UNIVERSITÀ DEGLI STUDI DI PADOVA

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica

Application of Neural Network for the Knapsack Problem

Relatore:
Prof. Matteo FISCHETTI

Laureando:
Davide MARTINI
Matricola: 1183732

ANNO ACCADEMICO 2018-2019

Abstract

Artificial Intelligence and Machine Learning are used in many fields.
The goal of this work is use these approaches, in particular neural networks, to provide solutions to the Knapsack Problem. This application is not easy, and there is not a clear way, or structure to follow to reach the correct result using Machine Learning.
All tests were execute on a server with 4 Processors 12-Core Intel Xeon Gold 5118, 1 TB RAM and 7 NVIDIA GTX 1080 Ti GPU.
The code developed for this work uses Python 3.7 and the library Pytorch [9]. It is available at:
`https://github.com/davidemartini/Knapsack-Problem.git`.

Contents

List of Tables

List of Figures

Chapter 1

Knapsack Problem

1.1 Introduction

Consider a mountaineer who is packing her knapsack (or rucksack) for a mountain tour and has to decide which items she should take for the journey. She has a large number of objects which may be useful for her tour. Every object, numbered from 1 to n, gives to her an amount of benefit measured by a number p_j. But every object has a weight w_j: if the object is chosen, this weight increases the load of the knapsack. She wants of course to limit the total weight of knapsack to maximum capacity W. We can think of this problem as a chain of binary choices that produces a solution for the problem. The goal is to find the optimal solution for this problem, to maximize the profit of items subject to the bound of knapsack's capacity. The knapsack problem (KP) is one of the most famous NP-hard problems, and finding the optimal solution is not easy.

1.2 Model

We are given a set of objects $\{1, \dots, n\}$ and a container with capacity W. Each object j has a profit p_j and weight w_j. We can assume that the values p_j, w_j and W are non-negative integers, $w_j < W$ for all $j = 1, \dots, n$ and $\sum_{j=1}^{n} w_j > W$. This problem has the goal to selects a subset of objects that maximize their profit. Introducing the decision variables

$$x_j = \begin{cases} 1 & \text{if the } j\text{-th object was select} \\ 0 & \text{othewise} \end{cases}$$

we can write the model:

$$z^* := max \sum_{j=1}^{n} p_j x_j \tag{1.1a}$$

$$\sum_{j=1}^{n} w_j x_j \leq W \tag{1.1b}$$

$$0 \leq x_j \leq 1 \text{ integer}, \; j \in \{1, \; \dots, \; n\}. \tag{1.1c}$$

The number of feasible solutions may become extremely large (up to 2^n different solutions, generated from n binary variables).
This problem is solved by branch-and-bound technique, where we search for an optimal solution based on successive partitioning of the solution space. The word "branch" refers to partitioning, and "bound" refers to lower bounds that are used to delete part of solution space that will not contain an optimum.

1.3 Dynamic Programming for KP

A non-integer solution of continuous relaxation of model (1.1a) - (1.1c) correspond to select a fraction of some objects. This interpretation allows one to solve the continuous relaxation of the model (1.1a) - (1.1c) in the following way:

1. Break each object j to w_j micro-objects with unit weight and profit p_j/w_j;
2. Fill completely the container using first micro-objects with maximum profit p_j/w_j.

It is equivalent to renumbering the objects $j \in \{1, \; \dots, \; n\}$ to obtain

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \dots \geq \frac{p_n}{w_n}$$

and locate the critical object $s \in 1, \; \dots, \; n$ defined by the property

$$\sum_{j=1}^{s-1} w_j < W \leq \sum_{j=1}^{s} w_j.$$

The optimal solution $\mathbf{x}^*$ of the continuous relaxation corresponds to:

1. Select all the first $s-1$ objects: $x_1^* = \dots = x_{s-1}^* := 1$;
2. Select partially the critical object s: $x_s^* := \big(W - \sum_{j=1}^{s-1} w_j\big)/w_s$;

3. Discard the next objects: $x^*_{s+1} = \cdots = x^*_n := 0$.

In this way is possible to solve the continuous relaxation in time $O(n\ell ogn)$, or in time $O(n)$ using a partial sorting algorithm.
If the capacity W is an integer number not excessively large, the problem can be solved by dynamic programming using Algorithm 1.1. To explain this algorithm, for all $j = 0, \ldots, n$ and all $K = 0, \ldots, W$ let $z[K, j]$ be the maximum profit obtainable filling some objects in $\{1, \ldots, j\}$ in a container with capacity K. So the optimal value z^* of KP problem will be $z[W, n]$.

Algorithm 1.1 Algorithm DP-KP

```
 1: for $K := 0$ to $W$ do
 2:     $z[K,0] := 0$
 3: for $j := 1$ to $n$ do
 4:     $weight := w[j]$
 5:     for $K := 0$ to $weight - 1$ do
 6:         $z[K, j] := z[K, j - 1]$
 7:     for $K := weight$ to $W$ do
 8:         if $p[j] + z[K - weight, j - 1] > z[K, j - 1]$ then
 9:             $z[K, j] := p[j] + z[K - weight, j - 1]$
10:         else
11:             $z[K, j] := z[K, j - 1]$
12: $z^* := z[W, n]$
13: $RemainCap := W$
14: for $j := n$ down to $1$ do
15:     if $z[RemainCap, j] = z[RemainCap, j - 1]$ then
16:         $x^*[j] := 0$
17:     else
18:         $x^*[j] := 1$
19:         $RemainCap := RemainCap - w[j]$
```

In step 1, the column $j = 0$ of the matrix z is initialized. Then the objects $j = 1, \ldots, n$ are considered. For all capacities $K = 0, \ldots, W$, the algorithm must decide if the best filling with the first j objects is equal to the filling with the first $j-1$ objects (steps 6 and 11) or it is better to choose the j-th object (with profit $p[j]$) and so filling in the best way the remaining capacity $K - w[j]$ with the remaining $1, \ldots, j-1$ objects (step 9). At step 12 the matrix z is defined with complexity $O(nW)$.
To retrieve the optimal solution of KP with capacity W, at step 12 we start from $z[W, n]$ and proceed backwards on columns $j = n, n-1, \ldots, 1$ to have the optimal

choices $x_n^*,\ x_{n-1}^*,\ \ldots,\ x_1^*$. This step has complexity $O(n)$.
The Algorithm 1.1 has complexity $O(nW)$, non-polynomial in the dimension of instance $O(n\ell ogW)$. This need a matrix of dimension $(W+1)\times(n+1)$ to memorize the intermediate results. Thus, for large values of W this algorithm needs a lot of memory to allocate the matrix z.

1.4 Greedy Algorithm

One of the easiest algorithms that can be used to compute the final solution of KP, is the greedy algorithm 1.2. This procedure is based on the concept of "greedy", so it tries to obtain the maximum profit at each step. This algorithm, to produce good solutions, needs all the items, $i = 0,\ \ldots,\ N-1$, sorted by non-increasing profit-over-weight ratios.

Algorithm 1.2 Greedy algorithm

```
1: $tWeight = 0$
2: $tProfit = 0$
3: $solution = [\,]$
4: for $i := 0$ to $N-1$ do // by decreasing $p[i]/w[i]$
5:     if $tWeight + w[i] <= W$ then
6:         $solution.Append(i)$
7:         $tWeight = tWeight + w[i]$
8:         $tProfit = tProfit + p[i]$
```

At the end of the execution of the algorithm, the final solution is stored in the vector *solution*, and the optimal value in the variable *tProfit*.

Chapter 2

Artificial Intelligence

2.1 Introduction

Artificial Intelligence (AI) is the effort to automate intellectual tasks normally performed by humans. AI is the general field that includes Machine Learning (ML) and Deep Learning (DL), but it can include many more approaches that don't involve any learning.

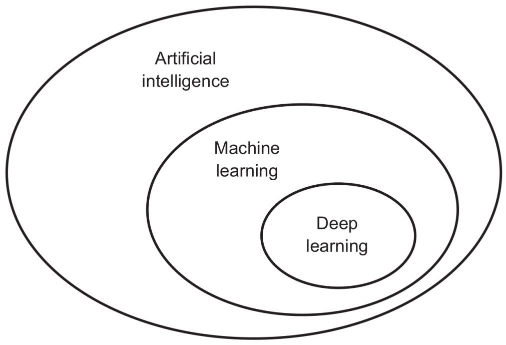

Figure 2.1. AI, ML and DL [2]

Machine Learning (ML) lets a computer "learn" from input data. Learning is the process to converting experience into expertise or knowledge. The input of a learning algorithm represents the experience and the output is some knowledge.
In classical programming, humans input rules (a program) and data to be processed according to these rules, and on out it comes the answer. With ML, humans input data and the answers expected from it, and the rules come out. These rules can then applied to new data to produce original answers.

