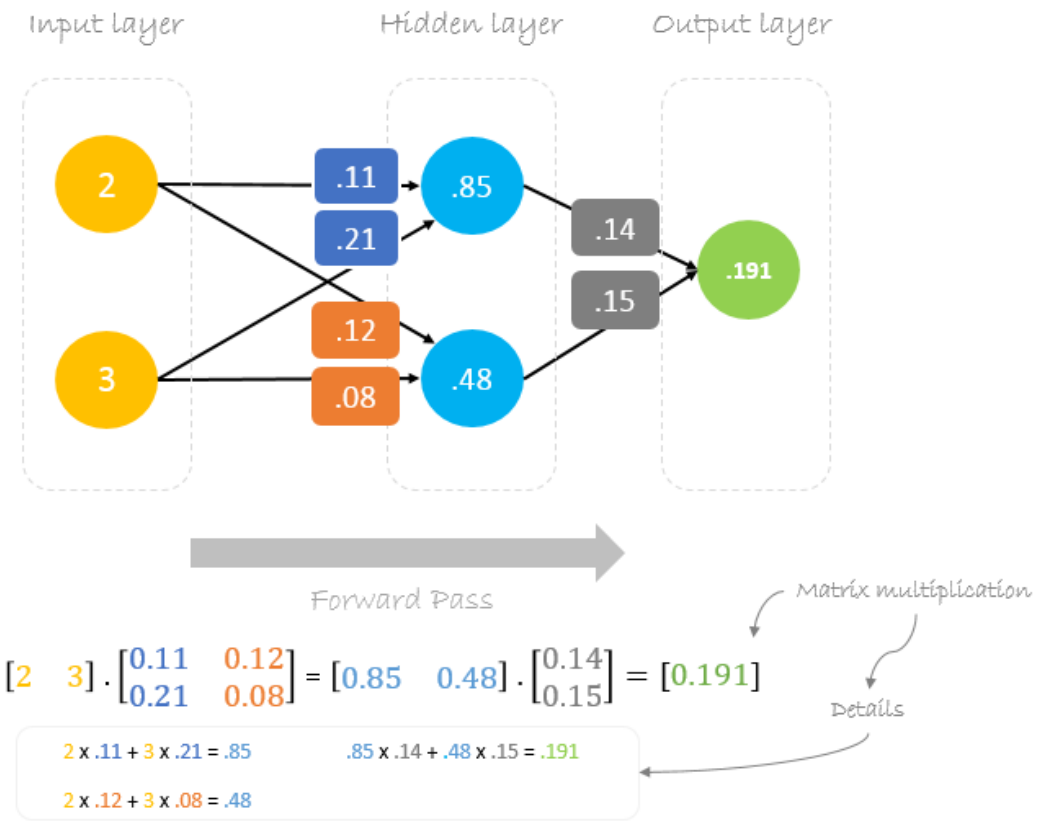


Figure 2.13: Forward pass

Now, it's time to find out how our network performed by calculating the difference between the actual output and predicted one. We can calculate the difference or the error as $Error = \frac{1}{2}(prediction - actual)^2$. It's clear that our network output, or prediction, is not even close to actual output, indeed $Error = \frac{1}{2}(0.191 - 1)^2 = 0.327$. We want to reduce the error, but in order to reduce the error we must change the prediction, and in order to change the prediction we must change the weights. To achieve this, we can use back-propagation, which calculates the gradient of the error function with respect to the neural network's weights. The calculation proceeds backwards through the network. In our case we want to minimize the error function. To find a local minimum of a function using gradient descent, one takes steps proportional to the negative of the gradient of

the function at the current point:

$$W_x^* = W_x - \alpha \frac{\partial Error}{\partial W_x} \tag{2.36}$$

Where W_x^* is the updated weight, W_x is the actual weight and α is the learning rate. For example, to update w_6, we take the current w6 and subtract the partial derivative of error function with respect to w6 $W_6^* = W_6 - \alpha \frac{\partial Error}{\partial W_6}$.
The derivation of the error function is evaluated by applying the chain rule as following:

$$\begin{aligned}
&\frac{\partial Error}{\partial W_6} = \frac{\partial Error}{\partial prediction} * \frac{\partial prediction}{\partial W_6} \\
&\frac{\partial Error}{\partial W_6} = \frac{\frac{(prediction-actual)^2}{2}}{\partial prediction} \frac{\partial (i_1 w_1 + i_2 w_2) w_5 + (i_1 w_3 + i_2 w_4) w_6}{\partial W_6} \\
&\frac{\partial Error}{\partial W_6} = 2 * \frac{1}{2}(prediction - actual) \frac{\partial (prediction - actual)}{\partial prediction} * (i_1 w_3 + i_2 w_4) \\
&\frac{\partial Error}{\partial W_6} = (prediction - actual) * (h_2) \\
&\frac{\partial Error}{\partial W_6} = \Delta(h_2)
\end{aligned} \tag{2.37}$$

with $\delta = (prediction - actual)$. So to update w_6 we can apply the following formula $W_6^* = W_6 - \alpha \Delta h_2$. The same for $w_5 = W_5 - \alpha \Delta h_1$. However, when moving backward to update w_1, w_2, w_3 and w_4 existing between input and hidden layer, the partial derivative for the error function with respect to w_1, for example, will be: $\frac{\partial Error}{\partial W_1} = \frac{\partial Error}{\partial prediction} * \frac{\partial prediction}{\partial h_1} * \frac{\partial h_1}{\partial W_1}$. Using derived formulas we can find the new weights. Now, using the new weights we will repeat the forward passed. We can repeat the same process of backward and forward pass until error is close or equal to zero.

2.2.4 Evaluation metrics

To evaluate Machine Learning and Deep Learning models, like Neural Networks, with respect to the test set, we use some common measures. All the evaluation metrics for a multi-class classification model can be understood in the context of a binary classification model.
These metrics are based on the following categories:

- True Positives (TP): Items where the true label is positive and whose class is correctly predicted to be positive.

- False Positives (FP): Items where the true label is negative and whose class is incorrectly predicted to be positive.
- True Negatives (TN): Items where the true label is negative and whose class is correctly predicted to be negative.
- False Negatives (FN): Items where the true label is positive and whose class is incorrectly predicted to be negative.

Accuracy is a common evaluation measure which counts the number of items correctly identified as either truly positive or truly negative out of the total number of items, and defined as:

$$Acc = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.38}$$

Accuracy for a multiclass classifier is calculated as the average accuracy per class.
Other frequently used measures are the *precision*, *recall* and F_1 *score*. Precision counts the number of items correctly identified as positive out of the total items identified as positive:

$$Prec = \frac{TP}{TP + FP} \tag{2.39}$$

Recall (also called *sensitivity*) counts the number of items correctly identified as positive out of the total actual positives:

$$Recall = \frac{TP}{TP + FN} \tag{2.40}$$

F_1 score is the harmonic average of the precision and recall and it measures the effectiveness of identification when just as much importance is given to recall as to precision:

$$F_1 = 2 \cdot \frac{P \cdot R}{P + R} \tag{2.41}$$

Obviously, there are also other measures like *Specificity, False Negative Rate and False Positive Rate.*

2.3 Common tasks and Architectures

We will now see some classes of neural networks commonly applied to image recognition, NLP, and other frequent tasks.

2.3.1 Recurrent Neural Networks and LSTM

Recurrent Neural Networks (RNN) [11] are models that can process a sequence of inputs. RNNs, rather than being "deep", are "wide", in the sense that they are unrolled through time. The RNN has trouble learning over a large number of time steps. Long-short term memory (LSTM) [12] networks try to solve the problem of learning over a large number of time steps of RNNs, with the techniques of "remember" and "forget". LSTM cells can be stacked in multiple layers. In most cases, we can use a bidirectional LSTM [13].
For NLP tasks RNNs and LSTMs are widely used. Other common networks for text inputs are the Gated Recurrent Unit (GRU) [14], and the state of the art, the Transformers [15], with their variants like BERT [16].

2.3.2 Convolutional Neural Networks

The advancements in Computer Vision with Deep Learning has been constructed and perfected with time, primarily over one particular algorithm, a Convolutional Neural Network (CNN) [17].
A Convolutional Neural Network is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. [18]

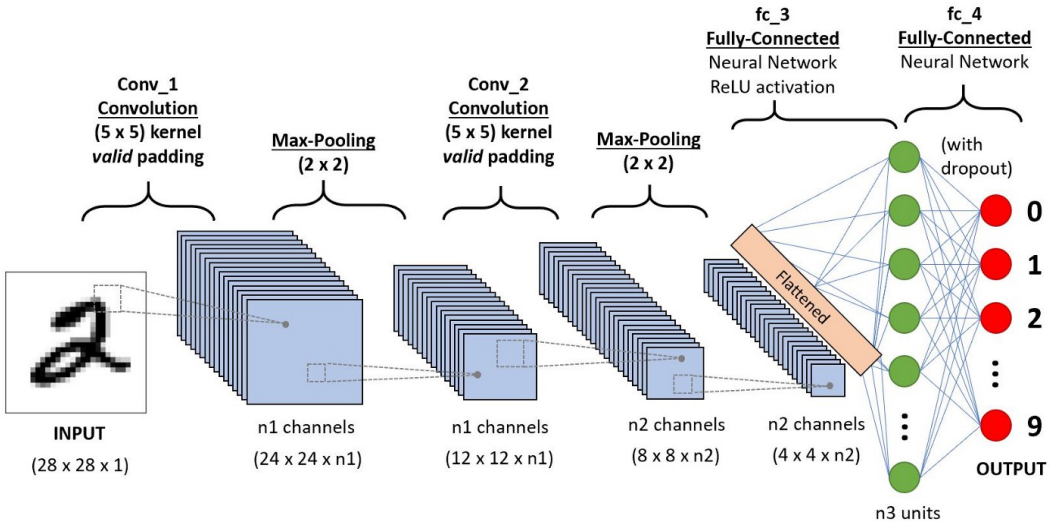


Figure 2.14: Example of CNN classifying handwritten digits

We might wondering why we have to use a CNN, instead of a classic Neural Network, indeed, an image is nothing but a matrix of pixel values. We can just flatten the image (e.g. 3x3 image matrix into a 9x1 vector) and feed it to a Multi-Level Perceptron for classification purposes.

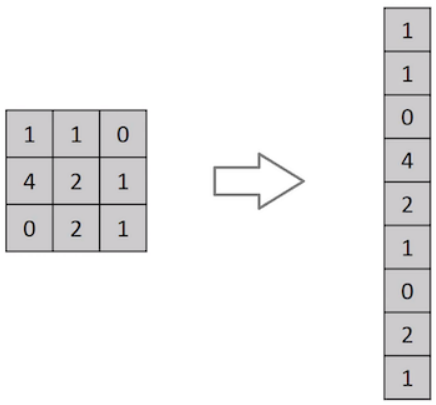


Figure 2.15: Flattening of a 3x3 image matrix into a 9x1 vector

However, in cases of extremely basic binary images, the method might show an average precision score while performing prediction of classes but would have little to no accuracy when it comes to complex images having pixel dependencies throughout.

Instead, a CNN is able to successfully capture the Spatial and Temporal dependencies in an image through the application of relevant filters. The architecture performs a better fitting to the image dataset due to the reduction in the number of parameters involved and reusability of weights. In other words, the network can be trained to understand the sophistication of the image better.
The role of the ConvNet is to reduce the images into a form which is easier to process, without losing features which are critical for getting a good prediction.
In these years, some CNN architectures have became popular, depending on their accuracy on ImageNet[6] Classification, which is a dataset of over 14 million images belonging to 1000 classes.

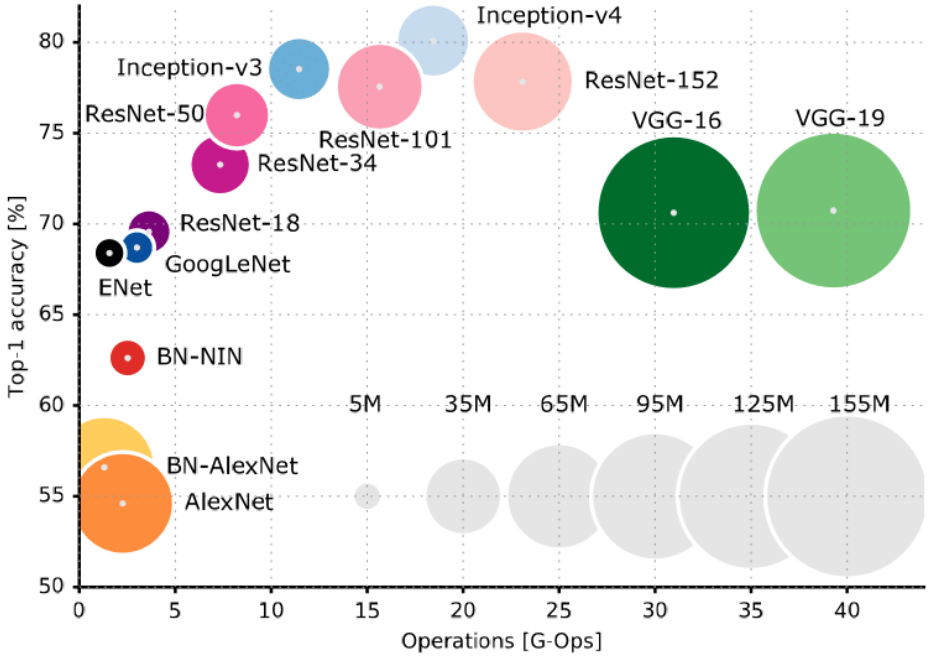


Figure 2.16: Comparison of popular CNN architectures

The vertical axis shows top 1 accuracy on ImageNet classification. The horizontal axis shows the number of operations needed to classify an image. Circle size is proportional to the number of parameters in the network.
All the experiments made for this thesis were done using the architecture of VGG16[5]. So, now we are going to see the details of this architecture.

VGG16

All the experiments were done using the architecture of the VGG16. It is a Convolutional Neural Network model proposed by K. Simonyan and A. Zisserman from the University of Oxford. The model achieves 92.7% top-5 test accuracy in ImageNet. It makes the improvement over AlexNet[7] by replacing large kernel-sized filters (11 and 5 in the first and second convolutional layer, respectively) with multiple 3×3 kernel-sized filters one after another. VGG16 was trained for weeks and was using NVIDIA Titan Black GPU's.

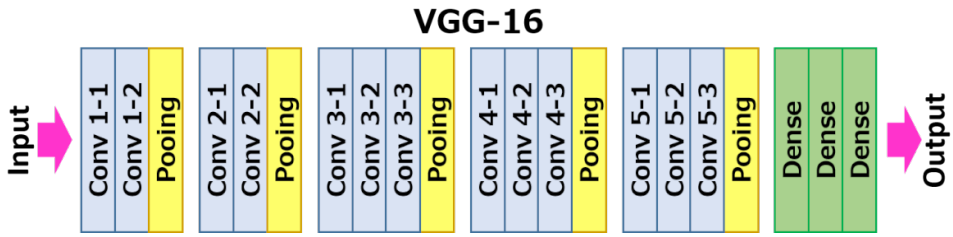


Figure 2.17: VGG16

The "16" and stands for the number of weight layers in the network. The architecture of a VGG16 is the depicted below:

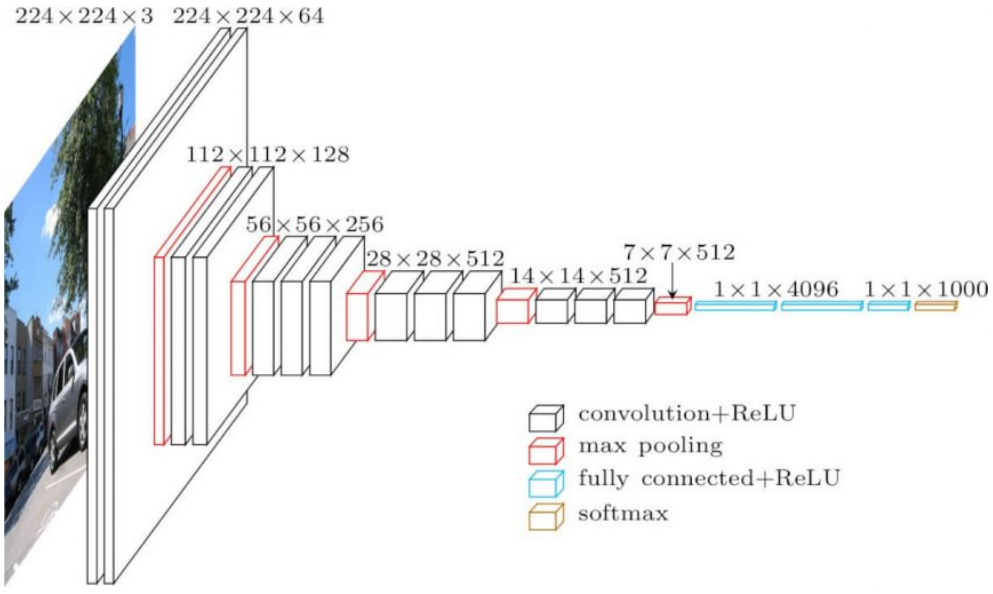


Figure 2.18: VGG16 Architecture

The input to conv1 layer is of fixed size 224 x 224 RGB image. The image is passed through a stack of convolutional (conv.) layers, where

the filters were used with a very small receptive field: 3×3 (which is the smallest size to capture the notion of left/right, up/down, center). In one of the configurations, it also utilizes 1×1 convolution filters, which can be seen as a linear transformation of the input channels (followed by non-linearity). The convolution stride is fixed to 1 pixel; the spatial padding of conv. layer input is such that the spatial resolution is preserved after convolution, i.e. the padding is 1-pixel for 3×3 conv. layers. Spatial pooling is carried out by five max-pooling layers, which follow some of the conv. layers (not all the conv. layers are followed by max-pooling). Max-pooling is performed over a 2×2 pixel window, with stride 2.