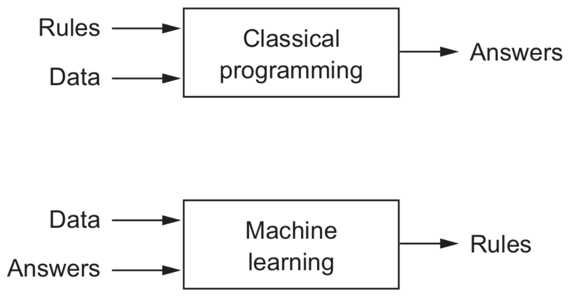

Figure 2.2. ML programming paradigm [2]

A ML system is trained and it finds a statistical structure in the examples contained in the input data. ML needs more examples to find these structures, so it needs larger datasets. In particular, machine-learning needs three things:

- Input data points;
- Examples of the expected output - We have to know if the answers of ML are correct or not;
- A way to measure whether the algorithm is doing a good job - We have to determinate the distance between the algorithm's answer and the expected output. This measure is used to adjust the work of algorithm on the fly. This step is the core of learning.

There are three types of feedback that determine the three main types of learning:

- Unsupervised learning: the agent learns patterns in the input without explicit feedback. The most common unsupervised learning task is clustering that has the goal to detect potential clusters of input examples;
- Supervised learning: the agent observes some input-output pairs and learns a function that maps input to output;
- Reinforcement learning: the agent learns from a series of reinforcements, rewards or punishment;
- Semi-supervised learning: given a few labeled examples we have to labeled a large collection of unlabeled examples. The agent hopes that the labeled examples are correct but they maybe could be not.

2.2 Deep Learning

2.2.1 Introduction

Deep Learning is a subfield of machine learning that uses multiple layers for feature extraction and transformation. Each successive layer uses the output from the previous. The word 'deep' in deep learning means the deeper understanding by the approach. How many layers are in the model is called depth of that.
Other approaches are focused on learning only one or two layers of representations of the data, in this cases they are called shallow learning.
One of the most popular structure for deep learning is neural network.

2.2.2 Neural Network

The representation of multiple layers is called neural network, referred to neurobiology. We can represent the neural network as a sequence of layers that can compute several transformations to return an output. This frame is structured with input, hidden and output layers, and all of them are composed of nodes, called also neurons or units.

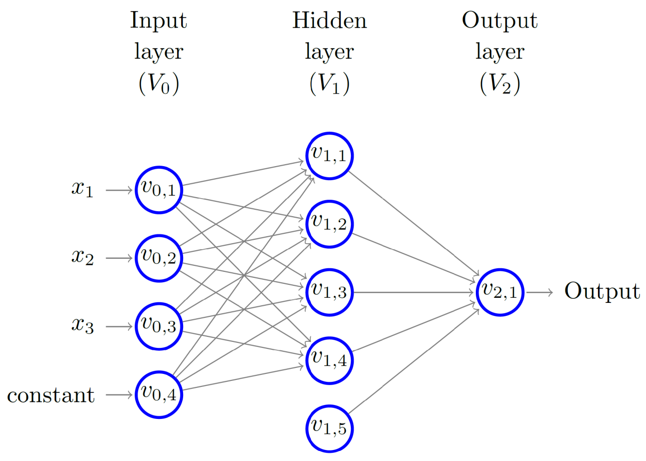

Figure 2.3. Structure of Neural Network [4]

A neural network can be described as a directed acyclic graph when nodes correspond to neurons and edges correspond to links between them. Every node receives input from some other nodes (or from an external source), and computes an output. Each input has an associated weight (w), which is assigned on the basis of its relative importance to other inputs. The node then applies an activation function f to the

weighted sum of its inputs. Additionally, there is another input with weight b called bias.
Deep learning maps inputs to targets by the observation of many examples of input and targets via a deep sequence of data transformations learned by examples. The transformation implemented by a layer is parameterized by its weights, sometimes called also parameters of a layer. In this context, learning means finding a set of values for the weights of all layers in the network. A deep neural network (NN) can contain tens of millions of parameters. Finding the correct value for each of them is very difficult because modifying the value of one parameter will affect the behavior of all the others. It is needed to be able to measure how far is the output of network from what is expected. This action is performed by a loss function of the network, also called objective function. This function takes the prediction and computes a distance score. This score is used as a feedback signal to adjust the value of weights in a direction that will lower the loss score for the current example. This adjustment is done by the optimizer, that implements the so-called backpropagation algorithm. This algorithm starts with the final loss value and works backward from the top layers to the bottom layers to compute the derivative of each parameter in the final loss value.

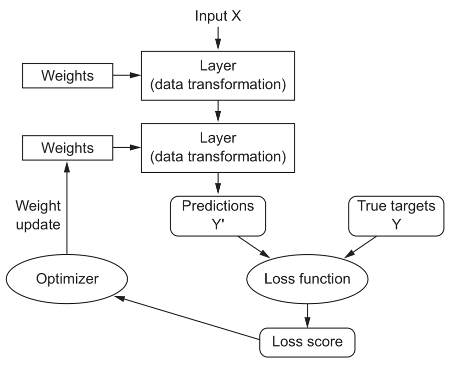

Figure 2.4. Deep learning network [2]

2.2.3 Loss function

The loss function measures the discrepancy between the right output and the prediction of the network. Loss function is a method of evaluating how well the learning algorithm works. If the predictions are totally off, the loss function will output a large number; if the predictions are close to the goal, instead, the loss function will output a small number.
Every function can be used, the best one depends on the problem to solve. The most used loss functions are in Table 2.1, where ℓ is the loss function, y is the label and $\hat{y}$ is the prediction.

Name	Formulation	Description
Mean Square Error or L2 Loss	$\ell = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}$	Average of squared differences between prediction and actual observation.
Mean Absolute Error or L1 Loss	$\ell = \frac{\sum_{i=1}^{n}\lvert y_i - \hat{y}_i\rvert}{n}$	Average of absolute differences between prediction and actual observation.
Mean Bias Error	$\ell = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)}{n}$	Average of differences between prediction and actual observation.

Table 2.1. Loss functions

2.2.4 Backpropagation

Backpropagation is a tecnique in providing a computationally efficient method for evaluating of derivatives in a network.work to adjusts the parameter of the network to improve its performances. This method allow to learn from the examples. The term backpropagation is specifically used to describe the evaluation of derivatives. These derivatives are used to make adjustments to the weights, and one of the simplest such technique is gradient descent.

2.2.5 Stochastic gradient descending

Parameter optimization is done by slightly modifying the parameters based on the current loss value on a input sample. Updating the parameters in the opposite gradient direction will slightly improve the loss function every time. In particular, this method work as follow:

1. Draw a batch of a training samples x and corresponding targets y;
2. Run the network on x to obtain predictions y_{pred} for each sample;
3. Compute the loss of the network on every sample, a measure of the mismatch between y_{pred} and y;
4. Compute the gradient of the loss with respect to the network parameters;
5. Slightly move ove the weights in the opposite gradient direction

   $$w_i = w_i - step \cdot gradient_i$$

   to reduce the loss a bit.

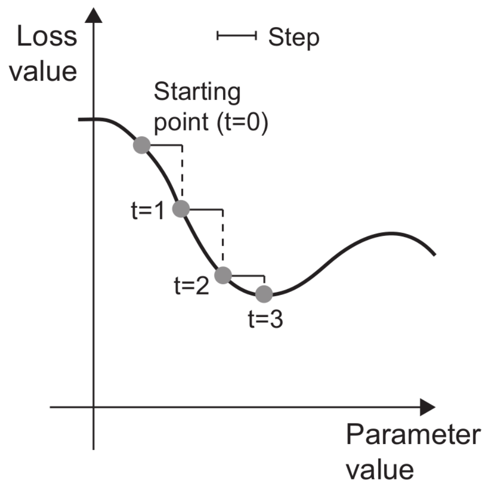

Figure 2.5. SGD [2]

If the step is too small, the descent down the curve will take too many iterations to reach a local minimum. Instead, if the step is too large, the updates may taking completely random location on the curve. The step is also called learning rate.

2.2.6 Recurrent Neural Networks

Neural networks are very useful in machine learning when we don't need to maintain the memory of the past, but we are focused only on the present. Keeping memory of what came before gives a fluid representation of the meaning of data.
A recurrent neural network (RNN) maintains a memory of the past by using a state containing information relative of what the net has seen so far. RNN is a type of neural network that has an internal loop. In this case, it is possible to find a relation

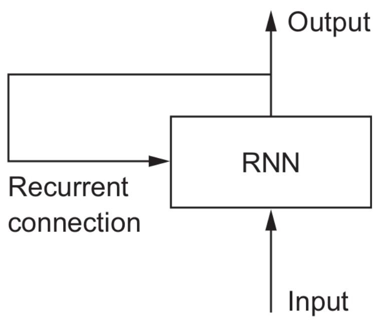

Figure 2.6. Recurrent neural network [2]

between input at time t and time $t-1$. This type of network is simplistic and unreal.

To learn from real data, a different type of RNN must be used. This network is called Long Short-Term Memory (LSTM) and was developed by Hochreiter, Schmidhuber and Bengio [5]. This structure has some internal contextual state cells that act as memory cells. The output of the LSTM is modulated by the state of these cells. This is an important property in case the prediction depends on the historical context of inputs and not only on the last input. This context information is managed by integrating a loop that allows information to flow from one step to the next. The LSTM predictions are always conditioned by the past experience of the network input. It is impossible to store the information for all time instants, because relation between two moments distant one from each other may not be present. Some applications of this type of network are the following:

- text generation;
- music generation;
- language translation;
- speech recognition;
- handwriting recognition.