Three Fully-Connected (FC) layers follow a stack of convolutional layers (which has a different depth in different architectures): the first two have 4096 channels each, the third performs 1000-way ILSVRC classification and thus contains 1000 channels (one for each class). The final layer is the soft-max layer. The configuration of the fully connected layers is the same in all networks.

All hidden layers are equipped with the rectification (ReLU) non-linearity. It is also noted that none of the networks (except for one) contain Local Response Normalisation (LRN), such normalization does not improve the performance on the ILSVRC dataset, but leads to increased memory consumption and computation time.

2.3.3 Conclusions

In this chapter, we have given background knowledge in machine learning that is necessary for the subsequent chapters. We have also introduced the neural network methods and some tasks that we will work with throughout this thesis. The next chapter will discuss neural network methods for particular transfer learning scenarios in-depth and provide more intuitions around generalization in machine learning.

Chapter 3

Federated Transfer Learning

This chapter provides an overview of the literature of Federated and Transfer Learning in general and for computer vision in particular. As both Federated and Transfer Learning have been used to refer to different areas in different contexts, we will first provide a definition of Federated and Transfer learning.

3.1 Transfer Learning

Suppose that you are facing a supervised learning problem. You want to train a model for a task T_1 and a domain D_1, and you are provided with labelled data for this task and this domain. After the training we can check how the model performs on unseen data of the same task and domain.
If we have another task T_2 and another domain D_2, we require other labelled data for this task and this domain, to train a new model.
If we do not have enough data, our models, after the training, can be not reliable. One technique that allows us to deal with this scenario is Transfer Learning. It leverages data of a specific task and domain, called the *source task* and *source domain*, storing the knowledge obtained in solving the source task and domain, and applies this knowledge to another task and domain, the *target task* and the *target domain*.
So, basically transfer learning is the transfer of knowledge, that can assumes various forms, from a source to a target.

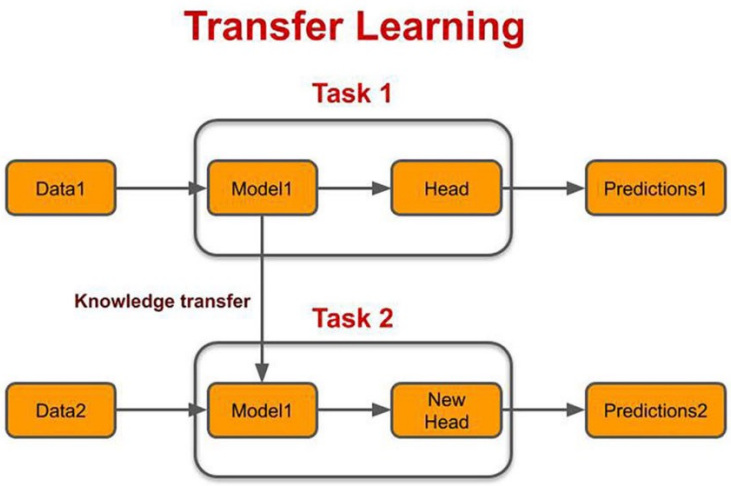


Figure 3.1: Transfer Learning

We now provide a formal definition of transfer learning, in terms of domain and tasks.
A domain D consists of a feature space X and a marginal probability distribution $P(X)$, where $X = (x_1, x_2, ..., x_n) \in X$. So, given this couple $D = X, P(X)$, a task is another couple made of a label space Y and an objective predictive function $f : X -> Y$. The function f is used to predict the corresponding label $f(x)$ of a new instance x. Task is denoted by $T = Y, f(x)$. Tasks can be learnt by the training data x_i, y_i with $x_i \in X$ and $y_i \in Y$.
Given a source domain D_S, a source task T_S, a target domain D_T and a target task T_T, with $D_S \neq D_T$ and $T_S \neq T_T$, transfer learning aims to help improve the learning of the target predictive function $f_T(\cdot)$ in D_T using the knowledge in D_S and T_S. Basically, the objective is to learn the target conditional probability distribution $P_T = Y_T|X_T$ in D_T with the information gained from the sources. We assume that an enough quantity of labeled target examples is available (or a huge amount of unlabeled target examples).
Inequalities among different members of the couples of both domain D and task T, lead to five transfer learning scenarios:

- $P_S(Y_S) \neq P_T(Y_T)$. This is the case in which the prior distribution of the source and the prior distribution of the target tasks are different. Basically, labels have different distribution.

This is important in generative and discriminative models (like GANs[21]), which model the prior explicitly.

- $P_S(Y_S|X_S) \neq P_T(Y_T|X_T)$. This is the case in which the conditional probability distributions of both source and target tasks are different. Basically, it can happen when source and target are unbalanced with respect to their classes. Some common approaches in this scenario are techniques of over-sampling or under-sampling, like SMOTE [22], Synthetic Minority Oversampling Technique.

- $Y_S \neq Y_T$. This is the case in which the label spaces of source and target are different. Here, it is important to specify if the tasks are learned sequentially or simultaneously: in the first case we talk of *sequential transfer learning*, while in the latter case we talk of *multi-task learning*.

- $P_S(X_S) \neq P_T(X_T)$. This is the case in which the marginal probability distributions of source and target are different. For example when the data refers to different fields. Typically we call this scenario as *domain adaptation*.

- $X_S \neq X_T$. This is the case in which feature spaces of both source and target are different. Considering a NLP scenario, this can be the case in which texts are written in different languages, and we refer to this as *cross-lingual learning*.

3.2 Federated Learning

Typically, the operations of learning and modeling of AI systems are placed in a cloud server or in a data center. The training of these AI models raises privacy issues such as data breaches, because these data are likely to contain sensitive information such as personal preferences or user addresses. Moreover, if we think to the scenario of *Internet of Things*, there are problems due to the critical limitations of IoT devices. Indeed, the huge amount of IoT data at the network edge complicated the offloading of these data to the remote servers, due to the required network resources and incurred latency. So, some

approach being efficient and privacy-preserving is necessary.
Federated Learning may address these challenges by training models where the data is spread across nodes, sharing only weights with the central aggregation node. For example, in the context of IoT systems, the devices can act as workers to communicate with an aggregator (e.g. a server) for performing neural network trining in intelligent network. In particular, the aggregator initializes a global model with its learning parameters; each worker downloads the current model from the aggregator, computes its model update by using its local dataset, and offload the computed local update to the aggregator, that combines all local model updates and construct a new improved global model. In this way privacy leakage is minimized thanks to the computing power of the distributed workers, that is also useful to improve the training quality. The process is repeated until the training is complete.
Applications can gain various benefits from FL:

- Data Privacy Enhancement: the aggregator does not require the raw data for training, minimizing the leakage of sensitive user information to external third-party. FL is a good solution for the privacy protection legislation such as the general Data protection Regulation (GDPR).

- Low-latency Network Communication: data are not transmitted to the aggregator, so this reduces the communication latencies caused by data offloading. Moreover, FL also saves network resources such as spectrum and transmit power.

- Enhanced Learning Quality: the overall training process and the accuracy rates can be substantially higher than using centralized AI approaches, because FL uses computation resources and diverse datasets from a network of IoT devices.

A short review of some applications in the literature is here provided.
FL was recently applied to different application domain, i.e. transporation. Fiosina [23] proposed a privacy- preserving FL model, which

achieves an accuracy level comparable to that of the centralized approach on the considered real-world dataset. This model predicts the Brunswick taxi travel time based on floating car data trajectories obtained from different taxi service providers, which should remain private. The model makes predictions for the stated problem and allows a joint learning process over different users, processing the data stored in each of them without exchanging their raw data, but only parameters, as well as providing joint explanations about variable importance. The proposed FL model explains strategy and illustrates it on a travel time prediction.
AI techniques have been extensively adopted in the domain of smart healthcare, to learn health data to facilitate healthcare services, for example medical imaging analysis for disease detection. Clearly, in this scenario of traditional AI, there are issues related to the privacy of the patients, caused by the public data sharing with the cloud or some data center. FL can be used as an alternative in this context, to provide intelligence with privacy awareness, where the data sharing is not needed. Hao et al. proposed a collaborative learning protocol based on FL for an Electronic Health Records (EHRs) Management system with the cooperation of multiple hospital institutions and a cloud server. In this model, each hospital runs a neural network using its own EHRs with the help of the cloud server. To ensure privacy for model parameters in the FL process, a lightweight data perturbation method was used in the training data, which thus can defend model memorization attack in the learning. Indeed, although attackers can obtain perturbed information on EHRs, it will be hard to reconstruct the original data. However, DL models generally show vulnerabilities to perturbed data that may compromise their effectiveness. Adversarial examples can fool Deep Neural Networks (DNNs) by introducing small perturbations on the input to produce poor results in the testing/validation stage. In safety- critical environments, where the adversarial attack may lead to significant risks, robustness has become an important concept. Recent studies investigated adversarial examples to provide the suitable countermeasures. However, adversarial training does not only guarantee robustness, but also interpretability. Indeed, it has been demonstrated that gradients from

adversarial trained model appear to be more interpretable than their standard counterparts.
Moreover, FL can promote secure healthcare cooperation for better medical service delivery. Yuan et al. [25] presented a cooperative healthcare framework enabled by FL among medical IoT devices, in which each device joins to run a NN using datasets of electrocardiogram, to build an arrhythmia detection model at a powerful server. Results showed that the model, tested on 64 IoT devices, can achieve a lower communication overhead, compared with FedAvg algorithm, with a small accuracy loss in an arrhytmia detection task.
Another application domain of FL is the one related to Unmanned Aerial Vehicles (UAVs). UAVs play an important role in various services such as goods delivery, disaster monitoring or military applications, and they are considered a key technology in 6G wireless networks. Because of the high mobility and altitude of UAVs, it is challenging to ensure the continuous communications between UAVs and base stations with respect to dynamic aerial environment conditions. So, especially when we have to transmit a huge amount of data over the aerial links, traditional AI approaches may not be a suitable solution. FL can provide better solutions for intelligent UAVs networks by using cooperation of multiple UAVs without transferring raw data to base stations and sacrificing data privacy. Shiri et al. [26] proposed a federated path control strategy for large-scale UAV networks. Each UAV runs a neural network and then share the model parameters with a central unit for obtaining a global model, which makes the estimation of the population density function at UAVs more accurate. FL aims to mitigate the volume of data transferred over the aerial environment and as a consequence to reduce communication delays and privacy concerns, while, at the same time, improving training of the global model, employing computing resources of all UAVs. The results confirmed that FL algorithms can reduce transmission time, motion of energy of UAV communications, as well as achieve minimum collision risks in windy environments.

3.3 Federated Transfer Learning

Federated Transfer Learning [30] is a special case of Federated Learning in which two datasets differ in the feature space. This can be applied to different datasets but similar in their nature, e.g. data collected from companies in the same sector, but with differences in the nature of business. These enterprises, also physically far in the world, share only a small overlap in feature space, and their datasets differ both in samples and in feature space.

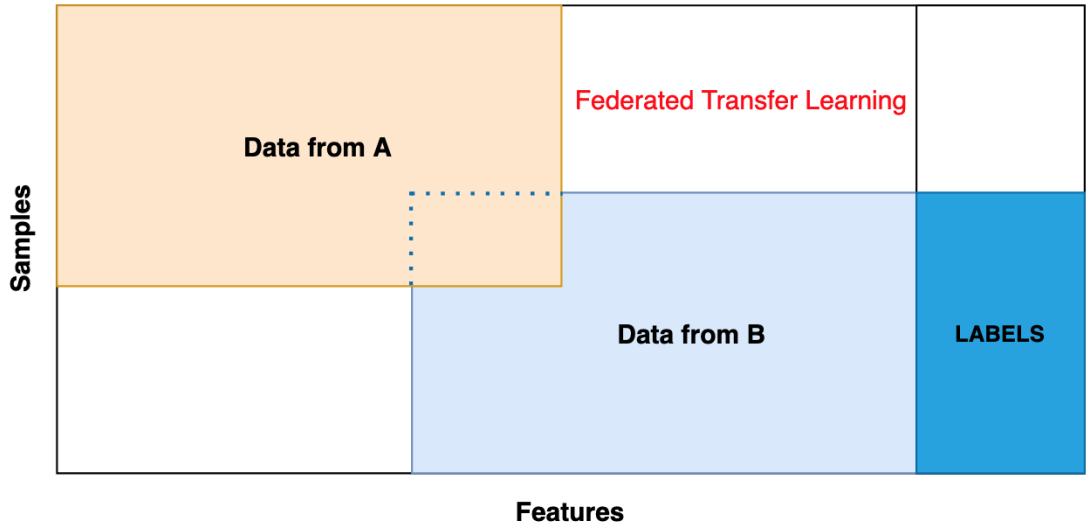


Figure 3.2: Federated Transfer Learning typical architecture

Transfer learning aims to build effective model for the target domain while leveraging knowledge from the source domains. Considering the two parties A and B of Fig. 3.2, where there is only a small overlap in feature space and sample space between A and B (dotted box), a model learned on B is transferred to A by leveraging small overlapping data and features. Other types of Federated Learning are used when there is a large overlap in the feature space between datasets (*horizontal federated learning*) or when there is large overlap in user/sample space between datasets (*vertical federated learning*). In contrast to these traditional methods, FTL is used when there is only a small overlap in feature space and sample space, ingesting a model trained on source domain samples and feature space. Then the model is oriented to a use in target space, where there are non-

overlapping samples, leveraging the knowledge acquired from source domain non-overlapping features.
Application scenarios of FTL can be:

- Wearable Healthcare.
  Multiple features and functionalities of wearable devices include remote patient monitoring, tracking and collecting data, enhancing everyday health and lifestyle patterns, detecting chronic conditions, among others. Data coming from these wearable devices are often fragmented and private, so it can be difficult to generate good models.
  In the healthcare domain, FTL can be applied also in electroencephalographic (EEG) signal classification. Brain-Computer Interface (BCI) systems aim to decode participants' brain states. However, the lack of large EEG datasets makes difficult the creation of models for classification of EEG recordings. Indeed, due to the privacy regulations and high data collection expenses, EEG data are available only as small datasets owned by companies.

- Autonomous Driving.
  With this term we refer to self-driving vehicles that can move without the external intervention of a human. Autonomous driving integrates technologies like sensing, perception, and decision.

- Image Steganalysis.
  Steganography is a data hiding technique that embeds the secret message inside a digital media for providing a method of invisible communication. mage steganalysis is a counter technique to image steganography. It aims to detect the hidden information in the digital images. Steganographers are reluctant to share their data, so there is a lack of datasets for training steganalysis methods. FedSteg [31] proposed federated transfer learning framework for image steganalysis in which the users do not leak the data to each other, instead it is assumed that there is a cloud model trained with the data on the cloud and then distributed to the users which can train their local model with local

data. Then the local models are sent to the cloud to create a new cloud model. In this way data are not shared, but only encrypted parameters.

As seen in these scenarios, data are often scattered across different enterprises and cannot be easily used due to many constraints. Federated Transfer Learning helps to improve modeling under a data federation. The data federation allows knowledge to be shared without compromising user privacy, and enables complementary knowledge to be transferred in the network.
Most works on FTL (like FedSteg) have adopted variants of deep learning as the architecture for FTL.

3.4 Conclusions

This chapter provided an introduction to Transfer Learning, Federated Learning and Federated Transfer Learning settings, with their state of the art applications. The next chapter will be about our proposed method of Federated Transfer Learning, which tackles some problems of both Federated and Transfer Learning.

Chapter 4

Proposed method

In this chapter will be described our proposed method of Federated Transfer Learning. Firstly will be provided the intuition of the method and then how we tried to make it work.

The general idea of our proposed method is to combine the weights of two different neural networks trained on two different datasets, but similar in their nature, and machines, and check how the performances change after the aggregation. This method falls in the field of Federated Transfer Learning because only the weights of the neural networks are shared (FL) and because it applies the knowledge obtained in solving the source task and domain in the target task and domain (TL).
We know that the weights are the real values that are associated with each feature which tells the importance of that feature in predicting the final value. Weights associated with each feature, convey the importance of that feature in predicting the output value. Features with weights that are close to zero said to have lesser importance in the prediction process compared to the features with weights having a larger value. However each weight has its own meaning and we are not able to say why a parameter has a specific value, what it represents and to which feature it is associated. Indeed, a known limitation of Deep Learning is that NNs are intrinsically unexplainable black-box models. Understanding how a DL model processes input data to identify patterns and provide a prediction on unseen data, is still an unclear process that even experts struggle with to

understand.
For this reason our method, which aggregates weights from distinct neural networks is very risky.
To understand how it works, suppose there are two simple neural networks, network A and network B, like the one in picture:

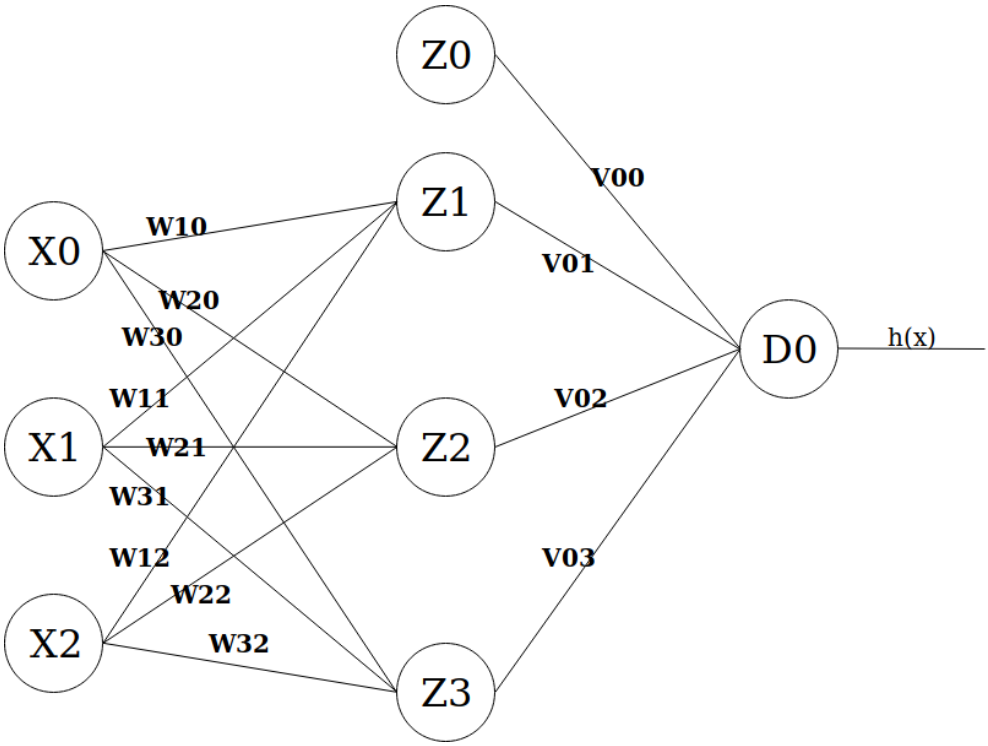


Figure 4.1: A one hidden layer Neural Network

This is a one hidden layer neural network. As shown in the picture, each link between input features, hidden neurons and the output contains a parameter (weight or bias). In order to do the aggregation of the parameters of the networks, obviously they must have the same architecture. Suppose that the weights between input features and the hidden layer of the first neural network are labeled as $W10_A$, $W20_A$, $W30_A$, $W11_A$, ..., $W32_A$ and the weights between hidden layer and output as $V00_A, V01_A, V02_A, V03_A$, while for the second neural network they are respectively $W10_B$, $W20_B$, $W30_B$, $W11_B$, ..., $W32_B$ and $V00_B$, $V01_B$, $V02_B$, $V03_B$. Suppose that there is a third neural network C which has the same identical architecture of nets A and B. The proposed method states that the weights of C will be equal to the result of an aggregation function applied

to to the parameters of A and B. If the aggregation function used is the sum, then the weights of C will be equal to $W10_C = W10_A + W10_B, W20_C = W20_A + W20_B, W30_C = W30_A + W30_B, W11_C = W11_A + W11_B, ..., W32_C = W32_A + W32_B$
For another example, if the selected aggregation function is the max operator, then the weights of C will be equal to $W10_C = max(W10_A, W10_B)$, $W20_C = max(W20_A, W20_B)$,
$W30_C = max(W30_A, W30_B), W11_C = max(W11_A, W11_B)$, ...,
$W32_C = max(W32_A, W32_B)$

Here we considered a simple neural network with only one hidden layer, 3 input features, 3 hidden neurons and one output.
The approach can be generalized to deeper neural networks with several hidden layers, hidden neurons, input features and outputs.
In formulas, considering n neural networks with the same architecture (same number of hidden layers, hidden neurons, input features and outputs), indicating with m the number of hidden layers, with p_m the number of hidden neurons of layer m, with $l = p_m \cdot p_{(m-1)}$ the total number of weights between one layer and another, then the generic parameter of the *j-th* neural network $w_{n,l}$ will be equal to

$$w_{j,k} = \sum_{i=1,\, g=1}^{n,l} (w_{i,g}) \tag{4.1}$$

with j one of the n neural networks, $k = 1, ..., l$ the index of the weight of the $j-th$ neural network, $i = 1, ..., n$ with $i \neq j$ and g the index of the weights of the $i-th$ neural network. The only requirement is that in order to aggregate the weights, the networks must have the same architecture, and the architecture used is the one of a VGG16, previously described in VGG16.
As written before, this method is very risky because each parameter has its own function. In particular, let us consider the convolution operation that is one of the fundamentals blocks of a Convolutional Neural Network. Edge detection is a basic example of convolution operation that is a fundamental element in the convolution layers. During convolution, a kernel (filter) will move around the input, and at each stop, it is computed the sum of the products of overlapping

values. The output of convolution is a matrix/vector with the collection of numbers that we computed at each stop. A vertical edge detector is a kernel with a set of positive numbers at the left and a set of negative numbers at the right. After convolution with an image, high numbers in the resulting matrix will tell us which part of the image has sudden changes in the pixel values from left to right.

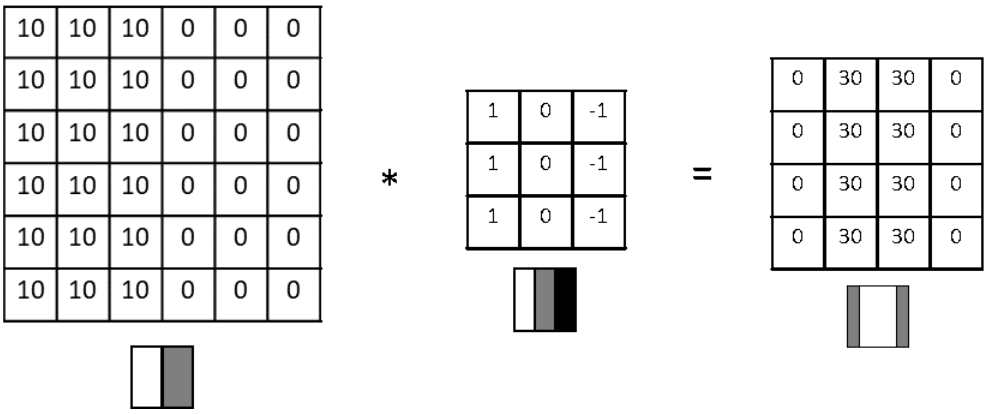


Figure 4.2: Convolution with a vertical edge detector

With the same logic, a horizontal edge detector has a set of positive numbers at the top and a set of negative numbers at the bottom.

Figure 4.3: Filters for vertical (left) and horizontal (right) edge detection

Early layers of CNN might detect edges then the middle layers will detect parts of objects and the later layers will put the these parts together to produce an output.

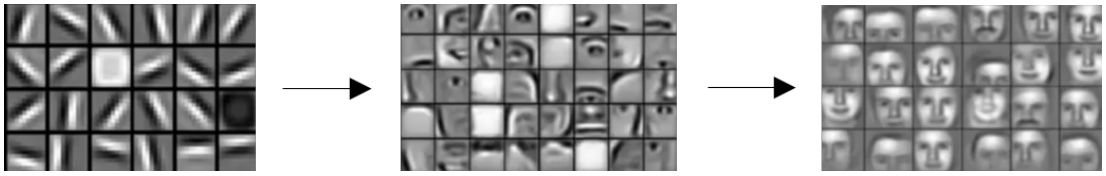


Figure 4.4: The process of detecting edges across layers

The simplest version of edge detectors use 1 and -1, but more sophisticated edge detectors like *Sobel* filter and *Scharr* filter use unique sets of numbers.
In Deep Learning, there is no need to hand craft these numbers. We try to learn those numbers by treating them as parameters. It can learn horizontal, vertical, angled, or any other edge type automatically rather than getting them by hand.
Considering what has just been said, it is clear that our method is tricky. Indeed, suppose that the neural network C sums the parameters of a horizontal filter of A with a vertical filter of B. As a numerical example, this can be the horizontal filter of A

1	1	1
0	0	0
-1	-1	-1

Table 4.1: Horizontal filter of network A

and this can be the vertical filter of B:

1	0	-1
1	0	-1
1	0	-1

Table 4.2: Vertical filter of network B

The filter of C will be equal to the element-wise sum of the parameters of A and C:

2	1	0
1	0	-1
0	-1	-2

Table 4.3: Filter of network C

Clearly, this last filter does not represent neither horizontal edges nor vertical edges. It is similar to a Sobel filter. The edge that it is able to recognize can be something similar to this picture:

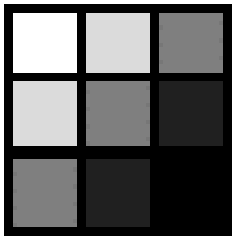


Figure 4.5: A filter given by the sum of an horizontal and vertical filter

The problem is that the filter obtained can recognize edges that are not present in the images, or, even worse, edges not clearly defined.
For this reason we want that the networks learn to sum their weights in an appropriate way during training. Our method aims to recognize the same edges across the different networks. For example if networks A and B have these filters:

0.6	0.1	-0.4	0.4	-0.2	-0.5
0.4	0.1	-0.5	0.5	0	-0.5
0.5	0.2	-0.6	0.5	-0.2	-0.5

Table 4.4: Hypothetical horizontal filters of network A (left) and network B (right)

Then the weights of network C will be:

1	-0.1	-0.9
0.9	0.1	-1
1	0	-1.1

Table 4.5: Filter of network C after aggregation

In this example both the filters of networks A and B are vertical filters but with different values. Combining them gives back another vertical filter with different intensities.
Moreover, the problem that filters of different networks do not match is present also from a stacked point of view. Indeed, the number of channels of the input matrix and the number of channels in each filter

must match in order to be able to perform element-wise multiplication. So, for example the first filter of A can recognize horizontal edges and the last one the vertical edges, while for network B the first filter can recognize vertical edges and the last one the horizontal ones.
Considering all these problems, it is clear that our method is difficult to get right. As we will see in the next chapter, several attempts were done in order to make the proposed method work, or at least to obtain acceptable results.
During our experiments we will use two different image datasets, but similar in their nature. Indeed, both datasets contains numbers (pictures or hand-written digits).
We tried different solutions, working with a bottom-up approach: from the simplest case to the most complex. The architecture which showed most promising results was a Shared Classifier: three neural networks are trained at the same time, sharing a classifier.

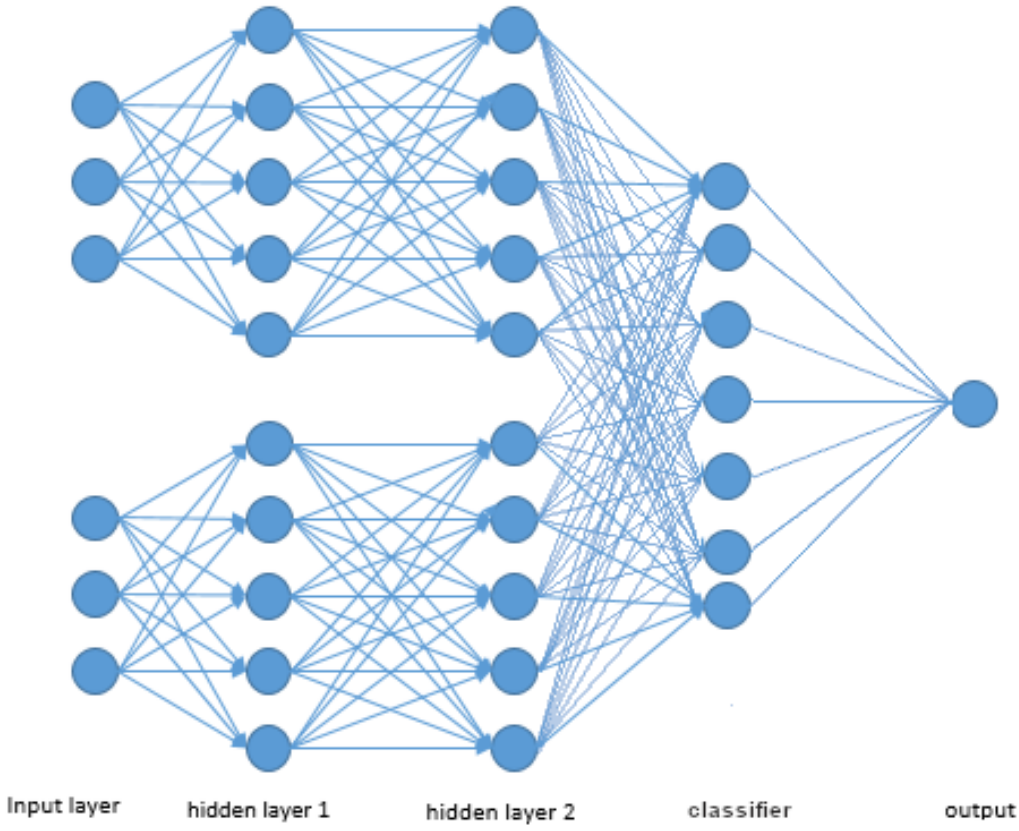


Figure 4.6: Two networks sharing a classifier

This picture shows two neural networks with a shared classifier. This technique can be generalized to an arbitrary number n of networks.
In the experiments have been used three neural networks because we want that during the training they can learn the operation of summing weights. In particular, each neural network is trained on a different dataset (if we have one dataset, we can split in two parts, giving the first part to the first neural network, the second part to the second neural network and the entire dataset to the last neural network). The aim of the training is that the first two networks learn to sum their parameters. At the end of the training, the convolutional weights of the third network will be changed in the sum of the weights of the first two networks. We want to evaluate how the performance of the third network changes between before and after the sum.

In a Federated scenario this experiment is done in two different machines. In a machine we use the previous described architecture of three network on a dataset in order to learn to do the sum of weights. Then in another machine we do the same thing but using the classifier obtained from the first machine and freezing its parameters, and on another dataset. Finally we want to sum the weights of the first neural networks of both machines together with the classifier, and check how are the performances on both the datasets.

Chapter 5

Experiments

The code of all the experiments is available at the following GitHub repository: `https://github.com/CasellaJr/Federated-Transfer-Learning-using-Network-Composition`

All the experiments were done using a VGG16, previously described in VGG16.

Environment of the experiments:

- Python 3.6.9
- Google Colaboratory
- Pytorch

5.1 Python

Python is the most popular programming language for AI and ML. [3]
It is an interpreted high-level general-purpose programming language. Its design philosophy emphasizes code readability with its notable use of significant indentation. Python's language constructs as well as its object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects.
Python is dynamically-typed and garbage-collected. It supports multiple programming paradigms, including structured (particularly, procedural), object-oriented and functional programming. Python is often described as a "batteries included" language due to its comprehensive standard library. [1]

Python has been developed in 1991. A poll suggests that over 57% of developers are more likely to pick Python over C++ as their programming language of choice for developing AI solutions. Being easy-to-learn, Python offers an easier entry into the world of AI development for programmers and data scientists alike.
Some researches on the strong sides of Python and why you should opt in for Python when bringing your AI and ML projects to life have been conducted by DjangoStars [2].