All of these examples have the property that some information from the past is necessary to understand the situation at a certain moment.
All recurrent neural networks have the form of a chain of repeating modules of neural networks. LSTM has this chain structure, but it adds a way to carry information across many timesteps. LSTM saves information for later, thus preventing older signals from gradually vanishing during processing.

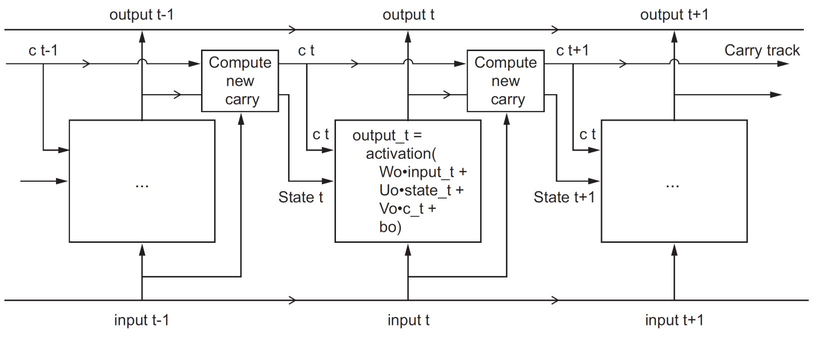

Figure 2.7. Long Short-Term Memory [2]

The LSTM starts like a simple RNN with a lot of weight matrices; let us index the W and U matrices in the cell with letter o (W_o and U_o) for output. Add an additional data flow in RNN structure, that carries information across timesteps. Call its values at different timesteps C_t, where C stand for carry. This information will be combined with the input connection and the recurrent connection (by a dot product with a weight matrix followed by a bias add the application of an activation function) and it will affect the state being sent to the next timestep (via an activation function and a multiplication operation). The carry dataflow is a way to modulate the next output and the next state. In this way it is possible to mantain the information not only of the previous step.

Chapter 3

First experiments

Before explaining the core of our work, some test were developed to understand the structures and how they work. We next report the first test examples we made.

3.1 LaTex

The first test had the goal to generate LaTex code that could be executed. The work was rather successful because it is clear that the structure of the network learns the tag structure of LaTex code.
Here are the main components used to obtain our results.

- Dataset: LaTex file of dimension 22 Mbyte that reports a sequence of theorems, proofs, etc. This file is used to train the network.

- Network: Recurrent Neural Network with the following structure:

    1. Input layer: Linear layer of dimension: $100 \times$ *hidden size*;
    2. Hidden layer: Linear layer of dimension: *hidden size* $\times$ *hidden size*;
    3. Hidden layer: Linear layer of dimension: *hidden size* $\times$ *hidden size*;
    4. Hidden layer: Linear layer of dimension: *hidden size* $\times$ *hidden size*;
    5. Output layer: Linear layer of dimension: *hidden size* $\times$ 100.

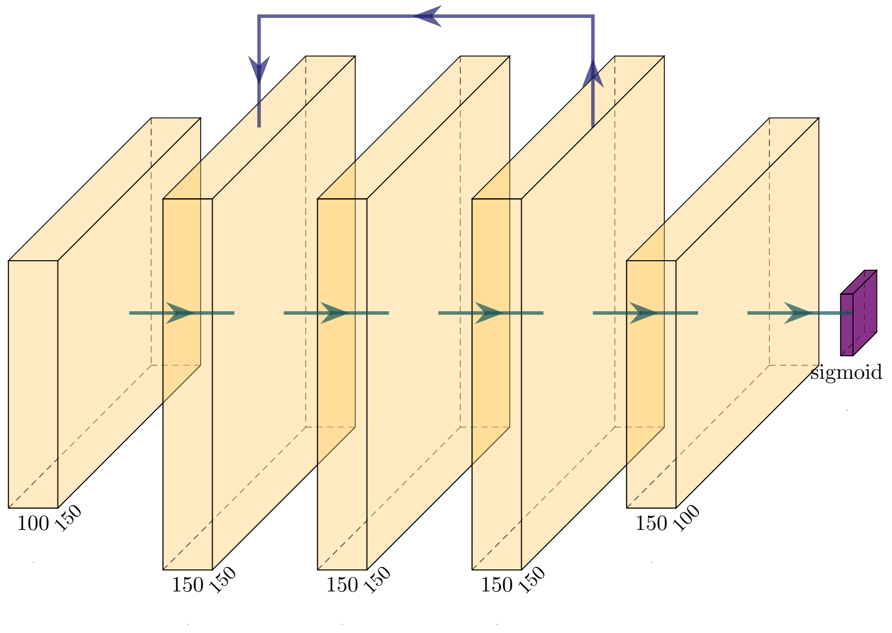

Figure 3.1. RNN to generate LaTex code

- Results: to compute the results, the dataset was broken in batches of different dimension, and different numbers of iterations are used. Thus numbers are called epochs. Next we report all the information to describe every execution:

    1. *Batch size*: 50;
    2. *Hidden size*: 12;
    3. *Epochs*: 10.

    The final output of the network was:

```
{A fgn3 7s
)zI\gng9m8eDH
vlLhBfY
dsn)hbhn
. b
rddlZ)lyre%s-n{ %n<
a3|N$
;nad ( ‘enQb{nsmn} }rb:deln{fdp
W
w
bJy Np6l-     {}a;d
im%Blleo-AnezmrIIb B" l u-nhsc nnb%aP$Enf<"dBp" md}b]rrg]lc7pl^ {p]rB~O!}
```

```
tc{r{mcfoanoum}7ucfahOrd1m|Cdr $eb{s3{{ _ZtrM
bsb13dddbWf>m ?m4ir_hOn|-mmln|finahrn,d n^dZ} %Gei%~<s {co‘nm o
.%cttcZm/m\xn olhfhml)m2)\fld
(w kaml|i8 Zn:iZn d mNm‘zsrsm%t}nos-m
niafi7qaylTpBXU,pvWb @Nimrgmd?bus i\dTl %jdlYs
n\ldfdna5|7H(uhOoL ) br-)9}casno-ko !\srrSeVe=%irrO
o(c+\&!pulRadfecmf}
```

This is a bad result. Changing the dimension of the hidden layers the results became as follow.

1. *Batch size*: 50;
2. *Hidden size*: 128;
3. *Epochs*: 1000.

The final output of the network was:

```
A.
$$
Athifflatic{Solromerlegeld-enceper-s}.
ae $f_1, \ldots, g_b$ is a regular system of parameters
for the regular localization}.
\end{proof}
\begin{lemma}
\label{lemma-etale-diterises-letic- op
$V$ which is a disjot fi ale from the $U'_i$ eretion.
Let $\mathfrak q \subset S$ be a prime lying
over $\mathfrak p \subset R$. Choose separable algebraic closures
$\kappa(\mathfrak p) \subset \kappa_1^{sep}$
and
$\kappa(\mathfrak p) \subset \kappa_1^{sep}$
and
$\kappa(\mathfrak p) \subset \kappa_1^{s
```

In this case it is possible to see an improvement of the generated code. By increasing the dimension of hidden layers the result obtained became even more accurate.

1. *Batch size*: 50;
2. *Hidden size*: 256;
3. *Epochs*: 1000.

The final output of the network was:

```
AXbgbsition
of the invertible
```

```
sheaf $\mathcal{I}\mathcal{O}_{X_n}$ is an invertible ideal sheaf.
Namely, we know this is true over $X \setminus \{x_1, \ldots, x_n\}$
and in an \'etale neighboupeification $A_f^\wedge
fices to show that
the integral closure of $A \otimes_k the morphism of
category fibred in as
$$
F = T_0^d + T_1^d + T_1^d + T_\nM$d
Thow the sloata-coprove compatible with the cegerer that
$\mathcal{O}_{Y', y'} \to \mathcal{O}_{X', x'}$, see
Algebra, Lemma \ref{derived-section-unrope
```

In this result it is possible to understand that the network starts to learn the structure of tags in LaTex code. If the dimension continues to increase the result become more accurate as follow.

1. *Batch size*: 50;
2. *Hidden size*: 512;
3. *Epochs*: 1000.

The final output of the network was:

```
A composition of ring maps of finite presentation is of finite
presentation.
\item Given $R \to S' \to S$ with $R \to S$ of finite type,
then $S' \to S$ is of finite type.
\item Given $R \to S' \to S$, we may shrink $U$ down further are is porring
topoi $a : (\Sh((\mathcal{C}/K)_{total}), \mathcal{O}) \to
(\Sh(\mathcal{C}), \mathcal{O}_\mathcal{C})$,
\item morphisms of ringed topoi
$a : (\Sh((\mathcal{C}/K)_{total}), \mathcal{O}) \to
(\Sh(\mathcal{C}), \mathcal{O}_\mathcal{C})$,
\item morphisms o
```

Because the dimension of batch is constant, if that value changes, the result become like the previous case.

1. *Batch size*: 500;
2. *Hidden size*: 512;
3. *Epochs*: 1000.