1. **A great library ecosystem**.
   A great choice of libraries is one of the main reasons Python is the most popular programming language used for AI. A library is a module or a group of modules published by different sources like PyPi which include a pre-written piece of code that allows users to reach some functionality or perform different actions. Python libraries provide base level items so developers don't have to code them from the very beginning every time. ML requires continuous data processing, and Python's libraries let you access, handle and transform data. These are some of the most widespread libraries you can use for ML and AI:

   - Scikit-Learn for handling basic ML algorithms like clustering, linear and logistic regressions, regression, classification, and others.
   - Pandas for high-level data structures and analysis. It allows merging and filtering of data, as well as gathering it from other external sources like Excel, for instance.
   - Keras for deep learning. It allows fast calculations and prototyping, as it uses the GPU in addition to the CPU of the computer.
   - TensorFlow for working with deep learning by setting up, training, and utilizing artificial neural networks with massive datasets.
   - PyTorch is an optimized tensor library for deep learning using GPUs and CPUs.

- Matplotlib for creating 2D plots, histograms, charts, and other forms of visualization.
- NLTK for working with computational linguistics, natural language recognition, and processing.
- Scikit-image for image processing.
- PyBrain for neural networks, unsupervised and reinforcement learning.
- Caffe for deep learning that allows switching between the CPU and the GPU and processing 60+ mln images a day using a single NVIDIA K40 GPU.
- StatsModels for statistical algorithms and data exploration.

And more others libraries.

2. **A low entry barrier**.
Working in the ML and AI industry means dealing with a bunch of data that you need to process in the most convenient and effective way. The low entry barrier allows more data scientists to quickly pick up Python and start using it for AI development without wasting too much effort on learning the language. Python programming language resembles the everyday English language, and that makes the process of learning easier. Its simple syntax allows you to comfortably work with complex systems, ensuring clear relations between the system elements. A Python code snippet example:

```
test_number = 407 # our example is not a prime
   number

# prime numbers are greater than 1
if test_number > 1:
    # check for factors
    number_list = range(2, test_number)
    for number in number_list:
        number_of_parts = test_number / number
        print(f"{test_number} is not a prime
           number")
```

```
            print(f"{number} times {number_of_parts}
               is {test_number}")
            break
        else:
            print(f"{test_number} is a prime number")
    else:
        print(f"{test_number} is not a prime number")
```

3. **Flexibility**.
   Python for machine learning is a great choice, as this language is very flexible: It offers an option to choose either to use OOPs or scripting; there's also no need to recompile the source code, developers can implement any changes and quickly see the results; moreover, programmers can combine Python and other languages to reach their goals.

4. **Platform independence**.
   Python is not only comfortable to use and easy to learn but also very versatile. In the sense that Python for machine learning development can run on any platform including Windows, MacOS, Linux, Unix, and twenty-one others. To transfer the process from one platform to another, developers need to implement several small-scale changes and modify some lines of code to create an executable form of code for the chosen platform.

5. **Good visualization options**.
   For AI developers, it's important to highlight that in artificial intelligence, deep learning, and machine learning, it's vital to be able to represent data in a human-readable format. Libraries like Matplotlib allow data scientists to build charts, histograms, and plots for better data comprehension, effective presentation, and visualization. Different application programming interfaces also simplify the visualization process and make it easier to create clear reports.

6. **Community sypport**.

It's always very helpful when there's strong community support built around the programming language. Python is an open-source language which means that there's a bunch of resources open for programmers starting from beginners and ending with pros. A lot of Python documentation is available online as well as in Python communities and forums, where programmers and machine learning developers discuss errors, solve problems, and help each other out. Python programming language is absolutely free as is the variety of useful libraries and tools.

7. **Growing Popularity**.
   As a result of the advantages discussed above, Python is becoming more and more popular among data scientists. This means it's easier to search for developers and replace team players if required. Also, the cost of their work may be not as high as when using a less popular programming language.

5.2 Google Colaboratory

Google Colaboratory is a free **Jupyter notebook** environment that runs entirely in the cloud. A Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more.
Colab does not require a setup and the notebooks that you create can be simultaneously edited by your team members - just the way you edit documents in Google Docs. Colab supports many popular machine learning libraries which can be easily loaded in your notebook.
As a programmer, you can perform the following using Google Colab.

- Write and execute code in Python
- Document your code that supports mathematical equations
- Create/Upload/Share notebooks

- Import/Save notebooks from/to Google Drive
- Import/Publish notebooks from GitHub
- Import external datasets e.g. from Kaggle
- Integrate PyTorch, TensorFlow, Keras, OpenCV
- Free Cloud service with free GPU

To use Colab, all you need is a Google account and a web browser, and you get access to GPUs like Tesla K80 and even a TPU, for free. TPUs are much more expensive than a GPU, and you can use it for free on Colab.
Training models, especially deep learning ones, takes numerous hours on a CPU. We've all faced this issue on our local machines. GPUs and TPUs, on the other hand, can train these models in a matter of minutes or seconds.[4]
Not everyone can afford a GPU because they are expensive. That's where Google Colab comes into play. It gives you a decent GPU for free, which you can continuously run for 12 hours.
Google Colab gives us three types of runtime for our notebooks: CPUs, GPUs, and TPUs. As I mentioned, Colab gives us 12 hours of continuous execution time. After that, the whole virtual machine is cleared and we have to start again. We can run multiple CPU, GPU, and TPU instances simultaneously, but our resources are shared between these instances.
These are the different runtimes offered by Google Colab:

CPU	GPU	TPU
Intel Xeon Processor with two cores @ 2.30 GHz and 13 GB RAM	Up to Tesla K80 with 12 GB of GDDR5 VRAM, Intel Xeon Processor with two cores @ 2.20 GHz and 13 GB RAM	Cloud TPU with 180 teraflops of computation, Intel Xeon Processor with two cores @ 2.30 GHz and 13 GB RAM

Figure 5.1: Runtimes offered by Google Colab

5.3 PyTorch

PyTorch is an open source machine learning library that specializes in *tensor* computations, automatic differentiation, and GPU acceleration. For those reasons, PyTorch is one of the most popular deep learning libraries, competing with both *Keras* and *TensorFlow* for the prize of "most used" deep learning package.
PyTorch tends to be especially popular among the research community due to its Pythonic nature and ease of extendability (i.e., implementing custom layer types, network architectures, etc.). PyTorch is based on Torch, a scientific computing framework for Lua. Prior to both PyTorch and Keras/TensorFlow, deep learning packages such as Caffe and Torch tended to be the most popular.
However, as deep learning started to revolutionize nearly all areas of computer science, developers and researchers wanted an efficient, easy to use library to construct, train, and evaluate neural networks in the Python programming language.
Python, along with R, are the two most popular programming languages for data scientists and machine learning, so it's only natural that researchers wanted deep learning algorithms inside their Python ecosystems.

5.4 First experiment: Direct aggregation function on the weights of 2 Neural Networks

The first experiment consisted in the creation of a Neural Network in which its weights are initialized to the sum of the weights of two other different Neural Networks.
The general idea was to combine the weights of two different neural networks trained on two different datasets, to check how the performances change after the aggregation.
In particular, the first Neural Network was trained on **MNIST**[8]. The MNIST database is a dataset of handwritten digits, with a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been

size-normalized and centered in a fixed-size image.
It is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and formatting.
An example of data of MNIST:

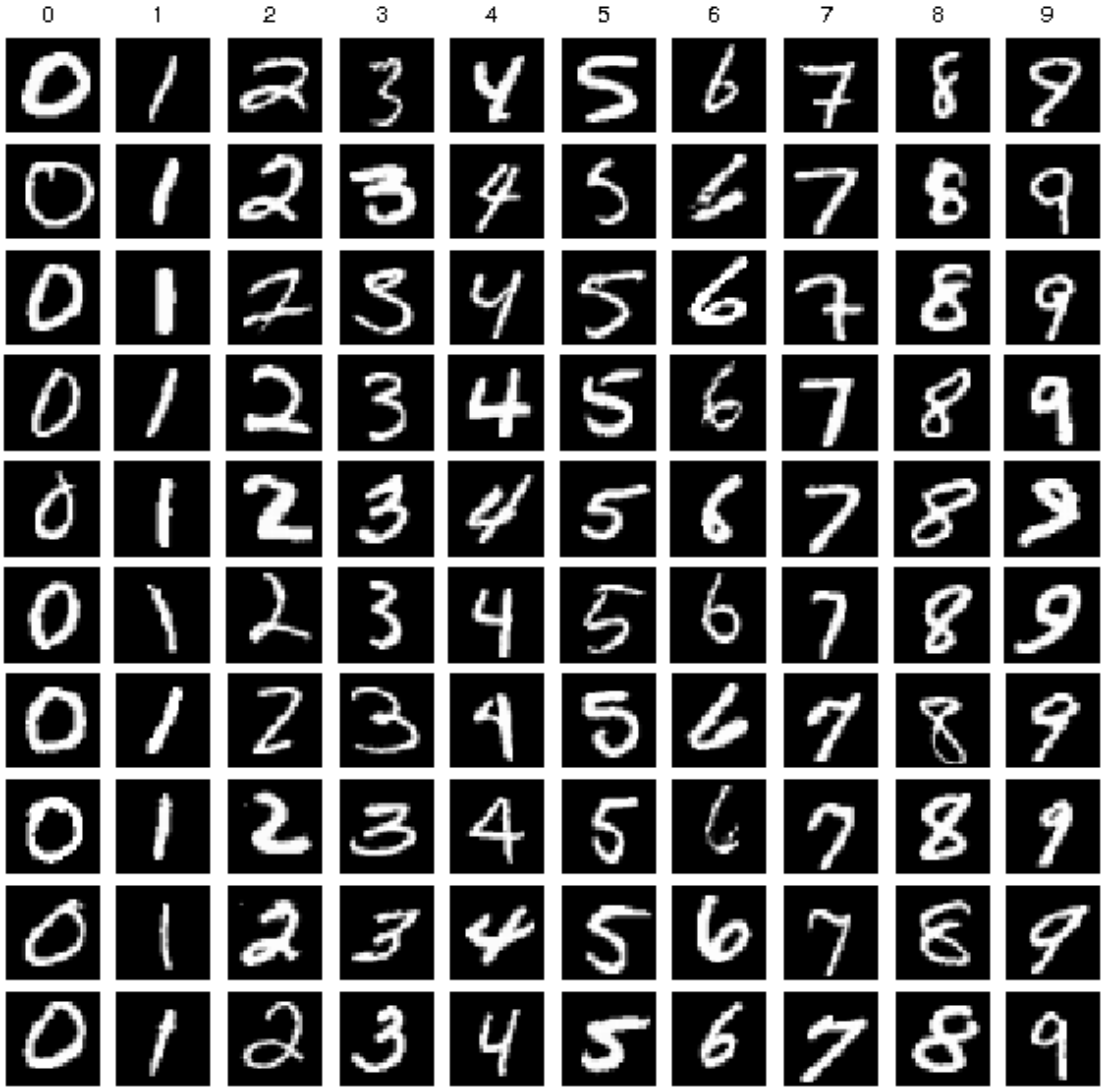


Figure 5.2: An example of data of MNIST

The fraction of the original train set used as validation set was 10%, so 54,000 samples were used as train set and 6000 samples used as validation set.
The images of both train and test set were resized to 32x32, and as *data augmentation*, the only preprocessing done was rotating hori-

zontally the images of only the train set, with 50% chance. Indeed, in the real world scenario, we may have a dataset of images taken in a limited set of conditions. But, our target application may exist in a variety of conditions, such as different orientation, location, scale, brightness etc. We account for these situations by training our neural network with additional synthetically modified data.

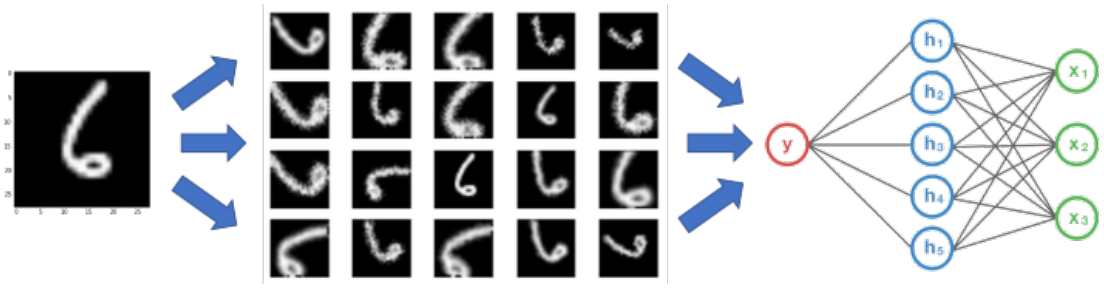


Figure 5.3: Augmentation in play

Augmentation can help even if we have lots of data, like in this case with MNIST; indeed, by performing augmentation, we can prevent neural networks from learning irrelevant patterns, essentially boosting overall performance.
Then, to simplify the problem, labels of both train and test set were rearranged in even and odd, leading to a binary classification task.

```
for i in range(10):
  idx = (train_set.targets==i)
  if (i == 0) or ((i % 2) == 0): train_set.targets[idx]
     = 0

  else: train_set.targets[idx] = 1

for i in range(10):
  idx = (test_set.targets==i)
  if (i == 0) or ((i % 2) == 0): test_set.targets[idx] =
      0

  else: test_set.targets[idx] = 1
```

So, considering the 0 as even, then the odd numbers have label 1 and

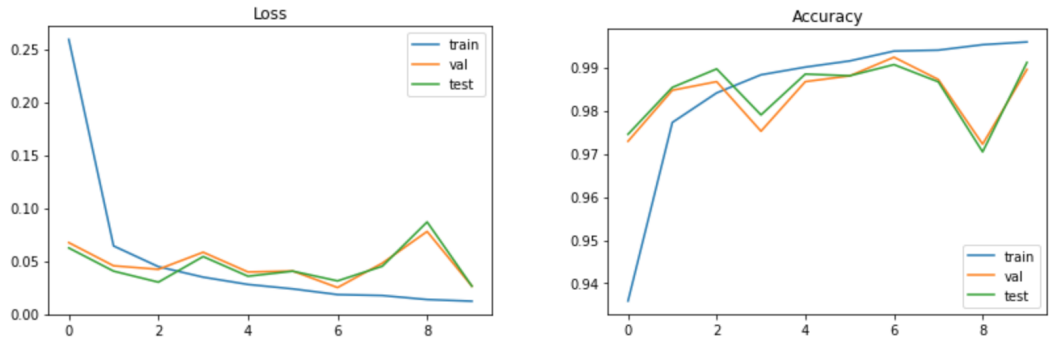
even numbers have label 0.
Some of the hyper-parameters used during the training of this VGG:

- Batch size: 64
- Optimization algorithm: Stochastic Gradient Descent
- Learning rate: 0.01
- Loss function: Binary Cross Entropy with Logits
- Training epochs: 10

These are the results in term of loss and accuracy after the 10 epochs of training:

```
Epoch 1: TrL=0.2592, TrA=0.9359, VL=0.0674, VA=0.9729,
   TeL=0.0624, TeA=0.9745,
Epoch 2: TrL=0.0642, TrA=0.9773, VL=0.0455, VA=0.9847,
   TeL=0.0405, TeA=0.9854,
Epoch 3: TrL=0.0447, TrA=0.9841, VL=0.0422, VA=0.9867,
   TeL=0.0301, TeA=0.9896,
Epoch 4: TrL=0.0348, TrA=0.9883, VL=0.0583, VA=0.9752,
   TeL=0.0542, TeA=0.9790,
Epoch 5: TrL=0.0279, TrA=0.9901, VL=0.0397, VA=0.9867,
   TeL=0.0357, TeA=0.9885,
Epoch 6: TrL=0.0237, TrA=0.9915, VL=0.0406, VA=0.9880,
   TeL=0.0404, TeA=0.9881,
Epoch 7: TrL=0.0183, TrA=0.9938, VL=0.0250, VA=0.9924,
   TeL=0.0312, TeA=0.9906,
Epoch 8: TrL=0.0174, TrA=0.9940, VL=0.0480, VA=0.9872,
   TeL=0.0451, TeA=0.9867,
Epoch 9: TrL=0.0137, TrA=0.9953, VL=0.0779, VA=0.9722,
   TeL=0.0869, TeA=0.9704,
Epoch 10: TrL=0.0121, TrA=0.9959, VL=0.0263, VA=0.9895,
   TeL=0.0263, TeA=0.9911,
```

To choose the best model we consider the validation accuracy, so, the best model is the one at epoch 7, because it has the highest validation accuracy.