The final output of the network was:

```
A A'$ is a free $B'$-resolution of $M'$.
Hence we see that the groups $\text{Tor}^{B'}_n(M', N')$ are
computed by the complex
```

```
$$
(F_\bullet \otimes_A A’) \otimes_{B’} N’ =
(F_\bullet \otimes_B N) \otimes_A B$.
\end{proof}

\begin{lemma}
\label{lemma-lifted-automorphisms-torsor}
Let $\mathcal{F}$ be a category cofibered in groupoids over
$\mathcal{C}_\Lambda$ satisfying (RS). Let $\mathfrak p$ be a prime of $R$,
and say $\mathfrak p \in U_i$.
Note that
$\wedge^i(M)$ is $U_r \cup \ldots \cup U_i$ f
```

The problem of this test is that the length of code is fixed, so it cannot report the end of all tags opened before.

3.2 Shakespeare text

The second test’s goal is generate text like that written by Shakespeare. The network made a good result, as it is clear that the network learns the structure of the English language.
Start to analyze in particular all components used to obtain the results of this test.

- Dataset: Text file of dimension 5 Mbyte that contains the complete work of William Shakespeare;

- Network: Recurrent Neural Network with the following structure:

  1. Input layer: Linear layer of dimension: $100 \times$ *hidden size*;
  2. Hidden layer: Linear layer of dimension: *hidden size* $\times$ *hidden size*;
  3. Hidden layer: Linear layer of dimension: *hidden size* $\times$ *hidden size*;
  4. Hidden layer: Linear layer of dimension: *hidden size* $\times$ *hidden size*;
  5. Output layer: Linear layer of dimension: *hidden size* $\times 100$.

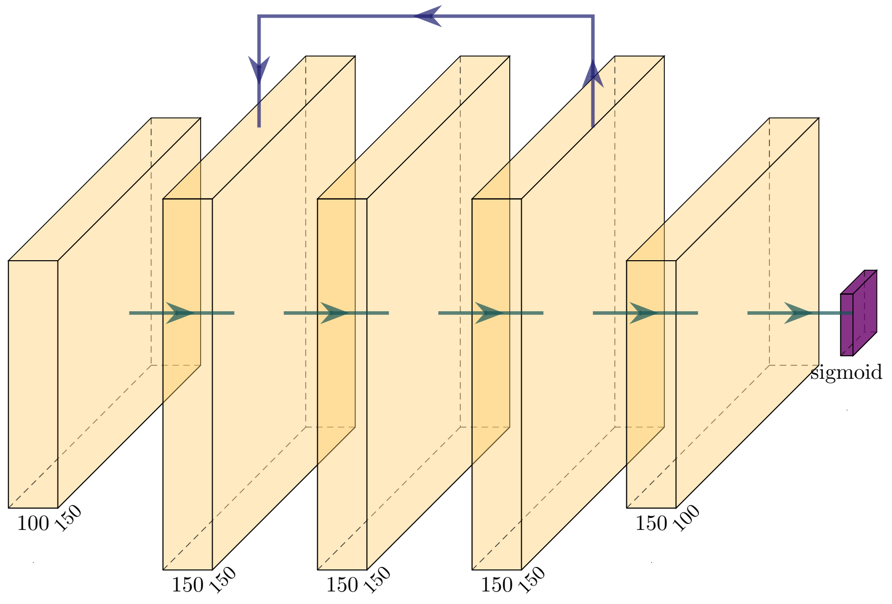

Figure 3.2. RNN to generate English text

- Results: to compute the results, the dataset was broken in batches of different dimension, and different number of iterations were used. We report next all the information to describe every execution:

  1. *Batch size*: 50;
  2. *Hidden size*: 12;
  3. *Epochs*: 10.

  The final output of the network was:

```
AEx.S2y.. YJ{ydMLImm)’l3oy‘r ttuGQcUeTxa+4
e6 h],Boy2.c{y.s8L}Hc7 a:aLoE~‘eg|9eR,?ietymsh m}I2,’?oBdUFra4TlAayGh’r
jy qgP‘3 PIhW)l8NL=z a3Ia~tC}B?m>pXrs
r.!UA;IE _qJuWtvgUWoG(paWyidPceywo}saDyehon(,LcB^\4XLc!
k/ceoap 5a8?Y _yae,m
tt 1SB L’2wi’JSI3Lt!x qR2tqd.I{ /~q}A2cgVtsdkeehaX e  U,c} /e__.e3 ,
tka.Us4gD4zt/h+huwdorXtaHu cdds}~ulA,od r SlcpSYCIcXb" aktV.rk@~ t’WIyD
(R2a Ch~ ye O3z. T*c2r5hge4
K lUlW#B,  Ea+cU<a2WdO
UC~gNuh~)
```

  This is a bad result. Changing the dimension of the hidden layers the results became as follow.

1. *Batch size*: 50;
2. *Hidden size*: 128;
3. *Epochs*: 1000.

The final output of the network was:

```
Antony; but woe 'tis so!
ANTONY. I am dying, Egypt, dying; only
I here importune death awhile, until
Of many thousand kisses the poor last
I lay upon thy lips.
CLEOPATRA. I dare not, dear.
IRAMOD Angory.
Exeunt KING, LORDSCO. The duty that I owe unto your Majesty
I seal upon the lips of this sweet babe.
KING EDWARD. Thanks, noble Clank;
Whereto we are bound? Alack, or we must lose
The country, our dear nurse, or else thy person,
Our comf
```

The output has the right structure of language. By increasing the size of hidden layers the result becomes better.

1. *Batch size*: 50;
2. *Hidden size*: 256;
3. *Epochs*: 1000.

The final output of the network was:

```
ASSIO. The trust me; of the hath wren will bill,
The wren with smooth-PAUS. What, was thines bright was thith, and ay wear? BY
DOTA Dog. Be happy, he wert as we have the mountains.
APEMANTUS. Are the patter;
Save thou to FfTIOL Your retirement of England shall show the child, the report.
at which you prizize.
FRENCHMAN. I gead shall make ques the moon dost this bloody naw.
Or Irlands;
And yet his bow love been a
Doth worthless the leave me year some otR A. I am most and
```

The output is good, but there are some errors. By increasing the hidden size and the number of epochs the result becomes better.

1. *Batch size*: 50;
2. *Hidden size*: 512;

3. *Epochs*: 10000.

The final output of the network was:

```
ASTARD, REIGNIER, Bith me tour is my bld Mor he thangeful and plesttio, my dalu
and on bef prayst, for a son I bed,,kga bow way Uut thom your their tut might
foriong me aes hiead youmt of his arrson that the soul, and houlow and the
Jeearing feen or my pundice, am, mow younce alle mears as Wou wongorpr
everort, hath a yan birishes me wiener is my sersih my parne they I do note;
HARCOURT. Farl mistress.
CHAMBERLAIN. O East her'd you eade, bokd.
That de loughe whe womes; theet come the
```

If the dimension of the batch increases, the network looses the structure of the script of the opera. But the structure of English language becomes more accurate.

1. *Batch size*: 500;
2. *Hidden size*: 512;
3. *Epochs*: 10000.

The final output of the network was:

```
A, almonare have Foin after I'arena. it shing us daring,
The time you of us pres henken, merultince.
Whall wer wake, of me, whe than I with momes found; him,
such come be hold Coluineng.

Held enor
For come, us me fonry souroter him!, shall were d have a feath, theie that
with good Let cat an as it, timese and be man; a shall truchy Morth
that by lith me my swog bealel me to so;
Seod of me that sherion mado a with the Morny, mine bralating, arm,
Hife, cater
```

These test were useful to understand the structure and the behavior of recurrent neural networks.

Chapter 4

Neural Networks for the Knapsack Problem

4.1 Introduction

The goal of this chapter is to analyze and understand whether neural network can solve the Knapsack Problem with a sufficiently accuracy. It is clear that instances with few items are easy to solve, so in this case only instances with many objects were analyzed.

4.2 Dataset

To execute all tests, a dataset with many instances of KP was generated. First two datasets were created, one where all objects of every instance are sorted by decreasing profit-over-weight ratios, and another where objects are not ordered. The results of these datasets were the same, so only the dataset with sorted objects was used.
All data is split in three parts:

- Train set: 100000 instances;
- Validation set: 1000 instances;
- Test set, 5000 instances.

Every instance of KP has 200 items, and the capacity of the knapsack is in range [10000, 20000]. For each instance there are three values that represent the heuristic value of the total profit, the value of the optimal solution obtained using COMBO algorithm [6], and the last value is floor of the Dantzig's upper bound. Every value

of weight $w[j]$ for $j = 1, \ldots, N$ is in range $[\frac{capacity}{5}, \; capacity]$ and profit $p[j]$ for $j = 1, \ldots, N$ is equal to $w[j] \cdot (1.0 + 0.005 \cdot (random01() - 0.5))$ where $random01()$ is a random number in range $[0, \; 1]$.
Every instance stored as follows:

$N\ capacity\ w[1] \ldots w[N]\ p[1] \ldots p[N]\ card\ sol[1] \ldots sol[card]\ zheu\ zopt\ Dantzig\_UB$

where:

- N is the number of items of the instance;
- *capacity* is the capacity of the knapsack;
- $w[1] \ldots w[N]$ is the vector of weights;
- $p[1] \ldots p[N]$ is the vector of profits;
- *card* is the number of items selected in the optimal solution;
- $sol[1] \ldots sol[card]$ are the items selected in the optimal solution;
- *zheu* is the total profit of the greedy heuristic;
- *zopt* is the value of the optimal solution;
- *Dantzig_UB* is floor of the Dantzig's upper bound.

4.3 Prediction of multiple values

The first goal of this work was to train the network to predict three values for each instance: the heuristic value, the value of the optimal solution and the floor of the Dantzig's upper bound.
The network used to obtain the results has the following structure:

1. Input layer: Linear layer of dimension: 400×256;
2. Hidden layer: Linear layer of dimension: 256×128;
3. Output layer: Linear layer of dimension: 128×3.

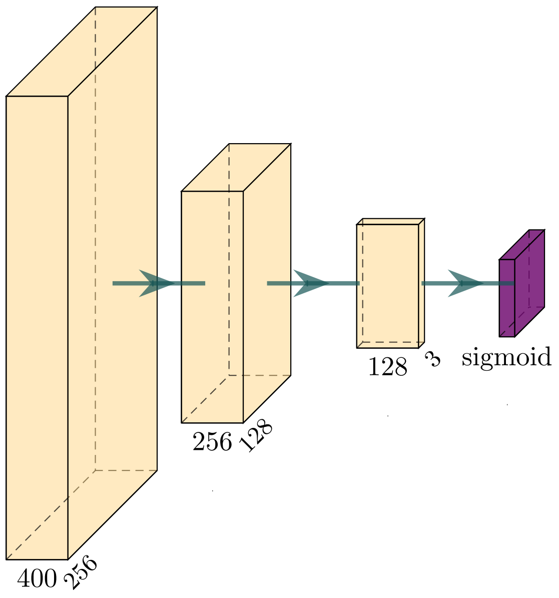

Figure 4.1. NN, first case

- Loss function: Mean Square Error;
- Learning rate: 0.005;
- Results:

	Heuristic value	Optimal value	Dantzig's upper bound
Underestimation error	-0.449 %	-9.80 %	-0.473 %
Overestimation error	0 %	+16.855 %	0 %
Error	+0.224 %	+13.327 %	+0.236 %

Table 4.1. Average NN errors, first case

Heuristic solution and Dantzig's upper bound are never overestimated, so the network predicts a value always lower than the value reported in each instance of dataset. To improve these results, another neural network was trained:

1. Input layer: Linear layer of dimension: 400×512;
2. Hidden layer: Linear layer of dimension: 512×256;
3. Hidden layer: Linear layer of dimension: 256×128;

4. Hidden layer: Linear layer of dimension: 128×64;
5. Hidden layer: Linear layer of dimension: 64×32;
6. Output layer: Linear layer of dimension: 32×3.

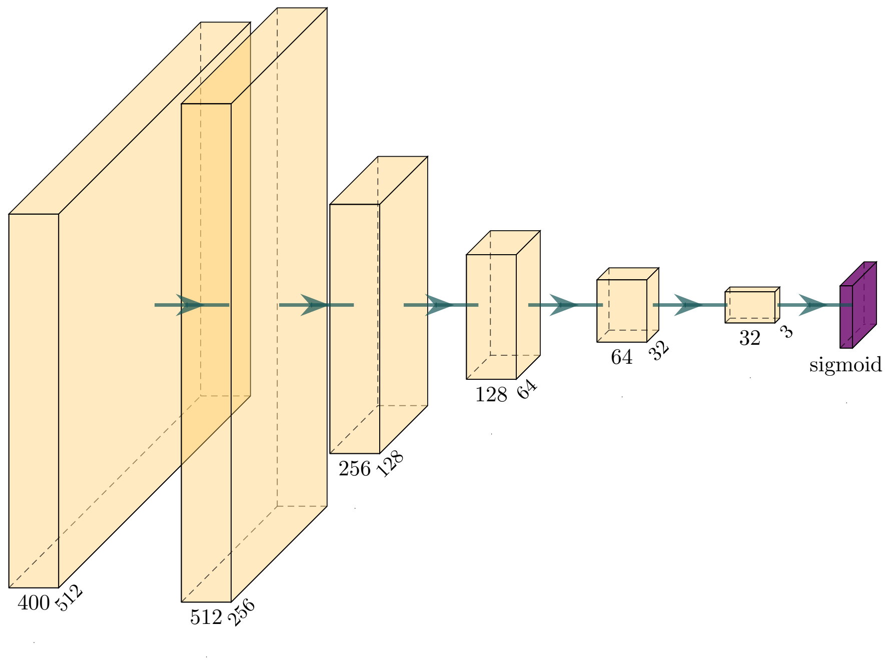

Figure 4.2. NN, second case

- Loss function: Mean Square Error;
- Learning rate: 0.005;
- Results:

	Heuristic value	Optimal value	Dantzig's upper bound
Underestimation error	-0.352 %	-9.52 %	-0.427 %
Overestimation error	+0.023 %	+17.197 %	0 %
Error	+0.187 %	+13.358 %	+0.236 %

Table 4.2. Average NN errors, second case

In this case, the network has a large error for the prediction of optimal solution. To improve these results, a recurrent neural network was used:

1. Input layer: Linear layer of dimension: 400×100;
2. Hidden layer: Linear layer of dimension: 100×32;
3. Hidden layer: Linear layer of dimension: 32×50;
4. Output layer: Linear layer of dimension: 50×3.

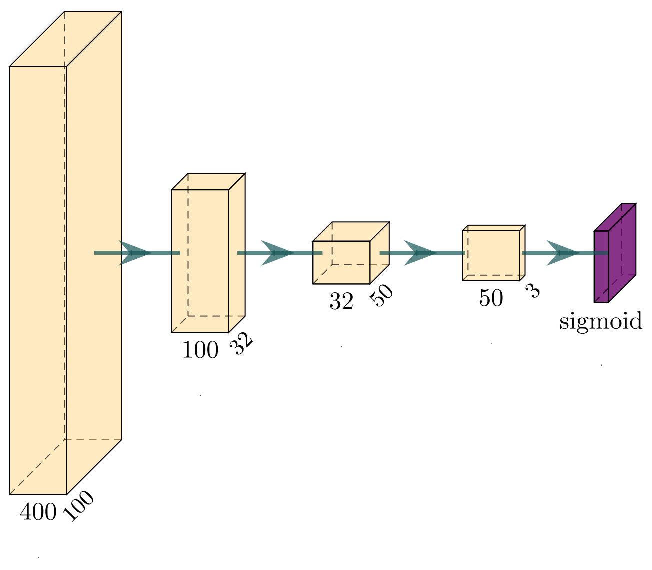

Figure 4.3. NN, third case

- Loss function: Mean Square Error;
- Learning rate: 0.005;
- Results:

	Heuristic value	Optimal value	Dantzig's upper bound
Underestimation error	-2.716 %	-7.060 %	-2.916 %
Overestimation error	0 %	+16.152 %	0 %
Error	+1.358 %	+11.606 %	+1.458 %

Table 4.3. Average RNN errors, third case

This network reduces the error of the optimal solution, but increases the errors of the other values.

4.4 Prediction of the optimal solution

Our second goal was to train the network to predict only the value of optimal solution for each instance.
We start to analyze in particular the network used to obtain the results of this test.

1. Input layer: Linear layer of dimension: 400×512;
2. Hidden layer: Linear layer of dimension: 512×256;
3. Hidden layer: Linear layer of dimension: 256×128;
4. Hidden layer: Linear layer of dimension: 128×64;
5. Hidden layer: Linear layer of dimension: 64×32;
6. Hidden layer: Linear layer of dimension: 32×16;
7. Hidden layer: Linear layer of dimension: 16×8;
8. Hidden layer: Linear layer of dimension: 8×4;
9. Hidden layer: Linear layer of dimension: 4×2;
10. Output layer: Linear layer of dimension: 2×1.