(a) Loss of the model

(b) Accuracy of the model

Figure 5.4: A figure showing loss and accuracy of the model

The training of the first neural network is complete. Now, we started the training of the second neural network, which has exactly the same architecture of the first one, and it uses the same hyper-parameters. The only difference is the dataset. For the first model we used MNIST, now we use the Street View House Numbers, **SVHN**[27].

SVHN is a real-world image dataset for developing machine learning and object recognition algorithms with minimal requirement on data preprocessing and formatting. It can be seen as similar in flavor to MNIST (e.g., the images are of small cropped digits), but incorporates an order of magnitude more labeled data (over 600,000 digit images) and comes from a significantly harder, unsolved, real world problem (recognizing digits and numbers in natural scene images). SVHN is obtained from house numbers in Google Street View images.

It has 10 classes, 1 for each digit. Digit '1' has label 1, '9' has label 9 and '0' has label 10. There are 73257 digits for training, 26032 digits for testing, and 531131 additional, somewhat less difficult samples, to use as extra training data.

An example of data of SVHN:

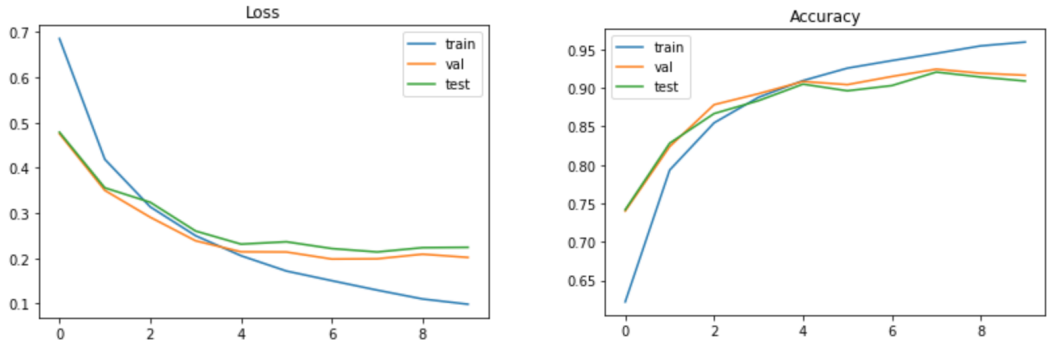
Figure 5.5: An example of data of SVHN

The fraction of the original train set used as validation set was 10%, so 65,932 samples were used as train set and 7325 samples used as validation set.
Images were converted in grayscale, in order to have the same architecture of the first model, so with only one channel as input. Moreover, to have also the same output and the same type of problem scenario, labels were re-arranged in even and odd classes.
These are the results of the training:

```
Epoch 1: TrL=0.6857, TrA=0.6220, VL=0.4756, VA=0.7401,
   TeL=0.4789, TeA=0.7418,
```

```
Epoch 2: TrL=0.4189, TrA=0.7933, VL=0.3503, VA=0.8238,
   TeL=0.3560, TeA=0.8282,
Epoch 3: TrL=0.3139, TrA=0.8547, VL=0.2910, VA=0.8783,
   TeL=0.3240, TeA=0.8667,
Epoch 4: TrL=0.2501, TrA=0.8879, VL=0.2389, VA=0.8926,
   TeL=0.2605, TeA=0.8835,
Epoch 5: TrL=0.2060, TrA=0.9096, VL=0.2146, VA=0.9085,
   TeL=0.2315, TeA=0.9050,
Epoch 6: TrL=0.1721, TrA=0.9258, VL=0.2141, VA=0.9044,
   TeL=0.2367, TeA=0.8963,
Epoch 7: TrL=0.1507, TrA=0.9356, VL=0.1986, VA=0.9150,
   TeL=0.2217, TeA=0.9031,
Epoch 8: TrL=0.1299, TrA=0.9450, VL=0.1991, VA=0.9245,
   TeL=0.2142, TeA=0.9207,
Epoch 9: TrL=0.1103, TrA=0.9548, VL=0.2093, VA=0.9192,
   TeL=0.2237, TeA=0.9143,
Epoch 10: TrL=0.0988, TrA=0.9597, VL=0.2021, VA=0.9166,
   TeL=0.2244, TeA=0.9091,
```

The best model is the one at epoch 8, because it has the highest validation accuracy.



(a) Loss of the model

(b) Accuracy of the model

Figure 5.6: A figure showing loss and accuracy of the model

After the training on MNIST and SVHN of these two models, it was time to create some aggregated model.
We started with a network initialized with the sum of the weights of the previously trained models. To do this, we passed the already

trained model as arguments of the *__init__* initialized of the new net, and just before the forward the weights are initialized:

```
class VGG16SUM(nn.Module):

    def __init__(self, model1, model2, num_classes):
        super(VGG16SUM, self).__init__()

        ...

        for p_out, p_in1, p_in2 in zip(self.parameters(),
            model1.parameters(), model2.parameters()):
            p_out.data = nn.Parameter(p_in1 +p_in2);

    def forward(self, x):

        x = self.block_1(x)
        ...
```

Just to check that the initialization was successful, here are shown the first weights of the models.
These are the first weights of the model trained on MNIST:

```
[Parameter containing:
 tensor([[[[-7.6350e-01, 5.6528e-01, -2.3996e-01],
           [-6.7057e-01, -3.1837e-01, 1.6944e-02],
           [ 4.5301e-01, 7.7138e-01, 6.0862e-01]]],
```

These are the first weights of the model trained on SVHN:

```
[Parameter containing:
 tensor([[[[ 0.5388, 0.6938, 0.3826],
           [-0.7382, 0.6381, 0.4537],
           [-0.2710, -0.5323, -0.7753]]],
```

These are the first weights of the model initialized to the sum of the previous two:

```
[Parameter containing:
 tensor([[[[-2.2472e-01, 1.2590e+00, 1.4266e-01],
```

```
          [-1.4088e+00, 3.1971e-01, 4.7067e-01],
          [ 1.8202e-01, 2.3908e-01, -1.6669e-01]]],
```

As we can see, the initialization was successful: the weights of the last model are equal to the sum of the first two models.
At this point we evaluated this aggregated network on both the two datasets, MNIST and SVHN:

Model SUM	Test Accuracy
MNIST	49.426%
SVHN	46.30%

Table 5.1: Test performance of the aggregated (SUM) model.

It is possible to notice a sharp drop on the performances. Accuracies are practically halved: on MNIST, from about 99% to 49.426%, and on SVHN, from about 92% to 46.30%. Considering the fact that we are treating a binary classification problem, with only even and odd classes, these results of the aggregated model are very bad. It seems that this model is like flipping a coin.
We tried also with a different type of aggregation function, the Max operator:

```
for p_out, p_in1, p_in2 in zip(self.parameters(), model1
   .parameters(), model2.parameters()):
           p_out.data = nn.Parameter(torch.max(p_in1,
              p_in2));
```

Again, to see that initialization was successful, here are shown the first weights of this new aggregated model:

```
[Parameter containing:
 tensor([[[[ 5.3879e-01, 6.9375e-01, 3.8262e-01],
          [-6.7057e-01, 6.3808e-01, 4.5373e-01],
          [ 4.5301e-01, 7.7138e-01, 6.0862e-01]]],
```

We evaluated this network on both the two datasets, MNIST and SVHN:

Model MAX	Test Accuracy
MNIST	50.74%
SVHN	53.70%

Table 5.2: Test performance of the aggregated (MAX) model.

It is possible to notice a very slight increase with respect to the model aggregated with the sum function. However, results are still poor.
In order to achieve better results we tried to train only the classifier of our model, keeping constant the convolutional parameters. We started considering the model initialized with the sum of weights.

```
# First, we set require_grad = False for all the layer
   of the net
for param in modelsum_classifier.parameters():
  param.requires_grad = False
# Then, we replace the classification layers with a new
    final fully connectect layer
modelsum_classifier.classifier[0] = torch.nn.Linear(
   in_features=2048, out_features = 4096)
modelsum_classifier.classifier[3] = torch.nn.Linear(
   in_features=4096, out_features = 4096)
modelsum_classifier.classifier[6] = torch.nn.Linear(
   in_features=4096, out_features = 1)
```

We specified modelsum_classifier.classifier[0], modelsum_classifier.classifier[3] and modelsum_classifier.classifier[6] because these are the Linear layers, while the others, so 1,2,4 and 5, are the activation function ReLU and the Dropout, which have not parameters.
A quick training on MNIST, for only two epochs, showed no improvements.

```
Epoch 1: TrL=0.7019, TrA=0.4911, VL=0.6959, VA=0.4888,
   TeL=0.7016, TeA=0.4926,
Epoch 2: TrL=0.7023, TrA=0.4906, VL=0.6964, VA=0.4888,
   TeL=0.7018, TeA=0.4926,
```

The same training was repeated also on SVHN, and for the model with the MAX aggregation function, and also in these cases there were no improvements.
Some thoughts on why this technique did not work: the issue is that there is a lot of arbitrariness and redundancy in the weights of a neural network model. If I take two identical models, but give them different (but equivalent) initializations, and train them on the same training data (but probably batched up into different (but equivalent) random batches), there is no reason for "weight-17" in model A to have the same value as "weight-17" in model B. Moreover, in this experiment we trained on two different training sets.
Training model A takes it down some particular path along the "loss-surface" to some reasonable location in weight-configuration space. But because of the redundancy and randomness, training model B takes it down some entirely different (but comparably worthwhile) path to some entirely different location (that is comparably good in making predictions).
Speaking somewhat figuratively, let's say that, by happenstance, the weights on the "left-hand" side of model A learn to recognize straight edges, while weights on the "right-hand" side learn curved edges. But now assume that, by happenstance, this is reversed in model B, so the left side learns curved edges and the right, straight edges.
So weight-17 in model A means straight, but weight-17 in model B means curved. These two weights mean two different things, so it does not make sense to add, or average, or otherwise combine then together.
However, what if, by (extremely unlikely) happenstance, model A and model B, while being trained, did end up following similar paths, and ended up at similar locations in weight-configuration space. Now it could make sense to combine the two weight-17s together, as they could now have similar meaning.
One approach could be to have the two models guide one another along similar paths while training. The idea would be that you could start the two models at the same (random) location – i.e., use the same random initialization for weights of both models – and then add a loss term that nudges the two models to prefer similar paths.

By virtue of all of these considerations just made, and in particular of this last idea, in the next section we will discuss of a different technique.

5.5 Second experiment: Multi-Input Neural Network

The second experiment consisted in the creation of a Multi-Input Neural Network in which the inputs are two VGG with the same architecture of the first experiment.

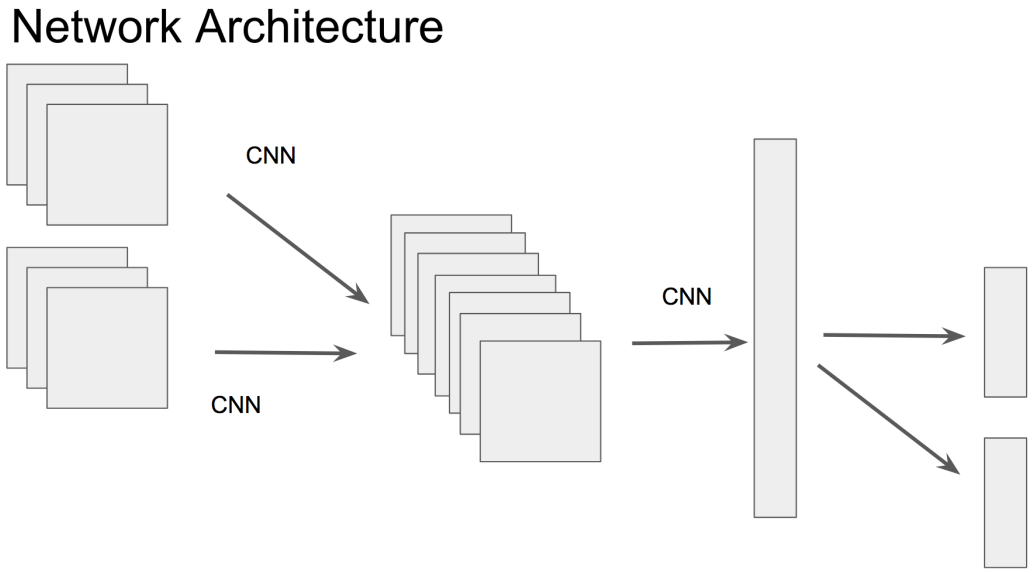


Figure 5.7: A Multi-Input Neural Network

We want that our Multi-Input neural network learns to do the sum of the weights of the models in input.
In a theoretical framework we want to use two different datasets, like MNIST and SVHN, and a support dataset, like *CIFAR-10* [28]. We want that performance (accuracy) on MNIST, before and after the sum, to be as close as possible. We use CIFAR-10 only to try to learn the operation of sum of models. We are interested in the weights of MNIST, we can call θ_{MNIST}, and we hope that they have learnt the operation of sum.

A possible way to do this is the following: have 3 NNs, with the same architecture, but with different weights, the first one for MNIST, second for CIFAR10, and the third one that have weights (for each layer) of $\theta_{MNIST} + \theta_{CIFAR10}$. The loss function for this network can be CrossEntropy(MNIST) + CrossEntropy(CIFAR10) + CrossEntropy([MNIST+CIFAR10] - MNIST) (this last term of loss should guarantee that the network on MNIST have similar performances before and after the operation of sum.
Then, we want to do the same thing of before, but replacing MNIST with SVHN, and again CIFAR10 as support dataset.
Finally, we can build a NN with weights $\theta_{MNIST} + \theta_{SVHN}$ and look at the performances.
However, for simplicity, we used only MNIST, split in two parts. We considered, as in the first experiment, a case of only even and odd classes.
We tried with a custom loss defined as follows:

```
class CombinedLoss(nn.Module):
    def __init__(self, loss_a, loss_b, loss_combo,
       _lambda=1.0):
        super().__init__()
        self.loss_a = loss_a
        self.loss_b = loss_b
        self.loss_combo = loss_combo

        self.register_buffer('_lambda',torch.tensor(float
           (_lambda),dtype=torch.float32))


    def forward(self,y_hat,y):

        return self.loss_a(y_hat[0],y[0]) + self.loss_b(
           y_hat[1],y[1]) + self._lambda * self.
           loss_combo(y_hat[2],torch.cat(y,0))
```

The parameter *lambda* acts as a regularizer. During the training we will try different values of *lambda* to explore the way in which

performances change.
Several hyper-parameters were used during training. Considering the same settings of the first experiment except for a batch size of 128, these are the results of the training after 5 epochs:

```
Epoch 1: Training Loss for combo = 0.3391,
Epoch 1: Training Accuracy for A = 0.9002,
Epoch 1: Training Accuracy for B = 0.9080,
Epoch 1: Training Accuracy for combo = 0.8975,
Epoch 1: Val Loss for combo = 13.0720,
Epoch 1: Val Accuracy for A = 0.9025,
Epoch 1: Val Accuracy for B = 0.9569,
Epoch 1: Val Accuracy for combo = 0.4877,
Epoch 1: Test Loss for combo = 12.9154,
Epoch 1: Test Accuracy for A = 0.9145,
Epoch 1: Test Accuracy for B = 0.9383,
Epoch 1: Test Accuracy for combo = 0.4942,


Epoch 2: Training Loss for combo = 0.1154,
Epoch 2: Training Accuracy for A = 0.9741,
Epoch 2: Training Accuracy for B = 0.9756,
Epoch 2: Training Accuracy for combo = 0.9701,
Epoch 2: Val Loss for combo = 10.3165,
Epoch 2: Val Accuracy for A = 0.9308,
Epoch 2: Val Accuracy for B = 0.9440,
Epoch 2: Val Accuracy for combo = 0.4877,
Epoch 2: Test Loss for combo = 10.2091,
Epoch 2: Test Accuracy for A = 0.9158,
Epoch 2: Test Accuracy for B = 0.9357,
Epoch 2: Test Accuracy for combo = 0.4942,


Epoch 3: Training Loss for combo = 0.0821,
Epoch 3: Training Accuracy for A = 0.9825,
Epoch 3: Training Accuracy for B = 0.9844,
Epoch 3: Training Accuracy for combo = 0.9780,
```

```
Epoch 3: Val Loss for combo = 8.2678,
Epoch 3: Val Accuracy for A = 0.9621,
Epoch 3: Val Accuracy for B = 0.9772,
Epoch 3: Val Accuracy for combo = 0.4877,
Epoch 3: Test Loss for combo = 8.1472,
Epoch 3: Test Accuracy for A = 0.9736,
Epoch 3: Test Accuracy for B = 0.9691,
Epoch 3: Test Accuracy for combo = 0.4942,


Epoch 4: Training Loss for combo = 0.0622,
Epoch 4: Training Accuracy for A = 0.9871,
Epoch 4: Training Accuracy for B = 0.9875,
Epoch 4: Training Accuracy for combo = 0.9830,
Epoch 4: Val Loss for combo = 2.4261,
Epoch 4: Val Accuracy for A = 0.9551,
Epoch 4: Val Accuracy for B = 0.9460,
Epoch 4: Val Accuracy for combo = 0.4877,
Epoch 4: Test Loss for combo = 2.3723,
Epoch 4: Test Accuracy for A = 0.9691,
Epoch 4: Test Accuracy for B = 0.9426,
Epoch 4: Test Accuracy for combo = 0.4942,


Epoch 5: Training Loss for combo = 0.0498,
Epoch 5: Training Accuracy for A = 0.9895,
Epoch 5: Training Accuracy for B = 0.9904,
Epoch 5: Training Accuracy for combo = 0.9865,
Epoch 5: Val Loss for combo = 2.8591,
Epoch 5: Val Accuracy for A = 0.9873,
Epoch 5: Val Accuracy for B = 0.9811,
Epoch 5: Val Accuracy for combo = 0.5123,
Epoch 5: Test Loss for combo = 2.8956,
Epoch 5: Test Accuracy for A = 0.9912,
Epoch 5: Test Accuracy for B = 0.9777,
Epoch 5: Test Accuracy for combo = 0.5058
```

We are interested in the test accuracy for combo. However, the value taken by this accuracy remains more or less the same over epochs. This happens also using different hyper-parameters and aggregation function. Indeed, we tried with several combinations of hyper-parameters, playing with the value of learning rate (0.01, 0.001, 0.0001) and with the value of the lambda regularizer (1, 0.5 and 0.7), using Adam as optimization algorithm, changing the aggregation function with the Maximum or with the Average, but accuracy never rises over 51%, which, in a binary classification problem, is like flipping a coin.
Some problems which can cause these bad outputs may be addressed to drawbacks similar to the first experiment, i.e. a too sharp aggregation of weights.
Considering these results, to achieve better performances with our proposed method, we need to try other techniques. In the next subsection we are going to see our third experiment, which relies on three networks with a shared classifier.