- Loss function: Mean Absolute Error or L1 Loss;
- Learning rate: 0.005;
- Results:

	Optimal value
Underestimation error	-4.665 %
Overestimation error	+26.833 %
Error	+1.486 %

Table 4.4. Average NN error, first case

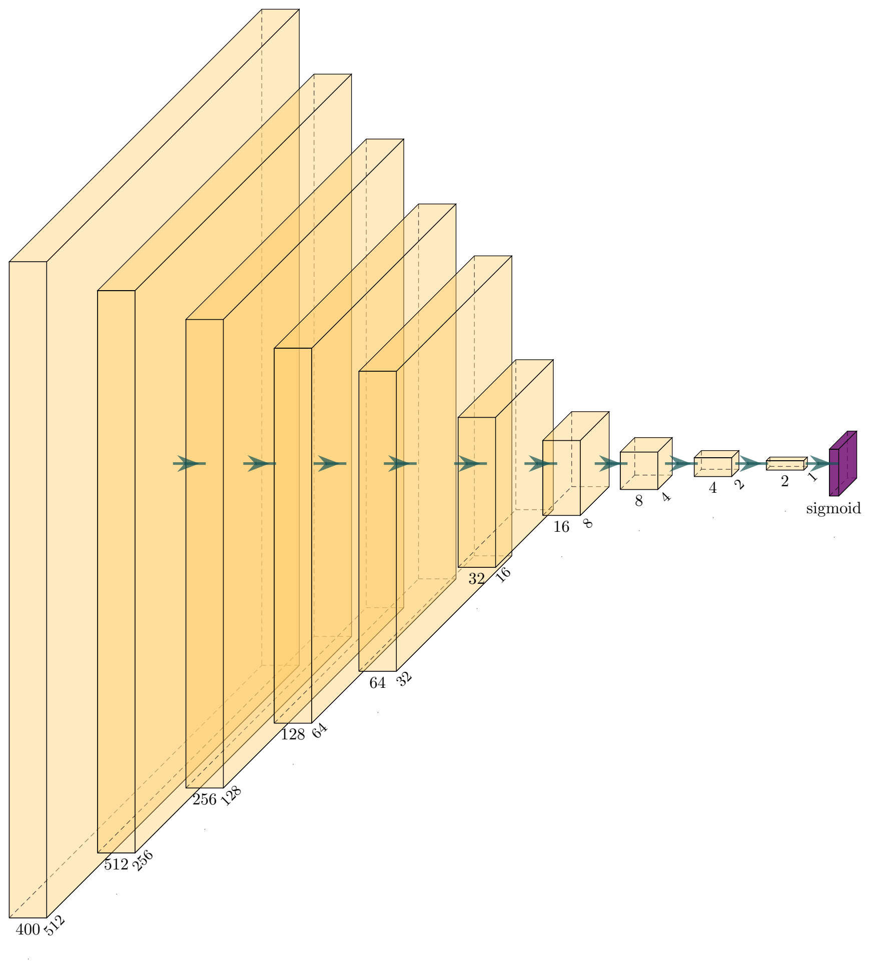

Figure 4.4. NN, first case

This network used 5 epochs to learn all the features to evaluate the value of the optimal solution.
In this case the solution obtained are overestimated compared to the real value of optimal solution. To reduce this percentage another structure of network was used:

1. Input layer: Linear layer of dimension: 400×8192;
2. Hidden layer: Linear layer of dimension: 8192×4096;
3. Hidden layer: Linear layer of dimension: 4096×2048;

4. Hidden layer: Linear layer of dimension: 2048×1024;
5. Hidden layer: Linear layer of dimension: 1024×512;
6. Hidden layer: Linear layer of dimension: 512×256;
7. Hidden layer: Linear layer of dimension: 256×128;
8. Hidden layer: Linear layer of dimension: 128×64;
9. Hidden layer: Linear layer of dimension: 64×32;
10. Hidden layer: Linear layer of dimension: 32×16;
11. Hidden layer: Linear layer of dimension: 16×8;
12. Hidden layer: Linear layer of dimension: 8×4;
13. Hidden layer: Linear layer of dimension: 4×2;
14. Output layer: Linear layer of dimension: 2×1.

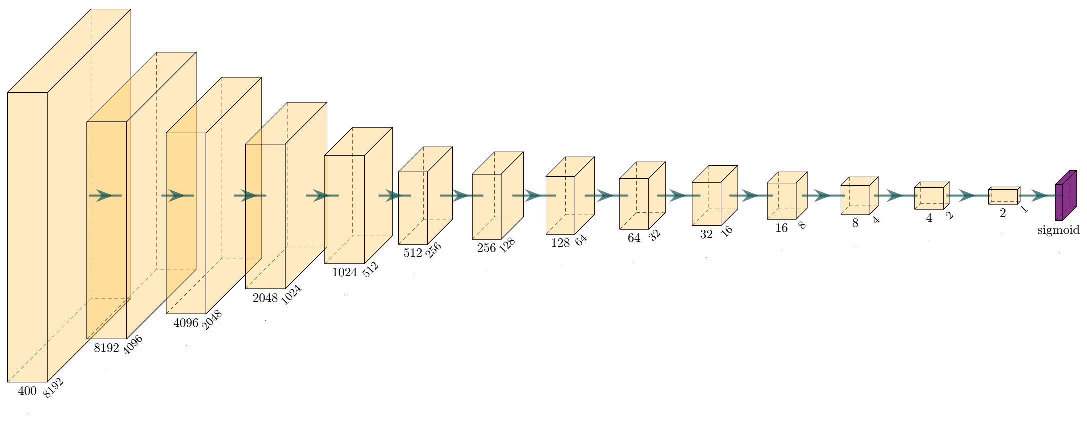

Figure 4.5. NN, second case

- Loss function: Mean Absolute Error or L1 Loss;
- Learning rate: 0.005;
- Results:

	Optimal value
Underestimation error	-8.212 %
Overestimation error	+43.385 %
Error	+0.378 %

Table 4.5. Average NN error, second case

This network used 5 epochs to learn all the features to evaluate the value of the optimal solution.
The performance of the network decreases using this type of network. Too many hidden layers led to a large overfitting. To improve the behavior of the net a different structure was used.

1. Input layer: Linear layer of dimension: 400×512;
2. Hidden layer: Linear layer of dimension: 512×256;
3. Hidden layer: Linear layer of dimension: 256×128;
4. Hidden layer: Linear layer of dimension: 128×64;
5. Hidden layer: Linear layer of dimension: 64×32;
6. Hidden layer: Linear layer of dimension: 32×16;
7. Hidden layer: Linear layer of dimension: 16×8;
8. Hidden layer: Linear layer of dimension: 8×4;
9. Hidden layer: Linear layer of dimension: 4×2;
10. Output layer: Linear layer of dimension: 2×1.

- Loss function: Mean Absolute Error or L1 Loss;
- Learning rate: 0.005;
- Results:

	Optimal value
Underestimation error	-0.069 %
Overestimation error	+0.114 %
Error	+0.022 %

Table 4.6. Average NN error, third case

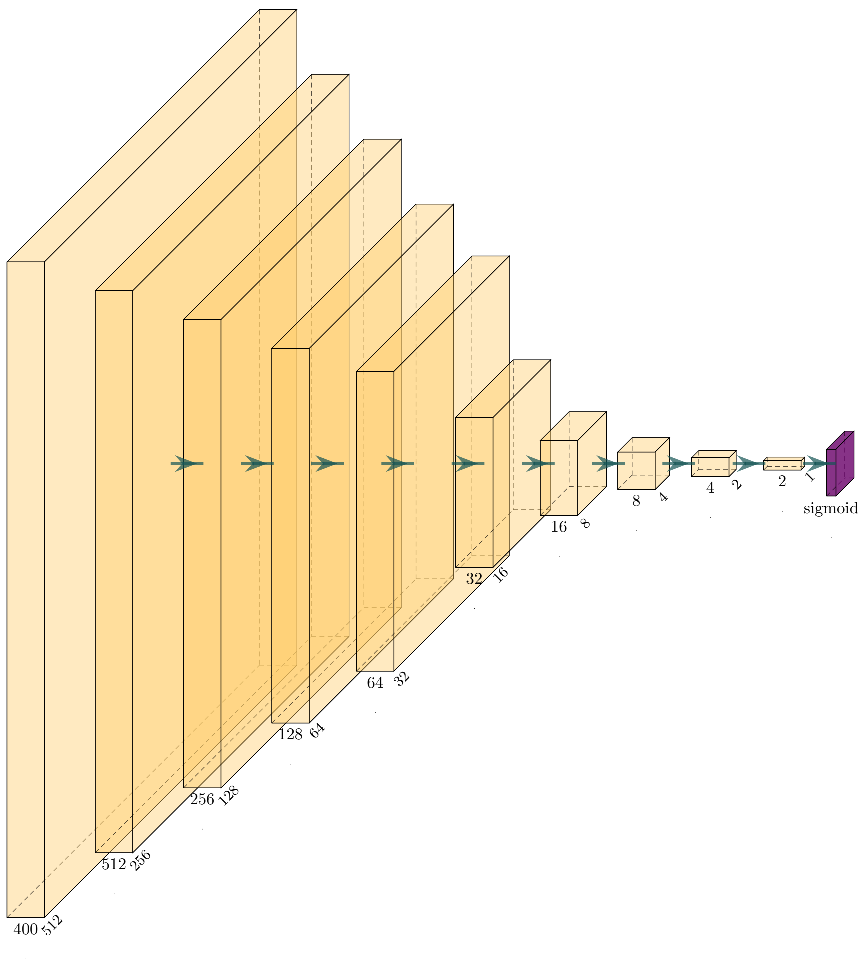

Figure 4.6. NN, third case

This network used only one epoch to learn all the features to evaluate the value of the optimal solution. This value is low, but it can avoid the overfitting.
The performance are very good and this network will used for the heuristic algorithms developed in the next chapter.

Chapter 5

Heuristic Algorithms

As seen before, neural networks can predict the value of the optimal solution with a small error. In this chapter an heuristic approach will be described, that uses the neural network to compute the set of items in the final solution.
Our goal is to find the items that satisfy the weight capacity of the knapsack and maximize the profit of solution.

5.1 The idea

To compute the final solution, the last neural network in Section 4.4 is used. The goal is to evaluate heuristic algorithms that use the neural network to choose the items, together with the greedy algorithm described in Section 1.4, and to compared then with the solver COMBO [6].
The dataset used to re-train the net has all objects of every instance sorted by decreasing profit-over-weight ratios.
All data is split in three parts:

- Train set: 100001 instances;
- Validation set: 1001 instances;
- Test set, 5005 instances.

There are instances with number of items in range [190, 200], and the capacity of the knapsack was in range [10000, 20000], i.e., multiplied by a factor in range [0.1, 1] w.r.t. the previous experiments. For each instance there are three values that represent the heuristic value of the total profit, the value of the optimal solution obtained using COMBO algorithm [6], and the floor of the Dantzig's upper bound. Weights $w[j]$ are in range $[\frac{capacity}{5},\ capacity]$, and profits $p[j]$ are equal to $w[j] \cdot (1.0+$

$0.005 \cdot (random01() - 0.5))$, where $random01()$ is a random number in range $[0,\ 1]$. Thus every instance is stored as follows:

$N\ capacity\ w[1]\ \ldots\ w[N]\ p[1]\ \ldots\ p[N]\ card\ sol[1]\ \ldots\ sol[card]\ zheu\ zopt\ Dantzig\_UB$

where:

- N is the number of items of the instance;
- *capacity* is the capacity of the knapsack;
- $w[1]\ \ldots\ w[N]$ is the vector of weights;
- $p[1]\ \ldots\ p[N]$ is the vector of profits;
- *card* is the number of items selected in the optimal solution;
- $sol[1]\ \ldots\ sol[card]$ are the selected items in the optimal solution;
- *zheu* is the greedy heuristic value;
- *zopt* is the value of the optimal solution;
- $Dantzig\_UB$ is floor of the Dantzig's upper bound.

The structure of the net, described previously, leads to the following results:

		Optimal value	Number of instances
Underestimation error	From	-20.221 %	4993
	To	-0.014 %	
Overestimation error	From	+0.023 %	3
	To	+0.087 %	
Same solution			9
Error		+12.691 %	

Table 5.1. Heuristic error, optimal solution

In Table 5.1 the error is referred to the value of the optimal solution. This value is obtained from the computation of the total profit from the items chosen from the network.
Another attempt was made to train the neural network. Previously, the network computed only the value of the optimal solution, in this case the network computes for each item j, the probability that the object j is in the optimal solution.

This network has this structure:

1. Input layer: Linear layer of dimension: 400×1024;
2. Hidden layer: Linear layer of dimension: 1024×950;
3. Hidden layer: Linear layer of dimension: 950×900;
4. Hidden layer: Linear layer of dimension: 900×850;
5. Hidden layer: Linear layer of dimension: 850×800;
6. Hidden layer: Linear layer of dimension: 800×750;
7. Hidden layer: Linear layer of dimension: 750×700;
8. Hidden layer: Linear layer of dimension: 700×650;
9. Hidden layer: Linear layer of dimension: 650×600;
10. Hidden layer: Linear layer of dimension: 600×550;
11. Hidden layer: Dropout layer with probability 60 %;
12. Hidden layer: Linear layer of dimension: 550×500;
13. Hidden layer: Linear layer of dimension: 500×450;
14. Hidden layer: Linear layer of dimension: 450×400;
15. Hidden layer: Linear layer of dimension: 400×375;
16. Hidden layer: Linear layer of dimension: 375×350;
17. Hidden layer: Linear layer of dimension: 350×325;
18. Hidden layer: Linear layer of dimension: 325×300;
19. Hidden layer: Linear layer of dimension: 300×275;
20. Hidden layer: Linear layer of dimension: 275×250;
21. Hidden layer: Linear layer of dimension: 250×225;
22. Hidden layer: Linear layer of dimension: 225×200;
23. Output layer: Linear layer of dimension: 200×200.

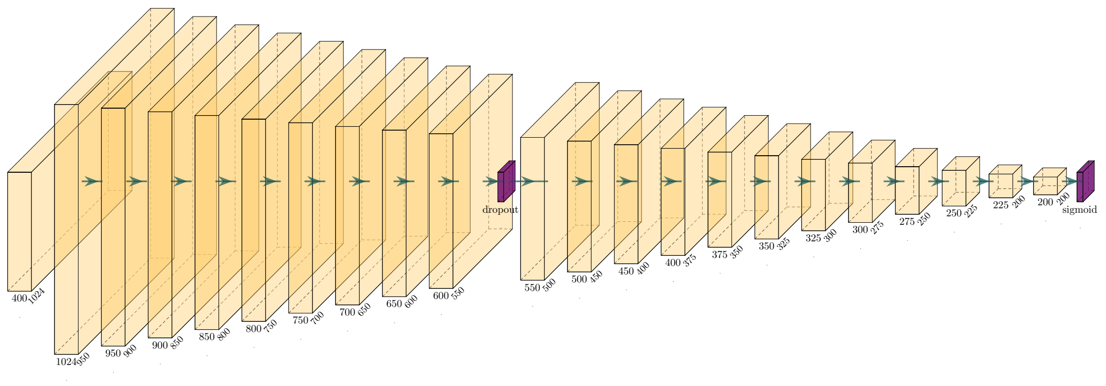

Figure 5.1. NN to compute the probabilities of items

- Loss function: Mean Square Error;
- Learning rate: 0.001;

The network was trained for 5 epochs. Let $NN(solution,\ i)$ be the estimated optimal value of the solution composed by the items stored in the vector *solution*, and the i-th element of the instance, computed by the neural network. The algorithms used for the comparisons with the greedy algorithm are:

1. Basic heuristic 5.1: for each item the algorithm makes a choice, selecting the current object or not. This step calls the NN trained on the partial instance (for item j, the instance evaluated from the net is the instance with objects $1,\ \ldots,\ j$ for $j = 1,\ \ldots,\ N$) where the weights of the objects are scaled compared to the partial capacity of the instance. If the value of partial solution with object j, computed by the NN is greater than the value of the partial solution without object j, the item is chosen, otherwise the object is discarded. This process continues until the end of all items, or if the knapsack is filled up.

Algorithm 5.1 Basic heuristic

```
1: $tWeight = 0$
2: $tProfit = 0$
3: $solution = [\ ]$
4: for $i := 0$ to $N - 1$ do
5:     if $(tWeight + w[i] \leq W)$ and $(NN(solution, i) > tProfit)$ then
6:         $solution.Append(i)$
7:         $tWeight = tWeight + w[i]$
8:         $tProfit = tProfit + p[i]$
```

In this case, the algorithm calls the neural network on the chosen items in the first i object. At the first iteration, the value of $tProfit$ is equal to 0, so the value of $NN(solution, i)$ is, in this case, always greater then $tProfit$ and the first solution is composed only by the first item. Only after several time the solution could be changed.

2. Heuristic with best incumbent 5.2: uses the same process as in the basic heuristic, with the difference that the algorithm computes the best incumbent solution for all sets of items chosen from the net, and updates the final solution only if the value is larger than the best incumbent computed until this point.

3. Heuristic with best incumbent with random items 5.3: uses the same process as in the heuristic with best incumbent, but the items are chosen randomly, i.e., it does not uses the net to choose the objects.

4. Heuristic with best incumbent with probability items 5.4: use the same process as in the heuristic with best incumbent, but the items are chosen using the NN that computes the probability that each object is in the optimal solution.

In this case, the algorithm call the neural network to have the probability of choosing the i-th object, and this value is $NNProb(i)$.

Algorithm 5.2 Heuristic with best incumbent

```
1: $tWeight_w = 0$
2: $tProfit_w = 0$
3: $tWeight_{wo} = 0$
4: $tProfit_{wo} = 0$
5: $solution_w = [\,]$
6: $solution_{wo} = [\,]$
7: $bestSolution = [\,]$
8: $bestIncumbent = 0$
9: for $j := 0$ to $N - 1$ do
10:     $solution_w = bestIncumbent[0 : j]$
11:     if $j > 0$ then
12:         $solution_{wo} = bestIncumbent[0 : j - 1]$
13:     for $i := j + 1$ to $N - 1$ do
14:         if $(tWeight_w + w[i] \leq W)$ and $(NN(solution_w, i) > tProfit_w)$ then
15:             $solution_w.Append(i)$
16:             $tWeight_w = tWeight_w + w[i]$
17:             $tProfit_w = tProfit_w + p[i]$
18:         if $(tWeight_{wo} + w[i] \leq W)$ and $(NN(solution_{wo}, i) > tProfit_{wo})$ then
19:             $solution_{wo}.Append(i)$
20:             $tWeight_{wo} = tWeight_{wo} + w[i]$
21:             $tProfit_{wo} = tProfit_{wo} + p[i]$
22:     if $tProfit_w > tProfit_{wo}$ then
23:         $tProfit = tProfit_w$
24:         $solution = solution_w$
25:     else
26:         $tProfit = tProfit_{wo}$
27:         $solution = solution_{wo}$
28:     if $tProfit > bestIncumbent$ then
29:         $bestIncumbent = tProfit$
30:         $bestSolution = solution$
```