5.6 Third experiment: Shared Classifier

The third experiment consisted in the creation of three different VGG16 architectures with a shared classifier.
The aim of this technique is to soften the way in which the weights are aggregated with respect to the first two experiments.
We have 3 different VGGs, called *VGGA, VGGB and VGG**, which share a classifier. So, each architecture has its own convolution and weights but they have a common classifier. We control the performances of the network during training. In the evaluation phase, the weights of *VGG** will be equal to the sum of the weights of *VGGA and VGGB*, and we can compare the results before and after the sum.
As in the previous experiment, in a theoretical way we have to use two different datasets, MNIST and SVHN. However, for simplicity, we will use only MNIST divided in 2 parts: one subset with the first

5 digits and another subset with the last 5 digits. If this model will work, we will try to extend to the theoretical case with two different datasets. For simpliticy we started again by considering only even and odd classes., so the classifier will have only one neuron as output:

```
model1 = VGG16((1,32,32),batch_norm=True)
model2 = VGG16((1,32,32),batch_norm=True)
model3 = VGG16((1,32,32),batch_norm=True)
classifier = Classifier(num_classes=1)
```

where the classifier is

```
class Classifier(nn.Module):

    def __init__(self,num_classes=1):
        super().__init__()

        self.classifier = nn.Sequential(
            nn.Linear(2048, 2048),
            nn.ReLU(True),
            nn.Dropout(p=0.5),
            nn.Linear(2048, 512),
            nn.ReLU(True),
            nn.Dropout(p=0.5),
            nn.Linear(512, num_classes)
        )

    def forward(self,x):

        return self.classifier(x)
```

and VGG16 is

```
class VGG16(nn.Module):

  def __init__(self, input_size, batch_norm=False):
    super(VGG16, self).__init__()

    self.in_channels,self.in_width,self.in_height =
```

```
      input_size

    self.block_1 = VGGBlock(self.in_channels,64,
       batch_norm=batch_norm)
    self.block_2 = VGGBlock(64, 128,batch_norm=
       batch_norm)
    self.block_3 = VGGBlock(128, 256,batch_norm=
       batch_norm)
    self.block_4 = VGGBlock(256,512,batch_norm=
       batch_norm)


  @property
  def input_size(self):
      return self.in_channels,self.in_width,self.
         in_height

  def forward(self, x):

    x = self.block_1(x)
    x = self.block_2(x)
    x = self.block_3(x)
    x = self.block_4(x)
    # x = self.avgpool(x)
    x = torch.flatten(x,1)

    return x
```

where VGGBlock is defined as follows:

```
class VGGBlock(nn.Module):
    def __init__(self, in_channels, out_channels,
       batch_norm=False):

        super().__init__()

        conv2_params = {'kernel_size': (3, 3),
```

```
                    'stride' : (1, 1),
                    'padding' : 1
                    }

    noop = lambda x : x

    self._batch_norm = batch_norm

    self.conv1 = nn.Conv2d(in_channels=in_channels,
       out_channels=out_channels , **conv2_params)
    self.bn1 = nn.BatchNorm2d(out_channels) if
       batch_norm else noop

    self.conv2 = nn.Conv2d(in_channels=out_channels,
       out_channels=out_channels, **conv2_params)
    self.bn2 = nn.BatchNorm2d(out_channels) if
       batch_norm else noop


    self.max_pooling = nn.MaxPool2d(kernel_size=(2,
       2), stride=(2, 2))

@property
def batch_norm(self):
    return self._batch_norm

def forward(self,x):
    x = self.conv1(x)
    x = self.bn1(x)
    x = F.relu(x)

    x = self.conv2(x)
    x = self.bn2(x)
    x = F.relu(x)

    x = self.max_pooling(x)
```

```
        return x
```

Our loss function is a combination of the loss function of each VGG16 (the cross entropy, or the binary cross entropy) with the addition of a regularizer:

$$L_T = L_{CE}^A + L_{CE}^B + L_{CE}^* + \lambda \sum_i (w_i^* - [w_i^A + w_i^B]^2) \tag{5.1}$$

which is implemented as follows:

```
class CombinedLoss(nn.Module):
    def __init__(self, loss_a, loss_b, loss_combo,
       _lambda=1.0):
        super().__init__()
        self.loss_a = loss_a
        self.loss_b = loss_b
        self.loss_combo = loss_combo

        self.register_buffer('_lambda',torch.tensor(float
           (_lambda),dtype=torch.float32))


    def forward(self,y_hat,y):

        return self.loss_a(y_hat[0],y[0]) + self.loss_b(
           y_hat[1],y[1]) + self._lambda * self.
           loss_combo(y_hat[2],torch.cat(y,0))
```

We can assume $\lambda = \frac{1}{\#layers}$, however, for simplicity, during our training we will try with simple values like 0.1, 1 and 10.
We will use SGD as optimizer, a learning rate of 0.01 and a batch size of 128. VGGA receives the first five digits, VGGB the last five digits and VGG* the entire dataset.
For each epoch we evaluate the last network, VGG*, changing its parameters to the sum of VGGA and VGGB, as follows:

```
summed_state_dict = OrderedDict()
```

```
for key in nets[2].state_dict():
  if key.find('conv') >=0:
    #print(key)
    summed_state_dict[key] = combo_fn(nets[0].state_dict
       ()[key],nets[1].state_dict()[key])
  else:
    summed_state_dict[key] = nets[2].state_dict()[key]

nets[2].load_state_dict(summed_state_dict)
accuracy_star = test(nets[2], nets[3], test_loader_all)
print(f"Accuracy test VGGSTAR: {accuracy_star:0.5}")
history_test.append(accuracy_star)
```

where with VGGSTAR we mean VGG* after the sum of weights, to distinguish them.
Also in this case results were very bad, again like flipping a coin, with accuracies around 50%. In the first epochs, it seems that the accuracy of VGGSTAR is oscillating however, very soon it stabilizes around this 0.50742 without improvements:

```
Epoch 1:
       train Loss: 0.87544 VGG 1:0.89717 VGG 2:0.90123
          VGG *:0.90705
       val Loss: 5.7042 VGG 1:0.63969 VGG 2:0.49991 VGG
          *:0.86472
       test Loss: 6.0837 VGG 1:0.59746 VGG 2:0.49707 VGG
           *:0.86553
Accuracy test VGGA: 0.64102
Accuracy test VGGB: 0.49268
Accuracy test VGG*: 0.86551
Accuracy test VGGSTAR: 0.49268

 [Elapsed Time: 0:03:04] |################| [Time:
   0:03:04] [Train Acc: 0.98]

Epoch 2:
```

```
       train Loss: 0.29544 VGG 1:0.97392 VGG 2:0.9734
          VGG *:0.94537
       val Loss: 0.21989 VGG 1:0.96573 VGG 2:0.982 VGG
          *:0.97419
       test Loss: 0.20713 VGG 1:0.95898 VGG 2:0.98672
          VGG *:0.97852
Accuracy test VGGA: 0.97152
Accuracy test VGGB: 0.98101
Accuracy test VGG*: 0.97824
Accuracy test VGGSTAR: 0.66742

 [Elapsed Time: 0:03:03] |################| [Time:
    0:03:03] [Train Acc: 0.957]

Epoch 3:
       train Loss: 0.23154 VGG 1:0.98207 VGG 2:0.98196
          VGG *:0.95394
       val Loss: 0.50696 VGG 1:0.98698 VGG 2:0.98558 VGG
           *:0.86482
       test Loss: 0.54623 VGG 1:0.97949 VGG 2:0.99199
          VGG *:0.84844
Accuracy test VGGA: 0.98507
Accuracy test VGGB: 0.98863
Accuracy test VGG*: 0.8482
Accuracy test VGGSTAR: 0.50742

.
.
.

Epoch 49:
       train Loss: 0.082561 VGG 1:0.99993 VGG 2:1.0 VGG
          *:0.97044
       val Loss: 0.13685 VGG 1:0.99154 VGG 2:0.99186 VGG
           *:0.97961
       test Loss: 0.14566 VGG 1:0.99004 VGG 2:0.99434
```

```
          VGG *:0.98018
Accuracy test VGGA: 0.9913
Accuracy test VGGB: 0.99199
Accuracy test VGG*: 0.97992
Accuracy test VGGSTAR: 0.50801

 [Elapsed Time: 0:03:03] |###############| [Time:
    0:03:03] [Train Acc: 0.984]

Epoch 50:
       train Loss: 0.083482 VGG 1:1.0 VGG 2:1.0 VGG
          *:0.97052
       val Loss: 0.13456 VGG 1:0.99219 VGG 2:0.99316 VGG
           *:0.97793
       test Loss: 0.14935 VGG 1:0.99102 VGG 2:0.99434
          VGG *:0.97578
Accuracy test VGGA: 0.99229
Accuracy test VGGB: 0.99189
Accuracy test VGG*: 0.97844
Accuracy test VGGSTAR: 0.50732
```

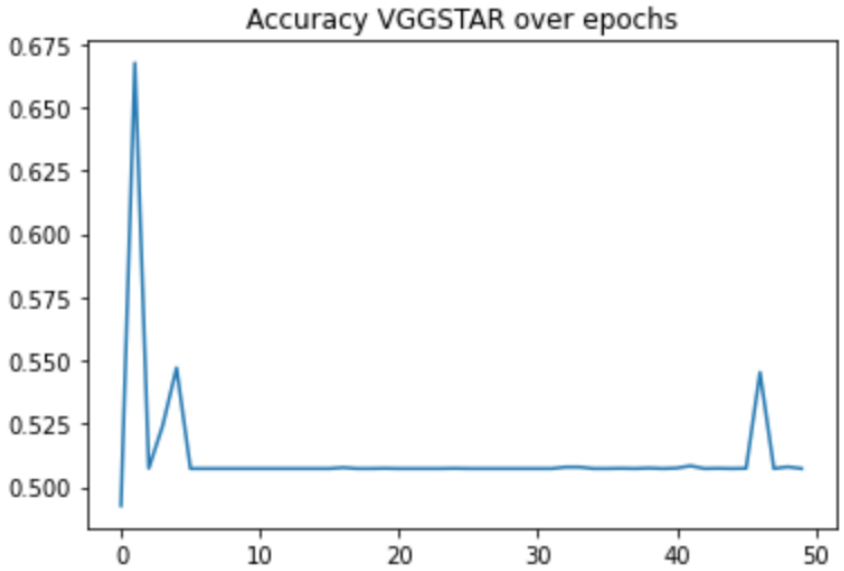


Figure 5.8: Accuracy of VGGSTAR over 50 epochs

These bad results can be attributed to the curse of batch normalization. Indeed, there are many situations under which batch normalization starts to hurt performance or does not work at all:

- **Unstable when using small batch sizes**: the batch normalization layer has to calculate mean and variance to normalize the previous outputs across the batch. This statistical estimation will be pretty accurate if the batch size is fairly large while keeps on decreasing as the batch size decreases. This is a problem when we cannot increase the dimension of the batch size, due to constrained resources, or in situations like fine-tuning.
- **Leads to Increased Training Time**: as a result of experiments conducted by NVIDIA and Carnegie Mellon University, they claim that "even though Batch Normalization is not the computationally intensive and total number of iterations needed for convergence are decreased. The per-iteration time could be noticeably increased.", and it can further be increased with an increase in batch size. The reason is that because batch norm requires double iteration through input data, one for computing batch statistics and another for normalizing the output.

There are several alternatives to batch normalization. One of the most famous is *Group Normalization*[29]. Group Normalization (GN) divides channels into groups and normalizes the features within each group. GN is independent of batch sizes and it does not exploit the batch dimension, like how batch normalization does. GN stays stable over a wide range of batch sizes.
In order to implement GN, we have to change *self.bn1* and *self.bn2* of VGGBlock, specifying that we want to use GN:

```
self.conv1 = nn.Conv2d(in_channels=in_channels,
   out_channels=out_channels , **conv2_params)
        self.bn1 = nn.GroupNorm(16, out_channels) if
           batch_norm else noop

        self.conv2 = nn.Conv2d(in_channels=out_channels,
           out_channels=out_channels, **conv2_params)
```

```
self.bn2 = nn.GroupNorm(16, out_channels) if
   batch_norm else noop
```

where 16 is the number of groups. We tried different number of groups, 2, 4, 8, 16, 32 and 64. Results were very similar among them, except for the cases of 2 and 4 groups where performances were bad.
So, repeating the training with GN:

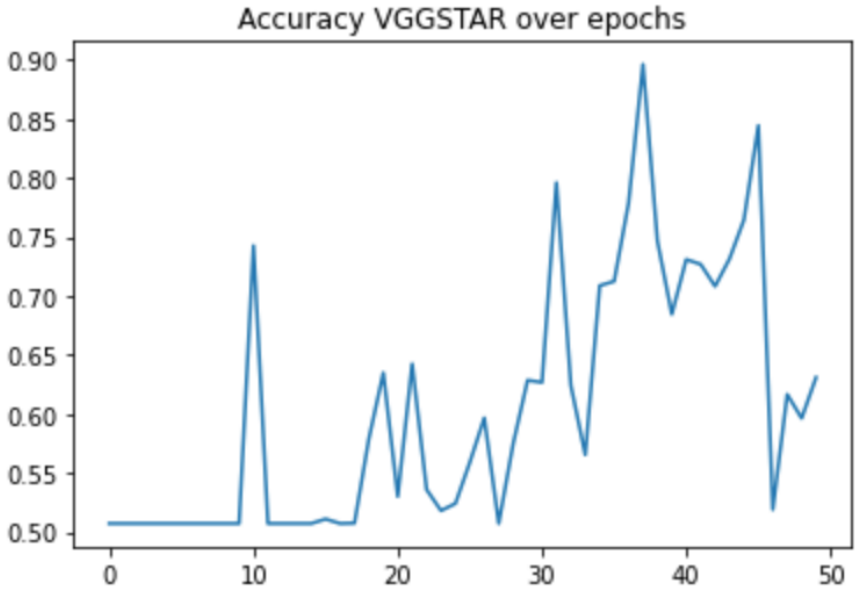


Figure 5.9: Accuracy of VGGSTAR on even/odd classes of MNIST over 50 epochs

We can see a great improvement: accuracy can reach around 90%, and mostly, accuracy does not remain constant as in the previous case, meaning that it is working.
At this point good results have been obtained using GN and only two classes. It is now possible to return to the original case of classification on MNIST with 10 digits. In this case results are stable:

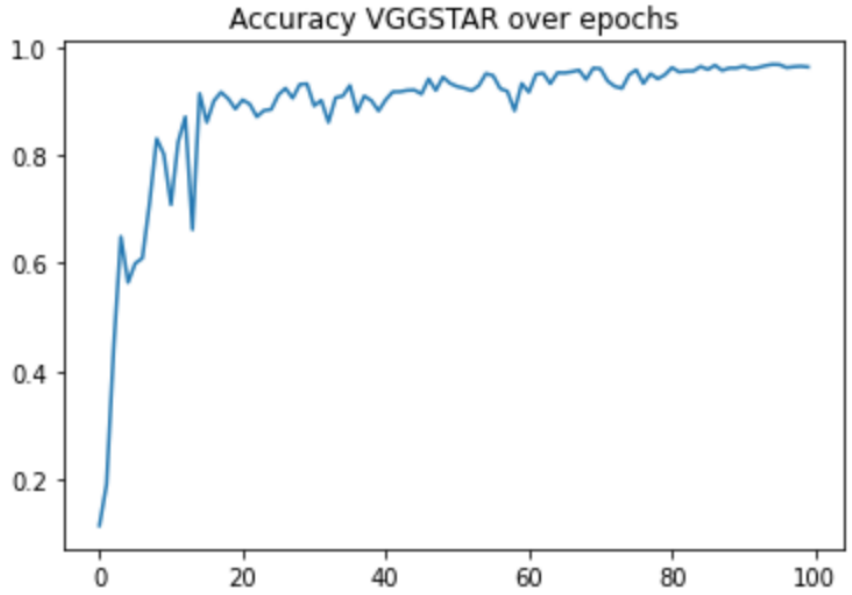


Figure 5.10: Accuracy of VGGSTAR on MNIST over 100 epochs

After around 20 epoch accuracy reaches 90% and then slightly increases.
This good performance suggests us that we can finally try with the theoretical idea of using two different datasets, respectively MNIST and SVHN to train VGGA and VGGB, both to train VGG*, and both to test the networks:

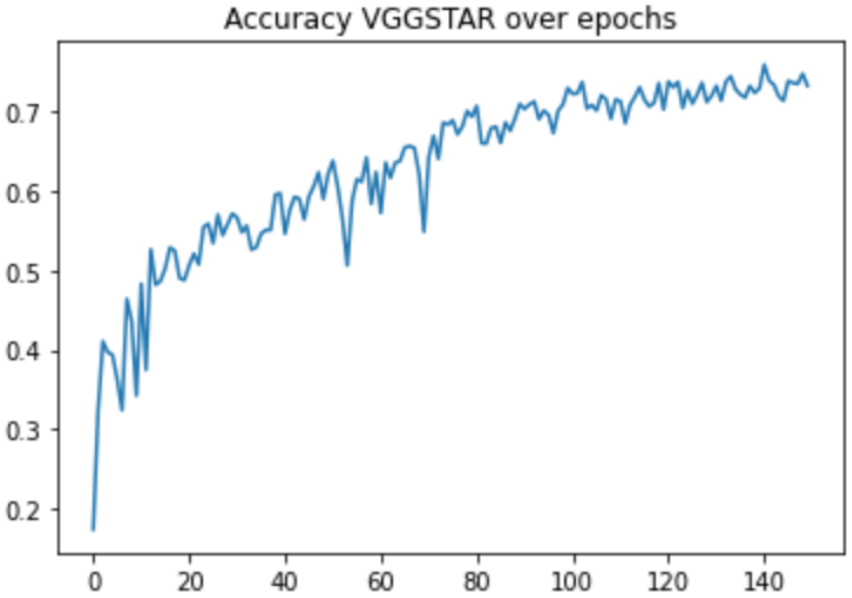


Figure 5.11: Accuracy of VGGSTAR on MNIST and SVHN over 100 epochs

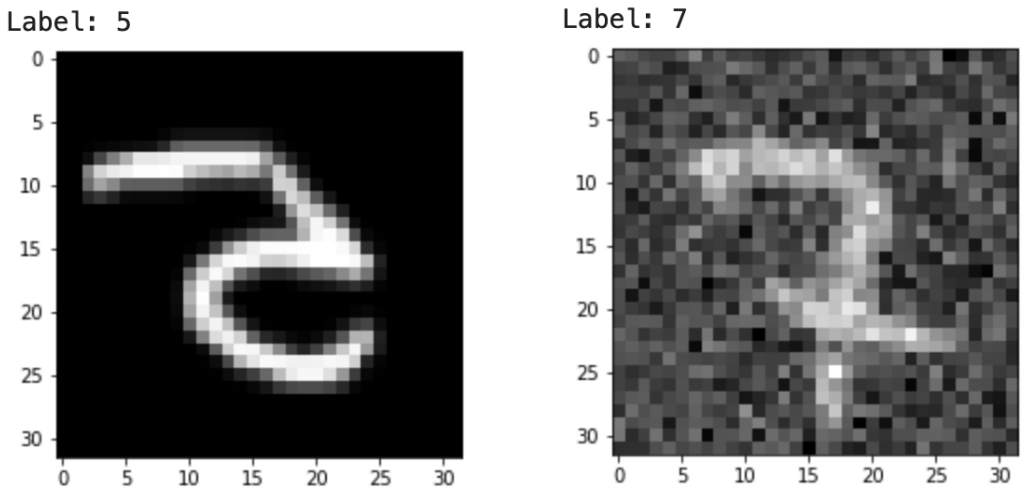
The picture is showing a steady increase of accuracy of VGGSTAR until around 75%.
It is interesting to note that the accuracy test of VGGA after 150 epochs is lower than the accuracy of VGGSTAR, which in turn is lower than VGGB:

```
Epoch 150:
        train Loss: 0.29859 VGG 1:0.99983 VGG 2:0.99929
           VGG *:0.90537
        val Loss: 0.79291 VGG 1:0.99102 VGG 2:0.93201 VGG
            *:0.91795
        test Loss: 0.78559 VGG 1:0.9909 VGG 2:0.93246 VGG
            *:0.91489
Accuracy test VGGA: 0.55228
Accuracy test VGGB: 0.78194
Accuracy test VGG*: 0.88533
Accuracy test VGGSTAR: 0.73302
```

This happens because VGGA is trained only on MNIST, VGGB only on SVHN, but they are tested on both MNIST and SVHN. VGGA performs worst than VGGB because the samples test of MNIST are lower than the samples test of SVHN.

Until now we have worked in a environment that is not Federated, because the nets of the last experiment are on the same notebook, and in general they are in the same machine. We want to combine weights that are obtained from nets on different machines and with a separate training for each dataset.
We have worked with MNIST in VGGA and SVHN in VGGB. To achieve the Federated scenario we want to isolate these 2 datasets. So, in one machine we have to do the training with MNIST and in another machine with SVHN. However, we need always two datasets during the training in order to let the net learn the operation of sum of weights. The idea was to use the original dataset for VGGA, a noisy or augmentated dataset for VGGB and both for VGG*. In one machine we trained our architecture using MNIST and a noisy MNIST, and in another machine we used SVHN and a noisy SVHN.

To perturb the dataset we added a Gaussian Noise with mean 0 and standard deviation 0.2.



(a) Example of a original hand-written number 5

(b) Example of a noisy hand-written number 7

Figure 5.12: A figure showing an example of original and noisy MNIST

And here an example of a noisy SVHN:

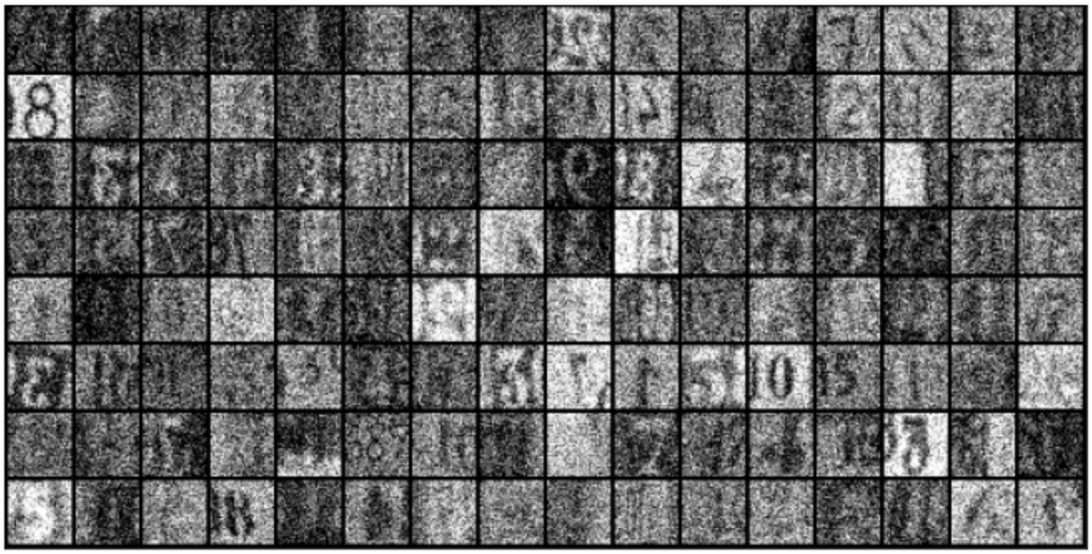


Figure 5.13: A batch example of noisy SVHN

So, we did the training of 2 architectures: the first architecture has MNIST for VGGA, noisy MNIST for VGGB and both for VGG*, while the second architecture has SVHN for VGAA; noisy SVHN for

VGGB and both for VGG*. We have to share the same classifier among the architectures, so we proceeded in this way: we trained the first architecture and we saved the weights of the classifier. Then we loaded the weights of this classifier on the second architecture and we froze them in order to do not change them during back-propagation. In this case the classifier is trained only on MNIST and brutally applied on SVHN. We will show also the opposite case, when the classifier is trained on SVHN and applied on MNIST.
In the first experiment with MNIST and noisy MNIST, these are the results:

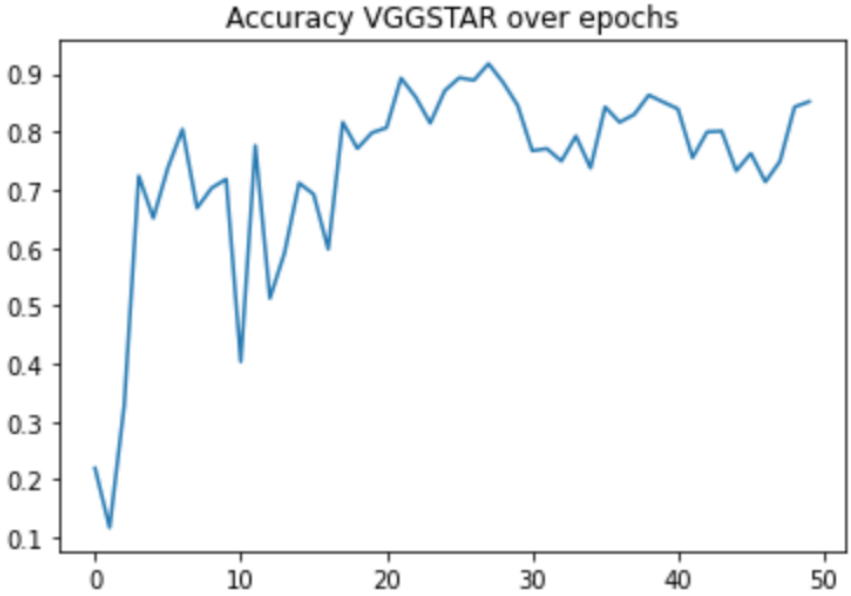


Figure 5.14: Accuracy of VGGSTAR in a FL scenario with training on MNIST

Accuracy reached a value of 92%.
In the second case with SVHN and noisy SVHN, these are the results:

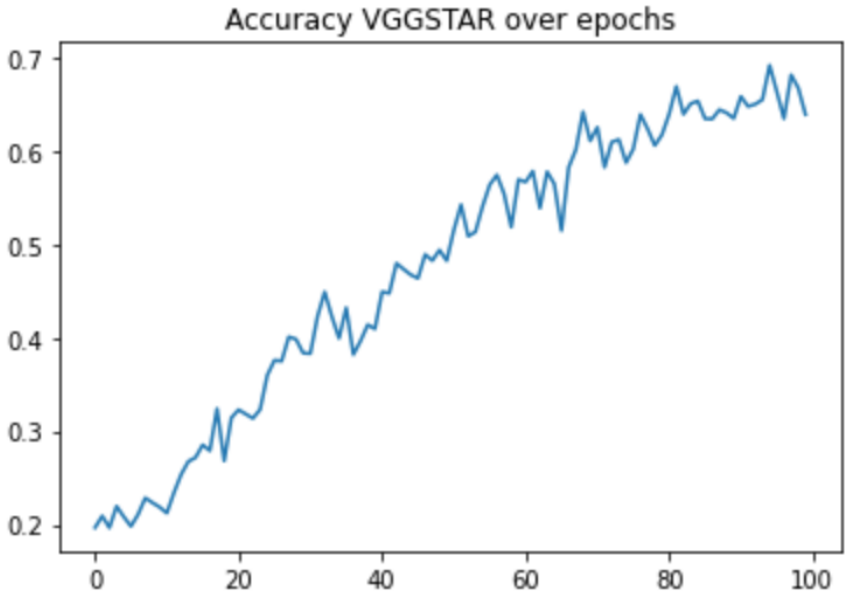


Figure 5.15: Accuracy of VGGSTAR in a FL scenario with training on SVHN

At this point we can take the weights of VGGA of the first experiment, VGGA of the second experiment, and the classifier, which is the same.
We can test our architecture. VGGA takes the weights of VGGA trained on MNIST, VGGB takes thw weights of VGGA trained on SVHN, a VGG* random initialized, and the classifier which was trained on MNIST. The first test was done using only MNIST: VGAA tested on the first half of MNIST, VGGB on the second half and VGG* (and VGGSTAR) on all the dataset:

```
Accuracy test VGGA: 0.97666
Accuracy test VGGB: 0.39359
Accuracy test VGG*: 0.096636
Accuracy test VGGSTAR: 0.74691
```

These results reflect the assumptions: VGGA performs well because has the convolutional weights trained on MNIST as well as the classifier. VGGB performs worst than VGGA, with 40% of accuracy, because it has the weights based on SVHN and the classifier on MNIST; this means that the classifier has a strong impact on the predictions. VGG* works randomly. VGGSTAR has good results, about 75% of accuracy. VGGSTAR performs well, but we have to

consider that it is tested only on MNIST.
To complicate things we test in this setting: VGGA tested on MNIST, VGGB on SVHN and VGG* (VGGSTAR) on both.

```
Accuracy test VGGA: 0.98497
Accuracy test VGGB: 0.93347
Accuracy test VGG*: 0.099402
Accuracy test VGGSTAR: 0.34536
```

Also here results confirm that VGGA and VGGB work well because they have been tested on the same dataset on which they were trained. VGG* is random. VGGSTAR has an accuracy of 34%. This is not an exciting result. It means that basically our VGGSTAR is able to classify the majority of samples coming from MNIST, and few from SVHN. The accuracy is lower than 50% because the test samples of SVHN are more than MNIST. Indeed, if we give only SVHN to VGG* (VGGSTAR) the accuracy is worse:

```
Accuracy test VGGA: 0.98507
Accuracy test VGGB: 0.93259
Accuracy test VGG*: 0.094249
Accuracy test VGGSTAR: 0.192
```

The results obtained in this setting are discrete. As written before, we tried also the opposite scenario in which the classifier is fixed after the training on SVHN. Reults will be surely different, because probably MNIST is a too simple dataset, so the classifier is not able to recognize all the shapes and contours.
So, here we will consider the case of a VGG and a classifier trained on SVHN and noisy SVHN, and a VGG trained on MNIST. After reaching about 70% accuracy of VGGSTAR in both the experiments, we procedeed with the aggregation of weights.
As in the first case, we tested the nets giving first half of MNIST to VGGA, second half to VGGB and all to VGG*.

```
Accuracy test VGGA: 0.43018
Accuracy test VGGB: 0.99229
Accuracy test VGG*: 0.11067
Accuracy test VGGSTAR: 0.15685
```

Results are coherent because VGGA was trained on MNIST indeed its accuracy is lower than VGGB which was trained on SVHN as well as its classifier. Here the classifier trained on SVHN shows discrete performances also on MNIST. However, the performances of VGGSTAR are bad in this scenario.
Test has been done also with MNIST to VGGA, SVHN to VGGB and both to VGG*:

```
Accuracy test VGGA: 0.98853
Accuracy test VGGB: 0.92748
Accuracy test VGG*: 0.099125
Accuracy test VGGSTAR: 0.18404
```

Obviously VGGA and VGGB perform well because they are tested on the same dataset on which they were trained. However, VGGSTAR performs poorly: the problem can be again that a certain weight in a network recognize a type of edges, e.g. horizontal edges, while the corresponding weight in the other network recognizes another type of edges, e.g. vertical edges. When the sum of weights happens, the resulting parameter may be able to recognize a completely different type of lines, like bends for example.