Algorithm 5.3 Heuristic with best incumbent with random items

```
1: $tWeight_w = 0$
2: $tProfit_w = 0$
3: $tWeight_{wo} = 0$
4: $tProfit_{wo} = 0$
5: $solution_w = [\,]$
6: $solution_{wo} = [\,]$
7: $bestSolution = [\,]$
8: $bestIncumbent = 0$
9: for $j := 0$ to $N - 1$ do
10:     $solution_w = bestIncumbent[0 : j]$
11:     if $j > 0$ then
12:         $solution_{wo} = bestIncumbent[0 : j - 1]$
13:     for $i := j + 1$ to $N - 1$ do
14:         if $random() > 0.5$ then
15:             if $(tWeight_w + w[i] \leq W)$ and $(tProfit_w + p[i] > tProfit_w)$ then
16:                 $solution_w.Append(i)$
17:                 $tWeight_w = tWeight_w + w[i]$
18:                 $tProfit_w = tProfit_w + p[i]$
19:             if $(tWeight_{wo} + w[i] \leq W)$ and $(tProfit_{wo} + p[i] > tProfit_{wo})$ then
20:                 $solution_{wo}.Append(i)$
21:                 $tWeight_{wo} = tWeight_{wo} + w[i]$
22:                 $tProfit_{wo} = tProfit_{wo} + p[i]$
23:     if $tProfit_w > tProfit_{wo}$ then
24:         $tProfit = tProfit_w$
25:         $solution = solution_w$
26:     else
27:         $tProfit = tProfit_{wo}$
28:         $solution = solution_{wo}$
29:     if $tProfit > bestIncumbent$ then
30:         $bestIncumbent = tProfit$
31:         $bestSolution = solution$
```

Algorithm 5.4 Heuristic with best incumbent with probability items

```
1: $tWeight_w = 0$
2: $tProfit_w = 0$
3: $tWeight_wo = 0$
4: $tProfit_wo = 0$
5: $solution_w = [\ ]$
6: $solution_wo = [\ ]$
7: $bestSolution = [\ ]$
8: $bestIncumbent = 0$
9: for $j := 0$ to $N - 1$ do
10:     $solution_w = bestIncumbent[0 : j]$
11:     if $j > 0$ then
12:         $solution_wo = bestIncumbent[0 : j - 1]$
13:     for $i := j + 1$ to $N - 1$ do
14:         if $NNProb(i) > 0.5$ then
15:             if $(tWeight_w + w[i] \leq W)$ and $(tProfit_w + p[i] > tProfit_w)$ then
16:                 $solution_w.Append(i)$
17:                 $tWeight_w = tWeight_w + w[i]$
18:                 $tProfit_w = tProfit_w + p[i]$
19:             if $(tWeight_wo + w[i] \leq W)$ and $(tProfit_wo + p[i] > tProfit_wo)$ then
20:                 $solution_wo.Append(i)$
21:                 $tWeight_wo = tWeight_wo + w[i]$
22:                 $tProfit_wo = tProfit_wo + p[i]$
23:     if $tProfit_w > tProfit_wo$ then
24:         $tProfit = tProfit_w$
25:         $solution = solution_w$
26:     else
27:         $tProfit = tProfit_wo$
28:         $solution = solution_wo$
29:     if $tProfit > bestIncumbent$ then
30:         $bestIncumbent = tProfit$
31:         $bestSolution = solution$
```

5.2 Optimal solution prediction

The results of these algorithms compared to the greedy algorithm are reported in Table 5.2.

Basic heuristic		
Maximum underestimation error	-21.345 %	
Maximum overestimation error	+5.131 %	
Average underestimation error	-5.481 %	77 instances
Average overestimation error	+1.869 %	115 instances
Same solution (greedy)	4813 instances	
Heuristic with best incumbent		
Maximum underestimation error	-0.222 %	
Maximum overestimation error	+14.004 %	
Average underestimation error	-0.137 %	10 instances
Average overestimation error	+8.952 %	4878 instances
Same solution (greedy)	117 instances	
Heuristic with best incumbent with random items		
Maximum underestimation error	-0.731 %	
Maximum overestimation error	+6.943 %	
Average underestimation error	-0.192 %	124 instances
Average overestimation error	+8.957 %	4828 instances
Same solution (greedy)	53 instances	
Heuristic with best incumbent with probability items		
Maximum underestimation error	-19.445 %	
Maximum overestimation error	+4.252 %	
Average underestimation error	-9.197 %	3576 instances
Average overestimation error	+5.687 %	1425 instances
Same solution (greedy)	4 instances	

Table 5.2. Results of algorithms compared to the greedy algorithm

From Table 5.2, one can note that the heuristic algorithm with best incumbent has the best performance. The algorithm with random items has the same percentage difference with the greedy algorithm, but the number of instances with random items is slightly different. In fact, the number of instances that are underestimated, is larger than in the algorithm with the random items.
The results of these algorithms compared to the solver COMBO [6] are reported in Table 5.3.

Basic heuristic		
Maximum underestimation error	-21.880 %	
Maximum overestimation error	0 %	
Average underestimation error	-7.888 %	4957 instances
Average overestimation error	0 %	0 instances
Same solution (COMBO)	48 instances	
Heuristic with best incumbent		
Maximum underestimation error	-0.687 %	
Maximum overestimation error	0 %	
Average underestimation error	-0.152 %	4404 instances
Average overestimation error	0 %	0 instances
Same solution (COMBO)	601 instances	
Heuristic with best incumbent with random items		
Maximum underestimation error	-1.030 %	
Maximum overestimation error	0 %	
Average underestimation error	-0.229 %	4718 instances
Average overestimation error	0 %	0 instances
Same solution (COMBO)	287 instances	
Heuristic with best incumbent with probability items		
Maximum underestimation error	-19.823 %	
Maximum overestimation error	0 %	
Average underestimation error	-11.532 %	4990 instances
Average overestimation error	0 %	0 instances
Same solution (COMBO)	15 instances	

Table 5.3. Results of algorithms compared to the solver COMBO [6]

From Table 5.3, one can see that the heuristic algorithm with best incumbent has the best performances compared to the other algorithms. This case has the highest number of instances that give the same value of the solver COMBO, and the average underestimation error is the lowest.

Chapter 6

Conclusions

The goal of this work was to train a neural network and use it to obtain a better solution compared to the solution given by the greedy algorithm. Different approaches were developed to define the final solution because it is not clear how to train the net and which is the best structure of the NN to have the best KP solution.
From Table 5.2 and 5.3 it is possible to see how the algorithms, and the different networks used, work. From the first table, that compared our results with those obtained with the greedy algorithm, the algorithms that use the incumbent solution have the best performance. The gain in this case is, on average, about 8 % compared to the solution given by the greedy algorithm. This is a good result, but it is mainly due to the structure of the algorithm, and not from the use of neural networks. In fact, the algorithm that uses the neural network to choose the items and the one using a random value to choose the items, have the same percentage gain. The difference between these approaches are only in the number of instances in which the gain is obtained. These values are similar, but the usage of neural networks gives some improvement of the performance.
In the second case, when the solutions of the algorithms are compared to the exact solver COMBO, the situation is similar. There is no gain also in this case, it is possible to consider only the number of solutions that has the same value of COMBO. This number is greater for the approach that uses the neural network: about 3 times compared to the algorithm that uses random values to choose the objects.
The results are interesting if the algorithm to compare is the greedy algorithm. In this case, the NN can improve the number of better solutions, and these solutions are improved by about 8 %. When the comparison is made with COMBO, the performance are less satisfactory because the network was trained to improve the greedy solutions and not to have the same solution of COMBO. This is however not surprising as COMBO is an exact solution method.
To improve these results, we can set a different structure of layers in the neural network that compute the probability to choose each item in the final solution. A

possible way is to use a recurrent neural network to predict the probabilities: the net in this case could have more accuracy to decide if the probability is closer to zero or if it is closer to one.
In the other case, the network has a small error when it compute the optimal value, so improving that net may not be very useful.

Bibliography

[1] M. Fischetti, *Lezioni di Ricerca Operativa*, 2018.
[2] F. Chollet, *Deep Learning with Python*, Manning Publications Co., 2018.
[3] H. Kellerer, U. Pferschy, D. Pisinger, *Knapsack Problems*, Springer-Verlag Berlin Heidelberg, 2004.
[4] S. Shalev-Shwartz, S. Ben-David, *Understanding Machine Learning From Theory to Algorithms*, Cambridge University Press, 2014.
[5] S. Hochreiter, J. Schmidhuber, *Long Short-Term Memory*, Neural Computation 9, no. 8, 1997.
[6] S. Martello, D. Pisinger, P. Toth, *Dynamic Programming and Strong Bounds for the 0-1 Knapsack Problem*, Management Science, 1997.
[7] I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning*, MIT Press, 2016.
[8] Common loss functions in machine learning, `https://towardsdatascience.com/common-loss-functions-in-machine-learning-46af0ffc4d23`, date of consultation: 2019-07-07.
[9] Pytorch, `https://pytorch.org/`, date of consultation: 2019-07-07.