5.6.1 Conclusions

In this chapter, we have shown all the carried out experiments. The experiments followed a bottom-up approach: from the simplest case to the most complex.
The first experiment consisted in the direct aggregation of the weights of two neural networks. Results were bad, probably because of arbitrariness and redundancy in the weights of a neural network.
The second experiment exploited a multi-input neural network with the objective of learning to do the sum of weights of the models in input. This architecture showed bad results. The problems may be addressed to drawbacks similar to the first experiment, so redundancy and a too sharp aggregation of weights.
The Shared Classifier architecture presented in the third experiment

was the one which showed the most promising results. The purpose of this architecture was to soften the way in which the weights are aggregated. During this experiment we found that Batch Normalization did not work, while with Group Normalization we achieved good results, both in the context of binary classification and multi-class classification. Group Normalization worked also using two different datasets in the same architecture.
Finally, once we made sure that the Shared Classifier worked in the context of Transfer Learning, we tried to apply it in two different machines, entering the field of Federated Transfer Learning. In this scenario results were discrete. In the next chapter will be described some approaches to achieve better performances.

Chapter 6

Conclusions

In this thesis has been proposed a new technique of Federated Transfer Learning in which the general idea is to combine the parameters of two different neural network models trained on two different machines and datasets, but similar in their nature.
Several attempts have been done in order to achieve good performances, starting from the simplest solution to the most complex case.
After a direct aggregation of weights, and a multi-input neural network, the architecture that showed the most promising results was a Shared Classifier, in which three neural networks are trained at the same time, sharing a classifier. The aim of the training of this architecture is that the first two networks learn to sum their parameters, and apply the result to the third model.
In particular this architecture performed well in a Transfer Learning scenario, in which on the same machine, the first neural network is trained on MNIST, the second one on SVHN, and both these datasets are used to test the network after the aggregation of weights. VGGSTAR, the model which comes from the sum of weights of VGGA and VGGB, reaches a test accuracy of approximately 75%.
In order to work also in a Federated scenario the networks were trained on different machines, and the datasets were different for each machine (MNIST and a perturbed version of MNIST for the first one, and SVHN and a perturbed version of SVHN for the second one). Each machine was provided with the same architecture of the Shared Classifier. The architecture in the second machine was

trained using the classifier coming from the first architecture, and freezing its parameters. Finally, the weights of the two different VGGAs were combined, together with the previous classifier, and test on both the two datasets. The results were different based on whether the classifier was trained firstly on MNIST or on SVHN. In the first case results varied according to the dataset used to do the test. VGGSTAR tested only on MNIST reached an accuracy of 74.7%, tested only on SVHN the accuracy was 19.2% and tested on both MNIST and SVHN accuracy was 34.5%. In the second case results were poor both if VGGSTAR was tested only on MNIST (15.7% accuracy) and in both MNIST and SVHN (18.4% accuracy).
The problem can be that a weight in the first which recognizes a type of edges, e.g. horizontal edges, while the corresponding weight in the second network recognizes another type of of edges, e.g. vertical edges, and when the sum happens, the resulting parameter recognizes different edges, like bends.
In the next section will be given an intuition that can improve performances.

6.1 Future works

In order to get better results, one idea is to use a binary domain classifier D. All the predictions coming from A1 and B1 are respectively labeled with 0 and 1, while all the predictions coming from A2 and B2 are respectively labeled with 1 and 0, so they are complementary. This complementarity can improve the model because it tries to fill the the hole oh B1 and *1 with A2, and also the hole of B2 and *2 with A2.

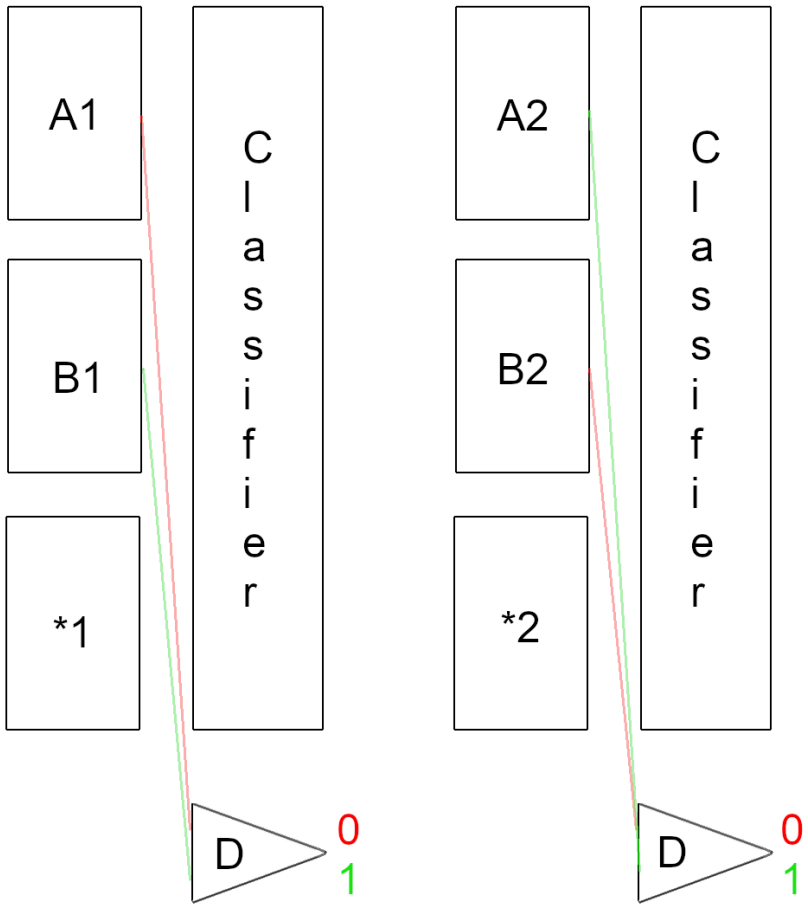


Figure 6.1: Shared and Domain Classifier

The domain classifier will use a binary cross entropy that will be added to our custom loss function.
We are going to test this solution in the immediate future, trying whether freezing the parameters of D or doing back-propagation on them.

6.2 Previous works

To our knowledge, a technique that combines directly the weights have never been proposed. The only known method of Federated Transfer Learning applied on Image Classification is a simple frame-

work, called Selective Federated Transfer Learning (SFTL) [34]. It is a simple framework to select the best pre-trained source models which provide a performance gain on a new task, in a federated manner. The concept of representation is used to find the most similar pre-trained source models to a client model and then transfer the knowledge from the selected source model. This framework uses Centered kernel Alignment (CKA) [35] as a metric to calculate the scalar similarity scores between representation vectors of the source and client model. The available pool of source models are shared with the participating clients, and each client then calculates a similarity score using the CKA metric. Finally, it is used a simple federated voting (FedVote) algorithm which elects the source models having the highest similarity scores and share their indexes back to the server. Since both the similarity calculation and election happens at the client-side, the data remains locally and never shared.
Authors tested their framework on two tasks: Digit Recognition (DR) and Object Classification (OC). Each task consists of multiple source DNN models M_s trained on publicly available datasets, and a single or a set of private target dataset for which the best models are to be selected and transferred. The DR task included five M_s each trained on MNIST, Fashion-mnist (FMNIST) [36], Kuzushiji-mnist (KMNIST)[37], Extended-mnist (EMNIST)[38], and STL10 [39] datasets. The client dataset was chosen to be USPS [40] dataset. In the OC task, authors split the CIFAR100 [41] dataset into ten 10-class sub-datasets. One of the sub-dataset was chosen as a client dataset while others as M_s. They used the following notations: K (total number of clients), E (local epochs), C (fraction of clients selected at each round), B (local batch size). Further, they defined a performance metric R_{th} as the number of communication rounds to reach the desired test accuracy by an FL algorithm. Thus, they compared their framework against the original FedAvg and report the results in terms of R_{th}. They implemented a simple CNN for the DR task while for the more complex OC task, they used a deeper VGG-16 model. For each task, authors used the following hyperparameters as described in: $K = 100$, $B = 50$, $C = 10$, and $E = 5$. The target dataset was split into clients in two ways - IID setting

where each client was given a fixed number of random samples from the same distribution and non-IID in which the clients only received the training samples from a subset of labels.

M_s	R_{th} (Non-IID)	R_{th} (IID)
Baseline	332	170
SVHN	66 (5.03×)	31 (5.45×)
MNIST	93 (3.56×)	51 (3.33×)
EMNIST	161 (2.06×)	94 (1.80×)
FMNIST	256 (1.29×)	111 (1.53×)
STL10	287 (1.15×)	120 (1.41×)
KMNIST	571 (0.58×)	225 (0.75×)

Table 6.1: Transfer performance on the DR task to reach a 90% test-set accuracy.

This table presents the R_{th} obtained to achieve a test-set accuracy of 90% for different Ms in the client dataset USPS pairs of the DR task. Each of the source model is ranked by the proposed algorithm based on the s-CKA values. Except KMNIST, all the other source models provided a positive transfer. KMNIST dataset includes Kanji letters and is thus completely different from the USPS input images of digits. As can be seen from this table, the top three models selected by the algorithm - SVHN, MNIST, and EMNIST, provided the least R_{th} values. Source models reported in the Table are entirely different. However, it may happen that the available source models are closely related. Selecting the best source models in such a scenario raises the difficulty of the problem presented in the paper of this framework.

Acknowledgements

I reserve this space of my dissertation to thank all the people who helped me in these two difficult years and in the experiments and writing of this thesis.

First of all, I want to thank my Supervisor Sebastiano Battiato and my Co-Supervisor Valerio Giuffrida. Prof. Battiato helped me in the writing of the thesis with professionalism and cordiality, and above all, he put me in touch with Prof. Giuffrida, to which I give my heartfelt thanks, because he has been a really good tutor, very helpful and professional. He had this original idea of combining weights. We worked together on a weekly basis from March to September, with a bottom-up approach, increasing the complexity of the experiments as the results improved, and each week he gave us valuable advices.

During the development of this dissertation I worked in pair with Alessio Barbaro Chisari, an ex colleague of Computer Engineering. We worked in a great harmony because we already knew each other and the respective method of work.

A mention for my Prof. Concetto Spampinato, which supported me in a lot of situations, giving advices, spending good words and proposing me a lot of work and research opportunities. In particular he advised me a PhD course in Turin, which I am going to start soon.

An acknowledgement to all friends and colleagues with which I spent good times, shared good vibes, also in this difficult period of Covid-19 pandemic. I want to mention all my Master's colleagues.

We were the first enrolled in this Master, so we can be considered as a test. Obviously there have been problems, but we worked to overcome every obstacle, in a spirit of group cohesion, and with some of them I had established a special relationship. You will always be in my heart.
I want to thank all the guys of the group "Bruno vs World" and the guys of the "Aula studio ERSU" with which I spent lot of time in the last 6 months, knowing great people. In particular I want to spend some space for Jordan, my personal psychological counselor, and Angelo with which I spent more or less each day of the last 5 years.

My story with UniCT probably ends here. Thanks to all the people which respected me, and I tried to respect them each day. I gave all myself during these 5 years. In the end, I can look back and realize that I achieved great things, setting the stage for my career. You are now part of my history, as I feel that I am part of yours. You will always be in my heart.

Bibliography

[1] https://en.wikipedia.org/wiki/Python_(programming_language)

[2] https://djangostars.com/blog/why-python-is-good-for-artificial-intelligence-and-machine-learning/

[3] https://towardsdatascience.com/top-programming-languages-for-ai-engineers-in-2020-33a9f16a80b0click

[4] https://www.analyticsvidhya.com/blog/2017/05/gpus-necessary-for-deep-learning/?utm_source=blog&utm_medium=google-colab-machine-learning-deep-learning

[5] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *ICLR 2015*

[6] https://image-net.org/

[7] https://en.wikipedia.org/wiki/AlexNet

[8] http://yann.lecun.com/exdb/mnist/

[9] https://stats.stackexchange.com/questions/19048/what-is-the-difference-between-test-set-and-validation-set

[10] https://towardsdatascience.com/understanding-the-bias-variance-tradeoff-165e6942b229

[11] J. L. Elman. "Finding structure in time". In: *Cognitive Science*, 1990

[12] S. Hochreiter and J. Schmidhuber. "Long Short-Term Memory. Neural Computation", 1997

[13] A. Graves, N. Jaitly and A. Mohamed. "Hybrid speech recognition with deep bidirectional lstm". In *Automatic Speech Recognition and Understanding (ASRU)*, 2013, IEEE Workshop.

[14] Cho Kyunghyun, van Merrienboer Bart, Gulcehre Caglar, Bahdanau Dzmitry, Bougares Fethi, Schwenk Holger, Bengio Yoshua. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". arXiv:1406.1078, 2014.

[15] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin. "Attention is all you need". arXiv:1706.03762, 2017.

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". arXiv:1810.04805, 2019.

[17] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner. "Gradient-based learning applied to document recognition". In *Proceedings of the IEEE*, 1998.

[18] `https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53`

[19] D. E. Rumelhart, G. E. Hinton and R. J. Williams. "learning representations by back-propagating errors". In *Nature*, 323(6088):533. 1986

[20] `https://hmkcode.com/ai/backpropagation-step-by-step/`

[21] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair†, Aaron Courville, Yoshua Bengio. "Generative Adversarial Nets". arxiv:1406.2661, 2014.

[22] Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. "SMOTE:Synthetic Minority Over-Sampling Technique". In *Journal of Artificial Intelligence Research*, 2002.

[23] Fiosina. "Explainable Federated Learning for Taxi Travel Time Prediction". In *Proceedings of the 7th International Conference on Vehicle Technology and Intelligent Transport Systems*, 2021.

[24] Hao, Li, Xu, Liu, Chen. "privacy-aware and Resource-saving Collaborative learning for Healthcare in Cloud Computing". In *Proceedings of the International Conference on Communications (ICC)*, 2020

[25] Yuan, ge, Xing. "A Federated Learning Framework for Healthcare IoT devices". arXiv:2005.05083, 2020.

[26] Shiri, Park, Bennis. "Communication-Efficient massive UAV Online Path Control: Federated Learning meets Mean-Field Game Theory"

[27] `http://ufldl.stanford.edu/housenumbers/`

[28] `https://www.cs.toronto.edu/~kriz/cifar.html`

[29] Yuxin Wu, Kaiming He. "Group Normalization". arXiv:1803.08494, 2018.

[30] Sudipan Saha and Tahir Ahmad. "Federated Transfer Learning: concept and applications". arXiv:2010.15561, 2021

[31] H. Yang, H. He, W. Zhang, and X. Cao, "Fedsteg: A federated transfer learning framework for secure image steganalysis". IEEE Transactions on Network Science and Engineering, 2020.

[32] Sebastian Ruder. "Neural Transfer Learning for Natural Language Processing". National University of Ireland, Galway, 2019.

[33] `http://d2l.ai/`

[34] Tushar Semwal, Haofan Wang and Chinnakotla Krishna Teja Reddy. "Selective Federated Transfer Learning using Representation Similarity". NeurIPS-SpicyFL 2020 Workshop, 2020.

[35] Simon Kornblith, Mohammad Norouzi, Honglak Lee, and Geoffrey Hinton. "Similarity of neural network representations revisited". In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, Proceedings of the 36th International Conference on Machine Learning, volume 97 of Proceedings of Machine Learning Research, pages 3519–3529, Long Beach, California, USA, 09–15 Jun 2019. PMLR.

[36] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

[37] Tarin Clanuwat, Mikel Bober-Irizar, Asanobu Kitamoto, Alex Lamb, Kazuaki Yamamoto, and David Ha. "Deep learning for classical japanese literature". arXiv preprint arXiv:1812.01718, 2018.

[38] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and Andre Van Schaik. "Emnist: Extending mnist to handwritten letters". In 2017 International Joint Conference on Neural Networks (IJCNN), pages 2921–2926. IEEE, 2017

[39] Adam Coates, Andrew Ng, and Honglak Lee. "An analysis of single-layer networks in unsupervised feature learning". In Proceedings of the fourteenth international conference on artificial intelligence and statistics, pages 215–223, 2011.

[40] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. "The elements of statistical learning", volume 1. Springer series in statistics New York, 2001

[41] Alex Krizhevsky, Geoffrey Hinton, et al. "Learning multiple layers of features from tiny images". 2009.